为了避免大量复制代码，我们拆分出很多关于发送最后成功信息或对客户端请求响应的代码。这些代码都存放在send_success以及send_failure函数中，这两个函数确保设置了正确的HTTP响应码并能够返回正确的JSON。

阅读完新版load_album函数后，我们会发现它和load_album_list函数有些类似。它枚举了相册文件夹内的所有项，然后检查并确保它们都是正常的文件，最后返回列表。另外我添加了几行代码，对load_album函数中的fs.readdir做了错误处理。

```
if (err.code == "ENOENT") {    // see text for more info
    callback(no_such_album());
} else {
    callback({ error: "file_error",
               message: JSON.stringify(err) });
}
```

如果fs.readdir失败，基本上都是因为查找不到相册文件夹，这是用户导致的错误：用户指定了一个非法的相册。这时如果想返回一个错误来指出这个事实，可以使用帮助函数no_such_album来完成这项任务。而对于其他大部分错误，则可能是服务器配置导致的错误，可以为这些报错场景返回更加通用的"file_error"。

现在获取的/albums.json的输出结果如下所示：

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

当上传一些图片文件到各个相册文件夹之后，获取的相册内容的输出结果（例如/albums/italy2012.json）则会与下面的类似（这里做了清理）：

```
{
  "error": null,
  "data": {
    "album_data": {
      "short_name": "/italy2012",
      "photos": [
        {
          "filename": "picture_01.jpg",
          "desc": "picture_01.jpg"
        },
        {
          "filename": "picture_02.jpg",
          "desc": "picture_02.jpg"
        },
        {
          "filename": "picture_03.jpg",
          "desc": "picture_03.jpg"
        },
        {
          "filename": "picture_04.jpg",
          "desc": "picture_04.jpg"
        },
        {
          "filename": "picture_05.jpg",
          "desc": "picture_05.jpg"
        }
      ]
    }
  }
}
```

4.4 请求和响应对象的更多细节

现在输入并运行下面的程序：

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

function handle_incoming_request(req, res) {
    console.log("-----------------------------------------------------");
    console.log(req);

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

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

然后，在另外一个终端窗口中运行curl命令：

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

客户端窗口应该会只打印error:null，但在服务器端窗口则打印了大量的额外文本信息，这些信息包含了传入HTTP服务器的请求以及响应对象。

我们已经使用过请求对象的两个属性：method和url。前一个属性标识传入的请求是GET、POST、PUT还是DELETE请求（也可能是HEAD等其他的），而后一个属性则包含了发送到服务器请求的URL。

请求对象是一个Node.js的HTTP模块提供的ServerRequest对象，可以查阅Node文档以了解更多细节。文档除了包含这两个属性，还包含了ServerRequest处理POST数据相关的知识。当然，也可以通过查看headers属性来检测传入的请求头。

如果查看curl程序发送过来的请求头，会看到：

```
{ 'user-agent': 'curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24.0 OpenSSL/0.9.8r
zlib/1.2.5',
  host: 'localhost:8080',
  accept: '*/*' }
```

如果在浏览器中调用JSON服务器，则会看到类似的信息：

```
{ host: 'localhost:8080',
  'user-agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10.8; rv:16.0) Gecko/20100101
Firefox/16.0',
  accept: 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8',
  'accept-language': 'en-US,en;q=0.5',
  'accept-encoding': 'gzip, deflate',
  connection: 'keep-alive' }
```

在响应端，我们也使用过两个方法：writeHead和end。对于传入的每个请求都必须调用一次且只能调用一次end方法。否则，客户端会无法获得响应而只会持续地监听连接以获得更多数据。

当编写自己的响应时，应该花点精力研究下HTTP状态码（请参见“HTTP状态码”）。编写服务器端代码包括考虑要如何与客户端进行通信并给它们发送尽可能多的信息用来帮助客户端理解服务器发送的响应。

HTTP状态码

HTTP规范规定了大量服务器能返回给请求客户端的状态码。我们可以从Wikipedia（http://en.wikipedia.org/wiki/List_of_HTTP_status_codes）上更深入地了解它们。

虽然有大量的状态码，但你会发现在应用中我们只会使用常见的几种：

- 200 OK——一切正常。
- 301 Moved Permanently——请求的URL已被移走，客户端应该重新请求响应中指定的URL。
- 400 Bad Request——客户端请求的格式是无效的，需要修复。
- 401 Unauthorized——客户端没有权限查看请求的资源，它应该先验证请求后再试。
- 403 Forbidden——无论出于何种原因，服务器拒绝处理这个请求。这和401不一样，401中客户端在验证通过后可以重试。
- 404 Not Found——客户端请求的资源不存在。
- 500 Internal Server Error——某些情况发生导致服务器无法处理请求。通常这个错误代表代码已经处于某种不一致的状态或出

现bug，需要开发人员关注。

- 503 Service Unavailable——这表明某种运行时错误，例如内存暂时不足或网络资源出现问题。它和500一样都是致命的错误，但它也表明客户端可以过段时间后再次尝试。

这些状态码都是我们最常用到的，但还是有很多其他情况会在浏览时遇到。如果不确定何时使用，可以看看已有的代码是如何处理的。为特定情况给出正确的响应码可能会有很多讨论，但通常我们都能使用正确的状态码。

4.5 提高灵活性：GET参数

当开始往相册中添加大量照片时，应用需要在一个“页面”中高效地展示大量照片，所以我们需要为应用提供分页功能，而客户端需要能够告诉我们需要多少张照片以及是哪一页的照片，就像这样：

```
curl -X GET 'http://localhost:8080/albums/italy2012.json?page=1&page_size=20'
```

如果你对这些术语还不熟悉，那这里就稍微解释下，上面代码中的URL加粗部分是查询字符串，通常作为请求的GET参数。如果我们在先前版本的程序上运行这个curl命令，会发现它一点用处都没有。如果把下面的代码添加到handle_incoming_request函数的开头部分，就会知道原因：

```
console.log(req.url);
```

现在URL显示如下：

```
/albums/italy2012.json?page=1&page_size=20
```

该代码会在字符串末尾而不是在中间查找.json，如果要修复代码来处理分页，需要做三件事情：

1）修改handle_incoming_request函数，让它能够正确解析URL。

2）解析查询字符串，获得page和page_size对应的值。

3）修改load_album函数以支持这些参数。

幸运的是，我们可以一举把前两件事情完成。当添加Node内置的url模块后，可以使用url.parse函数来提取核心的URL路径名以及查询参数。url.parse函数可以更进一步给函数添加第二个参数——true，这个参数告诉url.parse函数解析查询字符串，并生成包含GET参数的对象。如果我们使用url.parse解析前面的URL并打印结果，会看到如下结果：

```
{ search: '?page=1&page_size=20',
  query: { page: '1', page_size: '20' },
  pathname: '/albums/italy2012.json',
  path: '/albums/italy2012.json?page=1&page_size=20',
  href: '/albums/italy2012.json?page=1&page_size=20' }
```

现在可以修改handle_incoming_request函数来解析URL，并将parsed_url存储到请求对象中。函数现在修改如下：

```
function handle_incoming_request(req, res) {

    req.parsed_url = url.parse(req.url, true);
    var core_url = req.parsed_url.pathname;


    // test this fixed url to see what they're asking for
    if (core_url == '/albums.json') {
        handle_list_albums(req, res);
    } else if (core_url.substr(0, 7) == '/albums'
               && core_url.substr(core_url.length - 5) == '.json') {
        handle_get_album(req, res);
    } else {
        send_failure(res, 404, invalid_resource());
    }
}
```

最后，我们需要修改handle_get_album函数来查找页面和page_num查询参数。当没有提供传入值或者值不合法时，可以为查询参数设置合理的默认值（服务器应该总是认为传入的值非常危险，需要对其进行仔细的检查）。

```
function handle_get_album(req, res) {
    // get the GET params
    var getp = req.parsed_url.query;
    var page_num = getp.page ? getp.page : 0;
    var page_size = getp.page_size ? getp.page_size : 1000;

    if (isNaN(parseInt(page_num))) page_num = 0;
    if (isNaN(parseInt(page_size))) page_size = 1000;


    // format of request is /albums/album_name.json
```

```
    var core_url = req.parsed_url.pathname;

    var album_name = core_url.substr(7, core_url.length - 12);
    load_album(
        album_name,
        page_num,
        page_size,
        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_photos: album_contents });
            }
        }
    );
}
```

最后，修改load_album函数。当一切都顺利时，它会提取only_files数组的子数组：

```
function load_album(album_name, page, page_size, callback) {
    fs.readdir(
        "albums/" + album_name,
        function (err, files) {
            if (err) {
                if (err.code == "ENOENT") {
                    callback(nc_such_album());
                } else {
                    callback({ error: "file_error",
                               message: JSON.stringify(err) });
                }
                return;
            }

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

            (function iterator(index) {
                if (index == files.length) {
                    var ps;
                    // slice fails gracefully if params are out of range
                    ps = only_files.splice(page * page_size, page_size);
                    var obj = { short_name: album_name,
                                photos: ps };
                    callback(null, obj);
                    return;
                }

                fs.stat(
                    path + files[index],

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

4.6 修改内容：POST数据

目前，本章已经大体介绍了如何从JSON服务器中获取需要的东西，或许你已经开始想要给服务器发送数据、创建新的东西或者是修改已有的东西。通常这需要通过HTTP POST数据来完成，我们可以通过许多不同的格式来发送数据。为了能够使用curl客户端发送数据，我们需要做点事情：

1）设置HTTP方法参数为POST（或者PUT）。

2）为传入的数据设置Content-Type。

3）开始发送数据。

我们可以轻松地使用curl完成这些任务。首先，需要简单地修改方法的名字；其次，使用-H参数指定curl中的HTTP请求头；最后一步有许多方法可以实现，但在这里，我们使用-d参数将JSON写为字符串。

现在为服务器添加新功能，从而允许我们重命名相册。确保URL的格式如下所示，并指定它必须是一个POST请求：

```
http://localhost:8080/albums/albumname/rename.json
```

因此，重命名相册的curl命令看起来如下所示：

```
curl -s -X POST -H "Content-Type: application/json" \
     -d '{ "album_name" : "new album name" }' \
     http://localhost:8080/albums/old_album_name/rename.json
```

修改handle_incoming_request函数以接收新的请求类型，这很简单：

```
function handle_incoming_request(req, res) {

    // parse the query params into an object and get the path
    // without them. (2nd param true = parse the params).
```

```
    req.parsed_url = url.parse(req.url, true);
    var core_url = req.parsed_url.pathname;

    // test this fixed url to see what they're asking for
    if (core_url == '/albums.json' && req.method.toLowerCase() == 'get') {
        handle_list_albums(req, res);
    } else if (core_url.substr(core_url.length - 12)  == '/rename.json'
               && req.method.toLowerCase() == 'post') {
        handle_rename_album(req, res);
    } else if (core_url.substr(0, 7) == '/albums'
               && core_url.substr(core_url.length - 5) == '.json'
               && req.method.toLowerCase() == 'get') {
        handle_get_album(req, res);
    } else {
        send_failure(res, 404, invalid_resource());
    }
}
```

注意，我们必须将处理重命名请求的代码放置在加载相册请求的代码前面；否则，代码可能会认为它是一个叫做rename的相册，从而执行handle_get_album函数。

4.6.1 接收JSON POST数据

程序为了获取POST数据，会使用一种叫做数据流的Node特性。当使用Node的异步非阻塞特性时，数据流是传输大量数据的最佳方式。这会在第6章详细介绍，而现在我们只需知道使用数据流的主要方式：

```
.on(event_name, function (parm) { ... });
```

现在尤其需要关注两个事件：readable和end事件。数据流实际上是来自于http模块的ServerRequest对象（继承自Stream类；ServerResponse也一样），我们可以通过以下方式监听这两个事件：

```
var json_body = '';
req.on(
    'readable',
    function () {
        var d = req.read();
        if (d) {
            if (typeof d == 'string') {
                json_body += d;
            } else if (typeof d == 'object' && d instanceof Buffer) {
                json_body += d.toString('utf8');
            }
        }
    }
);

req.on(

    'end',
    function () {
        // did we get a valid body?
        if (json_body) {
            try {
                var body = JSON.parse(json_body);
                // use it and then call the callback!
                callback(null, ...);
            } catch (e) {
                callback({ error: "invalid_json",
                           message: "The body is not valid JSON" });
        } else {
            callback({ error: "no_body",
                       message: "We did not receive any JSON" });
            }
        }
    }
);
```

对于传入请求的主体中的每一块（组块）数据，传入on('readable',...)的处理程序函数都会被调用。上面的代码中，我们首先通过read方法读取来自数据流的数据并将这些传入的数据添加到json_body变量的后面；然后，当监听到end事件，会得到这些结果字符串，并尝试去解析它。当给定的字符串不是一个合法的JSON时，JSON.parse会抛出一个错误，所以必须用try/catch语句块封装这些代码。

处理重命名请求的函数代码如下所示：

```
function handle_rename_album(req, res) {

    // 1. Get the album name from the URL
    var core_url = req.parsed_url.pathname;
    var parts = core_url.split('/');
    if (parts.length != 4) {
        send_failure(res, 404, invalid_resource(core_url));
        return;
    }

    var album_name = parts[2];

    // 2. get the POST data for the request. this will have the JSON
    // for the new name for the album.
    var json_body = '';
    req.on(
        'readable',
        function () {
            var d = req.read();
            if (d) {
                if (typeof d == 'string') {
                    json_body += d;
                } else if (typeof d == 'object' && d instanceof Buffer) {

                    json_body += d.toString('utf8');
                }
            }
        }
    );
    // 3. when we have all the post data, make sure we have valid
    //    data and then try to do the rename.
    req.on(
        'end',
        function () {
            // did we get a body?
            if (json_body) {
                try {
                    var album_data = JSON.parse(json_body);
                    if (!album_data.album_name) {
                       send_failure(res, 403, missing_data('album_name'));
                       return;
                    }
                } catch (e) {
                    // got a body, but not valid json
                    send_failure(res, 403, bad_json());
                    return;
                }

                // 4. Perform rename!
                do_rename(
                    album_name,             // old
                    album_data.album_name, // new
                    function (err, results) {
                        if (err && err.code == "ENOENT") {
                            send_failure(res, 403, no_such_album());
                            return;
                        } else if (err) {
                            send_failure(res, 500, file_error(err));
                            return;
                        }
                        send_success(res, null);
                    }
                );
            } else { // didn't get a body
                send_failure(res, 403, bad_json());
                res.end();
            }
        }
    );
}
```

更新后的服务器的完整代码可以处理三种请求，将其保存为post_data.js放在GitHub仓库的Chapter4文件夹下。注意，低于0.10版本的node运行这个程序会出错。我会在第6章详细解释node都做了哪些修改。

4.6.2 接收表单POST数据

虽然现在的应用会尽量不去使用这种方式，但是在Web应用中，还是有大量的数据通过<form>元素提交到服务器，例如：

```
<form name='simple' method='post' action='http://localhost:8080'>
  Name: <input name='name' type='text' size='10'/><br/>
  Age: <input name='age' type='text' size='5'/><br/>
  <input type='submit' value="Send"/>
</form>
```

如果写一个小的服务器程序，如前一节使用readable和end事件来取得POST数据，表单生成的数据打印出来会是：

```
name=marky+mark&age=23
```

然而，实际上我们需要的与之前通过JSON获取的数据很相似：将发送给我们的数据整理成JavaScript对象。为了实现这个目的，可以使用另外一个Node.js内置的模块querystring，特别是这个模块的解析函数parse，如下所示：

```
var POST_data = qs.parse(body);
```

结果和期望的一样，如下所示：

```
{ name: 'marky mark++', age: '23' }
```

接收表单并打印其中内容，该服务器的完整代码清单如下所示：

```
var http = require('http'), qs = require('querystring');

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

    req.on(
        'end',
        function () {
            if (req.method.toLowerCase() == 'post') {
                var POST_data = qs.parse(body);
                console.log(POST_data);

             }
             res.writeHead(200, { "Content-Type" : "application/json" });
             res.end(JSON.stringify( { error: null }) + "\n");
         }
     );

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

但是在第6章的开头部分，当了解到如何使用Node的express Web应用框架之后，我们会发现这些功能早就为我们准备好了。

4.7 小结

本章介绍了许多新知识。我们编写了一个简单的JSON服务器作为我们的第一个Web应用。我喜欢这种方式的主要原因是：

- 它让我们聚焦于服务器，以及能够熟练使用Node.js所需要的核心概念。
- 能够确保服务器的API拥有良好的架构和效率。
- 一个好的、轻量的应用服务器应该是指运行它的计算机能够无碍地处理请求。当我们在后面添加HTML UI以及客户端特性时，可以尝试让客户端做尽可能多的工作。

现在，我们不仅拥有一个基本的服务器，而且看到如何根据不同的请求以及查询参数修改响应输出。我们也看到了如何发送新数据或者通过提交POST数据来修改已经存在的数据。虽然部分程序看起来非常冗长和复杂，但现在我们是使用最基本的模块来验证Node的核心原理。很快我们会开始将这些代码替换成更实用的、功能更丰富的模块。

但首先，让我们稍事休息并深入了解一下模块——更多的是如何使用模块并自己编写模块。

第5章　模块化

到目前为止，我们编写的Node.js服务器和脚本已经通过模块的形式使用了一些外部提供的功能。在本章中，我会解释这些模块是如何工作的，并告诉你如何自己开发模块。除了Node为我们提供了功能强大并且实用的模块之外，还有一个大型社区在不断地开发各种模块，我们可以在程序中好好利用这些模块。而实际上，我们甚至可以自己编写模块来实现某些功能。

Node的一个重要特性就是我们无需真正地辨别模块到底是我们自己编写的还是从外部仓库中获取的，比如本章中通过Node包管理器（Node Package Manager，npm）看到的那些模块。当在Node中编写类和函数时，确保它们大致拥有相同的格式——或许还有一些说明和文档——这与从互联网上下载并使用的模块一模一样。实际上，通常只需要一个额外的JSON文件以及一两行代码，别人就能获取并使用我们的代码！

Node附带了许多内置模块，这些模块都被打包在系统的node可执行文件中。如果从node.js网站上下载Node源代码，就可以查看它们的源文件，它们都在lib/子文件夹下。

5.1 编写简单模块

从更高层面上讲，模块是Node.js对常用功能进行分组的方式。例如，如果我们有一个为特定的数据库服务器工作的函数库或类库，那么将代码提交到模块中并将其打包使用会非常有意义。

虽然模块不会太简单，但实际上Node.js中的每个文件都是一个模块。我们可以将多个文件、单元测试、文档和其他支持文件放在文件夹中，并打包成复杂的模块。还可以通过模块中的单个JavaScript文件来使用它们（见本章后续章节）。

如果要自己编写一个模块，该模块暴露或者提供一个叫做hello_world的函数，可以编写如下代码，并保存为mymodule.js：

```
exports.hello_world = function () {
    console.log("Hello World");
}
```

exports对象是一个特殊的对象，在每个我们创建的文件中由Node模块系统创建，当引入这个模块时，会作为require函数的值返回。它被封装在每个模块的module对象中，用来暴露函数、变量或者类。在这个简单的例子中，模块在exports对象上暴露了一个函数，要使用它，可以编写以下代码，并保存到modtest.js文件中：

```
var mm = require ('./mymodule');
mm.hello_world();
```

运行node modtest.js，Node会打印出预期的结果"Hello World"。我们可以通过exports对象暴露任何我们想要暴露的函数和类。例如：

```
function Greeter (lang) {
    this.language = lang;
    this.greet() = function () {
        switch (this.language) {
          case "en": return "Hello!";
          case "de": return "Hallo!";
          case "jp": return "こんにちは!";
          default: return "No speaka that language";
        }
    }
}

exports.hello_world = function () {
    console.log("Hello World");
}

exports.goodbye = function () {
    console.log("Bye bye!");
}

exports.create_greeter = function (lang) {
    return new Greeter(lang);
}
```

每个模块的module变量会包含许多信息，例如当前模块的文件名、拥有的子模块和对应的父模块等。

模块和对象

当频繁地从模块中返回对象时，我们会发现有两种适用的核心模式。

工厂模式

前面的简单示例中包含了一个叫做Greeter的类。为了获取Greeter对象的实例，需要调用创建函数（或者是工厂函数）来创建并返回这个类的实例。基本模式如下：

```
function ABC (parms) {
    this.varA = ...;
    this.varB = ...;
    this.functionA = function () {
        ...
    }
}

exports.create_ABC = function (parms) {
    return new ABC(parms);
}
```

这种模式的优点是模块可以通过exports对象暴露其他的函数和类。

构造函数模式

另外一种让编写的模块暴露类的方式是完全把模块的exports对象替换成想让别人使用的类：

```
function ABC () {
    this.varA = 10;
    this.varB = 20;
    this.functionA = function (var1, var2) {
        console.log(var1 + " " + var2);
    }
}

module.exports = ABC;
```

为了使用该模块，需要修改代码，如下所示：

```
var ABCClass = require('./conmod2');
var obj = new ABCClass();
obj.functionA(1, 2);
```

因此，模块真正唯一暴露的是一个类的构造函数。这种方式非常不错，并且具有面向对象编程的风格，但它也有个缺点，就是不能让模块暴露更多的东西。因此，在Node中使用这种方式会觉得有些尴尬。我在这里将其展示出来，是为了当看到这种模式时能够清楚它到

底是什么，但我们几乎不会在本书中或者项目中使用到它——通常会使用工厂模式。

5.2 npm：Node包管理器

除了编写自己的模块和使用Node.js提供的模块，我们还会频繁地使用Node社区其他人编写并发布到互联网上的代码。现今最常用的做法是使用npm（Node包管理器）。npm会同node安装程序一起被安装（请参见第1章），可以使用命令行并输入npm help来验证npm是否成功安装并正常运行。

要通过npm安装模块，需要使用npm install命令。这里只需提供想要安装的模块包名称。许多npm模块将源代码托管在github.com上，所以通常它们只会告诉你需要的模块名，例如：

```
Kimidori:pl marcw$ npm install mysql
npm http GET https://registry.npmjs.org/mysql
npm http 200 https://registry.npmjs.org/mysql
npm http GET https://registry.npmjs.org/mysql/-/mysql-2.0.0-alpha4.tgz
npm http 200 https://registry.npmjs.org/mysql/-/mysql-2.0.0-alpha4.tgz
npm http GET https://registry.npmjs.org/require-all/0.0.3
npm http 200 https://registry.npmjs.org/require-all/0.0.3
npm http GET https://registry.npmjs.org/require-all/-/require-all-0.0.3.tgz
npm http 200 https://registry.npmjs.org/require-all/-/require-all-0.0.3.tgz
mysql@2.0.0-alpha4 node_modules/mysql
└── require-all@0.0.3
```

如果不确定想要安装的包名，可以使用npm search命令，如下所示：

```
npm search sql
```

该命令将打印出所有符合条件的模块的名字和描述。

npm将模块包安装到项目的node_modules/子目录下。如果一个模块包本身包含依赖，那么这些依赖都会被安装到这个模块所在文件夹下的node_modules/子目录下。

```
+ project/
    + node_modules/
        module1
        module2
            + node_modules
                dependency1
    main.js
```

如果想查看某个项目当前使用的所有模块清单，可以使用npm ls命令：

```
Kimidori:handlers marcw$ npm ls
/Users/marcw/src/scratch/project
├── async@0.1.22
├─┬ bcrypt@0.7.3
│ └── bindings@1.0.0
└─┬ mysql@2.0.0-alpha4
  └── require-all@0.0.3
```

要想更新一个已经安装的包到新版本，可以使用npm update命令。如果指定了一个包名字，那么只会更新这个包。如果没有指定包名字，这个命令会更新所有包到最新版本：

```
Kimidori:p1 marcw$ npm update mysql
npm http GET https://registry.npmjs.org/mysql
npm http 304 https://registry.npmjs.org/mysql
Kimidori:p1 marcw$ npm update
npm http GET https://registry.npmjs.org/bcrypt
npm http GET https://registry.npmjs.org/async
npm http GET https://registry.npmjs.org/mysql
npm http 304 https://registry.npmjs.org/mysql
npm http 304 https://registry.npmjs.org/bcrypt
npm http 304 https://registry.npmjs.org/async

npm http GET https://registry.npmjs.org/require-all/0.0.3
npm http GET https://registry.npmjs.org/bindings/1.0.0
npm http 304 https://registry.npmjs.org/require-all/0.0.3
npm http 304 https://registry.npmjs.org/bindings/1.0.0
```

5.3 使用模块

正如我们之前看到的，要在编写的Node文件中引入模块，需要使用require函数。为了能够引用模块里的函数和（或）类，需要将结果（被加载的模块的exports对象）赋值给一个变量：

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

引入的模块对于引入它们的模块是私有的，所以，如果a.js加载http模块，那么b.js是无法引用这个模块的，除非b.js自己也加载http模块。

5.3.1 查找模块

Node.js使用一组相当简单的规则来查找require函数请求的模块：

1）如果请求的是内置模块——例如http或者fs——Node会直接使用这些模块。

2）如果require函数的模块名以路径符开始（如./、../或者/），那么Node会在指定的目录中查找模块并尝试去加载它。如果没有在模块名中指定.js扩展名，Node会首先查找基于同名文件夹的模块。如果没有找到，它会添加扩展名.js、.json和.node，并依次尝试加载这些类型的模块（带有.node扩展名的模块会被编译成附加模块）。

3）如果模块名开始时没有路径符，Node会在当前文件夹的node_modules/子文件夹下查找模块。如果找到，则加载该模块；否则，Node会以自己的方式在当前位置的路径树下搜寻node_modules/文件夹。如果依然失败，它会在一些默认地址下进行搜寻，例如/usr/lib和/usr/local/lib文件夹。

4）如果在以上任何一个位置都没有找到模块，则抛出错误。

5.3.2 模块缓存

当模块从指定的文件或者目录上加载之后，Node.js会将它缓存。之后所有的require调用将会从相同的地址加载相同的模块——而这些模块已经初始化或者做过其他工作。这相当有趣，因为有时候两个文件中都尝试加载一个指定的模块，但得到的可能并不是同一个！考虑下面的项目结构：

```
+ project/
    + node_modules/
        + special_widget/
            + node_modules/
                mail_widget (v2.0.1)

        mail_widget (v1.0.0)
    main.js
    utils.js
```

这个例子中，如果main.js或者utils.js都引入mail_widget，由于Node的搜索规则，它会在项目的node_modules/子目录下发现该模块，最终获取mail_widget的1.0.0版本。但是，如果引入special_widget，而这个模块也希望使用mail_widget，special_widget会获取它自己私有的mail_widget版本，2.0.1版本在它自己的node_modules/文件夹下。

这是Node.js模块系统最强大和神奇的特性之一。许多其他的系统、模块、widget或动态库都集中存储在一个位置，当请求的包本身也需要请求其他不同版本的模块时，版本控制就成了梦魇。而在Node中，可以自由地引入其他不同版本的模块，Node的命名空间和模块规则意味着它们之间不会互相干扰！独立的模块和项目部分内容都可以自由地引入、更新或修改引入的模块，因为它们只能看到自己那部分而不影响系统其他部分。

简而言之，Node.js非常直观，这也许是你有生以来第一次无需坐在那里无休止地咒骂正在使用的包管理系统。

5.3.3 循环

考虑下面的情况：

- a.js引入b.js。
- b.js引入a.js。
- main.js引入a.js。

很显然，我们会发现上面的模块中存在一个循环。当Node探测到一个尚未初始化的返回模块时，它会阻止循环发生。在上面的示例中，会发生下面的事情：

- main.js被加载，运行引入a.js的代码。
- a.js被加载，运行引入b.js的代码。
- b.js被加载，运行引入a.js的代码。
- Node探测到循环并返回一个指向a.js的对象，但不会再执行其他代码——在这一刻，a.js的加载和初始化并没有完成。
- b.js、a.js和main.js都完成初始化（按顺序），然后b.js和a.js的引用都有效和可用。

5.4 编写模块

Node.js中每个文件都是一个含有module和exports对象的模块。但是我们也要清楚，模块也可以很复杂：包含一个目录用来保存模块的内容和一个含有包信息的文件。如果想要编写一系列支持文件，并将模块的功能拆散到多个JavaScript文件中，甚至包含单元测试，可以使用下面这种格式来编写模块。

基本格式如下：

1）创建一个文件夹来存放模块内容。

2）添加名为package.json的文件到文件夹中。文件至少包含当前模块的名字和一个主要的JavaScript文件——用来一开始加载该模块。

3）如果Node没有找到package.json文件或者没有指定主JavaScript文件，那它就会查找index.js（或者是编译后的附加模块index.node）。

5.4.1 创建模块

现在我们可以将在前面章节中为管理照片和相册而编写的代码提取出来并放到一个模块中。这样做就可以分享这些模块，以供将来编写的其他项目使用，并对代码进行隔离，因此也可以编写测试用例，等等。

首先，在源代码目录下（即~/src/scratch或其他运行Node的地方）创建如下目录结构：

```
+ album_mgr/
    + lib/
    + test/
```

在album_mgr文件夹下，创建一个叫做package.json的文件，