

PART

# Node重要的 API

CHAPTER 5 命令行工具（CLI）以及FS API：首个Node应用

CHAPTER 6 TCP

CHAPTER 7 HTTP

## 5

# 命令行工具（CLI） 以及FS API：首个 Node应用

本章将介绍使用Node.js中一些重量级API：处理进程（stdio）的stdin以及stdout相关的API，还有那些与文件系统（fs）相关的API。 51

第4章中我们介绍过，Node通过使用回调和事件机制来实现并发。这些API会让你首次接触到基于非阻塞事件的I/O编程中的流控制。

除了介绍如何使用这些API之外，你还可以通过本章来构建首个应用：一个简单的命令行文件浏览器，其功能是允许用户读取和创建文件。

## 需求

52

我们从定义需求开始：

- 程序需要在命令行运行。这就意味着程序要么通过node命令来执行，要么直接执行，然后通过终端提供交互给用户进行输入、输出。
- 程序启动后，需要显示当前目录下列表（见图5-1）。
- 选择某个文件时，程序需要显示该文件内容。



- 选择一个目录时，程序需要显示该目录下的信息。
- 运行结束后程序退出。

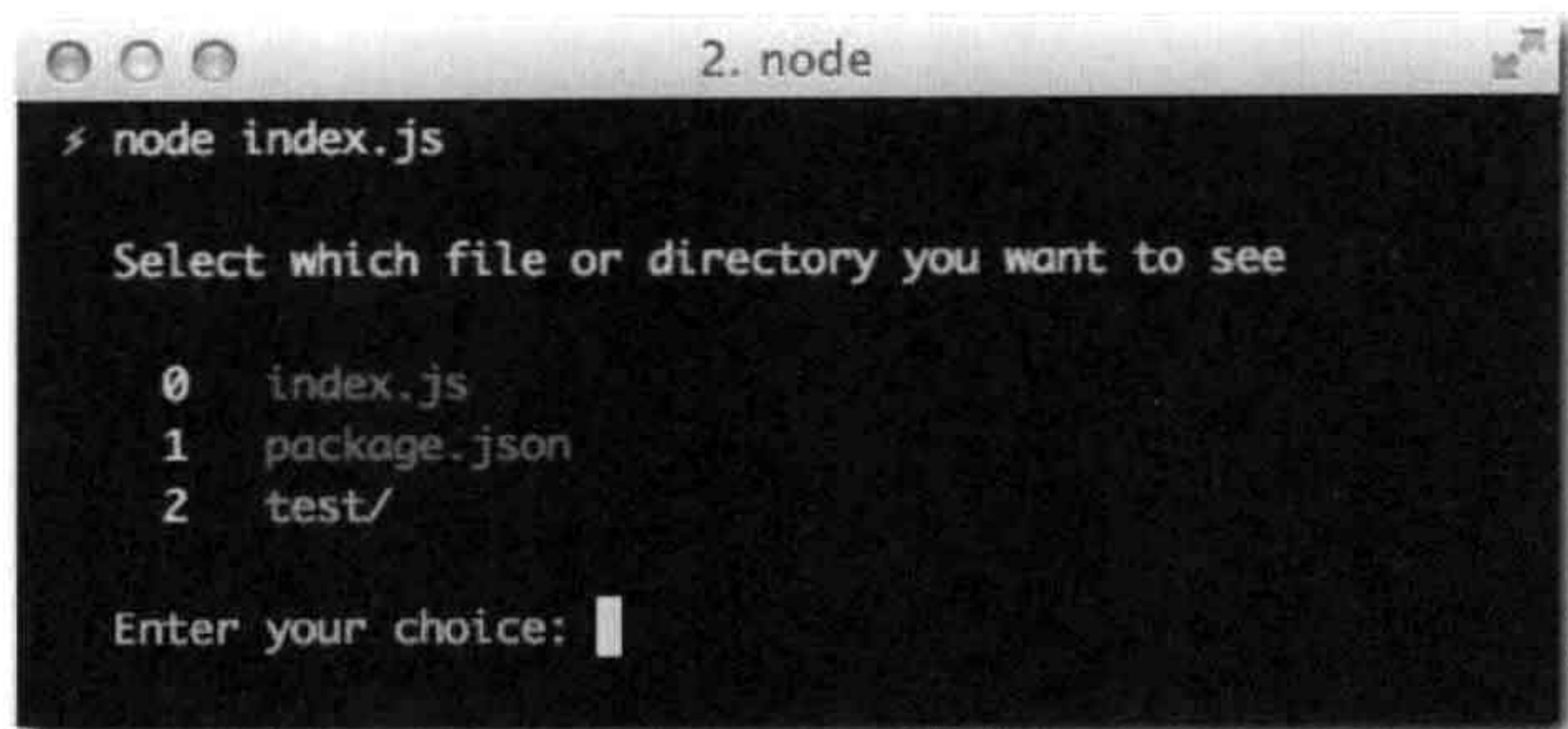


图5-1：程序启动后显示的目录列表

根据上述需求，你可以将此项目细分到如下几个步骤：

1. 创建模块。
2. 决定采用同步的fs还是异步的fs。
3. 理解什么是流（Stream）。
4. 实现输入输出。
5. 重构。
6. 使用fs进行文件交互。
7. 完成。

## 编写首个Node程序

现在我们开始基于上述步骤来编写一个模块。模块由几个文件组成，编写时可以使用任意文本编辑器。

通过本章，我们会完成一个具备完整功能的纯Node.js应用。

### 53 ➤ 创建模块

和本书其他例子一样，我们从创建一个项目目录开始。按照此项目的需求，我们将该目录命名为file-explorer。

正如其他章节所述，在项目中定义package.json文件始终是最佳实践。这样，既可以方便地对NPM中注册的模块依赖进行管理，将来也能对模块进行发布。

尽管此项目仅仅用到Node.js的核心模块API（因此，不会从NPM仓库中获取模块），但



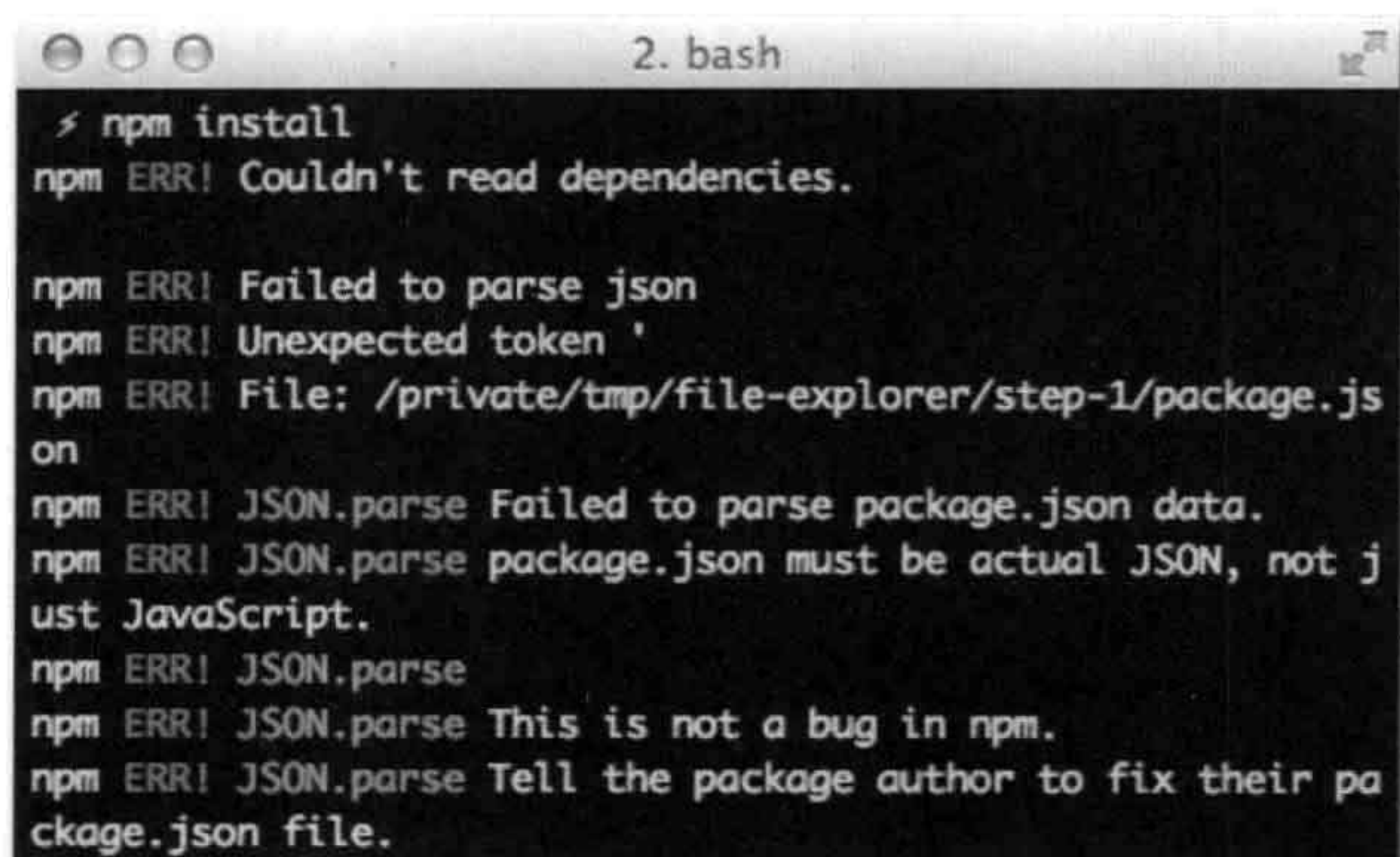
是，我们还是需要创建一个简单的package.json文件：

```
# package.json
{
  "name": "file-explorer"
  , "version": "0.0.1"
  , "description": "A command-file file explorer!"
}
```

注意：NPM遵循一个名为semver<sup>1</sup>的版本控制标准。这就是为何不使用“0.1”或者“1”作为版本号，而是用“0.0.1”的原因。

要验证你的package.json文件是否有效，可以运行\$ npm install。

要是没有问题，就不会有任何输出内容，否则会抛出JSON异常的错误（见图5-2）。

A terminal window titled '2. bash' showing the output of the 'npm install' command. The output consists of several error messages: 'npm ERR! Couldn't read dependencies.', 'npm ERR! Failed to parse json', 'npm ERR! Unexpected token ''', 'npm ERR! File: /private/tmp/file-explorer/step-1/package.json', 'npm ERR! JSON.parse Failed to parse package.json data.', 'npm ERR! JSON.parse package.json must be actual JSON, not just JavaScript.', 'npm ERR! JSON.parse', 'npm ERR! JSON.parse This is not a bug in npm.', and 'npm ERR! JSON.parse Tell the package author to fix their package.json file.'

```
2. bash
$ npm install
npm ERR! Couldn't read dependencies.

npm ERR! Failed to parse json
npm ERR! Unexpected token ''
npm ERR! File: /private/tmp/file-explorer/step-1/package.json
npm ERR! JSON.parse Failed to parse package.json data.
npm ERR! JSON.parse package.json must be actual JSON, not just JavaScript.
npm ERR! JSON.parse
npm ERR! JSON.parse This is not a bug in npm.
npm ERR! JSON.parse Tell the package author to fix their package.json file.
```

图5-2：package.json文件中有JSON错误的情况下运行npm install命令

接着，我们要创建一个简单的包含程序完整功能的JavaScript文件：index.js。

## 同步还是异步

54

我们从声明依赖关系开始。由于stdio API是全局process对象的一部分，所以，我们的程序唯一的依赖就是fs模块：

```
# index.js
/**
 * Module dependencies.
 */

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

我们首先要做的就是获取当前目录的文件列表。

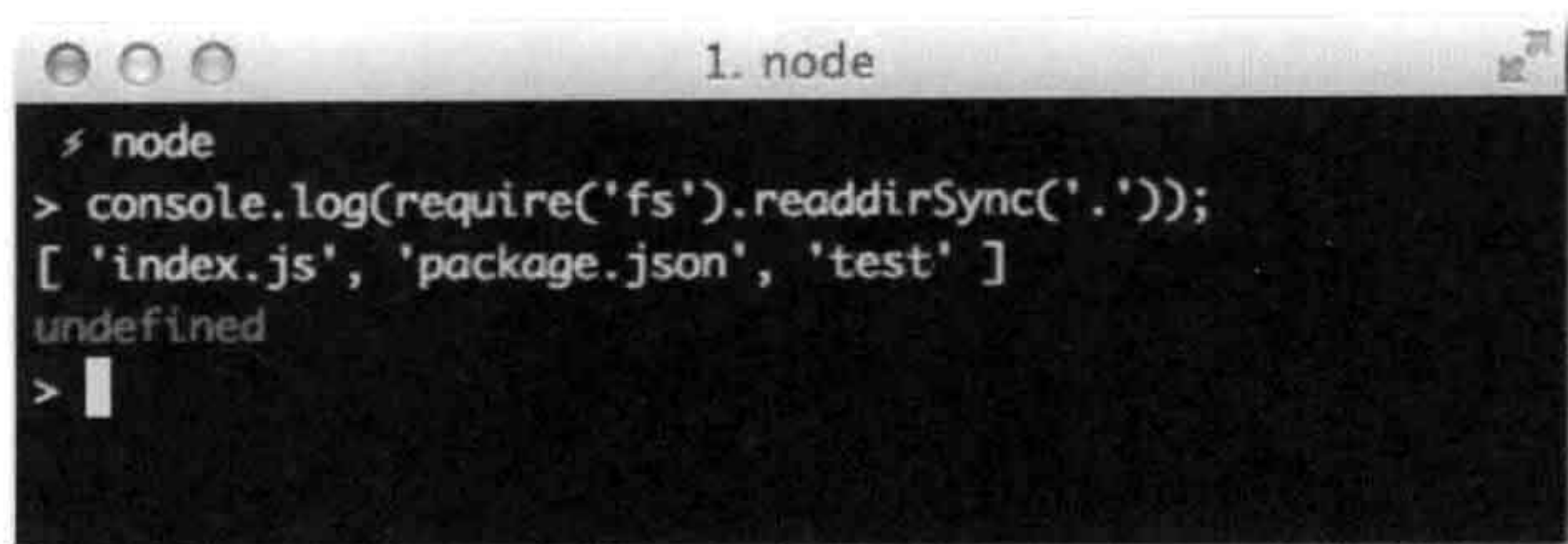
<sup>1</sup> 译者注：<http://semver.org/>。



有件重要的事情要记住：fs模块是唯一一个同时提供同步和异步API的模块。举个例子来说，要想获取当前目录的文件列表，可以使用如下方式：

```
> console.log(require('fs').readdirSync(__dirname));
```

它会立刻返回内容或者当有错误发生时抛出相应异常（见图5-3）。

A terminal window titled "1. node" showing the execution of a Node.js command. The prompt is "node". The user enters "> console.log(require('fs').readdirSync('.'));". The output is "[ 'index.js', 'package.json', 'test' ]". The prompt returns to ">".

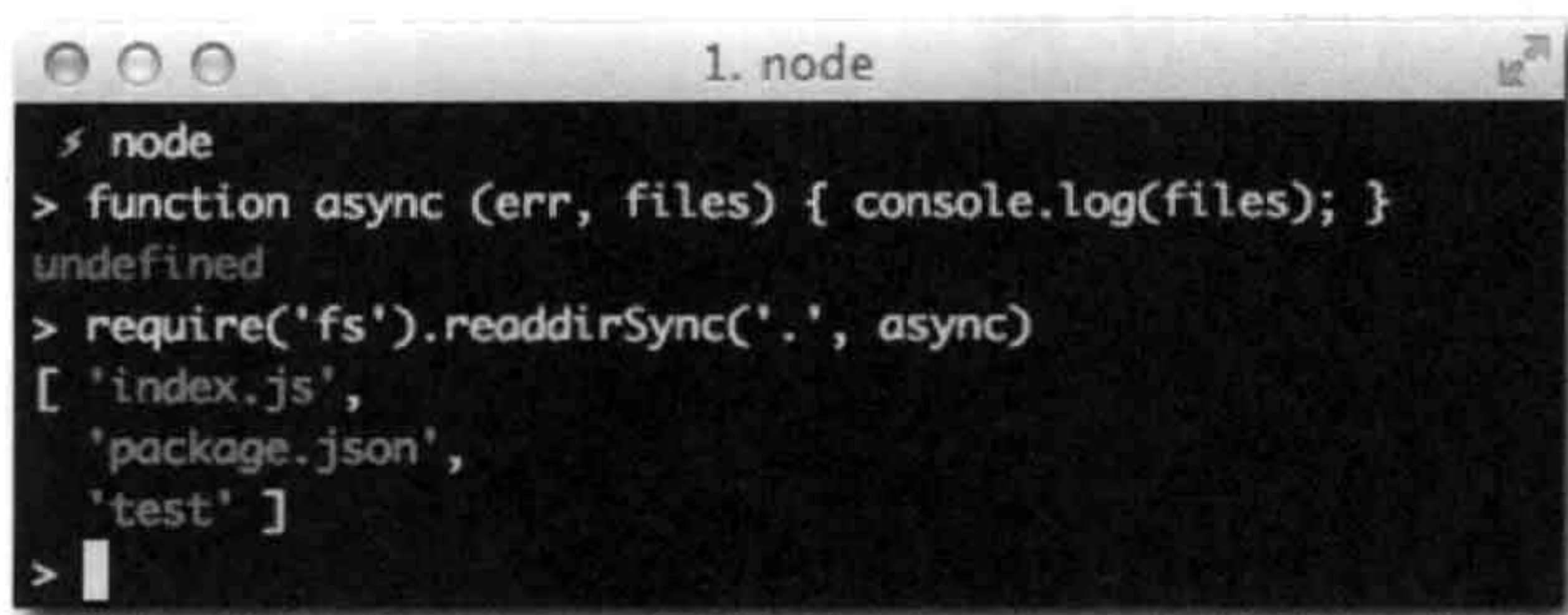
```
node
> console.log(require('fs').readdirSync('.'));
[ 'index.js', 'package.json', 'test' ]
>
```

图5-3：查看readdirSync的返回值

下面是异步的版本：

```
> function async (err, files) { console.log(files); };
> require('fs').readdir('.', async);
```

如图5-4所示，上述两个例子结果是一样的。

A terminal window titled "1. node" showing the execution of a Node.js command. The prompt is "node". The user enters "> function async (err, files) { console.log(files); }". The output is "undefined". The user then enters "> require('fs').readdirSync('.', async)". The output is "[ 'index.js', 'package.json', 'test' ]". The prompt returns to ">".

```
node
> function async (err, files) { console.log(files); }
undefined
> require('fs').readdirSync('.', async)
[ 'index.js',
  'package.json',
  'test' ]
>
```

图5-4：异步版本的readdir

55 第3章中提过，要在单线程中创建能够处理高并发的高效程序，就得采用异步、事件驱动的程序。

尽管本章中这个命令行程序并非此类型程序（因为同一时间只会有一人在读取文件），但是，为了学习Node.js中最重要也是最具挑战的部分，我们还是保持这种异步的代码风格。

为了获取文件列表，我们需要使用fs.readdir。我们提供的回调函数首个参数是一个错误对象（如果没有错误发生，该对象为null），另外一个参数是一个files数组：

```
# index.js
// ...
fs.readdir(__dirname, function (err, files) {
  console.log(files);
});
```



运行上述代码，会得到如图5-5所示的结果。

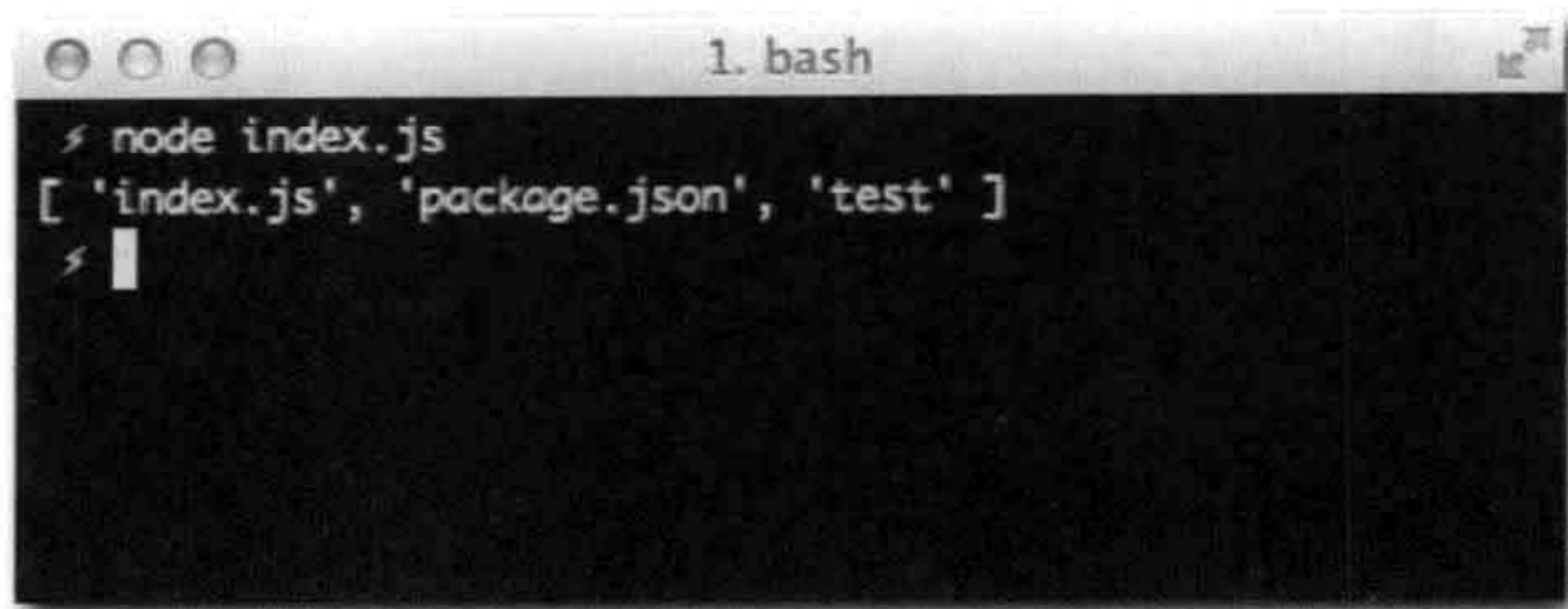


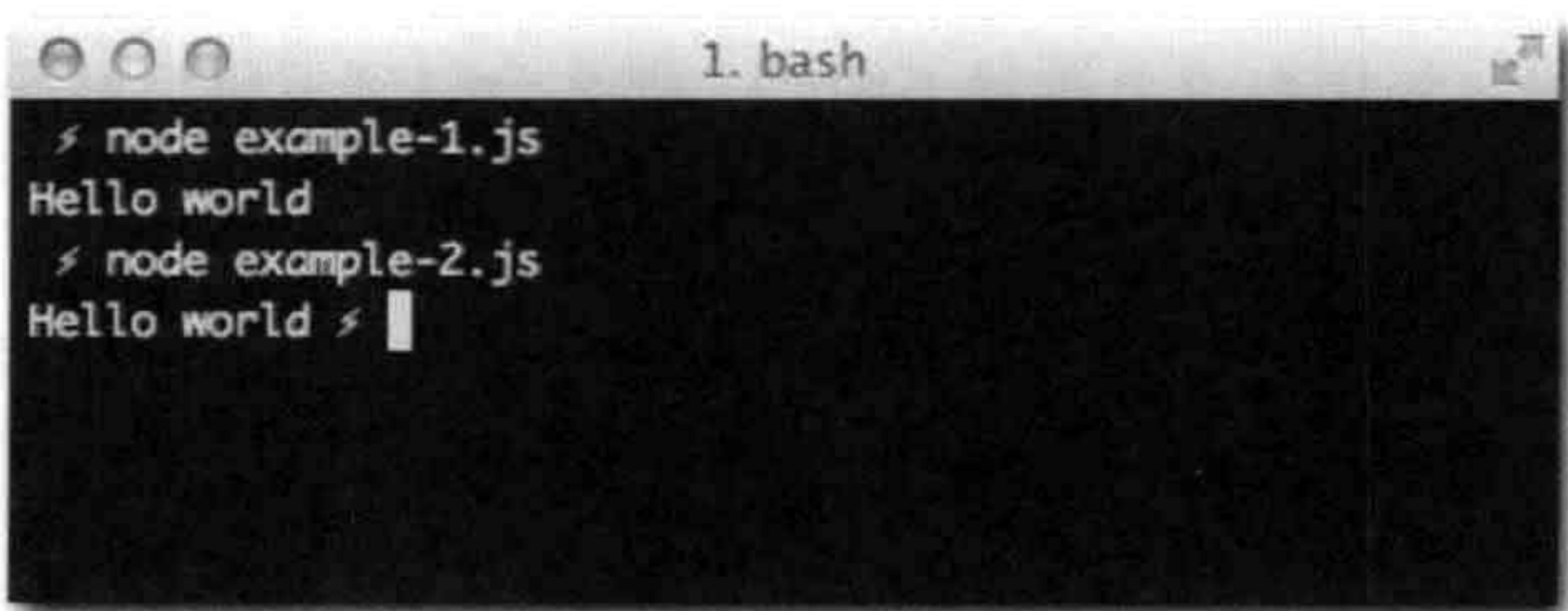
图5-5：运行自己的包含在index.js文件中的Node程序

好了，至此，你已经知道了fs模块同时提供了同步和异步的API来操作文件系统，接下来你需要学习Node.js中一个基础概念——流。

### 理解什么是流 (stream)

我们已经知道，console.log会输出到控制台。事实上，console.log内部做了这样的事：它在指定的字符串后加上\n（换行）字符，并将其写到stdout流中。

观察图5-6中显示的两个程序的不同点。



56

图5-6：第一个Hello World的程序加了一个换行符，第二个则没加

我们再来看下面的源代码：

```
# example-1.js
console.log('Hello world');
```

和：

```
# example-2.js
process.stdout.write('Hello world');
```

process全局对象中包含了三个流对象，分别对应三个UNIX标准流：

- **\*\*stdin\*\***: 标准输入
- **\*\*stdout\*\***: 标准输出
- **\*\*stderr\*\***: 标准错误

图5-7描述了这三个流对象。



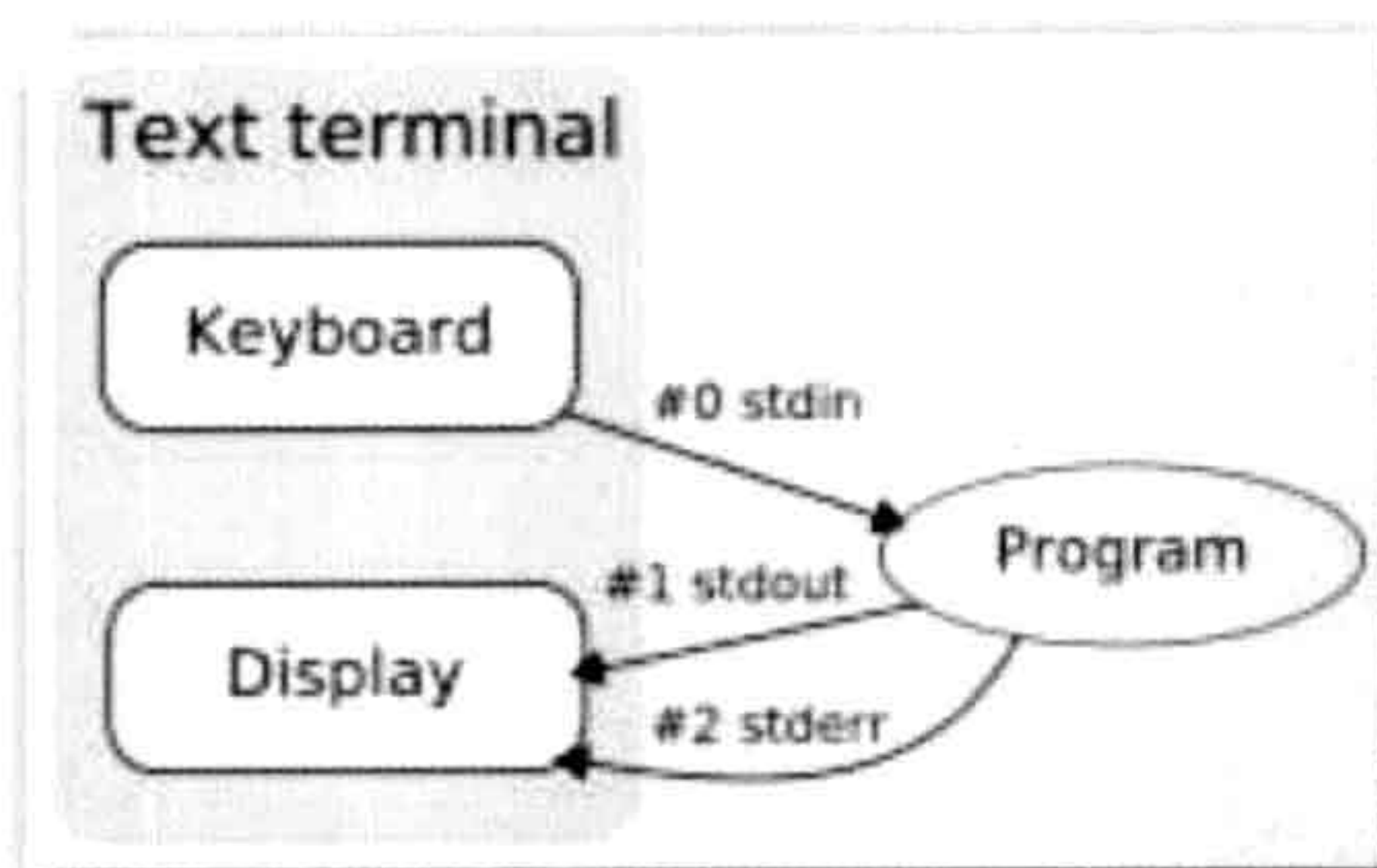


图5-7：传统文本终端下的stdio、stdout以及stderr对象

第一个stdin是一个可读流，而stdout和stderr都是可写流。

stdin流默认的状态是暂停的（paused）。通常，执行一个程序，程序会做一些处理，然后退出。不过，有些时候，就像本章中的这个应用一样，程序需要一直处在运行状态来接收用户输入的数据。

当恢复那个流时，Node会观察对应的文件描述符（在UNIX下为0），随后保持事件循环的运行，同时保持程序不退出，等待事件的触发。除非有IO等待，否则Node.js总是会主动退出。

57

流的另外一个属性是它默认的编码。如果在流上设置了编码，那么会得到编码后的字符串（utf-8、ascii等）而不是原始的Buffer作为事件参数。

Stream对象和EventEmitter很像（事实上，前者继承自后者）。在Node中，你会接触到各种类型流，如TCP套接字、HTTP请求等。简而言之，当涉及持续不断地对数据进行读写时，流就出现了。

## 输入和输出

既然已经知道运行程序后大概是怎样的一个情形了，我们来尝试写第一部分，列出当前目录下的文件，然后等待用户输入：

```
# index.js
// ...
fs.readdir(process.cwd(), function (err, files) {
  console.log('');

  if (!files.length) {
    return console.log('    \033[31m No files to show!\033[39m\n');
  }

  console.log('    Select which file or directory you want to see\n');
```



```

function file(i) {
  var filename = files[i];

  fs.stat(__dirname + '/' + filename, function (err, stat) {
    if (stat.isDirectory()) {
      console.log('    '+i+'    \033[36m' + filename + ' /\033[39m');
    } else {
      console.log('    '+i+'    \033[90m' + filename + '\033[39m');
    }

    i++;
    if (i == files.length) {
      console.log('');
      process.stdout.write('    \033[33mEnter your choice: \033[39m');
      process.stdin.resume();
    } else {
      file(i);
    }
  });
}

file(0);
});

```

下面，我们来逐行分析上述代码。

58

为了输出更加友好，我们首先输出一个空行：

```
console.log('')
```

如果files数组为空，告知用户当前目录没有文件。文本周围的\033[31m和\033[39m是为了让文本呈现为红色。例子中最后一个字符又是换行符\n，也是为了输出更友好。

```

if (!files.length) {
  return console.log('    \033[31m No files to show!\033[39m\n');
}

```

下一行很简单，一眼就看明白了：

```
console.log('    Select which file or directory you want to see\n');
```

紧接着，定义了一个函数，数组中每个元素都会执行该函数。这里也出现了贯穿本书始终的第一种异步流控制模式：串行执行。本章最后会对此做详细介绍。

```

function file (i) {
  // . . .
}

```

然后，先获取文件名，再查看文件名对应路径的情况。fs.stat会给出文件或者目录的



元数据：

```
var filename = files[i];

fs.stat(__dirname + '/' + filename, function (err, stat) {
    // . . .
});
```

回调函数中，同时还给出了错误对象（如果有的话）和一个Stat对象。本例中所使用到的Stat对象上的方法是isDirectory：

```
if (stat.isDirectory()) {
    console.log('      '+i+'    \033[36m' + filename + '/\033[39m');
} else {
    console.log('      '+i+'    \033[90m' + filename + '\033[39m');
}
```

如果路径所代表的是目录，我们就用有别于文件的颜色标识出来。

接下来就到了流控制中的核心部分了。计数器不断递增，与此同时，检查是否还有未处理的文件：

59 ▶

```
i++;
if (i == files.length) {
    console.log('');
    process.stdout.write('    \033[33mEnter your choice: \033[39m');
    process.stdin.resume();
    process.stdin.setEncoding('utf8');
} else {
    file(i);
}
```

如果所有文件都处理完毕，此时提示用户进行选择。注意，这里使用的是process.stdout.write而不是console.log；这样就无须换行，让用户可以直接在提示语后进行输入（见图5-8）：

```
console.log('');
process.stdout.write('    \033[33mEnter your choice: \033[39m');
```

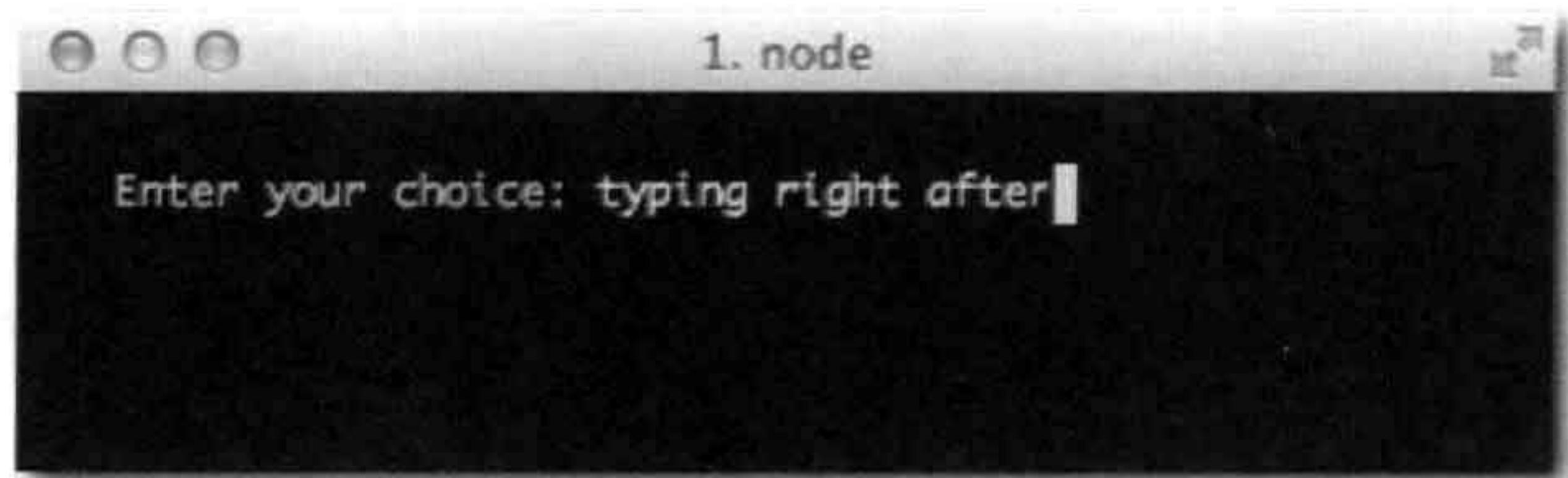


图5-8：程序提示用户向stdin进行输入

下面这行代码，此前介绍过，是用户等待用户输入：

```
process.stdin.resume();
```



紧跟着这行代码的是设置流编码为utf8，这样就能支持特殊字符了：

```
process.stdin.setEncoding('utf8');
```

要是还有未处理的文件，则递归调用该函数来进行处理：

```
file(i);
```

直到列出所有文件、用户输入完毕后，紧接着进行下一步串行处理。这是本章介绍的首个重要的模式。

## 重构

要做重构，我们从为几个常用的变量（如stdin和stdout）创建快捷变量开始：

```
# index.js
// . . .
var fs = require('fs')
    , stdin = process.stdin
    , stdout = process.stdout
```

60

由于我们书写的代码都是异步的，因此，会有这样的问题：随着函数数量的增长（特别是流控制层的增加），过多的函数嵌套会让程序的可读性变差。

为了避免此类问题，我们可以为每一个异步操作预先定义一个函数。

首先，我们抽离出一个读取stdin函数：

```
# index.js
// called for each file walked in the directory
function file(i) {
    var filename = files[i];

    fs.stat(__dirname + '/' + filename, function (err, stat) {
        if (stat.isDirectory()) {
            console.log('    '+i+'    \033[36m' + filename + '/\033[39m');
        } else {
            console.log('    '+i+'    \033[90m' + filename + '\033[39m');
        }

        if (++i == files.length) {
            read();
        } else {
            file(i);
        }
    });
}

// read user input when files are shown
function read () {
    console.log('');
    stdout.write('    \033[33mEnter your choice: \033[39m');
```



```
    stdin.resume();
    stdin.setEncoding('utf8');
}
```

注意，上述代码所使用的是新的stdin和stdout引用。

读取用户输入后，接下来要做的就是根据用户输入做出相应处理。用户需要选择要读取的文件，所以，代码层面，我们设置了stdin的编码后，就开始监听其data事件：

```
61 function read () {
    // . . .
    stdin.on('data', option);
}

// called with the option supplied by the user
function option (data) {
    if (!files[Number(data)]) {
        stdout.write('    \033[31mEnter your choice: \033[39m');
    } else {
        stdin.pause();
    }
}
```

这里，我们检查用户的输入是否匹配files数组中的下标。还记得files数组是fs.readdir回调函数中的一部分吧。另外，要注意的是，上述代码中，我们将utf-8编码的字符串类型data转化为Number类型来方便做检查。

如果检查通过，我们要确保再次将流暂停（回到默认状态），以便于之后做完fs操作后，程序能够顺利退出（见图5-9）。

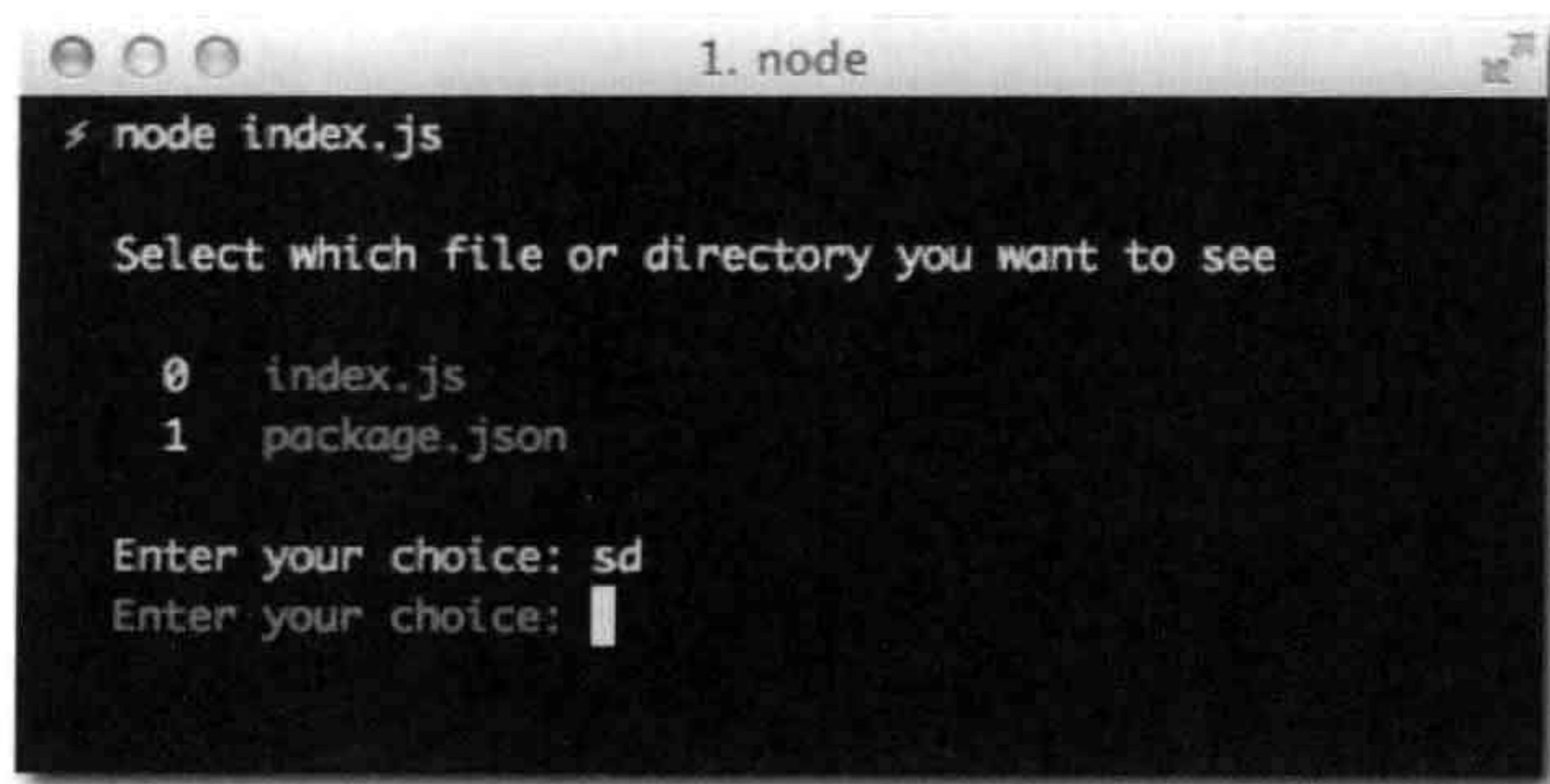


图5-9：用户选择错误的例子

至此，我们的程序已经能够与用户进行交互了，将当前目录中的文件列表展现给用户，下面我们来实现读取和显示文件内容。



## 用fs进行文件操作

既然都能定位到文件了，那是时候去读取它了！

```
function option (data) {
  var filename = files[Number(data)];
  if (!filename) {
    stdout.write('    \033[31mEnter your choice: \033[39m');
  } else {
    stdin.pause();
    fs.readFile(__dirname + '/' + filename, 'utf8', function (err, data) {

      console.log('');
      console.log('\033[90m' + data.replace(/(.*)/g, '    $1') + '\033[39m');
    });
  }
}
```

62

再次提醒，我们可以事先指定编码，这样我们得到的数据就是相应的字符串了：

```
fs.readFile(__dirname + '/' + filename, 'utf8', function (err, data) {
```

接着，可以使用正则表达式添加一些辅助缩进后将文件内容进行输出（见图5-10）：

```
data.replace(/(.*)/g, '    $1')
```

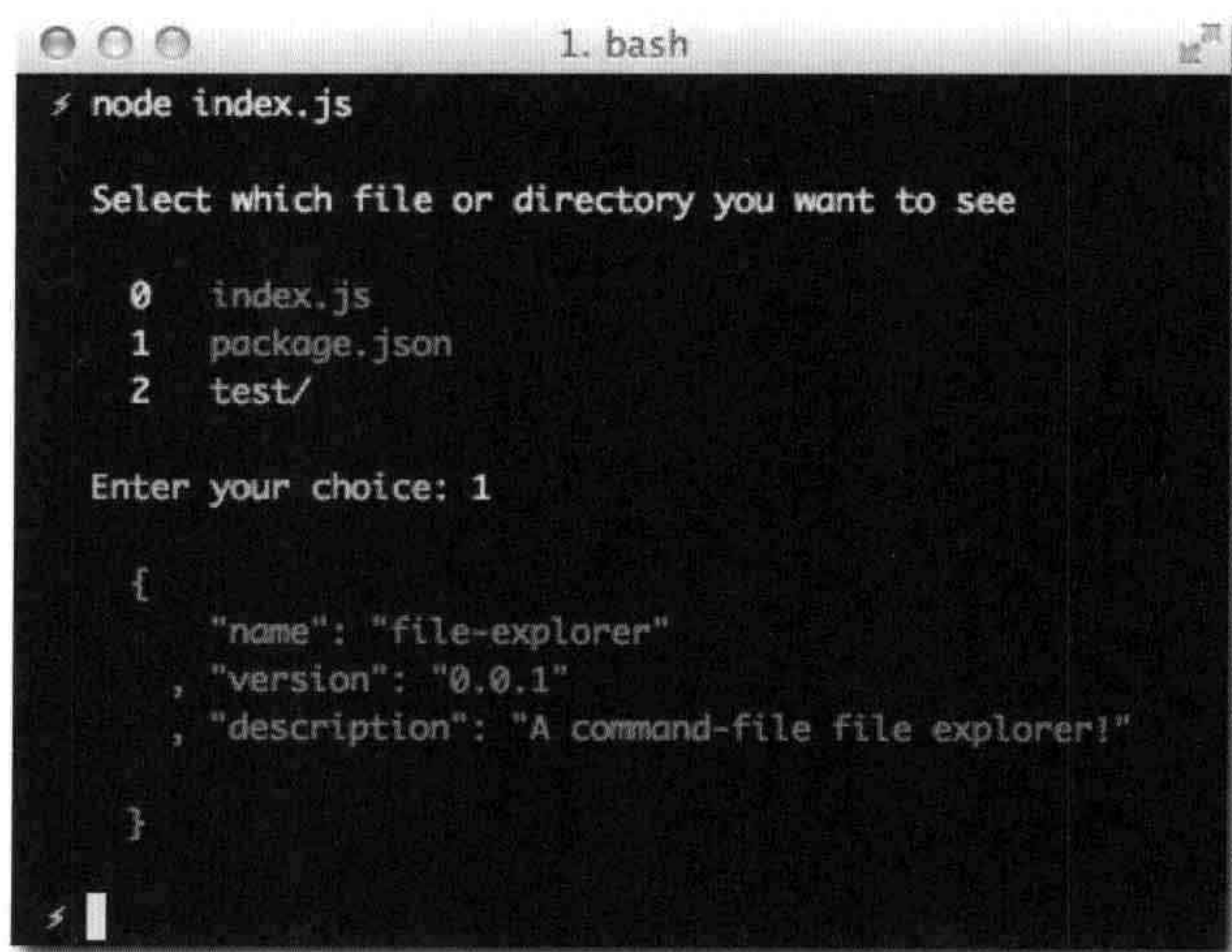


图5-10：读取简单文件的例子

不过，要是选择的是目录呢？这种情况下，我们就应当将其目录下的文件列表显示出来。

为了避免再次执行`fs.stat`，我们在`file`函数中，将`Stat`对象保存下来：

```
// . . .
var stats = [];
```



```
function file(i) {
  var filename = files[i];

  fs.stat(__dirname + '/' + filename, function (err, stat) {
    stats[i] = stat;
    // . . .
  });
}
```

63

好了，现在可以轻松地在`option`函数中进行检查操作了。下述代码对应原先执行`fs.readFile`的代码位置：

```
if (stats[Number(data)].isDirectory()) {
  fs.readdir(__dirname + '/' + filename, function (err, files) {
    console.log('');
    console.log('  (' + files.length + ' files)');
    files.forEach(function (file) {
      console.log('    - ' + file);
    });
    console.log('');
  });
} else {
  fs.readFile(__dirname + '/' + filename, 'utf8', function (err, data) {
    console.log('');
    console.log('\033[90m' + data.replace(/(.*)/g, ' $1') + '\033[39m');
  });
}
```

现在再运行该程序，就能够进行目录选择，并能看到该目录下的文件列表信息了（见图5-11）。

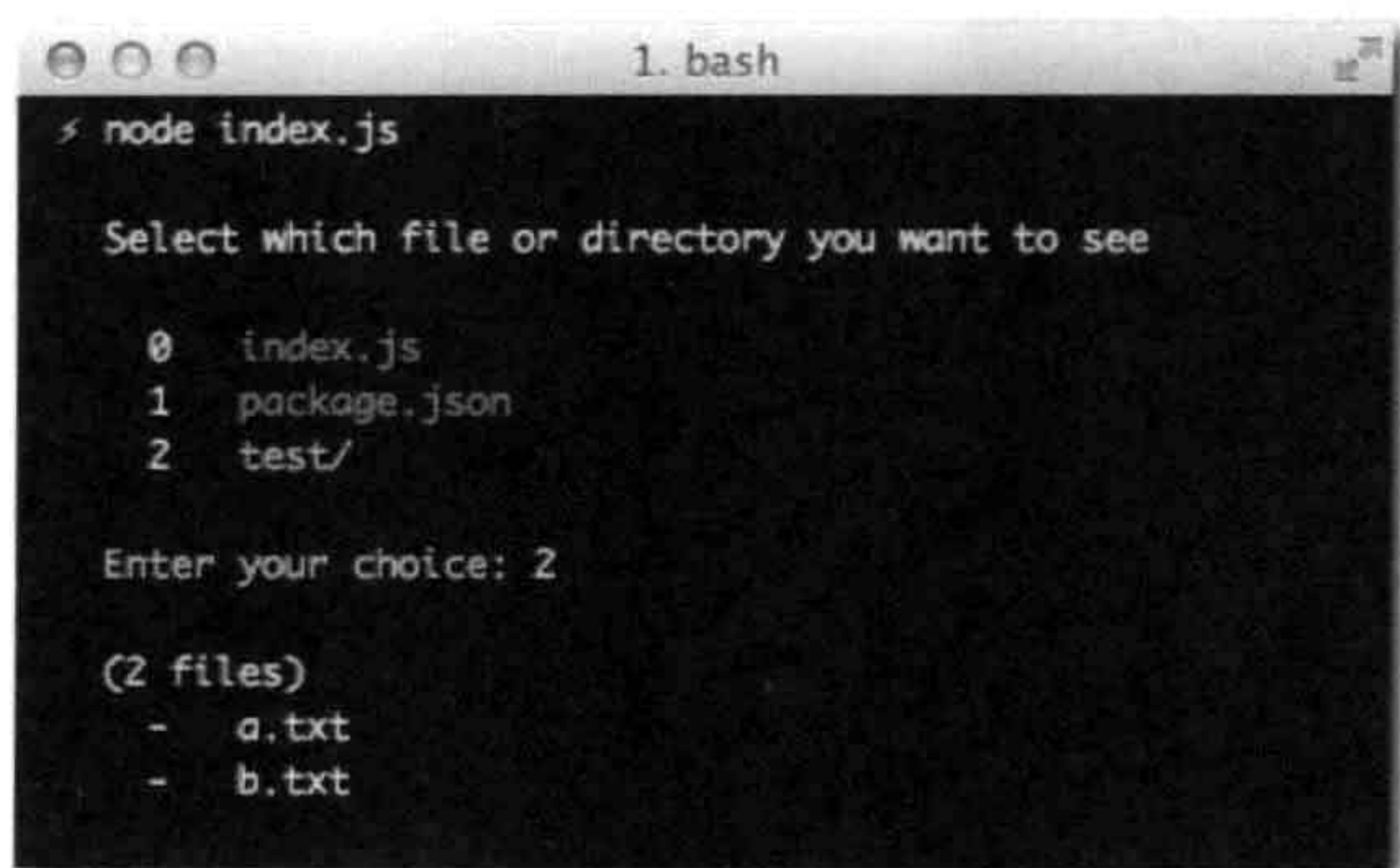


图5-11：读取`test/`文件夹的例子

完成！恭喜你完成了首个Node命令行（CLI）程序。

## 对CLI一探究竟

完成了首个命令程序之后，有必要学习一些API，它们对于书写在终端运行的类似程序很有帮助。



argv

process.argv包含了所有Node程序运行时的参数值:

```
# example.js
console.log(process.argv);
```

64

如图5-12所示, 第一个元素始终是node、第二个元素始终是执行的文件路径。紧接着是命令行后紧跟着的参数。



图5-12: 展示process.argv内容的例子

要获取这些真正的元素, 需要首先将数组的前两个元素去除掉 (见图 5-13):

```
# example-2.js
console.log(process.argv.slice(2));
```

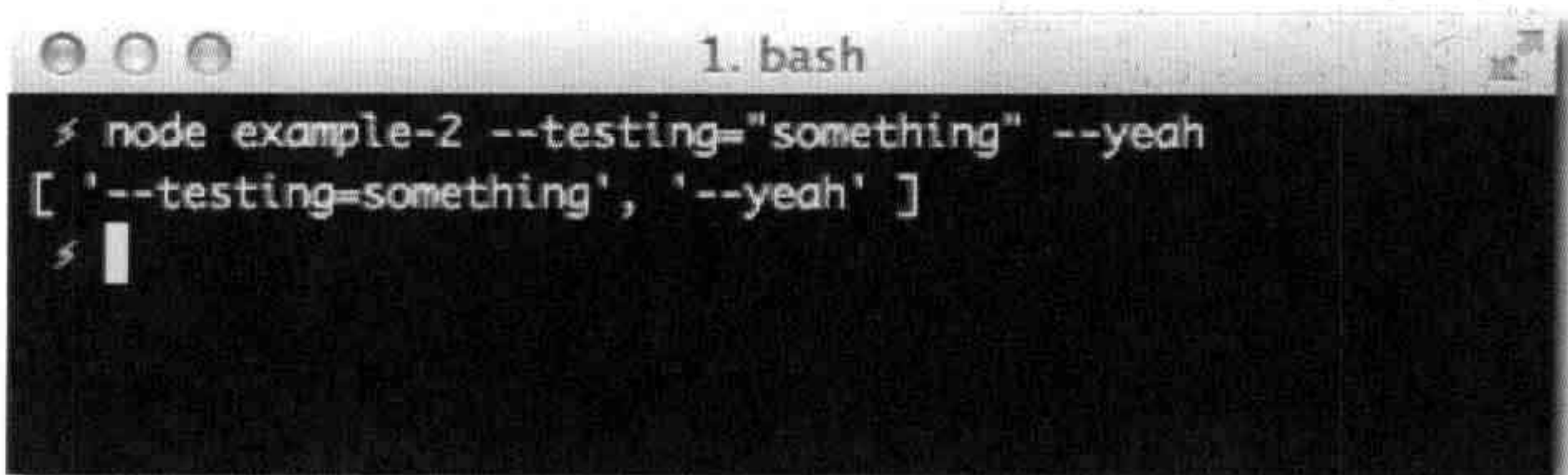


图5-13: 去除argv前两个元素, 显示真正元素的例子

接下来, 你需要学会如何获取两个不同的目录: 一个是程序本身所在的目录, 另外一个程序运行时的目录。

## 工作目录

在此前的例子中, 我们使用\_\_dirname来获取执行文件时该文件在文件系统中所在的目录。

不过, 有的时候, 更希望获得程序运行时的当前工作目录。以此前例子而言, 如果在home目录下运行该程序, 获得的当前工作目录和在其他目录下运行是一样的, 因为index.js文件的路径始终没变, 因此\_\_dirname也不会变。

要获取当前工作目录, 可以调用process.cwd方法:



```
> process.cwd()  
/Users/guillermo
```

65

Node还提供了`process.chdir`方法，允许灵活地更改工作目录：

```
> process.cwd()  
/Users/guillermo  
> process.chdir('/')  
> process.cwd()  
/
```

还有另外一个和程序运行上下文有关的方面就是环境变量。请接着往下看。

## 环境变量

Node允许通过`process.env`变量来轻松访问shell环境下的变量。

举例来说，一个最常见的环境变量就是`NODE_ENV`（见图5-14），该变量用来控制程序是运行在开发模式下还是产品模式下。

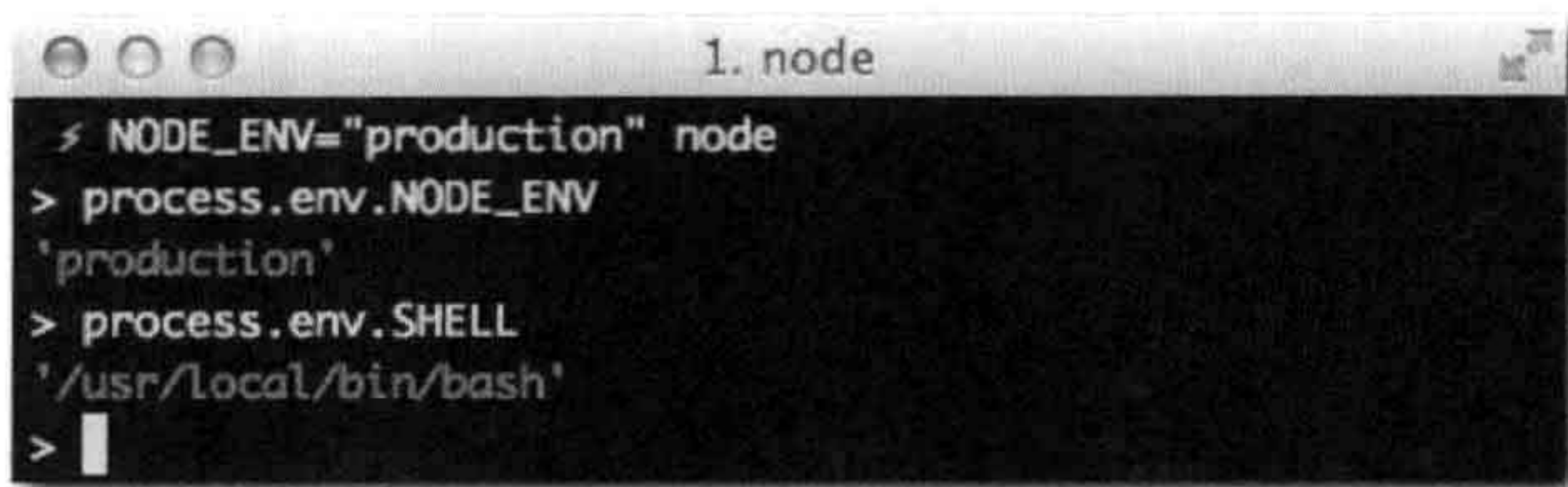


图5-14：NODE\_ENV环境变量

除此之外，在程序中控制程序自身的退出也是很有必要的。

## 退出

要让一个应用退出，可以调用`process.exit`并提供一个退出代码。比如，当发生错误时，要退出程序，这个时候最好是使用退出代码1：

```
console.error('An error occurred');  
process.exit(1);
```

这样可以让Node命令程序和操作系统中其他工具进行更好的协同。

另外还有一点就是进程信号。

## 信号

进程和操作系统进行通信的其中一种方式就是通过信号。比如，要让进程终止，可以发送SIGKILL信号。

66

Node程序是通过在`process`对象上以事件分发的形式来发送信号的：



```
process.on('SIGKILL', function () {
  // 信号已收到
});
```

接下来我们看看此前的程序是如何使用转义码让终端文本呈现不同颜色的。

## ANSI转义码

要在文本终端下控制格式、颜色以及其他输出选项，可以使用ANSI转义码。

在文本周围添加的明显不用于输出的字符，称为非打印字符。

比如，看下面这个例子：

```
console.log('\033[90m' + data.replace(/(.*)/g, '    $1') + '\033[39m');
```

- \033表示转义序列的开始。
- [表示开始颜色设置。
- 90表示前景色为亮灰色。
- m表示颜色设置结束。

或许你已经注意到了，最后用的是39，没错，这是用来将颜色再设置回去的，我们这里只想对部分文本着色。

[http://en.wikipedia.org/wiki/ansi\\_escape\\_code](http://en.wikipedia.org/wiki/ansi_escape_code)列出了一张完整的ANSI转义码表。

## 对fs一探究竟

fs模块允许你通过Stream API来对数据进行读写操作。与readFile及writeFile方法不同，它对内存的分配不是一次完成的。

比如，考虑这样一个例子，有一个大文件，文件内容由上百万行逗号分割文本组成。要完整地读取该文件来进行解析，意味着要一次性分配很大的内存。更好的方式应当是一次只读取一块内容，以行尾结束符（"\n"）来切分，然后再逐块进行解析。

下面要介绍的Node Stream就是对上述解决方案完美的实现。

## Stream

fs.createReadStream方法允许为一个文件创建一个可读的Stream对象。

为了更好地理解stream的威力，我们来看如下两个例子。

```
fs.readFile('my-file.txt', function (err, contents){
  // 对文件进行处理
});
```



上述例子中，回调函数必须要等到整个文件读取完毕、载入到RAM、可用的情况下才会触发。

而下面这个例子，每次会读取可变大小的内容块，并且每次读取后会触发回调函数：

```
var stream = fs.createReadStream('my-file.txt');
stream.on('data', function(chunk) {
    // 处理文件部分内容
});
stream.on('end', function(chunk) {
    // 文件读取完毕
});
```

为什么这种能力很重要呢？假设有个很大的视频文件需要上传到某个Web服务。这时，你无须在读取完整的视频内容后再开始上传，使用Stream就可以大大提速上传过程。

这对日志记录的例子也很有效，特别是使用可写stream。假设有个应用需要记录网站上的访问情况，这时，为了将记录写到文件中，让操作系统进行打开/关闭文件的操作可能就很低效（每次都得要在磁盘上进行查找文件操作）。

所以，这就是一个很好的使用fs.WriteStream的例子。打开文件操作只做一次，然后写入每个日志项时都调用.write方法。

另外一个很好的符合Node非阻塞设计的例子就是监视（Watch）。

## 监视

Node允许监视文件或目录是否发生变化。监视意味着当文件系统中的文件（或者目录）发生变化时，会分发一个事件，然后触发指定的回调函数。

该功能在Node生态系统中被广泛使用。举例来说，有人喜欢用一种可以编译为CSS的语言来书写CSS样式。这个时候，就可以使用监视功能，当源文件发生改变时，就将其编译为CSS文件。

68

我们来看下面这个例子。首先，查找工作目录下所有的CSS文件，然后监视其是否发生改变。一旦文件发生更改，就将该文件名输出到控制台：

```
var stream = fs.createReadStream('my-file.txt');
var fs = require('fs');
// 获取工作目录下所有的文件
var files = fs.readdirSync(process.cwd());
files.forEach(function (file) {
    // 监听“.css”后缀的文件
    if (/\.css/.test(file)) {
        fs.watchFile(process.cwd() + '/' + file, function () {
            console.log('- ' + file + ' changed!');
        });
    }
});
```



```
    });  
  }  
});
```

除了`fs.watchFile`之外, 还可以使用`fs.watch`来监视整个目录。

## 小结

本章介绍了书写Node.js程序的基础知识, 特别是如何书写一个与文件系统进行交互的命令程序。

尽管用同步的`fs` API来完成本章首个示例程序也没有什么不妥, 但本章着重是要介绍如何使用异步API来帮助大家掌握书写包含多层回调的复杂代码的方法。不管怎么样, 我们还是成功地完成了一个包含完整功能、代码整洁的应用。

本章还介绍了Node中最为重要的API之一——Stream, Stream会在本书中频繁出现。几乎所有涉及到I/O的地方都有它的身影, Stream真的非常棒。

除此之外, 本章还教会了你使用工具来创建有用的命令程序, 与文件系统、其他程序进行交互, 以及获取用户的输入。

作为Node.js开发者, 不论是写Web应用还是写更加复杂的应用, 这些API会经常使用到(特别是`process`对象上的API)。好好地记住这些API!