```
app.get('/pages/:page_name/:sub_page', serve_page);
```

现在，所有的请求页面路径会解析到正确的路由函数上，同时我们也能很清晰地识别请求的子页面。

现在这个应用的一个优点就是无须手动解析传入的请求或参数，express已经为我们做好这一切了。因此，之前需要引入url模块解析传入的URL，而现在，只要使用路由函数即可。同样，也不需要解析GET查询参数，因为它们已经被解析完成。因此，可以更新一些帮助函数，如下所示：

```
function get_album_name(req) {
    return req.params.album_name;
}

function get_template_name(req) {
    return req.params.template_name;
}
function get_query_params(req) {
    return req.query;
}
function get_page_name(req) {
    return req.params.page_name;
}
```

通过上面的改造，相册应用应该被更新并开始使用express。而功能上，也会和原版本保持一致。

7.3 REST API设计和模块

如果你正打算设计一个JSON服务器，如本书应用的JSON服务器，则需要花一些时间仔细斟酌API以及客户端会如何使用它。在设计API之前多花些时间思考，有助于理解用户如何使用应用并能帮助组织和重构代码，使其能够精确传达你对产品的深刻理解。

7.3.1 API设计

我们会为相册分享JSON服务器开发叫做RESTful JSON的服务器。单词REST是Rep-resentational State Transfer的缩写，表明可以从服务器上请求一个对象。REST API主要包含四种不同的核心操作（这和HTTP请求的方法保持一致性[2]）：

- 创建（PUT）
- 检索（GET）
- 更新（POST）
- 删除（DELETE）

许多人喜欢把这些操作称为CRUD，它包含了API操作对象所需要的一切，而不论这些对象是相册、照片、评论或者是用户。尽管很难设计出一个十全十美的API方案，但我还是想尽我所能设计出一个至少符合我们直觉的目标方案，不会让客户端开发者抓耳挠腮。下面是一些设计RESTful接口时的原则：

- URL有两种基本类型，绝大多数设计出来的URL会是下面两种的变形：
  - 集合——例如，/albums
  - 集合中指定的条目——例如，/albums/italy2012
- 集合的名称应当是名词，最好是复数名词，如albums、

users或photos等。

- PUT/albums.json——HTTP请求主体中包含了新相册所需要的JSON数据。

■ 指定集合中某个特定实例，分别使用GET和POST方法获取或更新对象：

- GET/albums/italy2012.json——请求返回该相册。
- POST/albums/italy2012.json——HTTP请求主体中包含了更新相册所需要的JSON数据。

■ 使用DELETE删除某个对象。

- DELETE/albums/italy2012.json——删除该相册。

■ 如果某个集合来自另一个集合，比如，照片和相册相关联，只需继续使用上面的模式即可：

- GET/albums/italy2012/photos.json——返回该相册中的所有照片。
- PUT/albums/italy2012/photos.json——向该相册中添加一张新照片。
- GET/albums/italy2012/photos/23482938424.json——获取指定ID的照片。

■ 如果需要稍微改变获取数据集合的方式，比如分页或过滤，应该通过添加GET请求参数来实现：

- GET/albums.json?located_in=Europe——只获取地点是欧洲的相册。
- GET/albums/japan2010/photos.json?page=1&page_size=25——获取japan2010相册中第1页的25张照片。

■ 需要对API版本化，以便将来对API进行较大的版本改动时能向后兼容，我们只需简单地更新版本号即可。因此，需要在API URL中添加/v1/前缀。

■ 最后，在所有需要返回JSON数据的URL中添加.json后缀，这样客户端就会知道返回的数据格式。将来，我们也可以添加任何想支持的格式，如.xml等。

谨记以上原则，相册应用的新API就会紧紧围绕相册这个主题。如下所示：

```
/v1/albums.json
/v1/albums/:album_name.json

/pages/:page_name
/templates/:template_name
/content/:filename
```

我们为什么没有将版本号添加到静态内容的URL上？因为它们不是JSON服务器的接口，而是一些为Web浏览器客户端提供静态内容的帮助函数。用户总是会获取最新版本的静态文件，这样就能获取最新版本API的版本号。

使用全新的API设计更新server.js的工作量并不大。但是别忘记，还需要通过新的URL更新/content/下的JavaScript加载器文件！在继续改造之前，请先确认是否能正常运行。

7.3.2 模块

当所有API更新成全新的REST接口之后，整个应用项目变得非常清晰。现在，我们已经了解了相册和页面的具体功能，因此可以将这些功能添加到各自的模块中。

在应用文件夹下创建一个名叫handlers/的文件夹，用来存放最新的模块。在该文件夹下为相册创建albums.js文件，并将四个相册操作函数放入文件中。你可以在Github上阅读该项目下handlers_as_modues/文件夹中的源代码。实际上，我们就是将

handle_get_album和handle_list_albums函数集成到新的帮助函数load_album和load_album_list中，从而形成新的模块。建议你花一分钟阅读一下源代码，看它是如何快速简单地组织代码结构的。

回顾一下前文中帮助我们提取相册名称和查询参数等的函数。现在，可以删除它们并在合适的地方使用req.params和req.query。

同时，还有一些帮助我们简化处理发送成功和失败状态的函数，比如send_success、send_failure和invalid_resource。我们可以将它们放到handlers/目录下的helpers.js模块中，然后在albums.js文件顶部引用即可，内容如代码清单7.1所示。

代码清单7.1　帮助函数（helpers.js）

```
exports.version = '0.1.0';

exports.make_error = function(err, msg) {
    var e = new Error(msg);
    e.code = err;
    return e;
}

exports.send_success = function(res, data) {
    res.writeHead(200, {"Content-Type": "application/json"});
    var output = { error: null, data: data };
    res.end(JSON.stringify(output) + "\n");
}
exports.send_failure = function(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");
}

exports.invalid_resource = function() {
    return make_error("invalid_resource",
                      "the requested resource does not exist.");
}

exports.no_such_album = function() {
    return make_error("no_such_album",
                      "The specified album does not exist");
}
```

现在，如果需要引用album函数，只需在server.js文件的顶部添加require即可：

```
var album_hdlr = require('./handlers/albums.js');
```

然后，在路由处理程序中添加相应的相册函数，如下所示：

```
app.get('/v1/albums.json', album_hdlr.list_all);
app.get('/v1/albums/:album_name.json', album_hdlr.album_by_name);
```

同样，可以将创建页面的功能迁移到handlers/目录下的

pages.js文件中，如代码清单7.2所示。这部分工作实际上只是将原先的serve_page函数移植过来。

代码清单7.2　创建页面（pages.js）

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

exports.version = "0.1.0";

exports.generate = function (req, res) {
    var page = req.params.page_name;

    fs.readFile(
        'basic.html',
        function (err, contents) {
            if (err) {
                send_failure(res, 500, err);
                return;
            }
            contents = contents.toString('utf8');

            // replace page name, and then dump to output.
            contents = contents.replace('{{PAGE_NAME}}', page);
            res.writeHead(200, { "Content-Type": "text/html" });
            res.end(contents);
        }
    );
};
```

可以更新页面的路由处理程序，如下所示：

```
var page_hdlr = require('./handlers/pages.js');
app.get('/pages/:page_name', page_hdlr.generate);
```

所有任务完成之后，server.js文件只剩下处理静态内容的函数，可以查看第7章的Github源代码，具体在handlers_as_modules/项目文件夹下。

现在，这个应用和原先的拥有相同的功能，但是代码却更加模块化，更具可读性。至此，我们应该已经知道如何向应用中添加功能了。

[2]根据HTTP　RFC，一般情况下，POST用于创建资源，而PUT用于修改资源。——译者注

7.4 中间件功能

我曾提到过express是构建在connect这个中间件类库之上的。这些组件链会依次处理每个请求，并会在组件调用next函数之后结束调用。而路由处理程序则是这个组件链的一部分。express和connect还提供了一些其他实用组件，让我们一起看看其中一些有趣的组件。

7.4.1 基本用法

在express和connect中可以通过use方法来使用中间件组件。例如要使用logging中间件，可以写成：

```
var express = require('express');
var app = express();

app.use(express.logger());
/* etc ... */
```

调用express.logger函数后会返回一个函数，它会构成应用中过滤层（filter layering）的一部分。尽管express内置了一些组件，但是它还允许使用其他兼容的中间件组件。因此，我们可以使用connect提供的所有组件（前提是将connect添加到package.json文件的dependencies中）：

```
var express = require('express');
var connect = require('connect');
var app = express();

app.use(connect.compress());
/* etc ... */
```

由于添加和引入connect模块是一件麻烦的事，所以express已经重新暴露了connect中的所有中间件。因此，只需要从express模块中直接引用这些中间件即可，如下所示：

```
var express = require('express');
var app = express();

app.use(express.compress());
/* etc ... */
```

当然，也可以直接从npm或者其他地方下载并安装第三方组件：

```
var express = require('express');
var middle_something = require('middle_something');
var app = express();

app.use(middle_something());
/* etc ... */
```

7.4.2 配置

在特定的配置环境中使用一些中间件非常实用。有些中间件能够修改响应的速率或者输出大小，因此这会对用curl测试服务器造成困扰。因此，可以在express应用中使用configure方法并指定配置名称，方法中提供的函数会在指定的配置环境下被调用。如果不提供名称，则函数会被所有配置环境调用，当然也可以在函数之前使用多个配置名称，用逗号隔开，如下所示：

```
app.configure(function () {
  app.use(express.bodyParser());
});

app.configure('dev', function () {
  app.use(express.logger('dev'));
});
app.configure('production', 'staging', function () {
  app.use(express.logger());
});
```

在configure函数外设置的中间件或者路由适用于所有的配置环境。

要想在某个特定的配置环境下运行应用，可以通过设置NODE_ENV环境变量达到目的。在Mac和UNIX电脑中，绝大部分

shell都支持以下方式运行node应用：

```
NODE_ENV=production node program.js
```

在Windows平台下，只需在命令提示符中设置NODE_ENV环境变量，并运行node应用：

```
set NODE_ENV=production
node program.js
```

7.4.3 中间件执行顺序

中间件组件的分层和顺序非常重要。正如前文所说，express和connect会依次经过这些组件，直到找到可以处理请求的组件。要理解这之间是如何关联的，可以参考下面的简单应用示例：

```
var express = require('express');
var app = express();

app.use(express.logger('dev'))
     // move this to AFTER the next use() and see what happens!
    .use(express.responseTime())
    .use(function(req, res){
        res.end('hello world\n');
    })
    .listen(8080);
```

调用该应用：

```
curl -i localhost:8080/blargh
```

可以得到如下结果：

```
HTTP/1.1 200 OK
X-Powered-By: Express
X-Response-Time: 0ms
Date: Wed, 05 Dec 2012 04:11:43 GMT
Connection: keep-alive
Transfer-Encoding: chunked

hello world
```

现在，如果将responseTime中间件组件移至最后，会发生什

么？如下所示：

```
var express = require('express');
var app = express();

app.use(express.logger('dev'))
     // move this to AFTER the next use() and see what happens!
    .use(function(req, res){
        res.end('hello world\n');
    })
    .use(express.responseTime())
    .listen(8080);
```

我们注意到响应头X-Response-Time已经消失了！

```
HTTP/1.1 200 OK
X-Powered-By: Express
Date: Wed, 05 Dec 2012 04:14:05 GMT
Connection: keep-alive
Transfer-Encoding: chunked

hello world
```

负责打印"Hello World!"的函数会接收该请求，并全权处理。它会使用res.end关闭响应，responseTime中间件甚至都没有机会处理该请求。因此，在添加中间件组件的时候，需要考虑它们之间是如何为应用中的路由协调工作的。

7.4.4 静态文件处理

前文中已经为应用写过一些处理静态文件的Node代码，但其实完全没必要那样做。express（通过connect）提供了static中间件组件，可以为我们完成这一切。

要使用它，只需要将存放静态文件的根目录的路径名赋给中间件创建函数，如下所示：

```
app.use(express.static("/secure/static_site_files"));
```

如果将其放在URL路由函数前面，那么当请求进来时，该静态层就会接收URL，并将其附加到根目录名称后面，然后检测该文件是否存在。例如，如果请求/content/style.css，该中间件就会检

测/secure/static_site_files/content/styles.css是否存在且可读。如果是，则该中间件就会处理该文件，并停止调用下一层；如果不是，则调用next函数。

如果想设置多个静态文件目录，只需添加多个组件即可：

```
app.use(express.static("/secure/core_site_content"));
app.use(express.static("/secure/templates_and_html"));
```

注意，尽管在技术上可以使用Node中预定义变量__dirname（即当前正在执行的Node脚本的路径）处理应用文件夹下的内容，但这是一件非常糟糕的事情，千万不要使用。

static中间件没有提供任何安全机制，因此如果在前文中的相册应用代码中添加上

```
app.use(express.static(__dirname));
```

然后，就可以如预期一样正常访问/content/style.css和/templates/home.html，不会出现任何问题。但问题是，如果用户访问/server.js或者/handlers/albums.js，那么static中间件也同样会处理它们，因为这些文件也在__dirname根目录下。

也许你会问，为什么static中间件组件没有提供任何安全机制，其实这个问题有些不恰当——它根本没有那么复杂。如果想使用该组件，只需要将静态文件放到应用代码树的外部即可。事实上，我曾提到过，甚至可以让Node应用避免处理静态内容，而是选择将静态内容放到CDN上。因此，从一开始就尝试将这些组件与代码树分离的做法吧。

如果使用static中间件重写server.js，它会看起来和下面的代码一样。首先，需要将content/、templates/和albums/文件夹放置到另一个位置，以避免前面提到的安全问题。更新后的应用代码结构如下所示：

```
+ root_folder/
  + static/
    + albums/
    + content/
    + templates/
  + app/
    + handlers/
    + node_modules/
```

server.js文件如代码清单7.3所示。新版server代码的好处就是可以移除原先处理静态内容和模板的路由函数。

代码清单7.3　使用static中间件（server.js）

```
var express = require('express');
var app = express();

var fs = require('fs'),
    album_hdlr = require('./handlers/albums.js'),
    page_hdlr = require('./handlers/pages.js'),
    helpers = require('./handlers/helpers.js');
app.use(express.static(__dirname + "/../static"));

app.get('/v1/albums.json', album_hdlr.list_all);
app.get('/v1/albums/:album_name.json', album_hdlr.album_by_name);
app.get('/pages/:page_name', page_hdlr.generate);
app.get('/pages/:page_name/:sub_page', page_hdlr.generate);

app.get("/", function (req, res) {
    res.redirect("/pages/home");
    res.end();
});

app.get('*', four_oh_four);

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

    res.end(JSON.stringify(helpers.invalid_resource()) + "\n");
}

app.listen(8080);
```

如果你足够细致，也许会注意到为/添加的新路由中，使用ServerResponse.redirect方法将请求重定向到/pages/home，以减少在浏览器端的重复输入。

7.4.5　POST数据、cookie和session

express和connect已经内置了帮助方法，可以解析请求的查询参数并保存到req.query对象中。在第4章中，我们通过数据流手动下载和解析表单POST数据。好消息是，我们不必手动做这一切了——中间件可以为我们实现这个功能，甚至还可以给应用设置

cookie和seesion，分别使用bodyParser、cookieParser和session中间件完成这些功能：

```
app.use(express.bodyParser());
app.use(express.cookieParser());
```

第一个bodyParser中间件会解析所有的请求体数据，并放置到req.body对象中。如果请求的数据是urlencoded类型或者JSON格式，每个数据字段都会保存到req.body中作为对象的字段。cookieParser也是类似的工作原理，但是会把值放到req.cookies中。

要在返回的响应中设置cookie，可以使用ServerResponse对象中的cookie函数，设置名称和值（以及可选参数——过期时间）：

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

var app = express()
    .use(express.logger('dev'))
    .use(express.cookieParser())
    .use(function(req, res){
        res.cookie("pet", "Zimbu the Monkey",
                   { expires: new Date(Date.now() + 86400000) });
        res.end(JSON.stringify(req.query) + "\n");
    })
    .listen(8080);
```

如果想要清空设置过的cookie，只需要使用ServerResponse对象中的clearCookie函数，指定想要移除的cookie的名称即可。

而设置session则会稍微复杂一点，需要提供存储session的空间和加密值所需的密钥：

```
var MemStore = express.session.MemoryStore;
app.use(express.session({ secret: "cat on keyboard",
                          cookie: { maxAge: 1800000 },
                          store: new MemStore()}));
```

在示例中，设置了密钥，然后可以使用connect模块提供的MemoryStore类存储session。这种方法十分方便易用，但是每次重启Node的时候都会重置session。如果期望每次重启服务器之后都能保存住session，可以在npm上找到一些其他的存储方式，如使用外部存储的memcache或者redis（具体可见下面两章内容）。默认情

况下，session cookie不会过期。如果想要添加有效期，可以在参数中添加有效期的毫秒数。这里，代码设置的session有效期为30分钟。

当添加session功能之后，session数据会填充到req.session中。而要想添加新的seesion数据，只需在该对象中添加新的值即可。它会持久化并随着响应返回：

```
var express = require('express');
var MemStore = express.session.MemoryStore;

var app = express()
    .use(express.logger('dev'))
    .use(express.cookieParser())
    .use(express.session({ secret: "cat on keyboard",
                           // see what happens for value of maybe 3000
                           cookie: { maxAge: 1800000 },
                           store: new MemStore()}))
    .use(function (req, res){
        var x = req.session.last_access;  // undefined when cookie not set
        req.session.last_access = new Date();
    res.end("You last asked for this page at: " + x);
})
.listen(8080);
```

文件上传

express的bodyParser中间件组件已经自动添加了connect multipart组件，可以提供文件上传功能。如果使用multipart/form-data格式向应用上传文件，可以在req.files对象中找到请求中包含的所有上传文件：

```
var express = require('express');
var app = express()
    .use(express.logger('dev'))
    .use(express.bodyParser())
    .use(function (req, res){
        if (!req.files || !req.files.album_cover) {
            res.end("Hunh. Did you send a file?");
        } else {
            console.log(req.files);
            res.end("You have asked to set the album cover for "
                    + req.body.albumid
                    + " to '" + req.files.album_cover.name + "'\n");
        }
    })
    .listen(8080);
```

我们仍然可以用curl测试该功能！可使用-F参数（表单数据）在请求中包含上传文件，使用@操作符指定上传的文件名：

```
curl -i -H "Expect:" --form 'album_cover=@oranges.jpg' \
 --form albumid=italy2012 http://localhost:8080
```

还有一个小技巧，当curl期望服务器返回响应100Continue并等待客户端继续发送数据时，可以向curl传递参数-H"Expect:"，通过这种方式，可以测试文件上传功能。

7.4.6 对PUT和DELETE更友好的浏览器支持

PUT（创建对象）和DELETE（删除对象）是REST API中的重要组成部分，客户端网页通过这些方法向服务器发送数据。例如，在jQuery中，要想删除一个评论，需要使用

```
$.ajax({
    url: 'http://server/v1/comments/234932598235.json',
    type: 'DELETE'
}
```

在大部分现代Web浏览器中，它都能正常工作，但在一些老版本的浏览器（IE 10之前的版本）中却不能正常工作。究其原因，是这些浏览器中XmlHttpRequest对象的实现不支持AJAX请求中的PUT和DELETE方法。

一个通用且优雅的解决方案是添加一个真正想用的方法到新的请求头X-HTTP-Method-Override上，然后使用POST上传请求到服务器。jQuery代码如下所示：

```
$.ajax({
   beforeSend: function(xhr) {
       xhr.setRequestHeader('X-HTTP-Method-Override', 'DELETE');
   },
   url: 'http://server/v1/comments/234932598235.json',
   type: 'POST'
}
```

要想在服务器上支持这种解决方案，需要添加methodOverride中间件，如下所示：

```
app.use(express.methodOverride());
```

这样服务器会寻找该请求头，并将POST请求转换成DELETE或者PUT方法。因此，app.delete和app.put路由函数就能正确工作

了。

7.4.7 压缩输出

为了减少带宽成本，很多Web服务器都会在将数据发送到客户端之前，使用gzip或者其他类似的算法压缩输出数据。前提条件是客户端需要在HTTP中使用请求头Accept-Encoding表明能够接收压缩后的数据。如果客户端提供了该请求头且服务器也支持压缩功能，则需要在输出的响应头Content-Encoding中指定合适的算法，然后将压缩数据发送到客户端。

要在应用中使用这个特性，可以使用压缩中间件。它使用起来非常简单，会通过请求头判断客户端是否支持压缩数据来设置响应头，并压缩输出数据，如下所示：

```
var express = require('express');
var app = express();

app.use(express.logger('dev'));
app.use(express.compress());

app.get('/', function(req, res){
    res.send('hello world this should be compressed\n');
});

app.listen(8080);
```

该模块的唯一缺点就是它会影响开发测试（curl只会打印出压缩后的二进制数据，不够直观），因此，一般只会在生产环境下使用该中间件，如下所示：

```
var express = require('express');
var app = express();

app.use(express.logger('dev'));
app.configure('production', function () {
    app.use(express.compress());
});

app.get('/', function(req, res){
    res.send('hello world this should be compressed\n');
});

app.listen(8080);
```

7.4.8 HTTP基本身份验证

如果想在应用中使用HTTP基本身份验证（basic authentication），中间件可以帮助我们实现。尽管其由于过时和不安全而倍遭诟病，但如果基本身份验证与SSL和HTTPS配合起来使用的话，应用将会变得足够健壮和安全。

要使用它，设置basicAuth中间件即可，它提供一个函数用来接收并验证用户名和密码。当验证信息正确时，返回true；反之，则返回false，如下所示：

```
var express = require('express');
var app = express();

app.use(express.basicAuth(auth_user));
app.get('/', function(req, res){
  res.send('secret message that only auth\'d users can see\n');
});

app.listen(8080);

function auth_user(user, pass) {
    if (user == 'marcwan'
        && pass == 'this is how you get ants') {
        return true;
    } else {
        return false;
    }
}
```

如果现在启动并访问服务器，它会在返回数据之前询问用户名和密码。

7.4.9 错误处理

尽管我们可以对每个请求单独进行错误处理，但有时也希望有一个全局的错误处理来解决一些常见问题。要实现这一点，可以在app.use方法中提供一个包含四个参数的函数：

```
app.use(function (err, req, res, next) {
    res.status(500);
    res.end(JSON.stringify(err) + "\n");
});
```

该方法会放置到所有其他中间件和路由函数的后面，从而成为express在应用中调用的最后一个函数。在函数中可以检测具体错误，并决定返回给用户的数据。例如，将err对象或者抛出的Error对象转成适当的JSON格式并作为错误信息返回给用户：

```
var express = require('express');
var app = express();

app.get('/', function(req, res){
    throw new Error("Something bad happened");
    res.send('Probably will never get to this message.\n');
});

app.use(function (err, req, res, next) {
    res.status(500);
    res.end(err + "\n");
});

app.listen(8080);
```

最后，Node.js可以通过process对象在全局应用范围内指定错误处理函数：

```
process.on('uncaughtException', function (err) {
    console.log('Caught exception: ' + err);
});
```

但是，如果使用它进行打印日志或者诊断之外的工作，那就不是一个明智的选择。当该函数被调用时，基本可以断定Node已经不能正常工作，处于不稳定的状态，需要重启。因此，在这些使用场合中，一般会使用该函数打印错误日志和结束node进程：

```
process.on('uncaughtException', function (err) {
    console.log('Caught exception: ' + err);
    process.exit(-1);
});
```

因此，最好在可能出现错误的地方使用try/catch来捕捉错误；而在一些灾难性错误和完全不可预期的场景下，则需关闭进程并重启应用——最好是自动重启。

7.5 小结

在这一大章中，我们了解了许多有趣且实用的知识。本章介绍了express应用开发框架、中间件以及如何使用cookie和session，并将相册应用的代码模块化。使用静态文件服务中间件剥除了大量无用代码，并且在应用中添加文件压缩、用户验证和更好的错误处理机制。

现在，相册应用已经成型，但是还有一些不尽如人意的地方。相册中只有照片，却没有存放更多额外的相册或照片信息。带着这些疑问，接下来我们将会学习数据库相关内容——首先是NoSQL（CouchDB），然后是MySQL——学习如何在Node中使用它们。同时，我们还会看一些缓存解决方案，让这个小应用更加贴切实际。

第8章　数据库I：NoSQL（MongoDB）

到目前为止，我们已经为Web应用打下了坚实的基础，应用完全由express Web应用框架和Mustache模板引擎构建而成，接下来我们会用两个章节介绍如何为应用添加后端数据库。这两章中，我们会学习两种最常用的数据库。首先，在本章中学习目前流行的NoSQL数据库——MongoDB，它能快速简单地直接将JSON数据序列化到数据库中。本章会介绍MongoDB的基本使用方法，并更新相册处理程序，使其可以将相册和照片数据存储到数据库中。

我们之所以选择MongoDB而不是其他流行的NoSQL数据库（尤其是CouchDB），是因为MongoDB极易上手而且提供了超赞的查询功能，而其他数据库则较为复杂。由于本书的目的之一就是教会你使用Node.js，因此我选择了Node.js中最容易搭配使用的MongoDB，这样能让我们更专注于自己的初衷。

如果你偏好于MySQL或者其他关系型数据库，本章也依然值得一读，因为你可以看到如何将相册和照片处理程序从前文中的文件模块迁移到数据库模块中。而在下一章中，我们会讨论关系型数据库，同时也会提供一个MySQL版本的应用代码。

8.1 设置MongoDB

要在我们的Node.js应用中使用MongoDB，需要做两件事：安装数据库服务器以及配合Node使用。

8.1.1 安装MongoDB

要使用MongoDB，首先要从http://mongodb.org上下载二进制文件。大部分平台下的发行版为zip文件格式，我们可以将其解压到任何想运行MongoDB的目录下。

在bin/子文件夹下，可以找到一个名叫mongod的文件，这就是基本数据库服务器守护进程（daemon）。要想在开发环境下运行服务器，只需运行：

```
./mongod --dbpath /path/to/db/dir                    # unix
mongod --dbpath \path\to\db\dir                      # windows
```

我们可以创建文件夹如~/src/mongodbs或者其他类似的文件夹来存储数据库数据。如果想要快速删除数据库并重启数据库服务器，只需按下Ctrl+C中止当前服务器进程并运行

```
rm /path/to/db/dir/* && ./mongod --dbpath /path/to/db/dir     # unix
del \path\to\db\dir\*                                        # windows 1
mongod --dbpath \path\to\db\dir                              # windows 2
```

要想测试是否安装成功，打开另一个终端窗口或命令提示符，运行mongo程序——一个类似于Node REPL的简易解释器：

```
Kimidori:bin marcw$ ./mongo
MongoDB shell version: 2.2.2
connecting to: test
> show dbs
local (empty)
>
```

而如果想要部署到生产环境中，则需要进一步阅读MongoDB文档，学习相关的最佳实践和配置选项，包括主从复制、备份等。

8.1.2 在Node.js中使用MongoDB

在验证MongoDB安装成功之后，需要把它与Node.js桥接到一起。现在Node上最流行的驱动就是10gen提供的官方原生驱动mongodb，10gen就是开发出MongoDB的公司。同时也有专为MongoDB和Node.js准备的关系对象映射（relational-object mapping，ROM）框架mongoose，但是本书中我们只关注简单的驱动，因为它已经提供了我们想要的一切（相册应用的数据需求并不复杂），同时也会让代码降低与数据库的耦合（我们会在下章中看到）。

向package.json文件中添加如下代码来安装mongodb模块：

```
{
  "name": "MongodB-Demo",
  "description": "Demonstrates Using MongoDB in Node.js",
  "version": "0.0.1",
  "private": true,
  "dependencies": {
    "async": "0.1.x",
    "mongodb": "1.2.x"
  }
}
```

然后可以运行npm update来获取最新的驱动。更新完成后，查看node_modules文件夹，可以看到mongodb/子文件夹。

8.2 MongoDB数据结构

我们的应用中非常适合使用MongoDB，因为MongoDB本身就是使用JSON存储数据。当在数据库中创建或者添加数据时，只需要传一个JavaScript对象过去即可——这简直就是为Node量身打造的！

如果你以前使用过关系型数据库，可能会对数据库的一些术语比较熟悉，如数据库、表（table）和行（row）。而在MongoDB中，这些元素分别对应着数据库、集合（collection）和文档（document）。数据库可以包含多个集合，而每个集合由JSON文档表示。

MongoDB中的所有对象都有一个唯一标识符_id。原则上，只要是唯一值，_id可以是任何类型。如果不手动提供该值，则MongoDB会自动为我们创建一个：它就是ObjectID类的一个实例。如果打印出来，它会返回一个24位的十六进制字符串，比如50da 80c0 4d40 3ebd a700 0012。

8.2.1 全是JavaScript的世界

我们需要为相册应用创建两个文档类型：相册和照片。我们不需要使用MongoDB自动生成的_id值，因为每个文档都已经有唯一的标识信息了：对于相册而言，相册名称（album_name）是唯一的；而对于照片而言，相册名称（album_name）和照片名称（filename）的组合也是唯一的。

因此，存储的相册数据如下所示：

```
{ _id: "italy2012",
  name:"italy2012",
  title:"Spring Festival in Italy",
  date:"2012/02/15",
  description:"I went to Italy for Spring Festival." }
```

照片文档则类似于：

```
{ _id: "italy2012_picture_01.jpg",
  filename: "picture_01.jpg",
  albumid: "italy2012",
  description: "ZOMGZ it's Rome!",
  date: "2012/02/15 16:20:40" }
```

当尝试向集合中添加重复的_id值时，会导致MongoDB报错。通过这种特性，可以保证相册是唯一的，同时每个相册中的图片文件也是唯一的。

8.2.2 数据类型

大多数情况下，在JavaScript中使用MongoDB非常舒适自然、简单直接。但是，还是会有一些特殊的使用场景会引起麻烦，值得我们注意。

回到第2章，JavaScript中所有数字都是64位双精度浮点数。它提供了53位整数精度，但是经常会有使用64位整数值的需求。当降低精度和准确度不可接受的时候，可以使用mongodb驱动为Node准备的Long类。它的构造器会接收一个字符串值，可供我们进行64位整数值的操作。它还为整数值提供了一些方法，如toString、compare和add/subtract/multiply，来模拟整数值的常用操作。

Java Script还提供了Binary类，可以让我们在文档中存储二进制数据。我们可以向它的构造器中传递字符串或者一个Buffer对象实例，这些数据会以二进制数据格式存储。当重新加载文档的时候，返回的值会包含一些额外的元数据，用来描述Mongo是如何在集合中存储的。

最后，我想介绍下Timestamp类，可以用来将时间日期存储到数据库文档中，只需要在写数据的时候添加即可！如下所示：

```
{ _id: "unique_identifier1234",
  when: new TimeStamp(),
  what: "Good News Everyone!" };
```

要想查看mongodb模块提供的所有数据类型帮助列表，可以查看github.com/mongodb/node-mongodb-native上提供的说明文档或者阅读node_modules/mongodb/node_modules/bson/lib/bson/下的源代码。

8.3 理解基本操作

至此，我们可以使用MongoDB和Node.js处理事情了。在每个文件的顶部使用mongodb模块，如下所示：

```
var Db = require('mongodb').Db,
    Connection = require('mongodb').Connection,
    Server = require('mongodb').Server;
```

而在一些场景中，会使用其他的数据类型，如Long或者Binary，我们可以通过mongodb模块引用它们：

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

var b = new mongodb.Binary(binary_data);
var l = new mongodb.Long(number_string);
```

8.3.1 连接并创建数据库

要想连接MongoDB服务器，需要使用Server和Db类。可以选择使用Connection类获取服务器监听的默认端口号。具体创建共享db对象的代码，如下所示：

```
var Db = require('mongodb').Db,
    Connection = require('mongodb').Connection,
    Server = require('mongodb').Server;

var host = process.env['MONGO_NODE_DRIVER_HOST'] != null
    ? process.env['MONGO_NODE_DRIVER_HOST'] : 'localhost';

var port = process.env['MONGO_NODE_DRIVER_PORT'] != null
    ? process.env['MONGO_NODE_DRIVER_PORT'] : Connection.DEFAULT_PORT;

var db = new Db('PhotoAlbums',
               new Server(host, port,
                         { auto_reconnect: true,
                           poolSize: 20}),
               { w: 1 });
```

MongoDB和mongodb模块在处理写数据和保持数据一致性上非常灵活。可以向数据库构造器传递标识{w:1}，以保证在调用提供给数据库操作的回调函数之前数据已经成功写入。可以在复制环境中指定更大的数字，或者在不关心写操作是否完成时指定0。