

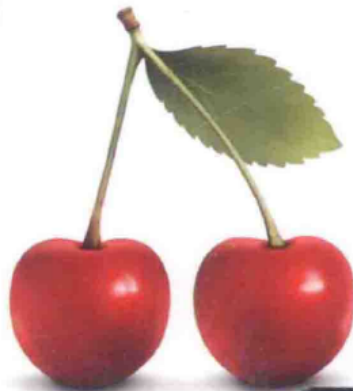
Node.js 开发实战详解

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

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
客，还介绍了5个大型系统的开发，并赠送8小时教学视频（需下载）



清华大学出版社

Node.js 开发实战详解



Web开发典藏大系



ISBN 978-7-302-28865-7



ISBN 978-7-302-31728-9



ISBN 978-7-302-28867-1



ISBN 978-7-302-28100-9



ISBN 978-7-302-32540-6



ISBN 978-7-302-32702-8



ISBN 978-7-302-31735-7



ISBN 978-7-302-33804-8



ISBN 978-7-302-34947-1



2014年6月上市

上架：计算机/JavaScript

清华大学出版社数字出版网站

WQBook 书网

www.wqbook.com



9 787302 349471 >

定价：59.80元

Web开发典藏大系

web design

server admin

consulting

marketing

mobile apps

domains

hosting

outsourcing

Q

Q



COMPANY INFORMATION
lorem ipsum dolor sit amet



SERVICES & SOLUTIONS
lorem ipsum dolor sit amet



DAILY NEWSLETTER
lorem ipsum dolor sit amet



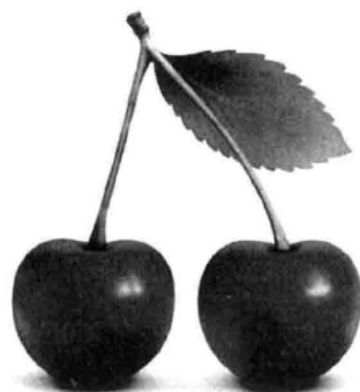
WORLDWIDE PARTNERS
lorem ipsum dolor sit amet



CUSTOMER SUPPORT
lorem ipsum dolor sit amet

Node.js 开发实战详解

黄丹华 等编著



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: 李红英

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: 北京鑫丰华彩印有限公司

装 订 者: 三河市新茂装订有限公司

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: 1~4000

定 价: 59.80 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	MongoDB 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 1 章 Node.js 基础知识

本章主要介绍 Node.js 的概念，以及 Node.js 的出现给我们程序员带来了什么方便、解决了什么问题、在项目开发中是否应该选择使用 Node.js。同时介绍 Node.js 的配置、开发，以及 Node.js 的入门程序 Hello world。最后介绍 Node.js 的编程思想、如何得到一个异步结果、区分同步和异步的编程思想的不同之处。

1.1 概 述

本节主要介绍 Node.js 基本概念，包括什么是 Node.js、它与其他编程语言之间的区别、它的特点和优势在哪里，以及项目开发过程中如何抉择是否使用 Node.js 开发应用。

1.1.1 Node.js 是什么

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。Node.js 对一些特殊用例进行了优化，提供了替代的 API，使得 V8 在非浏览器环境下运行得更好。其目的是可以在服务器端执行和运行 JavaScript 代码。长久以来 JavaScript 都是一个基于浏览器的客户端脚本语言，通过将其运行环境抽离出来，就可以在服务器端运行 JavaScript 代码，而并非仅仅依赖浏览器解析，从而就可以将其作为服务器端语言，由于其拥有异步非阻塞特性，因此其在长连接、多请求的环境下优势非常明显。

Node.js 的编程语言是基于 JavaScript 的语法，因此想深入学习 Node.js 的入门者，可以先熟悉 JavaScript 编程语言，了解其基本的语法（本书不会介绍 JavaScript 的语法），同时要对服务器端开发有所了解。当然，如果你是初学者也可以通过本书学习 Node.js 的编程，但需要你对知识了解更深入一些。Node.js 提供了一些特殊的 API（Node.js 官网提供了详细的 API 说明）库，因此在编写 Node.js 的时候可以理解为，使用 JavaScript 语言，利用 Node.js 的 API 库进行服务器端开发。

1.1.2 Node.js 带来了什么

初学者经常会提及一个问题——Node.js 到底有什么用？这个问题也经常令我苦恼，是不是有了 Node.js 我们就可以抛弃 PHP 或者其他服务器语言了呢？答案是否定的。

首先要理解 Node.js 的优点，以及 Node.js 与其他语言之间的区别和联系。传统的服务器语言，如 PHP 和 Java 等，每个 Web HTTP 请求连接都会产生一个线程，假设每个线程

需要 2MB 的配置内存，因此相对一个 8GB 的服务器主机，也只能承受来自 4000 个并发用户的请求，当服务器承受不了这么多用户的情况下就需要添加服务器，从而导致增加项目运营成本（当然现在有 Nginx 支撑，可以提供更高的并发量请求）。

其次，理解 Node.js 解决的问题是什么？Node.js 解决多请求的方法，在于其处理连接服务器的方式。在 Node.js 中每个 HTTP 连接都会发射一个在 Node.js 引擎的进程中运行的事件，而不是为每个连接生成一个新的 OS 进程（并为其分配一些配套内容）。

综上所述就可以清晰地看出 Node.js 相对 PHP 来说其优点在于能处理高并发请求，并且由于 Node.js 是事件驱动，因此可以更好地节约服务器内存资源。

在项目的开发中，我们应该如何抉择是否使用 Node.js 作为项目开发实现呢？首先，必须要了解项目的类型是不是适合 Node.js 去开发项目。例如需要开发一个博客、论坛或者微博，那么是否能使用 Node.js 去开发？回答是肯定的，但是不合适，相对来说 PHP 在这方面已经很成熟。其次，要理解一个事实，在本书编写过程中 Node.js 版本还不是很稳定，在 Node.js 版本升级的过程中，可能引发一些项目代码中的问题，例如 Node.js 官方提供的 API 有所改动时，以及当项目 Node.js 版本升级时，则必须修改代码。

Node.js 可以单独实现一个 server，这也是 Node.js 一个非常大的优点，对于那些简单的 server，通过 Node.js 实现比使用 C++ 实现会简单得多。最后，牢记 Node.js 解决了长连接、多请求引发的成本问题，因此在一些项目，例如实时在线 Game（如一起来画画、黑暗杀人游戏、实时休闲游戏等）、实时在线聊天室、实时消息推送功能、SNS 实时交流、实时监控系統（如股票、系统运行状态等）等开发过程中，应该把握住机会，应用 Node.js 来开发。那么在如此好的机会面前，应该如何去说服团队和老板呢？

在 CSDN 社区中有一篇博文《程序员如何说服老板采用 Node.js》¹，详细说明了一个项目是否适合用 Node.js 开发，当适合时应该如何用事实说服老板、团队和客户。总结其过程包括如下几个阶段，如图 1-1 所示。

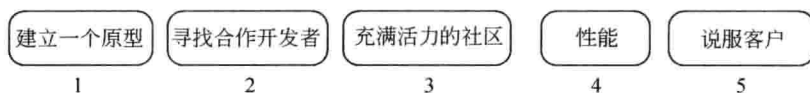


图 1-1 说服过程

- ❑ 建立原型：对于老板来说他希望看到的是结果，不管你使用什么开发，他的最终目的是节省资源、快速开发、满足客户。因此首先需要给老板看到一个系统原型，能够让老板明白你的系统开发是有条理的，不是简简单单的一个想法。
- ❑ 寻找合作开发者：项目开发是一个团队的事情，我们依靠的是团队力量，因此想成功开发一个 Node.js 应用，必须要说服团队的成员和你共同开发。有团队的依靠才能更进一步地让老板放心。
- ❑ 充满活力的社区：此刻，Node.js 社区用户数正以一个疯狂的速度增长，该社区吸引了众多顶尖开发者。也就是说，Node.js 生态系统每天都在完善，并且通过不同渠道获得了各个企业的免费支持。你需要告诉老板的是，现在这门新技术正在蓬勃的发展，并且也会是未来 Web 应用的一门比较重要的语言。


1 CSDN 社区博文网站 <http://www.csdn.net/article/2012-05-03/2805296>。

- ❑ 性能：在合适的项目中，更要突出 Node.js 开发项目的优点，以及可行性。可以在一个项目的会议上，通过你的讲解更进一步地说服老板，让老板认为你的想法是正确并且可靠的。
- ❑ 说服客户：最后你需要做的就是说服客户，需要让他们知道的是，Node.js 的开发更能够满足他们的需求，能够更快地开发出他们需要的产品。

把握以上几点以后，还需要我们加强个人的 Node.js 知识，多了解一些服务器端的知识扩展自己的知识层面。可以说 Node.js 给我们带来的是一个选择，在项目开发中遇到问题时，我们可以考虑使用 Node.js 来解决问题。对于程序员来说，给我们带来的的是一个机遇和挑战，能够让我们成为小组 Node.js 项目开发中不可或缺的成员。

1.2 Node.js 配置开发

本节将会介绍如何搭建 Node.js 的运行环境，包括 Linux 系统和 Windows 系统环境，同时学习一门语言的第一个程序“Hello World”。学完本节，需要掌握 Linux 系统和 Windows 系统下的 Node.js 环境搭建，以及编写简单的 Node.js 程序、运行和调式方法。

 **注意：**在学习本节时，需要注意在本书编写过程中安装配置可能会和读者阅读本书时的版本安装存在差异，这些知识笔者后续会通过论坛告知大家，但是大致的安装和配置过程是类似的。

1.2.1 Windows 配置

Node.js 的 Windows 运行环境安装步骤简单，只需要注意将执行文件 `node.exe` 路径添加到环境变量。安装主要分为 4 个步骤，其中安装步骤 1 中建议下载 `msi`¹ 的后缀执行文件，主要是便于后续的项目开发。安装步骤 4 中介绍如何将 `node.exe` 添加到全局执行环境中，本步骤没有介绍如何在 Window 中添加环境变量²，这些基本知识在这里就不详细介绍了。主要安装过程如下所述。

(1) 官网 (<http://nodejs.org>) 下载 Node.js 的 Windows 系统 (32 位和 64 位) 最新版本，建议下载 `msi` 为后缀的版本，如图 1-2 中方框内所示的版本。`exe` 为执行文件，但其缺少一些 Node.js 的模块和 `npm` 等。

(2) 下载完成后，执行 MSI 的安装文件。

(3) 安装完成，查看 Node.js 启动文件目录 (右键单击 Node.js 启动快捷方式)，如图 1-3 所示，启动文件目录一般默认情况下是在“`C:\Program Files\nodejs\node.exe`”。

(4) 将 `node.exe` 可执行文件路径添加到 Windows 的环境变量中；运行 `cmd`，进入 `dos`

1 MSI 文件是 Windows Installer 的数据包，它实际上是一个数据库，包含安装一种产品所需要的信息和在很多安装情形下安装 (和卸载) 程序所需的指令和数据。

2 环境变量设置方法参考：<http://www.cnblogs.com/307914070/archive/2011/02/11/1951725.html>。

操作命令窗口，输入 `node-v` 查看是否安装成功，如图 1-4 所示，表明已经成功安装 Node.js。



图 1-2 Node.js 官网下载页面

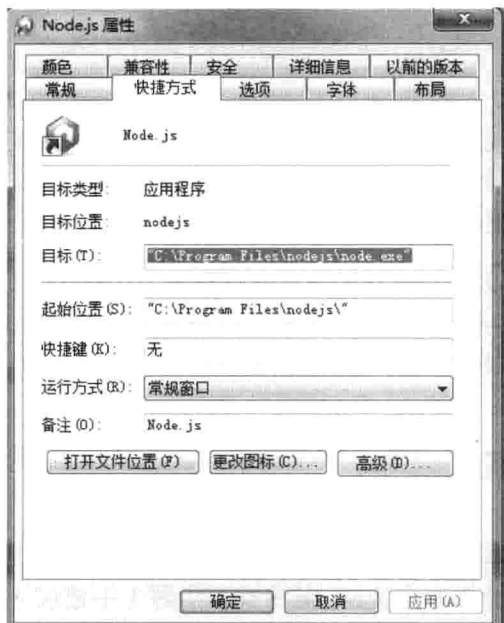


图 1-3 Node.js 右键快捷图标属性

注意：Windows 添加环境变量的方法：右键计算机桌面图片，打开右侧导航栏中的“高级系统设置”，打开以后选择其中的“环境变量”，之后再更改用户的 PATH 值，在其原有变量值上加上 Node.js 可执行文件路径，两者使用分号分割，例如：
`C:\Users\danhuang\AppData\Roaming\npm\;C:\Program Files\nodejs\node.exe`。

```
C:\Users\danhuang>node --version
v0.8.21
```

图 1-4 Node.js 查看版本信息

成功安装后可以看到当前的 Node.js 的版本信息，如图 1-4 所示。如果没有成功添加到系统的环境变量中，cmd 运行窗口会提示没有 `node` 指令，这时候应该检查是否已经将 Node.js 的可执行路径成功添加到环境变量中。

1.2.2 Linux 配置

Linux 下的 Node.js 环境部署方法较多,本书就一种方法进行讲解。官网现在提供了两种 Linux 包,一个是源码安装包,另外一个已经是编译好的可执行文件,本书介绍 Node.js 源码的编译安装方法。相对 Windows 下的环境部署, Linux 较为简单,里面只涉及一些 Linux 的解压和编译,以及软件安装指令。在编译安装 Node.js 时需要先查看一下 README.md 文件,其中会告诉你安装时依赖的库和软件版本信息。例如 0.8.14 版本中的 README.md 明确提示“Python 2.6 or 2.7”和“GNU Make 3.81 or newer”,以及“libexecinfo (FreeBSD and OpenBSD only)”。其说明 Python 版本在 2.6 或者 2.7,而 Make 版本要 3.81 以上。可以通过如下指令,查看系统的 Python 和 Make 版本信息是否符合 Node.js 安装环境。

```
python
make -v
```

Linux 系统下安装的相关步骤如下:

- (1) 从官网下载最新版的 Node.js for Linux 的源码安装包(32 位和 64 位)。
- (2) 解压压缩包 `tar -zxvf node-v0.8.8.tar.gz` (版本不同,名称也会不同)。
- (3) 进入解压文件夹 `cd node-v0.8.8.`。
- (4) 执行 `./configure`。
- (5) 执行 `make`。
- (6) 执行 `make install`。
- (7) 安装完成后执行 `node-v` 查看版本信息。

Node.js 在 `./configure` 时会先检查它安装所要依赖的包和库,如果系统已经安装,那么就可以正常通过,如果缺少安装包,系统会报错,把需要的安装包补上即可,如下是笔者在安装时遇到的依赖包问题,两个依赖包和库下载和配置方法如下:

```
sudo apt-get install build-essential //gcc
sudo apt-get install libssl-dev //ssl
```

以下一些错误信息,以及解决办法,可以作为参考。

```
[root@ouyo node-v0.8.14]# ./configure
File "./configure", line 355
1 if options.unsafe_optimizations else 0)
^
SyntaxError: invalid syntax
//=====
Env infos:
Linux ouyo.info 2.6.18-274.7.1.el5.028stab095.1 #1 SMP Mon Oct 24 20:49:24
MSD 2011 i686 i686 i386 GNU/Linux
Python 2.4.3
```

`./configure` 检查 Node.js 编译环境的依赖软件和依赖库是否支持,以上错误是 Python 版本过低,为 2.4.3,必须升级为 2.6 和 2.7 版本以上。Python 安装方法可以参考《Linux 下 Python 开发环境安装》¹文章。

¹ 该文章网址为: <http://www.cnblogs.com/chenzehe/archive/2010/10/20/1856437.html>。

```
make -C out BUILDTYPE=Release V=1
make[1]: Entering directory `/home/danhuang/node-v0.8.16/out'
  flock /home/danhuang/node-v0.8.16/out/Release/linker.lock g++ -pthread
-rdynamic -m32 -m32 -o /home/danhuang/node-v0.8.16/out/Release/mksnapshot
-Wl,--start-group
/home/danhuang/node-v0.8.16/out/Release/obj.target/mksnapshot/deps/v8/s
rc/mksnapshot.o
/home/danhuang/node-v0.8.16/out/Release/obj.target/deps/v8/tools/gyp/li
bv8_base.a
/home/danhuang/node-v0.8.16/out/Release/obj.target/deps/v8/tools/gyp/li
bv8_nosnapshot.a -Wl,--end-group
/home/danhuang/node-v0.8.16/out/Release/linker.lock: No such file or
directory
make[1]: *** [/home/danhuang
```

以上错误包含两个问题，第一是 Make 版本过低，第二个是需要创建一个 linker.lock 文件。可以首先检查 Make 的版本信息，之后再创建一个 linker.lock 文件，就能够成功安装了。

```
touch /home/danhuang/node-v0.8.16/out/Release/linker.lock
```

1.2.3 Hello World

成功搭建好 Node.js 的运行环境后，我们就可以开始编写第一个程序 Hello World。实现的主要需求是使用 Node.js 建立一个 Web 服务器，当请求服务器资源时，服务器响应一个 HTTP 请求，并显示 Hello World 字符串。

实现本需求会涉及 Node.js 的 HTTP 模块¹，该模块可以实现一个简单的 HTTP 服务器，相关代码以及实践步骤如下所述。

(1) 在任意文件夹中，创建 app（可以在系统的任何地方）文件夹，同时在文件夹 app 中创建 app.js 的 Node.js 文件，用任意编辑器打开 app.js 文件，输入如下 Node.js 代码后保存并退出。

```
/**
 *
 * Node.js base server
 */
var http = require('http'); //require http module
http.createServer(function(req, res) {
  res.writeHead(200, { 'Content-Type': 'text/plain' }); //set web http
  header
  res.end('Hello World\n'); //response http string to web client
}).listen(1337, "127.0.0.1"); //set http listen ip and port
console.log('Server running at http://127.0.0.1:1337/');
```

【代码说明】

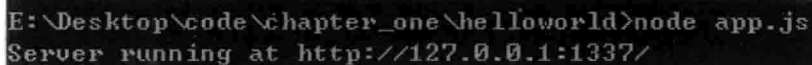
- ❑ require('http'): 获取 Node.js 原生模块提供的 HTTP 模块对象。
- ❑ res.writeHead: 通过 res 的 HTTP 响应对象，编写 HTTP 响应的头信息，并设定 Content-Type 指定返回的数据类型为文本 text，当然这里的数据类型也可以是其他格式，例如 html、css 和 image 等。

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

- ❑ `http.createServer()`: 使用 HTTP 对象 API 方法 `createServer` 来创建服务器。
- ❑ `listen`: 是 HTTP 对象的一个方法, 其主要是启动服务器监听的端口和 IP, 第二个参数为可选参数, 默认为本地 127.0.0.1。
- ❑ `console.log()`: 是 Node.js 和 JavaScript 共有的调试接口。

本段代码是应用 `require` 来调用 HTTP 模块中的方法和属性, 该 `require` 模块成功返回的是 Node.js 中 HTTP 模块中的方法和属性。如上代码中的 `listen` 接口有两个参数分别是运行端口号和服务器地址, 其中的端口号为一个 `Number` 类型, 服务器地址为字符串, 运行时如果端口号数据类型不正确会报错, 只要端口不被其他应用程占用, 都可以应用作为 Node.js 的服务端口。

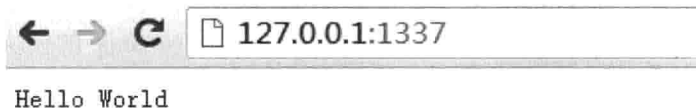
(2) 进入第一步创建的 `app` 文件夹, 运行 `node app.js` 指令, 执行正确后, 将会看到屏幕中输出的信息: `Server running at http://127.0.0.1:1337`, 如图 1-5 所示, 表明已经成功启动本地 IP 的 1337 端口 HTTP 服务器。



```
E:\Desktop\code\chapter_one\helloworld>node app.js
Server running at http://127.0.0.1:1337/
```

图 1-5 Node.js 的 `app.js` 运行结果

(3) 打开任意浏览器, 输入 `server` 服务器地址: `http://127.0.0.1:1337`, 在 Web 页面中我们将可以看到 `Hello World` 字符串, 如图 1-6 所示。



← → ↻ 127.0.0.1:1337

Hello World

图 1-6 Node.js Web 页面返回

上面就是我们的第一个程序 `Hello World` 的整个实践过程, 作为新的语言读者一般都会遇到一些棘手的问题, 在 1.2.4 节中列举了一些在 QQ 群以及论坛经常被提及的新手入门问题。

1.2.4 常见问题

初学者一般会遇到以下几个问题, 这里是个人在 QQ 群里面解答问题时积累下来的一些问题。

- ❑ 出现 `Error: Cannot find module`。这个问题主要原因是在当前目录没有找到 `app.js` 文件, 也就是执行的 `app.js` 文件不在当前目录, 需要指定执行 `app.js` 文件的路径。例如 `app.js` 在 `C:\Users\Administrator\Desktop` 文件夹, 而当我们在 `C:\Users\Administrator` 文件夹下运行 `app.js` 时就会出现如上问题, 如图 1-7 所示。通俗地说, 就是希望运行 `app.js`, 但是当前目录下没有该文件存在, 因此导致无法运行。

```

C:\Users\Administrator>node app.js

module.js:340
  throw err;
    ^
Error: Cannot find module 'C:\Users\Administrator\app.js'
    at Function.Module._resolveFilename (module.js:338:15)
    at Function.Module._load (module.js:280:25)
    at Module.runMain (module.js:492:10)
    at process.startup.processNextTick.process._tickCallback (node.js:244:9)

```

图 1-7 错误执行 app.js 脚本

- ❑ 出现 Error: listen EACCES。这个问题的主要原因是监听端口被其他应用程序占用了,可以修改当前监听端口,例如将 1337 修改为 3000,Windows 下可以通过 `netstat -aon|findstr "1337"` 来查看被哪个程序占用,Linux 下可以应用 `ps -ef|grep 1337` 来查看。这种异常情况很多时候是因为上一次服务器启动监听后非正常的退出,导致上一个端口未被释放,例如在 Linux 下使用快捷键 `Ctrl+Z` 退出。
- ❑ 如何退出当前运行程序? 正常退出监听端口的方法是按快捷键 `Ctrl+C` (Windows 和 Linux 是一样的)。
- ❑ 更改 Node.js 文件,希望服务器立即响应更改,而不需要重新运行文件。app.js 输出的是 Hello World,但现在想输出 hello baby,难道要重启 Node.js 的启动文件吗? 使用 `node` 命令运行 js 文件时,服务器是不会自动监控文件更改然后重启的,这里需要使用到一些 github 用户开发的 Node.js 开源模块。常见的有 `node-dev`¹ 和 `forever`²,其安装配置方法这里就不细介绍,关于 Node.js 的模块安装将会在第 2 章详细介绍。

1.3 异步编程

什么是异步编程? 异步编程给程序员带来了哪些编程困难? 我们应该如何克服异步编程带来的困难? 这些都是本节需要解决的问题,学完本节需要掌握异步编程的基本概念,以及异步编程的一些基本知识。

在笔者的博客中有一篇文章叫做《同步调用、回调和异步调用区别》³里面使用伪代码的“烽火长城”例子介绍这三者之间的关系,本节将会更详细地通过代码来阐述它们的区别和联系,同时介绍 Node.js 的异步编程思想和注意的问题。

1.3.1 同步调用和异步调用

同步调用是一种阻塞式调用,一段代码调用另一段代码时,必须等待这段代码执行结束并返回结果后,代码才能继续执行下去。例如考试的时候,有的同学是这样的:一道题

1 node-dev 网址: <https://github.com/fgnass/node-dev>。

2 forever 网址: <https://github.com/nodejitsu/forever>。

3 博客网址: <http://blog.csdn.net/danhuang2012/article/details/7897852>。

一道题往下做，当有一道题没做出来时，绝对不会继续做下去，这就如同同步调用的过程，一段逻辑没有执行完成时，代码会一直等待，直到代码执行结束，才执行下面的逻辑。下面看一个同步调用的 PHP 例子，通过例子来详细说明同步调用的思想。代码如下：

```
/**
 *
 * 文件名: answer.php, PHP 测试代码, 应用代码说明同步调用思想
 */
class Person
{
    public static $question = array();           //defined static parameters

    public static function answer(){             //defined static function
        echo 'success';
    }

    public static function think(){              //defined a think function
        sleep(5);                                //wait for 5 minutes
        return true;
    }
}
$ret = Person::think();                         //调用 Person 的静态方法 think, 等待 5 秒
if($ret){
    Person::answer();                           //调用 Person 的静态方法 answer 输出字符
} else {
    echo "answer wrong";
}
```

【代码说明】

- ❑ Person: Person 是一个静态类，其中有两个方法 think 和 answer。
- ❑ think 方法: 思考 5 秒钟，并返回成功结果。
- ❑ answer 方法: 回答问题，并且输出结果。

从代码运行结果可以看出，我们必须等待 think 方法执行 5 秒结束后，得到返回结果，才能执行 answer 方法。像这种必须等待一个程序运行结束，才能执行下一段程序的调用模式称为阻塞式调用，也就是同步调用。

异步调用是一种非阻塞式调用，一段异步代码还未执行完，可以继续执行下一段代码逻辑，当代码执行完以后，通过回调返回继续执行相应的逻辑，而不耽误其他代码的执行。

例如，一个经理拿到多个任务，他可以将任务同时分配给多个人去完成，而下属完成后就将结果反馈给他，当他接到所有反馈后就表明该任务完成了。经理分配给员工完成任务就相当于一个异步执行的代码执行过程，每个员工之间是不会相互阻碍的，当员工完成时，他会告知经理“我已经完成任务了”。这整个任务的执行就是一个异步执行的过程，代码的异步调用也一样。

同样，在考试时，有的学生思考一道问题，当问题没有解决时会继续看下一题，而当该问题有所思路时，则会返回继续解答该题，这就如同异步调用的过程。下面的代码是如上例子的一个异步调用方法，代码先执行 think 方法，当 think 方法没有结束时，代码会继续执行 answer 方法，当 think 方法结束时，通过回调函数执行相应逻辑，示例 Node.js 代码如下：

```

/**
 *
 * 文件名: answer.js, Node.js 测试代码, 应用代码说明异步调用思想
 */

function Person(){

    this.think = function(callback){ //定义 Person 对象的 think 方法

        setTimeout(function(){console.log('thinking~~~!');callback()},5000);
    }

    this.answer = function(){          //定义 Person 对象的 answer 方法
        console.log('I am answering other question');
    }
}

var person = new Person();           //new 创建 Person 对象
person.think(function(){              //根据 person 对象调用 think 方法
    console.log('tinking 5 second, get the right answer!')
});
person.answer();                      //根据 person 对象调用 answer 方法

```

【代码说明】

- ❑ Person: Person 是一个类, 其中有两个方法 think 和 answer;
- ❑ think 异步方法: 异步等待思考 5 秒钟, 思考结束后输出'thinking~~~'同时执行 callback()函数;
- ❑ answer 方法: 输出正在解答其他问题'I am answering other question'。

以上代码执行结果如图 1-8 所示。

```

C:\Users\Administrator\Desktop\code>node answer.js
I am answering other question
thinking~~~!
tinking 5 second, get the right answer!

```

图 1-8 Node.js 异步代码执行结果

如图 1-8 所示的结果中我们可以看到输出的首先是 person.answer()的执行结果, 而 person.think 输出结果在 5 秒后输出。因此可以看出当我们还在 think 的时候 (输出提示 'thinking~~~'), 就已经在解答其他问题了 (输出提示 'I am answering other question')。

代码执行过程是先输出 person.answer()的结果, 而当思考函数 think 方法结束后才执行输出'thinking~~~!'和'tinking 5 second, get the right answer!'. 由此看出, 虽然 person.think()先执行, 由于该方法是一个异步函数 (该函数中使用了 Node.js 中的 setTimeout 异步接口), 因此异步调用时并不等待 think 的思考结束, 而是先执行 person.answer()。像这样一段代码未执行完, 代码运行不会被阻塞, 而是继续执行下一段代码的调用模式称为异步调用。

上面代码在说明, 虽然 person.think()这个函数需要思考 5 秒, 但是由于 person.think()是一个异步执行代码, 因此整个代码逻辑的执行过程不会被阻塞, 所以会继续执行 person.answer()方法, 因此先返回 person.answer()执行的结果, 5 秒结束后才返回 person.think()执行结果。

 **注意:**有读者会提问“为什么 `person.think()` 是一个异步方法, 而 `person.answer()` 却不是”?

主要原因在于 `person.think()` 调用了 Node.js 中的异步函数 `setTimeout()`, 因此我们常说的 Node.js 拥有异步非阻塞特性, 并不是说 Node.js 中所有的代码逻辑都是异步执行的, 这要取决于是否在代码逻辑中应用其异步函数。

1.3.2 回调和异步调用

首先明确一点, 回调并非是异步调用, 回调是一种解决异步函数执行结果的处理方法。在异步调用, 如果我们希望将执行的结果返回并且处理时, 可以通过回调的方法解决。为了能够更好的区分回调和异步回调的区别, 我们来看一个简单的例子, 代码如下:

```
/**
 *
 * 文件名: callback.js, Node.js 示例代码, 说明非异步接口调用的 Node.js 代码执行过程
 同样是同步的-
 */

function waitFive(name, function_name){ //定义 waitFive 方法, 该方法回调等待 5 秒
    var pus = 0;
    var currentDate = new Date();
    while(pus<5000){ //等待 5 秒
        var now = new Date();
        pus = now - currentDate;
    }
    function_name(name); //执行回调函数
}
function echo(name){ //定义回调函数 echo()
    console.log(name);
}
waitFive("danhuang", echo); //调用 waitFive 方法
console.log("its over");
```

【代码说明】

- `waitFive()`: 同步函数, 逻辑是等待 5 秒钟后, 执行一个回调函数;
- `echo()`函数方法: 输出 `name` 信息。

上面代码逻辑是通过 `while` 循环来判断等待的时间是否超过 5 秒, 时间超出 5 秒后执行一个回调函数 `echo()`, 打印输出信息。注意这里的 `waitFive()` 是一个同步函数, 因为其实没有应用到任何异步接口。在代码等待 5 秒后执行结果如图 1-9 所示。



```
C:\Users\Administrator\Desktop\code>node callback.js
danhuang
its over
```

图 1-9 同步 Node.js 示例代码运行结果

从执行结果上来看, 其执行过程是一个同步执行过程。如果是异步执行的话, 其执行结果应该是先输出 `its over`, 再打印 `danhuang`。

以上代码是一个回调逻辑，但不是一个异步代码逻辑，因为其中并没有涉及 Node.js 的异步调用接口。从如图 1-9 所示的结果中能看出回调和异步调用的区别，当 waitFive() 函数执行时，整个代码执行过程都会等待 waitFive() 函数的执行，而并非如异步调用那样 waitFive 未结束，还会继续执行 console.log("its over")。因此，回调还是一种阻塞式调用。

1.3.3 获取异步函数的执行结果

异步函数往往不是直接返回执行结果，而是通过事件驱动方式，将执行结果返回到回调函数中，之后在回调函数中处理相应的逻辑代码。大家可以在官网看到很多的 API，都是通过异步回调方式来调用的。

Node.js 中很多 API 的调用模式是异步调用的，因此在学习 Node.js 过程中理解异步调用、同步调用和回调是非常重要的。例如之前的"Hello World"代码中涉及一个文件的读取，代码中使用的是 Node.js 中的同步执行的文件读取 fs.readFileSync (filename, [encoding]) API，该 API 是一个同步执行函数，与之对应的异步函数是 fs.readFile (filename, [encoding], [callback])，该 API 携带了一个 callback 参数，由于其 API 是一个异步执行函数，需要通过回调函数来处理异步返回结果。同步代码中获取执行结果方法：

```
var name = getName('danhuang');  
console.log(name);
```

而如果 getName() 函数是一个异步函数，我们用同样的方法来获取 name 值时，会出现异常，提示 name 没有定义 undefined。如下获取一个异步函数的结果的代码是错误的：

```
var dns = require('dns'); //require dns 模块  
var address = dns.resolve4('www.qq.com', function(address){});  
console.log(address); //dns 同步解析
```

打印出 address，可以看到第二个例子结果为 null，原因很简单，异步函数 dns.resolve4() 还未执行结束时，就已经执行到 console.log(address)，因此最终 address 为 null。既然异步函数出现这个问题，我们就可以使用回调去获取参数。如下代码，通过回调函数获取执行的结果 name 值：

```
var dns = require('dns');  
dns.resolve4('www.qq.com', function(address){ //dns 异步解析  
console.log(address);  
});
```

这样，同样可以实现打印 address 结果，主要的方法是通过回调函数 function(address){} 获取 resolve4 异步执行的结果。

1.4 本章实践

1. 在浏览器输入 http://127.0.0.1:2000，输出'you are so good'。

分析：根据 1.2 节中介绍的 Hello World 展示方法，将其中的 res.end 返回的字符串修

改为'you are so good'即可，代码如下：

```
var http = require('http');
http.createServer(function(req, res) {
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('you are so good\n');
}).listen(1337, "127.0.0.1");
console.log('Server running at http://127.0.0.1:1337/');
```

2. 在浏览器输入 `http://127.0.0.1:3000`，显示一个任意的 html 页面数据，其 html 页面数据字符代码如下：

```
"<html>
<head>
    <title>Node.js Test</title>
</head>
<body>
    <div>Hi Node, I like you so much</div>
</body>
</html>"
```

分析：res 对象可以响应多种方式，在 1.2 节的示例中只使用了其 text 返回类型，其还可以包含 css、html、jpg 等类型，根据本习题要求返回 html 格式，因此修改其 Content-Type 为 text/html，相关代码如下：

```
var http = require('http');
http.createServer(function(req, res) {
    var retHtml = "<html><head><title>Node.js Test</title></head><body>
    <div>Hi Node, I like you so much</div></body></html>"

    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.end(retHtml);
}).listen(1337, "127.0.0.1");
console.log('Server running at http://127.0.0.1:1337/');
```

3. 现在有 getFileData、returnData 和 fs.readFile 三个方法，其中 fs.readFile 是一个异步函数，是已知的，编程实现使用 returnData 调用 getFileData 方法，在 getFileData 中停留一秒后，打印出一个 index.conf 文件里面的二进制数据（内容为 hello world!）。

分析：参考代码如下：

```
var fs = require('fs');
function getFileData(callback){
    fs.readFile('index.conf', function(data){
        callback(data);
    });
}
/**
 *
 * @desc 异步执行等待 1 秒
 */
function returnData(callback){
    getFileData(function(data){
        setTimeout(function(){
            callback(data);
        }, 1000);
    });
}
```

本题的解决方案主要是应用 `callback` 参数来获取异步函数的返回结果。

1.5 本章小结

入门学习者经常会问一些关于 Node.js 开发的 IDE，笔者推荐轻量级的有 Nodepad++ 和 sublime¹，webstorm 就相对重一点，但是其拥有代码校验以及提醒功能，相对来说是非常完善，因此对于初学者建议其使用 webstorm，或者 eclipse 装 Node.js 插件。

Eclipse 对 Node.js 的支持还是比较全面的，其插件安装方法可以参照如下步骤：

(1) 启动 eclipse，并选择 help（帮助）下的 Install New Software，执行过程如图 1-10 所示。



图 1-10 插件安装指引

(2) 在弹出来的对话框中输入 `http://dl.bintray.com/nodeclipse/10.7.01`，查询相应的 eclipse 的 Node.js 插件。

(3) 查询结束后，选择 Node.js 的插件安装，如图 1-11 所示。

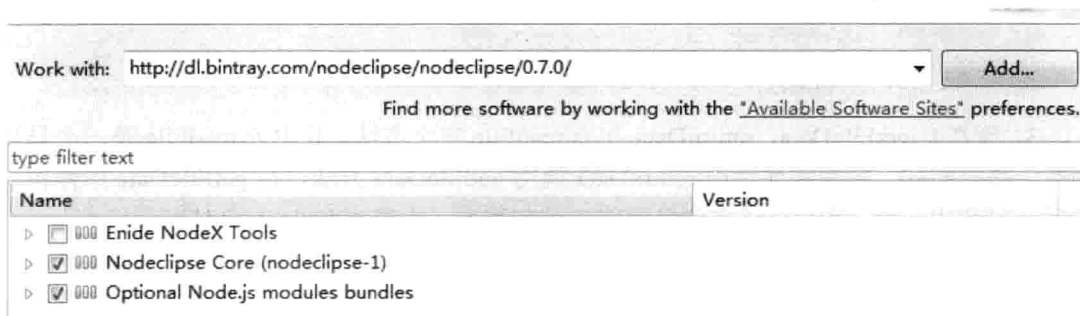


图 1-11 选择安装 Node.js 的插件提示

(4) 插件安装结束后，重启 eclipse，然后新建一个 Node.js 项目即可。

本章主要介绍了 Node.js 概述、Node.js 的环境搭建，以及入门开发 'Hello World' 程序。学完本章可以掌握 Node.js 的环境搭建，以及 Node.js 的基本程序的开发应用。

理解程序的同步调用、异步调用和回调对于 Node.js 的学习非常重要，因此在入门阶段给读者介绍这部分知识。

读者需要掌握 Node.js 的环境搭建、Node.js 的简单 HTTP 服务器的创建、了解 Node.js 中的异步编程思想。

本章涉及 Node.js 的 API 如表 1.1 所示。

¹ 代码 Sublime Text 具有漂亮的用户界面和强大的功能，例如代码缩略图、Python 的插件、代码段等。

表 1.1 本章Node.js的API列表

模块名	API 名	作用	调用示例
HTTP	<code>http.createServer([requestListener])</code>	创建 HTTP 服务器	<pre>http.createServer(function(req, res) { }).listen(1337, "127.0.0.1");</pre>

第 2 章 模块和 NPM

本章通过 DNS 解析的 Web 应用程序开发来介绍 Node.js 中模块的概念，同时了解在实际开发中模块的重要性。本章还会介绍一些开源的 NPM 应用模块，包括 jade、socket.io 和 express 等。Node.js 在模块化的应用中，exports 和 module.exports 的作用非常重要，因此在本章中会详细介绍 Node.js 提供的 exports 和 module.exports 的区别和联系。本章着重通过实例，引领读者深入了解 Node.js 的应用开发。

在介绍 NPM 模块后，还将扩展读者的阅读知识面，教大家如何发布自己的 NPM 模块，供他人学习和利用。

在模块介绍完以后，我们会和读者一起来探讨 Node.js 的设计模式。在学习 Node.js 的设计模式前，我们要了解以下几个问题，JavaScript 中类和对象是什么、Node.js 中如何实现类、类的继承应该如何实现、设计模式在 Node.js 中应该如何应用，这几点在 2.3 节介绍到。

本章主要是结合实例介绍 Node.js 的模块，希望学习本章时读者能够多实践、多编程、多思考。

2.1 什么是模块

本节主要介绍 Node.js 中模块的定义和一些基本概念，其中包含 Node.js 中的原生模块和文件模块的概念和应用。

通过实例来讲解 Node.js 中模块化的应用开发。本节应用到的一个 Web 实例叫做 DNS 解析服务，该应用的主要功能是实现一个在线的实时 DNS 解析工具。

2.1.1 模块的概念

模块分为两类：原生模块和文件模块。原生模块即 Node.js API 提供的原生模块，原生模块在启动时已经被加载。文件模块为动态加载模块，加载文件模块的工作主要由原生模块 module 来实现和完成。原生模块在启动时已经被加载，而文件模块则需要通过调用 Node.js 的 require 方法来实现加载。

需要了解的一点是，Node.js 会对原生模块和文件模块都进行缓存，因此在第二次 require 该模块时，不会有重复开销去加载模块，只需要从缓存中读取相应模块数据即可。

2.1.2 Node.js 如何处理模块

Node.js 中的原生模块和文件模块的调用方法存在一定的区别，但是二者的应用是大同

小异的。接下来我们通过实例代码的应用来学习一下两者的加载和调用方法。

1. 原生模块的调用

应用 Node.js 提供的 API `require` 来加载相应的 Node.js 模块, `require` 加载成功后会返回一个 Node.js 模块的对象, 该对象拥有该模块的所有属性和方法, 如下代码所示。

```
var httpModule = require('http'); //require http 模块
```

- ❑ "http": HTTP 是 Node.js 提供的原生模块, 该模块中有 `createServer`、`request` 和 `get` 等多个方法和属性, 这些属性和方法可以在 Node.js 的官方 API 文档¹中查找到。
- ❑ `httpModule`: 则为 `require` 原生 HTTP 模块后返回的对象, 通过该对象可以调用 HTTP 模块的所有属性和方法。

```
httpModule.createServer(function(res, req)).listen(port)
//调用 httpModule 的对象方法
```

- ❑ `httpModule` 对象调用了 HTTP 模块中的 `createServer` 和 `listen` 方法来创建简单的 HTTP 服务器。

以上就是一个简单的调用原生模块的方法, Node.js 中的其他原生模块的调用方法都是一样的, 主要是学会如何查看 Node.js 的 API 文档, 以及如何应用其中的模块提供的方法和属性。

2. 文件模块调用方法

文件模块的调用和原生模块的调用方式基本一致, 但是需要注意的是, 其两者的加载方式存在一定的区别, 原生模块不需要指定模块路径, 而文件模块加载时必须指定文件路径, 否则会出错提示 "can not find XX module"。如下代码为 Node.js 的文件模块加载方法。

```
var test = require('/path/.../test.js')
//也可 var test = require('/path/.../test')
```

- ❑ `/path/.../test.js`: 代表该 `test` 文件模块的绝对路径, 如果需要使用相对路径, 则需要路径前添加一个 `./`, 例如 `require("./test.js")`。
- ❑ `/path/.../test`: 可以省略模块 `js` 的后缀名。
- ❑ `test` 返回值: `require` 加载模块成功后返回的同样也是一个对象。

在得到 `require` 返回对象 `test` 以后, 那么该 `test` 对象可以调用文件模块的哪些属性和方法呢? Node.js 中明确指出, 在文件模块中, 只有 `exports` 和 `module.exports` 对象暴露给该外部的属性和方法, 才能够通过返回的 `require` 对象进行调用, 其他方法和属性是无法获取的, 因此 `test` 对象只能调用 `test.js` 文件模块中的 `exports` 和 `module.exports` 的方法和属性。例如 `test.js` 中的 `love` 方法和 `yourName` 属性不能通过返回的 `test` 对象进行调用, 而由于 `name` 属性和 `happy` 方法是应用 `exports` 暴露给调用者, 因此其二者可以通过 `test` 对象访问。相关代码如下所示。

¹ 参考网址 <http://nodejs.org/api/>。

```

/**
 *
 * exports test code
 */

exports.name = "danhuang"; //exports 暴露 name 属性
exports.happy = function(){console.log("mm")}; //exports 暴露 happy 方法
var yourName = "reader";
function love(){
    console.log("mm vs gg");
}

```

【代码说明】

- ❑ test 对象可见属性和方法：只有 name 属性和 happy 方法是通过调用 exports 的，因此对于 require 加载模块返回后的 test 对象来说，其可见的方法和属性只有 name 和方法 happy。
- ❑ test 对象不可见属性和方法：yourName 属性和 love 方法由于没有调用 exports，返回暴露给外部调用者，因此对 require 返回的对象 test 是不可见的，这点类似于类的私有成员和私有方法。

为了验证上述的理论，我们通过代码打印出 require 后的 test 对象信息，查看其拥有的属性和方法。

```

var test = require('./test.js');
console.log(test);

```

结果返回信息是 { name: 'aa', happy: [Function] } 的一个 Json 对象。可以看出，对于 require 返回的对象而言，仅有 exports 和 module.exports 的方法和属性是可见的，其他的方法和属性都是不可见的。很多读者在 require 对象的时候经常看到一些 Error: Cannot find module '/test/test.js' 错误提示。原因在第 1 章中我已提及过，主要是 require 的时候路径不正确，导致无法获取该模块。下一节将会根据实例来介绍 Node.js 中原生模块和文件模块的应用。

2.1.3 Node.js 实现 Web 解析 DNS

本节应用 Node.js 的原生模块和文件模块两个方法来实现 Node.js 的 Web DNS 解析工具，通过分析对比，来说明文件模块存在的必要性，以及其存在的意义。

下面利用 Node.js 的原生模块来实现一个 Web 版的 DNS 解析应用。本实例的代码可参考本书源码中的 chapter_two\module\parse_dns_1。下面会逐步地引导大家编程学习。

1. 创建 parse_dns_ex.js，加载需要的 Node.js 原生模块

首先了解我们应用中需要的 Node.js 原生模块、每个模块的作用，以及需要的 API。

- ❑ 创建 Web 的 HTTP 服务器，需要 Node.js 的 HTTP 模块。
- ❑ DNS 解析，需要 Node.js 的 dns 模块。
- ❑ 读取一个 HTML 页面，需要 Node.js 的 fileSystem 模块。
- ❑ 处理请求参数，需要 Node.js 的 querystring 模块。

分析完这些，接下来就可以先完成 Node.js 代码的第一部分，获取需要的 Node.js 原生


模块对象。

```
/* 首先 require 加载需要的 Node.js 原生模块 */
var http = require('http'),           //服务器创建
    dns  = require('dns'),             //DNS 查询
    fs   = require('fs'),              //文件操作
    url  = require('url'),             //url 处理
    querystring = require('querystring'); //字符串处理
```

【代码说明】

- ❑ http 模块负责 HTTP 服务器的创建。
- ❑ dns 模块主要负责解析当前 DNS 域名，返回 DNS 服务器 IP 地址。
- ❑ url 模块处理文件 url 路径。
- ❑ querystring 模块处理前端传回的字符串解析。

以上代码的作用就是引用 Node.js 的原生模块，将 require 返回的对象分别赋值给 http、dns、fs、url、querystring 这些变量。

 **注意：**作为好的代码设计，最好将所需要的对象统一定义初始化，或者使用作用域来初始原生模块对象。虽然 Node.js 提供了模块的缓存机制，但不建议在一个文件中多次 require 同一个对象。

2. parse_dns_ex.js 添加创建 HTTP 服务器代码，返回显示 index.html

由于需求实现的目的是读取一个 html 文件，并非和例子 Hello World 那么简单只需要返回显示 Hello World 文本字符。如果 HTTP 响应的是字符数据时，使用 Content-Type 类型值为 text/plain，而当 HTTP 响应的是 html 数据时，返回到客户端的 Content-Type 类型应该修改为 text/html。

其他代码实现方式与 Hello World 实例类似。思路主要是读取一个 html 页面的数据，将其得到的数据通过 HTTP 响应到 Web 客户端，并设置 HTTP 响应数据类型为 text/html，当数据响应到客户端时，客户端会根据 Content-Type 类型解析相应的数据并显示。服务器代码创建如下所示。

```
http.createServer(function(req, res) {
  /* 写 http head 返回 html，因此 Content-Type 为 html*/
  res.writeHead(200, { 'Content-Type': 'text/html' });
  /* 获取当前 index.html 的路径 */
  var readPath = __dirname + '/' + url.parse('index.html').pathname;
  var indexPage = fs.readFileSync(readPath);
  /* 返回 */
  res.end(indexPage);
}).listen(3000, "127.0.0.1");
```

【代码说明】

- ❑ res.writeHead: 添加 HTTP 响应的头信息。
- ❑ var readPath: 获取 html 文件路径。
- ❑ var indexPage: 读取 html 文件数据。
- ❑ res.end(indexPage): 执行 HTTP 响应返回到 Web 客户端。

http 对象的 `createServer` 方法创建一个 HTTP 的服务器，其监听 IP 和端口分别是 127.0.0.1、3000，当客户端请求服务器数据时，服务器获取 html 页面数据，并将数据以 text/html 数据类型返回到客户端。接下来创建一个显示的 html 页面，代码如下：

```
<html>
  <head>
    <title>DNS 查询</title>
    <meta http-equiv="content-type" content="text/html; charset=
      utf-8">
  </head>
  <body>
<!--html body-->
    <h1 style='text-align:center'>DNS 查询工具</h1>
    <div style='text-align:center'>
      <form action="/parse" method="post">
        查询 DNS: <input type='text' name='search_dns' />
        <input type='submit' value='查询' />
      </form>
    </div>
  </body>
</html>
```

表单提交一个 DNS 域名信息到 `/parse(127.0.0.1/parse)` URL。以上代码主要是创建一个 Web 表单页面，其将用户输入的文本 DNS 域名字符数据，提交到 `127.0.0.1/parse`，到这一步，我们就可以实现一个简单的 Web 页面展示功能了，下面介绍运行方式。

3. 运行 `parse_dns_ex.js`

运行命令脚本 `parse_dns_ex.js`：

```
node parse_dns_ex.js
```

打开任意浏览器，输入 Node.js 监听端口和 IP，`http://127.0.0.1:3000`，服务器会返回如下页面到客户端，如图 2-1 所示。

DNS查询工具

查询DNS:

图 2-1 DNS 查询工具 Web 页面

以下代码部分实现的仅仅是一个 DNS 表单提交页面，对于 DNS 解析部分还没有做任何处理，因此当我们提交数据的时候，系统会出现一些异常。接下来将介绍 Node.js 如何利用内置模块 API 来解析 DNS 域名。

4. 新增处理 `/parse` 请求路由

`parse_dns_ex.js` 代码中没有做任何的路由处理，所有的请求都显示 `index.html`，现在我

们需要对 `http://127.0.0.1:3000/parse` 做不同的请求处理，因此需要一个控制器来判断区分用户请求的数据显示方式，是显示页面 `index.html`，还是执行 `/parse` 解析域名。而这个控制器就是路由处理器。

在原有代码中的 `http.createServer` 回调函数中添加一个路由处理方法 `router(res, req, pathname)`，其主要作用是判断客户端请求资源是什么，根据不同的请求，执行不同的代码逻辑，并响应不同的数据。代码如下：

```
http.createServer(function(req, res) {
  /* 写 http head 返回 html，因此 Content-Type 为 html */
  var pathname = url.parse(req.url).pathname;
  req.setEncoding("utf8");
  res.writeHead(200, { 'Content-Type': 'text/html' });
  router(res, req, pathname); //调用 router 方法来处理 url 路由
}).listen(3000, "127.0.0.1");
/* 打印运行 log */
console.log('Server running at http://127.0.0.1:3000/');
```

【代码说明】

- ❑ `router(res, req, pathname)`：处理不同 url 请求资源，分发处理；
- ❑ `var pathname = url.parse(req.url).pathname`：获取当前请求资源的 url 路径，例如 `"/parse"`；
- ❑ `req.setEncoding("utf8")`：设置返回客户端页面的数据格式，这里设置为 `utf8`，如果不设置有可能会乱码情况。

在服务器 HTTP 响应回调函数中添加了 `router` 代码逻辑函数。接下来，我们看看该路由函数 `router()` 是如何实现处理不同的请求，并分发逻辑处理的，下面是 `router()` 函数的代码实现逻辑。

```
function router(res, req, pathname){
  switch (pathname){ //根据 pathname 不同，执行不同的处理逻辑
    case "/parse":
      parseDns(res, req)
      break;
    default:
      goIndex(res, req)
  }
}
```

【代码说明】

- ❑ `switch (pathname)`：判断当前的 `pathname` 是什么参数。
- ❑ `case "/parse"`：如果请求路径是 `"/parse"` 时，执行 `parseDns(res, req)` 域名解析。
- ❑ `default`：其他情况 `goIndex(res, req)` 显示 `index.html` 页面。

可以看到这里通过一个 `switch` 来处理请求的路径，分发处理逻辑，当请求路径为 `"/parse"` 时，才执行路径解析，其他情况都调用 `goIndex()` 函数，返回到 `index.html` 主页。

上面代码中既然涉及 `goIndex()` 和 `parseDns()` 方法，那么接下来我们来看一下，如何实现 `goIndex()` 和 `parseDns()` 这两部分逻辑。注意，两个函数 `goIndex()` 和 `parseDns()` 时刻携带了两个参数 `req` 和 `res`，为什么要这两个参数，以及它们的作用是什么？

回答这个问题前，首先要解决如下几个问题：

它们是如何产生？

它们两者的主要功能和作用是什么？

Node.js 的 API 提供的 `http.createServer` 方法，会有一个回调函数，而 `res` 和 `req` 就是回调函数的两个参数，那么两者作用在哪里？`res` 和 `req` 分别是 `response`（响应）和 `request`（请求），顾名思义 `res` 处理 HTTP 的响应事件，而 `req` 处理 HTTP 的请求事件。如果读者想了解这两个对象有哪些方法和属性的话，可以使用 `console.log` 将其打印查看其中的属性和方法。

通过小知识，我们可以了解到其中的 `req` 参数为客户端请求对象，该对象包含了用户请求的 HTTP 头以及请求的一些参数等，根据 `req` 对象中的方法和属性，我们可以获取客户端传递的参数。而 `res` 为响应客户端请求对象，该对象包含了一些 HTTP 响应属性和方法，通过该对象响应数据到客户端。

上面路由代码中涉及的 `goIndex()` 函数，其功能是显示一个 `index.html` 页面，只需要一个 `res` 响应事件对象，`req` 对象可以不需要，因为其不需要获取客户端数据。`goIndex()` 响应一个指定的 `html` 数据到客户端，当然如果是根据不同的请求路径，响应不同的 `html` 数据时，那么这两者则缺一不可。这部分没有那么复杂，因此可以忽略 `req` 参数对象，下面是实现的示例参考代码。

```
/**
 *
 * @desc 定义响应 html 页面的函数
 * @params res http 请求对象
 * @params req http 响应对象
 */
function goIndex(res, req) {
  /* 获取 index.html 的文件路径 */
  var readPath = __dirname + '/' + url.parse('index.html').pathname;
  /* 同步读取 index.html 文件的信息 */
  var indexPage = fs.readFileSync(readPath);
  /* 返回 */
  res.end(indexPage);
}
```

【代码说明】

- ❑ `var readPath`: 获取 `index.html` 文件的路径。
- ❑ `var indexPage`: 读取 `index.html` 文件数据，并将数据保存到 `indexPage` 变量中。
- ❑ `res.end(indexPage)`: 通过 `res` 响应 `html` 数据到客户端。

`goIndex()` 函数代码实现和响应一个 `html` 数据到客户端逻辑是类似的。接下来了解一下 DNS 解析函数 `parseDns()` 的实现。

`parseDns()` 函数主要应用是解析客户端传递来的域名，并且返回显示该域名相应的 IP 地址。在客户端中使用的是一个 `POST` 方法，因此这里涉及一个简单的问题：Node.js 中如何获取 HTTP 的 `POST` 参数？在 Node.js 中的 `req` 对象中有一个 `addListener` 方法，该方法就可以获取客户端 `POST` 来的数据。该接口是一个异步方法，携带的两个事件分别是 `data` 和 `end`，`data` 事件是指当有数据传递到服务器时触发的事件，而 `end` 则表示客户端所有数据传递完毕后触发的事件。因此在服务器端判断所有数据接收完成以后，执行回调函数来获取 `POST` 的参数，得到 `POST` 参数后才执行相应逻辑 `getDns`，也就是当 `end` 事件触发后 `req.addListener("end")`，才执行解析 DNS 函数并且返回结果，示例代码如下：

```

function parseDns(res, req){
    var postData = "";
    req.addListener("data", function(postDataChunk) {
        postData += postDataChunk;
    });
    /* HTTP 响应 html 页面信息 */
    req.addListener("end", function() {
        var retData = getDns(postData, function(domain, addresses){
            res.writeHead(200, { 'Content-Type': 'text/html' });
            res.end("<html>
<head>
    <meta http-equiv='content-type' content='text/html; charset=
utf-8'>
</head>
<div style='text-align:center'>
    Domain:<span style='color:red'>" + domain + "</span>
    IP:<span style='color:red'>" + addresses.join(',') + "</span>
</div></html>");
        });
        return;
    });
}

```

【代码说明】

- ❑ req.addListener("data",function(){}): 数据传递到服务器时触发的事件函数，读取客户端传递来的数据，获取数据后添加到 postData 中。
- ❑ req.addListener("end",function(){}): 数据接收完成，end 事件被触发后，执行 DNS 域名解析。
- ❑ getDns 异步解析域名，执行完成后回调执行 function(domain, addresses){}，domain 是传递的域名参数，addresses 是解析后返回的 IP 地址列表。

由于 HTTP 的 POST 数据是分段传递，因此我们需要使用一个监听事件来完成所有数据的接收逻辑。服务器端获取客户端传递的所有数据，将其有序地连接就是我们需要的 POST 数据。因此需要使用 addListener 来监听 POST 数据是否执行完成。

⚠注意：在上面的代码中，我们应用了 res.end 来响应一个 html 字符串，那么为什么不和原来一样读取一个 html 文件中的数据呢？原因在于我们需要在 html 中添加一些变量显示在 html 中，需要一个动态的 html。当然还有一种方法就是应用类模版，例如 jade 模版。

上面代码中涉及 getDns()方法，该方法是一个异步回调函数，通过异步执行 DNS 解析，然后回调执行数据返回到客户端，根据 domain 和 addresses 参数，参考代码如下：

```

function getDns(postData, callback){
    var domain = querystring.parse(postData).search_dns;
    /* 异步解析域名 */
    dns.resolve(domain, function(err, addresses){
        if(!addresses){
            addresses=['不存在域名']
        }
        callback(domain, addresses);
    });
}

```

【代码说明】

❑ `dns.resolve(domain, function(err, addresses){})`: 方法的使用可以查看官网 API 介绍。

❑ `var domain`: 应用 `querystring` 模块来获取 `post` 数据中键值为 `search_dns` 的值。

最后看一下 Node.js 是如何解析 DNS 的, 这里主要涉及 DNS 模块中的 `resolve` API 的使用。大家注意, `getDns()` 方法需要携带一个 `callback` 参数, 由于 `dns.resolve()` 方法是一个异步执行函数, 因此如果希望得到返回结果, 就需要添加一个回调函数, 将结果返回到回调函数中, 传递到函数外面, 关于这点在第 1 章中已经介绍过。

添加了一个 `callback` 参数应该如何得到返回结果呢? Node.js 中很多时候会涉及回调函数, 一般可以通过回调函数将执行结果返回给一个 `function`, 例如下面代码:

```
getDns(postData, function(domain, addresses){//do something})
```

代码中就是通过给 `getDns()` 方法添加一个回调函数 `function(domain, addresses){//do something}`, 获取异步函数 `dns.resolve` 执行的 `domain` 和 `addresses` 结果。在添加回调函数时, 也需要在异步执行函数中执行这个回调函数, 返回执行结果, 相关代码如下:

```
/* DNS 异步域名解析 */
dns.resolve(domain, function(err, addresses){
    if(!addresses){
        addresses=['不存在域名']
    }
    callback(domain, addresses);
});
```

代码中的 `callback(domain, addresses)` 目的就是在异步函数 `dns.resolve()` 执行结束后, 执行 `callback (domain, addresses)` 函数, 将执行结果 `domain` 和 `addresses` 返回到回调函数中处理。

最后运行 `parse_dns.js`, 打开浏览器输入 `http://127.0.0.1:1337/`, 然后在 `html` 表单中输入 `www.qq.com`, 提交表单后我们就可以查看到如图 2-2 所示的返回结果。

Domain:www.qq.com IP:182.254.0.217

图 2-2 应用解析返回信息

以上就是通过 Node.js 的原生模块实现了一个 Node.js 域名解析的网站, 当然其中还有很多的问题, 将在后面章节陆续介绍。

接下来将介绍如何使用 Node.js 的文件模块和原生模块来更好地实现域名解析这个 Web 应用。

2.1.4 Node.js 重构 DNS 解析网站

使用文件模块和 Node.js 原生模块重构 DNS 解析网站, 代码参考本书源码中的 `chapter_two\module\parse_dns_2`。文件模块的好处在于, 将业务处理分离, 每个模块处理相应职责, 避免业务混乱。例如火车站将整个系统分为售票服务、站前服务、站台服务、下车检票等, 那么我们可以利用文件模块, 将系统分为不同的功能模块去实现和完成, 例

如将系统分为服务器创建模块、路由处理模块、逻辑控制模块、数据处理模块、错误处理模块等。

接下来分析 DNS 解析系统需要划分哪些模块，以及这些模块之间的功能和作用分别是什么。根据业务和逻辑的不同，将本系统大致分为 4 个模块：入口模块、路由处理模块、DNS 解析模块、首页展示模块。入口模块负责服务器创建，路由处理模块负责 url 转发以及请求资源分配，DNS 解析模块负责 DNS 解析的业务逻辑，首页展示模块负责读取 index.html 页面并返回到客户端。接下来看一下各个模块是如何实现的，之后再学习如何将每个模块联系在一起，来保证整个项目的正常运行。

1. 入口模块 (index.js)

创建 http 服务器处理客户端请求，同时加载 router 文件模块，分发请求资源；在文件模块的调用方法上需要注意的是，文件模块必须使用 require 进行加载后返回文件模块的对象，所有 exports 的方法才能通过 require 返回的对象进行调用，其他 function 不能调用。代码中的 router(res, req, pathname)就是 router 文件模块中的 exports 方法，因此可以通过 router.router(res, req, pathname)调用。相关代码如下：

```
/* 首先 require 加载两个模块 */
var http = require('http'),
    url = require('url');
/* 加载文件模块 */
var router = require('./router.js');

/* 创建 http 服务器 */
http.createServer(function(req, res) {
  /* 写 http head 返回 html, 因此 Content-Type 为 html*/
  var pathname = url.parse(req.url).pathname;
  req.setEncoding("utf8");
  res.writeHead(200, { 'Content-Type': 'text/html' });
  router.router(res, req, pathname);
}).listen(3000, "127.0.0.1");
/* 打印运行 log */
```

【代码说明】

- ❑ router = require('./router.js'): require 文件模块 router.js，并且将返回对象赋值给 router。
- ❑ pathname: HTTP 请求路径。
- ❑ req.setEncoding(): 设置 HTTP 返回字符串编码，不设置可能出现乱码。

以上代码应用到了 router.js 文件模块，require 该模块后返回的 router 对象，拥有在文件模块 router.js 中 exports 的方法 router，该方法包含 3 个参数，分别是 res、pathname 和 req。req 是 HTTP 请求资源对象，res 是 HTTP 响应对象，pathname 是 HTTP 请求路径。接下来看 router.js 是如何处理 router 方法实现逻辑，以及如何实现一个路由处理。

2. 路由处理模块(router.js)

处理所有请求资源，分发到相应处理器。其处理逻辑主要是加载所有处理器文件模块，同时根据请求资源不同分发到不同的处理模块。与原生模块路由实现原理类似，都是使用

一个 router 函数中的 switch 方法，本模块的实现代码如下：

```
/* 路由模块处理 */
var ParseDns = require('./parse_dns.js'),
    MainIndex = require('./main_index.js');
/* 创建 router 方法，并将该方法暴露给外部接口 */
exports.router = function(res, req, pathname){
    switch (pathname){ //根据 pathname 来路由调用处理逻辑
        case "/parse":
            ParseDns.parseDns(res, req) //执行 DNS 解析
            break;
        default:
            MainIndex.goIndex(res, req) //响应 HTML 到客户端
    }
}
```

【代码说明】

- ❑ ParseDns = require('./parse_dns.js')：获取 DNS 解析的文件模块对象定义。
- ❑ MainIndex = require('./main_index.js')：获取处理首页显示的文件模块对象定义。
- ❑ exports.router = function(){}：目的是让 router 方法暴露给 require 返回的对象。
- ❑ switch(pathname)：根据请求参数不同，执行不同的模块函数。
- ❑ "/parse"：执行 DNS 模块 ParseDns 的方法 parseDns，解析域名。
- ❑ default：进入 index.html。

这里需要注意的是，router 方法必须应用 exports 暴露给 require 返回的对象，如果不使用 exports 方法，相对于 router.js 文件模块来说就是私有方法，require router 文件模块返回对象将无法调用。

3. DNS 解析模块 (parse_dns.js)

DNS 处理逻辑，根据获取的域名进行解析，返回相应的处理结果到页面，这部分代码和上面的原生模块的代码类似，主要是 parseDns 和 getDns 两个方法。代码如下：

```
/* dns 解析模块 */
var querystring = require("querystring"),
    dns = require('dns');
exports.parseDns = function(res, req){
    var postData = "";
    req.addListener("data", function(postDataChunk) {
        postData += postDataChunk;
    });
    /* HTTP 响应 html 页面信息 */
    req.addListener("end", function() {
        var retData = getDns(postData, function(domain, addresses) {
            res.writeHead(200, { 'Content-Type': 'text/html' });
            res.end("<html><head><meta http-equiv='content-type' content='text/html; charset=utf-8'></head><div style='text-align:center'>Domain:<span style='color:red'>" + domain +
                "</span> IP:<span style='color:red'>" + addresses.join(',') +
                "</span></div></html>");
        });
    });
}
```

```
});
return;
});
```

该部分代码主要是应用 `req.addListener` 获取 POST 数据。

```
/* 相当于私有方法 */
function getDns(postData, callback) {
  var domain = querystring.parse(postData).search_dns;
  /* 异步解析域名 */
  dns.resolve(domain, function(err, addresses) {
    if(!addresses) {
      addresses=['不存在域名']
    }
    callback(domain, addresses); //调用回调函数
  });
}
```

【代码说明】

- ❑ `exports.parseDns = function(res, req){}`: 将 `parseDns` 方法应用 `exports` 暴露给外部调用对象, 有点类似类中的 `public` 方法, 该方法的主要逻辑是执行 DNS 域名解析;
- ❑ `function getDns(){}`: 异步解析 DNS, 解析完成后执行回调函数 `callback()`, 通过 `callback()` 函数来传递执行的结果 `domain` 和 `addresses`。

```
var domain = querystring.parse(postData).search_dns
```

代码中涉及如何获取客户端提交的表单数据, `postData` 为客户端传递的所有数据, `search_dns` 为表单 `input` 的 `name`, 对应到 `index.html`, 可以看到 `dns` `input` 的 `name` 为 `search_dns`, 其他获取表单提交数据可以类同此方法。下面是 `html` 的表单代码:

```
<form action="/parse" method="post">
  查询 DNS: <input type='text' name='search_dns' />
  <input type='submit' value='查询' />
</form>
```

4. 首页展示模块 (main_index.js)

处理主页 `index.html` 页面的显示, 使用 `fs` 模块进行读取 `index.html` 页面字符数据, 然后返回到客户端。其方法和原生模块的代码类似, 这里不再详细介绍。

```
/* 处理首页逻辑信息 */
var fs    = require('fs'),
    url    = require('url');
/* 定义 goIndex 跳转首页函数 */
exports.goIndex = function(res, req) {
  var readPath = __dirname + '/' + url.parse('index.html').pathname;
  var indexPage = fs.readFileSync(readPath);
  /* 返回 */
  res.end(indexPage);
}
```

【代码说明】

- ❑ `fs = require('fs')`: 获取 `fs` 模块。
- ❑ `url = require('url')`: 获取 `url` 对象。
- ❑ `var indexPage = fs.readFileSync(readPath)`: 同步读取 `index.html` 页面数据。
- ❑ `res.end(indexPage)`: 响应返回一个 `index.html` 页面到客户端。

整个过程结束后, 运行 `index.js`, 同样可以实现 2.1.3 节中 DNS 解析实例中的功能, 那么为什么我们还需要文件模块来处理呢?

文件模块可以实现模块化编程, 便于项目的维护和开发。项目开发之前, 必须要做系统的架构分析, 将系统整个架构分析清晰后, 对 Node.js 的模块进行模块化面向接口编程, 这样不仅利于团队的合作开发, 同时在一定程度上还提高代码的可维护性和扩展性。通过上面两种方法的应用实现, 来突出显示 Node.js 中文件模块对代码设计的重要性。

2.1.5 exports 和 module.exports

`exports` 和 `module.exports` 的作用都是将文件模块的方法和属性暴露给 `require` 返回的对象进行调用。但是二者之间存在本质的区别, `exports` 的属性和方法都可以被 `module.exports` 替代, 如下面代码, 其作用都是一致的。

```
exports.mymame = 'danhuang'和 module.exports.mymame='danhuang'
```

但 `exports` 不能替代 `module.exports` 方法, 可以理解为包含关系。所有的 `exports` 对象最终都是通过 `module.exports` 传递执行, 因此可以更确切地说, `exports` 是给 `module.exports` 添加属性和方法。为了验证这一点, 我们将文件中的 `module.exports` 对象打印出来与 `exports` 方法和属性进行对比, 代码如下:

```
exports.name = "aa";
exports.happy = function(){console.log("mm")};
console.log(module.exports);
```

【代码说明】

- ❑ `exports.name = "aa"`: 使用 `exports` 将 `name` 属性暴露给 `require` 返回对象。
- ❑ `exports.happy = function(){console.log("mm")}`: 使用 `exports` 将 `happy` 方法暴露给 `require` 返回对象。
- ❑ `console.log(module.exports)`: 打印出 `module.exports` 对象。

执行以上代码返回如下结果:

```
{ name: 'aa', happy: [Function] }
```

从结果中可以看出, `module.exports` 相当于 `require` 返回的对象, 也就是所有 `require` 返回的对象, 实质上结果和 `module.exports` 是相同的。这里我们可以做个检测, 创建一个新的 `index.js` 文件, 代码如下:

```
var obj = require('./exports.js');
console.log(obj);
```

执行 `index.js` 文件, 运行结果如图 2-3 所示, 从结果中可以得出结论, `module.exports`

对象和 `require` 后返回的对象数据是相同的。

```
E:\Desktop\code\chapter_two\module\exports_module>node index.js
< name: 'aa', happy: [Function] >
< name: 'aa', happy: [Function] >
```

图 2-3 测试代码 `index.js` 运行结果

`module.exports` 方法还可以单独返回一个数据类型，而 `exports` 只能返回的一个 `object` 对象。当我们需要返回一个数组、字符串、数字等的类型时，就必须使用 `module.exports`。

例如，`name.js`，返回一个字符串数组，使用 `exports` 是无法实现的，但 `module.exports` 可以实现，代码如下：

```
module.exports = ['danhuang', 'is', 'my', 'name'];
```

在 `show.js` 中调用 `name.js` 的数组，同时利用数组的 `join` 方法，空格分割显示数组的每一项，代码如下：

```
msgArr = require('./name.js');
console.log(msgArr.join(' '));
```

执行 `node show.js` 后，结果中显示：

```
danhuang is my name
```

从结果可以看出，`msgArr` `require` 返回的是一个 `array` 数组，其中每一项分别是 `danhuang`、`is`、`my` 和 `name`。当使用了 `module.exports` 关键词以后，该模块中的所有 `exports` 对象执行的属性和方法都将被忽略。

例如：`ignore_exports.js`，通过 `module.exports` 返回一个字符串，`module.exports` 代码之后使用 `exports` 返回一个对象，其中有一个属性为 `name` 和一个方法为 `showName`。

```
module.exports = 'test for module.exorts ignore!';
exports.name = 'danhuang';
/* 定义 showName 函数，并暴露给外部接口 */
exports.showName = function() {
    console.log('My name is Danhuang');
};
console.log(module.exports );
```

【代码说明】

- ❑ `module.exports = 'test for module.exorts ignore!'`：使用 `module.exports` 返回一个字符串。
- ❑ `exports.name = 'danhuang'`：使用 `exports` 返回一个字符串。
- ❑ `exports.showName = function(){}:` 使用 `exports` 返回一个函数 `showName`。

创建执行文件 `ignore_exe.js`，主要作用是 `require ignore_exports.js` 文件，调用其中的 `name` 属性和 `showName` 方法。

```
var Book = require('./ignore_exports.js');
console.log(Book);
console.log(Book.name);
console.log(Book.showName());
```

运行 ignore_exe.js 文件，结果如图 2-4 所示。

```
E:\Desktop\code\chapter_two\module\exports_module\exports\ignore_exports>node ignore_exe.js
test for module.exports ignore!
test for module.exports ignore!
undefined

E:\Desktop\code\chapter_two\module\exports_module\exports\ignore_exports\ignore_exe.js:4
console.log(Book.showName());
               ^
```

图 2-4 测试代码 ignore_exe.js 运行结果

第一个 test for module.exports ignore!是在 ignore_exports.js 中打印 console.log(module.exports)module.exports 对象而来，而第二个 test for module.exports ignore!则是在 ignore_exports.js 文件 console.log(Book)中打印显示的，这也印证了上面所说的，module.exports 对象和 require 返回的对象数据是相同的。undefined 是 console.log(Book.name)，虽然 ignore_exports.js 中使用了 exports.name = 'danhuang'，但 require 无法获取该属性，主要原因是在 exports 之前执行了 module.exports 方法，导致 exports 失效，同时 showName()函数不存在也同样是这个问题。

2.2 NPM 简介

本节介绍 Node Packaged Modules（简称 NPM）的概述，以及 NPM 的安裝配置。同时介绍开发和运营的一些重要 Node.js 模块，其中包括 express 框架、request 模块、socket.io 模块、forever 模块，以及 jade 模版。如何自我开发和发布一个 Node.js 模块到 NPM 是本章的一个扩展知识点。本节的重点是介绍 Node.js 的重要模块的安裝配置，以及简单的入门知识。

2.2.1 NPM 和配置

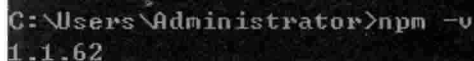
1. NPM 概述

Node Packaged Modules 简称 NPM¹，是 Node.js 的包管理器，也可以说是 Node.js 的一个库。Node.js 本身提供了一些基本的 API 模块，但这些基本的模块难以满足开发者的需求。例如开发者可能有多种数据库连接方式，但 Node.js 原生模块没有提供任何 API，因此 Node.js 需要使用 NPM 来管理开发者自我研发的一些模块，并使其能够共用于其他开发者。简单地说，NPM 就是 Node.js 的包管理器，里面的大部分应用模块，都是开发者提供的，例如 MySQL、jade、wind.js 等，开发者可以通过 NPM 提交个人 Node.js 模块成果，而其他开发者也可使用 NPM 下载这些 NPM 模块包，并将 NPM 模块应用到项目中。

¹ 参考网站 <https://npmjs.org/>。

2. NPM 配置

Windows 系统下使用以 msi 为后缀名的 Node.js 安装文件安装后，会默认安装 NPM 模块，通过 cmd dos 窗口输入 `npm -v` 查看是否安装 NPM，以及 NPM 版本信息，默认情况下是可以全局执行的，不需要添加 NPM 环境变量，如图 2-5 所示。



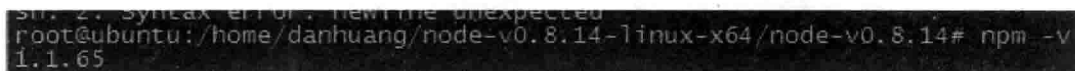
```
C:\Users\Administrator>npm -v
1.1.62
```

图 2-5 Windows 系统下查询 NPM 版本运行结果

Linux 系统下执行如下指令安装（Node.js 的 0.8 版本以上自带安装 NPM 工具）：

```
curl http://npmjs.org/install.sh | sudo sh
```


安装成功以后输入 `npm -v` 查看版本信息，以及检查是否安装成功，如图 2-6 所示。默认在 Linux 编译安装的 Node.js 是自带 NPM 工具的。



```
root@ubuntu:/home/danhuang/node-v0.8.14-linux-x64/node-v0.8.14# npm -v
1.1.65
```

图 2-6 Linux 系统下查看 NPM 版本运行结果

安装成功后，两个环境安装后的使用方法基本相同，可以通过 `npm - help` 查看 NPM 的指令使用方法，最常用的是 `npm install` 安装模块和 `npm uninstall` 卸载模块。

 **注意：**`npm install` 命令会在本目录下新增一个 `node_modules` 文件夹，并会在 `node_modules` 文件夹中添加 NPM 模块。

2.2.2 Express 框架

本节将介绍 Node.js 的项目从开发到项目运营可能需要的一些 NPM 模块，其中包括 Node.js 的项目框架 express 模块；socket.io 模块，使用 socket 协议处理长连接多请求的问题；forever 模块，来实现项目的运营、监控；jade 模板来处理 Node.js 无法内嵌 html 的问题；request 模块解决 Node.js 发起 HTTP 请求处理模块；Formidable 表单数据处理模块。

接下来将介绍 NPM 模块，在介绍这些模块前，首先看一下 NPM 模块安装指令和 NPM 模块的卸载指令。安装方法：

```
npm install module
```

卸载方法：

```
npm uninstall module
```

express 是一个 Node.js 的 Web 开源框架，一个非常适合 Node.js 入门学习的开发者，该框架可以快速搭建 Web 项目开发的框架。其主要集成了 Web 的 http 服务器的创建、静态文件管理、服务器 url 请求处理、GET 和 POST 请求分发、Session 处理等功能。

分享一些 express 的资源网站, Github 主页: <https://github.com/visionmedia/express>, 官网: <http://expressjs.com/>。

安装和配置方法和普通的 NPM 模块一样, 执行命令:

```
npm install -g express
```

参数-g是添加全局执行环境, 安装时可以设定安装的版本信息, 在express后添加@version即可, 例如:

```
npm install -g express@3.0
```

安装成功后, 简单地做一个 Node.js 应用, 来介绍如何利用 express 框架开发 Node.js 项目。进入任何目录, 执行命令:

```
express app
```

创建一个 app 应用。执行完成后, 可以看到在本路径下其创建了 app 项目, 该项目中有如图 2-7 所示的文件和文件夹。

```
C:\Users\Administrator\Desktop\code>express app

create : app
create : app/package.json
create : app/app.js
create : app/public
create : app/public/javascripts
create : app/public/images
create : app/public/stylesheets
create : app/public/stylesheets/style.css
create : app/routes
create : app/routes/index.js
create : app/routes/user.js
create : app/views
create : app/views/layout.jade
create : app/views/index.jade

install dependencies:
$ cd app && npm install

run the app:
$ node app
```

图 2-7 执行 express app 返回结果


cd 进入 app 文件夹, 运行 app.js 启动项目。

```
node app.js
```

执行 app.js 返回结果如图 2-8 所示, 表示安装成功。在浏览器中打开本地监听 IP 和端口 3000, <http://127.0.0.1:3000>, 看到如图 2-9 所示的页面, 表示已成功安装配置 express, 接下来就可以使用 express 框架开发项目了。

```
C:\Users\Administrator\Desktop\code\app>node app.js
Express server listening on port 3000
```

图 2-8 执行 app.js 返回结果

127.0.0.1:3000

Express

500 Error: Cannot find module 'jade'

图 2-9 浏览器打开 <http://127.0.0.1:3000> 返回页面

在项目开发中经常会遇到一些问题，如果该模块没有安装到 Node.js 的 `node_modules` 中时，应用 `express` 模块会出现 `Error: Cannot find module 'express'` 这个 error。解决办法是在本项目文件夹中执行，如下命令。

```
npm install express
```

安装成功后，会在本路径下生成一个 `node_modules`，里面包含了 `express` 框架代码。在应用其他 NPM 模块时，遇到类似无法找到一个 NPM 模块时可以手动将该模块包添加到 `node_modules` 文件夹，或者在本路径执行 `npm install express` 等，执行完成后会在本路径产生 `node_modules`，并且其中会存放需要安装的 NPM 模块。

2.2.3 jade 模板

`jade` 是一款高性能、简洁易懂的模板引擎¹，`jade` 是 `Hamlet` 的 JavaScript 实现，在服务端 (Node.js) 及客户端均有支持。本节主要介绍 `jade` 模板的安装配置和一些基本的语法，对于 `jade` 的用法可以参考其开源主页：<https://github.com/visionmedia/jade>。

为什么需要 `jade` 模板？Node.js 和 PHP、Java 不同，后两者都可以内嵌代码到 `html` 页面中，如下代码是实现一个内嵌 PHP 代码的例子，利用这点可以在 `html` 中应用 PHP 的一些变量。

```
<html>
<head></head>
<body>
  <!-- PHP 内嵌代码逻辑 -->
  <?php
    $name = 'danhuan';
    echo $name;
  ?>
</body>
</html>
```

上面的代码主要是介绍在 PHP 中通过内嵌来应用 PHP 的一些变量，或者 PHP 的一些条件判断、for 循环逻辑等。而对于 Node.js 来说，其没有类似的内嵌功能，因此在 Node.js

¹ 模板引擎（这里特指用于 Web 开发的模板引擎）是为了使用户界面与业务数据（内容）分离而产生的，它可以生成特定格式的文档，用于网站的模板引擎会生成一个标准的 HTML 文档。

中我们必须使用一些模板来处理。

对于一个优秀的 MVC 项目实现来说, PHP 和 Java 也应该使用模板完成 view 层页面的显示, 而不是内嵌, 当然内嵌代码是允许的。但对 Node.js 来说, 其不支持内嵌, 因此模板就显得非常重要。

jade 模板的安装配置方法如同 NPM 其他模块一样, 执行如下指令:

```
npm install jade
```

运行该指令的执行结果如图 2-10 所示。

```
jade@0.27.7 /usr/local/lib/node_modules/jade
├── commander@0.6.1
├── mkdirp@0.3.4
├── coffee-script@1.4.0
root@ubuntu: /home/danhuang#
```

图 2-10 运行 npm install jade 执行结果

注意: NPM 模块安装时会在当前文件夹中产生 node_modules 目录, 并在该目录中下载 NPM 模块。而 Node.js 项目运行 require 一个模块时, 会自动地在当前目录下的 node_modules 目录中加载所需的 NPM 模块, 因此也可以直接去 github 下载相应的模块, 并将其放入项目中的 node_modules 文件夹。

从图 2-10 可知已成功安装 jade 模板, 安装成功后将本文件夹中 node_modules 中的 jade 文件夹拷贝到上一节 express 应用项目 app 文件夹中的 node_modules 中, 该 node_modules 中包含了刚才安装的两个 NPM 模块包, express 和 jade, 如图 2-11 所示。

```
drwxr-xr-x 5 root root 4096 118 22:22 /
drwxr-xr-x 6 root root 4096 118 22:20 /
drwxr-xr-x 2 root root 4096 118 22:22 /
drwxr-xr-x 5 root root 4096 118 22:20 /
drwxr-xr-x 5 root root 4096 118 22:22 /
root@ubuntu: /home/danhuang/app/node_modules#
```

图 2-11 查看当前项目 node_modules 文件夹返回结果

继续执行 express 框架介绍时创建的例子, 重新运行 express 框架的 app 应用, 执行如下指令:

```
node app.js
```

```
← → ↺ 192.168.1.120:3000
```

Express

Welcome to Express

图 2-12 浏览 http://127.0.0.1:3000 返回 Web 页面

运行成功后，再次打开 `http://127.0.0.1:3000` 后，返回显示如图 2-12 所示，这样页面就没有提示刚才的错误信息 `can not find jade module error`。接下来主要是查看 `app` 应用中的 `jade` 模板文件 `index.jade` (`app/views/index.jade`) 代码。

```
extends layout

block content
  h1= title
  p Welcome to #{title}
```

第一行代码是 `extends layout` 这个 `jade` 模板，相当于在 `index.jade` 页面前加载 `layout.jade` 页面，这里的 `extends` 是 `jade` 模板的一个关键词，该关键词相当于 `Node.js` 中的 `require` 关键词。`h1` 和 `p` 分别表示 `html` 对应的标签名 `<h1></h1>` 和 `<p></p>`，其后的值是标签内的 `DOM` 元素。注意，上面代码中的 `#{title}`，这个 `title` 是服务器 `Node.js` 传递的参数，可以查看 `Node.js` 的源代码 `app/routes/index.js`。

```
/*
 * GET home page.
 */
exports.index = function(req, res){
  res.render('index', { title: 'Express' });
};
```

其中可以看到，`jade` 提供了 `render` API，该 API 指定显示的 `jade` 模板名，以及向该模板中传递的参数值，例如本例子的 `title` 变量值。而参数在 `jade` 模板中的调用方法是 `#{params}`。当然可以将 `title` 的值改为一个 `json` 或者其他格式的数据类型。

`jade` 模板中提供了灵活的设置 `html` 标签，以及 `DOM` 元素的 `id` 和 `class` 值，同时其提供了 `if` 条件语句、`for` 循环执行语句等，官网文档中有详细的说明和例子。

下面我们来看 `jade` 中的 `if` 条件语句、`for` 循环语句的例子。

1. if 条件判断语法

```
if ${name} == 'admin'
  p this is an admin
else
  p this is not an admin
```

2. for 循环语法

1) 应用 each:

```
var items = ["one", "two", "three"]
each item in items
  li= item
```

2) 应用 for 循环:

```
for user in users
  for role in user.roles
    li= role
```

以上代码还是比较容易理解的，其中需要注意，`for` 的第二种循环中的变量无需符号 `{}`，而是直接使用变量名。

2.2.4 forever 模块

Node.js 作为 HTTP 服务器，需要确保项目正常运行，要注意以下两点：

- ❑ 后台服务运行，监控运行日志，以及 HTTP 请求日志。
- ❑ 确保项目的正常安全运行。Node.js 的启动命令 `node`，很大程度上无法满足运行需求。

Node.js 的 `forever` 模块在这点上就可以起到很大的作用，同时其拥有监控文件更改、自动重启等其他功能。

`forever` 有两种使用方法，一种是在命令行中使用，另外一种是在 Node.js 的编码中 `require forever` 模块使用，本书只介绍 `forever` 在命令行中部分功能的使用方法。`forever` 安装方法如下：

```
npm install forever -g
```

使用 `forever --help` 查看是否安装成功，同时查看 `forever` 运行指令。如图 2-13 所示为执行 `forever -help` 后的返回结果。

```
root@ubuntu: ~/home/danhuang/app/routes# forever -help
usage: forever [action] [options] SCRIPT [script-options]

Monitors the script specified in the current process or as a daemon

actions:
  start      Start SCRIPT as a daemon
  stop       Stop the daemon SCRIPT
  stopall    Stop all running forever scripts
  restart    Restart the daemon SCRIPT
  restartall Restart all running forever scripts
  list       List all running forever scripts
  config     Lists all forever user configuration
  set <key> <val> Sets the specified forever config <key>
  clear <key> Clears the specified forever config <key>
  logs       Lists log files for all forever processes
  logs <script/index> Tails the logs for <script/index>
  columns add <col> Adds the specified column to the output in 'forever list'
  columns rm <col> Removed the specified column from the output in 'forever list'
  columns set <cols> Set all columns for the output in 'forever list'
  cleanlogs  [CAREFUL] Deletes all historical forever log files
```

图 2-13 执行 `forever -help` 返回结果

借助 `express` 的框架 `app` 例子，接下来使用 `forever` 运行该 `express` 项目，运行指令如下：

```
forever start -l forever.log -o out.log -e err.log app.js
```

`-l` 指定 `forever` 运行日志，`-o` 指定脚本流水日志，`-e` 指定脚本运行错误日志。启动后将会在本文件夹产生 `out.log`、`err.log` 文件，运行成功后，使用如下指令查看进程是否启动运行：

```
netstat -nap|grep node
```

使用 `netstat` 查看 `node` 进程返回结果如图 2-14 所示，表明已正常启动运行。

```
root@ubuntu: ~/home/danhuang/app# netstat -nap|grep node
tcp        0      0 0.0.0.0:0.0.0.0:3000 0.0.0.0:*        LISTEN    1219/
unix  2      [ ACC ]     5        LISTENING   8941      1219/      root/.forever/sock/worker.1332474500172haq.sock
```

图 2-14 使用 `netstat` 查看 `node` 进程返回结果

`forever` 其他的功能还包括监控脚本内容改动后自动重启 Node.js 服务器，该功能是很

多开发调式的好工具。关于该功能的使用以及配置方法可以参考 `forever` 的开源官网 [<https://github.com/nodejitsu/forever>]，其中有详细的介绍。

在 Windows 下安装后，启动 `forever` 时出现 `C:\root\forever\9Rah.log` 无法创建的问题时，只需要在 C 盘下创建 `root` 文件夹即可，问题原因是在 Windows 的 C 盘中其无权限创建 `root` 文件夹。为了让读者更好地学习利用 `forever` 工具，如下提供了一份来自 `forever` 官网的命令行工具文档¹。

```
$ forever --help
usage: forever [action] [options] SCRIPT [script-options]

Monitors the script specified in the current process or as a daemon

actions:
  start           Start SCRIPT as a daemon
  stop            Stop the daemon SCRIPT
  stopall         Stop all running forever scripts
  restart         Restart the daemon SCRIPT
  restartall      Restart all running forever scripts
  list            List all running forever scripts
  config          Lists all forever user configuration
  set <key> <val> Sets the specified forever config <key>
  clear <key>     Clears the specified forever config <key>
  logs            Lists log files for all forever processes
  logs <script|index> Tails the logs for <script|index>
  columns add <col> Adds the specified column to the output in 'forever list'
  columns rm <col> Removed the specified column from the output in 'forever list'
  columns set <cols> Set all columns for the output in 'forever list'
  cleanlogs       [CAREFUL] Deletes all historical forever log files

options:
  -m MAX          Only run the specified script MAX times
  -l LOGFILE       Logs the forever output to LOGFILE
  -o OUTFILE       Logs stdout from child script to OUTFILE
  -e ERRFILE       Logs stderr from child script to ERRFILE
  -p PATH          Base path for all forever related files (pid files, etc.)
  -c COMMAND       COMMAND to execute (defaults to node)
  -a, --append     Append logs
  -f, --fifo       Stream logs to stdout
  -n, --number     Number of log lines to print
  --pidFile        The pid file
  --sourceDir      The source directory for which SCRIPT is relative to
  --minUptime      Minimum uptime (millis) for a script to not be considered
                    "spinning"
  --spinSleepTime  Time to wait (millis) between launches of a spinning
                    script.
  --plain          Disable command line colors
  -d, --debug      Forces forever to log debug output
  -v, --verbose    Turns on the verbose messages from Forever
  -s, --silent     Run the child script silencing stdout and stderr
  -w, --watch      Watch for file changes
  --watchDirectory Top-level directory to watch from
  -h, --help       You're staring at it

[Long Running Process]
The forever process will continue to run outputting log messages to the
console.
```

1 参考网站 <https://github.com/nodejitsu/forever>。

```
ex. forever -o out.log -e err.log my-script.js

[Daemon]
The forever process will run as a daemon which will make the target process
start
in the background. This is extremely useful for remote starting simple
node.js scripts
without using nohup. It is recommended to run start with -o -l, & -e.
ex. forever start -l forever.log -o out.log -e err.log my-daemon.js
forever stop my-daemon.js
```

2.2.5 socket.io 模块

socket.io 是一个基于 Node.js 的项目,其作用主要是将 WebSocket 协议应用到所有的浏览器。该模块主要应用于实时的长连接多请求项目中,例如在线联网游戏、实时聊天、实时股票查看、二维码扫描登录等。

socket.io 的开源代码主页: <https://github.com/LearnBoost/socket.io>, 以及其官方网站: <http://socket.io/>。安装和配置方法和一般 NPM 模块安装配置一致:

```
npm install socket.io
```

安装成功后,来学习如何使用 socket.io 创建项目。socket 协议首先要了解其存在服务器端和客户端,因此要实现一个 socket 服务时,根据 socket 的服务器端和客户端 API 要分别实现其逻辑。服务器端启动进程等待客户端的连接,那么接下来就先创建一个服务器端 Node.js 脚本 index_server.js 代码。

```
var io = require('socket.io').listen(80);
/* 定义 socket 连接时执行的回调操作 */
io.sockets.on('connection', function (socket) {
  socket.emit('news', { hello: 'world' });
  socket.on('my other event', function (data) {
    console.log(data);
  });
});
```

【代码说明】

- ❑ `var io = require('socket.io').listen(80);`: 设置 socket 监听端口为 80 端口。
- ❑ `io.sockets.on('connection',function({}))`: 调用 socket API 中的 `sockets.on` 接口,当客户端 connection 时,执行回调函数 `function(socket){}`。
- ❑ `socket.emit('news',{ hello: 'world' });`: 连接成功后发送一个 news 消息,消息内容为 json 对象 `{hello: 'world'}`。
- ❑ `socket.on('my other event', function (data) {})`: 客户端发送 my other event 消息时,服务器端接受该消息,成功获取该消息后执行回调函数 `function(data){}`。

`socket.emit()` 为 socket 发送消息函数,其第一个参数为发送消息的 key 值,第二个参数为发送消息的内容,也就是发送的数据。`socket.on` 为 socket 接受消息函数,同理其第一个参数为接受消息的 key 值,第二个参数为回调函数,其中回调函数携带的参数为接受消息所发送的数据。

接下来查看在 Web 前端如何通过 JavaScript 来连接 socket 服务器。新建 index_client.html

文件，代码如下：

```
<script src="socket.js"></script>
<script>
  var socket = io.connect('http://localhost'); //创建本地 socket 连接
  socket.on('news', function (data) { //socket 接受 news 消息时执行回调函数
    console.log(data);
    socket.emit('my other event', { my: 'data' });
  });
</script>
```

【代码说明】

- ❑ `<script src="socket.js"></script>`：加载 socket.js 本地 JavaScript 文件；
- ❑ `var socket = io.connect('http://localhost')`：使用 socket 连接本地 socket 服务器，默认为 80 端口，因此无需填写端口号；
- ❑ `socket.on('news', function (data) {})`：接受到服务器端发送的 news 消息后，当服务器推送 news 消息后执行回调函数，回调函数的 data 参数为 news 消息发送的数据，如上面的 json 对象 { hello: 'world' }；
- ❑ `socket.emit('my other event', { my: 'data' })`：客户端接受 news 消息成功后，发送 my other event 消息到服务器端，其发送的消息内容为 json 对象 { my: 'data' }。

服务端 socket 启动运行成功后，浏览器（这里的浏览器相当于 socket 服务客户端）主动请求 socket 连接到 localhost，由于服务器监听的是 80 端口，因此这里可以省略端口号，如果服务器监听的是 8080 端口或者其他，那这里必须加上端口号，socket 连接可以修改为如下代码：

```
var socket = io.connect('http://localhost:8080');
```

运行服务器端的 index_server.js 脚本文件，来启动 socket 服务。

```
node index_server.js
```

启动成功后会输出 info-socket.io started 的 log 信息。使用浏览器直接打开 index_client.html，通过 JavaScript 客户端主动连接 socket 服务器。打开完成后服务器端将会接收到连接请求，因此会执行 io.sockets.on('connection',function(){})函数，当连接成功后，服务器端会执行 socket.emit 来发送 news 消息到客户端，同时客户端接收消息后，也同样通过 socket.emit 返回 my other event 消息内容，并将消息内容输出，同时可以在运行窗口查看服务器端运行日志，如图 2-15 所示。

```
info - socket.io started
debug - client authorized
info - handshake authorized qfIzFu_pP8n2ykojIdJv
debug - setting request GET /socket.io/1/websocket/qfIzFu_pP8n2ykojIdJv
debug - set heartbeat interval for client qfIzFu_pP8n2ykojIdJv
debug - client authorized for
debug - websocket writing 1::
debug - websocket writing 5::{"name":"news","args":[{"hello":"world"}]}
< my: 'data' >
```

图 2-15 socket.io 启动后 debug 日志

从图中可以查看 socket 的 debug 日志，包括请求连接以及连接成功后数据的交互。在

图中最后一行的 debug 日志中可以看到,连接成功后服务器端发送了一个 news 消息到客户端,客户端接收消息后发送一个 my other event 消息到服务器端,在图 2-15 中我们可以看到服务器端发送的数据和客户端输出的消息内容是一致的,都为 {my: 'data'}, 同样客户端接收到的数据是一个消息,消息的内容是一个 json 对象,其 key 为 hello 值,为 world,和服务器端发送的数据是一致的,如图 2-16 所示是客户端接收 news 消息后返回的数据。

```
▼ Object
  hello: "world"
  ► __proto__: Object
>
```

图 2-16 socket 客户端接受数据返回字段

上图是使用 Chrome 的 debug 工具显示。通过 socket 的客户端代码和服务端代码的举例说明,可以简单的实现了 socket 服务。

如何关闭 debug 和 info 日志呢?看到上面如此多的 debug 和 info 数据时,如果希望将其隐藏,只需要将 debug 日志修改或者用直接关闭的方式。

```
var io = require('socket.io').listen(80,{log:false});
```

在监听本地端口时,将 log 设置为 false 即可将日志关闭。也可以将 debug 和 info 日志级别降低。

```
io.set('log level', 1);
```


socket.io 其他设置可参考其官方文档¹。发现 warn - error raised: Error: listen EADDRINUSE 错误怎么办?启动 socket.io 时可能会出现异常 warn - error raised: Error: listen EADDRINUSE,错误原因是监听的端口被占用,换一个监听端口即可。

2.2.6 request 模块

request 模块是由 mikeal²在 github 发布,request 模块为 Node.js 开发者提供了一种简单访问 HTTP 请求的方法。request 还支持 HTTPS 的访问方法,这里不做详细介绍。安装该模块的方法如下:

```
npm install request
```

request 模块基本上覆盖了所有的 HTTP 请求方式如 GET、POST、HEAD、DEL 等,但最基本的两个方法分别是 request.get 和 request.post,这两个 API 是本节介绍的重点,前者是发起一个 HTTP 的 GET 请求,后者发起一个 HTTP 的 POST 请求,下面介绍 HTTP 中 GET 和 POST 之间的区别。

 **注意:** 根据 HTTP 规范,GET 用于信息获取,而且应该是安全的和幂等的,POST 表示可能修改服务器上的资源的请求。GET 请求的数据会附在 url 之后(就是把数据

1 网站 <https://github.com/LearnBoost/Socket.IO/wiki/Configuring-Socket.IO>。

2 参考网站 <https://github.com/mikeal/request>。

放置在 HTTP 协议头中)，以?分割 url 和传输数据，参数之间以&相连，如 login.action?name=hyddd&password=idontknow&verify=%E4%BD%A0%E5%A5%BD。如果数据是英文字母/数字，原样发送，如果是空格，转换为+，如果是中文/其他字符，则直接把字符串用 BASE64 加密，得出如%E4%BD%A0%E5%A5%BD，其中%XX 中的 XX 为该符号以十六进制表示的 ASCII。GET 方式提交的数据最多只能是 1024 字节，理论上 POST 没有限制，可上传较大量的数据！其次 POST 的安全性要比 GET 的安全性高，GET 方式会将请求参数暴露在 url 上。总结一下，GET 是向服务器索取数据的一种请求，而 POST 是向服务器提交数据的一种请求，在 FORM（表单）中，Method 默认为“GET”，实质上，GET 和 POST 只是发送机制不同，并不是一个取一个发！¹

接下来我们就来看一下 request.get 和 request.post 两个 API 的简单实例介绍。首先应用 Node.js 的 HTTP 模块创建两个服务器，一个是处理 GET 请求，另外一个则是处理 POST 请求，GET 服务器相关代码 app_get.js 如下：

```
var http = require('http');
/* 创建 HTTP 服务器 */
http.createServer(function(req, res) {
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('Hello World\n' + req.method);
}).listen(1337, "127.0.0.1");
console.log('Server running at http://127.0.0.1:1337/');
```

该服务器创建一个 HTTP 服务器，并返回字符 Hello World，同时将 req.method 打印客户端请求方式。然后通过 request 模块去请求该服务器数据，并将服务器返回结果打印显示，request_get.js 代码如下：

```
var request = require('request');
/* 应用 request 方法，获取 http://127.0.0.1:1337 的响应信息 */
request.get('http://127.0.0.1:1337', function(error, response, result){
    console.log(result);
});
```

应用 request 模块的 GET 方法，发起一个 HTTP 请求，请求本地 http://127.0.0.1:1337 服务器数据，request.get 的两个参数分别是请求 url 和回调函数。

接下来运行 app_get.js 代码，可以看到 server 启动消息，服务器启动成功后，我们再执行 request_get.js 脚本，依次执行如下指令：

```
node app_get.js
node request_get.js
```

执行完 request_get.js 以后，可以在命令窗口得到如图 2-17 所示的返回结果。

```
K:\桌面\code\chapter_two\npm\request>node request_get.js
Hello World
GET
```

图 2-17 运行 request_get.js 返回结果

1 参考网站 <http://www.cnblogs.com/hydd/archive/2009/03/31/1426026.html>。

从图中可以看到 request GET 请求返回了 Hello World, 并将该请求方式 GET 打印返回到客户端。

request.post 的应用和 request.get 类似。首先创建一个 HTTP 服务器, 该服务器接收客户端的 POST 参数, 并将该参数作为字符串响应到客户端, app_post.js 相关代码如下:

```
var http = require('http'),
    querystring = require('querystring'); //require querystring 模块
http.createServer(function(req, res) {
    var postData = "";
    /* 开始异步接收客户端 post 的数据 */
    req.addListener("data", function(postDataChunk) {
        postData += postDataChunk;
    });
    /* 异步 post 数据接收完成后执行匿名回调函数 */
    req.addListener("end", function() {
        var postStr = JSON.stringify(querystring.parse(postData));
        res.writeHead(200, { 'Content-Type': 'text/plain' });
        res.end(postStr+'\n' + req.method);
    });
}).listen(1400, "127.0.0.1");
console.log('Server running at http://127.0.0.1:1400/');
```

【代码说明】

- ❑ req.addListener: 获取 POST 数据;
- ❑ JSON.parse(querystring.parse(postData)): 解析客户端发送的 POST 数据, 并将其转化为字符;
- ❑ res.end(postStr): 响应客户端请求的 POST 数据。

以上代码创建了一个 HTTP 服务器并监听本地 1400 端口, 接下来应用 request POST 方法来访问 http://127.0.0.1:1400, request_post.js 代码如下:

```
var request = require('request');
/* 应用 request 方法, post 到 http://127.0.0.1:1400 的, 并获取响应信息 */
request.post('http://127.0.0.1:1400', {form:{'name':'danhuang',
'book':'node.js'}},function(error, response, result){
    console.log(result);
});
```

应用 request 发起 HTTP 的 POST 请求, 并通过表单数据 {'name':'danhuang', 'book':'node.js'}, 注意这里的 POST 的参数需要使用 form 作为键值。分别执行如下指令:

```
node app_post.js
node request_post.js
```

运行 request_post.js 结束后, 可以看到服务器的响应信息, 如图 2-18 所示。

```
K:\桌面\code\chapter_two\npm\request>node request_post.js
{"name":"danhuang","book":"node.js"}
POST
```

图 2-18 运行 request_post.js 返回结果

request_post.js 应用 request post 提交了一个 json 对象 {'name':'danhuang', 'book':'node.js'}, 而 HTTP 服务器通过获取该 POST 数据, 然后返回到客户端, 同时将 HTTP 的请求方式也

响应到客户端。

`request.post` 参数可以有两种传递方法。其中，第一种是将 `url` 和 `FORM` 表单数据作为一个 `json` 参数在 `request.post` 传递。

```
request.post({'url':'url',form:{}},function(error, response, result){});
```

举例如下：

```
request.post({'url':'http://127.0.0.1:1400',form:{'name':'danhuang',  
'book':'node.js'}},function(error, response, result){  
  console.log(result);  
});
```

另外一种则是将 `url` 和 `form` 作为两个参数。

```
request.post('url',{'form':{}},function(error, response, result){});
```

举例如下：

```
request.post('http://127.0.0.1:1400',{'form':{'name':'danhuang',  
'book':'node.js'}},function(error, response, result){  
  console.log(result);  
});
```

2.2.7 Formidable 模块

`Formidable` 模块由 `Transloadit`¹ 开发，该模块的目的是为了解决文件的上传。

其主要特性功能：

```
Fast (~500mb/sec), non-buffering multipart parser  
Automatically writing file uploads to disk  
Low memory footprint  
Graceful error handling  
Very high test coverage2
```

安装该模块的方法：

```
npm install formidable
```

在原生的 `Node.js` 模块中，提供了获取 `POST` 数据的方法，但是没有直接获取上传文件方法。因此需要利用 `Formidable` 模块来处理文件上传逻辑。接下来我们看一下 `Formidable` 提供的一个实例。首先 `require` 该模块，并 `new` 一个 `form` 对象，如下代码所示。

```
var formidable = require('formidable');  
var form = new formidable.IncomingForm();
```

为了避免传递数据的字符乱码，可以先设置接收数据的字符编码，例如如下设置为 `utf-8`。`form.encoding = 'utf-8'`；最后可以根据 `Formidable` 提供的 `parse` 方法解析 `POST` 数据，例如：

```
form.parse(req, function(err, fields, files) {  
  res.writeHead(200, {'content-type': 'text/plain'});
```

¹ 参考网站 <https://transloadit.com/>。

² 摘自 `github` 主页：<https://github.com/felixge/node-formidable>。

```
res.write('received upload:\n\n');
res.end(util.inspect({fields: fields, files: files}));
});
```

fields 为 POST 数据, files 为文件对象, 其中包含了上传的文件相关信息。接下来我们看一下该模块的应用实例。该应用是创建一个 HTTP 服务器, 并根据用户请求路径判断显示 html 页面还是提交表单。

```
var formidable = require('formidable'), //获取 formidable 模块对象
    http = require('http'),
    util = require('util');
/* 应用 HTTP 的 createServer 方法创建服务器 */
http.createServer(function(req, res) {
  if (req.url == '/upload' && req.method.toLowerCase() == 'post') {
    //parse a file upload
    var form = new formidable.IncomingForm();

    form.parse(req, function(err, fields, files) {
      //应用 form 对象解析并获取 http 的参数
      res.writeHead(200, {'content-type': 'text/plain'});
      res.write('received upload:\n\n');
      res.end(util.inspect({fields: fields, files: files}));
    });

    return;
  }

  //show a file upload form
  res.writeHead(200, {'content-type': 'text/html'});
  /* HTTP 响应 html 信息 */
  res.end(
    '<form action="/upload" enctype="multipart/form-data" method="post">'+
    '<input type="text" name="title"><br>'+
    '<input type="file" name="upload" multiple="multiple"><br>'+
    '<input type="submit" value="Upload">'+
    '</form>'
  );
}).listen(8080);
```

【代码说明】

- ❑ req.url == '/upload': 判断请求路径是否为 upload, 如果是则执行文件上传处理逻辑。
- ❑ req.method.toLowerCase() == 'post': 判断 HTTP 请求方式是否为 POST。
- ❑ var form = new formidable.IncomingForm(): 创建 form 对象。
- ❑ form.parse(req, function(err, fields, files): 解析 POST 数据。
- ❑ util.inspect({fields: fields, files: files}): 将 json 对象转化为字符串。

接下来我们运行该段代码, 打开浏览器输入 <http://127.0.0.1:8080/>, 可以在浏览器中看到如图 2-19 所示的页面。

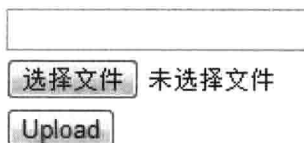


图 2-19 浏览 <http://127.0.0.1:8080/> 返回 Web 页面

输入相应的 title, 例如 test image, 并上传图片后, 单击 Upload 按钮, 可以看到如图 2-20 所示的返回信息。

```
received upload:
{ fields: { title: 'test image' },
  files:
    { upload:
      { size: 28093,
        path: 'C:\\Users\\ADMINI~1\\AppData\\Local\\Temp\\357836e6fa8ac57c822fb996a2fdb27f',
        name: '612edf3ajwle0pvvb8gesj.jpg',
        type: 'image/jpeg',
        hash: false,
        lastModifiedDate: Sun Mar 17 2013 15:26:37 GMT+0800 (中国标准时间),
        _writeStream: [Object] } } }
```

图 2-20 执行文件上传后返回 Web 页面

从图中可以看到 fields 的 POST 值, title 为 test image, 而 files 则为文件信息。这里需要注意的是, 应用该模块后, 原生的获取 POST 数据方法会有影响。

以上就是本节介绍的所有 NPM 模块, 这些模块在后续章节中都会经常使用到, 读者在学习这些模块时, 要多实践, 多了解。其中还有更多的模块信息, 大家可以前往 <https://npmjs.org/> 学习了解。

2.2.8 NPM 模块开发指南

NPM 的一些重要应用模块已经介绍了, 接下来介绍如何自我开发 NPM 模块, 提供他人学习参考和应用交流。笔者的个人博客中有一篇《How to create new module in npm》里面详细描述了如何创建 NPM 模块的过程。

这里同样借助上面讲过的 app 项目代码来说明, 该 app 在项目根目录下创建两个配置文件 package.json 和 README.md。package.json 为项目相关信息, 包括项目名称、版本、描述、执行文件、作者、搜索关键字、版本依赖库、源码地址和启动项目的 Node.js 版本等, 而 README.md 则为模块解析文档。例如下面的 package.json:

```
{
  "name": "myweb",
  "version": "0.0.1",
  "description": "Node.js module to do some thing",
  "preferGlobal": "true",
  "main": "index.js",
  "bin": { "myweb": "index.js" },
  "author": "Danhuang",
  "keywords": ["myweb", "nodejs", "nodejs framework"],
  "repository": {
    "type": "git",
    "url": "https://tnodejs@github.com/tnodejs/myweb-nodejs.git"
  },
  "dependencies": {
```

```
    "commander": "0.5.2"
  },
  "engines": { "node": "*" }
}
```

【代码说明】

- ❑ name: NPM 模块名称。
- ❑ version: 版本号。
- ❑ description: 模块描述。
- ❑ preferGlobal 和 bin: 设置是否使用命令行功能。
- ❑ main: 项目入口文件。
- ❑ author: 作者。
- ❑ keywords: NPM 模块搜索关键字。
- ❑ repository: 代码库, type 获取方式, url 为 github 源码位置, [https://username@github.com/username/ project](https://username@github.com/username/project)。
- ❑ engines: 项目依赖 Node.js 版本。

上面就是发布一个 NPM 模块时, 需要填写 package.json 的所有配置信息, 配置完成 package.json 参数后, 再为该模块添加相应的说明。README.md 为 NPM 模块, 以及 NPM 模块安装配置使用引导。第二步进入 app 项目根目录, 运行指令:

```
npm link
```

存在权限问题时请使用:

```
sudo npm link
```

执行完成后会在 app 文件夹生成一个 node_modules, 这个 node_modules 不是我们发布项目的必需文件夹, 因此需要将其排除, 执行如下指令:

```
echo node_modules/ >> .gitignore
```

执行完成后, 在 NPM 模块管理中注册新的用户:

```
npm adduser
```

执行以上代码后, 需要输入一些注册的基本信息, 按照要求填写, 完成后, 发布 NPM 项目模块。

```
npm publish
```

完成这些后就简单地实现了一个 NPM 模块的发布, 接下来进入 <https://npmjs.org>, 使用刚才注册的用户名和密码登录, 成功后会看到类似图 2-21 所示的页面信息, 表示成功发布该 NPM 模块, 也可通过设置的关键字搜索发布的 NPM 模块。

也可以使用 `npm install app_name` 测试是否可以成功安装该模块。本节没有实践的题目, 希望大家可以尝试使用刚才的过程发布一个简单测试 NPM 模块。介绍本节内容的目的是希望更多的开发者能够更好地完善 Node.js 的 NPM 模块, 让 Node.js 越来越强大。

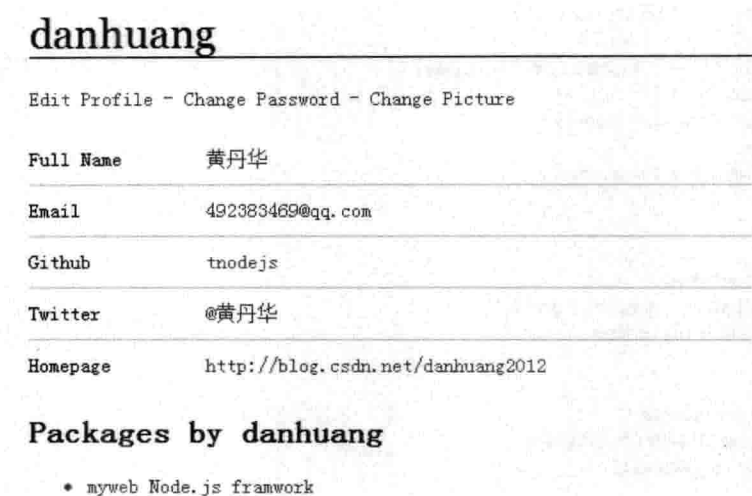


图 2-21 成功创建 NPM 模块主页信息

2.3 Node.js 设计模式

Node.js 的语法和一些常用的解释性语言（如 PHP）和编译性语言（如 Java）等都有很大的不同之处，Node.js 的编程语法和 JavaScript 一致，也可以说其就是 JavaScript，因此 Node.js 的类的概念和其他的解析性语法，如 PHP 和 Java 来说有点区别。本节将首先介绍 Node.js 中类和模块的概念，然后介绍 Node.js 中如何实现继承关系，最后介绍 Node.js 的一些设计模式。

本节的学习重点是了解 Node.js 中对象、类和模块的区别与联系，以及类之间的继承关系实现方法。

2.3.1 模块与类

模块是程序设计中，为完成某一功能所需的一段程序或子程序；或指能由编译程序、装配程序等处理的独立程序单位；或指大型软件系统的一部分。而在 Node.js 中可以理解为完成某一功能所需的程序或子程序，同时也可以将 Node.js 的一个模块理解为一个“类”，但注意，其本身并非是类，而只是简单意义上的一个对象，该对象拥有多个方法和属性，Node.js 的模块也拥有私有成员和公有成员。例如下面代码所示。

```
/* module.js */
/**
 * Created by JetBrains WebStorm.
 * User: danhuang
 */
//public parameter
exports.name = 'danhuang';
//private parameter
```

```

var myName = 'idanhuang';

exports.init = function(itName){
    if(!itName){
        setName(myName);
    } else {
        setName(itName);
    }
}

//public method
exports.show = function(){
    console.log(name);
}

//private method
function setName(myName){
    name = myName;
}

```

【代码说明】

- ☐ exports.name: 模块的公有属性 name。
- ☐ var myName: 模块的私有属性 myName。
- ☐ exports.init: 模块公有方法 init。
- ☐ exports.show: 模块公有方法 show。
- ☐ function setName(myName): 模块私有方法 setName。

现在大家对模块的公有和私有方法应该都有一定的认识，那么请大家看上面代码，不查看代码说明，能否指出其中“类”中的私有成员和公有成员？

首先分析对 Node.js 的模块来说私有成员和公有成员是如何区分的。一般来说 exports 和 module.exports 的成员为公有成员，而非 exports 和 module.exports 的成员为私有成员，因此上述代码中 name 变量、init 和 show 方法为公有成员，其他则为私有成员。接下来使用对象调用来验证上述结论。

```

/* show.js */
var Person = require('./module');
/* 输出 Person 对象中的 name 属性 */
console.log(Person.name);
/* 调用 Person 中的 init 方法初始化数据 */
Person.init('Node.js');
/* 调用 Person 中的 show 方法 */
Person.show();
/* 输出 Person 对象中的 myName 属性 */
console.log(Person.myName);

```

获取 Person 对象，输出 Person 对象的公有成员 name，调用公有方法 init 和 show，然后测试其私有成员 myName 是否可调用。执行 show.js 后，返回结果如图 2-22 所示。

```

root@ubuntu:/home/danhuang/module_and_class# node show.js
danhuang
Node.js
undefined

```

图 2-22 执行 show.js 脚本返回结果

结果中第一行输出的是公有变量 `name`，其次是执行 `init` 初始化和 `show` 方法的输出结果，最后测试 `Person` 的私有变量是否可显。从结果看其与类有点相似，但在这里再强调一次，`Node.js` 的模块不是类。

2.3.2 Node.js 中的继承

继承的方式主要是通过 `Node.js` 的 `util` 模块 `inherits` API 来实现继承 (`util.inherits(constructor, superConstructor)`)，将一个构造函数的原型方法继承到另一个构造函数中。`constructor` 构造函数的原型将被设置为使用 `superConstructor` 构造函数所创建的一个新对象。可以查看官网提供的例子，其目的是使用 `MyStream` 继承 `events.EventEmitter` 对象。代码如下：

```
var util = require("util");
var events = require("events");

function MyStream() {
  events.EventEmitter.call(this);
}
/* 应用 inherits 来实现 MyStream 继承 EventEmitter */
util.inherits(MyStream, events.EventEmitter);

MyStream.prototype.write = function(data) {
  this.emit("data", data);
}

var stream = new MyStream();

console.log(stream instanceof events.EventEmitter); //true
console.log(MyStream.super_ === events.EventEmitter); //true

stream.on("data", function(data) {
  console.log('Received data: ' + data + '');
})
stream.write("It works!"); //Received data: "It works!"
```

【代码说明】

- ❑ `var util = require("util") events = require("events");`: 获取 `util` 和 `event` 模块；
- ❑ `function MyStream(){}:` 目的是使用 `MyStream` 来继承 `events.EventEmitter` 的方法属性；
- ❑ `MyStream.prototype.write(){}:` 为 `MyStream` 类添加方法；
- ❑ `var stream = new MyStream():` 创建 `MyStream` 对象；
- ❑ `console.log(stream instanceof events.EventEmitter)`: 判断是否继承 `events.EventEmitter`；
- ❑ `console.log(MyStream.super_ === events.EventEmitter)`: 通过 `super_` 获取父类对象；
- ❑ `stream.on():` 调用继承来自 `events.EventEmitter` 的方法。

举例：学生、老师、程序员继承人这个类，实现学生学习、老师教书、程序员写代码。首先创建一个 `Person` 基类作为人。

```

/**
 *
 * class for the person
 * param: name
 * method: sleep, eat
 */
module.exports = function() {
  this.name = 'person';
  /* 定义 sleep 方法 */
  this.sleep = function() {
    console.log('sleep in the night');
  }
  /* 定义 eat 方法 */
  this.eat = function() {
    console.log('eat food');
  }
}

```

【代码说明】

- ❑ `this.sleep = function()`: 设置 `sleep` 方法。
- ❑ `this.eat = function()`: 设置 `eat` 方法。

这里涉及 `module.exports` 的使用。注意，为什么这里不用 `exports` 而使用 `module.exports`？主要的原因在第 2 章提及到，`exports` 返回的是一个 `json` 对象，而 `module.exports` 可以返回任何形式的数据格式，而这里需要 `require` 返回的是一个 `Class`，因此必须使用 `module.exports` 而不使用 `exports`。接下来我们看一个 `student` 的类如何实现继承，以及为 `student` 添加自我的方法 `study`。代码如下：

```

var util = require("util");
var Person = require('./person');
/**
 *
 * @desc 定义 Student 类
 */
function Student() {
  Person.call(this);
}
/* 将 Student 继承 Person */
util.inherits(Student, Person);
/* 重写 study 方法 */
Student.prototype.study = function() {
  console.log('I am learning');
}

/* 暴露 Student 类 */
module.exports = Student;

```

【代码说明】

- ❑ `var util = require("util")`: 获取 `util` 模块对象。
- ❑ `Function Student()`: 学生 `Student` 类。
- ❑ `util.inherits(Student, Person)`: 设置 `Student` 继承 `Person`。
- ❑ `Student.prototype.study`: 为 `student` 类添加 `study` 方法。
- ❑ `module.exports = Student`: `exports` 暴露出 `Student` 类。

这样就实现了 student 继承 person 类，同时新增类的自我方法 study，这里同样要注意，使用 module.exports 而不是使用 exports。其他两个类 teacher 和 coder 实现继承的方法和 student 一致，主要看下 app.js 脚本文件如何调用这 3 个类 student、teacher 和 coder，其他两个类的代码如下：

(1) teacher 类代码。

```
/**
 * @class Teacher
 */
var util = require("util");
var Person = require('./person');
/**
 *
 * @desc 定义 Teacher 类
 */
function Teacher(){
    Person.call(this);
}

/* 将 Teacher 继承 Person */
util.inherits(Teacher, Person);

/* 重写 teach 方法 */
Teacher.prototype.teach = function(){
    console.log('I am teaching');
}

/* 暴露 Teacher 类 */
module.exports = Teacher;
```

(2) Coder 类代码。

```
/**
 *
 * @class Coder
 */
var util = require("util");
var Person = require('./person');
function Coder(){
    Person.call(this);
}
/**
 *
 * @desc 定义 Coder 类
 */
util.inherits(Coder, Person);

/* 重写 code 方法 */
Coder.prototype.code = function(){
    console.log('I am coding');
}

/* 暴露 Coder 类 */
module.exports = Coder;
```

实现完 3 个继承类后，我们来看如何调用继承类。首先 require 获取 4 个类模块类。


```
var Person = require('./person');
var Student = require('./student');
var Teacher = require('./teacher');
var Coder = require('./coder');
```

使用 new 创建 4 个类的对象。

```
var personObj = new Person();
var studentObj = new Student();
var teacherObj = new Teacher();
var coderObj = new Coder();
```

分别使用 3 个类中来自 Person 的继承方法 sleep 和 eat，同时调用自身特有的方法。代码如下：

```
/* 执行 personObj 对象的所有方法 sleep 和 eat */
console.log('-----for base class of person-----');
personObj.sleep();
personObj.eat();
console.log('-----');

/* 执行 studentObj 对象的所有方法 sleep、eat 和 study */
console.log('-----for class of student-----');
studentObj.sleep();
studentObj.eat();
studentObj.study();
console.log('-----');

/* 执行 teacherObj 对象的所有方法 sleep、eat 和 teach */
console.log('-----for class of teacher-----');
teacherObj.sleep();
teacherObj.eat();
teacherObj.teach();
console.log('-----');

/* 执行 coderObj 对象的所有方法 sleep、eat 和 code */
console.log('-----for class of coder-----');
coderObj.sleep();
coderObj.eat();
coderObj.code();
console.log('-----');
```

【代码说明】

- ☐ personObj 为 Person 类的对象，因此其只有 sleep 和 eat 方法。
- ☐ studentObj 为 Student 类的对象，因此其有 sleep、eat 和 study 方法。
- ☐ teacherObj 为 Teacher 类的对象，因此其有 sleep、eat 和 teach 方法。
- ☐ coderObj 为 Coder 类的对象，因此其有 sleep、eat 和 code 方法。

预期执行成功后每个子类不仅可以调用 sleep 和 eat，还可成功调用其自身的方法。执行 app.js 脚本文件代码如下：

```
node app.js
```

执行 app.js 脚本返回结果如图 2-23 所示。

for base class of person 为 Person 类的对象调用 sleep 和 eat 方法输出的结果；for class of student 为调用 Student 类的对象调用的 sleep、eat 和 study 方法输出的结果；for class of teacher 为 Teacher 类的对象调用 sleep、eat 和 teach 方法的输出结果；for class of coder 为 Coder 类

的对象调用 `sleep`、`eat` 和 `code` 方法的输出结果。从结果看出我们成功实现了 Node.js 的类继承关系。

```
root@ubuntu:/home/danhuang/class_inherits# node app.js
-----for base class of person-----
sleep in the night .
eat food
-----
-----for class of student-----
sleep in the night
eat food
I am learning
-----
-----for class of teacher-----
sleep in the night
eat food
I am teaching
-----
-----for class of coder-----
sleep in the night
eat food
I am coding
-----
```

图 2-23 执行 `app.js` 脚本返回结果

提问：如何实现重定义父类函数的方法？

主要是在子类函数方法中添加被继承类的方法，例如 `overload.js` 脚本，代码如下：

```
var util = require("util");
var Person = require('./person');
/**
 *
 * @desc 定义Overload函数类
 */
function Overload(){
    Person.call(this);
    this.eat = function(){
        console.log('eat by overload function');
    }
}

util.inherits(Overload, Person);

module.exports = Overload;
```

【代码说明】

❑ `this.eat = function(){}:` 重定义 `Person` 类中的 `eat` 方法。

如图 2-24 所示为重定义后的 `eat` 方法输出的结果。

```
-----for class of overload-----
eat by overload function
-----
```

图 2-24 执行 `overload.js` 脚本返回结果

Node.js 中可以应用 `module.exports` 实现一个动态类对象，那么 Node.js 中如何实现一个静态类呢？例如：假设有一个基类 `Person`，其有继承类 `Student`，但我们希望使用静态调

用 `Student` 中的方法和属性，而不希望 `new` 一个对象。可以结合 `exports` 以及继承方法来实现静态类。首先实现一个基类 `Person`。代码如下：

```
/**
 *
 * class for the person
 * param: name
 * method: sleep, eat
 */
module.exports = function(){
  this.name = 'person';
  /* 定义 sleep 方法 */
  this.sleep = function(){
    console.log('sleep in the night');
  }
  /* 定义 eat 方法 */
  this.eat = function(){
    console.log('eat food');
  }
}
```

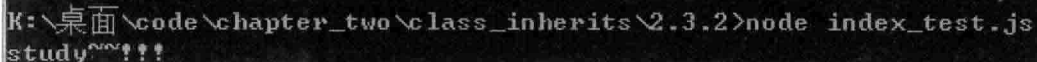
接下来我们应用继承的方法，将 `student` 继承 `person` 类，并实现一个动态的 `Student` 类。代码如下：

```
var util = require("util");
var Person = require('./person');
function Student(){
  Person.call(this);
  util.inherits(Student, Person);
  this.study = function(){
    console.log('study~~!!!');
  }
}
module.exports = Student;
```

如果是如上代码实现方式的话，当 `require` 该文件时，将会得到一个动态 `Student` 类，如果是动态类的话，那么调用方式可以如下 `index_test.js` 代码：

```
var Student = require('./student');
var student = new Student();
student.study();
```

执行 `index_test.js` 可以看到如图 2-25 所示返回信息。



```
K:\桌面\code\chapter_two\class_inherits\2.3.2>node index_test.js
study~~!!!
```

图 2-25 执行 `index_test.js` 脚本返回结果

现在使用静态调用方式，调用 `Student` 中的对象，那么我们可以将 `student` 这个模块的实现方式改为如下代码：

```
var util = require("util");
var Person = require('./person');
function Student(){
  Person.call(this);
```

```

/* 应用 util.inherits 实现继承方法 */
util.inherits(Student, Person);
this.study = function(){
    console.log('study~~!!!');
}
}

var person = new Person();

exports.study = person.study;
exports.eat = person.eat;
exports.sleep = person.sleep;

```

【代码说明】

- ❑ `var person = new Person();`: 创建 `Person` 对象;
- ❑ `exports.study = person.study;`: 暴露 `Person` 对象中的 `study` 方法;
- ❑ `exports.eat = person.eat;`: 暴露 `Person` 对象中的 `eat` 方法;
- ❑ `exports.sleep = person.sleep;`: 暴露 `Person` 对象中的 `sleep` 方法。

通过在类定义模块中 `new` 一个本身对象, 并将该方法全部通过 `exports` 暴露给外部接口, 就无需在每次调用该类的地方 `new` 一个该对象了, 调用方式如下 `static_index.js` 代码:

```

var student = require('./student_static');
student.study();
student.eat();
student.sleep();

```

如上代码就无需每次都在调用时 `new` 一个该对象了, 因此在使用上就可以将 `student` 这个类看成一个静态类, 执行 `static_index.js`, 可以看到如图 2-26 所示的返回信息。

```

K:\桌面\code\chapter_two\class_inherits\2.3.2>node static_index.js
study~~!!!
eat food
sleep in the night

```

图 2-26 执行 `static_index.js` 返回结果

这种方法实现很简单, 也很容易理解, 在很多时候非常有用。这样做的好处是可以避免代码的冗余, 当 `student` 这个类被多个地方调用时, 如果是动态调用的话, 就必须每次都去 `new` 一个该对象, 而如果使用类静态方法调用时, 就可以直接通过 `require` 返回的对象进行调用。当然, 不是所有的类都可以这样去调用, 如果每次 `new` 一个对象都需要初始化一些参量时, 可以选择使用动态调用方法。

2.3.3 单例模式

上一节介绍了 Node.js 中如何实现继承关系, 对于一种服务器端语言来说, 设计模式的应用将决定该项目的可维护性和可扩展性。本节将通过实例介绍使用 Node.js 编程语言实现单例模式、适配器模式、装饰模式、工厂模式和观察者模式。

一般认为单例就是保证一个类只有一个实例, 实现的方法一般是先判断实例存在与

否，如果存在，就直接返回，如果不存在，则会创建该对象，并将该对象保存在静态变量中，当下次请求时，则可直接返回该对象，这就确保了一个类只有一个实例对象。

Singleton
-instance : Singleton
-Singleton()
+Instance() : Singleton

图 2-27 单例模式

Node.js 中利用模块实现单例的方法和单例模式的思想是一致的，使用一个私有变量记录 new 的相应对象。创建 single_class.js，实现单例类。代码如下：

```
/* single_class.js */
var _instance = null; //定义初始化_instance
module.exports = function(time){ //定义单例类
    function Class(time){
        this.name = 'danhuang';
        this.book = 'Node.js';
        this.time = time;
    }
    Class.prototype = {
        constructor: Class,
        show: function(){
            console.log(this.book + ' is write by ' + this.name + ',time is ' + this.time);
        }
    }
    this.getInstance = function(){
        if(_instance === null){
            _instance = new Class(time);
        }
        return _instance;
    }
}
```

【代码说明】

- ❑ var _instance = null: 存储 Class 对象。
- ❑ function Class(time)和 Class.prototype: 创建 Class 类和方法。
- ❑ this.getInstance = function(){}: 获取单例对象接口。

注意，_instance 变量要放在单例方法之外，否则将无法实现单例模式。原因是当调用单例方法时每次都会重新将其赋值为 null，而放在单例函数之外时，调用单例函数不会影响到_instance 变量的值。接下来创建调用验证 single_app.js。代码如下：

```
/* single_app.js */
var Single = require('./single_class');
var singleObj1 = new Single('2012-11-10');
var singleClass1 = singleObj1.getInstance('2012-11-10');
singleClass1.show();
var singleObj2 = new Single('2012-11-20');
var singleClass2 = singleObj2.getInstance('2012-11-20');
singleClass2.show();
```

【代码说明】

- ❑ `var Single = require('./single_class')`: 获取单例类。
- ❑ `var singleObj1 = new Single('2012-11-10')`: new 一个新的单例对象，并设置时间为 2012-11-10。
- ❑ `singleClass1.show()`: 调用 `show`，显示当前类中的信息。

`singleObj2` 和 `singleObj1` 是一样的，只是在 `new` 对象的时候传递的时间不同，这个时间主要用来判断是否 `new` 了新的对象。运行 `single_app.js`:

```
node single_app.js
```

运行结果如图 2-28 所示。

```
root@ubuntu:/home/danhuang/design_patterns# node single_app.js
Node.js is write by danhuang,time is 2012-11-10
Node.js is write by danhuang,time is 2012-11-10
```

图 2-28 执行 `single_app.js` 返回结果

从图 2-28 中可看出，两次输出的结果都是 2012-11-10，因此说明第二次 `new` 单例对象的时候，没有创建新的 Class 类对象，而是返回了第一次创建的 Class 类对象。这样就应用 Node.js 编程语言实现了单例模式。

2.3.4 适配器模式

若将一个类的接口转换成客户希望的另外一个接口，Adapter（适配器）模式可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¹。适配器模式如图 2-29 所示。

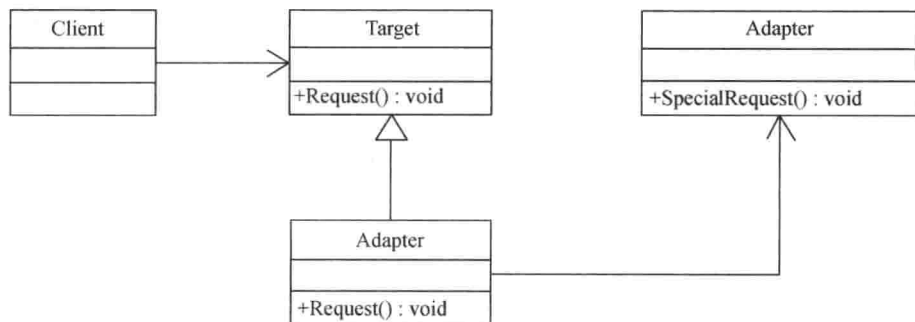


图 2-29 适配器模式

创建 3 个类：Target、Adapter 和 Adaptee。Adapter 继承 Target，Adapter 调用 Adaptee 中方法的具体实现。创建一个 Target 类，其中包含有 `request` 方法。代码如下：

```
/* target.js */
module.exports = function(){
  this.request = function(){
    console.log('Target::request');
  }
}
```

¹ 摘自百度百科：<http://baike.baidu.com/view/66964.htm>。

创建一个 Adapter（适配器）类，继承 Target 类，并重定义其 request 方法。代码如下：

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

/**
 *
 * @desc 定义 Adapter 函数类
 */
function Adapter(){
    Target.call(this);
    this.request = function(){
        var adapteeObj = new Adaptee();
        adapteeObj.specialRequest();
    }
}
/* 设置 Adapter 继承 Target 类 */
util.inherits(Adapter, Target);
module.exports = Adapter;
```

【代码说明】

- ❑ this.request = function(){}：重定义父类中的 request 方法。
- ❑ var adapteeObj = new Adaptee()：创建 Adaptee 类的对象 adapteeObj。
- ❑ adapteeObj.specialRequest()：调用 adapteeObj 中的 specialRequest 方法。
- ❑ util.inherits(Adapter, Target)：使用 util.inherits 继承 Target 类。

这里涉及类中如何重定义父类函数的实现，关于这部分知识在本书的 2.3.2 节中有详细的介绍。创建 Adaptee 类，实现 specialRequest 方法，代码如下：

```
/* adaptee.js */
module.exports = function () {
    this.specialRequest = function () {
        console.log('Adaptee::specialRequest');
    }
}
```

创建一个 Adaptee 类，并添加其 specialRequest 方法。最后创建测试脚本 client 通过适配器调用其中的 request 方法。代码如下：

```
/* client.js */
var Adapter = require('./adapter');

var target = new Adapter();
target.request();
```

【代码说明】

- ❑ var Adapter = require('./adapter')：获取适配器模块类。
 - ❑ var target = new Adapter()：使用 new 创建类的对象。
 - ❑ target.request()：调用适配器的 request 方法，其目的是执行 specialRequest 方法。
- 执行测试脚本 client：

```
node client.js
```


执行 client.is 返回结果如图 2-30 所示。

```
root@ubuntu: /home/danhuang/adapter# node client.js
adaptee::specialRequest
```

图 2-30 执行 client.js 返回结果

从如图 2-30 得知，其通过适配器调用了 Adaptee 中的 specialRequest 方法，这样就实现了 Node.js 编程语言下的适配器模式。

2.3.5 装饰模式

动态地给一个对象添加一些额外的职责。就扩展功能而言，它比生成子类方式更为灵活。根据 UML 类图，会创建 component 类和其继承类 Decorator、继承 Decorator 的 concreteDecoratorA 和 concreteDecoratorB 装饰类，UML 类图中的原生构件 concreteComponent。装饰模式如图 2-31 所示。

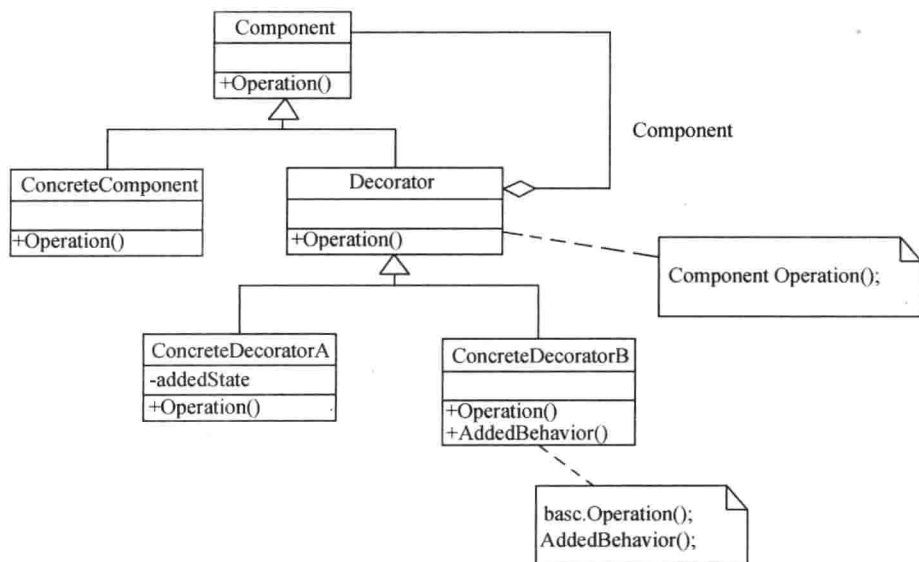


图 2-31 装饰模式

创建基类 Component，其添加 operation 方法和创建一个 Node.js 的模块类方法一致。代码如下：

```
/* component.js */
module.exports = function(){
  this.operation = function(){
    console.log('Component::operation');
  }
}
```

创建原生继承类 ConcreteComponent，其主要作用是展示 Component 装饰之前类中的属性和方法，重定义 operation 方法。代码如下：

```
/* concreteComponent.js */
var util = require("util");
```

```

var Component = require('./component');
/**
 *
 * @desc 定义 ConcreteComponent 函数类
 */
function ConcreteComponent() {
    Component.call(this);
    this.operation = function() {
        console.log('output by the concrete component');
    }
}

util.inherits(ConcreteComponent, Component);
module.exports = ConcreteComponent;

```

代码为一个简单的继承，这里就不详细描述这段代码的含义了。创建一个 **Decorator** 基类，用于装饰 **Component** 类。这部分代码也仅仅只是一个简单的继承关系。代码如下：

```

/* decorator.js */
var util = require("util");
var Component = require('./component');
/**
 *
 * @desc 定义 Decorator 函数类
 */
function Decorator() {
    Component.call(this);
}

util.inherits(Decorator, Component);
module.exports = Decorator;

```

创建 **ConcreteDecoratorA** 装饰类，该类的目的是为 **Component** 类的 **operation** 方法提供一些额外的操作。代码如下：

```

/* concreteDecoratorA.js */
var util = require("util");
var Decorator = require('./decorator');
/**
 *
 * @desc 定义 ConcreteDecoratorA 函数类
 */
function ConcreteDecoratorA() {
    Decorator.call(this);
    this.operation = function() {
        Decorator.operation;
        console.log('add some decorator by ConcreteDecoratorA');
    }
}

util.inherits(ConcreteDecoratorA, Decorator);
module.exports = ConcreteDecoratorA;

```

【代码说明】

- ❑ `this.operation = function(){}:` 重定义 Component 中的 operation 方法;
- ❑ `Decorator.operation:` 调用被装饰类的 operation 基本方法;
- ❑ `console.log('add some decorator by ConcreteDecoratorA');` 在 operation 方法上, 添加额外的装饰。

ConcreteDecoratorA 这个类仅仅只是将 Component 中的 operation 方法进行了一些装饰, 比如添加一些额外的计算规则和输出一些额外的数据。

创建 ConcreteDecoratorB, 该类的目的是为 Component 类的 operation 方法提供一些额外的操作, 同时添加新的功能方法 `addedBehavior`, 代码如下:

```
/* concreteDecoratorB.js */
var util = require("util");
var Decorator = require('./decorator');
/**
 *
 * @desc 定义 ConcreteDecoratorB 函数类
 */
function ConcreteDecoratorB() {
    Decorator.call(this);
    this.operation = function() {
        Decorator.operation
        console.log('add some decorator by ConcreteDecoratorB');
    }
    this.addedBehavior = function() {
        console.log('add new method by ConcreteDecoratorB');
    }
}

util.inherits(ConcreteDecoratorB, Decorator);

module.exports = ConcreteDecoratorB;
```

【代码说明】

- ❑ `this.operation = function(){}:` 装饰 Component 类中的 operation 方法。
- ❑ `this.addedBehavior = function(){}:` 装饰 Component 添加新的行为动作。

装饰类 ConcreteDecoratorB 在 ConcreteDecoratorA 的基础上装饰了一些动作和行为 `addedBehavior`。装饰模式的应用场景是在不改变基类的情况下, 为基类新增属性和方法。

2.3.6 工厂模式

定义一个用于创建对象的接口, 让子类决定将哪一个类实例化。Factory Method 使一个类的实例化延迟到其子类。如图 2-32 所示为工厂模式的 UML 类图, 从类图可以获取到工厂模式中需要有基类 Product 和其继承的产品类 ProductA、ProductB, 最后再添加工厂类方法 `creator`。

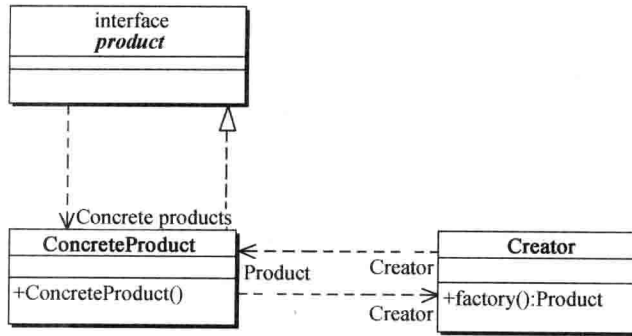


图 2-32 工厂模式

创建一个 **Product** 基类，同时添加 **getProduct** 方法。代码如下：

```

/* product.js */
module.exports = function(){
  this.getProduct = function(){
    console.log('product is get from class of Product');
  }
}

```

创建两个继承于 **Product** 的类 **ProductA** 和 **ProductB**。代码如下：

```

/* productA.js */
var Product = require('./product');
var util = require("util");

/**
 *
 * @desc 定义 ProductA 函数类
 */
function ProductA(){
  Product.call(this);
  this.getProduct = function(){
    console.log('product is get from class of ProductA');
  }
}

util.inherits(ProductA, Product);

module.exports = ProductA;

```

ProductA 和 **ProductB** 实现是一样的，只是在 **getProduct** 输出的一个是 'product is get from class of ProductA'，另外一个是 'product is get from class of ProductB'，通过两个对象方法的输出结果不同来区分两个对象。

创建 **productFactory** 来创建工厂对象，根据不同的参数获取不同的 **Product** 对象。这里需要注意的是，**createProduct** 使用 **exports**，而不是使用 **module.exports**，目的是传递一个 **ProductFactory** 对象，而非一个 **ProductFactory** 类。代码如下：

```

/* productFactory.js */
var ProductA = require('./productA');
var ProductB = require('./productB');

exports.createProduct = function(type){

```

```

switch(type) {
  case 'ProductA' : return new ProductA();
  break;
  default:
  case 'ProductB' : return new ProductB();
  break;
}
}

```

创建 client.js, 使用 ProductFactory 来调用 createProduct 创建 Product 对象。

```

var ProductFactory = require('./productFactory');
var ProductA = ProductFactory.createProduct('ProductA');
ProductA.getProduct();
var ProductB = ProductFactory.createProduct('ProductB');
ProductB.getProduct();

```

【代码说明】

- ❑ var ProductFactory = require('./productFactory'): 获取 ProductFactory 模块对象。
- ❑ var ProductA = ProductFactory.createProduct('ProductA'): 创建 ProductA 对象。
- ❑ ProductA.getProduct(): 调用对象的 getProduct 方法, 输出对象属性。
- ❑ ProductB 的代码和 ProductA 是类似的。

执行 client.js 脚本文件:

```
node client.js
```

如图 2-33 所示为执行结果。

```

root@ubuntu:/home/danhuang/decorator/factory# node client.js
product is get from class of ProductA
product is get from class of productB

```

图 2-33 执行 client.js 返回结果

从结果可以看出通过传递不同的字符串, 获取了不同的对象 ProductA 和 ProductB。工厂模式还包括工厂方法、抽象工厂两个模式, 有兴趣的同学可以尝试使用 Node.js 去实现。

2.4 本章实践

1. 本章介绍了 Node.js 中 exports 和 module.exports 之间的区别和联系, 下面请读者分别实现如下模块。

(1) 实现 person.js 文件模块, 其返回的是一个 person 函数, 该函数中有 eat 和 say 方法;

(2) 实现 person.js 文件模块, 其返回的是一个包含 eat 方法和 say 方法的对象;

(3) 实现 person.js 文件模块, 其返回的是一个数组, 数组内容为人名;

(4) 实现 person.js 文件模块, 其返回的是一个对象, 该对象中包含一个数组元素。

解答:

(1) 实现 person.js 文件模块, 其返回的是一个 person 函数, 该函数中有 eat 和 say 方

法，代码如下：

```
/**
 *
 * @type class
 * @class Person
 */
module.exports = function(){
  /**
   *
   * person eat action
   */
  this.eat = function(){
    console.log('eat');
  }

  /**
   *
   * person say action
   */
  this.say = function(){
    console.log('say');
  }
}
```

(2) 实现 person.js 文件模块，其返回的是一个包含 eat 方法和 say 方法的对象，代码如下：

```
/**
 *
 * @type object
 * @object object.person.say object.person.eat
 */
exports.Person = {
  /**
   *
   * person say action
   */
  'say' : function(){
    console.log('say');
  },
  /**
   *
   * person eat action
   */
  'eat' : function(){
    console.log('eat');
  }
}
```

(3) 实现 person.js 文件模块，其返回的是一个数组，数组内容为人名，代码如下：

```
/**
 *
 * @type array
 */
module.exports = ['danhuang', 'jimi', 'teley'];
```

(4) 实现 person.js 文件模块，其返回的是一个对象，该对象中包含一个数组元素，代

码如下:

```
/**
 *
 * @type array
 */
exports.arr = ['danhuang', 'jimi', 'teley'];
```

测试代码如下:

```
/**
 *
 * test code
 */
var PersonClass = require('./person_class');
var personObject = require('./person_object');
var personArray = require('./person_array');
var personArrayObj = require('./person_array_object');
/* person_class */
console.log('=====');
console.log(PersonClass);
var personClass = new PersonClass();
personClass.say();
personClass.eat();

/* person_object */
console.log('=====');
console.log(personObject);
personObject.Person.say();
personObject.Person.eat();

/* person_array */
console.log('=====');
console.log(personArray);

/* person_array_object */
console.log('=====');
console.log(personArrayObj);
console.log(personArrayObj.arr);
```

测试代码运行结果, 如图 2-34 所示。

```
L:\Code\exercises\2.1>node index.js
=====
[Function]
say
eat
=====
< Person: < say: [Function], eat: [Function] > >
say
eat
=====
[ 'danhuang', 'jimi', 'teley' ]
=====
< arr: [ 'danhuang', 'jimi', 'teley' ] >
[ 'danhuang', 'jimi', 'teley' ]
```

图 2-34 测试代码 index.js 运行结果

2. 本章 2.2 节介绍了 jade 模板的应用, 下面需要根据 jade 模板的介绍实现一个简单的 Web 页面。

现有的 app 应用中, 在 app/routers/index.js 代码中传递一个名为 info 的 json 对象, json 值为 `{ "name": "danhuang", "book": "Node.js" }`。同时在 index.jade 页面使用 div 展示 name, 使用 h2 标签展示 book。

分析: 修改 express 框架中的 app/routers/index.js 文件, 将其代码改为如下:

```
/*
 * GET home page.
 */
exports.index = function(req, res){
  res.render('index.jade', {name:"danhuang", book:"Node.js", title:"a"});
};
```

修改该 index.js 指定显示的 index.jade 文件, 修改后代码如下:

```
extends layout

block content
  h1 #{book}
  p Welcome to #{name}
```

修改成功后, 启动 app 文件, 打开浏览器 `http://127.0.0.1:3000`, 可以看到如图 2-35 所示的响应 Web 页面。

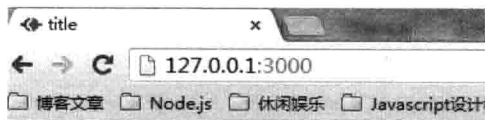


图 2-35 浏览 `http://127.0.0.1:3000` 响应 Web 页面

3. 本章 2.2 节中介绍了 socket.io 模块, 现使用 socket.io 创建一个 socket 服务器端, 监听本地端口 3001, 客户端连接成功时, 输出 success 信息, 并将消息 `{state:"success"}` 发送到服务器。服务器收到消息后, 输出收到的消息, 判断 state 是否为 success, 是 success 就发送 `{me:"very good"}`, 若为其他信息则返回 `{other:"that is all"}`, 客户端收到 very good 后, 返回 `{connection:'good'}`。

分析: 根据问题描述, 我们可以将整个交互过程使用图 2-36 展示出来。

以上充分地分析了 socket 整个交互过程, 接下来我们看一下具体的实现。

首先我们看一下服务器端需要处理的逻辑包含, 启动 Socket 服务、接受 msg 消息, 发送 `{me:"very good"}` 和 `{other:"that is all"}` 消息, 因此其代码逻辑如下:

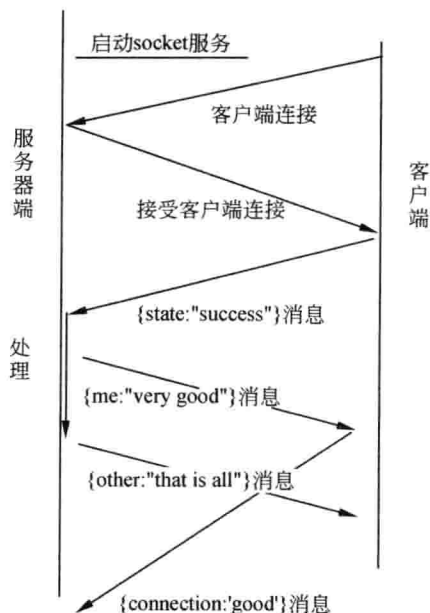


图 2-36 socket.io 交互图

```

/* index_server.js */
var io = require('socket.io').listen(8080,{log:false});
io.sockets.on('connection', function (socket) {
  socket.on('msg', function (data) { //监听 msg 消息
    console.log(data);
    if(data.state){
      if(data.state == 'success'){
        socket.emit('msg', { 'me': 'very good' });
      } else {
        socket.emit('msg', {other:"that is all"});
      }
    } else {
      socket.emit('msg', {other:"that is all"});
    }
  });
});

```

【代码说明】

- ❑ `require('socket.io').listen(8080,{log:false})`: 监听本地 8080 端口, 并关闭日志打印功能;
- ❑ `io.sockets.on('connection'...`: 启动 socket 服务;
- ❑ `socket.on('msg'...`: 监听 socket 客户端发送的 msg 消息;
- ❑ `socket.emit('msg'...`: 向该 socket 客户端发送 msg 消息。

接下来看一下 Web 客户端是如何实现 socket 连接, 并发送消息的。

```

/* index_client.html */
<script src="socket.js"></script>
<script>
  var socket = io.connect('http://localhost:8080');
  socket.emit('msg', { 'state': 'success' });
  socket.on('msg', function (data) {

```

```

    console.log(data);
    if(data.me){
        socket.emit('msg', {connection:'good'});
    }
  });
</script>

```

【代码说明】

- ❑ `<script src="socket.js"></script>`: 加载 socket.js 前端 javascript 脚本文件;
- ❑ `io.connect('http://localhost:8080')`: 连接本地 8080 端口下的 socket 服务;
- ❑ `socket.emit('msg', { 'state': 'success' })`: 发送 socket 消息, 消息名为 msg;
- ❑ `socket.on('msg'...)`: 响应服务器端发送的 socket 的 msg 消息。

客户端代码需要加载 socket.io 官网提供的前端 JavaScript 脚本, 该脚本提供了 socket 连接, 以及 socket 之间的数据交互 API 接口。

运行 index_server.js 脚本, 接下来直接打开 index_client.html 文件, 在客户端和服务端分别可以得到如图 2-37 所示的返回信息。

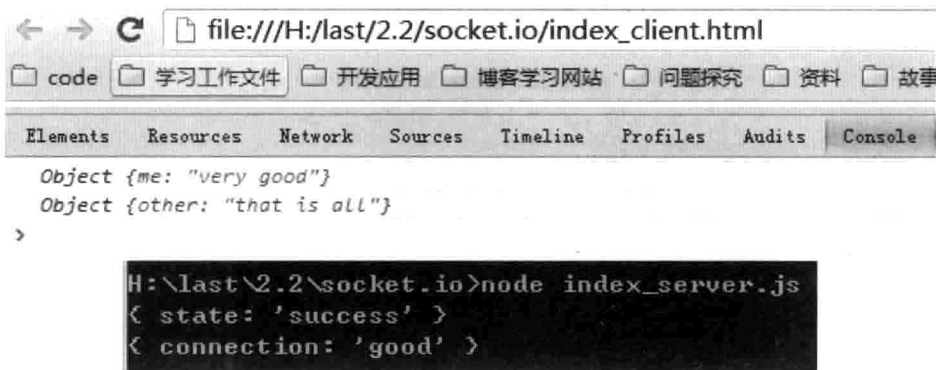


图 2-37 Socket.IO 测试示例

如上就实现了一个简单的 Socket 交互过程。

4. PHP 中提供了 curl¹, 现在我们使用 Node.js 的 request 模块实现类似 curl 的模块, 该模块提供两个方法, 分别是发起 GET 和 POST 的 HTTP 请求方法。两个方法同样接收三个参数: url、param (需要 get 或者 post 的参数) 和 callback (回调函数)。

分析: 实现 curl 模块时, 首先要了解一下 request 模块, 该模块包含了发起 HTTP 请求的相关 API 接口, request.get 和 request.post。下面我们将介绍两个接口的实现。

curl 中的 GET 方法, 需要处理的主要逻辑是将 json 参数对象传递转化为 HTTP 的 GET 请求参数字符, 相关处理代码如下:

```

var params = {};
//为 url 后缀添加?或者&符号
if(get){
    if(url.indexOf( '?') > -1){
        url = url + '&';
    } else {

```

¹ curl 是一个利用 URL 语法规则来传输文件和数据的工具, 支持很多协议, 如 HTTP、FTP、TELNET 等。更让人高兴的是, PHP 也支持 curl 库。本书介绍的是 PHP 中如何运用它。

```

        url = url + '?';
    }
}
url = url + querystring.stringify(get);

```

GET 方法实现代码:

```

get : function(){
    //获取 get 方法的参数 url、get 和 callback
    var url = arguments[0]
        , get = arguments[1]
        , callback = arguments[2];
    if(!callback && typeof get == 'function'){
        get = {};
        callback = arguments[1];
    }

    if(!url){
        callback('');
    }

    var params = {};
    //为 url 后缀添加?或者&符号
    if(get){
        if(url.indexOf( '?') > -1){
            url = url + '&';
        } else {
            url = url + '?';
        }
    }
    url = url + querystring.stringify(get);

    params[ 'url' ] = url;
    params[ 'json' ] = true;
    //调用 request 请求资源
    request.get(params, function(error, response, result){
        if(error){
            console.log(error);
            callback(result);
        } else {
            callback(result);
        }
    });
}

```

相应的 POST 方法和 GET 方法相同, 整体的 CURL 模块代码如下:

```

var request = require('request');
var querystring = require('querystring');

module.exports = {
    get : function(){
        //获取 get 方法的参数 url、get 和 callback
        var url = arguments[0]

```

```

    , get = arguments[1]
    , callback = arguments[2];
    if(!callback && typeof get == 'function'){
        get = {};
        callback = arguments[1];
    }
    //判断 url 是否存在
    if(!url){
        callback('');
    }

    var params = {};
    //为 url 后缀添加?或者&符号
    if(get){
        if(url.indexOf( '?') > -1){
            url = url + '&';
        } else {
            url = url + '?';
        }
    }
    /* url 字符拼凑 */
    url = url + querystring.stringify(get);

    params[ 'url' ] = url;
    params[ 'json' ] = true;
    //调用 request 请求资源
    request.get(params, function(error, response, result){
        if(error){
            console.log(error);
            callback(result);
        } else {
            callback(result);
        }
    });
},

post : function(){
    //获取 get 方法的参数 url、get 和 callback
    var url = arguments[0]
        , post = arguments[1]
        , callback = arguments[2];
    if(!callback && typeof post == 'function'){
        post = {};
        callback = arguments[1];
    }
    //判断是否存在 url
    if(!url){
        callback('');
    }

    var params = {};

```

```

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

    //应用 request 的 post 方法, 提交数据
    request.post(params, function(error, response, result){
        if(error){
            callback(result);
        } else {
            callback(result);
        }
    });
}
}
}

```

5. 应用本章 2.3 节介绍的知识, 实现一个基类 `animal`, 该基类包含方法 `say`, 该方法输出 `noting`, 接下来实现两个继承类 `duck` 和 `bird`, 其中 `duck` 是一个静态类模块, 其有方法 `say`, 该方法输出 `ga...ga`, 而 `bird` 则是一个动态调用模块, 其有方法 `say`, 该方法输出 `ji...ji`。

分析: 首先了解 Node.js 的类和继承实现方法, 接下来就需要应用到 2.3 节的知识“如何实现一个类静态方法”, 了解以上几点后, 我们来实现如下代码。

`animal.js` 代码:

```

/**
 *
 * @class Animal
 */
module.exports = function(){
    this.say = function(){
        console.log('noting');
    }
}

```

`duck.js` 代码:

```

/**
 *
 * @class Duck
 */
var Animal = require('./animal');
function Duck(){
    /* 应用 argument 对象获取函数参数 */
    var _res = arguments[0];
    var _req = arguments[1];

    /* 将 Animal 方法全部复制给 Duck 对象 */
    Animal.call(this);

    /* 应用 util 的 inherits 实现继承 */
    util.inherits(this, Animal);

    this.say = function(){
        console.log('ga...ga');
    }
}

```



```

/* 将 Duck 的 say 方法暴露给外部接口调用 */
var duck = new Duck();
exports.say = duck.say;

```

bird.js 代码:

```

/**
 *
 * @class Bird
 */
var Animal = require('./animal');
function Bird() {

  /* 应用 argument 对象获取函数参数 */
  var _res = arguments[0];
  var _req = arguments[1];

  /* 将 Animal 方法全部复制给 Bird 对象 */
  Animal.call(this);

  /* 应用 util 的 inherits 实现继承 */
  util.inherits(this, Animal);

  this.say = function() {
    console.log('ji...ji');
  }
}

```

接下来写测试方法, 代码 index.js 如下:

```

var duck = require('./duck'); //获取 duck 对象;
var Bird = require('./bird'); //获取 bird 对象;

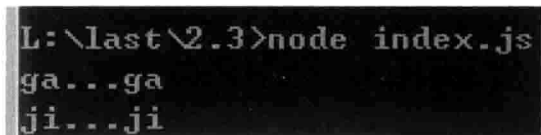
duck.say();

var bird = new Bird();

bird.say();

```

运行结果如图 2-38 所示。



```

L:\last\2.3>node index.js
ga...ga
ji...ji

```

图 2-38 index.js 测试代码运行结果

6. 结合图 2-39 观察者模式的 UML 类图, 使用 Node.js 编程实现观察者模式代码例子, 注意这里涉及的 interface 接口, 我们统一使用 Node.js 的类来代替。

分析: 这里应该包含 3 个类 (观察者接口类、具体观察者类和被观察者类), 这里设置两个具体观察者类 FirstObser 和 SecondObser, 设定观察者接口类为 IObserver, 被观察者类为 Observable, 接下来看每个类的实现。

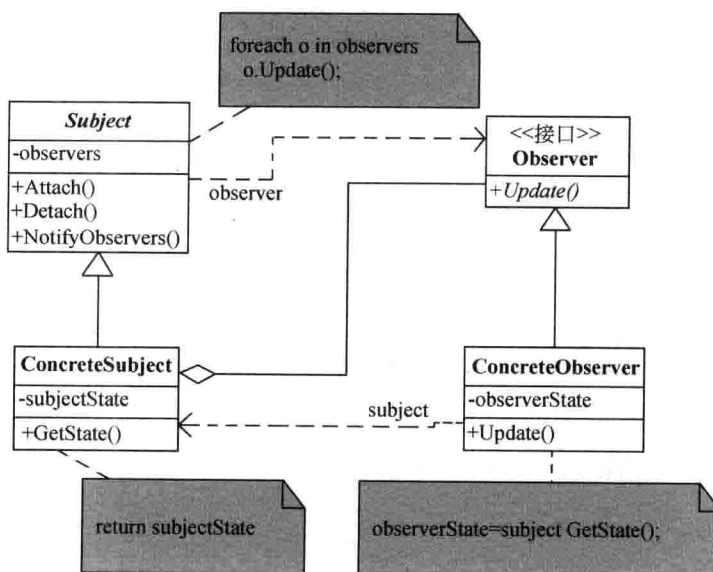


图 2-39 观察者模式 UML 类图

观察者接口类 IObserverP:

```

/**
 *
 * @class Observer
 */
/**
 *
 * @desc Observer 匿名函数
 */
module.exports = function(){
  this.update = function(){
    console.log('base observer');
  }
}

```

具体观察者类 FirstObser:

```

/**
 *
 * @class FirstObserver
 */
var util = require('util');
var Iobserver = require('./iobserver'); //获取 iobserver 类
/**
 *
 * @desc 匿名函数 FirstObserver, 实现继承 Iobserver 这个类
 */
module.exports = function(){
  Iobserver.call(this);
  util.inherits(this, Iobserver);
}
/* 重写父类的 update 方法 */

```

```

    this.update = function(){
        console.log('first observer');
    }
}

```

具体观察者类 SecondObserver:

```

/**
 *
 * @class SecondObserver
 * @desc 匿名函数 FirstObserver, 实现继承 Iobserver 这个类
 */
var util = require('util');
var Iobserver = require('./iobserver');

module.exports = function(){
    Iobserver.call(this);
    util.inherits(this, Iobserver);

    /* 重写父类的 update 方法 */
    this.update = function(){
        console.log('second observer');
    }
}

```

被观察者类为 Observable:

```

/**
 *
 * @class Observable
 * @desc 观察者类
 */

module.exports = function(){
    var m_obserSet = [];
    var _self = this;

    /**
     *
     * @desc 添加观察者
     * @params object observer
     */
    this.addObser = function(observer){
        m_obserSet.push(observer);
    }

    /**
     *
     * @desc 删除观察者
     * @params object observer
     */
    this.removeObser = function(observer){
        if(m_obserSet[observer]){
            delete m_obserSet[observer];
        }
    }
}

/**
 *

```

```

* @desc 通知所有观察者
*/
    this.doAction = function(){
        console.log("Observable do some action");
        _self.notifyAllObser();
    }

/**
 *
 * @desc 执行所有观察者中的 update 方法
 */
    this.notifyAllObser = function(){
        for(var key in m_obserSet){
            m_obserSet[key].update();
        }
    }
}

```

最后新增一个测试代码 index.js:

```

/* 获取所有观察者类 */
var Fober = require('./firstObserver')
    , Sober = require('./secondObserver')
    , Oble = require('./observable');

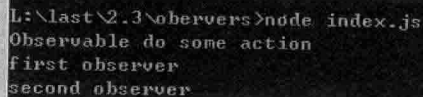
/* 创建观察者类对象 */
var fober = new Fober()
    , sober = new Sober()
    , oble = new Oble();

/* 添加观察者对象 */
oble.addObser(fober);
oble.addObser(sober);

/* 通知所有观察者更改状态 */
oble.doAction();

```

执行测试代码，得到如图 2-40 所示的运行结果，则表明实现了一个观察者模式。



```

L:\last\2.3\observers>node index.js
Observable do some action
first observer
second observer

```

图 2-40 观察者模式运行结果图

2.5 本章小结

本章主要介绍的是 Node.js 编程相关的 NPM 模块，通过实例介绍 Node.js 的原生模块和文件模块的用法。简单地介绍了 NPM 中一些较为常用的模块，现阶段需要掌握的是 express 框架和 socket.IO 的使用，对 forever 和 jade 模板有个大概的了解即可。

本章 2.1 节中重点介绍了 exports 和 module.exports 之间的区别和联系，以下是关于两者之间的总结。

- ❑ 通过 `exports` 返回的属性和方法，都可以通过 `module.exports` 返回。
- ❑ 所有的 `exports` 返回的属性和方法，最终都是通过 `module.exports` 返回的。
- ❑ `module.exports` 可以返回多种数据类型，而 `exports` 返回的只能是一个 `object` 对象。
- ❑ `module.exports` 执行时，其他通过 `exports` 返回的属性和方法都将被忽略。

本章重点是 2.3 节知识点，其中涉及类和模块的概念、Node.js 类继承的实现和 Node.js 编程语言实现设计模式。本章需要掌握 Node.js 的类继承实现方法，以及应用设计模式的 UML 类图编写 Node.js 的设计模式的简单实例代码。

介绍本章的目的是为了初学者能够对模块化编程有一个基本的了解，以及了解 Node.js 中设计模式的应用。下一章将重点转到 Node.js Web 应用上。

本章涉及 Node.js 的 API 列表如表 2.1 所示。

表 2.1 本章Node.js的API列表

模块名	API 名	作用	调用示例
Modules	<code>module.exports</code>	暴露文件模块属性	<code>module.exports = {'a': '1'}</code>
	<code>exports</code>	暴露文件模块属性	
DNS	<code>dns.resolve</code>	Dns 域名解析	<code>dns.resolve(domain, [rrtype], callback)</code>
querystring	<code>querystring.parse</code>	HTTP 参数解析	<code>querystring.parse(str, [sep], [eq], [options])</code>
HTTP	<code>http.createServer</code>	HTTP 服务器创建	<code>http.createServer([requestListener])</code>
	<code>server.listen</code>	启动 HTTP 服务监听地址	<code>server.listen(port, [hostname], [backlog], [callback])</code>
util	<code>util.inherits(constructor, superConstructor)</code>	类继承	

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`。

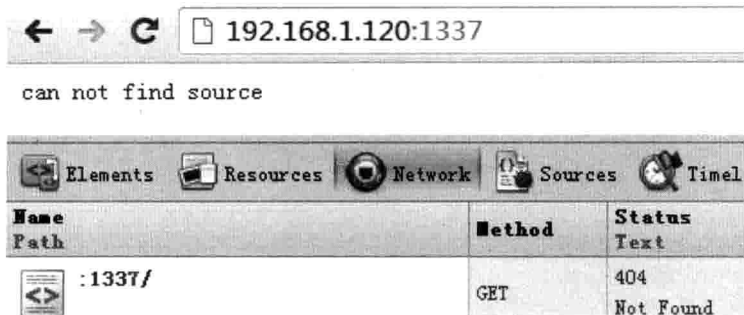


图 3-1 浏览 http://127.0.0.1:1337 返回 Web 页面图

(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。

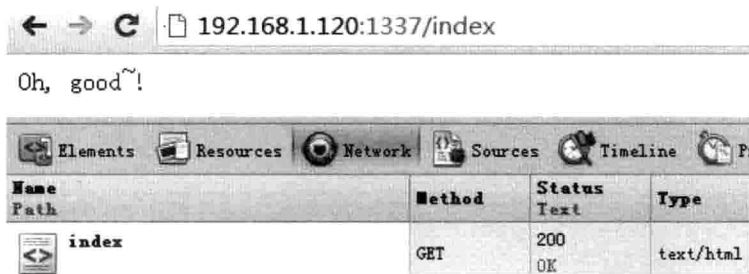


图 3-2 浏览 127.0.0.1:1337/index 返回 Web 页面图

(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。

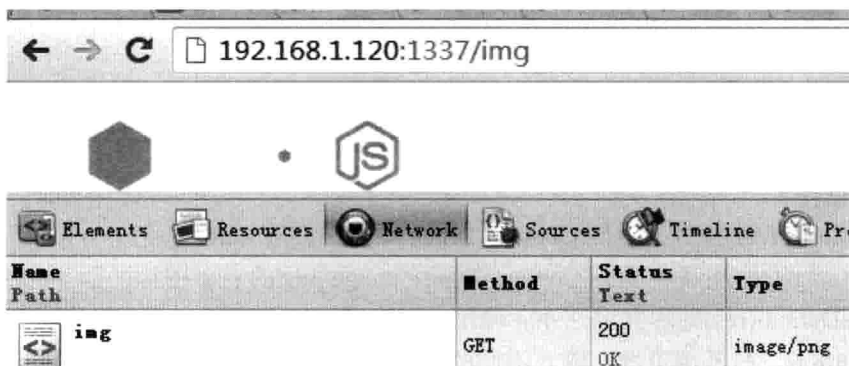


图 3-3 浏览 127.0.0.1:1337/img 返回 Web 页面图

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。

```

/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',
  accept: 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8',
  'user-agent': 'Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.4 (KHTML, like Gecko) Chrome/22.0.1229.79 Safari/537.4',
  '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, 避免不必要的麻烦。

```
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。

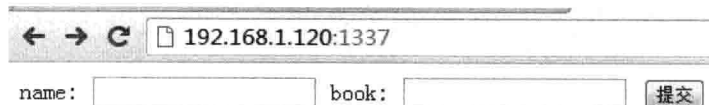


图 3-9 浏览 http://127.0.0.1:1337 返回的 Web 页面图

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。

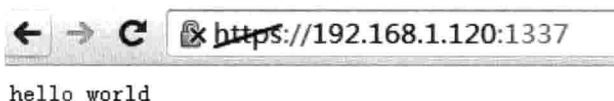


图 3-11 浏览 `https://127.0.0.1:1337` 返回 web 页面图

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。

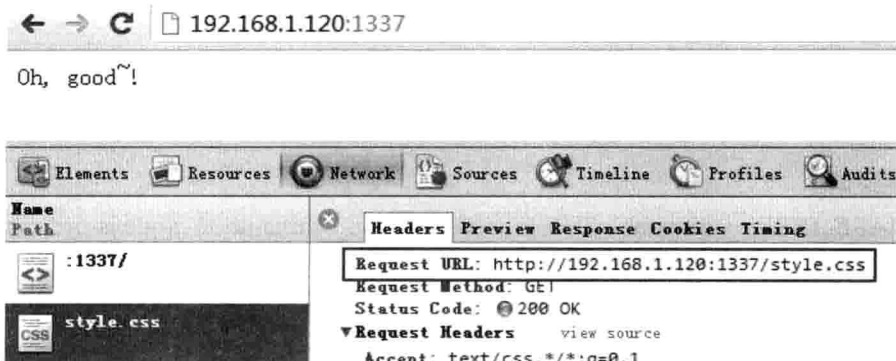


图 3-12 浏览 http://127.0.0.1:1337 返回结果图

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 pointPosition = pathname.lastIndexOf('.'),
                mmieString    = pathname.substring(pointPosition+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.substr(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

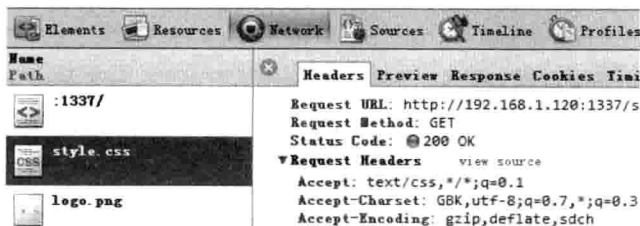


图 3-14 查看 style.css 文件的请求返回 headers 信息图

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 字符串转化为 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。

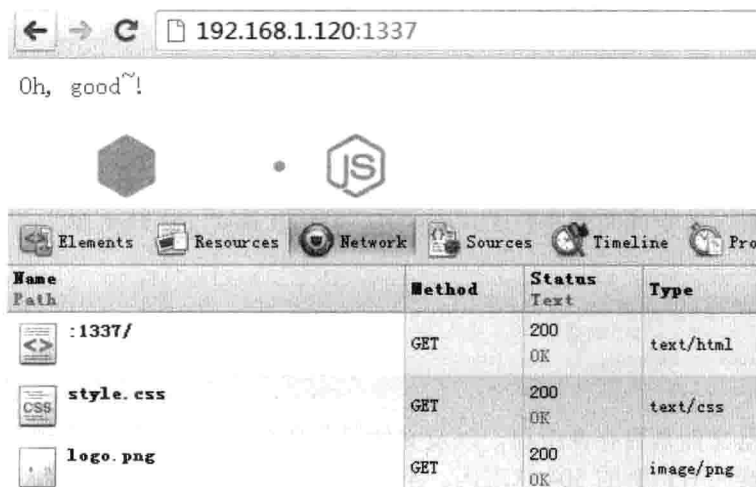


图 3-15 浏览 `http://127.0.0.1:1337` 加载静态资源后返回结果图

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
}

```

这里我们有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。

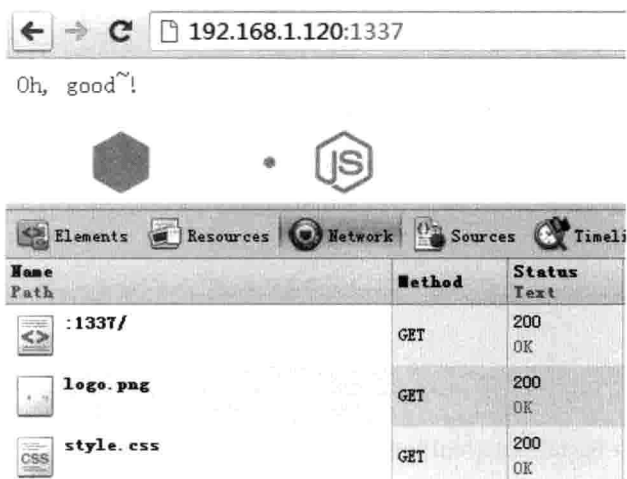


图 3-16 第一次请求无缓存 HTTP 结果图

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。

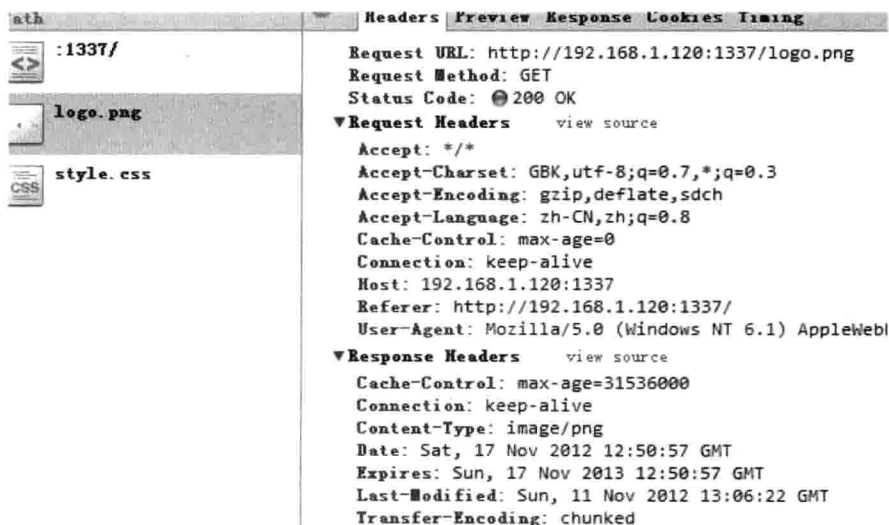


图 3-17 logo.png 请求 headers 信息图

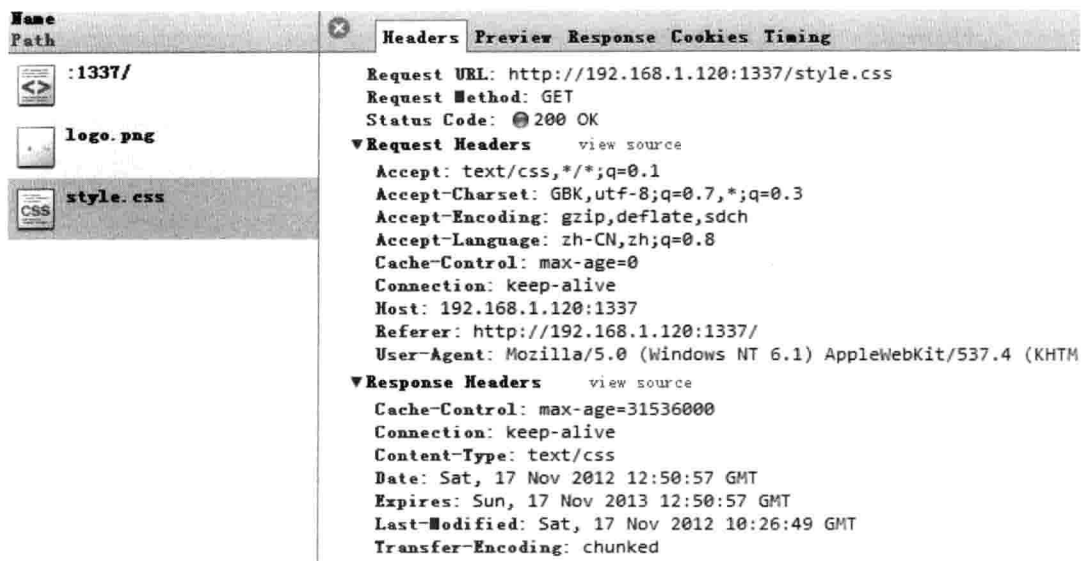


图 3-18 style.css 请求 headers 信息图

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。

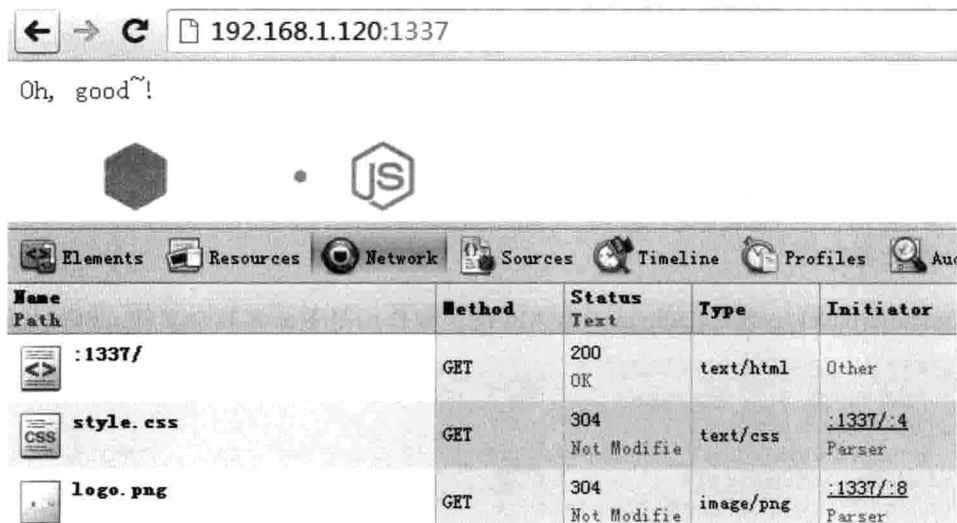


图 3-19 请求有缓存页面返回 HTTP 结果图

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。

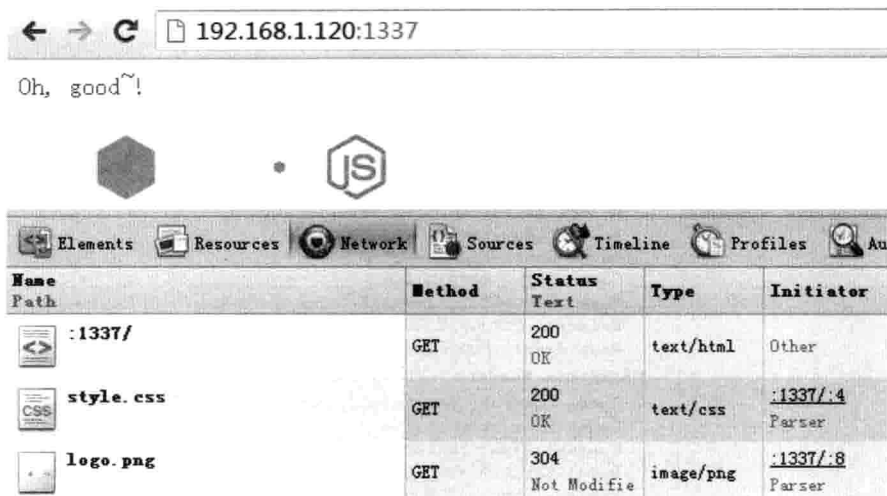


图 3-20 更改 style.css 后重新刷新页面 HTTP 返回图

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

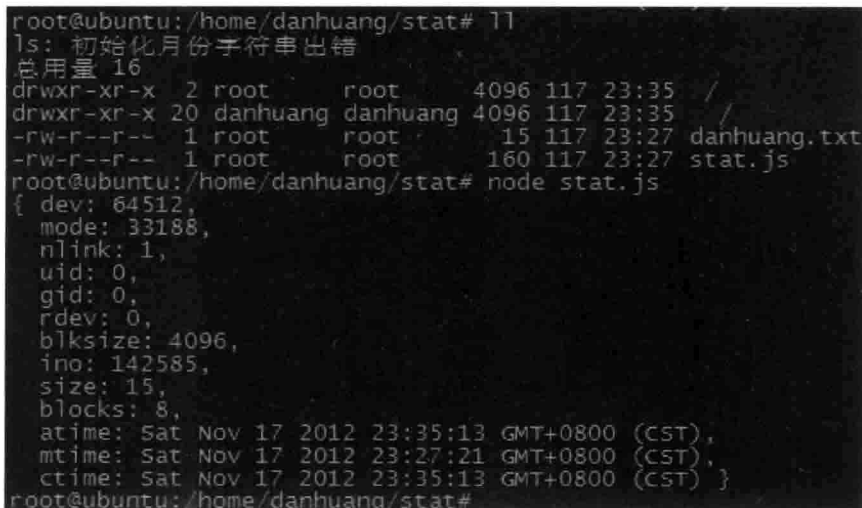
`fs.stat(path, [callback])` 该 API 的作用是读取文件元信息，回调函数将返回两个参数(`err`, `stats`)，其中 `stats` 是 `fs.Stats` 的一个对象。在 3.2 小节中使用了 `fs` 的 `stat` API 来获取文件的最后更改时间。

```
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。



```
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。

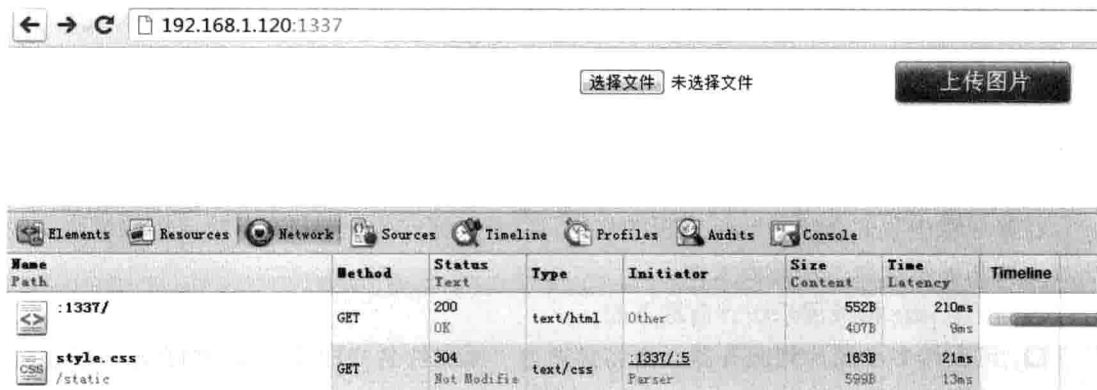


图 3-26 Web 页面以及 HTTP 响应图

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。

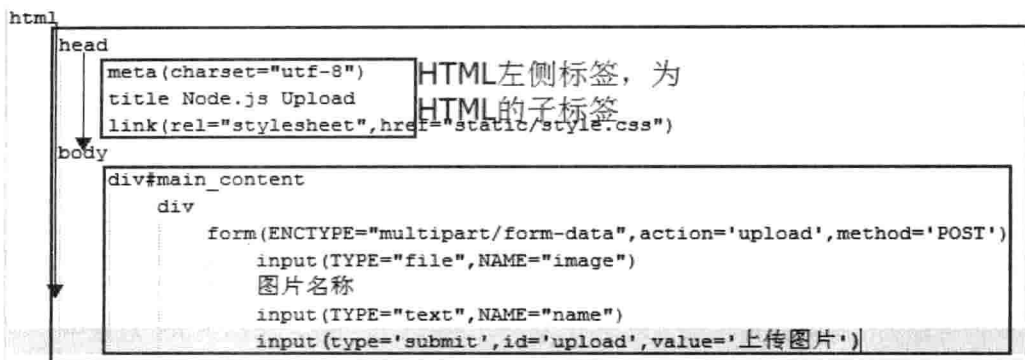


图 3-28 jade 模板代码结构图

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

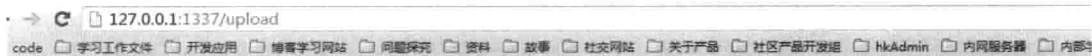
选择相应的图片，单击“上传图片”按钮后，可以看到图片上传信息，如图 3-30 和图 3-31 所示是两次上传图片的结果展示，需要注意的是，图片上传了多次，目的是验证图片上传多次是否会覆盖原有图片。上传文件以及路径中请确认不含有中文字符，具体原因及解决办法可以查看下一节介绍的本应用存在的问题。



图片展示



图 3-30 文件成功上传页面返回图



图片展示



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。

```
fs.js:439
    return binding.rename(pathModule._makeLong(oldPath),
                        ^
Error: EXDEV, cross-device link not permitted 'C:\Users\ADMINI~1\AppData\Local\
emp\606ec6d73b32a29868b250227e5d43e6'
```

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/avatar/" + timestamp + "_" + 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：

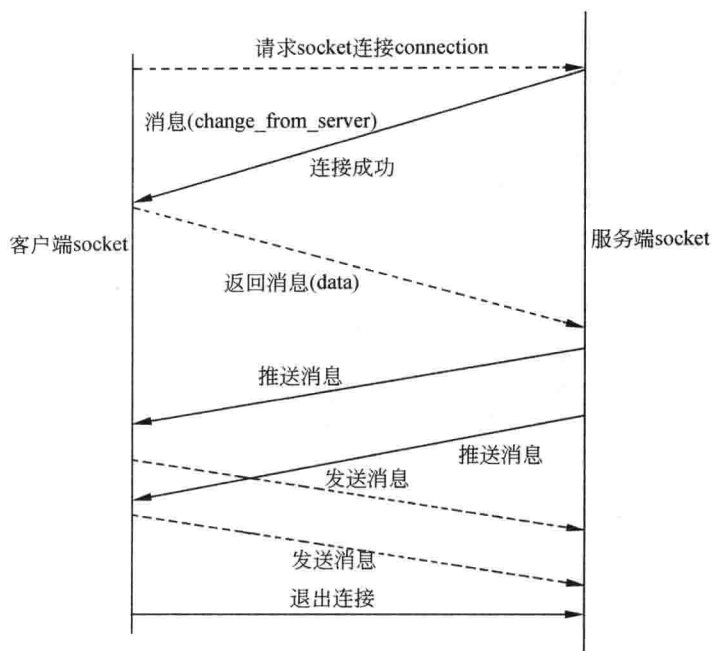


图 3-34 socket 信息交互图

```

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
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!">]>
debug - websocket writing 5:::<"name":"change_from_server","args":[<"msg":"嘿嘿, Node.js 欢迎您! h">]>
```

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[sess.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了解密过程, 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。

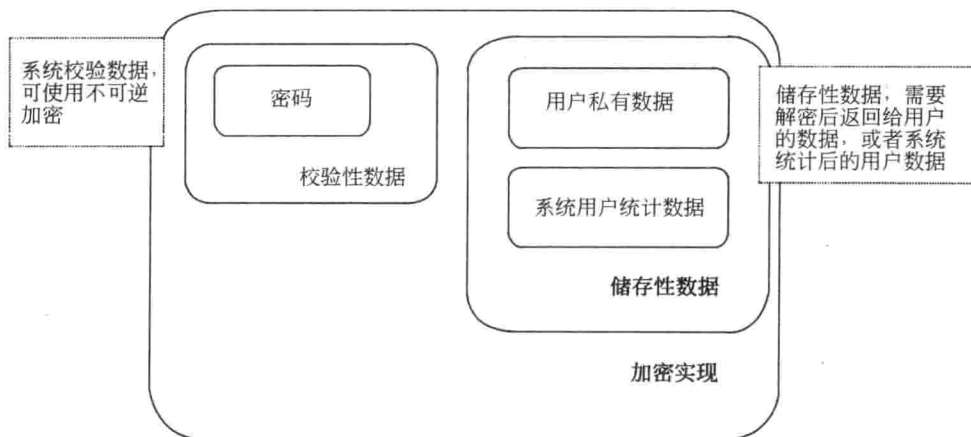


图 3-42 数据加密分析

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'` 或 `'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，因此在上面的代码中，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', 'danhuan', 'hex');
console.log(hashEncodeStr);

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

/*encode with cipher*/
console.log('-----encode with cipher-----');
var cipherEncodeStr = encodeModule.encode('cipher', 'aes-256-cbc', 'danhuan', '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\node>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` 的安裝配置, 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。其能够支持高达 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
```

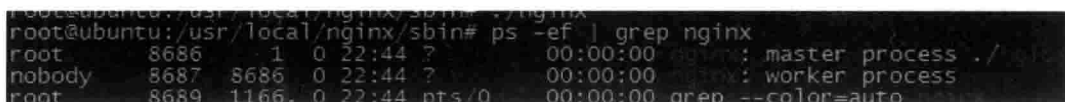
启动时，系统会响应一个错误，错误的原因主要是安装过程中没有关联 `prec library` 库的位置，错误提示是 `error while loading shared libraries: libpcre.so.0: cannot open shared object file: No such file or directory`。解决办法是创建一个软连接将 `PREC` 库和系统 `lib` 库进行关联，方法如下：

```
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# ps -ef | grep nginx
root      8686      1  0 22:44 ?        00:00:00 nginx: master process ./
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
```

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 mysrv {
    #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。

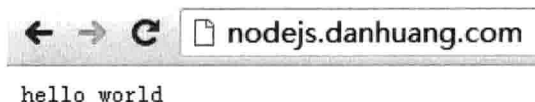


图 3-47 nodejs.danhuang.com 页面响应

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。

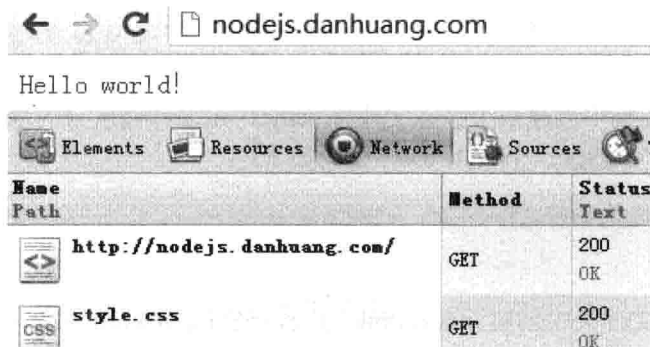


图 3-48 静态资源页面响应

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。

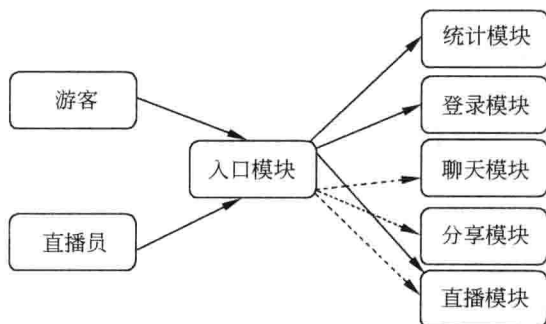


图 3-51 需要设计的模块

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。

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：

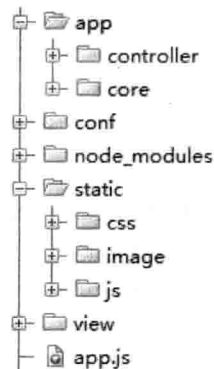


图 3-52 项目目录结构

```

/**
 *
 * 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，因此需要注意分清楚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。

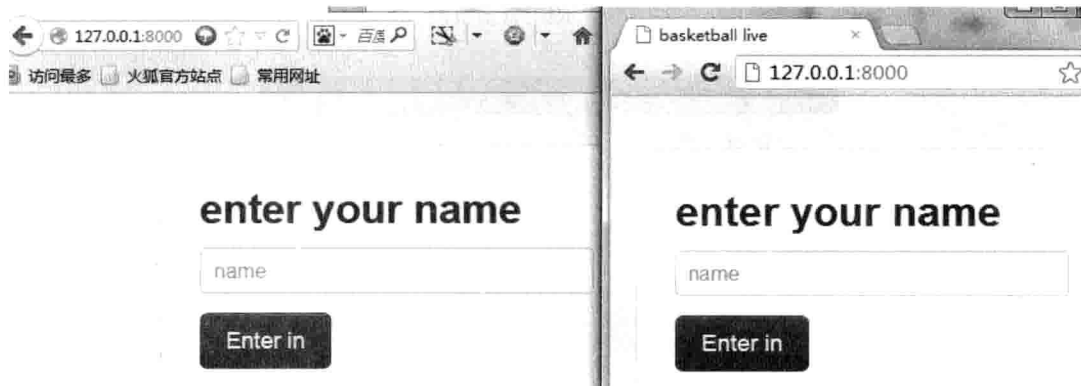


图 3-53 web 登录页面

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/>。

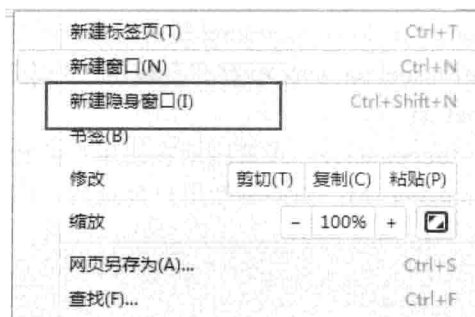


图 3-54 Chrome 打开多个无影响 tab 方式

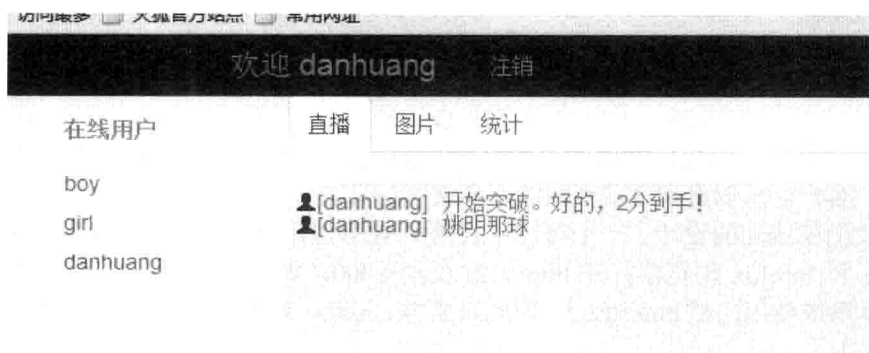


图 3-55 应用直播页面

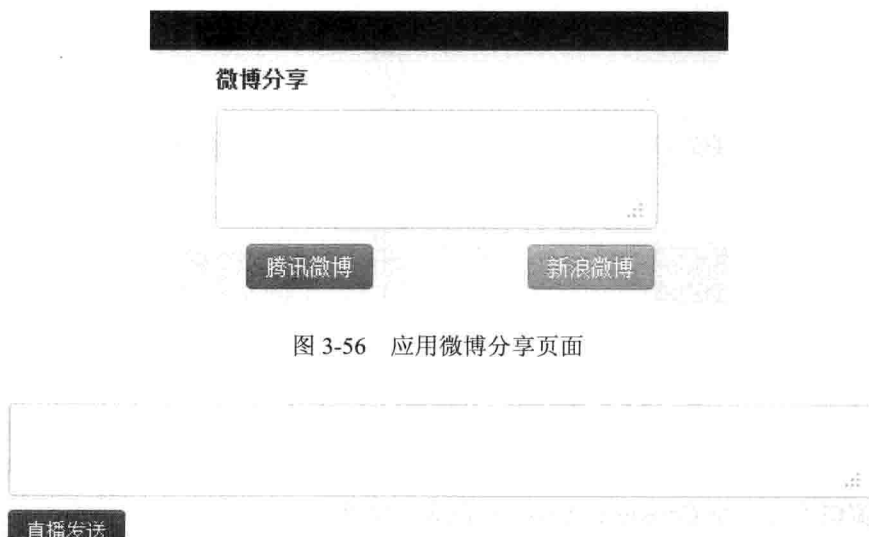


图 3-56 应用微博分享页面



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 `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>。

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: 9a096ed3796c4545aa...
oauth_token_secret: d45267a16e7ab5040ca2...
openid: C81A4B8658E051...EFBE4CE29B68B6
openkey: EE86C1B9D6E...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。

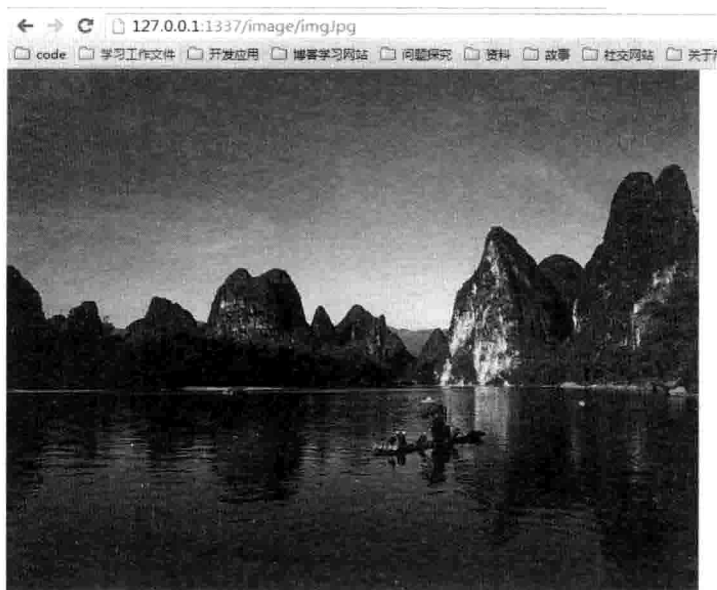


图 3-60 HTTP Web 响应页面

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。



图 3-61 HTTP Web 响应页面

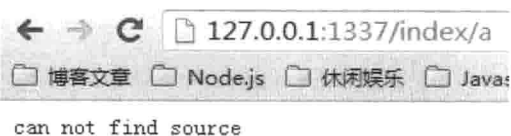
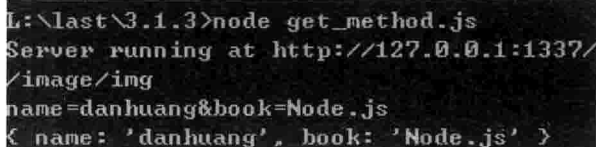


图 3-62 HTTP Web 响应页面

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。

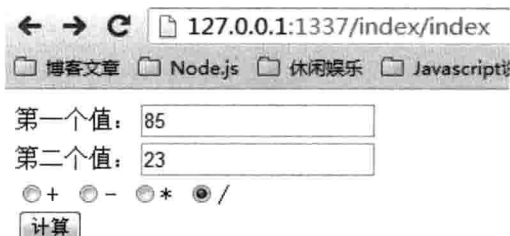


图 3-64 Web 计算器页面

运行结束后, 可以得到执行结果, 如图 3-65 所示。

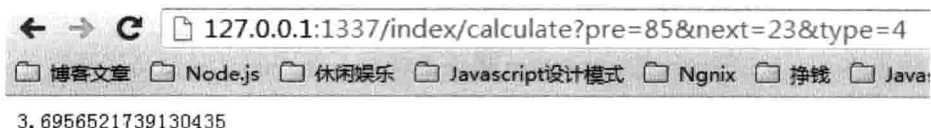


图 3-65 Web 计算器运行结果

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 pointPosition = pathname.lastIndexOf('.'),
                mimeType = pathname.substring(pointPosition+1),
                mmieType;
            switch (mimeType){
                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。

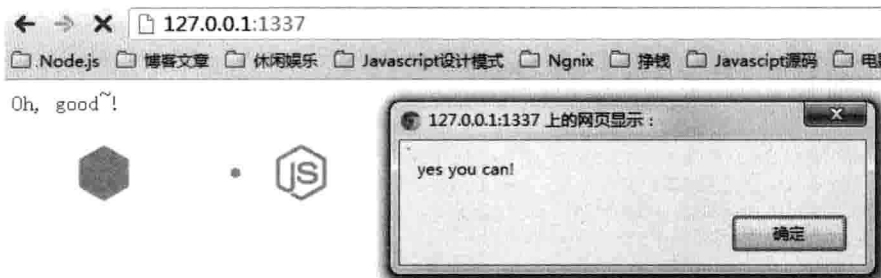


图 3-66 浏览 <http://127.0.0.1:1337/> 响应 Web 页面

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。

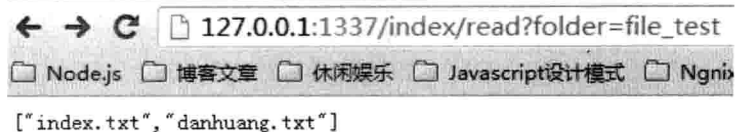


图 3-67 read 文件夹响应页面信息



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 87e46c5cd210bfale162929d25aea8af。同时应用该 key 为 a, 解密字符串为 87e46c5cd210bfale162929d25aea8af, 将会得到如图 3-70 所示返回的 danhuang 字符串。

加密

key:

plaintext:

解密

key:

plaintext:

图 3-69 加密 web 页面

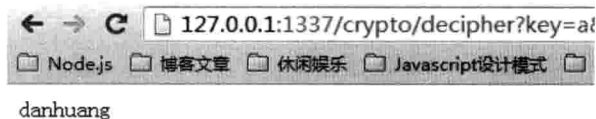


图 3-70 解密结果

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。






	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。






	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)	

第 4 章 Node.js 高级编程

Node.js 其原生提供了一些与网络操作相关的模块，例如 TCP、UDP 和 HTTP 等，应用这些基础模块，就可以实现一些功能复杂的网络服务端应用程序。第 3 章主要是基于 Node.js 的 Web 应用，其集中于 TCP（Transmission Control Protocol）的应用层 HTTP 服务器中的一些开发实践。本章为了进一步加深读者对 Node.js 的网络编程的了解，将介绍 Node.js 的 UDP 服务器的相关知识点，包含 UDP 服务构建，以及简单功能实例等。在本章末尾将介绍 Node.js 执行脚本编写，以及 Node.js 的一些进程管理的相关知识点。

关于本章中 UDP¹（User Datagram Protocol）服务器构建知识点，将从 Node.js 的原生 UDP 模块开始介绍，到实践的构建 UDP 服务器以及 UDP 服务器之间的数据交互等方面会详细阐述。Node.js 网络编程的相关知识点，主要是应用 Node.js 来编写一些可执行脚本，从而辅助服务系统的正常运行。

本章不会深入地阐述 UDP 和 TCP 的相关知识点，如果读者有兴趣，推荐大家通过计算机网络基础知识相关的书籍进行学习，本章的目的是让读者了解 Node.js 网络编程相关知识。

4.1 构建 UDP 服务器

Node.js 原生模块中提供了 UDP / Datagram Sockets 模块，该模块是应用于 UDP 服务器的构建和 IP/UDP 消息来处理等。

学习本章前，首先我们提出一个问题，为什么我们需要 UDP 服务？当网络质量令人不是十分满意的环境下，UDP 协议数据包丢失会比较严重。但是由于 UDP 的特性不属于连接型协议，因而具有资源消耗小、处理速度快的优点，所以通常音频、视频和普通数据在传送时使用 UDP 较多，因为它们即使偶尔丢失一两个数据包，也不会对接收结果产生太大影响。因此，在学习 Node.js 网络编程中，更应该着重了解 UDP 服务器的实现。

4.1.1 UDP 模块概述

Node.js 的 UDP 模块又称做数据报套接字模块，该模块的应用方法类似于 Node.js 源码其他模块，具体就是 `require('dgram')` 后保存其返回对象，并应用其返回对象调用官网提供

¹ UDP 是 OSI 参考模型中一种无连接的传输层协议，提供面向事务的简单不可靠信息传送服务。UDP 协议基本上是 IP 协议与上层协议的接口。UDP 协议适用端口分别运行在同一台设备上的多个应用程序。（摘自百度百科：<http://baike.baidu.com/view/30509.htm>）。

的该模块 API。require 成功地返回对象中包含 createSocket 方法，该方法的应用与 HTTP 模块中的 createServer 方法基本一致。相应示例如下：

```
/*dgram.createSocket.js*/
var dgram = require("dgram");
var server = dgram.createSocket("udp4");
```

4.1.2 UDP Server 构建

了解该模块的 require 机制后，接下来就介绍应用 UDP 模块创建简单的服务器，并学习应用 Node.js 代码来实现 UDP 的客户端，从而可以通过 UDP 客户端连接 UDP 服务，能够实现 UDP 客户端与服务端 UDP 之间的数据交互。下面是关于该实例的一些实践过程。

1. UDP server 的启动运行

根据 4.1.1 节中了解的该模块 require 机制，成功获取 dgram 模块返回对象，并应用其 API 方法 dgram.createSocket 来构建服务器，相关实现代码如下所示。

```
/*dgram.createSocket.js*/
var dgram = require("dgram");
var server = dgram.createSocket("udp4");
/* socket 监听 listening 事件 */
server.on("listening", function () {
    var address = server.address();
    console.log("server listening " + address.address + ":" + address.port);
});
server.bind(41234);
```

【代码说明】

- ❑ dgram = require("dgram"): 获取 dgram 对象。
- ❑ server = dgram.createSocket("udp4"): 应用 dgram.createSocket 接口，创建 udp4 socket 对象。
- ❑ server.on("listening"...: 监听服务器的 listening 事件，当服务器开始启动监听时，执行该回调函数。
- ❑ server.address(): 获取服务器监听端口和 IP 对象。
- ❑ server.bind(41234): 绑定并开始监听端口，启动 udp socket。

上面就是一个简单的 udp4 socket 服务器的构建，该 UDP server 的创建和 socket.io 类似，其包含的过程主要是 require 模块、创建 server 对象，以及启动监听这 3 个过程。本段代码中的 socket.on 接口主要是用来监听服务器运行情况，通过不同的 event 事件，来执行不同的回调函数，代码中应用到 Node.js 官网提供的 listening event 事件，该事件表示在服务器开始启动的时候触发的事件函数。官网还提供了其他 event 事件，如下所列。

- ❑ message: 监听 server 是否有接收到新消息，监听到新消息后执行回调函数。
- ❑ close: 监听是否关闭 UDP 服务器连接，当客户端连接调用 close() 时，执行该事件的回调函数。
- ❑ error: 监听 server 是否出现 error 异常。

根据如上所列事件，可以将代码优化，为其添加一个 message 事件处理函数。代码修

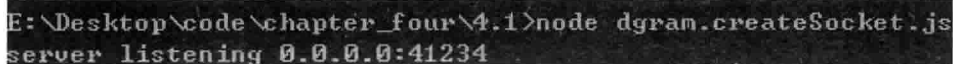
改如下：

```
/*dgram.createSocket.js*/
var dgram = require("dgram");
var server = dgram.createSocket("udp4");
/* 监听 message 消息事件 */
server.on("message", function (msg, rinfo) {
    console.log("server get: " + msg + " from " + rinfo.address + ":" +
rinfo.port);
});
/* 监听 listening 事件 */
server.on("listening", function () {
    var address = server.address();
    console.log("server listening " + address.address + ":" + address.port);
});
server.bind(41234);
```

【代码说明】

- ❑ `server.on("message"...)`: 监听 `message` 事件，当接收到客户端数据时，执行回调函数事件。

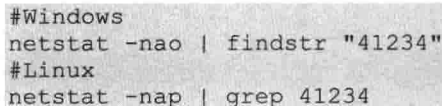
接下来我们运行该 `dgram.createSocket.js` 脚本。为了能够查看该服务是否正常运行，可以通过查询 41234 端口（该 UDP 服务的运行端口），来检查 UDP server 是否成功启动。代码运行后，可以看到如图 4-1 所示结果，从结果中可以了解，在服务启动时，触发了其 `listening` 事件，从而打印出一段提示信息“`server listening 0.0.0.0:41234`”。




```
E:\Desktop\code\chapter_four\4.1>node dgram.createSocket.js
server listening 0.0.0.0:41234
```

图 4-1 UDP 运行返回结果

如图表明 UDP 服务已正常启动。当然，我们可以应用其他方法来验证，例如应用相关命令来查询该 UDP 服务是否已经正常启动运行，下面分别是 Windows 和 Linux 下的查询命令。



```
#Windows
netstat -nao | findstr "41234"
#Linux
netstat -nap | grep 41234
```

 **注意：**上面只是提供了一种查询方法，Linux 下也可以使用 `ps -ef | grep 41234` 查询，更多的方法大家可以多了解 Linux 的常用指令。

查询后的结果如图 4-2 所示。从图中我们可以看到，一个 UDP 服务已经启动，其进程 ID 为 8168。



```
C:\Users\Administrator>netstat -ano | findstr "41234"
UDP        0.0.0.0:41234          *:*      8168
```

图 4-2 Windows 系统下进程查询结果

以上就是 UDP server 的实现和启动运行。在学习完该模块的 UDP 服务器构建后，接

下来我们需要掌握下面两个功能的实现，包含应用客户端连接 UDP server 和通过 UDP 客户端发送数据到服务端，实现客户端和服务端之间的数据交互。

2. UDP client 的启动运行

UDP 服务器启动后，请不要关闭当前运行窗口，避免关闭窗口后进程被 kill。UDP 客户端原理主要是应用 dgram 模块创建与 UDP 服务连接的句柄，并应用该连接向服务端发送数据和接收数据，从而可以实现 UDP 服务之间的数据交互。UDP 客户端相关的实现代码如下：

```
var dgram = require('dgram');
/* 创建 udp4 socket 对象 */
var client = dgram.createSocket("udp4");

/* 创建发送数据 message */
var message = new Buffer("hi danhuang, node.js is waiting for you");

/* 发送数据到本地 41234 端口 */
client.send(message, 0, message.length, 41234, "127.0.0.1");
client.close();
```

【代码说明】

- ❑ dgram.createSocket("udp4"): 创建 udp socket 对象。
- ❑ message = new Buffer("Some bytes"): 将 some bytes 字符转换为数据流对象。
- ❑ client.send: 向绑定的 IP 和端口服务器发送 message 的 Buffer 数据流对象。
- ❑ client.close(): 关闭 UDP 客户端连接。

应用 client.send 指定将 buffer 数据流对象数据发送到相应的 IP/端口的 UDP 服务端。其中，该方法（client.send）的第一个字段为数据流对象，0 为数据偏移量，message.length 为发送数据的字节长度，41234 为 UDP 服务器的监听端口，最后一个为需要连接的 UDP 服务器的 IP 地址（注意其两个参数的类型，端口为 Number 类型，而 IP 为 String 类型）。

代码实现完成以后，我们需要重新打开一个 Windows 下的 cmd 窗口来运行 UDP 的客户端服务。

```
node dgram.createSocket.js
```

执行如上命令，运行成功后可以看到服务器端输出的信息，如图 4-3 所示。

注意：不要将 UDP 服务端的运行窗口关闭，如果关闭，UDP server 的启动进程将会被 kill，那样客户端就无法连接服务器。

```
E:\Desktop\code\chapter_four\4.1>node dgram.createSocket.js
server listening 0.0.0.0:41234
server get: hi danhuang, node.js is waiting for you from 127.0.0.1:59067
```

图 4-3 客户端连接 UDP 服务服务端日志信息


如图 4-3 所示 UDP 服务端接收到了 UDP 客户端发送的字符串数据"hi danhuang, node.js is waiting for you"，这表明已经成功地通过 UDP 客户端连接 UDP 服务，并成功地与 UDP 服务端进行了数据交互。

客户端发送消息到服务器端应用的是其模块提供的 `send` 方法, 同样如果需要服务器端发送消息到客户端时, 其原理是一样的, 应用 `server` 对象的 `send` 方法。但需要注意的是, 服务器端向客户端发送数据时, 需要指定客户端的 IP 和端口, 例如下面代码:


```
/* 监听 message 消息, 消息到达时执行回调函数 */
server.on("message", function (msg, rinfo) {
    var message = new Buffer("success get message for client");
    server.send(message, 0, message.length, rinfo.port, rinfo.
        address);
    });
});
```

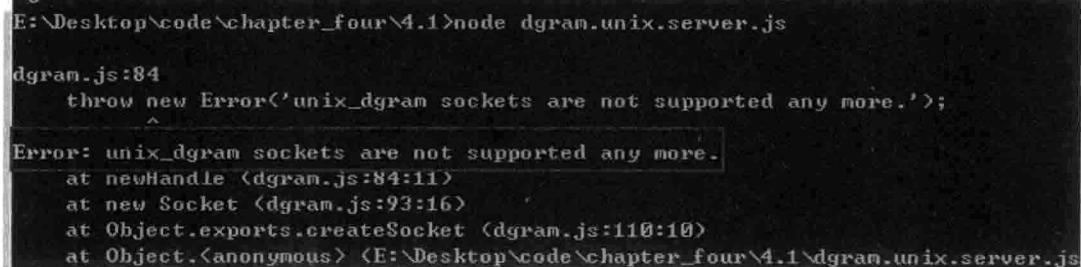
`server.on` 接口监听 `message` 事件, 当有数据接收时, 会执行回调函数, 其 `msg` 为客户端发送的数据, `rinfo` 为发送消息的 UDP 客户端的相关信息, 包括该客户端的 IP 地址、端口等, 服务器端根据客户端的 IP 地址和端口就可以发送消息到指定的 UDP 客户端。客户端接收消息的方法和服务器端相似, 应用如下方法。

```
client.on("message", function(msg, rinfo){})
```

 **注意:** 希望大家根据上面的知识自己实现一个客户端发送消息, 服务器端接受相应的消息, 并经过服务器端的处理后, 将执行结果返回到客户端, 并展示给用户。

上面介绍的主要是应用目标端口和 IP 地址来实现 UDP 通信, 当然和其他 UDP 服务器类似, 其还提供了 Unix 域数据报套接字之间的数据交互方式。我们可以通过指定一个 `path` (路径) 开始在套接字上监听数据报。这种方式和 PHP 的 Unix socket 类似。

 **注意:** 最新版本的 Node.js 0.8 已经不再支持这部分功能了, 因此这里不再讲解, 如果大家应用中有使用这种功能的实例, 可以使用本地 IP 监听方式来解决。如果大家 在 0.8 以上版本还使用该功能的话, 会得到如图 4-4 所示的错误信息。



```
E:\Desktop\code\chapter_four\4.1>node dgram.unix.server.js
dgram.js:84
    throw new Error('unix_dgram sockets are not supported any more.');
```

Error: unix_dgram sockets are not supported any more.
 at newHandle (dgram.js:84:11)
 at new Socket (dgram.js:93:16)
 at Object.exports.createSocket (dgram.js:110:10)
 at Object.<anonymous> (E:\Desktop\code\chapter_four\4.1\dgram.unix.server.js

图 4-4 unix_socket 运行错误信息

以上主要是介绍 UDP 的一些 API, 接下来我们应用 UDP 的 API 来实践开发一个小应用。

4.2 UDP 服务器应用

UDP 服务器可以应用于一些特殊的数据传输, 例如图片、视频和音频信息等。PHP 中

我应用过 UDP 来和 C++ server 交互，主要的目的是希望将 PHP 无法处理的逻辑业务，通过 UDP 服务器发送请求给其他 server 来处理。现在有一个需求，我们有两个 Node.js 服务器分别是 A 和 B，我们希望 A 处理所有业务逻辑，处理完成后交由 B 去做数据库更新。

有多个设计思想可以解决上面的问题，最简单的就是通过 HTTP 发送请求的方式，将 A 处理后的参数通过 HTTP 方式传递给 B 服务器，然后 B 服务器获取参数后更新数据库。这种方式对于 Node.js 来说非常简单，但是 HTTP 是一个 TCP 协议，对于我们自己的两台可信赖的服务器，我们更希望使用 UDP 来传送协议，避免 TCP 中一些不必要的数据传输。

接下来我们要介绍的一个应用，就是使用 Node.js 来处理图片上传切割（图片处理），并通过客户端显示所有的经过处理后的图片列表，而这个功能也将应用 UDP 模块来实现。

4.2.1 节将会介绍本应用的一些功能，以及设计方法；4.2.2 节是介绍本应用的实现过程；4.2.3 节则简单展示本应用，并总结本应用中的知识点。目的是能够让读者掌握 Node.js 中基本的 UDP 服务器构建和应用。

4.2.1 应用分析介绍

作为开发者来说，本应用包含两个部分，一个是图片上传的 Web 服务功能模块、图片处理后的页面展示功能；另外一个则是图片的处理，主要是图片的切割保存。作为用户，希望是这样的一个工具，上传一个图片，并指定其需要切割的长和宽，通过系统处理后返回给用户一个切割好的图片，并通过页面返回展示。

根据以上的需求分析，我们将上面的需求转化为如图 4-5 所示的系统运行流程图。根据应用的分析，我们会设计两个服务器，一个是 Web 服务器，另外一个则是图片处理服务器，两者通过 UDP 协议进行交互。

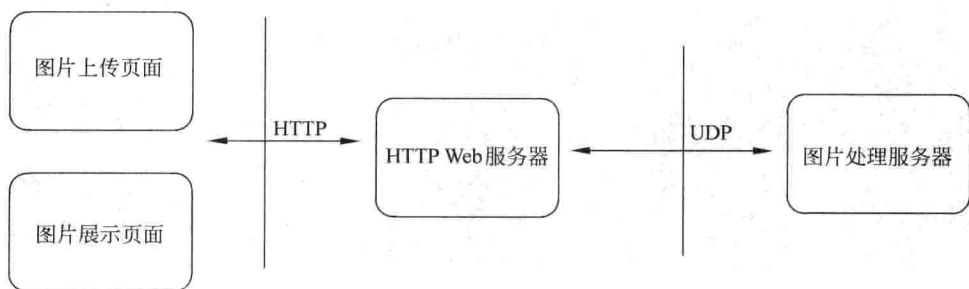


图 4-5 应用分析结构

图片上传页面，主要是图片的上传和预览功能页面；图片展示页面，展示通过图片处理后返回的图片；HTTP Web 服务器主要的作用是文件上传和图片展示；图片处理服务器，将 Web 服务器的数据通过 UDP 协议传递给图片处理服务器，图片处理服务器做一定的处理后返回相应的数据到 Web 服务器。

以上就是本系统需要实现的功能模块。接下来我们讲解具体的实现方法和细节，同时希望读者能够自己实践开发，进一步学习 Node.js 的 UDP 模块的应用。

4.2.2 UDP Server 端（图片处理服务器）实现

根据上面的分析，本应用需要实现的 3 个功能模块分别是，UDP Server 端、UDP Client 端（Web Server）和 Jade 页面功能。

那么首先我们从该应用的 UDP Server 端代码实现原理开始介绍（也就是图片处理服务器）。图片处理服务器作为 UDP 的 server 端，要应用 UDP 模块实现 UDP server，由于该 UDP server 依赖图片处理工具，因此在 UDP 服务程序中会应用 github 中的一个开源 Node.js 图片处理工具——node-imagemagick 来辅助实现图片处理功能。根据上面的分析，我们简单设计出 UDP Server 的代码框架，代码如下：

```
/*dgram.createSocket.js*/
var dgram = require("dgram");
var server = dgram.createSocket("udp4");
/* 监听 message 消息事件 */
server.on("message", function (msg, rinfo) {
    resizeImage();
});
/* 监听 listening 消息事件 */
server.on("listening", function () {
    var address = server.address();
    console.log("server listening " + address.address + ":" + address.port);
});
server.bind(41234);
function resizeImage(){
}
```

【代码说明】

❑ `resizeImage()`：进行相应的图片处理。

上面代码和 4.1 节中的 UDP Server 最大的区别就是新增了一个 `resizeImage` 方法，该方法的主要作用就是应用 `node-imagemagick` 工具实现图片处理。

`node-imagemagick` 工具是由 `rsms` 在 github 上发布的一个开源的图片处理工具，其中包含多个图片的处理方法。具体可以查看 github 主页 <https://github.com/rsms/node-imagemagick>。

该模块 NPM 安装方法是，在 Linux 或者 Windows 的 cmd 命令窗口模式下，执行如下命令：

```
npm install imagemagick
```

需要注意的是，在使用该工具时，必须安装 `imagemagick` CLI 系统工具软件，如果不安装，在运行期间会抛出异常消息：Command failed: `execvp()`: No such file or directory。以下是笔者在 github 找到的一个关于这个问题的疑问以及相应的解答。

```
Problem:
Command failed: execvp(): No such file or directory
Answer[这个是图吗 如果是图 就按图来]:
I met the same problem...
At last, I found the reason of that is node-imagemagick depend on the
imagemagick module, unfortunately, this module is not installed in my system.
```

```
My system is Ubuntu. just run the following command.
sudo apt-get update
sudo apt-get install imagemagick --fix-missing
```

因此在运行本应用代码时，需要执行 `sudo apt-get install imagemagick --fix-missing` 来安装 `imagemagick`。在 Windows 下还没有办法找到更好的解决办法。

接下来我们就应用 `node-imagemagick` 工具提供的 `resizeImage` 方法来实现 `resizeImage`。根据 github 主页提供的 API `resize(options, callback(err, stdout, stderr))`，该方法的 `options` 对象包含如下参数：

```
{
  srcPath: undefined,
  srcData: null,
  srcFormat: null,
  dstPath: undefined,
  quality: 0.8,
  format: 'jpg',
  progressive: false,
  width: 0,
  height: 0,
  strip: true,
  filter: 'Lagrange',
  sharpening: 0.2,
  customArgs: []
}
```

结合本应用设计的需要，我们只应用 `options` 中的 `srcPath`、`dstPath`、`width` 和 `height` 参数，分别对应于图片源路径、图片处理后存储路径、图片压缩宽和图片压缩高。`resizeImage` 方法的具体实现代码如下：

```
function resizeImage(url, width, height, newName, callback){
  var im = require('imagemagick');
  im.resize({
    srcPath: url,
    dstPath: newName,
    width: width,
    height: height
  }, function(err, stdout, stderr){ //回调返回执行结果
    if (err){
      callback(-1);
      console.log(err);
    } else {
      console.log(stdout);
      callback(stdout);
    }
  });
}
```

【代码说明】

- ❑ `var im = require('imagemagick')`: 获取 `imagemagick` 图片处理模块对象。
- ❑ `im.resize...`: 调用 `resize` 方法实现图片的压缩处理。
- ❑ `function(err, stdout, stderr)`: 压缩完成后，回调处理返回结果。
- ❑ `callback(-1)`: `error` 异常时，返回-1表示执行出现错误。
- ❑ `callback(stdout)`: 返回图片处理结果。

该函数包含了4个执行参数，url 是源图片路径，width 是压缩宽，height 为压缩高，由于 resize 是一个异步处理方法，因此需要使用 callback 回调函数来返回异步执行结果。

resizeImage 实现后，我们再来梳理一下 UDP Server 端需要处理的逻辑。UDP Server 的启动监听、Clinet 连接 Server 端，并发送相应的图片处理参数，Server 接受参数，并调用 resizeImage 方法实现图片处理后，通过 UDP 数据传输执行结果到 Clinet。了解以上 UDP Server 需要处理的过程后，我们再来看一下其实现代码，如下为 UDP Server 启动 UDP 服务器的代码：

```
var dgram = require("dgram");
var server = dgram.createSocket("udp4");
/* 监听 listening 消息事件 */
server.on("listening", function () {
    var address = server.address();
    console.log("server listening " + address.address + ":" + address.port);
});
server.bind(41234);
```

这部分代码和 4.1 节的 UDP Server 实现是一样的，这里就不再详细介绍。我们主要看一下 server.on("message", function) 函数的实现，其需要接收客户端的数据，并调用 resizeImage 处理图片，处理完成后通过 UDP 的 server send 方法返回相应的处理结果。其具体实现代码如下：

```
server.on("message", function (msg, rinfo) {
    var imageObject = JSON.parse(msg);
    /* 调用 resizeImage 方法，实现图片压缩 */
    resizeImage(imageObject.url, imageObject.width, imageObject.height,
        imageObject.new_name, function (ret) { // 匿名回调函数
            var retJson;
            if (ret == -1) { // 压缩图片错误时，返回错误信息
                retJson = { 'code': -1, 'msg': 'some error in resize image',
                    'data': {} };
            } else { // 压缩图片成功时，返回成功信息
                retJson = { 'code': 0, 'msg': 'success', 'data': { 'name':
                    imageObject.new_name } };
            }
            /* 将 json 对象转化为 json 字符串，便于数据传输 */
            var retStr = JSON.stringify(retJson);
            var message = new Buffer(retStr);
            /* 回复字符数据到发送数据的客户端 */
            server.send(message, 0, message.length, rinfo.port, rinfo.
                address);
        });
});
```

【代码说明】

- ❑ var imageObject = JSON.parse(msg): 将 json 数据转化为对象。
- ❑ resizeImage(imageObject.url...): 调用 resizeImage 方法，根据相应参数压缩图片。
- ❑ var retStr = JSON.stringify(retJson): 将响应数据对象转化为 json 字符串，便于传递。
- ❑ server.send(message...): 响应数据到客户端。

为了便于数据的传递，我们将 message 信息转化为 json 字符串，因此在读取 message 信息和发送 message 时，都需要做相应的数据类型转化。JavaScript 提供了 JSON.parse 来

将字符串转化为 json 对象，JSON.stringify 将 json 对象转化为 json 字符串。

server.on("message",callback(msg, rinfo))回调函数中有 msg 和 rinfo 参数，其中 msg 为客户端发送的消息数据，而 rinfo 则为客户端信息，服务器端根据客户端信息中的端口 port 和 IP 地址 address，应用 server.send 响应数据到客户端即可。到这里我们就实现了一个图片处理的 UDP 服务器，接下来介绍 Web 服务器端是如何与其交互的。

4.2.3 UDP Client 端（Web Server）

Web Server 主要是应用 3.7 节中的“文件上传”实例的框架，其中包含了基本的路由处理和静态服务器功能。我们着重看下该 Web Server 中的 controller (upload.js) 的实现细节。

Web Server 主要有 3 个功能：图片上传处理页面、图片上传功能业务逻辑处理和与 UDP Server 之间的交互，根据这 3 个部分，我们将 upload.js 的代码框架设计如下：

```
module.exports = function(){
  var _res = arguments[0];
  var _req = arguments[1];
  /**
   *
   * @desc HTTP 响应文件上传页面
   */
  this.uploadPage = function(){
  };
  /**
   *
   * @desc 文件上传处理逻辑
   */
  this.uploadAction = function(){
  };
  /**
   *
   * @desc 图片压缩处理函数
   */
  function imageResize(){
  }
}
```

【代码说明】

- ❑ var _res = arguments[0]: 获取函数的第一个参数 res 对象。
- ❑ var _req = arguments[1]: 获取函数的第二个参数 req 对象。
- ❑ this.uploadPage: 图片上传页面访问接口。
- ❑ this.uploadAction: 图片上传和图片压缩逻辑处理。
- ❑ function imageResize: 与 UDP Server 交互压缩图片（UDP Client）。

this.uploadPage 根据框架中提供的方法 _res.render 即可指定显示相应的 jade 模板文件。其实现代码如下：

```
this.uploadPage = function(){
  _res.render(VIEW + 'index.jade');
};
```


imageResize 函数的主要功能是应用 UDP 模块连接 UDP Server, 将相应的参数数据转化为 json 字符通过 UDP 协议传递到 UDP Server, 并将 UDP Server 响应的数据通过 res.render 直接返回显示到相应的页面。其具体实现如下:

```
/**
 *
 * @desc 对图片的压缩处理
 * @parameters width 图片压缩的宽度
 * @parameters height 图片压缩的高度
 * @parameters imagePath 图片源地址
 * @parameters newName 图片处理后的路径名称
 */
function imageResize(width, height, imagePath, newName){
    var imageJson = {
        'width' : width,
        'height' : height,
        'url' : imagePath,
        'new_name' : newName
    };
    var jsonStr = JSON.stringify(imageJson);
    var client = lib.dgram.createSocket("udp4");
    var message = new Buffer(jsonStr);
    /* 应用 UDP 客户端发送信息到 UDP 到服务端 */
    client.send(message, 0, message.length, 41234, "127.0.0.1",
    function(){
        client.on("message", function (msg, rinfo) {
            var retJson = JSON.parse(msg);
            if(retJson.code == 0){ //成功处理后响应正确信息到 WEB 客户端
                console.log(pathName);
                res.render(VIEW + 'main.jade', {'url' : pathName,
                    'err':'ok'});
            } else {
                res.render(VIEW + 'main.jade', {'url' : '',
                    'err':'error'});
            }
        })
    });
}
```

【代码说明】

- ❑ var jsonStr = JSON.stringify(imageJson): 将 imageJson 对象转化为 json 字符, 便于 UDP 协议之间的数据传输。
- ❑ var client = lib.dgram.createSocket("udp4"): 创建 UDP Client 对象。
- ❑ client.send(message, 0, message.length, 41234, "127.0.0.1",...: 将 json 字符应用 client.send 将数据传递到 UDP Server 端。
- ❑ client.on("message", function (msg, rinfo)...: 监听 UDP Server 传递的消息。
- ❑ var retJson = JSON.parse(msg): 将 json 字符 msg 转化为 json 对象。

imageResize 函数的带有 4 个 width、height、imagePath 和 newName, 分别对应于压缩宽、压缩高、源图片路径和新文件路径。客户端应用 client.on 的 message 事件来监听服务器是否有消息传递过来。由于我们这里选用的是 json 方式传递数据, 因此对 message 消息需要做一定的数据类型转化。当然, 如果为了安全考虑的话, 还可以加发送的消息进行相

应的加密，避免抓包后被他人获取。页面数据根据 UDP Server 响应的信息的不同而展示不同数据，因此 `_res.render` 返回的数据有所不同。

接下来我们看一下文件上传功能的实现。这部分在 3.7 节中已经介绍过，本节就会简单介绍一下，示例代码如下：

```
this.uploadAction = function() {
    var now = Date.parse(new Date())/1000;
    var form = new lib.formidable.IncomingForm(),
        fields = [],
        baseName = BASE_DIR + '/uploadFile/' + now;
    imageName = baseName + '.png',
    newName = baseName + '_small' + '.png',
    pathName = '/uploadFile/' + now + '_small' + '.png';
    /* 根据 form 表单获取其中的 HTTP 参数 */
    form.on('field', function(field, value) {
        fields.push([field, value]);
    });
    form.parse(_req, function(error, fields, files) {
        var size = '' + fields.width + 'x' + fields.height;
        lib.fs.renameSync(files.image.path, imageName);
        imageResize(fields.width, fields.height, imageName, newName);
    });
};
```

文件上传功能同样是应用 `formidable` 模块，当然这里还应用到其获取 POST 数据的方法。`formidable` 模块提供了获取 field 参数的 API `form.on` 的 field 事件，监听 POST 数据传递。所有的 POST 数据都需要应用 `form.parse` 进行解析，解析返回 `fields` 对象和文件对象 `files`。根据获取的 `width` 和 `height`，调用 `imageResize` 对图片进行相应的压缩处理。至此我们已经将所有的 Web Server 需要处理的业务逻辑功能介绍完了，最后我们再看一下 `index.jade` 和 `main.jade` 的相关代码。

4.2.4 Jade 页面实现

本项目主要包含两个 Web 页面：文件上传页面和压缩图片展示页面，分别对应于 `index.jade` 和 `main.jade`。为了迅速地开发，这里应用了 `bootstrap` 的前端 UI 框架。因此在 `header.jade` 中添加了 `bootstrap` 的 CSS 文件引入，其中的 `header.jade` 代码如下所示。

`Bootstrap` 是 `Twitter` 推出的一个开源的用于前端开发的工具包。它由 `Twitter` 的设计师 `Mark Otto` 和 `Jacob Thornton` 合作开发，是一个 CSS/HTML 框架。`Bootstrap` 提供了优雅的 HTML 和 CSS 规范，它是由动态 CSS 语言 `Less` 写成。`Bootstrap` 一经推出后颇受欢迎，一直是 `GitHub` 上的热门开源项目，包括 `NASA` 的 `MSNBC`（微软全国广播公司）的 `Breaking News` 都使用了该项目¹。

```
html(lang="en")
  head
    meta(charset="utf-8")
    title image upload
    meta(name="viewport", content="width=device-width, initial-scale=1.0")
```

1 来自百度百科：<http://baike.baidu.com/view/1489977.htm#2>。

```

    meta(name="description", content="")
    script(src="static/js/jquery-1.8.3.min.js")
    script(src="static/js/socket.js")
    link(href="static/js/bootstrap/css/bootstrap.css", rel="
    stylesheet")
    link(href="static/css/style.css", rel="stylesheet")
  body
    div#container

```

【代码说明】

❑ `link(href="static/js/bootstrap/css/bootstrap.css",rel="stylesheet")`: 导入 bootstrap 的核心 CSS 文件。

header 主要是引入一些公用的 CSS 和 JavaScript 库, 以及一些 html header 信息, 接下来在 index.jade 和 main.jade 中只需要 include 即可。如下是 index.jade 的代码实现:

```

include header
form.form-signin(action='upload?c=uploadAction',
method='post', ENCTYPE='multipart/form-data')
  h2.form-signin-heading IMAGE SIZE
  input.input-block-level(type="text", placeholder="width",
name="width")
  input.input-block-level(type="text", placeholder="height",
name="height")
  input(type="file", name="image")
  button.btn.btn-large.btn-primary(type="submit")
    | UPLOAD IMAGE
include footer

```

其中应用到了大部分 bootstrap 的 class 属性, 包括 input-block-level 和 .btn.btn-large.btn-primary 等。index.jade 主要是展示一个 Web 表单, 其包含 width 和 height 的文本输入, 以及文件上传功能。相对于 index.jade 来说, main.jade 比较简单, 代码如下:

```

include header
div #{err}
  img(src='#{url}')
include footer


```

需要大家注意的是, 对于 img 的 src 属性使用 jade 的变量时, 必须使用引号括起来, 否则会在 src 属性的前后添加两个 undefined 字符, 例如 jade 中的 url 为 /home/hi.jpg, 如果未加引号, 展示在页面上时将会是 ``。

至此我们已经实现了本应用的功能, 包括 UDP Server 端、Web Server 端和页面都已经完成。接下来一节我们将着重介绍本应用的一些体验, 以及本项目开发中需要了解和掌握的知识点。

4.2.5 应用体验

本系统的代码在相应的源码库的 chapter_four 的 4.2 文件夹中, 其中包含了 Web Server 和 UDP Server 两部分代码。

 **注意:** 由于 imagemagick 图片处理工具的局限性, 我们只能在 Linux 运行 (Max 下也可以运行, 需要安装相应的 imagemagick 工具软件)。

打开终端窗口运行 4.2 文件夹中的 app.js 文件, 执行指令:

```
node app.js
```

打开另外一个终端窗口运行 4.2 文件夹中的 `udp_server` 文件夹下的 `dgram.createSocket.js`, 执行指令:

```
node dgram.createSocket.js
```

可以看到如图 4-6 所示的提示, 表明成功启动 UDP Server。

```
root@ubuntu:/home/danhuang/4.2/udp_server# node dgram.createSocket.js
server listening 0.0.0.0:41234
```

图 4-6 udp 服务运行日志信息

成功运行两个 Server 以后, 我们打开 Web 浏览器, 输入 `http://127.0.0.1:8000` 进入图片上传页面 (bootstrap 页面在 IE 下可能会有异常, 建议使用非 IE 浏览器, 如 Chrome 或者 Firefox), 页面如图 4-7 所示。



图 4-7 应用 Web 页面

进入页面后, 我们输入相应的 `width` 为 200 和 `height` 为 200, 并上传一张尺寸为 768*480, 大小为 1.05MB 的图片, 单击 `UPLOAD IMAGE` 后, 再来看一下图片是否按照我们的参数进行了相应的压缩。单击 `UPLOAD IMAGE` 后, 可以看到如图 4-8 所示的返回。

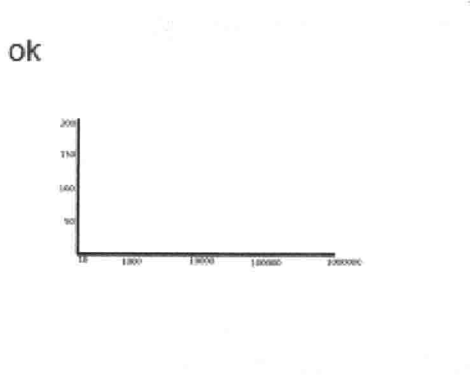


图 4-8 应用图片上传结果

通过查看页面图片的属性，我们可以看到如图 4-9 所示的数据信息。

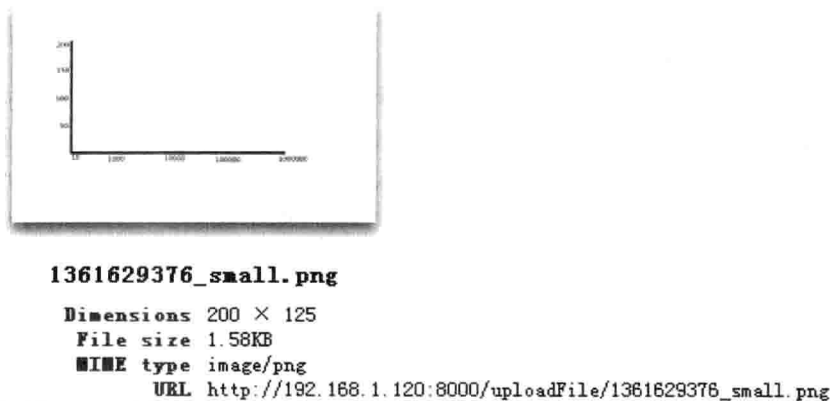


图 4-9 应用图片返回具体信息

其中被压缩后的图片为 200*125，图片大小变为 1.58KB，可以看到其做了相应的压缩。由于这里是等比例压缩，因此图片尺寸大小由原来的 768*480 变为了 200*125，而不是 200*200。

本应用可以应用多个场景，在软件开发过程中，我们经常需要对图片进行压缩，降低用户在移动客户端产生的流量。本应用使用的是 Node.js 的 Web Server 作为 UDP Client 端，在实际应用开发中可以应用其他 UDP Client 来连接，并做图片的压缩处理。希望读者能够通过本章的学习来完善该应用，从而能够灵活的处理一些图片压缩功能，也希望有一天我可以在 github 上看到大家出色的图片处理 UDP Server。本应用中的 imagemagick NPM 模块还提供了多种图片处理方法，大家可以前往 github 主页学习，源码中笔者只使用其 resize 接口来做示范。

4.3 Node.js 与 PHP 合作

现在我们有一个项目，其原来一直都是使用 PHP 的，但现在我们希望将其中的某一个功能使用 Node.js 来完成，因此这就需要解决一个问题，Node.js 与 PHP 如何进行交互。当然，这不仅仅是 PHP 和 Node.js 的问题，其他语言和 Node.js 的交互也是一样的。

Node.js 与 PHP 之间的交互方式很多，包括 4.2 节介绍的 UDP、脚本执行，以及 HTTP GET 和 POST 传递等。每一种方式都有其优缺点，实现方式都有所不同，本节就教你如何灵活应用三者与 PHP 或者其他语言进行合作开发。

4.3.1 UDP 方式

4.2 节中介绍了 Node.js 中如何构建 UDP 服务器以及连接 UDP 服务器的方式，那么接下来我们将应用 Node.js 来构建 UDP 服务器，并使用 PHP 端来连接该 UDP 服务器，通过 UDP 来进行数据传输，达到两者交互的目的。当然，也可以使用 PHP 来构建 UDP 服务器，

而使用 Node.js 来连接 UDP 服务器。两者的区别在于你希望谁来作为服务对象，例如，想在 PHP 中调用 Node.js 的图片处理功能，那么 Node.js 就是一个服务，因此我们将其作为服务器，而 PHP 相对就是客户端了。

在介绍该功能之前，首先需要安装一下 PHP 的运行环境，apache+wampserver，具体配置使用方法可以参照 <http://www.iplaysoft.com/wamp-server.html>。Linux 下需要安装 PHP 和 Apache，相关安装方法参照 <http://www.cnblogs.com/lufangtao/archive/2012/12/30/2839679.html>。如果你不了解 PHP 语法也没关系，只需要了解如何运行 PHP 代码即可。

首先创建一个 Node.js 的 UDP 服务器，代码如下：

```
/*nodeserver.js*/
var dgram = require("dgram");
var server = dgram.createSocket("udp4");

//get message from client, and send back result message
server.on("message", function (msg, rinfo) {
    var msgJson = JSON.parse(msg);
    var ret = 'my name is ' + msgJson['name'] + ', and my book is ' +
        msgJson['book'];
    var message = new Buffer(ret);
    server.send(message, 0, message.length, rinfo.port, rinfo.address);
});
/* 监听 listening 消息事件 */
server.on("listening", function () { //监听 listening 方法
    var address = server.address();
    console.log("server listening " + address.address + ":" + address.port);
});

server.bind(41234);
```

上面代码和之前的 UDP 服务器创建类似，只是在服务器端接收到客户端发送的 message 后，通过 server.send 将处理后的数据响应到客户端。这部分代码只是做了一个简单的字符处理，实际应用中可以灵活地应用 Node.js 的异步功能来处理一些复杂的逻辑。

接下来我们使用 PHP 来连接该 UDP 服务器，并传递相应的数据到服务器端，本部分为 PHP 知识，读者了解一下。其主要应用 PHP 的 stream_socket_server、stream_socket_accept、stream_socket_recvfrom 和 stream_socket_sendto 函数，如表 4.1 所示。

表 4.1 PHP Socket API

函 数 名	函 数 作 用	参 数 以 及 说 明
stream_socket_server	创建 socket 连接	socket url
stream_socket_accept	接收一个连接	stream_socket_server 返回的对象
stream_socket_recvfrom	获取服务器数据	stream_socket_accept 返回对象、数据长度和数据类型
stream_socket_sendto	发送数据到 socket 服务端	stream_socket_accept 返回对象、数据和数据类型

接下来我们应用这 4 个函数实现 UDP 客户端连接和数据发送功能，代码如下：

```
<?php
/* phpclient.php */
$sendArr = array();
```

```
$sendArr['name'] = 'danhuang';
$sendArr['book'] = 'node.js';
$str = json_encode($sendArr);
$socketUrl = "udp://127.0.0.1:41234";
$socket = stream_socket_client($socketUrl);
stream_socket_sendto($socket, $str, STREAM_OOB);
$ret = stream_socket_recvfrom($socket, 1500, STREAM_OOB);
echo $ret;
```

连接本地 UDP 服务器 127.0.0.1:41234, 并 stream_socket_sendto 发送 json 数据, 再应用 stream_socket_recvfrom 接收数据返回结果。

运行 nodeserver.js 脚本文件, 然后运行如下指令:

```
php phpclient.php
```

这样就实现了通过 UDP 服务来实现 Node.js 与 PHP 的交互。

4.3.2 脚本执行

脚本执行方式主要是应用 Node.js 来执行 PHP 的脚本文件, 通过执行 shell 命令来运行 PHP 的文件。同样也可以应用 PHP 来执行 shell 命令调用 Node.js 的服务。可以应用 Node.js 作为一个简单的 crontab 使用。

这部分实现较为简单, 只需要使用 Node.js 的 setInterval 和 child_process 模块。例如, 我们希望每五分钟执行一次 phpscript.php 脚本文件, 如下代码所示。

```
var spawn = require('child_process').spawn;
/* 定时执行 PHP 脚本 */
setInterval(function() {
    var phpShell = spawn('php', ['phpscript.php']);
    phpShell.stdout.on('data', function (data) {
        console.log('out put: \n' + data);
    });
}, 3000000);
```

【代码说明】

- ❑ require('child_process').spawn: 调用 child_process 模块的 spawn 对象;
- ❑ phpShell = spawn('php', ['phpscript.php']): 应用 spawn 对象执行 shell 命令;
- ❑ setInterval: 执行定时任务, 每五分钟执行一次。

在 Node.js 中, child_process 模块可以执行 shell 命令。其中的 spawn 方法第一个参数调用的是命令, 第二个参数是一个数组, 该数组为执行命令携带的参数, 可以为多个元素。这种方式较为简单, 但无法传递参数到 PHP 中, 因此这种方式只能应用在那些 Node.js 项目中无需获取返回结果的交互中。

4.3.3 HTTP 方式

将 Node.js 和 PHP 分别作为两个 server, 利用 Node.js 的 request 模块或者 PHP 的 curl 模块来实现 API 方式的调用。

例如腾讯微博中没有提供 Node.js 的 sdk, 因此对于 Node.js 来说需要自己去实现该过

程。如果独立去写一套腾讯微博的相关 sdk 代价将会很大,那么我们可以考虑其他的实现方式。例如腾讯微博虽然没有提供 Node.js 的 sdk,但其提供了 PHP 的 sdk,因此我们可以通过 PHP 暴露接口给 Node.js 来调用,通过 Node.js 传递相应的 token 信息和微博内容,PHP 来发表微博,发表完成后,通过 HTTP 响应到 Node.js。这个过程可以使用如图 4-10 所示的流程图来表示。

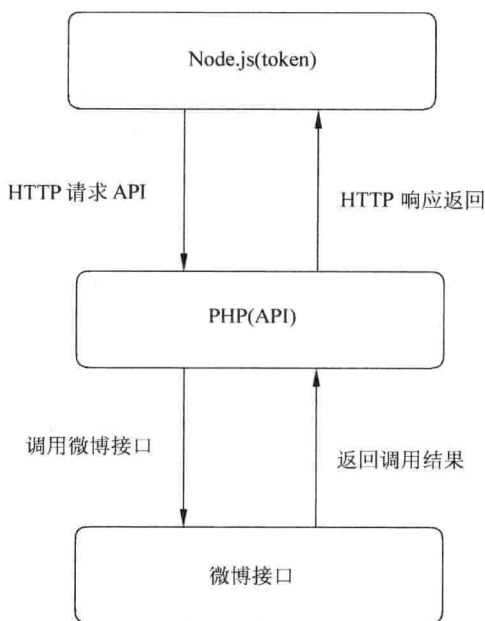


图 4-10 Node.js API 方式与 PHP 交互

具体实现方法这里就不再详细介绍,读者可以下载 iweibo 的一套关于 PHP 的代码,然后暴露一个 HTTP 的接口给 Node.js 来实现。

4.3.4 三种方式的比较

本节介绍了三种 Node.js 与 PHP 或者其他语言交互的方法,但三者各有各的优点和缺点。

对于脚本调用方法,其主要特点是提供一个事件驱动的功能,它并不管驱动之后的结果。因此有时候可以尝试利用 Node.js 来替代 Linux 中的 crontab 功能,在此种场景中这种方式非常有效,但必须注意的是,这种方法无法传递 PHP 或者其他语言的参数,同时其必须是 PHP 和 Node.js 项目在同一台服务主机上。

HTTP 调用方法的特点是必须有一方提供 HTTP 的 API,并且所有的交互都是通过 HTTP 的 TCP 连接方式。这种方式的缺点就在于 API 接口的暴露有可能被第三方调用,因此必须做相应的安全处理,由于利用的是 HTTP,所以在网络耗时上会比其他两种方式更久。

UDP 方式,这种方式较为成熟,可以应用于很多项目中,并且安全可行。上面的两种方式可以应用到一些小的项目或者小功能上,而 UDP 方式,则应用于大项目中的核心功能

模块中。

4.4 本章实践

应用本章知识，完善 UDP 图片处理 Server，在原有的 `resizeImage` 接口上增加 `imagemagick` 模块的其他图片处理方法，并且通过 Web Server 提交图片处理请求。

分析：在 `upload.js` 中添加 `cropImage` 方法，同时在 UDP server 中新增 `crop` 处理方法。
`upload.js` 修改后代码如下：

```
module.exports = function(){
  var _res = arguments[0];
  var _req = arguments[1];
  /* 文件上传 web 页面响应接口 */
  this.uploadPage = function(){
    _res.render(VIEW + 'index.jade');
  };
  /* 文件上传逻辑处理接口 */
  this.uploadAction = function(){
    var now = Date.parse(new Date())/1000;
    var form = new lib.formidable.IncomingForm(),
        fields = [],
        baseName = BASE_DIR + '/uploadFile/' + now;
    imageName = baseName + '.png',
    newName = baseName + '_small' + '.png',
    pathName = '/uploadFile/' + now + '_small' + '.png';
    /* 获取客户端表单提交的数据 */
    form.on('field', function(field, value) {
      fields.push([field, value]);
    });
    form.parse(_req, function(error, fields, files) {
      var size = '' + fields.width + 'x' + fields.height;
      lib.fs.renameSync(files.image.path, imageName);
      switch(fields.type){
        case 1: imageResize(fields.width, fields.height, imageName,
          newName);
          break;
        case 2: imageCrop(fields.width, fields.height, imageName,
          newName);
          break;
        default:
          imageResize(fields.width, fields.height, imageName,
            newName);
          break;
      }
    });
  };
};

/**
 *
 * 图片处理，根据 width 和 height 来压缩图片
 * @params width int 压缩图片宽度
 * @params height int 压缩图片高度
 * @params imagePath string 源文件路径
 * @params newName string 压缩后文件名
 */
```

```

function imageResize(width, height, imagePath, newName){
    var imageJson = {
        'width'    : width,
        'height'   : height,
        'url'      : imagePath,
        'new_name' : newName
    };
    var jsonStr = JSON.stringify(imageJson);
    var client = lib.dgram.createSocket("udp4");
    var message = new Buffer(jsonStr);
    client.send(message, 0, message.length, 41234, "127.0.0.1",
    function(){
        client.on("message", function (msg, rinfo) {
            var retJson = JSON.parse(msg);
            if(retJson.code == 0){
                console.log(pathName);
                _res.render(VIEW + 'main.jade', {'url' : pathName,
                'err':'ok'});
            } else {
                _res.render(VIEW + 'main.jade', {'url' : '',
                'err':'error'});
            }
        })
    });
}
/**
 *
 * 图片剪切函数
 * @params width int 压缩图片宽度
 * @params height int 压缩图片高度
 * @params imagePath string 源文件路径
 * @params newName string 压缩后文件名
 */
function imageCrop(width, height, imagePath, newName){
    var imageJson = {
        'width'    : width,
        'height'   : height,
        'url'      : imagePath,
        'new_name' : newName
    };
    var jsonStr = JSON.stringify(imageJson);
    var client = lib.dgram.createSocket("udp4");
    var message = new Buffer(jsonStr);
    /* 调用 UDP 服务器来处理图片剪切功能 */
    client.send(message, 0, message.length, 41234, "127.0.0.1",
    function(){
        client.on("crop", function (msg, rinfo) {
            var retJson = JSON.parse(msg);
            if(retJson.code == 0){
                console.log(pathName);
                _res.render(VIEW + 'main.jade', {'url' : pathName,
                'err':'ok'});
            } else {
                _res.render(VIEW + 'main.jade', {'url' : '',
                'err':'error'});
            }
        })
    });
}
}

```

udp_server 文件夹下的 dgram.createSocket.js 修改代码如下:

```

/*dgram.createSocket.js*/
var dgram = require("dgram");
var server = dgram.createSocket("udp4");
server.on("message", function (msg, rinfo) {
    var imageObject = JSON.parse(msg);
    /* 调用 resizeImage 方法来处理图片的压缩功能 */
    resizeImage(imageObject.url, imageObject.width, imageObject.height,
        imageObject.new_name, function(ret){
            var retJson;
            if(ret == -1){
                retJson = {'code':-1, 'msg':'some error in resize image',
                    'data':{}}
            } else {
                retJson = {'code':0, 'msg':'success','data':
                    {'name':imageObject.new_name}}
            }
            var retStr = JSON.stringify(retJson);
            var message = new Buffer(retStr);
            server.send(message, 0, message.length, rinfo.port,
                rinfo.address);
        });
});

server.on("crop", function (msg, rinfo) {
    var imageObject = JSON.parse(msg);
    resizeImage(imageObject.url, imageObject.width, imageObject.height,
        imageObject.new_name, function(ret){
            var retJson;
            if(ret == -1){
                retJson = {'code':-1, 'msg':'some error in resize image',
                    'data':{}}
            } else {
                retJson = {'code':0, 'msg':'success','data':{'name':
                    imageObject.new_name}}
            }
            var retStr = JSON.stringify(retJson);
            var message = new Buffer(retStr);
            server.send(message, 0, message.length, rinfo.port,
                rinfo.address);
        });
});

server.on("listening", function () {
    var address = server.address();
    console.log("server listening " + address.address + ":" + address.port);
});
server.bind(41234);

function resizeImage(url, width, height, newName, callback){
    var im = require('imagemagick');
    im.resize({
        srcPath: url,
        dstPath: newName,
        width: width,
        height: height
    }, function(err, stdout, stderr){
        if (err){
            callback(-1);
            console.log(err);
        }
    });
}

```

```

    } else {
        console.log(stdout);
        callback(stdout);
    }
    });
}

function cropImage(url, width, height, newName, callback){
    var im = require('imagemagick');
    im.crop({
        srcPath: url,
        dstPath: newName,
        width: width,
        height: height
    }, function(err, stdout, stderr){
        if (err){
            callback(-1);
            console.log(err);
        } else {
            console.log(stdout);
            callback(stdout);
        }
    });
}

```

在 jade 代码中新增一个图片剪切功能，代码如下：

```

include header
    form.form-signin(action='upload?c=uploadAction', method='post',
        ENCTYPE='multipart/form-data')
        h2.form-signin-heading IMAGE SIZE
        input.input-block-level(type="text", placeholder="width",
            name="width")
        input.input-block-level(type="text", placeholder="height",
            name="height")
        div
            input.input-block-level(type="radio", value="1", name=
                "type")
            | 比例缩减
        div
            input.input-block-level(type="radio", value="2", name=
                "type")
            | 剪切
        input(type="file", name="image")
        button.btn.btn-large.btn-primary(type="submit")
            | UPLOAD IMAGE
include footer

```

分别执行如下命令，启动服务，打开浏览器输入 <http://127.0.0.1:8000> 实现图片压缩和剪切功能。

```

node app.js
node dgram.createSocket.js

```

4.5 本章小结

本章介绍了一些 UDP 的相关知识，并通过实践应用学习 Node.js 的 UDP 开发。本节主要

概述 Node.js UDP 项目开发中一些常用知识点，以及本章 4.1 节和 4.2 节中的一些知识总结。

Node.js 在大多读者的认识中是一个 Web 编程语言，很少有人会了解其网络编程相关知识。但是随着项目的不断壮大以及团队的需要，会经常和后台服务器进行一些网络交互，因此掌握 Node.js 网络编程也是非常重要的一课。

读者需要掌握 Node.js 的 UDP 模块的应用，包括创建 UDP Server、UDP Client，以及 Server 和 Client 之间如何进行信息交互。表 4.2 是本章代码实践中的几个重要 API。

表 4.2 本节的重要API

API	参 数	返 回	备 注
dgram.createSocket	udp4、udp6	callback	
Server.on/Client.on	message	callback(msg, rinfo)	注意，其中的消息是通过 buffer 传递，因此字符串必须转化为 Buffer 数据再传递
	listening	callback	
	close		
	error		
Server.send/Client.send	buff	new Buffer(string)	
	offset	偏移量	
	length	Buffer 长度	
	port	消息发送端口	
	address	消息发送 IP 地址	

对于表格中的 API，需要说明的是，在应用 Server.send 或者 Client.send 时，可以通过多种网络协议方式传递数据。本章的应用为了简单起见，只使用了 json 格式，当然还可以通过其他多种方式传递其他数据结构。例如希望传递数组、Int 或者其他数据结构，只要 Server 端和 Client 端的网络协议一致即可。

网络协议的定义：为计算机网络中进行数据交换而建立的规则、标准或约定的集合。例如，网络中一个微机用户和一个大型主机的操作员进行通信，由于这两个数据终端所用字符集不同，因此操作员所输入的命令彼此不认识。为了能进行通信，规定每个终端都要将各自字符集中的字符先变换为标准字符集的字符后，才进入网络传送，到达目的终端之后，再变换为该终端字符集的字符。当然，对于不相容终端，除了需变换字符集字符外，其他特性，如显示格式、行长、行数、屏幕滚动方式等也需做相应的变换。¹

本章中应用到了一个 imagemagick 模块，该模块主要是让开发者能够更好地对一些图片进行处理（压缩和截取）。前面也介绍过图片的压缩对于移动客户端用户的作用，主要是减少移动客户端用户图片所产生的流量。

imagemagick 模块还有如下一些 API，此处仅供参考，具体请参考 github 官网。如表 4.3 所示。

表 4.3 Imagemagick API

API	描 述	Example
identify(path, callback(err, features))	Identify file at path and return an object features.	<pre>im.identify('kittens.jpg', function(err, features){ if (err) throw err; console.log(features); //{ format: 'JPEG', width: 3904, height: 2622, depth: 8 } });</pre>

1 摘自百度百科：<http://baike.baidu.com/view/860535.htm>。

续表

API	描 述	Example
<code>readMetadata(path, callback(err, metadata))</code>	Read metadata (i.e. exif) in path and return an object metadata. Modelled on top of identify.	<pre>im.readMetadata('kittens.jpg', function(err, metadata){ if (err) throw err; console.log('Shot at '+metadata.exif.dateTimeOriginal); //-> Shot at Tue, 06 Feb 2007 21:13:54 GMT });</pre>
<code>convert(args, callback(err, stdout, stderr))</code>	Raw interface to convert passing arguments in the array args.	<pre>im.convert(['kittens.jpg', '-resize', '25x120', 'kittens-small.jpg'], function(err, stdout){ if (err) throw err; console.log('stdout:', stdout); });</pre>
<code>crop(options, callback)</code>	Convenience function for resizing and cropping an image. crop uses the resize method, so options and callback are the same. crop uses options.srcPath, so make sure you set it :) Using only options.width or options.height will create a square dimensioned image. Gravity can also be specified, it defaults to Center. Available gravity options are [NorthWest, North, NorthEast, West, Center, East, SouthWest, South, SouthEast]	<pre>im.crop({ srcPath: path, dstPath: 'cropped.jpg', width: 800, height: 600, quality: 1, gravity: "North" }, function(err, stdout, stderr){ //foo });</pre>

本章涉及 Node.js 的 API 列表如表 4.4 所示。

表 4.4 本章Node.js的API列表

模 块 名	API 名	作 用	调 用 示 例
UDP	<code>dgram.createSocket(type, [callback])</code>	创建 Socket object	<pre>var s = dgram.createSocket('udp4'); s.bind(1234, function() { s.addMembership('224.0.0.114'); });</pre>
	<code>socket.send(buf, offset, length, port, address, [callback])</code>	发送数据	
	<code>socket.bind(port, [address], [callback])</code>	绑定监听的 IP 地址和端口	
	<code>socket.address()</code>	获取当前 socket 对象的相关的 IP 地址和端口信息	

第 5 章 深入 Node.js

本书的第 3 章和第 4 章介绍的 Node.js 的相关知识点只是停留在 Node.js 的应用层面上，本章将会介绍 Node.js 底层的一些实现，包括 Node.js 的实现框架，以及如何开发 Node.js 的扩展功能模块。希望通过本章的学习，读者能够了解如何调用底层的 Node.js 做一些应用的优化。

学习本章需要简单地了解 Node.js 的一些基类框架实现，掌握应用 C++ 编写 Node.js 扩展的方法，以及扩展编译后使用的方法。

5.1 Node.js 的相关实现机制

Node.js 中有文件模块和原生模块，虽然两种模块的加载机制不同，但两种模块的加载都进行了缓存，因此在调用期间无需考虑多次 `require` 导致的性能问题。原生模块在 Node.js 源代码编译期间，将其以二进制形式保存，因此其加载是速度最快的。而文件模块则是动态加载的方式，相对来说会较慢一些。那么在 Node.js 中两者的加载方式，以及优先级是怎样的呢？

对于开发者来说，加载一个 Node.js 模块很简单，只需要使用一个 `require` 关键字，但其内部的加载原理以及方式是很复杂的。我们首先看一下 Node.js 中模块加载的优先级的流程图，如图 5-1 所示。

当在 Node.js 代码中 `require` 模块时，Node.js 解析逻辑会首先去文件模块缓存中查询是否存在该模块。如果缓存中存在该文件模块时，就直接返回缓存中该文件模块的 `exports` 对象。当缓存数据中不存在该文件模块时，则会判断其是否为原生模块，如果是原生模块，就从原生模块缓存中读取；查询缓存中没有该原生模块时，则添加该模块缓存，并加载原生模块。如果非原生模块，则会根据相应文件路径查找该文件模块，并载入文件模块和缓存，最后返回 `exports` 对象。

虽然原生模块与文件模块的优先级有所不同，但原生模块和文件模块在加载前都会优先从文件模块的缓存中加载已经存在的缓存模块。这里需要大家了解，原生模块的 `require` 仅次于文件模块缓存方式。当缓存中没有模块时，`require` 方法机制在解析文件名之后，优先检查模块是否为原生模块列，如果非原生模块则使用文件模块加载方式加载，在成功加载文件模块后，将文件模块信息缓存到内存中，并返回 `exports` 对象。

关于这点知识的讲解，田永强（朴灵）在《深入浅出 Node.js（三）：深入 Node.js 的模块机制》文章中，经过其充分的分析后，从源代码中整理出了整个文件模块的查找流程，以下是其博文的相关知识内容。

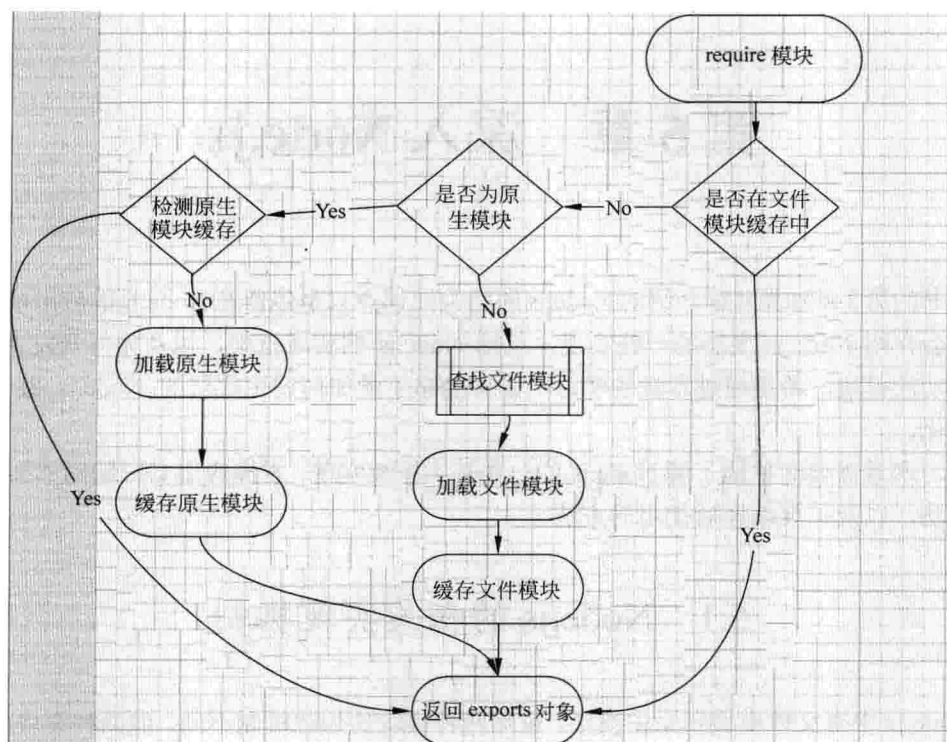


图 5-1 require 加载过程

require 方法接受以下几种类型的参数：原生模块名（例如：http、fs、path 等），文件模块路径（相对路径./mod，绝对路径/path/module）。在进入路径查找之前有必要描述一下 module path 这个 Node.js 中的概念。对于每一个被加载的文件模块，创建这个模块对象的时候，这个模块便会有一个 paths 属性，其值根据当前文件的路径计算得到。我们创建 modulepath.js 这样一个文件，其内容为：

```
console.log(module.paths);
```

我们将其放到任意一个目录中执行 node modulepath.js 命令，将得到以下的输出结果。

```
['/home/jackson/research/node_modules',
'/home/jackson/node_modules',
'/home/node_modules',
'/node_modules']
```

Windows 下：

```
['c:\\nodejs\\node_modules', 'c:\\node_modules']
```

可以看出 module path 的生成规则为从当前文件目录开始查找 node_modules 目录；然后依次进入父目录，查找父目录下的 node_modules 目录；依次迭代，直到根目录下的 node_modules 目录。

除此之外还有一个全局 module path，是当前 node 执行文件的相对目录（../lib/node）。如果在环境变量中设置了 HOME 目录和 NODE_PATH 目录的话，整个路径还包含 NODE_PATH 和 HOME 目录下的 node_modules 与 node_modules。其最终值大致如下：

```
[NODE_PATH, HOME/.node_modules, HOME/.node_modules, execPath/../../lib/node]
```

简而言之，如果 require 绝对路径的文件，查找时不会去遍历每一个 node_modules 目

录，其速度最快。其余流程如图 5-2 所示，相关步骤如下：

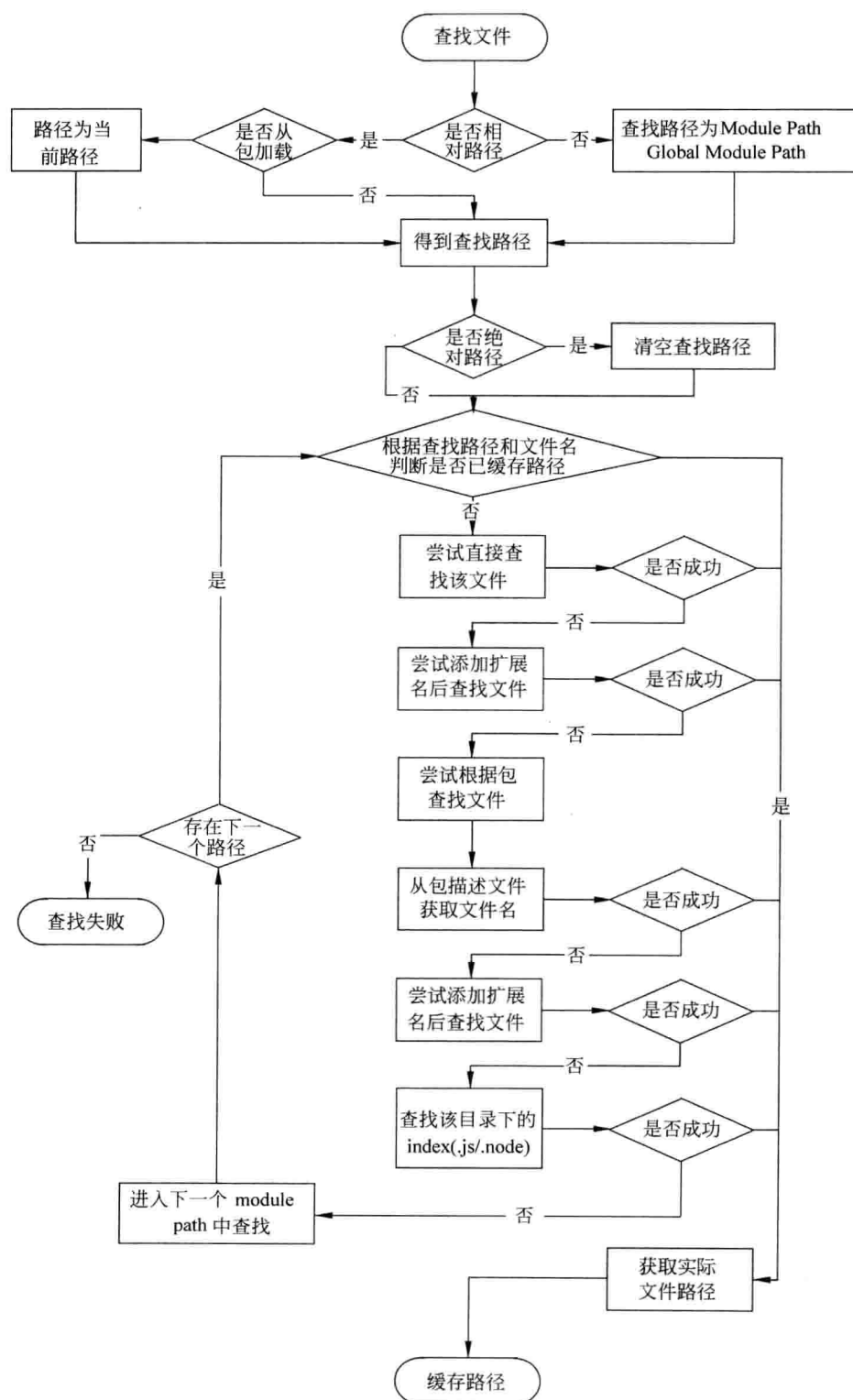


图 5-2 Node.js 文件查询

从 `module path` 数组中取出第一个目录作为查找基准。

直接从目录中查找该文件，如果存在，则结束查找。如果不存在，则进行下一条查找。

尝试添加 `.js`、`.json`、`.node` 后缀后查找，如果存在文件，则结束查找。如果不存在，则进行下一条。

尝试将 `require` 的参数作为一个包来进行查找，读取目录下的 `package.json` 文件，取得 `main` 参数指定的文件。

尝试查找该文件，如果存在，则结束查找。如果不存在，则进行第 3 条查找。

如果继续失败，则取出 `module path` 数组中的下一个目录作为基准查找，循环第 1 至 5 个步骤。

如果继续失败，循环第 1 至 6 个步骤，直到 `module path` 中的最后一个值。

如果仍然失败，则抛出异常。

整个查找过程十分类似原型链的查找和作用域的查找。所幸 Node.js 对路径查找实现了缓存机制，否则由于每次判断路径都是同步阻塞式进行，会导致严重的性能消耗¹。

以上是田永强（朴灵）的原文，其中有一点关于 `module.paths` 的查询机制需要大家掌握，而关于文件模块的查询机制实现的原理，可以做简单的认识。


5.2 Node.js 原生扩展

本节是基于移动博客翻译的一篇《编写 Node.js 原生扩展》（参见网络 <http://www.grati.org/?p=413>）文章而写的，其中大部分的知识点源自该文。基于《编写 Node.js 原生扩展》博文，笔者亲自实践扩展的开发，通过实践编码和读者一起学习 Node.js 原生扩展。本文主要分为 Node.js 原生扩展理论基础与实践和 Node.js 原生扩展实例应用介绍两个部分的内容。

5.2.1 Node.js 扩展开发基础 V8

虽然 JavaScript 可以实现任何 Node.js 平台的一些库，例如我们应用了很多 NPM 模块，但是如果将其中的一些库和系统，应用 C++ 编写其扩展时，会显得更加完美，甚至可以更大地提升该模块的效率。编写 Node.js C++ 扩展很大程度上就像是写 V8 的扩展，这是因为 Node.js 封装了 V8 的接口，大部分时间 Node.js 开发者都是在使用原始的 V8 数据类型和方法。

为了学习 Node.js 的原生扩展，读者首先需要了解 V8 的一些编程基础。

 **注意：**关于 V8 本书只是简单介绍，详细学习知识可参阅 <https://developers.google.com/v8/intro?hl=zh-CN>，链接地址是 V8 的开发文档。下面一段是关于 V8 文档的说明：This documentation is aimed at C++ developers who want to use V8 in their applications, as well as anyone interested in V8's design and performance. This

¹ 田永强《深入浅出 Node.js（三）：深入 Node.js 的模块机制》，参见网络 <http://www.infoq.com/cn/articles/nodejs-module-mechanism>。

document introduces you to V8, while the remaining documentation shows you how to use V8 in your code and describes some of its design details, as well as providing a set of JavaScript benchmarks for measuring V8's performance¹。译文：这份文档是给那些将 V8 应用到他们软件开发中的，和所有对 V8 设计、性能感兴趣的 C++ 开发者。这份文档主要是介绍 V8，其次文档会展示如何应用 V8 代码开发，接下来会介绍一些 V8 的设计细节并提供一组 JavaScript benchmarks 来测量 V8 的性能。

接下来介绍如何应用 V8 的库来做一些简单的实例应用。要应用 V8 的库做 C++ 开发，包含以下几个过程。

首先需要下载 V8 的源码，可以通过 Subversion 取得 V8 的源码。当然要先下载安装 Subversion。

Subversion，简称 SVN，是一个开放源代码的版本控制系统，相对于 RCS 和 CVS，采用了分支管理系统，它的设计目标就是取代 CVS。互联网上越来越多的控制服务从 CVS 转移到 Subversion。Subversion Linux 下安装执行 `apt-get install subversion` 即可，Windows 下可以下载相应的 Subversion 软件。

执行如下指令下载 V8 源码：

```
svn checkout http://v8.googlecode.com/svn/trunk/ v8
```

checkout 代码成功以后，就需要对 V8 的代码进行编译，这里需要借助 GYP 工具²。

依次执行如下指令，MakeFile 包含了多个环境和系统下的编译方式，以及编译时需要选择是发布模式还是 debug 开发模式，因此在编译 V8 时要选择相应的参数进行编译。

32 位 Linux 系统执行如下命令：

```
make ia32.release
```

64 位 Linux 系统执行如下命令：

```
make x64.release
```

编译完成后，可以执行如下命令来做一个简单的测试，代码如下：

```
tools/run-tests.py --arch-and-mode=ia32.release /home
```

执行完成后，就可以看到成功编译结果。成功编译后我们就可以应用 V8 的库来做一些应用的编程开发了。

以上是 V8 官网提供的方法，具体可以参考 <https://developers.google.com/v8/intro?hl=zh-CN>，我们也可以应用 Node.js 中提供的 node-gyp NPM 工具来编写 Node.js 的插件。由于笔者主要是研究 Node.js 的知识，因此没有操作上面的过程，希望大家可以掌握 5.2.2 节中介绍的 node-gyp 编译插件的应用过程。

1 摘自 <https://developers.google.com/v8/intro?hl=zh-CN>。

2 GYP (Generate Your Projects) 是由 Chromium 团队开发的跨平台自动化项目构建工具，功能 GYP 和 CMake 相似。

5.2.2 Node.js 插件开发介绍

上一节中我们提到了 `node-gyp` npm 工具，该工具的主要作用就是应用 V8 源码，编译生成一个 Node.js 可以调用的二进制 “.node” 文件模块。接下来介绍该工具的使用方法和扩展实现过程。

在应用该工具前，我们需要应用 NPM 模块来安装 `node-gyp` 工具模块库，执行如下命令：

```
npm install -g node-gyp
```

安装完成后我们来看一下 `node-gyp` 编译 Node.js 的 C++ 扩展实例。在介绍 Node.js 的 C++ 扩展前，首先我们来熟悉一下 Node.js 中是如何实现一个等价 C++ 的扩展模块。以下代码是一个最简单的 Hello World 程序，它展示了一个简单的 `node` 模块，而其接口也和大多数 C++ 扩展要提供的接口类似，具体代码如下：

```
HelloWorldJs = function() {
  this.m_count = 0;
};
/* 创建 HelloWorldJs 对象的 hello 方法 */
HelloWorldJs.prototype.hello = function()
{
  this.m_count++;
  return "Hello World";
};
/* 暴露 HelloWorldJs 对象 */
exports.HelloWorldJs = HelloWorldJs;
```

以上代码定义 `HelloWorldJs` 对象，然后应用 `prototype` 为该对象添加 `hello` 方法，并通过 Node.js 的 `exports` 将对象暴露给调用者。下面是调用者应用该对象的示例：

```
var helloworld = require('helloworld_js');
var hi = new helloworld.HelloWorldJs();
console.log(hi.hello()); //prints "Hello World" to stdout
```

相信学习了本书前几章的知识以后，对这些代码应该非常熟悉，至于代码的含义，本节无需再详加说明。希望读者可以应用 `module.exports` 重新实现一下以上代码，来进一步了解 `exports` 和 `module.exports` 的区别。

接下来我们看一下 C++ 版本的 Hello World 程序。编写 C++ 扩展前首先要学会 5.1 节中如何编译 V8 源码相关知识。在使用 V8 前我们需要包含 `node.h` 和 `v8.h` 头文件，代码如下：

```
#include <v8.h>
#include <node.h>

using namespace v8;
Handle<Value> Method(const Arguments& args) {
  HandleScope scope;
  return scope.Close(String::New("world"));
}

/* 初始函数 */
```

```
void init(Handle<Object> target) {
    target->Set(String::NewSymbol("hello"),
        FunctionTemplate::New(Method)->GetFunction());
}
NODE_MODULE(hello, init)
```

该代码就是拓展一个 `hello` 方法，该 `hello` 方法打印一个 `world` 字符串。简单的模块代码实现完成以后，就需要应用工具 `node-gyp`，将 C++ 源码编译生成可调用的 Node.js 模块。`node-gyp` 编译时，需要一个 `building.gyp` 配置文件，配置代码如下：

```
{
  "targets": [
    {
      "target_name": "hello",
      "sources": [ "hello.cc" ]
    }
  ]
}
```

以上指定编译的源文件路径为 `hello.cc`，以及生成的文件名 `hello`。创建好 C++ 扩展代码后，我们使用 `node-gyp` 来对扩展代码进行编译，依次运行如下指令：

```
node-gyp configure
node-gyp build
```

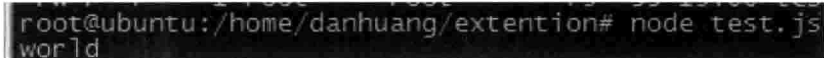
运行完成以后会在该目录下的 `build/Release` 文件夹下生成一个 `hello.node` 二进制文件。接下来我们使用如下测试代码，来测试是否已成功创建扩展。

```
var Hello = require('./build/Release/hello.node');
console.log(Hello.hello());
```

代码中第一步是 `require` 通过 `node-gyp` 编程生成的 `hello.node` 模块，`require` 该模块成功后会返回一个 JavaScript 对象。根据上面的扩展代码，该 `require` 返回的对象包含一个 `hello` 方法，因此通过运行该测试代码后会打印一个 `world` 字符，运行如下命令来查看测试结果。

```
node test.js
```

执行结果如图 5-3 所示。



```
root@ubuntu:/home/danhuang/extention# node test.js
world
```

图 5-3 test 脚本运行结果

这样就简单实现了 Node.js 的一个 C++ 扩展。以上是利用 `node-gyp` 工具进行编译的，当然也可以直接通过编译 Node.js 源码添加扩展，具体方法可以参考《编写 Node.js 原生扩展》¹ 博客。

5.3 Node.js 异步扩展开发与应用

5.2 节中简单介绍了 Node.js 的一些同步接口的开发应用实例。这里为了体现 Node.js

¹ 参见网络 <http://www.grati.org/?p=413>。

异步扩展的重要性，本节将进一步介绍 Node.js 的异步扩展实现和应用实例。

对于实际的应用来说，Node.js 主要的优势是提供异步 IO，因此 5.2 节中的 HelloWorld 实例相对来说太过简单了一些。Node.js 内部通过 libeio 将那些会产生阻塞的操作全都放入线程池中去执行。如果需要和遗留的 c 库交互，通常需要使用异步 IO 来为 JavaScript 代码提供回调接口。

Node.js 的异步 API，通常的模式是提供一个回调，在异步操作完成时触发回调事件函数，大家可以在 Node.js 的 API 中看到很多这种调用模式。Node.js 的 filesystem 模块提供了一个很好的例子，其中大多数的函数都在操作完成后，通过调用回调函数来传递数据。和许多传统的 GUI 框架一样，Node.js 只在主线程中执行 JavaScript，因此主线程以外的任何操作都不应该直接和 V8 或 JavaScript 交互。下面代码是由 pquerna 在 github 公布的源代码。

在使用 Node.js 或 V8 之前，我们需要包括相关的头文件 V8.h 和 node.h。代码如下：

```
#include <v8.h>
#include <node.h>

using namespace node;
using namespace v8;
```

在本例子中直接使用了 V8 和 Node.js 的命名空间，使代码更易于阅读。

接下来，需要声明 HelloWorldEio 类。它继承自 node::ObjectWrap 类，这个类提供了几个如引用计数、在 V8 内部传递 contex 等实用的功能。一般来说，所有对象应该继承 ObjectWrap。代码如下：

```
class HelloWorld: ObjectWrap
{
private:
    int m_count;
public:
```

声明类之后，我们定义了一个静态成员函数，用来初始化对象并将其导入 Node.js 提供的 target 对象中。这个函数基本上是告诉 Node.js 和 V8 你的类是如何创建的，以及它将包含什么方法。代码如下：

```
static Persistent<FunctionTemplate> s_ct;
static void Init(Handle<Object> target)
{
    /* 定义 scope 变量 */
    HandleScope scope;

    Local<FunctionTemplate> t = FunctionTemplate::New(New);

    s_ct = Persistent<FunctionTemplate>::New(t);
    s_ct->InstanceTemplate()->SetInternalFieldCount(1);
    s_ct->SetClassName(String::NewSymbol("HelloWorldEio"));

    NODE_SET_PROTOTYPE_METHOD(s_ct, "hello", Hello);

    target->Set(String::NewSymbol("HelloWorldEio"),
                s_ct->GetFunction());
}
```

用 `NODE_SET_PROTOTYPE_METHOD` 宏将 `hello` 方法绑定到该对象。最后，一旦我们建立好这个函数模板后，将它分配给 `target` 对象的 `HelloWorld` 属性，将类暴露给用户。

接下来的部分是一个标准的 C++ 构造函数。

```

HelloWorld() :
    m_count(0)
{
}

~HelloWorld()
{
}

```

在 `New()` 方法中 V8 引擎将调用这个简单的 C++ 构造函数。

```

static Handle<Value> New(const Arguments& args)
{
    HandleScope scope;
    /* 创建 HelloWorld 对象 */
    HelloWorld* hw = new HelloWorld();
    hw->Wrap(args.This());
    return args.This();
}

```

此段代码相当于上面 JavaScript 代码中使用的构造函数。它调用 `new HelloWorld` 创造了一个普通的 C++ 对象，然后调用从 `ObjectWrap` 继承的 `Wrap` 方法，它将一个 C++ `HelloWorld` 类的引用保存到 `args.This()` 的值中。在包装完成后返回 `args.This()`，整个函数的行为和 JavaScript 中的 `new` 运算符类似，返回 `this` 指向的对象。下面介绍在 `Init` 函数中被绑定到 `hello` 的函数，代码如下：

```

static Handle<Value> Hello(const Arguments& args)
{
    HandleScope scope;

    REQ_FUN_ARG(0, cb);

    HelloWorldEio* hw = ObjectWrap::Unwrap<HelloWorldEio>(args.This());
}

```

函数中首先使用 `ObjectWrap` 模板的方法提取出指向 `HelloWorldEio` 类的指针，然后和 JavaScript 版本的 `HelloWorldEio` 一样递增计数器。在 `Hello` 函数的入口处，使用宏从参数列表的第一个位置获取回调函数。然后，使用相同的 `Unwrap` 方法提取指向类对象的指针。

```

hello_baton_t *baton = new hello_baton_t();
baton->hw = hw;
baton->increment_by = 2;
baton->sleep_for = 1;
baton->cb = Persistent<Function>::New(cb);

```

这里我们创建一个 `baton` 结构，并将各种参数保存在里面。请注意，我们为回调函数创建了一个永久引用，因为我们想要在超出当前函数作用域的地方使用它。如果不这么做，在本函数结束后将无法再调用回调函数。代码如下：

```

hw->Ref();

eio_custom(EIO_Hello, EIO_PRI_DEFAULT, EIO_AfterHello, baton);

```

```

    ev_ref(EV_DEFAULT_UC);

    return Undefined();
}

```

下面的代码是真正的重点。首先，我们增加 HelloWorld 对象的引用计数，这样在其他线程执行的时候它就不会被回收。函数 `eio_custom` 接受两个函数指针作为参数。`EIO_Hello` 函数将在线程池中执行，然后 `EIO_AfterHello` 函数将回到“主线程”中执行。我们的 `baton` 结构也被传递进各函数，这些函数可以使用 `baton` 结构中的数据完成相关的操作。同时，我们也增加 `event loop` 的引用。这很重要，因为如果 `event loop` 无事可做，Node.js 就会退出，最终，函数返回 `Undefined`，因为真正的工作将在其他线程中完成。代码如下：

```

static int EIO_Hello(eio_req *req)
{
    hello_baton_t *baton = static_cast<hello_baton_t *>(req->data);

    sleep(baton->sleep_for);

    baton->hw->m_count += baton->increment_by;

    return 0;
}

```

这个回调函数将在 `libeio` 管理的线程中执行。首先，解析出 `baton` 结构，这样可以访问之前设置的各种参数。然后执行 `sleep(baton->sleep_for)`，这么做是安全的，因为这个函数运行在独立的线程中并不会阻塞主线程中 JavaScript 的执行。然后我们自增计数器，这些操作在实际的系统中，通常需要使用 `Lock/Mutex` 进行同步。

当上述方法返回后，`libeio` 将会通知主线程它需要在主线程上执行代码，此时 `EIO_AfterHello` 将会被调用，代码如下：

```

static int EIO_AfterHello(eio_req *req)
{
    HandleScope scope;
    hello_baton_t *baton = static_cast<hello_baton_t *>(req->data);
    ev_unref(EV_DEFAULT_UC);
    baton->hw->Unref();
}

```

进入此函数时，我们提取出 `baton` 结构，删除事件循环的引用，并减少 HelloWorld 对象的引用。

```

Local<Value> argv[1];

argv[0] = String::New("Hello World");

TryCatch try_catch;

baton->cb->Call(Context::GetCurrent()->Global(), 1, argv);

if (try_catch.HasCaught()) {
    FatalException(try_catch);
}

```

新建要传递给回调函数的字符串参数，并放入字符串数组中。然后我们调用回调传递一个参数，并检测可能抛出的异常。

```

baton->cb.Dispose();

    delete baton;
    return 0;
}

```

详细代码请查看本节最后的代码示例。

接下来我们创建一个测试代码，代码如下：

```

var Hello = require('./build/Release/helloworld_eio.node');
var hi = new Hello.HelloWorldEio();
hi.hello(function(data) {
    console.log(data);
});

```

同样应用 node-gyp 对代码进行编译，执行如下命令：

```

node-gyp configure
node-gyp build

```

编译完成后，我们通过运行测试代码后，可以看到如图 5-4 所示的返回结果，表明成功为 Node.js 添加 C++异步扩展了。

```

root@ubuntu:/home/danhuang/exten_asy# node test.js
WARNING: ev_ref is deprecated, use uv_ref
WARNING: ev_unref is deprecated, use uv_unref
Hello world

```

图 5-4 扩展模块应用示例运行结果

其中会有一个 WARNING: ev_ref is deprecated, use uv_ref 这样的 warning 提示。在使用 node-gyp 编译 Node.js 扩展时，需要注意其 target_name 必须和模块名一致，不然会出现类似下面的错误。

```

module.js:480
    process.dlopen(filename, module.exports);
              ^
Error: The specified procedure could not be found.

```

到这里，我们将 Node.js 的 C++扩展知识介绍完了，希望读者能够自己去实践这些功能。以下是本章中异步例子的代码，仅做参考。

```

/* This code is PUBLIC DOMAIN, and is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND. See the accompanying
 * LICENSE file.
 */

#include <v8.h>
#include <node.h>

#include <unistd.h>

using namespace node;
using namespace v8;

#define REQ_FUN_ARG(I, VAR)                                     \
    if (args.Length() <= (I) || !args[I]->IsFunction())      \
        return ThrowException(Exception::TypeError(          \
            String::New("Argument " #I " must be a function"))); \
    VAR = args[I].As<Function>();

```

```

Local<Function> VAR = Local<Function>::Cast(args[I]);
/**
 *
 * @desc 异步 helloworld 函数
 */
class HelloWorldEio: ObjectWrap
{
private:
    int m_count;
public:

    static Persistent<FunctionTemplate> s_ct;
    /* 初始函数 */
    static void Init(Handle<Object> target)
    {
        HandleScope scope;

        Local<FunctionTemplate> t = FunctionTemplate::New(New);

        s_ct = Persistent<FunctionTemplate>::New(t);
        s_ct->InstanceTemplate()->SetInternalFieldCount(1);
        s_ct->SetClassName(String::NewSymbol("HelloWorldEio"));

        NODE_SET_PROTOTYPE_METHOD(s_ct, "hello", Hello);

        target->Set(String::NewSymbol("HelloWorldEio"),
                    s_ct->GetFunction());
    }

    /* 构造函数 */
    HelloWorldEio() :
        m_count(0)
    {
    }

    /* 析构函数 */
    ~HelloWorldEio()
    {
    }

    static Handle<Value> New(const Arguments& args)
    {
        HandleScope scope;
        HelloWorldEio* hw = new HelloWorldEio();
        hw->Wrap(args.This());
        return args.This();
    }

    struct hello_baton_t {
        HelloWorldEio *hw;
        int increment_by;
        int sleep_for;
        Persistent<Function> cb;
    };

    static Handle<Value> Hello(const Arguments& args)
    {
        HandleScope scope;

        REQ_FUN_ARG(0, cb);

```



```

    HelloWorldEio* hw = ObjectWrap::Unwrap<HelloWorldEio>(args.This());

    hello_baton_t *baton = new hello_baton_t();
    baton->hw = hw;
    baton->increment_by = 2;
    baton->sleep_for = 1;
    baton->cb = Persistent<Function>::New(cb);

    hw->Ref();

    eio_custom(EIO_Hello, EIO_PRI_DEFAULT, EIO_AfterHello, baton);
    ev_ref(EV_DEFAULT_UC);

    return Undefined();
}

static int EIO_Hello(eio_req *req)
{
    hello_baton_t *baton = static_cast<hello_baton_t *>(req->data);

    sleep(baton->sleep_for);

    baton->hw->m_count += baton->increment_by;

    return 0;
}

static int EIO_AfterHello(eio_req *req)
{
    HandleScope scope;
    hello_baton_t *baton = static_cast<hello_baton_t *>(req->data);
    ev_unref(EV_DEFAULT_UC);
    baton->hw->Unref();

    Local<Value> argv[1];

    argv[0] = String::New("Hello World");

    TryCatch try_catch;

    baton->cb->Call(Context::GetCurrent()->Global(), 1, argv);

    if (try_catch.HasCaught()) {
        FatalException(try_catch);
    }

    baton->cb.Dispose();

    delete baton;
    return 0;
}
};

Persistent<FunctionTemplate> HelloWorldEio::s_ct;
extern "C" {
    static void init (Handle<Object> target)
    {
        HelloWorldEio::Init(target);
    }
}

```

```

    NODE_MODULE(helloworld_eio, init);
}

```

5.4 本章实践

1. 应用 node-gyp C++扩展编译工具，实现一个异步的模块，该模块中包含一个异步接口 `sendMsg` 方法，该方法接受一个字符串参数，然后为该字符串添加“ok”结尾，通过回调函数返回该执行结果，并打印显示。

分析：依据 5.4 节中的 `helloworld_eio.cc`，实现代码如下所示。

```

/* This code is PUBLIC DOMAIN, and is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND. See the accompanying
 * LICENSE file.
 */

#include <v8.h>
#include <node.h>

#include <unistd.h>

using namespace node;
using namespace v8;

#define REQ_FUN_ARG(I, VAR)                                     \
    if (args.Length() <= (I) || !args[I]->IsFunction())      \
        return ThrowException(Exception::TypeError(          \
            String::New("Argument " #I " must be a function"))); \
    Local<Function> VAR = Local<Function>::Cast(args[I]);

class SendMsgEio: ObjectWrap
{
private:
    int m_count;
public:

    static Persistent<FunctionTemplate> s_ct;
    static void Init(Handle<Object> target)
    {
        HandleScope scope;

        Local<FunctionTemplate> t = FunctionTemplate::New(New);

        s_ct = Persistent<FunctionTemplate>::New(t);
        s_ct->InstanceTemplate()->SetInternalFieldCount(1);
        s_ct->SetClassName(String::NewSymbol("SendMsgEio"));

        NODE_SET_PROTOTYPE_METHOD(s_ct, "sendMsg", SendMsg);

        target->Set(String::NewSymbol("SendMsgEio"),
            s_ct->GetFunction());
    }

    SendMsgEio() :
        m_count(0)
    {

```



```

}

~SendMsgEio()
{
}

static Handle<Value> New(const Arguments& args)
{
    HandleScope scope;
    SendMsgEio* hw = new SendMsgEio();
    hw->Wrap(args.This());
    return args.This();
}

struct hello_baton_t {
    SendMsgEio *hw;
    int increment_by;
    int sleep_for;
    Persistent<Function> cb;
};

static Handle<Value> SendMsg(const Arguments& args)
{
    HandleScope scope;

    REQ_FUN_ARG(0, cb);

    SendMsgEio* hw = ObjectWrap::Unwrap<SendMsgEio>(args.This());

    hello_baton_t *baton = new hello_baton_t();
    baton->hw = hw;
    baton->increment_by = 2;
    baton->sleep_for = 1;
    baton->cb = Persistent<Function>::New(cb);

    hw->Ref();

    eio_custom(EIO_Hello, EIO_PRI_DEFAULT, EIO_AfterHello, baton);
    ev_ref(EV_DEFAULT_UC);

    return Undefined();
}

static int EIO_Hello(eio_req *req)
{
    hello_baton_t *baton = static_cast<hello_baton_t *>(req->data);

    sleep(baton->sleep_for);

    baton->hw->m_count += baton->increment_by;

    return 0;
}

static int EIO_AfterHello(eio_req *req)
{
    HandleScope scope;
    hello_baton_t *baton = static_cast<hello_baton_t *>(req->data);
    ev_unref(EV_DEFAULT_UC);
    baton->hw->Unref();
}

```

```

    Local<Value> argv[1];

    argv[0] = String::New("ok");

    TryCatch try_catch;

    baton->cb->Call(Context::GetCurrent()->Global(), 1, argv);

    if (try_catch.HasCaught()) {
        FatalException(try_catch);
    }

    baton->cb.Dispose();

    delete baton;
    return 0;
}

};

Persistent<FunctionTemplate> SendMsgEio::s_ct;
extern "C" {
    static void init (Handle<Object> target)
    {
        SendMsgEio::Init(target);
    }

    NODE_MODULE(sendmsg_eio, init);
}

```

新增测试代码 test.js 如下:

```

var SendMsg = require('./build/Release/sendmsg_eio.node');
var hi = new SendMsg.SendMsgEio();
hi.sendMsg(function(data) {
    console.log(data);
});

```

运行测试代码 test.js, 结果如图 5-5 所示。

```

root@ubuntu:/home/danhuang/exten_asy# node test.js
WARNING: ev_ref is deprecated, use uv_ref
WARNING: ev_unref is deprecated, use uv_unref
ok

```

图 5-5 C++扩展测试实例执行结果

5.5 本章小结

本章介绍了一些关于 Node.js 的源码知识。重点介绍了 Node.js 中 C++扩展的代码实现方法, 如何应用 node-gyp 工具实现 C++扩展代码的编译, 以及如何应用通过 C++扩展编译后产生的 C++文件扩展模块。学完本章, 读者需要了解 Node.js 源码的相关知识, 掌握 Node.js 的 C++扩展开发。

V8 源码相关知识可以通过其官方网站学习，网址 <https://developers.google.com/v8/intro?hl=zh-CN>。

Node-gyp 工具应用参见 github 开源官网 <https://github.com/TooTallNate/node-gyp>。

本章节的相关代码实例，来自 pquerna 提供的 C++扩展源码网址 <https://github.com/pquerna/node-extension-examples>。

本章涉及 Node.js 的 API 列表如表 5.1 所示。

表 5.1 本章Node.js的API列表

模块名	API 名	作用	调用示例
HTTP	<code>http.createServer([requestListener])</code>	创建 HTTP 服务器	<code>http.createServer(function(req, res) { }).listen(1337, "127.0.0.1");</code>

第 6 章 Node.js 编码习惯

本章着重来介绍一些 Node.js 的编码习惯，当然这里所说的习惯都是一家之言，希望这些介绍能够对初学者有所帮助。大家在学习本章的过程中，随时可以提出疑问，通过邮件或者其他方式与笔者联系来探讨这些问题。写本章的目的，也是希望大家取长补短。本章内容包含 Node.js 中变量命名规范、函数命名规范、模块命名规范、异步逻辑编程规范和如何模块化编程。

读者肯定会觉得这部分很无味，因为这里讲解的不是技术，而是编程的思想，目的是规范化程序员的代码，让你更快地融入团队，更快地被他人认可。

在学习本章之前，希望读者可以去阅读《代码整洁之道》这本书。《代码整洁之道》介绍的规则均来自作者多年的实践经验，涵盖从命名到重构的多个编程方面，虽为一“家”之言，但有很多可借鉴的价值。

6.1 Node.js 规范

本节介绍 Node.js 中对变量、函数和模块进行制定命名规范，以及其中需要注意的问题，并通过实例对比来说明两者之间的优越性。编程规范是一门学问，就像师傅码砖头一样，码的好的会让房子看起来更齐整，码的不规范，房子看起来就歪歪斜斜没有章法。

鲁迅有说过“其实地上本没有路，走的人多了，也便成了路”，代码规范同样是这样，软件的世界里没有任何规范，当然一种方式走的人多了，被大多数人认可了，那这种方式就变成了一种规范。真正的代码规范是需要自我的约束和他人的共识，规范不仅仅是一个人的想法，它是编程者经过长期的实践所总结出的一套编码方式，这种方式更利于团队的合作和成长。

6.1.1 变量和函数命名规范

变量是什么？变量是指没有固定的值，其可以改变的数值。那么为什么要一个变量，而不是要一个具体的值呢？因为我们需要在代码执行期间，不断地改变它的值，因此这里可以总结出一点，对于那种从未改变的值，我们可以使用静态的数据来保存，而不是作为变量。

对于常量我们可以使用全大写字母来声明，由于 Node.js 中不存在静态变量，因此这里没有办法使用类似 PHP 中 `const` 来定义，静态变量可参考如下代码格式：

```
BASE_DIR = __dirname;  
APP      = BASE_DIR + "/app/";
```

```

CON    = APP + "/controller/";
CORE   = APP + "/core/";
LIB    = BASE_DIR + "/node_modules/";
CONF   = BASE_DIR + "/conf/";
STATIC = BASE_DIR + "/static/";
VIEW   = BASE_DIR + "/view/";

```

类似这些全局使用并且不更改的变量，依据笔者个人习惯是将其设置为 `global` 全大写字母来存储，目的是避免其他可执行文件中变量的命名冲突¹。

变量的用处在于能一般化描述指令的方式。这句话的意思是，变量一般会存储一个指令执行的结果，例如函数，因此这个变量代表着一定的意义。例如 JavaScript 内置属性 `length` 的返回结果，其代表是这个字符或者数组的长度，那么我们就应该用特定的字符去表示，如下代码的两种方式。

```

var name = 'danhuang';
/* 1 */
var a = name.length;
/* 2 */
var nameLength = name.length;

```

笔者在初学时不会想任何代码规范相关的东西，因此在当时直接使用第一种方式，为什么呢？原因很简单，不需要任何的思考。第二种方式从英文字符中我们就可以明白，该变量代表的是 `name` 的长度，因此在下一次使用到该变量时，我们就非常清楚该变量在代码中的含义，以及该变量产生的目的。两者在代码运行时并没有任何区别，只是程序员的编码习惯不同。

变量必定有一定的含义，在命名时可以使用相应的英文来表示（切忌使用汉语拼音），当由多个单词组成时，建议使用骆驼峰，当然也可以使用下划线相连，例如 `nameStr`、`nameArr` 和 `myName` 等。在《代码整洁之道》这本书中有提到，变量最好使用名词来描述。

在 Node.js 中变量声明原则上，笔者习惯使用骆驼峰来命名。其中变量可以统一定义的地方，最好统一定义，例如下面代码。

```

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] : '';

```

这种方式来统一定义需要的所有变量，既能够很好地管理，看起来也很整洁。其中也很清晰地表明了每个变量所代表的含义，`encodeModule` 是编码模式，`algorithm` 是算法类型，`enstring` 是需要编码字符，`returnType` 是返回类型等。

“要对自己命名的变量负责”，这句话的目的是希望每个变量都代表着其特定的意义，避免出现简单的字符或者数字来命名变量，如下代码：

```

var a = 'danhuang';    //字符串定义
var b = a.length;      //字符串长度定义
console.log(b);        //打印变量b
var c = a + b;         //执行a与b相加

```

1 所谓的变量命名冲突是指，同一个变量名在两处以上被声明，造成第一次声明时的变量失效。

```
console.log(c); //打印 c 变量
```

大家怎么看？通过代码大家当然可以知道 `c` 代表着什么意义。但是这样的代码显示会让人难以接受，如果这样编码的同学在某一天遇到 26 个字母全都用完时，他会怎样做？下面一段代码就是一个反面例子。

```
var b1 = 'a';
var b2 = 'b';
var b3 = 'c';
var b4 = 'd';
```

举这些例子并不是嘲笑这些程序员的编程有多糟糕，而是希望能够通过这些反面的例子提高读者的编程规范，提高编程能力。

对于对象命名最好使用首字母大写规范，例如代码：

```
var Util = new UtilClass();
```

根据上面介绍的内容，总结一下对于变量的命名规范。

- ❑ 变量是在运行期间会更改的数值，因此对于那些固定的变量，可以使用全局大写字母来定义。
- ❑ 变量代表着一定的意义，每个程序员都需要对变量负责，变量的产生就好比给孩子取名字一样，伴随着整个代码的执行过程，因此一定要描述其含义。
- ❑ 变量的命名建议使用骆驼峰规则，变量的名称描述使用英文，最好不要使用汉语拼音。
- ❑ 多个变量同时需要定义时，定义方式可使用首个变量使用关键字 `var`，其他变量使用逗号分割进行定义。
- ❑ 对于类和对象的变量建议使用首字母大写的骆驼峰规则。
- ❑ Node.js 中对于全局变量声明最好使用命名空间，避免变量冲突。例如以下代码，其中使用 `lib` 命名空间来存储变量，避免代码在其他文件中遇到变量冲突问题。

```
global.lib = {
  http      : require('http'), //Node.js 的 HTTP 模块
  fs        : require('fs'),   //Node.js 的 FileSystem 模块
  url       : require('url'),  //Node.js 的 url 模块
  querystring : require('querystring'), //Node.js 的 querystring 模块
  httpParam  : require(LIB + 'http_param'), //文件模块 http_param
  staticModule: require(LIB + 'static_module'), //文件模块 static_module
  router     : require(CORE + 'router'), //文件模块路由处理 router
  action     : require(CORE + 'action'), //文件模块 controller 基类
  jade       : require('jade'), //npm 模块 jade
  socket     : require('socket.io'), //npm 模块 socket.io
  path       : require('path'), //Node.js 的 path 模块
  parseCookie : require('connect').utils.parseCookie,
  session    : require(LIB + 'node_session'), //文件模块 session 管理
  util       : require('util') //Node.js 的 util 工具模块
}
```

以上几点变量命名规范希望读者在之后的 Node.js 编程或者其他编程中牢记于心，把握这些自我的编程能力会有一个提升，同时对工作中团队开发也会有很大的帮助。

接下来介绍函数的命名规范，函数和变量一样，其代表着一定的意义，只不过函数代

表的是一种动作，而变量代表的是一个名称。相对来说，函数的命名一般使用的是动词短语，而变量则是名词短语。

由于 Node.js 中方法就是对象，因此很多读者不能区分函数和对象，也就很难去命名函数。从笔者的个人编程经验来说，如果是使用 `require` 获取的模块对象，并且模块中使用 `module.exports` 返回函数的时候，一般认为这是一个类而不是一个方法，因此对于这种会使用首字母大写骆驼峰命名，而其他则会使用首字母小写的骆驼峰命名。示例代码如下：

```
/**
 *
 * @desc 暴露一个函数类
 */
module.exports = function(){
  this.show = function(){
    console.log('this is a class');
  }
}
```

上述代码中使用 `module.exports` 返回一个 `function`。根据上面的思想，大家会将其认为是一个 `Class`，因此在 `require` 返回该对象后，统一使用首字母大写赋值变量。

```
exports.show = function(){
  console.log('this is a object');
}
```

如果使用的是 `exports` 返回的，我们则看成一个 `object`，因此使用字母小写骆驼峰，调用赋值过程如下代码：

```
var Class = require('./class');
var object = require('./object');
console.log('*****is a class*****');
var classObj = new Class();
classObj.show();
console.log('*****is a object*****');
object.show();
```

【代码说明】

- ❑ `Class = require('./class')`: `class.js` 模块返回的是一个 `function`，因此这里看成类，使用首字母大写骆驼峰命名方法；
- ❑ `object = require('./object')`: `object` 模块返回的是一个 `object`，因此这里看成对象，使用首字母小写骆驼峰命名方法；
- ❑ `classObj = new Class()`: 作为一个类，必须先实例化对象后，才能调用其方法；
- ❑ `object.show()`: 对象可以直接调用其中的方法。

以上介绍可以总结为，对象使用首字母小写骆驼峰，类则使用首字母大写骆驼峰，判断是否命名为类和对象的一种方法就是，看它使用时是否需要实例化。

6.1.2 模块编写规范

Node.js 中涉及最多的就是模块，因此对模块规范化非常重要，很多时候我们都会困惑一个方法是使用 `exports` 返回，还是使用 `module.exports`。

理解上面的问题，首先要清楚两个 API 的作用，在前面几章中我都有提到两者的主要区别，一个是只能返回对象，而另外一个可以返回任何类型数据。在本书的 2.3 节中的“Node.js 中的设计模式”中介绍了如何在 Node.js 中应用类的设计思想，其中大部分使用的是 `module.exports` 来返回模块数据，这样才能获取函数类，而不是得到一个对象。对于对象和类的区别，读者还是不清楚的话可以学习《C++面向对象程序设计》这本书。

因此在写一个模块时，必须首先明白该模块返回的是对象还是函数类，我们可以将工具模块作为一个对象返回，而父类或者基类则使用函数类返回。

什么是工具类？例如静态文件模块、HTTP 参数获取模块和加密解密模块等。

在涉及应用的是 Node.js 的设计模式应用时，例如在之前的直播应用中的 `core` 文件夹下的一些模块，一般在设计该类函数名称时，应用的是函数模块名称，也就是将其模块名作为函数名。例如 `base.js` 文件代表的是 `Base` 函数类，但该函数是一个匿名函数。需要注意的是 `base.js` 中并不能指明 `base.js` 中返回的是一个 `Base` 类，因为 `module.exports` 返回的是一个 `function` 函数类，该函数是一个无名函数，因此可以将其设置为任意名称，但是建议大家规范为模块文件名，如 `base.js` 对应 `Base` 类名。其他非函数类的模块返回时按照对象命名规则进行赋值即可。

```
/* base.js */
module.exports = function(){};
/* util */
exports.util = function(){};
/* index.js */
var Base = require('./base');
var utile = require('./util');
```

这段代码和上一节的代码有所相同，其目的都是想凸显出函数类和模块返回对象的区别。希望通过本节介绍，能让读者进一步了解 `module.exports` 和 `exports` 的区别，同时规范化函数类和对象的赋值与命名。

6.1.3 注释

注释是否应该存在？《代码整洁之道》中有一句很经典的话“代码本身就是注释”。因此函数的命名和规范很重要，当然“代码本身就是注释”这样的编程者已经达到了一定的高度，但是还是建议读者可以做适当的注释说明，在一些关键逻辑或者关键算法的时候提倡大家使用注释，既能够让他人看懂你的代码，在长时间之后也能够让自己理解负责算法的逻辑。如何灵活的应用注释可以看《代码整洁之道》或者英文版本《clean code》。

Node.js 中的注释要注意以下几点：

- ☐ 模块最前面提供注释说明模块返回的属于类还是对象。
- ☐ 模块中如果返回的是对象，简单描述对象中的外部调用方法。
- ☐ 模块如果返回的是类，则说明该类的名称。
- ☐ 模块中的函数添加必要注释。

例如如下代码：

```
/**
 * @type class Note
```

```

* @author danhuang
* @time 2012-12-17
* @desc desc note.js
*
*/
module.exports = function() {
/**
*
* @desc show a short message
* @param null
* @return null
*/
  this.show = function() {
    console.log('this is a note test');
  }
}

```

【代码说明】

- ❑ `@type class Note`: 模块类型，返回的是 `object` 还是 `class`。
- ❑ `@author danhuang`: 编写该模块的作者，便于团队开发学习。
- ❑ `@time 2012-12-17`: 编码时间。
- ❑ `@desc desc note.js`: 模块描述，用来说明该模块的作用。
- ❑ `@param null`: 函数需要的参数。
- ❑ `@return null`: 函数的返回数据类型。

这些注释关键字主要有 `type`、`author`、`time`、`desc`、`param` 和 `return`，目的是希望大家在团队开发过程中能够统一规范进行编码。自己编码时更需要注意这些，尤其是初学者，更要在起步的时候规范自己的编码习惯。

6.2 Node.js 异步编程规范

在本书的 1.3 节中简单地介绍了同步、异步和回调函数的区别和联系，对于 Node.js 的异步编程思想和规范并没有涉及很深。本节会从异步代码执行底层原理、异步代码调用执行过程、异步编程中的回调深度和异步代码编程遇到的问题这几个方面进行介绍。通过本节的介绍，读者能够了解异步代码的底层实现原理，掌握异步函数的调用，以及结果的获取方法，了解如何解决 Node.js 的回调深度问题。

6.2.1 Node.js 的异步实现

本节参考博客园 orlla 的一篇文章《初探 Node.js 的异步 I/O 实现》¹。我们经常说的 Node.js 是异步非阻塞，并不是说所有的 Node.js 代码执行都是非阻塞，例如如下代码：

```

var date = new Date();
var millSec = date.getSeconds();
var minSec = 1;

```

1 参见网站 <http://www.cnblogs.com/orlla/archive/2012/07/12/2588066.html>。

```

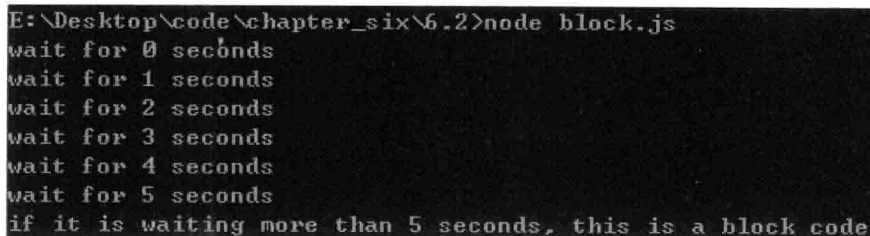
/* while 循环等待 5 秒 */
while(minSec<5){
    /* 获取当前时间 */
    var nowDate = new Date();
    var nowMillSec = nowDate.getSeconds();
    var tempSec = nowMillSec - millSec;
    if(tempSec != minSec){
        minSec = tempSec;
        console.log('wait for ' + minSec + ' seconds');
    }
}
console.log('if it is waiting more than 5 seconds, this is a block code');

```

【代码说明】

- ❑ `millSec = date.getSeconds()`: 获取代码最初执行时间。
- ❑ `while(minSec<5){}`: 等待代码 5 秒执行时间。
- ❑ `if(tempSec != minSec){}`: 避免输出相同的字符串, 保存一秒钟输出一字符串。
- ❑ `tempSec = nowMillSec - millSec`: 获取代码当前的等待时间。

这是很多程序员提到的问题, 大家都认为 Node.js 是异步非阻塞, 因此想到了一个问题, 按照大家的想法应该会先输出 `if it is waiting more than 5 seconds, this is a block code` 这段文字。现在我们运行一下该段代码, 执行结果如图 6-1 所示。



```

E:\Desktop\code\chapter_six\6.2>node block.js
wait for 0 seconds
wait for 1 seconds
wait for 2 seconds
wait for 3 seconds
wait for 4 seconds
wait for 5 seconds
if it is waiting more than 5 seconds, this is a block code

```

图 6-1 同步测试代码执行结果

从图 6-1 中可以看到这段代码是一个阻塞的, 结果中显示系统从 0 秒开始等待, 一直到 5 秒结束后才执行最后一个字符串输出。而如果是非阻塞的情况时, 应该会先输出 `if it is waiting more than 5 seconds, this is a block code` 字符串, 再输出等待 5 秒的过程。

因此并非 Node.js 所有代码执行都是异步的。这里所说的异步是指 Node.js 中提供的 API, 在 API 中我们也看到了很多同步调用的函数, 例如 `filesystem` 模块中的 `fs.realpathSync(path, [cache])`, 就是一个同步阻塞 API, 在文件模块中 Node.js API 中都提供了异步和同步调用的两种方法。

既然知道了异步代码执行依赖的是 Node.js 提供的异步调用 API, 那么我们就将上述等待 5 秒的阻塞性执行过程修改为非阻塞异步调用过程。这里需要使用到 Node.js 中提高的异步等待方法 `setTimeout`, 代码如下:

```

setTimeout(function(){
    console.log('wait for 5 seconds over');
}, 5000)
console.log('if it is waiting more than 5 seconds, this is a block code');

```

【代码说明】

- ❑ `setTimeout`: 两个参数分别为回调函数和等待时间 (单位为毫秒)。

运行上面代码后，我们可以看到如图 6-2 所示的返回信息。

```
E:\Desktop\code\chapter_six\6.2>node asy.js
if it is waiting more than 5 seconds, this is a block code
wait for 5 seconds over
```

图 6-2 异步 Node.js 代码执行结果

从结果可以很清晰地看到，先输出的是最后一行代码逻辑，而 `setTimeout` 回调函数中的输出是在 5 秒等待结束后才输出，这样执行的过程才是一个异步非阻塞代码逻辑。第二个 `console` 逻辑不依赖前面代码执行结果，而异步代码在执行完成后，通过事件驱动来调用回调函数，来获取异步逻辑的执行结果。

上面这个对比是一个很常见的例子，用来说明 Node.js 的代码执行中的异步非阻塞原理。Node.js 中的异步 API 是需要操作系统交互和支撑的，如果没有应用到 Node.js API 提供的异步调用过程，其代码逻辑运行还是同步阻塞的。根据以上介绍的那些知识点，总结一点非常重要的知识是，并非 Node.js 中的所有代码逻辑都是异步非阻塞，只有在调用了 Node.js 提供的异步 API 时才是一个异步非阻塞逻辑。

上面主要区分了 Node.js 中的异步和同步代码逻辑。接下来会简单介绍 Node.js 中如何应用操作系统来实现 API 的异步非阻塞，其中涉及很深入的内容大家可以不用深究，只需要了解该过程即可。

在操作系统中，程序运行的空间分为内核空间和用户空间。我们常常提起的 Node.js 的异步 I/O，其实质是用户空间中的程序不用依赖内核空间中的 I/O 操作完成，可以运行之后的操作任务。Node.js 在文件 I/O 这方面与普通的业务逻辑的回调函数的不同在于，它不是由我们自己的代码所触发的，而是系统调用结束后，由系统触发的。

异步 I/O 是应用程序发起异步调用，进而处理下一个任务，当 I/O 执行完成时通过信号或是回调将数据传递给应用程序即可。如图 6-3 所示是一个简单的异步过程。

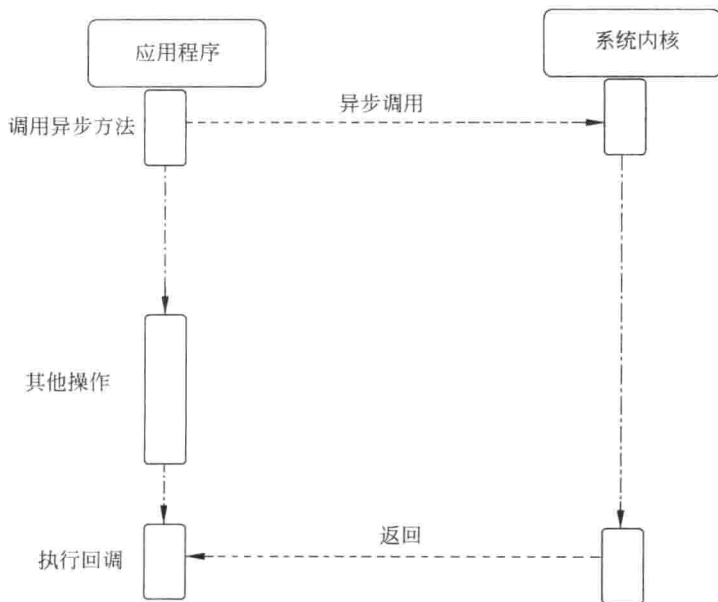


图 6-3 异步逻辑系统运行结构图

之前介绍的一个例子用两种方法实现时间等待的过程，虽然都是等待 5 秒，但使用 `while` 的代码实现等待的过程，仅仅是停留在应用程序的层面上，而简单停留在应用程序运行层面是无法实现异步代码过程的。而 `setTimeout` 的例子与系统内核交互实现异步过程，而无需等待 I/O，直接执行其他操作，当执行完成后返回信息，执行回调函数。

6.2.2 异步函数的调用

对于异步函数来说存在一个问题，就是如何去获取返回的结果，对于熟悉同步代码编写的读者来说，可能会对这种编码习惯非常不适应。最近在 CNode 社区看到很多用户讨论了一个问题，问题的代码如下：

```
function select(a_where,a_data)
{
var t_ret=0;
t_ret= db_connector.collection(a_where, function(err, collection) {
    collection.find(a_data, function(err, value){
        value.toArray(function(err,arr){
            return "true";
        })
    })
});
return "false";
}
```

CNode 用户的问题描述说，现在调用 `select` 函数时，虽然里面的代码执行了，可是一直返回 `false`，用户明白它为什么会返回 `false`，但是不知道该如何修改才能够让该函数返回 `true`。

上面代码中 `db_connector.collection` 是一个异步执行函数，因此 `db_connector.collection` 返回的任何值都不会影响 `select` 最后的返回。程序在执行 `db_connector.collection` 时会和内核交互，而这个时候不会等待异步函数的运行，程序会继续运行到 `return "false"`，因此无论如何最后返回的都是 `"false"`。而读者试图在回调函数中 `return` 一个值，这样做是无意义的，因为在回调函数中 `return` 出去的也只是 `db_connector.collection` 回调函数的返回结果，并非 `select` 函数的返回结果。同样，将 `db_connector.collection` 执行结果赋值给 `t_ret` 也是无意义的，因为该异步函数并没有任何的返回值，因此该处获取的 `t_ret` 也只是一个 `null`，这样不仅无意义，而且还消耗系统资源。

那么我们应该如何正确地调用异步函数呢？

首先，对于一个异步函数，我们必须传递一个回调函数来执行回调后的程序，例如上面代码中 `function(err, collection)` 函数，其作用就是在异步代码执行结束后执行的回调函数程序。代码如下：

```
function(err, collection) {
    collection.find(a_data, function(err, value){
        value.toArray(function(err,arr){
            return "true";
        })
    })
}
```

根据之前该用户遇到的问题，他的目的是调用 `select` 函数来获取 `value.toArray` 的执行

结果 `arr` 的值。因为 `select` 函数中调用了一个异步函数，因此 `select` 也是一个异步执行函数，获取异步调用的返回结果的方法就是添加一个回调函数变量，如 `select(a_where,a_data, callback)`，因此上述代码重构为如下：

```
function select(a_where,a_data,callback)
{
    var t_ret=0;
    t_ret= db_connector.collection(a_where, function(err, collection) {
        collection.find(a_data, function(err, value){
            value.toArray(function(err,arr){
                callback(arr);
            })
        })
    });
    return "false";
}
```

【代码说明】

- ❑ `a_where,a_data,callback`：添加一个回调函数来处理异步执行返回结果。
- ❑ `callback(arr)`：异步逻辑执行结束后，调用 `select` 传递的回调函数，并将需要返回的数据作为 `callback` 的参数传递。

通过 `callback` 回调函数就可以将返回结果传递到其他函数中进行处理，例如添加如下代码：

```
function getSelectArr(arr){
    console.log(arr);
}
select(a_where,a_data,getSelectArr);
```

函数 `getSelectArr` 就可以处理 `select` 返回的结果。调用异步函数的主要方法就是添加一个回调函数，然后把需要获取的返回结果作为回调函数的参数。下面介绍异步函数调用公式。

假设 `A` 为异步函数，`A` 有参数 `a` 和回调函数 `function(b)`，其中回调函数带有一个返回值 `b`，`A` 函数的调用方式则为 `A(a, function(b))`，`C` 为一个封装函数，其中调用了异步函数 `A`。那么希望在外调用 `C` 函数时，获取 `A` 函数异步执行的结果 `b` 值，则可以为 `C` 添加一个回调函数名为 `c`，调用方法为 `C(a, c)`。上面公式用伪代码可以表示如下：

```
function C(a, c){
    A(a, function(b){
        c(b);
    })
}
function c(b){
    console.log(b)
}
```

【代码说明】

- ❑ `function C(a, c)`：`c` 为 `C` 函数的一个回调函数名。
- ❑ `A(a, function(b){})`：`A` 为是一个异步函数，其中 `a` 为其参数，`function(b){}` 为无名回调函数。
- ❑ `c(b)`：调用 `C` 的回调函数名，将 `b` 返回值传递给 `c` 函数。

❑ **function c(b):** 处理异步调用函数的返回结果。

上面公式代码使用了很多简单的字母，主要是希望能够将公式这种数学利器应用到我们的程序教学中。

6.2.3 Node.js 异步回调深度

回调深度是指在调用异步函数时，需要依赖多层异步函数来处理返回结果。公式法 A、B、C 和 D 假设都为异步函数，并且 A 依赖 B，B 依赖 C，C 依赖 D，则产生的回调深度公式过程如下代码所示。

⚠注意：异步回调深度并没有明确地说明异步调用层数是多少时，才会出现异步回调深度这个问题，只是说明异步编程会给程序员带来编程困难。

```
/**
 *
 * @desc 异步回调深度公式说明法
 * @parameters a 该参数为 A 异步函数依赖参数
 * @parameters callback 为回调函数
 */
A(a, function(b){
  B(b, function(c){
    C(c, function(d){
      D(d, function(e){
        //do some thing
      })
    })
  })
})
})
```

【代码说明】

- ❑ a 为函数 A 调用需要的参数，b 为 A 异步函数执行返回的结果。
- ❑ b 为函数 A 执行的结果，同时为异步 B 函数的参数。
- ❑ c 为函数 B 执行的结果，同时为异步 C 函数的参数。
- ❑ d 为函数 C 执行的结果，同时为异步 D 函数的参数。

D 中是我们需要处理返回的逻辑，而 A、B、C 异步函数的目的都是为了将 a 参数转化为 d 参数的结果。根据上一章节介绍的 Node.js 如何处理异步调用返回结果公式方法，现在我们需要将上面公式代码封装成一个异步函数，应该如何做呢？

根据上一节中的公式方法，为函数添加一个异步函数，例如封装后的函数如下代码：

```
/**
 *
 * @desc 异步回调深度公式说明法
 * @parameters a 该参数为 A 异步函数依赖参数
 * @parameters callback 为回调函数
 */
function deaWithResult(a, callback){
  A(a, function(b){
    B(b, function(c){
      C(c, function(d){
        D(d, function(e){
```



```

        callback(e);
    })
  })
}

```

【代码说明】

❑ `function deaWithResult(a, callback)`: 为异步函数添加一个回调函数参数名。

❑ `callback(e)`: 异步函数执行完后调用回调函数，处理返回结果。

根据定义中的公式来看一下实例代码，代码如下：

```

var fs = require('fs');
fs.exists('index.txt', function(stats){
  if(stats){
    fs.rename('index.txt', 'test.txt', function(err){
      fs.unlink('test.txt', function(err){
        if(err) console.log(err);
        else
          console.log('delete success');
      });
    });
  }
});

```

本部分代码中应用到了 Node.js 异步函数中的 `exists`、`rename`、`unlink` 这 3 个异步函数，而且每一个异步函数都依赖上一个异步函数的返回结果。这样就导致我们的代码写了 3 层嵌套，该例子中的情况就产生了所谓的 Node.js 中的异步回调深度问题。

这里我们并没有严格的定义异步调用深度，只是简单说明在编码过程中会遇到此类问题。在编写异步代码时，问题的查找往往非常困难，同时也不利于 Node.js 的模块化编程，会造成一个函数执行多层逻辑直到得到正确的返回值才结束。

6.2.4 解决异步编程带来的麻烦

封装异步函数是一个解决异步回调深度的问题的办法。当然还有其它方法，笔者最常使用的方法是在异步函数调用处进行封装添加一个匿名回调函数参数，就如上一节提到的公式方法。例如，上面代码可以应用这种方法进行优化，代码如下：

```

var fs = require('fs');
/* delete file */
deleteFile('index_callback.txt');

```

`deleteFile` 为调用函数，其处理逻辑包括检验文件存在与否、重命名文件和删除文件，代码如下：

```

/**
 *
 * @desc check file and delete file
 * @param file string
 * @return null
 */
function deleteFile(file){
  fs.exists(file, function(stats){

```

```

    if(stats){
        rename(file, 'test.txt');
    }
  });
}

```

删除文件前需要检验文件是否存在, 如果存在则会调用异步函数 **rename** 来重命名文件为 **test.txt**, 代码如下:

```

/**
 *
 * @desc rename file
 * @param file string
 * @param newName string
 * @return null
 */
function rename(file, newName){
  fs.rename(file, newName, function(err){
    unlink(newName, showResult);
  });
}

```

rename 异步函数重命名文件, 并调用异步函数 **unlink** 来删除文件, 代码如下:

```

/**
 *
 * @desc delete file
 * @param file string
 * @param callback function
 * @return null
 */
function unlink(file, callback){
  fs.unlink(file, function(err){
    if(err) console.log(err);
    else
      callback('delete success');
  })
}

```

unlink 删除文件, 删除成功后执行 **callback** 函数 **showResult** 来显示删除是否成功的信息, 代码如下:

```

/**
 *
 * @desc show message
 * @param msg string
 * @return null
 */
function showResult(msg){
  console.log(msg);
}

```

以上过程就是将一个异步回调深度的代码, 转化为一个多个函数之间调用的过程。方法主要是将每个或者每两个异步函数进行函数封装, 每次在调用依赖异步函数时添加一个回调参数, 来执行依赖函数。这种方法依然存在异步回调深度, 异步编程对于开发者来说十分麻烦, 它会将程序逻辑拆得分支离破碎, 语义完全丢失。

1. 老赵的 Wind.js

接下来介绍一下老赵的 wind.js, 在介绍他的 Wind.js 前, 我们先来了解一下老赵。以下是摘自他被采访时的自我介绍内容:

赵劫, 在网上一般被称为老赵或者赵姐夫, 经常沉迷于代码世界里, 关注技术发展或是程序员的成长等话题, 而不太愿意接触如今比较“流行”的产品设计、项目管理、产业分析等, 连所谓的“大规模应用架构设计”都不太关心, 甚至还略有抵触, 所以可以说他是个“纯码农”。之前呆过外企、创过业, 也呆过国内的互联网企业, 现在则是在做外资银行的内部系统, 可以说是一块没有接触过的领域, 痛并快乐着¹。

Wind.js 的前身为 Jscecx, 即 JavaScript Computation EXpressions 的缩写, 它为 JavaScript 语言提供了一个 monadic 扩展, 能够显著提高一些常见场景下的编程体验(例如异步编程)。Wind.js 完全使用 JavaScript 编写, 能够在任意支持 JavaScript 的执行引擎里使用, 包括各浏览器及服务器端 JavaScript 环境²。

Node.js 的 Wind 安装方法是执行 `npm install wind`。先看一个简单的使用实例, 代码如下:

```
function loadData(cb) {
  setTimeout(function() { cb(null, "blabla"); }, 5000);
}
function loadError(cb) {
  setTimeout(function() { cb("Some error"); }, 5000);
}

var loadDataAsync = Wind.Async.Binding.fromStandard(loadData);
var loadErrorAsync = Wind.Async.Binding.fromStandard(loadError);

var someTestFunction = eval(Wind.compile('async', function() {
  var data = $await(loadDataAsync());
  console.log('Some Result: ' + data);
  try {
    $await(loadErrorAsync());
  } catch(e) {
    console.log('Some Error Occurred: ' + e);
  }
}));

someTestFunction().start();
```

代码摘自 <http://www.greatony.com/?p=195>。

【代码说明】

- ❑ `loadData(cb)`: 异步时间等待函数。
- ❑ `loadError(cb)`: 异步时间等待函数。

2. EventProxy

EventProxy³ 仅仅是一个很轻量的工具, 但是能够带来一种事件式编程的思维变化,

1 参见网络 <http://www.infoq.com/cn/articles/interview-jscecx-author-part-1>。
 2 摘自 wind.js 官网介绍, 参见网络 <http://windjs.org/cn/docs/>。
 3 参见网络 <https://github.com/JacksonTian/eventproxy>。

它有以下几个特点：

- ❑ 利用事件机制解耦复杂业务逻辑。
- ❑ 移除被广为诟病的深度 `callback` 嵌套问题。
- ❑ 将串行等待变成并行等待，提升多异步协作场景下的执行效率。
- ❑ 友好的 `Error handling`。
- ❑ 无平台依赖，适合前后端，能用于浏览器和 `Node.js`。
- ❑ 兼容 `CMD`、`AMD`，以及 `CommonJS` 模块环境。

应用 `EventProxy` 实现的无深度嵌套的 `Node.js` 代码，并能够使用并行代码来实现编程，代码如下：

```
var ep = EventProxy.create("template", "data", "l10n", function (template, data, l10n) {
  _ .template(template, data, l10n);
});

$.get("template", function (template) {
  //something
  ep.emit("template", template);
});
$.get("data", function (data) {
  //something
  ep.emit("data", data);
});
$.get("l10n", function (l10n) {
  //something
  ep.emit("l10n", l10n);
});
```

解决原有的 `Node.js` 异步回调深度的问题，代码如下：

```
var render = function (template, data) {
  _ .template(template, data);
};
$.get("template", function (template) {
  //something
  $.get("data", function (data) {
    //something
    $.get("l10n", function (l10n) {
      //something
      render(template, data, l10n);
    });
  });
});
```

3. Step¹

使用 `step` 来实现同步执行的过程代码，代码如下：

```
step(
  function thefunc1(){
    func1(this);
  },
  function thefunc2(finishFlag){
    console.log(finishFlag);
  }
```

¹ 参见网络 <https://github.com/creationix/step>。

```
func2(this);
},
function thefunc3(finishFlag) {
  console.log(finishFlag);
}
);
```

其异步编程代码如下：

```
/* 异步回调深度示例 */
func1(req, res, function() {
  func2(req, res, function() {
    func3(req, res, function() {
      process.exit(0);
    })
  });
});
```

其他的解决办法还包含 `async` 的 `series` 方法，这些都可以在 [github](#) 上找到开源的代码实例。

6.3 异常逻辑的处理

在其他编程语言中，对异常逻辑处理的要求，可能并不是非常高，遇到几个异常逻辑的时候会出现一些问题，但不会导致整个项目或者服务无法使用的情况。对于 Node.js 的项目来说，则有所不同。由于 Node.js 是单线程的，其容错相对来说不是很好，因此如果遇到异常逻辑的话，就会导致进程中断，从而导致整个服务器无法使用。本节将举例说明进程中断导致的问题，以及我们应该如何去处理这类异常情况。

6.3.1 require 模块对象不存在异常

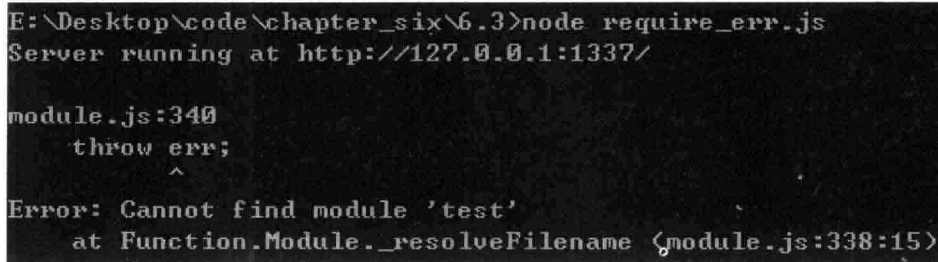
在 Node.js 中如果出现异常将会导致项目无法正常使用。很多时候，在项目启动期间是不会出现严重的异常逻辑，当执行到相应逻辑时才出现问题，从而导致 HTTP 服务器挂死的情况。下面举几个很简单的例子，说明在 Node.js 中的异常逻辑，当然不是所有的异常情况都会导致进程挂死的情况。

`require` 一个不存在的文件模块时，如果我们不能保证该模块是否存在时，必须要做一定的异常逻辑处理，否则会导致项目的进程中断。如下代码：

```
var http = require('http');
/* 创建 HTTP 服务器 */
http.createServer(function(req, res) {
  var test = require('test');
  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/');
```

上面代码中 `test` 模块是不存在的，但是在启动运行该文件时，不会出现任何异常情况，

但是当浏览器访问 `http://127.0.0.1:1337/` 时会导致异常，从而中断 Node.js 进程。访问后可以看到运行窗口上出现如图 6-4 所示的异常信息。



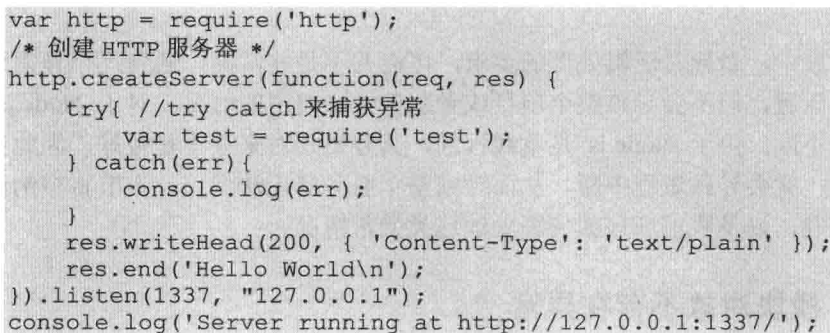
```
E:\Desktop\code\chapter_six\6.3>node require_err.js
Server running at http://127.0.0.1:1337/

module.js:340
  throw err;
  ^
Error: Cannot find module 'test'
    at Function.Module._resolveFilename (module.js:338:15)
```

图 6-4 require 异常返回错误信息

出现上面异常后，我们再想访问该 url 地址时，则无任何响应信息，因为该进程已经中断。那么对于这种情况，当不确定一个模块是否正常或者存在与否，我们应该如何去处理这种异常逻辑呢？

可以应用 Node.js 的 `try catch` 逻辑来处理。例如将上面代码修改为如下方式，就可以很好地保证代码的健壮性：



```
var http = require('http');
/* 创建 HTTP 服务器 */
http.createServer(function(req, res) {
  try{ //try catch 来捕获异常
    var test = require('test');
  } catch(err){
    console.log(err);
  }
  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/');
```

使用 `try catch` 来捕获 `require` 一个模块的异常情况，同样运行上面代码，然后打开浏览器输入 `http://127.0.0.1:1337/`，可以看到页面可以正常地响应数据，如图 6-5 所示。

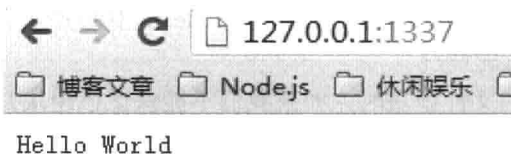
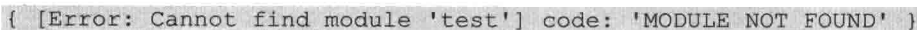


图 6-5 浏览 `http://127.0.0.1:1337/` 响应 Web 页面

然后查看运行窗口，可以看到其异常信息，提示如下字符，但不会导致进程直接挂掉的情况：



```
{ [Error: Cannot find module 'test'] code: 'MODULE_NOT_FOUND' }
```

灵活地应用 `try catch` 是一种非常好的避免 Node.js 中异常的处理办法。

6.3.2 对象中不存在方法或者属性时的异常

当有一个对象我们不知道其是否存在某个属性或者方法时，贸然的使用该对象的方法，同样会导致 Node.js 的异常情况，例如下面的异常代码。

首先创建一个 test_module.js 代码，代码如下：

```
exports.name = 'danhuang';
exports.say = function(){
    console.log('test');
}
//将该模块中的 show 方法进行隐藏
/**
exports.show = function(){
    console.log('hi');
}
*/
```

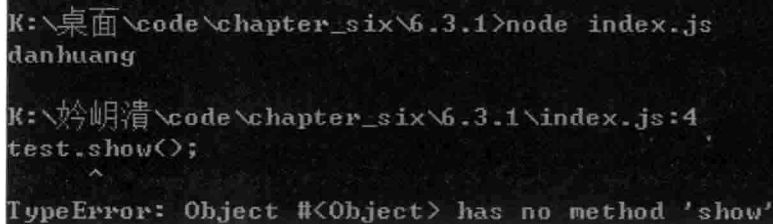
如上代码暴露了 name 属性和 say 方法，但将 show 方法给注释了。接下来我们在 index.js 中尝试调用 name 属性、say 和 show 方法，代码如下：

```
var test = require('./test_module');
console.log(test.name);

/* 无任何判断，直接调用 test 对象中的 show 方法 */
test.show();

test.say();
```

结果大家可以看到如图 6-6 所示的返回。



```
K:\桌面\code\chapter_six\6.3.1>node index.js
danhuang

K:\桌面\code\chapter_six\6.3.1\index.js:4
test.show();
    ^
TypeError: Object #<Object> has no method 'show'
```

图 6-6 调用不存在接口异常错误

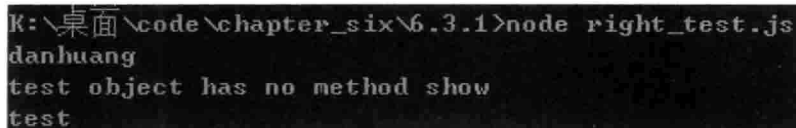
从图 6-6 中可以看到其正常地执行了打印 name 属性，但是遇到了一个没有定义的 show 方法时，Node.js 就抛出了相应的异常消息，并强制退出了，导致我们正常的 say 方法却无法调用。对于这种属性和方法不存在的情况，只需要做一定的判断就可以了，例如以下代码：

```
var test = require('./test_module');
console.log(test.name);

/* 判断 test 方法中是否存在 show 方法，不存在就提示错误信息，存在的话 show 方法才执行 */
test.show ? test.show() : console.log('test object has no method show');

test.say();
```


这里使用了一个三元判断符来避免异常情况中断 Node.js 进程，首先判断该对象中是否有该方法，如果存在的话就调用，否则，提示相应的错误信息，而不是直接中断程序，如图 6-7 所示为执行结果。



```
K:\桌面\code\chapter_six\6.3.1>node right_test.js
danhuang
test object has no method show
test
```

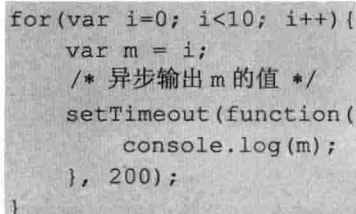
图 6-7 添加异常处理执行结果

从上图执行过程中可以看到正常地执行了 test.say 逻辑，并非因为 show 方法不存在的情况导致不能执行 say 逻辑。在适当的地方使用 if 或者三元条件来判断对象中的属性和方法的存在与否，也是解决异常的一种常用的办法。

如果确定该对象中存在该方法，可以无需判断。但是调用的方法是一个变量时，则需要做相应的判断，否则会出现异常情况，特别是在路由处理的逻辑部分。

6.3.3 异步执行的 for 循环异常

在介绍该方法前，首先来看一段异常，代码如下：



```
for(var i=0; i<10; i++){
  var m = i;
  /* 异步输出 m 的值 */
  setTimeout(function(){
    console.log(m);
  }, 200);
}
```

如上代码的输出结果是什么呢？运行如上代码后，可以看到如图 6-8 所示的输出信息。



```
L:\6>node for_err.js
9
9
9
9
9
9
9
9
9
9
```

图 6-8 for 循环异常执行结果

这些返回值时与我们想象的是不是有很大的区别？原本我们的目的是希望每隔 200 毫秒打印一个 m 的值（0~9 之间），但最后全部打印显示的是 9，而没有输出 0~8，这是为什么呢？

setTimeout 是一个异步函数，for 循环时不会等待 setTimeout 异步函数，因此在第一个异步 setTimeout 还未执行时，for 循环已经结束，此时的 m 已经从 0 循环变为 9，所以十次执行的 m 值都为 9。可以使用图 6-9 来表示该过程。

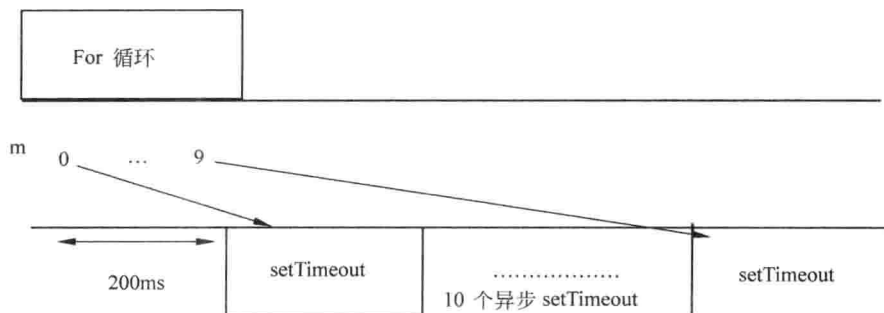


图 6-9 for 循环执行逻辑与 setTimeout 执行时间线对比图

for 循环的时间不超过 200ms，而每次 for 循环时都会执行一个异步的 setTimeout 函数，在第一个 setTimeout 还未执行时，for 循环已经在 200ms 内结束了执行，并将 m 变为了 9，导致所有的 setTimeout 执行时打印的都是 m 为 9 的结果。

既然存在这种异步循环，循环中的变量该如何在异步函数中使用呢？如下为一种解决办法，代码如下：

```
for(var i=0; i<10; i++){ //for 循环打印 m 值
    var m = i;
    exec(m, function(m){
        console.log(m);
    });
}
/**
 * @desc 定义一个回调函数
 */
function exec(param, callback){
    setTimeout(function(){
        callback(param);
    }, 200);
}
```

执行如上代码，可以看到如图 6-10 所示的返回结果。

```
l:\6>node for_err.js
0
1
2
3
4
5
6
7
8
9
```

图 6-10 正常 for 循环执行结果

从图中可以看到，这次正常地按照我们的设计从 0~9 进行打印。上面代码和之前的异步逻辑代码的区别主要是，这部分代码是将 `m` 的值传递进入一个异步函数中，作为异步函数的参数，而不是直接打印 `for` 循环中的 `m` 值，这样在每次 `for` 循环时，传入到异步函数中的参数 `m` 值都是不同的，因此可以打印相应的信息。

这部分代码主要告诉读者，当 `for` 循环中存在异步函数，并且异步函数依赖 `for` 循环中的某些参数时，我们必须将依赖值作为参数传递到异步函数中，而非直接在异步函数中使用 `for` 循环中的变量。如果直接使用 `for` 循环中的变量，会导致异步函数中的参数永远都是 `for` 循环的最后一个值。

例如，`a` 是一个随着 `for` 循环变化的参数，而在异步函数 `B` 中需要使用 `a` 参数，那么这时我们必须将 `a` 参数作为异步函数 `B` 的一个参数传递到异步函数 `B` 中，示例公式如下：

```
var A = Array;
/* 异步接口依赖 for 循环变量时，处理方法公式 */
for(var i=0; i<A.length; i++){
  a = A[i];
  B(a, function(a){
    console.log(a);
  })
}
```

如果我们需要传递多个依赖 `for` 循环的参数时，可以将 `a` 作为一个 `Json` 对象传递到异步函数 `B` 中，这样就可以有效地解决异步 `for` 循环问题。

6.3.4 利用异常处理办法优化路由

既然上面介绍了两种异常的问题，现在我们利用上面两种解决办法来优化我们的路由处理模块。现在有一个这样的模块，其请求路径规则方式如图 6-11 所示。

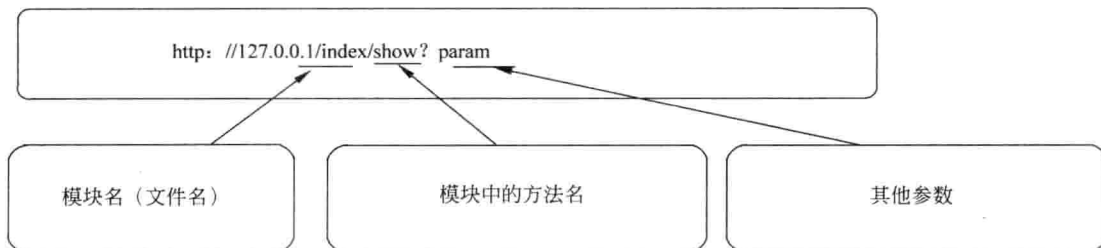


图 6-11 url 路径分析

其中的 `index` 为模块名，`show` 为模块中的方法，而 `param` 则为请求 `get` 参数，接下来我们需要写一个对应的解析方法，代码如下：

```
var url = require('url');
exports.router = function(res, req){
  var pathname = decodeURI(url.parse(req.url).pathname);
  var pathArr = pathname.split('/')
  , model = pathArr[1]
  , controller = pathArr[2]
  , Class = '';
  if(model == 'favicon.ico'){ //去除浏览器自带 favicon.ico 请求
```

```

        return;
    }
    /* require model 方法, 获取 model 类 */
    Class = require('./controller/' + model);
    var object = new Class(res, req);
    object[controller].call();
}

```

【代码说明】

- ❑ `pathname = decodeURI(url.parse(req.url).pathname)`: 解码 url, 避免中文字符导致无法解析;
- ❑ `pathArr = pathname.split('/')`: 将 path 字符串使用/分割, 并得到数组对象, 例如['', 'index', 'show'];
- ❑ `model = pathArr[1]`: 根据路由规则, 第二个数组元素为模块名;
- ❑ `controller = pathArr[2]`: 根据路由规则, 第三个数组元素为方法名;
- ❑ `Class = require(CON + model)`: require 该模块;
- ❑ `var object = new Class(res, req)`: 创建模块对象;
- ❑ `object[controller].call()`: 动态调用模块中的方法;
- ❑ `model == 'favicon.ico'`: 去除浏览器自带 favicon.ico 请求。

上面的代码是根据路由的规则进行了一个简单的解析过程, 用户访问 `http://127.0.0.1/index/show` 时, `pathname` 将会是 `/index/show`, 当我们使用/分割时, 将会得到一个三个元素的数组, 其中第一个元素是空字符串, 第二个元素是模块名, 而第三个字符则是方法名。如果 `index` 模块和 `show` 方法都存在, 以上代码是完全没有问题的, 但是作为一个用户他可不管你有没有该模块和方法, 当他想访问 `index` 模块中的 `say` 方法时, 由于 `index` 模块中没有提供该方法, 就会导致代码异常, 从而会将整个 Node.js 进程退出, 一个用户的行为会影响整个系统的用户。因此这样的代码设计的健壮性是非常低的, 那么我们应该如何去解决这个问题呢?

利用本章中介绍的两个异常处理办法, 可以将上面的代码修改如下:

```

var url = require('url');
/**
 * @desc HTTP 路由处理逻辑, 将 HTTP 的 url 请求路径进行转发到相应的逻辑处理函数中
 */
exports.router = function(res, req){
    var pathname = decodeURI(url.parse(req.url).pathname);
    var pathArr = pathname.split('/')
        , model = pathArr[1]
        , controller = pathArr[2]
        , Class = '';
    if(model == 'favicon.ico'){ //去除浏览器默认请求 favicon.ico
        return;
    }

    try{ //try catch 捕获异常
        Class = require('./controller/' + model);
    } catch(err){

```

```

        console.log('no this module');
    }

    if(Class){ //判断 class 是否存在
        var object = new Class(res, req);
        if(object[controller]){
            object[controller].call();
        } else {
            console.log(model + ' has no this ' + controller + ' action');
        }
    }
}
/* 无法处理 HTTP 请求逻辑时, 返回默认处理信息 */
var str = 'get static file, but can not find it in server';
res.writeHead(200, { 'Content-Type': 'text/html' });
res.end(str);
}

```

以上代码和前面代码的基本功能是相同的, 但是上面代码中添加了异常处理, 因此用户访问一些不存在的 controller 和 method 的时候就不会导致整个进程中断的情况了。为了验证上面两个方法, 我们创建两个不同的服务器, 分别使用两种路由处理, 来分析异常处理的重要性。

接下来为了测试需要, 我们创建一个 controller 文件夹, 其中有一个 index.js 模块, 该模块中有一个 show 方法, 代码如下:

```

module.exports = function(res, req){
    var _res = res;
    var _req = req;
    /* 定义 show 方法, 响应 success 信息到客户端 */
    this.show = function(){
        var str = 'success, you get show method';
        _res.writeHead(200, { 'Content-Type': 'text/html' });
        _res.end(str);
    }
}

```

【代码说明】

- this.show = function: 响应 HTTP 请求, 并返回字符 success, you get show method;
- _res.writeHead、_res.end: HTTP 响应。

然后我们创建一个 app.js 服务器入口文件, 代码如下:

```

var http = require('http');
var router = require('./router');
var router_update = require('./router_update');

var app = http.createServer(function(req, res) {
    router.router(res, req);
    //router_update.router(res, req);
}).listen(8000);

```

【代码说明】

- router = require('./router'): 应用没有异常处理的路由模块。
- router_update = require('./router_update'): 应用有异常处理的路由模块。

为了测试两者的区别, 我们先将异常处理的路由模块处理方法给注释了。接下来我

们运行 `app.js` 后,在浏览器中输入 url 为 `http://127.0.0.1:8000/index/show`,可以看到如图 6-12 所示的返回信息。

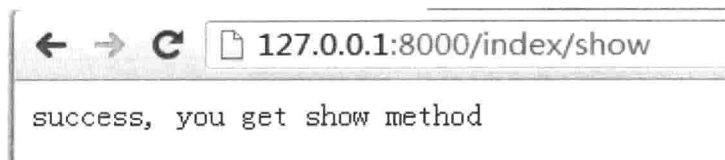
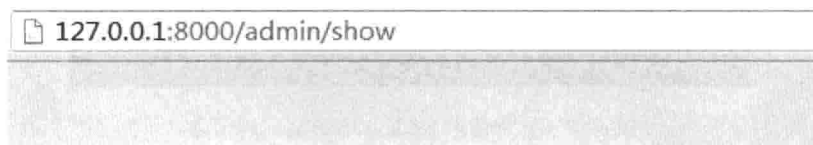


图 6-12 浏览 `http://127.0.0.1:8000/index/show` 响应 Web 页面

上图是正常的访问请求,正好 `index` 模块和 `show` 方法都存在。如果我们现在将 url 请求改为 `http://127.0.0.1:8000/index/say` 或者 `http://127.0.0.1:8000/admin/show` 时,HTTP 无任何返回,并且服务器的 Node.js 进程将会中断,如图 6-13 所示。



糟糕! 谷歌浏览器无法连接到 127.0.0.1:8000

图 6-13 访问异常 url 系统响应页面

由于该服务器没有提供 `admin` 模块和 `show` 方法,因此服务器无法正常地返回,由于服务器没有做异常处理,从而导致服务器直接返回如图 6-14 所示的错误信息。

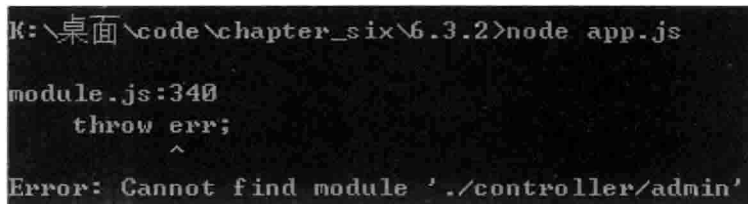


图 6-14 Node.js 服务端异常错误提示

如图 6-14 所示,当用户请求一个不存在的模块或者方法时,导致了进程中断,然后我们再试图访问 `http://127.0.0.1:8000/index/show` 时,也没有办法成功访问了,因此可以说一个用户的行为影响了整个系统的运行,这是不可取的。接下来我们使用异常处理后的路由来运行 `app.js`,将 `app.js` 中的 `router.router(res, req)` 注释,并将 `router_update.router(res, req)` 注释去掉,执行 `app.js` 脚本,打开浏览器再次访问 `http://127.0.0.1:8000/index/show`,同样可以看到如图 6-15 所示的返回。

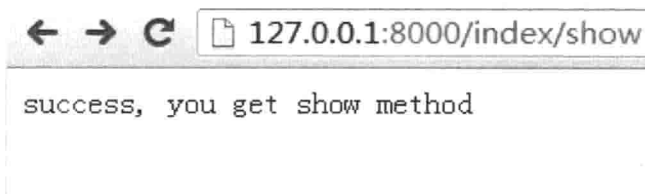


图 6-15 正常 HTTP 响应返回

依照前面的测试方法,我们现在请求 `http://127.0.0.1:8000/index/say` 或者 `http://127.0.0.1:8000/admin/show`, 页面会正常地返回如图 6-16 所示的信息, 同样服务器端也不会中断进程, 而是提示了相应的错误信息。

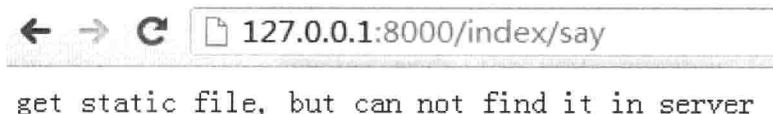


图 6-16 异常 HTTP 响应返回

由于该路由处理部分做了相应的异常处理逻辑, 因此当遇到不存在模块或者方法时, 服务器端则会提示相应的错误, 而不是直接中断退出。如图 6-17 所示为服务器的正常输出。

```
K:\桌面\code\chapter_six\6.3.2>node app.js
index has no this say action
```

图 6-17 异常请求 Node.js 服务端日志信息

那么我们再次访问 `http://127.0.0.1:8000/index/show` 时, 服务器还能够正常地返回, 而非和之前一样导致进程中断, 服务器无法响应的情况。

从上面我们可以看到异常处理对服务器健壮性的重要性, 避免一个用户的行为影响整个系统用户这种低级错误。

6.3.5 异常情况汇总

本节只介绍 3 种常见的异常逻辑, 以及解决办法, 下面简单地回顾这 3 种异步问题, 以及解决原理。

1. require 模块不存在

当 `require` 一个不存在的模块时导致系统异常退出, 解决办法主要是应用 `try catch` 来捕获异常情况, 避免异步退出。

2. 对象的属性或者方法不存在

很多时候我们对 Node.js 的一些 `Json` 对象不做任何判断就调用其属性和方法, 情况最多的就是调用一个 Node.js 的数组, 例如 `a[8]`, 而当 `a[8]` 根本就不存在时, 则会导致 Node.js 代码异常, 从而进程退出。解决办法主要是在应用一些未知对象和数组时, 首先对其返回的结果做判断, 再做赋值操作。

3. for 循环体中的异步函数

`for` 循环中如果包含异步函数时, 而异步函数的执行有依赖 `for` 循环体中的变量时, 如果直接在异步函数中调用 `for` 循环题中的值, 则会导致数据出错, 并不会导致系统强制退出的情况。解决办法是将异步函数依赖的值作为异步函数的参数传递进去即可。

6.4 本章实践

1. Node.js 中编码习惯对变量的命名有哪些要求？

分析：变量命名要求如下所示。

- ❑ 变量是在运行期间会更改的数值，因此对于那些固定的变量，可以使用全局大写字母来定义。
- ❑ 变量代表着一定的意义，每个程序员都需要对变量负责，变量的产生就好比孩子的名字一样，伴随着整个代码的执行过程中，因此一定要描述其含义。
- ❑ 变量的命名建议使用骆驼峰规则，变量的名称描述使用英文，最好不要使用汉语拼音。
- ❑ 多个变量同时需要定义时，定义方式可使用首个变量使用关键字 `var`，其他变量使用逗号分割进行定义。
- ❑ 对于类和对象的变量的命名建议使用首字母大写的骆驼峰规则。

2. Node.js 中有哪几种方法可以帮助开发者优化异步编程带来的困扰？同时用简单的实例来说明如何将 Node.js 中的异步代码应用这些方法进行相应的转化。

分析：有 `Wind.js`、`EventProxy` 和 `Step3` 种方法。相应的解决办法代码请参考本书代码。

3. 本章中介绍了哪几种 Node.js 的代码异常，每种代码异常解决办法是什么？

分析：本章介绍了 3 种异常逻辑，分别是 `require` 模块不存在、对象和属性不存在和 `for` 循环中的异步函数。这 3 种异常逻辑的处理方法见 6.3.5 节。

4. 请选择本章所介绍的任何一种异常逻辑使用 Node.js 进行重现，同时给予一段解决该问题的重构代码。

分析：开发性题目请读者自行完成。

6.5 本章小结

本章主要介绍了 3 个部分：编码规范、Node.js 异步编程和 Node.js 的异常，其中需要了解编码规范，掌握 Node.js 的异步编程，以及处理 Node.js 异常情况的方法。

在 Node.js 的编码规范中介绍了变量、函数、模块和注释的一些规范。对于其中的规范也不是硬性的规定，只是希望大家在编码过程中统一编码习惯，有利于团队合作。

Node.js 的异步编程给程序员带来了许多困扰，打乱了以前的编码常规思维方法，希望读者在阅读本书之后能够对异步编程有一定的了解，同时学会应用 Node.js 中的 NPM 模块来优化异步代码，从而更易于团队合作。

本章的重点是 Node.js 中的异常问题，通过实例介绍来告诉读者在 Node.js 中处理代码异步的必要性。同时举例分析了 Node.js 中常见的几个异常逻辑问题，也给予了相应的解决方式。希望在阅读本章后读者能够掌握这几种异常逻辑的处理办法，在编码过程中遇到异常情况时能够将其总结，并分享给其他 Node.js 的编程爱好者。本章涉及 Node.js 的 API

列表如表 6.1 所示。

表 6.1 本章Node.js的API列表

模 块 名	API 名	作 用
File System	fs.readdirSync(path)	同步读取文件信息
Query String	querystring.parse(str, [sep], [eq], [options])	字符串与对象之间的转化
Utilities	util.inherits(constructor, superConstructor)	类之间的继承

第 7 章 Node.js 与数据库

本章介绍 Node.js 与数据库的相关知识，其中包含 MySQL 和 MongoDB 在 Node.js 编程语言中的实例应用。通过本章的学习，希望读者能够掌握 Node.js 操作 MySQL 和 MongoDB 数据库的知识。本章会简单概述 MySQL 和 MongoDB 的概念，然后介绍 MySQL 和 MongoDB 的配置安装和 Node.js 的 MySQL、MongoDB 实例应用。根据实例应用，实践开发 Node.js 的 MySQL、MongoDB 的操作基类模块，本章后部分则将 MySQL 和 MongoDB 二者数据库进行一个压测对比，通过对比诠释两者在 Node.js 应用性能的优点和不足之处。

通过本章的学习希望大家掌握 Node.js 的 MySQL 和 MongoDB 数据库连接，在应用开发中能够灵活的选择 Node.js 数据，并掌握 Node.js 数据库操作的两个基类模块的应用。

7.1 两种数据库介绍

本节介绍 Node.js 操作的两种数据，分别为 MySQL 和 MongoDB。本节前面部分会简单阐述 MySQL 和 MongoDB 的一些概念性的知识，读者可以进行简单了解。在本节后半部分，则会介绍 Node.js 中的 MySQL 和 MongoDB 的 NPM 模块，而这部分知识点需要读者能够掌握，并能简单地应用 NPM 模块，实现简单的例子。

7.1.1 MySQL 介绍

MySQL 是一个关系型数据库管理系统，由瑞典 MySQL AB 公司开发，目前属于 Oracle 公司。MySQL 是一种关联数据库管理系统，关联数据库将数据保存在不同的表中，而不是将所有数据放在一个大仓库内，这样就增加了速度并提高了灵活性。MySQL 的 SQL 语言是用于访问数据库的最常用标准化语言¹。

MySQL 在过去由于性能高、成本低、可靠性好，已经成为最流行的开源数据库，因此被广泛地应用在 Internet 上的中小型网站中。随着 MySQL 的不断成熟，它也逐渐用于更多大规模网站和应用，比如维基百科、Google 和 Facebook 等网站。非常流行的开源软件组合 LAMP 中的“M”指的就是 MySQL²。

MySQL 数据库被广泛应用在 PHP、Java 和 Python 编程语言中，其中每一种语言都提供了相应连接 MySQL、操作 MySQL 的 API。Node.js 同样也有相应的 API 方法，该 MySQL

1 参见网站 <http://baike.baidu.com/view/24816.htm>。

2 参见网站 <http://zh.wikipedia.org/wiki/MySQL>。

API 方法是由 `felixge` 提供的 NPM 模块¹。接下来我们看一下 Node.js 的 MySQL 提供了哪些 API，以及每个接口的用法和作用，如表 7.1 所示。

表 7.1 Node.js 的 MySQL API 接口

MySQL API	接口作用	参数说明	返回
<code>mysql.createConnection()</code>	创建 MySQL 连接	参数见本章的表格 7.2	Connection 对象
<code>Connection.connect(function(err){})</code>	判断创建 MySQL 连接是否成功	<code>function(err){}</code> ，接受一个回调函数，其中 <code>err</code> 判断是否成功连接	无
<code>Connection.query(sql, function(err, rows))</code>	执行 sql 查询或者删除语句，执行结束后调用回调函数，通过回调函数传递异步执行的返回结果 <code>rows</code> 或者 <code>err</code>	sql: 需要执行 sql 语句 <code>function()</code> {}: 执行 sql 后的回调函数	无
<code>Connection.query(sql, paramInfo, function(err, rows))</code>	该接口和上面接口类似，但其可将 sql 的参数作为对象通过 <code>paramInfo</code> 传递。	sql: 需要执行 sql 语句 paramInfo: sql 中对应的变量 <code>function()</code> {}: 执行 sql 后的回调函数	无
<code>Connection.escape(str)</code>	sql 转义，防止 sql 注入	str 为需要转义的字符串	转义后的字符串

接下来我们来看一些 Node.js 操作 MySQL 的例子。在应用该模块前，我们需要在项目根目录中安装 MySQL 模块。执行如下命令下载安装：

```
npm install mysql
```

相应的 Node.js 操作 MySQL 的实例代码如下：

```
var mysql = require('mysql');
/* 创建 MySQL 连接对象 */
var connection = mysql.createConnection({
  host      : 'example.org',
  user      : 'bob',
  password  : 'secret',
});

connection.connect(function(err) {
  //connected! (unless 'err' is set)
});
connection.end(function(err) {
  //The connection is terminated now
});
```

【代码说明】

- ❑ `mysql = require('mysql')`: 获取 MySQL 的模块对象。
- ❑ `connection = mysql.createConnection({})`: 创建 MySQL 连接。
- ❑ `connection.connect(function(err){})`: 判断是否成功连接 MySQL。

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

❑ `connection.end(function(err){})`: 关闭数据库连接。

上述代码中 `mysql.createConnection` 方法接受了一个 `json` 对象参数, 该对象中 `host` 为连接的 MySQL 服务器, `user` 是连接 MySQL 的用户名, `password` 则为连接的密码, 同时应用 `connection.connect` 来判断是否成功连接 MySQL。

上述代码中, 还涉及关于关闭 MySQL 数据库的方法。关闭数据库有两种方式, 上面的代码中所使用的是 `connection` 对象的 `end` 方法, 这种方式可以获取关闭失败后的返回结果。另外一种关闭 MySQL 连接的方法, 代码如下:

```
connection.destroy();
```

这种关闭方式略显暴力, 就是强制性关闭连接。如果希望在 MySQL 关闭时做一些其他处理, 建议使用 `connection.end`。接下来我们看一下 Node.js 操作查询 MySQL 的实例, 代码如下:

```
var mysql = require('mysql');
/* 创建 MySQL 连接对象 */
var connection = mysql.createConnection({
  host    : 'example.org',
  user    : 'bob',
  password : 'secret',
});

connection.query('SELECT * FROM table', function(err, rows) {
  //connected! (unless 'err' is set)
});
```

【代码说明】

❑ `connection.query('SELECT * FROM table', function(err, rows){})`: 执行 MySQL 语句。

上述代码中 `connection.query` 函数的第一个参数为 MySQL 命令字符串, 例如本例中的 `SELECT * FROM table`。该函数的第二个参数为一个回调函数, 大部分同学在习惯了写同步代码之后 (如 PHP), 再使用 Node.js 代码来实现某些需求时, 往往会在获取函数的执行结果上不适应。按照同步代码的写法, 一般大部分读者会通过如下方式:

```
var rows = connection.query('SELECT * FROM table', function(err, rows) {
  //connected! (unless 'err' is set)
});
console.log(rows);
```

上述代码运行后, 我们才发现输出的结果是一个 `undefined`, 这个结果并不是我们需要查询返回的数据。其主要问题是 `rows` 这个变量值是 `connection.query` 函数的返回结果, 而这个函数是一个异步函数, 其函数结果并没有返回任何值, 因此会出现 `undefined`。细心的读者可以看到, 在回调函数内部有一个 `rows` 变量, 因此又有部分读者会使用下面的代码来获取返回结果:

```
var result;
connection.query('SELECT * FROM table', function(err, rows) {
  result = rows;
});
console.log(result);
```

结果同样输出 `undefined`。原因是 `connection.query` 函数是一个异步执行函数，在 `connection.query` 函数还未结束时，已经执行了 `console.log(result)`，因此 `result` 是无法成功赋值的。有很多读者会在 `console.log(result)` 前加一个等待，等待 `connection.query` 函数的执行结束以后，再返回执行结果。当然，这样的方式可以得到返回结果，但是却忽略 Node.js 本身就是一个异步代码执行过程，这样强加同步执行过程，会降低 Node.js 的语言的作用。既然上面两种方式都无法达到获取返回结果，那么我们应该如何获取呢？代码如下：

```
/**
 *
 * @desc 应用 query 来执行 MySQL 查询数据
 * @params callback 回调函数
 */
function querySql(callback){
    connection.query('SELECT * FROM table', function(err, rows) {
        if(err) {
            console.log(err);
            return;
        }
        callback(rows);
    });
}
/**
 *
 * @desc 定义回调函数 callback
 * @params rows array
 */
function callback(rows){
    console.log(rows);
}
```

上述代码中 `querySql` 函数中添加了一个函数参数，作为回调函数，在 `query` 异步执行结束后调用 `callback` 函数，由此 `callback` 函数就可以获取 `query` 查询返回的结果 `rows`，从而我们可以在 `callback` 中处理返回的 `rows` 结果。这就是编写异步执行代码的方式，当然这种方式会造成异步回调深度的问题，NPM 模块中也有很多处理回调深度的方式，例如老赵的 `wind.js`¹、`step`² 和 `async`³，如果大家有兴趣可以查看脚注的 url 地址学习了解。

在 Node.js 中拼凑 sql 语句的方式有多种，其中较为常用的是应用问号 (?) 替代 json 对象和直接字符串连接方法。

```
/* 应用连接对象 connection 执行 MySQL 插入 */
connection.query('INSERT INTO posts SET ?', {title: 'test'}, function(err, result) {
    if (err) throw err;
    console.log(result.insertId);
});
```

【代码说明】

❑ `INSERT INTO posts SET ?`: 这句 sql 语句相当于 `INSERT INTO posts SET title =`

1 参见网站 <https://github.com/JeffreyZhao/wind>。
 2 参见网站 <https://github.com/creationix/step>。
 3 参见网站 <https://github.com/caolan/async> caolan async。

'test'。

还可以使用如下方式在 sql 语句中填写参数，代码如下：

```
/* 应用连接对象 connection 执行 MySQL 插入 */
connection.query('INSERT INTO posts SET title = :title', {title: 'test'},
function(err, result) { //异步获取 MySQL 执行的返回结果
    if (err) throw err;
    console.log(result.insertId);
});
```

当然，还有最简单的方式，直接将参数和字符拼凑成一个字符，代码如下：

```
Var title = 'test';
connection.query('INSERT INTO posts SET title = ' + title, function(err,
result) {
    if (err) throw err;
    console.log(result.insertId);
});
```

connection.query 方法还可以同时执行多个 query 的 sql 语句，代码如下：

```
/* 应用连接对象 connection 执行 MySQL 查询 */
connection.query('SELECT 1; SELECT 2', function(err, results) {
    if (err) throw err;

    //'results' is an array with one element for every statement in the query:
    console.log(results[0]); //{1: 1}
    console.log(results[1]); //{2: 2}
});
```

这里执行了两个 sql 语句，返回的结果 results 中是一个数组，其中包含了两个 sql 执行的结果。

```
var sorter = 'date';
var query = 'SELECT * FROM posts ORDER BY ' + mysql.escapeId(sorter);

console.log(query); //SELECT * FROM posts ORDER BY 'date'
```

Node.js 为了防止 MySQL 的注入¹，提供了 mysql.escapeId 转义字符方法。关于更多 MySQL 注入的知识大家可以去网上学习了解。

以上就是关于本节 MySQL 的知识点介绍，在 7.2 节中会通过实例来实践应用 Node.js 操作 MySQL。MySQL 的 NPM 模块文档中介绍，该模块有一个优点，就是传递参数类型出现错误时，导致系统运行出现异常，也不会立即中断 Node.js 的代码运行，而是会抛出异常。这点是我们在 Node.js 编码时需要非常注意的一个问题，要保持代码的健壮性，避免出现一个细节问题，导致系统无法运行，或者直接奔溃。对于 PHP 来说当某部分出现代码异常时，不会导致整个项目崩溃，而 Node.js 如果没有很好的处理异常是会导致系统退出运行的。

7.1.2 MongoDB 模块介绍

MongoDB 是一个基于分布式文件存储的数据库，由 C++语言编写，旨在为 Web 应用

1 程序执行中未对敏感字符进行过滤，使得攻击者传入恶意字符串与结构化数据查询语句合并，并且执行恶意代码。

提供可扩展的高性能数据存储解决方案。

MongoDB 是一个介于关系数据库和非关系数据库之间的产品，是非关系数据库当中功能最丰富，最像关系数据库的。它支持的数据结构非常松散，是类似 json 的 bson 格式，因此可以存储比较复杂的数据类型。MongoDB 最大的特点是它支持的查询语言非常强大，其语法类似面向对象的查询语言，几乎可以实现类似关系数据库单表查询的绝大部分功能，而且还支持对数据建立索引¹。

Node.js 的 MongoDB NPM 模块是由 MongoDB 提供，其中包括数据库连接和一些基本操作。接下来将着重介绍 Node.js 的 MongoDB 驱动 NPM 模块。

node-mongodb-native 是一个操作 MongoDB 的 NPM 模块。下面我们先看一下 node-mongodb-native 的安装。在项目根目录执行如下命令下载安装 MongoDB 驱动模块。

```
npm install node-mongodb-native
```

首先看 node-MongoDB-native 模块提供数据库的连接方法，代码如下：

```
var MongoDB = require('mongodb'); //获取 MongoDB 模块
/* 连接本地 27017 端口下的 MongoDB 服务器 */
var server = new mongodb.Server("127.0.0.1", 27017, {});

/* 创建数据库操作对象 */
var db = new mongodb.Db('test', server, {});
db.open(function (error, client) {
  client.Collection('test_collection', function(err, collection){
    //
  });
});
```

【代码说明】

- ❑ var mongodb = require('mongodb'): 获取 MongoDB 对象。
- ❑ var server = new mongodb.Server("127.0.0.1", 27017, {}): 创建本地数据库连接服务器。
- ❑ client.Collection('test_collection'...: 创建 MongoDB 连接对象，其中 test_collection 类似于 MySQL 的表。
- ❑ db.open(function (error, client): 创建数据库连接。
- ❑ collection = client.Collection: 获取需要操作的表句柄。

mongodb.Server 对象接收三个参数，第一个为 IP 地址；第二个为端口号，端口号使用 Number 类型，使用字符串会出现异常；第三个为可选配置参数。db.open(function (error, client) 类似于创建数据库连接。

通过 client.Collection 异步获取 collection 函数后，接下来应用 collection 表的句柄对象来操作表单的数据插入、查询、修改和删除操作的方法实现。

数据插入接口 collection.insert(docs, options, [callback]) 如果需要执行回调函数时，在插入时都需要设置 options = {safe:true}。数据插入示例如下：

```
collection.insert({hello: 'world'}, {safe:true},
  function(err, objects) {
    if (err) console.warn(err.message);
```

1 参见网站 <http://baike.baidu.com/view/3385614.htm>。

```

        Console.log(objects);
    }
});

```

向数据库插入 {hello: 'world'} 的 json 对象数据, 数据插入后执行回调函数, objects 返回的是插入后的 ObjectId。

数据更新接口为 collection.update(whereJson, newInfo, options, [callback]), whereJson 为条件查询, newInfo 为需要更新的数据, options 和 insert 接口中的 options 作用一致。下面是一个 update 接口使用实例, 代码如下:

```

collection.update({hi: 'here'}, {$set: {hi: 'there'}}, {safe:true},
    function(err) {
        if (err) console.warn(err.message);
        else console.log('successfully updated');
    });

```

collection.update 的第一个参数是条件查询, 第二个为 update 的数据, 使用 \$set 表示只更新 hi 这个 key 值的数据, 不会修改其他数据, {safe:true} 为 options 参数。

数据删除接口是 collection.remove(whereJson, callback), whereJson 为条件查询, callback 带有一个参数, 判断是否删除成功。示例代码如下:

```

collection.remove({'id':mongoId}, function(err){
    if (err) {
        console.log(err);
    } else {
        Console.log('remove success');
    }
});

```

删除 MongoDB 中 id 为 mongoId 的数据, 删除成功, 输出 'remove success', 否则输出 err 信息。

数据查询接口是 collection.find(query, [fields], options), query 是查询条件, fields 是需要返回的字段, 其格式为 {field1: -1, field2: 1}, 1 为需要返回的字段, -1 为不需要返回的字段, options 和 insert、update 中的 options 作用是一致的。find 方法类似于一个工厂对象创建的方法, 通过 find 方法创建返回一个 Cursor 对象, 该对象还可以附加查询条件, 例如 sort、limit、nextObject、each 和 toArray 等。示例代码如下:

```

var cursor = collection.find(query, [fields], options);
//设置数据库查询时 sort 排序以及数量限制
cursor.sort(fields).limit(n).skip(m).

cursor.nextObject(function(err, doc) {});
//获取查询结果
cursor.each(function(err, doc) {});
//将结果转化为数组
cursor.toArray(function(err, docs) {});

cursor.rewind() //reset the cursor to its initial state.

```

collection.find 方法创建并返回 cursor 对象, 之后调用其 sort、limit 和 skip 等方法, 通过 nextObject、each 和 toArray 方法来获取最后查询的数据, 最后再将 cursor 对象重置为初始状态。

以上就是关于本节 MongoDB 和 MongoDB 的 NPM 模块介绍。在 7.3 节中会通过实例来介绍使用 Node.js 操作 MongoDB、实践 MongoDB 的操作基类，以及该基类的应用。本节是基于最基本的一些 MongoDB 的驱动操作 API 接口的应用，关于其封装实现方法还需要进一步地深入实现。

7.2 Node.js 与 MySQL

本节将介绍 MySQL 的安裝配置，结合 Node.js 的 MySQL 模块实践性的介绍一些小应用，最后根据实践应用总结出一个 MySQL 操作基类。

7.2.1 MySQL 安裝配置应用

MySQL 安裝配置在网上有一些教程，这里只简单介绍安裝配置过程，以及使用 MySQL 创建一些基础的表单和字段的方法。

执行指令安装 mysql-server，需要注意的是 MySQL 有 MySQL server 和 client 两个模块。这两个模块中 client 是用来连接 server 的，当然 client 既可以使用 MySQL 自带的软件连接测试，也可以使用其他语言的 MySQL 驱动，例如 PHP 和 Node.js。

```
sudo apt-get install mysql-server
```

执行 mysql --version 查看是否安装成功，版本信息如图 7-1 所示。

```
root@ubuntu:~# mysql --version
mysql Ver 14.14 Distrib 5.5.28, for debian-linux-gnu (x86_64) using readline 6.2
root@ubuntu:~#
```

图 7-1 执行 mysql --version 返回结果

确认成功安装 MySQL 后，我们在 MySQL 中创建一个 node_test 数据库，在数据库中添加 node_book 表，该表有 book_id、book_name、author 和 time 字段。设计的表单如表 7.2 所示。

表 7.2 MySQL 数据库表单字段设计

字 段 名	字 段 类 型	字 段 描 述
book_id	int(11)	主键
book_name	varchar(100)	书籍名称
author	varchar(100)	作者
time	timestamp	书本时间

分别执行下面的命令：

```
mysql          ——进入 MySQL 操作命令模式
create database node_test; ——创建 node_test 数据库。
CREATE TABLE IF NOT EXISTS 'node_book' (
  'book_id' int(11) NOT NULL,
  'book_name' varchar(100) COLLATE utf8_bin NOT NULL,
  'author' varchar(100) COLLATE utf8_bin NOT NULL,
```

```
'time' timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
PRIMARY KEY ('book_id')
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_bin;
```

以上语法创建数据库 `node_book` 表。MySQL 语法与应用也是一门必修的软件工程课程，如果希望学习更多的关于 MySQL 的应用可以查看《高性能 MySQL》¹这本书。

上面简单地介绍了 MySQL 数据库在 Linux 下的配置安装。Windows 下可以去软件应用中搜索 `apache` 和 `wampserver`，`wampserver` 中有相应的数据库应用。

关于 MySQL 的应用这里需要说明：本书的着重点不是介绍 MySQL 的应用，因此如果大家遇到关于 MySQL 的问题，可以找其他书籍学习了解，如果有问题也可以联系笔者。

7.2.2 MySQL 数据库接口设计

Node.js 使用 MySQL 前必须要先安装 MySQL 的 Node.js 驱动，也就是 Node.js 的 MySQL 模块。安装方法在 7.1 节中已经提及，执行 `npm install mysql` 安装到项目目录中。安装完成后我们来实现一个 Node.js 的 MySQL 操作基类 `BaseModel`，该 `BaseModel` 基类包含 Node.js 的 MySQL 数据库连接、插入、修改、删除、条件获取数据列表和数据转义等操作方法。`BaseModel` 类的设计方法，如表 7.3 所示。

表 7.3 Node.js 的 MySQL 基类设计接口明细

方法名	参数	参数说明	返回值	描述
<code>__constructor</code>	无	无	无	数据库连接
<code>insert</code>	<code>tableName</code>	string 表名	回调函数返回，成功返回 <code>id</code> ，失败返回 <code>false</code>	数据插入
	<code>rowInfo</code>	array 插入数据		
<code>modify</code>	<code>tableName</code>	string 表名	回调函数返回，成功返回 <code>true</code> ，失败返回 <code>false</code>	数据修改
	<code>id</code>	int 主键值		
	<code>rowInfo</code>	array 插入数据		
<code>remove</code>	<code>tableName</code>	string 表名	回调函数返回，成功返回 <code>true</code> ，失败返回 <code>false</code>	数据删除
	<code>id</code>	int 主键值		
<code>findOneById</code>	<code>tableName</code>	string 表名	回调函数返回，成功返回查询的数据，失败返回 <code>false</code>	查询单条
	<code>id</code>	int 主键值		
<code>find</code>	<code>whereJson</code>	Json 查询条件	回调函数返回，成功返回查询的数据，失败返回 <code>false</code>	条件查询
	<code>orderByJson</code>	Json 排序		
	<code>limitJson</code>	Json 数量		
	<code>fieldsJson</code>	Json 获取字段		

¹ 荣获 2009 年 Jolt 图书大奖，是不可多得的分享 MySQL 实用经验的图书。它不但可以帮助 MySQL 初学者提高使用技巧，更为有经验的 MySQL DBA 指出了开发高性能 MySQL 应用的途径。全书包含 14 章和 4 个附录，内容覆盖 MySQL 系统架构、设计应用技巧、SQL 语句优化、服务器性能调优、系统配置管理和安全设置、监控分析，以及复制、扩展和备份/还原等主题，每一章的内容自成体系，适合各领域技术人员做选择性的阅读。

7.2.3 数据库连接

因为上述 MySQL 操作的方法都是一个异步函数，因此我们需要为每个异步调用函数添加一个 `callback` 回调函数，来获得执行结果。综上设计方案我们可以简单地绘制出 `BaseModel` 的大体类代码结构，代码如下：

```
/**
 * @type class BaseModel
 * @author danhuang
 * @time 2012-12-22
 * @desc desc base_model.js
 */
module.exports = function() {
  __constructor();
  /* 数据查询接口 */
  this.findOneById = function(tableName, id, callback){
  };
  /* 数据插入接口 */
  this.insert = function(tableName, rowInfo, callback){
  };
  /* 数据修改接口 */
  this.modify = function(tableName, id, rowInfo, callback){
  };
  /* 数据删除接口 */
  this.remove = function(table, id, callback){
  };
  /* 数据条件查询接口 */
  this.find = function(tableName, whereJson, orderByJson, limitArr,
    fieldsArr, callback){
  };
  function __constructor(){}
}
```

【代码说明】

- ❑ `__constructor()`：调用自身的构造函数，其方法在本函数中定义。
- ❑ `this.findOneById`：此处主要是为模块类添加公有方法 `findOneById`，本模块涉及的其他 `this` 方法功能都与此处类似。
- ❑ `function __constructor(){}：`定义私有构造函数 `__constructor`。

接下来我们就先从 `BaseModel` 的构造函数开始实现。`__constructor` 涉及数据库的连接，因此需要一些配置信息，这里通过一个 `util.js` 工具类来存放一些公有方法，包含 `json` 配置文件解析的方法。相关代码如下：

```
/**
 * @type module
 * @author danhuang
 * @time 2012-12-22
 * @desc desc util.js
 */
var fs = require('fs')
, sys = require('util');
exports.util = function(fileName, key){
  var configJson = {};
```

```

try{ //try catch 读取文件信息
    var str = fs.readFileSync(fileName,'utf8');
    configJson = JSON.parse(str);
}catch(e){
    sys.debug("JSON parse fails")
}
return configJson[key];
}

```

【代码说明】

- ❑ `fs.readFileSync(fileName,'utf8')`: 以 utf8 格式同步读取配置文件信息。
- ❑ `configJson = JSON.parse(str)`: 使用 JSON 的 `parse` 方法解析读取后的配置文件内容转化为 json 对象。
- ❑ `return configJson[key]`: 返回需要的 key 值配置信息。

需要注意的是，配置文件必须使用 utf8 编码格式，因为我们使用的是 utf8 格式读取，否则会出现乱码或者 JSON 解析出错等问题。当然也可以使用其他编码来保存配置信息，但必须保证写入的编码格式和读取的格式是一致的。

应用 util 工具类模块，实现 BaseModel 的构造函数，代码如下：

```

function __constructor(){
    var dbConfig = Util.get('config.json', 'db');
    /* 获取 MySQL 配置信息 */
    client = {};
    /* 读取配置文件中 MySQL 的 host 值 */
    client.host = dbConfig['host'];

    /* 读取配置文件中 MySQL 的 port 端口 */
    client.port = dbConfig['port'];

    /* 读取配置文件中 MySQL 的数据库用户名 */
    client.user = dbConfig['user'];

    /* 读取配置文件中 MySQL 的数据库密码 */
    client.password = dbConfig['password'];

    /* 根据配置文件，创建 MySQL 连接 */
    dbClient = mysql.createConnection(client);
    dbClient.connect();
    /* 执行 MySQL 指令，连接 MySQL 服务器的一个数据库 */
    dbClient.query('USE ' + dbConfig['dbName'], function(error,
    results) {
    /* 回调处理 MySQL 连接结果 */
        if(error) {
            console.log('ClientConnectionReady Error: ' +
            error.message);
            dbClient.end();
        }
        console.log('connection local mysql success');
    });
}

```

【代码说明】

- ❑ `dbConfig = Util.get('config.json', 'db')`: 读取 config.json 配置文件，并获取其中 db 的配置文件信息。

- ❑ `client.host = dbConfig['host']`: 数据库连接服务器的 `host`。
- ❑ `client.port = dbConfig['port']`: 数据库连接服务器的端口。
- ❑ `client.user = dbConfig['user']`: 数据库连接的用户名。
- ❑ `client.password = dbConfig['password']`: 数据库连接的用户密码。
- ❑ `dbClient = mysql.createConnection(client)`: 创建 MySQL 服务器连接对象。
- ❑ `dbClient.connect()`: 连接 MySQL 服务器。
- ❑ `client.query('USE ' + dbConfig['dbName'] + '');`: 执行数据的 MySQL 操作, `use` 关键字是来连接 MySQL 中的一个 MySQL 数据库。

这里的 `Util` 和 `client` 变量是 `BaseModel` 模块类的全局变量, 两者分别是 `require` 工具模块 `util` 和模块 `MySQL` 返回的对象。其声明方法如下:

```
var Util = require('./util')
, mysql = require('mysql')
, dbClient;
```

【代码说明】

- ❑ `mysql = require('mysql')`: 获取 MySQL 模块对象。
- ❑ `dbClient`: 全局的 MySQL 连接句柄。

以上实现了 Node.js 的 MySQL 数据库连接功能, 现在我们添加一个测试脚本文件 `index.js` 来验证该函数是否能够成功连接数据库, `index.js` 测试代码如下:

```
var BaseModel = require('./base_model');
var baseModel = new BaseModel();
```

简单实例化 `BaseModel` 的一个 `baseModel` 对象, 在实例化时会调用 `BaseModel` 的构造函数, 从而验证是否能够成功执行构造函数中的数据库连接。执行 `index.js` 脚本文件, 返回结果如图 7-2 所示, 表明已经成功连接到 `node_test` 数据库。

```
E:\Desktop\code\chapter_seven\7.2>node index.js
connection local mysql success
```

图 7-2 执行 `index` 脚本返回结果

看到了这个结果是不是觉得很开心? 成功连接 MySQL 数据库后, 我们就可以向数据库插入数据了。接下来我们看一下 `insert` 函数的实现。

7.2.4 数据库插入数据

```
/**
 *
 * @desc 向数据库插入数据
 * @param tableName string
 * @param rowInfo json
 * @param callback function
 * @return null
 */
this.insert = function(tableName, rowInfo, callback){
    dbClient.query('INSERT INTO ' + tableName + ' SET ?', rowInfo,
        function(err, result) {
```



```

        if (err) throw err;
        callback(result.insertId);
    });
};

```

【代码说明】

- ❑ dbClient.query('INSERT INTO ' + tableName + ' SET ?', rowInfo...: ?字符会将 rowInfo 的 json 数据转化为 key=value 的形式。
- ❑ callback(result.insertId): result.insertId 为插入数据后返回的主键值，并将执行结果交由回调函数处理。

MySQL 模块自带了将?替换为 json 数据的 key=value 形式，同时还提供了获取插入数据的 key 值方法（应用 result.insertId）。方法实现完毕以后，我们在 index.js 中新增插入数据测试方法，修改后的 index.js 代码如下：

```

var BaseModel = require('./base_model')
,   BaseModel = new BaseModel()
,   rowInfo = {};

/* 设置数据库插入对象 rowInfo */
rowInfo.book_name = 'nodejs book';
rowInfo.author = 'danhuang';

/* 执行 MySQL 数据库插入 */
baseModel.insert('node_book', rowInfo, function(ret){
    console.log(ret);
});

```

【代码说明】

- ❑ rowInfo = {}: 初始化一个 json 对象。
- ❑ rowInfo.book_name: 设置 json 需要插入数据库的 book_name 变量。
- ❑ rowInfo.author: 设置 json 需要插入数据库的 author 变量。
- ❑ baseModel.insert('node_book', rowInfo...: 调用 baseModel 中的 insert 基类方法，向数据库插入数据。
- ❑ console.log(ret): 打印回调函数获取的返回结果。

因为 baseModel.insert 是一个异步函数，因此我们需要添加一个回调函数来获取执行结果 ret，如上代码中的 function(ret)。两次执行 index.js，可以看到如图 7-3 所示的返回结果。

```

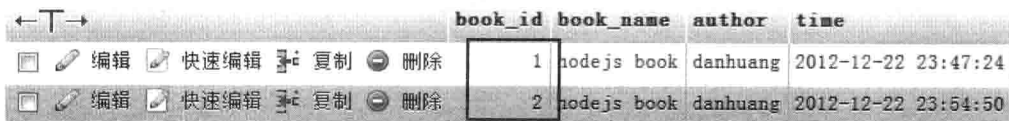
E:\Desktop\code\chapter_seven\7.2>node index.js
connection local mysql success
1
^C
E:\Desktop\code\chapter_seven\7.2>node index.js
connection local mysql success
2

```

图 7-3 MySQL 插入数据返回结果

执行第一次时，成功连接并插入数据后返回 key 值为 1，执行第二次时，同样连接数据库并返回插入的 key 值为 2，由于这里是两次独立的运行 index.js 脚本，因此会连接数据库两次。接下来我们看一下数据库是否成功插入两个数据。

在 Windows 版本下可以通过 wampserver 中的 phpMyAdmin 进行查看，Linux 下进入终端窗口依次执行下面指令：mysql、use node_test;、select * from node_book。



	book_id	book_name	author	time
	1	nodejs book	danhuang	2012-12-22 23:47:24
	2	nodejs book	danhuang	2012-12-22 23:54:50

图 7-4 MySQL 数据库查询结果

如图 7-4 所示是 Windows 版本下 phpMyAdmin 数据库管理显示的数据，其中 book_id 的两个 key 值分别是 1 和 2，与之前执行返回的结果一致，同时成功插入了两条数据，表明 BaseModel 的 insert 接口成功实现。

7.2.5 查询一条数据记录

接下来我们看一下 BaseModel 的 findOneById 实现，要注意的是，该函数的实现只返回一条 json 数据，而不是一个数组。因此需要将返回的数组转化为单条数据，代码实现如下所示。

```
/**
 *
 * 根据主键 id 值查询数据库的一条记录
 * @param tableName string
 * @param idJson id
 * @param callback function
 * @return null
 */
this.findOneById = function(tableName, idJson, callback){
    dbClient.query('SELECT * FROM ' + tableName + ' where ?', idJson,
        function(error, results) {
            if (error) {
                console.log('GetData Error: ' + error.message);
                dbClient.end();
                callback(false);
            } else {
                if(results){ //如果查询到数据则返回一条数据
                    callback(results.pop());
                } else{ //查询数据为空则返回空数据
                    callback(results);
                }
            }
        })
    };
};
```

【代码说明】

- ❑ 'SELECT * FROM ' + tableName + ' where ?': 此处的字符?会将当前的 idJson 转化为 key=value 的字符串数据。
- ❑ callback(false): 数据库查询出现异常时，返回 false，而不是空数据。
- ❑ callback(results.pop()): 如果数据不为空，应用 pop 函数返回得到单条 json 数据。
- ❑ callback(results): 如果是空数据，则直接将执行结果 results 返回。

在上述代码中，为了保存返回的是单条数据，使用 `pop` 数组函数将数据对象转换为单条 `json` 数据。因为通过主键查询时，如果有数据返回，则一定只有一条数据，但 `select` 返回的是一个数组，该数组元素只有一条记录（也就是查询的结果），因此我们需要将数组的一个元素弹出来并返回。

添加测试代码，`select` 一个 `id` 为 1 的数据，为了更好地体现使用 `pop` 的目的，我们把最初 `select` 的 `results` 也打印出来进行对比。代码如下：

```
var BaseModel = require('./base_model')
,   BaseModel = new BaseModel()
,   rowInfo = {}
,   tableName = 'node_book';

/* 设置数据库插入对象 rowInfo */
rowInfo.book_name = 'nodejs book';
rowInfo.author = 'danhuang';

/* findOneById 验证 */
var idJson = {'book_id': 1};

/* 执行 MySQL 数据查询 */
baseModel.findOneById(tableName, idJson, function(ret){
    console.log(ret);
});
```

上述调用方法和 `insert` 插入数据函数接口的调用方法相同。执行 `index.js` 进行查询数据，返回结果如图 7-5 所示。

```
E:\Desktop\code\chapter_seven\7.2>node index.js
connection local mysql success
results:
[ < book_id: 1,
  book_name: 'nodejs book',
  author: 'danhuang',
  time: Sat Dec 22 2012 23:47:24 GMT+0800 (中国标准时间) > ]
< book_id: 1,
  book_name: 'nodejs book',
  author: 'danhuang',
  time: Sat Dec 22 2012 23:47:24 GMT+0800 (中国标准时间) >
```

图 7-5 MySQL 单条记录查询返回结果

如图 7-5 所示，第一次打印出来的是 `results` 查询的结果，可以看到是一个数组，数组元素就是查询主键值返回的结果。第二个打印的是一个 `json` 数据，为函数执行后返回的结果，满足我们 `findOneById` 的要求。为什么这里一定要强调使用 `pop` 弹出数组元素呢？因为很多程序员在执行完查询语句后，都习惯于直接使用如下访问方式：

```
console.log(ret.book_name);
console.log(ret.author);
```

这样访问如果没有 `pop` 时都会提示 `undefined`，这种情况往往会让读者感到疑惑，明明查询成功并且也返回了成功数据，为什么提示 `undefined`？原因就在于 `select` 返回的是数组。如果不使用 `pop` 的话，就必须用如下方式调用：

```
console.log(ret[0].book_name);
console.log(ret[0].author);
```

7.2.6 修改数据库记录

我们继续看 BaseModel 中的方法实现，modify 接口的实现，如下代码：

```
/**
 *
 * @desc 修改数据库的一条数据
 * @param tableName string
 * @param idJson json
 * @param rowInfo json
 * @param callback function
 * @return null
 */
this.modify = function(tableName, idJson, rowInfo, callback){
  dbClient.query('update ' + tableName + ' SET ? where ?', [rowInfo,
    idJson], function(err, result) { //回调调用数据修改模块
    if(err) {
      console.log("ClientReady Error: " + err.message);
      callback(false);
    } else {
      callback(result);
    }
  });
};
```

实现的方式还是通过执行 MySQL 的 update 语句来实现。这里需要注意的是，多个问号如何传递参数，以上代码中涉及两个问号，那么我们必须把第二个参数作为一个含有两个 json 元素的数组。因此这里的参数为[rowInfo, idJson]，总之有多少个问号，第二个参数数组就必须有多少个元素（如果一个问号，只需要将第二个参数设为 json），并且需要对应第一个问号代替第一个数组元素，依次类推。测试代码如下：

```
var BaseModel = require('./base_model')
, BaseModel = new BaseModel()
, rowInfo = {}
, tableName = 'node_book';

/* modify 验证 */
var newInfo = {};
newInfo.book_name = 'nodejs book-by danhuang';
newInfo.author = 'Jimi';
var idJson = {'book_id': 2};
BaseModel.modify(tableName, idJson, newInfo, function(ret){
  console.log(ret);
});
```

执行测试代码，执行返回如图 7-6 所示的结果，数据库如图 7-7 所示。

```
E:\Desktop\code\chapter_seven\7.2>node index.js
connection local mysql success
{ fieldCount: 0,
  affectedRows: 1,
  insertId: 0,
  serverStatus: 2,
  warningCount: 0,
  message: '(<Rows matched: 1 Changed: 1 Warnings: 0',
  changedRows: 1 }
```

图 7-6 MySQL 数据修改返回结果

← T →				book_id	book_name	author	time				
<input type="checkbox"/>		编辑	快速编辑		复制		删除	1	nodejs book	danhuang	2012-12-22 23:47:24
<input type="checkbox"/>		编辑	快速编辑		复制		删除	2	nodejs book-by danhuang	Jimi	2012-12-23 12:47:31

图 7-7 数据修改后 MySQL 数据库查询结果

从图 7-7 中可以看到,已经成功地将 book_id 为 2 的数据更改,book_name 修改为 nodejs book-by danhuang, author 修改为 Jimi。

7.2.7 删除数据库记录

BaseModel 的 remove 方法实现,代码如下:

```
/**
 *
 * @desc 删除数据库的一条数据
 * @param tableName string
 * @param idJson json
 * @param rowInfo json
 * @param callback function
 * @return null
 */
this.remove = function(tableName, idJson, callback){
    dbClient.query('delete from ' + tableName + ' where ?', idJson,
        function(error, results) { //回调获取数据删除结果
            if(error) {
                console.log("ClientReady Error: " + error.message);
                dbClient.end();
                callback(false);
            } else {
                callback(true);
            }
        });
};
```

remove 方法实现和 modify、insert 方法都有相似之处,主要是修改了一下执行的 MySQL 语句。在 index.js 中添加如下测试代码:

```
var BaseModel = require('./base_model')
, baseModel = new BaseModel()
, rowInfo = {}
, tableName = 'node_book';

/* remove 验证 */
var idJson = {'book_id': 2};
baseModel.remove(tableName, idJson, function(ret){
    console.log(ret);
});
```

执行 index.js, 执行返回结果如图 7-8 所示, 如图 7-9 所示为已经被删除后的数据库记录。

```
E:\Desktop\code\chapter_seven\7.2>node index.js
connection local mysql success
true
```

图 7-8 MySQL 执行删除返回结果

	book_id	book_name	author	time
1	nodejs	book	danhuang	2012-12-22 23:47:24

图 7-9 MySQL 数据库查询结果

从图 7-9 中可以看到, book_id 为 2 的记录已经被删除。

7.2.8 数据条件查询

接下来我们要实现其中最复杂的方法 find, 其中包含了 MySQL 的排序、返回数量控制和返回字段选择等。首先看一下本函数的所有字段类型, 代码如下:

```
/**
 *
 * @desc 条件查询数据
 * @param tableName string
 * @param whereJson json desc (and 和 or 区别, 其中的条件为 key 值、连接符大于
 小于还是等于 value 值)
 * @param orderByJson json desc ({'key': 'time', 'type': 'desc'})
 * @param limitArr array desc (第一个元素是返回偏移量, 第二个元素是返回数量, 如果
 为空, 则返回全部)
 * @param fieldsArr array desc (返回哪些字段)
 * @param callback function
 * @return null
 */
```

其中的 whereJson 较为复杂, 此处只包含两种条件查询, and 和 or。其中该 whereJson 的定义方法可以参照下面的方式, 代码如下:

```
var whereJson = {
  'and': [{'key': 'book_name', 'opts': '=', 'value': '"nodejs"'},
    {'key': 'author', 'opts': '=', 'value': '"danhuang"' }],
  'or': [{'key': 'book_id', 'opts': '<', 'value': 10}]
};
```

【代码说明】

- ❑ [{"key": "book_id", "opts": "<", "value": 10}]: key 为条件键名, opts 为条件, value 为对比的值。

orderByJson 只有两个字段, 分别为排序键名和排序方法, 包括 desc 和 asc。limitArr 是一个包含两个元素的数组, 其中第一个元素是偏移量, 第二个是返回数量。fieldsArr 保存查询后需要返回所有字段的数组。接下来看一下如何组织执行的 MySQL 语句, 代码如下:

```
var andWhere = whereJson['and']
```



```

    , orWhere    = whereJson['or']
    , andArr = []
    , orArr  = [];
    /* 将数组转换为 where and 条件 array */
    for(var i=0; i<andWhere.length; i++){
        andArr.push(andWhere[i]['key']+andWhere[i]['opts']+andWhere
[i]['value']);
    }
    /* 将数组转换为 where or 条件 array */
    for(var i=0; i<orWhere.length; i++){
        orArr.push(orWhere[i]['key'] + orWhere[i]['opts'] +orWhere[i]
['value']);
    }
    /* 判断条件是否存在, 如果存在则转换相应的添加语句 */
    var filedStr = fieldsArr.length>0 ? fieldsArr.join(',') : '*'
    , andStr    = andArr.length>0    ? andArr.join(' and ') : ''
    , orStr     = orArr.length>0     ? ' or '+orArr.join(' or ') : ''
    , limitStr  = limitArr.length>0 ? 'limit'+limitArr.join(',') : ''
    , orderStr  = orderByJson ? ' order by ' + orderByJson['key'] +
' + orderByJson['type'] : '';
    var sql = 'SELECT ' + filedStr + ' FROM ' + tableName +'where '
+ andStr + orStr + orderStr + limitStr;

```

【代码说明】

- ❑ andWhere = whereJson['and']: 获取条件 and 数组。
- ❑ orWhere = whereJson['or']: 获取条件 or 数组。
- ❑ filedStr = fieldsArr ? fieldsArr.join(',') : '*': 获取 select fields, 如果参数为空则返回所有字段。
- ❑ andStr = andArr ? andArr.join(' and ') : '': 使用 and 连接所有 and 条件语句。
- ❑ orStr = orArr ? ' or '+orArr.join(' or ') : '': 使用 or 连接所有 or 条件语句。
- ❑ limitStr = ' limit ' + limitArr.join(','): 根据 limitArr 设置偏移量以及返回数量。
- ❑ orderStr: 设置排序字段和排序方式。
- ❑ sql: 需要执行的 sql 查询语句。

这段代码需要大家注意的是, 在将数组转化为 sql 语句字符串时, 添加一个判断语句, 确认是否需要该类型条件查询, 例如 limitArr.length>0 ? 'limit '+limitArr.join(',') : '', 如果 limitArr 存在, 并且不为空的时候, 就返回 limit 条件语句, 否则返回一个空字符串, 如果不加这层判断会导致 sql 语句出错。最后就是使用 dbClient.query 来执行 MySQL 语句, 其用法和前面几个函数的实现方法一样, 代码如下:

```

dbClient.query('SELECT ' + filedStr + ' FROM ' + tableName + ' where ' +
andStr + orStr + orderStr + limitStr,
    function(error, results) {
        if (error) {
            console.log('GetData Error: ' + error.message);
            dbClient.end();
            callback(false);
        } else {
            callback(results);
        }
    });

```

最后我们为 find 函数接口添加一个测试代码, 代码如下:


```

var BaseModel = require('./base_model')
, BaseModel = new BaseModel();

/* find 验证 */
var whereJson = {
  'and' : [{ 'key': 'book_name', 'opts': '=', 'value' : '"nodejs"',
    { 'key': 'author', 'opts': '=', 'value' : '"danhuang"' } ],
  'or' : [{ 'key': 'book_id', 'opts': '<', 'value' : 10 } ]
};
/* 返回字段数组 */
var fieldsArr = ['book_name', 'author', 'time'];
/* 排序对象 */
var orderByJson = { 'key': 'time', 'type': 'desc' };
/* 数据偏移, 以及数据返回数量 */
var limitArr = [0, 10];
baseModel.find(tableName, whereJson, orderByJson, limitArr, fieldsArr,
function(ret) {
  console.log(ret);
});

```

在参数填写时一定要注意传入的参数类型, 这几个参数的设置方法在开始已经介绍过, 这里就不再详细描述了。这里着重说明 whereJson 中如果 key 值是一个字符串, 那么必须给 value 值 "nodejs" 和 "danhuang" 添加一个双引号, 如果是字符串或者其他类型则不需要。接下来我们运行 index.js 脚本, 如图 7-10 所示为运行日志返回结果。

```

E:\Desktop\code\chapter_seven\7.2>node index.js
connection local mysql success
[ < book_name: 'nodejs book',
  author: 'danhuang',
  time: Sat Dec 22 2012 23:47:24 GMT+0800 (中国标准时间) > ]

```

图 7-10 多条件查询结果返回信息

这里大家可以看到, 返回的数据是一个 array 数组, 数组中为每条 json 数据, 因为我们只 select 三个字段不包含主键 id, 所以这里没有返回 book_id 字段数据。

至此我们就实现了 Node.js 中 MySQL 的操作基类, 这个基类可以作为所有 Model 层的父类, 也可以单独作为一个项目中所有 Model 层操作类。使用方法可以参考本书源码中 7.2 节的 index.js, 其中包含了所有函数接口的使用方法。需要特别留意的是, 本基类没有进行 MySQL 执行语句的转义处理, 在真正的项目应用时, 需要应用 Node.js 的 MySQL 模块提供的 mysql.escape 和 mysql.escapeId, 避免 MySQL 注入。

7.3 Node.js 与 MongoDB

7.2 节是从 MySQL 的配置安装开始, 介绍到 Node.js 的 MySQL 应用。本节同样是从 MongoDB 的配置安装介绍开始到 Node.js 的 MongoDB 应用, 其中同样会引导大家实现一个 MongoDB 操作基类。学习本节和学习上一节的思想是相同的, 希望大家多实践多应用。本节的内容包括 MongoDB 的配置安装、Node.js 的 MongoDB 操作基类实现和 MongoDB 操作基类的简单应用。

7.3.1 MongoDB 的安装以及工具介绍

MongoDB Windows 和 Linux 下 MongoDB 都无需安装编译, 直接前往官网¹下载最新版本的可执行文件即可, 需要注意的是, 下载的时候注意自己的系统是多少位的, 下载完成后将文件夹压缩包解压到任何盘 (笔者解压到 D 盘的 MongoDB 下)。MongoDB 是基于文件的读写存储, 因此需要指定文件夹来存放其文件数据, 这里笔者在 Windows 下存储在 c:\data\db 下, 因此需要在 C 盘下创建 data 和 db 文件夹。

运行 cmd 进入 Windows 的命令行窗口, 然后进入 MongoDB 解压文件夹 D 盘中的 MongoDB 下的 bin 目录, 依次执行如下指令:

```
d:
cd d:\MongoDB\mongodb-win32\bin
dir
```

如果放在 E 盘时, 执行:

```
e:
```

进入 E 盘, 执行完 dir 后可以看到如图 7-11 所示的目录结构。

```
d:\mongodb\mongodb-win32\bin 的目录
2012/12/23  20:11    <DIR>          .
2012/12/23  20:11    <DIR>          ..
2012/11/27  05:00           6,714,880  bsondump.exe
2012/11/27  03:49           3,440,640  mongo.exe
2012/11/27  04:10           6,767,104  mongod.exe
2012/11/27  04:10          68,054,016  mongod.pdb
2012/11/27  04:21           6,747,648  mongodump.exe
2012/11/27  04:31           6,717,952  mongoexport.exe
2012/11/27  04:55           6,730,240  mongofiles.exe
2012/11/27  04:36           6,735,360  mongoimport.exe
2012/11/27  04:50           6,716,416  mongooplog.exe
2012/11/27  05:05           6,710,784  mongoperf.exe
2012/11/27  04:26           6,733,824  mongorestore.exe
2012/11/27  04:16           4,850,176  mongos.exe
2012/11/27  04:16          52,612,096  mongos.pdb
2012/11/27  04:40           6,745,088  mongostat.exe
2012/11/27  04:45           6,717,440  mongotop.exe
```

图 7-11 MongoDB 工具

下面就 MongoDB 提供的所有工具做一个简单的介绍。

bsondump 工具主要是将 bson 类型二进制数据转化为可读的 JSON 数据, 该工具可以很好地将 mongodump 导出的数据转化为可读 json 数据。

mongo.exe 是用来启动 MongoDB shell 的, 即 MongoDB 客户端, 相当于一个连接 MongoDB 的客户端, 例如 Node.js 也是一个 MongoDB 数据的客户端。

mongod.exe 是用来连接到 mongo 数据库服务器的, 即服务器端。所有的客户端都需要连接该服务器, 因此对于 MongoDB 来说必须启动该进程, 而 mongo 可以不用启动, 因为我们可以通过其他的客户端连接服务器 mongod。

1 参见网站 <http://www.MongoDB.org/downloads> MongoDB 官网。

mongodump 是一个数据库二进制数据导出备份工具，其相反的导入工具为 **mongoexport**，可以应用这套工具导入导出数据库二进制文件信息。

mongoexport 同样是一个数据库导出工具，其导出的是一个 json 格式数据，会将每个表作为一个 JSON 文件存储，导出到相应的文件夹。

mongofiles 工具主要应用于文件的存储，MongoDB 应用 GridFS¹ 规范来存储大型文件信息，例如视频图片等。该工具可以使用如下方式进行文件的存储：

```
mongofiles.exe -host 127.0.0.1:27017 -d mydb put 文件名
```

通过如下指令可以查看当前所有的存储文件列表。

```
db.fs.files.find()
```

如果需要大型文件存储数据库的话，可以去官网文档进行学习了解，地址是 <http://cn.docs.MongoDB.org/manual/reference/mongofiles/>。

mongoimport 工具和 **mongoexport** 是相对应的，一个为数据导出，一个为数据导入。

mongooplog 工具是将远程的一个 MongoDB 数据导入到本地或者其他服务器，其与 **mongodump** 和 **mongoexport** 最大的区别在于该工具在迁移过程中一直保持运行中。如果不是需要保证在 MongoDB 数据库迁移过程中，保持运行状态的话，最好建议使用 **mongodump** 和 **mongoexport** 工具，执行指令方式如下：

```
mongooplog --from mongodb0.example.net --host mongodb1.example.net
```

mongostat 实用工具提供了一个当前正在运行的 **mongod** 或 **mongos** 的实例的状态的快速概览。**mongostat** 在功能上类似于 UNIX / Linux 的文件系统工具 **vmstat** 的，但提供数据是关于 **mongod** 和 **mongos** 的实例。

mongotop 提供了一个方法来跟踪一个 MongoDB 的实例花费在读取和写入数据的时间，默认情况下，**mongotop** 的值每一秒刷新一次。

如果需要了解更多的关于 MongoDB 的一些工具的介绍，以及工具的使用方法可以去官网中查看了解，官网文档地址为 <http://cn.docs.mongodb.org/manual/reference/components/>。

本书涉及的工具只有 **mongod**、**mongoimport** 和 **mongoexport**，其中的 **mongod** 为 MongoDB 服务器的启动程序。Windows 下和 Linux 下都有一套相似的工具，其使用方法，以及所需参数都大致相同。

7.3.2 mongod 的启动运行方法

7.3.1 节中有介绍我们需要创建一个文件夹来存储 MongoDB 的文件数据。因此首先创建 C 盘的 data\db 作为 MongoDB 的文件数据存储目录，进入 MongoDB 目录，运行如下指令启动 MongoDB，在 Windows 下依次执行如下指令：

```
d:
cd d:\mongodb\mongodb-win32\bin
```

1 GridFS 是一种将大型文件存储在 MongoDB 的文件规范。所有官方支持的驱动均实现了 GridFS 规范。缘起实现语言支持命令行工具参见。

```
mongod.exe --dbpath c:\data\db
```

Linux 下可以直接根据 mongod 可执行文件的目录来运行该脚本。例如如下指令方式（假设 MongoDB 解压到/usr/local/mongodb/文件夹下）：

```
/usr/local/mongodb/bin/mongod --dbpath /home/data/db/
```

这样是启动了 mongod，但是如何应用守护进程方式来运行 mongod 呢？

如表 7.4 所示是一份 MongoDB 的执行--help 返回的关于 mongod 使用的一些参数说明，根据这份文档我们来讲解如何设置 mongod 的数据库运行参数。执行方法一栏中的 mongod 指令是/usr/local/mongodb/bin/mongod 的简写。

表 7.4 MongoDB 命令行执行参数

参 数	描 述	执行方法（举例）
-h [--help]	显示 mongod 的命令帮助信息	mongod -h 获取 mongod --help
--version	查看当前 mongod 的版本信息	mongod --version
-f	配置文件中设置一些 mongod 启动的配置，关联该配置运行 mongod 服务器	mongod -f /usr/local/mongod.conf
-v	增加了大量的内部报告返回标准输出或指定的日志文件-日志路径。使用 V 形控制的详细程度，包括选项多次	mongod -v mongod -vv mongod -vvv mongod -vvvv
--quiet	安静输出	mongod --quiet
--port	指定 mongod 运行端口，默认是 27017	mongod --port 27018
--bind_ip	绑定服务器运行 IP 地址，默认是本地 127.0.0.1	mongod --bind_ip 127.0.0.1
--maxConns	服务器最大客户端连接数，默认是 2 万	mongod --maxConns 200
--logpath	指定 MongoDB 日志文件，注意是指定文件不是目录	mongod --logpath /data/mongodb/ mongodb_log.log
--logappend	指定方式，使用追加的方式写日志	mongod --logappend --logpath /home/data/mogodb_log.log
--pidfilepath	PID File 的完整路径，如果没有设置，则没有 PID 文件	mongod --pidfilepath /data/mongodb/mongod.pid
--fork	以守护进程的方式运行 MongoDB，创建服务器进程	mongod --fork
--syslog		
--auth	启用验证	mongod --auth
--cpu	定期显示 CPU 的利用率和 io wait	mongod --cpu
--dbpath	指定数据库路径，默认数据路径在/data/db/，Windows 下默认是在 c:/data/db/下	mongod --dbpath /data/db/
--diaglog	diaglog 选项 0=off 1=W 2=R 3=both 7=W+some reads	mongod --diaglog 1
--directoryperdb	设置每个数据库将被保存在一个单独的目录	mongod --directoryperdb

续表

参 数	描 述	执行方法（举例）
--ipv6	启用 IPv6 选项	mongod --ipv6
--jsonp	允许 JSONP 形式通过 HTTP 访问（有安全影响）	mongod --jsonp
--noauth	不启用验证	mongod --noauth
--nohttpinterface	关闭 http 接口，默认关闭 27018 端口访问	mongod --nohttpinterface
--nounixsocket	禁用 Unix 套接字监听	mongod --nounixsocket
--journal	启用日志选项，MongoDB 的数据操作将会写入到 journal 文件夹的文件里	mongod --journal
--repair	修复所有数据库 run repair on all dbs	mongod --repair
--rest	开启简单的 rest API	mongod --rest
--sysinfo	打印一些诊断系统信息	mongod --sysinfo
--shutdown	关闭正在运行的服务器	mongod --shutdown
--upgrade	如果需要升级数据库	mongod --upgrade
--fastsync	从一个 dbpath 里启用从库复制服务，该 dbpath 的数据库是主库的快照，可用于快速启用同步	mongod --fastsync
--autoresync	如果从库与主库同步数据差得多，自动重新同步	mongod --autoresync
--oplogSize	设置 oplog 的大小（MB）	mongod --oplogSize 2M

上述参数都可以写入 mongod.conf 配置文档里，配置文档示例如下所示，下方的参数可以在表中进行查找。

```
dbpath = /data/mongodb
logpath = /data/mongodb/mongodb.log
logappend = true
port = 27017
fork = true
auth = false
smallfiles = true
```

根据上面的配置文件内容方式，我们就简单地配置一个配置文件在 /usr/local/mongodb/conf 下，命名为 mongodb.conf，相应配置信息如上代码所示。如果不使用 smallfiles 参数时，启动 MongoDB 时会报一个错误提示：Please make at least 3379MB available in /data/db/journal or use --smallfiles，该错误提示说明在 /data/db/journal 文件目录，必须要保证有 3379MB 空间以上。由于我使用的是虚拟机空间，比较小，因此启动会出现异常。按照系统提示，我们只需要在配置中添加一个 smallfiles 参数即可，如果是在项目开发过程中，建议不要打开该字段。

7.3.3 MongoDB 的启动运行

接下来我们使用该配置文件运行 MongoDB，运行方式如下指令（以下皆为 Linux 下运行方法）：

```
/usr/local/mongodb/bin/mongod -f /usr/local/mongodb/conf/mongodb.conf
```

或者：

```
/usr/local/mongodb/bin/mongod --config /usr/local/mongodb/conf/mongodb.conf
```

启动成功后，按 Ctrl+C 快捷键退出启动程序，然后输入 `ps -ef | grep MongoDB` 指令查看是否成功启动。如果成功运行，可以看到如图 7-12 所示是返回结果。



图 7-12 ps 查询 MongoDB 启动进程

运行成功后，我们就可以使用 `mongo` 工具连接该 MongoDB 服务器，执行如下指令，运行 `mongo` 工具，如果在 Windows 下要进入 MongoDB 安装盘，然后运行 `mongo` 工具即可。

Linux 下指令：`/usr/local/mongodb/bin/mongo`

Windows 下方法如下：

Windows 下指令：
d:
\\mongodb\\mongodb-win32\\bin\\mongo

进入 `mongo` 操作窗口后，就可以使用一些基本 MongoDB 指令来对数据库进行增、删、改、查了。MongoDB 相对 MySQL 有一套对应表格，可以让用户很好地从 MySQL 数据库适应到 MongoDB 数据库开发。基本的对应关系如表 7.5 所示。

表 7.5 MySQL 于 MongoDB 操作命令对比表

MySQL	MongoDB	操 作 功 能
show databases	show dbs	查看数据库列表
use database	use database	选择需要操作使用的数据库名
show tables	show collections	查看数据库中所有的表
create table user	db.createCollection('user')	创建表结构 (需要注意的是 MySQL 需要添加 table 的字段信息，而对于 MongoDB 来说是无需添加)
create index idx_user_id_asc on users(user_id)	db.users.ensureIndex({ user_id: 1 })	添加表的字段索引，如果需要设置为唯一字段时，只需要修改为 db.users.ensureIndex({ user_id: 1 }, {unique:true})
insert into users	db.users.insert	db.users.insert({'name':'danhuang'})
select * from users	db.users.find()	db.users.find({ }, { user_id: 1, status: 1 }) 第一个参数为查询的条件，第二个参数为需要 select 的字段，值为 1 时为 select 字段

续表

MySQL	MongoDB	操 作 功 能
select count(user_id) from users	db.users.count({ user_id: { \$exists: true } })	参数为查询条件
select distinct(status) from users	db.users.distinct("status")	获取不重复记录
select * from users limit 1	db.users.find().limit(1)	设置记录条数
delete from users	db.users.remove()	参数为删除的条件，不添加参数，默认删除表的所有数据

对于数据库的一些操作指令还可以通过执行 `db.help()` 和 `db.user.help()` 查看，如图 7-13 所示是执行了 `db.help()` 返回的结果，如图 7-14 所示是执行 `db.local.help()` 返回的结果。在 MongoDB 的 mongo 客户端中可以使用 `tab` 键来提示指令。

```
root@ubuntu:/usr/local/mongodb/bin# ./mongo
MongoDB shell version: 2.2.2
connecting to: test
> db.help()
DB methods:
  db.addUser(username, password[, readonly=false]) - adds a new user
  db.adminCommand(nameOrDocument) - switches to
  db.auth(username, password)
  db.cloneDatabase(fromhost)
  db.commandHelp(name) returns the help for the
  db.copyDatabase(fromdb, todb, fromhost)
  db.createCollection(name, { size : ..., capped
  db.currentop() displays currently executing op
  db.dropDatabase()
  db.eval(func, args) run code server-side
  db.fsyncLock() flush data to disk and lock serv
  db.fsyncUnlock() unlocks server following a db
  db.getCollection(cname) same as db['cname'] or
  db.getCollectionNames()
```

图 7-13 MongoDB 数据库操作命令

```
> db.local.help();
DBCollection help
  db.local.find().help() - show DBCursor help
  db.local.count()
  db.local.copyTo(newColl) - duplicates collec
  db.local.convertToCapped(maxBytes) - calls {
  db.local.dataSize()
  db.local.distinct( key ) - e.g. db.local.dis
  db.local.drop() drop the collection
  db.local.dropIndex(index) - e.g. db.local.dr
  db.local.dropIndexes()
  db.local.ensureIndex(keypattern[,options]) -
  db.local.reIndex()
  db.local.find([query],[fields]) - query is a
```

图 7-14 MongoDB 数据库表操作命令

综上所述我们介绍了 MongoDB 的配置运行和管理，同时介绍了 mongo 客户端管理数据库 MongoDB 数据库的一些指令。接下来将会介绍 Node.js 如何使用 MongoDB 的 Node.js 驱动来连接操作存储数据到 MongoDB 中。

7.3.4 MongoDB 数据库接口设计

Node.js 提供了多种 MongoDB 的驱动，本书介绍的 MongoDB 驱动是 `node MongoDB`

native¹，npm 模块的下载方式是在项目目录执行 `npm install MongoDB`，成功下载后 MongoDB 会自动编译安装。

类似于 MySQL 数据接口设计，我们将设计一个 Node.js 的 MongoDB 操作的基类，该基类包括数据库连接、数据库增、删、改、查。初步设计该类为 `BaseMongoDB`。其方法如下，类同 MySQL 的 `BaseModel` 设计，如表 7.6 所示。

表 7.6 MongoDB基类API设计

方 法 名	参 数	参 数 说 明	返 回 值	描 述
constructor	无	无	无	
insert	tableName	string 表名	回调函数返回，成功返回 id，失败返回 false	
	rowInfo	array 插入数据		
modify	tableName	string 表名	回调函数返回，成功返回 true，失败返回 false	
	id	int 主键值		
	rowInfo	array 插入数据		
remove	tableName	string 表名	回调函数返回，成功返回 true，失败返回 false	
	id	int 主键值		
findOneById	tableName	string 表名	回调函数返回，成功返回查询的数据，失败返回 false	
	id	int 主键值		
find	whereJson	Json 查询条件：{'id':'asdasdas111', 'name':'danhuang'}，如果需要使用 or 或者 and 可以查询 MongoDB 的语法，将 whereJson 设置类似。	回调函数返回，成功返回查询的数据，失败返回 false	
	orderByJson	Json 排序：{'nook_book':1}，1 为正序，-1 为倒序		
	limitJson	Json 数量：{'num':10,skip:5}，10 为一次返回的数据条数，skip 相当于偏移量。		
	fieldsJson	Json 获取字段：{'node_book':1,'author':-1}，1 为返回，-1 为不返回。		
filterSelfRow	rowInfo	Json 查询返回数据	Json	对 MongoDB 中自带 objectId 进行转化

根据上表 `BaseMongoDB` 类的设计，我们可以将基类的代码架构设计如下：

```
/**
```

1 这是一个 MongoDB 的 Node.js 的驱动程序。来自 Ruby 的 MongoDB 驱动程序的一个部分。参考网站为 <https://github.com/christkv/node-MongoDB-native>。

```

* @type class Basemongodb
* @author danhuang
* @time 2012-12-26
* @desc desc base_mongodb.js
*/
var Util = require('./util')
    , mongodb = require('mongodb')
    , dbClient;

module.exports = function(){
  __constructor();

  /* MongoDB 单条数据查询 */
  this.findOneById = function(tableName, idJson, callback){
  };

  /* MongoDB 数据插入 */
  this.insert = function(tableName, rowInfo, callback){
  };

  /* MongoDB 数据修改 */
  this.modify = function(tableName, idJson, rowInfo, callback){
  };

  /* MongoDB 数据删除 */
  this.remove = function(tableName, idJson, callback){
  };

  /* MongoDB 条件查询 */
  this.find = function(tableName, whereJson, orderByJson, limitArr,
    fieldsArr, callback){
  };

  /* 数据转化 */
  this.filterSelfRow= function(rowInfo){
  };

  /* 构造函数 */
  function __constructor(){
  }
}

```

这些方法都类似于 MySQL 的操作基类。这相比之前的 MySQL 操作多了一个 `filterSelfRow` 函数，该函数的作用是将 MongoDB 自带主键 `ObjectId` 进行替换为键值为 `id` 的字符串。

接下来主要看一下每个方法的主要实现逻辑，通过 MongoDB 实现逻辑与 MySQL 的对比，希望大家更进一步地了解 Node.js 操作数据的一些实现原理。

创建 MongoDB 的一个数据连接的配置文件。配置文件如下：

```

{
  "db" : {
    "host" : "127.0.0.1",
    "port" : 27017,
    "user" : "",
    "password" : "",
    "db_name" : "node_test"
  }
}

```

```

}

```

host 为连接服务器, port 为连接端口 (使用 number 类型, 不可使用字符串类型), user 为连接用户名, password 为连接数据库密码, dbName 为连接的数据名。既然涉及配置文件就需要一个 util 工具来读取配置文件信息, 该模块和 MySQL 中的 util 工具模块是同一个方法。

下面代码是使用 MongoDB 驱动连接 MongoDB 数据库的构造函数:

```

/**
 *
 * 数据库连接构造函数
 */
function connection(callback){
    if(!db){
        var dbConfig = Util.get('config.json', 'db');
        /* 获取 MySQL 配置信息 */
        var host = dbConfig['host'] //MongoDB 数据库服务器
        , port = dbConfig['port'] //MongoDB 服务器端口
        , dbName = dbConfig['db_name'] //MongoDB 数据库名

        /* 创建 MongoDB 服务器对象 */
        , server = new mongodb.Server(host, port);

        /* 创建 MongoDB 客户端对象, 连接 MongoDB 服务器 */
        dbClient = new mongodb.Db(dbName, server, {safe:false});
        dbClient.open(function (err, dbObject) {
            db = dbObject;
            callback(dbObject);
            console.log('connection success');
        });
    } else {
        callback(db);
    }
}

```

【代码说明】

- ❑ new mongodb.Db(dbName, server, {safe:false}): 创建 MongoDB 对象。
- ❑ dbClient.open(function (err, dbObject)...: 创建 MongoDB 连接, 并通过回调函数返回 MongoDB 连接句柄。
- ❑ if(!db){...: 避免多次打开数据库连接。

MongoDB 的数据库连接和 MySQL 一样, 使用 Util.get 方法获取 MongoDB 数据库配置信息, 应用 MongoDB 的 NPM 模块创建 MongoDB 数据库连接对象, MongoDB 对象中包含 open 方法, 该方法创建数据库连接功能。注意, 这里避免数据库多次连接, 最好使用一个全局变量来存储该数据库连接句柄。

如果数据库中有使用到用户名和密码时, 需要应用 dbClient 的 authenticate 方法, 可以将以上代码修改如下:

```

/**
 *
 * 数据库连接构造函数
 */
function connection(callback){

```

```

if(!db){
    /* 获取 MySQL 配置信息 */
    var host = dbConfig['host'] //MongoDB 数据库服务器
        , port = dbConfig['port'] //MongoDB 服务器端口
        , dbName = dbConfig['db_name'] //MongoDB 数据库名

    /* 创建 MongoDB 服务器对象 */
    , server = new mongodb.Server(host, port);

    /* 创建 MongoDB 客户端对象, 连接 MongoDB 服务器 */
    dbClient = new mongodb.Db(dbName, server, {safe:false});

    dbClient.open(function (err, dbObject) {
        dbObject.authenticate(user , password, function(err,
        dbObject){
            db = dbObject;
            callback(dbObject);
            console.log('connection success');
        })
    });
} else {
    callback(db);
}
}

```

在配置文件中新增 user 和 password 值, 如果没有用户名和密码则设置为空。

```
dbObject.authenticate(user , password, function(err, dbObject)
```

代码是调用 authenticate 验证数据库用户名密码。这里需要做一个判断, 避免 dbClient.open 失败时, 执行数据库验证而导致异常, 可以添加条件判断, 代码如下:

```

dbClient.open(function (err, dbObject) {
    if(dbObject){
        dbObject.authenticate(user , password, function(err,
        dbObject){
            db = dbObject;
            callback(dbObject);
            console.log('connection success');
        });
    } else {
        console.log('err');
    }
});

```

7.3.5 数据插入

实现 BaseMongoDB 类中的 insert 方法, 同时在 index 脚本中调用 BaseMongoDB 对象的 insert 方法向数据库插入数据, insert 实现方法如下:

```

/**
 *
 * @desc 向数据库插入数据
 * @param tableName string
 * @param rowInfo json
 * @param callback function
 * @return null

```

```

    */
    this.insert = function(tableName, rowInfo, callback){
        connection(function(db){
            db.collection(tableName, function (err, collection) {
                collection.insert(rowInfo, function(err, objects){
                    if (err) {
                        callback(false);
                    } else {
                        callback(objects);
                    }
                });
            });
        });
    };
};

```

【代码说明】

- ❑ connection(function(db){...: 连接数据库，通过回调函数获取数据库连接句柄；
- ❑ db.collection(tableName, function (err, collection)...: 选择需要操作的数据库表名 tableName，回调函数获取 collection 表操作对象；
- ❑ collection.insert(rowInfo...: 执行数据库插入数据，回调函数返回插入后的插入结果，如果成功则返回插入后的 MongoDB 的 objectId。

在数据库插入出现 err 时，我们是直接返回 false，而在实际项目中希望大家在出现 err 时记录一个数据库插入失败日志。下面我们看一下数据库插入的实例，代码如下：

```

var BaseMongoDB = require('./base_mongodb')
, baseMongoDB = new BaseMongoDB()
, rowInfo = {}
, tableName = 'node_book';
rowInfo.book_name = 'nodejs book';
rowInfo.author = 'danhuang';

/*数据插入验证 */
rowInfo.book_name = 'nodejs book1';
baseMongoDB.insert(tableName, rowInfo, function(ret){
    console.log(ret);
});

```

【代码说明】

- ❑ BaseMongoDB=require('./base_mongodb'): 获取 BaseMongoDB 类；
- ❑ baseMongoDB = new BaseMongoDB(): 创建 BaseMongoDB 类的 baseMongoDB 对象；
- ❑ baseMongoDB.insert...: 调用 baseMongoDB 对象的 insert 方法向数据库插入数据。

调用方法和之前的 MySQL 插入调用方法类似，这里为了测试需要，添加多个测试插入用例，代码如下所示。

```

var rowInfoNextOne = {}
, rowInfoNextTwo = {}
, rowInfoNextThree = {};
/* 执行单条数据插入 */
rowInfoNextOne.book_name = 'nodejs book2';
rowInfoNextOne.author = 'danhuang';
baseMongoDB.insert(tableName, rowInfoNextOne, function(ret){
    console.log(ret);
});

```

```

/* 单条数据插入 */
rowInfoNextTwo.book_name = 'nodejs book3';
rowInfoNextTwo.author = 'danhuang';
baseMongoDB.insert(tableName, rowInfoNextTwo, function(ret){
    console.log(ret);
});
/* 单条数据插入 */
rowInfoNextThree.book_name = 'nodejs book34';
rowInfoNextThree.author = 'danhuang';
baseMongoDB.insert(tableName, rowInfoNextThree, function(ret){
    console.log(ret);
});

```

以上代码和插入实例代码是一致的，执行 index.js 后，可以看到如图 7-15 所示的返回结果。通过 db.node_book.find() 获取的数据库结果如图 7-16 所示。

```

root@ubuntu:/home/danhuang/mongodb_nodejs# node index.js
[ { book_name: 'nodejs book1',
  author: 'danhuang',
  _id: 50dc67d18f6711e106000001 } ]
connection success
[ { book_name: 'nodejs book2',
  author: 'danhuang',
  _id: 50dc67d18f6711e106000002 } ]
connection success
[ { book_name: 'nodejs book3',
  author: 'danhuang',
  _id: 50dc67d18f6711e106000003 } ]
connection success
[ { book_name: 'nodejs book34',
  author: 'danhuang',
  _id: 50dc67d18f6711e106000004 } ]
connection success

```

图 7-15 MongoDB 数据插入返回结果

```

> db.node_book.find();
{"book_name": "nodejs book1", "author": "danhuang", "_id": ObjectId("50dc67d18f6711e106000001")}
{"book_name": "nodejs book2", "author": "danhuang", "_id": ObjectId("50dc67d18f6711e106000002")}
{"book_name": "nodejs book3", "author": "danhuang", "_id": ObjectId("50dc67d18f6711e106000003")}
{"book_name": "nodejs book34", "author": "danhuang", "_id": ObjectId("50dc67d18f6711e106000004")}
{"book_name": "nodejs book4", "author": "danhuang", "_id": ObjectId("50dc67d18f6711e106000005")}

```

图 7-16 MongoDB 数据库查询结果

如图 7-16 所示，成功执行了 3 条数据的插入，并且返回了一个含有 ObjectId 的 Json 数组。图 7-16 是通过使用 mongo 工具，查询数据库 node_book 表返回的结果，可以看到数据库成功地插入了 5 条数据。

7.3.6 数据修改

实现 BaseMongoDB 类中的 modify 方法，同时在 index 脚本中调用 BaseMongoDB 对象的 modify 方法实现修改数据库记录。modify 实现方法代码如下：

```

/**
 *
 * @desc MongoDB 数据修改接口
 * @parameters tableName 是需要操作的表单名
 * @parameters id 是需要修改的数据库主键
 * @parameters rowInfo 是需要修改的数据

```



```

* @parameters callback 回调函数
*/
this.modify = function(tableName, id, rowInfo, callback){
    connection(function(db){
        db.collection(tableName, function (err, collection) {
            var mongoId = new mongodb.ObjectId(id);
            collection.update({'_id':mongoId},rowInfo,{safe:true},
            function(err){
                if (err) {
                    callback(false);
                } else {
                    callback(true);
                }
            });
        });
    });
};

```

【代码说明】

❑ `var mongoId = new MongoDB.ObjectId(id)`: 将 `id` 值转换为 MongoDB 中的 `objectId`。

这部分代码和 `insert` 方法是相似的, 只是 `insert` 中调用的是 `collection.insert`, 而 `modify` 调用的是 `collection.update`。由于 MongoDB 的 `id` 是一个 `ObjectId`, 因此我们需要将字符串 `id` 值转化为 `ObjectId`, 这时候就需要应用到该模块提供的 API `MongoDB.ObjectId(id)`, 将 `id` 值字符串转换为 MongoDB 中的 `ObjectId` 的方法。MongoDB 还提供了如下 API, 用来转换数据类型为 MongoDB 数据类型。

```

new mongo.Long(numberString)
new mongo.ObjectId(hexString)
new mongo.Timestamp() //the actual unique number is generated on insert.
new mongo.DBRef(collectionName, id, dbName)
new mongo.Binary(buffer) //takes a string or Buffer
new mongo.Code(code, [context])
new mongo.Symbol(string)
new mongo.MinKey()
new mongo.MaxKey()
new mongo.Double(number) //Force double storage

```

添加 `index` 脚本测试代码, 验证 `modify` 接口, 其中需要一个 `id` 值。根据之前查询的结果我们使用 `id` 为 `50db1e69d923dbfe06000001` 的数据 `{"book_name": "nodejs book1", "author": "danhuang", "_id": "objectId(50db1e69d923dbfe06000001)"}`, 代码如下:

⚠注意: 在源码的测试实例中使用的是 `50db1e69d923dbfe06000001` 的 `id` 值, 但读者在实践时需要更改该 `id` 值来测试, 否则会 `update` 失败。

```

var BaseMongoDB = require('./base_mongodb')
, baseMongoDB = new BaseMongoDB()
, rowInfo = {};
var newInfo = {};
newInfo.book_name = 'nodejs book-by danhuang';
newInfo.author = 'Jimi';
var id = '50db1e69d923dbfe06000001';
baseMongoDB.modify(tableName, id, newInfo, function(ret){
    console.log(ret);
});

```

将 `id` 为 `50db1e69d923dbfe06000001` 的数据中的 `book_name` 改为 `nodejs book-by`

danhuang, author 改为 Jimi, 运行 index.js 脚本。窗口运行结果如图 7-17 所示, 通过 mongo 工具查询的数据返回结果如图 7-18 所示。

```
root@ubuntu:/home/danhuang/mongodb_nodejs# node index.js
connection success
true
```

图 7-17 MongoDB 数据库修改执行结果

```
db.node_book.find():
{"book_name": "nodejs book2", "author": "danhuang", "_id": ObjectId("50dc67d18f6711e106000002")}
{"book_name": "nodejs book3", "author": "danhuang", "_id": ObjectId("50dc67d18f6711e106000003")}
{"book_name": "nodejs book34", "author": "danhuang", "_id": ObjectId("50dc67d18f6711e106000004")}
{"book_name": "nodejs book4", "author": "danhuang", "_id": ObjectId("50dc67d18f6711e106000005")}
{"_id": ObjectId("50dc67d18f6711e106000001"), "book_name": "nodejs book-by danhuang", "author": "Jimi"}
```

图 7-18 查询 MongoDB 数据返回结果

从图 7-18 中可以看到, 数据库连接成功, 并 update 成功后 console.log 打印出 true。如图 7-18 所示为 mongo 工具查询数据库返回的数据, 从中可以看到 id 为 50db1e69d923dbfe06000001 的数据已经被修改, 并且数据也是正常的, 说明我们成功实现了 modify 接口。大家可以看到数据是正常的, 但是存储数据的前后顺序有更改, 对存储的数据是没有影响的。

7.3.7 查询一条数据

实现 BaseMongoDB 类中的 findOneById 方法, 同时在 index 脚本中调用 BaseMongoDB 对象的 findOneById 方法实现修改数据库记录。findOneById 实现代码如下:

```
var self = this;
/**
 *
 * 根据主键 id 值查询数据库的一条记录
 * @param tableName string
 * @param id number
 * @param callback function
 * @return null
 */
this.findOneById = function(tableName, id, callback){
    connection(function(db){
        db.collection(tableName, function(err, collection){
            var mongoId = new mongodb.ObjectId(id);
            var cursor = collection.find({'_id':mongoId});
            cursor.toArray(function(err, docs){
                if(err){
                    callback(false);
                } else {
                    var row = {};
                    if(docs){
                        row = self.filterSelfRow(docs.shift());
                    }
                    callback(row);
                }
            });
            cursor.rewind();
        });
    });
};
```

```
});
};
```

【代码说明】

- ❑ `var self = this`: 存储调用该函数的对象;
- ❑ `var mongoId = new mongodb.ObjectId(id)`: 将 `id` 转化为 MongoDB 的 `objectId`;
- ❑ `var cursor = collection.find({'_id':mongoId})`: 创建查询 `cursor` 对象;
- ❑ `cursor.toArray(function(err, docs) {...}`: 将查询结果转换为 `array` 数组;
- ❑ `row = self.filterSelfRow(docs.shift())`: 调用 `filterSelfRow` 函数过滤对象中的 `_id`, 将其转换为 `id`。

该方法实现使用到了该基类中的 `filterSelfRow` 方法, 代码如下:

```
this.filterSelfRow = function(rowInfo){
    if(rowInfo['_id']){
        rowInfo['id'] = rowInfo['_id'];
        delete rowInfo['_id'];
    }
    return rowInfo;
};
```

`filterSelfRow` 方法的目的是将查询返回的 `Json` 数据的 `key` 值为 `_id` 的数据转化为 `key` 为 `id` 的数据, 同时删除 `key` 为 `_id` 的数据。

如果在其他函数域内, 需要调用 `BaseMongoDB` 基类的内部方法时, 请将 `self` 对象保存在函数域内。由于 `self` 在不同函数体内是不同的, 因此在 `BaseMongoDB` 函数体内需要应用变量将该 `self` 对象存储保护, 然后通过 `self.filterSelfRow` 来调用。如果在其他函数域内, 使用 `this.filterSelfRow` 函数, 是无法成功地调用的。比如说上面的代码, `filterSelfRow` 域是在 `cursor.toArray` 函数中被调用, 这里的 `this` 就不是调用基类 `BaseMongoDB` 的对象, 而是 `cursor` 这个对象, 对于 `cursor` 这个对象是没有 `filterSelfRow` 方法的, 因此这样会出现异常错误。如果希望使用基类的 `filterSelfRow` 方法, 可以在基类中将调用基类 `BaseMongoDB` 对象的 `this` 赋值给该类中的一个全局变量 `self`, 然后再使用 `self.filterSelfRow` 就可以成功调用了。在 `Node.js` 中必须要注意的一点是, `this` 对象是时刻在变动的。使用 `cursor` 对象时, 最好在查询结束后使用 `cursor.rewind()` 方法重置 `cursor` 对象。

因为使用 `cursor.toArray` 接口, 所以即使返回的数据只有一条, 其也是一个含有一条数据的数组, 这里需要使用 `docs.shift()` 获取该 `Json` 数据。

添加 `index` 脚本测试代码, 验证 `findOneById` 接口, 其中需要一个 `id` 值。根据之前查询的结果, 我们使用 `id` 为 `50db1e69d923dbfe06000001` 的数据, 其中这个数据是在 `modify` 之前的结果, 代码如下:

```
var BaseMongoDB = require('./base_mongodb')
, baseMongoDB = new BaseMongoDB();
var id = '50db1e69d923dbfe06000001';
baseMongoDB.findOneById(tableName, id, function(ret){
    console.log(ret);
});
```

执行 index.js 的代码，返回结果如图 7-19 所示。

```
root@ubuntu:/home/danhuang/mongodb_nodejs# node index.js
connection success
{ book_name: 'nodejs book1',
  author: 'danhuang',
  id: 50dc67d18f6711e106000001 }
```

图 7-19 单条数据查询返回结果

成功地获取了 id 为 50db1e69d923dbfe06000001 的数据。因为在数据返回的时候已经做了处理，所以这里得到的是一个 Json 数据，而不是一个数组。

7.3.8 删除数据

实现 BaseMongoDB 类中的 remove 方法，同时在 index 脚本中调用 BaseMongoDB 对象的 remove 方法实现修改数据库记录。remove 实现代码如下：

```
/**
 *
 * @desc 删除数据库的一条数据
 * @param tableName string
 * @param id number
 * @param rowInfo json
 * @param callback function
 * @return null
 */
this.remove = function(tableName, id, callback){
    connection(function(db){
        db.collection(tableName, function (err, collection) {
            var mongoId = new mongodb.ObjectID(id);
            collection.remove({'_id':mongoId}, function(err){
                if (err) {
                    callback(false);
                } else {
                    callback(true);
                }
            });
        });
    });
};
```

remove 的代码通过使用 collection.remove 接口实现数据删除，添加一个条件查询_id 为 mongoId 即可，成功通过回调函数返回 true，失败通过回调函数返回 false。在 index.js 脚本中添加测试代码，代码如下：

```
/* remove 验证 */
var id = '50db1e69d923dbfe06000001';
baseMongoDB.remove(tableName, id, function(ret){
    console.log(ret);
});
```

删除数据库中 ObjectId 为 50db1e69d923dbfe06000001 的数据，执行结果如图 7-20 所示。通过 mongo 工具查询数据库后，返回结果如图 7-21 所示，也可以看到该记录已经被

删除。

```
index.js [0.5] 0.1.1588 [2] 1
root@ubuntu:/home/danhuang/mongodb_nodejs# node index.js
true
connection success
```

图 7-20 单条数据删除接口返回结果

```
db.node_book.find();
{ 'book_name' : 'nodejs book2', 'author' : 'danhuang', '_id' : ObjectId('50dc67d18f6711e106000002') }
{ 'book_name' : 'nodejs book3', 'author' : 'danhuang', '_id' : ObjectId('50dc67d18f6711e106000003') }
{ 'book_name' : 'nodejs book34', 'author' : 'danhuang', '_id' : ObjectId('50dc67d18f6711e106000004') }
{ 'book_name' : 'nodejs book4', 'author' : 'danhuang', '_id' : ObjectId('50dc67d18f6711e106000005') }
```

图 7-21 查询 MongoDB 数据返回结果

图 7-20 中 console.log 打印出 true 表示删除成功。通过数据库查询结果如图 7-21 所示，_id 为 50db1e69d923dbfe06000001 的数据已经被成功删除。

7.3.9 查询数据

实现 BaseMongoDB 类中的 find 方法，同时在 index 脚本中调用 BaseMongoDB 对象的 find 方法实现修改数据库记录。find 实现方法比较复杂，首先我们看一下在 find 方法中对参数的处理，代码如下：

```
this.find = function(tableName, whereJson, orderByJson, limitJson,
fieldsJson, callback){
    if(whereJson['id']){
        whereJson['_id'] = new mongodb.ObjectId(whereJson['id']);
        delete whereJson['id'];
    }
}
```

判断 whereJson 条件 Json 中是否存在 id 查询字段，如果有，则将 id 的 key 值转换为 MongoDB 中的 ObjectId 的 key 值 _id，处理这些以后我们就可以实现整个逻辑，代码如下所示：

```
var retArr = [];
connection(function(db){ //数据库连接
    /* 数据表单操作 */
    db.collection(tableName, function (err, collection) {
        /* 数据库查询句柄 */
        var cursor = collection.find(whereJson, fieldsJson);
        if(orderByJson){ //判断是否需要排序
            cursor.sort(orderByJson);
        }
        if(limitJson){ //判断是否需要设置返回数目以及偏移量
            var skip = limitJson['skip'] ? limitJson['skip'] : 0;
            cursor.limit(limitJson['num']).skip(skip);
        }
        /* 获取查询结果并转化为数组 */
        cursor.toArray(function(err, docs) {
            if(err){
                callback(false);
            } else {
                if(docs){
```

```

        for(var i=0; i<docs.length; i++){
            row = self.filterSelfRow(docs[i]);
            retArr.push(row);
        }
        callback(retArr);
    }
    });
    cursor.rewind();
});
});

```

【代码说明】

- ❑ `var retArr = []`: 初始化数据返回对象;
- ❑ `var cursor = collection.find(whereJson, fieldsArr)`: 创建数据查询对象 `cursor`;
- ❑ `cursor.sort(orderByJson)`: 排序字段, 参数格式如 `{'node_book':1}`, 1 为正序, -1 为倒序;
- ❑ `cursor.limit(limitJson['num']).skip(skip)`: 返回记录条数, 以及 `skip` 条数设置;
- ❑ `cursor.toArray(function(err, docs)...`: 将查询数据通过回调函数使用数组返回;
- ❑ `row = self.filterSelfRow(docs[i])`: 将数据中的 `_id` 转化为 `id` 值。

该方法的每个参数设置方法已经在本节的开头 API 介绍中提及。添加 `index.js` 脚本中的测试代码, 来验证 `find` 接口, 测试代码如下:

```

/* find 验证 */
var whereJson = {'author':'danhuang'};
var fieldsJson = {'book_name':1, 'author':1};
var orderByJson = {'book_name':1};
var limitJson = {'num':10};
/* 执行数据查询 */
baseMongoDB.find(tableName, whereJson, orderByJson, limitJson, fieldsJson,
function(ret){
    console.log(ret);
});

```

执行 `index.js`, 窗口 `console.log` 打印返回结果如图 7-22 所示。

```

root@ubuntu:/home/danhuang/mongodb_nodejs# node index.js
connection success
[ { book_name: 'nodejs book2',
  author: 'danhuang',
  id: 50dc67d18f6711e106000002 },
  { book_name: 'nodejs book3',
  author: 'danhuang',
  id: 50dc67d18f6711e106000003 },
  { book_name: 'nodejs book34',
  author: 'danhuang',
  id: 50dc67d18f6711e106000004 },
  { book_name: 'nodejs book4',
  author: 'danhuang',
  id: 50dc67d18f6711e106000005 } ]

```

图 7-22 多条件查询返回结果

从图 7-22 中可以看到, 其排序使用的是 `book_name`, 因此结果从 `book_name` 的 `nodejs book2` 一直升序到 `nodejs book4`。为了再次验证该接口的正确性, 我们将 `book_name` 的排序方式改为降序, 把 `orderByJson` 修改为 `{'book_name':-1}`, 再次执行 `index.js` 代码, 返回

结果如图 7-23 所示。

```
root@ubuntu:/home/danhuang/mongodb_nodejs# node index.js
connection success
[ { book_name: 'nodejs book4',
  author: 'danhuang',
  id: 50dc67d18f6711e106000005 },
  { book_name: 'nodejs book34',
  author: 'danhuang',
  id: 50dc67d18f6711e106000004 },
  { book_name: 'nodejs book3',
  author: 'danhuang',
  id: 50dc67d18f6711e106000003 },
  { book_name: 'nodejs book2',
  author: 'danhuang',
  id: 50dc67d18f6711e106000002 } ]
```

图 7-23 多条数据查询返回结果

从图 7-23 中可以看到，成功地返回了使用 `book_name` 倒序排列的数据，这样我们就实现了 `BaseMongoDB` 基类的 `find` 方法。

读者在自己学习这部分内容的时候不要直接运行源码中 `index.js` 的 `modify`、`findOneById` 和 `remove` 例子，需要将源码中的 `id` 值替换。比如源码 `index.js` 中的 `id` 为 `50db1e69d923dbfe06000001` 的值，在实际插入数据库的时候 `objectId` 和源码是有区别的，最好先插入数据，插入成功后再拿一个 `ObjectId` 来测试其他接口。

到此为止我们已经将 `BaseMongoDB` 基类的所有方法都实现了。这部分接口已经提供了大部分 Node.js 操作 MongoDB 的接口，这些 API 在设计上还存在一些缺陷，例如配置文件不要每次连接的时候都去读取，可以做一些缓存处理等，希望读者能够在开发中发挥自己的想法来完善这个类。这个类的用途有两个方面，如果是在一个需要长期维护的项目中，建议将该类作为一个 `Model` 的基类，在 `Model` 层使用其他 `Model` 来继承该类，如图 7-24 所示。如果是一个小项目，建议直接将该类作为一个 `Model` 去操作所有的数据，而不需要强制性的实现一些 `Model` 去继承该 `BaseMongoDB` 基类。

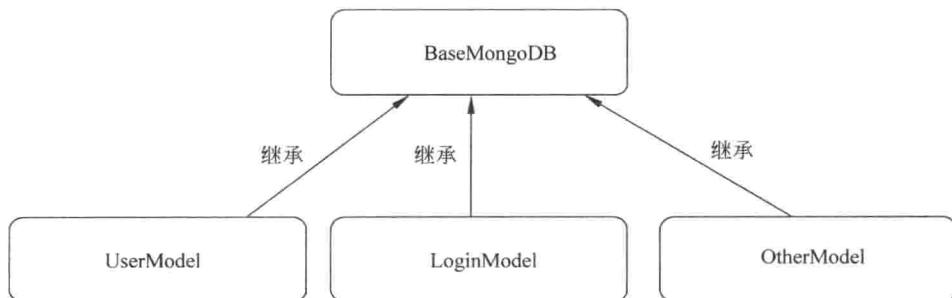


图 7-24 基类模块应用继承模式

7.4 MySQL 与 MongoDB 性能

前面两节介绍了 MySQL 和 MongoDB 的配置安装，并应用 NPM 的两个模块 MySQL

和 MongoDB 实现了两者的数据库操作基类，包括数据库的连接、数据插入、数据更新、数据删除和数据查询接口。本节为了比较两者的性能，会使用两者接口来对比 MySQL 和 MongoDB 在数据插入、数据查询、数据更新、数据删除的性能。本章使用的测试工具是 super-smack，关于该工具的介绍，以及使用方法会在 7.4.1 节中详细介绍。

7.4.1 测试工具及测试逻辑

我们选用 Siege 作为压力测试工具，Siege 是一个压力测试和评测工具，设计用于 Web 开发过程中，评估 Web 应用在压力下的承受能力，其可以根据配置对一个 Web 站点进行多用户的并发访问，记录每个用户所有请求过程的相应时间，并在一定数量的并发访问下重复进行。通过 Siege 工具我们模拟高并发请求时数据库查询和插入在 MongoDB 和 MySQL 之间的性能对比。

使用 Siege 做并发测试的时候，我们希望能看到测试结果反馈的折线图，那样就可以清晰明了地得到数据反馈。下面介绍一下 Siege 的安装配置和使用方法。

从 <http://www.joedog.org/> 上下载，然后编译，代码如下：

```
./configure --prefix=/usr/local/siege --mandir=/usr/local/man
make
make install
```

注意，在 configure 的时候，一定要设置 mandir 参数，否则当通过 `man siege` 查看 Siege 帮助的时候会看不到 manual。安装完成后，运行 bin 中的 `siege_config` 命令来创建 siege 文件之后，可以通过 `./siege -C` 命令来查看当前配置。

-cNUM：设置并发的用户（连接）数量。默认的连接数量可以到 `./siegerc` 中查看，指令为 `concurrent = x`。比如 `-c10`，表示设置并发 10 个连接。

-rNUM (repetitions)：重复数量，即每个连接发出的请求数量，设置这个，就不需要设置 -t 了。对应 `./siegerc` 配置文件中的 `reps = x` 指令。

-tNUM (time)：持续时间，即测试持续时间，在 NUM 时间后结束，单位默认为分。比如 `-t10`，表示测试时间为 10 分钟；`-t10s`，表示测试时间为 10 秒钟。对应 `./siegerc` 中的指令为 `time = x` 指令。

-b(benchmark)：基准测试，如果设置这个参数，那么 delay 时间为 0。

数据库性能测试只需要以上几个参数。现在需要通过在本章实现的 MySQL 和 MongoDB 基类来做数据插入和数据查询的 Web 服务器测试代码。

7.4.2 MySQL 性能测试代码

应用 MySQL 数据库的操作基类，以及 HTTP 模块实现一个数据库插入和数据库查询接口，利用 Siege 来设置高并发量来测试，从而来查询 MySQL 性能。下面是 MySQL 的测试代码，其中 BaseModel 为 7.2 节实现的 MySQL 操作基类。

```
var BaseModel = require('./base_model')
, BaseModel = new BaseModel();

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



```

/* 创建 HTTP 服务器 */
http.createServer(function(req, res) {
  var rowInfo = {}
  , tableName = 'node_book';
  rowInfo.book_name = 'nodejs book';
  rowInfo.author = 'danhuang';
  /* 执行数据插入 */
  baseModel.insert(tableName, rowInfo, function(ret){
    console.log(ret);
  });
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello World\n');
}).listen(1337);
console.log('Server running at http://127.0.0.1:1337/');

```

以上代码很简单地实现了一个服务器,该服务器调用 **BaseModel** 来插入一条记录数据,应用 **Siege** 来访问该 HTTP 服务器,通过设置高并发、高访问量来测试 MySQL 在数据插入上的性能。

7.4.3 MongoDB 性能测试代码

应用 MongoDB 数据库的操作基类,以及 HTTP 模块实现一个数据库插入和数据库查询接口,利用 **siege** 来设置高并发量来测试,从而来测试 MongoDB 的性能。下面是 MongoDB 的测试代码,其中 **BaseMongoDB** 为 7.3 节实现的 MySQL 操作基类。代码如下:

```

var BaseMongoDB = require('./base_mongodb')
, baseMongoDB = new BaseMongoDB();

var http = require('http');
/* 创建 HTTP 服务器 */
http.createServer(function(req, res) {
  var rowInfo = {}
  , tableName = 'node_book';
  rowInfo.book_name = 'nodejs book';
  rowInfo.author = 'danhuang';
  /* 执行数据插入 */
  baseMongoDB.insert(tableName, rowInfo, function(ret){
    console.log(ret);
  });
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello World\n');
}).listen(1200);
console.log('Server running at http://127.0.0.1:1200/');

```

代码实现和逻辑与 MySQL 的测试代码一致,这里主要是测试 MongoDB 的数据插入性能。接下来我们将介绍应用 **Siege** 来测试对比两者的性能。

7.4.4 性能测试数据分析

根据现有的测试机(测试机是 Linux 虚拟机,内存为 512MB,因此测试时间比较长,而且处理数据效率也会相应较低),我们需要设计一组测试数据,测试数据如表 7.7

所示。

表 7.7 压测数据设计表

每个连接请求数-r	并发用户数-c	插入数据条数	是否启用基准测试-b	备注
100	10	100*10=1000	是	
1000	10	10000*10=10000	是	
10000	10	10000*10=100000	是	
10000	100	10000*100=10 ⁶	是	

接下来我们只需要执行如下测试指令即可。

MySQL 测试指令：

```
siege -r10 -c100 http://127.0.0.1:1337-b
```

MongoDB 测试指令：

```
siege -r10 -c100 http://127.0.0.1:1200-b
```

100 个并发用户，每个用户发出 10 个插入数据的请求，接下来只需要将相应的 10 和 100 替换为表格中的-r 和-c 参数。

通过测试以后我们可以得到下面一组数据，如表 7.8 所示，是关于 MongoDB 和 MySQL 测试的一些数据。

表 7-8 MySQL压测结果图表

插入数据条数	实际插入条数	耗费总时间（秒）	Transaction rate (trans/sec)	Throughput	失败条数
100*10=1000	1000	8.97	111.48	0	0
1000*10=10000	10000	79.84	125.25	0	0
10000*10=100000	90751	2138.63	42.43	0	0
100000*10=10 ⁶	144608	8450.46	17.11	0	0

由于时间原因，这里没有测试 100 万以上的 MySQL 处理数据能力，这里只测试到 14 万数据的处理效率，测试结果如表 7.9 所示。

表 7-9 MongoDB压测结果图表

插入数据条数	实际插入条数	耗费总时间（秒）	Transaction rate (trans/sec)	Throughput	失败条数
100*10=1000	1000	8.76	114.16	0	0
1000*10=10000	10000	84.77	117.97	0	0
10000*10=100000	100000	784.48	127.47	0	0
100000*10=10 ⁶	1000000	8316.77	120.24	0	0

由于上述数据在 1 万~10 万之间，MySQL 的性能出现了很大的差距，因此这里我们添加了一组关于 MySQL 的测试数据，测试用例以及结果如表 7.10 所示。

表 7.10 细化压测数据表

插入数据条数	实际插入条数	耗费总时间（秒）	Transaction rate (trans/sec)	Throughput	失败条数
2000*10=20000	20000			0	0
4000*10=40000	40000			0	0

续表

插入数据条数	实际插入条数	耗费总时间（秒）	Transaction rate (trans/sec)	Throughput	失败条数
5000*10=50000	50000			0	0
6000*10=60000	60000			0	0
7000*10=70000	70000			0	0
8000*10=80000	80000			0	0
9000*10=90000	90000			0	0

由于虚拟机性能问题，这里没有给出相应的测试结果。从表格中可以看出，在数据量小于 100 000 时 MySQL 的性能和 MongoDB 的性能基本差不多，当大于 100 000 以后很明显可以看出 MongoDB 的性能远远的超出 MySQL。这里的 MySQL 配置是没有经过优化的，因此测试存在一定的局限性，同时测试机是 512MB 的虚拟机，也存在一定的问题。

7.5 本章实践

1. 根据 7.2 节介绍的 MySQL 知识点，设计一个 MySQL 学生数据库，学生数据库要设置超过两个表，例如 student_info 和 class。其中，student_info 表数据有如 student_id、name 和 age 等字段，确定另外表的字段以及每个字段类型，最后绘制一张数据设计表格。

分析：以上问题可以通过下面 3 个步骤分析。

(1) 使用 MySQL 操作数据库，可以应用 phpMyAdmin 工具进行创建。

(2) 应用 BaseModel 基类，向数据库插入数据、查询数据和删除数据等操作。

(3) 应用 BaseModel 基类以及前面介绍的 Node.js 的继承方法，实现多个 Model 层继承 BaseModel，并应用这些 Model 层实现增、删、改和查操作。

 注意：数据库的设计可以尽量简单，Node.js 的继承方法可以参考本书的 2.3 节。

表 7-11 数据表 student_info

字段名	字段类型	字段描述
student_id	int(11)	主键
name	varchar(100)	姓名
age	varchar(100)	年龄
class_id	int(11)	班级 id

表 7-12 数据表 class

字段名	字段类型	字段描述
class_id	int(11)	主键
name	varchar(100)	班级名

如表 7-11 和表 7-12 所示为本习题需要实现的 MySQL 数据库表单。应用 phpMyAdmin 创建 school 数据库，然后执行源码库中 school.sql 的 sql 语句。sql 代码如下：

```
-- phpMyAdmin SQL Dump
-- version 3.4.10.1
```

```
-- http://www.phpmyadmin.net
--
-- 主机: localhost
-- 生成日期: 2013 年 05 月 02 日 07:38
-- 服务器版本: 5.5.20
-- PHP 版本: 5.3.10

SET SQL_MODE="NO_AUTO_VALUE_ON_ZERO";
SET time_zone = "+00:00";

--
-- 数据库: 'school'
--
-----

--
-- 表的结构 'class'
--
CREATE TABLE IF NOT EXISTS 'class' (
  'class_id' int(11) NOT NULL AUTO_INCREMENT,
  'name' varchar(100) COLLATE utf8_bin NOT NULL,
  PRIMARY KEY ('class_id'),
  UNIQUE KEY 'class_id' ('class_id')
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_bin AUTO_INCREMENT=1 ;

-----

--
-- 表的结构 'student_info'
--
CREATE TABLE IF NOT EXISTS 'student_info' (
  'student_id' int(11) NOT NULL AUTO_INCREMENT,
  'name' varchar(100) COLLATE utf8_bin NOT NULL,
  'age' varchar(100) COLLATE utf8_bin NOT NULL,
  'class_id' int(11) NOT NULL,
  PRIMARY KEY ('student_id'),
  UNIQUE KEY 'student_id' ('student_id')
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_bin AUTO_INCREMENT=1 ;
```

通过运行如上 MySQL 的 sql 语句后,我们可以通过 phpMyAdmin 工具查看到如图 7-25 所示后执行结果。

表		操作		行数	类型	整理	大小	多余
<input type="checkbox"/>	class			0	InnoDB	utf8_bin	32.0 KB	-
<input type="checkbox"/>	student_info			0	InnoDB	utf8_bin	32.0 KB	-
2 张表		总计		0	InnoDB	utf8_bin	64.0 KB	0 字节

图 7-25 数据库查询结果

修改源码中 sql 数据库的配置文件 config.json，修改后代码如下：

```
{
  "db" : {
    "host" : "127.0.0.1",
```

```

    "port" : "3306",
    "user" : "root",
    "password" : "",
    "dbName" : "school"
  }
}

```

添加测试代码 index.js，代码如下：

```

/* index.js */
var BaseModel = require('./base_model')
,   BaseModel = new BaseModel()
,   studentInfo = {}
,   classInfo = {}
,   studentName = 'student_info'
,   className = 'class';
studentInfo.name = 'danhuang';
studentInfo.age = '22';
studentInfo.class_id = '1';

classInfo.class_id = '1';
classInfo.name = '三年级二班';

/*数据插入验证 */

baseModel.insert(studentName, studentInfo, function(ret){
  console.log(ret);
});

baseModel.insert(className, classInfo, function(ret){
  console.log(ret);
});

```

执行 index.js 代码后，再查看 school 数据中的两个表 student_info 和 class 中的数据。执行结果如图 7-26 和图 7-27 所示。

+ 选项

← T →	class_id	name
<input type="checkbox"/> 编辑 快速编辑 复制 删除	1	三年级二班

图 7-26 数据库 class 插入后查询结果

+ 选项

← T →	student_id	name	age	class_id
<input type="checkbox"/> 编辑 快速编辑 复制 删除	1	danhuang	22	1

图 7-27 数据库 student_info 插入后查询结果

最后实现一个简单的类继承 base_model.js，代码如下：

```

var BaseModel = require('./base_model')
,   util = require('util');
function StudentInfo(){
  BaseModel.call(this);
  util.inherits(BaseModel, this);
}

```

```

var tableName = 'student_info'
, _self = this;
/* 数据新增 */
this.doAddStudent = function(studentInfo){
  _self.insert(tableName, studentInfo, function(ret){
    console.log(ret);
  });
}

var studentInfo = new StudentInfo();
exports.doAddStudent = studentInfo.doAddStudent;

```


2. 根据 7.3 节介绍的 MongoDB 知识设计一个 MongoDB 学生数据库，学生数据库请设置超过两个表，如 student_info 和 class。其中 student_info 表数据有如 student_id、name 和 age 等字段，确定另外表的字段以及每个字段类型，最后绘制一张数据设计表格。

分析：本习题可以根据如下步骤来分析。


(1) 使用 MongoDB 的工具 mongo 操作数据库，创建 student 数据库。

(2) 应用 BaseMongoDB 基类，向数据库插入数据、查询数据和删除数据等操作。

(3) 应用 BaseMongoDB 基类以及前面介绍的 Node.js 的继承方法，实现多个 Model 层继承 BaseMongoDB，并应用这些 Model 层实现增、删、改和查操作。

 **注意：**数据库的设计可以尽量简单，Node.js 的继承方法可以参考本书的 2.3 节。

3. 本章应用了 Siege 工具进行了数据的插入性能测试，现在大家使用同样的方法和工具来完善本书中 MySQL 和 MongoDB 性能测试对比。

 **提示：**首先应用 7.2 节和 7.3 节的 BaseModel 和 BaseMongoDB 来创建一个数据查询代码，之后应用 Siege 工具来产生大量的并发请求，Siege 测试结束后，查看 Siege 产生的数据测试返回日志来绘制曲线图，通过对比曲线图来分析两者性能的差异。

分析：MongoDB 无需创建表和初始化表单字段，因此我们直接编写插入 MongoDB 的代码，代码如下所示。

```

var BaseMongoDB = require('./base_mongodb')
, baseMongoDB = new BaseMongoDB()
, studentInfo = {}
, classInfo = {}
, studentName = 'student_info'
, className = 'class';
studentInfo.name = 'danhuan';
studentInfo.age = '22';
studentInfo.class_id = '1';

classInfo.class_id = '1';
classInfo.name = '三年级二班';
/* 数据插入 */
baseMongoDB.insert(studentName, studentInfo, function(ret){
  console.log(ret);
});

```



```
baseMongoDB.insert(className, classInfo, function(ret){
    console.log(ret);
});
```

执行 index.js 代码，即可实现数据插入到 MongoDB。

接下来实现一个 model 层类 StudentInfo 继承 BaseMongoDB 基类。

```
var BaseMongoDB = require('./base_mongodb')
    , util = require('util');
function StudentInfo(){
    BaseMongoDB.call(this);
    util.inherits(BaseMongoDB, this);

    var tableName = 'student_info'
        , _self = this;

    this.doAddStudent = function(studentInfo){
        _self.insert(tableName, studentInfo, function(ret){
            console.log(ret);
        });
    };
}

var studentInfo = new StudentInfo();
exports.doAddStudent = studentInfo.doAddStudent;
```

分析：本习题请读者独立完成。

7.6 本章小结

本章中介绍了 Node.js 操作的两个数据库，分别是 MySQL 和 MongoDB，其中需要读者掌握两个数据库的安裝配置，以及基本的命令行操作方法，同时理解两者之间的区别和联系。

在学习本章时需要掌握 Node.js 操作 MySQL 和 MongoDB 的基类实现过程，同时要灵活地应用两个基类，以及应用其他模块来继承该基类的方法。学习本章知识要多注重代码实践，其中需要读者亲自应用基类来操作数据库，并应用多个模块来实现继承基类应用。

本章中最后一节介绍的是 MySQL 于 MongoDB 的性能测试对比，目的是让大家了解一些测试工具，以及了解如何应用测试工具进行一些 Web 应用的性能测试。

本章介绍的是 Siege 工具，类似的工具还有 apache 自带的 ab 测试工具，如果了解更多的 Web 性能测试工具，可以前往笔者的个人博客查看《十个免费的 Web 压力测试工具》，网址为 <http://blog.csdn.net/danhuang2012/article/details/8246616>。

第 8 章 MyWeb 框架介绍

Node.js 的 MyWeb 是一个基于 Node.js 的 Web 应用框架，其中应用到了 `express` 模块和 `jade` 解析模板。整体上来说，该框架只是在 `express` 框架上搭建了一层 MVC 实现方式。本章介绍该框架的目的如下：

- ❑ 进一步了解 Node.js 的 Web 应用开发。
- ❑ 更熟悉 `express` 框架的应用。
- ❑ 快速地搭建一个 Node.js 应用。

MyWeb 框架现在笔者已经没有再维护，但是还是希望通过介绍本章能够给大家带来更多的知识，虽然 MyWeb 框架的实现非常简单，也没有太多的亮点，但是其中的一些编程思想，以及解决问题的办法还是有学习和利用的价值。本章介绍主要分为以下 3 方面：

- ❑ 框架的基本介绍，以及应用的一些模块概述。
- ❑ 框架的整体设计类图和框架的文件目录结构介绍。
- ❑ 框架的部分重要逻辑实现原理。

本章不涉及框架的应用，该框架的应用会在第 9 章介绍。学习本章的重点是框架的设计思想，以及框架的一些重要逻辑实现代码，其中涉及路由处理、代码日志工具类和邮件 NPM 模块的应用等。

8.1 MyWeb 框架介绍

MyWeb 是一个基于 Node.js 的 Web 应用框架，其中应用到了 `express` 框架和 `jade` 解析模板。整体上来说，该框架只是在 `express` 搭建了一层 MVC 实现方式。那么 MyWeb 框架给开发者带来了什么便利呢？

8.1.1 MyWeb 框架涉及的应用

MyWeb 框架的开发中，我们借鉴了一些 PHP 的开发经验，包括其中的一些路由处理、MVC 模式的实现和日志处理。以下是本框架涉及的一些知识点。

1. MVC 开发模式

在 PHP 开发中我们经常会应用 MVC 设计模式开发出一套框架，而 Node.js 现在处于发展初期，还没有比较成熟的 MVC 开发框架，大部分都是通过自我根据 `express` 框架进行搭建。本框架也属于个人搭建的 MVC 框架。该框架很适合 PHP 开发者，框架实现主要是借鉴 PHP 的 MVC 模式来搭建的。

2. 路由处理

路由处理是框架的一个基本的也是关键的部分，路由处理的方式直接影响到 URL 的显示方式。本框架路由实现逻辑经历了两个过程。

(1) 直接通过 key 值来判断用户请求的资源。例如 `http://127.0.0.1:3000/index`，这个请求我们利用 `express` 获取 key 值 `index`，然后读取配置文件信息。

```
index{file:'index_controller',class:'indexController',function:'loginAct'}
```

从而根据 `index` 值得到了所有的 `controller` 和 `action`。关于如何去访问这个 `action` 将在 8.3 节详细介绍。

(2) 第一种方式，不知道大家是否会发现一个问题。当我们有 300 个 `action` 时（很正常），我们需要写入 300 条记录到一个配置文件中，这样会导致配置文件越来越繁重，因此就出现了第二种路由处理方式。

```
http://127.0.0.1:3000/index?c=login
```

根据 `index` 获取当前的 `controller`，根据 `c` 的参数获取当前需要访问的 `action`，这样我们的一个 `controller` 只对应一个配置信息，从而减少读取配置文件的时间。

3. 数据库链接

本框架只提供了一种链接 `MySQL` 的方式，希望在后期能够提供更多的数据库链接。链接 `MySQL` 数据库的方式，这里就不需要细讲，大家可以通过在 `github` 官网上查找关键字 `node MySQL` 字符来搜索相应的学习资源。

本框架封装了一些基本的数据库操作，从而减少开发者开发的时间，只需要通过 `add`、`update`、`delete`、`query` 来 `select` 简单接口的调用即可。

4. 日志处理

本框架实现了一个自我的日志记录处理功能，主要是便于系统运营。其中的日志设定错误、警告和流水记录，同时可以设定日志错误级别，以便系统出错时即时地定位。

5. 邮件发送功能

这个功能没有做进一步的封装，只是利用了他人的库进行配置，之后会做进一步封装。

6. 简单易用

只需要一步就可以实现本系统运行，下载该框架代码，执行 `node index.js` 就可以运行本框架（如果使用到 `Session` 登录，请先下载 `Windows` 版本的 `redis-server.exe`，该工具在 `redis` 的 `github` 源码库中可以下载获取，如果是 `Linux` 可以直接安装该 `NPM` 模块）。

8.1.2 MyWeb 框架应用模块

本框架主要应用到了 3 个重要的模块，`express`、`jade` 和 `connect-redis` 模块。其中 `express`

和 jade 模块在前面几章中已经有详细的介绍，本节只对其应用做简单的概述。

1. redis 模块

redis 是一个 key-value 存储系统，和 Memcached 类似，它支持存储的 value 类型相对更多，包括 string（字符串）、list（链表）、set（集合）、zset（sorted set --有序集合）和 hashes（哈希类型）。这些数据类型都支持 push/pop、add/remove 取交集、并集和差集及更丰富的操作，而且这些操作都是原子性的。在此基础上，redis 支持各种不同方式的排序。与 memcached 一样，为了保证效率，数据都是缓存在内存中。区别是 redis 会周期性地把更新的数据写入磁盘或者把修改操作写入追加的记录文件，并且在此基础上实现了 master-slave（主从）同步。

redis 是一个高性能的 key-value 数据库。redis 的出现，在很大程度上补偿了 memcached 这类 key/value 存储的不足，在部分场合可以对关系数据库起到很好的补充作用。它提供了 Python、Ruby、Erlang 和 PHP 客户端，使用很方便。

在 express 框架中主要是应用 redis 来存储系统的 Session。其中 NPM 中有一个 connect-redis 模块，该模块就是一个基于 node_redis 模块的 Session 存储管理应用。

connect-redis 的 NPM 模块和下载安装方法和一般的 NPM 一样，执行 npm install connect-redis 命令安装 connect-redis 模块。

需要注意的是，在应用 redis 时，首先需要安装其服务端，其包含了 Windows 系统版本和 Linux 系统版本，其官网只提供了 Linux 版本的安装配置方法，相应内容可以在 <http://redis.io/download> 中了解学习。

2. MySQL 模块

本模块在第 7 章中有详细的介绍，包含其具体的应用，以及接口封装实现，而本框架应用 MySQL 模块的基本操作 API 来封装实现应用层的 MySQL 基类。

3. Express 模块

本框架主要是基于 express 接口 API 实现的一个 MVC 框架模型，其大部分接口都依赖于 express 框架，包括其中的路由处理和静态文件资源服务器等。

4. jade 模块

jade 模块在第 2 章中已经简单的介绍过，该模块是本框架前端的一个模版。

8.2 MyWeb 源码架构

上一节简单介绍 MyWeb 框架的功能，以及 redis 的应用模块，本节会介绍本框架源码的架构，以及文件夹构建说明，通过框架架构和文件目录来进一步熟悉本框架的实现。

8.2.1 框架 MVC 设计图

如图 8-1 所示是本框架设计的一个简单的流程图。从图中体现出 MVC 框架中最重要

的 3 个部分 M、V 和 C，M 为 Model 数据库操作控制模块，V 为 Web 客户端 Web 页面处理模块，C 为 Controller 业务逻辑处理模块。

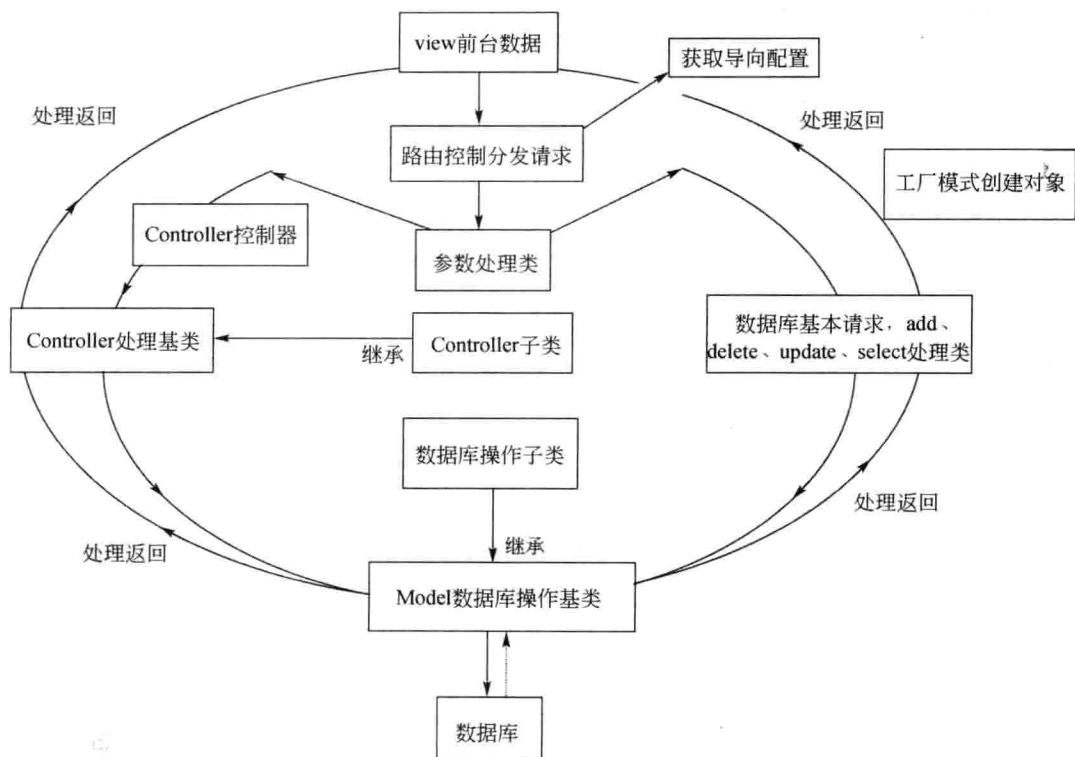


图 8-1 框架模块设计

本框架的请求过程：客户端用户请求资源，路由处理模块通过分析用户请求的 url，将其转发到相应的处理 Controller 层，Controller 层通过数据组装调用 Model 层模块类向数据库插入数据，执行结束后通过反向的返回执行结果。

这个流程图是根据 MVC 设计模式设计出来的，其结构、设计思想和其他 MVC 设计模式的应用框架类似。主要是理解一个请求过程中是如何串联起 M、V、C 三者的关系。根据图可以了解到本框架实现的三个重要模块是：1. 路由控制分发处理逻辑；2. Controller 基类逻辑模块；3. Model 数据库操作基类模块。关于这三点的实现，将在 8.3 节中重点介绍，希望读者在该框架的实现过程中学习其应用方法。

8.2.2 框架文件结构

项目框架决定着一个项目的文件结构，同时通过文件结构也可以清晰的了解一个项目。本项目的文件结构如图 8-2 所示。

index.js 为项目框架的启动入口文件，通过运行 index.js

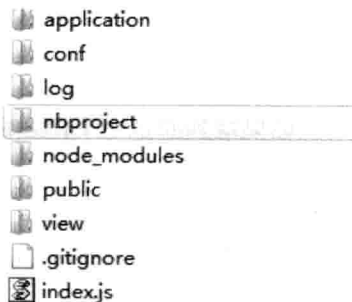


图 8-2 框架文件夹结构

来启动服务器的运行。如表 8.1 所示为主文件目录结构和子目录结构说明。

表 8.1 文件夹结构分析表

主 文 件 夹	子 文 件 夹		文 件 说 明
application	common	存放 application 公用文件	本文件夹存放核心的 MVC 设计模式中的 M 和 C，控制应用的整体逻辑
	controller	逻辑处理层	
	core	处理基类文件	
	model	数据处理层	
conf	无		配置文件：路由处理、log 处理、数据结构、数据库链接数据
log	无		记录系统运行日志
public	css	view 层的样式	主要是 css 文件、JavaScript 文件和图片资源
	js	view 层的 JavaScript	
	img	图片资源文件	
	express、jade、MySQL、socket.io、qs、connect 等		所有的 Node.js 的模块
view	source	资源文件	jade 文件.html 文件

项目的处理核心在 application 文件夹中，Model 层和 Controller 层分别在 application 文件夹下的 model 和 controller 下，Model 基类和 Controller 基类在 core 文件夹下。view 文件夹为 Web 前端页面处理逻辑，主要是一些 jade 的模板文件和 html 文件。Public 为项目的静态资源文件，log 为项目的运行日志，conf 则为项目的配置信息，包括数据库用户名、密码和路由处理控制等。

8.2.3 扩展阅读之更快地了解新项目

了解一个大项目必须要有一定的技巧和方法。本节简单地总结一下个人的认识和想法。

1. 首先必须了解项目的目录结构

拿到一个项目，首先必须要了解这个项目的文件结构，有时候通过文件结构我们就能够清楚这个项目使用什么框架。比如 thinkphp、strut 和 django 这些框架的文件目录结构都非常的清晰明了，只要看到结构就能够明白。对于之前的开发者，他对文件结构肯定有自己的一套想法，所以要分析清楚每一个文件夹的主要作用。最好记录下每个文件夹下面的文件的主要作用，可以通过使用目录树注释的方法。

2. 了解文件命名规范

如果拿到的是一个比较成熟的项目，项目的文件命名一定是有规范的。当然，小项目那就可以随便一点，但是对于一个大项目必须有一定的命名规范，这样才能使开发者之间有效沟通。文件的命名规范一般有以下几种。

- ❑ 根据文件夹来命名，比如 app 文件夹下的 admin 下的 indexAction.php，那么 indexAction 中的类名可以命名为 App_Admin_indexAction。
- ❑ 根据文件的命名，直接根据类的名称来命名，比如类名为 BaseMode，则该类的文

件为 `base_mode.php`。

- 根据文件的作用命名，比如是 `class` 的就使用 `name.class.php`，是 `view` 的，就使用 `name.view.php`，为 `model` 类型的命名为 `name.model.php`。其实命名规范是有很多的，而且关键是看项目开发人员之间的沟通协调，或者是根据框架的命名规范而来。

3. 了解代码的处理过程（如果有框架先学习框架）

如果有框架的话，建议大家先学习框架，明白框架的基本调用关系，再来了解项目，那就是轻而易举的了。

没有框架的话，大家还是一步一步地来。一般的项目都有一个入口文件，`index.html` 或者 `index.php`、`index.jsp` 等，一定要把这个入口文件的代码看懂，不要轻易放弃。如果入口文件没有了解清楚的话就很难了解整个项目是如何调用的，如何运行起来的。一般情况，入口文件会跳转其他文件，或者是包含其他文件，这时候也要了解清楚。如果是使用了一些设计模式的话，还应该要先了解一下设计模式，比如说 `MVC`，现在的开发大部分都使用 `MVC` 设计模式，所以大家还是要了解一些基本的知识。比如基本的设计模式、基本的框架，以及那些框架下的一些文件结构和命名规范，以备以后自我的开发使用。了解了项目的入口文件后要达到一个目标，就是能够明白输入一个 `URL` 后，项目是如何获取 `url` 上的信息，从而访问得到用户所需要的信息的。

4. 根据上述的项目为项目添加一些新功能

如果已经了解了项目的代码处理过程，那么我们就可以基于这个项目开发一个小的应用。建议：根据文件结构、文件命名规范，添加一个新的类、新的方法，添加一个新的页面。最后通过输入相应的 `URL` 访问数据，如果成功显示你想要的信息的话，那么恭喜你，你已经对这个项目有了一个较为基本的理解。接下来就是去做相应的需求分析，然后添加新的应用，如果这四步能够做好的话，接下来添加新的应用就只是实现的问题了。

8.3 框架源码分析

前面两节介绍框架的基本功能模块和框架设计流程图，本节主要介绍本框架的一些重要模块代码的实现逻辑，包括入口文件模块、路由处理逻辑模块、数据层基类模块和逻辑层基类模块。本框架代码由于没有维护，因此现在讲解，会看出一些问题，我们通过问题来分析框架需要优化并重构的一些东西。

8.3.1 框架入口文件模块

本框架的入口文件为 `index.js`，该入口添加了项目所需的多种全局静态变量，分别是项目文件夹的路径全局变量和 `require` 返回的模块对象全局变量。实现代码如下：

```
//=====全局变量定义=====
global.BASE_DIR = __dirname;
```

```

global.APP      = global.BASE_DIR + "/application/";
global.CON      = global.APP + "/controller/";
global.CORE     = global.APP + "/core/";
global.MODEL    = global.APP + "/model/";
global.CONF     = global.BASE_DIR + "/conf/";
global.LOG      = global.BASE_DIR + "/log/";
global.PUBLIC    = global.BASE_DIR + "/public/";
global.VIEW     = global.BASE_DIR + "/view/";

```

【代码说明】

- ❑ `global.BASE_DIR = __dirname`: 定义项目的根目录全局变量。
- ❑ `global.APP`: 定义 `application` 文件夹的全局变量。
- ❑ `global.CORE`、`global.CON`、`global.MODEL` 等类同 `APP` 全局变量的作用。

入口文件定义项目所需的所有路径全局变量。当需要修改项目的文件结构时,只需要更改相应的全局变量值,不需要去查询并修改项目中的路径,那样不仅会影响开发者的工作效率,而且也可能增加项目代码更改后的风险(产生一些不可预估的缺陷)。代码如下:

```

/**
 * modules 引入
 */
global.Module = {
  express : require('express'), // express 框架
  sio : require('socket.io'), // socket.io 模块
  fs : require('fs'), // fs 文件处理模块
  path : require('path'), // path 路径处理模块
  url : require('url'), // url 路径处理模块
  parseCookie : require('connect').utils.parseCookie,
  MemoryStore : require('./node_modules/connect/lib/middleware/session/memory'),
  Session : require('./node_modules/connect/lib/middleware/session/session'),
  sys : require('util')
}

```

【代码说明】

- ❑ `global.Module = {...: require 项目所需的模块。`

入口文件定义项目所需的所有模块的全局变量,当然这里可以考虑,如果使用较为频繁的模块对象,可以使用全局变量进行存储,而那些使用次数只有一次的模块对象可以不用添加到全局设置中。代码如下:

```

/*
 *初始变量,主要是初始一些静态变量
 */
global.initVar = {
  routerConfig : "",
  errorConfig : "",
  serverConfig : ""
}

```

【代码说明】

- ❑ `global.initVar = {}`: 初始化项目中所需要的部分全局变量。

这部分在框架设计时的目的是为了存储读取配置后的 `Json` 信息,但是介于当初设计的原因没有将配置文件读取作为一个公用模块来缓存这些数据,所以使用了全局变量来定义。

配置文件读取并缓存的功能需求，可以在后期通过一个配置文件管理模块来处理，而无需在入口文件中进行初始化全局变量来缓存。

入口文件中还有一个关键逻辑，就是创建 HTTP 服务器，并进行路由分发处理的功能，代码如下：

```
urlResolve = require(CORE + "url_resolve");
urlResolve.getActionInfo();
```

【代码说明】

❑ urlResolve = require(CORE + "url_resolve")：获取 url 解析模块。

❑ urlResolve.getActionInfo()：进行 url 解析并分发请求资源的逻辑处理模块。

这部分代码中的 getActionInfo 命名不规范，按道理来说该函数应用是执行 HTTP 请求资源逻辑，而 getActionInfo 仅仅是获取请求的 Action 信息。

介绍并学习完入口文件后，分析出对于路径常量，以及常用的模块对象可以使用全局变量进行缓存，减少代码重构和扩展带来的问题，增加代码的可维护性。本入口文件中存在的缺陷：配置文件信息不应该使用全局变量进行缓存，而应该使用配置文件读取模块进行统一管理；urlResolve 接口函数命名存在问题，会误导代码的阅读者，需要进行重构。

8.3.2 路由处理模块

本框架的路由处理模块在 application 文件夹下 core 中的 url_resolve.js 脚本文件。该路由模块包含了 HTTP 服务器构建、url 解析和 url 解析后转化为逻辑处理参数，最后根据转化后的参数分发请求到相应的逻辑处理 Controller 层。

入口文件模块中介绍到了 urlResolve 模块对象中的 getActionInfo 逻辑处理函数，该函数为 url_resolve.js 脚本文件中的入口函数接口。其实现代码如下：

```
exports.getActionInfo = function(){
  systemConfig(); // 系统配置，包含 express 的初始化过程
  /* 处理 HTTP 所有 get 请求 */
  app.get('/:key', function(req, res){
    callUrlRequest(req, res);
  });
  /* 处理 HTTP 所有 post 请求 */
  app.post('/:key', function(req, res){
    callUrlRequest(req, res);
  });
  /* 服务器启动并监听 */
  listenPort();
};
```

使用 exports 来返回 getActionInfo 函数接口，其中的 systemConfig 函数为 urlResolve 模块的私有方法，其主要是设置 express 框架中的一些常有变量，主要有项目的静态资源配置和 Session 存储管理配置等。相关 express 框架的配置可以参考 express 官网¹。

app.get 和 app.post 分别处理 GET 和 POST 请求，callUrlRequest 方法是根据 URL 请求资源路径分发处理请求逻辑，listenPort 启动服务器监听端口。

¹ <http://expressjs.com/guide.html>。

我们接着看一下 `callUrlRequest` 是如何分析服务器请求资源并转发请求逻辑实现。下面的代码是 `callUrlRequest` 的实现逻辑：

```
function callUrlRequest(req, res){
    //获取配置，如果配置信息不变，直接获取
    var routerMsg = initVar.routerConfig ? initVar.routerConfig :
    getUrlConf(),
        key = req.params.key,
        fun = (req.query.c ? req.query.c : "loginPage")+ "Act";
    var session = checkSession(req, fun);
    //去除浏览器自带 favicon.ico 请求
    if(key == "favicon.ico"){return;};
    if(session == 0){
        res.redirect('/index');
        return;
    }
    // 打印参数信息
    console.log("[key: "+ key +"] " + "[class: " + routerMsg[key].cla + "] "
+ "[controller: " + fun + "]");
    require(CON + routerMsg[key].con);
    var controllerObj = eval("new " + routerMsg[key].cla);
    controllerObj.init(req, res);
    controllerObj[fun].call();
}
```

【代码说明】

- ❑ `routerMsg`：路由配置信息。
- ❑ `key = req.params.key`：利用 `express` 框架的 `req.params` 获取 URL 请求路径。
- ❑ `fun = (req.query.c...)`：获取 URL 中的 GET 请求参数，请求的参数 `key` 值为 `c`。
- ❑ `if(key == "favicon.ico")...`：去除浏览器 `favicon.ico` 自带请求。
- ❑ `require(CON + routerMsg[key].con)`：require 相应的处理 `controller`。
- ❑ `controllerObj = eval("new " + routerMsg[key].cla)`：应用 `eval` 来 `new` 一个 `controller` 的字符串名。
- ❑ `controllerObj.init(req, res)`：初始化 `controllerObj` 对象的两个私有变量 `req` 和 `res`。
- ❑ `controllerObj[fun].call()`：执行 `controller` 中相应的 `action` 函数。

根据 url 请求的 `key` 值得到相应的配置信息。配置文件可以展示如下：

```
{
  "test": {
    "con" : "test",
    "cla" : "test",
    "fun" : "test"
  },
  "favicon.ico" : {
    "con" : "",
    "cla" : "",
    "fun" : ""
  },
  "login" : {
    "con" : "index_controller",
    "cla" : "IndexController",
    "fun" : "loginAct"
  },
  "index" : {
    "con" : "index_controller",
```

```

    "cla" : "IndexController",
    "fun" : "loginPageAct"
  }
}

```

配置文件中的 test、favicon.ico、login 和 index 为 URL 中的请求路径中的 key 值。例如请求 url 为 http://127.0.0.1:1337/test，则 test 为请求 URL 的 key 值，根据该 key 值 test 获取到其中的 con、cla 和 fun。其中，con 为 controller 文件名，cla 为 controller 类名，fun 为需要处理请求 URL 的逻辑 action。

根据 URL 中的 key 和配置文件，获取到 URL 请求需要处理的 controller、class 和 function 参数。require 相应的 controller，使用 eval 来 new 相应对象的字符串类名，使用 controllerObj[routerMsg[key].fun].call() 方法进行调用。由于这里涉及的类名、方法名都为字符串，如果直接 new 一个字符串就会报错，因此需要通过 eval 来 new。同样，function 名也是一个字符，因此不能通过 controllerObj.fun 直接调用。

路由处理模块方面，我们可以总结出以下几个知识点：

- ❑ 路由配置信息，可以通过配置文件进行存储管理。
- ❑ Express 框架获取 HTTP 的 URL 请求路径方式 req.params.key，获取 HTTP 请求的 get 参数方式 req.query.c。
- ❑ 调用一个字符串的类和类中的字符串函数名的方法，调用方式使用 eval 来 new 一个类对象，通过调用类对象的函数名方法获取函数对象，最后使用 call 方法实现函数对象的执行。

上面只是其中的一种路由实现方法，主要是通过一个 key 来实现路由判断。在第 1 章中我们介绍了多种路由实现方式，大家可以参考前面的知识优化本框架的路由处理模块。

8.3.3 Model 层基类

本框架的 Model 基类和数据库一章中的 MySQL 基类有点类似，其主要的的作用就是数据库连接和数据库的基本操作。

该类的 MySQL 数据库连接方式已经更改过，连接方式修改为数据库一章中的 MySQL 基类的数据库连接方式，代码如下：

```

var MySQL = require('mysql'),
    cfg = initVar.serverConfig ? initVar.serverConfig : getServerConf(),
    dbClient,
    Log = require(CORE + "log.js");
function init(){
  /* 获取 MySQL 配置信息 */
  client = {};
  /* 获取 MySQL 的服务器地址 */
  client.host = cfg['host'];
  /* 获取 MySQL 的服务器端口 */
  client.port = cfg['port'];
  /* 获取 MySQL 的数据库用户名 */
  client.user = cfg['user'];
  /* 获取 MySQL 的数据库密码 */
  client.password = cfg['password'];
  /* 创建 MySQL 数据库连接 */

```

```

dbClient = MySQL.createConnection(client);
dbClient.query('USE ' + cfg['dbName'], function(error, results) { // use
                                                                    数据库
    if(error) {
        console.log('ClientConnectionReady Error: ' + error.message);
        dbClient.end();
        return false;
    }
});
}

```

【代码说明】

- ❑ mysql = require('mysql'): 调用 MySQL 模块对象。
- ❑ cfg = initVar.serverConfig: 读取 MySQL 数据库连接配置信息。
- ❑ function init(){...: 基类构造函数，创建数据连接。

本框架还包括 add、update、deleteItem、select 和 query 的 API 接口，对应数据库的增加、修改、删除和查询操作。这几个接口的实现类似于数据库一章的 MySQL 基类操作实现，这里只简单介绍其中的 add 操作方法实现。其代码如下：

```

/**
 *
 * @desc 数据插入
 * @params object values
 * @params function callback
 */
this.add = function(values,callBack){
    tableMsg = getTableConfigFromJson(this._table);
    /* 数据转化 */
    var addMsg = changeJsonToString(values);
    /* 执行 MySQL 数据插入 */
    dbClient.query('INSERT INTO ' + this._table + "(" + addMsg["key"] + ")"
values(" + addMsg["value"] + ")",
        function(error, results) { // 回调函数获取执行结果
            if(error) {
                dbClient.end();
                callBack(0);
                return false;
            }
            /* 回调函数返回数据插入后的 ID */
            callBack(results.insertId);
        });
}

```

tableMsg 和 addMsg 变量操作的目的是为了组合一个 MySQL 数据 add 语句。通过调用 MySQL 模块对象的 query 方法来执行该 sql 的数据库数据增加操作，通过传递 callback 回调函数来返回数据 add 的操作执行结果，成功，就返回插入后数据的 id 值；不成功，则返回数据参数 0，表示失败。

这部分代码实现相对第 7 章的 MySQL 数据库操作基类的实现略显粗糙，没有很好地利用到 MySQL NPM 模块对象中的 ? 操作符，以及 dbClient.query 接口。? 字符变量可以自动地将 json 对象转化为 key=value 的形式，而不需要自己通过代码去实现。

我们对比一下之前的 MySQL 基类对象的代码实现，代码如下：

```

/**
 *
 * @desc 向数据库插入数据
 * @param tableName string
 * @param rowInfo json
 * @param callback function
 * @return null
 */
this.insert = function(tableName, rowInfo, callback){
    dbClient.query('INSERT INTO ' + tableName + ' SET ?', rowInfo,
function(err, result) {
    if (err) throw err;
    callback(result.insertId);
});
};
};

```

相对框架中的 Model 基类来说，该方法简单清晰，其利用了 dbClient.query 接口中的自带替换符?，使 sql 语句显得更清晰易懂。

Model 层基类可以总结出以下知识点：

- ❑ Node.js 使用 MySQL 模块对象连接 MySQL 的方法。
- ❑ MySQL 模块对象 db.query 接口的使用方法。
- ❑ MySQL 模块对象 db.query 接口中的?替换符的作用，以及使用方法。

8.3.4 Controller 层基类

本框架也有一个 Controller 基类，该基类在 core 文件下，名为 base_controller.js。该类是所有 controller 的父类，其主要是所有 controller 一些公有方法实现，本框架已实现的接口有 displayHtml、displayJade 和 returnError。

下面是 Controller 基类的代码实现。代码如下：

```

require (MODEL + "user_model.js");
function BaseController(){
    var _self = this;
    _self._req;
    _self._res;

    // 数据初始化
    this.init = function(req, res){
        _self._req = req;
        _self._res = res;
    };

    //显示一个html 文件
    this.displayHtml = function(htmlName){
        var file = Module.fs.readFileSync(VIEW + htmlName);
        _self._res.end(file);
    };

    //指定显示一个jade 文件
    this.displayJade = function(jadeName, json){
        _self._res.render(jadeName, json);
    };

    //返回 json 数据到客户端

```

```


    this.returnError = function(code,msg,dataJson){
        _self._res.send({"code":code, "msg":msg, "data":dataJson});
    };
}
global.BaseController = BaseController;

```

【代码说明】

- ❑ `this.init = function(req, res){...}`: `init` 为 `BaseController` 基类的构造函数。
- ❑ `this.displayHtml`: 响应一个 `html` 页面数据到客户端。
- ❑ `this.displayJade`: 指定解析一个 `jade` 模板文件, 并返回其解析后的数据到客户端。
- ❑ `this.returnError`: 响应一个异步请求的数据返回, 返回的信息包含错误码、错误信息和返回的数据。

这里涉及的 `res.send` 方法是 `express` 框架提供的 `jade` 解析 API 接口, 该接口接受两个参数, 分别为 `jade` 模块文件名和需要向 `jade` 模板文件中传递的数据。对于一般的 `html` 页面内容, 我们可以直接调用 `res.end` 方法。

 提问: `controller` 基类中含有一个私有变量 `_self`, 该变量被赋值为 `this`, 请问是否有必要将 `this` 赋值给 `_self`, 如果有必要那么原因是什么?

在介绍本章之前, 有讲解到 `MongoDB` 的数据库基类实现, 其中就涉及了将 `this` 指针赋值给其他变量的情况, 原因主要是在一个类中 `this` 指针是时刻在变动的。

8.4 本章实践

1. 应用本框架实现两个 `model` 层模块继承 `MySQL` 基类模块。

分析: 本章有介绍如何应用其他模块来实现 `MySQL` 继承方式, 此题为开发性题目请读者自行完成。

2. 应用本框架实现两个 `controller` 继承 `controller` 基类模块。

分析: 本章有介绍如何应用其他模块来实现 `MySQL` 继承方式, 此题为开发性题目请读者自行完成。

3. 应用本框架实现一个简单的登录系统, 用户登录后, 提示用户登录成功, 用户登录失败, 则提示用户登录失败。

分析: 简单的登录系统在本章的应用中已经有简单的实现, 请读者依据本章的代码实现。

8.5 本章小结

本章介绍了 `MyWeb 1.0` 框架的一些基本设计架构, 以及实现的功能。在本章中还涉及 `MyWeb` 框架的实现, 包含 `MyWeb` 框架的入口启动文件、路由模块、`Model` 层和 `Controller` 层逻辑实现。

学习本章需要掌握 `express` 框架的应用基本应用, 包括 `express` 配置启动、`express` 中的 `HTTP` 参数获取、`express` 中的客户端请求响应数据返回 `html` 和 `jade` 等。重点是掌握 `MyWeb`

框架中的路由模块实现逻辑，学习本框架中的文件路径管理和 NPM 模块管理方式。

下一章将介绍 MyWeb 框架的一个简单聊天室的应用。根据本书中的代码规范，下面是本框架的一个简单的代码规范。这只是一个规范并不是要求，大家可以根据下面的规范做一个参考。

- ❑ 变量命名：私有变量统一使用 “_name”，全局变量使用大写 “VIEW”，简单变量请使用驼峰峰 “myName”；
- ❑ 方法命名：所有方法请使用驼峰峰 “getUrlRequest”；
- ❑ 类命名：统一使用首字母大写驼峰峰 “BaseController”；
- ❑ 文件命名：统一使用下划线分割，类使用下划线分割 base_controller.js。

第9章 框架应用 MyChat

上一章主要是从框架的设计，以及框架源代码层面上做介绍，本章将会从框架的应用上介绍使用框架来实践开发简单的项目。“300 块立即建站，有数据库”，这就是本框架的最终目的。本框架的设计实现不是很复杂，但是非常适合初学者进行快速地项目开发。

本框架的应用可以总结为 4 个过程，新增 M、V、C 模块文件和修改路由配置。相应文字描述如下：

- ❑ 添加一个 Model 类继承 Model 基类，操作数据库中表单的数据。
- ❑ 添加一个 Controller 类继承 Controller 基类，处理业务逻辑。
- ❑ 添加一个 jade 模块页面，显示前端页面信息。
- ❑ 修改配置文件夹 conf 中的 router.json 文件，添加一条新的 controller 路由映射关系。

相关流程如图 9-1 所示。

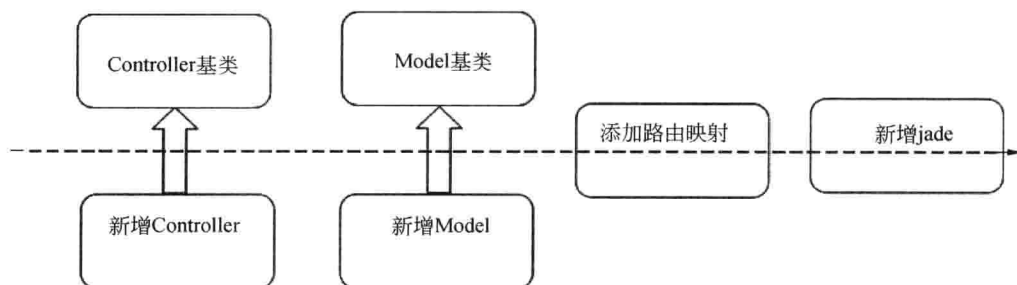


图 9-1 框架应用过程

如果你的项目是应用 MyWeb 框架去实现，那么项目的开发就是重复着这 4 个过程。本章将会实践性的根据这 4 个过程一步步开发，实现一个简单的聊天室项目应用。本章主要分为两节，9.1 节主要介绍聊天室应用的需求描述、数据库的设计和框架的应用配置，9.2 节实践 4 个过程来开发项目。

9.1 编码前的准备

本节以 Node.js 实现的一个 Web 聊天室 MyChat 应用来介绍本框架的应用。本应用主要包含 3 个过程的实现：controller 实现、model 实现和 view 的 jade 模板实现。学习本节需要了解如何继承本框架的 Model 基类和 Controller 基类，进一步熟悉本框架的运行机制。

9.1.1 应用分析

本应用主要包含 3 个功能模块，用户注册、用户个人信息管理和用户聊天功能。用户注册功能主要是应用 Node.js 邮件功能，通过发送邮件验证来激活账号。用户个人信息管理，主要是简单的对用户头像和用户基本信息的修改。聊天功能主要是应用 socket.io 模块来实现。

针对以上需求介绍，我们设计出本系统的数据库，详细数据库 db_chating 如表 9.1 所示。

表 9.1 MySQL数据库db_chating字段设计表

表 名	表 描 述	字 段	字 段 类 型	字 段 描 述	备 注
t_user	用户个人信息	f_uid	int	主键	
		f_uname	varchar(100)	登录用户名	
		f_nike_name	varchar(100)	昵称	
		f_sex	int	性别	0 为男， 1 为女
		f_email	varchar(100)	邮箱	
		f_password	varchar(100)	密码	
		f_position	int	地理位置	
		f_university	varchar(100)	大学	
		f_hometown	varchar(100)	家乡	
		f_high_school	varchar(100)	高中	
		f_atvatar	varchar(100)	头像	
		f_create_time	timestamp	注册时间	
		f_login_state	int	登录状态	
		f_state	int	账号状态	0 为非激活， 1 为激活

由于这里只涉及一个用户功能，因此这里就不详细介绍其他表单的设计。根据如上数据设计，接下来就可以创建数据库（字符编码为 utf8），可以通过将源码中的 db_chating.sql 导入数据库，当然也可以应用 MySQL 工具创建，操作使用方法请参考第 7 章。

9.1.2 应用模块

MyWeb 框架的实现是基于 express 模块框架，因此 express 为本应用的一个重要模块，其次由于本系统存在用户的登录和管理，因此需要使用 redis 做 Session 的存储管理。数据库使用的是 MySQL，因此需要使用 Node.js 的 MySQL 驱动模块。socket.io 模块实现聊天功能。在前面几章中已经介绍了 MySQL、express 和 socket.io 模块，本节重点介绍 redis，以及 redis 的应用。

redis 是一个 key-value 存储系统，它支持存储的 value 类型相对更多，包括 string（字符串）、list（链表）、set（集合）和 zset（有序集合）。这些数据类型都支持 push/pop、add/remove 及取交集、并集和差集及更丰富的操作，而且这些操作都是原子性的。在此基础上，redis 支持各种不同方式的排序。与 Memcached 一样，为了保证效率，数据都是缓存在内存中。区别是 redis 会周期性地把更新的数据写入磁盘或者把修改操作写入追加的记录文件，并且

在此基础上实现了 master-slave（主从）同步，对于 Memcached 来说其是没有数据源的，一般情况下如果缓存时间过期，必须从数据库获取数据。

redis 可以从官网 <http://redis.io> 下载最新版本，其支持 Windows 版本和 Linux 版本。Windows 下的 redis 无需安装，直接下载可执行文件即可，下载地址 <http://code.google.com/p/servicestack/wiki/RedisWindowsDownload>，Windows 版本的 redis 软件 bin 目录下有如图 9-2 所示的可执行文件。



图 9-2 redis 应用程序列表

如图 9-2 中的 redis-server.exe 为 redis 运行服务端，相当于 Memcached 中的 memcached 和 MongoDB 中的 mongod 工具。启动运行方式很简单，只需要单击 redis-server.exe 运行即可。

Linux 下的安装需要编译安装。下载最新版本的 Linux redis，下载地址 <http://redis.io/download>，也可使用如下指令下载：

```
wget http://redis.googlecode.com/files/redis-2.6.8.tar.gz
```

下载成功后解压 redis-2.6.8.tar.gz 文件。

```
tar -zxvf redis-2.6.8.tar.gz
```

进入 redis-2.6.8 文件夹编译安装，执行如下指令：

```
make
make install
```

如果成功安装 redis 工具，在 /usr/local/bin/ 系统路径下会安装放置 redis 的工具，如图 9-3 所示。

图 9-3 Linux 系统下 redis 应用工具列表

这些工具和 Windows 下的工具的功能一致，具体使用方法，以及作用可以前往官网查看官方文档。

运行 redis-server 启动 redis 服务。

现在有了 redis 的 server，那么接下来我们就需要了解一下 Node.js 的 redis 驱动模块。在 github 上有 redis 的 NPM 模块 connect-redis。

```
npm install connect-redis
```

执行上面的指令安装 connect-redis。

本系统只是为了使用 redis 来存储管理 express 的 Session，因此这里只需要使用官网提供的代码，代码如下：

```
var express = require('express')
    , RedisStore = require('connect-redis')(express);
```

在 express 中使用该 redis 时，需要在 express 框架配置中添加如下代码：

```
app.use(express.session({ secret: "keyboard cat", store: new
RedisStore(options) }));
```

其中的 app 为 express.createServer()返回的对象。

上面代码中 RedisStore 对象有一个 options 方法，该方法主要包含的字段，以及各自段的功能说明如下：

- 1. **client** An existing redis client object you normally get from redis.createClient()
- 2. **host** Redis server hostname
- 3. **port** Redis server portno
- 4. **ttd** Redis session TTL in seconds
- 5. **db** Database index to use
- 6. **pass** Password for Redis authentication
- 7. **prefix** Key prefix defaulting to "sess:"
- 8. ... Remaining options passed to the redis createClient() method.

redis 的主要功能是用来缓存数据，其提供了 Python、Ruby、Erlang 和 PHP 客户端驱动连接，在缓存功能方面和 Memcached 相似。

9.1.3 功能模块设计

在编码之前，我们首先需要对整个系统的架构做一个分析。对于框架的应用，不需要详细到每个功能的细节，例如数据库连接、路由实现等，我们只需要关注 Model 层实现、Controller 层实现，以及需要开发的 View 页面即可。接下来从 3 个层来分析我们需要实现的接口和页面。

本系统只涉及一个表，因此只需要实现一个 user_model 模块类继承基类 model 即可。user_model 主要负责用户个人信息字段的增、删、改、查功能，以及用户登录、用户邮箱验证激活等功能。根据以上分析，我们可以设计出 UserModel 这个类所需的接口，如表 9.2 所示。

表 9.2 UserModel接口设计表

模 块 层	文 件 名	类 名	接 口 名	接 口 说 明
Model 层	user_model.js	UserModel	getUserByUsername	根据用户名获取用户信息
			checkUser	验证用户登录
			addNewUser	用户注册
			updateUser	修改用户信息
			searchUser	用户查询
			confirmEmailCode	用户邮箱验证

逻辑层需要两个类 `index_controller` 和 `person_controller`，`index` 负责系统的首页显示，以及登录注册功能，`person` 负责用户的个人信息修改。根据上面的分析，可以设计出两个类的接口，如表 9.3 所示。

表 9.3 Controller层模块接口设计表

模块层	文件名	类名	接口名	接口说明
Controller 层	index_controller.js	IndexController	loginPageAct	用户登录页面
			toMainPageAct	登录首页
			loginAct	登录接口
			loginOutAct	登出接口
			signUpAct	注册接口
			signUpPageAct	注册页面
			emailConfirmAct	邮箱激活接口
	person_controller.js	PersonController	emailPageAct	邮箱验证页面
			toMainPageAct	用户个人主页
			changeInfoPageAct	修改用户个人信息页面
			changeInfoAct	修改用户个人信息接口
			changeAtvatarPageAct	修改用户头像页面
			changeAtvatarAct	修改用户头像接口

`view` 层有多个页面展示，包括登录页面、注册页面、邮箱验证页面、用户个人信息页面和聊天主页。根据上面的分析，我们将开发如表 9.4 所示页面信息。

表 9.4 View层jade文件设计说明表

模块层	文件名	文件描述	对应逻辑层接口	主要作用
View 层	main_view.jade	首页展示	loginPageAct	
	register.jade	用户注册	signUpPageAct	数据 submit 接口为 <code>signUpAct</code>
	change_atvatar.jade	头像修改页面	changeAtvatarPageAct	数据 submit 接口为 <code>changeAtvatarAct</code>
	change_msg.jade	个人信息	changeInfoPageAct	数据 submit 接口为 <code>changeInfoAct</code>
	login_index.jade	登录首页	toMainPageAct	
	email_page.jade	邮箱验证页面	emailPageAct	数据 submit 接口为 <code>emailConfirmAct</code>

通过分析 `Model` 层、`Controller` 层和 `View` 层的文件逻辑接口的实现，从一个整体的架构上对本系统进行了一个充分的分析和设计，接下来我们只需要编码实现每一个接口和 `jade` 页面即可。

从上面我们可以学习到作为软件开发员，不能局限自己为一个编码的工作者，更多的时候我们应该考虑到设计。请记住一句话：写任何代码之前都要注重设计，大部分问题都不该出现在编码期间，而应该在编码之前就解决。

充分分析完整个实现细节，接下来我们从 `Model` 层到 `View` 层开始编码实现系统功能。

9.2 系统的编码开发

通过上一节的系统分析,已经清楚了每个模块需要实现的类,以及接口逻辑,接下来我们从 Model 层逻辑实现、Controller 层逻辑实现和 View jade 页面模板 3 个部分的实现来介绍 MyWeb 应用的开发。重点需要了解如何实现 Model 层、Controller 层代码继承框架代码,并应用框架提供的 API 接口。

9.2.1 Model 层

上一节中已经明确了该模块需要实现的接口,下面代码是根据 9.1 节中的表格设计的一个未实现的类。代码如下:

```
function UserModel(){
  _self = this;
  /* 根据用户名获取用户信息 */
  this.getUserByUsername = function(username, callBack){
  };
  /* 根据用户名用户密码验证用户是否存在 */
  this.checkUser = function(username, password, callBack){
  };
  /* 新增用户信息 */
  this.addNewUser = function(userParameters, callBack){
  };
  /* 修改用户信息 */
  this.updateUser = function(uid, userParameters, callBack){
  };
  /* 删除用户信息 */
  this.deleteUser = function(uid, callBack){
  };
  /* 搜索用户 */
  this.searchUser = function(userName, callBack){
  };
  /* 邮箱验证 */
  this.confirmEmailCode = function(verificationCode, callBack){
  };
  /* 验证邮箱是否已经存在 */
  this.searchExistEmail = function(userParameters, callBack){
  }
}
```

【代码说明】

- `_self = this`: 保存 `this` 指针对象。
- `this.getUserByUsername`: 根据用户名获取用户信息接口。
- `this.deleteUser`: 删除用户信息。
- `this.searchUser`: 查询用户信息。
- `this.confirmEmailCode`: 邮箱激活。
- `this.searchExistEmail`: 验证邮箱是否已经被使用。

由于这里 MySQL 基类的操作都是异步的，因此在 Model 层所有的函数的参数中都添加了一个 `callBack` 回调函数来获取异步的操作结果。使用 `_self` 来保存 `this` 指针对象，因为在类的函数实现过程中 `this` 的指针会改变。

在介绍每个函数接口的实现前，我们先了解一下如何实现 `UserModel` 继承 `BaseModel` 基类。主要是应用 JavaScript 的 `prototype` 原型继承，实现代码如下：

```
UserModel.prototype = new BaseModel("t_user");
global.UserModel = UserModel;
```

接下来我们看一下每个接口的具体实现。

```
this.getUserByUsername= function(username, callBack){
    var whereArea = "f_uname = '" + username + "'",
        result = this.query(whereArea,
false,function(result){callBack(result)});
    return result;
};
```

应用父类的 `this.query` 查询条件 `f_uname` 为 `username` 的所有用户，`whereArea` 为查询条件字符串。代码如下：

```
/*
 *判断用户是否存在
 * 变量: username, values 需要更新的数组 json 格式, 需要更新的表名
 * 返回: 更新成功返回 id 的值, 更新失败返回 false
 */
this.checkUser = function(username, password, callBack){
    var whereArea = "f_uname = '" + username + "' and f_password = '" +
password + "'",
        result = this.query(whereArea,
false,function(result){callBack(result)});
    return result;
};
```

同样，应用父类的 `this.query` 条件查询来验证用户的用户名和密码是否匹配，如果存在，就返回查询后的用户信息。代码如下：

```
/*
 *添加新用户
 * 变量: userParameters: json 数组
 * 返回: 添加成功返回 id 的值, 更新失败返回 false
 */
this.addNewUser = function(userParameters, callBack){
    _self.searchExistEmail(userParameters,function(result){
        if(result == 1){
            callBack({code:-2, msg:"exist
user",data:userParameters});
        } else if(result == 2){
            callBack({code:-1, msg:"exist
email",data:userParameters});
        } else {
            _self.add(userParameters,function(result){callBack({code:0,
msg:"sucess",data:{'result':result}})});
        }
    });
    return;
};
```


向数据库插入一条新的记录。在插入数据前我们进行了一个邮箱验证，确保该用户邮箱没有被应用过，如果没有应用我们才将其插入数据库，否则返回错误提示。该接口是调用父类的 `add` 方法，向数据库插入数据。代码如下：

```
/*
 * 修改用户资料
 * 变量: id 和需要更新的 json 数据
 * 返回: 更新成功返回 true, 更新失败返回 false
 */
this.updateUser = function(uid, userParameters, callBack){
    var result = this.update(uid, userParameters,
function(result){callBack(result);});
    return result;
};
```

该 `updateUser` 接口应用父类的 `update` 接口实现数据的更新。代码如下：

```
this.confirmEmailCode = function(verificationCode, callBack){
    var whereArea = "f_email = '" + verificationCode + "'",
        result = this.query(whereArea, false,
function(result){callBack(result);});
    return result;
}
```

应用父类 `this.query` 接口查询用户邮箱和验证码是否匹配，如果匹配返回用户信息，否则返回 `false`。代码如下：

```
this.searchExistEmail = function(userParameters, callBack){
    var whereArea = "f_email = '" + userParameters.f_email.split("_")[1]
+ "' or f_uname='" + userParameters.f_uname + "'",
        result = this.query(whereArea, false, function(result){
            if(result.length > 0){
                if(result[0].f_uname == userParameters.f_uname){
                    callBack(1);
                } else {
                    callBack(2);
                }
            } else {
                callBack(false);
            }
        });
    return result;
}
```

应用父类 `this.query` 接口验证用户邮箱是否已经被注册，如果已经被注册，返回 1 或者 2，如果没有被注册返回 `false`。

以上接口的实现都非常简单，主要是应用基类的接口，如 `add`、`update`、`delete` 和 `query` 等方法。主要掌握调用基类接口的方法。到此为止，我们将 9.1 节中提供的 API 全部实现了，接下来我们实现 `controller` 层的逻辑。

说明：这些接口的实现都略微粗糙，由于时间原因没有将其改进，希望读者谅解。

9.2.2 Controller 层

`Controller` 层有两个需要实现的逻辑类 `IndexController` 和 `PersonController`。根据 9.1 节

分析的类接口，设计出两个逻辑类的基本代码架构如下所示。

```
function IndexController() {
  var _parent = Object.getPrototypeOf(this);
  var _self=this;
  this._obj = new UserModel();
  this.loginPageAct = function(){
  };
  this.toMainPageAct = function(){
  };
  this.loginAct = function(){
  };
  this.loginOutAct = function(){
  }
  this.signUpAct = function(){
  }
  this.signUpPageAct = function(){
  }
  this.emailPageAct = function(){
  }
  this.emailConfirmAct = function(){
  }
};
```

【代码说明】

- ❑ `_parent = Object.getPrototypeOf(this)`: 获取父类操作对象；
- ❑ `this._obj = new UserModel()`: 获取 user 的 model 对象；
- ❑ `this.loginPageAct`: 登录显示页面；
- ❑ `this.toMainPageAct`: 登录主页信息显示；
- ❑ `this.loginAct`: 登录操作接口；
- ❑ `this.loginOutAct`: 页面登出接口；
- ❑ `this.signUpAct`: 页面注册接口；
- ❑ `this.signUpPageAct`: 注册页面展示；
- ❑ `this.emailPageAct`: 邮箱验证页面；
- ❑ `this.emailConfirmAct`: 邮箱确认接口。

对于一个操作逻辑来说，一般都会包含两个操作接口，页面展示接口和逻辑处理接口。因此本接口中的登录、注册和邮箱验证都包含了两个接口，例如登录包含了 `loginPageAct` 和 `loginAct`。代码如下：

```
this.loginPageAct = function(){
  if(_parent._req.session.nickName){
    _parent.displayJade(VIEW
'chat',{username:_parent._req.session.nickName});
  } else {
    _parent.displayJade(VIEW + 'main_view');
  }
};
```

【代码说明】

- ❑ `_parent._req.session.nickName`: 判断用户是否登录成功，是否有 session 存在；
- ❑ `_parent.displayJade(VIEW + 'chat'...)`: 进入展示 chat 页面；
- ❑ `_parent.displayJade(VIEW + 'main_view')`: 进入登录页面。

本段代码中展示了 express 框架中的 Session 存储方法。express 框架中的 Session 值保存在 HTTP 的 req 请求参数中,通过设置 req 中的 Session 值来保存系统的 Session 数据,在用户登录接口中介绍如何设置 Session。代码如下:

```
this.loginAct = function(){
    var loginJson = _parent._req.body.v;

    _self._obj.checkUser(loginJson.username,loginJson.password,function(res
ult){
        if(result.length > 0){
            if(result[0].f_state != 1){
                _parent._req.session.userState = true;
                _parent.returnError("1", "success, but not email confirm",
                "");
                return;
            }
            _parent._req.session.username = loginJson.username;
            _parent.returnError("0", "success", "");
            return;
        }
        _parent.returnError("-1", "no user", "");
        return;
    });
};
```

【代码说明】

- `_self._obj.checkUser`: 验证用户密码和账号是否匹配;
- `result.length > 0`: 判断是否存在该用户名和密码信息;
- `_parent._req.session.username`: 登录成功时设置用户 Session。

应用 UserModel 类中的 checkUser 接口验证用户是否已登录,因为在类中定义了其私有变量 `_obj` 为 UserModel 的对象,因此其拥有 UserModel 对象的方法 checkUser。上一段接口代码中学习到 express 框架的 session 设置方法,本段代码学习 express 框架中的 post 参数获取方法 `req.body.v`。

用户登录验证成功后,再判断用户是否为激活状态。如果激活就应用 `_parent._req.session.username` 设置 Session 数据,验证用户名和密码失败后,则返回-1 错误码。

其他 controller 接口实现方式这里就不详细介绍了。主要看一下 controller 层中的用户注册邮件发送功能,该接口为 `signUpAct`,在 `index_controller.js` 中。

首先需要 require nodemailer 这个 NPM 模块,如果没有安装可以使用 `npm install nodemailer` 进行安装。下面一段代码中使用的是 qq 邮箱发送邮件(也可以使用 126 或者 163 邮箱),port 默认为 25 端口,如果需要做账户验证的情况下,需要将 `use_authentication` 设置为 true,如果需要账号验证, user 为发件人邮箱, pass 为该邮箱密码,配置完成后就可以使用 mail 这个对象来发送邮件了。邮件配置信息如下代码:

```
var signJson = _parent._req.body.v,
    mail = require('nodemailer');
mail.SMTP = {
    host: 'smtp.qq.com',
    port: 25,
    use_authentication: true,
    user: '492383469@qq.com',
```

```
pass: '*****'
};
```

mail 对象提供了一个 send_mail 接口，用来发送邮件。该接口有多个配置参数，下面代码中使用了 sender（发送者邮件地址）、to（发送的目的地址）、subject（发送的邮件主题）和 html（发送邮件正文 html 页面），邮件发送接口中的回调函数来返回邮件发送是否成功，具体代码如下：

```
mail.send_mail(
{
    sender: '492383469@qq.com', //发送邮件的地址
    to : emailAddress, //发给谁
    subject: Email_Config.getSubject(), //主题
    html: Email_Config.getHtml(resetJson.f_email) //如果要
发送 html
},
//回调函数，用户判断发送是否成功，如果失败，输出失败原因
function(error, success) {
    if (!error) {
        console.log('message success');
    } else {
        _parent.returnError("-2", "can not send email",
""");
        return;
    }
});
```

index_controller.js 的其他接口实现方式大致相同，大家可以参考源码来进一步完善该 controller 层逻辑的实现。接下来介绍的是 person_controller.js 接口实现，其中大部分接口实现和 index_controller.js 相同，这里重点介绍几个重要的逻辑实现。

根据 9.1 节中的 person_controller.js 接口分析，我们首先编写一个接口架构，代码如下：

```
function PersonController() {
    var _parent = Object.getPrototypeOf(this);
    var _self = this;
    this._obj = new UserModel();
    this.changeInfoPageAct = function() {
    };
    this.changeInfoAct = function() {
    };
    this.changeAtvatarPageAct = function() {
    };
    this.changeAtvatarAct = function() {
    };
    this.loginOutAct = function() {
    }
}
```

【代码说明】

- ☐ this.changeInfoPageAct: 用户个人信息修改页面；
- ☐ this.changeInfoAct: 用户个人信息修改接口；
- ☐ this.changeAtvatarPageAct: 用户头像修改页面；
- ☐ this.changeAtvatarAct: 用户头像修改接口。

该 controller 模块 _parent、_obj 对象获取方式和 index_controller.js 接口是一样的，这里

重点介绍 `changeAtvatarAct` 实现。该接口涉及第3章介绍的文件上传功能，以及 `express` 框架中获取文件数据的方式。

参数初始化以及 `express` 框架文件数据获取方式，如下代码所示。

```
var imageData = _parent._req.files["atvatar_image"],
    tmp_path = imageData.path,
    username = _parent._req.session.username,
    timestamp = Date.parse(new Date()),
    imageType = checkImageData(imageData["name"]);
if (!imageType) {
    _parent.returnError("-1", "type is invaild", "");
    return;
}
var target_path = "/public/images/atvatar/" + timestamp + "_" +
    username + "." + imageType;
```

【代码说明】

- ❑ `imageData = _parent._req.files["atvatar_image"]`: `express` 框架获取文件上传数据方式，`atvatar_image` 为文件上传 `input` 的 `name` 值；
- ❑ `_parent._req.session.username`: 获取当前登录用户名；
- ❑ `timestamp = Date.parse(new Date())`: 获取当前时间戳；
- ❑ `checkImageData(imageData["name"])`: 验证图片格式；
- ❑ `target_path`: 根据当前时间戳重命名上传文件，避免文件重复被覆盖。

`Express` 框架获取文件数据的方式是通过 `_parent._req.files[input name]` 来获取，相对第3章中的方法就显得非常简单。接下来我们看一下如何将缓存文件数据保存到指定目录 `target_path` 中。

第3章中介绍了两种方法，应用 `fs.readFile` 读取文件数据并将读取的数据写到指定保存文件路径下和直接将缓存数据 `fs.rename` 保存到指定目录下。在 `Windows` 下使用第二种方法的时候，缓存数据是保存在 `Windows` 的系统安装 `C` 盘，如果指定文件目录不在 `C` 盘时会出错。`Windows` 不允许扩盘操作文件数据，由于本系统是在 `Windows` 下运行，因此我们选择了第一种方式。下面代码提供了两种代码实现方法，如果非 `Windows` 下时建议使用 `fs.rename` 方法。代码如下：

```
/*Module.fs.rename(tmp_path, target_path, function(err) {
    if (err) throw err;
    // 删除临时文件夹文件,
    Module.fs.unlink(tmp_path, function() {
        if (err) throw err;
        _self._res.send('File uploaded to: ' + target_path + ' - ' +
            imageData.size + ' bytes');
    });
});*/

Module.fs.readFile(tmp_path, function(error, data){
    Module.fs.writeFile(BASE_DIR + target_path, data, function (err)
    {
        if (err) throw err;
        _self._obj.updateUser(_parent._req.session.uid,
        {'f_atvatar':target_path}, function(updateResult){
            if(updateResult){
                _self._req.session.atvatarUrl = target_path;
                _parent.redirectUrl("/person?c=toMainPage");
            }
        });
    });
});
```



```

        return;
    } else {
        _parent.returnError("-1", "upload failed", "");
        return;
    }
    });
});
});
});

```

【代码说明】

- ❑ `fs.rename`: 将缓存文件保存一份数据到指定目录, 然后通过 `unlink` 删除源文件;
- ❑ `fs.readFile`: 读取文件数据, 回调函数返回读取的数据;
- ❑ `fs.writeFile`: 将数据写入文件中, 应用 `readFile` 读取返回的数据写入到 `writeFile` 中。

使用文件读写数据方式主要是为了解决 Windows 下的扩盘移动文件, 在实际项目中不推荐该方式。如果需要在 Windows 下运行项目, 最好的办法是保证项目也在 C 盘运行。

本节重点介绍了 Controller 层的两个逻辑类的重要接口的实现, 其中应用到了多个 `express` 框架的知识, `express` 参数获取方式、`express` 的 Session 管理和 `express` 文件上传获取方式等。为了更好地帮助大家了解 `express` 参数获取方法, 下面做一个简单的总结。

- ❑ 获取路径参数方法, 例如 `127.0.0.1:3000/index`, 为了得到 `index`, 可以通过使用 `req.params` 得到, 通过这种方法我们就可以很好地处理 Node.js 中的路由处理问题;
- ❑ 获取 GET 传值方法, 例如 `127.0.0.1:3000/index?id=12`, 通过使用 `req.query.id` 就可以获得客户端 GET 方式传递过来的值;
- ❑ 获取 POST 传值方法, 例如 `127.0.0.1: 300/index`, 然后 POST 了一个 `id=2` 的值, 可以通过 `req.body.id` 获取 POST 传递的值。

参考示例代码如下:

```

var app = require('express').createServer();

app.get('/:key', function(req, res){
    console.log(req.params.key); //输出 index
    console.log(req.query.id); //输出表单 get 提交的 id
    res.send('great you are right for get method!'); //显示页面文字信息
});

app.post('/:key', function(req, res){
    console.log(req.params.key); //输出 index
    console.log(req.body.id); //输出表单 post 提交的 id
    res.send('great you are right for post method!'); //显示页面文字信息
}); app.listen(3000);

```

9.2.3 View 层

本框架应用的是 `jade` 模板, 因此在设计页面前, 我们首先定义好页面的 `header.jade` 和 `footer.jade`。 `header.jade` 为头文件, 包含大部分页面的 logo 处理, 以及常用 JavaScript 和 css 库引入管理, `footer.jade` 则负责页面的底部信息。下面是 `header.jade` 的代码:

```

div#header
  div.main_content_header
    a(href='/person?c=toMainPage', style="float:left;margin-left:
150px;width: 160px;")

```

```

img(src="public/images/MyChat.png")
div.user_info
  div#user_head
    a(href='/person?c=changeAtvatarPage')
      img(src='#{atvatarUrl}')
    #personBoxMsg
    a(href='/person?c=toMainPage')    #{nickName}
    ul
      li
        a(href='/person?c=changeInfoPage')    个人设置
      li
        a(href='/index')    系统消息
      li
        a(href='/person?c=loginOut')    注销

```

【代码说明】

- ❑ div.main_content_header: 头部信息模块;
- ❑ a(href='/person?c=toMainPage'...: logo 连接标签 a;
- ❑ img(src="public/images/MyChat.png"): logo 的 image 图片标签;
- ❑ div.user_info: 用户个人信息模块。

jade 模板使用时要严格注意空格对齐, 了解 jade 模板的一些常用 html 标签转化就可以了, View 页面的知识点不是很多。下面是 footer.jade 的代码:

```

div#show_image
  <!--img(src='public/images/show_user.jpg')-->
div#footer
  ul
    li: a(href='#') 帮助
    li: a(href='#') 联系我们
    li: a(href='http://nodejs.org') nodejs 官网
    li &copy; Copyright 2012 by danhuang

```

【代码说明】

- ❑ div#footer: footer 底部模块;
- ❑ li: a: 底部列表数据展示, 包括帮助、联系我们、Node.js 官网和©。

那么如何在其他页面应用 header.jade 和 footer.jade 呢? 接下来我们看一下 main_view.jade 中是如何应用的, 代码如下:

```

include common/header
link(rel='stylesheet', type='text/css', href='/css/main_view.css')
div.main_content
  div.login_form
    form
      ul
        li 用户名
          input(type="text",class='input_text',size='25',value='
用户名')
        li
          密<span class='blank'>&nbsp;&nbsp;&nbsp;</span>码
          input(type='password',class='input_text',size='25')
        div.description
          记住密码
          <input type='checkbox' />
          a(href='javascript:void(0)')
            忘记密码

```



```

        li.operation
          a(href='javascript:void(0)',class='button submit') 登录
        li
          .errorMsg
            账号和密码不匹配
      include common/footer

```

【代码说明】

❑ include common/header: 包含头文件 header.jade。

❑ include common/footer: 包含底部文件 footer.jade。

jade 模板中如果需要包含其他 jade 模板时,只需要使用关键字 include,可以省略“.jade”后缀名,需要注意,这里 include 的文件路径是相对路径,相对的是当前 jade 文件路径。

View 层的代码没有太多的学习知识点,主要是 jade 模板的应用。这里就不再详细介绍,如果希望了解更多 jade 模板的应用可以参考本应用的源码。其他 view 页面还包括 change_atvatar.jade、change_msg.jade、email_page.jade、login_index.jade、main_view.jade 和 register.jade。

9.3 项目总结

本章主要是基于 MyWeb 框架的应用,实现一个用户登录管理系统,主要是让读者能够了解本框架的应用。接下来我们会进一步让大家了解启动运行项目,以及项目的前端体验。最后基于本项目的开发过程做一个简单的总结。

学习本章需要进一步熟悉 forever 模块的应用,要掌握本章中介绍的项目开发过程以及框架应用方法,了解本应用的交互设计思想。

9.3.1 forever 启动运行项目

为什么要使用 forever 工具运行项目?

首先我们必须知道使用 node xxx.js 运行项目,当遇到异常时会导致运行中断,必须重新启动服务器;其次服务器运行过程中的日志信息没办法查询,错误日志、流水日志和 debug 日志都混杂在一起,即使项目运行出错,也无从查询;另外, node xxx.js 运行项目无法使用守护进程进行运行,必须保留运行窗口,但运行窗口关闭进程也关闭了。

forever 工具就很好地解决了上面的几个问题。它是一个命令行工具,可以让 node 服务器以守护进程方式运行,其可以指定错误日志、流水日志和 debug 日志存放文件。

其安装配置方法这里在介绍一下,使用 NPM 安装,执行如下指令:

```
npm install forever -g
```

NPM 一章中已经详细地介绍了 forever 这个模块的一些参数应用。现在我们使用 forever 来运行聊天室项目。我们希望 forever 运行的 log 放在 forever.log 中,debug 日志放在 debug.log 中,错误日志放在 error.log 中,日志使用追写文件方式。根据上面的目的,我们可以将其

转化为 forever，运行指令如下：

```
forever start -a -l forever.log -o debug.log -e error.log app.js
```

使用 forever 时可能会出现如图 9-4 所示的异常问题。

```
fs.js:338
  return binding.open(pathModule._makeLong(path), stringToFlags(flags), mode);
                    ^
Error: ENOENT, no such file or directory 'D:\root\forever\forever.log'
    at Object.fs.openSync (fs.js:338:18)
    at Object.forever.startDaemon (C:\Users\Administrator\AppData\Roaming\npm\node_modules\forever\lib\forever.js:391:14)
    at app.cmd.cli.startDaemon (C:\Users\Administrator\AppData\Roaming\npm\node_modules\forever\lib\forever\cli.js:229:13)
    at C:\Users\Administrator\AppData\Roaming\npm\node_modules\forever\lib\forever\cli.js:134:5
    at forever.stat (C:\Users\Administrator\AppData\Roaming\npm\node_modules\forever\lib\forever.js:340:24)
    at Object.oncomplete (fs.js:297:15)
```

图 9-4 forever 启动异常错误信息

主要原因是需要在 D 盘创建一个 root 文件夹来存放运行日志，那么我们就手动创建后重新运行。创建成功后，由于我们还需要使用 redis，以及 MySQL 数据库，因此我们先启动 redis-server.exe 和 MySQL 数据库，再运行如上指令。运行结束后可以看到如图 9-5 所示的信息，表明已经成功启动 Node.js 项目。

```
D:\webStorm\web_chating>forever start -a -l forever.log -o debug.log -e error.log app.js
forever: Forever processing file: app.js
```

图 9-5 forever 正常启动运行日志信息

成功运行项目后，我们再看一下其中的运行日志 forever.log、debug.log 和 error.log。debug.log 部分信息如图 9-6 所示。

```
info: socket.io started
app listening on http://127.0.0.1: 3000
info: socket.io started
app listening on http://127.0.0.1: 3000
[key:index] [class:IndexController] [controller:loginPageAct]
[key:index] [class:IndexController] [controller:loginPageAct]
[key:index] [class:IndexController] [controller:loginPageAct]
[key:index] [class:IndexController] [controller:loginPageAct]
```

图 9-6 forever debug 日志信息

forever.log 部分信息如图 9-7 所示。

其中，方框中的数据提示了之前运行中没有启动 redis-server 的运行错误，启动成功后 forever.log 也记录了 debug 日志信息，其应该包含了所有的服务器运行日志信息。

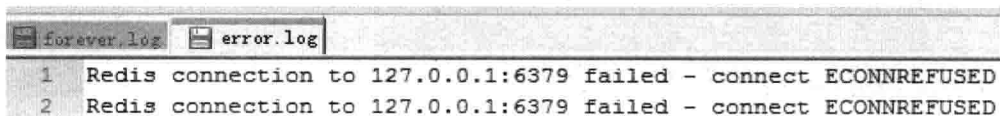
error.log 部分信息如图 9-8 所示。

```

info: socket.io started
app listening on http://127.0.0.1: 3000
Forever detected script exited with code: -1073741510
Forever restarting script for 1 time
Forever detected script exited with code: -1073741510
Forever restarting script for 2 time
info: socket.io started
app listening on http://127.0.0.1: 3000
Redis connection to 127.0.0.1:6379 failed - connect ECONNREFUSED
Redis connection to 127.0.0.1:6379 failed - connect ECONNREFUSED
[key:index] [class:IndexController] [controller:loginPageAct]

```

图 9-7 forever forever.log 日志信息



```

1 Redis connection to 127.0.0.1:6379 failed - connect ECONNREFUSED
2 Redis connection to 127.0.0.1:6379 failed - connect ECONNREFUSED

```

图 9-8 forever error.log 日志信息

如图 9-8 所示的信息所示没有启动 redis-server。

这样即使服务器遇到异常错误，也不会中断程序运行，而是会报相应的 error 信息，利用该方法我们就可以保证项目正常运行。

9.3.2 系统应用体验

本节主要介绍一下本系统的一些交互设计，主要通过本节介绍更清晰地了解本项目的一些应用。本部分会依次介绍本系统的登录页面、注册页面、邮箱验证页面、用户头像和个人信息修改页面。

如图 9-9 所示是本系统的登录页面，其中标号 1 部分为相应的 header.jade 模块页面内容，标号 2 部分的“帮助”和“联系我们”则为 footer.jade 模块页面内容。标号 3 部分为注册页面的 jade 模块页面内容。



图 9-9 系统登录页面

注册页面和登录页面类似，只是在设计上将 form 表单的数据进行了更换，如图 9-10 所示。

图 9-10 系统注册页面

相应地在注册页面我们输入数据，账号为 danhuang，昵称为丹华，邮箱为 492383469@qq.com，密码 123456，输入完成后提交信息。如果信息正确会跳转到如图 9-11 所示页面，表示用户已经成功注册，只需要登录邮箱获取验证码就可以激活用户。



图 9-11 系统注册成功后页面信息

笔者登录 492383469@qq.com 邮箱，可以看到如下注册激活验证邮件。这里需要说明的是，源码中使用的是笔者的邮箱账号和密码，但是为了个人安全笔者没有写密码，大家需要修改 index_controller.js 中的第 120 行的 QQ 账号和第 121 行的 QQ 密码。

node聊天室欢迎您 ☆

发件人: 黄丹华 <492383469@qq.com> 国

时 间: 2013年1月16日(星期三) 晚上10:30

收件人: 492383469 <492383469@qq.com>

大 小: 1K

打印 | 显示邮件原文 | 导出为eml文件 | 邮件有乱码? | 转发到群邮件 | 保存到记事本 | 添加到日历 | 作为附件转发

欢迎注册node聊天室，请点击以下链接完成注册http://127.0.0.1:3000/index?c=emailConfirm&m=XKNF_492383469@qq.com

图 9-12 验证邮件信息

从图 9-12 中可以看到,发件人是在代码中设置的用户,收件人为刚才注册填写的邮件,邮件内容为一个邮箱激活链接。接下来复制 url 链接,进行激活认证。

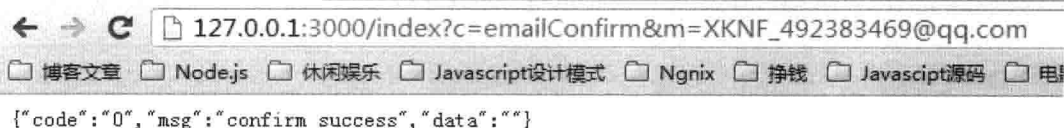


图 9-13 验证邮件返回结果

运行后可以看到一个 json 数据返回,由于这部分没有添加页面,如果出现如图 9-13 所示 json 字符串代表已经成功激活。那么接下来我们再登录系统。

登录系统后跳转到 <http://127.0.0.1:3000/person?c=toMainPage>,可以看到如图 9-14 所示的页面。中间部分为一个可以翻动的菜单导航功能。这部分功能会应用到 jquery 的一个书本翻页效果插件,如果大家有兴趣可以去了解,插件叫做 jquery.trunpage。

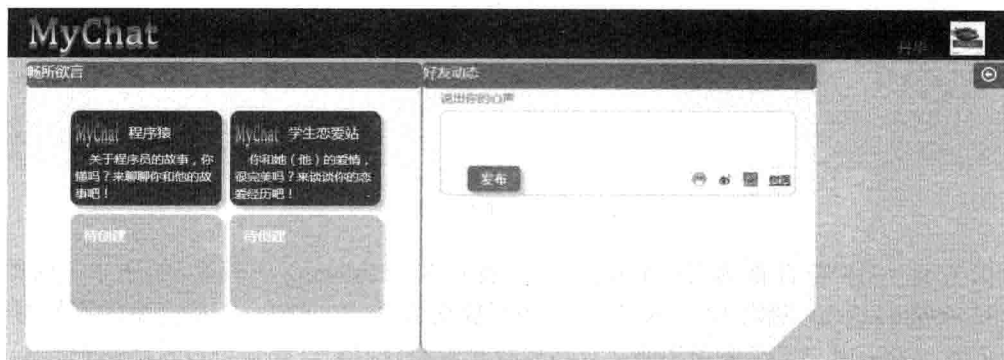


图 9-14 登录首页

右侧为在线用户列表,可以点开箭头查看在线用户,单击开后如图 9-15 所示的效果。这部分功能主要是参考了腾讯 QQ 空间的右侧隐藏栏功能。

其中的“丹华”为系统注册时候的昵称,单击昵称可以修改用户信息和退出登录,单击用户头像可以进入头像修改页面。下面我们单击进入头像修改页面,这部分会涉及一个笔者个人开发的插件——本地图片预览功能。

单击头像进入头像修改页面,本页面的图片预览上传功能并非 flash 插件,该功能是利用 JavaScript 的前端技术实现,有关本插件的应用可以在笔者的 CSDN 博客网站中了解《JavaScript 实现本地图片上传预览功能(兼容 IE、chrome、FF)》¹这篇文章。其中有示例代码,以及简单的插件介绍,其应用和效果体验还是非常不错的。

最后一个功能页面就是个人信息修改页面,此页面是展示一个用户信息的列表,修改提交即可。



图 9-15 用户列表导航栏

¹ 参见网站 <http://blog.csdn.net/danhuang2012/article/details/7703606>。

9.3.3 系统开发总结

本系统开发包含几个过程：系统分析、系统数据库设计、系统架构设计和系统编码等。本系统着重分析了数据库的设计和系统架构代码框架的实现。系统开发初期不要急于编码，而是重在架构和接口的设计。本系统的开发是面向接口编程，优势在于，前期对系统的分析非常充分，接口设计完成以后，其实已经编写好了程序，只是没有将其中的代码实现而已。

举个很简单的例子，高速公路在铺建之前，就已经计划好在建设中需要开凿多少个山洞，搭建多少个桥梁。软件工程也是一样，在一个系统实现之前，我们应该计划好这个系统实现需要多少个类，每个类的作用是什么，类中的接口有哪些等。

本系统在开发过程中应用到多个 Node.js 的开发知识和前端 JavaScript 插件应用，希望读者能够总结学习。接下来是本章的扩展阅读，主要是 MyWeb 框架 2.0 的介绍。

9.4 扩展阅读之 MyWeb 2.0 的介绍

MyWeb 1.0 框架主要还是基于 express 框架的一些应用的设计，在 MyWeb 1.0 框架中，主要还是复用 express 框架中的静态文件处理模块、Session 处理模块和路由服务器创建功能等。而在 MyWeb 2.0 中完全脱离了该 express 框架。在本书中有提到如何实现静态服务器模块、Session 处理模块，以及服务器路由设计处理方法等。MyWeb 2.0 充分应用了本书的两个模块：静态服务器模块和 Session 处理模块，而路由处理方法也从 MyWeb 1.0 的自带参数方法改为路径请求方法。

在 MyWeb 1.0 中路由的处理原则是：http://127.0.0.1/index?c=login，其中的 index 为 controller 文件名，login 则为该 controller 中逻辑处理函数。而 MyWeb 2.0 中的路由处理原则是：http://127.0.0.1/index/login，其中的 index 为 controller 文件名，login 则为该 controller。MyWeb 1.0 和 MyWeb 2.0 之间的入口文件以及框架设计方面都类似，主要的区别在路由处理方面。下面我们看一下路由处理方法与 MyWeb 1.0 之间的区别。

```
exports.router = function(res, req){
  var logInfo = {};
  // url 解码，避免 url 路径出现中文字符
  var pathname = decodeURI(lib.url.parse(req.url).pathname);
```

【代码说明】

❑ decodeURI：解码经过 HTTP 的 urlencode 后的字符串。

使用 decodeURI 来解析 HTTP 的 url，因为浏览器会自动地将中文或者空格等特殊字符进行转义，而服务器端则必须进行相应的解码。

```
// 初始化 http 参数获取模块
lib.httpParam.init(req, res);
```

上面代码主要是初始化我们实现的 httpParam 模块，该模块主要应用于 HTTP 中的 GET 和 POST 参数获取。

```
// 初始化 Session 管理模块
global.sessionLib = lib.session.start(res, req);
```

以上代码主要是本书中的 Session 模块数据初始化。

```
// 获取 http 请求路径, 使用斜杠获取请求的 controller 类名, 以及 action 方法
var pathArr = pathname.split('/');
// 弹出第一个空字符
pathArr.shift();
var model = pathArr.shift()
    , controller = pathArr.shift()
    , Class = '';
```

应用 split 来分割 HTTP 的 url 路径。HTTP 的 url 如果为 http://127.0.0.1/index/login 时, 则相应的请求路径为/index/login, 那么通过 split 分割该字符, 从而获取 controller 文件名和 controller 中对应的处理函数。代码如下:

```
// 添加日志信息
logInfo['pathname'] = pathname;
logInfo['model'] = model;
logInfo['controller'] = controller;

// 过滤 favicon 请求
if(pathname == FAVICON){
    return;
}else if(pathname == '/'){
    res.render(VIEW + 'index.jade');
    return;
}
```

将 favicon 请求过滤, 同时判断用户请求路径是否为空, 如果为空则显示 index.jade。代码如下:

```
if(!controller || !model){
    returnDefault(res, 'can not find source');
    return;
}
```

判断请求数据是否为空, 如果为空则执行 returnDefault 函数, 返回 404 错误信息。代码如下:

```
// 尝试 require 一个 controller 类名, 如果失败则认为是一个静态资源文件请求
try {
    Class = require(CON + model);
}
catch (err) {
    console.log(err);
    lib.staticModule.getStaticFile(pathname, res, req, BASE_DIR);
    return;
}
```

上面代码使用 try catch 来 require 一个 controller 文件, 如果无法找到该 controller 模块文件时, 则认为该请求 url 是一个静态资源请求。lib.staticModule.getStaticFile 为本书实现的一个静态文件模块, 应用该方法可以返回相应的静态资源文件内容, 如果未找到则会返回一个 404。这种设计原则, 必须禁止 controller 模块的文件名不能和静态文件夹同名。代码如下:

```
if(Class){
    var object = new Class(res, req);
```



```

try{
    object[controller].call();
} catch(err){
    console.log(err);
    returnDefault(res, 'no this action');
}
} else {
    returnDefault(res, 'can not find source');
}
}

```

如果成功 `require controller` 文件模块后, 则创建该对象类的实例, 并调用其逻辑处理函数。如果执行函数失败, 则调用 `returnDefault` 返回 404。

以上的路由处理模块相对 `MyWeb 1.0` 在设计方法上添加了更多的异常处理逻辑。由于没有应用 `express` 框架因此需要自己实现静态服务器、`Session` 管理和 `HTTP` 参数获取管理。当然其中和 `MyWeb 1.0` 的区别还很多, 在这里只详细地分析其路由处理的不同点。

两个框架对于开发者应用来说是一致的, 实现 `MyWeb 2.0` 的目的是希望能够通过 `Node.js` 的原生 `API` 实现一个框架, 而不依赖其他框架。能够学到更多关于 `Node.js` 的知识, 例如其中的 `Session` 设计、静态服务器的设计等。

在本书撰写过程中 `MyWeb 2.0` 框架源码还未在 `github` 上发布, 该代码在本书的源码第 9 章中。开发 `MyWeb 2.0` 框架的目的是希望大家能够通过框架来学到更多的 `Node.js` 知识, 并非要求大家使用 `MyWeb 2.0` 框架进行应用开发。

9.5 本章实践

1. 本书中有介绍 `Node.js` 与 `PHP` 实现微博功能, 现在需要读者能够通过本书的学习为 `MyChat` 应用新增发表和分享功能。

分析: 根据本书中介绍的 `PHP` 与 `Node.js` 微博分享功能实现, 读者自行完成, 相应代码可以参考源码。

2. 在本章的扩展阅读中介绍了 `MyWeb 2.0` 的一些信息, 请读者概述出 `MyWeb 1.0` 与 `MyWeb 2.0` 之间最大的区别。

分析: 两个框架都是基于 `Node.js` 的 `MVC` 框架代码, 并都包含了基本的 `HTTP` 服务功能。`MyWeb 1.0` 框架依赖开源框架 `express` 的实现, 在实现方法上更为简单。而 `MyWeb 2.0` 完全是基于 `Node.js` 源码进行开发的一个系统框架, 其中包含了更多的可供读者学习的知识点: `HTTP` 参数获取、静态服务器管理和路由处理实现等。

9.6 本章小结

本章的重点是应用 `MyWeb 1.0` 框架来实现一个简单的 `Web` 聊天室 `MyChat` 应用, 对于应用来说其实现较为简单, 但是需要读者注重其开发过程。一个应用的实现包含了前期的功能需求分析、系统功能的设计、编码, 以及系统的测试体验过程。希望通过本章的学习能够让读者进一步了解框架的实现细节, 同时对系统的开发有一个初步的认识了解。

第 10 章 Node.js 实例应用

本章主要介绍 Node.js 的一些实际应用，通过应用的系统设计分析，以及编码开发，来进一步熟悉 Node.js 的项目开发。本章主要介绍两个应用，分别是实时聊天对话和联网中国象棋游戏。实时聊天对话主要是介绍 Node.js 在实时 SNS 交友方面的应用，而联网中国象棋游戏则是从 Node.js 的联网游戏方面来介绍。

10.1 实时聊天对话

聊天室在很多年以前就已经不再新鲜，今天我们主要来了解使用 Node.js 如何来实现该应用。本节着重介绍的是系统的实现，并不着眼于其应用的创意性。因为篇幅原因主要介绍一个大概的实现方法，以及如何去扩展该应用，不会详细地实现一个完整的应用。

10.1.1 系统设计

本应用重点在于群聊机制，没有更深入地介绍其私聊等功能。群聊主要包含登录、房间选择和聊天 3 个功能模式。系统的交互设计流程如图 10-1 所示。

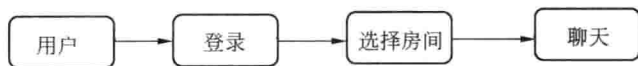


图 10-1 系统设计流程图

如图 10-1 所示为一个简单的用户操作流程，作为开发者我们应该从中了解到哪些信息呢？

- ❑ 有用户登录状态，那么就应该存在 Session 管理和控制；
- ❑ 涉及聊天功能，则相对应联想到 socket 机制；
- ❑ 存在房间选择问题，则对应会联想到 socket 推送方式问题。

以上 3 个问题是本应用实现前需要解决的问题，而这些解决方法也是本书前面几章中介绍到的。

1. 用户登录

用户登录可以应用本书中介绍的 Session 模块。该模块包含简单的 Session 管理功能，同时应用简单方便，将用户登录时的用户名使用 Session 存储起来。

2. 聊天功能

可以应用 socket 实现消息的推送，以及拉取功能。socket 的应用实例在本书中也有简

单的介绍，在本应用中将会详细介绍其如何发送，以及监听消息。

3. 多房间功能

小知识：当用户进入系统后，会发送一个 socket 连接请求，连接成功后，服务器会产生一个该用户私有的 socket 对象，而该对象就可以给用户推送特定的消息，以及其他数据。因此当用户连接 socket 时，我们就要保存用户 socket 对象，以便推送公有和私有消息。

对于没有房间的应用来说，socket 只需要推送给所有的 Session 用户，但由于本系统存在房间机制，因此在存储 Session 用户的 socket 时，需要按用户房间类型进行保存。

例如，用户名 danhuang 进入房间 ID 为 1001 的聊天室时，系统会将 danhuang 用户名连接的 socket 实例对象进行保存，假设所有的用户 socket 都保存在全局对象 userlist 中，则 danhuang 用户保存方式为 userlist[1001][danhuang]=[socket object]。当某个用户在 1001 房间发送消息时，则对应地推送消息到 userlist[1001]数组的所有对象上。

分析完本系统的 3 个问题后，我们就来看看本系统应用的框架，以及其中的模块设计方法。

10.1.2 系统的模块设计

本系统应用的框架是一个 MVC 框架，本框架包含了基本的路由处理，以及基本项目架构设计，下面有关于本系统框架的详细介绍。本系统只有一个 login 逻辑处理层，该 controller 包含了本系统的 3 个功能：用户登录、socket 服务和聊天功能。

本系统应用的是 MyWeb 框架的 2.0 版本，该框架摒弃了 express 框架，使用 Node.js 原生 API 实现的一套 MVC 框架，在第 9 章的扩展阅读中有介绍到其与 MyWeb 1.0 框架之间的区别。

该 login 模块的函数结构如下代码：

```
module.exports = function(){  
  
    var _res = arguments[0];  
    var _req = arguments[1];  
  
    this.checkSession = function(model){  
    }  
  
    this.login = function(){  
    }  
  
    this.enterRoom = function(){  
    }  
  
    this.say = function(){  
    }  
}
```

【代码说明】

- ❑ this.checkSession：验证用户是否已登录；
- ❑ this.login：用户登录验证接口；
- ❑ this.enterRoom：用户进入房间接口；

□ `this.say`: 用户聊天对话接口。

该 `login` 模块中的 3 个逻辑函数 `login`、`enterRoom` 和 `say` 分别对应本系统的 3 个功能: 登录、房间管理和聊天室功能。具体实现方法在下一节将会详细介绍。

10.1.3 系统编码实现

通过 10.1.2 节中的分析可知, 本系统其实只需要 `login` 一个模块就可以处理。其中的 3 个主要方法 `login`、`enterRoom` 和 `say` 方法分别对应需求分析中的 3 个功能点。下面我们就简单地分析 3 个函数的实现。

在解析函数的实现前, 我们必须清楚 `MyWeb` 框架中新增一个 `controller` 的方法, 以及如何通过 `url` 来访问新增的 `controller` 的规则。

- (1) 在 `MyWeb` 框架下的 `app/controller` 下新建一个 `login.js` 文件;
- (2) 在 `login.js` 中新增 `controller` 的代码, 代码如下:

```
module.exports = function(){
    var _res = arguments[0];
    var _req = arguments[1];

    this.test = function(){
        _res.render(VIEW+'index.jade');
    }
}
```

以上代码新增了一个 `login` 的 `controller`, 并为该 `controller` 添加了 `test` 方法, 同时指定解析 `index.jade` 文件。下面在 `view` 文件夹下新增 `index.jade` 来显示一个 `hello world` 信息。`index.jade` 代码如下:

```
html(lang="en")
  head
    meta(charset="utf-8")
    title chat
    meta(name="viewport", content="width=device-width,
initial-scale=1.0")
    meta(name="description", content="")
  body
    div hello world
```

(3) 运行项目的入口文件 `app.js`, 打开 `http://127.0.0.1:8000/login/test`, 就可以看到刚才的 `hello world` 了。

以上就是新增一个 `controller` 和方法的简单过程。接下来我们为 `login.js` 模块新增 3 个方法 `login`、`enterRoom` 和 `say`。首先看下 `login` 方法的实现, 代码如下:

```
this.login = function(){
    var room = lib.config.get(CONF + 'room.json', '');
    lib.httpParam.POST('username', function(value){
        sessionLib.username = value;
        _res.render(VIEW + 'main.jade', {'user': value, 'rooms': room});
    });
    return;
}
```

【代码说明】

- ❑ `var room = lib.config.get(CONF + 'room.json', "):` 读取配置获取房间信息;
- ❑ `lib.httpParam.POST:` 获取 post 参数 `username`;
- ❑ `sessionLib.username = value:` 存储用户登录后的 Session 值;
- ❑ `_res.render(VIEW + 'main.jade'...):` 显示主页。

`login` 函数获取登录用户名, 并将登录用户名存储在 Session 中。用户登录后跳转到房间选择页面 `main.jade`。相关代码如下:

```
include header
  .span
    div.navbar.navbar-inverse.navbar-fixed-top
      div.navbar-inner
        div.container

a.btn.btn-navbar(data-toggle="collapse", data-target=".nav-collapse")
  span.icon-bar
  span.icon-bar
  span.icon-bar
  a.brand(href="javascript:void(0)") 欢迎 #{user}
  div.nav-collapse.collapse
    ul.nav
      li
        a(href="javascript:void(0)") 注销
```

以上 jade 代码为该系统的导航栏。显示效果如图 10-2 所示。



图 10-2 系统导航栏设计

```
each room in rooms
  div.hero-unit.span3.offset1
    h1 #{room.name}
    p #{room.desc}
    p

a.btn.btn-primary.btn-large(href="/login/enterRoom/?room_id=#{room.id}")
Enten In
include footer
```

以上代码的主要功能是应用 jade 模板中的 `each` 循环展示系统配置中的所有的 `room` 信息。其展示在页面的信息如图 10-3 所示。

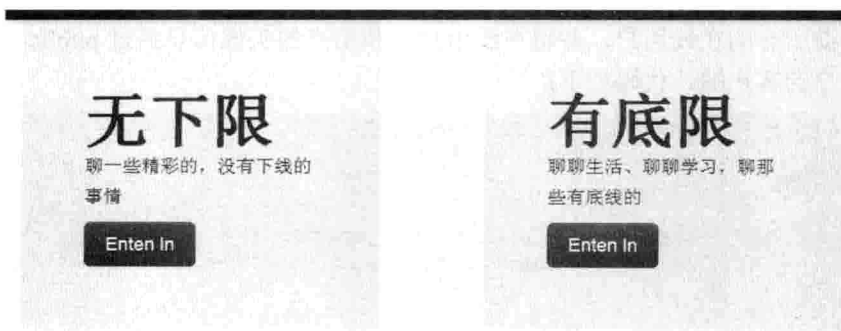


图 10-3 系统登录首页

接下来我们看一下 `enterRoom` 函数。该函数的主要作用是启动 `socket` 服务，同时刷新所有的 `socket` 连接，并将所有在线用户通过 `public` 发送给所有用户。代码如下：

```
this.enterRoom = function(){
    var roomId = lib.httpParam.GET('room_id');
```

通过 HTTP 的 GET 获取客户端传递的 `room_id` 值。代码如下：

```
    if(!onlineList[roomId]){
        onlineList[roomId] = [];
    }
    if(!onlineList['pic']){
        onlineList['pic'] = [];
    }
}
```

初始化房间在线用户 `onlineList` 对象。代码如下：

```
var time = 0;
io.sockets.on('connection', function (socket){
    var username = sessionLib.username;
    if(!username){
        return;
    }
    if(!onlineList[roomId][username] ){
        onlineList[roomId][username] = socket;
        onlineList['pic'][username] = {'pic':'picture' +
parseInt(Math.random()*2+1) + '.png', 'name':username};
    }
}
```

启动 `sockets` 连接服务，并将当前登录用户连接的 `socket` 保存在全局变量 `onlineList` 中。代码如下：

```
var refresh_online = function(){
    var n = [];
    var p = [];
    for (var i in onlineList[roomId]){
        n.push(i);
        p.push(onlineList['pic'][i]);
    }
    io.sockets.emit('online_list', n);//所有人广播
    console.log(onlineList['pic']);
    io.sockets.emit('pic_list', p);//所有人广播
}
refresh_online();
```

获取当前所有的在线用户，并将在线用户名和用户的头像信息通过 `public` 信息发送到所有在线用户的客户端。代码如下：

```
//确保每次发送一个 socket 消息
if(time > 0){
    return;
}
socket.on('disconnect', function(){
    delete onlineList[roomId][username];
    refresh_online();
});
```

如果取消 `socket` 连接，则将该用户名从 `onlineList` 的相应 `room` 删除。代码如下：

```

        time++;
        console.log(onlineList);
    });
    _res.render(VIEW + 'live.jade', {'user': sessionLib.username});
}

```

该函数最后会跳转进入房间页面。其页面如图 10-4 所示。



图 10-4 系统房间页面

该房间页面有一段前端的 JavaScript 代码，该段代码主要是在前端应用 sockets 的 API 连接 socket 服务。以下是前端 JavaScript 的部分代码：

```

var socket = io.connect();
socket.on('online_list', function (data) {
    var Dom = '';
    for(var i=0; i<data.length; i++){
        Dom = Dom + "<li><a href='javascript:void(0)'>" + data[i] +
    "</a></li>";
    }
    $(".friend_list").html(Dom);
});

```

io.connect()连接服务器端 socket 服务，socket.on 主要是监听消息事件，如果有 online_list 消息推送，则会执行回调 function 的逻辑。

最后再看一下聊天功能函数 say 方法的实现。代码如下：

```

this.say = function(){
    var username = sessionLib.username;
    var msg = lib.httpParam.GET('msg');
    var retJson = {};
    retJson['msg'] = msg;
    retJson['name'] = username;
    io.sockets.emit('say_msg', retJson);
}

```

获取当前登录用户的 username，GET 客户端发送的 msg 字符，然后应用 io.sockets 广

播信息发送给所有在线用户。客户端处理代码如下：

```
socket.on('say_msg', function (data) {
    var username = data['name'];
    var dom = "<div class='alert'><button type='button' class='close'
data-dismiss='alert'>&times;</button><strong>Tips:</strong>" + data['msg']
+ "</div>";
    $('#'+username).find('.msg_detail').html(dom);
});
```

上面代码主要是监听 say_msg 方法，如果有推送该条消息，则在该用户名的 dom 节点上添加一段 html 代码。实现的效果如图 10-5 所示。

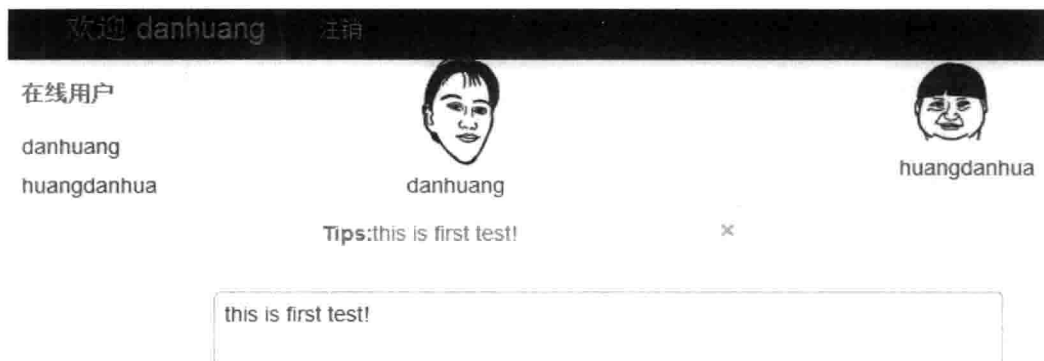


图 10-5 系统聊天页面

可以看到当前在线的两个用户 danhuang 和 huangdanhua。danhuang 这个用户发送的 this is first test 字符将会推送到所有的在线用户的客户端。图 10-5 是 danhuang 的客户端截图，图 10-6 则是 huangdanhua 的客户端的截图。



图 10-6 系统聊天信息

从图中可以看到，两个在线用户的客户端中都展示了 danhuang 这个用户发送的 this is first test 字符，这样就简单的实现了多人聊天功能。

10.2 联网中国象棋游戏

Node.js 在联网在线游戏方面也很有优势。本节介绍联网的中国象棋游戏，游戏的难度

主要是前端的规则设定，而 Node.js 主要是用来做用户之间的数据交互。本章将简单地描述前端的游戏规则实现，对 Node.js 数据交互会进行详细介绍。

10.2.1 系统设计

本系统包含用户登录、房间选择和联网游戏。和 10.1 节中的设计相同的是用户登录和房间选择功能，而主要区别在于其使用 socket 交互设计。登录功能和房间选择功能本节就不介绍了，主要设计方法和 10.1 节介绍的一致。

游戏交互设计思想是，客户端传递棋子的有效移动坐标，发送到服务器，服务器广播到本局游戏的所有用户客户端，客户端接收到数据后，移动相应的棋子系统设计流程如图 10-7 所示。



图 10-7 系统设计流程图

图 10-7 所展示的就是该游戏从数据产生，到数据演变为游戏的规则交互的一个流程图。接下来我们看一下该游戏的整个 socket 交互设计原理，如图 10-8 所示。

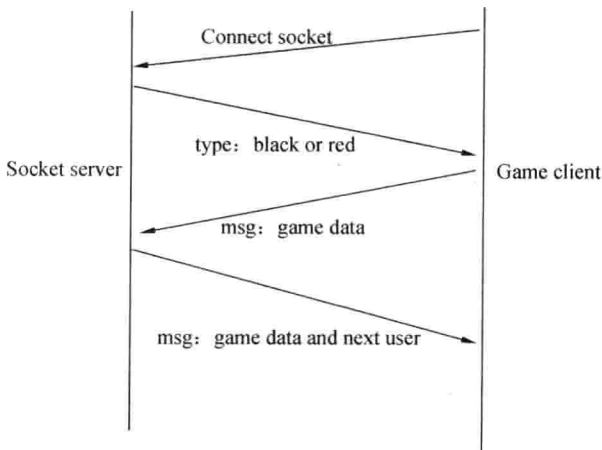


图 10-8 服务 socket 交互设计

- ❑ 游戏用户发起 socket 连接请求；
- ❑ 系统判断当前房间是否已经满两个人，如果不足两个人，则分配用户为游戏的 black 或者是 red 方；
- ❑ 游戏开始后，用户将发送游戏的坐标信息到 socket 服务端；
- ❑ 服务端接收客户端数据，并判断数据的有效性后，将数据发送给房间的所有游戏者。

以上就是整个游戏的 socket 交互过程。其中每个房间可容纳无限制人数，但是规定第一个进入的为 red 方，而第二个进入的为 black 方。

10.2.2 系统的模块设计

同样，该游戏也是应用 MyWeb 2.0 框架。登录和房间选择两个模块功能，可以完全复用 10.1 节应用的 login 模块。本模块还包含了 login 功能模块，该模块的主要作用是处理用户的登录逻辑。因此，通过以上分析，可以得到本 login 模块的代码结构如下：

```
module.exports = function(){
    var _res = arguments[0];
    var _req = arguments[1];

    this.login = function(){
    }

    this.enterRoom = function(){
    }

    this.publicMsg= function(){
    }
}
```

【代码说明】

- ❑ this.login: 用户登录模块；
- ❑ this.enterRoom : 进入游戏房间；
- ❑ this.publicMsg: 向在线游戏用户发送 socket 消息。

该模块主要是处理用户的一些动作：登录、进入游戏等功能。login 处理用户登录，enterRoom 处理用户游戏 socket 逻辑，而 publicMsg 则是向该房间游戏玩家发送游戏的 socket 数据。

10.2.3 系统编码实现

login 函数和 10.1 介绍的类似，主要是实现用户登录功能。代码如下：

```
this.login = function(){
    var room = lib.config.get(CONF + 'room.json', '');
    lib.httpParam.POST('username', function(value){
        sessionLib.username = value;
        _res.render(VIEW + 'main.jade', {'user' : value, 'rooms' : room});
        var time = 0;
    });
    return;
}
```

接下来我们来看核心函数 enterRoom 的实现。代码如下：

```
this.enterRoom = function(){
    var roomId = lib.httpParam.GET('room_id');
```

获取当前 roomId, roomId 用来作为房间的唯一标识。代码如下:

```
if(!onlineList[roomId]){
    onlineList[roomId] = [];
}

if(!onlineList[roomId]['type']){
    onlineList[roomId]['type'] = {};
}
```

onlineList[roomId]['type']存储 room 房间正在游戏的两个用户的角色 (red 和 black)。例如当 danhuang 用户进入房间后, 则存储 onlineList['1001']['type']['danhuang'] = 'red'。

```
var time = 0;
io.sockets.on('connection', function (socket){
    var username = sessionLib.username;
    if(!username){
        return;
    }
    if(!onlineList[roomId][username] ){
        onlineList[roomId][username] = socket;
    }
}
```

启动 socket 服务, 当用户连接 socket 时, 将连接用户的 socket 对象存储在该 roomId 对象中, 以便可以向该用户推送实时消息数据。代码如下:

```
var refresh_online = function(){
    var n = [];
    var athlete = '';
    for (var i in onlineList[roomId]){
        if(i != 'type'){
            n.push(i);
        }
    }
    switch(n.length){
        case 1 : athlete = 'red';
        break;
        case 2 : athlete = 'black';
        break;
        default : athlete = 'visitor';
        break;
    }
}
```

n 数组为当前房间的用户列表。根据当前房间的人数 n.length 来分配进入房间的用户角色, 房间人数为 1 时, 该用户默认为该房间游戏的 red 方, 人数为 2 时, 该用户为游戏的 black 方, 其他后面进来的所有用户都为 visitor。代码如下:

```
if(n.length > 1){
    publicMsg(roomId, 'start', {'msg':'start game'})
}
```

当前游戏房间的用户大于 2 时, 则提示开始游戏。代码如下:

```
onlineList[roomId]['type'][username] = athlete;
onlineList[roomId][username] = socket;
socket.emit('type', {'type' : athlete});
}

//确保每次发送一个 socket 消息
if(time > 0){
    return;
}
```

```

    }

    refresh_online();

    socket.on('msg', function(data){
        data['ret'] = 0;
        data['next'] = current == 'red' ? 'black' : 'red';
        current = data['next'];
        publicMsg(roomId, 'msg', data)
    });

```

接收到用户客户端的消息数据时，切换下一个走动棋子的用户类型，同时广播给该房间中所有的游戏用户。代码如下：

```

    socket.on('disconnect', function(){
        delete onlineList[roomId][username];
        delete onlineList[roomId]['type'][username];
        publicMsg(roomId, 'game_over', {'msg': 'user has left
the room, you are win!'});
        refresh_online();
    });

```

当该用户取消 socket 连接时，系统将该用户的 socket 信息删除。代码如下：

```

        time++;
    });
    var readPath = VIEW + lib.url.parse('index.html').pathname;
    var indexPage = lib.fs.readFileSync(readPath);
    _res.writeHead(200, { 'Content-Type': 'text/html' });
    _res.end(indexPage); //返回展示 html 页面
}

```

以上是服务器端 Node.js 的代码。而客户端的逻辑非常复杂，包含游戏规则的判定，以及客户端与服务器端直接的交互。接下来我们只介绍客户端与服务器端之间的代码交互，游戏的逻辑处理这里就不再介绍，有兴趣的读者，可以在本章的源码中学习。

客户端 socket 服务主要是在 changeIt.js 文件中，该文件中与服务器端交互的代码如下所示。

```
var socket = io.connect('http://127.0.0.1:8000');
```

连接本地 socket 服务。

```

socket.on('msg', function (data) {
    var ret = data['ret'];
    var current = data['next'];
    var clickList = data['click_list'];
    var value = data['value'];
    var clickListStack = new Array(2);
    if(ret == 0){
        for(var key in clickList){
            var x = clickList[key]['x'];
            var y = clickList[key]['y'];
            var location = new Location();
            location.setX(x);
            location.setY(y);
            clickListStack[key] = location;
        }
        moveTo(clickListStack, value);
        changeStyle();
    }
}

```

```

        winGame();
    }
    next = current;
    $('#msg_info').innerHTML = 'you are ' + type + '<br/>next is ' + current;
});

```

【代码说明】

- ❑ `current = data['next']`: 获取下一个移动棋子的游戏角色方;
- ❑ `moveTo(clickListStact, value)`: 移动棋子;
- ❑ `changeStyle()`: 修改相应棋子的属性;
- ❑ `winGame()`: 判断是否结束游戏;
- ❑ `$('#msg_info').innerHTML`: 添加游戏的提示信息。

当服务器端发送游戏数据时, 客户端会根据游戏数据移动相应的棋子, 并提示下一步移动方。

```

socket.on('type', function (data) {
    type = data['type'];
})

```

接收服务器端分配的角色 socket 消息, 得到角色后存储在本地全局变量 `type` 中。

```

socket.on('start', function (data) {
    $('#msg_info').innerHTML = 'game start';
})

```

接收 `start` 消息, 当系统提示游戏开始时, 则可以进行游戏。

```

socket.on('game_over', function (data) {
    alert(data.msg);
})

```

接收服务器端 `game_over` 消息, 当游戏对方退出游戏时, 则直接提示用户赢得游戏。

10.2.4 系统体验

运行 `app.js` 项目的入口文件, 分别使用 Firefox 和 Chrome 浏览器打开 `http://127.0.0.1:8000`, 可以看到如图 10-9 所示的登录页面。

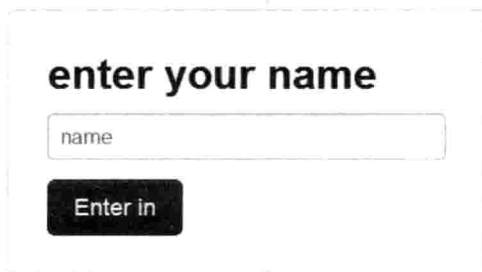


图 10-9 系统登录界面

输入相应的用户名, 单击用户登录按钮即可将登录系统, 登录后进入一个房间 (房间信息和 10.1 节介绍的类似), 就可以看到游戏界面, 如图 10-10 所示。

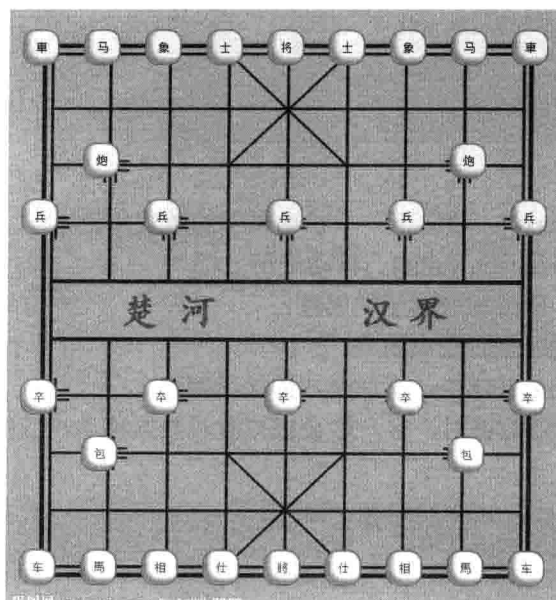


图 10-10 系统游戏界面

⚠注意：图 10-10 为游戏截图，其中包含的棋子“包”是游戏中的“炮”，是为了游戏设计的需要而用“包”

进入房间后将处于等待状态。当两个用户同时进入后，系统会在左侧提示游戏开始。游戏开始后 red 方先移动棋子，移动完成后，在界面左侧会提示游戏双方下一个移动棋子的角色，如图 10-11 所示。

you are black
next is black

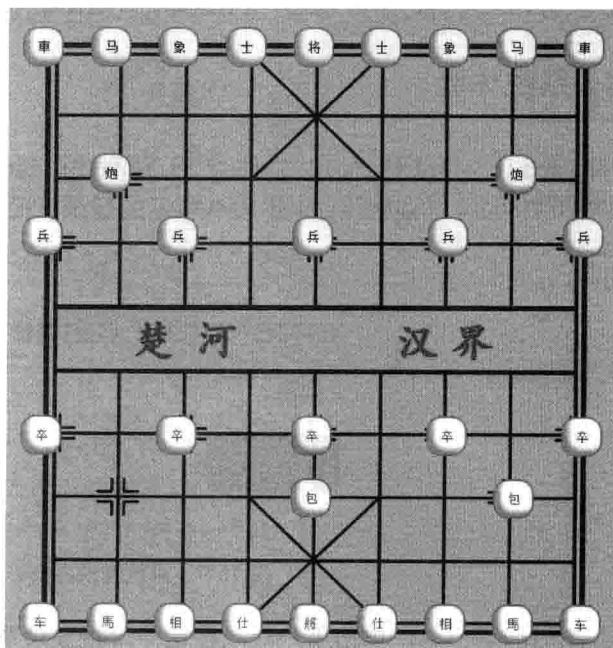



图 10-11 游戏角色分配

 注意：图 10-11 为游戏截图，其中的“包”为棋子“炮”，主要是代码设计的需要。该图左上方的英文的游戏提示信息，提示游戏角色，以及游戏下一步进行的环节。

接下来双方就可以移动棋子，进行游戏。当游戏双方中的任何一方的将被吃掉，游戏结束，并提示对方获胜；如果在游戏过程中有游戏者退出时，同样会提示对方获胜。

10.3 本章小结

本章介绍了一个实时应用和一个联网游戏，通过两个应用的介绍希望读者能够了解 Node.js 的应用开发。本章的两个应用都是应用 MyWeb 2.0 框架，希望通过本应用的开发，读者能够更快地了解 MyWeb 2.0 框架的一些应用，以及内部实现的一些机理。

第 11 章 Node.js 实用工具

在项目开发中，经常需要一些公共模块来辅助项目的开发，例如日志管理模块、配置文件读取模块、curl 模块、crontab 模块、forever 运行脚本、xml 解析模块和邮件发送模块等。这些公共模块在项目的应用开发中，都是必不可少的。本章节将会介绍 Node.js 项目开发中一些常用的工具模块，这些工具都是根据 github 上的 NPM 公共库来封装的。

11.1 日志模块工具

在项目运营以及开发过程中，日志功能管理是一个非常重要的模块，通过查看系统运行中打印的日志，我们可以查看项目的整体运行情况，以及运行过程中出现的一些异常问题。当然，对于 Node.js 项目来说，日志管理功能是非常关键的。Node.js 第三方库中有一个 log4js 模块，该模块有基本的日志信息打印功能，其提供了日志分类信息，包含 warn、error、info 和 debug。应用此第三方库模块，我们就可以做一些接口的封装，为其添加一些错误码，同时在日志记录时，新增一些数据展示功能，来满足我们项目开发中的需求。接下来我们将从日志模块介绍、日志模块实现，以及应用三个方面来介绍该模块。

11.1.1 日志模块介绍

在实现该模块的封装前，首先我们要明确该模块要如何实现，以及日志信息应该如何打印？一般情况下，单条日志信息中应该包含其日志打印的日期、打印模块、日志的错误类型、日志对应的错误码和错误相关的提示数据。为了更好地查看日志信息，可以将日志分日期、分模块、分类型来记录，因此我们希望设计出如图 11-1 所示的日志模块架构。

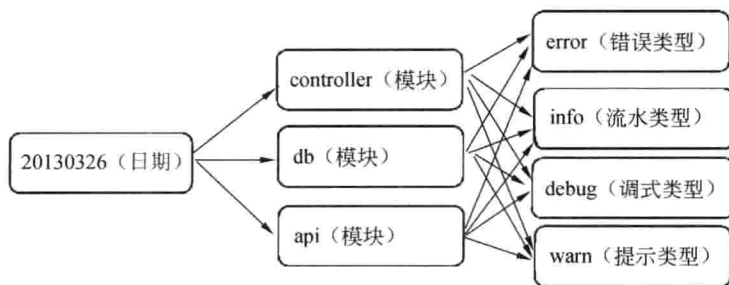


图 11-1 日志架构

如图 11-1 所示，当一条日志信息打印显示时，该模块将日期（例如 2013 年 3 月 26 日）

命名为 20130326 当天的日志文件夹。在当天日志文件夹中，包含了当天需要打印的日志模块（controller、db 和 api），每一种日志模块会对应多种类型的错误。因此在查看日志信息时，我们只需要根据日期、模块和错误类型来查询相应的错误信息。

根据如上设计思想，将会产生如图 11-2 所示的文件结构。

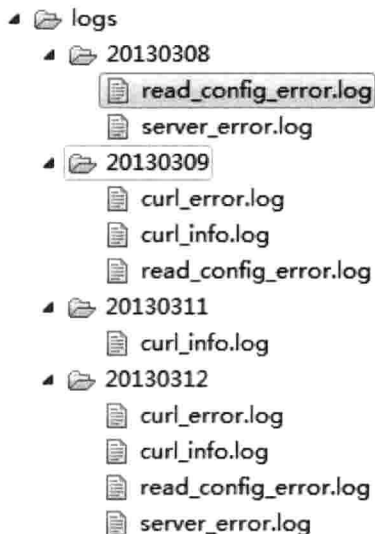


图 11-2 日志生成架构图

在图 11-2 中 20130308、20130309、20130311 和 20130312 为日志的产生日期，而这些文件夹下的 read_config、server、curl 则为相应的日志记录模块，error 和 info 则为错误类型。

以上是日志的模块设计介绍，接下来我们看看单条日志信息中包含的信息。在单条日志信息里面，应该也要包含日志记录的时间、日志类型、日志操作用户、错误码、错误信息，以及相关 json 数据信息。单条错误信息，代码如下：

```
[2013-03-08 23:16:39.972] [ERROR] danhuang - [code -3] [msg parse configure file error] {"filename":"test.xml"}
```

[2013-03-08 23:16:39.972]为日志记录时间，[ERROR]为日志类型，danhuang 为该条记录产生操作者，[code -3]为错误码-3，[msg parse configure file error]为错误信息，{"filename":"test.xml"}为该条错误信息的相关 json 数据。

综上所述，我们希望设计一个结构清晰的日志记录模块，通过清晰的日志，能够更快地定位系统运行中出现的异常问题。接下来我们就介绍如何实现该模块。

11.1.2 日志模块实现

根据 11.1.1 节的模块分析，需要应用到 NPM 中的 log4js 日志模块。下载该模块可执行如下语句：

```
npm install log4js
```

为了更好地管理日志，我们可以将错误码作为一个文件模块，该文件模块中包含错误

码、错误码对应的错误信息，以及日志的对应模块信息。初步设计错误码模块的代码框架如下：

```
/* error num */
var ERR_NUM = {
  'RET_SUCC'      : '0'
  , 'RET_ERR'     : '-1'
}

/* error message */
var ERR_MSG = {
  '0'      : 'success'
  , '-1'   : 'error'
}

/* controller */
var LOG_CONTR = {
  SERVER : 'server'
  , API   : 'api'
  , CONTR : 'controller'
  , DB    : 'database'
}

function getMsg(errorCode){
  return ERR_MSG[errorCode];
}
```

【代码说明】

- ❑ ERR_NUM：错误码对象。
- ❑ ERR_MSG：错误码对应的错误信息。
- ❑ LOG_CONTR：日志对应的模块信息。
- ❑ getMsg(errorCode)：根据错误码获取错误信息接口。

在以上代码中，应用大写字母来定义错误码和错误码对应的错误信息。目的是为了便于项目中将该模块定义为一个 global 变量，以便全局使用，同时避免全局变量带来的变量污染。代码中的 getMsg 方法主要是通过错误码获取错误信息。LOG_CONTR 是在 11.1.1 节中介绍的日志模块，这里包含了 server 模块的错误信息、API 模块的错误信息、controller 模块的错误信息和 database 模块的错误信息。

如上错误码方法添加过程非常简单，只需要在 ERR_NUM 中新增一个元素错误码，同时为该错误码添加对应的错误信息。最后我们需要将这些错误码、错误信息，以及日志模块暴露给外部调用者。

```
exports.ERR_NUM = ERR_NUM;
exports.ERR_MSG = ERR_MSG;
exports.LOG_CONTR = LOG_CONTR;
exports.getMsg = getMsg;
```

介绍完错误码模块的实现以后，我们再来看一下日志模块的实现。该模块包含 error、info、warn 和 debug 4 个接口，分别对应 4 种日志类型，代码如下：

```
this.error = function(errNum, controller, logInfo){
  var errType = 'error';
```

```

        log(errType, errNum, controller, logInfo);
    }

    // for info log
    this.info = function(errNum, controller, logInfo){
        var errType = 'info';
        log(errType, errNum, controller, logInfo);
    }

    // for warn log
    this.warn = function(errNum, controller, logInfo){
        var errType = 'warn';
        log(errType, errNum, controller, logInfo);
    }

    // for debug log
    this.debug = function(errNum, controller, logInfo){
        var errType = 'debug';
        log(errType, errNum, controller, logInfo);
    }
}

```

【代码说明】

- ❑ this.error = function(errNum, controller, logInfo): error 日志接口。
- ❑ this.info = function(errNum, controller, logInfo): info 日志接口。
- ❑ this.warn = function(errNum, controller, logInfo): warn 日志接口。
- ❑ this.debug = function(errNum, controller, logInfo): debug 日志接口。

由于 4 种类型的错误日志记录方式都是一样的，只是在日志类型上有所区别，因此我们使用统一的 log(errType, errNum, controller, logInfo) 函数来处理所有的日志记录，但该方法中必须对不同的类型做不同的处理。该方法实现如下：

```

// add log in log file
function log(errType, errorCode, controller, otherInfo){
    var otherInfo = otherInfo ? otherInfo : {};
    /* 错误日志信息 */
    var errorMsg = appLog.getMsg(errorCode);
    /* 记录日志的文件名 */
    var errorLog = getLogFileName(errType, controller);
    log4js.addAppender(lib.log4js.appenders.file(errorLog),
loguser);
    var jsonStr = JSON.stringify(otherInfo);
    errorMsg = '[code ' + errorCode + ']' + ' ' + '[msg ' + errorMsg + ']'
+ jsonStr;
    /* 记录日志 */
    addLog(errType, errorMsg);
    log4js.clearAppenders();
}

```

【代码说明】

- ❑ errorMsg = appLog.getMsg(errorCode): 根据 error code 获取错误信息。
- ❑ errorLog = getLogFileName(errType, controller): 根据错误类型以及日志模块，获取记录日志的文件。
- ❑ lib.log4js.addAppender(lib.log4js.appenders.file(errorLog), loguser): log4js 模块初始化，为其新增日志操作用户。
- ❑ jsonStr = JSON.stringify(otherInfo): 将 json 对象转化为字符，便于存储在日志信

息中。

- ❑ `errorMsg`: 日志信息字符串。
- ❑ `addLog(errType, errorMsg)`: 向文件中写入日志信息。
- ❑ `log4js.clearAppenders()`: 清空本次 `appenders` 的文件。

上面代码中的 `appLog` 对象是本节开头部分实现的错误码模块，该对象的声明方法为 `var appLog = require('./appLog')`。应用模块返回的对象，调用其模块方法 `getMsg`，获取该错误码和错误信息。在 `log4js` 的 `github` 示例代码中，可以看到该模块需要 `log4js.appenders.file` 来初始化 `log4js` 中的日志文件路径和日志操作者，`log4js` 为本模块的一个全局对象，该对象的申明方法，可以看本节最后两段内容。

代码中的 `log4js.clearAppenders` 非常重要，如果不添加这段代码，会导致一个日志文件记录多条重复的日志信息。该接口 `log4js` 在示例中没有提供，可以查看源码的第 131 行。

这部分功能中的 `getLogFileName` 函数会根据当前时间、错误类型和日志模块名，生成一个日志文件名，并将该日志文件字符串返回。例如，当前如果是 2013-03-26，日志类型是 `error`，日志模块是 `server`，根据这些基本的信息，`getLogFileName` 函数会返回一个路径名 `20130326/server_error.log` 字符串。相关该函数的代码如下：

```
function getLogFileName(errType, controller){
  var logPrefix = logpath + '/' + dateStr + '/';
  // create log file path
  try{
    fs.readdirSync(logPrefix);
  } catch (err){
    fs.mkdirSync(logPrefix);
  }
  switch(errType){ // 根据日志类型的不同，返回不同的日志文件路径
    case 'error' : return logPrefix + controller + '_error.log';
    break;
    case 'info' : return logPrefix + controller + '_info.log';
    break;
    case 'warn' : return logPrefix + controller + '_warn.log';
    break;
    case 'debug' : return logPrefix + controller + '_debug.log';
    break;
  }
}
```

【代码说明】

- ❑ `logPrefix = logpath + '/' + dateStr + '/'`: `dateStr` 为时间字符串生成路径前缀。
- ❑ `fs.readdirSync(logPrefix)`: 判断该文件路径是否存在。
- ❑ `fs.mkdirSync(logPrefix)`: 不存在该路径时，创建文件路径。
- ❑ `switch(errType)`: 根据日志类型，返回不同的文件名。

在上述代码中，使用 `switch` 来判断日志类型，根据日志类型返回不同的文件名，例如 `error` 时，为其文件名添加 `_error` 后缀，`info` 时为其文件名添加 `_info` 后缀。`dateStr` 为本模块的一个全局变量，其主要是将当前时间转化为时间字符串。其逻辑实现如下：

```
function getDateTime(timestamp) {
  var timeTemp = timestamp ? new Date(timestamp) : new Date(),
  currentTime;
  var yy = timeTemp.getFullYear();
  var MM = timeTemp.getMonth();
```

```

var dd = timeTemp.getDate();

// fixed MM, value of MM from 0 to 11
MM === 0 ? MM = 12 : MM++;

// fixed time format
MM < 10 ? MM = '0'.concat(MM) : null;
dd < 10 ? dd = '0'.concat(dd) : null;
return ' ' + yy + MM + dd;
}

```

以上是一个 JavaScript 的时间处理逻辑，该函数功能由于没有涉及调用浏览器对象，因此在 Node.js 中也是适用的。该函数的主要功能是获取当前时间，将当前时间组合成 yy + MM + dd 这种类型的字符串，并返回。

最后我们再看一下 log4js 是如何记录日志的，代码如下：

```

function addLog(errType, errorMsg){
  switch(errType){ // 根据日志类型的不同，记录不同的日志信息
    case 'error' : logger.error(errorMsg);
    break;
    case 'info' : logger.info(errorMsg);
    break;
    case 'warn' : logger.warn(errorMsg);
    break;
    case 'debug' : logger.debug(errorMsg);
    break;
  }
}

```

logger 为该模块的一个全局变量，log4js 提供了 4 个接口，分别是 error、info、warn 和 debug，因此我们根据不同的类型调用 log4js 不同的接口。

本模块涉及两个全局变量，分别是 dateStr 和 logger，其申明方式如下：

```

var log4js = require('log4js');
log4js.loadAppender('file');
var loguser = user
  , logpath = path
  , dateStr = getDateTime()
  , logger = log4js.getLogger(loguser);

```

【代码说明】

- ❑ var log4js = require('log4js')：获取 log4js 模块；
- ❑ log4js.loadAppender('file')：设置 log4js 日志的存储类型；
- ❑ dateStr = getDateTime()：获取时间戳字符串；
- ❑ log4js.getLogger(loguser)：初始化产生日志的操作者。

以上所介绍的就是本书的日志模块，该日志功能主要的逻辑流程是：产生一个路径文件（20130326/server_error.log）；应用 log4js 将一些日志信息写入文件；分类保存产生的日志信息。完成该模块后，我们来看一下该模块的应用方法。

11.1.3 日志模块应用

该模块是使用 module.exports 返回的一个函数，因此在 require 得到该函数后，需要为

该函数创建实例对象。在初始化该对象时，该模块需要两个参数，两个参数的作用分别是设置日志的存储的根路径和日志操作用户字符名称。相关代码如下：

```
/* index.js */
var Log = require('./log.js');
var log = new Log('danhuang', 'logs/');
var appLog = require('./appLog');
```

上述代码首先是 require 该 log 模块，获取该模块返回的 Log 函数类，并通过 new 关键字来创建该函数类的实例对象 log。由于在日志的记录过程中，需要获取打印日志的错误码和错误信息，因此需要应用到日志错误码模块 appLog。根据 log 模块和 appLog 模块，创建生成的 log 和 appLog 对象，就可以实现简单的日志信息打印功能。代码如下：

```
log.info(appLog.ERR_NUM.RET_SUCC, appLog.LOG_CONTR.SERVER, {'data':'is ok'});
log.error(appLog.ERR_NUM.RET_ERR, appLog.LOG_CONTR.API, {'data':'is not ok'});
```

上面代码分别记录了 info 和 error 日志信息，分别对应 server 模块和 API 模块。接下来我们运行该段代码：

```
node index.js
```

首先可以看到在运行窗口返回如图 11-3 所示的信息。

```
H:\11.1>node index.js
[2013-03-27 19:10:55.755] [INFO] danhuang - [code 0] [msg success] <"data":"is ok">
[2013-03-27 19:10:55.755] [ERROR] danhuang - [code -1] [msg error] <"data":"is not ok">
```

图 11-3 index.js 脚本运行结果

在图 11-3 中所观察到的打印信息，就是记录的日志信息，接下来我们查看 logs 文件夹结构，如图 11-4 所示。

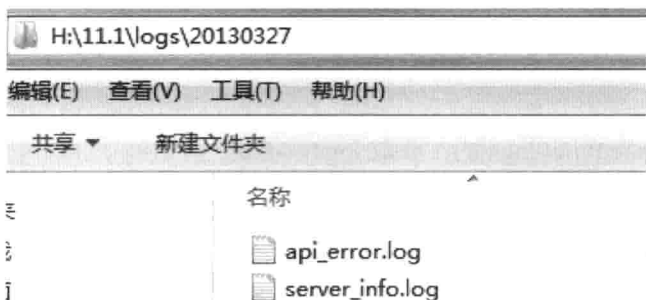


图 11-4 产生日志文件

根据本章日志模块的设计思想，图 11-4 中生成了一个 20130327 当天日志信息管理文件夹，并在该文件夹下生成了两个日志文件 api_error.log 和 server_info.log，这和模块要求实现的一致。打开两个日志文件，可以查看到如下日志信息：

api_error.log

```
[2013-03-27 20:25:49.884] [ERROR] danhuang - [code -1] [msg error]
```

```
{"data":"is not ok"}
```

```
server_info.log
```

```
[2013-03-27 20:25:49.923] [INFO] danhuang - [code 0] [msg success]
{"data":"is ok"}
```

综上所述，就实现了我们的日志管理模块。本节主要是从模块的实现到模块的应用来介绍，重点是介绍日志管理的实现，读者在学习 log 功能模块时需要了解该模块的实现机制，而不是简单地应用该模块。

在本次模块的开发过程中遇到的最大问题是日志被重写的问题。主要原因是每次使用 log4js 对象后，没有调用 log4js 对象的 clearAppenders 方法，导致前面的日志文件被重复写入多个文件中。这个知识点需要读者了解就可以了。

11.2 配置文件读取模块

在项目开发过程中，开发者为了后续改动较少，会将那些与项目运行环境相关的变量值，例如数据库配置信息、用户名、密码、数据库名存储在配置文件中，以便项目运营过程中修改，如需切换环境，只需要改动配置文件即可。因此对于一个成熟的项目来说，配置文件是必不可少的，而 Node.js 也不可缺少配置文件解析模块。既然配置文件在项目是必不可少的，那么配置文件解析的模块也是项目开发中的一部分。本章同样从模块的应用分析、模块的实现，以及模块的应用 3 个方面来阐述 Node.js 的配置文件解析模块。

11.2.1 配置文件解析模块介绍

一般项目中可能存在多种配置文件，例如 json 格式、conf 类型或者 xml 类型，对于这些类型都有相应的解析方式。本模块只介绍解析 json 格式和 conf 类型的配置文件。

解析配置文件的一般过程如图 11-5 上半部分所示。

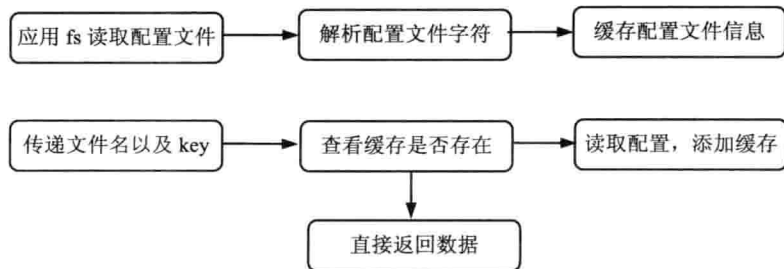


图 11-5 配置文件解析流程图

图 11-5 的下半部分主要说明，当请求一个配置文件数据时，首先会去缓存中判断是否已经存在数据，当缓存中没有时，会根据相应路径读取配置文件内容，并将读取返回的数据内容添加到缓存，同时返回配置文件解析后返回的数据。

根据上面的分析，该模块的大致架构设计如下：

```

global.CONFIG = {};

function get(fileName, type, key){
}

function getConf(filePath, key){
}

function getJson(filePath, key){
}

exports.get = get;

```

【代码说明】

- ❑ global.CONFIG: 用于配置文件缓存。
- ❑ get(fileName, type, key): 根据文件名、类型和键值获取配置信息。
- ❑ getConf(filePath, key): 解析 conf 类型的配置文件。
- ❑ getJson(filePath, key): 解析 json 类型配置文件。

作为配置文件模块建议只暴露一个接口给外部调用，便于其他接口的使用。本模块只暴露了 GET 方法给外部调用。上面是本模块的代码架构，下一节我们将实践该模块中的 3 个方法。

11.2.2 配置文件解析模块实现

首先我们看一下 getJson(fileName, filePath, key)方法实现。该函数接口的实现逻辑，主要是应用 Node.js 原生模块 fs 读取配置文件内容，应用 JSON.parse 解析该 json 配置文件信息，最后将相应数据做缓存处理。实现代码如下：

```

function getJson(fileName, filePath, key){
  try{
    var str = fs.readFileSync(filePath, 'utf8');
    configJson = JSON.parse(str);
  }catch(e){
    console.log(e);
    return '';
  }
  CONFIG[fileName] = configJson;
  return configJson[key] ? configJson[key] : configJson;
}

```

【代码说明】

- ❑ fs.readFileSync(filePath, 'utf8'): 以 utf8 格式读取配置文件信息。
- ❑ configJson = JSON.parse(str): 解析 json 数据。
- ❑ CONFIG[fileName] = configJson: 缓存配置文件信息。

以上代码中应用到 Node.js 原生模块中的 fs 模块的 readFileSync 方法，来实现同步读取配置文件内容。该函数接口需要两个参数，其中，第一个是文件路径字符，第二个参数是读取配置文件的编码类型，本配置文件解析模块应用的是 utf8 编码。该参数主要是为了避免编码不同，导致读取出的字符出现乱码现象，在本配置文件解析模块应用时，必须保证写入配置文件格式也是 utf8 编码类型。

json 格式的配置文件如下：

```
{
  "db" : {
    "user" : "danhuang",
    "password" : "aa",
    "database" : "test_db"
  }
}
```

接下来看一下 getConf(fileName, filePath, key)方法的实现。实现的逻辑方法如下：

```
function getConf(fileName, filePath, key){
  try{
    var r = [],
        q = require("querysting"),
        f = fs.readFileSync(filePath, "utf8"),
        v = q.parse(f, '[', ']'),
        t;
  }catch(e){
    console.log(e);
    return '';
  }
  for (var i in v) {
    if (i !== '' && v[i] !== '') {
      r = {};
      t = q.parse(i, v[i], '=');
      for (var j in t) {
        if (j !== '' && t[j] !== '')
          r[j] = t[j];
      }
    }
  }
  cache(filePath, fileName, r);
  return r[key] ? r[key] : r;
}
```

【代码说明】

- ❑ f = fs.readFileSync(filePath, "utf8"): utf8 格式读取配置文件。
- ❑ v = q.parse(f, '[', ']'): 使用[]解析配置文件。
- ❑ cache(filePath, fileName, r): 缓存配置文件内容。

这部分由于 Node.js 没有提供相应的解析模块，因此需要实现一个简单的算法来实现该类型的配置文件解析。上述代码中应用到了 Node.js 的 querysting 模块中的 parse 方法，该方法的主要功能是将一个查询字符串反序列化为一个对象，该 parse 包含 3 个参数，其中后两个参数为配置文件字符分割参数。

conf 类型的配置文件格式如下：

```
[user=root
name=testdb]
```

两个方法都实现后，我们再来看一下在 GET 方法中如何处理资源的请求分配，以及缓存处理。代码如下：

```
function get(fileName, type, key){
  if(CONFIG[fileName]){
    return CONFIG[fileName][key] ? CONFIG[fileName][key] :
    CONFIG[fileName];
  }
}
```

```

    }
    var filePath = 'conf/' + fileName;
    switch(type){ // switch 判断文件类型，然后根据不同类型解析文件
        case 'conf' :
            return getConf(fileName, filePath, key);
            break;
        case 'json' :
            return getJson(fileName, filePath, key);
            break;
    }
}

```

【代码说明】

- ❑ fileName: 配置文件名;
- ❑ type: 请求配置文件类型, conf 或 json 格式;
- ❑ key: 需要返回的对象数据;
- ❑ var filePath = 'conf/' + fileName: 根据文件名, 获取配置文件路径;
- ❑ switch(type): 根据请求 type 调用不同的配置文件解析方法。

这里应用的是一个 global 对象来做配置文件缓存, 缓存的数据 key 值为其文件名, 在每次读取该配置文件信息时, 会优先判断该文件是否被解析过。如果有做解析, 则直接返回数据。这样应用的时候会出现一个问题, 就是当配置文件更新时, 必须要重启 Node.js 服务器, 不能自动判断配置文件更新。因此接下来我们将缓存这部分工作剥离出来, 用一个单独的函数来处理。getCache 就是来处理是否直接读取缓存内容的功能函数, 该函数的主要逻辑是获取文件的最后更新时间, 然后将最后更新时间作为其缓存键值的一部分, 当下一次请求该配置文件信息时, 首先会读取其最后更新时间, 然后判断最后更新时间的缓存配置信息是否存在。相关代码如下:

```

function getCache(filePath, fileName, key){
    var stat = fs.statSync(filePath);
    var timestamp = Date.parse(stat['mtime']);
    if(CONFIG[fileName+timestamp]){
        return CONFIG[fileName+timestamp][key]
    }
    CONFIG[fileName+timestamp][key] : CONFIG[fileName+timestamp];
    return null;
}

```

【代码说明】

- ❑ var stat = fs.statSync(filePath): 获取文件信息;
- ❑ var timestamp = Date.parse(stat['mtime']): 将文件最后修改值转化为 timestamp;
- ❑ CONFIG[fileName+timestamp]: 将文件名和文件最后修改值作为键值来缓存配置文件信息。

使用 fs 模块中的 statSync 同步获取一个文件的信息, 调用这个函数后, 其返回的信息大致如下:

```

{ dev: 0,
  ino: 0,
  mode: 33206,
  nlink: 1,
  uid: 0,
  gid: 0,

```

```
rdev: 0,
size: 1747,
atime: Tue, 03 Jan 2012 13:35:51 GMT,
mtime: Tue, 03 Jan 2012 13:35:51 GMT,
ctime: Wed, 21 Dec 2011 14:31:59 GMT }
```

其中的 `mtime` 就是我們所需要的文件的最后修改时间，为了做主键需要，应用 `date` 的 `parse` 方法将其转化为 `timestamp` 值。既然获取缓存的方法有了更改，那么必须要修改 `getJson` 和 `getConfig` 中的添加缓存的方法。当然这里我们同样剥离出该函数为 `cache`，其实现代码如下：

```
function cache(filePath, fileName, data){
    var stat = fs.statSync(filePath);
    var timestamp = Date.parse(stat['mtime']);
    if(data){
        CONFIG[fileName+timestamp] = data;
    }
}
```

以上代码方法的实现较为简单，第一步获取文件缓存的键值，然后将数据添加到 `CONFIG` 全局变量中作为缓存。

最后是修改其中的 `getJson` 和 `getConfig` 方法，将原有的类似：

```
CONFIG[fileName] = configJson;
```

更改为：

```
cache(filePath, fileName, data);
```

综上所述就是配置文件的读取模块，其中包含了两种配置文件的读取，以及配置文件缓存。为了更好地运营项目，我们在配置文件缓存的方式上做了部分修改，以便配置文件更改时，缓存内容可以立即作相应的修改，而不是通过重启服务来解决。

前面介绍的分别是两个配置解析的方法。本模块只暴露给一个 `GET` 方法，该方法在前面已经做了介绍，由于缓存方法做了更改，因此 `GET` 方法也做了修改，实现代码如下：

```
function get(){
    var fileName = arguments[0],
        type     = arguments[1],
        key      = arguments[2] ? arguments[2] : '';
    var filePath = 'conf/' + fileName;
    // 从 cache 中获取数据
    var cacheData = getCache(filePath, fileName, key);
    if(cacheData){
        return cacheData;
    }
    // 读取配置文件信息，并做缓存
    switch(type){
        case 'conf' :
            return getConfig(fileName, filePath, key);
            break;
        case 'json' :
            return getJson(fileName, filePath, key);
            break;
    }
}
```

【代码说明】

- ❑ `fileName = arguments[0]`: 获取第一个参数, 文件名。
- ❑ `type = arguments[1]`: 获取第二个参数, 解析配置文件类型。
- ❑ `key = arguments[2]`: 返回数据的主键值。
- ❑ `cacheData = getCache(filePath, fileName, key)`: 读取 `cache` 中的数据。

下面我们来简单看一下该模块的应用方法。创建一个 `index.js`, 代码如下:

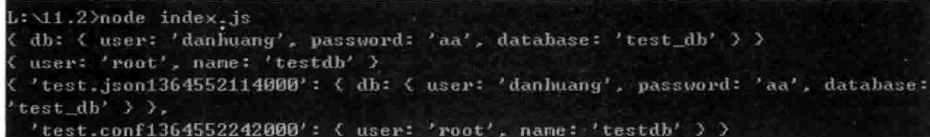
```
var config = require('./config');
var testConf = config.get('test.json', 'json');
console.log(testConf);
var info = config.get('test.conf', 'conf');
console.log(info);
console.log(CONFIG);
```

【代码说明】

- ❑ `config = require('./config')`: `require` 该配置模块;
- ❑ `config.get('test.json', 'json')`: 获取 `json` 配置文件;
- ❑ `config.get('test.conf', 'conf')`: 获取 `conf` 配置文件;
- ❑ `console.log(CONFIG)`: 打印所有的缓存配置信息。

以上测试代码逻辑是 `require` 该模块, 并调用该模块对象的 `GET` 方法来读取配置信息, 同时为了查看缓存的数据, 本测试代码将全局变量 `CONFIG` 打印显示出来。

执行 `index.js` 脚本, 可以看到如图 11-6 所示的返回信息。



```
L:\11.2>node index.js
< db: { user: 'danhuang', password: 'aa', database: 'test_db' } >
< user: 'root', name: 'testdb' >
< 'test.json1364552114000': { db: { user: 'danhuang', password: 'aa', database:
'test_db' } },
'test.conf1364552242000': { user: 'root', name: 'testdb' } >
```

图 11-6 配置文件解析结果

其中输出的第一行为读取 `json` 配置文件返回的一个 `json` 对象, 第二行为读取 `conf` 配置文件的返回对象, 最后两行则为配置文件缓存数据。

以上就是本模块的实现和应用实例介绍。读者在实际应用中需要对本模块进行完善, 从而更好地适用于自己的项目开发。

11.3 curl 模块

在第 3 章中已涉及 `curl` 模块的应用, 本节将完善第 3 章中所应用到的 `curl` 模块。本节中的 `curl` 模块主要是通过 Node.js 发起 `GET` 和 `POST` 请求。本节包含 `curl` 模块的设计和实际应用开发两个方面的知识。

11.3.1 curl 模块介绍


在介绍该模块前, 我们首先了解一下 `curl` 模块是什么?

在 Linux 系统环境下, curl 主要是利用 url 语法, 在 Linux 的命令行方式下工作的文件传输工具。简单地说, 就是应用 curl 模块模拟浏览器的 HTTP 请求, 例如下面指令:

```
curl http://www.qq.com
```

如果这里的 url 指向的是一个文件或者一幅图片, 都可以直接下载到本地。如果下载的是 html 文档, 那么默认不显示文件头部, 即 html 文档的 header。

而 Node.js 中的 curl 实际就是一个模拟浏览器的 HTTP 请求的模块, Node.js 通过应用 curl 模块请求外部 url 接口。在本模块中我们借助第三方 NPM 中的 request 模块, 实现 HTTP 和 HTTPS 中的 GET、POST 的常用方法, 当然该模块也提供 form 表单文件上传的功能, 下面是本模块的代码结构。

 **注意:** 在介绍本模块时, 首先要下载 request 模块, 源码中已经下载了该模块, 该模块的下载方法是执行 npm install request。

```
var request = require('request');
var querystring = require('querystring');

module.exports = {
  // params: url, get, callback
  get : function(){
  },
  // params: url, get, callback
  post : function(){
  },
  // params: url, get, callback
  form_post : function(){
  }
}
```

【代码说明】

- ☐ request = require('request'): require request 模块。
- ☐ get : function(): HTTP 的 GET 方法。
- ☐ post : function(): HTTP 的 POST 方法。
- ☐ file_post : function(): form 文件上传。

为了调用方便, 我们不限限制接口的参数, 所有的参数在函数内部使用 arguments 来获取, 通过这种方式, 调用者可以自由选择是否传递 GET 或者 POST 参数。代码中的 3 个接口分别对应 HTTP 或者 HTTPS 中的 GET、POST 和文件上传功能函数。下面我们来看这 3 个函数的实现。

11.3.2 curl 模块实现

首先我们看一下 GET 方法, 该函数包含 3 个参数: url、get 和 callback。url 为 GET 请求链接, get 为需要传递的参数对象, callback 为回调函数。下面是 get 方法的实现代码:

```
get : function(){
  // 获取 get 方法的参数 url、get 和 callback
  var url = arguments[0]
    , get = arguments[1]
    , callback = arguments[2];
```

```

        if(!callback && typeof get == 'function'){
            get = {};
            callback = arguments[1];
        }

        if(!url){
            callback('');
        }

        var params = {};
        // 为 url 后缀添加?或者&符号
        if(get){
            if(url.indexOf( '?') > -1){
                url = url + '&';
            } else {
                url = url + '?';
            }
        }
        url = url + querystring.stringify(get);

        params[ 'url' ] = url;
        params[ 'json' ] = true;
        // 调用 request 请求资源
        request.get(params, function(error, response, result){
            if(error){
                console.log(error);
                callback(result);
            } else {
                callback(result);
            }
        });
    }
}

```

【代码说明】

- ❑ `if(!callback && typeof get == 'function')`: 判断函数中的参数列表是否有传递 `get` 参数对象, 如果没有, 则第二个参数默认为 `callback` 函数。
- ❑ `url = url + querystring.stringify(get)`: 将 `get` 参数对象转化为 `url` 请求参数字符。
- ❑ `request.get(params, function(error, response, result))`: 调用 `request` 的 `get` 方法发起 HTTP 请求。

从上面代码可以看出, 该 `get` 函数接口的参数是动态的, 有两种类型: 携带两个参数 `url` 和 `callback` 的方式或者携带 3 个参数 `url`、`get` 和 `callback` 的方式。为了可以动态地使用参数方法, 在函数内部使用了 `arguments` 对象来获取参数列表, 并根据其参数类型的不同来判断参数对象。

一般在 HTTP 的 `url` 请求参数字符串中会自带一些参数 HTTP 的 GET 参数, 例如 `http://www.qq.com/?a=test`。当然 HTTP 请求的 `url`, 有时候也不会携带任何 HTTP 的 GET 参数, 例如 `http://www.qq.com/`。两者 HTTP 请求 `url` 的区别在于, 当我们要为该 `url` 新增一个参数时, 前一种方式是 `url+'&'+b=test` 转化为 `http://www.qq.com/?a=test&b=test`, 而后面一种则是 `url+'?' +a=test` 转化为 `http://www.qq.com/?a=test`。因此在将 `get` 参数字符串和 `url` 合并的时候, 首先需要判断 `url` 字符中是否已经包含请求参数 (或者? 字符), 如果存在那么为其添加一个 `&` 字符, 如果没有, 则需要在 `url` 后面添加? 字符。上面代码中的 `if(url.indexOf('?') > -1)` 部分就是做这个简单的处理。

接下来我们看一下 POST 方法的实现。和 GET 方法类似，但其实现相对 GET 更为简单，下面是实现代码：

```
post : function(){
    // 获取 get 方法的参数 url、get 和 callback
    var url = arguments[0]
        , post = arguments[1]
        , callback = arguments[2];
    if(!callback && typeof post == 'function'){
        post = {};
        callback = arguments[1];
    }

    if(!url){
        callback('');
    }

    var 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】

□ `params['form'] = post`: request 模块中的参数使用 form 为键值传递。

这部分功能相对 GET 方法实现更为简单，了解 request 中的 post 方法的使用即可。

request 中的 post 可以有如下多种调用方式：

```
request.post({url:'url', json:boolean, form:Object}, function(error,
response, result){})
request.post('url', {form:Object}, function(error, response, result){})
```

最后看一下文件上传相关的 form 表单提交方式，代码如下：

```
form_post : function(){
    var url = arguments[0]
        , data = arguments[1]
        , callback = arguments[2];
    if(!callback && typeof data == 'function'){
        data = {};
        callback = arguments[1];
    }

    if(!url){
        callback('');
    }

    var request = request.post(url);
    var form = request.form();
    for(var key in data){
```

```

        form.append(key, data[key]);
    }
}

```

该方法与 post 功能函数的主要不同点是，form 表单的 post 数据的方式做了改变。该 form 对象中可以为其添加 post 的文件数据，例如如下方式向 form 对象中 append 数据，其中的 post 文件需要使用 fs 对象读取其数据，才能 append 到 form 中进行提交，代码如下：

```

form.append('my_field', 'my_value')
form.append('my_buffer', new Buffer([1, 2, 3]))
form.append('my_file', fs.createReadStream('doodle.png'))

```

以上实现了一个包含 HTTP 的 GET 和 POST 方法的 curl 模块，下一节我们简单介绍一下该模块的应用方法以及应用场景。

11.3.3 curl 模块应用

curl 模块的应用很简单，只需要 require 该模块，使用 require 返回的对象，然后应用返回的对象调用其暴露的 3 个方法 GET、POST 和 form_post 就可以了。示例代码 index.js 如下：

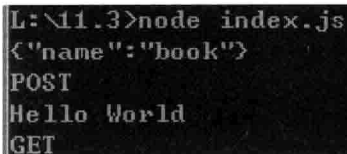
```

var curl = require('./curl');
curl.get('http://127.0.0.1:1337', function(data){
    console.log(data);
});

curl.post('http://127.0.0.1:1400', {'name':'book'}, function(data){
    console.log(data);
});

```

上面的代码就是简单地调用了 curl 模块的两个方法 GET 和 POST，当然在执行如上脚本时，我们必须添加两个本地的 server，分别对应 http://127.0.0.1:1337 和 http://127.0.0.1:1400，两个 server 的代码可以参考源码中的代码。首先打开两个窗口运行 server_get.js 和 server_post.js，最后执行 index.js，可以看到如图 11-7 所示的返回结果。



```

L:\11.3>node index.js
{"name":"book"}
POST
Hello World
GET

```

图 11-7 curl 执行结果

图 11-7 显示的是通过 curl 模块请求两个 server 返回的结果，其中第一个返回的是 post 的参数，以及请求方式 POST，第二个返回的是请求 get 返回后的字符数据以及请求方式 GET。

curl 模块主要应用于外部 API 的调用上，例如当我们需使用 Node.js 调用 CGI 的一个 API 时，使用 curl 是一个非常合适的选择。当然还有其他功能，例如定时抓取某一个网页上面的数据，利用数据字符分析得到我们想要的结果。

本实践应用模块存在的问题是 url 未经过 url 编码，因此当遇到特殊字符（空格和中文

等) 会出现访问异常。为了解决这个 url 转义问题, 在使用该模块时, 需要将其优化, 服务端应用对客户端请求的 url 字符串, 应用 `encodeURIComponent` 函数来进行转义处理, 将 GET 方法修改如下:

```
get : function(){
    // 获取 get 方法的参数 url、get 和 callback
    var url = arguments[0]
        , get = arguments[1]
        , callback = arguments[2];
    if(!callback && typeof get == 'function'){
        get = {};
        callback = arguments[1];
    }

    if(!url){
        callback('');
    }

    var params = {};
    // 为 url 后缀添加?或者&符号
    if(get){
        if(url.indexOf( '?') > -1){
            url = url + '&';
        } else {
            url = url + '?';
        }
    }
    url = encodeURIComponent(url + querystring.stringify(get));

    params[ 'url' ] = url;
    params[ 'json' ] = true;
    // 调用 request 请求资源
    request.get(params, function(error, response, result){
        if(error){
            console.log(error);
            callback(result);
        } else {
            callback(result);
        }
    });
},
```

11.4 crontab 模块

`crontab` 是 Linux 的一个定时任务工具, 但每次我们需要定时执行一个脚本时, 需要在 Linux 命令行窗口下执行 `crontab -e`, 然后在 `crontab` 编辑文件中, 添加一个定时任务命令。但是在 Windows 下运行项目时, 我们就无法添加定时任务了, 因此 Node.js 中的 `crontab` 模块的存在意义就非常重要了。并且 Linux 下的 `crontab` 是不可运维的, 例如服务器迁移时, 都需要手动拷贝, 这给运行带来大量的工作。

11.4.1 crontab 模块介绍

在 Linux 运行环境下，我们经常会应用 **crontab** 功能来定时执行任务（在指定的一个时间点来执行某个命令）和隔断时间任务（每隔间断时间执行任务）。那么我们在本模块设计上将包含该两个功能，分别是定时任务和隔断时间任务函数接口。

在实现该模块前，先来了解如何将一个脚本命令作为一个任务来执行的问题。在实际应用场景中可能有多种 Linux 命令任务模型，该 **crontab** 模块只考虑 3 种任务模型：1. 函数任务，定时执行一个 Node.js 的函数任务；2. shell 脚本任务，定时调用一个系统 shell 脚本；3. API 任务，定时调用一个 API 接口。

下面是本模块的两个任务执行函数接口，代码如下：

```
/*
 *
 * crontab.js
 */
module.exports = {
  /**
   * 定点任务
   */
  time_task : function(){

  },

  /**
   * 隔断时间任务
   */
  circle_task : function(){

  }
}
```

【代码说明】

- **time_task**: 执行定点任务；
- **circle_task**: 执行隔断时间任务。

两个接口函数分别对应于定点任务和隔断时间任务接口，两个函数都是动态参数调用，其参数获取方法在函数中使用 **arguments** 对象获取其参数。

11.4.2 crontab 模块设计实现

定时任务和隔断时间任务都可以采用 Node.js 的 **setTimeout** 来实现。我们先来实践隔断时间接口的实现方法，代码如下：

```
/**
 *
 * 隔断时间任务
 */
```



```

circle_task : function(){
    var sec      = arguments[0],
        task     = arguments[1];
    if(!sec || !task){
        callback('');
    }
    // 如果是函数任务时, 则执行函数
    if(typeof task == 'function'){
        setInterval(function(){
            task.call();
        }, sec);
    }

    // 如果是字符类型时, 则执行当作 shell 执行
    if(typeof task == 'string'){
        setInterval(function(){
            var spawn = require('child_process').spawn;
            var shell = spawn(task);
            shell.stdout.on('data', function (data) {
                console.log('stdout: ' + data);
            });
        }, sec);
    }
}

```

【代码说明】

- ❑ sec: 隔断时间 (单位毫秒)。
- ❑ task: 定时任务模型。
- ❑ if(typeof task == 'function'): 判断任务类型是否为函数。
- ❑ if(typeof task == 'string'): 判断任务类型是否为 shell 字符。
- ❑ setInterval: 应用 setInterval 执行定时任务。
- ❑ var spawn = require('child_process').spawn: 应用 child_process 模块执行 shell 任务。

上面我们设计了两种任务模式, 分别是函数定时执行方法和 shell 脚本执行方法。函数执行任务方法可以直接通过 Node.js 代码来调用。shell 脚本执行方法, 则需要应用 Node.js 的 child_process 模块执行 Linux 中的 shell 命令。以上代码中, 通过使用 typeof 来区分任务模型, 如果 typeof 返回的是字符串, 则系统认为是 shell 命令, 如果 typeof 返回的是 function, 则系统认为是本地函数任务模型。

定点任务原理是每隔一秒去扫描一个任务表, 如果当前时间点等于某条任务执行时间点时, 则根据任务模型来执行该任务。我们将所有需要定点执行的任务列表, 存储在本模块的一个全局变量 TASK_ARR 中。在 time_task 函数中为全局变量 TASK_ARR 新增任务列表, 最后通过 setInterval 定时扫描整个任务数组, 任务到点时, 则执行该任务。以下是为 TASK_ARR 新增任务的实现, 代码如下:

```

var TASK_ARR = [];
module.exports = {
    time_task : function(){
        // time=12:12:00
        var time      = arguments[0]
            , task     = arguments[1]
            , splitArr = time.split(':')
            , hour     = splitArr[0]
            , minute   = splitArr[1]

```



```

    , sec      = splitArr[2] ? splitArr[2] : '00'
    , taskObj  = {
      'hour'   : parseInt(hour),
      'minute' : parseInt(minute),
      'sec'    : parseInt(sec),
      'task'   : task
    };
    TASK_ARR[TASK_ARR.length] = taskObj;
  },
  circle_task : function(){
  }
}

```

【代码说明】

- ❑ `splitArr=time.split(':')`: 使用:分割字符, 获取其中的 `hour`、`minute` 和 `second`。
- ❑ `taskObj`: 任务对象, 包括执行 `hour`、`minute`、`second` 和任务对象信息。
- ❑ `TASK_ARR[TASK_ARR.length]=taskObj`: 新增任务列表。

`time_task` 函数的目的只是为 `TASK_ARR` 任务列表中新增定时任务, 有点类似我们向 Linux 的 `crontab` 中新增任务列表。在 `time_task` 函数中新增完任务以后, 会通过 `setInterval` 每秒去扫描一次任务, 当时间到时则执行该任务。下面是 `setInterval` 的实现代码:

```

var TASK_ARR = [];

module.exports = {
  time_task : function(){
  },
  circle_task : function(){
  }
}

// 定点任务
setInterval(function(){
  var now= new Date()
  , hour=now.getHours()
  , minute=now.getMinutes()
  , second=now.getSeconds();

  for(var i=0; i<TASK_ARR.length; i++){
    var timeTask = TASK_ARR[i];
    if(hour == timeTask['hour'] && minute == timeTask['minute'] &&
second == timeTask['sec']){
      if(typeof timeTask['task'] == 'function'){
        timeTask['task'].call();
      } else if(typeof timeTask['task'] == 'string'){
        var spawn = require('child_process').spawn;
        var shell = spawn(timeTask['task']);
        shell.stdout.setEncoding('utf8');
        shell.stdout.on('data', function (data) {
          console.log('stdout: ' + data);
        });
      }
    }
  }
}, 1000);

```

【代码说明】

- ❑ `now= new Date()`: 获取当前时间。

- ❑ `now.getHours()`、`now.getMinutes()`、`now.getSeconds()`: 对应当前时间的小时、分钟和秒。
- ❑ `for(var i=0; i<TASK_ARR.length; i++)`: 循环扫描任务列表。
- ❑ `if(hour === timeTask['hour'] &&...)`: 判断任务的执行时间是否为当前时间。

定时扫描任务列表，并判断时间是否符合，如果符合则执行任务。以上就是 `crontab` 模块的实现，接下来我们看一下该模块的应用。

11.4.3 crontab 模块应用

该模块的应用非常简单，下面是示例代码。代码如下：

```
/* index.js */
var crontab = require('./crontab');
//crontab.circle_task(3000, show);
//crontab.circle_task(3000, doIt);

crontab.time_task('12:24:00', show);
crontab.time_task('12:24:00', doIt);

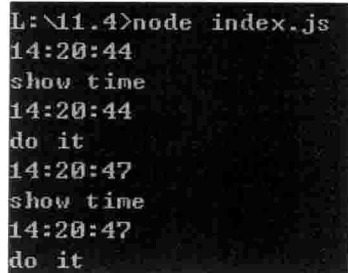
function show(){
    console.log('show time');
}

function doIt(){
    console.log('do it');
}
```

【代码说明】

- ❑ `crontab = require('./crontab')`: require 该 `crontab` 模块。
- ❑ `crontab.circle_task(3000, show)`: 该接口的时间单位是毫秒，因此这里是 3 秒执行一次函数任务 `show`。
- ❑ `crontab.circle_task(3000, doIt)`: 3 秒执行一次函数任务 `doIt`。
- ❑ `crontab.time_task('12:24:00', show)`: 每天的 12:24:00 秒执行函数任务 `show`。
- ❑ `crontab.time_task('12:24:00', doIt)`: 每天的 12:24:00 秒执行函数任务 `doIt`。

上面是两个接口的应用方法举例，在编写实例代码时，最好是填写一个合适的时间点，以便能够验证定时任务的正确性。运行 `index.js` 可以看到，每隔 3 秒会执行一次 `show` 和 `doIt` 任务，同时在定点功能函数中也执行了 `show` 和 `doIt` 两个函数任务。运行结果如图 11-8 所示。



```
L:\11.4>node index.js
14:20:44
show time
14:20:44
do it
14:20:47
show time
14:20:47
do it
```

图 11-8 定时脚本运行结果

为了更清晰地说明隔断时间执行任务，我们在 `setInterval` 的回调函数中添加一个时间打印：

```
var now= new Date();
console.log('' + now.getHours() + ':' + now.getMinutes() + ':' +
now.getSeconds());
```

因此，在图 11-8 中可以看到，每隔 3 秒钟就会打印显示 show time 和 do it 信息。

在实际项目中，可能需要在某个时刻清空数据、某个时刻去更新一些数据或者每秒、每分、每小时扫描某些数据等，这些都需要应用 crontab 来做定时任务。

11.5 forever 运行脚本

forever 是一个 Node.js 的命令行工具，主要是 Node.js 启动运行的一个工具。Node.js 是一个独立的 server，而其脚本的运行方式是应用 node script.js 方法，这种简单的脚本运行方式，并不能作为系统的项目运营。主要原因在于该方法不稳定，并且可运维程度较低，因此使用 forever 工具实现一个简单的脚本来管理 Node.js 服务器进程的关闭和启动，同时可以起到对服务运行进行监控的作用。

11.5.1 forever 运行脚本介绍

本节需要实现的是一个 shell 脚本，因此本节会涉及部分 shell 的语法。该脚本的作用包含 Node.js 的进程管理和 forever 运行日志管理。如图 11-9 所示为我们希望实现的该脚本的输入输出流程图。

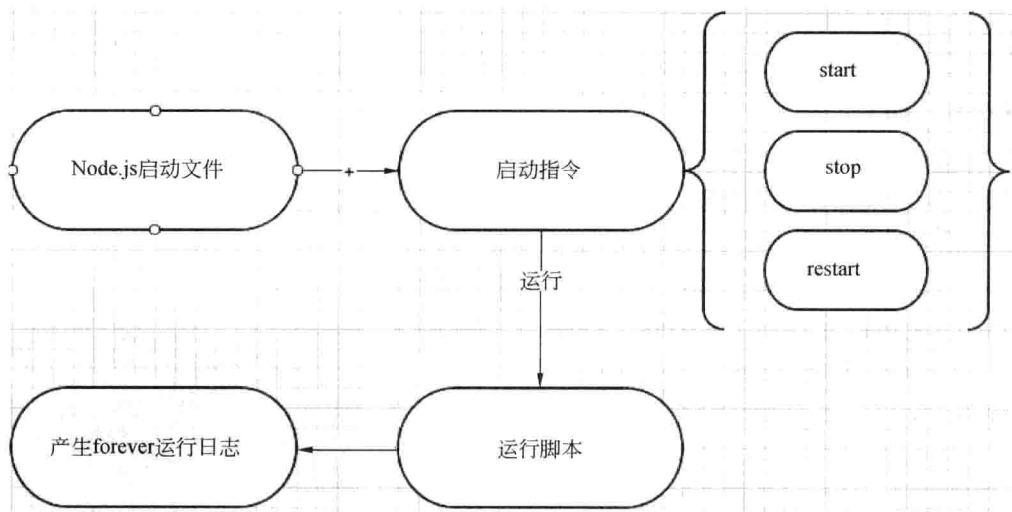


图 11-9 forever 脚本结构

图 11-9 主要是告诉我们这个脚本需要两个参数，Node.js 的启动文件路径和运行方式（启动、关闭和重新启动），最后运行方式可以理解为如下指令：

```
./node_server.sh /data/node_project/app.js start
./node_server.sh /data/node_project/app.js stop
./node_server.sh /data/node_project/app.js restart
```


以上 3 个指令 start、stop 和 restart 分别对应项目启动、项目停止和重新启动。

为了更好地查看项目的运行情况，在脚本中我们将相应 forever 运行日志存放在特定的

文件夹（例如/data/node_log/）。下面我们看 shell 脚本如何实现该功能。

11.5.2 forever 运行脚本实现

在介绍该脚本实现前，首先介绍一些关于 shell 脚本的知识。在 Linux 脚本中，首行会指定解析该脚本的方式，本 forever 脚本功能是应用#!/bin/sh 来指定解析该 shell 脚本的方式。

 **注意：**如果希望使用源码进行测试使用的话，请将 node_server 脚本放入/usr/bin 目录下。

本脚本应用到 shell 的条件判断语句——if/else。以下是该语法的示例。

```
if[ conditions ] ;then
    ....
elif[ conditions ] ;then
    ....
else
    ....
fi
```

在该脚本使用该语法来判断执行的命令是 start、stop 还是 restart 或者其他无效命令。脚本 3 个参数分别对应 shell 脚本的\$1、\$2 和\$3，同时需要初始化项目运行日志文件夹。相关代码如下：

```
#!/bin/sh
app=$1
action=$2
project=$3
mkdir -p /data/node_log/${project}
```

【代码说明】

- ❑ app=\$1：项目启动文件路径，请使用绝对路径，这里没有支持相对路径的方法。
- ❑ action=\$2：命令类型有 start、stop 和 restart 3 种。
- ❑ project=\$3：项目名，用于初始化 forever 日志文件夹。
- ❑ mkdir -p /data/node_log/\${project}：初始化 forever 运行日志文件夹。

本段代码主要是获取脚本执行参数，同时初始化 forever 运行的 debug、info 和 error 日志。下面来看 start 命令的实现，代码如下：

```
# start server
if [ "$action" == "start" ] ;then
echo `/usr/bin/node_server/forever/bin/forever start -a -l /data/node_log/
${project}/forever.log -o /data/node_log/${project}/out.log -e
/data/node_log/${project}/err.log ${app}`
echo "server start!"
```

【代码说明】

- ❑ if ["\$action" == "start"]：判断是否为 start 执行指令。
- ❑ /usr/bin/node_server/forever/bin/forever：该路径是 forever 的可执行文件路径。
- ❑ -a -l -o -e：分别对应 forever 的运行指令。
- ❑ \${app}：项目启动文件绝对路径。


上面就是应用 `node_server` 来启动一个服务的相关代码，细心的读者有没有发现一个异常问题？如果我们两次运行该脚本命令时，会导致同时启动两个进程，其中这两个进程中的一个会出现异常情况，而另一个则会正常运行。因此这里必须要做一定的判断，避免这种情况的发生。下面是该异常处理的解决代码：

```
# start server
if [ "$action" == "start" ] ;then
idppstr=`ps -ef | grep "/usr/local/bin/node ${app}"`
str=`echo $idppstr| awk -F ' ' '{print $8}'`
# check server exist
if [ "$str" == "grep" ] ;then
echo `/usr/bin/node_server/forever/bin/forever start -a -l /data/node_log/
${project}/forever.log -o /data/node_log/${project}/out.log -e /data/node_
log/${project}/err.log ${app}`
echo "server start!"
else
echo "exist this server"
fi
```

【代码说明】

- ❑ `idppstr=`ps -ef | grep "/usr/local/bin/node ${app}"``：查询该进程是否存在。
- ❑ `str=`echo $idppstr| awk -F ' ' '{print $8}'``：获取查询后的第八个字符。
- ❑ `if ["$str" == "grep"] ;then`：判断该字符是否为 `grep`，去除 `grep` 产生的进程的查询结果，避免影响查询 `Node.js` 进程。

在以上代码中通过使用 `ps -ef` 来查询该进程。需要注意的是，在 `ps -ef` 的时候会返回一条 `grep` 进程字符数据，该进程是在查看字符的时候产生的，因此需要将该进程排除在外，如图 11-10 所示。



```
root@ubuntu:/usr/bin/node_server# ps -ef | grep node
root      2257      1   0 15:29 ?        00:00:01 /usr/local/bin/node /usr
root      2261    2257   0 15:29 ?        00:00:00 /usr/local/bin/node /ho
root      2273    1178   0 15:34 pts/0    00:00:00 grep --color=auto
```

图 11-10 ps 查询结果

为了区分该进程是否为 `grep` 时，我们使用 `awk` 获取该返回字符串的第 8 个字符是否为 `grep`，如果不是 `grep`，就表明该进程存在，否则进程不存在。进程存在时无需再次启动，直接返回；不存在该进程时则启动进程。

接下来我们再看 `stop` 命令的实现，其原理是：应用 `ps -ef` 查询进程 `forever` 产生的两个进程，通过字符串分析获取两个进程的 `id`，最后再使用 `kill -9` 将两个进程杀死。代码如下：

```
# stop server
elif [ "$action" == "stop" ] ;then
idstr=`ps -ef | grep "/usr/local/bin/node /usr/bin/node_server/forever/
bin/monitor ${app}"`
idppstr=`ps -ef | grep "/usr/local/bin/node ${app}"`
# find server id
str=`echo $idstr| awk -F ' ' '{print $8}'`
forever_id=`echo $idstr | awk -F ' ' '{print $2}'`
pp_id=`echo $idppstr| awk -F ' ' '{print $2}'`
# check server exist
if [ "$str" == "grep" ] ;then
```

```

echo "can not find this server id"
exit
fi

kill -9 $forever_id $pp_id
echo "server stop!"

```

【代码说明】

- ❑ `elif ["$action" == "stop"];then`: 判断是否为 stop 指令。
- ❑ `idstr=`ps -ef | grep...``: 应用 `ps -ef` 查询 forever 进程。
- ❑ `idppstr=`ps -ef...``: `ps -ef` 查询项目运行进程。
- ❑ `str=`echo $idstr...``: `awk` 获取进程查询后的第 8 个字符。
- ❑ `forever_id=`echo $idstr...``: `awk` 获取 forever 的进程 id。
- ❑ `pp_id=`echo $idppstr...``: `awk` 获取项目进程的 id。
- ❑ `if ["$str" == "grep"]`: 判断查询后返回的命令信息是否为 `grep` 产生的进程。
- ❑ `kill -9 $forever_id $pp_id`: 杀死查询后的 forever 和项目运行进程。

使用 forever 运行项目的时候会产生两个进程，如图 11-11 所示。

```

ps -ef | grep node
00:00:01 /usr/local/bin/node /usr/bin/node_server/forever/bin/monitor /home/danhuang/static_css/http.js
00:00:00 /usr/local/bin/node /home/danhuang/static_css/http.js

```

图 11-11 ps 查询 Node.js 进程

图 11-11 中的第一个进程是 forever 监控进程，而第二个则是项目启动运行进程，因此我们要终止进程时，需要同时 kill 两个进程。

应用 `ps -ef` 查看进程，`awk` 分析查询后的返回字符串，获取其中的进程 id 值和进程的相关描述。在 `ps -ef` 的返回进程中，和启动进程时查询一样，也会包含一个 `grep` 进程，因此这里需要对进程的描述做一个判断，如果不是 `grep` 进程才进行 kill。

最后一个就是 `restart` 指令，该指令的作用就是 kill 进程，然后重新运行。相关代码如下：

```

# restart server
elif [ "$action" == "restart" ];then
idstr=`ps -ef | grep "/usr/local/bin/node /usr/bin/node_server/forever/
bin/monitor ${app}"`
idppstr=`ps -ef | grep "/usr/local/bin/node ${app}"`
str=`echo $idstr | awk -F ' ' '{print $8}'`
forever_id=`echo $idstr | awk -F ' ' '{print $2}'`
pp_id=`echo $idppstr | awk -F ' ' '{print $2}'`

if [ "$str" == "grep" ];then
echo "can not find this server id"
exit
fi

kill -9 $forever_id $pp_id
echo ` /usr/bin/node_server/forever/bin/forever start -a -l /data/node_log/
${project}/forever.log -o /data/node_log/${project}/out.log -e
/data/node_log/${project}/err.log ${app}`
echo "restart success"

```

脚本提供了 3 个执行指令方法 `start`、`stop` 和 `restart`，对于其他的操作提示相应的错误信息即可。如下是其他指令的返回结果：


```
else
echo "node_server [nodeapp] [command] [project name]\n [command] below\n\t
start: for start server\n\t stop: for stop server \n\t restart: for restart
server"
```

主要是返回一个提示信息，提示的内容如下：

```
node_server [nodeapp] [command] [project name]
[command] below
start: for start server
stop: for stop server
restart: for restart server
```

以上就是我们的 forever 运行脚本。下面来看本脚本的一些简单应用。

11.5.3 forever 运行脚本应用

在应用该模块前，首先需要将该模块放入系统的/usr/bin 目录下。成功放入以后我们就可以来简单执行一个错误的指令：

```
/usr/bin/node_server/node_server.sh a
```

```
node_server [nodeapp] [command] [project name]
[command] below
start: for start server
stop: for stop server
restart: for restart server
```

图 11-12 脚本 help 信息

如果传递的非 start、stop 和 restart 命令时，则返回提示信息，如图 11-12 所示。

接下来我们使用该脚本运行一个正常的 Node.js 服务，执行如下命令。

```
/usr/bin/node_server/node_server.sh /home/danhuang/static_css/http.js
start http
```

执行如上指令后脚本返回如图 11-13 所示的信息。

```
root@ubuntu:/usr/bin/node_server# /usr/bin/node_server/node_server
info: Forever processing file: /home/danhuang/static_css/http.js
server start!
```

图 11-13 执行脚本返回结果

图 11-13 表明已经成功启动该 http.js 文件脚本。那么根据设计思想，会在/data/node_log 文件夹下产生 HTTP 的相关操作日志，进入该/data/node_log/http 可以看到如图 11-14 所示的文件。

```
root@ubuntu:/data/node_log/http# pwd
/data/node_log/http
root@ubuntu:/data/node_log/http# ls
err.log  forever.log  out.log
```

图 11-14 forever 脚本日志目录文件列表

图 11-4 表明已经成功运行该 Node.js 脚本。那么我们再次运行脚本时，是否会提示相应的信息呢？接下来我们再次运行如下指令：


```
/usr/bin/node_server/node_server.sh    /home/danhuang/static_css/http.js
start http
```

可以看到这次返回的是如图 11-15 所示的信息。

```
err.log  forever.log  out.log
root@ubuntu:/data/node_log/http# /usr/bin/node_server/node_server.sh
exist this server
```

图 11-15 多次启动同一个脚本执行返回

图 11-15 提示我们已经存在该进程服务，这样就很好地保证多次运行脚本只启动一次同样的 Node.js 服务。

接下来我们 restart 一次该 Node.js 服务，执行如下指令：

```
/usr/bin/node_server/node_server.sh    /home/danhuang/static_css/http.js
restart http
```

返回如图 11-16 所示的结果表明已成功重启服务。

```
root@ubuntu:/data/node_log/http# /usr/bin/node_server/node_server
info: Forever processing file: /home/danhuang/static_css/http.js
restart success
```

图 11-16 restart 启动脚本返回

最后我们看一下该脚本的 stop 使用方法。

```
/usr/bin/node_server/node_server.sh    /home/danhuang/static_css/http.js
stop http
```

运行指令后脚本返回 server stop 信息，如图 11-17 所示。我们通过 ps -ef | grep node 查询是否还有该进程。查询后返回的只有一个 grep 进程，HTTP 的两个进程已经成功的 kill 了，如图 11-18 所示。

```
restart success
root@ubuntu:/data/node_log/http# /usr/bin/node_server/node_server.sh
server stop!
```

图 11-17 stop 命令执行返回

```
server stop!
root@ubuntu:/data/node_log/http# ps -ef | grep node
root      2519  1178  0 16:43 pts/0    00:00:00 grep --color=auto
root@ubuntu:/data/node_log/http#
```

图 11-18 ps 查询 node 进程执行返回

本节介绍了该脚本的应用实现。该脚本也可以作为项目的实践应用，本节只是一个简单的代码实践，在实际项目中希望读者能够根据自己的项目来完善该脚本。

11.6 xml 模块的应用

项目开发中需要使用到 json 配置文件，当然也可能在项目中应用到 xml 的配置文件，因此这里我们实现一个简单的 xml 解析模块工具。在 Node.js 的第三方库中有很多开源的解析工具，本节应用的是在 github 上发布的 xml2js 工具模块。本章的模块实践开发，只是

在 xml2js 第三方库的基础上做了部分的封装。

11.6.1 xml 解析模块介绍

本节介绍如何应用 xml2js 封装一个简单的 xml 解析模块。本节介绍两种方式的 xml 解析，分别是含有属性值的 xml 和不含属性值的 xml 解析方法。

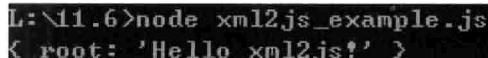
在学习本模块前首先需要下载 NPM 的 xml2js 模块包，执行 `npm install xml2js`，源码中已经安装了该模块。该模块的应用非常简单，下面是该模块的一个实例，代码如下：

```
var parseString = require('xml2js').parseString;
var xml = "<root>Hello xml2js!</root>";
parseString(xml, function (err, result) {
  console.dir(result);
});
```

【代码说明】

- ❑ `require('xml2js').parseString`: `require` xml2js 模块并调用 `parseString` 方法。
- ❑ `xml = "<root>Hello xml2js!</root>"`: xml 字符串。
- ❑ `parseString(xml...)`: 解析 xml 字符。
- ❑ `console.dir(result)`: 打印返回对象。

执行该段代码后，可以发现其返回的是如图 11-19 所示结果。



```
L:\11.6>node xml2js_example.js
{ root: 'Hello xml2js!' }
```

图 11-19 xml 解析结果

下面我们只需要对该模块的应用做一个简单的封装，其中包括解析带属性的 xml 文件和不带属性 xml 的两种方法。xml 解析也需要做一个数据缓存的功能，避免每次都去读取磁盘文件，代码如下：

```
var XML_TMP_DATA;
module.exports = {
  /**
   * 解析 xml 模块
   */
  parse : function(xmlPath){

  }
}
/* 添加缓存信息 */
function cacheXml(){
}
/* 获取缓存数据 */
function getCache(){
}
```

【代码说明】

- ❑ `XML_TMP_DATA`: 数据缓存对象。
- ❑ `parse : function(xmlPath)`: xml 解析模块。

- ❑ cacheXml: 缓存 xml 数据。
- ❑ getCache: 获取缓存数据。

为了方便我们只暴露一个外部接口 `parse` 方法。下面来看一下该模块的几个方法的实现。

11.6.2 xml 模块设计实现

本模块需要实现的主要是 `parse` 模块，而缓存原理和 11.1.2 节介绍的配置文件读取模块方法类似。应用 Node.js 中的 `fs.exists` 对象判断文件是否存在，如果存在，则应用 Node.js 中的原生 API `fs.readFile` 来取 xml 内容，最后通过 `xml2js` 进行解析。

`fs.exists` 判断文件是否存在，代码如下：

```
var fs = require('fs');
fs.exists(xmlPath, function (existBool) {
  if(existBool){
    // do it
  }
});
```

文件存在后，然后再应用 `fs.readFile` 读取 xml 内容，并应用 `xml2js` 进行解析解析，代码如下：

```
var xml2js = require('xml2js');
var parseString = xml2js.parseString;
var xmlInfo = fs.readFileSync(xmlPath);
parseString(xmlInfo, function (err, result) {
  if(err){
    retInfo['code'] = -1;
  } else {
    retInfo['code'] = 0;
    retInfo['data'] = result;
  }
  callback(retInfo);
});
```

【代码说明】

- ❑ `xml2js = require('xml2js')`: 获取 `xml2js` 模块。
- ❑ `parseString = xml2js.parseString`: 获取 `xml2js` 对象的 `parseString` 方法。
- ❑ `xmlInfo = fs.readFileSync(xmlPath)`: 同步读取 xml 文件数据。
- ❑ `parseString(xmlInfo, function (err, result)...`: 异步解析 xml 字符串数据。
- ❑ `callback(retInfo)`: 应用 `callback` 返回异步执行结果。

由于 `xml2js` 的 `parseString` 方法是一个异步方法，因此必须通过回调函数 `callback` 将结果进行返回。同时为了更好地处理结果，使用错误码来区分解析 xml 错误的原因。

该模块的数据缓存方法和配置文件模块的缓存方式相同，缓存方法如下：

```
function cache(filePath, fileName, data){
  var stat = fs.statSync(filePath);
  var timestamp = Date.parse(stat['mtime']);
  if(data){
    XML_TMP_DATA[fileName+timestamp] = data;
  }
}
```

```
}
```

下面是读取缓存数据的方法，代码如下：

```
function getCache(filePath, fileName, key){
    var stat = fs.statSync(filePath);
    var timestamp = Date.parse(stat['mtime']);
    if(XML_TMP_DATA[fileName+timestamp]){
        return XML_TMP_DATA[fileName+timestamp][key]
    }
    XML_TMP_DATA[fileName+timestamp][key] : XML_TMP_DATA[fileName+timestamp];
    return null;
}
```

既然现在有了缓存数据，那么在读取 xml 文件数据前，先进行缓存检查，如果存在缓存则无需读取 xml 文件信息。修改 parse 方法如下：

```
parse : function(xmlPath, pathname, callback){
    var retInfo = {};
    // 读取缓存数据
    var cacheData = getCache(filePath, fileName);
    if(cacheData){
        retInfo['code'] = 0;
        retInfo['data'] = cacheData;
        callback(retInfo);
        return;
    }

    var xml2js = require('xml2js');
    var parseString = xml2js.parseString;
    fs.exists(xmlPath, function (existBool) {
        if(existBool){
            var xmlInfo = fs.readFileSync(xmlPath);
            parseString(xmlInfo, function (err, result) {
                if(err){
                    retInfo['code'] = -1;
                } else {
                    retInfo['code'] = 0;
                    retInfo['data'] = result;
                    cache(filePath, fileName, result);
                }
                callback(retInfo);
                return;
            });
        } else {
            retInfo['code'] = -2;
            retInfo['data'] = {};
            callback(retInfo);
        }
    });
}
```

以上代码主要是在原来基础上增加了缓存处理部分。首先会去缓存中查询数据，如果没有则读取 xml 文件数据，并将数据进行缓存。下面介绍该模块的应用方法。

11.6.3 xml 模块应用

在介绍实例应用前，首先要创建多个 xml 用来做测试，这里选择了两种 xml，分别是带属性值的和不带属性的。我们先来看不带属性值的 xml 解析示例，xml 如下：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note>
  <to>George</to>
  <from>John</from>
  <heading>Reminder</heading>
  <body>Don't forget the meeting!</body>
</note>
```

接下来我们创建一个 index.js 应用 xml 模块来解析该 xml 文件，代码如下：

```
var xml = require('./xml');
xml.parse('xml/keyvalue.xml', 'keyvalue.xml', function(ret){
  console.log(ret);
});
```

运行测试代码，得到如图 11-20 所示的返回信息。

```
H:\11.6>node index.js
{ code: 0,
  data:
    { note:
      { to: [Object],
        from: [Object],
        heading: [Object],
        body: [Object] } } }
```

图 11-20 xml 解析结果

在图 11-20 中我们可以看到一个现象，返回的数据中很多键值都是一个 Object 对象，例如其中的 to 对应的键值和 from 对应的键值。将其中的一个 Object 打印出来后，发现这个 Object 其实是一个数组对象，如图 11-21 返回的 ['George']，而数组元素就是 xml 中标签对应的值。但在实际应用中，需要的往往是一个元素而不是数组，因此在应用过程中，如果是这种没有属性值的 xml 文件解析后，需要应用其键值时，必须从数组中取出该元素。

```
H:\11.6>node index.js
{ code: 0,
  data:
    { note:
      { to: [Object],
        from: [Object],
        heading: [Object],
        body: [Object] } } }
[ 'George' ]
```

图 11-21 xml 解析结果

例如，下面是将所有的键值打印出来的实例代码：

```
var xml = require('./xml');
xml.parse('xml/keyvalue.xml', 'keyvalue.xml', function(ret){
  console.log('to:', ret['data']['note']['to'][0]);
  console.log('from:', ret['data']['note']['from'][0]);
  console.log('heading:', ret['data']['note']['heading'][0]);
  console.log('body:', ret['data']['note']['body'][0]);
});
```

执行代码后，可以看到这些 xml 标签对应的所有键值，如图 11-22 所示。

```
H:\11.6>node index.js
to: George
from: John
heading: Reminder
body: Don't forget the meeting!
```

图 11-22 xml 解析结果

以上是不带属性的 xml 解析方法。需要注意的是，所有的键值解析返回后都是一个数组对象，因此在应用 xml 解析后的键值时，需要读取其数组的首个元素。接下来我们再看一下带属性的 xml 解析过程中需要注意的一些问题。

所谓的带属性的 xml 就是在标签中添加了一些属性值，例如下面的 xml 格式：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note>
  <to sex='man'>George</to>
  <from sex='female' age='23'>John</from>
  <heading>Reminder</heading>
  <body time='2013-04-01'>Don't forget the meeting!</body>
</note>
```

相对前面简单的 xml，这里为 3 个标签添加了属性，主要是为了辅助说明该标签对应的键值。接下来使用如下测试代码 index_attr.js 来查看返回的数据结构：

```
var xml = require('./xml');
xml.parse('xml/attribute.xml', 'attribute.xml', function(ret){
  console.log(ret);
});
```

执行代码后，可以看到如图 11-23 所示的结果。

```
H:\11.6>node index_attr.js
{ code: 0,
  data:
    { note:
      { to: [Object],
        from: [Object],
        heading: [Object],
        body: [Object] } } }
```

图 11-23 xml 解析结果

返回的结果和前面的没有属性的 xml 返回的数据结构是完全一致的。接下来打印 to 对

应的数组元素，在 index_attr.js 中添加如下代码，重新运行 index_attr.js 后，返回结果如图 11-24 所示。

```
H:\11.6>node index_attr.js
{ code: 0,
  data:
    { note:
      { to: [Object],
        from: [Object],
        heading: [Object],
        body: [Object] } } }
to: [ { _: 'George', '$': { sex: 'man' } } ]
```

图 11-24 xml 解析结果

这次返回的并非一个数组元素，而是一个对象，该对象中包含两个键值，分别是_和\$。_代表是键值，而\$代表的是所有的属性值，_是一个元素，而\$则是一个对象，可能包含多个属性值。

接下来我们使用如下代码将该 xml 的所有属性值以及键值打印显示：

```
var xml = require('./xml');
xml.parse('xml/attribute.xml', 'attribute.xml', function(ret){
  console.log('to:', ret['data']['note']['to'][0]['_'], ' sex:',
ret['data']['note']['to'][0]['$']['sex']);
  console.log('from:', ret['data']['note']['from'][0]['_'], ' sex:',
ret['data']['note']['from'][0]['$']['sex'],
age:',ret['data']['note']['from'][0]['$']['age']);
  console.log('heading:', ret['data']['note']['heading'][0]);
  console.log('body:', ret['data']['note']['body'][0]['_'], ' time:',
ret['data']['note']['body'][0]['$']['time']);
});
```

执行如上代码，可以得到如图 11-25 所示的返回结果。

```
H:\11.6>node index_attr.js
to: George sex: man
from: John sex: female age: 23
heading: Reminder
body: Don't forget the meeting! time: 2013-04-01
```

图 11-25 xml 解析结果

从应用层面上讲，在解析带属性的 xml 后，获取其返回结果比较复杂，为了得到其键值，我们共使用了 5 个键，例如：ret['data']['note']['to'][0]['_']。因此在实际应用过程中，希望读者能够分析出，应用该模块解析返回的 xml 对象的数据结构。

【小牛试刀】

本节应用到了 xml 解析模块，在学习本节后，应用该模块解析如下两个 xml 文件信息，并将其中所有的键值打印出来。

该文件是一个不含属性的 xml 配置文件，请参照 xml 的解析方法一。

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<book>
```



```
<name>Node</name>
<author>somebody</author>
<desc>introduce node</desc>
</note>
```

该文件是一个含属性的 xml 配置文件，请参照 xml 的解析方法二。

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<book>
  <name country="china">Node</name>
  <author sex="man">somebody</author>
  <desc>introduce node</desc>
</note>
```

11.7 邮件发送模块应用

发送邮件的功能是如何实现的？如何才能应用 Node.js 发送电子邮件呢？这是本节主要关注的问题。本节主要从模块的原理、实现方法，以及应用三个方面来做介绍。

11.7.1 邮件模块介绍

我们经常会应用邮件功能来验证用户信息，也可以来做一些产品的推广功能，或者推送一些用户信息到用户邮箱等。因此在项目的实际开发过程中，邮件功能模块也是一个非常重要的模块，本节就简单阐述邮件功能模块的实现。

首先我们需要了解一下如何实现一个邮件发送功能。

- ❑ 指定发送者邮箱地址、密码和邮箱服务器；
- ❑ 通过指定的账号、密码，以及邮箱服务器验证用户信息；
- ❑ 成功验证通过后，根据传递的邮件内容，发送到指定用户邮箱地址。

简单的 3 个步骤中，我们需要哪些信息呢？用户邮箱账号、密码、该账号对应的服务器（例如 qq.com 的邮箱对应 smtp.qq.com，163 的邮箱对应 smtp.163.com 等）、邮箱收件人地址、邮件主题、邮箱内容（多种格式）或者抄送人邮箱地址。对于以上信息，我们可以将数据分为两类：固定数据和变动数据。固定数据可以作为邮件发送的配置文件进行存储，例如用户邮箱账号、密码、该账号对应的服务器。而变动数据则是每次发送邮件都会有所更改的。例如邮箱收件人地址、邮件主题、邮箱内容（多种格式）或者抄送人邮箱地址。接下来我们设计整个模块的实现流程，主要包含读取配置信息、验证发送账号信息和发送邮件。由于这个功能实现过程需要使用到配置文件，因此需要使用到本章中的配置文件读取模块。根据如上设计思想，下面来介绍该模块的实现。

11.7.2 邮件模块设计实现

本模块实现需要使用到第三方 github 开源的 NPM 模块 Nodemailer¹，该模块是一个易

¹ 参建网站 <https://github.com/andris9/Nodemailer>。

于使用的 Node.js 邮件模块，可以选择你喜欢的邮件类型（可以使用 SMTP、sendmail 或 Amazon SES）。该模块没有依赖编译，因此其可以在任何系统环境下使用。下载方法请执行 `npm install nodemailer`。

首先读取 Email 中的配置文件信息，并初始化 Nodemailer 对象数据。在查看代码前，我们先看一下 `email.json` 配置文件数据（创建文件夹 `conf`，然后在该文件夹中创建 `email.json` 文件），代码如下：

```
{
  "host" : "",
  "port" : 465,
  "user" : "huangdh3@gmail.com",
  "pass" : "*****",
  "service" : "Gmail"
}
```

该配置文件中包含了 Email 发送者的邮件服务器、端口、发送用户邮箱地址和该邮箱地址对应的账号和密码。接下来我们看一下读取配置文件内容，并初始化 Nodemailer 对象的代码实现：

```
var config = require('./config');
var nodemailer = require('nodemailer');

var emailConf = config.get('email.json', 'json');
var mail = nodemailer.createTransport("SMTP", {
  service: emailConf['service'],
  auth: {
    user: emailConf['user'],
    pass: emailConf['pass']
  }
});
```

【代码说明】

- ❑ `nodemailer = require('nodemailer')`: require 该模块；
- ❑ `emailConf = config.get('email.json', 'json')`: 读取 `email.json` 中的配置文件内容；
- ❑ `nodemailer.createTransport = ...`: 初始化 mail 对象数据。

最后我们来看一下主要的邮件发送功能的实现代码：

```
exports.sendMail = function(emailAddress, title, content){
  mail.sendMail(
    {
      sender:emailConf['host'],
      to : emailAddress,
      subject:title,
      html:content
    },function(error,success){
      if(!error){
        console.log('message success', success);
      }else{
        console.log('failed '+error);
      }
    })
};
```

【代码说明】

- ❑ sender:emailConf['host']: 发送邮件者名称;
- ❑ to : emailAddress: 接收邮件的用户的邮箱地址;
- ❑ html:content: 邮件内容。

sendMail 为外部调用接口, 其中包含 3 个参数 emailAddress、subject 和 content, 分别表示收件箱、地址, 邮件主题和邮件 html 内容。实践完成该模块后, 我们来介绍该模块的使用方法。

11.7.3 邮件模块应用

在使用该模块时, 要将配置文件中的数据进行更改, 否则无法正常使用。在该配置文件中笔者使用的是个人的 Gmail 邮箱作为测试, 通过 Gmail 邮箱发送数据到 QQ 邮箱 492383469 中。测试代码如下:

```
/* index.js */
var email = require('./email');
var emailAddress = '492383469@qq.com';
var subject = 'test for email module';
var content = '<html><head><title>test</title></head><body><div>Node.js book test</div></body></html>';
email.sendMail(emailAddress, subject, content);
```

执行该 index.js 脚本后, 在执行窗口可以看到如图 11-26 所示的返回信息。

```
L:\11.7>node index.js
message success < failedRecipients: [],
  message: '250 2.0.0 OK 1365240120 g8sm17196684pae.7 - gsmtip',
  messageId: '1365240116997.4ca2b0d@Nodemailer' >
```

图 11-26 Email 执行结果

图 11-26 提示已经发送成功。那么接下来打开 QQ 邮箱, 可以查看到如图 11-27 所示的邮件信息。

test for email module ☆

发件人: huangdh3 <huangdh3@gmail.com> 图

时 间: 2013年4月6日(星期六) 凌晨2:21 (UTC-07:00 休斯顿、底特律时间)

收件人: 492383469 <492383469@qq.com>

Node.js book test

图 11-27 Email 邮件内容

可以看到我们已经成功的应用 huangdh3@gmail.com 发送电子邮件到 492383469 邮箱账号。以上就是该邮件模块的实现。

Nodemailer 模块现在只支持如下邮箱作为 smtp 服务发送邮件到其他邮箱。

DynectEmail、Gmail、hot.ee、Hotmail、iCloud、mail.ee、Mail.Ru、Mailgun、Mandrill、Postmark、SendGrid、SES、Yahoo、yandex 和 Zoho。

因此在国内，最好就使用 Gmail 作为服务邮箱来发送电子邮件。

11.8 本章小结

本章主要介绍了 7 个实用的 Web 开发工具，在本章的学习中，需要大家掌握这些模块的实现原理，而不是简单满足于模块的应用。以下是本章中需要读者了解和掌握的所有模块的知识。

日志模块：掌握 log4js 模块的应用、日志模块的设计方法和日志模块的应用方法；

配置文件模块：掌握数据缓存方式、conf 配置文件和 json 配置解析方法；

curl 模块：掌握 curl 中的 request 模块的应用，以及可以应用 curl 模块发起 HTTP 的 GET 或者 POST 请求；

crontab 模块：掌握 Node.js 中 setTimeout 的使用、Node.js 中应用 setTimeout 替换 Linux 中 crontab 的实现方法，以及 crontab 模块的应用；

forever 模块：了解 forever 模块的应用、shell 脚本简单编程，掌握 forever 的脚本的实现原理，会应用该脚本来启动运行项目；

xml 模块：了解 xml2js 模块的应用，掌握 xml 模块解析和缓存的实现；

邮件发送模块：掌握 Nodemailer 模块的应用，学会应用 Gmail 来发送电子邮件。

这些模块大部分都是基于 github 上的开源模块进行封装实现的，很多工具模块类都无需自我开发。在实际项目开发过程中，如果需要一些应用模块时，可以在 github 上搜索开源的 NPM 模块，然后基于自己的项目进行一个简单的封装。

下面是本章的应用模块汇总，如表 11.1 所示。

表 11.1 应用模块汇总

模块名	应用场景	相关 NPM 模块
日志模块	项目的日志打印	log4js
配置文件模块	配置文件的读取	无
curl 模块	发起 HTTP 请求	request
crontab 模块	定时定点任务	无
forever 模块	项目启动运行脚本	forever
xml 模块	解析 xml 文件	xml2js
邮件发送模块	Node.js 邮件发送功能	Nodemailer

[General Information]

书名=Node.js开发实战详解

作者=黄丹华编著

丛书名=Web开发典藏大系

页数=377

SS号=13495646

出版日期=2014.04

出版社=北京：清华大学出版社

ISBN号=978-7-302-34947-1

中图法分类号=TP312

原书定价=59.80

参考文献格式=黄丹华编著.Node.js开发实战详解.北京：清华大学出版社,2014.04.

内容提要=本书介绍了Node.js的环境构建和基础知识、模块与设计模式、Web应用、网络编程实践、深入探究、Node.js与数据库之间的交互、Node.js中编码规范等。重点是Node.js的Web应用，介绍如何用Node.js原生代码(不借助于框架)构建一个Web服务器，包括如何处理：静态资源、路由解析、session管理和MVC模块搭建等。