CHAPTER 9

Express

鉴于Connect基于HTTP模块提供了开发Web应用常用的基础功能，Express基于Connect为构建整个网站以及Web应用提供了更为方便的 API。

纵观第8章中的例子，你可能已经发现了，绝大部分Web服务器和浏览器之间交互的任务都是通过URL和method来完成的。这两者的组合有时又称为路由，通过Express创建的应用中的一个基础概念。

Express基于Connect构建而成，因此，它也保持了重用中间件来完成基础任务的想法。这就意味着，通过Express的API方便地构建Web应用的同时，又不失构建于HTTP模块之上高可用中间件的生态系统。

接下来，我们通过Express构建一个查询Twitter API的小应用，来了解Express的用法和功能。

一个小型Express应用

应用虽小但五脏俱全。在用户通过查询关键词查询“推文”时，需要返回包含查询结果的HTML页面。另外，我们这次不在请求处理器中采用字符串拼接的方式来返回HTML页面内

容，而是使用一个简单的模板语言来实现，并将逻辑从视图层分离到控制层。

那么第一步就是要确保将所需的模块引入进来。

创建模块

按照惯例，先创建一个package.json文件，这次需要多添加两个额外的依赖：ejs。本例中要使用的模板引擎，以及superagent来简化向 Twitter发送HTTP查询请求的实现。

```
{
    "name": "express-tweet"
  , "version": "0.0.1"
  , "dependencies": {
        "express": "2.5.9"
      , "ejs": "0.4.2"
      , "superagent": "0.3.0"
    }
}
```

这里请注意，尽管本例中我们使用的是Express 2，但代码应该是能和 Express 3完全兼容的（截止本书撰写期间Express 3还处在开发阶段）。

定义好项目的元数据之后，接下来创建HTML模板。

HTML

和上一个应用不同的是，为了避免将HTML代码嵌入到应用逻辑（通常叫做处理器或者路由处理器）中，这次我们要使用一个简单的模板语言来处理。模板语言的名字是EJS（内嵌（embedded）的js），和在HTML中内嵌PHP类似。

从在views/文件夹中创建一个index.ejs文件开始。事实上，模板可以放在任何一个地方，不过考虑到本例的项目结构，我们就将其单独放到views目录下。

首个模板对应默认的路由路径（首页）。它提供一个入口让用户提交搜索推文的关键字：

```
<h1>Twitter app</h1>
<p>Please enter your search term:</p>
<form action="/search" method="GET">
  <input type="text" name="q">
  <button>Search</button>
</form>
```

147

另一个模板search.ejs用于展示查询结果。高亮查询关键词并将查询结果逐一显示出来（如果有的话），否则就显示一条消息：

```
<h1>Tweet results for <%= search %></h1>
<% if (results.length) { %>
  <ul>
```

```
    <% for (var i = 0; i < results.length; i++) { %>
    <li><%= results[i].text %> - <em><%= results[i].from_user %></li>
    <% } %>
  </ul>
<% } else { %>
  <p>No results</p>
<% } %>
```

如上面所示，我们将JavaScript代码嵌在<%和%> EJS标签中。另外，我们通过在<%之后加入“=”符号将变量值打印出来。

SETUP

我们现在server.js文件中引入依赖的模块：

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

引入Express之后，需要用它来初始化Web服务器。和Connect类似，Express提供了一个createServer快捷方法返回一个Express HTTP服务器。就像这样：

```
var app = express.createServer()
```

和其他流行的Web框架不同，Express并不要求任何必要的配置，也不对文件结构有特殊的要求。它足够灵活，允许你对每一个单独的功能点进行自定义。

本例中，我们需要指定使用的模板引擎（这样就不需要每次载入视图时都去引入）以及视图文件（模板）所在的目录。刚刚介绍的express.createServer方法返回的HTTP服务器自带配置系统。我们可以通过调用该对象上的set方法来修改默认的配置项。添加如下代码：

```
app.set('view engine', 'ejs');
app.set('views', __dirname + '/views');
app.set('view options', { layout: false });
```

第三个view options参数所定义的选项，在渲染视图时，会传递到每个模板中。这里layout的值设置为false，是为了匹配Express 3中的默认值。

要获取配置信息，可以调用app.set方法并传递对应要获取配置的标志。比如，要获取views的配置信息，可以通过如下方式： 148

```
console.log(app.set('views'));
```

接下来，我们会使用Express提供的方法来定义路由，关于路由的有关内容，我们其实在第7章和第8章就已经接触过了。

定义路由

通过使用Express来定义路由就无须手动地每次去检查method和url属性，只需调用

Express提供的对应HTTP method的方法，并将URL和对应的处理中间件传递进去就可以了。

Express支持的方法有get、put、post、del、patch以及head，分别对应HTTP的GET、PUT、POST、DELETE、PATCH以及HEAD。下面是定义路由的例子：

```
app.get('/', function (req, res, next) {});
app.put('/post/:name', function (req, res, next) {});
app.post('/signup', function (req, res, next) {});
app.del('/user/:id', function (req, res, next) {});
app.patch('/user/:id', function (req, res, next) {});
app.head('/user/:id', function (req, res, next) {});
```

第一个参数是路由地址，第二个参数是路由处理程序。路由处理程序就和中间件一样。

注意了，路由部分还可以通过特殊格式来定义变量。如上述例子中的/user/:id，哪怕id值不同，路由也能匹配到：如/user/2、/user/3，等等。下一章会对这部分内容做详细介绍。

下面我们来定义首页的路由：添加如下代码到server.js文件中：

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

到目前为止，完整代码如下所示：

```
/**
 * 模块依赖
 */
var express = require('express')
  , search = require('./search')

/**
 * 创建app
 */
var app = express.createServer();

/**
 * 配置
 */

app.set('view engine', 'ejs');
app.set('views', __dirname + '/views');
app.set('view options', { layout: false });

/**
 * 路由
 */

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

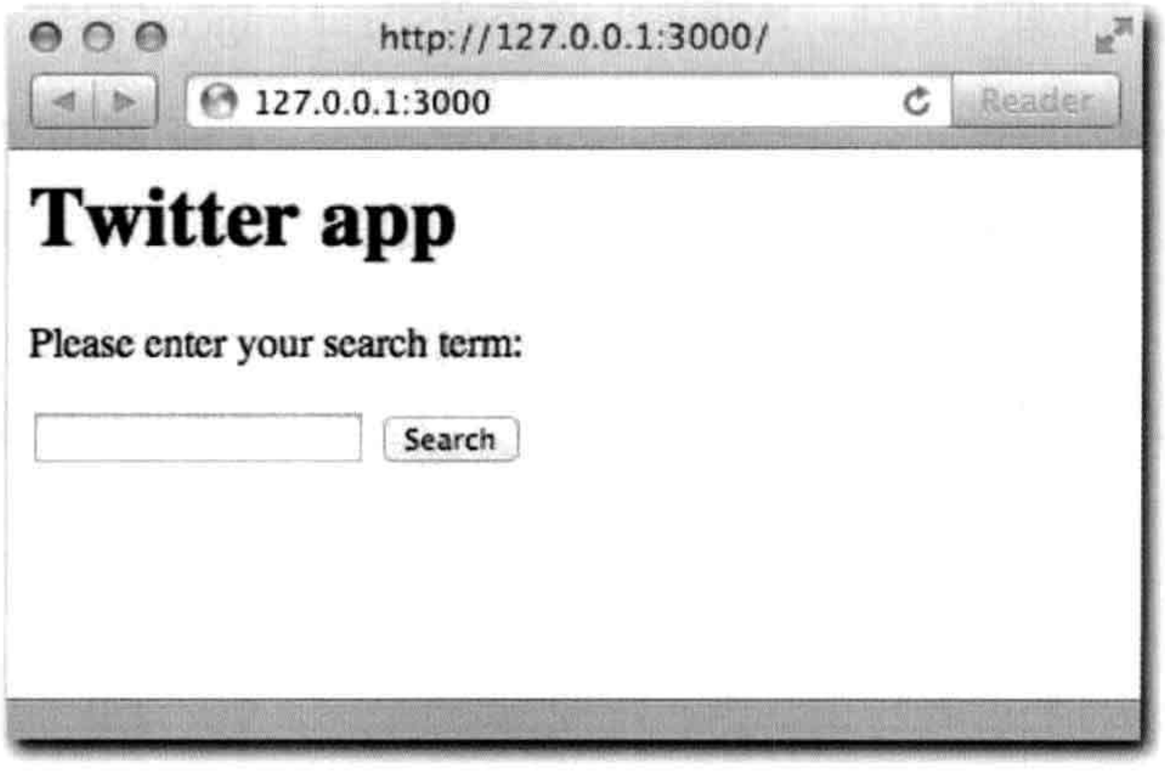
```
  res.render('index');
});

/**
 * 监听
 */
app.listen(3000);
```

Express为response对象提供了render方法。该方法完成如下三件事：

1. 初始化模板引擎。
2. 读取视图文件并将其传递给模板引擎。
3. 获取解析后的HTML页面并作为响应发送给客户端。

由于在前一步中将ejs指定为视图引擎，因此无须显式地指明index.ejs了。

运行之后结果如图9-1所示（别忘记调用listen方法）。



150

图9-1：路由处理器/渲染index视图

第二个路由中，我们调用一个名为search的方法（将在另外一个模块中定义）：

```
app.get('/search', function (req, res, next) {
  search(req.query.q, function (err, tweets) {
    if (err) return next(err);
    res.render('search', { results: tweets, search: req.query.q });
  });
});
```

接着，在express之后添加对search模块的依赖：

```
var express = require('express')
  , search = require('./search')
```

这里要注意，如果search方法回调函数中接收到错误对象，那么我们就直接把它传递给

next。后面章节会介绍错误处理的相关内容，到时你就能明白为什么要这么做了，目前，我们就假设Express会将错误信息通知给用户。

这个路由处理器中，我们也调用了render方法，不过不同的是，这次我们传递了一个对象作为第二个参数。该对象的内容会暴露给视图。注意这里我们是如何把tweets和search传递过去的，这两个变量可以在search.ejs视图中直接使用。这类对象称为本地变量，因为它的内容只对其传递的视图可见。

查询

查询模块暴露一个简单的方法提供对推文的查询，其内部调用了Twitter的查询API。本例中，我们将查询模块search.js文件和server.js文件放到同一目录中。

上述调用search方法的代码中，传递了一个查询关键字以及回调函数，回调函数接收两个参数，其一是错误对象（如果有的话），其二是一个包含了查询到的推文的数组。

151 要写这样一个模块，我们先来定义依赖的模块。本例中，只需用到 superagent：

```
var request = require('superagent')
```

由于向Twitter的Web服务发送HTTP请求是本例中重要的功能，我们需要确保查询模块正确地进行了错误处理。

比如，要是Twitter API服务宕机了，我们就发送一个错误对象，来给用户显示一个错误页面（如：显示HTTP错误状态码500）。

```
/**
 * Search function.
 *
 * @param {String} search query
 * @param {Function} callback
 * @api public
 */

module.exports = function search (query, fn) {
  request.get('http://search.twitter.com/search.json')
    .data({ q: query })
    .end(function (res) {
      if (res.body && Array.isArray(res.body.results)) {
        return fn(null, res.body.results);
      }
      fn(new Error('Bad twitter response');
    });
};
```

和其他superagnet例子类似，我们发送一个GET请求，将查询关键字作为数据字段q的值

以查询字符串的形式发送出去。以hello world为查询关键字的URL类似`http://search.twitter.com/search.json? q=hello+world`。

在响应处理程序中，我们确保请求发送成功并且完全符合我们的预期。相比查看HTTP响应码是否为200，我们采用更加聪明的方式：直接查看响应结果是否包含含有推文内容的数组。

第7章中我们就介绍过，如果superagent获取到一个JSON形式的响应消息，它会自动进行解码，并将解码后的内容放到res.body变量中。由于Twitter API会返回一个JSON对象，对象中的results字段内容就是包含了推文的数组，因此，下述代码段对于进行错误处理的判断来说足矣：

```
if (res.body && Array.isArray(res.body.results)) {
   return fn(null, res.body.results);
}
```

运行

152

运行上述服务器代码，并通过浏览器访问`http://localhost:3000`（见图9-2），试着输入推文关键字（见图9-3）。

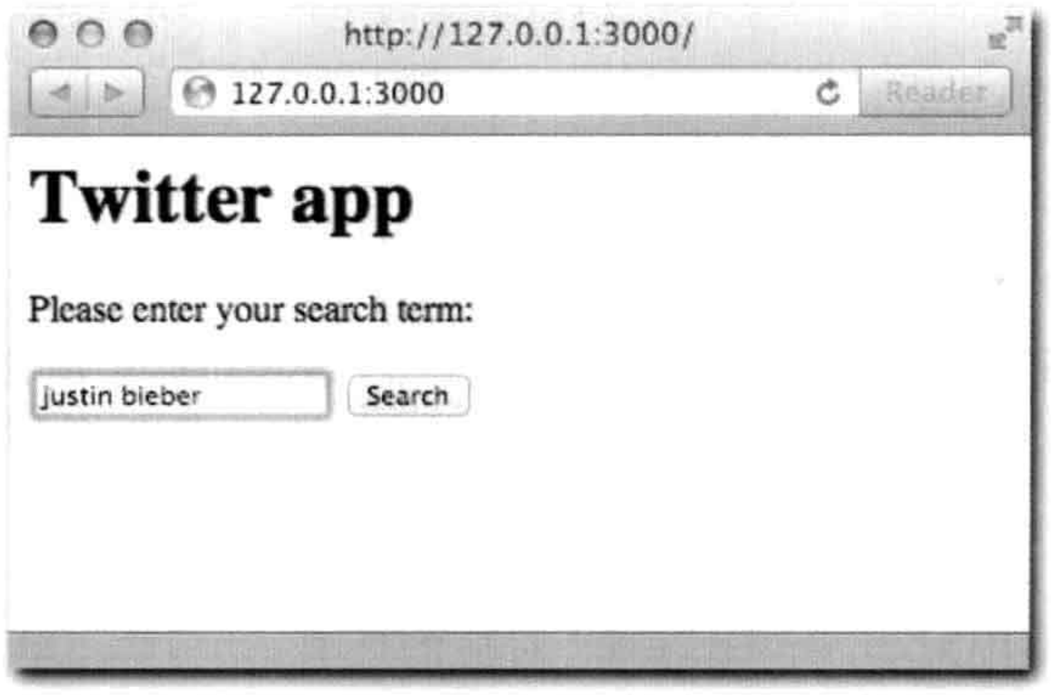


图9-2：输入查询推文关键字并提文的例子

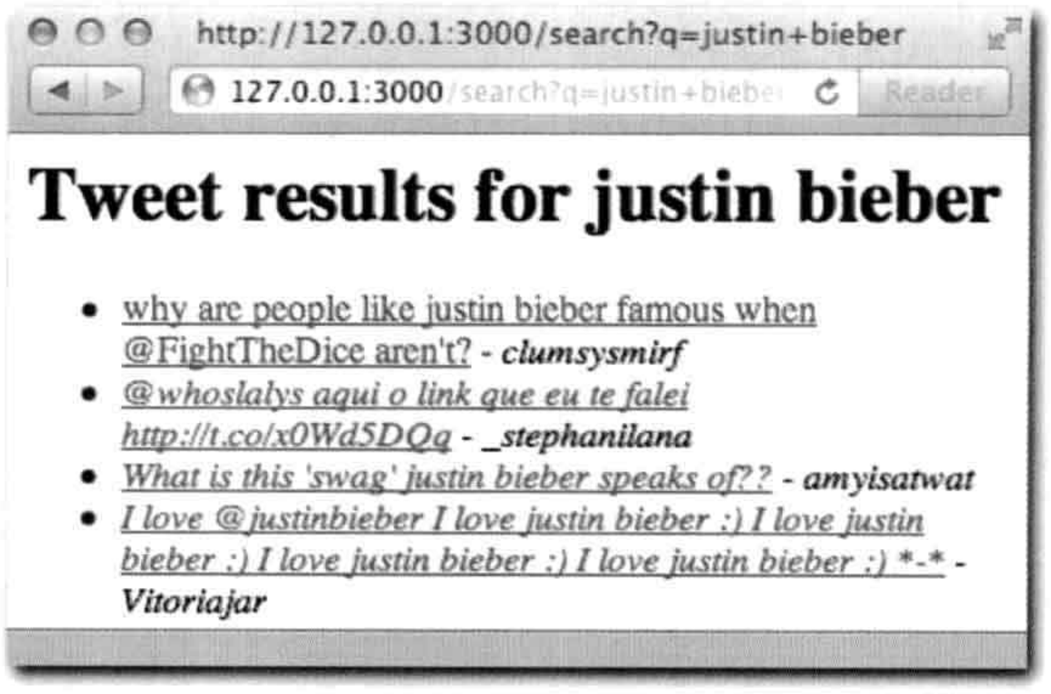


图9-3：查询结果

成功！向Twitter查询推文后，我们获得了一个包含推文的数组。最终它会显示在根据推文数组动态生成的search.ejs模板中。

HTML页面由Express的渲染引擎产生，/search路由成功地将完整的页面显示给用户，如图9-3所示。

在这个简单的示例后，接下来该是深入研究Express特性的时候了，我们来看一些新的功能。

153 设置

Express提供了最有意思的特性之一，同时也是对任意类型Web应用都不可或缺的，就是对环境设置的管理能力。

比如，在生产环境下，为了提高性能，我们可以让express将模板缓存起来，这样可以加快响应速度。然而，若在开发模式下开启这个功能，每次对模板稍做改动就需要重启Node服务程序才能生效。

通过如下方式就可以达到让Express对模板进行缓存的效果：

```
app.configure('production', function () {
  app.enable('view cache');
});
```

在上述代码中，app.enable就等同于此前例子中为了配置views config标志所用到的app.set。

```
app.set('view cache', true);
```

要查看配置标志是否启用，也可以调用app.enabled。对应的还有app.disable和app.disabled方法。

当环境变量NODE_ENV设置为production时，我们在app.configure定义的对应的回调函数就会被执行。

要测试上述代码，运行如下命令：

```
$ NODE_ENV=production node server
```

要是NODE_ENV没有设置，则默认会调用development的配置：

```
app.configure('development', function () {
  // gets called in the absence of NODE_ENV too!
});
```

下面还有一些内置的实用设置：

- 大小写敏感的路由：启用大小写敏感的路由。默认情况下，当定义如下路由：

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

Express能匹配到/my/route和/MY/ROUTE。开启之后，路由只会匹配定义的路由，上述例子中就只会匹配/my/route。

- 严格路由：启用后，最后的斜杠就不会自动忽略。什么意思呢？比如，此前例子中能匹配/my/route和/my/route/，而一旦开启严格路由模式，只会匹配到/my/route，因为路由就是这样在app.set中定义的。 154
- jsonp回调：启用res.send() / res.json()对jsonp的支持。JSONP是一项解决跨域JSON请求的技术，它将响应结果包裹在一个用户指定的回调函数中。
  - 当有JSONP请求时，其URL类似这样：/my/route?callback=myCallback。Express会自动检测callback参数，并将相应结果包裹在myCallback文本中。要启用该功能，可以调用app.enable('jsonp callback')。这里要注意，这只作用于res.send和res.json，后面的章节会有相关介绍。

模板引擎

要像此前例子那样使用ejs，必须完成以下两个步骤：

1. 通过NPM安装ejs模块。
2. 声明view engine为ejs。

如下的其他模板引擎也可以在Express中使用：

- Haml
- Jade
- CoffeeKup
- jQuery Templates for node

Express会试着以模板文件扩展名或者以配置的view engine的值为名去调用require方法。比如，可以调用：

```
res.render('index.html')
```

这时，Express会尝试着去require html引擎。找不到该引擎，就会报错。

也可以通过app.register API将扩展名匹配到指定的模板引擎。比如，通过如下方式

就可以将html扩展名匹配到jade模板引擎：[1]

```
app.register('.html', require('jade'));
```

Jade是最流行的模板语言之一，绝对值得一学。要想了解更多关于Jade的内容可以访问其官方网站http://jade-lang.org。

155 错误处理

在Node中，将错误对象作为非阻塞I/O回调函数的第一个参数是很常见的。在本章的例子中，我们也有可能在调用Twitter API进行查询时接收到错误对象。

面对这种情况，在Express中常规的做法就是将该错误对象传递给next函数。默认情况下，Express会展示一个错误页面并发送500状态码。

绝大多数Web应用都会自定义错误页面，或者甚至自建一套后台的错误处理机制。

我们可以通过app.error方法定义一个特殊的错误处理器作为错误处理的中间件：

```
app.error(function (err, req, res, next) {
  if ('Bad twitter response' == err.message) {
    res.render('twitter-error');
  } else {
    next();
  }
});
```

注意在上述代码中，通过检测错误消息内容来判断是否要对错误进行处理，要是检测结果不匹配就直接调用next。

还可以设置多个.error处理器来执行不同的处理。比如，最后一个错误处理器就可以发送500 Internal Server Error并展示一个通用的错误页面。

```
app.error(function (err, req, res) {
  res.render('error', { status: 500 });
});
```

当调用next并且对应的处理器无法找到时，默认的Express错误处理器就会触发。

快捷方法

Express为Node中的Request和Response对象提供了一系列扩展来简化开发。

Request对象上的扩展如下。

---

1 译者注：Express 3.x中需要使用app.engine方法。

- header：此扩展能够让程序以函数的方式获取头信息，并且还是大小写不敏感的。

```
req.header('Host')
req.header('host')
```

- accepts：此扩展会分析请求中的Accept头信息，并根据提供的值返回true或者false。 156

```
req.accepts('html')
req.accepts('text/html')
```

- is：此扩展和accepts类似，但它检查Content-Type头信息。

```
req.is('json')
req.is('text/html')
```

Response对象上的扩展如下。

- header：此扩展接收一个参数来检查对应的头信息是否已经在 response上设置了。

```
res.header('content-type')
```

或设置header的两个参数：

```
res.header('content-type', 'application/json')
```

- render：对render相信你已经了解不少了。不过，在此前的例子中，你也许注意到了我们传递了status值。这是一个特殊的类型，设置了它就等于为response响应对象设置了状态码。
  - 除此之外，还可以提供第三个参数给render方法来获取 HTML内容而不是直接将其作为响应消息自动传递出去。

```
res.render('template', function (err, html) {
  // 处理收到的 html
});
```

- send：此扩展很奇特。它会根据提供参数的类型执行响应的行为。
  - Number：发送状态码。

```
res.send(500);
```

  - String：发送HTML内容。

```
res.send('<p>html</p>');
```

  - Object/Array：序列化成JSON对象，并设置相应的Content-Type头信息。

```
res.send({ hello: 'world' }); res.send([1,2,3]);
```

- json：此扩展在绝大多数场景下和send类似。只是它会显式地将内容作为JSON对象发送。

```
res.json(5);
```

在发送值类型未知的情况下可以使用此方法。res.send会判断发送值的类型，并且依据判断结果来选择是否调用JSON.stringify方法。如果是数字类型，那么会认为发送的是状态码，并直接结束响应。而res.json会把数字类型也进行JSON.stringify转换。

由于绝大部分情况下我们会对发送对象进行编码，所以，res.send还是很不错的选择。

157

- redirect：redirect等效于发送302（暂时移除）状态码以及 Location头信息。如下所示。

```
res.redirect('/some/other/url')
```

就等效于：

```
res.header('Location', '/some/other/url');
res.send(302);
```

上述代码在Node.js内部其实是这样处理的：

```
res.writeHead(302, { 'Location': '/some/other/url' });
```

- redirect还可以接收自定义的状态码作为第二个参数。假设你不想发送302而是发送表示永久性移除的301状态码，可以采取如下方式。

```
res.redirect('/some/other/url', 301)
```

- sendfile：此扩展和Connect中的static中间件类似，不同之处在于它用于单个文件。

```
res.sendfile('image.jpg')
```

路由曾在我们的示例应用中有所应用，事实上，路由还有很多内容有待我们了解，它们对大型Web应用都非常有帮助。

路由

在定义路由时，可以使用自定义参数：

```
app.get('/post/:name', function () {
  // . . .
})
```

上述代码中，name变量值会注入到req.params对象上。比如，当通过浏览器访问'/post/hello-world'时，req.params对象会变为如下形式：

```
app.get('/post/:name', function () {
  // req.params.name == "hello-world"
})
```

还可以通过在变量后添加问号（？）来表示该变量是可选的。在此前的路由示例中，要是通过浏览器访问/post，那么该路由是不会匹配到的。要匹配到可以采用如下方式：

```
app.get('/post/:name?', function (req, res, next) {
  // this will match for /post and /post/a-post-here
})
```

像这样定义了参数的路由，内部会当正则表达式处理。也就是说，定义路由时也可以直接使用RegExp对象。比如，只想匹配字母、数字以及中划线的话，可以这样： 158

```
app.get(/^\/post\/([a-z\d\-]*)/, function (req, res, next) {
  // req.params contains the matches set by the RegExp capture groups
})
```

和中间件一样，在路由处理程序中也可以使用next。即使当一个路由匹配并得到处理，还是可以强制Express去继续匹配其他路由的。

比如，让路由只接受以'h'开头的参数：

```
app.get('/post/:name', function (req, res, next) {
  if ('h' != req.params.name[0]) return next();
  // . . .
});
```

就是因为Express提供了灵活的路由，才能实现像上述这样的处理代码来解决多变的情况。

举例来说，许多Web应用允许如/home、/about这样的路由，但它们同时又希望能够让动态的内容也能有永久的URL。

在定义好所有的路由之后，可以再定义一个路由来获取用户名，并进行数据库调用。如果该用户未能找到，就调用next并发送404，否则就渲染该用户个人页面。

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

app.get('/:username', function (req, res, next) {
  // if you got here, no prior application routes matched
  getUser(req.params.username, function (err, user) {
    if (err) return next(err);

    if (exists) {
      res.render('profile')
    } else {
```

```
        next();
      }
    });
});
```

如你所知，Express也沿袭了中间件的概念。下面我们就来深入了解一下。

159 中间件

由于Express是构建于Connect之上的，所以，当创建Express服务器时可以使用Connect兼容的中间件。比如，要托管images/目录下的图片，就可以像这样使用static中间件：

```
app.use(express.static(__dirname + '/images'));
```

或者，要想使用connect的session，也很容易，像这样就可以了：

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

注意了，在引入了Express之后就可以直接使用Connect的中间件了。不需要require('connect')或者把connect作为项目依赖添加到package.json文件中。

中间件是易理解的。

更有意思的是，和全局中间件（针对每个请求）不同，Express还允许只在特定匹配到的路由中才使用中间件。

想象一下这样一个场景：你需要检查用户是否已经登录，并且这部分检查只在特定受保护的路由中进行。这个时候，就可以定义一个secure中间件，判断若req.session.logged_in不为true时就发送403 Not Authorized状态码。

```
function secure (req, res, next) {
  if (!req.session.logged_in) {
    return res.send(403);
  }

  next();
}
```

然后，将它应用到对应的路由中：

```
app.get('/home', function () {
  // accessible to everyone
});

app.get('/financials', secure, function () {
  // secure!
});

app.get('/about', function () {
  // accessible to everyone
```

```
});

app.get('/roadmap', secure, function () {
  // secure!
});
```

还可以像这样为路由定义多个中间件：

160

```
''app.post('/route', a, b, c, function () { });
```

有的时候，在中间件中调用next就可以跳过该路由的其他中间件，这样Express就会紧接着在下一个路由中做相应处理。

比如，若相比发送403，你更希望Express去检查其他的路由，那么就可以采用如下方式来实现：

```
function secure (req, res, next) {
  if (!req.session.logged_in) {
    return next('route');
  }

  next();
}
```

通过调用next('route')，就能确保当前路由会被跳过。

随着项目规模的扩大，路由和中间件的数量也会越来越多，因此，如何更好地组织代码就变得非常有用。下面我们就来介绍这部分的内容。

代码组织策略

对于任意一个Node.js应用（包括Express Web应用）来说，第一条准则都是模块化。Node.js通过提供一个简单的require API来提供一个强大的代码组织策略。

比如，一个应用包含三块独立的内容：/blog、/pages以及/tags。每块都包含各自的路由。例如：/blog/search、/pages/list以及tagst/cloud。

好的代码组织方式应当是维护一个server.js文件，该文件中包含了路由表，同时将每一部分的路由处理器都通过模块化的方式来引入，如blog.js、pages.js以及tags.js。首先，定义依赖的模块并初始化 app、引入中间件等：

```
var express = require('express')
  , blog = require('./blog')
  , pages = require('./pages')
  , tags = require('./tags')

// initialize app
```

```
var app = express.createServer();

// here you would include middleware, settings, etc
```

161 接着定义之前提到的路由表，这里简单地将所有的路由信息都罗列出来放在一个地方：

```
// blog routes
app.get('/blog', blog.home);
app.get('/blog/search', blog.search);
app.post('/blog/create', blog.create);

// pages routes
app.get('/pages', pages.home);
app.get('/pages/list', pages.list);

// tags routes

app.get('/tags', tags.home);
app.get('/tags/search', tags.search);
```

然后，针对每个模块需要使用exports函数。以blog.js为例：

```
exports.home = function (req, res, next) {
  // home
};

exports.search = function (req, res, next) {
  // search functionality
};
```

模块化提供了很强大的扩展性。上述代码还可以根据http方法进行进一步扩展。例如：

```
exports.get = {};
exports.get.home = function (req, res, next) {})
exports.post = {};
exports.post.create = function (req, res, next) {})
```

另外一种对应用进行解耦的方式叫做app挂载。你可以将整个Express app作为一个模块（模块中依然可以使用NPM模块），并将其挂载到现有的应用中，这样就能够让路由无缝对接起来。

考虑这样一个博客的应用。你可以将一个博客相关的路由都通过/来定义，/categories以及/search，然后将其以blog.js文件暴露出来：

```
var app = module.exports = express.createServer();
app.get('/', function (req, res, next) {});
app.get('/categories', function (req, res, next) {});
app.get('/search', function (req, res, next) {});
```

注意，上述路由都是通过绝对路径来定义的，不包含任何前缀。接着，在主程序中，要做的就是将其引入并传递给app.use方法：

```
app.use(require('./blog'));
```

162

这样一来，所有博客相关的路由就都可以使用了。除此之外，你还可以为它们定义一个前缀：

```
app.use ('/blog', require('./blog'));
```

现在，/blog/、/blog/categories以及/blog/search以后都可以直接给其他express应用使用，并且它自身拥有完全独立的依赖、中间件、配置，等等。

小结

本章介绍了如何使用最流行的Node.js Web框架Express。

使用Express最大的益处就在于，它简洁、配置少但却又不失灵活，同时它和Connect一样构建于测试全面、抽象简洁的基础之上。

和其他Web框架或者类库不同，Express非常容易进行模块化来满足不同的需求、结构以及模式。本章介绍了如何使用Express以最少的代码来构建首个示例应用，就像Node.js Hello World程序那样。

事实上，你或许已经注意到了，与重新在Node.js core API基础上构建一套新的方式不同，Express尝试与core API靠拢，并扩展它。这就是路由处理器可以直接接收原生的Node request和response对象的原因，就像我们在首个HTTP服务器应用中那样。本章介绍了一些Express实用的扩展，比如，如何使用res.send来以JSON的形式进行响应等。

本章最后介绍了如何将不同功能的代码组织起来创建一个可维护的应用。介绍了最好的方式就是通过Node.js核心的API：通过使用require就是其中一个最强大的工具，来进行更上层的代码组织。