

*Broadview*

*[www.broadview.com.cn](http://www.broadview.com.cn)*

始于功能，终于代码，拿来即用

语法、规范、理念、工具、技巧，尽在超实用代码段中

onsole , info();

/buf.writeUInt8() usage ar buf8 = new Buffer(4); uf 8 a^£UInt8 ( 0x5m. 0 );

超实用的

Node.js代码段

**周敏编著**

基于实践和学以致用的原理，全是干货。涵盖控制台、模块、 包、异步I/O, Async流程控制、进程通信、TCP/UDP网络编 程、流、Web开发、MySQL、MongoDB .....

回中国工信出版集团



*Broadview'*博文视声如出版旗舰品牌

65356 技术凝聚实力.专业创新出版

**模块加载机制**

Node.js**的安装配置**

Node.js**的常用扁卞**

Node.js**的面试难点**

**模块与包•**

Buffer**模块**

0STO

**《超实用的**JavaScript**代码段》**

**《超实用的**jQuery**代码段》**

**《超实用的**CSS**代码段》**

**《超实用的**HTML**代码段》**

Node.js

MySQL

MongoDB

**服务器端编程**

JavaScript

代码热身

前端读书规划

涵盖技术

**近**100**个技巧提示**

**近**200**幅效果页面**

**近**300**段实用代码**

大散据

博文视点Broadview

责任编辑：董英  
封面设计：李玲

**文件系统模块**

**异步**I0 •

代码实歇

**进程通信•**

**数据交互•**

**异步机制**

Async**流程控制**

child process

Web**开发**

TCP/UDP**网络应用**

**进程管理**

***O****0新浪微博*

*weibo.com*

@博文视点Broadview

五星服务\*\*\*\*\*

**与**MySQL**的交互**

**与**MongoDB**的交互**

**微信**O**答案互动**

**微博。实时信息**

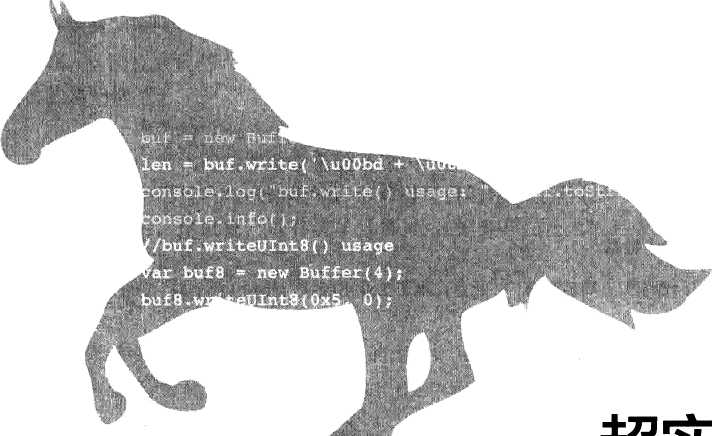
QQ**群**O 24**小时答疑**

Github • **实时代码**

**终生免费服务**

跨浏贸器、跨平台、跨设备





Nodejs代码段

**周敏编著**

**宙孑工甘出版孙**

Publishing House of Electronics Industry  
北京-BEIJING

**内容简介**

本书精选300余段Node.js代码，涵盖了服务器端脚本开发中的绝大多数要点、技巧与方法，堪称史 上最实用的Node.js框架开发方面的参考书籍，是网站建设与服务器端开发人员的好帮手。本书的代码跨 平台、跨设备、跨浏览器，充分向读者演示了如何使用Node.js框架的各项技术。

本书从Node.js框架的使用原理与应用场景出发，对最实用的Node.js代码段进行了全方位的介绍和演 示。全书分为15章，包含控制台、模块和包管理、异步I/O与Async流程控制库、Buffer,进程管理、*子* 进程通信、OS操作系统、文件系统、路径处理、TCP/UDP网络编程、流(Stream)、Web开发、常用工 具及MySQL与MongoDB数据库交互等Node.js框架技术的内容，对提高网站建设与服务器端开发人员的 Node.js技术水平有着非常重要的指导作用。

本书内容简洁明了、代码精练、重点突出、实例丰富、语言通俗易懂、原理清晰明白，是网站建设与 服务器端开发人员的良好选择，同时也非常适合大中专院校学生学习阅读。

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

**图书在版编目**(CIP )**数据**

超实用的Node.js代码段/周敏编著.一北京：电子工业出版社，2015.11

(代码逆袭)

ISBN 978-7-121-27431-2

I .①超…II.①周…IU.①JAVA语言一程序设计IV.①TP312

中国版本图书馆CIP数据核字(2015)第249419号

责任编辑：董英

印 刷：北京中新伟业印刷有限公司 装 订：北京中新伟业印刷有限公司 出版发行：电子工业出版社

北京市海淀区万寿路173信箱

邮编：100036

印张：22.75 字数：553千字

:787X 1092 1/16

本次次数  
开版印印

:2015年11月第1版

:2015年11月第1次印刷

:3000册 定价：59.00元

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

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

服务热线:(010) 88258888=

==

近些年Node.js框架作为一种服务器端脚本语言的开发技术，在IT圈内可谓是刮起了 一股热旋风。因为大家发现，原来仅仅运行于浏览器端的JavaScript脚本也可以运行在服 务器端，这简直太震撼人心了。于是，学习和掌握Node.js框架开发技术成了众多开发设计 人员的热切期望。

本书是一本讲解代码实践的书，它为读者全面深入地讲解了针对各种场景的Node.js 技术。300余段的代码给读者带来的不仅仅是全面的基础知识，更是为读者提供了设计简 洁高效的服务器端代码与网站架构、应对跨平台与跨浏览器兼容、优化服务器性能等切实 问题的解决之道。可以说，本书是学习Node.js框架开发技术的高效助手。

Node.js框架中的那些事儿

对于大多数刚刚接触Node.js框架的读者，可能一时无从下手，那么Node.js技术的难 点都有哪些方面呢？

,跨平台与浏览器的兼容

* 正确理解服务器端脚本使用

・模块和包管理

* 异步I/O与Async流程控制库

・进程与子进程通信

* 文件与路径处理

. Node.js事件处理机制

* TCP/UDP网络编程
* HTTP/HTTPS应用编程

. Node.js 与 AJAX 应用

* Node.js与数据库交互

以上所有内容在本书的代码中都有讲解，除了这些常见Node.js技术难点外，本书还力 求将最有用的Node.js代码段汇总在一起，提供各种解决实际问题的方案。

Node.js框架学习方法

11个字就能帮助我们更好地学习Node.js«

* **多看、多**练：观摩成功的网页设计，分析并练习网页设计中常用的代码。
* **多想、多问：**思考设计实现的原理，提出自己的问题并通过各种渠道来寻找答案。
* **多总结**：记录前人已经探索出来的Node.js技巧，总结实战中碰到的问题及解决方案。 只要真正能做到勤思考、勤动手、勤总结，Node.js学习定能一马平川。

本书的编写特点

* **独特的**Node.js**切入点**

与市面上其他Node.js相关的书不同，本书从最常见的场景应用角度出发，直接应 用Node.js代码段实现功能操作，全部是最实用的例子，全部是最透彻的分析！

* **内容丰富，知识全面**

本书以Node.js框架各个模块的场景应用作为基础，立体式、全方位地解释各种场 景下的Node.js代码段应用，场景丰富、实例丰富，并拥有良好的可扩展性、可复 用性。

**. 去中心化，分布式学习**

本书的代码实例都是独立的，读者可以从中间开始学，也可以从头开始学。代码 跨平台、跨设备，可以在平板电脑上学，也可以在PC上学，甚至，手机上写代 码也是完全可能的。

* Node.js**框架原理与实践相结合**

Node.js框架是一个强大的服务端脚本开发库，无论读者具有什么编程背景，都可 以通过它来优化、改进自己的服务器网站。本书的代码段基于简单易学的原理， 提供了丰富多样的操作特性，这使得本书成为了适用于各类脚本编程的必备工具。

**・涵盖内容全面**

本书的实例从控制台应用、模块和包管理、异步I/O与Async流程控制库、Buffer、 进程管理、子进程通信、OS操作系统、文件系统、路径处理、TCP/UDP网络编 程、流(Stream)、Web开发、常用工具及数据库交互等方面的内容出发，由简 入繁、点面结合、配合原理解释，给读者呈现了一场代码盛宴。

* **自发式学习**

在学习代码前，先让读者练习实际上最基础却最容易做错的Node.js框架代码和面 试题，激发读者的学习斗志。

本书的设计始于功能、终于代码，是IT设计人员的案头必备参考书。

本书的内容安排

本书共15章，各章节实现了不同类别的Node代码段。

**第**1**章 主要介绍了** Node.js**框架的控制台模块，**通过该模块的方法可以向操作系统 控制台实现各种格式化输入和输出等操作，也就是读者所熟知的“读取-求值-输出”循环

(Read-Eval-Print Loop,简称REPL)交互式的编程环境。

第2章 主要介绍了 Node.js框架自有的一套模块加载系统，通过该模块可以把各个 功能拆分、封装到不同的模块之中，在需要的时候调用该模块。Node.js框架使用模块和包 来组织管理，从性质及加载方式上可以分为以下几类内容：核心模块、文件模块、从文件 夹加载、文件夹作为模块和模块缓存。

第3章 主要介绍了 Node.js异步I/O编程，Node.js框架在设计之初就被考虑作为一 个高效的Web服务器而存在，因此高效的异步机制贯穿于整个Node.js框架的编程模型之 中。通过本章的介绍，读者可以学到异步I/O机制、异步I/O应用和Async流程控制库应 用等Node.js框架异步编程的内容。

第4章 主要介绍了 Node.js框架中Buffer的概念，它可以理解为是缓冲区或临时存 储区，是暂时存放输入、输出数据的一小块内存。如果读者学过C语言编程，对指针数组 的概念有一定的了解的话，那么学习和掌握Node.js框架的Buffer就会容易很多。

第5章 主要介绍了使用Node.js框架中功能强大的进程管理模块(Process)的方法。 Process模块是Node.js框架的一个全局内置对象,Node.js代码可以在任何位置访问该对象， 实际上这个对象就是Node.js代码宿主的操作系统进程对象。使用Process模块可以截获进 程的异常、退出等事件，可以获取进程的环境变量、当前目录、内存占用等信息，还可以 进行工作目录切换、进程退出等操作。

第6章 主要介绍了使用Node.js框架的child\_process模块创建子进程的四个方法， 分别是spawn。、exec()、execFile。和fbrk()。其中spawn()是最原始的创建子进程的方法，其 他三个都是通过对spawn。方法不同程度的进一步封装实现的。使用child\_process模块提供 的这些方法，可以实现多进程任务、操作shell和进程通信等操作，实用功能是非常强大的。

第7章 主要介绍了 Node.js框架中的操作系统(OS)模块，该模块提供了一系列与 操作系统相关的函数方法，不过这些方法相对简单，实现的功能也十分有限。

第8章 主要介绍了 Node.js框架中的文件系统(File System)模块如何来支持I/O 操作的方法，这些操作方法是对标准POSIX函数的简单封装，它提供了文件的读取、写入、 更名、删除、遍历目录、链接等POSIX文件系统操作。

第9章 主要介绍了 Node.js框架中的路径处理(Path)模块、url路径处理(url)模 块以及字符串解析(Query String)模块，这些模块提供了一系列与路径解析处理相关的函 数方法，这些方法对于处理常规的需求是足够的。

第10章 主要介绍了 Node.js框架中对TCP/UDP网络编程方面的支持，Node.js框 架为设计人员提供了网络(Net)模块来支持TCP协议应用，数据报套接字(UDP)模块 来支持UDP协议应用，这两个模块提供了一系列与网络应用相关的函数方法，通过这些方 法可以构建基本的网络应用。

第11章 主要介绍了 Node.js框架中的抽象接口流(Stream)模块，流(Stream)模 块操作最主要的是.pipe()方法，它可以为开发者提供可以重复使用的统一的接口，通过抽象 的流(Stream)接口来控制流(Stream)之间的读写平衡。

第12章 主要介绍了应用Node.js框架中的HTTP模块与HTTPS模块开发Web应 用的方法，这两个模块基于HTTP协议与HTTPS协议开发，提供了一系列与Web应用开 发相关的函数方法，通过这些方法可以构建各种功能复杂且强大的Web应用。

**第**13**章 主要介绍了** Node.js**框架中的常用工具**（Util）**模块，**该模块是为了解决核 **心**JavaScript的功能过于精简而设计的。该模块可以实现对一个原型对象的继承功能，实 现对象格式化操作、将任意对象转换为字符串操作、调试输出功能、正则表达式验证等。

**第**14**章 主要介绍了** Node.js**框架与**MySQL**数据库交互的方法，**主要选用的是目前 人气最高的node-mysql开源项目作为Node.js框架MySQL扩展库，该开源项目提供了 MySQL数据库对Node.js框架的完整支持，具有一套与数据库开发相关的函数方法，通过 这些方法就可以非常方便地构建Node.js数据库应用。

**第**15**章 主要介绍了** Node.js**框架与**MongoDB**数据库交互的方法，**主要选用的是同 名的mongodb开源项目作为Node.js框架的MongoDB扩展库，该扩展库具有一套与数据库 开发相关的函数方法，通过这些方法就可以非常方便地构建Node.js数据库应用。

本书面对的读者

* Web服务器设计入门者
* Web服务器开发入门者

.Node.js框架学习爱好者

. 由JavaScript向Node.js框架转型的开发人员

* 中小型企业网站开发者

・大中专院校的学生

* 各种IT培训学校的学生
* 网站后台开发人员
* 网站建设与网页设计的相关威客兼职人员

本书的服务

笔者能力有限，如果读者发现我们在写作过程中有什么疏漏，或者您对本书有什么疑 问，可通过以下方式与我们沟通。

* QQ群：296811675,作者在线答疑。

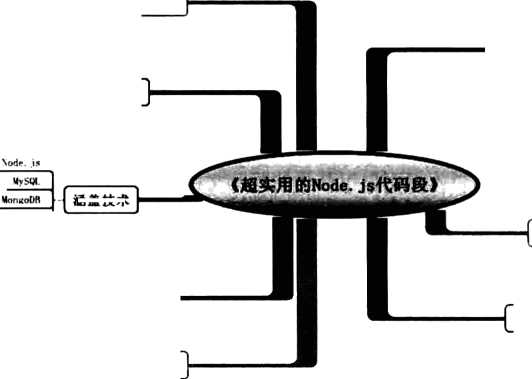
・扫描封底的微信二维码，时刻参与我们的图书互动。

・通过新浪微博@博文视点Broadview, 了解我们发布的信息和各种前端流行技术。

. 博文视点官方网站[http://www.broadview.com.cn/,下载本书所有实例源代码。](http://www.broadview.com.cn/,%e4%b8%8b%e8%bd%bd%e6%9c%ac%e4%b9%a6%e6%89%80%e6%9c%89%e5%ae%9e%e4%be%8b%e6%ba%90%e4%bb%a3%e7%a0%81%e3%80%82)

* Github, <https://github.com/kingwjz/Node.js-code>, 了解代码的实时更新和迭代过程, 可以在每章代码下参与讨论，也可以观看其他读者提出的问题，还可以随时随地 下载代码。

很多读者在学习过程中苦于无法交流，小故障无法及时解决，加入我们的服务方阵， 我们将为您提供终生免费的服务。一本书、一段情、一辈子。



本书的思维导图

1代码热身］

《超实用的,RvMrripi代码段，

前端读书规划

涌盖技术

五星服务\*\*\*\*\*

近100个技巧推京

1大数据］■

近20。幅效果贝向

跨浏览器、跨平台、跨设备

300余段实用代码

原生、混合、移动网页

服务IS•编巧

.[MvaSrr ipl

\<xh・.J5的安装R2置

、M,，. js的审川代码

Nd，, js的而试庵六

《超实用的.iQuerv代码段\* e超实用的css代码段》 《超实用的H1HL代码段）

| 眼块包  异切**0** -匚 | 模块加栽**HI**■剌  **Buffer**模块  **OS**模块  异步机捌 **AsvncififtffM** |
| --- | --- |
|  | **child.process** |
| 进毋域信- | **Web**开发  **TOPEDP**网常应用 |
| 散据交厅- | 进界管理  **^MvSUL**的交 **0** |
| 马”明。**1»**的女坯 |

代码实战

| 微信- | 答案五动 |
| --- | --- |
|  | 实时信•& |
| **QQ**群- | **24**小时答登 |
| **Github** | 安时代码 |
| 为作免林服务 | |

编者推荐

本书摒弃传统的说教模式，每段代码都是单独的功能型页面，读者可以从全书的任意 一点开始线性阅读。本书的目的就是将最有用的代码与读者分享，包含了服务器设计人员 在实战中必须具备的所有技巧和方法，读者可以拿来就用。本书300余段代码也许并不是 最优的代码，但笔者提供了 Github地址，与全世界IT工程师一起优化了这些代码，并实 现了更新迭代，以保证读者始终能看到最好的、最高效的、最实用的Node代码段。这是 一本市场上绝无仅有的Node实战书，是一本值得拥有的Node设计书。

序1 Node.js的前世今生

这是一本是有显著特点的Node.js书，它涉及了脚本编程的技巧、脚本运行性能的优化、 快速开发服务器的应用。所以本书不仅仅是你的帮助文档，更是你的良师益友，拥有这些 代码，你将会在服务器脚本工作中战无不胜！

Node.js框架概述

Node.js是一个基于Google公司Chrome浏览器的JavaScript运行时建立的平台，主要 用于搭建响应速度快、方便扩展的服务器Web应用。其实，Node本身就是一个JavaScript 运行环境，是对Google V8引擎的完美封装。Google V8引擎执行JavaScript脚本的速度非 常快、性能非常好，Node.js框架对它进行了优化，提供了相应替代的API,保证V8引擎 在非浏览器环境下运行得更好。

Node.js框架基于事件驱动、非阻塞I/O模型，因而得以轻量和高效，非常适合在分布 式设备终端上运行数据密集型的实时应用。Node.js框架采用一套"非阻塞” I/O库来支持 事件循环的方式，更好地为文件系统、数据库之类的资源提供了接口。目前，由于Web服 务器与富客户端浏览器应用程序之间共享代码的方法只能通过JavaScript脚本语言来实现, 因此对服务器脚本语言支持最完美的Node.js框架发展十分迅猛，实际上己经成为了服务器 端的JavaScript脚本语言开发平台。

更为重要的是，在Node.js框架发布以来的很短时间内，各大开源社区就已经贡献了大 量的扩展库(模块)，提供了对各种开源软件或数据库驱动的支持。因此，虽然Node项 目还非常年轻，但众多的软件开发人员都对Node项目展现出了极大的热情，并贡献出了 自己的一份力量，改进并完善了 Node.js框架。

Node.js框架环境搭建

学习Node.js框架的第一步，需要先搭建好其工作环境，这样才能运行并测试Node脚 本文件。下面具体介绍一下。

(1)获取Node安装包，可以访问以下Node.js框架官方网址：

<https://nodejs.org/download/>

在该页面中，可以看到适用于不同操作系统的、各个版本的Node安装包源文件，选 中我们需要的版本类型直接下载就可以。

在Windows操作系统环境下安装Node.js框架，直接运行下载好的Node安装包源文件 即可，目的路径一般选择如下：

D:/nodejs/

安装完毕后，读者可以到该目录下浏览一下具体内容，这样会对Node环境有一个大 致的了解。

(2)在Ubuntu (Linux)环境下需要运行apt命令进行安装，具体方法如下：

sudo apt-get install nodejs

如果想获取最新的Node安装源，具体方法如下：

sudo add-apt-repository ppa:chris-lea/node.js

sudo apt-get update

sudo apt-get install nodejs

一般情况下，node命令会被自动安装在以下路径之中(视不同的Ubuntu系统版本而 定)：

/usr/local/bin/node

大致经过以上这几步的操作，Node.js框架就安装完毕了，为了检验Node环境是否正 确，可以使用以下命令来检测Node的版本：

node -v

如果显示出正确的版本号，则说明Node环境搭建成功了。

Node.js框架推荐工具

1. **开发工具，平台类**
2. jetBrains公司的WebStorm开发工具

WebStorm是jetBrains公司开发的一款JavaScript开发工具，被众多JavaScript开发者 作为首选的开发平台，完全支持Windows. Linux、Mac OS X等操作系统，有“Web前端 开发神器”的美誉。读者可以从下面的网址获取该工具：

<http://www.jetbrains.com/webstorm/>

1. Sublime Text代码编辑器

Sublime Text 是一款代码编辑器(支持 C、Java、JavaScript、HTML、CSS 等)，同时 也可以作为文本编辑器使用。Sublime Text具有漂亮的用户界面和强大的扩展功能，比如 支持代码缩略图、各种工具插件、多窗口任务等功能。同时，Sublime Text也是一款跨平 台的编辑器，完全支持Windows. Linux、Mac OS X等操作系统。读者可以从下面的网址 获取该工具：

<http://www.sublimetext.com/>

1. EclipseNode.js IDE 插件

对于已经习惯了使用Eclipse开发工具的读者来讲，可以使用Eclipse Node.js IDE这款 插件来开发Node程序。读者可以从下面的网址获取该插件工具：

<http://www.nodeclipse.org/>

1. Node.js Tools for Visual Studio 插件

如果读者习惯使用微软公司的Visual Studio系列开发工具，同样有一款Node.js Tools for Visual Studio插件可以提供对开发Node程序的支持。读者可以从下面的网址获取该插 件工具：

<https://nodejstools.codeplex.com>

1. **测试调试工具类**
2. NodeUnit

NodeUnit是一款应用于Node.js的单元测试框架，基于assert模块开发，读者可以从下 面的网址获取该测试工具：

<https://github.com/caolan/nodeunit>

1. Supervisor

如果希望在Node代码修改后立即看到效果，而不是每次都要重启Node服务才能看到 效果，Supervisor调试工具可以帮助实现这个功能，它会监视代码的改动并自动重启Node 服务。读者可以从下面的网址获取该测试工具：

<http://supervisord.org>

1. Mocha

Mocha是一款完整的测试工具，可安装在NPM和CI服务器之上。读者可以从下面的 网址获取该测试工具：

<https://github.com/mochajs/mocha>

1. should.js

should.js模块是assert模块的扩展，其语法与日常用的语法几乎一模一样，非常易于使 用。读者可以从下面的网址获取该测试工具：

<https://github.com/visionmedia/should.js>

1. **第三方开发包类**

(1) Async异步处理

Async是一个流程控制工具包，提供了直接而强大的异步功能，该插件基于JavaScript 语言专门为Node.js设计，同时也可以直接在浏览器中使用。读者可以从下面的网址获取该 开发工具：

<https://github.com/caolan/async>

1. Express 开发包

Express是一个简洁而灵活的Node.js Web应用框架，提供一系列强大特性，帮助我们 创建各种Web应用。Express不对Node.js已有的特性进行二次抽象，只是在其之上扩展了 Web应用所需的功能。Express开发包所具有的丰富的HTTP工具以及来自Connect框架的 中间件可以为审计人员随取随用，并创建出友好的、快速且简单的API。读者可以从下面 的网址获取该开发工具：

<https://www.npmjs.com/package/express>

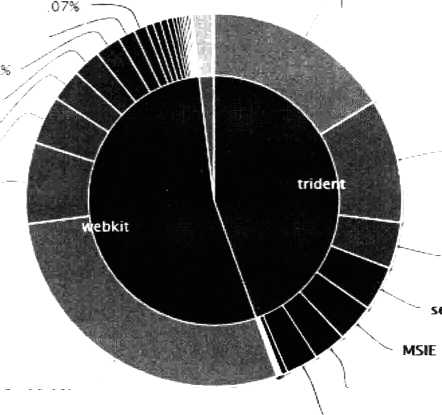
1. node-mysql 开发包

node-mysq开发包是一个纯Node.js的用JavaScript实现的MySQL客户端程序。 node-mysql封装了 Node.js框架对MySQL的基本操作，拥有100%的MIT公共许可证。读 者可以从下面的网址获取该开发工具：

<https://github.com/felixge/node-mysql>

本书浏览器约定

本书涉及的浏览器测试基准如图1所示，可以看到内核和外壳的占有情况，内环为内 核，外环为外壳，这也是接下来要提到的浏览器同类项的数据基础。



firefox: 1 84%

MSIE 8 0: 16.17%

360sei 10.71%

chrome 29.0: 1

chrome 28.01 1.47%

Ibbfowser 210 2

chrome 1 *7.0.* 2.42%

chrome 26.0: *2.77%*

chroma 21.0: *28* 39%

MSIE 9 0: 4 01%

chrome 20.0: 4 19%

MSIE 6 Or 2.81%

chrome 24.0: 7.02%

MSIE 10 0: 2.79%

图1浏览器基准

不同的浏览器可能会采用相同的内核（渲染引擎）。一方面，外壳实现的差异性会影 响整个网站功能，这也是为什么内核无法完全代表所有浏览器的原因。另一方面，某些浏 览器具有相同的血统，是可以归为同一类的，同一类中，如果测试通过其中一个，那么其 他与之同类的浏览器也基本上可以通过测试。图1是根据相同的内核或对W3C标准有类似 的支持程度对浏览器进行归类的。之所以这样做，是因为测试基准除了要基本反映浏览器 的市场占有率，还要考虑到开发测试的成本。因此，测试基准中的浏览器应当具有典型代 表性。

浏览器的不同版本的内核也不一样，因此通常要针对不同版本的浏览器做测试，开发 者要了解内核对标准的支持，比如IE的渲染引擎Trident的不同版本差别较大，因此IE需 要测试6〜10的所有版本，然而由于Firefox和Chrome升级覆盖面广，因此基准中只保留 其最新版本。

本书涉及的代码会根据需要，对不同浏览器进行有针对性的测试。

序2最简单却必须看的Node代码

Node代码基于JavaScript代码，却也有所不同，相信通过对本书的学习，读者会对Node 技术爱不释手！下面介绍一些很简单、却又很有特点的Node代码。

1. **控制台运行**Node**代码的方法：**

01 $ node

02 > console.logfHello Node!');

03 Hello Node!

第01行代码运行Node环境；第02行代码调用Node.js框架的打印输出命令；第03 行代码为控制台打印输出的结果。

1. **控制台运行**Node**脚本的方法：**

01 $ node hello.js

02 Hello Node!

第01行代码使用Node命令运行hello.js脚本文件(脚本文件中代码同第1个例子中的 第02行代码)；第02行代码为控制台打印输出的结果。

1. **使用**require**命令加载模块的方法：**

01 varhttp = requireChttp'); //TODO:加载核心模块'http'

02 var circle = require('./ch02.module-file-circle.js,);

第01行代码使用require命令加载核心模块'http'；第02行代码使用require命令加 载自定义模 ^i'./ch02.module-file-circle.js'o

1. **使用**exports**命令输出对象的方法：**

01 exports.area = function(r) (

02 return PI\*r\*G

03 };

这段代码使用exports命令输出一个area方法，用于计算圆的面积。

1. **使用**async**流程库的方法：**

01 async.series([

02 fundion(callback) (

03 callback(null, 'hello');

04 ),

05 function(callback) (

06 callback(null, 'async');

07 ),

08 function(callback) (

09 callback(null, 'series'); r，

1. )
2. ],function(err, results) {
3. console.log(results);
4. })；

这段代码使用了 async流程库的series方法来控制流程，async流程库还有其他几种控 制方法，读者可以阅读本书的详细内容。

1. I/O**异常捕获的方法：**

01 try(

02 setTimeout(function(X

03 var data = a/0; //TODO:错误的计算

04 },1000);

05 Jcatch (eX

06 console.log(e);

07 }

这段代码演示了使用try/catch语法进行I/O异常捕获的方法，第03行代码输出了一个 错误的计算，第06行代码输出捕获的错误异常。

1. **初始化**Buffer**对象的方法：**

01 var buffer = new Buffer("This is Buffer", "utfS"); *H* TODO:初始化 Buffer

这段代码演示了初始化Buffer对象的方法，并定义了，tf8”编码格式。

1. **获取进程信息的方法：**

01 console.info('当前进程 id:');

02 console.info(process.pid);

03 console.infof 当前进程名称:')；

04 console.info(process.title);

这段代码演示了通过process模块获取进程id和进程名称的方法。

1. **创建子进程的方法：**

01 var spawn = require('child\_process').spawn; // TODO:弓I 入 child\_process 模块

02 var ls\_var = spawnCIs', 7varl); *II*TODO:定义命令行'Is -Ih /var1

这段代码演示了通过child\_process模块创建子进程的方法，并实现了查看/var目录的 命令。

10.

**获取操作系统信息的方法:**

01

02

03

04

01

02

03

04

05

06

if(fs.existsSync(file\_path)) {

var file\_contents = fs.readFileSync(file\_path, 'utf-8'); console.info(\*read txt/readFileSync.txt contents:'); console.info(file\_contents);

console.info();

// TODO:

//TODO:

读文件(同步方式)

打印输出文件内容

var type = os.type();

console.infof当前操作系统类型为：1 + type); var platform = os.platform();

console.info(，当前操作系统平台为：'+ platform);

这段代码演示了通过操作系统OS模块获取操作系统类型和平台信息的方法。

**读文件的方法:**

这段代码演示了通过文件系统fe模块读文件的方法。

12.

**规范化字符串路径的方法:**

01

02

03

//TODO:

var path = require('path');

var path\_a = "/home//king";

console.infofpath.normalize(7home//king,') is :' + path.normalize(path\_a));

引入路径处理模块

这段代码演示了通过路径path模块规范化字符串路径的方法。

13.

**创建**TCP**服务器的方法:**

01

02

03

04

05

06

07

08

//TODO:引入网络(Net)模块

//TODO:定义服务器地址

//TODO:定义端口号

var net = require('net');

var HOST = ,127.0.0.1,; var PORT = 9696;

net.createServer(function(sock) ( /\*\*

\*添加事件处理方法

7

)).listen(PORTs HOST);

这段代码演示了通过网络net模块创建TCP服务器的方法。

14.**创建**HTTP**服务器的方法：**

01 var http = require('http'); // TODO:引入 http 模块

02 http.createServer(function(req, res) {

03 /\*\*

04 \*通过res.writeHeader()函数方法写HTTP文件头

05 \*/

06 res.writeHead(200, {'Content-type': 'text/html\*});

07 /\*\*

08 \*通过res.write()函数方法写页面内容

09 \*/

1. res.writeC<h3>Node.js — HTTP〈/h3>');
2. /\*\*
3. \*通过res.end()函数方法发送响应状态码，并通知服务器消息完成
4. \*/
5. res.endf<p>Create Basic HTTP Server!v/p>，)；
6. )).listen(6868); Z/TODO:监听 6868 端口号

这段代码演示了通过HTTP模块创建Web服务器的方法。

15. Node.js**框架连接**MySQL**数据库的方法:**

|  |  |  |  |
| --- | --- | --- | --- |
| 01 | var http = require("http"); // TODO:引入 HTTP 模块 | | |
| 02 | var mysql = require(7usr/local/lib/node\_modules/mysql'); //TODO:引入 mysql 模块 | | |
| 03 | var connection = mysql.createConnection(( | |  |
| 04 | host: "localhost", | //TODO: | 主机地址 |
| 05 | user: "root", | //TODO: | 数据库用户名 |
| 06 | password: “root”， | //TODO: | 数据库密码 |
| 07 | database: "nodejs", | //TODO: | 数据库名称 |
| 08 | port: 3306 | //TODO: | 端口号 |
| 09 | }); |  |  |

10 http.createServer(function (req, res) (

1. res.writeHead(200, ("Content-Type": "text/html;charset=utf8n});
2. res.write("<h3>测试 Node.js - MySQL 数据库连g!</h3><br/>");
3. connection.connect(function(err) (
4. if(err) (
5. res.end(\*<p>Error Connected to MySQL!v/p>');
6. return;
7. } else (
8. res.end(\*<p>Connected to MySQL!</p>');
9. }
10. ))；
11. )).listen(6868); *II*TODO:监听 6868 端口号

这段代码演示了 Node.js框架连接MySQL数据库的方法。

序3 Node.js最流行的面试题•

面试题一般不是多么高精尖的问题，在世界500强和中国100强的科技公司中，绝 大多数的考题都是基础技术类知识，原因是他们招聘的不是经理，而是具体干活的，所 以要考查的就是基本功。面试题不一定有统一的标准答案，读者可以自己测试。

1. 请简要描述一下Node.js框架的概念。

【参考答案】Node.js是一个基于Chrome JavaScript运行时建立的平台，用于方便地搭 建响应速度快、易于扩展的网络应用。Node.js框架使用事件驱动、非阻塞I/O模型而得以 轻量和高效，非常适合在分布式设备上运行数据密集型的实时应用。Node实际上是一个 JavaScript运行环境，对Google V8引擎进行了封装，因为V8引擎执行JavaScript的速度 非常快，性能非常好。

1. 请简要说明几个Node.js框架的优点。

【参考答案】

* RESTful API：支持Web服务和动态Web应用程序的多层架构，实现可重用性、 简单性、可扩展性和组件可响应性的清晰分离。开发人员可以轻松使用Ajax和 RESTful Web服务一起创建丰富的界面。
* 单线程：Node.js单线程是指Node并没有创建一个线程的能力，所有代码都是单 线程执行的。不过Node宿主环境并不是单线程的，它会维护一个执行队列，循 环检测并调度JavaScript线程来执行，因此单线程执行和并发操作并不冲突。
* 事件轮询机制：Node.js可以在不新增额外线程的情况下，依然可以对任务进行并 行处理，它是通过事件轮询(event loop)来实现并行操作的。
* 非阻塞I/O：由于Node.js是事件驱动的，因此其使用了事件循环来解决I/O操作 带来的瓶颈问题。在Node.js中，一个I/O操作通常会带有一个回调函数，当1/0 操作完成并返回时，就会调用这个回调函数，而主线程则继续执行接下来的代码。

. V8虚拟机：Node.js是一个基于Google Chrome V8 JavaScript引擎之上的平台，可

用以创建轻量级、快速、可扩展、事件驱动和非阻塞I/O的应用。

* 事件驱动：Node.js框架使用事件驱动模型，即当Web Server接收到请求时，就将 其关闭然后进行处理，然后去服务下一个Web请求。当这个请求完成后，它被放 回处理队列中，当到达队列开头时，这个结果被返回给客户端。

3.**简述**Node.js**框架适用的应用场景。**

【参考答案】

1. 面向服务的架构

面向服务的架构就是做好前、后端的依赖分离，将所有的关键业务逻辑都封装成REST 调用，上层只需要考虑如何用这些REST接口来构建具体的应用。这样，后台程序员们不 需要知道具体数据是如何从一个页面传递到另一个页面的，也不需要知道用户数据更新是 通过Ajax异步获取的还是通过刷新页面获取的。Node.js正适合这样的开发场景。

1. RESTful API

前面，我们讲到了 RESTftdAPl是Node.js框架的主要优点，因此也是其最理想的应用 场景。应用RESTftd API场景可以处理数万条连接请求，该操作没有太复杂的逻辑，仅仅 就是请求APL将数据进行返回即可。简而言之，其本质就是从数据库中查找一些值并将 其组成一个响应，由于这类响应是很小的文本，同时连接请求也是很小的文本，因此整体 流量不高，一台服务器往往也可以处理很繁忙的连接请求。

1. Ajax请求应用

目前，个性化Web应用十分流行，大数据时代针对个人用户页面的定制信息已经成为 主流，当缓存失效后需要在页面加载时发起Ajax请求，此时应用Node.js框架可以响应大 量的并发请求。因此Node.js框架适合运用在高并发、I/O密集、少量业务逻辑情况下的Ajax 请求应用场景。

1. **简述**Node.js**框架不适用的应用场景。**

【参考答案】

当然，Node.js框架也不是万能皆适的，有些应用场景也不是它的强项。下面可以列举 几种供读者参考。

1. 实时性要求很高的场景，例如工程交换机、工控机器人和DCS集控系统等。这 些场景基本通过垃圾回收机制来管理系统内存，因此Node.js框架将会影响响应速度，并且 很难进行优化。
2. 计算密集型系统，这些应用基本都是C语言的天下，基于JavaScript语言的Node.js 框架在计算性能上是很难与C语言代码相比的。
3. 需要单一进程控制大内存的场景，由于Google V8引擎的设计限制，在32-bit T 有1GB最大内存尺寸的限制，在64-bit T是1.7GB。虽然Node.js框架的Buffer分配可以 不超过此限制，但是也会带来垃圾回收机制上性能的退化。
4. **谈谈**Node**异步机制的种类。**

【参考答案】异步机制方法大致包括回调函数、pub/sub模式(事件模式)、异步库控 制库(async、when 等)、promise 项目、Generator 项目等。

1. **简述** exports **与** module.exports **的区别。**

【参考答案】需要明确的是，module.exports才是真正的接口，而exports只不过是 module.exports的一个辅助工具，exports是基于module.exports而实现的。实际上，全部由 exports获取的属性和方法，最后都赋给了 module.exports接口，不过有个前提就是 module.exports本身不具备任何属性和方法。换言之，如果module.exports接口已经具备一 些属性和方法，那么exports获取的属性和方法将被忽略。

目 录

[**第**1**章** Node.js**控制台模块应用** 1](#bookmark208)

1. [Node.js 版之“Hello World”程序 1](#bookmark214)
2. [日志输出与信息输出功能 3](#bookmark223)

[1.3字符串格式化的应用 5](#bookmark226)

1.4数值型格式化的应用 6

[1.5 JSON数据交换格式应用 7](#bookmark230)

[1.6测试特殊运算符如何取值 8](#bookmark234)

[1.7特殊格式化参数应用 9](#bookmark238)

[1.8 斐波那契(Fibonacci)数列应用 10](#bookmark244)

[1.9错误与警告提示应用 11](#bookmark258)

1. [重定向标准错误输出流 12](#bookmark261)
2. [查看Node.js对象的属性和方法 13](#bookmark264)

1.12时间计时器应用 14

[1.13查看当前调用栈的方法 16](#bookmark272)

[1.14对表达式结果进行评估 17](#bookmark285)

1.15读取命令行输入信息 18

1. [阶乘运算输出 20](#bookmark293)
2. [向控制台输出组合控制键 22](#bookmark325)

[1.18模拟一个简单的控制台界面 23](#bookmark341)

[1.19输出水仙花数 26](#bookmark344)

[1.20 输出质数 27](#bookmark347)

[1.21打印输出杨辉三角 28](#bookmark376)

[1.22 汉诺塔(Hanoi)问题 30](#bookmark400)

[1.23简单四则运算应用 33](#bookmark445)

[**第**2**章** Node.js**模块与包管理** 36](#bookmark496)

1. [Node.js框架加载核心模块 36](#bookmark502)
2. [Node.js框架加载文件模块 37](#bookmark506)
3. [从 node modules 文件夹中加载 39](#bookmark519)

[2.4模块的循环调用问题 40](#bookmark523)

1. [module.exports 对象应用 42](#bookmark526)
2. [module.exports 对象与 exports 对象 43](#bookmark530)

[2.7汇率换算实用程序 45](#bookmark536)

[2.8简单计数器实用程序 46](#bookmark552)

1. [包管理之 package.]son 详解 49](#bookmark598)

[2.10基于包管理的四则运算应用 52](#bookmark635)

[**第**3**章** Node.js**异步**I/O**与**Async**流程控制库** 58](#bookmark653)

1. [Node.js框架异步机制基础 58](#bookmark660)
2. [Node.js框架异常捕获机制 60](#bookmark709)
3. [Async串行流程控制 62](#bookmark725)
4. [Async瀑布模式流程控制 64](#bookmark761)
5. [Async并行流程控制 66](#bookmark785)
6. [Async限制性并行流程控制 67](#bookmark805)
7. [Async循环流程控制 69](#bookmark809)
8. [Async 队列流程控制 72](#bookmark843)

[**第**4**章** Buffer**模块处理** 75](#bookmark920)

1. [定义Buffer对象的基本方法 75](#bookmark926)
2. [定义Buffer对象的特别方法 77](#bookmark948)
3. [如何判断Buffer对象 79](#bookmark960)
4. [获取Buffer对象字节长度 80](#bookmark982)
5. [读取Buffer对象 82](#bookmark987)
6. [写入Buffer对象 84](#bookmark1042)
7. [Buffer对象转字符串 86](#bookmark1082)
8. [Buffer对象裁剪 88](#bookmark1103)
9. [拷贝Buffer对象 89](#bookmark1119)
10. [正确拼接Buffer 91](#bookmark1155)
11. [应用 Buffer 操作 HTTP Request Header 93](#bookmark1189)

[**第**5**章** Node.js**进程管理** 98](#bookmark1253)

[5.1获取应用程序当前目录 98](#bookmark1259)

1. [改变应用程序目录 99](#bookmark1263)
2. [获取系统相关信息 100](#bookmark1268)

[5.4实现标准输出流 105](#bookmark1304)

[5.5实现标准错误流 107](#bookmark1290)

1. [实现标准输入流 108](#bookmark1281)
2. [Kill 当前进程 : 109](#bookmark1323)
3. [Process模块的异步方法 111](#bookmark1336)

[**第**6**章** child\_process**及进程通信** 113](#bookmark1366)

1. [使用spawn方法创建子进程 113](#bookmark1371)
2. [使用exec方法创建子进程 115](#bookmark1387)
3. [使用spawn方法绑定系统事件 117](#bookmark1402)
4. [使用exec方法绑定错误事件 118](#bookmark1421)

[6.5创建子进程查看用户目录 120](#bookmark1462)

[6.6查看用户目录出现错误的处理方法 122](#bookmark1487)

[6.7查看物理内存使用状态 123](#bookmark1512)

1. [查看子进程pid的方法 125](#bookmark1538)

[6.9创建子进程统计系统登录次数 127](#bookmark1563)

1. [使用execFile方法创建子进程 128](#bookmark1581)
2. [使用execFile方法绑定错误事件 130](#bookmark1595)
3. [使用fork方法获取CPU信息 131](#bookmark1602)
4. [使用fork方法实现进程间通信 132](#bookmark1614)

[6.14使用fork方法实现斐波那契数列 134](#bookmark1630)

[**第**7**章**OS**模块应用** 138](#bookmark1662)

[7.1获取当前操作系统类型与平台 138](#bookmark1667)

[7.2获取操作系统默认的临时文件目录 139](#bookmark1678)

[7.3获取操作系统主机名 140](#bookmark1682)

[7.4获取操作系统架构 141](#bookmark1687)

[7.5获取操作系统的发行版本 142](#bookmark1695)

[7.6返回操作系统运行时间 143](#bookmark1705)

[7.7获取操作系统内存状态 145](#bookmark1729)

1. [获取CPU内核信息 146](#bookmark1739)

[7.9获取网络接口信息 147](#bookmark1743)

[**第**8**章 文件系统模块应用** 149](#bookmark1747)

[8.1 重命名文件 149](#bookmark1752)

* 1. [打开与关闭文件 152](#bookmark1800)
  2. [截取文件内容 155](#bookmark1819)
  3. [修改文件长度 156](#bookmark1836)
  4. [获取文件信息 158](#bookmark1866)

[8.6更改文件所有权 159](#bookmark1872)

[8.7更改文件权限 161](#bookmark1875)

[8.8创建与删除文件硬链接 163](#bookmark1921)

[8.9创建文件符号链接 165](#bookmark1960)

[8.10读取文件符号链接 168](#bookmark1966)

[8.11相对路径转绝对路径 169](#bookmark1975)

1. [创建与删除目录 170](#bookmark1981)
2. [读取文件目录 172](#bookmark2020)
3. [读文件的应用 173](#bookmark2025)
4. [功能增强的读文件应用 175](#bookmark2037)
5. [写文件的应用 176](#bookmark2072)

[8.17功能增强的写文件应用 178](#bookmark2092)

1. [追加写入文件 180](#bookmark2132)
2. [监控文件的应用 182](#bookmark2160)

[**第**9**章 路径处理应用** 185](#bookmark2179)

1. [规范化字符串路径 185](#bookmark2184)

[9.2合并字符串路径 187](#bookmark2218)

9.3 解析绝对路径 188

[9.4解析相对路径 189](#bookmark2245)

1. [获取文件夹路径 191](#bookmark2263)

[9.6获取路径中文件扩展名 192](#bookmark2276)

[9.7如何获取路径中最后部分 193](#bookmark2290)

1. [解析与格式化url路径 194](#bookmark2325)
2. [url路径转化 196](#bookmark2344)
3. [url参数转化url对象 197](#bookmark2357)

[**第**10**章** TCP/UDP**网络应用** 199](#bookmark2376)

* 1. [创建基本的TCP服务器 199](#bookmark2379)
  2. [创建基本的TCP客户端 201](#bookmark2409)
  3. [创建简单的TCP通信应用 202](#bookmark2432)

[10.4 创建TCP服务器的另一种方式 206](#bookmark2517)

[10.5服务器端绑定事件 207](#bookmark2525)

[10.6获取服务器地址参数 210](#bookmark2565)

[10.7获取当前服务器连接数 211](#bookmark2577)

[10.8获取套接字地址 215](#bookmark2609)

[10.9 获取远程地址 216](#bookmark2624)

[10.10使用套接字写数据 218](#bookmark2651)

10.11控制套接字数据流的应用 221

1. [创建UDP服务器 225](#bookmark2757)
2. [创建UDP客户端 227](#bookmark2764)
3. [创建简单的UDP应用 229](#bookmark2797)
4. [UDP广播服务的实现 233](#bookmark2863)

[10.16模仿简单的聊天室应用 236](#bookmark2975)

[**第** 11 **章** Node.js **流**（Stream ）**应用** 241](#bookmark3017)

1. [仓Li建可读 （Readable） 流 241](#bookmark3020)
2. [测试可读readable事件处理 244](#bookmark3063)
3. [使用data事件读取文件 246](#bookmark3097)
4. [可读流错误事件处理 247](#bookmark3124)
5. [暂停与恢复可读（Readable）流 248](#bookmark3142)
6. [使用可读（Readable）流发送数据 251](#bookmark3160)
7. [使用可读（Readable）流读取文件 252](#bookmark3164)
8. [使用pipe方法实现复制文件 253](#bookmark3191)
9. [解除pipe方法设置的流 255](#bookmark3203)
10. [使用可写（Writable）流写文件 257](#bookmark3228)
11. [可写（Writable）流 drain 事件处理 258](#bookmark3243)
12. [可写（Writable）流 finish 事件处理 260](#bookmark3247)

[**第** 12 **章** Node.js Web**开发** 263](#bookmark3258)

* 1. [构建一个基本的HTTP服务器 263](#bookmark3263)
  2. [编写一个简单的HTTP客户端 265](#bookmark3281)
  3. [HTTP响应状态码 268](#bookmark3315)
  4. [设定和获取HTTP头文件 271](#bookmark3378)
  5. [写HTTP头文件的方法 273](#bookmark3405)
  6. [发送与处理GET请求 276](#bookmark3430)

[12.7进行重定向操作 278](#bookmark3487)

[12.8服务器多路径处理方式 281](#bookmark3518)

1. [模拟ping命令连接服务器 284](#bookmark3550)
2. [安装 Express开发框架 286](#bookmark3595)
3. [使用Express开发框架开发的Hello World 287](#bookmark3599)
4. [Express开发框架路由处理 292](#bookmark3655)
5. [应用Express框架实现登录页面 296](#bookmark3676)
6. [Express框架实现Ajax方式操作 301](#bookmark3733)

[**第**13**章** Node.js Util**常用工具** 306](#bookmark3807)

1. [原型对象继承 306](#bookmark3813)

[13.2将任意对象转换为字符串 308](#bookmark3817)

[13.3验证是否为数组 310](#bookmark3834)

[13.4验证是否为日期格式 310](#bookmark3838)

[13.5 验证是否为正则表达式 311](#bookmark3841)

[13.6验证是否为错误类型 312](#bookmark3844)

[13.7格式化字符串 313](#bookmark3850)

[**第** 14 **章** Node.js **与** MySQL **交互** 315](#bookmark3855)

1. [连接MySQL数据库 315](#bookmark3863)
2. [查询MySQL数据库 317](#bookmark3873)
3. [插入MySQL数据库 318](#bookmark3886)
4. [删除MySQL数据库 320](#bookmark3894)
5. [更新MySQL数据库 321](#bookmark3902)
6. [操作MySQL数据库连接池 322](#bookmark3910)

**第** 15 **章** Node.js **与** MongoDB **交互** 325

1. [连接MongoDB数据库 326](#bookmark3944)
2. [连接MongoDB数据集合 328](#bookmark3971)
3. [查询MongoDB数据集合 330](#bookmark3979)
4. [插入 MongoDB 数据集合 333](#bookmark3994)
5. [删除MongoDB数据集合 335](#bookmark4007)
6. [更新MongoDB数据集合 336](#bookmark4016)

第1章 Node.js控制台模块应用

Node.js是一个基于Google V8 JavaScript引擎建立的服务器端平台，可以用来方便地 搭建快速的、易于扩展的Web应用。众所周知，V8引擎执行JavaScript的速度非常快，性 能非常好。Node.js采用了一个称为“事件循环（event loop） ”的架构，使得编写可扩展性 高的服务器变得既容易又安全。其中，Node.js提供了控制台模块，通过该模块的方法可以 向操作系统控制台实现输入和输出等操作（比如console.log> console.error等方法），实际 上也就是目前非常流行的“读取-求值-输出”循环（Read-Eval-Print Loop,简称REPL）交 互式的编程环境。

本章主要包括以下内容。

* 控制台下console模块各种方法的应用。

・数据的各种格式化输出。

* Node.js框架的跨平台应用。
* Node.js框架下解决几个知名的数学算法问题。

1. Node.js 版之"Hello World"程序

我们还是以一个“Hello World"程序开始第一个示例，通过这个例程将Node.js平台的 几种主要开发环境简要介绍给读者。目前，Node.js平台开发在Windows环境与Ubuntu环 境下均十分流行，本书尽可能将两种操作系统环境下的开发细节呈现给读者，以帮助读者 全方位地了解Node.js平台的强大之处。

Node.js平台环境的安装这里就不赘述了，刚刚起步的读者可以参考Node.js官方网站 或者互联网上相关的资源。

下面就让我们进入主题，看看基于Node.js的“Hello World”具体是什么样子的。

本例chOl.hello-world.js主要代码如下：

01 /\*\*

02 \* ch01.hello-world.js

03 7

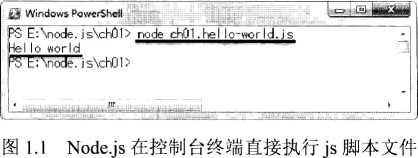
04 console.logCHello World!”）； 〃输出日志内容

1. Windows环境下

读者可以使用文本编辑器（如EditPlus、NotePad++. UltraEdit等）将以上代码保存为

ch01.hello-world.js脚本文件来直接使用oNode.js平台强大之处就是可以将js脚本程序直接 以控制台方式运行，具体执行方式有以下几种。

1. 控制台终端直接执行js脚本文件，如图1.1所示。



从图中可以看到，使用node命令直接执行ch01.hello-world.js脚本文件，会在控制台 输出运行结果。

1. 控制台终端以参数方式直接运行，如图1.2所示。

|  |  |  |  |
| --- | --- | --- | --- |
| 序 **Windows** PowerShell | | | |
| PS E:\node.js\ch01〉 node r  He I Io world | 'console. losCHel Io w |  | 1 |
| PS E:\node.js\chO1> |  |  |  |
|  | i |  |  |

图1.2 Node.js在控制台终端以参数方式执行js脚本语言

从图中可以看到，使用node命令带-e参数直接执行脚本语言，并在控制台输出运行结 果。读者可以在终端使用node -help命令查看关于-e参数的说明，其含义是eval script,也 就是执行脚本语言的意思。其实，了解JavaScript语言eval()函数的读者对此并不陌生，两 者的功能是基本一致的，就是计算字符串并执行其中的JavaScript代码。

(3) REPL (Read-eval-print loop)方式，如图 1.3 所示。



图1.3 Node.js在控制台终端以REPL方式执行js脚本语言

从图中可以看到，使用node命令进入Node.js平台的shell界面(具有“>”提示符的 状态)，这就是REPL方式。REPL (Read-Eval-Print Loop,简称REPL)表示“输入-求值 -输出”循环的含义，如果读者输入了一个函数并回车，那么REPL方式会直接在下面输出 这个函数的执行结果与返回值，本例中的“undefined”就是console.log()函数的返回值。但 如果读者输入了一个错误的指令，REPL则会立即显示错误并输出调用栈。

1. Ubuntu环境下

读者可以使用系统自带的文本编辑器(如vi、gedit、sublime-text等)将以上代码保存 为ch01.hello-world.js脚本文件来直接使用。在Ubuntu环境下执行Node.js控制台的方法与 在Windows环境下基本类似，以下是几种具体的实现过程。

(1)控制台终端直接执行js脚本文件，如图1.4所示。

**□ king(9kfng-Thlni(P»d-X61: -/no4«ls.**

[klng@ki.ng](mailto:klng@ki.ng) - ThinkPad • X61:~/nodejs$ node uhei.hello-world, js Hello, world

ktnggking-ThinkPad•X61:-/nodejs$ |

图1.4 Node.js在控制台终端直接执行js脚本文件

从图中可以看到，在Ubuntu环境下需要先进入js脚本文件所在的目录，然后再使用 node命令直接执行ch01.hello-world.js脚本文件，运行结果会在控制台进行输出。

1. 控制台终端以参数方式直接运行，如图1.5所示。

**ktnggktng-ThinkPad-X61:~/nodejs$ node -e "cocsole.log('Hell**。 **World')": Hello World '**一

**ktng@klng-ThinkPad•XG1:-/nodejsS |**

图1.5 Node.js在控制台终端以参数方式执行js脚本语言

从图中可以看到，在Ubuntu环境下Node.js在控制台终端以参数方式执行js脚本语言 的方法与Windows环境下是一致的。

(3) REPL (Read-eval-print loop)方式，如图 1.6 所示。

**ktng^ktng-ThinkPad-X61:-/nodejs$ node > console.log('Hello world");**

**Hell**。 **world**

**undentned**

图1.6 Node.js在控制台终端以REPL方式执行js脚本语言

从图中可以看到，在Ubuntu环境下Node.js在控制台终端以REPL方式执行js脚本语 言的方法与Windows环境下是一致的。

说明：因为这是本书第一个例程，我们着重介绍了 Node.js平台下几种控制台运行js 脚本语言的方法。在后面例程中我们会选择其中一种方法来运行js脚本语言，并将陆续使 用几款支持Node.js平台的集成开发软件来运行js脚本语言。

提示：在Windows环境下与Ubuntu环境下搭建Node.js开发平台的方法略有不同， Ubuntu环境下需要编译源程序包后再执行安装，且对Ubuntu文件系统的权限相对也有限 制，而在Windows环境下安装则相对简单得多，刚刚起步的读者如果遇到问题可以发邮件 联系我们进行咨询。

1.2日志输出与信息输出功能

既然Node.js平台能够支持控制台操作，那么常规的日志输出与信息输出应该也是必有 的功能。在下面这一例程中，我们将演示console.log()与console.info()这两个方法具体的应用。

本例ch01.hello-world.js主要代码如下：

01 /\*\*

02 \* ch01.log-info.js

03 \*/

04 console.log(\*'Node.js - console.log()方法，

05 console.info("Node.js - console.info()方法，

上面的代码演示了 Node.js平台中console.log()方法与console.info()方法的应用，通过 这两个方法向控制台输出内容完全一致的字符串，下面是这两个方法的语法：

*II*输出日志内容

*II*输出信息内容

语法:console.log([data], 语法：console.info([data],

*II*打印输出1行数据

*II* 同 console.log()方法

console.log()方法的功能是向控制台实现标准输出，该方法也可以像C语言的printf() 一样接受多个参数。而console.info。方法的功能与console.logQ方法是完全一•致的，输出的 效果也是一致的。

下面向读者演示使用基于JavaScript语言的集成开发软件WebStorm开发Node.js平台 应用的方法，如图1.7所示。

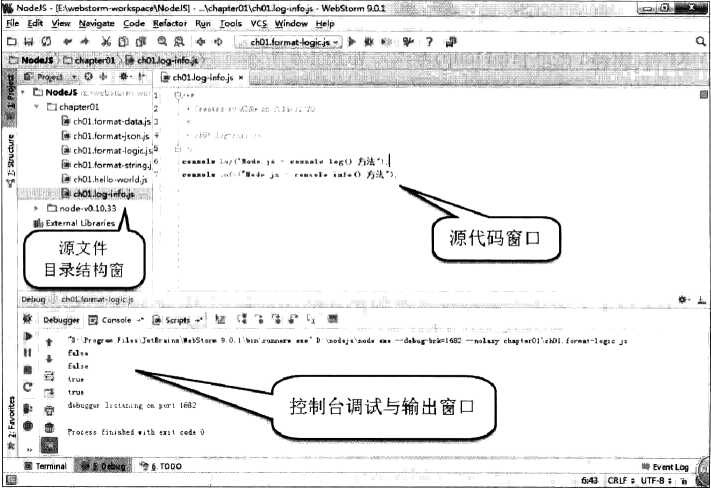


图1.7应用WebStorm集成开发软件运行js脚本文件

下面我们看一下图中源代码窗口，如图1.8所示。



碰 chOUog-infojs «

console. I og(\*\*Node, js - console.log()方法")； console.info(MHode.js - console.Info()方.法”)；

图1.8 WebStorm集成开发软件源代码窗口

我们单击工具栏中的Debug 'chOl-log-info.js'按钮，查看调试运行后结果输出，如图

1.9所示。

**Debuggor** 回 **Console -@** -\* 拦 **4 '• 1 <■**

**Debug chOllog-inloji**

**4-**

目

©

"0：\Program Files\JetBrains\WebStorm 9.0.l\bin\runnerw.exe' debugger listening on port 3035

Node.js - console.log()方法 \_ *，,*二…/、、『「、

Node, js - console.info()方法 市V 口调试

Process finished with exit code 0 与辎I 出窗 口 ,

图1.9 WebStorm集成开发软件调试输出窗口

从图中可以看到，使用console.log()与console.info()这两个方法输出的结果是完全一致 的，说明这两个方法实现的功能是完全一致的。

提示：在Windows环境下与Ubuntu环境下使用WebStorm集成开发软件开发Node.js 平台应用的方法大同小异，读者可以到jetBrain.com官方网站上下载相应的WebStorm软件 版本。配置WebStorm软件支持Node.js平台的方法这里就不具体讲解了，读者可以参考互 联网上的相关资源，刚刚起步的读者如果遇到问题也可以发邮件联系我们进行咨询。

1.3字符串格式化的应用

对于任何一种编译执行或解释执行的编程语言来讲，数据格式化输出都是最基本的功 能之一，所谓格式化就是将数据的表现形式固定为某一种常用格式，例如自然数、保留三 位小数、日期、时间、IP地址等。在Node.js平台下，数据格式化主要针对以下三种类型， 分别是字符型格式、数值型格式和JSON数据交换格式。从本例开始，我们将通过三节分 别向读者进行介绍。

首先，在本节介绍一下字符型格式化字符串的使用方法，本例ch01.format-string.js主 要代码如下：

01 /\*\*

02 \* chO1 .format-string.js

03 \*/

04 console.log("%s", "This is a format-string Note.js Program."); /\* %s ・ String 7

05 console.log("%s %s %s", "You can”，"connect1\*, "several strings.");

06 console.log("%sN, "You can", "connect", "several strings.");

07 console.log("%s:%s", "object", "string");

08 console.log("%s-%s", "object", "string");

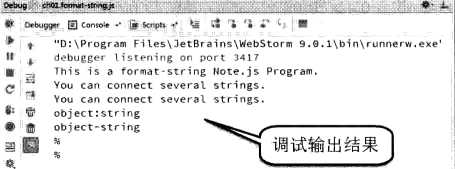
09 console.logf%");

1. console.log("%%");

**【代码分析】**

第04行是字符型格式的应用，将log()方法中的第1个参数设定为“％s”，则第2个 参数会按照字符型格式进行输出：第05行是字符串连接的应用，将log()方法中的第1个 参数设定为"％s %s %s”，将会把第2个到第4个参数连接起来并按照字符型格式进行输 出；第06行与第05行实现的功能一样，不同的是将log()方法中的第1个参数设定为“％s”与设定为“％s %s %s”的输出效果是完全一样的；在第07行与第08行中，我们在log() 方法中的第1个参数中加入特殊字符(":”与,它将会按照其位置在格式化字符串 中进行输出；在第09行与第10行中，我们看到了特殊字符“％”的使用，在这样使用“％%” 时，第-•个“％”字符将会被当作占位符进行处理。

图1.10是本例程调试输出后的结果。



Process finished with exit

图1.10字符型格式化调试输出结果

提示：在Node.js框架的官方文档中指出，单个百分号“％”在格式化字符串中使用时, 将占用一个占位符空间，但如果占位符没有对应的参数，则占位符不会被替换。

1.4

数值型格式化的应用

延续上一节的内容，本节我们将通过一个例程来介绍在Node.js平台下使用数值型格式 化字符串的具体方法。本例ch01 .format-data.js主要代码如下：

01 /\*\*

02 \* ch01 .format-data.js

03 \*/

04 console.log(8+0.8); *II* 数值运算

05 console.log("%d", 8, 8.8); 〃数值格式化输出

06 console.log("%d-%d", 8.8, 8.0); *II* 含有特殊字将的格式化

07 console.log("%d", 8+8, 8-8, 8\*8, 8/8); *II* 数值运算格式化输出

08 console.logC%d", "8+8");

09 console.log(8/0); *II* 除。运算

**【代码分析】**

第04行是直接在log()方法中将参数定义为数值(非字符串)，将会把该数值直接进 行输出；第05行是标准数值型格式化的应用，将log。方法中的第1个参数设定为“％d”， 将会把从第2个参数开始的参数依次按照数值型格式进行输出；在第06行中，我们在log() 方法中的第1个参数中加入特殊字符(,它将会按照其位置在格式化字符串中进行 输出；在第07行中，我们看到了一个简单的四则运算数值型格式化输出；在第08行中， 我们测试了一下使用数值型格式化输出字符串的效果，输出结果为“NaN”，表示格式化 出现异常；在第08行中，我们测试了除数为0时的结果，与其他编程语言有所不同的是，

输出结果为"Infinity”。

图1.11是本例程调试输出后的结果。

酷 **Debugger** 园 **Console -•**区 **S<rtpts**『陛。：搭 **Ti** 尸**C**

• t '•D:\Program Fi les \ Jet Bra ins\WebStorm 9.0. l\bin\runnerw.exe'

II , debugger listening on port 3534

M *一*—一｛调试输出结果、

\*& 8.8-8 J - '

•毯 **16 © 64 1** ”格式错误〃与〃除**0”**输出结果］)

£ Process finished with exit code 9

图LU数据型格式化调试输出结果

提示：关于第04行，在Node.js平台的官方文档中有这样解释，如果第一个参数没有 定义格式化类型，则将会把该参数进行必要的运算后转化为字符串格式进行输出。在实际 应用中，开发者几乎不用考虑该转化，因为最终在控制台输出的就是字符串格式。

1. JSON数据交换格式应用

本节我们将通过一个例程来介绍在Node.js平台下使用JSON数据交换格式的具体方 法。有关JSON数据交换格式的简要介绍，读者可以参看本节后面的提示部分的内容。本 例ch01.format-json.js主要代码如下：

01 *I\*\**

02 \* ch01 .format-json.js

03 7

04 console.log("%jw, (OS:nWindows',,Version:'7.T,,Language:[MEnglish,,,HChinese"]});

05 var vjson = ( /\* JSON 交换格式定义 \*/

06 OS:"Windows",

07 Version"”，

08 Language：rEnglish","ChineseT

09 }

10 console.log("%jn, vjson);

**【代码分析】**

第04行是直接在log。方法中将第一个参数定义为JSON格式("％j” ),然后把第 二个参数定义的JSON数据进行输出；第05〜10行是另一种实现方式，首先定义了一个 JSON格式的变量并对该变量进行赋值，然后再由log。方法对该变量进行输出。

图1.12是本例程调试输出后的结果。从图中可以看到，两种方法执行后的效果是完全 一致的，可见Node.js平台对JSON数据交换格式有很好的支持。



♦

$

启

HD：\Program FilesVJetBrains\WebStorm 9.0.1\bin\runnerw.exe" D:\nod debugger listening on port 3624

("OS": "Windows", "Version'1: "7*.V*,•'Language" ： ["English","Chinese"]) ("OS\*\*："Windows","Version',："7.1" /'Language": ["English”,"Chinese"]}

Process finished with exit code 0

调试输出结果

一 *― J*

图1.12 JSON对象格式化调试输出结果

提示：JSON的英文全称为JavaScript Object Notation,是一种轻量级的数据交换格式。 JSON是JavaScript的一个子集，是一种全新的、与编程语言无关的、易于解析和生成的文 本格式。基于以上这些特点，JSON数据已经无可厚非地成为了 一种用于网络数据传输的标 准数据交换格式。

1. 测试特殊运算符如何取值

如果读者有学习过C、Java和JavaScript这样的基础编程语言，一定知道对于含有比较 运算符或逻辑运算符的表达式，同普通表达式一样会有一个表达式值。那么在Node.js平台 下，比较运算符和逻辑运算符的特性是怎样的呢？在本节我们就将通过一段代码来测试说 明在Node.js平台下，特殊运算符是如何进行判断取值的。本例ch01.format-logic.js主要代 码如下：

/\*\*

01

02

03

04

05

06

07

08

09

\* ch01 .format-logic.js

7

var a=0;

var b=1;

*II*使用比较运算符，输出false //使用比较运算符，输出false

*II*使用比较运算符，输出true

console.iog(a==b);

console.log(a>=b);

console.log(a<=b);

console.log(a==0 && b==1); //使用逻辑运算符，输出 true

**【代码分析】**

第06〜08行是使用比较运算符（"==”、">=”、“v=”）进行运算，然后输出运 算结果（true或者false）；第09行是综合使用比较运算符（）与逻辑运算符（"&&” ： 逻辑“与”运算符）进行运算，然后输出运算结果（true或者false）。

图1.13是本例程调试输出后的结果。从图中输出结果可以看到，运算法比较运算与逻 辑运算的结果为true和false两个真值，基本延用了 C、Java、JavaScript等编程语言的传统 规范。

提示：在Node.js平台下，关于比较运算符与逻辑运算符的详细用法，读者可以借鉴 JavaScript语言，两者基本上是完全一致的。



"D:\Program Files\JetBrains\WebStorm 9.Sl\bin\runnerw.exe" debugger listening <»n port 3683

—■J调试输出结果:

Process finished with exit code G

图1.13运算符调试输出结果

1. 特殊格式化参数应用

在前面几节中，我们向读者演示了几种常规数据格式化的应用。其实在实际应用开发 过程中，我们可能还会碰到许多特殊情况，例如：当仅有第一个参数出现，其写法有类似 “％8s”这样的格式化参数时，Node.js平台会如何处理呢？本节我们将通过一段代码来说 明在Node.js平台下关于特殊格式化参数的用法。本例ch01.fbrmat-argument.js主要代码如 下：

01 /\*\*

02 \* ch01 .format-argument.js

03 \*/

04 console.log("%sH, "argument");

05 console.log("%s");

06 console.log("%d", 8);

07 console.log("%d");

08 console.log("%8s");

09 console.log(w%8dM);

1. console.Iog(”％8s”, "%8s”)；
2. console.log(n%8d", 8);

**【代码分析】**

第04〜05行是“％s”的比较用法，如第05行单独使用“％s”且仅有一个参数时， Node.js平台将会把“％s”当做字符串直接输出；第06〜07行是“％d”的比较用法，其输 出结果与“％s”是一致的；第08〜09行是格式化参数的特殊用法，如果格式化参数写成 如“％8s”或“％8d”这样的格式，Node.js平台则会把它当做字符串来处理；第10〜11行 告诉我们，即使console.log。方法含有第2个参数,第1个参数(如写成“％8s”或“％8d” 这样的格式)也不会具有数据格式化功能，仍将直接当做字符串来处理并输出。

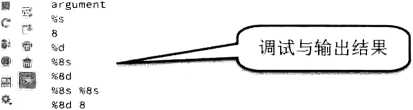
图1.14是本例程调试输出后的结果。

提示：在Node.js平台下，单独使用console模块进行数据格式化的功能比起Java和 JavaScript语言还是要弱一些的，一般需要借助于JavaScript语言才能完成更强大的操作， 在后续的例程中我们会逐步介绍给读者，



擀 十 "D:\Program Files\JetBrains\WebStorm 9.Q.l\bin\runnerw.exe"

H 爹 debugger listening on port 7215



Process finished with exit code 0

图1.14格式化参数调试与输出结果

1. 斐波那契(Fibonacci)数列应用

通过前面几节的介绍，读者应该对Node.js平台的控制台信息输出有了一定的了解。那 么在本节中，我们将著名的Fibonacci数列融合到Node.js平台控制台开发中，向读者演示 使用console.info()方法输出Fibonacci数列的过程。

本例 chO 1 .console-fibonacci.js 主要代码如下：

01 /\*\*

02 \* ch01 .console-fibonacci.js

03 *\*1*

04 console.info("\nn);

05 console.info(M Fibonacci 数列应用 ”)；

06 console.info(',\nH);

07 var strFibonacci; //定义字符串变量，用于输出Fibonacci数列

08 vari=1J=1,s=i+j; 〃初始化变量 i,j

09 strFibonacci ="输出 Fibonacci 数列>>> 1 1// 初始化 Fibonacci 数列

1. //循环结束判断条件
2. while(s <= 1000)
3. {

\*

1. strFibonacci = strFibonacci + s + M

:-r'H-A-- 11 > - :: - ~ - •- •\_ # - 腕“ *.•- n -■<：* -»- •-： -- •-： --'1 --•-/> H-'-i'项：密 & 。窗::矣....：：■■、。二者“ ：：：.-•»■-« *\* a ■.*

1. i = j;
2. j = s;
3. s = i + j; // Fibonacci 数列:F(n) = F(n-1) + F(n-2)
4. }
5. console.info(strFibonacci); // 打印输出 Fibonacci 数列
6. console.info(H\nM);

**【代码分析】**

第07行定义了保存Fibonacci数列所需的变量strFibonacci；第08行定义了三个变量并 进行了初始化(我们知道Fibonacci数列的第1个和第2个数列项均为数值1,之后的数列 项基于它进行叠加);第09行对变量strFibonacci进行初始化;第11〜17行即实现Fibonacci

数列的具体算法，感兴趣的读者可以参考有关讲述Fibonacci数列算法的书籍；其中第13 行用于实现将计算出的数列项叠加保存在变量strFibonacci中；第18行实现输出Fibonacci 数列。

图1.15是在Ubuntu环境下本例程调试输出后的结果。

**X □ klngOt(lng-ThfatkP\*d-X230：**

ktng^king-ThinkPad-X230:~/webstornprojects/NodeJS/chapter®l$ node chei ,console-ftbonacct.js

1 Fibonacci数列应用

输出Fibonacci数列>>> 1 1 2 3 5 8 13 21 34 S5 89 144 233 377 61© 987 ktng^king-ThinkPad-X230:-/webstornproJects/NodeJS/chapter61$ |

图1.15 Fibonacci数列输出结果

提示：在Node.js平台下，使用console.info()方法输出后会自动换行，因此我们无法计 算出一个数列项随几进行输出。本例采用变通的方法，将叠加计算出的每一个数列项保存 在一个变量中，全部计算完毕后再一次性输出数列。

1.9错误与警告提示应用

当用户在界面中输入发生错误时，系统通常会给出一个错误或警告提示。如果想在 Node.js平台中实现这个错误或警告提示功能，可以通过console.error()与console.wam()这 两个方法来实现，下面是一个演示这种错误提示的例程。

本例ch01.error-wam.js主要代码如下：

|  |  |  |
| --- | --- | --- |
| 01 | /\*\* |  |
| 02 | \* ch01 .error-wam.js |  |
| 03 | \*/ |  |
| 04 | var fe = require('信')； | *II*引入fs模块 |
| 05 | var file = ,chapter01 /error-wam.txt\*; | *II*定义文件源 |
| 06 | var encoding = 'UTF・8'; | //定义字符编码 |
| 07 | fs.readFile(file, encoding, function(err, data) ( | *II*读取本地文件函数 |
| 08 | if(err)( |  |
| 09 | console.error(werror - \n %s", err); |  |
| 10 | console.wamCwam ・ \n %s", err); |  |
| 11 | ) else ( |  |
| 12 | console.log(data); |  |
| 13 | } |  |
| 14 | }); |  |

**【代码分析】**

这个例程主要功能为读取本地文本文档，并将其中内容进行输出。下面我们只看第09〜

10行，这两行分别使用console.error()方法与console.wam()方法将err参数的内容输出到终 端。关于console.error()方法与console.wam()方法的语法如下：

语法：console.error([data],

语法：console.wam([data],

〃打印输出1行错误数据.

// 同 console.error()方法

图1.16是本例程调试输出后的结果，从中可以看到，使用console.error()方法与console, warn。方法运行效果是完全一致的，这两个方法实现的功能也是一样的。

就 **0\*bu99\*f E Consol\* -•**白 **Scnpts ■»,**恒**f & ・**

,'D:\Program Files\JetBrains\WebStorm 9.0.l\bin\runnerw.exe" D:\nodejs\node.e: debugger listening on port 49143

error -

Error ： ENOENT, open \* e： \websvorm-works-pace\NodeJS\chapter61\error-warn .txt' warn -

Error: ENOENT, open 'e:\webstorm-workspace\NodeJS\chapter01\error-war n.txt'

Process finished with exit code 0 调试与输出结果'

图1.16错误信息方法、警告信息方法调试与输出结果

提示：在 Node.js 平台下，console.error()方法与 console.log()方法一样，在 console.error() 方法中也可以通过参数指定输出字符串的格式，可使用参数与console.log。方法中使用的参 数相同Q

1.10重定向标准错误输出流

Node.js平台的控制台操作同样可以实现重定向标准错误输出流的功能。在执行node 命令时，可以指定命令的标准错误输出流形式，缺省状态是控制台屏幕，也可以为文件重 定向。本例我们实现重定向标准错误输出流到文件的功能。

本例ch01.stderr-file.js主要代码如下：

01 /\*\*

|  |  |  |
| --- | --- | --- |
| 02 | \* ch01 .stderr-file.js |  |
| 03 | 7 |  |
| 04 | var fs = require('fs\*); | //引入fs模块 |
| 05 | var file = 'chapter01 /not-found .txt1; | //定义文件源 |
| 06 | var encoding = 'UTF-81; | //定义字符编码 |
| 07 | fs.readFile(file, encoding, function(err, data)( | //读取本地文件函数 |
| 08 | if(err) ( |  |
| 09 | console.error(Merror • \n %sM, err); |  |
| 10 | console.wam(Mwarn - \n %s", err); |  |
| 11 | } else ( |  |
| 12 | console.log(data); |  |
| 13 | } |  |

14 })；

**【代码分析】**

这个例程主要功能是通过读取一个本地不存在的文本文档来产生错误，并将系统错误 提示输出到本地错误文件中。我们看第05行，定义了一个本地不存在的文本文档(文件名 为not-found.txt)；再看第07行，通过readFile。函数来读取该文档，由于该文档不存在， 所以系统会报错。

在控制台使用node命令重定向标准错误输出流到文件的语法如下：

语法:node js-filename.js 2> error.log

图1.17是在控制台使用重定向标准错误输出流到文件的方法。

**QWindowsPawerShefl ： \ I BfM**

阳 E:\webs1 orm-workspace\nodejs\chapterO 1 > nod^h0^stdcu^^HeJ^2^errorJog ■\* |PS E:\webstomrworkspace\nodej s\chapterQ]> ； *J*

**I ‘2 A**表示重定向标准错误输出膈到又件  
山  ~~一史~~，％

图1.17在控制台使用重定向标准错误输出流到文件的方法

输出到本地错误日志文件(文件名为error.log)的内容如图1.18所示。

MTor.ios -

g- EC)h(v)~ — 33ZTT Z

hode.exe : error - \*

所主位宜if：i字符：5 + node «C< chOl. sidout-file. js 2> error, log

+ CategoryInfo : NotSpecified： (error - :String) [], RemoteExcsption

+ FullyQuolificdErrDrld : Nat ivcConmandError

Error： ENOEKT, open \*E:\websTomrworkspace\nodejs\chapTer01\cha|：Ter01\noT-found, txt\*

warn

Error： ENCOT, open E:\wehstorm-morkspace\nodejs\ch^>terOl\ch«^ter01 \not —found, txt

图1.18本地错误日志文件的内容

提示：在Node.js平台下，任何运行程序引发的错误信息均可被重定向到某个文件中“ 当输出目标是文件时,console模块方法是同步执^■的，这样可以防止过早退出时丢失信息。 console函数根据输出目标的不同分为“同步/异步”两种方式，在平常使用中用户不需要大 担心“阻塞/非阻塞”的差别，除非需要记录大量数据。

1.11查看Node.js对象的属性和方法

在Node.js平台下，可以使用console.dir()方法实现快速查看对象实例所包含属性和方 法的功能。在我们不知道某个Node.js对象实例所包含的属性和方法时，这个功能还是非常 实用的。在这个例程中，我们看一下console.dir()方法对于不同的对象类型是如何进行输出 的。

本例ch01.obj-dir.js主要代码如下:

01 /\*\*

02 \* ch01 .obj-dir.js

*II*使用console.dir()方法查看整型对象

console.dir(123); console.dir(Mabcw); console.dir(rabc": 123}); console.dir(1 +2\*3+1); console.dir(console);

//使用console.dir()方法查看字符串对象

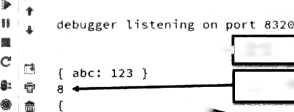
*II*使用console.dir()方法查看JSON对象

*II*使用console.dir()方法查看表达式对象

//使用console.dir()方法查看Node.js核心模块

**【代码分析】**

第04行实现的是查看整型对象(123)的功能，其输出结果仍是整型对象；第05行实 现的是查看字符串对象(“abc”)的功能，其输出结果仍是字符串对象；第06行实现的 是查看JSON类型对象(( “abd” :123})的功能，其输出结果仍是JSON类型对象；第07 行实现的是查看运算表达式对象(1+2\*3+1)的功能，其输出结果是该运算表达式的计算结 果(本例程计算结果等于8)；第08行实现的是查看console核心模块对象的功能,console.dir。 方法将会输出console对象所包含的全部属性和方法明细。

图1.19是本例程调试输出后的结果。

Debugge\* 回 Console 巨 Scripts 蛙，S

log: [Function],— info: [Function], warn: [Function], error: [Function], dir: [Function], time: [Function],

"D:\Program Files\JetBrains\WebStorm 9.0. l\bin\runnerw.exe\*\*

§ 盘c，  整型、字符串与JSO\类型|

整型、字符串与JSO\类型

运算表达式值

timeEnd: [Function],

*T rstff\* '* f Fi mennl

Process finished with exit code

console对象所包含的 属性和方法

1.12

时间计时器应用

图1.19使用console.dir()方法查看对象实例的属性和方法

提示：如果查阅Node.js文档规范，我们会知道console.dir()方法实际上在底层是通过 对obj对象使用util.inspect()方法、并将结果字符串输出到stdout来实现的°关于util模块 的方法我们会在后面章节做详细介绍。

Node.js平台的console模块提供了时间计时器的功能，分别是通过console.time()与 console.timeEnd()这两个方法组合使用来完成的。在本节的例程中，我们通过console.time() 与console.timeEnd()这两个方法来实现计算代码执行时间的功能并进行屏幕输出。

本例chOl.time-timeend.js主要代码如下：

01 /\*\*

02 \* ch01.time-timeend.js

03 7

04 var n=0;

05 console.time('10000x10000-times'); 〃使用 console.time()方法开始时间计时器

06 for(var i=0; i<10000; i++)(

07 for(varj=0; j<10000; j++)(

08 n++;

09 )

1. )
2. console.timeEnd('10000x1 OGOO-times\*); 〃使用 console.timeEnd()方法结束时间计时器
3. console.info(n);

**【代码分析】**

该例程首先定义一个整型变量n,初始值设定为0,然后通过一个简单的双重循环体来 叠加增加变量n的值，实际上n值也相当于记录了循环叠加的次数。第05行通过 console.time()方法标记一个时间开始点，并带有一个字符串(110000x 10000-times\*)参数。 关于console.time()方法的语法如下：

语法：console.time(label) // mark a time

第06〜10行实现的是双重循环体来叠加增加变量n值的功能，也相当于记录了循环叠 加的次数，实际上时间计时器记录的就是这段时间间隔；第11行通过console.timeEnd()方 法标记一个时间结束点并将时间记录值进行输出(时间单位为毫秒ms)，该方法同样带有 —个字符串(40000x 10000-timesf)参数：console.timeEnd()方法的语法如下：

语法：console.timeEnd(label) // Finish timer, record output.

第12行通过变量n的值来输出循环计算的次数。

图1.20是本例程调试输出后的结果。

**Debug chOLfane-twnetfidji** 彖.*虹*

* **Debugger (rj Console -•**窟 **Scripts** 竖**’"i 7 4?**战 **R**
* 十 "D:\Program Files\JetBrains\WebStorm 9.0.l\bin\

**II 4** debugger listening on port 3760

. 10000x10000-times： 782ms

**C 3** 时间计时器调试输出|

®s ProcessVfinished with exit code 0

, a 1

**™M** 双重循环次数值

图1.20时间计时器应用调试输出

提示：console.time()与console.timeEnd()这两个方法都仅使用一个参数，其参数值可以 为任何字符串形式，但需要读者注意的是这两个方法所使用的参数字符串必须相同，这样 才能正确地统计出开始时间与结束时间之间所经过的毫秒数。

1.13查看当前调用栈的方法

如果用户有过C、Java等高级语言的开发经验，一定知道在调试程序时debug工具会 将当前程序调用栈的情况展示给用户，方便用户了解程序的运行情况。同样，console模块 的trace()方法实现了向标准错误流输出当前调用栈的功能。在本节例程中，我们研究一下 console.trace()方法具体向用户输出了什么内容。

本例chO 1 .console-trace.js主要代码如下：

01 /\*\*

02 \* ch01 .console-trace.js

03 7

04 console.trace(); 写！ 〃第一次调用 console.trace()方法

05 console.trace("startH); // 第二次调用 console.trace()方法

06 var fs = require('fs');

07 var file = 'chapter01 /not-found .txt';

08 var encoding = 'UTF-8';

09 fs.readFile(file, encoding, functionferr, data) (

1. console.traceC'fs-start"); *II* 第三次调用 console.trace()方法
2. if(err) (
3. // console.errorC'error - \n %s", err);
4. // console.warnC'warn • \n %s", err);
5. } else (
6. // console.log(data);
7. }
8. console.trace(nfs-endM); // 第四次调用 console.trace()方法
9. })；
10. console.trace("endH); // 第五次调用 console.trace()方法

**【代码分析】**

以上代码演示了在代码的不同位置使用console.trace()方法的输出效果。该例程总体上 是一个读取本地文件的程序，我们分别在代码执行的不同位置调用console.traceO方法，目 的就是研究一下当前调用栈中的具体内容是什么。关于console.trace()方法的语法说明如下:

语法：console.trace(message, [...])

console.trace。方法的功能是打印当前位置的栈，并跟踪到标准错误输出流(process.stderr)*。* 第04行直接使用console.traceO方法进行输出，而且不使用参数；第05行使用带有一 个字符串start参数console.traceO方法进行输出；从图1.21中可以看到，第04行使用不带 参数与第05行使用带参数的console.traceO方法输出当前堆栈内容的结果是一致的，唯一 一点小区别就是带参数的console.trace。方法在输出时会在trace字符串后加上该参数。

第06〜18行实现的是一个通过fs.readFile()方法读取本地文本文件并记录错误的功能; 其中分别在第10行与第17行使用console.trace。方法进行输出，这两个方法使用不同的参 数(fs-start与fs-end)；从图1.21中可以看到，第10行与第17行使用console.traceO方法

**X X ?**

**5**

KD:\Program Files\JetBrains\WebStorm 9.0.l\bin\runnerw.exe'\* D:\nodejs\node.exe -一debug-brk=： \* debugger listening on port 211S4

Trace

at

at

at

at

at

at

at

Trace：

at

at

at

at

at

at

at

Trace：

at

at

at

at

at

at

at

Trace：

at

at

at

Trace：

at

at

at

Object.<anonymous> (e：\厂m-wo厂・ console-trace,js；6；9 Module.\_co^pile (module.js：456：26)

Object.Module..extensions..js (module.js：474:16)

Module.load (module.js:356：32)

Function, Module— load (module. js：312 :12) ♦"

Module.runMain [as \_onTimeout] (module.js；497：18)

Timer ・ ListOnTimeout [as ontimeout] (timers.js:112:15) start

0bjec t. <anonymous^ (e； \ \c/ill ・ *7:9*

Module.\_compile (module.js：456：26)

Object.Module..extensions..js (module-js：474：10)

Module.load (module.js:356：32)

Function,Module — load (rnodule.js:312:12) . 一

Module.runMain [as \_onTimeout] (module< js：49 7：10)

Timer.IistOnTimeout [as ontimeout] (timers.js:112:15)

end

Object. < anonymous^ (e： IwebstorflT-wo/JcsDadNQduJSlc/iaEtefSI'cnn, console・ t厂ace, js；21：!

Module..compile (module・js:456：26)

Object• Module.\_extens\*ions.. js (module, js：474： 1。)

Module.Load (module.js：3S6:32)

Function.Module.\_load (module.]s:312：12) ■ —

Hodule.runMain [as \_onTimeout] (module.js：497：10) Timer ・ UstOnTimeout [as ontimeout] (timers.js:112：15) fs-start e； \ we/>s to, m-wQ，\c方■ co〃5ole-rface・js； 12:13 fs.js：207：2O

Object.oncomplete (fs.js：107:IS)

fs-end

console, traced)方法

console, trace ("start")方法

console, trace ("end")方法

console. trace("fs-start”)方法

*e：*lonsole-trace.js:I9：I3

fs.js：207：2G

Object.oncomplete (fs.js：107：15)

console, trace("fs-end")方法

第1章Node.js控制台模块应用・17…＞

输出当前堆栈内容的结果是一致的，但与第04、05行的输出的堆栈内容是有区别的，在执 行第10行与第17行时，当前堆栈是在fs.js模块中，由Node.js官方文档可以知道目前程 序正在使用文件系统(仅)模块；第19行是在代码运行最后执行一次console.trace()方法， 带有一个字符串end参数，从图1.21中调试输出结果可以看到，当前堆栈己经从文件系统 (fs)模块中跳出，返回到系统模块中了。

**Pre**。好。。**f "ini wi th exir rarip G**

图1.21 console.trace()方法调试输出

提示：console.traceQ方法不使用参数时，其调试输出结果仅显示trace字符串。

1.14 对表达式结果进行评估

Node.js平台console模块的assert()方法实现了对表达式结果进行评估的功能。在本节 的例程中，我们通过一段代码向读者演示如何应用console.assert()方法实现对表达式结果进 行评估的功能。

本例chO 1 .console-assert.js主要代码如下:

01 /\*\*

02 [[1]](#footnote-2) ch01 .console-assert.js

03 7

04 var i=0;

*II*定义变量i

*II*定义变量j

05 var j=1;

06 console.assert(i==0, "OK");

//评估表达式“i==0”

//评估表达式

07 console.assert(i==j, "throw an exception1\*);

【代码分析】

该例程比较简单，我们首先定义了两个整型变量，然后通过console.assert()方法对这两 个整型变量进行断言判断。关于console.assert()方法的语法说明如下：

语法：console.assert(expression, [message], [...])

console.assert()方法的功能是进行断言判断，如果expression执行结果为false,则抛出 一个带上 message提示信息的 AssertionError。

第04~05行定义了两个值不相等的整型变量；第06行使用console.assertQ方法对 “i==0”这个表达式进行判断，并带有一个OK参数；从图1.22中可以看到，第06行的 console.assert()方法没有对执行结果进行输出，我们知道对“i=0”这个表达式判断的结果 为true,所以console.assert()方法对于判断结果为“真”的表达式不执行结果输出；第07 行使用console.assert()方法对"i=』”这个表达式进行判断，并带有一个throw an exception 参数。

**Debug** *L* **chOLtonsGle-Mwrtjs .，" " , \_** *" \_ \_ 一* **土土**

Debugger 回 Console 演 Scripts 曜 'i *h* \*•£ S

渺 营 ”D:\Program Files\3etBrains\WebStorm 9.G.l\bin\runnerw.exe" D:\nodejs\node.exe

debugger listening on port 33Q5

message M参数信息异常提示

AsserfionError： throw an exception ，—BMMnMMMmMWRMwa at Console.assert (console.js:102：23) at Object. <anonymous> (e: 'webstonn-ty。厂牝console at Module.\_compile (module.jsJ456：26)

at Object.Module.^extensions.«js (module.js：474:10)

at Module.load (module.js：3S6：32)

at Function.Module. load (module.js:312：12)

at Module.runMain [as \_onTimeout] (module.js：497:10)

at Timer.ListOnTimeout [as ontimeoutJ (timers.js：112：15)

Process finished with exit code 8

图1.22 console.assert ()方法调试输出

从图1.22中可以看到，根据throw an exception参数提示知道第07行的console.assert() 方法抛出了异常输出，从对整型变量i和j的赋值可以得出“i==j”这个表达式判断的结果 为false,所以console.assert()方法对于判断结果为"假”的表达式抛出了异常输出。

提示：从Node.js官方文档中的解释可以知道，console.trace。方法与assert.ok()方法相 同，实际上console.trace。方法在底层的实现就是通过Assert模块的assert.ok()方法完成的。

1.15

读取命令行输入信息

在前面的例子中，向读者演示了 Node.js平台通过console模块向控制终端输出信息的 功能。那么Node.js平台能不能实现从控制台读取用户输入的功能呢？答案是肯定的。

Node.js平台实现了几种从控制台读取用户输入的方法。在本节例程中，我们通过一个 例程向读者演示如何应用readline模块的question()方法实现从控制台读取一行用户输入的 功能。

本例 chO 1 .console-readline.js 主要代码如下：

|  |  |  |
| --- | --- | --- |
| 01 | 尸\* |  |
| 02 | \* ch01 .console-readline.js |  |
| 03 | 7 |  |
| 04 | var readline = require(Teadline'); | //引入readline模块 |
| 05 | var rl = readline.createlnterface(( | //初始化rl对象 |
| 06 | input: process.stdin, |  |
| 07 | output: process.stdout |  |
| 08 | })； |  |
| 09 | rl.question("What is your name?", function(answer) ( | *II*读取命令行输入信息 |
| 10 | //Log the answer in a database |  |
| 11 | console.log(nMy name isn, answer); |  |
| 12 | rl.close(); |  |
| 13 | })； |  |
| 14 | rl.question("How old are you? ", function(answer) ( | *II*读取命令行输入信息 |
| 15 | //Log the answer in a database |  |
| 16 | console.logfTm M, answer); |  |
| *17* | rl.close(); |  |
| 18 | })； |  |

**【代码分析】**

在Node.js平台下使用readline模块，首先需要使用require方法加载该模块，如第04 行所示。

代码：var readline = require('readline');

加载好readline模块后，需要创建具体readline模块的对象(本例为变量rl),如第 05〜08行所示。使用readline模块的createlnterface()方法来实现，并在该方法内定义好标 准输入流和标准输出流。关于readline.createlnterface()方法的语法说明如下：

语法：readline.createlnterface(options)

createlnterface()方法创建一个readline模块的接口实例，并接受一个object类型参数 “options”，该参数可传递以下几个值。

* input：要监听的可读流(必需)。
* output：要写入readline的可写流(必需)*。*

. completer：用于Tab自动补全的可选函数。

* terminal：如果希望input和output流像TTY (计算机终端设备)一样，那么通过 传递参数true来实现，并且经由ANSI/VT100转码，默认情况下检查isTTY是否 在output流上实例化。

从第06行与第07行可以看到，本例程是通过使用input和output两个参数值来实现的; 在预定义工作做好后，使用定义好的变量rl引用question。方法进行从控制台读取一行用 户输入的操作。关于readline.question()方法的语法说明如下：

语法：readline.question(query, callback)

question。方法预先提示指定的query,然后在终端接受到用户输入后再触发指定的 callbackt回调函数)继续执行:第09~13行与第14~18行定义了两个使用readline.question() 方法的操作，分别实现了两段从控制台读取一行用户输入的功能；从图1.23中可以看到， 第09〜13行在询问了用户姓名后，通过answer参数接受用户在控制台的输入，并通过 console. log()方法配以一段文字将其显示在控制台终端中。

同样从图1.23中可以看到，第14〜18行实现了与前一段代码相同的功能，但是在控 制台终端中并没有得到显示，原因是在执行readline.question()方法的回调函数过程中，执 行了第12行rl.closeO方法后就返回到了系统上下文环境下，所以第14〜18行并没有被执 行。

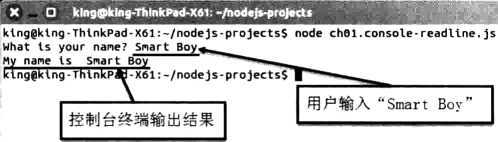


图1.23读取命令行输入信息

**提示：**

1. Node.js 官方文档中关于 readline.createlnterface()方法“options”参数使用“terminal” 值的解释比较晦涩，本书尽量做到按照官方文档的原意向读者进行解读„
2. Node.js官方文档对readline.question()方法“query”参数做了进一步描述，指出在显 示指定的“query”参数给用户后，当用户的应答被输入后，就触发了指定的callback (回 调函数)并返回。

1.16阶乘运算输出

上一节向读者演示了从控制台命令行读入用户输入信息的方法，那么在本节我们借助 这个方法实现一个Node.js平台下的阶乘运算输出应用。说起阶乘运算读者肯定不会陌生, 在概率与统计计算中阶乘是最常用的功能之一。通过计算机程序实现阶乘运算一般有常规 的循环计算方法与递归计算方法，在本节的例子中我们使用简洁易读的递归计算方法。

本例 ch01 .console-factorial.js 主要代码如下：

01 /\*\*

02 \* ch01 .console-factorial.js

03 \*/

04 console.infoCAn");

05 consoleJnfoC 阶乘计算 ");

06 console.info("\nn);

07 var iNum, iFactorial=1; //定义并初始化变量

08 varstrFactorial; 〃定义字符串变量，用于阶乘运算结果输出

09 〃从控制台读取用户输入

10

11

12

13

14

15

16

17

18

19

20

21

22

var readline = require('readline');  
var rl = readline.createlnterface({  
input: process.stdin,  
output: process.stdout

})；

ri.questionf请输入阶乘运算起始数值Ann =", function(iNum) ( //计算阶乘结果

strFactorial = iNum + "! = " + factorial(iNum); console.info(strFactorial); // 输出计算结果 console.info(M\nw);

rl.close();

})；

1. \*定义阶乘运算的递归方法
2. \*/
3. function factorial(n) (
4. if(n > 0)
5. (

28

29

30

31

32

33

34

35

1. )
2. return iFactorial;
3. }

if(n == 1)

( iFactorial = 1;

}

else

{

iFactorial = n \* factorial(n-1);

**【代码分析】**

该例程首先接受一个用户输入信息，用于定义阶乘运算的初始数值，然后通过递归计 算得出阶乘运算结果并输出。第07行定义了两个阶乘计算的变量，其中变量iNum用于接 受用户输入数值，变量iFactorial用于保存阶乘计算结果；第08行定义了一个字符串变量 strFactorial,用于保存阶乘计算输出；第10-21行用于接受用户输入的阶乘运算初始数值, 并调用factorial。方法进行计算；其中第17行通过将factorial()方法的返回值(即阶乘运算 结果)保存到变量strFactorial中，最后由第18行通过console.infb。方法进行输出；第25〜 38行定义了通过递归计算阶乘的factorial()方法，感兴趣的读者可以参考有关讲述阶乘运算

与递归算法的书籍；其中第37行用于将阶乘运算的结果进行返回。 图1.24是在Ubuntu环境下本例程调试输出的结果。

ktnggktng-ThtnkPad-X230:-/webstornprojects/NodeJS/chapterei$ node chOl. console-factorial.js

……-阶栗计算——

请辅入附乘运算起始数值：

6! = 726— 1—

介乘运算结果|

输入阶乘运算的初始数值

ktng^ktng-ThinkPad-X23O:~/webstormprojects/NodeJS/chapterei$ |

图1.24阶乘运算输出

提示：本节与前面1.8节一样，由于使用console.info()方法输出后会自动地换行，因此 我们将阶乘计算出的结果信息保存在一个变量中，全部计算完毕后再一次性进行输出„

1.17 向控制台输出组合控制键

在很多情况下，应用程序需要模拟组合控制键操作(如复制、粘贴、撤销等)来代替 用户完成一些键盘操作功能。而在Node.js平台下，通过readline模块的write()方法可以实 现向控制台输出控制键的功能。本节我们通过一个例程向读者演示如何应用readline.write() 方法，来实现对向控制台输出的内容进行撤销操作的功能。

本例ch01.console-write.js主要代码如下：

|  |  |  |
| --- | --- | --- |
| 01 | /\*\* |  |
| 02 | \* ch01 .console-write.js |  |
| 03 | \*/ |  |
| 04 | console.infofNode.js ・ readline.write() Usage'); | |
| 05 | var readline = require('readline'); | *II*引入readline模块 |
| 06 | var rl = readline.createlnterface(( | *II*初始化rl对象 |
| 07 | input: process.stdin, |  |
| 08 | output: process.stdout |  |
| 09 | })； |  |
| 10 | rl.writeCDelete me! Wait for 3 seconds...r | )； |
| 11 | var timeoutLength = 3 \* 1000; | 〃3秒 |
| 12 | var timeout = setTimeout(function() ( | //调用 setTimeout。方法 |
| 13 | //模仿ctrl+u快捷键，删除之前所写行 | |
| 14 | rl.write(null, (ctrl:true, name/u'}); |  |
| 15 | }, timeoutLength); |  |

**【代码分析】**

第04行通过console.info()方法向控制台终端输入一行内容提示文字；第05~09行执

行的是加载readline模块并进行初始化定义的操作；第10行使用readline模块的write。方 法来向控制台终端写入一行文字内容，关于readline.write()方法的语法如下：

语法：readline.write(data, [key])

write。方法将参数data的内容写入到控制台标准输出流，参数key是一个代表键序列 的对象，当终端是一个TTY (计算机终端设备)时可用；第11行通过变量timeoutLength 定义了一段时间间隔(时长为3秒)；第12~15行借助setTimeout。方法实现了经过一段 时间延迟后，具体是通过第14行中的readline.write()方法执行向控制台模拟输出撤销操作 快捷键、并删除之前写入的一行文字内容的功能；其中readline.writeO方法的key参数为一 个JSON对象({ctrl:true, name:U}),该对象定义了一个组合控制键"ctrl+u” ,该组合控 制键实现了撤销上一步操作的功能。

图1.25和图1.26是本例程调试输出后的结果。

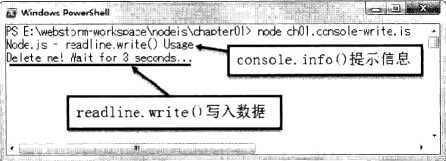


图1.25向控制台输出组合控制键(一)

*Q* Windows PowerShell IjEL上

PS E:\wei)storm-workspa^eXnodej s\chapterO1> node chOI.ccnsole-wr i te.j s Node.js - rsadlirw.wr ile() Usa&e



经过3秒的时间延迟后，通过组合控制键撤销 上一步readline.write。方法写入的数据



图1.26向控制台输出组合控制键(二)

从以上两张结果图中可以看到，图1.25中通过readline.write()方法向控制台终端写入 的数据，在经过一段时间延退后，再次通过readline.writeO方法以向控制台终端写入具有撤 销操作的组合控制键的形式实现了删除操作的功能，撤销操作后结果如图1.26所示。

**提示：**

1. 关于setTimeoutQ方法的具体应用方法可以参考JavaScript文档，实际上本例程中的 setTimeout()方法是属于JavaScript语言范畴之中的。
2. 如果在readline.writeO +执行了暂停操作，则会重置控制台标准输入流，直到执行 恢复操作后才会继续进行写入操作。

1.18模拟一个简单的控制台界面

Node.js平台的readline模块提供了很多实用的方法，可以实现足够强大的应用。本节通过一个例程向读者演示如何应用readline模块提供的方法，来模拟一个简单的控制台界 面应用。

本例chO 1 .console-tinyCLI.js主要代码如下：

20

21

22

23

24

25

26

27

28

29

30

|  |  |  |
| --- | --- | --- |
| 01 | /\*\* |  |
| 02 | \* ch01.console-tinyCLI.js |  |
| 03 | \*/ |  |
| 04 | var readline = requirefreadline1); | //引入readline模块 |
| 05 | rl = readline.createlnterface( | *II*初始化rl对象 |
| 06 | process.stdin, |  |
| 07 | process.stdout |  |
| 08 | )> |  |
| 09 | rl.setPrompt('NodeJS>'); | *II*定义模拟控制台命令行提示符 |
| 10 | rl.prompt(); | *II*初始化模拟控制台 |
| 11 | ri.onfline1, function(line) { | *II*激活readline模块的line事件 |
| 12 | switch(line.trim()) ( |  |
| 13 | case 'name': |  |
| 14 | console.logfkingr); |  |
| 15 | break; |  |
| 16 | case 'code': |  |
| 17 | console.logCNode.js!'); |  |
| 18 | break; |  |
| 19 | case 'time': |  |

console.logf2015r);

break;

default:

console.log('Say what? I might have heard + line.trim() + break;

rl.prompt();

//激活readline模块的close事件

}).on('close‘, function。{ console.log('Have a great day!'); process.exit(O); // 退出进程

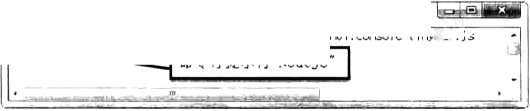
})；

**【代码分析】**

第04〜08行执行的是加载readline模块并进行初始化定义的操作；第09行通过 readline.setPrompt()方法模拟输出控制台的命令行提示符，本例程提示符为“NodeJS>”， 关于readline.setPrompt()方法的语法说明如下：

语法：readline.setPrompt(prompt, length)

其中，prompt参数用来定义命令行提示符，length参数用来定义命令行提示符的长度, length参数为可选参数，具体效果如图1.27所示。

P3 E:\websloi nr wo r ki»pace\i wdej bXJiapletOI > rtode diO1.3fisuleTiwCI\_I.Js NodeJ$><

命令行提示符"XodeJS"

H Windows PowerShell

图1.27模拟控台tinyCLI应用效果(-)

第10行通过readline.prompt()方法模拟实现控制台，等待接收用户的输入，关于readline. prompt()方法的语法说明如下：

语法：readline.prompt([preserveCursor])

其中，当preservecursor参数设置为true时，用来阻止命令行提示符的光标位被重置 为0,通常preserveCursor参数可以不用设定。

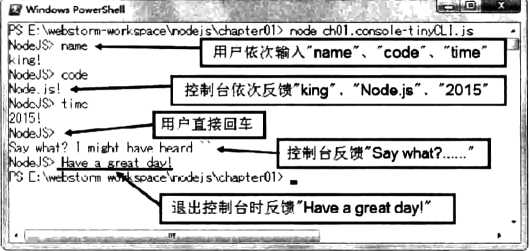
第11行通过readline模块提供的line事件来激发命令行输入功能，激活事件的方法为 Node.js平台Event模块提供的on()方法，有关Event模块的详细介绍放在后续章节，此处 读者了解on()方法的功能即可；第12〜24行通过switch选择语法，用来实现接收用户输入 后模拟控制台所做出的应答，具体效果如图1.28所示。

图1.28模拟控台tinyCLI应用效果(二)

从图1.28中可以看到，在命令行提示符“NodeJS>”出现后，用户依次输入name、 code、time,控制台依次向用户反馈king、Node.js> 2015；而当用户无输入直接回车时， 控制台则会向用户反馈默认定义的字符串(Say what? I might have heard),表示用户没有 任何输入；以上这些与我们在第12〜24行中实现的结果预期是完全一致的。

第26行通过readline.prompt()方法用来实现在接受完一次用户输入后，再次返回模拟 控制台命令行，等待用户下一次输入；如果没有该行，模拟控制台在接受完一次用户输入 后会失去命令行界面，读者可以自行验证一下；第27〜29行用来实现退出模拟控制台的操 作，此处通过readline模块提供的close事件来激发退出模拟控制台的操作，并且使用process 模块的exit。方法实现退出的功能；另外，本例程在模拟控制台退出前向控制台界面输出一 段字符串(Have a great day!),用于提示退出操作已完成。

提示：关于readline模块的事件说明及具体应用方法可以参考Node.js文档，实际上通 过本例程可以看到使用readline模块可以实现控制台的绝大多数功能。

1.19输出水仙花数

通过前面若干节的介绍，相信读者对于控制台模块的应用有了一定的了解。从本节开 始，我们通过对几个常见的数学算法问题使用Node.js框架方式编程求解，来加深读者对控 制台模块的理解。

本节我们来尝试使用Node.js框架方式编程求解数字100至999之间的水仙花数，所谓 水仙花数就是一个三位自然数的百位、十位和个位数字的立方和。在数学计算上我们不难 理解其计算方法，那么通过程序如何实现呢？

本例 ch01.console-narcissus.js 主要代码如下：

01 *I\*\**

02 \* ch01 .console-narcissus.js

|  |  |  |
| --- | --- | --- |
| 03 | \*/ |  |
| 04 | var m, a, b, c, sum; |  |
| 05 | for(m=100; m<1000; m++) ( | //循环判断100-999整数中的水仙花数 |
| 06 | a h parselnt(m/100); | //计算出百位数 |
| 07 | b = parselnt((m-a\*100)/10); | 〃计算出十位数 |
| 08 | c = parselnt(m-a\*100-b\*10); | 〃计算出个位数 |
| 09 | sum = a\*a\*a + b\*b\*b + c\*c\*c; |  |
| 10 | if(m == sum) { | //判断是否满足水仙花规则 |
| 11 | console.info(H%dM, m); |  |
| 12 | } |  |
| 13 | } |  |

**【代码分析】**

第04行定义了验证计算需要的变量，其中变量m用于for循环计数，变量a、b、c分 别用于保存百位、十位和个位数字，变量sum用于保存计算结果；第06-08行分别用于 计算出三位自然数的百位、十位和个位数字，其中parselntO方法的功能是对计算结果的小 数进行取整；第09行将百位、十位和个位数字的立方和计算结果保存在变量sum中；第 10-12行通过条件判断语句来验证该自然数是否满足水仙花数的条件，如果判断结果为真, 则使用console.infb()方法进行直接打印输出。

图1.29是在Ubuntu环境下本例程调试输出的结果。

ktng@king-ThinkPad-X23®:-/webstorm-js/projects/Node3S/chapter01$ node chfll.conso le-narcissus.js

153

376

371

407

100-999之间的水仙花数

ktng^ktng- ThinkPad-X236:~/webstorn-js/projects/NodeJS/chapter01$ |

图1.29水仙花数输出

提示：parselnt()方法属于JavaScript语言范畴，当其用于对一个小数进行取整时，读者 要注意的是其取整过程是将小数部分直接舍弃,而非四舍五入。

1.20输出质数

本节我们来尝试使用Node.js框架方式编程求解质数，所谓质数就是一个整数，该整数 只能被1和其自身整除。关于质数的概念读者应该是比较熟悉的，那么通过程序如何实现呢？

本例ch01.console-primeNum.js主要代码如下：

01

02

03

04

05

06

07

08

09

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

/\*\*

\* ch01 .console-primeNum.js

•/

var i, j, iNum, strPrimeNum;

varbFlag; //定义该变量，用于标记是否为质数

var readline = require('readline');

var rl = readline.createlnterface((

input: process.stdin,

output: process.stdout

})；

/\*\*

•用户输入一个整数作为上限，通过程序计算从2开始到该整数之间的全部质数 7

rl.question(MPlease enter number? ", function(iNum) ( //Calculate prime number strPrimeNum =" 2 M;

for(i=3; i<=iNum; i++)

*II*利用for循环计算质数

})；

bFlag = true; for0=2;j<i;j++) (

if(i%j == 0)

bFlag = false; break;

}

}

if(bFlag == true)

*II*通过if条件判断是否满足质数的条件

strPrimeNum = strPrimeNum + i + "n; //将质数保存在一个字符串变量中 }

}

console.log("Prime number > ", strPrimeNum); // 打印输出质数序列 rl.close();

**【代码分析】**

第04〜05行定义了验证计算需要的变量，其中变量bFlag可以理解为一个布尔型变量,

用于保存验证成功的质数，而变量strPrimeNum则用于保存全部验证成功的质数序列；第 06行开始的关「接受用户控制台输入的代码与前面若干节基本类似,在此就不过多阐述了 ； 第16行我们先将整数2 （最小的质数）保存到字符串变量strPrimeNum中；第17~32行通 过for循环语句来逐次验证从1至用户输入的整数之间的全部整数，其中第20〜27行通过 for循环语句来验证是否满足质数的条件（质数只能被1和其自身整除），通过第24行的 bFlag变量来进行标记，然后通过第28-31行将质数保存在字符串strPrimeNum变量中， 最后由第33行使用console.logO方法进行直接打印输出。

图1.30是在Ubuntu环境下本例程调试输出的结果。



Js/projects/NodeJS/chapte「61$ node chOl .conso 盅罢暨旨籍蜘"？ ,"11用户输入整数上限I

Prime number > 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 *73 79* 8

3 89 *97*

ktng^ktng-ThinkPad-X230:-/webstorm-js/projects/NodeJS/chapter61$ |

|质数序列输出|

图L30质数输出

提示：在实际编程求解过程中，类似布尔型bFlag变量的用法是很常见的，许多基础 编程语言的算法书中都有论述，感兴趣的读者可以自行参考学习。

1.21打印输出杨辉三角

本节我们来尝试使用Node.js框架方式编程打印输出杨辉三角，所谓杨辉三角最本质的 特征是，它的两条斜边都是由数字1组成的，而其余的数则是等于它肩上的两个数之和。 我们选择一个古典的杨辉三角样式供读者了解，如图1.31所示。

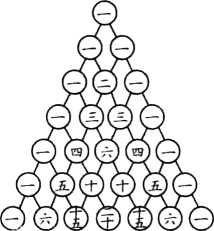


图1.31古典杨辉三角样式

如果通过编程方式打印输出杨辉三角，除了需要了解其实现算法外，格式控制也是很 重要的环节，否则无法输出有效的三角形形状。

本例 chO 1 .console-YHTriangle.js 主要代码如下：

01 /\*\*

02 \* ch01.console-YHTriangle.js

\*/

var i, j, k, strLine;

var readline = require('readline');

rl = readline.createlnterface(

process.stdin, process.stdout

)；

rl.question(Hline is： ", function(iLine) ( // 请输入杨辉三角行数

〃输出杨辉三角第一行

strLine = ”>”；

for(i=1; i<=10\*iLine; i++)

{

if(i == 5\*iLine)

(

strLine = strLine + 1;

} else {

strLine = strLine + "

)

}

console.info(strLine);

〃输出杨辉三角后续行

strLine =

f0r(j=1；j<=iLine-1;j++)

{

for(k=1; k<5\*(iLine-j-1); k++)

( strLine = strLine +

}

y = 1;

strLine = strLine + FormatNumber(y);

for(k=1; k<=j; k++)

(

y = y\*(j.k+1)/k；

strLine = strLine + FormatNumber(y);

)

console.info(strLine);

strLine =

}

rl.close();

})；

/,•

* FormatNumber()方法用于格式化输出杨辉三角
* @param y
* ©returns {string}
* ©constructor

1. 7
2. function FormatNumber(y) (
3. var strL =
4. if(y<10)
5. (
6. strL = strL + ” " + y +
7. )
8. else if((y >=10) && (y<100))
9. (
10. strL = strL + ” ” + y + v
11. }
12. else if(y >= 100)
13. (
14. strL = strL + ” ” + y + ”
15. }
16. return strL;
17. )

**【代码分析】**

该例程首先通过第13〜21行打印输出了杨辉三角的第一行，之所以单独打印输出第一 行主要是为了控制输出格式，然后通过第25〜40行依次打印输出了剩余的行，打印输出过 程中调用了一个用于控制格式的自定义函数FormatNumber（）,该函数通过判断数字位数来 调整其间距。该例程可以接受用户的输入，根据用户输入的具体行数来打印输出杨辉三角。

图1.31是一个七行的古典杨辉三角，下面我们在Ubuntu环境下打印输出现代版的七 行杨辉三角，如图1.32所示。

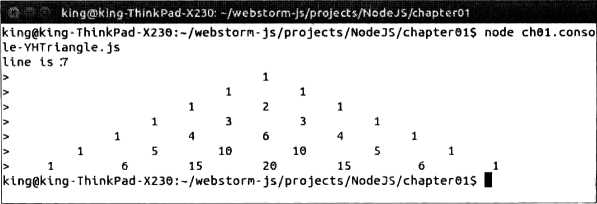


图1.32打印输出杨辉三角

提示：杨辉三角，又称帕斯卡三角形，是二项式系数在三角形中的一种几何排列。我 们之所以习惯称其为杨辉三角是因为它最早出现在我国南宋数学家杨辉1261年所著的《详 解九章算法》一书里。

1.22 汉诺塔（Hanoi）问题

还记得在大学C语言基础课上讲递归算法时的经典汉诺塔（Hanoi）例子吧，本节我们

通过Node.js框架方式编程来实现。

汉诺塔（Hanoi）是根据一个古老传说而形成的数学问题，有3根柱子A, B, C,其 中A柱上有N个（N>1）穿孔圆盘，盘的尺寸由下到上依次变小，要求按下列规则将所有 圆盘移至C杆。

规则一：每次只能移动一个圆盘。 规则二：大盘不能叠在小盘上面。

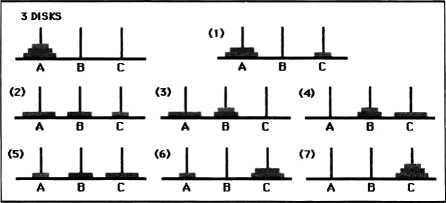
下面具体通过三个盘子的移动示 意图让读者对汉诺塔（Hanoi）问题有 一个感性的认识，如图1.33所示。

图1.33汉诺塔（Hanoi）示意图

上图中左上角的图例是初始状态， 所有圆盘按照从大到小顺序依次放在 A柱上，然后按照汉诺塔（Hanoi）规 则依次通过7步的移动将圆盘移动到 C柱上，移动过程中所有圆盘的摆放顺 序均满足从大到小的方式，右下角图例 为最终状态。

其实通过递归算法编程模拟汉诺塔（Hanoi）问题，还是非常容易的，本例 chOl .console-hanoi.js 主要代码如下：

01 /\*\*

02 \* ch01 .console-hanoi.js

03 *\*1*

04 var iCount = 0; //TODO:定义移动次数

05 var strMoveStep =//保存移动步骤

06 var readline = require('readline');

07 var rl = readline.createlnterface((

08 input: process.stdin,

09 output: process.stdout

1. ))；
2. rl.question("请输入汉诺塔问题的圆盘数目:\t", function(count) (
3. //print user input
4. console.log("输入汉诺塔问题的圆盘数目为:\t”, count);
5. hanoi(count, "A", "B", "C");
6. console.info("移动步骤如下:\t%s”, strMoveStep);
7. console.info("移动次数为:\t%s", iCount);
8. rl.close(); .
9. })；
10. /\*\*
11. \*移动圆盘的方法
12. \* @param x
13. \* @param y
14. 7
15. function move(x, y) (

25

26

27

1. ]
2. j

30

31

32

33

34

35

1. 1

37

38

39

40

41

42

43

44

45

46

47

iCount++; //console.info("%s->%s", x, y); strMoveStep += x + n->M + y + ”\t”；

/\*\* '

\*完成汉诺塔问题的递归方法

* @param m
* @param a
* @param b
* @param c

7

function hanoi(m, a, b, c)( if(rn== 1)

move(a, c); } else

hanoi(m-1, a, c, b); move(a, c);

hanoi(m-1, b, a ,c);

**【代码分析】**

该例程的核心部分为第36〜47行实现的递归方法hanoi(),该方法模拟了根据汉诺塔 (Hanoi)规则的移动步骤；其中move。方法用于保存每一步移动的记录，为了方便最后输 出在一行当中，我们将移动步骤保存在字符串变量strMoveStep之中，move()方法的具体实 现见第24〜28行；该例程可以接受用户的输入，根据用户输入的具体圆盘数目来打印输出 移动步骤。

下面我们在Ubuntu环境下打印输出汉诺塔(Hanoi)问题的移动步骤，如图1.34所示。

**king<®klng-ThinkPad-X230;** 戏**oefn-js/projects/god/JS/chapterOI**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ktng®ktng-ThtnkPad-X230:~/webstorn-js/projects/Node3S/chapter01$ node ch©l.conso le-hanot.js | | | | | | | | | |
| 款佥潴噩黯雅鬻 | | |  |  |  |  |  |  |  |
| ***6***  移动步骤如下： | A->B | A->C | B->C | A->B | C->A | C->B | A->B | A->C | **B** |
| ->C B->A | C->A | B->C | A->B | A->C | B->C | A->B | C->A | C->B | **A** |
| ->B C->A | B->C | B->A | C->A | C->B | A->B | A->C | B->C | A-aB | **C** |
| ->A C->B | A->B | A>>C | B->C | B->A | C->A | B->C | A->B | A->C | **B** |
| ->C B->A | C->A | C->B | A-aB | C->A | B->C | B->A | C->A | B->C | **A** |
| ->B A->C | B->C | A->B | C->A | C-aB | A->B | A->C | B->C | B->A | **C** |
| ->A B->C | A->B | A->C | B->C |  |  |  |  |  |  |
| 移动次数为： | 63 |  |  |  |  |  |  |  |  |

ktng@ktng-ThtnkPad-X230:~/webstorn-js/projects/NodeJS/chapter©l$ |

提示：现代人根据汉诺塔（Hanoi ）问题原理，又衍生出单层、双层汉诺塔（Hanoi ） 问题，四柱、多柱汉诺塔（Hanoi）问题，感兴趣的读者可以自行查阅有关资料，做更深一 步的学习研究

1.23简单四则运算应用

在本章最后一节，我们实现一个简单的四则运算应用。在数学中，我们把包含一级运 算（加减）和二级运算（乘除）、以及括号运算符（优先级）的运算叫四则运算。之所以 称为简单四则运算应用，是因为将其简化为仅仅包含两个运算数，且不含括号的二元运算, 主要目的就是测试一下在Node.js框架下通过编程实现数值运算的方法。

本例 chO 1 .console-calculate.js 主要代码如下：

|  |  |  |
| --- | --- | --- |
| 01 |  |  |
| 02 | \* ch01 .console-calculate.js |  |
| 03 | \*/ |  |
| 04 | var a, b, sign, summary; |  |
| 05 | console.infof'Calculate start!"); |  |
| 06 | /\*\* |  |
| 07 | \*模拟一个控制台输入界面 |  |
| 08 | \* @type (exports} |  |
| 09 | \*/ |  |
| 10 | var readline = require('readline'); | 〃引入'readline'模块 |
| 11 | rl = readline.createlnterface( |  |
| 12 | process.stdin, |  |
| 13 | process.stdout |  |
| 14 | ); |  |
| 15 | rl.setPrompt('Calculate>'); |  |
| 16 | rl.prompt(); |  |
| 17 | a = Math.round(Math.random。\* 100); | 〃随机生成100以内的整数 |
| 18 | b = Math.round(Math.random() \* 100); | 〃随机生成100以内的整数 |
| 19 | sign = Math.round(Math.random() \* 3); | 〃随机选取四则运算符 |
| 20 | summary = switchSign(a, b, sign); |  |
| 21 | rl.onCline\*, function(line) ( |  |
| 22 | if(line.trim() == summary) ( | //判断用户给出的运算结果是否正确 |
| 23 | console.info("answer is right!"); |  |
| 24 | } else ( |  |
| 25 | console.info("answer is error! The correct answer is" + summary +，”)； | |
| 26 | } |  |
| 27 | rl.prompt(); |  |
| 28 | a = Math.round(Math.random() \* 100); | 〃随机生成100以内的整数 |
| 29 | b = Math.round(Math.random() \* 100); | 〃随机生成100以内的整数 |

1. sign = Math.round(Math.random。\* 3); //随机选取四则运算符
2. summary = switchSign(a, b, sign);
3. }).on('close', function() {
4. console.log('Calculate exits!');
5. process.exit(O);
6. })；
7. /\*\*
8. \*四则运算方法-switchSign()
9. \* @param a
10. \* @param b
11. \* @param sign -运算符
12. \* ©returns {\*}
13. \*/
14. function switchSign(a, b, sign) {
15. var c;
16. switch(sign) (
17. case 0:
18. c = a + b;
19. console.info(a +"+" + b +
20. break;
21. case 1:
22. c = a - b;
23. console.info(a + 七” + b +
24. break;
25. case 2:
26. c = a \* b;
27. console.info(a + + b + ”=”)；
28. break;
29. default:
30. c w a + b;
31. consolednfo(a + "+" + b +
32. break;
33. }
34. return c;
35. }

**【代码分析】**

该例程首先基于第1.18节(模拟一个简单的控制台界面)的内容实现一个带提示符 (Calculate»的四则运算控制台界面，然后循环输出系统给出的随机计算表达式，由用户 一一做出解答。每次做答后系统都会提示用户给出的结果是否正确，并给出提示信息，如 果计算结果有误还会给出正确的答案。另外，无论计算正确与否都不会终止应用运行，直 到用户强行终止这个模拟出来的控制台界面。

klng@klng-ThinkP<!d-X230: -/webstorm-js/projects/NodeJS/chapterO!

ktng^ktng-ThinkPad-X23e：-/webstorm-js/projects/NodeJS/chapter01$ node chOl.conso le-calculate.js Calculate start!

Calculate\* 42\*24=

66

answer is rtghtl

Calculate〉 34-30=

5

answer is errorf The correct answer is 4.

Calculate〉 17\*80=

123

answer is error! The correct answer ts 1360.

Calculate\* 74-49=

25

answer is rtghtl

Calculate〉 27+34=

61

answer is right J

Calculate\* 99+93=

Calculate exits!

ktng@ktng-ThinkPad -X230:~/webstom-js/projects/NodeJS/chapterMS ■

图1.35简单四则运算应用

提示：本例程中用到的Math.random()方法属于JavaScript语言范畴，其功能是令系统 随机选取大于等于0.0且小于1.0的伪随机double值，感兴趣的读者可以自行参考相关文 档进行学习。

第2章 Node.js模块与包管理

Node.js框架有一套简单的模块加载系统，在Node.js框架中的文件和模块都是一一对 应联系的。众所周知，在开发一个复杂的应用级框架的时候，需要把各个功能拆分、封装 到不同的模块之中，在需要的时候调用该模块。高水平的应用开发不会出现一个模块文件 有成千上万行的代码，这样的模块文件在可读性、复用性和维护性上都是很差的。

几乎所有的高级编程语言都有自己的模块组织方式，比如Java中的包(package)、PHP 中的inc文件等。同样的，Node.js框架使用模块和包来组织管理，其实现机制参照了 CommonJS标准，因此Node.js框架对开发人员来讲是易学易懂、使用方便的。

Node.js框架的模块及包管理从性质及加载方式上可以分为4类内容：核心模块、文件 模块、文件夹加载和模块缓存。本章将从这几类入手，向读者介绍一些非常实用的应用例 程。

本章主要包括以下内容。

・Node.js框架的模块化与包管理机制。

. Node.js框架的核心模块应用。

* Node.js框架的文件模块应用。
* Node.js框架的文件夹加载。
* Node.js框架的模块缓存。

1. Node.js框架加载核心模块

Node.js框架中有一些模块是被编译成了二进制的，这些模块一般称为核心模块。核心 模块定义在Node.js框架源代码文件夹的lib目录下，使用require()方法进行加载。在Node.js 框架中，console模块、URL与QueryString模块、Path与File System模块都是核心模块， 在第1章已经详细地介绍了 console模块，其他模块之后也会陆续被介绍到。下面我们通过 展示一张结构图，让读者对核心模块有一个感性的认识，如图2.1所示。

从图中我们可以看到，最底层的是Google V8 JavaScript引擎，在V8引擎之上是基于 C/C"语言实现的核心模块、并提供向上的接口，在最上层使用JavaScript语言对这些接口 进行封装、再向外提供给用户使用这些核心模块。

在Node.js框架应用中，require。方法总是会优先加载核心模块。例如，require(http) 总是返回编译好的HTTP模块，而不管是否有同名的文件。



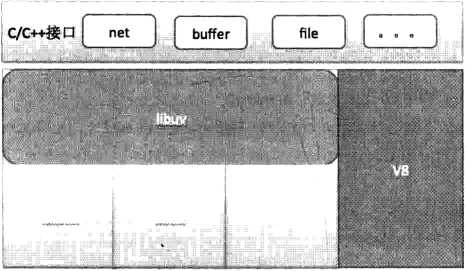


图2.1 Node.js框架核心模块示意图

本例ch02.module-core.js主要代码如下：

01 /\*\*

02 \* ch02.module-core.js

03 \*/

04 var http = requireChttp\*); 〃加载核心模块'http'

提示：关于Node.js框架总是优先加载核心模块的说法，我们可以这样去理解:在Node.js 框架安装好后，这些核心模块以编译好的二进制形式作为框架原生的组成部分存在，因此 require()方法即使遇到与模块同名的文件也不予理睬，而是直接返回已经编译好的核心模 块。

1. Node.js框架加载文件模块

Node.js框架有一个简易的模块加载系统，在Node.js框架中的文件和模块是一一对应 的。那么Node.js框架如何加载文件模块呢？下面我们借用官方文档中提供的一个示例来阐 述加载文件模块的方法。

这个例子是foo.js文件加载同一目录下的circle.js文件，其功能是通过给定的半径来计 算圆的周长和面积。

本例ch02.module-file-foo.js主要代码如下：

01 /\*\* '

02 \* ch02.module-file-foo.js

03 \*/

04 var circle = require('./ch02.module-file-circle.js');

05 console.info();

06 var readline = require('readline');

07 var ri = readline.createlnterface((

08 input: process.stdin,

09 output: process.stdout

1. ))；
2. rl.question("Please input radius:\tw, function(answer) (
3. 〃打印用户输出
4. console.logfThe radius answer);
5. console.Iog( 'The area of a circle of radius \* + answer + ' is \* + circle.area(answer));
6. console.log( 'The circumference of a circle of radius' + answer +' is \* + circle.circumference (answer));
7. console.info();
8. rl.close();
9. })；

ch02.module-file-circle.js文件接受用户输入的半径数值，然后调用ch02.module-file- circle.js文件模块计算圆的面积和周长。

本例ch02.module-file-circle.js文件模块主要代码如下：

01 /\*\*

02 \* ch02.module-file-circle.js

03 7

04 var PI = Math.PI;

05 exports.area = function(r) {

06 return PI \* r \* r;

07 };

08 exports.circumference = function(r) (

09 return 2 \* PI \* r

10 };

**【代码分析】**

ch02.module-file-circle.js 文件模块输出了 area()和 circumference()两个函数，分别用于 计算圆的面积和周长。当文件模块要输出某个对象时，把其加到exports这个特殊对象下即 可。在这里需要读者注意的是，exports是对module.exports的一个引用，使用exports仅仅 是为了用户使用方便。因此，如果想输出的是诸如构造函数这样的单个项目，那么就必须 使用 module.exportso

关于正确输出构造函数的语法如下：

语法：module.exports = MyConstructor;

另外，模块内的本地变量是私有的，因此在本例中，PI这个变量就是ch02.module-file- circle.js模块所私有的。

图2.2是在Ubuntu环境下本例程调试输出的结果。

king（3）king-ThlnkPad-X230: ~/webstorm-js/proJects/NodeJS/chapter02

ktnggktng-ThinkPad-X23e：~/webstorn-js/projects/Node3S/chapter©2$ node ch02.modul e-file-foo.js

Please input radius: 6

The radius is: 6

The area of a circle of radius ***6*** Is 113.69733552923255

The circumference of a circle of radius 6 is 37.69911184307752

ktng^ktng-ThinkPad-X230："-/webstorn-js/projects/NodeJS/chapter02$ |

图2.2文件模块调试输出窗口

提示：使用require()方法引用文件模块时，如果按文件名没有查找到，那么Node.js框 架会添加.js和.json后缀名尝试加载，如果还是没有找到，最后会加上.node的后缀名再次 尝试加栽 其中，.js会被解析为JavaScript纯文本文件，.json会被解析为JSON格式的纯 文本文件，.node则是通过C/C++进行编写的插件，最终会被解析为编译后的插件模块，由 dlopen()方法(Linux C/C++编程范畴)进行加载。

1. 从node modules文件夹中加载

如果require。方法中的模块名不是一个本地模块，也没有以'/'、或是'•/' 这样的相对路径开头，那么Node.js框架会从当前模块的父目录开始，尝试在node modules 文件夹里加载相应模块。如果仍旧没有找到相应模块，则再移动到上一级父目录，直到到 达顶层目录位置。

我们来举个例子，如果位于/home/king/webstorm-js/projects/chapter02/foo.js的文件调用 了 requireCcircle.js,),因为其没有'/'、'.・/'、或是'・/'这样的相对路径开头，那么 Node.js框架查找的位置依次为：

/home/king/webstorm-js/projects/chapter02/node\_modules/foo.js

/home/king/webstorm-js/projects/node\_modules/foo.js

/home/king/webstorm-js/node\_modules/foo.js

/home/king/node\_modules/foo.js

Zhome//node\_\_modules/foo.js

/node\_modules/foo.js »

这就要求用户编写程序时，尽量把依赖放在就近的位置，以防程序崩溃。

Node.js框架在module模块中为设计人员提供了 paths这个参数，在这里讲解一下这个 参数的含义。Node.js框架对每一个被加载的文件模块创建对象的时候，这个模块便会有一 个paths属性，其值则根据当前文件模块的实际路径而计算得到。

我们可以通过一段代码测试一下这个paths属性，本例ch02.module-node-modules.js主 要代码如下：

01 /\*\*

02 \* ch02.module-node-modules.js

03 \*/

04 console.logf—打印 paths 属性一');

05 console.log(module.paths);

06 console.logf—打印完成一')；

图2.3是在Ubuntu环境下本例程调试输出的结果。从图中可以看到，通过Node.js框 架打印输出的结果与之前我们根据规则写出的结果完全一致。

king^)king-ThinkPad-X230: ~/webstorm-js/projects/NodeJS/chapter02

ktng@ki.ng-ThinkPad-X230:--/webstorn-js/projects/NodeJS/cbapter02$ node ch02 .modul e-node-nodules.js

—打印paths属性---

['/home/ktng/webstorm-js/projects/NodeJS/chapter02/node\_nodules',

'/home/king/webstorm-js/projects/NodeJS/node\_nodules',

'/hone/king/webstorm-js/projects/node\_modules',

'/hone/ktng/webstorm-js/node\_modules \*,

'/home/king/node\_nodules',

'/hone/node\_nodules,,

'/node\_nodules']

打印完成…

kinggkxng-ThinkPad-X23e：~/webstorm-js/projects/NodeJS/chapter82$ |

图2.3 module.paths属性输出结果

提示：如果使用require()方法加载模块时引用的是绝对路径文件，则查找时不会去遍 历每一个node modules目录，这样其速度是最快的°

2.4模块的循环调用问题

在Node.js框架中，当引用模块存在循环的require。调用时，则一个模块可能在返回时 并不会被执行。在本节中，我们参考Node.js官方文档中给出的一个循环require()调用的例 程，测试一下循环引用模块会带来什么样的效果。本例程需要三个js文件，其中一个为主 js文件，分别调用另外两个辅助js文件，同时这两个辅助js文件进行了相互引用，也就是 循环的require。调用。

本例主 js 文件 ch02.module-conflict-main.js 代码如下：

01 /\*\*

02 \* ch02.module-conflict-main.js

'03 \*/

04 console.log();

05 console.logfmain starting\*);

06 var a = require('./ch02.module-conflict-a.js');

07 var b = require('./ch02.module-conflict-b.js,);

08 console.logfin main, a.done=%j, b.done=%j', a.done, b.done);

**【代码分析】**

第 06~07 行分别引用了 ch02.module-conflict-a.js 和 ch02.module-conflict-b.js 这两个文 件模块；第 08 行使用 console.log()方法打印输出了 ch02.module-conflict-a.js 和 ch02.module- conflict-b.js这两个文件模块导出的.done对象。

本例辅助js文件ch02.module-conflict-a.js代码如下：

01 /\*\*

02 \* ch02.module-conflict-a.js

03 \*/

04 console.logfa starting');

05 exports.done = false; 二…登萨二二

06 var b = require(,./ch02.module-conflict-b.js');

07 console.logfin a, b.done = %j‘，b.done);'

08 exports.done = true;

09 console.Iog('a done');

【代码分析】

第05行设定导出对象.done的值为逻辑值false,注意这里设定的是ch02.module-conflict- a.js文件模块的导出对象.done；第06行通过require()方法引用ch02.module-conflict-b.js文 件模块，并保存在变量b中；第07行打印输出在ch02.module-conflict-a.js文件中变量b引 用的.done导出对象的逻辑值；第08行重新设定导出对象.done的值为逻辑值false,注意这 里设定的也是ch02.module-conflict-a.js文件模块的导出对象.done。

本例辅助js文件ch02.module-conflict-b.js代码如下：

01 /\*\*

02 \* ch02.module-conflict-b.js

03 \*/

04 console.logfb starting');

05 exports.done = false;

06 var a = require('./ch02.module-conflict-a.js,);

07 console.Iog(\*in b, a.done = %j', a.done);

08 exports.done = true;

09 console.Iog('b done');

从以上代码可以看到，ch02.module-conflict-b.js 文件模块与 ch02.module-conflict-a.js 文 件模块类似，这里就不做重复解释了。

现在，一个主文件与两个辅助文件都已经给出，我们在总体上进行一下分析。

首先主 js 文件加载 ch02.module-conflict-a.js 文件模块，接着 ch02.module-conflict-a.js 文件又去加载 ch02.module-conflict-bjs 文件模块。这时，ch02.module-conflict-b.js 文件又会 尝试去加载ch02.module-conflict-a.js文件模块。

这时，Node.js框架为了防止无限地循环调用，ch02.module-conflict-a.js会返回一个 unfinished copy 给 ch02.module-conflict-b.jso 然后 ch02.module-confliet-b.js 就会停止加载， 并将其 exports 对象.done 返回给 ch02.module-conflict-a.js 文件模块。

这样主js文件ch02.module-conflict-mainjs就把这两个模块都加载完成了。

图2.4是在Ubuntu环境下本例程调试输出的结果。

提示：关于文中的unfinished copy这个概念，可以理解为未结束完成的对象副本，具体到 本例程可以理解为ch02.module-conflict-a.js文件模块的对象副本，其保留在 ch02.module-conflict-b.js文件模块中指导其结束并返回到ch02.module-conflict-a.js文件模块中。



king^king-ThinkPad-X230:~/webstorm-js/projects/NodeJS/chapter02$ node ch02.modul e-confItct-main.js

■I主js文件启动|

-I调用a.js文件模块启动|

main starting a starting^ b starting ，

tn b, a.done = false b done '

tn a, b.done = true —

& done 七 1返回a.js模块中，输出信息并结束a.js模块

in main, a.oone=true, b.done=true ' 、 , ~~-,.一.——一 -~~

调用b.js文件模块启动，发现重复加载a.js时，则停止加载|

在b.js文件中输出信息，并结束b.js模块|

js/projects/Node3S/chapter62$ | 画主js模块中，输出信息并结束|

rq-js/projects/Node3S/chapter62$ |

ktng@ktng-ThinkPad-X230:~/webi

图2.4循环加载文件模块的输出结果

1. module.exports 对象应用

在Node.js框架中，module.exports对象是通过模块系统产生的。在本章前几节中，我 们看到了 exports对象的用法，在本节的例程中演示一下module.exports对象的用法。

本例主 js 文件 ch02.module-exports-main.js 代码如下：

01 /\*\*

02 \* ch02.module-exports-main.js

03 *\*1*

04 console.logO；

05 var obj = require('./ch02.module-exports-obj.js');

06 obj.on('ready\ function() (

07 console.logfmodule ch02.module-exports-obj.js is ready');

08 console.logO；

09 })；

【代码分析】

第05行引用了 ch02.module-exports-obj.js这个文件模块，并将其保存在变量obj之中; 第06~09行通过on()方法激活obj对象的teady，事件，并在控制台打印输出提示信息。

本例引用js文件ch02.module-exports-obj.js代码如下：

01 /\*\*

02 \* ch02.module-exports-obj.js

03 \*/

04 var EventEmitter = require('events').EventEmitter;

05 module.exports = new EventEmitter。；

06 // Do some work, and after some time emit the 'ready' event from the module itself.

07 setTimeout(function() (

08 module.exports.emitCready');

09 }, 1000);

【代码分析】

第04行引用事件(event)对象的事件激发方法EventEmitter；第05行将module.exports

对象定义为EventEmitter方法;第07~09行通过设定一个时间间隔(使用setTimeout方法), 来激活EventEmitter方法中的Yeady\*事件。

图2.5是在Windows环境下本例程调试输出的结果。

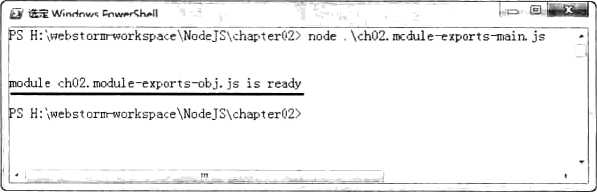


图2.5 module.exports对象的测试输出

提示：在Node.js框架中，module.exports才是module模块公开的接口，每个模块都 会自动创建一个module对象，对象有一个exports的属性，初始值是个空对象{}, module 的公开接口就是这个属性 module.exports o

1. module.exports 对象与 exports 对象

在Node.js框架中，module模块为了设计人员使用方便，在模块中提供一个exports对 象，其和module.exports对象指向同一个变量，因此在修改exports对象的时候也会修改 module.exports对象。但是，有时设计人员对module.exports对象进行了操作，在其不为空 的情况下，会发现exports对象就失效了，这是为什么呢？

本节针对该情况给出一个测试例程，向读者阐述一下module.exports对象与exports对 象的区别所在。

本例主 js 文件 ch02.module-exports-compare-main.js 代码如下:

01 /\*\*

02 \* ch02.module-exports-compare-main.js

03 \*/

04 console.logO；

05 var obj = require(,./ch02.module-exports-compare-obj.js,);

06 var user = new obj('king', 26);

07 user.userinfo();

08 console.log(user.tag);

09 console.logO；

**【代码分析】**

第05行引用了 ch02.module-exports-compare-obj.js这个文件模块，并将其保存在变量 obj之中；第06~07行通过new方法对user对象进行初始化，将姓名和年龄参数赋值，并 调用obj对象的userinfb()方法进行打印输出，关于userinfb()方法的定义见下面ch02.module- exports-compare-obj.js这个文件；第08行打印输出obj对象输出的tag对象。

本例引用 js 文件 ch02.module-exports-compare-obj.js 代码如下:

01 /\*\*

02 \* ch02.module-exports-compare-obj.js

03 \*/

module.exports = function(name, age) ( this.name = name;

04

05

06

07

08

09

10

11

12

13

this.age = age;

this.userinfo = function() (

console.log(this.name +' is '+ this.age +' years old\*);

｝；

｝；

exports.tag = function() ｛ console.logCThis is a tag function.\*);

｝；

**【代码分析】**

第04〜10行实际上是定义了一个js形式的类，这个类包含两个字段属性与一个函数 方法，这个函数方法实际是将这个两个字段属性打印输出，读者一定发现了这个类被赋给 T module.exports对象；而第11~13行是一个标准的exports对象的使用方法，定义了 一个 函数，用于输出提示信息，然后将这个函数赋给.tag对象进行导出。

图2.6是在Windows环境下本例程调试输出的结果。

**gSSSheB** 」回 **iMSHl**

PS H：\webstorm-workspace\NodeJS\chapter02> node . \ch02. nrodule-exports-conpare-rnain. js \* king is 26 years aid ~~"一~~| module exports对象导出的函数

undefined —… .一

—u..... ] - ]

PS H: \webstornrworkspace\NodeJS\chapter破罗exports tag对愈导出的函数 |

图2.6 module.exports对象与exports对象的对比

读者一定发现出问题了，.tag对象导出的函数打印输出的信息为'undefined',其含 义是未定义。这是什么原因呢？

参阅Node.js框架的官方文档，我们找到了答案。其实，module.exports才是module 模块的真正接口，而exports可以理解为它的一个副本，虽然修改exports对象的时候也会 修改module.exports对象，但最终返回给调用的是module.exports对象而不是exports对象。 因此，当module.exports对象通过赋值方式进行设定后，已经和exports对象指向的变量不 同了*，*这时无论exports对象怎么修改，都己经和module.exports对象没关系了。

具体到本例中，因为引用js文件ch02.module-exports-compare-obj.js的第04~10行已 经对module.exports对象进行了重新定义，所以由exports对象导出的.tag对象所定义的函 数就失效了，打印输出时提示了 "undefined未定义”。

**提不：**

1. 在Node.js框架中，如果想定义的模块是一个特定的类型就用module.exports对象， 如果想定义的模块是一个典型的"实例化对象”就用exports对象；
2. 对于模块可以是任何合法的JavaScript对象，例如：booleanx number、date、JSON、 strings function array 等。

2.7汇率换算实用程序

在Node.js框架中使用module模块导出对象时，一般建议还是使用exports对象，在上 —节我们已经阐述了 exports对象与module.exports对象的区别。在本节这个例程中，我们 利用module模块的exports对象编写一个汇率换算的实用程序，让读者对exports对象有一k 个更深入的了解。

在日常生活中，汇率换算对于普通大众说近不近、说远也不远。在购买外汇、海外留 学、出国旅游等情况下，都免不了要对汇率有个详细的了解。下面我们就以人民币和美元 之间的汇率换算为例，介绍一下exports对象的使用方法。

本例主 js 文件 ch02.module-exchangerate-main.js 代码如下：

01 /\*\*

02 \* ch02.module-exchangerate-main.js

03 •/

04 consoleJnfo("\nH);

05 console.info(" 汇率换算应用 ")；

06 console.info(M\nn);

07 var exRate = require('./ch02.module-exchangerate-obj.js'); // 加载自定义模块

08 var readline = requirefreadline1); *II* 加载核心模块

09 var rl = readline.createlnterface({

1. input: process.stdin,
2. output: process.stdout
3. })；
4. rl.question(nPlease enter count:H, function(iMoney) (
5. //调用dollar2rmb()方法将美元换算成人民币
6. console.info(iMoney +1 US-Dollar exchange to RMB equals' + exRate.dollar2rmb(iMoney));
7. //调用rmb2dollar()方法将人民币换算成美元'
8. console.info(iMoney + \* RMB exchange to US-Dollar equals' + exRate.rmb2dollar(iMoney));
9. console.info("\n");
10. rl.close();
11. })；

**【代码分析】**

第07行引用了 ch02.module-exchangerate-obj.js这个文件模块，并将其保存在变量 ex Rate之中；第08〜20行总体上完成了接受用户控制台输入数值，然后根据该数值分别输 出美元对人民币和人民币对美元的换算结果，其中关于接受用户控制台输入的代码在第1 章有过详细的介绍；第15行与第17行通过分别调用exRate对象的dollar2rmb()方法和 rmb2dollar()方法，打印输出美元对人民币和人民币对美元的换算结果，关于这两个方法的 定义见下面 ch02.module-exchangerate-obj.js 这个文件。

本例引用 js 文件 ch02.module-exchangerate-obj.js 代码如下：

01 /\*\*

02 \* ch02.module-exchangerate-obj.js

03 7

04 *II*使用module模块的exports方法导出rmb2dollar函数

05 exports.rmb2dollar = function (m) (

06 return m /6.2467 + " US Dollar"; //将人民币按照最新汇率换算成美元并返回

07 ｝；

08 //使用module模块的exports方法导出dollar2rmb函数

09 exports.dollar2rmb = function (m) (

1. return m \* 6.2467 + " Yuan of PRC"; //将美元按照最新汇率换算成人民币并返回
2. ｝；

**【代码分析】**

第04~07行通过exports对象向外导出了一个rmb2dollar()方法，该方法接受一个数值 参数作为人民币数额，然后根据当日人民币兑换美元的实时汇率(6.2467 : 1),换算出美 元数额并返回；而第08~11行通过exports对象向外导出了一个dollar2rmb ()方法，该方法 与rmb2dollar()方法类似，只是功能上是将美元换算成人民币。

图2.7是在Ubuntu环境下本例程调试输出的结果。

'"、 klng@iking-ThinkPad-X230: -•/webstorm-js/projects/NodeJS/chapterOZ

king@ki.ng-ThinkPad-X230:-/webstorm-js/projects/NodeJS/chapter02$ node ch©2.modul e-exchangerate-main.js

汇率换算应用

Please enter count: 8

8 US-Dollar exchange to RMB equals 49.9736 Yuan of PRC

8 RMB exchange to US-Dollar equals 1.280676197032033 US Dollar

ktng@ktng-ThinkPad-X230:~/webstorn-js/projects/NodeJS/chapter02$ |

图2.7汇率换算实用程序输出结果

提示：在Node.js框架中，规规矩矩地使用exports对象导出典型的“实例化对象"可 以避免不必要的程序错误，如无特殊需求，建议使用exports对象。

2.8简单计数器实用程序

计数器应用极为广泛，比如在手机的通话、短信、数据等类别的记录中非常实用，完 整的计数器应用应支持用户自主选择初始值、自动清零，以及按照类别添加提醒数值等功

能。本节这个例程基于Node.js框架的module模块，向用户演示一个简单计数器实用程序 的编写方法。

本例主 js 文件 ch02.module-counter-main.js 代码如下：

01 /\*\*

02 \* ch02.module-counter-main.js

03 \*/

04 var counter = require('./ch02.module・counter-obj.js');

05 console.logf第一次调用模块[ch02.module-counter-objjs]');

06 counter.seOutputVal(10); 〃设置从 10 开始计数

07 counter.setlncrement (10); //设置增量为 10

08 counter.printNextCount();

09 counter.printNextCount();

1. counter.printNextCount();
2. counter.printNextCount();
3. /\*•
4. \* require()多次调用同一模块不会重复加载
5. \*/
6. var counter = require('./ch02.module-counter-obj.js');
7. console.logf第二次调用模块[ch02.module-counter-obj.js]');
8. counter.printNextCount();
9. counter.printNextCount();
10. counter.printNextCount();
11. counter.printNextCount();
12. counter.printNextCount();
13. /\*\*
14. \*自动清零操作
15. \*/
16. console.logf自动清零操作，)；
17. counter.autoSetZero();

**【代码分析】**

第04行引用了 ch02.module-counter-obj.js这个文件模块，并将其保存在变量counter 之中；第05行输出一行提示信息，告诉这是第一次调用ch02.module-counter-obj.js这个文 件模块；第06〜II行分别调用counter对象的seOutputVal()方法、setincrement ()方法和 printNextCount()方法，分别完成设置初始输出值、增量值和增量打印输出功能，关于这三 个方法的定义见下面ch02.module-counter-obj.js这个文件；第15行第2次引用了 ch02.module-counter-obj.js这个文件模块，并将其保存在变量counter之中；第17〜21行再 次调用counter对象的printNextCount()方法，完成增量打印输出功能；第26行调用counter 对象的autoSetZero()方法，完成自动清零功能。

本例引用js文件ch02.module-counter-obj.js代码如下：

01 /\*\*

02 \* ch02.module-counter-obj.js

03 \*/

04 var outputVal = 0; 〃输出值

05 var increment = 1; //增量

06 /\*\*

07 \*设置输出值

08 \*/

09 exports.seOutputVal = function(val) (

1. outputVal = val;
2. }
3. /\*\*
4. \*设置增量值
5. \*/
6. exports.setlncrement = function(incrementVal) (
7. increment = incrementVal;
8. *}*
9. /\*\*
10. \*增量打印输出
11. \*/
12. exports.printNextCount = function() (
13. outputVal += increment;
14. console.log(outputVal);
15. }
16. /\*\*
17. \*自动清零
18. \*/
19. exports.autoSetZero = function() (
20. if(outputVal >= 100) (
21. outputVal = 0;
22. }
23. console.log(outputVal);
24. )

**【代码分析】**

第04行定义了一个全局变量。utputVaL用于保存输出值：第05行定义］'•个全局变 量increment,用于保存增量值;第09~11行通过exports对象向外导出了一个seOutputVal() 方法，该方法接受一个increment Vai参数值作为增量值，并赋给全局变量increment：第15〜 17行通过exports对象向外导出了一个setlncrement()方法，该方法接受一个val参数值作为 输出值，并赋给全局变量outputVal；第21~24行通过exports对象向外导出了一个 printNextCount()方法，该方法根据增量值进行叠加计算，并打印输出；第28〜33行通过 exports对象向外导出了一个autoSetZero()方法，该方法判断数值是否大于100,如果判断 结果为真，则进行清零操作，然后打印输出。

图2.8是在Ubuntu环境下本例程调试输出的结果。

|ktng^ktng-ThinkPad-X230:-/webstorn-js/projects/NodeJS/chaptere2$ node ch©2.nodul e-counter-main.js

klng(5>king-ThlnkPad-X230: -/webstorm-js/projects/NodeJS/chapterOZ

|第一次调用模块[ch82. module-counter-obj.js]

20

40

!se 第二次调用模块[ch02.module-counter-obj .js]

60

70

80

99

100

自动清零操作 e

klng@ktng-ThinkPad-X230:~

图2.8简单计数器实用程序输出结果

注意：在Node.js框架中，通过require()多次调用同一模块不会重复加载，原因是Node.js 会根据文件名缓存所有加载过的文件模块，所以不会重新加载了，通过文件名缓存是指实 际文件名并不会因为传入的路径形式不一样而被认为是不同的文件。

1. 包管理之package.json详解

本章除了关于Node.js框架模块这一内容外，另一块内容就是包管理。既然把模块和包 放在同-•章来讲，那么可以肯定的是两者之间关系紧密，所以我们先了解一下官方文档对 于两者的阐述。

在Node.js框架的官方描述中，其实包和模块并没有本质的不同。包是在模块的基础上 更深一步的抽象，包用于将独立的功能封装起来，用于发布、更新、依赖管理和进行版本 控制。相比JavaScript语言规范来讲，Node.js框架最大的进步就是根据CommonJS规范实 现了包机制，并开发了 npm实用工具来解决包的发布、更新、管理和获取需求。

Node.js框架的包是一个目录，该目录基本遵循CommonJS规范，其中包含JSON格式 的包说明文件package.jsono CommonJS规范定义的包特性如下。

. 顶层目录包含package.json文件。

* bin目录存放二进制文件。

. lib目录存放JavaScript文件。

* doc目录存放文档。
* test目录存放单元测试。

既然Node.js框架的包通常是一些模块的集合，并在模块的基础上提供了更高层的抽 象，这就相当于提供了一套固定接口的函数库。那么，为了开发人员能够创建更复杂、更 完善、更符合规范的包来用于发布，就需要通过定制package.json这个包说明文件来实现。

综上所述，了解和掌握Node.js框架包的关键在于对package.json包说明文件的理解， 下面我们就着重讲解一下package.json文件。

Node.js在调用包时，首先会检查包中的package.json文件的main字段，将其作为包的 接口模块，如果package.json文件的main字段不存在，那么Node.js会尝试寻找index.js或index.node作为包的接口。

package.json文件是CommonJS规范用于描述包的文件，完全符合规范的package,json 文件应该包含以下字段。

. main：包主入口程序路径描述。主入口字段描述是一个模块id,它是一个指向包

主程序的路径。

.name：包名。包名是唯一的，由小写字母、数字和下画线组成，不能含空格。

. description：包说明。对包进行简要描述。

* version：版本号。

. keywords：关键字数组，通常用于搜索。

* maintainers：维护者数组。每个元素包含name、email （可选）、web （可选）字

段。

. contributors：贡献者数组。格式与maintainer数组相同。包作者应该是贡献者数组

的第一个元素。

. bugs：提交bug的地址，可以是网址或电邮地址。

* licenses：许可证数组。每个元素要包含type （许可证名称）和url （链接到许可证 文本的地址）字段。
* repositories：仓库托管地址数组。每个元素要包含type （仓库的类型，如Git）、 url （仓库地址）和path （相对于仓库的路径，可选）字段。

. dependencies：包依赖。是一个关联数组，由包名和版本号组成°

下面展示一个包说明文件的例子ch02.module-packagejson.json,代码如下：

01 /\*\*

02 \* ch02.module-packagejson.json

03 \*/

04 {自动生成 package.json

05 "name": "myNodeJSApp",

06 “version”： ”0.01',

07 "author": "king",

08 "description": "Introduce to Nodejs Package.json",

"main” ： "./lib/main.js”，

09

10

11

12

13

14

15

16

17

18

"keywords'\*: "javascript, node.js", "respository": (

“type” ： “git”，

HurlH:,,<http://urr>

},

“bugs”：［

"url": ”[http://url/bug”，](http://url/bug%e2%80%9d%ef%bc%8c)

"email": "[bug@example.com](mailto:bug@example.com)"

1. }
2. ］,
3. “contributors” ：［
4. (
5. "name” ： ”xxx”，
6. "email": "[xxx@example.com](mailto:xxx@example.com)"
7. )
8. ],
9. "license": ”MIT”，
10. "engines": (“node” ： ”0.10.x”},
11. "script” ： {
12. "start": "node index.js"
13. },
14. "private": true,
15. "scripts": (
16. "start”： ''node ./bin/www"
17. },
18. "dependencies": (
19. "express”： ”~5.0.0”，
20. "debug”: ”~2.0.0”，
21. “jade”: ”~1.6.0”
22. },
23. "devDependencies": (
24. “grunt” ：”~0.5.1”
25. )
26. }

**【代码分析】**

整个包说明文件实际上是一个JSON格式的描述文件，用于描述Node.js开发包的全部 信息；第05行name字段描述了该包的名称，这个字段是必须定义的；第06行version字 段描述了该包的版本号，这个字段是建议一定要定义，便于后续地开发与升级；第08行 main字段描述了该包的主入口程序路径，这个字段是package.json文件最核心的部分，用 于指引Node.js框架去寻找主程序的路径，大多数情况下该字段是要定义的，不过前文也 提到，如果未定义main字段，则Node.js框架将会在包的第一层目录下搜索index.js或 index.json文件；其余的字段均是用于描述包的，作为良好的编程习惯建议填充完整，当然 这些字段都不是必需的，开发人员可根据需要定义。

Node.js框架同时为开发人员提供了一个很实用的npm包管理工具，通过该工具也可以 自动生成package.json包说明文件，下面是npm工具在Ubuntu环境下(Windows环境类同) 的运行方法：

/\*\*

\*执行npm命令自动生成package.json包说明文件需要进入到该包的第一层目录下

\*/

npm init 〃自动生成package.json包说明文件

图2.9是在Ubuntu环境下自动生成package.json包说明文件的过程。从图中可以看到， 开发人员只需要根据系统的提示一步一步地输入信息就可以了，最后系统会生成一个预览 版并提示“Is this ok? (yes)”，如果一切正常，输入yes并回车就可以自动创建package.json 包说明文件了。

关于CommonJS的说明：

• CommonJS是一种规范，Node.js框架是对这种规范的一种实现。

. CommonJS定义很多普通应用程序(主要指非浏览器的应用)使用的API,其终 极目标是提供一个类似Python, Ruby和Java的标准库。这样的话，开发者可以 使用CommonJS API编写应用程序，然后这些应用可以运行在不同的JavaScript 解释器和不同的主机环境中。

. CommonJS有很多具体的实现项目，比如Apache大名鼎鼎的CouchDB和Node.js 等。但是，这些项目大部分只实现了 CommonJS的部分规范。有关于具体项目及 实现部分，读者可以参见其官方网站([http://commonjs.org/impl/)的说明。](http://commonjs.org/impl/)%e7%9a%84%e8%af%b4%e6%98%8e%e3%80%82)



图2.9自动生成package.json包说明文件示意图

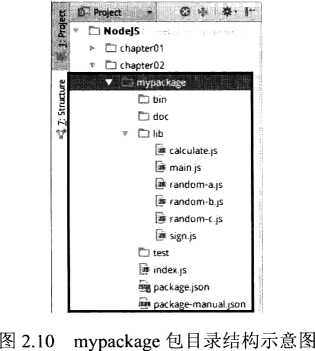
2.10基于包管理的四则运算应用

上一节我们详细阐述了 Node.js框架下关于包的概念，以及包管理的基本方法。那么在 本节，我们基于包管理的方式，开发一个简单的四则运算应用，这个应用相比第1章的简 单四则运算应用有两个方面的改进。

第一，运算符由一维改为二维，当然读者可以以此类推改进为多维。

第二，该应用基于包管理的方式编写，让读者对Node.js框架下包管理有了一个初步的 了解，为之后深入学习Node.js编程做了一个很好的铺垫。

下面我们看一下该包(mypackage)的目录结构，如图2.10所示。



下面，我们对图2.10展示的包的目录结构做一个说明：包的名称为mypackage,用户 可以根据具体应用设定合适的名称；该包下首层目录包括bin、doc、lib. test共四个文件 夹，以及index.js和package.json两个文件，其中四个文件夹与文件的含义在上一节有过详 细说明，读者可以自行参考；其中，在lib文件夹内定义了具体实现本应用的几个功能脚本 文件，下面我们逐一介绍这几个脚本文件。

其中，main.js定义为包的主入口文件，其代码如下：

01 /\*\*

02 \* main.js

03 \*/

04 var cal = requiref./calculate');

第04行调用同级目录(lib)下的calculate.js文件。 下面看一下calculate.js脚本文件，其代码如下：

01 /\*\*

02 \* calculate.js

03 \*/

04 var a, b, c, sign\_a, sign\_b, summary;

05 var ra = requiref./random-a\*);

06 var rb = requireC./random-b1);

07 var rc = requireC./random-c1);

08 var sign = requiref./sign');

09 console.info(nCalculate start!");

1. /\*\*
2. \*模拟一个控制台输入界面
3. \* @type (exports}
4. \*/

var readline = require('readline'); 〃引 入'readline'模块

rl = readline.createlnterface( process.stdin, process.stdout

);

rl.setPrompt('Calculate>');

rl.prompt();

a = ra.generate\_a();

b = rb.generate\_b();

c = rc.generate\_c();

sign\_a = sign.generate\_sign\_a();

sign\_b = sign.generate\_sign\_b();

summary = cal(a, b, c, sign\_a, sign\_b);

rl.onfline', function(line) (

if(line.trim() == summary) (

console.info(Manswer is right!");

} else (

console.info(Manswer is error! The correct answer is " + summary +

rl.prompt();

a = ra.generate\_a();

b = rb.generate\_b();

c = rc.generate\_c();

sign\_a = sign.generate\_sign\_a(); sign\_b = sign.generate\_sign\_b();

〃随机生成100以内的整数

〃随机生成100以内的整数

〃随机生成100以内的整数

〃随机选取四则运算符

〃随机选取四则运算符

//判断用户给出的运算结果是否正确

〃随机生成100以内的整数 〃随机生成100以内的整数 〃随机生成100以内的整数 〃随机选取四则运算符

〃随机选取四则运算符

summary = cal(a, b, c, sign\_a, sign\_b);

}).on('close', function。{ console.logfCalculate exits!'); process.exit(O);

})；  
/\*\*

\*四则运算方法・cal()

* @param a
* @param b
* @param sign -运算符
* ©returns {\*}

7

function cal(stra, strb, strc, signa, signb) (

var s =

var result = 0;

s = stra + signa + strb + signb + strc;

console.log(s);

result = eval(s); 〃调用eval()方法计算字符串表达式

return result;

}

**【代码分析】**

整个calculate.js脚本文件用于计算四则运算表达式，其中随机生成运算数与运算符是 通过调用几个模块实现的；第04行定义了运算数、运算符和结果所需的几个变量；第05〜 08行引用了随机牛成运算数与运算符的几个模块；从第10行开始，一直到第43行结束， 用于模拟实现一个四则运算的控制台界面应用，读者可以参考第1章最后一节关于该方面 内容的阐述，在此就不做重复解释了：第21〜25行通过调用几个自定义模块，分别随机生 成运算数与运算符，这几个自定义模块下面再详细介绍。

第26行通过调用自定义方法cal()用来计算随机生成的四则运算式的正确结果，并将该 结果保存在变量summary中，变量summary用于在第28行中进行条件判断；第28〜32行 通过条件判断语句，判断用户在控制台给出的答案是否正确，并根据判断结果给出提示信 息；第34〜38行与第21〜25行功能完全一样；第39行与第26行功能完全一样：第51〜 58行就是自定义方法cal()的实现过程，这里应用了 JavaScript语言的eval()方法来实现字符 串运算式的计算。

下面看一下几个自定义模块的实现，用于随机生成运算数的三个模块(random-a.js、 random-b.js> random-c.js)的过程方法基本类似，我们以random-a.js模块为例，其代码如下:

01 /\*\*

02 \* random-a.js

03 \*/

04 vara;

05 console.infof随机生成100以内的整数...')；

06 /\*\*

07 \* exports random a

08 \* @returns (number)

09 \*/

1. exports.generate\_a = function() (
2. a = Math.round(Math.random() \* 100); 〃随机生成 100 以内的整数
3. return a.toString();
4. )

该random-a.js模块应用了 JavaScript语言的Math.round()方法来实现随机运算数的生成。 而用于随机生成运算符模块(sign.js)的实现原理与随机生成运算数的原理类似，其代 码如下：

01 /\*\*

02 \* sign.js

03 \*/

04 var sign\_a, sign\_b;

05 console.info('随机生成运算符...')；

06 exports.generate\_sign\_a = function() (

07 sign\_a = switchSign(Math.round(Math.random() \* 3)); 〃随机选取四则运算符

08 return sign\_a;

09 )

56 •超实用的Node.js代码段

exports.generate\_sign\_b = function() (

sign\_b = switchSign(Math.round(Math.random() \* 3)); //随机选取四则运算符

return sign\_b;

* generate random sign
* @param i
* ©returns {\*}

\*/

function switchSign(i) (

var s;

switch(i) (

case 0:

break;

case 1:

break;

case 2:

break;

default:

s = 7\*;

break;

return s;

2.11是在Ubuntu环境下本例程调试输出的结果。

king5>king-ThinkPad-X230: \*/webstorm-Js/projects/Node JS/chapterOJ/rnypackage

ktng^ktng-ThtnkPad-X23e：~/webstorm-js/projects/NodeJS/chaptere2/mypackage$ node index.js

随机生成以内的整数...

随机生成1的以内的85数•・・

随机生成1阳以内的整敬..・ 随机生成运算符..・ Calculate start!

Calculate> 38-54+43

27

answer is right!

Calculate\* 49-77\*41

123

answer is error! The

Calculate> 91\*73-74

321

answer is error! The

Calculate〉 90\*44\*43

333

answer is error! The

Calculate〉 95-33\*78

146

answer is right!

Calculate〉 86-52-86

Calculate：\* -58

answer Is right I

Calculate\* 37+25/31

12321

answer Is error! The

Calculate〉 97/8S-36

correct

correct

correct

correct

answer Is -3168.

answer is 6S69.

answer ts 4083.

answer is 37.886451612903224.

图2.11基于包管理的四则运算应用调试结果

说明：JavaScript语言的eval()方法可以用来计算字符串表达式，并执行其中的JavaScript 代码 需要读者注意的是，eval()方法只接受原始字符串作为输入参数，如果输入参数非原 始字符串，则该方法将不会做任何改变的直接返回”更详细的解■释说明读者可以参见 JavaScript的官方网站「＞

第3章 Node.js异步I/O与

Async流程控制库

绝大部分开发人员在初次接触Node.js框架的时候，首先了解到的就是该框架的最突出 特点：异步I/O和事件驱动。毋庸置疑，异步I/O和事件驱动确实是Node.js框架令开发人 员心潮澎湃之处。Node.js框架拥有这样的先天优势，随着其不断的改进，一定会在Web 服务端大放异彩。

关于事件驱动我们会在后续的章节做详细介绍，本章的重点就是Node.js异步I/O编程。 由于异步的高效性，Node.js框架在设计之初就被考虑作为一个高效的Web服务器，因此 异步机制贯穿于整个Node.js框架的编程模型中。

Node.js框架的异步I/O编程可以分为以下几类内容：异步I/O机制、异步I/O应用和 Async流程控制库应用。本章将从这几类内容入手，向读者介绍一些非常实用的应用例程。

本章主要包括以下内容。

* Node.js框架异步I/O机制。
* Node.js框架异步I/O应用。
* Ansyc流程控制库。

1. Node.js框架异步机制基础

既然说到Node.js框架异步I/O编程，那就要先了解什么是异步编程。所谓异步编程是 指由于异步I/O等因素，在无法同步获得执行结果时，在回调函数中进行下一步操作的代 码编写风格。常见的异步编程方式有Ajax异步请求、通过setTimeout()方法设定回调函数 以及事件监听等。

在本节这个例程中，我们使用setTimeoutO方法来实现一个异步打印输出控制台内容的 应用。本例ch03.asyn-basic.js主要代码如下：

01 /\*\*

02 \* ch03.asyn-basic.js

03 \*/

04 console.info("\nn);

05 console.infoC Node.]s 异步•编程:基础初步！ ');

06 console.info("\nH);

07 *II*使用setTimeout()＞步方法初探异步机制

08 setTimeout(function(){

09 console,log('asyn - print it now!\*); // 在回调函数内输出信息

1. ),3000);
2. console.logfasyn - print it 3 second later!'); // 异步方法后输出信息
3. console.infoOn");
4. console.infof Node.js 异步编程 \*);
5. console.info("\n");

**【代码分析】**

第08〜10行使用setTimeout。方法实现了一个异步回调过程，在回调过程中通过第09 行向控制台打印输出了一个字符串('asyn-print it now!')，顾名思义，就是要马上打印输 出的意思；第11行向控制台打印输出了另一个字符串('asyn - print it 3 second later!'),其 含义是延迟3秒打印的意思。

那么该例程执行后效果如何呢？图3.1演示的是本例程在Windows环境下，使用 WebStorm开发工具调试输出的结果。



"D:\Program Files\JetBrains\WebStorm 9.0.l\bin\run debugger listening on port 3462

Node.js异步编程:基础初步！



asyn - print i t 3 second later I

Node. j s异步编程

asyn - print it now!

Process finished with exit code 0

图3.1异步打印输出控制台内容

从上图输出的结果可以看到，第11行的字符串('asyn - print it 3 second later!')首先被 打印输出了，而第09行的字符串Casyn-print it now!')是之后打印输出的。希望延迟打印 输出的先被执行了，而希望马上打印输出的却延迟执行了，原因就是第09行是通过 setTimeout()方法的异步回调方式执行的。

提示：关于setTimeout()方法的内容在前面的章节有过使用和介绍，读者也可以自行 参考HTML和JavaScript的官方文档，里面有详细的讲解、另外，细心的读者在调试本例 程源码时就会发现，第09行的字符串(，asyn - print it now!，)打印输出时会有一个明显的3 秒延迟，

1. Node.js框架异常捕获机制

Node.js框架异步I/O的实现主要有两个阶段。

・第一个阶段是提交请求。

・第二个阶段是处理结果。

事实上，这两个阶段彼此并不关联，而异常也并不一定发生在请求提交(即调用函数) 时，因此平常的try/catch语句并不能有效地捕获到程序的异常，也就是发生了失效捕获。

那么如何在Node.js框架异步I/O编程中正常地实现异常捕获功能呢？在本节通过两个 小例程的对比，告诉读者实现异常捕获的正确方法。

第一个小例程是异步I/O捕获失效的情况，该例程ch03.asyn-try-wrong.js的主要代码 如下；

01 *I\*\**

02 \* ch03.asyn-try-wrong .js

03 \*/

04 console.infof Node.js 异步编程:异常捕获！ ')；

05 /\*\*

06 \*错误方式

07 \*/

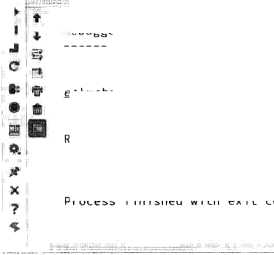
08 try(

09 setTimeout(function()(

1. var data = a/0; //错误的计算
2. },1000);
3. }catch (eX
4. console.log(e);
5. }
6. console.info();

**【代码分析】**

第08〜14行使用try/catch语句实现了一个异常捕获功能，其中在第08~11行使用



f 9 ，

**«r** 国 **Console k** 窟 **Scripts**

"D:\Program Files\JetBrains\WebStorm 9.GJ.l\bin\runnerw.exe" D

debugger listening on port 3367

**Node.js^**步编程：异常捕获！

»

p； \ivebstonn-wQrkspacelNQdeJS、c/?a£>ter@3\ chG3, a.svn- trv-wrong, js var data - a/Q ； // TODO：错误的计算

ReferenceError: a is not defined

at null. \_onTimeotlt /e; \webstom-workseac"NodeWS\cEoter。 at Timer.listOnTimeout [as ontimeout] (timers.js:112:15)

Process finished with exit code 8

图3.2发生问题的异常捕获机制

setTimeout()方法完成了一个异步 回调过程，在该过程中执行了第 10行错误的表达式计算(a/0)； 因为第10行的计算并不会马上被 执行，所以即便是计算发生了错 误,try/catch语句也无法捕获到相 关信息。

那么该例程执行后效果如 何呢？图3.2演示的是本例程在 Windows 环境下，使用 WebStorm 开发工具调试输出的结果。

从上图输出的结果可以看到，第10行的错误计算并没有被try/catch语句很好地捕捉到， 因此会打印出一些系统错误提示信息。

第二个小例程是异步I/O捕获正确的情况，该例程ch03.asyn-try-right.js的主要代码 如下：

01 /\*\*

02 \* ch03.asyn-try-right.js

03 •/

04 console.info(~— Node.js 异步编程:异常捕获！ ，)；

05 /\*\*

06 \*正确方式 ’

07 7

08 setTimeout(function(X

09 try{

1. var data = b/0; //错误的计算
2. }catch(e)(
3. console.log(e);
4. )
5. ),1000);
6. console.info();

**【代码分析】**

第08〜14行使用setTimeoutO方法完成了一个异步回调过程，在该回调过程中使用 try/catch语句实现了 -个异常捕获功能，其中在第10行执行了错误的表达式计算(b/0)； 因为第09〜12行的try/catch语句并不会马上被执行，而是与第10行的错误代码一同被执 行，因此try/catch语句正确地捕获到了相关信息。

那么该例程执行后效果如何呢？图3.3演示的是本例程在Windows环境下，使用 WebStorm开发工具调试输出的结果。

\_ a l

Debugger 回 Console 食 Scrg tS 7 Z F h ■

» t "D:\Program Files\JetBra\*ins\WebStorm 9.6.l\bin\ II . debugger listening on port 3506

■寻 Node.js异步编程；异常捕获！

0 二

巳 [ReferenceError: b is not defined]

*9'*旨

够 疆 Process finished with exit code Q

*s* **El**

**图**3.3**正确的异常捕获机制**

从上图输出的结果可以看到，第10行的错误计算被try/catch语句很好地捕捉到了，因 此会打印出正确的异常提示信息。

说明：将可能引发异常的代码节放在try块中，而将处理异常的代码放在catch块中， catch块是一系列以关键字catch开头的语句，语句后跟异常类型和要执行的操作*。*

1. Async串行流程控制

Async是一个基于JavaScript语言的流程控制库，专为Node.js框架而设计，同时也可 以直接在浏览器中使用。

Async流程控制库提供了简单而强大的异步功能，一共包括大约20个函数。比较常用 的函数有map> reduce> filter•和forEach等，异步流程控制模式包括串行(series)、并行 (parallel)和瀑布(waterfall)等。

本节我们先从Async流程控制库中最基本的串行流程控制开始讲起，所谓串行流程控 制就是指串行执行一个函数数组中的每个函数，且需要每一个函数执行完成之后才能执行 下一个函数，Async库串行流程控制是通过series函数来实现的。

本节第一个例程ch03.asyn-series-array.js的主要代码如下：

01 /\*\*

02 \* ch03.asyn-series-array.js

03 \*/

04 console.infof Node.js 异步编程：Async series ');

05 console.info();

06 var async = require('async'); 〃弓| 用'async'包

07 /\*\*

08 \*使用series函数方法

09 7

1. async.series([
2. function(callback) (
3. callback(null, 'hello');
4. ),
5. function(callback) (
6. callback(null, 'async');
7. ),
8. function(callback) {
9. callback(null, 'series');
10. }
11. ],function(err, results) {
12. console, log(results);
13. })；
14. console.info();
15. console.info(' Node.js 异步编程：Async series ');

**【代码分析】**

第06行通过require('async')方法引用Async流程控制库，引用后的对象名称为async； 第10〜22行使用async.series()函数实现串行流程控制过程;我们看到第11〜13行、第14〜 16行和第17〜19行定义了一个包含三个函数的函数组，读者应该有注意到这个函数组是 通过[]中括号(从第10行到第20行)包含在一起的，实际上这个函数组是async.series() 函数的第1个参数；第20〜22行定义的函数是async.series()函数的第2个参数，其实也是 回调函数，回调函数中的第21行打印输出result参数(此时result参数为['hello', 'async', 'series'])*。*

关于async.series()函数的语法说明如下：

语法：series(tasks,callback);

其中，第1个参数tasks可以是一个数组也可以是一个JSON对象，参数类型不同，影 响的是返回数据的格式；第2个参数callback是一个回调函数，用于执行async.series。函数 完成后的操作。

那么该例程执行后效果如何呢？图3.4演示的是本例程在Windows环境下，使用 WebStorm开发工具调试输出的结果。

北 Detougg« 0 Console 匾 Scripts -•\* 些 M Z 7 亍 > 戛

"O:\Program Files\JetBrains\WebStorm 9.0.1\bin\

debugger listening on port 2766

Node.js异步编程：Async series

[,hello', •async\* , 'series1 ]

Node.js异程：Async series

Process finished with exit code 0

图3.4 Async串行流程控制(数组格式)

前一个例程async.series()函数的第1个参数是一个数组，那么如果是JSON数据格式的 参数会是什么结果呢？

我们通过第二个例程 ch03.asyn-series-json.js 向读者演示一下，ch03.asyn-series-json.js 的主要代码如下：

01 /\*\*

02 \* ch03.asyn-series-json.js

03 7

04 console.infof—— Node.js 异步编程:Async series ');

05 console.info();

06 var async = require('async'); //引用'async'包

07 /\*\*

08 \*使用series函数方法

09 \*/

1. async.series((
2. one: function(callback) (
3. callback(null, 'hello');
4. },
5. two: function(callback) (
6. callback(null, 'async');
7. },
8. three: function(callback) (
9. callback(null, 'series');
10. }
11. ),function(err, results) (
12. console.log(results);
13. })；
14. console,info();
15. console.infof Node.js 异步编程：Async series ');

我们看到第11〜13行、第14〜16行和第17〜19行定义了一个包含三个JSON数据格 式的函数组，读者应该有注意到这个JSON数据格式的函数组是通过{}大括号(从第10行 到第20行)包含在一起的，这个就是async.series。函数的第1个参数；此时第20〜22行 中的 result 参数的值为{ one: \*hello\ two: 'async\*, three: 'series\* },这其实是一个 JSON 格式的 数据。

图3.5演示的是本例程在Windows环境下输出的结果。

Debug ，三： ~ 上

# Debugs S Comok § Soipte ■\*\*, vS '• •\* «l ®

* t "O:\Program Files\JetBrain$\WebStorm 9.9.l\bin\

II $ debugger listening on port 2815

* Node.js异步编程：Async series

*e c*

e { one: 'hello', two: , async', three: 'series' }

**B：**济

S 技 Node.js异步编程：Async series

. Process finished with exit code 9

图3.5 Async串行流程控制(JSON格式)

说明：在应用Async流程控制库过程中，async.series。函数是完全严格按照串行流程来 执行的，从这层意义上来说，它完成的是同步操作的功能。

1. Async瀑布模式流程控制

本节介绍一下Async瀑布模式流程控制waterfall()函数。其实waterfall()和series()函数 在功能上有很多的相似之处，基本上都是按顺序依次执行一组函数，但不同之处是通过 waterfall()这个函数产生的值，都将传给下一个函数，而series。函数则没有这个功能。

本例程ch03.async-waterfall.js的主要代码如下：

01 /\*\*

02 \* ch03.async-waterfall.j$

03 \*/

04 console.info(' Node.js 异步编程:Async waterfall ');

05 console.info();

06 var async = require('async'); //引用'async'包

07 /\*\*

08 \*使用waterfall函数方法

09 7

1. async.waterfall([
2. function(callback) (
3. //当回调函数的第一个参数为非空值时，waterfall会停止执行剩余任务
4. callback(null, 1);
5. },
6. function(data, callback) (
7. console.info(data);
8. callback('test', 2);
9. },
10. function(data, callback) (
11. console.info(data);
12. callback(null, 3);
13. }
14. ],function(err, results) (
15. console.log(results);
16. })；
17. console.info();
18. console.info(' Node.js 异步编程：Async waterfall ');

**【代码分析】**

第06行通过requireCasync1)方法引用Async流程控制库，引用后的对象名称为async； 第10〜25行使用async.waterfall()函数实现瀑布模式流程控制；我们看到第11〜14行、第 15〜18行和第19〜22行定义了一个包含三个函数的函数组，读者应该有注意到这个函数 组是通过[]中括号(从第10行到第23行)包含在一起的，与async.series()g数类同，实际 上这个函数组是async.waterfall()函数的第1个参数。

这里我们需要注意一下第13行，当回调函数callback(null, 1)的第一个参数为非空时， 流程就会在此终止，后面的函数组将不会被执行；因为此处第一个参数为null,因此从第 15行开始的函数组的第二函数将会被继续执行；第23〜25行定义的函数是async.waterfall() 函数的第2个参数，其实也是回调函数，回调函数中的第24行打印输出result参数(此时 result 参数为 1, 2)。

关于async.waterfall()函数的语法说明如下：

语法：waterfall(tasks, [callback]);

其中，第1个参数tasks可以是一个数组对象；第2个参数callback是一个回调函数， 用于执行async.waterfall()函数完成后的操作。

图3.6演示的是本例程在Windows环境下输出的结果。

注意：

1. 需要注意的是waterfall()函数的tasks参数只能是数组类型。
2. 当回调函数的第一个参数为非空值时，waterfall会停止执行剩余任务。

3.5 Async并行流程控制

本节我们介绍一下Async并行流程控制parallel()函数。关于并行的概念读者一定 都不陌生，是指两个或两个以上事件或活动在同一时刻发生。因此，我们可以理解为 parallel()函数是指并行执行多个函数，且每个函数都是立即执行，不需要等待其他函 数先执行。本节通过一个具体例程，来演示一下parallel。函数是如何实现并行流程控 制的。

本节例程ch03.async-parallel.js的主要代码如下：

01 /\*\*

02 \* ch03.async-parallel.js

03 \*/

04 console.info(' Node.js 异步编程:Async parallel ');

05 console.info();

06 var async = require('async'); 〃弓 I 用'async'包

07 /\*\*

08 \*使用parallel函数方法

09 \*/

1. async.parallel([
2. function(callback) {
3. setTimeout(function() {
4. callback(null, 'one');
5. }, 2000);
6. },
7. function(callback) (
8. setTimeout(function() (
9. callback(null, 'two');
10. ), 1000);
11. }
12. ],
13. function(err, results) (
14. console.log(results);
15. })；
16. console.info();

1. console.info(' Node.js 异步编程：Async parallel '); 行)包含在一起的，实际上这个函数组是async.parallel()函数的第1个参数；第22〜24行 定义的函数是async.parallelO函数的第2个参数，其实也是回调函数，回调函数中的第23 行打印输出result参数(此时result参数为数组['one,, \*two\* ])；关于async.parallelO函数的 语法说明如下：



**【代码分析】**

第06行通过require('async')方法引用 Async流程控制库，引用后的对象名称为 async；第 10~24 行使用 async.parallel() 函数实现并行流程控制；我们看到第il­ls 行与第16〜20行定义了一个包含两个 函数的函数组，读者应该有注意到这个函 数组是通过口中括号(从第10行到第23

**Debugger** 回 **Console**

\* "D: \Progratn Fi les\JetBrains\WebStorm 9.0. l\bin\rc

*jj.* debugger Li stening on port 4G80 *寸* Node.j碍步编程：Async waterfall

> ® 2

Process finished with exit code 0

图3.6 Async瀑布模式流程控制

语法:parallel(tasks, [callback]);

其中，第1个参数tasks可以是一个数组或是JSON对象，其与series()函数一样，tasks 参数类型不同，返回的results格式也会不一样；第2个参数callback是一个回调函数，用 于执行async.parallelO函数完成后的操作。

图3.7演示的是本例程在Windows环境下输出的结果。从结果可以看到，由于在第12〜 14行与第17~19行中使用了 setTimeout()方法设定了 1秒的时间延迟，因此第23行打印 输出的result参数是在第26行执行完毕后才完成的。

另外，由于async.parallelO函数是并行流程控制，因此打印输出result参数的过程一共 只耗费了 1秒的时间。

注意：async.parallelO函数传给最终callback的数组中的数据是按照tasks中声明的顺序, 而不是执行完成的顺序。

\_\_ "芸

***9(* Debugger 0 Console** 区 **Scripts** 曜**'>2 "• 1 •\*** 队 **H**

» t "D：\Program Files\JetBrains\WebStorm 9.0.l\bin\rc II + debugger listening on port 4575

■ Node.js异步编程：Async parallel

暮 Node.js异步编程：Async parallel

辱壶 ['one', \* two']

**g**寻

Process finished with exit code 0

*令.*

图3.7 Async并行流程控制

1. Async限制性并行流程控制

本节我们介绍一下Async限制性并行流程控制parallelLimit()函数，所谓限制性并行流 程控制，就是对并行数量限制一个上限，而不是无限制地并发oparallelLimitO函数和parallel() 函数类似，只是多了一个参数limit, limit参数限制任务只能同时并发一定的数量，而不是 全部并发。本节通过一个具体例程，来演示一下parallelLimitO函数是如何实现限制性并行 流程控制的。

本节例程chO3.async-parallelLimit.js的主要代码如下：

01 /\*\*

02 \* chO3.async-parallelLimit.js

03 \*/

04 console.infoC Node.js 异步编程:Async parallelLimit ');

console.info();

05

06

07

08

09

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

var async = require('async'); 〃引 用'async'包 /\*\*

\*使用parallelLimit函数方法

7

async.parallelLimit([

function(callback) ( setTimeout(function() ( callback(null, 'one');

}, 1000);

}.

function(callback) { setTimeout(function() { callback(null, 'two');

), WOO);

} ]，

1, function(err, results) {

console.log(results);

})；

console.info();

console.infof Node.js 异步编程：Async parallelLimit —');

**【代码分析】**

第06行通过require('async')方法引用Async流程控制库，引用后的对象名称为async； 第10~25行使用async.parallelLimit()函数实现并行流程控制。

关于async.parallelLimit()函数的语法说明如下：

语法:parallelLimit(tasks, limit, [callback])输出result参数的过程一共耗费了 2秒的时间。

parallelLimit()函数和parallel()函数类似，但是它多了一个limit参数，limit参数限制同 时并发任务有一定数量上限，而不是无限制并发。我们看第22行的数字1就是limit参数, 本例程设定并行执行的函数数量最大为1个。

图3.8是本例程在Windows环境 下输出的结果。从结果可以看到，由 于在第12〜14行与第17〜19行中使 用了 setTimeout。方法设定了 1秒的 时间延迟，因此第24行打印输出的 result参数是在第26行执行完毕后才 完成的。

另外,由于 async.parallelLimit() 函数的limit参数设定为1,因此打印

B Scripts，•牖 C：饴估 F R 麟

"D:\Program Files\JetBrains\WebSterm 9.0.l\bin\runner debugger listening on port 5145

Node.js异步编程：Async parallelLimit

debugger 回 Console -•

t

Node.js异步编程：Async parallelLimit ['one', 'two\* ]

Process finished with exit code 0

图3.8 Async限制性并行流程控制

1. Async循环流程控制

本节我们介绍一下Async循环流程控制函数，分别是whilst。、doWliilst()> until。、与 doUntil()0读者是不是觉得似曾相识，这些循环流程控制函数基本是按照while, until和 foreach这些循环语句的方式设计的。下面分别通过几个例程逐一介绍Async循环流程控制 函数的使用方法。

第一个例程(ch03.async-whilst.js)用来介绍whilst()函数的用法,有关ch03.async- whilst.js的主要代码如下：

/\*\*

01

02

03

04

05

06

07

08

09

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

* ch03.async-whilst.js

\*/

console.infof Node.js 异步编程:Async whilst

console.info();

var async = require('async');

var count = 0;

/\*\*

* Define JSON Array
* @type ({name: string, age: number}。}

\*/

var list =[

{name:'Jack',age:20},

{name:'Lucy\*,age:18},

{name:'Jack',age:20},

{name:'Lucy',age:18},

{name:'Lucy',age:18}

】；

async. whilst( function () (

return count < 5;

}.

function (callback) { console.log(count); list[count].age += 1; count++;

setTimeout(callback, 1000);

}，

function (err) ( console.log(count); console.log(list);

1. )
2. )；
3. console.info();
4. console.info(\* Node.js 异步编程:Async whilst ');

**【代码分析】**

第06行通过requireCasync)方法引用Async流程控制库，引用后的对象名称为async； 第12〜18行定义了一个名称为list的JSON数组，并初始化了数据；第19〜32行使用 async.whilst()函数实现循环流程控制；第20〜22行用于测试循环结束的条件；第23〜28 行是要异步执行的操作；第29〜32行是回调函数。

关于async.whilst()函数的语法说明如下：

语法：whilst(test, fn, callback)

whilst()函数相当于while语句，但是区别是其中的异步调用将在完成后才会进行下一 次循环。因此当需要循环异步操作的时候，该函数是非常适用的。

test参数是一个返回布尔值结果的函数，通过返回值来决定循环是否继续，作用等同于 while语句循环停止的条件；ft)参数是我们要异步执行的操作，每次fn执行完毕后才会进 入下一次循环；callback参数是whilstQ函数执行完后的回调函数；第25行将list中所有人 的age属性值加1»

注意：当需要以循环的方式执行异步操作时，async.whilst()函数是一个好的选择。

图3.9是本例程在Windows环境下输出的结果。

Debugger 国 Console J'陲 Sai"，•登 山胃 7 F 队 疆

f "D:\Program FiT.es\3etBrains\WebStorm 9.0. l\bin\runner

debugger listening on port 7392

name: 'Jack', name： 'Lucy', name： 'Jack，, name: 'Lucy', name: 'Lucy',

age: age： age: age： age：

21 ),

19

21 },

19 },

19 }

Process finished with exit code

图3.9 async.whilst()循环流程控制

第二个例程(ch03.async-doWhilst.js )用来介绍doWhilst()函数的用法，有关 ch03.async-doWhilst.js 的主要代码如下：

01 /\*\*

02 \* ch03.async-doWhilst.js

03 \*/

04 console.info(' Node.js 异步编程:Async doWhilst \*);

05 console.info();

06 var async = require('async');

07 var count = 0;

08 /\*\*

09 \* Define JSON Array

1. \* @type ((name: string, age: number}。}
2. \*/
3. var list =[
4. (name:'Jack',age:20},
5. (name/Luc/.age: 18},
6. {name:'Jack',age:20},
7. (namei'Lucy'.age: 18),
8. {name:'Lucy',age: 18)
9. ];
10. async.doWhilst(
11. function (callback) (
12. console.log(count);
13. list[count].age += 1;
14. count++;
15. setTimeout(callback, 1000);
16. },
17. function () (return count < 5;},
18. function (err) (
19. console.log(list);
20. }
21. )；
22. console.info();
23. console.info(' Node.js 异步编程:Async doWhilst )

**【代码分析】**

ch03.async-doWhilst.js 与 ch03.async-whilst.js 代码大部分一致，不同的地方是第 19〜29 行使用async.doWhilst(涵数实现循环流程控制。

关于async.doWhilst()函数的语法说明如下：

语法：doWhilst(fn, test, callback)

doWhilst。函数相当于do-•-while语句，相比whilst。函数而言，doWhilst()函数交换了 th和test的参数位置，即先执行一次循环，再做test判断；因此，在需要先执行一次异步 操作再循环的情况下，应用doWhilist()函数是最合适的方法。

图3.10是本例程在Windows环境下输出的结果。

说明：async.until()函数与 async.whilst()函数、async.doUntil()*函数与* async.doWhilst()函 数这两组函数在循环逻辑上正好相反，即当test条件函数返回值为fhlse时继续循环，而返 回值为true时则跳出循环，其他参数与函数特性是完全一致的，

**Mug chOJ**

**Debugger 0 Comole -•❷ Scnptt •\*** 恒**-X i '1 •\* ■**

I "D：\Progiam Files\3etBrains\WebStorm 9-O-lXbinXrunner , debugger listening on port 8444

Node.j程：Async doWhn 1st

Node.js暑步编程：Async doWhilst

[{ name

{ name

(name

{ name

( name

'lack', ,Lucy\*,

'Jack', 'Lucy\*

age： age： age: age:

'Lucy', age：

21 },

19 ),

21 },

19 },

19 )]

Process finished with exit code e

图3.10 async.doWhilst()循环流程控制

1. Async队列流程控制

本节我们介绍一下Async队列流程控制queue()函数。queue()函数可以认为是一个加强 版的parallel。函数，其功能实际上是一个串行的消息队列，通过限制了 worker数量，不再 一次性全部执行。当worker数量不够用时，新加入的任务将会排队等候，直到有新的worker 可用。该函数有多个点可供回调，如worker用完时、无等候任务时、全部执行完时等。本 节通过一个具体例程，来演示queue()函数是如何实现并行流程控制的。

本节例程ch03.async-queue.js的主要代码如下：

01 /\*\*

02 \* ch03.async-queue.js

03 \*/

04 console,infof Node.js 异步编程：Async queue ');

05 console.info();

06 var async = requirefasync1)；

07 /\*\*

08 \*定义一个queue,设worker数量为2

09 \*/

1. var q = async.queue(function(task, callback) (
2. console.log('worker is processing task:task.name);
3. callback();
4. }, 2);
5. /\*\*
6. •独立加入5个任务
7. \*/
8. q.push((name: 'foo'}, function (err) (
9. console.log('finished processing foo');
10. })；
11. q.push({name: 'bar'}, function (err) {
12. console.log(\*finished processing bar');
13. ))；
14. q.push((name: 'cap'}, function (err) (
15. console.log('finished processing cap\*);
16. })；
17. q.push((name: 'egg'}, function (err) (
18. console.log('finished processing egg');
19. })；
20. q.push({name: \*app'}, function (err)(
21. console.logffinished processing app');
22. })；
23. /\*\*
24. \* listen：当最后一个任务交给worker时,将调用该函数
25. \*/
26. q.empty = function() (
27. console.log('no more tasks wjating1);
28. )
29. /\*\*
30. \* listen：当所有任务都执行完以后，将调用该函数
31. \*/
32. q.drain = function() (
33. console.log('all tasks have been processed1);
34. )

【代码分析】

第06行通过requireC'async\*)方法引用Async流程控制库，引用后的对象名称为async； 第10~13行使用async.queue()函数实现队列流程控制。

关于async.queue()函数的语法说明如下：

语法：queue(worker, concurrency);

其中，第1个参数worker是执行任务的回调函数形式，当使用push()方法加入新任务 时,将会调用worker来执行；第2个参数concurrency定义了 worker同时执行任务的数量 上限。

我们看到第17〜19行、第20〜22行、第23〜25行、第26〜28行与第29〜31行使用 q.push()方法独立地加入了 5个任务；第*35〜37*行使用q.empty定义了一个事件监听函数， 即当最后一个任务交给worker时，将调用该函数；第41~43行使用q.drain定义了另一个 事件监听函数，即当所有任务都执行完以后，将调用该函数。

图3.11是本例程在Windows环境下输出的结果。从结果可以看到，第17〜19行、第 20〜22行、第23〜25行、第26〜28行与第29〜31行加入的5个任务全部得到了执行； 当最后一个任务(task.name: app)交给worker时，q.empty事件监听函数被激活，并打印 输出了一行提示信息(\*no more tasks wating')；当全部任务都执行完以后，q.drai事件监听

函数被激活，也打印输出了一行提示信息Call tasks have been processed\*)。

**Debugger** 回 **Console i:**通 **Scripts •\*, i** 曾’**\*'l '1 S'H**

"D:\Program Files\JetBrains\WebStorm 9.0.l\bin\runnerw.exe'

debugger listening on port 62927

Node.js异if•编程：Async queue all tasks have been processed

worker is processing task: finished processing foo worker is processing task： finished processing bar worker is processing task: finished processing cap worker is processing task： finished processing egg no more tasks wating worker is processing task: finished processing app

foo

bar

cap

egg

app

Process finished with exit code 6

图3.11 Async队列流程控制

注意：相对async.parallel()函数而言，async.queue()函数在很多关键点提供了回调处理, 而且push新的任务的功能也是async.parallel()函数所做不到的，由此可见async.queue()函 数的并行处理能力更为强大。

第4章 Buffer模块处理

众所周知，Node.js框架编程是基于JavaScript语言进行开发的，因而IT界也给其起了 一个别名 服务端的JavaScript语言。我们知道，JavaScript语言自身仅仅支持Unicode 字符串数据类型，还不能很好地支持二进制数据类型。因此，Node.js框架的设计者针对这 个情况进行了改进，提供了一个与字符串对等的全局核心模块Buffer来让Node.js框架能 够很好地操作二进制数据类型。

所谓Node.js框架的Buffer这个概念，我们可以通俗地理解为缓冲区，也就是"临时 存贮区”的含义，是用来暂时存放输入、输出数据的一小块内存。如果读者学习过C语言 编程，对于指针数组的概念有一定的了解的话，那么学习和掌握Node.js框架的Buffer就 会容易很多。

本章主要包括以下内容。

. Buffer基本操作。

* Buffer内存分配。
* 字符串转Buffer。
* Buffer转字符串。

・Buffer的拼接方法。

1. 定义Buffer对象的基本方法

Node.js框架为开发人员提供了多种定义Buffer对象的方法，开发人员可以根据实际需 要选择最简洁实用的一种方法来定义并初始化Buffer对象。本节通过一个实例，向读者演 示定义并初始化Buffer对象的基本过程。

本例ch04.buffer-define.js主要代码如下：

01 /\*\*

02 \* ch04.buffer-define.js

03 7

04 〃定义并初始化Buffer对象一 buffer16

05 var bufferl 6 = new Buffer([0x6e, 0x6f, 0x64, 0x65, 0x6a, 0x73]); 〃数组形式

06 console.log("The variable bufferl 6's length is" + bufferl 6.length +

07 console.log(\*The variable bufferl 6 is" + bufferl 6 +

08 〃使用数组下标形式读取Buffer对象

09 for(var i=0; i<buffer16.length; i++)

1. (
2. console.logC'bufferier + i + "] is" + buffer16[i]);
3. console.log("buffer16[" + i + T.toString is " + buffer16[i].toString());
4. )
5. console.log(buffer16.toString('utf8')); Z/utf8 编码
6. console.log(buffer16.toString('hex')); //hex 编码
7. 〃定义并初始化Buffer对象 一bufferUTF8
8. var bufferUTF8 = new Buffer('nodejs', 'utf8'); 〃编码形式
9. console.logf'The variable bufferUTF8's length isn + bufferUTF8.length +
10. console.log("The variable bufferllTF8 isM + bufferUTF8 +
11. 〃使用数组下标形式读取Buffer对象
12. for(var i=0; i<bufferUTF8.length; i++)
13. (
14. console.log(MbufferUTF8r + i + ”] is ” + bufferUTF8[i]);
15. console.log("bu1ferUTF8r + i + "].toString is" + bufferUTF8[i].toString());
16. }
17. console.log(bufferUTF8.toString('utf8,)); //utf8 编码
18. console.log(bufferllTF8.toString('hex,)); //hex 编码

**【代码分析】**

以上代码分别使用了两种方法对Buffer对象进行了定义和初始化，并且对其进行了 一 些简单的测试操作，让读者对Buffer对象有「第一步的感性认识。

第05行代码使用了数组形式定义并初始化了一个Buffer对象，其变量名为buffer!6, 我们之所以使用这个变量名称，主要是因为初始化数组里的数据均为16进制数值。这种方 法其实是Node.js官方文档中推荐的一种定义方法，其语法形式如下：

语法：new Buffer(array)

第06行代码使用console.log。方法打印了 buffer 16.length这个变量，实际上是打印输 出了 Buffer对象的长度(本例中buffer 16.1ength的值等于6,读者可以参见图4.1的调试输 出结果)，关于bufferl6.1ength的语法说明如下：

语法：buf.length // Note: length is Number

第07行代码使用console.log。方法直接打印输出了变量buffer 16,从图4.1的调试输出 结果中可以看到，实际输出了字符串nodejs,读者返回第05行代码研究一下就会发现那几 个16进制计算机编码正好是字符串nodejs,这是因为Node.js框架对于Buffer对象的定义 默认就是utf8格式的编码。

我们再看第09〜13行代码，在这个for循环代码中，我们将变量bufferl6按照数组的 形式逐个打印输出，从图4.1的调试输出结果中可以看到，第05行代码定义的16进制编 码转换为十进制数值进行了输出。

那么如何打印输出定义时的16进制数值呢？我们对比一下第14行与第15行代码，这 两行代码同样使用了 Buffer对象的toString()方法，区别在于参数不同。关于buffer 16.toString()的语法说明如下：

语法：buf.toString([encoding][, start][, end])

其中，encoding参数表示编码方式，该参数是可选的，默认编码方式为"utf8" =

从图4.1的调试输出结果中可以看到，在使用hex作为参数时，输出的结果为初始化 时的16进制编码(具体值为6e6f64656a73)；第17行代码则是使用编码形式定义并初始 化了一个Buffer对象，其变量名为bufferUTF8,顾名思义是因为初始化时直接定义其为 UTF8编码格式；之后第18-27行代码与前面第06〜15行代码类同，均是对bufferUTF8 对象的具体测试操作。

**Debug chM buHcn define^**

**® Debugget** 回 **Conwte** 窟 **Saipts -♦**喧= **S**

The variable buf-ferlS\*s length is 6. The variable bufferl6 is nodejs. bufferl6[0] is 110 bufferl6[0].toString is 110 bufferl6[l] is ill buffer16[1].toString is 111 bufferl6[2] is 100 bufferl6[2].toString is 100 buffer16(3] is 101 bufferl6[3].toString is 101 bufferl6[4] is 106 bufferl6[4].toString is 106 bufferl6[5] is 115 bufferl6[5].toString is 115 nodejs 6e6f64656a73

**II**

■吕

**C 3**

部昏

**\* ®**

碧**Q**

The variable bufferUTF8,s length is 6. The variable bufferUTF8 is nodejs. bufferUTF8[0] is 110 bufferUTF8(0].toString is 110 bufferUTF8[l] is 111 bufferllIF8[l) - toString is 111 bufferUTF8[2] is 100 bufferUTF8[2].toString is 100 bufferUTF8(3] is 101 bufferUTF8(3]-toString is 101 bufferUTF8[4] is 106 bufferUTF8[4].toString is 106 bufferUTF8[5] is 115 bufferUTF8[5].toString is 115 nodejs 6e6f646S6a73

图4.1 WebStorm开发工具下的调试输出效果图

提示：关于encoding参数的详细说明读者可以参考官方文档，官方文档提到了以下几 种编码方式：asciiutf8、utfl61ex base64、binary、hex,本节用到了最常用的 utf8 与 hex 这两种编码方式D

1. 定义Buffer对象的特别方法

上一节我们向读者介绍了定义并初始化Buffer对象的两种基本方法，本节再向读者介 绍另一种特别的方法。该方法首先定义Buffer对象的大小，后续根据需要再进行初始化。

本例ch04.buffer-definesize.js主要代码如下：

01 /\*•

02 \* ch04.buffer-definesize.js

03 \*/

04 〃定义并初始化Buffer对象 ―buffer

05 var buffer = new Buffer(HThis is Buffer", Hutf8M); //初始化 buffer

06 console.log("The variable buffer's length is ” + buffer.length);

07 console.log(MThe variable buifer is" + buffer);

08 〃定义并初始化Buffer对象 —buffed6

09 var buffed 6 = new Buffer(16);

1. console.logf'The variable buffer16\*s length is" + buffer16.length);
2. console.log("The variable buffer16 is " + buffer16);
3. buffer16.write(MThis is Buffer11, 0, //初始化 buffer16
4. console.logf'The variable buffer16's length isH + buffer16.length);
5. console.logf'The variable buffer16 is " + buffer!6);

**【代码分析】**

以上代码分别使用两种方法对两个Buffer对象进行了定义和初始化，并且通过一些简 单的测试操作来比较这两种方法的异同点。

第05行代码使用编码形式定义并初始化了一个Buffer对象，其变量名为buffer,这里 使用编码方式主要是为了与下面的另一种方式进行对比；第06行代码通过buffer.length打 印输出了变量buffer的长度(本例中buffer.length的值等于14,读者可以参见图4.2的调试 输出结果)；第07行代码则打印输出了变量buffer,从图4.2的调试输出结果中可以看到， 实际输出了字符串This is Buffer,这与第05行代码中对变量buffer的定义是完全一致的； 第09行代码通过new Buffer()方法分配一个新的buffer大小是16的8位字节，其语法说明 如下：

语法：new Buffer(size) // Note: encoding is String, Optional, Default: 'utf8'

第10行代码通过buffer 16.1ength打印输出了变量buffer 16的长度(本例中 buffer 16.1ength的值等于16,读者可以参见图4.2的调试输出结果)，这与第9行代码中对 变量bufferl6的定义是一致的；第11行代码则打印输出了变量bufferl6,从图4.2的调试 输出结果中可以看到，由于此时的变量bufferl6未进行初始化，其输出的结果是一个大小 为16的8字节乱码数据；第12行代码使用buffer.write。方法对变量buffer 16进行初始化， 初始化字符串("This is Buffer")与变量buffer的完全一致，这样方便我们对这两种Buffer 对象的定义方法进行比较。关于buffer.write()方法的语法说明如下：

语法：buf.write(string, [offset], [length], [encoding]) // Note: Return Nubmer

其参数说明如下。

・string： String类型，将要被写入Buffer的数据。

* offset： Number类型，可选参数，默认为0。
* length： Number 类型，可选参数，默认为 buffer.length - offset。
* encoding： String类型，可选参数，默认为'utf8'o

之后第13~14行代码与前面第06~07行代码类同，均是对变量buffer 16的具体测试

操作，第13行代码是输出变量buffers的大小，第14行代码是输出变量buffer 16的数据。 图4.2是本例程在Windows环境下输出的效果。



Debugger (3 Console 屈 Scripts -»\* 曜 7 玲『S

\*\*D: \Program Files\JetBr3ins\WebStorm 9.0-l\bin\runnerw.exe" debugger listening on port 16676

The variable buffer's length is 14

The variable buffer is This is Buffer

The variable bufferl6\*s length is 16

The variable bufferl6 is S

The variable bufferl6's length is 16

The variable bufferl6 is This is Buffer

■目

**C E**

**• ffi**

Process finished with exit code 0

图4.2 WebStorm开发工具下的调试输出效果图

提示:本例程中我们使用两种方法对两个不同的Buffer对象初始化完全一样的字符串， 第一种方法的length对象返回的是该字符串的实际大小，而第二种方法的length对象返回 的是定义好的固定大小，与字符串实际大小无关，此时Buffer对象将会自动补齐数据长度。

1. 如何判断Buffer对象

开发人员在很多情况下，可能无法判断某个变量是否为有效的目标对象，这样往往会 给编程带来一些困难。不过对于Node.js框架下Buffer对象来说，大家不用为此担心，因 为Node.js框架为开发人员提供了一个名称为Buffer.isBufferQ的判断方法，通过该方法可以 判断一个目标对象是否为有效的Buffer对象。下面通过一个简单的例程，向读者演示一下 Buffer. isBuffer()的应用方法。

本例ch04.buffer-isBuffer.js主要代码如下：

01 /\*\*

02 \* chO4.buffer-jsBuffer.js

03 \*/

04 console.infof Buffer.isBuffer() ");

05 console.info();

06 var buffer = new Buffer('nodejs', 'utf8'); 〃编码形式

07 〃判断是否为Buffer对象

08 if (Buffer. isBuffer(buffer))

09 (

1. console.info("The 'buffer1 is a Buffer obj.w);
2. )
3. else
4. (
5. console.info("The 'buffer\* is not a Buffer obj.");
6. )
7. var str = "nodejs"; 〃定义字符串变量
8. 〃判断是否为Buffer对象
9. if(Buffer.isBuffer(str))
10. (
11. console.info("The 'str\* is a Buffer obj.");
12. )
13. else 二广匚
14. (
15. console.info(HThe 'str' is not a Buffer obj.°);
16. }
17. console.info();
18. console.info(H Buffer.isBuffer() ");

**【代码分析】**

以上代码分别定义了数据内容完全相同的两个变量，其中一个为Buffer对象，另一个 为字符串变量，我们通过Buffer.isBufferO方法来检验这两个变量是否为有效的Buffer对象。

第06行代码使用编码形式定义并初始化了一个Buffer对象，其变量名为buffer,数据 内容为nodejs；与之对比的，在第16行代码定义了一个字符串变量，其变量名为str,数据 内容同样为nodejs；然后，分别在第08-15行与第18-25行代码通过Buffer.isBufferO方 法来判断这两个变量是否为有效的Buffer对象，关于Buffer.isBuffer()7j法的语法说明如下:

语法：Buffer.isBuffer(obj) *II* Note: Tests if obj is a Buffer

该方法返回一个布尔值，用来表示obj对象是否为一个有效的Buffer对象。

图4.3是本例程在Windows环境下输出的效果。

Debug 巾04 buffer缶航浪淳 \_露~ X.

黎■ Orbu^er 团 Console 匾 Scripto 惶 唔 7『6 ®

穿\* f "D:\Program Files\3etBrains\WebStorm 9.0.1\bin\runnerw.exe"

II 劣 debugger listening on port 1089

嬲 Buffer.isBuffer()

c户

'四 The ' buffer' is *a* Buffer obj . th 膏 The \* strf is not a Buffer obj. e e

B

， Buffer. isBuf f er ()

电 Process finished with exit code 0

图4.3判断是否为一个有效的Buffer对象

提示:Node.js框架除了提供了 Buffer.isBufferO^法夕卜，还提供了一个 Buffer. isEncoding ()方法用来判断是否为有效的编码，读者可以参考Node.js的官方文档进一步了解。

1. 获取Buffer对象字节长度

我们知道Node.js框架下的Buffer对象能够为二进制数据提供很好的支持，那么获取 一个Buffer对象真实的字节长度则是必须要用到的功能了。Node.js框架为开发人员提供了 一个Buffer.byteLength()方法，下面借助一个官方文档提供的例程，向读者演示一下该方法 的使用过程。

本例 chO4.buffer-byteLength.js 主要代码如下：

01 /\*\*

02 \* ch04.buffer-byteLength js

03 \*/

04 console.infoC-——Buffer.byteLength() ”)；

05 console.info();

06 str = '\u00bd + \u00bc = \u00be1; 〃定义字符串

07 //% + \* = %: 9 characters, 12 bytes

08 console.log(str +":n + str.length +" characters,n + Buffer.byteLength(str, 'utf8') + " bytes");

09 console.info();

1. console.info(" Buffer.byteLength() ");

**【代码分析】**

第06行代码定义并初始化了一个字符串变量，其变量名为str,数据内容为\u00bd + \u00bc = \u00be,读者可以去相关网站查阅一下这几个16进制编码，\u00bd代表字符”％", \u00bc代表字符"%", \u00be代表字符"久";然后，在第08行代码通过打印输出str.length 属性来显示字符串变量str的长度，通过Buffer.byteLength()方法来显示字符串变量str的真 实字节长度，关于Buffer.byteLengthO方法的语法说明如下：

语法：Buffer.byteLength(string[, encoding])

该方法返回一个Number数字，用来表示string参数的真实字节长度，encoding参数默 认为” utf8”编码格式。

从图4.4中显示的结果可以看到，字符串str的length属性为9个字符长度，而占用字 节长度为12个，因此我们可以知道叫和"％”这3个字符其实占用了两个字节的长度。

Debug cbOUxiffer-byteLength.js *i. |*

群 Debugger 回 Console 0 Scripts -\*•些 Z 丁 ' L ■

\* t "D:\Program Files\3etBrains\WebStorm 9.0.l\bin\runnerw.exe" I

II . debugger listening on port 1129

. 一. Buffer. byteLength ()

**C**斗

r1 X + % = %: 9 characters, 12 bytes

为辱

鼻 盘 Buffer . byteLength()

囹 13 Process finished with exit code 0

*d*

x

图4.4获取Buffer对象字节长度的方法

提示：本节我们需要了解字符与字节这两个概念的异同，在计算机编码中一个字节占 用8 bit ( 1 byte = 8 bit),而一个字符可能是一个单字节字符，也可能是双字节字符、另夕卜， Buffer.byteLength()方法在写http响应头时经常要用到，如果想改写http响应头Cotent-Length 时，记得一定要用 Buffer.byteLength()方法，而不要使用 String.prototype.length 属性。

4.5读取Buffer对象

Node.js框架为开发人员提供了一组功能强大的读取Buffer对象的方法，这些方法可以 使得Buffer对象能够很友好地完成一些复杂的二进制操作，这就极大地方便了开发人员对 Buffer对象的使用。下面我们在官方文档提供的示例的基础上，向读者演示这一组方法如 何使用。

本例ch04.buffer-read.js主要代码如下：

01 /\*\*

02 \* ch04.buffer-read.js

03 \*/

04 console.info(" Buffer Read —");

05 console.info();

06 var str\_readUlnt8 = "M;

07 var str\_readlnt8 =

08 var str\_readUlnt16LE =

09 var str\_readlnt16LE =

1. var str\_readUlnt16BE =
2. var str\_readlnt16BE =
3. var but = new Buffer(4);
4. buf[0] = 0x6;
5. buf[1] = 0x8;
6. buf[2] = 0x23;
7. buf[3] = 0x57;
8. str\_readUlnt8 += Hbuf.readUlnt8(i) is :”；
9. for (i=0; i<buf.length; i++) (
10. str\_readUlnt8 += buf.readlllnt8(i).toString(16) +
11. }
12. console.log(str\_\_readUlnt8);
13. console.info();
14. str\_readlnt8 += Mbuf.readlnt8(i) is :
15. for (i=0; i<buf.length; i++) {
16. str\_\_readlnt8 += buf.readlnt8(i).toString(16) +
17. }
18. console.log(str\_readlnt8);
19. console.info();
20. str\_readUlnt16LE += nbuf.readUlnt16LE(i) is :
21. for (i=0; i<buf.length-1; i++) (
22. str\_readUlnt16LE += buf.readUlnt16LE(i).toString(16) +"
23. )
24. console.log(str\_readlllnt16LE);
25. console.info();
26. str\_readlnt16LE += nbuf.readlnt16LE(i) is :
27. for (i=0; i<buf.length-1; i++) {
28. str\_readlnt16LE += buf.readlntl 6LE(i).toString(16) +w ”；
29. )
30. console.log(str\_readlnt16LE);
31. console.info();
32. str\_\_readUlnt16BE += Hbuf.readUlnt16BE(i) is : ”；
33. for (i=0; i<buf.length-1; i++) {
34. str\_readlllnt16BE += buf.readlllntl 6BE(i).toString(16) + "
35. )
36. console.Iog(str\_readUInt16BE);
37. console.info();
38. str\_readlnt16BE += "buf.readlntl6BE(i) is : ”；
39. for (i=0; i<buf.length-1; i++) {
40. str\_readlnt16BE += buf.readlntl 6BE(i).toString(16) + "
41. }
42. console.log(str\_readlnt16BE);
43. console.info();
44. console.info("buf.readUlnt32LE(i) is :" + buf.readlllnt32LE(0).toString(16));
45. console.info();
46. console.info("buf.readUlnt32BE(i) is : " + buf.readUlnt32BE(0).toString(16));
47. console.info();
48. console.info("buf.readlnt32LE(i) is :" + buf.readlnt32LE(0).toString(16));
49. console.info();
50. console.info("buf.readlnt32BE(i) is :M + buf.readlnt32BE(0).toString(16));
51. console.info();
52. console.info(M Buffer Read ");

**【代码分析】**

第06-11行代码定义了本例程后面需要使用的几个字符串变量；第12-16行代码定 义了一个大小为4的8字节Buffer对象，变量名为buf,然后对其进行了初始化；第19行 代码通过buf.readUInt8()方法依次读取了变量buf的每个字节的数据，然后保存在一个字符 串变量(变量名为str\_readUInt8)中并打印输出。

第37行代码通过buf.readlntl6LE()方法依次读取了变量buf的两个字节的数据，然后 保存在一个字符串变量(变量名为str\_readIntl6LE)中；第43行代码通过buf.readUlntl6BE() 方法依次读取了变量buf的两个字节的数据，然后保存在一个字符串变量(变量名为 str readUInt 16BE)中；第49行代码通过buf.readlntl6BE()方法依次读取了变量buf的两个 字节的数据，然后保存在一个字符串变量(变量名为str\_readIntl6BE)中；第53行代码通 过buf.readUInt32LE()方法依次读取了变量buf的4个字节的数据；第55行代码通过 buf.readUInt32BE()方法依次读取了变量buf的4个字节的数据；第57行代码通过 buf.readlnt32LE()方法依次读取了变量buf的*4*个字节的数据；第59行代码通过 buf.readInt32BE()方法依次读取了变量buf的4个字节的数据。

图4.5演示的是本例程在Windows环境下，使用WebStorm开发工具调试输出的效果。



Debugger 国 Console 窿 Scrips \*\* 您儒 \*\* 2

"D:\Program Files\7et0rains\WebStorm 9.0.l\bin\runnerw.exe" debugger listening on port 1118

Buffer Read

buf.readUInt8(i) is : 6 8 23 57

buf.readlnt8(i) is ; 6 8 23 57

buf.readUIntl6LE(i) is : 806

bu€.readIntl6LE(i) is : 806

buf.readUIntlSBE(i) is : 608

buf.readXntl6BE(i) is : 608

2308

2308

823

823

5723

5723

2357

2357

buf.readUInt32LE(i) is : 57230806 buf.readUInt32BE(i) is : 6082357 buf.read!nt32LE(i) *is* : 57230806 buf.readInt32BE(i) is : 6082357

Buffer Read

Process finished with exit code 0

图4.5读取Buffer对象的方法

提示：Node.js框架还提供了读取单浮点型（float）和双浮点型（double） Buffer对象 的方法，使用方法与本例程中应用过的方法类似，读者可以参考Node.js官方文档进一步学 习。

4.6 写入Buffer对象

Node.js框架同样为开发人员提供了一组功能强大的写入Buffer对象的方法，这些写入 方法可以对Buffer对象实现一些复杂的二进制操作，极大地方便了开发人员对Buffer对象 的使用。本节我们同样在官方文档提供的示例的基础上，向读者演示这一组方法如何使用。

本例ch04.buffer-write.js主要代码如下：

01 /\*\*

02 \* ch04 .buffer-write .js

03 \*/

04 console.info(" Buffer Write ・一")；

05 console.info();

06 //buf.write() usage

07 buf = new Buffer(32);

08 len = buf.write('\u00bd + \u00bc = \u00be\*, 0);

09 console.log("buf.write() usage:" + buf.toStringCutf8', 0, len) + "," + len + " bytes");

1. console.info();
2. //buf.writelllnt8() usage
3. var buf8 = new Buffer(4);
4. buf8.writelllnt8(0x5, 0);
5. buf8.writeUlnt8(0x6,1);
6. buf8.writeUlnt8(0x12, 2);
7. buf8.writeUlnt8(0x34, 3);
8. console.log("buf.writeUlnt8() usage:");
9. console.log(buf8);
10. console.info();
11. Z/buf.writeUlntl 6BE() usage
12. var buf16BE = new Buffer(4);
13. buf16BE.writeUlnt16BE(0x1234, 0);
14. buf16BE.writeUlnt16BE(0xabcd, 2);
15. console.log(Hbuf.writeUlnt16BE() usage:");
16. console.log(buf 16BE);
17. console.info();
18. //buf.writelllntl6LE() usage
19. var buf16LE = new Buffer(4);
20. buf16LE.writeUlnt16LE(0x1234, 0);
21. buf16LE.writeUlnt16LE(0xabcd, 2);
22. console.log(nbuf.writeUlnt16LE() usage: °);
23. console.log(buf16LE);
24. console.info();
25. //buf.writeUlnt32BE() usage
26. var buf32BE = new Buffer(4);
27. buf32BE.writeUlnt32BE(0xfeedface, 0);
28. console.log(Mbuf.writelllnt32BE() usage:");
29. console.log(buf32BE);
30. console.info();
31. Z/buf.writeUlnt32LE() usage
32. var buf32LE = new Buffer(4);
33. buf32LE.writeUlnt32LE(0xfeedface, 0);
34. console.log(\*,buf.writeUlnt32LE() usage:");
35. console.log(buf32LE);
36. console.info();
37. console.info(M Buffer Write ”);

**【代码分析】**

第07〜09行代码使用buf.write。方法向变量buf写入了一个字符串('\u00bd + \u00bc = \u00be，)编码，该字符串编码代表一个字符表达式*("A+'/, = %')*,然后将其打印输出；关 于buf.write。方法的语法说明在4.1节有过说明，这里补充一点，buf.write()方法返回一个 Number数字，表示写入的字节数。

从图4.6中显示的结果可以看到，buf.write()方法写入了一个字符串编码，与代码在第 08行定义的完全一致。

第12~18行代码使用buf.writeUInt8()方法分4次向变量buf8中写入了一组16进制编 码，然后在第18行代码中将其打印输出;第21〜25行代码使用buf.writeUIntl6BE()方法

分2次向变量bufl6BE中写入了一组16进制编码，然后在第25行代码中将其打印输出； 第28〜32行代码使用buf.writeUIntl6LE()方法分2次向变量bufl6LE中写入了一组16进 制编码，然后在第32行代码中将其打印输出；第35-38行代码使用buf.writeUInt32BE() 方法分1次向变量buf32BE中写入了一组16进制编码，然后在第38行代码中将其打印输 出；第41-44行代码使用buf.writeUInt32LE()方法分1次向变量buf32LE中写入了一组16 进制编码，然后在第44行代码中将其打印输出。

腿眄 缺""**5," A：**

粉' **Debugg^ g} Console - "**

狰 *-f* "D:\Program Files\UetBrains\WebStorm 9-0.1\bin\runnerw.exe"

li g debugger listening on port 1186

Buffer Write

....督

**0**

隧 buf.wnite() usage: *%+%=%, 12* bytes

挚替

ig 鑫 buf.writeUInt8() usage:

E <Buffer 05 06 12 34>

蠲H

徽 buf.writeUIn-tl68E() usage:

a <Buffer 12 34 ab cd>

X buf.writeUIntl6LE() usage:

? <Buffer 34 12 cd ab>

buf.writellInt32BE() usage:

〈Buffer fe ed fa ce>

buf.writeUInt:32LE() usage:

〈Buffer ce fa ed fe>

Buffer Write

I

Process finished with exit code 0

图4.6写入Buffer对象的方法

提示：Node.js框架同样也提供了写入单浮点型(float)和双浮点型(double) Buffer 对象的方法，使用方法与本例程中应用过的方法类似，读者可以参考Node.js官方文档进一 步学习。

4.7 Buffer对象转字符串

既然我们都知道Buffer对象直接操作字节，这样的效率是比直接操作字符串要快很多 的，因为省去了 Buffer对象转字符串的中间过程。但是，对于习惯于JavaScript编程的开 发者来讲，可能还是会任性地去操作字符串，Node.js框架提供了 buf.toStringO方法来实现 这个功能，下面基于一个官方文档提供的例程，向读者演示一下buf.toStringO的使用方法。

本例ch04.buffer-toString.js主要代码如下：

01 /\*\*

02 \* chO4.buffer-toString.js

03 \*/

04 console.info(H-—- Buffer toString ");

05 console.info();

06 //define a buffer obj

07 buf = new Buffer(26);

08 //init a buffer obj

09 for (var i = 0 ; i < 26 ; i++)(

1. buf[i] = i + 97; //97 是 ASCII 'a\*
2. }
3. 〃输出：abcdefghijklmnopqrstuvwxyz
4. console.info("buf.toString('ascii,):" + buf.toString('ascii'));
5. console.info();
6. 〃输出:abode
7. console.info("buf.toString('ascii', 0, 5):" + buf.toString('ascii', 0, 5));
8. console.info();
9. 〃输出：abcdc
10. console.info("buf.toString(,utf8,, 0, 8):" + buf.toString(,utf8,, 0, 8));
11. console.info();
12. 〃输出：abcde
13. console.info("buf.toString('utf8', 0, 8):" + buf.toString('hex', 0,12));
14. console.info();
15. //\*utf8\*,输出 abcde
16. console.info(Mbuf.toString(undefined, 0,18):" + buf.toString(undefined, 0,18));
17. console.info();
18. console.info(n Buffer toString ");

**【代码分析】**

第07〜11行代码定义了一个Buffer对象，变量名为buf,然后通过for循环使用buf[index] 方式进行初始化，写入了一个包含26个小写字母的字符串编码；第13行代码通过 buf.toString('ascii')方法将Buffer对象buf转为字符串，并使用console.infb()方法打印输出转 化后的字符串，其中参数，ascii，表示以ASCH编码方式进行转换。

从图4.7中显示的结果可以看到，因为使用buf.toString('ascii')方法时，为了定义第二 个与第三个参数，所以打印输出了一个26个小写字母的字符串，与第07-11行代码定义 的完全一致。

雄 Debugger 回 Console 启 Scripts ■»,些Z Z S'■

"D:\Program Files\3etBrains\WebStorm 9.0.l\bin\runnerw.exe- debugger listening on port 2941 Buffer toString

buf.toString('ascii'): abcdefghijklmnopqrstuvwxyz

buf.toString('ascii', 0, 5): abcde

buf.toString(\*utf8', 0, 8): abcdefgh

buf.toString(,hex,, 0, 8): 6162636465666768696a6b6c

buf.toString(undefined, 0, 18): abcdefghijklmnopqr

Buffer toString

Process finished with exit code 0

图4.7 Buffer对象转字符串的方法

提示：当 buf.toString()方法的 encoding 参数为 undefined 或 null 时，直接将 encoding 参数定义为utf8编码格式。

1. Buffer对象裁剪

如果我们想将一个Buffer对象按照需求进行裁剪、生成一个新的Buffer对象，该如何 去做呢？ Node.js框架提供了 buf.sliceQ方法来实现这个功能，下面通过一个例程，向读者 演示一下buf.slice()方法的使用过程。

本例ch04.buffer-slice.js主要代码如下：

01 /\*\*

02 \* ch04.buffer-slice.js

03 7

04 console.info(M Buffer slice ");

05 console.info();

06 /\*\*

07 \* Create a new Buffer obj

08 7

09 var but = new Buffer(26);

1. for(var i = 0; i < 26; i++) (
2. buf[i] = i + 97; //97 是 ASCII 'a'
3. }
4. var buf\_slice\_a = buf.slice(0, 5);
5. console.log(buf\_$li8\_a.toString('ascii', 0, buf\_slice\_a.length));
6. for(var j = 0; j < 26; j++) {
7. buf[j] = 122-j; //122 是 ASCII 'z'
8. }
9. var buf\_slice\_b = buf.slice(0, 5);
10. console.log(buf\_slice\_b.toString(,ascii', 0, buf\_slice\_b.length));
11. console.info();
12. console.info(" Buffer slice ——”)；

**【代码分析】**

第09〜12行代码定义了一个Buffer对象,变量名为buf,然后通过for循环使用buf[index] 方式进行初始化，写入了一个包含26个小写字母的字符串编码；第13行代码通过 buf.slice(0,5)方法将Buffer对象buf裁剪为一个新的Buffer对象buf^slice a,并使用 console.log()方法打印输出，其中参数(0,5)表示将buf对象按照从下标0到5进行裁剪，关 于buf.slice(0,5)方法的语法说明如下：

语法：buf.slice([start][, end]) // Note: Returns a new buffer

参数说明如下。

. start: Number,可选的参数，默认值为0。

• end: Number,可选的参数，默认值为buffer.length。

该方法返回一个新的Buffer对象，这个Buffer对象将会和老的Buffer对象引用相同的 内存地址，只是根据start **(默**认是0)和end **(默**认是buffer.length)的偏移裁剪了索引。 需要注意的是，负的索引是从Buffer尾部开始计算的。

从图4.8中显示的结果可以看到，因为使用buf.slice(0,5)方法对buf对象进行了裁剪， 并生成了一个新的Buffer对象bujslice\_a,所以在第14行代码打印Buffer对象bujslice\_a 时输出了，abcde，这个小写字母的字符串，这与第09-12行代码定义的是完全一致的。



t

abcde zyxw

/usr/local/bin/node chO4.buffer-slice.js Buffer slice

Buffer slice

Process finished with exit code 0

图4.8 Buffer对象裁切的方法

提示：当使用buf.sliceQ方法修改一个Buffer对象、并生成一个新的Buffer对象的slice 切片时，该操作也会改变原来的Buffer对象。因为新老两个Buffer对象引用的是一个相同 的内存地址。

1. 拷贝Buffer对象

Node.js框架为开发人员提供了一个拷贝Buffer对象的方法，其方法名称为buf.copyO。 在Node.js的官方文档中对该方法有详细的解释说明，该方法并没有直接拷贝副本，而是新 建一个长度相等的Buffer,然后在原Buffer对象上进行拷贝。下面通过一个例程，向读者 演示一下buf.copy()方法的使用过程。

本例ch04.buffer-copy.js主要代码如下：

01 /\*\*

02 \* ch04.buffer«copy.js

03 \*/

04 console.info(M Buffer copy ");

05 console.info();

06 *I\*\**

07 \* define Buffer obj

08 \* @type (Buffer)

09 7

1. buf1 = new Buffer(26);
2. buf2 = new Buffer(26);
3. /\*\*
4. \* init a new Buffer obj
5. \*/
6. for(var i=0; i<26; i++) {
7. 匕 buf1[i] = i + 97; //97 是 ASCII 'a'
8. buf2[i] = 35; //ASCII \*#\*
9. )
10. buf1.copy(buf2, 6, 0, 10);
11. console.log(buf2.toStringCascii', 0, 25));
12. console.Iog();
13. /\*\*
14. \* define Buffer obj
15. \* @type (Buffer}
16. \*/
17. buf = new Buffer(26);
18. /\*\*
19. \* init a new Buffer obj
20. 7
21. for(var i=0; i<26; i++) (
22. buffi] = i + 97; //97 是 ASCII 'a'
23. }
24. console.info(buf.toString());
25. console.log();
26. buf.copy(buf, 0, 6, 10);
27. console.log(buf.toString());
28. console.info();
29. console.info(H Buffer copy ■一n);

**【代码分析】**

第10-11行代码定义了两个Buffer对象，变量名分别为bufl和buf2；第15〜18行代 码通过for循环使用buflindex]方式进行初始化，bufl写入了 一个包含26个小写字母的字 符串编码，bufl全部写入字符编码#;第19行代码通过bufl.copy(buf2,6,0,10)方法将bufl 中的部分编码写入到buf2中，关于bufl,copy(buf2,6,0,10)方法的语法说明如下：

语法：buf.copy(targetBuffer, [targetstart], [sourceStart], [sourceEnd])

参数说明如下。

. targetBuffer： Buffer类型对象，将要进行拷贝的Buffero

* targetstart： Number类型，可选参数，默认为0。
* sourceStart： Number类型，可选参数，默认为0。
* sourceEnd： Number 类型，可选参数，默认为 buffer.length。

该方法进行Buffer对象的拷贝，源和目标可以是重叠的。targetstart参数为目标开始偏 移，sourceStart为源开始偏移，默认都是0, sourceEnd为源结束位置偏移，默认是源的长 度buffer.length；如果传递的值是undefined/NaN或者是out of bounds (超越边界)，就将 其设置为默认值。

从图4.9中显示的结果可以看到，本例程使用bufl.copy(buf2,6,0,10)方法将bufl中的从0到10的编码写入到buf2中从第6个字节开始的位置，然后在第20行代码中将buf2以，ascii， 编码方式打印输出，这与第15〜18行代码对变量bufl和buf2的定义是完全一致的。

Debu. chG4.buffer-cofjji.js. *费-i*

棘 Debugger ® Console / 播 Scripts t[[2]](#footnote-3),螭整；儒 [[3]](#footnote-4) 队 ®

» "D:\Program Files\3etBra-ins\WebStorm 9.G. 1\ II 毒 debugger listening on port 8255

Buffer copy

####sabcdefghij ######\*## abcdefghi j klmnopqrstuvwxyz ghijefghij klmnopqrstuvwxyz

Buffer copy

Process finished with exit code 0

图4.9拷贝Buffer对象的方法

提示：相对来说并没有直接拷贝副本的方法，只能新建一个长度相等的Buffer,然后 在原Buffer上调用copy方法，参数中还可以设置copy的开始和结束位置等。

1. 正确拼接Buffer

对于初学Node.js框架的开发人员来说，可能认为Buffer模块易学、重要性也不是那 么突出。其实，Buffer模块在文件I/O和网络I/O中应用非常广泛，其处理二进制的性能比 普通字符串性能要高出很多，重要性可谓是举足轻重。下面我们通过一个例程，向读者演 示一下使用buf.concat()方法进行拼接的过程。

本例ch04.buffer-concat.js主要代码如下：

/\*\*

01

02

03

04

05

06

07

08

09

10

11

12

13

14

15

16

* ch04.buffer-concat.js

\*/

console.infof Buffer concat vs String concat

console.info();

1. var len = 100000 \* buf.length;
2. for(var i=0; i<100000; i++)(
3. list.push(buf);
4. len += buf.length;
5. )
6. /\*\*
7. \* Buffer对象拼接
8. \*/
9. var s1 = Buffer.concat(list, len).toString();
10. console.timeEnd(Mbuffer concat test!11);
11. console.info();
12. console.time(Mstring concat test!");
13. var list = 0;
14. for(var i=100000; i>=0; i-)(
15. list.push(str);
16. }
17. /\*\*
18. \* String对象拼接
19. \* @type {string)
20. \*/
21. var s2 = list.join(M,');
22. console.timeEnd("string concat test!");
23. /\*\*
24. • end record time
25. \*/
26. console.info();
27. console.info(" Buffer concat vs String concat \*');

**【代码分析】**

第10行代码定义了一个Buffer对象，变量名为buf,并初始化了一个字符串数据(“this is Bufferconcat test!")；第11行代码定义了一个字符串变量str,并初始化了一个字符串数 据("this is String concat test!\*\*)；从第15行代码开始到第26行代码结束，通过console.time() 和console.timeEnd。方法完成了一段时间间隔记录；第16〜21行代码定义了一个数组变量 list[],并使用buf变量对该数组变量进行初始化；第25行代码通过Buffer.concat(Iist,len)方 法将list[]数组中的编码重新拼接成一个Buffer对象，关于Buffer.concat(list,len)方法的语法 说明如下：

语法：Buffer.concat(list, [totalLength])

参数说明如下。

• list {Array)：数组类型，Buffer数组，用于被连接。

. totalLength： {Number}类型，第一个参数Buffer数组对象的总大小。

该方法返回一个保存着将传入buffer数组中所有buffer对象拼接在一起的buffer对象; 如果传入的数组没有内容，或者totalLength参数是0,那将返回一个zero-length的buffer： 如果数组中只有一项，那么这第一项就会被返回；如果数组中的项多于一个，那么一个新 的Buffer对象实例将被创建；如果total Length参数没有提供，虽然会从buffer数组中计算 读取，但是会增加一个额外的循环来计算该长度，因此提供一个明确的totalLength参数将 会使得Buffer.concat()方法执行得更快；

从图4.10中显示的结果可以看到，使用Bu任er.concat(list,len)方法进行拼接的耗时为48 毫秒。

**Debug .，Wtw**♦陛竺竺虹 — • ，二 . **4 4**

德 Debug" E Console •• g Scripts \*•幅 f〔I H

* f "D:\Pggram Fi les\JetBrain$\WebStorm 9.0.l\bin\run

11 & debuggei listening on port 2806

■ Buffer concat vs String concat

C r

buffer concat test!: 48ms

\*守

* 血 string concat test!: 24ms

留Q

Buffer concat vs String concat

*d*

，噂 Process finished with exit code 9

图4.10 Buffer对象拼接功能

注意：Buffer.concat(list, [totalLength])方法的第2个参数totalLength比较特别，这里的 totalLength不是数组长度而是数组里Buffer实例的大小总和。

1. 应用 Buffer 操作 HTTP Request Header

在上一节里面，我们测试了 Buffer拼接字符串的性能，相比字符串直接拼接性能还是 差一些。其实，按照Node.js官方文档的说明，Buffer模块的强项在于操作字节流，一般使 用Buffer操作字节流通常会比转化成String效率要高很多。

本节通过一个简单的解析HTTP Request Header的例程，来实际测试一下Buffer操作 字节流的效率。

本例程我们封装成一个包来实现，先看一下该包(ch04.buffer.concat)的目录结构，如 图4.11所示。

Project .三J O ❖ 。•上

**▼ r.] NodeJS ur, •,.，c L ic.jri-** ."'Jl

A 口 **chapterOl**

* **chapterOi**
* **Qj chapter<)3**

**▼ chapterO4**

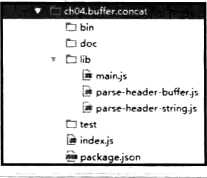


图4.11 ch04.buffer.concat包目录结构示意图 关于包结构的说明读者可以参考第2.10节的内容，下面逐一介绍这几个脚本文件。 其中，index.js定义为包的主入口文件，其代码如下：

01 /\*\*

02 \* index.js

03 \*/ ,

04 var cal = requireC./lib/main\*);

第04行代码调用目录(lib)下的main.js文件。

下面看一下main.js脚本文件，其代码如下：

01 〃\*

02 \* main.js

03 7

04 console.info(" Buffer Parse Header ”)；

05 console.info();

06 /\*\*

07 \* 导入模块 parse-header-buffer.js

08 \* 导入模块 parse-header-string.js

09 \*/

1. var buffer\_parse = requireC./parse-header-buffer^.parse;
2. var string\_parse = require('./parse-header-string,).parse;
3. /\*\*
4. \*定义Buffer对象
5. 7
6. var data = new Buffer('POST /too HTTP/1.1\r\nHost: foo.example.com\r\nContent-Length: 5\r\nConnection:keep-alive\r\nAccept:text/html,application/xhtml+xml,application/xml;q=0.9.\*/ \*;q=0.8\r\nCookie:connect.sid=OY2nKGql3obs5IYee0JKTjhf.FDtbY1Jz5Ngw5So9Jv3MUetl5l TvrlfwgCkRw%2FcXUCk\r\nllser-Agent:Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_7\_2) AppleWebKiV535.7 (KHTML, like Gecko) Chrome/16.0.912.41 Safari/535.7\r\n\r\nq=bar,);
7. /\*\*
8. \*打印输出Buffer对象
9. \*/
10. console.log(Mbuffer\_parse(data):" + buffer\_parse(data));
11. console.info();
12. varn = 1000000;
13. /\*\*
14. \*计算Buffer操作字节流的时间
15. \* @type (Date}
16. \*/
17. var start = new Date();
18. for(var i=0; i<n; i++)(
19. buffer\_parse(data);
20. )
21. console.log('buffer\_parse take:' + (new Date()・ start) + 1 ms');
22. console.info();
23. /\*\*
24. \*打印输出String对象及其长度
25. \*/
26. console.log(Mstring\_parse(data):" + string\_parse(data));
27. console.log("data.length:" + data.length);
28. console.info();
29. /\*\*
30. \*计算String操作字节流的时间
31. \* @type {Date}
32. \*/
33. start = new Date();
34. for(var i=0; i<n; i++) (
35. string\_parse(data);
36. }
37. console.log('string\_parse take:1 + (new Date() - start) + ' ms');
38. console.info();
39. console.info(M Buffer Parse Header ”)；

**【代码分析】**

整个main.js脚本文件通过调用两个模块来比较Buffer和String操作HTTP Request Header字节流的时间效率；第10〜11行代码引用了两个模块的parse方法，分别为parse­header-buffer.js 模块和parse-header-string.js模块；从第15行代码开始定义了一个Buffer对 象，名称为data,并将HTTP Request Header的POST数据赋给data；第26〜30行代码通 过buffer\_parse()方法操作字节流，并通过开始时间与结束时间这两个时间节点的差值，来 计算buffer\_parse()方法操作字节流所用的时间；同样，第42~46行代码与第26~30行代 码功能类似，不同之处就是通过string\_parse()方法操作字节流。

下面看一下parse-header-buffer.js模块，其代码如下：

01 /\*\*

02 \* parse-header-buffer.js

03 \*/

04 var SPACE = 0x20, *IP'*

05 COLON = 0x3a, //58,:

06 NEWLINE = 0x0a, /An

07 ENTER = OxOd; /Ar

08 〃\*

09 \* exports parse function

1. \* @param data
2. \* ©returns {\*}
3. \*/
4. exports.parse = function parse(data) (
5. var line\_start = 0, len = data.length;
6. for(var i=0; j<len; i++) (
7. *II* Host: xxx.abc.com

if(data[i] === COLON) (

17

18

19

20

21

22

23

*24*

*25*

*26*

*27*

28

29

30

31

32

33

34

35

36

37

38

39

｝；

var key = data-toStringCascii\*, line\_start, i).toLowerCase();

i++; //skip

if(key === 'host') (

var value\_start = i;

while(i < len)(

if(data[i] -== ENTER) (

return data.toStringfascii', value\_start, i).trim().toLowerCase(); } if

)

}

} else if(data[i] === ENTER && data[i+1] === NEWLINE) ( i +=' 2;

line\_start = i;

if(data[i] === ENTER && data[i+1] === NEWLINE) {

// \M\r\n

return 'Host header not found';

}

}

)

return null;

该模块导出了一个parse方法，用于实现Buffer对象操作字节流的功能。 而使用字符串操作字节流是通过parse-header-string.js模块来实现：

/\*\*

01 /\*\*

02 \* parse-header-string.js

03 \*/

04

05

06

07

08

09

10

11

12

13

14

15

16

17

18

* exports parse function
* @param data
* ©returns {\*}

/

exports.parse = function parse(data) {

var lines = data.toStringfascii'J.splitOn"); var cut, name, host;

for (var i=0, len=iines.length; i < len; i++) { cut = lines[i].splitC：');

name = cut[0];

if (name ===== 'Host') {

if (cut[1] === undefined) { return \*Host header not found\*;

}

1. host = cut[1].trim().toLowerCase();
2. return host;
3. }
4. )
5. return null;
6. );

而该模块同样导出\ 一个parse方法，用于实现String对象操作字节流的功能。

图4.12是本例程在Windows环境下输出的效果。从图中的数据可以看到，使用Buffer 对象操作字节流的时间效率远高于String对象操作的时间效率。所以对于二进制数据来说， Buffer比String的性能要高出很多，因此建议使用Buffer模块。

Osbug 'ntieqs , 营金

* Debugger E Console ® Scripts 竖 '• •\* ti 徵
* t "D:\Program Files\JetBrains\WebStorm 9.©.l\bin\run

II $ debugger listening on port 21099

H — Buffer Parse Header

S C r buffer\_parse(data): foo.example.com

® ® buffer\_parse take: 2236 ms

gq

string\_parse(data): foo.example.com

data.length: 384

*\**

X string\_parse take： 4031 ms

**7**

Buffer Parse Header

Process finished with exit code 0

图 4.12 应用 Buffer 操作 HTTP Request Header

注意：Node.js框架的Buffer对象的优势是处理字节流，因此对于HTTP Request Header 的POST字节流数据来说，Buffer对象的优势是非常明显的,

第5章 Node.js进程管理

对于操作系统而言，进程是正在运行的程序的载体，这个载体包括了程序中所占用的 全部系统资源，比如CPU (寄存器)、I/O、内存、网络资源等。因此，对进程进行有效的 管理是高级编程语言的重要功能之一。

Node.js框架提供了功能强大的进程管理模块(Process)。Process模块是Node.js框架 的一个全局内置对象，Node.js代码可以在任何位置访问该对象，实际上这个对象就是 Node.js代码宿主的操作系统进程对象。使用Process模块可以截获进程的异常、退出等事 件，可以获取进程的环境变量、当前目录、内存占用等信息，还可以进行工作目录切换、 进程退出等操作。

本章主要包括以下内容。

* Node.js框架进程管理机制。
* Node.js框架进程管理事件。
* Node.js框架进程管理应用。

1. 获取应用程序当前目录

查看应用程序的当前目录(完整路径)是一项非常实用的功能，Node.js框架为我们提 供了 process.cwd。函数来完成这项工作。

在本节这个简单的例程中，我们使用process.cwdO函数来获取应用程序当前的完整路 径。本例ch05.process-cwd.js主要代码如下：

01 *I\*\**

02 \* ch05.process-cmd.js

03 7

04 console.info(" Process cwd ");

05 console.info();

06 console.logfCurrent directory:' + process.cwdO);

07 console.info();

08 console.infof-— Process cwd ");

**【代码分析】**

第06行使用process.cwdO函数查看并打印输出了应用程序当前的完整路径，关于 process.cwdO®数的语法说明如下：

语法：process.cwdO；

图5.1是本例程在Ubuntu环境下输出的结果。图5.2是本例程在Windows环境下输出 的结果。



/usr/local/bin/node chO5.process-cwd.js

Process cvd

Current directory： Thome/king/webstorin.]s/pro]ects/NodeJS/chapterQ5

Process cwd

Process finished with exit code □

Current directory: h:\web5torm-workspace\NodeJS\chapter05

唾 "D:\Program Files\JetBrains\WebStorm 9.0.l\ban\runnerw.exe" » Process cwd

Process cwd

Process finished with exit code 0

图5.1获取应用程序当前目录(Ubuntu版)

图5.2获取应用程序当前目录(Windows版)

提示：对于process对象来讲，其本身是一个全局对象，设计人员可以在代码的任何地 方访问该对象；另外，使用process对象时不像使用其他对象时那样，无须先使用require 引用Process模块。

5.2改变应用程序目录

前一节介绍了查看应用程序当前目录的方法，本节我们再进一步介绍改变应用程序当 前目录的方法，Node.js框架为我们提供了 process.chdir()函数来完成这项工作。

在本节这个简单的例程中，我们结合使用process.cwd()和process.chdir()这两个函数来 查看并改变应用程序当前目录，同时将操作结果反馈给用户。本例ch05.process-chdir.js主 要代码如下：

01 /\*\*

02 \* ch05.process-chdir.js

03 7

04 console.info(n一-— Process chdir

05 console.info();

06 console.log('Current directory:' + process.cwdO);

07 console.log('Change directory to: /home/king');

08 process.chdirfVhome/king1);

09 console.log('Current directory:1 + process.cwdO);

1. console.info();
2. console.info(" Process chdir ™");

**【代码分析】**

第06行使用process.cwd()函数查看并打印输出了应用程序当前的目录(完整路径)； 第08行使用process.chdir()函数将应用程序当前的完整路径改变为新的路径(7home/king')； 第09行使用process.cwd()函数查看并打印输出了应用程序更改后的完整路径。

那么该例程执行后究竟有没有有效地改变了目录呢？图5.3演示的是本例程在Ubuntu 环境下的结果。图5.4是Windows环境下的结果。

**Ito <M&4»rota»d>cirJi \_ \_ \_** 冬

Run

卜 十 /usr/local/bin/node ch05 process chdir ]s Process chdir

■ 4-

"Current directory： /hotne/king/vebstorm js/projects/NodeJS/chapter05 由 曰 change directory to： /honeAing

r Current directory： /hoie/kmg

**X**

\* Precess chdir

? $ Process finished with exit code 0

图5.3改变应用程序目录(Ubuntu版)

提示：使用process.chdir（）函数改变目录时，

**5 cM&preea»fMM»uijt L**

► , "D:'Program Files\JetBraini\WebStorm 9.0.l\bin\runnerw.exe"

. . Process chdir

S % Current directory: h:\webstorm-workspace\NodelS\chapter05 匿 (3 Change directory to: h:\

@ Current directory: h:\

:IB

*f* Process chdir

Process finished with exit code 0

**I**

图5.4改变应用程序目录(Windows版)

其路径参数必须是完整路径，如将本例程

中的路径（7home/king，）改为非完整路径（例如:'/king' ） , Node.js框架会报错并提示 process.chdir（7king'）这个方法有问题

1. 获取系统相关信息

Process模块提供了一系列属性参数，用于返回非常有用的系统相关信息，设计人员可

以根据这些信息完成相关的实际工作。

在本节这几个例程中，我们使用process这个对象来查看这些系统信息。首先，我们看 一下如何获取进程pid与进程名称的方法，本例ch05.process-sysinfo-pid.js主要代码如下：

01 /\*\*

02 \* ch05.process-sysinfo-pid.js

03 \*/

04 console.info(M Process System Info ");

05 console.info();

06 console.infofS 前进程 id:');

07 console.info(process.pid);

08 console.info();

09 console.infof当前进程名称:')；

1. console.info(process.title);
2. console.info();
3. console.info(M-— Process System Info ”);

**【代码分析】**

第06行使用process.pid属性查看并打印输出了进程pid（如图5.5所示,pid为26168）； 第10行使用process.title属性查看并打印输出了进程名称（如图5.5所示，进程名称为/usr/ local/bin/node\*）。

图5.5是在Ubuntu环境下的结果，图5.6是在Windows环境下的结果。

图5.5

/usr/local/bin/node ch05.processsysinfo-pid ]S Process System Info

'D: \Program i les\ JetBrain$\WebStorm 9.0. l\bin\runnerw.exe' Process System Info

该带逐程ad：

26168

蚩前•适心祢：

/usr/local/bin/node

Process System Info

Process finished with exit code 0

查看进程pid和进程名称（Ubuntu版）

**3**野当前进程**id：**

虐廖7156

x |当前进程名称：

? D:\Program Files\3etBrains\WebStorm 9.G.1\bin\runnerw.exe

Process System Info

Protess finished with exit code 0

图5.6查看进程pid和进程名称（Windows版）

再次，我们看一下如何获取Node版本信息与属性的方法，本例ch05.process-sysinfo-

node.js主要代码如下:

01

\* ch05.process-sysinfo-node.js

02

03

\*/

04

05

06

07

08

09

console.infof Process System Info

console.info();

console.infofNode.js 版本号:')；

console.info(process.version); console.info();

console.infofNode.js 版本属性:')；

10

console.info(process.versions);

11

12

console.info();

console.info(" Process System Info ");

**【代码分析】**

第07行使用process.version属性查看Node的版本号（如图5.7所示'当前Node版本

号为v0.10.33）；第10行使用process.versions属性查看Node的版本属性（如图5.7所下）。

图5.7是在Ubuntu环境下的结果，图5.8是在Windows环境下的结果。

Run

/usr/local/bin/node ch05.process-sysinfo-node.js

Process System Info

"DzXProgram Files\JetBrains\WebStorm 9.0.l\bin\runnerw.exeM  Process System Info —

Node, js版本号：

甥弓 X a X曾 ? ®

Node, 本号： vO.lG.33

v0.10.35

Node, js版本尽性：

{ http parser： 'l.G',

node? 10.33',

v8: '3,14.5,9', ares： '1.9.0-DEV'. uv： '0.10 29', zlib： '1.23, modules： 'll', openssl： '1.0.lj' }

Node, j 5版本属性：

( http\_parser: '1.0', node: '0.10.35\*, v8: '3,14.5.9', ares: '1.9.0-DEV', uv: '0.10.30', zlib: 11.2.8', modules: 'll', openssl: \*1.0.lj' }

Process System Info

Process System Info

Process finished with exit code 0

Process finished with exit code 0

图5.7查看Node版本信息（Ubuntu版） 图5.8查看Node版本信息（Windows版）



Node, j 5配置选项：

{ target\_defaults: { cflags:[], default\_configuration; 'Release,, defines:[], include\_dirs:[], libraries: [] },

variables:

( clang: 0, host\_arch: ' ia32', node\_has\_winsdk: true, node\_install\_npm: true, node\_prefix: '\*, node\_shared\_cares: false, node\_shared\_http\_parser: false, node\_shared\_libuv: false, node\_shared\_openssl: false, node\_shared\_v8: false, node\_shared\_rlib: false, node\_\_tag: \* ', node\_\_unsafe\_optimizations: 0, node\_use\_dtrace: false, node\_use\_etw: true, node\_use\_\_openssl: true, node\_use\_perfctr: true, node\_\_use\_systemtap: false, openssl\_no\_asm: 0, python: \*C:\\Python27\\python.exe', target\_arch: \* ia321t v8\_enable\_gdbjit: 0, v8\_no\_strict\_aliasing: 1, v8\_use\_snapshot: true, visibility: want\_separate\_host\_toolset: 0 ) }

图5.9查看Node配置信息（Ubuntu版）

图5.10查看Node配置信息（Windows版）

102 -超实用的Nodejs代码段

然后，我们获取Node配置信息，本例ch05.process-sysinfo-nodeconfig.js代码如下:

01 /\*\*

02 \* ch05.process-sysinfo-nodeconfig.js

03 7

04 console.info("—-— Process System Info ");

05 console.info();

06 console.info('Node.js 配置选项:')；

07 console.info(process.config);

08 console.info();

09 console.info(n Process System Info ―");

【代码分析】

第07行使用process.config属性查看Node的配置信息（如图5.9所示）,从图中可以 看到，关于Node.js框架详细的系统配置信息，是以JSON数据格式进行了打印输出。

图5.9是在Ubuntu环境下的结果，图5.10是在Windows环境下的结果。



**. 亍** /usr/local/bin/node ch05.process-sysinfo-nodeconfxg.js Process Systea Info

■ 4

Node.

**囹 闩** { target\_defaults： E { cflags：[), *弁 '* default\_confiquration： 'Release'.

defines? 11. *X* include\_dirs：[).

> O libraries： [] ). variables：

{ clang： 0. gcc\_version： 48. host\_arch： 'ia32'. node\_install\_np•: true. node\_prefix-''. node\_shared\_cares: false. node\_shared\_hftpjiarser： false. node\_shared\_libuv: false, node\_shared\_openssl: false, node\_shared\_v8： false. node\_shared\_2lib： false. node\_tag： ,T.

node\_unsafe\_optimizations: 0, node\_use\_dtrace： false, node~use\_etv： false. node\_use\_openssl: true. node\_use\_perfctr： false, node\_use\_syste«tap: false. openssl\_no\_as■: 0, python： ,/usr/bin/python,. target arch： 'laS?'. v8\_enable\_gdb)it： 0. v8\_no\_stnct\_al.iasing： 1. v8\_use\_snapshot: true, want\_separate\_host\_toolset: 0 } }

Process System Info

Process finished with exit code 0

下面，我们看一下如何获取当前进程相关参数的方法，本例ch05.process-sysinfo- process.js 主要代码如下：

01 /\*\*

02 \* ch05.process-sysinfo-process.js

03 7

04

05

06

07

08

09

10

11

12

console.info(n—~ Process System Info ™H);

console.info();

console.info('运行当前进程可执行文件的绝对路径

console.info(process.execPath);

console.info();

console.infof当前进程的命令行参数数组:，)；

console.info(process.argv);

console.info();

console.info(\*'-

Process System Info

【代码分析】

第07行使用process.execPath属性查看当前进程可执行文件的绝对路径（如图5.11所 示，绝对路径为7usr/local/bin/node\*）；第10行使用process.argv属性查看当前进程的命令 行参数数组（如图5.11所示）。图5.12是本例程在Windows环境下的结果。

Run

Run chGS pra<ess-5-/stnfo.pfixes.54S 襟-L

A § /usr/local/bin/node chQ5.process-sysinfo-process.js

Process System Info

斐行罗辑进程可执行丈律的姑好瘙袅；

囹宕 /usr/local/bin/node 螺 世 声前恐租的会令杼名簌薮炽：

昏 I •/usr/local/bin/node'.

其— '/home/king/webstorin.js/pro]ects/NodeJS/chapterO5/chO5.process-sysinfo-process.is'】

Process System Info

Process finished with exit code 0

图5.11查看当前进程的相关参数（Ubuntu版）

► "D:\Program Files\3etBrains\WebStorm 9.0.1\bin\runnerw.exeM D:\nodejs\node.exe 箍 寻 Process System Info 旧；运行当前进程可执行文件的绝对路径：

3 D:\nodejs\node.exe

t ；当前进程的命§行参数数组： J；

,['D:\\nodejs\\node.exe , ,

'h:\\webstorm-workspace\\NodeJS\\chapter05\\ch05.process-sysinfo-process.js \麝

Process System Info

Process finished with exit code 0

图5.12查看当前进程的相关参数（Windows版）

下面，我们看一下如何获取系统和CPU架构相关参数的方法，本例ch05.process- sysinfb-cpu.js主要代码如下：

01 /\*\*

02 \* ch05. process-sysi nfo-cpu Js

03 \*/

04 console.info("—- Process System Info -—”);

05 console.info();

06 console.info('当前系统平台

07 console.info(process.platform);

08 console.info();

09 console.infof 当前 CPU 架构:，)；

1. console.info(process.arch);
2. console.info();
3. console.info(" Process System Info ");

**【代码分析】**

第07行使用process.platform属性查看当前系统信息（如图5.13所示，当前系统为 Linux）；第10行使用process.arch属性查看当前CPU架构信息（如图5.13所示，当前CPU 架构为ia32,说明本机为ia32指令集架构）。图5.14是本例程在Windows环境下的结果。

Run

竺**：chOS-procEss-syWo^yiMs** \* 七

A ▲ /usr/locaVbin/node chQ5,process-sysinfo-cpu. ]s Process System Info

**■ 4** 事牯厘统辛台：

囹linux

**X** *芯* 多君CPU架构：

x t32

? ® Process System Info

Process finished with exit code 0

"D:\Program Files\3etBrains\WebStorm 9.0.l\bin\runnerw.exe" Process System Info

当前系统平台:

Win32

当前**CP**嗥构：

ia32

—— Process System Info —

Process finished with exit code 0

图5.13系统与CPU架构信息（Ubuntu版） 图5.14系统与CPU架构信息（Windows版）

最后，我们看一下如何获取当前shell环境变量参数的方法，本例ch05.process- sysinfb-shell.js主要代码如下：

01 /\*\*

02 \* ch05.process-sysinfo-shell.js

03 \*/

04 console.infof1 Process System Info "）;

05 console.info（）;

06 console.infof指向当前shell的环境变量:'）；

07 console.info（process.env）;

08 console.info（）;

09 console.infof Process System Info "）;

第07行使用process.platform属性查看shell环境变量参数信息并进行打印输出（如图 5.15所示）。图5.16是本例程在Windows环境下的结果。

从图5.15和图5.16输出的信息中，我们可以找到很多非常有用的shell环境变量，例 如：PATH （系统环境路径）、JRE\_HOME （jre环境路径）、PWD （用户路径）、LANGUAGE （语言版本）、LANG （编码版本）、SHELL （shell环境路径）与JAVA HOME （jdk环 境路径）等。

说明：CPU架构ia32特指英特尔32位元架构（Intel Architecture, 32-bit,缩写为ia-32 ）, 即我们常说的i386或是x86架构，是由英特尔公司推出的目前最流行的指令集架构,

第5章 Node.js进程管理-105

Run

> a /usr/local/bm/node chOS process-sysinfo-shell. js

Process System Info

■ 4

— 桁姻莎ashell的环蝴Wk

国弓 { PATH： \* Zusr/l>ib/]ava/]dkl. 8.0^25/bm： /usr/lib/]ava/]dkl, 8,0\_25/] re/bin: /usr/lib/ ■ XAUTHORITY: \*/home/king/.Xauthonty1.

弁 了 XMODIFIERS： ・@m=fcitx,，

昏 XDG\_DATA\_\_DIRS: \* /us r/sha re/ubuntu ：/usr/sha re/gnome : /usr/local/share/ - /usr/share/ X — ®MSESSION \* ubuntu\*

7 ® MANDATORY\_PATH： 1/usr/share/qconf/ubuntu nandatory path\*,

TEXTDOHAINDIR： ' /usr/sha「e/locaR/‘,

GTK\_IM\_MODULE: efcitx\

DeuS^SESSION\_BUS^ADDRESS: 1 unix ；abstract^/tip/dbus- XORrvgvwic \* r

DffAULTS^PATH： '/usr/share/gconf/ubuntu.default.pathL

XDG\_CURR改T\_DESKTOP： 'UrntyL

JRE\_HOME： 17usr/Xib/iava/]dkl.8.Q\_25/jre \

LD\_LIBRARY\_PATH: '/hone/king/webstorm.js/WebStorm/bin：'.

UPSTART\_SESSION: \* unix:abstract«/com/ubuntu/upstart-session/l000/1925\*. QT4\_IM\_M00ULE: -fcitx\*.

SE^I函\_HANAGB^ localAing ThinkPad• X61: @/t叩/ ICE-urux/2052. unix/kinq-ThinkPa "

LOGNAME- -king\*.

jC© \*dbus\

PWO ,/hoae/king', IMJZONFIG PHASE： T. LANGUAGE： 'zh\_CN"h'. SHELL； •/bm/bash,.

GIO\_LAUNCF£D\_DESKTOP\_FILE: \*/howe/kmq/. local/share/applications/jetbrams-webst INSTANCE 1 \*7

GNOHE\_DESKTOP\_SESSIGN\_ID： 1this-is-deprecated\*.

GTK\_WDULES: ' overlay-scrollbar： unity - gtk-»odule1. CLUTT^IM\_MOOULE 、顼 L

XDG\_5EsGl丽\_PATH '/org/freedesktop/DisplayManager/SessionQ'. TEXTDOMAIN. 'im-config', COMPIZ^BIN\_PATH: ,/usr/bin/\

SESSIONTYPE: \* gnome-session \*,

SHLVL: 'O',

*COMPI*Z^COM=IG\_PROFILE: \*ubuntu\

(JT\_I侦ODULE \_\顼\

JAVA\_HOME： /usr/Ub/]ava/jdkl.8 0^25\*.

XFILESEARCHPATH: \* /usr/dt/app - def aults/%L/Dt ,.

XDG\_C0NFIG\_DIRS: '/etc/xdg/xdq-ubuntu：/usr/share/upstart/xdg：/etc/xdg,, LANG： \*zh CNUTF-e'.

图5.15 shell环境变量信息（Ubuntu版）

**chOS pcocMt-wfinfo-sJwtf**

**Run**

COMHJTERI4AME: 'KING-PC',

ComSpec: \*C:\\Windows\\sy$temB2\\cmdYxe',

CRX\_PATH\_2013\_x86: \*D:\\Program Files\\CAXA\\CAXA DRAFT MECHANICAL\\2013r2. FP\_NO\_HOST\_CHKK: \*NO\

HOMEDRIVE : 'C：',

HOMEPATH: ,\\Users\\KING,, 1LBDIR: \*C:\\Program Files\\Ccmwion Files\\Autodesk ILLDIR： 'C: \\Program FilesWCowmon Files\\Autcxiesk ILMDIR: \*C:\\Program Files\\Common Files\\Autodesk JAVA\_HOME: ,D:\\Program Files\\Java\\jdkl.8.0\_0S1, LOCALAPPDATA: aC：\\Users\\KING\\AppData\\Local', LOGONSERVER: 1\\\\KING-PC \ NUMBE R\_OF \_PROC ESSOR S: '4', OMP\_NUM\_THREADS: \*4\*, OS: \*Windows\_NT', Path: \*D:\\Python34\\;D:\\PythcxiB4\\Scripts;C:\\Program files\\HVIDIA Corpo PATHEXT: •.COMj.EXE;.BAT;.CMO;.VBS;.VBE;.JS;.3SE;.WSF;.WSH;.MSC;.PY\*, PROCESSOR^ARCHITECTURE: \*86',

Shared\\Haterials\\\*, Sbared\\Materials\\', Shared\\Materials\\\*,

PROCESSOR^IDENTIFIER:、86 Family *6* Model 58 Stepping 9, Genuinelntel,, PROCESSOR\_LEVEL: '6‘,

PROCESSOR\_REVISION: \* 3a09 \*,

ProgramData: \* C:\\ProgramData',

ProgramFiles: \*C:\\Program Files',

PSModulePath: 'C:\\Windows\\system32\\WindowsPowerShell\\vl,0\\Modules\\;C:

PUBLIC: 'CzWUsersWPublic \

SESSIONNAME: \* Console\*,

SystemDrive: 'C:',

Systemroot: 'C:\\Windows',

TEMP: •C:\\Users\\KING\\AppData\\Local\\Temp\*,

IMP： \*C:\\Users\\KING\\AppData\\Local\\Temp-,

TVT: \*C:\\Program Files\\Lenovo\*,

USERDOMAIN: \* KING-PC4,

图5.16 shell环境变量信息（Windows版）

5.4 实现标准输出流

在第1章我们给读者介绍了使用console.logO或console.info（）方法向控制台实现标准输

出流的方法。其实，如果感兴趣的读者查阅一下Node.js源码，就会发现console.log()与 console.info()方法实际上是通过封装Process模块的stdout.write。方法来实现的。

在本节这个例程中，我们通过对比使用console.log。、console.info()与process.stdout() 方法，向读者演示一下如何使用process.stdout.write()方法实现标准输出流。本例 ch05.process-stdout.js 主要代码如下：

01 /\*\*

02 \* ch05.process-stdout.js

03 7

04 console.info(" Process stdout ");

05 console.info();

06 console.log('Node.js Process Module - stdout method.');

07 process.stdout.write(,Node.js Process Module - stdout method/);

08 console.info^n');

09 console.infoCconsole.log()方法封装 了 process.stdout.write()方法')；

1. console.log = function(dX
2. process.stdout.write('process.stdout.write:' + d +
3. )
4. console.log('Node.js Process Module - stdout method.');
5. console.info();
6. console.info(" Process stdout ");

**【代码分析】**

第06行使用console.log()方法打印输出了一个字符串，内容为'Node.js Process Module - stdout method.1；第07行使用process.stdout.write()方法打印输出了与第06行同样的一个字 符串；从图5.17中的输出结果来看，第06行与第07行的打印输出结果是完全一致的。

第10〜12行参考了 Node.js源码中关于console.log()方法的实现过程，我们看到 console.log()方法是通过封装process.stdout.write()方法来实现的，区别是添加了 一段字符串 (^rocess.stdout.write: \*)作为标记；第13行使用新封装的console.log()方法打印输出了同 一个字符串，内容为\*Node.js Process Module - stdout method.'；从图5.17中的输出结果来看, 第13行打印输出的结果加上了人为添加的标记字符串(process.stdout.write: \*)。



►

. /usr/local/bin/node chG5.process-stdout.]S Process stdout

Node.js Process Module - stdout wethod. 耳 Node.]s Process Module - stdout aethod.

& console, log ()方注封笊了 process, stdout, write ()方法

后• process.stdout.write: Node.js Process Module - stdout nethod. 帅

Process stdout

Process finished with exit code 0

图5.17实现标准输出流的方法

提示：其实console.log()方法与process.stdout.write()方法还是有一点小小的区别的，如 果细心的读者去阅读源码，就会发现console.log()方法在封装process.stdout.write()方法时， 在结尾增加了一个换行符。所以，在使用console.log()方法打印输出一行后会自动换行，而 process.stdout.write()方法则不会自动换行

1. 实现标准错误流

任本节这个例程中，我们通过-个读取本地文本文件的例程，向读者演示一下如何使 用process.stderr.write()方法实现标准错误流。本例ch05.process-stderr.js主要代码如下：

01 /\*\*

02 \* ch05.process-stderr.js

03 7

04 console.info("——- Process stderr ");

05 console.info(); .

06 varfs = requireffs');

07 var file = \*err.txt';

08 var encoding = 'UTF-8\*;

09 fs.readFile(file, encoding, function(err, data) (

1. if(err) (
2. setTimeout(function() {
3. process.stderr.write('err: \* + err + \*\n\*); //打印输出标准错误流
4. }, 1000);
5. } else {
6. console.log(data);
7. ) .…
8. })；
9. console.info();
10. console. info(" Process stderr ”)；

**【代码分析】**

整个第06〜17行实现了一个读取本地文本文件的过程，为了测试本例程，代码实现了 一个打开读取文件的错误；第07行定义了一个不存在的文本文件(Hrr.txt，)，第09行使 用fs.readFile()方法读取该不存在的文本文件并抛出错误参数(err)；在第12行使用process, stderr, write。方法打印了该错误参数。

从图5.18中的输出结果来看，第12行打印输出的标准错误流信息(err: Error: ENOENT, open 'en.txt'),告诉设计人员这是一个打开读取文件的错误。

提示：本例程中使用setTimeout()方法主要是为了控制输出格式，如果直接使用 process.stderr.write()方法打印标准错误流，则错误信息会在正常信息输出之前就全部输出到 控制台中了

/usr/local/bin/node ch05,process-stderr.js

Process stderr

Process stderr err： Error： ENt^MT, open 'err.txt'

Process finished with exit code 0

图5.18实现标准错误流的方法

1. 实现标准输入流

在第1章我们给读者介绍了使用readline模块在控制台实现标准输入流的方法。现在, 读者再回去阅读一下相关代码，就会发现其中使用了 Process模块的process.stdout与 process.stdin这两个对象。

在本节这个例程中，我们通过使用process.stdin.read()方法实现一个基本的控制台回写 应用，向读者演示一下如何使用process.stdin.read()方法实现标准输入流。本例 ch05.process-stdin.js 主要代码如下：

01 /\*\*

02 \* chO5.prQcess-stdin.js

03 7

04 console.info(" Process stdin ");

05 console.info();

06 console.infof用户输入数据')；

07 /\*\*

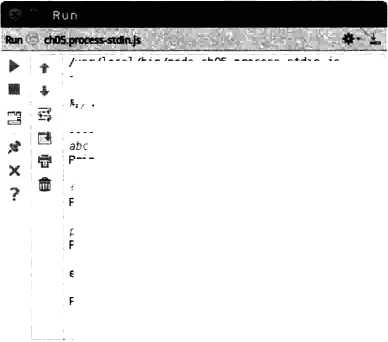
08 \* readable •接受控制台用户输入事件

09 \*/

1. process.stdin.setEncoding(,utf8,);
2. process.stdin.onfreadable1, function() (
3. var chunk = process.stdin.read(); //process.stdin.read()方法
4. if (chunk !== null) (
5. process.stdout. write(\*Print Data:1 + chunk + Vf); 〃打印输出
6. )
7. })；
8. /\*\*
9. \*end-结束控制台输入事件
10. 7
11. process.stdin.onCend1, function() (
12. process.stdout.writefend.\n,);
13. ))；
14. console.info();
15. console.info(" Process stdin ”)；

**【代码分析】**

第10行使用process.stdin.setEncoding('utfS')方法为输入流设定编码，此处设定为'utfB， 编码；在第11~16行实现了标准输入流的功能：第11行通过为process.stdin对象注册 readable事件，实现输入流回调函数；第12行通过process.stdin.read()方法读入用户在控制 台终端的输入信息，并返回保存在字符串变量chunk中；第14行将用户输入信息回写在控 制台终端中。

从图5.19中可以看到，用户输入的每一行数据信息(包含中英文混合字符)，均完成 了成功的回写功能；中英文混合字符的回写不出现乱码，需要感谢代码在第10行使用了 process.stdin.setEncoding()方法设定了 utf8 编码；在第 20~22 行通过为 process.stdin 对象注 册end事件，来实现控制台输入流结束的功能(用户可以输入Ctrl+D结束控制台输入流)， 控制台输入结束后，通过process.stdout.write()方法向控制台中输出提示信息(字符串，end，)； 从图5.19中可以看到，在用户键入Ctrl+D组合键后，控制台输入流结束，控制台中输出提 示信息“end”。

Process stdin

Print Data： abc

end.

Process finished with exit code 0

图5.19实现标准输入流的方法

孝好, *fiode js*

Print Data：作好• Node. js

*perfect.丈4 '*

Print Data： perfect.大事T !

/usr/local/bin/node chQ5.process-stdin.js Process stdin

周尹瓮入敷博

提示：当我们的输入、输出的内容中有中文的时候，可能会出现乱码，这是由于编码 不同造成的，所以在这种情况下需要使用process.stdin.setEncoding(编码)方法为输入、输出 流设置编码。

1. Kill当前进程

相信熟悉Unix/Linux系统的用户对kill命令一点也不会陌生，想要杀死进程基本就靠 这个命令了。Node.js框架的Process模块也提供了一个kill()方法，是不是也能完成杀死进 程的任务呢？

在本节这个例程中，我们通过使用Process模块提供的一些方法，实现一个杀死进程的 应用。本例ch05.process-kill.js主要代码如下：

01 /\*\*

02 \* ch05.process-kill.js

03 \*/

04 console.info(" Process kill pid ——\*')；

05 console.info();

06 console.info('当前进程 id:' + process.pid);

07 console.info();

08 process.on('SIGHUP,, function() (

09 console.log(\*Got SIGHUP signal.');

TO })；

1. setTimeout(function() (
2. console.log('Exiting.,.');
3. process.exit(O);
4. console.info('已经退出进程 id:' + process.pid);
5. }, 1000);
6. process.kill(process.pid, 'SIGHUP');
7. console.info('正在退出进程 id:'牛 process.pid);
8. console.info();
9. console.info(H Process kill pid

**【代码分析】**

第06行先使用process.pid属性打印当前进程pid；第08~10行使用process.on()方法 注册SIGHUP信号，用于终止控制台Node进程；第11-15行使用setTimeout()方法设定一 个1000毫秒的时间延迟;第12行打印了一个字符串Exiting...；第13行先使用process.exit(O) 方法终止当前进程；第14行打印了一个字符串，表示已经退出的进程pid；第16行先使用 process.kill()方法向当前进程发送了一个'SIGHUP'信号；第17行打印了一个字符串，表示 正在退出的进程pid。

图5.20是本例程在Ubuntu环境下输出的结果。

Zusr/local/bin/node chG5,process-kill.js

Process kill pid

多前送程id： 31913

正在也出适程^d： 31913

Process kill pid

Got SIGHUP signal. Exiting

Process finished with exit code 0

图5.20 Kill当前进程的方法

从图5.20中可以看到，第06行正常打印输出了提示信息，而在第16行的process.killO 方法执行后，第17行的提示信息也正常打印输出了，这就表明process.killO方法并没有杀 死(kill)当前进程。

同样在图5.20中，当第13行的process.exit(O)方法执行后，第14行的提示信息并没有 正常打印输出，这表明了反而是process.exit(O)方法杀死(kill) 了当前进程。

注意：process.killO这个方法的名称虽然使用杀死(kill)字符，但其与Unix/Linux系 统命令kill不一样，该方法仅仅只是一个信号发送器，且信号的发送不仅仅只是用来杀死 (kill )目标进程的。

1. Process模块的异步方法

在前面的很多例程中，我们都使用了 setTimeout。方法完成异步操作的执行。其实， Process模块同样提供了 一个process.nextTick()方法用于完成异步操作的执行。按照Node.js 框架的官方文档给出的解释*，*process.nextTick()方法比setTimeout。方法效率要高出很多， 实际情况是不是这样呢？

在本节这个例程中，我们通过对比使用setTimeout()方法与process.nextTick()方法，测 试一下这两个方法效率的差异有多大，具体体现在什么方面。本例ch05.process-nextTick.js 主要代码如下：

01 /\*\*

02 \* ch05.process-nextTick.js

03 7

04 console.infof Process nextTick ");

05 console.info();

06 /\*\*

07 \*使用setTimeout()方法执行异步操作

08 \*/

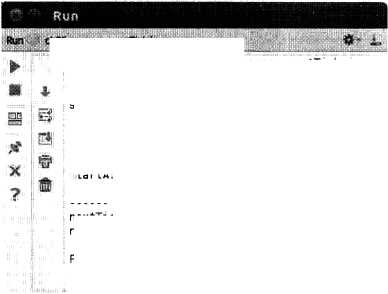
09 console.timeCstartB\*); //console.time()计时器 B

1. console.logfstart - setTimeout1);
2. /\*\*
3. \* setTimeout()方法
4. \*/
5. setTimeout(function() {
6. console.logfnextTick callback 2');
7. }, 0);
8. console.logfscheduled - setTimeout1);
9. console.timeEnd('startB,); //console.time()计时器 B
10. /\*\*
11. \*使用process.nextTick()方法执行异步操作
12. \*/
13. console.time('startA,); //console.time()计时器 A
14. console.logfstart ・ nextTick');
15. 〃\*
16. \* process.nextTick()方法
17. 7
18. process.nextTick(function() {
19. console.log('nextTick callback 1');
20. });
21. console.Iog('scheduled - nextTick\*);
22. console.timeEndCstartA'); Z/console.time()计时器 A
23. console.info();
24. console.info(n Process nextTick ");

【代码分析】

整个代码段分为两大部分：第06〜18行使用setTimeout()方法完成了一个异步操作功 能；第19~31行使用process.nextTick()方法完成了一个同样效果的异步操作功能。

图5.21是本例程在Ubuntu环境下的结果。从图中可以看到，执行setTimeout()方法异 步操作时，计时器标记(startB)显示为1毫秒；而执行process.nextTick()方法异步操作时， 计时器标记(startA)显示为0毫秒，表明process.nextTick()方法在执行过程中几乎没有被 阻塞。

同样在图5.21中，第15、28行在第33行的提示信息执行完毕后再打印输出，说明这 两行均是异步执行的；而第28行的提示信息是先于第15行的提示信息打印输出的，这更 加表明了 process.nextTick()方法在执行过程中几乎没有被阻塞；由此可见，Node.js框架官 方文档关于process.nextTick()方法的执行效率高于setTimeout。方法的说法还是有可信度的。

Process finished with exit code 0

Process nextTick nextTick callback 1 nextTick callback 2

图 5.21 process.nextTick()异步方法

膏；/usr/local/bin/node ch05.process-nextTick.]s

Process nextTick start - setTimeout scheduled - setTimeout startB: 1ms

start - nextTick scheduled - nextTick startA： 0ms

**提**ZF：

1. 我们知道Node.js是单线程执行的，除了系统I/O之外，在其事件轮询过程中，同 一时间只会处理一个事件。也就是说无论用户电脑有多少个CPU核心，也无法同时并行地 处理多个事件。
2. 然而，恰恰是Node.js的这种特性使得其更适合处理I/O型应用，却不适合诸如CPU 运算型的应用。在每个I/O型的应用中，设计人员只需要给每一个输入、输出定义一个回 调函数即可，系统会将其自动加入到事件轮询的处理队列里。当I/O操作完成后，这个回 调函数会被触发，然后系统会继续处理其他的请求。

第6章 child\_process及进程通信

众所周知，Node.js是基于单线程模型的架构(Google V8引擎)。在第5章介绍Process 模块的时候，读者也了解到Process模块是单线程的、无法完全利用多核CPU的先进性能 的。诚然，这样的设计虽然无法利用多个核心的CPU,但是却可以带来CPU高效的利用率， 这是其优势之处。然而，支持多进程任务还是我们大家所希望具备的功能的。为了解决这 个问题，Node.js框架提供了 child\_process模块，通过实现子进程来实现对多核心CPU的 有效利用。

Node.js框架设计的child\_process模块提供了 4个创建子进程的方法，分别是spawn。、 exec()、execFile。和fork。。其中spawn()是最原始的创建子进程的方法，其他三个都是通过 对spawn()方法不同程度地进一步封装实现的。使用child\_process模块提供的这些方法，可 以实现多进程任务、操作shell和进程通信等操作，实用功能是非常强大的。

本章主要包括以下内容。

. child\_process模块实现多进程任务。

* child\_process 模块操作 shell o
* child\_process模块实现进程通信。

1. 使用spawn方法创建子进程

Node.js框架child\_process模块为我们提供了四种方法来创建子进程，分别是spawn() 方法、exec()方法、execFile()方法以及fork。方法。这四种方法各有特点，从本节开始我们 将通过几个例程分别对这四种方法进行介绍。

首先，在本节这个简单的例程中，通过使用spawn。方法创建子进程来调用系统命令(Is) 查询目录清单的过程。本例ch06.child\_process\_spawn\_usage.js主要代码如下：

01 /\*\*

02 \* ch06.child\_process\_spawn\_usage.js

03 \*/

04 console.info(" child\_process spawn usage ——

05 console.info();

06 /\*\*

07 \* child\_process spawn

08 \* @type (exports.spawn|\*}

09 7

10

11

12

13

14

15

16

var spawn = require('child\_process').spawn; var ls\_var = spawn's',「-lh', Vvar\*]);

/\*•\*

•捕获控制台输出对象stdout,输出捕获数据 \*/

ls\_var.stdout.on('data', function (data) { console.Iog('stdout: \* + data);

〃引入 child\_process 模块

〃定义命令行'ls-lh/var'

〃打印输出/var目录清单

17 ));

**【代码分析】**

第10行引入child\_process模块，同时将spawn()方法赋于变量spawn,这样就可以通 过变量spawn使用spawn()方法了 ；第11行使用spawn()方法创建了一个子进程，用来调用 系统命令Is,同时定义了一个命令行Cis -lh /var\*)赋于变量ls\_var中，这个命令行执行的 作用就是查询，/vaF目录下的清单；第15〜17行通过绑定标准输出流的data事件，将命令行 (Is-lh/varO执行的结果打印输出。

图6.1是本例程在Ubuntu环境下输出的结果。



/usr/local/bin/node chG6.child\_process\_spawn\_usage js child\_process create

Stdout:忠巧士 44K drvxr-xr-x 2 root root

drvxr-xr-x 19 root root

drvxrvsrvt 2 root whoopsie 4

drvxr-xr-x 74 root root '

drvxrwsr-x 2 root staff Irvxrwxrvx 1 root root drvxrwxr-x 15 root syslog drvxrvsr-x 2 root mail drvx rvsrvt 2 root whoopsie 4 drwxr-xr-x 2 root root Irvxrwxrwx 1 root root drvxr-xr-x 1G root root drvxrvxrvt *2* root root

**0KOKOKGK0K90K0K0K0K4OKOK**

4.耳 3 Q8：27 backups

3 月 13 19:51 cache

4月 3 10：43 crash

3患 26 17:38 lib

4月 11 2014 local

3莒 13 15:10 lock -> /run/lock

4月 3 08：27 log

7月 23 2014 nail

7月 23 2014 aetrics

7患 23 2014 opt

1. 弓 13 15:10 run -> /run

3月 13 20:20 spool

1. 弓 3 07:54 tfflp

| king^>king-ThinkPad-X61: - 1 |
| --- |
| ktng^ktng-ThinkPad-X61:-$ Is -lh /var 总房置44K  drwxr-xr-x *2* root root 4.9K 4月 3 08:27 backups  drwxr-xr-x 19 root root 4.OK 3月 13 19:51 cm  drwxrwsrwt 2 root whoopsie 4.OK 4月 3 10:43 HUM  drwxr-xr-x 74 root root 4.OK 3月 26 17:38 lib  drwxrwsr-x 2 root staff 4 .OK 4月 11 2014 local  Irwxrwxrwx 1 root root 9 3月 13 15:10 -> HUH  drwxrwxr-x 15 root syslog 4.OK 4月 3 08:27 log  drwxrwsr -x 2 root nail 4.0K 7月 23 2614 Ell  drwxrwsrwt 2 root whoopsie 4.0K 7月 23 2614  drwxr-xr-x 2 root root 4.6K 7月 23 2014 opt  Irwxrwxrwx 1 root root 4 3月 13 15:10 •> /run  drwxr-xr-x 10 root root 4.ok 3月 13 28:20 spool  drwxrwxrwt 2 root root 4.®K 4月 3 07:54 M  ktnggktng-ThinkPad-X61:-$ | |

图6.2使用控制台命令行的执行效果

Process finished with exit code 0

图6.1使用spawn方法创建子进程

上图演示的是通过spawn()方法 创建子进程调用系统命令('Is -lh /vaf)的执行效果。既然是系统命令, 当然也可以在控制台中直接执行，那 么在控制台中执行的效果与通过 spawn()方法调用系统命令执行的效 果是不是一样的呢？图6.2演示的是 该命令行在Ubuntu环境下，使用控制 台执行后输出的结果。

从图6.1与图6.2的对比来看，输 出的结果是完全一致的，这也就表明了 child\_process模块的spawn。方法是值得信赖的。

提示：关于linux命令(例如本例中用到的Is命令)的使用方法，读者可以参考linux 官方网 站(http ://www. linux. org/ )或者专 门介绍linux 命令 的网站(例 如： <http://linux.about.com>) □

6.2使用exec方法创建子进程

上一节介绍了使用spawn()方法来创建子进程的过程，本节我们介绍使用exec()方法创 建子进程的过程。通过对比使用，我们可以找到这两种方法创建子进程的区别所在。

在本节这个简单的例程中，通过使用exec。方法创建子进程来调用系统命令(cat)查 看并打印输出指定文件内容的过程。本例ch06.child\_process\_spawn\_usage.js主要代码如下：

01 /\*\*

02 \* ch06.child\_process\_exec\_usage.js

03 7

04 console.info(M~ child\_process exec usage ");

05 console.info();

06 /\*\*

07 \* child\_process exec

08 \* @type (exports.exec|\*)

09 \*/

1. var exec = requirefchildjDrocess'J.exec; 〃弓| 入 child\_process 模块
2. var child = exec(\*cat ch06.child\_process\_exec\_usage.js',
3. function (error, stdout, stderr) (
4. console.infofcat ch06.child\_process\_exec\_usage.js stdout:');
5. console.log(stdout); 〃打印输出 stdout
6. })；

**【代码分析】**

第10行引入child\_process模块，同时将exec()方法赋于变量exec,这样就可以通过变 量exec使用exec()方法了 ；第11~15行使用exec()方法创建了一个子进程，用来调用系统 命令cat,同时使用cat命令查看一个js脚本文件(该脚本文件为ch06.child\_process\_spawn\_ usage.js,读者一定注意到了，查看的就是本例程的脚本文件)；第12〜15行即为callback 回调函数，在回调函数中将标准输出流(本例程实际为ch06.child\_process\_spawn\_usage.js 脚本文件的内容)进行了打印输出。

图6.3是本例程在Ubuntu环境下输出的结果。



A a. /usr/local/bin/node chO6.child\_process\_exec\_usage.js child\_process exec usage

**B 4**

cat chO6.child\_process\_exec\_usage.js stdout：

雪虾 /»• 一 —

& • Created by king on 15-4-3.

X 曾 • chO6.child\_process\_exec\_usage.js

? 览 console.mfo(" child\_process exec usage ")：

console.info()；

/•«

* child\_process exec
* etype (exports.exec I\*}

*\*/*

var exec = require('child\_process').exec； // TODO, *i>*Axhild\_process<^ var child = exec(1 cat ch(^,child\_process\_exec\_usage.js', function (error, stdout, stderr) {

console.info('cat chO6.child\_process\_exec\_usage js stdout ')； console. log (stdout): // TOGO： 印互出 stdout

});

Process finished with exit code 0

图6.3使用exec方法创建子进程

上图演示的是通过exec()方法创建子进程调用系统命令('cat')的执行效果。下面我们 在控制台中直接执行该命令打印输出ch06.child\_process\_spawn\_usage.js脚本文件的内容， 效果是不是一样的呢？图6.4演示的是该命令行在Ubuntu环境下，使用控制台执行后输出 的结果。

king(J)kIng-ThinkPad-X61: -/webstorm.js/projects/NodeJS/chapterOfi

ktngeking-ThinkPad-X61:-/webstorR.js/projects/NodeJS/chapter06$ cat che6.chtld\_p rocess\_exec\_usage.js

/\*.

* Created by king on 15-4-3.
* ch06.chtld\_process\_exec\_usage.js

\*/

console.info(" chtld\_process exec usage ");

console.info();

/\*\*

* chtld\_process exec
* @type {exports.exec I\*)

*7*

var exec = requtre('chtld\_process').exec; // TODO:引入chtld\_process模块

var child = exec('cat ch©6.chtld\_process\_exec\_usage.js\*,

function (error, stdout, stderr) (

console.tnfo('cat ch06.chtld\_process exec\_usage.js stdout:

console.log(stdout): // TODO:打印稔出 stdout

));ktng®ktng-ThinkPad-X61:-/webstorn.js/projects/NodeJS/chaptere6$ |

图6.4使用控制台命令行的执行效果

从图6.3与图6.4的对比来看，输出的结果是完全一致的，说明child\_process模块的 exec()方法是有效的。

关于spawn方法与exec方法的区别：

我们已经向读者介绍了 child\_process模块的spawn方法与exec方法的具体使用过程， 细心的读者会发现二者似乎功能相近，那么它们之间有没有区别呢？其实，还是有所区别 的〉

第一，spawn方法的参数必须要放到arg数组参数中，而不能放到command参数里面， 也就是说，这些参数都是不能带有空格的；而exec方法不存在这个问题，可以将参数直接 放在command参数里面。

第二，child\_process模块的spawn方法是"异步中的异步”，意思是指在子进程开始 执行时，它就开始从一个流总将数据从子进程返回给Nodeo具体实践中，当想要子进程返 回大量数据给Node.js框架时(比如说图像处理，读取二进制数据等)，最好使用spawn 方法

第三，child\_process模块的exec方法是“同步中的异步”,意思是尽管exec是异步的， 但是它一定要等到子进程运行结束以后再一次性返回所有的buffer数据。具体实践中，如 果exec的buffer体积设置得不够大，它将会以一个“maxBuflfer exceededn错误失败而告终°

1. 使用spawn方法绑定系统事件

我们在使用spawn。方法创建子进程后，就可以通过绑定系统事件的方式进行各种操 作，child\_process模块为开发人员设计了非常完善的操作方法，开发人员可以非常方便地 使用。

在本节这个简单的例程中,我们通过使用spawn。方法创建子进程并调用系统命令(cat) 来查看文件内容，然后通过绑定系统事件的方式来与控制台进行交互。本例 ch06.child\_process\_spawn\_std.js 主要代码如下：

01 /\*\*

02 \* ch06.child\_process\_spawn\_std.js

03 7

04 console.info(w child\_process spawn std ");

05 console.info();

06 var cp = require(\*child\_process'); 〃引 入 child\_process 模块

07 var cat = cp.spawn('caf); //定义命令行 cat

08 /\*\*

09 \*捕获控制台输出对象stdout的data事件

1. \*/
2. cat.stdout.onfdata', function(d) (
3. console.log(d.toString());
4. })；
5. /\*\*
6. \*绑定系统exit事件
7. 7
8. cat.on('exif, function() (
9. console.log('cat on exit!');
10. })；
11. /\*•
12. •绑定系统close事件
13. \*/
14. cat.onCclose1, function() (
15. console.log(\*cat on close!');
16. ))；
17. cat.stdin.write('cat on data!1);
18. cat.stdin.end();

〃通过控制台输入对象stdin写入数据

〃结束控制台输入对象stdin

**【代码分析】**

第06行引入child\_process模块并赋于变量cp,这样就可以通过变量cp使用child\_ process模块的方法了；第07行使用spawn()方法创建了一个子进程，并调用系统命令cat (cat命令用『查看文件内容)，同时赋于变量cat中；第11〜13行通过绑定标准输出流的 data事件，将用户在第26行通过控制台输入的数据(，cat on data!，)进行打印输出；第17〜 19行通过绑定系统exit事件，在子进程退出时捕获该事件并在控制台打印输出提示信息 ('cat on exit!'):第23~25行通过绑定系统close事件，在子进程的标准输入、输出流被 终止时捕获该事件并在控制台打印输出提示信息('cat on close!')。

图6.5是通过spawn()方法创建子进程调用系统命令(cat)并绑定系统事件的操作结果。 第26彳丁通过控制台输入的结果首先被打印输出；然后是系统exit事件被触发时打印输出的 提示信息；最后是系统close事件被触发时打印输出的提示信息。



/usr/local/bin/node ch06.child\_process\_spavn\_std.)s child\_process spawn std

cat on data! cat on exitl cat on close!

Process finished with exit code G

图6.5使用spawn方法绑定系统事件

提示：Node.js框架中的exit事件和close事件虽然均是表示退出、结束这样概念的事 件，但二者之间也有着明显的不同：对于exit事件，子进程的标准输入、输出(stdio)流 可能仍为开启状态；而close事件是在一个子进程的所有标准输入、输出(stdio )流被终止 时触发，因为多进程有时候会共享同一个标准输入、输出(stdio)流。因此，读者在使用 这两个事件时需要注意这点区别。

1. 使用exec方法绑定错误事件

在6.2节介绍了使用exec()方法来创建子进程的过程，本节我们介绍使用exec()方法绑 定错误事件的过程。

本节这个例程是对6.2节例程的引深，通过使用exec()方法创建子进程来调用系统命令 (cat)查看并打印输出指定文件内容的过程，但本例程在查看文件时发生了错误，因为读 取的是一个不存在的文件。本例ch06.child\_process\_exec\_std.js主要代码如下：

02

03

04

05

06

07

08

09

10

11

12

13

14

15

16

17

18

19

20

21

*a*

**au tS3**

* ch06.child\_process\_exec\_std.js

\*/

console.infoC\*— child\_process exec std

")；

console.info();

/\*\*

* child\_process exec
* @type {exports.exec|\*}

\*/

var exec = require('child\_process,).exec; 〃弓 I 入 child \_process 模块

var child = exec('cat ch06.child\_process\_exec\_usage',

function (error, stdout, stderr) (

console.info('cat ch06.child\_process\_exec\_usage.js stdout:'); console.log(stdout); 〃打印输出 stdout

console.info('cat ch06.child\_process\_exec\_usage.js stderr:'); console.log(stderr); 〃打印输出 stderr

if (error !== null) (

console.infofcat ch06.child\_process\_exec\_usage.js error:')； console.log(error); //打印输出错误信息

}

})；

**【代码分析】**

第10行引入child\_process模块，同时将exec()方法赋于变量exec,这样就可以通过变 量exec使用exec()方法了；第11〜21行使用exec()方法创建了一个子进程，用来调用系统 命令cat,同时使用cat命令查看一个js脚本文件(该脚本文件为ch06.child\_process\_ spawn\_usage,读者一定注意到了，虽然文件名称与本例程文件名相同，但没有文件后缀) 的内容；第12〜21行即为callback回调函数，在回调函数中我们分别对它的三个参数进行 了打印输出。

图6.6是本例程在Ubuntu环境下输出的结果。

Run

节 /usr/local/bin/node ch^.child\_process\_exec\_std.js

x child\_process exec std

4

\_ cat chO6.child\_process\_exec\_usaqe.js stdout：

矽 -

g cat chO6.child\_process\_exec\_usaqe.js stderr：

cat： ch06.child\_process\_exec\_usage:既有娜个史伴或&条

辱

cat ch06.child\_process\_exec\_usage.js error：

® 1 ( [Error： Command failed： cat： chCffi.chil.d\_process\_exec\_usage:没育邨\*文件我笛录 ]killed： false, code； 1, signal： null }

Process finished with exit code 0

图6.6使用exec方法绑定错误事件

下面我们对图6.6的输出结果进行分析：

第13~ 14行打印输出回调函数的第2个参数stdout,由于读取ch06.cbild\_process\_ spawn\_usage文件(无文件后缀名)时发生了错误，因此该参数没有输出任何有效数据；第 15〜16行打印输出回调函数的第3个参数stderr,由于读取ch06.child\_process\_spawn\_usage 文件(无文件后缀名)时发生了错误，因此该参数包含了错误提示信息(“没有那个文件 或目录”)；第17〜20行打印输出回调函数的第1个参数error,首先使用if条件判断语 句判断该参数是否为空(null),由于读取ch06.child\_process\_spawn\_usage文件(无文件后 缀名)时发生了错误，因此该参数不为空(包含了错误提示信息)。

关于exec()方法参数的说明。

1. 对于exec()方法回调函数的第1个参数error,当回调函数成功时，参数error取值为 空(null);而当回调函数发生错误时，参数error则为一个Error实例，并且err.code会是 子进程的退出代码，同时err.signal会被设置为结束进程的信号名°
2. 对于exec()方法的第2个可选参< options,该参数为用于指定一些选项的JSON数 据，其缺省选项为：

(

encoding: \*utf8\

timeout: 0,

maxBuffer: 200\*1024,

killSignal: 'SIGTERM',

cwd: null,

env: null

}

如果timeout大于0,则当进程运行超过timeout毫秒后会被终止；子进程使用killSignal 进行信号结束(缺省为'SIGTERM') ； maxBuffer指定了 stdout或stderr所允许的最大数据 量，如果超出这个值则子进程会被终止。

6.5创建子进程查看用户目录

本节我们通过一个例程向读者介绍基于child\_process模块创建子进程查看用户目录的 方法。这个例程的特别之处是给出了一个比较常规的查看目录的方式，不但涵盖了操作成 功的处理方式，也包含了操作失败的处理方式，这样可以有效地避免异常情况的发生。

本例程通过使用spawn。方法创建子进程来调用系统命令(Is)查看用户目录，并通过 绑定多个系统事件来监控操作状态。本例ch06.child\_process\_spawn\_ls.js主要代码如下：

01 /\*\*

02 \* ch06.child\_process\_spawn\_ls.js

03 \*/

04 console.info("—- child\_process spawn Is ");

05 console.info();

06 var spawn = require('child\_process').spawn; //引入 child\_process 模块

07 var ls\_var = spawn(1s', f-lh\*, Vusi3]); //定义命令行'ls・lh/usr'

08 /\*\*

09 \*捕获控制台输出对象stdout,输出捕获数据

1. \*/
2. ls\_var.stdout.on('data\*, function (data) (
3. console.logfstdout:' + data);
4. })；
5. /\*\*
6. \*绑定系统error事件
7. \*/
8. IS-Var.onCerror1, function (code) (
9. console.log('child process error with code ' + code);
10. ))；
11. /\*\*
12. \*绑定系统close事件
13. \*/
14. ls\_var.on('close', function (code) (
15. console.logfchild process closed with code' + code);
16. });
17. /\*\*
18. \*绑定系统exit事件
19. \*/
20. ls\_var.on('exit‘，function (code) {
21. console.logfchild process exited with code' + code);
22. })；

**【代码分析】**

第06行引入child\_process模块，同时将spawn()方法赋于变量spawn,这样就可以通 过变量spawn使用spawn。方法了 ；第07行使用spawn()方法创建一个子进程，用来调用系 统命令Is,并生成查询用户目录的命令行(Is； r-lh\ 7usf])赋于变量ls\_var；第11〜13行 通过变量ls\_var绑定标准输出流的data事件，将命令行。1礼]-lhJ/usF])执行的结果打印 输出；第17〜19行通过绑定系统error事件，在子进程发生错误时捕获该事件并在控制台

打印输出提示信息('child process error

with code ' + code),其中 code 为子进程

Run

发生错误时的错误码；第23〜25行通过

,

/usr/local/bin/node chQ6.child\_process\_spawn\_ls.js

child\_process spawn Is

绑定系统close事件，在子进程所有标准

输入、输出流被终止时触发；第29〜31

行通过绑定系统exit事件，在子进程结

束的时候被触发。

4 :

i stdout:总用量 148K

翌 i drvxr-xr-x 2 root root 68K 3月 12 12：G3 bin

怵:drvxr-xr-x 2 root root 4.OK 7J? 23 2014 games

阵,drwxr-xr-x 41 root root 2QK 3月 10 0Q：G3 include

曾 drwxr-xr-x 165 root root 20K 3月 12 12:G3 lib

drvxr-xr-x 13 root root 4.QK 】2月 9 10:15 local

drvxr-xr-x *2* root root 12K 3月 11 23:49 sbin

；drvxr-xr-x 334 root root 12K *3A* 10 00：G9 share

:drvxr-xr-x 1G root root 4.OK 2月 *28* 18:14 src

图6.7是本例程在Ubuntu环境下输

child process exited with code 0 child process closed with code G

出的结果。

；Process finished with exit code Q

下面我们对图6.7的输出结果进行

分析:

图6.7创建子进程查看用户目录

第11〜13行打印输出了回调函数的参数data,该参数中包含了用户目录(，/usr，)的相 关信息，由于该信息正确无误地被打印输出，由此可判定命令行执行完全成功；第23〜25 行与第29〜31行同样被成功执行了，正确打印输出了系统close事件与系统exit事件的退 出码(该退出码为0)；而第17〜19行却没有得到任何被执行的反馈信息，说明系统error 事件没有被触发。

说明：对于Node.js框架中的系统close与exit事件，其在正常退出时的退出码定义为 0,而在非正常退出时的退出码均不为0, —般定义为1~3的数字“

6.6查看用户目录出现错误的处理方法

本节我们对上一节的例程做一下扩展，向读者介绍基于child\_process模块创建子进程 查看用户目录时出现错误的处理方法。本例ch06.child\_process\_spawn\_lserr.js主要代码如 下：

01 /\*\*

02 \* ch06.child\_process\_\_spawnjserr.js

03 \*/

04 console.infof child\_process spawn Iserr M);

05 console.info();

06 var spawn = require(,child\_process,).spawn; //引入 child\_\_process 模块

07 var ls\_err = spawnfls', f-lh', 7errl); //定义命令行'ls・lh/usr'

08 /\*\*

09 •捕获控制台错误对象stderr,输出捕获数据

1. \*/
2. ls\_err.stderr.onCdata', function (data) (
3. console.logCstderr: \* + data);
4. })；
5. /\*\*
6. \*绑定系统error事件
7. \*/
8. Is^err-onCerror1, function (code) (
9. console.logfchild process error with code' + code);
10. ))；
11. /\*\*
12. \*绑定系统close事件
13. *\*/*
14. ls\_err.on('close', function (code) (
15. console.logfchild process closed with code1 + code);
16. })；
17. /\*\*
18. \*绑定系统exit事件
19. \*/
20. ls\_err.on('exit', function (code) (
21. console.log('child process exited with code' + code);
22. })；

**【代码分析】**

第07行使用spawn。方法创建了一个子进程，用来调用系统命令Is,并生成查询用户目 录的命令行Cis', ['-lh','/err'])赋于变量ls\_err；但这个用户目录('/e")其实是不存在的；第 17〜19行通过绑定系统error事件，在子进程发生错误时捕获该事件并在控制台打印输出提 示信息('child process error with code ' + code),其中code为子进程发生错误时的错误码。

图6.8是本例程在Ubuntu环境下输出的结果。



A 普 /usr/local/bin/node chG6.child\_process\_spawn\_lserr.]s

child\_process spawn Iserr

H 4 \_

child process exited with code 2

囹 耳 stderr： Is：无法访问/err：沃有耶个文件或,启泉

虚 隹 child process closed with code 2 x昏

k „ Process finished with exit code Q ? ®

图6.8查看用户目录出现错误的处理方法

下面我们对图6.8的输出结果进行分析：

第11〜13行打印输出了回调函数的参数data,但该参数没有打印输出任何信息，由此 可判定命令行执行失败；第23〜25行与第29-31行同样被成功执行了，正确打印输出了 系统close事件与系统exit事件的退出码(但该退出码为2):而第17〜19行打印输出了 反馈信息(“无法访问/err:没有那个文件或目录”)，说明系统error事件被成功触发。

说明：对于Node.js框架中系统error事件，其触发事件的对象均是事件方法 events.EventEmitter()的实例；在这一点上，我们可以将其理解为类似于JavaScript语言中的 addListener()方法、

6.7查看物理内存使用状态

在这一节，我们通过一个例程向读者介绍基于child\_process模块创建子进程查看物理 内存使用状态的方法。这个例程给出了一个比较常规的处理方式，既包含了操作成功的处 理方式，也包含了操作失败的处理方式，这样可以有效地避免异常情况的发生。

本例程通过使用spawn()方法创建子进程来调用系统命令(fVee)查看物理内存，并通 过绑定系统事件来监控操作状态。本例ch06.child\_process\_spawn\_free.js主要代码如下：

01 /\*\*

02 \* ch06.child\_process\_spawn\_free.js

03 7

04 console.info(n child\_process spawn free ");

05 console.info();

06 /\*\*

07 \* child\_process spawn

08 •利用子进程获取系统内存使用情况

09 \* @type {exports.spawn|\*)

1. \*/
2. var spawn = require('child\_process').spawn; 〃引入 child\_process 模块
3. var free = spawnCfree\*, 〃定义命令行 4 free -m,
4. /\*\*
5. \*捕获标准输出并将其打印到控制台
6. \*/
7. free.stdout.on(\*data', function (data) (
8. console.logf 标准输出:\n' + data);
9. })；
10. /\*\*
11. •捕获标准错误输出并将其打印到控制台
12. \*/
13. free.stderr.onfdata', function (data) (
14. console.logf 标准错误输出:\n' + data);
15. })；
16. /\*\*
17. \*注册子进程关闭事件
18. \*/
19. free.on('exit', function (code, signal) {
20. console.logf子进程已退出，代码：1 + code);
21. ))；

【代码说明】

第11行引入child\_process模块，同时将spawn。方法赋于变量spawn,这样就可以通 过变量spawn使用spawn（）方法了 ；第12行使用spawn。方法创建一个子进程，用来调用系 统命令ftee,并生成查看物理内存使用状态的命令行（Ee矿『m，］）,同时赋于变量free； 第16〜18行通过变量fYee绑定标准输出流的data事件，将命令行C-m'］）执行的结 果打印输出；第22~24行通过变量free绑定标准错误流的data事件，如果命令行（'free',



子是租我.3： 0

Process finished with e«xt code 0

free

124 1407 2532

used

2372 1090

0

cached

1102

shared buffers

142 IS?

**»06xNlq\_l>nxra>.sp«Mtfree.is**

Zusr/local/bin/node chO6.child\_process\_spavn\_free.)s child\_process spawn free --

total 2497 •/+ buffers/cache： Swap： 2532

执行时发生错误，则将错误信 息打印输出。

那么通过创建子进程执行命令 行See，, ［'-m'］）输出的结果如何呢？ 图6.9演示的是本例程在Ubuntu环 境下输出的结果。

下面我们对图6.9的输出结果进

第16〜18行被正确执行了，命令行打印输出的数据信息是物理内存的实时使用状态； 而第22〜24行没有被执行，说明执行命令行时未发生错误；第28〜30行同样被正确执行 了，打印输出了包含系统exit事件退出码的提示信息('子进程已退出，代码:0')。

图6.9演示的是通过spawn()方法创建子进程调用系统命令('free', ['-m'])的执行效果。 该命令行在控制台中执行的效果与通过spawn。方法调用系统命令执行的效果是不是一样 的呢？图6.10演示的是该命令行在Ubuntu环境下，使用控制台执行后输出的结果。

king@)king-ThlnkPad-X61: ~

ktng^ktng-ThinkPad-X61free -m total used

Mem： 2497 2373 -/+ buffers/cache: 1110 Swap: 2532 0 ktng^ktng-ThinkPad-X61|

free

123 1387 2532

shared buffers

151 181

cached

1082

图6.10使用控制台命令行的执行效果

从图6.9与图6.10的对比来看，两者输出结果中的主要数据是完全一致的，其中部分 数据产生变化是因为内存状态是实时变化的，这一点是正常的。

关于物理内存使用状态的简单说明

本例程查看的是Ubuntu系统的物理内存使用状态，其与Windows系统还是有一定区 别的，我们根据本例程的效果图做如下几点说明。

1. 我们看到的buffers关键字，其含义表示的是缓冲区；而cache关键字，其含义表示 的是高速缓存；二者的区别如果详细解释起来还是非常复杂的，如果用非常简单的理解方 式，那就是cache更趋向于是一个硬件概念，而buffers更趋向于一个软件概念；感兴趣的 读者可去查阅相关书籍，
2. 我们看到的Swap关键字，其含义表示的是交换分区，如果读者知道Windows系统 的虚拟内存的概念，那就太好了，二者原理功能类似。

6.8查看子进程pid的方法

在这一节，我们通过一个简单的例程向读者介绍基于child\_process模块查看子进程pid 的方法。child\_process模块为开发人员提供了一个child.pid属性，通过该属性可以得到子 进程的pido

本例 ch06.child\_process\_spawn\_pid.js 主要代码如下：

01 /\*\*

02 \* ch06.child\_process\_spawn\_pid.js

03 \*/

04 console.info("—— child\_process spawn pid -•—");

05 console.info();

06 〃•

07 \* child\_process spawn

08 \* @type {exports.spawn|\*}

09 7

1. var spawn = require('child\_process').spawn; //引入 child\_process 模块
2. /\*\*
3. \* grep child pid of node
4. \*/
5. var grep\_node = spawn('grep', rnode\*]);
6. console.log(,Spawned child pid of node:' + grep\_node.pid);
7. grep\_node.stdin.end();
8. console.info();
9. /\*•
10. \* grep child pid of top
11. \*/
12. var grep\_top = spawnCgrep',「top1]);
13. console.log('Spawned child pid of top:' + grep\_top.pid);
14. grep\_top.stdin.end();
15. console.info();
16. /\*\*
17. \* grep child pid of ssh
18. 7
19. var grep\_\_ssh = spawn('grep',「ssh]);
20. console.log(\*Spawned child pid of ssh:' + grep\_ssh.pid);
21. grep\_ssh.stdin.end();
22. console.info();

**【代码分析】**

第10行引入child\_process模块，同时将spawn()方法赋于变量spawn,这样就可以通 过变量spawn使用spawn。方法了 ；第14行使用spawn。方法创建-■个子进程，用来调用系

统命令grep,并生成检索node进程的命令行('grep', ['node']),同时赋于变量grep node； 第15行通过变量grep node打印输出该子进程的pid；第16行通过标准输入流的end。方法,

|  | Run 1 |
| --- | --- |
| **Run** | **diO6.cMltLixoas\_spBwruMMs A-** |
| **A** | . /usr/local/bin/node chO6.child\_procss spawn\_pid.]s |
|  | child\_process spawn pid |
| ■ | **4** |
|  | Spawned child pid of node： 32290 |
|  | **§** |
|  | V Spawned child pid of top. 32291 |
|  | 目 Spawned child pid of ssh： 32292 |
| ? | 血[  Process finished with exit code 0 |

图6.11查看子进程pid的方法

终止这个刚刚创建的子进程；第21-23行 和第28〜30行这两段代码与第14〜16行类 似，不同的地方是这两段代码分别生成的是 检索了 top进程和ssh进程的命令行，再分 别打印输出其child.pid属性值。

图6.11是本例程在Ubuntu环境下输出 的结果。从结果可以看到，因为在代码中r 进程是被连续创建的，因此三个新创建的子 进程的pid值是连续的。

说明：在Linux系统中,关键字pid的名称是进程控制符，英文全拼为Process Identifier, 其含义相当于进程身份标识；pid就像个人身份证一样，在操作系统中是唯一的、独一无二 的标识°

1. 创建子进程统计系统登录次数

在这一节，我们通过一个例程向读者介绍基于child\_process模块创建子进程统计系统 登录次数的方法。该例程通过使用exec()方法创建子进程来调用系统命令(last)统计系统 登录次数，并通过绑定系统事件来监控操作状态。

本例 ch06.child\_process\_exec\_last.js 主要代码如下：

01 /\*\*

02 \* ch06.child\_process\_execjast.js

03 7

04 console.info(" child\_process exec last -—

05 console.info();

06

07 \*创建子进程统计系统登录次数

08 \* @type (exports.exec|\*}

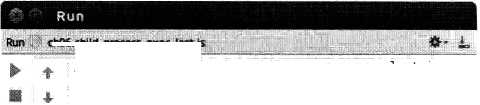
09 7

1. var exec = requirefchild\_process'J.exec;
2. var last = exec('last | wc -I');
3. /\*\*
4. \*捕获控制台输出对象stdout的data事件
5. 7
6. last.stdout.onfdata', function (data) (
7. console.logf系统登录次数统计:'+ data);
8. ))；
9. /\*\*
10. \*绑定系统exit事件
11. \*/
12. last.on(\*exit', function (code) (
13. cxjnsole.logf 子进程已关闭，代码:\* + code);
14. ))；

**【代码分析】**

第10行引入child\_process模块，同时将exec()方法赋于变量exec,这样就可以通过变 量exec使用exec()方法了 ；第12行使用exec()方法创建一个子进程，用来调用系统命令last, 并生成统计系统登录次数的命令行(\*last | wc -D ,同时赋于变量last；第15-17行通过变 量last绑定标准输出流的data事件，将命令行Clast | wc -D执行的结果打印输出。

那么通过创建子进程执行命令行Clast | wc -D输出的结果如何呢？图6.12演示的是 本例程在Ubuntu环境下，使用WebStorm开发工具输出的结果。



7usr/locat/bin/node ch06.child\_process\_exec\_last.js

child\_p rocess exec last

系绒■警录次薮纱存：31

子透程巳美困，宿码：0

Process finished with exit code 0

图6.12创建子进程统计系统登录次数

从图6.12输出的结果来看，系统登录次数为31次。如果该命令行在控制台中执行， 其输出结果与通过exec。方法调用系统命令执行的结果是不是一样的呢？图6.13演示的是 该命令行在Ubuntu环境下，使用控制台执行后输出的结果。



king@king-ThinkPad-X61:~$ last | wc -I 31

king^king-ThinkPad-X61:|

图6.13使用控制台命令行的执行效果

从图6.12与图6.13的对比来看，系统登录次数是完全一致的(均为31次)，说明这 种创建子进程统计系统登录次数的方法是有效的。

说明：根据Linux系统相关官方网站的解释，last命令用于显示近期用户或者终端的登 录情况，其作用权限是所有用户。

6.10使用execFile方法创建子进程

在前面，介绍了使用spawn。方法和exec()方法创建子进程的方法，在本节中我们介绍 使用execFile()方法创建子进程的方法。在本节这个简单的例程中，通过使用execFile()方法 创建子进程来调用程序文件(/bin/ls),来查看当前目录的内容。

本例 ch06.child\_process\_execFile\_usage.js 主要代码如下：

01 /\*\*

02 \* ch06.child\_process\_execFile\_usage.js

03 \*/

04 console.infof'—— child\_process execFile usage ");

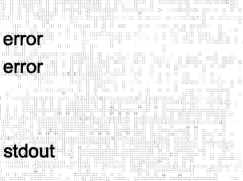
05 console.infoO；

06 /\*\*

07 child\_process execFile

08 \*查看当前目录内容

09 \* @type (exports)

1. 7
2. var child\_process = require('child\_\_process'); 〃引入 childjDrocess 模块
3. child\_process.execFile(7bin/ls', ['-Ih', function (error,stdout,stderr) (
4. if (error !== null) (

〃打印输出

〃打印输出

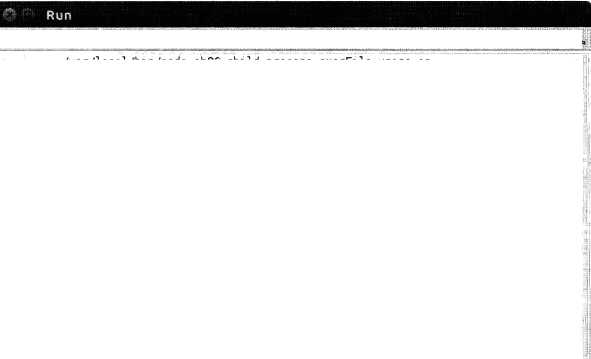
〃打印输出

1. console.logCexecFile error:1 + error);
2. console.log('execFile stderr:' + stderr);
3. } else (
4. console.infof 当前目录内容:');
5. console.info(stdout);
6. )
7. ))；

【代码分析】

第11行引入child\_process模块，同时将模块对象赋于变量child\_process；第12〜20 彳亍使用变量child\_process调用execFile。方法创建一个子进程，用来调用/bin目录下的程序 文件Is,同时生成了一个命令(Finns', f-lh1, 7])来查询当前目录的内容，这个命令的目标 路径(？)表示当前目录。

那么该例程执行后效果如何呢？图6.14演示的是本例程在Ubuntu环境下，使用 WebStorm开发工具输出的结果。从显示结果可以看到，打印输出的为当前目录全部js代 码文件，本例程 ch06.child\_process\_execFile\_usage.js 文件也在其中。



3月 3月 4月 4月

4月

4月

4月

4月

4月

4月 4E

4另

4月

4月

Process finished with exit code 0

冰2K23992922116287819781917329 *hr* - 7 585788877751

/usr/local/bin/node chO6.child\_prQcess\_execFile\_usage.]s childprocess execFile usage

CD

1 king king 1 king king 1 king king 1 king king 1 king king 1 king king 1 king king 1 king king 1 king king 1 king king 1 king king 1 king king 1 king king 1 king king 1 king king

3Q 21:56 chG6.chil.d\_process

30 21:49 chQ6.child\_process 1G 14；46 ch06.child\_process

10 11:56 ch06,child\_process 7 14:41 chG6.child\_p rocess

3 11:57 chG6.child\_process

9 16；57 chG6.child\_process IO G8：18 chG6.childj)rocess

9 09:59 chG6.child\_p rocess

9 09:59 chG6.child\_process IQ 11:31 ch06,child\_process

9 17:06 chQ6.child\_process

7 G9:17 chQ6,child\_process

3 11;46 ch06.child\_process

7 14:55 worker.is

COffiffl.tXt execFile\_usage.js exec\_last,]s exec\_std.js exec\_usage.js fork\_usage.js spawn\_free. js spavn\_lserr. js spawn\_ls.js spawn\_pid.js spavn\_ps.js spawn\_std.js spawn usage.js

图6.14使用execFile方法创建子进程

说明：child\_process模块的exec()方法与execFile()方法均支持回调函数，二者的区别 就是execFile()方法不用启动独立的shell,因此相对来说，它更加轻量级。

1. 使用execFile方法绑定错误事件

在6.10节介绍了使用execFileO方法来创建子进程的过程，这一节我们介绍使用execFile() 方法绑定错误事件的过程。本节这个例程与6.10节的例程基本一样，唯一的区别就是目标 路径是一个不存在的错误路径，因此在使用execFileO方法读取该错误路径时会触发系统错 误事件。

本例 ch06.child\_process\_execFile err.js 主要代码如下：

01 /\*\*

02 \* ch06.child\_process\_execFile\_err.js

03 \*/

04 console.info(M child\_process execFile error ”)；

05 console.info();

06 /\*\*

07 \* child\_process execFile

08 •查看当前目录内容

09 \* @type (exports}

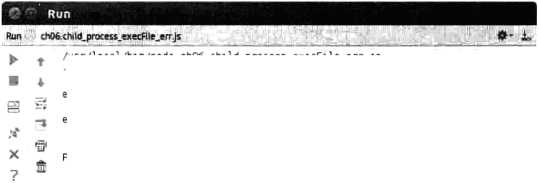
1. \*/
2. var child\_process = require('child\_process'); 〃引 入 child\_process 模块
3. child\_process.execFile(7bin/ls', f-lh', Verf], function (error,stdout,stderr) (

|  |  |  |  |
| --- | --- | --- | --- |
| 13 | if (error !== null) ( |  |  |
| 14 | console.logfexecFile error:' + error); | 〃打印输出 | error |
| 15 | console.logfexecFile stderr:' + stderr); | 〃打印输出 | stderr |
| 16 | } else ( |  |  |
| 17 | console.infof当前目录内容:')； |  |  |
| 18 | console.info(stdout); | 〃打印输出 | stdout |
| 19 | ) |  |  |

20 ));

**【代码分析】**

第11行引入child\_process模块,同时将模块对象赋于变量child\_process；第12〜20 行使用变量child\_process调用execFileO方法创建一个子进程，用来调用/bin目录下的程序 文件Is,同时生成了一个命令(7bin/ls\ f-lh1, 7errf])来查询目录的内容，我们注意到这个

目标路径(7eL)是一个不存在 的错误路径；第13〜15行使用 if条件判断语句对'error，参数进 行了判断，如果条件为真(true), 则分别执行第14行与第15行代 码，打印输出错误提示信息。

execFile error： Error： Conmand failed： /bin/ls：王法'VM/err：会膏令文等衣&求

execFile stderr： /bin/ls：壬注甘问/err；殳布旺＜-史伴溢.0垂

Process finished with exit code 0

/usr/local/bin/node chO6.child\_process\_execFileerr.]s childprocess execFile error

图6.15使用execFile方法绑定错误事件

图6.15演示的是本例程在 Ubuntu环境下，使用WebStorm开发工具输出的结果。

从图6.15的显示结果可以看到"error，参数打印输出的内容为一个信息数组，说明代码 中定义的目标文件或目录不存在；而'stderr'参数打印输出的内容略有不同，但也同样说明目 标文件或目录不存在。

1. 使用fork方法获取CPU信息

在前面几节中，分别向读者介绍了使用spawn。方法、exec()方法和execFile()方法创建 子进程的方法，在本节中我们介绍最后一种fork()方法。fork。方法也是基于spawn。方法实 现的，不过该方法是一种特殊实现，下面就介绍一下其究竟特殊在哪些地方。

同样，本节通过一个例程使用fork()方法创建子进程来调用系统OS模块，通过OS模 块实现获取CPU信息的功能。本例ch06.child\_process\_fork\_usage.js主要代码如下：

/\*\* :….wu•盘、嘟空:第材弋，

01

02

03

04

05

06

07

08

09

10

11

12

13

14

15

16

17

18

19

20

21

22

* ch06.child\_process\_fork\_usage.js

\*/

console.info(H child\_process fork usage

console.info();

/\*\*

* child\_process fork
* @type (exports.fork}

\*/

var fork = require(,child\_process,).fork;

/\*\*

\*获取当前机器的CPU内核数量

\*/

var cpus = require('os').cpus();

console.info('当前机器 CPU 内核数量：1 + cpus.length); for (var i = 0; i < cpus.length; i++) (

\*通过fork。方法创建新的子进程

\*/

console.log('Fork a new child\_process now...'); fork('./worker.js\*);

【代码说明】

〃引入 child\_process 模块

〃获取CPU内核数量

〃生成新子进程

第10行引入child\_process模块，同时将fork()方法赋于变量fork,这样就可以通过变 量fork使用fork()方法了 ；第14行引入OS模块，同时将cpus()方法赋于变量cpus,然后 第15行通过cpus.length属性打印输出当期机器CPU的内核数量；第16~22行通过fbr循 环语句依据上面得到的CPU内核数量，使用fork。方法创建子进程,我们注意到第21行fork()方法的使用方式，其直接调用了 js脚本文件或称模块文件（L/worker.js，）。 本例worker.js主要代码如下：

01 /\*\*

02 \* ch06.child\_process\_fork\_usage.js worker.js

03 7 :

04 console.infofThis is a child\_process.'）;

图6.16演示的是本例程在Ubuntu环境下，使用WebStorm开发工具输出的结果。从显 示结果可以看到，当前笔者机器的CPU内核数量为2,是属于一款比较古老的机器了；根 据内核数量2,使用fork。方法创建了 2个子进程，新生成了子进程调用的模块文件 ('./worker.js')*，*也相应打印输出了提示信息('This is a child\_process.')。

Run

**Run** 匚 **diOBiMjrocEJfcriUBgft**

\* /usr/local/bin/node chG6.child\_process\_fork\_usage js

T child\_process fork usage

**4**

f 2

3

\* x ?

£? Fork a new child\_process now... Fork a new child\_process now...

Y This is a child\_process.

昏 This is a child\_process.

应 Process finished with exit code 0

图6.16使用fork方法获取CPU信息

关于fork。方法的说明。

1. 该方法除了具有普通child\_process实例所具有的所有方法外，其所返回的对象还具 有内置的通讯通道。
2. 该方法在缺省情况下，所派生的Node进程的stdout、stderr会关联到父进程，如要 更改该行为，可将options对象中的silent属性设置为true。
3. 该方法创建的子进程运行完成后并不会自动退出，用户需要明确地调用process.exit() 进行退出。
4. 这些派生的Node进程是全新的Google V8实例，假设每个新的Node进程大致需要 至少30毫秒的启动时间和10MB内存，也就是说用户不能创建成百上千个这样的实例
5. Node.js框架虽然自身存在多个线程，但是运行在Google V8 ±的JavaScript是单线 程的；Node.js框架的child\_process模块用于创建子进程，设计人员就可以通过子进程充分 利用CPU的性能了。

6.13使用fork方法实现进程间通信

child\_process模块还提供了一个强大的功能就是进程间通信，有了这项功能就可以实 现高效率的多进程应用，充分利用多核CPU的性能优势。本节我们通过参考一个官方文档 中的简单例程，介绍使用fbrk()方法和child.send()方法实现进程间通信的过程。

本例主进程ch06.child\_process\_fbrk\_main.js主要代码如下：

01 /\*\*

02 \* ch06.child\_process\_fork\_main.js

03 7

04 console.infof childjorocess fork main ");

05 console.info();

06 /\*•

07 \* child\_process fork

08 \* @type (exports}

09 \*/

1. var cp = requirefchild\_process'); //引入 child\_process 模块
2. var n = cp.fork('ch06.child\_process\_fork\_sub.js\*); //fork 子进程
3. /\*\*
4. \* fork message event
5. \*/
6. n.onCmessage1, function(m) (
7. console.log('PARENT got message:', m);
8. })；
9. /\*\*
10. \* child\_process send message
11. \*/
12. n.send(( main: 'sub')); //send message
13. console.info();

**【代码分析】**

第10行引入child\_process模块，同时将模块对象赋于变量cp；第11行使用变量cp 调用fork()方法创建一个子进程，并调用了 js脚本文件或称模块文件 (,ch06.child\_process\_fork\_sub.js,),同时赋于变量 n；第 15〜17 行通过绑定系统 message 事件，在该进程接收到子进程模块(,ch06.child\_process\_fork\_sub.js,)发来的信息时被触发， 并通过第16行打印输出该信息({ sub:，main，})；第21行通过child.send()方法发送信息 给子进程模块，该子进程模块就是前面fbrk()方法调用的文件(,ch06.child\_process\_fork\_ sub.js') o

本例模块文件ch06.child\_process\_fork\_sub.js主要代码如下：

01 /\*\*

02 \* ch06.child\_process\_\_fork\_sub.js

03 7

04 console.info(" childjjrocess fork sub ");

05 console.info();

06 /\*\*

07 \* process on message

08 7

process.on('message', function(m) { console.logfCHILD got message:', m);

09

10

11

12

13

14

15

16

});

/\*\*

\* process send message

7

process.send(( sub: 'main'}); 〃发送信息

console.info(); *a*

【代码分析】

第09〜11行通过绑定系统message事件，接收主进程模块文件(,ch06.child\_process\_ fork main.js1)发来的信息，并通过第10行打印输出该信息({ main: \*subr })；第15行通 过process,send()方法发送信息给主进程模块，该主进程模块就是前面的文件

('ch06.child\_process\_fork main.js\*)*。*

图6.17是本例程在Ubuntu环境下输出的结果。从显示结果可以看到，由于主进程 (ch06.child\_process\_fork\_main.js)的第 11 行调用了模块文件(ch06.child\_process\_fbrk\_ sub.js),因此该模块文件被调用后向主进程发送了信息((sub: -main\*})*，*主进程第15〜 17行代码捕获了该信息并打印输出；而后主进程第21行向子进程发送了信息({ main: 'sub，}),子进程第09〜11行代码捕获了该信息并打印输出。

.除■七；

/usr/local/bin/node chO6.child\_process\_fork\_main.]s

child\_process fork main

耳

B 昏 蠢■

child\_process fork sub

PARENT got message： { sub： 'main' }

CHILD got message： { main： 'sub' }

图6.17使用fork方法实现进程间通信

说明：child.send()方法是同步的，因此本例程先打印输出了子进程发向主进程的信息, 而后才打印输出了主进程发向子进程的信息。

1. 使用fork方法实现斐波那契数列

经过上一节的介绍，读者应该对child\_process模块的进程间通信功能有了 一定的了解。 本节我们就引深一步，使用fork方法实现进程通信版的斐波那契数列，充分利用多核CPU 的性能优势。

本例主进程ch06.child\_process fib main.js主要代码如下：

01 *I\*\**

02 \* ch06.child\_process\_fib\_main.js

03 \*/

04 console.info(" child\_process fib main ");

05 console.info();

06尸

07 \* child\_process fork

08 \* @type {exports}

09 \*/

1. var cp = require('child\_process'); //引入 child\_process 模块
2. varchild = cp.forkfch06.child\_process\_fib\_sub.js,); 〃每个请求都单独生成一个新的子进程
3. /\*\*
4. \* fork message event
5. \*/
6. child.on('message', function(m) {
7. console.info('斐波那契数列：'+ m.result + \*\n');
8. })；
9. /\*\*
10. \* child\_process send message
11. 7
12. var input = parselnt('1 O');
13. child.send({input: 1)); //send message
14. for(var i = 1; i<input; i++)(
15. child.send((input: i}); //send message
16. }

**【代码分析】**

第11行使用变量cp调用fork。方法创建了一个子进程，并调用了 js脚本文件或称模 块文件('ch06.child\_process\_fib\_sub.js，),同时赋于变量child；第15~17行通过绑定系统 message事件，在该进程接收到子进程模块('ch06.child\_process\_fib\_sub.js')发来的信息时 被触发，并通过第16行打印输出斐波那契数列；第21行定义了斐波那契数列的项数(本 例为10)；第22行先行打印了斐波那契数列的第1项，这是由斐波那契数列的特性所决 定的；第23〜25行使用for循环语句并通过child.send()方法发送信息给子进程模块，该子 进程模块就是前面fork。方法调用的文件(lh06.child\_process\_fib\_sub.js，)。

本例模块文件ch06.child\_process\_fib\_sub.js主要代码如下：

01 /\*\*

02 \* ch06.childjorocess\_fib\_sub.js

03 7

04 console.info(" child\_process fork sub ”)；

05 console.info();

06 /\*\*

07 \* function fib

08 7

09 function fib(n) (

1. if (n < 2) (
2. return 1;
3. ) else (
4. return fib(n -2) + fib(n -1);
5. }
6. }
7. /\*\* f
8. \* process send message
9. \*/ !
10. process.on('message', function(m) {
11. process.send({result: fib(m.input)}); //send message
12. })；

**【代码分析】**

第09〜15行定义了一个递归计算斐波那契数列的函数(fib(n)),关于斐波那契数列 在第1章有过详细的介绍，读者可去参阅；第19〜21行通过绑定系统message事件，接收 主进程模块文件('ch06.child\_process\_fib\_main.js')发来的信息；第 20 行通过 process.send() 方法发送信息({result: fib(m.input)))给主进程模块，其中该信息中包含对递归函数fib(n) 的调用，且主进程模块就是前面的文件('ch06.child\_process\_fib\_main.js') o

图6.18是本例程在Ubuntu环境下，使用WebStorm开发工具输出的结果。从显示结果 可以看到，本例程打印的10项斐波那契数列被正确无误地输出了，由此可见child\_process 模块进程通信的强大功能。

Run

**ch06 cNW\_process\_fib\_malaj$**

.| a /usr/local/bin/node chO6.child\_process\_fib\_Bain.js

i T child\_process fib aain

■，

| child\_process fork sub

囹；目

X I 叟波邪奥敷列：1

? © 笑波雾捉H列：2

斐波郎美敷列；3

叟成畀挺饭珂：5

笑痘邪美装列：8

关波邪美敷列：13

要波邨矣金河：21

雯波那莫列：34

55

Process finished with exit code 13G

图6.18使用fork方法实现斐波那契数列

关于child\_process模块的说明。

Node.js框架的child\_process模块提供了 4个创建子进程的函数，分别是spawn()方法、 exec()方法、execFile()方法、和fbrk()方法°其中spawn()方法是最原始的创建子进程的方法, 其他3个都是对spawn()方法不同程度地封装◎

spawn()方法只能运行指定的程序，参数需要在列表中给出；而exec()方法可以直接运 行复杂的命令；与默认的spawn()方法不同的是，fork。方法会在主进程与子进程之间直接 建立一个IPC管道，用于主进程与子进程之间的通信。

另外，fbrk()方法会有一个问题，就是它只能运行JavaScript代码，CoffeeScript等其他 任何可编译到JavaScript的语言，则是无法通过fbrk()方法进行调用的。

第7章OS模块应用

对于软件开发而言，操作系统的重要性不言而喻，它是一切软件开发的平台和载体。 同样，Node.js框架为设计人员提供了操作系统(OS)模块，该模块提供了一系列与操作系 统相关的方法，不过这些方法相对简单，实现的功能也十分有限。下面我们一起来学习os 模块的使用方法。

本章主要包括以下内容。

* OS模块方法。
* OS模块常量。

・OS模块简单应用。

7.1获取当前操作系统类型与平台

无论你是工作在Windows系统下、还是linux系统中，再或是热门的iOS系统下,获 取当前操作系统类型与平台是一项非常基础的功能，Node.js框架为我们提供了 os.typeQ与 os.platform()方法来完成这项工作。

在本节这个简单的例程中，我们使用os.typeO方法打印输出当前操作系统类型，使用 os.platform()方法来打印输出当前操作系统平台。

本例ch07.os typejplatform.js主要代码如下：

01 /\*\*

02 \* ch07.os\_type\_platfbrm.js

03 7

04 consoleJnfo(M—- OS type() & platfrom() n);

05 console.tnfbO；

06 var os = require('os\*);

07 var type = os.typeQ;

08 console.infof 前操作系统类型为:’+ type);

09 console.info();

1. var platform = os.platform();
2. console.infof 当前操作系统平台为：’+ platform);
3. console.info();
4. console.info("«—- OS type() & platfromQ -");

**【代码分析】**

第06行引入OS模块，同时赋于变量os；第07行调用os.type。方法将当前操作系统 类型的信息赋于变量type；第08行使用变量type打印输出了操作系统类型；第10行调用 os.platfbrm（）方法将当前操作系统平台的信息赋于变量platform；第11行使用变量platform 打印输出了操作系统平台。

图7.1是本例程在Ubuntu环境下，使用WebStorm开发工具输出的结果。

Run

Run ch074»jype\_ptatform.js \* %

A . /usr/local/bin/node chQ7.os\_type\_platform js OS type() & platfro«()

■ 4

, 多疏续作票虻奏型方：Linux

E 3常操作彖统\*•舍力：linux

*枣L*

\* 冒 OS type() & platfrom()

? ® Process finished with exit code 0

图7.1

当前操作系统类型与平台（Ubuntu版）

从图7.1输出的结果可以看到，在Ubuntu系统下输出的操作系统类型信息与平台信息

是基本一致的，唯一的区别是首字母大小写的不同，细心的读者从上图中应该发现了。

同样，我们看看Windows环境下使用WebStorm开发工具输出的结果，如图7.2所示。

*•幼*

"D:\Program Files\JetBrains\WebStorm 9.0.l\bin\runnerw.exe" OS type() & platfrom()

当前操作系统类型为：Windows\_NT

当前操作系统平台为：**Win32**

OS type( ) & platfroni()

Process finished with exit code 0

图7.2当前操作系统类型与平台（Windows版）

**提**ZF：

1. 由于Ubuntu系统是基于Linux系统开发的，因此Node.js框架将其识别为Linux系 统类型与平台。
2. 对于Windows系统而言，笔者的操作系统为Windows 7 Ultimate 32bit Preview版（预 览版），因此Node.js框架将其类型识别为Windows\_NT （ 一般认为Windows 7系列是NT 系统的延续）、平台识别为Win32。
3. 获取操作系统默认的临时文件目录

在本节这个简单的例程中，我们使用os.tmpdir（）方法获取操作系统默认的临时文件目 录，本例ch07.os\_tmpdir.js主要代码如下：

/\*\*

01

02

03

04

05

06

07

08

09

10

11

\* ch07.os\_tmpdir.js

\*/

console.info(M OS tmpdir()

console.info();

var os = require('os\*);

var type = os.type();

var tmpdir = os.tmpdir();

console.info(当前，+ type + '系统临时目录:\* + tmpdir); console.info();

console.info("— OS tmpdir()

**【代码分析】**

第08行调用os.tmpdir()方法将当前操作系统类型的信息赋于变量tmpdir；第09行使用 变量tmpdir打印输出了系统默认的临时文件目录。

**图**7.3演示的是在Ubuntu环境下的结果。图7.4是在Windows环境下的结果。从图7.2 与图7.4输出的结果看，Ubuntu系统默认的临时文件目录为"/tmp” , Windows 7系统默 认的临时文件目录为“C:\Users\用户名\AppData\Local\Temp"。当然，系统临时文件目录 是可以通过设置改变的，但一般没有特殊情况，用户不会去这么做。

► \* MD:\Program Files\JetBrainsXWebStorm 9.0.l\bin\runnerw.exe"

**Hur ch07-os tmpdkrjt**

/usr/local/bin/node chG7.os\_t«pdir.]S OS t«pdir()

多潸Linux彖匏皓时目幻 /tup

OS tmpdirO

Process finished with exit code 0

. j OS tmpdir()

图 当前Windows\_NT系统临时目录：C:\Users\KING\AppData\Local\Temp

痒3

X if}» OS tmpdir()

- Process finished with exit code 0

**图**7.3

默认的临时文件目录(Ubuntu版)

图7.4默认的临时文件目录(Windows版)

说明：对于用户在操作系统中的每一个操作，在系统中都可能会留下“痕迹”，而记 录这个痕迹的一种常用方法就是保存成临时文件、并放置市系统临时文件夹中”用户在下 载、安装/卸载软件时、浏览网页时、在线观看视频时、收发电子邮件时、用QQ传输文件 时……都会留有“痕迹”，因此系统临时文件夹是非常重要的。

1. 获取操作系统主机名

在本节这个简单的例程中，我们使用os.hostname()方法获取操作系统主机名，本例 ch07.os\_hostname.js 主要代码如下：

01 /\*\*

02 \* ch07.os\_hostname.js

03 \*/

04 console.info("™- OS hostname()

05 console.info();

06 var os = require('os');

07 var hostname = os.hostname();

08 console.infoC操作系统主机名：1 + hostname);

09 console.info();

1. console.info(" OS hostname() ");

【代码分析】

第07行调用os.hostname()方法将当前操作系统主机名的信息赋于变量hostname；第 08行使用变量hostname打印输出了操作系统的主机名。

图7.5演示的是本例程在Ubuntu环境下的结果。图7.6是在Windows环境下的结果。

图7.5

/usr/local/bin/node ch07.os\_hostname.]S

OS hostname 0

操作系义主枇君：king-ThinkPad-X61

OS hostnameO

Process finished with exit code Q

获取操作系统主机名(Ubuntu版)

弱：§ ' "D: \Program Files\3etBrains\WebStcrm 9.0.1\ 螺、1 OS hostname。

翻W操作系统主机名：KING-PC 普H

**X**譬

*f '* ' Process finished with exit code 0

OS hostname()

图7.6获取操作系统主机名(Windows版)

说明：主机名的概念主要针对在一个局域网中，为方便主机与主机之间的区分，可以 为每台主机设置一个主机名(或称主机别名)，这样主机之间的相互访问就会容易得多(比 如，在局域网中根据每台主机的功能来为其命名主机名，这样用户在访问时就能很容易知 道该主机的用途)..

7.4获取操作系统架构

操作系统架构是一个相对专业的概念，如果读者学习过操作系统原理或计算机组成原 理相关知识的话，对其应该是不陌生的。简单来说，操作系统架构对应着操作系统指令集 的概念。

在本节这个简单的例程中，我们使用os.arch()方法获取操作系统架构的信息，本例 ch07.os\_arch.js主要代码如下：

01 /\*\*

02 \* ch07.os\_arch.js

03 \*/

04 console.infoC OS arch() —

05 console.info();

06 var os = requirefos\*);

07 var platform = os.platform();

08 console.infof 操作系统平台：'+ platform);

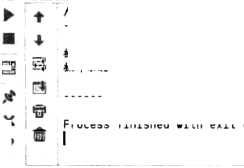
09 var arch = os.arch();

1. console.infof操作系统架构：\* + arch);
2. console.info();
3. console.info(" OS arch()*二*—”)；

**【代码分析】**

第07行调用os.platfbrm()方法将当前操作系统平台的信息赋于变量platform,并在第 08行打印输出了操作系统平台的信息：第09行调用os.archO方法将当前操作系统架构的信 息赋于变量arch；第10行使用变量arch打印输出了操作系统架构的信息(如图7.7所示， 本例程显示的信息为“ia32”，其含义是英特尔32位元架构)。

图7.7是在Ubuntu环境下的结果。图7.8是在Windows环境下的结果。

Rw" ♦-

/usr/local/bin/node ch07.os\_arch.]s OS arch()

OS archO

X

?

操作系St平台：linux 嫌作.彖统架构：ia32

Process finished with exit code 0

► ■> "D:\Program Files\JetBrains\WebStorm 9.0.1\

. $ OS arch()

**3**弓操作系统平台：**Win32**

**y r**操作系统架构：**ia32 X**昏

? © °S arch()

Process finished with exit code 0

图7.7获取操作系统架构(Ubuntu版)

关于操作系统CPU架构ia32的说明

图7.8获取操作系统架构(Windows版)

1. 英特尔32位元架构(英语：Intel Architecture, 32-bit,缩写为IA-32或ia32 ),也常 被称为i386、x86-32或x86,是由英特尔公司推出的指令集架构，至今英特尔最受欢迎的 处理器仍然采用此架构“
2. 操作系统CPU架构也可以理解为指令集架构，IA-32属于复杂指令集
3. 获取操作系统的发行版本

操作系统发行版本也是一个相对专业的概念，不同类型的操作系统均有其自身的发行 版本代号。

在本节这个简单的例程中，我们使用os.releaseO方法获取操作系统架构的发行版本， 本例ch07.os\_release.js主要代码如下：

01*广\**

02 \* ch07.os\_release.js

03 \*/

04 console.info(H OS release() ");

05 console.info(); *，*

06 var os = require('os');

07 var type = os.type();

08 console.info('操作系统类型:'+ type);

09 console.info();

1. var release = os.release();
2. console. info('操作系统发行版本:[[4]](#footnote-5) [[5]](#footnote-6) + release);
3. console. info();
4. console.info(H OS release() ");

**【代码分析】**

第09行调用os.release()方法将当前操作系统发行版本的信息赋于变量release；第10 行使用变量release打印输出了操作系统发行版本的信息(如图7.9所示，本例程显示的信 息为“3.13.0-49-generic”,其含义是基于GNU/linux的内核版本代号)。图7.10是在Windows 环境下的结果。

Run

Run ch07xsjtiQse.js 4

> . /usr/local/bin/node ch07.os\_release.js

OS release()

■ 4

撮作系晚矣空：Linux

昌弓

D 掘作票统复行帔本：3.13.0-49-genenc

X上

冒 OS released

X "

? ® Process finished with exit code 0

图7.9获取操作系统发行版本(Ubuntu版)

关于操作系统发行版本的说明

1. Linux其实是一个操作系统的内核，而一个完整的操作系统不仅仅是内核而已，还包 括很多扩展功能与应用。因此，许多个人、组织和企业开发了基于GNU/Linux的Linux发 行版(比如：Ubuntu系列、Debian系列、Fedora系列等)，但这些发行版使用的基本都是 Linux系统内核，本例程获取的“3.13.0-49-generic”信息，就是基于GNU/Linux的Linux 内核版本代号。
2. 同样，Windows系列操作系统也有发行版本代号，本例程中打印输出的数字序列

(6.1.7610 )信息，就是本机系统Windows 7 Ultimate Preview (预览)版的发行版本代号。

1. 返回操作系统运行时间

Node.js框架的OS模块为设计人员提供了一个非常有趣的功能，就是计算操作系统的 运行时间，这项功能是通过os.uptime()方法实现的。在本节这个简单的例程中，我们使用 os.uptime()方法获取操作系统运行时间，并通过一个转换函数将该时间按照“小时/分钟/秒” 的格式进行显示。

本例ch07.os\_uptime.js主要代码如下：

01 /\*\*

02 \* ch07.os\_uptime.js

03 \*/

04 console.info(M—■— OS uptime() —M);

05 console.info();

06 var os = requirefos\*); 〃引入OS操作系统模块

07 varuptime = os.uptime(); //计算操作系统运行时间

08 console.inf。('操作系统巳经运行了:' + uptime + \* seconds.\*);

09 console.info();

1. var sTime = convertTime(uptime);
2. console.info(\* 操作系统已经运行 了：' + sTime);
3. console.info();
4. console.info(" OS uptime() ");
5. /\*\*
6. \* Format time to Hours/Minutes/Seconds
7. \* @param uptime
8. \* ©returns {string}
9. 7
10. function convertTime(uptime) (
11. var runTime;
12. varhour
13. var minute;
14. varsecond;
15. hour = parselnt(uptime / 3600);
16. minute = parselnt((uptime - hour \* 3600) / 60);
17. second = uptime - hour \* 3600 - minute \* 60;
18. runTime - hour +' hours' + minute + 'minutes' + second + ' seconds.1;
19. return runTime;
20. }

**【代码分析】**

第07行调用os.uptime()方法将当前操作系统运行时间的信息赋于变量uptime,并在第 08行打印输出了操作系统运行时间的信息；第10行调用convertTime。方法将获取的操作 系统运行时间(单位为秒)转换为“小时/分钟/秒”的格式，并在第11行进行打印输出： 关于convertTime()方法的实现见第19〜29行，该方法分别折合计算出了小时、分钟和秒的 具体数值，其中小时和分钟数值使用了 JavaScript语言的parselnt()格式化整数的方法，秒 的数值仍按照实际小数保存。

图7.11是在Ubuntu环境下的结果，图7.12是在Windows环境下的结果。

说明：JavaScript语言的parselnt。方法用于将小数转换为整数，具体转换采用直接“丢 弃”小数部分的方式，而非“四舍五入”的方式。

Run

知n ch07.os\_uptimejs

> + /usr/Iocal/bin/node chO7.os\_uptime.js  
 OS uptime()

■，

续作员统巳菱走行了 ： 1909.8882切84 seconds.

F 操作彖统巳经适行了： 0 hours 31 minutes 49.888283583999964 seconds.

*X* 冒 OS uptime 0

? 微 Process finished with exit code 0

图7.11返回操作系统运行时间(Ubuntu版)

Run cWXoyiptiinajs ，， \_ . ■ r ■

A f "D:\Progr-am Files\]etBrains\WebStorni 9.0.l\bin\runnerw.exe" D:\no， 癫 4 OS uptime()

云M操作系统已经运行了 ： 1773.8362751 seconds.

-4

© 操作系统巳经运彳T了 ： 0 hours 29 minutes 33.836275099999966 seconds. 血'

? ------ OS uptime()

Process finished with exit code 0

图7.12返回操作系统运行时间（Windows版）

1. 获取操作系统内存状态

关于系统的内存状态,os模块为设计人员提供了两个方法，分别用于获取内存总量(使 用os.totalmem()方法获取)和空闲内存(使用os.freemem()方法获取)的信息。在本节这个 简单的例程中，我们使用这两个方法获取操作系统的内存状态信息。

本例ch07.os\_mem.js主要代码如下：

01 /\*\*

02 \* ch07.os\_mem.js

03 \*/

04 console.info(" OS mem() ”)；

05 console.info();

06 varos = require('os\*); //引入 OS 模块

07 var totalmem = os.totalmem();

08 console.info('操作系统内存总量为：'+ totalmem +1 byte.');

09 console.info('操作系统内存总量为：'+ parselnt(totalmem /1024 /1024) + 1 Mb.');

1. console.info();
2. var freemem = os.freemem();
3. console.infof操作系统空闲内存为：，+ freemem + ' byte.');
4. console.infof操作系统空闲内存为：'+ parselnt(freemem /1024 /1024) +' Mb.');
5. console.info();
6. console.infoC OS mem() ");

**【代码分析】**

第07行调用os.totalmem()方法将当前操作系统内存总量的信息赋于变量total mem,并 在第08行和第09行分别打印输出了以字节单位计算和以Mb单位计算的内存总量信息； 第11行调用os.freemem。方法将当前操作系统空闲内存大小的信息赋于变量freemem,并 在第12行和第13行分别打印输出了以字节单位计算和以Mb单位计算的空闲内存大小信 息。

图7.13是在Ubuntu环境下的结果，图7.14是在Windows环境下的结果。

/usr/local/bin/node chO7.os\_Ben.js

OS meiiO

“D:\Prografn Files\]etBrains\WebStorm 9.0.l\bin\runnerw.exe" OS mem()

埃作票饨内存垣•为：2619142144 byte. 操作素貌有奔忌量力：2497 Mb.

操作系统内存总量为：3493965824 byte. 操作系统内存怠量为：3332 Mb.

操作蜃统主同龙存为；230633472 byte. 操作京史主闵内存无：219 Mb

操作系统空闲内存为：1767100416 byte. 操作系统空闲内存为：1685 Mb.

OS mem()

OS mem()

图7.13操作系统内存状态(Ubuntu版)

图7.14操作系统内存状态(Windows版)

Process finished with exit code 0

Process finished with exit code 0

7.8获取CPU内核信息

在6.12节中，我们使用过os.cpus。这个方法获取CPU内核数量的信息，在本节这个简 单的例程中，我们继续使用os.cpus()这个方法获取CPU内核的相关信息。

本例ch07.os\_cpus.js主要代码如下：

01 *I\*\**

02 \* ch07.os\_cpus.js

03 \*/

04 console.info(" OS cpus()

05 console.info();

06 var os = require('os');

〃引入OS模块

07 var cpus = os.cpus();

08 console.info(cpus);

09 console.info();

1. console.info(" OS cpus()

")；

**【代码分析】**

第07行调用os.cpus()方法获取当前CPU内核的信息并腻于变量cpus,并在第08行中 进行打印输出。

图7.15是在Ubuntu环境下的结果，从打印输出的信息可以看到，笔者安装的这台 Ubuntu 系统机器为双核 CPU,型号为 “Intel(R) Core(TM)2 Duo CPU T9300” ,主频为 2.50GHz,属于较古老的硬件配置。

Run 60?6\_中习$ £.





/usr/local/bin/node chO7.os\_cpus.js OS cpusO

[{ model： 'Intel(R) Core(TM)2 Duo CPU T93G0 *9* 2.50GHz'.

speed； 2501,

tines：

{ user： 3769100, nice； 8400. sys： 934600. idle- 36043500. irq： 2000 } }.

( nodel. •Intel(R) Core(TM)2 Duo CPU T9300 @ 2.50GHz', speed I 2500, times;

{ user； 3843400, nice： 5700, sys： 934000. idle： 35981800, irq： 2600 } } I

OS cpusO

Process finished with exit code 0

图7.15获取CPU内核信息

(Ubuntu 版)

图7.16是在Windows环境下的结果，从打印输出的信息可以看到,

笔者安装的这台

Windows 7 系统机器为四核 CPU,型号为 “Intel(R) Core(TM) i5-3210M

CPU” ,主频为

2.50GHz,属于较新的硬件配置。

"D:\Program Files\JetBrains\Web5torm 9.0.1\bin\runnervj.exe" D:\nodejs\node.exe OS cpus()

( model: ,Intel(R) Core(TM) 15-3210M CPU @ 2.50GHz',

speed: 2494,

times: { user: 184361, nice: *Q,* sys: 203784, idle: 1538450, irq: 936 } ), { model: •intel(R) Core(TM) i5-3210M CPU fi 2.50GHz',

speed: 2494,

times: ( user: 52603, nice: 0, sys: 91978, idle: 1781359, irq: 22308 } },

( model: 'lntel(R) Core(TM) i5-3210M CPU @ 2.50GHz',

speed: 2494,

times: ( user: 209665, nice: 0, sys: 201397, idle: 1514754, irq: 514 } }, { model: 'lntel(R) Core(TM) i5-3210M CPU @ 2.50GHz',

speed: 2494,

times: ( user: 307©©, nice: *0,* sys: 36809, idle: 18S8190, irq: 296 } )]

OS cpus()

Process finished with exit code 0

图7.16获取CPU内核信息(Windows版)

7.9获取网络接口信息

Node.js框架的OS模块为设计人员提供了一个获取网络接口信息的简单方法。在本节 这个例程中，我们使用os.networkinterfaces()这个方法看一下具体获取了多少关于网络接口 的信息。

本例 ch07.os\_networklnterfaces.js 主要代码如下：

01 /\*\*

02 \* ch07.os\_networklnterfaces.js

03 7

148 -超实用的Node.js代码段

04 console.infof OS os\_networklnterfaces() ”);

05 console.info();

06 var os = require('os'); //引入 OS 模块

07 var networkinterfaces = os.networklnterfaces();

08 console, info(networklnterfaces);

09 console.info();

1. console.info("-- OS os\_networklnterfaces() ");

【代码分析】

第07行调用os.networklnterfaces（）方法获取网络接口的有关信息并赋于变量

networkinterfaceso图7.17是在Ubuntu环境下的结果，图7.18是在Windows环境下的结果。

/usr/bin/node chO7.os networkinterfaces.js

- OS os\_networklnterfaces()

仕 11°；

座[整： [{ address: '127.0.0.1', family： 'IPv4', internal； true },

'鼻： { address； ' ：: 1'. family； 'IPv6', internal； true }],

瓣 J : ethO:

,唇 [{ address： ■192.168,1.114', family： 'IPv4', internal： false }. X :{ address: 'feSQ: :21d:72ff ^686:2810'. 争号募 family： 'IPv6',

internal： false } ] }

OS os\_netvorklnterfaces()

Process finished with exit code 0

图7.17获取网络接口信息（Ubuntu版）

MD:\Program Files\3etBrains\WebStorm 9.0.l\bin\runnerw.exe" D:\nodejs\node.exe

OS os\_networklnterfaces() {，本地部，：

[{ address: \* fe80::4d9:15e6:e72c:54f31,

family: 'IPv6',

internal: false ),

{ address: \*192.168.1.115', family: \* IPv4\*, internal: false }],

1 Loopback Pseudo-Interface 1':

[{ address: \*::1\*, family: \* IPv6', internal: true },

( address: '127.0.0.1\*, family: \* IPv4', internal: true }],

, Teredo Tunneling Pseudo-Interface \*:

[{ address: '2001:0:5ef5:79fb:2400:le84:3f57:fe8c \*,

family: 'IPv6')

internal: false }4

( address: 'fe80::2400:le84:3f57:fe8c1,

***family:*** 'IPv6\*,

internal: false } ] }

Process finished with exit code 0

图7.18获取网络接口信息（Windows版）

第8章文件系统模块应用

设计人员在程序开发过程中，常常会有文件I/O操作的需求。Node.js框架为设计人员 提供了一个文件系统(File System)模块来支持I/O操作，文件系统模块的I/O操作是对标 准POSIX函数的简单封装，其提供了文件的读取、写入、更名、删除、遍历目录、链接等 POS1X文件系统操作。

POSIX的含义为可移植操作系统接口 (Portable Operating System Interface,缩写为 POSIX) , POSIX标准定义了操作系统应该为应用程序提供的接口标准，意在实现源代码 级别的软件可移植性。换而言之，为一个POSIX兼容的操作系统编写的程序，应该可以在 任何其他的POSIX操作系统上编译执行。

文件系统模块与Node.js框架的其他模块不同的是，该模块中对文件的所有操作都提供 了异步的和同步的两个版本。例如，读取文件内容的方法有异步的fs.readFile()和同步的 fs.readFileSync()o

本章主要包括以下内容。

* 文件系统模块同步方法。
* 文件系统模块异步方法。

8.1重命名文件

本章关于文件系统模块的第一个应用例程，我们向读者介绍文件重命名的方法。重命 名文件是任何一款操作系统都提供的最基础的功能，连网盘也提供重命名文件的功能。文 件系统模块提供了 fs.rename()与fs.renameSync()方法来完成重命名操作。

在本节这个例程中，我们分别使用这两个方法完成同一项功能操作，测试一下同步方 法与异步方法的异同点。

1. 重命名同步方法

首先，我们先介绍同步方法的例程(代码文件名称：chO8.fs\_renameSync.js),其主要 代码如下：

01 /\*\*

02 \* chO8.fs\_renameSync.js

03 \*/

04 console.info(" fs rename()

05 console.info();

06 varfs = require('fs'); //引入文件系统模块

07 console-timeffs-rename-syn\*); 〃定义时间开始标记

08 fs.renameSyncCtxt/helloworld-syn.txt', 'txt/helloworld-syn-rename.txt'); //文件重命名

09 console.timeEndCfs-rename-syn\*); //定义时间结束标记

1. console.info();
2. var exec = require(\*child\_process').exec; //引入 childjorocess 模块
3. var child\_pre = exec(\*cat txt/helloworld-syn.txt',
4. function (error, stdout, stderr) (
5. console.info('cat txt/helloworld-syn.txt stdout:');
6. console.log(stdout); 〃打印输出 stdout
7. console.log(stderr); "打印输出 stderr
8. ))；
9. console.info();
10. var child\_suf = exec('cat txt/helloworld-syn-rename.txf,
11. function (error, stdout, stderr) {
12. console.info('cat txt/helloworld-syn-rename.txt stdout: \*);
13. console.log(stdout); 〃打印输出 stdout
14. console.log(stderr); 〃打印输出 stdout
15. })；

【代码分析】

第06行引入文件系统(fs)模块，同时赋于变量fs；第07行定义一个时间开始标记 (fs-rename-syn),用于标记fs.renameSync()方法(同步版本)的时间；第08行调用 fs.renameSync()方法执行文件重命名操作;第09行定义一个时间结束标记('fs-rename-syn1 )； 第12~17行使用exec()方法打印输出了 helloworld-syn.txt的内容。同样的，在第19~24 行使用exec()方法打印输出了重命名后的helloworld-syn-rename.txt的内容。

图8.1是在Ubuntu环境下的结果。 一

从中可以看到，在Ubuntu系统中执行 fs.renameSync()方法用于文件重命名(同步 版本)的操作耗时0毫秒，并且没有提示操 作发生错误；在第12〜17行打印输出 helloworld-syn.txt文件的内容时，发成了错 误(提示：“没有那个文件和目录”)，表 明该文件不存在(已经被重命名了)；而第 19〜24 行打印输出 helloworld-syn-rename.txt 文件的内容时，操作成功完成了(输出了文 本内容：“hello world - syn ”)，说明 fs.renameSync()方法操作确实成功完成了。

Run

Run , chOBfs.reffiimeSyncjs \_ . ； 〔奉，4-

萨 a /usr/local/bin/node ch08.fsrenaoeSync.js

fs rename 0

» 4 ；\_\_ fs-rename-syn： Oms

n j

*沸* J cat txt/helloworld-syn.txt stdout：

K \_ cat: txt/helloworld-syn.txt:及考那个文井或舍录

*7 :®*

\* cat txt/helloworld-syn-rename.txt stdout：

' hello world - syn

Process finished with exit code 0

图8.1文件系统重命名的方法(同步版本)

1. **重命名异步方法**

上面介绍的是文件重命名同步方法，下面我们介绍文件重命名异步方法的例程(代码

文件名称：chO8.fs\_rename.js),其主要代码如下:

01

02

03

04

05

06

07

08

09

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

/\*\*

\* ch08.fs\_rename.js

\*/

console.info(" fs renamef)

console.info();

var fs = requireffs');

console.time('fs-rename-asyn');

•文件重命名

7

fs.renameftxt/helloworld-asyn.txt\*, 'txt/helloworld-asyn-rename.txt\*, function (err)( console.timeftime-10000x10000');

for (var i=0; i<10000; i++) (

for (var j=0;j<10000; j++)(

}

}

console.timeEnd('time-10000x10000'); if (err) throw err;

console.logfrenamed complete');

//引入文件系统模块 〃定义时间开始标记

〃定义时间开始标记

//定义时间结束标记

})；

console.timeEnd('fs-rename-asyn')；

console.info();

var exec = requirefchild\_process^.exec;

var child\_pre = execfcat txt/helloworld-asyn.txt\*, function (error, stdout, stderr) ( console.infofcat txVhelloworld-asyn.txt stdout:'); console.log(stdout); 〃打印输出 stdout

console.log(stderr); 〃打印输出 stderr

//定义时间结束标记

//引入 child\_process 模块

})；

console.info();

var child\_suf = exec(\*cat txt/helloworld-asyn-rename.txt', function (error, stdout, stderr) (

console.infofcat txt/helloworld-asyn-rename.txt stdout:'); console.log(stdout); 〃打印输出 stdout

console.log(stderr); 〃打印输出 stdout

})；

**【代码分析】**

第11-21行调用fe.rename()方法执行文件重命名操作；为了测试文件重命名异步操作， 我们预先在代码文件目录中新建了--个名称为txt的子目录，并在该子目录下创建了一个名 称为helloworld-asyn.txt的文件，然后通过fs.rename()方法进行文件重命名操作。

第11〜21行我们使用fbr语句打印输出了一个二维空循环(即无任何具体操作，仅仅

是用于执行一个时间延迟)，并使用时间标记^-10000x10000记录了时间；第19行表示 如果文件重命名操作发生错误，使用err参数打印输出错误提示信息；第20行表示如果文 件重命名操作成功，则打印输出成功提示信息；第22行定义了一个时间结束标记!fs-rename- asyno

图8.2是在Ubuntu环境下的结果。从中可以看到，在Ubuntu系统中执行fs.rename() 方法用于文件重命名(异步版本)的操作耗时4毫秒，并且提示操作成功完成；而在第 11〜21行打印输出的空循环耗时188毫秒，远远大于异步文件重命名操作的用时，表明 了 fs.rename()方法确实是异步执行的。

另外，第24~30行在打印输出helloworld-asyn.txt文件的内容时，发成了错误(提示： 没有那个文件和目录)，表明该文件不存在(已经被重命名了)；而第32〜37行打印输出 helloworld-asyn-rename.txt文件的内容时，操作成功完成了(输出了文本内容：uhello world -asyn”)*，*说明fs.rename()方法操作确实成功完成了。

Run

： ch08.fs\_rename.js \*\* ■

> I， 1 /usr/local/bin/node ch08.fs rename.js

fs renaie ()

**■ 4 :**

fs-rename-asyn： 4«s

图殍:

X- ti«e-100G0x10000: 188ms

辱 renamed complete

***X*** cat txt/hellovorld-syn.txt stdout：

7 © ：

' cat： txt/helloworld-asyn.txt;；殳有邪令丈伴或留条

cat txt/helloworld-syn-renaaie.txt stdout： hello vorid - asyn

Process ■finished with exit code 0

图8.2文件系统重命名的方法(异步版本)

提示：在使用文件重命名操作时，一般建议优先选用异步方法。其实，这样做的原因 很简单，在使用同步方法操作时，假如发生了异常情况，文件操作可能会陷入无效的终止 状态，程序就会无法继续执行了。

8.2打开与关闭文件

对一个文件的操作，最简单的就是打开和关闭了。文件系统模块提供了 fs.open()与 fs.openSync()这两个方法来完成打开文件的操作，同时相对应地也提供了 fs.close()与 fs.closeSync()这两个方法来完成关闭文件的操作。

1. **文件打开的同步方法**

首先，我们先介绍同步方法的例程(代码文件名称：ch08.fs\_openSync.js),其主要代 码如下：

01 /\*\*

02 \* ch08.fs\_openSync.js

03 \*/

04 console.info(" fs openSync() ");

05 console.info();

06 var exec = require(，child\_process)exec; 〃引 入 child\_process 模块

07 var child\_pre = exec('cat txt/openSync.txt\

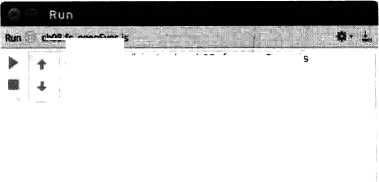
|  |  |  |
| --- | --- | --- |
| 08 | function (error, stdout, stderr) { |  |
| 09 | console.info('cat txt/openSync.txt stdout:'); | |
| 10 | console.log(stdout); | 〃打印输出stdout |
| 11 | console.log(stderr); | 〃打印输出stderr |
| 12 | })； |  |

1. console.info();
2. varfs = require('fs'); 〃引入文件系统模块
3. var fd = fs.openSync(\*txt/openSync.txt', 'a'); 〃打开文件(同步方法)
4. console.infof\*文件描述符："+ fd);
5. console.log('fs.openSync() Done.');
6. console.info();
7. fs.closeSync(fd);
8. console.infof 文件描述符：” + fd);
9. console.log('fs.closeSync() Done.');
10. console.info();

**【代码分析】**

为了测试文件打开与关闭操作，我们在代码文件目录中的txt子目录下创建了一个名称 为openSync.txt的文件，然后通过fs.openSync()与fs.closeSyncQ方法进行文件打开与关闭的 操作；第06~12行使用exec()方法打印输出了 openSync.txt文件的内容；第14行引入文件 系统(fs)模块，同时赋于变量fs；第15行调用fs.openSync()方法执行文件打开操作；第 16行通过变量fd打印输出了 openSync.txt文件描述符具体的数值:第19行调用fs.closeSync() 方法执行文件关闭的操作。

图8.3是本例程在Ubuntu环境下的输出结果。从中可以看到，在Ubuntu系统中执行 fs.openSync()与fs.closeSync。方法进行文件打开与关闭(同步版本)的操作均获得了成功； 其返回的文件描述符参数也被成功打印输出了(本例为数值11) O



cat txt/open^nc.txt stdout： hello world - openSync 0

昏•丈件摭返符：11

fs.closeSyncO Done.

骂丈择撞返符；11 存 =fs.openSync() Done.

/us r/local/bin/node chOB.fs\_openSync.js

fs openSync0

Process finished with exit code Q

1. 文件打开的异步方法

上面介绍的是文件打开的同步方法，下面向读者介绍文件打开的异步方法例程(代码 文件名称：ch08.fs\_open.js),其主要代码如下：

|  |  |  |
| --- | --- | --- |
| 01 | /\*\* |  |
| 02 | \* ch08.fs\_open.js |  |
| 03 | 7 |  |
| 04 | console.infof\* fs open() |  |
| 05 | console.info(); |  |
| 06 | var exec = require('child\_process').exec; | 〃引入 child \_\_process 模块 |
| 07 | var child\_\_pre = exec('cat txt/open.txf, |  |
| 08 | function (error, stdout, stderr) ( |  |
| 09 | console.infbfcat txt/open.txt stdout:'); |  |
| 10 | console.log(stdout); | //打印输出stdout |
| 11 | console.log(stderr); | 〃打印输出stderr |
| 12 | })； |  |
| 13 | var fs = requireCfe\*); | 〃引入文件系统模块 |
| 14 |  |  |
| 15 | \*打开文件(异步方法) |  |
| 16 | \*/ |  |
| 17 | fe.open(\*txt/open.txt', 'a', function (err, fd) { |  |
| 18 | if (err)( |  |
| 19 | throw err; |  |
| 20 | } else ( |  |
| 21 | console.info("文件描述符:"+ fd); |  |
| 22 | console.log(fs.open() Done'); |  |
| 23 | console.info(); |  |
| 24 | ) |  |
| 25 | fe.close(fd, function (){ |  |
| 26 | console.info("文件描述符:"+ fd); |  |
| 27 | console.log(\*fs.dose() Done'); |  |
| 28 | console.info(); |  |
| 29 | }); |  |
| 30 | }); |  |
| 31 | cx)nsole.info(); |  |

**【代码分析】**

同样的，为了测试文件打开与关闭操作，我们在代码文件目录中的txt子目录下创建一 个名称为open.txt的文件，然后通过fs.open()与fs.close()方法进行文件打开与关闭的操作； 第06〜12行使用exec()方法打印输出了 open.txt文件的内容；第17—30行调用fs.open() 方法执行文件打开操作；第18〜12行表示如果文件打开操作发生错误，使用err参数打印 输出错误提示信息；第21〜23行表示如果文件打开操作成功，则打印输出文件描述符与操 作成功提示信息；第25〜29行调用fs.closeO执行文件关闭的操作。

图8.4是本例程在Ubuntu环境下的输出结果。从中可以看到，在Ubuntu系统中执行fs.open()与fs.close()方法进行文件打开与关闭(异步版本)的操作均获得了成功；其文件描 述符参数也被成功打印输出了(本例为数值11)。

/usr/local/bin/node ch08.fs\_open.js

fs open()

文锌播廷符：11

fs.openO Done

丈件儡也■符：11

.fs.close() Done

cat txt/open.txt stdout： hello world - open()

Process finished with exit code 0

图8.4打开与关闭文件的方法(异步版本)

说明：对于操作系统内核而言，所有打开的文件都通过文件描述符引用。文件描述符是 一个非负整数，当打开一个现有文件或创建一个新文件时，内核向进程返回一个文件描述符°

8.3截取文件内容

文件截取其实就是按照设定好的长度保留文件内容，其余部分将会被舍弃。假设我们 将辞海存放在文件中，当用户输入一个词时，我们找到这个词的解释并截取出来给用户看。 文件系统模块提供了 fs.ftruncate()与fs.ftruncateSync()这两个方法来完成文件截取的操作。

说明：前两节我们都分别举例介绍了同步方法和异步方法，读者应该已经知道了两者 的区别，鉴于篇幅，从本节开始，将只介绍一种方法的使用，但可以参考源代码给出的另 一种方法的使用。

下面我们介绍同步方法的例程(代码文件名称：chO8.fs\_ftruncateSync.js),其主要代 码如下：

/\*\*

01

02

03

04

05

06

07

08

09

10

11

12

13

\* chO8.fs\_ftruncateSync.js

\*/

console.infof' fs ftruncateSync(> ");

console.infoQ；

varfs = requireffs1); //引入文件系统模块

var exec = require('child\_process').exec; 〃引入 child\_process 模块 var child\_pre = exec(\*cat txt/ftruncateSync.txt',

function (error, stdout, stderr) {

console.infofcat txt/ftruncateSync.txt stdout: \*);

console.log(stdout); 〃打印输出 stdout

console.log(stderr); 〃打印输出 stderr

varfd = fs.openSyncCtxt/ftruncateSync.txf, 'a'); 〃打开文件(同步方法)

1. fs.ftruncateSync(fd, 10);

〃文件内容我取

〃打印输出文件描述符

1. console.info(H文件描述符："+ fd);
2. console.info();
3. console.logCfs.ftruncateSync() Done');
4. fs.closeSync(fd); 〃关闭文件(同步方法)
5. var child\_suf = exec('cat txVftruncateSync.txt',
6. function (error, stdout, stderr) (
7. console.info('cat txt/ftruncateSync.txt stdout:');
8. console.log(stdout); 〃打印输出 stdout
9. consoleJog(stderr); 〃打印输出 stdout

24

})；

1. console.info();
2. })；
3. console.info();

**【代码分析】**

为了测试文件截取操作，我们在代码文件目录中的txt子目录下创建一个名称为 ftruncateSync.txt的文件，然后通过fs.ftruncateSync。方法进行文件截取的操作；第06行引 入文件系统(fk)模块，同时赋于变量fs。

第13行调用fs.openSync()方法执行文件的打开操作(本例以“附加”的方式打开文件),

并返回文件描述符，同时赋于变量第14 行调用fs.ftruncateSync()方法(同步方法)执 行文件截取的操作；第15行通过变量fti打印 输出文件描述符信息(本例为10)；第17行 打印输出文件截取操作成功的信息。

Run

**Run thakfcJtwhcattSyntjs**

A + Zusr/local/bm/node chO8.fs\_ftruncateSync. ]s

fs ftruncateSyncO

■ #

留 野 cat txt/ftruncateSync.txt stdout:

— hello world - ftruncateSyncO

X 昏丈悴\*SiS导：10

? | © fs.ftruncateSyncO Done

cat txt/ftruncateSync.txt stdout： hello worl

Process finished with exit code 0

图8.5文件内容截取的方法(同步版本)

图8.5是本例程在Ubuntu环境下的输出结 果。从中可以看到，在Ubuntu系统中执行 fs.ftruncateSync()方法用于文件截取(同步版 本)的操作后，文件(flruncateSync.txt)的内 容由“hello world - ftruncateSync()"变为“hello worl”，表明操作成功完成了。

8.4修改文件长度

文件系统模块提供了 fs.truncate()与fs.truncateSync()这两个方法来完成修改文件长度的 操作。fs.truncate()与fs.truncateSync()这两个方法在使用上与8.3节的 fc.ftruncate()与 fs.ftruncateSync()这两个方法类似，区别是这两个方法可以直接使用文件路径进行操作，具 体看下面的例程。

我们介绍同步方法的例程(代码文件名称:chO8.fs\_truncateSync.js)*，*其主要代码如下:

01

02

03

04

05

06

07

08

09

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

})；

console.info();

/\*\*

\* chO8.fs\_truncateSync.js

\*/

console.info(M- console.info();

fs truncateSync()

varfs = require(fs'); //引入文件系统模块

var exec = require('child \_\_process\*).exec; //引入 child\_process 模块

var child\_pre = exec('cat txt/truncateSync.txt,,

function (error, stdout, stderr) (

console.infoCcat txt/truncateSync.txt stdout:'); console.log(stdout); 〃打印输出 stdout

console.logflength:' + stdout.length); 〃输出数据长度 console.log(stderr); 〃打印输出 stderr

console.info();

fs.truncateSync('txt/truncateSync.txt', 6); 〃修改文件内容长度(同步方法) console.log('fs.truncateSync() Done');

var child\_suf = exec('cat txt/truncateSync.txt\

function (error, stdout, stderr) (

console.infofcat txt/truncateSync.txt stdout:'); console.log(stdout); 〃打印输出 stdout

console.log('length:' + stdoutlength); 〃输出数据长度 console.log(stderr); 〃打印输出 stdout

console.info();

})；

**【代码分析】**

为了测试修改文件长度的操作，我们在代码文件目录中的txt子目录下创建一个名称为 truncateSync.txt的文件，然后通过fs.truncateSync()方法修改文件长度。

第07~13行使用exec()方法打印输出了 truncateSync.txt文件的内容与长度，；第12 行通过参数stdout的length属性,打印输出了 truncateSync.txt文件内容的长度(本例为27)； 第15行调用fs.truncateSync()方法(同步方法) 执行修改文件长度的操作；第16行打印输出修 改文件长度操作成功的信息(fs.truncateSync() Done.):第21行通过参数stdout的length属性， 打印输出了 truncateSync.txt文件内容修改后的 长度(本例为6) o

图8.6是本例程在Ubuntu环境下的输出结 果。从中可以看到，在Ubuntu系统中执行 fs.truncateSync()方法修改文件长度(同步版本)

Rui

**\* %**

/usr/local/bm/node ch08. fs\_truncateSync ]s fs truncateSync()

cat txt/truncateSync.txt stdout： 1234567890 - truncateSync() length： 27

fs.truncateSyncO Done

cat txt/truncateSync.txt stdout： 123456

length： 6

Process finished with exit code 0

的操作后，文件(truncateSync.txt)的内容由"1234567890 - truncate。” 变为"123456”， 表明操作成功完成了。

提示：从8.3与8.4这两节的例程来看，fs.truncate()和fs.truncateSync()这两个方法与 fs.ftruncate()和fs.ftruncateSync()这两个方法在功能与效果上基本类似。唯一有区别的是， fs.truncate()和fs.truncateSyncO这两个方法直接使 用文件路径，而fs.ftruncate()和 fs.ftruncateSync()这两个方法需要使用文件描述符进行操作。相对而言，fs.truncate()和 fs.truncateSync()这两个方法更方便些，因为其可以直接使用文件路径进行操作。

8.5获取文件信息

获取文件信息的方法有 4 个，分别是 fs.stat。、fs.statSync。、fs.fstat。、fs.fstatSync()o 其中，fs.statO与fs.statSync()这两个方法可以直接使用文件路径进行操作，而fs.&tat()与 fs.fstatSync()这两个方法需要使用文件描述符进行操作。

我们以fs.statSync()这个同步方法为例(代码文件名称：chO8.fs\_statSync.js),其主要 代码如下：

01 /\*\*

02 \* chO8.fs\_statSync.js

03 \*/

04 console.info(" fs statSync() M);

05 console.info();

06 varfs = requireffs\*);

//引入文件系统模块

//获取文件信息(同步方法)

//打印输出文件信息

07 var statSync = fs.statSync(\*txt/statSync.txt');

08 console.info('txt/statSync.txt file info:');

09 console.info(statSync);

1. console.info();
2. console.infoCfs.statSyncO Done.');
3. console.info();

**【代码分析】**

为了测试获取文件信息的操作，我们在代码文件目录中的txt子目录下创建了一个名称 为statSync.txt的文件，然后通过fs.statSync()方法进行获取文件信息的操作；第07行调用 fs.statSync()方法(同步方法)实现获取文件信息的操作，该方法返回一个fs.Stats对象(本 例保存在变量statSync中)，用来保存文件信息；下面是官方文档中，对于fs.Stats对象的 举例说明：

{dev: 2114, ino: 48064969, mode: 33188, nlink: 1, uid: 85,

gid: 100,

rdev: 0,

size: 527,

blksize: 4096,

blocks: 8,

atime: Mon, 10 Oct 2011 23:24:11 GMT,

mtime: *Mon,* 10 Oct 2011 23:24:11 GMT,

ctime: Mon, 10 Oct 2011 23:24:11 GMT,

birthtime: Mon, 10Oct2011 23:24:11 GMT}

上面各项数据均是对文件信息的详细描述，比如dev表示设备号，size表示文件大小, birthtime表示创建时间等，更详细的说明，读者可以参考官方文档，在此就不详细解释了。 图8.7是本例程在Ubuntu环境下的输出结果。

Run

财— \_

/usr/local/bm/node chG8.fs\_statSync. ]s

 fs statSync()

txt/statSync.txt file info：

{ dev： 2049,

node： 332G4,

nlink： 1,

uid: 1000,

gid: 1000, rdev ； 0, blksize： 4096, ino： 3808574.

size： 32,

blocks： 6,

atine： Tue Apr 21 2Q15 11:21:24 GMT+G80G (CST),

ntue： Tue Apr 21 2015 11:14:51 GMT+0800 (CST).

ctiae： Tue Apr 21 2015 11:14:51 GMT+0800 (CST) }

fs.statSync() Done.

Process finished with exit code Q

图8.7获取文件信息的方法(fs.statSync。同步版本)

8.6更改文件所有权

使用过ERP或者数据库的人都知道，为了保护数据的重要性，不同的角色会具备不同

的权限。在Windows操作中，有Administrator用户和Administrators群组，某些文件的所 有者一般都是群组，具备对文件“完全控制”的能力。

Node.js 中，文件系统模块提供了 fs.chown()>fs.chownSync()>fs.fchownO>fs.fchownSyncO 这四个方法来更改文件的所有权。

其中，fs.chown()与fs.chownSync()这两个方法可以直接使用文件路径进行操作，而 fs.fchown()与fs.fbhownSync。这两个方法需要使用文件描述符进行操作。具体操作时，这4 个方法均需要用户具有操作文件的权限。

我们以fs.chownSync()这个同步方法为例(代码文件名称：ch08.fs\_chownSync.js),其

主要代码如下：

**【代码分析】**

01 /\*\*

02 \* ch08.fs\_chownSync.js

03

04

05

06

07

\*/

console.infof' fs chownSync()

console.info();

var fs = requireffs');

var statSync\_pre = fs.statSyncCtxt/chownSync.txt1);

”)；

//引入文件系统模块

//获取文件信息（同步方法）

08 console.infoCtxVchownSync.txt file info:');

09

10

11

12

13

14

15

16

*17*

18

19

console.info(statSync\_pre);

console.info();

console.infoCfs.chownSyncO Done.');

console.info();

fs.chownSyncCtxt/chownSync.txt', 1001,1001); console.infoCfs.chownSyncO Done.\*);

console.info();

var statSync\_suf = fs.statSync('txVchownSync.txt'); console.infoftxt/chownSync.txt file info:'); console.info(statSync\_suf);

console.info();

//打印输出文件信息

〃更改文件所有权（同步方法）

//获取文件信息（同步方法）

//打印输出文件信息

为了测试更改文件所有权的操作，我们在代码文件目录中的txt子目录下创建一个名称 为chownSync.txt的文件，然后通过fs.chownSync（）方法更改文件所有权。

第09行通过变量statSync\_pre打印输出文件chownSync.txt的文件信息；第13行调用

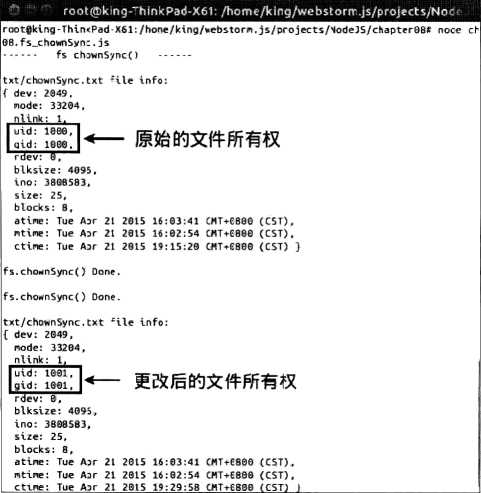
fs.chownSync（）方法（同步方式） 实现更改文件所有权的操作：第 16行再次调用fs.statSync（）方法 （同步方式）获取文件信息，并 保存在变量statSync\_suf中；第 18行通过变量statSync\_suf打印 输出已经更改了所有权的文件 chownSync.txt的文件信息。

图8.8更改文件所有权的方法（fs.chownSync。同步版本）

图8.8是本例程在Ubuntu环 境下，使用控制台终端调试输出 的结果（注意需要管理员权限进 行操作）。从中可以看到，在 Ubuntu 系统中执行 fs.chownSync（） 方法,将文件chownSync.txt的uid 与gid数值由1000更改为1001, 说明更改文件所有权的操作成功 完成了。

提示：在Ubuntu环境下，更改文件所有权的操作需要用户具有一定的权限，本节的例 程均是在系统管理员权限下的控制台窗口完成的。如果读者在WebStorm集成开发环境中 调试运行，可能会出现无法操作文件的错误，这是因为WebStorm开发软件没有系统管理 员的权限，

8.7更改文件权限

普通的用广"-般对文件有读、写权限，如果要操作系统文件，大部分用户就只具备读 权限了，要更改文件的读、写权限，可怎么办呢？文件系统模块提供了 fs.chmod。、 fs.chmodSync。、fs.fbhmod。、fs.fchmodSync()这四个方法来完成更改文件的权限。

其中，fs.chmod()与fs.chmodSync()这两个方法可以直接使用文件路径进行操作，而 fs.fchmod()与fs.fchmodSync。这两个方法需要使用文件描述符进行操作。具体操作时，这4 个方法均需要用户具有操作文件的权限。

我们以fs.chmodSync()这个同步方法为例进行介绍(代码文件名称:ch08.fs\_chmodSync.js), 其主要代码如下：

01 /\*\*

02 \* ch08.fs\_chmodSync.js

03 \*/

04 console.info(" fs chmodSync()-—");

05 console.info();

06 var fs = require('fs');

〃引入文件系统模块

〃引入 child\_process 模块

//定义命令行’Is-lh/txt'

07 var spawn = require('child\_process').spawn;

08 var ls\_txt\_pre = spawn('ls', 'txt/chmodSync.txt\*]);

09 /\*\*

1. \*捕获控制台输出对象stdout,输出捕获数据
2. \*/
3. ls\_txt\_pre.stdout.on('data', function (data) (
4. console.log(\*stdout:' + data);
5. fs.chmodSync('txt/chmodSync.txt\ 777); 〃更改文件权限(同步方法)
6. console.info('chmodSync() Done.');
7. console.info();
8. var ls\_txt\_suf = spawnfls', ['-lh', 'txt/chmodSync.txt']); 〃定义命令行 \*ls -lh /txt'
9. /\*\*
10. \*捕获控制台输出对象stdout,输出捕获数据
11. \*/
12. ls\_txt\_suf.stdout.on('data', function (data) (
13. console.Iog('stdout: \* + data);
14. ))；
15. /\*\*
16. \*绑定系统error事件
17. \*/
18. ls\_txt\_suf.on(,error\*, function (code) (
19. console.log('child process error with code' + code);
20. console.info();
21. })；
22. /\*\*
23. \*绑定系统close事件
24. \*/
25. ls\_txt\_suf.on('close', function (code) (
26. console.logfchild process closed with code' + code);
27. console.info();
28. ))；
29. })；
30. /\*\*
31. \*绑定系统error事件
32. \*/
33. ls\_txt\_pre.on(,error,, function (code) (
34. console.logfchild process error with code' + code);
35. console.info();
36. ))；
37. /\*\*
38. \*绑定系统close事件
39. \*/
40. ls\_txt\_pre.on(,close,, function (code) (
41. console.log('child process closed with code ' + code);
42. console.info();
43. })；

**【代码分析】**

为了测试更改文件权限的操作，我们在代码文件目录中的txt子目录下创建一个名称为 chmodSync.txt文件，然后通过fs.chmodSync()方法更改文件权限。

第07行引入子进程(child\_process)模块的spawn()方法创建子进程，同时赋于变量 spawn；第08行通过spawn。方法对文件chmodSync.txt执行命令行(Ts-lh")来查看文 件信息；第12行通过捕获控制台stdout对象打印输出查询到的信息，这些信息是 chmodSync.txt文件未被更改权限前(实际权限为000)的文件信息(如图8.17所示)。

第40〜45行通过绑定系统error事件，监控执行spawn。方法时是否出现错误异常情况, 如果出现则打印输出错误信息；第46〜52行通过绑定系统close事件监控spawn。方法是否 执行完毕。。

第14行调用fs.chmodSync()方法(同步方法)实现更改文件权限的操作；第15行打印 输出了 fs.chmodSync()方法操作成功的提示信息('chmodSync() Done.')；第16~37行再次 通过调用子进程的方法，查看了文件chmodSync.txt被更改了权限后的文件信息(如图8.9 所示)。



Run chOa-fs.chmodSyncjs

king 14 4-弓 22 09：59 txt/chinodSync. txt

文件更改前的权限

, /usr/local/bin/node ch08. fs\_chi»odSync. js fs chmodSync()

stdout：

chaodSynct) Done.

child process closed with code G stdout：!-r- — x--t 1 king king 14 4-月 22 09：59 txt/ch«odSync.txt child process close^^ith code G

'文件更改后的权限

Process finished with exit code 0

图8.9更改文件权限的方法(fkchmodSync()同步版本)

从图8.9可以看到，在Ubuntu系统中执行fs.chmodSync()方法，将文件chmodSync.txt 的mode权限由000更改为777,说明更改文件权限的操作成功了。

8.8创建与删除文件硬链接

链接是对文件的引用，硬链接只能引用同一系统中的文件。当移动或删除原始文件时， 硬链接不会被破坏，因为它所引用的是文件的物理数据而不是文件在文件结构中的位置。 硬链接的文件不需要用户有访问原始文件的权限，也不会显示原始文件的位置，这样有助 于文件的安全。

文件系统模块提供了 fs.link()> fs.unlink()> fs.linkSync()> fs.unlinkSync()这四个方法来 完成创建与删除文件硬链接的操作。其中，fs.linkSync()与fs.unlinkSync()是同步方式的一组 方法，而fs.link()与fs.unlink()是异步方式的一组方法。

我们介绍同步方法的例程(代码文件名称：chO8.fs\_linkSync.js),其主要代码如下：

01 /\*\*

02 \* chO8.fs\_linkSync.js

03 7

|  |  |  |
| --- | --- | --- |
| 04 | console.info(n™ fs linkSync()—"); | |
| 05 | console.info(); |  |
| 06 | var exec = require('child\_process)exec; | 〃引入 child\_process 模块 |
| 07 | var child\_pre = exec('cat txt/linkSync.txt,, |  |
| 08 | function (error, stdout, stderr) ( |  |
| 09 | console.infofcat txt/linkSync.txt stdout: \*); | |
| 10 | console.log(stdout); | "打印输出stdout  *i* |
| 11 | console.log(stderr); | 〃打印输出stderr |
| 12 | })； |  |
| 13 | console.info(); |  |
| 14 | var srcpath = "txt/linkSync.txt"; |  |
| 15 | var dstpath = Htxtlink/linkSync.txt"; |  |
| 16 | var fs = require('fs\*); | 〃引入文件系统模块 |

1. fs.linkSync(srcpath, dstpath); 〃链接文件(同步方式)
2. console.log('fs.linkSync() Done.\*);
3. console.info();
4. var child\_suf = exec('cat txtlink/linkSync.txt',
5. function (error, stdout, stderr) (
6. console.infofcat txtlink/linkSync.txt stdout:');
7. console.log(stdout); 〃打印输出 stdout
8. console.log(stderr); 〃打印输出 stderr
9. fs.unlinkSync(dstpath);
10. console.logCfs.unlinkSyncO Done.');
11. console.info();
12. var spawn = require('child\_process').spawn; //引 入 child\_process 模块
13. var ls\_txt = spawnCIs\*, f-lh', 'txtlinkl); //定义命令行 Ms -Ih /txtlink\*
14. /\*\*
15. \*捕获控制台输出对象stdout,输出捕获数据
16. 7
17. ls\_txt.stdout.on('data\*, function (data) {
18. console.info('ls -Ih /txtlink stdout:');
19. console.log(\*stdout:' + data);
20. console.info();
21. })；
22. /\*\*
23. \*绑定系统error事件

40 7

1. ls\_txt.on(\*error', function (code) (
2. console.logfchild process error with code ' + code);
3. console.info();
4. })；
5. /\*\*
6. \*绑定系统close事件
7. \*/
8. ls\_txt.onfclose\*, function (code) (
9. console.logCchild process closed with code' + code);
10. console.info();
11. })；
12. })；
13. console.info();

**【代码分析】**

为了测试创建与删除文件硬链接的操作，我们预先在代码文件目录中新建两个子目录, 一个名称为txt,用于存放原始文件，另一个名称为txtlink,用于存放硬链接生成的文件。 本例程需要先在txt子目录下创建一个名称为linkSync.txt的文件，然后通过fs.linkSync() 方法进行创建与删除文件硬链接的操作。

第25行调用fs.unlinkSync()方法(同步方式)执行删除文件硬链接的操作；第28〜51 行再次通过调用子进程的方法，创建了查看txtlink子目录情况的命令行，并在第35行打印 输出了 txtlink子目录中的文件内容，本例由于第25行删除硬链接文件的操作成功执行了， 因此打印输出txtlink子目录的内容为空（如图8.10所示）。

Run

**Run tftO8A.Jinifiync.js S**

A , /usr/local/bin/node ch08.+s\_linkSync.]s

fs UnkSyncO

**■ 8**

四 H fs.linkSyncO Done.

£〔

昏 cat txt/UnkSync.txt stdout：

X fs.linkSyncO

cat txtlink/linkSync.txt stdout：

fs. UnkSyncO

fs.uniinkSync() Done

Is -th /txtlink stdout：

stdout: -t i 0 child process closed with code 0

Process finished with exit code 0

I

图8.10创建与删除文件硬链接的方法（同步版本）

从图8.10可以看到，在第10行打印输出的原始文件txt/linkSync.txt的内容（具体内容 为“fs.linkSync。"*）*与第23行打印输出的硬链接文件txtlink/linkSync.txt的内容（具体内 容为“fs.linkSync。"）是完全一致的，说明fs.linkSyncQ方法执行的创建硬链接文件的操作 成功完成了。

另外，第25行fs.unlinkSync（）方法的成功执行，第35行打印输出的内容为空（具体内 容为“stdout：总用量0”），说明fs.unlinkSync（）方法执行的删除硬链接文件的操作也成功 完成了。

说明：结合本节例程的演示结果，我们知道使用白.link。与fs.linkSync（）方法创建的文 件硬链接实际上是创建文件新的副本，而不是文件的符号链接“关于创建文件的符号链接 的方法，我们在下一节介绍'

8.9创建文件符号链接

符号链接，也称为软链接，它是一类特殊的文件，这个文件包含了另一个文件的路径 名（绝对路径或相对路径）。路径可以是任意文件或目录，同硬链接不同的是，它可以链 接不同文件系统的文件。

文件系统模块提供了 fs.symlink。和fs.symlinkSync（）这两个方法来完成创建文件符号链 接的方法的操作。

我们介绍同步方法的例程（代码文件名称：ch08.fs symlinkSync.js）,其主要代码如下:

01 /\*\*

02 \* chO8.fs\_symlinkSync.js

console.info(" fs symlinkSync() --・一

console.info();

〃引入 child\_process 模块

var exec = require('child\_process').exec;

var child\_pre = exec(\*cat txt/symlinkSync.txf, function (error, stdout, stderr) ( console.info('cat txt/symlinkSync.txt stdout:'); console.log(stdout); console.log(stderr);

〃打印输出stdout 〃打印输出stderr

})；

console.info();

var srcpath = "txt/symlinkSync.txt";

var dstpath = "txtlink/symlinkSync.txt";

〃引入文件系统模块

〃链接文件（同步方式）

varfs = require('fs');

fs.symlinkSync(srcpath, dstpath); console.log('fs.symlinkSync() Done.1);

console.info();

var child\_suf = execfcat txtlink/symlinkSync.txt',

function (error, stdout, stderr) (

console.info(\*cat txtlink/symlinkSync.txt stdout: \*); console.log(stdout); 〃打印输出 stdout

console.log(stderr); 〃打印输出 stderr

var spawn = require('child\_process').spawn; 〃弓| 入 child\_process 模块 var ls\_txt = spawnCIs\*, 'txtlink]); 〃定义命令行'Is -Ih Ztxtlink\* /\*\*

\*捕获控制台输出对象stdout,输出捕获数据

\*/ >

ls\_txt.stdout.on('data', function (data) ( console.info('ls -Ih /txtlink stdout:'); console.logfstdout:' + data); console.info();

})；

*Hrlt*

\*绑定系统error事件

7

ls\_txt.on(,error', function (code) (

console.log('chlld process error with code' ■ code); console.info();

})；

/\*\*

\*绑定系统close事件

\*/

ls\_txt.on('close', function (code) (

console.log('child process closed with code \* + code);

console. info();

47

1. })；
2. })；
3. console.info();

**【代码分析】**

为了测试创建文件符号链接的操作，我们预先在代码文件目录中新建两个子目录，一 个名称为txt,用于存放原始文件，另一个名称为txtlink,用于存放链接文件生成的文件。 本例程需要先在txt子目录下创建一个名称为symlinkSync.txt的文件，然后通过 fs.symlinkSync()方法进行创建文件符号链接的操作。

第18行打印输出了 fs.symlinkSync()方法操作成功的信息(Ts.symlinkSyncO Done.\*)； 第20〜24行通过调用子进程的方法，查看了新创建的硬链接文件txtlink/symlinkSync.txt 的文件内容;第25〜48行通过调用子进程的方法，创建了查看txtlink子目录情况的命令行， 并在第32行打印输出了 txtlink子目录中的文件内容。

图8.11是本例程在Ubuntu环境下的输出结果。从中可以看到，在第10行打印输出的 原始文件txt/symlinkSync.txt的内容为"fs.symlinkSync。”，而第23行打印输出符号链接 文件txtlink/symlinkSync.txt的内容时却提示“没有那个文件或目录”，这是什么原因呢？

第32行打印输出的提示信息告诉我们txtlink子目录下存在一个名称为symlinkSync.txt 的文件，但是该文件却是指向txt/symlinkSync.txt文件的符号链接，这说明fs.symlinkSync() 方法执行的文件符号链接的操作成功完成了，同时我们也知道了文件符号链接并不是文件 副本，它仅仅是一个快捷方式。

Run

> \* /usr/local/bin/node chO8.fs\_syalinkSync.js

T fs UnkSyncO

**■ 4**

留 3 fs.symlinkSync0 Done.

**X a**

爵 cat txt/symUnkSync.txt stdout：

X \* fs.syilinkSync()

**7 ®**

cat txtlink/syalinkSync.txt stdout：

cat： txtlink/syalmkSync.txt: &有衅个丈件或3 it

Is -th /txtlink stdout：

stdout：总用童0

Irvxrvxrwx 1 king king 19 44 23 22:57 synlinkSync.txt -> txt/synlinkSync.txt

, child process closed with code 0

Process finished with exit code 0

图8.11创建文件符号链接的方法(同步版本)

说明：经过本节与上一节例程的演示结果，我们知道使用fc.link()与fs.linkSync()方法 创建的文件硬链接实际上是创建了文件新的副本，而使用fc.symlink()与fs.symlinkSync()方 法创建的文件符号链接实际上是快捷方式。

8.10读取文件符号链接

文件系统模块提供了 fs.readlink()和fs.readlinkSync()这两个方法来完成读取文件符号链 接的方法的操作。

我们介绍同步方法的例程(代码文件名称:chO8.fs\_readlinkSync.js),其上要代码如下：

01 /\*\*

02 \* chO8.fs\_readlinkSync.js

03 \*/

04 console.info(n fs readlinkSync() ”);

05 console.info();

06 var srcpath = "txt/symlinkSync.txtM;

07 var linkpath = "txtlink/symlinkSync.txt”；

08 var fs = require('fs');

//引入文件系统模块

//读取链接文件(同步方式)

//读取链接文件(同步方式)

09 var linkstring = fs.readlinkSync(linkpath);

1. console.info('read linkpath:' + linkstring);
2. console.log('fs.readlinkSync() Done.');
3. var srcpath = fs.readlinkSync(srcpath);
4. console.info('read srcpath:' + srcpath);
5. console.log(\*fs.readlinkSync() Done.');
6. console.info();

**【代码分析】**

本例程为了测试读取文件符号链接的操作，借用了 8.9节第一个例程中的两个文件： 第1个文件是txt子目录下创建的一个名称为symlinkSync.txt的文件；第2个文件是代码文 件(名称为ch08.fs\_symlinkSync.js)中生成的符号链接文件(symlinkSync.txt),该文件保 存在名称为txtlink的子目录中。

第09行调用fs.readlinkSync()方法(同步方式)

Run

» 全 /usr/local/bin/node ch08.fs\_readlmkSync. js fs readlinkSyncO

■ ♦ read Unkpath： txt/synlinkSync. txt

囹 目 fs.readlinksync() Done.

X 区 fs.]S：7O6

return binding readlink(pathModule.\_nakeLong(path)):

X 4

7 ® Error： EINVAL, invalid argument 'txt/syilinkSync txt'

at Object.fs.readlinkSync (fs.]S：706：18)

at Object.<anony»ous> *l/hge/king/vebstoe. ]s/Dn>?ects/NodeJ5/* at Module.\_co«pile (nodule.js 456:26)

at Object.Module.\_extensions. js (nodule.js；474:1。)

at Module.load (nodule.]s：356:32)

at Function.Module.\_load (nodule.js：312:12)

at Function.Module runMam («odule.]S：497 10)

at startup (node.js：119：16)

at node.js：906：3

Process finished with exit code 8

图8.12读取文件符号链接的方法(同步版本)

执行读取文件符号链接的操作：第10 行通过变量linkString打印输出了*文* 件符号链接(txtlink/symlinkSync.txt) 的具体值；第12〜13行再次通过调 用fs.readlinkSync()方法，打印输出 了文件 txt/symlinkSync.txt 的符号链 接值。

图8.12是本例程在Ubuntu环境 下的输出结果。

从图8.12可以看到，第10行打 印输出了文件符号链接txtlink/ symlinkSync.txt 的具体值为 txt/symlinkSync.txt , 说明 txtlink/ symlinkSync.txt是一个文件符号链

接，链接的对象就是txt/symlinkSync.txt, fs.readlinkSync()方法执行的读取文件符号链接的 操作成功了。第12行通过fs.readlinkSync()方法读取txt/symlinkSync.txt的操作却出现了错 误，这说明txt/symlinkSync.txt并不是一个有效的文件符号链接。

说明：通过本节例程的演示结果，我们知道文件符号链接实际上就是文件快捷方式， 因此使用fs.readlink()与fs.readlinkSync()方法读取文件符号链接时会成功操作，而读取正常 文件时会发生异常错误。

8.11 相对路径转绝对路径

相对路径和绝对路径是我们操作系统中的文件时必须要面对的两个概念，本节最后会 有对这两个概念的详细描述。路径就是告诉我们，去哪里找到这个文件，就像问你住在哪 里，你报出的是“XX路XX号楼”这种地址是相似的。

文件系统模块提供了 fs.realpath()和fs.realpathSync()这两个方法来完成相对路径转绝对 路径的操作。

我们介绍同步方法的例程(代码文件名称：chO8.fs\_realpathSync.js),其主要代码如下：

01 /\*\*

02 \* chO8.fs\_realpathSync.js

03 \*/

04 console.info(M fs realpathSync() ”)；

05 console.info();

06 varfs = require('fs'); //引入文件系统模块

07 var cache = f/king^Vhome/king'}; //定义 cache

08 var relpath = 'txtZrealpathSync.txt\*; //定义相对路径

09 var resolvedPath = fs.realpathSync(relpath, cache); 〃相对路径转绝对路径

1. console.infoftxt/realpathSync.txt realpath is :');
2. console.info(resolvedPath); 〃打印输出绝对路径
3. console.info();

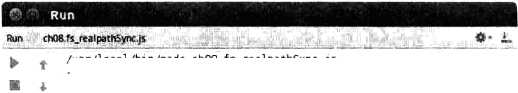
**【代码分析】**

本例程为了测试相对路径转绝对路径的操作，我们预先在代码文件目录中新建一个名 称为txt的子目录，在txt子目录下创建一个名称为realpathSync.txt的文件，然后通过 fs.realpathSync()方法进行相对路径转绝对路径的操作。

第 07 行定义了 fs.realpathSync()方法的 cache 参数(本例为{Vking^/home/king'})；第 08行定义了测试文件的相对路径(本例为txt/realpathSync.txt)；第09行调用fs.realpathSync() 方法(同步方式)执行相对路径转绝对路径的操作；第11行通过变量resolvedPath打印输 出了 txt/realpathSync.txt转换后得到的绝对路径(具体值如图8.13所示)。

从图8.13可以看到，第11行打印输出的绝对路径为：

/home/king/webstorm.js/projects/NodeJS/chapter08/txt/realpathSync.txt



二

/usr/iocai/bin/node ch08.fs\_reaipathSync.]s fs realpathSyncO "

txt/realpathSync txt realpath is

耳 /honeAing/webston# js/pro)ects/NodeJS/chapterO8/txt/realpdthSync.txt

\*4

碧 Process finished with exit code 0

图8.13相对路径转绝对路径的方法(同步版本)

说明fs.realpathSync()方法执行的相对路径转绝对路径的操作成功完成了。

8.12创建与删除目录

为了对纷繁复杂的文件分类，我们规划了不同的目录，如有专门负责用户配置的目录、 有专门保存系统文件的目录。这跟现实生活中不同的部门一样，有交通部、财政部等。

文件系统模块提供了 fs.mkdir()^ &rmdir()、fs.mkdirSync。、fs.rmdirSync()这四个方法 来完成创建与删除目录的操作。

我们介绍同步方法的例程(代码文件名称：chO8.fs\_mkdirSync.js),其主要代码如下:

|  |  |  |
| --- | --- | --- |
| 01 | *1\*\** |  |
| 02 | \* chO8.fs\_mkdirSync.js |  |
| 03 | \*/ |  |
| 04 | console.infof fs mkdirSync() "); |  |
| 05 | console.info(); |  |
| 06 | var fs = requireffs'); | 〃引入文件系统模块 |
| 07 | var mkdirsync ='dir/rnkdirsync1; | 〃定义目录 |
| 08 | fs.mkdirSync(mkdirsync, 0777); | 〃创建目录(同步方式) |
| 09 | console.infoCfs.mkdirSyncO done!1); |  |
| 10 | console.info(); |  |
| 11 | var spawn = require('child\_process').spawn; | 〃引入 child\_process 模块 |
| 12 | var ls\_di jmkdir = spawn('ls','dir1]); | 〃定义命令行’ls-lh/dir' |
| 13 |  |  |
| 14 | \*捕获控制台输出对象stdout,输出捕获数据 |  |
| 15 | \*/ |  |
| 16 | ls\_dir\_mkdir.stdout.onCdata', function (data) { |  |
| 17 | console.info('ls -Ih /dir stdout:'); |  |
| 18 | console.logfstdout:' + data); |  |
| 19 | console.info(); |  |
| 20 | fs.rmdirSync(mkdirsync); |  |
| 21 | console.info('fs.rmdirSync() done!'); |  |
| 22 | var ls\_dir\_rmdir = spawn('ls', 'dir'])； | 〃定义命令行'ls-lh/dir' |
| 23 | y\*\* |  |
| 24 | \*捕获控制台输出对象stdout,输出捕获数据 | |

1. \*/
2. ls\_dir\_rmdir.stdouton(,data,, function (data) (
3. console, infofls -Ih /dir stdout:');
4. console.log('stdout:' + data);
5. console.info();
6. ))；
7. /\*\*
8. \*绑定系统error事件
9. 7
10. ls\_dir\_rmdir.on(,error\*, function (code) (
11. console.logfchild process error with code ' + code);
12. console.info();
13. })；
14. /\*\*
15. \*绑定系统close事件
16. \*/
17. ls\_dir\_rmdir.on(,close,, function (code) (
18. console.logfchild process closed with code ' + code);
19. console.info();
20. })；
21. })；
22. /\*\*
23. \*绑定系统error事件
24. \*/
25. ls\_dir\_mkdir.on('error,, function (code) (
26. console.Iog('child process error with code ' + code);
27. console.info();
28. ))；
29. /\*\*
30. \*绑定系统close事件
31. \*/
32. ls\_dir\_mkdir.on('close', function (code) (
33. console.logfchild process closed with code' + code);
34. console.info();
35. })；
36. console.info();

**【代码分析】**

为了测试创建与删除目录的操作，我们预先在代码文件目录中新建一个名称为dir的子 目录，用于存放创建的目录。

第07行定义了打算新建目录的路径名称(本例为dir/mkdirsync)；第08行调用 fs.mkdirSync()方法(同步方式)执行创建目录的操作；第11〜12行、第16〜18行与第46〜 59行通过调用子进程的方法，创建了查看dir子目录内容的命令行，并在第18行打印输出 了 dir子目录中的文件内容；第20行调用fs.rmdirSync()方法(同步方式)执行删除文件目录的操作；第22〜44行再次通过调用子进程的方法，创建了查看dir子目录内容的命令行, 并在第27行打印输出了 dir子目录中的文件内容(如图8.14所示)。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  | |  |  |  |  |

\* /usr/local/bin/node ch08.fs\_«kdirSync.]s T fs ■kdir^ncO

fs.tkdirSync0 done!

® Is -Ih /dir stdout:

导 stdout： 4.OK

drwxrvxr-x 2 king king 4.0K 4月 27 1Q：12 nkdirsync fs. mdirSyncO done!

child process closed with code 0

Is -Ih /dir stdout： stdout：色雨董0 child process closed with code 0

Process finished with exit code 0

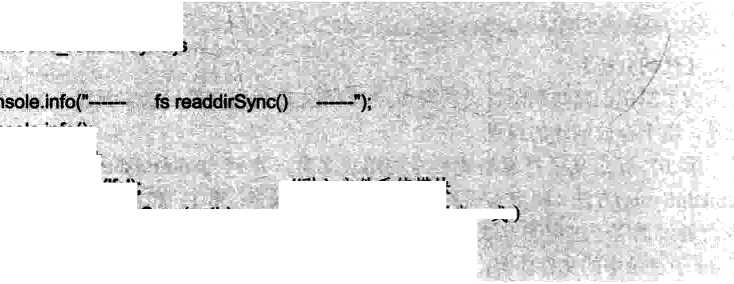
图8.14创建与删除文件目录的方法(同步版本)

从图8.14可以看到，在第08行创建目录的方法执行完毕后，第18行打印输出目录dir 的内容包含该新建的目录(目录名称为mkdirsync),说明fs.mkdirSync()方法执行的创建 目录的操作成功完成了。第20行删除目录的方法执行完毕后，第27行打印输出目录dir 的内容为空(具体内容为“stdout：总用量0” ),说明fs.rmdirSync()方法执行的删除目录 的操作也成功完成了。

说明：fs.mkdirSync()方法会有一个返回值，感兴趣的读者可以试着获取该值并打印输 出，一般系统会输出一个undefined关键字。

8.13读取文件目录

文件系统模块提供了 fs.readdir()和fs.readdirSync()这两个方法来完成读取文件目录的 方法的操作。

我们介绍同步方法的例程(代码文件名称：chO8.fs\_readdirSync.js)*，*其主要代码如下:

console.infoQ\* var path = "txt";

/\*\*

01

02

03

04

05

06

07

08

09

10

\* chO8.fs\_readdirSync.js

V

var fe = requlreffe'); //SI入文件系统模块

var files - fe.FeaddirSync(path); //读取文件目录(同步方式)

console.infofread directory txt:');

console.info(files); 〃打印输出文件目录

1. console.log(\*fs.readclirSync() Done.');
2. console.info();

**【代码分析】**

本例程借用了本书代码中公用的txt子目录，读取该子目录下的文件信息。第06行定 义了子目录的路径（本例为txt）；第08行调用fs.readdirSync（）方法（同步方式）执行读取 文件目录的操作；第10行通过变量files打印输出了目录txt中以数组格式保存的文件信息 （如图8.15所示）。

从图中可以看到，第10行通过变量files打印输出了 txt目录中以数组格式保存的文件

信息（读者会注意到该文件列表中的文件都是前面各个例程用到的），这就说明

fs.readdirSync（）方法执行的读取文件目录的操作成功完成了。

/usr/local/bin/node ch08.fs\_readdirSync.js fs readdirSync()

read directory txt: ('chaod.txt\*, ,chiodSync.txt', 'chovn.txf, 昏 'chovnSync.txt\*,

\* fchiod.txt\*,

'fchiodSync.txt',

'fchown.txt', •fchovnSync.txt', \*fstat.txt', ,fstatsync.txt \ ■ftruncate.txt', 'ftruncateSync.txt\*, 'helloworld-asyn.txt1, •heUovorld-syn.txt\*, 'link,txt',

,linkSync.txt', 'note\*,

'open.txt', 'openSync.txt\*, 'realpath.txt\*, 'realpathSync.txt', •stat.txt'. ,statSync.txt\*. ■syilink.txt\*4 ,sytlinkSync.txf, ,truncate.txt', ,truncateSync.txt' 1 fs.readdir^ncO Done.

Process finished with exit code 0

图8.15读取文件目录的方法（同步版本）

说明：files参数是一个数组格式的数据，因此可以对其进行排序等相关操作。

8.14读文件的应用

本节我们向读者介绍读文件的应用，在该应用中将会用到判断文件是否存在并读文件 的方法。文件系统模块提供了 fs.readFile（）和G.readFileSync（）这两个方法来完成读文件的操 作。至于判断文件是否存在，则有fk.exists**。和**fk.existsSync。这两个方法来实现。

我们介绍同步方法的应用例程（代码文件名称：chO8.fs\_readFileSync.js）,其主要代码 如下：

01 /\*\*

02 \* chO8.fs\_readFileSync.js

03 \*/

04 console.info(M一》— fs readFileSync()

05 console.info(); ；

06 var file\_\_path = “txt/readFileSync.txt”；

07 var fs = requireffs1); //引入文件系统模块

08 if(fs.existsSync(file\_path)) (

09 var file\_contents = fs.readFileSync(file\_path, 'utf-81); //读文件(同步方式)

1. console.infofread txt/readFileSync.txt contents:');
2. console.info(file\_\_contents); 〃打印输出文件内容
3. console.info();
4. } else (
5. console.log(file\_j)ath + 'is not exists/);
6. console.info();
7. )
8. console.log(\*fs.readfileSync() Done.');
9. console.info();

**【代码分析】**

本例程预先在代码文件目录中新建一个名称为txt的子目录，在txt子目录下创建--个 名称为readFileSync.txt的文件，并写入方法字符串fs.readFileSync(),然后通过fs.existsSync()

方法与fs.readFileSync()方法进行读文件的操作。 第06行定义了目标文件的路径(本例为 txt/readFileSync.txt)；第 08 行调用 fs.existsSync() 方法(同步方式)执行判断目标文件是否存在 的操作；第09行调用fs.readFileSync()方法(同

chOS.^reKKIeSync.jS *L*

参 + /usr/local/bin/node chO8.fs\_readFileSync. ]s

I T fs readFileSyncO

O & p read txt/readFileSync.txt contents；

•遇’灯 fs.readfileSync0



步方式)执行读文件的操作；第11行通过变量 file\_contents打印输出了文件内容(如图8.15 所示)。

从图8.16可以看到，第11行通过变量

file contents 打印输出的 txt/readFileSync.txt 文

就 5 fs.readfileSync0 Done,

? 曲 Process finished with exit code 0

图8.16读文件的方法(同步版本)

件中的内容与预先定义的内容完全一致，说明fs.readFileSync()方法执行的读文件的操作成 功完成了。

说明：fs.readFile()方法与fe.readFileSync()方法是根据绝对路径或相对路径来执行读文 件操作的。但在某些情况下，设计人员需要根据文件描述符来执行读文件的操作，下一节 我们向读者介绍fs.read()方法与fs.readSync()方法，这两个方法可以根据文件描述符来执行 读文件的操作。

8.15功能增强的读文件应用

本节我们向读者介绍一个功能增强的读文件应用，该应用将会根据偏移量执行读文件 的操作。文件系统模块提供了 fs.read()和fs.readSync()这两个方法来完成功能增强的读文件 操作。

我们介绍同步方法的应用例程(代码文件名称：chO8.fs\_readSync.js),其主要代码如下:

01 /\*\*

02 \* chO8.fs\_readSync.js

03 \*/

04 console.info(" fs readSync() ");

05 console.info();

06 var readpathSync = "txt/readSync.txt";

07 var fs = require('fs');

08 var contents = fs.readFileSync(readpathSync, 'utf-8');

09 console.infofread file txt/readSync.txt:

1. console.info(contents);
2. console.info('fs.readFileSync() done.');
3. console.info();
4. fs.open(readpathSync, Y, function (err, fd) (
5. if(err) (
6. console.error(err);
7. console.info();
8. return;
9. } else {
10. var buf^a = new Buffer(8);
11. var readbyte\_a = fs.readSync(fd, bu[\_a, 0, 8, null);
12. console.info(\* 读取的字节数：1 + readbyte\_a);
13. console.info('读取的内容:')；
14. console.info(but\_a);
15. console.infoCfs.readSyncO done.1);
16. console.info();
17. var buf\_b = new Buffer(4);
18. var readbyte\_b = fs.readSync(fd, buf\_b, 0, 4, 4);
19. console.infof 读取的字节数：'+ readbyte\_b);
20. console.infof读取的内容：');
21. console.info(buf\_b);
22. console.info('fs.readSync() done.');
23. console.info();
24. var buf\_c = new Buffer(6);
25. var readbyte\_c = fs.readSync(fd, buf\_c, 2, 4, 2);
26. console.infoC读取的字节数：'+ readbyte\_c);
27. console.infoC读取的内容:');
28. console.info(buf^c);
29. console.infoffs.readSyncO done.');
30. console.info();
31. )
32. })；

**【代码分析】**

本例程预先在代码文件目录中新建一个名称为txt的子目录，在txt子目录下创建一个 名称为readSync.txt的文件，并写入方法字符串fs.readSync(),然后通过fs.readSync()方法 进行读文件的操作。

第06行定义了目标文件的路径；第08—12行调用fs.readFileSync()方法执行读文件的 操作；第13行调用fs.open()方法执行打开文件的操作；第19~39行调用fs.readSync。方法 (同步方式)执行了三种方式的读文件操作。

图8.17是本例程在Ubuntu环境下的输出结果。从中可以看到，读取到的字节数大小 与fs.readSync()方法定义的完全一致，写入到数据缓冲区的二进制编码与文件中的内容也完 全符合(读者可自行参考ASCII编码表进行对比)，说明fs.readSyncQ方法执行的读文件 操作成功完成了。



Process finished with exit code 0

read file txt/readSync.txt fs.readSync()

fs.readFileSyncO done.

读散的孝节数：4 浅取的内拿； 〈Buffer 28 7b 2e 72 65 61> fs.readSync() done.

注尊.的宇节薮：8 浚取约为容：

〈Buffer 66 73 2e 72 65 61 64 53a fa.readSync() done.

淀取的字节敷：4 渎取的内容：

<Buffer 65 61 64 53> fs.readSync() done.

/usr/local/bin/node chQ8.fs\_readSync.]s fs readSync()

图8.17功能增强的读文件的方法(同步版本)

说明：ASCII ( American Standard Code fbr Information Interchange,美国标准信息交换 代码)是基于拉丁字母的一套电脑编码系统，主要用于显示现代英语和其他西欧语言“它 是现今最通用的单字节编码系统，等同于国际标准ISO/IEC 646。

8.16写文件的应用

所谓写文件就是将数据写入到空白的文件之中。文件系统模块提供了 fs.writeFileO和 fs.writeFileSync()这两个方法来完成写文件的操作。

我们介绍同步方法的应用例程(代码文件名称：chO8.fs\_writeFileSync.js),其主要代 码如下：

nd /\*\* x 。； ：泌？¥，傍.嫌物务惑理衫'燹鼠

02 \* chO8.fs\_writeFileSync.js

03 \*/

04 console.info(" fs writeFileSyncf) ");

05 console.info();

06 var file\_path = Mtxt/writeFileSync.txtH;

07 varfs = require('fs'); 〃引入文件系统模块

08 if(fs.existsSync(file path)) {

09 var file\_contents\_pre = fs.readFileSync(file\_path, 'utf-8'); //读文件(同步方式)

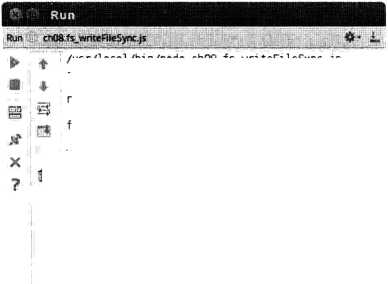
1. console.infofread txt/writeFileSync.txt contents:');
2. console.info(file\_contents\_pre); //打印输出文件内容
3. console.log('fs.readFileSync() Done.');
4. console.info();
5. console.infofwrite to txt/writeFileSync.txt:
6. fs.writeFileSync(file\_path, 'fs.writeFileSync。')； 〃写文件(同步方式)
7. console.log('fs.writeFileSync() Done.');
8. console.info();
9. var file\_contents\_suf = fs.readFileSync(file\_path, ,utf-8,); 〃读文件(同步方式)
10. console.infofread txt/writeFileSync.txt contents:');
11. console.info(file\_contents\_suf); //打印输出文件内容
12. console.logffs.readFileSync() Done.');
13. console.info();
14. } else (
15. console.log(file\_path + 'is not exists.');
16. console.info();
17. )

**【代码分析】**

本例程预先在代码文件目录中新建一个名称为txt的子目录，在txt子目录下创建一个 名称为writeFileSync.txt的空白文件，然后通过fs.writeFileSync()方法进行写文件的操作。

第06行定义了目标文件的路径(本例为txt/writeFileSync.txt)；第08行通过调用 fs.existsSync()方法判断目标文件是否存在；第09〜13行调用fs.readFileSync()方法先执行一 次读文件的操作，查看一下目标文件(txt/writeFileSync.txt)中的内容；其中，第11行通 过变量file\_contents\_pre打印输出了文件内容；第15行调用fs.writeFileSync()方法(同步方 式)执行写文件的操作；第16行打印输出fs.writeFileSyncQ方法执行成功的提示信息；第 18〜22行再次调用fs.readFileSync()方法执行读文件的操作，查看一下目标文件 (txt/writeFileSync.txt)被改写后的内容。

图8.18是本例程在Ubuntu环境下的输出结果。从中可以看到，第11行通过变量 file\_contents\_pre打印输出的txt/writeFileSync.txt文件中的内容为空，这与预先定义的文件 的内容完全一致；而第20行通过变量file contents suf打印输出的文件内容与第15行执行 的写文件方法中定义的内容(fs.writeFileSync。)完全一致；这说明fs.writeFileSync。方法 执行的写文件操作成功完成了。



read txt/vriteFileSync.txt contents；

fs.readFileSyncO Done.

Process finished with exit code 0

Zusr/iocal/bin/node chO8.fs\_vnteFil.eSync.]s fs writeFileSyncO

瞥 ' wnte to txtArriteFileSyne. txt : k fs.writeFileSyncO Done.

;read txtZwriteFileSync,txt contents： :fs.writeFileSyncO

j fs.readFileSync() Dane,

图8.18写文件的方法(同步版本)

说明：fs.writeFile()方法与fs.writeFileSync()方法是根据绝对路径或相对路径执行写文 件的操作的。但在某些情况下需要根据文件描述符来执行读文件的操作，下一节我们向读 者介绍fs.write()方法与fs.writeSync()方法，这两个方法可以根据文件描述符来执行读文件 的操作。

8.17功能增强的写文件应用

本节我们向读者介绍功能增强的写文件应用，与上一节相比，本节的写文件应用可以 实现偏移量写入、追加写入等增强功能。文件系统模块提供了 fs.write()和fs.writeSync()这 两个方法来完成功能增强的写文件操作。

我们介绍同步方法的应用例程(代码文件名称：chO8.fs\_writeSyncBuf.js),其主要代 码如下：

01 /\*\*

02 \* chO8.fs\_writeSyncBufJs

03 \*/

04 console.info(M fs writeSync() ");

05 console.info();

06 var file\_path = Htxt/writeSync.txtM;

07 varfs = requireffs'); //引入文件系统模块；

08 /\*\*

09 \*打开文件(异步方式)

1. \*/
2. fs.open(file\_path, W, function (err, fd) (
3. if(err)(
4. throw err;
5. } else (
6. console.logffs.openO done.');
7. console.info();
8. var file\_contents\_pre = fs.readFileSync(file\_path, 'utf-8'); 〃读文件(同步方式)
9. console.info(\*read txt/writeSync.txt contents:');
10. console.info(file\_contents\_pre); 〃打印输出文件内容
11. console.log('fs.readFileSync() Done.');
12. console.info();
13. console.info('write to txt/writeSync.txt:');
14. var but = new Buffer('fs.writeSync(fd, buffer, offset, length[, position])\n');
15. var len\_buf = fs.writeSync(fd, buf, 0, buf.length, 0); 〃写文件(同步方式)
16. console.log(\*fs.writeSync() Done.');
17. console.info();
18. console.log('写入数据的字节数:1 + len\_buf);
19. console.info();
20. var file\_contents\_\_suf = fs.readFileSync(file\_path, 'utf-8'); //读文件(同罗方式)
21. console.infoCread txt/writeSync.txt contents:');
22. console.info(file\_contents\_suf); //打印输出文件内容
23. console.log('fs.readFileSync() Done.\*);
24. console.info();
25. /\*\*
26. \*关闭文件(异步方式)
27. \*/
28. fs.close(fd, function (err) (
29. if (err) (
30. throw err;
31. } else {
32. console.Iog('fs.close。done.\*);
33. console.info();
34. )
35. })；
36. }
37. ))；

**【代码分析】**

本例程预先在代码文件目录中新建一个名称为txt的子目录，在txt子目录下创建一个 名称为writeSync.txt的空白文件，然后通过fs.writeSync()方法进行写文件的操作。

第17〜20行通过调用fs.readFileSyncQ方法先执行一次读文件的操作，查看一下目标 文件(txt/writeSync.txt)中的内容；第23行定义了一个Buffer类型的变量，并进行了初始 化操作，初始化内容为本例程用到的fs.writeSync()方法的语法形式(“fs.writeSync(fcl, buffer, offset, length[, position])\nn )；第24行调用fs.writeSync()方法(同步方式)执行写文件的 操作；第25行打印输出fs.writeSync()方法执行成功的提示信息；第27行通过变量len\_buf 打印输出了实际写入数据的长度为53；第29〜32行再次调用fs.readFileSync()方法再次执 行读文件的操作，查看一下目标文件(txt/writeSync.txt)被改写后的内容：第37~44行调 用fs.close()方法执行关闭文件的操作。

图8.19是本例程在Ubuntu环境下的输出结果。从中可以看到，第19行通过变量hle\_ contents\_pre打印输出的txt/writeSync.txt文件中的内容为空，这与预先定义的文件的内容 完全一致；而第31行通过变量file\_contents\_suf打印输出的文件内容与第23〜24行执行 的写文件方法中定义的内容完全一致；这说明fs.writeSync()方法执行的写文件操作成功 完成了。

/usr/local/bin/node ch08.fs\_vriteSyncBuf.js fs vriteSyncO

fs.openO done.

read txt/wnteSync.txt contents：

fs, readFileSyncO Done.

write to txt/writeSync.txt : fs.wnteSyncO Done.

read txt/wnteSync.txt contents：

fs.writeSync(fd, buffer, offset, length|, position]) fs.readFileSyncO Done.

fs.closeO done.

Process finished with exit code 0

图8.19第一种写文件的方法(同步版本)

说明：关于重载，简单说就是方法有同样的名称，但是参数列表不相同的情形，这样 的同名不同参数的方法之间，互相称之为重载方法。如果读者对这方面的概念了解不多， 可以去参考一本面向对象方面(C廿、Java)的书籍，

8.18追加写入文件

所谓追加写入文件就是将数据写入到文件原有内容之后。文件系统模块提供了 fs.appendFileO和fs.appendFileSync()这两个方法来完成追加写入文件的操作。其中， fs.appendFileSync()是同步方式的方法，而fs.appendFileO是异步方式的方法。

我们介绍同步方法的应用例程(代码文件名称：chO8.fs\_appendFileSync.js),其主要 代码如下：

01 /\*\*

02 \* chO8.fs\_\_appendFileSync.js

03 7

04 console.info(" fs appendFileSync()

05 console.info();

06 var file\_\_path = "txt/appendFileSync.txt";

07 varfs = require(fs')； //引入文件系统模块

08 if(fs.existsSync(file\_path)) (

09 var file\_contents\_pre = fs.readFileSync(file\_path, 'utf-8'); //读文件(同步方式)

1. console.infofread txtZappendFileSync.txt contents:');
2. console.info(file\_contents\_pre); 〃打印输出文件内容
3. console.log('fs.readFileSync() Done.\*);
4. console.info();
5. console.info(\*write to txt/appendFileSync.txt:');
6. fs.writeFileSync(file\_\_path, \*fs.appendFileSync(filename, data[, options])\n\*); //写文件(同 步方式)
7. console.logCfs.writeFileSyncO Done.');
8. console.info();
9. var file\_contents\_suf = fs.readFileSync(file\_path, 'utf-8'); 〃读文件(同步方式)
10. console.info('read txt/appendFileSync.txt contents:');
11. console.info(file\_contents\_suf); 〃打印输出文件内容
12. console.log('fs.readFileSync() Done.');
13. console.info();
14. fs.appendFileSync(file\_path, 'add filename to appendFileSync.txt\n\*); //追加写入文件(同

步方式)

1. console.logCfs.appendFileSync() Done.1);
2. console.info();
3. var file\_contents\_\_add = fs.readFileSync(file\_path, 'utf-8'); //读文件(同步方式)
4. console.infofread txt/appendFileSync.txt contents:');
5. console.info(file\_contents\_add); 〃打印输出文件内容
6. console.log('fs.readFileSync() Done.');
7. console.info();
8. } else (
9. console.log(filej3ath + 'is not exists?);
10. console.info();
11. }

**【代码分析】**

本例程预先在代码文件目录中新建一个名称为txt的子目录，在txt子目录下创建一个 名称为appendFileSync.txt的空白文件，然后通过fs.appendFileSync()方法进行写文件的操 作。

第14〜17行调用fs.writeFileSync()同步方法执行写文件的操作，将ufs.appendFileSync (filename, data[, options])” 函数语法写入文件；第 18~22 行调用 fs.readFileSync()方法先执 行一次读文件的操作，查看一下目标文件(txt/appendFileSync.txt)被改写后的内容；第23 行调用fs.appendFileSync()方法(同步方式)执行追加写入文件的操作；第24行打印输出 fs.appendFileSync()方法执行成功的提示信息；第26〜30行再次调用fs.readFileSync()方法 执行读文件的操作，查看一下目标文件(txt/appendFileSync.txt)被追加写入后的内容。

图8.20是本例程在Ubuntu环境下的输出结果。从图中可以看到，第11行通过变量 file\_contents\_pre打印输出的txt/appendFileSync.txt文件中的内容为空，这与预先定义的文 件的内容完全一致；而第20行通过变量file\_contents\_suf打印输出的文件内容与第15行执

行的写文件方法中定义的内容("fs.appendFileSync(filename, data[, options])")完全一致， 说明fs.writeFileSync()方法执行的写文件操作成功完成了 ；第23行调用fs.appendFileSync() 方法执行追加写入文件的操作后，在第28行通过变量file contents add进行了打印输出， 说明fs.appendFileSync()方法执行的追加写入文件的操作也成功完成了。

Run

/usr/local/bin/node chQS.fs\_appendFileSync.]s fs appendFileSync(T read txt/appendFileSync.txt contents：

fs.readFileSyncO Done.

wnte to txt/appendFileSync.txt ； fs.writeFileSync() Done.

read txt/appendFileSync.txt contents:

fs appendFileSync(filename. data I, options))

fs.readFileSyncO Done.

fs.appendFileSync() Done.

read txt/appendFileSync.txt contents：

fs.appendFileSync(filename. data[. options]) add filename to appendFileSync.txt

fs readFileSync() Done.

Process finished with exit code 0

图8.20追加写入文件的方法(同步版本)

说明：执行fs.appendFile()方法与fs.appendFileSync()方法进行追加写入操作时，如果 目标文件不存在，则会直接创建该文件，读者可以利用本节的两个例程自行测试一下。

8.19监控文件的应用

所谓监控文件就是对指定的文件或路径添加监听事件，当文件或路径发生改变时，触 发该事件。文件系统模块提供了 fs.watch()和fs.watchFile()这两个方法来完成监控文件的操 作。这里要特别指出的是，根据Node.js框架官方文档的说明，fs.watch()和fs.watchFile() 这两个方法针对不同的系统平台使用起来不是很稳定，设计人员需要谨慎使用。

下面我们介绍本节的例程(代码文件名称：chO8.fs\_watchFile.js),其主要代码如下：

01 /\*\*

02 \* chO8.fs\_watchFile.js

03 \*/

04 console.info(w fs watchFile() ");

05 console.info();

06 var file\_path = "txt/watchFile.txt";

07 var fs = requireffs1); //引入文件系统模块

08 〃\*

09 \*监控文件方法

1. \*
2. \* fs.watchFile(filename, [options], listener);
3. \*/
4. fs.watchFile(file\_path, function (curr, prev) (
5. console.log('the current mtime is:' + curr.mtime);
6. console.logfthe previous mtime was:' + prev.mtime);
7. /\*\*
8. \*监控文件方法
9. \*
10. \* fs.watch(filename, [options], [listener]);
11. 7
12. fs.watch(file\_path, function (event, filename) (
13. console.logfthe event is:' + event);
14. console.log('the filename is:' + filename);
15. })；
16. ))；

**【代码分析】**

本例程预先在代码文件目录中新建一个名称为txt的子目录，在txt子目录下创建一个 名称为watchFile.txt的空白文件，然后通过fs.watchFile()方法与fs.watchQ方法进行监控文 件的操作。

第13行调用fs.watchFile()方法执行了监控文件的操作；第14〜15行通过参数listener 打印输出了目标文件(txt/watchFile.txt)的实时状态；第21行调用fs.watch()方法再次执行 了监控文件的操作；第22~23行通过参数listener打印输出了目标文件txt/watchFile.txt的 状态信息，图8.21与图8.22是本例程在Ubuntu环境下的输出结果。



爵 毒

/usr/local/bin/node chG8.fs\_watcHFile.js fs watchFileO



图8.21监控文件的方法应用(一)

M

/usr/local/bin/node chG8.fs\_watchFile.js

fs watchFileO

the current Mtime is： Thu Apr 30 2015 16:34:19 GMT+0800 (CST) the previous mtime was： Thu Apr 30 M15 15：54：32 GMT+O8G0 (CST) the event is： rename

the filenasie is： watchFile.txt

the event is： change

the ■filename is： watchFile.txt

the event is： rename

the filenaae is； watchFile.txt

the event is； rename

the filename is： watchFile.txt

the current mtime is： Thu Apr 30 2G15 16:34:31 GMT+Q8G0 (CST) the previous ntine was： Thu Apr 30 2015 16：34:19 GMT+G800 (CST)

Process finished with exit code 130

图8.22监控文件的方法应用(二)

从图8.21可以看到，程序运行后其一直处于监控状态(程序未被终止)；此时读者可 以人工操作一下txt/watchFile.txt文件，改变一下文件的内容；经过几秒钟的刷新，监控信 息就全部打印出来了(如图8.22所示)。

说明：执行fe.watch()方法与fs.watchFile。方法不是完全跨平台的，且在某些情况下是 不可用的。这是因为此功能需要依赖于操作系统底层提供的方法来监视文件系统的变化“ 如果操作系统底层函数出于某些原因不可用，那么fs.watch()方法与fs.watchFileQ方法也就 无法工作、，例如，监视网络文件系统(如NFS、SMB等)的文件或者目录，就时常不能稳 定地工作，有时甚至完全不起作用。彳艮多时候，在调用fs.watch()方法与fs.watchFile。方法 后会没有反应，有可能是因为比较慢造成的，可能等待一段时间或者重新刷新一下才会打 印出反馈信息，可以说Node.js框架的文件系统提供的监控文件的方法是比较不可靠的

第9章路径处理应用

对于编程开发而言，路径处理是必不可少的重要环节，一款优秀的编程语言都会包含 功能完善的路径处理方法。同样，Node.js框架为设计人员提供了文件路径处理(Path)模 块、url路径处理(url)模块以及字符串解析(Query String)模块，这些模块提供了一系 列与路径解析处理相关的方法，这些方法对于处理常规的需求来说是足够的了。

本章主要包括以下内容。

. Path模块方法与应用。

* url模块方法与应用。
* Query String模块方法与应用。

9.1规范化字符串路径

设计人员在开发程序时，可能会在定义好的路径中不小心混入多余的字符，看似很细 小的问题却会带来很严重的后果。不过不用担心，Node.js框架提供了 path.normalize()方法 来处理这个问题。

在本节这个简单的测试例程中，我们使用path.normalize。方法规范化输出各种非常规 的路径，本例ch09.path normalize.js主要代码如下:

01 /\*\*

02 \* ch09.path\_normalize.js

03 7

04 console.info(" path normalize() w);

05 console.info();

06 var path = require(\*path'); 〃引入路径处理模块

07 /\*\*

08 \* 测试 path.normalize()方法

09 • @type (string}

1. \*, ，
2. var path\_a = 7home//king";
3. console.info(,path.normalize(B/hoine//king") is : \* + path.normalize(path\_\_a));
4. console.info();
5. var path\_b = Vhome//king/";
6. console.info('path.normalize("/home//king/") is :1 + path.normalize(path\_b));
7. console.info();
8. var path\_\_c = "/home/Z/king//\*\*;
9. console.infofpath.normalizeCVhome/Z/king/r) is :' + path.normalize(path\_c));
10. console.info();
11. var path\_d = "/home/king/.”；
12. console.infoCpath.normalize(7home/king/.w) is :' + path.normalize(path\_d));
13. console.info();
14. var path\_\_e = "/home/king//.'1;
15. console.infoCpath.normalizeCVhome/king//.") is :' + path.normalize(path\_e));
16. console.info();
17. var path\_f = 7home/king//.abc";
18. console.info(\*path.normalize(7home/king//.abc") is : \* + path.normalize(path\_f));
19. console.info();
20. var path\_g = "/home/king//..";
21. console.infofpath.normalizef^ome/king//..") is : \* + path.noimalize(path\_\_g));
22. console.info();
23. var path\_h = 7home/king//..abcM;
24. console.info('path.normalize(,7home/king//..abcM) is :' + path.normalize(path\_h));
25. console.info();
26. var pathj = M/home/king/../..M;
27. console.info('path.normalize('7home/king/.is :' + path.normalize(pathj));
28. console.info();
29. var pathj = 7home/king//.
30. console.infot'path.normalizeCVhome/king//..//..") is :' + path.normalize(pathj));
31. console.info();

**【代码分析】**

第06行引入路径处理(path)模块，同时赋于变量path；第07~40行调用path.normalize() 方法规范化了多种非常规的字符串路径。

图9.1是本例程在Ubimtu环境下的输出结果。

Run

> 舍 J /usr/local/bin/node chO9 path\_nornalize. ]s

path noraalizeO

path. norBalize(\*/hoBe/Aing") is : /hoieAing path. noraalize (\* /ho«e//king/") is : /hoieAing/ path.nomalize('/ho«e///king//") is : /hoaeAing/ path.norinalizeC/hoieAing/.") is : /honeAing path.noraalizeC/honeAing//.") is : /homeAing path.normalize(\*/hoie/king//.abc\*) is : /hoieAing/. abc path,nor»alizeC/ho»e/king//. •) is : /ho«e path.no nalize('/ho»e/king//..abc\*) is : /hoae/king/. abc path.normalize("/honie/king/. ./..\*) is : / path.nor«alizeC/ho«e/king//..//..") is : /

Process finished with exit code 0

提示：读者一定注意到了，本例程中字符串路径结尾处含有字符的，在使用 path.normalize()方法规范化输出后，均会跳转到上一级目录，如果连续使用则会连续向上一 级跳转。

9.2合并字符串路径

设计人员在操作字符串路径时，如果需要将多个断开的字符串路径进行合并，那么 Node.js框架提供了 path Join。方法来处理这个问题。

本例ch09.path\_join.js主要代码如下：

01 /\*\*

02 \* ch09.pathjoin.js

03 *\*1*

04 console.infoC'— path join()——

05 console.info();

06 var path = require('path'); 〃引入路径处理模块

07 /\*\*

08 \* 测试 path.join()方法

09 \* @type {string}

1. \*/
2. var path\_a = "home”;
3. var path\_b = "king";
4. var path\_\_c = nwebstorm.jsM;
5. console.info(,path.join(nhomeM,nkingtt,Mwebstorm.jsn) is :' + path.join(path\_a, path\_b, path\_c));
6. console.info();
7. var path\_d = "home";
8. var path\_e = "king";
9. console.info(,path.join(whomew,{},okingw) is :' + path.join(path\_d, 0, path\_e));
10. console.info();

**【代码分析】**

第07〜19行调用path.join()方法进行了合并字符串路径的操作；第11〜13行分别定义 了三个字符串路径；第14行使用path.joinO方法对这三个字符串路径进行了合并的操作； 第16-17行分别定义了两个字符串路径；第18行使用path.join()方法对这两个字符串路径 进行了合并的操作，与第14行不同的是，这里path.joinO方法的第二个参数使用了非字符 串路径({})。

从图9.2的输出结果看，程序抛出了异常。这是因为第二个合并参数使用了非字符串 字符({})，从异常信息也可以得出path.joinO的参数必须使用字符串。

提示：在使用path.joinO方法进行合并字符串路径的操作时，其路径参数必须为字符串。

188 •超实用的Node.js代码段

**产** Run

Zusr/local/bin/node chG9.path\_]oin.js path ]om() 一

path. joinChome". "king". ^webstor®.)sw) is : ho«e/kmg/vebstonii. ]s

path.js 360

throw n^w TypeErrar(\*Arquaents to path join Rust be strings ).

TypeError： Arguments to path,join «ust be strings

at path,js 360:15

at Array filter (native)

at Cbjeci.exports.join (path.]Sl358.36)

at Object ,<anonynious> 1 Tdnr is/njpct*5/Node*re*r*

at Module.^compile (module,js；456：28)

at Object.Module.\_extension9..js (nodule.js 474：10)

at Module.load (module.]s:356:32)

at Function.Module \_load (module )s：312.12)

at Function.Module.runMain (nodule js 497 10)

at startup (node *js.119.16)*

Process finished with exit code 8

图9.2合并字符串路径的方法

9.3解析绝对路径

在己知一个相对路径的时候，如何将其解析为一个绝对路径呢？ Node.js框架提供了 path.resolve ()方法来处理这个问题。

本例ch09.path\_resolve.js主要代码如下：

01 /\*\*

02 \* ch09.path\_resolve.js

03 7

04 console.info(" path resolve() ");

05 console.info();

06 var path = require('path'); 〃引入路径处理模块

07 /\*\*

08 \* 测试 path.resolve()方法

09 \* @type (string}

1. 7
2. console.info("Current working directory: w);
3. console.info(process.cwd());
4. console.info();
5. var path\_a = Hch09.path\_resolve.js";
6. console.info(\*path.resolve("ch09.path\_resolve.js") is :');
7. console.info(path.resolve(path\_a));
8. console.info();
9. var path\_b = "test";
10. var path\_c = "test.txtn;
11. console.info('path.resolve("test","test.txT) is :');
12. console.info(path.resolve(path\_b, path\_c));
13. console.info();

**【代码分析】**

第11~12行调用process.cwd()方法打印输出了当前的工作目录；第14~16行调用path, resolve。方法执行了解析绝对路径的操作；第14行定义了本例程文件名(ch09.path\_ resolve.js)；第16行使用path.resolve()方法进行了解析绝对路径的操作，从图9.3的输出 结果看，返回的字符串路径进行了规范化操作，输出结果为：

“/home/king/webstomi.js/projects/NodeJS/chapter09/ch09.path\_resolve.js”

为了测试本例程，我们在当前工作目录下新建一个子目录test,并在该子目录下新建一 个文本文件test.txt；第18~19行分别定义了这两个字符串路径test和test txt；第21行使用 path.resolve()方法对这两个字符串路径执行了解析绝对路径的操作，输出结果如图9.3所示。

从输出的结果看，第*21*行的path.resolve()方法输出了文本文件test.txt的绝对路径：

“/home/king/webstorm.js/projects/NodeJS/chapter09/test/test.js”

提示：在使用path.resolve()方法执行解析绝对路径的操作后，返回路径均为规范化的 字符串绝对路径。



Current working directory :

/home/kmgArebsto rm. js/proj ects/NodeJS/chapte r09

/usr/local/bin/node ch09.path\_resolve.js path resolve()

path.resolve(Hch09.path\_resolve.jsM) is :

/home/king/webstarm.]s/projects/NodeJS/chapter09/ch09,path\_resolve.js path.resolve("test","test.txt") is :

/home/king/vebstorin. ]s7pro]ects/NodeJS/chapterO9/test/test. txt

Process finished with exit code 0

图9.3解析绝对路径的方法

9.4解析相对路径

上一节我们阐述了解析绝对路径的方法，本节我们在此基础上阐述解析相对路径的方 法。Node.js框架提供了 path.relative()方法来处理这个问题。

本例ch09.path\_relative.js主要代码如下：

01 /\*\*

02 \* ch09.path\_relative.js :;据

03 \*/

04 console.info(H---- - path relative() ——°);

05 console.info();

06 var path = requirefpath'); //引入路径处理模块

07 〃\*

08 \* 测试 path.resolve()方法

09 \* @type (string}

1. \*/
2. console.info("Current working directory :M);
3. console.info(process.cwd());
4. consoleJnfo();
5. var path\_a = wch09.path\_relative.js";
6. console.infoCpath.resolvef^chOG.path^relative.js") is:');
7. var path\_resolve\_a = path .resolve(path\_a);
8. console.info(path\_resolve\_\_a);
9. console.info();
10. var path\_b = "test";
11. var path\_c = "testtxt”；
12. 8nsole.1nfo(，path.resolve(”test”,'testtxr)is: \*);
13. var path\_resolve\_b = path.resolve(path\_b, path\_c);
14. console.info^he relative path is:n + path.relative(path\_resolve\_a, path\_resolve\_b));
15. console.info();

【代码说明】

第14行定义了本例程文件名(ch09.path\_relative.js)；第16行使用path,resolve()方法 进行了解析绝对路径的操作，返回结果保存在变量path\_resolve\_a中；从图9.4的输出结果 看，返回的字符串路径进行了规范化操作，输出结果为：

<7home/king/webstorm.js/projects/NodeJS/chapter09/ch09.path\_relative.js>,

第22行再次使用path.resolve()方法对这两个字符串路径执行了解析绝对路径的操作， 返回结果保存在变量path resolve b中；第23行调用path.relative。方法执行了解析相对路 径的操作。

从图9.4的输出结果看，第23行的path.relative()方法输出了本例程代码文件ch09.path\_ relative.]s相对于文本文件test.txt的起源路径：

"../test/test.txt”

提示：在使用path.relative()方法执行解析相对路径的操作时，两个参数必须为字符串 绝对路径。



Current working directory :

/hoie/king/pro]ects/NodeJS/chapter09

/usr/local/bin/node ch09.path\_relative.js path relative()

path.resolve("chG9.path\_relative.js") is :

/ho«eAing/pro]ects/NodeJS/chapter09/ch09.path\_relative. js path.resolve("test",'test.txt\*) is : The relative path is : ../test/test.txt

Process finished with exit code 0

9.5获取文件夹路径

本节我们介绍获取文件夹路径的方法，具体来讲就是根据已经得到的文件名称，获得该 文件所在文件夹绝对路径的方法。Node.js框架提供了 path.dimame()方法来处理这个问题。

本例ch09.path\_dimame.js主要代码如下：

01 /\*\*

02 \* ch09.path\_dimame.js

03 \*/

04 console.info(" path dimame() ");

05 console.info();

06 var path = requireCpath\*); *II*引入路径处理模块

07 /\*\*

08 \* 测试 path.dimame()方法

09 \* @type (string)

1. 7
2. var path\_a = "test";
3. var path\_b = "test.txt";
4. var path\_resolve = path .resol ve(path\_a, path\_b);
5. console.info(,path.resolve("test",',test.txt") is :');
6. console.info(path\_\_resolve);
7. console.info();
8. console.info(MThe file test.txfs dirname is :");
9. console.info(path.dirname(path\_resolve));
10. console.info();

**【代码分析】**

第13行使用path.resolve()方法进行了解析绝对路径的操作，返回结果保存在变量path\_ resolve中，并在第15行进行了打印输出；从图9.5的输出结果看，返回的字符串路径进行 了规范化操作，输出结果为：

\*7home/king/projects/NodeJS/chapter09/test/test.txt"

第18行调用path.dimame()方法执行了获取文件夹路径的操作。

Run

**ton chO94»tlLclin»me4K** \*、

> 旁 /usr/local/bin/node ch09.path\_dima»e. js

\* path dima«e()

■，

path.resolve("tesf,"test.txt\*) is :

离弓 /ho«e/king/projects/NodeJS/chapterO9/test/test.txt

E

戒 心 The file test.txt's dimane is :

扃i /ho«e/king/projects/NodeJS/chapterO9/test

:t

Process finished with exit code 0

从输出的结果看，第18行的path.dimame()方法输出了本例程文本文件test.txt所在文 件夹的绝对路径：

“/home/king/projects/NodeJS/chapter09/test”

提示：在使用path.dimame()方法执行获取文件夹路径的操作时，*参数*“p”必须为字 符串绝对路径。

1. 获取路径中文件扩展名

我们经常在一些论坛中上传头像(图片)，系统如何判断我们上传的是否是图像呢， 最简单的方法就是判断文件的扩展名是不是.jpg、.png等图像扩展名。Node.js框架提供了 path.extname()方法来完成获取路径中文件扩展名的功能。

本例ch09.path\_extname.js主要代码如下：

01 /\*\*

02 \* ch09.path\_extname.js

03 7

04 console.info(" path extname() ");

05 console. info();

06 var path = require('path'); //引入路径处理模块

07 /\*\*

08 \* 测试 path.extname()方法

09 \* @type (string)

1. \*/
2. var path\_a = "test";
3. var path\_b = "test.txt";
4. var path\_resolve = path.resolve(path\_a, path\_b);
5. console.infofpath.resolve("test",,,test.txtn) is :');
6. console.info(path\_resolve);
7. console.info();
8. console.info('path.extname("test.txt") is :');
9. console.info(path.extname(path\_resolve));
10. console.info();

**【代码分析】**

path.resolve("test\*."test.txt") is :

/hone/king/pro]ects/NodeJS/chapterO9/test/test.txt

lui *5* cho岫th芸恐咿毕

A f /usr/local/bin/node ch09.path\_extna»e.js

path extnaaeO

path.extname("test.txt\*) is ,txt

Process finished with exit code 0

第18行调用path.extname()方法执行了获 取路径中文件扩展名的操作。从图9.6输出的 结果看，第18行的path.extname()方法输出了 本例程文本文件test.txt的文件扩展名o

提示：在使用path.extname()获取文件扩展 名时，参数p必须为字符串绝对路径。

9.7如何获取路径中最后部分

本节我们介绍一个比较特殊的功能，就是如何获取路径中最后部分的方法，具体来讲, 就是根据一个巳知的绝对路径及其文件后缀名，获取绝对路径最后一个“/”字符后面并除 去文件后缀名的部分。Node.js框架提供了 path.basename()方法来完成这个功能。

本例ch09.path basename.js主要代码如下：

01 /\*\*

02 \* ch09.path\_basename.js

°3 •/

04 console.info("—— path basename() -—");

05 console.info();

06 var path = requirefpath'); *II*引入路径处理模块

07 /\*\*

08 \* 测试 path.basename()方法；

09 \* @type {string}

1. \*/
2. var path\_a = "test'\*;
3. var path\_b = "test.txt1\*;
4. console.info('path.resolve(MtestH,'\*test.txt") is : \*);
5. var path\_resolve = path.resolve(path\_a, path\_b);
6. console.info(path\_resolve);
7. console.info();
8. console.info('path.extname(ntest.txtH) is :');
9. var path\_extname = path.extname(path\_resolve);
10. console.info(path\_extname);
11. console.info();
12. /\*\*
13. \* 测试 path.basenameC'test.txt", n.txtn)方法
14. \*/
15. console.infofpath.basename("test.txtMl ".txt") is:
16. var path\_basename = path.basename(path\_resolve, path\_extname);
17. console.info(path\_basename);
18. console.info();
19. /\*\*
20. \* 测试 path-basenameClest-txt", "sttxt'\*)方法
21. 7
22. console.info(,path.basename('\*test.txt,', "st.txt") is : \*);
23. var path\_basename\_\_st = path.basename(path\_resolve, "st.txt”)；
24. console.info(path\_basename\_\_st);
25. console.info();
26. /\*\*
27. \* 测试 path.basename("test.txf, "test.txt")方法
28. 7
29. console.info('path.basename(ntest.txt", "testtxt") is :');
30. var path\_basename\_\_test = path.basename(path\_resolve, "test.txtM);
31. console.infoCvariable path\_basename\_test is :" + path\_basename\_\_test);
32. console.info();

**【代码分析】**

第13行使用path.resolve()方法进行了解析绝对路径的操作，返回结果保存在变量path\_ resolve中，并在第15行进行了打印输出；从图9.7的输出结果看，返回的字符串路径进行 了规范化操作，输出结果为：

u/home/king/projects/NodeJS/chapter09/test/test.txtM

第21〜41行分别三次调用path.basename()方法，执行了三种不同类型的获取路径中最 后部分的方法。

图9.7是本例程在Ubuntu环境下的输出结果。从结果看，第25行调用的path.basename() 方法输出了去除test.txt文本文件的文件扩展名“.txt”后的路径部分，第26行输出的字符 串结果为test；第32行调用的path.basename()方法输出了去除test.txt文本文件的文件后缀 “st.txt”后的路径部分，第33行输出的字符串结果为te；第39行调用的path.basename() 方法输出了去除test.txt文本文件的路径部分，第26行输出的字符串结果为空。



/usr/local/bin/node chQ9.path\_basename.js

path basenameO

path.resolve("test","test,txt") is : /home/king/pro]ects/NodeJS/chapter09/test/test.txt

path.extnaie("test.txt") is :

.txt path.basename(-test.txt", ".txt") is : test

path.basena«e(\*test.txt".

-st.txt") is :

path.basenaieC\*test.txt\*, "test.txt") is : variable path\_basename\_test is :

Process finished with exit code 0

图9.7如何获取路径中最后部分

1. 解析与格式化url路径

本节我们介绍一下解析与格式化url路径的方法，例如我们常常用到的网页链接地址就 属于url路径的一种，可见掌握解析与格式化url路径的方法是非常重要的。在Node.js fll 架中，url模块提供了 url.parse()方法与urLfbrmat()方法来完成这项功能。

本例ch09.url\_parse format.js主要代码如下：

01 /\*\*

02 \* ch09.url\_parse\_format.js

03 \*/

04 console.infof\* url parse() ");

05 console.info();

06 var url = require('urT);//引入路径处理模块

07 /\*\*

08 \* 测试 url.parse()方法

09 \* @type {string}

1. \*/
2. var queryUrl = M<http://localhost:6666/index?name=king&gender=male&level=high>";
3. console.info('typeof queryUrl is : \* + typeof url.parse(queryUrl));
4. console.info();
5. console.info('url.parse("queryUri") is :');
6. var url\_\_parse = url.parse(queryllrl); //调用 url.parse()方法
7. console.info(url\_parse);
8. console.info();
9. /\*\*
10. \* 测试 urLformat()方法
11. \*/
12. console.info('url.format(Murl\_parseM) is :');
13. var url\_format = url.format(url\_parse); *II*调用 url.format。方法
14. console.info(url\_format);
15. console.info();

**【代码分析】**

第06行引入路径处理(url)模块，同时赋于变量url：

为了测试本例程，我们在第11行中定义了一个url字符串路径：

■,<http://localhost:6666/index?name=king&gender=male&level=high>"

第12行使用url.parse()方法执行了解析url字符串路径的操作，并通过“typeof”关键 字进行了打印输出返回值的类型，如图9.8所示；第15行再次调用url.parse()方法执行了解 析url字符串路径的操作，返回结果保存在变量url\_parse中，并在第16行进行了打印输出； 第22行调用了 url.format()方法执行了格式化url路径对象的操作，返回结果保存在变量 url fbrmat 中。

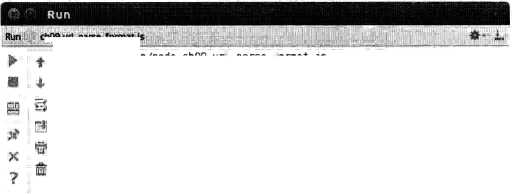
从图9.8输出的结果看，第12行通过typeof关键字并调用url.parse()方法获得的输出 结果为Object,说明urLparse()方法返回值是一个url路径对象；第15〜16行再次调用 url.parse()方法打印输出了 url路径对象的内容，如图9.8所示，该对象是一个JSON格式的 数据，罗列了与url相关的各个属性名称及属性值；第22〜23行调用urLfbnnat()方法输出 了重新格式化后的url路径，如图9.8所示，其具体结果为：

“http:〃localhost:6666/index?name=king&gender=male&level=high”

从结果来看，其值与第11行定义的url字符串路径完全一致，可见url.fbrmatO方法相

当于url.parseQ方法的逆操作。

提示：从本例打印输出的结果来看，url.parse()与url.fdrmat()为一组功能互逆的方法'



typeof queryUrl is : object

/usr/local/bin/node ch09.url\_pars.e\_forniat. js

url parsed

url.parse("queryUrl") is :

( protocol： 'http；'.

slashes: true,

auth： null,

host： 'localhosti6666',

port： '6666',

hostname； 'localhost',

hash: null,

search: '?name=king&gender=male&level=high', query: ' name=king&gender=siale&level=high', pathname： '/index'.

path: '/index?name=king&gender=male&level=high',

href： 1 <http://localhost:6666/index?name=king&gender=oiale&level=high'> }

url.format("url\_parse") is :

http ：//localhost ：6666/index?name=king&gender=mal6&l.evel=high

Process finished with exit code 0

图9.8解析与格式化url路径的方法

1. url路径转化

本节我们介绍一下url路径转化的方法，所谓路径转化就是在原有url路径的基础上增 加或替换标签的操作。在Node.js框架中，url模块提供了 url.resolveQ^\*法来实现这项功能。

本例ch09.url\_resolve.js主要代码如下：

01 /\*\*

02 \* ch09.url\_resolve.js

03 7

04 console.info(" url parse() '\*);

05 console.info();

06 var url = require('urr); //引入路径处理模块

07 /\*\*

08 \* 测试 url.resolve()方法

09 \* @type (string}

1. \*/
2. console.info(url.resolve(Vone/two/three', ’four3));
3. console.info(url.resolve('/one/two/three/\ 'four'));
4. console.info(url.resolve(7one/two/three', '/four'));
5. console.infoCurl.resolveC/one/two/three/\*, 7four'));
6. console.info(url.resolve(,<http://www.example.com/one'>, 'two'));
7. console.info(url.resolve('<http://www.example.com/one>,, '/two'));
8. console.info(url.resolve('<http://www.example.com/one/'>, 'two'));
9. console,info(url.resolve(,<http://www.example.com/one/'>, Vtwo\*))；

**【代码分析】**

第11-18行分别使用url.resolveO方法执行了 8种不同类型的url路径转化的操作；第 11行通过url.resolveO方法执行了第1种url路径转化的操作，其中源地址路径为 /one/two/three,插入或替换的标签为four,输出结果为/one/two/fbur：从第15〜18行打印输 出的结果来看，如果源地址路径是一个href类型的链接地址，则插入或替换的标签会以源 地址路径中域名后第1个“/”字符为基础路径进行操作，而如果插入或替换的标签以“/” 字符开头，则会完全替换掉源地址基础路径后的全部路径。

图9.9是本例程在Ubuntu环境下的输出结果。

Run

**Run <M».urt\_resolve.js** *L,*

能 寺 /usr/local/bin/node chO9,url\_resoive.js

url parseO

**» 4**

\_ /one/two/four

启 野 /one/tvo/three/four

r /four

*阳 '* /four

辱 [http://vwv.example.coin/tw。](http://vwv.example.coin/tw%e3%80%82)

X <http://www.exafliple.coin/two>

**7 ®** http://wwv.example,con/one/two

<http://wvw.example.com/tvo>

Process finished with exit code 0

图9.9 url路径转化的方法

提示：从本例打印输出的结果来看，url.resolveO方法的第一个源地址路径参数需要区 别是否为“href”链接地址路径，对于“href”链接地址路径的执行结果是与普通url路径 不同的-，

1. url参数转化url对象

本节我们介绍一下url参数转化url对象的方法，对于url对象我们在9.8节中有过介绍， 读者可以去参考一下。在Node.js框架中，执行url参数转化url对象的操作可以通过Query String模块的parse()方法来完成。

本例 ch09.querystring\_parse.js 主要代码如下：

01 /\*\*

02 \* ch09.querystring\_parse.js

03 \*/

04 console.info(" querystring parse() 一一")；

05 console.info(); ,

06 var url = require('uri\*); 〃引入路径处理模块

07 var querystring = requirefquerystring'); 〃引入路径处理模块

08 /\*\*

09 \* 测试 uri.parse()方法

1. \* @type {string}
2. \*/
3. var queryUrl = H[http://localhost:6666/index?name=king&gender=male&level=high](http://localhost:6666/index?name=king&gender=male&level=highM)[M](http://localhost:6666/index?name=king&gender=male&level=highM);
4. console.infoCtypeof queryUrl is :' + typeof uri.parse(queryUrl));
5. console.info();
6. console.info(\*querystring.parse(nqueryUrl") is : \*);
7. var url\_parse = url.parse(queryllrl).query;
8. /\*\* 「
9. \* 测试 querystring.parse()方法
10. \* @type {\*}
11. 7
12. var querystring\_parse = querystring.parse(url \_j)arse);
13. console.info(querystring\_\_parse);
14. console.infoQ;

【代码分析】

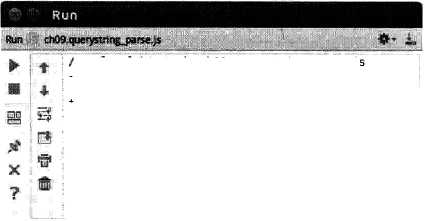
第07行引入Query String模块，同时赋于变量querystring；为了测试本例程，我们在 第12行中定义了一个url字符串路径：

w[http://localhost:6666/index?name=king!&gender=male&level=high](http://localhost:6666/index?name=king!&gender=male&level=highw)[w](http://localhost:6666/index?name=king!&gender=male&level=highw)

第16行调用url.parseO方法执行了解析url字符串路径的操作，返回结果保存在变量 url\_parse中；第21行调用了 querystring.parse()方法执行了 url参数转化url对象的操作，返 回结果保存在变量querystring\_parse中。

从图9.10的输出结果来看，第21行通过调用querystring.parse()方法，将第16行获取 的url参数返回值进行了序列化操作，操作后的结果为一个url对象，如图9.10所示，其具 体结果为：

{name: 'king\*, gender: 'male\*, level: 'high')



/usr/local/bin/node ch09. querystnng\_parse. js

querystring parseO

typeof queryUrl is : object

；Process finished with exit code Q

querystring.parse("queryUrl") is :

：{ name： 1 king \*, gender： 'male', level: 'high\* }

图9.10 url参数转化url对象的方法

第10章 TCP/UDP网络应用

本章我们向读者介绍Node.js框架对于TCP/UDP网络编程方面的支持，TCP (Transmission Control Protocol)和 UDP (User Datagram Protocol)协议属于 ISO 七层网络 模型中的传输层协议。其中TCP为IP环境下的数据提供可靠性传输，它提供的服务包括 数据流传送、可靠性、有效流控、全双工操作和多路复用，通过面向连接、端到端和可靠 的数据包进行发送；而UDP则不为IP环境提供可靠性、流控或差错恢复功能。简单来说， TCP对应的是可靠性要求高的应用，而UDP对应的则是可靠性要求低、传输经济的应用。

Node.js框架为设计人员提供了网络(Net)模块来支持TCP协议应用，数据报套接字 (UDP)模块来支持UDP协议应用，这两个模块提供了一系列与网络应用相关的方法，通 过这些方法就可以构建基本的网络应用了。

本章主要包括以下内容。

・Net模块方法与应用。

・UDP模块方法与应用。

・ 套接字网络应用。

10.1创建基本的TCP服务器

首先，我们从创建最基本的TCP服务器开始，所谓TCP就是指网络通信协议，是互 联网通信的最基本协议。大家所熟知的很多互联网通信应用(例如ICQ、QQ等)都是基 于TCP协议开发的，可见开发一个功能强大的TCP服务器难度还是很高的。Node.js框架 提供了一个网络(Net)模块来支持TCP协议，通过Net模块的net.createServer()方法来完 成创建TCP服务器的功能。

在本节这个基本的例程中，我们使用net.createServer()方法来创建基本的TCP服务器, 本例ch 10.net createServer.js主要代码如下：

01 /\*\*

02 \* Created by king on 15-5-19.

03 \*

04 \* ch10.net\_createServer.js

05 \*/

06 console.infoC net createServer() ");

07 console.info();

08 var net = require('nef);

09 var HOST = ,127.0.0.1,;

1. var PORT = 9696;

〃引入网络(Net)模块

〃定义服务器地址

//定义端口号

1. /\*\*
2. \*使用net.ServerClient()方法创建一个TCP服务器实例
3. \*同时调用listen()方法开始监听指定端口
4. \*传入net.ServerClient()的回调函数将作为connection事件的处理函数
5. \*/
6. console.infofNow create Server...');
7. console.info();
8. net.createServer(function(sock) (
9. /\*\*
10. \*打印输出服务器监听提示信息
11. \*/
12. console.log(\*Server listening on ' + HOST PORT); //服务器已经建立连接
13. console.info();
14. /\*\*
15. \*为socket实例添加一个data事件处理函数
16. \*/
17. sock.onCdata\*, function(data) (
18. console.log(\*socket on data...');
19. ))；
20. /\*\*
21. \*为socket实例添加一个close事件处理函数
22. 7
23. sock.onfclose', function(data) (
24. console.logfsocket on close..
25. })；

36

1. }).listen(PORT, HOST);

**【代码分析】**

第08行引入网络(Net)模块，同时赋于变量net；第09行定义了服务器地址名称 (HOST=127.0.0.1),该地址表示本机服务器地址；第10行定义了服务器端口号(PORT=9696), 注意定义端口号时要避免与其他端口发生冲突；第18~37行通过调用net.createServer()方 法创建了一个最基本的TCP服务器，net.createServer()语法如下：

语法：net.createServer([options][, connectionListener]); // 创建 TCP 服务器

net.createServer()方法用于创建TCP服务器；可选的第一个参数options是一个包含缺 省值｛allowHalfOpen:false)的对象，allowHalfOpen属性用于定义连接方式(全开或半开)， 缺省状态为全开方式，如果定义该属性值为true,则为半开连接方式；可选的第二个参数 connectionListener会被自动定义为connection事件的监听器，在实际应用中它被定义为一 个事件监听器的回调函数。

图10.1是本例程在Ubuntu环境下的输出结果。



3 辱 碓

图10.1创建基本的TCP服务器

因为本例程没有添加实际操作代码，也没有定义客户端，因此调试输出的结果没有实 际内容，仅仅是打印了一行提示信息“Now create server../'。

说明：本例程中提到了套接字(Socket)的概念，网络应用程序通常通过Socket向网 络发出请求或者应答网络请求。我们熟知的C++、C#、Java等主流编程语言均实现了 Socket 功能，Node.js框架虽为服务器端编程语言也不例外。

10.2创建基本的TCP客户端

上一节我们介绍了如何创建基本的TCP服务器，这一节我们介绍如何创建基本的TCP 客户端，有了客户端就可以与服务器进行通信了。所谓客户端，一般就是指安装在本机(个 人电脑)上与服务器进行通信的工具，例如我们在个人电脑上安装的QQ、MSN和飞鸽传 书等工具软件，都属于TCP客户端。在Node.js框架中通过Net模块的net.connect()方法来 完成创建TCP客户端的功能。

在本节这个基本的例程中，我们使用net.connect()方法来创建基本的TCP客户端功能, 本例chl0.net\_client.js主要代码如下：

01 /\*\*

02 \* ch10.net\_client.js

03 7

04 console.infoC net ServerClient() —

05 console.info();

06 var net = require(Mnetn); 〃引入网络(Net)模块

07 var HOST = '127.0.0.T; 〃定义服务器地址

08 var PORT = 9696; 〃定义端目导

09 /\*\*

1. \*使用net.connect()方法创建一个TCP客户端实例
2. \*/
3. var client = net.connect(PORT, HOST, function() {
4. console.log('client connected...');
5. console.info();
6. })；
7. /\*\*
8. \*为TCP客户端实例添加一个data事件处理函数
9. \*/
10. client.onCdata', function(data) (
11. console.info('client on data...'); \*
12. console.info();
13. ));
14. /\*\*
15. \*为TCP客户端实例添加一个end事件处理函数
16. \*/
17. dient.onCend', function() (
18. console.log('client disconnected1);
19. cx)nsole.info();
20. })；

**【代码分析】**

第12〜15行通过调用net.connect()方法创建了一个最基本的TCP客户端连接，并将返 回的客户端对象赋于变量client；第19〜21行为socket对象实例添加了数据data事件的处 理函数，在客户端，通过数据data事件可以向服务器端发送数据；第26〜28行为socket 对象实例添加了结束end事件的处理函数，客户端关闭时会触发end事件。

图10.2是本例程在Ubuntu环境下的输出结果。



,/usrAocat/bin/node chlQ.net\_client.js t net Serverclient（）

events.js：72

throw er； // Unhandled 'error' event

@ Error： connect ECONNREFUSED at errnoException （net.js：904：ii）

® at Object.afterConnect [as oncomplete] （net.js：895：19）

Process finished with exit code 8

图10.2创建基本的TCP客户端

因为本例程的客户端程序没有服务器可以连接，因此调试输出后的结果会报出错误信 息，没有关系，这个例程仅仅是向读者介绍客户端的基本编写方法，在后面的节中我们会 给出完整的服务器与客户端通信的例程。

10.3创建简单的TCP通信应用

这一节，我们在本章前两节内容的基础上创建一个简单的基于TCP协议的通信应用。 这个应用具有了基本的服务器端与客户端通信交互的功能，通过这个应用例程，读者将会 了解到Node.js框架是如何实现TCP通信功能的。

在本节这个应用例程中，我们将创建两个js脚本文件，一个用于实现服务器端代码, 另一个用于实现客户端代码。

其中，服务器端脚本文件ch 10.sc net createServer.js主要代码如下：

01 /\*\*

02 \* ch10.sc\_net\_createServer.js

03 \*/

04

05

06

07

08

09

console.info(M- console.info(); var net = require('net'); var HOST = ,127.0.0.r； var PORT = 9696;

/\*\*

net ServerClient()

〃引入网络(Net)模块

〃定义服务器地址

//定义端口号

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

\*使用net.ServerClient()方法创建一个TCP服务器实例

\*同时调用listen()方法开始监听指定端口

\*传入net.ServerClient()的回调函数将作为connection事件的处理函数

7

console.info('Now create Server...1);

console.info();

net,createServer(function(sock) {

*I\*\**

\*打印输出服务器监听提示信息

\*/

console.logfServer listening on 1 + HOST PORT); //服务器已经建立连接 console.info();

\*回调函数获得一个参数,该参数自动关联一个socket对象

\*在每一个connection事件中，该回调函数接收到的socket对象是唯一的

7

console.logf CONNECTED:' + sock.remoteAddress ++ sock.remotePort); console.info();

/\*\*

\*为socket实例添加一个data事件处理函数

\*/

sock.on('data', function(data) (

/\*\*

\*打印输出由客户端发来的消息

\*/

console.info('DATA \* + sock.remoteAddress + \*:"' + data + console.info();

尸\*

\*回发该数据，客户端将收到来自服务端的数据

7

sock.write('Server write :+ data +

})；

sock.on('close', function(data) (

42

43

44

45

46

47

为socket实例添加一个close事件处理函数

console.logfCLOSED:' + sock.remoteAddress + \* \* + sock.remotePort); console.info();

1. ))；
2. *I*
3. )).listen(PORT, HOST);

**【代码分析】**

第16~50行通过调用net.createServer()方法创建了一个简单的TCP服务器；第50行 通过调用net.server类的listen。方法在指定的主机(HOST=127.0.0.1)和端口(PORT=9696) 上接受连接；第18行回调函数中的参数sock是一个套接字(Socket)对象实例，Socket 其实是对TCP协议的一个基本封装接口(API),利用socket对象实例可以操作TCP协议 的基本功能。

第31〜41行为socket对象实例添加了数据data事件的处理函数，其回调函数中的参 数data用于接受客户端发来的数据；第35行打印输出了客户端发来的数据，同时打印输 出了 socket对象的remoteAddress属性值，该属性值描述了远程客户端的地址；第40行在 服务器端使用第18行定义的参数sock,通过socket.write()方法将客户端发来的数据加工后, 再次回传给客户端。

第45〜48行为socket对象实例添加了关闭close事件的处理函数，服务器关闭时会触 发close事件;其中，第46行打印输出了 socket对象的remoteAddress和remotePort属性值, 这两个属性分别用于描述远程地址和远程端口号。

我们再来分析客户端代码，客户端脚本文件chiO.sc net client.js主要代码如下：

01 /\*\*

02 • ch 10.sc\_net\_ciient.js

03 \*/

04 console.info(" net ServerClient() ”)；

05 console.infb();

06 var net = require("netn); //引入网络(Net)模块

07 var HOST = ,127.0.0.T; 〃定义服务器地址

08 var PORT = 9696; 〃定义端口号

09 /\*\*

1. \*使用netxonnect。方法创建一个TCP客户端实例
2. \*/
3. var client = net.connect(PORT, HOST, function() (
4. console.log('client connected\*);
5. console.info();
6. client.writefclient write : Hello Server!1);
7. })；
8. /\*\*
9. \*为TCP客户端实例添加一个data事件处理函数
10. \*/
11. client.on('data', function(data) {
12. console.log(data.toString());
13. console.info();
14. client.end();
15. ))；
16. /\*\*
17. \*为TCP客户端实例添加一个end事件处理函数
18. \*/
19. client.onfend\*, function() (
20. console.log('client disconnected1)；
21. console.info();
22. })；

**【代码分析】**

第15行通过使用client.writeO方法向服务器端发送数据，该方法其实就是socket.write() 方法，因为client参数就是-•个套接字对象；第20〜24行为变量client添加了数据data事 件的处理函数，在客户端打印输出服务器端发来的数据：其中，第23行调用Z client.end() 方法，在打印输出数据后，执行关闭客户端的操作；client.end()方法其实就是套接字上的 socket.cnd()方法。

图10.3是在Ubuntu环境下，使用WebStorm开发工具调试输出时，刚刚初始化的结果， 图10.4是服务器端输出的结果，图10.5是客户端输出的结果。

Run

**Ojc\_net\_c|ieHijs**

Run:

/usr/local/bin/node chlO.sc\_net\_createServer,]s net Serverclient 0

\_ Now create Server.,. 留w

图10.3创建简单的TCP通信应用(服务器端初始化)

如图10.3所示，我们先启动服务器端代码，运行后打印输出提示信息"Now create Server...”；在服务器成功启动后，我们运行客户端代码，如图10.5所示，打印输出了提示 信息"client connected":在客户端成功启动后，客户端第15行向服务器端发送了数据信 息(client write : Hello Server!)；此时，服务器被客户端连接成功，服务器端代码依次打 印输出了若干条提示信息，其中第35行打印输出了客户端发来的信息(DATA 127.0.0.1 : ''client write : Hello Server!M),如图10.4所示；服务器端第40行对接收到的客户端数据进 行了加工，并回写给了客户端；客户端第21行对服务器端回写的数据进行了打印输出 (Server write : "client write : Hello Server!M)，如图 10.5 所示。

说明：socket.end()方法用于半关闭套接字(Socket),如果第一个可选的参数data被 传入数据，则其等同于先调用了 socket.write(data, encoding)方*法,*然后再调用了 socket.end() 方法。

Nov create Server...

Server listening on 127.0.0.1:9696

/usr/local/bin/node chlO\_sc\_net\_createServer.】s net Serverclient()~

client connected

§

3

待

/usr/local/bm/node chio.sc\_net\_client. js net SenrerClientO

g弓

CONNECTS): 127.0.0.1:59602

DATA 127.0.0.1 : "client vnte : Hello Server? \*

CLOSH)： undefined undefined

Process finished with exit code 130

Server vnte ； "client wnte : Hello Server!\*

client disconnected

Process finished with exit code 0

图10.4创建简单的TCP通信应用(服务器端)

图10.5创建简单的TCP通信应用(客户端)

1. 创建TCP服务器的另一种方式

这一节，我们介绍使用另一种方式创建TCP服务器，该方式与本章10.1节的方式略有 不同，但是其本质原理是相同的。Node.js框架的网络(Net)模块提供了 connection事件 来完成该功能。

在本节这个例程中，我们使用connection事件来创建基本的TCP服务器，本例chlO.net\_ connection.js主要代码如下：

01 /\*\*

02 \* ch10.net\_connection.js

03 7

console.infof net connection()

04

05

06

07

08

09

10

11

12

13

14

15

16

17

18

19

20

console.info(); var net = require("net"); *i*

var HOST = '127.0.0. T; /

var PORT = 9696; h *i*

/\*\*

\*创建TCP服务器

\*/

var server = net.createServer();

尸\*

\*监听端口和主机

\*/

server.listen(PORT, HOST);

console.log(\*Server listening on ' + server.address()); console.info();

/\*\*

\*通过显式调用connection事件建立TCP连接

//引入网络(Net)模块 〃定义服务器地址

//定义端口号

1. \*/
2. server.on(,connection,, function(sock) (
3. console.logfCONNECTED:' + sock.remoteAddress ++ sock.remotePort);
4. });

**【代码分析】**

第16行通过调用net.server类的listen。方法在指定的主机和端口上接受连接；第17行 通过调用server.address()方法打印输出了绑定的服务器地址与端口号；第22〜24行通过显 式调用connection事件来建立TCP连接，其回调函数中会得到一个Socket对象实例，通过 该实例可以获取客户端的信息。

图10.6是本例程在Ubuntu环境下的输出结果。

Run

\_ ' - 勺 \*

G ' . /usr/tocal/bin/node chlO.net\_connection.js

" net connection()

■ 4

\_ Server listening on null

**X 3 1**

**X ®** ? 立

图10.6创建TCP服务器的另一种方式

因为本例程没有添加实际操作代码，也没有定义客户端，因此调试输出的结果没有实 际内容，仅仅是打印了一行提示信息“Server listening on null"。

说明：本例程通过显式调用connection事件来建立TCP连接，虽然该方式能达到同样 的功能效果，但我们还是建议使用10.1节中的方式来创建TCP服务器。

1. 服务器端绑定事件

Connection、listening 和 close 事件都是属于 net.Server 类的范畴。connection 事件是在 客户端向服务器端发送连接请求后被触发的，listening事件是在服务器端调用server.listen() 方法后被触发的，close事件是在调用server.close()方法后被触发的。

在本节这个应用例程中，我们将创建两个js脚本文件，一个用于实现在服务器端绑定 close事件方法的代码，另一个用于实现客户端代码。

其中，服务器端脚本文件ch 10.net close server.js主要代码如下：

01 /\*\*

02 \* ch10.net\_close\_server.js

03 \*/

04 console.info(H—— net close()

05 console.info();

〃引入网络(Net)模块

〃定义服务器地址

06 var net = requirefnef);

07 var HOST = \*127.0.0.1\*;



08

09

10

11

12

13

14

15

var PORT = 8877; //定义端口号

var server = net.createServer();

\*监听listening事件

\*/

**/**

1. serveronClistening', function() (
2. console.log('Server is listening on port1, PORT);
3. console.info();
4. });
5. /\*\*
6. \* 监听 connection 事件
7. \*/
8. server.onfconnection,, function(socket) {
9. console.log('Server has a new connection');
10. console.info();
11. server.close(); 〃调用 server.close()方法
12. })；
13. /\*\*
14. \*监听close事件
15. \*/
16. server.onCclose', function() {
17. console.log('Server is now closed1);
18. console.info();
19. });

35/\*\*

1. \*调用server.listen()监听服务器端口
2. \*/
3. server.listen(PORT, HOST);

**【代码分析】**

第30行调用server.listen。方法启动监听服务器端口的操作，该方法执行后，listening 监听事件将会被触发；第23〜26行通过变量server绑定connection连接事件来监听来自客 户端的连接请求，并在第24行打印输出提示信息；第26行通过调用server.close()方法执 行服务器关闭操作；第31〜34行通过变量server绑定close关闭事件来响应第26行执行的 关闭操作，并在第32行打印输出提示信息。

我们再来分析客户端代码，客户端脚本文件chlO.net\_close\_client.js主要代码如下：

/\*\*

02 \* ch10.net\_close\_client.js

03 \*/

04 console.infoC\* net close() —―");

05 console.info();

06 varnet = require("net"); 〃引入网络(Net)模块

07 var HOST =，127.0.0.T; 〃定义服务器地址

08 var PORT = 8877; //定义端口号

09 /\*\*

1. \*使用net.connect()方法创建一个TCP客户端实例
2. \*/
3. var client = net.connect(PORT, HOST, function() (
4. console.logCclient connected1);
5. console.info();
6. clientend();
7. ))；
8. /\*\*
9. \*为TCP客户端实例添加一个end事件处理函数
10. \*/
11. clientonfend', function() (
12. console.logCclient disconnected\*);
13. console.info();
14. })；

**【代码分析】**

客户端代码主要是通过net.connect。方法连接到服务器端，触发服务器端connection事 件，然后在connection事件的回调函数中执行server.close()方法，通过该方法再次触发close 事件。

图10.7是本例程在Ubuntu环境下的初始化效果。



Run cM0.net\_cIme\_ser^r.js *类' •*

q 舍 /usr/local/bm/node chlQ.net\_close\_server.js

net listening。

Server is listening on port 8877



图10.7服务器端绑定close事件(服务器初始化)

如图10.7所示，我们先启动服务器端代码，运行后打印输出提示信息"Server is listening on port 8877"；在服务器成功启动后，运行客户端代码，如图10.8所示；此时，客户端向 服务器端发送连接请求，服务器端接受客户端连接并打印输出提示信息“Server has a new connection,,,如图10.9所示；此时，服务器端第26行server.close()方法执行，触发了 close 事件处理方法，然后第32行打印输出了提示信息“Server is now closed”(如图10.9所示), 之后服务器端绑定的close事件响应完毕。

图10.8是客户端的输出结果。图10.9是服务器端的输出结果。



Run

**chid net., ciose.cfient**

/usr/local/bm/node chlQ.net\_close\_server. js

net listening()

Server is listening on port 8877

Server has a new connection

Server is now closed

/usr/local/bin/node chlQ.net\_dose\_ciient. js

net closeO

client connected

client disconnected

X导 ?愈

Process finished with exit code 0

Process finished with exit code 0

图 10.8

服务器端绑定close事件(客户端)

10.9服务器端绑定close事件(服务器端)

通过本例程代码可以看到，close事件会在服务器端通过执行server.close()方法 成功触发close事件后,用户可以在绑定close事件方法的回调函数中执行自定

说明: 后被触发, 义操作。

10.6获取服务器地址参数

这一节，我们介绍获取服务器地址参数的方法，服务器地址参数一般包括ip地址、端 口号和协议簇等信息。在Node.js框架中，网络(Net)模块提供了 scrver.address()方法来完 成此功能。

在本节这个例程中，实现了获取服务器地址参数的方法，本例程chlO.net\_server\_ address.js主要代码如下：

|  |  |  |
| --- | --- | --- |
| 01 | /\*\* |  |
| 02 | \* ch10.net\_server\_address.js |  |
| 03 | \*/ |  |
| 04 | console.infoC net server.address() "); | |
| 05 | console.info(); |  |
| 06 | var net = require('net'); | 〃引入网络(Net)模块 |
| 07 | var HOST = \*127.0.0.r； | 〃定义服务器地址 |
| 08 | var PORT = 7878; | //定义端口号 |
| 09 |  |  |
| 10 | \*创建TCP服务器 |  |
| 11 | 7 |  |
| 12 | var server = net.createServer(); |  |
| 13 | /\*\* |  |
| 14 | \*监听listening事件 |  |
| 15 | 7 |  |

1. server-onflistening\*, function() (
2. console.log('Server is listening on port\*, PORT);
3. console.info();
4. var addr = server.address();
5. console.info("opened server on");
6. console.info(M%i", addr);
7. console.info();
8. server.close();
9. })；

*II*……此处省略部分定事件的代码

42 server.listen(PORT, HOST);

**【代码分析】**

第42行调用server.listen()方法启动监听服务器端口的操作，该方法执行后，listening 监听事件将会被触发；第19行调用server.addressO方法执行获取服务器地址参数的操作， server.addressO方法用于执行获取服务器地址参数的操作，其返回值是一个包含服务器绑定 的地址、端口和协议簇的JSON格式数据(本例程中该返回值被赋于变量addr)。

第21行通过变量addr打印输出了服务器地址参数，注意这里使用了徇格式化参数， 因为参数addr是一个JSON格式的数据,关于格式化参数的内容在第1章中有过详细介绍。

图10.10是本例程在Ubuntu环境下的输出结果。



opened server on

("address"：-127.Q,0,l"/,faniily":"IPv4,\uport,,:7878)

/usr/iocal/bin/node chlO.net\_server\_address.]s net server.addressO

I Server is listening on port 7878 3耳

Server is now closed

©

Process finished with exit code 0

图10.10获取服务器地址参数的方法

如图10.10所示，通过server.addressO方法成功打印输出了服务器地址信息 ({”address"127.0.01',”fhmily”:“IPv4叩port”:7878})。

注意：在使用server.addressO方法时需要注意的一点是，server.addressO方法必须在 listening事件被触发后再使用，在listening事件发生前调用server.address()方法是无效的°

10.7获取当前服务器连接数

这一节，我们介绍获取当前服务器连接数的方法。一般来讲，一个服务器能够满足多 个客户端的连接请求，这个时候获取当前服务器的连接数就是非常重要的功能，在Node.js 框架中，网络(Net)模块提供了 server.getConnections()方法来完成此功能。

在本节这个应用例程中，我们将创建一个服务器端js脚本文件以及若干个客户端js脚 本文件来测试获取当前服务器连接数的方法。

其中，服务器端脚本文件chiO.net getconnections server.js主要代码如下：

Q4 /\*\*

w I / , :好.乎冷(冬厂；‘：技'笋凌出晶#务费秘魅脖”气睇咨Z " 1 "

02 \* ch10.net\_getconnections\_server.js

\*/

console.info(" ne

console.info();

var net = require(\*net'); var HOST = ,127.0.0.1,; var PORT = 8877;

net getconnections()

〃引入网络（Net）模块

〃定义服务器地址

〃定义端口号

/[[6]](#footnote-7)

\*创建TCP服务器

\*/

var server = net.createServer();

/[[7]](#footnote-8)•

\*监听listening事件

\*/

server.on('listening', function() {

console.logfServer is listening on port', PORT); console.info();

监听connection事件

server.on('connection', function(socket) ( console.logfServer has a new connection'); console.info(); server.getConnections(function (err, count) {

if(err) {

console.info(err.message);

} else (

console.info("current connections is ” + count); console.info();

})；

//server.close();

/\*\*

\*监听close事件

\*/

server.on('close', function() { console.log('Server is now closed'); console.info();

46

47

1. console.info();
2. })；
3. /\*\*
4. •调用server.listen()监听服务器端口
5. 7
6. server.listen(PORT, HOST);

【代码分析】

第53行通过调用net.server类的listen()方法在指定的主机(HOST=127.0.0.1)和端口 (PORT=8877)上接受连接；第23~35行通过变量server绑定connection连接事件来监听 来自客户端的连接请求，其中，第26〜33行通过server.getConnections()方法来获取当前服 务器连接数；第30行通过参数count打印输出了当前服务器的活跃连接数。

说明：被注销的第34行通过调用server.close()方法执行关闭服务器的操作，之所以注 销这行，是因为需要保持服务器运行状态，这样才能接受来自客户端的连接请求n

本例程创建了三个客户端代码，这三个客户端代码主要是实现连接服务器的功能。选 取其中一个客户端脚本文件ch 10.net getconnections clientA.js为例，其主要代码如下：

01 /\*\*

02 \* ch10.net\_getconnections\_clientA.js

03 \*/

04 console.info(n net getconnections() ");

05 console.info();

06 var net = requireC'net\*\*); 〃引入网络(Net)模块

07 var HOST = ,127.0.0.1,; 〃定义服务器地址

08 var PORT = 8877; //定义端口号

09 /\*\*

1. \*使用net.connect()方法创建一个TCP客户端实例
2. 7
3. var client = net.connect(PORT, HOST, function() {
4. console.logfclientA connected1);
5. console.info();
6. client.writefclient write : Hello Server!');
7. //client.end();
8. ))；
9. /\*\*
10. \*为TCP客户端实例添加一个end事件处理函数
11. 7
12. client.on('end\*, function() (
13. console.log('clientA disconnected');
14. console.info();
15. })；

我们注意到，该客户端代码与本章前面几个例程的客户端代码基本一致，主要是完成

J'连接服务器的功能；这里面需要强调的一点是第16行，其注销了 cliend.end()方法，主要

是为了保持客户端始终处于连接服务器的状态。

图10.11演示的是本例程客户端代码在Ubuntu环境下，使用WebStorm开发工具调试 输出时，刚刚初始化的结果。



Run 05 d\*W.neLge«»nnea3on^.senier4s

/usr/local/bin/node chlQ.net\_getconnections\_server.js net getconnections(T

Server is listening on port 8877

图10.11获取当前服务器连接数的方法（服务器端初始化）

服务器初始化成功后，我们选择启动第一个客户端，图10.12是本例程第一个客户端 代码在Ubuntu环境下输出的结果。图10.13演示的是第一个客户端连接完成后，服务器端 的状态变化。



/usr/Local/bin/node chlG.net\_getconnections\_clientA.js net getconnectionsO

■

clientA connected

/usr/local/bin/node chlQ.net\_getconnections\_server.js net getconnectionsO

a

Server is listening on port 8877

Server has a new connection

current connections is 1

图10.12当前服务器连接数的方法（客户端）

图10.13当前服务器连接数的方法（服务器端）

如图10.13所示，服务器通过server.getConnections。方法检测到一个当前的活动连接, 并打印输出了提示信息acurrent connections is 1”，这个就是我们刚才启动的第一个客户端; 下面，再依次启动第二个和第三个客户端，在成功连接到服务器后，服务器端的状态变化 如图10.14和图10.15所示。





/usr/tocal/bin/node chlO,net\_getconnections\_ser¥er.js net getconnectionsO

ctilO r»et\_get connection5\_setver. js

t ' /usrAocal/bin/node chlG.net\_getconnections\_server. js net getconnectionsO

Server is listening on port 8877

Server has a new connection

Server is listening on port 8877

Server has a new connection current connections is 1

current connections is 1

Server has a new connection

current connections is 2

Server has a new connection

current connections is 2

Server has a new connection

current connections is 3

10.14当前服务器连接数的方法（服务器端）

图10.15当前服务器连接数的方法（服务器端）

如图10.14与图10.15所示，在第二个和第三个客户端成功连接服务器后，服务器端依

次打印输出了提示信息"current connections is 2”和 ucurrent connections is 3",表明服务 器端同时检测到有多个客户端连接到了服务器。

10.8获取套接字地址

这一节，我们介绍获取套接字地址参数的方法，套接字地址参数是指由套接字绑定的 IP地址的参数信息。在Node.js框架中，网络(Net)模块提供了 socket.address()方法来完 成此功能，另外还提供了两个属性(socket.localAddesss和socket.localPort)来分别获得地 址和端口值。

在本节这个例程中，我们介绍获取套接字地址参数的方法，本例程服务器端脚本文件 ch 10.net socketaddr server.js 主要代码如下：

|  |  |  |
| --- | --- | --- |
| 01 |  |  |
| 02 | \* ch10.net\_socketaddr\_server.js |  |
| 03 | 7 |  |
| 04 | console.info(" net socket.address() "); | |
| 05 | console.infoO; |  |
| 06 | var net = require('nef); | 〃引入网络(Net)模块 |
| 07 | var HOST = \*127.0.0.1'; | 〃定义服务器地址 |
| 08 | var PORT = 6677; | 〃定义端口号 |
| 09 | /\*\* |  |
| 10 | \*创建TCP服务器 |  |
| 11 | \*/ |  |
| 12 | var server = net.createServer(); |  |
| 13 | y\*\* |  |
| 14 | \*监听listening事件 |  |
| 15 | \*/ |  |
| 16 | server.onflistening', function() ( |  |
| 17 | console.log('Server is listening on port1, PORT); | |
| 18 | console.infoO； |  |
| 19 | }); |  |
| 20 | /\*\* |  |
| 21 | \*监听connection事件 |  |
| 22 | 7 |  |

1. server.on('connection', function(socket) (
2. console.logfServer has a new connection');
3. console.infoO；
4. console.info(socket.address());
5. console.infoO;
6. console.info(socket.localAddress);
7. console.infoO；
8. console.info(socket.localPort);
9. console.info();
10. server.close();
11. })；

此处，省略部分绑定事件的代码

1. server.listen(PORT, HOST);

**【代码分析】**

第26行调用socket.address()方法执行获取套接字地址参数的操作，socket.address() 方法用于执行获取被套接字绑定的IP地址参数的操作，其返回值是一个包含被绑定的 地址、端口和协议簇的JSON格式数据；第28行与第30行使用socket.localAddress和 socket.localPort属性来获得被套接字绑定IP的地址和端口值。

另外，本例程还包括一个简单的客户端脚本文件，主要是完成连接服务器端并触发 connection事件的功能，具体代码与之前几个例程类似，在此就不详细地解释说明了。

图10.16是本例程在Ubuntu环境下的输出结果。通过socket.address()方法成功打印输 出了套接字地址信息({"address":"127.0.0.1","family":"IPv4","port":7878))；另外，分别打 印输出的 socket.localAddress 和 socket.localPort 属性值也与 socket.address()方法的返回值一 一对应。

ch? 5 js ■

/usr/local/bin/node chl0.net\_socketaddr\_server.}s net socket.address()

*土* Server is listening on port 6677 a ts

Server has a new connection

@ { address： '127.0.0.1', family： 'IPv4', port： 6677 } X 4 i

-127.0.Q.1 6677

Server is now closed

Process finished with exit code 0

图10.16获取套接字地址参数的方法

说明：通过本例程的输出结果，我们知道使用socket.address()方法得到的结果与 server.address()方法是一致的，这是因为socket.address()方法绑定的就是服务器的IP地址。

1. 获取远程地址

这一节，我们接着上一节的内容，介绍获取远程地址参数的方法，远程地址参数对于 服务器端来讲，就只是指客户端的参数信息。在Node.js框架中，网络(Net)模块提供了 两个属性(socket.remoteAddesss和socket.remotePort)来分别获得远程地址和端口值。

在本节这个例程中，介绍了获取远程地址参数的方法，本例程服务器端脚本文件 ch 10.net remoteaddr server.js 主要代码如下：

01 /\*\*

02 \* ch10.net\_remoteaddr\_server.js

03 \*/

04 console.info(" net socket.remoteaddress()-—")

05 console.info();

06 var net = require('net');

〃引入网络(Net)模块

〃定义服务器地址

//定义端口号

07 var HOST = \*127.0.0.1';

08 var PORT = 6677;

09 /\*\*

1. \*创建TCP服务器
2. \*/
3. var server = net.createServer();
4. /\*\*
5. \*监听listening事件
6. \*/
7. server.onClistening\*, function() (
8. console.logfServer is listening on port', PORT);
9. console.info();
10. })；
11. /\*\*
12. \* 监听 connection 事件
13. \*/
14. server.on('connection', function(socket) (
15. console.log(\*Server has a new connection');
16. console.info();
17. console.infoC'socket.remoteAddress is" + socket.remoteAddress);
18. console.info();
19. console.info("socket.remotePort is" + socket.remotePort);
20. console.info();
21. server.close();
22. })；

//……此处省略部分绑定事件的代码

1. server.listen(PORT, HOST);

**【代码分析】**

第26行与第28行使用socket.remoteAddress和socket.remotePort属性来获得远程地址 和端口值。另外，本例程还包括一个简单的客户端脚本文件，主要是完成连接服务器端并 触发connection事件的功能，具体代码与之前几个例程类似，在此就不详细地解释说明了。

图10.17是本例程在Ubuntu环境下的输出结果。通过socket.localAddress和socketJocalPort 属性成功打印输出了远程地址信息(远程地址为127.0.0.1,远程端口为56905)；其中， 远程地址与服务器端地址一致，这是因为我们的服务器与客户端是在同一台主机上测试的。

说明：关于远程地址和端口，感兴趣的读者可以在不同的主机上测试，看看 socket,remoteAddress 属性的取值°

> , ./usr/local/bin/node chlO.net\_reaoteaddr\_server.js

net socket.re»oteaddress()

Server is listening on port 6677

Server has a new connection socket.reaoteAddress is 127.0.0.1 socket.reaotePort is 56905

Server is now closed

Process finished with exit code 0

图10.17获取远程地址参数的方法

10.10使用套接字写数据

*这一节,*我们介绍一下使用套接字写数据的方法。在Node.js框架中，网络(Net)*模* 块提供了 socket.write()方法用于在服务器端与客户端进行相互写数据的操作，同时还提供 了多个属性(例如socket.bytesRead> socket.bytesWritten等)来获取相关的数据特性。

在本节这个应用例程中，将创建两个js脚本文件，一个用于实现服务器端写数据的代 码，另一个用于实现客户端写数据的代码。

其中，服务器端脚本文件chiO.net socketwrite server.js主要代码如下：

01 /\*\*

02 \* ch10.net\_socketwrite\_\_server.js

03 \*/

|  |  |  |
| --- | --- | --- |
| 04 | console.info("—— net socketwrite | "); |
| 05 | console.info(); | 〃引入网络(Net)模块 |
| 06 | var net = require('net'); |
| 07 | var HOST = \*127.0.0. V； | 〃定义服务器地址 |
| 08 | var PORT = 8877; | 〃定义端口号 |
| 09 | /\*\* |  |
| 10 | \*创建TCP服务器 |  |
| 11 | 7 |  |
| 12 | var server = net.createServer(); |  |
| 13 | *L* |  |
| 14 | \*监听listening事件 |  |
| 15 | \*/ |  |

1. server.onflistening\*, function() (
2. console.logCServer is listening on port', PORT);
3. console.info();
4. ))；
5. /\*\*
6. \* 监听 connection 事件
7. 7

23 server.on('connection', function(socket) (

1. console.logf Server has a new connection');
2. console.info();
3. /\*\*
4. \*为socket实例添加一个data事件处理函数
5. \*/
6. socket.on('data', function(data) (
7. /\*\*
8. \*打印输出由客户端发来的数据字节长度
9. \*/
10. console.infoCsocket.bytesRead is' + socket.bytesRead);
11. console.info();
12. /\*\*
13. \*打印输出由客户端发来的数据
14. \*/
15. console.infofDATA' + socket.remoteAddress + + data +
16. console.info();
17. /\*\*

\*回发该数据，客户端将收到来自服务端的数据

41

42

43

44

45

46

47

48

\*/

socket.write(\*Server write : \*" + data + '”')；

/\*\*

\*打印输出回发到客户端的数据字节长度

\*/

console.infoCsocket.bytesWritten is' + socket.bytesWritten); console.info();

1. })；
2. /\*•
3. \*关闭服务器
4. 7
5. server.close();
6. ))；

//……此处省略部分绑定事件的代码

65 server.listen(PORT, HOST);

**【代码分析】**

第29〜49行通过参数socket绑定了 data事件来监听由客户端发来的数据信息，其回 调函数中定义的参数data用来表示客户端发来的数据信息；第33行使用socket.bytesRead 属性来获得客户端发来的数据信息的字节长度值；第38行通过参数data打印输出了由客 户端发来的数据信息：第43行将客户端发来的数据进行改写后，再回传给客户端：第47 行使用socket.bytesWritten属性打印输出回传给客户端数据信息的字节长度值；第53行调 socket.close()方法关闭服务器。

另外，本例程还包括-个简单的客户端脚本文件，主要是完成与服务器端相互收发数 据的功能，其脚本文件ch 10.net socketwrite client.js主要代码如下：

01 /\*\*

02 \* ch10.net\_socketwrite\_\_client.js

03 \*/

04 console.info(M net socketwrite —・”);

05 console.info();

06 var net = require("netw);

07 var HOST ='127.0.0.1';

08 var PORT = 8877;

09 var sWriteContent = "client write

〃引入网络(Net)模块 〃定义服务器地址 〃定义端口号

Hello Server!"; 〃定义字符串数据

1. /\*\*
2. \*使用netxonnect()方法创建一个TCP客户端实例
3. \*/
4. var client = net.connect(PORT, HOST, function() (
5. console.logfclient connected1);
6. console.info();
7. client.write(sWriteContent);
8. })；
9. 〃\*
10. \*为TCP客户端实例添加一个data事件处理函数
11. \*/
12. client.onCdata1, function(data) (
13. console.log(data.toString());
14. console.info();
15. console.infoCsocket.bytesRead is' + client.bytesRead);
16. console.info();
17. client.end();
18. ))；
19. *I\*\**
20. \*%TCP客户端实例添加一个end事件处理函数
21. \*/
22. client.onCend\*, function() (
23. console.logfclient disconnected1);
24. console.info();
25. })；

**【代码分析】**

第13〜17行通过调用net.connect()方法创建了一个最基本的TCP客户端连接，并将返 回的客户端套接字对象赋于变量client；第16行通过使用client.write0方法向服务器端发,送 数据，该方法其实就是socket.write。方法，因为client参数就是一个套接字对象：第21〜 27行为变量client添加了数据data事件的处理函数，在客户端打印输出服务器端发来的数 据；其中，第26行调用了 client.end()方法关闭客户端；第24行在客户端通过使用 socket.bytesRead属性获得服务器端发来的数据信息的字节长度值。

图10.18演示的是本例程服务器端代码在Ubuntu环境下，使用WebStorm开发工具调 试输出时，刚刚初始化的结果。

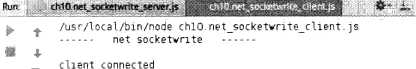


Zusr/local/bin/node chlO.net\_9Qcketwrite\_server.]s net socketwrite

Server is listening on port 的77

图10.18使用套接字写数据的方法(服务器端初始化)

图10.19演示的是本例程客户端代码在Ubuntu环境下输出的结果。图10.20演示的是 本例程服务器端代码在Ubuntu环境下输出的结果。



Zusr/local/bin/node chlO.net\_socketwnte\_server.)s net socketwrite

Server is listening on port 8677

Server has a new connection

Server write : "client write : Hello Server!

DATA 127.0.0.1 : "client write ； Hello Server!

Process finished with exit code 0

Process finished with exit code 0

图10.19使用套接字写数据(客户端)

图10.20使用套接字写数据(服务器端)

如图10.18所示，我们先启动服务器端代码，运行后打印输出提示信息“ Server is listening on port 8877":在服务器成功启动后，运行客户端代码，如图10.19所示，打印输 出了提示信息“client connected"；在客户端成功启动后，客户端第16行向服务器端发送 了数据信息“client write : Hello Server!":此时，服务器被客户端连接成功，服务器端代 码依次打印输出了若干条提示信息，其中第33行打印输出了客户端发来数据信息的字节长 度(socket.bytesReadis28),第 38 行打印输出了数据信息的内容(DATA 127.0.0.1 : "client write : Hello Server!"),如图10.20所示；服务器端第43行向对接收到的客户端数据进行 了加工，并回写给了客户端，并在第47行打印输出了回写到客户端数据信息的字节长度 (socket.bytesWritten is 45),如图10.20所示；客户端第22行对服务器端回写的数据进行 了打印输出(Server write : "client write : Hello Server!"),并在第24行打印输出了客户端 收到的数据信息的字节长度(socket.bytesWritten is 45),如图10.19所示。

说明：socket.bytesRead与socket.bytesWritten属性分别用于表示套接字接受和发送数 据的字节长度“

10.11

控制套接字数据流的应用

据流是非常重要的，它可以在文件上传和下载操作中提供完整的支持。在Node.js框架中， 网络(Net)模块提供了 socket.pause()^ socket.resume()方法用于暂停和恢复data事件的操 作，同时还提供了多个属性(例如socket.bytesRead> socket.bytesWritten等)来获取相关的 数据特性。

在本节这个应用例程中，将创建多个js脚本文件，其中一个用于实现服务器端控制数 据流的代码，另外几个用于实现客户端发起数据流的代码。

其中，服务器端脚本文件ch 10.net\_socketdata\_server.js代码如下：

01

02

03

04

05

06

07

08

/\*\*

\* ch 10.net\_socketdata\_server.js 7

console.info("- console.info(); var net = requirefnet'); var HOST = '127.0.0.1'; var PORT = 8877;

net socketdata ");

〃引入网络(Net)模块 〃定义服务器地址

//定义端口号

09 var bSockData = true;

1. /\*\*
2. \*创建TCP服务器
3. 7
4. var server = net.createServer();
5. /\*\*
6. \*监听listening事件
7. 7
8. server.on('listening\ function() (
9. console,logCServer is listening on port1, PORT);
10. console.info();
11. })；
12. 〃\*
13. \* 监听 connection 事件
14. \*/
15. server.onCconnection1, function(socket) (
16. console.logfServer has a new connection');
17. console.info();
18. if(bSockData) (
19. socket.resume();
20. bSockData = false;
21. } else(
22. socket.pause();
23. bSockData = true;
24. }
25. /\*\*
26. \*为socket实例添加一个data事件处理函数
27. *\*1*
28. socket.on('data', function(data) (
29. /\*\*
30. \*打印输出由客户端发来的消息
31. \*/
32. if(socket.bytesRead > 32) (
33. console.info('DATA' + socket.remoteAddress + ':n, + Mis too long!" +,H,);
34. console.info();
35. } else (
36. console.infofDATA' + socket.remoteAddress + \*:"' + data +
37. console.info();
38. )
39. /\*\*
40. \*回发该数据，客户端将收到来自服务端的数据
41. 7
42. socket.write(\*Server write :+ data +
43. })；
44. /\*\*
45. \*关闭服务器
46. \*/
47. //server.close();
48. ))；

//……此处省略部分绑定事件的代码

68 server.listen(PORT, HOST);

**【代码分析】**

第24~57行通过变量server绑定connection事件来监听来自客户端的连接请求，其回 调函数中定义了套接字参数socket；第27〜33行通过if-else条件判断语句依次调用 socket.pause。方法与socket.resume()方法，完成依次暂停或恢复套接字data事件的操作；关 于socket.pause()方法和socket.resume()方法的语法如下：

语法：socket.pause(); // 暂停套接字(socket) data 事件

语法：socket.resume(); // 恢复套接字(socket) data 事件

第41 ~47行使用if^else条件判断语通过socket.bytesRead属性值来判断客户端发来的

数据信息的字节长度值，凡是长度大于32

的则不在服务器端打印输出该数据信息；第51



行将客户端发来的数据进行改写后，再 回传给客户端。另外，为了保证服务器 一直处于监听状态，注销了第56行的 关闭服务器(server.close。)的代码。

图10.21演示的是本例程服务器端 代码在Ubuntu环境下，使用WebStorm 开发工具调试输出时，刚刚初始化的结 果。

Run

Mg虫地弊理遮 一 . 壁，上

G a /usr/local/bin/node chlG.net\_socketdata\_server.js net socketdata

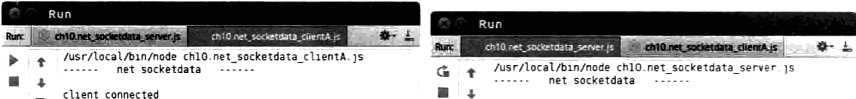
**■ 4**

\_ Server is listening on port 8877

图10.21控制套接字数据流的应用(服务器端初始化)

下面，简单介绍一下本应用的几个客户端脚本文件，其主要是完成与服务器端相兀收 发数据的功能，第一个客户端脚本文件chiO.net socketdata clientA.js （鉴于篇幅，省略了 代码，可参考源文件）。这个客户端代码与之前几个应用例程的客户端代码功能类似，需 要说明的就是代码里定义了一个字符串数据"client write : Hello Server A!"，该字符串在 由客户端向服务器端发送数据时使用。

图10.22演示的是第一个客户端（将其命名为客户端A）代码在Ubuntu环境下，使用 WebStorm开发工具输出的结果。图10.23演示的是客户端A成功连接到服务器后，服务器 端代码调试输出的结果变化情况。



Server is listening on port 8877

Server write : "client write : Hello Server A!"

Server has a new connection

*，*client disconnected

DATA 127.0.Q.1 : "client write : Hello Server A!'

X

X 1

Process finished with exit code 0

图10.22控制套接字数据流的应用（客户端A）

图10.23控制套接宇数据流的应用（服务器端）



如图10.23所示，在客户端A成功连接到服务器后，服务器端打印输出了相关提不信 息以及客户端发来的数据信息“DATA 127.0.0.1 : "client write : Hello Server A!""=而客户 端也打印输出了服务器端回写的数据信息"Server write : "client write : Hello Server A!"", 如图10.22所示。

这里，需要读者明确的是服务器端代码中的变量bSockData此时已经被设置为false, 因此socket.pause（）方法被执行，此时套接字data事件被暂停触发。

下面，我们启动第二个客户端脚本文件chIO.net\_socketdata\_clientB.js，该脚本文件的 代码与客户端A的代码基本一样，仅仅是代码里定义了一个新的字符串数据,l client write: Hello Server BB!"。

图10.24演示的是第二个客户端（将其命名为客户端B）代码在Ubuntu环境下输出的 结果。我们注意到仅仅打印输出了连接服务器成功的提示信息（client connected）,并没有 打印输出服务器回写的数据信息。

因此，我们要了解一下服务器端发生了什么情况呢？图10.25演示的是客户端B成功 连接到服务器后，服务器端代码调试输出的结果的变化情况。

Run

**ch10.net\_soctetaaca\_a»eiufl.ji**

/usr/local/bin/node chlO.net\_socketdata\_server.js net socketdata

Server is listening on port 8877

囹

弓

Zusr/local/bin/node chl0.net\_socketdata\_clientB.js net socketdata

client connected

Server has a new connection

DATA 127,0.0.1 : "client write : Hello Server A!"

Server has a new connection

从图10.25打印输出的结果可以看到，客户端B肯定是成功连接到了服务器，服务器 也打印输出了成功连接的提示信息，但仅仅是连接成功了，而数据操作并没有任何实际响 应；因此，套接字data事件确实是被暂停了，由于socket.pause()方法的执行，data事件没 有被客户端B触发。

到这里还没有结束，我们继续本应用的测试，创建第三个客户端脚本文件 ch 10.net\_socketdata\_clientC.js,尝试去恢复data事件；第三个客户端脚本文件与前两个客户 端脚本文件类似，仅仅是代码里定义了一个新的字符串数据“client write : Hello Server CCC!" »

图10.26演示的是第三个客户端(将其命名为客户端C)代码在Ubuntu环境下输出的 结果。我们注意到不仅打印输出了连接服务器成功的提示信息(client connected),还打印 输出了服务器回写的数据信息 uServer write : "client write : Hello Server CCC!""。

下面，我们看一下服务器端发生了什么变化呢？图10.27演示的是客户端C成功连接 到服务器后，服务器端代码调试输出的结果的变化情况。



client connected

X

目

Server write ； "client vnte : Hello Server CCC!M

7

冒 DATA 127 0.0.1 : "client vnte : Hello Server A!'

*懿* Server has a new connection

client disconnected

Server has a new connection

Process finished with exit code 0

DATA 127.0.0.1 : "client vnte : Hello Server CCC!"

图10.26控制套接字数据流的应用(客户端C)

图10.27控制套接字数据流的应用(服务器端)

如图10.27所示，在客户端C成功连接到服务器后，服务器端打印输出了相关提示信 息以及客户端发来的数据信息 “DATA 127.0.0.1 : "clientwrite : Hello Server CCC!",,。

10.12创建UDP服务器

首先，我们从创建最基本的UDP服务器开始，UDP (User Datagram Protocol,用户数 据报协议)是开放式系统互联(OSI, Open System Interconnection)参考模型中一种无连接 的传输层协议，提供面向事务的简单不可靠信息传送服务。

Node.js框架提供了一个数据报套接字(UDP/Datagram)模块来支持UDP协议，通过 UDP/Datagram模块的dgram.createSocket()方法来完成创建UDP服务器的功能。

本节将使用dgram.createSocket()方法来执行创建数据报套接字的操作，本例程 ch 10.udp createSocket.js 主要代码如下：

01 /\*\*

02 \* ch10.udp\_createSocket.js

7

console.info(M UDP Server

console.info();

var dgram = requirefdgram');

〃引入网络(UDP)模块

〃定义服务器地址

〃定义端口号

var HOST = '127.0.0.1';

var PORT = 12345;

,\*

\*创建UDP服务器

*\*1*

console.info('Now create UDP Server...'); console.info();

\*使用dgram.createSocket()^法创建一个UPD服务器

\*/ var server = dgram.createSocket('udp4');

/\*\*

\*为UDP服务器添加一个listening事件处理函数

7

server.onflistening', function () ( console.log('UDP Server listening on...'); console.info();

})；

/\*\*

\*为UDP服务器添加一个message事件处理函数

\*/

server.on('message', function (message, remote) { console.info('emitted ''message" event...'); console.info();

server.close();

})；

/\*\*

\*为UDP服务器添加一个error事件处理函数

\*/

server.onferror', function(err) { console.log("server error:\n" + err.stack); console.info();

server.close();

})；

/\*\*

\*为UDP服务器添加一个close事件处理函数

\*/

server.onCclose', function() ( console.log(\*server closed1); console.info();

})；

1. /\*\*
2. \*为UDP服务器绑定主机和端口
3. \*/
4. server.bind(PORT, HOST);

【代码分析】

第06行引入UDP/Datagram模块，同时赋于变量dgram；第07行定义了服务器地址名 称(HOST=127.0.0.1),该地址表示为本机服务器地址；第17行通过调用dgram.createSocket() 方法创建了一•个udp4类型的数据报套接字，并将其返回值赋于变量server；第51行通过 调用server.bind()方法在指定的主机和端口上绑定UDP数据报。

第28~32行通过变量server绑定message事件的处理函数，在服务器端通过message 事件接收客户端发送的数据报；第31行通过调用server.close()方法关闭UDP服务器。

图10.28是本例程在Ubuntu环境下的输出结果。 '



Run 轲 diW.udp\_servet.js (1]

/usr/local/bin/node chlO.udp\_server.js UDP Server

Nov create UDP Server...

UDP Server listening on...

Cl *t* a a

图10.28创建UDP服务器的基本方法(服务器端)

因为本例程没有添加实际操作代码，也没有定义客户端，因此服务器启动后仅仅是打 印了一行提示信息<4Now create UDP server...v ；第51行server.bind()函数反复执行后，变 量server绑定的listening事件被触发，第22行执行后打印输出了提示信息“UDP Server listening on...,,。

说明：本例程中使用到了 udp4与udp6类型参数，分别对应IPv4与IPv6协议组，在 使用dgram.createSocket()方法创建数据报套接字时，需要•选择udp4与udp6类型参数中的 一个；目前，由于IPv4协议组还在被广泛地使用中，因此dgram.createSocketO方法一般会 选择udp4类型参数。

10.13 创建UDP客户端

上一节，我们从创建最基本的UDP服务器开始，这一节介绍创建基本的UDP客户端 的方法。Node.js框架提供了一个数据报套接字(UDP/Datagram)模块来支持UDP协议， 通过UDP/Datagram模块的dgram.createSocketO方法来完成创建UDP客户端的功能。

本节将使用dgram.createSocketO方法来完成创建UDP客户端的操作，本例程 chlO.udp\_client.js 主要代码如下：

02 \* ch10.udp\_client.js

03 \*/

04 console.info(" UDP Client

05 console.info();

〃引入网络(UDP)模块

〃定义服务器地址

〃定义瑞口号

06 var dgram = require('dgram');

07 var HOST = \*127.0.0. V;

08 var PORT = 12345;

09 var message = new Buffer('UDP Client to Server: Hello Server!'); 〃定义数据包

1. /\*\*
2. \*创建UDP客户端
3. \*/
4. console.info('Now create UDO Client...');
5. console.info();
6. /\*\*
7. \*使用dgram.createSocket方法创建一个UDP客户端
8. \*/
9. var client = dgram.createSocket('udp4');
10. /\*\*
11. \*向服务器发送UDP数据报
12. \*/
13. client.send(message, 0, message.length, PORT, HOST, function(err, bytes) (
14. if(err) (
15. throw err;
16. }
17. console.logfUDP message sent to...');
18. console.info();

28

1. \*关闭客户端
2. \*/
3. client.close();
4. })；
5. /\*\*
6. \*为UDP客户端添加一个close事件处理函数
7. \*/
8. client.on('close', functionf) (
9. console.log('client disconnected');
10. console.info();
11. ))；

**【代码分析】**

第18行通过调用dgram.createSocket。方法创建了一个udp4类型的数据报套接字，并 将其返回值赋于变量client；第22〜32行通过调用client.send()方法在指定的主机和端口上 向服务器端发送数据报；第31行通过调用client.close()方法关闭客户端。

图10.29是本例程在Ubuntu环境下的输出结果。



Run ch10.i«ip\_die(«Js(i)

Now create UDO Client...

/usr/local/bm/node chlO.udp\_cl.ient. js UDP Client

IDF message sent to.

client disconnected

Process finished with exit code 0

图10.29创建UDP客户端的基本方法（客户端）

因为本例程没有添加实际操作代码，也没有可连接的服务器，因此客户端启动后仅仅 是打印了一行提示信息“Now create UDP Client..."；第22行clinet.sned（）方法被执行后, 依次打印输出了提示信息“UDP message sent 。

说明：一个绑定了的数据报套接字会保持node进程运行来接收数据报；如果使用 socket.send（）方法绑定失败，则error事件会被触发；在极少情况下（例如，客户端尝试绑 定一个已关闭的套接字），该方法才会抛出error事件以

1. 创建简单的UDP应用

这一节，我们在前两节内容的基础上创建一个简单的基于UDP协议的通信应用。这个 应用具有基本的服务器端与客户端通信交互的功能，通过这个应用例程，读者将会了解到 Node.js框架是如何实现UDP通信功能的。

在本节这个应用例程中，将创建两个js脚本文件，一个用于实现服务器端的代码，另 一个用于实现客户端代码。

其中，服务器端脚本文件chlO.udp\_server.js主要代码如下：

|  |  |  |
| --- | --- | --- |
| 01 |  |  |
| 02 | \* ch10.udp\_server.js |  |
| 03 | 7 |  |
| 04 | console.info("-——UDP ServerClient | ***h*** |
| 05 | console.info(); |  |
| 06 | var dgram = require('dgram'); | //引入网络（UDP）模块 |
| 07 | var HOST = ”27.0.0.1'; | 〃定义服务器地址 |
| 08 | var PORT = 12345; | 〃定义端口号 |
| 09 | /\*\* |  |
| 10 | •创建UDP服务器 |  |
| 11 | \*/ |  |

1. console.infofNow create UDO Server...1);
2. console.info();
3. /\*\*
4. \*使用dgram.createSocket。方法创建一个UPD服务器
5. \*/
6. var server = dgram.createSocket('udp4');
7. /\*\*
8. \*为UDP服务器添加一个listening事件处理函数
9. \*/
10. server.onClistening\*, function () (
11. var address = server.address();
12. console.logfUDP Server listening on ' + address.address + ":" + address.port);
13. console.info();
14. ))；
15. /\*\*
16. \*为UDP服务器添加一个,'message1\*事件处理函数
17. \*/
18. server.onCmessage1, function (message, remote) (
19. console.log(MUDP Server received from " + remote.address + + remote.port);
20. console.logf ・ " + message);
21. console.info();
22. server.close();
23. })；
24. /\*\*
25. \*为UDP服务器添加一个error事件处理函数
26. 7
27. server.onC^rror', function(err) (
28. console.log("server error:\n" + err.stack);
29. server.close();
30. })；
31. /\*\*
32. \*为UDP服务器添加一个close事件处理函数
33. \*/
34. server.onfclose\*, function() (
35. console.logfserver closed\*);
36. console.info();
37. })；
38. 尸\* /
39. \*为UDP服务器绑定主机和端口
40. \*/
41. server.bind(PORT, HOST);

**【代码分析】**

第21〜25行通过变量server绑定listening事件来监听服务器端口，第22行通过调用 server.address。方法获得服务器地址和端口号，并在第23行打印输出了地址和端口信息； 关于socket.address()方法的语法说明如下：

语法：socket.address。； *II*返回套接字地址信息

socket.address()方法返回了一个包含套接字地址信息的对象；对于UDP套接字而言，

这个对象会包含地址(address)、地址簇(family)和端口号(port)这些数据信息。

下面，我们再来分析客户端代码，客户端脚本文件chlO.udp\_client.js主要代码如下:

01 /\*\*

|  |  |  |
| --- | --- | --- |
| 02 | \* ch10.udp\_client.js |  |
| 03 | 7 |  |
| 04 | console.info(n UDP Serverclient ")； | |
| 05 | console.info(); |  |
| 06 | var dgram = require('dgram'); | 〃引入网络(UDP)模块 |
| 07 | var HOST = '127.0.0.1'; | //定义服务器地址 |
| 08 | var PORT = 12345; | 〃定义端口号 |

09 var message = new Buffer('UDP Client to Server: Hello Server!');

10 /•\*

1. •创建UDP客户端
2. \*/
3. console.info('Now create UDO Client...');
4. console.info();
5. /\*\*
6. \*使用dgram.createSocket方法创建一个UDP客户端
7. 7
8. var client = dgram.createSocket('udp4,);
9. /\*\*
10. \*向服务器发送UDP数据报
11. \*/
12. client.send(message. 0, message.length, PORT, HOST, function(err, bytes) (
13. if (err) throw err;
14. console.logfUDP message sent to' + HOST PORT);
15. console.info();
16. console.info(bytes);
17. console.info();
18. client.close();
19. ))；
20. /\*\*
21. \*为UDP客户端添加一个close事件处理函数
22. \*/
23. client.on('close', function。{
24. console.logfclient disconnected1);
25. console.info();
26. })；

**【代码分析】**

第23行通过回调函数err参数，判断发送的数据报是否出现了异常；第24行打印输出 了客户端地址和端口信息；第26行通过回调函数bytes参数，打印输出了客户端发送到服 务器端数据报的字节长度信息；第28行通过调用client.close()方法关闭客户端；第33〜36行通过变量client绑定了关闭close事件的处理函数，客户端关闭时会触发close事件。 首先，我们启动UDP服务器，图10.30演示的是本例程服务器端代码在Ubuntu环境 下，使用WebStorm开发工具调试输出时，刚刚初始化的结果。

Run

**Rwi® chiO.udp\_sefver.js —**

G a /usr/local/bm/node chlG.udp server. js

UDP Serverclient

**Ei ，**

\_ Now create UDO Server,..

as m

W UDP Server listening on 127.0.0.1:12345

统U

x ® 1

*7* 盈

图10.30创建简单的UDP应用（服务器端初始化）

然后启动本例程客户端，图10.31演示的是本例程客户端代码在Ubuntu环境下，使用 WebStorm开发工具输出的结果。

从图10.31的显示结果可以看到，不仅打印输出了发送数据报的服务器的地址和端口 的信息（UDP message sent to 127.0.0.1:12345）,还打印输出了数据报的字节长度（36）；

下面，我们看一下服务器端发生了什么变化呢？图10.32演示的是客户端数据报成功 发送到服务器后，服务器端代码调试输出的结果的变化情况。

A 金；/usrAocal/bin/node chlQ.udp\_server.js UDP Serverclient ~

■ 4

\_ Nov create UDO Server...

Now create UDO Client...

UDP Hessage sent to 127.0.0.1:12345

36

client disconnected

Process finished with exit code 0

Zusr/local/bin/node chlO.udp\_ciient.js UDP Serverclient

图10.31创建简单■的UDP应用（客户端）

UDP Server listening on 127.0.0.1:12345 X」

导 UDP Server received fro« 127.0.0.1:35020 X - UDP Client to Server : Hello Server! 7 面

server closed

Process finished with exit code Q

图10.32创建简单的UDP应用（服务器端）

如图10,32所示，服务器成功接收到客户端发来的数据报后，服务器端代码依次打印 输出了若干条提示信息，其中第30行打印输出了客户端的地址和端口信息"UDP Server received from 127.0.0.1:35020”，第31行打印输出了客户端发来的数据报信息“UDP Client to Server : Hello Server!"。

说明：通过本例程的演示结果可以看到，UDP数据报与TCP数据包还是有着非常明显 的区别的，UDP数据报发送到服务器后，服务器一般是不能回写给客户端的，而TCP数据 包是完全可以的a当然，这点区别也是符合UDP与TCP两种协议的设计原理的，UDP协 议就是为了快速而安全地发送大数据包而设计的，所以不需要考虑回写等复杂且影响效率 的操作。

1. UDP广播服务的实现

在前面几个介绍UDP的例程中，我们使用的都是UDP单播服务方式，这一节将介绍 UDP广播服务方式。所谓广播方式，就是将数据报发送到网络中的每一台主机上的方式， 其与单播方式是相互对应的。Node.js框架的UDP/Datagram模块提供了一个 socket.setBroadcast()方法来实现广播服务的功能。

在本节这个应用例程中，将创建两个js脚本文件，一个用于实现服务器端的代码，另 一个用于实现客户端代码。

其中，服务器端脚本文件ch 10.udp broadcast server.js主要代码如下：

01 /\*\*

02 \* ch10.udp\_broadcast\_server.js

03 \*/

04 console.info(H UDP broadcast

05 console.info();

〃引入网络（UDP）模块

〃定义服务器地址

〃定义端口号

06 var dgram = require('dgram');

07 var HOST = \*127.0.0.T;

08 var PORT = 12345;

09 /\*\*

1. \*创建UDP服务器
2. \*/
3. console.info('Now create UDO Server../);
4. console.info();
5. /\*\*
6. \*使用dgram.createSocket()方法创建一个UPD服务器
7. \*/
8. var server = dgram.createSocket('udp4\*);
9. /\*\*
10. \*为UDP服务器添加一个listening事件处理函数
11. 7
12. server.onClistening', function ()(
13. var address = server.address();
14. console.logfUDP Server listening on' + address.address +":" + address.port);
15. console.info();
16. })；
17. /\*\*
18. \*为UDP服务器添加一个message事件处理函数
19. \*/
20. server.onCmessage', function (message, remote) {
21. console.log(MUDP Server received from " + remote.address + + remote.port);
22. console.logC+ message);
23. console.info();
24. server.close();
25. })；
26. /\*\*
27. \*为UDP服务器添加一个error事件处理函数
28. \*/
29. server.on(\*error\*» function(err) (
30. console.log(wserver error:\n" + err.stack);
31. server.close();
32. })；
33. /\*\*
34. \*为UDP服务器添加一个close事件处理函数
35. \*/
36. server.on('close', function() (
37. console.log('server closed');
38. console.info();
39. ))；
40. /\*\*
41. \*为UDP服务器绑定主机和端口
42. 7
43. server.bind(PORT);

**【代码分析】**

第52行通过调用server.bind()方法在指定的端口上绑定UDP数据报，虽然第07行定 义了服务器地址，由于服务器要接收客户端的广播数据报，因此server.bind()方法仅仅绑定 了端口号。

我们再来分析客户端代码，客户端脚本文件ch 10.udp broadcast client.js主要代码如下:

01尸\*

|  |  |  |
| --- | --- | --- |
| 02 | \* ch10.udp\_broadcast\_\_client.js |  |
| 03 | \*/ |  |
| 04 | console.info(M-— UDP broadcast - | "); |
| 05 | console.info(); |  |
| 06 | var dgram = require('dgram'); | 〃引入网络(UDP)模块 |
| 07 | var HOST = '255.255.255.255'; | 〃定义服务器地址 |
| 08 | var PORT = 12345; | 〃定义端口号 |

09 var message = new Buffer('UDP Client to Server: Hello Server!\*);

1. /\*\*
2. \*创建UDP客户端
3. \*/
4. console.infofNow create UDO Client...');
5. console.info();
6. /\*\*
7. \*使用dgram.createSocket方法创建一个UDP客户端
8. \*/
9. var client = dgram.createSocket(,udp4,);
10. /\*\* ,
11. \*绑定套接字方法函数
12. \*/
13. client.bind(function () (
14. client.setBroadcast(true);
15. ))；
16. /\*\*
17. \*向服务器发送UDP数据报
18. \*/
19. client.send(message, 0, message.length, PORT, HOST, function(err, bytes) (
20. if (err) throw err;
21. console.log('UDP message sent to ' + HOST +\*:'+ PORT);
22. console.info();
23. console.info(bytes);
24. console.info();
25. client.close();
26. ))；
27. /\*\*
28. \*为UDP客户端添加一个close事件处理函数
29. \*/
30. client.on('close', function。(
31. console.logfclient disconnected\*);
32. console.info();
33. ))；

**【代码分析】**

第22~24行调用了 client.bind()方法，并在该方法的回调函数中通过调用client. setBroadcast()方法将数据报发送到广播网络中的每一台主机,关于socket.setBroadcast()方法 的语法如下：

语法：socket.setBroadcast(flag); // 发送广播数据报

socket.setBroadcast()方法用于向广播网络上发送数据报；其中，参数flag用于设置或清 除SO\_BROADCAST套接字选项；当该选项被设置后，UDP数据报就会被发送到一个本地 接口的广播地址。

第23行由于client.setBroadcast()方法的flag参数被设置为true,所以第28~35行通过 调用client.send()方法向广播网络发送数据报；第34行通过调用client.close()方法关闭客户 端。

图10.33演示的是本例程服务器端代码在Ubuntu环境下，使用WebStorm开发工具调 试输出时，刚刚初始化的结果。

然后，我们启动本例程客户端，图10.34演示的是本例程客户端代码在Ubuntu环境下, 使用WebStorm开发工具输出的结果。

从图10.34的显示结果可以看到，不仅打印输出了发送广播数据报的服务器地址和端 口的信息(UDP message sent to 255.255.255.255:12345),还打印输出了广播数据报的字节

长度（36）；

f. Run



/usr/local/bin/node chlO.udp\_broadcast\_server.]s lK)p broadcast

Now create UDO Server

**B**UDP Server listening on 0.0.0.0:12345

*X ®*

? a

图10.33服务器端初始化

» a /usr/local/bin/node chlO.udp\_broadcast\_cUent )s UOP broadcast

**S 4**

\_ Now create UDO Client.

,p UDP message sent to 255 255.255 255:12345

暮 36

X u

?篇 client disconnected

Process finished with exit code 0

图10.34客户端

下面，我们看一下服务器端发生了什么变化呢？图10.35演示的是客户端数据报成功 发送到服务器后，服务器端代码调试输出的结果的变化情况。

Run

**chio udp,broa(Jcast.5e<wr** 户

十 /usr/local/bm/node chlG. udp\_broadcast\_server. ]s

UDP broadcast

**4**

“ Nov create UDO Server..

*B*

X X ?

UDP Server listening on 0.0.0.0:12345

昏 UDP Server received fro® 192.168.G.2：3X）80

-UDP Client to Server : Hello Server!

**©**

server closed

Process finished with exit code *0*

图10.35 UDP广播服务的实现（服务器端）

如图10.35所示，服务器成功接收到客户端发来的广播数据报后，服务器端代码依次 打印输出了若干条提示信息，其中第30行打印输出了客户端的地址和端口信息"UDP Server- received from 192.168.0.2:33080”,第 31 行打印输出了广播数据报信息“UDP Client to Server : Hello Server!"。

说明：UDP “广播”与UDP “单播”的区别就是通信的IP地址不同，广播使用广播地 址255.255.255.255,将消息发送到在同一广播网络上的每个主机、另外，值得强调的是本 地广播信息是不会被路由器转的，当然这是十分容易理解的，因为如果路由器转发了广播 信息，那么势必会引起网络瘫痪，这也是为什么IP协议的制定者人为地没有定义广域网的 广播机制。

10.16模仿简单的聊天室应用

这一节，我们基于TCP协议创建一个简单的模仿聊天室的应用。网络聊天室是大家所 熟知的互联网通信应用，例如早期的门户网站都提供过聊天室的服务。但随着P2P聊天工 具的出现，聊天室已经慢慢淡出大家的视线了，不过在类似游戏大厅这样的互联网应用中，聊天室还是一项很必要的服务的。

本节这个例程是使用Node.js框架的网络（Net）模块来实现的，通过Net模块提供的 一些方法来完成关键的功能。

本例chlO.net\_chat.js主要代码如下：

01 *I\*\**

02 \* ch10.net\_chat.js

03 *\*1*

console.info(" net chat room ");

04

05

06

07

08

09

10

11

12

13

14

15

16

17

18

console.info();

var net = require('nef); var HOST = '127.0.0.1\*; var PORT = 6688; var clientList =[];

console.info('Now create Chat Server...'); console.info();

〃引入网络（Net）模块

〃定义服务器地址 〃定义端口号

〃定义客户端列表

/\*\*

\*创建TCP服务器

7

var server = net.createServer();

/\*\*

\*监听connection事件

\*/

1. server.onCconnection1, function(client) (
2. clientList.push(client); //socket 入栈
3. clientname = client.remoteAddress + + client.remotePort;
4. broadcast('hi,' + client.name + ' join in!\r\n', client);
5. client. writeChi,1 + client.name + '!\r\n');
6. /\*\*
7. \*监听data事件

〃保存客户端地址和端口

〃调用broadcast（）方法

〃向客户端发信息

1. \*/
2. client.onfdata', function(data) {
3. broadcast(client.name + ' say:1 + data + '\r\n', client); //调用 broadcast。方法
4. });
5. /\*\*
6. \*监听end事件
7. \*/
8. client.on('end', function。{
9. broadcast('hi,' + client.name + 1 quit!\r\n', client); //调用 broadcast()方法
10. clientList.splice(clientList.indexOf(client), 1); 〃删除客户端 socket
11. })；
12. })
13. /\*\*
14. \* broadcast function -向全部客户端发通知消息

* @param message

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

* @param client

\*/

function broadcast(message, client) (

var cleanup = □;

for(var i=0, len=clientList.length; i<len; i++) { if(client !== clientList[i]) ( if(clientList[i].writable) (

clientList[i].write(message); 〃向客户端发送信息

) else (

cleanup.push(clientList[i]);

clientList[i].destroy(); 〃清除客户端 socket

}

}

for(var i=0, len=cleanup.length; Hen; i++) { clientList.splice(clientList.indexOf(cleanup[i]), 1); //删除客户端 socket

) }

/\*\*

\* listen host and port

\*/

server.listen(PORT, HOST);

**【代码分析】**

第20行使用push()方法将客户端套接字保存进数组变量clientList；第21行将远程客 户端地址和端口信息保存在参数client的name属性中；第22行调用broadcast。自定义函数 发送广播消息；第23行向客户端回写数据信息，内容为第21行定义的client.name属性值;

第27~29行为参数client实例添加了数据data事件的处理函数，接收来自客户端发来的数 据信息，并通过broadcastO自定义函数向客户端发送广播消息。

第43〜58行定义了一个名称为broadcast()的自定义函数，第44行定义了一个名称为 cleanup的数组，用于保存被清除的客户端Socket；第45~54行使用了一个fbr循环用于判 断全部客户端Socket,如果不是当前活动的客户端，则向其发送广播消息；如果客户端

Socket为不可写状态，则使用Javascript语言的array.destory。方法将其删除；第55~57行 使用了另一个for循环用于判断是否为被 关闭的客户端Socket,如果是则使用 Javascript语言的array.splice()方法将其删 除。

Zusr/loca'l/bm/node chlO.net\_chat. ]s

net chat room

Nov create Chat Server...

图10.36模仿简单的聊天室应用(服务器端)

下面，我们测试一下本例程，先启动 服务器端代码，图10.36演示的是本例程 在Ubuntu环境下，服务器端代码使用 WebStorm开发工具输出的结果。

从图10.36显示的结果可以看到，服务器端启动后处于监听状态，并没有具体输出， 仅仅是打印了一行提示信息Now create Chat Server...M ；然后，我们尝试启动3个控制台 终端作为客户端，分别连接到服务器端的地址和端口号，客户端的启动方式如下：

语法:telnet 127.0.0.1 6688

第一个控制台终端客户端启动后，效果如图10.37所示。

从图10.37显示的结果可以看到，服务器端向客户端发送了一条欢迎消息，打印输出 了一行提示信息“hi, 127.0.0.1:53931!",其中127.0.0.1是客户端的地址,53931是客户端 的端口号。

然后，启动第二个控制台终端客户端，效果如图10.38所示。

king@king・ThlnkPad-X61: ~

ktng@klng-ThinkPad-X61:-5 telnet 127.6.0.1 6688

Trying 127.0.6.1...

Connected to 127.0.0.1.

Escape character is 'A]'.

hi,127.0.0.1:53931!

king^)klng-ThinkPad-X6i:-

ktng^ktng-ThinkPad-X61:telnet 127.0.0.1 6688

Trying 127.6.0.1...

Connected to 127.0.0.1.

Escape character ts 'A]1.

ht,127.0.6.1:53932!

图10.37模仿简单的聊天室应用（客户端一）

图10.38模仿简单的聊天室应用（客户端二）

从图10.38显示的结果可以看到，服务器端同样向第二个客户端发送了一条欢迎消息, 打印输出了 一行提示信息“hi, 127.0.0.1:53932!” 。

然后，我们再看看第…个客户端有什么变化呢？其效果如图10.39所示。

从图10.39显示的结果可以看到，服务器端向第一个客户端发送了一条广播消息，打 印输出了 -行提示信息“hi, 127.0.0.1:53932 join in!",其实是告诉客户端一，客户端二加 入到聊天室了。

最后，启动第三个控制台终端客户端，效果如图10.40所示。

klng®)king-ThinkPad-X61: ~

kinggktng-ThinkPad -X61:telnet 127.0.8.1 6688 Trying 127.9.0.1...

Connected to 127.6.0.1.

Escape character ts 'A]'.

hi,127.0.0.1:53931!

127.0.0.1:53932 join in!

king®>king-ThinkPad-X61:-

ktnggktng-ThinkPad -X61:telnet 127.6.0.1 6688

Trying 127.0.0.1...

Connected to 127.0.9.1.

Escape character ts 'A]1.

hi,127.6.0.1:539331

图10.39模仿简单的聊天室应用（客户端一）

图10.40模仿简单的聊天室应用（客户端三）

从图10.40显示的结果可以看到，服务器端同样向第三个客户端发送了一条欢迎消息, 打印输出了一行提示信息“hi, 127.0.0.1:53933!"。

然后，我们再看看第一个和第二个客户端有什么变化呢？其效果如图10.41和图10.42 所示。

klng(ttking-ThinkPad-X61:-

[king@ki.ng](mailto:king@ki.ng) - ThinkPad - X61: telnet 127.0.0.1 6688

Trying 127.0.0.1...

Connected to 127.0.0.1.

Escape character ts 'A]'.

hi,127.9.0.1:539311

hi,127.0.0.1:53932 join tn!

hl,127.0.6.1:53933 Join tn!

king@klng-ThinkPad-X6l: ~

ktng^king-ThinkPad-X61:~$ teinet 127.0.6.1 6688

Trying 127.6.0.1...

Connected to 127.0.0.1.

Escape character ts ' \*\*] \*.

hi,127.0.9.1:53932!

hi,127.8.0.1:53933 join In!

从图10.41和图10.42显示的结果可以看到，服务器端同时向第一个客户端和第二个客 户端发送了一条广播消息，打印输出了一行提示信息“hi, 127.0.0.1:53933 join in!”，其实 是告诉客户端一和客户端二，客户端三加入到聊天室了。

最后，我们测试一下聊天功能，在客户端三的控制台终端中，输入字符串"Hello, Node.js”并回车，如图10.43所示。

king@king-ThinkPad-X61: ~

ktng^ktng-ThinkPadtelnet 127.0.0.1 6688

Trying 127.6.0.1...

Connected to 127.6.0.1.

Escape character is 'A]'.

hi,127.0.6.1:53933!

hello node.js

图10.43模仿简单的聊天室应用（客户端三）

然后，我们再看看第一个和第二个客户端有什么变化呢？其效果如图10.44和图10.45 所示。

king（g）king-ThlnkPad-X6l: ~

Connected to 127.8.0.1.

Escape character is ，A]'. hi,127.0.0.1:53931!

ht,127.0.0.1:53932 join tn!

hi,127.0.9.1:53933 join tn I 127.0.8.1:53933 say:hello node.js

kihg@king-ThinkPad-X61:

Connected to 127.0.9.1.

Escape character is \*A]'.

hi,127.0.0.1:53932!

hi,127.0.0.1:53933 join tn! 127.0.0.1:53933 say:hello node.js

图10.44聊天室应用（客户端一）

图10.45聊天室应用（客户端二）

从图10.44和图10.45显示的结果可以看到，服务器端将客户端三发送的广播消息同时 向第一个客户端和第二个客户端转发了，客户端一和客户端二则同时打印输出了一行提示 信息“127.0.0.1:53933 say:hello node.js”,可见聊天室的基本功能已经实现了。

同样的，读者可以测试一下使用客户端一或客户端二发送广播消息的功能，与客户端 三是完全一样的。

说明：本例程仅仅是实现了一个简单的聊天室功能，例如单点对单点、单点对多点的 通信功能没有加入其中，但实现的原理是基本一样的，读者可以进一步研究,

第 11 章 Node.js 流(Stream )应用

在很多实际应用中，会涉及到大量的文件读写、上传、下载操作。Node.js开发人员遇 到此类情景时一般会比较头痛，如果文件的内容很小，文件系统(fs)模块应该能够应付 一下，而如果操作的文件很大时，使用文件系统(fs)模块就会严重影响系统性能。因为 文件系统(fs)模块每次操作都要将数据全部写入内存，然后再从内存取出，这样势必会 影响文件I/O操作的速度。

为了解决这个棘手的问题，Node.js框架为设计人员提供了一个流(Stream)模块来实 现I/O操作。流是个抽象的概念，当程序需要从某个数据源读入数据的时候，就会开启一 个数据流，数据源可以是文件、内存或网络等。相反地，需要写出数据到某个数据源目的 地的时候，也会开启一个数据流，这个数据源目的地也可以是文件、内存或网络等。

在Node.js框架中，流(Stream)是一个抽象接口，被Node.js框架中的很多对象所实 现。例如：对一个HTTP服务器的请求是一个流，stdout/stdin也是一个流，流是可读 (Readable)、可写(Writable)或双工的(Duplex,可读可写)。Node.js 框架中流(Stream) 模块操作最主要的是.pipe。方法，其可以为开发者提供可以重复使用的统一接口，通过抽象 的流(Stream)接口来控制流(Stream)之间的读写平衡。

在Node.js框架中，流(Stream)模块是一个非常重要的功能模块，之前在第10章讲 到的TCP网络编程中，TCP连接既是可读流，又是可写流，而在后续章节中将要讲到的 HTTP连接则略有不同，一个HTTP Request对象仅仅就是可读流，而一个HTTP Response 对象则仅仅是可写流。可以说，流(Stream)对象的使用已经被涵盖到整体的Node.js框架 之中了。

本章主要包括以下内容。

流(Stream)模块可读(Readable)流。

(Writable)流。 (Duplex )流。

流(Stream) 流(Stream) 流(Stream)

模块可写 模块双工 模块转移

(Transform) 流。

流(Stream)模块事件处理。

面向流消费者与面向流实现者API。

1. 创建可读(Readable )流

本章关于流(Stream)模块的第一个应用例程，我们向读者介绍创建可读(Readable)

流的基本方法。可读(Readable)流接口是对正在读取的数据的来源的抽象，或者可以理 解为数据出自一个Readable流。在程序明确就绪接收之前，可读(Readable)流并不会开 始发生数据。

在本节这个例程中，将使用文件系统模块的fs.createReadStream()方法创建一个可读 (Readable)流，同时测试一下可读(Readable)流的各个事件的基本使用方法。

本例程(代码文件名称：chll.stream\_readable\_basic.js)的主要代码如下：

01 /\*\*

02 \* ch11 .stream\_readable\_basic.js

03 \*/

04 console.info(" stream createReadStream() ");

05 console.info();

06 varfs = require('fs'); 〃引入文件系统模块

07 /\*\*

08 \*调用文件系统(fs)模块的createReadStream()方法创建可读(Readable)流

09 \*/ 〔

1. var rs = fs.createReadStreamfdata/basic.txt');
2. /\*\*
3. \* 调用 readable.setEncoding()方法设置编码
4. \*/
5. rs.setEncoding('utf8‘)；
6. /\*\*
7. \*流(Stream)模块 一readable事件处理函数
8. 7
9. rs.onfreadable', function() (
10. console.logfreadable event emitted.\*);
11. console.info();
12. })；
13. /\*\*
14. \*流(Stream )模块 —data事件处理函数
15. \*/
16. rs.on('data\ function (chunk) (
17. console.Iog('data event emitted.');
18. console.info();
19. });
20. /\*\*
21. \*流(Stream)模块 一error事件处理函数
22. \*/
23. rs.onCerror', function () (
24. console.log('error event emitted.');
25. console.info();
26. })；
27. /\*\*
28. \*流(Stream)模块 一 end事件处理函数

38 7

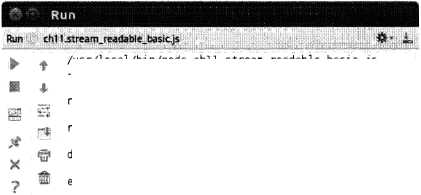
1. rs.onfend', function () (
2. console.logfend event emitted.');
3. console.info();
4. ))；
5. /\*\*
6. \*流(Stream )模块 —close事件处理函数
7. 7
8. rs.onfclose', function () (
9. console.logfclose event emitted.\*);
10. console.info();
11. })；

**【代码分析】**

第06行引入文件系统(fs)模块，同时赋于变量fs；第10行调用fs.createReadStream() 方法返回了一个可读(Readable)流对象，并将该对象保存为变量名rs；第14行调用 rs.setEncoding()方法设置使用utf8格式的编码：第18~21行为rs对象实例添加了可读 readable事件的处理函数，当一个数据块可以从流中被读出时，则会触发一个readable事件。

第25〜28行为rs对象实例添加了数据data事件的处理函数，用于读取数据块的操作； 第32〜35行为rs对象实例添加了错误error事件的处理函数，该事件在当数据接收发生错 误时被触发；第39〜42行为rs对象实例添加了结束end事件的处理函数，该事件会在没 有更多数据能够提供时被触发；第46〜49行为rs对象实例添加了关闭close事件的处理函 数，当底层数据源(例如数据源的文件描述符)被关闭时触发；注意，并不是所有流都会 触发close事件。

图11.1是本例程在Ubuntu环境下输出的结果。从结果可以看到，可读readable事件被 成功触发了两次，说明fs.createReadStream()方法操作成功完成了；同时，数据data事件、 结束end事件与关闭close事件也被成功触发了，而由于操作没有发生错误，所以错误error 事件没有被触发。



readable event emitted.

readable event emitted.

data event emitted,

end event emitted.

close event emitted.

/usr/local/bin/node chll.streafn\_readable\_basic.js stream createReadStreamO

Process finished with exit code 0

图11.1创建可读(Readable)流的基本方法

说明：可读(Readable )流有两种“模式”：流动模式和暂停模式。当处于流动模式 时，数据由底层系统读出，并尽可能快地提供给程序进行处理；当处于暂停模式时，程

序必须明确地调用stream.read()方法来取出若干数据块.可读(Readable)流默认处于暂 停模式，’

1. 测试可读readable事件处理

本节我们向读者介绍创建测试可读readable事件处理的方法。前面11.1节的例程中， 介绍了 stream.Readable类的大部分事件处理的方法，细心的读者一定注意到了可读readable 事件的输出有些特别，被触发了两次。在本节中，专门创建两个小例程，测试一下可读 readable事件处理的基本方法。

本节的第一个小例程(代码文件名称：ch 11 .stream readable readableA.js)的主要代码 如下：

01 /\*\*

02 \* ch11 .stream\_readable\_readableA.js

03 \*/

04 console.infoC1・・■― stream readable event ”)；

05 console.info();

06 varfs = require('fs'); 〃引入文件系统模块

07 /\*\*

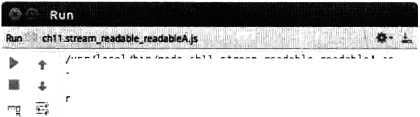
08 \*调用文件系统(fs)模块的createReadStream()方法创建可读(Readable)流

09 \*/

1. var rs = fs.createReadStream('dataZreadable.txt');
2. *I\*\**
3. \*流(Stream )模块 一readable事件处理函数
4. \*/
5. rs.onfreadable', function() (
6. console.log('readable event emitted?);
7. console.info();
8. })；

**【代码分析】**

为了测试可读readable事件处理的基本方法，我们预先在代码文件目录中新建『一个 名称为data的子目录，并在其中创建了--个名称为readable.txt的文本文件，然后通过

fs.createReadStream()方法进行操作；第14〜 17行为rs对象实例添加了可读readable事 件的处理函数，当一个数据块可以从流中被 读出时，则会触发一个readable事件。

readable event emitted.

3

Process finished with exit code 0

X

7

©

/usr/locai/bin/node chll. stream\_readabl.e\_readableA.)s stream readable event

图11.2测试readable事件处理(一)

图11.2是本例程在Ubuntu环境下输出 的结果。从输出的结果可以看到，可读 readable事件仅被成功触发了一次，说明是 fs.createRead StreamO方法操作成功时触发的。

本节的第二个小例程(代码文件名称：ch 11 .stream readable readableB.js)的主要代码 如下：

01 /\*\*

02 \* ch11 .stream\_readable\_readableB.js

03 7

04 console.info(n stream readable event ");

05 console.info();

06 varfs = require('fs'); //引入文件系统模块

07 /\*\*

08 \*调用文件系统(fs)模块的createReadStream()方法创建可读(Readable)流

09 \*/

1. var rs = fs.createReadStreamCdata/createReadStream.txf);
2. /\*\*
3. \*流(Stream)模块 一readable事件处理函数
4. \*/
5. rs.onfreadable', function() (
6. console.logfreadable event emitted.');
7. console.info();
8. ))；
9. /\*\*
10. \*流(Stream )模块 一data事件处理函数
11. \*/
12. rs.onfdata', function (chunk) (
13. console.logfdata event emitted.');
14. console.info();
15. })；
16. /\*\*
17. \*流(Stream)模块 一end事件处理函数
18. •/
19. rs.onfend', function () (
20. console.log('end event emitted.1);
21. console.info();
22. })；

**【代码分析】**

为了进行有效对比，在前一个小例程的基础上，在第21-24行中为rs对象实例添加 了数据data事件的处理函数，用于读取数据块的操作。

图11.3是本例程在Ubuntu环境下输出的结果。从结果可以看到，可读readable事件被 成功触发了两次，说明不仅是fs.createReadStream()方法的操作成功触发了可读readable事 件，同时数据data事件也成功触发了可读readable事件。

说明：当一个数据块可以从流中被读出时，会触发一个readable事件；而当内部缓冲 区被排空后，一旦发生更多数据时，readable事件会被再次触发，



/usr/local/bin/node chll.streani\_readable\_readableB. js

stream readable event

readable event emitted.

readable event emitted.

data event emitted,

end event emitted.

Process finished with exit code 0

图11.3测试readable事件处理(二)

1. 使用data事件读取文件

本节我们向读者介绍使用data事件读取文件的方法。在前面11.1节的例程中介绍了 stream.Readable类的大部分事件处理的方法，其中有关于data事件的简单处理方式。在本 节中，将专门创建一个小例程，测试一下使用data事件读取文件的基本方法。

本例程(代码文件名称：chil.stream\_readable\_data.js)的主要代码如下：

01 /\*\*

02 \* ch11 .stream\_readable\_data.js

03 \*/

04 console.infof\* stream data event ");

05 console.info();

06 varfs = require('fs'); 〃引入文件系统模块

07 /\*\*

08 \*调用文件系统(fs)模块的createReadStreamO^-法创建可读(Readable)流

09 \*/

1. var rs = fs.createReadStreamCdata/data.txf);
2. /\*\*
3. \* 调用 readable.setEncoding()方法设置编码
4. 7
5. rs.setEncodingCutfS')；
6. /\*\*
7. \*流(Stream)模块 一readable事件处理函数
8. \*/
9. rs.on('readable', function() (
10. console.logt'readable event emitted.');
11. consoleJnfoQ;
12. })；
13. /\*\*
14. \*流(Stream)模块一data事件处理函数
15. \*/
16. rs.on('data', function (chunk) {
17. console.info("读取'data/data.txt'文本文件的内容...”)；
18. console.info();
19. console.logf读取到了 %d 字节的数据chunk.length);
20. console.info();
21. console.logC读取到的数据:chunk);
22. console.info();
23. });

//……省略部分事件，参考第11.1节各事件的代码

**【代码分析】**

第14行调用rs.setEncoding()方法设置使用utfS格式的编码；第18〜21行为rs对象实 例添加了可读readable事件的处理函数；第25〜32行为rs对象实例添加了数据data事件 的处理函数，该事件处理函数中的回调函数包含一个参数chunk,该参数为一个Buffer或 String类型的数据块，用于保存data事件读取到的文本文件data.txt的内容；第28行通过 参数属性chunk.length打印输出了数据长度，第30行通过参数chunk打印输出了数据内容。

图11.4是本例程在Ubuntu环境下输出的结果。从结果可以看到，通过数据data事件 成功读取到了文本文件data.txt的内容。

Run

**ftm chiiqream\_reada\*jiadj5** 音• £.

A 。 /usr/local/bm/node chll.strea«\_readable\_data. js  
 stream data event

**■ i**

readable event emitted.

r\* readable event emitted.

^^.'data/data.txt'.

7 质学2 了 28 .

学均敷鼻：Streasi - Readable Data Event

end event esitted.

close event eaitted.

Process finished with exit code 0

I

图11.4使用data事件读取文件

11.4可读流错误事件处理

本节我们向读者介绍可读流错误error事件处理方法。在前面11.1节的例程中介绍了 stream.Readable类的大部分事件处理的方法，其中有关于error事件的简单处理方式。在本 节中，将专门创建一个小例程，测试一下在读取文件时发生错误的处理方法。

本例程(代码文件名称：chil.stream\_readable\_error.js)的主要代码如下：

01 "\*

02 \* ch11.stream\_readable\_error.js

03 *\*1*

04 console.info(" stream error event ");

05 console.info();

06 varfs = requireffs\*); 〃引入文件系统模块

07 /\*\*

08 \*调用文件系统(fs)模块的createReadStream()方法创建可读(Readable)流

09 \*/

1. var rs = fs.createReadStreamC'data/error.txf);
2. /\*\*
3. \*流(Stream)模块 一readable事件处理函数
4. \*/
5. rs.onfreadable\*, function() {
6. console.log('readable event emitted.');
7. console.info();
8. ))；
9. /\*\*
10. \*流(Stream)模块 一error事件处理函数
11. 7
12. rs.on('error', function ()(
13. console.Iog('error event emitted.');
14. console.info();
15. ))；

**【代码分析】**

第21-24行为rs对象实例添加了错误error事件的处理函数，并打印输出了错误提示 信息。

图11.5是本例程在Ubuntu环境下输出的结果。从结果可以看到，第10行在尝试读取 实际不存在的文本文件error.txt时，错误error事件被成功触发了，并打印输出了错误提示 信息 (error event emitted.)。



error event emitted,

+ /usr/tocal/bin/node chil.strea«\_readable\_error.js stream error event

Process finished with exit code 0

图11.5可读流错误事件处理方法

11.5暂停与恢复可读（Readable）流

本节我们向读者介绍暂停与恢复可读（Readable）流的方法。暂停方法会使一个处于 流动模式的流停止触发data事件，并切换到非流动模式，并且让后续可用数据留在内部缓 冲区中。在本节中，创建一个简单的应用例程，看一下如何实现暂停与恢复可读（Readable）流的应用。

本例程（代码文件名称：chil.stream\_readable\_pause.js）的主要代码如下:

01 /\*\*

02 \* ch11 .stream\_readable\_pause.js

03 \*/

04 console.info(" stream pause() & resume() ——~H);

05 console.info();

06 varfs = require('fs'); //引入文件系统模块

07 /\*\*

08 \*调用文件系统(fs)模块的createReadStream()方法创建可读(Readable)流 09 7

1. var rs = fs.createReadStream('data/pause.txt,);
2. /\*\*
3. \* 调用 readable.setEncoding()方法设置编码
4. \*/
5. rs.setEncoding('utf8');
6. /\*\*
7. \*流(Stream)模块 一 readable事件处理函数
8. \*/
9. rs.onfreadable', function() (
10. console.log('readable event emitted.');
11. console.info();
12. })；
13. /\*\*
14. \*流(Stream )模块 ——data事件处理函数
15. 7

rs.on('data\*, function (chunk) ( console.infof 读取'data/pause.txt'文本文件的内容...")； console.info();

25

26

*27*

28

29

30

31

32

33

34

35

36

37

38

39

40

//••

console.logf读取到了 %d 字节的数据/, chunk.length); console.info();

rs.pause();

console.logf接下来3秒内不会有数据...')； console.info();

setTimeout(function() (

console.logf现在数据会再次开始流动…，)；

console.info();

rs.resume();

console.logf读取到的数据:\n', chunk); console.info();

}, 3000);

｝）；

••省略部分事件，参考第11.1节各事件的代码

**【代码分析】**

第30行调用rs.pause()方法暂停了可读(Readable)流，rs.pause()的语法如下：

语法：rs.pause(); // 暂停一个可读(Readable)流

rs.pause()方法会使一个处于流动模式的流停止触发data事件，切换到非流动模式，并 让后续可用数据留在内部缓冲区中。

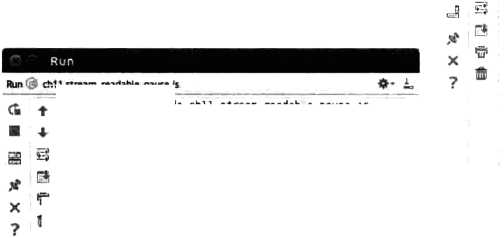
第33行调用setTimeout(涵数设置了一个3000毫秒的时间延迟；第36行调用rs.resume() 方法恢复了可读(Readable)流，rs.resume()的语法如下：

语法：rs.resume。； *II* 恢复一个可读(Readable)流

rs.resume()方法让一个可读流可以继续触发data事件。

图11.6与图11.7是本例程在Ubuntu环境下输出的结果。

Run



readable event emitted.

readable event emitted.

暴下表\*内不会有ficl\* ...

end event eaitted.

冒 学取’data/pause.txt'文本文伴的内\* .. 血就孕剁了 40字节的薮拷.

/usr/local/bm/node chll.streaa\_readable\_pause.js streai pause() & resuaeO

**chi > Mream\_rejdabte\_pause js**

close event emitted.

图11.6方法(一)

**Run ch1 iMneam.reBdriMejnusejs S %**

A a /usr/local/bin/node chll.streai\_readable\_pause )s

strean pause() & re$uae()

readable event emitted.

readable event emitted.

学塞’data/pause.txt'文本文许的力M.

式欧到T 40字罕的轰择.

挂•下表3®方壬会本薮拷 .

end event eaitted.

close event emitted.

现&敷择会再次升必(渔功...

淡到的敦捧：

/\*•

, chll.stream readable\_pause.js

•/

readable event enitted.

Process finished with exit code 0

图11.7方法(一)

从图11.6输出的结果可以看到，在第28行打印输出了数据长度后，提示信息显示“接 下来3秒内不会有数据...”，说明第30行执行暂停可读(Readable)流与data事件的操作 成功了，之后结束end事件和关闭close事件依次被触发执行，然后第33行设置的时间延 迟终止了屏幕输出。

从图11.7输出的结果可以看到，时间延时过后屏幕输出恢复了，第36行恢复了可读 (Readable)流与data事件，接着第37行打印输出了数据内容，并再次触发了可读readable 事件。

说明：暂停可读(Readable)流方法会使一个处于流动模式的流停止触发data事件， 切换到非流动模式，并让后续可用数据留在内部缓冲区中；恢复可读(Readable )流方法 会将流切换到流动模式，但如果不想从流中消费数据，却想得到它的end事件，此时可以 调用rs.resume()方法来启动数据流。

1. 使用可读(Readable )流发送数据

本节我们向读者介绍使用可读(Readable)流发送数据的方法。在前面的几个例程中 基本都是使用文件模块的createReadStream()方法创建可读(Readable)流。在本节中，将 使用流(Stream)模块的Readable类创建可读(Readable)流，并测试一，下使用可读(Readable) 流发送数据的基本方法。

本例程(代码文件名称：chi l.stream\_readable\_pipe\_push.js)的主要代码如下：

|  |  |  |
| --- | --- | --- |
| 01 | /\*\* |  |
| 02 | \* ch11 ,stream\_readable\_pipe\_push.js | |
| 03 | \*/ |  |
| 04 | console.info(H stream readable pipe push "); | |
| 05 | console.info(); |  |
| 06 | var stream = requireCstream1); | //引入流(Stream)模块 |
| 07 | var rs = new stream.Readable; | •//定义Readable对象 |
| 08 | rs.push(\*Stream'); | 〃使用push()方法发送数据 |
| 09 | rs.pushfReadable | 〃使用push()方法发送数据 |
| 10 | rs.push('Push()'); | 〃使用push。方法发送数据 |
| 11 | rs.push(\*Pipe()'); | 〃使用push()方法发送数据 |
| 12 | rs.push('\n'); | 〃使用push()方法发送数据 |
| 13 | rs.push(null); | 〃使用push()方法通知发送数据完毕 |
| 14 | rs.pipe(process.stdout); | 〃使用pipe。方法导出数据 |

**【代码分析】**

第06行引入流(stream)模块，同时赋于变量stream；第07行通过流(stream)模块 的Readable类创建可读(Readable)流对象，同时赋于变量rs；第08〜12行分几次调用rs.push() 方法向数据接收方发送数据(Stream> Readable、Push。、Pipe()和山)，其中流(stream) 模块的push()方法用于将字符串或Buffer类型数据发送给接收方;第13行使用rs.push(null) 方法通知接收方数据发送完毕；第14行调用rs.pipe。方法导出数据，并ifiii process.stdout 对象进行打印输出。

图11.8是本例程在Ubuntu环境下输出的结果。从结果可以看到，通过多次调用push() 方法发送的数据后，再次通过pipe()方法成功导出了。

Run

**Run** cM1 **5tream\_rea<lable\_plpe\_pu5h.j5 !** 泰，二

粉 蕾 /usr/locai/bm/node chll.stream\_readable\_pipe\_push. js

寥 stream readable pipe push

■ 务

• Stream Readable Push() PipeO

g Process finished with exit code 0

说明：我们注意到在将所有数据内容压入可读流之前并没有调用rs.pipe(process.stdout) 方法，但是压入的所有数据内容还是完全地输出了，这是因为可读流在接收者没有读取数 据之前，会缓存所有压入的数据，

1. 使用可读(Readable )流读取文件

本节我们向读者介绍使用可读(Readable)流读取文件的方法。与前面11.3节的例程 中介绍的通过data事件的方法不同，在本节这个例程中，将测试-下使用可读(Readable) 流读取文件的基本方法。

本例程(代码文件名称：ch 11 .stream readable read.js)的主要代码如下：

01 /\*\*

02 \* ch11 .stream\_readable\_read.js

03 \*/

04 cx)nsole.info(" stream read() ");

05 console.info();

06 varfs = require('fs\*)； 〃引入文件系统模块

07 L

08 \*调用文件系统(fs)模块的createReadStream()方法创建可读(Readable )流

09 \*/

1. var rs = fs.createReadStreamCdata/read.txf);
2. /\*\*
3. \* 调用 readable.setEncoding()方法设置编码
4. \*/
5. rs.setEncodingCutfS\*);
6. /\*\*
7. \*流(Stream)模块 一readable事件处理函数
8. \*/
9. rs.on('readable\ function() (
10. console.log('readable event emitted?);
11. console.info();
12. var chunk;
13. while (null !== (chunk = rs.read())) (
14. consoie.logC得到了 %d 字节的数chunk.length);
15. console.info();
16. console.logf读取到的数据:\ chunk);
17. console.info();
18. )
19. })；
20. rs.on('encf, function () (
21. console.logfend event emitted.');
22. console.info();
23. })；

//……省略部分事件，参考第11.1节各事件的代码

**【代码分析】**

第22~27行通过一个while循环判断语句，使用rs.read()方法读取文本文件(read.txt) 内容，rs.read。的语法如下：

语法：rs.read([size]); *II*返回一个新的可读(Readable)流对象

rs.read()方法用于从内部缓冲区中读取并返回若干数据；其中，可选的size参数用于定 义读取数据的大小：这里需要注意的是，若使用了 size参数，那么它会返回相当字节的数 据；若没有指定size参数，那么它会返回内部缓冲区中的所有数据；而当size字节不可用 时，则返回null。

从图11.9的结果可以看到，通过数据data事件成功读取到了 data.txt的内容。

Run

Run

A 学 /usr/local/bm/node chll. strea»\_readable\_read. js streal read()

B $ readable event eaitted.

匡耳

口 等到了 24字节的敦择

x 1

冒 ■到竹数搏：St ream - Readable readO

X w

? 费 end event e«itted.

close event emtted.

Process finished with exit code 0

图11.9使用可读(Readable)流读取文件

说明：rs.read()方法仅在暂停模式时被调用。在流动模式中，该方法会被自动调用直到 内部缓冲区排空，

11.8使用pipe方法实现复制文件

本节我们向读者介绍使用pipe方法实现复制文件的应用。在前面的第8章中，介绍了 基于文件系统(File System)模块的读写文件的应用，本节将介绍基于流(Stream)模块的 pipe。方法的复制文件的应用，同时还借用了第8章的一个读文件的例程。

本例程(代码文件名称：chll.stream\_readable\_pipe.js)的主要代码如下：

01 /\*\*

02 \* ch11 .stream\_readable\_pipe.js

03 \*/

04 console.info(" stream pipe write file ");

05 console.info();

06 varfs = require(\*fs'); //引入文件系统模块

07 /\*\*

08 \*调用文件系统(fs)模块的createReadStream()方法创建可读(Readable)流

09 \*/

1. var readable = fs.createReadStream('data/srcFile.txt')；
2. /\*\*
3. \*调用文件系统(fs)模块的createWriteStream()方法创建可写(Writable)流
4. 7
5. var writable = fs.createWriteStream('data/destFile.txt');
6. /\*\*
7. \*所有来自readable的数据会被写入M'data/destFile.txt'文本文件中
8. \*/
9. readable.pipe(writable);

【代码分析】

第 10 行调用 fs.createReadStream()方法返回了源文件 srcFile.txt 的一个可读(Readable) 流对象，并将该对象保存为变量名readable；第14行调用fs.createWriteStream()方法返回了 目标文件destFile.txt的一个可写(Writable)流对象，并将该对象保存为变量名writable； 第18行调用pipe。方法将源文件的内容写入到目标文件之中。

图11.10演示的是本例程在Ubuntu环境下，使用控制台调试输出的结果，在调试过程 中借用了 8.14节读文件的例程(脚本文件名称为fs readFileSync.js)。



图11.10使用pipe方法实现复制文件

从图11.10输出的结果可以看到，首先读取目标文件destFile.txt的内容时为空，然后 我们调用本例程脚本文件(文件名称为chll.stream\_readable\_pipe.js)复制文件，当再次读 取目标文件destFile.txt的内容时，控制台打印输出了复制到的源文件的内容(Stream - pipe -srcfile.txt)。

说明：在使用pipe。方法复制文件的过程中，自动调用了 data与end事件

11.9解除pipe方法设置的流

本节我们向读者介绍解除pipe方法设置的流的方法。在上一节介绍了基于流(Stream) 模块的pipe()方法的复制文件的应用，这-节介绍一下使用impipeO方法解除pipe。方法设 置的流的应用。

本例程(代码文件名称：chi l.stream\_readable\_unpipe.js)的主要代码如下：

01 /\*\*

02 \* ch11 .stream\_readable\_unpipe.js

03 7

04 console.info(n stream pipe write file ");

05 console.info();

06 varfs = require(,fs,)； 〃引入文件系统模块

07 /\*\*

08 \*调用文件系统(fs)模块的createReadStream()方法创建可读(Readable)流

09 \*/

1. var readable = fs.createReadStream('data/srcUnFile.txt');
2. /\*\*
3. \*调用文件系统(fs)模块的createWriteStream()方法创建可写(Writable)流
4. 7
5. var writable = fs.createWriteStreamCdata/destUnFile.txt');
6. /\*\*
7. \*所有来自readable的数据会被写入到'data/destUnFile.txt'文本文件中
8. \*/
9. readable.pipe(writable);
10. /\*\*
11. \*通过setTimeout。方法设置1毫秒时间延迟
12. •并通过unpipe()方法解除pipe。方法设置的流
13. \*最后自行关闭文件流
14. \*/ 、
15. setTimeout(function() (
16. console.log(\*停止写入到 data/destFile.txt');
17. console.info();
18. readable.unpipe(writable); //解除设置的流
19. console.logf自行关闭文件流')；
20. console.info();
21. writable.end(); //关闭文件流
22. },1)；

**【代码分析】**

第18行调用pipe。方法将源文件的内容写入到目标文件之中；第24〜31行调用setTimeout() 函数设置了一个1毫秒的时间延迟；第27行调用readable.unpipe()方法解除pipe。方法设置 的流；第30行调用writable.end()方法关闭文件流。

图11.11是本例程在Ubuntu环境下使用控制台输出的结果。从输出的结果可以看到， 首先读取目标文件destFile.txt的内容时为空，然后我们调用本例程脚本文件（文件名称为 chi 1 ,stream\_readable\_u叩ipe.js）复制文件，但由于调用unpipe。方法解除了 pipe。方法设置 的流，所以再次读取目标文件destUnFile.txt的内容时仍为空。

**ing-Tlj|nkPac-XC 1: ~/p(oject$/NodeJ5/Chdpter ,**

**ktng^king-ThinkPad-X61: - /project£/HodeJS/Chapterll/data$ node fs\_readFileSync.js**  fs readFlleSyncO

**read destrile.txt contents:**

L 目标文件内容为空

**r**、. **I ead**「**lltfSym.() Done.**

**ktngfking-ThtnkPad-X61:~/projects/ModeJS/Chapterll/data$ cd ..**

**ktng^klng-ThinkPad-X61 ->/projects/NodeJS/Chapterll$ node chll.streaH\_readable\_un** pipe.js

**pipe write** 作**no —**

停 **1L**与入到 **da:a/destunFlle.txt**

自行关闭文**4**流

king^ktng-ThinkPad-X61:-/projects/NodeJS；ChapterlIS cd data/

king (I Id ng-TMnkPad-X61: ~/projerts/tttxi\*1S/Chapter11 /data$ node f jroadFilNyn。. jw fs readFtlesync()

**read destFile.txt contents:**

I I J 目标文件内容为空

**fs.read£ileSync() Done.** ktnfl0klng-ThtnkPad-X61:-7projects/NodeJS/Chapterll/dataS ,

图11.Il使用pipe方法实现复制文件(•)

但如果我们将通过setTimeout（）方法设置的时间延迟增加到1000毫秒时，结果会怎么 样呢？

图11.12演示的是本例程在时间延迟增加到1000毫秒时，使用控制台输出的结果。从 结果可以看到，由于设置了时间延退为3000毫秒，所以再次读取目标文件destUnFile.txt 的内容时，打印输出了读到的源文件的内容。

**raoteiklng-ThlnkPacl-XSI' /hcmey'king/projcctc/NodeJS/Chaptem/data**

root0klnc-ThtnkPadX61:/ho^e/ktnq/proiects/NodeJ5/CtaDterll/data# node fs readFt leSync.js

**fs rcadFtlcSynct)**

read destUnFile.txt contents:

fs.readfileSync(' Dene.

tootaking-ThinkPad-xeii/hone/king/projects/bodeJS/ChaDterll/datoW cd ..

rootaking-IhtnkPad-xoi:/hone/ktng/proJecTs/NodeJS/Lrapterii»j node chii.s:rean\_re edable\_urptpe.js

**stream pipe write file**

停止写入到 data/destunnie. txt

自行关闭女件流

root^ktng-ThtnkPad-X61:/ho'ie/ktng/projects/NodeJS/Cbapterll» cd data/

rootfikinc-ThinkPad-XGlz/ho^e/king/orojects/NodeJS/Ztaoterll/data# node fs\_readFt, **leSync.js**

**fs readFileSync()**

**r«»ad dKtUnFtldtxt content\*;:**

SI edii - pipe - :HiWFlle.lxl 目标文代的内容

St\_ean - pipe ♦ destunFtle.txz

**fs.rcadfilcSync(； Done.**

rootaktnc•ThinkPad-X61:/hone/ktnq/orojects/NodeJS/Chaoter11/data# |

说明：unpipe。方法会解除之前调用pipe()所设定的流；如果不指定目标，所有导流都 会被解除；如果指定了目标，但并没有与之建立导流，则什么事都不会发生。

1. 使用可写(Writable )流写文件

本节我们向读者介绍使用可写(Writable)流写文件的方法。在前面的几个例程中，介 绍了几种写文件或复制文件的方法，本节将通过流(Stream)模块可写(Writable)流的 Write。方法执行写文件的操作。

本例程(代码文件名称：chi l.stream\_writable\_file.js)的主要代码如下：

01 /\*\*

02 \* ch11 .stream\_writable\_file.js

03 \*/

04 console.info(H stream writable file ");

05 console.info();

06 var fs = require('fs'); //引入文件系统模块

07 /\*\*

08 \*调用文件系统(fs) createWriteStream。方法创建可写(Writable)流

09 7

1. var ws = fs.createWriteStreamCdata/writable.txt')；
2. ws.write('stream -'); //写数据
3. ws.write('writable -'); //写数据
4. ws.writeCfile -'); //写数据
5. /\*\*
6. \*通过setTimeout()方法设置1000ms时间延迟
7. \*自行关闭文件流
8. \*/
9. setTimeout(functior)() (
10. ws.end('end.\n\*); 〃关闭文件流
11. }, 1000);

【代码分析】

第11〜13行通过多次调用ws.write()方法向目标文件writable.txt中写入数据，关于 ws. write。方法的语法如下：

语法：writable.write(chunk, [encoding], [callback]); // 向底层系统写入数据

ws.write()方法用于执行向底层系统写入数据的操作；其中，chunk参数用于表示数据; 可选的encoding参数用于定义字符编码；可选的callback参数用于定义回调函数；当数据 已被全部处理后，该方法返回true。

图11.13演示的是本例程在Ubuntu环境下，使用控制台调试输出的结果，在调试过程 中借用了 8.14节读文件的例程(脚本文件名称为fs readFileSync.js)。

**kingljktng-ThinkPad-X01;~/project5/Mode3S/Chapterll/data$ node fs\_readFileSync.js**

fs readFtleSyncC)

**i eod wi ildble. Lx I «.un Lentb；**

fs. eddrtleSjfiicl) Done.

**king^king-ThinkPad-X61:-/projpcts/MndelS/Chapterl1/data$ cd -.**

**kinggktng-ThinkPad-X61:- /projects/Mode3S/Chapterll$ node chll.stresH\_v»ritable\_fi le.**

stream writable file

**kinggking- ThinkPad-X61:■/projects/McdeJS/Chapterll? cd da~a/**

**kliiy^kLiiy ■ ThinkPad-X61:-/pi oJeLls/MudeJS/ChapLei 11/ddLdS node F**i\_t **tddFileSynt.. js fs rparfFilpSynrO**

**i edd wi lldble. LxL LUiileuL2>.**

**strME - urltabl**。- **Filo - end. V** ，一 写入的数据

fs.-eadftleSync() Done.

**ktnggklng ThtnkPod X61:-/projcct5/Mod2S/Chapterll/dat3$**，

图11.13使用可写（Writable）流写文件

从图11.13输出的结果可以看到，首先读取目标文件writable.txt的内容时为空，然后 我们调用本例程脚本文件（文件名称为chll.stream\_writable\_file.js）执行写文件的操作，当 再次读取目标文件writable.txt的内容时，控制台打印输出了写入的内容（stream- writable - file ・ end.）。

说明：ws.write（）方法向底层系统写入数据，并在数据被处理完毕后调用其回调函数； 其返回值表明是否应该立即继续写入，如果数据需要滞留在内部，则其会返回false,否则 返.回true。

1. 可写*（*Writable ）流drain事件处理

本节我们向读者介绍可写（Writable）流drain事件处理方法。在11.8节的例程中，介 绍了一种复制文件的方法，但是这个例程有个小问题，那就是如果写入速度跟不上读取速 度的话，可写（Writable）流内部的缓存会溢出。本节将通过可写（Writable）流的drain 事件实现另一种复制文件的操作。

本例程（代码文件名称：chil.stream\_writable\_drain.js）的主要代码如下：

01 /\*\*

02 \* ch11.stream\_writable\_finish.js

03 \*/

04 console.infoC\*—- stream writable file -—"）;

05 console.info（）;

06 varfs = requireCfs\*）; 〃引入文件系统模块

07 /\*\*

08 \*调用文件系统（fs）模块的createReadStream（）方法创建可读（Readable）流

09 \*/

10 var rs = fs.createReadStream（,,data/srcDrain.mp3M）;

**第** 11 **章** Node.js **流(**Stream )**应用楼** 259 /\*\*

\*调用文件系统(fs)模块的createWriteStream()^法创建可写(Writable)流

\*/

var ws = fs.createWriteStreamC'data/destDrain.mpS")；

*/\*\**

\*调用readable.setEncoding()方法设置编码

\*/

rs.setEncoding('utf8');

/\*\*

\*流(Stream)模块 一data事件处理函数

\*/

rs.on('data', function(chunk) {

console.log('data event emitted.');

console.info();

/\*\*

\*通过ws.write()方法返回值判断

\*数据是已经写入目标

\*还是仅仅写入缓存中

\*/ ,

if(ws.write(chunk) === false) (

console.logf数据写入缓存...')；

console.info();

rs.pause(); 〃暂停可读(Readable )流

}

))；

/\*\*

\*流(Stream)模块—drain事件处理函数

7

ws.on('drain', function。{

console.log('drain event emitted.');

console.info();

rs.resume(); 〃恢复可读(Readable)流

})；

广

\*流(Stream )模块一end事件处理函数

\*/

rs.on('end', function。{

console.log('end event emitted?);

console.info();

ws.end();

【代码分析】

第30〜34行通过if条件判断语句，来判断ws.write()方法将缓存区数据写入目标文件 的操作是否完成，如果没有完成，则调用rs.pauseQ方法暂停可读(Readable)流，并触发

可写(Writable)流的drain事件，这种方式可以有效地避免缓存溢出问题。

第39〜43行为ws对象实例添加了 drain事件的处理函数，该事件用于向流写入更多 的数据：第42行通过调用rs.resume()方法恢复可读(Readable)流；第47~51行为rs对 象实例添加了结束end事件的处理函数，第50行调用ws.end()方法关闭了文件流。

图11.14是本例程在Ubuntu环境下输出的结果。从输出的结果可以看到，可与 (Writable)流drain事件被多次调用(提示信息为“数据写入缓存...”、“drain event emitted"), 这是因为源MP3文件较大，在使用ws.writeO方法写入时，分若干次从缓存中取出数据写 入目标文件。图11.14显示的仅仅是刚刚开始时的提示信息，实际输出了很长一段提示信 息，读者可以实际测试一下，或者更换更大的视频文件进行测试。

r Run

**Rin ch11 Mream\_wntatte\_<lrain.js** 豪”£,

A 个 /usr/local/bin/node chll.streaB\_wntable\_drain. js

strean writable file 一

■ 外

— data event emitted.

X 昏 dram event emitted.

? 壶 data event emitted.

drain event emitted.

data event emitted.

drain event emitted.

data event emtted

图11.14 可写（Writable）流drain事件处理方法

说明：如果在调用writable.write(chunk)方法后返回false,则表明drain事件可以继续向 流写入更多的数据，通过对drain事件的合理使用，可以有效地避免缓存溢出的情况

1. 可写(Writable )流finish事件处理

本节我们向读者介绍可写(Writable)流finish事件处理方法，在可写(Writable)流 调用ws.endO方法后，并且所有数据已被写入到底层系统后，该事件会被触发。本节将在 上一节例程的基础上，向读者演示一下可写(Writable)流finish事件处理的方法。

本例程(代码文件名称：chi l.stream\_writable\_finish.js)的主要代码如下：

01 /\*\*

02 \* ch11 .stream\_writable\_finish.js

03 \*/

04 console.info(" stream writable file ");

05 console.info();

06 varfs = require('fs'); *II*引入文件系统模块

/\*\*

\*调用文件系统(fs)模块的createReadStream()方法创建可读(Readable)流

7

var rs = fs.createReadStreamC'data/srcFinish.txt");

/\*\*

\*调用文件系统(fs)模块的createWriteStream()方法创建可写(Writable)流 \*/

var ws = fs.createWriteStreamC'data/destFinish.txt");

/\*\*

\*调用readable.setEncoding()方法设置编码

7

rs.setEncoding('utf8');

/\*\*

\*流(Stream)模块 一 data事件处理函数

\*/

rs.on('data\ function(chunk) (

console.logfdata event emitted.1);

console. info();

/\*\*

\*通过ws.write()方法返回值判断

\*数据是已经写入目标

\*还是仅仅写入缓存中

7

if(ws.write(chunk) === false) {

console.logf数据写入缓存...')； console.info();

rs.pause(); 〃暂停可读(Readable)流

}

})；

/\*\*

\*流(Stream)模块 一drain事件处理函数

7

ws.on('drain', function。{

console.logfdrain event emitted.');

console.info();

rs.resume(); 〃恢复可读(Readable)流

})；

/\*\*

\*流(Stream)模块— finish事件处理函数

\*/

ws.onffinish', function() (

console.logCfinish event emitted.');

console.info();

})；

/\*\*

1. \*流(Stream)模块 —end事件处理函数
2. \*/
3. rs.on('end', function() {
4. console.logfend event emitted.');
5. console.info();
6. ws.end();
7. })； "

【代码分析】

第47〜50行ws对象实例添加了结束finish事件的处理函数，并打印输出了提示信息； 可写（Writable）流finish事件是当ws.endQ方法被调用、且所有数据已被写入到底层系统 后被触发;第54〜58行为rs对象实例添加了结束end事件的处理函数，第57行调用ws.end（） 方法关闭了文件流。

图11.15是本例程在Ubuntu环境下输出的结果。从结果可以看到，在调用ws.end（）方 法后，可写（Writable）流finish事件被触发（提示信息为finish event emittedw ）；另外, 读者一定注意到，本节例程与上一节例程类似，为什么可写（Writable）流drain事件没有 被触发呢？这是因为源文本文件srcFinish.txt很小，通过缓存区一次性就被写入目标文件 destFinish.txt 了。



data event emitted.

/usr/local/bin/node chll.streaiD\_writable\_finish. js stream writable file

偶 end event emitted.

@ finish event emitted.

**X**寸：

7 ®

• u Process finished with exit code G

图11.15 可写（Writable）流finish事件处理方法

第12章 Node.js Web开发

本章我们向读者介绍应用Node.js框架进行Web开发的内容，主要是基于超文本传输 协议(HTTP, HyperText Transfer Protocol)和安全套接字层超文本传输协议(HTTPS, Hyper Text Transfer Protocol over Secure Socket Layer)如何实现 Web 应用开发。因此在阅读本章 内容之前，读者应该学习了解一下HTTP与HTTPS协议的基本知识，便于加深对本章例程 的理解。

Node.js框架为设计人员提供了 HTTP模块与HTTPS模块来实现Web应用，这两个模 块基于HTTP协议与HTTPS协议开发，提供了一系列与Web应用开发相关的方法，通过 这些方法就可以构建各种功能的Web应用。

本章主要包括以下内容。

.HTTP模块方法与应用。

* HTTPS模块方法与应用。
* Exoress开发框架应用。

1. 构建一个基本的HTTP服务器

首先，我们从构建一个基本的HTTP服务器开始，HTTP是构建于TCP/IP协议上的最 基本的互联网通信协议。HTTP协议的应用十分广泛，很多互联网应用服务器(例如门户 网站、社交网络、电子商城等)主要都是应用HTTP协议开发的，可见HTTP服务器的功 能是多么强大。Node.js框架提供了一个HTTP模块来支持HTTP协议，通过HTTP模块的 http.createServer()方法来完成创建HTTP服务器的功能。

在本节这个基本的例程中，将使用http.createServer。方法来创建基本的HTTP服务器, 本例 ch 10.http createServer basic.js 主要代码如下：

01 /\*\*

02 \* ch12.http\_createServer\_basic.js

03 \*/

04 console.info(" http - create basic server ");

05 console.info();

06 var http = require('http'); 〃引入 http 模块

07 /\*\*

08 \*调用http.createServer。方法创建服务器

09 \*/

1. http.createServer(function(req, res) {
2. 〃\*
3. \*通过res.writeHeader()方法写HTTP文件头
4. \*/
5. res.writeHead(200, ('Content-type': 'text/html'});
6. /\*\*
7. \*通过res.write()方法写页面内容
8. \*/
9. res.write('<h3>Node.js 一 HTTP</h3>,);
10. /\*\*
11. \*通过res.end()方法发送响应状态码，并通知服务器消息完成
12. \*/
13. res.endC<p>Create Basic HTTP Server!</p>');
14. )).listen(6868); 〃监听 6868 端口号

**【代码分析】**

第06行引入HTTP模块，同时赋于变量http；第10〜23行通过调用http.createServer() 方法创建了一个基本的HTTP服务器，关于http.createServeir()方法的语法如下：

语法：http.createServer([requestListener]); // 创建 HTTP 服务器

http.createServer()方法用于创建一个HTTP服务器，并将参数requestListener作为request 事件的监听函数；可选的参数requestListener是一个请求处理函数，自动添加到request事 件，函数传递两个参数：第一个req参数用于请求对象，包含…些常用属性；第二个res参 数用于响应对象，当收到请求后要做出响应。

第14行通过res.writeHead()方法写HTTP文件头，关于res.writeHeader()方法的语法如 下：

语法：response.writeHead(statusCode, [reasonPhrase], [headers]); // 向请求回复响应头

res.writeHeader()方法用于向请求回复响应头；第一个参数statusCode是一个三位数的 HTTP状态码，例如404等；可选的第二个参数reasonPhrase用于表示原因短句；可选的第 三个参数headers用于定义响应头的内容。

第18行通过调用res.write()方法写页面内容，关于res.write()方法的语法如下：

语法：response.write(chunk, [encoding]); // 发送一个响应体的数据块

res.writeO方法用于发送一个响应体的数据块；第一个参数chunk可以是字符串或者缓 存；如果第一个参数chunk是一个字符串，可选的第二个参数encoding用于表示如何将这 个字符串编码为一个比特流，默认的编码是utf8格式。

第22行通过调用res.end()方法发送页面内容，并通知HTTP服务器消息完成，关于 res.end()方法的语法如下：

语法：response.end([data], [encoding]);

*II*发送一个响应体的数据块

res.end()方法是一个每次响应完成之后必须调用的方法；如果指定了第一个参数data, 就相当于先调用response.write(data, encoding)方法再调用response.end()方法；可选的第二 个参数encoding用于表示如何将这个字符串编码为一个比特流，默认的编码是utf8格式； 当所有的响应报头和报文被发送完成时，res.end。方法将信号发送给服务器，服务器会认为 这个消息完成了。

第23行通过调用server.listen()方法监听端口，关于server.listen()方法的语法如下：

语法：server.listen(port, [hostname], [backlog], [callback]);// 开始在指定的主机名和端口接收连接

server.listen。方法用于开始在指定的主机名和端口接收连接；第一个参数port是端口 号，本例程端口为6868；可选的第二个参数hostname用于表示主机名，本例程主机名为本 机地址；可选的第三个参数backlog用于积压量，为连接等待队列的最大长度：这个函数 是异步的，可选的第四个参数callback会被作为事件监听器添加到listening事件中。

图12.1是本例程在Ubuntu环境下输出的结果。从中可以看到，我们创建的HTTP服 务器已经启动运行，不过没有任何屏幕输出；然后，我们打开浏览器，并在地址栏中输入 以下地址：http:〃127.0.0.1:6868,如图 12.2 所示。

图12.1创建基本的HTTP服务器

Lj <http://127.0.0.1:6868/> X

**Node.js — HTTP**

Create Basic HTTP Server!

localho成(127Q0.1)峻

图12.2创建基本的HTTP服务器

从图12.2中可以看到，我们访问到刚刚创建的服务器后，得到了服务器第18行与第 22行的打印输出信息。

说明：当第一次调用response.write()方法时，将会发送缓存的header信息和第一个报 文给客户端；当第二次调用response.write ()方法时，Node.js框架假设用户将发送数据流， 然后分别地进行发送；这意味着响应是缓存到第一次报文的数据块。如果response.write() 方法的所有数据被成功刷新到内核缓冲区，则返回true;如果所有或部分数据在用户内存 里还处于队列中，则返回false；当缓冲区再次被释放时，drain事件会被分发。

1. 编写一个简单的HTTP客户端

上一节我们介绍了如何创建基本的HTTP服务器，这一节将介绍如何编写一个简单的 HTTP客户端，有了客户端就可以访问服务器了。其实，在12.1节测试HTTP服务器时所 使用的浏览器就是HTTP客户端，通过在浏览器地址栏输入服务器地址就可以访问服务器 资源了。在这一节中，将通过编程的方法实现一个客户端，用来访问服务器资源。

.在Node.js框架中，主要是通过HTTP模块的http.request()方法来完成完成HTTP客尸 端的功能。本例chl2.http\_request\_basic.js主要代码如下：

01 /\*\*

|  |  |  |
| --- | --- | --- |
| 02 | \* ch12.http\_request\_basic.js |  |
| 03 | 7 |  |
| 04 | console.info(M http - create basic client ”)； | |
| 05 | console.info(); |  |
| 06 | var http = require('http'); | //引入http模块 |
| 07 | /\*\* |  |
| 08 | \*定义服务器参数字段 |  |
| 09 | \* @type ((hostname: string, port: number, path: string, method: string}} | |
| 10 | \*/ |  |
| 11 | var options = ( |  |
| 12 | hostname: 'localhost', | //定义服务器主机地址 |
| 13 | port: 6868, | 〃定义服务器主机端口号 |
| 14 | path: 7', | 〃定义服务器路径 |
| 15 | method: 'POST | //定义服务器访问方式i |
| 16 | ｝； |  |
| 17 | /\*\* |  |
| 18 | \* 通过 http.request()方法 |  |
| 19 | \*由客户端向HTTP服务器发起请求 | |
| 20 | 7 |  |

1. var req = http.request(options, function(res) (
2. console.log('STATUS: \* + res.statusCode);
3. console.log(\*HEADERS: \* + JSON.stringify(res.headers));
4. res.setEncodingfutf8\*);
5. res.on('data'» function (chunk) (
6. console.log('BODY:' + chunk);
7. })；
8. })；
9. /\*\*
10. \*监听request对象的'error'事件
11. \*/
12. req-onCerror1, function(e) (
13. console.logCproblem with request:' + e.message);
14. ))；
15. /\*\*
16. \* write data to request body
17. \*/
18. req.writeCdata\n')；
19. /\*\*
20. \* write end to request body
21. \*/
22. req.end();

**【代码分析】**

第11〜16行定义了一个JSON数组对象(options)，该对象内包含了若干HTTP服务 器信息，分别包括服务器主机地址(localhost)、端口(6868)、路径(/)、访问方式(POST) 等。

第21~28行通过调用http.request()方法创建了一个简单的HTTP客户端，并将返回的 http.ClientRequest实例赋于变量req,关于http.request()方法的语法如下：

语法：http.request(options, callback); *II* 创建 HTTP 客户端连接

http.request()方法用于创建HTTP客户端连接并向HTTP服务器发起请求；第一个参数 options为一个JSON数组对象，用于定义HTTP服务器主机地址、端口号、路径、请求访 问方式、请求头以及身份验证等信息；第二个参数callback用于定义一个回调函数，其包 含一个http.ClientResponse实例类型参数(在本例程中定义为参数res)；更详细的说明读 者可以参考Node.js框架官方文档中关于http.request()方法的说明。

为了测试HTTP客户端连接请求，我们参考12.1节的内容，编写一个简单的HTTP服 务器端代码(代码名称：chl2.http\_response\_basic.js),其主要代码如下：

01 /\*\*

02 \* ch12.http\_response\_basic.js

03 7

04 console.info(M—— http - create basic server ");

05 console.info();

06 var http = require('http'); //引入 http 模块

07 〃调用http.createServer()方法创建服务器

1. http.createServer(function(req, res) (
2. //通过 res.writeHeader()方法写 HTTP 文件头
3. res.writeHead(200, fContent-type\*: 'text/html1));
4. 〃通过res.write()方法写页面内容
5. res.writeC<h3>Node.js 一 HTTP</h3>\*);
6. 〃通过res.end ()方法发送响应状态码，并通知服务器消息完成
7. res.endf<p>Create Basic HTTP Server Response to clients request!</p>\*);
8. )).listen(6868); 〃监听 6868 端口号

读者阅读以上代码可以发现，ch 12.http\_response\_basic.js代码的功能与前面12.1节中 chl2.http\_createServer\_basic.js代码的内容基本类似，实现的功能也基本一致，在此就不进 一步详细阐述了。

为了测试本例程，先启动chl2.http\_response\_basic.js服务器代码。图12.3所示是本例 程在Ubuntu环境下输出的结果。

然后再启动ch 12.http\_request\_basic.js客户端代码。图12.4所示是本例程在Ubuntu环 境下输出的结果。

说明：请求路径"path"相对于根的路径，默认值是7二而且QueryString也应该包含 在其中(例如：/index.html?name=king):

'就广Run

/usr/local/bin/node chl2.http\_response\_basj.c. ]S

http - create basic server

*L.*

/usr/local/bin/node chl2 http\_request\_basic；js

http - create basic client

"I

STATUS: 2G0

HEADERS: {"content-type": "text/htitil", "date" "Tue. 16 Jun 2015 06.46.28

BODY: <h3>Node. ]S -- - Hn?«/h3>

BODY： vp乂reate Basic HTTP Server Response to clients request I</p\*

Process finished with exit code 0

图12.3服务器端

图12.4客户端

12.3 HTTP响应状态码

这一节我们介绍一下HTTP响应状态码，所谓HTTP响应状态码其实就是服务器返回 给客户端的响应状态信息。大体上，HTTP响应状态码有正确和错误两种状态，用于表示 服务器返回状态。由于HTTP响应状态码可以反映出HTTP服务器与客户端的状态，因此 其作用还是非常重要的。

在Node.js框架中，HTTP模块的http.STATUS\_CODES属性用来描述全部的HTTP响 应状态码。本节将通过一个简单的例程，测试一下如何得到我们需要的HTTP响应状态码。

本例程客户端的主要代码（脚本文件名称为chl2.http\_request\_statuscodes.js）如下所示。

01

02

03

04

05

y\*\*

\* ch 12.http\_request\_\_statuscodes.js

7

console.infoC\* http - create basic client

console.info();

——”)；

06 var http = requireChttp1); 07 08 09

10

11

12

13

14

15

16

17

18

19

20

21

22

23

var querystring = requireCquerystring\*); /\*\*

\*定义响应状态码数组

\*/

var status\_codes = new Array(); status\_codes[0] = "201"; status\_codes[1] = "202"; status\_codes[2] = "203"; status\_codes[3] = ”401”； status\_codes[4] = "402";

/\*\*

\*定义查询字段

\*/

var postData = new Array(); for(var n=0; n<5; n++) (

postData[n] = querystring.stringify(( statuscodes: status\_codes[n]

//引入http模块

〃引入Query String模块



})；

)

/\*\*

\*模拟HTTP客户端向HTTP服务器端连续发送request请求

\*/

for(var i=0; i<5; i++) (

/\*\*

\*定义服务器参数字段

* @type ((hostname: string, port: number, path: string, method: string, headers: (Content-Type: string, Content-Length: [[8]](#footnote-9)}})

\*/

var options = {

hostname: 'localhost',

port: 6868,

path:'/' + postData[i],

method: POST,

headers: {

'Content-Type': 'application/x-www-form-urlencoded\*,

'Content-Length1: postData.length

}

)；

/\*\*

* 通过 http.request()方法

\*由客户端向HTTP服务器发起请求

\*/

var req = http.request(options, function(res) (

console.logCSTATUS\_CODES:' + res.statusCode);

console.logfHEADERS:1 + JSON.stringify(res.headers));

console.info();

res.setEncodingCutfS1);

res.onfdata', function (chunk) (

console.log('BODY:' + chunk);

console.info();

})；

})；

\*监听request对象的'error'事件

\*/

req.on('error', function(e) (

console.Iog('problem with request:' + ©.message);

console.info();

})；

/\*\*

1. req.write("\n");
2. /\*\*
3. \* write end to request body
4. \*/
5. req.end();
6. }

【代码分析】

第11〜16行定义了一个数组对象(status\_codes),用于定义一组HTTP响应状态码； 第20〜25行定义了另一个数组对象(变量名称为postData)，用于定义一组查询字段：其 中，第22〜24行通过调用querystring.stringify()方法进行序列化查询字段的操作；第29~ 73行使用一个for循环语句，模拟HTTP客户端向HTTP服务器端连续发送request请求； 第48〜57行通过调用http.request()方法创建了一个简单的HTTP*客户端,*并将返回的 [http.C1](http://http.C1) ientRequest 实例赋于变量 req。

为了测试HTTP客户端连接请求，我们参考12.1节的内容，编写一个简单的HTTP服 务器端代码(代码名称：ch 12.http\_response\_statuscodes.js),其主要代码如下：

01 /\*\*

02 \* ch12.http\_response\_statuscodes.js

03 7

04 console.infoC-— http STSTUS\_CODES ——”)；

05 console.info();

06 var http = require('http'); //引入 http 模块

07 console.log("Now start HTTP server...");

08 console.info();

09 /\*\*

1. \*调用http.createServer()方法创建服务器
2. 7
3. http.createServer(function(req, res) {
4. var status = req.url.substr(l); //获取 url 查询字段
5. var status\_codes = status.substring(12); 〃获取 HTTP.STATUS\_CODES
6. 〃判断http.STATUS\_CODES响应状态码集合是否有效
7. if(!http.STATUS\_CODES[status\_codes])(
8. status\_codes = '404';
9. )
10. //通过 res.writeHeader()方法写 HTTP 文件头
11. res.writeHeader(statuscodes, ('Content-Type'/text/plain\*});
12. //通过res.end()方法发送响应状态码，并通知服务器消息完成
13. res.end(http.STATUS\_CODES[status\_code 司)；
14. )).listen(6868); 〃监听 6868 端口号

读者阅读以上代码可以发现，chl2.http\_response\_statuscodes.js代码的功能与前面12.1 节中ch 10.http createServer basic.js代码的内容基本类似，实现的功能也基本一致；不同之 处为第13行和第14行，分别提取了 url查询字段与HTTP响应状态码(HTTP.STSTUS\_

CODES) o

为了测试本例程，我们先启动chl2.http\_response\_statuscodes.js服务器代码。图12.5是 本例程在Ubuntu环境下输出的结果。然后再启动chl2.http\_request\_statuscodes.js客户端代

码。图12.6是本例程在Ubuntu环境下输出的结果。



STATUS\_CODES： 201

HEADEFS： {"content-type-:\*text/plain","date":"Wed, 17 Jun 2015

chi Z.ratp\_re5pcM®e^st»tuscoOM.ji

/usr/local/bm/node chl2 http\_request\_statuscodes. ]s http - create basic client



BODY： Created

BXJY: accepted

202

ntent-type"："text/plain","date":\*Wed. 17 Jun 2015

STATUS COOES： 203

HEADERS Ccontent-type-:'text/plain-.-date\* -Wed. *17* Jun 2015

BODY- Non-Authoritative Information

Mow start HTTP server .

/usr/local/bm/node chl2.http\_respon$e\_statuscodes. js

http STSTUS\_CODES ~ '

STATUS\_CODES： 401

l-EACERS. {"content-type": •text/plain". ■'date': 'Wed. 17 Jun 2015

BODY： unauthorized

STATUS\_CODES 402

HEADER： ("content-type"；"text/plain"."date"："Wed. 17 Jun 2015

BODY： Payment Required

Process finished with exit code 0

图12.5 HTTP响应状态码(服务S端)

图12.6 HTTP响应状态码(客户端)

从图12.6中可以看到，客户端连续几次的连接请求均被服务器成功处理并返回，HTTP 响应状态码信息及其对应的功能描述也被依次打印输出。

12.4设定和获取HTTP头文件

这一节我们介绍一下设定和获取HTTP头义件的方法，所谓HTTP头文件其实就是包 含」'一系列用于控制服务器与客户端的信息，这些控制信息大体包括通用头信息、请求头 信息、响应头信息和实体头信息这四个部分。HTTP头文件在实际应用中是很重要的，开 发人员可以根据头文件包含的控制信息，获取浏览器类型、字符集、编码方式、语言、主 机地址与端口、授权信息、正文长度、Cookie等非常重要的信息或内容。

在 Node.js 框架中，HTTP 模块的 response.getHeader()1 j response.setHeader()方法用来 设定和获取HTTP头文件控制信息。本节将通过一个简单的例程，测试-•下设定和获取 HTTP头文件的方法。

本例 ch 12.http\_response\_header.js 主要代码如下:

01 /\*\*

02 \* ch12.http\_response\_header.js

03 \*/

04 console.infof http - create basic server ");

05 console.info();

06 var http = require('http\*); 〃引入 http 模块

07 /\*\*

08 \*调用http.createServer()方法创建服务器

09 7

1. http.createServer(function(req, res) {
2. /\*\*
3. \*通过res.setHeader()方法设定HTTP文件头
4. \*/
5. res.setHeader("Content-Type", "text/html");
6. res.setHeader("Set-Cookie", [,,type=king", "language=javascriptH]);
7. /•\*
8. \*通过res.getHeader()方法获取HTTP文件头
9. 7
10. var content\_Type = res.getHeader('Content-Type');
11. console.info(content\_Type)；
12. var set\_cookie = res.getHeader('Set-Cookie');
13. console.info(set\_cookie);
14. *I\*\**
15. \*通过res.write()方法写页面内容
16. *7*
17. res.writeC<h3>Node.js — HTTPv/h3>');
18. res.write('<p>, + content\_Type + \*</p>');
19. res.write(\*<p>' + set\_cookie + '</p>');
20. /\*\*
21. \*通过res.end()方法发送响应状态码，并通知服务器消息完成
22. \*/
23. res.end();
24. }).listen(6868); 〃监听 6868 端口号

**【代码分析】**

第14-15行通过res.setHeader。方法设定了 HTTP文件头信息：第19〜22行通过 res.getHeader()方法获取了 HTTP文件头信息；第27〜28行通过调用res.write()方法向客户 端写了 Content-Type和Set-Cookie文件头属性的内容；第32行通过调用res.end。方法发送 页面内容，并通知HTTP服务器消息完成；当所有的响应报头和报文被发送完成肘res.end() 方法将信号发送给服务器，服务器会认为这个消息完成了；第33行通过调用server.listen() 方法在指定的主机名和端口接收连接，并监听该端口的连接请求。

我们先启动chl2.http\_response\_header.js服务器代码。图12.7所示是本例程在Ubuntu 环境下输出的结果。然后打开浏览器，并在地址栏中输入以下地址：http:〃127.0.0.1:6868, 如图12.8所示。

Run

Run chi ZJttitresponse.headecjs (1)

/usr/local/bin/node ch!2 http\_response\_header.)s http - create basic server

eg

图12.7获取HTTP头文件(服务器端)

<http://localhost:6868/> X 令

**Node.js —HTTP**

tcxlhtml

typc=king. language=ja vascripi

localhost (127Q0.1) O

图12.8获取HTTP头文件(客户端)

从图12.8中可以看到，客户端浏览器成功连接到服务器后，得到了服务器第27行与 第28行通过res.write()方法写到客户端的HTTP头文件信息。

下面再回过头看一下服务器端的变化，如图12.9所示。

Run

cM2,http\_respcnse\_hea<ter.j5 (1} 境H

C 唐 /usr/local/bin/node chl2.http\_response\_header.js  
 http - create basic server

H 4 text/html

四 W [ 'type=king', 'language=javascript']

pr I

图12.9设定和获取HTTP头文件的方法(服务器端)

从图12.9中可以看到，客户端浏览器成功连接到服务器后，服务器第19〜22行打印 输出了通过res.getHeader。方法获取的HTTP文件头信息，与第14~15行通过res.setHeader。 方法设定的HTTP文件头信息是一致的。

说明：response.getHeader。*方*法用于读取一个在队列中但是还没有被发送至客户端的 header；但需要注意的是name参数是不区分大小写的，而且该方法只能在header还没被清 除掉之前调用

1. 写HTTP头文件的方法

上一节我们介绍『设定和获取HTTP头文件的方法，这一节将介绍写HTTP头文件的 方法，通过写HTTP头文件可以自定义属性值。

在Node.js框架中，HTTP模块的response.writeHead。方法用来实现写HTTP头文件的 操作。本节将通过一个简单的例程，测试一下写自定义HTTP头文件的方法。

本例程HTTP服务器端chl2.http\_response\_writeheader.js主要代码如下：

01 /\*\*

02 \* ch12.http\_response\_writeheader.js

03 \*/

04 console.info(M http - server write header \*')；

05 console.info();

06 var http = require('http'); //引入 http 模块

07 console.log(MNow start HTTP server...");

08 console.info();

09 /\*\*

1. \*调用http.createServer。方法创建服务器
2. \*/
3. http.createServer(function(req, res) (
4. /\*\*
5. \*通过res.writeHead()方法写HTTP文件头
6. \*/
7. var body = 'write header1;
8. res.writeHead(200, {
9. 'Content-Length\*: body.length,
10. 'Content-Type': 'text/plairf
11. ))；
12. /\*\*
13. \*通过res.write()方法写页面内容
14. \*/
15. res.write("Node.js");
16. res.write(MHTTPH);
17. /\*\*
18. \*通过res.end()方法发送响应状态码，并通知服务器消息完成
19. \*/
20. res.end(); ° 旅
21. )).listen(6868); 〃监听 6868 端口号

【代码分析】

第16行定义了一个字符串变量body；第17〜20行通过res.writeHead()方法写入了 HTTP 文件头信息，其中Content-Length属性值为字符串变量body的长度，Content-Type属性值 为text/plain；第24〜25行通过调用res.write()方法向客户端写了 Node.js和HTTP字符串信 息；第29行通过调用res.end()方法通知HTTP服务器消息完成；第30行通过调用 server.listen()方法在指定的主机名和端口接收连接,并监听该端口的连接请求。

本例程 HTTP 客户端 ch 12.http\_request\_writeheader.js 主要代码如下：

01 /\*\*

02 \* ch12.http\_request\_writeheader.js

03 \*/

04 console.infoC-— http - client write header ");

05 console.info();

06 var http = require(\*http'); //引入 http 模块

07 /\*\*

08 \*定义服务器参数字段

09 \* @type ({hostname: string, port: number, path: string, method: string})

10 \*/

var options = { hostname: localhost', port: 6868, path: 7\*, method: 'POST'

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

\* 通过 http.request()方法

//定义服务器主机地址 〃定义服务器主机端口号

〃定义服务器路径

//定义服务器访问方式i

\*由客户端向HTTP服务器发起请求

\*/

var req = http.request(options, function(res) ( console.log('STATUS:1 + res.statusCode); console.logfHEADERS:' + JSON.stringify(res.headers)); res.setEncoding('utf8');

res.onfdata', function (chunk) ( console.log('BODY:' + chunk);

})； })；

//监听request对象的，error\*事件 req.onCerror\*, function(e) {

console.log('problem with request:' + ©.message);

})； req.write('data\n'); req.end();

以上HTTP客户端ch 12.http\_request\_writeheader.js代码的功能与前面12.2节中 ch 12.http\_request\_basic.js代码的内容基本类似，实现的功能也基本一致，在此就不进一步 详细阐述了。

为了测试本例程，我们先启动ch 12.http\_response\_writeheader.js服务器代码。图12.10 是本例程在Ubuntu环境下输出的结果。

然后再启动ch 12.http\_request\_writeheader.js客户端代码。图12.11所示是本例程在 Ubuntu环境下输出的结果。

宅

\_ Now start HTTP server.., a曲

G a /usr/local/bin/node chl2.http\_response\_vriteheader.js http - server write header

蠲3

STATUS: 2G0

HEADERS： {"content-length":12", "content-type": "text/plam\ BODY： Node.js

*BODY:* HTTP

/usr/local/bin/node chl2.http\_request\_wnteheader. ]s http - client write header

It

图12.10写HTTP头文件的方法(服务器端)

图12.11写HTTP头文件的方法(客户端)

说明：res.writeHead()方法只能在当前请求中使用一次，并且必须在res.end()之前调用 如果在调用这之前调用了 res.write()或者response.end()方法，就会调用这个函数。

1. 发送与处理GET请求

前面几节我们介绍了 POST请求方式的处理方法，这一节将介绍如何发送与处理GET 请求，很多情况下GET请求方式也是常用到的。关于GET与POST请求方式的异同之处, 读者可以参考相关文档，这方面的介绍还是很多的。在Node.js框架中，主要是通过HTTP 模块的http.get()方法来完成HTTP客户端GET请求方式的操作。

本例程客户端ch 12.http request get.js主要代码如下：

01 /\*\*

02 \* ch12.http\_request\_get.js

03 7

04 console.info(H—— http - client get ”)；

05 console.info();

06 var http = require('http\*); 〃引入 http 模块

07 /\*\*

08 \*发送HTTP GET请求

09 7

1. http.get("<http://localhost:6868/signature=12345678&echostr=78787878&timestamp=168>",
2. /\*\*
3. \*GET回调函数
4. \* @param res
5. \*/
6. function(res) (
7. console.logfSTATUS:1 + res.statusCode);//打印输出 Status\_Codes 响应状态码
8. console.info();
9. res.setEncoding('utf8');
10. /\*\*
11. \*监听”data”事件处理函数
12. 7
13. res.on('data', function (chunk) (
14. console.log('BODY:，+ chunk); 〃打印输出服务器回写内容
15. console.info();

26 })；

1. console.info();
2. }).on('error\*,

28

1. \* "error"事件回调函数
2. \* @param e
3. \*/
4. function(e) (
5. console.log(HGot error:" + e.message); 〃打印输出"error” 信息
6. console.info();
7. ))；

**【代码分析】**

第10〜35行通过调用http.get()方法在HTTP客户端发送GET请求方式的连接，关于 http.get()方法的语法如下:

语法：http.get(options, callback);

//发送GET请求方式的连接

http.get()方法用于在HTTP客户端向HTTP服务器端发送GET请求方式的连接；第一 个参数options为一个JSON数组对象，用于定义HTTP服务器主机地址、端口号、路径、 查询请求字段等信息；第二个参数callback用于定义一个回调函数，其包含一个 http.ClientResponse实例类型参数(在本例程中定义为参数res)；更详细的说明读者可以 参考Node.js框架官方文档中关于http.get()方法的说明。

为了测试HTTP客户端连接请求，我们参考12.2节的内容，编写一个简单的HTTP服 务器端代码(代码名称：ch 12.http response get.js),其主要代码如下：

01 /\*\*

02 \* ch12.http\_response\_get.js

03 \*/

04

05

06

07

08

http - server get

console.info(" http - ser

console.info();

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

var qs = require('querystring');

〃引入http模块

//引入url模块

//引入 querystring 模块

09 /\*\*

1. \*调用http.createServer()方法创建服务器
2. 7
3. http.createServer(function(req, res) {
4. /\*\*
5. \*通过res.writeHeader()方法写HTTP文件头
6. 7
7. res.writeHead(200, {'Content-type\*: 'text/plain1});
8. /\*\*
9. \*通过url.parse()方法获取查询字段
10. \*/
11. var query = url.parse(req.url).query;
12. console.info(query);
13. console.info();
14. /\*\*
15. \*通过res.end()方法发送响应状态码，并通知服务器消息完成
16. \*/
17. var qs\_parse = qs.parse(query);
18. console.info(qs\_parse);
19. console.info();
20. res.end(JSON.stringify(qs\_parse));
21. }).listen(6868); 〃监听 6868 端口号

**【代码分析】**

第29行通过调用res.end。方法将序列化后的变量qs\_parse内容写到客户端，并通知 HTTP服务器消息完成；第30行通过调用server.listen()方法在指定的主机名和端口接收连 接，并监听该端口的连接请求。

为了测试本例程，我们先启动chl2.http\_response\_get.js服务器代码。图12.12所示是本 例程在Ubuntu环境下输出的结果。然后再启动ch 12.http\_request\_get.js客户端代码。图12.3 所示是本例程在Ubuntu环境下输出的结果。

A a. /usr/local/bin/node chl2.htTp\_request\_qet.]S

http , client get

■，

1 ― STATUS; 200

图12.12如何发送与处理GET请求(服务器端)

*yt* 1— BODY： {-signature':\*12345678".'echostr\**:"78797878".*•timestamp". \*168"}

X '

» 愈 Process finished with exit cod? 0

图12.13如何发送与处理GET请求(客户端)

从图12.13中可以看到，客户端第16行打印输出了 HTTP响应状态码信息(STSTUS: 200)；客户端第23行打印输出了服务器端写到客户端页面的数据信息，该信息就是客户 端通过http.get()方法发送到服务器端的被序列化处理后的查询请求字段。

最后，再回头看一下服务器端输出状态有没有变化。图12.14是本例程在Ubuntu环境 下输出的结果。

Run

**Otl2-tfttp.r?5poree\_£ET.js**

G 号 /usr/tocal/bin/node chl2.http\_response\_get.js  
 http - server get

■ a signature=12345678&echostr-78787878&tinestaip=168

囹玲

*四* { signature： '12345678', echostr： 78787878'. txaestanp： '168' }

J &

图12.14如何发送与处理GET请求（服务器端）

从图12.14中可以看到，服务器端第21行打印输出了客户端发来的查询请求字段；服 务器端第27行打印输出了格式化后的查询请求字段。这里如果读者细心的话会发现，通过 qs.parse()方法与JSON.stringify()^j法处理查询字段后的结果会略有不同。

说明：http.get()方法是为了满足没有报文体的GET请求而提供的这种便捷的方法 该方 法与http.request()的唯一区别是它设置的是GET方法会自动调用req.end()方法进行处理的“

12.7进行重定向操作

这一节我们介绍如何进行重定向操作的方法，这项功能在页面跳转过程中是常用到的 操作。在Node.js框架中，可以通过HTTP模块的res.writeHead。方法，在写HTTP头文件 的操作中实现该功能。

在本节这个简单的重定向例程中，将通过控制HTTP头文件实现从一个服务器页面跳 转到另一个服务器页面的功能。

本例程第一个服务器端脚本文件(文件名称：chl2.http\_createServer\_redirectA.js)的主 要代码如下：

01 /\*\*

02 \* ch 12.http\_createServer\_redirectA.js

03 \*/

04 console.info(" http - create redirect server ―--");

05 console.info();

06 var http = requireChttp'); //引入 http 模块

07 console.log(HNow start HTTP server on port 6868...");

08 console.info();

09 /\*\*

1. \*调用http.createServer()方法创建服务器
2. \*/
3. http.createServer(function(req, res) (
4. /\*\*
5. \*通过res.writeHeader()方法写HTTP文件头
6. \*/
7. res.writeHead(301, {
8. 'Location1: '<http://localhost:8686/'>
9. })；
10. /\*\*
11. \*通过res.write。方法写页面内容
12. 7
13. res.write('<h3>Node.js — HTTP</h3>');
14. /\*\*
15. \*通过res.end()方法发送响应状态码，并通知服务器消息完成
16. \*/
17. res.end('<p>Create Redirect HTTP Server on Port 6868!</p>');
18. )).listen(6868); 〃监听 6868 端口号

**【代码分析】**

第16〜18行通过res.writeHeadO方法写入了 HTTP文件头信息，其中[http.STATUS](http://http.STATUS) CODES 响应状态码值为301 (该值代表的含义为“Moved Permanently”，翻译过来可以理解为“永 久跳转”)，Location属性值为http://localhost:8686,该字符串为另一个HTTP服务器的地 址；第22行通过调用res.write()方法向客户端写了 “Node.js--HTTP”字符串信息；第26 行通过调用res.end()方法向客户端写了一段提示信息，并通知HTTP服务器消息完成；第 27行通过调用server.listen()方法在指定的主机名和端口接收连接，并监听该端口的连接 请求。

ch 12.http\_createServer\_redirectA.js服务器脚本的第17行定义了另一个HTTP服务器地 址，为了测试重定向操作，我们再根据该地址编写一个简单的HTTP服务器端代码(代码 名称：ch 12.http createServer redirectB.js),其主要代码如下：

01 /\*\*

02 \* ch12.http\_createServer\_redirectA.js

03 7

04 console.info(M http - create redirect server **”)；**

05 console.info();

06 var http = require('http) 〃引入 http 模块

07 console.log(MNow start HTTP server on port 8686...”)；

08 console.info();

09 /\*\*

1. \*调用http.createServer()方法创建服务器
2. 7
3. http.createServer(function(req, res)(
4. //通过 res.writeHeader()方法写 HTTP 文件头
5. res.writeHead(200, {'Content-type\*: 'text/html\*});
6. 〃通过res,write()方法写页面内容
7. res.writeC<h3>Node.js — HTTP</h3>,);
8. //通过res.end()方法发送响应状态码，并通知服务器消息完成
9. res.end(\*<p>Create Redirect HTTP Server on Port 8686!v/p>\*
10. )).listen(8686); 〃监听 8686 端口号

**【代码分析】**

该服务器端脚本代码与第一个服务器端脚本代码内容基本类似，主要区别就是服务器 监听的端口变更为8686,同时没有定义重定向代码；为了测试重定向操作的结果，我们先 依次启动 chl2.http\_createServer\_redirectA.js 与 ch 12.http createServer redirectB,js 这两个服 务器代码。

**图**12.15与图12.16是本例程在Ubuntu环境下输出的结果。

Run \*' Run

|  |  |  |  |
| --- | --- | --- | --- |
| 5] |  | Run: | 通\_曲**5**好**igffoA**月 **MSKiEESSSiBS \*\* -** |
| *n* | , /usr/local/bm/node chl2.http createServer redirectA. js | **G** | , Zusr/local/bm/node chl2.http createServer redirects, js |
|  | http - create redirect server |  | http - create redirect server |
| . | **4** | 占 | . |
|  | Nov start HTTP server on port 6858 .. |  | Now start HTTP server on port 8686 |
|  | 目 | g |  |
|  | **a** | **X** | r? |
| X | 守 | X | 冒 |
| ? |  | ? | ft |

图12.15进行重定向操作的方法（服务器A）

图12.16进行电定向操作的方法（服务器B）

从图12.15与图12.16输出的结果可以看到，两个HTTP服务器己经启动运行，但监听 的端口不一样；然后，我们打开浏览器，并在地址栏中输入以下地址：<http://localhost:6868>, 访问第一个服务器，如图12.17所示。从图中可以看到，我们明明访问的是第一个服务器

（端口号为6868），但页面实际已经重定向到了第二个服务器（端口号8686）,从页面得 到的打印输出信息可以验证确实进行了重定向操作。

说明：重定向操作的方法有很多种，本节实现的这种重定向操作比较适用于在页面刚 刚初始化、还没有全部解析出来时需要实现重定向功能的情况。

|j <http://localhost:8686/> X，唾

**Node.js— HTTP**

Create Redirect HTTP Server on Port 8686!

tocalhost(127.0.0；1) O

图12.17进行重定向操作的方法（浏览器）

12.8服务器多路径处理方式

这一节我们介绍服务器多路径处理方式，这项功能在服务器处理客户端请求方式时非 常实用*。*在Node.js框架中，需要通过处理req.url参数的pathname属性字段来实现该功能。

本例程服务器端脚本文件(文件名称：chl2.http\_createServer\_pathname.js)的主要代码 如下：

01 /\*\*

02 \* ch12.http\_createServer\_pathname.js

03 7

04 console.infof http - server pathname —M)；

05 console.info();

06 var http = require('http'); 〃引入 http 模块

07 var url = require('uri'); 〃引入 url 模块

08 console.logf'Now start HTTP server on port 6868...M);

09 console.info();

1. /\*\*
2. \*调用http.createServer()方法创建服务器
3. \*/
4. http.createServer(function(req, res) (
5. /\*\*
6. \* 获取 url.pathname 路径
7. \* @type(pathname|\*|req.pathname|parseTests.pathname|parseTestsWithQueryString}
8. 7
9. var pathname = url.parse(req.url).pathname;
10. /\*\*
11. \* Responding to multi type of request
12. 7
13. if (pathname ===== 7\*) {
14. /\*\*
15. \*通过res.writeHead()*方法写*HTTP文件头

\*/

res.writeHead(200, {

'Content-Type': 'text/plain'

})；

/\*\*

•通过res.write。方法写页面内容

7

res.write('Node.js — HTTP\n');

/\*\*

\*通过res.end()方法发送响应状态码，并通知服务器消息完成

\*/

res.endCHome Page\n')

} else if (pathname === VabouV) (

尸\*

\*通过res.writeHead()方法写HTTP文件头

7

res.writeHead(200, (

'Content-Type': 'text/plain'

})；

/\*\*

\*通过res.write()方法写页面内容

7

res.write('Node.js — HTTP\n');

/\*\*

•通过res.end()方法发送响应状态码，并通知服务器消息完成

7

res.end('About Us\n\*)

} else if (pathname === '/redirect') (

/\*\*

•通过res.writeHead()^法写HTTP文件头

7

res.writeHead(301, {

'Location': *T*

})；

/\*\*

\*通过res.write。方法写页面内容

7

res.write(\*Node.js — HTTP\n');

/\*\*

\*通过res.end()方法发送响应状态码，并通知服务器消息完成

\*/

res.end();

} else (

/\*\*

\*通过res.writeHead。方法写HTTP文件头

1. \*/
2. res.writeHead(404J (
3. 'Content-Type': 'text/plain'
4. ))；
5. /\*\*
6. \*通过res.write。方法写页面内容
7. \*/
8. res.writeCNode.js — HTTP\n');
9. /\*\*
10. \*通过res.end()方法发送响应状态码，并通知服务器消息完成
11. 7
12. res.endfPage not found\n\*)
13. )
14. )).listen(6868); 〃监听 6868 端口号

**【代码分析】**

第18行先通过url.parse()方法对req.url参数进行解析，然后获取了其查询路径名称 pathname属性，并保存在变量pathname中；第22〜82行通过if条件选择判断语句，分别 处理了三种客户端向服务器端的请求方式(分别为："/”、"/about”与“/redirect” )。

对于这三种客户端请求方式，均先通过调用res.writeHead()方法对HTTP头文件进行了 写操作，然后使用res.write()方法向客户端写入信息，最后通过调用res.end()方法向客户端 写了一段提示信息，并通知HTTP服务器消息完成；第83行通过调用server.listen()方法在 指定的主机名和端口接收连接，并监听该端口的连接请求。

下面，我们先启动chl2.http\_createServer\_pathname.js服务器代码。图12.18所示是本 例程在Ubuntu环境下输出的结果。

然后打开浏览器，并在地址栏中输入 以下地址：[http://Iocalhost:6868/,访问服](http://Iocalhost:6868/,%e8%ae%bf%e9%97%ae%e6%9c%8d) 务器根路径，如图12.19所示。从中可以 看到，浏览器页面打印输出了服务器端第 32行与第36行写给客户端的信息(Home Page) o然后，我们在地址栏中输入以下 地址：[http://localhost:6868/about,得到的](http://localhost:6868/about,%e5%be%97%e5%88%b0%e7%9a%84) 结果如图12.20所示。

**cM2.http\_createServer\_pathname-j$** 籍-**i,**

蓍 /usr/local/bin/node chl2.http\_createServer\_pathname.js http - server pathname

. ■ Nov start HTTP server on port 6868... 耳

图12.18服务器多路径处理方式(服务器)

Ld <http://localhost:6868/> X 吵

曲 <http://localhost:6868/about> M 勒

Node.js HTTP

About Us

Node.js HTTP

Hon；e Page

localhost (12 700.1)

图12.19服务器多路径处理方式(浏览器)

图12.20服务器多路径处理方式(浏览器)

从图12.20中可以看到，浏览器页面打印输出了服务器端第47行与第51行写给客户 端的信息(About Us):再次在地址栏中输入以下地址：[http://localhost:6868/redirect,重定](http://localhost:6868/redirect,%e9%87%8d%e5%ae%9a)

向地址的结果如图12.21所示。从图12.21中可以看到，浏览器页面重定向操作后返回到服 务器主页面，结果与图12.19完全一致；

最后，我们在地址栏中输入以下地址：[http://localhost:6868/nopage,访问不存在地址的](http://localhost:6868/nopage,%e8%ae%bf%e9%97%ae%e4%b8%8d%e5%ad%98%e5%9c%a8%e5%9c%b0%e5%9d%80%e7%9a%84) 结果如图12.22所示。

Lj <http://localhost:6868/> **X**。

Node.js HTTP

Home Page

localhost: (127.0.0.1) ◎

图12.21服务器多路径处理方式(浏览器)

j <http://localhost:6868/nop>.. **X v**

siode. js HTTP

Page not fcund

localhost (127.0.0.1) ◎ 图12.22服'务器多路径处理方式(浏览器)

从图12.22中可以看到，浏览器页面打印输出了服务器端第77行与第81行写给客户 端的信息(Page not found)。

说明：参数url的属性pathname用于链接地址的路径，比如U<http://localhost:6868/about>? name=king&id=123456” 的属性 pathname 值为/about。

1. 模拟ping命令连接服务器

这一节我们介绍一个模拟ping命令尝试连接服务器的应用，ping命令大家都很熟悉， 其主要功能是检测网络是否连通、帮助分析网络故障及其原因。在Node.js框架中，可以通 过HTTP模块的http.get()方法，实现模拟ping命令连接服务器的功能。

本例程客户端脚本文件(文件名称：chl2.http\_client\_ping.js)的主要代码如下：

01 *C\**

02 \* ch 12.http\_client\_ping.js

03 \*/

04 console.infof\* http - client ping ");

05 console.infb();

06 var http = require('http'); **//引入** http **模块**

07 console.log("Now start ping HTTP server...");

08 console.info();

09**尸**

1. **\*定义查询字段**
2. \*/
3. var options = (
4. host: localhost',
5. port: 6868,
6. path: 7
7. ｝；
8. /\*\*
9. **\* 自定义函数** ping\_server()
10. 7
11. function ping\_server() {
12. /\*\*
13. \*发送HTTP GET请求
14. \*/
15. http.get(options, function(res) ( L
16. if (res.statusCode == 200) (
17. console.logC'The site is up!");
18. )
19. else (
20. console.logC'The site is down!");
21. )
22. }).on('error', function(e) (
23. console.log(HThere was an error:" + e.message);
24. })；
25. }
26. /\*\*进行重定向操作的方法 ,
27. \*通过setlnterval()方法设定时间间隔
28. 7
29. setlnterval(ping\_server, 1000);

【代码分析】

第20~34行定义了一个名称为ping\_server()的自定义函数，用于实现ping服务器的功 能；第24〜33行通过调用http.get()方法在HTTP客户端发送GET请求方式的连接，在该 方法的回调函数中，通过判断Status\_Codes响应状态码打印输出不同的响应提示信息。

为了测试模拟ping命令连接服务器的操作，我们编写一个简单的HTTP服务器端代码 (代码名称：chl2.http\_server\_ping.js),用来随机响应客户端发来的连接请求，其主要代 码如下：

01 /\*\*

02 \* ch12.http\_server\_\_ping.js

03 7

04 console.info(H-— http - server ping ——

05 console.info();

06 var http = require('http'); //引入 http 模块

07 console.log("Now start HTTP server...");

08 console.info();

09 /\*\*

1. \*调用http.createServer()方法创建服务器
2. 7
3. http.createServer(function(req, res) {
4. //通过res.writeHead()方法写HTTP文件头
5. if(Math.round(Math.random())) (
6. res.writeHead(200, ('Content-type\*: 'text/html'})；
7. } else (
8. res.writeHead(404, ('Content-type\*: 'text/html'});
9. )
10. 〃通过res.end()方法发送响应状态码，并通知服务器消息完成
11. res.end();
12. }).listen(6868); 〃监听 6868 端口号

**【代码分析】**

该服务器端脚本代码与前面几个例程的服务器端脚本代码内容基本类似，主要区别就 是通过随机函数的方法响应客户端的连接。

下面，我们先依次启动chl2.http\_server\_ping.js服务器端代码。图12.23所示是本例程 在Ubuntu环境下输出的结果。然后启动ch 12.http\_client\_ping.js客户端代码。

图12.24是本例程在Ubuntu环境下输出的结果。从图中可以看到，客户端得到了服务 器端响应的连接请求，并随机给出了不同的响应信息。



/usr/local/bin/node chl2.http\_server\_ping.js http - server ping

/usr/local/bin/node chl2.http\_clientj)ing.js http • client ping

Now start ping HTTP server...

Now start HTTP server.

The site is down!

The site is up!

The site is up I The site is up!

The site is down!

图12.23模拟ping命令连接服务器（服务器） 图12.24模拟ping命令连接服务器（客户端）

说明：根据官方文档的说明，ping命令是指端对端的连通，通常用来检测网络是否连 通、帮助分析网络故障及其原因。该命令在Unix、Linux以及Windows系统中均可以使用， 本例程就是对ping命令功能的一种简单的模拟实现。

1. 安装Express开发框架

从这一节开始我们介绍基于Node.js的Express开发框架及其应用。Express框架目前 十分流行，它是一个简洁、灵活的基于Node.js的Web应用开发框架，提供了一系列强大 的特性，帮助开发人员创建各种Web和移动设备应用。Express开发框架不是对Node.js 已有的特性进行了二次抽象，而是在Node.js框架之上扩展了 Web应用所需的各种基本功 能。讲到这里，读者应该对Express开发框架有了一个初步的了解，下面还是以一个实际 例程介绍Express开发框架具体如何应用。

目前，Express开发框架的最新版本为4.x,其与早期的3.x版本在使用上略有不同， 我们就以4.x版本的内容开始。首先，我们需要安装Express开发库（系统此时已经默认安 装好Node开发环境），具体方法如下：

npm install -g express-generator

安装完毕后，可以使用以下命令查看Express开发库的版本号:

express -V 〃注意大写'V'

如果想查看express的帮助命令，可以使用以下命令查看：

express -h

注意：以上命令在Windows系统与Ubuntu系统下均是通用的'

12.11 使用Express开发框架开发的Hello World

下面，我们通过创建一个简单的Hello World应用，具体介绍一下Express开发框架的 使用方法。

1.**创建工程**

（I）在源代码目录中通过express命令创建一个新的Express工程目录（名称定义为 express-helloworld）,命令行如下：

express -e express-helloworld

在Ubuntu环境下，使用控制台工具输出的实际效果如图12.25所示。从中可以看到， express命令自动创建了工程目录及其一系列工作子目录和文件。

klng（aking-ThinkPad-X61: ~/webstorm.Js/projects/NodeJS/Chapterl2

ktng^ki.ng-ThtnkPad-X61:-/webstorm.js/pro]ects/NodeJS/Chapterl25 express e expre ss・helloworld

create : express-helloworId

Liedte : express-helloworld/package.json

: express-helloworld/app.js

create : express-helloworld/publtc

<re.ne : express-helloworld/publtc/tmages

cre<aTe : express-helloworld/pubUc/stylesheets

ci eate : express-helloworld/publtc/stylesheets/style.css

create : express-helloworld/routes

create : express-helloworld/routes/tndex.js

create : express-helloworld/routes/users.js

: express-helloworld/vtews

crests : express-helloworld/vlews/tndex.ejs

create : express-helloworld/vtews/error.ejs

create : express-helloworld/bin

create : express-helloworld/btn/www

install dependencies:

$ cd express-helloworld && npw install.

run the app:

5 DEBUG=express-helloworld:\* npn start

create : express-helloworld/public/javascrtpts

ktng^ktng-ThinkPad-X61:-/webstoE.js/projects/NodeJ5/Chapter12$ ,

图12.25 Express开发框架初步（创建工程目录）

，2）进入工程目录中通过npm命令下载依赖库（下载好的依赖库默认存放在目录 node modules中），命令行如下：

npm install

在Ubuntu环境下，使用控制台工具输出的实际效果（仅包含express库部分）如图12.26所示。从中可以看到，npm命令自动下载了一系列依赖库，图12.25所示仅仅截取了其中 express库的部分，还有其他一些依赖库没有截取出来，读者可以自行测试结果。

下面我们使用WebStorm查看一下express-helloworld工程的目录概况，如图12.27所 示。从工程图来看，目录结构还很复杂、内容也不少；没关系，我们就其中关键且常用的 内容做一下介绍，读者就会大致了解该目录结构的含义。

* bin,用于存放启动项目的脚本文件。
* node modules,存放所有的项目依赖库。
* public,用于存放静态文件（css> js、img等）。
* routes,用于存放路由文件（类似于MVC模型中的控制器controller概念）。
* views,用于存放页面文件（Ejs模板）。
* package.json,项目依赖配置及开发者信息。
* app.js,应用核心配置文件。



express@4.12.4 node\_modules/express —(nerge-descriptors@l.e.0

—uttls-merge^l.S.0

—cookie-signature^!.0.6

—methodsgl.1.1

—fresh^e.2.4

—cookie解.1.2

—escape-htmlgl.@.l

» E3 node\_modules

* 亡1 body-parser

,» & cookie-parser

* D debug
* 亡3 ejs
* Cj express a t] morgan

» O serve-favicon

—range-parser@1.0.2

—content-type^l.e.l

—ftnalhandler@0.3.6

—varyQl.e.e

— parseurl@1.3.&

—serve-statical.9.3

—content-disposition^®.5.6

* path - to - regex^Q .1.3

—depdgi.e.l

—qs@2.4.2

—on-ftntshedg2.2.1 (ee-ftrstgl.1.0) s

—etag@l.6.6 (crc^3.2.1)

—sendee.12.3 (destroy@1.0.3, ms@fl.7.1, mime@1.3.4)

* proxy-addr@1.6.8 (forwarded@0.1.0, tpaddr.js@l.8.1)

—type-ts@1.6.3 (media-typer@0.3.e, mine-types@2.1.1)

—accepts@1.2.9 (negotiator@6.5.3, mime-typesg2.1.1)

kina@kir»g-ThinkPad-X61:~/webstorm.js/projects/NodeJS/Chapterl2/express-helloworl

javascnpts 警 D stylesheets

图12.26 Express开发框架初步（下载依赖库） 图12.27 Express开发框架初步（工程目录概况）

1. **详解工程中的文件**

（1） package.json文件用于定义项目依赖配置及开发者信息，是整个工程首先需要关 注的文件，其主要代码如下：

"name'1: "express-helloworld",

“version”： ”0.0.0”，

"private": true,

"scripts,1: (

"start": "node ./bin/www"

},

"dependencies1': (

"body-parser11: ”~1.12.4”，

"cookie-parser\*': "~1.3.5",

"debug”: ”~2.2.0”，

"ejs":

"express'\*: "-4.12.4",

"morgan":

Mserve-faviconM: "~2.2.1M

}

)

**【代码分析】**

name属性用于定义项目名称；version属性用于定义版本号；scripts属性用于定义操作 命令，其可以非常方便地增加启动命令，比如默认的start参数，如使用npm start命令则代 表执行node ./bin/www命令；最后一个dependencies属性用于定义依赖库。

(2) app.js应用核心配置文件，Express开发框架从3.x升级到4.x版本，主要的变化 就在app.js文件中，其主要代码如下：

01 /\*\*

02 \*加载依赖库，原来这个类库都封装在connect中，现在需要单独加载

03 \* @type {[exports}

04 \*/

05 var express = require('express');

06 var path = require('path');

07 , var favicon = requireCserve-favicon1)；

08 var logger = require('morgan');

09 var cookieParser = requireCcookie-parser\*);

1. var bodyParser = requireCbody-parser1);
2. /\*\*
3. \*加载路由控制
4. \* @type (router|exports}
5. \*/
6. var routes = require('./routes/index');
7. var users = require(,./routes/users,)；
8. /\*\*
9. \*创建项目实例
10. \*/
11. var app = express();

*II*……此处省略部分代码

93 module.exports = app;

**【代码分析】 '**

我们注意到在app.js中，原来调用connect库的部分都被其他的库所代替了 (serve-favicon、morgan、cookie-parser、body-parser 等)；在默认项目中，只用到了其中 最基本的几个库，还没有用到其他需要替换的库；另外，原来用于项目启动的代码也被移 到./bin/www的文件，www文件也是一个Node的脚本，用于分离配置和启动程序，后面会 介绍。

1. 下面看一下www文件，其主要代码如下：

01 #!/usr/bin/env node

//……此处省略部分代码

1. var port = normalizePort(process.env.PORT || '6868');
2. app.set('port', port);

17

1. /\*\*
2. \* Create HTTP server.
3. \*/

21

1. var server = http.createServer(app);

23

1. /\*\*
2. \* Listen on provided port, on all network interfaces.
3. \*/

27

1. server.listen(port);
2. server.on('error\ onError);
3. server.on(\*listening\*, onListening);

31

*II*……此处省略部分代码

1. function onListening() {
2. var addr = server.address();
3. var bind = typeof addr === 'string' .
4. ? \*pipe1 + addr
5. : 'port' + addr.port;
6. debugfListening on ' + bind);
7. )

**【代码分析】**

WWW文件也是一个Node的脚本，主要用于项目启动代码、分离配置和启动程序；该 文件的内容很多，我们仅对其中重要的部分代码做一下解释，第15〜16行用于定义服务器 端口号，此处修改为6886；第22行使用http.createServer()方法创建了 HTTP服务器；第 28行使用server.on。方法监听服务器端口；第29行定义了 error错误事件处理方法onError, 具体实现在第56~78行；第30行定义了 listening监听事件处理方法onListening,具体实 现在第84-90行；以上这些代码的定义是不是很熟悉呢？这与使用Node.js框架的HTTP 模块创建服务器的过程是十分相似的，由此可见Express开发框架是完全基于Node.js框架 开发的。

1. 用于存放路由文件的route文件夹也是非常重要的，在应用程序加载隐含路由中 间件时，不用担心在中间件被加载时相对于路由器中间件的顺序；下面看一下路由脚本文

件index.js,其主要代码如下：

01 /\*\*

02 \* module define

03 \* @type {\*|exports}

04 \*/

05 var express = require('express');

06 var router = express.Router();

07 /\*\*

08 \* GET home page.

09 7

1. router.getC/1, function(req, res, next) (
2. res.render('index't (title: 'Express\*});
3. ))；

13

module.exports = router; 〃输出模型 router

【代码分析】

第05行引入express模块，同时赋于变量express；第06行通过express.Router()方法定 义路由变量router：第10~12行通过调用router.get()方法发送GET方式的连接请求，其回 调函数包含了 res、req和next三个参数；res与req参数读者肯定熟悉，是HTTP模块中 http.get()方法原始就有的；而next参数是新的特性，当连接请求能够匹配多个路由时，通 过调用next。方法进行处理；第11行通过调用res.render()7j法渲染view目录下的index页 面文件，本例程中定义了一个{ title: Express' }数组对象，该对象将在index页面文件中进 行使用。

1. 最后，介绍一下用于页面文件的view文件夹，该文件夹下面的页面均通过Ejs 模板文件实现；下面看一下页面文件index.ejs,其主要代码如下：

<!DOCTYPE html>

<html>

<head>

<title><%= title %></title>

<link rel='stylesheet' href=7stylesheets/style.css' />

<link rel="stylesheetH href=M<http://cdn.bootcss.com/bootstrap/3.3.2/css/bootstrap.min.css>"> </head> <body>

<div class=,,well jumbotron">

<h1><%= title %></h1>

<p>Hello World, this is a simple Express Page with Bootstrap.</p>

<p><a class=Mbtn btn-primary btn-lg" href=M#M role=Hbuttonn>About Express</a></p>

</div>

<script src=n[http://cdn.bootcss.com/jquery/1.11.2/jquery.min.js](http://cdn.bootcss.com/jquery/1.11.2/jquery.min.jsM%3e%3c/script)[M](http://cdn.bootcss.com/jquery/1.11.2/jquery.min.jsM%3e%3c/script)[></script](http://cdn.bootcss.com/jquery/1.11.2/jquery.min.jsM%3e%3c/script)>

<script src="[http://cdn.bootcss.com/bootstrap/3.3.2/js/bootstrap.min.js](http://cdn.bootcss.com/bootstrap/3.3.2/js/bootstrap.min.jsn%3e%3c/script)[n](http://cdn.bootcss.com/bootstrap/3.3.2/js/bootstrap.min.jsn%3e%3c/script)[></script](http://cdn.bootcss.com/bootstrap/3.3.2/js/bootstrap.min.jsn%3e%3c/script)>

</body>

</html>

为了展现出更美观的页面效果，整个index.ejs页面使用Bootstrap框架进行了重构，感 兴趣的读者可以找一些关于Bootstrap框架的文档了解一下；另外，使用到了 index.js路由 文件传来的title属性进行了页面输出。

1. 调试项目

通过npm命令启动express框架服务器，命令行如下：

npm start

在Ubuntu环境下，使用控制台工具输出的实际效果（仅包含express库部分）如图12.28 所示。

kihg^king^ThlnkPad-XiSl: -/webstortn.js/projects/NodeJS/Chapterr:

kinggking-ThinkPad-X61:~/webstorm.js/projects/NodeJS/Chapterl2/express-he lloworldS npn start

* express-helloworld@e.8.8 start /hone/ktng/webstorm.js/projects/NodeJS/C hapter!2/express-helloworld
* node ,/btn/www

图12.28 Express开发框架初步（启动Express框架服务器）

图12.28显示出Express框架服务器已经成功启动运行，打开浏览器，在地址栏中输入 以下地址：[http://localhost:6868/,其效果如图12.29所示。浏览器页面打印输出了模板文件](http://localhost:6868/,%e5%85%b6%e6%95%88%e6%9e%9c%e5%a6%82%e5%9b%be12.29%e6%89%80%e7%a4%ba%e3%80%82%e6%b5%8f%e8%a7%88%e5%99%a8%e9%a1%b5%e9%9d%a2%e6%89%93%e5%8d%b0%e8%be%93%e5%87%ba%e4%ba%86%e6%a8%a1%e6%9d%bf%e6%96%87%e4%bb%b6) index.ejs定义的内容，整个页面也是Bootstrap风格的。

说明：路由功能是Express4.x版本以后全面改进的功能。在应用程序加载隐含路由中 间件时，不用担心在中间件被装载相对于路由器中间件的顺序，其可以帮助我们更好地组 织代码结构。如果我们要管理不同的路径，那么可以直接配置为多个不同的路由。



图12.29 Express开发框架初步（浏览器效果）

1. Express开发框架路由处理

这一节我们介绍基于Node.js的Express开发框架的路由处理方法。所谓“路由”就是 指为不同的URL访问路径，指定不同的处理方法。HTTP服务器提供路由服务，对于浏览 器发过来的不同的URL请求，根据路由规则进行解释与响应，这就是HTTP服务器的路由 功能。

本节这个例程在上一节例程的基础上，实现了 Express开发框架路由处理的功能。关 于Express开发框架的搭建，读者可以参考上-节的内容，这里主要介绍路由功能的处理 方法。

1. 首先，我们通过express命令创建一个新的Express I程目录express-routes,命 令行如下：

express -e express-routes

1. 进入工程目录中通过npm命令下载依赖库(下载好的依赖库默认存放在目录

node modules中),命令行如下：

npm install

下面通过修改叩p.js脚本文件，并在路由文件夹routes和视图文件夹views中添加相应 代码文件，来实现本例程路由处理的功能。我们先浏览一下express-routes工程的目录概况, 如图12.30所示。

铲 routes

匾 aboutjs

& index.js

陲 narnejs

& pwd .js

& users.js

亨 O views

国 aboutejs

0 error.ejs

国 index.ejs

由 name.ejs

El pwd.ejs

国 appjs

0 npm-debug.log 的 package. json

图12.30 Express开发框架初步(工程目录概况)

1. 下面是app.js应用核心配置文件，其主要代码如下：

01 /\*\*

02 \*加载依赖库，原来这个类库都封装在connect中，现在需要单独加载

03 \* @type {\*|exports}

04 \*/

05 var express = require('express\*);

06 var path = require('path');

07 var favicon = require('serve-favicon');

08 var logger = require('morgan');

09 var cookieParser = require('cookie-parser');

1. var bodyParser = require('body・parser');
2. /\*\*

|  |  |  |
| --- | --- | --- |
| 12 | \*加载路由控制 | |
| 13 | \* @type (router|exports} |  |
| 14 | *\*1* |  |
| 15 | var routes = requireC./routes/index1); |  |
| 16 | var users = requireC./routes/users'); |  |
| 17 | var about = require('./routes/about'); |  |
| 18 | var name = require('./routes/name'); |  |
| 19 | var pwd = require('./routes/pwd'); |  |
| 20 | /\*\* |  |
| 21 | \*创建项目实例 |  |
| 22 | \*/ |  |
| 23 | var app = express(); |  |
| 24 | /\*\* |  |
| 25 | \*定义EJS模板引擎和模板文件位置，也可以使用jade或其他模型引擎 | |
| 26 | \* view engine setup |  |
| 27 | \*/ |  |
| 28 | app.set('views', path.join( dirname, 'views')); | |
| 29 | app.setfview engine', 'ejs'); |  |
| //... | …此处省略部分代码 |  |
| 52 | /\*\* |  |
| 53 | \*匹配路径和路由 |  |
| 54 | 7 |  |
| 55 | app.use(7\ routes); | 〃访问根路径 |
| 56 | app.use(7users', users); | 〃访问路径/users |
| 57 | app.getC/test1, function(req, res) ( | 〃访问路径/test |
| 58 | res.send(\*Test Routes!')； |  |
| 59 | }); |  |
| 60 | app.get(7about', about); | 〃访问路径/about |
| 61 | app.get(7name', name); | 〃访问路径/name |
| 62 | app.get(7wildcard/\*\*, function(req, res) ( | 〃访问通配符路径/wildcard/\* |
| 63 | res.send(req.originalUrl); //req.originalllrl 获取当前 URL | |
| 64 | )); |  |
| 65 | app.get(7pwd/:passwd', pwd); | 〃访问路径/pwd/:passwd |
| 66 | app.get(7next/\*', function(req, res, next) ( | 〃访问路径/next/\* |
| 67 | req.control ="转移控制权”； |  |
| 68 | next();//TODO:把权限转移到下一个路由 | |
| 69 | })； |  |
| //… | …•此处省略部分代码 |  |

**【代码分析】**

第55行使用app.use()方法设定了访问根路径(“/”)的路由配置，其中app.use()方 法中的第一个参数设定了根路径，第二个参数定义回调方法routes；而参数routes的定义见 第15行，该行引用了路由文件夹routes中的脚本文件index.js。

1. 下面看看index.js路由脚本文件的内容，其主要代码如下：

01 /\*\*

02 \* module define

03 \* @type『exports}

04 7

05 var express = require('express\*);

06 var router = express.Router();

07 /\*\*

08 \* GET root page.

09 7

1. router.get(7, function(req, res, next) (
2. res.renderfindex1, (
3. title: 'Express - Routes'
4. })；
5. })；
6. module.exports = router;// 输出模型 router

**【代码分析】**

第10〜14行通过调用router.get。方法发送GET方式的连接请求，其回调函数包含了 res、req和next三个参数；第11~13行通过调用res.render()方法渲染view目录下的index 页面文件，本例程中定义了一个{ title: Express - Routes\*}数组对象，该对象将在index模板 页面文件中进行使用。

1. 然后，再看看通过Ejs模板文件实现的view页面文件夹，其中index.js路由文件 中渲染的模板文件index.ejs的主要代码如下：

01 <!DOCTYPE html>

02 <html>

03 <head>

04 <title><%= title %></title>

05 <link rel=,stylesheet' href=7stylesheets/style.css' />

06 </head>

07 <body>

08 <h1><%= title %></h1>

09 <p>Welcome to <%= title %></p>

1. </body>
2. </html>

第08〜09行使用了 index.js路由文件传来的title属性值在页面中进行了渲染输出。

1. 最后通过叩m命令(nprn start)启动express框架服务器；Express框架服务器成 功启动运行后，打开浏览器，在地址栏中输入以下地址：[http://localhost:6868/,来访问根路](http://localhost:6868/,%e6%9d%a5%e8%ae%bf%e9%97%ae%e6%a0%b9%e8%b7%af) 径，其效果如图12.31所示。

从图12.31中可以看到，浏览器页面通过title属性打印输出了模板文件index.ejs定义 的内容，说明路由配置成功了。

我们返回app.js应用核心配置文件，第56行使用app.use。方法设定了访问路径“/users” 的路由配置,其中app.use。方法中的第一个参数设定了 “/users”路径,第二个参数定义回调方法users；而参数users的定义见第16行，该行引用了路由文件夹routes中的脚本文件 users .j So

lj Express - Routes A **会 v**

Express - Routes

Welcome to Express - Routes

localhost (127.0.0.1) O

图12.31 Express开发框架路由处理(访问根路径)

(7)下面我们看一下users.js路由脚本文件的内容，其主要代码如下：

01 /\*\*

02 \* module define

03 \* @type {[exports}

04 \*/

05 var express = require('express\*);

06 var router = express.Router();

07 router.getf/\*, function(req, res, next) (

08 res.sendfrespond with a resource');

09 })；

1. module.exports = router; // 输出模型 router

**【代码分析】**

第08行通过调用res.send()方法直接在根页面文件打印输出了提示信息''respond with a resource；第 10 行通过 module.exports 方法输出 router 模型。

在确认Express框架服务器处于运行状态后，在浏览器地址栏中输入以下地址：

<http://localhost:6868/users> ,来访问 “/users”路径,其效果如图12.32 所示。

从图12.32中可以看到，浏览器 页面打印输出了提示信息"respond with a resource ”,说明路由配置成功

i\_j <http://localhost:6868/users> M **令**

respond with a resource

localhost (127Q0.1)**。**

图12.32 Express开发框架路由处理(访问“/users”路径)

T»

12.13应用Express框架实现登录页面

这一节我们介绍应用Express开发框架模拟实现登录页面的方法。本节这个例程省略了连接数据库获取用户名和密码进行校验的过程，仅仅是应用Express框架模拟一个登录 的过程，相对比较简单。

首先，在源代码目录中通过express命令创建一个新的Express I程目录（名称定义为 express-login），命令行如下：

express -e express-login

然后，进入工程目录中通过 叩m命令下载依赖库（下载好的依赖库默认存放在目录 “Dode\_modules"中）,命令行如下：

npm install

下面通过修改app.js脚本文件，并在路由文件夹routes和视图文件夹views中添加相应 代码文件，来实现本例程登录的功能。

我们先浏览一下express-login工程的目录概况，如图12.33所示。

▼ - express\*login

► Dbin

妨 D node\_modules 粉 & public

*初* D routes

陲 indexjs

& login.js usersjs

T D views

E| error.ejs mdex.ejs

@ login.ejs

& app.js

& packagejson

图12.33应用Express框架实现登录页面(工程目录概况)

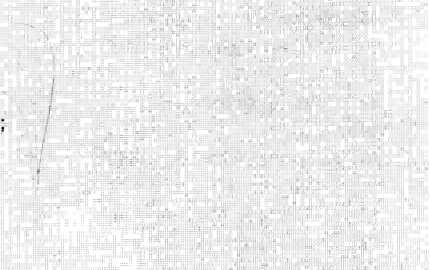
下面是app.js应用核心配置文件，其主要代码如下：

01 /\*\*

02 \*加载依赖库，原来这个类库都封装在connect中，现在需要单独加载

03 \* @type (\*|exports}

04 \*/

05 var express = require('express');

06 var path = require('path\*);

07 var favicon = require('serve-favicon');

08 var logger = require('morgan');

09 var cookieParser = requireCcookie-parser1)

1. var bodyParser = requirefbody-parser\*);
2. /\*\*
3. \*加载路由控制
4. \* @type (router|exports}
5. \*/
6. var routes = requireC./routes/index1);
7. var users = require('./routes/users\*)；
8. var login = require('./routes/login,);
9. /\*\*
10. \*创建项目实例
11. \*/
12. var app = express();
13. *r\**
14. \*定义EJS模板引擎和模板文件位置，也可以使用jade或其他模型引擎
15. \* view engine setup
16. \*/
17. app.set('views', path.ioint dirname, 'views'));
18. app.set(\*view engine\*, \*ejs');

//•…••此处省略部分代码

1. /\*\*
2. •定义静态文件目录
3. \*/
4. app.use(express.static(path.join( dimame, 'public')));
5. /\*\*
6. \*匹配路径和路由
7. \*/
8. app.useC/1, routes);
9. app.use(7users\*, users);
10. app.post(7login', login);
11. /\*\*

*II*……此处省略部分代码

第55行使用app.post()方法设定了访问路径(“/login”)的路由配置，第17行定义了 "/login” 路由文件的位置("/routes/login.js” )。

下面看一下loginjs路由脚本文件的内容，其主要代码如下：

01 /\*\*

02 \* module define

03 \* @type {\*|exports}

04 \*/

05 var express = require(\*express');

06 var router = express.Routerf);

07 /\*\*

08 \* GET login page.

09 \*/

1. router.post(7login\ function(req, res, next) (
2. res.renderflogin', (
3. title: 'Express - Login',
4. userid: req.body.userid,
5. pwd: req.body.pwd
6. ))；
7. })；
8. module.exports = router; // 输出模型 router

**【代码分析】**

第11~15行通过调用res.render()方法渲染view目录下的login页面文件，本例程中定 义了一个数组对象{title: Express - GET/POST\*, userid: req.body.userid, pwd: req.body.pwd }, 该对象将在login模板页面文件中进行使用。

然后，再看一下通过Ejs模板文件实现的view页面文件夹，其中login.js路由文件中 渲染的模板文件login.ejs的主要代码如下：

01 <!DOCTYPE html>

02 <html>

03 <head>

04 <title><%= title %></title>

05 <link rek'stylesheet\* href=7stylesheets/style.css' />

06 </head>

07 <body>

08 <h1><%= title %></h1>

09 <p>Welcome to <%= title %></p>

1. <p>User id: <%= userid %></p>
2. <p>Password: <%= pwd %></p>
3. </body>
4. </html>

第08〜11行使用到了 login.js路由文件传来的title> userid和pwd属性值在页面中进 行了渲染输出，即打印输出了登录用户名和密码。

那么，登录用户名和密码来自什么地方呢？这里需要编写一个登录页面，我们就使用 已有的index.js路由文件和index.ejs模板页面文件。

下面我们看一下index.js路由脚本文件的内容，其主要代码如下：

01 /\*\*

02 \* module define

03 \* @type『exports}

04 \*/

05 var express = require('express\*);

06 var router = express.Router();

07 /\*\*

08 \* GET root page.

09 \*/ ./

1. router.getC/1, function(req, res, next) ( \*
2. res.renderfindex', {
3. title: 'Express - Login'
4. ))；
5. })；
6. module.exports = router;// 输出模型 router :匏震客

【代码分析】

第11〜13行通过调用res.render()方法渲染view目录下的index页面文件，本例程中定 义了一个数组对象｛title: Express - Login\*),该对象将在index模板页面文件中进行使用。

然后，再看一下通过Ejs模板文件实现的view页面文件夹，其中inidex.js路由文件中 渲染的模板文件index.ejs的主要代码如下：

01 <!DOCTYPE html>

02 <html>

03 <head>

04 <title><%= title %></title>

05 <link rel=,stylesheet href=7stylesheets/style.css' />

06 </head>

07 <body>

08 <h1><%= title %></h1 >

09 <p>Welcome to <%= title %></p>

1. <form method=Hposf action=7loginw>
2. <p>User id:〈input id="use「id" name="use「id” type=Htext"></p>
3. <p>Password: <input id="pwd" name=,,pwd,' type=,'password,,x/p>
4. <input id=”submit" name="submit” type="submitM value=M提交”〉
5. </form>
6. </body>
7. </html>

login.ejs模板文件读者是不是很熟悉呢？第10〜14行实际上就是一个Form表单，包 含了 userid和pwd两个text类型文本框、以及一个提交按钮；第10行在vfbrm＞标签内定 义了访问方式method=post, url地址为action=/login,而路由地址"/login”在前面的app.js 文件中有定义；

下面我们测试一下，在确认Express框架服务器处于运行状态后，在浏览器的地址栏 中输入地址：http://localhost:6868/post来进行访问，其效果如图12.34所示。我们分别在 User id和Password文本框中输入用户ID和密码，然后点击提交按钮模拟登录，其效果如 图12.35所示。

**Express - Login**

图12.34 表单页面

Welcome to Express - Login

User id: king

Password: 123456

图12.35模拟登录

从图12.35中可以看到，Form表单中的userid和pwd文本框中的数据成功提交了，说 明登录操作完成了。

12.14 Express框架实现Ajax方式操作

这一节我们介绍应用Express开发框架实现Ajax方式操作的方法。首先，通过express 命令创建一个新的Express工程目录express-ajax,命令行如下：

express -e express-ajax

然后，进入工程目录中通过 叩m命令下载依赖库（下载好的依赖库默认存放在目录 node modules中），命令行如下：

npm install

下面通过修改app.js脚本文件，并在路由文件夹routes和视图文件夹views中添加相应 代码文件，来实现本例程登录的功能。

我们先浏览一下express-ajax工程的目录概况，如图12.36所示。

勘也bin

» 亡I node\_module$

* public

D images

\* D javascripts

» C3 stylesheets

* D routes

indexjs  
屋 users.js

*9* 匚］views

gj error.ejs

国 index.ejs

& app-js

B npm-debug.log

W package.json

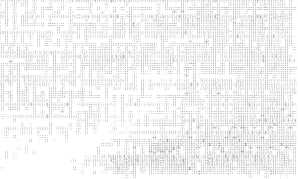
图12.36 Express框架实现Ajax方式操作（工程目录概况）

下面是app.js应用核心配置文件，其主要代码如下:

01 /\*\*

02 \*加载依赖库，原来这个类库都封装在connect中，现在需要单独加载

03 \* @type {^exports}

04 \*/

05 var express = require('express');

06 var path = require('path');

07 var favicon = require('serve-favicon');

08 var logger = requireCmorgan1);

09 var cookieParser = require('cookie-parser');

10 var bodyParser = require('body-parser\*)；

1. /\*\*
2. \*加载路由控制
3. \* @type (router|exports)
4. 7
5. var routes = require(,./routes/index');
6. var users = require(\*./routes/users');
7. /\*\*
8. •创建项目实例
9. \*/
10. var app = express();
11. /\*\*
12. \*定义EJS模板引擎和模板文件位置，也可以使用jade或其他模型引擎
13. \* view engine setup
14. \*/
15. app.set(\*views\ path.ioin( dirname, 'views'));
16. app.setfview engine\*, 'ejs');

〃……此处省略部分代码

1. /\*\*
2. •匹配路径和路由
3. 7
4. app.use(7, routes);
5. app.use(7users\ users);
6. app.get(7jsonp', function(req, res, next) (
7. res.jsonp((
8. status/status : jsonp'
9. ))；
10. })；
11. app.get(7json', function(req, res, next) (
12. res.send({
13. status:'status : json'
14. })；
15. })；

*II*……此处省略部分代码

**【代码分析】**

第54〜58行使用app.get。方法设定了访问路径(“/jsonp” )的路由配置，第55~57 行通过调用res.jsonp()方法实现跨域的JSON格式数据传送，该方法内定义了一个数组 (status:'status : jsonp'}用于传送：第59〜63行使用app. get。方法设定了访问路径“/json” 的路由配置，第60〜62行通过调用res. send。方法向页面发送JSON格式数据，该方法内 定义了一个数组{status:'status : json，}用于数据发送；

下面我们看一下index.js路由脚本文件的内容，其主要代码如下：

01 /\*\*

02 \* module define

03 \* @type {"exports}

04 \*/

05 var express = require('express');

06 var router = express.Router();

07 /\*\*

08 \* GET root page.

09 \*/

1. router.getf/', function(req, res, next) (
2. res.renderfindex\*, (
3. title: 'Express - Ajax'
4. ))；
5. })；
6. module.exports = router; 〃输 出模型 router

**【代码分析】**

第11~13行通过调用res.render()方法渲染view目录下的index页面文件，本例程中定 义了一个数组对象{title: 'Express - Ajax'),该对象将在index模板页面文件中进行使用。

然后，再看一下通过Ejs模板文件实现的view页面文件夹，其中index.js路由文件中 渲染的模板文件index.ejs的主要代码如下：

01 <!DOCTYPE html>

02 <html>

03 <head>

04 <title><%= title %></title>

05 <link rel=,stylesheet' href=7stylesheets/style.css\* />

06 </head>

07 <body>

08 <h1><%= title %></h1>

09 <p>Welcome to <%= title %></p>

1. <p><a href=,,ajax.htmr,>Click to ajax.html</a></p>
2. </body>
3. </html>

第10行定义了一个超链接，连接地址为一个静态页面ajax.htmlo 那么，Express框架将静态页面放置在什么地方呢？答案是public目录。下面看一下 ajax.html静态文件的内容，其主要代码如下：

01 <!DOCTYPE html>

02 <html>

03 <head lang=''enH>

04 <meta charset=,,UTF-8,,>

05 <script src=n[http://code.jquery.com/jquery-latest.js"></script](http://code.jquery.com/jquery-latest.js%22%3e%3c/script)>

06 <title>Express - Ajax</title>

07 </head>

08 <body>

09 <script type=,'text/javascriptM>

1. function getjsonp() {
2. $.getJSON(”http://127.0.0.1:6868/jsonp?callback=?H, function(data) {
3. $('#input-jsonp)val('JSONP info: J data.status);
4. });
5. } ；
6. function getjson() (
7. $.getJSON("json", function(data) {
8. $('#input-json').valCJSON info :' + data.status);
9. });

“ 二

1. }
2. </script>
3. <!— titl© —a
4. <p>Express - Ajax</p>
5. <!- jsonp method ~>
6. <p>
7. <a href=”javascript:get\_Jsonp();">点 击调用-jsonp</a><br/>
8. <input id=',input-jsonp° type="text" size="48', /><br/>
9. </p>
10. <!- json method ->
11. <p>
12. <a href="javascript:getjson();''>点击调用-json</a><br/>
13. <input id=,'input-json,' type="textn size=M48M /><br/>
14. v/p>

n ! y 罚药M i! s y % " !,i i H 忍 " *H H ll \* 0* H 0 =：：：：：：=：=。=：" ”："。::

1. </body>
2. </html>

**【代码分析】**

第05行引用了 CDN方式的jQuery库文件，因为下面的脚本代码需要调用jQuery库 的Ajax方法；第25行定义了 一个超链接，在href属性内使用javascript语法调用了一个自 定义函数getjsonp()；第10〜14行是自定义函数getJsonp()的内容，其中第11~13行通过 调用 jQuery 的$.getJSON()方法请求了链接地址“http:〃127.0.0.1:6868/jsonp?callback=?”， 这个链接地址的写法涉及到了 JSONP的原理，读者可以参考相关内容，这里只需要知道必 须写成这种形式就可以了；在该方法的回调函数内，第12行将通过Ajax方式获取的JSON 数据(data.status)内容显示在第26行定义的id值为input-jsonp的vhiput＞文本框中。

第30行定义了一个超链接，在href属性内使用javascript语法调用了一个自定义函数 getjson()；第15~19行是自定义函数get\_json()的内容，其中第16〜18行通过调用jQuery 的$.getJSON()方法请求了路由地址json；在该方法的回调函数内，第17行将通过Ajax方 式获取的JSON数据(data.status)内容显示在第31行定义的id值为input-json的＜input＞ 文本框中。

下面我们测试一下，在确认Express框架服务器处于运行状态后，在浏览器的地址栏 中输入地址：http:〃localhost:6868/来进行访问，其效果如图12.37所示。

localhost

Express - Ajax

Welcome to Express - Ajax

图12.37 Express框架实现Ajax方式操作（首页）

然后单击超链接"Click to ajax.html”，其效果如图12.38所示。分别点击超链接“单 击调用-jsonp”和“单击调用-json",其效果如图12.39所示，从图中可以看到，通过 无页面刷新的Ajax方式，页面成功读取到了 JSON格式的数据，并在文本框中进行了显示。



|  |  |  |
| --- | --- | --- |
| 《 C D localhost ☆号 = |  | < 0 D localhost 6868Ajax.htiTil ☆ g 三 |
| Express - Ajax |  | Express - Ajax |
| 点击调用- jsonp |  | 点击调用- .isonp |
|  |  | JSONP info : status jsonp |
| 点击调用- json |  | 点击调用- json |
|  |  | JSON info status json |
| 图12.38 Ajax页面 | | 图12.39完成Ajax调用 |

说明：JSONP （英文全写为JSON with Padding ）是JSON的一种“使用模式”，可用 于解决浏览器的跨域数据访问问题，

第13章 Node.js Util常用工具

在Node.js框架中，常用工具（Util）模块是一个核心模块，其存在是为了解决核心 JavaScript的功能过于精简而设计的。例如：对一个原型对象的继承功能、对象格式化操作、 将任意对象转换为字符串的操作、调试输出功能、正则表达式验证等，常用工具（Util）模 块均给出了很好的实现。因此，我们可以将Node.js框架的常用工具（Util）模块视为最好 的脚本开发辅助工具。

本章主要包括以下内容。

* 常用工具（Util）模块继承方法。
* 常用工具（Util）模块对象转化方法。
* 常用工具（Util）模块测试工具方法。

13.1原型对象继承

本章关于常用工具（Util）模块的第一个应用例程，我们向读者介绍原型对象继承的方 法。众所周知，JavaScript语言的面向对象特性是基于原型的，这与常见的基于类的高级源 语言是不同的，这是因为JavaScript语言没有提供对象继承的语言级别特性，而是通过原 型复制来实现的。在本节这个例程中，将使用常用工具（Util）模块的util.inherits。方法实 现原型对象继承的功能。

本例程ch 13.util inherits.js的主要代码如下：

01 /\*\*

02 \* ch13.utiljnherits.js

03 \*/

04 console.infoC'-—- util inherits() •—-H);

console.info();

05

06

07

08

09

10

11

12

13

var util - require('utir)； 〃引入常用工具(Util)模块

/\*\*

\* - :" " ' ■: / """ : ■: X-： : ;： ■:■•:■■: - X : / X

\*定义原型基类Base

\* ©constructor

function Base() (

this.name = 'base\*;

this.year = 2015;

this.sayHello = function() ( console.log('Hello ' + this.name + + 'this is [[9]](#footnote-10) + this.year +

)；

}

/[[10]](#footnote-11)

\*定义基类Base的方法showName()

7

Base.prototype.showName = function() (

console.log(this.name);

)；

/\*\*

•定义基类Base的方法showYear()

\*/

Base.prototype.showYear = function() (

console.log(this.year);

}；

/\*\*

\*定义原型子类Child

* ©constructor

7

function Child() {

this.name = 'child';

)

**【代码分析】**

第06行引入常用工具（Util）模块，同时赋于变量util；第11〜17行定义了基类Base, 该基类构造方法内定义了两个属性（name与year）和一个方法（sayHello）,并进行了初 始化操作；第21〜23行通过基类Base的原型方法定义了一个方法（showName）,用于打 印输出构造方法内的name属性；第27~29行通过基类Base的原型方法定义了另一个方法 （showYear）,用于打印输出构造方法内的year属性；

第34-36行定义了子类Child,该子类构造方法内定义了一个属性（name）,并进行 了初始化操作：

第40行调用utiLinherits。方法实现了原型对象继承的操作，关于util.inherits。方法的语 法如下：

语法：util.inherits（constructor, superConstructor）; 〃原型对象继承

utiLinherits。方法用于实现原型对象继承的操作：其中，第一个constructor参数用于定 义继承的类对象（子类）,第二个superConstructor参数用于定义被继承的类对象（基类）。

第45〜49行定义了一个基类对象（变量名称为objBase）,并通过该变量调用了前面 基类中定义的三个方法（sayHello、showName与show Year）,最后打印输出了该对象；第 54〜58行定义了一个子类对象（变量名称为objChild）,并通过该变量调用了前面基类中 定义的两个方法（showName与show Year）,最后打印输出了该对象。

图13.1所示是本例程在Ubuntu环境下输出的结果。从结果可以看到，第46行输出的

内容为base,第47行输出的内容为 2015,第48行通过sayHello。方法输出 的内容为 Hello base,this is 2015w ,第 49行输出的objBase对象为｛ name: 'base', year: 2015, sayHello: [Function] ｝: 第55行输出的内容为“child”，第56 行输出的内容为“undefined”,第58 行输出的内容为｛ name: 'child' ｝，说明 基类Base构造函数内部的属性与方法 没有被子类Child所继承。

**© Run**

A 令 /usr/local/bin/node chl3. util\_inhents js

util inherits()

a 4 —base

g 目 2015

■ Hello base.this is 2015.

K 上 { naiie base1. year： 2015, sayHello： [Function) ) 昏 child

X undefined

•p ® { nane. 'child\* }

Process finished with exit code 0

图13.1原型对象继承的方法

说明：使用util.inherits（）继承方法时，基类构造函数内部创造的属性和方法均不会被子 类所继承，只有通过原型方法创建的属性和方法才会被子类所继承。

13.2将任意对象转换为字符串

本节我们向读者介绍将任意对象转换为字符串的方法。在很多情况下，需要将不同类 型的对象统一转换为字符串格式进行操作，Node.js框架的常用工具（Util）模块提供了一个utiLinspect()方法来实现该功能。

本例程(代码文件名称：chl3.util\_inspect.js)的主要代码如下：

01 /\*\*

02 \* ch13.util\_inspect.js

03 \*/

04 console.info(" util inspect()

05 console.info();

//引入常用工具(Util)模块

06 var util = require('utir);

07 /\*\*

08 \*定义原型类Person

09 \* ©constructor

1. \*/
2. function Person() {
3. this.name = 'person';
4. this.toString = function() (
5. return this.name;
6. }；
7. }
8. /\*\*
9. \*定义Person对象
10. \* @type (Base)
11. 7
12. var obj = new Person();
13. conso!e.log(util.inspect(obj));
14. console.log(util.inspect(obj, true));

**【代码分析】**

第11~16行定义了一个类Person,该基类构造方法内定义了一个属性(name)和一个 方法(toString),并进行了初始化操作；第21行定义了一个Person类对象(变量名称为 obj)；第22〜23行调用util.inspect()方法实现了将对象转换为字符串的操作。

图13.2是本例程在Ubuntu环境下的结果。从结果可以看到，第22行仅仅打印输出了 obj对象的字符串表现形式，而第23行将obi对象的不可枚举属性也一并打印输出了。



/usr/local/bin/node chl3.util\_inspect.]s util inspectO

{ name： 'person', toString： (Function) } { name： 'person'.

toString：

{ [Function] [length]: 0. [nanej： [arguaents]: null, (caller]: null, [prototype]: { (constructor): (Circular]

Process finished with exit code 0

图13.2将任意对象转换为字符串的方法

1. 验证是否为数组

本节我们向读者介绍如何验证一个对象是否为数组的方法。当我们需要对一个数组对 象进行操作时，该方法是非常实用的，Node.js框架的常用工具(Util)模块提供了一个 util.isArray()方法来实现该功能。

本例程(代码文件名称：ch 13.util isArray.js)的主要代码如下：

01 /\*\*

02 \* ch13.utiIJsArray.js

03 \*/

04 consolednfoC™- util isArray()—

05 console.info();

06 var util = requirefutil\*); 〃引入常用工具(Util)模块

07 console.log(util.inspect(util.isArray(n)))；

08 console.log(util.inspect(utiLisArray(new Array)));

09 console.log(util.inspect(utiLisArray({})));

**【代码分析】**

第07〜09行通过调用util.isArray()方法来判断给定的对象是否为数组类型，关于 util.isArray ()方法的语法如下：

语法：util.isArray(object); *II*判断给定的对象是否为数组类型

util.isArray()方法用于判断给定的对象是否为数组类型，如果是就返回true,否则返回 false o

图13.3是本例程在Ubuntu环境下的结果。

Run chU.util\_i5AFray.)s 蔡\*

X 普 ;/usr/local/bin/node chl3.util\_isArray.js util isArray ()

true true false

Process finished with exit code G

图13.3验证是否为数组的方法

说明：utiLisArray()方法用于判断给定的对象是否为数组类型。

13.4验证是否为日期格式

本节我们向读者介绍如何验证一个对象是否为日期格式的方法。当我们需要对一个日 期格式对象进行操作时，该方法是非常实用的，Node.js框架的常用工具(Util)模块提供

了一个util.isDate ()方法来实现该功能。

本例程(代码文件名称：chl3.util\_isDate.js)的主要代码如下:

01 /\*\*

02 \* ch13.utilJsDate.js

03 \*/ ' 二" *，*

04 console.info(H util isDate()—一 ");

05 console.info();

06 varutil = requirefutir); *II*引入常用工具(Util)模块

07 console.log(util.inspect(util.isDate(new Date()))); //返回 true

08 console.log(util.inspect(util.isDate(Date()))); //返回 false

09 console.log(util.inspect(util.isDate(0))); //返回 false

**【代码分析】**

第07〜09行通过调用util.isDate()方法来判断给定的对象是否为日期格式类型，关于 util.isDate()方法的语法如下：

语法：util.isDate(object); 〃判断给定的对象是否为日期格式类型

util.isArrayO方法用于判断给定的对象是否为日期格式类型，如果是就返回true,否则 返回false □

图13.4所示是本例程在Ubuntu环境下输出的结果。



/usr/local/bin/node chl3.util\_isDate.js

util isDateO

野 Process finished with exit code 0

图13.4验证是否为日期格式的方法

说明：util,isDate()方法用于判断给定的对象是否为日期格式类型。

13.5验证是否为正则表达式

本节我们向读者介绍如何验证一个对象是否为正则表达式的方法。当我们需要使用正 则表达式时，该方法是非常实用的，Node.js框架的常用工具(Util)模块提供了一个 util.isRegExp ()方法来实现该功能。

本例程(代码文件名称：ch 13.util isRegExp.js)的主要代码如下：

01 /\*\*

02 \* ch 13. util JsReg Exp.js

03 \*/

04 console.info(" util isRegExp()

05 console.info();

|  |  |
| --- | --- |
| 06 var util = require('util'); //引入常用工具(Util)模块  07 console.log(util.inspect(util.isRegExp(/some regexp/)));  08 console.log(util.inspect(util.isRegExp(new RegExp('another regexp'))));  09 console.log(util.inspect(util.isRegExp({})));  **【代码分析】** | //true //true //false |

第06行引入常用工具(Util)模块，同时赋于变量util；

第07~09行通过调用util.isRegExp()方法来判断给定的对象是否为正则表达式，关于 util.isRegExp ()方法的语法如下：

语法：util.isRegExp(object); *II*判断给定的对象是否,为正则表达式

util.isArray。方法用于判断给定的对象是否为正则表达式，如果是就返回true,否则返 回 false o

第07行判断"/some regexp/”是否为正则表达式，根据正则表达式的定义，可以判断 出该行将返回true；第08行判断'anotherregexp'是否为日期格式类型，同样我们可以判断该 行将返回true；第09行判断“ {}”是否为日期格式类型，根据正则表达式的定义，同样也 可以判断出该行将返回falseo

图13.5所示是本例程在Ubuntu环境下输出的结果。从图13.5所示输出的结果可以看 到，我们对第07〜09行代码的判断与通过utiLisRegExp0方法判断后的结果是一致的。

Run

Run chi3.utiLisRegExp.j5

费 :/usr/local/bin/node chl3.util\_isReg£xp.js

' util isRegExpO

霾毛.

—true 囹殍true

:曲；false

会 ：Process finished with exit code 0

X ! i| ? ® I

图13.5验证是否为正则表达式的方法

说明：正则表达式(Regular Expression )使用单个字符串来描述、匹配一系列符合某 个句法规则的字符串，在字符串编程实践中是重要且实用的一项功能。

13.6验证是否为错误类型

本节我们向读者介绍如何验证一个对象是否为错误类型的方法。当我们需要对一个错 误对象进行操作时，该方法是非常实用的，Node.js框架的常用工具(Util)模块提供了一 个util.isError()方法来实现该功能。

本例程(代码文件名称：ch 13.util isError.js)的主要代码如下：

01 /\*\*

02 \* ch13.util\_isError.js

03 •/

04 console.info(H-— util isError() ");

05 console.info();

//引入常用工具(Util)模块

//true

//true

// false

06 var util = require('util');

07 console.log(util.inspect(util.isError(new Error())));

08 console.log(util.inspect(util.isError(new TypeError())));

09 console.log(util.inspect(util.isError({

1. name: 'Error1,
2. message: 'an error occurred'
3. ))))；

**【代码分析】**

第07〜09行通过调用util.isError()方法来判断给定的对象是否为错误类型，关于

util.isError()方法的语法如下：

语法：util.isError(object); *II*判断给定的对象是否为错误类型

util.isError()方法用于判断给定的对象是否为错误类型，如果是就返回true,否则返回 false <,

第07行判断“new Error。”是否为错误类型，我们可以判断出该行将返回true；第08 行判断"new TypeError()”是否为错误类型，同样可以判断出该行也将返I可true；第09行 判断一个JSON数组是否为错误类型，同样也可以判断出该行将返回falseo

图13.6所示是本例程在Ubuntu环境下输出的结果。从结果可以看到，我们对第07〜 09行代码的判断与通过util.isError。方法判断后的结果是一致的。

Process finished with exit code 0

X

true false

**cM3.utn\_isErrof.js**

/usr/local/bin/node ch!3.util\_isError ]s  
util isErrorO

图13.6验证是否为错误类型的方法

说明：Error()错误类型与TypeError()错误类型均用于表述对错误的定义，其中Error() 为通用的错误类型，而TypeError()为类型错误的错误类型\*

13.7格式化字符串

本节我们向读者介绍格式化字符串的方法，Node.js框架的常用工具(Util)模块提供 了--个utiLformat()方法来实现该功能。

本例程(代码文件名称：chl3.util\_fbrmat.js)的主要代码如下：

01 /\*\*

02 \* ch13.util\_format.js

03 \*/

04 console.info(" util format()—");

05 console.info();

06 var util = require('util'); 〃引入常用工具(Util)模块

07 utjl.fbrmat(\*%s:%s,t 'foo');

08 util.format('%s:%s', 'too', 'bar1, 'baz');

09 util.format(1, 2, 3);

**【代码分析】**

第07〜09行通过调用util.formatO方法来格式化字符串，关于util.format。方法的语法 如下：

语法：util.format(format, //格式化字符串

util.format()方法用于根据第一个参数，返回一个格式化字符串，类似printf()方法的格 式化输出。

第07行表明使用util.formatO方法时，如果占位符没有相对应的参数，占位符将不会 被替换；第08行表明如果有多个参数占位符，额外的参数将会调用utiLinspect()转换为字 符串，这些字符串被连接在一起，并且以空格分隔；第09行表明如果第一个参数是一个非 格式化字符串，那么util.formatO将会把所有的参数转成字符串，并以空格隔开拼接在一块, 然后返回该字符串。

图13.7是本例程在Ubuntu环境下的结果。从结果可以看到，我们对第07〜09行代码 的判断与通过util.formatO方法操作后的结果是一致的。

Run

Run；" eft俱域心参 4 £>

> + /usr/local/bin/node chl3.util\_foraat.js

util fomatO

■，

foo：%s

典 A too：bar baz

X 3 123

Process finished with exit code 0

**X u**

? ©

图13.7格式化字符串的方法

关于占位符的说明:

* %s-字符串。
* %d-数字(整型和浮点型)。

. %j-JSON,如果这个参数包含循环对象的引用，将会被替换成字符串'[Circular]'3 . %%-单独一个百分号(％'),不会消耗一个参数。

第14章 Node.js与MySQL交互•

这一章我们向读者介绍应用Node.js框架与MySQL数据库进行交互的内容，通常在 Node.js开发中经常涉及到操作数据库，作为目前应用最为广泛的互联网开源数据库一一 MySQL,则成为了 Node.js数据库开发的首选，本章主要介绍了如何通过Node.js框架来操 作MySQL数据库。

众多高水平的开源设计人员为Node.js框架开发了 MySQL扩展库，读者可以在网联网 中检索到这些MySQL扩展库的介绍，本章选用的是目前人气最高的node-mysql开源项目 作为Node.js框架MySQL扩展库。可以说，node-mysql开源项目提供了 MySQL数据库对 Node.js框架的完整支持，具有一套与数据库开发相关的方法，通过这些方法就可以非常方 便地构建Node.js数据库应用。

本章主要包括以下内容。

* 连接MySQL数据库操作。

. 查询MySQL数据库操作。

* 修改MySQL数据库操作。
* 插入MySQL数据库操作。
* 删除MySQL数据库操作。
* MySQL数据库连接池。

1. 连接MySQL数据库

首先，我们从Node.js框架连接MySQL数据库的操作开始，如果想对数据库进行操作, 必须要先成功连接到数据库。从node-mysql开源项目的文档中，我们知道node-mysql扩展 库提供了一个mysql.createConnectionO方法来完成连接MySQL数据库的功能。在本节这个例 程中，将使用mysql.createConnection()方法来实现Node.js框架连接MySQL数据库的操作。

本例程ch 14.mysql conn.js主要代码如下：

01 /\*\*

02 \* ch 14.mysql\_conn.js

03 \*/

04 console.info(M—— mysql connnection() -——,

05 console.info();

06 var http = require(Hhttpn); 〃引入 HTTP 模块

|  |  |  |  |
| --- | --- | --- | --- |
| 07 | var mysql = require(7usr/local/lib/node\_modules/mysqr)； 〃引入 mysql 模块 | | |
| 08 | console.log('\*Now start HTTP server on port 6868..."); | | |
| 09 | console.info(); | |  |
| 10 | /\*\* |  |  |
| 11 | \* | 创.建数据库连'接 |  |
| 12 | \*/ |  |  |
| 13 | var connection = my sq I. create Con nection (( | | |
| 14 |  | host: "localhost", | 〃主机地址 |
| 15 |  | user: "root", | 〃数据库用户名 |
| 16 |  | password: "root'1, | 〃数据库密码. |
| 17 |  | database: Hnodejs", | 〃数据库名称 |
| 18 |  | port: 3306 | 〃端P号 |
| 19 | }); |  |  |
| 20 | /\*\* |  |  |
| 21 | \* | 创建HTTP服务器 |  |
| 22 | \*/ |  |  |

1. http.createServer(function (req, res) (
2. res.writeHead(200, ("Content-Type": "text/html;charset=utf8n));
3. res.write(M<h3>测试 Node.js - MySQL 数据库连接!<Zh3><br/>H);
4. /\*\*
5. \*测试数据库连接
6. \*/

|  |  |
| --- | --- |
| 29 | connection.connect(function(err) ( |
| 30 | if(err) ( |
| 31 | res.end('<p>Error Connected to MySQL!</p>'); |
| 32 | return; |
| 33 | ) else { |
| 34 | res.end('<p>Connected to MySQL!</p>'); |
| 35 | } |
| 36 | }); |

37 )).listen(6868); //监听 6868 端口号

**【代码分析】**

第06行引入HTTP模块，同时赋于变量http；第07行引入mysql模块，同时赋于变 量mysql；第13〜19行通过调用mysql.createConnection()方法创建了一个基本的MySQL数 据库连接，包括对数据库主机地址、数据库用户名与密码、数据库名称以及数据库监听端 口号的定义；同时，该方法返回一个connection变量用于保存数据库连接；第23~37行通 过调用http.createServer()方法创建了一个基本的HTTP服务器，其回调函数传递了两个参 数(req和res)；第24行通过调用res.writeHead()方法写HTTP文件头，并定义了页面字 符编码为utf8»

第25行通过调用res.write()方法向页面输出了提示信息；第29~36行通过调用 connection.connectO方法测试服务器连接，其回调函数包含一个参数err,用于保存连接错 误信息；第31行与第34行通过调用res.endQ方法发送页面内容，并通知HTTP服务器消息完成；如果数据库连接错误，则通过if语句判断参数err时为真，第31行被执行并向页 面输出错误信息；如果数据库连接成功，则通过if语句判断参数err时为假，第34行被执 行并向页面输出成功信息；第37行通过调用server.listenO方法在指定的主机名和端口接收 连接，并监听该端口的连接请求。

图14.1是本例程在Ubuntu环境下的结果。从图中可以看到，我们创建的HTTP服务 器已经启动运行，并打印输出了…行提示信息。

然后打开浏览器，并在地址栏中输入以下地址：http://127.0.0.1:6868,如图14.2所示。 从图中可以看到，Node.js框架连接MySQL数据库的操作成功完成了，第34行打印输出的 信息成功显示出来了。



Run 盲 <hM.mysql\_CDnn.)$

/usr/local/bm/node ch!4 nysql conn )s

mysql connnectionO

*犬r* x方 ? \*

Now start HTTP server on port 6868.

图14.2连接MySQL数据库(浏览器)

图14.1连接MySQL数据库(服务器)

说明：如果想成功调试本节的例程，我们需要先成功安装部署MySQL数据库，以及 node-mysql扩展库；关于MySQL数据库的相关内容，读者可去参考这个网址： http://wiki.ubuntu.org.cn/MySQL;关于node-mysql扩展库的相关内容，读者可去参考这个 网 址:https://github.com/felixge/node-mysql

1. 查询MySQL数据库

这一节我们介绍Node.js框架查询MySQL数据库的操作方法，在连接数据库操作成功 后，就可以查询数据库中的数据了。从node-mysql开源项目的文档中，我们知道node-mysql 扩展库提供了 一个connection.query()方法来完成查询MySQL数据库的操作。在本节这个例 程中，将使用connection.query()方法来实现Node.js框架查询MySQL数据库的功能。

本例程chl4.mysql\_query.js主要代码如下：

01 /\*\*

02 \* ch14.mysql\_query.js

03 7

04 console.info(H mysql query() ”)；

05 cx)nsole.info();

06 var http = require("http”)； 〃引入 HTTP 模块

07 var mysql = require(7usr/local/lib/node\_modules/mysqr); *//引* 入 mysql 模块

//……此处省略12行代码，参考14.1节

20 /\*\*

1. \*创建HTTP服务器
2. \*/
3. http.createServer(function (req, res) {
4. connection.query('select \* from userinfo;', function (error, rows, fields) (
5. res.writeHead(200, ("Content-Type": "text/html;charset=utf8M));
6. res.write(n<h3>测试 Node.js - MySQL 数据库查询操作!</h3><br/>n);
7. res.end(JSON.stringify(rows));
8. })；
9. }).listen(6868); 〃监听 6868 端口号

**【代码分析】**

第24〜28行通过调用connection.queryO方法测试数据库查询操作，其第一个参数为一 条SQL查询语句(select \* from userinfo;),其中userinfb为数据库nodejs中定义好的一张 表(table)；其第二个参数为一个包含三个参数的回调函数，参数error为错误信息，参数 rows为查询返回行的结果，参数fields为查询返回字段的结果。

图14.3是本例程在Ubuntu环境下的结果。从中可以看到，我们创建的HTTP服务器 已经启动运行，并打印输出了一行提示信息。然后打开浏览器，并在地址栏中输入以下地 址：http://127.0.0.1:6868,如图14.4所示。从图中可以看到，Node.js框架查询MySQL数 据库的操作成功完成了，第27行成功打印输出了 userinfb表中存储的内容。

127.0.0.1:6868 - Chromium

Run

127.0.0.1:6868

G 十 Zusr/local/bin/node ch!4.«ysql\_query.js

nysql queryO

**■ 4**

Now start HTTP server on port 6868 ,

**x**曾

? 咨

图14.3查询MySQL数据库（服务器）

0 J 127.0.0.1

测试**Node.js - MySQL**敷提庠查询操作!

[|"id": 1 ."name"：Hkingn.,'pwd,'" 12345t»78M:]

图14.4查询MySQL数据库(浏览器)

1. 插入MySQL数据库

这一节我们介绍Node.js框架插入MySQL数据库的操作方法，在需要将新的数据项添 加入数据库存储时，就会用到插入数据库的操作了。从node-mysql开源项目的文档中，我 们知道node-mysql扩展库提供了一个connection.query()方法来完成插入MySQL数据库的 操作。在本节这个例程中，将使用connection.queryO方法来实现Node.js框架插入MySQL 数据库的功能。

本例程ch 14.mysql\_insert.js主要代码如下：

01 /\*\*

02 \* ch14.mysqljnsert.js

03 \*/

04 console.infof mysql insert() ");

05 console.info();

06 var http = require(nhttpH); 〃引入 HTTP 模块

07 var mysql = require(7usr/local/lib/node\_modules/mysqr); //引 入 mysql 模块

//……此处省略连接数据库的代码，可参考本书源代码

1. connection.queryfinsert into userinfo(id,name,pwd) values(7,"genius","12345678”);', function (errorinsert, resinsert) (
2. if (errorinsert) console.log(errorinsert);
3. console.logC'INSERT Return ==>");
4. console.log(resinsert);

//……此处省略连接数据库的代码，可参考本书源代码

**【代码分析】**

第27~30行通过调用connection.query（）方法测试数据库插入操作，其第一个参数为一 条 SQL 插入语句（insert into userinfo（id,name,pwd） values（7,"genius"," 12345678"）;）,其中 userinfo 数据库nodejs中定义好的一张表（table） , values括号内为新插入的数据；其第 二个参数为一个包含两个参数的回调函数，参数errorinsert为错误信息，参数resinsert为插 入数据库操作返回的结果。

图14.5是本例程在Ubuntu环境下的结果。从图中可以看到，我们创建的HTTP服务 器巳经启动运行，并打印输出了一行提示信息。然后打开浏览器，并在地址栏中输入以下 地址：http://localhost:6868,如图14.6所示。从图14.6中可以看到，Node.js框架插入MySQL 数据库的操作成功完成了（ “id”=7这一行为新插入的数据）。最后再返回查看一下服务 器端有什么变化，如图14.7所示。从中可以看到，第30行成功打印输出了插入数据库操 作完成后的返回值。

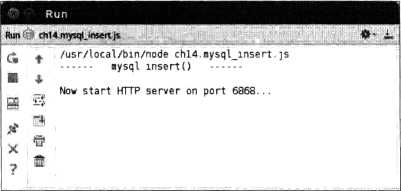


图14.5插入MySQL数据库（服务器）

<http://localhost:6868/> - QupZtlla

《 渤 ©佥I 口慷ir - ® \* Q

Qupzilla

, <http://localhost:6868/> 就咎 v

测试**Nodejs- MySQL**数据库插入操作！

({"id": 1 ,"name":"kin「"pwd":" I2345O7X"

| "id" :〕famc":"wang"."pwd”:" 1234567铲 |.

{" id" :3." name-:"jin"pwd" 12345678"}. r‘id":4.”Rame":"zhuypwd":" 12345078"[,

! ”id”:5.”namc":"gcnius”,"pwd”:” 12345678"

2345078"}.

r'id":7.”name":"gue8l","pwd":" 12345o78H •]

kxalhQSA (127.0.0, T)

INSERT Return => { ■fieldCount i 0, affectedRovs： 1. insertld； 7, serverstatus： 2, wamingCount： Q, message：''. protocol41： true. changedRows： 0 }

nysql insenC；

Mov start HTTP server on port 6868...

1. 删除MySQL数据库

这一节我们介绍Node.js框架删除MySQL数据库的操作方法，在需要将原始数据项删 除出数据库时，就会用到删除数据库的操作了。从node-mysql开源项目的文档中，我们知 道node-mysql扩展库提供了一个connection.query()方法来完成删除MySQL数据库的操作。 在本节这个例程中，将使用connection.query。方法来实现Node.js框架插入MySQL数据库 的功能。

本例程chl4.mysql\_delete.js主要代码如下：

01 /\*\*

02 [[11]](#footnote-12) ch14.mysql\_delete.js

03 7 •

04 console.info(" mysql delete() -——");

05 console.info();

06 var http = require("http"); 〃引入 HTTP 模块

07 var mysql = require(7usr/local/lib/node\_modules/mysqr); 〃引入 mysql 模块

//……此处省略连接数据库的代码，可参考本书源代码

1. connection.query(\*delete from userinfo where id=7;\ function (errordelete, resdelete) (
2. if (errordelete) console.log(errordelete);
3. console.logCDELETE Return ==> ")；
4. console.log(resdelete);

*II*……此处省略连接数据库的代码，可参考本书源代码

**【代码分析】**

第27-30行通过调用connection.query()方法测试数据库删除操作，其第一个参数为• 条 SQL 删除语句(delete from userinfb where id=7;),其中 userinfo 为数据库 nodejs 中定义 好的一张表(table) , “id=7”为要删除的数据项；其第二个参数为-个包含两个参数的 回调函数，参数errordelete为错误信息，参数resdelete为删除数据库操作返I可的结果。

图14.8是本例程在Ubuntu环境下的结果。从图中可以看到，我们创建的HTTP服务 器已经启动运行，并打印输出了一行提示信息。然后打开浏览器，并在地址栏中输入以下 地址：http:〃localhost:6868,如图14.9所示。从图14.9中可以看到，Node.js框架删除MySQL 数据库的操作成功完成了( “id”=7这一行数据被成功删除了)。最后，我们再返回查看 一下服务器端有什么变化，如图14.10所示。从中可以看到，第30行成功打印输出了删除 数据库操作完成后的返回值。

**Run**

G 金 /usr/local/bm/node chl4 «ysql\_delete.is  
 mysql delete()

**» 4** Now start rfTTP server on port 6868 ..



测试Node.js - MySQL数据库删除操作!

H "id”: 1 ."name":"ldng”,”pwd":" 12345678-；.

| “id":2,"ntinK”:”wang","pwd”：" 12345678° ；, rid":3."name”:"jin”,"pwd”:" 12345078"}.

! "id”.4.”nanw":"zhu”."pwd":” 12S45&78-1.

「'id” :5,"name”:"gcniu$"."pwd":" 12345678"}. 「'id"：o,"namc”:"super'."pwd":" 12345078" |j

Now start HTTP server on port 6868.-.

DELETE Return => { fieldCount： 0. affectedRows： 1. insertld： 0, serverstatus： 2, wamingCoiint： 0. message：'', protocols： true, changedRows: G }

Zusr/iocal/bm/node chl4,(nysqL\_delete.]s mysql deleteO

locathost (127,001)的

图14.9删除MySQL数据库(浏览器)

图14.10删除MySQL数据库(服务器)

1. 更新MySQL数据库

这一节我们介绍Node.js框架更新MySQL数据库的操作方法，在需要修改原始数据项 的内容时，更新数据库操作是最常使用的方法。从node-mysql开源项目的文档中，我们知 道node-mysql扩展库提供了一个connection.query()方法来完成更新MySQL数据库的操作。 在本节这个例程中，我们使用connection.query()方法来实现Node.js框架插入MySQL数据 库的功能。

本例程ch 14.mysql\_update.js主要代码如下：

01 /\*\*

02 \* ch14.mysql\_update.js

03 \*/

04 console.info(" mysql update() ");

05 console.info();

06 var http = requi「e(”http。； 〃引入 HTTP 模块

07 var mysql = require(7usr/local/lib/node\_\_modules/mysqr); 〃引 入 mysql 模块

//……此处省略连接数据库的代码，可参考本书源代码

1. connection.query('update userinfo set pwd="8765432T' where pwd=n12345678";', function (errorupdate, resupdate) (
2. if (errorupdate) console.log(errorupdate);
3. console.log("Update Return ==> ”)；
4. console.log(resupdate);

//……此处省略连接数据库的代码，可参考本书源代码’

**【代码分析】**

第27〜30行通过调用connection.queryQ方法测试数据库更新操作，其第一个参数为一 条 SQL 删除语句(update userinfo set pwd="87654321" where pwd=" 12345678";),是将所 有pwd字段内容为“12345678”数值的更新为新的“87654321” ；其第二个参数为一个包 含两个参数的回调函数，参数errorupdate为错误信息，参数resupdate为更新数据库操作返 回的结果。

图14.11是本例程在Ubuntu环境下的结果。从图14.11可以看到，我们创建的HTTP服务器己经启动运行，并打印输出了一行提示信息。然后打开浏览器，并在地址栏中输入 以下地址：http://localhost:6868,如图14.12所示。从图14.12中可以看到，Node.js框架更 新MySQL数据库的操作成功完成了（全部pwd= "12345678”的数据项均被更新为pwd=

“87654321 ” 了）。最后再返回查看一下服务器端有什么变化，如图14.13所示。从图中 可以看到，第30行成功打印输出了更新数据库操作完成后的返回值。



Nov start HTTP server on port 6868,,,

/usr/local/bin/node chl4.raysql\_update.js

mysql update 0

图14.11更新MySQL数据库（服务器）

<http://localhost:6868/> - QupZIlla

《 .localhost 0868/ ☆- 口- **Q**

QupZilla

i 1 <http://locaiho5t:6868/> X v

测试**Nodejs- MySQL**数据库更新操作！

Run

H”idT ."namc”:”king”,”pwd”：.87©54321 ”.

；"ki"：2."name":"wang"."pwd":"8765432rj.

|"id":3."namc":"jin".-pwd"^87o5432r

{”id・：4.”namc”:"zhu”,”pwd":”87t>54321"}.

| ”id":5."name":"gcnius”."pwd":”87654321”.

{"id":0."namc”:"supCT"「pwd":"87654321"} j

localhost (127Q0.1) &

S璋 L

G \* /usr/local/bin/node chl4.iysql\_update js

・ysql update()

■ 4-

Nov start HTTP server on port 6868.

g目，

日 Update Return —>

弁• J-《fieldCount： 0.

号 affectedRows： 6.

X insertld： 0.

? ® serverstatus： 34,

wamingCount: 0, message； '(Rows Batched： 6 Changed： 6 Warnings O', protocol41: true, changedRows： 6 }

图14.12更新MySQL数据库（浏览器）

图14.13更新MySQL数据库（服务器）

14.6操作MySQL数据库连接池

这一节我们介绍Node.js框架MySQL数据库连接池的使用方法。数据库连接池负责分 配、管理和释放数据库连接，它允许应用程序重复使用一个现有的数据库连接，而不需要 每次重新建立一个；数据库连接池还可以释放超过最大空闲时间的数据库连接，来避免因 为没有释放数据库连接而引起的数据库连接遗漏。由此可见，数据库连接池技术能明显提 高对数据库操作的性能。

从node-mysql开源项目的文档中，我们知道node-mysql扩展库提供了 pool.getConnection（） 方法来建立数据库连接池,同时还是使用conn.query。方法来完成查询MySQL数据库的操作。

本例程chl4.mysql\_pool.js主要代码如下：

01 *I\*\**

02 \* ch14.mysqljDool.js

03 \*/

04 console.info（" mysql pool（） "）;

|  |  |
| --- | --- |
| 05 | console.info(); |
| 06 | var http = require("http")； 〃引入 HTTP 模块 |
| 07 | var mysql = require(7usr/local/lib/node\_modules/mysqr); //引入 mysql 模块 |
| //... | …此处省略12行代码，参考14.1节 |
| 20 | /\*• |
| 21 | •创建HTTP服务器 |
| 22 | \*/ |
| 23 | http.createServer(function (req, res) ( |
| 24 | /\*\* |
| 25 | \*获取数据库连接池 |
| 26 | \*/ |
| 27 | pool.getConnection(function (err, conn) ( |
| 28 | if(enj { |
| 29 | console.log(MPOOL ==> " + err); |
| 30 | console.Iog(); |
| 31 | ) |
| 32 | /\*\* |
| 33 | \*定义SQL查询语句 |
| 34 | \* @type (string) |
| 35 | \*/ |
| 36 | var selectSQL = 'select \* from userinfo1; |
| 37 | /\*\* |
| 38 | \*执行数据查询操作 |
| 39 | 7 |
| 40 | conn.query(selectSQL, function(err, rows) ( |
| 41 | if(err) { |
| 42 | console.log(err); |
| 43 | console.log(); |
| 44 | } |
| 45 | console.log(wSELECT =>"); |
| 46 | for (var i in rows) ( |
| 47 | console.log(rows[i]); |
| 48 | } |
| 49 | conn. release(); 〃释放数据库连接 |
| 50 | res.writeHead(200, ("Content-Type": ntext/html;charset=utf8M}); |
| 51 | res.write(n<h3>测试 Node.js - MySQL 数据库连接池操作!</h3><br/>"); |
| 52 | res.end(JSON.stringify(rows)); |
| 53 | )); |
| 54 | }); |
| 55 | )).listen(6868); 〃监听 6868 端口号 |

**【代码分析】**

第36行定义了一个SQL查询语句(select \* from userinfo)用于查询u；serinfo表中的内 容：第40〜53行通过调用conn.query。方法测试数据库仓询操作；第46〜48行通过使用for

循环遍历参数rows数据集中的数据，并在服务器端依次打印输出；第49行通过调用 conn.release。方法释放数据库连接。

图14.14是本例程在Ubuntu环境下的结果。从中可以看到，我们创建的HTTP服务器 己经启动运行，并打印输出了一行提示信息。

然后打开浏览器，并在地址栏中输入以下地址：http://localhost:6868,如图14.15所示。 从图中可以看到，Node.js框架MySQL数据库连接池的查询操作成功完成了，第52行打印 输出了 userinfb表中的内容。

<http://localhost:6868/> - QupZIlla

《 / ； localhost 5ata/ @- Q I

/usr/local/bin/node chl4. a/sq^pool. js nysql poolO

Now start HTTP server on port 6868..

Run

QupZilla

h«p://localhost6868/ X vl

测试**Nodejs- MySQL**数据库连接池操作！

[I ."fiamc" :"king"."pwdM:"8705432!"：.

【“id”:2."namc”:"wiing”.”psd”."87o54321 .

厂过:3."name-"jin"."pwd":"87O54321"|.

{name" "zhu"."pwd":"87054321"|. rid":5."namcu:"genius"."pwd":"87654321"|.

| "kF :"supcrVpwd"：”87o54321 ”} j

localhosl (127.0.0.1)**。**

**图**14.14 MySQL**数据库连接池（服务器）**

**图**14.15 MySQL**数据库连接池（浏览器）**

最后再返回查看一下服务器端有什么变化，如图14.16所示。从图中可以看到，第46〜 48行同样在服务器端打印输出了 userinfo表中的内容。



/usr/local/bin/node chl4.«ysql\_pool.js ・ysql poolO

Nov start HTTP server on port 6868...

SELECT =>

{ id： 1, nane { id： 2, name ( id： 3, name { id： 4, name ( id： 5, name { id： 6, nane

'king,, pwd： '87654321' ) ■wang', pvd： '876543211 ) 'jin', pvd: '87654321' }

'Zhu', pwd: ,87654321' ) ■genius'. pwd： '876543211 } 'super', pwd: '87654321' }

**图**14.16 Node.js**框架**MySQL**数据库连接池（服务器）**

**说明：数据库连接池基本的思想是在系统初始化时，将数据库连接作为对象存储在内 存之中，这样当用户访问数据库时，并非新建立一个数据库连接，而是从内存中取出一个 已建立的空闲连接对象，使用完毕后也并不需要将连接关闭，而是将连接放回内存中，以 供下一个请求访问使用。这个使用内存来管理数据库建立连接与断开连接的载体就称为连 接池。**

第 15 章 Node.js 与 MongoDB 交互•

这一章我们向读者介绍应用Node.js框架与MongoDB数据库进行交互的内容。读者可 以将本章理解为是对上一章内容的延伸，不过需要明确的是，MongoDB与MySQL虽然都 是轻量级数据库中的佼佼者，但二者在本质上有很显著的区别。MySQL还是传统意义上的 关系型数据库（非常流行），但MongoDB确是地地道道的非关系型数据库，也就是大家 熟悉的NoSQL范畴。

下面简要介绍一下MongoDB数据库及其特点。

MongoDB是一种非关系型数据库（NoSQL范畴）,因其灵活的数据存储方式得到了 当前IT业内的一致肯定与支持，尤其在移动互联网开发方面是应用最广泛的NoSQL数据 库之一。

MongoDB最大的优势就是对面向对象的思想的完美实现，MongoDB数据库中的每一 条记录都是一个文档对象，因此对于MongoDB所有数据的持久操作都不需要开发人员手 动编写SQL语句，只需要直接调用方法就可以轻松地实现CRUD操作。

MongoDB作为NoSQL类型数据库与传统关系型数据库相比，可以说各有优势，也同 样各具不足。但对于移动互联网应用开发来讲，使用MongoDB这种类型的NoSQL非关系 型数据库，相比于传统关系型数据库，在技术先进性、操作简便性、效率优化性和速度提 高性这几方面优势还是很明显的，尤其是在处理大规模并发数据模型时，其优势更为突出， 是传统关系型数据库无法比拟的。随着近些年NoSQL技术的不断完善，完全性也在不断提 高，目前已经有很多大型门户网站和专业网站采用了 NoSQL非关系型数据库，其中 MongoDB数据库是企业的首选。

综上所述，将Node.js框架与MongoDB数据库完美结合起来进行应用开发，是能够将 各自优势最大化的方案，绝对是“1 + 1>2”的天作之合。在开源社区，众多高水平的设计人 员为Node.js框架开发了 MongoDB扩展库，读者可以在网联网中检索到这些MongoDB扩 展库的介绍，目前首选方案就是使用同名的mongodb开源项目作为Node.js框架的 MongoDB扩展库，该扩展库具有一套与数据库开发相关的方法，通过这些方法就可以非常 方便地构建Node.js数据库应用。

本章主要包括以下内容。

* 连接MongoDB数据库操作。
* 查询MongoDB数据库操作。
* 修改MongoDB数据库操作。
* 插入MongoDB数据库操作。
* 删除MongoDB数据库操作*。*

1. 连接 MongoDB数据库

首先，我们从Node.js框架连接MongoDB数据库的操作开始，如果想对数据库进行操 作，必须要先成功连接到数据库。从mongodb开源项目的文档中，我们知道mongodb扩展 库提供了一个db.open()方法来完成连接MongoDB数据库的操作。如果想使用mongodb扩 展库，就先要安装它，安装方法与其他Node.js扩展库类似，这里就不详细阐述了。然后在 MongoDB数据库中新建一个名称为nodejs的数据库，用于测试数据库连接操作。

本例程ch 15,mongodb\_conn.js主要代码如下：

|  |  |  |
| --- | --- | --- |
| 01 | /\*\* |  |
| 02 | • ch15.mongodb\_conn.js |  |
| 03 | \*/ |  |
| 04 | console.info(M mongodb connnection() "); | |
| 05 | console.info(); |  |
| 06 | var http = requirefhttp"); | 〃引入HTTP模块 |
| 07 | var mongodb = require(7usr/local/lib/node\_modules/mongodb');//TODO:引入 mongodb | |
| 08 | console.log("Now start HTTP server on port 6868...”)； | |
| 09 | console.info(); |  |
| 10 | /\*\* |  |
| 11 | \*创建数据库服务器连接 |  |
| 12 | \*/ |  |
| 13 | var server = new mongodb.Server( |  |
| 14 | 'localhost. | 〃主机地址 |
| 15 | 27017, | //端口号 |
| 16 | { |  |
| 17 | auto\_reconnect: true | 〃自动重连 |
| 18 | } |  |
| 19 | ); |  |
| 20 | ,\* |  |
| 21 | \*创建数据库连接 |  |
| 22 | 7 |  |
| 23 | var db = new mongodb.Db( |  |
| 24 | 'nodejs', //数据库名称 | |
| 25 | server, | 〃数据库服务器 |
| 26 | ( |  |
| 27 | safe: true |  |
| 28 | } |  |
| 29 | ); |  |
| 30 | /\*\* |  |
| 31 | \*测试数据库连接 |  |
| 32 | 7 |  |
| 33 | db.open(function(err, db) ( |  |
| 34 | if(!err) { |  |

1. console.logflog - connect mongdb successfully!\*);
2. } else {
3. console.logClog -' + err);
4. }
5. })；
6. /\*\*
7. \*创建HTTP服务器
8. 7
9. http.createServer(function (req, res) (
10. res.writeHead(200, ("Content-Type": "text/htmI;charset=utf8" ))•
11. /\*\*
12. \*测试数据库连接
13. \*/
14. db.open(function(err, db) (
15. if(lerr) (
16. res.write('<p>Connected MongoDB:nodejs successfully!</p>,);
17. } else (
18. res.write(\*<p>Error Connected to MySQL!</p>\*);
19. console.log('log - \* + err);
20. }
21. res.end("<h5> end operations </h5><br/>");
22. ))；
23. }).listen(6868); 〃监听 6868 端口号

**【代码分析】**

第06行引入HTTP模块，同时赋于变量http；第07行引入mongodb模块，同时赋于 变量mongodb；第13~19行通过调用mongodb.Server()方法创建了一个基本的MongoDB 数据库服务器连接，包括对数据库主机地址、服务器端口(MongoDB服务器端口为27017) 号以及自动重连的定义；同时，该方法返回一个server变量用于保存数据库服务器连接。

第23〜29行通过调用mongodb.Db()方法创建了一个数据库连接，包括对数据库名称 (“nodejs” )、数据库服务器连接以及安全性的定义；同时，该方法返回一个db变量用 于保存数据库连接；第33〜39行通过调用db.open()方法打开数据库连接，其回调函数传递 两个参数，第一个err参数用于定义错误信息，第二个db参数用于定义数据库连接；第34 行通过对参数err进行判断，测试数据库连接是否成功，提示信息将在服务器端进行打印输 出。

第43〜57行通过调用http.createServerO方法创建了一个基本的HTTP服务器，其回调 函数传递两个参数(req和res)*；*第44行通过调用res.writeHead()方法写HTTP文件头， 并定义了页面字符编码为utf8；第48〜56行与第33〜39行类似，均是通过调用db.open() 方法打开数据库连接，不同的是连接测试的结果将通过res.write()方法在浏览器客户端打印 输出；第57行通过调用server.listen()方法在指定的主机名和端口接收连接，并监听该端口 的连接请求。

图15.1是本例程在Ubuntu环境下的结果。从图中可以看到，我们创建的HTTP服务

器己经启动运行，并打印输出了两行提示信息，表明数据库连接成功J'。然后打开浏览器， 并在地址栏中输入以下地址：http://localhost:6868.如图15.2所示。从图中可以看到,Node.js 框架连接MongoDB数据库nodejs的操作成功完成了，第50行打印输出的信息成功显示出 来了。



| [ Run 1 |
| --- |
|  |
| G . /usr/local/bin/node chl5.«ongodb\_conn.js nongodb connnectionO  圈 ■  Now start HTTP server on port 6868,..  圈目  « log - connect nongdb successfully I  X』  X冒  ? s |

图15.1连接MongoDB数据库(服务器)

图15.2连接MongoDB数据库(浏览器)

说明：如果想成功调试本节的例程，我们需要先成功安装部署MongoDB数据库，以 及用于Node.js框架的mongodb扩展库；关于MongoDB数据库的相关内容，读者可以参考 这个网址：http://www.mongodb.org

1. 连接MongoDB数据集合

上一节，我们介绍了 Node.js框架连接MongoDB数据库的操作，本节将介绍连接 MongoDB数据集合的操作。根据mongodb开源项目文档的介绍，我们知道mongodb扩展 库提供了两个方法来完成连接数据集合的操作，一个为db.collection()方法，另一个为 db.createCollectionO方法。在本节的例程中，需要先在之前已经创建的nodejs数据库中创建 一个名称为userinfo的数据集合，然后分别使用这两个方法对对userinfo数据集合进行连接 测试。

先看第一个使用 db.collection()方法的例程 ch 15.mongodb conn collection.js,其主要代 码如下：

01 /\*\*

02 \* ch15.mongodb\_conn\_collection.js

03 \*/

04 console.info("~ mongodb connnection collection() ");

05 console. info();

06 var http = require("httpM); //引入 HTTP 模块

07 var mongodb = require('/usi7local/lib/node\_modules/mongodb');//TODO:引入 mongodb 模块

//・••…此处省略33行代码，参考15.1节

1. /\*\*
2. \*创建HTTP服务器
3. 7
4. http.createServer(function (req, res){

res.writeHead(200, {"Content-Type” ： ,,text/html;charset=utf8n}); res.write(H<h3>测试 Node.js - MongoDB 数据库操作!</h3><br/>n);

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

/\*\*

\*测试数据库连接

\*/

db.open(function(err, db) (

if(!err) { res.write('<p>Connected MongoDB:nodejs successfully!</p>');

/\*\*

\*连接数据集合

\*/

db.collection('userinfo', (safe: true}, function(errcollection, collection) { if(!errcollection) (

res.write(\*<p>Connected MongoDB:nodejs:userinfo successfully!</p>,);

} else ( console.log(1og -' + err);

}

})；

} else { res.write('<p>Error Connected to MySQL!</p>'); console.Iog('log -' + err);

} res.end("<h5> end operations </h5><br/>");

})； }).listen(6868); 〃监听 6868 端口号

**【代码分析】**

第57~63行通过调用db.collection()方法连接MongoDB数据库数据集合；其中，第一 个参数userinfo为数据集合名称；第二个参数(safe: true｝用于保证操作成功后执行回调函 数；第三个参数为回调函数，该回调函数包含两个参数，一个用于定义错误信息，一个用 于返回数据集合。

图15.3是本例程在Ubuntu环境下的结果。从中可以看到，我们创建的HTTP服务器 已经启动运行，并打印输出了两行提示信息，表明数据库连接成功了。然后打开浏览器， 并在地址栏中输入以下地址：http://localhost:6868,如图15.4所示。从图中可以看到,Node.js 框架连接MongoDB数据库数据集合 userinfo的操作成功完成了,第59行 打印输出的信息成功显示出来了。最 后，我们将第57行的db.collectionQ 方法换成 db.createCollection()方法，在 浏览器地址栏中输入以下地址： http://localhost:6868,再次进行测试, 如图15.5所示。从图中可以看到，

**chi 5 marigc-Gb,conn,cclIeftion js**

登 :/usr/local/bin/node chl5, mongodb\_conn\_coUection. js mongodb connnection collection()

4 ,Now start HTTP server on port 6868.,.

ng ilog - connect mongdb successfully I

Node.js框架连接MongoDB数据库数据集合userinfo的操作同样成功完成了，新的提示信 息也成功显示出来了。

图15.4连接MongoDB数据集合（浏览器）

Ubuntu 起始员-Mozilla Firefox

〉正sam -

♦ # localhost ▼ x » 三

测试**Node.js - MongoDB**数据库操作！

Connected MongoDB:nodejs successfully! createCollection MongoDB:nodejs:userinfo successfully! 正在侍辎来自localhost的故徭.

图15.5连接MongoDB数据集合（浏览器）

说明：根据MongoDB数据库官方网站的介绍，MongoDB数据集合的概念相当于 MySQL数据库中表的概念，只不过MySQL数据库中的表是关系型的，而MongoDB数据 集合是非关系型的，

1. 查询MongoDB数据集合

这一节我们介绍Node.js框架查询MongoDB数据集合的操作方法，经过前面两节的基 础性介绍，在成功连接数据库后，就可以进行查询数据库的操作了。根据mongodb开源项 目文档的介绍，我们知道mongodb扩展库提供了一个collection.find()方法来执行查询操作。

在本节的例程中，将对之前创建的userinfo数据集合进行查询测试，向读者介绍 collection.find()方法的使用过程。

本例程 chl5.mongodb\_conn\_collection find.js 主要代码如下：

01 /\*\*

02 \* ch15.mongodb\_conn\_collection\_find.js

03 \*/

04 console.infof mongodb connnection collection find() ");

05 console.info();

06 var http = require(Hhttp"); //引入 HTTP 模块

07 var mongodb = requireC/usr/local/lib/node^odules/mongodb\*); //TODO:?| 入 mongodb 模块 //……此处省略33行代码，参考15.1节

1. /\*\*
2. •创建HTTP服务器
3. 7
4. http.createServer(function (req, res) {
5. res.writeHead(200, ("Content-Type": Mtext/html;charset=utf8"});
6. res.write("<h3>S'J试 Node.js - MongoDB 数据库操作!</h3><br/>");
7. /\*\*

\*测试数据库连接

\*/

db.open(function(err, db) (

if(!err) (

res.write('<p>Connected MongoDB:nodejs successfully!</p>\*);

db.collection('userinfo', (safe: true}, function(errcollection, collection) ( if(!errcollection) (

res.wnte(\*<p>Connected MongoDB:nodejs:userinfo successfully!</p>r);

\*查询数据集合，方式一

\*/

collection.find().toArray(function(errorfind, cols) (

if(!errorfind) (

res.wnte('<p>collection.find() is: </p>'); res.write(JSON.stringify(cols));

console.log(cols);

}

});

/\*•

\*查询数据集合，方式二

7

collection.find({userid:'king,}).toArray(function(errorfincl, cols) { if(!errorfind) (

res.writeC<p>collection.find(()) is: </p>')f res.write(JSON.stringify(cols));

console.log(cols);

,)

});

/\*\*

\*查询数据集合，方式三

7

collection.findOne((usemame:'king,}, function(errorfind, col) ( if(!errorfind) (

res.write('<p>collection.findOne((}) is: </p>')； res.write(JSON.stringify(col));

console.log(col);

)

))；

) else (

console.logflog -' + err);

}

})；

} else {

res.end('<p>Error Connected to MySQL!</p>');

console.Iog('log -' + err);

1. }
2. //res.end("<h5> end operations </h5><br/>");
3. });
4. }).listen(6868); 〃监听 6868 端口号

**【代码分析】**

第54行通过调用db.collection()方法连接userinfo数据集合，并将数据集合连接保存在 回调函数中的第二个参数collection中；第60〜66行为查询数据集合的第一种方式，即通 过调用collection.find().toArray()方法查询数据集合，并将查询结果转换为数组格式进行存 储；其回调函数中包含两个参数，第一个参数errorfind用于定义错误信息；第二个参数cols 用于保存查询到的文档对象数组；第63行通过调用JSON.stringify。方法将参数cols转换为 字符串格式数据，并在浏览器客户端进行打印输出；第64行将参数cols在服务器端进行打 印输出。

第70~76行为查询数据集合的第二种方式，仍旧是通过调用collection.find().toArray() 方法查询数据集合，只不过是在find()方法内增加了查询条件({userid： 'king' }),其 会将满足该条件的文档对象查询结果转换为数组格式进行存储；其回调函数中包含两个参 **数，**第一个参数errorfind用于定义错误信息；第二个参数cols用于保存查询到的文档对象 数组；第73行通过调用JSON.stringify()方法将参数cols转换为字符串格式数据，并在浏览 器客户端进行打印输出。

第80~86行为查询数据集合的第三种方式，该方式通过调用collection.findOne()方法 查询数据集合，在\* findOne ()方法内增加了查询条件({username： 'king' }),其会将满 足该条件的第一条文档对象查询结果进行存储；其回调函数中包含两个参数，第一个参数 errorfind用于定义错误信息；第二个参数col用于保存查询到的唯 条文档对象；第73 行通过调用JSON.stringifyO方法将参数col转换为字符串格式数据，并在浏览器客户端进 行打印输出。

图15.6是本例程在Ubuntu环境下的结果。从图中可以看到，我们创建的HTTP服务器 已经启动运行，并打印输出了一行提示信息。然后打开浏览器，并在地址栏中输入以下地址: http://localhost:6868,如图15.7所示。从图中可以看到，Node.js框架查询MongoDB数据库 数据集合(“userinfb” )的操作成功完成了，三种方式查询的结果也完全打印输出了。

最后，我们再返回到服务器界面，看看服务器端打印输出的结果如何，如图15.8所示。 从图15.8与图15.7的对比中可以看到，查询得到的结果是完全一致的，说明无论服务器端 还是客户端，操作的方法是一样的。



漆"；/usr/local/bin/node chl5.mongodb\_conn\_collection\_find.js

T ' mongodb connnection collection find() 一

Now start HTTP server on port 6868..

i 两:log - connect mongdb successfully!

Ubuntu jESfclH - Mozilla Firefox

Run

■ °正在连:搐 h '♦

♦ -i locdthosl "•.， ☆自 》 三

测试 **Node.js - MongoDB RSWft**作**I**

Connected MongoDB:nodejs successfully!

Connected MongoDB:nodep:userinfo successfully!

colleaion.fincK) is：

H' ld":"559482d \ f8d634ca8a 112d66"/usend":"kingH.'pwd":\*»23456\*."usemame■:"king^.

{■userid^Vang-.-pwd" "123456".'usemame":'wang".-\_id-："5595d4d1b7c118df0d73c436,}]

collection ,find((D is:

H"\_fd-："559482d1f8d634ca8a112d66-.-userid\*:Mking",,pwd':,123456'/usemame,:"king^J

collection.findOne({}) is:

r.ld":HS59482d1 f8d634ca8a112d66\*."userid"i1ing",'pwtf,:"123456M/usemame":"king^

正未目localhost的孜揖…

图15.7套询MongoDB数据集合(浏览器)

5 暮 **Ch15jnoflBodb\_com\_coileaiDnjini£js** 。・ **L**

G ? /usr/local/bin/node chi5.nongodb\_conn\_collection\_find.js mongodb connnection collection fmd()

B 4

— Now start HTTP server on port 6868..

囹弓

f log - connect nongdb successfully!

^ — ( { \_id 559482dlf8d634ca8all2d66.

言 usend： 'king',

X pwd： '123456'.

? El usernaae： 'king- },

{ userid： Sng，.

pvd： 123456'. usemane： \* wang'.

Id： 5595d4dlb7cll8df0d73c436 } !

I { \_id： 559482dlf8d634ca8aL12d66, userid： 'king', pwd； 123456-. username： 'king' }]

{ \_ld： 559482dlf8d634ca8all2d66. usend： 'king', pwd： •123456'. username： 'king' )

图15.8查询MongoDB数据集合(服务器)

1. 插入MongoDB数据集合

这一节我们介绍使用Node.js框架插入MongoDB数据集合的操作方法，在需要将新的 文档对象添加入数据库时，就会用到插入数据集合的操作了。根据mongodb开源项目文档 的介绍，我们知道mongodb扩展库提供了一个collection.insert()方法来执行插入操作。

在本节的例程中，将对之前创建的userinfo数据集合进行插入测试，向读者介绍 collection.insert ()方法的使用过程。

本例程 ch 15.mongodb\_conn\_collection\_insert.js 主要代码如下：

01 /\*\*

02 \* ch15.mongodb\_conn\_collection\_insert.js

03 \*/

04 console.info(H mongodb connnection collection insert() ”)；

05 console.info();

06 var http = require(nhttpM); //引入 HTTP 模块

07 var mongodb = require('/usr/local/lib/node\_modules/mongodb');//TODO:引入 mongodb 模块

//……此处省略连接数据库的代码，可参考本书源代码

1. var inserted = (userid: 'wang', pwd: '123456', username: 'wang');
2. /\*\*
3. \*插入数据集合
4. •/
5. collection.insert(insertCol, {safe: true}, function(errinsert, result) {
6. res.write(,<p>collection.insert() is: </p>');
7. res.write(JSON.stringify(result));
8. console.log(result);
9. ))；

*II*……此处省略连接数据库的代码，可参考本书源代码

**【代码分析】**

第60行定义了一个JSON格式的文档对象inserted,用于插入数据集合userinfo；第 64~68行通过调用collection.insert。方法执行插入数据集合的操作；该方法第一个参数为将 要插入的文档对象;第二个参数｛safe: true｝用于确保插入操作成功完成，如果不设定该参数， 则插入操作出错后无反馈提示；第三个参数为I可调函数，其包含两个参数，第一个参数 errorinsert用于定义错误信息，第二个参数result用于保存插入数据集合后的结果；第66 行通过调用JSON.stringify。方法将参数result转换为字符串格式数据，并在浏览器客户端 进行打印输出。

图15.9是本例程在Ubuntu环境下的结果。从图15.9可以看到，我们创建的HTTP服 务器已经启动运行，并打印输出了一行提示信息。然后打开浏览器，并在弛址栏中输入以 下地址：http://localhost:6868»如图15.10所示。从图中可以看到，Node.js框架插入MongoDB 数据集合的操作成功完成了，打印输出了一行提示信息（｛“ok” :1, “n” ：1｝）。

最后再返回查看一下服务器端有什么变化，如图15.11所示。从图中可以看到，第60 行定义的文档对象插入成功的信息在服务器端也显示出来了。



/usr/Tocal/bin/node chl5.mongodb\_conn\_collection\_insert.]s

mongodb connnection collection insert()

Nov start HTTP server on port 6868...

log - connect mongdb successfully I

图15.9 Node.js框架插入MongoDB数据集合（服务器）

Ubuntu 起官员-Mozilla Firefox



正在连接…

♦ & localhost \*8

亲试**Node.js • MongoDB**敷质库操作！

Connected MongoDB:node|s successfully!

Connected MongoDB:nodejs:userlnfo successfully!

collection.insert() is:

正在侍输来自localhost的散据...

**Run cht5.mw<odb\_conn.a)Ueaiarv.>n^-j\***

A . pause： [FunctionI.

resume： IFunction I.

read IFunction]. \_consu«ing true }. wnteStr«ani. null

ops：

I { userid. -wanq■, pwd： '123456•. username： 'wang'.

\_id； 55965bd43ee83e074e3a4551

Process finished with exit code 13Q

图15.10插入MongoDB数据集合（浏览器）

图15.11插入MongoDB数据集合（服务器）

说明：设置参数｛safe: true｝可以确保插入操作成功，因为设置该参数是可以对主键重 复的插入操作进行报错的；如果不设置参数｛safe: true｝,当发生错误但并没有报错，而且 回调函数还得到了反馈数据时，mongodb drvier将会直接转入回调函数，并且设置error 为nullo因此，如果要确保数据操作（insert/叩date/remove ）成功必须要设置参数｛safe: true｝ 选项。

1. 删除MongoDB数据集合

这一节我们介绍Node.js框架删除MongoDB数据集合的操作方法，在需要将数据集合 中的文档对象删除时，或者是删除整个数据集合时，就会用到删除数据集合的操作了。根 据mongodb开源项目文档的介绍，我们知道mongodb扩展库提供了一个collection.remove() 方法来执行删除操作。

在本节的例程中，将对之前创建的userinfo数据集合进行删除测试，向读者介绍 collection.remove()方法的使用过程。

本例程 chl5.mongodb conn\_collection\_remove.js 主要代码如下：

01 /\*\*

02 \* ch15.mongodb\_conn\_collection\_remove.js

03 \*/

04 console.info(" mongodb connnection collection remove() ");

05 console.info();

06 var http = requiref'http"); 〃引入 HTTP 模块

07 var mongodb = require(7usr/local/lib/node\_modules/mongodb');//TODO:引入 mongodb 模块

*II*……此处省略连接数据库的代码，可参考本书源代码

1. collection.remove((userid: "wangw}, (safe: true), function(errremove, count) (
2. res.write('<p>collection.remove() is: </p>');
3. res.write(JSON.stringify(count));
4. console.log(count);
5. })；

//……此处省略连接数据库的代码，可参考本书源代码

【代码分析】

第63~67行通过调用collection.remove()方法执行删除数据集合的操作；第71〜77行 通过调用collection.find()方法将userinfo数据集合中内容全部打印输出，查看一下删除操作 后的结果，该方法在15.3节中有详细介绍，读者可自行参阅。

图15.12是本例程在Ubuntu环境下的结果。从图中可以看到，我们创建的HTTP服务 器已经启动运行，并打印输出了一行提示信息。然后打开浏览器，并在地址栏中输入以下 地ilk： http://localhost:6868,如图15.13所示。从图中可以看到，Node.js框架删除MongoDB 数据集合的操作成功完成了 ({userid: wang}这条文档对象被成功删除了)，第65行打印输 出了 count参数值为1 o

最后再返回查看一下服务器端有什么变化，如图15.14所示。从图中可以看到，服务 器端第66行成功打印输出了 count参数的值，与前面浏览器客户端的结果是一致的。

说明：根据mongodb开源项目文档的介绍，collection.remove()方法可以删除选定的文 档对象，也可以删除整个数据集合；如果想删除整个数据集合时，collection.remove()方法 不设定删除条件，就可以执行删除整个数据集合的操作。

**C Mozilla Firefox**

刘 **Run**

**Run&W *i.***

G ? /tisr/local/bin/node ch!5 monqodb\_conn\_coUection\_rei»ove js

5 nonqodb connnectlon coUection removed

**■ 4**

\_ Now start HTTP server on port 6968 .

F log - connect manqdb successfully!

, ° 正

令locjlhost • x < 况 i： ☆碧 》 三

测试**Node.js - MongoDB**敷据棒操作I

Connected MongoDB:nodejs successfully\*

Connected MongoDB:node|s:userinfo successfully'

collection.remove() is:

collection.findO is:

If Jd":"559482d1f8d634ca8a112d66"：'u5erid',:',king",-pwd','-123456".,'username":,,king-h ru5eridH:nsuper-."pwd"',123456,,,"usetname"rsuper；\_JdK:H55965d1f3a0d5fed57d9f08d'}) 正任僖罐&自locelhost的ttia..

图15.12删除MongoDB数据集含(服务器)

图15.13删除MongoDB数据集合(浏览器)



/usr/local/bin/node ch!5 Bongodb\_conn\_coHection\_rewove is lonqodb connnection collection remove。

Now start HTTP server on port 6868..

log - connect mongdb successfully!

( result： {ok： 1. m 1 }.

connection：

( domain： null,

\_events；

{ close- [Object], error： IObject], timeout： [Object]. parseError： (Object!. connect： IFunction] },

\_maxListeners： 10.

图15.14删除MongoDB数据集合(服务器)

1. 更新MongoDB数据集合

这一节我们介绍Node.js框架更新MongoDB数据集合的操作方法，在需要修改文档对 象的某项内容时，更新数据集合的操作是最实际的方法。根据mongodb开源项目文档的介 绍，我们知道mongodb扩展库提供了一个collection.update()方法来执行更新操作。

在本节的例移中，将对之前创建的userinfo数据集合进行更新测试，向读者介绍 collection.update ()方法的使用过程。

本例程 ch 15.mongodb\_conn\_collection update.js 主要代码如下：

01 /\*\*

02 \* ch15.mongodb\_conn\_collection\_update.js

03 \*/

04 console.info(" mongodb connnection collection update() ");

05 console.info();

06 var http = require(MhttpM); //引入 HTTP 模块

07 var mongodb = require(7usr/local/lib/node\_modules/mongodb');//TODO:引入 mongodb 模块 //……此处省略连接数据库的代码，可参考本书源代码

63 collection.update((userid: "king\*\*), {$push: {email:king@email.com\*}},

function(errupdate, cols) (

|  |  |
| --- | --- |
| 64  65  66  67  68 | res.write(,<p>collection.update() is: </p>\*);  res.write(JSON.stringify(cols));  console.log(cols);  })；  collection.update({userid: "super"}, ($push: (email:,super@email.com}), function(errupdate, cols) ( |
| 69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86 | res.write('<p>collection.update() is: </p>');  res.write(JSON.stringify(cols));  console.log(cols);  })；  /\*■  \*查询数据集合  \*/  collection.find().toArray(function(errorfind, cols) (  if(!errorfind) (  res.write('<p>collection.find() is: </p>\*);  res.write(JSON.stringify(cols));  console.log(cols);  }  })；  /\*\*  \*更新数据集合，方式二  7  collection. update((userid: "king"), ($set: {username:king-update'}), function(errupdate, cols) ( |
| 87  88  89  90  91 | res.write('<p>collection.update() is: </p>');  res.write(JSON.stringify(cols));  console.log(cols);  })；  collection.update((userid: "super"}, ($set: (username/super-update'}}, function(errupdate, cols) { |
| 92  93  94  95  96  97 | res.write('<p>collection.update() is: </p>');  res.write(JSON.stringify(cols));  console.log(cols);  });  \*查询教据集合 |

1. \*/
2. collection.find().toArray(function(errorfind, cols) (
3. if(!errorfind) (
4. res.writeC<p>collection.find() is: </p>');
5. res.write(JSON.stringify(cols));
6. console.log(cols);
7. )
8. })；

//……此处省略连接数据库的代码，可参考本书源代码

**【代码分析】**

第63〜67行通过调用collection.update()方法对userinfb数据集合执行更新数据集合的 操作；第68~72行再次通过调用collection.update()方法执行更新数据集合的操作，向 userinfo数据集合中｛userid: Ms叩er”｝的这一条文档对象同样增加一个email字段，内容为 "s叩er@email.com” ；第76〜82行通过调用collection.find()方法将userinfo数据集合中内 容全部打印输出。

第86〜90行通过调用collection.update()方法对userinfo数据集合执行了另一种方式的 更新数据集合的操作；该方法第一个参数为设定更新文档对象的条件，本例程中设定为 (｛userid: "king”｝)的文档对象，即刚刚更新过的文档对象；第二个参数｛$set: (usemame:king-叩date，｝｝ ｝用于将该条文档对象的username字段更新为king-update，其中$set 关键字代表更新设定的含义。

第91~95行再次通过调用collection.update()方法执行更新文档对象的操作，将userinfo 数据集合中｛userid: ”s叩er”｝这一条文档对象的usename字段更新为super・update；第99〜105 行再次通过调用collection.find()方法将userinfo数据集合中内容全部打印输出。

图15.15是本例程在Ubuntu环境下的结果。从中可以看到，我们创建的HTTP服务器 己经启动运行，并打印输出了一行提示信息。然后打开浏览器，并在地址栏中输入以下地 址：http://localhost:6868,如图 15.16 所示。

**Mozilla Firefox**

Run

T — /usr/local/bin/node chl5. Bongo(jb\_conn\_coUection\_update.)s

1 mongodb connnection collection update()

一 Nov start HTTP server on port 6868., 昭弓

m log - connect ・ongdb successfully I

■。正在匹植… \*

♦ \* locaihost oBuH » x ☆也

Connected MongoDB:nodejs:userinfo successfully)

collect! on.updatef) Is:

rok"：i,wn"：n

collection.updateO is:

collection.findO Is:

({MdH;"559482dirad634ca8a112d66H."emall"; ridng@email.com1,"pwd":"1234S67userid"：-king-.，username-;\*ing\*y ("」cr：”S596Sd1f3a0dSfed57d9f0&r.-emair：

rsuper@email.com\*l,\*pwd':\*123456" ■\*usend':-super-."usemame";'super\*)]

collection.update() is:

Cok-：1,"n";1}

collectlon.flnd() is:

H”」d":”559482d 1 f8d634ca8a 112d66','emallB: rking@email.coml,\*pwd":\*123456',,u5end■■:■ktng\*,"username":\*king-update,'). r.KJT55965d1 f3a(M5fe<157d9ro8d-."emair； rsuper@email.com"],"pwd\*:"123456\*,,'user»(r："super'■usemame,:"super-updatel)

coilection.update() is：

图15.15更新MongoDB数据集合(服务器)

图15.16更新MongoDB数据集合(浏览器)

从图15.16中可以看至U, Node.js框架更新MongoDB数据集合的操作成功完成了(email 字段成功添加进去了)，第79行与第102行成功打印输出了 userinfo数据集合中更新后的 内容。最后再返回查看一下服务器端有什么变化，如图15.17与图15.18所示。从图中可以 看到，第80行与第103行在服务器端成功打印输出了更新数据库操作完成后的返回值。

第 15 章 Node.js 与 MongoDB 交互・ 339 ••・\*

Run 早 cm》inongo<8^ccHrvgeciiorLup(ine^

vntebtrean： null })

[( .id： 559482dlf8d634ca8ail2d66. email： ( •king^emaiLcom\* I. pwd： \* 123456 \ userid： 'king', username： 'king' },

{ .id： 55965dlf3aQd5fed57d9f08d. email： I 'super^email com' J. pwd: \*123456\*, userid： 'super\* e username 'super' }]

{ result ( ok, 1. n： 1 }.

弓

r?

stubduTter： null > j

\_M： 555482dlf8d634ca8all2d66l emaili [ \*king^euail.com\* I. pvd '123456'.

userid； 'king', usernaaie： 'kxng-update\* }.

\_id： 55965dlf3a0d5fed57d9f08d, email: [ \*super(aeinail.com' I, pvd： \* 123456\*, userid 'super'.

username： 'super-update\* } I

( result： { ok： 1, n： 1 }, connection：



图15.17更新MongoDB数据方式一（服务器） 图15.18更新MongoDB数据方式二（服务器）

1. Debugger 回 Console •• (9 Scripts •・ hS U ' [↑](#footnote-ref-2)
2. define variable

   * @type {Buffer}

   [↑](#footnote-ref-3)
3. \*/

   var buf = new Buffer("this is Buffer concat test!");

   var str = "this is String concat test!";

   /\*\*

   * start record time

   7

   console.timef'buffer concat test!");

   var list = □; [↑](#footnote-ref-4)
4. **s**

   * **f "D:\Program Files\3etBrains\WebStor« 9.0.1\ I**

   **. & OS release()**

   国弓操作系统类型：**Windows\_NT 3**

   **X @** 操作系统发行版本：**6.1.7601**

   **-ffi**  [↑](#footnote-ref-5)
5. **OS release()**

   **Process finished with exit code 0**

   图7.10获取操作系统发行版本(Windows版) [↑](#footnote-ref-6)
6. server.on(\*error\*, function(err) ( console.logfError occurred:', err.message); [↑](#footnote-ref-7)
7. 监听error事件 \*/ [↑](#footnote-ref-8)
8. write data to request body [↑](#footnote-ref-9)
9. 调用util.inherits。方法实现原型对象继承 [↑](#footnote-ref-10)
10. 7

    util.inherits(Child, Base);

    /\*\*

    \*定义基类Base对象

    * @type (Base)

    7 ，

    var objBase = new Base();

    objBase.showName();

    objBase.showYear();

    objBase.sayHello();

    console.log(objBase);

    /\*\*

    •定义子类Child对象

    * @type {Child}

    \*/

    var objChild = new Child();

    objChild.showName();

    objChild.showYear();

    //objChild.sayHello();

    console.log(objChild); [↑](#footnote-ref-11)
11. *K*

    X旨 [↑](#footnote-ref-12)