3.3 错误处理和异步函数

在前一章中，我们讨论了JavaScript中的错误处理、事件和try...catch代码块。在本章中，我们将会介绍非阻塞IO和异步函数回调，但这也会带来新的问题。请思考如下代码：

```
try {
    setTimeout(function () {
        throw new Error("Uh oh!");
    }, 2000);
} catch (e) {

    console.log("I caught the error: " + e.message);
}
```

如果运行以上代码，期望能看到输出结果为“I caught the error:Uh oh!”。但是可惜的是，我们实际上会看到：

```
timers.js:103
            if (!process.listeners('uncaughtException').length) throw e;
                                                                ^
Error: Uh oh, something bad!
    at Object._onTimeout errors_async.js:5:15)
    at Timer.list.ontimeout (timers.js:101:19)
```

为什么会这样？难道try...catch代码块不会捕获错误么？不，它会捕获错误。但是，异步回调会给捕获错误带来一点小麻烦。

事实上，对setTimeout的调用的确是在try...catch代码块中执行的。如果函数抛出错误，catch就会捕获它，然后你就能看到期望的结果了。但是setTimeout函数只是在Node的事件队列中添加了一个新事件（是为了告诉Node在给定的等待时间以后调用该函数——在本例中是2000ms），然后返回。而这个回调函数实际上是在一个全新的上下文和作用域中执行的。

因此，当在非阻塞IO中调用异步函数的时候，只有极小一部分会抛出错误；大部分情况下，编译器会告诉你出错了。

在Node中，我们使用一些核心模式（core pattern）来帮助规范化编码，以避免出错。这些模式不是JavaScript这门语言或者引擎

强制要求的，但经常可以在项目中看到这些模式，甚至你也会自己使用到这些模式。

回调函数和错误处理

你第一个会看到的模式就是格式化传递给异步函数使用的回调函数。回调函数一般至少包含一个参数，即最后操作的成功或者失败状态；一般也会包含第二个参数，即最后操作返回的结果或信息（比如文件句柄、数据库连接、查询到的数据集等）；一些回调函数还可能包含更多的参数：

```
do_something(param1, param2, ..., paramN, function (err, results) { ... });
```

参数err的值一般会是：

- null，表明操作成功，并且会有一个返回值（如果有需要的话）。

- 一个Error对象的实例，你偶尔会看到一些不一致的地方：有些人喜欢在Error对象上添加code字段，并且用message字段保存错误信息；反之，有些人喜欢用一些其他的模式。本书中的所有代码使用的模式都会包含code字段，并使用message字段来尽可能提供更多的错误信息。所有的模块中code字段都是字符串类型，因为这样更具可读性。一些类库则会在Error对象中提供更多的信息，但至少要包含上述两个字段。

这种方式可以让我们写出的非阻塞代码更具可控性。整本书中，我将会使用回调函数中处理错误的两种不同的代码风格。下面是第一种：

```
fs.open(
    'info.txt', 'r',
    function (err, handle) {
        if (err) {
            console.log("ERROR: " + err.code + " (" + err.message ")");
            return;
        }
        // success!! continue working here
    }
);
```

这种风格需要先检查错误，如果有错误就直接返回；否则，继续

处理结果。接下来是第二种风格：

```
fs.open(
    'info.txt', 'r',
    function (err, handle) {
        if (err) {
            console.log("ERROR: " + err.code + " (" + err.message ")");
        } else {
            // success! continue working here
        }
    }
);
```

这种风格中，使用if...then...else语句来处理错误。

这两者之间的区别看似有些微不足道，但是前者更容易产生bug和错误，因为可能会忘记在if语句中添加return语句。但是后者会让代码不断地缩进，这样会导致每行的代码冗长且可读性差。当然，你会在第5章中找到这个问题的解决方案。

全新的包含错误处理的文件加载代码如代码清单3.1所示。

代码清单3.1　包含完整错误处理的文件加载

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

fs.open(
    'info.txt', 'r',
    function (err, handle) {
        if (err) {
            console.log("ERROR: " + err.code
                        + " (" + err.message + ")");
            return;
        }
        var buf = new Buffer(100000);
        fs.read(
            handle, buf, 0, 100000, null,
            function (err, length) {

                if (err) {
                    console.log("ERROR: " + err.code +
                                " (" + err.message + ")");
                    return;
                }
                console.log(buf.toString('utf8', 0, length));
                fs.close(handle, function () { /* don't care */ });
            }
        );
    }
);
```

3.4 我是谁——如何维护本体

现在，需要写一个小小的类来处理一些普通的文件操作。

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

function FileObject () {
    this.filename = '';

    this.file_exists = function (callback) {
        console.log("About to open: " + this.filename);
        fs.open(this.filename, 'r', function (err, handle) {
            if (err) {
                console.log("Can't open: " + this.filename);
                callback(err);
                return;
            }
            fs.close(handle, function () { });
            callback(null, true);
        });
    };
}
```

上述代码中添加了一个属性——filename和一个方法——file_exists。该方法会做如下事情：

- 尝试以只读方式打开filename属性指定的文件。
- 如果文件不存在，则打印日志信息，并把err作为参数来调用回调函数。
- 如果文件存在，调用回调函数，以表明打开文件成功。

现在，在下述代码中使用该类：

```
var fo = new FileObject();
fo.filename = "file_that_does_not_exist";

fo.file_exists(function (err, results) {
    if (err) {

        console.log("Aw, bummer: " + JSON.stringify(err));
        return;
    }

    console.log("file exists!!!");
});
```

希望看到如下输出结果：

```
About to open: file_that_does_not_exist
Can't open: file_that_does_not_exist
```

但事实上，你会看到：

```
About to open: file_that_does_not_exist
Can't open: undefined
```

怎么回事？大多数情况下，当一个函数嵌套在另一个函数中时，它就会自动继承父/宿主函数的作用域，因而就能访问所有的变量了。那么，为什么嵌套的回调函数却没有返回正确的filename属性的值呢？

这个问题归根于this关键字和异步回调函数本身。别忘了，当你调用fs.open这样的函数的时候，它会首先初始化自己，然后调用底层的操作系统函数（本例中，就是打开文件），并把回调函数插入到事件队列中去。执行完会立即返回给FileObject#file_exists函数，然后退出。当fs.open函数完成任务后，Node就会调用该回调函数，但此时，该函数已经不再拥有FileObject这个类的继承关系了，所以回调函数会被重新赋予新的this指针。

坏消息是，在这一过程中已经丢失了指向FileObject的this指针。但好消息是，fs.open的回调函数还保留着它的函数作用域。一个常用的解决方法就是把消失的this指针“保存”到变量中，变量名可以是self、me或者其他类似的名称。现在，重写file_exists函数并利用this指针：

```
this.file_exists = function (callback) {
    var self = this;

    console.log("About to open: " + self.filename);
    fs.open(this.filename, 'r', function (err, handle) {
        if (err) {
            console.log("Can't open: " + self.filename);
            callback(err);
            return;
        }

        fs.close(handle, function () { });
        callback(null, true);
    });
};
```

由于函数作用域是通过闭包保留的，所以self变量会被一直保持着，即使回调函数是被Node.js异步执行的。在后面的应用中，this会有更广泛的使用场景。有些人喜欢用me而不是self，因为这一命名更简短；而有些人则会使用其他完全不同的命名。你可以选择一个喜欢的命名，然后一直用下去，以保持一致性。

3.5 保持优雅——学会放弃控制权

Node运行在单线程中，使用事件轮询来调用外部函数和服务。它将回调函数插入事件队列中来等待响应，并且尽快执行回调函数。那么，如果一个函数需要计算两个数组的交叉元素，会发生些什么呢：

```
function compute_intersection(arr1, arr2, callback) {
    var results = [];
    for (var i = 0 ; i < arr1.length; i++) {
        for (var j = 0; j < arr2.length; j++) {
            if (arr2[j] == arr1[i]) {
                results[results.length] = arr1[j];
                break;
            }
        }
    }
    callback(null, results);   // no error, pass in results!
}
```

当面对拥有数千个元素的数组的时候，该函数就会耗费大量的计算时间，大约会花费1秒甚至更多。在单线程模式中，Node.js在同一时间只做一件事情，所以，这个数量级的时间就会成为一个问题。像一些计算哈希、摘要（digest）或者一些其他消耗时间的操作，都会导致应用在处理过程中处于“假死”状态。那么，我们能做些什么呢？

在本书的前言中，我曾提到，Node.js不会特别适合某些特定的应用场景，其中之一就是作为计算服务器。Node更适合一些常见的网络应用任务，比如那些需要大量I/O或者需要向其他服务请求的任务。如果你想要编写一个需要大量计算的服务器，可能需要考虑把这些操作迁移到其他服务上去，这样Node应用就可以远程调用了。

但是，这并不意味着Node必须避免高强度计算的任务。如果只是偶尔执行这种任务，那么就可以在Node.js中使用它们，并且可以利用全局对象process中的nextTick方法。这种方法就好像在跟系统说“我放弃执行控制权，你可以在你空闲的时候执行我提供给你的函数”。相较于使用setTimeout函数，这种方式会显著提高执行速

度。

代码清单3.2是最新版本的compute_intersection函数，可以让Node每隔一段时间就处理一次其他任务。

代码清单3.2　使用process#nextTick，让代码变优雅

```
function compute_intersection(arr1, arr2, callback) {
    // let's break up the bigger of the two arrays
    var bigger = arr1.length > arr2.length ? arr1 : arr2;
    var smaller = bigger == arr1 ? arr2 : arr1;
    var biglen = bigger.length;

    var smlen = smaller.length;

    var sidx = 0;           // starting index of any chunk
    var size = 10;          // chunk size, can adjust!
    var results = [];       // intermediate results

    // for each chunk of "size" elements in bigger, search through smaller
    function sub_compute_intersection() {
        for (var i = sidx; i < (sidx + size) && i < biglen; i++) {
            for (var j = 0; j < smlen; j++) {
                if (bigger[i] == smaller[j]) {
                    results.push(smaller[j]);
                    break;
                }
            }
        }

        if (i >= biglen) {
            callback(null, results);   // no error, send back results
        } else {
            sidx += size;
            process.nextTick(sub_compute_intersection);
        }
    }

    sub_compute_intersection();
}
```

在这个新版的函数中，只需简单地将较大的输入数组分割成10个元素一组的数据块（可以是任意大小的数据块），分别计算交叉元素，然后调用process#nextTick函数，从而允许Node处理其他事件或者请求。只有当该任务的队列前面没有事件时，才会继续执行该任务。别忘记将回调函数sub_compute_intersection传递给process#nextTick，这样才能保证当前作用域能被作为闭包保存下来。通过这种方式，我们就能将中间结果保存到compute_intersection函数中的变量中了。

代码清单3.3展示的是compute_intersection函数的测试代码。

代码清单3.3　测试compute_intersection函数

```
var a1 = [ 3476, 2457, 7547, 34523, 3, 6, 7,2, 77, 8, 2345,
           7623457, 2347, 23572457, 237457, 234869, 237,
           24572457524] ;
var a2 = [ 3476, 75347547, 2457634563, 56763472, 34574, 2347,
           7, 34652364 , 13461346, 572346, 23723457234, 237,
           234, 24352345, 537, 2345235, 2345675, 34534,
           7582768, 284835, 8553577, 2577257,545634, 457247247,
           2345 ];

compute_intersection(a1, a2, function (err, results) {

    if (err) {
        console.log(err);
    } else {
        console.log(results);
    }
});
```

尽管现在已经比最初的计算交叉元素的函数版本要稍许复杂，但在单线程的Node事件处理和回调的世界中已经运行得非常好了。当然，我们可以在任何复杂的、计算速度慢的场景中使用process.nextTick。

3.6 同步函数调用

至此，我已经花了几乎整个章节介绍Node.js是如何异步运行的，并且介绍了很多技巧和非阻塞IO编程中的陷阱。但我必须提醒，Node确有一些核心API的同步版本，尤其是在操作文件的API中。在第11章中，我们将会使用这些同步API编写命令行工具。

为了扼要说明，可以重写本章的第一个脚本，如下所示：

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

var handle = fs.openSync('info.txt', 'r');
var buf = new Buffer(100000);
var read = fs.readSync(handle, buf, 0, 10000, null);
console.log(buf.toString('utf8', 0, read));
fs.closeSync(handle);
```

在你阅读本书的过程中，我希望你能意识到Node.js不仅仅可以编写网络或者Web应用，还可以使用它做任何其他一些事情，包括命令行工具、原型设计、服务器管理等。

3.7 小结

传统的编程模型在执行一系列同步、阻塞IO函数调用时，会等待这些函数执行完成才会继续执行下去；而Node的编程模型则是异步执行，当任务执行完成后，会由Node进行通知，并不会阻塞其他任务的执行。从前者到后者的转变需要大家多思考并付诸实践。不过，我坚信一旦掌握了这些窍门，你便再也不想回到以前那种编写Web应用的方式中去了。

下一章，我们将要开始尝试编写第一个简单的JSON应用服务器。

第二部分　提高篇

第4章　编写简单应用

现在我们已经对JavaScript语言如何工作有了深入的了解，现在是时候运用Node.js编写Web应用了。正如本书前言中提到的，我们将会在整本书中编写一个小型的相册网站。本章中，我们将会编写一个JSON服务器，用以提供相册列表以及每个相册对应的照片列表等服务，最后，还会添加为相册重命名的功能。在这个过程中，我们会理解JSON服务器运行的基本知识，这其中包含了服务器与HTTP进行交互的基础知识，例如GET和POST参数、头信息、请求和响应。第一代相册应用会使用文件相关的API来完成工作，这是用来理解上述知识点的最佳方式，可以使我们专注于要学习的新概念。

4.1 第一个JSON服务器

在第1章的最后，我们编写了一点关于HTTP服务器的代码，对于任何传入的请求都会返回纯文本"Thanks for calling!\n"。现在我们可以对其稍微做些修改，让它变得有些不同：

1）指明返回的数据格式是application/json，而不是text/plain。

2）使用console.log打印获取到的请求。

3）返回JSON字符串。

这是一个非常小的服务器，它保存到simple_server.js文件中：

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

function handle_incoming_request(req, res) {
    console.log("INCOMING REQUEST: " + req.method + " " + req.url);
    res.writeHead(200, { "Content-Type" : "application/json" });
    res.end(JSON.stringify( { error: null }) + "\n");
}

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

通过终端窗口（Mac/Linux）或者命令提示符（Windows）运行程序，如下所示：

```
node simple_server.js
```

上述代码只是等待请求而不会做其他事情。现在，在另外一个终端窗口中输入以下代码：

```
curl -X GET http://loc.lhost:8080
```

如果一切都没有问题，在运行服务器的窗口中应该能够看到如下信息：

```
INCOMING REQUEST: GET /
```

而在运行curl命令的窗口，则应该能看到：

```
{"error":null}
```

可以尝试运行不同的curl命令，并观察都会发生什么。例如：

```
curl -X GET http://localhost:8080/gobbledygook
```

随着第一个程序的运行，我们可以规范JSON响应的输出，在输出中都会有一个error字段。通过这种方式，应用可以快速判断请求是成功还是失败。万一出现失败的情况，则会包含一个message字段，用来存放更多的错误信息；反之，JSON响应会返回数据，并会包含一个data字段：

```
// failure responses will look like this:
{ error: "missing_data",
  message: "You must include a last name for the user" }

// success responses will usually have a "data" object
{ error: null,
  data: {
      user: {
          first_name: "Horatio",
          last_name: "Gadsplatt III",
          email: "horatio@example.org"
      }
  }
}
```

一些应用选择使用数字编码来对应它们的错误类型。是否使用这种方式完全取决于你自己，但是我更倾向于使用文本，因为文本更具描述性，而且能够在使用类似curl等命令行测试程序时省下一个查询错误类型的步骤。no_such_user显然比-325来得更翔实些。

数据返回

一开始，我们的相册应用相当简单：它实际上是一个相册的集合，每个相册则是照片的集合，如图4.1所示：

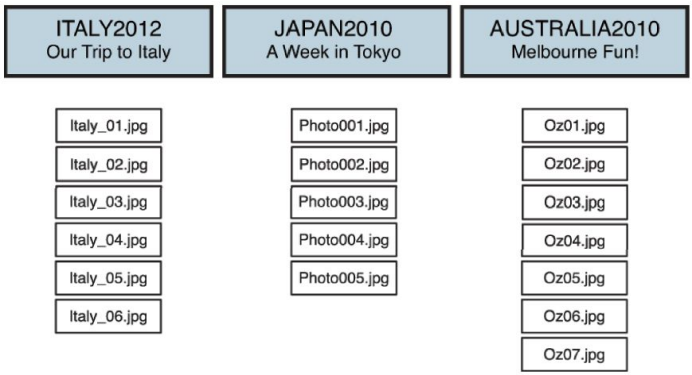

图4.1　相册和照片

现在，所有的相册都在执行脚本所在位置的子文件夹albums/下的子文件夹下：

```
scripts/
scripts/albums/
scripts/albums/italy2012
scripts/albums/australia2010
scripts/albums/japan2010
```

因此，为了获取相册列表，我们只需找到albums/子文件夹下的项。可以使用fs.readdir函数，这个函数会返回指定文夹件下的所有项（除了“.”和“..”）。load_album_list函数的代码如下所示：

```
function load_album_list(callback) {
    fs.readdir(
        "albums/",
        function (err, files) {
            if (err) {
                callback(err);
                return;
            }
            callback(null, files);
        }
    );
}
```

我们仔细地阅读下这个函数的代码。首先，函数调用fs.readdir函数并提供了一个回调函数，当目录下的所有项都加载完毕后，就调

用这个函数。该回调函数拥有大多数回调都有的原型：一个error参数和一个result参数，我们可以在任何需要的地方使用这些参数。

注意，load_album_list函数本身的唯一参数是一个回调函数。因为load_album_list函数本身是异步的，它需要知道当自己完成工作之后要将相册列表传递到哪里。它不能将结果返回给调用者，因为它会在fs.readdir函数调用回调并给出结果之前就已经执行完毕。

这就是Node应用编程的核心技术：告诉Node去做某件事情，并在Node完成时告知Node将结果传递给谁。而与此同时，我们继续执行其他的工作。很多执行的任务基本上都是以一系列长长的回调作为结束。

代码清单4.1包含了新相册列表服务器的完整代码。

代码清单4.1　相册列表服务器（load_albums.js）

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

function load_album_list(callback) {
    fs.readdir(
        "albums",
        function (err, files) {
            if (err) {
                callback(err);
                return;
            }
            callback(null, files);
        }
    );
}

function handle_incoming_request(req, res) {
    console.log("INCOMING REQUEST: " + req.method + " " + req.url);
    load_album_list(function (err, albums) {
      if (err) {
          res.writeHead(503, {"Content-Type": "application/json"});
          res.end(JSON.stringify(err) + "\n");
          return;
      }

      var out = { error: null,
                  data: { albums: albums }};
      res.writeHead(200, {"Content-Type": "application/jsor"});
      res.end(JSON.stringify(out) + "\n");
   });
}

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

在代码清单4.1中，fs.readdir执行完毕后，我们会检查执行结果。如果产生错误，则会返回错误给调用者（在handle_incoming_request函数中，作为参数传入load_album_list函数的函数）；否则，我们会发送文件夹（相册）列表给调用者，结

果包含了null以表明没有产生错误。

代码清单还加了一些新的错误处理代码到handle_incoming_request函数中：如果fs.readdir函数报告已经发生一些错误，我们也会让调用者知晓这个情况，所以服务器还是能够返回一些JSON数据和HTTP响应码503，用来表明发生了意外情况。JSON服务器需要返回尽可能多的信息给它们的客户端，用来帮助客户端判断问题是由客户端本身产生还是因为服务器内部出现了问题。

如果要测试程序，请确保将要运行脚本的文件夹中包含albums/子文件夹，并且其中包含相册子文件夹。要运行服务器，可以再次运行脚本：

```
node load_albums.js
```

然后运行以下命令来获取结果：

```
curl -X GET http://localhost:8080/
```

curl命令返回的结果应该和下面类似：

```
{"error":null,"data":{"albums":["australia2010","italy2012","japan2010"]}}
```

4.2 Node模式：异步循环

如果在albums/文件夹中创建文本文件info.txt，那么会发生什么呢？相册列表服务器会返回什么结果？我们很可能看到如下结果：

```
{"error":null,"data":{"albums":["australia2010","info.txt","italy2012","japan2010"]}}
```

对于这个程序，我们真正需要的是检查fs.readdir的结果并返回所有文件夹而不是文件。为了这个目标，可以使用fs.stat函数，它传入一个对象，可以检测该对象是否是一个文件夹。

所以，重写load_album_list函数并遍历fs.readdir的结果，判断它们是否是文件夹：

```
function load_album_list(callback) {
    fs.readdir(
        "albums",
        function (err, files) {
            if (err) {
                callback(err);
                return;
            }

            var only_dirs = [];
            for (var i = 0; i < files.length; i++) {
                fs.stat(
                    "albums/" + files[i],
                    function(err, stats) {
                        if (stats.isDirectory()) {
                            only_dirs.push(files[i]);
                        }
                    }

                );
            }

            callback(null, only_dirs);
        }
    );
}
```

保持程序其他部分不变，然后运行curl命令。它总会返回如下结果：

```
{"error":null,"data":{"albums":[]}}
```

服务器崩溃了！发生了什么？

问题出在新添加的for循环代码上，因为大多数循环和异步回调不能兼容。之前的代码是这样写的：

- 创建一个only_dirs数组来缓存响应。
- 对于文件数组的每一项，调用非阻塞函数fs.stat并将其传入给定的函数，测试这个文件是否是个目录。
- 当所有的非阻塞函数都开始后，退出for循环并调用callback参数。因为Node.js是单线程运行的，所以所有的fs.stat函数都没有机会执行及调用回调函数，最后导致only_dirs的值一直是null，并将这个值传给提供的回调函数。实际上，当fs.stat的回调函数最终被执行时，已经无人在乎了。

为了解决这个问题，必须使用递归。我们可以快速创建一个具有以下格式的新函数，然后立刻调用它：

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

实际上我们可以做到一步到位，即使用一个命名匿名函数，这样就不会因为函数名而弄乱函数的作用域了：

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

因此，要想重写循环代码来测试fs.readdir的结果是否为文件夹，可以编写如下代码：

```
function load_album_list(callback) {
    fs.readdir(
        "albums",
        function (err, files) {
            if (err) {
                callback(err);
                return;
            }

            var only_dirs = [];
            (function iterator(index) {
                if (index == files.length) {
                    callback(null, only_dirs);
                    return;
                }
                fs.stat(
                    "albums/" + files[index],
                    function (err, stats) {
                        if (err) {
                            callback(err);
                            return;
                        }
                        if (stats.isDirectory()) {
                            only_dirs.push(files[index]);
                        }
                        iterator(index + 1)
                    }
                );
            })(0);
        }
    );
}
```

保存最新版本的JSON服务器代码，并运行curl命令，现在应该能够看到只包含相册文件夹而不包含文件的结果。

4.3 小戏法：处理更多的请求

目前相册JSON服务器只能响应一种请求：针对相册列表的请求。实际上，服务器并不在意如何调用这个请求，因此总是返回相同的结果。

我们可以扩展这个服务器的功能，允许处理以下两种请求：

1）可用相册列表——调用/albums.json请求。

2）相册中的照片列表——调用/albums/album_name.json请求。

为请求添加.json后缀，强调当前编写的JSON服务器只用于这类请求。

新版的支持这两种请求的handle_incoming_request函数代码如下：

```
function handle_incoming_request(req, res) {
    console.log("INCOMING REQUEST: " + req.method + " " + req.url);
    if (req.url == '/albums.json') {
        handle_list_albums(req, res);
    } else if (req.url.substr(0, 7) == '/albums'
               && req.url.substr(req.url.length - 5) == '.json') {
        handle_get_album(req, res);
    } else {
        send_failure(res, 404, invalid_resource());
    }
}
```

上面代码中两个if语句块都被加粗显示，它们会检测传入的请求对象的url属性。如果是简单的/albums.json请求，那么可以和之前一样处理。如果是/albums/something.json请求，则可以假设这是处理某个相册的内容列表的请求，并对它进行相应处理。

生成并返回相册列表的代码已经迁移到叫做handle_list_albums的函数，而获取某个独立相册内容的代码同样被组织到两个函数中，分别叫做handle_get_album函数和load_album函数。代码清单4.2包含了服务器的所有代码。

对于新版的代码，可以稍微修改JSON服务器的输出：所有的返回都是对象，而不仅仅是一个字符串数组。当在本书后面章节开始生成UI来匹配JSON响应时，这会帮助到我们。在代码清单4.2中，我使用斜体代码来标识这种变化。

尽管我尝试避免在本书后面包含冗长、乏味的代码，但是服务器代码的首个版本还是值得完整的浏览，因为后面我们做的大多数事情都是建立在当前这些代码之上。

代码清单4.2　处理多种请求类型

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

function load_album_list(callback) {
    // we will just assume that any directory in our 'albums'
    // subfolder is an album.
    fs.readdir(
        "albums",
        function (err, files) {
            if (err) {
                callback(make_error("file_error",  JSON.stringify(err)));
                return;
            }

            var only_dirs = [];
            (function iterator(index) {
                if (index == files.length) {
                    callback(null, only_dirs);
                    return;
                }
```

```
                fs.stat(
                    "albums/" + files[index],
                    function (err, stats) {
                        if (err) {
                            callback(make_error("file_error",
                                                JSON.stringify(err)));
                            return;
                        }
                        if (stats.isDirectory()) {
                            var obj = { name: files[index] };
                            only_dirs.push(obj);
                        }
                        iterator(index + 1)
                    }
                );
            })(0);
        }
    );
}

function load_album(album_name, callback) {
    // we will just assume that any directory in our 'albums'
    // subfolder is an album.
    fs.readdir(
        "albums/" + album_name,
        function (err, files) {
            if (err) {
                if (err.code == "ENOENT") {
                    callback(no_such_album());
                } else {
                    callback(make_error("file_error",
                                        JSON.stringify(err)));
                }
                return;
            }

            var only_files = [];
            var path = "albums/" + album_name + "/";

            (function iterator(index) {
                if (index == files.length) {
                    var obj = { short_name: album_name,
                                photos: only_files };
                    callback(null, obj);
                    return;
                }

                fs.stat(
                    path + files[index],
                    function (err, stats) {
```

```
                        if (err) {
                            callback(make_error("file_error",
                                                JSON.stringify(err)));
                            return;
                        }
                        if (stats.isFile()) {
                            var obj = { filename: files[index],
                                        desc: files[index] };
                            only_files.push(obj);
                        }
                        iterator(index + 1)
                    }
                );
            })(0);
        }
    );
}

function handle_incoming_request(req, res) {
    console.log("INCOMING REQUEST: " + req.method + " " + req.url);
    if (req.url == '/albums.json') {
        handle_list_albums(req, res);
    } else if (req.url.substr(0, 7) == '/albums'
               && req.url.substr(req.url.length - 5) == '.json') {
        handle_get_album(req, res);
    } else {
        send_failure(res, 404, invalid_resource());
    }
}

function handle_list_albums(req, res) {
    load_album_list(function (err, albums) {
        if (err) {
            send_failure(res, 500, err);
            return;
        }

        send_success(res, { albums: albums });
    });
}

function handle_get_album(req, res) {
    // format of request is /albums/album_name.json
    var album_name = req.url.substr(7, req.url.length - 12);
    load_album(
        album_name,
        function (err, album_contents) {
            if (err && err.error == "no_such_album") {
                send_failure(res, 404, err);
            } else if (err) {
                send_failure(res, 500, err);

            } else {
                send_success(res, { album_data: album_contents });
            }
        }
    );
}

function make_error(err, msg) {
    var e = new Error(msg);
    e.code = err;
    return e;
}
function send_success(res, data) {
    res.writeHead(200, {"Content-Type": "application/json"});
    var output = { error: null, data: data };
    res.end(JSON.stringify(output) + "\n");
}
function send_failure(res, code, err) {
    var code = (err.code) ? err.code : err.name;
    res.writeHead(code, { "Content-Type" : "application/json" });
    res.end(JSON.stringify({ error: code, message: err.message }) + "\n");
}
function invalid_resource() {
    return make_error("invalid_resource",
                      "the requested resource does not exist.");
}
function no_such_album() {
    return make_error("no_such_album",
                      "The specified album does not exist");
}

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