

Kenneth Kan

Pixbi 联合创始人、CTO

何翊宇 (dead_horse)

天猫前端技术专家

倾情力荐



Node.js 实战

(第2季)

— 吴中骅 雷宗民 赵坤 刘亚中 著 —

Node.js 实战

(第2季)

The rapidly maturing JavaScript ecosystem is a blessing to those of us in the technology industry who constantly need to speed up development in response to change. The dynamic nature of JavaScript and the burgeoning community around it have propelled it to be the golden standard in “Fail Early, Fail Often” development, in which extremely fast iterations often give its adherents a competitive edge.

However, have you considered what lurks around behind those quick iterations? The unintended result of failing early and often is a potentially unmanageable mess that a fast, but unstructured, development workflow usually leaves behind. This book offers countermeasures to this often unconsidered issue.

Testing is one of the most overlooked aspects of development, yet it is one of the most foundational elements of any successful project. Sometimes it is left behind because “we just don’t have time”; other times it is because setting up proper testing is a time-consuming project in and of itself.

Fear no more. Yorkie is a veteran in putting in place automated test systems in the still relatively nascent world of Node.js. In our previous partnership at Pixbi, he single-handedly introduced proper testing to a startup that needed to change course on a weekly basis. Yorkie has consolidated the lessons that he has learned into a concise yet comprehensive book with a hands-on approach.

There is no reason to spend more of your precious time in learning what has already been learned. If you recognize that testing is fundamental to sustainable development, this is the book to have in your hands.

Kenneth Kan Pixbi 联合创始人、CTO

将你的想法运行在你的Node进程上

《Node.js 实战（第2季）》读者反馈网站：<http://nodejs.ucdok.com/>

《Node.js 实战（第2季）》官方QQ群：156627943



博文视点Broadview



@博文视点Broadview



策划编辑：张国霞
责任编辑：徐津平
封面设计：李玲

上架建议：Node.js

ISBN 978-7-121-27139-7



9 787121 271397 >

定价：59.00元

Node.js 实战

— 吴中骅 雷宗民 赵坤 刘亚中 著 —

(第2季)

電子工業出版社

Publishing House of Electronics Industry

北京•BEIJING

内 容 简 介

本书通过7个实例分别讲解了Node.js在实战开发中的应用,这些章节既涉及Docker、Koa等最新技术,也涉及OAuth2、命令行工具、消息队列、单元测试、编写C/C++模块等实战中经常会遇到的问题和应用场景。本书章节大体按照从简单到复杂的难度编排,每一章都通过一个实例指引读者从头开发一个Node.js应用,让读者循序渐进地学习Node.js,以及在实战开发中的编程技巧。本书不但着重讲解了每个实战案例所涉及的基础知识、思路和方法,也详细解释了源码的关键部分,希望有利于读者的学习和理解。

本书适合有一定Node.js基础及服务器端开发基础的读者阅读,也适合想了解Node.js可以做什么、想迅速上手实践的读者阅读。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有,侵权必究。

图书在版编目(CIP)数据

Node.js实战.第2季 / 吴中骅等著. —北京: 电子工业出版社, 2015.10
ISBN 978-7-121-27139-7

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

中国版本图书馆CIP数据核字(2015)第216018号

策划编辑: 张国霞

责任编辑: 徐津平

印 刷: 三河市双峰印刷装订有限公司

装 订: 三河市双峰印刷装订有限公司

出版发行: 电子工业出版社

北京市海淀区万寿路173信箱 邮编: 100036

开 本: 720×1000 1/16 印张: 19.25 字数: 340千字

版 次: 2014年5月第1版

2015年10月第2版

印 次: 2015年10月第1次印刷

印 数: 3000册 定价: 59.00元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888。

质量投诉请发邮件至zltz@phei.com.cn, 盗版侵权举报请发邮件至dbqq@phei.com.cn。

服务热线: (010) 88258888。

推荐序

写一本与Node.js实战相关的书是一项非常有挑战性的任务。众所周知，Node.js 自身现在正处于一个飞速发展的阶段，它的第三方库同样正处于一个爆发增长期，各种新兴技术和框架层出不穷，在短短几年里，它的第三方库的数量就已经超越了Java。因此，要想完成这个任务，除了需要扎实的技术功底，还需要常人难及的耐心和表达能力，才能够跟进社区的快速发展，并将知识同步给读者。

本书的几位作者都是Node.js 领域的牛人，不论是在社区、个人博客还是在GitHub 上，都一直在孜孜不倦地分享与Node.js 实战相关的经验和文章。他们不仅有扎实的技术功底，对于如何分享自己所掌握的知识也非常有经验。

本书在内容上同大部分技术入门书籍相比有所不同，它并没有花费太多篇幅来讲解具体的语言、框架等基础内容，而是通过7个实战项目来介绍Node.js，能够让读者直接感知到Node.js到底能做什么，以及怎样才能写出可靠的Node.js代码。这些项目既包含了Docker、Koa等最新技术，也包含了OAuth2、单元测试、消息队列等在实战中经常会遇到的问题和应用场景，每个项目都有详尽的源码，因此在读本书时可边读边实践，通过实际的项目来学习Node.js。如果你是一名Node.js新手，那么通过本书能够快速地掌握如何使用Node.js来搭建应用；如果你具备一定的Node.js开发经验，那么也可以从中学习到许多优秀的开发技巧。

翻开本书，打开编辑器，跟着这7个小项目，一步一步地开始深入了解Node.js的世界吧！

何翊宇（dead_horse），天猫前端技术专家

前言

自本书第1季《Node.js实战（双色）》出版以来，JavaScript界又发生了许多重大事件：React.js和AngularJS 2.0相继出现；ES6于2015年年中正式定稿；io.js从Node.js社区中分裂出来，后又与其合并。截至本书出版时，npm上有接近18万的模块，是去年同期的3倍，周下载量接近5.6亿次，是2014年同期的10倍。越来越多的创业公司和大公司都不同程度地使用了Node.js，Node.js已经成为一门成熟、稳定且具有独特魅力的技术。

延续《Node.js实战（双色）》的写作思路，本书不会从头讲解Node.js是什么，而是面向有一定Node.js基础的读者，建议读者把本书当作入门与进阶之间的过渡书籍来阅读。本书通过7个实例来讲解Node.js在不同场景下的应用，通过阅读本书，读者可以快速熟悉并使用Node.js进行开发。本书由4位作者共同编写，其中吴中骅完成了第1、3章的创作，雷宗民完成了第2、4章的创作，赵坤完成了第5章的创作，刘亚中完成了第6、7章的创作。

第1章主要介绍了如何使用Docker快速发布一个Nginx+Express+Redis项目，然后使用Jenkins进行简单的持续集成发布工作，其中介绍了Docker的基础概念、用法和Jenkins的安装配置方法。

第2章介绍了当前比较流行的OAuth2认证。OAuth2认证是当API服务器对外提供服务时，验证API使用者权限的有效的认证方式。本章主要介绍了使用Node.js搭建一个基本的API服务器所涉及的组件、方法和技术细节。

第3章主要介绍了如何在Node.js中使用消息队列软件RabbitMQ来解决Web服务器或应用服务器间的通信问题。对于服务器间的跨语言通信，以前

一般采用XMLRPC方式，而现在比较流行采用HTTP的RESTful方式，使用RabbitMQ能够很灵活地处理这些事情。

第4章以一个静态博客系统构建工具作为实例，介绍如何使用Node.js的commander模块来编写一个命令行工具。

第5章介绍了ES6中生成器、yield，以及Node.js下一代Web开发框架Koa及其中间件的用法，最后通过搭建一个简单的论坛系统，让读者从实践中学习如何基于Koa快速开发Web应用。

第6章分享了作者在一家时尚杂志相关的互联网创业团队工作的部分经历：为该团队搭建一套较基础的Node.js测试服务，测试范围涵盖服务器、浏览器、Mag+、Adobe InDesign等平台。希望读者通过这个分享能够找到一种更为全面的方式去保证项目代码的质量。

第7章分享了作者与51Degrees团队远程工作的细节，可以让读者大致了解如何将一个已有的C/C++代码库拓展到Node.js平台上进行使用。

感谢电子工业出版社博文视点张国霞编辑的热心帮助和指导；感谢神猪、河蟹、AKI、何翊宇（dead_horse）、Kenneth Kan、Baptiste Gaillard、Jorn Zaefferer和Fish对本书的校验；再次感谢何翊宇为本书慷慨作序。

如果您对本书有任何评论或建议，可加入本书官方QQ群（156627943）进行讨论，或者到读者反馈网站<http://nodejs.ucdok.com>进行反馈，真诚欢迎您的意见与反馈。

目 录

第1章 通过Docker快速发布Node.js应用	1
1.1 什么是Docker	1
1.2 Nginx作为Node.js前端Web Server的作用	3
1.3 安装Docker和下载Images镜像	5
1.4 Docker常用命令	8
1.5 启动Container盒子	10
1.6 文件卷标加载	11
1.7 将多个Container盒子连接起来	13
1.8 不要用SSH连接到你的Container盒子	15
1.9 配置DockerImages镜像和发布应用	19
1.10 什么是Jenkins	26
1.11 通过Docker安装和启动Jenkins	28
1.12 配置Jenkins并自动化部署Node.js项目	29
1.13 小结	36
1.14 参考文献	37
第2章 开发OAuth2认证服务器	38
2.1 本章所用到的第三方模块	38
2.2 REST风格的API	39
2.3 定义返回数据格式	40
2.4 实现简单的API	41
2.4.1 扩展Response对象	41

2.4.2	统一处理出错信息	43
2.4.3	实现简单的API	43
2.4.4	API版本	44
2.5	关于OAuth认证	45
2.5.1	OAuth 2.0授权流程	45
2.5.2	OAuth 2.0授权详解	45
2.5.3	定义授权接口	48
2.6	实现OAuth认证	48
2.6.1	OAuth2/authorize接口	48
2.6.2	OAuth2/access_token接口	52
2.6.3	在处理API请求前验证Access Token	55
2.6.4	Access Token过期的问题	56
2.7	实现API客户端	58
2.8	API传输过程中的安全问题	62
2.9	API请求频率限制	63
2.10	让API返回结果支持不同的格式	65
2.10.1	通过后缀来指定返回的数据格式	65
2.10.2	通过Accept请求头来指定返回的数据格式	67
2.11	生成随机的测试数据	68
2.12	小结	69
2.13	参考文献及开源项目	70
第3章	基于RabbitMQ搭建消息队列	72
3.1	什么是消息队列，消息队列的优势	72
3.2	安装和启动RabbitMQ	75
3.3	RabbitMQ的Hello World	76
3.4	RabbitMQ的工作队列	80
3.5	RabbitMQ的PUB/SUB队列	84
3.6	RabbitMQ的队列路由	89
3.7	RabbitMQ的RPC远程过程调用	94

3.8	基于RabbitMQ的Node.js和Python通信实例	99
3.9	RabbitMQ方案和HTTP方案的对比	103
3.10	小结	117
3.11	参考文献	117
第4章	编写命令行工具——打造一个静态博客系统	118
4.1	本章所使用到的第三方模块	119
4.2	命令格式	120
4.2.1	常见的命令格式	121
4.2.2	定义静态博客命令格式	121
4.3	编写命令行工具	122
4.4	实时预览	126
4.4.1	启动Web服务器	127
4.4.2	渲染文章页面	128
4.4.3	文章元数据	131
4.4.4	增加模板	132
4.4.5	渲染文章列表	136
4.5	生成静态博客	140
4.6	配置文件	146
4.7	创建空白博客模板	150
4.8	一些有用的第三方服务	153
4.8.1	评论组件	153
4.8.2	分享组件	154
4.9	小结	155
4.10	参考文献	156
第5章	基于Koa快速开发Web应用	157
5.1	ES6时代的来临	157
5.1.1	function和function*	158
5.1.2	yield和yield*	160

5.1.3	co和Koa	162
5.2	模板系统	170
5.2.1	ejs和co-ejs	170
5.2.2	过滤器	173
5.3	路由	173
5.4	参数验证与错误处理	175
5.4.1	koa-scheme	175
5.4.2	koa-errorhandler	178
5.5	缓存和配置	182
5.5.1	koa-router-cache和co-cache	182
5.5.2	config-lite	184
5.6	测试	184
5.6.1	单元测试	184
5.6.2	co-mocha和co-supertest	185
5.7	开发一个论坛系统	189
5.7.1	基础项目搭建	189
5.7.2	路由和功能设计	193
5.7.3	自定义模型	194
5.7.4	theme的设计	200
5.7.5	注册	206
5.7.6	登录与登出	213
5.7.7	主页与版块	216
5.7.8	用户页	221
5.7.9	发表页与话题页	222
5.7.10	测试	228
5.7.11	部署	231
5.8	小结	233
5.9	参考文献	233

第6章 Node.js测试服务搭建 235

6.1	概述	235
6.1.1	目的	235

6.1.2 Pixbi	236
6.2 搭建后端测试服务	238
6.2.1 单元测试	239
6.2.2 功能性测试	259
6.2.3 可拓展性测试	260
6.3 搭建前端测试服务	261
6.3.1 PhantomJS	262
6.3.2 BrowserStack	266
6.3.3 Adobe CEP（Common Extensibility Platform）	269
6.4 加入持续集成工作流	271
6.5 小结	274
6.6 参考资料	276
 第7章 使用Node.js绑定C语言库——51Degrees.node	277
7.1 开发背景	277
7.2 预备知识	279
7.2.1 51Degrees-C	279
7.2.2 C/C++中的Node.js API	282
7.2.3 使用nan	284
7.3 编码	285
7.3.1 项目初始化	285
7.3.2 创建v8胶水层接口	286
7.3.3 创建JavaScript代码	293
7.4 构建与发布	294
7.4.1 node-gyp与binding.gyp	294
7.4.2 发布	296
7.5 如何从nan 1.x升级到nan 2.x	296
7.6 后记	298

第1章

通过Docker快速发布Node.js应用

本章将主要介绍如何利用Docker快速发布一个Nginx+Express+Redis项目，然后使用Jenkins进行简单的持续集成发布工作，其中将介绍Docker的基础概念、用法和Jenkins的安装配置方法。Node.js配合Docker和Jenkins可以更加方便地管理我们的应用。

在学习本章之前，读者需要对Linux基本命令行操作、Nginx简单配置、Express框架、Redis有所了解。

1.1 什么是Docker

Docker在2013年首次进入业界的视线，其受到广泛关注则是在2014年的下半年。Docker 1.0自2014年6月公布后的短短几个月内，人气一路飙升。红帽公司在新的RHEL 7版本中增添了支持Docker的功能，IBM公开拥抱Docker和容器，亚马逊推出了EC2容器服务，就连公认的竞争对手VMware也宣布支持Docker。

2014年8月，Linux.com和The New Stack在于美国芝加哥举办的CloudOpen大会上公布了一项由550名从业者参与的调查结果。在最受欢迎的开源云项目

评选中，OpenStack位列第一，Docker位列第二。

业界人士认为，Docker技术在2015年将不会停留于“热度”层面，而会深入地走向部署和应用，因此也将会进一步激发不同开源技术与平台之间的碰撞和整合，最终推动开源及容器技术的向前发展。在中国，将会有更多的云厂商宣布支持Docker。

那么Docker到底是什么？我们该如何定义这个开源界的新宠呢？

Docker官网对Docker的定义如下：

“Docker是一个为开发者和运维管理员搭建的开放平台软件，可以在这个平台上创建、管理和运行生产应用。Docker Hub是一个云端服务，可以用它共享应用或自动化工作流。Docker能够从组件快速开发应用，并且可以轻松创建开发环境、测试环境和生产环境。”

通俗地说，Docker是一个开源的应用容器引擎，可以让开发者打包自己的应用及依赖包到一个可移植的容器中，然后发布到任何流行的Linux机器上，也可以实现虚拟化。Docker容器完全使用沙箱机制，独立于硬件、语言、框架、打包系统，相互之间不会有任何接口，几乎没有任何性能开销，便可以很容易地在机器和数据中心中运行。最重要的是，它不依赖于任何语言、框架或系统。

比如，作为一名开发者，在自己的电脑上开发应用程序时，一切都运行正常，但如果将其部署到其他环境中就可能不能正常工作了。由于开发者使用了自己喜欢的栈、开发语言和版本，所以把它们部署到新的环境如测试环境或生产环境时，就会出现这个问题。这时，运维工程师和开发者需要花费大量的时间、精力和财力，通过进行大量的沟通才能达成一致。但如果使用Docker进行开发，就可以将这一切封装到一个或者几个可相互通信的容器中，而这个容器自身就可以完成所有工作。之后，开发者只需将该容器部署到其他环境中即可。

其次，相对于虚拟机，由于Docker容器不必运行操作系统，所以其体积更小。底层的Linux容器已经被包含在内核当中，这意味着镜像体积非常小、

非常快。虚拟机的体积以GB为单位，需要一到两分钟的启动时间，而容器只以MB为单位，并且可以在几毫秒内启动。这可以加速开发进度，允许开发者轻松地移动容器。

此外，由于容器体积小，可以快速部署，所以有助于开发者进行超大规模的部署。相对于虚拟机，开发者可以使用更少的存储空间、内存和CPU，因为其在性能方面基本上不需要系统开销。

1.2 Nginx作为Node.js前端Web Server的作用

在开始Docker之旅前，我想先说明将Nginx放置在Node.js前端的作用。

对于Nginx想必大家都不会陌生，不过在这里我还是要不厌其烦地再介绍一下。

Nginx（发音同engine x）是一款由俄罗斯程序员Igor Sysoev开发的轻量级网页服务器、反向代理服务器及电子邮件（IMAP/POP3）代理服务器。起初是供俄国大型门户网站及搜索引擎Rambler（俄语为Рамблер）使用的，并因其稳定性、丰富的功能集、示例配置文件和低系统资源的消耗而闻名。此软件在BSD-like协议下发行，可以在UNIX、GNU/Linux、BSD、Mac OS X、Solaris及Windows等操作系统中运行。

这里正是看重了Nginx出色的HTTP反向代理能力，才把它放置在Node.js前端，用来处理我们的各种需求。可能有读者不理解“反向代理”这个名词，笔者在这里稍作解释。

有反向代理就肯定有正向代理，对于正向代理我们接触得比较多，比如我们想访问一些国外的网站，可是由于某些原因无法正常打开该网站或者打开缓慢，这时我们通过香港的HTTP代理就可以正常访问一些国外网站了，在此，香港的这个HTTP代理就是正向代理。反向代理的情况正好相反，比如我们有一个对外的API服务api.nodeInAction.com，初期我们启动一台服务器、

一个Node.js进程就可以完成负载，但是随着后期访问量的加大，一个进程、一台服务器已经不能满足我们的需要了，这时Nginx就可以发挥自己反向代理的能力。我们可以在Nginx后端添加多个服务器或启动多个进程来分担访问压力。在这里，Nginx的作用就是反向代理。

理解了Nginx的反向代理原理后，现在说说为什么要把它放在Node.js的应用之前。其实，这样做大致有如下好处。

1. 静态文件性能

Node.js的静态文件处理性能受制于它的单线程异步I/O模型，注定了静态文件性能不会很好（所以在某些情况下，单线程异步I/O并不是性能的代名词）。在一台普通的4CPU服务器上，使用Nginx处理静态文件的性能基本上是纯Node.js的2倍以上，所以我们应该避免在生产环境下直接使用Node.js来处理静态文件。关于Node.js处理静态文件的更多内容，在《Node.js实战（双色）》中有详细的对比和介绍，欢迎读者阅读。

2. 反向代理规则

有时会存在反向代理服务器配置多样化的情况，有时我们希望配置较好的机器能够分担更多的压力，有时又因为session的关系，我们需要将同一来源IP的客户端全转发到同一个进程上，对于诸如此类的规则，使用Nginx的配置文件就可以简单实现。

3. 扩展性

Nginx可以加入许多扩展来帮助我们处理业务，最典型的就是加入Lua语言的扩展。胶水语言Lua赋予了Nginx复杂逻辑判断的能力，并且保持了一贯的高效性。例如我们有一个API服务，对访问会进行MD5签名或对同一客户端来源有访问频率限制，这部分代码是后端业务处理前必须通过的验证，具有卡口的作用。利用Lua扩展，我们就可以高效、简单地完成这个卡口。

4. 稳定性和转发性能

对Nginx的稳定性大家有目共睹，Nginx在同样的负载下，相比Node.js占

用的CPU和内存资源更少。同时，高效地转发性能、便捷地转发配置也是我们选择它作为反向代理的原因之一，比如我们可以根据不同的URL请求路径转发到不同的后端机器上，也可以设定超时时间等，方便管理。

5. 安全性

Nginx已经被各大互联网公司广泛应用，经过一些配置可以有效抵挡类似slowloris等的DoS攻击，而Node.js在这方面做得还不够，关于Node.js开发安全方面的更多内容，可以参考《Node.js实战（双色）》一书，其中的第8章专门讨论了如何更安全地开发Node.js的Web应用。

6. 运维管理

如果我们目前只有一台Web服务器，同时有多个站点需要占用80端口，这时我们只需让Node.js服务监听本地的特殊端口如3000，通过Nginx的反向代理配置，就可以将多个站点域名指向一台机器了。当然，如果公司配置了专门的运维部门来管理服务器，相信他们对Nginx的熟悉程度一定远远大于Node.js，他们自己就可以轻松地修改一些配置，而不用来麻烦我们了。

所以，一个好习惯就是，在生产环境中，永远把Nginx放置在Node.js的前端，对性能、安全性和将来的扩展都有益处。

1.3 安装Docker和下载Images镜像

在介绍完Nginx之后，我们继续Docker之旅，在Docker官网有详细的各个系统安装流程，这里介绍CentOS下的安装流程，其他系统的安装地址为<https://docs.docker.com/>。

对于使用CentOS 7的用户，直接运行如下命令，就可以安装最新版本的Docker。

```
$ sudo yum install docker
```


使用CentOS 6.5的用户需要先获取epel源并导入。

```
$ wget -c http://dl.fedoraproject.org/pub/epel/6/x86_64/epel-release-6-8.noarch.rpm
$ rpm -ivh epel-release-6-8.noarch.rpm
$ rpm -import /etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL-6
```

接着通过yum安装Docker。

```
$ yum install docker-io --enablerepo=epel
```

启动Docker服务，并且把Docker服务注册为开机启动。

```
$ sudo service docker start
$ sudo chkconfig docker on
```

我们可以输入如下命令，检查Docker进程是否已经启动。

```
$ ps -ef|grep docker
```

如果发现Docker进程未成功启动，就需要进入/var/log/目录下查看Docker日志文件的信息了，应用CentOS 6.5的用户可能会发现系统报出如下错误。

```
/usr/bin/docker: relocation error: /usr/bin/docker: symbol
dm_task_get_info_with_deferred_remove, version Base not defined
in file libdevmapper.so.1.02 with link time reference
```

执行如下命令可以修复，然后重新启动Docker服务。

```
$ yum-config-manager --enable public_ol6_latest
$ yum install -y device-mapper-event-libs
$ yum update -y device-mapper-libs
```

现在，Docker服务已经安装并启动了，我们需要下载Image镜像，镜像就是我们应用运行的环境，比如我们可以自己安装好Node.js和npm，然后发布到Docker Hub上供自己或者别人下载，我们也可以下载安装一些官方镜像，把它作为自己镜像的基础。下面我们先下载CentOS镜像。

```
$ sudo docker pull centos:7
```

等待片刻，等下载完毕后执行命令查看镜像是否安装成功。

```
$ sudo docker images centos
```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
centos	centos7	8efe422e6104	2 weeks ago	224 MB
centos	latest	8efe422e6104	2 weeks ago	224 MB
centos	7	8efe422e6104	2 weeks ago	224 MB

我们可以去Docker官网搜索一些我们需要的镜像，根据下载次数、推荐次数来排序搜索结果，访问地址为<https://registry.hub.docker.com/>。例如，在Docker官网搜索Node.js关键字，在结果列表中我们可以看到被人赞星星的次数和下载次数，如图1-1所示，二者的分数越高，就越说明这个镜像值得信赖。

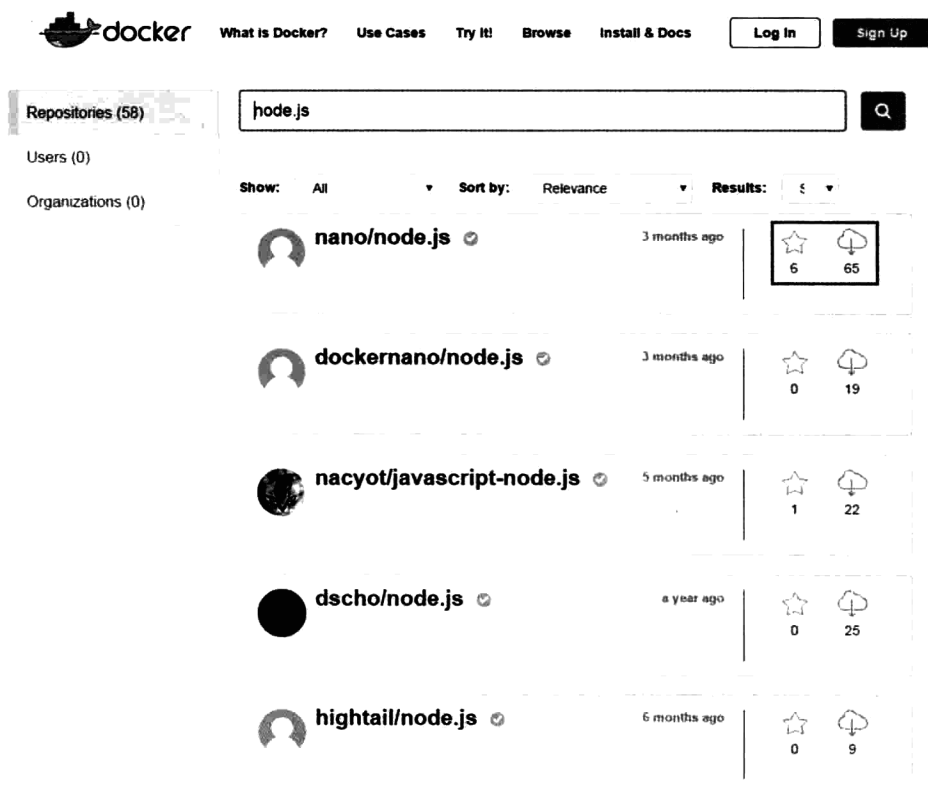


图1-1 Docker hub下载页面

1.4 Docker常用命令

在深入学习Docker之前，先罗列一些Docker常用命令，一时半会看不懂也没关系，在深入学习的过程中很容易掌握这些常用命令。关于详细命令的相关说明，可通过输入如下命令获取帮助。

```
$ docker -h
```

1. 获取镜像

```
$ sudo docker pull NAME[:TAG]
```

命令示例：

```
$ sudo docker pull centos:latest
```

2. 启动Container盒子

```
$ sudo docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

一个启动Container盒子的简单示例：

```
$ sudo docker run -t -i centos /bin/bash
```

启动Container盒子是非常重要的命令，我们会在下一节详细介绍Container盒子及启动命令。

3. 查看镜像列表，列出本地的所有images

```
$ sudo docker images [OPTIONS] [NAME]
```

命令示例：

```
$ sudo docker images centos
```

4. 查看容器列表，可看到我们创建过的所有Container

```
$ sudo docker ps [OPTIONS]
```

查看所有运行中或者停止运行的盒子的命令示例：

```
$ sudo docker ps -a
```

5. 删除镜像，从本地删除一个已经下载的镜像

```
$ sudo docker rmi IMAGE [IMAGE...]
```

命令示例：

```
$ sudo docker rmi centos:latest
```

6. 移除一个或多个容器实例

```
$ sudo docker rm [OPTIONS] CONTAINER [CONTAINER...]
```

移除所有未运行的容器的命令示例：

```
$ sudo docker rm $(sudo docker ps -aq)
```

7. 停止一个正在运行的容器

```
$ sudo docker kill [OPTIONS] CONTAINER [CONTAINER...]
```

命令示例：

```
$ sudo docker kill 026e
```

8. 重启一个正在运行的容器

```
$ sudo docker restart [OPTIONS] CONTAINER [CONTAINER...]
```

命令示例：

```
$ sudo docker restart 026e
```

9. 启动一个已经停止的容器

```
$ sudo docker start [OPTIONS] CONTAINER [CONTAINER...]
```

命令示例：

```
$ sudo docker start 026e
```

基本上常用的命令就是这些，读者第一次阅读只需对它们有大体了解，

随着对Docker的不断使用，我们很容易熟练运用上面的这些命令。在1.5节我们将详细介绍Docker的精髓，即Container。

1.5 启动Container盒子

1.4节所讲述的命令，再加上Image、Container，让我们觉得眼花缭乱。既然下载了Image，为什么又会多一个Container呢？在Docker的世界里，它们之间的关系又是怎样的呢？

接下来简单说明Image和Container的关系。Image顾名思义就是镜像，我们可以把它理解为一个执行环境（env），在我们执行了docker run命令之后，Docker就会根据当前的Image创建一个新的Container，Container是一个程序运行的沙箱，它们互相独立，但都运行在由Image创建的执行环境之上。

然后启动一段小程序，基于我们刚才下载的CentOS镜像，启动一个Container，让控制台输出一行Hello world。

```
$ sudo docker run b15 /bin/echo 'Hello world'
Hello world
```

其中，b15是我们之前下载的镜像的ID，在Docker中不必输入完整的ID，只要保证输入的ID前几位能让Docker找到唯一的Image或Container即可，这同样适用于删除操作。

现在我们尝试启动一个稍复杂的Container，每秒钟打印一个Hello world。

```
$ sudo docker run -i -t b15 /bin/sh -c "while true; do echo
hello world; sleep 1; done "
Hello world
Hello world
Hello world
Hello world
Hello world
...
```


命令中的参数-i表示同步Container的stdin，-t表示分配一个伪终端。

我们在上面执行了两个dockerrun的任务，其实就是创建了两个独立的Container，我们通过命令dockerps-a，就可以列出我们创建过的所有Container了，出于版面显示的原因，这里做了部分修改，把COMMAND改短了。

```
CONTAINER ID    IMAGE    COMMAND    CREATED    STATUS    PORTS    NAMES
026ec6c8802c   centos:7 ...    4 minutes Up 4 minutes
focused_bartik
cc5105d4e6f5   centos:7 ...    4 minutes Exited (0) 10 minutes
ago    determined_pare
```

我们看到026ec6c8802c这个每隔一秒打印Hello world的Container在4分钟前创建，并且一直在运行，已经运行了4分钟。另一个cc5105d4e6f5即一次性打印Hello world的Container已经退出了。

由于每隔一秒进行执行的Container永远不会停止，所以我们现在需要手动把它删除，删除运行中的Container时需要加上参数-f。

```
$ sudo docker rm -f 026
026
```

对于Container的停止、重启和启动的操作命令，请参考1.4节的内容。

1.6 文件卷标加载

我们在1.5节学习了Container的基本概念，并启动了几个输出Hello world的例子，初步理解Container的读者可能会把Docker的Container理解为一个虚拟机，虽然这并不完全正确，但是在本节我不会去纠正他们，这样的理解对我们深入学习Docker会有所帮助。

我们可能有这样的需求，应用程序跑在Container里，但我们不想在里面记录日志，因为万一Image升级，我们就必须重新执行docker run命令，这样

日志文件处理就比较麻烦，而且记录在Container文件系统里的日志也不方便我们查看。这时就需要将主机的文件卷标挂载到Container中去，在Container中写入和读取的某个文件目录，其实就是挂载进去的主机目录，我们通过参数-v把主机文件映射到Container中。

下面的命令就是把本机的/etc目录挂载到Container里的/opt/etc下面，并且打印Container的/opt/etc目录。

```
$ docker run --rm=true -i -t --name=ls-volume -v /etc:/opt/etc/
etc/ centos ls /opt/etc

boot2docker  hostname      ld.so.conf passwd-  securetty  sysconfig
default      hosts          mke2fs.conf pcrmcia  services  sysctl.conf
fstab        hosts.allow   modprobe.conf  profile   shadow
udev
group        hosts.deny    motd          profile.d shadow-    version
group-       init.d        mtab          protocols shells
gshadow      inittab       netconfig     rc.d      skel
gshadow-     issue        nsswitch.conf resolv.conf ssl
host.conf    ld.so.cache  passwd       rpc       sudoers
```

参数-v后面的冒号的左侧部分是本地主机路径，冒号的右侧部分是对应Container的路径，--rm=true表示这个Container运行结束后自动删除。

如果想要挂载后的文件是只读的，那么需要这样挂载。

```
-v /etc:/opt/etc/:ro #read only
```

我们也可以挂载一个已经存在的Container中的文件系统，需要用到--volumes-from参数，先创建一个Container，它共享/etc/目录给其他Container。

```
$ sudo docker run -i -t -p 1337:1337 --name=etc_share -v /
etc/ centos mkdir /etc/my_share && /bin/sh -c "while true; do
echo hello world; sleep 1; done "
```

参数-p表示端口映射，上面的命令将Container的1337端口映射到主机的1337端口，对外共享/etc/目录，给这个Container取名为etc_share。

然后我们启动一个ls_etc的Container来挂载并打印etc_share共享的目录。

```
$ sudo docker run --rm=true -i -t --volumes-from etc_share
--name=ls_etc centos ls /etc

...
my_share      pki           rc0.d         rpc           shells
tmpfiles.d
...
```

我们看到，ls_etc这个Container打印出来的/etc目录包含了我们之前在etc_share这个Container中创建的my_share目录。

1.7 将多个Container盒子连接起来

我们在1.6节学习了如何将主机或者Container的文件系统挂载起来，本节我们将学习如何把各个Container连接在一起。

先下载一个Redis数据库的镜像，这是使用Docker的常规做法，数据库和程序各自单独用一个Image，利用Docker的link属性将它们连接起来配合使用。

```
$ sudo docker pull redis:latest #下载官方的Redis最新镜像，耐心等待一段时间
```

一般我们使用docker pull命令时后面都会跟着版本号（例如，截稿时Redis的最新版本是2.8.19）或者latest，这样就不用重复地下载这个镜像的老版本了，可以加快速度。接着我们执行命令，启动Redis镜像的Container，开启redis-server持久化服务。

```
$ sudo docker run --name redis-server -d redis redis-server
--appendonly yes
```

然后启动一个Redis镜像的Container作为客户端，连接刚才启动的redis-server。

Node.js实战（第2季）

```
$ sudo docker run --rm=true -it --link redis-server:redis redis  
/bin/bash
```

执行上面的命令后，就进入了Container内部的bash，可以直接在里面执行一些linux命令。

```
redis@7441b8880e4e:/data$ env
```

Shell在主机中还是在Container中，主要根据\$符号左侧的用户名来区分，上面的命令将打印Container里系统的环境变量，输出如下。

```
REDIS_PORT_6379_TCP_PROTO=tcp  
REDIS_ENV_REDIS_DOWNLOAD_URL=http://download.redis.io/releases/  
redis-2.8.19.tar.gz  
HOSTNAME=7ff5092b69fa  
REDIS_ENV_REDIS_DOWNLOAD_SHA1=3e362f4770ac2fdbdce58a5aa951c19  
67e0facc8  
TERM=xterm  
REDIS_DOWNLOAD_URL=http://download.redis.io/releases/redis-2.  
8.19.tar.gz  
REDIS_NAME=/trusting_mayer/redis  
REDIS_PORT_6379_TCP_ADDR=172.17.0.16  
REDIS_PORT_6379_TCP_PORT=6379  
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:  
/bin  
PWD=/data  
REDIS_PORT_6379_TCP=tcp://172.17.0.16:6379  
REDIS_PORT=tcp://172.17.0.16:6379  
HOME=/root  
SHLVL=1  
REDIS_VERSION=2.8.19  
REDIS_DOWNLOAD_SHA1=3e362f4770ac2fdbdce58a5aa951c1967e0facc8  
REDIS_ENV_REDIS_VERSION=2.8.19  
_=/usr/bin/env  
  
$ redis-cli -h "$REDIS_PORT_6379_TCP_ADDR" -p "$REDIS_PORT_  
6379_TCP_PORT"  
$ 172.17.0.34:6379> set a 1 #成功连入Redis数据库服务器  
OK
```

```
$ 172.17.0.34:6379> get a  
" 1 "
```

我们成功地利用环境变量连接上了一台为我们提供数据库服务的Redis的Container，用同样的方法，我们可以在程序里连接其他数据库，比如MySQL或者MongoDB等。

1.8 不要用SSH连接到你的Container盒子

学习了前面的几个章节，相信大家已经不算是一个Docker新手了，越来越多的情况会让你想到：怎么进入自己的Container中呢？其他人会告诉你：在你的Container里面装一个SSH Server，这样你就可以连入你的Container了。但这是一种糟糕的做法，下面我将告诉大家为什么这么做是错误的，如果实在万不得已，那么我们又能用什么方式来替代它。

在Container里安装一个SSH Server是非常诱人的，因为这样我们就可以直接连接到Container，并且进入它的内部，我们可以使用在前面几节学到的端口映射方式，让本地的SSH客户端连入Container。

所以也不奇怪，人们会建议在Container中创建一个SSH Server，但是，我们在这么做之前需要考虑以下几个问题。

（1）你需要SSH来干什么？

大部分需求是，你要检查日志、做备份、重启进程、调整配置或者查看服务器情况，下面将介绍如何不使用SSH来做到以上事情。

（2）如何管理密钥和密码？

大部分的可能是，你将你的密钥和密码一起装进你的Image中，或者将它们放在文件卷中。想一想如果你要更新你的密钥或者密码，那么你应该怎么做呢？

如果你把它们装载进Image中，那么你每次更新都需要重新创建Image，重新发布Image，然后重启Container。这样做不是很优雅。

一个更好的办法是将这些东西放在一个文件卷标中，它能工作，但是也有显著的缺点，你必须保证你的Container对这个文件卷标没有写的权限，否则可能会污染你的密钥和密码，从而造成你无法登录这个Container。而且可能因为多个Container共享这些东西而变得更加难以管理。

（3）如何管理你的密码升级？

SSH Server是非常安全的，但是一旦你的密钥或密码泄漏，你将不得不升级所有使用SSH的Container，并且重启它们。这也可能让Memcache这样的内存缓存服务器的缓存全部丢失，你不得不重建缓存。

（4）是否加入SSH Server就能工作？

不是的，你还需要加入进程管理软件、Monit或Supervisor等监控软件，让应用开启多个进程运行。换言之，你把一个简单的Container转变为一个复杂的东西了。如果你的应用停止了，那么你不得不从你的进程管理软件那里获得信息，因为Docker只能管理单进程。

但是若不使用SSH，我们该如何做以下事情呢？

（1）备份数据。

我们的数据必须是一个volumn，这样我们可以启动另外一个Container，并且通过--volumes-from来共享应用的Container数据，这个新的Container会来处理数据备份的事情。额外的好处是，如果只对我们的数据文件（比如日志）进行压缩并长久保存，那么完全可以在一个新的Container中处理，这样，我们的应用Container就是干净的。

（2）检查日志。

使用文件volumn，用和之前一样的方法重新启动一个日志分析的Container，让它来处理日志和检查日志。

（3）重启应用服务。

这个问题的解决办法更简单，我们只需要重启Container即可。

（4）修改配置文件。

如果我们正在执行一个持久的配置变更，那么最好把这个变更放在Image中，如果又启动了一个Container，服务还是使用老的配置，那么我们的配置变更将丢失。如果需要在应用存活期间改变自己的配置，例如增加一个新的虚拟站点，那么还是需要使用volumn来处理，这样，所有的应用Container都可以快速地临时变更配置。

（5）调试应用。

这可能是唯一需要进入Container的场景了，这样你就需要用到nsenter软件。

下面笔者利用类似机器猫的任意软件nsenter，带你进入Container中。nsenter是一个小的工具，用来进入现有的命名空间。命名空间是什么？它们是Container的重要组成部分。简单点说，通过使用nsenter可以进入一个已经存在的Container中，尽管这个Container没有安装SSH Server或者其他类似软件。nsenter项目的地址为<https://github.com/jpetazzo/nsenter>。

我们可以通过命令来安装nsenter，这个命令会自己下载nsenter镜像，并且把nsenter命令安装到主机的/usr/bin中，这样，我们就可以很方便地使用它了。

```
$ sudo docker run -v /usr/local/bin:/target jpetazzo/nsenter
```

我们先要找出需要进入的Container的pid。

```
PID=$(docker inspect --format '{{.State.Pid}}' <container_name_or_ID>)
```

命令实例：

```
$ sudo docker inspect --format {{.State.Pid}} 9479
7026
```

这里我们得到了id为9479的Container的pid号为7026，这句话有点拗口，其实我们只需关心7026这个pid号就可以了。我们根据刚才获得的pid就能顺利进入Container的内部了。

```
$ sudo nsenter --target $PID --mount --uts --ipc --net --pid
```

这里我们把\$PID替换为7026即可，命令如下。

```
$ sudo nsenter --target 7026 --mount --uts --ipc --net --pid
```

如果你想要远程访问这个Container，那么可以通过SSH链接到你的主机，并且使用nsenter连接进入Container，所以大家是不是觉得完全没有必要在Container里安装一个SSH Server了？

如果在pull镜像nsenter时出现错误，那么估计是CentOS内核的版本问题，所以尽量使用较新的内核版本来启动Docker，出现下面的错误可能就是内核版本过低。

```
Error pulling image (latest) from jpetazzo/nsenter, Unknown
filesystem type on /dev/mapper/...
```

无法进入Container的错误，也是因为内核过低，没有正确安装镜像。

```
nsenter: cannot open /proc/27797/ns/ipc: No such file or
directory
```

如果在运行安装nsenter时出现如下错误：

```
$ sudo docker run -v /usr/local/bin:/target jpetazzo/nsenter
Installing nsenter to /target
cp: cannot create regular file '/target/nsenter': Permission
denied
Installing docker-enter to /target
cp: cannot create regular file '/target/docker-enter': Permission
denied
```


那么就需要手动将Container里的nsenter命令复制到/usr/local/bin目录下了，先把jpetazzo/nsenter运行起来，然后手动进入文件系统将命令复制出来。下面的<containid>就是我们使用docker ps查到的ID，目录devicemapper是centos下的路径名，在Windows下是aufs。

```
$ cp /var/lib/docker/devicemapper/mnt/<containid>/rootfs/
nsenter /usr/local/bin/
$ cp /var/lib/docker/devicemapper/mnt/<containid>/rootfs/
docker-enter /usr/local/bin/
```

1.9 配置DockerImages镜像和发布应用

我们已经学习了很多关于Docker的知识，Docker之旅也渐渐接近尾声，本节我们就要简单制作一个Node.js包含Express.js环境的镜像，通过pm2来启动Web应用，然后发布到Docker云上；我们还会使用Redis数据库来暂存用户的访问次数；在Node.js应用前端，我们需要放置一个Nginx作为反向代理。现在让我们开始吧。

首先，我们把需要用到的Image镜像统统下载到本地，执行如下命令，等待片刻就能下载成功。

```
$ sudo docker pull redis:2.8.19
$ sudo docker pull node:0.10.36
```

为了加快下载速度和本书代码的兼容性，我们指定了下载各个镜像的版本，读者可以根据当时的最新版本进行下载。执行docker images检查一下这些镜像是否都安装完毕，正常情况下会打印出各个镜像列表。

```
node      0.10.36      20fbb0b572a2    5 hours ago    705.4 MB
node      latest       20fbb0b572a2    5 hours ago    705.4 MB
redis     latest       5e0586116d76    5 days ago     110.8 MB
redis     2.8.19      5e0586116d76    5 days ago     110.8 MB
jpetazzo/nsenter latest       6ed3dald7fa6    9 weeks ago    367.7 MB
```

我们先在本地创建一个部署Node.js应用的目录，然后写上package.json。

```
$ mkdir /var/node/  
$ mkdir /var/node/docker_node
```

在创建我们的应用之前，我们从node 0.10.36这个镜像基础上开始制作自己的镜像，这个镜像只是比node 0.10.36镜像多了一个pm2命令。运行如下命令，进入Container的命令行，然后安装pm2软件。如果读者对Node.js比较熟悉，那么相信对pm2不会陌生，它是Node.js进程管理软件，可以方便地重启进程和查看Node.js日志。

```
$ sudo docker run -i -t node /bin/bash  
#进入Container的bash  
$ npm install pm2 -g  
$ pm2 -v  
0.12.3  
#考虑国内的网络，再装下cnpm更靠谱些  
$ npm install cnpm -g --registry=https://registry.npm.taobao.org  
#从Container的bash退出  
$ exit
```

这样我们就成功地在node 0.10.36这个镜像的基础上安装了pm2，然后我们要把这个新的Container保存为镜像，这样以后我们要用到带pm2的Node.js镜像时，只需下载它即可。执行命令进行登录，然后把镜像push到云上，非官方不允许直接提交根目录镜像，所以必须以<用户名>/<镜像名>这样的方式提交，比如doublespout/node_pm2。

```
#查看所有Container，找到刚才的id  
$ sudo docker ps -a  
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS        PORTS   NAMES  
...  
7a3e85bfaddf   node:0    "/bin/bash"             5 minutes ago   Exited  
(130)...      goofy_fermi  
...  
  
#使用docker官网注册的用户名和密码进行登录  
$ sudo docker login
```

```
Username: <Your Docker Account>
```

```
Password:
```

```
Email: <Your Email>
```

```
Login Succeeded
```

```
#登录成功之后，把Container提交为Images
```

```
$ sudo docker commit 7a3e doublespout/node_pm2
```

```
#然后查看Images列表
```

```
$ sudo docker images node_pm2
```

```
REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE
node_pm2 latest 9a418757ae2b About a minute ago 714.8 MB
```

```
#把镜像提交到云上
```

```
$ sudo docker push doublespout/node_pm2
```

等待片刻后，我们新的镜像就保存到了Docker云上，然后我们把本地的doublespout/node_pm2删除，试着从云上下载这个镜像。

```
$ sudo docker rmi doublespout/node_pm2
```

```
$ sudo docker images doublespout/node_pm2
```

```
REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE
```

```
#发现是空的，然后我们从云上pull
```

```
$ sudo docker pull doublespout/node_pm2
```

```
#稍等片刻即可下载完毕
```

接下来我们将通过Redis镜像启动一个Redis的Container，命令如下：

```
docker run --name redis-server -d redis redis-server
--appendonly yes
```

然后我们要准备编写Node.js代码来实现这个计数访问应用的功能。在/var/node/docker_node目录下创建如下package.json文件，这里对依赖包写上版本号是比较稳妥的方式，可以免去因为依赖包升级而造成的应用不稳定的情况，实在有必要升级时，可以单独升级某几个依赖测试。

```
{
  "name": "docker_node",
```

```
"version": "0.0.1",
"main": "app.js",
"dependencies": {
  "express": "4.10.2",
  "redis": "0.12.1",
},
"engines": {
  "node": ">=0.10.0"
}
}
```

然后我们创建app.js，启动并监听8000端口，同时通过Redis记录访问次数。

```
var express = require('express');
var redis = require("redis");
var app = express();
//从环境变量里读取Redis服务器的ip地址
var redisHost = process.env['REDIS_PORT_6379_TCP_ADDR'];
var redisPort = process.env['REDIS_PORT_6379_TCP_PORT'];

var reidsClient = redis.createClient(redisPort, redisHost);

app.get('/', function(req, res){
  console.log('get request')
  reidsClient.get('access_count', function(err, countNum){
    if(err){
      return res.send('get access count error')
    }
    if(!countNum){
      countNum = 1
    }
    else{
      countNum = parseInt(countNum) + 1
    }
    reidsClient.set('access_count', countNum, function
    (err){
      if(err){
```

```

        return res.send('set access count error')
    }
    res.send(countNum.toString())
  })
})

app.listen(8000);

```

我们先启动一个Container把依赖包装一下，命令如下：

```
$ sudo docker run --rm -i -t -v /var/node/docker_node:/var/
node/docker_node -w /var/node/docker_node/ doublespout/node_pm2
cnpm install
```

-w参数表示命令执行的当前工作目录，屏幕会打印依赖包的安装过程，等所有Node.js的包安装完成后，这个Container会自动退出，然后我们进入/var/node/docker_node/目录，就可以看到node_modules文件夹，说明我们的依赖包安装完毕了。

如果出现EACCESS的权限错误，那么可以执行如下命令，许可SELinux的工作状态，不过这只是临时修改，重启系统后会恢复。

```
su -c "setenforce 0"
```

代码开发完毕，基于刚才我们提交的doublespout/node_pm2镜像，我们要启动一个运行这个程序的Container，要求这个Container有端口映射、文件挂载，并同时加载Redis的那个Container，命令如下：

```

#挂载pm2的日志输出
$ mkdir /var/log/pm2
#使用pm2启动app应用，但是会有问题
$ sudo docker run -d --name "nodeCountAccess" -p 8000:8000
-v /var/node/docker_node:/var/node/docker_node -v /var/log/pm2:/
root/.pm2/logs/ --link redis-server:redis -w /var/node/docker_
node/ doublespout/node_pm2 pm2 start app.js

```

但是我们在执行docker ps后会发现这个Container并没有启动，这是什么

原因呢？因为我们利用pm2的守护进程方式启动了应用，所以Container会认为进程已经运行结束，所以自己退出了，这时我们让pm2以非守护进程的方式运行在Container里即可，我们的命令要做一些更改。

```
$ sudo docker run -d --name "nodeCountAccess" -p 8000:8000
-v /var/node/docker_node:/var/node/docker_node -v /var/log/pm2:/
root/.pm2/logs/ --link redis-server:redis -w /var/node/docker_
node/ doublespout/node_pm2 pm2 start --no-daemon app.js
```

这时我们再执行docker ps，就可以看到nodeCountAccess这个名字的Container在运行了，使用浏览器打开主机的8000端口，也能看到访问的计数次数。

接下来就轮到作为反向代理的Nginx出场了。

由于使用Docker的Container，所以它的IP地址是动态变化的，若我们想要使用Nginx容器来做反向代理，那么配置写起来会比较困难，这里我们就暂不使用Docker容器来管理Nginx了，而是直接编译安装Nginx。

我们使用Nginx的分支版本openresty来做反向代理，openresty比Nginx内置了ngx-lua模块，让Nginx具有逻辑处理能力，我们用yum安装依赖包，然后编译安装openresty。

```
yum install -y gcc gcc-c++ kernel-devel
yum install -y readline-devel pcre-devel openssl-devel openssl
zlib zlib-devel pcre-devel
wget http://openresty.org/download/nginx_openresty-1.7.2.1.tar.
gz
tar -zxvf ngx_openresty-1.7.2.1.tar.gz
cd ngx_openresty-1.7.2.1
./configure --prefix=/opt/openresty \
            --with-pcre-jit \
            --with-ipv6 \
            --without-http_redis2_module \
            --with-http_iconv_module \
            -j2
make && make install
```

```
ln -s /opt/openresty/nginx/sbin/nginx /usr/sbin/
```

修改openresty的默认配置文件，配置文件在/opt/openresty/nginx/conf/nginx.conf中，我们将其修改为如下内容，出于篇幅的考虑，此配置文件是精简版的配置，不要用于生产环境，大家主要看Server那段配置的内容。

```
worker_processes 1;
events {
    worker_connections 1024;
}
http {
    include mime.types;
    default_type application/octet-stream;
    server_names_hash_bucket_size 64;
    access_log off;

    sendfile on;
    keepalive_timeout 65;

    server {
        listen 3001;
        location / {
            proxy_pass http://127.0.0.1:8000;
            proxy_redirect default;
            proxy_http_version 1.1;
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Connection $http_connection;
            proxy_set_header X-Forwarded-For $proxy_add_x_
forwarded_for;
            proxy_set_header Host $http_host;
        }
    }
}
```

执行命令nginx就可以将openresty运行起来，然后打开浏览器，输入主机IP:3001就可以正常访问我们之前启动的Node.js访问计数应用。

另外，如果遇到在Container里无法解析域名的情况，就需要手动增加DNS服务器，方法如下：

```
DOCKER_OPTS=" --dns 8.8.8.8 "  
service docker restart
```

1.10 什么是Jenkins

Jeknins是一款由Java开发的开源软件项目，旨在提供一个开放易用的软件平台，使持续集成变成可能，它的前身就是大名鼎鼎的Hundson。Hundson是收费的商用版本，Jenkins是它的一个免费开源的分支，所以我们选择使用Jenkins，毕竟能省则省。

那什么是持续集成呢？以下概念摘自IBM团队的定义：

“随着软件开发复杂度的不断提高，团队开发成员间如何更好地协同工作以确保软件开发的质量已经慢慢成为开发过程中不可回避的问题，持续集成正是针对这类问题的一种软件开发实践。它倡导团队开发成员必须经常集成他们的工作，甚至每天都可能发生多次集成。而每次的集成都是通过自动化的构建来验证的，包括自动编译、发布和测试，从而尽快发现集成错误，让团队能够更快地开发内聚的软件。”

持续集成的核心价值在于如下几点。

（1）持续集成中的任何一个环节都是自动完成的，无须太多的人工干预，有利于减少重复过程，以节省时间、费用和工作量。

（2）持续集成保障了每个时间点上团队成员提交的代码是能成功集成的。也就是说，在任何时间点都能第一时间发现软件的集成问题，使任意时间发布可部署的软件成为可能。

（3）持续集成还利于软件本身的发展趋势，这点在需求不明确或者频繁性变更的情景中尤其重要，持续集成的质量能帮助团队进行有效决策，同时

建立团队开发产品的信心。

估计大家看完这些定义，对Jenkins能做什么依然没什么概念，简而言之，我们利用Jenkins持续集成Node.js项目之后，就不用每次都登录到服务器，执行`pm2 restart xxx`或者更原始一点的`kill xx`，然后`node xxx`。如图1-2所示是某个已经配置好的项目在Jenkins中的截图，我们只需单击“立即构建”按钮，就可以自动从Git仓库获取代码，然后远程部署到目标服务器，执行一些安装依赖包和测试的命令，最后启动应用。



图1-2 Jenkins项目主页

可能有朋友会说，每次我通过SSH登录服务器执行上面的命令也挺方便的，但是我们的Node.js程序不止在一台服务器上，每次部署的重复劳动会让人觉得这就是一场灾难；当有些采用其他语言的项目如Python或C++的程序需要管理时，对于重启的方式和编译的选项我们都要烂熟于心，否则就会出错，为什么不把这一切都自动化呢？

接下来介绍如何从零开始，搭建一个Jenkins的持续集成软件，自动化部署我们之前开发的那个记录访问次数的Node.js应用。

1.11 通过Docker安装和启动Jenkins

有了Docker这个利器，我们省去了安装Java环境的麻烦，所以安装Jenkins异常简单，只需执行如下一行命令即可。

```
#截稿时，Docker中最新版本的Jenkins是1.554.1
docker pull jenkins:1.554.1
```

拉取镜像之后，我们先创建目录，然后就可以启动Jenkins的Container了，我们要把Jenkins的文件存储地址挂载到主机上，万一Jenkins的服务器重装或者迁移，我们都可以很方便地把之前的项目配置保留，否则就只能进入Container的文件系统里去复制了。另外，Jenkins会搭建在内网的服务器上，而非生产服务器，如果外网能直接访问，那么可能会造成一定的风险。

```
#创建本地的Jenkins配置文件目录
$ mkdir /var/jenkins_home
$ sudo docker run -d --name myjenkins -p 49001:8080 -v /var/
jenkins_home:/var/jenkins_home jenkins
```

这样我们就顺利启动了Jenkins的服务，8080端口是Jenkins的默认监听端口，我们把它映射到了本地主机的49001端口，要注意把搭建Jenkins服务器的iptables关闭，一切顺利的话，我们就可以看到Jenkins的欢迎页面了，如图1-3所示。建议去系统管理→管理用户栏目中创建几个用户和权限，方便多人协同操作。

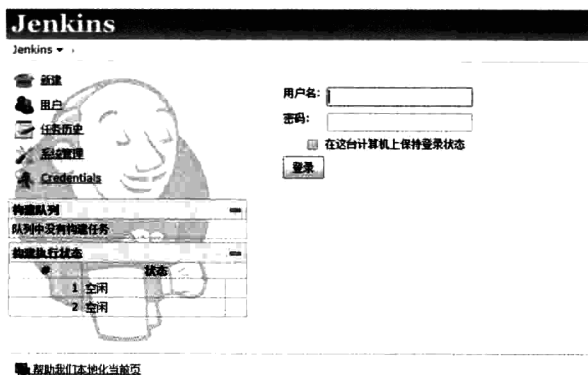


图1-3 登录Jenkins

1.12 配置Jenkins并自动化部署Node.js项目

我们需要对Jenkins进行一些简单的配置，才能让它自动化地部署应用，由于我们的代码是部署在GitHub仓库中的，所以我们先要对Jenkins安装几个插件，让它可以从GitHub中获取代码，并远程部署到我们生产的服务器上。

进入系统管理→管理插件→可选插件，在右上侧的筛选框中输入Git，并安装Git Plugin（This plugin integrates GIT with Jenkins）插件；然后安装插件Publish Over SSH（Send build artifacts over SSH）插件，检查网络这一步骤可能时间较长，请耐心等待。

插件安装完成后，我们需要重新启动Jenkins，一般安装完毕后会自动重启，如果自动重启失败，那么可以进入Jenkins的目录/restart下手动重启。

#进入目录手动重启

```
http://192.168.1.116:49001/restart
```

如果可选插件列表为空，那么进入“高级”选项卡，单击“立即获取”按钮，就可以获取可选插件列表了，如图1-4所示。

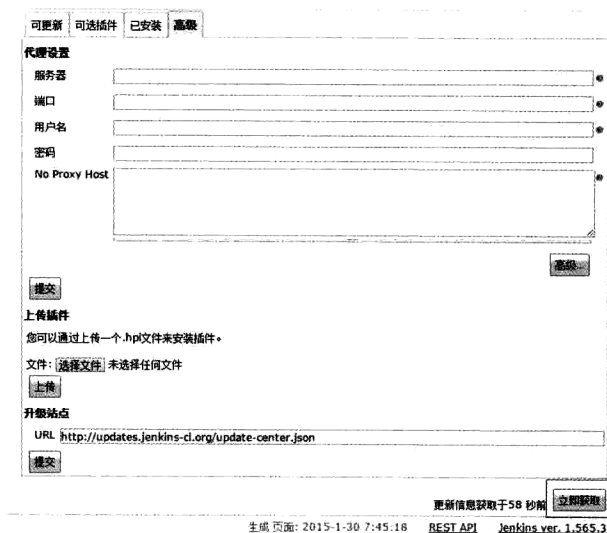


图1-4 Jenkins安装插件界面

重启完成之后，进入系统管理→系统设置来对插件进行简单的设置，增加远程的服务器配置。在如图1-5所示的界面中，填入我们待发布的生产服务器的IP地址、SSH端口及用户名、密码等信息。如果远程服务器是通过key来登录的，那么还需要把key的存放路径写上。

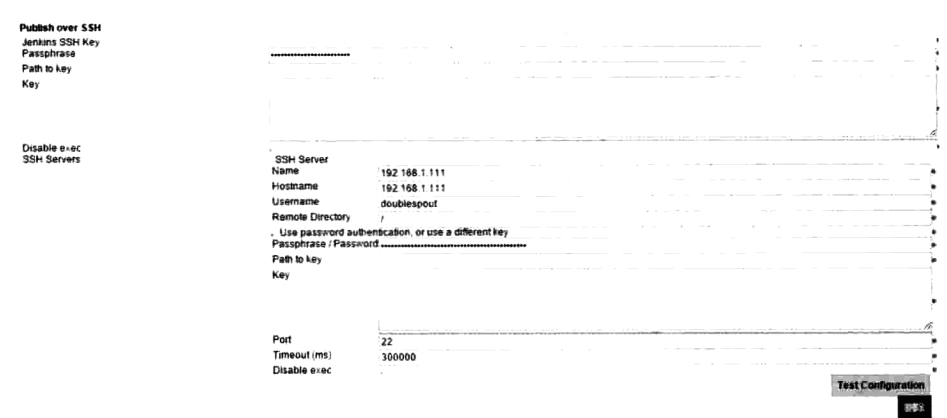


图1-5 Jenkins配置Publish over SSH插件界面

可以单击“Test Configuration”按钮来测试服务器是否能连接成功，服务器添加完毕之后，我们现在开始创建一个新的项目。

回到Jenkins主页，单击左上角的“新建”按钮就可以开启一个新项目，给项目起名node_access_count，选择创建一个自由风格的软件项目，单击“ok”按钮，就进入了此项目的创建页面，如图1-6所示。

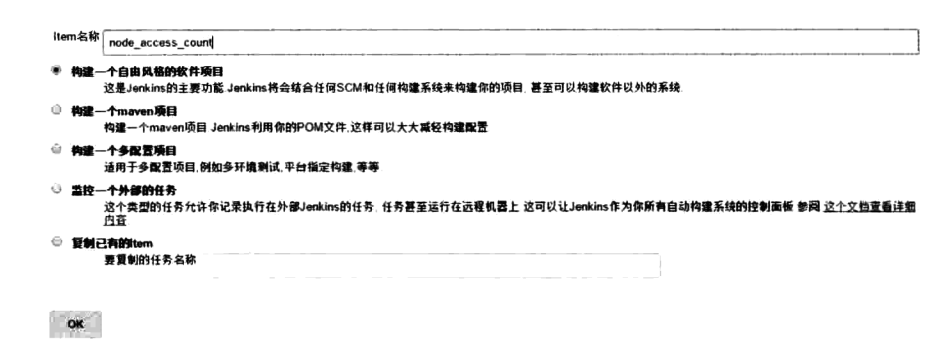


图1-6 Jenkins创建项目界面

在配置页，我们找到“源码管理”选项，然后填入GitHub上的源码地址，如图1-7所示。

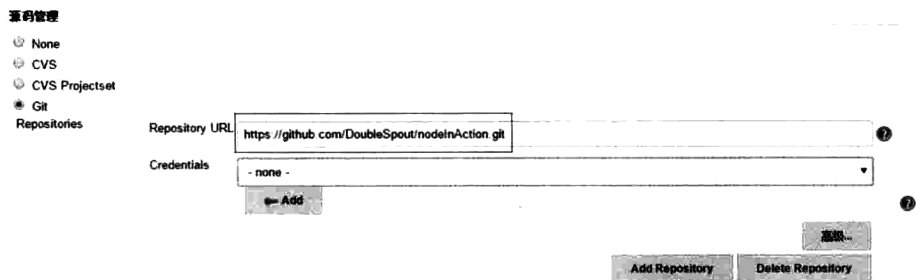


图1-7 Jenkins从GitHub拉取代码的界面

如图1-8所示，单击Add按钮，添加GitHub账号，我们就是通过这个账号来拉取源代码的。

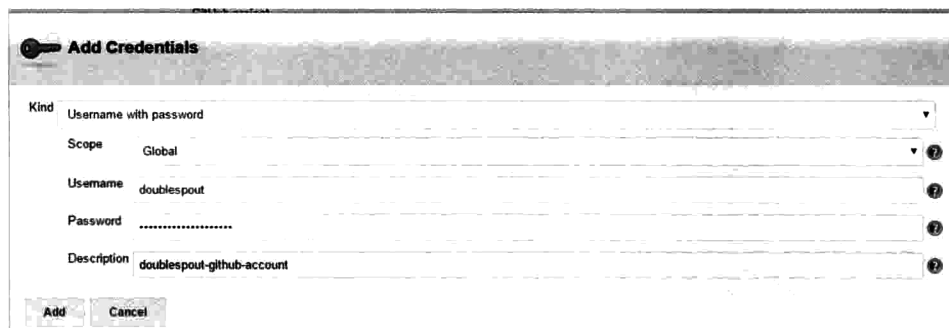


图1-8 Jenkins配置GitHub账号

把配置页向下滚动，在“构建”一栏处，单击下拉菜单，选择Execute shell。“构建”表示我们如何向生产服务器发布一个应用，简单地说，就是把原来手动要做的操作和要输入的命令通过配置来自动执行。发布一个Node.js程序由于不需要编译，所以大致的流程如下。

- (1) Jenkins从代码库（SVN或Git）获取最新代码。
- (2) 本地将所需的代码打包，需要排除一些文件，比如.git文件等。

（3）把代码包通过SSH发送到远程的服务器上。

（4）停止远程服务器的服务，删除远程服务器上的代码，解压缩新的代码包。

（5）通过新的package.json安装依赖包，然后重新启动服务。

增加构建步骤的界面如图1-9所示。

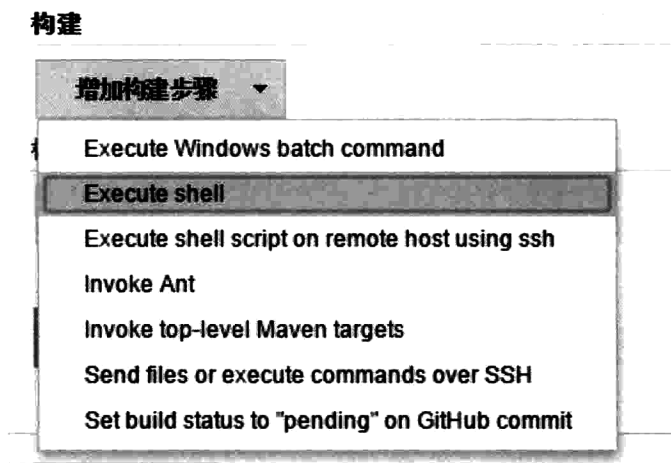


图1-9 增加构建步骤

从代码库获取最新代码是Jenkins自动执行的，每次构建都会去做，所以我们不必去配置，接着我们开始第一步：打包最新代码。我们在文本框中输入如下命令，先删除之前的tar包，然后重新打包代码。

```
rm -rf /var/jenkins_home/jobs/node_access_count/node_access_count.tar.gz
tar -zcvf /tmp/node_access_count.tar.gz -C /var/jenkins_home/jobs/node_access_count/workspace/docker_node . --exclude="*.git"
mv /tmp/node_access_count.tar.gz /var/jenkins_home/jobs/node_access_count/workspace/
```

然后我们需要把代码包发送到远程的生产服务器上，这时需要选择Send files...选项，如图1-10所示。

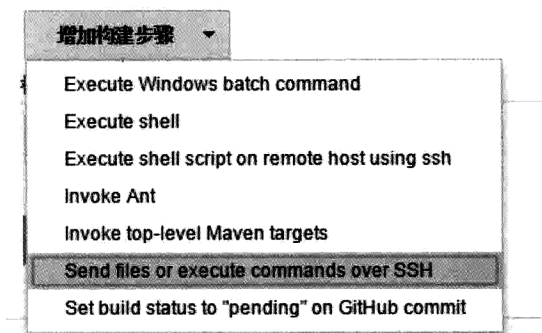


图1-10 选择Send files...选项

在SSH Server的下拉菜单中，选择我们刚刚添加的服务器。

在Source files一行中，填写我们要发送到远程服务器的文件，我们把刚才打包的文件名node_access_count.tar.gz填入，这里的工作路径是本项目下的workspace，在这里就是/var/jenkins_home/jobs/node_access_count/workspace/。

在Remote directory一行中，填写发送代码包的远程保存地址，我们在这里写入var/，我们创建这台服务器时填入的远程默认地址是“/”，所以我们发送到这台服务器上的代码包node_access_count.tar.gz会被保存在/var/node_access_count.tar.gz路径下。

接下来先把旧的代码删除，然后解压缩新的代码，并安装依赖包和重启服务，还记得我们之前启动的Container叫什么名字吗？我们在Exec command一栏中填入如下命令。

```
docker rm -f nodeCountAccess

mkdir /var/node
mkdir /var/node/docker_node
mkdir /var/log/pm2

rm -rf /var/node/docker_node/app.js
rm -rf /var/node/docker_node/package.json
```

Node.js实战（第2季）

```
tar -xvf /var/node_access_count.tar.gz -C /var/node/docker_node

docker run --rm -v /var/node/docker_node:/var/node/docker_node
-w /var/node/docker_node/ doublespout/node_pm2 cnpm install

docker run -d --name "nodeCountAccess" -p 8000:8000 -v /var/
node/docker_node:/var/node/docker_node -v /var/log/pm2:/root/.
pm2/logs/ --link redis-server:redis -w /var/node/docker_node/
doublespout/node_pm2 pm2 start --no-daemon app.js

rm -rf /var/node_access_count.tar.gz
```

下面我们简单说明这些命令的含义。

（1）`docker rm -f nodeCountAccess`命令，我们会强制删除之前运行的一个Container，第一次发布时会触发一个“没有这个Container”的错误，对此无须理会。

（2）两个`rm`操作则是删除原来项目的源代码，但是保留`node_modules`文件夹，免去了我们只改代码，重复去获取依赖包而导致发布程序时间过长的的问题。

（3）`mkdir/var/node/docker_node`，第一次启动会自动创建目录，如果已经存在，则无须理会。

（4）`tar`命令表示把源码解压缩到指定目录。

（5）两个`docker run ...`则先是执行`cnpm install`安装依赖包，然后将整个应用启动起来，注意这里我们启动的这个命令不要带上`-i -t`参数，否则Jenkins命令是无法退出的，直到超时报错。

（6）最后执行删除发送过来的代码包的操作。

如果服务器在国内，那么我们需要将`Exec timeout (ms)`设置得长一些，如图1-11所示，这样在Git操作和`cnpm`操作时便不会因为超时而报错。

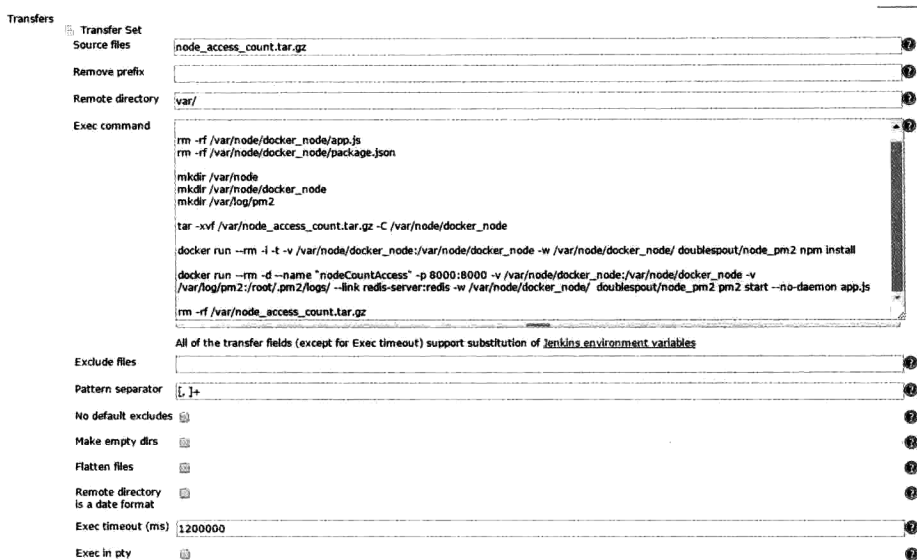


图1-11 Jenkins编写Shell脚本构建项目界面

至此，我们的项目配置完毕，单击页面底部的“保存”按钮将会返回到工程的首页，如图1-12所示，这时我们可以单击左侧的“立即构建”按钮，就可以看到构建历史中小球在闪烁并构建进度条。如果构建出错，那么构建历史中就会有红色小球出现，成功的话就是蓝色小球，黄色小球表示构建时虽然有错误，但最终还是成功的，不过这也是我们需要注意的。



图1-12 Jenkins项目发布进度条界面

单击如图1-12所示的“构建历史”，进入Console Output且能看到当前的构建进度，如图1-13所示，如果构建出错，那么也可以从这里找到错误原因，并修改构建配置。所以真正部署到生产服务器，由于权限、路径等，在执行命令上可能有所差异，我们在编写部署脚本时需要多尝试几次，仔细查看打印出错的信息，相应地修改脚本。



图1-13 Jenkins发布控制台输出界面

耐心等待一会，构建成功后，我们再打开浏览器，访问之前的Node.js访问计数路径，修改后的代码就被成功发布了。以后每次有代码改动，就再也不需要使用SSH登录到远程服务器，执行重复劳动的操作了，只需进入Jenkins，然后在项目主页单击“立即构建”按钮。另外，如果需要同时部署多台机器，那么只需要在构建时添加多台机器的配置脚本就可以了。

如果在最后一步构建失败，那么请大家自行在Jenkins的控制台查看出错的原因，然后相应地修改配置脚本。

1.13 小结

我们在本章初步学习了Docker的使用方法，基本上算是入门了。对于

Docker构建应用还有一个比较好的办法就是使用Dockerfile，Dockerfile更加清晰、简单，我们在使用Dockerfile之前还需要学习一下它的基本语法，这个就留给大家自行研究了。

Docker的魅力就是没有局限，如果我们偷懒，那么完全可以把整个运行环境，包括db、Nginx、Node.js运行环境等打包成一个镜像，这样每次部署只需要启动一个Container就可以了，不过这不是Docker推荐的做法。总之我们可以充分发挥想象，尽情体验Docker带给我们的乐趣。

本章最后介绍了利用Jenkins来管理发布我们的Node.js应用的方法，其实Jenkins不仅可以用来管理Node.js应用，还可以用来做其他项目，包括测试环境发布、需要远程执行的一系列shell脚本等，Jenkins可以解放我们的双手，自动化操作可以避免人为的失误所造成的损失。

1.14 参考文献

- <https://www.docker.com/whatisdocker> what is docker
- <http://www.cbinews.com/software/news/2015-01-20/228094.htm> 2015: Docker将走向深入应用
- <http://blog.docker.com/2014/06/why-you-dont-need-to-run-sshd-in-docker/> why you dont need to run sshd in docker
- <http://jpetazzo.github.io/2014/06/23/docker-ssh-considered-evil/> docker ssh considered evil
- <http://www.ibm.com/developerworks/cn/java/j-lo-jenkins/> 基于 Jenkins 快速搭建持续集成环境

第2章

开发OAuth2认证服务器

REST+JSON风格的API与SOAP+XML相比，其好处是调用更加灵活，也更容易扩展，另外JSON格式传输信息比XML减少约30%的数据量，效率更高。因此在搭建API服务器时，往往首选REST风格的API。当API服务器对外提供服务时，需要一种方式来验证API使用者的权限，我们选用了当前比较流行的OAuth2认证作为例子。本章主要介绍了搭建一个基本的API服务器所需要组件的方法和技术细节。

2.1 本章所用到的第三方模块

本章用到的第三方模块如下，读者可根据需要阅读该模块的详细文档。由于这些第三方模块可能在未来改变其接口参数形式，如果在学习本章的过程中出现问题，则可尝试安装附录中标出的指定版本。

1. Express

- 用途：Web框架。
- 版本：4.x。
- 主页：<http://expressjs.com/>。

2. js2xmlparser

- 用途：将JavaScript对象转换成XML格式的字符串。
- 版本：1.0.0。
- 主页：<https://www.npmjs.com/package/js2xmlparser>。

3. faker

- 用途：生成随机的测试数据。
- 版本：2.1.2。
- 主页：<https://www.npmjs.com/package/faker>。

4. request

- 用途：HTTP客户端。
- 版本：2.53.0。
- 主页：<https://www.npmjs.com/package/request>。

2.2 REST风格的API

REST（Representational State Transfer，REST）即表述性状态传递，是Roy Fielding博士于2000年在其博士论文中提出的一种软件架构风格。它是一种针对网络应用的设计和开发方式，可以降低开发的复杂性，提高系统的可伸缩性。

需要注意的是REST是设计风格而不是标准，其定义了一组体系架构原则，我们可以根据这些原则设计以系统资源为中心的Web服务，包括使用不同语言编写的客户端如何通过HTTP处理和传输资源状态。

REST架构风格最重要的架构约束有以下6个。

- 客户-服务器（Client-Server）：通信只能由客户端单方面发起，表现为请求-响应形式。
- 无状态（Stateless）：通信的会话状态（Session State）应该全部由客户端负责维护。

- 缓存（Cache）：响应内容可以在通信链的某处被缓存，以改善网络效率。
- 统一接口（Uniform Interface）：通信链的组件之间通过统一的接口相互通信，以提高交互的可见性。
- 分层系统（Layered System）：通过限制组件的行为（即每个组件只能“看到”与其交互的紧邻层），将架构分解为若干等级的层。
- 按需代码（Code-On-Demand，可选）：支持通过下载并执行一些代码（例如Java Applet、Flash或JavaScript），对客户端的功能进行扩展。

REST软件架构使用了CRUD原则，对于资源只需要4种行为：创建（Create）、获取（Read）、更新（Update）和销毁（DELETE），与之对应的是HTTP协议的4种请求方法：POST、GET、PUT和DELETE。

我们可以参考一下某网站提供的RESTful API。

- GET /api/articles.json：返回所有文章。
- GET /api/articles/count.json：返回文章数量。
- GET /api/articles/{id}.json：返回指定ID的文章。
- POST /api/articles.json：创建新文章。
- PUT /api/articles/{id}.json：更新指定ID的文章。
- GET /api/authors.json：返回所有作者。
- GET /api/tags.json：返回所有标签。
- DELETE /api/articles/{id}.json：删除指定ID的文章。

说明：#{id} 表示这部分是可变的，比如 id=123，则请求的URL为 /api/articles/123.json。

2.3 定义返回数据格式

API返回的数据格式，一般常见的有XML和JSON两种格式。在二者之中，JSON格式会显得更轻量级些，因此一些新兴的网络服务也将JSON设为

首选的数据格式，本章的实例也将使用JSON格式来返回数据。

在API返回的数据中，我们需要知道当前请求是否成功，如果成功，则需要返回详细的结果，如果失败，则返回相应的出错描述信息，比如我们可以这样定义：

成功时，返回格式如下：

```
{
  "status": "OK",
  "result": "相应的结果"
}
```

出错时，返回格式如下：

```
{
  "status": "Error",
  "error_code": "出错代码",
  "error_message": "详细的出错信息"
}
```

客户端可以通过判断status是否为“OK”来得知当前的请求是否成功。

2.4 实现简单的API

2.4.1 扩展Response对象

为了方便，我们需要给Response对象扩展两个方法：`apiSuccess()`和`apiError()`，分别用于响应请求成功和请求失败时的结果。

在初始化Express时，我们可以引入以下中间件来完成：

```
function extendAPIOutput (req, res, next) {

  //响应API成功结果
  res.apiSuccess = function (data) {
```

```
    res.json({
      status: 'OK',
      result: data
    });
  };

  //响应API出错结果，err是一个Error对象，
  //包含两个属性：error_code和error_message
  res.apiError = function (err) {
    res.json({
      status: 'Error',
      error_code: err.error_code || 'UNKNOWN',
      error_message: err.error_message || err.toString()
    });
  };

  next();
}

//初始化Express时引入中间件
app.use(extendAPIOutput);
```

我们还需要定义一个函数来生成统一的错误对象，以便作为`res.apiError()`的参数：

```
//code=出错代码，msg=出错描述信息
function createApiError (code, msg) {
  var err = new Error(msg);
  err.error_code = code;
  err.error_message = msg;
  return err;
}
```

在`res.apiError()`的代码中，如果我们传入的参数不是由`createApiError()`返回的对象，则API的相应结果中`error_code`为`UNKNOWN`，表示一个未知错误，在进行一些操作时，如果我们不确定所返回的出错信息的具体含义，则可以直接将回调函数的`err`对象直接响应给客户端：


```
function callback (err, ret) {
  if (err) return res.apiError(err);

  //其他操作...
}
```

2.4.2 统一处理出错信息

为了方便集中处理API请求过程中产生的错误，我们可以使用专门的出错处理中间件来完成：

```
function apiErrorHandle (err, req, res, next) {

  //如果有res.apiError()则使用其来输输出错信息
  if (typeof res.apiError === 'function') {
    return res.apiError(err);
  }

  next();
}

//初始化Express时引入中间件
app.use(apiErrorHandle);
```

在处理API请求时，如果需要返回出错信息，则可以直接使用`next(err)`；而不是`res.apiError(err)`了。

2.4.3 实现简单的API

接着我们就可以实现一个简单的API：

```
app.get('/api/articles.json', function (req, res, next) {

  queryArticles({
    author_id: req.query.author_id,
    $skip: req.query.$skip,
```

```
    $limit: req.query.$limit,  
    $sort: req.query.$sort  
  }, function (err, ret) {  
    if (err) return res.apiError(err);  
  
    res.apiSuccess({articles: ret});  
  });  
  
});
```

在上面的例子中，`queryArticles()`为我们模拟的一个查询数据库中文章列表的函数。

- 第一个参数是查询参数对象，比如参数中的`author_id`表示作者的ID，`$skip`和`$limit`分别表示数据的起始位置和数量，`$sort`表示数据的排序方式，这些参数均通过HTTP请求的QueryString来获取。
- 第二个参数是回调函数，回调函数接收两个参数：第一个参数是一个出错对象，表示是否出错，如果为空则表示查询成功；第二个参数为查询的结果。

在查询完数据之后，就可以通过`res.apiError()`和`res.apiSuccess()`来响应对应的结果。

由于本章重点是如何实现API服务器的框架，关于`queryArticles()`这些查询数据库的操作在此不做具体介绍。

2.4.4 API版本

随着系统的不断完善和升级，API接口会有所变动，但是为了保证旧的API接口还能正常使用（或者在一段过渡期内还能正常使用），需要在请求API时提供要请求的API版本号，一般常见的有两种方式。

- 在HTTP Header或请求参数中增加一个`api-version`参数，用于指定API版本号。

- 在API名称前面加上版本号前缀，比如/api/article.json的不同版本分别为/api/v1/article.json和/api/v2/article.json。

一般而言，如果API的名称和参数等都变动较大，则推荐使用版本号前缀的方式；如果变动不大，则可以通过api-version参数来指定不同的版本号。

2.5 关于OAuth认证

2.5.1 OAuth 2.0授权流程

OAuth 2.0授权流程如图2-1所示。

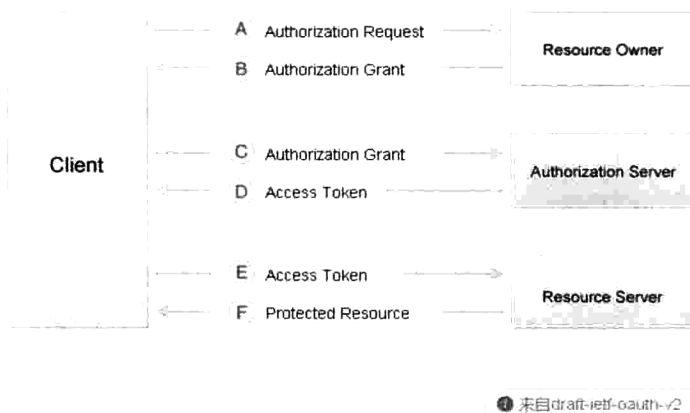


图2-1 OAuth 2.0授权流程图

其中，Client指第三方应用，Resource Owner指用户，Authorization Server指我们的授权服务器，Resource Server指API服务器。

2.5.2 OAuth 2.0授权详解

第1步引导需要授权的用户到Web授权页面：<https://example.com/oauth2/>

`authorize?client_id=YOUR_CLIENT_ID&response_type=code&redirect_uri=YOUR_REGISTERED_REDIRECT_URI`。

说明：

- `https://example.com/oauth2/authorize`为API服务提供方的授权页面；
- `client_id=YOUR_CLIENT_ID`中的YOUR_CLIENT_ID为API服务提供方给当前应用分配的app_key；
- `response_type=code`为返回的授权码类型，一般为code，如果有提供多种获取授权的方式，则会有其他值可选；
- `redirect_uri=YOUR_REGISTERED_REDIRECT_URI`中的YOUR_REGISTERED_REDIRECT_URI为授权成功后的回调地址，如果用户在此页面中点击了【确定授权】按钮，则API服务提供方会引导浏览器跳转到此地址，同时会在URL中加上参数`code=AUTHORIZATION_CODE`（用于获取access_token的code'）。

第2步，如果用户同意授权，则页面跳转至 `YOUR_REGISTERED_REDIRECT_URI/?code=CODE`，生成用于获取access_token的authorization_code，跳转回申请授权的应用。

说明：基于对安全的考虑，API服务提供方会对redirect_uri指定的回调地址进行检查，只有符合申请应用app_key时设置的回调地址规则才能正确显示授权页面，以免一些非法程序冒用当前应用来申请对用户的授权。

第3步，应用接收到第2步中回调的code之后，请求一下地址获得access_token：`https://example.com/oauth2/access_token?client_id=YOUR_CLIENT_ID&client_secret=YOUR_CLIENT_SECRET&grant_type=authorization_code&redirect_uri=YOUR_REGISTERED_REDIRECT_URI&code=CODE`。

说明：

- `client_id=YOUR_CLIENT_ID`中的YOUR_CLIENT_ID为API服务提供方给当前应用分配的app_key，通过它来唯一标识某个应用，这个值

一般是不可变的;

- `client_secret=YOUR_CLIENT_SECRET`中的`YOUR_CLIENT_SECRET`为当前应用对应的`app_secret`, 通过它和`app_key`来确定应用的身份, 当此`app_secret`泄露时, 可以为应用设置一个新的值;
- `redirect_uri=YOUR_REGISTERED_REDIRECT_URI`中的`YOUR_REGISTERED_REDIRECT_URI`为当前用于接收`authorization_code`的回调地址, 基于对安全的考虑, API服务提供方一般要求校验此值;
- `grant_type=authorization_code`为用来换取`access_token`的`code`的类型, 当有多种授权方式时才需要提供此参数;
- `code=CODE`中的`CODE`为API服务提供方回调时传过来的`authorization_code`。

如果换取`access_token`成功, 则返回结果格式如下:

```
{
  "access_token": "SlAV32hkKG",
  "remind_in": 3600,
  "expires_in": 3600
}
```

说明:

- `access_token`为本次授权申请所获得的授权码, 在执行后续的API请求时需要同时提供此`access_token`以验证当前用户的身份;
- `remind_in`和`expires_in`为此`access_token`的有效期限相关信息, 基于对安全的考虑, 每个`access_token`都是有有效期限限制的, 超过一定时间后应用需要重新向用户申请授权, 这个有效期一般与不同的API应用提供商及其对应用的评级有关。

第4步, 在成功获取到`access_token`后, 使用获得的OAuth 2.0 Access Token调用API, 在请求API时一般需要带上以下两个参数:

- `source`, 为当前应用的`app_key`;
- `access_token`, 为在第3步中获取到的`access_token`。

2.5.3 定义授权接口

如表2-1所示是某API服务提供方用于授权的接口文档。

表2-1 某API服务商的授权接口文档

接口	说明
OAuth2/authorize	请求用户授权Token
OAuth2/access_token	获取授权过的Access Token
OAuth2/get_token_info	授权信息查询接口
OAuth2/revookeoauth2	授权回收接口

其中，OAuth2/authorize和OAuth2/access_token是必需的，OAuth2/get_token_info和OAuth2/revookeoauth2是不同API服务提供方提供的额外接口，分别用于查询当前access_token的信息和回收授权（相当于注销）。

2.6 实现OAuth认证

2.6.1 OAuth2/authorize接口

1. 功能概述

应用请求用户的授权时，需要先跳转到此页面，由API服务提供方来检查用户是否已登录，并显示授权页面。授权页面中包含了申请授权的应用的名称和相关介绍，以及要申请的权限（比如获取用户的邮箱地址和以用户的身份发布内容等，不同的API服务提供方对权限的定义不同，拥有相应的权限才能成功请求相应的API），当用户在页面中点击【确认授权】按钮时，生成authorization_code并跳转回应用提供的URL。

2. 验证用户是否已登录

先编写一个简单中间件用于检查用户是否已登录：

```
function ensureLogin (req, res, next) {
  //先检查用户是否已在网站中登录
  //如果未登录则跳转到登录页面，在此处不作详细说明
  //如果已登录，记录用户相关的信息
  req.loginUserId = 'xxxxxx'; //当前登录的用户ID
  next();
}
```

3. 显示授权页面

跳转到授权页面时，应用需要提供client_id和redirect_uri两个参数，我们可以编写一个中间件专门用于校验此参数：

```
function missingParameterError (name) {
  return createApiError( 'MISSING_PARAMETER' , '缺少参数`' +
name + '`');
}

function redirectUriNotMatchError (url) {
  return createApiError( 'REDIRECT_URI_NOT_MATCH' , '回调地址不
正确: ' + url);
}

function checkAuthorizeParams (req, res, next) {
  //检查参数
  if (!req.query.client_id) {
    return next(missingParameterError( 'client_id' ));
  }
  if (!req.query.redirect_uri) {
    return next(missingParameterError( 'redirect_uri' ));
  }

  //验证client_id是否正确，并查询应用的详细信息
  getAppInfo(req.query.client_id, function (err, ret) {
    if (err) return next(err);

    req.appInfo = ret;
  });
}
```

```
//验证redirect_uri是否符合该应用设置的回调地址规则
verifyAppRedirectUri(req.query.client_id, req.query.
redirect_uri, function (err, ok) {
    if (err) return next(err);
    if (!ok) {
        return next(redirectUriNotMatchError(req.query.
redirect_uri));
    }

    next();
});
});
}
```

说明：**checkAuthorizeParams**中间件可以校验**client_id**和**redirect_uri**两个参数是否正确，同时可以查询申请授权的应用的详细信息，接着显示授权页面：

```
app.get('/OAuth2/authorize', ensureLogin,
checkAuthorizeParams, function (req, res, next) {
    res.locals.loginUserId = req.loginUserId;
    res.locals.appInfo = req.appInfo;
    res.render( 'authorize' );
});
```

模板**authorize**显示的效果大致如图2-2所示。

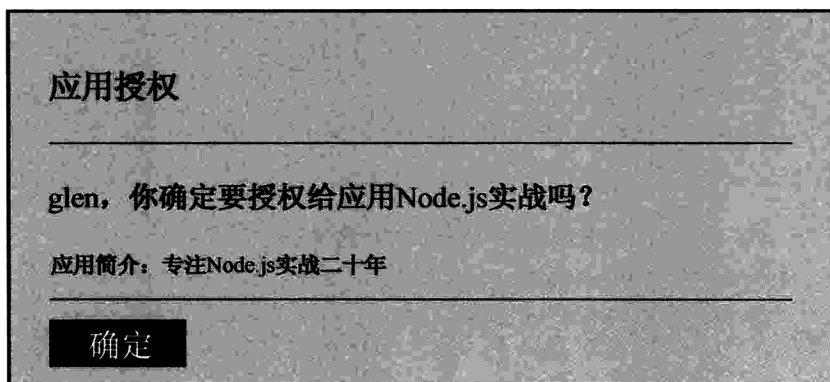


图2-2 应用授权页面

4. 应用授权页面

当用户点击【确定】按钮时，以POST方式提交表单。

5. 处理授权

当用户点击了授权页面的【确定】按钮时，生成authorization_code并跳转回来源应用：

```
app.post('/OAuth2/authorize', ensureLogin,
checkAuthorizeParams, function (req, res, next) {
    //生成authorization_code
    generateAuthorizationCode(req.loginUserId, req.query.client_id,
req.query.redirect_uri, function (err, ret) {
        if (err) return next(err);

        //跳转回来源应用
        res.redirect(addQueryParamsToUrl(req.query.redirect_uri, {
            code: ret
        }));
    });
});
```

addQueryParamsToUrl()函数用于在URL中增加一些参数，并返回新的URL，代码如下：

```
var parseUrl = require('url').parse;
var formatUrl = require('url').format;

function addQueryParamsToUrl (url, params) {
    var info = parseUrl(url, true);
    for (var i in params) {
        info.query[i] = params[i];
    }
    delete info.search;
    return formatUrl(info);
}
```

`generateAuthorizationCode()`用于生成唯一的`authorization_code`，通过这个`authorization_code`能找到对应的用户和请求授权的应用，在获取`access_token`时需要做校验。代码大致如下：

```
function randomString (size, chars) {
  size = size || 6;
  var codeString = chars || 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789';
  var maxNum = codeString.length + 1;
  var newPass = '';
  while (size > 0) {
    newPass += codeString.charAt(Math.floor(Math.random() *
maxNum));
    size--;
  }
  return newPass;
}

function generateAuthorizationCode (userId, appKey,
redirectUri, callback) {
  //生成code
  var code = randomString(20);

  //将code、userId、appKey、redirectUri 存储到数据库
  //此处省略相关代码

  callback(null, code);
}
```

2.6.2 OAuth2/access_token接口

应用拿到申请用户授权成功后的`authorization_code`，就可以通过OAuth2/`access_token`接口来获取`access_token`，代码大致如下：

```
app.post('/OAuth2/access_token', function (req, res, next) {
  //检查参数
```

```

    var client_id = req.body.client_id || req.query.client_id;
    var client_secret = req.body.client_secret || req.query.
client_secret;
    var redirect_uri = req.body.redirect_uri || req.query.redirect_
uri;

    var code = req.body.code || req.query.code;
    if (!client_id) return next(missingParameterError( 'client_
id' ));
    if (!client_secret) return next(missingParameterError( 'clie
nt_secret' ));
    if (!redirect_uri) return next(missingParameterError( 'redir
ect_uri' ));
    if (!code) return next(missingParameterError( 'code' ));

    //验证authorization_code
    verifyAuthorizationCode(code, client_id, client_secret,
redirect_uri, function (err, userId) {
        if (err) return next(err);

        //生成access_token
        generateAccessToken(userId, client_id, function (err,
accessToken) {
            if (err) return next(err);

            //生成access_token后需要删除旧的authorization_code
            deleteAuthorizationCode(code, function (err) {
                if (err) console.error(err);
            });

            res.apiSuccess({
                access_token: accessToken,
                expires_in: 3600 * 24 // access_token的有效期为1天
            });
        });
    });
});
});

```

verifyAuthorizationCode()用于验证authorization_code是否正确，如果不正

确则返回出错信息；如果正确，则返回对应的userId，代码大致如下：

```
function verifyAuthorizationCode (code, appKey, appSecret,
redirectUri, callback) {
    //从数据库中查找对应code的记录
    //检查appKey、appSecret和redirectUri是否正确
    //此处省略相关代码

    //userId为该code对应的userId
    callback(null, userId);
}
```

generateAccessToken()用于生成唯一的access_token，每个access_token均对应一个user_id和app_key。在以后处理API请求时，会先判断提交的client_id和access_token是否一致，再从中取出user_id，即可指定当前请求发起者的身份。代码大致如下：

```
function generateAccessToken (userId, appKey, callback) {
    //生成code
    var code = randomString(20);

    //将code, userId, appKey 存储到数据库
    //此处省略相关代码

    callback(null, code);
}
```

在使用authorization_code换取了access_token之后，该authorization_code应该失效，我们使用deleteAuthorizationCode()来删除它，代码大致如下：

```
function deleteAuthorizationCode (code, callback) {
    //从数据库中删除对应的code的记录
    //基础省略相关代码
    callback(null, code);
}
```

至此，整个OAuth授权的流程就完成了。

2.6.3 在处理API请求前验证Access Token

既然实现了OAuth认证，那么在处理API请求之前就需要验证access_token，以确认请求者的身份。除了OAuth2/authorize和/OAuth2/access_token这两个请求，所有请求在处理之前都需要使用以下中间件来验证：

```
function invalidParameterError (name) {
    return createApiError( 'INVALID_PARAMETER' , '参数' + name
+ '不正确');
}

function getAccessTokenInfo (token, callback) {
    //查询数据库中对应token的信息
    callback(null, info);
}

function verifyAccessToken (req, res, next) {
    var accessToken = (req.body && req.body.access_token) || req.
query.access_token;
    var source = (req.body && req.body.source) || req.query.
source;

    //检查参数
    if (!accessToken) return next(missingParameterError( 'acces
s_token' ));
    if (!source) return next(missingParameterError( 'source' ));

    //查询access_token的信息
    database.getAccessTokenInfo(accessToken, function (err,
tokenInfo) {
        if (err) return next(err);

        //检查appKey是否一致
        if (source !== tokenInfo.clientId) {
            return next(invalidParameterError( 'source' ));
        }

        //保存当前access_token的详细信息
```

```
    req.accessTokenInfo = tokenInfo;

    next();
  });
}
```

由于我们定义的API路径均为/api开头的，所以可以通过以下方法来统一引入此中间件：

```
app.use('/api', verifyAccessToken);
```

或者在注册路由时，单独引入此中间件：

```
app.get('/api/v1/articles.json', verifyAccessToken, function
(req, res, next) {
  //处理API请求
})
```

2.6.4 Access Token过期的问题

由于几乎所有的API都需要先验证access_token，而大量的access_token在使用一段时间后会过期，为了减轻数据库的压力，在验证access_token时我们可以通过一些小技巧来先过滤掉一些过期的access_token，再进行access_token有效性验证。

在生成access_token的函数generateAccessToken()中，我们可以让access_token带上一个过期时间戳，比如：

```
//获取秒时间戳
function getTimestamp () {
  return parseInt(Date.now() / 1000, 10);
}

//增加了expires参数，表示access_token的有效期，单位为秒
function generateAccessToken (userId, appKey, expires, callback) {
  //生成code，后面为
  var code = randomString(20) + '.' + (getTimestamp() +
```

```
expires);
```

```
    //将code, userId, appKey 存储到数据库
```

```
    //此处省略相关代码
```

```
    callback(null, code);
```

```
}
```

那么在验证access_token时，我们可以先取出其中的时间戳来判断是否已过期：

```
//从access_token中取出时间戳
```

```
function getTimestampFromAccessToken (token) {
```

```
    return Number(token.split( '.' ).pop());
```

```
}
```

```
//access_token已过期，错误
```

```
function accessTokenExpiredError () {
```

```
    return createApiError( 'ACCESS_TOKEN_EXPIRED' , 'access_
token expired' );
```

```
}
```

```
function verifyAccessToken (req, res, next) {
```

```
    var accessToken = (req.body && req.body.access_token) ||
req.query.access_token;
```

```
    var source = (req.body && req.body.source) || req.query.
source;
```

```
    //检查参数
```

```
    if (!accessToken) return next(missingParameterError( 'acces
s_token' ));
```

```
    if (!source) return next(missingParameterError( 'source' ));
```

```
    //验证access_token是否已过期
```

```
    if (getTimestamp() > getTimestampFromAccessToken(accessToke
n)) {
```

```
        return next(accessTokenExpiredError());
```

```
    }
```

```
//查询access_token的信息
database.getAccessTokenInfo(accessToken, function (err,
tokenInfo) {
    if (err) return next(err);

    //检查appKey是否一致
    if (source !== tokenInfo.clientId) {
        return next(invalidParameterError('source'));
    }

    //保存当前access_token的详细信息
    req.accessTokenInfo = tokenInfo;

    next();
});
}
```

2.7 实现API客户端

前面已经基本实现了API服务器的OAuth认证及API请求的处理，为了方便使用这些API，我们一般会发布一个API客户端，让开发者可以通过简单地配置参数完成OAuth授权申请和请求API的操作。

初始化API客户端时，需要提供以下三个参数。

- **appKey**：为API服务提供方分配的app_key。
- **appSecret**：为应用对应的app_secret。
- **callbackUrl**：为回调地址，即用户在授权页面点击【确认授权】按钮时跳转回来的地址，用于接收authorization_code，若设置为callbackUrl='http://example.com/auth/callback'，则在获取授权后跳转回来的地址是http://example.com/auth/callback?code= AUTHORIZATION_CODE（其中AUTHORIZATION_CODE为用于换取access_token的授权码）。


```
function APIClient (options) {
  this._appKey = options.appKey;
  this._appSecret = options.appSecret;
  this._callbackUrl = options.callbackUrl;
}
```

API客户端需要实现的第一个功能是获取用户授权，分为两部分：生成获取授权的跳转地址和通过authorization_code换取access_token。代码如下：

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

//定义请求API的地址
var API_URL = 'http://example.com' ;
var API_OAUTH2_AUTHORIZE = API_URL + '/OAuth2/authorize' ;
var API_OAUTH2_ACCESS_TOKEN = API_URL + '/OAuth2/access_
token' ;

//生成获取授权的跳转地址
APIClient.prototype.getRedirectUrl = function () {
  return addQueryParamsToUrl(API_OAUTH2_AUTHORIZE, {
    client_id: this._appKey,
    redirect_uri: this._callbackUrl
  });
};

//发送请求
APIClient.prototype._request = function (method, url, params,
callback) {
  method = method.toUpperCase();

  //如果已经获取了access_token，则字加上source和access_token两个参
数
  if (this._accessToken) {
    params.source = this._appKey;
    params.access_token = this._accessToken;
  }

  //根据不同的请求方法，生成用于request模块的参数
```

Node.js实战（第2季）

```
var requestParams = {
  method: method,
  url: url
};

if (method === 'GET' || method === 'HEAD') {
  requestParams.qs = params;
} else {
  requestParams.formData = params;
}

request(requestParams, function (err, res, body) {
  if (err) return callback(err);

  //解析返回的数据
  try {
    var data = JSON.parse(body.toString());
  } catch (err) {
    return callback(err);
  }

  //判断是否出错
  if (data.status !== 'OK') {
    return callback({
      code: data.error_code,
      message: data.error_message
    });
  }

  callback(null, data.result);
});

//获取access_token
APIClient.prototype.requestAccessToken = function (code,
callback) {
  var me = this;
  this._request('post', API_OAUTH2_ACCESS_TOKEN, {
    code: code,
```

```

    client_id: this._appKey,
    client_secret: this._appSecret,
    redirect_uri: this._callbackUrl
  }, function (err, ret) {
    //如果请求成功, 则保存获取得的access_token
    if (ret) me._accessToken = ret.access_token;
    callback(err, ret);
  });
};

```

说明：`addQueryParamsToUrl()`函数用于在URL中增加一些参数，并返回新的URL字符串，在前面有关于这个函数的具体实现方法。

由于获取`authorization_code`是通过跳转到某个地址，才从URL参数中获取到的，因此我们一般还需要一个Web页面用来接收此值，比如：

```

app.get('/auth/callback', function (req, res, next) {
  client.requestAccessToken(req.query.code, function (err,
ret) {
    if (err) return next(err);

    //ret.access_token即为获取的授权码

    //显示授权成功页面
    res.end('获取授权成功');
  });
});

```

说明：`client`为一个`APIClient`实例。

当然，我们也不一定需要在应用中实现请求用户授权的操作，只要能得到一个有效的`access_token`，即可请求提供的各种API，比如我们在初始化`APIClient`时增加一个`accessToken`参数：

```

function APIClient (options) {
  this._appKey = options.appKey;
  this._appSecret = options.appSecret;
  this._callbackUrl = options.callbackUrl;

```

```
    this._accessToken = options.accessToken;
  }
```

这样，如果不需要获取access_token，在初始化APIClient时传入appKey和accessToken两个参数即可。

在获取到access_token后，请求API就方便得多了，比如要实现查询所有文章的方法，代码如下：

```
var API_ARTICLES = API_URL + '/api/v1/articles';

APIClient.prototype.getArticles = function (params, callback)
{
    this._request('get', API_ARTICLES, params, callback);
};
```

说明：params为查询文章列表时需要的一些额外参数，比如指定查询位置和结果数量：{\$skip: 0, \$limit: 100}。

其他API方法参考以上例子实现即可。

2.8 API传输过程中的安全问题

这里的例子是通过HTTP协议来请求API服务的，而HTTP协议本身完全使用明文来传输所有数据，因此传输的数据很容易被第三方截获，若应用的app_secret不幸泄漏了，则可以通过重置app_secret来解决，若用户的access_token泄漏，则可以删除此access_token。

旧版本的OAuth认证要求在每个API请求中都附带一个“参数签名”，这样可以保证请求参数不被第三方篡改。另外，计算“参数签名”时增加了一个时间戳，可以保证即使数据被第三方完整截获，也无法通过重新发送这些数据给API服务器来达到“重放攻击”的目的。

以下是某API服务提供方对于OAuth认证时的“参数签名”的实现方法描述：

“所有OAuth请求使用同样的算法来生成（signature base string）签名字符串基串和签名。base string是把HTTP方法名、请求URL及请求参数用&字符连起来后做URL Encode编码。具体来讲，base string的前半部分依次由HTTP方法名、&、经过URL编码（url-encoded）处理后的请求URL及&组成。接下来把所有的请求参数包括POST方法体中的参数，经过排序（按参数名进行文本排序，如果参数名有重复则再按参数值进行重复项目排序），使用“%3D”替代“=”号，并且使用“%26”作为每个参数之间的分隔符，拼接成一个字符串。”

这个算法可以简单表示为：

```
httpMethod + "&" +
  url_encode( base_uri ) + "&" +
  sorted_query_params.each { | k, v |
    url_encode ( k ) + "%3D" +
    url_encode ( v )
  }.join("%26")
```

无论生成何种OAuth 1.0请求，生成BASE STRING的规则始终不变。微博要求所有OAuth请求都使用HMAC-SHA1算法生成签名。

OAuth 2.0实现规范规定必须使用HTTPS协议来传输数据，因此也不存在上面所提到的“中间人攻击”和“重放攻击”等问题。但HTTPS一般仅认证服务器端，并没有对客户端进行认证，在一些对安全要求比较高的场合，可能还需要在服务器端认证客户端的证书。

关于API传输过程中的安全问题，可阅读本章末尾的参考文献。

2.9 API请求频率限制

大多数API服务提供方都会设置一个请求频率限制，比如限定同一个app_key在一个小时内请求/api/articles不超过10000次，超过此数量时直接返回出错信息，即“超出一小时内请求频率限制”，这样可以保证系统资源不被某个应用霸占。

我们可以很容易地借助Redis来完成这个功能：

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

//连接的redis
var redisClient = redis.createClient();

function outOfRateLimitError () {
    return createApiError( 'OUT_OF_RATE_LIMIT' , '超出请求频率限制');
}

//生成检测请求频率的中间件
function generateRateLimiter (getKey, limit) {
    return function (req, res, next) {

        var source = req.body.source || req.query.source;
        var key = getKey(source);

        redisClient.incr(key, function (err, ret) {
            if (err) return next(err);
            if (ret > limit) return next(outOfRateLimitError());

            next();
        });
    };
}
```

通过generateRateLimiter()来生成用于检测某个API的请求中间件，其第一个参数getKey为一个函数，用来根据请求参数中的source及当前时间来生成一个唯一的Key，这个函数可以保证在某一段时间内，相同的source返回的Key都是一样的，接着我们就可以借助Redis来统计请求的次数，每次收到请求都把这个Key的值加1，再判断是否超出限制的数量。

假如要限制一个小时内的请求次数，则getKey()返回的Key可以是这样的：

```
hash(apiName, source) + ':' + date('YYYYMMDDHH')
```

其中`hash(apiName, source)`根据`apiName`和`source`生成了一个唯一的字符串（保证不与其他API或者不同的`api_key`产生冲突即可，可以直接使用MD5摘要）。`date('YYYYMMDDHH')`则根据当前时间返回精确到小时的日期字符串，比如2015年3月12日12点0分至当天12点59分返回的字符串为“2015032012”。将这两段字符串拼接起来就可以唯一标识某段时间内某应用对某API的请求。

以下是其简单的使用示例：

```
app.get('/api/v1/articles.json',
  verifyAccessToken,
  generateRateLimiter(function (source) {
    //计算key, hash() 和date() 方法自己实现
    return hash( '/api/v1/articles' , source) + ':' + date
    ('YYYYMMDDHH' );
  }, 10000),
  function (req, res, next) {
    //处理API请求
  });
```

2.10 让API返回结果支持不同的格式

2.10.1 通过后缀来指定返回的数据格式

假如让API结果支持不同的返回数据格式，比如查询文章列表时，`/api/articles.json`表示返回JSON格式的数据，而`/api/articles.xml`表示返回XML格式的数据，甚至根据不同的后缀名，可以支持更多格式，则要怎么实现呢？

首先我们在注册路由时，需要更改一下，比如原来注册路由是这样的：

```
app.get('/api/articles.json', function (req, res, next) {
  //处理API请求
});
```

现在需要把`.json`部分改为可变的，可以这样：

```
app.get('/api/articles.*', function (req, res, next) {
  //处理API请求
});
```

其中“*”表示此部分是可变的，在响应数据时，需要根据此可变部分来返回不同格式的数据。

上文的“扩展Response对象”部分，需要更改extendAPIOutput中间件：

```
var path = require('path');
var parseUrl = require('url').parse;

function extendAPIOutput (req, res, next) {

  //输出数据
  function output (data) {
    //取得请求的数据格式
    var type = path.extname(parseUrl(req.url).pathname);
    switch (type) {
      case '.xml':
        return res.xml(data);
      default:
        return res.json(data);
    }
  }

  //响应API成功结果
  res.apiSuccess = function (data) {
    output({
      status: 'OK',
      result: data
    });
  };

  //响应API出错结果，err是一个Error对象，
  //包含两个属性：error_code和error_message
  res.apiError = function (err) {
    output({
      status: 'Error',

```



```

        error_code: err.error_code || 'UNKNOWN' ,
        error_message: err.error_message || err.toString()
    });
};

next();
}

```

在extendAPIOutput中间件里，res.apiSuccess()和res.apiError()均统一使用output()来输出。在output()里会根据URL最后“*”部分指示的格式类型选择相应的输出方法：

- 如果为'.json'，则使用res.json()输出；
- 如果为'.xml'，则使用res.xml()输出；
- 如果为其他值，则默认使用res.json()输出。

若要支持更多的数据格式，则可以按照output()里面的格式进行扩展。

由于Response对象并没有xml()方法，所以我们还得先实现它：

```

var js2xmlparser = require('js2xmlparser');

res.xml = function (data) {
    res.setHeader('content-type', 'text/xml');
    res.end(js2xmlparser('data', data));
};

```

我们使用了js2xmlparser模块来将一个对象转换成XML格式的字符串，关于js2xmlparser模块的具体使用方法，可以参考其主页：<https://www.npmjs.com/package/js2xmlparser>。

2.10.2 通过Accept请求头来指定返回的数据格式

另外我们也可以采用Accept请求头代替后缀的方式，这样做的好处是资源的URL更统一，只需改变一下请求头即可适应不同的数据格式。

比如上例中在注册路由时可以改为这样：

```
app.get('/api/articles', function (req, res, next) {  
    //处理API请求  
});
```

extendAPIOutput中间件里面的**output()**函数代码改为这样：

```
//输出数据  
function output (data) {  
    //取得请求的数据格式  
    var type = req.accepts([ 'json' , 'xml' ]);  
    switch (type) {  
        case 'xml':  
            return res.xml(data);  
        default:  
            return res.json(data);  
    }  
}
```

说明：在Express中我们可以通过req.accepts()来检查优先接收的数据类型，在上面的例子中，请求头是Accept: Application/json时返回的值是json，请求头是Accept: Application/xml时返回的值是xml。对于其具体规则读者可以详细阅读Express文档。

2.11 生成随机的测试数据

在编写演示程序时，我们往往需要通过一些测试数据来预览显示效果，这时可以使用faker模块来生成一些随机数据，比如生成一些随机的文章数据：

```
var faker = require('faker');  
//设置语言为简体中文  
faker.locale = 'zh_CN';
```

```

var dataArticles = [];
var ARTICLE_NUM = 100;

//生成文章列表
for (var i = 0; i < ARTICLE_NUM; i++) {
  dataArticles.push({
    id: faker.random.uuid(),
    author: faker.name.findName(),
    title: faker.lorem.sentence(),
    createdAt: faker.date.past(),
    content: faker.lorem.paragraphs(10)
  });
}

function defaultNumber (n, d) {
  n = Number(n);
  return n > 0 ? n : d;
}

//查询文章列表的函数
function queryArticles (query, callback) {
  query.$skip = defaultNumber(query.$skip, 0);
  query.$limit = defaultNumber(query.$limit, 10);
  //返回指定范围的文章数据
  callback(null, dataArticles.slice(query.$skip, query.$skip
+ query.$limit));
}

```

在编写API服务器的Demo时，我们可以使用这个queryArticles()来模拟数据库查询，以便快速看到大概的效果。

2.12 小结

本章关于OAuth 2.0认证的介绍，主要参考了国内一些知名API服务提供商的实现文档，由于作者水平有限，如有错漏之处，欢迎指正。关于API服

服务器的安全问题涉及的知识领域较广，也是一个与攻击者斗智斗勇的漫长过程，读者可以根据本章末尾提供的参考文献链接自行研究。

由于篇幅所限，本章只讲解了实现API服务器实例中关键部分的代码，关于完整源码，读者可以通过访问项目地址<https://github.com/leizongmin/book-nodejs-in-action-season-2>来浏览，如果有问题，则可以在该项目主页上提交Issue。

- API服务端：<https://github.com/leizongmin/book-nodejs-in-action-season-2/tree/master/api-server>。
- API客户端：<https://github.com/leizongmin/book-nodejs-in-action-season-2/tree/master/api-client>。

2.13 参考文献及开源项目

- 《REST风格》-<http://baike.baidu.com/subview/1077487/16816671.htm>
- 《理解本真的REST架构风格》-<http://www.infoq.com/cn/articles/understanding-restful-style>
- 《六步实现Rest风格的API》-<http://blog.csdn.net/yanical/article/details/7856670>
- 《REST风格的软件架构》-<http://www.jianshu.com/p/ff18818a4c60>
- 《HTTP请求方法》-<http://www.w3cschool.cc/http/http-methods.html>
- 《OAuth 2.0规范》-<http://oauth.net/2/>
- 《新浪微博API授权机制说明》-<http://open.weibo.com/wiki/授权机制说明>
- 《程序员与黑客》-<http://www.infoq.com/cn/presentations/programmers-and-hackers>
- 《重放攻击》-<http://baike.baidu.com/view/1569933.htm>
- 《OAuth 2.0安全案例回顾》-<http://drops.wooyun.org/papers/598>
- 《腾讯开放平台第三方应用签名参数sig的说明》-<http://wiki.open.qq.com/wiki/第三方应用签名参数sig的说明>

qq.com/wiki/腾讯开放平台第三方应用签名参数sig的说明

- 《OAuth2授权原理》-<http://www.cnblogs.com/neutra/archive/2012/07/26/2609300.html>
- 《新浪微博OAuth授权机制说明》-<http://open.weibo.com/wiki/Oauth>
- oauth2-server（简单的OAuth2认证服务器）-<https://github.com/thomseddon/node-oauth2-server>
- <https://www.npmjs.com/package/weibo>
- oauth-sign（OAuth请求参数签名）-<https://www.npmjs.com/package/oauth-sign>

第3章

基于RabbitMQ搭建消息队列

本章主要介绍如何利用消息队列软件RabbitMQ来解决Web服务器或应用服务器间的通信问题。通常我们对处理大并发而带来的CPU或I/O密集型问题最好的控制方法就是使用消息队列。对于服务器间跨语言通信，以前我们一般使用XMLRPC，现在比较流行HTTP协议的RESTful方式，而使用RabbitMQ也能够很灵活地处理这些事情。

在学习本章之前，读者需要对Linux基本命令行操作、Express框架、Python语言有简单了解。

3.1 什么是消息队列，消息队列的优势

大家对队列肯定都不陌生，例如去KFC排队就餐，去银行排队取号办理业务，等等。我们在现实生活中遇到人流过多的情况时，一般用排队的方式解决抢占资源的情况，其实这种方式同样适用于互联网领域。

设想下，如果同时有上千个客户端要求处理某件事情，没有队列的话，我们的CPU就会像KFC的服务员那样，给A配了一个汉堡，接着切换到B，给B再配一杯可乐……然后CPU就因为频繁切换任务而导致执行效率低下，前面

的客户端还没来得及处理，后面的就又上来了，最终导致大量的连接出现超时错误。

理解队列的概念后，我们就要解释“消息”了，消息是在两台计算机间传送的数据单位。消息可以非常简单，例如只包含文本字符串；也可以更复杂，可能包含嵌入对象（比如XML或JSON）。对于消息，大家可以理解为去KFC点餐的菜单，告诉服务员你想要什么。

消息队列具有如下几种优势。

1、应用解耦

在项目启动之初，预测项目将会碰到什么需求是极其困难的。消息队列在处理过程的中间插入了一个隐含的基于数据的接口层，两边的处理过程都要实现这一接口。这允许你独立地扩展或修改两边的处理过程，只要确保它们遵守同样的接口约定。

2、冗余存储

处理数据有时会失败。除非数据被持久化，否则将永远丢失。消息队列把数据进行持久化直到它们被完全处理，通过这一方式避免了数据丢失的风险。在被许多消息队列所采用的“插入-获取-删除”范式中，把一个消息从队列中删除之前，需要你的处理过程明确指出该消息已经被处理完毕。

3、可扩展性

因为消息队列解耦了处理过程，所以增大消息入队和处理的频率是很容易的，不需要改变代码，不需要调节参数，扩展就像调整风扇按钮一样简单。

4、灵活性和峰值处理能力

当你的应用访问流量突然迅速攀升时，当然，应用仍然需要继续提供服务，但是这样的突发流量并不常见，如果以能处理这类峰值访问为标准来投入资源并为其随时待命，无疑是巨大的浪费。使用消息队列能够使关键组件顶住突发增长的访问压力，而不因超出负荷的请求而完全崩溃。

5、可恢复性

体系的一部分组件失效时，不会影响到整个系统。消息队列降低了进程间的耦合度，所以即使一个处理消息的进程挂掉，加入队列中的消息仍然可以在系统恢复后被处理。而这种允许重试或者延后处理请求的能力通常是造就用户略感不便、沮丧透顶的体验的关键要素。

6、送达保证

消息队列提供的冗余机制保证了消息能被实际地处理。获取一个消息只是“预定”了这个消息，暂时把它移出了队列。除非客户端明确表示已经处理完了这个消息，否则这个消息会被放回队列中，经过一段时间后再次被处理。

7、排序保证

在某些情况下，数据处理的顺序都很重要。消息队列本来就是排序的，并且能保证数据会按照特定的顺序来处理。RabbitMQ保证了消息通过FIFO（先进先出）的顺序来处理。

8、缓冲

在任何重要的系统中，都会有不同处理时间的场景。例如，加载一张图片比应用过滤器要花费更少的时间。消息队列通过一个缓冲层来帮助任务最高效率地执行，写入队列的处理尽可能快速，它不受队列另一端消费者的处理速度的约束，这样有助于提升整体系统的性能，不至于由于消费速度慢而导致流程不顺畅。

9、理解数据流

在分布式系统里，用户操作的时间长短，会关乎用户体验。消息队列通过消息被处理的频率和时长，来方便地定位那些处理时间过长的服务。

10、异步通信

大多数时候，你不想也不需要立即处理消息。消息队列提供了异步处理

机制，允许你把一个消息放入队列，但不立即处理它，然后在你乐意时再去处理它们。

相信上述10个原因，使得消息队列成为在进程或应用之间进行通信的较好形式，队列是创建强大的分布式应用的关键。

3.2 安装和启动RabbitMQ

RabbitMQ是消息队列产品，它虽然不是速度性能最快的，却是应用相当广泛、稳定的，而且由于它是由Erlang语言开发的，所以具有天生的分布式优势，这些都是我推荐RabbitMQ部署在生产环境中的理由，官网对RabbitMQ的定义非常简单，如下所述。

- 为应用而生的强大的消息队列。
- 使用简单。
- 支持跨平台。
- 支持大量的开发平台和语言。
- 开源，有社区支持（言下之意就是免费哦）。

关于各操作系统下载和启动的方式，官网有比较详细的文档，这里就不再赘述了，访问地址为<http://www.rabbitmq.com/download.html>，这里主要介绍如何使用第1章的Docker来安装和启动它，手握利器也要加以善用。

```
#截稿时，RabbitMQ的最新版本是3.4.3
$ sudo docker pull rabbitmq
#启动rabbitmq服务
$ docker run -d -e RABBITMQ_NODENAME=my-rabbit --name some-
rabbit -p 5672:5672 rabbitmq:3
```

用了Docker我们就可以更加专注地开发业务代码了，不用因安装环境而浪费时间。下面我们将学习RabbitMQ的各种队列。

3.3 RabbitMQ的Hello World

要连接RabbitMQ，我们需要安装连接包。依次执行命令，创建环境并安装依赖，这里使用官方推荐的npm包amqplib。

```
$ mkdir /var/node
$ mkdir /var/node/rabbit_hello
$ cd /var/node
$ npm install amqplib
```

我们打算建立这样一个队列，一个生产者往队列中填充数据，一个消费者对队列的数据进行消费，如图3-1所示。



图3-1 RabbitMQ一个生产者对应一个消费者的示意图

写一个Hello World示例的服务器部分，并把它保存在/var/node/rabbit_hello/server.js中。

```
var amqp = require('amqplib');
amqp.connect('amqp://127.0.0.1').then(function(conn) {      (1)
  process.once('SIGINT', function() { conn.close(); });      (2)
  return conn.createChannel().then(function(ch) {            (3)

    var ok = ch.assertQueue('hello', {durable: false});      (4)

    ok = ok.then(function(_qok) {
      return ch.consume('hello', function(msg) {              (5)
        console.log(" [x] Received '%s'", msg.content.
toString());
        }, {noAck: true});
      });
    });
  });
```

```

        return ok.then(function(_consumeOk) {
            console.log(' [*] Waiting for messages. To exit press
CTRL+C');
        });
    });
}).then(null, console.warn);

```

在解释上述代码之前，需要简单解释一下then，它是Node.js用来处理异步回调的方法之一。一般我们处理Node.js异步回调嵌套的方法有两种：Promise和async，我们这次用的amqplib包处理异步就推荐使用Promise方式。

那Promise究竟是什么呢？

Promise是对异步编程的一种抽象实现。它是一个代理对象，代表一个必须进行异步处理的函数返回的值或抛出的异常。callback是编写Node.js异步代码最简单的机制，可是用这种原始的callback必须以牺牲控制流、异常处理和函数语义为代价，而我们在同步代码中已经习惯了它们的存在，Promises能带它们回来。

Promises对象的核心部件是它的then方法。

我们可以用这个方法从异步操作中得到返回值（传说中的履约值）或抛出的异常（传说中的拒绝的理由）。then方法有两个可选的参数，它们都是callback函数，分别是onFulfilled和onRejected，现在让代码说话：

```

var promise = doSomethingAync()
promise.then(onFulfilled, onRejected)

```

Promises被解决时（异步处理已经完成）会调用onFulfilled或onRejected。因为只会有一种结果，所以这两个函数中仅有一个会被触发。

下面是一个readfile返回Promises的例子，同时then之后返回的仍然是那个Promises对象，也就是说我们可以像下面这样进行链式调用。

```

var promise = readFile()

var promise2 = promise.then(function (data) {

```

```
//如果读取文件成功，则开始读取另外一个文件readAnotherFile
return readAnotherFile()
}, function (err) {
    //如果读取出错，则打印错误信息，并且仍然继续读取另外一个文件
    readAnotherFile
    console.error(err)
    return readAnotherFile()
})
//将读取的另外一个文件（readAnotherFile）异步的处理结果打印出来
.then(console.log, console.error)
```

使用**Promise**最显著的特性就是我们不必对嵌套的所有异常都做显式处理，看下面的伪代码。

```
doThisAsync()
    .then(doThatAsync)
    .then(doanotherAsync)
    .then(null, console.error)
```

上述doThisAsync、doThatAsync或doanotherAsync异步中的任何一个错误，都会被最后的then语句捕获并处理。

现在我们回到Hello World示例代码处。

代码（1）表示我们连接本地127.0.0.1的RabbitMQ队列，并返回一个**Promise**对象。

代码（2）表示接收CTRL+C的退出信号时，关闭和RabbitMQ的连接conn.close()。

代码（3）处的conn.createChannel()表示创建一个通道。

在代码（4）处，我们通过ch.assertQueue在这个通道上监听hello队列，并设置durable队列持久化为false，表示队列保存在内存中，最后返回一个**Promise**对象。

代码（5）用于让通道消费hello队列，并写上处理函数，打印消息数据，同时返回一个**Promise**。

代码（6）用于在设置监听消费成功之后，打印一行文本，表示服务器正常工作，等待客户端的数据。

代码（7）用于当操作过程中有错误时，执行`console.warn`打印错误。

解读完服务器的代码后，我们创建`client.js`来发送消息。

```
var amqp = require('amqplib');
var when = require('when');

amqp.connect('amqp://localhost').then(function(conn) {           (1)
  return when(conn.createChannel()).then(function(ch) {           (2)
    var q = 'hello';
    var msg = 'Hello World!';

    var ok = ch.assertQueue(q, {durable: false});                 (3)

    return ok.then(function(_qok) {                                (4)
      ch.sendToQueue(q, new Buffer(msg));
      console.log(" [x] Sent '%s'", msg);
      return ch.close();
    });
  })).ensure(function() { conn.close(); });                         (5)
}).then(null, console.warn);
```

我们还需要安装一下`when`这个模块才能让`client.js`工作，执行`npm install when`。`when`模块是一个高性能的稳定的实现`Promise`异步的处理包，我们接着解释客户端代码的运行流程。

代码（1），连接本地的`RabbitMQ`队列，这个和服务器代码一样。

代码（2），用`when(x)`包装一个`Promise`对象，仍然返回一个`Promise`对象，这边我们创建一个通道。

代码（3），我们同样把通道绑定到`hello`队列，并且设定持久化为`false`。

代码（4），我们向这个队列发送`Buffer`，内容为`'Hello World!'`，发送完毕之后关闭通道。

代码（5），在通道关闭之后，我们把整个连接也关闭，`ensure`就类似执行扫尾工作的函数，是`promise.finally`的别名。

接着进入命令行，执行如下命令来启动消息队列服务器。

```
$ node server.js
[*] Waiting for messages. To exit press CTRL+C
```

然后启动客户端，发送消息给服务器。

```
$ node client.js
[x] Sent 'Hello World!'
```

客户端消息发送完毕后，就自动退出了，这时切换到服务器的ssh窗口，我们看到服务器这边打印了客户端发送过来的数据。

```
[*] Waiting for messages. To exit press CTRL+C
[x] Received 'Hello World!'
```

一个简单的RabbitMQ例子就跑起来了。接下来我们要分别学习几种不一样的队列样式，这些队列在日常开发中都非常有用。

3.4 RabbitMQ的工作队列

现在我们来看一个稍微复杂点的队列，一个生产者配合多个消费者的队列，这样的队列场景可能在日常生产环境中使用得比较多，将一个复杂的任务负载均衡到各个节点，相比只有一个消费者的队列，这样不至于队列堆积过长，同时也能保证队列的响应时间，如图3-2所示，这条队列拥有两个消费者。



图3-2 RabbitMQ一个生产者轮询多个消费者的示意图

在下面的所有示例中，我们将不再使用Promise风格的代码，而使用更加普遍的callback方式来展现，3.3节的Promise示例主要是想让读者了解Node.js处理异步回调的另一种方式。

首先是server的代码，保存为receive.js。

```
var amqp = require('amqplib/callback_api'); (1)

function bail(err, conn) { (2)
  console.error(err);
  if (conn) conn.close(function() { process.exit(1); });
}

function on_connect(err, conn) { (3)
  if (err !== null) return bail(err);
  process.once('SIGINT', function() { conn.close(); }); (4)

  var q = 'task_queue';

  conn.createChannel(function(err, ch) { (5)
    if (err !== null) return bail(err, conn);
    ch.assertQueue(q, {durable: true}, function(err, _ok) { (6)
      ch.consume(q, doWork, {noAck: false}); (7)
      console.log(" [*] Waiting for messages. To exit press
CTRL+C ");
    });

    function doWork(msg) { (8)
      var body = msg.content.toString();
      console.log(" [x] Received '%s'", body);
      var secs = body.split('.').length - 1; (9)
      setTimeout(function() {
        console.log(" [x] Done");
        ch.ack(msg);
      }, secs * 1000);
    }
  });
}
```

```
}  
  
amqp.connect('amqp://localhost', on_connect);
```

针对上述代码简单做一下解释，如下所述。

- （1）表示引入回调函数的API对象来处理RabbitMQ队列。
- （2）定义了出错的函数，出现错误后，会打印错误并且关闭连接，退出进程。
- （3）定义了程序成功启动并且成功连入RabbitMQ后执行的回调函数，包括接收数据、处理异常等操作。
- （4）监听进程信号，进程接收到SIGINT信号后，将关闭连接，SIGINT信号就是我们熟知的CTRL+C。
- （5）对连接conn对象执行创建channel的操作，创建成功后将做监听操作。
- （6）断言监听队列q，即名为task_queue的队列。
- （7）在收到q队列消息后，执行ch.consume()方法来把这部分数据丢到doWork方法中去消费，并且将noAck设置为false，表示对消费的结果做出响应。
- （8）定义doWork函数，用来消费数据，接收到数据后，将数据toString()转为字符串，然后打印数据。
- （9）根据数据字符串中“.”的个数，模拟处理这个数据所要消耗的描述，在等待secs*1000s之后，将msg响应回客户端。

下面我们来看将数据发送到队列的客户端的代码，保存为new_task.js。

```
var amqp = require('amqplib/callback_api');  
  
function bail(err, conn) {
```



```

    console.error(err);
    if (conn) conn.close(function() { process.exit(1); });
  }

  function on_connect(err, conn) {
    if (err !== null) return bail(err);

    var q = 'task_queue'; // (1)

    conn.createChannel(function(err, ch) {
      if (err !== null) return bail(err, conn); // (2)
      ch.assertQueue(q, {durable: true}, function(err, _ok) { // (3)
        if (err !== null) return bail(err, conn);
        var msg = process.argv.slice(2).join(' ') || "Hello
World! "; // (4)
        ch.sendToQueue(q, new Buffer(msg), {persistent: true}); // (5)
        console.log(" [x] Sent '%s'", msg);
        ch.close(function() { conn.close(); });
      });
    });
  }

  amqp.connect('amqp://localhost', on_connect);

```

客户端的代码同服务器的代码相似，解释如下。

- (1) 定义队列名`task_queue`，这里要和服务器那边的名字相同。
- (2) 在创建channel出错后执行关闭连接和退出进程操作。
- (3) 连入队列q，然后传入回调函数。
- (4) 接收进程启动参数，如果没有参数，则使用Hello World!字符串。
- (5) 将消息msg发送到队列中。

我们分别执行两次如下命令，启动2个receive.js来处理任务。

```
$ node receive.js
```

```
[*] Waiting for messages. To exit press CTRL+C
```

```
$ node receive.js
```

```
[*] Waiting for messages. To exit press CTRL+C
```

然后执行多次客户端来发送消息，查看两个receive.js的打印数据。

```
$ node new_task.js First message.
```

```
$ node new_task.js Second message..
```

```
$ node new_task.js Third message...
```

```
$ node new_task.js Fourth message....
```

```
$ node new_task.js Fifth message.....
```

第一个receive.js打印的数据如下。

```
[*] Waiting for messages. To exit press CTRL+C
```

```
[x] Received 'First message.'
```

```
[x] Done
```

```
[x] Received 'Third message...'
```

```
[x] Done
```

```
[x] Received 'Fifth message.....'
```

```
[x] Done
```

第二个receive.js打印的数据如下。

```
[*] Waiting for messages. To exit press CTRL+C
```

```
[x] Received 'Second message..'
```

```
[x] Done
```

```
[x] Received 'Fourth message....'
```

```
[x] Done
```

这样就将任务平均分给两个消费者来处理了，随着任务量的增大，可以逐步地增加消费者来增强队列的计算能力。

3.5 RabbitMQ的PUB/SUB队列

我们在3.4节将一个比较复杂的任务轮询分配给多个消费者来处理，这样

有助于减轻整个系统的负载，但也可能需求是一个消息有多个消费者订阅，对于这样的情境我们就需要使用到PUB/SUB了，队列如图3-3所示。

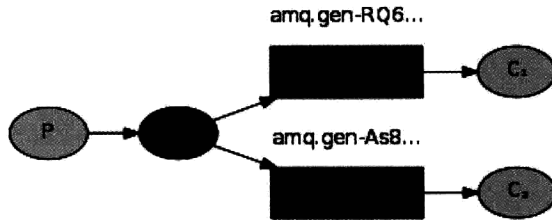


图3-3 RabbitMQ一个生产者广播多个消费者的示意图

图3-3中的两个消费者，C1将会记录日志，C2将会打印日志，每个消费者都将接收到数据包。RabbitMQ中的核心思想就是每个生产者都不直接将消息丢入队列中，甚至于生产者根本不知道这个消息将会被送到哪个或哪几个消费者手中，这样的解耦思想有利于我们的整个系统。

所以在这个系统中，生产者只是把消息传递给Exchange。Exchange非常简单，一边连接着生产者，接收生产者发送过来的数据；另一边连接着队列，负责把数据推送到队列中去。Exchange必须知道它将对接收到的数据做什么处理，例如将数据推送到指定的队列，又或者推送到多条队列中，等等，我们将这样类似的行为称为Exchange类型。这些类型是预设好的，有如下几种类型供我们选择：direct、topic、headers和fanout。

本节我们将使用fanout类型的Exchange来为整个系统服务，fanout中文直译就是扇出，表示将消息广播出去。

我们先看生产者的代码，保存为emit_log.js。

```

var amqp = require('amqplib/callback_api');

function bail(err, conn) {
  console.error(err);
  if (conn) conn.close(function() { process.exit(1); });
}
  
```

```
function on_connect(err, conn) {
  if (err !== null) return bail(err);

  var ex = 'logs';                                     (1)

  function on_channel_open(err, ch) {
    if (err !== null) return bail(err, conn);
    ch.assertExchange(ex, 'fanout', {durable: false}); (2)
    var msg = process.argv.slice(2).join(' ') ||
      'info: Hello World!';
    ch.publish(ex, '', new Buffer(msg));                (3)
    console.log(" [x] Sent '%s'", msg);
    ch.close(function() { conn.close(); });
  }

  conn.createChannel(on_channel_open);
}

amqp.connect('amqp://localhost', on_connect);
```

生产者的代码较之前没有什么大的变化，主要的区别就是生产者不直接把数据推送给队列，而是推送给Exchange。

(1) 命名Exchange为logs。

(2) 定义Exchange，命名为logs，类型为fanout，durable持久化队列为false。

(3) 将消息msg推送给Exchange节点，其中publish函数的第二个参数为路由配置，我们在3.6节中会对其详细说明。

当消息推送给Exchange后，消费者就需要从队列中获取消息并做处理，我们保存消费者的代码为receive_logs.js。

```
var amqp = require('amqplib/callback_api');

function bail(err, conn) {
  console.error(err);
```

```

    if (conn) conn.close(function() { process.exit(1); });
  }

  function on_connect(err, conn) {
    if (err !== null) return bail(err);
    process.once('SIGINT', function() { conn.close(); });

    var ex = 'logs'; (1)

    function on_channel_open(err, ch) {
      if (err !== null) return bail(err, conn);
      ch.assertExchange(ex, 'fanout', {durable: false}, function
(err){
        if (err !== null) return bail(err, conn);
        ch.assertQueue('', {exclusive: true}, function(err,
ok) { (2)
          var q = ok.queue; (3)
          ch.bindQueue(q, ex, ''); (4)
          ch.consume(q, logMessage, {noAck: true},
function(err, ok) { (5)
            if (err !== null) return bail(err, conn);
            console.log(" [*] Waiting for logs. To exit press
CTRL+C. ");
          });
        });
      })
    }

    function logMessage(msg) { (6)
      if (msg)
        console.log(" [x] '%s'", msg.content.toString());
    }

    conn.createChannel(on_channel_open);
  }

  amqp.connect('amqp://localhost', on_connect);

```

之前消费的队列都是命名过的，这里不需要对队列命名，让RabbitMQ随机给队列命名即可，我们只需在声明队列时不传入名称，就可以生成一个有随机名称的队列。

（1）声明fanout类型的Exchange，命名为logs。

（2）声明随机名称队列，exclusive参数为true表示当消费者断开队列连接时，此队列就会删除。

（3）通过ok.queue获取队列对象。

（4）将队列和Exchange绑定在一起，第三个参数是路由配置，我们暂时留空，在3.6节会有说明。

（5）开始消费队列中的数据，这里我们还定义了消费完成后的回调函数。

（6）定义了如何消费这些数据的logMessage函数。

我们先启动两个receive_logs.js，等待消费队列数据。

```
$ node receive_logs.js
[*] Waiting for logs. To exit press CTRL+C.

$ node receive_logs.js
[*] Waiting for logs. To exit press CTRL+C.
```

然后启动生产者，向这两个消费者广播数据，我们分别抛出3条Hello World消息。

```
$ node emit_log.js
[x] Sent 'info: Hello World!'
$ node emit_log.js
[x] Sent 'info: Hello World!'
$ node emit_log.js
[x] Sent 'info: Hello World!'
```

两个消费者的打印信息分别如下，我们成功地将消息广播出去了。

第一个receive_logs进程打印的信息如下。

```
[x] 'info: Hello World!'
[x] 'info: Hello World!'
[x] 'info: Hello World!'
```

第二个receive_logs进程打印的信息如下。

```
[x] 'info: Hello World!'
[x] 'info: Hello World!'
[x] 'info: Hello World!'
```

3.6 RabbitMQ的队列路由

我们在3.5节实现了生产者对多个消费者的广播消息，但实际上还有很多情况，我们的消费者是多种多样的，比如我们有对日志error专门分析的消费者，有对日志消息纯打印的消费者，等等，这些消费者要根据自己的需要去获取队列里的数据，这样我们就要用到Exchange的路由功能了，模型图如图3-4所示。

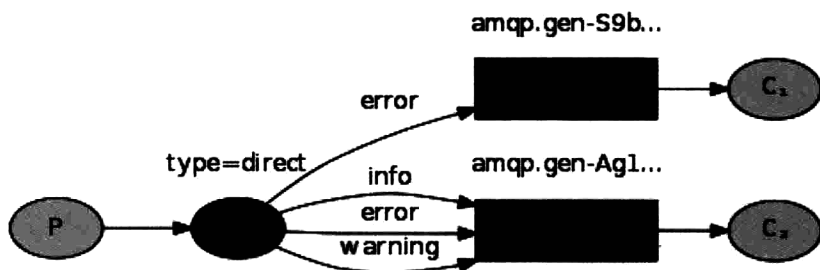


图3-4 一个生产者路由多个消费者的示意图

要使用Exchange的路由功能，我们就需要在定义Exchange时修改它的类型了，3.5节的fanout类型在这里已经不适用了，它只能无脑地广播，所以我们需要将Exchange修改为direct类型。direct类型的Exchange算法很简单，它会

把消息推送到绑定这个路由key的队列中去。简单地说，就是生产者将消息和路由的key推送给Exchange，Exchange则根据哪个或哪几个消费队列绑定了这个路由key，把消息再推送到这些队列中，供消费者消费。

我们先看生产者的代码，保存为emit_log_direct.js。

```
var amqp = require('amqplib/callback_api');

var args = process.argv.slice(2);
var severity = (args.length > 0) ? args[0] : 'info';      (1)
var message = args.slice(1).join(' ') || 'Hello World!';

function bail(err, conn) {
    console.error(err);
    if (conn) conn.close(function() { process.exit(1); });
}

function on_connect(err, conn) {
    if (err !== null) return bail(err);

    var ex = 'direct_logs';                                (2)
    var exopts = {durable: false};

    function on_channel_open(err, ch) {
        if (err !== null) return bail(err, conn);
        ch.assertExchange(ex, 'direct', exopts, function(err,
ok) {                                                       (3)
            ch.publish(ex, severity, new Buffer(message));    (4)
            ch.close(function() { conn.close(); });
        });
    }
    conn.createChannel(on_channel_open);
}

amqp.connect('amqp://localhost', on_connect);
```

（1）我们根据启动参数，定义了路由key变量severity，默认值为info。

(2) 我们将Exchange节点命名为direct_logs。

(3) 声明Exchange，类型为direct。

(4) 向名为direct_logs的Exchange节点推送数据，并且带上路由key变量severity。

然后我们来看看，消费者是如何绑定路由key从而消费这些数据的，保存代码为receive_logs_direct.js。

```
var amqp = require('amqplib/callback_api');

var basename = require('path').basename;

var severities = process.argv.slice(2);                                     (1)
if (severities.length < 1) {
    console.log( 'Usage %s [info] [warning] [error]',
                  basename(process.argv[1]));
    process.exit(1);
}

function bail(err, conn) {
    console.error(err);
    if (conn) conn.close(function() { process.exit(1); });
}

function on_connect(err, conn) {
    if (err !== null) return bail(err);
    process.once( 'SIGINT' , function() { conn.close(); });

    conn.createChannel(function(err, ch) {
        if (err !== null) return bail(err, conn);
        var ex = 'direct_logs' , exopts = {durable: false}; (2)

        ch.assertExchange(ex, 'direct' , exopts);                (3)
        ch.assertQueue( '', {exclusive: true}, function(err, ok)
        {                                                            (4)
            if (err !== null) return bail(err, conn);

```

```
var queue = ok.queue, i = 0; (5)

function sub(err) { (6)
    if (err !== null) return bail(err, conn);
    else if (i < severities.length) {
        ch.bindQueue(queue, ex, severities[i], {}, sub);
        i++;
    }
}

ch.consume(queue, logMessage, {noAck: true},
function(err) { (7)
    if (err !== null) return bail(err, conn);
    console.log( ' [*] Waiting for logs. To exit press
CTRL+C.' );
    sub(null);
});
});
});
}

function logMessage(msg) { (8)
    console.log( " [x] %s: '%s' ",
        msg.fields.routingKey,
        msg.content.toString());
}

amqp.connect('amqp://localhost', on_connect);
```

消费者的代码稍微有点长，不过我们对大部分代码还是比较熟悉的，所以理解起来也不难。

（1）我们从启动命令中获取一个需要绑定路由key的数组，比如我们的启动命令`node receive_logs_direct.js error info`，这样变量`severities`就保存了`["error", "info"]`。如果没有任何路由key，则程序将打印提示信息并退出。

（2）这里和生产者一样，我们定义了Exchange的名字为`direct_logs`。

(3) 声明Exchange，这里需要和生产者声明的一样。

(4) 声明队列，在队列声明成功之后，我们要定义消费函数。

(5) 获得已经声明的队列对象queue。

(6) 定义sub函数，用来绑定Exchange和队列，如果i小于severities数组的长度，则绑定队列并且完成绑定后再调用sub继续绑定队列。

(7) 定义消费函数logMessage，在定义消费函数后，执行sub()函数开始第(6)步的绑定操作。

(8) logMessage是消费队列数据函数，其操作就是打印路由key和消息内容。

同样，我们还是先启动消费者，启动一个绑定路由key为error的消费者，再启动一个绑定路由key为error、warning、info的消费者。

```
$ node receive_logs_direct.js info warning error
[*] Waiting for logs. To exit press CTRL+C

$ node receive_logs_direct.js error
[*] Waiting for logs. To exit press CTRL+C
```

然后我们分别发送error级别的消息和info级别的消息，在发送error级别的消息时，两个消费者都打印了消息，但是在发送info级别的消息时，就只有其中一个打印了消息。

```
$ node emit_log_direct.js error "Run. Run. Or it will
explode."
[x] Sent 'error': 'Run. Run. Or it will explode.'

$ node emit_log_direct.js info "Run. Run. Or it will
explode."
[x] Sent 'info': 'Run. Run. Or it will explode.'
```

消费info warning error日志的消费者打印的信息如下。

```
[*] Waiting for logs. To exit press CTRL+C.
```

```
[x] error:'Run. Run. Or it will explode.'
```

```
[x] info:'Run. Run. Or it will explode.'
```

消费error日志的消费者，只会打印error的消息了。

```
[*] Waiting for logs. To exit press CTRL+C.
```

```
[x] error:'Run. Run. Or it will explode.'
```

3.7 RabbitMQ的RPC远程过程调用

RabbitMQ还有另外一项功能，那就是提供类似RPC（Remote Procedure Call，远程过程调用）的服务。这里是建立提供RPC服务的server端代码，保存为rpc_server.js。

```
var amqp = require('amqplib/callback_api');

function fib(n) {
    var a = 0, b = 1;
    for (var i=0; i < n; i++) {
        var c = a + b;
        a = b; b = c;
    }
    return a;
}

function bail(err, conn) {
    console.error(err);
    if (conn) conn.close(function() { process.exit(1); });
}

function on_connect(err, conn) {
    if (err !== null) return bail(err);

    process.once('SIGINT', function() { conn.close(); });
```

```

var q = 'rpc_queue';                                     (2)

conn.createChannel(function(err, ch) {
  ch.assertQueue(q, {durable: false});                  (3)
  ch.prefetch(1);                                       (4)
  ch.consume(q, reply, {noAck:false}, function(err) { (5)
    if (err !== null) return bail(err, conn);
    console.log( ' [x] Awaiting RPC requests' );
  });

  function reply(msg) {                                  (6)
    var n = parseInt(msg.content.toString());           (7)
    console.log( ' [.] fib(%d)', n);
    ch.sendToQueue(msg.properties.replyTo,              (8)
      new Buffer(fib(n).toString()),
      {correlationId: msg.properties.
correlationId});
    ch.ack(msg);                                         (9)
  }
});
}

amqp.connect( 'amqp://localhost' , on_connect);

```

服务端代码较之前几节还是有明显区别的，这里的服务端代码将计算之后的结果返回给了客户端，之前都是没有返回值的。

(1) 定义了一个斐波那契数组求和的函数，主要靠这个函数来模拟耗时的计算。

(2) 定义了队列的名字为`rpc_queue`。

(3) 绑定队列`q`，就是绑定了`rpc_queue`队列。

(4) 设置公平的调度，因为RabbitMQ只会简单地调度。例如一共有2个消费者A和B，RabbitMQ只会将奇数的消息推送到消费者A去处理，偶数的消息推送到消费者B去处理，万一奇数的消息耗时非常大，则可能会出现消费者

A忙得处理不过来，而消费者B就非常空闲。这显然不符合我们负载均衡的思想，所以执行`ch.prefetch(1)`，这表示RabbitMQ不会向一个繁忙的消费者再推送超过1条的新消息，从生产者推送来的新消息将由空闲的消费者去处理。

（5）定义消费这些数据的方式，注意这里的`{noAck:false}`，我们将无响应设置为`false`，表示这条消息在处理之后将会响应给生产者。

（6）定义`reply`函数，这个函数是说明如何处理收到的消息的。

（7）将接收到的字符串数据转为整数，方便计算斐波那契数组的和。

（8）把计算的结果发送回队列，关联`id`就是之前消息的属性`correlationId`。

（9）响应队列，告知消息已经处理完毕。注意这里的`ch.ack(msg)`，如果遗漏了响应队列操作，则后果比较严重，客户端退出后，消息还是会重复推送给消费者，这样就会造成内存泄漏。我们可以通过如下命令检查是否存在此问题。

```
$ sudo rabbitmqctl list_queues name messages_ready messages_unacknowledged
```

下面是发起调用RPC的客户端代码，保存为`rpc_client.js`。

```
var amqp = require('amqplib/callback_api');
var basename = require('path').basename;
var uuid = require('node-uuid');                                     (1)

var n;
try {                                                                (2)
  if (process.argv.length < 3) throw Error('Too few args');
  n = parseInt(process.argv[2]);
}
catch (e) {
  console.error(e);
  console.warn('Usage: %s number', basename(process.
argv[1]));
```

```

    process.exit(1);
}

function bail(err, conn) {
    console.error(err);
    if (conn) conn.close(function() { process.exit(1); });
}

function on_connect(err, conn) {
    if (err !== null) return bail(err);
    conn.createChannel(function(err, ch) {
        if (err !== null) return bail(err, conn);

        var correlationId = uuid();
        function maybeAnswer(msg) {
            if (msg.properties.correlationId === correlationId) { (4)
                console.log( ' [.] Got %d' , msg.content.toString());
            }
            else return bail(new Error( 'Unexpected message' ),
conn);

            ch.close(function() { conn.close(); });
        }

        ch.assertQueue( '' , {exclusive: true}, function(err, ok)
{
            if (err !== null) return bail(err, conn);
            var queue = ok.queue;
            ch.consume(queue, maybeAnswer, {noAck:true}); (6)
            console.log( ' [x] Requesting fib(%d)' , n);
            ch.sendToQueue( 'rpc_queue' , new Buffer(n.toString()),
{
            replyTo: queue, correlationId: correlationId
            });
        });
    });
}

amqp.connect( 'amqp://localhost' , on_connect);

```

客户端代码较之前多了不少，我们来简单分析一下这些代码。

（1）加载uuid包，用来生成一个唯一的不重复的字符串。

（2）如果启动参数少于3个，或者第三个启动参数不是数字，则都会报错。第三个参数就是丢给消费者计算斐波那契数组之和的数字。

（3）获取uuid，并赋值给变量correlationId，用来做消息的关联id。

（4）定义maybeAnswer函数，用作接收消费者的计算结果的响应，如果关联id匹配，则打印结果，如果不匹配则抛出错误。

（5）声明队列操作。

（6）监听并接收这个队列的消费者回复的数据。

（7）向队列中发送数据，并传入关联id和回复队列对象。

同样，我们还是先启动消费者，等待消息的推送。

```
$ node rpc_server.js  
[*] Waiting for logs. To exit press CTRL+C
```

```
$ node rpc_server.js  
[*] Waiting for logs. To exit press CTRL+C
```

然后启动生产者，发送几个数字让消费者计算斐波那契数组之和。

```
$ node rpc_client.js 30  
[x] Requesting fib(30)  
[.] Got 832040
```

```
$ node rpc_client.js 35  
[x] Requesting fib(35)  
[.] Got 9227465
```

```
$ node rpc_client.js 40  
[x] Requesting fib(40)  
[.] Got 102334155
```


如果发现消费者计算不过来了，则可以启动多个进程来增加整个系统的性能，扩展起来非常方便，下面是两个消费者打印的信息。

第一个rpc_server打印的信息如下：

```
[x] Awaiting RPC requests
[.] fib(30)
[.] fib(40)
```

第二个rpc_server打印的信息如下：

```
[x] Awaiting RPC requests
[.] fib(35)
```

3.8 基于RabbitMQ的Node.js和Python通信实例

如今我们构建了整个互联网后端架构，跨语言通信需求非常多，比如原有的系统是用Java开发的，但是在一些非常适合Node.js发挥场景的地方又要使用Node.js来开发，而两者之间的通信方法也有多种，目前跨语言最流行和轻量级的通信方式就是用HTTP的RESTful，也可以选择性能更好的Thrift。

关于HTTP协议通信的优点和缺点，本节不做阐述，在3.9节将会对它进行详细的分析，本节主要介绍如何通过RabbitMQ这个媒介，让Node.js和Python建立起通信的桥梁。

我们还是从最简单的人手，以Node.js端作为生产者，通过RabbitMQ消息队列发送一个Hello World，然后以Python端作为消费者，打印这个Hello World字符串。

把Python作为跨语言通信实例的语言，有几方面考虑。

- Python是各个Linux流行的发行版本自带的语言，CentOs或Ubuntu都会在系统中预装Python语言，大部分是2.6.x或2.7.x版本，所以在Linux上

运行这个实例就非常简单，不需要安装其他语言环境。

- Python语言以简洁闻名，就算你没有任何Python基础，凭借其他语言的开发经验，仍然能够很轻松地读懂Python，所以没有接触过Python也没关系，看着代码大致是能够看懂流程的。
- RabbitMQ官方提供的示例，默认就是Python语言，所以拿Python作为实例更贴切不过。

我们先看生产者Node.js的代码，套用第一个例子，保存为send.js。

```
var amqp = require('amqplib/callback_api');

function bail(err, conn) {
  console.error(err);
  if (conn) conn.close(function() { process.exit(1); });
}

function on_connect(err, conn) {
  if (err !== null) return bail(err);

  var q = 'hello';
  var msg = 'Hello World!';

  function on_channel_open(err, ch) {
    if (err !== null) return bail(err, conn);
    ch.assertQueue(q, {durable: false}, function(err, ok) {
      if (err !== null) return bail(err, conn);
      ch.sendToQueue(q, new Buffer(msg));
      console.log(" [x] Sent '%s'", msg);
      ch.close(function() { conn.close(); });
    });
  }

  conn.createChannel(on_channel_open);
}

amqp.connect('amqp://localhost', on_connect);
```

接下来看看消费者Python的代码，在运行Python之前，需要安装Python的RabbitMQ连接客户端pika。我们分别执行如下命令，安装Python的Pip（和Node.js中的Npm一样，是包管理软件），然后通过Pip安装pika。

```
$ wget https://bootstrap.pypa.io/get-pip.py
$ python get-pip.py
$ pip install pika
$ pip list
iniparse (0.3.1)
**pika (0.9.14)**
pip (6.0.8)
pycurl (7.19.0)
pygpme (0.1)
setuptools (14.0)
urlgrabber (3.9.1)
yum-metadata-parser (1.1.2)
```

现在贴上Python端的代码，保存为receive.py，然后把它运行起来。

```
import pika                                                    (1)

connection = pika.BlockingConnection(pika.Connection
Parameters(                                                    (2)
    host='localhost' ))
channel = connection.channel()                                  (3)

channel.queue_declare(queue='hello' )                          (4)

print ' [*] Waiting for messages. To exit press CTRL+C'

def callback(ch, method, properties, body):                    (5)
    print " [x] Received %r" % (body,)

channel.basic_consume(callback,                                (6)
    queue='hello' ,
    no_ack=True)
```

```
channel.start_consuming() (7)
```

```
print 'never print me!' (8)
```

(1) 引入pika包，和Node.js的require功能相同。

(2) 建立连接，然后返回连接对象。

(3) 声明一个频道channel，和Node.js的用法相同。

(4) 对这个频道声明队列，对名字和Node.js声明的相同，都是hello。

(5) 定义消费的回调函数，和Node.js定义回调函数相似，只不过Python不支持像Node.js那样的匿名函数写法，需要定义一个变量。

(6) 声明消费。

(7) 开始执行消费，这里也是类似事件循环的机制，当有消息推送到达时，就会触发消费事件，执行callback函数了。

(8) 因为第7步进入了事件循环，所以第8步的打印信息永远不会被输出。

运行脚本和Node.js也一样，直接输入如下命令。

```
$ python receive.py
```

启动Node.js，向Python发送消息。

```
$ node send.js  
[x] Sent 'Hello World!'
```

这时Python端就会收到信息，然后打印这条消息的内容。

```
[*] Waiting for messages. To exit press CTRL+C  
[x] Received 'Hello World!'
```

通过这个简单的实例，我们可以扩散出很多利用RabbitMQ跨语言通

信的消息队列，比如带路由的、带消费者响应的队列，等等。总之，有了RabbitMQ，跨语言异步通信将不再是问题了。

3.9 RabbitMQ方案和HTTP方案的对比

对于整个后端系统，使用HTTP协议和JSON格式进行多进程或多服务器通信是非常常用的方式，它最突出的优点就是简单。

- 通信协议简单。双方开发者都非常熟悉HTTP协议，所以对于通信协议方面的事情几乎没有任何障碍，只需要商量和约定数据包JSON格式就可以了。
- 不依赖第三方软件或者库。现在几乎每个主流语言都会把HTTP库集成进去，所以使用HTTP协议就无须费神找第三方库，也没有因为第三方库的兼容性等导致的通信问题。
- 不错的性能。利用HTTP协议的keepalive可以免去多次重复创建和断开连接的开销，相比传统的XML数据协议，JSON包格式更小、更紧凑，数据包的大小当然直接影响到整个后端系统的通信性能。
- JSON格式和HTTP协议一样，每个热门语言也都内置了对JSON字符串互转成语言对象的库，这些库通常也都经过严格的测试和广泛的使用，这比自己商量一种新的数据格式更简单，不易出错。

但是，HTTP协议有一个缺陷，它无法控制请求的频率，比如有这样的场景通过HTTP协议就可能会有问题了。

我们现在有一个抢购活动，每小时只有前100名用户可以拿到某个诱人的奖品，这时系统就会在每个整点遭受到大量类似DDoS攻击的用户请求。我们的前端Web服务器性能非常出色，出色地完成了接收用户大量请求的任务。

因为要保证每小时仅能有100个人获得奖品，所以必须保证数据库的操作

是原子性的，同时不能有多个插入操作，否则可能会有超过100个人获奖。

这样的需求对于HTTP方案可能无法顺利完成任务，就好比千军万马争先抢后地去过独木桥。在这个过程中可能有人从独木桥上掉下去，也可能一下子好几个人都跑过了独木桥。

这样RabbitMQ消息队列就派上用场了，把千军万马排个队，这样大家依次通过这个独木桥，开销和错误就更少。

接下来我们拿数据说话，分别模拟HTTP的处理场景和RabbitMQ的场景，然后利用压力测试软件查看在各个并发和持续请求的情况下，两种处理方案的性能和稳定性。

我们先看第一种情况，即HTTP的RESTful方式来处理抢购的场景，我们规定只有前100名用户才能抢购秒杀到某一种商品。

这里数据库利用MongoDB，链接库使用Mongoose。MongoDB也是大家使用Node.js时很常用的数据库，因为它的查询语句和返回结果都是JSON格式的，对Node.js非常友好。程序设计流程如下。

（1）利用count操作获取订单集合order中的记录条数；如果count结果小于100，则执行2，否则执行3。

（2）当count结果小于100时，我们就向order集合中插入一条记录，并返回秒杀成功的信息。

（3）当count结果大于等于100时，我们就返回用户秒杀失败的信息。

注意：下面的代码并不是秒杀活动的最佳解决方案，也不是性能最优的代码，只是为了说明HTTP通信方案和RabbitMQ通信方案的区别，生产例子中秒杀服务的设计是根据实际的业务需求而架构的，并没有万能方案。

用HTTP方式设计的各个系统节点的结构如图3-5所示。

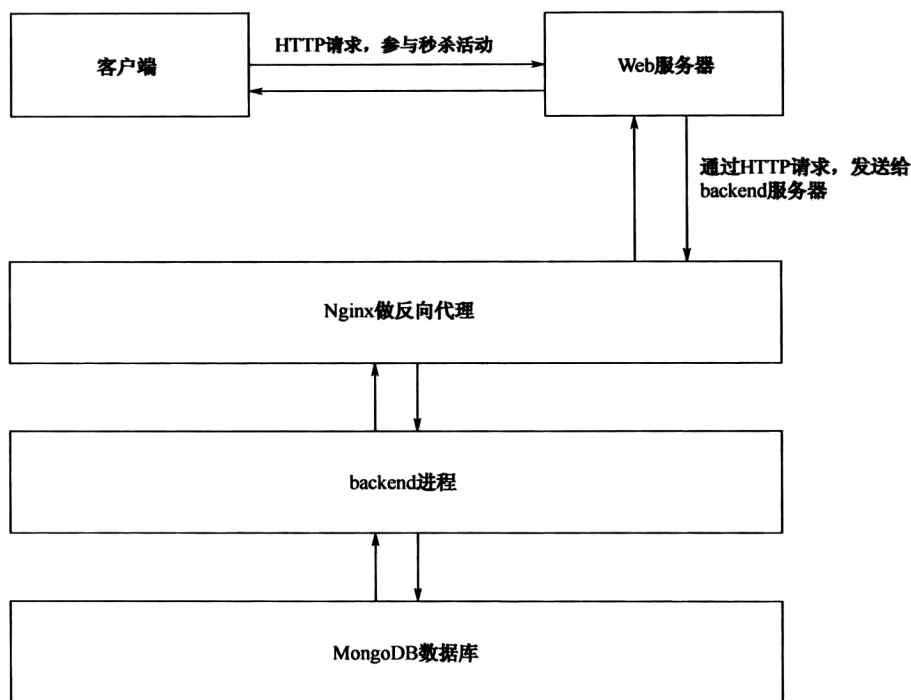


图3-5 HTTP作为后端通信的架构

我们先编写使用HTTP方式的Web服务器的代码，保存为http_web_server.js。测试时，我们使用的Express版本为4.12.2，Request版本为2.53.0，Mongoose版本为3.8.24，Node.js版本为0.10.32。

```
var express = require('express');
var request = require('request');
var util = require('util');
var app = express();
//模拟的用户Id号
var globalUserId = 1;

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

Node.js实战（第2季）

```
//定义路由
var uri = 'http://127.0.0.1:8000/buy/%d';//定义请求到后端的URL地址
var timeOut = 30*1000;//超时时间为30s

app.get('/buy/', function(req, res){
    var num = globalUserId++;
    //利用Request库发送HTTP请求
    request({
        method:'GET',
        timeout:timeOut,
        uri:util.format(uri, num)
    }, function(error, req_res, body){
        if(error){
            res.status(500).send(error)
        }
        else if(req_res.statusCode != 200){
            res.status(500).send(req_res.statusCode)
        }
        else{
            res.send(body);
        }
    });
});

app.listen(5000);
console.log('server listen on 5000');
执行如下命令，启动http_web_server.js。
$ node http_web_server.js
```

在192.168.1.110服务器上安装Nginx，Nginx的配置文件如下，我们在前面已经对Nginx作为Node.js的反向代理做过介绍，如果忘记了，那么可以翻回去看一下，这里我们重温一下关于Nginx反向代理的设置。

```
#定义启动2个Nginx进程
worker_processes 2;
events {
    use epoll;
    #设置最大连接数为2048个
    worker_connections 2048;
```



```

}
http {
    include      mime.types;
    default_type application/octet-stream;
    server_names_hash_bucket_size 64;
    access_log off;

    sendfile      on;
    keepalive_timeout 65;

    #定义后台地址, Node.js计算斐波那契数组的服务器监听3000-3002端口
    upstream backend {
        server 127.0.0.1:3000;
    }

    #反向代理配置
    server {
        listen 8000;
        location / {
            proxy_pass http://backend;
            proxy_redirect default;
            proxy_http_version 1.1;
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Connection $http_connection;
            proxy_set_header X-Forwarded-For $proxy_add_x_
forwarded_for;
            proxy_set_header Host $http_host;
        }
    }
}

```

启动好Nginx之后, 我们设计了一个公用的MongoDB的数据模型, 代码如下, 保存为orderModel.js。

```

var mongoose = require('mongoose');
//定义MongoDB连接字符串
var connstr = 'mongodb://:@127.0.0.1:27017/http_vs_rabbit';
//连接池大小
var poolsize = 50;

```

```
//建立db的连接
mongoose.connect(connstr,{server:{poolSize:poolsize}});

var Schema = mongoose.Schema;

var obj = { //定义结构
  userId:{ type:Number, required:true},
  writeTime: { type: Date, default: function(){return .
Date.now()} },    //写入时间
}
var objSchema = new Schema(obj);
//count函数
objSchema.statics.countAll = function (obj,cb) {
  return this.count(obj||{}, cb);
}
//insert函数
objSchema.statics.insertOneByObj = function (obj,cb) {
  return this.create(obj||{},cb)
}
module.exports = mongoose.model('orders', objSchema);
```

接着，开始编写判断订单数量和生成订单的代码，保存为http_backend.js。

```
var express = require('express');
//加载order订单集合的model
var orderModel = require('./orderModel.js');
var app = express();
var listenPort = 3000;

app.get('/', function(req, res){
  res.send('hello world, listenPort: '+listenPort);
});

//定义路由，执行订单操作
app.get('/buy/:userid([0-9]+)', function(req, res){
  var userid = req.params.userid;
  orderModel.countAll({}, function(err, orderCount){
    if(err) return res.status(500).send(err);
    if(orderCount >= 100){//表示已经卖完了
```

```

        return res.send('sold out!');
    }
    else{
        //说明还有库存，没卖完，这时就要向数据库插入一条带这个userid的
        订单记录了
        orderModel.insertOneByObj({
            'userId':userid,
            'writeTime':new Date()
        }, function(err, obj){
            //创建订单成功，响应成功
            if(err) return res.status(500).send(err);
            return res.send('buy success, orderid: '+obj._
            id.toString());
        });
    }
});

app.listen(listenPort);
console.log('server listen on '+listenPort);

```

通过下面的命令启动backend的Node.js服务。

```
$ node http_backend.js
```

我们可以直接访问Nginx那台服务器的8000端口，查看相应情况，检查Nginx的反向代理是否正常工作，正常情况下会出现hello world, listenPort:3000的字符串响应。

接下来我们就要对这个HTTP的系统架构进行压力测试了，这里我们使用轻量级的压力测试软件siege。下载和安装siege的命令如下，本书编写时最新的版本为3.0.9，如果下面的地址无法提供下载，则大家可以通过Google自行搜索最新版本的Siege软件下载地址。

```

$ wget http://download.joedog.org/siege/siege-latest.tar.gz
$ tar -zxvf siege-latest.tar.gz
$ cd siege-3.0.9/

```

```
$ ./configure
$ make && make install
```

Siege命令的常用参数如下。

- -c 200 指定并发数200。
- -r 5 指定测试的次数5。
- -f urls.txt 制定URL的文件。
- -i internet系统，随机发送URL。
- -b 请求无须等待 delay=0。
- -t 5 持续测试5分钟。

注：# -r和-t一般不同时使用。

现在分别模拟100个并发、300个并发和500个并发发送HTTP请求，并循环发送10次，看看在这样的压力负载情况下，系统的处理能力和数据的准确性。

```
$ siege -c 100 -r 10 -q http://192.168.1.150:5000/buy
```

如果在压力测试时出现如下错误，则执行下面的命令可以修复：

```
$ [fatal] Unable to allocate additional memory.: Cannot
allocate memory
$ ulimit -s unlimited
```

表3-1是压力测试的结果，供大家参考，成功率都是100%。trans/sec表示系统每秒处理的事物数量，longest(sec)表示最长的返回请求时间，单位是秒，orderCount表示抢购成功的用户数量。测试的3台服务器都是2CPU、4G内存的Linux x64云服务器。其中Nginx和http_backend_fib.js在一台服务器上，http_web_server.js在一台服务器上，压力测试服务器是另外一台，网络环境都是内网的1G交换机。

表3-1 压力测试结果

<div>并发数 \ 系数</div>	trans/sec	longest(sec)	orderCount
c100	97.18	3.16	101

续表

<div>并发数 \ 系数</div>	trans/sec	longest(sec)	orderCount
c300	221	3.99	101
c500	212.68	4.8	103

为什么在秒杀订单中会有超过100个的订单呢？我们先暂时把这个疑问搁置一边，开始设计如何利用RabbitMQ来处理同样的问题和同样的负荷，系统结构图如图3-6所示。

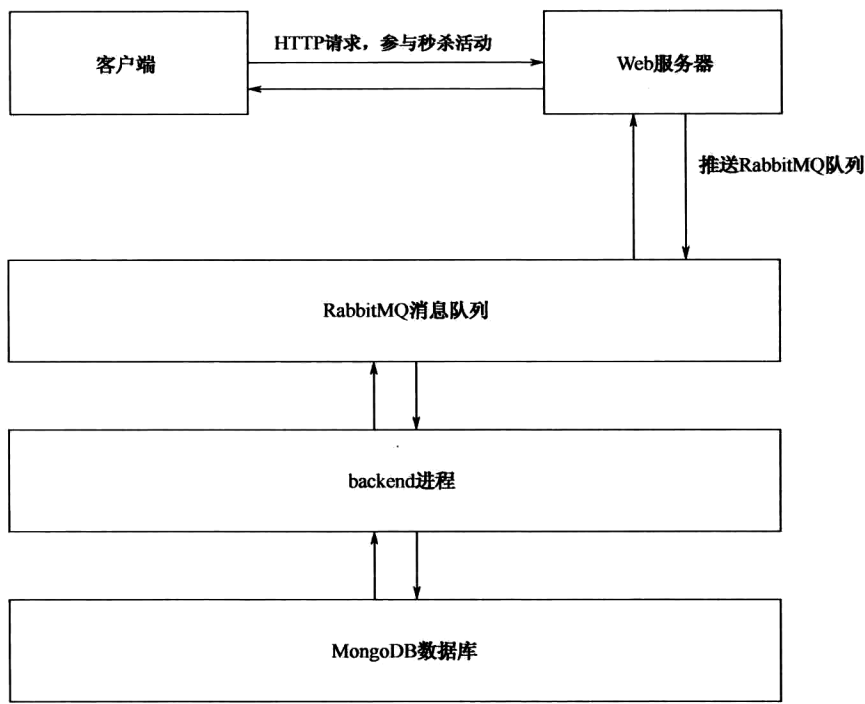


图3-6 RabbitMQ作为后端通信的架构

我们把RabbitMQ安装在backend服务器上，然后开始编写Web服务器的代码，保存为rabbit_web_server.js。

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

```
var amqp = require('amqplib/callback_api');
var uuid = require('node-uuid');

var correlationId = uuid();
var app = express();
var q = 'fibq';
var q2 = 'ackq'

var bail = function(err, conn) {
    console.error(err);
    if (conn) conn.close(function() {
        process.exit(1);
    });
}
var conn;
//模拟的用户Id号
var globalUserId = 1;

app.get('/', function(req, res){
    res.send('hello world');
});

//定义路由
app.get('/buy/', function(req, res){
    var num = globalUserId++;

    //创建channel
    conn.createChannel(function(err, ch){
        if (err !== null) return bail(err, conn);
        ch.assertQueue(q2, {durable: false}, function(err,
ok) {

            if (err !== null) return bail(err, conn);
            //定义消费函数
            ch.consume(q2, function(msg){
                //将返回值设置和HTTP方式相同
                //避免因为返回值的大小造成的测试数据偏差
                res.send(msg.content.toString());
                /*这里为了提升性能，我们不关闭链接，只关闭
```

```

channel, 链接可以重用*/
        ch.close();
    }, {noAck:true});
    //发送数据到消费者
    ch.sendToQueue(
        q,
        new Buffer(num.toString()),
        {
            replyTo:q2,
            correlationId:correlationId
        }
    );
    });
});
})

var on_connect = function(err, rabbit_conn) {
    if (err !== null) return bail(err);
    conn = rabbit_conn;
}

//建立连接
amqp.connect('amqp://127.0.0.1', {noDelay:true}, on_
connect);

app.listen(5001);
console.log('server listen on 5001');

```

执行如下命令，启动rabbitmq_server.js服务。

```
$ node rabbitmq_server.js
```

接着编写消费者的代码，同样会去加载公共orderModel数据模型文件，保存为rabbit_backend.js，代码如下：

```

var amqp = require('amqplib/callback_api');
//加载order订单集合的model
var orderModel = require('./orderModel.js');

```

```
function bail(err, conn) {
    console.error(err);
    if (conn) conn.close(function() { process.exit(1); });
}

function on_connect(err, conn) {
    if (err !== null) return bail(err);

    process.once('SIGINT', function() { conn.close(); });

    var q = 'fibq';

    conn.createChannel(function(err, ch) {
        ch.assertQueue(q, {durable: false});
        ch.prefetch(1);

        var ackSend = function(msg, content){
            ch.sendToQueue(msg.properties.replyTo,
                new Buffer(content.toString()),
                {correlationId: msg.properties.correlationId});
            ch.ack(msg);
        }

        var reply = function(msg) {
            var userid = parseInt(msg.content.toString());
            orderModel.countAll({}, function(err, orderCount){
                if(err) return ackSend(msg, err);
                if(orderCount >= 100){//表示已经卖完了
                    return ackSend(msg, 'sold out!');
                }
                else{
                    //说明还有库存，没卖完，这时就要往数据库插入一条
带这个userid的订单记录了
                    orderModel.insertOneByObj({
                        'userId':userid,
                        'writeTime':new Date()
                    }, function(err, obj){
                        //创建订单成功，响应成功

```



```

        if(err) return ackSend(msg,err);
        return ackSend(msg, 'buy success,
orderid: '+obj._id.toString());
    });
}

});
}

ch.consume(q, reply, {noAck:false}, function(err) {
    if (err !== null) return bail(err, conn);
    console.log(' [x] Awaiting RPC requests');
});
});
}

amqp.connect('amqp://127.0.0.1',on_connect);

```

输入下面的命令，启动消费者实例。

```
$ node rabbit_backend.js
```

然后用同样的压力负荷来测试利用RabbitMQ的表现。把rabbitmq_server.js放在一台服务器上，将RabbitMQ和rabbit_backend.js放在一台服务器上，压力测试单独一台服务器，结果如下。

	trans/sec	longest(sec)	orderCount
c 100	50.95	4.55	100
c 300	31.05	11.73	100
c 500	25.72	17.95	100

对比一下HTTP的压力测试结果，我们发现在每秒事务处理能力上，RabbitMQ是落后于HTTP方案的，也就是说整体系统的吞吐率RabbitMQ是不如HTTP的。好吧，笔者承认确实对RabbitMQ的测试结果有点意外，但是考虑到RabbitMQ的方案，需要先使用队列获取数据，再创建队列返回这些数

据，所以比无序的HTTP请求性能差了许多，这也是意料之中的。

但是在orderCount这一项上，RabbitMQ做到了没有偏差，HTTP方案会有所误差，这在某些特殊情况下是不被允许的，所以如果我们的业务有严格的先后顺序需求，则利用RabbitMQ消息队列是一个靠谱的选择，虽然牺牲了些性能，但是换来了业务的稳定，而且我们可以很方便地增加RabbitMQ的消费进程，来提高整体系统的吞吐率，同时，多种多样的路由规则实用性很高，加上广播、多播或单播，让我们在使用上非常灵活。

最后我们来看一下为什么HTTP方案会出现并发量大、多插入记录的问题。

首先需要明确的是，Node.js的所有I/O操作都是异步的，也就是说，两次数据库操作是不会阻塞整个进程的。

（1）在http_backend.js运行时，A请求运行到了count检查订单数量处，就向数据库发送一条count订单集合记录条数的命令。由于命令是异步的，所以Node.js进程发完这条指令后，就开始继续接收其他请求，并等待A请求指令的返回。

（2）这时B请求被Node.js接收了，然后也向数据库发送了一条count订单集合记录条数的命令，因为也是异步操作，所以Node.js此时就在等待A、B两个请求的回调函数执行。

（3）A请求的回调函数执行了，返回结果是99，表示还有1个库存，于是A请求就向数据库发出一条插入数据的指令。插入操作同样是异步的，Node.js进程继续等待回调。

（4）接着B请求的回调函数也执行了，由于A请求的插入指令是在B请求的count指令之后发出的，所以B请求的count回调函数返回的结果也是99。通过程序判断，B请求也发出了一个插入记录的指令给数据库。

（5）最后，原本99条记录的数据集合，在执行了连续A、B两个请求发出的插入指令后，最终变为了101条，我们的秒杀就多卖出了一件商品。

3.10 小结

本节介绍了Node.js如何使用成熟的消息队列方案RabbitMQ，并提供了一个简单的利用消息队列跨语言通信的例子。最后我们对比了HTTP通信方式和RabbitMQ通信方式的区别和应用场景，这可以帮助我们以后开发系统应用，考虑何时使用简单的HTTP通信，何时需要架上RabbitMQ。另外，RabbitMQ也是可以搭建集群来提供它的稳定性和处理性能的，我们可以在它的官方文档中找到搭建集群的详细方法。

RabbitMQ是目前消息队列解决方案中比较成熟、稳定的方案，虽然它的性能并不是最好的，但是可靠性已经被大家所认可，所以我们在有合适的使用消息队列的场景情况下，还是优先考虑RabbitMQ。

本节最后的一个秒杀示例并不是处理秒杀的很好方案，因为利用RabbitMQ来一条条地插入数据库，实际上是给数据库增加了一个表级别的锁，不利于性能。所以我们应该在设计秒杀、抢购这类系统时利用数据库的行级锁，这样就可以大大提高处理抢购业务时每秒的下单量。

3.11 参考文献

- <http://blog.iron.io/2012/12/top-10-uses-for-message-queue.html?spref=tw>
Top 10 Uses For A Message Queue
- <https://github.com/squaremo/rabbit.js> rabbit.js
- <http://www.ituring.com.cn/article/54547> 在Node.js 中用 Q 实现Promise – Callbacks之外的另一种选择

第4章

编写命令行工具——打造一个静态博客系统

Node.js除了可以编写服务器端程序，也可以用来编写一些命令行工具，比如比较流行的前端自动化构建工具gulp.js就是使用Node.js编写的。用Node.js编写命令行工具除了能使用NPM上十几万个各类模块资源，还具有程序启动快的优势。本章将介绍如何使用commander模块来编写一个命令行工具，并以一个静态博客系统构建工具作为实例。具体内容包含以下几部分：

- commander模块介绍；
- markdown-it模块介绍；
- 将Markdown转换成HTML；
- 实时监控文件变化；
- 给Markdown内容套用模板；
- 实时预览；
- 生成整站静态页面；
- 小结；
- 参考文献。

本文用到的第三方模块将在附录的“第三方模块”中列出，读者可根据

需要阅读该模块的详细文档。由于这些第三方模块将来可能改变其接口参数形式，如果在学习本章的过程中出现问题，则可尝试安装附录中标出的指定版本。

4.1 本章所使用到的第三方模块

1. commander

- 用途：解析命令行参数。
- 版本：2.8.1。
- 主页：<http://tj.github.io/commander.js/>。

2. Express

- 用途：Web框架。
- 版本：4.x。
- 主页：<http://expressjs.com/>。

3. serve-static

- 用途：静态文件服务中间件。
- 版本：1.9.3。
- 主页：<https://www.npmjs.com/package/serve-static>。

4. markdown-it

- 用途：渲染Markdown格式的文档。
- 版本：4.2.2。
- 主页：<https://www.npmjs.com/package/markdown-it>。

5. swig

- 用途：swig语法模板引擎。

- 版本：1.4.2。
- 主页：<http://paularmstrong.github.io/swig/>。

6. rd

- 用途：遍历目录下的所有文件，包括子目录。
- 版本：0.0.2。
- 主页：<https://www.npmjs.com/package/rd>。

7. fs-extra

- 用途：扩展了fs模块的一些方法。
- 版本：0.19.0。
- 主页：<https://www.npmjs.com/package/fs-extra>。

8. open

- 用途：使用系统程序打开指定文件或网址。
- 版本：0.0.5。
- 主页：<https://www.npmjs.com/package/open>。

9. moment

- 用途：解析、格式化日期时间。
- 版本：2.10.3。
- 主页：<http://momentjs.com/docs/>。

4.2 命令格式

在编写命令行工具时，我们首先要定义命令的使用方法，比如：

\$ myblog create表示创建一个空的博客；\$ myblog build表示生成整站静态HTML页面等。

4.2.1 常见的命令格式

一条命令一般包含以下几部分：

```
command [options] [arguments]
```

各部分的含义如下。

- **command**：命令名称，比如node。
- **options**：--单词或-单字，比如--help或-h。
- **arguments**：参数，有时选项也带参数，比如：xss。

在查看命令帮助时，会出现“[]”“<>”“|”等符号，它们的含义如下。

- []：表示是可选的。
- <>：表示可变选项，一般是多选一，而且必须要选其一。
- x|y|z：多选一，如果加上“[]”，则可不选。
- -abc：多选，如果加上“[]”，则可不选。

比如，NPM命令的使用方法描述如下：

```
Usage: npm <command>
```

其中，<command>表示可变选项，可以为list、install、config等，比如：

```
npm list
```

以上是大多数命令行工具遵循的语法格式，详细内容可参考本章末尾列出的部分参考文献。

4.2.2 定义静态博客命令格式

我们要实现的静态博客生成工具包含以下功能：

- 创建一个空的博客；

- 文章使用Markdown格式编写；
- 本地实时预览；
- 生成整站静态HTML。

根据以上描述，我们先来定义这个命令行工具的使用方法，如表4-1所示。

表4.1 命令行工具的使用方法

格式	描述
myblog create [dir]	创建一个空的博客，dir为博客所在目录（可选，默认为当前目录）
myblog preview [dir]	实时预览，dir为博客所在目录（可选，默认为当前目录）
myblog build [dir] [--output target]	生成整站静态HTML，dir为博客所在目录（可选，默认为当前目录），target为生成的静态HTML存放目录

4.3 编写命令行工具

在Node.js中，我们可以通过process.argv变量来取得当前程序启动时的参数，它是一个数组。比如，执行以下命令启动的Node.js程序：

```
$ node test.js build xxx
```

则process.argv的值为：

```
['node', 'test.js', 'build', 'xxx']
```

其中，第一个参数为Node.js的命令名，第二个参数为当前js程序的文件名，从第三个参数起则依次是启动这个Node.js程序所传进去的参数，在命令行中每个参数使用空格隔开。由于这些参数都是一个个的字符串，为了支持更灵活的参数组合方法，需要编写一个专门的程序先来解析这些参数字符串，而commander模块已经为我们做了这些。

首先创建一个空的项目文件夹，再通过npm init来初始化package.json文件：


```
$ mkdir myblog
$ cd myblog
$ npm init
```

执行npm init时，按提示填写相应的信息，或者直接按回车键即可。

接下来安装commander模块：

```
$ npm install commander --save
```

在安装完模块之后，新建文件bin/myblog，代码如下：

```
#!/usr/bin/env node

var program = require( 'commander' );

//命令版本号
program.version( '0.0.1' );

//help命令
program
  .command( 'help' )
  .description( '显示使用帮助' )
  .action(function () {
    program.outputHelp();
  });

//create命令
program
  .command( 'create [dir]' )
  .description( '创建一个空的博客' )
  .action(function (dir) {
    console.log( 'create %s' , dir);
  });

//preview命令
program
  .command( 'preview [dir]' )
  .description( '实时预览' )
```

```
.action(function (dir) {
    console.log( 'preview %s' , dir);
});

//build命令
program
    .command( 'build [dir]' )
    .description( '生成整站静态HTML')
    .option( '-o, --output <dir>' , '生成的静态HTML存放目录')
    .action(function (dir, options) {
        console.log( 'create %s, output %s' , dir, options.output);
    });

//开始解析命令
program.parse(process.argv);
```

文件的第一行`#!/usr/bin/env node`用于指定当前文件使用哪个解释器来执行。在Linux Shell环境下，文件具有执行权限时，可以直接通过`./xxxx`来执行（一般我们要执行Node.js程序时是执行命令`node xxxx.js`），如果没有指定解释器，则默认是使用`bash`来执行的。同理，使用Python编写的可执行脚本第一行是`#!/usr/bin/env python`，具体介绍可详细阅读本章末尾的参考文献。

上面的程序在使用`commander`模块解析命令时，大多是如下格式：

```
program
    .command( 'help' )
    .description( '显示使用帮助')
    .action(function () {
        program.outputHelp();
    });
```

其中代码解释如下：

- `command('help')`表示当前是什么命令；
- `description('显示使用帮助')`为当前命令的简单描述，在查看命令帮助时会显示出来；
- `action(callback)`为解析到当前命令时执行的回调函数。

在build命令中，多了下面这一行：

```
.option('-o, --output <dir>', '生成的静态HTML存放目录')
```

其表示在执行build命令时，还可以附加一些可选项，比如-o <dir>用来指定生成的文件输出到哪里。

现在编辑文件package.json，增加bin属性：

```
{
  "name": "myblog",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "MIT",
  "bin": {
    "myblog": "./bin/myblog"
  },
  "dependencies": {
    "commander": "^2.8.1"
  }
}
```

bin属性用来指定当前模块需要链接的命令，在这里我们指定了myblog命令是执行文件./bin/myblog。

为了让这个设置生效，还需要执行以下命令来进行链接：

```
$ sudo npm link
```

执行成功后，控制台大概会打印这样的内容：

```
/usr/local/bin/myblog -> /usr/local/lib/node_modules/myblog/
bin/myblog
/usr/local/lib/node_modules/myblog -> /Users/tmp/myblog
```

现在执行以下命令：

```
$ myblog help
```

若看到打印出这样的内容，则说明我们这个命令行工具的基本框架已经完成了：

```
Usage: myblog [options] [command]
```

```
Commands:
```

help	显示使用帮助
create [dir]	创建一个空的博客
preview [dir]	实时预览
build [options] [dir]	生成整站静态HTML

```
Options:
```

-h, --help	output usage information
-V, --version	output the version number

4.4 实时预览

我们现在已经实现了整个命令行工具的基本框架，但在功能上还是空白的。一个静态博客工具包含以下这些功能模块：

- 渲染文章内容页面和文章列表页面；
- 修改模板实时预览；
- 创建基本的博客模板。

在本章的例子中，我们编写出来的简单博客模板也将作为create命令时自动创建的初始文件，使用build命令生成文章页面时需要渲染文章内容和列表页面，而preview命令需要实现的功能则包含了create和build命令的功能，所以为了开发方便，我们先从实现preview命令入手。

4.4.1 启动Web服务器

为了使代码结构更加清晰，首先我们将bin/myblog文件中preview命令的回调函数改为require('../lib/cmd_preview'):

```
//preview命令
program
  .command( 'preview [dir]' )
  .description( '实时预览' )
  .action(require( '../lib/cmd_preview' ));
```

新建文件lib/cmd_preview.js，代码如下：

```
var express = require('express');
var serveStatic = require( 'serve-static' );
var path = require( 'path' );

module.exports = function (dir) {
  dir = dir || '.';

  //初始化Express
  var app = express();
  var router = express.Router();
  app.use( '/assets' , serveStatic(path.resolve(dir,
'assets' )));
  app.use(router);

  //渲染文章
  router.get( '/posts/*' , function (req, res, next) {
    res.end(req.params[0]);
  });

  //渲染列表
  router.get( '/' , function (req, res, next) {
    res.end( '文章列表' );
  });

  app.listen(3000);
};
```

在这段代码中，我们使用Express模块来启动了一个Web服务器，主要处理以下三部分内容：

- 以“/assets”开头的URL为博客中用到的静态资源文件，对应的是博客根目录下的assets目录；
- 以“/posts”开头的URL为文章内容页面，比如访问的URL是/posts/2016-06/hello-world.html，对应的是源文件_posts/2016-06/hello-world.md；
- “/”为文章列表页面。

在前文定义的命令格式中，`preview [dir]`表示`preview`命令可提供一个`dir`参数，用于指定当前博客项目所在的目录，如果没有指定则默认为当前目录，所以我们在程序中使用以下代码来处理：

```
dir = dir || '.';
```

由于使用到了Express和serve-static两个模块，所以我们还要先安装它们：

```
$ npm install express serve-static --save
```

现在我们执行以下命令来启动它：

```
$ myblog preview
```

分别在浏览器中打开<http://127.0.0.1:3000/>和<http://127.0.0.1:3000/posts/2015-05/hello-world.html>来看看效果。

4.4.2 渲染文章页面

文章内容使用Markdown语法来编写，我们可以使用markdown-it模块来解析并将其转为相应的HTML。模板引擎我们选用swig，其语法看起来比ejs更优雅，可读性会强很多。

文章源文件存储在_post目录下，比如文件_posts/2015-06/hello-world.md对应的URL是/posts/2015-06/hello-world.html。

文件lib/cmd_preview.js的代码如下：

```
var express = require('express');
var serveStatic = require('serve-static');
var path = require('path');
var fs = require('fs');
var MarkdownIt = require('markdown-it');
var md = new MarkdownIt({
  html: true,
  langPrefix: 'code-' ,
});

module.exports = function (dir) {
  dir = dir || '.';

  //初始化Express
  var app = express();
  var router = express.Router();
  app.use('/assets', serveStatic(path.resolve(dir,
'assets')));
  app.use(router);

  //渲染文章
  router.get('/posts/*', function (req, res, next) {
    var name = stripExtname(req.params[0]);
    var file = path.resolve(dir, '_posts', name + '.md');
    fs.readFile(file, function (err, content) {
      if (err) return next(err);
      var html = markdownToHTML(content.toString());
      res.end(html);
    });
  });

  //渲染列表
  router.get('/', function (req, res, next) {
    res.end('文章列表');
  });
};
```

```
app.listen(3000);

};

//去掉文件名中的扩展名
function stripExtname (name) {
  var i = 0 - path.extname(name).length;
  if (i === 0) i = name.length;
  return name.slice(0, i);
}

//将Markdown转换为HTML
function markdownToHTML (content) {
  return md.render(content || '');
}
```

再写下我们的第一篇文章，保存到`example/_posts/2015-06/hello-world.md`（为了不让博客内容与博客程序混淆在一起，我们假定博客项目在`example`目录下）：

```
# hello, world
```

这是我写下的第一篇文章

现在，安装`markdown-it`模块并启动程序（注意`preview`命令后面有指定博客项目所在的目录，为`example`）：

```
$ npm install markdown-it --save
$ myblog preview example
```

然后在浏览器中打开`http://127.0.0.1:3000/posts/2015-06/hello-world.html`，我们可以看到如图4-1所示的页面。

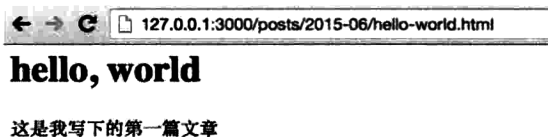


图4-1 文章页面

4.4.3 文章元数据

一篇文章除内容外，一般还会带上一些元数据，比如文章标题、发表时间、标签等。我们假设每篇文章是以下格式：

```
---
title: hello, world
date: 2015-06-16
---
```

这是我写下的第一篇文章

文件顶部在“---”之间的部分是文章的元数据，格式为名称: 值，剩下部分则为文件的内容。

修改文件lib/cmd_preview.js，在末尾增加以下函数来解析文章元数据：

```
//解析文章内容
function parseSourceContent (data) {
  var split = '---\n';
  var i = data.indexOf(split);
  var info = {};
  if (i !== -1) {
    var j = data.indexOf(split, i + split.length);
    if (j !== -1) {
      var str = data.slice(i + split.length, j).trim();
      data = data.slice(j + split.length);
      str.split('\n').forEach(function (line) {
        var i = line.indexOf(':');
        if (i !== -1) {
          var name = line.slice(0, i).trim();
          var value = line.slice(i + 1).trim();
          info[name] = value;
        }
      });
    }
  }
  info.source = data;
}
```

```
    return info;
}
```

再修改渲染文章路由处理部分，将原来的

```
var html = markdownToHTML(content.toString());
res.end(html);
```

改为以下代码：

```
var post = parseSourceContent(content.toString());
console.log(post);
var html = markdownToHTML(post.source);
res.end(html);
```

重新启动程序并刷新页面，可看到文章的标题不见了，而在控制台中可看到这样的输出：

```
{ title: 'hello, world',
  date: '2015-06-16',
  source: '\n这是我写下的第一篇文章\n' }
```

其中title和date为我们在文章源文件中设置的元数据，而source则为文章的内容。有了这些元数据，我们接下来就可以做更多的事情了。

4.4.4 增加模板

在上面的例子中，在浏览器中打开文章页面时，只显示了文章的内容，也没有各种样式，显得略丑，接下来我们给它接上一个模板。

修改文件lib/cmd_preview.js，再末尾增加以下代码：

```
var swig = require('swig');
swig.setDefaults({cache: false});

//渲染模板
function renderFile (file, data) {
    return swig.render(fs.readFileSync(file).toString(), {
```

```

    filename: file,
    autoescape: false,
    locals: data
  });
}

```

说明：为了编程方便，此处我们使用`fs.readFileSync()`这个方法读取文件内容，在读取过程中它会造成进程阻塞，但在本例中不会造成影响。

将原来渲染文章内容部分

```

var post = parseSourceContent(content.toString());
console.log(post);
var html = markdownToHTML(post.source);
res.end(html);

```

改为以下代码：

```

var post = parseSourceContent(content.toString());
post.content = markdownToHTML(post.source);
post.layout = post.layout || 'post';
var html = renderFile(path.resolve(dir, '_layout', post.layout
+ '.html'), {post: post});
res.end(html);

```

说明：

- 在渲染模板时，传递`post`变量进去，在模板中可以通过`post.content`来取得文章的内容，通过`post.xxx`来取得文章的元数据`xxx`；
- 可以通过元数据`layout`来指定要渲染的模块，默认为`post`，模板文件存储在`_layout`目录下。

新建模板文件，保存到`example/_layout/post.html`：

```

<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>{{config.title|escape}}</title>

```

```
<link rel="stylesheet" href="/assets/style.css">
</head>
<body>
  <h1>{{post.title|escape}}</h1>
  <p>日期: {{post.date|escape}}</p>
  <hr>
  <div class="post-content">{{post.content}}</div>
</body>
</html>
```

在模板中引用了一个CSS文件，新建文件`example/assets/style.css`，内容如下：

```
html {
  background-color: #EEE;
}
body {
  margin: 0 auto;
  width: 800px;
  background-color: #FFF;
  padding: 40px;
  min-height: 500px;
  line-height: 1.6;
  font-family: "Helvetica Neue", Helvetica, Arial, sans-serif;
  font-weight: 300;
}
ul {
  list-style: none;
  margin: 0;
  padding: 0;
}
li {
  margin-top: 20px;
  border-bottom: 1px solid #888;
  display: block;
}
pre {
  background-color: #F5F5F5;
  margin: 0;
  padding: 8px 12px;
```

```

font-family: "Helvetica Neue", Helvetica, Arial, sans-serif;
font-size: 14px;
font-weight: 300;
color: #000;
}

.post-date {
  background-color: #888;
  color: #FFF;
  padding: 0px 16px;
  display: inline-block;
}

.post-title {
  text-decoration: none;
  font-weight: bold;
  display: inline-block;
  margin-left: 6px;
}

.post-title:hover {
  color: #F00;
}

```

安装swig模块并启动程序:

```

$ npm install swig --save
$ myblog preview example

```

然后在浏览器中打开<http://127.0.0.1:3000/posts/2015-06/hello-world.html>, 现在的页面显得美观多了, 如图4-2所示。

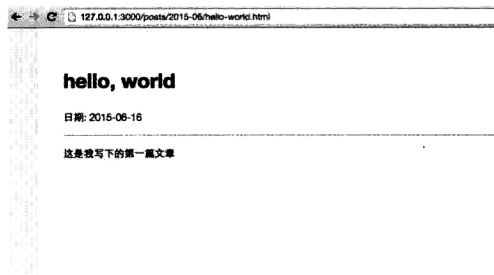


图4-2 带样式的文意页面

现在我们来换一下模板试试。

给文件`example/_posts/2015-06/hello-world.md`添加以下元数据：

```
layout: post2
```

再新建模板文件`example/_layout/post2.html`：

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>{{config.title|escape}}</title>
</head>
<body>
  <h1>{{post.title|escape}}</h1>
  <p>日期： {{post.date|escape}}</p>
  <hr>
  <div class="post-content">{{post.content}}</div>
</body>
</html>
```

说明：与`post.html`的区别是这里没有引用CSS文件。

重新运行程序并刷新页面，可看到页面中的样式已经去掉了，如图4-3所示。



图4-3 使用了其他模板的文章页面

4.4.5 渲染文章列表

要渲染文章列表，则首先要遍历所有文章，并且按照发表时间来排序，

然后将其标题渲染出来。每篇文章均存储在_posts目录下，为了方便管理，我们采用的格式是发表年月/文件名.md，比如2015年6月份发表的文章hello-world.md保存的文件名为_posts/2015-06/hello-world.md，我们可以借助rd模块来遍历整个_posts目录下的.md文件。

修改文件lib/cmd_preview.js，在顶部增加以下代码：

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

然后将原来的渲染列表部分：

```
//渲染列表
router.get('/', function (req, res, next) {
  res.end('文章列表');
});
```

改为以下代码：

```
//渲染列表
router.get('/', function (req, res, next) {
  var list = [];
  var sourceDir = path.resolve(dir, '_posts');
  rd.eachFileFilterSync(sourceDir, /\.md$/, function (f, s) {
    var source = fs.readFileSync(f).toString();
    var post = parseSourceContent(source);
    post.timestamp = new Date(post.date);
    post.url = '/posts/' + stripExtname(f.slice(sourceDir.
length + 1)) + '.html';
    list.push(post);
  });

  list.sort(function (a, b) {
    return b.timestamp - a.timestamp;
  });

  var html = renderFile(path.resolve(dir, '_layout', 'index.
html'), {
    posts: list
```

```
});  
res.end(html);  
});
```

说明：

- `rd.eachFileFilterSync(dir, pattern, callback)`用于遍历目录下的所有文件，`dir`为要遍历的目录，`pattern`为过滤规则正则表达式，在此例子中我们指定只读取.md后缀的文件，`callback`为回调函数，每读取到一个文件就会执行一次此函数，它的第一个参数为这个文件的完整路径；
- 在得到文章列表之后，我们还需要对文章按发表时间降序排序，首先我们先通过`post.timestamp = new Date(post.date);`得到文章发表时间的戳，然后通过数组的`sort()`来进行排序；
- `post.url`为文章的链接，主要为了在渲染文章列表时，点击文章的链接以打开文章的详细内容页面，通过`'/posts/' + stripExtname(f.slice(sourceDir.length + 1)) + '.html'`来取得，其原理为先取得文章源文件在`_posts`目录下的相对路径，然后将其后缀名.md改为.html即可。

新建文章列表的模板文件为`_layout/index.html`，代码如下：

```
<!doctype html>  
<html>  
<head>  
  <meta charset="utf-8">  
  <title>{{config.title|escape}}</title>  
  <link rel="stylesheet" href="/assets/style.css">  
</head>  
<body>  
  <h1>{{config.title|escape}}</h1>  
  <ul>  
    {% for post in posts %}  
      <li>
```



```
<span class="post-date">{{post.date|escape}}</span>
<a class="post-title" href="{{post.url|escape}}"
>{{post.title|escape}}</a>
</li>
{% endfor %}
</ul>
</body>
</html>
```

为了能看到渲染文章列表页面的效果，我们还需要再添加几篇文章。

新建文件`example/_posts/2015-05/book.md`，内容如下：

```
---
title: Node.js实战
date: 2015-05-15
layout: post
---
```

开始编写《Node.js实战》

新建文件`example/_posts/2015-06/myblog.md`，内容如下：

```
---
title: 打造静态博客系统
date: 2015-06-17
layout: post
---
```

在本章我们将编写一个静态博客系统生成工具。

安装`rd`模块并启动程序：

```
$ npm install rd --save
$ myblog preview example
```

再在浏览器中打开`http://127.0.0.1/`，可以看到如图4-4所示的文章列表页面：

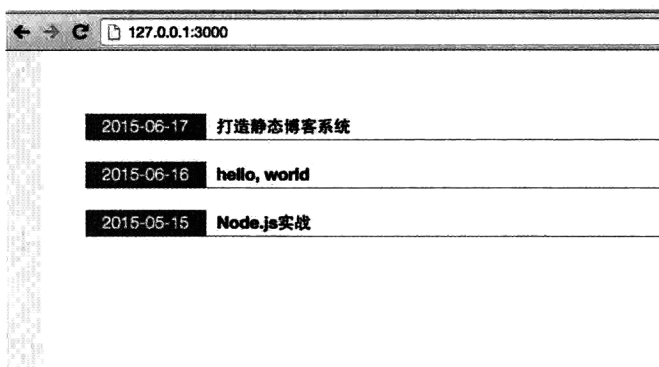


图4-4 文章列表页面1

从图4-4可以看到，首页中已经列出了我们编写的三篇文章，并且按照发表时间排序了。点击上面文章的链接可以进入“打造静态博客系统”文章的内容页面，如图4-5所示。

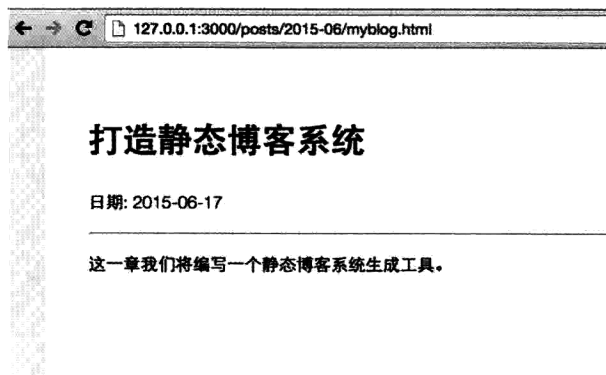


图4-5 内容页面

至此，我们这个静态博客系统的实时预览功能已经基本完成了。

4.5 生成静态博客

生成静态博客内容时渲染文章的程序与实时预览时基本一样，区别是这

一步不是等待用户访问时再渲染文章，而是直接遍历所有文章并直接渲染，然后把渲染后的页面直接保存为文件，因此我们可以先把这些公共的程序提取出来。

首先将渲染文章的内容部分和渲染文章的列表部分的程序提取处理，分别命名为`renderPost()`和`renderIndex()`，新建文件`lib/utlis.js`，代码如下：

```
var path = require('path');
var fs = require('fs');
var MarkdownIt = require('markdown-it');
var md = new MarkdownIt({
  html: true,
  langPrefix: 'code-' ,
});
var swig = require('swig');
swig.setDefaults({cache: false});
var rd = require('rd');
```

//去掉文件名中的扩展名

```
function stripExtname (name) {
  var i = 0 - path.extname(name).length;
  if (i === 0) i = name.length;
  return name.slice(0, i);
}
```

//将Markdown转换为HTML

```
function markdownToHTML (content) {
  return md.render(content || '');
}
```

//解析文章内容

```
function parseSourceContent (data) {
  var split = '---\n';
  var i = data.indexOf(split);
  var info = {};
  if (i !== -1) {
```

```
var j = data.indexOf(split, i + split.length);
if (j !== -1) {
  var str = data.slice(i + split.length, j).trim();
  data = data.slice(j + split.length);
  str.split( '\n' ).forEach(function (line) {
    var i = line.indexOf( ':' );
    if (i !== -1) {
      var name = line.slice(0, i).trim();
      var value = line.slice(i + 1).trim();
      info[name] = value;
    }
  });
}
}
info.source = data;
return info;
}

//渲染模板
function renderFile (file, data) {
  return swig.render(fs.readFileSync(file).toString(), {
    filename: file,
    autoescape: false,
    locals: data
  });
}

//遍历所有文章
function eachSourceFile (sourceDir, callback) {
  rd.eachFileFilterSync(sourceDir, /\s.md$/, callback);
}

//渲染文章
function renderPost (dir, file) {
  var content = fs.readFileSync(file).toString();
  var post = parseSourceContent(content.toString());
  post.content = markdownToHTML(post.source);
  post.layout = post.layout || 'post';
}
```

```

    var html = renderFile(path.resolve(dir, '_layout', post.
layout + '.html' ), {post: post});
    return html;
}

//渲染文章列表
function renderIndex (dir) {
    var list = [];
    var sourceDir = path.resolve(dir, '_posts' );
    eachSourceFile(sourceDir, function (f, s) {
        var source = fs.readFileSync(f).toString();
        var post = parseSourceContent(source);
        post.timestamp = new Date(post.date);
        post.url = '/posts/' + stripExtname(f.slice(sourceDir.
length + 1)) + '.html' ;
        list.push(post);
    });

    list.sort(function (a, b) {
        return b.timestamp - a.timestamp;
    });

    var html = renderFile(path.resolve(dir, '_layout', 'index.
html' ), {posts: list});
    return html;
}

//输出函数
exports.renderPost = renderPost;
exports.renderIndex = renderIndex;
exports.stripExtname = stripExtname;
exports.eachSourceFile = eachSourceFile;

```

将文件lib/cmd_preview.js改为以下代码：

```

var express = require('express');
var serveStatic = require( 'serve-static' );
var path = require( 'path' );

```

```
var utils = require( './utils' );

module.exports = function (dir) {
  dir = dir || '.';

  //初始化Express
  var app = express();
  var router = express.Router();
  app.use( '/assets' , serveStatic(path.resolve(dir,
'assets' )));
  app.use(router);

  //渲染文章
  router.get( '/posts/*' , function (req, res, next) {
    var name = utils.stripExtname(req.params[0]);
    var file = path.resolve(dir, '_posts' , name + '.md' );
    var html = utils.renderPost(dir, file);
    res.end(html);
  });

  //渲染列表
  router.get( '/' , function (req, res, next) {
    var html = utils.renderIndex(dir);
    res.end(html);
  });

  app.listen(3000);
};
```

重新启动预览程序:

```
$ myblog preview example
```

在浏览器中打开<http://127.0.0.1:3000>可以看到程序能正常运行,说明公共部分代码已经抽离出来了,接下来开始实现build命令了。

修改文件bin/myblog, 将build命令部分改成如下代码:

```
//build命令
program
  .command( 'build [dir]' )
  .description( '生成整站静态HTML' )
  .option( '-o, --output <dir>' , '生成的静态HTML存放目录' )
  .action(require( '../lib/cmd_build' ));
```

新建文件lib/cmd_build.js，代码如下：

```
var path = require('path');
var utils = require( './utils' );
var fse = require( 'fs-extra' );

module.exports = function (dir, options) {
  dir = dir || '.';
  var outputDir = path.resolve(options.output || dir);

  //写入文件
  function outputFile (file, content) {
    console.log( '生成页面: %s', file.slice(outputDir.length + 1));
    fse.outputFileSync(file, content);
  }

  //生成文章内容页面
  var sourceDir = path.resolve(dir, '_posts' );
  utils.eachSourceFile(sourceDir, function (f, s) {
    var html = utils.renderPost(dir, f);
    var relativeFile = utils.stripExtname(f.slice(sourceDir.
length + 1)) + '.html';
    var file = path.resolve(outputDir, 'posts',
relativeFile);
    outputFile(file, html);
  });

  //生成首页
  var htmlIndex = utils.renderIndex(dir);
  var fileIndex = path.resolve(outputDir, 'index.html' );
  outputFile(fileIndex, htmlIndex);
};
```

说明：

- **build**命令允许通过`--output <dir>`选项来指定文件的输出路径，如果没有指定则默认输出到当前博客项目所在的目录，通过`path.resolve(options.output || dir)`来取得，并保存到变量`outputDir`中；
- 保存文件时使用`fs-extra`模块的`fse.outputFileSync()`函数来做，其好处是可以不用管目录是否存在，如果目录不存在，则模块会自动帮我们创建。

安装`fs-extra`模块并执行`build`命令：

```
$ npm install fs-extra --save
$ myblog build example
```

程序执行后，可以看到控制台输出：

```
生成页面: posts/2015-05/book.html
生成页面: posts/2015-06/hello-world.html
生成页面: posts/2015-06/myblog.html
生成页面: index.html
```

文件均保存在`example`目录下，我们可以检查一下这些文件是否存在。

如果要将博客页面输出到别的目录，则可以尝试指定`--output`选项，比如：

```
$ myblog build example --output abc
```

程序执行完毕后，我们可以看到博客页面实际上输出到了当前的`abc`目录。

4.6 配置文件

有时我们需要在渲染页面时用到一些公共的数据，而不仅仅是文章内容中设置的元数据。另外，在启动实时预览程序时希望可以自己指定要监听的端口，这时我们可以通过博客目录下的`config.json`来指定这些数据。

修改文件lib/utils.js，增加以下代码：

```
//读取配置文件
function loadConfig (dir) {
  var content = fs.readFileSync(path.resolve(dir, 'config.
json')).toString();
  var data = JSON.parse(content);
  return data;
}

exports.loadConfig = loadConfig;
```

新建文件example/config.json，内容如下：

```
{
  "port": 3001
}
```

修改文件lib/cmd_preview.js，在顶部增加载入open模块代码：

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

将监听端口的代码：

```
app.listen(3000);
```

改为以下代码：

```
var config = utils.loadConfig(dir);
var port = config.port || 3000;
var url = 'http://127.0.0.1:' + port;
app.listen(port);
open(url);
```

说明：

- 通过配置文件的port属性来指定要监听的端口，如果没有指定则默认使用3000端口；
- open模块用于调用系统浏览器打开指定网址，在preview命令执行后，将自动在浏览器中打开博客首页。

安装open模块并执行preview命令：

```
$ npm install open --save
$ mybolg preview example
```

执行命令后，我们可以看到在浏览器中自动打开了<http://127.0.0.1:3001>。

接下来我们实现在模板中也可以通过config变量来访问到配置数据。

修改文件lib/utlis.js，将renderPost()和renderIndex()改为以下代码：

```
//渲染文章
function renderPost (dir, file) {
  var content = fs.readFileSync(file).toString();
  var post = parseSourceContent(content.toString());
  post.content = markdownToHTML(post.source);
  post.layout = post.layout || 'post';
  var config = loadConfig(dir);
  var layout = path.resolve(dir, '_layout', post.layout +
    '.html');
  var html = renderFile(layout, {
    config: config,
    post: post
  });
  return html;
}

//渲染文章列表
function renderIndex (dir) {
  var list = [];
  var sourceDir = path.resolve(dir, '_posts');
  eachSourceFile(sourceDir, function (f, s) {
    var source = fs.readFileSync(f).toString();
    var post = parseSourceContent(source);
    post.timestamp = new Date(post.date);
    post.url = '/posts/' + stripExtname(f.slice(sourceDir.
length + 1)) + '.html';
    list.push(post);
  });
}
```

```
list.sort(function (a, b) {
  return b.timestamp - a.timestamp;
});

var config = loadConfig(dir);
var layout = path.resolve(dir, '_layout', 'index.html');
var html = renderFile(layout, {
  config: config,
  posts: list
});
return html;
}
```

说明：在渲染时通过loadConfig()来读取配置信息，并作为config变量传递给模板。

修改文件example/config.json，增加title属性：

```
{
  "port": 3001,
  "title": "Node.js实战"
}
```

因为在我们的模板文件中已经有<h1>{{config.title|escape}}</h1>来引用到配置文件的title数据，所以我们现在重新执行preview，此时浏览器将显示如图4-6所示的页面。



图4-6 文章列表页面2

4.7 创建空白博客模板

在创建一个新的博客项目时，为了方便使用，我们希望这个工具能自动创建一些必需的文件，比如页面模板和默认配置文件，这样就可以马上编写文章了。

我们将在上例中创建的一些文件保存在tpl目录下：

- 将example/_layout/index.html复制到tpl/_layout/index.html；
- 将example/_layout/post.html复制到tpl/_layout/post.html；
- 将example/assets/style.css复制到tpl/assets/style.css；
- 将example/config.json复制到tpl/config.json。

一个空的博客项目包含以下目录：

- _layout目录，存放模板文件；
- _posts目录，存放文章内容源文件；
- posts目录，存放生成的博客页面；
- assets目录，存放博客页面中引用到的静态资源。

为了使程序更有趣些，我们还可以自动创建一篇hello, world文章。

修改文件bin/myblog，将create命令部分改为：

```
//create命令
program
  .command( 'create [dir]' )
  .description( '创建一个空的博客' )
  .action(require( '../lib/cmd_create' ));
```

新建文件lib/cmd_create.js，代码如下：

```
var path = require('path');
var utils = require( './utils' );
var fse = require( 'fs-extra' );
var moment = require( 'moment' );
```

```

module.exports = function (dir) {
  dir = dir || '.';

  //创建基本目录
  fse.mkdirsSync(path.resolve(dir, '_layout'));
  fse.mkdirsSync(path.resolve(dir, '_posts'));
  fse.mkdirsSync(path.resolve(dir, 'assets'));
  fse.mkdirsSync(path.resolve(dir, 'posts'));

  //复制模板文件
  var tplDir = path.resolve(__dirname, '../tpl');
  fse.copySync(tplDir, dir);

  //创建第一篇文章
  newPost(dir, 'hello, world', '这是我的第一篇文章');

  console.log('OK');
};

//创建一篇文章
function newPost (dir, title, content) {
  var data = [
    '---',
    'title: ' + title,
    'date: ' + moment().format('YYYY-MM-DD'),
    '---',
    '',
    content
  ].join('\n');
  var name = moment().format('YYYY-MM') + '/hello-world.md';
  var file = path.resolve(dir, '_posts', name);
  fse.outputFileSync(file, data);
}

```

说明:

- 使用fs-extra模块提供的mkdirsSync()来创建目录，可以不用管其父目录是否存在，模块会自动帮我们创建；

- 使用fs-extra模块提供的copySync()来复制一个目录；
- 使用moment模块来生成日期字符串。

执行以下命令安装moment模块，创建一个空的项目并预览：

```
$ npm install moment --save
$ myblog create new_blog_example
$ myblog preview new_blog_example
```

此时已经在new_blog_example目录中创建了一个空白的博客项目，并且在浏览器中打开了博客首页，如图4-7所示。

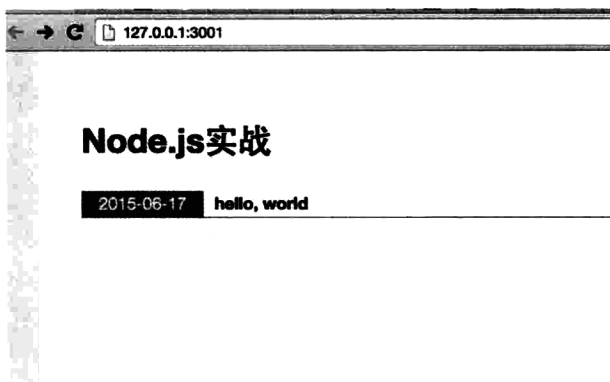


图4-7 博客首页

单击文章链接，可以看到如图4-8所示的文章页面。

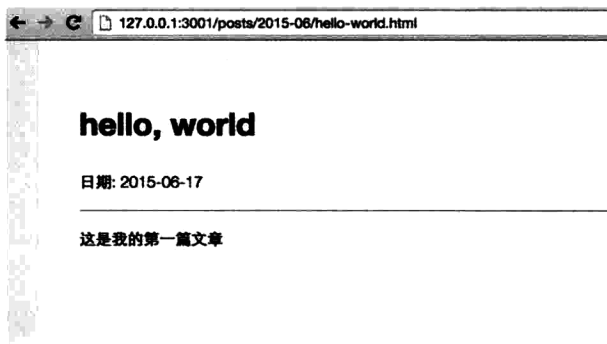


图4-8 文章页面

至此，静态博客的基本功能已经完成了。

4.8 一些有用的第三方服务

4.8.1 评论组件

由于我们的这个博客完全是静态页面，所以无法在服务器端处理用户对文章内容的评论。所幸的是，网上已有很多第三方的评论组件，我们可以在模板中简单地填写一些HTML标签和JavaScript代码，即可使用到这些服务。以下是一些常用的第三方评论组件。

- 多说 - 社会化评论系统 - 社交评论插件，详细介绍见<http://duoshuo.com/>。
- Disqus - The Web's Community of Communities，详细介绍见<https://disqus.com>。

比如我们在模板文件_layout/post.html末尾添加以下多说评论组件的代码：

```
<!-- 多说评论框 start -->
<div class="ds-thread" data-thread-key=" " data-title="
{{post.title|escape}} " data-url=" " ></div>
<!-- 多说评论框 end -->
<!-- 多说公共JS代码 start (一个网页只需插入一次) -->
<script type="text/javascript">
var duoshuoQuery = {
  short_name: "node-in-action-blog "
};
(function() {
  var ds = document.createElement( 'script' );
  ds.type = 'text/javascript';
  ds.async = true;
  ds.src = (document.location.protocol == 'https:' ?
'https:' : 'http:') + '//static.duoshuo.com/embed.js';
```

```
ds.charset = 'UTF-8';  
(document.getElementsByTagName( 'head' )[0] || document.  
getElementsByTagName( 'body' )[0]).appendChild(ds);  
})();  
</script>  
<!-- 多说公共JS代码 end -->
```

再刷新文章页面时，可以看到如图4-9所示的效果。

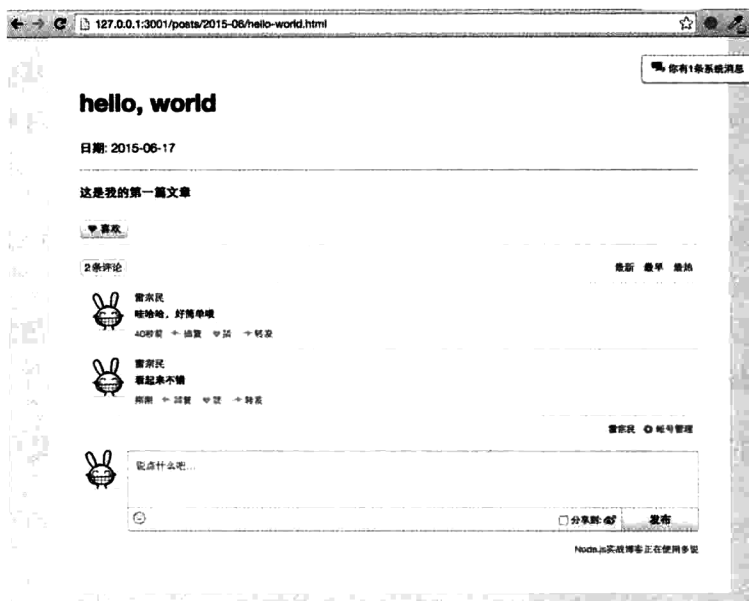


图4-9 带评论组件的文章页面

4.8.2 分享组件

分享组件主要为了方便读者将文章链接分享到微博、QQ空间等社交平台，我们可以使用国内的“加网”提供的分享组件，详细介绍见<http://www.jiathis.com/>。

比如我们在模板文件_layout/post.html末尾添加以下JiaThis分享组件的代码：


```

<!-- JiaThis Button BEGIN -->
<div class="jiathis_style_32x32">
  <a class="jiathis_button_qzone"></a>
  <a class="jiathis_button_tsina"></a>
  <a class="jiathis_button_tqq"></a>
  <a class="jiathis_button_weixin"></a>
  <a class="jiathis_button_renren"></a>
  <a href="http://www.jiathis.com/share" class="jiathis
jiathis_txt jtico jtico_jiathis" target="_blank"></a>
  <a class="jiathis_counter_style"></a>
</div>
<script type="text/javascript" src="http://v3.jiathis.com/
code/jia.js" charset="utf-8"></script>
<!-- JiaThis Button END -->

```

再刷新文章页面时，可以看到如图4-10所示的效果。



图4-10 带分享栏的文章页面

4.9 小结

本章实现了一个简单的静态博客系统程序，读者如果感兴趣，则可以在此基础上选用自己喜欢的模板引擎并增加一些有趣的功能。

由于篇幅所限，本章只简单说明了所用到的第三方模块的使用方法，

读者可以根据本章提供的第三方模块列表，自己去阅读该模块的详细使用方法。

本章示例中的代码可以通过<https://github.com/leizongmin/book-nodejs-in-action-season-2/tree/master/static-blog>获得，如果有问题，则可以在该项目主页上提交Issue。

4.10 参考文献

- 《轻触shell脚本编程》-http://happypeter.github.io/LGCB/book/11_shell_prog.html
- 《当Shell遇上了NodeJS》-<http://www.infoq.com/cn/articles/yph-shell-meet-nodejs>
- 《命令行语法格式及特殊字符》-<http://blog.sciencenet.cn/blog-548663-710438.html>
- 《解读Linux命令格式》-<http://lavasoft.blog.51cto.com/62575/533131>
- 《命令行语法》-http://publib.boulder.ibm.com/tividd/td/ITCM/SC23-4706-01/zh_CN/HTML/cmmcmst17.htm

第5章

基于Koa快速开发Web应用

本章将会讲解ES6中生成器、yield，以及Node.js下一代Web开发框架——Koa及其中间件的用法。最后，通过搭建一个简单的论坛系统，让读者从实践中学习如何基于Koa快速开发Web应用。

5.1 ES6时代的来临

ECMAScript 6，简称ES6，也称作ECMAScript 2015，于2015年6月正式定稿。ES6是ECMAScript的一次重大升级，是自2009年发布的ES5规范后的首次更新。

关于ES6的新特性，网上已经有许多关于这方面的文章，阮一峰老师的《ECMAScript 6入门》也对ES6的语法知识进行了详细介绍，这里不再赘述。本章只对所涉及的ES6中的生成器函数（GeneratorFunction）做一些简要说明。启用ES6的生成器函数特性需要io.js或者Node.js为0.11.9及以上版本并开启--harmony。

5.1.1 function和function*

大家对function想必都烂熟于心，那什么是function*呢？function*就是上面提到的生成器函数。我们来看一个例子，在Node的REPL中运行：

```
function* helloWorldGeneratorFunction() {  
  yield 'hello';  
  yield 'world';  
  return '!';  
}  
  
var helloWorldGenerator = helloWorldGeneratorFunction();  
  
> helloWorldGenerator.next()  
{ value: 'hello', done: false }  
> helloWorldGenerator.next()  
{ value: 'world', done: false }  
> helloWorldGenerator.next()  
{ value: '!', done: true }  
> helloWorldGenerator.next()  
{ value: undefined, done: true }
```

注意，有以下几点需要说明。

（1）生成器函数（GeneratorFunction）执行后会返回一个生成器（Generator）。

（2）在生成器函数内可以使用yield（或者yield*，这个在后面介绍），函数执行到每个yield时都会暂停执行并返回yield的右值（函数的上下文，如变量绑定等信息会保留），通过调用生成器的next方法返回一个包含value和done的对象，value为当前返回的值，done表明函数是否执行结束。每次调用next，函数都会从yield的下一个语句继续执行。等到整个函数执行完，next方法返回的value变为undefined，done变为true。

稍微改一下上面的例子：

```
function* helloWorldGeneratorFunction() {
```

```

    var hello = yield 'hello';
    console.log(hello);
    var world = yield 'world';
    console.log(world);
    return '!';
}

var helloWorldGenerator = helloWorldGeneratorFunction();

> helloWorldGenerator.next()
{ value: 'hello', done: false }
> helloWorldGenerator.next()
undefined
{ value: 'world', done: false }
> helloWorldGenerator.next()
undefined
{ value: '!', done: true }
> helloWorldGenerator.next()
{ value: undefined, done: true }
```

这样，在第一次调用生成器的next方法时，函数执行到yield 'hello'时会返回{ value: 'hello', done: false }，但此时yield没有返回值，或者说返回值为undefined，所以在下一次调用next方法时，相当于执行了var hello = undefined; console.log(hello); yield 'world'。因而console.log(hello);会打印 undefined，同理 console.log(world); 也会打印undefined。

再稍微修改一下上面的例子，在next方法中传入参数：

```

function* helloWorldGeneratorFunction() {
    var hello = yield 'hello';
    console.log(hello);
    var world = yield 'world';
    console.log(world);
    return '!';
}

var helloWorldGenerator = helloWorldGeneratorFunction();
```

```
> helloWorldGenerator.next(1)
{ value: 'hello', done: false }
> helloWorldGenerator.next(2)
2
{ value: 'world', done: false }
> helloWorldGenerator.next(3)
3
{ value: '!', done: true }
> helloWorldGenerator.next(4)
{ value: undefined, done: true }
```

我们往next方法里传入了参数，那么该参数就成为上一个yield语句的返回值。即调用helloWorldGenerator.next(2)相当于执行了var hello = 2; console.log(hello); yield 'world'。由于next方法的参数当作了上一个yield语句的返回值，所以第一次调用next方法时传入的参数将被忽略。

生成器函数也是函数，所以拥有所有函数的特性。比如作用域、闭包，以及遇到第一个return会执行结束等。生成器函数又有些普通函数没有的特性，比如可以使用yield并且yield只能在生成器函数内使用，如果在普通函数内使用yield将会报错。

5.1.2 yield和yield*

我们在前面了解了yield的作用，即暂停函数的执行并返回右值。那么yield*是什么呢？举几个例子进行说明：

1. Array与String

```
function* GenFunc() {
  yield [1, 2];
  yield* [3, 4];
  yield "56";
  yield* "78";
}

var gen = GenFunc();
```

```
console.log(gen.next().value); //[1, 2]
console.log(gen.next().value); //3
console.log(gen.next().value); //4
console.log(gen.next().value); // "56"
console.log(gen.next().value); //7
console.log(gen.next().value); //8
```

2. arguments

```
function* GenFunc() {
  yield arguments;
  yield* arguments;
}
var gen = GenFunc(1, 2);
console.log(gen.next().value); //{ '0' : 1, '1' : 2 }
console.log(gen.next().value); //1
console.log(gen.next().value); //2
```

3. Generator

```
function* Gen1() {
  yield 2;
  yield 3;
}
function* Gen2() {
  yield 1;
  yield* Gen1();
  yield 4;
}
var g2 = Gen2();
console.log(g2.next().value); //1
console.log(g2.next().value); //2
console.log(g2.next().value); //3
console.log(g2.next().value); //4
```

4. Object

```
function* GenFunc() {
```

```
    yield {a: '1', b: '2'};
    yield* {a: '1', b: '2'};
  }
  var gen = GenFunc();
  console.log(gen.next()); //{ value: { a: '1', b: '2' },
done: false }
  console.log(gen.next()); //TypeError: undefined is not a
function
```

从上面的例子可以看出，**yield**与**yield***的区别在于：**yield**只返回右值，而**yield***则将函数委托（**delegate**）到另一个生成器（**Generator**）或可迭代的对象（如字符串、数组、类数组**arguments**，以及ES6中的**Map**、**Set**等）。

5.1.3 co和Koa

1. co

我们已经了解了生成器函数及**yield**的用法，它们可以解决JavaScript中被诟病已久的回调金字塔问题。**co**是个典型的库，我们通过分析**co**库的核心代码来了解如何通过生成器函数及**yield**解决回调嵌套。举个例子，假如我们需要顺序读取**a.js**、**b.js**和**c.js**并打印文件内容，则通常会写出如下代码：

```
var fs = require('fs');
function readFile(path, cb) {
  fs.readFile(path, {encoding: 'utf8'}, cb);
}

readFile('a.js', function (err, dataA) {
  console.log(dataA);
  readFile('b.js', function (err, dataB) {
    console.log(dataB);
    readFile('c.js', function (err, dataC) {
      console.log(dataC);
      ...
    })
  })
})
```



```

    });
  });
});

```

如果需要读取的文件很多，那么写出的代码就像倒立的金字塔了。使用 **co** 模块后，代码如下：

```

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

function readFile(path) {
  return function (cb) {
    fs.readFile(path, {encoding: 'utf8'}, cb);
  };
}

co(function* () {
  var dataA = yield readFile('a.js');
  console.log(dataA);
  var dataB = yield readFile('b.js');
  console.log(dataB);
  var dataC = yield readFile('c.js');
  console.log(dataC);
}).catch(function (err) {
  console.log(err);
});

```

我们对 **readFile** 函数进行了改造，使它返回一个 **thunk** 函数（即有且只有一个参数是 **callback** 的函数，且 **callback** 的第一个参数为 **error**），这样我们就可以用同步的方式书写异步的代码了。我们以 **co@4** 版本的核心代码来讲解它是如何将回调金字塔扁平化的，**co@4** 的核心代码如下：

```

function co(gen) {
  var ctx = this;

  return new Promise(function(resolve, reject) {
    if (typeof gen === 'function') gen = gen.call(ctx);
    if (!gen || typeof gen.next !== 'function') return

```

```
resolve(gen);

    onFulfilled();

    function onFulfilled(res) {
        var ret;
        try {
            ret = gen.next(res);
        } catch (e) {
            return reject(e);
        }
        next(ret);
    }

    function onRejected(err) {
        var ret;
        try {
            ret = gen.throw(err);
        } catch (e) {
            return reject(e);
        }
        next(ret);
    }

    function next(ret) {
        if (ret.done) return resolve(ret.value);
        var value = toPromise.call(ctx, ret.value);
        if (value && isPromise(value)) return value;
    }
    then(onFulfilled, onRejected);
    return onRejected(new TypeError('You may only yield a
function, promise, generator, array, or object, '
+ 'but the following object was passed: "' +
String(ret.value) + '"'));
    }
    });
}
```

不难看出，co将所有yield后面的表达式都封装成了Promise对象（本身也

返回一个Promise对象），只有当前表达式执行结束后（即调用.then），才会在回调函数内执行gen.next(res)，将res赋值给yield左侧的变量并执行到下一个yield，下一个表达式执行结束后又调用gen.next()，如此循环，直至done变为true。需要特别注意的是：ES6中的yield后面可以跟任意类型的值，但co对此做了限制，只允许yield后跟thunk、promise、generator、generatorFunction、array或者object。试想一下，如果var book = yield 'Node.js实战'和var book = 'Node.js实战'的作用一样，那我们何必去用yield呢？

2. 上下文

从上面co的核心代码可以看出，co会将所有可遍历或可迭代的对象转换成Promise对象，并在当前的上下文中执行：

```
toPromise.call(ctx, ret.value);
```

这在某些情况下会导致一些意想不到的结果，如使用构造函数的情况：

```
function Person(name) {
  this.name = name;
}

Person.prototype.getName = function (callback) {
  var self = this;
  setImmediate(function () {
    callback(null, self.name);
  });
};

var person = new Person('nswbmw');

person.getName(function (err, name) {
  console.log(name);
});
```

这里通过构造函数Person生成了一个person实例，其中getName模拟了一个异步函数，调用后将会打印'nswbmw'。使用co改写如下：

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

function Person(name) {
  this.name = name;
}

Person.prototype.getName = function (callback) {
  var self = this;
  setImmediate(function () {
    callback(null, self.name);
  });
};

var person = new Person('nswbmw');

co(function*() {
  var name = yield person.getName;
  console.log(name);
}).catch(function (err) {
  console.log(err);
});
```

结果`console.log(name);`却打印`undefined`，这是因为`getName`中的`this`已经不再指向`person`，而指向了`co`的上下文。针对这种情况，有两种解决方式：一是使用`bind`，即将上例中的`yield person.getName`修改为`yield person.getName.bind(person)`；二是使用生成器函数和`yield*`，改写如下：

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

function Person(name) {
  this.name = name;
}

Person.prototype.getName = function* () {
  return this.name;
};

var person = new Person('nswbmw');
```

```

co(function*() {
  var name = yield* person.getName();
  console.log(name);
}).catch(function (err) {
  console.log(err);
});

```

有了co，我们还可以使用try...catch捕获异步函数抛出的错误，如上例可以这样写：

```

try {
  var dataA = yield readFile('a.js');
} catch(e) {
  console.log('Cannot read a.js!');
}

```

似乎，try...catch 的春天来了。

3. Koa

Koa是基于Generator和co开发的新一代中间件框架。由Express原班人马打造的Koa，致力于成为一个更小、更健壮、更富有表现力的Web框架。使用Koa编写Web应用，通过组合不同的Generator，可以免除重复烦琐的回调函数嵌套，并极大地提升常用错误处理效率。Koa不在内核方法中绑定任何中间件，它仅仅提供了一个轻量优雅的函数库，使得编写Web应用变得得心应手。Koa需要io.js或者node >= 0.11.16并开启--harmony。按照惯例，我们来看个Koa版的Hello World：

```

var koa = require('koa');
var app = koa();

app.use(function *(){
  this.body = 'Hello World';
});

app.listen(3000);

```

与Express相比，似乎并没有多少变化，无非是将res.send('hello world');

变成了`this.body = 'Hello World'`。假如我们想在每个返回请求头中都加上一个`x-response-time`的字段以表明服务器的响应时间，则在Express中实现起来比较麻烦，我们需要为`res`添加一个事件监听器，当返回的头信息有变化时，才会往头信息里面添加`x-response-time`，而这在Koa中可以轻松实现：

```
var koa = require('koa');
var app = koa();

// x-response-time

app.use(function *(next) {
  var start = new Date; //(1)
  yield next;
  var ms = new Date - start; //(3)
  this.set('X-Response-Time', ms + 'ms'); //(4)
});
app.use(function *(){
  this.body = 'Hello World'; //(2)
});

app.listen(3000);
```

引用出自Koa.rednode.cn的一段话：

“Koa中间件以一种更加传统的方式级联起来，跟你在其他系统或工具中碰到的方式非常相似。然而在以往的Node开发中，级联是通过回调实现的，想要开发对用户友好的代码是非常困难的，Koa借助Generators实现了真正的中间件架构，与Connect实现中间件的方法对比，Koa不是简单地将控制权依次移交给一个又一个的方法，直到某个结束，其执行代码的方式有点像回形针，用户请求通过中间件，遇到`yield next`关键字时，会被传递到下游中间件（downstream），在`yield next`捕获不到下一个中间件时，则逆序返回，继续执行代码（upstream）。”

所以，请求到来时上面的代码的执行顺序依次是(1)→(2)→(3)→(4)，

Koa中的`yield next`跟Express中的`next()`作用类似，都是将请求传递到下一个中间件。在Koa中，当某个中间件捕获不到下一个中间件或者遇到`return`、抛出异常时，都会终止`downstream`，即上述 (1) 和 (2)，逆序返回，继续执行`upstream`即上述 (3) 和 (4)。

图5-1可以很好地说明 Koa 的运行机制。

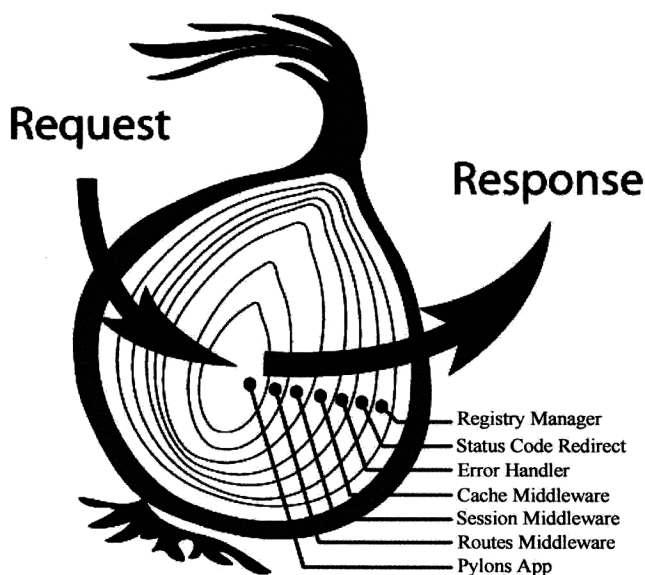


图5-1 Koa的运行机制

4. Context

Koa中有个特殊的概念——Context。Context并不是什么新鲜事物，它只是封装了`request`与`response`对象到一个对象`this`中，并提供一些快捷方法方便开发者使用。代码如下：

```
app.use(function *(){  
  this; //即Context  
  this.request; //Koa的request对象  
  this.response; //Koa的response对象  
  
  //delegate request
```

```
this.request.path === this.path //true
this.request.method === this.method //true

//delegate response
this.type = 'json';
this.length = 20;
this.response.type === 'json' //true
this.response.length === 20 //true
});
```

5.2 模板系统

5.2.1 ejs和co-ejs

ejs是常用的模板引擎之一，**co-ejs**是经过改造**ejs**源码以支持生成器函数的一种尝试。我们通常所理解的模板引擎是将定义好的模板与静态数据（如JSON）结合渲染生成HTML。而有了**co-ejs**，我们可以做些更有意思的事情，我们来看有关**co-ejs**的例子：

1. app.js

```
var path = require( 'path' );
var app = require( 'koa' )();
var wait = require( 'co-wait' );
var render = require( 'co-ejs' );

var locals = {
  version: '1.0.0',
  now: function () {
    return new Date();
  },
  ip: function *() {
    yield wait(1000);
    return this.ip;
  }
};
```



```

    },
    callback: function() {
      return function (cb) {
        cb(null, '<p>callback</p>');
      }
    },
    callbackGen: function() {
      return function* () {
        yield wait(3000);
        return '<p>callbackGen</p>';
      };
    },
    doNothing: function() {
      console.log('this will not print');
    }
  };

  var filters = {
    format: function (time) {
      return time.getFullYear() + '-' + (time.getMonth() + 1)
+ '-' + time.getDate();
    }
  };

  app.use(render(app, {
    root: path.join(__dirname, 'views'),
    layout: 'layout',
    viewExt: 'html',
    cache: true,
    debug: true,
    locals: locals,
    filters: filters
  }));

  app.use(function *() {
    console.time('time');
    yield this.render('content', {

```

Node.js实战（第2季）

```
      users: [{name: 'John' }, {name: 'Jack' }, {name:
'Tom' }]
    });
    console.timeEnd( 'time' );
  });

  app.listen(3000, function () {
    console.log( 'listening on 3000.' );
  });
  content.html

<div>
<p>version: <%= version %></p>
<p>now: <%=: now() | format %></p>
<p>ip: <%= ip %></p>
<% var cb = yield callback() %>
<p>callbackGen: <%= callbackGen() %></p>
<p>callback: <%- cb %></p>
<p><%=: users | map : 'name' | join %></p>
<% include user.ejs %>
</div>
```

可以看出，在co-ejs中，我们不仅可以传入一个普通变量、函数，甚至可以直接在模板中使用生成器函数、生成器及yield。co-ejs将会以<%= %>或<%- %>包裹的表达式并行执行，即ip函数延时1s执行结束，callbackGen函数延时3s结束，最终渲染模板只需3s。而在<% %>内使用yield则会顺序执行，假如将上例中的callback函数修改为：

```
callback: function() {
  return function (cb) {
    setTimeout(function () {
      cb(null, '<h1>callback</h1>');
    }, 2000);
  };
}
```

则渲染模板需要花5s。

5.2.2 过滤器

ejs中的过滤器由来已久，在以“<%=:”或“<%-:”开头的标签内使用管道符“|”分隔每个过滤器，过滤器的参数为管道符左侧的值，返回值为过滤或者格式化后的结果。如上例中的“<%=: now() | format %>”将当前的时间格式化成为特定的格式，然后在页面中显示。

5.3 路由

在使用Express开发时，比较好的方式就是将路由写到一个文件中，如router.js：

```
app.post('/signin', ...); //登录
app.post('/signup', ...); //注册
app.get('/signout', ...); //登出

app.get('/users', ...); //获取所有用户信息
app.get('/user/:id', ...); //获取单个用户信息
app.patch('/user/:id', ...); //修改单个用户信息

app.get('/topic/:id', ...); //获取单个话题
app.post('/topic/create', ...); //发表话题
app.delete('/topic/:id', ...); //删除某个话题
.....
```

然后将这些路由对应的controller分别放到不同的文件里，我们将这称为手工映射。手工映射十分方便，也十分灵活，但假如路由过多的话会导致定位路径及其控制器文件变得困难。而koa-frouter则尝试实现了自然映射的一种路由设计思路：文件路径即路由。我们来看koa-frouter是如何实现的，假如我们的工程目录结构如图5-2所示，其中router文件夹存放了一系列路由文件。

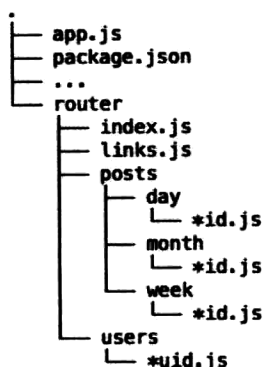


图5-2 工程目录结构

这些路由文件的代码如下。

1. *uid.js

```
exports.post = function* (uid) { ... }
```

2. *id.js

```
exports.get = function* (id) { ... }
```

3. index.js

```
exports.get = function* () { ... }
links.js
exports.get = function* () { ... }
```

4. app.js

```
var koa = require( 'koa' );
var router = require( 'koa-frouter' );

var app = koa();
app.use(router(app, '/router' ));
app.listen(3000);
```

koa-frouter只是做了些简单的工作，即遍历指定目录下所有文件，将所有以 ***** 开头的替换为：并忽略后缀名（文件名不能包含冒号），如**week**目

录下`*id.js`的路径为`/posts/week/*id.js`最终会转换为`/posts/week/:id`，同时`/posts/week/*id.js`文件导出了`get`、`post`等方法对应不同的`method`请求，通过`require`引入并通过`app.use`挂载。相当于以下代码：

```
var koa = require('koa');
var _ = require('koa-route');

var users = require('./router/users/*uid.js');
var month = require('./router/posts/month/*id.js');
var week = require('./router/posts/week/*id.js');
var day = require('./router/posts/day/*id.js');
var links = require('./router/links.js');
var index = require('./router/index.js');

var app = koa();

app.use(_.post('/users/:uid', users.post));
app.use(_.get('/posts/month/:id', month.get));
app.use(_.get('/posts/week/:id', week.get));
app.use(_.get('/posts/day/:id', day.get));
app.use(_.get('/links', links.get));
app.use(_.get('/', index.get));
app.use(_.get('/index', index.get));

app.listen(3000);
```

其中，`koa-route`是Koa的一个简单的路由中间件。相比之下，文件路径即路由的使用方式更加直观、简捷，也更容易维护，修改路由只需修改或移动文件。

5.4 参数验证与错误处理

5.4.1 koa-scheme

不管以什么语言编写程序，我们都会写大量的代码来做参数验证及错误处理。Node.js更甚，JavaScript的单线程特性意味着一个简单的语法错误都

会导致整个程序崩溃。毫不客气地说，在编写Node.js应用时，有三分之一的代码都是在验证参数及处理错误，而这些代码通常与业务逻辑代码杂糅在一起，随着应用越写越大，代码变得臃肿、难以维护。中间件的设计哲学则让参数验证及错误处理变得极为简单与优雅，我们只需将相关代码编写成中间件，并在路由之前加载，每个请求到来时都会找到对应的规则，如果验证失败则直接返回错误，验证成功后才会将请求传递到下一个中间件，这样就实现了验证逻辑与业务逻辑的分离。koa-scheme是Koa的一个参数验证中间件，同时它还可以对请求做一些过滤和格式化的工作。我们来看如下的例子。

1. app.js

```
var koa = require( 'koa' );
var bodyParser = require( 'koa-bodyparser' );
var scheme = require( 'koa-scheme' );

var app = koa();
app.use(bodyParser());
app.use(scheme(__dirname + '/scheme', {debug: true}));
app.use(function* () {
  if (this.path === '/signup') {
    console.log(this.request.body.password);
    this.body = {
      user: {
        id: this.request.body.password
      }
    };
  }
});
app.listen(3000);
```

2. scheme.js

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

module.exports = {
```

```

" /(.*)" : {
  "request" : {
    "header.version" : /^[0-9]+$/
  }
},
" (POST|put) /signup" : {
  "request" : {
    "body" : {
      "name" : /^[a-zA-Z]+$/,
      "age" : validator.isNumeric,
      "email" : validator.isEmail,
      "gender" : /^(male|female)$/ ,
      "password" : /^[a-zA-Z0-9]{6,12}$/ ,
      "repassword" : checkRepassword
    }
  },
  "response" : {
    "body.user.id" : /^[a-zA-Z0-9]{32}$/
  }
}
};

function md5 (str) {
  return crypto.createHash('md5').update(str).digest('hex');
}

function checkRepassword(repassword) {
  var body = this.request.body;
  if (repassword !== body.password) {
    return this.throw(400, '两次密码不一致!');
  }
  body.password = md5(body.password);
  return true;
}

```

建议读者亲自运行以上代码调试一下。从scheme.js中可以看出，我们对所有接收的请求都会检查头信息中的version字段，假如没有该字段或该字段

不是数字类型，则都会返回一个错误。对于post或put到/signup 路径的请求，既会检查头信息中的version字段，又会分别检查请求体和响应体中的字段。需要说明的有两点：我们在checkRepassword中先检查了两次输入的密码是否一致，若不一致则返回错误，若一致则将密码md5加密，这样我们在app.js中打印的就是加密后的密码了；假如我们只想验证嵌套比较深的某个字段而不是所有的字段，则只需像body.user.id那样使用“.” 拼接即可。

我们可以把所有的参数验证及格式化的工作都放到scheme.js中，如果需要验证的路径及规则过多，则可以按路径拆分成不同的小文件引入到scheme.js，我们唯一需要保证的就是后续的中间件收到的数据格式都是正确的，这样我们就不必在业务逻辑中编写大量重复的烦琐的参数验证代码了。

5.4.2 koa-errorhandler

在使用Node.js开发时，我们经常会写这样的代码：

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

...
readFile('a.js', {encoding:'utf8'}, function (err, dataA) {
  if (err) {
    return callback(err);
  }
  console.log(dataA);
  readFile('b.js', {encoding:'utf8'}, function (err, dataB) {
    if (err) {
      return callback(err);
    }
    console.log(dataB);
    readFile('c.js', {encoding:'utf8'}, function (err, dataC) {
      if (err) {
        return callback(err);
      }
      console.log(dataC);
    });
  });
});
```



```

    ...
    });
  });
});
...

```

每个函数执行完毕都会检查是否有错误，有错误则callback返回，没有错误则继续往下执行。使用async模块后的代码如下：

```

var async = require('async');
var readFile = require('fs').readFile;

...
async.series([
  function (done) {
    readFile('a.js', {encoding:'utf8'}, function (err, dataA) {
      console.log(dataA);
      done(err);
    });
  },
  function (done) {
    readFile('b.js', {encoding:'utf8'}, function (err, dataB) {
      console.log(dataB);
      done(err);
    });
  },
  function (done) {
    readFile('c.js', {encoding:'utf8'}, function (err, dataC) {
      console.log(dataC);
      done(err);
    });
  }
], callback);
...

```

使用Promise（以fs-promise模块为例）的代码如下：

```

var readFile = require('fs-promise').readFile;

```

```
readFile('a.js', {encoding:'utf8'})
  .then(function (dataA) {
    console.log(dataA);
    return readFile('b.js', {encoding:'utf8'})
  })
  .then(function (dataB) {
    console.log(dataB);
    return readFile('c.js', {encoding:'utf8'})
  })
  .then(function (dataC) {
    console.log(dataC);
  })
  .catch(function (err) {
    console.log(err);
  });
```

使用co模块（以co-fs模块为例）的代码如下：

```
var readFile = require('co-fs').readFile;
var co = require('co');

co(function* () {
  var dataA = yield readFile('a.js', {encoding:'utf8'});
  console.log(dataA);
  var dataB = yield readFile('b.js', {encoding:'utf8'});
  console.log(dataB);
  var dataC = yield readFile('c.js', {encoding:'utf8'});
  console.log(dataC);
}).catch(function (err) {
  console.log(err);
});
```

可以看出，使用生成器函数+yield不仅解决了回调嵌套的问题，还使代码变得简洁、易读，使错误处理变得更为优雅，任意一个文件读取失败时抛出的异常都会被co的catch捕获。

Koa是基于co开发的，所以可以在Koa中使用co支持的所有特性。因为每个中间件都是一个生成器函数，中间件又是通过next层层传递的，所以我们通

常将错误处理放到第一个中间件中，代码如下：

```
var app = koa();

app.use(function* (next) {
  try {
    yield next; // 此处next代表下游所有中间件
  } catch (err) {
    console.log(err);
  }
});

app.use(logger());
app.use(bodyParser());
...
```

这种方式能够捕获大多数错误，但捕获不了没有使用yield的异步函数的错误和事件错误等。**koa-errorhandler**是Koa的一个错误处理中间件，使用如下：

```
var koa = require('koa');
var errorHandler = require('koa-errorhandler');

var app = koa();
app.use(errorHandler());
app.use(function* () {
  foo(); // 将会被 errorHandler 捕获 foo is not defined
});

app.listen(3000, function () {
  console.log('listening on port 3000.');
```

koa-errorhandler能够根据请求头中Accept的不同返回不同类型的错误响应（html/json/text），用户也可自定义错误处理器，详见**koa-errorhandler**的readme。

5.5 缓存和配置

5.5.1 koa-router-cache和co-cache

当一个网站的访问量越来越大后，增加缓存是提升性能的一个既简单又有效的方式。增加缓存的方式大同小异，无非是将缓存层放到业务逻辑层之前，当请求到达时，首先经过缓存层，如果命中缓存则直接返回，如果没有命中则传递到业务逻辑层。缓存的更新机制也大同小异，通常有两种方式：一种是设置一个定时器定时更新缓存，一种是当业务逻辑层执行结束后更新缓存。第一种方式简单却浪费资源，第二种方式代码耦合严重也不优雅。而基于Koa的中间件特性，我们可以写出既简单又优雅也不耦合的缓存中间件。以koa-router-cache为例，我们看看它是如何实现的：

```
var app = require('koa')();
var cache = require('koa-router-cache');

var count = 0;

app.use(cache(app, {
  '/': 5 * 1000
}));

app.use(function* () {
  this.body = count++;
  if (count === 5) {
    count = 0;
    this.app.emit(this.url);
  }
});

app.listen(3000, function () {
  console.log('listening on 3000.');
```

建议读者亲自运行以上代码调试一下，以上代码的意思是：缓存主页并5s更新一次，第一次请求到来时缓存中间件中没有内容，所以传递到下一个

中间件，此时将`this.body`赋值为0，`count`变为1，当中间件的`downstream`执行完毕后执行`upstream`，此时将`this.body = 0`缓存到内存中，并设置5s的生存期，所以，后续5s内的所有请求都会因命中缓存而返回0。5s过后，因为缓存中的内容已经过期被删除，所以下个请求到来时没有命中缓存，此时传递到下一个中间件，将`this.body`赋值为1，`count`变为2，并更新缓存。直到`count`变为5时，`count`被重置为0，并通过事件触发该路径对应的监听器立即删除缓存中的老数据，这样保证了缓存中的数据都是最新的。

用户还可通过传入`getter`和`setter`参数手动管理缓存的读取和写入。`koa-router-cache`适用于无状态的页面缓存或者api服务器，不适用于页面会根据用户的登录状态的不同而渲染不同的情况，而且`koa-router-cache`是针对请求路径的缓存实现，粒度较大，我们可以尝试给每一个数据库读取函数加一层`cache`，如`co-cache`：

```
var cache = require('co-cache');

var getTopicsByTab = function* (tab, p) {
  var query = {};
  if (tab) { query.tab = tab; }
  p = p || 1;
  return yield Topic.find(query).skip((p - 1) * 10).limit(10).
exec();
};

getTopicsByTab = cache(getTopicsByTab, 10000);

co(function* () {
  var topics = yield getTopicsByTab('question', 2);
  ...
}).catch(onerror);
```

`getTopicsByTab`是一个通过标签（`Tab`）和页码（`p`）去数据库中查询话题（`topic`）的函数。每次调用`getTopicsByTab`都会查询数据库，用`co-cache`包裹并加上过期时间（10s）后，10s内每次调用该函数都是返回缓存中的结果，10s后再次调用会执行一次数据库查询操作并缓存结果。`co-cache`的适用场景

也十分有限，通常用来包裹诸如数据库查询的函数。

5.5.2 config-lite

不管是小项目还是大项目，将配置与代码分离是一种非常好的做法。我们通常将配置写到一个配置文件里，如`config.js/config.json`，并放到项目的根目录下。但项目也会分测试环境和线上环境等部署，不同环境的配置文件不同，我们不可能每次部署时都去修改引用`config.test.js`或者`config.production.js`。`config-lite`模块正是你所需要的，`config-lite`模块会根据环境变量（`NODE_ENV`）的不同加载不同的配置文件而无须修改任何代码。默认加载当前执行进程所在目录下的`config`目录，如果程序以`NODE_ENV=test node app`启动，则通过`require('config-lite')`会依次降级查找`config/test.js`、`config/test.json`、`config/test.node`并加载；如果程序以`NODE_ENV=production node app`启动，则通过`require('config-lite')`会依次降级查找`config/production.js`、`config/production.json`、`config/production.node`并加载。

5.6 测试

测试是软件开发过程中必不可少的一环，没有经过测试的程序是不完整的，也是没有保证的。手工测试既烦琐又容易出现人为失误，所以自动化测试变得越来越流行。在项目开发过程中，如何保证添加新功能后，之前的功能也都是可用的呢？如果没有写测试用例，那么需要从头挨个测试功能点；如果之前为每个功能都编写了测试，那么只需运行一遍测试程序。

5.6.1 单元测试

单元测试是自动化测试中的一种，是针对程序模块（软件设计的最小单位）来进行正确性检验的测试工作。程序单元是应用的最小的可测试部件。

在过程化编程中，一个单元就是单个程序、函数、过程等；对于面向对象编程，最小单元就是方法，包括基类（超类）、抽象类、或者派生类（子类）中的方法。

单元测试又有许多风格，常见的两大风格有：测试驱动开发（TDD）和行为驱动开发（BDD），二者最主要的区别在于：TDD关注所有功能是否被正确实现，每个功能都具备对应的测试用例；BDD强调的是系统最终的实现与用户期望的行为是否一致，验证代码实现是否符合设计目标。

5.6.2 co-mocha和co-supertest

1. mocha和co-mocha

mocha是一个流行的 Node.js测试框架，它同时支持TDD、BDD和exports风格的测试，并且支持许多优秀的断言库，支持异步和同步的测试，支持多种方式导出结果，同时支持浏览器端的测试。

mocha的BDD支持 describe()、context()、it()、before()、after()、beforeEach()、afterEach()；TDD支持suite()、test()、suiteSetup()、suiteTeardown()、setup() 和 teardown()，具体使用方法见mocha 官方网站。

我们常用的有describe()和it()，解释如下。

- describe：用来描述一组测试的目的或功能，可以嵌套。
- it：用来描述测试的期望值，一个describe可以有多个it，一个it至少有一个断言。

mocha的官方BDD示例如下：

```
$ npm install -g mocha
$ mkdir test
$ vim test/test.js

var assert = require("assert")
```

Node.js实战 (第2季)

```
describe('Array', function(){
  describe('#indexOf()', function(){
    it('should return -1 when the value is not present',
function(){
    assert.equal(-1, [1,2,3].indexOf(5));
    assert.equal(-1, [1,2,3].indexOf(0));

    })
  })
})
```

\$ mocha

```
Array
  #indexOf()
    ✓ should return -1 when the value is not present

1 passing (12ms)
```

这里使用了Node.js自带的断言库Assert，npm上还有些流行的断言库如should.js、chai和expect.js，我们使用should.js改写如下：

```
var should = require("should")
describe('Array', function(){
  describe('#indexOf()', function(){
    it('should return -1 when the value is not present', function(){
      [1,2,3].should.not.containDeep(5);
      [1,2,3].should.not.containDeep(0);

    })
  })
})
```

\$ mocha

```
Array
  #indexOf()
    ✓ should return -1 when the value is not present

1 passing (12ms)
```


可以看出，使用should.js可以写出十分语意化的测试代码。

co-mocha是mocha经过co库包装后以支持生成器函数的一个库。使用方式跟mocha类似：

```
var should = require("should")
describe('User', function(){
  describe('.addUser', function(){
    it('should return user object contains `name`, function* (){
      var user = yield User.addUser({name: 'nswbmw'});
      should(user).have.property('name', 'nswbmw');
    })
  })
})
```

2. supertest和co-supertest

supertest是一个HTTP断言库，使用它可以测试HTTP服务，如使用Express、Koa搭建的服务等。结合Express使用如下：

```
var request = require('supertest')
    , express = require('express');

var app = express();

app.get('/user', function(req, res){
  res.status(200).send({ name: 'tobi' });
});

describe('express', function () {
  it('GET /user', function (done) {
    request(app)
      .get('/user')
      .expect('Content-Type', /json/)
      .expect('Content-Length', '15')
      .expect(200, done);
  });
});
```

```
});  
  
$ mocha test.js  
  
express  
  ✓ GET /user (50ms)  
  
1 passing (60ms)
```

co-supertest是**supertest**经过**co**库包装后以支持生成器函数的一个库。使用方法也跟**supertest**类似，结合**co-mocha**使用如下：

```
var request = require('co-supertest')  
  , koa = require('koa');  
  
var app = koa();  
  
app.use(function* () {  
  this.body = { name: 'tobi' };  
});  
  
describe('koa', function () {  
  it('GET /user', function* () {  
    yield request(app.callback())  
      .get('/user')  
      .expect('Content-Type', /json/)   
      .expect('Content-Length', '15')  
      .expect(200)  
      .end();  
  });  
});  
  
$ mocha --require co-mocha test.js  
  
koa  
  ✓ GET /user (46ms)  
  
1 passing (58ms)
```

5.7 开发一个论坛系统

5.7.1 基础项目搭建

前面铺垫了那么多，现在我们将利用介绍的内容和其他Koa中间件开发一个简单的论坛系统。Nodeclub是一个优秀的开源论坛系统，已经在Node.js中文技术社区CNode(<http://cnodejs.org>)得到应用，我们仿照并借鉴Nodeclub从头搭建一个论坛系统。首先，构建基础项目结构如图5-3所示。

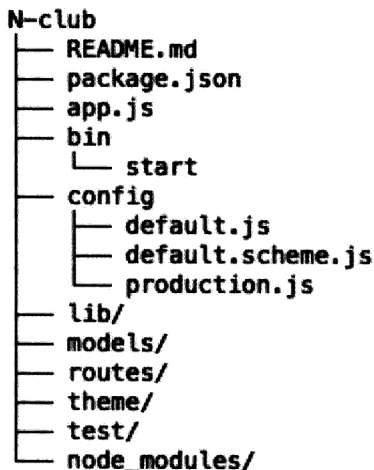


图5-3 基础项目结构

其中，需要进行如下说明。

- bin/start：启动脚本。
- config/：存放配置文件的目录，default.scheme.js 为 koa-scheme所用。
- lib/：存放一般代码文件的目录。
- models/：存放模型文件的目录。
- routes/：存放路由文件的目录。
- theme/：存放主题模板文件的目录。
- test/：存放测试文件的目录。

修改package.json，添加如下内容：

```
{
  "name": "N-club",
  "version": "0.0.1",
  "description": "N-club for koa.",
  "scripts": {
    "start": "NODE_ENV=default DEBUG=* node --harmony app"
  },
  "dependencies": {
    "co-cache": "0.0.5",
    "co-ejs": "1.5.2",
    "config-lite": "0.2.0",
    "koa": "0.18.1",
    "koa-bodyparser": "1.5.0",
    "koa-errorhandler": "0.1.1",
    "koa-flash": "0.1.0",
    "koa-frouter": "0.3.2",
    "koa-generic-session": "1.8.0",
    "koa-generic-session-mongo": "0.1.1",
    "koa-gzip": "0.1.0",
    "koa-logger": "1.2.2",
    "koa-router-cache": "0.3.0",
    "koa-scheme": "1.4.2",
    "koa-static-cache": "3.1.0",
    "merge-descriptors": "1.0.0",
    "mongoose": "4.0.0",
    "validator": "3.35.0"
  },
  "engines": {
    "node": "0.12.2"
  }
}
```

保存并运行npm install，对其中部分代码的解释如下。

- koa-bodyparser：为请求体解析中间件，相当于Express中的body-parser。

- koa-flash: 相当于connect-flash。
- koa-generic-session: 通用的session中间件, 可结合MongoDB、Redis等使用。
- koa-generic-session-mongo: 结合koa-generic-session, 将session存储到MongoDB的中间件。
- koa-static-cache: 静态文件缓存中间件。
- merge-descriptors: 合并两个对象的工具模块。
- mongoose: MongoDB驱动模块。
- validator: 参数验证工具模块。

修改 app.js, 添加如下代码:

```
var app = require('koa')();
var logger = require('koa-logger');
var bodyparser = require('koa-bodyparser');
var staticCache = require('koa-static-cache');
var errorHandler = require('koa-errorhandler');
var session = require('koa-generic-session');
var MongoStore = require('koa-generic-session-mongo');
var flash = require('koa-flash');
var gzip = require('koa-gzip');
var scheme = require('koa-scheme');
var router = require('koa-router');
var routerCache = require('koa-router-cache');
var render = require('co-ejs');
var config = require('config-lite');

//不放到default.js是为了避免循环依赖
var merge = require('merge-descriptors');
var core = require('./lib/core');
var renderConf = require(config.renderConf);
merge(renderConf.locals || {}, core, false);

app.keys = [renderConf.locals.$app.name];

app.use(errorHandler());
```

```
app.use(bodyParser());
app.use(staticCache(config.staticCacheConf));
app.use(logger());
app.use(session({
  store: new MongoStore(config.mongodb)
}));
app.use(flash());
app.use(scheme(config.schemeConf));
app.use(routerCache(app, config.routerCacheConf));
app.use(gzip());
app.use(render(app, renderConf));
app.use(router(app, config.routerConf));

app.listen(config.port, function () {
  console.log('Server listening on: ', config.port);
});
```

中间件的加载顺序十分重要，例如上面的`errorhandler`中间件需要放到最上面，才能捕获下游抛出的错误。`Flash`中间件需要放到`session`中间件之后，因为`Flash`功能是基于`session`实现的。`routerCache`需要放到`router`前面，而`scheme`需要放到`routerCache`前面。`gzip`压缩中间件需要放到`routerCache`之后，这样 `routerCache`缓存的就是`gzip`压缩后的内容了，大大减少了内存消耗量。

修改`default.js`，添加如下代码：

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

module.exports = {
  port: process.env.PORT || 3000,
  mongodb: {
    url: 'mongodb://127.0.0.1:27017/club'
  },
  schemeConf: path.join(__dirname, '../default.scheme'),
  staticCacheConf: path.join(__dirname, '../theme/publices'),
  renderConf: path.join(__dirname, '../theme/config'),
  routerConf: 'routes',
  routerCacheConf: {
    '/': {
```

```

    expire: 10 * 1000,
    condition: function() {
        return !this.session || !this.session.user;
    }
}
}
};

```

我们尽量把在app.js中使用的配置信息放到了配置文件里，其中 ./lib/core.js是暴露出来的核心文件，将它与模板中自定义的locals合并作为co-ejs渲染时的本地变量，模板中还自定义了一个\$app变量，保存了模板的主题信息。我们规定模板目录下的publices目录用来存放静态文件，config.js保存了co-ejs的配置。我们还针对未登录的用户对主页进行了缓存，并设置10s生存期。

5.7.2 路由和功能设计

本节只是以开发一个简单的论坛原型为例学习Koa的使用方法，所以只需有基本的注册、登录、登出、发帖、评论、主页、用户页、话题页，有简单的版块和分页功能即可。

设计路由如下：

```

GET /signup //注册页
POST /signup //注册

GET /signin //登录页
POST /signin //登录

GET /logout //登出

GET /create //发帖页
POST /create //发帖

GET //主页

GET /user/:name //用户页

```

```
GET /topic/:id //话题页
POST /topic/:id //评论
```

对应构建的routes目录结构如图5-4所示。

```
routes/
├── create.js
├── index.js
├── logout.js
├── signin.js
├── signup.js
├── topic
│   └── *id.js
└── user
    └── *name.js
```

图5-4 routes目录结构

5.7.3 自定义模型

因为我们的论坛很简单，所以设计users、topics和comments三个集合即可满足需求，其中users集合用来存储用户的信息，topics集合用来存储话题信息，comments集合用来存储话题的评论。构建models目录结构，如图5-5所示。

```
models/
├── comment.js
├── index.js
├── topic.js
└── user.js
```

图5-5 models目录结构

我们使用Mongoose的Schema来定义我们的数据模型，对应文件修改如下。

1. user.js

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



```
var UserSchema = new Schema({
  name: { type: String, required: true },
  email: { type: String, required: true },
  password: { type: String, required: true },
  gender: { type: String, required: true },
  signature: { type: String },
  created_at: { type: Date, default: Date.now },
  updated_at: { type: Date, default: Date.now }
});

UserSchema.index({name: 1});

module.exports = mongoose.model( 'User' , UserSchema);
```

2. topic.js

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

var TopicSchema = new Schema({
  user: {
    name: { type: String, required: true },
    email: { type: String, required: true }
  },
  title: { type: String, required: true },
  content: { type: String, required: true },
  tab: { type: String, required: true },
  pv: { type: Number, default: 0 },
  comment: { type: Number, default: 0 },
  created_at: { type: Date, default: Date.now },
  updated_at: { type: Date, default: Date.now }
});

TopicSchema.index({tab: 1, updated_at: -1});
TopicSchema.index({ 'user.name' : 1, updated_at: -1});

module.exports = mongoose.model( 'Topic' , TopicSchema);
```

3. comment.js

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

var CommentSchema = new Schema({
  topic_id: { type: ObjectId, required: true },
  user: {
    name: { type: String, required: true },
    email: { type: String, required: true }
  },
  content: { type: String, required: true },
  created_at: { type: Date, default: Date.now },
  updated_at: { type: Date, default: Date.now }
});

CommentSchema.index({topic_id: 1, updated_at: 1});

module.exports = mongoose.model( 'Comment' , CommentSchema);
```

我们根据需求添加了合理的索引，比如可以通过版块名（tab）或用户名（user.name）按时间降序查找话题，topics集合则添加了对应的索引：

```
TopicSchema.index({tab: 1, updated_at: -1});
TopicSchema.index({'user.name': 1, updated_at: -1});
```

最后，通过index.js导出以供其他模块调用。

1. index.js

```
var mongoose = require( 'mongoose' );
var config = require( 'config-lite' ).mongodb;

mongoose.connect(config.url, function (err) {
  if (err) {
    console.error( 'connect to %s error: ', config.url, err.
message);
    process.exit(1);
  }
});
```

```

    }
  });

  exports.User = require( './user' );
  exports.Topic = require( './topic' );
  exports.Comment = require( './comment' );

```

接下来我们根据需求定义对应的模型函数，构建lib目录结构，如图5-6所示。

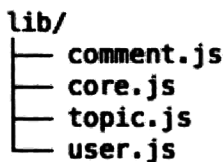


图5-6 lib目录结构

2. user.js

```

var User = require( '../models' ).User;
//新建一个用户
exports.addUser = function (data) {
  return User.create(data);
};

//通过id获取用户
exports.getUserById = function (id) {
  return User.findbyId(id).exec();
};

//通过name获取用户
exports.getUserByName = function (name) {
  return User.findOne({name: name}).exec();
};

```

3. topic.js

```

var Topic = require( '../models' ).Topic;
var cache = require( 'co-cache' );

```

```
//新建一个话题
exports.addTopic = function (data) {
    return Topic.create(data);
};

//通过id获取一个话题,并增加pv 1
exports.getTopicById = function (id) {
    return Topic.findByIdAndUpdate(id, {$inc: {pv: 1}}).exec();
};

//通过标签和页码获取10个话题
exports.getTopicsByTab = cache(function getTopicsByTab(tab, p) {
    var query = {};
    if (tab) { query.tab = tab; }
    return Topic.find(query).skip((p - 1) * 10).sort('-updated_at').limit(10).select('-content').exec();
}, 10000);

//获取用户所有话题
exports.getTopicsByName = function (name) {
    return Topic.find({'user.name': name}).sort('-updated_at').exec();
};

//通过ID增加一个话题的评论数
exports.incCommentById = function (id) {
    return Topic.findByIdAndUpdate(id, {$inc: {comment: 1}}).exec();
};

//获取5条最新未评论的话题
exports.getNoReplyTopics = cache(function getNoReplyTopics() {
    return Topic.find({comment: 0}).sort('-updated_at').limit(5).select('title').exec();
}, 10000);

//获取不同标签的话题数
exports.getTopicsCount = cache(function (tab) {
```

```

var query = {};
if (tab) { query.tab = tab; }
return Topic.count(query).exec();
}, 10000);

```

4. comment.js

```

var Comment = require( '../models' ).Comment;

//添加一条评论
exports.addComment = function (data) {
    return Comment.create(data);
};

//根据话题ID获取对应评论
exports.getCommentsByTopicId = function (id) {
    return Comment.find({topic_id: id}).sort('updated_at').
exec();
};

```

最后，通过core.js聚合并导出模型函数。

5. core.js

```

var Comment = require( './comment' );
var Topic = require( './topic' );
var User = require( './user' );

module.exports = {
    get $User () {
        return User;
    },

    get $Comment () {
        return Comment;
    },

    get $Topic () {
        return Topic;
    }
};

```

```
    }  
  };  
};
```

不知读者有没有发现，迄今为止我们并没有写一行错误处理的代码，数据库查询或更新抛出的错误最终会被 `koa-errorhandler` 捕获。

5.7.4 theme的设计

现在我们来完成论坛设计的最后一步——页面的设计。我们使用 `Semantic UI` 进行开发，最终效果图如图5-7~图5-16所示。

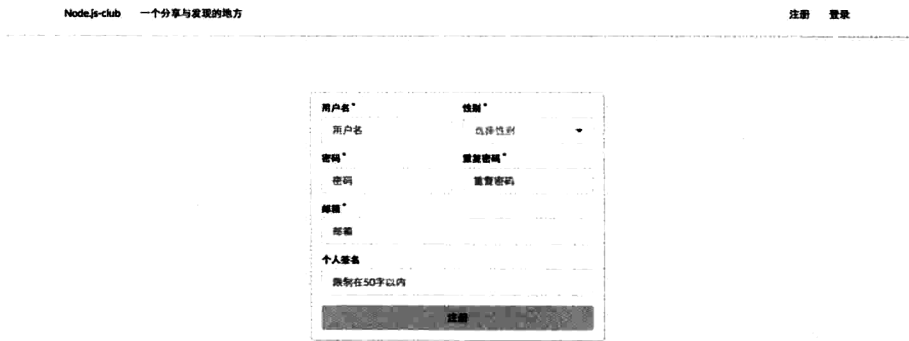


图5-7 注册页

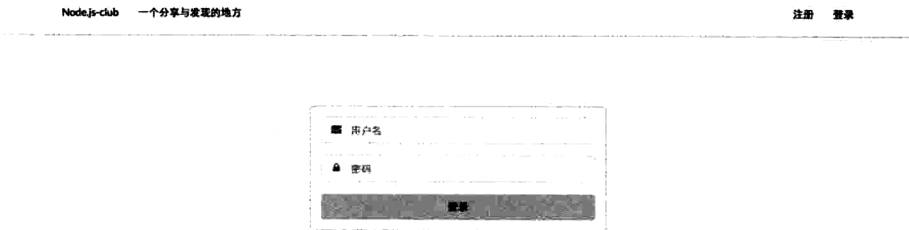


图5-8 登录页

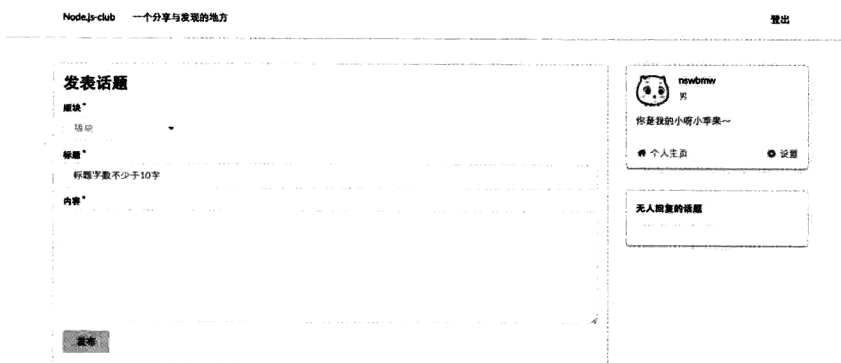


图5-9 发表话题页

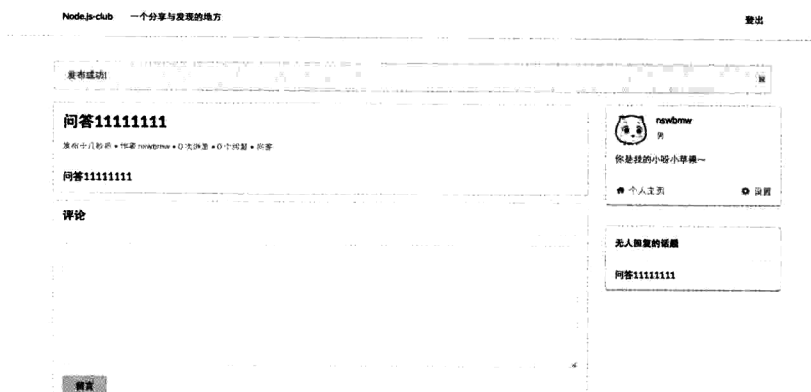


图5-10 登录后的话题页

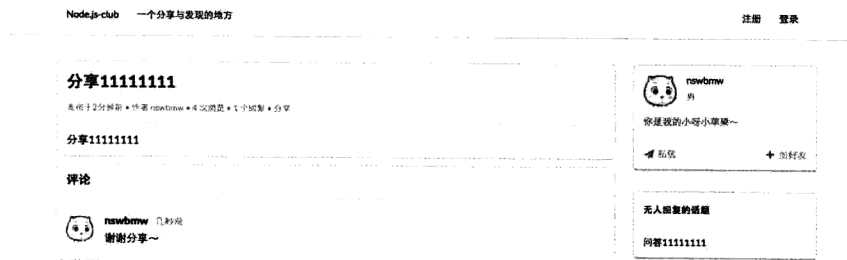


图5-11 未登录时的话题页



图5-12 未登录时的主页

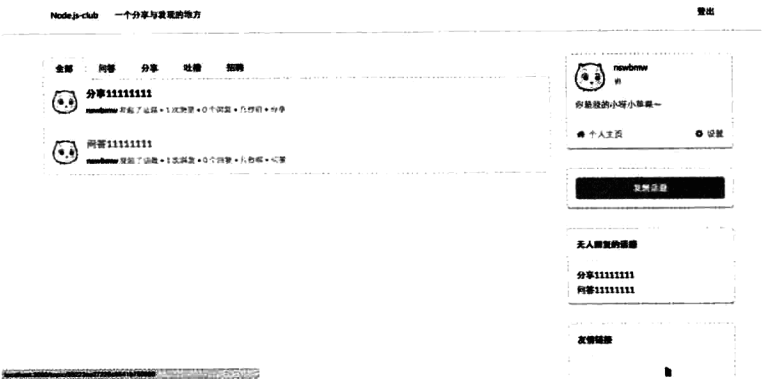


图5-13 登录后的主页

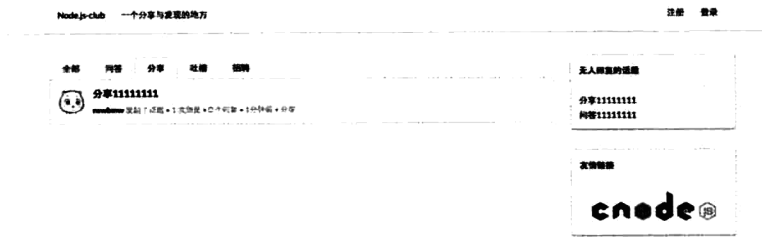


图5-14 版块



图5-15 用户主页

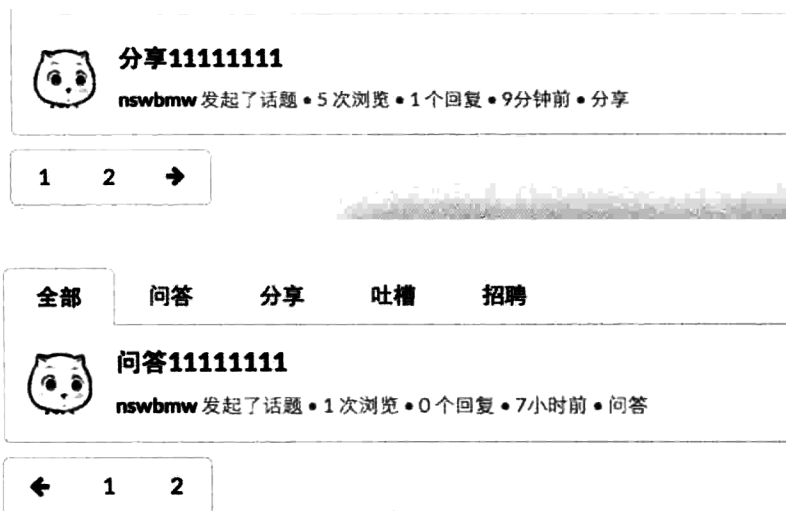


图5-16 分页

我们将页面划分成许多小而可重用的视图片段，如右侧个人信息的卡片userCard.ejs、无人回复的话题的卡片noReplyCard.ejs、友情链接的卡片linkCarde.js，用来发表话题的绿色按钮卡片createCard.ejs、标签视图片段tab.ejs、分页视图片段pagination.ejs，以及主页及用户页都要用到的展示话题列表

的list.ejs。最终，构建theme目录，如图5-17所示。

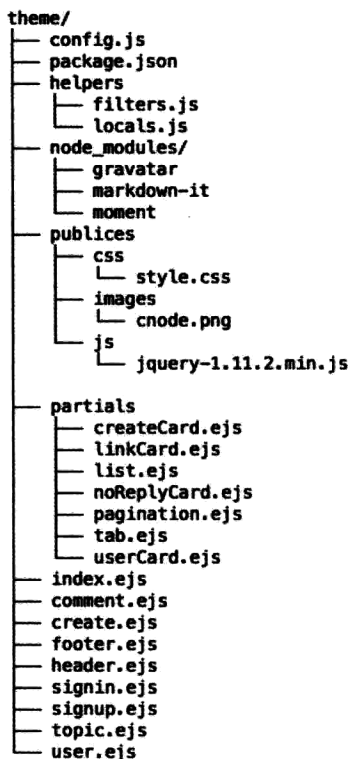


图5-17 theme目录

其中，config.js保存了co-ejs的配置。

1. config.js

```
module.exports = {
  root: __dirname,
  layout: false,
  viewExt: 'ejs',
  cache: true,
  debug: false,
  filters: require( './helpers/filters' ),
  locals: require( './helpers/locals' )
};
```

我们扩展了package.json并保存了主题的相关信息。

2. package.json

```
{
  "name": "Node.js-club",
  "version": "0.0.1",
  "description": "一个分享与发现的地方",
  "locale": "zh-cn",
  "tabs": [ "全部", "问答", "分享", "吐槽", "招聘" ],
  "dependencies": {
    "gravatar": "1.1.0",
    "moment": "2.9.0",
    "markdown-it": "4.0.3"
  }
}
```

helpers/filters.js保存了自定义的过滤器函数，这里我们导出了三个函数分别用于格式化时间、根据email计算gravatar头像及markdown格式转换。

3. filters.js

```
var gravatar = require( 'gravatar' );
var moment = require( 'moment' );
var md = require( 'markdown-it' )();
var pkg = require( '../package' );

moment.locale(pkg.locale);

module.exports = {
  get fromNow () {
    return function (date) {
      return moment(date).fromNow();
    };
  },
  get gravatar () {
    return gravatar.url;
  },
  get markdown () {

```

```
    return function (content) {  
        return md.render(content);  
    };  
}  
};
```

`helpers/locals.js`保存了自定义的本地变量,用于模板的渲染,其中`$app`保存了`package.json`中的数据。

4. locals.js

```
var app = require( '../package' );  
  
module.exports = {  
    get $app () {  
        return app;  
    }  
};
```

最后,修改根目录下的`package.json`,在`scripts`字段添加:

```
"preinstall": "cd theme && npm install"
```

并且`npm install`安装主题所依赖的模块。

5.7.5 注册

我们首先来实现用户的注册功能。修改`default.scheme.js`,添加如下代码:

```
var validator = require('validator');  
var crypto = require('crypto');  
  
module.exports = {  
    "(GET|POST) /signup": {  
        "request": {  
            "session": checkNotLogin  
        }  
    }  
};
```

```
    },  
    "POST /signup": {  
      "request": {  
        "body": checkSignupBody  
      }  
    }  
  }  
};  
  
function md5 (str) {  
  return crypto.createHash('md5').update(str).digest('hex');  
}  
  
function checkNotLogin() {  
  if (this.session && this.session.user) {  
    this.flash = {error: '已登录!'};  
    this.redirect('back');  
    return false;  
  }  
  return true;  
}  
  
function checkLogin() {  
  if (!this.session || !this.session.user) {  
    this.flash = {error: '未登录!'};  
    this.redirect('/signin');  
    return false;  
  }  
  return true;  
}  
  
function checkSignupBody() {  
  var body = this.request.body;  
  var flash;  
  if (!body || !body.name) {  
    flash = {error: '请填写用户名!'};  
  }  
  else if (!body.email || !validator.isEmail(body.email)) {  
    flash = {error: '请填写正确邮箱地址!'};  
  }  
}
```

```
    }
    else if (!body.password) {
      flash = {error: '请填写密码!'};
    }
    else if (body.password !== body.re_password) {
      flash = {error: '两次密码不匹配!'};
    }
    else if (!body.gender || ![ '男', '女' ].indexOf(body.gender)) {
      flash = {error: '请选择性别!'};
    }
    else if (body.signature && body.signature.length > 50) {
      flash = {error: '个性签名不能超过50字!'};
    }
    if (flash) {
      this.flash = flash;
      this.redirect('back');
      return false;
    }
    body.name = validator.trim(body.name);
    body.email = validator.trim(body.email);
    body.password = md5(validator.trim(body.password));
    return true;
  }
}
```

这里我们定义了两个函数`checkNotLogin`和`checkLogin`，用于检查用户的登录状态，并规定只有非登录状态才能访问`signup`页或者提交注册信息。在用户提交注册信息后，`koa-scheme`会对请求体做严格的验证，验证失败则直接返回，验证通过则对请求体做过滤和格式化（将密码md5加密），然后请求才会传递到下一个中间件。

修改`signup.js`，添加如下代码：

```
var Models = require('../lib/core');
var $User = Models.$User;

exports.get = function* () {
  yield this.render('signup');
```

```

};

exports.post = function* () {
  var data = this.request.body;

  var userExist = yield $User.getUserByName(data.name);
  if (userExist) {
    this.flash = {error: '用户名已存在!'};
    return this.redirect('/');
  }

  yield $User.addUser(data);

  this.session.user = {
    name: data.name,
    email: data.email
  };

  this.flash = {success: '注册成功!'};
  this.redirect('/');
};

```

以上代码的意思是：在用户访问/signup时，渲染signup.ejs并返回页面；在用户提交注册信息时，首先检查用户名是否存在，存在则返回“用户名已存在!”的错误提示，不存在则添加该用户，并将该用户的信息保存到session中，返回“注册成功!”的提示。接下来我们完成模板文件和基本的CSS代码。

1. header.ejs

```

<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>N-club</title>
    <link rel="stylesheet" href="https://cdnjs.cloudflare.
com/ajax/libs/semantic-ui/1.12.0/semantic.min.css">
    <link rel="stylesheet" href="/css/style.css">

```

```
<script src="/js/jquery-1.11.2.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/
semantic-ui/1.12.0/semantic.min.js"></script>
</head>
<body>
  <div class="ui fixed menu navbar">
    <div class="container">
      <a href="/" class="item"><%= $app.name %></a>
      <div class="item"><%= $app.description %></div>
      <div class="right menu">
        <% if ($this.session.user) { %>
          <a href="/logout" class="item">登出</a>
        <% } else { %>
          <a href="/signup" class="item">注册</a>
          <a href="/signin" class="item">登录</a>
        <% } %>
      </div>
    </div>
  </div>

  <% if ($this.flash && $this.flash.success) { %>
    <div class="flash">
      <div class="ui green message"><i class="close icon"
></i><%= $this.flash.success %></div>
    </div>
    <% } %>
    <% if ($this.flash && $this.flash.error) { %>
      <div class="flash">
        <div class="ui red message"><i class="close icon"
></i><%= $this.flash.error %></div>
      </div>
      <% } %>

  <script type="text/javascript">
    $(' .message .close').on('click', function() {
      $(this).closest(' .message').fadeOut();
    });
```



```
</script>
```

2. footer.ejs

```
</body>
```

```
</html>
```

3. signup.ejs

```
<% include header %>
```

```
<div class="container">
```

```
  <div class="ui form segment" style="
width:400px;margin:40px auto">
```

```
    <form method="post">
```

```
      <div class="two fields">
```

```
        <div class="field required">
```

```
          <label>用户名</label>
```

```
          <input placeholder="用户名" type="text" name="name">
```

```
        </div>
```

```
        <div class="field required">
```

```
          <label>性别</label>
```

```
          <div class="ui selection dropdown">
```

```
            <div class="default text">选择性别</div>
```

```
            <i class="dropdown icon"></i>
```

```
            <input type="hidden" name="gender">
```

```
            <div class="menu">
```

```
              <div class="item" data-value="男">男</div>
```

```
              <div class="item" data-value="女">女</div>
```

```
            </div>
```

```
          </div>
```

```
        </div>
```

```
      </div>
```

```
    <div class="two fields">
```

```
      <div class="field required">
```

```
        <label>密码</label>
```

```
        <input placeholder="密码" type="password" name="
password">
```

```

        </div>
        <div class="field required">
            <label>重复密码</label>
            <input placeholder=" 重复密码 " type="password "
name=" re_password ">
        </div>
    </div>
    <div class="field required">
        <label>邮箱</label>
        <input placeholder=" 邮箱 " type="email " name="
email ">
    </div>
    <div class="field">
        <label>个人签名</label>
        <input type="text " placeholder=" 限制在50字以内 "
name=" signature ">
    </div>
    <input type="submit " class="ui button fluid" value="
注册 ">
</form>
</div>
</div>

<script type="text/javascript">
    $(''.ui.dropdown').dropdown();
</script>

<% include footer %>
```

4. style.css

```
body{background-color:#f9f9f7;padding-top:100px}
.container{width:100%;max-width:1100px;margin:0 auto}
.navbar{height:60px;line-height:60px;background-
color:rgba(255,255,255,.9) !important}
.flash{max-width:1100px;margin:0 auto 20px}
.summary{display:block;white-
space:nowrap;overflow:hidden;text-overflow:ellipsis}
```

```

.topic{margin-top:10px;color:rgba(0,0,0,.4);font-size:.9rem}
.content{font-size:16px;line-height:24px}
.userCard{height:48px;margin-bottom:10px}
.userCard img{width:48px!important;height:48px;float:left;margin-right:15px}
.userCard .info{line-height:48px}
.list img{width:40px}
.ui.pagination{margin-top:10px !important}
.ui.comments{max-width:100%;font-size:16px}
.ui.menu .item:before{width:0}

```

至此我们就完成了用户注册的功能，启动MongoDB并运行npm start试试吧。

5.7.6 登录与登出

现在，我们来实现用户登录的功能。修改default.scheme.js，在适当的位置添加如下代码：

```

" (GET|POST) /signin": {
  "request": {
    "session": checkNotLogin
  },
  "POST /signin": {
    "request": {
      "body": checkSigninBody
    }
  }
}

function checkSigninBody() {
  var body = this.request.body;
  var flash;
  if (!body || !body.name) {
    flash = {error: '请填写用户名!'};
  }
}

```

```
else if (!body.password) {
  flash = {error: '请填写密码!'};
}
if (flash) {
  this.flash = flash;
  this.redirect('back');
  return false;
}
body.name = validator.trim(body.name);
body.password = md5(validator.trim(body.password));
return true;
}
```

前面介绍过signup的验证逻辑，这里与之类似，不再赘述。修改routes/signin.js，添加如下代码：

```
var Models = require('../lib/core');
var $User = Models.$User;

exports.get = function* () {
  yield this.render('signin');
};

exports.post = function* () {
  var data = this.request.body;

  var userInfo = yield $User.getUserByName(data.name);
  if (!userInfo || (userInfo.password !== data.password)) {
    this.flash = {error: '用户名或密码错误!'};
    return this.redirect('back');
  }

  this.session.user = {
    name: userInfo.name,
    email: userInfo.email
  };

  this.flash = {success: '登录成功!'};
```

```
this.redirect('/user/' + userInfo.name);
};
```

在用户登录时，首先通过用户名查找该用户，如果不存在或存在但密码不匹配，则返回“用户名或密码错误!”的提示。存在且密码匹配则将用户信息写入session并返回“登录成功!”的提示。修改signin.ejs，添加如下代码：

```
<% include header %>

<div class="container">
  <div class="ui form segment" style="
width:400px;margin:60px auto">
    <form method="post">
      <div class="ui left icon input">
        <i class="mail icon"></i>
        <input type="text" placeholder="用户名" name="name">
      </div>
      <br><br>
      <div class="ui left icon input">
        <i class="lock icon"></i>
        <input type="password" placeholder="密码" name="
password">
      </div>
      <br><br>
      <input type="submit" class="ui button fluid" value="
登录">
    </form>
  </div>
</div>

<% include footer %>
```

最后，修改routes/logout.js，添加如下代码：

```
exports.get = function* () {
  this.session = null;
  this.redirect('back');
};
```

至此，我们就完成了用户登录与登出的功能，重启程序试试吧。

5.7.7 主页与版块

现在，我们来实现主页的功能。修改routes/index.js，添加如下代码：

```
var Models = require('../lib/core');
var $Topic = Models.$Topic;

exports.get = function* () {
  var tab = this.query.tab;
  var p = this.query.p || 1;

  yield this.render('index', {
    topics: $Topic.getTopicsByTab(tab, p)
  });
};
```

为了显示主页，我们需将partials目录下的所有卡片的视图都补齐，对应的修改如下。

1. partials/userCard.ejs

```
<% var userInfo = yield $User.getUserByName(name) %>

<div class="ui cards">
  <div class="card">
    <div class="content">
      <div class="userCard">
        
        <div class="info">
          <p><%= userInfo.name %></p>
          <p class="meta"><%= userInfo.gender %></p>
        </div>
      </div>
      <div class="description">
        <%= userInfo.signature %>
      </div>
    </div>
  </div>
</div>
```

```

    </div>
  </div>
  <% if ($this.session.user) { %>
    <div class="extra content">
      <a href="/user/<%= userInfo.name %>" class=""><i
class="home icon"></i>个人主页</a>
      <a href="#" class="right floated"><i class="
setting icon"></i>设置</a>
    </div>
    <% } else { %>
      <div class="extra content">
        <a class=""><i class="send icon"></i>私信</a>
        <a class="right floated"><i class="plus icon"></i>
加好友</a>
      </div>
    <% } %>
  </div>
</div>

```

2. partials/createCard.ejs

```

<div class="ui cards">
  <div class="card">
    <div class="content">
      <a class="ui green button fluid" href="/create">发表
话题</a>
    </div>
  </div>
</div>

```

3. partials/noReplyCard.ejs

```

<% var noReplyTopics = yield $Topic.getNoReplyTopics() %>

<div class="ui cards">
  <div class="card">
    <div class="content">
      <b>无人回复的话题</b>
    </div>
  </div>
</div>

```

```
<div class="ui divider"></div>
<div class="ui list">
  <% noReplyTopics.forEach(function (topic) { %>
    <a class="item summary" href="/topic/<%= topic._
id %>"><%= topic.title %></a>
    <% }) %>
  </div>
</div>
</div>
</div>
```

4. partials/linkCard.ejs

```
<div class="ui cards">
  <div class="card">
    <div class="content">
      <b>友情链接</b>
      <div class="ui divider"></div>
      <div class="ui list">
        <div class="item">
          <a href="https://cnodejs.org/" target="_blank"
></a>
        </div>
      </div>
    </div>
  </div>
</div>
```

5. partials/tab.ejs

```
<div class="ui top attached tabular menu">
  <a href="/" class="item <%= (!this.query.tab || (this.
query.tab == $app.tabs[0])) ? 'active' : '' %>"><%= $app.tabs[0]
%></a>
  <% $app.tabs.slice(1).forEach(function (tab) { %>
    <a href="/?tab=<%= tab %>" class="item <%= (this.
query.tab == tab) ? 'active' : '' %>"><%= tab %></a>
    <% }) %>
  </div>
```


6. partials/list.ejs

```

<div class="ui bottom attached active tab segment">
  <div class="ui list">
    <% var topicsLen = topics.length; %>
    <% topics.forEach(function (topic, index) { %>
      <div class="item">
        <a class="ui image" href="/user/<%= topic.user.name
%>"></a>
        <div class="content">
          <a class="header" href="/topic/<%= topic._id %>"
><%= topic.title %></a>
          <p class="topic"><a href="/user/<%= topic.user.
name %>"><b><%= topic.user.name %></b></a> 发起了话题 • <%= topic.
pv %> 次浏览 • <%= topic.comment %> 个回复 • <%=: topic.updated_at
| fromNow %> • <%= topic.tab %></p>
        </div>
      </div>
      <% if (topicsLen > index + 1) { %>
        <div class="ui divider"></div>
      <% } %>
    <% } }); %>
  </div>
</div>

```

7. partials/pagination.ejs

```

<%
  var count = yield $Topic.getTopicsCount($this.query.tab);
  var pages = Math.ceil(count / 10);
  var page = +$this.query.p || 1;
  var tab = $this.query.tab || '';
  var items = [page-2, page-1, page, page+1, page+2];
  var prePage = '?tab=' + tab + '&p=' + (page - 1);
  var nextPage = '?tab=' + tab + '&p=' + (page + 1);
%>
<div class="ui pagination menu">
  <% if (page && page > 1) { %>

```

```

    <a class="icon item" href="?tab=<%= tab %>&p=<%= page - 1 %>">
      <i class="left arrow icon"></i>
    </a>
  <% } %>

  <% items.forEach(function (item) { %>
    <% if (item && (item > 0) && (item <= pages) && (pages >
1)) { %>
      <a class="<%= (page == item) ? 'active': '' %> item"
href="?tab=<%= tab %>&p=<%= item %>"><%= item %></a>
      <% } %>
    <% }); %>

    <% if (page && page < pages) { %>
      <a class="icon item" href="?tab=<%= tab %>&p=<%= page + 1 %>">
        <i class="right arrow icon"></i>
      </a>
    <% } %>
  </div>

```

从中可以看出，我们可以使用一个特殊的变量`$this`，它就是Koa中的`this`。我们还可以直接在co-ejs模板中使用`yield`并执行一个数据库操作的函数。

最后，修改 `index.ejs` 如下：

```

<% include header %>

<div class="container">
  <div class="ui two column centered grid">
    <div class="right floated left aligned four wide column">
      <% if ($this.session.user) { %>
        <% var name = $this.session.user.name; %>
        <% include partials/userCard %>
        <% include partials/createCard %>
      <% } %>
      <% include partials/noReplyCard %>
      <% include partials/linkCard %>
    </div>
  </div>
</div>

```

```

    </div>
    <div class="left floated twelve wide column">
      <% include partials/tab %>
      <% include partials/list %>
      <% include partials/pagination %>
    </div>
  </div>
</div>

<% include footer %>

```

至此，我们就实现了主页和版块的功能。

5.7.8 用户页

实现主页后，用户页就很容易实现了，因为它们的界面是差不多的。修改`user/*name.js`如下：

```

var Models = require('../lib/core');
var $Topic = Models.$Topic;

exports.get = function* (name) {
  yield this.render('user', {
    topics: $Topic.getTopicsByName(name),
    name: name
  });
};

```

修改`user.ejs`，添加如下代码：

```

<% include header %>

<div class="container">
  <div class="ui two column centered grid">
    <div class="right floated left aligned four wide column">
      <% include partials/userCard %>
      <% if ($this.session.user) { %>

```

```
        <% include partials/createCard %>
    <% } %>
    <% include partials/noReplyCard %>
    <% include partials/linkCard %>
</div>
<div class="left floated twelve wide column">
    <% include partials/list %>
</div>
</div>
</div>

<% include footer %>
```

至此，我们就完成了用户页。

5.7.9 发表页与话题页

现在，我们来实现发表话题、话题页及评论的功能。首先，我们来实现发表页的功能，修改`default.scheme.js`，在适当的位置添加如下代码：

```
" (GET|POST) /create": {
  "request": {
    "session": checkLogin
  }
},
" POST /create": {
  "request": {
    "body": checkCreateBody
  }
}

function checkCreateBody() {
  var body = this.request.body;
  var flash;
  if (!body || !body.title || body.title.length < 10) {
    flash = {error: '请填写合法标题!'};
  }
}
```

```

else if (!body.tab) {
    flash = {error: '请选择版块!'};
}
else if (!body.content) {
    flash = {error: '请填写内容!'};
}
if (flash) {
    this.flash = flash;
    this.redirect('back');
    return false;
}
body.title = validator.trim(body.title);
body.tab = validator.trim(body.tab);
body.content = validator.trim(body.content);
return true;
}

```

以上代码用来验证用户状态及发帖内容是否合法，并对请求体格式化。

修改create.js，添加如下代码：

```

var Models = require('../lib/core');
var $Topic = Models.$Topic;

exports.get = function* () {
    yield this.render('create');
};

exports.post = function* () {
    var data = this.request.body;
    data.user = this.session.user;
    var topic = yield $Topic.addTopic(data);

    this.flash = {success: '发布成功!'};
    this.redirect('/topic/' + topic._id);
};

```

最后，将create.ejs的内容修改如下：

```
<% include header %>
```

```

<div class="container">
  <div class="ui two column centered grid">
    <div class="right floated left aligned four wide column">
      <% var name = this.session.user.name; %>
      <% include partials/userCard %>
      <% include partials/noReplyCard %>
    </div>
    <div class="left floated twelve wide column">
      <form method="post">
        <div class="ui form segment">
          <h2 style="margin-top:0">发表话题</h2>
          <div class="field">
            <div class="fields">
              <div class="field required">
                <label>版块</label>
                <div class="ui selection dropdown">
                  <input type="hidden" name="tab">
                  <div class="default text">版块</div>
                  <i class="dropdown icon"></i>
                  <div class="menu">
                    <% $app.tabs.slice(1).forEach(function (tab) { %>
                      <div class="item" data-value="<%= tab %>">
                        <%= tab %>
                      </div>
                    <% }) %>
                  </div>
                </div>
              </div>
              <div class="field required">
                <label>标题</label>
                <input placeholder=" 标题字数不少于10字 " type="
text " name="title">
              </div>
              <div class="field required">
                <label>内容</label>
                <textarea name="content"></textarea>
            </div>
          </div>
        </div>
      </form>
    </div>
  </div>

```

```

        </div>
        <input type="submit" class="ui button" value="发布">
    </div>
</form>
</div>
</div>
</div>

<script type="text/javascript">
    $('.ui.dropdown').dropdown();
</script>

<% include footer %>

```

至此就完成了发帖的功能，接下来实现话题页和评论的功能。修改 `default.scheme.js`，在适当位置添加如下代码：

```

"POST /topic/:id": {
  "request": {
    "session": checkLogin,
    "body": checkReplyTopic
  }
}

function checkReplyTopic() {
  var body = this.request.body;
  var flash;
  if (!body || !body.topic_id || !validator.isMongoId(body.
topic_id)) {
    flash = {error: '回复的帖子不存在!'};
  }
  else if (!body.content) {
    flash = {error: '回复的内容为空!'};
  }
  if (flash) {
    this.flash = flash;
    this.redirect('back');
    return false;
  }
}

```

```
body.content = validator.trim(body.content);  
return true;  
}
```

修改topic/*id.js，添加如下代码：

```
var Models = require('../../lib/core');  
var $Topic = Models.$Topic;  
var $Comment = Models.$Comment;  
  
exports.get = function* (id) {  
  yield this.render('topic', {  
    topic: $Topic.getTopicById(id),  
    comments: $Comment.getCommentsByTopicId(id)  
  });  
};  
  
exports.post = function* (id) {  
  var data = this.request.body;  
  data.user = this.session.user;  
  
  yield [  
    $Comment.addComment(data),  
    $Topic.incCommentById(id)  
  ];  
  
  this.flash = {success: '回复成功!'};  
  this.redirect(this.query.redirect || 'back');  
};
```

需要说明的是，因为在co或者Koa中，yield后面跟的数组或对象，都是并行执行的，所以我们在传入topic.ejs中的两个数据库查询的函数都是并行执行的。同样，添加一条评论及更新评论数也是并行执行的。

修改topic.ejs，添加如下代码：

```
<% include header %>  
  
<div class="container">
```



```

<div class="ui two column centered grid">
  <div class="right floated left aligned four wide column">
    <% var name = topic.user.name; %>
    <% include partials/userCard %>
    <% include partials/noReplyCard %>
  </div>
  <div class="left floated twelve wide column">
    <div class="ui segment">
      <h2 style="margin-top:0">
        <%= topic.title %>
      </h2>
      <div class="topic">
        发布于<%=: topic.created_at | fromNow %> •
        作者 <%= topic.user.name %> •
        <%= topic.pv %> 次浏览 •
        <%= topic.comment %> 个回复 •
        <%= topic.tab %>
      </div>
      <div class="ui divider"></div>
      <div class="content">
        <%-: topic.content | markdown %>
      </div>
    </div>

    <% include comment %>
  </div>
</div>

<% include footer %>

```

修改comment.ejs，添加如下代码：

```

<div class="ui segment">
  <div class="ui comments">
    <h3 class="ui header">评论</h3>
    <div class="ui divider"></div>
    <% comments.forEach(function (comment) { %>
      <div class="comment">

```

```
<a class="avatar" href="/user/<%= comment.user.name %>">
  
</a>
<div class="content">
  <a class="author" href="/user/<%= comment.user.name %>">
    <%= comment.user.name %>
  </a>
  <div class="metadata">
    <span class="date">
      <%=: comment.updated_at | fromNow %>
    </span>
  </div>
  <div class="text">
    <%-: comment.content | markdown %>
  </div>
</div>
</div>
<% }) %>

<% if ($this.session.user) { %>
  <form class="ui reply form" method="post">
    <input type="hidden" name="topic_id" value="<%=
topic._id %>">
    <div class="field">
      <textarea name="content"></textarea>
    </div>
    <input type="submit" class="ui button" value="留言">
  </form>
<% } %>
</div>
</div>
```

至此，我们就完成了论坛的所有功能，重启下程序试试吧。

5.7.10 测试

没有经过测试的程序是不完整的，现在我们来编写测试用例。首先通过：

```
npm i mocha co-mocha co-supertest --save-dev
```

安装测试依赖的模块。然后修改 **app.js**，将

```
app.listen(config.port, function () {
  console.log('Server listening on: ', config.port);
});
```

修改为：

```
if (module.parent) {
  module.exports = app.callback();
} else {
  app.listen(config.port, function () {
    console.log('Server listening on: ', config.port);
  });
}
```

这里通过**module.parent**判断该文件是否被引用，被引用则导出**app.callback()**以供测试，否则通过**app.listen**启动程序。以**signup**为例，我们来看看如何编写测试用例，读者可自行完成其余部分。新建**test/signup.js**，添加如下代码：

```
var User = require( '../models' ).User;
var request = require( 'co-supertest' );
var app = require( '../app' );

describe( '/signup' , function () {
  var agent = request.agent(app);

  before(function (done) {
    User.remove({name: 'nswbmw' }, done);
  });

  after(function (done) {
    User.remove({name: 'nswbmw' }, done);
  });
```

```
it( 'GET /signup without cookie', function* () {
  yield agent.get( '/signup' ).expect(200).end();
});

it( 'POST /signup without cookie', function* () {
  yield agent
    .post( '/signup' )
    .send({
      name: 'nswbmw',
      email: 'nswbmw@example.com',
      password: '123456',
      re_password: '123456',
      gender: '男',
      signature: '你是我的小呀小苹果～',
    })
    .expect(302)
    .end();
});

it('GET /signup with cookie', function* () {
  yield agent.get('/signup').expect(302).end();
});

it('POST /signup with cookie', function* () {
  yield agent
    .post('/signup')
    .send({
      name: 'nswbmw',
      email: 'nswbmw@example.com',
      password: '123456',
      re_password: '123456',
      gender: '男',
      signature: '你是我的小呀小苹果～',
    })
    .expect(302)
    .end();
});
```

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

这里我们通过`var agent = request.agent(app)`使得之后每次请求（`agent.get`或`agent.post`等）都带上cookies，`before()`和`after()`分别用来在测试前和测试后清理测试数据。最后，修改根目录下的`package.json`，在`scripts`中添加：

```
"test": "./node_modules/mocha/bin/mocha --require co-mocha"
```

运行以下命令进行测试：

```
npm test
```

5.7.11 部署

1. 使用 pm2

我们的论坛要部署到线上时，不能靠`npm start`来启动，因为我们`^C`或者断掉SSH连接后服务就终止了，这时我们就需要像`pm2`或者`forever`这样的进程管理器了。以`pm2`为例，首先运行：

```
npm install pm2 -g
```

全局安装`pm2`，修改`bin/start`如下：

```
#!/bin/bash
DEBUG=* NODE_ENV=default pm2 start app.js --node-args="--
harmony" --name "N-club"
```

然后，我们就可以使用以下命令启动论坛了：

```
sh bin/start
```

最后，不要忘记，如果要部署到线上，我们就要切换到`production`环境（即`NODE_ENV=production`），并且实现相应的`config/production.js`和`config/production.scheme.js`。

2. 申请 MongoLab

MongoLab是一个MongoDB云数据库提供商，用户可选择500MB空间的免费套餐用来测试。注册成功后，单击右上角的**Create New**创建一个数据库，成功后单击进入该数据库详情页，注意页面中有一行黄色的警告：

```
A database user is required to connect to this database.  
Click here to create a new one.
```

每个数据库至少需要一个user，所以我们单击Click here创建一个用户。最后将分配给我们的类似下面效果的mongodb url：

```
mongodb://<dbuser>:<dbpassword>@ds045478.mongolab.com:45478/  
n-club
```

覆盖配置文件中的：

```
mongodb://127.0.0.1:27017/club
```

并将 <dbuser> 和 <dbpassword> 替换为刚才创建的用户名和密码。

3. 部署到Heroku

Heroku是一个支持多种编程语言的云服务平台，Heroku也提供免费的基礎套餐供开发者测试使用。现在，我们将论坛部署到Heroku。

首先，需要到<https://toolbelt.heroku.com/>下载安装 Heroku 的命令行工具包 toolbelt；然后登录（如果没有账号，请注册）到Heroku的Dashboard，点击右上角的“+”，创建一个应用。创建成功后运行：

```
$ heroku login
```

填写正确的Email和password验证通过后，本地会产生一个SSH public key，然后输入以下命令：

```
$ git init  
$ heroku git:remote -a 你的应用名称  
$ git add .  
$ git commit -am "first blood"
```

```
$ git push heroku master
```

稍后，我们的论坛就部署成功了。访问：

```
https://你的应用名称.herokuapp.com/
```

试试吧。笔者部署的论坛地址为：

```
https://n-club.herokuapp.com/
```

5.8 小结

本章通过搭建一个简单的论坛讲解了ES6中generator和Koa的使用方法。当然，这个论坛还远远没有完成，它还缺少了诸如用户管理（删号，禁言等）、权限管理（管理员，普通用户）、帖子管理（删帖，锁帖等）、站内搜索、安全管理（防xss和csrf）等功能，读者可自行尝试编写实现代码。除了generator，ES6还引入了许多激动人心的新特性。2015年6月ES6规范已正式发布，V8很快就会跟进。没有了浏览器的包袱，笔者非常鼓励各位Noder尝试新鲜的东西。

本章代码已托管到GitHub：<https://github.com/nswbmw/N-club>。

5.9 参考文献

- es6features - <https://github.com/lukehoban/es6features>
- ECMAScript6入门 - <http://es6.ruanyifeng.com/>
- koa - <http://koa.js.com/>
- Harmony Generator、yield、ES6、co框架学习 - <http://bg.biedalian.com/2013/12/21/harmony-generator.html>
- 红豆园 - <http://koa.rednode.cn/>

- nodeclub - <https://github.com/cnodejs/nodeclub>
- koa-guide - <https://github.com/turingou/koa-guide>
- jonomieberry - <http://www.jonomieberry.com/koa.html>
- 单元测试 - <http://zh.wikipedia.org/wiki/%E5%8D%95%E5%85%83%E6%B5%8B%E8%AF%95>

第6章

Node.js测试服务搭建

6.1 概述

6.1.1 目的

笔者自2013年开始使用Node.js进行服务器开发以来，经历过不少大大小小的商业项目，有写过电子邮件爬虫、基于RESTful的API服务器、服务中间件、类似于ZooKeeper的集群管理系统，以及应用服务器，在此期间也产出了不下100个与Node.js相关的模组。不过说到Node.js项目/模组的测试，可能大部分人的第一反应就是没有反应，笔者也遇到过类似情况，在选择一个应用的测试框架之前，大部分人都会纠结一番，是使用更加极客一点的node-tape，还是选择更加产品化的mocha，其实无论选择哪种测试架构，大家都不得面对选择此类框架所带来的一些潜在的不便。

如果选择了node-tape，虽然API看似简单不少，但tape的测试结果及在代码覆盖率的兼容性方面都太过简陋，原因是这个模组本身的粒度太小，并不能称之为框架，因此在使用它的时候，开发者还需要集成其他模组来完成较为全面的测试服务，这对开发者的技能要求会相对较高；相反，mocha的门槛

相对较低，却缺少灵活性。

本章主要介绍笔者在过去半年内与一家时尚杂志相关的互联网创业团队合作的部分经历，其中最重要的部分工作就是为该团队搭建了一套较基础的测试服务，测试范围涵盖服务器、浏览器、Mag+、Adobe InDesign等平台。希望通过分享这几个月的实践，能够帮助大家找到一种更为全面、完整的方式去保证项目代码的质量。

6.1.2 Pixbi

Pixbi是一家连接时尚杂志与电子商品（亚马逊）的互联网创业团队，通过给电子杂志出版商开发一个专用的Adobe InDesign插件，以让他们在发布的电子杂志中提供一些商品的购买链接，这样消费者在阅读相应的页面时，可以直接购买、收藏及分享自己喜欢的商品如手表、外套、运动短裤等，也可以通过安装了Pixbi应用的iPhone手机对已经打开的杂志进行扫码购买。

从整体上来看，Pixbi与亚马逊、iOS用户之间的数据流如图6-1所示。



图6-1 Pixbi与亚马逊、iOS之间的数据流

有以下三点需要说明：

（1）Pixbi通过分发Adobe Indesign插件提供给“杂志发行商”，同时从亚马逊商城中获取商品数据；

（2）绑定了Pixbi的杂志分发给iPhone/iPad用户使用；

(3) iPhone/iPad用户可跳转至商品的链接。

完成之后，我们在iOS客户端所看到的如图6-2所示。



图6-2 Pixbi客户端产品显示界面

以上并非是对Pixbi这个产品进行硬广告，只是为了更好地让你了解到具体的商业逻辑，一并更好地理解笔者是在跟什么样的服务及平台打交道。

项目主要的技术栈如下所述。

- 使用Node.js(v0.10.x)作为API服务器平台，实现一个基础的用户系统。
- 由于Pixbi的主要服务是将商品的图片转换为购买的链接，因此需要使用诸如AWS Product Adverting之类的第三方数据接口，这也是之后会引入SinonJS的主要原因。
- Pixbi的UI逻辑代码会被集成到iOS的杂志应用中，当时使用了Mag+平台，我们通过构建工具将前端代码按照Mag+的要求打包，然后在用户浏览时以file://协议进行访问。所以对于此部分代码的测试，为了尽量保证测试环境与实际运行环境的相同，就需要在测试前端功能时，同样以文件协议打开。

- Pixbi会提供给杂志出版商一个Adobe InDesign插件，Adobe InDesign插件的环境本身比较复杂，相当于在浏览器中添加了一些可以控制InDesign控件的若干接口（称之为Adobe InDesign DOM：<http://jongware.mit.edu/idcs6js/>），并且混合了Node.js的API。因此在对这个插件进行测试覆盖时，确实遇到了不小的麻烦。

总之，在对Pixbi进行完整的测试覆盖之前，唯一的难点就是平台的多元化，特别是对于像Adobe InDesign这样的“非主流”平台，网络资源的匮乏（基本只能到Adobe的官方论坛进行提问）及官方对于测试环境的支持度不够。所以在某种程度上，要覆盖此类环境的测试，只有通过一些特别的Hack，或是人工手段，这也是我们当时所能做的。

6.2 搭建后端测试服务

针对后端测试服务的搭建，一般分为单元测试、功能性测试及可拓展性测试，通常来说，也会按照单元测试、功能性测试及可拓展性测试对服务器（集群）进行分步验证。

在本次实战中，我们会使用如下技术栈来进行后端测试服务的搭建：

- MochaJS，参见<http://mochajs.org/>；
- SinonJS，参见<http://sinonjs.org/>；
- Node.js，参见官方模组assert <https://nodejs.org/api/assert.html>；
- jscoverage，参见<https://github.com/fishbar/jscoverage>。

在上述4个选项中，读者对于SinonJS可能会比较陌生，但不必担心，稍后将会详细介绍如何使用这个经常容易被人忽略的框架。

另外笔者并没有选择使用Chai.js/Should.js这样的TDD/BDD框架，而是原生的assert，原因是assert足够简单并且可替换，若你比较喜欢使用Chai.js/Should.js，则可直接替代。

6.2.1 单元测试

单元测试用于确保项目源代码的可用性，并按照我们预期的结果运行。这里可以简单看出单元测试主要分为以下三部分。

- (1) 通过书写测试用例来描述最终的商业逻辑及预期结果。
- (2) 确保1中的测试用例拥有足够高的测试覆盖率。
- (3) 单元测试只需确保项目“源”代码的可用性，对于node_modules的代码，我们需要在对应的模組中进行测试。

下面我们就从写测试用例开始，首先在根目录创建一个名为test的文件夹，目录结构如图6-3所示。

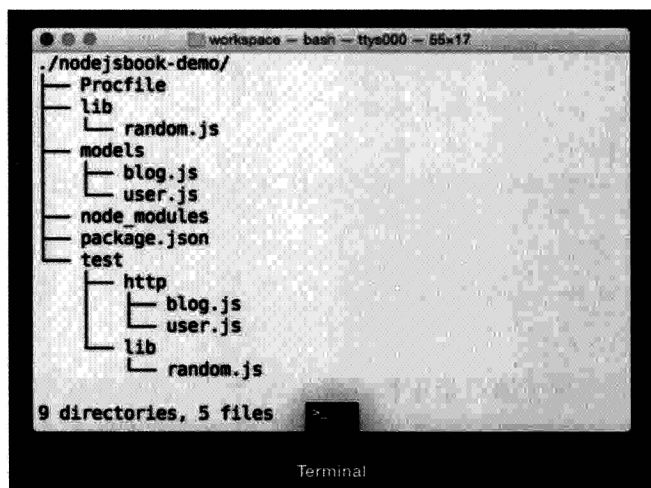


图6-3 nodejsbook-demo目录结构

我们在根目录下创建了test目录，并分为两个子目录。

- (1) lib：编程接口(API)的测试文件放至该目录下。
- (2) http：与http接口相关的测试用例放在此目录下。

关于测试文件是放在lib中还是放在http目录中，最简单的评判标准就是：

如果在测试文件中使用了`supertest`（一个用于测试HTTP接口的开源模组），则可以将这个文件放在`test/http`目录下，反之则放在`test/lib`目录下。

`lib`相对比较简单，我们先来看看`lib/random.js`的逻辑代码：

```
/**
 * random utilities
 *
 * @module random
 */

exports.digits = function (len) {
  if (typeof len !== 'number')
    throw new Error('invalid arguments');

  var bufs = [];
  var collection = '0123456789'

  for (var i=0; i<len; i++) {
    bufs.push(
      collection.charAt(Number(Math.random() * collection.length))
    )
  }
  return bufs.join('');
};
```

上面的函数实现了通过指定长度来生成对应长度的随机字符串的算法，并且在传入非数字类型参数时会抛出相应的异常。下面我们来看看如何测试这段代码：

```
var random = require('../..lib/random')
var assert = require('assert')

describe('lib.random', function () {
  describe('digits()', function () {
    it('should returns fixed length of string', function () {
      var expect = 10
      var actual = random.digits(expect)
```

```

    assert.equal(expect, actual.length)
  })
  it('should catch an error with undefined value', function () {
    assert.throws(
      function () {
        random.digits()
      },
      Error
    )
  })
  it('should catch an error with boolean', function () {
    assert.throws(
      function () {
        random.digits(false)
      },
      Error
    )
  })
})
})

```

通过如下命令安装MochaJS并将其存储在package.json中:

```
$ npm install mocha --save-dev
```

然后更新package.json中的scripts字段:

```

{
  "name": "nodejsbook-demo",
  "version": "1.0.0",
  "description": "nodejs book demo",
  "main": "index.js",
  "directories": {
    "test": "test"
  },
  "dependencies": {
    "body-parser": "^1.12.4",
    "express": "^4.12.4"
  },

```

```
"devDependencies": {
  "mocha": "^2.2.5",
  "supertest": "^1.0.1"
},
"scripts": {
  "test": "NODE_ENV=test mocha test/**/*.js"
},
"author": "",
"license": "MIT"
}
```

在上面代码中的第14行，我们添加了运行单元测试的命令，接下来通过运行命令：

```
$ npm test
```

来获得测试报告，如图6-4所示。

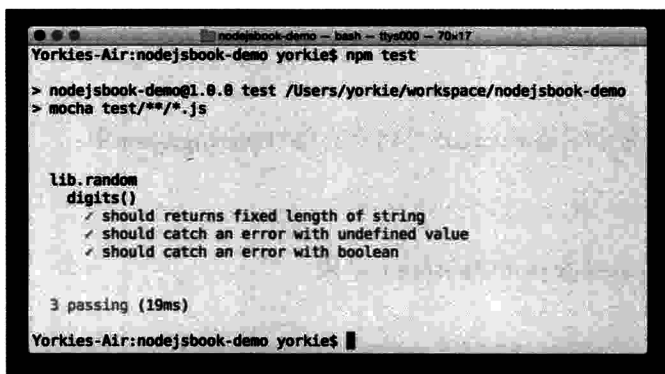


图6-4 单元测试结果

1. 使用supertest

到目前为止，我们已经完成了对lib目录的测试，下面是对http部分的测试。首先我们需要创建如下模型。

```
models/blog.js:
```

```
var Blog = {
```



```

    title    : String,
    text     : String,
    tags     : Array
  };

  module.exports = Blog;

```

接下来我们需要在根目录下新建app.js来使用Blog:

```

var express = require('express')
var bodyParser = require('body-parser')
var random = require('./lib/random')
var app = express()
var prefix = '/api'
var tables = {};

app.use(bodyParser.json())
createModel('blog')

var port = process.env.PORT || 3000

if (process.env.NODE_ENV !== 'test') {
  app.listen(port)
  console.log('start from http://localhost:' + port)
} else {
  module.exports = app
}

function createModel (name) {

```

由于在代码中使用了Express及body-parser模组，因此需要进行安装，代码如下：

```
$ npm install express body-parser --save
```

在上面的代码段的最后一行（17行）定义了createModel函数，其功能即通过我们定义的模型（User和Blog）生成对应的REST风格的HTTP接口。以下是createModel的全部代码：

```

function createModel (name) {
  var model = require('./models/' + name + '.js')
  var createValid = function (inputs, output) {
    return function valid (key) {
      if (!(inputs[key] instanceof model[key] ||
        typeof inputs[key] === model[key].name.toLowerCase())
        && inputs[key])
        return new Error('invalid type on: ' + key)
      else if (inputs[key])
        output[key] = inputs[key]
    }
  }
  var createResponzor = function (errs, input) {
    if (errs.length > 0) {
      return function badRequest (res) {
        res.status(422).end(errs.join(', '))
      }
    } else {
      return function ok (res, code) {
        if (!input.id) input.id = random.digits(10)
        tables[name][input.id] = input
        res.status(code || 200).json(input)
      }
    }
  }

  tables[name] = {}
  app.get(
    prefix + '/' + name + 's',
    function (req, res) {
      res.send(tables[name])
    }
  )
  app.post(
    prefix + '/' + name + 's',
    function (req, res) {
      var data = {}
      var errs = Object.keys(model).map(

```

```

        createValid(req.body, data)
    ).filter(
        function (item) { return !!item }
    )
    createResporndor(errs, data)(res, 201)
}
)
app.get(
    prefix + '/' + name + 's/:id',
    function (req, res) {
        var record = tables[name][req.params.id]
        if (record)
            res.status(200).json(record)
        else
            res.status(204).end()
    }
)
app.patch(
    prefix + '/' + name + 's/:id',
    function (req, res) {
        var record = tables[name][req.params.id]
        if (record) {
            var errs = Object.keys(model).filter(
                createValid(req.body, record)
            )
            createResporndor(errs, record)(res, 200)
        } else {
            res.status(404).end()
        }
    }
)
app.delete(
    prefix + '/' + name + 's/:id',
    function (req, res) {
        delete tables[name][req.params.id]
        res.status(200).end()
    }
)
}

```

现在我们就获得了一个可以正常提供HTTP服务的Node.js实例，不过在开始HTTP测试之前，我们需要先安装supertest模组：

```
$ npm install supertest --save-dev
```

这个模组提供了比较直观的描述性API来简化我们测试http服务的代码，你可以在下面给出的链接中详细学习它。

（1）<https://github.com/visionmedia/supertest>，这个是supertest主页。

（2）<http://visionmedia.github.io/superagent>，由于supertest的大部分API继承于superagent，因此若要查阅API，则可直接到supertest的文档中进行查看。

下面我们开始测试HTTP接口。

```
var app = require('.././app')
var assert = require('assert')
var request = require('supertest')

describe('http.blog', function () {

  var post = {
    title   : 'title',
    text    : 'content',
    tags    : ['tag1']
  }

  describe('GET /blogs/', function () {
    it('should just return the empty object', function (next) {
      request(app).get('/api/blogs/').expect(200, onResponse)
      function onResponse (err, res) {
        next(err)
        assert.deepEqual(res.body, {})
      }
    })
  })

  describe('POST /blogs/', function () {
    it('should post a blog', function (next) {
```

```

    request(app).post('/api/blogs')
    .send(post)
    .expect(201, function (err, res) {
      next(err)
      post.id = res.body.id
      assert.deepEqual(post, res.body)
    })
  })
})

describe('PATCH /blogs/:id', function () {
  it('should update the title', function (next) {
    request(app).patch('/api/blogs/' + post.id)
    .send({'title': 'new title'})
    .expect(200, function (err, res) {
      next(err)
      assert.equal('new title', res.body.title)
      post.title = res.body.title
    })
  })
})

describe('GET /blogs/:id', function () {
  it('should get the updated blog', function (next) {
    request(app).get('/api/blogs/' + post.id)
    .expect(200, function (err, res) {
      next(err)
      assert.deepEqual(post, res.body)
    })
  })
})

describe('DELETE /blogs/:id', function () {
  it('should delete the blog by id', function (next) {
    request(app).delete('/api/blogs/' + post.id)
    .expect(200, next)
  })

  it('should respond null because the id has been erased',
function (next) {
    request(app).get('/api/blogs/' + post.id)
    .expect(204, next)
  })
})

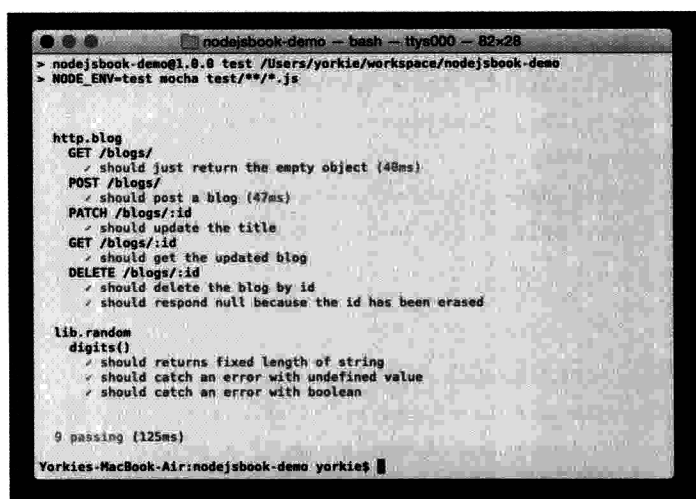
```

```
    })  
  })  
})
```

在上面的代码中，我们分别测试了GET、POST、PUT、PATCH及DELETE接口，现在再次运行：

```
$ npm test
```

可得到如图6-5所示的结果。



```
nodejsbook-demo -- bash -- ttys000 -- 82x28  
> nodejsbook-demo@1.0.0 test /Users/yorkie/workspace/nodejsbook-demo  
> NODE_ENV=test mocha test/**/*.js  
  
http.blog  
  GET /blogs/  
    ✓ should just return the empty object (48ms)  
  POST /blogs/  
    ✓ should post a blog (47ms)  
  PATCH /blogs/:id  
    ✓ should update the title  
  GET /blogs/:id  
    ✓ should get the updated blog  
  DELETE /blogs/:id  
    ✓ should delete the blog by id  
    ✓ should respond null because the id has been erased  
  
lib.random  
  digits()  
    ✓ should returns fixed length of string  
    ✓ should catch an error with undefined value  
    ✓ should catch an error with boolean  
  
9 passing (125ms)  
Yorkies-MacBook-Air:nodejsbook-demo yorkie$
```

图6-5 HTTP测试结果

可看到所有测试通过，HTTP接口测试完成。

2. 使用SinonJS

在实际项目中，我们经常会使用第三方服务，例如：

（1）连接SMTP服务器发送电子邮件；

（2）接入微博，将指定内容转变为一条微博；

（3）从第三方数据公司获取数据，例如通过电子邮件地址获取用户的社交账号信息。

这时运行单元测试，可能无法避免地需要向以上服务器发送请求以完成整个测试流程，我们这时可以使用SinonJS来截断这些额外的请求。下面先来看看什么样的代码需要SinonJS的介入。

首先在lib目录下添加一个lib/mail.js:

```
var nodemailer = require('nodemailer')
var transporter = nodemailer.createTransport({
  service: 'Gmail',
  auth: {
    user: '地址' ,
    pass: 密码
  }
})

module.exports = transporter
```

然后在app.js中添加路由:

```
var mail = require('./lib/mail')
// ... app
app.post('/api/blogs', function (req, res, next) {
  mail.sendMail(
    {
      from: req.query.from || 'Yorkie Liu <yorkiefixer@gmail.com>',
      to: 'Blog Admin <admin@blog.com>',
      subject: 'yorkie is posting a blog',
      text: 'yorkie is posting a blog'
    },
    next
  )
})
```

以上代码在添加Blog之前会发送一封电子邮件至admin@blog.com。关于nodemailer的用法，大家可以去<https://github.com/andris9/Nodemailer>了解。

现在再运行测试，则会得到如图6-6所示的错误。

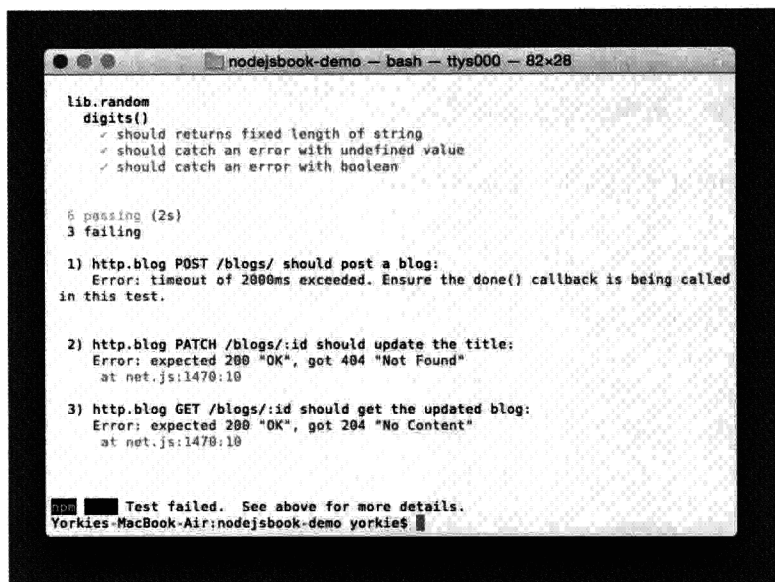


图6-6 在单元测试中请求Gmail超时

这是因为请求Gmail服务器的时间超过了2s（mocha默认超时时间），这里再具体看看Blog模型POST的代码：

```
mail.sendMail(
  {
    from: req.query.from || 'Yorkie Liu <yorkiefixer@gmail.com>',
    to: 'Blog Admin <admin@blog.com>',
    subject: 'yorkie is posting a blog',
    text: 'yorkie is posting a blog'
  },
  next
)
```

我们发现，这里只需确保正确地调用mail.sendMail函数即可，接下来看看如何使用SinonJS优化这个流程：

```
var sinon = require('sinon')
var mail = require('../lib/mail')
```

首先在test/http/blog.js中加载sinon及mail。在本例中会使用SinonJS的

`sinon.stub(object, "method")`来为我们指定的函数`mail.sendMail`创建一个`stub`:

```
describe('POST /blogs/', function () {
  it('should post a blog', function (next) {
    // 创建mail.sendMail的stub对象
    var send = sinon.stub(mail, 'sendMail')
    send.onFirstCall().callsArg(1)
    // 原有的http测试
    request(app).post('/api/blogs')
      .send(post)
      .expect(201, function (err, res) {
        next(err)
        post.id = res.body.id
        assert.deepEqual(post, res.body)
        // 进行参数的确认
        assert.deepEqual({
          from: 'Yorkie Liu <yorkiefixer@gmail.com>',
          to: 'Blog Admin <admin@blog.com>',
          subject: 'yorkie is posting a blog',
          text: 'yorkie is posting a blog'
        }, send.args[0][0])
        // 销毁stub
        send.restore()
      })
  })
})
```

如上代码所示，首先需要熟悉SinonJS中`stub`的概念，SinonJS可以从一个指定的函数中创建`stub`对象，对生成的对象我们可以通过双括号（即JavaScript中执行某函数的语法）来执行`stub`函数，但`stub`执行后，并不会执行原函数中的内容，相应地，其会记录整个运行周期内该函数被执行的次数、参数等数据，用于在测试中做断言（`assert`）。

这里举个例子来更好地说明一下`sinon.stub`的含义。首先声明一个函数：

```
function example_func (next) {
```

```
    console.log( 'executed' )  
    next()  
  }  
}
```

以上函数`example_func`的作用是在调用之后向终端输出`executed`的字符串，并继续执行`next`函数。那么将这个函数变为`stub`后，代码如下：

```
global.example_func = example_func  
var stubbed_example_func = sinon.stub(global, 'example_  
func' )  
    example_func(function () {  
    console.log( 'next' )  
  })  
    stubbed_example_func.called // true
```

由于`sinon.stub`无法直接将函数转换为`stub`对象，需要搭载在一个对象上，所以这里将`example_func`添加到`global`对象中。代码运行到第3行既不会输出`executed`，也不会输出`next`，这是因为`example_func`并没有被执行，但是在第6行中我们仍然得到了`true`的结果。

`stub`的中文译意为“截断的木桩”，这里大家可以把整个运行时函数看成树与树之间的连接，它们彼此之间进行数据（参数）的传递，`stub`则砍掉其中的部分树干或树枝，至于之后如何把数据（参数）再接入到其他树干中，则完全看我们如何使用`stub`的API了。

理解了`stub`的大致意思之后，就可以开始详细说明上一段代码了。

（1）在第4行，通过`sinon.stub`将`mail.sendMail`函数变为`stub`函数，此时所有调用`mail.sendMail`的上下文都将被截断。

（2）在第5行，我们通过`stub`的API做如下声明：在`mail.SendMail`第1次被调用时，将会获取第2个参数，并假设第2个参数为函数类型，并执行这个函数，即函数`mail.sendMail`在服务器中被调用时会直接调用Express中的`next`函数。只有在添加了这一行之后，才可以确保整个流程可以流畅地继续进行。

（3）从第13～19行，由于处在`supertest`的响应回调上下文中，因此可以

明确地知道mail.sendMail函数调用成功，并已经在服务器上创建了Blog的对象，然后返回响应至客户端。因此我们通过stub.args来获取mail.sendMail在刚才调用时的参数列表，并且对其做断言以确保函数得到正确的参数，这一步骤也是我们引入SinonJS最重要的目的之一。

(4) 最后，至第20行，需要对函数mail.sendMail恢复正常的调用状态，因此使用了restore方法。

经过上述的4步之后，再运行测试后的结果如图6-7所示。

```

nodejsbook-demo -- bash -- ttys000 -- 82x28
> nodejsbook-demo@1.0.0 test /Users/yorkie/workspace/nodejsbook-demo
> NODE_ENV=test mocha test/**/*.js

http.blog
  GET /blogs/
    ✓ should just return the empty object (30ms)
  POST /blogs/
    ✓ should post a blog (51ms)
  PATCH /blogs/:id
    ✓ should update the title
  GET /blogs/:id
    ✓ should get the updated blog
  DELETE /blogs/:id
    ✓ should delete the blog by id
    ✓ should respond null because the id has been erased

lib.random
  digits()
    ✓ should returns fixed length of string
    ✓ should catch an error with undefined value
    ✓ should catch an error with boolean

9 passing (125ms)
Yorkies-MacBook-Air:nodejsbook-demo yorkie$

```

图6-7 在单元测试中使用SinonJS后的测试结果

接下来我们需要使用SinonJS的另一个重要API来完成单元测试中对定时任务的精准化测试，与之前一样，先来看看在什么样的应用场景下会遇到此类测试需求。

比如Pixbi，服务器会接收客户上传的小型文件，首先会将其存储在一个临时的s3目录下，经过一段时间后，将这个文件从临时目录转移至永久目录并且删除临时目录下的文件。类似的场景比如用户在上传一张图片时，考虑到如果用户在上传后，一直没有使用，即没有保存，那么此时我们就需要为用户保存该图片一段时间。

在单元测试或功能性测试中，不可能等待这部分时间结束，可选的解决方案有：

- （1）将两个部分拆分为两个部分，分别测试；
- （2）使用SinonJS的fakeTimers。

因此在这里要详细说明了即第二种方案，我们还是先来看看业务逻辑代码是怎样的。

新建一个lib/cache.js:

```
function clearCache () {  
    //清理Blog中的临时数据  
}  
exports.clear = clearCache
```

在app.js中添加中间件:

```
var cache = require('./lib/cache')  
// ... app  
app.post('/api/blogs', function (req, res, next) {  
    setTimeout(cache.clear, 30*60*1000)  
    next()  
})
```

此时我们在运行测试时并不会失败，因为setTimeout并没有阻塞next函数的运行，因此我们需要在测试用例中确保cache.clear在一定时间内被执行过：

```
diff --git a/test/http/blog.js b/test/http/blog.js  
index c3f5dc6..2332a7d 100644  
--- a/test/http/blog.js  
+++ b/test/http/blog.js  
@@ -4,6 +4,7 @@ var assert = require('assert')  
    var request = require('supertest')  
    var sinon = require('sinon')  
    var mail = require('../../lib/mail')  
+var cache = require('../../lib/cache')
```

```

describe('http.blog', function () {

@@ -25,11 +26,16 @@ describe('http.blog', function () {
  describe('POST /blogs/', function () {
    it('should post a blog', function (next) {
      var send = sinon.stub(mail, 'sendMail')
+     var clear = sinon.stub(cache, 'clear')
      send.onFirstCall().callsArg(1)
      request(app).post('/api/blogs')
        .send(post)
        .expect(201, function (err, res) {
-       next(err)
+       if (err) return next(err)
+       setTimeout(function () {
+         assert.ok(clear.calledOnce)
+         next(err)
+       }, 30*60*1000)
      post.id = res.body.id
      assert.deepEqual(post, res.body)
      assert.deepEqual({

```

按照以上diff文件修改app.js后运行测试后的结果如图6-8所示。

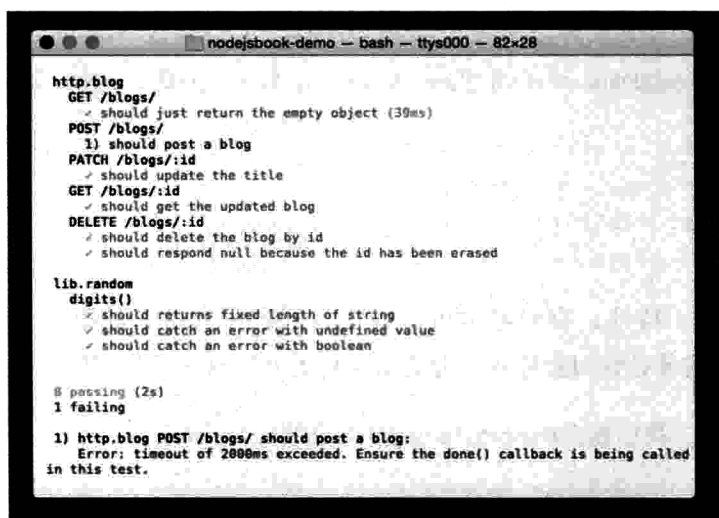


图6-8 未使用Fake Timers时的测试结果

因为我们在测试中添加了`setTimeout(fn, 30*60*1000)`，并将测试结束的函数`next`放在`fn`中执行，不过`mocha`默认超时为`2000ms`，所以明显超时了，测试失败。

现在我们引入`fakeTimers`来规避这个问题，`SinonJS`提供了`useFakeTimers`函数，它会这样做：

(1) 返回一个`clock`对象，可以通过调用`clock.tick`来手动累加`SinonJS`定时器的计数；

(2) 注入`setTimeout/setInterval/setImmediate`等定时器函数，在调用这些函数时，会判断`SinonJS`内置定时器计数与调用时`timeout`的值来控制是否执行定时器内的函数。

举个例子来说明：

```
var clock = sinon.useFakeTimers()
setTimeout(function () { console.log(100) }, 100)
setTimeout(function () { console.log(200) }, 200)
clock.tick(100) //输出100
clock.tick(200) //输出200
```

以上代码在真实环境中并不会运行`300ms`，并且是在一次事件循环内就结束了，并且通过`Date.now()`得到的时间戳也并非真实的值，比如：

```
var clock = sinon.useFakeTimers()
console.log(Date.now()) //0
clock.tick(1000)
console.log(Date.now()) //1000
```

使用这一特性，有时经常可以很快找到程序内潜在的跟定时器有关的Bug，同时也能让你对程序每个分支所消耗的时间有比较具体的预估。

接下来，我们就在`test/http/blog.js`中使用`fakeTimers`：

```
describe('POST /blogs/', function () {
  it('should post a blog', function (next) {
```

```

var clock = sinon.useFakeTimers()
var send = sinon.stub(mail, 'sendMail')
var clear = sinon.stub(cache, 'clear')
send.onFirstCall().callsArg(1)
request(app).post('/api/blogs')
.send(post)
.expect(201, function (err, res) {
  clock.tick(30*60*1000)
  assert.ok(clear.calledOnce)
  next(err)
  post.id = res.body.id
  assert.deepEqual(post, res.body)
  assert.deepEqual({
    from: 'Yorkie Liu <yorkiefixer@gmail.com>',
    to: 'Blog Admin <admin@blog.com>',
    subject: 'yorkie is posting a blog',
    text: 'yorkie is posting a blog'
  }, send.args[0][0])
  send.restore()
})
})
})

```

运行后的测试结果通过，因为我们首先在expect的回调函数中调用clock.tick()将定时器累加到会触发cache.clear的数值，也就可以得到clear.calledOnce等于true的结果了。

3. jscoverage: 覆盖率测试

这里的覆盖率主要指的是基于代码的覆盖率测试，与之相对应的还有基于产品的覆盖率测试。覆盖率是对测试完整程度的评估，与测试系统共同为衡量代码是否稳定、可靠的标准，被植入开发流程中重要的部分。

在Node.js平台下，常用的覆盖率测试工具有：

- (1) <https://github.com/fishbar/jscoverage>;
- (2) <https://github.com/gotwarlost/istanbul>。

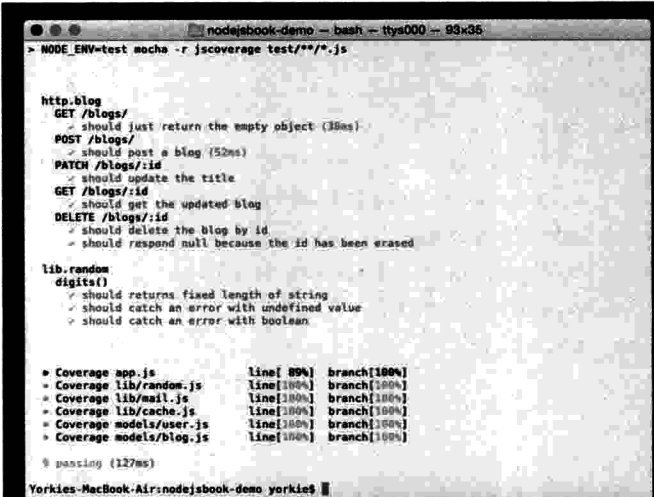
这里我个人比较偏爱jscoverage:

```
$ npm install jscoverage --save-dev
```

按照jscoverage的文档修改测试运行命令如下:

```
NODE_ENV=test mocha -r jscoverage test/**/*.js
```

现在运行测试后得到如图6-9所示的结果。



```
nodejsbook-demo - bash - ttys000 - 93x35
> NODE_ENV=test mocha -r jscoverage test/**/*.js

http.blog
  GET /blogs/
    ✓ should just return the empty object (38ms)
  POST /blogs/
    ✓ should post a blog (52ms)
  PATCH /blogs/:id
    ✓ should update the title
  GET /blogs/:id
    ✓ should get the updated blog
  DELETE /blogs/:id
    ✓ should delete the blog by id
    ✓ should respond null because the id has been erased

lib.random
  digits()
    ✓ should returns fixed length of string
    ✓ should catch an error with undefined value
    ✓ should catch an error with boolean

# Coverage app.js          line[ 89%]  branch[100%]
# Coverage lib/random.js   line[100%]  branch[100%]
# Coverage lib/mail.js     line[100%]  branch[100%]
# Coverage lib/cache.js    line[100%]  branch[100%]
# Coverage models/user.js  line[100%]  branch[100%]
# Coverage models/blog.js  line[100%]  branch[100%]

$ passing (127ms)
Yorkies-MacBook-Air:nodejsbook-demo yorkies$
```

图6-9 增加了覆盖率测试的单元测试结果

至此，我们已经介绍了如下内容。

- (1) 编程接口单元测试。
- (2) 使用supertest测试HTTP接口。
- (3) SinonJS。
 - a. 使用stub专注源代码的测试;
 - b. 使用fakeTimers模拟计时器。
- (4) 使用jscoverage增加覆盖率测试。

6.2.2 功能性测试

单元测试的着眼点为代码库，而功能性测试则偏重于产品方向，因此检验服务器的各个接口是否完整，以及主要业务流程是否完善即为功能性测试的主要作用，也是产品上线前的重要工序。

1. 部署环境

在开始功能性测试之前，首先来了解一些与部署相关的配置。作为最简单的上线部署流程，Pixbi将运行环境分为：

- (1) dev开发环境；
- (2) staging预发布环境；
- (3) production生产环境。

开发环境会紧密地与代码仓库进行绑定，每次提交完成后，都会自动从Github拉取代码并部署至开发环境服务器中，在开发环境中只会运行单元测试中的代码，以及用于提供给前端开发人员日常的开发及调试所用。

预发布环境作为上线之前的测试环境，不仅需要运行单元测试，还需要进行完整的产品功能性测试，在测试中会使用真实的staging服务器作为测试源。除此之外，可拓展性测试也是上线前的重要测试标准。

只有通过了开发环境和预发布环境测试后的代码版本才能被部署至生产环境的服务中。

2. 黑盒

功能性测试并不需要对代码进行校验，因此属于黑盒测试。而对于一个Web服务器或API服务器来说，就是进行HTTP接口测试。这里可能大家会产生疑问：“咦？我们不是已经在单元测试中使用过supertest测试吗？”

答案是使用过，其实HTTP接口测试本身就属于以数据为驱动的黑盒测试（功能测试），因此服务器这边的功能性测试用例已经在test/http目录下完成了，

唯一不同的是我们需要将HTTP服务器指向预发布环境的服务器，而非本地。

3. 编码

那么首先我们要更新app.js中的逻辑：

```
if (process.env.NODE_ENV !== 'test') {
  app.listen(port)
  console.log('start from http://localhost:' + port)
} else {
  if (process.env.TEST_TYPE === 'functional') {
    module.exports = 'https://staging-server'
  } else {
    module.exports = app
  }
}
```

然后在package.json的scripts中添加新的脚本：

```
"scripts": {
  "test": "NODE_ENV=test mocha -r jscoverage test/**/*.js",
  "test-func": "NODE_ENV=test TEST_TYPE=functional mocha test/http/*.js"
},
```

我们增加了环境变量TEST_TYPE=functional，之后测试文件在加载app.js时就会返回预发布环境的URL，下面我们运行：

```
$ npm run test-func
```

时，即可得到服务器端的功能性测试报告。

6.2.3 可扩展性测试

Pixbi的可扩展性测试主要基于以下两方面：

- （1）服务可扩展（水平）；
- （2）功能性测试。

可拓展性测试主要用于测试服务在进行拓展前后的产品行为是否一致，以及性能是否达到预期效果等。主要的步骤有：

- (1) 使用x1的进程实例；
- (2) 进行功能性测试；
- (3) 使用x2的进程实例；
- (4) 进行功能性测试；
- (5) 使用x4的进程实例；
- (6) 进行功能性测试；
- (7) 使用x1的进程实例；
- (8) 进行功能性测试。

可以看出上述测试步骤主要集中在：使用了不同倍数的进程实例，然后进行统一的产品测试，其预期结果应当是完全一致的。

另外，为了保证上述步骤中的单数操作能顺利执行，因此需要你的托管服务器实现拓展接口，Pixbi使用了Heroku作为预发布环境的运行时，因此可以轻松使用scale接口来操作Node.js的实例数量。

众所周知，国内环境在使用Heroku这样的国外服务提供服务时，网络延迟会比较大，体验不会很好。因此对于不愿意或没有使用Heroku的开发者，可使用基于Nginx的负载均衡或nscale来实现服务可拓展性的管理并用于测试。

6.3 搭建前端测试服务

作为最接近用户层的技术，前端在被测体中占据着非常重要的一部分，然而要完整、高自动化地对前端进行测试，还有着非常大的挑战及难度，特

别是针对Pixbi这种多终端的客户体验，其客户端包括Adobe插件、Mag+、Chrome插件及浏览器环境等。

另外，由于前端需要UI渲染，所以测试代码的代价高于服务器端的单元测试，自动化难度系数相对较高。不过好在我们找到了PhantomJS和BrowserStack这两个开源服务，使整个前端测试流程变得足够简单，下面我们就先从PhantomJS开始。

6.3.1 PhantomJS

在开始测试之前，首先要对被测体进行说明，即Pixbi内部的前端架构。Pixbi并未使用AngularJS、React或WebComponents等人们所熟知的框架，而是在内部实现了一个前端构建工具：pixbi/duplo，其网址为<https://github.com/pixbi/duplo>

本节并不会详细介绍duplo如何工作及构建的具体流程，但为了让大家能够明白测试思路，在这里稍微对duplo做一个简单的介绍还是很有必要的。

首先，一个简单的duplo项目结构如图6-10所示。

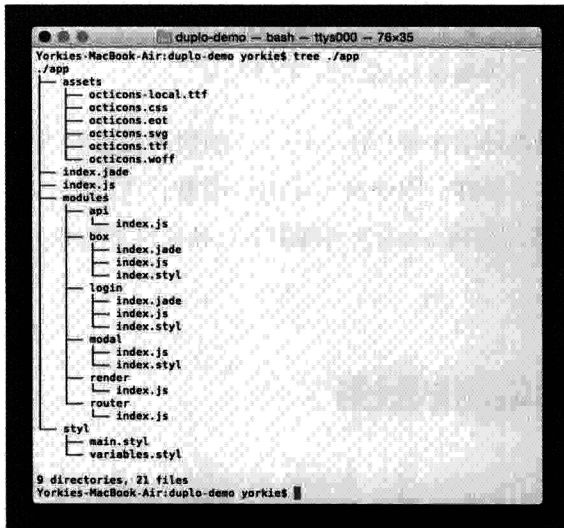


图6-10 duplo项目目录结构

duplo的构建工作如下所述。

(1) 将app目录下的所有*.js文件根据一定规则合并, 并生成至public/index.js。

(2) 将app目录下的所有*.jade文件合并、编译成html, 并生成至public/index.html。

(3) 将app目录下的所有*.styl文件合并、编译成css, 并生成至public/index.css。

(4) 将app/assets目录下的字体文件移至public目录下。

因此, 经过duplo构建完成后的public目录除了字体或资源文件, 应该只包含index.html、index.css及index.js。而接下来, 我们的任务可以拆分为以下4部分。

(1) 增加test目录, 用于存放前端项目单元测试的测试用例代码。

(2) 生成index.html与index.js, 其中包含mocha、SinonJS及test目录内的代码。

(3) 使用PhantomJS加载index.html。

(4) 将测试结果返回到终端。

前两个任务已经在duplo中完成了。由于duplo内部使用Haskell实现, 若转换为gulp, 则只需在原有的gulp.src中加入test/*.js:

```
gulp.src([
  'app/modules/**/*.js',
  'test/*.js'
])
// ...
```

然后使用mocha-phantomjs完成第3个任务和第4个任务:

```
$ npm install mocha-phantomjs --save-dev
$ cd public && ../node_modules/.bin/mocha-phantomjs index.html
```

下面我们来看看实际的应用例子。

在我们服务器的示例项目目录下，新建一个public文件夹，并创建index.html，代码如下：

```
<html>
<head>
  <title> blog </title>
  <link rel="stylesheet" type="text/css" href="mocha.css">
  <script type="text/javascript" src="mocha.js"></script>
  <script type="text/javascript" src="should.js"></script>
</head>
<body>
  <div id="mocha">
    <p>
      <a href=".">Index</a>
    </p>
  </div>
  <script type="text/javascript">
    mocha.setup('bdd');
  </script>
  <script type="text/javascript" src="index.js"></script>
  <script type="text/javascript">
    if (window.mochaPhantomJS) {
      mochaPhantomJS.run()
    } else {
      mocha.run()
    }
  </script>
</body>
</html>
```

由于在浏览器环境中并不自带assert函数，所以这里使用了should.js，因此需要先运行：

```
$ npm install should --save-dev
```

然后创建index.js，这里为了方便，直接将lib/random.js与test/lib/random.js

的代码复制过来并改为should的方式:

```
exports.digits = function (len) {
  if (typeof len !== 'number')
    throw new Error('invalid arguments');

  var bufs = [];
  var collection = '0123456789'

  for (var i=0; i<len; i++) {
    bufs.push(
      collection.charAt(Number(Math.random() * collection.
length))
    )
  }
  return bufs.join('');
};

describe('lib.random', function () {
  describe('digits()', function () {
    it('should returns fixed length of string', function () {
      var expect = 10
      var actual = random.digits(expect)
      actual.length.should.be.eql(expect)
    })
  })
})
```

添加package.json中的scripts如下:

```
"scripts": {
  "test": "NODE_ENV=test mocha -r jscoverage test/**/*.js",
  "test-func": "NODE_ENV=test TEST_TYPE=functional mocha
test/http/*.js",
  "test-fe": "NODE_ENV=test ./node_modules/.bin/mocha-
phantomjs public/index.html "
},
```

运行npm run test-fe后可得到正确的运行结果。至此,我们了解到如

何通过mocha-phantomjs在headless浏览器运行一些通用、简单的单元测试（headless浏览器：即提供一个虚拟的浏览器运行环境，但并没有实际的用户界面）。

6.3.2 BrowserStack

在Headless浏览器/PhantomJS环境下，只能用来测试一些比较基础的JavaScript代码。为了更全面、可靠地测试与用户界面相关的部分，我们需要一个真实的浏览器环境来进行测试。

browserstack.com集成了将近700个浏览器的测试环境，并内建WebDriver的API。下面就来看看如何在项目中使用BrowserStack服务。

1. 注册browserstack账号

首先需要到browserstack.com注册一个账号。

2. browserstack-runner

接下来在项目根目录下安装browserstack-runner：

```
$ npm install browserstack-runner --save-dev
```

然后在public目录下运行：

```
$ cd public && ./node_modules/.bin/browserstack-runner init
> Generated `browserstack.json` using preset "default"
having 8 browsers.
```

现在可以看到新生成的browserstack.json：

```
{
  "username": "BROWSERSTACK_USERNAME",
  "key": "BROWSERSTACK_KEY",
  "test_path": "path/to/test/runner",
  // ...
}
```


我们使用的是mocha，因此这里还需要添加：

```
{
  // ...
  "test_framework": "mocha",
  "timeout": 60,
  // ...
}
```

更新之后，登入BrowserStack账号，访问<https://www.browserstack.com/accounts/automate>。

可以看到username和access key的值，把它们分别复制到browserstack.json文件中的BROWSERSTACK_USERNAME与BROWSERSTACK_KEY。最后我们将test_path的值改为public/index.html，运行：

```
$ ./node_modules/.bin/browserstack-runner
```

在第一次运行或者没有BrowserStackLocal文件时，browserstack-runner会根据操作系统的不同下载对应的BrowserStackLocal，这个文件本身相对较大（7MB），所以会花费比较长的时间进行下载。

下面我们就来看看BrowserStackLocal的作用，如图6-11所示，了解为什么我们需要花费时间下载。

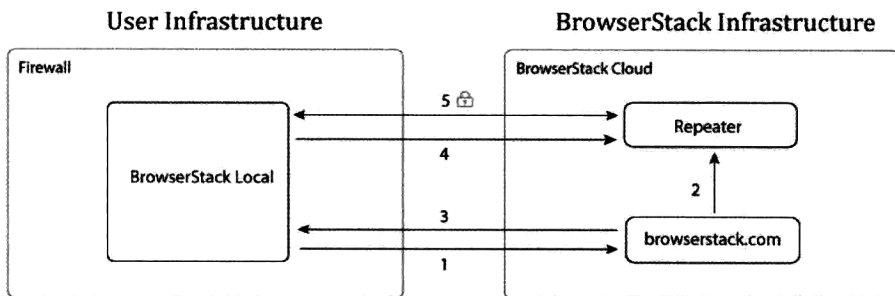


图6-11 Browser Stack数据流

BrowserStack具体是如何工作的呢？简单地说，在BrowserStack云上部

署了各种不同平台（Windows/Linux/MacOSX）下不同版本的浏览器，可以让我们在不同的浏览器上测试我们的代码。那么相应地，我们就需要往BrowserStack上传代码及浏览器列表，因此大家可以看到在browserstack.json的browsers字段中就定义了这样一个列表，用来告诉BrowserStack代码需要运行在哪些浏览器环境下。下面是一个browser定义的示例：

```
{
  "browser": "chrome",
  "browser_version": "latest",
  "os": "OS X",
  "os_version": "Lion"
}
/*以上JSON声明了一个BrowserStack的运行环境为：OSX Lion上最新的
Chrome浏览器*/
```

显然，BrowserStackLocal就是测试服务器与BrowserStack服务器之间的通信层，用于发送测试请求与接收测试结果，而browserstack-runner则将BrowserStackLocal的接口简化并和mocha这样的测试框架进行整合，使整个测试流程更加简单、顺畅。

现在，运行了browserstack-runner命令后的结果如下：

```
Yorkies-MacBook-Air:public yorkie$ ../node_modules/.bin/
browserstack-runner --verbose
Using config: /Users/yorkie/workspace/nodejsbook-demo/public/
browserstack.json
Launching server on port: 8888
[Sun Jun 14 2015 13:53:48 GMT+0800 (CST)] Launching tunnel
[Sun Jun 14 2015 13:53:56 GMT+0800 (CST)] Tunnel launched
Launching 1 worker(s) for 1 run(s).
[OS X Lion, Chrome latest] Finding version.
[OS X Lion, Chrome latest] Version is 43.0.
[OS X Lion, Chrome 43.0] Launching
[OS X Lion, Chrome 43.0] Launched
[OS X Lion, Chrome 43.0] Acknowledged
[OS X Lion, Chrome 43.0] Passed: 1 tests, 1 passed, 0 failed;
ran for 47ms
```

```
[OS X Lion, Chrome 43.0] Terminated
All tests done, failures: 0.
Exiting
```

这证明我们的测试成功运行并返回了结果，现在打开browserstack.com网站的Automation部分，如图6-12所示。

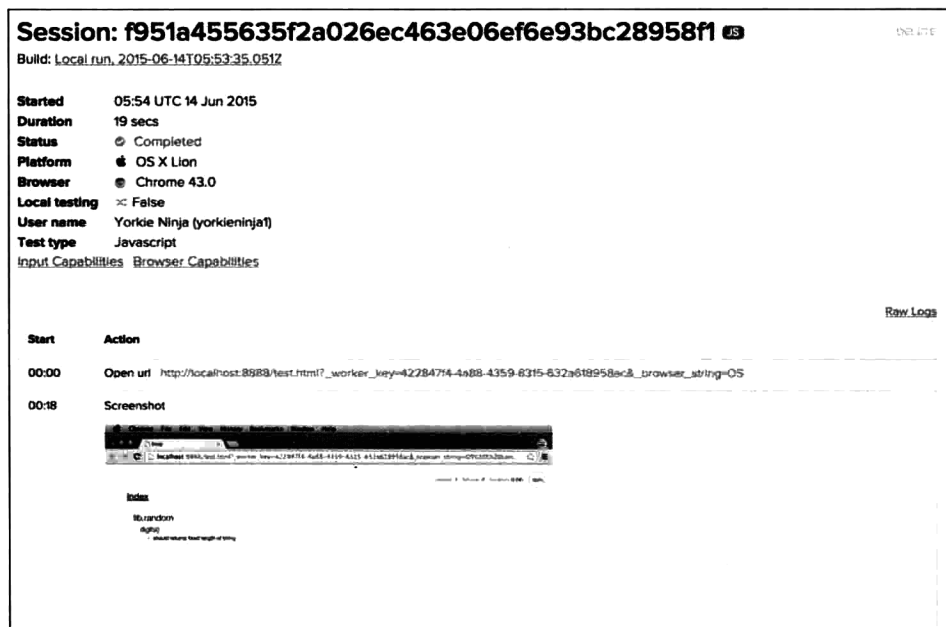


图6-12 Browser Stack测试截图

6.3.3 Adobe CEP (Common Extensibility Platform)

经过BrowserStack的测试后，大部分的环境都已经被其囊括其中了，不过在Pixbi中，还需要分发Adobe的插件给客户，其实这部分内容本身和iOS/Android一样，并不属于Node.js的范畴，然而，CEP却是基于Node.js、HTML及Adobe自身的DOM混合而成的一个运行时，因此这里会顺带介绍一下，在非主流平台并且本身对于测试支持度又不够的条件下，如何确保软件的可靠性和可用性。

首先，简单介绍一下CEP环境。简而言之，如果我们要给任何一个Adobe软件写插件，则需要与CEP架构打交道，比如Photoshop、Lightroom等。当然在Pixbi中，我们实现的是一个InDesign插件。前面也已经提到过CEP的主要组成，更清晰的结构如下。

1. Web

- （1）Node.js：与node-webkit类似，可以使用Node.js的API。
- （2）HTML/CSS/JavaScript：插件的结构基本与前端架构类似。

2. CEP DOM

- （1）Common DOM
- （2）Photoshop DOM
- （3）InDesign DOM

通过上面的结构可以看出，与我们之前所看到的前端项目中，CEP除了可以在运行时直接访问到Node.js的API，最重要的是引入了CEP的DOM树，比如可以通过Photoshop DOM访问到当前在PS中打开的图片的像素大小，或者可以将图片的灰度变为100%，当然前提是需要将插件安装到PS中。

相信看到这里，大家已经能想象到自动化测试的难度，比如插件的其中一个功能是在页面内增加一个下拉菜单选项，把图片的灰度设置为5%，饱和度设置为20%及透明度设置为30%。我们可以很容易地在本地运行测试，并看到运行的结果是否合格，但Adobe官方并未提供一种可以像WebDriver一样通过代码来操作DOM及验证结果的方式，因此这里不得不人工介入。

不过好在CEP DOM树其实被封装成一个单独的运行时独立存在，插件通过类似eval的方式传入命令，再获取结果，或者直接执行命令。所以，我们完全可以使用SinonJS将与CEP DOM相关的函数截断（stub），完成Web部分的测试自动化。

6.4 加入持续集成工作流

至此我们已经完成了从后端到前端所有测试服务的自动化步骤，运行后端测试服务需要使用如下命令：

```
$ npm run test
```

运行前端测试服务需要使用如下命令：

```
$ npm run test-fe
```

总之，我们已经把所有需要的测试需求封装成了简单的命令，现在需要做的只有将测试运行、测试结果加入日常的开发中，即接下来要说的“持续集成”。

持续集成的好处如下。

（1）如果有一名程序员提交了错误的代码，那么持续集成系统会报警，避免错误被忽略。

（2）在同行代码审核中，持续集成的结果是作为代码可审核的一个重要指标，比如只有在单元测试全部通过的情况下，需要进行人工的代码审核，节省了代码审核的时间成本。

（3）持续集成可以加入代码规范检查，比如可以在单元测试之前，先运行jshint。

在Pixbi内部，我们使用CircleCI作为第三方的持续集成平台，添加方法也相当简单，在需要测试的根目录下添加circle.yml文件：

```
machine:
  node:
    version: 0.12.0
  services:
    - redis
    - postgresQL
test:
  pre:
```

```
- jshint .
post:
- npm run test-fe
- npm run test-browserstack
```

CircleCI使用YAML语法的配置文件如下。

- (1) 第3行，声明测试服务器的Node.js版本号，必须与生产环境保持一致。
- (2) 第4~6行，由于Pixbi的服务使用了Redis与PostgreSQL，因此测试服务器也需要提供相应的环境。
- (3) CircleCI在接收到测试请求后，会自动运行npm test，如果希望运行额外的脚本，则可以使用test.pre与test.post，分别对应于测试前后所运行的脚本。因此在第9行，在开始测试之前，我们先进行jshint，检查是否存在语法错误，然后在服务器测试之后，又运行了前端的headless与browserstack测试。

在真实的项目中，前后端可能是被分离至不同的Git仓库的，所依赖的环境也会有所不同，所以circle.yml的写法也会有所变化，需要读者根据CircleCI的文档自行调整。

接下来，我们需要到CircleCI网站将项目添加到集成列表中。找到项目之后，单击“build project”按钮，之后就会被引导至如图6-13所示的页面。

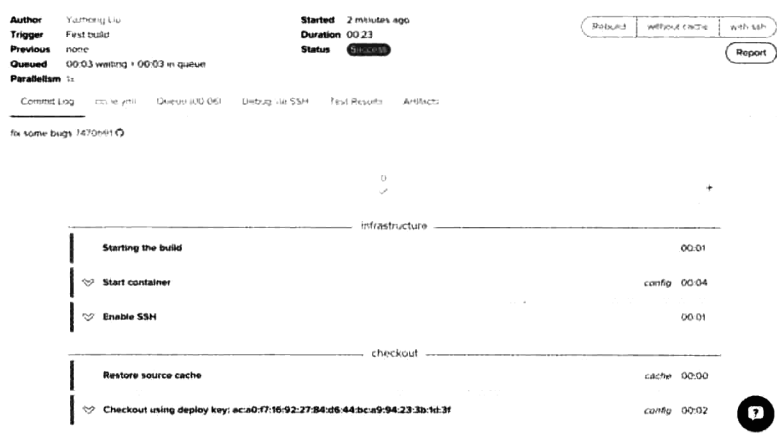


图6-13 CircleCI测试截图

这里可以给大家介绍一个在遇到测试出错时调试的小技巧。首先单击右上角的“with ssh”按钮，此时CircleCI会重新运行任务，不过此时可以看到在底部多出了为图6-14所示的“Wait for SSH”界面。

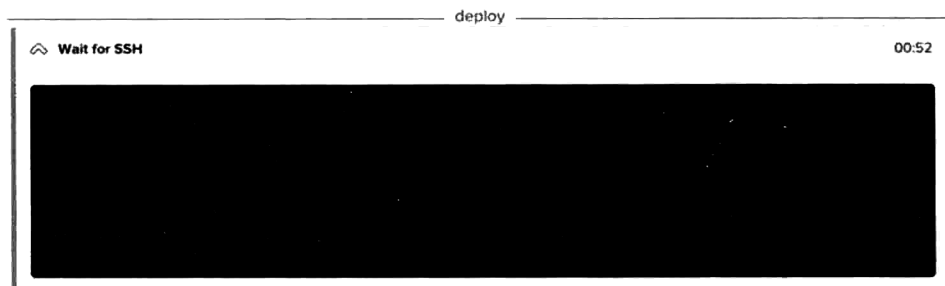


图6-14 SSH在CircleCI上的测试截图

我们按照给出的提示，连接：

```
$ ssh -p 64713 ubuntu@54.197.121.120
```

连接成功后，就可以使用终端对测试服务器进行检查了，如图6-15所示。

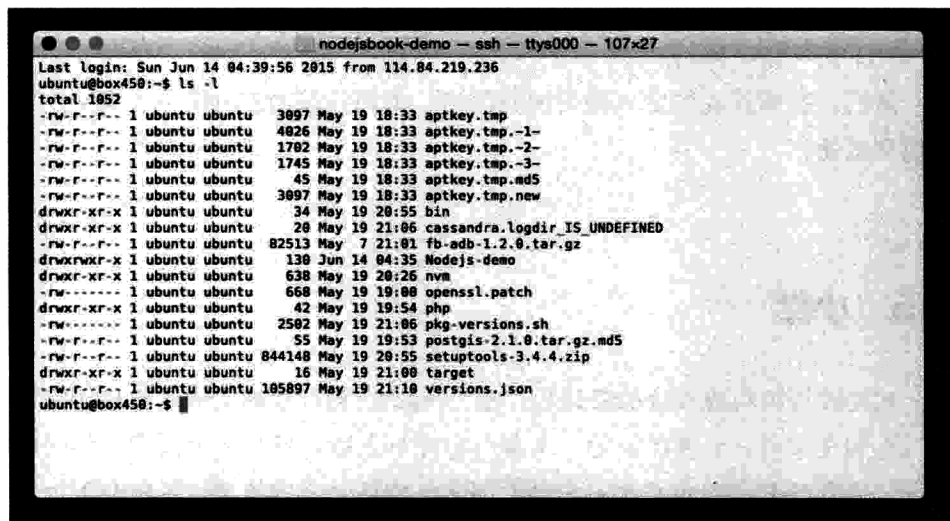
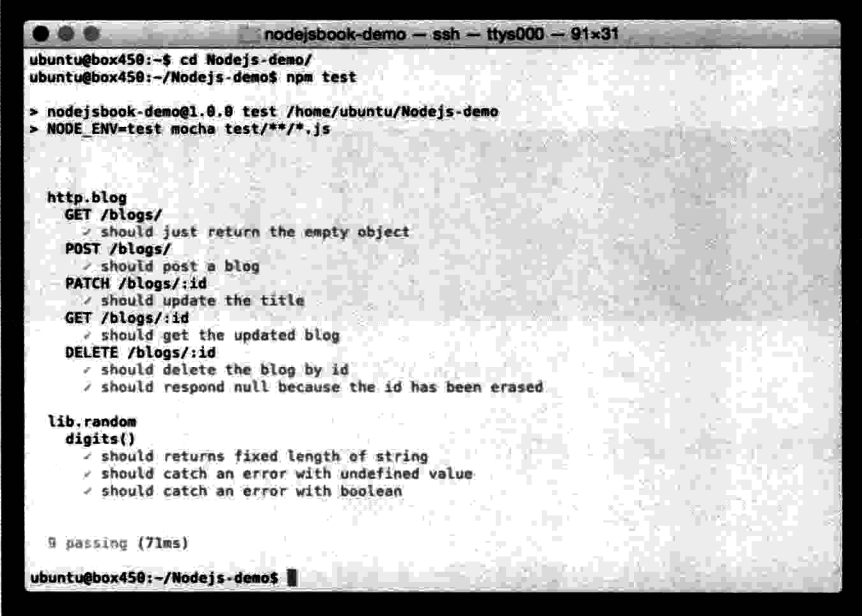


图6-15 用终端对测试服务器进行检查

甚至可进入项目根目录下手动运行测试，检查具体是什么问题导致的测试失败，如图6-16所示。

A terminal window titled 'nodejsbook-demo - ssh - ttys000 - 91x31'. The user is at 'ubuntu@box450:~\$ cd Nodejs-demo/'. They run 'npm test', which runs 'nodejsbook-demo@1.0.0 test /home/ubuntu/Nodejs-demo'. The test command is 'NODE_ENV=test mocha test/**/*.js'. The output shows test results for 'http.blog' and 'lib.random.digits()'. All tests pass, resulting in '9 passing (71ms)'.

```
ubuntu@box450:~$ cd Nodejs-demo/
ubuntu@box450:~/Nodejs-demo$ npm test

> nodejsbook-demo@1.0.0 test /home/ubuntu/Nodejs-demo
> NODE_ENV=test mocha test/**/*.js


http.blog
  GET /blogs/
    ✓ should just return the empty object
  POST /blogs/
    ✓ should post a blog
  PATCH /blogs/:id
    ✓ should update the title
  GET /blogs/:id
    ✓ should get the updated blog
  DELETE /blogs/:id
    ✓ should delete the blog by id
    ✓ should respond null because the id has been erased

lib.random
  digits()
    ✓ should returns fixed length of string
    ✓ should catch an error with undefined value
    ✓ should catch an error with boolean

9 passing (71ms)

ubuntu@box450:~/Nodejs-demo$
```

图6-16 在项目根目录下手动进行测试

最后，在调试完毕之后需要回到网站，关掉挂起的SSH任务，避免阻塞其他测试任务。

6.5 小结

在完成持续集成系统的搭建后，整个Pixbi的测试基础架构就算搭建完成了。正如在本章开头所提及的，本章内容并不太适合目前在大公司工作的程序员，毕竟Pixbi本身也只是一个创业团队，相信大公司的测试系统会更为完善、全面，针对性也更强。

这套方法论可能更加适合那些在创业项目中主要使用JavaScript进行产品

开发，并且一直为产品稳定性所困扰的开发者。这里分享了一种相对全面的解决方案，从各门各类的第三方服务中筛选出了一套成本低、性价比高、适合商业开发的产品链。下面我们就来一起回顾这个产品链中的重要部分。

1. 基础测试框架

- (1) Mocha
- (2) SinonJS
- (3) assert/should/expect
- (4) jscoverage

2. 测试服务器端代码

- (1) 测试编程接口
- (2) 使用supertest测试HTTP
- (3) sinon.stub
- (4) sinon.useFakeTimers
- (5) 功能测试
- (6) 可拓展性测试

3. 测试前端代码

- (1) headless浏览器：PhantomJS或mocha-phantomjs
- (2) Web浏览器：BrowserStack或browserstack-runner

4. 持续集成

- (1) CircleCI

你完全可以在下次的创业产品中使用这套方法构建你的技术栈，从一开

始即保证可以稳定、快速地迭代，进而增加项目成功的概率。

6.6 参考资料

- MochaJS, <https://github.com/mochajs/mocha>
- SinonJS, <http://sinonjs.org>
- supertest, <https://github.com/visionmedia/supertest>
- pixbi, <https://github.com/pixbi>
- mocha-phantomjs, <https://github.com/metaskills/mocha-phantomjs>
- browser-runner, <https://github.com/browserstack/browserstack-runner>
- jscoverage, <https://github.com/fishbar/jscoverage>
- web driver, <https://w3c.github.io/webdriver/webdriver-spec.html#the-webdriver-protocol>
- nscale: <https://github.com/nearform/nscale>

第7章

使用Node.js绑定C语言库—— 51Degrees.node

7.1 开发背景

本章将通过分享笔者与51Degrees团队远程工作的细节，来让各位读者大致了解如何将一个已有的C/C++代码库拓展到Node.js平台上使用先来看一下51Degrees.node开发背景，见图7-1。

51Degrees Brings Device Detection to Node.js

Open source developer community extends 51Degrees to Node.js. The performance enhancing features of device detection are highly relevant to websites built on Node.js.

Reading, UK (PRWEB) March 17, 2015



Yorkie Liu, developer, comments: "The original Node.js implementation was developed for a client in China who agreed to release it as open source on GitHub. 51Degrees were quick to support me with changes and integrate into their official API set."

James Rosewell, CEO of 51Degrees says: "The Node.js implementation of 51Degrees is a really good example of how an open source project supported with a freemium business model can work. Some of the largest web properties in the world use Node.js and they now have access to the most accurate and fastest device detection solution deployed by millions."

For more information on 51Degrees' products and services please visit <http://www.51degrees.com>

图7-1 51Degrees.node开发背景

有些英文底子的读者可以访问以下链接获取这个项目的更多细节。

(1) <http://www.prweb.com/releases/51Degrees/Nodejs/prweb12582979.htm>

(2) <https://github.com/51Degreesmobi/51degrees.node>

下面先来简单介绍51Degrees是做什么的，这里有来自51Degrees的一段自我介绍：

Use this project to detect device properties using HTTP browser user agents as input. It can be used to process server log files, or for real time device detection to support web optimisation.

简单地说，就是在HTTP协议中每一个客户端都会把头中加入一个User-Agent的行，比如在笔者的浏览器中显示如下：

Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/43.0.2357.124 Safari/537.36

然后通过51Degrees所提供的API，传入客户端 / 浏览器的User-Agent，就可获得如下的一组数据：

```
{ Id: '17595-21721-21635-18092',  
  Canvas: true,  
  CssTransforms: true,  
  CssTransitions: true,  
  History: true,  
  Html5: true,  
  IndexedDB: true,  
  IsMobile: false,  
  Json: true,  
  PostMessage: true,  
  Svg: true,  
  TouchEvents: true,  
  WebWorkers: true,  
  method: 'trie' }
```

从函数返回的结果来看，通过使用51Degrees的API可以让我们知道用户

所使用的浏览器的一些细节，比如是否支持如下内容：

- 支持画布（Canvas）；
- 支持HTML5；
- 支持SVG；
- 支持触摸事件等；

现在，简单看看应该如何完成整个工作流程。现在手头上可用的资源有：

- Node.js开发以及运行环境；
- C/C++开发以及运行环境（Linux/Windows）；
- 51Degrees的C语言函数库，可访问<https://github.com/51Degrees/mobi-51degrees-C>。

接下来需要实现的是：

- （1）任务1在Node.js环境下运行C/C++代码；
- （2）任务2通过调用51Degrees-C的API获得需要返回的结果；
- （3）任务3返回到JavaScript/Node.js层；
- （4）任务4完成调用。

对于任务1和任务3，我们可以通过Node.js提供的C++插件来完成，而对于任务2则需要开发者通过阅读C/C++的文档（部分头文件）来访问对应的程序接口。

7.2 预备知识

7.2.1 51Degrees-C

我们先来看看51Degrees原生C语言库的结构，只有在了解底层的程序是

如何工作的及如何让它工作之后，才能有效地完成剩下的编码及测试工作。

由于在51Degrees-C的源代码中嵌入了PHP及其他与代码库无关的代码和工具，所以这里会直接使用在Node.js项目内部的C语言代码作为示例，请访问<https://github.com/51Degreesmobi/51degrees.node/tree/master/src>获得其细节。

它分为如下3个子目录：

（1）`snprintf`读者可直接忽略该目录，51Degrees需要运行一些非标准的C语言运行时，所以它们使用了轻量级的`snprintf`来完成这个兼容；

（2）`pattern`及与`pattern`相关的源代码，稍后会进行简单介绍；

（3）`trie`及与`trie`相关的源代码，稍后会进行简单介绍。

51Degrees为了适应不同场景下的解析需求，提供了两种不同的方式进行UserAgent的解析，它们对应于以上的两个目录：

（1）`pattern`模式适用于更加敏捷的使用场景，可快速更新数据库；

（2）`trie`模式适用于高性能场景，拥有更快的解析速度。

考虑到`pattern`本身的逻辑更为复杂，使用场景更符合Node.js的应用场景，所以主要以`pattern`的实现为主，读者若有兴趣，可在读完本章后去Github上阅读`trie`的相关代码。

为了方便阅读，省掉了命名前缀及一些不相干的函数，若读者需要完整的版本，则可自行与源代码中的函数相对应。接下来简单介绍每个函数的作用：

```
status InitWithPropertyString(const char *fileName, DataSet
*dataSet, char* properties);
// 初始化DataSet对象，第一个参数指定51Degrees提供的设备数据库文件

Workset* CreateWorkset(const DataSet *dataSet);
```

```

// 通过DataSet初始化Workset对象

void FreeWorkset(const Workset *ws);
// 释放指定的Workset对象

void Match(Workset *ws, char* userAgent);
// 在创建好Workset之后, 传入User-Agent后对其进行解析, 并将结果更新到
ws对象

int32_t SetValues(Workset *ws, int32_t requiredPropertyIndex);
// 判断Workset对象是否经过设置

const char* GetPropertyName(const DataSet *dataSet, const
Property *property);
// 获取属性名, 如之前我们看到的canvas, html5, svg等

const char* GetValueName(const DataSet *dataSet, const Value
*value);
// 通常是在调用了GetPropertyName之后, 用于获取对象属性的值

```

在大致清楚了主要函数的功能后, 我们通过熟悉的JavaScript代码描述函数的基本用法:

```

var ret = new Object()
var ua = '...' // User-Agent
var ds = new DataSet()
var status = InitWithPropertyString( 'trie-lite' , ds, 'id,
canvas, html5' )
var ws = CreateWorkset(ds)
if (!Match(ws, ua))
    throw new Error( 'Invalid User-Agent' )
ws.dataSet.properties.forEach(
    function (prop) {
        var key = GetPropertyName(ws.dataSet, prop)
        var val = GetValueName(ws.dstaSet, prop)
        ret[key] = val
    }
)

```

```
FreeWorkset(ws)
return ret
```

在上述代码中：

- （1）第6行调用Match函数，传入User-Agent，并完成解析；
- （2）第8行中的ws.dataSet.properties是一个数组，通过forEach获取每一个属性的对象；
- （3）从第10～12行，分别使用GetPropertyNames和GetValueName获得属性的键值对，并添加到ret对象中；
- （4）倒数第2行，调用FreeWorkset释放ws对象的内存空间（51Degrees-C是基于C语言实现的，因此需要我们手动释放内存）；
- （5）最后一行返回更新后的ret对象。

现在，大致的业务逻辑已经清楚了，下面会介绍如何在C/C++环境下编写Node.js应用。

7.2.2 C/C++中的Node.js API

Node.js的整体架构是基于v8（由Google开源的一款JavaScript高性能解释器，提供C++的编程接口）和C++实现的，因此若要在Node.js中使用C++进行功能拓展，则需要做的就是：

- （1）包含（Include）要使用的C/C++代码库（51Degrees-C）；
- （2）通过调用v8接口，将业务逻辑发布到JavaScript层供上层使用者调用。

Node.js为了让开发C/C++插件的流程更加流畅且基于v8之上，也提供了NODE_MODULE、NODE_SET_METHOD这样的宏来方便开发者。下面简单看看在51Degrees.node中可能会使用到的一些C/C++接口。

首先，在JavaScript层我们所期望的API使用方式为：

```
var PatternParser = require( 'pattern.node' )
var parser = new PatternParser(filename)
parser.parse(ua)
```

可见，这里需要实现的是一个JavaScript可new的函数对象（类），Node.js提供了一个叫作node::ObjectWrap的C++类，让JavaScript中的“类”与我们在C++中定义的类（class）尽量保持一致。这里引用Node.js官方的例子来说明ObjectWrapper的用法：

```
class PatternParser : public node::ObjectWrap {
public:
    static void Init(v8::Local<v8::Object> exports);

private:
    explicit MyObject(double value = 0);
    ~MyObject();

    static void New(const v8::FunctionCallbackInfo<v8::Value>&
args);
    static void Parse(const v8::FunctionCallbackInfo<v8::Value>& args);
    static v8::Persistent<v8::Function> constructor;
    double value_;
};
```

为了方便，此处暂时不对函数进行扩展，不过还是简单介绍下每个函数的大致逻辑：

（1）第3行的Init用于模块的初始化，它定义类PatternParser在JS中准确的变量名，并且通常与宏NODE_MODULE一起使用；

（2）在第9行，函数New()会在调用new PatternParser()，即构造函数时执行其中的代码；

（3）在第10行，Parse对应于parser.parse()的代码。

7.2.3 使用nan

nan也是一个开源的Node.js工具，可参见<https://github.com/nodejs/nan>。在产生之前，开发Node.js的C++插件，需要根据不同的Node.js版本并使用不同的源码，彼此之间的修复并不能共享，版本维护十分复杂，因此nan被开发了出来，用于兼容v8.h与node.h的版本问题，从而使C++拓展的开发与维护工作变得更加简单。现在来看看在51Degrees.node中所使用到的nan宏。

本书出版时Node.js 4.0刚刚发布，而该版本在C++插件上有较大的兼容性问题，因此本章内容只保证适用于Node.js 0.12.x。若读者需要学习Node.js 4.x的相关内容，则将相关API与7.5节相对应即可。

1. NAN_METHOD与NanScope

NAN_METHOD用于定义有效的JavaScript函数：

```
NAN_METHOD(PatternParser::New) {  
    NanScope();  
    // args  
}
```

在定义的函数内，需要先使用NanScope()来进入对应的作用域，NAN_METHOD提供了一个隐形的变量args，开发者可通过它访问到JS函数的参数。

2. NanThrowError

其在NAN_METHOD内使用，效果等同于throw new Error()。

3. NanReturnValue

在NAN_METHOD内使用，用来设置函数返回值，效果等同于return val。

4. NanFalse

等同于return false。

5. NanNew<T>

用于创建类T的实例对象，比如：

(1) `NanNew<String>("new string instance")` 等同于 `new String('new string instance')`;

(2) `NanNew<Boolean>(true)` 等同于 `new Boolean(true)`;

(3) `NanNew<Object>()` 等同于 `new Object()`。

6. NODE_SET_PROTOTYPE_METHOD

假设我们已经通过 `NAN_METHOD` 定义了一个 `PatternParser::Parse` 函数，并创建了一个类（实例模板），然后就可以通过这个宏将该函数设置为该类的方法，代码如下：

```
Local<FunctionTemplate> t = NanNew<FunctionTemplate>(PatternParser::New);
NODE_SET_PROTOTYPE_METHOD(t, "parse", PatternParser::Parse);
```

7.3 编码

在7.2节中，我们基本了解了开发51Degrees.node所需的一些预备知识，现在从代码开始进行51Degrees.node的开发（本章将不再使用简化后的函数名，建议读者在<https://github.com/51Degreesmobi/51degrees.node>上进行参照及对比）。

7.3.1 项目初始化

使用 `npm init` 初始化 `package.json`，然后新建文件/目录，如下所示。

(1) `package.json`：由 `npm init` 生成。

(2) `index.js`：为主模组文件。

(3) `binding.gyp`：之后被 `node-gyp` 使用。

(4) `src`。

- `snprintf`。
- `trie`。
- `pattern`: 51Degrees.* 51degrees-C源代码; `api`.* v8的胶水层代码。

7.3.2 创建v8胶水层接口

我们先创建`src/pattern/api.h`, 代码如下:

```
class PatternParser : public ObjectWrap {
public:

    // @constructor
    // @filename: the database of file, must be *.dat
    // @required_properties: specify properties that would be
returned
    PatternParser(char * filename, char * required_properties);

    // @destructor
    // will release dataset, workset
    ~PatternParser();

    // land this class to node.js runtime
    static void Init(Handle<Object> target);

    // PatternParser.prototype.constructor
    static NAN_METHOD(New);

    // PatternParser.prototype.parse
    static NAN_METHOD(Parse);

private:
    int result;
    fiftyoneDegreesDataSet *dataSet;
    fiftyoneDegreesWorkset *workSet;
};
```

这里需要注意如下四点。

(1) 声明了**PatternParser**，它继承了之前提到的**ObjectWrap**。

(2) 在构造函数中，定义了两个参数：

- **filename** 字符串类型，用于指定*.dat文件的路径；
- **required_properties** 字符串类型，用于定义返回的属性列表，以逗号间隔。

(3) 从倒数第4行，也就是**private**下面开始，分别定义了3个私有成员：

- **result** 整型，调用**InitWithPropertyString()**的返回值会保存在其中；
- **dataSet**在一个**PatternParser**的实例中应该共享一个**fiftyoneDegreesDataSet**实例，因此将其作为私有成员保存于此；
- 与**dataSet**的原理相同。

(4) 另外，我们还定义了析构函数**~PatternParser()**，该函数会在**PatternParser**对象被回收时被动调用，因此只需在此函数内释放**dataSet**和**workSet**的内存即可。

好了，完成了较为简单的头文件后，开始下面的重头戏——**api.cc**，由于这部分内容会比较多，所以这里会将这部分代码以函数为单元分别进行说明。

```
PatternParser::PatternParser(char * filename, char *
requiredProperties) {
    dataSet = (fiftyoneDegreesDataSet *) malloc(sizeof(fiftyoneDe
greesDataSet));
    result = fiftyoneDegreesInitWithPropertyString(filename,
dataSet, requiredProperties);
    workSet = fiftyoneDegreesCreateWorkset(dataSet);
}
```

在上面的代码中完成了**dataSet**和**workSet**的初始化，以及**result**的赋值。

接下来是析构函数：

```
PatternParser::~~PatternParser() {
    if (dataSet)
        free(dataSet);
    if (workSet)
        fiftyoneDegreesFreeWorkset(workSet);
}
```

正如之前提到的，释放dataSet和workSet的内存即可。

```
NAN_METHOD(PatternParser::New) {
    NanScope();
    char *filename;
    char *requiredProperties;

    // convert v8 objects to c/c++ types
    v8::String::Utf8Value v8_filename(args[0]->ToString());
    v8::String::Utf8Value v8_properties(args[1]->ToString());
    filename = *v8_filename;
    requiredProperties = *v8_properties;

    // create new instance of C++ class PatternParser
    PatternParser *parser = new PatternParser(filename, required
Properties);
    parser->Wrap(args.This());

    // valid the database file content
    switch(parser->result) {
        case DATA_SET_INIT_STATUS_INSUFFICIENT_MEMORY:
            return NanThrowError("Insufficient memory");
        case DATA_SET_INIT_STATUS_CORRUPT_DATA:
            return NanThrowError("Device data file is corrupted");
        case DATA_SET_INIT_STATUS_INCORRECT_VERSION:
            return NanThrowError("Device data file is not correct");
        case DATA_SET_INIT_STATUS_FILE_NOT_FOUND:
            return NanThrowError("Device data file not found");
        default:
            NanReturnValue(args.This());
    }
}
```

上面的函数会在new PatternParser()时被执行，我们来看看具体会发生什么：

(1) 从第1行开始，我们使用NAN_METHOD定义了一个JavaScript函数；

(2) 在第6～第10行，完成了从JavaScript/v8类型到C++类型的转换，因为需要将参数传递给51Degrees，而它们不接收v8类型；

(3) 在第13行创建了一个C++类PatternParser的实例parser，然后会调用我们在第一部分所说的构造函数，即会完成dataSet与workSet的初始化工作；

(4) 在第14行调用了Wrap函数，并传入args.This()，Wrap函数（方法）继承自ObjectWrap类，与之对应的函数（方法）还有Unwrap，此处调用Wrap的作用也是在其他函数中通过Unwrap访问到刚创建的PatternParser实例parser；

(5) 接下来的代码都是在对result做判断，如果发现它的值不正确，则使用NanThrowError抛出错误，反之则返回this，也就是实例本身。

接下来看看最重要的Parse函数。由于函数本身逻辑太长，为了简化阅读，将分两步完成，我们首先来看一下相对完整的逻辑：

```
NAN_METHOD(PatternParser::Parse) {
    NanScope();

    // convert v8 objects to c/c++ types
    PatternParser *parser = ObjectWrap::Unwrap<PatternParser>(a
args.This());
    Local<Object> result = NanNew<Object>();
    v8::String::Utf8Value v8_input(args[0]->ToString());

    fiftyoneDegreesWorkset *ws = parser->workSet;
    int maxInputLength = (parser->dataSet->header.
maxUserAgentLength + 1) * sizeof(char);
    if (strlen(*v8_input) > maxInputLength) {
        return NanThrowError("Invalid userAgent: too long");
    }

    // here we should initialize the ws->input by hand for avoiding
```

```

    // memory incropted.
    memset(ws->input, 0, maxInputLength);
    memcpy(ws->input, *v8_input, strlen(*v8_input));
    fiftyoneDegreesMatch(ws, ws->input);

    if (ws->profileCount > 0) {
        // 生成要返回的result对象
    } else {
        return NanFalse();
    }

    NanReturnValue(result);
}

```

在代码中需要注意的有以下几点：

（1）我们在第5行完成了之前所说的对Unwrap函数的使用，读者可以发现，Unwrap最后返回了之前在构造函数中新建的PatternParser实例parser；

（2）在第6~7行，对Parse函数的第一个参数即ua做了一个类型转换，因为我们这里需要C/C++的字符串类型传入51Degrees函数；

（3）倒数第10行调用了fiftyoneDegreesMatch，并传入ua/input，此时51Degrees会对ua进行解析，并更新ws；

（4）接下来会对ws->profileCount进行判断，如果不大于0，则直接返回false，或者执行生成result对象的代码（也是我们在第一步先忽略的）并在最后返回。

然后是Parse函数的第二步，即生成result对象的部分，代码如下：

```

// here we fetch ID
int32_t propertyIndex, valueIndex, profileIndex;
int idSize = ws->profileCount * 5 + (ws->profileCount - 1) + 1;
char *ids = (char*) malloc(idSize);
char *pos = ids;
for (profileIndex = 0; profileIndex < ws->profileCount; profileIndex++) {
    if (profileIndex < ws->profileCount - 1)

```



```

        pos += sprintf(pos, idSize, "%d-", (*(ws->profiles +
profileIndex))->profileId);
        else
            pos += sprintf(pos, idSize, "%d", (*(ws->profiles +
profileIndex))->profileId);
    }
    result->Set(NanNew<v8::String>("Id"), NanNew<v8::String>(ids));
    free(ids);

    // build JSON
    for (propertyIndex = 0;
        propertyIndex < ws->dataSet->requiredPropertyCount;
        propertyIndex++) {

        if (fiftyoneDegreesSetValues(ws, propertyIndex) <= 0)
            break;

        const char *key = fiftyoneDegreesGetPropertyNames(ws->dataSet,
            *(ws->dataSet->requiredProperties + propertyIndex));

        if (ws->valuesCount == 1) {
            const char *val = fiftyoneDegreesGetValueName(ws->dataSet,
*(ws->values));
            // convert string to boolean
            if (strcmp(val, "True") == 0)
                result->Set(NanNew<v8::String>(key), NanTrue());
            else if (strcmp(val, "False") == 0)
                result->Set(NanNew<v8::String>(key), NanFalse());
            else
                result->Set(NanNew<v8::String>(key), NanNew<v8::String>(val));
        } else {
            Local<Array> vals = NanNew<Array>(ws->valuesCount - 1);
            for (valueIndex = 0; valueIndex < ws->valuesCount;
valueIndex++) {
                const char *val = fiftyoneDegreesGetValueName(ws->dataSet,
*(ws->values + valueIndex));
                vals->Set(valueIndex, NanNew<v8::String>(val));
            }
            result->Set(NanNew<v8::String>(key), vals);
        }
    }
}

```

```
    }

    Local<Object> meta = NanNew<Object>();
    meta->Set(NanNew<v8::String>("difference"),
NanNew<v8::Integer>(ws->difference));
    meta->Set(NanNew<v8::String>("method"), NanNew<v8::Integer>
(ws->method));
    meta->Set(NanNew<v8::String>("rootNodesEvaluated"), NanNew<v8::
Integer>(ws->rootNodesEvaluated));
    meta->Set(NanNew<v8::String>("nodesEvaluated"), NanNew<v8::
Integer>(ws->nodesEvaluated));
    meta->Set(NanNew<v8::String>("stringsRead"), NanNew<v8::
Integer>(ws->stringsRead));
    meta->Set(NanNew<v8::String>("signaturesRead"), NanNew<v8::
Integer>(ws->signaturesRead));
    meta->Set(NanNew<v8::String>("signaturesCompared"), NanNew<v8::
Integer>(ws->signaturesCompared));
    meta->Set(NanNew<v8::String>("closestSignatures"), NanNew<v8::
Integer>(ws->closestSignatures));
    result->Set(NanNew<v8::String>("__meta__"), meta);
```

代码仍然很长，这里我再把代码分成三小块进行说明：

（1）前12行（中间有空白）为第一块，主要是将ID解析到result对象上；

（2）第2块从注释“build JSON”开始，一直到倒数第18行（也就是for循环结束那里）结束。其作用主要是访问51Degrees返回的属性值，并将它们添加到result对象中；

（3）最后一块就是剩下的部分，可以让使用者通过JavaScript访问解析过程中的元数据meta。

最后，我们来定义Node.js初始化模组的逻辑，并在其中定义具体的变量和函数名称：

```
void PatternParser::Init(Handle<Object> target) {
    NanScope();
    Local<FunctionTemplate> t = NanNew<FunctionTemplate>(New);
    NODE_SET_PROTOTYPE_METHOD(t, "parse", Parse);
```

```
target->Set (NanNew<v8::String>( " PatternParser " ), t->GetFunction());
}

NODE_MODULE(pattern, PatternParser::Init)
```

这个函数非常简单，不过这里注意：

(1) 在第1行，也就是函数Init定义处，会传入一个target变量，相当于JS模组中的exports或module.exports；

(2) 在最后一行，需要使用宏NODE_MODULE来让Node.js找到该模组的初始化函数，即PatternParser::Init，否则仍然无法在JS中找到辛苦写出来的函数。

截至目前，pattern部分的v8胶水层代码就结束了，读者对trie部分如有兴趣，则可对照着源代码和本节内容进行了解。

7.3.3 创建JavaScript代码

在完成前面的工作之后，可以说与开发一个一般的Node.js模组没什么区别了，所以本节只简单说明如何在JS中访问我们之前创建的函数。

在开始JavaScript代码之前，其实还需要借助node-gyp来编译C/C++代码，不过该部分将在后面进行详细解释，这里先大致介绍下基本流程。

使用node-gyp会根据你所定义的binding.gyp配置生成对应的目标文件，也就是*.node文件。该类文件也可以直接在JS中通过require函数直接加载。而存放此类文件的路径一般为build/Release或build/Debug等。笔者在51Degrees.node项目中就使用了如下代码来加载pattern模组：

```
var PatternParser = require('./build/Release/pattern.node').PatternParser
```

当然，node-gyp的作者创建了一个更为方便的node-bindings来帮助开发者对可能存在的*.node进行扫描，比如上面的代码可以直接改为：

```
var PatternParser = require('bindings')('pattern.node').
PatternParser
```

关于node-bindings的使用方法，读者可参照<https://github.com/TooTallNate/node-bindings>。

7.4 构建与发布

7.4.1 node-gyp与binding.gyp

node-gyp可以看作gyp的node特殊版，不过读者可能对gyp是何物还不甚了解，gyp的英文全名为generate your project，是v8团队所使用的基于Makefile开发的高级构建工具，当然这里的构建指的是C/C++项目，因此，Node.js借助于v8也使用了gyp来提供底层C/C++插件的构建与编译。

在使用node-gyp的过程中，最重要的部分就是编写binding.gyp了，下面我们通过51Degrees.node项目下的binding.gyp来进行学习：

```
{
  "targets": [
    {
      "target_name": "pattern",
      "sources": [
        "src/snprintf/printf.c",
        "src/pattern/51Degrees.c",
        "src/pattern/api.cc",
      ],
      "cflags": [
        "-Wno-trigraphs"
      ],
      "defines": [
        "HAVE_SNPRINTF"
      ],
      "include_dirs": [
        "<!(node -e \"require('nan')\")\" "
      ],
    },
  ],
}
```

这里就不对gyp怎么生成makefile进行展开，只简要说明以下几点。

(1) targets作为一个数组，如无特殊声明，则每个target都会被编译成一个.node文件。

(2) target_name对应于最终生成的.node文件的名字，如上文件编译后会存储为pattern.node。

(3) sources用于指定需要进行编译的源文件，可以不包含头文件，所以对于pattern来说，这里只需要pattern下的源文件及sprintf。

(4) 关于include_dirs，读者可能对 "`<!(node -e \"require('nan')\")`" 这样的语法感到比较陌生，没关系，这其实是使用nan所必需的一步，即可以让源代码直接包含nan模组下的头文件，这也是include_dirs的作用。了解其含义之后来看看上面的语法，我们可以把它分解成两个部分。

- `<!(...)`部分会被node-gyp展开成最后的路径。
- `node -e \"require('nan')\"`部分又可拆分为以下两部分。

a. `node -e <cmd>` 执行cmd内的JS代码，因此-e可理解为execute。

b. `require('nan')`会直接返回nan头文件所在的目录路径（可参考https://github.com/nodejs/nan/blob/master/include_dirs.js），因此与以上几点一起使用后，即可等同于将nan头文件的路径添加至include_dirs。

完成了binding.gyp的配置后，就可以通过如下命令进行编译：

```
$ npm install node-gyp -g
$ node-gyp configure
$ node-gyp build
```

或者通过如下命令进行编译：

```
$ node-gyp rebuild
```

在Windows平台下，有时需要手动指示VS的版本，笔者在51Degrees.node下使用了如下编译脚本：

```
$ node-gyp rebuild --msvs_version=2012
```

7.4.2 发布

Node.js C/C++模组发布的过程非常简单，与一般的JavaScript模组并无不同，我们只需确保用户在安装了所需的源代码后执行node-gyp进行编译即可，可以使用npm的scripts来完成：

```
"scripts": {  
  "install": "node-gyp rebuild --msvs_version=2012"  
},
```

打开package.json，并在scripts下添加如上install行，在用户在安装完成后会自动进行编译，另外，用户有时并未下载node-gyp，也可能是路径、环境变量配置有误，此时可通过如下更新来避免安装或编译失败：

```
"dependencies": {  
  "node-gyp": "某个版本"  
},  
// ...  
"scripts": {  
  "install": "./node_modules/.bin/node-gyp rebuild  
--msvs_version=2012"  
},
```

最后使用npm publish就可以发布给其他用户公开使用了。

7.5 如何从nan 1.x升级到nan 2.x

由于Node.js v4.0的发布，开发者可以使用越来越多的诸如箭头函数、class关键字、模版字符串等这样的ES6的新特性，这一切都得益于v8的升级，不过随之而来的是Node.js对C++插件的兼容性历史遗留问题。由于v8大量引入类似MaybeLocal的新概念，nan不得不重新构建其API，因此大量地使用nan作为API的C++插件也需要升级nan 2.x来完成对Node.js v4.x的兼容，在这里大

致分享一下笔者在升级51degrees.node时所用到的部分API。

(1) `NanScope`被`Nan::HandleScope scope`替代。

(2) `NanEscapableScope`被`Nan::EscapableScope scope`替代，不再提供`NanEscapeScope(exports)`这样的宏，而是直接声明`scope.Escape(exports)`。

(3) 表示函数参数列表的默认变量由`args`变为`info`，因此`args.This()`需要改为`info.This()`。

(4) 在返回函数时，不再需要使用`NanReturnValue()`这样的宏，而是使用：`info.GetReturnValue().Set(info.This())`。

(5) `_NAN_METHOD_ARGS`被`const Nan::FunctionCallbackInfo<v8::Value>& info`替代。

(6) `NanThrowError`被`Nan::ThrowError`替代。

(7) `NanNew`被`Nan::New`替代，另外，当使用`Nan::New`初始化一个`v8::String`类型实例时，如果需要返回`Local<String>`类型的实例，则需要调用`ToLocal()`或`ToLocalChecked()`，例如：`Nan::New<String>("Node.js book").ToLocalChecked()`。

(8) `NanTrue`被`Nan::True`替代。

(9) `NanFalse`被`Nan::False`替代。

(10) `NanUndefined`被`Nan::Undefined`替代。

(11) `obj->Set(key, val)`被`Nan::Set(obj, key, val)`替代，参数的类型分别是：`Local<Object>`、`Local<String>`、`Local<Value>`。其中`key`如果使用`Nan::New`进行初始化，则需要调用`ToLocal()`/`ToLocalChecked()`。

(12) `obj->Set(index, val)`被`Nan::Set(obj, index, val)`替代，与第10条类似，把`key`的`Local<String>`换成`int`类型即可。

(13) `obj->Get(key)`被`Nan::Get(obj, key)`替代，参数类型与第10条类似。

（14）`obj->Get(index)`被`Nan::Get(obj, index)`替代，参数类型与第11条类似。

（15）`NODE_SET_PROTOTYPE_METHOD`被`Nan::SetPrototypeMethod`替代。

如果读者想要系统地学习新的API，则可访问 <https://github.com/nodejs/nan#api>，也可以查看51degrees.node最新版本的代码库（使用了nan 2.0.9），网址为<https://github.com/51Degrees/51degrees.node>。若读者对51degrees.node或笔者其他代码库的nan2.x升级代码感兴趣，则可访问<https://github.com/51Degrees/51degrees.node/commit/93e7becee37f96780c4adeb38f20cafe335509d7>或<https://github.com/Southern/node-x509/commit/96ed4661619320f5237c254b591eb83179fec81f>。

7.6 后记

就笔者个人的工作经验来说，是应当尽量避免使用C/C++模组的，因为有时往往会因为一点点的性能提升，而牺牲掉跨平台的优势，甚至有时只是对Node.js进行版本的切换，都需要重新编译相关的C/C++模组，得不偿失。

不过，笔者很想通过这个项目让大家了解一个Node.js开源模组是如何产生的，由于篇幅有限，没有办法将单元测试及基准测试一一进行详细说明，但还是希望大家借助51Degrees.node能有以下收获：

（1）了解v8及Node的C/C++部分编程接口；

（2）了解nan的部分编程接口；

（3）了解基本的gyp写法；

（4）在C/C++模组中，需要权衡C/C++代码与JavaScript代码的分配方式，不同的比例与粒度可能会带来项目不同的敏捷性。