

20

第

2

章 安装和配置

Node.js

Commands:

n *Output versions installed*

n latest [config ...] *Install or activate the latest node release*

n <version> [config ...] *Install and/or use node <version>*

n use <version> [args ...] *Execute node <version> with [args ...]*

n bin <version> *Output bin path*

for

<version>

n rm <version ...> *Remove the given version(s)*

n --latest *Output the latest node version available*

n ls *Output the versions of node available*

Options:

-V, --version *Output current version of n*

-h, --help *Display help information*

Aliases:

- rm

which bin

use as

list ls

运行

n

版本号

可以安装任意已发布版本的

Node.js

,

n

会从

<http://nodejs.org>

下载源代码包，

然后自动编译安装，例如：

```
$ n 0.7.5
```

```
#####
```

```
##### 100.0%
```

```
{ 'target_defaults': { 'cflags': [],
```

```
'defines': [],
```

```
'include_dirs': [],
```

```
'libraries': ['-lz']},
```

```
'variables': { 'host_arch': 'x64',  
  
'node_install_npm': 'true',  
  
'node_install_waf': 'true',  
  
'node_prefix': '/usr/local/n/versions/0.7.5',  
  
'node_shared_cares': 'false',  
  
'node_shared_v8': 'false',  
  
'node_use_dtrace': 'false',  
  
'node_use_openssl': 'true',  
  
'node_use_system_openssl': 'false',  
  
'target_arch': 'x64',  
  
'v8_use_snapshot': 'true'}}
```

creating ./config.gypi

creating ./config.mk

make -C out BUILDTYPE=Release

2.6

参考资料

21

1

2

3

5

7

10

8

9

4

6

`CC(target) /usr/local/n/node-v0.7.5/out/Release/obj.target/http_parser/deps/`

`http_parser/http_parser.o`

`LIBTOOL-STATIC /usr/local/n/node-v0.7.5/out/Release/libhttp_parser.a`

...

通过

`n`

获取的

`Node.js`

实例都会安装在

`/usr/local/n/versions/`

目录中。

之后再运行

`n`

即可列出已经安装的所有版本的

`Node.js`

，其中“

*

”后的版本号为默认的

Node.js

版本，即可以直接使用

node

命令行调用的版本：

\$ n

0.6.11

** 0.7.5*

和安装新版本一样，运行

n

版本号

也可以在已安装的

Node.js

实例中切换环境，再运行

node

即为

n

指定的当前版本，例如：

\$ n 0.6.11

** 0.6.11*

0.7.5

\$ node -v

v0.6.11

如果你不想切换默认环境，可以使用

```
n use
```

版本号

```
script.js
```

直接指定

```
Node.js
```

的运

行实例，例如：

```
$
```

```
n use 0.6.11 script.js
```

```
n
```

无法管理通过其他方式安装的

```
Node.js
```

版本实例（如官方提供的安装

包、发行版软件源、手动编译），你必须通过

```
n
```

安装

```
Node.js
```

才能管理多版

本的

```
Node.js
```

。

关于

n

的更多细节，请访问它的项目主页

<https://github.com/visionmedia/n>

获取信息。

2.6

参考资料

“Building and Installing Node.js”:

<https://github.com/joyent/node/wiki/Installation>

。

“Node package manager”: <http://npmjs.org/doc/README.html>

。

“Node version management”: <https://github.com/visionmedia/n>

。

22

第

2

章 安装和配置

Node.js

“

深入浅出

Node.js

(二)

: *Node.js & NPM*

的安装与配置

”: <http://www.infoq.com/cn/>

[articles/nodejs-npm-install-config](http://www.infoq.com/cn/articles/nodejs-npm-install-config)

。

“*Node.js Now Runs Natively on Windows*”:

[http://www.infoq.com/n](http://www.infoq.com/news/2011/11/Nodejs-Windows)

[ews/2011/11/Nodejs-](http://www.infoq.com/news/2011/11/Nodejs-Windows)

[Windows](http://www.infoq.com/news/2011/11/Nodejs-Windows)

。

《

Node Web

开发》，

David Herron

著，人民邮电出版社出版。

“

如何在

Mac OS X Lion

上设定

node.js

的开发环境

”： [http://dreamerslab.com/blog/tw/](http://dreamerslab.com/blog/tw/how-to-setup-a-node-js-development-environment-on-mac-osx-lion/)

[how-to-setup-a-node-js-developme](http://dreamerslab.com/blog/tw/how-to-setup-a-node-js-development-environment-on-mac-osx-lion/)

[nt-environment-on-mac-osx-lion/](http://dreamerslab.com/blog/tw/how-to-setup-a-node-js-development-environment-on-mac-osx-lion/)

。

3.1

开始用

Node.js

编程

23

1

2

3

5

7

10

8

9

4

6

Node.js 快速入门

第

3

章

24

第

3

章

Node.js

快速入门

Node.js

是一个方兴未艾的技术。一直以来，关于

Node.js

的宣传往往针对它“与众不同”

的特性，这使得它显得额外扑朔迷离。事实上，

Node.js

的绝大部分特性跟大多数语言一样都

是旧瓶装新酒，只是一些激进的特性使它显得很神秘。在这一章中，我们将
会讲述

Node.js

的

种种特性，让你对

Node.js

本身以及如何使用

Node.js

编程有一个全局性的了解，主要内容有：

编写第一个

Node.js

程序；

异步式

I/O

和事件循环；

模块和包；

调试。

让我们开始这个激动人心的旅程吧。

3.1

开始用

Node.js

编程

Node.js

具有深厚的开源血统，它诞生于托管了许多优秀开源项目的网站——

github

。和

大多数开源软件一样，它由一个黑客发起，然后吸引了一小拨爱好者参与贡献代码。一开始

它默默无闻，靠口口相传扩散，直到某一天被一个黑客媒体曝光，进入业界视野，随后便有

一些有远见的公司提供商业支持，使其逐步发展壮大。

用

Node.js

编程是一件令人愉快的事情，因为你将开始用黑客的思维和风格编写代码。

你会发现像这样的语言是很容易入门的，可以快速了解到它的细节，然后掌握它。

3.1.1

Hello World

好了，让我们开始实现第一个

Node.js

程序吧。打开你常用的文本编辑器，在其中输入：

```
console.log('Hello World');
```

将文件保存为

helloworld.js

，打开终端，进入

helloworld.js

所在的目录，执行以下命令：

```
node helloworld.js
```

如果一切正常，你将会在终端中看到输出

Hello World

。很简单吧？下面让我们来解

释一下这个程序的细节。

console

是

Node.js

提供的控制台对象，其中包含了向标准输出写

入的操作，如

console.log

、

console.error

等。

`console.log`

是我们最常用的输出

指令，它和

`C`

语言中的

`printf`

的功能类似，也可以接受任意多个参数，支持

`%d`

、

`%s`

变

量引用，例如：

```
//consolelog.js
```

```
console.log('%s: %d', 'Hello', 25);
```

输出的是

`Hello: 25`

。这只是一个简单的例子，如果你想了解

`console`

对象的详细功能，

请参见

4.1.3

节。

3.1

开始用

Node.js

编程

25

1

2

3

5

7

10

8

9

4

6

3.1.2

Node.js

命令行工具

在前面的

Hello World

示例中，我们用到了命令行中的

node

命令，输入

node --help

可以看到详细的帮助信息：

Usage: node [options] [-e script | script.js] [arguments]

node debug script.js [arguments]

Options:

-v, --version print node's version

-e, --eval script evaluate script

-p, --print print result of --eval

--v8-options print v8 command line options

--vars print various compiled-in variables

--max-stack-size=val set max v8 stack size (bytes)

Environment variables:

NODE_PATH ';' -separated list of directories

prefixed to the module search path.

NODE_MODULE_CONTEXTS Set to 1 to load modules in their own

global contexts.

NODE_DISABLE_COLORS Set to 1 to disable colors in the REPL

Documentation can be found at <http://nodejs.org/>

其中显示了

node

的用法，运行

Node.js

程序的基本方法就是执行

node

script.js

,

其中

script.js

①

是脚本的文件名。

除了直接运行脚本文件外，

node --help

显示的使用方法中说明了另一种输出

Hello

Wo r l d

的方式：

\$ node -e "console.log('Hello World');"

Hello World

我们可以把要执行的语句作为

node -e

的参数直接执行。

使用

node

的

REPL

模式

REPL

(

Read-eval-print loop

)，即输入—求值—输出循环。如果你用过

Python

，就会知

道在终端下运行无参数的

python

命令或者使用

Python IDLE

打开的

shell

，可以进入一个即

时求值的运行环境。

Node.js

也有这样的功能，运行无参数的

node

将会启动一个

JavaScript

的交互式

shell

:

①

事实上脚本文件的扩展名不一定是

.js

，例如我们将脚本保存为

script.txt

，使用

node script.txt

命令同样可

以运行。扩展名使用

.js

只是一个约定而已，遵循了

JavaScript

脚本一贯的命名习惯。

26

第

3

章

Node.js

快速入门

\$ node

> console.log('Hello World');

Hello World

undefined

```
> consol.log('Hello World');
```

ReferenceError: consol is not defined

at repl:1:1

at REPLServer.eval (repl.js:80:21)

at repl.js:190:20

at REPLServer.eval (repl.js:87:5)

at Interface.<anonymous> (repl.js:182:12)

at Interface.emit (events.js:67:17)

at Interface._onLine (readline.js:162:10)

at Interface._line (readline.js:426:8)

at Interface._ttyWrite (readline.js:603:14)

at ReadStream.<anonymous> (readline.js:82:12)

进入

REPL

模式以后，会出现一个“

>

”提示符提示你输入命令，输入后按回车，

Node.js

将会解析并执行命令。如果你执行了一个函数，那么

REPL

还会在下面显示这个函数的返回

值，上面例子中的

undefined

就是

console.log

的返回值。如果你输入了一个错误的

指令，

REPL

则会立即显示错误并输出调用栈。在任何时候，连续按两次

Ctrl + C

即可推出

Node.js

的

REPL

模式。

node

提出的

REPL

在应用开发时会给人带来很大的便利，例如我们可以测试一个包能

否正常使用，单独调用应用的某一个模块，执行简单的计算等。

3.1.3

建立

HTTP

服务器

前面的

Hello World

程序对于你来说可能太简单了，因为这个例子几乎可以在任何语言的教科书上找到对应的内容，既无聊又乏味，让我们来点儿不一样的东西，真正感受一下

Node.js

的魅力所在吧。

Node.js

是为网络而诞生的平台，但又与

ASP

、

PHP

有很大的不同，究竟不同在哪里呢？

如果你有

PHP

开发经验，会知道在成功运行

PHP

之前先要配置一个功能强大而复杂的

HTTP

服务器，譬如

Apache

、

IIS

或

Nginx

，还需要将

PHP

配置为

HTTP

服务器的模块，或者使用

FastCGI

协议调用

PHP

解释器。这种架构是“浏览器

HTTP

服务器

PHP

解释器”的组织

方式，而

Node.js

采用了一种不同的组织方式，如图

3-1

所示。

我们看到，

Node.js

将“

HTTP

服务器”这一层抽离，直接面向浏览器用户。这种架构

从某种意义上来说是颠覆性的，因而会让人心存疑虑：

Node.js

作为

HTTP

服务器的效率

足够吗？会不会提高耦合程度？我们不打算在这里讨论这种架构的利弊，后

面章节会继续

说明。

3.1

开始用

Node.js

编程

27

1

2

3

5

7

10

8

9

4

6

图

3-1

Node.js

与

PHP

的架构

好了，回归正题，让我们创建一个

HTTP

服务器吧。建立一个名为

app.js

的文件，内容

为：

```
//app.js
```

```
var
```

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

```
http.createServer(
```

```
function
(req, res) {
  res.writeHead(
    200
    , {'Content-Type': 'text/html'});
  res.write('<h1>Node.js</h1>');
  res.end('<p>Hello World</p>');
}.listen(
  3000
);
console.log("HTTP server is listening at port 3000.");
```

接下来运行

```
node app.js
```

命令，打开浏览器访问

```
http://127.0.0.1:3000
```

，即可看到图

3-2

所示的内容。

图

3-2

用

Node.js

实现的

HTTP

服务器

28

第

3

章

Node.js

快速入门

用

Node.js

实现的最简单的

HTTP

服务器就这样诞生了。这个程序调用了

Node.js

提供的

http

模块，对所有

HTTP

请求答复同样的内容并监听

3000

端口。在终端中运行这个脚本

时，我们会发现它并不像

Hello World

一样结束后立即退出，而是一直等待，直到按下

Ctrl +

C

才会结束。这是因为

listen

函数中创建了事件监听器，使得

Node.js

进程不会退出事件

循环。我们会在后面的章节中详细介绍这其中的奥秘。

小技巧

——

使用

supervisor

如果你有

PHP

开发经验，会习惯在修改

PHP

脚本后直接刷新浏览器以观察结果，而你

在开发

Node.js

实现的

HTTP

应用时会发现，无论你修改了代码的哪一部份，都必须终止

Node.js

再重新运行才会奏效。这是因为

Node.js

只有在第一次引用到某部份时才会去解析脚

本文件，以后都会直接访问内存，避免重复载入，而

PHP

则总是重新读取并解析脚本（如

果没有专门的优化配置）。

Node.js

的这种设计虽然有利于提高性能，却不利于开发调试，因

为我们在开发过程中总是希望修改后立即看到效果，而不是每次都要终止进

程并重启。

supervisor

可以帮助你实现这个功能，它会监视你对代码的改动，并自动重启

Node.js

。

使用方法很简单，首先使用

npm

安装

supervisor

:

\$ npm install -g supervisor

如果你使用的是

Linux

或

Mac

，直接键入上面的命令很可能会有权限错误。原因是

npm

需要把

supervisor

安装到系统目录，需要管理员授权，可以使用

sudo npm install -g

supervisor

命令来安装。

接下来，使用

supervisor

命令启动

app.js

:

\$ supervisor app.js

DEBUG: Running node-supervisor with

DEBUG: program 'app.js'

DEBUG: --watch '.'

DEBUG: --extensions 'node|js'

DEBUG: --exec 'node'

DEBUG: Starting child process with 'node app.js'

DEBUG: Watching directory '/home/byvoid/'

for

changes.

HTTP server is listening at port 3000.

当代码被改动时，运行的脚本会被终止，然后重新启动。在终端中显示的结果如下：

DEBUG: crashing child

DEBUG: Starting child process with 'node app.js'

HTTP server is listening at port 3000.

supervisor

这个小工具可以解决开发中的调试问题。

3.2

异步式

I/O

与事件式编程

29

1

2

3

5

7

10

8

9

4

6

3.2

异步式

I/O

与事件式编程

Node.js

最大的特点就是异步式

I/O

（或者非阻塞

I/O

）与事件紧密结合的编程模式。这

种模式与传统的同步式

I/O

线性的编程思路有很大的不同，因为控制流很大程度上要靠事件

和回调函数来组织，一个逻辑要拆分为若干个单元。

3.2.1

阻塞与线程

什么是阻塞（

block

）呢？线程在执行中如果遇到磁盘读写或网络通信（统称为

I/O

操作），

通常要耗费较长的时间，这时操作系统会剥夺这个线程的

CPU

控制权，使其暂停执行，同

时将资源让给其他的工作线程，这种线程调度方式称为

阻塞

。当

I/O

操作完毕时，操作系统

将这个线程的阻塞状态解除，恢复其对

CPU

的控制权，令其继续执行。这种

I/O

模式就是通

常的同步式

I/O

（

Synchronous I/O

）或阻塞式

I/O

（

Blocking I/O

）。

相应地，异步式

I/O

（

Asynchronous I/O

）或非阻塞式

I/O

（

Non-blocking I/O

）则针对

所有

I/O

操作不采用阻塞的策略。当线程遇到

I/O

操作时，不会以阻塞的方式等待

I/O

操作

的完成或数据的返回，而只是将

I/O

请求发送给操作系统，继续执行下一条语句。当操作系统完成

I/O

操作时，以事件的形式通知执行

I/O

操作的线程，线程会在特定时候处理这个事件。为了处理异步

I/O

，线程必须有

事件循环

，不断地检查有没有未处理的事件，依次予以处理。

阻塞模式下，一个线程只能处理一项任务，要想提高吞吐量必须通过多线程。

而非阻塞

模式下，一个线程永远在执行计算操作，这个线程所使用的

CPU

核心利用率永远是

100%

,

I/O

以事件的方式通知。在阻塞模式下，多线程往往能提高系统吞吐量，因为一个线程阻塞

时还有其他线程在工作，多线程可以让

CPU

资源不被阻塞中的线程浪费。而在非阻塞模式

下，线程不会被

I/O

阻塞，永远在利用

CPU

。多线程带来的好处仅仅是在多核

CPU

的情况

下利用更多的核，而

Node.js

的单线程也能带来同样的好处。这就是为什么

Node.js

使用了单

线程、非阻塞的事件编程模式。

图

3-3

和图

3-4

分别是多线程同步式

I/O

与单线程异步式

I/O

的示例。假设我们有一项工

作，可以分为两个计算部分和一个

I/O

部分，

I/O

部分占的时间比计算多得多（通常都是这

样）。如果我

们使用阻塞

I/O

，那么要想获得高并发就必须开启多个线程。而使用异步式

I/O

时，单线程即可胜任。