fs.createReadStream读取的文件不存在，就没有办法通知到客户端。要解决这个问题，需要首先确认文件是否存在，然后再执行pipe操作：

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

        var rs = fs.createReadStream(file);
        rs.on(
            'error',
            function (e) {
                res.end();
            }
        );

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

事件

Stream实际上是Node.js的Event类的子类，JavaScript文件中Event类提供连接和触发事件的所有功能。我们可以继承这个类来创建自己的事件触发类。例如，创建一个dummy下载器类，通过延迟2秒调用，伪装成一个远程下载：

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

function Downloader () {
}
Downloader.prototype = new events.EventEmitter();
Downloader.prototype.__proto__ = events.EventEmitter.prototype;
Downloader.prototype.url = null;
Downloader.prototype.download_url = function (path) {
    var self = this;
    self.url = path;
    self.emit('start', path);
    setTimeout(function () {
        self.emit('end', path);
    }, 2000);
}

var d = new Downloader();
d.on("start", function (path) {
    console.log("started downloading: " + path);
});
d.on("end", function (path) {
    console.log("finished downloading: " + path);
});
d.download_url("http://marcwan.com");
```

可以通过调用带有事件名称的emit方法来触发事件，并且向监听函数传递任何参数。我们需要确保所有可能的代码块（包括错误）都会触发某个事件；否则，Node程序可能会挂起。

6.2 在客户端组装内容：模板

在传统的Web应用模型中，客户端发送一个HTTP请求到服务器，服务器收集所有的数据并生成对应的HTTP响应，最后以文本形式发送出去。这种处理方式虽然非常可靠，但还是有一些明显的缺点：

- 没有充分利用当今普通客户端电脑的计算能力。现在即使普通的移动电话或者平板电脑都比10年前的PC强大许多倍。

- 当有多种类型的客户端时，会非常难处理。许多人可能通过Web浏览器访问，有些则通过移动应用访问，甚至有人会通过桌面应用或者第三方应用访问。

- 在服务器上运行这么多种类型的东西会非常糟糕。如果只让服务器专注于处理、存储以及生成数据，而让客户端决定如何展现数据会更合理一些。

这种方式日益常见，我还发现Node有一点特别引人注目和好玩，就是服务器只提供JSON格式数据，或者尽可能小的数据。客户端可以选择如何展现返回的数据。脚本、样式表甚至大部分HTML都可以托管在文件服务器或内容分发网络（CDN）中。

对于Web浏览器端的应用，可以使用客户端模板（见图6.1）。这种方式需要：

1.客户端下载HTML页面的骨架，包含了JavaScript文件、CSS文件以及一个来自Node服务器的空body元素的指针。

2.引入的JavaScript文件中有个加载器（bootstrapper），它会处理包括收集所有数据和组装页面等所有工作。需要给HTML模板的名字以及一个供调用的服务器JSON API。引导程序下载模板，然后使用模板引擎将返回的JSON数据应用到模板文件中。接下来我们会使用一个叫"Mustache"的模板引擎（参见补充内容“模板引擎”）。

3.返回的HTML代码被插入页面的body元素中，服务器所需要做的仅仅是处理一点点JSON数据。

这里有个前提，就是现在已经把相册应用开发成一个JSON服务器。尽管还会为它添加处理静态文件的能力，但这主要是为了方便和学习的目的；在生产环境中，应该将这些文件移到CDN中，并适当地更新指向它们的URL。

模板引擎

目前有大量的客户端（甚至是服务器端）模板引擎和模板库。本书中最常用的是jQuery JavaScript库，另外还有一些，例如JSRender或者jQote的模板引擎，这两个引擎也都非常不错。同样还有很多其他不需要适配本书选用的JavaScript类库的模板引擎。

对于本书中的需求，这些模板都大致相同。它们的特性集都基本相同，具有类似的处理方式以及基本一样的性能。当要选用模板时，建议选择正在持续开发和（或）维护的、有合理语法的模板。

本书和我最近的一些项目都使用了叫做Mustache的模板引擎。正如之前所说，这个决定有些随意，但mustache.js有以下优点：

- 它是一个相当小巧且快速的JavaScript库。
- 它是一个功能完全的模板解决方案。
- 它有一个相当酷的名字。Mustache的标签用{{和}}分割，它们看起来确实很像胡须。

这个解决方案并不完美，但我们会发现Mustache在特性和速度之间不断权衡。我们鼓励大家多去寻找其他解决方案，并阅读相关的博客和教程。

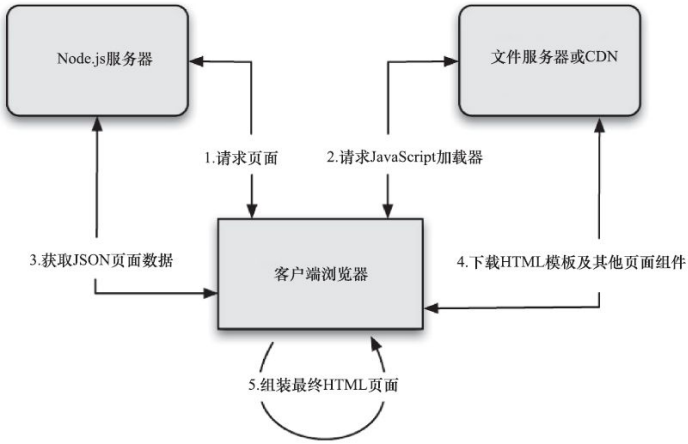

图6.1　使用模板生成客户端页面

为了将相册转换成模板形式，需要完成如下工作：

1）为页面生成HTML骨架。

2）为应用程序添加处理静态文件内容功能。

3）修改支持的URL列表，分别为HTML页面和模板添加对应的/pages/和/templates/。

4）编写模板和加载这些模板的JavaScript文件。

一切就绪，那我们出发吧！

6.2.1　HTML骨架页面

尽管可以尝试在客户端上生成所有的HTML页面，但让服务器完全不做任何事情却是不可能的。我们还是需要服务器输出初始页面的骨架，而客户端可以用它来生成剩余的部分。

在下面的例子中，使用一个合理而简单的HTML页面，保存为basic.html，如代码清单6.1所示。

代码清单6.1　简单的应用页面加载器（basic.html）

```
<!DOCTYPE html>
<html>
<head>
  <title>Photo Album</title>

  <!-- Meta -->
  <meta http-equiv="content-type" content="text/html; charset=utf-8" />

  <!-- Stylesheets -->
  <link rel="stylesheet" href="http://localhost:8080/content/style.css"
        type="text/css" />

  <!-- javascripts -->
  <script src="http://localhost:8080/content/jquery-1.8.3.min.js"
          type="text/javascript"></script>
  <script src="http://localhost:8080/content/mustache.js"
          type="text/javascript"></script>
  <script src="http://localhost:8080/content/{{PAGE_NAME}}.js"
          type="text/javascript"></script>
</head>
<body></body>

</html>
```

这个文件相当直截了当，它有一个样式表，然后引入jQuery和Mustache。最后一个script标签是最有意思的：这个文件是页面的加载器，其中的代码会下载模板，从服务器中获取这个页面对应的JSON数据，并为它生成HTML。可以看到在任何时候，加载器通过参数{{PAGE_NAME}}替换成实际将要加载的页面。

6.2.2 处理静态内容

现在我们需要对应用服务器文件夹的布局稍作修改，如下所示：

```
+ project_root/
    + contents/     # JS, CSS, and HTML files
    + templates/    # client-side HTML templates
    + albums/       # the albums we've seen already
```

之前看到的是albums文件夹，而现在需要添加内容和模板文件夹，用于处理即将添加的额外内容。

更新后的handle_incoming_request函数如下所示。serve_static_file函数则和前面章节中所写的一样：

```
function handle_incoming_request(req, res) {
    // parse the query params into an object and get the path
    // without them. (true for 2nd param means parse the params).
    req.parsed_url = url.parse(req.url, true);
    var core_url = req.parsed_url.pathname;
```

```
    // test this updated url to see what they're asking for
    if (core_url.substring(0, 9) == '/content/') {
        serve_static_file("content/" + core_url.substring(9), res);
    } else 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());
    }
}
```

例如，在content/文件夹中，首先会有三个文件：

- jquery-1.8.3.min.js——可以直接从jquery.com上下载。过去几年的任何jQuery版本应该都能正常工作。

- mustache.js——可以从很多途径下载这个文件。在这里，简单地使用从github.com/janl/mustache.js下载的通用版本。还有许多服务器端的mustache版本，但这不是这一章我们需要的。

- style.css——可以创建一个空文件或写入一些CSS来修改即将为相册应用生成的页面。

6.2.3 修改URL解析机制

目前应用服务器只支持/albums.json和/albums/album_name.json。现在可以添加对静态内容、页面以及模板文件的支持，如下所示：

```
/content/some_file.ext
/pages/some_page_name[/optional/junk]
/templates/some_file.html
```

对于内容和模板文件，可以直接从硬盘下载真实的文件。而对于页面请求，则总是返回之前看到的basic.html的定制版本。它有个{{PAGE_NAME}}宏，用于替换通过URL传入的页面名字，即/page/home。

更新后的handle_incoming_request函数如下所示：

```
function handle_incoming_request(req, res) {
    // parse the query params into an object and get the path
    // without them. (true for 2nd param means 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.substring(0, 7) == '/pages/') {
        serve_page(req, res);
    } else if (core_url.substring(0, 11) == '/templates/') {
        serve_static_file("templates/" + core_url.substring(11), res);
    } else if (core_url.substring(0, 9) == '/content/') {
        serve_static_file("content/" + core_url.substring(9), res);

    } else 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());
    }
}
```

实际上，serve_page函数决定了客户端请求的是什么页面，它会修改basic.html模板文件，并发送最终的HTML骨架到浏览器：

```
/**
 * All pages come from the same one skeleton HTML file that
 * just changes the name of the JavaScript loader that needs to be
 * downloaded.
 */
function serve_page(req, res) {

    var core_url = req.parsed_url.pathname;
    var page = core_url.substring(7);       // remove /pages/
    // currently only support home!
    if (page != 'home') {
        send_failure(res, 404, invalid_resource());
        return;
    }

    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);
        }
    );
}
```

我们使用fs.readFile替代fs.createReadStream读取整个文件的

内容到Buffer。必须将该Buffer转换成字符串，然后修改其内容以指定正确的JavaScript加载器。不推荐对大文件使用这种方法，因为这会浪费大量内存，但对于与basic.html一样短的文件，这会相当方便。

6.2.4 JavaScript加载器

HTML页面非常简单，因此我们可以从一个很小的JavaScript加载器开始。当本书后续章节添加更多功能后，它们会变得略微复杂，代码清单6.2展示了一个现在可用的JavaScript加载器，保存为home.js。

代码清单6.2 JavaScript页面加载器（home.js）

```
$(function(){

    var tmpl,   // Main template HTML
    tdata = {};  // JSON data object that feeds the template

    // Initialize page
    var initPage = function() {
        // Load the HTML template
        $.get("/templates/home.html", function(d){                    // 1
            tmpl = d;
        });

        // Retrieve the server data and then initialize the page
        $.getJSON("/albums.json", function (d) {                      // 2
            $.extend(tdata, d.data);
        });

        // When AJAX calls are complete parse the template
        // replacing mustache tags with vars
        $(document).ajaxStop(function () {
            var renderedPage = Mustache.to_html(tmpl, tdata);         // 3
            $("body").html(renderedPage);
        })
    }();
});
```

$(function(){...语法和$(document).ready(function()...在jQuery中基本是等价的，只是更加简短。该函数会在所有资源（特别是jQuery和Mustache）都加载完后被调用。它会完成以下三件事情：

1）通过调用URL/tmpl/home.html从服务器请求模板文件，即home.html。

2）在应用里请求展示相册的JSON数据，即/albums.json。

3）最后，将两者赋予Mustache，让其完成模板组装。

6.2.5 使用Mustache模板化

相册应用选择的模板引擎Mustache具有易用、快速、小巧的特点。更有趣的是，它只有标签，封装在{{和}}中间，与胡须极为相似。它没有if语句表达式或者循环结构。标签有一系列的规则，根据提供的数据进行工作。

如前面章节所看到的，Mustache的基本用法是：

```
var html_text = Mustache.to_html(template, data);
$("body").html(html_text);
```

要想打印对象的属性，可以使用下面的代码：

Mustache template:

```
The album "{{name}}" has {{photos.length}} photos.
```

JSON:

```
{
  "name": "Italy2012",
  "photos" : [ "italy01.jpg", "italy02.jpg" ]
}
```

Output:

```
The album "Italy2012" has 2 photos.
```

为集合或者数组中的每一项生成模板，可使用#字符。例如：

Mustache:

```
{#albums}}
   * {{name}}
{{/albums}}
```

JSON:

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

Output:

```
* italy2012
* australia2010
* japan2010
```

如果没有匹配的结果，则什么都不会输出。通过^字符可以捕获到这类情况：

Mustache:

```
{{#albums}}
   * {{name}}
{{/albums}}
{{^albums}}
   Sorry, there are no albums yet.
{{/albums}}
```

JSON:

```
{ "albums" : [ ]  }
```

Output:

```
Sorry, there are no albums yet.
```

如果跟在#字符后的对象不是一个集合或者数组，而是一个对象，那么其中的值会被使用，但不会遍历：

Mustache:

```
{{#album}}
  The album "{{name}}" has {{photos.length}} photos.
{{/album}}
```

JSON:

```
{
  "album": {  "name": "Italy2012",
              "photos" : [ "italy01.jpg", "italy02.jpg" ] }
}
```

Output:

```
The album "Italy2012" has 2 photos.
```

默认情况下，所有的HTML字符都会被转义，<和>会分别被转义成<及>，等等。如果不想让Mustache做这样的处理，则使用额外的Mustache三重括号，{{{和}}}：

Mustache:

```
{{#users}}
   * {{name}} says {{{ saying }}}
     * raw saying: {{ saying }}
{{/users}}
```

JSON:

```
{
  "users" : [ { "name" : "Marc", "saying" : "I like <em>cats</em>!" },
              { "name" : "Bob", "saying" : "I <b>hate</b> cats" },
}
```

Output:

```
* Marc says I like <em>cats</em>!
  * raw saying: I like &lt;em&gt;cats&lt;/em&gt;!
* Bob says I <b>hate</b> cats
  * raw saying: I  &lt;b&gt;hate&lt;/b&gt; cats
```

6.2.6 首页Mustache模板

现在，是时候为首页编写第一个模板了，可以保存为home.html，如代码清单6.3所示。

代码清单6.3 首页模板文件（home.html）

```
<div id="album_list">
  <p> There are {{ albums.length }} albums</p>
  <ul id="albums">
    {{#albums}}
      <li class="album">
```

```
      <a href='http://localhost:8080/pages/album/{{name}}'>{{name}}</a>
    </li>
  {{/albums}}
  {{^albums}}
    <li> Sorry, there are currently no albums </li>
  {{/albums}}
 </ul>
</div>
```

如果有相册，代码会遍历每一个相册项，提供一个<li>元素，包含了相册的名字和指向相册页面的超链接，可以通过URL/page/album/album_name访问。当没有相册时，则会打印简单的信息。

6.2.7 整合应用

上面几节介绍了许多新知识，现在要使它们运行起来，则需要为Web应用提供如下文件布局：

```
+ project_root/
    server.js                                  // GitHub source
    basic.html                                 // Listing 6.1
    + content/
        home.js                                // Listing 6.2
        album.js                               // Listing 6.5
        jquery-1.8.3.min.js
        mustache.js
        style.css
    + templates/
        home.html                              // Listing 6.3
        album.html                             // Listing 6.4
+ albums/
        + whatever albums you want
```

album.html和album.js的代码之前并没有见过，附在代码清单6.4和6.5上。

要运行该项目，需要输入如下代码：

```
node server.js
```

然后打开Web浏览器并浏览http://localhost:8080/page/home。如果想在命令行里看看都发生了什么，可以像下面一样用curl试试：

```
curl -i -X GET http://localhost:8080/page/home
```

因为curl不能执行客户端的JavaScript代码，所以模板和JSON数据不能通过Ajax加载，只能看到来自basic.html的基本HTML骨架。

代码清单6.4　另外一个Mustache模板页（album.html）

```
<div id="album_page">
  {{#photos}}

  <p> There are {{ photos.length }} photos in this album</p>
  <div id="photos">
      <div class="photo">
        <div class='photo_holder'>
          <img style='width: 250px; height: 250px; background-color: #fbb'
               src="{{url}}" border="0"/></div>
        <div class='photo_desc'><p>{{ desc }}</p></div>
      </div>
  </div> <!-- #photos -->
  <div style="clear: left"></div>
  {{/photos}}
  {{^photos}}
      <p> This album doesn't have any photos in it, sorry.<p>
  {{/photos}}
</div> <!-- #album_page -->
```

代码清单6.5　相册页面加载器JavaScript文件（album.js）

```
$(function(){

    var tmpl,   // Main template HTML
    tdata = {};  // JSON data object that feeds the template

    // Initialize page
    var initPage = function() {
        // get our album name.
        parts = window.location.href.split("/");
        var album_name = parts[5];

        // Load the HTML template
        $.get("/templates/album.html", function(d){
            tmpl = d;
        });

        // Retrieve the server data and then initialize the page
        $.getJSON("/albums/" + album_name + ".json", function (d) {
            var photo_d = massage_album(d);
            $.extend(tdata, photo_d);
        });

        // When AJAX calls are complete parse the template
        // replacing mustache tags with vars
        $(document).ajaxStop(function () {
            var renderedPage = Mustache.to_html( tmpl, tdata );
            $("body").html( renderedPage );
        })
    }();
});

function massage_album(d) {

    if (d.error != null) return d;
    var obj = { photos: [] };

    var af = d.data.album_data;

    for (var i = 0; i < af.photos.length; i++) {
        var url = "/albums/" + af.short_name + "/" + af.photos[i].filename;
        obj.photos.push({ url: url, desc: af.photos[i].filename });
    }
    return obj;
}
```

如果上面所有步骤都正确完成，应该可以看到如图6.2所示的页面。页面不是很美观，但可以通过修改模板文件以及style.css文件来

美化它。

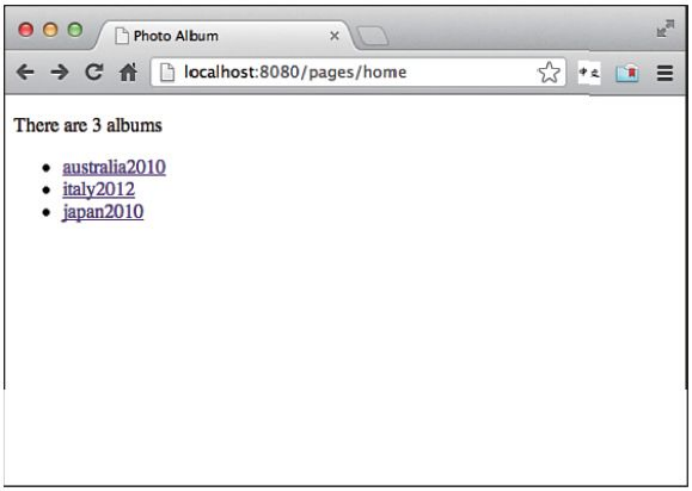

图6.2　运行于浏览器的基于模板的JSON应用程序

6.3 小结

本章介绍了Node.js的两个关键特性：数据流和事件，使用它们可以为Web应用处理静态内容。同样重要的是，本章还介绍了客户端的模板引擎，以及使用Mustache将服务器和相册应用程序的前端部分联系起来。

但大家可能注意到，目前做的许多事情貌似被不必要地复杂化，非常冗长乏味，甚至很可能出现bug（尤其是handle_incoming_request函数）。因此需要一些优秀的模块来协助我们完成这些事情。

因此，接下来我不打算进一步开发相册，而是看一些可以显著清理代码的模块，这些模块还能帮助我们以很少的代码来快速添加功能。我们将会使用Node的express应用框架完成这些工作。

第三部分　实战篇

第7章　使用express构建Web应用

第8章　数据库I：NoSQL（MongoDB）

第9章　数据库II：SQL（MySQL）

第7章　使用express构建Web应用

目前为止，我们已经学习了Node.js的基本原理及核心概念。利用这些知识，我们创建了一些简单的应用，但还是需要用冗长的代码实现简单的功能。现在是做一点改变的时候了：利用Node.js的一大优势——npm上拥有不计其数的类库和模块可供使用——编写一些有趣的Web应用。既然我们已经了解了Node及其模块的核心工作方式，那么现在就探索一下让编程变得简单和快速的方式。

在本章中，我们将会开始学习express——一个为Node量身打造的Web应用框架。它能完成许多前面章节中我们学习到的基本操作，甚至提供了许多强大的功能，让我们在应用中使用。这里，我会演示如何使用express构建应用，以及如何使用它更新相册应用。最后，我们会深入应用设计，学习如何设计JSON服务器的API，并探索一些以前从来没有见过的新功能。

7.1 安装express

在本章中，我们会改变原先使用npm安装模块的方式。前文中，如果想要安装express，只需输入：

```
npm install express
```

这没有任何问题！但是一般情况下，这种方式会安装最新版本的express模块；而在实际生产环境中部署应用时，往往不希望安装模块的最新版本。尤其当开发和测试只针对某个特定版本时，最好在生产服务器中使用相同的版本。只有更新到最新版本并经过严格测试，才能在生产环境中使用新版本。

因此，构建Node应用的基本步骤如下所示：

1）创建应用文件夹。

2）编辑package.json文件，该文件用来描述软件包并指定所需的npm模块。

3）运行npm install命令，确保这些模块都正确安装。

创建名为trivial_express/的新文件夹用来存放express测试应用，然后把下面的代码编辑到该文件夹下的package.json文件中：

```
{
  "name": "Express-Demo",
  "description": "Demonstrates Using Express and Connect",
  "version": "0.0.1",
  "private": true,
  "dependencies": {
    "express": "3.x"
  }
}
```

name和description字段顾名思义，极易理解。现在这个应用还很简单，拥有一个对应的版本号0.0.1。正如第5章里所讲的，我们还可以将private设置为true，表明npm不会将其发布到npm模块注册中心，因为这是一个仅供自己使用的测试项目。

完成文件创建以后，可以运行以下命令，它会读取配置文件中的依赖部分，并安装所有合适的模块：

```
npm install
```

由于express本身就依赖很多不同的模块，因此，在安装过程中会看到大量的打印信息。安装完成之后，可以在node_modules/文件夹中找到安装的模块。如果修改了package.json文件，可以使用npm update命令来确保所有模块都正确安装。

使用express创建"Hello World"

回顾下第1章，第一个Node.js的Web服务器非常粗糙，如下所示：

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

function process_request(req, res) {
    res.end("Hello World");
}

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

现在使用express实现二者，看起来非常相似：

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

app.get('/', function(req, res){
  res.end('hello world');
});

app.listen(8080);
```

保存并运行上述代码，使用curl或者在Web浏览器中输入localhost:8080，可以看到预期的返回结果hello world。最棒的事情是，express是构建在目前我们所学到的知识之上的，因此HTTP请求和响应对象会拥有之前的所有方法和属性。同时，它们还拥有许多

其他有趣和强大的东西，但我们无须操作完全不熟悉的东西，因为这些都是我们之前熟悉的对象的扩展版本。

7.2 express中的路由和分层

express完全构建在一个名叫connect的Node模块之上。connect作为中间件（middleware）类库而为大家所熟知，它提供了一系列网络应用所需的常用功能。该模型的核心是函数流，也就是大家常说的函数分层机制。

中间件：名称的奥秘

当第一次看到中间件这个术语时，也许你会担心错过了什么关键的新概念或技术变革。但幸运的是，它并不是那么复杂。

实际上，中间件一开始是被用来描述软件组件的（一般是服务器端的东西），这种组件会将两个东西连接起来，如事务逻辑层和数据库服务器的连接或者存储服务和应用模块的连接等。

随着时间的迁移，这个术语的定义变得更加通用和宽泛，现在可以用来描述任何连接两个事物的软件片段。

从本书的角度来看，express中使用的connect提供的中间件只是一组单纯的组件，可以让Web应用无缝集成服务器端或浏览器端提供的功能。

使用express创建一系列的函数（层）来构建应用；一旦其中一个函数接管了请求，它就会处理并停止后面的函数序列。express会直接跳到执行函数序列的最后（一些层用来“进栈”，一些用来“出栈”）。

因此，在前面的express应用中，执行序列中仅有一层：

```
app.get('/', function(req, res){ });
```

该函数是express提供的帮助函数，只要满足下述两个条件，就可以用来处理传入的请求（通过调用提供的函数）：

- HTTP请求的方法为GET。

- 请求的URL路径为/。

如果传入的请求不满足以上条件，该函数就不会被调用。如果没有函数与请求的URL匹配——例如，使用curl或浏览器请求/some/other/url——express就会返回一个默认的响应：

```
Cannot GET /some/other/url
```

7.2.1 路由基础

URL路由函数的一般格式如下：

```
app.method(url_pattern, optional_functions, request_handler_function);
```

前面的路由处理程序用来处理URL/传递过来的GET请求。如果想要支持处理某个特定URL下的POST请求，可以使用express应用对象上的post方法：

```
app.post("/forms/update_user_info.json", function (req, res) { ... });
```

同样，express也支持DELETE和PUT方法，通过delete和put方法，我们会在后面的相册应用中使用到它们。还可以通过all方法让路由函数接受所有指定URL的HTTP方法：

```
app.all("/users/marcwan.json", function (req, res) { ... });
```

路由函数的第一个参数使用正则表达式匹配传入的URL。所以，它支持简单的URL：

```
/path/to/resources
```

也支持正则表达式，如：

```
/user(s)?/list
```

上述表达式会匹配/users/list和/user/list，而

```
/users/*
```

会匹配所有以/users/开头的URL。

express路由最强大的特性之一就是支持占位符从请求路由中提取指定值，使用冒号（:）来标记。当解析路由的时候，express会将匹配到的占位符值保存到req.params对象中，例如：

```
app.get("/albums/:album_name.json", function (req, res) {
    res.end("Requested " + req.params.album_name);
});
```

如果调用上面的app请求/albums/italy2012.json，可以得到以下输出结果：

```
Requested italy2012.
```

同样，我们也可以在同一个URL上添加多个参数：

```
app.get("/albums/:album_name/photos/:photo_id.json", function (req, res) { ... );
```

一旦该函数被调用，req.params中的album_name和photo_id就会被赋予传递进来的值。除了斜杠（/），占位符可以匹配任意字符串。

正如我们看到的一样，上述路由方法中提供的回调函数已经被赋予请求和响应对象。而实际上它还被赋予了第三个参数，你可以选择使用或者忽略它。该参数一般使用next这个变量名（当函数调用next以后，分层函数流就会传递给下一个分层函数）。我们还可以对传入的URL请求做一些额外的检测，或者选择忽略它，如下所示：

```
app.get("/users/:userid.json", function (req, res, next) {
    var uid = parseInt(req.params.userid);
    if (uid < 2000000000) {
        next();              // don't want it. Let somebody else process it.
    } else {
        res.end(get_user_info(uid));
    }
});
```

7.2.2 更新相册应用路由

该相册应用很容易使用express重构。我们只需要做一点工作就能让它正常运行：

- 创建package.json文件，并安装express。
- 使用express替换http模块的服务器。
- 使用路由处理程序替换handle_incoming_request函数。
- 更新在express中获取查询参数的方式（为了方便使用，这些参数会存放到req.query对象中）。

现在，将第6章最后创建的相册应用复制到一个新文件夹中（将其命名为basic_routing/），并将下面的package.json文件放到该文件夹下：

```
{
  "name": "Photo-Sharing",
  "description": "Our Photo Sharing Application",
  "version": "0.0.2",
  "private": true,
  "dependencies": {
    "async": "0.1.x",
    "express": "3.x"
  }
}
```

运行npm install命令，确保express和async安装到node_modules/文件夹下。可以将server.js文件顶部的http模块替换成express：

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

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

并将

```
http.listen(8080);
```

修改成

```
app.listen(8080);
```

接下来需要替换handle_incoming_request函数，代码如下所示：

```
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.substring(0, 7) == '/pages/') {
        serve_page(req, res);
    } else if (core_url.substring(0, 11) == '/templates/') {
        serve_static_file("templates/" + core_url.substring(11), res);

    } else if (core_url.substring(0, 9) == '/content/') {
        serve_static_file("content/" + core_url.substring(9), res);
    } else 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());
    }
}
```

使用下面的路由函数替换上述代码：

```
app.get('/albums.json', handle_list_albums);
app.get('/albums/:album_name.json', handle_get_album);
app.get('/content/:filename', function (req, res) {
    serve_static_file('content/' + req.params.filename, res);
});
app.get('/templates/:template_name', function (req, res) {
    serve_static_file("templates/" + req.params.template_name, res);
});
app.get('/pages/:page_name', serve_page);
app.get('*', four_oh_four);

function four_oh_four(req, res) {
    send_failure(res, 404, invalid_resource());
}
```

现在的路由函数不仅比原先的函数更加干净简洁，而且能直观地看出URL是如何匹配请求及整个应用是如何工作的。我们还添加了一个新的路由匹配“*”，这样其他所有的请求将会返回404响应。

但是，现在路由函数面临一个棘手的问题。如果用户请求/pages/album/italy2012会怎样？就目前而言，这个应用会失败：因为正则表达式不能匹配带有斜杠（/）字符的参数，所以路由/pages/:page_name匹配不到该请求路径。为了解决这个问题，必须添加一个新的路由函数：