CHAPTER

13 MySQL

尽管现如今NoSQL渐行其道，越来越流行，但是SQL数据库依旧是绝大多数应用的选择。

Node.js丰富的生态系统中，有许多模块都是为SQL数据库设计开发的，特别是本章要介绍的：MySQL。

与第12章中介绍MongoDB一样，本章会先介绍一个原生的MySQL驱动器（一个名为node-mysql的项目）。

通过node-mysql，我们可以书写自己的SQL查询语句来操作数据库。

除了驱动器之外，本章还会介绍如何使用MySQL的对象关系映射器（ORM）——node-sequelize。正如上一章介绍的，ORM提供了一个MySQL数据库中数据到JavaScript模型对象的映射，使得操作数据关系、数据处理等变得更加容易。

node-mysql

要学习如何使用node-mysql，我们从创建一个简单的购物车应用的模型开始。

初始化项目

按照惯例，我们从添加对`express`、`jade`还有`node-mysql`的依赖开始：

package.json

```
{
    "name": "shopping-cart-example"
  , "version": "0.0.1"
  , "dependencies": {
        "express": "2.5.2"
      , "jade": "0.19.0"
      , "mysql": "0.9.5"
    }
}
```

Express应用

接下来，我们创建一个简单的Express应用，并添加如下路由：

- /：展示所有的商品以及创建商品的表单。
- /item/<id>：展示指定的商品以及用户评价。
- /item//review (POST)：创建一个评价。
- /item/create (POST)：创建一个商品。

对于首页以及商品的路由，我们将渲染简单的模板。注意了，我们配置了Express的view options，将模板布局取消了，这符合Express 3的行为。模板布局将直接通过jade来实现。

server.js

```
/**
 * 模块依赖
 */

var express = require('express')

/**
 * 创建应用
 */

app = express.createServer();

/**
 * 配置应用
 */

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

```
/**
 * 首页路由
 */

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

/**
 * 创建商品路由
 */

app.post('/create', function (req, res, next) {
});

/**
 * 查看商品路由
 */

app.get('/item/:id', function (req, res, next) {
  res.render('item');
});

/**
 * 创建商品评价路由
 */

app.post('/item/:id/review', function (req, res, next) {
});

/**
 * 监听
 */

app.listen(3000, function () {
  console.log(' - listening on http://*:3000');
});
```

连接MySQL

234

下一步是添加对node-mysql的依赖：

server.js

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

要初始化连接，和创建net客户端所使用的Node API方式一样，我们通过调用createClient

方法来实现。

和Mongoose对mongodb的处理方式一样，node-mysql在真正连接到MySQL前就可以接收指令，并将它们缓存起来（也就是将它们存储在内存中），当连接建立后，就一次性将它们全部发送给MySQL。

所以，我们无须监听connection事件或者提供回调函数，只需简单地初始化客户端，提供相应的设置即可。将下列代码添加到应用配置部分的下方：

server.js

```
/**
 * 连接MySQL
 */

var db = mysql.createClient({
    host: 'localhost'
  , database: 'cart-example'
});
```

若要设置数据库的用户名和密码，就在调用createClient方法时在参数中传递user和password选项。要了解更多关于node-mysql的用法，可以查看其官方文档http://github.com/felixge/node-mysql。

初始化脚本

在应用程序中使用SQL数据库前，我们总是先要创建必要的数据库和表。

为了让代码重用，我们创建一个名为setup.js的简单node脚本来运行必要的CREATE TABLE命令。

由于连接数据库所需参数和此前书写在应用中的是一样的，所以我们先将这些参数的配置抽象到一个config.json文件中：

235

config.json

```
{
    "host": "localhost"
  , "database": "cart-example"
}
```

注意了，有效的JavaScript代码未必一定是有效的JSON。本例中，我们将所有的键都用引号括起来，并且要确保所有的值都使用的是双引号而不是单引号。

从Node 0.6开始，就可以直接使用require来引入JSON文件，而无须再用JSON.parse和

fs#readFileSync了。接下来，编辑修改依赖的模块：

server.js

```
/**
 * 模块依赖
 */

var express = require('express')
  , mysql = require('mysql')
  , config = require('./config')

// . . .
```

在连接数据库的代码部分，使用config来替换原先的参数对象：

server.js

```
var db = mysql.createClient(config);
```

接下来准备创建启动脚本。该脚本只依赖mysql和config，因为它是从命令行直接运行的。

setup.js

```
/**
 * 模块依赖
 */

var mysql = require('mysql')
  , config = require('./config')
```

下面初始化客户端，由于数据库还未创建好，所以需要将config中的database字段删除：

236

setup.js

```
/**
 * 初始化客户端
 */
delete config.database;
var db = mysql.createClient(config);
```

node-mysql提供的执行查询语句的API非常简单：client.query(<sql>, <callback>)。关闭连接的API是client.end。

由于我们使用单个TCP连接，所以服务器接收到的指令顺序和我们书写的顺序是一致的。也就意味着，不需要为了确保执行顺序而嵌套回调函数：

```
// 没有必要这么做!
db.query('CREATE TABLE. . .', function (err) {
  db.query('CREATE TABLE. . .', function (err) {
    db.query('CREATE TABLE. . .', function (err) { });
  });
});
```

为了确保能够将错误报告给用户，还需要监听db error事件：

```
db.on('error', function () {
  // handle error
});
```

对于中断程序而言，当错误发生时，最好的处理方式是将错误和调用堆都展示给用户并终止程序。你或许还记得第4章中介绍的，当错误对象通过EventEmitter分发出来，但又没有对应的监听器时（也就是说事件未处理），Node会将错误抛出，确保开发者能够意识到错误的发生，而不是将错误“吞”掉。因此，事实上，对于本例，我们无须专门添加一个错误处理器，因为Node会将未处理的错误直接抛出。

首先，我们需要创建数据库并告诉MySQL要使用该数据库：

setup.js

```
/**
 * 创建数据库
 */

db.query('CREATE DATABASE IF NOT EXISTS `cart-example`');
 db.query('USE `cart-example`');
```

237

setup.js

```
/**
 * 创建表
 */

db.query('DROP TABLE IF EXISTS item');
 db.query('CREATE TABLE item (' +
  'id INT(11) AUTO_INCREMENT,' +
  'title VARCHAR(255),' +
  'description TEXT,' +
  'created DATETIME,' +
  'PRIMARY KEY (id))');
db.query('DROP TABLE IF EXISTS review');
db.query('CREATE TABLE review (' +
  'id INT(11) AUTO_INCREMENT,' +
  'item_id INT(11),' +
  'text TEXT,' +
```

```
  'stars INT(1),' +
  'created DATETIME,' +
  'PRIMARY KEY (id))');
```

setup.js

```
/**
 * 关闭客户端
 */

db.end();
```

正如我们在第3章中介绍的，当事件轮询中没有任务要处理时，Node就会退出该进程。在连接MySQL服务器的过程中，我们打开了一个文件描述符，于是Node事件轮询机制就会开始监听。当调用结束客户端时，文件描述符也会被关闭，因此，也要结束程序。

代码如下所示：

setup.js

```
db.end(function () {
  process.exit();
});
```

接下来测试刚刚书写的脚本：

```
$ node setup.js
```

然后，我们可以用mysql客户端确认下，数据库和表都已经创建好了（见图13-1）：

图13-1：通过MySQL命令行客户端来查看利用setup.js创建的数据库和表是否创建成功了

```
$ mysql
> show databases;
. . .
> use cart-examples;
. . .
> SHOW TABLES;
. . .
```

创建数据

接下来，我们在views目录下创建一个简单的布局。如下所示，该文件包含一个特殊的jade `block body`声明，用于将其他视图嵌入：

views/layout.jade

```
doctype 5
html
  head
    title My shopping cart
  body
    h1 My shopping cart
    #cart

block body
```

index文件展示了一个包含所有商品的列表，以及用于创建新商品的表单：

239

views/index.jade

```
extends layout
block body
h2 All items
if (items.length)
  ul
    each item in items
      li
        h3: a(href="/item/#{item.id}")= item.title
        = item.description
else
  p No items to show

h2 Create new item

form(action="/create", method="POST")
  p
    label Title
    input(type="text", name="title")
  p
    label Description
```

```
    textarea(name="description")
  p
    button Submit
```

因为在上述代码中，通过检查length属性来展示items数组项，所以，我们暂且先确保在/路由中传递一个空数组，如下所示。当然了，真正的数据稍后肯定是从数据库中获得的。

server.js

```
app.get('/', function (req, res, next) {
  res.render('index', { items: [] });
});. . .
```

对于商品查看页面，我们需要商品本身、关于它的评价以及创建新评价的表单。

views/item.jade

```
extends layout

block body
  a(href="/") Go back

  h2= item.title
  p= item.description
  h3 User reviews

  if (reviews.length)
    each review in reviews
      .review
        b #{review.stars} stars
        p= review.text
      hr
  else
    p No reviews to show. Write one!

form(action="/item/#{item.id}/review", method="POST")
  fieldset
    legend Create review
    p
      label Stars
      select(name="stars")
        option 1
        option 2
        option 3
        option 4
        option 5
    p
      label Review
```

240

```
    textarea(name="text")
  p
    button(type="submit") Send
```

注意在上述代码中，表单的action属性部分使用了jade的插补特性。通过使用#{}以一种安全的方式（HTML的实体会被转义）来引入变量。如果想要引入不需要转义的字符串变量，可以使用!{}。

在开始获取数据前，我们需要先创建数据来做简单的测试。

在项目配置代码下方，添加bodyParser中间件来处理POST请求：

server.js

```
/**
 * 中间件
 */

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

241 接着，完成/create路由：

server.js

```
/**
 * 创建商品路由
 */

app.post('/create', function (req, res, next) {
  db.query('INSERT INTO item SET title = ?, description = ?',
  [req.body.title, req.body.description], function (err, info) {
    if (err) return next(err);
    console.log(' - item created with id %s', info.insertId);
    res.redirect('/');
  });
});
```

上述代码有两部分非常意思。第一部分是db.query允许用后面提供的数据替换?记号。通过这种替换记号的方式，可以有效地避免字符串的拼接，从而避免SQL注入攻击。如果在查询语句中包含了?记号，那么需要提供一个包含要替换数据的数组作为第二个参数。

另外一部分有意思的是info对象。本例中，我们通过insertId来获取创建商品的ID。只要不发生错误，这个属性一直都在。如果有错误发生，我们就终止处理，并调用next方法。

创建评价的路由也类似：

```
/**
 * 创建商品评价路由
 */

app.post('/item/:id/review', function (req, res, next) {
  db.query('INSERT INTO review SET item_id = ?, stars = ?, text = ?',
  [req.params.id, req.body.stars, req.body.text], function (err, info) {
    if (err) return next(err);
    console.log(' - review created with id %s', info.insertId);
    res.redirect('/item/' + req.params.id);
  });
});
```

通过运行上述应用并创建一个商品来做测试（见图13-2）。

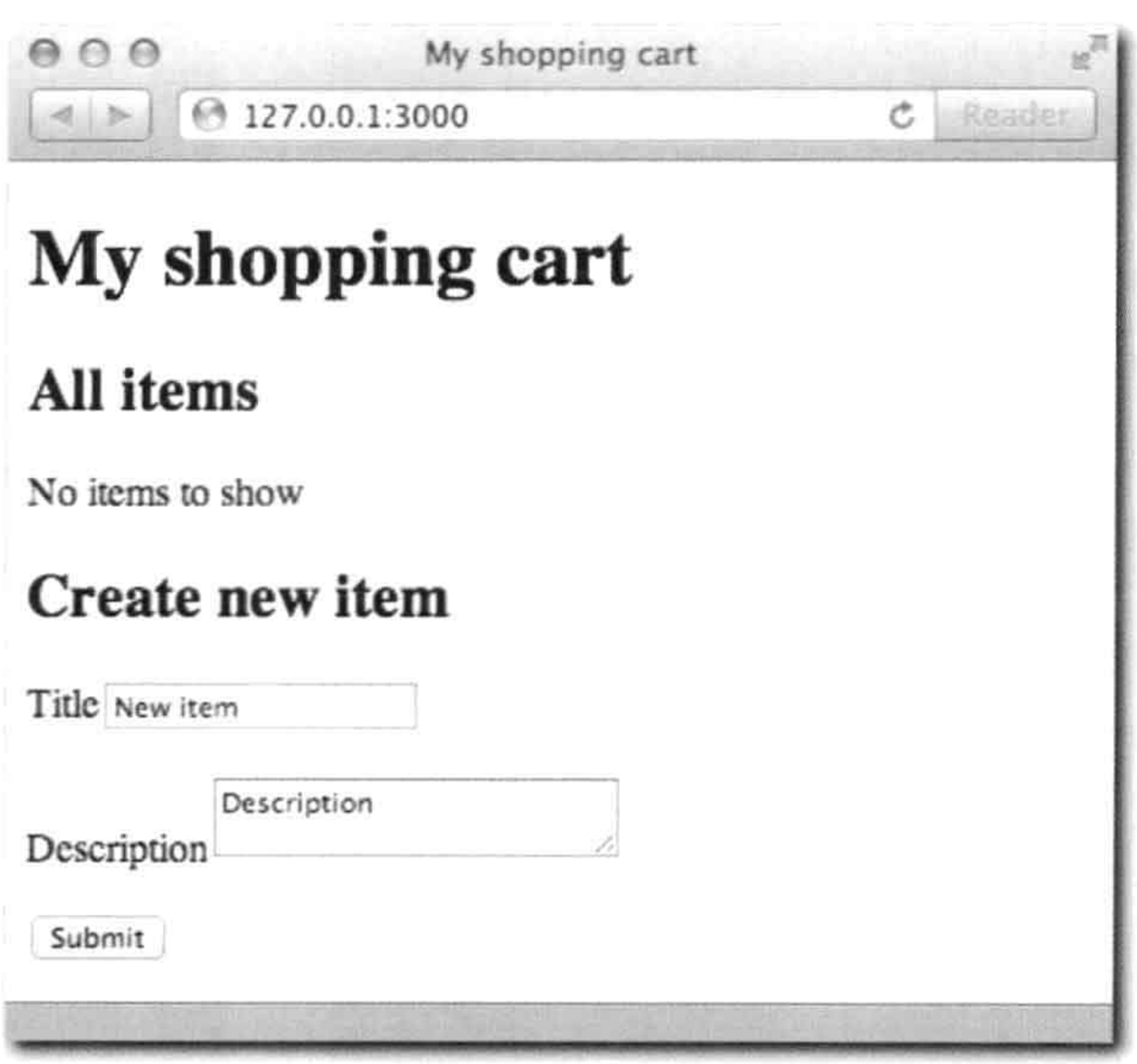

图13-2：在首页路由中，填写创建商品的表单

之后，就能在控制台看到如图13-3所示的内容了。

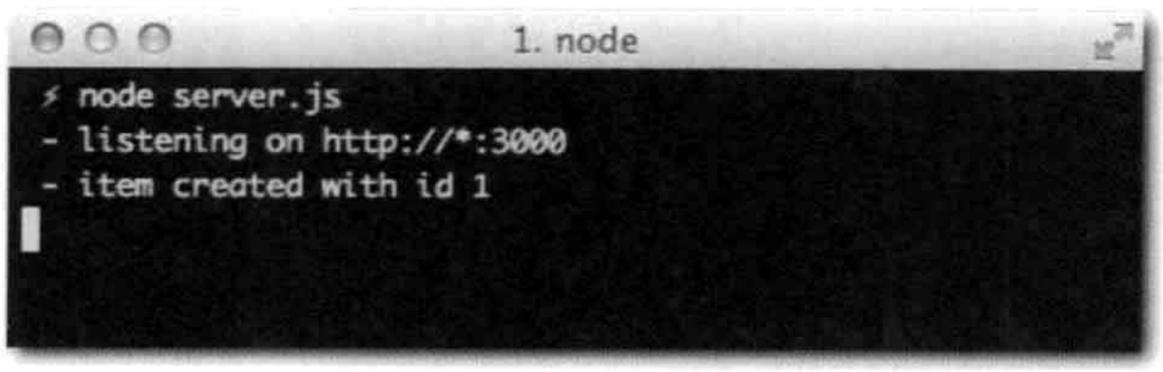

图13-3：商品创建成功后，在控制台显示了其id

获取数据

通过node-mysql从MySQL中获取数据是非常简单直观的。当执行的命令是SELECT时，回调函数中会接收一个包含查询结果对象的数组。数组中的对象包含了指定返回的字段。根据本章所属范畴，我们将只讨论SELECT指令。

回调函数中接收到的是数组，这其实和我们在index.jade模板中想要的数据结构是一致的，模板中，我们需要遍历数组，并显示其id、title以及description属性。

因此，我们要做的就是检查是否有错误发生，如果没有，就把查询结果传递给视图：

```
/**
 * 首页路由
 */
app.get('/', function (req, res, next) {
  db.query('SELECT id, title, description FROM item', function (err, results) {
res.render('index', { items: results });
  });
});
```

243

/路由完成后，就可以重启应用，查看所有商品的列表了。

对于列表中包含的查看商品路由，我们需要获取商品数据，确保它存在，并获取相关的评价。如果商品不存在，就返回404状态码。

为了让代码更可读，我们将逻辑按照执行顺序拆分到几个函数中：

server.js

```
/**
 * 查看商品路由
 */
app.get('/item/:id', function (req, res, next) {
  function getItem (fn) {
    db.query('SELECT id, title, description FROM item WHERE id = ? LIMIT 1',
    [req.params.id], function (err, results) {
      if (err) return next(err);
      if (!results[0]) return res.send(404);
      fn(results[0]);
    });
  }

  function getReviews (item_id, fn) {
    db.query('SELECT text, stars FROM review WHERE item_id = ?',
    [item_id], function (err, results) {
      if (err) return next(err);
      fn(results);
    });
```

```
  }

  getItem(function (item) {
    getReviews(item.id, function (reviews) {
      res.render('item', { item: item, reviews: reviews });
    });
  });
});
```

图13-4展示了完成后的应用。现在可以浏览商品、提交评价以及在界面上看到它们的信息了。

图13-4：完整应用展示

sequelize

在此前的例子中，直接操作SQL数据库的方式多少有些问题。

第一个问题就是建表的过程是手动的（耗时），而且表的定义并非项目本身的一部分。应用程序根本无法得知商品的title属性只能允许最多255个字符。如果能够知道，那么应用程序就可以对用户的输入做校验，并显示错误信息。

解决这个问题的方式就是使用sequelize：通过它可以定义schema和模型，同时还可以使用其同步的特性，根据那些定义来创建要使用的数据库表。这样一来，此前例子中的`setup.js`就不再需要了。

因为schema也是应用程序的一部分，我们可以使用它们来做类型转化。要存储包含指定数据的商品，我们可以直接传递一个JavaScript的Date对象，而无须手动构造MySQL需要的日期格式。

最后一点，同时也是很重要的一点就是联合。在此前的例子中，我们是手动去获取商品的评价信息的，而通过sequelize，可以自动获取这部分数据。

245 我们通过创建一个简单的任务列表应用，来运用sequelize为数据库表带来的不同的概念和特性。任务可以根据项目来分组。我们可以添加、创建和删除项目，同时也可以在指定项目中，添加、创建和删除任务。

初始化sequelize

由于sequelize内部使用了node-mysql驱动器，所以依赖列表就可以是如下形式：

package.json

```
{
    "name": "todo-list-example"
  , "version": "0.0.1"
  , "dependencies": {
        "express": "2.5.2"
      , "jade": "0.19.0"
      , "sequelize": "1.3.7"
    }
}
```

初始化Express应用

这部分应用将与以往传统的应用有所不同，这次在创建和删除商品时将采用Ajax的方式。我们可以通过使用DELETE方法让应用程序更加RESTful。若你对REST还不熟悉，那么简单来说它就是一系列准则，引入了一种HTTP协议更宽泛的使用方式，使得像HTTP的PATCH、DELETE以及平时不常使用的状态码为我们所用。

定义的路由如下所示：

- / (GET)：获取所有项目。
- /projects (POST)：创建项目。
- /project/:id (DELETE)：删除项目。
- /project/:id/tasks (GET)：获取任务。
- /project/:id/tasks (POST)：添加任务。
- /task/:id (DELETE)：删除任务。

```
/**
 * 模块依赖
 */

var express = require('express')

/**                                                          246
 * 创建应用
 */

app = express.createServer();

/**
 * 配置应用
 */

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

/**
 * 首页路由
 */

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

/**
 * 删除项目路由
 */

app.del('/project/:id', function (req, res, next) {
});

/**
 * 创建项目路由
 */

app.post('/projects', function (req, res, next) {
});

/**
 * 展示指定项目中的任务
 */

app.get('/project/:id/tasks, function (req, res, next) {
});

/**
```

```
 * 为指定项目添加任务
 */

app.post('/project/:id/tasks, function (req, res, next) {
});

/**
 * 删除任务路由
 */

app.del('/task/:id', function (req, res, next) {
});
/**
 * 监听
 */

app.listen(3000, function () {
  console.log(' - listening on http://*:3000');
});
```

247

我们再定义一个简单的布局，这次使用jQuery来更容易地发送Ajax请求：

views/layout.jade

```
doctype 5
html
  head
    title TODO list app
    script(src="http://code.jquery.com/jquery-1.7.2.js")
    script(src="/js/main.js")
  body
    h1 TODO list app
    #todo
      block body
```

注意了，在上述代码中，载入了main.js文件，该文件将包含所有的客户端逻辑（如：发送Ajax请求来提交表单）。

项目和任务列表展示方式一样，因为它们都有添加和删除操作：

views/index.jade

```
extends layout

block body
  h2 Projects

  #list
    ul#projects-list
```

```
    each project in projects
      li
        a(href="/project/#{project.id}/items")= project.title
        a.delete(href="/project/#{project.id}") x

  form#add(action="/projects", method="POST")
    input(type="text", name="title")
```

248

button(type="submit") Addviews/tasks.jade

```
h2 Tasks for project #{project.title}

#list
  ul#tasks-list
    each task in tasks
      li
        span= task.title
        a.delete(href="/task/#{task.id} ") x

form#add(action="/project/#{project.id}/tasks", method="POST")
  input(type="text", name="title")
  button Add
```

连接sequelize

现在，我们需要将sequelize添加到模块依赖中：

server.js

```
/**
 * 模块依赖
 */

var express = require('express')
  , Sequelize = require('sequelize')
```

接下来初始化主类。我们可以直接在模块依赖后做这个初始化工作，也可以出于对程序结构清晰的考虑，在应用设置后做。

server.js

```
/**
 * 初始化 sequelize
 */

var sequelize = new Sequelize('todo-example', 'root')
```

Sequelize构造器接收如下参数：

- database (String)
- username (String) - 必要
- password (String) - 可选
- other options (Object) - 可选
  - host (String)
  - port (Number)

249 可以使用如下命令行来创建数据库。记住将root替换为在Sequelize构造器中使用的MySQL用户名：

```
$ mysqldmin -u root -p create todo-exmaple
```

定义模型和同步

要定义模型，需要调用sequelize.define方法。我们可以在引入sequelize后直接调用该方法。该方法第一个参数是模型名，第二个参数是包含了属性的对象。

server.js

```
var Project = sequelize.define('Project', {
    title: Sequelize.STRING
  , description: Sequelize.TEXT
  , created: Sequelize.DATE
});
```

在上述代码中，属性的类型对应如下sequelize中的类型。接下来，每种类型都对应MySQL中的类型：

- Sequelize.STRING // VARCHAR(255)
- Sequelize.BOOLEAN // TINYINT(1)
- Sequelize.TEXT // TEXT
- Sequelize.DATE // DATETIME
- Sequelize.INTEGER // INT

除了传递类型之外，还可以传递一个包含选项的对象，比如，要设置默认值，就可以传递：

```
title: { type: Sequelize.STRING, defaultValue: 'No title' }
```

接下来定义任务模型：

server.js

```
/**
 * 定义任务模型
 */

var Task = sequelize.define('Task', {
    title: Sequelize.STRING
});
```

最后，设置一个hasMany的联合：

250

server.js

```
/**
 * 设置联合
 */

Task.belongsTo(Project);
 Project.hasMany(Task);
```

为了设置联合，Sequelize会处理构建响应的列、主键以及索引的任务。

belongsTo联合意味着每个Task都有一个指向它所属项目的字段。另外，每个任务模型都会有一个名为getProject的方法来获取其所属的项目。

对于hasMany而言，调用find查询到项目后，它们都会有一个名为getTasks的方法来获取项目中的任务。

除此之外，sequelize还支持另一种关系：hasOne。不过本例中用不到这种联合，它与belongsTo是相对的。

最后，我们要确保schema都同步到数据库中了，并且不需要手动运行CREATE TABLE命令：

```
/**
 * 同步
 */

sequelize.sync();
```

在开发阶段，我们会经常对数据库表做修改操作。因此，我们可以在调用sync方法时，传递一个{force: true}参数来让sequelize始终先删除已有的表，再重新创建，以确保数据变化总能同步到数据库中。

server.js

```
sequelize.sync();
```

创建数据

对于项目和任务列表，我们都需要在表单提交时，绑定一个jQuery的监听器。

当发送Ajax调用时，我们希望返回的是JSON格式的模型实例数据。

251 获取数据后，我们就将其添加到DOM中。

在这之前，我们需要添加一个static中间件来托管public/js目录（也需要创建好）。因为还需要使用jQuery来POST数据，所以还需要bodyParser中间件。

server.js

```
/**
 * 中间件
 */

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

public/js/main.js

```
$(function () {
  $('form').submit(function (ev) {
    ev.preventDefault();
    var form = $(this);
    $.ajax({
        url: form.attr('action')
      , type: 'POST'
      , data: form.serialize()
      , success: function (obj) {
          var el = $('<li>');
          if ($('#projects-list').length) {
            el
              .append($('<a>').attr('href', '/project/' + obj.id
+ '/tasks').text(obj.title + ' '))
              .append($('<a>').attr('href', '/project/' + obj.id)
.attr('class', 'delete').text('x'));
          } else {
            el
              .append($('<span>').text(obj.title + ' '))
              .append($('<a>').attr('href', '/task/' + obj.id)
.attr('class', 'delete').text('x'));
          }
          $('ul').append(el);
        }
    });
form.find('input').val(''); // clear the input
  });
});
```

代码很简单。捕获到网站中所有的表单提交，并利用Ajax的方式来提交：

1. 捕获表单提交。
2. 通过调用preventDefault方法来阻止默认行为。也就是说，阻止浏览器试图自动POST表单，因为我们想要用Ajax的方式来提交。 252
3. 调用jQuery的$.ajax方法来提交一个POST请求，同时将表单数据序列化成查询字符串发送过去（通过form.serialize序列化数据，并将其作为data属性值）。

JSON数据返回后，我们重新构造该数据项，并将其添加到项目列表或者任务列表中。如果该数据项是项目，那么我们将其链接添加到任务列表中，同时再添加一个删除链接。若是任务，则简单地添加一个span和删除链接。

现在我们来实现Express应用中的.post路由。这里我们使用模型上的.build方法：

server.js

```
/**
 * 创建项目路由
 */

app.post('/projects', function (req, res, next) {
  Project.build(req.body).save()
    .success(function (obj) {
       res.send(obj);
     })
    ..error(next)
});

/**
 * 为项目添加任务
 */

app.post('/project/:id/tasks', function (req, res, next) {
   res.body.ProjectId = req.params.id;
  Task.build(req.body).save()
    .success(function (obj) {
       res.send(obj);
     })
    .error(next)
});
```

需要记住很重要的一点：像本例这样，如果用户可以设置数据库中的字段，并且没有安全隐患的情况下，我们只需传递整个请求体即可（如：传递req.body）。哪怕我们在表单中只创建了一些输入项，但是不要忘记，用户是可以手动伪造任意请求类型的。

253 如第9章中介绍的，在Express中使用res.send方法可以很容易地发送 JSON数据。

如下所示，当调用模型实例上的.save方法时，sequelize会分发一个success事件并传递构建好的对象，或者一个failure事件并传递一个错误对象。在sequelize中，如下代码是有效的：

```
Task.build(req.body).save()
    .on('success', function (obj) {
      res.send(obj);
     })
    .on('failure', next)
```

可以使用success和error方法来更容易地添加事件处理器：

```
Task.build(req.body).save()
  .success(function (obj) {
    res.send(obj);
   })
  .error(next)
```

注意了，为了确保任务和项目之前的关系能够保留，我们在使用Task.build创建任务时，在Task对象上添加了ProjectId字段。回过头来，当我们在模型上设置了belongsTo关系后，sequelize会自动在schema定义中添加ProjectId字段。

获取数据

每个sequelize模型都有简单的方法可以获取指定数据表中单个或多个实例。

调用Model#find方法时，可以直接提供一个主键，并监听success和failure事件：

```
/**
 * 首页路由
 */

app.get('/', function (req, res, next) {
  Project.findAll()
    .success(function (projects) {
      res.render('index', { projects: projects });
    })
    .error(next);
});
```

254 由于此前建立了项目——任务的联合，在/project/:id/items路由，我们可以使用getTasks方法将项目和任务都传递给视图层：

server.js

```
app.get('/project/:id/tasks', function (req, res, next) {
  Project.find(Number(req.params.id))
    .success(function (project) {
```

```
      project.getTasks().on('success', function (tasks) {
        res.render('tasks', { project: project, tasks: tasks });
      })
    })
    .error(next)
});
```

另外，还要注意的是，当使用模型实例的find方法时，需要将参数转化为Number类型。这很重要，因为这样sequelize才能知道这里是用主键去查询。

接下来，我们实现剩下的路由：删除项目和删除任务。

删除数据

接下来，我们利用事件委派来捕获所有delete类的链接，并发送DELETE请求。将如下代码添加到$(form).submit处理器中：

public/js/main.js

```
$('ul').delegate('a.delete', 'click', function (ev) {
  ev.preventDefault();
  var li = $(this).closest('li');
  $.ajax({
      url: $(this).attr('href')
    , type: 'DELETE'
    , success: function () {
        li.remove();
      }
  });
});
```

jQuery的delegate方法允许捕获任意包含了delete类的超链接，不管它是本来就在DOM中的还是后台动态加进去的。

注意，在点击了含有delete类的超链接后，我们查找它的父元素li，并在Ajax请求成功后，将其删除。

接着，定义删除路由：

255

```
/**
 * 删除项目路由
 */

app.del('/project/:id', function (req, res, next) {
  Project.find(Number(req.params.id)).success(function (proj) {
    proj.destroy()
      .success(function () {
```

```
        res.send(200);
      })
      .error(next);
  }).error(next);
});

/**
 * 删除任务路由
 */

app.del('/task/:id', function (req, res, next) {
  Task.find(Number(req.params.id)).success(function (task) {
    task.destroy()
      .success(function () {
        res.send(200);
      })
      .error(next)
  }).error(next);
});
```

如上述代码所示，首先获取任务或者项目的实例，然后，调用destroy将其删除。当destroy指令成功后，发送200状态码给浏览器。

同样的，要想修改获取到的数据项的属性，可以调用updateAttributes。下述代码修改指定任务实例的标题：

```
task.updateAttributes({
   title: 'a new title'
});
```

图13-5展示了完整的应用。可以浏览项目和任务，异步地对它们进行添加和删除。

256

图13-5：为某个项目创建一个新任务

完整地完成应用

sequelize还有许多功能。就像Mongoose操作MongoDB那样，Sequelize可以在MySQL和模

型数据之间添加一层验证层，除了定义类型之外，这个功能也非常有用。

在模型中定义字段时，可以通过传递validate选项来设置验证机制。类型则定义在type中。

比如，要想任务标题只允许大写字母，那么模型定义可以采用如下方式：

```
var Task = sequelize.define('Task', {
    title: { type: Sequelize.STRING, isUppercase: true }
});
```

要设置自定义验证，只需传递任意的函数名和验证函数。要想查看Sequelize自带的完整验证函数列表，可以前往其官方文档查看：http://sequelizejs.com/?active=validations#validations。

还可以通过自定义的类和实例方法来扩展模型：

```
var Task = sequelize.define('Task', {
    title: { type: Sequelize.STRING, isUppercase: true }
  , classMethods: {
      staticMethod: function(){}
    }
  , instanceMethods: {
      instanceMethod: function(){}
    }
});
```

上述例子中的staticMethod方法可以通过如下方式进行调用：

257

```
Task.staticMethod()
```

instanceMethod则可以在查询到的实例上使用：

```
Task.find(4).success(function (task) {
  task.instanceMethod();
});
```

小结

MySQL仍旧是最流行、最可靠的开源数据库之一。无论新的趋势如何，毋庸置疑，MySQL依然是构建各类应用不错的选择。

本章介绍了一个优秀的MySQL Node.js驱动器。不过，需要手动书写 SQL语句来创建数据库、表才能查询。

对于开发Web应用，ORM通常是一种非常有用的武器。在本章第二个例子中，得益于Sequelize，我们无须再手写查询语句，而是可以直接通过模型类和实例来操作数据了。

尽管总要在选择正确的工具这件事情上面多加谨慎，不过，现在你应该懂得了在Node.js中什么样的项目适合使用MySQL。