CHAPTER

4 Node中的JavaScript

在Node.js中写JavaScript和在浏览器中写JavaScript截然不同。Node.js除了提供和浏览器一样的基本语言之外，还在语言基础上提供了构建强大网络应用所需的API。

通过本章的介绍，你会看到一些API，这些API是Node和浏览器都有的，但却是语言本身之外、不在标准中定义的。而更重要的是，如本章标题——“Node中的JavaScript”所暗示的，本章会介绍Node.js的核心扩展相关内容。

首先我们来看看global对象的区别。

global对象

在浏览器中，全局对象指的就是window对象。在window对象上定义的任何内容都可以被全局访问到。比如，setTimeout其实就是window.setTimeout，document其实就是window.document。

Node中有两个类似但却各自代表着不同含义的对象，如下所示。

- global：和window一样，任何global对象上的属性都可以被全局访问到。

- process：所有全局执行上下文中的内容都在process对象中。在浏览器中，只有一个window对象，在Node中，也只有一个process对象。举例来说，在浏览器中窗口的名字是window.name，类似的，在Node中进程的名字是process.title。

下一章内容会深入介绍process对象，因为它提供了丰富有趣的功能，尤其是对于命令行程序来说。

实用的全局对象

浏览器中有些函数和工具虽然并非语言标准的一部分，但却非常实用，如今，它们已经被人们看作是JavaScript的一部分了。它们都是以全局的形式暴露出来的。

举例来说，setTimeout并非ECMAScript的一部分，但浏览器却仍将其视作重要的特性来实现。事实上，该函数是无法通过纯JavaScript重写的，不信的话你可以去试试。

另外有些API，人们还在讨论是否要加入到语言规范中（处在建议阶段），不过，Node.js为了让编写Node应用效率更高就把它们加进来了。setImmediate API就是一个例子，在Node中，它的作用和process.nextTick相当。

process.nextTick函数可以将一个函数的执行时间规划到下一个事件循环中：

```
console.log(1);
process.nextTick(function () {
  console.log(3);
});
console.log(2);
```

把它想象成是setTimeout(fn,1)或者“通过异步的方法在最近的将来调用该函数”，你就很容易能理解为什么上述例子的输出结果是1,2,3了。

41 还有一个类似的例子是console，console最早由Firefox中辅助开发的插件——Firebug实现。最后，Node也引入了一个全局console对象，该对象有一些如console.log和console.error这样的很有用的方法。

模块系统

JavaScript原生态是一个全局的世界。所有如setTimeout、document等这样在浏览器端使用的API，都是全局定义的。

当你引入第三方模块时，最好它们也暴露一个（或者多个）全局变量。比如，当你在HTML文档中引入<script src="http://code. jquery.com/jquery-1.6.0.js">后，你可以通过该模块上的jQuery对象来使用：

```
<script>
  jQuery(function () {
    alert('hello world!');
  });
</script>
```

之所以这样的根本原因是，JavaScript语言标准中并未为模块依赖以及模块独立定义专门的API。因此，就导致了通过这种方式引入的多个模块会出现对全局命名空间的污染及命名冲突的问题。

Node内置了很多实用的模块作为基础的工具集来帮助构建现代应用，包括http、net、fs，等等。此前在第1章“安装”中也介绍过，通过NPM你可以安装更多的模块。

Node摒弃了采用定义一堆全局变量（或者跑很多可能根本就不会用到的代码）的方式，转而引入了一个简单但却强大无比的模块系统，该模块系统有三个核心的全局对象：require、module和exports。

绝对和相对模块

这里，绝对模块是指Node通过在其内部node_modules查找到的模块，或者Node内置的如fs这样的模块。

正如在第1章中介绍的，当你安装好了colors模块，其路径就变成了./node_modules/colors。

这个时候，你就可以直接通过名字来require这个模块，无须添加路径名：

```
require('colors')
```

colors模块修改了String.prototype，因此，它无须暴露API。而fs模块，则暴露了一系列函数：

42

```
var fs = require('fs');
fs.readFile('/some/file', function (err, contents) {
  if (!err) console.log(contents);
});
```

模块还可以使用模块系统的功能来提供更加简洁独立的API以及抽象。然而，不一定非要将模块或者应用每一个部分都作为一个单独的模块和各自单独的package.json文件，你可以使用我所说的相对模块。

相对模块将require指向一个相对工作目录中的JavaScript文件。为了证明这一点，我们在同一目录中创建名为module_a.js、module_b.js以及main.js的三个文件。

module_a.js

```
console.log('this is a');
```

module_b.js

```
console.log('this is b');
```

main.js

```
require('module_a');
require('module_b');
```

然后，运行main文件（见图4-1）：

```
$ node main
```

如图4-1所示，Node未能找到module_a和module_b。原因就在于它们并没有通过NPM来安装，也不在node_modules目录中，而且 Node自带模块中没有以此为名的模块。

43

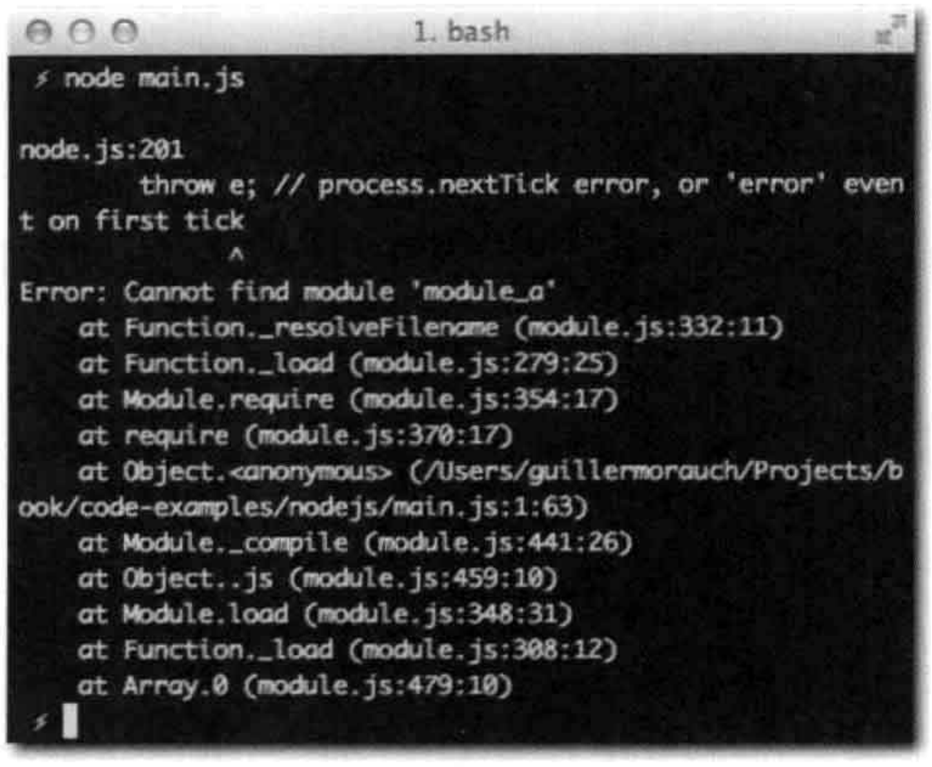

图4-1：加载module_a模块时，出错告知未能找到

要修复上述例子中这个问题，需要在require参数前加上./：

main.js

```
require('./module_a')
require('./module_b')
```

现在再次运行main文件（见图4-2）。

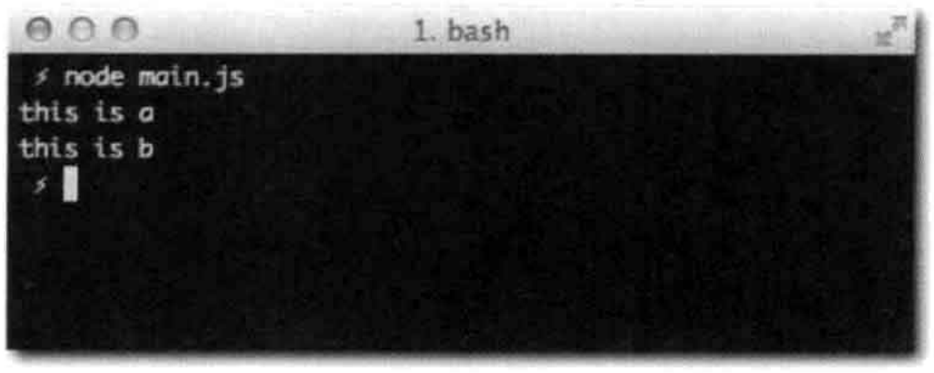

图4-2：模块加载成功

成功！这两个模块都执行了。接下来，我要介绍如何让这些模块暴露API，从而当调用require时，可以将其赋值给一个变量。

暴露API

44

要让模块暴露一个API成为require调用的返回值，就要依靠module和exports这两个全局变量。

在默认情况下，每个模块都会暴露出一个空对象。如果你想要在该对象上添加属性，那么简单地使用exports即可：

module_a.js

```
exports.name = 'john';
exports.data = 'this is some data';

var privateVariable = 5;

exports.getPrivate = function () {
  return privateVariable;
};
```

我们来测试下（见图4-3）：

index.js

```
var a = require('./module_a');
console.log(a.name);
console.log(a.data);
console.log(a.getPrivate());
```

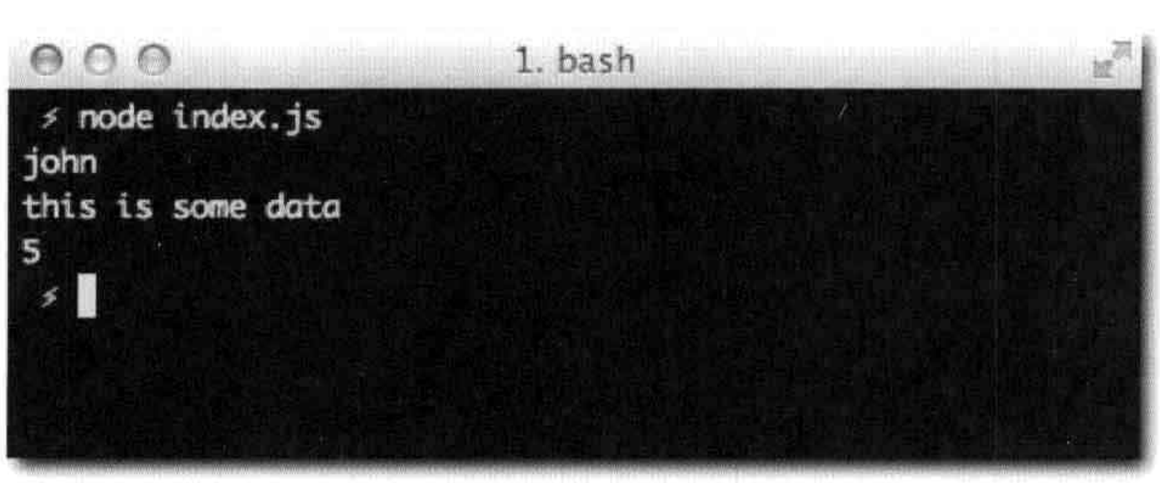

图4-3：查看module_a暴露出来的API

在上述例子中，exports其实就是对module.exports的引用，其在默认情况下是一个对象。要是在该对象上逐个添加属性无法满足你的需求，你还可以彻底重写module.exports。下面就是一个常见的将模块中构造器暴露出来的例子（见图4-4）：

45

person.js

```
module.exports = Person;

function Person (name) {
  this.name = name;
};

Person.prototype.talk = function () {
  console.log ( '我的名字是', this.name );
};
```

index.js

```
var Person = require('./person');
var john = new Person('john');
john.talk();
```

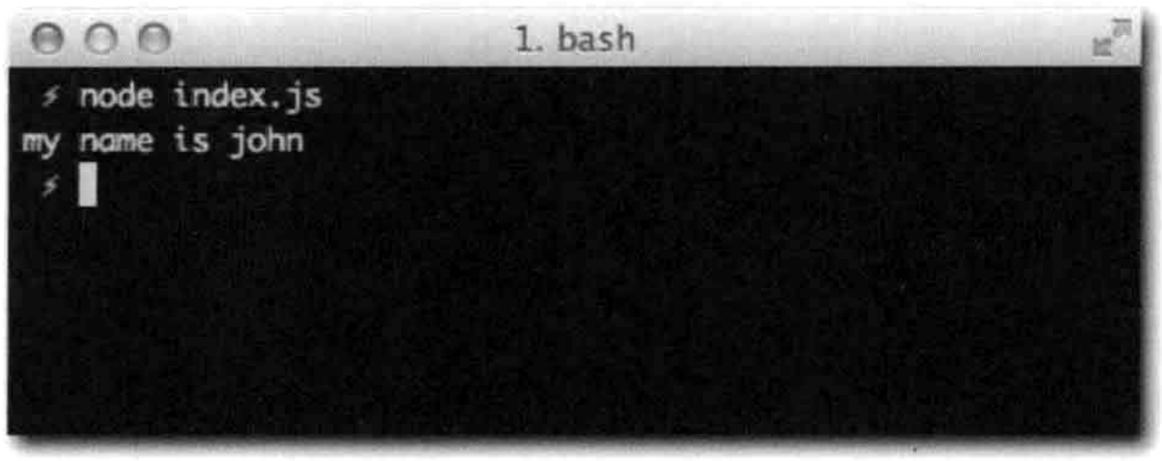

图4-4：一个具备JavaScript OOP风格的Node.js模块的例子

如上述例子所述，在index文件中，你不再是接收一个对象作为返回值，而是函数，这得归功于对module.exports的重写。

事件

Node.js中的基础API之一就是EventEmitter。无论是在Node中还是在浏览器中，大量代码都依赖于所监听或者分发的事件：

```
window.addEventListener('load', function () {
  alert ('窗口已加载！')   ;
});
```

浏览器中负责处理事件相关的DOM API主要包括addEventListener、removeEventListener以及dispatchEvent。它们还用在一系列从window到XMLHTTPRequest等的其他对象上。

46

下述例子发起一个Ajax请求（现代浏览器中），并通过监听stateChange事件来获知数据何时到达：

```
var ajax = new XMLHTTPRequest
ajax.addEventListener('stateChange', function () {
  if (ajax.readyState == 4 && ajax.responseText) {
    alert('we got some data: ' + ajax.responseText);
  }
});
ajax.open('GET', '/my-page');
ajax.send(null);
```

在Node中，你也希望可以随处进行事件的监听和分发。为此，Node暴露了Event EmitterAPI，该API上定义了on、emit以及removeListener方法。它以process.EventEmitter形式暴露出来：

eventemitter/index.js

```
var EventEmitter = require('events').EventEmitter
  , a = new EventEmitter;
a.on('event', function () {
  console.log('event called');
});
a.emit('event');
```

这个API相比DOM中的更简洁，Node内部在使用，你也可以很容易地将其添加到自己的类中：

```
var EventEmitter = process.EventEmitter
  , MyClass = function (){};

MyClass.prototype.__proto__ = EventEmitter.prototype;
```

这样，所有MyClass的实例都具备了事件功能：

```
var a = new MyClass;
a.on('某一事件', function () {
  // 做些什么
});
```

事件是Node非阻塞设计的重要体现。Node通常不会直接返回数据（因为这样可能会在等待某个资源的时候发生线程阻塞），而是采用分发事件来传递数据的方式。

我们再以HTTP服务器为例。当请求到达时，Node会调用一个回调函数，这个时候数据可能不会一下子都到达。POST请求（用户提交一个表单）就是这样的例子。

当用户提交表单时，你通常会监听请求的data和end事件：

```
http.Server(function (req, res) {
  var buf = '';
  req.on('data', function (data) {
    buf += data;
```

```
  });
  req.on('end', function () {
    console.log('数据接收完毕！');
  });
});
```

这是Node.js中很常见的例子：将请求数据内容进行缓冲（data事件），等到所有数据都接收完毕后（end事件）再对数据进行处理。

不管是否“所有的数据”都已到达，Node为了让你能够尽快知道请求已经到达服务器，都需要分发事件出来。在Node中，事件机制就是一个很好的机制，能够通知你尚未发生但即将要发生的事情。

事件是否会触发取决于实现它的API。比如，你知道了ServerRequest继承自EventEmitter，现在你也知道了它会分发data和end事件。

有些API会分发error事件，该事件也许根本不会发生。有些事件只会触发一次（如end事件），而有些则会触发多次（如data事件）。有些API只会在特定情况下触发某种事件。又比如，在特定的事件发生后，某些事件就不再触发。在上述HTTP的例子中，你肯定不希望在end事件触发后还触发data事件，否则，你的应用就会发生故障了。

同样的，有的时候，会有这样的需求：不管某个事件在将来会被触发多少次，我都希望只调用一次回调函数。Node为这类需求提供了一个名字简洁的方法：

```
a.once('某个事件'  function () {
  //尽管事件会触发多次，但此方法只会执行一次
});
```

通常，要弄明白哪些事件是可用的，以及它们的“联系方式”（即触发它们的条件），需要查看模块的API文档。本书中会介绍核心Node模块的API以及一些重要的事件，不过，带上API手册会是个不错的习惯，帮助也会很大。

buffer

除了模块之外，Node还弥补了语言另外一个不足之处，那就是对二进制数据的处理。

48

buffer是一个表示固定内存分配的全局对象（也就是说，要放到缓冲区中的字节数需要提前定下），它就好比是一个由八位字节元素组成的数组，可以有效地在JavaScript中表示二进制数据。

该功能一部分作用就是可以对数据进行编码转换。比如，你可以创建一幅用base64表示的图片，然后将其作为二进制PNG图片的形式写入到文件中：

buffers/index.js

```
var mybuffer = new Buffer('==ii1j2i3h1i23h', 'base64');
console.log(mybuffer);
require('fs').writeFile('logo.png', mybufffer);
```

base64主要是一种仅用ASCII字符书写二进制数据的方式。换句话说，它可以让你用简单的英文字符来表示像图片这样的复杂事物（所以会占用更多的硬盘空间）。

在Node.js中，绝大部分进行数据IO操作的API都用buffer来接收和返回数据。在上述例子中，filesystem模块中的`writeFile` API就接收buffer作为参数，并将其写到`logo.gif`文件中。

运行该代码，并打开gif图片（见图4-5）：

```
$ node index
$ open logo.png
```

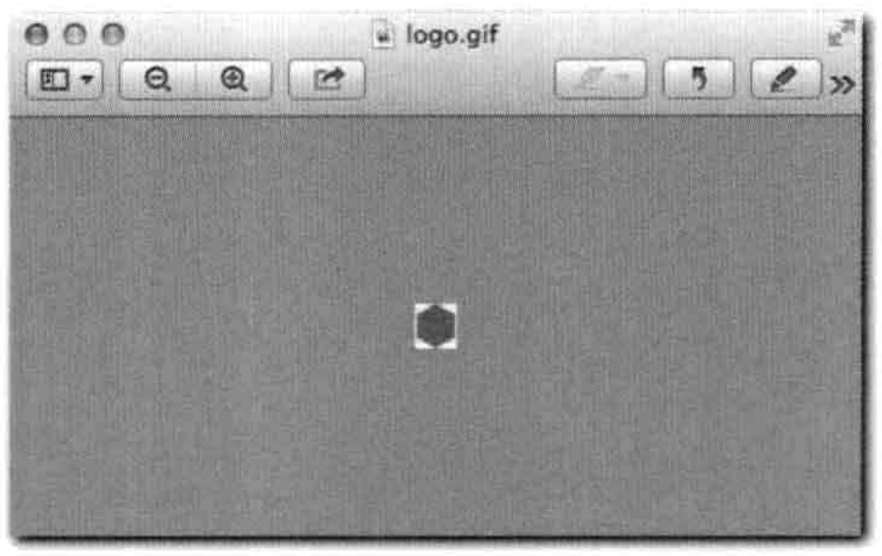

图4-5：上述脚本中，从用base64表示的buffer中创建的GIF图片显示了Node.js的logo

如上图所示，对于要调用`console.log`方法输出buffer对象而言，`writeFile`的确是个简单的接口，让原生字节数据生成图片。

小结

通过本章，你应该了解到了在浏览器端和Node.js中书写JavaScript的主要区别。

你还应该了解了Node添加的但在语言标准中没有的JavaScript中的常用API，如定时器、事件、二进制数据以及模块。

你也应该知道了Node中也有类似window的对象，也可以使用如console这样的开发者工具。