第 6 章

在 Node.js 应用中使用 session 和 OAuth 进行用户认证和授权

近年来，Web 应用逐渐不再相互孤立，安全性也日益重要。作为开发者，我们不仅被鼓励使用市面上众多的第三方服务（如 Twitter、Github 等），也被希望作为服务商向外界提供服务（如提供 API 接口）。在这种情况下，我们需要使用某些手段来确保我们的应用以及应用间通信的安全，例如：基于 token 的用户认证、OAuth 授权协议[1]等。

所以这里我打算用一章的篇幅来详细介绍授权、认证、OAuth 以及最佳实践。具体而言，本章分为以下小节：

- 使用 Express.js 中间件实现权限管理
- 基于 token 的用户认证
- 基于 session 的用户认证
- 项目实践：为博客增加邮箱和密码登录功能用户认证
- Node.js OAuth 组件
- 项目实践：为博客增加 Twitter OAuth 1.0 第三方登录（使用 Everyauth[2]实现）

使用 Express.js 中间件权限管理

在 Web 应用中，“权限管理”指面向不同的用户（也指客户端）开放不同的页面（或接口）权限。

[1] http://oauth.net

[2] https://github.com/bnoguchi/everyauth

可以使用 Express.js 中间件实现复杂的规则设置，比如限制全部 URL、限制部分 URL、限制单个 URL 等：

- 全部 URL：`app.get('*', auth)`
- 部分 URL：`app.get('/api/*', auth)`
- 单个 URL：`app.get('/admin/users', auth)`

例如，如果我们需要限制整个/api/目录的访问，可以使用以下语句：

```
app.all('/api/*', auth);
app.get('/api/users', users.list);
app.post('/api/users', users.create);
...
```

或者这样：

```
app.get('/api/users', auth, users.list);
app.post('/api/users', auth, users.create);
...
```

在前面的例子中，`auth()`方法接收三个参数：`req`、`res` 和 `next`。类似这样：

```
var auth = function(req, res, next) {
//鉴定用户
//如果鉴定失败，则调用 next(new Error('Not authorized'));
//或者 res.send(401);
    return next();
}
```

切记，不要忘记调用 `next()` 函数，否则 Express.js 将无法进行后续的处理（包括调用其他回调、继续尝试匹配其他路由规则等）。

基于 token 的用户认证

在应用中，会为不同的用户赋予不同的权限（比如为管理员账户赋予较高的权限），所以我们需要在 `auto()` 函数中添加用户认证的流程。

一般来讲，最常见的方案是基于 cookie 或 session 授权管理，关于这一点我们将会在下一小节中详细介绍。但某些场景下这种方案并不适用，比如对要求使用 REST 架构的应用，或客户端对 cookie\session 支持不佳（如移动端）等。更有效的方案是在每次请求中都携带 token（比较常见的 OAuth2.0 协议[3]），并在服务端通过 token 进行独立的认证。这里既可以把 token 字段加载到请求参数中，也可以添加到 HTTP 请求头中。当然这里也可以是其他

[3] http://tools.ietf.org/html/rfc6749

认证信息，比如 E-mail 和密码、API 密钥、API 密码等。

在我们的示例中，每个请求都会提交 token 字段，并在接收时把 token（通过 `req.query.token` 获取）和应用中储存的 token（通常使用数据库储存，或如本例中简单地保存在 `SECRET_TOKEN` 常量中）进行比对。如果比对通过则调用 `next()` 方法继续后续处理，如果不通过则调用 `next(error)` 触发 Express.js 的错误响应：

```
var auth = function(req, res, next) {
  if (req.query.token && token === SECRET_TOKEN) {
  // 校验通过，进行下一阶段处理
    return next();
  } else {
    return next(new Error('Not authorized'));
    // 也可以 res.send(401);
  }
};
```

在实践中，一般使用 API 的 key 和 secret 生成 HMAC-SHA1（一种基于散列的信息加密算法）字符串，并把它和接收到的 token（`req.query.token`）进行比对。

■**注意** 在调用 `next()` 方法时传入一个 error 对象作为参数，表示放弃请求处理，这时会触发 Express.js 的错误模式，并进入错误处理流程。

我们刚才介绍了 REST API 中常用的基于 token 的认证模式。另外一种常见模式是使用 cookie 进行用户认证，这种模式在含有用户界面的应用中经常使用。我们使用 cookie 储存 session ID，并在请求时自动提交。从某种意义上讲，cookie 有些类似于 token，但是 cookie 使用较为方便，并不需要开发者做太多的工作。基于 session 的认证就是使用这种模式。基于 session 的认证在 Web 应用中十分常见，也更受推崇，因为浏览器可以自动处理带有 session 的请求头，而且大多数的后端平台或框架也能原生支持 session。接下来，就让我们一起进入在 Node.js 中实现基于 session 的用户认证这一小节吧。

基于 session 的用户认证

基于 session 的用户认证借助于请求体对象 `req` 中的 `session` 对象完成。简单地说，session 可以鉴别客户端，并对应地储存信息，供同一客户端所有的后续请求读取。

在 Express.js 4.x 版（4.1.2 版以及写本书时使用的 4.2.0 版）中，我们需要手动引入（`require()`）操作 session 所依赖的模块，因为 Express.js 4.x 把它们从核心包中剔除了。例如，引入并使用 `cookie-parser` 和 `express-session` 模块：

```
var cookieParser = require('cookie-parser');
var session = require('express-session');
...
app.use(cookieParser());
app.use(session());
```

当然，在进行这些操作之前，`cookie-parser` 模块和 `express-session` 模块需要通过 NPM 安装到项目的 `node_modules` 文件夹中。

如果是在经典的 Express.js 3.x 版本中，则需要在配置文件中加入下面两个中间件。

1. `express.cookieParser()`：解析发送的和接收的 cookie。
2. `express.session()`：在每个请求体中暴露 `res.session` 对象，并且在内存或持久化存储中（如 MongoDB、Redis 等）储存 session 数据。

在后文的例子中，如果没有特别提及 Express.js 的版本，就表示代码能兼容 3.x 和 4.x 版本。

啰唆一句，我们可以在 `req.session` 中储存任何数据，它们会自动出现在来自同一个客户端的所有后续请求中（在客户端支持 cookie 的前提下）。在这个例子中，认证信息用 session 储存的一个标记（布尔值），我们在授权函数中去检查这个标记，为真放行，为假则退出。像这样：

```
app.post('/login', function(req, res, next) {
  // 检查凭证
  // 在请求的有效负载中进行传递
  if (checkForCredentials(req)) {
    req.session.auth = true;
    res.redirect('/dashboard'); // 非公开内容
  } else {
    res.send(401); // 认证不通过
  }
});
```

■**警告** 避免在 cookie 中储存任何敏感信息。因为 cookie 十分不安全，而且储存长度存在限制（不同的浏览器限制不同，IE 最小）。所以推荐的方法是：不去手动操作 cookie，cookie 中只保留 session ID 字段，这个字段由 Express.js 中间件自动控制。

Express.js 默认使用内存来储存 session 数据，这就表示每次应用崩溃或手动重启时 session 数据都会丢失。我们可以使用 Redis 或者 MongoDB 储存 session 数据，这样既可以保证 session 数据能够持久化存储也可以实现 session 数据可跨服务器读取。

项目实践：为博客增加邮箱和密码登录功能

为了在博客中实现基于 session 的用户认证，我们需要完成以下步骤：

1. 在 `app.js` 的配置部分中增加引入和使用 session 中间件的代码。
2. 实现一个基于 session 的用户认证中间件，以便我们在多个路由规则之间复用这些代码。
3. 在 `app.js` 文件中添加上一步骤中的中间件，以控制非公开页面的访问。像这样：`app.get('/api/', authorize, api.index)`
4. 在 `user.js` 中实现包含认证过程的登录路由 `POST /login` 和登出路由 `GET /logout`。

session 中间件

我们需要在 `app.js` 中加入下面两行代码，用来解析 cookie 和提供对 session 的支持。

```
// 其他中间件配置
app.use(cookieParser('3CCC4ACD-6ED1-4844-9217-82131BDCB239'));
app.use(session({secret: '2C44774A-D649-4D44-9535-46E296EF984F'}));
// 路由
```

适用于 Express.js 3.x 版本的代码略有不同：

```
// 其他中间件配置
app.use(express.cookieParser('3CCC4ACD-6ED1-4844-9217-82131BDCB239'));
app.use(express.session({secret: '2C44774A-D649-4D44-9535-46E296EF984F'}));
// 路由
```

■**警告** 你应该用自己生成的随机字符串替换示例代码中的值。

注意，`cookieParser()` 需要在 `session()` 之前执行，因为 session 需要依赖 cookie 才能正常工作。如果想更深入地了解其他关于 Express.js/Connect 中间件的知识，可以参考 *Pro Express.js 4*（Apress，2014 年出版）这本书。

`cookie-session`（在 Express.js 3.x 中是 `express.cookieSession()`）有多种用法，一种是 `var cookieSession = require('cookie-session'); app.use(cookieSession({secret: process.env.SESSION_SECRET}));`，这种方式在浏览器 cookie 中只储存作为 session 键值的 session ID，并把 session 信息储存到内存或者 Redis 中。另一种方式是，在 cookie 中储存序列化后的整个 session 对象，当然，由于安全性和 cookie 长度限制等原因，这种方式非常不推荐使用。

为了把用户是否经过认证的信息传递给模板，这里我们实现了一个中间件，在判断 req.session.admin 为 true 时，会在 res.locals 中增加一个属性：

```
app.use(function(req, res, next) {
  if (req.session && req.session.admin)
    res.locals.admin = true;
  next();
});
```

博客中的权限管理

权限管理同样通过中间件完成，不过这次我们不再直接使用 app.use，而是定义一个检验函数，通过函数来检查 req.session.admin 标记的是否为真，为真则继续处理，为假则抛出 401 Not Authorized 的错误：

```
// 权限管理
var authorize = function(req, res, next) {
  if (req.session && req.session.admin)
    return next();
  else
    return res.send(401);
};
```

现在我们添加相关页面路由规则，并通过中间件来控制访问权限：

```
...
app.get('/admin', authorize, routes.article.admin);
app.get('/post', authorize, routes.article.post);
app.post('/post', authorize, routes.article.postArticle);
```

用同样的方法，添加 API 接口路由以及访问权限控制:

```
app.all('/api', authorize);
app.get('/api/articles', routes.article.list)
app.post('/api/articles', routes.article.add);
app.put('/api/articles/:id', routes.article.edit);
app.del('/api/articles/:id', routes.article.del);
```

app.all('/api', authorize);是为全部/api/...子路由添加用户认证的便捷方法。

添加 session 支持以及权限管理中间件后的 app.js 源代码如下（在 ch6/password 目录下）：

```
var express = require('express'),
  routes = require('./routes'),
  http = require('http'),
  path = require('path'),
```

```
  mongoskin = require('mongoskin'),
  dbUrl = process.env.MONGOHQ_URL ||
    'mongodb://@localhost:27017/blog',
  db = mongoskin.db(dbUrl, {safe: true}),
  collections = {
    articles: db.collection('articles'),
    users: db.collection('users')
  };

// 引入 Express.js 中间件
var session = require('express-session'),
  logger = require('morgan'),
  errorHandler = require('errorhandler'),
  cookieParser = require('cookie-parser'),
  bodyParser = require('body-parser'),
  methodOverride = require('method-override');

var app = express();
app.locals.appTitle = 'blog-express';

// 处理请求中的查询
app.use(function(req, res, next) {
  if (!collections.articles || ! collections.users)
    return next(new Error('No collections.'));
  req.collections = collections;
  return next();
});

// Express.js 配置
app.set('port', process.env.PORT || 3000);
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');

// Express.js 中间件配置
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded());
app.use(cookieParser('3CCC4ACD-6ED1-4844-9217-82131BDCB239'));
app.use(session({secret: '2C44774A-D649-4D44-9535-46E296EF984F'}));
app.use(methodOverride());
app.use(require('stylus').middleware(__dirname + '/public'));
app.use(express.static(path.join(__dirname, 'public')));
```

```
// 用户认证中间件
app.use(function(req, res, next) {
  if (req.session && req.session.admin)
    res.locals.admin = true;
  next();
});

// 权限管理
var authorize = function(req, res, next) {
  if (req.session && req.session.admin)
    return next();
  else
    return res.send(401);
};

if ('development' == app.get('env')) {
  app.use(errorHandler());
}

// 页面路由
app.get('/', routes.index);
app.get('/login', routes.user.login);
app.post('/login', routes.user.authenticate);
app.get('/logout', routes.user.logout);
app.get('/admin', authorize, routes.article.admin);
app.get('/post', authorize, routes.article.post);
app.post('/post', authorize, routes.article.postArticle);
app.get('/articles/:slug', routes.article.show);

// REST API 路由
app.all('/api', authorize);
app.get('/api/articles', routes.article.list);
app.post('/api/articles', routes.article.add);
app.put('/api/articles/:id', routes.article.edit);
app.del('/api/articles/:id', routes.article.del);

app.all('*', function(req, res) {
  res.send(404);
})

var server = http.createServer(app);
var boot = function () {
  server.listen(app.get('port'), function(){
    console.info('Express server listening on port ' +
```

```
        app.get('port'));
    });
}
var shutdown = function() {
    server.close();
}
if (require.main === module) {
    boot();
}
else {
    console.info('Running app as a module')
    exports.boot = boot;
    exports.shutdown = shutdown;
    exports.port = app.get('port');
}
```

博客中的用户授权

对基于 session 的权限控制来说，最后一步必然是允许用户或客户端控制 req.session.admin 的开关，也就是控制管理后台的登录状态。

在验证了用户的身份是管理员后，我们设置 admin=true，这一步操作放在 user.js 文件中的 routes.user.authenticate 方法中。验证身份的操作是通过在 app.js 中定义的 POST /login 路由实现的，像这样：app.post('/login', routes.user.authenticate);。

在 user.js 中，我们把方法暴露出来，给引用 user.js 模块的文件调用：

```
exports.authenticate = function(req, res, next) {
```

用户在注册页面上填写的表单会被提交到这个方法里。通常，我们需要对收到的信息进行校验，如果值不正确（包括为空），就会重新显示登录页面，并提示用户输入有误。return 语句确保了后续代码不会被执行。如果收到的值非空（或者正确），这个请求就不会被中止，而是进入下一个处理流程：

```
if (!req.body.email || !req.body.password)
    return res.render('login',{
        error: 'Please enter your email and password.'
    });
```

由于在 app.js 中引入了数据库中间件，所以这里我们可以简单地通过 req.collections 来访问数据库。在我们的应用架构中，E-mail 作为用户的唯一标识（不存在 E-mail 相同的两个账户），所以我们使用 findOne 函数来查找 E-mail 和密码相匹配的账户（二者为逻辑与关系）：

```
req.collections.users.findOne({
```

```
  email: req.body.email,
  password: req.body.password
}, function(error, user){
  ...
```

■**注意** 在几乎任何情况下，我们都不应该储存用户的明文密码，而是储存原始密码“加盐”（指通过在密码任意固定位置插入特定的字符串，让散列后的结果和使用原始密码的散列结果不相符）后的散列值。这样，即便以后数据库被泄露，用户的真实密码也不会被暴露。加密过程可以借助 Node.js 的核心模块 `crypto` 实现。

`findOne` 方法会返回一个错误对象（error）和查询结果对象（user）。我们需要手动进行错误处理：

```
if (error) return next(error);
if (!user) return res.render('login', {error: 'Incorrect email&password
    combination.'});
```

现在，情况比较明朗了（因为上面已经把错误的请求结束掉了），我们可以认为用户为管理员用户，而授予他管理员的权限：

```
    req.session.user = user;
    req.session.admin = user.admin;
    res.redirect('/admin');
  })
};
```

剩下注销操作就非常简单了，只需要在 `req.session` 上调用 `destroy()` 方法来清空 session 就可以了：

```
exports.logout = function(req, res, next) {
  req.session.destroy();
  res.redirect('/');
};
```

完整的 `user.js` 代码如下，供你参考：

```
exports.list = function(req, res){
  res.send('respond with a resource');
};
exports.login = function(req, res, next) {
  res.render('login');
};
exports.logout = function(req, res, next) {
  req.session.destroy();
  res.redirect('/');
```

```
};
exports.authenticate = function(req, res, next) {
  if (!req.body.email || !req.body.password)
    return res.render('login', {
      error: 'Please enter your email and password.'
    });
  req.collections.users.findOne({
    email: req.body.email,
    password: req.body.password
  }, function(error, user){
    if (error) return next(error);
    if (!user) return res.render('login', {
      error: 'Incorrect email&password combination.'
    });
    req.session.user = user;
    req.session.admin = user.admin;
    redds.redirect('/admin');
  })
};
```

运行应用

现在我们的博客已经一切准备就绪了。与第 5 章不同，现在非公开的页面只能在登录后才能访问。我们可以在这些页面中进行创建、发布或撤下文章等操作。但是，当单击菜单中的“登出”按钮后，我们将不再拥有访问这些页面的权限。可以执行的演示代码放在 practicalnode 仓库中的 `ch6/password` 目录下[4]。

Node.js OAuth

OAuth 模块是使用 Node.js 开发 OAuth 1.0/2.0 的利器。你可以在 NPM[5]或者 GitHub[6]上找到它。它可以帮我们计算签名、编码信息、生成 HTTP 头，最后发送请求。但是还有一些工作仍需要我们完成：发起 OAuth 握手（即在服务商、用户以及我们的应用之间的一系列请求）、添加回调路由、在 session 或数据库中储存信息等。我们可以参考服务商提供的文档，来获取关于接口、方法、参数等内容更详细的说明。

在处理复杂的场景或只有部分流程使用 OAuth 时（比如，需要使用 node-auth 生成请求头，但使用 superagent 库发送请求），推荐使用 `node-auth` 模块。

[4] https://github.com/azat-co/practicalnode

[5] https://www.npmjs.org/package/oauth

[6] https://github.com/ciaranj/node-oauth

只需运行下面的命令，便可以把 OAuth 模块 0.9.11 版（写本书时的最新版本）添加到你的项目中：

```
$ npm install oauth@0.9.11
```

使用 Node.js OAuth 实现 Twitter OAuth 2.0 的示例

OAuth 2.0 比 OAuth 1.0 使用起来更加简便（当然，肯定也有人不这么认为），但安全性略差一些。有许多原因导致了这种改变，如果希望了解其中缘由可以参看 OAuth 2.0 标准制定者之一 Eran Hammer 的一篇文章——*OAuth 2.0 and the Road to Hell*。

从本质上讲，OAuth 2.0 有些类似于我们之前讨论过的基于 token 的用户认证，之前提到的 token 在这里被叫作 bearer，在每一次请求中都会携带它。我们通过提供 app token 和 secret 来获取 bearer。

通常，bearer 的有效时间会比 OAuth 1.x 中的 token 长一些（当然，这取决于具体服务商的设置），并且它可以被作为唯一依据去鉴别用户是否有权访问非公开的资源。所以这里 bearer 扮演着基于 token 认证中 token 的角色。

这是 Node.js OAuth 模块文档[7]中提供的一个经典的示例。首先，我们创建一个 `oauth2` 对象并传入 Twitter 的 API key 和 secret（注意替换成你自己获取到的值）：

```
var OAuth = require('OAuth');
var OAuth2 = OAuth.OAuth2;
var twitterConsumerKey = 'your key';
var twitterConsumerSecret = 'your secret';
var oauth2 = new OAuth2(server.config.keys.twitter.consumerKey,
  twitterConsumerSecret,
  'https://api.twitter.com/',
  null,
  'oauth2/token',
  null
);
```

接下来，我们从服务商处获取到 token/bearer：

```
oauth2.getOAuthAccessToken(
  '',
  {'grant_type': 'client_credentials'},
  function (e, access_token, refresh_token, results){
  console.log('bearer: ', access_token);
  // 储存 bearer，后续的 OAuth2 请求会使用到它
  }
);
```

[7] https://github.com/ciaranj/node-oauth#oauth20

现在我们把获取到的 bearer 保存下来，后续的请求需要携带它才能通过接口的校验。

■**注意** Twitter 对应用专用接口使用 OAuth 2.0 协议认证授权，访问这些接口的请求必须由应用自身发出（而不是用户通过应用发出）。而对普通接口（用户通过应用访问的接口）Twitter 则使用了 OAuth 1.0。并非所有的应用专用接口都可用，每个接口都有不同访问配额。详情请参考 Twitter 官方文档[8]。

Everyauth

使用 Everyauth 模块只需要短短几行代码，就可以在任何基于 Express.js 的应用中实现 OAuth。它自带了市面上大部分第三方服务商的 OAuth 配置，包括接口地址、参数名称等，省去了我们查资料的麻烦。同时，Everyauth 会默认用户信息储存在 session 中，不过可以通过修改 `findOrCreate` 的回调函数，实现把用户信息存在数据库中。

■**提示** Everyauth 内置一套 E-mail 和密码策略供使用，可以参考 Everyauth 在 GitHub 仓库中的文档[9]。

Everyauth 内置配置的服务商列表如下（截至本书编写时，来源为该 GitHub[10]模块）：

- Password
- Facebook
- Twitter
- Google
- Google Hybrid
- LinkedIn
- Dropbox
- Tumblr
- Evernote
- GitHub
- Instagram
- Foursquare
- Yahoo!
- Justin.tv

[8] http://dev.twitter.com

[9] https://github.com/bnoguchi/everyauth#password-authentication

[10] https://github.com/bnoguchi/everyauth/blob/master/README.md

- Vimeo
- 37signals（Basecamp、Highrise、Backpack、Campfire）
- Readability
- AngelList
- Dwolla
- OpenStreetMap
- VKontakte（俄罗斯社交网站）
- Mail.ru（俄罗斯社交网站）
- Skyrock
- Gowalla
- TripIt
- 500px
- SoundCloud
- mixi
- Mailchimp
- Mendeley
- Stripe
- Datahero
- Salesforce
- Box.net
- OpenId
- LDAP（实验性质，未在正式产品中测试）
- Windows Azure Access Control Service

项目实践：为博客增加 Twitter OAuth 1.0 第三方登录（使用 Everyauth 实现）

标准的 OAuth 1.0 登录流程包含以下步骤（精简后）：

1. 用户访问一个地址，初始化 OAuth 流程。这时，我们的 APP 会通过 GET/POST 请求发送计算后的 API key 和 secret 去申请一个 token。例如，使用 Everyauth，`/auth/twitter` 会被自动添加。
2. 携带第一步中获取到的 token，把用户跳转至第三方服务商（Twitter）的页面，并等待回调。

3. 第三方服务商把用户跳转回我们提供的回调地址（如：`/auth/twitter/callback`）。然后应用可以从 Twitter 返回的响应中获取到 token、token 对应的 secret 以及用户信息。

不过，由于这一切均由 Everyauth 为我们代劳，我们并不需要手动实现请求以及回调接口路由。

现在让我们在页面上添加“使用 Twitter 账户登录”的按钮。我们需要准备：按钮（可以是图片，也可以是链接）、app key、secret[11]，接下来我们需要调整博客的授权管理部分，为通过 Twitter 登录的特殊用户授予管理员权限。

添加“使用 Twitter 账户登录”链接

在默认情况下，使用 Everyauth 接入第三方登录的链接格式为`/auth/:service_provider_name`。当然，这个格式可以修改。不过，本着 KISS（Keep It Short and Simple）的原则，我们不去动它，直接添加到 `menu.jade` 模板中：

```
li(class=(menu === 'login') ? 'active' : '')
  a(href='/auth/twitter') Sign in with Twitter
```

完整的 `menu.jade` 文件如下：

```
.menu
  ul.nav.nav-pills
    li(class=(menu === 'index') ? 'active' : '')
      a(href='/') Home
    if (admin)
      li(class=(menu === 'post') ? 'active' : '')
        a(href="/post") Post
      li(class=(menu === 'admin') ? 'active' : '')
        a(href="/admin") Admin
      li
        a(href="/logout") Log out
    else
      li(class=(menu === 'login')? 'active' : '')
        a(href='/login') Log in
      li
        a(href='/auth/twitter') Sign in with Twitter
```

配置 EveryauthTwitter 模块

要为博客添加 Everyauth 模块，请在命令行中输入下面的命令：

[11] 可以在 http://dev.twitter.com 上获取到

```
$ npm install everyauth@0.4.5 --save
```

我们需要在 app.js 中配置 Everyauth 的 Twitter 模块，但是对于一个大型应用来说，更好的做法是，用常量来记录这些配置，并保存在一个单独的文件中。需要注意的是，这些配置代码必须放在 app.route 方法调用之前。为了能更好地保护 consumer key 和 secret，这里把它们存在环境变量 process.env 中：

```
var TWITTER_CONSUMER_KEY = process.env.TWITTER_CONSUMER_KEY
var TWITTER_CONSUMER_SECRET = process.env.TWITTER_CONSUMER_SECRET
```

一种方案是在 Makefile 时把这些参数传入。在 Makefile 文件中加入下面几行，记得要把 ABC 和 XYZ 换成你自己的值：

```
start:
  TWITTER_CONSUMER_KEY=ABCABC \
  TWITTER_CONSUMER_SECRET=XYZXYZXYZ \
  node app
```

同时，也要把 start 命令加到.PHONY 中:

```
.PHONY: test db start
```

还有一种传值方案，创建一个脚本文件 start.sh：

```
TWITTER_CONSUMER_KEY=ABCABC \
TWITTER_CONSUMER_SECRET=XYZXYZXYZ \
node app
```

现在回过头来看 app.js，在其中加上引用 Everyauth 的语句：

```
everyauth = require('everyauth');
```

打开调试模式，在开发初期使用调试模式是一种好习惯：

```
everyauth.debug = true;
```

Everyauth 的子模块全部支持链式调用和 promise 协议，用下面的语句传入之前定义的 key 和 secret：

```
everyauth.twitter
  .consumerKey(TWITTER_CONSUMER_KEY)
  .consumerSecret(TWITTER_CONSUMER_SECRET)
```

接下来，添加 Twitter 返回响应时的回调函数：

```
.findOrCreateUser(function (session, accessToken, accessTokenSecret,
    twitterUserMetadata) {
```

我们本可以在这里直接返回用户对象，不过为了更真实地模拟写入数据库是异步过程，我们在这里创建了一个 promise 对象：

```
var promise = this.Promise();
```

然后使用 process.nextTick 函数（与 setTimeout(callback, 0);作用相似）来模拟一次异步请求。在真实的应用中，这里应该是读写数据库相关的语句：

```
process.nextTick(function(){
```

把 azat 替换成你自己的名字：

```
if (twitterUserMetadata.screen_name === 'azat_co') {
```

把 user 对象存在 session 中，和在/login 的路由中写的一样：

```
session.user = twitterUserMetadata;
```

最重要的一点，要把 admin 标记设成 true：

```
session.admin = true;
}
```

按 Everyauth 的规范，需要在最后返回 promise 对象：

```
    promise.fulfill(twitterUserMetadata);
  })
  return promise;
  // return twitterUserMetadata
})
```

在完成这些之后，配置认证用户身份后的跳转地址：

```
.redirectPath('/admin');
```

Everyauth 会自动添加一条/logout 路由，这样原本的登出路由(app.get('/logout',routes.user.logout);)就可以省略了。但是我们需要修改 Everyauth 的默认登出逻辑，在 handleLogout 这一步中调用 user.js 中提供的登出方法，否则 admin 标志会恒为 true：

```
everyauth.everymodule.handleLogout(routes.user.logout);
```

下面几行代码的作用是告诉 Everyauth 如何根据用户参数查找到用户对象，不过由于我们已经把用户对象储存到 session 中了，所以在这里可以直接返回：

```
everyauth.everymodule.findUserById( function (user, callback) {
  callback(user);
});
```

最后，需要添加下面一行代码来启用 Everyauth 路由规则，它必须添加在处理 cookie 和 session 的中间件之后，并且在其他的普通路由（如 app.get()、app.post()等）之前：

```
app.use(everyauth.middleware());
```

在添加了 Everyauth Twitter OAuth 1.0 模块之后，完整的 app.js 文件如下：

```
var TWITTER_CONSUMER_KEY = process.env.TWITTER_CONSUMER_KEY;
var TWITTER_CONSUMER_SECRET = process.env.TWITTER_CONSUMER_SECRET;
```

```
var express = require('express'),
  routes = require('./routes'),
  http = require('http'),
  path = require('path'),
  mongoskin = require('mongoskin'),
  dbUrl = process.env.MONGOHQ_URL ||
    'mongodb://@localhost:27017/blog',
  db = mongoskin.db(dbUrl, {safe: true}),
  collections = {
    articles: db.collection('articles'),
    users: db.collection('users')
  }
  everyauth = require('everyauth');

// Express.js 中间件
var session = require('express-session'),
  logger = require('morgan'),
  errorHandler = require('errorhandler'),
  cookieParser = require('cookie-parser'),
  bodyParser = require('body-parser'),
  methodOverride = require('method-override');

everyauth.debug = true;
everyauth.twitter
  .consumerKey(TWITTER_CONSUMER_KEY)
  .consumerSecret(TWITTER_CONSUMER_SECRET)
  .findOrCreateUser(function(
    session,
    accessToken,
    accessTokenSecret,
    twitterUserMetadata) {
      var promise = this.Promise();
      process.nextTick(function(){
        if (twitterUserMetadata.screen_name === 'azat_co') {
          session.user = twitterUserMetadata;
          session.admin = true;
        }
        promise.fulfill(twitterUserMetadata);
      })
      return promise;
    }
  )
  .redirectPath('/admin');

everyauth.everymodule.handleLogout(routes.user.logout);
```

```
everyauth.everymodule.findUserById(function (user, callback) {
  callback(user);
});

var app = express();
app.locals.appTitle = 'blog-express';

app.use(function(req, res, next) {
  if (!collections.articles || ! collections.users)
    return next(new Error('No collections.'));
  req.collections = collections;
  return next();
});

// Express.js 配置
app.set('port', process.env.PORT || 3000);
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');

// Express.js 中间件配置
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded());
app.use(cookieParser('3CCC4ACD-6ED1-4844-9217-82131BDCB239'));
app.use(session({secret: '2C44774A-D649-4D44-9535-46E296EF984F'}));
app.use(everyauth.middleware());
app.use(methodOverride());
app.use(require('stylus').middleware(__dirname + '/public'));
app.use(express.static(path.join(__dirname, 'public')));

// 用户认证中间件
app.use(function(req, res, next) {
  if (req.session && req.session.admin)
    res.locals.admin = true;
  next();
});

// 权限管理
var authorize = function(req, res, next) {
  if (req.session && req.session.admin)
    return next();
  else
    return res.send(401);
};
```

```
// 开发环境使用
if ('development' == app.get('env')) {
  app.use(errorHandler());
}

// 页面路由
app.get('/', routes.index);
app.get('/login', routes.user.login);
app.post('/login', routes.user.authenticate);
app.get('/logout', routes.user.logout);
app.get('/admin', authorize, routes.article.admin);
app.get('/post', authorize, routes.article.post);
app.post('/post', authorize, routes.article.postArticle);
app.get('/articles/:slug', routes.article.show);

// REST API 路由
app.all('/api', authorize);
app.get('/api/articles', routes.article.list);
app.post('/api/articles', routes.article.add);
app.put('/api/articles/:id', routes.article.edit);
app.del('/api/articles/:id', routes.article.del);
app.all('*', function(req, res) {
  res.send(404);
})

var server = http.createServer(app);
var boot = function () {
  server.listen(app.get('port'), function(){
    console.info('Express server listening on port ' +
      app.get('port'));
    }
  );
}
var shutdown = function() {
  server.close();
}
if (require.main === module) {
  boot();
} else {
  console.info('Running app as a module');
  exports.boot = boot;
  exports.shutdown = shutdown;
  exports.port = app.get('port');
}
```

不要忘了把配置中的 Twitter 用户名、密码、consumer key 和 secret 换成你自己的值。然后执行`$make start`就可以运行了。在你单击“使用 Twitter 账户登录”按钮时就会跳

转至 Twitter 的登录页面，认证完成后重新跳转回博客页面，这时你应该能看到管理员菜单。至此，我们已经完成了第三方用户认证。这时你也可以把用户信息保存到数据库中，供应用以后使用。在完成 Twitter 的授权后，页面跳转会很快完成。图 6-1 展示了整个 Everyauth 处理流程中每一步的输出内容，包括获取 token、返回响应等。当然，这里的每一步都可以根据应用的实际需要进行定制。

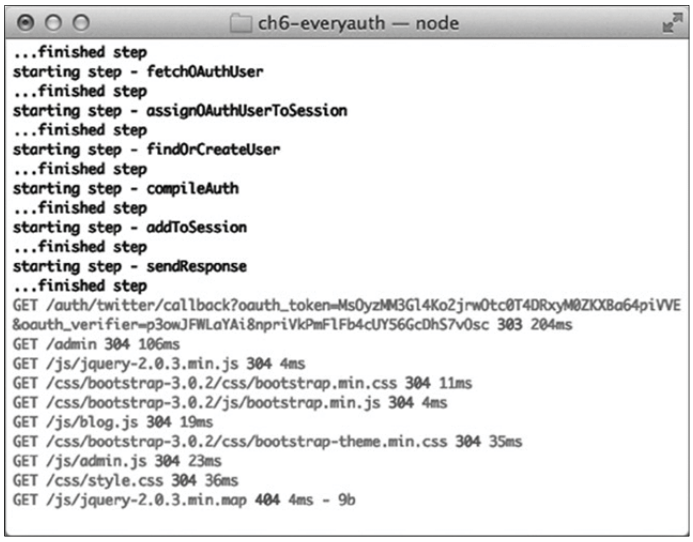

图 6-1　调试模式下通过 Everyauth 登录 Twitter 的过程

小结

在这一章中，我们学习了怎样实现一个由 E-mail 和密码组成的标准用户认证，以及在博客中怎样通过 Express.js 中间件限制非公开页面的访问。然后，我们又分别介绍了使用 Everyauth 和 OAuth 模块实现 OAuth 1.0 和 OAuth 2.0。

现在我们的博客已经有一些安全防护措施了。在下一章中我们将会探索 Node.js 中针对 MongoDB 的 ORM 库[12]——Mongoose[13]。Mongoose 对实体关系复杂的系统来说是一个绝佳的选择，它对数据库中的关系和数据进行了抽象和封装，通过它提供的工具方法就可以访问到全部数据。在下一章中，我们会详细介绍 Mongoose 类，解释它先进的设计理念并继续完善我们的博客。

[12] http://en.wikipedia.org/wiki/Object-relational_mapping

[13] http://mongoosejs.com