PART IV

数据库

CHAPTER 12 MongoDB

CHAPTER 13 MySQL

CHAPTER 14 Redis

CHAPTER

12 MongoDB

MongoDB是一个面向文档，schema无关（schema-less）的数据库，它非常适合于Node.js应用以及云端部署。

205

与MySQL及PostgreSQL是根据固定的结构设计（schema）将数据存储在表中不同，MongoDB可以将任意类型的文档数据存储到集合中（schema无关），这也是MongoDB最有意思的特性之一。

例如，创建下面这张为Web应用保存用户信息的表：

206

First	Last	Email	Twitter
Guillermo	Rauch	rauchg@gmail.com	rauchg

在构建应用时，决定将用户信息按照上面这样的结构设计进行存储。需要如下这些信息：first name、last name、email以及Twitter ID。

随着应用的发展、需求发生了改变，或者随着时间的推移，又有了新的需求，可能需要增加或者删除表中的某些列。

然而，这样一个基础性问题，若要通过传统的（SQL）数据库来实现，从操作上和性能上

来讲都需要耗费非常高的成本来修改表设计。

比如，在MySQL中，每一次修改表的设计结构，都需要运行如下这个命令才能实现添加一个新的列：

```
$ mysql
  > ALTER TABLE profiles ADD COLUMN . . .
```

对于删除一列或多列的情况也是如此。

在MongoDB中，则可以将数据都看作文档，其设计非常灵活。当有数据存储后，这些文档就会以一种非常接近（或者说在绝大多数情况下就是JSON格式）JSON格式的形式存储：

```
{
    "name": "Guillermo"
  , "last": "Rauch"
  , "email": "rauchg@gmail.com"
  , "age": 21
  , "twitter": "rauchg"
}
```

MongoDB还有一个非常重要的特性，能够将其与其他键—值形式的 NoSQL数据库区别开来，就是文档可以是任意深度的。

例如，可以将社交信息以如下结构进行存储，而不是全都将它们直接作为文档的键来存储：

```
{
    "name": "Guillermo"
  , "last": "Rauch"
  , "email": "rauchg@gmail.com"
  , "age": 21
  , "social_networks": {
        "twitter": "rauchg"
      , "facebook": "rauchg@gmail.com"
      , "linkedin": 27760647
    }
}
```

207

如上述代码所示，数据类型可以混用。这里，twitter和facebook信息都是字符串类型的，而linkedin是数字类型。当通过Node.js获取到存储的文档数据后，拿到的数据类型也是和存储时一模一样的。

本章将会介绍MongoDB最常用的功能，以及如何获得最灵活、最高效（通过索引）存储方案的最佳实践。本章还会介绍多种查询文档的方法，以及如何使用Mongoose来简化使用方式，Mongoose是我和 Nathan White一同书写的Node.js模块，它为MongoDB和JavaScript提供了传统关系型数据库ORM（Object-Relational Mapper）的部分功能。对MongoDB来说，这类项目

更贴切的叫法应该是ODM： Object Document Mapper。

安装

记住，本章使用的是MongoDB分支上最新的2.x版本。[1]

通过官网：www.mongodb.org/的下载页面就能获取到MongoDB。与此同时，你或许也会有兴趣看下各平台下MongoDB的快速指南： www.mongodb.org/display/DOCS/Quickstart。

通过执行mongo客户端，看到如图12-1所示的界面就表示安装成功了。

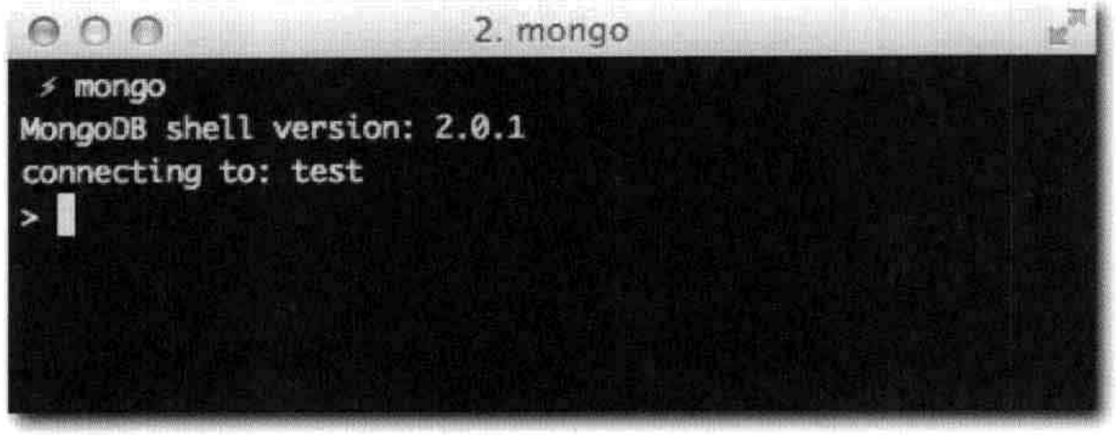


图12-1：MongoDB shell

要是无法连接到MongoDB，再检查下安装过程是否正确，同时通过进程管理器查看确保MongoDB服务器（mongod）正常运行。

使用MongoDB：一个用户认证的例子

208

通过Node.js操作MongoDB文档数据最主要的方式就是通过驱动器（driver）。通常在Node.js中驱动器指的就是一些基本的API，它懂得数据库网络层协议和其通信，并知道如何编码和解码存储的数据。

本例中选择的是由Christian Amor Kvalheim开发的node-mongodb-native。我们可以在Node包管理器（NPM）通过mongodb名字查找到这个驱动器。

第一个例子，我们创建一个简单的Express应用，实现将用户信息存储到MongoDB中，并实现用户注册、登录的功能。

构建应用程序

首先为项目创建package.json文件，声明项目所依赖的包。本例中，我们只需要express和mongodb。除此之外，我们还需要使用jade模板引擎：

```
{
    "name": "user-auth-example"
```

1 译者注：截止到本文翻译期间，MongoDB最新版本已经是2.4.6版本了。

```
  , "version": "0.0.1"
  , "dependencies": {
        "express": "2.5.8"
      , "mongodb": "0.9.9"
      , "jade": "0.20.3"
    }
}
```

创建Express App

首先，我们通过使用require引入所需的依赖包：

```
/**
 * 模块依赖
 */

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

由于本应用需要进行表单处理，所以，要用到bodyParser中间件。

因为我们要对用户进行认证，并将该信息保留下来，所以还需要用到 session中间件（它依赖Connect中的cookieParser中间件，在第8章中做过介绍）。

```
/**
 * 构建应用程序
 */
```

209

```
app = express.createServer()

/**
 * 中间件
 */

app.use(express.bodyParser());
app.use(express.cookieParser());
app.use(express.session({ secret: 'my secret' }));
```

因为本例中选用了jade模板引擎，所以还需要设置Express的view engine配置：

```
/**
 * 指定视图选项
 */

app.set('view engine', 'jade');

// 若使用了Express 3，则不需要下面这行代码


app.set('view options', { layout: false });
```

默认情况下，视图查找路径是views/。我们需要创建该目录，并在该目录下创建一个layout.jade文件来嵌入所有其他的视图：

```
doctype 5
html
  head
    title MongoDB example
  body
    h1 My first MongoDB app
    hr
    block body
```

尽管这并不在本章范畴，但是，作为Node.js应用中最流行的模板引擎之一，学习一些关于jade的内容还是很重要的。

- jade使用的是缩进（默认两个空格，应当避免使用tab），而不是复杂的嵌套XML、HTML标签。代码如下所示：

  ```
  p
     span Hello world
  ```

  等效于 Hello world 。

- 使用jade只需输入标签名，后面紧跟内容h1 My First MongoDB app即可，不需要这样完整地书写<h1>My first MongoDB app</h1>。
  这里使用doctype 5自动插入了HTML5的doctype。 210

- 代码中还使用了特殊的关键字block，这样其他视图文件就能嵌入到这个位置。这就是为什么要称该文件为layout的原因。其他还包括if和else这样特殊的关键字。

- 属性的写法看起来就像是HTML和JavaScript代码的混合体，并且非常容易嵌入变量（或者locals，express将从controller中暴露给视图层的变量称为locals）：

  ```
  a(href=#, another=attribute, dynamic=someVariable) My link
  ```

- 可以通过#{}这样的写法来嵌入变量：

  ```
  p Welcome back, #{user.name}
  ```

接着定义路由。我们需要一个主页的路由（/）、注册页面的路由（/signup）以及登录页面的路由（/login）：

```
/**
 * 默认路由
 */

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

```
/**
 * 登录路由
 */

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

/**
 * 注册路由
 */

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

在主页的路由中（/），我们传递了一个值为false的本地变量authenticated。等实现了登录功能后，我们会动态输出该变量。

我们在index模板中需要用到authenticated变量：

index.jade

```
extends layout
block body
if (authenticated)
  p Welcome back, #{me.first}
  a(href="/logout") Logout
else
  p Welcome new visitor!
  ul
    li: a(href="/login") Login
    li: a(href="/signup") Signup
```

211

如图12-2所示，注册和登录视图就是简单的表单视图：

signup.jade

```
extends layout
block body
form(action="/signup", method="POST")
  fieldset
    legend Sign up
    p
      label First
      input(name="user[first]", type="text")
    p
      label Last
```

```
    input(name="user[last]", type="text")
  p
    label Email
    input(name="user[email]", type="text")
  p
    label Password
    input(name="user[password]", type="password")
  p
    button Submit
  p
    a(href="/") Go back
```

login.jade

```
extends layout
block body
form(action="/login", method="POST")
  fieldset
    legend Log in
    p
      label Email
      input(name="user[email]", type="text")
    p
      label Password
      input(name="user[password]", type="password")
    p
      button Submit
    p
      a(href="/") Go back
```

图12-2：/signup路由

212

最后，我们需要让应用程序监听3000端口：

```
/**
 * 监听
 */

app.listen(3000);
```

要是通过浏览器访问，就能够很容易地访问到所有定义好的路由。

连接MongoDB

在查找文档（登录功能所需）和插入文档（注册功能所需）之前，先要连接MongoDB服务器并选择正确的数据库。

我们需要在服务器监听前就连接数据库。因为应用逻辑完全依赖数据库的操作，在还未能查询数据前就接收请求显然是不行的。

由于我们直接使用了MongoDB的驱动器，所以API有些长。不过，我们的目的是要暴露像app.users这样MongoDB集合的API，方便在路由定义中轻松对数据库进行操作。

首先通过创建一个mongodb.Server并提供IP和端口来初始化服务器：

213

```
/**
 * 连接数据库
 */

var server = new mongodb.Server('127.0.0.1', 27017)
```

接着告诉驱动器去连接数据库。比如，取一个叫my-website的数据库名字。在MongoDB中，要是指定的名字不存在，就会创建一个数据库。

```
new mongodb.Db('my-website', server).open(function (err, client) {
```

要是连接数据库失败，我们就得终止进程：

```
// 在有错误的情况下不允许应用程序启动
if (err) throw err;
```

要是连接成功则打印成功的消息：

```
console.log('\033[96m  + \033[39m connected to mongodb');
```

接着，建立集合：

```
// 建立集合快捷方式
app.users = new mongodb.Collection(client, 'users');
```

最后，让Express服务器监听端口准备操作集合：

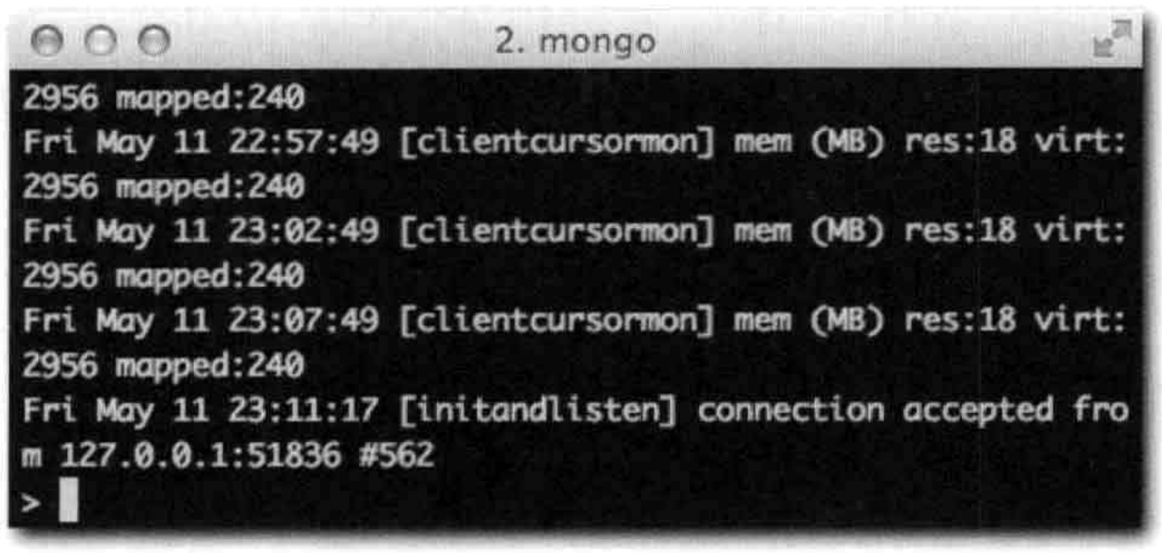
```
  // 监听
  app.listen(3000, function () {
    console.log('\033[96m  + \033[39m app listening on *:3000');
  });
});
```

如果运行应用程序（确保数据库运行正常），大致会输出如下形式的消息：

```
$ node server.js
  +  connected to mongodb
  +  app listening on *:3000
```

要是这个时候通过mongo客户端，运行show log global命令，应当会看到如图12-3所示的连接成功的消息！

```
$ mongo
> show log global;
[. . .]
{date} [initandlisten] connection accepted from 127.0.0.1:53649 #16
```



214

图12-3：如最后一行日志所示，Mongo从你的本地Web服务器获取了连接请求

创建文档

插入文档的API很简单。简单地调用Collection#insert方法，并提供要插入的文档以及一个回调函数就可以了。和绝大多数Node中的回调函数一样，第一个参数是一个错误对象，本例中，第二个参数是一个插入的文档数组：

```
collection.insert({ my: 'document' }, function (err, docs) {
  // . . .
});
```

另外还有一个options对象，是可选的第二个参数，后面会做相应介绍。

要是回头再来看注册表单，就会发现输入框的名字遵循这样的格式：user[field]。例如：

```
input(name="user[name]", type="text")
```

下面会看到，当bodyParser遇到这样的格式，会产生req.body.user.name这样的字段。

这个功能可以很方便地让我们直接将文档插入到MongoDB中。对于本例，我们忽略数据校验（非常重要）。

这样一来，处理注册的路由就变得非常简单：

```
/**
 * 处理注册的路由
 */

app.post('/signup', function (req, res, next) {
  app.users.insert(req.body.user, function (err, doc) {
    if (err) return next(err);
    res.redirect('/login/' + doc[0].email);
  });
});
```

215 若遇到错误，需要调用next，这样就可以显示一个“错误500”页面。尽管错误不会频繁发生，但重要的是必须要对其进行处理。

在处理了错误之后，最常犯的错误就是忘记return。这种错误会在应用程序中产生无法预知的行为。比如，这个时候有错误发生了，doc变量就会是undefined，这样一来代码就会抛出无法捕获的异常。

插入文档成功后，将应用程序重定向到登录路由，并提供email字段。

在登录路由中，我们获取email参数，并将其暴露给视图：

```
/**
 * 登录路由
 */

app.get('/login/:signupEmail', function (req, res) {
  res.render('login', { signupEmail: req.params.signupEmail });
});
```

在视图中，我们显示一个消息：

```
if (signupEmail)
  p Congratulations on signing up! Please login below.
```

然后，将email变量输出到email输入框中：

```
input(name="user[email]", type="text", value=signupEmail)
```

现在启动应用程序验证一下注册功能！这时，若启动mongo客户端，在新创建的集合上运行find命令，应该就能看到刚刚创建的文档内容了：

```
 Brian: in the mongo client things appear like this$ mongo my-website
> db.users.find()
{ "first" : "A", "last" : "B", "email" : "a@b.com", "password" : "d", "_id" : Object
 Id("4ef2cbd77bb50163a7000001") }
```

注意，上述文档看起来和你插入的完全一样，相当直观。除此之外， Mongo自动添加了_id字段，用来唯一确定文档内容。非常方便!

查找文档

至此我们已经创建好了文档，接下来就可以在/login路由中对文档进行查询了。我们要获取的是匹配对应email和password的文档。

在MongoDB中，没有固定的schema能确定一个集合，因此，每次对集合进行特定方式查询时，确保正确地对其进行了索引会是个不错的主意。特别是当某个键是在嵌套结构中时，要是没有对其进行索引，会导致一次扫描整表的查询操作，会导致应用程序性能的下降。 216

MongoDB有一个ensureIndex命令，顾名思义，不管索引是否存在，都可以调用这个命令来确保在查询前建立了索引。我们可以在应用初始化的时候调用它。

在设置了app.users后，我们应当调用两次ensureIndex：

```
client.ensureIndex('users', 'email', function (err) {
  if (err) throw err;
  client.ensureIndex('users', 'password', function (err) {
    if (err) throw err;

    console.log('\033[96m  + \033[39m ensured indexes');

    // 监听
    app.listen(3000, function () {
      console.log('\033[96m  + \033[39m app listening on *:3000');
    });
  });
});
```

重启应用后，会发现有额外日志：

```
$ node server.js
+  connected to mongodb
+  ensured indexes
+  app listening on :3000
```

现在就可以查询了!

```
/**
 * 登录处理路由
 */

app.post('/login', function (req, res) {
  app.users.findOne({ email: req.body.user.email, password: req.body.user.password
 }, function (err, doc) {
    if (err) return next(err);
    if (!doc) return res.send('<p>User not found. Go back and try again</p>');
```

```
      req.session.loggedIn = doc._id.toString();
      res.redirect('/');
    });
});
```

和insert命令一样，findOne命令可以对MongoDB进行查询文档的操作。

217 我们将_id存储为session的一部分，这样可以在用户访问其他路由时，能够获取当前登录用户的信息。注意这里，我们显式地将MongoDB ObjectId存储为字符串类型，其表现形式是十六进制的。

最后，还要实现/logout路由，处理非常简单，清除session就好了。记住，req.session对象可以随意修改，在做出响应后（比如本例中的重定向），Express会自动将其保存下来。

```
/**
 * 登出路由
 */

app.get('/logout', function (req, res) {
  req.session.loggedIn = null;
  res.redirect('/');
});
```

在这个例子中，我们保留了session，并将其ID设置成了null。或者，若想完全清楚session，可以直接调用req.session.regenerate()。

身份验证中间件

我们开发的绝大多数应用貌似在不止一处都需要访问验证后登录用户的信息。

若回过头来再看一下index.jade，我们需要访问me对象来获取匹配登录用户的文档信息，与此同时，还要通过authenticated变量来判断用户是否已经通过了身份验证。

```
if (authenticated)
  p Welcome back, #{me.name}
  a(href="/logout") Logout
```

我们可以定义一个中间件，将这两个变量（authenticated、me）暴露给视图使用。这里需要用到Express的res.local API：

```
/**
 * 身份验证中间件
 */

app.use(function (req, res, next) {
  if (req.session.loggedIn) {
    res.local('authenticated', true);
    app.users.findOne({ _id: { $oid: req.session.loggedIn } }, function (err, doc) {
      if (err) return next(err);
      res.local('me', doc);
```

```
      next();
    });
  } else {
    res.local('authenticated', false);
    next();
  }
});
```

218

注意了，在调用findOne时，我们传递了$oid修饰符。它允许直接传递一个字符串而不用传递一个ObjectId对象。回忆一下，之前我们调用了toString来确保存储的loggedIn是字符串。

记得移除此前在index路由中用来测试的{authenticated:false}，现在这部分代码应该是这样的（见图12-4）：

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

图12-4：用户成功登录后的界面

此前的应用没有考虑实际场景下必要的一些基础特性。接下来的三部分内容会对其进行介绍。

校验

若用户提交的表单太大，该怎么办呢？按照此前例子的处理方式，就会直接向数据库中插入这么大的文档数据了。

除此之外，我们也许还应该在存储数据前确保email字段确实是email字段，密码应该是不少于6个字符的字符串，而不是Date或者是Number。

我们也不想在每次对数据库进行插入、更新、查询操作的时候都要重复上述校验规则。

Mongoose通过允许在应用层定义schema来解决这个问题，它在保持文档灵活性和易改动的前提下，引入了特定的属性对其做一定的约束，称为模型。

219

原子性

假设我们基于Express和MongoDB书写一个博客引擎。可想而知，其中一部分功能会是允许用户修改博文的标题和内容，可能还有一部分功能是允许编辑和删除标签。

面向文档设计的MongoDB非常适合这样的场景。在posts集合中，文档可能会是这样的：

```
{
    "title": "I just bought Smashing Node.JS"
  , "author": "John Ward"
  , "content": "I went to the bookstore and picked up. . ."
  , "tags": ["node.js", "learning", "book"]
}
```

假设，这个时候，有两个人，用户A想要编辑文档的标题，与此同时，用户B想要添加一个标签。

如果两个用户都传递了一份完整的文档拷贝来进行更新操作，那么只有一个会胜出。另外一个将无法成功完成对文档的修改。

要确保某个操作的原子性，MongoDB提供了$set和$push这样不同的操作符：

```
db.blogposts.update({ _id: <id> }, { $set: { title: 'My title' } })
db.blogposts.update({ _id: <id> }, { tags: { $push: "new tag" })
```

Mongoose则是通过检查要对文档做的修改，并只修改受影响的字段来解决这个问题。就算操作的是数组（包括文档数组），原子性依然能够得到保证。

安全模式

此前介绍过，在使用驱动器时，我们可以在操作文档时提供一个可选的options参数：

```
app.users.insert({ }, { <options> })
```

其中一个选项叫safe，它会在对数据库进行修改时启动安全模式。

220 默认情况下，在操作完成后，如果有错误发生，MongoDB不会及时通知你。驱动器需要在操作完成后进行一个特殊的函数调用db.getLastError，来验证数据修改是否成功。

这背后的原因在于对于许多应用来说，相比于要知道某个操作是否失败而言，速度更为重要。比如，丢了某些日志并不是什么世界末日，但是导致性能低下就无法接受了。

Mongoose默认会对所有操作启用安全模式，当然了，你可以关闭这个选项。

Mongoose介绍

按照惯例，在开始使用Mongoose前，先要在package.json文件中定义对Mongoose的依赖，

然后通过require将其引入：

```
var mongoose = require('mongoose')
```

相比原生的驱动器，Mongoose做的第一个简化的事情就是它假定绝大部分的应用程序都是用一个数据库，这大大简化了使用方式。要连接数据库，只需要调用mongoose.connect并提供mongodb://URI即可：

```
mongoose.connect('mongodb://localhost/my_database');
```

另外，使用Mongoose，就无须关心连接是否真的已经建立了，因为，它会先把数据库操作指令缓存起来，在连接上数据库后就会把这些操作发送给MongoDB。这就意味着，我们无须监听connection的回调函数。连接后就可以直接像下面要介绍的这样开始查询数据了。

定义模型

模型是Schema类的简单实例。在指定字段时，简单地使用对应类型的JavaScript原生的构造器即可：

```
var Schema = mongoose.Schema
  , ObjectId = Schema.ObjectId;

var PostSchema = new Schema({
    author    : ObjectId
  , title     : String
  , body      : String
  , date      : Date
});
```

这些类型是：

- Date
- String
- Number
- Array
- Object

221

除此之外，MongoDB还提供了一种ObjectId类型，这种类型可以通过Schema.ObjectId来获得。

在这个博客例子中，我们可以将创建博文的用户存储为ObjectId类型。

Mongoose还为给定的字段提供了一些不同的选项。在提供选项时，需要在一个对象中引用对应字段类型的构造器。比如，要给字段提供了一个默认值，可以按照如下方式提供

default和type选项：

```
var PostSchema = new Schema({
    author     : ObjectId
  , title      : { type: String, default: 'Untitled' }
  , body       : String
  , date       : Date
});
```

创建好Schema后，通过mongoose来注册一个模型：

```
var Post = mongoose.model('BlogPost', PostSchema);
```

对于本例，Mongoose将集合名字设置为blogposts，除非我们通过第三个参数来指定集合名。Mongoose默认会对模型名字使用小写复数形式。

随后要想获取模型，可以通过调用mongoose.model方法并提供模型名：

```
var Post = mongoose.model('BlogPost');
```

接着就可以操作模型了。要创建一个博客文章，只要使用new操作符就可以了：

```
new Post({ title: 'My title' }).save(function (err) {
  console.log('that was easy!');
});
```

有一点很重要：Schema只是一种简单的抽象，用以描述模型的样子以及它是如何工作的。数据的交互发生在模型上，而不是Schema上。

因此，若要查询，与使用new关键字来初始化博文相对，需要执行静态的Post.find方法（或者其他一些会在下面介绍的方法）。

222 定义嵌套的键

考虑到数据的组织，有的时候以子结构的形式来组织键也非常有帮助：

```
var BlogPost = new Schema({
    author     : ObjectId
  , title      : String
  , body       : String
  , meta       : {
      votes : Number
    , favs  : Number
  }
});
```

在MongoDB中，可以使用点来操作这些属性。比如，要查找拥有指定投票数的博文，可以通过如下方式来实现：

```
db.blogposts.find({ 'meta.votes': 5 })
```

定义嵌套文档

在MongoDB中，文档可以很大，也可以层次很深。也就是说，如果博文有留言的话，可以直接将留言定义在博文中，而不需要将其定义为单独的集合。

```
var Comments = new Schema({
    title     : String
  , body      : String
  , date      : Date
});

var BlogPost = new Schema({
    author    : ObjectId
  , title     : String
  , body      : String
  , buf       : Buffer
  , date      : Date
  , comments  : [Comments]
  , meta      : {
      votes : Number
    , favs  : Number
  }
});
```

Mongoose还允许为该字段定义想要的类型。

构建索引

正如此前提到过的，索引是在MongoDB数据库中确保快速查询的重要因素。

要对指定的键做索引，需要传递一个index选项，并将值设置为true。

比如，要对title键做索引，并将uid键设置为唯一，可以这样：

```
var BlogPost = new Schema({
    author    : ObjectId
  , title     : { type: String, index: true }
  , uid       : { type: Number, unique: true }
});
```

要设置更复杂的索引（如组合索引），可以使用静态的index方法：

```
BlogPost.index({ key: -1, otherKey: 1 });
```

中间件

在相当一部分应用中，有的时候会在不同的地方以不同的方式对同样的数据进行修改。

通过模型接口将这部分对数据库的交互集中处理是避免代码重复很有效的方式。

Mongoose通过引入中间件来实现。Mongoose中间件的工作方式和 Express中间件非常相似。你可以定义一些方法，在某些特定动作前执行：save和remove。

比如，要在博文删除时，发送电子邮件给作者，可以通过下面的方式来实现：

```
Blogpost.pre('remove', function (next) {
  emailAuthor(this.email, 'Blog post removed!');
  next();
});
```

还可以对单个动作定义多次中间件来执行各类操作，特别是异步操作。

探测模型状态

很多时候，我们需要根据要对当前模型做的不同更改进行不同的操作：

```
Blogpost.pre('save', function (next) {
  if (this.isNew) {
    // doSomething
  } else {
    // doSomethingElse
  }
});
```

224

还可以通过this.dirtyPaths来探测什么键被修改了。

查询

在Model实例上暴露的所有常见的操作有：

- find
- findOne
- remove
- update
- count

Mongoose还添加了findById，该方法接受一个ObjectId去匹配文档的_id属性。

扩展查询

如果对某个查询不提供回调函数，那么直到调用run它才会执行：

```
Post.find({ author: '4ef2cbffb1d9807fa7000001' })
  .where('title', 'My title')
  .sort('content', -1)
  .limit(5)
  .run(function(err, post) {
    // . . .
  })
```

排序

要进行排序，只需提供排序的键以及排序的顺序即可：

```
query.sort('key', 1)
query.sort('some.key', -1)
```

选择

若文档很大，而想要的只是部分指定的键，那么就可以调用Query#select方法。

比如，要显示带链接的博文列表，无须获取所有的字段（有些字段可能数据量很大）：

```
Post.find()
  .select('field', 'field2')
```


限制

要限制查询结果的数量，可以调用Query#limit方法：

```
query.limit(5)
```

跳过

要跳过指定数量的文档数据，可以通过如下方式：

```
query.skip(10);
```

这个功能结合Model#count对做分页非常有用：

```
Post.count(function (err, totalPosts) {
  var numPages = Math.ceil(totalPosts / 10);
});
```

自动产生键

在BlogPost模型例子中，我们将博文作者的ID存储为author属性。

很多时候，在查询一个博文时，我们还需要获取对应的作者。这个时候，就可以为ObjectId类型提供一个ref属性：

```
var BlogPost = new Schema({
    author    : { type: ObjectId, ref: 'Author' }
  , title     : String
  , body      : String
  , meta      : {
      votes : Number
    , favs  : Number
  }
});
```

之后，查询文档时就能自动产生作者数据！通过简单地对指定键调用populate方法即可：

```
BlogPost.find({ title: 'My title' })
  .populate('author')
  .run(function (err, doc) {
    console.log(doc.author.email);
  })
```

转换

因为Mongoose提前知道需要什么样的数据类型，所以它总是会尝试去做类型转换。

226

例如，有一个年龄字段，在Schema中描述的是Number类型。如果有人在网站中提交了一个普通表单，那么在没有JSON或者特定逻辑处理的情况下，我们得到的是字符串而不是数字。 Mongoose利用了动态语言的特性，所以对它来说在存储前将'21'（字符串）转化为21（数字）就没有什么问题了。

同样的情况还会发生在ObjectId。在此前的例子中，我们不得不使用$oid修饰符来让使用字符串形式ObjectID的查询操作成功执行，然而，这样的方式过于冗长。我们可以直接传递"4ef2cbd77bb50163a7000001"给Mongoose，它会自动将其转化为ObjectId("4ef2cbd77bb50163a7000001")。

除此之外，当类型不匹配并且转换失败时，Mongoose会抛出一个校验错误，并且放弃对文档的存储。这种行为在一致性和文档的整洁性上确保了易用性。

一个使用Mongoose的例子

和此前我们做的一样：先使用node http模块，然后再使用Connect和 Express来对其改进。接下来，我们要用Mongoose对此前的例子进行重构，以此来展现Mongoose在表现形式上对数据表提供的好处。

我们从创建一个新的package.json并添加对mongoose的依赖开始。

构建应用

新的package.json文件如下所示：

```
{
    "name": "mongoose-example"
  , "version": "0.0.1"
  , "dependencies": {
        "express": "2.5.2"
      , "mongoose": "2.5.10"
    }
}
```

和往常一样，运行npm install来安装所需的依赖。接下来，我们重构一下服务器端主要的代码。

重构

我们从此前例子中将server.js文件和views目录复制过来。

首先要重构的就是将对mongodb的依赖替换为mongoose，因为mongodb并未出现在package.json文件中。事实上，mongoose内部就是使用了mongodb。

server.js顶部代码如下所示：

227

```
/**
 * 模块依赖
 */

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

现在，我们要将注意力放在该文件底部代码上，也就是连接数据库的代码：

```
/**
 * 连接数据库
 */

var server = new mongodb.Server('127.0.0.1', 27017)
// . . .
```

正如此前提到的，mongoose大大简化了连接数据库、操作集合、建立索引等的操作。server.js文件最后一部分代码就可以简化为如下形式：

```
/**
 * 连接数据库
 */

mongoose.connect('mongodb://127.0.0.1/my-website');
app.listen(3000, function () {
  console.log('\033[96m  + \033[39m app listening on *:3000');
});
```

接下来，我们要定义模型来取代app.users这种引用方式，并建立此前建立的索引。

建立模型

模型可以在文件任意位置通过Mongoose定义。无所谓Mongoose是否已经与数据库连接。

在文件最后，我们加上对模型的定义：

```
/**
 * 定义模型
```

```
 */

var Schema = mongoose.Schema

var User = mongoose.model('User', new Schema({
    first: String
  , last: String
  , email: { type: String, unique: true }
  , password: { type: String, index: true }
}));
```

228

接下来我们再看用它来替换app.users的地方。第一处就是身份认证中间件。我们用Mongoose提供的便利的findById方法来替换使用$oid的地方：

```
app.use(function (req, res, next) {
  if (req.session.loggedIn) {
    res.local('authenticated', true);
    User.findById(req.session.loggedIn, function (err, doc) {
      if (err) return next(err);
      res.local('me', doc);
      next();
    });
  } else {
    res.local('authenticated', false);
    next();
  }
});
```

在登录的POST路由中，需要再次用到模型方法。这次是findOne：

```
app.post('/login', function (req, res) {
  User.findOne({ email: req.body.user.email, password: req.body.user.password },
function (err, doc) {
    if (err) return next(err);
    if (!doc) return res.send('<p>User not found. Go back and try again');
    req.session.loggedIn = doc._id.toString();
    res.redirect('/');
  });
});
```

正如此前提到过的，模型可以静态方式使用（正如那两个例子所示的），也可以当构造器使用。

注册的POST路由应该重构成如下形式：

```
app.post('/signup', function (req, res, next) {
  var user = new User(req.body.user).save(function (err) {
    if (err) return next(err);
    res.redirect('/login/' + user.email);
  });
});
```

注意，我们不再需要一个包含了文档信息的回调函数。我们只需在回调函数中使用创建的模型实例即可（使用user.email而不是doc[0].email）。 229

重构完成！再次运行server.js，一切都应该运行正常，不过，代码变得更为整洁，更容易理解了。

小结

本章介绍了Node.js世界中最流行的数据库之一：MongoDB。

本章介绍了关于文档数据库是如何工作的基本知识，还介绍了Node.js中 MongoDB的驱动器。

你会注意到，它处理数据的方式非常自然，另外，对数据在浏览器和Web服务器之间传递的映射也非常好。

通过重构第一个例子，现在，你应当能够体会到本章介绍的引入了模型概念以及非常方便的API框架——Mongoose的好处了。