PART 1

从安装与概念开始

CHAPTER

1 安装

安装Node.js比较容易。其设计理念之一就是只维护少量的依赖，这使得编译、安装Node.js变得非常简单。

本章介绍如何在Windows、OS X、以及Linux系统下安装Node.js。在Linux系统下，要以编译源代码的方式进行安装[1]，得先确保正确安装了其依赖的软件包。

注意：在本书中，若看到代码片段前有$符号，就表示需要将其代码输入到操作系统的shell中。

在Windows下安装

Windows用户要安装Node.js，只需前往其官网http://nodejs.org下载MSI安装包即可。每个Node.js的发行版都有对应的MSI安装包供用户下载和安装。

安装包文件名遵循`node-v?.?.?.msi`[2]的格式，运行安装包之后，简单地根据图1-1所示的安装指引进行安装即可。

---

1 译者注：在Linux下，官方还提供了二进制包进行安装。

2 译者注：截止到本书翻译期间，目前的格式为node-v?.?.?-bit.msi，这里的bit表示几位的操作系统，如32位就是x86、64位就是x64。

图1-1：Node.js安装指引

要验证是否安装成功，可以打开shell或者通过执行cmd.exe打开命令行工具并输入$ node -version。

如果安装成功的话，就会显示安装的Node.js的版本号。

在OS X下安装

在Mac下和在Windows下安装类似，可通过对应的安装包进行。从Node.js官网下载PKG文件，其文件名格式遵循node-v.?.?.?.pkg。若要通过手动编译来进行安装，请确保机器上已安装了XCode，然后根据Linux下的编译步骤进行编译安装。

运行下载好的安装包，并根据图1-2所示的安装步骤进行安装。

图1-2：Node.js安装包

要验证是否安装成功，打开shell或者运行Terminal.app打开终端工具（也可以在Spotlight中输入“Terminal”来搜索该软件），接着，输入$ node -version。

如果安装成功，就会显示安装的Node.js的版本号。

在Linux下安装

和直接用二进制包安装类似，编译安装Node.js也很简单。要在绝大多数*nix系的系统中编译Node.js，只需要确保系统中有C/C++编译器以及 OpenSSL库就可以了。

要是没有，安装起来也比较容易，大部分的Linux发行版都自带包管理器，通过它可以很方便地进行安装。 9

比方说，在Amazon Linux中，可以通过如下命令来安装依赖包：

```
> sudo yum install gcc gcc-c++ openssl-devel curl
```

在Ubuntu中，安装方式稍有不同，如下所示：

```
> sudo apt-get install g++ libssl-dev apache2-utils curl
```

编译

在操作系统终端下，运行如下命令：

注意：将下面例子中的?替换成最新的Node.js的版本号[3]。

```
$ curl -O http://nodejs.org/dist/node-v?.?.?.tar.gz
$ tar -xzvf node-v?.?.?.tar.gz
$ cd node-v?.?.?
$ ./configure
$ make
$ make test
$ make install
```

如果make test命令报错。我建议你停止安装，并将./configure、make以及make test命令产生的日志信息发送给 Node.js的邮件列表。

确保安装成功

打开终端或者类似XTerm这样的应用，并输入$ node -version。

如果安装成功的话，就会显示安装的Node.js的版本号。

Node REPL

要运行Node的REPL，在终端输入node即可。

可以试试运行一些JavaScript表达式。例如：

```
> Object.keys(global)
```

---

3 译者注：截止到本书翻译期间，Node.js发行版的下载目录已经更改为http://nodejs.org/dist/v?.?.?/node-v?.?.?.tar.gz。

注意：如果看到本书中的示例代码段前有>，就说明要在REPL中输入。

10 REPL是我最喜欢的工具之一，它能让我很方便地验证一些Node API和JavaScript API是否正确。若有时忘记了某个API的用法，就可以用REPL来验证下，非常有用，尤其是在开发大型模块的时候。我一般都新开一个单独的终端tab，快速在REPL中尝试一些JavaScript的原生用法，真的非常方便。

执行文件

和绝大多数脚本语言一样，Node.js可以通过node命令来执行Node脚本。

用你喜欢的编辑器，创建一个名为my-web-server.js的文件，输入如下内容：

```
var http = require('http');
var serv = http.createServer(function (req, res) {
  res.writeHead(200, { 'Content-Type': 'text/html' });
  res.end('<marquee>Smashing Node!</marquee>');
});
serv.listen(3000);
```

使用如下命令来执行此文件：

```
$ node my-web-server.js
```

接着，如图1-3所示，在浏览器中输入http://localhost:3000。

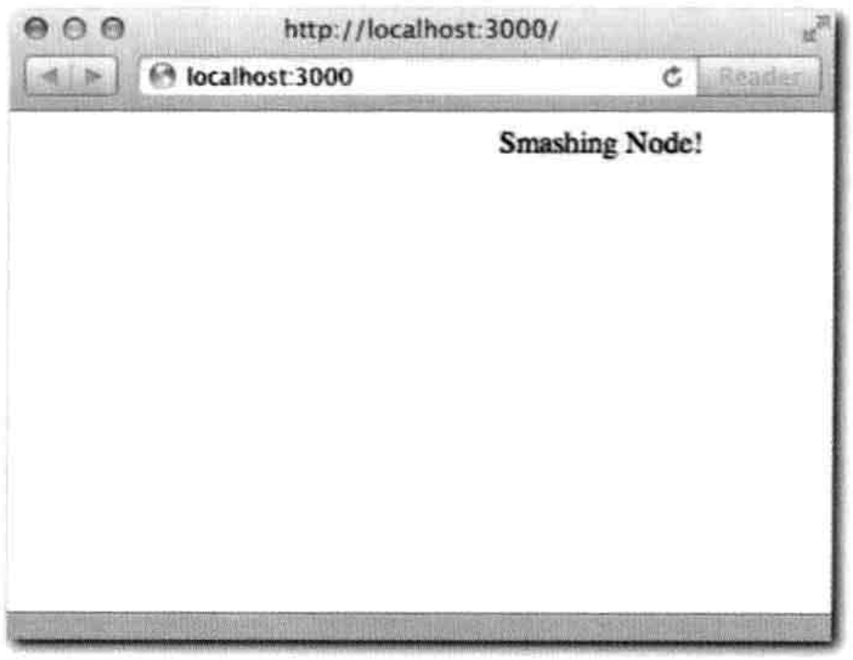

图1-3：使用Node托管一个简单的HTML文件

上述代码展示了如何使用Node书写一个完整的HTTP服务器，来托管一个简单的HTML文档。这是一个Node.js的经典例子，因为它证明了Node.js的强大，仅通过几行JavaScript代码就能创建出一个像Apache或者IIS的Web服务器。

NPM

Node包管理器（NPM）可以让你在项目中轻松地对模块进行管理，它会下载指定的包、

解决包的依赖、运行测试脚本以及安装命令行脚本。

尽管这些工作并非你项目的核心功能，但使用第三方发布的模块可以提高项目的开发效率。

NPM本身是用Node.js开发的，有二进制包的发布形式（Windows下有MSI安装器，Mac下有PKG文件）。若要从源码进行编译安装，可以使用如下命令[4]：

11

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

通过如下命令可以检查NPM是否安装成功：

```
$ npm --version
```

安装成功的话，会显示出所安装NPM的版本号。

安装模块

为了展示如何通过NPM来安装模块，我们创建一个my-project目录，安装colors模块，然后创建一个index.js文件：

```
$ mkdir my-project/
$ cd my-project/
$ npm install colors
```

要验证模块是否安装成功，可以在该目录下查看是否有node_modules/colors目录。

然后，用你最喜欢的编辑器编辑index.js文件：

```
$ vim index.js
```

在该文件中添加如下内容：

```
require('colors');
console.log('smashing node'.rainbow);
```

运行此文件的结果应该如图1-4所示。

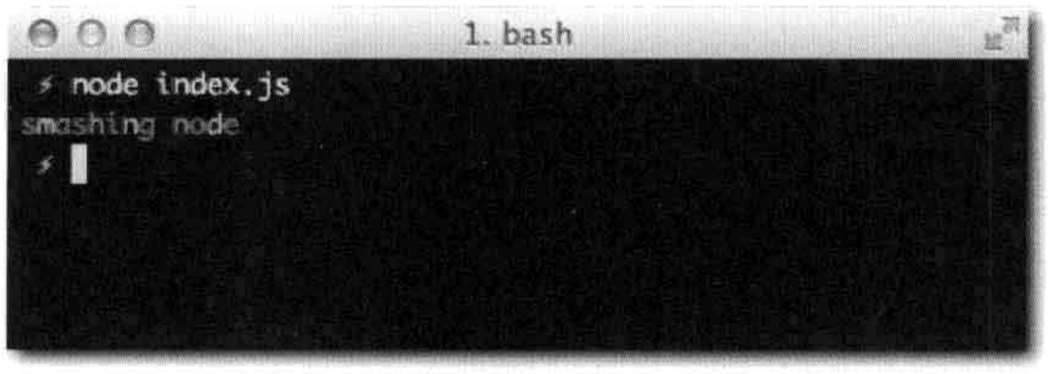

图1-4：模块安装成功验证结果

4 译者注：截止到本书翻译期间，NPM会随着Node.js的安装自动就安装好了，无须手动再去安装NPM，并且 http://nodejs.org/install.sh脚本已经被官方移除。

12 自定义模块

要自定义模块，你需要创建一个package.json文件。通过这种方式来定义模块有三种好处：

- 可以很方便地将项目中的模块分享给其他人，不需要将整个node_modules目录发给他们。因为有了package.json之后，其他人运行npm install就可以把依赖的模块都下载下来，直接将node_modules目录给别人根本就是个馊主意。特别是当用Git这样的SCM系统进行代码控制的时候。
- 可以很方便地记录所依赖模块的版本号。举个例子来说，当你的项目通过npm install colors安装的是0.5.0的colors。一年后，由于colors模块API的更改，可能导致与你的项目不兼容，如果你使用npm install并且不指定版本号来安装的话，你的项目就没法正常运行了。
- 让分享更简单。如果你的项目不错，你是否想将它分享给别人？这时，因为有package.json文件，通过npm publish就可以将其发布到NPM库中供所有人下载使用了。

在原先创建的目录（my-project）中，删除node_modules目录并创建一个package.json文件：

```
$ rm -r node_modules
$ vim package.json
```

然后，将如下内容添加到该文件中[5]：

```
{
    "name": "my-colors-project"
  , "version": "0.0.1"
  , "dependencies": {
      "colors": "0.5.0"
    }
}
```

注意：此文件内容必须遵循JSON格式。仅遵循JavaScript格式是不够的。举例来说，你必须要确保所有的字符串，包括属性名，都是使用双引号而不是单引号。

package.json文件是从Node.js和NPM两个层面来描述项目的。其中只有name和version是必要的字段。通常情况下，还会定义一些依赖的模块，通过使用一个对象，将依赖模块的模块名及版本号以对象的属性和值将其定义在package.json文件中。

保存上述文件，安装依赖的模块，然后再次运行index.js文件：

13

```
$ npm install
$ node index    # 注意了，这里文件名不需要加上“.js”后缀
```

5 译者注：不建议示例代码中逗号的书写风格，个人建议将逗号写在行末。

在本例中，自定义模块是内部使用的。不过，如果想发布出去，NPM提供了如下这种方式，可以很方便地发布模块：

```
$ npm publish
```

当别人使用require('my-colors-project')时，为了能够让Node知道该载入哪个文件，我们可以在package.json文件中使用main属性来指定：

```
{
    "name": "my-colors-project"
  , "version": "0.0.1"
  , "main": "./index"
  , "dependencies": {
      "colors": "0.5.0"
    }
}
```

当需要让模块暴露API的时候，main属性就会变得尤为重要，因为你需要为模块定义一个入口（有的时候，入口可能是多个文件）。

要查看package.json文件所有的属性文档，可以使用如下命令：

```
$ npm help json
```

小贴士：如果你不想发布你的模块，那么在package.json中加入"private": "true"。这样可以避免误发布。

14

安装二进制工具包

有的项目分发的是Node编写的命令行工具。这个时候，安装时要增加-g标志。

举例来说，本书中要介绍的Web框架express就包含一个用于创建项目的可执行工具。

```
$ npm install -g express
```

安装好后，新建一个目录，并在该目录下运行express命令：

```
$ mkdir my-site
$ cd mysite
$ express
```

小贴士：要想分发此类脚本，发布时，在package.json文件中添加"bin": "./path/to/script"项，并将其值指向可执行的脚本或者二进制文件。

浏览NPM仓库

等掌握第4章关于Node.js模块系统的内容后，你就能编写出可以使用Node生态系统中任意类型模块的程序了。

NPM有一个丰富的仓库，包含了上千个模块。NPM有两个命令可以用来在仓库中搜索和查看模块：search和view。

例如，要搜索和realtime相关的模块，就可以执行如下命令：

```
$ npm search realtime
```

该命令会在已发布模块的name、tags以及description字段中搜索此关键字，并返回匹配的模块。

找到了感兴趣的模块后，通过运行npm view命令，后面紧跟该模块名，就能看到package.json文件以及与NPM仓库相关的属性，举个例子：

```
$ npm view socket.io
```

小贴士：输入npm help可以查看某个NPM命令的帮助文档，如npm help publish就会教你如何发布模块。

小结

通过本章的学习，你应当已经搭建好了Node.js + NPM的环境。

除了能够运行node和npm命令外，你现在也应当学会了如何执行简单脚本以及如何声明模块依赖。

相信你还学会了Node.js中一个重要的关键词require，它用来载入模块和系统API，在快速介绍完语言基本知识后，第4章中会对这部分内容做着重介绍。

最后相信你了解了NPM仓库，它是Node.js模块生态系统的入口。Node.js是开源项目，所以大部分Node.js编写的程序也都是开源的，供其他人重用。