并编写如下的代码：

```
{ "name": "album-manager",
  "version": "1.0.0",
  "main": "./lib/albums.js" }
```

这是个最基本的package.json文件。它告诉npm当前这个包有一个叫做album-manager的名字，这个包的“默认”或开始JavaScript文件是存放在lib/子目录下的albums.js。

上面的目录结构没有强制性。这是包的通用布局中最简单的一种，我发现它很有用，因此一直使用它，但没有义务遵循它。但是，我还是建议以这种方式开始工作，最好对整个系统都非常熟悉之后，再开始尝试其他方式。

用来托管Node模块源代码的诸如github.com的网站如果发现有Readme文档，会自动地渲染并展示Readme文档。因此，通常都会引入一个Readme.md文件（"md"代表markdown，是github.com使用的标准文档格式）。强烈建议为模块编写一个文档用来帮助人们开始使用它。对于album-manager模块，编写如下的Readme文件：

```
# Album-Manager

This is our module for managing photo albums based on a directory. We
assume that, given a path, there is an albums sub-folder, and each of
its individual sub-folders are themselves the albums.  Files in those
sub-folders are photos.

## Album Manager

The album manager exposes a single function, `albums`, which returns
an array of `Album` objects for each album it contains.

## Album Object

The album object has the following two properties and one method:
* `name` -- The name of the album
* `path` -- The path to the album
* `photos()` -- Calling this method will return all the album's photos
```

现在我们可以开始编写真正的模块文件。首先，从约定的lib/albums.js开始，它含有第4章的album-loading的部分代码，把代码重新打包成一个类似模块的JavaScript文件：

```
var fs = require('fs'),
    album = require('./album.js');

exports.version = "1.0.0";

exports.albums = function (root, callback) {
    // we will just assume that any directory in our 'albums'
    // subfolder is an album.
    fs.readdir(
        root + "/albums",
        function (err, files) {
            if (err) {
                callback(err);
                return;
            }

            var album_list = [];

            (function iterator(index) {
                if (index == files.length) {
                    callback(null, album_list);
                    return;
                }

                fs.stat(
                    root + "albums/" + files[index],
                    function (err, stats) {
                        if (err) {
                            callback({ error: "file_error",
                                       message: JSON.stringify(err) });
                            return;
                        }
                        if (stats.isDirectory()) {
                            var p = root + "albums/" + files[index];
                            album_list.push(album.create_album(p));
                        }
                        iterator(index + 1)
                    }
                );

            })(0);
        }
    );
};
```

版本字段是模块提供的标准功能之一。虽然我并不经常使用它，但通过调用模块来检查版本并根据不同的版本执行不同的代码还是很有用的。

我们可以看到相册的功能被拆分到一个叫做/album.js的新文件中，其中有一个叫做Album的新类。这个类的代码如下所示：

```
function Album (album_path) {
    this.name = path.basename(album_path);
    this.path = album_path;
}

Album.prototype.name = null;
Album.prototype.path = null;
Album.prototype._photos = null;

Album.prototype.photos = function (callback) {
    if (this._photos != null) {
        callback(null, this._photos);
        return;
    }

    var self = this;

    fs.readdir(
        self.path,
        function (err, files) {
            if (err) {
                if (err.code == "ENOENT") {
                    callback(no_such_album());
                } else {
                    callback({ error: "file_error",
                               message: JSON.stringify(err) });
                }
                return;
            }

            var only_files = [];

            (function iterator(index) {
                if (index == files.length) {
                    self._photos = only_files;
                    callback(null, self._photos);
                    return;
                }

                fs.stat(

                  self.path + "/" + files[index],
                  function (err, stats) {
                      if (err) {
                          callback({ error: "file_error",
                                     message: JSON.stringify(err) });
                          return;
                      }
                      if (stats.isFile()) {
                          only_files.push(files[index]);
                      }
                      iterator(index + 1)
                  }
              );
            })(0);
        }
    );
};
```

如果你对上面源代码中已经使用过几次的prototype关键字感到困惑，那或许应该跳回第2章，重新回顾如何编写JavaScript类。这里的prototype关键字是为Album类的所有实例设置属性的方法。

再次说明，上面的代码与第4章中基本的JSON服务器非常相似。它们之间真正的区别在于——它被包装成一个含有原型对象以及photos方法的类。

我希望你注意到以下两件事情：

1）我们正在使用一个叫做path的新内置模块，并且使用它的basename函数用来从路径中提取相册名。

2）正如在第3章中所提到的，非阻塞的异步IO和this指针会有一些小问题，所以需要使用var self=this小技巧来帮助我们记住对象的引用。

album.js剩下的部分就很简单了，如下所示：

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

// Album class code goes here
exports.create_album = function (path) {
    return new Album(path);
};

function no_such_album() {
    return { error: "no_such_album",
             message: "The specified album does not exist" };
}
```

这就是我们需要为album-manager模块做的所有事情！如果要测试它，回到根目录并输入下面的测试程序，保存为atest.js文件：

```
var amgr = require('./album_mgr');

amgr.albums('./', function (err, albums) {
    if (err) {
        console.log("Unexpected error: " + JSON.stringify(err));
        return;
    }

    (function iterator(index) {
        if (index == albums.length) {
            console.log("Done");
            return;
        }

        albums[index].photos(function (err, photos) {
            if (err) {
                console.log("Err loading album: " + JSON.stringify(err));
                return;
            }

            console.log(albums[index].name);
            console.log(photos);
            console.log("");
            iterator(index + 1);
        });
    })(0);
});
```

现在，所需要做的事情就是确保在当前目录下有一个albums/子文件夹，运行atest.js文件，我们会看到类似下面的结果：

```
Kimidori:Chapter05 marcw$ node atest
australia2010
[ 'aus_01.jpg',
  'aus_02.jpg',
  'aus_03.jpg',
  'aus_04.jpg',
  'aus_05.jpg',
  'aus_06.jpg',
  'aus_07.jpg',
  'aus_08.jpg',
  'aus_09.jpg' ]

italy2012
[ 'picture_01.jpg',
  'picture_02.jpg',
  'picture_03.jpg',
  'picture_04.jpg',
  'picture_05.jpg' ]

japan2010
```

```
[ 'picture_001.jpg',
  'picture_002.jpg',
  'picture_003.jpg',
  'picture_004.jpg',
  'picture_005.jpg',
  'picture_006.jpg',
  'picture_007.jpg' ]

Done
```

5.4.2 使用模块进行开发

现在我们已经有一个用于相册的模块。如果想在多个项目中使用它，可以把它拷贝到其他项目的node_modules/文件夹下，但这样会遇到一个问题：当想要对相册模块做一些修改时究竟会发生什么？真的需要拷贝源代码到使用它的所有位置中并且每次都去修改它吗？

幸运的是，npm在这里派上用场了。可以修改package.json文件，增加下面的代码：

```
{ "name": "album-manager",
  "version": "1.0.0",
  "main": "./lib/albums.js",
  "private": true }
```

这些代码告诉npm不能将现在还不想发布的模块发布到外部的npm源中。

接下来使用npm link命令，这个命令会告诉npm创建一个链接，指向当前机器默认公开包库的album-manager包（例如Linux和Mac机器中的/usr/local/lib/node_modules，或Windows中的C:\Users\username\AppData\location\npm）。

```
Kimidori:Chapter05 marcw$ cd album_mgr
Kimidori:album_mgr marcw$ sudo npm link
/usr/local/lib/node_modules/album-manager ->
/Users/marcw/src/scratch/Chapter05/album_mgr
```

注意，这些都依赖于本地机器的权限以及其他设置，有可能需要

使用sudo作为超级用户来运行命令。

现在，要使用这个模块，还需做两件事情：

1）在代码中引用'album-manager'而不是'album_mgr'（因为npm使用了package.json的name字段）。

2）通过npm为每个想使用这个模块的项目创建一个到album-manager模块的引用。可以只输入npm link album-manager：

```
Kimidori:Chapter05 marcw$ mkdir test_project
Kimidori:Chapter05 marcw$ cd test_project/
Kimidori:test_project marcw$ npm link album-manager
/Users/marcw/src/scratch/Chapter05/test_project/node_modules/album-manager ->
   /usr/local/lib/node_modules/album-manager ->
   /Users/marcw/src/scratch/Chapter05/album_mgr
Kimidori:test_project marcw$ dir
drwxr-xr-x   3 marcw  staff  102 11 20 18:38 node_modules/
Kimidori:test_project marcw$ dir node_modules/
lrwxr-xr-x  1 marcw  staff   41 11 20 18:38 album-manager@ ->
   /usr/local/lib/node_modules/album-manager
```

现在，我们可以随意地修改最初的相册管理源代码，而所有引用这个模块的项目都可以立即看到这些修改。

5.4.3 发布模块

如果编写了一个模块，并想要把它共享给其他的用户，可以使用npm publish把它发布到官方的npm模块注册中心。这需要做以下几件事情：

- 删除package.json文件的"private":true这一行。
- 在npm注册服务器上使用npm adduser命令创建一个账户。
- 可以选择向package.json文件中添加更多字段（运行npm help json可以获取更多关于想添加的字段信息），例如描述、作者的联系信息和托管网站等。
- 最后，在模块的目录上运行npm publish命令，把模块发布到npm。

```
Kimidori:album_mgr marcw$ npm adduser
Username: marcwan
Password:
Email: marcwan@example.org
npm http PUT https://registry.npmjs.org/-/user/org.couchdb.user:marcwan
npm http 201 https://registry.npmjs.org/-/user/org.couchdb.user:marcwan
Kimidori:album_mgr marcw$ npm publish
npm http PUT https://registry.npmjs.org/album-manager
npm http 201 https://registry.npmjs.org/album-manager
npm http GET https://registry.npmjs.org/album-manager
npm http 200 https://registry.npmjs.org/album-manager
npm http PUT https://registry.npmjs.org/album-manager/1.0.0/-tag/latest
npm http 201 https://registry.npmjs.org/album-manager/1.0.0/-tag/latest
npm http GET https://registry.npmjs.org/album-manager
npm http 200 https://registry.npmjs.org/album-manager
npm http PUT https://registry.npmjs.org/album-manager/-/album-manager-1.0.0.tgz/
-rev/2-7f175fa335728e1cde4ce4334696bd1a
npm http 201 https://registry.npmjs.org/album-manager/-/album-manager-1.0.0.tgz/
-rev/2-7f175fa335728e1cde4ce4334696bd1a
+ album-manager@1.0.0
```

如果不小心发布了不想要发布的模块，或者想从npm模块注册中心中移除模块，可以运行npm unpublish命令：

```
Kimidori:album_mgr marcw$ npm unpublish
npm ERR! Refusing to delete entire project.
npm ERR! Run with --force to do this.
npm ERR! npm unpublish <project>[@<version>]
npm ERR! not ok code 0
Kimidori:album_mgr marcw$ npm unpublish --force
npm http GET https://registry.npmjs.org/album-manager

npm http 200 https://registry.npmjs.org/album-manager
npm http DELETE https://registry.npmjs.org/album-manager/-rev/
3-fa97bb84fa1cb8d6d6a3f57ad4a2cf2f
npm http 200 https://registry.npmjs.org/album-manager/-rev/
3-fa97bb84fa1cb8d6d6a3f57ad4a2cf2f
- album-manager@1.0.0
```

5.5 应当内置的通用模块

目前为止，我们已经在编写的代码中使用过不少Node.js的内置模块（http、fs、path、querystring和url），而且还会在本书剩下部分里使用更多的内置模块。但是，有一个模块，它可以从npm上安装，并且几乎每个编写的独立项目中都会使用到它。因此，需要花一节的篇幅在这里介绍它。

5.5.1 常见问题

考虑这种情况，我们想要编写一些异步的代码：

- 打开路径的句柄。
- 判断路径是否指向一个文件。
- 如果文件指向一个文件，加载这个文件的内容。
- 关闭文件句柄并将内容返回给调用者。

这种代码我们之前也见过，函数如下所示。被调用函数的字体加粗，回调函数的字体设置为加粗且斜体：

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

function load_file_contents(path, callback) {
    fs.open(path, 'r', function (err, f) {
        if (err) {
            callback(err);
            return;
        } else if (!f) {
            callback({ error: "invalid_handle",
                       message: "bad file handle from fs.open"});
            return;
        }
        fs.fstat(f, function (err, stats) {
            if (err) {
                callback(err);
                return;
            }
            if (stats.isFile()) {
                var b = new Buffer(10000);
                fs.read(f, b, 0, 10000, null, function (err, br, buf) {
                    if (err) {
                        callback(err);
```

```
                    return;
                }
                fs.close(f, function (err) {
                    if (err) {
                        callback(err);
                        return;
                    }
                    callback(null, b.toString('utf8', 0, br));
                });
            });
        } else {
            callback({ error: "not_file",
                       message: "Can't load directory" });
            return;
        }
    });
});
}
```

我们可以发现，即使在上示简单的例子中，代码从一开始就深度嵌套。当嵌套层数超过一定程度时，就会发现不能在80列宽的终端或打印纸中单行显示了。这会导致代码可读性降低，不知道变量在哪里被使用，很难理解函数调用和返回的流程。

5.5.2 解决方案

要解决这个问题，可以使用一个叫做async的npm模块。async提供一个直观的方式来构造和组织异步调用，并会消除一些（如果不是全部的话）在Node.js异步编程会遭遇到的诡异问题。

串行执行代码

以串行方式执行异步代码的方式有两种：分别是通过waterfall函数和series函数（见图5.1）。

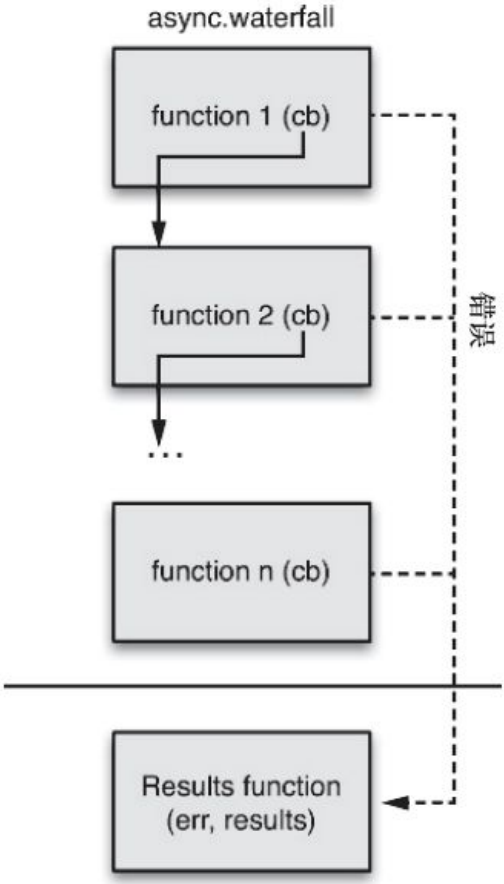

图5.1　使用async.waterfall串行执行

waterfall函数接收一个函数数组作为参数并一次一个地执行它们，然后把每个函数的结果传给下一个函数。结束时，结果函数会接收函数数组中最后一个函数的返回结果作为参数并执行。这种方式下，如果在任何一步出现错误，执行都会停止，结果函数会接收错误信息。

例如，我们可以很容易地使用async.waterfall干净利落地重写之前的代码（存放在Github源代码树中）：

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

function load_file_contents(path, callback) {
    async.waterfall([
        function (callback) {
            fs.open(path, 'r', callback);
        },
        // the f (file handle) was passed to the callback at the end of
        // the fs.open function call. async passes all params to us.
        function (f, callback) {
            fs.fstat(f, function (err, stats) {
                if (err)
                    // abort and go straight to resulting function
                    callback(err);
                else
                    // f and stats are passed to next in waterfall
                    callback(null, f, stats);
            });
        },
        function (f, stats, callback) {
            if (stats.isFile()) {
                var b = new Buffer(10000);
                fs.read(f, b, 0, 10000, null, function (err, br, buf) {
                    if (err)
                        callback(err);
                    else
                        // f and string are passed to next in waterfall
                        callback(null, f, b.toString('utf8', 0, br));
                });
            } else {
                callback({ error: "not_file",
                           message: "Can't load directory" });
            }
        },
        function (f, contents, callback) {
            fs.close(f, function (err) {
                if (err)
                    callback(err);
                else
                    callback(null, contents);
            });
        }
    ]
      // this is called after all have executed in success
      // case, or as soon as there is an error.
      function (err, file_contents) {
        callback(err, file_contents);
    });
}
```

虽然代码看起来有点长，但当我们像数组一样串行组织函数时，代码会看起来非常清晰，阅读起来也很简单。

async.series函数和async.waterfall有两个关键的不同点：

- 来自一个函数的结果不是传到下一个函数，而是收集到一个数组中，这个数组作为“结果”（第二个）参数传给最后的结果函数。依次调用的每一步都会变成结果数组中的一个元素。

■ 我们可以传给async.series一个对象，它会枚举每个key并执行每个key对应的函数。在这种方式下，结果不是作为一个数组传入，而是作为拥有相同key的对象被函数调用。

考虑下面的例子：

```
var async = require("async");

async.series({
    numbers: function (callback) {
        setTimeout(function () {
            callback(null, [ 1, 2, 3 ]);
        }, 1500);
    },
    strings: function (callback) {
        setTimeout(function () {
            callback(null, [ "a", "b", "c" ]);
        }, 2000);
    }
},
function (err, results) {
    console.log(results);
});
```

该函数会生成下面的输出结果：

```
{ numbers: [ 1, 2, 3 ], strings: [ 'a', 'b', 'c' ] }
```

并行执行

在前面的async.series例子中，函数没有理由使用串行执行顺序：第二个函数并不依赖第一个函数的结果，所以这些函数可以以并行的方式执行（见图5.2）。对于这种情况，async提供了async.parallel函数，如下所示：

```
var async = require("async");

async.parallel({

    numbers: function (callback) {
        setTimeout(function () {
            callback(null, [ 1, 2, 3 ]);
        }, 1500);
    },

    strings: function (callback) {
        setTimeout(function () {
            callback(null, [ "a", "b", "c" ]);
        }, 2000);
    }
},
function (err, results) {
    console.log(results);
});
```

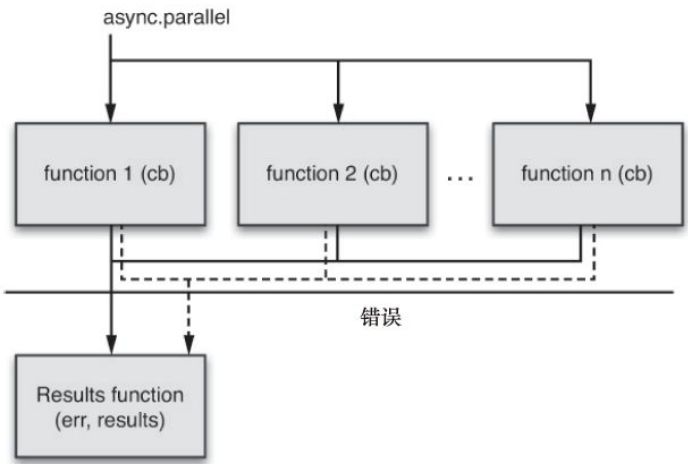

图5.2　使用async.parallel并行执行

该函数生成与之前一样的输出。

串行和并行组合起来使用

它们当中功能最强大的函数是async.auto，能够让我们将顺序执行和非顺序执行的函数混合起来成为一个功能强大的函数序列。在这里，我们传入一个对象，它的key包含了：

- 将要执行的函数，或者
- 一个依赖数组和一个将要执行的函数。这些依赖都是些字符

串，是提供给async.auto的对象的属性名。auto函数会等待这些依赖都执行完毕才会调用我们提供的函数。

async.auto函数会弄清楚所有要执行的函数的顺序，包括哪些要以并行的方式执行，而哪些需要等待其他函数先执行完（见图5.3）。就像使用async.waterfall那样，我们可以将一个函数的结果通过callback参数传给下一个函数：

```
var async = require("async");

async.auto({
    numbers: function (callback) {
        setTimeout(function () {
            callback(null, [ 1, 2, 3 ]);

        }, 1500);
    },
    strings: function (callback) {
        setTimeout(function () {
            callback(null, [ "a", "b", "c" ]);
        }, 2000);
    },
    // do not execute this function until numbers and strings are done
    // thus_far is an object with numbers and strings as arrays.
    assemble: [ 'numbers', 'strings', function (callback, thus_far) {
        callback(null, {
            numbers: thus_far.numbers.join(",  "),
            strings: "'" + thus_far.strings.join("',  '") + "'"
        });
    }]
},
// this is called at the end when all other functions have executed. Optional
function (err, results) {
    if (err)
        console.log(err);
    else
        console.log(results);
});
```

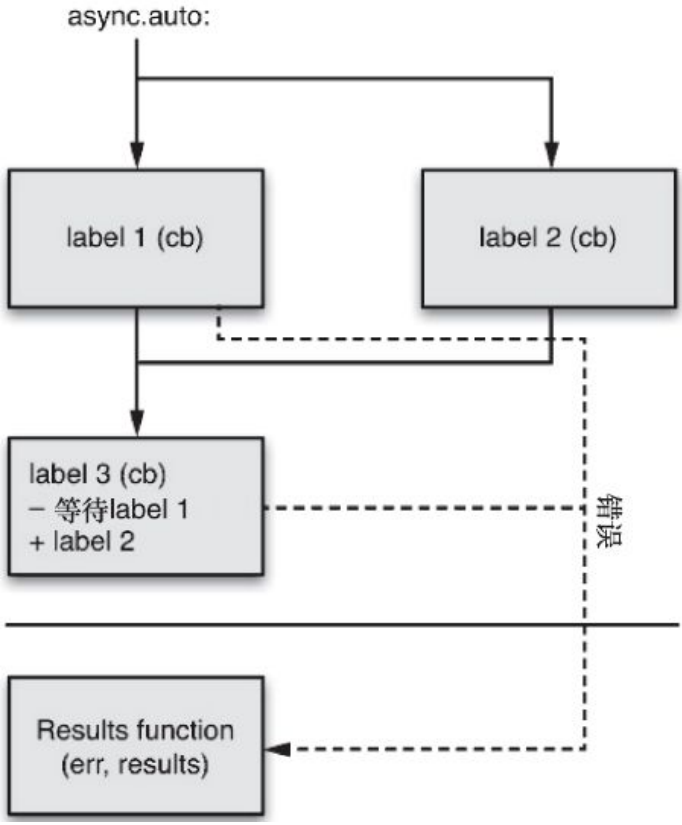

图5.3　通过async.auto实现混合执行模式

传给最后的结果函数的results参数是一个对象，这个对象的属性包含了对象中每个被执行函数的结果：

```
{ numbers: [ 1, 2, 3 ],
  strings: [ 'a', 'b', 'c' ],
  assemble: { numbers: '1,  2,  3', strings: '\'a\',  \'b\',  \'c\'' } }
```

异步循环

在第3章中，我为大家展示了如何使用下面的模式通过异步函数调用的方式来遍历数组中的项：

```
(function iterator(i) {
  if( i < array.length ) {
     async_work( function(){
       iterator( i + 1 )
     })
  } else {
    callback(results);
  }
})(0);
```

虽然这种处理方式能够很好地工作，但实际上有些怪异，它看起来比我想象中的复杂很多。async的async.forEachSeries再次帮到我们，它会遍历我们提供的数组中的每一个元素，为每一个元素调用我们给定的函数。不仅如此，它会在调用序列中的下一个元素之前等待前一个执行完毕：

```
async.forEachSeries(
    arr,
    // called for each element in arr
    function (element, callback) {
         // use element
         callback(null);  // YOU MUST CALL ME FOR EACH ELEMENT!
    },
    // called at the end
    function (err) {
        // was there an error?  err will be non-null then
    }
);
```

要简单地遍历循环中的每一个元素，然后让async等待所有的元素执行完毕，可以使用async.forEach，它会以相同的方式被调用，不同之处在于它不会串行地执行函数。

async还包含了很多功能，它是当今Node.js编程中真正不可或缺的一个模块。强烈建议大家都浏览下https://github.com/caolan/async上的文档并实际使用下。它带来了真正愉悦的Node.js编程环境，并让其变得更美好。

5.6 小结

本章我们在Node.js中正式引入了模块。虽然我们之前见过它，但现在终于知道模块是如何编写的，Node是如何发现需要引入的模块以及如何使用npm来查找并安装模块。现在可以通过package.json文件自己编写复杂的模块并把它提供给我们的所有项目，甚至可以通过npm发布模块供其他人使用。

最后，我们重点介绍了async的知识，async是最强大、最酷的模块之一，从现在开始我们编写的每一个单独的Node项目都会用到它。

接下来，我们将重新回归到Web服务器的"Web"上。我们会看一些如何在Web应用中使用JSON和Node的方法，并了解如何处理Node的其他核心技术，例如事件和数据流等。

第6章　扩展Web服务器

第二部分的最后这一章将会扩展Web服务器的一些新的关键功能，我们将学习如何处理静态内容，例如HTML页面、JavaScript文件、CSS文件甚至是图片文件。掌握了这些知识之后，就可以把焦点从服务器转移到客户端编程上。

使用Node.js编写网站，将改变传统服务器的工作模式——在发送到客户端前，将HTML生成出来。现在服务器只需处理静态文件或者JSON数据。当用户在站点上浏览时，Web浏览器能够使用AJAX调用，根据模板库生成对应的HTML页面。最后，我们会学习如何上传文件到服务器，并看一些能让这项工作变得简单的工具。

首先从学习Node.js的数据流开始。

6.1 使用Stream处理静态内容

在Node.js异步、非阻塞IO的世界里，我们已经看过在循环中使用fs.open以及fs.read来读取文件的内容。不过，Node.js提供了另一种更为优雅、用来读取（甚至写入）文件的机制，这就是Stream。它和UNIX pipe所扮演的角色很像——我们已经在4.6.1节加载用户数据的场景中，看到过它的简单用法。

Stream最基本的用法是使用on方法，将监听函数（listener）添加到事件（event）上。当事件触发时就会调用所提供的函数。readable事件会在输入流（read stream）读取了进程里的一些内容之后触发。end事件在Stream不再进行内容读取时触发，而error事件会在错误发生时触发。

6.1.1 读取文件

一个简单的例子，编写下面的代码并保存为simple_stream.js：

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

// INCEPTION BWAAAAAAA!!!!
var rs = fs.createReadStream("simple_stream.js");

rs.on('readable', function () {
    var str;
    var d = rs.read();
    if (d) {
        if (typeof d == 'string') {
            str = d;
        } else if (typeof d == 'object' && d instanceof Buffer) {
            str = d.toString('utf8');
        }
        if (str) {
            if (!contents)
                contents = d;
            else
                contents += str;
        }
    }
});

rs.on('end', function () {
    console.log("read in the file contents: ");
    console.log(contents.toString('utf8'));
});
```

如果看了上面的代码（创建一个对象，添加两个监听函数，之后似乎就没再做什么），但不知道为什么代码没有在加载结束时退出，那么请回顾第3章，我曾经提到过Node运行在一个事件循环中并等待某些事情发生，而当事情最终发生时，就去执行相应的代码。

持续变化的Stream

当Node.js发布0.10版本时，Stream经历了重大变化。在这个版本之前，Stream曾有一些很好的功能，但因为计时（timing）和暂停（pausing）等关键问题，这些功能都被废弃了。经过几年的推迟，Node的核心团队最终解决和修复了这些问题，并规避了相关应用的风险。

在这之前，如果想要使用输入流，需要添加data事件的监听函数，让Stream读取的数据作为参数传递给回调函数。而现在，只需监听readable事件并在事件通知数据已经可读时，调用这个数据流的read方法即可。

如果看到任何使用旧的data事件的代码，不需要为此担心，因为新模型兼容99%的老代码，我只是想让你知道下面会用到这两种模式。

现在就很清楚了，当事件挂起或者即将发生时，例如有个输入流处于打开状态并且正在调用文件系统等待内容读取完成时，就会发生这样的情况。只要预期会有事情发生，Node就会在所有的事件都已完成且用户的代码都已执行完后才退出。

之前的例子通过fs.createReadStream函数和提供的路径创建一个新输入流，然后简单地读取并将内容打印到输出流。当监听到readable事件时，调用Stream的read方法，取回任何当前可见的数据。如果没有返回数据，它会等待，直至监听到另一个readable事件或者接收到一个end事件。

使用Buffer操作二进制数据

到目前为止，我们已经使用过Node.js字符串（通常是UTF-8字

符串）来工作。但在使用数据流和文件时，实际上主要是在与Buffer类打交道。

Buffer和我们期望的多少有些类似：Buffer暂存二进制数据，将数据转换成其他格式，或将数据写入文件，或将数据打散并重新组合。

关于缓冲需要提醒的重要一点是，缓冲对象的length属性并不会返回内容的实际大小，而是返回缓冲本身的大小！例如：

```
var b = new Buffer(10000);
var str = "我叫王马克";
b.write(str); // default is utf8, which is what we want
console.log( b.length ); // will print 10000 still!
```

Node.js不会在缓冲中跟踪把什么数据写入到什么地方，因此，必须自己跟踪写入的数据。

有时字符串的字节长度和字符数不一定相同——例如上面的字符串“我叫王马克”。Buffer.byteLength返回结果如下：

```
console.log( str.length );                 // prints 5
console.log( Buffer.byteLength(str) );     // prints 15
```

要将缓冲转换成字符串，需使用toString方法。一般都是将缓冲转换成UTF-8字符串：

```
console.log(buf.toString('utf8'));
```

为了将一个缓冲插入到另外一个的末尾，要使用concat方法，如下所示：

```
var b1 = new Buffer("My name is ");
var b2 = new Buffer("Marc");
var b3 = Buffer.concat([ b1, b2 ]);
console.log(b3.toString('utf8'));
```

最后，可以“清零”或者使用fill方法填充缓冲中所有的值，例如buf.fill("\0")。

6.1.2 在Web服务器中使用Buffer处理静态文件

接下来，我们练习编写一个小小的Web服务器，这个服务器会使用Node的Buffer来处理静态内容（一个HTML文件）。可以从handle_incoming_request函数开始：

```
function handle_incoming_request(req, res) {
    if (req.method.toLowerCase() == 'get'
        && req.url.substring(0, 9) == '/content/') {
        serve_static_file(req.url.substring(9), res);
    } else {

        res.writeHead(404, { "Content-Type" : "application/json" });

        var out = { error: "not_found",
                    message: "'" + req.url + "' not found" };
        res.end(JSON.stringify(out) + "\n");
    }
}
```

如果传入的请求访问的是/content/something.html，则会尝试调用serve_static_file函数来处理。Node.js的http模块设计得足够聪明，服务器上每次收到的请求对应的ServerResponse对象本身实际上就是一个数据流，我们可以在其上编写对应的输出。可以通过调用Stream类的write方法来完成：

```
function serve_static_file(file, res) {
    var rs = fs.createReadStream(file);
    var ct = content_type_for_path(file);
    res.writeHead(200, { "Content-Type" : ct });

    rs.on(
        'readable',
        function () {
            var d = rs.read();
            if (d) {
                if (typeof d == 'string')
                    res.write(d);
                else if (typeof d == 'object' && d instanceof Buffer)
                    res.write(d.toString('utf8'));
            }
        }
    );

    rs.on(
        'end',
        function () {
            res.end();  // we're done!!!
        }
    );
}

function content_type_for_path(file) {
    return "text/html";
}
```

剩余的服务器代码就与之前见到的一样了：

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

var s = http.createServer(handle_incoming_request);
s.listen(8080);
```

新建一个叫做test.html的文件，它含有一些简单的HTML内容。随后在服务器上运行

```
node server.js
```

接着使用curl请求test.html：

```
curl -i -X GET http://localhost:8080/content/test.html
```

应该看到和下面类似的输出（这取决于真正写到test.html里的内容）：

```
HTTP/1.1 200 OK
Date: Mon, 26 Nov 2012 03:13:50 GMT
Connection: keep-alive
Transfer-Encoding: chunked

<html>
<head>
  <title> WooO! </title>
</head>
<body>
  <h1> Hello World! </h1>
</body>
</html>
```

现在我们能够处理静态内容了！但还有两个问题。首先，如果请求/content/blargle.html，那会发生什么？当前的脚本会抛出一个错误并终止，这并不是我们想要的结果。这种情况下，我们期望返回404的HTTP状态码以及可能的错误信息。

为此，可以监听输入流的error事件。添加以下几行代码到serve_static_file函数中：

```
rs.on(
    'error',
    function (e) {
        res.writeHead(404, { "Content-Type" : "application/json" });
        var out = { error: "not_found",
                    message: "'" + file + "' not found" };
        res.end(JSON.stringify(out) + "\n");
        return;
    }
);
```

现在，捕捉到错误后（之后没有data或者end事件被调用），就更新响应头的代码为404，设置新Content-Type，并返回包含错误信息的JSON，客户端可以使用这些JSON信息告诉用户发生了什么错误。

6.1.3 不仅仅支持HTML

第二个问题是现在只能处理HTML格式的静态内容。content_type_for_path函数只能返回"text/html"。如果能够更灵活

则会更好，可以通过以下的代码完成这一需求：

```
function content_type_for_file (file) {
    var ext = path.extname(file);

    switch (ext.toLowerCase()) {
        case '.html': return "text/html";
        case ".js": return "text/javascript";
        case ".css": return 'text/css';
        case '.jpg': case '.jpeg': return 'image/jpeg';
        default: return 'text/plain';
    }
}
```

现在可以对不同文件类型调用curl命令并得到预期的结果。对于类似JPEG图片的二进制文件，可以对curl使用-o参数来告诉curl将输出写入到指定的文件中。首先，复制一个JPEG文件到当前文件夹中，通过node server.js运行服务器，随后输入下面的代码：

```
curl -o test.jpg http://localhost:8080/content/family.jpg
```

机会来了，现在有一个叫做test.jpg的文件，它正是我们想要的。

从数据流（上面例子中的rs）到数据流（res）的数据传递是很常见的场景，Node.js的Stream类有一个很便捷的方法来为你做这件事：pipe。这使得serve_static_file函数变得更简单：

```
function serve_static_file(file, res) {
    var rs = fs.createReadStream(file);
    rs.on(
        'error',
        function (e) {
            console.log("oh no! Error!! " + JSON.stringify(e));
            res.end("");
        }
    );

    var ct = content_type_for_path(file);
    res.writeHead(200, { "Content-Type" : ct });
    rs.pipe(res);
}
```

遗憾的是，pipe有一个小问题：一旦调用pipe，它会立即发送HTTP响应头，发送之后，后面就无法再调用ServerResponse.writeHead了。如果想要通过