

Web开发技术丛书

Node.js实战

Learning Node.js:A Hands-On Guide to Building Web Applications in JavaScript

(美) 温施耐德 (Wandschneider,M.) 著

姚立 彭森林 译

ISBN : 978-7-111-45969-9

本书纸版由机械工业出版社于2014年出版，电子版由华章分社（北京华章图文信息有限公司）全球范围内制作与发行。

版权所有，侵权必究

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.hzmedia.com.cn

新浪微博 @研发书局

腾讯微博 @yanfabook

目 录

译者序

前言

第一部分 基础篇

第1章 入门

1.1 安装Node.js

1.2 "Hello World!"

1.3 第一个Web服务器

1.4 调试Node.js程序

1.5 保持最新及获取帮助

1.6 小结

第2章 进一步了解JavaScript

2.1 数据类型

2.2 类型比较和转换

2.3 函数

2.4 语言结构

2.5 类、原型和继承

2.6 错误和异常

2.7 几个重要的Node.js全局对象

2.8 小结

第3章 异步编程

3.1 传统编程方式

3.2 Node.js的编程方式

3.3 错误处理和异步函数

3.4 我是谁——如何维护本体

3.5 保持优雅——学会放弃控制权

3.6 同步函数调用

3.7 小结

第二部分 提高篇

第4章 编写简单应用

4.1 第一个JSON服务器

- 4.2 Node模式：异步循环
- 4.3 小戏法：处理更多的请求
- 4.4 请求和响应对象的更多细节
- 4.5 提高灵活性：GET参数
- 4.6 修改内容：POST数据
- 4.7 小结

第5章 模块化

- 5.1 编写简单模块
- 5.2 npm : Node包管理器
- 5.3 使用模块
- 5.4 编写模块
- 5.5 应当内置的通用模块
- 5.6 小结

第6章 扩展Web服务器

- 6.1 使用Stream处理静态内容
- 6.2 在客户端组装内容：模板
- 6.3 小结

第三部分 实战篇

第7章 使用express构建Web应用

- 7.1 安装express
- 7.2 express中的路由和分层
- 7.3 REST API设计和模块
- 7.4 中间件功能
- 7.5 小结

第8章 数据库I : NoSQL (MongoDB)

- 8.1 设置MongoDB
- 8.2 MongoDB数据结构
- 8.3 理解基本操作
- 8.4 更新相册应用
- 8.5 应用结构回顾
- 8.6 小结

第9章 数据库II : SQL (MySQL)

- 9.1 准备工作
- 9.2 创建数据库模式
- 9.3 基本数据库操作
- 9.4 添加应用身份验证
- 9.5 资源池
- 9.6 验证API
- 9.7 小结

第四部分 进阶篇

第10章 部署和开发

- 10.1 部署
- 10.2 多处理器部署：使用代理
- 10.3 虚拟主机
- 10.4 使用HTTPS/SSL保障项目安全
- 10.5 多平台开发
- 10.6 小结

第11章 命令行编程

- 11.1 运行命令行脚本
- 11.2 同步处理文件
- 11.3 用户交互：标准输入和输出
- 11.4 进程处理
- 11.5 小结

第12章 测试

- 12.1 测试框架选择
- 12.2 编写测试用例
- 12.3 RESTful API测试
- 12.4 小结

译者序

从1995年诞生至今，JavaScript已经走过了18个年头，很难想象，当初Brendan Eich在10天内写出来的语言竟然会成为如今最热门的语言之一。

这些年，JavaScript一直被当做Web浏览器端脚本使用，但在这个过程中，ECMA标准也在不断地演变，不断地加入新的特性；同时，各个厂商的JavaScript实现（特别是Chrome的V8）也在不断地提升性能和稳定性，这为后来Node.js的流行埋下了伏笔。

2009年，Node.js出现在大家的视野中，它在诞生伊始就获得了众多开发者的关注。由于使用了性能优越的Chrome浏览器V8引擎和全新的事件驱动、异步编程模式，它从一开始就显得“与众不同”。Node.js是Ryan Dahl发起的开源项目，这些新特性给它带来巨大的性能提升，并且大量的第三方模块也让其开发效率得到了极大提高。如今，诸多大型互联网公司在其产品中广泛使用Node.js，如LinkedIn、淘宝等；同时大部分云计算平台也支持部署Node.js应用。由此足以看出，Node.js是今后的发展趋势之一。

本书一共分为四大部分，由易入难，循序渐进，既适合Node.js新手入门，也能让使用过Node.js的开发者在阅读过程中收获“惊喜”。如果你是一名初学者，可以跟着书中的示例一步步成长；如果你是一名有经验的开发者，也可以从后面的章节中获取灵感。本书和其他介绍Node.js的书籍的最大不同在于，书中会包含大量的后端知识，如shell脚本等，这会让读者（特别是Web前端开发者）脱离原先的思维桎梏，从后端开发的角度来学习这个全新的开发平台。

本书由姚立、彭森材合作翻译完成。其中，彭森材翻译第2章、第4~6章、第9章和第10章；姚立翻译其余部分。在翻译本书的过程中得到了华章公司责任编辑关敏老师的帮助，她多方协调并给出中肯的建议，让译者收获颇丰，在此表示衷心感谢。

由于时间仓促，加之译者水平有限，书中难免会有疏漏之处，希

望读者能够提供反馈，不胜感激。

最后，我们要感谢家人，感谢他们的理解和支持，感谢他们在本书翻译过程中所做的一切。一条毛毯、一杯热茶，都是我们翻译的动力和源泉。感激、感恩！

前言

欢迎来到Node.js的世界。Node.js是专为编写网络和Web应用而生的全新平台，在过去的几年中，迅速积累了大量的人气并在开发社区拥有一大批拥趸。在本书中，我会介绍更多关于Node.js的知识，告诉你为什么它会如此迷人。我会教你如何快速上手并编写Node.js程序。很快，你会发现Node.js名字的叫法非常灵活，它经常被叫做Node，甚至被称为"node"。本书中我也会经常这么称呼。

为什么使用Node.js

Node.js的兴起有两个主要原因，我会在下面加以解释。

万维网

在过去，编写Web应用是一个相当标准化的流程。你有一个或多个服务器部署在机器上并监听某个端口（如HTTP的80端口），每当服务器接收到一个请求时，它会创建一个进程或者线程去处理和响应请求。处理这些请求会频繁地与外部服务进行通信，比如数据库、内存缓存、外部计算服务器或者仅仅是文件系统。当所有的工作最终结束之后，线程或者进程会返回到“可用”服务器池中，这样服务器就可以处理更多的请求了。

这是一个合理的线性过程，容易理解且极易编码。但是，有两个缺点一直困扰着这种模型：

1.每一个线程或进程都会消耗一些资源。在一些机器中，对于使用PHP和Apache构建的应用，每个进程甚至会消耗高达10~15MB的内存。特别是在大型服务器不断运行并创建线程来处理请求的时候，每一个线程都会消耗一些资源来创建新的堆栈和执行环境。很快，服务器就会陷入没有内存可用的境地。

2.在最常见的使用场景下，Web服务器会与数据库、缓存服务器、外部服务器或文件系统通信，这会消耗大量的等待时间，直到这些服务处理完并返回响应。当服务器等待响应的时候，当前线程就会

被“阻塞”，无法继续执行下去。因此，消耗的服务器资源和当前服务器进程或线程会被完全冻结，直到返回响应相关资源才会得到释放。

只有当外部组件返回响应之后，进程或线程才会继续完成处理，并把结果返回给客户端，然后重置并等待处理下一个请求。

因此，尽管这种模型非常容易理解和编码，但是一旦脚本花费大量的时间来等待数据库服务器结束查询，这种模型就略显低效——而这却是现代Web应用极其常见的应用场景。

当然，这种问题已经有了很多通用的解决方案。我们可以购买更强大、拥有更多内存的Web服务器；可以使用更小的轻量级服务器如lighttpd或者nginx来替代Apache这样功能强大的HTTP服务器；可以定制或者精简你喜欢的编程语言版本，比如PHP或Python（事实上，Facebook已经先行一步，并且开发出将PHP转换成原生C++代码的系统，以达到最大执行速度和最优代码规模的目的）；甚至，可以通过购置更多服务器的方式来提高所能容纳的并发连接数。

前沿技术

多年来，Web开发者一直在为优化服务器资源和提高并发数而努力；而这期间，其他一些有趣的事情已经悄然发生。

JavaScript，这门在Web浏览器中以客户端脚本语言而闻名于世（同时也经常遭人诟病）的古老（1995年左右面世）语言，已经重新焕发生机。诸多现代Web浏览器重构了JavaScript的实现，并添加了许多新的特性，让它变得更加强大和稳定。随着诸多浏览器客户端类库的出现，如jQuery、script.aculo.us、Prototype等，JavaScript编程变得有趣且富有成效。这些类库封装了原先臃肿的API，并添加了很多有趣、动态的效果。

同时，各大浏览器群雄割据的时代业已到来，谷歌的Chrome、Mozilla的Firefox、苹果的Safari和微软的IE浏览器一起争夺浏览器之王的桂冠。作为其中的一部分，这些公司加大了在JavaScript上的

研发比重，从而让Web应用可以更可靠地依赖脚本，并且拥有更强大的动态效果。特别是谷歌公司的Chrome浏览器使用的V8JavaScript引擎，性能尤为卓越，并且是开源软件，可供任何人使用。

天时地利人和，JavaScript的机遇随着开发人员的全新网络（Web）应用开发方式而出现。这，就是Node.js的新生。

Node.js的前世今生

2009年，一位名叫Ryan Dahl的研究员（后来他加入Joyent，该公司是一家坐落于加利福尼亚州的云计算和可视化服务公司）一直在寻求开发一种专为Web应用提供类似于Gmail推送服务的功能，但是一直没有找到一个完全合适的解决方案。最后，他把目光落到了JavaScript上，因为这门语言缺乏健壮的输入/输出（I/O）模型（这也就意味着他可以自己写一个全新的I/O系统），同时还有性能优越的V8引擎可供编程使用。

受Ruby和Python社区中类似项目的启发，他最终选择了Chrome的V8引擎和一个叫做libev的事件处理类库，并且很快推出了第一版的Node.js系统。Node.js的主要理论和创新就是它完全构建在事件驱动、非阻塞的编程模型之上。简而言之，你完全不会（或者极少）编写线程阻塞的代码。

一个Web应用——为了处理请求并返回响应——一般需要执行数据库查询。Web应用处理请求，并会在返回响应之后告诉Node.js如何处理。同时，应用代码也可以开始处理其他传入的请求，实际上，它可以处理任何需要处理的任务，比如清理系统数据或分析数据。

通过这种应用处理请求和工作方式的改变，可以简单地编写能够同时处理几百甚至几千个请求的Web服务器，同时也不用消耗太多的资源。Node是在单进程上运行的，并且Node代码一般也是单线程执行，因此，相较于其他平台而言，Node对资源的需求就小得多。

但是，这种执行速度和能力是有一些瑕疵的。因此，必须充分认识到它们，然后就可以开始小心翼翼地使用Node进行工作了。

首先，这种全新的模型机制不同于以往你所接触过的模型，因此你可能会感到困惑。在充分理解这些核心概念之前，你可能会对这些Node.js代码感到奇怪。本书中的大部分篇幅会讨论编程人员在迎接Node异步、非阻塞编程挑战时所使用的核心模式，并指导你如何开发自己的Node程序。

这种编程模型的另一个限制是：它真的是在为处理大量不同任务的应用程序服务，会同很多不同的进程、服务器或服务通信。当Web应用需要连接到数据库、缓存服务器、文件系统、应用服务器或其他服务时，Node.js便会大放异彩。但是另一方面，实际上它并不是那些需要做长时间精密计算的服务器的最佳运行环境。因此，单进程、单线程的Node模型在处理一个给定的请求时，如果该请求需要花费大量的时间生成一个复杂的密码摘要（password digest）或处理图像，就会带来问题。在那些需要更多的计算密集型工作的情况下，我们必须小心谨慎地处理应用程序使用资源的情况，甚至可以考虑把这些任务移植到其他平台，并把它们作为第三方服务供Node.js程序调用。

最后，Node.js是一个值得信赖的全新平台，并且正在积极发展中。它还没有（截至2013年2月）发布1.0版本，同时Node.js的版本更新非常频繁，甚至到了让人眼花缭乱的地步。

为了减少这些频繁的更新所引起的不确定性和烦恼，开发者已经使用标签来标识系统各个部分的不同稳定级别，从不稳定（Unstable）到稳定（Stable）再到锁定（Locked）。系统中稳定或锁定的部分几乎不会改动；如需改动，社区会通过大量的讨论以确定是否会产生太多问题。在你阅读本书的过程中，我们会指出哪些地方不太稳定，并提供相应的解决方法，从而减少因API改变而带来的危险。

好消息是，Node.js已经有一个庞大而活跃的用户社区和一堆邮

件列表、论坛及用户组，它们致力于促进平台的发展并提供力所能及的帮助。通过简单的谷歌搜索，就可以在几秒钟之内回答你99%的问题，所以，请不要害怕Node.js！

本书读者

本书的读者要喜欢编程，并且要熟悉至少一门主流编程语言的功能和语法，比如Java、C/C++、PHP或者C#等。读完本书之后，你不必成为一名专家，但你一定要高于“Y天精通X语言”的水平。

如果像我一样，你可能已经写过一些HTML/CSS/JavaScript代码，从而和JavaScript“打过交道”。但是你可能并不是非常熟悉JavaScript，只是单纯地从博客或者邮件列表中把代码复制出来。事实上，由于笨重的用户界面和令人沮丧的浏览器不匹配问题，你可能在一开始就对JavaScript没有好感。但是不用害怕——在本书第一部分结束之后，你对这门语言的糟糕印象从此会变成过去时。我希望你能放松心情，面露微笑，轻松愉快地编写你的第一个Node.js程序。

你还需要了解Web应用的基本工作原理：浏览器向远程服务器发送HTTP请求；服务器处理请求并返回表明成功或者失败的标识，返回对象中有可能会附带一些数据（比如渲染页面用的HTML代码或者请求返回的JSON数据）。在过去，你可能会在连接数据库服务器并发送查询请求后，等待返回数据集等。而在使用示例和程序来描述一些全新的概念的时候，我会详细讲解并帮助大家理解这些全新或者生僻的概念。

如何使用本书

本书实质上就是一本新手教程。我会尽量使用代码去解释原理，避免使用冗长的篇幅和晦涩的语言。因为我觉得好的解释说明应该是生动有趣的，如果你感兴趣的话，我还会告诉你一些资源或者一些其他的文档（当然这些并不是必需的）。

本书一共分为四大部分。

第一部分 基础篇——首先，你会学习如何安装并运行Node；

然后，会进一步了解JavaScript这门语言以及在V8和Node.js中使用的扩展；最后，会编写你的第一个Node.js应用。

第二部分 提高篇——在本篇中，你会开始学习开发更强大且有趣的应用服务器。同时，我会教你编写Node.js程序时会用到的一些核心概念和最佳实践。

第三部分 实战篇——在本篇中，你会看到一些强大的工具和模块，利用它们，你可以编写自己的Web应用，比如Web服务器的中间件及与数据库服务器的通信。

第四部分 进阶篇——最后，我会以一些高级特性的介绍结束本书，比如如何在生产环境中运行应用，如何测试代码以及如何使用Node.js编写命令行实用程序。

在阅读本书的过程中，最好花些时间打开编辑器，输入这些代码，然后看看这些代码在你当前的Node.js版本下是如何运行的。或者是在看书的时候，编写并开发你自己的小程序。在阅读本书的过程中，你会开发一个小小的相册应用，希望能给你编写其他应用提供一些灵感和想法。

[下载源代码](#)

本书示例的源代码可以在github.com/marcwan/LearningNodeJS上找到。希望你能下载代码并运行一遍。如果条件允许，不要放弃亲手把代码敲出来的机会，可以多尝试尝试。

GitHub上托管了功能齐全的示例代码，并且已经使用最新版的Node.js，在Mac、Linux和Windows下测试过。如果最新版的Node需要更新源代码，我会在GitHub上及时更新并做出注释，所以请确保每隔几个月就获取一次最新的代码。

如果你对本书中的示例代码有任何疑问，请在github.com/marcwan/LearningNodeJS上提交issue。我们会看到这些issue，并且会及时做出合理的解答。

致谢

我要感谢PHPTR各位朋友给予的支持和建议，是你们的帮助让本书和其他项目付诸实践。感谢本书文字编辑的杰出贡献。

非常感谢Bill Glover和Idriss Juhoor，是你们出色的技术和样式审校让本书愈加精彩。

最后，感谢至爱Tina，感谢一路有你。

第一部分 基础篇

第1章 入门

第2章 进一步了解JavaScript

第3章 异步编程

第1章 入门

在本章中，我们会开始投入到相关的学习中，并在电脑上安装Node.js。在继续深入学习语言和编写网络应用之前，要确保Node.js能正常运行。本章的最后，应该已经成功地在电脑上安装Node.js并正常运行。我们还会使用一些小的测试程序来熟悉它，以及学会如何使用内置的Node调试器。

1.1 安装Node.js

首先，让我们来看看如何在Windows下安装Node。除非同时拥有Windows操作系统，否则Mac和Linux用户可以跳过本节，去阅读相应的章节。

1.1.1 在Windows上安装

要想在Windows电脑上安装Node.js，可以使用nodejs.org网站上提供的简易安装程序。可以访问下载页面，然后选择32位或者64位的Node.js安装程序（.msi）。当然，这完全取决于运行Node的操作系统。我们将会展示Windows 7/64位操作系统下Node.js的安装过程。

下载完MSI文件之后，双击该文件，将会看到如图1.1所示的安装程序界面。

阅读并同意授权协议之后，点击安装（Install）。安装过程非常快捷和方便，几十秒之后，点击完成（Finish）结束安装。

验证安装

为了测试Node.js是否正确安装，你可以使用Windows命令提示符cmd.exe（如果使用PowerShell，也是可行的）。如果你对此不熟悉的话，可以先找到开始（Start）/运行（Run），然后输入cmd，如图1.2所示。



图1.1 Windows下的Node安装程序

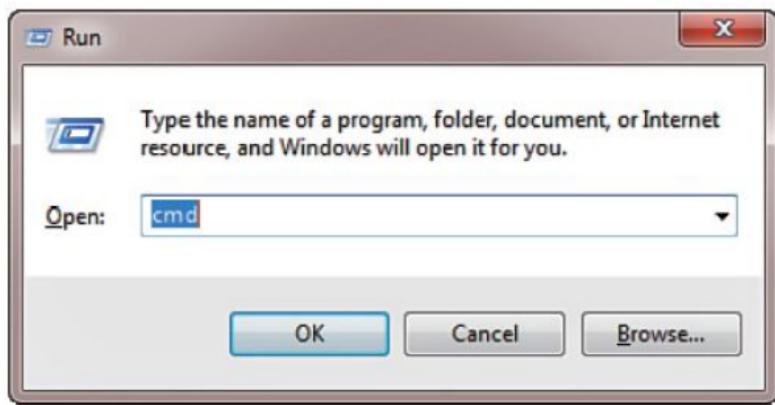


图1.2 启动Windows命令提示符

如图1.3所示，你会看到一个命令解释器。如果你想学习更多关于命令提示符的知识，可以在互联网上通过搜索“学习使用Windows cmd.exe”或者“PowerShell入门”（如果你使用的是Windows 7操作系统）来找到更多信息。

为了确认Node已经正确安装，在命令窗口中输入node--version，可以看到如图1.4所示的输出。

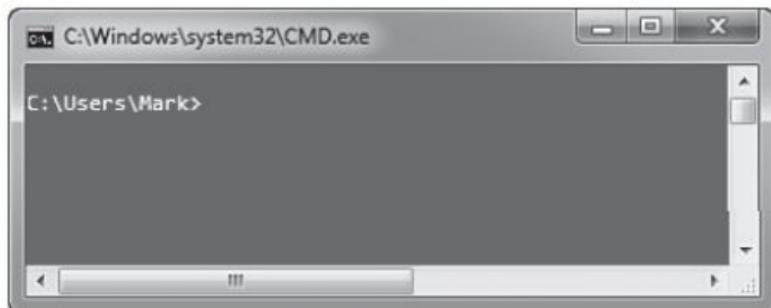


图1.3 Windows命令提示符

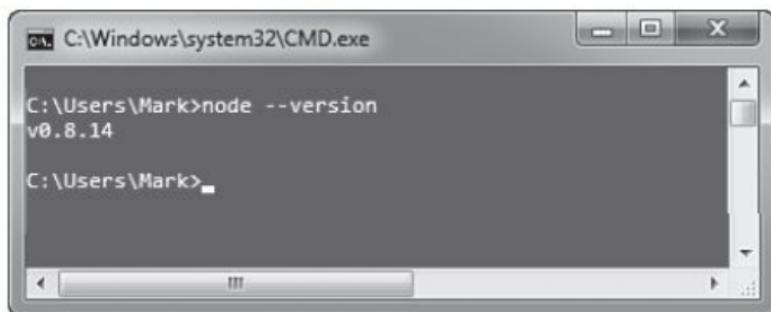
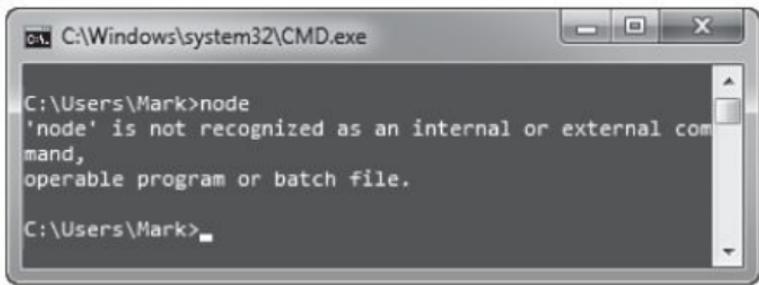


图1.4 验证Node是否正确安装，检测版本号

命令提示符窗口会打印出刚刚安装完的Node的版本号。（如果图中版本号和你看到的不一致，不用担心——事实上，如果一样的话，才会让我大吃一惊呢！）如果你没有看到版本号或者你看到的输出是“'node'不是外部或内部命令”（见图1.5），那肯定是出问题了，你需要进行如下操作。

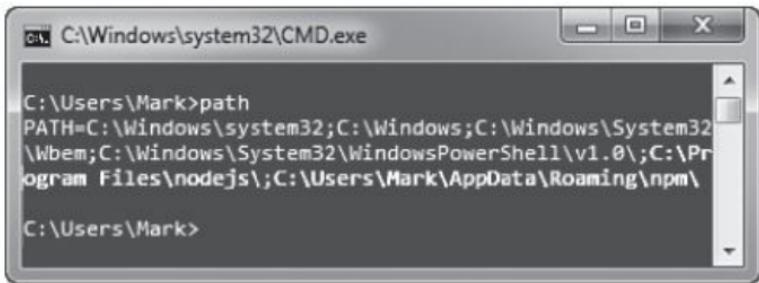
- 进入控制面板/程序中，查看程序有没有安装成功。如果没有，则再次安装，并且注意安装过程，有可能会出现安装出错。
- 进入Program Files\nodejs目录，确保node.exe文件存在。如果不存在，尝试再次安装（如有需要，请先卸载旧版本）。
- 确保node.exe在PATH环境变量中。在命令提示符中可以看到，Windows有一个目录列表，当输入程序名称时，会在该列表中进行搜索。只要在命令提示符窗口中输入path就可以看到该列表，类似图1.6所示。尤其要仔细看后面两个高亮显示的目录名，如果Node.js安装正确的话，就会在PATH中看到相似的目录名。



```
C:\Windows\system32\cmd.exe
C:\Users\Mark>node
'node' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\Mark>
```

图1.5 Node不是内部或外部命令



```
C:\Windows\system32\cmd.exe
C:\Users\Mark>path
PATH=C:\Windows\system32;C:\Windows;C:\Windows\System32
\WBem;C:\Windows\System32\WindowsPowerShell\v1.0\;C:\Pr
ogram Files\nodejs\;C:\Users\Mark\AppData\Roaming\npm\

C:\Users\Mark>
```

图1.6 检测PATH环境变量

■ 如果Program Files\nodejs或者Users..\AppData..\npm不在环境变量PATH中，而这些文件夹却真实存在，你可以手动将它们添加到PATH环境变量中，可以在系统控制面板窗口中设置它。点击高级系统设置，然后是环境变量。将npm文件夹的路径（类似于C:\Users\UserName\Local\npm）添加到“用户名”的用户变量下的PATH变量中。将Node.js文件夹（类似于C:\Program Files\nodejs）添加到系统变量下的PATH变量中。注意：npm文件夹可能会在Username\Remote\npm路径下，而不是Username\Local\npm下，这完全取决于你的电脑是如何设置的。

在确认node.exe正确安装和运行之后，就可以开始你的JavaScript之旅了。

1.1.2 在Mac上安装

在Mac上安装Node.js有两种不同的方式——使用PKG安装程序或者编译源代码——这里，我只想介绍前一种安装方式，这是目前最快最便捷的安装方式。如果你倾向于以编译源代码的方式安装Node.js，我已经在本书的git资源中准备了安装指南。

使用PKG安装程序

目前为止，在运行OS X的苹果Mac电脑中安装Node.js最快的方式就是下载并运行Node PKG安装程序，该安装程序文件在nodejs.org网站中提供下载。

下载完安装程序以后，双击程序，你会看到如图1.7所示的安装界面。我倾向于使用默认安装，因为我想使用所有的组件，并且这个默认安装路径（/usr/local/bin）也正是我想要的，建议你也这样做。

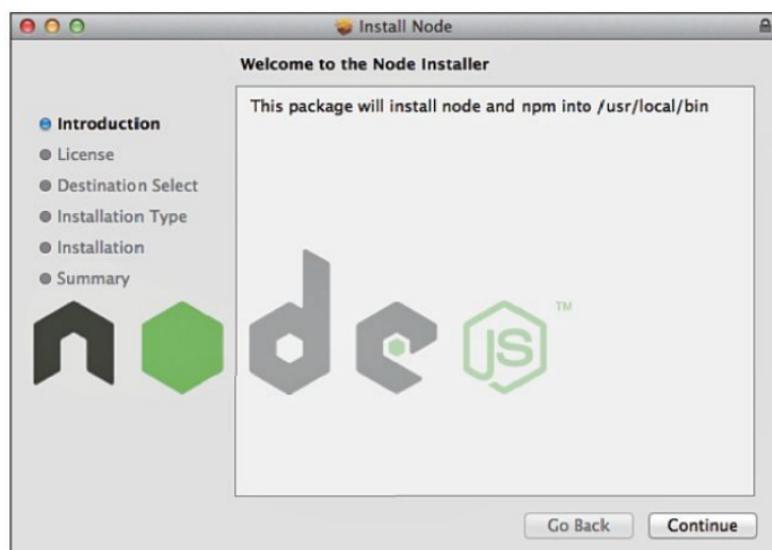


图1.7 在Mac上运行Node.js的PKG安装程序

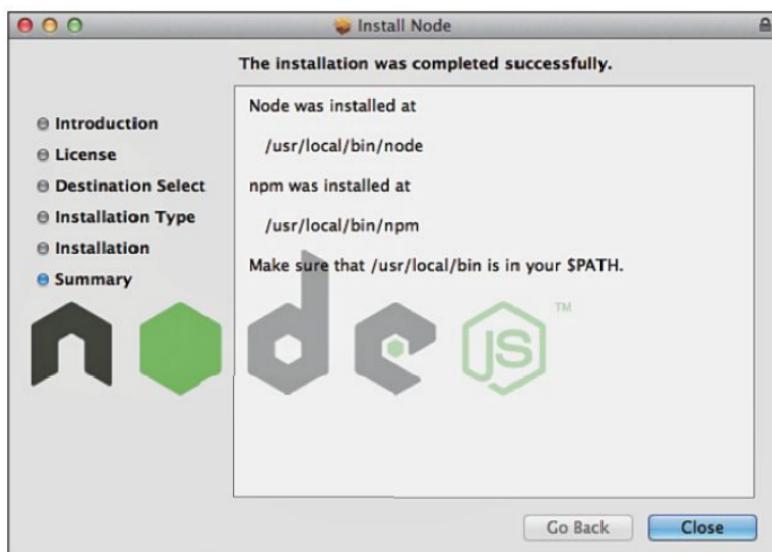


图1.8 确认path环境变量正确设置

当安装结束之后，可以看到如图1.8所示的界面，正如图中包安装程序解释的那样，一定要确认/usr/local/bin在PATH环境变量中。你可以打开Terminal.app程序（在/Applications/Utilities下启动终端），在终端窗口中，输入：

```
echo $PATH
```

在我的电脑中，输出如下：

```
/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/Users/marcw/bin:/usr/local/git/bin
```

你可以看到/usr/local/bin就在PATH环境变量中；如果没有，就需要编辑~/.bash_profile文件（如果不存在，可以创建该文件），添加下面这段脚本：

```
PATH=${PATH}:/usr/local/bin
```

关闭该终端窗口，并重新启动一个新的终端窗口，验证/usr/local/bin已经在PATH环境变量中，现在可以输入：

```
node --version
```

可看到如下所示内容：

```
client:LearningNode marcw$ node --version
v0.10.3
client:LearningNode marcw$
```

如果版本号和你电脑上的版本号不一致，请不用担心，这是正常情况。

在确认node程序已经正确安装并运行之后，就可以编写JavaScript代码了。

1.1.3 在Linux上安装

安装Node.js

尽管许多的Linux发行版已经预置了Node.js安装包，但我还是更喜欢手动在Linux上安装Node。安装过程非常简单，所以无需过多额外的工作。

在Linux上安装命令行编译器

要想在Linux上编译源码包，首先要确保已经安装了命令行编译器工具。要安装编译器工具，可以输入：

```
g++
```

如果在终端窗口中看到如下所示：

```
marcw@hostname:~$ g++  
g++: no input files
```

则编译器已经成功安装，可以编译源码了。否则，你会看到：

```
marcw@hostname:~$ g++  
-bash: g++: command not found
```

你需要清楚如何在你当前的Linux发行版上安装编译工具。在绝大部分Ubuntu Linux版本中，可以如下所示使用apt-get工具。如果要安装编译工具，则需要知道在某个特定版本下应该安装哪些安装包。对于Ubuntu 8，必须安装：

```
# apt-get install build-essential libssl-dev libxml2-dev autoconf2.13
```

而在Ubuntu 10上，则需要安装：

```
# apt-get install pentium-builder g++ libssl-dev libxml2-dev autoconf2.13
```

当所有安装结束之后，可以再次输入：

```
g++
```

一切运行正常。

下面的安装指南已经在Ubuntu Linux版本中使用多年，前提是
你使用 (ba) sh作为默认的shell解释器。首先，创建一个暂存空间
用来下载和编译文件：

```
cd  
mkdir -p src/scratch  
cd src/scratch
```

下一步就是下载并解压node源代码，可以使用curl或者wget命
令：

```
curl http://nodejs.org/dist/v0.10.1/node-v0.10.1.tar.gz -o node-v0.10.1.tar.gz  
tar xfz node-v0.10.1.tar.gz  
cd node-v0.10.1
```

接下来运行配置脚本来为编译做准备：

```
./configure
```

可以使用默认的/usr/local作为安装目录，因为这是运行这个软
件的最佳位置。如果想把软件安装到其他目录下，可以指定--prefix
来修改配置脚本，如下所示：

```
./configure --prefix =/opt/nodejs
```

这个配置脚本会执行得非常快，并会在结束的时候打印出一些
JSON信息。现在，你可以编译Node了。输入下面的命令，然后喝
一杯咖啡，静静等待（也许是两杯哦，速度快慢完全取决于电脑的性
能）。

```
make
```

在成功编译完成之后（如果编译失败，可以谷歌搜索下这个问
题，因为一般情况下，不只你一个人遇到相同的问题），可以将软件
安装到指定的目录下（如果没有指定，则默认目录是/usr/local）：

```
sudo make install
```

当完成这一切，输入：

```
node --version  
npm --version
```

可以得到如下输出结果：

```
marcw@hostname:~$ node --version  
v0.10.1  
marcw@hostname:~$ npm --version  
1.1.65
```

1.2 "Hello World!"

在电脑上使用Node.js有两种主要方式：直接使用Node Shell或者保存JavaScript文件后运行。

1.2.1 Node Shell

运行Node.js的第一种方式就是使用Node Shell，这种方式一般会被称为Node REPL——REPL是Read-Eval-Print-Loop的缩写。这是一种快速测试Node的途径。如果你不能准确记住某个函数的用法，可以使用REPL，把东西输入进去，看看会发生什么。

如果想启动Node Shell，可以在任何shell中输入node：

```
client:node marcw$ node  
>
```

Shell会返回>符号，然后你就可以输入一些代码了：

```
> console.log("Hello World!");  
Hello World!  
undefined  
>
```

第一行输出就是刚才输入的代码执行之后的结果。在上述示例中，我们使用了Node的全局变量console，并使用log函数打印出"Hello World!"（下一章中会介绍更多关于console和其他全局变量的内容）。该语句的期望结果是，打印出"Hello World!"。

最后一行输出结果往往是最后一行语句的返回值。每一个语句、函数调用或者表达式都有一个相关联的值，这个值会在Node shell中打印出来。如果表达式或者被调用的函数没有任何返回值，则会返回一个特殊的值undefined。

如果想要退出REPL，只需要按下Ctrl+D（在Windows平台下也一样）。

如果你在Node REPL中看到三个点（...），这就意味着你需要输入更多的代码去完成当前的表达式、语句或者函数。如果你实在不明白它为什么会提示省略号，可以输入`.break`（包括前面的那个点）来消除当前的省略号：

```
> function () {  
... }  
... what?  
... .break  
>
```

1.2.2 编辑并运行JavaScript文件

另一种运行Node.js的方式就是：可以选择自己喜欢的文本编辑器，然后把JavaScript代码写到文件中，最后在命令行中使用`node`命令编译并执行代码。

为了演示这一点，把下面的代码保存为`hello.js`文件：

```
/**  
 * Let's be sociable.  
 */  
console.log("Hello World!");
```

现在，你可以使用命令行执行该文件：

```
node hello.js
```

可以看到如下的输出结果：

```
Hello World!
```

因为没有使用Node shell，所以不会返回任何代码执行返回值的相关信息。

1.3 第一个Web服务器

你现在一定打算写一些更有趣的代码，并准备写一个小小的Web服务器。幸运的是，Node让这一切变得易如反掌。输入并保存代码到web.js文件中：

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

function process_request(req, res) {
    var body = 'Thanks for calling!\n';
    var content_length = body.length;
    res.writeHead(200, {
        'Content-Length': content_length,
        'Content-Type': 'text/plain'
    });
    res.end(body);
}

var s = http.createServer(process_request);
s.listen(8080);
```

要运行这个文件，需要输入：

```
node web.js
```

现在我们已经在电脑的8080端口上运行了一个Web服务器，可以使用命令行程序curl来测试它，这个命令在大部分Mac和Linux电脑上都是预置的（Windows用户请阅读下面的“Windows下如何下载Web资源”。也可以在浏览器中输入http://localhost:8080，但是，只有打开调试控制台，才会看到返回的响应代码）。

```
curl -i http://localhost:8080
```

现在，可以看到如下内容：

```
HTTP/1.1 200 OK
Content-Length: 20
Content-Type: text/plain
Date: Tue, 15 Feb 2013 03:05:08 GMT
Connection: keep-alive
```

Thanks for calling!

Windows下如何下载Web资源

默认情况下，Windows并没有提供可以从统一资源定位符（Uniform Resource Locator，URL）获取内容的命令行工具。但是，使用命令行工具下载非常有趣，因此我强烈推荐使用Windows版的curl（从现在起，我会称之为curl）或者wget。

Curl：

可以访问<http://curl.haxx.se/download.html>，找到"Win32-Generic"区域，下载Windows二进制安装文件。

只需下载高亮的二进制文件中的一个，最好选择支持SSL和SSH功能的版本（如果跳转到另一个页面，点击下载"Download WITH SUPPORT SSL"），解压文件，将curl.exe放到PATH路径下或者用户目录下。要启动它，只需在命令提示符或者PowerShell中输入：

```
C:\Users\Mark\curl --help
```

Wget：

如果curl不能在你的Windows电脑上使用，那么wget是一个不错的替代品。可以从<http://users.ugent.be/~bpuype/wget/>上下载。

Wget的工作原理和curl差不多，但是命令行参数有所不同。要了解更多信息，可以查看帮助：

```
C:\Users\Mark\wget --help
```

Node.js本身提供了许多强大的功能，前面程序的第一行使用了内置模块之一——http模块，它允许程序创建Web服务器。可以使用require函数引用该模块，从而拥有http的引用变量。

createServer函数只会接受一个函数参数，它会在用户连接到服务器时被调用。前面写的process_request函数会被作为参数传递进去，该函数会被赋予一个请求对象（即ServerRequest^[1]实例）和一个响应对象（即ServerReponse实例）。服务器创建之后，会指定在某个特定的端口上监听传入的请求——这里，启动该程序时，使用了8080端口。

前面将-i参数传给curl命令，会将请求内容和响应的头信息一起打印出来。这样便于我们更加理解Node的工作原理。

从返回的信息中可以看到，ServerResponse#writeHead函数返回的200（OK）响应就在HTTP响应头里面，同时还包含了内容长度（Content-Length）和类型（Content-Type）。默认情况下，Node.js会将服务器的HTTP连接设置为keep-alive，表明可以在同一个连接中发送多个请求，当然，本书的大部分示例中都用不到该功能。

要停止运行的服务器，只需要简单地按下Ctrl+C即可。它会自动清理系统资源并停止服务。

[1]最新版本的API doc中已替换成IncomingMessage类。——译者注

1.4 调试Node.js程序

现在，重写前面的Web服务器，但是这次可能会由于粗心而引入一个小错误——body.length的拼写错误，将代码保存到debugging.js文件中，如下所示：

```
var http = require("http");
function process_request(req, res) {
    var body = 'Thanks for calling!\n';
    var content_length = body.length;
    res.writeHead(200, {
        'Content-Length': content_length,
        'Content-Type': 'text/plain'
    });
    res.end(body);
}

var s = http.createServer(process_request);
s.listen(8080);
```

再次运行该程序：

```
node debugging.js
```

但是，当尝试连接到http://localhost:8080的时候，可能会得到：

```
client:- marcw$ curl -i localhost:8080
HTTP/1.1 200 OK
Content-Length: undefined
Content-Type: text/plain
Date: Tue, 30 Oct 2012 04:42:44 GMT
Connection: keep-alive
```

上面并没有得到"Thanks for calling!"这样的信息，而且响应头Content-Length也不是预期的结果。

像这样的小程序中，这些错误会很容易被找到，但在一些大型程

序中，将很难定位具体的错误信息。为了解决这个问题，Node.js内置了一个调试器。如果想使用，只需要在启动的时候将debug参数添加到程序名称的前面即可：

```
node debug debugging.js
```

可以得到如下所示的信息：

```
client:Chapter01 marcw$ node debug debugging.js
< debugger listening on port 5858
connecting... ok
break in debugging.js:1
1 var http = require("http");
2
3 function process_request(req, res) {
debug>
```

Node调试器中有一些关键的命令可供使用：

- **cont**——继续执行。
- **next**——跳到下一个语句。
- **step**——进入当前执行函数中的语句（如果是函数的话；否则，跳过）。
- **out**——跳出当前执行函数。
- **backtrace**——显示当前调用执行帧或调用栈。
- **repl**——启动Node REPL，允许查看变量值和执行代码。
- **watch(expr)**——向观察列表中添加表达式，这样在调试器中进入函数或者移动时会显示出来。
- **list(n)**——列出调试器中当前停止行的前面和后面的n行代码。

现在，假设程序中的Content-Length可能出错，则需要在行
var content_length=body.length;上添加断点，即第5行：

```
debug> setBreakpoint(5)
 1 var http = require("http");
 2
 3 function process_request(req, res) {
 4     var body = 'Thanks for calling!\n';
* 5     var content_length = body.length;
 6     res.writeHead(200, {
debug>
```

现在第5行已经多了一个星号（ * ），表明存在断点。当启动调试器时，程序会在第一行停下。这时，可以使用cont命令恢复执行：

```
debug> cont
debug>
```

打开另外一个终端窗口或者命令行提示符，输入：

```
curl -i http://localhost:8080
```

我们注意到发生了两件事：

1) curl命令并没有立即返回信息。

2) 在node debug进程中，可以看到：

```
break in debugging.js:5
 3 function process_request(req, res) {
 4     var body = 'Thanks for calling!\n';
* 5     var content_length = body.length;
 6     res.writeHead(200, {
 7         'Content-Length': content_length,
```

我们可以跳过这一行：

```
debug> next
break in debugging.js:7
* 5      var content_length = body.length;
  6      res.writeHead(200, {
  7          'Content-Length': content_length,
  8          'Content-Type': 'text/plain'
  9      });
```

现在，开启Node REPL，这样就可以检查一些变量值了：

```
debug> repl
Press Ctrl + C to leave debug repl
>
```

让我们分别看一下变量body和content_length的值：

```
> body
'Thanks for calling!\n'
> content_length
>
```

变量body的值与预期的一样。而变量content_length的值应该是20，但是却没有显示出来。由此可以得知给content_length赋值时出错，问题找到了！

最后，可以通过Ctrl+D关闭系统结束调试，或者输入cont继续运行服务器。在REPL中输入cont不会生效，会显示如下错误信息：'ReferenceError:cont is not defined'。因此，首先需要按下Ctrl+C退出REPL，然后使用cont命令。

尽管对调试器的介绍有些简短，但是绝对值得一试，因为它真的非常强大和实用。另外，还有一些Node.js社区成员编写的基于浏览器的调试器，其中最有前景的当属node-inspector。可以尽情尝试，看看它们是否对你的开发有帮助。

必要时，还可以使用最为简单的
console.log(variable_name);，将其添加到代码中，然后可以在终端窗口中打印结果。它是获取信息最快最便捷的方法，可以有效定位和追溯错误或问题。

1.5 保持最新及获取帮助

正如前文所提到的那样，使用Node.js的挑战之一就是它不断地更新变化。尽管越来越多的API被标记为稳定或者锁定，但每一个新版本都会有些新变化，而几乎每周都会发布新版本。

通过以下方式，可以保持最新并且不会错过任何重要的变更和消息：

- 加入Node.js的邮件列表

<http://groups.google.com/group/nodejs>。很多Node核心开发者都在该邮件列表中，并会在每一个新版本发布或变更的时候发布消息。

- 如果你拥有推特（Twitter）账号，可以关注@nodejs；当有新版本或者其他重要消息发布的时候，你会收到相应的推特消息。

- 经常访问nodejs.org，以保证了解最新的信息。

要想获取帮助，nodejs谷歌论坛（Google Groups）和node官网一样举足轻重。同样，StackOverflow.com也是一个非常活跃的社区，可以帮助解决Node相关的问题，你可以在那里找到许多非常棒的答案。

但是，我还发现，很多问题只需要通过简单的谷歌搜索就可以找到最好的答案。许多开发者经常会遇到类似的问题，并撰写一些博客和消息。因此，我一般都能通过简单的搜索找到理想的答案。

1.6 小结

至此，你已经成功地将Node.js安装到电脑上，并验证过能够正常运行，甚至已经运行并调试过一些问题了。那么，现在是时候进一步了解JavaScript这门语言了。

第2章 进一步了解JavaScript

如果你正在阅读本书，那么很可能你以前使用过JavaScript。也许你已经使用HTML、CSS及JavaScript编写过一个Web应用程序，通过直接操作浏览器文档对象模型（Document Object Model，DOM）或者使用框架（例如，使用jQuery、Prototype等来屏蔽一些混乱的细节）使客户端更加动态、更具交互性。可能你已经发现使用JavaScript工作是一件令人沮丧的事情，需要花费大量的时间处理不同浏览器的兼容性。而且很可能，你未曾从最基础的语言特性方面研究过JavaScript语言，那么，阅读本书将会让你受益匪浅。

好消息是现代Web浏览器正在逐步对JavaScript语言做一些必要的清理。此外，所有现代浏览器实现所遵守的规范——ECMAScript也一直在不断发展。Chrome V8 JavaScript引擎本身也在不断地改进和清理一些JavaScript语言糟粕，并添加许多被忽略的重要特性。

因此，即使你过去使用过JavaScript，本章仍然值得一读，你能够了解到一些被忽略的细节、一些新特性或者由V8和Node.js带来的改变。虽然本章的大部分内容适用于标准的JavaScript，但其中还是会经常展示一些由Google V8带来的新特性。对于非标准特性，我会使用（V8 JS）来标注。

2.1 数据类型

本节将会回顾一下JavaScript，看一下这门语言所提供的数据类型。对于本节中的大部分内容，我都会使用Node.js的Read-Eval-Print-Loop (REPL) 来演示代码是如何工作的。代码中，使用粗体字来标识需要读者输入到解释器中的代码。

```
client:LearningNode marcw$ node  
>
```

2.1.1 类型基础

Node.js的核心类型有：number（数字）、boolean（布尔值）、string（字符串）以及object（对象）。另外两种类型——函数（function）和数组（array）实际上是object的特殊形式。因为它们在语言以及运行时层面有一些额外的特性，因此将object、function（函数）以及array（数组）归类为复杂数据类型。null和undefined也是object的特殊形式，在JavaScript语言中有特殊作用。

undefined值代表还没有赋值或者不存在：

```
> var x;  
undefined  
> x = {};  
{}  
> x.not_valid;  
undefined  
>
```

null的另外一个准确的意思是“没有值”：

```
> var y;  
undefined  
> y  
undefined  
> y = null;  
null  
>
```

在JavaScript中，可以通过`typeof`操作符查看任何数据的类型：

```
> typeof 10  
'number'  
> typeof "hello";  
'string'  
> typeof function () { var x = 20; }  
'function'  
>
```

2.1.2 常量

虽然Node.js理论上支持`const`关键字，`const`关键字扩展在许多现代的JavaScript实现中也都被实现了，然而`const`关键字仍没有被广泛使用。对于常量，标准实践仍然是使用大写字母和变量声明：

```
> var SECONDS_PER_DAY = 86400;  
undefined  
> console.log(SECONDS_PER_DAY);  
86400  
undefined  
>
```

2.1.3 number类型

JavaScript中所有数字都采用IEEE 754标准定义的64位双精度浮点数表示。所有的正负整数都可以使用 2^{53} 位准确表示，JavaScript中的数字类型和其他语言的整数数据类型非常相似：

```
> 1024 * 1024
1048576
> 1048576
1048576
> 32437893250 + 3824598359235235
3824630797128485
> -38423538295 + 35892583295
-2530955000
>
```

然而，使用数字类型最棘手的部分是，许多数字的真实值实际上是实际数值的一个近似值。例如：

```
> 0.1 + 0.2
0.3000000000000004
>
```

当对浮点数执行算术运算时，仅仅操作任意的实际数字并不一定能够得到准确值：

```
> 1 - 0.3 + 0.1 == 0.8
false
>
```

对于这些情况，并不需要检查某个值是否在某个近似范围内，它的大小是由与之比较的数值的大小定义的（搜索[stack overflow.com](http://stackoverflow.com)网站，可以找到一些与比较浮点数相关的策略和思路的文章及问题）。

在某些情况下，JavaScript需要严格意义上的64位的整数值，而不能出现近似值这样的错误，我们可以使用字符串（string）类型，手动操纵这些数字，或者使用一些能够进行大整数运算的模块。

在某个数被0整除这点上，JavaScript和其他语言有很大的不同，它只会简单地返回一个正无穷大（Infinity）或者负无穷大（-Infinity），而不会抛出一个运行时异常：

```
> 5 / 0  
Infinity  
> -5 / 0  
-Infinity  
>
```

正无穷大和负无穷大在JavaScript里都是合法的值，并且可以使用它们进行比较。

```
> var x = 10, y = 0;  
undefined  
> x / y == Infinity  
true  
>
```

我们可以使用parseInt和parseFloat函数将字符串转换为数字：

```
> parseInt("32523523626263");  
32523523626263  
> parseFloat("82959.248945895");  
82959.248945895  
> parseInt("234.43634");  
234  
> parseFloat("10");  
10  
>
```

如果这些函数无法解析传入的参数，将会返回一个特殊值NaN (not-a-number)：

```
> parseInt("cat");  
NaN  
> parseFloat("Wankel-Rotary engine");  
NaN  
>
```

为了测试NaN，我们需要使用isNaN函数：

```
> isNaN(parseInt("cat"));
true
>
```

最后，为了测试一个给定的值是否是一个合法的有限数（不是 Infinity、-Infinity或者NaN），我们需要使用isFinite函数：

```
> isFinite(10/5);
true
> isFinite(10/0);
false
> isFinite(parseFloat("banana"));
false
>
```

2.1.4 boolean类型

JavaScript中的布尔（boolean）数据类型不仅简单而且容易使用。布尔值可以是true或者false，虽然技术上可以使用Boolean函数将其他值转换为布尔值，但实际上却很少需要使用这个函数，因为JavaScript语言会在需要时自动将任何值转换为布尔值，转换规则如下：

1) false、0、空字符串（""）、NaN、null以及undefined都等价于false。

2) 其他值都等价于true。

2.1.5 string类型

JavaScript中的字符串（string）是一组Unicode字符（内部以16位UCS-2格式实现）组成的序列，可以表示世界上绝大部分字符，包括大部分亚洲语言中使用的字符。JavaScript语言没有单独的字符（char）或者字符数据类型，可以使用只有一个字符的字符串来表示字符。对于大部分使用Node.js编写的网络应用，需要使用UTF-8格式对外通信，Node会自动处理转换细节。而如果是操纵二

进制数据，那么处理字符串和字符集相关的经验会非常重要。

字符串可以使用单引号或者双引号封装。单引号和双引号在功能上是等价的，可以选择任意一个使用。如果想要在使用单引号的字符串中包含单引号，可以使用\'；同理，如果在使用双引号的字符串中包含双引号，可以使用\"：

```
> 'Marc\'s hat is new.'  
'Marc\'s hat is new.'  
> "\"Hey, nice hat!\"", she said."  
"Hey, nice hat!", she said.'  
>
```

要想获得一个JavaScript字符串的长度，只需使用length属性：

```
> var x = "cat";  
undefined  
> x.length;  
3  
> "cat".length;  
3  
> x = null;  
null
```

在JavaScript中尝试获取值为null或undefined的字符串的长度时，将会抛出错误：

```
> x.length;  
TypeError: Cannot read property 'length' of null  
    at repl:1:2  
    at REPLServer.self.eval (repl.js:109:21)  
    at rli.on.self.bufferedCmd (repl.js:258:20)  
    at REPLServer.self.eval (repl.js:116:5)  
    at Interface.<anonymous> (repl.js:248:12)  
    at Interface.EventEmitter.emit (events.js:96:17)  
    at Interface._onLine (readline.js:200:10)  
    at Interface._line (readline.js:518:8)  
    at Interface._ttyWrite (readline.js:736:14)  
    at ReadStream.onkeypress (readline.js:97:10)
```

要想将两个字符串组合在一起，可以使用+操作符：

```
> "cats" + " go " + "meow";
'cats go meow'
>
```

如果将其他类型的数据混入到字符串中，JavaScript将尽可能将其他数据转换成字符串：

```
> var distance = 25;
undefined
> "I ran " + distance + " kilometres today";
'I ran 25 kilometres today'
>
```

注意，当混入的表达式过多时，则可能出现许多有趣的结果：

```
> 5 + 3 + " is my favourite number";
'8 is my favourite number'
>
```

如果想将“53”作为我最喜欢的数字，可以在表达式前加上一个空字符串，用来提前强制转换数据类型。

```
> "" + 5 + 3 + " is my favourite number";
'53 is my favourite number'
>
```

许多人担心在处理字符串时使用连接运算符+会导致严重的性能问题。好消息是几乎所有现代浏览器的JavaScript实现——包括Node.js使用的ChromeV8引擎，已经对该问题进行了深度优化，因此现在的运行性能非常好。

字符串函数

JavaScript中为字符串提供了许多有趣的函数。使用`indexOf`函数可以在一个字符串中搜索另外一个字符串。

```
> "Wishy washy winter".indexOf("wash");
6
>
```

从一个字符串中截取一个子串，可以使用substr或splice函数（前者会接受一个开始索引和一个需要截取的字符串长度；而后者则会接受一个开始索引和一个结束索引）：

```
> "No, they're saying Booo-urns.".substr(19, 3);
'Boo'
> "No, they're saying Booo-urns.".slice(19, 22);
'Boo'
>
```

如果字符串中有某个分隔符，可以使用split函数将字符串分割成子字符串并返回一个数组：

```
> "a|b|c|d|e|f|g|h".split("|");
[ 'a',
  'b',
  'c',
  'd',
  'e',
  'f',
  'g',
  'h' ]
>
```

最后，可以使用trim函数（V8 JS）清除字符串前后的空白字符：

```
> '      cat      \n\n\n      '.trim();
'cat'
>
```

正则表达式

JavaScript支持功能强大的正则表达式，正则表达式的具体细节不在本书讨论范围之内，但我会简单明了地展示如何以及在什么时候使用正则表达式。有几个字符串函数可以接收正则表达式作为参数并执行。正则表达式不仅可以使用字面量格式（literal format）（通过将正则表达式放入两个斜杠字符[/]之间表示），也可以通过调用RegExp对象的构造器来表示：

```
/[aA]{2,}/  
new RegExp("[Aa]{2,}")
```

以上两个都是正则表达式，用来表示一组两个或两个以上a字符的序列（大写或者小写）。

为了将字符串对象中两个或两个以上的a字符序列替换成b字母，我们可以使用replace函数，以下两种写法都是可行的：

```
> "aaoo".replace(new RegExp("[Aa]{2,}"), "b");  
'boo'  
> "aaoo".replace(/ [Aa]{2,} /, "b");  
'boo'  
>
```

与indexOf函数类似，search函数接收一个正则表达式参数，并返回第一个匹配此正则表达式的子字符串的位置索引，如果匹配不存在则返回-1：

```
> "aaoo".search(/ [Aa]{2,} /);  
0  
> "aoo".search(/ [Aa]{2,} /);  
-1  
>
```

2.1.6 object类型

对象（object）是JavaScript语言的核心之一，我们总会使用到它。对象是一种相当动态和灵活的数据类型，可以轻松地为其新增或删除属性。我们可以使用以下两种方式创建对象，而后者就是所谓的对象字面量语法（object literal syntax），它是目前最推荐的写法。

```
> var o1 = new Object();  
undefined  
> var o2 = {};  
undefined  
>
```

我们还可以使用对象字面量语法指定对象的内容，在初始化时，可以指定对象成员的名字以及对应的值：

```
var user = {  
    first_name: "marc",  
    last_name: "wandschneider",  
    age: Infinity,  
    citizenship: "man of the world"  
};
```

关于JSON

本书中最常用的东西之一就是JSON（事实上，包括网络和Web应用也是），即JavaScript对象表示法（JavaScript Object Notation）。这种数据交换格式充分发挥了基于文本的数据格式的灵活性，却没有像XML这样的其他格式所带来的麻烦（公平地讲，相较于后者，JSON缺少一些格式验证功能，但它仍然是最好使用的格式）。

JSON和对象字面量表示法非常相似，但是二者之间有两个关键的区别：对象字面量表示法使用单引号或双引号封装属性名，甚至可以不使用任何引号，而在JSON中却是强制使用引号的。另外，JSON中所有字符串都需要包含在双引号中：

```
// valid object literal notation, INVALID JSON:  
var obj = {  
    // JSON strings are supposed to use ", not '  
    "first_name": 'Marc',  
    // Must wrap property names for JSON  
    last_name: "Wandschneider"  
}  
  
// valid JSON and object literal notation:  
var obj = {  
    "first_name": "Marc",  
    "last_name": "Wandschneider"  
}
```

实际上大部分JSON库兼容单引号字符串，但是为了提高兼容性，无论是编写还是生成JSON数据，最好还是使用双引号。

通常我们可以使用V8的JSON.parse和JSON.stringify函数来生成JSON数据。前者接收一个JSON字符串作为参数，并将其转换成一个对象（如果失败，则抛出一个错误），而后者接收一个对象作为参数，并返回一个JSON字符串表示。

当在代码中编写对象时，我们经常使用对象字面量表示法，但同时本书中也会编写大量的JSON。因此，了解二者之间的区别非常重要。这里我需要特别指出，无论何时，JSON都是必要的。

我们可以使用以下任意一种方法来给自己的对象添加新属性：

```
> user.hair_colour = "brown";
'brown'
> user["hair_colour"] = "brown";
'brown'
> var attribute = 'hair_colour';
undefined
> user[attribute] = "brown";
'brown'
> user
{ first_name: 'marc',
  last_name: 'wandschneider',
  age: Infinity,
  citizenship: 'man of the world',
  hair_colour: 'brown' }
>
```

如果尝试访问一个不存在的属性，并不会报错，而是会得到 undefined这样的结果。

```
> user.car_make
undefined
>
```

当我们需要删除对象的某个属性时，可以使用delete关键字：

```
> delete user.hair_colour;
true
> user
{ first_name: 'marc',
  last_name: 'wandschneider',
  age: Infinity,
  citizenship: 'man of the world' }
>
```

JavaScript对象非常灵活，这使得它与其他语言的关联数组、哈希表、字典极其相似，但还是有一点不同：在JavaScript中获取一个关联数组对象的大小有些棘手。对象没有size或者length等属性或者方法。而为了得到对象的大小，可以使用如下写法（V8 JS）：

```
> Object.keys(user).length;  
4
```

请注意，这里使用了非标准的JavaScript扩展Object.keys，但V8和大多数浏览器（除了IE）都已经支持了这个方法。

2.1.7 array类型

JavaScript中的数组（array）类型实际上是JavaScript对象的一个特殊形式，它拥有一系列额外特性，这使得数组非常实用和强大。我们可以使用传统的表示法或者数组字面量语法（array literal syntax）来创建数组：

```
> var arr1 = new Array();  
undefined  
> arr1  
[]  
> var arr2 = [];  
undefined  
> arr2  
[]  
>
```

和对象一样，我倾向于使用字面量语法来创建数组，而很少使用传统表示法创建数组。

如果我们对数组使用typeof运算符，会得到一个令人惊讶的结果：

```
> typeof arr2  
'object'  
>
```

因为数组实际上就是对象，所以typeof运算符会返回"object"，而这不是我们想要的结果。幸运的是，V8有一个语言扩展，可以确定是否为一个数组：Array.isArray函数（V8 JS）。

```
> Array.isArray(arr2);
true
> Array.isArray({});
false
>
```

JavaScript中数组类型的一个关键特性是length属性，使用方法如下：

```
> arr2.length
0
> var arr3 = [ 'cat', 'rat', 'bat' ];
undefined
> arr3.length;
3
>
```

默认情况下，JavaScript数组是通过数字来进行索引的：

```
// this:
for (var i = 0; i < arr3.length; i++) {
    console.log(arr3[i]);
}
// will print out this:
cat
rat
bat
```

我们可以通过以下两种方式在数组的末尾添加新项：

```
> arr3.push("mat");
4
> arr3
[ 'cat', 'rat', 'bat', 'mat' ]
> arr3[arr3.length] = "fat";
'fat'
> arr3
[ 'cat', 'rat', 'bat', 'mat', 'fat' ]
>
```

可以通过指定特定的元素索引插入新元素。如果该元素的索引起超

过了最后一个元素，则两者之间的元素会被创建，并初始化为 undefined 值：

我们可以尝试使用`delete`关键字从数组中删除元素，但是结果可能会使我们感到惊讶：

可以看到索引2位置对应的值仍然“存在”，只是值被设置为undefined。

要想真正地从数组中删除某一项，可以使用splice函数，它会接收删除项的起始索引和数目作为参数。该函数会返回被删除的数组项，并且原始数组已经被修改，这些项不再存在：

实用函数

数组中有许多常用的函数。push和pop函数可以让我们向数组的末尾添加或者删除元素：

```
> var nums = [ 1, 1, 2, 3, 5, 8 ];
undefined
> nums.push(13);
7
> nums
[ 1, 1, 2, 3, 5, 8, 13 ]
> nums.pop();
13
> nums
[ 1, 1, 2, 3, 5, 8 ]
>
```

如果想在数组的头部插入或者删除元素，可以分别使用unshift和shift函数：

```
> var nums = [ 1, 2, 3, 5, 8 ];
undefined
> nums.unshift(1);
6
> nums
[ 1, 1, 2, 3, 5, 8 ]
> nums.shift();
1
> nums
[ 1, 2, 3, 5, 8 ]
>
```

与之前提到过的字符串函数split作用相反的是数组函数join，它会返回一个字符串：

```
> var nums = [ 1, 1, 2, 3, 5, 8 ];
undefined
> nums.join(", ");
'1, 1, 2, 3, 5, 8'
>
```

我们可以使用sort函数对数组进行排序，这里使用了内置的排序函数：

```
> var jumble_nums = [ 3, 1, 8, 5, 2, 1];
undefined
> jumble_nums.sort();
[ 1, 1, 2, 3, 5, 8 ]
>
```

而对于那些和预期不符的情况，我们可以自己提供排序函数，并将其作为参数传入sort函数中：

```
> var names = [ 'marc', 'Maria', 'John', 'jerry', 'alfred', 'Moonbeam'];
undefined
> names.sort();
[ 'John', 'Maria', 'Moonbeam', 'alfred', 'jerry', 'marc' ]
> names.sort(function (a, b) {
    var a1 = a.toLowerCase(), b1 = b.toLowerCase();
    if (a1 < b1) return -1;
    if (a1 > b1) return 1;
    return 0;
});
[ 'alfred', 'jerry', 'John', 'marc', 'Maria', 'Moonbeam' ]
>
```

遍历数组有许多方式，包括前文中的for循环，或者使用forEach函数（V8 JS），如下所示：

```
[ 'marc', 'Maria', 'John', 'jerry', 'alfred', 'Moonbeam'].forEach(function (value) {
    console.log(value);
});
marc
Maria
John
jerry
alfred
Moonbeam
```

2.2 类型比较和转换

前文提到过，大部分情况下，JavaScript数据类型的行为会如你期望的那样，与其他编程语言并无差异。JavaScript有两种相等运算符，相等运算符`==`（判断两个操作数有没有相同的值）和严格相等运算符`===`（判断两个操作数有没有相同的值以及是否为相同的数据类型）：

```
> 234 == '234'  
true  
> 234 === '234'  
false  
> 234234.235235 == 'cat'  
false  
> "cat" == "CAT"  
  
false  
> "cat".toUpperCase() == "CAT";  
true
```

可以看到很多不同的值都等价于`false`，事实上它们完全不同：

```
> '' == false == null == undefined == 0  
true  
> null === undefined  
false  
>
```

在处理任务时，验证函数的参数可以节省很多时间：

```
function fine(param) {  
    if (param == null || param == undefined || param == '')  
        throw new Error("Invalid Argument");  
}  
  
function better(param) {  
    if (!param) throw new Error("Invalid Argument");  
}
```

如果使用对象构造器来赋值而不是使用原始类型，比较类型时会比较诡异：

```
> var x = 234;
undefined
> var x1 = new Number(234);
undefined
> typeof x1
'object'
> typeof x
'number'
> x1 == x
true
> x1 === x
false
>
```

对象构造器在功能上和原始数据类型是一样的：一样的操作、操作符和函数会产生同样的结果，但是，严格相等运算符和typeof操作符则会产生不同的结果。因此，强烈建议在任何可能的情况下都只使用原始数据类型。

2.3 函数

虽然第一眼看起来不像（名字并没有帮到它），但其实JavaScript是一门函数式编程语言（functional programming language），这意味着函数是完全意义上的对象，可以被操纵、扩展，还可以作为数据进行传递。Node.js充分利用了这种能力，因此，我们会在网络和Web应用中广泛地使用函数。

2.3.1 基本概念

我们能想到的最简单的函数是这样的：

```
function hello(name) {  
    console.log("hello " + name);  
}  
> hello("marc");  
hello marc  
undefined  
>
```

要想在JavaScript函数中声明参数，可以在括号中简单地罗列参数，而且完全不需要在运行时检查这些参数：

```
> hello();  
hello undefined  
undefined  
> hello("marc", "dog", "cat", 48295);  
hello marc  
undefined  
>
```

当函数被调用时，如果传入的参数不够，剩下的变量会被赋予undefined值。而如果传入的参数过多，则多余的参数会被简单地做无用处理。

所有函数在函数体内都会有一个叫做arguments的预定义数组。它拥有函数调用时所有传入的实参，让我们可以对参数列表做額

外的检查。实际上，可以更进一步使用这个数组，从而让函数更加强大和灵活。

假设想初始化一个自己编写的缓存子系统。函数会接收一个大小值来创建缓存，而其他参数则使用默认值，例如缓存地址、过期算法、最大缓存项大小以及存储类型等。可以编写如下代码：

```
function init_cache(size_mb, location, algorithm, item_size, storage_type) {  
    ...  
}  
  
init_cache(100, null, null, null, null);
```

不过，如果我们让函数变得更加“聪明”从而可以通过多种方式调用，那将会非常酷：

```
function init_cache() {  
    var init_data = {  
        cache_size: 10,  
        location: '/tmp',  
        algorithm: 'lru',  
        item_size: 1024,  
        storage_type: 'btree'  
    };  
  
    var a = arguments;  
  
    for (var i = 0; i < a.length; i++) {  
        if (typeof a[i] == 'object') {  
            init_data = a[i];  
            break;  
        } else if (typeof a[i] == 'number') {  
            init_data.cache_size = a[i];  
            break;  
        } else {  
            throw new Error("bad cache init param");  
        }  
    }  
  
    // etc  
}
```

现在，有许多不同的方式来调用该函数：

```
init_cache();
init_cache(200);
init_cache({ cache_size: 100,
             location: '/exports/dfs/tmp',
             algorithm: 'lruext',
             item_size: 1024,
             storage_type: 'btree' } );
```

JavaScript中的函数甚至不需要名字：

```
var x = function (a, b) {
    return a + b;
}
> x(10, 20);
30
```

这些缺少名字的函数通常叫做匿名函数 (anonymous function)。完全匿名的函数有一个缺陷，它会在调试函数时出现：

```
var x = function () {
    throw new Error("whoopsie");
}

> x();
Error: whoopsie
    at x (repl:2:7)
    at repl:1:1
    at REPLServer.self.eval (repl.js:109:21)
    at rli.on.self.bufferedCmd (repl.js:258:20)
```

抛出异常时，匿名函数不会告知出现异常的函数名称。这会导致调试变得更加困难。

简单的解决办法就是为匿名函数命名：

```
var x = function bad_apple() {
    throw new Error("whoopsie");
}

> x();
```

```
Error: whoopsie
  at bad_apple (repl:2:7)
  at repl:1:1
  at REPLServer.self.eval (repl.js:109:21)
  at rli.on.self.bufferedCmd (repl.js:258:20)
```

在复杂的程序中，如果有一个具体的报错地址指针则会节约很多时间。因此，很多人都选择给它们所有的匿名函数命名。

在前文“Array类型”一节中，我们已经看到一个关于匿名函数的例子，该例使用匿名函数作为参数调用排序函数，用来进行大小写不敏感的字符串的比较。接下来我们会在本书中频繁地使用到匿名函数。

2.3.2 函数作用域

每次调用函数，都会创建一个新的变量作用域。父作用域中声明的变量对该函数是可见的，但是，当函数退出后，该函数作用域中声明的变量就会失效。参考以下代码：

```
var pet = 'cat';

function play_with_pets() {
  var pet = 'dog';
  console.log(pet);
}

play_with_pets();
console.log(pet);
```

它会输出如下结果：

```
dog
cat
```

我们可以将作用域和匿名函数结合起来做一些快速或私有的工作。这样，当匿名函数退出后，里面的私有变量也会消失。下面的示例用来计算一个圆锥体的体积：

```
var height = 5;
var radius = 3;
var volume;
// declare and immediately call anon function to create scope
(function () {
    var pir2 = Math.PI * radius * radius;    // temp var
    volume = (pir2 * height) / 3;
})();

console.log(volume);
```

在第3章中，我们将会学习到更多函数相关的常用模式。

2.4 语言结构

JavaScript包含了几乎所有的语言操作符和语句结构，其中包括绝大部分逻辑操作符和算术操作符。

JavaScript同样支持三元运算符：

```
var pet = animal_meows ? "cat" : "dog";
```

尽管大多数的数字都是作为双精度浮点数实现的，JavaScript仍然支持位操作符：&（与）、|（或）、~（非）以及^（异或XOR）操作符都能正常工作：

1.首先将操作数转换成32位整数。

2.进行位操作。

3.最后，将得到的32位整数再转换成JavaScript数字。

另外，除了标准的while、do...while和for循环，JavaScript还支持新的for循环语言扩展，叫做for...in循环（V8 JS）。这种循环用于获取对象的所有属性名：

```
var user = {
    first_name: "marc",
    last_name: "wandschneider",
    age: Infinity,
    occupation: "writer"
};

for (key in user) {
    console.log(key);
}
first_name
last_name
age
occupation
undefined
>
```

2.5 类、原型和继承

JavaScript的面向对象编程与其他语言有很大不同，因为JavaScript没有明确的类（class）关键字或类型。事实上，JavaScript中所有的类都是以函数的形式定义的：

```
function Shape () {
    this.X = 0;
    this.Y = 0;

    this.move = function (x, y) {
        this.X = x;
        this.Y = y;
    }
    this.distance_from_origin = function () {
        return Math.sqrt(this.X*this.X + this.Y*this.Y);
    }
}

var s = new Shape();
s.move(10, 10);
console.log(s.distance_from_origin());
```

上述程序产生下面的输出结果：

14.142135623730951

只要喜欢，我们可以在任何时候添加任意多的属性和方法到类中：

```
var s = new Shape(15, 35);
s.FillColour = "red";
```

声明类的函数同样也是这个类的构造函数！

然而，这种创建类的方式存在两个问题。首先，效率似乎有点低下，每一个对象都必须自己实现类方法。（每当创建一个新的Shape实例，都要创建move和distance_from_origin函数）其次，我们可能需要继承这个类来创建圆形和方形，并让新的类继承基类（base class）的方法和属性而不必做任何额外的工作。

[原型和继承](#)

默认情况下，所有的JavaScript对象都有一个原型（prototype）对象，它是一种继承属性和方法的机制。原型是多年来JavaScript中很多混乱的源头，往往是因为不同的浏览器使用不同的命名和略微不同的实现。因为原型与这一章有关，所以接下来将演示V8（也就是Node）使用的模型，以及其他现代JavaScript实现未来可能的走向。

修改之前创建的Shape类，从而使所有继承该类的对象都获得X和Y属性以及所有定义在这个类上的方法：

```
function Shape () {  
}  
  
Shape.prototype.X = 0;  
Shape.prototype.Y = 0;  
  
Shape.prototype.move = function (x, y) {  
    this.X = x;  
    this.Y = y;  
}  
Shape.prototype.distance_from_origin = function () {  
    return Math.sqrt(this.X*this.X + this.Y*this.Y);  
}  
Shape.prototype.area = function () {  
    throw new Error("I don't have a form yet");  
}  
var s = new Shape();  
s.move(10, 10);  
console.log(s.distance_from_origin());
```

运行这段脚本，会得到与前面相同的输出结果。事实上，除了可能稍微提升内存的效率之外（如果创建了大量的实例，它们都将共享而不是自己创建move和distance_from_origin方法），两者在功能上并无差异。如果添加一个area方法，则所有的shape实例都会拥有这个方法。而在基类中，这种方式却行不通，它会抛出错误。

更为重要的是，我们可以很容易地对类进行扩展：

```
function Square() {  
}  
  
Square.prototype = new Shape();  
Square.prototype.__proto__ = Shape.prototype;  
Square.prototype.Width = 0;  
Square.prototype.area = function () {  
    return this.Width * this.Width;  
}  
  
var sq = new Square();  
sq.move(-5, -5);  
sq.Width = 5;  
console.log(sq.area());  
console.log(sq.distance_from_origin());
```

新Square类的代码用到了一个在V8和其他实现中出现的新JavaScript语言特性：`__proto__`属性。它能够告诉JavaScript声明的新类的基本原型应该是指定的类型，因此也就可以从指定的类进行扩展。

可以进一步扩展新的类，叫做Rectangle，它会从Square类继承：

```
function Rectangle () {  
}  
  
Rectangle.prototype = new Square();  
Rectangle.prototype.__proto__ = Square.prototype;  
Rectangle.prototype.Height = 0;  
  
Rectangle.prototype.area = function () {  
    return this.Width * this.Height;  
}  
  
var re = new Rectangle();  
re.move(25, 25);  
re.Width = 10;  
re.Height = 5;  
console.log(re.area());  
console.log(re.distance_from_origin());
```

为了确认代码能正常运行，可以使用一个之前没有见过的操作符
instanceof :

```
console.log(sq instanceof Square);      // true
console.log(sq instanceof Shape);       // true
console.log(sq instanceof Rectangle);    // false
console.log(re instanceof Rectangle);    // true
console.log(re instanceof Square);       // true
console.log(re instanceof Shape);        // true
console.log(sq instanceof Date);         // false
```

2.6 错误和异常

JavaScript中，通常使用Error对象和一条信息来表示一个错误。当遇到错误情况时，可以抛出错误：

```
function uhoh () {
    throw new Error("Something bad happened!");
}

> uhoh();
Error: Something bad happened!
at uhoh (repl:2:7)
at repl:1:1
at REPLServer.self.eval (repl.js:109:21)
at rli.on.self.bufferedCmd (repl.js:258:20)
```

和其他语言一样，可以通过try/catch语句块来捕捉错误：

```
function uhoh () {
    throw new Error("Something bad happened!");
}

try {
    uhoh();
} catch (e) {
    console.log("I caught an error: " + e.message);
}

console.log("program is still running");
```

该程序的输出结果如下：

```
I caught an error: Something bad happened!
program is still running
```

在下一章中我们会发现，如果使用Node.js中的异步编程模式，这种处理错误的方式会导致一些问题。

2.7 几个重要的Node.js全局对象

Node.js有几个关键的全局对象，这些全局对象总是可见的。

2.7.1 global对象

当在Web浏览器上编写JavaScript代码时，会有一个window对象表现得像"global"变量。任何附加到它上面的变量或成员在应用中的任何地方都是可见的。

Node.js有个类似的东西，叫做global对象。任何附加到该对象上的东西在node应用中的任何地方都是可见的：

```
function printit(var_name) {  
    console.log(global[var_name]);  
}  
  
global.fish = "swordfish";  
global.pet = "cat";  
  
printit("fish"); // prints swordfish  
printit("pet"); // prints cat  
printit("fruit"); // prints undefined
```

2.7.2 console对象

在经常使用的console.log函数中，我们会看到Node.js有一个全局变量console。不仅如此，它还有一些有趣的函数：

- warn(msg)——与log类似的函数，但打印的是标准错误(stderr)。
- time(label)和timeEnd(label)——第一个函数被调用时会标识一个时间戳，而当第二个函数被调用时，会打印出从time函数被调用后中间经过的时间。

- assert(cond,message)——如果cond等价于false，则抛出一个AssertionFailure异常。

2.7.3 process对象

Node中另外一个关键的全局变量就是process对象，它包含许多信息和方法，如果继续阅读本书，就会接触到这些信息和方法。exit方法是中止Node.js程序的方式之一，而env函数会返回一个对象，它包含了当前用户的环境变量，而cwd则返回应用程序当前的工作目录。

2.8 小结

本章快速回顾了JavaScript语言，阐明了一些迷惑或未知的领域，希望能帮助大家提高一点语言方面的知识。有了这些基本的知识，我们现在可以开始学习如何使用Node.js创建强大快速的应用了。

第3章 异步编程

现在你应该对JavaScript编程有了全新的认识，因此是时候深入了解Node.js的核心概念了：非阻塞IO和异步编程。稍后你会看到这种机制给Node.js带来了巨大的优势和好处，但同时它也带来了许多问题和挑战。

3.1 传统编程方式

在这之前（2008年左右），当我们坐下来准备编写一个程序去加载一个文件时，代码会如下所示（假设正在使用类PHP语言编写示例）：

```
$file = fopen('info.txt', 'r');
// wait until file is open

$contents = fread($file, 100000);
// wait until contents are read

// do something with those contents
```

如果我们分析这段代码的执行情况，会发现大部分时间它什么都没有做。事实上，这段代码所花的绝大部分时间是在等待电脑的文件系统执行，最后才会返回请求的文件内容。进一步分析，对于绝大部分基于IO的应用——那些需要频繁连接数据库、和外部服务器通信或者读写文件的应用——脚本将会花费大量时间用来等待处理结果（见图3.1）。

服务器同时处理多个请求的方法就是并行这些脚本。现代电脑操作系统能够很好地支持多任务处理，所以可以很容易地在进程阻塞的时候切换任务，以便让其他进程访问CPU资源。而有些系统环境做得更好，使用多线程来替代多进程。

现在的问题是，每一个进程或者线程都会消耗大量的系统资源。我曾经看到在一些使用Apache和PHP编写的大型系统中，每一个进程占有高达10-15M的内存资源——它们从不关心操作系统不断切换上下文过程中所消耗的资源和时间。这种应用如果同时处理100个任务，甚至会消耗超过1GB的内存资源。使用多线程解决方案或者轻量级HTTP服务器可以获得更好的效果，但是仍然会遇到电脑消耗大量的时间用在等待阻塞进程返回结果的情况，甚至会面临没有足够的容量来处理更多请求的风险。

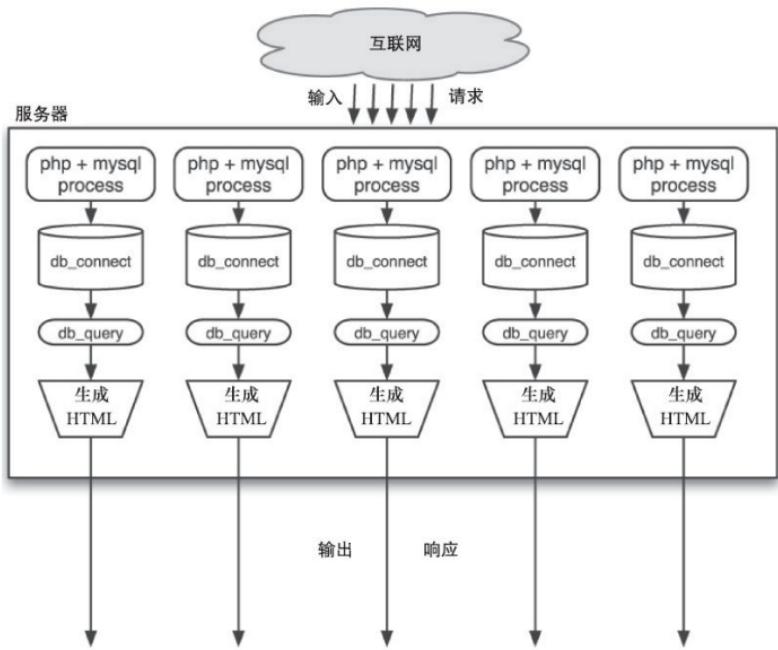


图3.1 传统阻塞IO Web服务器

如果有方法可以最大化利用CPU的计算能力和可用内存以减少资源浪费，那将再好不过了。而这，正是Node.js的精髓所在。

3.2 Node.js的编程方式

要理解Node.js如何将前文中展示的方法变成非阻塞异步模型，首先要看下JavaScript中的setTimeout这个函数。该函数的参数包含一个函数调用和等待时间，参数函数会在等待时间到达后执行：

```
// blah
setTimeout(function () {
    console.log("I've done my work!");
}, 2000);

console.log("I'm waiting for all my work to finish.");
```

如果运行上面这段代码，可以看到如下输出结果：

```
I'm waiting for all my work to finish.
I've done my work!
```

希望没有吓着你：这段程序设置了2000ms (2s) 的等待时间和需要调用的函数。程序继续执行，打印出"I'm waiting..."的文字。2秒钟以后，可以看到"I've done..."这样的信息，然后程序退出。

现在看一下，每次调用函数时，都需要等待外部资源（数据库服务器、网络请求或者文件系统的读写操作），这些情况非常类似。因此，我们需要选择调用fopen(path,mode,function callback(file_handle){...})函数来替代调用fopen(path,mode)并等待响应。

现在使用新的异步函数来重写前面的同步脚本。可以在命令行中使用node输入并运行该程序。注意，要确保创建了info.txt来供程序读取。

```
var fs = require('fs'); // this is new, see explanation

var file;
var buf = new Buffer(100000);
fs.open(
  'info.txt', 'r',
  function (handle) {
    file = handle;
  }
);

fs.read() // this will generate an error.
          file, buf, 0, 100000, null,
          function () {
            console.log(buf.toString());
            file.close(file, function () { /* don't care */ });
          }
);
});
```

也许你从没见过这段代码的第一行：require函数可以用来在Node.js程序中引入外部功能。Node拥有一套非常棒的模块（module），当想使用其中的某个功能时，可以单独地引入到代码中。后面我们将会进一步学习使用模块，还会在第5章中学习如何使用它们，并编写属于自己的模块。

如果就这么运行上面这段程序，它会抛出错误并退出。到底怎么回事？这是因为fs.open函数是异步执行的，它会在文件打开之前立刻返回，而handle值会返回给回调函数。在文件打开之前，file变量不会被赋值，它的值会在fs.open函数的回调函数中被接收。因此，如果在它后面立刻调用fs.read函数，file的值仍然是undefined。

修复这个程序很简单：

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

fs.open(
  'info.txt', 'r',
  function (err, handle) { // we'll see more about the err param in a bit
    var buf = new Buffer(100000);
    fs.read(
      handle, buf, 0, 100000, null,
      function (err, length) {
        console.log(buf.toString('utf8', 0, length));
        fs.close(handle, function () { /* don't care */ });
      }
    );
  }
);
```

思考这些异步函数如何工作的关键方法包括以下几个方面：

- 检查和验证参数

- 通知Node.js核心去排队调用相应的函数（如前面示例中，操作系统的open或者read函数），并在返回结果的时候通知（调用）回调函数

- 返回给调用者

也许你会问：如果open函数立刻返回，为什么node进程没有在函数返回后立刻退出？这是因为Node使用了事件队列（event queue）。如果有挂起的事件等待响应，它就不会退出，除非代码执行结束并且在队列上没有任何其他事件。如果你正在等待某个函数调用的响应（如open或read函数），Node就会一直等待。见图3.2，从概念上理解运行机制。

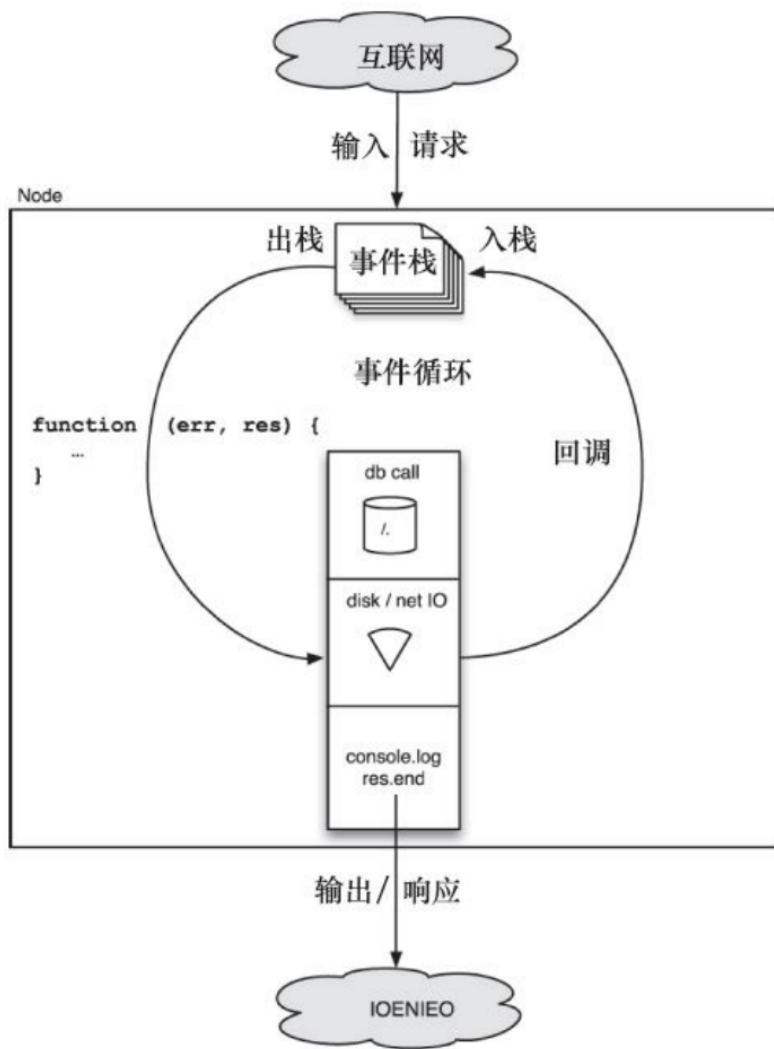


图3.2 只要代码还在执行或等待响应，Node就会一直运行

3.3 错误处理和异步函数

在前一章中，我们讨论了JavaScript中的错误处理、事件和try...catch代码块。在本章中，我们将会介绍非阻塞IO和异步函数回调，但这也会带来新的问题。请思考如下代码：

```
try {
    setTimeout(function () {
        throw new Error("Uh oh!");
    }, 2000);
} catch (e) {

    console.log("I caught the error: " + e.message);
}
```

如果运行以上代码，期望能看到输出结果为 “I caught the error:Uh oh!” 。但是可惜的是，我们实际上会看到：

```
timers.js:103
    if (!process.listeners('uncaughtException').length) throw e;
^

Error: Uh oh, something bad!
at Object._onTimeout errors_async.js:5:15)
at Timer.list.ontimeout (timers.js:101:19)
```

为什么会这样？难道try...catch代码块不会捕获错误么？不，它会捕获错误。但是，异步回调会给捕获错误带来一点小麻烦。

事实上，对setTimeout的调用的确是在try...catch代码块中执行的。如果函数抛出错误，catch就会捕获它，然后你就能看到期望的结果了。但是setTimeout函数只是在Node的事件队列中添加了一个新事件（是为了告诉Node在给定的等待时间以后调用该函数——在本例中是2000ms），然后返回。而这个回调函数实际上是在一个全新的上下文和作用域中执行的。

因此，当在非阻塞IO中调用异步函数的时候，只有极小一部分会抛出错误；大部分情况下，编译器会告诉你出错了。

在Node中，我们使用一些核心模式（core pattern）来帮助规范化编码，以避免出错。这些模式不是JavaScript这门语言或者引擎

强制要求的，但经常可以在项目中看到这些模式，甚至你也会自己使用到这些模式。

回调函数和错误处理

你第一个会看到的模式就是格式化传递给异步函数使用的回调函数。回调函数一般至少包含一个参数，即最后操作的成功或者失败状态；一般也会包含第二个参数，即最后操作返回的结果或信息（比如文件句柄、数据库连接、查询到的数据集等）；一些回调函数还可能包含更多的参数：

```
do_something(param1, param2, ..., paramN, function (err, results) { ... });
```

参数err的值一般会是：

- null，表明操作成功，并且会有一个返回值（如果有需要的话）。
- 一个Error对象的实例，你偶尔会看到一些不一致的地方：有些人喜欢在Error对象上添加code字段，并且用message字段保存错误信息；反之，有些人喜欢用一些其他的模式。本书中的所有代码使用的模式都会包含code字段，并使用message字段来尽可能提供更多的错误信息。所有的模块中code字段都是字符串类型，因为这样更具可读性。一些类库则会在Error对象中提供更多的信息，但至少要包含上述两个字段。

这种方式可以让我们写出的非阻塞代码更具可控性。整本书中，我将会使用回调函数中处理错误的两种不同的代码风格。下面是第一种：

```
fs.open(
  'info.txt', 'r',
  function (err, handle) {
    if (err) {
      console.log("ERROR: " + err.code + " (" + err.message ")");
      return;
    }
    // success!! continue working here
  });
});
```

这种风格需要先检查错误，如果有错误就直接返回；否则，继续

处理结果。接下来是第二种风格：

```
fs.open(
  'info.txt', 'r',
  function (err, handle) {
    if (err) {
      console.log("ERROR: " + err.code + " (" + err.message ")");
    } else {
      // success! continue working here
    }
  }
);
```

这种风格中，使用if...then...else语句来处理错误。

这两者之间的区别看似有些微不足道，但是前者更容易产生bug和错误，因为可能会忘记在if语句中添加return语句。但是后者会让代码不断地缩进，这样会导致每行的代码冗长且可读性差。当然，你会在第5章中找到这个问题的解决方案。

全新的包含错误处理的文件加载代码如代码清单3.1所示。

代码清单3.1 包含完整错误处理的文件加载

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

fs.open(
  'info.txt', 'r',
  function (err, handle) {
    if (err) {
      console.log("ERROR: " + err.code
                  + " (" + err.message + ")");
      return;
    }
    var buf = new Buffer(100000);
    fs.read(
      handle, buf, 0, 100000, null,
      function (err, length) {

        if (err) {
          console.log("ERROR: " + err.code +
                      " (" + err.message + ")");
          return;
        }
        console.log(buf.toString('utf8', 0, length));
        fs.close(handle, function () { /* don't care */ });
      }
    );
  }
);
```

3.4 我是谁——如何维护本体

现在，需要写一个小小的类来处理一些普通的文件操作。

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

function FileObject () {
    this.filename = '';

    this.file_exists = function (callback) {
        console.log("About to open: " + this.filename);
        fs.open(this.filename, 'r', function (err, handle) {
            if (err) {
                console.log("Can't open: " + this.filename);
                callback(err);
                return;
            }
            fs.close(handle, function () { });
            callback(null, true);
        });
    };
}
```

上述代码中添加了一个属性——filename和一个方法——file_exists。该方法会做如下事情：

- 尝试以只读方式打开filename属性指定的文件。
- 如果文件不存在，则打印日志信息，并把err作为参数来调用回调函数。
- 如果文件存在，调用回调函数，以表明打开文件成功。

现在，在下述代码中使用该类：

```
var fo = new FileObject();
fo.filename = "file_that_does_not_exist";

fo.file_exists(function (err, results) {
    if (err) {

        console.log("Aw, bummer: " + JSON.stringify(err));
        return;
    }

    console.log("file exists!!!");
});
```

希望看到如下输出结果：

```
About to open: file_that_does_not_exist  
Can't open: file_that_does_not_exist
```

但事实上，你会看到：

```
About to open: file_that_does_not_exist  
Can't open: undefined
```

怎么回事？大多数情况下，当一个函数嵌套在另一个函数中时，它就会自动继承父/宿主函数的作用域，因而就能访问所有的变量了。那么，为什么嵌套的回调函数却没有返回正确的filename属性的值呢？

这个问题归根于this关键字和异步回调函数本身。别忘了，当你调用fs.open这样的函数的时候，它会首先初始化自己，然后调用底层的操作系统函数（本例中，就是打开文件），并把回调函数插入到事件队列中去。执行完会立即返回给FileObject#file_exists函数，然后退出。当fs.open函数完成任务后，Node就会调用该回调函数，但此时，该函数已经不再拥有FileObject这个类的继承关系了，所以回调函数会被重新赋予新的this指针。

坏消息是，在这一过程中已经丢失了指向FileObject的this指针。但好消息是，fs.open的回调函数还保留着它的函数作用域。一个常用的解决方法就是把消失的this指针“保存”到变量中，变量名可以是self、me或者其他类似的名称。现在，重写file_exists函数并利用this指针：

```
this.file_exists = function (callback) {
  var self = this;

  console.log("About to open: " + self.filename);
  fs.open(this.filename, 'r', function (err, handle) {
    if (err) {
      console.log("Can't open: " + self.filename);
      callback(err);
      return;
    }

    fs.close(handle, function () { });
    callback(null, true);
  });
};
```

由于函数作用域是通过闭包保留的，所以self变量会被一直保持着，即使回调函数是被Node.js异步执行的。在后面的应用中，this会有更广泛的使用场景。有些人喜欢用me而不是self，因为这一命名更简短；而有些人则会使用其他完全不同的命名。你可以选择一个喜欢的命名，然后一直用下去，以保持一致性。

3.5 保持优雅——学会放弃控制权

Node运行在单线程中，使用事件轮询来调用外部函数和服务。它将回调函数插入事件队列中来等待响应，并且尽快执行回调函数。那么，如果一个函数需要计算两个数组的交叉元素，会发生些什么呢：

```
function compute_intersection(arr1, arr2, callback) {
    var results = [];
    for (var i = 0 ; i < arr1.length; i++) {
        for (var j = 0; j < arr2.length; j++) {
            if (arr2[j] == arr1[i]) {
                results[results.length] = arr1[i];
                break;
            }
        }
    }
    callback(null, results); // no error, pass in results!
}
```

当面对拥有数千个元素的数组的时候，该函数就会耗费大量的计算时间，大约会花费1秒甚至更多。在单线程模式中，Node.js在同一时间只做一件事情，所以，这个数量级的时间就会成为一个问题。像一些计算哈希、摘要（digest）或者一些其他消耗时间的操作，都会导致应用在处理过程中处于“假死”状态。那么，我们能做些什么呢？

在本书的前言中，我曾提到，Node.js不会特别适合某些特定的应用场景，其中之一就是作为计算服务器。Node更适合一些常见的网络应用任务，比如那些需要大量I/O或者需要向其他服务请求的任务。如果你想要编写一个需要大量计算的服务器，可能需要考虑把这些操作迁移到其他服务上去，这样Node应用就可以远程调用了。

但是，这并不意味着Node必须避免高强度计算的任务。如果只是偶尔执行这种任务，那么就可以在Node.js中使用它们，并且可以利用全局对象process中的nextTick方法。这种方法就好像在跟系统说“我放弃执行控制权，你可以在你空闲的时候执行我提供给你的函数”。相较于使用setTimeout函数，这种方式会显著提高执行速

度。

代码清单3.2是最新版本的compute_intersection函数，可以让Node每隔一段时间就处理一次其他任务。

代码清单3.2 使用process#nextTick，让代码变优雅

```
function compute_intersection(arr1, arr2, callback) {
    // let's break up the bigger of the two arrays
    var bigger = arr1.length > arr2.length ? arr1 : arr2;
    var smaller = bigger == arr1 ? arr2 : arr1;
    var biglen = bigger.length;

    var smlen = smaller.length;

    var sidx = 0;           // starting index of any chunk
    var size = 10;          // chunk size, can adjust!
    var results = [];        // intermediate results

    // for each chunk of "size" elements in bigger, search through smaller
    function sub_compute_intersection() {
        for (var i = sidx; i < (sidx + size) && i < biglen; i++) {
            for (var j = 0; j < smlen; j++) {
                if (bigger[i] == smaller[j]) {
                    results.push(smaller[j]);
                    break;
                }
            }
        }

        if (i >= biglen) {
            callback(null, results); // no error, send back results
        } else {
            sidx += size;
            process.nextTick(sub_compute_intersection);
        }
    }

    sub_compute_intersection();
}
```

在这个新版的函数中，只需简单地将较大的输入数组分割成10个元素一组的数据块（可以是任意大小的数据块），分别计算交叉元素，然后调用process#nextTick函数，从而允许Node处理其他事件或者请求。只有当该任务的队列前面没有事件时，才会继续执行该任务。别忘记将回调函数sub_compute_intersection传递给process#nextTick，这样才能保证当前作用域能被作为闭包保存下来。通过这种方式，我们就能将中间结果保存到compute_intersection函数中的变量中了。

代码清单3.3展示的是compute_intersection函数的测试代码。

代码清单3.3 测试compute_intersection函数

```
var a1 = [ 3476, 2457, 7547, 34523, 3, 6, 7,2, 77, 8, 2345,
7623457, 2347, 23572457, 237457, 234869, 237,
24572457524] ;
var a2 = [ 3476, 75347547, 2457634563, 56763472, 34574, 2347,
7, 34652364 , 13461346, 572346, 23723457234, 237,
234, 24352345, 537, 2345235, 2345675, 34534,
7582768, 284835, 8553577, 2577257,545634, 457247247,
2345 ];
compute_intersection(a1, a2, function (err, results) {
  if (err) {
    console.log(err);
  } else {
    console.log(results);
  }
});
```

尽管现在已经比最初的计算交叉元素的函数版本要稍许复杂，但在单线程的Node事件处理和回调的世界中已经运行得非常好了。当然，我们可以在任何复杂的、计算速度慢的场景中使用 `process.nextTick`。

3.6 同步函数调用

至此，我已经花了几乎整个章节介绍Node.js是如何异步运行的，并且介绍了很多技巧和非阻塞IO编程中的陷阱。但我必须提醒，Node确有一些核心API的同步版本，尤其是在操作文件的API中。在第11章中，我们将会使用这些同步API编写命令行工具。

为了扼要说明，可以重写本章的第一个脚本，如下所示：

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

var handle = fs.openSync('info.txt', 'r');
var buf = new Buffer(100000);
var read = fs.readSync(handle, buf, 0, 10000, null);
console.log(buf.toString('utf8', 0, read));
fs.closeSync(handle);
```

在你阅读本书的过程中，我希望你能意识到Node.js不仅仅可以编写网络或者Web应用，还可以使用它做任何其他一些事情，包括命令行工具、原型设计、服务器管理等。

3.7 小结

传统的编程模型在执行一系列同步、阻塞IO函数调用时，会等待这些函数执行完成才会继续执行下去；而Node的编程模型则是异步执行，当任务执行完成后，会由Node进行通知，并不会阻塞其他任务的执行。从前者到后者的转变需要大家多思考并付诸实践。不过，我坚信一旦掌握了这些窍门，你便再也不想回到以前那种编写Web应用的方式中去了。

下一章，我们将要开始尝试编写第一个简单的JSON应用服务器。

第二部分 提高篇

第4章 编写简单应用

第5章 模块化

第6章 扩展Web服务器

第4章 编写简单应用

现在我们已经对JavaScript语言如何工作有了深入的了解，现在是时候运用Node.js编写Web应用了。正如本书前言中提到的，我们将会在整本书中编写一个小型的相册网站。本章中，我们将会编写一个JSON服务器，用以提供相册列表以及每个相册对应的照片列表等服务，最后，还会添加为相册重命名的功能。在这个过程中，我们会理解JSON服务器运行的基本知识，这其中包含了服务器与HTTP进行交互的基础知识，例如GET和POST参数、头信息、请求和响应。第一代相册应用会使用文件相关的API来完成工作，这是用来理解上述知识点的最佳方式，可以使我们专注于要学习的新概念。

4.1 第一个JSON服务器

在第1章的最后，我们编写了一点关于HTTP服务器的代码，对于任何传入的请求都会返回纯文本"Thanks for calling!\n"。现在我们可以对其稍微做些修改，让它变得有些不同：

- 1) 指明返回的数据格式是application/json，而不是text/plain。
- 2) 使用console.log打印获取到的请求。
- 3) 返回JSON字符串。

这是一个非常小的服务器，它保存到simple_server.js文件中：

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

function handle_incoming_request(req, res) {
    console.log("INCOMING REQUEST: " + req.method + " " + req.url);
    res.writeHead(200, { "Content-Type" : "application/json" });
    res.end(JSON.stringify( { error: null } ) + "\n");
}

var s = http.createServer(handle_incoming_request);
s.listen(8080);
```

通过终端窗口（Mac/Linux）或者命令提示符（Windows）运行程序，如下所示：

```
node simple_server.js
```

上述代码只是等待请求而不会做其他事情。现在，在另外一个终端窗口中输入以下代码：

```
curl -X GET http://localhost:8080
```

如果一切都是正确的，在运行服务器的窗口中应该能够看到如下信息：

```
INCOMING REQUEST: GET /
```

而在运行curl命令的窗口，则应该能看到：

```
{"error":null}
```

可以尝试运行不同的curl命令，并观察都会发生什么。例如：

```
curl -X GET http://localhost:8080/gobbledygook
```

随着第一个程序的运行，我们可以规范JSON响应的输出，在输出中都会有一个error字段。通过这种方式，应用可以快速判断请求是成功还是失败。万一出现失败的情况，则会包含一个message字段，用来存放更多的错误信息；反之，JSON响应会返回数据，并会包含一个data字段：

```
// failure responses will look like this:  
{ error: "missing_data",  
  message: "You must include a last name for the user" }  
  
// success responses will usually have a "data" object  
{ error: null,  
  data: {  
    user: {  
      first_name: "Horatio",  
      last_name: "Gadsplatt III",  
      email: "horatio@example.org"  
    }  
  }  
}
```

一些应用选择使用数字编码来对应它们的错误类型。是否使用这种方式完全取决于你自己，但是我更倾向于使用文本，因为文本更具描述性，而且能够在使用类似curl等命令行测试程序时省下一个查询错误类型的步骤。`no_such_user`显然比-325来得更翔实些。

数据返回

一开始，我们的相册应用相当简单：它实际上是一个相册的集合，每个相册则是照片的集合，如图4.1所示：

ITALY2012
Our Trip to Italy

JAPAN2010
A Week in Tokyo

AUSTRALIA2010
Melbourne Fun!

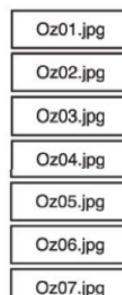
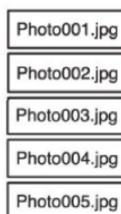


图4.1 相册和照片

现在，所有的相册都在执行脚本所在位置的子文件夹albums/下的子文件夹下：

```
scripts/  
scripts/albums/  
scripts/albums/italy2012  
scripts/albums/australia2010  
scripts/albums/japan2010
```

因此，为了获取相册列表，我们只需找到albums/子文件夹下的项。可以使用fs.readdir函数，这个函数会返回指定文夹件下的所有项（除了“.” 和 “..”）。load_album_list函数的代码如下所示：

```
function load_album_list(callback) {  
    fs.readdir(  
        "albums/",  
        function (err, files) {  
            if (err) {  
                callback(err);  
                return;  
            }  
            callback(null, files);  
        }  
    );  
}
```

我们仔细地阅读下这个函数的代码。首先，函数调用fs.readdir函数并提供了一个回调函数，当目录下的所有项都加载完毕后，就调

用这个函数。该回调函数拥有大多数回调都有的原型：一个error参数和一个result参数，我们可以在任何需要的地方使用这些参数。

注意，load_album_list函数本身唯一的参数是一个回调函数。因为load_album_list函数本身是异步的，它需要知道当自己完成工作之后要将相册列表传递到哪里。它不能将结果返回给调用者，因为它会在fs.readdir函数调用回调并给出结果之前就已经执行完毕。

这就是Node应用编程的核心技术：告诉Node去做某件事情，并在Node完成时告知Node将结果传递给谁。而与此同时，我们继续执行其他的工作。很多执行的任务基本上都是以一系列长长的回调作为结束。

代码清单4.1包含了新相册列表服务器的完整代码。

代码清单4.1 相册列表服务器 (load_albums.js)

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

function load_album_list(callback) {
    fs.readdir(
        "albums",
        function (err, files) {
            if (err) {
                callback(err);
                return;
            }
            callback(null, files);
        }
    );
}

function handle_incoming_request(req, res) {
    console.log("INCOMING REQUEST: " + req.method + " " + req.url);
    load_album_list(function (err, albums) {
        if (err) {
            res.writeHead(503, {"Content-Type": "application/json"});
            res.end(JSON.stringify(err) + "\n");
            return;
        }

        var out = { error: null,
                   data: { albums: albums }};
        res.writeHead(200, {"Content-Type": "application/json"});
        res.end(JSON.stringify(out) + "\n");
    });
}

var s = http.createServer(handle_incoming_request);
s.listen(8080);
```

在代码清单4.1中，fs.readdir执行完毕后，我们会检查执行结果。如果产生错误，则会返回错误给调用者（在handle_incoming_request函数中，作为参数传入load_album_list函数的函数）；否则，我们会发送文件夹（相册）列表给调用者，结

果包含了null以表明没有产生错误。

代码清单还加了一些新的错误处理代码到handle_incoming_request函数中：如果fs.readdir函数报告已经发生一些错误，我们也会让调用者知晓这个情况，所以服务器还是能够返回一些JSON数据和HTTP响应码503，用来表明发生了意外情况。JSON服务器需要返回尽可能多的信息给它们的客户端，用来帮助客户端判断问题是由客户端本身产生还是因为服务器内部出现了问题。

如果要测试程序，请确保将要运行脚本的文件夹中包含albums/子文件夹，并且其中包含相册子文件夹。要运行服务器，可以再次运行脚本：

```
node load_albums.js
```

然后运行以下命令来获取结果：

```
curl -X GET http://localhost:8080/
```

curl命令返回的结果应该和下面类似：

```
{"error":null,"data":{"albums":["australia2010","italy2012","japan2010"]}}
```

4.2 Node模式：异步循环

如果在albums/文件夹中创建文本文件info.txt，那么会发生什么呢？相册列表服务器会返回什么结果？我们很可能看到如下结果：

```
{"error":null,"data":{"albums":["australia2010","info.txt","italy2012","japan2010"]}}
```

对于这个程序，我们真正需要的是检查fs.readdir的结果并返回所有文件夹而不是文件。为了这个目标，可以使用fs.stat函数，它传入一个对象，可以检测该对象是否是一个文件夹。

所以，重写load_album_list函数并遍历fs.readdir的结果，判断它们是否是文件夹：

```
function load_album_list(callback) {
    fs.readdir(
        "albums",
        function (err, files) {
            if (err) {
                callback(err);
                return;
            }

            var only_dirs = [];
            for (var i = 0; i < files.length; i++) {
                fs.stat(
                    "albums/" + files[i],
                    function(err, stats) {
                        if (stats.isDirectory()) {
                            only_dirs.push(files[i]);
                        }
                    }
                );
            }

            callback(null, only_dirs);
        }
    );
}
```

保持程序其他部分不变，然后运行curl命令。它总会返回如下结果：

```
{"error":null,"data":{"albums":[]}}
```

服务器崩溃了！发生了什么？

问题出在新添加的for循环代码上，因为大多数循环和异步回调不能兼容。之前的代码是这样写的：

- 创建一个only_dirs数组来缓存响应。
- 对于文件数组的每一项，调用非阻塞函数fs.stat并将其传入给定的函数，测试这个文件是否是个目录。
- 当所有的非阻塞函数都开始后，退出for循环并调用callback参数。因为Node.js是单线程运行的，所以所有的fs.stat函数都没有机会执行及调用回调函数，最后导致only_dirs的值一直是null，并将这个值传给提供的回调函数。实际上，当fs.stat的回调函数最终被执行时，已经无人在乎了。

为了解决这个问题，必须使用递归。我们可以快速创建一个具有以下格式的新函数，然后立刻调用它：

```
function iterator(i) {  
    if( i < array.length ) {  
        async_work( function(){  
            iterator( i + 1 )  
        })  
    } else {  
        callback(results);  
    }  
}  
iterator(0)
```

实际上我们可以做到一步到位，即使用一个命名匿名函数，这样就不会因为函数名而弄乱函数的作用域了：

```
(function iterator(i) {
    if( i < array.length ) {
        async_work( function(){
            iterator( i + 1 )
        })
    } else {
        callback(results);
    }
})(0);
```

因此，要想重写循环代码来测试fs.readdir的结果是否为文件夹，可以编写如下代码：

```
function load_album_list(callback) {
    fs.readdir(
        "albums",
        function (err, files) {
            if (err) {
                callback(err);
                return;
            }

            var only_dirs = [];
            (function iterator(index) {
                if (index == files.length) {
                    callback(null, only_dirs);
                    return;
                }
                fs.stat(
                    "albums/" + files[index],
                    function (err, stats) {
                        if (err) {
                            callback(err);
                            return;
                        }
                        if (stats.isDirectory()) {
                            only_dirs.push(files[index]);
                        }
                        iterator(index + 1)
                    }
                );
            })(0);
        }
    );
}
```

保存最新版本的JSON服务器代码，并运行curl命令，现在应该能够看到只包含相册文件夹而不包含文件的结果。

4.3 小戏法：处理更多的请求

目前相册JSON服务器只能响应一种请求：针对相册列表的请求。实际上，服务器并不在意如何调用这个请求，因此总是返回相同的结果。

我们可以扩展这个服务器的功能，允许处理以下两种请求：

- 1) 可用相册列表——调用/albums.json请求。
- 2) 相册中的照片列表——调用/albums/album_name.json请求。

为请求添加.json后缀，强调当前编写的JSON服务器只用于这类请求。

新版的支持这两种请求的handle_incoming_request函数代码如下：

```
function handle_incoming_request(req, res) {  
    console.log("INCOMING REQUEST: " + req.method + " " + req.url);  
    if (req.url == '/albums.json') {  
        handle_list_albums(req, res);  
    } else if (req.url.substr(0, 7) == '/albums'  
              && req.url.substr(req.url.length - 5) == '.json') {  
        handle_get_album(req, res);  
    } else {  
        send_failure(res, 404, invalid_resource());  
    }  
}
```

上面代码中两个if语句块都被加粗显示，它们会检测传入的请求对象的url属性。如果是简单的/albums.json请求，那么可以和之前一样处理。如果是/albums/something.json请求，则可以假设这是处理某个相册的内容列表的请求，并对它进行相应处理。

生成并返回相册列表的代码已经迁移到叫做handle_list_albums的函数，而获取某个独立相册内容的代码同样被组织到两个函数中，分别叫做handle_get_album函数和load_album函数。代码清单4.2包含了服务器的所有代码。

对于新版的代码，可以稍微修改JSON服务器的输出：所有的返回都是对象，而不仅仅是一个字符串数组。当在本书后面章节开始生成UI来匹配JSON响应时，这会帮助到我们。在代码清单4.2中，我使用斜体代码来标识这种变化。

尽管我尝试避免在本书后面包含冗长、乏味的代码，但是服务器代码的首个版本还是值得完整的浏览，因为后面我们做的大多数事情都是建立在当前这些代码之上。

代码清单4.2 处理多种请求类型

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

function load_album_list(callback) {
    // we will just assume that any directory in our 'albums'
    // subfolder is an album.
    fs.readdir(
        "albums",
        function (err, files) {
            if (err) {
                callback(make_error("file_error", JSON.stringify(err)));
                return;
            }

            var only_dirs = [];
            (function iterator(index) {
                if (index == files.length) {
                    callback(null, only_dirs);
                    return;
                }
            })
        }
    );
}
```

```

        fs.stat(
            "albums/" + files[index],
            function (err, stats) {
                if (err) {
                    callback(make_error("file_error",
                        JSON.stringify(err)));
                    return;
                }
                if (stats.isDirectory()) {
                    var obj = { name: files[index] };
                    only_dirs.push(obj);
                }
                iterator(index + 1)
            }
        );
    })(0);
}

function load_album(album_name, callback) {
    // we will just assume that any directory in our 'albums'
    // subfolder is an album.
    fs.readdir(
        "albums/" + album_name,
        function (err, files) {
            if (err) {
                if (err.code == "ENOENT") {
                    callback(no_such_album());
                } else {
                    callback(make_error("file_error",
                        JSON.stringify(err)));
                }
                return;
            }

            var only_files = {};
            var path = "albums/" + album_name + "/";

            (function iterator(index) {
                if (index == files.length) {
                    var obj = { short_name: album_name,
                               photos: only_files };
                    callback(null, obj);
                    return;
                }

                fs.stat(
                    path + files[index],
                    function (err, stats) {

```

```

        if (err) {
            callback(make_error("file_error",
                JSON.stringify(err)));
            return;
        }
        if (stats.isFile()) {
            var obj = { filename: files[index],
                desc: files[index] };
            only_files.push(obj);
        }
        iterator(index + 1)
    }
);
})(0);
}
};

function handle_incoming_request(req, res) {
    console.log("INCOMING REQUEST: " + req.method + " " + req.url);
    if (req.url == '/albums.json') {
        handle_list_albums(req, res);
    } else if (req.url.substr(0, 7) == '/albums' &&
        req.url.substr(req.url.length - 5) == '.json') {
        handle_get_album(req, res);
    } else {
        send_failure(res, 404, invalid_resource());
    }
}

function handle_list_albums(req, res) {
    load_album_list(function (err, albums) {
        if (err) {
            send_failure(res, 500, err);
            return;
        }

        send_success(res, { albums: albums });
    });
}

function handle_get_album(req, res) {
    // format of request is /albums/album_name.json
    var album_name = req.url.substr(7, req.url.length - 12);
    load_album(
        album_name,
        function (err, album_contents) {
            if (err && err.error == "no_such_album") {
                send_failure(res, 404, err);
            } else if (err) {
                send_failure(res, 500, err);

            } else {
                send_success(res, { album_data: album_contents });
            }
        });
}

function make_error(err, msg) {
    var e = new Error(msg);
    e.code = err;
    return e;
}

function send_success(res, data) {
    res.writeHead(200, {"Content-Type": "application/json"});
    var output = { error: null, data: data };
    res.end(JSON.stringify(output) + "\n");
}

function send_failure(res, code, err) {
    var code = (err.code) ? err.code : err.name;
    res.writeHead(code, { "Content-Type" : "application/json" });
    res.end(JSON.stringify({ error: code, message: err.message }) + "\n");
}

function invalid_resource() {
    return make_error("invalid_resource",
        "the requested resource does not exist.");
}

function no_such_album() {
    return make_error("no_such_album",
        "The specified album does not exist");
}

var s = http.createServer(handle_incoming_request);
s.listen(8080);

```

为了避免大量复制代码，我们拆分出很多关于发送最后成功信息或对客户端请求响应的代码。这些代码都存放在send_success以及send_failure函数中，这两个函数确保设置了正确的HTTP响应码并能够返回正确的JSON。

阅读完新版load_album函数后，我们会发现它和load_album_list函数有些类似。它枚举了相册文件夹内的所有项，然后检查并确保它们都是正常的文件，最后返回列表。另外我添加了几行代码，对load_album函数中的fs.readdir做了错误处理。

```
if (err.code == "ENOENT") {      // see text for more info
    callback(no_such_album());
} else {
    callback({ error: "file_error",
        message: JSON.stringify(err) });
}
```

如果fs.readdir失败，基本上都是因为查找不到相册文件夹，这是用户导致的错误：用户指定了一个非法的相册。这时如果想返回一个错误来指出这个事实，可以使用帮助函数no_such_album来完成这项任务。而对于其他大部分错误，则可能是服务器配置导致的错误，可以为这些报错场景返回更加通用的"file_error"。

现在获取的/albums.json的输出结果如下所示：

```
{"error":null,"data":{"albums":[{"name":"australia2010"}, {"name":"italy2012"}, {"name":"japan2010"}]}}
```

当上传一些图片文件到各个相册文件夹之后，获取的相册内容的输出结果（例如/albums/italy2012.json）则会与下面的类似（这里做了清理）：

```
{  
    "error": null,  
    "data": {  
        "album_data": {  
            "short_name": "/italy2012",  
            "photos": [  
                {  
                    "filename": "picture_01.jpg",  
                    "desc": "picture_01.jpg"  
                },  
                {  
                    "filename": "picture_02.jpg",  
                    "desc": "picture_02.jpg"  
                },  
                {  
                    "filename": "picture_03.jpg",  
                    "desc": "picture_03.jpg"  
                },  
                {  
                    "filename": "picture_04.jpg",  
                    "desc": "picture_04.jpg"  
                },  
                {  
                    "filename": "picture_05.jpg",  
                    "desc": "picture_05.jpg"  
                }  
            ]  
        }  
    }  
}
```

4.4 请求和响应对象的更多细节

现在输入并运行下面的程序：

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

function handle_incoming_request(req, res) {
    console.log("-----");
    console.log(req);

    console.log("-----");
    console.log(res);
    console.log("-----");
    res.writeHead(200, { "Content-Type" : "application/json" });
    res.end(JSON.stringify( { error: null } ) + "\n");
}

var s = http.createServer(handle_incoming_request);
s.listen(8080);
```

然后，在另外一个终端窗口中运行curl命令：

```
curl -X GET http://localhost:8080
```

客户端窗口应该会只打印error:null，但在服务器端窗口则打印了大量的额外文本信息，这些信息包含了传入HTTP服务器的请求以及响应对象。

我们已经使用过请求对象的两个属性：method和url。前一个属性标识传入的请求是GET、POST、PUT还是DELETE请求（也可能是HEAD等其他的），而后一个属性则包含了发送到服务器请求的URL。

请求对象是一个Node.js的HTTP模块提供的ServerRequest对象，可以查阅Node文档以了解更多细节。文档除了包含这两个属性，还包含了ServerRequest处理POST数据相关的知识。当然，也可以通过查看headers属性来检测传入的请求头。

如果查看curl程序发送过来的请求头，会看到：

```
{ 'user-agent': 'curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24.0 OpenSSL/0.9.8r
 zlib/1.2.5',
  host: 'localhost:8080',
  accept: '*/*' }
```

如果在浏览器中调用JSON服务器，则会看到类似的信息：

```
{ host: 'localhost:8080',
  'user-agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10.8; rv:16.0) Gecko/20100101
Firefox/16.0',
  accept: 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8',
  'accept-language': 'en-US,en;q=0.5',
  'accept-encoding': 'gzip, deflate',
  connection: 'keep-alive' }
```

在响应端，我们也使用过两个方法：writeHead和end。对于传入的每个请求都必须调用一次且只能调用一次end方法。否则，客户端会无法获得响应而只会持续地监听连接以获得更多数据。

当编写自己的响应时，应该花点精力研究下HTTP状态码（请参见“HTTP状态码”）。编写服务器端代码包括考虑要如何与客户端进行通信并给它们发送尽可能多的信息用来帮助客户端理解服务器发送的响应。

HTTP状态码

HTTP规范规定了大量服务器能返回给请求客户端的状态码。我们可以从

[Wikipedia \(http://en.wikipedia.org/wiki/List_of_HTTP_status_codes \)](http://en.wikipedia.org/wiki/List_of_HTTP_status_codes)上更深入地了解它们。

虽然有大量的状态码，但你会发现在应用中我们只会使用常见的几种：

- [200 OK](#)——一切正常。
- [301 Moved Permanently](#)——请求的URL已被移走，客户端应该重新请求响应中指定的URL。
- [400 Bad Request](#)——客户端请求的格式是无效的，需要修复。
- [401 Unauthorized](#)——客户端没有权限查看请求的资源，它应该先验证请求后再试。
- [403 Forbidden](#)——无论出于何种原因，服务器拒绝处理这个请求。这和401不一样，401中客户端在验证通过后可以重试。
- [404 Not Found](#)——客户端请求的资源不存在。
- [500 Internal Server Error](#)——某些情况发生导致服务器无法处理请求。通常这个错误代表代码已经处于某种不一致的状态或出

现bug，需要开发人员关注。

■ [503 Service Unavailable](#)——这表明某种运行时错误，例如内存暂时不足或网络资源出现问题。它和500一样都是致命的错误，但它也表明客户端可以过段时间后再次尝试。

这些状态码都是我们最常用到的，但还是有很多其他情况会在浏览时遇到。如果不确定何时使用，可以看看已有的代码是如何处理的。为特定情况给出正确的响应码可能会有很多讨论，但通常我们都能够使用正确的状态码。

4.5 提高灵活性：GET参数

当开始往相册中添加大量照片时，应用需要在一个“页面”中高效地展示大量照片，所以我们需要为应用提供分页功能，而客户端需要能够告诉我们需要多少张照片以及是哪一页的照片，就像这样：

```
curl -X GET 'http://localhost:8080/albums/italy2012.json?page=1&page_size=20'
```

如果你对这些术语还不熟悉，那这里就稍微解释下，上面代码中的URL加粗部分是查询字符串，通常作为请求的GET参数。如果我们在先前版本的程序上运行这个curl命令，会发现它一点用处都没有。如果把下面的代码添加到handle_incoming_request函数的开头部分，就会知道原因：

```
console.log(req.url);
```

现在URL显示如下：

```
/albums/italy2012.json?page=1&page_size=20
```

该代码会在字符串末尾而不是在中间查找.json，如果要修复代码来处理分页，需要做三件事情：

- 1) 修改handle_incoming_request函数，让它能够正确解析URL。
- 2) 解析查询字符串，获得page和page_size对应的值。
- 3) 修改load_album函数以支持这些参数。

幸运的是，我们可以一举把前两件事情完成。当添加Node内置的url模块后，可以使用url.parse函数来提取核心的URL路径名以及查询参数。url.parse函数可以更进一步给函数添加第二个参数——true，这个参数告诉url.parse函数解析查询字符串，并生成包含GET参数的对象。如果我们使用url.parse解析前面的URL并打印结果，会看到如下结果：

```
{ search: '?page=1&page_size=20',
query: { page: '1', page_size: '20' },
pathname: '/albums/italy2012.json',
path: '/albums/italy2012.json?page=1&page_size=20',
href: '/albums/italy2012.json?page=1&page_size=20' }
```

现在可以修改handle_incoming_request函数来解析URL，并将parsed_url存储到请求对象中。函数现在修改如下：

```
function handle_incoming_request(req, res) {
  req.parsed_url = url.parse(req.url, true);
  var core_url = req.parsed_url.pathname;

  // test this fixed url to see what they're asking for
  if (core_url == '/albums.json') {
    handle_list_albums(req, res);
  } else if (core_url.substr(0, 7) == '/albums'
             && core_url.substr(core_url.length - 5) == '.json') {
    handle_get_album(req, res);
  } else {
    send_failure(res, 404, invalid_resource());
  }
}
```

最后，我们需要修改handle_get_album函数来查找页面和page_num查询参数。当没有提供传入值或者值不合法时，可以为查询参数设置合理的默认值（服务器应该总是认为传入的值非常危险，需要对其进行仔细的检查）。

```
function handle_get_album(req, res) {
  // get the GET params
  var getp = req.parsed_url.query;
  var page_num = getp.page ? getp.page : 0;
  var page_size = getp.page_size ? getp.page_size : 1000;

  if (isNaN(parseInt(page_num))) page_num = 0;
  if (isNaN(parseInt(page_size))) page_size = 1000;

  // format of request is /albums/album_name.json
```

```

var core_url = req.parsed_url.pathname;

var album_name = core_url.substr(7, core_url.length - 12);
load_album(
    album_name,
    page_num,
    page_size,
    function (err, album_contents) {
        if (err && err.error == "no_such_album") {
            send_failure(res, 404, err);
        } else if (err) {
            send_failure(res, 500, err);
        } else {
            send_success(res, { album_photos: album_contents });
        }
    }
);
}

```

最后，修改load_album函数。当一切都顺利时，它会提取only_files数组的子数组：

```

function load_album(album_name, page, page_size, callback) {
    fs.readdir(
        "albums/" + album_name,
        function (err, files) {
            if (err) {
                if (err.code == "ENOENT") {
                    callback(no_such_album());
                } else {
                    callback({ error: "file_error",
                               message: JSON.stringify(err) });
                }
                return;
            }

            var only_files = [];
            var path = "albums/" + album_name + "/";

            (function iterator(index) {
                if (index == files.length) {
                    var ps;
                    // slice fails gracefully if params are out of range
                    ps = only_files.splice(page * page_size, page_size);
                    var obj = { short_name: album_name,
                               photos: ps };
                    callback(null, obj);
                    return;
                }

                fs.stat(
                    path + files[index],
                    function (err, stats) {
                        if (err) {
                            callback({ error: "file_error",
                                       message: JSON.stringify(err) });
                            return;
                        }
                        if (stats.isFile()) {
                            var obj = { filename: files[index], desc: files[index] };
                            only_files.push(obj);
                        }
                        iterator(index + 1)
                    }
                );
            })();
        })(0);
    }
}

```

4.6 修改内容：POST数据

目前，本章已经大体介绍了如何从JSON服务器中获取需要的东西，或许你已经开始想要给服务器发送数据、创建新的东西或者是修改已有的东西。通常这需要通过HTTP POST数据来完成，我们可以
通过许多不同的格式来发送数据。为了能够使用curl客户端发送数
据，我们需要做点事情：

- 1) 设置HTTP方法参数为POST (或者PUT) 。
- 2) 为传入的数据设置Content-Type。
- 3) 开始发送数据。

我们可以轻松地使用curl完成这些任务。首先，需要简单地修改
方法的名字；其次，使用-H参数指定curl中的HTTP请求头；最后一步
有许多方法可以实现，但在这里，我们使用-d参数将JSON写为字
符串。

现在为服务器添加新功能，从而允许我们重命名相册。确保URL
的格式如下所示，并指定它必须是一个POST请求：

```
http://localhost:8080/albums/albumname/rename.json
```

因此，重命名相册的curl命令看起来如下所示：

```
curl -s -X POST -H "Content-Type: application/json" \  
      -d '{ "album_name" : "new album name" }' \  
      http://localhost:8080/albums/old_album_name/rename.json
```

修改handle_incoming_request函数以接收新的请求类型，这
很简单：

```
function handle_incoming_request(req, res) {  
    // parse the query params into an object and get the path  
    // without them. (2nd param true = parse the params).
```

```
req.parsed_url = url.parse(req.url, true);
var core_url = req.parsed_url.pathname;

// test this fixed url to see what they're asking for
if (core_url == '/albums.json' && req.method.toLowerCase() == 'get') {
    handle_list_albums(req, res);
} else if (core_url.substr(core_url.length - 12) == '/rename.json'
    && req.method.toLowerCase() == 'post') {
    handle_rename_album(req, res);
} else if (core_url.substr(0, 7) == '/albums'
    && core_url.substr(core_url.length - 5) == '.json'
    && req.method.toLowerCase() == 'get') {
    handle_get_album(req, res);
} else {
    send_failure(res, 404, invalid_resource());
}
}
```

注意，我们必须将处理重命名请求的代码放置在加载相册请求的代码前面；否则，代码可能会认为它是一个叫做rename的相册，从而执行handle_get_album函数。

4.6.1 接收JSON POST数据

程序为了获取POST数据，会使用一种叫做数据流的Node特性。当使用Node的异步非阻塞特性时，数据流是传输大量数据的最佳方式。这会在第6章详细介绍，而现在我们只需知道使用数据流的主要方式：

```
.on(event_name, function (parm) { ... });
```

现在尤其需要关注两个事件：readable和end事件。数据流实际上是来自于http模块的ServerRequest对象（继承自Stream类；ServerResponse也一样），我们可以通过以下方式监听这两个事件：

```

var json_body = '';
req.on(
  'readable',
  function () {
    var d = req.read();
    if (d) {
      if (typeof d == 'string') {
        json_body += d;
      } else if (typeof d == 'object' && d instanceof Buffer) {
        json_body += d.toString('utf8');
      }
    }
  }
);

req.on(
  'end',
  function () {
    // did we get a valid body?
    if (json_body) {
      try {
        var body = JSON.parse(json_body);
        // use it and then call the callback!
        callback(null, ...);
      } catch (e) {
        callback({ error: "invalid_json",
                   message: "The body is not valid JSON" });
      } else {
        callback({ error: "no_body",
                   message: "We did not receive any JSON" });
      }
    }
  }
);

```

对于传入请求的主体中的每一块（组块）数据，传入on('readable',...)的处理程序函数都会被调用。上面的代码中，我们首先通过read方法读取来自数据流的数据并将这些传入的数据添加到json_body变量的后面；然后，当监听到end事件，会得到这些结果字符串，并尝试去解析它。当给定的字符串不是一个合法的JSON时，JSON.parse会抛出一个错误，所以必须用try/catch语句块封装这些代码。

处理重命名请求的函数代码如下所示：

```

function handle_rename_album(req, res) {

    // 1. Get the album name from the URL
    var core_url = req.parsed_url.pathname;
    var parts = core_url.split('/');
    if (parts.length != 4) {
        send_failure(res, 404, invalid_resource(core_url));
        return;
    }

    var album_name = parts[2];

    // 2. get the POST data for the request. this will have the JSON
    // for the new name for the album.
    var json_body = '';
    req.on(
        'readable',
        function () {
            var d = req.read();
            if (d) {
                if (typeof d == 'string') {
                    json_body += d;
                } else if (typeof d == 'object' && d instanceof Buffer) {

                    json_body += d.toString('utf8');
                }
            }
        }
    );
    // 3. when we have all the post data, make sure we have valid
    // data and then try to do the rename.
    req.on(
        'end',
        function () {
            // did we get a body?
            if (json_body) {
                try {
                    var album_data = JSON.parse(json_body);
                    if (!album_data.album_name) {
                        send_failure(res, 403, missing_data('album_name'));
                        return;
                    }
                } catch (e) {
                    // got a body, but not valid json
                    send_failure(res, 403, bad_json());
                    return;
                }
            }

            // 4. Perform rename!
            do_rename(
                album_name,           // old
                album_data.album_name, // new
                function (err, results) {
                    if (err && err.code == "ENOENT") {
                        send_failure(res, 403, no_such_album());
                        return;
                    } else if (err) {
                        send_failure(res, 500, file_error(err));
                        return;
                    }
                    send_success(res, null);
                }
            );
        }
    );
}

```

更新后的服务器的完整代码可以处理三种请求，将其保存为post_data.js放在GitHub仓库的Chapter4文件夹下。注意，低于0.10版本的node运行这个程序会出错。我会在第6章详细解释node都做了哪些修改。

4.6.2 接收表单POST数据

虽然现在的应用会尽量不去使用这种方式，但是在Web应用中，还是有大量的数据通过<form>元素提交到服务器，例如：

```
<form name='simple' method='post' action='http://localhost:8080'>
  Name: <input name='name' type='text' size='10' /><br/>
  Age: <input name='age' type='text' size='5' /><br/>
  <input type='submit' value="Send"/>
</form>
```

如果写一个小的服务器程序，如前一节使用readable和end事件来取得POST数据，表单生成的数据打印出来会是：

```
name=marky+mark&age=23
```

然而，实际上我们需要的与之前通过JSON获取的数据很相似：将发送给我们的数据整理成JavaScript对象。为了实现这个目的，可以使用另外一个Node.js内置的模块querystring，特别是这个模块的解析函数parse，如下所示：

```
var POST_data = qs.parse(body);
```

结果和期望的一样，如下所示：

```
{ name: 'marky mark++', age: '23' }
```

接收表单并打印其中内容，该服务器的完整代码清单如下所示：

```
var http = require('http'), qs = require('querystring');

function handle_incoming_request(req, res) {
    var body = '';
    req.on(
        'readable',
        function () {
            var d = req.read();
            if (d) {
                if (typeof d == 'string') {
                    body += d;
                } else if (typeof d == 'object' && d instanceof Buffer) {
                    body += d.toString('utf8');
                }
            }
        }
    );
    req.on(
        'end',
        function () {
            if (req.method.toLowerCase() == 'post') {
                var POST_data = qs.parse(body);
                console.log(POST_data);

            }
            res.writeHead(200, { "Content-Type" : "application/json" });
            res.end(JSON.stringify( { error: null } ) + "\n");
        }
    );
}

var s = http.createServer(handle_incoming_request);
s.listen(8080);
}
```

但是在第6章的开头部分，当了解到如何使用Node的express Web应用框架之后，我们会发现这些功能早就为我们准备好了。

4.7 小结

本章介绍了许多新知识。我们编写了一个简单的JSON服务器作为我们的第一个Web应用。我喜欢这种方式的主要原因是：

- 它让我们聚焦于服务器，以及能够熟练使用Node.js所需要的核心概念。
- 能够确保服务器的API拥有良好的架构和效率。
- 一个好的、轻量的应用服务器应该是指运行它的计算机能够无碍地处理请求。当我们在后面添加HTML UI以及客户端特性时，可以尝试让客户端做尽可能多的工作。

现在，我们不仅拥有一个基本的服务器，而且看到如何根据不同的请求以及查询参数修改响应输出。我们也看到了如何发送新数据或者通过提交POST数据来修改已经存在的数据。虽然部分程序看起来非常冗长和复杂，但现在我们是使用最基本的模块来验证Node的核心原理。很快我们会开始将这些代码替换成更实用的、功能更丰富的模块。

但首先，让我们稍事休息并深入了解一下模块——更多的是如何使用模块并自己编写模块。

第5章 模块化

到目前为止，我们编写的Node.js服务器和脚本已经通过模块的形式使用了一些外部提供的功能。在本章中，我会解释这些模块是如何工作的，并告诉你如何自己开发模块。除了Node为我们提供了功能强大并且实用的模块之外，还有一个大型社区在不断地开发各种模块，我们可以在程序中好好利用这些模块。而实际上，我们甚至可以自己编写模块来实现某些功能。

Node的一个重要特性就是我们无需真正地辨别模块到底是我们自己编写的还是从外部仓库中获取的，比如本章中通过Node包管理器（Node Package Manager，npm）看到的那些模块。当在Node中编写类和函数时，确保它们大致拥有相同的格式——或许还有一些说明和文档——这与从互联网上下载并使用的模块一模一样。实际上，通常只需要一个额外的JSON文件以及一两行代码，别人就能获取并使用我们的代码！

Node附带了许多内置模块，这些模块都被打包在系统的node可执行文件中。如果从node.js网站上下载Node源代码，就可以查看它们的源文件，它们都在lib/子文件夹下。

5.1 编写简单模块

从更高层面上讲，模块是Node.js对常用功能进行分组的方式。例如，如果我们有一个为特定的数据库服务器工作的函数库或类库，那么将代码提交到模块中并将其打包使用会非常有意义。

虽然模块不会太简单，但实际上Node.js中的每个文件都是一个模块。我们可以将多个文件、单元测试、文档和其他支持文件放在文件夹中，并打包成复杂的模块。还可以通过模块中的单个JavaScript文件来使用它们（见本章后续章节）。

如果要自己编写一个模块，该模块暴露或者提供一个叫做hello_world的函数，可以编写如下代码，并保存为mymodule.js：

```
exports.hello_world = function () {
    console.log("Hello World");
}
```

exports对象是一个特殊的对象，在每个我们创建的文件中由Node模块系统创建，当引入这个模块时，会作为require函数的值返回。它被封装在每个模块的module对象中，用来暴露函数、变量或者类。在这个简单的例子中，模块在exports对象上暴露了一个函数，要使用它，可以编写以下代码，并保存到modtest.js文件中：

```
var mm = require ('./mymodule');
mm.hello_world();
```

运行node modtest.js，Node会打印出预期的结果"Hello World"。我们可以通过exports对象暴露任何我们想要暴露的函数和类。例如：

```
function Greeter (lang) {
    this.language = lang;
    this.greet() = function () {
        switch (this.language) {
            case "en": return "Hello!";
            case "de": return "Hallo!";
            case "jp": return "こんにちは!";
            default: return "No speaka that language";
        }
    }
}

exports.hello_world = function () {
    console.log("Hello World");
}

exports.goodbye = function () {
    console.log("Bye bye!");
}

exports.create_greeter = function (lang) {
    return new Greeter(lang);
}
```

每个模块的module变量会包含许多信息，例如当前模块的文件名、拥有的子模块和对应的父模块等。

模块和对象

当频繁地从模块中返回对象时，我们会发现有两种适用的核心模式。

工厂模式

前面的简单示例中包含了一个叫做Greeter的类。为了获取Greeter对象的实例，需要调用创建函数（或者是工厂函数）来创建并返回这个类的实例。基本模式如下：

```
function ABC (parms) {
    this.varA = ...;
    this.varB = ...;
    this.functionA = function () {
        ...
    }
}

exports.create_ABC = function (parms) {
    return new ABC(parms);
}
```

这种模式的优点是模块可以通过exports对象暴露其他的函数和类。

构造函数模式

另外一种让编写的模块暴露类的方式是完全把模块的exports对象替换成想让别人使用的类：

```
function ABC () {
    this.varA = 10;
    this.varB = 20;
    this.functionA = function (var1, var2) {
        console.log(var1 + " " + var2);
    }
}

module.exports = ABC;
```

为了使用该模块，需要修改代码，如下所示：

```
var ABCClass = require('./conmod2');
var obj = new ABCClass();
obj.functionA(1, 2);
```

因此，模块真正唯一暴露的是一个类的构造函数。这种方式非常不错，并且具有面向对象编程的风格，但它也有个缺点，就是不能让模块暴露更多的东西。因此，在Node中使用这种方式会觉得有些尴尬。我在这里将其展示出来，是为了当看到这种模式时能够清楚它到底

底是什么，但我们几乎不会在本书中或者项目中使用到它——通常会使用工厂模式。

5.2 npm : Node包管理器

除了编写自己的模块和使用Node.js提供的模块，我们还会频繁地使用Node社区其他人编写并发布到互联网上的代码。现今最常用的做法是使用npm（Node包管理器）。npm会同node安装程序一起被安装（请参见第1章），可以使用命令行并输入npm help来验证npm是否成功安装并正常运行。

要通过npm安装模块，需要使用npm install命令。这里只需提供想要安装的模块包名称。许多npm模块将源代码托管在github.com上，所以通常它们只会告诉你需要的模块名，例如：

```
Kimidori:pl marcw$ npm install mysql
npm http GET https://registry.npmjs.org/mysql
npm http 200 https://registry.npmjs.org/mysql
npm http GET https://registry.npmjs.org/mysql/-/mysql-2.0.0-alpha4.tgz
npm http 200 https://registry.npmjs.org/mysql/-/mysql-2.0.0-alpha4.tgz
npm http GET https://registry.npmjs.org/require-all/0.0.3
npm http 200 https://registry.npmjs.org/require-all/0.0.3
npm http GET https://registry.npmjs.org/require-all/-/require-all-0.0.3.tgz
npm http 200 https://registry.npmjs.org/require-all/-/require-all-0.0.3.tgz
mysql@2.0.0-alpha4 node_modules/mysql
└── require-all@0.0.3
```

如果不确定想要安装的包名，可以使用npm search命令，如下所示：

```
npm search sql
```

该命令将打印出所有符合条件的模块的名字和描述。

npm将模块包安装到项目的node_modules/子目录下。如果一个模块包本身包含依赖，那么这些依赖都会被安装到这个模块所在文件夹下的node_modules/子目录下。

```
+ project/
  + node_modules/
    module1
    module2
      + node_modules
        dependency1
main.js
```

如果想查看某个项目当前使用的所有模块清单，可以使用npm ls命令：

```
Kimidori:handlers marcw$ npm ls  
/Users/marcw/src/scratch/project  
├─ async@0.1.22  
├─ bcrypt@0.7.3  
| └─ bindings@1.0.0  
└─ mysql@2.0.0-alpha4  
    └─ require-all@0.0.3
```

要想更新一个已经安装的包到新版本，可以使用npm update命令。如果指定了一个包名字，那么只会更新这个包。如果没有指定包名字，这个命令会更新所有包到最新版本：

```
Kimidori:p1 marcw$ npm update mysql  
npm http GET https://registry.npmjs.org/mysql  
npm http 304 https://registry.npmjs.org/mysql  
Kimidori:p1 marcw$ npm update  
npm http GET https://registry.npmjs.org/bcrypt  
npm http GET https://registry.npmjs.org/async  
npm http GET https://registry.npmjs.org/mysql  
npm http 304 https://registry.npmjs.org/mysql  
npm http 304 https://registry.npmjs.org/bcrypt  
npm http 304 https://registry.npmjs.org/async  
  
npm http GET https://registry.npmjs.org/require-all/0.0.3  
npm http GET https://registry.npmjs.org/bindings/1.0.0  
npm http 304 https://registry.npmjs.org/require-all/0.0.3  
npm http 304 https://registry.npmjs.org/bindings/1.0.0
```

5.3 使用模块

正如我们之前看到的，要在编写的Node文件中引入模块，需要使用require函数。为了能够引用模块里的函数和（或）类，需要将结果（被加载的模块的exports对象）赋值给一个变量：

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

引入的模块对于引入它们的模块是私有的，所以，如果a.js加载http模块，那么b.js是无法引用这个模块的，除非b.js自己也加载http模块。

5.3.1 查找模块

Node.js使用一组相当简单的规则来查找require函数请求的模块：

- 1) 如果请求的是内置模块——例如http或者fs——Node会直接使用这些模块。
- 2) 如果require函数的模块名以路径符开始（如./、../或者/），那么Node会在指定的目录中查找模块并尝试去加载它。如果没有在模块名中指定.js扩展名，Node会首先查找基于同名文件夹的模块。如果没有找到，它会添加扩展名.js、.json和.node，并依次尝试加载这些类型的模块（带有.node扩展名的模块会被编译成附加模块）。
- 3) 如果模块名开始时没有路径符，Node会在当前文件夹的node_modules/子文件夹下查找模块。如果找到，则加载该模块；否则，Node会以自己的方式在当前位置的路径树下搜寻node_modules/文件夹。如果依然失败，它会在一些默认地址下进行搜寻，例如/usr/lib和/usr/local/lib文件夹。
- 4) 如果在以上任何一个位置都没有找到模块，则抛出错误。

5.3.2 模块缓存

当模块从指定的文件或者目录上加载之后，Node.js会将它缓存。之后所有的require调用将会从相同的地址加载相同的模块——而这些模块已经初始化或者做过其他工作。这相当有趣，因为有时候两个文件中都尝试加载一个指定的模块，但得到的可能并不是同一个！考虑下面的项目结构：

```
+ project/
  + node_modules/
    + special_widget/
      + node_modules/
        mail_widget (v2.0.1)

        mail_widget (v1.0.0)
      main.js
      utils.js
```

这个例子中，如果main.js或者utils.js都引入mail_widget，由于Node的搜索规则，它会在项目的node_modules/子目录下发现该模块，最终获取mail_widget的1.0.0版本。但是，如果引入special_widget，而这个模块也希望使用mail_widget，special_widget会获取它自己私有的mail_widget版本，2.0.1版本在它自己的node_modules/文件夹下。

这是Node.js模块系统最强大和神奇的特性之一。许多其他的系统、模块、widget或动态库都集中存储在一个位置，当请求的包本身也需要请求其他不同版本的模块时，版本控制就成了梦魇。而在Node中，可以自由地引入其他不同版本的模块，Node的命名空间和模块规则意味着它们之间不会互相干扰！独立的模块和项目部分内容都可以自由地引入、更新或修改引入的模块，因为它们只能看到自己那部分而不影响系统其他部分。

简而言之，Node.js非常直观，这也许是您有生以来第一次无需坐在那里无休止地咒骂正在使用的包管理系统。

5.3.3 循环

考虑下面的情况：

- a.js引入b.js。
- b.js引入a.js。
- main.js引入a.js。

很显然，我们会发现上面的模块中存在一个循环。当Node探测到一个尚未初始化的返回模块时，它会阻止循环发生。在上面的示例中，会发生下面的事情：

- main.js被加载，运行引入a.js的代码。
- a.js被加载，运行引入b.js的代码。
- b.js被加载，运行引入a.js的代码。
- Node探测到循环并返回一个指向a.js的对象，但不会再执行其他代码——在这一刻，a.js的加载和初始化并没有完成。
- b.js、a.js和main.js都完成初始化（按顺序），然后b.js和a.js的引用都有效和可用。

5.4 编写模块

Node.js中每个文件都是一个含有module和exports对象的模块。但是我们也要清楚，模块也可以很复杂：包含一个目录用来保存模块的内容和一个含有包信息的文件。如果想要编写一系列支持文件，并将模块的功能拆散到多个JavaScript文件中，甚至包含单元测试，可以使用下面这种格式来编写模块。

基本格式如下：

- 1) 创建一个文件夹来存放模块内容。
- 2) 添加名为package.json的文件到文件夹中。文件至少包含当前模块的名字和一个主要的JavaScript文件——用来一开始加载该模块。
- 3) 如果Node没有找到package.json文件或者没有指定主JavaScript文件，那它就会查找index.js（或者是编译后的附加模块index.node）。

5.4.1 创建模块

现在我们可以将在前面章节中为管理照片和相册而编写的代码提取出来并放到一个模块中。这样做就可以分享这些模块，以供将来编写的其他项目使用，并对代码进行隔离，因此也可以编写测试用例，等等。

首先，在源代码目录下（即~/src/scratch或其他运行Node的地方）创建如下目录结构：

```
+ album_mgr/  
  + lib/  
  + test/
```

在album_mgr文件夹下，创建一个叫做package.json的文件，

并编写如下的代码：

```
{ "name": "album-manager",
  "version": "1.0.0",
  "main": "./lib/albums.js" }
```

这是个最基本的package.json文件。它告诉npm当前这个包有一个叫做album-manager的名字，这个包的“默认”或开始JavaScript文件是存放在lib/子目录下的albums.js。

上面的目录结构没有强制性。这是包的通用布局中最简单的一种，我发现它很有用，因此一直使用它，但没有义务遵循它。但是，我还是建议以这种方式开始工作，最好对整个系统都非常熟悉之后，再开始尝试其他方式。

用来托管Node模块源代码的诸如github.com的网站如果发现有Readme文档，会自动地渲染并展示Readme文档。因此，通常都会引入一个Readme.md文件（"md"代表markdown，是github.com使用的标准文档格式）。强烈建议为模块编写一个文档用来帮助人们开始使用它。对于album-manager模块，编写如下的Readme文件：

```
# Album-Manager

This is our module for managing photo albums based on a directory. We
assume that, given a path, there is an albums sub-folder, and each of
its individual sub-folders are themselves the albums. Files in those
sub-folders are photos.

## Album Manager

The album manager exposes a single function, `albums` , which returns
an array of `Album` objects for each album it contains.

## Album Object

The album object has the following two properties and one method:
* `name` -- The name of the album
* `path` -- The path to the album
* `photos()` -- Calling this method will return all the album's photos
```

现在我们可以开始编写真正的模块文件。首先，从约定的lib/albums.js开始，它含有第4章的album-loading的部分代码，把代码重新打包成一个类似模块的JavaScript文件：

```

var fs = require('fs'),
    album = require('./album.js');

exports.version = "1.0.0";

exports.albums = function (root, callback) {
    // we will just assume that any directory in our 'albums'
    // subfolder is an album.
    fs.readdir(
        root + "/albums",
        function (err, files) {
            if (err) {
                callback(err);
                return;
            }

            var album_list = [];

            (function iterator(index) {
                if (index == files.length) {
                    callback(null, album_list);
                    return;
                }

                fs.stat(
                    root + "albums/" + files[index],
                    function (err, stats) {
                        if (err) {
                            callback({ error: "file_error",
                                      message: JSON.stringify(err) });
                            return;
                        }
                        if (stats.isDirectory()) {
                            var p = root + "albums/" + files[index];
                            album_list.push(album.create_album(p));
                        }
                        iterator(index + 1)
                    }
                );
            })(0);
        }
    );
};

```

版本字段是模块提供的标准功能之一。虽然我并不经常使用它，但通过调用模块来检查版本并根据不同的版本执行不同的代码还是很有用的。

我们可以看到相册的功能被拆分到一个叫做/album.js的新文件中，其中有一个叫做Album的新类。这个类的代码如下所示：

```

function Album (album_path) {
    this.name = path.basename(album_path);
    this.path = album_path;
}

Album.prototype.name = null;
Album.prototype.path = null;
Album.prototype._photos = null;

Album.prototype.photos = function (callback) {
    if (this._photos != null) {
        callback(null, this._photos);
        return;
    }

    var self = this;

    fs.readdir(
        self.path,
        function (err, files) {
            if (err) {
                if (err.code == "ENOENT") {
                    callback(no_such_album());
                } else {
                    callback({ error: "file_error",
                               message: JSON.stringify(err) });
                }
                return;
            }

            var only_files = [];

            (function iterator(index) {
                if (index == files.length) {
                    self._photos = only_files;
                    callback(null, self._photos);
                    return;
                }
            })

            fs.stat(
                self.path + "/" + files[index],
                function (err, stats) {
                    if (err) {
                        callback({ error: "file_error",
                                   message: JSON.stringify(err) });
                        return;
                    }
                    if (stats.isFile()) {
                        only_files.push(files[index]);
                    }
                    iterator(index + 1)
                }
            );
        })(0);
    );
};

}

```

如果你对上面源代码中已经使用过几次的prototype关键字感到困惑，那或许应该跳回第2章，重新回顾如何编写JavaScript类。这里的prototype关键字是为Album类的所有实例设置属性的方法。

再次说明，上面的代码与第4章中基本的JSON服务器非常相似。它们之间真正的区别在于——它被包装成一个含有原型对象以及photos方法的类。

我希望你注意到以下两件事情：

1) 我们正在使用一个叫做path的新内置模块，并且使用它的basename函数用来从路径中提取相册名。

2) 正如在第3章中所提到的，非阻塞的异步IO和this指针会有一些小问题，所以需要使用var self=this小技巧来帮助我们记住对象的引用。

album.js剩下的部分就很简单了，如下所示：

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

// Album class code goes here
exports.create_album = function (path) {
    return new Album(path);
};

function no_such_album() {
    return { error: "no_such_album",
            message: "The specified album does not exist" };
}
```

这就是我们需要为album-manager模块做的所有事情！如果要测试它，回到根目录并输入下面的测试程序，保存为atest.js文件：

```
var amgr = require('./album_mgr');

amgr.albums('../', function (err, albums) {
  if (err) {
    console.log("Unexpected error: " + JSON.stringify(err));
    return;
  }

  (function iterator(index) {
    if (index == albums.length) {
      console.log("Done");
      return;
    }

    albums[index].photos(function (err, photos) {
      if (err) {
        console.log("Err loading album: " + JSON.stringify(err));
        return;
      }

      console.log(albums[index].name);
      console.log(photos);
      console.log("");
      iterator(index + 1);
    });
  })(0);
});
```

现在，所需要做的事情就是确保在当前目录下有一个albums/子文件夹，运行atest.js文件，我们会看到类似下面的结果：

```
Kimidori:Chapter05 marcw$ node atest
australia2010
[ 'aus_01.jpg',
  'aus_02.jpg',
  'aus_03.jpg',
  'aus_04.jpg',
  'aus_05.jpg',
  'aus_06.jpg',
  'aus_07.jpg',
  'aus_08.jpg',
  'aus_09.jpg' ]

italy2012
[ 'picture_01.jpg',
  'picture_02.jpg',
  'picture_03.jpg',
  'picture_04.jpg',
  'picture_05.jpg' ]

japan2010
```

```
[ 'picture_001.jpg',
  'picture_002.jpg',
  'picture_003.jpg',
  'picture_004.jpg',
  'picture_005.jpg',
  'picture_006.jpg',
  'picture_007.jpg' ]
```

Done

5.4.2 使用模块进行开发

现在我们已经有一个用于相册的模块。如果想在多个项目中使用它，可以把它拷贝到其他项目的node_modules/文件夹下，但这样会遇到一个问题：当想要对相册模块做一些修改时究竟会发生什么？真的需要拷贝源代码到使用它的所有位置中并且每次都去修改它吗？

幸运的是，npm在这里派上用场了。可以修改package.json文件，增加下面的代码：

```
{ "name": "album-manager",
  "version": "1.0.0",
  "main": "./lib/albums.js",
  "private": true }
```

这些代码告诉npm不能将现在还不想发布的模块发布到外部的npm源中。

接下来使用npm link命令，这个命令会告诉npm创建一个链接，指向当前机器默认公开包库的album-manager包（例如Linux和Mac机器中的/usr/local/lib/node_modules，或Windows中的C:\Users\username\AppData\location\npm）。

```
Kimidori:Chapter05 marcw$ cd album_mgr
Kimidori:album_mgr marcw$ sudo npm link
/usr/local/lib/node_modules/album-manager ->
/Users/marcw/src/scratch/Chapter05/album_mgr
```

注意，这些都依赖于本地机器的权限以及其他设置，有可能需要

使用sudo作为超级用户来运行命令。

现在，要使用这个模块，还需做两件事情：

- 1) 在代码中引用'album-manager'而不是'album_mgr'（因为npm使用了package.json的name字段）。
- 2) 通过npm为每个想使用这个模块的项目创建一个到album-manager模块的引用。可以只输入npm link album-manager：

```
Kimidori:Chapter05 marcw$ mkdir test_project
Kimidori:Chapter05 marcw$ cd test_project/
Kimidori:test_project marcw$ npm link album-manager
/Users/marcw/src/scratch/Chapter05/test_project/node_modules/album-manager ->
/usr/local/lib/node_modules/album-manager ->
/Users/marcw/src/scratch/Chapter05/album_mgr
Kimidori:test_project marcw$ dir
drwxr-xr-x  3 marcw  staff  102 11 20 18:38 node_modules/
Kimidori:test_project marcw$ dir node_modules/
lrwxr-xr-x  1 marcw  staff   41 11 20 18:38 album-manager ->
/usr/local/lib/node_modules/album-manager
```

现在，我们可以随意地修改最初的相册管理源代码，而所有引用这个模块的项目都可以立即看到这些修改。

5.4.3 发布模块

如果编写了一个模块，并想要把它共享给其他的用户，可以使用npm publish把它发布到官方的npm模块注册中心。这需要做以下几件事情：

- 删除package.json文件的"private":true这一行。
- 在npm注册服务器上使用npm adduser命令创建一个账户。
- 可以选择向package.json文件中添加更多字段（运行npm help json可以获取更多关于想添加的字段信息），例如描述、作者的联系信息和托管网站等。
- 最后，在模块的目录上运行npm publish命令，把模块发布到npm。

```
Kimidori:album_mgr marcw$ npm adduser
Username: marcwan
Password:
Email: marcwan@example.org
npm http PUT https://registry.npmjs.org/-/user/org.couchdb.user:marcwan
npm http 201 https://registry.npmjs.org/-/user/org.couchdb.user:marcwan
Kimidori:album_mgr marcw$ npm publish
npm http PUT https://registry.npmjs.org/album-manager
npm http 201 https://registry.npmjs.org/album-manager
npm http GET https://registry.npmjs.org/album-manager
npm http 200 https://registry.npmjs.org/album-manager
npm http PUT https://registry.npmjs.org/album-manager/1.0.0/-tag/latest
npm http 201 https://registry.npmjs.org/album-manager/1.0.0/-tag/latest
npm http GET https://registry.npmjs.org/album-manager
npm http 200 https://registry.npmjs.org/album-manager
npm http PUT https://registry.npmjs.org/album-manager/-/album-manager-1.0.0.tgz/
-rev/2-7f175fa335728e1cde4ce4334696bd1a
npm http 201 https://registry.npmjs.org/album-manager/-/album-manager-1.0.0.tgz/
-rev/2-7f175fa335728e1cde4ce4334696bd1a
+ album-manager@1.0.0
```

如果不小心发布了不想要发布的模块，或者想从npm模块注册中心中移除模块，可以运行npm unpublish命令：

```
Kimidori:album_mgr marcw$ npm unpublish
npm ERR! Refusing to delete entire project.
npm ERR! Run with --force to do this.
npm ERR! npm unpublish <project>[@<version>]
npm ERR! not ok code 0
Kimidori:album_mgr marcw$ npm unpublish --force
npm http GET https://registry.npmjs.org/album-manager

npm http 200 https://registry.npmjs.org/album-manager
npm http DELETE https://registry.npmjs.org/album-manager/-rev/
3-fa97bb84falcb8d6d6a3f57ad4a2cf2f
npm http 200 https://registry.npmjs.org/album-manager/-rev/
3-fa97bb84falcb8d6d6a3f57ad4a2cf2f
- album-manager@1.0.0
```

5.5 应当内置的通用模块

目前为止，我们已经在编写的代码中使用过不少Node.js的内置模块（`http`、`fs`、`path`、`querystring`和`url`），而且还会在本书剩下部分里使用更多的内置模块。但是，有一个模块，它可以从npm上安装，并且几乎每个编写的独立项目中都会使用到它。因此，需要花一节的篇幅在这里介绍它。

5.5.1 常见问题

考虑这种情况，我们想要编写一些异步的代码：

- 打开路径的句柄。
- 判断路径是否指向一个文件。
- 如果文件指向一个文件，加载这个文件的内容。
- 关闭文件句柄并将内容返回给调用者。

这种代码我们之前也见过，函数如下所示。被调用函数的字体加粗，回调函数的字体设置为加粗且斜体：

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

function load_file_contents(path, callback) {
    fs.open(path, 'r', function (err, f) {
        if (err) {
            callback(err);
            return;
        } else if (!f) {
            callback({ error: "invalid_handle",
                      message: "bad file handle from fs.open"});
            return;
        }
        fs.fstat(f, function (err, stats) {
            if (err) {
                callback(err);
                return;
            }
            if (stats.isFile()) {
                var b = new Buffer(10000);
                fs.read(f, b, 0, 10000, null, function (err, br, buf) {
                    if (err) {
                        callback(err);
                    } else {
                        callback(null, b);
                    }
                });
            }
        });
    });
}
```

```
        return;
    }
    fs.close(f, function (err) {
        if (err) {
            callback(err);
            return;
        }
        callback(null, b.toString('utf8', 0, br));
    });
});
}};

} else {
    callback({ error: "not_file",
        message: "Can't load directory" });
    return;
}
});
});
}
}
```

我们可以发现，即使在上示简单的例子中，代码从一开始就深度嵌套。当嵌套层数超过一定程度时，就会发现不能在80列宽的终端或打印纸中单行显示了。这会导致代码可读性降低，不知道变量在哪里被使用，很难理解函数调用和返回的流程。

5.5.2 解决方案

要解决这个问题，可以使用一个叫做async的npm模块。async提供一个直观的方式来构造和组织异步调用，并会消除一些（如果不是全部的话）在Node.js异步编程会遭遇到的诡异问题。

串行执行代码

以串行方式执行异步代码的方式有两种：分别是通过waterfall函数和series函数（见图5.1）。

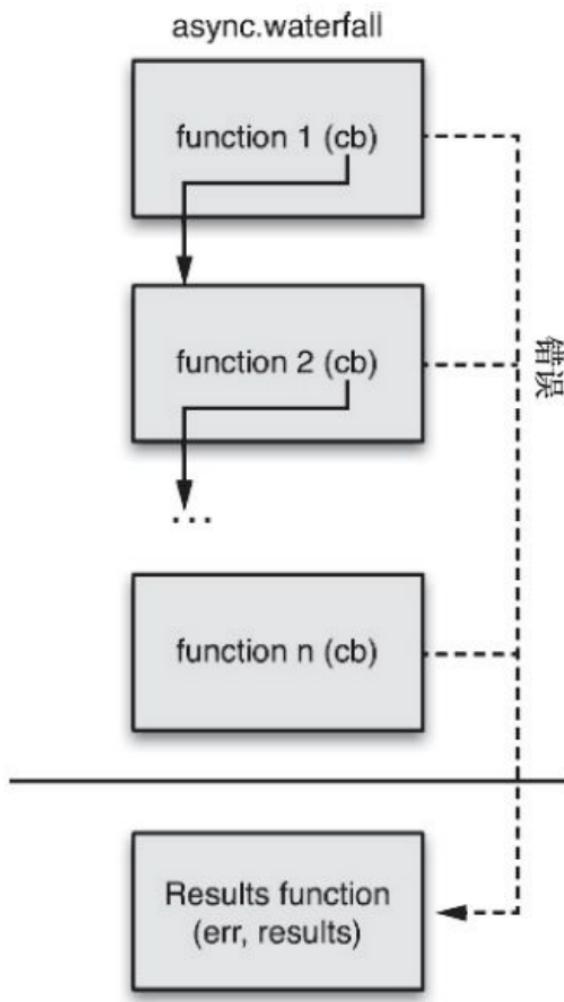


图5.1 使用async.waterfall串行执行

waterfall函数接收一个函数数组作为参数并一次一个地执行它们，然后把每个函数的结果传给下一个函数。结束时，结果函数会接收函数数组中最后一个函数的返回结果作为参数并执行。这种方式下，如果在任何一步出现错误，执行都会停止，结果函数会接收错误信息。

例如，我们可以很容易地使用async.waterfall干净利落地重写之前的代码（存放在Github源代码树中）：

```

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

function load_file_contents(path, callback) {
  async.waterfall([
    function (callback) {
      fs.open(path, 'r', callback);
    },
    // the f (file handle) was passed to the callback at the end of
    // the fs.open function call. async passes all params to us.
    function (f, callback) {
      fs.fstat(f, function (err, stats) {
        if (err)
          // abort and go straight to resulting function
          callback(err);
        else
          // f and stats are passed to next in waterfall
          callback(null, f, stats);
      });
    },
    function (f, stats, callback) {
      if (stats.isFile()) {
        var b = new Buffer(10000);
        fs.read(f, b, 0, 10000, null, function (err, br, buf) {
          if (err)
            callback(err);
          else
            // f and string are passed to next in waterfall
            callback(null, f, b.toString('utf8', 0, br));
        });
      } else {
        callback({ error: "not_file",
                   message: "Can't load directory" });
      }
    },
    function (f, contents, callback) {
      fs.close(f, function (err) {
        if (err)
          callback(err);
        else
          callback(null, contents);
      });
    }
  ],
  // this is called after all have executed in success
  // case, or as soon as there is an error.
  function (err, file_contents) {
    callback(err, file_contents);
  });
}

```

虽然代码看起来有点长，但当我们像数组一样串行组织函数时，代码会看起来非常清晰，阅读起来也很简单。

async.series函数和async.waterfall有两个关键的不同点：

- 来自一个函数的结果不是传到下一个函数，而是收集到一个数组中，这个数组作为“结果”（第二个）参数传给最后的结果函数。依次调用的每一步都会变成结果数组中的一个元素。

- 我们可以传给async.series一个对象，它会枚举每个key并执行每个key对应的函数。在这种方式下，结果不是作为一个数组传入，而是作为拥有相同key的对象被函数调用。

考虑下面的例子：

```
var async = require("async");

async.series({
    numbers: function (callback) {
        setTimeout(function () {
            callback(null, [ 1, 2, 3 ]);
        }, 1500);
    },
    strings: function (callback) {
        setTimeout(function () {
            callback(null, [ "a", "b", "c" ]);
        }, 2000);
    }
},
function (err, results) {
    console.log(results);
});
```

该函数会生成下面的输出结果：

```
{ numbers: [ 1, 2, 3 ], strings: [ 'a', 'b', 'c' ] }
```

并行执行

在前面的async.series例子中，函数没有理由使用串行执行顺序：第二个函数并不依赖第一个函数的结果，所以这些函数可以以并行的方式执行（见图5.2）。对于这种情况，async提供了async.parallel函数，如下所示：

```

var async = require("async");

async.parallel({
    numbers: function (callback) {
        setTimeout(function () {
            callback(null, [ 1, 2, 3 ]);
        }, 1500);
    },
    strings: function (callback) {
        setTimeout(function () {
            callback(null, [ "a", "b", "c" ]);
        }, 2000);
    }
},
function (err, results) {
    console.log(results);
});

```

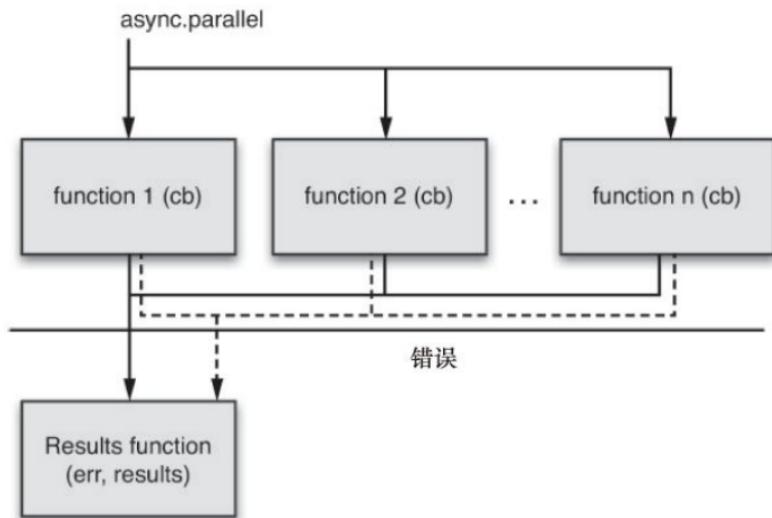


图5.2 使用`async.parallel`并行执行

该函数生成与之前一样的输出。

串行和并行组合起来使用

它们当中功能最强大的函数是`async.auto`，能够让我们将顺序执行和非顺序执行的函数混合起来成为一个功能强大的函数序列。在这里，我们传入一个对象，它的key包含了：

- 将要执行的函数，或者
- 一个依赖数组和一个将要执行的函数。这些依赖都是些字符

串，是提供给async.auto的对象的属性名。auto函数会等待这些依赖都执行完毕才会调用我们提供的函数。

async.auto函数会弄清楚所有要执行的函数的顺序，包括哪些要以并行的方式执行，而哪些需要等待其他函数先执行完（见图5.3）。就像使用async.waterfall那样，我们可以将一个函数的结果通过callback参数传给下一个函数：

```
var async = require("async");

async.auto{
    numbers: function (callback) {
        setTimeout(function () {
            callback(null, [ 1, 2, 3 ]);
        }, 1500);
    },
    strings: function (callback) {
        setTimeout(function () {
            callback(null, [ "a", "b", "c" ]);
        }, 2000);
    },
    // do not execute this function until numbers and strings are done
    // thus_far is an object with numbers and strings as arrays.
    assemble: [ 'numbers', 'strings', function (callback, thus_far) {
        callback(null, {
            numbers: thus_far.numbers.join(", "),
            strings: "" + thus_far.strings.join(", ") + ""
        });
    }]
},
// this is called at the end when all other functions have executed. Optional
function (err, results) {
    if (err)
        console.log(err);
    else
        console.log(results);
});
```

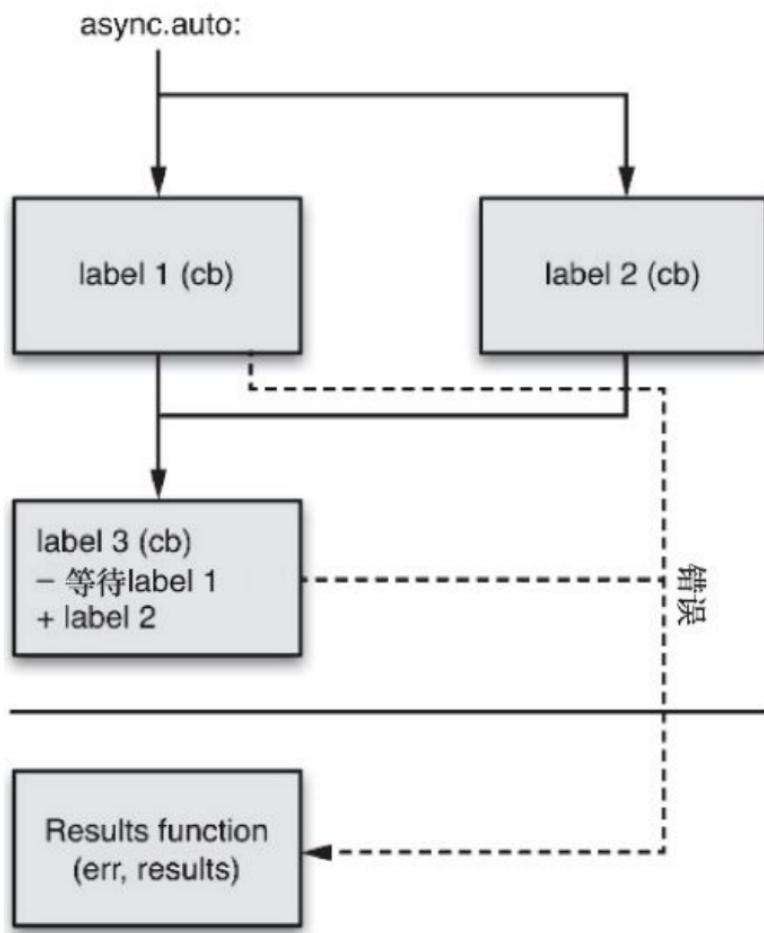


图5.3 通过async.auto实现混合执行模式

传给最后的结果函数的results参数是一个对象，这个对象的属性包含了对象中每个被执行函数的结果：

```
{ numbers: [ 1, 2, 3 ],
  strings: [ 'a', 'b', 'c' ],
  assemble: { numbers: '1, 2, 3', strings: "'\\\'a\\\'', '\\\'b\\\'', '\\\'c\\\'' } }
```

异步循环

在第3章中，我为大家展示了如何使用下面的模式通过异步函数调用的方式来遍历数组中的项：

```
(function iterator(i) {
    if( i < array.length ) {
        async_work( function(){
            iterator( i + 1 )
        })
    } else {
        callback(results);
    }
})(0);
```

虽然这种处理方式能够很好地工作，但实际上有些怪异，它看起来比我想象中的复杂很多。async的async.forEachSeries再次帮到我们，它会遍历我们提供的数组中的每一个元素，为每一个元素调用我们给定的函数。不仅如此，它会在调用序列中的下一个元素之前等待前一个执行完毕：

```
async.forEachSeries(
    arr,
    // called for each element in arr
    function (element, callback) {
        // use element
        callback(null); // YOU MUST CALL ME FOR EACH ELEMENT!
    },
    // called at the end
    function (err) {
        // was there an error? err will be non-null then
    }
);
```

要简单地遍历循环中的每一个元素，然后让async等待所有的元素执行完毕，可以使用async.forEach，它会以相同的方式被调用，不同之处在于它不会串行地执行函数。

async还包含了很多功能，它是当今Node.js编程中真正不可或缺的一个模块。强烈建议大家都浏览下<https://github.com/caolan/async>上的文档并实际使用下。它带来了真正愉悦的Node.js编程环境，并让其变得更美好。

5.6 小结

本章我们在Node.js中正式引入了模块。虽然我们之前见过它，但现在终于知道模块是如何编写的，Node是如何发现需要引入的模块以及如何使用npm来查找并安装模块。现在可以通过 package.json文件自己编写复杂的模块并把它提供给我们的所有项目，甚至可以通过npm发布模块供其他人使用。

最后，我们重点介绍了async的知识，async是最强大、最酷的模块之一，从现在开始我们编写的每一个单独的Node项目都会用到它。

接下来，我们将重新回归到Web服务器的"Web"上。我们会看一些如何在Web应用中使用JSON和Node的方法，并了解如何处理Node的其他核心技术，例如事件和数据流等。

第6章 扩展Web服务器

第二部分的最后这一章将会扩展Web服务器的一些新的关键功能，我们将学习如何处理静态内容，例如HTML页面、JavaScript文件、CSS文件甚至是图片文件。掌握了这些知识之后，就可以把焦点从服务器转移到客户端编程上。

使用Node.js编写网站，将改变传统服务器的工作模式——在发送到客户端前，将HTML生成出来。现在服务器只需处理静态文件或者JSON数据。当用户在站点上浏览时，Web浏览器能够使用AJAX调用，根据模板库生成对应的HTML页面。最后，我们会学习如何上传文件到服务器，并看一些能让这项工作变得简单的工具。

首先从学习Node.js的数据流开始。

6.1 使用Stream处理静态内容

在Node.js异步、非阻塞IO的世界里，我们已经看过在循环中使用fs.open以及fs.read来读取文件的内容。不过，Node.js提供了另一种更为优雅、用来读取（甚至写入）文件的机制，这就是Stream。它和UNIX pipe所扮演的角色很像——我们已经在4.6.1节加载用户数据的场景中，看到过它的简单用法。

Stream最基本的用法是使用on方法，将监听函数（listener）添加到事件（event）上。当事件触发时就会调用所提供的函数。readable事件会在输入流（read stream）读取了进程里的一些内容之后触发。end事件在Stream不再进行内容读取时触发，而error事件会在错误发生时触发。

6.1.1 读取文件

一个简单的例子，编写下面的代码并保存为simple_stream.js：

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

// INCEPTION BWAAAAAAA!!!!
var rs = fs.createReadStream("simple_stream.js");

rs.on('readable', function () {
  var str;
  var d = rs.read();
  if (d) {
    if (typeof d == 'string') {
      str = d;
    } else if (typeof d == 'object' && d instanceof Buffer) {
      str = d.toString('utf8');
    }
    if (str) {
      if (!contents)
        contents = d;
      else
        contents += str;
    }
  }
});
rs.on('end', function () {
  console.log("read in the file contents: ");
  console.log(contents.toString('utf8'));
});
```

如果看了上面的代码（创建一个对象，添加两个监听函数，之后似乎就没再做什么），但不知道为什么代码没有在加载结束时退出，那么请回顾第3章，我曾经提到过Node运行在一个事件循环中并等待某些事情发生，而当事情最终发生时，就去执行相应的代码。

持续变化的Stream

当Node.js发布0.10版本时，Stream经历了重大变化。在这个版本之前，Stream曾有一些很好的功能，但因为计时（timing）和暂停（pausing）等关键问题，这些功能都被废弃了。经过几年的推迟，Node的核心团队最终解决和修复了这些问题，并规避了相关应用的风险。

在这之前，如果想要使用输入流，需要添加data事件的监听函数，让Stream读取的数据作为参数传递给回调函数。而现在，只需监听Readable事件并在事件通知数据已经可读时，调用这个数据流的read方法即可。

如果看到任何使用旧的data事件的代码，不需要为此担心，因为新模型兼容99%的老代码，我只是想让你知道下面会用到这两种模式。

现在就很清楚了，当事件挂起或者即将发生时，例如有个输入流处于打开状态并且正在调用文件系统等待内容读取完成时，就会发生这样的情况。只要预期会有事情发生，Node就会在所有的事件都已完成且用户的代码都已执行完后才退出。

之前的例子通过fs.createReadStream函数和提供的路径创建一个新输入流，然后简单地读取并将内容打印到输出流。当监听到Readable事件时，调用Stream的read方法，取回任何当前可见的数据。如果没有返回数据，它会等待，直至监听到另一个Readable事件或者接收到一个end事件。

使用Buffer操作二进制数据

到目前为止，我们已经使用过Node.js字符串（通常是UTF-8字

字符串)来工作。但在使用数据流和文件时，实际上主要是在与Buffer类打交道。

Buffer和我们期望的多少有些类似：Buffer暂存二进制数据，将数据转换成其他格式，或将数据写入文件，或将数据打散并重新组合。

关于缓冲需要提醒的重要一点是，缓冲对象的length属性并不会返回内容的实际大小，而是返回缓冲本身的大小！例如：

```
var b = new Buffer(10000);
var str = "我叫王马克";
b.write(str); // default is utf8, which is what we want
console.log( b.length ); // will print 10000 still!
```

Node.js不会在缓冲中跟踪把什么数据写入到什么地方，因此，必须自己跟踪写入的数据。

有时字符串的字节长度和字符数不一定相同——例如上面的字符串“我叫王马克”。Buffer.byteLength返回结果如下：

```
console.log( str.length );           // prints 5
console.log( Buffer.byteLength(str) ); // prints 15
```

要将缓冲转换成字符串，需使用toString方法。一般都是将缓冲转换成UTF-8字符串：

```
console.log(buf.toString('utf8'));
```

为了将一个缓冲插入到另外一个的末尾，要使用concat方法，如下所示：

```
var b1 = new Buffer("My name is ");
var b2 = new Buffer("Marc");
var b3 = Buffer.concat([ b1, b2 ]);
console.log(b3.toString('utf8'));
```

最后，可以“清零”或者使用fill方法填充缓冲中所有的值，例如buf.fill("\0")。

6.1.2 在Web服务器中使用Buffer处理静态文件

接下来，我们练习编写一个小小的Web服务器，这个服务器会使用Node的Buffer来处理静态内容（一个HTML文件）。可以从handle_incoming_request函数开始：

```
function handle_incoming_request(req, res) {
    if (req.method.toLowerCase() == 'get'
        && req.url.substring(0, 9) == '/content/') {
        serve_static_file(req.url.substring(9), res);
    } else {

        res.writeHead(404, { "Content-Type" : "application/json" });

        var out = { error: "not_found",
                   message: "'" + req.url + "' not found" };
        res.end(JSON.stringify(out) + "\n");
    }
}
```

如果传入的请求访问的是/content/something.html，则会尝试调用serve_static_file函数来处理。Node.js的http模块设计得足够聪明，服务器上每次收到的请求对应的ServerResponse对象本身实际上就是一个数据流，我们可以在其上编写对应的输出。可以通过调用Stream类的write方法来完成：

```
function serve_static_file(file, res) {
    var rs = fs.createReadStream(file);
    var ct = content_type_for_path(file);
    res.writeHead(200, { "Content-Type" : ct });

    rs.on(
        'readable',
        function () {
            var d = rs.read();
            if (d) {
                if (typeof d == 'string')
                    res.write(d);
                else if (typeof d == 'object' && d instanceof Buffer)
                    res.write(d.toString('utf8'));
            }
        }
    );
}

rs.on(
    'end',
    function () {
        res.end(); // we're done!!!
    }
);
}

function content_type_for_path(file) {
    return "text/html";
}
```

剩余的服务器代码就与之前见到的一样了：

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

var s = http.createServer(handle_incoming_request);
s.listen(8080);
```

新建一个叫做test.html的文件，它含有一些简单的HTML内容。

随后在服务器上运行

```
node server.js
```

接着使用curl请求test.html：

```
curl -i -X GET http://localhost:8080/content/test.html
```

应该看到和下面类似的输出（这取决于真正写到test.html里的内容）：

```
HTTP/1.1 200 OK
Date: Mon, 26 Nov 2012 03:13:50 GMT
Connection: keep-alive
Transfer-Encoding: chunked
```

```
<html>
<head>
    <title> WooO! </title>
</head>
<body>
    <h1> Hello World! </h1>
</body>
</html>
```

现在我们能够处理静态内容了！但还有两个问题。首先，如果请求/content/blargle.html，那会发什么？当前的脚本会抛出一个错误并终止，这并不是我们想要的结果。这种情况下，我们期望返回404的HTTP状态码以及可能的错误信息。

为此，可以监听输入流的error事件。添加以下几行代码到serve_static_file函数中：

```
rs.on(
  'error',
  function (e) {
    res.writeHead(404, { "Content-Type" : "application/json" });
    var out = { error: "not_found",
               message: "'" + file + "' not found" };
    res.end(JSON.stringify(out) + "\n");
    return;
  }
);
```

现在，捕捉到错误后（之后没有data或者end事件被调用），就更新响应头的代码为404，设置新Content-Type，并返回包含错误信息的JSON，客户端可以使用这些JSON信息告诉用户发生了什么错误。

6.1.3 不仅仅支持HTML

第二个问题是现在只能处理HTML格式的静态内容。`content_type_for_path`函数只能返回"text/html"。如果能够更灵活

则会更好，可以通过以下的代码完成这一需求：

```
function content_type_for_file (file) {
    var ext = path.extname(file);

    switch (ext.toLowerCase()) {
        case '.html': return "text/html";
        case ".js": return "text/javascript";
        case ".css": return 'text/css';
        case '.jpg': case '.jpeg': return 'image/jpeg';
        default: return 'text/plain';
    }
}
```

现在可以对不同文件类型调用curl命令并得到预期的结果。对于类似JPEG图片的二进制文件，可以对curl使用-o参数来告诉curl将输出写入到指定的文件中。首先，复制一个JPEG文件到当前文件夹中，通过node server.js运行服务器，随后输入下面的代码：

```
curl -o test.jpg http://localhost:8080/content/family.jpg
```

机会来了，现在有一个叫做test.jpg的文件，它正是我们想要的。

从数据流（上面例子中的rs）到数据流（res）的数据传递是很常见的场景，Node.js的Stream类有一个很便捷的方法来为你做这件事：pipe。这使得serve_static_file函数变得更简单：

```
function serve_static_file(file, res) {
    var rs = fs.createReadStream(file);
    rs.on(
        'error',
        function (e) {
            console.log("oh no! Error!! " + JSON.stringify(e));
            res.end("");
        }
    );
    var ct = content_type_for_path(file);
    res.writeHead(200, { "Content-Type" : ct });
    rs.pipe(res);
}
```

遗憾的是，pipe有一个小问题：一旦调用pipe，它会立即发送HTTP响应头，发送之后，后面就无法再调用ServerResponse.writeHead了。如果想要通过

fs.createReadStream读取的文件不存在，就没有办法通知到客户端。要解决这个问题，需要首先确认文件是否存在，然后再执行pipe操作：

```
function serve_static_file(file, res) {
  fs.exists(file, function (exists) {
    if (!exists) {
      res.writeHead(404, { "Content-Type" : "application/json" });
      var out = { error: "not_found",
                 message: file + ' not found' };
      res.end(JSON.stringify(out) + "\n");
      return;
    }

    var rs = fs.createReadStream(file);
    rs.on('error',
          function (e) {
            res.end();
          }
    );

    var ct = content_type_for_file(file);
    res.writeHead(200, { "Content-Type" : ct });
    rs.pipe(res);
  });
}
```

事件

Stream实际上是Node.js的Event类的子类，JavaScript文件中Event类提供连接和触发事件的所有功能。我们可以继承这个类来创建自己的事件触发类。例如，创建一个dummy下载器类，通过延迟2秒调用，伪装成一个远程下载：

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

function Downloader () {
}
Downloader.prototype = new events.EventEmitter();
Downloader.prototype.__proto__ = events.EventEmitter.prototype;
Downloader.prototype.url = null;
Downloader.prototype.download_url = function (path) {
    var self = this;
    self.url = path;
    self.emit('start', path);
    setTimeout(function () {
        self.emit('end', path);
    }, 2000);
}

var d = new Downloader();
d.on("start", function (path) {
    console.log("started downloading: " + path);
});
d.on("end", function (path) {
    console.log("finished downloading: " + path);
});
d.download_url("http://marcwan.com");
```

可以通过调用带有事件名称的emit方法来触发事件，并且向监听函数传递任何参数。我们需要确保所有可能的代码块（包括错误）都会触发某个事件；否则，Node程序可能会挂起。

6.2 在客户端组装内容：模板

在传统的Web应用模型中，客户端发送一个HTTP请求到服务器，服务器收集所有的数据并生成对应的HTTP响应，最后以文本形式发送出去。这种处理方式虽然非常可靠，但还是有一些明显的缺点：

- 没有充分利用当今普通客户端电脑的计算能力。现在即使普通的移动电话或者平板电脑都比10年前的PC强大许多倍。
- 当有多种类型的客户端时，会非常难处理。许多人可能通过Web浏览器访问，有些则通过移动应用访问，甚至有人会通过桌面应用或者第三方应用访问。
- 在服务器上运行这么多种类型的东西会非常糟糕。如果只让服务器专注于处理、存储以及生成数据，而让客户端决定如何展现数据会更合理一些。

这种方式日益常见，我还发现Node有一点特别引人注目和好玩，就是服务器只提供JSON格式数据，或者尽可能小的数据。客户端可以选择如何展现返回的数据。脚本、样式表甚至大部分HTML都可以托管在文件服务器或内容分发网络（CDN）中。

对于Web浏览器端的应用，可以使用客户端模板（见图6.1）。这种方式需要：

1. 客户端下载HTML页面的骨架，包含了JavaScript文件、CSS文件以及一个来自Node服务器的空body元素的指针。
2. 引入的JavaScript文件中有个加载器（bootstrapper），它会处理包括收集所有数据和组装页面等所有工作。需要给HTML模板的名字以及一个供调用的服务器JSON API。引导程序下载模板，然后使用模板引擎将返回的JSON数据应用到模板文件中。接下来我们会使用一个叫"Mustache"的模板引擎（参见补充内容“模板引擎”）。

3. 返回的HTML代码被插入页面的body元素中，服务器所需要做的仅仅是处理一点点JSON数据。

这里有个前提，就是现在已经把相册应用开发成一个JSON服务器。尽管还会为它添加处理静态文件的能力，但这主要是为了方便和学习的目的；在生产环境中，应该将这些文件移到CDN中，并适当地更新指向它们的URL。

模板引擎

目前有大量的客户端（甚至是服务器端）模板引擎和模板库。本书中最常用的是jQuery JavaScript库，另外还有一些，例如JSRender或者jQote的模板引擎，这两个引擎也都非常不错。同样还有很多其他不需要适配本书选用的JavaScript类库的模板引擎。

对于本书中的需求，这些模板都大致相同。它们的特性集都基本相同，具有类似的处理方式以及基本一样的性能。当要选用模板时，建议选择正在持续开发和（或）维护的、有合理语法的模板。

本书和我最近的一些项目都使用了叫做Mustache的模板引擎。正如之前所说，这个决定有些随意，但mustache.js有以下优点：

- 它是一个相当小巧且快速的JavaScript库。
- 它是一个功能完全的模板解决方案。
- 它有一个相当酷的名字。Mustache的标签用{}和{}分割，它们看起来确实很像胡须。

这个解决方案并不完美，但我们会发现Mustache在特性和速度之间不断权衡。我们鼓励大家多去寻找其他解决方案，并阅读相关的博客和教程。

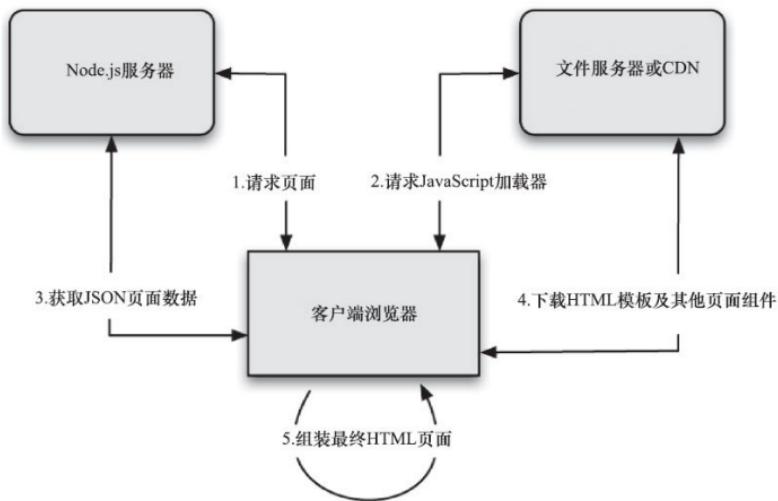


图6.1 使用模板生成客户端页面

为了将相册转换成模板形式，需要完成如下工作：

- 1) 为页面生成HTML骨架。
- 2) 为应用程序添加处理静态文件内容功能。
- 3) 修改支持的URL列表，分别为HTML页面和模板添加对应的/pages/和/templates/。
- 4) 编写模板和加载这些模板的JavaScript文件。

一切就绪，那我们出发吧！

6.2.1 HTML骨架页面

尽管可以尝试在客户端上生成所有的HTML页面，但让服务器完全不做任何事情却是不可能的。我们还是需要服务器输出初始页面的骨架，而客户端可以用它来生成剩余的部分。

在下面的例子中，使用一个合理而简单的HTML页面，保存为 basic.html，如代码清单6.1所示。

代码清单6.1 简单的应用页面加载器（basic.html）

```
<!DOCTYPE html>
<html>
<head>
    <title>Photo Album</title>

    <!-- Meta -->
    <meta http-equiv="content-type" content="text/html; charset=utf-8" />

    <!-- Stylesheets -->
    <link rel="stylesheet" href="http://localhost:8080/content/style.css"
        type="text/css" />

    <!-- javascripts -->
    <script src="http://localhost:8080/content/jquery-1.8.3.min.js"
        type="text/javascript"></script>
    <script src="http://localhost:8080/content/mustache.js"
        type="text/javascript"></script>
    <script src="http://localhost:8080/content/{{PAGE_NAME}}.js"
        type="text/javascript"></script>
</head>
<body></body>

</html>
```

这个文件相当直截了当，它有一个样式表，然后引入jQuery和Mustache。最后一个script标签是最有意思的：这个文件是页面的加载器，其中的代码会下载模板，从服务器中获取这个页面对应的JSON数据，并为它生成HTML。可以看到在任何时候，加载器通过参数{{PAGE_NAME}}替换成实际将要加载的页面。

6.2.2 处理静态内容

现在我们需要对应用服务器文件夹的布局稍作修改，如下所示：

```
+ project_root/
    + contents/      # JS, CSS, and HTML files
    + templates/     # client-side HTML templates
    + albums/        # the albums we've seen already
```

之前看到的是albums文件夹，而现在需要添加内容和模板文件夹，用于处理即将添加的额外内容。

更新后的handle_incoming_request函数如下所示。
serve_static_file函数则和前面章节中所写的一样：

```
function handle_incoming_request(req, res) {
    // parse the query params into an object and get the path
    // without them. (true for 2nd param means parse the params).
    req.parsed_url = url.parse(req.url, true);
    var core_url = req.parsed_url.pathname;
```

```
// test this updated url to see what they're asking for
if (core_url.substring(0, 9) == '/content/') {
    serve_static_file("content/" + core_url.substring(9), res);
} else if (core_url == '/albums.json') {
    handle_list_albums(req, res);
} else if (core_url.substr(0, 7) == '/albums'
           && core_url.substr(core_url.length - 5) == '.json') {
    handle_get_album(req, res);
} else {
    send_failure(res, 404, invalid_resource());
}
}
```

例如，在content/文件夹中，首先会有三个文件：

- jquery-1.8.3.min.js——可以直接从jquery.com上下载。过去几年的任何jQuery版本应该都能正常工作。
- mustache.js——可以从很多途径下载这个文件。在这里，简单地使用从github.com/janl/mustache.js下载的通用版本。还有许多服务器端的mustache版本，但这不是这一章我们需要的。
- style.css——可以创建一个空文件或写入一些CSS来修改即将为相册应用生成的页面。

6.2.3 修改URL解析机制

目前应用服务器只支持/albums.json和/albums/album_name.json。现在可以添加对静态内容、页面以及模板文件的支持，如下所示：

```
/content/some_file.ext
/pages/some_page_name[/optional/junk]
/templates/some_file.html
```

对于内容和模板文件，可以直接从硬盘下载真实的文件。而对于页面请求，则总是返回之前看到的basic.html的定制版本。它有个{{PAGE_NAME}}宏，用于替换通过URL传入的页面名字，即/page/home。

更新后的handle_incoming_request函数如下所示：

```

function handle_incoming_request(req, res) {
    // parse the query params into an object and get the path
    // without them. (true for 2nd param means parse the params).
    req.parsed_url = url.parse(req.url, true);
    var core_url = req.parsed_url.pathname;

    // test this fixed url to see what they're asking for
    if (core_url.substring(0, 7) == '/pages/') {
        serve_page(req, res);
    } else if (core_url.substring(0, 11) == '/templates/') {
        serve_static_file("templates/" + core_url.substring(11), res);
    } else if (core_url.substring(0, 9) == '/content/') {
        serve_static_file("content/" + core_url.substring(9), res);
    } else if (core_url == '/albums.json') {
        handle_list_albums(req, res);
    } else if (core_url.substr(0, 7) == '/albums'
               && core_url.substr(core_url.length - 5) == '.json') {
        handle_get_album(req, res);
    } else {
        send_failure(res, 404, invalid_resource());
    }
}

```

实际上，serve_page函数决定了客户端请求的是什么页面，它会修改basic.html模板文件，并发送最终的HTML骨架到浏览器：

```

/**
 * All pages come from the same one skeleton HTML file that
 * just changes the name of the JavaScript loader that needs to be
 * downloaded.
 */
function serve_page(req, res) {

    var core_url = req.parsed_url.pathname;
    var page = core_url.substring(7);           // remove /pages/
    // currently only support home!
    if (page != 'home') {
        send_failure(res, 404, invalid_resource());
        return;
    }

    fs.readFile(
        'basic.html',
        function (err, contents) {
            if (err) {
                send_failure(res, 500, err);
                return;
            }

            contents = contents.toString('utf8');

            // replace page name, and then dump to output.
            contents = contents.replace('{{PAGE_NAME}}', page);
            res.writeHead(200, { "Content-Type": "text/html" });
            res.end(contents);
        }
    );
}

```

我们使用fs.readFile替代fs.createReadStream读取整个文件的

内容到Buffer。必须将该Buffer转换成字符串，然后修改其内容以指定正确的JavaScript加载器。不推荐对大文件使用这种方法，因为这会浪费大量内存，但对于与basic.html一样短的文件，这会相当方便。

6.2.4 JavaScript加载器

HTML页面非常简单，因此我们可以从一个很小的JavaScript加载器开始。当本书后续章节添加更多功能后，它们会变得略微复杂，代码清单6.2展示了一个现在可用的JavaScript加载器，保存为home.js。

代码清单6.2 JavaScript页面加载器 (home.js)

```
$(function(){
    var tmpl,      // Main template HTML
        tdata = {}; // JSON data object that feeds the template

    // Initialize page
    var initPage = function() {
        // Load the HTML template
        $.get("/templates/home.html", function(d){           // 1
            tmpl = d;
        });

        // Retrieve the server data and then initialize the page
        $.getJSON("/albums.json", function (d) {             // 2
            $.extend(tdata, d.data);
        });

        // When AJAX calls are complete parse the template
        // replacing mustache tags with vars
        $(document).ajaxStop(function () {
            var renderedPage = Mustache.to_html(tmpl, tdata); // 3
            $("body").html(renderedPage);
        });
    }();
});
```

\$(function(){...语法和\$(document).ready(function()...在jQuery中基本是等价的，只是更加简短。该函数会在所有资源（特别是jQuery和Mustache）都加载完后被调用。它会完成以下三件事情：

- 1) 通过调用URL/tmp/home.html从服务器请求模板文件，即home.html。
- 2) 在应用里请求展示相册的JSON数据，即/albums.json。
- 3) 最后，将两者赋予Mustache，让其完成模板组装。

6.2.5 使用Mustache模板化

相册应用选择的模板引擎Mustache具有易用、快速、小巧的特点。更有趣的是，它只有标签，封装在{{和}}中间，与胡须极为相似。它没有if语句表达式或者循环结构。标签有一系列的规则，根据提供的数据进行工作。

如前面章节所看到的，Mustache的基本用法是：

```
var html_text = Mustache.to_html(template, data);
$("body").html(html_text);
```

要想打印对象的属性，可以使用下面的代码：

Mustache template:

The album "{{name}}" has {{photos.length}} photos.

JSON:

```
{
  "name": "Italy2012",
  "photos" : [ "italy01.jpg", "italy02.jpg" ]
}
```

Output:

The album "Italy2012" has 2 photos.

为集合或者数组中的每一项生成模板，可使用#字符。例如：

Mustache:

```
{#albums}
  * {{name}}
{{/albums}}
```

JSON:

```
{
  "albums" : [ { "name" : "italy2012" },
                { "name" : "australia2010" },
                { "name" : "japan2010" }]
}
```

Output:

```
* italy2012
* australia2010
* japan2010
```

如果没有匹配的结果，则什么都不会输出。通过^字符可以捕获到这类情况：

Mustache:

```
{{#albums}}
  * {{name}}
{{/albums}}
{{^albums}}
  Sorry, there are no albums yet.
{{/albums}}
```

JSON:

```
{ "albums" : [ ] }
```

Output:

```
Sorry, there are no albums yet.
```

如果跟在#字符后的对象不是一个集合或者数组，而是一个对象，那么其中的值会被使用，但不会遍历：

Mustache:

```
{{#album}}
  The album "{{name}}" has {{photos.length}} photos.
{{/album}}
```

JSON:

```
{
  "album": { "name": "Italy2012",
              "photos" : [ "italy01.jpg", "italy02.jpg" ] }
}
```

Output:

```
The album "Italy2012" has 2 photos.
```

默认情况下，所有的HTML字符都会被转义，<和>会分别被转义成<及>，等等。如果不想要Mustache做这样的处理，则使用额外的Mustache三重括号，{{{{和}}}：

Mustache:

```
{{{#users}}
  * {{{name}}} says {{{ saying }}}
    * raw saying: {{ saying }}
{{{/users}}}
```

JSON:

```
{
  "users" : [ { "name" : "Marc", "saying" : "I like <em>cats</em>!" },
               { "name" : "Bob", "saying" : "I <b>hate</b> cats" },
  ]
```

Output:

```
* Marc says I like <em>cats</em>!
  * raw saying: I like &lt;em&gt;cats&lt;/em&gt;!
* Bob says I <b>hate</b> cats
  * raw saying: I &lt;b&gt;hate&lt;/b&gt; cats
```

6.2.6 首页Mustache模板

现在，是时候为首页编写第一个模板了，可以保存为home.html，如代码清单6.3所示。

代码清单6.3 首页模板文件（home.html）

```
<div id="album_list">
  <p> There are {{ albums.length }} albums</p>
  <ul id="albums">
    {{{#albums}}}
      <li class="album">
```

```
<a href='http://localhost:8080/pages/album/{{name}}'>{{name}}</a>
</li>
{{/albums}}
{{^albums}}
<li> Sorry, there are currently no albums </li>
{{/albums}}
</ul>
</div>
```

如果有相册，代码会遍历每一个相册项，提供一个元素，包含了相册的名字和指向相册页面的超链接，可以通过URL/page/album/album_name访问。当没有相册时，则会打印简单的信息。

6.2.7 整合应用

上面几节介绍了许多新知识，现在要使它们运行起来，则需要为Web应用提供如下文件布局：

```
+ project_root/
    server.js                                // GitHub source
    basic.html                                 // Listing 6.1
    + content/
        home.js                                  // Listing 6.2
        album.js                                 // Listing 6.5
        jquery-1.8.3.min.js
        mustache.js
        style.css
    + templates/
        home.html                               // Listing 6.3
        album.html                              // Listing 6.4
    + albums/
        + whatever albums you want
```

album.html和album.js的代码之前并没有见过，附在代码清单6.4和6.5上。

要运行该项目，需要输入如下代码：

```
node server.js
```

然后打开Web浏览器并浏览http://localhost:8080/page/home。如果想在命令行里看看都发生了什么，可以像下面一样用curl试试：

```
curl -i -X GET http://localhost:8080/page/home
```

因为curl不能执行客户端的JavaScript代码，所以模板和JSON数据不能通过Ajax加载，只能看到来自basic.html的基本HTML骨架。

代码清单6.4 另外一个Mustache模板页 (album.html)

```
<div id="album_page">
{{#photos}}
<p> There are {{ photos.length }} photos in this album</p>
<div id="photos">
<div class="photo">
<div class='photo_holder'>
</div>
<div class='photo_desc'><p>{{ desc }}</p></div>
</div>
</div> <!-- #photos -->
<div style="clear: left"></div>
{{/photos}}
{{^photos}}
<p> This album doesn't have any photos in it, sorry.</p>
{{/photos}}
</div> <!-- #album_page -->
```

代码清单6.5 相册页面加载器JavaScript文件 (album.js)

```
$(function(){

    var tmpl, // Main template HTML
        tdata = {} // JSON data object that feeds the template

    // Initialize page
    var initPage = function() {
        // get our album name.
        parts = window.location.href.split("/");
        var album_name = parts[5];

        // Load the HTML template
        $.get("/templates/album.html", function(d){
            tmpl = d;
        });

        // Retrieve the server data and then initialize the page
        $.getJSON("/albums/" + album_name + ".json", function (d) {
            var photo_d = massage_album(d);
            $.extend(tdata, photo_d);
        });

        // When AJAX calls are complete parse the template
        // replacing mustache tags with vars
        $(document).ajaxStop(function () {
            var renderedPage = Mustache.to_html( tmpl, tdata );
            $("body").html( renderedPage );
        })
    }();

    function massage_album(d) {

        if (d.error != null) return d;
        var obj = { photos: [] };

        var af = d.data.album_data;

        for (var i = 0; i < af.photos.length; i++) {
            var url = "/albums/" + af.short_name + "/" + af.photos[i].filename;
            obj.photos.push({ url: url, desc: af.photos[i].filename });
        }
        return obj;
    }
})
```

如果上面所有步骤都正确完成，应该可以看到如图6.2所示的页面。页面不是很美观，但可以通过修改模板文件以及style.css文件来

美化它。

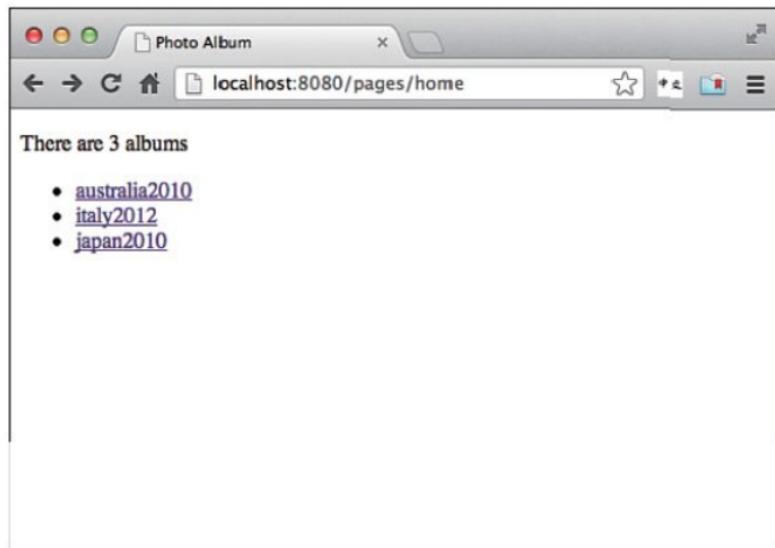


图6.2 运行于浏览器的基于模板的JSON应用程序

6.3 小结

本章介绍了Node.js的两个关键特性：数据流和事件，使用它们可以为Web应用处理静态内容。同样重要的是，本章还介绍了客户端的模板引擎，以及使用Mustache将服务器和相册应用程序的前端部分联系起来。

但大家可能注意到，目前做的许多事情貌似被不必要地复杂化，非常冗长乏味，甚至很可能出现bug（尤其是`handle_incoming_request`函数）。因此需要一些优秀的模块来协助我们完成这些事情。

因此，接下来我不打算进一步开发相册，而是看一些可以显著清理代码的模块，这些模块还能帮助我们以很少的代码来快速添加功能。我们将会使用Node的express应用框架完成这些工作。

第三部分 实战篇

第7章 使用express构建Web应用

第8章 数据库I : NoSQL (MongoDB)

第9章 数据库II : SQL (MySQL)

第7章 使用express构建Web应用

目前为止，我们已经学习了Node.js的基本原理及核心概念。利用这些知识，我们创建了一些简单的应用，但还是需要用冗长的代码实现简单的功能。现在是做一点改变的时候了：利用Node.js的一大优势——npm上拥有不计其数的类库和模块可供使用——编写一些有趣的Web应用。既然我们已经了解了Node及其模块的核心工作方式，那么现在就探索一下让编程变得简单和快速的方式。

在本章中，我们将会开始学习express——一个为Node量身打造的Web应用框架。它能完成许多前面章节中我们学习到的基本操作，甚至提供了许多强大的功能，让我们在应用中使用。这里，我会演示如何使用express构建应用，以及如何使用它更新相册应用。最后，我们会深入应用设计，学习如何设计JSON服务器的API，并探索一些以前从来没有见过的新功能。

7.1 安装express

在本章中，我们会改变原先使用npm安装模块的方式。前文中，如果想要安装express，只需输入：

```
npm install express
```

这没有任何问题！但是一般情况下，这种方式会安装最新版本的express模块；而在实际生产环境中部署应用时，往往不希望安装模块的最新版本。尤其当开发和测试只针对某个特定版本时，最好在生产服务器中使用相同的版本。只有更新到最新版本并经过严格测试，才能在生产环境中使用新版本。

因此，构建Node应用的基本步骤如下所示：

- 1) 创建应用文件夹。
- 2) 编辑package.json文件，该文件用来描述软件包并指定所需的npm模块。
- 3) 运行npm install命令，确保这些模块都正确安装。

创建名为trivial_express/的新文件夹用来存放express测试应用，然后把下面的代码编辑到该文件夹下的package.json文件中：

```
{
  "name": "Express-Demo",
  "description": "Demonstrates Using Express and Connect",
  "version": "0.0.1",
  "private": true,
  "dependencies": {
    "express": "3.x"
  }
}
```

name和description字段顾名思义，极易理解。现在这个应用还很简单，拥有一个对应的版本号0.0.1。正如第5章里所讲的，我们还可以将private设置为true，表明npm不会将其发布到npm模块注册中心，因为这是一个仅供自己使用的测试项目。

完成文件创建以后，可以运行以下命令，它会读取配置文件中的依赖部分，并安装所有合适的模块：

```
npm install
```

由于express本身就依赖很多不同的模块，因此，在安装过程中会看到大量的打印信息。安装完成之后，可以在node_modules/文件夹中找到安装的模块。如果修改了package.json文件，可以使用npm update命令来确保所有模块都正确安装。

使用express创建"Hello World"

回顾下第1章，第一个Node.js的Web服务器非常粗糙，如下所示：

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

function process_request(req, res) {
    res.end("Hello World");
}

var s = http.createServer(process_request);
s.listen(8080);
```

现在使用express实现二者，看起来非常相似：

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

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

app.listen(8080);
```

保存并运行上述代码，使用curl或者在Web浏览器中输入localhost:8080，可以看到预期的返回结果hello world。最棒的事情是，express是构建在目前我们所学到的知识之上的，因此HTTP请求和响应对象会拥有之前的所有方法和属性。同时，它们还拥有很多

其他有趣和强大的东西，但我们无须操作完全不熟悉的东西，因为这些都是我们之前熟悉的对象的扩展版本。

7.2 express中的路由和分层

express完全构建在一个名叫connect的Node模块之上。

connect作为中间件（middleware）类库而为大家所熟知，它提供了一系列网络应用所需的常用功能。该模型的核心是函数流，也就是大家常说的函数分层机制。

中间件：名称的奥秘

当第一次看到中间件这个术语时，也许你会担心错过了什么关键的新概念或技术变革。但幸运的是，它并不是那么复杂。

实际上，中间件一开始是被用来描述软件组件的（一般是服务器端的东西），这种组件会将两个东西连接起来，如事务逻辑层和数据库服务器的连接或者存储服务和应用模块的连接等。

随着时间的迁移，这个术语的定义变得更加通用和宽泛，现在可以用来描述任何连接两个事物的软件片段。

从本书的角度来看，express中使用的connect提供的中间件只是一组单纯的组件，可以让Web应用无缝集成服务器端或浏览器端提供的功能。

使用express创建一系列的函数（层）来构建应用；一旦其中一个函数接管了请求，它就会处理并停止后面的函数序列。express会直接跳到执行函数序列的最后（一些层用来“进栈”，一些用来“出栈”）。

因此，在前面的express应用中，执行序列中仅有一层：

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

该函数是express提供的帮助函数，只要满足下述两个条件，就可以用来处理传入的请求（通过调用提供的函数）：

- HTTP请求的方法为GET。

- 请求的URL路径为/。

如果传入的请求不满足以上条件，该函数就不会被调用。如果没有函数与请求的URL匹配——例如，使用curl或浏览器请求/some/other/url——express就会返回一个默认的响应：

```
Cannot GET /some/other/url
```

7.2.1 路由基础

URL路由函数的一般格式如下：

```
app.method(url_pattern, optional_functions, request_handler_function);
```

前面的路由处理程序用来处理URL/传递过来的GET请求。如果想要支持处理某个特定URL下的POST请求，可以使用express应用对象上的post方法：

```
app.post("/forms/update_user_info.json", function (req, res) { ... });
```

同样，express也支持DELETE和PUT方法，通过delete和put方法，我们会在后面的相册应用中使用到它们。还可以通过all方法让路由函数接受所有指定URL的HTTP方法：

```
app.all("/users/marcwan.json", function (req, res) { ... });
```

路由函数的第一个参数使用正则表达式匹配传入的URL。所以，它支持简单的URL：

```
/path/to/resources
```

也支持正则表达式，如：

```
/user(s)?/list
```

上述表达式会匹配/users/list和/user/list，而

```
/users/*
```

会匹配所有以/users/开头的URL。

express路由最强大的特性之一就是支持占位符从请求路由中提取指定值，使用冒号（:）来标记。当解析路由的时候，express会将匹配到的占位符值保存到req.params对象中，例如：

```
app.get("/albums/:album_name.json", function (req, res) {  
    res.end("Requested " + req.params.album_name);  
});
```

如果调用上面的app请求/albums/italy2012.json，可以得到以下输出结果：

```
Requested italy2012.
```

同样，我们也可以在同一个URL上添加多个参数：

```
app.get("/albums/:album_name/photos/:photo_id.json", function (req, res) { ... });
```

一旦该函数被调用，req.params中的album_name和photo_id就会被赋予传递进来的值。除了斜杠（/），占位符可以匹配任意字符串。

正如我们看到的一样，上述路由方法中提供的回调函数已经被赋予请求和响应对象。而实际上它还被赋予了第三个参数，你可以选择使用或者忽略它。该参数一般使用next这个变量名（当函数调用next以后，分层函数流就会传递给下一个分层函数）。我们还可以对传入的URL请求做一些额外的检测，或者选择忽略它，如下所示：

```
app.get("/users/:userid.json", function (req, res, next) {  
    var uid = parseInt(req.params.userid);  
    if (uid < 2000000000) {  
        next(); // don't want it. Let somebody else process it.  
    } else {  
        res.end(get_user_info(uid));  
    }  
});
```

7.2.2 更新相册应用路由

该相册应用很容易使用express重构。我们只需要做一点工作就能让它正常运行：

- 创建package.json文件，并安装express。
- 使用express替换http模块的服务器。
- 使用路由处理程序替换handle_incoming_request函数。
- 更新在express中获取查询参数的方式（为了方便使用，这些参数会存放到req.query对象中）。

现在，将第6章最后创建的相册应用复制到一个新文件夹中（将其命名为basic_routing/），并将下面的package.json文件放到该文件夹下：

```
{  
  "name": "Photo-Sharing",  
  "description": "Our Photo Sharing Application",  
  "version": "0.0.2",  
  "private": true,  
  "dependencies": {  
    "async": "0.1.x",  
    "express": "3.x"  
  }  
}
```

运行npm install命令，确保express和async安装到node_modules/文件夹下。可以将server.js文件顶部的http模块替换成express：

```
var express = require('express');  
var app = express();  
  
var path = require("path"),  
    async = require('async');  
    fs = require('fs');
```

并将

```
http.listen(8080);
```

修改成

```
app.listen(8080);
```

接下来需要替换handle_incoming_request函数，代码如下所示：

```
function handle_incoming_request(req, res) {
    req.parsed_url = url.parse(req.url, true);
    var core_url = req.parsed_url.pathname;

    // test this fixed url to see what they're asking for
    if (core_url.substring(0, 7) == '/pages/') {
        serve_page(req, res);
    } else if (core_url.substring(0, 11) == '/templates/') {
        serve_static_file("templates/" + core_url.substring(11), res);
    } else if (core_url.substring(0, 9) == '/content/') {
        serve_static_file("content/" + core_url.substring(9), res);
    } else if (core_url == '/albums.json') {
        handle_list_albums(req, res);
    } else if (core_url.substr(0, 7) == '/albums' &&
               && core_url.substr(core_url.length - 5) == '.json') {
        handle_get_album(req, res);
    } else {
        send_failure(res, 404, invalid_resource());
    }
}
```

使用下面的路由函数替换上述代码：

```
app.get('/albums.json', handle_list_albums);
app.get('/albums/:album_name.json', handle_get_album);
app.get('/content/:filename', function (req, res) {
    serve_static_file('content/' + req.params.filename, res);
});
app.get('/templates/:template_name', function (req, res) {
    serve_static_file("templates/" + req.params.template_name, res);
});
app.get('/pages/:page_name', serve_page);
app.get('*', four_oh_four);

function four_oh_four(req, res) {
    send_failure(res, 404, invalid_resource());
}
```

现在的路由函数不仅比原先的函数更加干净简洁，而且能直观地看出URL是如何匹配请求及整个应用是如何工作的。我们还添加了一个新的路由匹配“*”，这样其他所有的请求将会返回404响应。

但是，现在路由函数面临一个棘手的问题。如果用户请求/pages/album/italy2012会怎样？就目前而言，这个应用会失败：因为正则表达式不能匹配带有斜杠（/）字符的参数，所以路由/pages/:page_name匹配不到该请求路径。为了解决这个问题，必须添加一个新的路由函数：

```
app.get('/pages/:page_name/:sub_page', serve_page);
```

现在，所有的请求页面路径会解析到正确的路由函数上，同时我们也能很清晰地识别请求的子页面。

现在这个应用的一个优点就是无须手动解析传入的请求或参数，express已经为我们做好这一切了。因此，之前需要引入url模块解析传入的URL，而现在，只要使用路由函数即可。同样，也不需要解析GET查询参数，因为它们已经被解析完成。因此，可以更新一些帮助函数，如下所示：

```
function get_album_name(req) {
    return req.params.album_name;
}

function get_template_name(req) {
    return req.params.template_name;
}

function get_query_params(req) {
    return req.query;
}

function get_page_name(req) {
    return req.params.page_name;
}
```

通过上面的改造，相册应用应该被更新并开始使用express。而功能上，也会和原版本保持一致。

7.3 REST API设计和模块

如果你正打算设计一个JSON服务器，如本书应用的JSON服务器，则需要花一些时间仔细斟酌API以及客户端会如何使用它。在设计API之前多花些时间思考，有助于理解用户如何使用应用并能帮助组织和重构代码，使其能够精确传达你对产品的深刻理解。

7.3.1 API设计

我们会为相册分享JSON服务器开发叫做RESTful JSON的服务器。单词REST是Rep-rentational State Transfer的缩写，表明可以从服务器上请求一个对象。REST API主要包含四种不同的核心操作（这和HTTP请求的方法保持一致性^[2]）：

- 创建（PUT）
- 检索（GET）
- 更新（POST）
- 删除（DELETE）

许多人喜欢把这些操作称为CRUD，它包含了API操作对象所需要的一切，而不论这些对象是相册、照片、评论或者是用户。尽管很难设计出一个十全十美的API方案，但我还是想尽我所能设计出一个至少符合我们直觉的目标方案，不会让客户端开发者抓耳挠腮。下面是一些设计RESTful接口时的原则：

- URL有两种基本类型，绝大多数设计出来的URL会是下面两种的变形：
 - 集合——例如，/albums
 - 集合中指定的条目——例如，/albums/italy2012
- 集合的名称应当是名词，最好是复数名词，如albums、

users或photos等。

- PUT/albums.json——HTTP请求主体中包含了新相册所需要的JSON数据。

- 指定集合中某个特定实例，分别使用GET和POST方法获取或更新对象：

- GET/albums/italy2012.json——请求返回该相册。
- POST/albums/italy2012.json——HTTP请求主体中包含了更新相册所需要的JSON数据。

- 使用DELETE删除某个对象。

- DELETE/albums/italy2012.json——删除该相册。
- 如果某个集合来自另一个集合，比如，照片和相册相关联，只需继续使用上面的模式即可：

- GET/albums/italy2012/photos.json——返回该相册中的所有照片。

- PUT/albums/italy2012/photos.json——向该相册中添加一张新照片。
- GET/albums/italy2012/photos/23482938424.json——获取指定ID的照片。

- 如果需要稍微改变获取数据集合的方式，比如分页或过滤，应该通过添加GET请求参数来实现：

- GET/albums.json?located_in=Europe——只获取地点是欧洲的相册。
- GET/albums/japan2010/photos.json?page=1&page_size=25——获取japan2010相册中第1页的25张照片。

- 需要对API版本化，以便将来对API进行较大的版本改动时能向后兼容，我们只需简单地更新版本号即可。因此，需要在API URL中添加/v1/前缀。
- 最后，在所有需要返回JSON数据的URL中添加.json后缀，这样客户端就会知道返回的数据格式。将来，我们也可以添加任何想支持的格式，如.xml等。

谨记以上原则，相册应用的新API就会紧紧围绕相册这个主题。如下所示：

```
/v1/albums.json  
/v1/albums/:album_name.json  
  
/pages/:page_name  
/templates/:template_name  
/content/:filename
```

我们为什么没有将版本号添加到静态内容的URL上？因为它们不是JSON服务器的接口，而是一些为Web浏览器客户端提供静态内容的帮助函数。用户总是会获取最新版本的静态文件，这样就能获取最新版本API的版本号。

使用全新的API设计更新server.js的工作量并不大。但是别忘记，还需要通过新的URL更新/content/下的JavaScript加载器文件！在继续改造之前，请先确认是否能正常运行。

7.3.2 模块

当所有API更新成全新的REST接口之后，整个应用项目变得非常清晰。现在，我们已经了解了相册和页面的具体功能，因此可以将这些功能添加到各自的模块中。

在应用文件夹下创建一个名叫handlers/的文件夹，用来存放最新的模块。在该文件夹下为相册创建albums.js文件，并将四个相册操作函数放入文件中。你可以在Github上阅读该项目下handlers_as_modues/文件夹中的源代码。实际上，我们就是将

`handle_get_album`和`handle_list_albums`函数集成到新的帮助函数`load_album`和`load_album_list`中，从而形成新的模块。建议你花一分钟阅读一下源代码，看它是如何快速简单地组织代码结构的。

回顾一下前文中帮助我们提取相册名称和查询参数等的函数。现在，可以删除它们并在合适的地方使用`req.params`和`req.query`。

同时，还有一些帮助我们简化处理发送成功和失败状态的函数，比如`send_success`、`send_failure`和`invalid_resource`。我们可以将它们放到`handlers/`目录下的`helpers.js`模块中，然后在`albums.js`文件顶部引用即可，内容如代码清单7.1所示。

代码清单7.1 帮助函数（`helpers.js`）

```
exports.version = '0.1.0';

exports.make_error = function(err, msg) {
    var e = new Error(msg);
    e.code = err;
    return e;
}

exports.send_success = function(res, data) {
    res.writeHead(200, {"Content-Type": "application/json"});
    var output = { error: null, data: data };
    res.end(JSON.stringify(output) + "\n");
}

exports.send_failure = function(res, code, err) {
    var code = (err.code) ? err.code : err.name;
    res.writeHead(code, { "Content-Type" : "application/json" });
    res.end(JSON.stringify({ error: code, message: err.message }) + "\n");
}

exports.invalid_resource = function() {
    return make_error("invalid_resource",
                      "the requested resource does not exist.");
}

exports.no_such_album = function() {
    return make_error("no_such_album",
                      "The specified album does not exist");
}
```

现在，如果需要引用`album`函数，只需在`server.js`文件的顶部添加`require`即可：

```
var album_hdldr = require('./handlers/albums.js');
```

然后，在路由处理程序中添加相应的相册函数，如下所示：

```
app.get('/v1/albums.json', album_hdldr.list_all);
app.get('/v1/albums/:album_name.json', album_hdldr.album_by_name);
```

同样，可以将创建页面的功能迁移到`handlers/`目录下的

pages.js文件中，如代码清单7.2所示。这部分工作实际上只是将原先的serve_page函数移植过来。

代码清单7.2 创建页面 (pages.js)

```
var helpers = require('./helpers.js'),
    fs = require('fs');

exports.version = "0.1.0";

exports.generate = function (req, res) {
    var page = req.params.page_name;

    fs.readFile(
        'basic.html',
        function (err, contents) {
            if (err) {
                send_failure(res, 500, err);
                return;
            }
            contents = contents.toString('utf8');

            // replace page name, and then dump to output.
            contents = contents.replace('{{PAGE_NAME}}', page);
            res.writeHead(200, { "Content-Type": "text/html" });
            res.end(contents);
        }
    );
};
```

可以更新页面的路由处理程序，如下所示：

```
var page_hdlr = require('./handlers/pages.js');
app.get('/pages/:page_name', page_hdlr.generate);
```

所有任务完成之后，server.js文件只剩下处理静态内容的函数，可以查看第7章的Github源代码，具体在handlers_as_modules/项目文件夹下。

现在，这个应用和原先的拥有相同的功能，但是代码却更加模块化，更具可读性。至此，我们应该已经知道如何向应用中添加功能了。

[2]根据HTTP RFC，一般情况下，POST用于创建资源，而PUT用于修改资源。——译者注

7.4 中间件功能

我曾提到过express是构建在connect这个中间件类库之上的。这些组件链会依次处理每个请求，并会在组件调用next函数之后结束调用。而路由处理程序则是这个组件链的一部分。express和connect还提供了一些其他实用组件，让我们一起看看其中一些有趣的组件。

7.4.1 基本用法

在express和connect中可以通过use方法来使用中间件组件。例如要使用logging中间件，可以写成：

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

app.use(express.logger());
/* etc ... */
```

调用express.logger函数后会返回一个函数，它会构成应用中过滤层（filter layering）的一部分。尽管express内置了一些组件，但是它还允许使用其他兼容的中间件组件。因此，我们可以使用connect提供的所有组件（前提是将connect添加到package.json文件的dependencies中）：

```
var express = require('express');
var connect = require('connect');
var app = express();

app.use(connect.compress());
/* etc ... */
```

由于添加和引入connect模块是一件麻烦的事，所以express已经重新暴露了connect中的所有中间件。因此，只需要从express模块中直接引用这些中间件即可，如下所示：

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

app.use(express.compress());
/* etc ... */
```

当然，也可以直接从npm或者其他地方下载并安装第三方组件：

```
var express = require('express');
var middle_something = require('middle_something');
var app = express();

app.use(middle_something());
/* etc ... */
```

7.4.2 配置

在特定的配置环境中使用一些中间件非常实用。有些中间件能够修改响应的速率或者输出大小，因此这会对用curl测试服务器造成困扰。因此，可以在express应用中使用configure方法并指定配置名称，方法中提供的函数会在指定的配置环境下被调用。如果不提供名称，则函数会被所有配置环境调用，当然也可以在函数之前使用多个配置名称，用逗号隔开，如下所示：

```
app.configure(function () {
  app.use(express.bodyParser());
});

app.configure('dev', function () {
  app.use(express.logger('dev'));
});
app.configure('production', 'staging', function () {
  app.use(express.logger());
});
```

在configure函数外设置的中间件或者路由适用于所有的配置环境。

要想在某个特定的配置环境下运行应用，可以通过设置NODE_ENV环境变量达到目的。在Mac和UNIX电脑中，绝大部分

shell都支持以下方式运行node应用：

```
NODE_ENV=production node program.js
```

在Windows平台下，只需在命令提示符中设置NODE_ENV环境变量，并运行node应用：

```
set NODE_ENV=production
node program.js
```

7.4.3 中间件执行顺序

中间件组件的分层和顺序非常重要。正如前文所说，express和connect会依次经过这些组件，直到找到可以处理请求的组件。要理解这之间是如何关联的，可以参考下面的简单应用示例：

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

app.use(express.logger('dev'))
    // move this to AFTER the next use() and see what happens!
    .use(express.responseTime())
    .use(function(req, res){
        res.end('hello world\n');
    })
    .listen(8080);
```

调用该应用：

```
curl -i localhost:8080/blargh
```

可以得到如下结果：

```
HTTP/1.1 200 OK
X-Powered-By: Express
X-Response-Time: 0ms
Date: Wed, 05 Dec 2012 04:11:43 GMT
Connection: keep-alive
Transfer-Encoding: chunked
```

```
hello world
```

现在，如果将responseTime中间件组件移至最后，会发生什

么？如下所示：

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

app.use(express.logger('dev'))
    // move this to AFTER the next use() and see what happens!
    .use(function(req, res){
        res.end('hello world\n');
    })
    .use(express.responseTime())
    .listen(8080);
```

我们注意到响应头X-Response-Time已经消失了！

```
HTTP/1.1 200 OK
X-Powered-By: Express
Date: Wed, 05 Dec 2012 04:14:05 GMT
Connection: keep-alive
Transfer-Encoding: chunked

hello world
```

负责打印"Hello World!"的函数会接收该请求，并全权处理。它会使用res.end关闭响应，responseTime中间件甚至都没有机会处理该请求。因此，在添加中间件组件的时候，需要考虑它们之间是如何为应用中的路由协调工作的。

7.4.4 静态文件处理

前文中已经为应用写过一些处理静态文件的Node代码，但其实完全没必要那样做。express（通过connect）提供了static中间件组件，可以为我们完成这一切。

要使用它，只需要将存放静态文件的根目录的路径名赋给中间件创建函数，如下所示：

```
app.use(express.static("/secure/static_site_files"));
```

如果将其放在URL路由函数前面，那么当请求进来时，该静态层就会接收URL，并将其附加到根目录名称后面，然后检测该文件是否存在。例如，如果请求/content/style.css，该中间件就会检

测`/secure/static_site_files/content/styles.css`是否存在且可读。如果是，则该中间件就会处理该文件，并停止调用下一层；如果不是，则调用`next`函数。

如果想设置多个静态文件目录，只需添加多个组件即可：

```
app.use(express.static("/secure/core_site_content"));
app.use(express.static("/secure/templates_and_html"));
```

注意，尽管在技术上可以使用Node中预定义变量`_dirname`（即当前正在执行的Node脚本的路径）处理应用文件夹下的内容，但这是一件非常糟糕的事情，千万不要使用。

`static`中间件没有提供任何安全机制，因此如果在前文中的相册应用代码中添加上

```
app.use(express.static(__dirname));
```

然后，就可以如预期一样正常访问`/content/style.css`和`/templates/home.html`，不会出现任何问题。但问题是，如果用户访问`/server.js`或者`/handlers/albums.js`，那么`static`中间件也同样会处理它们，因为这些文件也在`_dirname`根目录下。

也许你会问，为什么`static`中间件组件没有提供任何安全机制，其实这个问题有些不恰当——它根本没有那么复杂。如果想使用该组件，只需要将静态文件放到应用代码树的外部即可。事实上，我曾提到过，甚至可以让Node应用避免处理静态内容，而是选择将静态内容放到CDN上。因此，从一开始就尝试将这些组件与代码树分离的做法吧。

如果使用`static`中间件重写`server.js`，它会看起来和下面的代码一样。首先，需要将`content/`、`templates/`和`albums/`文件夹放置到另一个位置，以避免前面提到的安全问题。更新后的应用代码结构如下所示：

```
+ root_folder/
  + static/
    + albums/
    + content/
    + templates/
+ app/
  + handlers/
  + node_modules/
```

server.js文件如代码清单7.3所示。新版server代码的好处就是可以移除原先处理静态内容和模板的路由函数。

代码清单7.3 使用static中间件 (server.js)

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

var fs = require('fs'),
  album_hdrl = require('./handlers/albums.js'),
  page_hdrl = require('./handlers/pages.js'),
  helpers = require('./handlers/helpers.js');
app.use(express.static(__dirname + '/../static'));

app.get('/v1/albums.json', album_hdrl.list_all);
app.get('/v1/albums/:album_name.json', album_hdrl.album_by_name);
app.get('/pages/:page_name', page_hdrl.generate);
app.get('/pages/:page_name/:sub_page', page_hdrl.generate);

app.get("/", function (req, res) {
  res.redirect("/pages/home");
  res.end();
});

app.get('*', four_oh_four);

function four_oh_four(req, res) {
  res.writeHead(404, { "Content-Type" : "application/json" });

  res.end(JSON.stringify(helpers.invalid_resource()) + "\n");
}

app.listen(8080);
```

如果你足够细致，也许会注意到为/添加的新路由中，使用ServerResponse.redirect方法将请求重定向到/pages/home，以减少在浏览器端的重复输入。

7.4.5 POST数据、cookie和session

express和connect已经内置了帮助方法，可以解析请求的查询参数并保存到req.query对象中。在第4章中，我们通过数据流手动下载和解析表单POST数据。好消息是，我们不必手动做这一切了——中间件可以为我们实现这个功能，甚至还可以给应用设置

cookie和sesion，分别使用bodyParser、 cookieParser和session中间件完成这些功能：

```
app.use(express.bodyParser());
app.use(express.cookieParser());
```

第一个bodyParser中间件会解析所有的请求体数据，并放置到req.body对象中。如果请求的数据是urlencoded类型或者JSON格式，每个数据字段都会保存到req.body中作为对象的字段。cookieParser也是类似的工作原理，但是会把值放到req.cookies中。

要在返回的响应中设置cookie，可以使用ServerResponse对象中的cookie函数，设置名称和值（以及可选参数——过期时间）：

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

var app = express()
  .use(express.logger('dev'))
  .use(express.cookieParser())
  .use(function(req, res){
    res.cookie("pet", "Zimbu the Monkey",
      { expires: new Date(Date.now() + 86400000) });
    res.end(JSON.stringify(req.query) + "\n");
  })
  .listen(8080);
```

如果想要清空设置过的cookie，只需要使用ServerResponse对象中的clearCookie函数，指定想要移除的cookie的名称即可。

而设置session则会稍微复杂一点，需要提供存储session的空间和加密值所需的密钥：

```
var MemStore = express.session.MemoryStore;
app.use(express.session({ secret: "cat on keyboard",
  cookie: { maxAge: 1800000 },
  store: new MemStore()}));
```

在示例中，设置了密钥，然后可以使用connect模块提供的MemoryStore类存储session。这种方法十分方便易用，但是每次重启Node的时候都会重置session。如果期望每次重启服务器之后都能保存住session，可以在npm上找到一些其他的存储方式，如使用外部存储的memcache或者redis（具体可见下面两章内容）。默认情

况下，session cookie不会过期。如果想要添加有效期，可以在参数中添加有效期的毫秒数。这里，代码设置的session有效期为30分钟。

当添加session功能之后，session数据会填充到req.session中。而要想添加新的sesion数据，只需在该对象中添加新的值即可。它会持久化并随着响应返回：

```
var express = require('express');
var MemStore = express.session.MemoryStore;

var app = express()
    .use(express.logger('dev'))
    .use(express.cookieParser())
    .use(express.session({ secret: "cat on keyboard",
        // see what happens for value of maybe 3000
        cookie: { maxAge: 1800000 },
        store: new MemStore()}))
    .use(function (req, res){
        var x = req.session.last_access; // undefined when cookie not set
        req.session.last_access = new Date();
        res.end("You last asked for this page at: " + x);
    })
.listen(8080);
```

文件上传

express的bodyParser中间件组件已经自动添加了connect multipart组件，可以提供文件上传功能。如果使用multipart/form-data格式向应用上传文件，可以在req.files对象中找到请求中包含的所有上传文件：

```
var express = require('express');
var app = express()
    .use(express.logger('dev'))
    .use(express.bodyParser())
    .use(function (req, res){
        if (!req.files || !req.files.album_cover) {
            res.end("Hunh. Did you send a file?");
        } else {
            console.log(req.files);
            res.end("You have asked to set the album cover for "
                + req.body.albumid
                + " to '" + req.files.album_cover.name + "'\n");
        }
    })
.listen(8080);
```

我们仍然可以用curl测试该功能！可使用-F参数（表单数据）在请求中包含上传文件，使用@操作符指定上传的文件名：

```
curl -i -H "Expect:" --form 'album_cover=@oranges.jpg' \
--form albumid=italy2012 http://localhost:8080
```

还有一个小技巧，当curl期望服务器返回响应100Continue并等待客户端继续发送数据时，可以向curl传递参数-H"Expect："，通过这种方式，可以测试文件上传功能。

7.4.6 对PUT和DELETE更友好的浏览器支持

PUT（创建对象）和DELETE（删除对象）是REST API中的重要组成部分，客户端网页通过这些方法向服务器发送数据。例如，在jQuery中，要想删除一个评论，需要使用

```
$.ajax({
  url: 'http://server/v1/comments/234932598235.json',
  type: 'DELETE'
})
```

在大部分现代Web浏览器中，它都能正常工作，但在一些老版本的浏览器（IE 10之前的版本）中却不能正常工作。究其原因，是这些浏览器中 XMLHttpRequest对象的实现不支持AJAX请求中的PUT和DELETE方法。

一个通用且优雅的解决方案是添加一个真正想用的方法到新的请求头X-HTTP-Method-Override上，然后使用POST上传请求到服务器。jQuery代码如下所示：

```
$.ajax({
  beforeSend: function(xhr) {
    xhr.setRequestHeader('X-HTTP-Method-Override', 'DELETE');
  },
  url: 'http://server/v1/comments/234932598235.json',
  type: 'POST'
})
```

要想在服务器上支持这种解决方案，需要添加methodOverride中间件，如下所示：

```
app.use(express.methodOverride());
```

这样服务器会寻找该请求头，并将POST请求转换成DELETE或者PUT方法。因此，app.delete和app.put路由函数就能正确工作

了。

7.4.7 压缩输出

为了减少带宽成本，很多Web服务器都会在将数据发送到客户端之前，使用gzip或者其他类似的算法压缩输出数据。前提条件是客户端需要在HTTP中使用请求头Accept-Encoding表明能够接收压缩后的数据。如果客户端提供了该请求头且服务器也支持压缩功能，则需要在输出的响应头Content-Encoding中指定合适的算法，然后将压缩数据发送到客户端。

要在应用中使用这个特性，可以使用压缩中间件。它使用起来非常简单，会通过请求头判断客户端是否支持压缩数据来设置响应头，并压缩输出数据，如下所示：

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

app.use(express.logger('dev'));
app.use(express.compress());

app.get('/', function(req, res){
    res.send('hello world this should be compressed\n');
});

app.listen(8080);
```

该模块的唯一缺点就是它会影响开发测试（curl只会打印出压缩后的二进制数据，不够直观），因此，一般只会在生产环境下使用该中间件，如下所示：

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

app.use(express.logger('dev'));
app.configure('production', function () {
    app.use(express.compress());
});

app.get('/', function(req, res){
    res.send('hello world this should be compressed\n');
});

app.listen(8080);
```

7.4.8 HTTP基本身份验证

如果想在应用中使用HTTP基本身份验证（basic authentication），中间件可以帮助我们实现。尽管其由于过时和不安全而倍遭诟病，但如果基本身份验证与SSL和HTTPS配合起来使用的话，应用将会变得足够健壮和安全。

要使用它，设置basicAuth中间件即可，它提供一个函数用来接收并验证用户名和密码。当验证信息正确时，返回true；反之，则返回false，如下所示：

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

app.use(express.basicAuth(auth_user));
app.get('/', function(req, res){
    res.send('secret message that only auth\'d users can see\n');
});

app.listen(8080);

function auth_user(user, pass) {
    if (user == 'marcwan'
        && pass == 'this is how you get ants') {
        return true;
    } else {
        return false;
    }
}
```

如果现在启动并访问服务器，它会在返回数据之前询问用户名和密码。

7.4.9 错误处理

尽管我们可以对每个请求单独进行错误处理，但有时也希望有一个全局的错误处理来解决一些常见问题。要实现这一点，可以在app.use方法中提供一个包含四个参数的函数：

```
app.use(function (err, req, res, next) {
    res.status(500);
    res.end(JSON.stringify(err) + "\n");
});
```

该方法会放置到所有其他中间件和路由函数的后面，从而成为express在应用中调用的最后一个函数。在函数中可以检测具体错误，并决定返回给用户的数据。例如，将err对象或者抛出的Error对象转成适当的JSON格式并作为错误信息返回给用户：

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

app.get('/', function(req, res){
  throw new Error("Something bad happened");
  res.send('Probably will never get to this message.\n');
});

app.use(function (err, req, res, next) {
  res.status(500);
  res.end(err + "\n");
});

app.listen(8080);
```

最后，Node.js可以通过process对象在全局应用范围内指定错误处理函数：

```
process.on('uncaughtException', function (err) {
  console.log('Caught exception: ' + err);
});
```

但是，如果使用它进行打印日志或者诊断之外的工作，那就不是一个明智的选择。当该函数被调用时，基本可以断定Node已经不能正常工作，处于不稳定的状态，需要重启。因此，在这些使用场合中，一般会使用该函数打印错误日志和结束node进程：

```
process.on('uncaughtException', function (err) {
  console.log('Caught exception: ' + err);
  process.exit(-1);
});
```

因此，最好在可能出现错误的地方使用try/catch来捕捉错误；而在一些灾难性错误和完全不可预期的场景下，则需关闭进程并重启应用——最好是自动重启。

7.5 小结

在这一大章中，我们了解了许多有趣且实用的知识。本章介绍了express应用开发框架、中间件以及如何使用cookie和session，并将相册应用的代码模块化。使用静态文件服务中间件剥除了大量无用代码，并且在应用中添加文件压缩、用户验证和更好的错误处理机制。

现在，相册应用已经成型，但是还有一些不尽如人意的地方。相册中只有照片，却没有存放更多额外的相册或照片信息。带着这些疑问，接下来我们将会学习数据库相关内容——首先是NoSQL（CouchDB），然后是MySQL——学习如何在Node中使用它们。同时，我们还会看一些缓存解决方案，让这个小应用更加贴切实际。

第8章 数据库I：NoSQL（MongoDB）

到目前为止，我们已经为Web应用打下了坚实的基础，应用完全由express Web应用框架和Mustache模板引擎构建而成，接下来我们会用两个章节介绍如何为应用添加后端数据库。这两章中，我们会学习两种最常用的数据库。首先，在本章中学习目前流行的NoSQL数据库——MongoDB，它能快速简单地直接将JSON数据序列化到数据库中。本章会介绍MongoDB的基本使用方法，并更新相册处理程序，使其可以将相册和照片数据存储到数据库中。

我们之所以选择MongoDB而不是其他流行的NoSQL数据库（尤其是CouchDB），是因为MongoDB极易上手而且提供了超赞的查询功能，而其他数据库则较为复杂。由于本书的目的之一就是教会你使用Node.js，因此我选择了Node.js中最容易搭配使用的MongoDB，这样能让我们更专注于自己的初衷。

如果你偏好于MySQL或者其他关系型数据库，本章也依然值得一读，因为你可以看到如何将相册和照片处理程序从前文中的文件模块迁移到数据库模块中。而在下一章中，我们会讨论关系型数据库，同时也会提供一个MySQL版本的应用代码。

8.1 设置MongoDB

要在我们的Node.js应用中使用MongoDB，需要做两件事：安装数据库服务器以及配合Node使用。

8.1.1 安装MongoDB

要使用MongoDB，首先要从<http://mongodb.org>上下载二进制文件。大部分平台下的发行版为zip文件格式，我们可以将其解压到任何想运行MongoDB的目录下。

在bin/子文件夹下，可以找到一个名叫mongod的文件，这就是基本数据库服务器守护进程（daemon）。要想在开发环境下运行服务器，只需运行：

```
./mongod --dbpath /path/to/db/dir          # unix  
mongod --dbpath \path\to\db\dir           # windows
```

我们可以创建文件夹如~/src/mongodbs或者其他类似的文件夹来存储数据库数据。如果想要快速删除数据库并重启数据库服务器，只需按下Ctrl+C中止当前服务器进程并运行

```
rm /path/to/db/dir/* && ./mongod --dbpath /path/to/db/dir      # unix  
del \path\to\db\dir\*                                         # windows 1  
mongod --dbpath \path\to\db\dir                               # windows 2
```

要想测试是否安装成功，打开另一个终端窗口或命令提示符，运行mongo程序——一个类似于Node REPL的简易解释器：

```
Kimidori:bin marcw$ ./mongo  
MongoDB shell version: 2.2.2  
connecting to: test  
> show dbs  
local (empty)  
>
```

而如果想要部署到生产环境中，则需要进一步阅读MongoDB文档，学习相关的最佳实践和配置选项，包括主从复制、备份等。

8.1.2 在Node.js中使用MongoDB

在验证MongoDB安装成功之后，需要把它与Node.js桥接到一起。现在Node上最流行的驱动就是10gen提供的官方原生驱动mongodb，10gen就是开发出MongoDB的公司。同时也有专为MongoDB和Node.js准备的关系对象映射（relational-object mapping，ROM）框架mongoose，但是本书中我们只关注简单的驱动，因为它已经提供了我们想要的一切（相册应用的数据需求并不复杂），同时也会让代码降低与数据库的耦合（我们会在下章中看到）。

向package.json文件中添加如下代码来安装mongodb模块：

```
{  
  "name": "MongoDB-Demo",  
  "description": "Demonstrates Using MongoDB in Node.js",  
  "version": "0.0.1",  
  "private": true,  
  "dependencies": {  
    "async": "0.1.x",  
    "mongodb": "1.2.x"  
  }  
}
```

然后可以运行npm update来获取最新的驱动。更新完成后，查看node_modules文件夹，可以看到mongodb/子文件夹。

8.2 MongoDB数据结构

我们的应用中非常适合使用MongoDB，因为MongoDB本身就是使用JSON存储数据。当在数据库中创建或者添加数据时，只需要传一个JavaScript对象过去即可——这简直就是为Node量身打造的！

如果你以前使用过关系型数据库，可能会对数据库的一些术语比较熟悉，如数据库、表（table）和行（row）。而在MongoDB中，这些元素分别对应着数据库、集合（collection）和文档（document）。数据库可以包含多个集合，而每个集合由JSON文档表示。

MongoDB中的所有对象都有一个唯一标识符_id。原则上，只要是唯一值，_id可以是任何类型。如果不手动提供该值，则MongoDB会自动为我们创建一个：它就是ObjectID类的一个实例。如果打印出来，它会返回一个24位的十六进制字符串，比如50da 80c0 4d40 3ebd a700 0012。

8.2.1 全是JavaScript的世界

我们需要为相册应用创建两个文档类型：相册和照片。我们不需要使用MongoDB自动生成的_id值，因为每个文档都已经有唯一的标识信息了：对于相册而言，相册名称（album_name）是唯一的；而对于照片而言，相册名称（album_name）和照片名称（filename）的组合也是唯一的。

因此，存储的相册数据如下所示：

```
{ _id: "italy2012",
  name:"italy2012",
  title:"Spring Festival in Italy",
  date:"2012/02/15",
  description:"I went to Italy for Spring Festival." }
```

照片文档则类似于：

```
{ _id: "italy2012_picture_01.jpg",
  filename: "picture_01.jpg",
  albumid: "italy2012",
  description: "ZOMGZ it's Rome!",
  date: "2012/02/15 16:20:40" }
```

当尝试向集合中添加重复的_id值时，会导致MongoDB报错。

通过这种特性，可以保证相册是唯一的，同时每个相册中的图片文件也是唯一的。

8.2.2 数据类型

大多数情况下，在JavaScript中使用MongoDB非常舒适自然、简单直接。但是，还是会有一些特殊的使用场景会引起麻烦，值得我们注意。

回到第2章，JavaScript中所有数字都是64位双精度浮点数。它提供了53位整数精度，但是经常会有使用64位整数值的需求。当降低精度和准确度不可接受的时候，可以使用mongodb驱动为Node准备的Long类。它的构造器会接收一个字符串值，可供我们进行64位整数值的操作。它还为整数值提供了一些方法，如toString、compare和add/subtract/multiply，来模拟整数值的常用操作。

Java Script还提供了Binary类，可以让我们在文档中存储二进制数据。我们可以向它的构造器中传递字符串或者一个Buffer对象实例，这些数据会以二进制数据格式存储。当重新加载文档的时候，返回的值会包含一些额外的元数据，用来描述Mongo是如何在集合中存储的。

最后，我想介绍下Timestamp类，可以用来将时间日期存储到数据库文档中，只需要在写数据的时候添加即可！如下所示：

```
{ _id: "unique_identifier1234",
  when: new TimeStamp(),
  what: "Good News Everyone!" };
```

要想查看mongodb模块提供的所有数据类型帮助列表，可以查看github.com/mongodb/node-mongodb-native上提供的说明文档或者阅读

`node_modules/mongodb/node_modules/bson/lib/bson/`下的源代码。

8.3 理解基本操作

至此，我们可以使用MongoDB和Node.js处理事情了。在每个文件的顶部使用mongodb模块，如下所示：

```
var Db = require('mongodb').Db,
    Connection = require('mongodb').Connection,
    Server = require('mongodb').Server;
```

而在一些场景中，会使用其他的数据类型，如Long或者Binary，我们可以通过mongodb模块引用它们：

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

var b = new mongodb.Binary(binary_data);
var l = new mongodb.Long(number_string);
```

8.3.1 连接并创建数据库

要想连接MongoDB服务器，需要使用Server和Db类。可以选择使用Connection类获取服务器监听的默认端口号。具体创建共享db对象的代码，如下所示：

```
var Db = require('mongodb').Db,
    Connection = require('mongodb').Connection,
    Server = require('mongodb').Server;

var host = process.env['MONGO_NODE_DRIVER_HOST'] != null
    ? process.env['MONGO_NODE_DRIVER_HOST'] : 'localhost';

var port = process.env['MONGO_NODE_DRIVER_PORT'] != null
    ? process.env['MONGO_NODE_DRIVER_PORT'] : Connection.DEFAULT_PORT;

var db = new Db('PhotoAlbums',
    new Server(host, port,
        { auto_reconnect: true,
            poolSize: 20}),
    { w: 1 });
```

MongoDB和mongodb模块在处理写数据和保持数据一致性上非常灵活。可以向数据库构造器传递标识{w:1}，以保证在调用提供给数据库操作的回调函数之前数据已经成功写入。可以在复制环境中指定更大的数字，或者在不关心写操作是否完成时指定0。

我们不需要为每一个应用请求创建一个新的Db对象，也不需要关心多个用户同时操作数据库服务器的情况：mongodb模块使用连接池来同时处理多个请求。我们可以通过调整Server构造器中的poolSize参数的值来选择MongoDB服务器同时连接的个数，如上述代码所示。

想要创建或者使用某个已经存在的数据库，可以将数据库名称传递给Db对象的构造器。如果数据库不存在，则会自动创建。

8.3.2 创建集合

正如前文所说，MongoDB中的集合等价于关系型数据库系统中的表，我们可以使用Db对象中的collection或者createCollection方法创建集合。通过在第二个参数中指定{safe:true}选项，可以控制如何处理存在或不存在的集合（见表8.1）。

表 8.1 创建

函 数	{ safe: true }	功 能
collection	是	若集合存在，打开集合；反之，返回错误
collection	否	若集合存在，打开集合；反之，在插入第一条数据时创建集合
createCollection	是	若集合不存在，创建集合；反之，返回错误
createCollection	否	若集合不存在，创建集合；反之，返回一个已存在的集合

想要使用{safe:true}选项，代码如下所示：

```
db.collection("albums", { safe: true }, function (err, albums) {
    // ... do stuff
});
```

下面这段代码是最常见的创建或打开集合的方式，但是它也非常简单：

```
db.collection("albums", function (err, albums) {
    if (err) {
        console.error(err);
        return;
    }

    // albums can now be used to do stuff ...
});
```

8.3.3 向集合中插入文档

如果要在集合中插入新数据，可以使用insert方法，如下所示：

```
var album = { _id: "italy2012",
    name: "italy2012",
    title: "Spring Festival in Italy",
    date: "2012/02/15",
    description: "I went to Italy for Spring Festival." };

albums.insert(album, { safe: true }, function (err, inserted_doc) {
    if (err && err.name == "MongoError" && err.code == 11000) {
        console.log("This album exists already, sorry.");
        return;
    } else if (err) {
        console.log("Something bad happened.");
        return;
    }
    // continue as normal
});
```

可以看到，代码中指定了文档中的_id字段。如果不指定，MongoDB会自动生成。如果有已经有文档包含相同的_id值，回调函数就会返回错误信息。我们可以从代码中看到在insert函数中传递了{safe:true}参数选项，以确保写数据成功。但是，如果是为数据分析应用编写存储，则允许偶尔丢失数据，所以可以省略该选项。

我们还可以同时向集合中插入多份文档，只需向insert函数中传递数组即可：

```
var docs = [{ _id: "italy2012",
    name: "italy2012",
    title: "Spring Festival in Italy",
    date: "2012/02/15",
    description: "I went to Italy for Spring Festival." },
    { _id: "australia2010",
        name: "australia2010",
        title: "Vacation Down Under",
        date: "2010/10/20",
        description: "Visiting some friends in Oz!" },
    { _id: "japan2010",
        name: "japan2010",
        title: "Programming in Tokyo",
        date: "2010/06/10",
        description: "I worked in Tokyo for a while."
];
albums.insert(docs, { safe: true }, callback);
```

8.3.4 更新文档内容

要更新文档，可以调用集合中的update方法。第一个参数用来匹配一个或一组文档，第二个参数是对象描述（object description），表明如何修改已匹配的文档。对象描述包含命令格

式和需要修改的一个或多个字段：

```
photos.update({ filename: "photo_03.jpg", albumid: "japan2010" },
    { $set: { description: "NO SHINJUKU! BAD DOG!" } },
    { safe: true },
    callback);
```

该对象描述中，使用了\$set命令，MongoDB会使用提供的新值更新字段。上述代码中，使用新值更新了description字段。

MongoDB提供了许多不同的命令，其中有趣的更新命令如表8.2所示：

表 8.2 更新命令

命 令	功 能
\$set	设置指定字段值
\$inc	增加指定字段值
\$rename	修改指定字段名称
\$push	若字段为数组，向数组尾部添加新值
\$pushAll	若字段为数组，向数组尾部添加多个新值
\$pop	移除数组字段的最后一个元素
\$pull	移除数组字段中的指定值

8.3.5 删除集合中的文档

要想删除集合中的数据，可以使用集合对象中的remove方法。可以指定一组字段来匹配一个或一组文档：

```
photos.remove({ filename: "photo_04.jpg", albumid: "japan2010" },
    { safe: true },
    callback);
```

如果不需要确认是否删除成功，可以跳过safe:ture选项和回调函数。

```
photos.remove({ filename: "photo_04.jpg", albumid: "japan2010" });
```

当然，也可以简单地通过调用不带任何参数的remove函数，删除集合中的所有文档：

```
photos.remove(); // DANGER ZONE!!
```

8.3.6 查询集合

到目前为止，之所以MongoDB成为最流行的NoSQL数据库引擎，是因为它能够像传统的SQL数据库查询一样在集合中找到文档。

而这一切，只需要集合对象中的find函数就能完成。

基本查询

开始之前需要注意，find方法本身并不做任何与查询相关的工作；它只是设置了查询结果的游标（cursor）（即可以用来迭代查询结果的对象）。在调用nextObject、each、toArray和streamRecords中任何一个函数之前，查询不会真正执行，游标的内容也没有真正生成出来。

顾名思义，前三个操作分别为：调用nextObject可以获得一个文档；调用each可以迭代查询到的文档数组；toArray可以获取所有文档，并作为回调函数中的参数：

```
var cursor = albums.find();

cursor.nextObject(function(err, first_match) {});
cursor.each(function(err, document) {});
cursor.toArray(function(err, all_documents) {});
```

如果直接调用不带任何参数的find方法，就会匹配集合中所有的文档。

我们可以使用游标上的streamRecords方法来创建Stream对象，这样就可以像使用其他数据流一样使用它：

```
var stream = collection.find().streamRecords();
stream.on("data", function(document) {});      // why data and not readable? See text!
stream.on("end", function() {});
```

这是获取大量数据集的最佳方式，因为相较于toArray这些方法而言，它占用更少的内存空间，而toArray则会在一个数据块中返回所有文档。在编写本书的时候，mongodb模块还没有更新到Node.js中最新的Stream事件模型"readable"，而这极有可能在你使用的时候已经更新了。因此，还是双击鼠标，亲自去验证一下吧（最好的选择莫过于驱动的官网了）：

<http://mongodb.github.com/mongodb/node-mongodb-native/>。

要想在集合中找到指定的文档，可以在find函数的第一个参数中

指定需要匹配的字段：

```
photos.find({ albumid: "italy2012" }).toArray(function (err, results));
```

我们还可以在find查询中使用类似于前面update函数中的操作符。例如，现在有一些关于个人信息的文档，想要找到所有年龄大于20岁的人，可以使用：

```
users.find({ age: { $gt: 20 } }).toArray(callback);
```

还可以使用逻辑操作符\$and或者\$or将这些操作符组合起来，以提供更强大的查询功能。例如，想要返回所有年龄在20岁到40岁（包括在内）之间的用户，可以使用

```
users.find({ $and: [ { age: { $gte: 20 } }, { age: { $lte: 40 } } ] });
```

其他常见操作符可见表8.3。想要获取所有的操作符列表，可以检索MongoDB查询文档。

表 8.3 查询操作符

操作符	含义
\$ne	不相等
\$lt	小于
\$lte	小于或等于
\$gt	大于
\$gte	大于或等于
\$in	若字段值在给定的数组中，则匹配
\$nin	若字段值不在给定的数组中，则匹配
\$all	若给定的字段是数组，且包含所有给定的数组值，则匹配

查询MongoDB生成的ID

正如我在前文中提到的一样，如果插入的文档中没有提供_id字段，MongoDB会自动为我们生成一个。这些自动生成的_id字段都是ObjectId类型，并且是24位16进制字符串。

但问题是，如果使用这些字段，则根本无法通过指定ID字符串的值来找到它们。需要将字符串包到ObjectId类的构造器里，如下所示：

```
var ObjectId = require('mongodb').ObjectId;
var idString = '50da9d8d138cbc5da9000012';
collection.find({ _id: new ObjectId(idString) }, callback);
```

如果执行下面的代码，则无法得到任何匹配的结果：

```
collection.find({ _id: idString }, callback);
```

在本书应用的开发过程中不会遇到这个问题，因为我们使用的是自己的_id字段。

进一步改进查询

要将页面进行分页和排序，需要操作或者修改find操作的结果。在调用任何生成游标的函数前，mongodb模块可以让我们在find操作后面链式调用其他函数来实现这些功能。

最常用的方法包括skip、limit和sort。第一个方法指定在返回数据集前有多少文档需要跳过；第二个方法用来控制执行跳过后返回的文档数；最后一个用来整理和排序——支持在多个字段上的排序。

因此，要想将相册中所有的照片文档按照日期进行升序排列，代码如下：

```
photos.find({ albumid: "italy2012" })
  .sort({ date: 1 })          // 1 is asc, -1 is desc
  .toArray(function (err, results));
```

假设每个页面有20张照片，要获取第三页的照片，需要使用下述代码：

```
photos.find({ albumid: "italy2012" })
  .sort({ date: 1 })          // 1 is asc, -1 is desc
  .skip(40)
  .limit(20)
  .toArray(function (err, results) { });
```

同样，任何在toArray函数之前调用的函数只是设置结果游标，在最后一个函数调用之前，这些查询是不会被执行的。

我们还可以在sort函数的多个字段上排序：

```
collection.find()
  .sort({ field1: -1, field2: 1 })
  .toArray(callback);
```

8.4 更新相册应用

学会Node.js中使用MongoDB的基本操作之后，可以进一步更新相册应用，从而使用数据库替代简单的文件系统来存储相册和照片信息。这里，我们依然将图片文件存储在硬盘上，但在真实的生产环境中，可以将它们存储到一些存储服务器或者内容分发网络（content delivery network，CDN）中。

8.4.1 编写基本操作

首先，我们需要向应用中添加一个新文件夹data/，可以将一些后端的基本操作放在该文件夹下。我们创建一个名叫backend_helpers.js的文件，包含错误处理、生成错误、验证参数和一些其他的常用后端操作。这些操作都非常简单实用，你可以在Github上查看其源代码。

创建配置文件

在应用根目录下，创建一个名叫local.config.js的新文件，包含以下JavaScript：

```
exports.config = {
  db_config: {
    host: "localhost",
    // use default "port"
    poolSize: 20
  },
  static_content: "../static/"
};
```

只要引用该文件，就能确保所有的配置选项都能保存在一起。这样，只需要在这个文件中稍做修改，而不需要在代码中翻来覆去，就能更新配置。

创建数据库和集合

接下来，在data/下创建db.js文件。该文件用来创建相册应用所需的数据库连接和集合。同时，在该文件中，创建PhotoAlbums数据库连接也使用到了local.config.js文件中的配置信息：

```
var Db = require('mongodb').Db,
    Connection = require('mongodb').Connection,
    Server = require('mongodb').Server,
    async = require('async'),
    local = require("../local.config.js");

var host = local.config.db_config.host
    ? local.config.db_config.host
    : 'localhost';
var port = local.config.db_config.port
    ? local.config.db_config.port
    : Connection.DEFAULT_PORT;
var ps = local.config.db_config.poolSize
    ? local.config.db_config.poolSize : 5;

var db = new Db('PhotoAlbums',
    new Server(host, port,
        {
            auto_reconnect: true,
            poolSize: ps}),
    { w: 1 });


```

现在，要获取相册和照片的集合，需要打开数据库连接，并确保这些集合都存在。向db.js中添加相关函数，如下所示：

```
exports.init = function (callback) {
    async.waterfall([
        // 1. open database connection
        function (cb) {
            db.open(cb);
        },
        // 2. create collections for our albums and photos. if
        //     they already exist, then we're good.
        function (opened_db, cb) {
            db.collection("albums", cb);
        },
        function (albums_coll, cb) {
            exports.albums = albums_coll;
            db.collection("photos", cb);
        },
        function (photos_coll, cb) {
            exports.photos = photos_coll;
            cb(null);
        }
    ],
    // we'll just pass results back to caller, so can skip results fn
    [callback]);
};

exports.albums = null;
exports.photos = null;
```

我们可以看到albums和photos已经成为db.js中的输出对象，因

此可以在任何时候获取到它们：

```
var db = require('./db.js');
var albums = db.albums;
```

最后，需要确保db.init函数在应用启动前已经被调用，因此，需要将server.js中的调用

```
app.listen(8080);
```

替换成

```
db.init(function (err, results) {
  if (err) {
    console.error("** FATAL ERROR ON STARTUP: ");
    console.error(err);
    process.exit(-1);
  }
  app.listen(8080);
});
```

[创建相册](#)

而创建新相册的基本代码如下所示：

```

exports.create_album = function (data, callback) {
    var final_album;
    var write_succeeded = false;
    async.waterfall([
        function (cb) {
            try {
                backhelp.verify(data,
                    [ "name", "title", "date", "description" ]);
                if (!backhelp.valid_filename(data.name))
                    throw invalid_album_name();
            } catch (e) {
                cb(e);
            }
            cb(null, data);
        },
        // create the album in mongo.
        function (album_data, cb) {
            var write = JSON.parse(JSON.stringify(album_data));
            write._id = album_data.name;
            db.albums.insert(write, { w: 1, safe: true }, cb);
        },
        // make sure the folder exists.
        function (new_album, cb) {
            write_succeeded = true;
            final_album = new_album[0];
            fs.mkdir(local.config.static_content
                + "albums/" + data.name, cb);
        }
    ],
    function (err, results) {
        if (err) {
            if (write_succeeded)
                db.albums.remove({ _id: data.name }, function () {});
            if (err instanceof Error && err.code == 11000)
                callback(backhelp.album_already_exists());
            else if (err instanceof Error && err.errno != undefined)
                callback(backhelp.file_error(err));
            else
                callback(err);
        } else {
            callback(err, err ? null : final_album);
        }
    });
});

```

尽管async模块让代码看起来有点“长”，但我们能看出代码变得整洁了许多。所有的异步操作都被表示成任务序列，之后async会为我们打理所有的细节！

这里可以使用一个小技巧来克隆一个对象：

```
var write = JSON.parse(JSON.stringify(album_data));
```

可以看出，序列化后再反序列化一个对象是JavaScript中最快的克隆对象的方式之一。之所以在之前的代码中克隆对象，是因为我们不想修改“不属于我们”的对象本身。直接修改函数中的对象是不合适的（或者说是彻底错误的），因此在添加_id字段前会先快速克隆

该对象。注意，backend_helpers.js类似于前文中的helpers.js，只是简单的后端（在data/文件夹下）帮助函数。

查询相册

使用指定的名称来查询一个相册，非常简单：

```
exports.album_by_name = function (name, callback) {
    db.albums.find({ _id: name }).toArray(function (err, results) {
        if (err) {
            callback(err);
            return;
        }

        if (results.length == 0) {
            callback(null, null);
        } else if (results.length == 1) {
            callback(null, results[0]);
        } else {
            console.error("More than one album named: " + name);
            console.error(results);
            callback(backutils.db_error());
        }
    });
};
```

从代码中可以看出，我们一般会将更多的时间花在错误处理和验证上。

相册列表

类似的，列出所有的相册信息也非常简单：

```
exports.all_albums = function (sort_field, sort_desc, skip, count, callback) {
    var sort = {};
    sort[sort_field] = sort_desc ? -1 : 1;
    db.albums.find()
        .sort(sort)
        .limit(count)
        .skip(skip)
        .toArray(callback);
};
```

获取相册中的照片

获取指定相册下的所有照片信息，同样轻而易举：

```
exports.photos_for_album = function (album_name, pn, ps, callback) {
    var sort = { date: -1 };
    db.photos.find({ albumid: album_name })
        .skip(pn)
        .limit(ps)
        .sort("date")
        .toArray(callback);
};
```

向相册中添加照片

事实上，稍微有些复杂的操作是将照片添加到相册中。该功能会多花费一些时间，因为需要将上传的临时文件拷贝到最终的 static/albums/ 文件夹下：

```
exports.add_photo = function (photo_data, path_to_photo, callback) {
    var final_photo;
    var base_fn = path.basename(path_to_photo).toLowerCase();
    async.waterfall([
        function (cb) {
            try {
                backhelp.verify(photo_data,
                    [ "albumid", "description", "date" ]);
                photo_data.filename = base_fn;
                if (!backhelp.valid_filename(photo_data.albumid))
                    throw invalid_album_name();
            } catch (e) {
                cb(e);
            }
            cb(null, photo_data);
        },
        // add the photo to the collection
        function (pd, cb) {
            pd._id = pd.albumid + "_" + pd.filename;
            db.photos.insert(pd, { w: 1, safe: true }, cb);
        },
        // now copy the temp file to static content
        function (new_photo, cb) {
            final_photo = new_photo[0];
            var save_path = local.config.static_content + "albums/"
                + photo_data.albumid + "/" + base_fn;
            backhelp.file_copy(path_to_photo, save_path, true, cb);
        }
    ],
    function (err, results) {
        if (err && err instanceof Error && err(errno != undefined)
            callback(backhelp.file_error(err));
        else
            callback(err, err ? null : final_photo);
    });
};
```

8.4.2 修改JSON服务器的API

接下来，需要为JSON服务器添加两个新的API函数，以便创建相册和添加照片：

```
app.put('/v1/albums.json', album_hdrl.create_album);
app.put('/v1/albums/:album_name/photos.json', album_hdrl.add_photo_to_album);
```

还好express让这个过程变得非常简单，添加这两行代码后，API就扩展了新功能，现在需要更新相册处理程序来支持这些新

特性。

由于API现在还不支持上传数据，包括文件和POST数据，因此需要添加一些其他的中间件来支持这个功能。于是，将这些代码添加到server.js文件顶部：

```
app.use(express.logger('dev'));
app.use(express.bodyParser({ keepExtensions: true }));
```

代码中，我们添加了日志功能和bodyParser功能，bodyParser可以将请求的数据保存到req.body和req.files对象中。注意，需要在bodyParser中间件中指定保留文件扩展名。默认情况下，它会移除扩展名。

8.4.3 更新处理程序

现在，我们已经为相册和照片操作添加了数据库功能，但是还需要修改相册处理程序，以替换现有的文件系统操作。

帮助类

首先，需要创建两个帮助类。其中Photo类如下所示：

```
function Photo (photo_data) {
    this.filename = photo_data.filename;
    this.date = photo_data.date;
    this.albumid = photo_data.albumid;
    this.description = photo_data.description;
    this._id = photo_data._id;
}
Photo.prototype._id = null;
Photo.prototype.filename = null;
Photo.prototype.date = null;
Photo.prototype.albumid = null;
Photo.prototype.description = null;
Photo.prototype.response_obj = function() {
    return {
        filename: this.filename,
        date: this.date,
        albumid: this.albumid,
        description: this.description
    };
};
```

其中最有趣的函数当属response_obj函数。因为理论上Photo类可以包含一个照片所有的信息，而当把它作为JSON数据传递给API调用者时，有些数据是不想包含在其中的。假设有一个User对象，我们一般都会剔除密码和其他敏感数据。

一个基本的Album对象应当如下所示：

```
function Album (album_data) {
    this.name = album_data.name;
    this.date = album_data.date;
    this.title = album_data.title;
    this.description = album_data.description;
    this._id = album_data._id;
}

Album.prototype._id = null;
Album.prototype.name = null;
Album.prototype.date = null;
Album.prototype.title = null;
Album.prototype.description = null;

Album.prototype.response_obj = function () {
    return { name: this.name,
              date: this.date,
              title: this.title,
              description: this.description };
};
```

接下来，让我们看下处理程序如何使用前文中所写的相册基本操作。

创建相册

编写应用过程中，检测和捕捉错误的代码量往往要大于正常操作的代码量，这才是良好的代码风格。很多书籍和教程都忽略了这点，也许这就是世界上有那么多糟糕代码的原因之一。

```
var album_data = require('../data/album.js');
// ... etc ...
exports.create_album = function (req, res) {
    async.waterfall([
        function (cb) {
            if (!req.body || !req.body.name || !helpers.valid_filename(req.body.name))
            {
                cb(helpers.no_such_album());
                return;
            }
            cb(null);
        },
        function (cb) {
            album_data.create_album(req.body, cb);
        }
    ],
    function (err, results) {
        if (err) {
            helpers.send_failure(res, err);
        } else {
            var a = new Album(results);
```

```
        helpers.send_success(res, {album: a.response_obj()});
    });
});
};
```

根据名称检索相册

再次强调，错误检测和处理占了所有工作量的百分之九十以上。

这里，我高亮显示了调用后端获取相册数据的代码：

```
exports.album_by_name = function (req, res) {
  async.waterfall([
    function (cb) {
      if (!req.params || !req.params.album_name)
        cb(helpers.no_such_album());
      else
        album_data.album_by_name(req.params.album_name, cb);
    }
  ],
  function (err, results) {
    if (err) {
      helpers.send_failure(res, err);
    } else if (!results) {
      helpers.send_failure(res, helpers.no_such_album());
    } else {
      var a = new Album(album_data);
      helpers.send_success(res, { album: a.response_obj() });
    }
  });
};
```

相册列表

在相册应用中，每次只获取25条相册数据，这样页面不会太复杂。如果需要，可以将其改成通过查询参数进行设置：

```
exports.list_all = function (req, res) {
  album_data.all_albums("date", true, 0, 25, function (err, results) {
    if (err) {
      helpers.send_failure(res, err);
    } else {
      var out = [];
      if (results) {
        for (var i = 0; i < results.length; i++) {
          out.push(new Album(results[i]).response_obj());
        }
      }
      helpers.send_success(res, { albums: out });
    }
  });
};
```

让处理程序和数据库代码分离开来会增加一些额外的工作（其实并不多），但是这样会让后端变得更加灵活。在下一章中，我们会将相册和照片的数据存储迁移到另一个数据库系统中，而处理程序却不

需要做任何修改！只需要修改data/文件夹下的类即可。

获取相册中所有照片

代码清单8.1中展示的是如何浏览相册中的照片。它包含两个新方法：exports.photos_for_album和Album对象中的photos函数。这些函数中最复杂的部分就是处理分页和切割照片数组。

代码清单8.1 获取相册中所有照片

```
Album.prototype.photos = function (pn, ps, callback) {
    if (this.album_photos != undefined) {
        callback(null, this.album_photos);
        return;
    }
    album_data.photos_for_album(
        this.name,
        pn, ps,
        function (err, results) {
            if (err) {
                callback(err);
                return;
            }
            var out = [];
            for (var i = 0; i < results.length; i++) {
                out.push(new Photo(results[i]));
            }
            this.album_photos = out;
            callback(null, this.album_photos);
        }
    );
};

exports.photos_for_album = function(req, res) {
    var page_num = req.query.page ? req.query.page : 0;
    var page_size = req.query.page_size ? req.query.page_size : 1000;

    page_num = parseInt(page_num);
    page_size = parseInt(page_size);
    if (isNaN(page_num)) page_num = 0;
    if (isNaN(page_size)) page_size = 1000;

    var album;
    async.waterfall([
        function (cb) {
            // first get the album.
            if (!req.params || !req.params.album_name)
                cb(helpers.no_such_album());
            else
```

```
        album_data.album_by_name(req.params.album_name, cb);
    },

    function (album_data, cb) {
        if (!album_data) {
            cb(helpers.no_such_album());
            return;
        }
        album = new Album(album_data);
        album.photos(page_num, page_size, cb);
    },
    function (photos, cb) {
        var out = [];
        for (var i = 0; i < photos.length; i++) {
            out.push(photos[i].response_obj());
        }
        cb(null, out);
    }
],
function (err, results) {
    if (err) {
        helpers.send_failure(res, err);
        return;
    }
    if (!results) results = [];
    var out = { photos: results,
                album_data: album.response_obj() };
    helpers.send_success(res, out);
});
```

添加照片

最后，编写添加照片的API，如代码清单8.2所示。该API会向Album对象中添加一个新方法。

代码清单8.2 使用API添加照片

```
Album.prototype.add_photo = function (data, path, callback) {
    album_data.add_photo(data, path, function (err, photo_data) {
        if (err)
            callback(err);
        else {
            var p = new Photo(photo_data);
            if (this.all_photos)
                this.all_photos.push(p);
            else
                this.app_photos = [ p ];
            callback(null, p);
        }
    });
};
```

```

exports.add_photo_to_album = function (req, res) {
  var album;
  async.waterfall([
    function (cb) {
      if (!req.body)
        cb(helpers.missing_data("POST data"));
      else if (!req.files || !req.files.photo_file)
        cb(helpers.missing_data("a file"));
      else if (!helpers.is_image(req.files.photo_file.name))
        cb(helpers.not_image());
      else
        album_data.album_by_name(req.params.album_name, cb);
    },
    function (album_data, cb) {
      if (!album_data)
        cb(helpers.no_such_album());
      return;
    }
    album = new Album(album_data);
    req.body.filename = req.files.photo_file.name;
    album.add_photo(req.body, req.files.photo_file.path, cb);
  ],
  function (err, p) {
    if (err) {
      helpers.send_failure(res, err);
      return;
    }
    var out = { photo: p.response_obj(), album_data: album.response_obj() };
    helpers.send_success(res, out);
  });
};

```

8.4.4 为应用添加新页面

目前为止，JSON服务器已经使用MongoDB完成应用中相册和照片的存储功能。剩下需要做的就是添加几个新页面，可以让用户通过Web界面创建新相册或向相册中添加新照片。现在，我们着手解决这个问题。

定义页面URL

这里，将两个新页面分别放到文件夹/pages/admin/add_album和/pages/admin/add_photo下。幸运的是，我们不需要为此在express应用中修改URL处理程序。

创建相册

不要忘记，在使用Mustache模板的网站中，每个页面都需要两个文件：

- JavaScript加载器
- HTML模板文件

添加相册页面的加载器代码微不足道，因为只需要模板文件，而不需要从服务器加载任何JSON数据，如代码清单8.3所示。

代码清单8.3 admin_add_album.js

```
$(function(){
    var tmpl, // Main template HTML
        tdata = {} // JSON data object that feeds the template

    // Initialize page
    var initPage = function() {

        // Load the HTML template
        $.get("/templates/admin_add_album.html", function (d){
            tmpl = d;
        });

        // When AJAX calls are complete parse the template
        // replacing mustache tags with vars
        $(document).ajaxStop(function () {
            var renderedPage = Mustache.to_html( tmpl, tdata );
            $("body").html( renderedPage );
        });
    };
});
```

添加相册的HTML页面代码有一点复杂，因为需要用JavaScript实现Ajax表单提交，如代码清单8.4所示。而代码中的dateString变量是为了确保时间格式一直是yyyy/mm/dd，而不会偶尔出现yyyy/m/d的情况。

代码清单8.4 admin_add_album.html

```
<form name="create_album" id="create_album"
      enctype="multipart/form-data"
      method="PUT"
      action="/v1/albums.json">

    <h2> Create New Album: </h2>
    <dl>
      <dt>Album Name:</dt>
      <dd><input type='text' name='name' id="name" size='30' /></dd>
      <dt>Title:</dt>
      <dd><input id="photo_file" type="text" name="title" size="30" /></dd>
      <dt>Description:</dt>
      <dd><textarea rows="5" cols="30" name="description"></textarea></dd>
    </dl>
    <input type="hidden" id="date" name="date" value="" />
</form>
```

```

<input type="button" id="submit_button" value="Upload"/>

<script type="text/javascript">

    $("input#submit_button").click(function (e) {
        var m = new Date();
        var dateString =
            m.getUTCFullYear() + "/" +
            ("0" + (m.getUTCMonth()+1)).slice(-2) + "/" +
            ("0" + m.getUTCDate()).slice(-2) + " " +
            ("0" + m.getUTCHours()).slice(-2) + ":" +
            ("0" + m.getUTCMinutes()).slice(-2) + ":" +
            ("0" + m.getUTCSeconds()).slice(-2);

        $("input#date").val(dateString);

        var json = "{ \"name\": \"" + $("input#name").val() +
            "\", \"date\": \"" + $("input#date").val() +
            "\", \"title\": \"" + $("input#title").val() +
            "\", \"description\": \"" + $("#textarea#description").val() +
            "\" }";

        $.ajax({
            type: "PUT",
            url: "/v1/albums.json",
            contentType: 'application/json', // request payload type
            "content-type": "application/json", // what we want back
            data: json,
            success: function (resp) {
                alert("Success! Going to album now");
                window.location = "/pages/album/" + $("input#name").val();
            }
        });
    });
</script>

```

向相册中添加照片

要想向相册中添加一张新照片，必须编写更加复杂的代码。在加载器中，需要一个所有相册的列表，这样用户可以选择添加照片的相册，如代码清单8.5所示。

代码清单8.5 admin_add_photo.js

```

$(function(){

    var tmpl, // Main template HTML
        tdata = {} // JSON data object that feeds the template

    // Initialize page

    var initPage = function() {

        // Load the HTML template
        $.get("/templates/admin_add_photos.html", function(d){
            tmpl = d;
        });

        // Retrieve the server data and then initialize the page
        $.getJSON("/v1/albums.json", function (d) {
            $.extend(tdata, d.data);
        });

        // When AJAX calls are complete parse the template
        // replacing mustache tags with vars
        $(document).ajaxStop(function () {
            var renderedPage = Mustache.to_html( tmpl, tdata );
            $("body").html( renderedPage );
        });
    }();

});

```

最后，需要花些时间看一下Github代码库中第8章

create_album/文件夹下的HTML页面代码，看下表单和上传文件到服务器的代码（admin_add_photo.html）。该文件的最大亮点就是使用了XmlHttpRequest对象的FormData扩展来实现Ajax文件上传功能，如下所示：

```
$("input#submit_button").click(function (e) {
    var m = new Date();
    var dateString = /* process m -- see GitHub */
    $("input#date").val(dateString);

    var oOutput = document.getElementById("output");
    var oData = new FormData(document.forms.namedItem("add_photo"));
    var oReq = new XMLHttpRequest();
    var url = "/v1/albums/" + $("#albumid").val() + "/photos.json";
    oReq.open("PUT", url, true);
    oReq.onload = function(oEvent) {
        if (oReq.status == 200) {
            oOutput.innerHTML = "\nUploaded! Continue adding or <a href='/pages/album/\n" +
                + $("#albumid").val() + "'>View Album</a>";
        } else {
            oOutput.innerHTML = "\nError " + oReq.status + " occurred uploading your file.<br \/\>";
        }
    };
    oReq.send(oData);
});
```

FormData非常强大和神奇，但在低版本的Internet Explorer（IE 10之前）浏览器中不支持。Firefox、Chrome和Safari都已经支持它了。如果想要在旧版本的IE浏览器中支持Ajax文件上传功能，可以尝试其他文件上传方法，如使用Flash或者传统的HTML表单上传。

8.5 应用结构回顾

至此，应用已经变得有些复杂了，需要花些时间回顾下整个应用是如何组织架构的。将所有的静态内容移至static/文件夹下，并将代码都放置到app/文件夹下，因此我们拥有如下基本结构：

static/文件夹包含以下几个子文件夹：

- albums/——包含所有相册及图片文件
- content/——包含样式表（CSS）和渲染页面模板所需的JavaScript加载器文件
- templates/——浏览器渲染页面所需的HTML模板

在app/文件夹下，拥有：

- ./——包含核心的服务器端脚本和package.json文件
- data/——和后端数据存储相关的所有代码种类
- handlers/——包含所有处理客户端请求的代码

从本章起，所有版本的应用都会使用以上所示代码结构。

8.6 小结

现在，我们不仅升级了相册应用的版本，它使用MongoDB存储相册和照片数据；还多了一些有趣的页面，可以用来创建相册和上传照片到服务器。

唯一的缺陷就是所有人都可以访问这些页面，并使用API处理相册和照片。所以，接下来我们需要把目光转向添加用户权限上，以确保用户登录后才能使用应用。

第9章 数据库II：SQL (MySQL)

虽然NoSQL数据库在迅速普及，但我们仍然有许多理由继续使用关系型数据库，它们一如既往地受欢迎，特别是最常用的两个开源数据库：MySQL和PostgreSQL。好消息是，在Web应用中，Node.js的异步特性能够与这些数据库完全吻合，同时还有一个优秀的npm模块能够支持它们。

在本章中，我们会介绍如何在Node中使用MySQL和npm中的mysql模块。因为在上一章中，我们已经学习了如何将相册和照片数据存储到数据库，现在可以将关注点转向应用中的用户注册，并要求用户在创建任何相册或者添加照片之前必须处于登录状态。

即使你不打算使用诸如MySQL这样传统的关系型数据库，但本章仍然值得一读，因为本章会介绍express的一系列重要特性，还会讨论资源池（resource pooling）——一种控制和限制系统宝贵资源的方法。本章还会升级相册示例，以便相册和照片可以和MySQL一起工作。

9.1 准备工作

在Node.js中使用MySQL需要先做两件事情：确保MySQL已经安装并且安装了mysql npm模块。

9.1.1 安装MySQL

如果还没有在开发机器上安装MySQL，可以访问dev.mysql.com/downloads/mysql并下载合适的社区版服务器。对于Windows和Mac OS X系统，可以直接下载安装包；而对于Linux和其他的Unix系统，则解压.tar.gz文件到合适的位置（通常是/usr/local/mysql）。

如果在Windows和Mac OS X上运行安装包，请注意安装的细节，而对于二进制发行包，则需要阅读INSTALL-BINARY文本文件并按照其中的步骤来完整安装并运行MySQL。当完成之后，应该能够通过命令提示符或者终端启动并运行mysql命令：

```
Kimidori:Learning Node marcw$ /usr/local/mysql/bin/mysql -u root  
Welcome to the MySQL monitor. Commands end with ; or \g.  
Your MySQL connection id is 1  
Server version: 5.5.29 MySQL Community Server (GPL)  
  
mysql>
```

9.1.2 从npm添加mysql模块

要为Node.js安装mysql模块，可以修改package.json并添加以下所示依赖：

```
"dependencies": {  
  "async": "0.1.x",  
  "mysql": "2.x"  
}
```

现在应该能在项目根目录的node_modules/下看到mysql/目录。请注意，2.x系列的mysql模块相较于0.x系列有了显著的性能提

升。另外，Node.js的一大棘手之处就是它还一直处于开发和改进中，任何事情都很可能发生改变。本书使用2.x系列，因为它比之前的版本健壮许多。

9.2 创建数据库模式

使用MySQL工作时，需要为应用创建数据库模式，它保存在 schema.sql文件中。第一部分是创建一个以UTF-8为默认字符集和排序规则的数据库，代码如下所示：

```
DROP DATABASE IF EXISTS PhotoAlbums;

CREATE DATABASE PhotoAlbums
    DEFAULT CHARACTER SET utf8
    DEFAULT COLLATE utf8_general_ci;

USE PhotoAlbums;
```

接着为User表创建表结构，用来存放相册应用的注册用户信息。我们仅仅需要用户的email地址、显示名和密码，并存储一些额外的信息，包括账户的创建时间、上次修改时间以及是否标记为删除状态。User表如下所示：

```
CREATE TABLE Users
(
    user_uuid VARCHAR(50) UNIQUE PRIMARY KEY,
    email_address VARCHAR(150) UNIQUE,

    display_name VARCHAR(100) NOT NULL,
    password VARCHAR(100),

    first_seen_date BIGINT,
    last_modified_date BIGINT,
    deleted BOOL DEFAULT false,

    INDEX(email_address),
    INDEX(user_uuid)
)
ENGINE = InnoDB;
```

在MySQL中运行这些命令（mysql -u user -p secret < schema.sql）来建立开始编写代码所需的合适的数据库和表

结构。如果照着GitHub仓库上的代码编写，我们会发现用于相册和照片的表已经添加完成。

9.3 基本数据库操作

MySQL的大部分工作都是创建到MySQL服务器的连接并执行查询和语句。这能让Web应用更简单地运行并挖掘数据库服务器的能力。

9.3.1 连接数据库

为了连接远程服务器，需要通过mysql模块创建一个连接，然后调用connect函数，如下所示：

```
dbclient = mysql.createConnection({
  host: "localhost",
  user: "root",
  password: "",
  database: "PhotoAlbums"
});

dbclient.connect(function (err, results) { });


```

如果连接失败，则会接收到一个错误。而如果成功，则可以使用客户端对象来执行查询。当完成数据库相关的工作之后，应该通过调用end方法来关闭连接：

```
dbclient.end();
```

9.3.2 添加查询

连接上数据库服务器之后，我们可以向query方法传入需要执行的SQL语句来执行查询：

```
dbclient.query("SELECT * FROM Albums ORDER BY date", function (err, results, fields)
{});
```

如果查询成功，results参数会传给回调函数，它包含了请求的相应数据，另外，还有第三个参数——fields，如果有额外的信息需要指定则可以使用它（如果没有额外的信息，它通常为空）。对于SELECT语句，回调函数的第二个参数是查询出来的所有行的数组。

```
dbclient.query("SELECT * FROM Albums", function (err, rows) {
  for (var i = 0; i < rows.length; i++) {
    console.log(" -> Album: " + rows[i].name
      + " (" + rows[i].date + ")");
  }
});
```

如果使用INSERT、UPDATE或者DELETE，则query方法返回的结果与以下所示结果非常相似：

```
{ fieldCount: 0,
  affectedRows: 0,
  insertId: 0,
  serverStatus: 2,
  warningCount: 0,
  message: '',
  changedRows: 0 }
```

我们可以使用这些信息，通过查看affectedRows属性来确保受影响的行数和预期的一样，或通过insertId属性获取最后插入的行，即autogenerated ID。

Node.js中还有许多MySQL的使用方法。现在就让我们开始深入学习吧！

9.4 添加应用身份验证

在上一章中，我们展示了如何将应用的相册和照片存储转换成使用数据库存储，但存在一个问题，即任何人都可以匿名访问添加相册和照片页面。因此，需要对应用做如下改造：

- 为应用提供新用户注册功能。
- 添加相册或者照片时要求用户已经登录和注册。

9.4.1 更新API以支持用户

为了支持新的用户子系统，需要为这些API添加两个新的路由：

```
app.put('/v1/users.json', user_hdlr.register);
app.post('/service/login', user_hdlr.login);
```

第一个路由是个CRUD方法（请参见7.3.1节），通过这个API创建用户。第二个路由用来支持Web浏览器版本应用的登录和验证。我会在后续章节展示这两个方法的实现。

9.4.2 检测核心用户数据操作

要获得数据库的连接，需要为应用程序创建一个data/文件夹，并将一个叫db.js的文件保存到文件夹中（另外还需要一个包含简单错误处理的帮助文件backend_helpers.js）。该文件如代码清单9.1所示，文件提供一个函数，通过使用local.config.js文件提供的连接信息，能够高效地连接MySQL数据库，正如在第8章看到的一样：

代码清单9.1 db.js

```
var mysql = require('mysql'),
local = require("../local.config.js");

exports.db = function (callback) {

  conn_props = local.config.db_config;
  var c = mysql.createConnection({
    host: conn_props.host,
    user: conn_props.user,
    password: conn_props.password,
    database: conn_props.database
  });
  callback(null, c);
};
```

当连接使用结束之后，我们有义务通过调用end方法来关闭连接，以释放它所消耗的资源。有了这个文件，可以开始在data/user.js文件中为新API编写后端代码了。

创建用户

在数据库中注册一个新用户所需的代码如下所示：

```
exports.register = function (email, display_name, password, callback) {
  var dbc;
  var userid;
  async.waterfall([
    // validate ze params
    function (cb) { // 1.
      if (!email || email.indexOf('@') == -1)
        cb(backend.missing_data("email"));
      else if (!display_name)
        cb(backend.missing_data("display_name"));
      else if (!password)
        cb(backend.missing_data("password"));
      else
        cb(null);
    },
    function (cb) {
      db.db(cb);
    },
    function (dbcclient, cb) {
      dbc = dbcclient;
      bcrypt.hash(password, 10, cb); // 3.
    },
    function (hash, cb) {
      userid = uuid(); // 4.
      dbc.query( // 5a.
        "INSERT INTO Users VALUES (?, ?, ?, ?, ?, UNIX_TIMESTAMP(), NULL, 0)",
        [email, display_name, password, hash, null, null]
      );
    }
  ], function (err) {
    if (err)
      cb(err);
    else
      cb(null, userid);
  });
};
```

```

        [ userid, email, display_name, hash ],
        cb);
    },

    function (results, fields, cb) { // 5b.
        exports.user_by_uuid(userid, cb);
    }
],
function (err, user_data) {
    if (dbc) dbc.end();
    if (err) {
        if (err.code
            && (err.code == 'ER_DUP_KEYNAME'
                || err.code == 'ER_EXISTS'
                || err.code == 'ER_DUP_ENTRY'))
            callback(backhelp.user_already_registered());
        else
            callback (err);
    } else {
        callback(null, user_data);
    }
}
);
}
);

```

代码做了以下几件事：

- 1) 验证传入的参数，尤其要确保email地址是合法的。如果发送一个验证链接到该email地址以激活账户，则更严格。
- 2) 获取数据库连接。
- 3) 使用bcrypt模块哈希密码。bcrypt是一个生成密码的方法，这使得暴力破解密码变得极其困难。
- 4) 为用户生成一个UUID，稍后会在API中使用它来识别用户。这些ID比简单的整数的用户ID好得多，因为它们难以猜测，也没有明显的顺序。
- 5) 在数据库中执行查询来注册用户，并让数据库返回刚刚创建的用户给调用者。

上面代码中使用了两个新模块：bcrypt（用来密码加密）和node-uuid（生成GUID，使用它来识别用户）。因此需要更新package.json文件的依赖如下所示：

```
    "dependencies": {
        "express": "3.x",
        "async": "0.1.x",
        "mysql": "2.x",
        "bcrypt": "0.x",
        "node-uuid": "1.x"
    }
}
```

注意，这里使用了BIGINT而不是通常的DATETIME字段来存储账户创建日期。因为JavaScript在任何地方都使用时间戳来表示日期和时间，在数据库里同样使用BIGINT类型，在存储和操作这些字段时会更加简单。幸运的是，MySQL新提供了一个函数来协助处理这些日期。

[获取用户（通过Email地址或UUID）](#)

现在，已经能够保存用户数据到数据库，我们可以开始编写提取用户的函数。首先，编写一个通用函数，根据数据库中的特殊字段来查询用户：

```
function user_by_field (field, value, callback) {
    var dbc;
    async.waterfall([
        function (cb) {
            db.db(cb);
        },
        function (dbclient, cb) {
            dbc = dbclient;
            dbc.query(
                "SELECT * FROM Users WHERE " + field
                + " = ? AND deleted = false",
                [ value ],
                cb);
        },
        function (rows, fields, cb) {
            if (!rows || rows.length == 0)
                cb(backhelp.no_such_user());
            else
                cb(null, rows[0]);
        }
    ],
    function (err, user_data) {
        if (dbc) dbc.end();
        callback(err, user_data);
    });
}
```

现在，编写需要输出的函数来获取用户：

```
exports.user_by_uuid = function (uuid, callback) {
    if (!uuid)
        cb(backend.missing_data("uuid"));
    else
        user_by_field("user_uuid", uuid, callback);
};

exports.user_by_email = function (email, callback) {
    if (!email)
        cb(backend.missing_data("email"));
    else
        user_by_field("email_address", email, callback);
};
```

这些就是用户管理中数据部分的全部代码。其他事项都在前端的处理程序上进行处理。

9.4.3 更新express应用

express服务器用来记录用户是否登录的主要方法是session cookie。浏览器每次请求时都会发送cookie，因此可以使用session数据（请参见7.4.5节）来记住用户的登录状态。要设置该功能，需要添加下面的代码到server.js顶部：

```
app.use(express.cookieParser("secret text"));
app.use(express.cookieSession({
    secret: "FLUFFY BUNNIES",
    maxAge: 86400000,
    store: new express.session.MemoryStore()
}));
```

如果用户已经登录，则设置req.session.logged_in=true并将req.session.logged_in_email_address设置为一个合法值。如果用户没有登录，则这些值都设置为undefined。

为了确保用户在使用管理页面之前已经登录，可以引入一个之前没有见过的express功能：能将自己的中间件函数注入到路由声明中。

通常情况下，可以为某个URL指定路由函数，如下所示：

```
app.get(URL, HANDLER_FUNCTION);
```

不仅如此，在URL和处理函数之间可以插入任何数量的中间件函数。这些中间件会按照顺序依次被调用。它们提供三个参数：req、res和next。如果它们想处理请求，则处理请求并发送一个响应（并调用res.end）。如果它们不想处理，则只需简单地调用next()函数。

因此，要求访问管理页面的用户需要处于登录状态，必须修改对应页面的URL路由，如下所示：

```
app.get('/pages/:page_name', requirePageLogin, page_hdlr.generate);
app.get('/pages/:page_name/:sub_page', requirePageLogin, page_hdlr.generate);
```

requirePageLogin函数检查用户请求的页面是否需要认证。如果需要，则检查用户是否已登录。如果已经登录，则允许继续并调用next()；如果还未登录，则拦截URL并重定向到登录页面：

```
function requirePageLogin(req, res, next) {
  if (req.params && req.params.page_name == 'admin') {
    if (req.session && req.session.logged_in) {
      next();
    } else {
      res.redirect("/pages/login");
    }
  } else
    next();
}
```

9.4.4 创建用户处理程序

为支持账户管理，可以在/users.js文件中创建一个新的用户处理程序。与前文中对相册和照片的做法一样，创建一个新的User类以帮助封装用户并实现一个response_obj方法来过滤不需要返回的数据：

```
function User (user_data) {
  this.uuid = user_data["user_uuid"];
  this.email_address = user_data["email_address"];
  this.display_name = user_data["display_name"];
  this.password = user_data["password"];
  this.first_seen_date = user_data["first_seen_date"];
  this.last_modified_date = user_data["last_modified_date"];
  this.deleted = user_data["deleted"];
}

User.prototype.uuid = null;
User.prototype.email_address = null;
User.prototype.display_name = null;
User.prototype.password = null;
User.prototype.first_seen_date = null;
User.prototype.last_modified_date = null;
User.prototype.deleted = false;
User.prototype.check_password = function (pw, callback) {
  bcrypt.compare(pw, this.password, callback);
};

User.prototype.response_obj = function () {
  return {
    uuid: this.uuid,
    email_address: this.email_address,
    display_name: this.display_name,
    first_seen_date: this.first_seen_date,
    last_modified_date: this.last_modified_date
  };
};
```

创建新用户

接下来，实现创建用户的函数。基本流程如下：

- 1) 检查传入的数据，确保数据合法。
- 2) 在后端创建一个用户账户并返回原始数据。
- 3) 标识用户已经登录。
- 4) 返回最新创建的用户对象给调用者。

该函数的代码如下所示：

```
exports.register = function (req, res) {
    async.waterfall([
        function (cb) { // 1.
            var em = req.body.email_address;

            if (!em || em.indexOf "@" == -1)
                cb(helpers.invalid_email_address());
            else if (!req.body.display_name)
                cb(helpers.missing_data("display_name"));
            else if (!req.body.password)
                cb(helpers.missing_data("password"));
            else
                cb(null);
        },
        function (cb) { // 2.
            user_data.register(
                req.body.email_address,
                req.body.display_name,
                req.body.password,
                cb);
        },
        function (user_data, cb) { // 3.
            req.session.logged_in = true;
            req.session.logged_in_email_address = req.body.email_address;
            req.session.logged_in_date = new Date();
            cb(null, user_data);
        }
    ],
    function (err, user_data) { // 4.
        if (err) {
            helpers.send_failure(res, err);
        } else {
            var u = new User(user_data);
            helpers.send_success(res, {user: u.response_obj()});
        }
    });
};
```

用户登录

用户登录系统，需要进行以下操作：

- 1) 根据提供的email地址匹配用户对象（如果email地址不存在则抛出一个错误）。
- 2) 使用bcrypt模块比较密码。

3) 如果密码匹配则设置sesion变量来标识用户已经登录。否则 , 标识验证失败。

整个流程如下 :

```
exports.login = function (req, res) {
  async.waterfall([
    function (cb) {
      var em = req.body.email_address;
      if (!em || em.indexOf('@') == -1)
        cb(helpers.invalid_email_address());
      else if (req.session && req.session.logged_in_email_address
              == em.toLowerCase())
        cb(helpers.error("already logged in", ""));
      else if (!req.body.password)
        cb(helpers.missing_data("password"));
      else
        user_data.user_by_email(req.body.email_address, cb);           // 1.
    },
    function (user_data, cb) {
      var u = new User(user_data);
      u.check_password(req.body.password, cb);                         // 2.
    },
    function (auth_ok, cb) {
      if (!auth_ok) {
        cb(helpers.auth_failed());
        return;
      }
      req.session.logged_in_email_address = req.body.email_address;   // 3.
      req.session.logged_in_date = new Date();
      cb(null);
    }
  ],
  function (err, results) {
    if (!err || err.message == "already_logged_in") {
      helpers.send_success(res, { logged_in: true });
    } else {
      helpers.send_failure(res, err);
    }
  });
};
```

测试登录状态

要想测试用户是否已经登录 , 只需查看对应的sesion变量是否已经设置 :

```
exports.logged_in = function (req, res) {
  var li = (req.session && req.session.logged_in_email_address);
  helpers.send_success(res, { logged_in: li });
};
```

注销

最后 , 注销用户只需清除sesion数据 , 这样系统就不再认为用户处于登录状态 :

```
exports.logout = function (req, res) {
    req.session = null;
    helpers.send_success(res, { logged_out: true });
};
```

9.4.5 创建登录和注册页面

对于新的用户子系统，应用中新增两个页面：登录页面和注册页面。两个页面都和之前一样由两个文件组成：一个JavaScript加载器和一个HTML文件。两者的JavaScript加载器也都非常标准：

```
$(function(){
    var tmpl,    // Main template HTML
        tdata = {}; // JSON data object that feeds the template

    // Initialize page
    var initPage = function() {

        // Load the HTML template
        $.get("/templates/login OR register.html", function(d){
            tmpl = d;
        });

        // When AJAX calls are complete parse the template
        // replacing mustache tags with vars
        $(document).ajaxStop(function () {
            var renderedPage = Mustache.to_html( tmpl, tdata );
            $("body").html( renderedPage );
        });
    };
});
```

注册页面的HTML如代码清单9.2所示。除了显示注册表单的HTML，还有一些JavaScript以确保用户已经填入所有的字段，验证两个密码是否一致，然后提交数据到后端服务器。如果登录成功，则重定向到应用首页；否则，会显示错误并让用户再试一次。

代码清单9.2 注册页面的Mustache模板 (register.html)

```

<div style="float: right"><a href="/pages/login">Login</a></div>
<form name="register" id="register">
    <div id="error" class="error"></div>
    <dl>
        <dt>Email address:</dt>
        <dd><input type="text" size="30" id="email_address"
            name="email_address"/></dd>
        <dt>Display Name:</dt>
        <dd><input type="text" size="30" id="display_name" name="display_name"/></dd>
        <dt>Password:</dt>
        <dd><input type="password" size="30" id="password" name="password"/></dd>
        <dt>Password (confirm):</dt>
        <dd><input type="password" size="30" id="password2" name="password2"/></dd>
        <dd><input type="submit" value="Register"/>
    </dl>
</form>

<script type="text/javascript">
$(document).ready(function () {
    if (window.location.href.match(/(fail)/) != null) {
        $("#error").html("Failure creating account.");
    }
});

$("form#register").submit(function (e) {
    if (!$("#input#email_address").val()
        || !$("#input#display_name").val()
        || !$("#input#password").val()
        || !$("#input#password2").val()) {
        $("#error").html("You need to enter an email and password.");
    } else if ($("#input#password2").val() != $("#input#password").val()) {
        $("#error").html("Passwords don't match.");
    } else {
        var info = { email_address: $("#input#email_address").val(),
                    display_name: $("#input#display_name").val(),
                    password: $("#input#password").val() };
        $.ajax({
            type: "PUT",
            url: "/v1/users.json",
            data: JSON.stringify(info),
            contentType: "application/json; charset=utf-8",
            dataType: "json",
            success: function (data) {
                window.location = "/pages/admin/home";
            },
            error: function () {
                var ext = window.location.href.match(/(fail)/) ? "" : "?fail";
                window.location = window.location + ext;
                return false;
            }
        });
        return false;
    }
});
</script>

```

最后，登录页面的代码如代码清单9.3所示。它和注册页面非常相似：显示表单的HTML并包含一些JavaScript代码用来处理数据并提交到服务器。

代码清单9.3 登录页面的Mustache模板 (login.html)

```

<div style='float: right'><a href='/pages/register'>Register</a></div>
<form name="login" id="login">
    <div id="error" class="error"></div>
    <dl>
        <dt>Email address:</dt>
        <dd><input type="text" size="30" id="email_address"
            name="email_address"/></dd>
        <dt>Password:</dt>
        <dd><input type="password" size="30" id="password" name="password"/></dd>
        <dd><input type="submit" value="Login"/>
    </dl>
</form>

<script type="text/javascript">

```

```
$(document).ready(function () {
    if (window.location.href.match(/(fail)/) != null) {
        $("#error").html("Invalid login credentials.");
    }
});
$("form#login").submit(function (e) {
    if (!$("input#email_address").val() || !$("input#password").val()) {
        $("#error").html("You need to enter an email and password.");
    } else {
        var info = { email_address: $("input#email_address").val(),
                    password: $("input#password").val() };
        $.ajax({
            type: "POST",
            url: "/service/login",
            data: JSON.stringify(info),
            contentType: "application/json; charset=utf-8",
            dataType: "json",
            success: function (data) {
                window.location = "/pages/admin/home";
            },
            error: function () {
                var ext = window.location.href.match(/(fail)/) ? "" : "?fail";
                window.location = window.location + ext;
                return false;
            }
        });
    }
    return false;
});
</script>
```

当添加完这些文件 (data/user.js和handlers/user.js、 login.js 和login.html以及register.js和register.html) 之后，我们就拥有了一个完整的Web浏览器端的登录系统。

9.5 资源池

在上一章中，我们了解到Node.js中的MongoDB驱动会管理自己的连接集合，或连接“池”。但mysql模块没有类似的功能，因此需要我们自己处理连接池机制。幸运的是，这一点非常容易做到，不必担心模块没有实现这个功能。

在npm中，有个叫做generic-pool的模块允许池化（pooling）任何东西。要使用它，需要创建一个池，指定要创建的项的上限，并提供一个函数，用于在池中创建对象的新实例。然后，要想从池中获取一个项，我们可以调用acquire函数，这个函数会一直等待，直到有可用的池对象出现。当任务完成之后，调用release函数将对象返回给池。

接下来，我们一起看看具体是如何实现的。

9.5.1 入门

添加generic-pool到package.json文件的依赖中：

```
"dependencies": {  
    "express": "3.x",  
    "async": "0.1.x",  
    "generic-pool": "2.x",  
    "mysql": "2.x",  
    "bcrypt": "0.x",  
    "node-uuid": "1.x"  
}
```

要在文件中引入这个模块，需要require它，如下所示：

```
var pool = require('generic-pool'),
```

如果想创建一个池，需要创建含有配置信息的池类的新实例，即指定需要的池中项的数量，以及创建新对象的函数：

```
conn_props = local.config.db_config;
mysql_pool = pool.Pool({
    name      : 'mysql',
    create    : function (callback) {
        var c = mysql.createConnection({
            host:      conn_props.host,
            user:      conn_props.user,
            password: conn_props.password,
            database: conn_props.database
        });
        callback(null, c);
    },
    destroy     : function (client) { client.end(); },
    max         : conn_props.pooled_connections,
    idleTimeoutMillis : conn_props.idle_timeout_millis,
    log         : false
});
});
```

9.5.2 处理连接

现在，在代码中，无论何时想要一个MySQL服务器的连接，都无需创建一个新连接，而是让池管理这一切。acquire函数会阻塞住，直到有可用的连接出现。db.js中的db函数变成如下所示：

```
exports.db = function (callback) {
    mysql_pool.acquire(callback);
};
```

当连接使用完毕，只需要将连接释放回连接池，代码如下所示：

```
conn.release();
```

而其他等待连接的人就可以立刻获得该连接。

9.6 验证API

现在，你可能会闲下心来并自鸣得意，因为应用会在检测到登录之后，才允许创建相册或者添加照片到相册中，但仍然有一个严重安全问题：API本身并不足够安全。仍然可以运行以下代码而不需要任何验证：

```
curl -X PUT -H "Content-Type: application/json" \
      -d '{ "name": "hk2012", "title" : "moo", "description": "cow", "date":
"2012-12-28" }' \
      http://localhost:8080/v1/albums.json
```

如果系统只有某些部分是安全的，而其他部分则完全没有受到保护，那么这样的系统不会令人印象深刻，我们需要立即解决这个问题。

可以通过以下方式创建系统：

- Web浏览器的前端部分继续使用登录和注册页面，使用登录服务来管理对服务器的访问。
- API用户在发送请求时会使用HTTP基本身份验证传入用户名和密码，从而验证受限制API的访问权。

HTTP基本身份验证：安全吗？

正如本章中解释的一样，HTTP基本身份验证是一个相对简单的安全形式：每次给服务器发送请求，客户端都会将用户名和经过base64编码的密码传入请求头，每次处理前，都经过服务器的认证。因为base64编码并不是一种加密形式，所以它相当于在互联网上通过纯文本传递用户名和密码。因此，我们会有这样的疑问：本质上，它是不安全的吗？

答案是，很大程度上它是安全的。大多数需要安全性的网站都会使用SSL/HTTPS来加密和传输数据。在客户端和服务器之间进行加密数据传输的方法是相当安全的，而在服务器上绝对也会做一些处理。

部分人会认为Web浏览器会记住基本身份验证的用户名和密码，而这是不安全的。这种看法是合理的，这也是为什么只针对API服务器使用基本认证。对于基于浏览器的应用，应该坚持使用常规的登录页面和session cookie。

因此，虽然HTTP基本身份验证可能并不安全，但加上一点点的准备和预防措施之后，我们的REST API服务器就能成为可信赖的产品级别的安全系统。

客户端不能记住来自API用户的sesion，因为它们没有传入cookie，因此，没有登录状态会被存储下来。这些用户每次独立的请求都需要验证。要达到这个目的，它们传入以下格式的头部信息：

```
Authorization: Basic username:password base64 encoded
```

例如，要发送含有密码kittycatonkeyboard的marcwan@example.org的认证，需要发送

```
Authorization: Basic bWFyY3dhbkBleGFtcGx1Lm9yZzpraXR0eWNhdG9ua2V5Ym9hcmQ=
```

幸运的是，我们不必自己完成这些头信息，因为有程序（例如curl）会为我们完成这样的工作。要通过curl请求传递HTTP基本身份验证信息，可以添加下面的代码到请求中：

```
curl -u username:password ...
```

因此，我们可以对系统稍作修改，给这些API添加些安全措施：修改这两个创建相册和添加照片的PUT API。要做到这一点，需要编写一段叫做requireAPILogin的中间件：

```
app.put('/v1/albums.json', requireAPILogin, album_hdlr.create_album);
app.put('/v1/albums/:album_name/photos.json',
       requireAPILogin, album_hdlr.add_photo_to_album);
```

requireAPILogin函数的代码如下所示：

```

function requireAPILogin(req, res, next) {
  if (req.session && req.session.logged_in) { // 1.
    next();
    return;
  }
  var rha = req.headers.authorization; // 2.
  if (rha && rha.search('Basic ') === 0) {
    var creds = new Buffer(rha.split(' ')[1], 'base64').toString();
    var parts = creds.split(":");
    user_hdrl.authenticate_API( // 3.
      parts[0],
      parts[1],
      function (err, resp) {
        if (!err && resp) {
          next(); // 4.
        } else
          need_auth(req, res);
      }
    );
  } else
    need_auth(req, res);
}

```

代码的工作原理如下：

- 1) 为了保持与Web应用的兼容性，首先会检查用户是否已经在浏览器中使用session登录。如果是，表明用户很清楚需要使用什么API。因为命令行或简单客户端的纯API调用不会传入cookie（但浏览器里的Ajax调用API则会），因此没有session，必须提供额外的认证信息。
- 2) 查看请求头Authentication:。
- 3) 解码并分离头部信息中的用户名和密码。
- 4) 在用户处理程序中把这些信息传递给authenticate_API函数。

用户处理程序中的authenticate_API函数如下所示——通过给定的email地址获取用户，使用bcrypt npm模块验证密码，最后返回成功或失败：

```
exports.authenticate_API = function (un, pw, callback) {
  async.waterfall([
    function (cb) {
      user_data.user_by_email(un, cb);
    },
    function (user_data, cb) {
      var u = new User(user_data);
      u.check_password(pw, cb);
    }
  ],
  function (err, results) {
    if (!err) {
      callback(null, un);
    } else {
      callback(new Error("bogus credentials"));
    }
  });
};
```

最后，如果确定用户不符合对应请求的验证，则需要做一些工作，告诉用户不允许使用请求的资源：

```
function need_auth(req, res) {
  res.header('WWW-Authenticate',
             'Basic realm="Authorization required"');
  if (req.headers.authorization) {
    // no more than 1 failure / 5s
    setTimeout(function () {
      res.send('Authentication required\n', 401);
    }, 3000);
  } else {
    res.send('Authentication required\n', 401);
  }
}
```

该函数完成了API服务器在HTTP基本身份验证方面的工作。如果验证失败，它返回响应头WWW-Authenticate。为了额外的安全性，如果看到用户发送错误的用户名/密码组合，则可以暂停几秒钟以阻止进一步的外部攻击。这之后，相册应用程序算是有了一个相对安全的Web和API服务器。

而要真正的安全，则应该确保任何请求都要求一个密码，这个密码是通过HTTPS安全连接发送的，因此没有人能够使用数据包嗅探器在线路上查看密码。我们会在第10章详细讨论HTTPS。

9.7 小结

本章介绍了两个新知识点：在Node应用中使用MySQL数据库和添加用户验证到Web应用以及API服务器。同时，我也在GitHub代码树上更新了含有用户验证子系统的MongoDB版本应用，这样我们就知道它具体是如何实现的。通过创建两个平行的验证系统，我们为浏览器应用创造了良好的用户体验，而让API用户仍然得到最为简单而强大的JSON API。

在结束本书第三部分时，我们学习到如何添加功能强大的新技术到相册应用，包括express和数据库，并创建了功能完整（可能只是基本的）且可以构建的项目。在本书的最后一部分，会覆盖本书之前一直忽略的一些细节。首先，我们会在第10章中学习如何部署应用。

第四部分 进阶篇

第10章 部署和开发

第11章 命令行编程

第12章 测试

第10章 部署和开发

在充分掌握构建Node.js应用的能力之后，我们现在可以将注意力转到部署和开发应用等主题上来。在本章中，我们首先了解人们在产品服务器上部署和运行Node应用的各种方式，包括UNIX/Mac和Windows平台。接着会学习如何利用拥有多核处理器的机器，虽然事实上Node.js是一个单线程平台。

接下来我们会关注如何在服务器上添加对虚拟主机的支持，同时添加SSH/HTTPS来确保应用安全。最后，会快速浏览在Windows和UNIX/Mac机器上跨平台开发Node应用时带来的问题。

10.1 部署

目前，要想运行Node.js应用，只需要在命令行中运行如下代码：

```
node script_name.js
```

在开发阶段采用这种方式还不错。然而，要想部署应用到生产服务器中，则需要添加额外的可靠层，当应用崩溃时能够提供帮助。虽然我们力争避免服务器出现bug，但难免会有意外，因此需要服务器尽快从问题中恢复。

接下来让我们看一些解决方案。

端口号

本书中大部分示例应用使用的端口号数字比较大，基本上都是8080。因为某些操作系统（特别是UNIX/Mac OS X系统）小于1024的端口号需要超级用户权限。因此，对于简单的调试和测试，8080端口比一般的80端口更合适。

但是，当我们部署应用时，通常又想要它们运行在80端口上，因为80端口是Web浏览器的默认端口。因此，需要以超级用户身份运行脚本。有两种方式可以完成这件事：

- 1) 以超级用户的身份登录系统并运行程序。
- 2) 使用sudo命令，在实际运行时能够以超级用户权限执行进程。

对于以上两种策略，我一般都使用后者。

10.1.1 级别：基础

最基本的级别就是使用无限循环运行脚本。该循环会在Node进程崩溃时自动重启Node进程。在UNIX和Mac计算机上运行bash，

其脚本如下所示：

```
#!/bin/bash
while true
do
    node script_name.js
done
```

在Windows平台上可以使用batch文件完成类似的工作，如下所示（保存该脚本到run_script.bat文件或其他类似文件中并运行）：

```
: loop
node script_name.js
goto loop
: end
```

这些脚本会确保Node进程在崩溃或者意外终止时立即重启。

更进一步，我们可以使用tee命令将node进程的标准输出写入到日志文件中。通常在命令行中使用| ("pipe") 操作符，将进程的输出pipe到tee命令中，然后tee再重新打印输出到标准输出，并写入到特定的日志文件中，如下所示：

```
node hello_world.js | tee -a /var/server/logs/important.log
```

tee的-a参数表示添加到特定日志文件而不是简单地覆盖这个文件。因此，可以通过运行如下脚步进一步改进部署：

```
bash ./run_script.sh | tee -a /var/server/logs/important.log
```

Window平台可以使用下面的命令：

```
run_script.bat | tee -a /var/server/logs/important.log
```

要获取更多关于tee的信息并真正运用该技术，请参见：

[Windows下的Tee（和其他有用的命令）](#)

Windows下的shell并不强大（虽然PowerShell正朝着正确的方向发展）。为了解决这个问题，人们通常会下载外部工具包，让

shell拥有UNIX平台上（如Linux或Mac OS X）一些非常棒的脚本功能。

对于本章提到的tee编程，有两个工具可供选择：

- <http://getgnuwin32.sourceforge.net/>
- <http://unxutils.sourceforge.net/>

也可以搜索"Windows tee"，查找针对Windows的特定tee工具。事实上，许多本书中描述的类UNIX工具基本上在Windows上都有同样功能的替代品或者版本。

使用screen监控输出

虽然在Windows上总是处于登录状态并很容易回到桌面查看Node服务器的运行状况，但是在UNIX服务器上（例如Linux或Mac OS X）经常需要注销而服务器仍然保持运行。问题在于当这样做时，会失去标准输出（stdout）处理，输出也会丢失，除非将输出写入到一个日志文件中。即使这样，日志文件也经常会错过一些重要信息，这些重要信息主要是Node或者操作系统打印出来的相关问题的内容。

为了解决这个问题，大多数UNIX系统都支持一个叫做screen的神奇小工具，即使已经注销，它也能够运行程序，就像一直拥有一个控制终端一样（也称之为tty）。不同平台下的安装不一样，但Mac OS X已经默认安装，而大多数的Linux平台可以通过类似apt-get screen的操作获取它。

现在，在运行Node服务器之前，可以运行screen并在其中运行Node。要离开screen，可以按下键序列Ctrl+A Ctrl+D。要重新进入screen，可以运行命令screen -r（恢复）。如果有多个screen session在运行，每个screen都有一个名字，想要恢复则需要输入对应session的名字。

我们可以使用screen运行每个独立的Node服务器，这是每隔一

段时间回来并查看运行情况的最佳方式。

10.1.2 级别 : Ninja

前面的脚本对运行中的应用非常有用，可以在不会经常崩溃的低流量网站上使用它们。但是，它们存在两个严重的缺陷：

1) 如果一个应用进入持续崩溃的状态，即使重启了，这些脚本也会盲目地不断地重启它，导致变成“丢失的进程”。服务器永远不可能正常启动，一直处于“假死”状态。

2) 虽然这些脚本可以保持进程启动和运行，但当应用遇到麻烦时很难通知我们，特别是它们使用太多内存的时候。同时，还存在Node进程消耗过多内存或者遇到其他系统资源问题的情况。如果能够检测出这些情况并终止进程，随后再次启动，那就非常棒了。

因此，在下一节，我们会看到关于两个部署的高级选项，在这一点上，UNIX和Windows的区别开始变大。

Linux/Mac

以前版本的脚本（以伪代码展示）非常有效：

```
while 1
    run node
end
```

现在可以升级这些脚本来做更多的事情，脚本如下所示：

```
PID = run node
while 1
    sleep a few seconds
```

```
if PID not running
    restart = yes
else
    check node memory usage
    if using too much memory
        restart = yes

    if restart and NOT too many restarts
        PID = restart node
end
```

要使上面的代码工作，有两个关键任务需要执行：

- 1) 获得最新启动的Node服务器的进程ID (pid) 。
- 2) 弄清楚Node进程使用了多少内存。

关于第一点，可使用pgrep命令，Mac和大部分UNIX平台都支持。当带有-n参数时，它会返回给定进程名的最新实例的进程ID。在计算机运行多个Node服务器的情况下，也可以使用-f参数来告诉它要匹配的脚本的名字。因此，要获得最新创建的运行script_name.js的Node进程，可以使用：

```
pgrep -n -f "node script_name.js"
```

现在，要获得进程的内存使用量，需要用到两个命令。首先是ps命令，当提供wux（在某些平台是wup，请检查相关文档）参数时，会显示进程的虚拟内存消耗量，同时显示当前驻留内存的大小（这是我们想要的）。ps wux \$pid的输出如下所示：

```
USER      PID %CPU %MEM      VSZ      RSS      TT STAT STARTED      TIME COMMAND
marcw  1974   6.6  3.4  4507980  571888  ?? R      12:15AM  52:02.42 command name
```

我们需要这个输出的第二行第六列（RSS）。要得到该数据，首先需要运行awk命令获得第二行，然后再次运行获得第六列，如下所示：

```
ps wux $PID | awk 'NR>1' | awk '{print $6}'
```

获取当前进程的内存消耗量后，就可以判断进程是否消耗太多内

存，以及有没有必要终止并重启该进程。

在这里我不会打印shell脚本的所有代码，你可以查看GitHub仓库第10章中的代码，叫做node_ninja_runner.sh。可以使用如下方式运行：

```
node_ninja_runner.sh server.js [extra params]
```

当然，如果要监听例如80或443端口号，必须确保已经提升为超级用户权限。

Windows

在Windows上，可靠地运行并监控应用的最佳方式是使用Windows服务。Node.js本身并没有设置成Windows服务，但幸运的是，可以使用一些技巧使它的行为看起来像是一个Windows服务。

通过使用一个叫做nssm.exe (Non-Sucking Service Manager) 的小程序和一个叫做winser的npm模块，可以将Node的Web应用作为服务进行安装并通过Windows管理控制台 (Windows Management Console) 管理它们。要完成这些设置，需要做两件事情：

1) 在package.json中为Web应用安装新的自定义操作。

2) 下载和安装winser，并让其工作。

对于第一步，仅需添加如下配置到package.json文件：

```
{
  "name": "Express-Demo",
  "description": "Demonstrates Using Express and Connect",
  "version": "0.0.1",
  "private": true,
  "dependencies": {
    "express": "3.x"
  },
  "scripts": {
    "start": "node 01_express_basic.js"
  }
}
```

对于第二步，只需如下进行：

```
C:\Users\Mark\LearningNode\Chapter10> npm install winser
```

(或者添加winser到依赖中)然后，当准备好将应用安装成一个服务时，我们可以前往包含package.json文件的项目文件夹下并运行

```
C:\Users\Mark\LearningNode\Chapter10> node_modules\.bin\winser -i  
The program Express-Demo was installed as a service.
```

要卸载服务，只需运行winser-r，如下所示：

```
C:\Users\Mark\LearningNode\Chapter10> node_modules\.bin\winser -r  
The service for Express-Demo was removed.
```

现在可以前往Windows管理控制台，启动、停止、暂停，或者根据需求管理该服务！

10.2 多处理器部署：使用代理

之前提到多次，Node.js作为一个单线程的进程其运行会非常高效：脚本的所有代码都在同一个线程里执行，并使用异步回调来提高CPU效率。

那么，你一定会问：那在多CPU的系统中会怎么处理呢？如何获取服务器的最大能力？许多现代服务器都是强大的8-16核机器，可以的话我们都想使用。

幸好，这个问题的答案相当简单：即在每个想利用的内核上都运行node进程（见图10.1）。我们可以选择众多策略中的某一种将请求路由到各个不同的node进程中，就像匹配需求一样。

接下来面临的问题是，系统不是只有一个，而是n个Node进程在运行，而这些进程必须监听不同的端口（不可能要求多个用户监听同一个端口号）。那如何才能将来自mydomain:80的流量转入这些服务器呢？

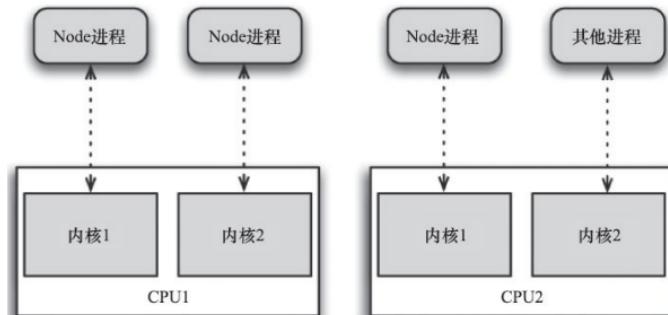


图10.1 为相同的应用运行多个node进程

我们可以通过实现一个简单的循环负载均衡器来解决这个问题，它提供一个运行特定Web应用的node服务器列表。接下来，每次将传入到该域的请求重定向到其中一个服务器中。当然，也可以使用更多高级策略，比如监听加载、可用性以及响应性。

要实现负载均衡器，首先要收集一个运行中的服务器列表。假设服务器有四个内核，分配其中三个给这个应用（保留第四个运行其他的系统服务）。在8081、8082、8083端口上运行这些应用服务器。那么，服务器列表就是：

```
{ "servers": [ {
    "host": "localhost",
    "port": "8081"
}, {
    "host": "localhost",
    "port": "8082"
}, {
    "host": "localhost",
    "port": "8083"
} ] }
```

这些简单服务器的代码展示在代码清单10.1中。这是本书中最简单的Web服务器，只有一些简单的代码用来从命令行获取需要监听的端口号（可以在第11章深入了解命令行参数）。

代码清单10.1 简单的HTTP服务器

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

if (process.argv.length != 3) {
  console.log("Need a port number");
  process.exit(-1);
}

var s = http.createServer(function (req, res) {
  res.end("I listened on port " + process.argv[2]);
});

s.listen(process.argv[2]);
```

我们可以在UNIX/Mac平台通过输入以下命令启动三个服务器：

```
$ node simple.js 8081 &
$ node simple.js 8082 &
$ node simple.js 8083 &
```

对于Windows平台，可以简单地在三个不同的命令提示符中运行下面的三个命令来启动服务器：

```
node simple.js 8081
node simple.js 8082
node simple.js 8083
```

现在可以使用npm模块http-proxy来构建自己的代理服务器。

package.json如下所示：

```
{
  "name": "Round-Robin-Demo",
  "description": "A little proxy server to round-robin requests",
  "version": "0.0.1",
  "private": true,
  "dependencies": {
    "http-proxy": "0.8.x"
  }
}
```

代理服务器的代码相当简单，如代码清单10.2所示。它维护了一个可用的服务器数组，然后遍历这些可用的服务器，将每个传入到该服务的请求都转发到服务器中。

代码清单10.2 循环代理负载均衡器 (roundrobin.js)

```
var httpProxy = require('http-proxy'),
    fs = require('fs');

var servers = JSON.parse(fs.readFileSync('server_list.json')).servers;

var s = httpProxy.createServer(function (req, res, proxy) {
    var target = servers.shift();           // 1. Remove first server
    proxy.proxyRequest(req, res, target);   // 2. Re-route to that server
    servers.push(target);                  // 3. Add back to end of list
});

s.listen(8080);
```

为了让所有代码生效，需要运行node roundrobin.js，接下来就可以开始查询，如下所示：

```
curl -X GET http://localhost:8080
```

如果多次运行该命令，我们可以看到输出结果为当前服务器监听的实际端口号。

多服务器和会话

在不同的CPU和服务器上运行多个服务器是分散负载的好办法，这给我们提供了一个提高可扩展性的低廉途径。（服务器超负荷了？那就再加个服务器！）但这会产生并发症，即需要在有效使用这些服务器前能够定位到它们：目前session数据通过本地MemoryStore对象存储在每个进程中（请参见7.4.5节）。这会导致一个严重的问题，如图10.2所示：

因为每个Node服务器都在自己的内存存储中储存本身的session记录，当两个来自相同客户端的请求分配到两个不同的node进程时，它们对来自客户端的session数据的当前状态会产生疑惑并感到迷茫。我们有两种办法来解决这个问题：

1) 实现一个更高级的路由，可以记住来自特定客户端的请求发送到哪一台Node服务器，并确保对于同一个客户端的所有请求持续地以相同方式进行路由。

2) 将session数据使用的内存存储放到存储池中，这样所有Node进程都可以访问它。

我倾向于选择第二种解决方案，因为它是一个容易实现且并不复杂的解决方案。

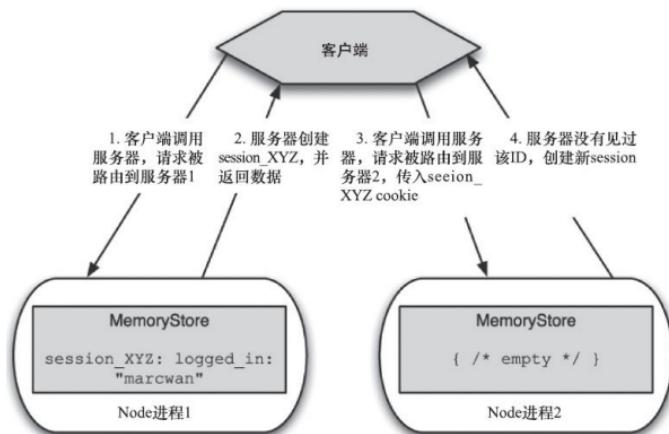


图10.2 多服务器实例和session数据

要实现该解决方案，我们首先需要选择一个内存存储池。最佳候选方案是memcached和Redis，两者都是基于内存的键/值存储（memory-based key/value stores），可以在不同计算机之间传递数据。本章使用的是memcached，因为它完全满足我们的需要而且非常轻量。设置基本上如图10.3所示。

[在Windows上安装](#)

对于Windows用户，可以安装memcached的二进制安装包，在互联网上可以搜索到很多安装包。我们也不需要最新或最强大的版本，1.2.x或之后的版本就可以。为了将其安装成服务，需要运行下面的脚本：

```
c:\memcached\memcached -d install
```

然后可以运行memcached服务，输入如下代码：

```
net start memcached
```

最后，我们可以编辑

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\memcached Server并修改ImagePath为：

```
"C:/memcached/memcached.exe" -d runservice -m 25
```

设置memcached可用的内存量为25MB，通常这对开发已经足够了（当然，可以设置成任何想要的值）。它的监听端口为11211。

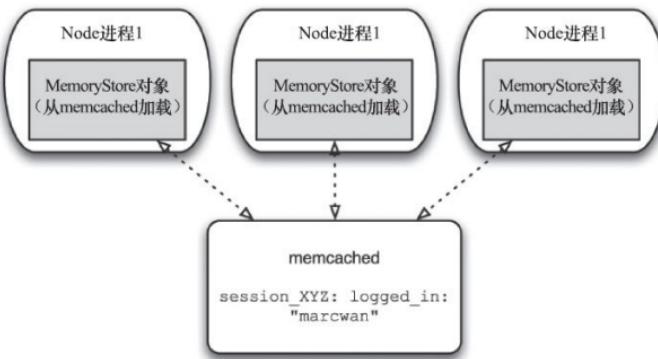


图10.3 使用基于内存的键/值存储来实现session

在Mac和类UNIX系统上安装

如果我们的系统有某种包管理器，则通常可以像下面一样使用（这可能需要sudo）：

```
apt-get install memcached
```

这样就可以了。如果我们想要通过源代码构建，首先需要从 <http://libevent.org> 上获取 libevent。下载、构建以及安装（以超级用户身份）这个库。它应该安装在 /usr/local/lib 目录下。

接下来，访问 memcached.org 并下载最新的源代码包 (.tar.gz 文件) 并再次以超级用户的身份配置、构建和安装 memcached。

安装后要运行 memcached，可以运行如下命令：

```
/usr/local/bin/memcached -m 100 -p 11211 -u daemon
```

该命令告诉服务使用 100MB 内存运行并监听 11211 端口，以 daemon 身份运行。通常以 root 身份运行会被拒绝。

在 express 中使用 memcached

现在我们已经安装并运行 memcached（假设是 localhost 的 11211 端口），那接下来需要为它配一个 MemoryStore 对象，这样 session 数据才可以使用 memcached。幸运的是，Node.js 社区相当活跃，已经有个叫做 connect-memcached 的 npm 模块。因此，我们只需添加下面的配置到 package.json 文件依赖中：

```
"connect-memcached": "0.0.x"
```

之后，可以修改创建 session 的代码，如下所示：

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

// pass the express obj so connect-memcached can inherit from MemoryStore
var MemcachedStore = require('connect-memcached')(express);
var mcds = new MemcachedStore({ hosts: "localhost:11211" });

var app = express()
  .use(express.logger('dev'))
  .use(express.cookieParser())
  .use(express.session({ secret: "cat on keyboard",
    cookie: { maxAge: 1800000 },
    store: mcds}))
  .use(function(req, res){
    var x = req.session.last_access;
    req.session.last_access = new Date();
    res.end("You last asked for this page at: " + x);
  })
  .listen(8080);
```

现在，所有的Node服务器的session数据都配置成使用同一个MemoryStore，无论请求是哪个服务器在处理，都可以获取到正确的信息。

10.3 虚拟主机

多年来，在同一个服务器上运行多个网站对于Web应用平台都是一个主要的需求。幸运的是，当通过express构建应用时，Node.js在这方面为我们提供了两种相当可行的解决方案。

通过向HTTP请求中添加"Host:"头来实现虚拟主机，这是HTTP/1.1的主要特性之一（见图10.4）。

10.3.1 内置支持

express直接内置了运行多虚拟主机的功能。要让它工作，需要为每一个支持的主机创建一个express服务器，然后为虚拟服务器提供一个“主”服务器，用来将请求转发到合适的虚拟服务器。最后，使用vhost连接中间件组件将所有的服务器都组合起来，如代码清单10.3所示：

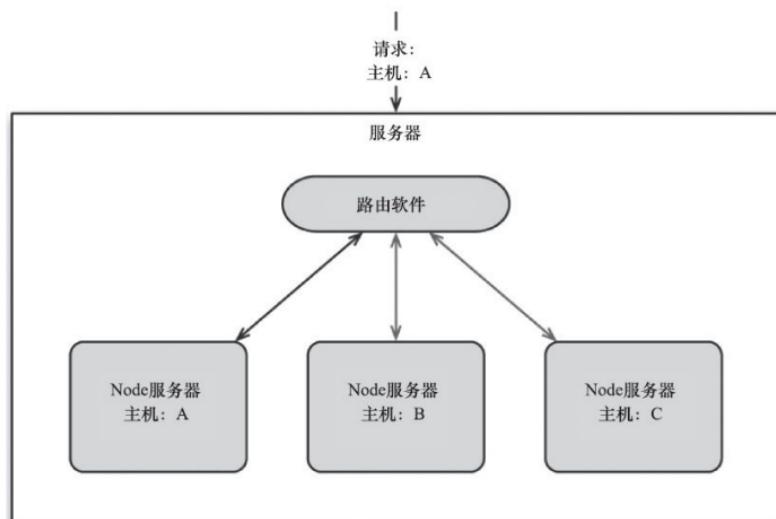


图10.4 虚拟主机的基本概念

代码清单10.3 express中的虚拟主机 (vhost_server.js)

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

var one = express();
one.get('/', function(req, res){
    res.send("This is app one!")
});

// App two
var two = express();
two.get('/', function(req, res){
    res.send("This is app two!")
});

// App three
var three = express();
three.get("/", function(req, res){
    res.send("This is app three!")
});

// controlling app
var master_app = express();

master_app.use(express.logger('dev'));
master_app.use(express.vhost('app1', one))

master_app.use(express.vhost('app2', two));
master_app.use(express.vhost('app3', three));

master_app.listen(8080);
```

测试虚拟主机

测试虚拟机是个不小的挑战，因为所有服务器都运行在localhost上，而我们需要获取服务器并识别出具体请求的服务器名。

如果通过命令行使用curl来测试，只需添加一个头信息到请求中来指定我们请求的是哪个主机，如下所示：

```
curl -X GET -H "Host: hostname1.com" http://localhost:8080
```

然而，如果想在浏览器中进行测试，则需要修改计算机的DNS条目。最常见的方法就是编辑/etc/hosts (UNIX/Mac机器) 或者 C:\Windows\System32\drivers\etc\hosts，这需要高级权限才能完成。在UNIX/Mac上，使用sudo打开编辑器；而在Windows上，只需简单地使用Notepad，但要确保使用管理员权限打开。

然后我们就可以使用如下格式在文件中添加条目了：

127.0.0.1 *hostname*

这里，127.0.0.1是本地计算机的IPv4地址，接下来，只需把想要的名称映射到该地址即可。我们可以添加任意多的主机映射（例如，app1、app2、app3等）。

对于这两种方式，无论是命令行或者是浏览器，用来测试虚拟主机都没有问题。

10.3.2 代理服务器支持

虽然express内置支持虚拟主机，但一个进程运行多个Web应用程序还是会存在风险。如果有人胡作非为或者出现问题，则所有的站点都会出现问题。

为了解决这种情况，我们来看看另外一种管理虚拟主机的方法，这再次使用到了http-proxy模块。该模块功能非常强大，可以很轻松地将请求路由到不同的虚拟主机，只需要几行代码即可。事实上，这种路由方式是基于站点主机名的，我们可以运行代码清单10.4，如下所示：

代码清单10.4 代理服务器虚拟主机

(proxy_vhosts_server.js)

```
var httpProxy = require('http-proxy');

var options = {
  hostnameOnly: true,
  router: {
    'app1': '127.0.0.1:8081',
    'app2': '127.0.0.1:8082',
    'app3': '127.0.0.1:8083'
  }
}

var proxyServer = httpProxy.createServer(options);
proxyServer.listen(8080);
```

我们为代理服务器提供主机列表，用来寻找主机并将请求路由到这些主机中。要测试该功能，只需简单地运行三个不同的应用服务器，然后运行代理服务器，请求就会自动寻找到目标。

10.4 使用HTTPS/SSL保障项目安全

对于应用中处理敏感数据的那部分，例如用户密码、个人数据、银行账户或付款信息，实际上任何在本质上是私人的东西，我们都应该使用SSL加密的HTTPS传输来保护应用。虽然创建SSL连接和加密传输会带来额外的开销，但是安全上得到的收益却更重要。

要为应用添加SSL/HTTPS支持，首先需要生成一些测试证书，并为应用程序添加加密传输的支持。对于后者，我们也有两种方式来实现：express的内置支持或者使用代理服务器。

10.4.1 生成测试证书

要让加密传输在我们的开发机器上正常工作，需要生成一些测试证书，包括两个文件——privkey.pem和newcert.pem——私有密钥和对应的证书。所有的UNIX/Mac机器都带有叫做openssl的工具，可以用它来生成这两个文件。

对于Windows用户，可以访问<http://gnuwin32.sourceforge.net/packages/openssl.htm>并下载Win32版本的openssl.exe安装程序。然后就可以和其他平台一样运行命令了。

要生成这两个证书文件，运行下面的三个命令：

```
openssl genrsa -out privkey.pem 1024
openssl req -new -key privkey.pem -out certreq.csr
openssl x509 -req -days 3650 -in certreq.csr -signkey privkey.pem -out newcert.pem
```

有了这两个文件，就可以在应用中用它们来工作。注意，绝不能在生产环境中使用它们，如果尝试在浏览器中查看受这些证书保护的站点，就会听到异常大声的危险声音，警告这个网站是不被信任的。通常必须从受信任的证书颁发机构购买生产环境的证书。

10.4.2 内置支持

不必惊讶，通过Node提供的内置https模块，Node.js和express提供了对SSL/HTTPS数据流的支持。实际上我们是运行一个https模块服务器来监听HTTPS端口（默认是443，但在开发阶段，可以用8443替代，以避免需要提升Node进程的权限），然后当加密数据流（encrypted stream）经过握手和创建之后，https模块会将请求传输到express服务器。

我们创建HTTPS服务器，并将对站点签名的私有密钥和证书文件的地址作为可选参数传入。也可以将它传给express服务器，它能在加密建立之后发送数据。具体实现如代码清单10.5所示：

代码清单10.5 express/https模块的SSL支持

(https_express_server.js)

```
var express = require('express'),
    https = require('https'),
    fs = require('fs');

// 1. Load certificates and create options
var privateKey = fs.readFileSync('privkey.pem').toString();
var certificate = fs.readFileSync('newcert.pem').toString();
var options = {
  key : privateKey,
  cert : certificate
}
// 2. Create express app and set up routing, etc.
var app = express();
app.get("*", function (req, res) {
  res.end("Thanks for calling securely!\n");
});

// 3. start https server with options and express app.
https.createServer(options, app).listen(8443, function(){
  console.log("Express server listening on port 8443");
});
```

可以通过在Web浏览器的地址栏打开https://localhost:8443来访问这些加密页面。首次浏览它们时，会提示它们是不安全的。

10.4.3 代理服务器支持

正如在本章中探究的其他特性一样，我们也可以使用强大的http-proxy模块来处理SSL/HTTPS传输。对比内置的HTPPS支持，使用这个模块有两个明显的优势，它能够让应用服务器像普通的HTTP服务器一样运行，让它们“隐藏”在HTTPS代理服务器背后并让我们从乏味的加密工作中释放出来。也可以运行使用之前见过的循环负载均衡，或者看看有没有其他创建方式来设置我们的配置。

这种支持的使用方法和之前express的例子没有太大差异：创建一个Node的内置HTTPS服务器类的实例，再创建一个用来路由请求到普通HTTP服务器（应用服务器）的代理服务器的实例。然后运行HTTPS服务器，当它建立安全连接后，会将请求传给代理服务器，再传递给应用服务器。具体实现如代码清单10.6所示。

代码清单10.6 http-proxy SSL支持 (https_proxy_server.js)

```
var fs = require('fs'),
    http = require('http'),
    https = require('https'),
    httpProxy = require('http-proxy');

// 1. Get certificates ready.
var options = {
  https: {
    key: fs.readFileSync('privkey.pem', 'utf8'),
    cert: fs.readFileSync('newcert.pem', 'utf8')
  }
};

// 2. Create an instance of HttpProxy to use with another server
var proxy = new httpProxy.HttpProxy({
  target: {
    host: 'localhost',
    port: 8081
  }
});

// 3. Create https server and start accepting connections.
https.createServer(options.https, function (req, res) {
  proxy.proxyRequest(req, res)
}).listen(8443);
```

上示代码看起来有很多的路由和重定向，但Node开发团队非常注重性能，这些组件也相当轻便，所以在为Web应用处理请求时不会增加太多明显的延迟。

10.5 多平台开发

Node.js的另一优势是，它不仅能很好地支持类UNIX和Mac系统，也可以运行在Windows机器上。在编写本书时，对所有的示例都测试过，并且在Windows上运行这些示例也没有任何问题。

实际上，我们可以让人们在任何他想要的平台上进行项目开发，只要做好应付两个可能突发的小问题的准备即可：配置上的不同和路径区别。

10.5.1 位置和配置文件

Windows和类UNIX操作系统在不同的地方保存文件。一个能减轻该问题带来的影响的方法就是使用配置文件来存储这些地址。这样，代码会变得足够灵活，它知道去哪里查找东西，而不必根据不同平台分别处理。

实际上，在第8章和第9章中就开始使用这种技术，我们将数据库配置信息保存到一个叫做local.config.js的文件。现在可以继续使用这种技术，并扩大它的使用范围，通常可以保存任何会影响应用运行的信息到这个文件。事实上，这种技术并不仅限于保存文件的位置或路径，还可以在文件中配置端口号或构建类型：

```
exports.config = {
  db_config: {
    host: "localhost",
    user: "root",
    password: "",
    database: "PhotoAlbums",

    pooled_connections: 125,
    idle_timeout_millis: 30000
  },

  static_content: "../static/",
  build_type: "debug"      // show extra output
};
```

实际上，通常我们需要做的就是确认该文件处于版本控制系统之下（例如Github），而且是以local.config.js-base而不是local.config.js保存在源代码树中。要运行应用，只需拷贝这个基础文件并修改成local.config.js，针对本地的运行设置更新成合适的值，然后运行应用。任何时候想使用这些本地配置变量，只需添加如下代码：

```
var local = require('local.config.js');
```

然后，在代码中直接引用变量local.config.variable_xyz即可。

10.5.2 处理路径差异

Windows和类UNIX操作系统的另外一个主要区别就是路径。当面对诸如需要require处理程序，而这些处理程序却存放在项目的子文件夹下等问题时，我们应该如何编码呢？

好消息是，绝大多数情况下，我们会发现Node接受斜杠（/）字符。当我们从相对路径（例如“.. / path / to / sth”）引用模块时，Node会和你期望的一样去“工作”。即使是使用fs模块的API，大部分的API也能够处理两个平台之间不同的路径类型。

对于那些必须去处理不同路径格式的情况，可以使用Node.js的path.sep属性，它能够灵活使用数组的连接和分割，例如：

```
var path = require('path');
var comps = [ '..', 'static', 'photos' ];
console.log(comps.join(path.sep));
```

Node.js的process全局对象总是能够告诉我们当前的运行平台，可以通过如下代码进行查询：

```
if (process.platform === 'win32') {
    console.log('Windows');
} else {
    console.log('You are running on: ' + process.platform);
}
```

10.6 小结

在本章中，我们了解了如何在生产环境中运行Node应用，涵盖了脚本执行、负载均衡和多进程应用。本章还展示了如何基于SSL使用HTTPS保护Web应用安全，最后描述了多平台开发，结论就是它们并没有想象中的那么吓人。

现在我们有一系列漂亮的工具集合来构建和运行异步应用。不过，在下一章，我们会看到Node.js更酷的用法，会发现Node.js还有一堆同步API，能让Node.js完成命令行编程和脚本编程。

第11章 命令行编程

本书花了大量的篇幅和时间来阐述Node.js平台的强大之处，它可以创建异步、非阻塞应用。而在Node中使用传统的同步、阻塞IO编程同样出色——编写命令行程序。事实表明，编写JavaScript脚本十分有趣，很多人开始使用Node编写日常脚本和其他小程序。

在本章中，我们会学习到在Node中运行命令行脚本的基本知识，包括一些常用的同步文件系统API等。然后会学习如何利用缓冲和无缓冲输入与用户交互，最后会看一下进程的创建和执行。

11.1 运行命令行脚本

在类UNIX、Mac操作系统以及Windows平台下，从命令行运行Node.js脚本时有很多选项。因此，Node.js脚本表现得与操作系统命令几乎一模一样，并且可以将命令行参数传递给Node。

11.1.1 UNIX和Mac

在类UNIX操作系统中，如Linux或者Mac OS X，绝大部分用户使用的是Bourne Shell，即sh（或者另外一个流行的变型——bash）。尽管shell脚本语言功能齐全、性能强大，但是如果有时使用JavaScript来编写脚本的话，则会更有趣。

在shell中运行Node程序有两种方式。其中最简单的运行方式和平常运行其他程序没有任何不同——指定node程序和需要运行的脚本（事实上，.js扩展名一般是可选的）：

```
node do_something
```

第二种方式则是直接修改文件的第一行代码，使其变为可执行文件，如下所示：

```
#!/usr/local/bin/node
```

在计算机科学中，文件中第一行的前两个字符（#!）一般被称为shebang。在文件中存在shebang的情况下，类Unix操作系统的程序载入器会分析Shebang后的内容，将这些内容作为解释器指令，并调用该指令，将载有shebang的文件路径作为该解释器的参数。在前面的示例中，操作系统会在/usr/local/bin目录下查找可执行的node，运行Node并将文件路径作为参数传递给Node。Node解释器会忽略第一行中以#!开头的代码，因为它知道这些信息是给操作系统用的。

所以，可以在Node中写出如下所示的可执行代码：

```
#!/usr/local/bin/node  
console.log("Hello World!");
```

要让该程序可运行，需要使用chmod命令，将其标记为可执行文件：

```
chmod 755 hello_world
```

现在就可以直接运行该脚本啦，输入

```
$ ./hello_world  
Hello World!  
$
```

但是，这种实现方式还有一点小问题：当脚本文件拷贝到其他电脑中，而node解释器却没有在/usr/local/bin目录下而是在/usr/bin目录下会怎样呢？通常的做法是修改shebang行的内容，如下所示：

```
#!/usr/bin/env node
```

env命令会查询PATH环境变量，找到第一个node程序实例然后运行。所有的UNIX/Mac平台在/usr/bin目录下都存在env命令，因此，只要系统PATH变量中包含node，这种方式会更便捷。

如果经常在各种操作系统的可执行目录或者其他安装软件的目录下闲逛，就可以发现一些常用的程序，实际上，就是一些脚本。

小贴士：Node解释器非常智能，即使在Windows平台下也可以省略#!语法，而写出来的脚本仍然可以被识别并运行！

11.1.2 Windows

微软的Windows平台从来就没有过一个出色的脚本环境。虽然在Windows Vista和Windows 7中都引入了PowerShell，相较于以前有了很大的提高，但还是不如类UNIX平台下的shell强大和易用。好消息是，这并不会产生太大的影响，因为我们只是想启动并运行脚

本，仅此而已。而所有现代（或半现代）版的Windows平台满足这些需求绰绰有余。

我们还可以使用批处理文件（带有.bat扩展名的可执行文件）运行这些脚本。比如前文中所见的hello_world.js文件，可以创建一个hello_world.bat文件来执行它，如下所示：

```
@echo off  
node hello_world
```

默认情况下，Windows会输出批处理文件中所有的命令。

@echo off用来禁止输出。现在，要运行程序，只需要简单地输入

```
C:\Users\Mark\LearningNode\Chapter11> hello_world  
Hello World!  
C:\Users\Mark\LearningNode\Chapter11>
```

但是，该程序中有一个bug！如果hello_world.bat和hello_world.js文件在同一个文件夹下，但却在其他目录下执行该批处理文件，如下所示：

```
C:\Users\Mark\LearningNode\Chapter11> cd \  
C:\> \Users\Mark\LearningNode\Chapter11\hello_world  
module.js:340  
    throw err;  
^  
Error: Cannot find module 'C:\hello_world'  
    at Function.Module._resolveFilename (module.js:338:15)  
    at Function.Module._load (module.js:280:25)  
    at Module.runMain (module.js:492:10)  
    at process.startup.processNextTick.process._tickCallback (node.js:244:9)
```

由于没有指定hello_world.js的完整路径，因此，Node找不到该文件。只需要修改hello_world.bat脚本就可以修复这个问题，如下所示：

```
@echo off  
node %~d0%~p0\hello_world
```

这里利用了两个便捷的批处理文件的宏：%~pXX和%~dXX，即分别指定第n个参数（即第0个或者运行脚本的名称）的盘符和路径。现在，在计算机中任何地方运行hello_world.bat文件都能正常工作：

```
Z:\> C:\Users\Mark\LearningNode\Chapter11\hello_world  
Hello World!
```

11.1.3 脚本和参数

要想让脚本变得更加有趣，可以向脚本中传递参数。在这种情况下，一般需要关注两个问题：如何向脚本传递参数和如何在脚本中访问参数。

如果一直是通过调用解释器来运行Node脚本（在所有平台下都有效），就无需额外做任何事情——参数会直接传递给正在执行的脚本：

```
node script_name [args]*
```

在UNIX/Mac下传递参数

在类UNIX操作系统中，如果使用#!语法启动脚本，这些参数会直接传递给运行的脚本，因此，完全不需要做任何额外操作。

在Windows下传递参数

在Window平台下，如果使用带.bat扩展名的批处理文件运行Node脚本，可以将参数传递给这些批处理文件，然后相应地，将这些参数通过使用宏%*的方式传递给Node.js脚本。因此，批处理文件会如下所示：

```
@echo off  
node %~d0\%~p0\params %*
```

在Node中访问参数

所有传递进来的参数都会被保存到全局对象process的argv数组中。数组的前两个元素一般是当前的node解释器路径和正在运行的脚本路径。因此，任何传递进来的参数都是从第三个元素，即索引下标2开始的。现在运行如下代码：

```
#!/usr/local/bin/node
console.log(process.argv);
```

可以看到如下输出结果：

```
Kimidori:01_running marcw$ ./params.js 1 2 3 4 5
[ '/usr/local/bin/node',
  '/Users/marcw/src/misc/LearningNode/Chapter11/01_running/params.js',
  '1',
  '2',
  '3',
  '4',
  '5' ]
```

11.2 同步处理文件

几乎所有的文件系统模块（fs）的每一个API都同时提供异步和同步两个版本。到目前为止，大部分都采用了异步版本的API。但是，一旦要编写命令行程序，同步版本的API就尤为重要了。基本上，fs中几乎每一个以func命名的API都有一个相应的叫做funcSync的API。

接下来的章节中，将会用些示例来演示和说明如何使用同步API。

11.2.1 基本文件API

fs模块不提供文件拷贝函数，因此我们可以自己写一个。open、read和write这些API都有对应的同步版本：openSync、readSync和writeSync。在同步版本中，一旦API出问题，就会抛出错误。当我们将文件从位置a拷贝到位置b的时候，可以使用一个缓冲对象来保存数据。在文件拷贝完成以后，一定要关闭文件接口。

```
var BUFFER_SIZE = 1000000;

function copy_file_sync (src, dest) {
    var read_so_far, fdsrc, fddest, read;
    var buff = new Buffer(BUFFER_SIZE);

    fdsrc = fs.openSync(src, 'r');

    fddest = fs.openSync(dest, 'w');
    read_so_far = 0;

    do {
        read = fs.readSync(fdsrc, buff, 0, BUFFER_SIZE, read_so_far);
        fs.writeSync(fddest, buff, 0, read);
        read_so_far += read;
    } while (read > 0);

    fs.closeSync(fdsrc);
    return fs.closeSync(fddest);
}
```

在调用该函数的时候，为了确保有足够的参数，可以写一个

file_copy.js脚本来调用该拷贝函数，从而可以处理调用过程中抛出的错误：

```
if (process.argv.length != 4) {
    console.log("Usage: " + path.basename(process.argv[1], '.js')
        + " [src_file] [dest_file]");
} else {
    try {
        copy_file_sync(process.argv[2], process.argv[3]);
    } catch (e) {
        console.log("Error copying file:");
        console.log(e);
        process.exit(-1);
    }

    console.log("1 file copied.");
}
```

可以看到上面的代码中使用到了一个新函数——`process.exit`。该函数会立即终止Node.js程序并将状态码返回给调用程序（一般会是shell解释器或者命令提示符）。Bourne shell（sh或者bash）的标准是：当返回的状态码为0时表明执行成功；非0的状态码则表明执行失败。当执行该拷贝函数出错时，就会返回-1。

这里，我们还可以稍作修改，将文件拷贝函数改造成文件移动函数。首先，执行文件拷贝操作，然后在确定目标文件已经完全写入并成功关闭以后，删除源文件。可以使用`unlinkSync`函数进行删除文件操作：

```
function move_file_sync (src, dest) {
    var read_so_far, fdsrc, fddest, read;
    var buff = new Buffer(BUFFER_SIZE);

    fdsrc = fs.openSync(src, 'r');
    fddest = fs.openSync(dest, 'w');
    read_so_far = 0;

    do {
        read = fs.readSync(fdsrc, buff, 0, BUFFER_SIZE, read_so_far);
        fs.writeSync(fddest, buff, 0, read);
        read_so_far += read;
    } while (read > 0);

    fs.closeSync(fdsrc);
    fs.closeSync(fddest);
    return fs.unlinkSync(src);
}
```

其余的脚本保持不变，只需要将函数名由`copy_file_sync`改为`move_file_sync`即可。

11.2.2 文件和状态

在Node中，可以使用文件系统模块中的mkdir函数创建文件夹，在下面的脚本中，使用了mkdirSync函数。现在，我们要写一个程序，其功能相当于UNIX shell中的mkdir-p命令：指定一个完整路径，然后创建该目录以及该路径中间所有缺失的目录。

该过程一共包含两步：

1) 首先，拆分路径，然后从上至下判断每一级目录是否存在。如果在某一级路径下存在文件，却不是目录时（也就是说，想要使用mkdir a/b/c创建文件夹，但是a/b已经存在，而且只是一个普通文件），则抛出错误。要判断一个文件对象是否存在，则使用existsSync函数；而要确定该文件对象是否是一个目录，则可以调用statsSync函数，该函数会返回一个Stats对象，通过它可以判断是否为目录。

2) 遍历所有的路径，并为所有缺失目录的路径创建目录。

下面是mkdirs函数的代码：

```
function mkdirs (path_to_create, mode) {
  if (mode == undefined) mode = 0777 & (~process.umask());

  // 1. What do we have already or not?
  var parts = path_to_create.split(path.sep);
  var i;
  for (i = 0; i < parts.length; i++) {
    var search;
    search = parts.slice(0, i + 1).join(path.sep);
    if (fs.existsSync(search)) {
      var st;
      if ((st = fs.statSync(search))) {
        if (!st.isDirectory())
          throw new Error("Intermediate exists, is not a dir!");
      }
    } else {
      // doesn't exist. We can start creating now
      break;
    }
  }

  // 2. Create whatever we don't have yet.

  for (var j = i; j < parts.length; j++) {
    var build = parts.slice(0, j + 1).join(path.sep);
    fs.mkdirSync(build, mode);
  }
}
```

该函数的第一行是用来设置权限掩码的，可以用来设置刚创建的

目录的读写权限。函数在调用的时候就可以直接设置这些权限，或者如果当前shell用户不想赋予新文件（或者目录）某个权限，可以使用umask过滤掉。在Windows下，如果umask返回0，则表明没有屏蔽任何权限；Windows使用了和UNIX完全不一样的文件权限机制。

11.2.3 目录内容

要想列出某个目录中所有的内容，可以使用readdirSync函数，它会返回指定文件夹下的所有文件名数组，不包括“.”和“..”。

```
#!/usr/local/bin/node
var fs = require('fs');

var files = fs.readdirSync(".");
console.log(files);
```

11.3 用户交互：标准输入和输出

你可能会对所有进程的IO处理三元组很熟悉：stdin、stdout和stderr，分别代表了标准输入、标准输出和错误输出。同样，在Node.js脚本中也提供相同的功能，该功能被挂载到process对象中。在Node中，它们实际上都是Stream对象的实例（请参见第6章）。

实际上，Node中的console.log函数等价于

```
process.stdout.write(text + "\n");
```

而console.error则与

```
process.stderr.write(text + "\n");
```

等价。

但是，输入提供了很多选项，在下面的章节中，将会介绍到缓冲输入（逐行输入）和无缓冲输入（一旦输入一个字符，就会立刻输出出来）的对比。

11.3.1 基本缓冲输入和输出

默认情况下，每次只能从stdin数据流中读取和缓冲一行数据。因此，想要从stdin中读取数据，只有当用户按下回车（Enter）键，程序才会从输入流中读取整行数据。可以通过向stdin添加readable事件监听器实现，如下所示：

```
process.stdin.on('readable', function () {
    var data = process.stdin.read();

    // do something w input
});
```

但是默认情况下，stdin输入流处于暂停状态。因此，需要调用resume函数才能开始从输入流中接收数据：

```
process.stdin.resume();
```

下面写一个小程序，读取输入行，然后使用md5加密输入数据并将其打印出来，不断重复循环，直到按下Ctrl+C或者空行才退出：

```
process.stdout.write("Hash-o-tron 3000\n");
process.stdout.write("(Ctrl+C or Empty line quits)\n");
process.stdout.write("data to hash > ");

process.stdin.on('readable', function () {
    var data = process.stdin.read();
    if (data == null) return;
    if (data == "\n") process.exit(0);

    var hash = require('crypto').createHash('md5');
    hash.update(data);
    process.stdout.write("Hashed to: " + hash.digest('hex') + "\n");
    process.stdout.write("data to hash > ");
});

process.stdin.setEncoding('utf8');
process.stdin.resume();
```

上述代码在用户按下回车键的一瞬间就开始工作了：首先，检查输入是否为空，如果不为空，就用md5加密并打印出来；然后出现新的提示符，等待用户输入其他内容。由于stdin处于非暂停状态，因此Node程序不会退出（如果尝试暂停接收stdin输入流，并且没有其他任务需要执行，则程序会立即退出）。

11.3.2 无缓冲输入

在某些情况下，如果想让用户按下键盘以后程序立即响应，可以通过使用setRawMode函数来开启stdin输入流中的原始模式（raw mode），参数可以使用布尔值来设置是否开启（true为开启）原始模式。

同时，更新前一小节中的代码，让用户可以任意选择喜欢的哈希类型。在用户输入一行文本并按下回车键以后，程序会让用户按下数字键1到4以选择不同的哈希算法。该程序的完整代码如代码清单11.1所示。

代码清单11.1 使用标准输入中的RawMode (raw_mode.js)

```
process.stdout.write("Hash-o-tron 3000\n");
process.stdout.write("(Ctrl+C or Empty line quits)\n");
process.stdout.write("data to hash > ");

process.stdin.on('readable', function (data) {
    var data = process.stdin.read();
    if (!process.stdin.isRaw) { // 1.
        if (data == "\n") process.exit(0);
        process.stdout.write("Please select type of hash:\n");
        process.stdout.write("(1 - md5, 2 - sha1, 3 - sha256, 4 - sha512) \n");
        process.stdout.write("[1-4] > ");
        process.stdin.setRawMode(true);
    } else {
        var alg;
        if (data != '^C') { // 2.
            var c = parseInt(data);
            switch (c) {
                case 1: alg = 'md5'; break;
                case 2: alg = 'sha1'; break;
                case 3: alg = 'sha256'; break;
                case 4: alg = 'sha512'; break;
            }
            if (alg) { // 3.
                var hash = require('crypto').createHash(alg);
                hash.update(data);
                process.stdout.write("\nHashed to: " + hash.digest('hex'));
                process.stdout.write("\ndata to hash > ");
                process.stdin.setRawMode(false);
            } else {
                process.stdout.write("\nPlease select type of hash:\n");
                process.stdout.write("[1-4] > ");
            }
        } else {
            process.stdout.write("\ndata to hash > ");
            process.stdin.setRawMode(false);
        }
    }
});

process.stdin.setEncoding('utf8')
process.stdin.resume();
```

由于该脚本从stdin中接收数据，既支持缓冲输入（当请求输入需要哈希的文本时），也支持无缓冲输入（当请求选择哈希算法时），因此，它稍微有点复杂。让我们看下它究竟是如何工作的。

1) 首先，会检查接收的输入是缓存输入还是无缓存输入。如果是前者，则是一行需要哈希的输入文本，然后会打印出选择使用算法的请求。由于脚本希望用户只能按下一个1到4的数字键，因此将stdin切换到RawMode模式，现在，一旦输入任何一个键都会触发readable事件。

2) 如果输入流是RawMode模式，这就意味着用户按下一个按键就会响应哈希算法的请求。这部分的第一行会检测输入的键值是否为Ctrl+C（注意，可以进入一个文本编辑器；如在Emacs中，可以使用Ctrl+Q，然后按下Ctrl+C，它会输出^C字符。每个编辑器都会稍有不同）。如果用户按下了Ctrl+C，脚本会中断请求，并返回到哈希提示符。

如果用户输入了其他键值，脚本会判断是否为有效键值（1到4），如果不是，脚本会提示让用户再重新输入一遍。

3) 最后，根据选择的算法，脚本生成相应的哈希值并打印出来，然后返回到原先的请求输入数据的提示符。在这之前，一定要记得关闭RawMode模式，这样才能返回到正常的缓冲输入模式。

同时，由于在程序中调用了stdin的resume函数，所以只有在调用process.exit、用户在输入空行或者在缓冲输入模式下输入Ctrl+C（这会导致Node终止程序）时，程序才会正常退出。

11.3.3 Readline模块

另一种使用Node.js中输入流的方式就是使用readline模块。由于它仍然被标记为不稳定状态，极有可能更改API，因此，我不会花费大量的时间和精力在它上面，但是，它还是有一些精巧的特性值得我们在程序中使用。

要想使用readline模块，需要调用它的createInterface方法，指定参数选项中的输入流和输出流：

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

var rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout
});
```

完成这些以后，程序只有在调用rl.close之后才会正常退出。

逐行提示

如果调用readline的prompt方法，该程序就会等待一行的输入（直到回车）。当程序捕捉到回车键按下，就会触发readline对象中的line事件，这样该事件就可以处理输入数据了：

```
rl.on("line", function (line) {  
    console.log(line);  
    rl.prompt();  
});
```

如果需要继续监听事件，则需要再次调用prompt方法。

readline接口最神奇的地方就是一旦用户按下Ctrl+C，SIGINT事件就会被调用，那么就可以选择关掉或者恢复状态，继续监听。这里，通过关闭readline接口，让程序停止监听输入流并退出。

```
rl.on("SIGINT", function ()  
    rl.close();  
});
```

现在，可以尝试使用readline模块来编写一个简易的逆波兰式计算器。计算器代码如清单11.2所示。

如果你从来没有听说过逆波兰式表示法或者忘记它是如何运行的，没有关系，它只是一种后缀数学符号格式。当计算 $1+2$ 时，该表示法会写成 $1\ 2+$ ；当计算 $5^*(2+3)$ 时，它会写成 $5\ 2\ 3+*$ ，等等。这个简易计算器一次只接收一个字符串，并使用空格将字符分开，并做简单的计算。

代码清单11.2 基于readline的简易后缀计算器（ readline.js ）

```
var readline = require('readline');
var rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout
});

var p = "postfix expression > "
rl.setPrompt(p, p.length);
rl.prompt(); // 1.

rl.on("line", function (line) { // 2.
    if (line == "\n") {
        rl.close(); return;
    }

    var parts = line.split(new RegExp("[ ]+"));
    var r = postfix_process(parts);
    if (r !== false)
        process.stdout.write("Result: " + r + "\n");
    else
        process.stdout.write("Invalid expression.\n");
    rl.prompt(); // 3.
});

rl.on("SIGINT", function () { // 4.
    rl.close();
});

// push numbers onto a stack, pop when we see an operator.
function postfix_process(parts) {
    var stack = [];
    for (var i = 0; i < parts.length; i++) {
        switch (parts[i]) {
            case '+': case '-': case '*': case '/':
                if (stack.length < 2) return false;
                do_op(stack, parts[i]);
                break;
            default:
                var num = parseFloat(parts[i]);
                if (isNaN(num)) return false;
                stack.push(num);

                break;
        }
    }
    if (stack.length != 1) return false;
    return stack.pop();
}

function do_op(stack, operator) {
    var b = stack.pop();
    var a = stack.pop();
    switch (operator) {
        case '+': stack.push(a + b); break;
        case '-': stack.push(a - b); break;
        case '*': stack.push(a * b); break;
        case '/': stack.push(a / b); break;
        default: throw new Error("Unexpected operator");
    }
}
```

该程序工作流程如下：

- 1) 首先，创建readline模块对象，设置默认的提示符文本，然后输出该提示符并等待输入。
- 2) 当接收到一行输入的时候，检测它是否为空（如果为空，则关闭readline接口，退出整个程序），否则解析输入的字符串，并将其作为参数传递给计算函数。计算完成以后，打印计算结果（成功或失败）。
- 3) 完成本次计算，告知readline打印提示符并继续等待下次输入

入。

4) 如果用户按下Ctrl+C , 则程序会关闭readline接口实例 , 继而程序正常退出。

至此 , 可以完整测试该程序了 :

```
Kimidori:03_stdinout marcw$ node readline_rpn.js
postfix expression > 1 2 +
Result: 3
postfix expression > 2 3 4 + *
Result: 14
postfix expression > cat
Invalid expression.
postfix expression > 1 2 4 cat dog 3 4 + - / *
Invalid expression.
postfix expression > 2 3 + 5 *
Result: 25
postfix expression >
```

问题

readline模块另一个重要功能就是可以提问 , 并直接在回调函数中接收答案。基本格式如下 :

```
rl.question("hello? ", function (answer) {
    // do something
});
```

接下来 , 我们要写一个调查问卷程序 : 它包含一组问题 (可以将问题放到文件中 , 这样问题是可配置的) 。每次向用户问一个问题 , 然后使用fs模块中的appendFileSync函数将用户的答案写进answers.txt中。

由于question函数是异步的 , 所以必须使用async.forEachSeries来迭代调查的每一个问题。调查程序如代码清单11.3所示。

代码清单11.3 调查问卷程序 (question.js)

```
var readline = require('readline'),
    async = require("async"),
    fs = require('fs');

var questions = [ "What's your favorite color? ",
                 "What's your shoe size? ",
                 "Cats or dogs? ",
                 "Doctor Who or Doctor House? " ];

var rl = readline.createInterface({ // 1.
    input: process.stdin,
    output: process.stdout
});
var output = [];
async.forEachSeries(
    questions,
    function (item, cb) { // 2.
        rl.question(item, function (answer) {
            output.push(answer);
            cb(null);
        });
    },
    function (err) { // 3.
        if (err) {
            console.log("Hunh, couldn't get answers");
            console.log(err);
            return;
        }
        fs.appendFileSync("answers.txt", JSON.stringify(output) + "\n");
        console.log("\nThanks for your answers!");
        console.log("We'll sell them to some telemarketer immediately!");
        rl.close();
    }
);
}
```

该程序主要包含如下流程：

- 1) 首先，初始化readline模块，设置stdin和stdout数据流。
- 2) 然后，对于数组中的每一个问题，调用readline上的question函数（由于question是异步函数，因此需要使用async.forEachSeries）并将结果添加到输出数组中。
- 3) 最后，所有问题回答结束以后，async会调用回调函数，当出错时，会打印出错误信息；或者将用户的答案附加到answers.txt文件中，然后关闭readline对象，退出程序。

11.4 进程处理

Node中还可以使用命令行（甚至在Web应用中）启动其他程序。使用child_process模块，有两种不同复杂程度的选项，我们将从简单的exec开始。

11.4.1 简单进程创建

Child_process模块中的exec函数会接收一个命令并在系统shell中执行（UNIX/Mac平台下的sh/bash，或者Windows下的cmd.exe）。因此，可以指定一个简单的命令程序如"date"来运行，或者稍微复杂点的命令，比如"echo'Mancy'|sed s/M/N/g"。所有的命令运行以后，将输出缓存，并会在命令执行完成以后返回给调用者。

基本格式如下：

```
exec(command, function (error, stdout, stderr) {  
    // error is if an error occurred  
    // stdout and stderr are buffers  
});
```

当命令执行完成以后，回调函数会被调用。当出错时，第一个参数为非空。否则，所有的内容将会被写到对应输出流stdout和stderr的Buffer对象中。

现在，我们尝试写一个程序使用exec函数来运行cat程序（即Windows中的type程序），它需要指定一个文件名称。该程序使用exec函数启动cat/type程序，并在执行完成以后打印所有输出信息：

```
var exec = require('child_process').exec,
    child;

if (process.argv.length != 3) {
    console.log("I need a file name");
    process.exit(-1);
}

var file_name = process.argv[2];
var cmd = process.platform == 'win32' ? 'type' : "cat";
child = exec(cmd + " " + file_name, function (error, stdout, stderr) {
    console.log('stdout: ' + stdout);
    console.log('stderr: ' + stderr);

    if (error) {
        console.log("Error exec'ing the file");
        console.log(error);
        process.exit(1);
    }
});
```

11.4.2 使用Spawn创建进程

另一种高级的创建进程的方式是使用child_process模块中的spawn函数。该函数拥有其创建出来的子进程的stdin和stdout的完整控制权，这样可以使用一些奇妙的功能，如将输出从一个子进程通过管道传入另一个子进程中。

下面的程序需要传入一个JavaScript文件的名称，并使用node程序运行该脚本：

```
var spawn = require("child_process").spawn;
var node;

if (process.argv.length != 3) {
    console.log("I need a script to run");
    process.exit(-1);
}

var node = spawn("node", [ process.argv[2] ]);
node.stdout.on('readable', print_stdout);
node.stderr.on('readable', print_stderr);
node.on('exit', exited);

function print_stdout() {
    var data = process.stdout.read();
    console.log("stdout: " + data.toString('utf8'));
}
function print_stderr(data) {
    var data = process.stderr.read();
    console.log("stderr: " + data.toString('utf8'));
}
function exited(code) {
    console.error("--> Node exited with code: " + code);
}
```

当调用spawn时，第一个参数是需要执行的命令的名称，第二个参数是传进来的参数数组。可以看到子进程中任何输出流写入到stdout和stderr时，都会立即触发对应数据流上的事件，并可以实时看到发生的一切。

现在，我们可以尝试写一些更高级的功能。在前一章中，可以看到利用shell脚本或者命令行提示符实现下述功能非常高效：

```
while 1
    node script_name
end
```

而要使用JavaScript实现相同的功能，可以使用spawn函数。当然，这比shell脚本要复杂一些。但是这是有益的，它可以让我们做一些额外的工作，以便得到任何想要的监控数据。新的启动器如代码清单11.4所示。

代码清单11.4 全Node执行程序 (node_runner.js)

```
var spawn = require("child_process").spawn;
var node;

if (process.argv.length < 3) {
  console.log("I need a script to run");
  process.exit(-1);
}

function spawn_node() {
  var node = spawn("node", process.argv.slice(2));
  node.stdout.on('readable', print_stdout);
  node.stderr.on('readable', print_stderr);
  node.on('exit', exited);
}

function print_stdout() {
  var data = process.stdout.read();
  console.log("stdout: " + data.toString('utf8'));
}
function print_stderr(data) {
  var data = process.stderr.read();
  console.log("stderr: " + data.toString('utf8'));
}
function exited(code) {
  console.error("--> Node exited with code: " + code + ". Restarting");
  spawn_node();
}

spawn_node();
```

该程序监听新创建的子进程中的exit事件，当该事件被触发时，程序会使用node解释器重启想要运行的脚本。

现在给你留一份练习作业：更新上述脚本，让其功能与前一章中的node_ninja_runner一样。使用exec函数获得ps aux命令执行以后的所有输出内容；该输出结果可以使用JavaScript解析。最后，如果检测到输出内容过大，可以使用kill方法来结束子进程。

11.5 小结

现在，我们见识到Node不仅擅长编写网络应用，而且在同步的命令行应用中也有独到之处。在经过简单的创建、运行脚本并传递参数给脚本这一系列的旅程后，你应该对Node程序操作输入流和输出流得心应手，甚至在必要的时候能在缓冲输入流和无缓冲输入流中来回切换。最后，我们学习了如何使用exec和spawn创建和运行Node.js脚本程序。

在结束这一章的Node.js编程学习之后，我们将会把注意力集中到最重要的脚本和应用测试上面。

第12章 测试

程序写到现在，我们准备看一下如何测试以保证程序的正常运行。现在有很多成熟的测试模型和范式，而Node.js支持其中的绝大部分。在本章中，我们会集中了解其中最常用的一些测试模型，然后学习如何进行功能测试——不仅仅只测试同步API，还要测试Node的异步代码。最后，为相册应用添加相应的测试代码。

12.1 测试框架选择

现今，有很多流行的测试模型，包括测试驱动开发（test-driven development，TDD）、行为驱动开发（behavior-driven development，BDD）等。前者是用来确保所有的代码都拥有合适的测试接口（实际上，很多情况下，必须先有测试用例，后有开发代码）；而后者则是专注于某个单元或者代码模块的业务需求，要求测试得更全面，而不仅仅是简单的单元测试。

不考虑使用哪一种测试模型（或者打算使用，如果这是你第一次写测试的话），将测试代码添加到代码库中是个不错的主意，因为不仅仅要确保现在的代码库没有问题，而且要保证将来代码库修改以后不会带来其他问题。在Node应用中添加测试是需要一些挑战的，因为经常需要将同步、异步和RESTful服务API功能混合到一起。但是，不用担心，因为Node.js平台足够健壮和优秀，它已经准备了一些不错的选择来满足我们所有的需求。

在拥有琳琅满目的测试框架的今天，有三款出色的测试框架广受欢迎，脱颖而出：

- [nodeunit](#)——这是一款简单易用的测试框架，同时支持Node和浏览器端测试。它极易使用，并且在定制测试框架方面极为灵活。
- [Mocha](#)——这款测试框架基于一个名叫Expresso的旧测试框架，Mocha是一款功能齐全的TDD Node测试框架，专注于易用性和愉悦编程的理念。它拥有一些非常棒的异步测试的功能，并提供了格式化输出的API。
- [VowsJS](#)——它是Node.js平台下最为卓越的BDD测试框架，VowsJS不仅仅包含非常描述性的语法和结构，能将测试和BDD理念完美结合；更能让测试用例并行，从而提高执行效率。

尽管它们都非常完美，并且有各自的定位，但是在本章中，我们

将专注于简单的nodeunit，很大一部分原因是，它使用起来非常简单且容易演示。

安装Nodeunit

现在，可以在项目的根目录中创建test/子文件夹，并将项目中与测试相关的文件、示例、数据文件等都放进去。第一个放进该文件夹的是package.json文件，如下所示：

```
{  
  "name": "API-testing-demo",  
  "description": "Demonstrates API Testing with nodeunit",  
  "version": "0.0.1",  
  "private": true,  
  "dependencies": {  
    "nodeunit": "0.7.x"  
  }  
}
```

当运行npm update命令时，nodeunit模块就会自动安装，然后就可以开始编写和运行测试用例了。

12.2 编写测试用例

Nodeunit将测试集成到模块内部，即把每一个暴露的函数都当成一个测试，而每一个暴露的对象都会被当成一个测试组。对于每一个测试，都会赋予一个对象参数，它会帮助执行测试用例，并在完成测试时通知nodeunit：

```
exports.test1 = function (test) {
    test.equals(true, true);
    test.done();
}
```

在每一个测试的最后都需要调用`test.done`；否则，nodeunit就无法知道测试是否已经完成。要想运行该测试用例，可以将其保存到名叫`trivial.js`的文件中，并运行`node_modules/.bin`文件夹下的脚本。在Unix/Mac平台或者Windows平台下，可以运行如下命令（当然，在Windows平台下，可以将“/”字符替换成“\”字符）：

```
node_modules/.bin/nodeunit trivial.js
```

执行结束以后，可以看到如下结果：

```
C:\Users\Mark\a> node_modules\.bin\nodeunit.cmd trivial.js

trivial.js
✓ test1

OK: 1 assertions (0ms)

C:\Users\Mark\a>
```

12.2.1 简单功能测试

每写一个测试，都需要做三件事：

1) 调用参数`test`中的`expect`方法，以确认nodeunit在该测试下期望验证的“条件”次数。这一步是可选的，但如果偶尔在一些测试中需要跳过一些测试代码，那么这会是一个不错的选择。

2) 对于每一个需要验证的条件 , 需要用到以下一些断言函数 (见表 12.1) 来验证期望的结果。前面第一个示例中的 `test.equals` 就是其中之一。

3) 在每一个测试的最后调用test.done来通知nodeunit结束测试。

现在，将前一章中编写的逆波兰式计算器代码放进一个叫做 rpn.js 的文件中（见代码清单 12.1）。

代码清单12.1 rpn.js文件

```

// push numbers onto a stack, pop when we see an operator
exports.version = "1.0.0";

exports.compute = function (parts) {
    var stack = [];
    for (var i = 0; i < parts.length; i++) {
        switch (parts[i]) {
            case '+': case '-': case '*': case '/':
                if (stack.length < 2) return false;
                do_op(stack, parts[i]);
                break;
            default:
                var num = parseFloat(parts[i]);
                if (isNaN(num)) return false;
                stack.push(num);
                break;
        }
    }
    if (stack.length != 1) return false;
    return stack.pop();
}

function do_op(stack, operator) {
    var b = stack.pop();
    var a = stack.pop();
    switch (operator) {
        case '+': stack.push(a + b); break;
        case '-': stack.push(a - b); break;
        case '*': stack.push(a * b); break;
        case '/': stack.push(a / b); break;

        default: throw new Error("Unexpected operator");
    }
}

```

现在，为它编写测试代码（千万不要忘记在测试文件中使用 rpn.js 文件中的 require）：

```
exports.addition = function (test) {
    test.expect(4);
    test.equals(rpn.compute(prepare("1 2 +")), 3);
    test.equals(rpn.compute(prepare("1 2 3 + +")), 6);
    test.equals(rpn.compute(prepare("1 2 + 5 6 + +")), 14);
    test.equals(rpn.compute(prepare("1 2 3 4 5 6 7 + + + + + +")), 28);
    test.done();
};
```

`prep`函数用来将提供的字符串分割成数组：

```
function prep(str) {
    return str.trim().split(/\s+/);
}
```

我们可以在测试中添加任何计算器中支持的运算符（减法、乘法和除法），甚至还可以添加小数的测试。

```
exports.decimals = function (test) {
    test.expect(2);
    test.equals(rpn.compute(prep("3.14159 5 *")), 15.70795);
    test.equals(rpn.compute(prep("100 3 /")), 33.33333333333336);
    test.done();
}
```

到目前为止，只使用了`test.equals`断言来验证期望的值。然而，nodeunit还使用了一个叫做`assert`的模块，它提供了其他一些方法，如表12.1所示。

表 12.1 测试断言

方 法	测试说明
<code>ok (value)</code>	测试 value 值是否为真 (true)
<code>equal (value, expected)</code>	测试 value 值是否为 expected 值 (仅使用 == 进行值比较, 而非 ===)
<code>notEqual (value, expected)</code>	确保 value 值不为 expected 值 (仅使用 == 进行值比较)
<code>deepEqual (value, expected)</code>	确保 value 值与 expected 值相等, 如果需要, 会比较子值, 使用 === 进行比较
<code>notDeepEqual (value, expected)</code>	确保 value 值不为 expected 值, 如果需要, 会比较子值, 使用 == 进行比较
<code>strictEqual (value, expected)</code>	测试值是否相等, 使用 === 操作符
<code>throws (code, [Error])</code>	确保代码段抛出错误, 并可选择是否为指定错误类型
<code>doesNotThrow (code, [Error])</code>	确保代码段不抛出错误 (可选, 指定错误类型)

现在，可以添加一个新测试，确保计算器不接受空表达式：

```
exports.empty = function (test) {
    test.expect(1);
    test	throws(rpn.compute([]));
    test.done();
};
```

测试失败以后，nodeunit会报错，并将失败条件和引起错误的完整调用栈信息打印出来：

```
01_functional.js
✗ addition

AssertionError: 28 == 27
    at Object.assertWrapper [as equals]
    [...tests/node_modules/nodeunit/lib/types.js:83:39)
    at Object.exports.addition [...tests/01_functional.js:9:10)
    (etc)
✓ subtraction
✓ multiplication
✓ division
✓ decimals
✓ empty

FAILURES: 1/17 assertions failed (5ms)
```

通过打印信息，可以查看代码中哪个测试导致了错误，分析错误原因并修复。

12.2.2 异步功能测试

在Node.js中会用到很多异步编程，因此很多测试也需要是异步的。Nodeunit将这种理念融于设计之中：无论测试执行多长时间或者异步与否，都没有问题，只要在执行结束时调用`test.done`即可。例如，现在编写两个异步测试：

```
exports.async1 = function (test) {
    setTimeout(function () {
        test.equal(true, true);
        test.done();
    }, 2000);
};

exports.async2 = function (test) {
    setTimeout(function () {
        test.equal(true, true);
        test.done();
    }, 1400);
};
```

执行上面的测试模块会得到如下测试结果（注意：运行时一般会将这两个测试合并起来顺序执行）：

```
Kimidori: functional_tests marcw$ node_modules/.bin/nodeunit 02_async.js

02_async.js
✓ async1
✓ async2

OK: 2 assertions (3406ms)
```

12.3 RESTful API测试

现在可以给相册应用添加两种不同的测试：第一种是功能测试，保证前面编写的各类模块能正常运行；另一种是确保服务器提供一致的功能性REST API。

只要添加一个名叫request的npm模块，就可以使用nodeunit实现第二种测试，request允许使用不同的HTTP方法调用远程URL，并将结果返回给Buffer对象。

例如，要想调用服务器上的/v1/albums.json，可以调用：

```
request.get("http://localhost:8080/v1/albums.json", function (err, resp, body) {  
    var r = JSON.parse(body);  
    // do something  
});
```

上述回调函数的参数包括：

- 错误对象，当调用过程中发生错误时，会将错误信息返回给该对象。
- HttpResponse对象，从中可以获得返回的头信息和状态码。
- Buffer对象，用来保存服务器返回的数据（如果有数据的话）

还记得在第9章中编写的MySQL版相册应用么？把它拷贝到一个新的地方，将数据库清理干净，并重新运行数据库初始脚本schema.sql：

```
mysql -u root < schema.sql
```

然后在根目录下创建一个新的叫做test/的目录：

```
+ photo_album/
+ static/
+ app/
+ test/
```

在test文件夹下，创建一个包含nodeunit和request的package.json文件：

```
{
  "name": "API-testing-demo",
  "description": "Demonstrates API Testing with request and nodeunit",
  "version": "0.0.1",
  "private": true,
  "dependencies": {
    "nodeunit": "0.7.x",
    "request": "2.x"
  }
}
```

现在可以编写第一个RESTful API的测试了：

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

var h = "http://localhost:8080";
exports.no_albums = function (test) {
  test.expect(5);
  request.get(h + "/v1/albums.json", function (err, resp, body) {

    test.equal(err, null);
    test.equal(resp.statusCode, 200);
    var r = JSON.parse(body);
    test.equal(r.error, null);
    test.notEqual(r.data.albums, undefined);
    test.equal(r.data.albums.length, 0);
    test.done();
  });
};

});
```

一开始，数据库中并没有相册信息，因此在获取所有相册信息的时候会返回一个空数组。

要测试相册的创建功能，需要使用PUT方法，将数据发送到服务器中，测试返回的结果。Request模块可以指定JSON数据格式来传递数据，返回的结果中会自动指定Content-Type:application/json响应头，返回结果为JSON格式数据：

```

exports.create_album = function (test) {
  var d = "We went to HK to do some shopping and spend New Year's. Nice!";
  var t = "New Year's in Hong Kong";
  test.expect(7);
  request.put(
    { url: h + "/v1/albums.json",
      json: { name: "hongkong2012",
              title: t,
              description: d,
              date: "2012-12-28" } },
    function (err, resp, body) {
      test.equal(err, null);
      test.equal(resp.statusCode, 200);
      test.notEqual(body.data.album, undefined);
      test.equal(body.data.album.name, "hongkong2012");
      test.equal(body.data.album.date, "2012-12-28");
      test.equal(body.data.album.description, d);
      test.equal(body.data.album.title, t);
      test.done();
    }
  );
}

```

当请求结束以后，可以测试所有的内容，包括HTTP响应的状态码（在成功请求以后，一般会返回200）；可以分解JSON对象，确认数据是否与期望值一致。在一开始，你会注意到上述代码指定必须检测七次条件，这会帮助nodeunit更好地执行测试。最后，不仅需要确认API是否按预期一样正常工作，而且需要确保出错是否在预期之内。因此当一个函数期望返回错误时，一定要进行检测！事实上，当我在写这本书的时候，下面的测试代码帮我检测出
handlers/albums.js中的一些参数验证的错误：

```

exports.fail_create_album = function (test) {
  test.expect(4);
  request.put(
    { url: h + "/v1/albums.json",

      headers: { "Content-Type": "application/json" },
      json: { name: "Hong Kong 2012",           // no spaces allowed!
              title: "title",
              description: "desc",
              date: "2012-12-28" } },
    function (err, resp, body) {
      test.equal(err, null);
      test.equal(resp.statusCode, 403);
      test.notEqual(body.error, null);
      test.equal(body.error, "invalid_album_name");
      test.done();
    }
  );
}

```

因此，不仅要检测HTTP返回的状态码是否为403，而且要检测错误的内容是否与预期一致。

测试受保护的资源

当使用用户名和密码保护服务器资源时（可以使用HTTPS协

议，这样就无法查看用户名和密码了），如果需要测试API，利用URL可以内置HTTP基本身份验证的原理，在request的URL中包含用户名和密码，如下所示：

```
https://localhost:username@password:8080/v1/albums.json
```

因此，要测试站点中的安全部分，需要写如下所示的nodeunit 测试：

```
var h = "localhost:username@secret:8080";
exports.get_user_info = function (test) {
    test.expect(5);
    request.get(h + "/v1/users/marcwan.json", function (err, resp, body) {
        test.equal(err, null);
        test.notEqual(resp.statusCode, 401);
        test.equal(resp.statusCode, 200);
        var r = JSON.parse(body);
        test.equal(r.data.user.username, "marcwan");
        test.equal(r.data.user.password, undefined);
        test.done();
    });
};
```

12.4 小结

通过npm可以找到很多可用的测试框架来测试Node.js应用和脚本，既简单又快速。在本章中，我不仅展示了如何使用目前最流行的TDD框架nodeunit测试应用中的同步和异步代码，还演示了如何与request模块结合，完整地测试JSON服务器提供的API。

学会了这些知识，我们可以结束这次奇妙的Node.js之旅了。我希望已经原汁原味地将Node.js非凡而有趣之处传递给你，并让你在读书的过程中，体验到Node平台的魅力。

请坐下来，开始编写代码。如果你有什么好的关于网站的点子，就开始动手实现它吧！如果你是移动端开发者，请想想如何将页面展现给移动用户，并且如何使用Node实现。即使你只是想学习一下服务器端脚本，也还是动手亲身体验下吧。因为想要拥有更好的编程技术的唯一途径就是使用这些技术。

如果编程过程中遇到了困难和问题，别忘记Node.js社区可是非常活跃和乐于助人的！在此如此多的热心积极的开发者中，你一定能找到志同道合的朋友，并得到所需的资源和帮助，来编写有趣实用的应用。