第5章

数据库开发

5.1 非关系型数据库 MongoDB

伴随着设计工作和开发环境设置的完成，就可以正式进入开发阶段了。好的开始是成功的一半，只要严格遵循此前设计好的“图纸”，中小团队甚至个人也可以以极快的速度完成整个项目开发工作。

正式开发的第一步就是解决数据的持久化存储问题，即数据库问题。数据库开发的过程也就是将第 3.1.3 节设计好的 E-R 图实际转化为数据库的过程。

数据库方面有很多选择，既有以 Oracle、MySQL 为代表的关系型数据库，也有以 MongoDB 为代表的非关系型数据库。本书将基于整个技术栈的全面考虑，选用后者作为项目数据持久化的解决方案。

5.1.1 优势与基本概念

MongoDB 作为非关系型数据库的代表，其主要优势也就在于“非关系型”：

(1) NoSQL。意味着去掉了传统 SQL 数据库的“关系型”特性，数据之间并无关系。这非常有利于多服务器分布式存储，对大数据场景极为适用。

(2) 高性能。MongoDB 和传统 SQL 数据库相比，特别在读写方面，具有极高的性能。这也是得益于其“非关系型”的特性。

(3) 可扩展性。还是得益于 NoSQL 的特性，MongoDB 不必事先为数据建立字段，而是可随时扩充。与此相比，在传统 SQL 数据库中增、删字段是一件非常麻烦的事情。

(4) 基于 JavaScript。比起传统的 SQL 语句，MongoDB 完全通过操作 JSON 对象实现增、查、改、删，数据本身也完全以 JSON 格式存储；不仅易于编程控制，也与本书所使用的技术栈浑然相成。

在学习 MongoDB 之前，非常有必要掌握 MongoDB 的几个基本概念。

文档(document)。类似一个 JSON 对象(object)，相当于关系型数据库中的“行(row)”。

字段(field)。类似 JSON 对象中的各个键(key)，相当于关系型数据库中的“列(column)”。

集合(collection)。多个文档构成一个集合，相当于关系型数据库中的“表(table)”。

数据库(database)。多个集合构成一个数据库。一般来说，一个项目使用一个数据库。

参照此前设计好的 E-R 图(见图 3.2)：用户(user)、帖子(thread)和回复(comment)这三个实体在 MongoDB 中就是集合；而实体的属性，比如用户名(username)、密码(password)等就对应字段；如果将实体具体化，比如一条用户名为“Wu”、密码为“123456”的数据就可以看作一个文档；三个集合放在同一个数据库 BBS 之中。

可用图表示出来，如图 5.1 所示。

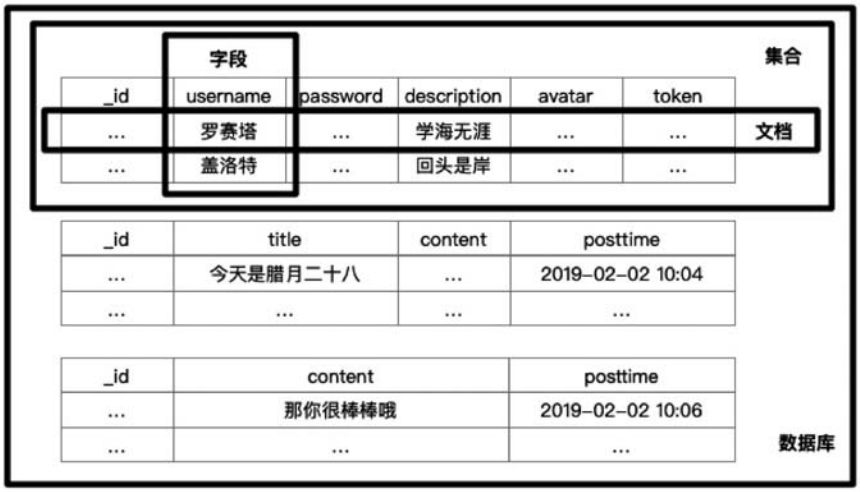

图 5.1 MongoDB 的示例

5.1.2 安装

在使用 MongoDB 之前，需要先安装。在各操作系统上的安装方法也有所不同。

1. Windows

在 https://www.mongodb.com/download-center/community 的 Server 标签处下载 MongoDB 的 Windows 版软件，进行安装。

注意，在安装过程中，需取消选中 Install MongoDB Compass 的复选框，否则极有可能在最后一步卡住。

安装完成后右击“计算机”，选择“属性”→“高级”→“环境变量”选项，在弹出的“环境变量”对话框中的“系统变量”中选择 path→“编辑”，在弹出的“编辑环境变量”对话框中添加刚才 MongoDB 的安装路径。比如，按照默认路径安装的话，就是 C:\Program Files\MongoDB\Server\4.2\bin，如图 5.2 所示。

此外，还要在 C 盘新建用来存储数据的文件夹 C:\data\db\，即在 C 盘根目录下新建文件夹 data，并在 data 下新建子文件夹 db。

完成后打开命令行工具，输入

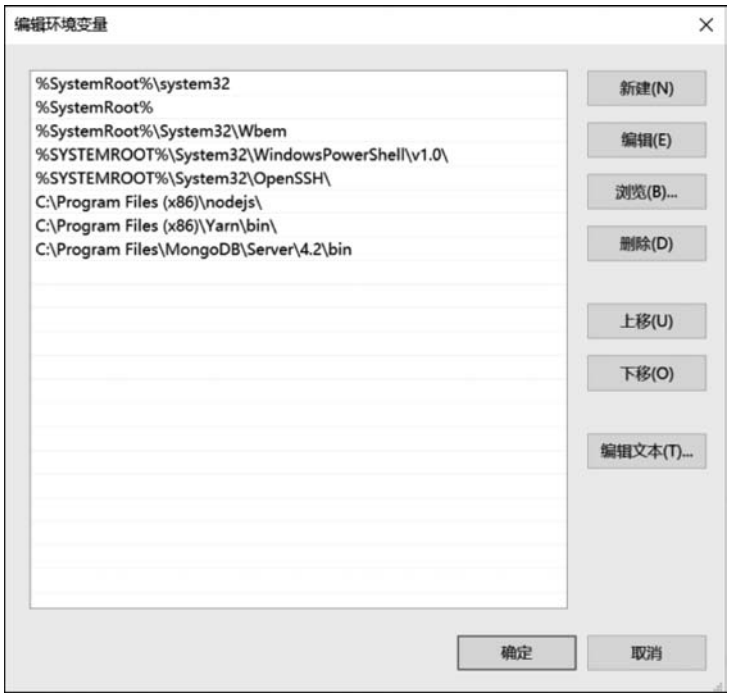

图 5.2　为 MongoDB 添加环境变量

```
$ mongo -- version
```

显示版本号,说明安装成功。

2. Mac OS

根据 https://brew.sh/index_zh-cn 的指示在终端安装 brew 后,执行如下命令即可:

```
$ brew install mongodb
```

3. Linux

在 Ubuntu 系统下可用如下命令直接安装:

```
$ apt-get install mongodb
```

在 CentOS 系统下则用如下命令安装:

```
$ yum install -y mongodb-org
```

5.1.3　基本操作

在命令行工具中输入如下命令可启动 MongoDB 的服务,默认占用 27017 号端口:

```
$ mongod
```

在保持该窗口启动 MongoDB 服务的前提下，不要关闭，再另开一个命令行工具的新窗口，执行如下命令：

```
$ mongo
```

即可启动 MongoDB Shell 交互界面，接下来的所有以>为起始的命令都要在 MongoDB Shell 交互界面下输入。比如：

```
> show dbs
```

是显示所有数据库。新安装 MongoDB 后的默认数据库只有 admin、config 和 local 三个。

```
> use test
```

则是使用名为 test 的数据库。这里请注意，虽然 test 数据库目前还不存在，但在插入记录后 MongoDB 就会自动创建。

显示该数据库下所有集合(表)：

```
> show collections
```

或

```
> show tables
```

返回结果都是空的，因为目前 test 数据库下还没有任何集合，那么接下来就添加一些数据。

5.1.4 MongoDB 的 CRUD

所谓 CRUD，就是指对数据进行创建、读取、更新、删除四种基本操作。

1. 创建

MongoDB 使用 insert()创建文档，其语法为：

```
db.集合名.insert(文档)
```

因为在上节使用了“use test”，所以现在 db 指代的就是 test 数据库(虽然还没有正式创建)；集合名和数据库一样，也是“先上车后补票”，先使用不存在的集合，等实际插入了数据再自动创建；至于文档，则完全以 JSON 格式定义，比如，在 MongoDB Shell 中插入这样两个文档(即关系型数据库中的“行”)：

```
> db.users.insert({"name":"Lee","age":18})
> db.users.insert({"name":"Wu","age":38})
```

分别返回 WriteResult({"nInserted":1})字样,说明各成功插入了一条数据。

2. 读取

MongoDB 使用 find()读取文档,其语法为:

```
db.集合名.find(查询条件)
```

查询条件也是以 JSON 的形式规定,如果不设置查询条件,find()就会返回该集合下的所有文档。比如:

```
> db.users.find()
```

会返回刚才插入的两条数据。此外还会发现,在返回的 JSON 数据中,除了刚才插入的 name 和 age 字段外,多出了一个_id 字段,这是 MongoDB 为每一个文档自动加上的独一无二的标志。换句话说,在实际数据库开发过程中,除非特殊需要,并不需要手动加入 ID 字段。

输入如下命令则只返回名字为“Lee”的数据:

```
> db.users.find({"name":"Lee"})
```

当需要通过比较来获取数据时,就需要使用条件操作符。MongoDB 的条件操作符如表 5.1 所示。

表 5.1 MongoDB 的条件操作符

操 作 符	英 文	语 义	符 号
$gt	greater than	大于	>
$lt	less than	小于	<
$gte	greater than equal	大于或等于	>=
$lte	less than equal	小于或等于	<=

这样,就可以用年龄条件来查询刚才插入的文档,比如:

```
> db.users.find({"age":{$gt:30}})
```

只会返回年龄在 30 岁以上的用户数据。

3. 更新

MongoDB 使用 update()更新文档,其语法为:

```
db.集合名.update(查询条件,更新对象)
```

其中,无论查询条件还是更新对象都以 JSON 格式定义。比如:

```
> db.users.update({"name":"Lee"},{"age":16})
```

就是将名为 Lee 的用户的年龄更新为 16。

更新完成后，再重新执行如下命令：

```
> db.users.find({"name":"Lee"})
```

查找 Lee 的数据，会发现什么都没有返回。但是执行

```
> db.users.find()
```

却发现两条数据都还在。这是为什么？

原因在于，这种更新方式会将原对象彻底更新为新对象。也就是说，更新对象中只有年龄信息的话，原对象的名字信息会彻底消失。如果只需要更新原对象的一个或几个字段，就得使用更新操作符。MongoDB 的更新操作符如表 5.2 所示。

表 5.2 MongoDB 的更新操作符

操 作 符	语 义
$ set	设置
$ push	插入
$ pull	抽出

使用如下命令将 Lee 的数据恢复原状：

```
> db.users.update({"age":16}, {"name":"Lee","age":18})
```

用更新操作符 $ set 修改 Lee 的年龄：

```
> db.users.update({"name":"Lee"}, { $ set:{"age":16}})
```

再次查询 Lee 的数据，会看到只有年龄信息做了改动。

另两个操作符 $ push 和 $ pull 都是对数组类数据做出修改。比如，随着产品需求发生变化，现在要给某个用户增添一个新字段——children，用来存放他/她的孩子们：

```
> db.users.update({"name":"Wu"}, { $ set:{"children":[]}})
```

此时的 children 字段为一个空的数组。另外，在用如下命令查询时，会发现 Lee 并没有 children 字段：

```
> db.users.find()
```

这是因为与 SQL 数据库不同，MongoDB 每条数据都是独立存储的，这也正是“行”被称作“文档”的原因所在。执行如下命令：

```
> db.users.update({"name":"Wu"}, { $ push:{"children":"Loli"}})
```

再次查询用户 Wu 的数据时，其 children 数组中多了一个 Loli，说明追加孩子成功。

在执行如下命令后重新查询，会发现孩子数据又消失了：

```
> db.users.update({"name":"Wu"}, { $pull:{"children":"Loli"}})
```

4. 删除

MongoDB 使用 remove()删除文档，其语法为：

```
db.集合名.remove(查询条件)
```

比如，使用如下命令删掉姓名为 Lee 的文档：

```
> db.users.remove({"name":"Lee"})
```

再用 find()查询，会发现只剩下一条数据了。

5.1.5　数据库可视化

如 5.1.4 节所示，通过 MongoDB Shell 交互界面也可以查看或修改数据。但毕竟命令行的方式使用起来不是那么直观。如果想用更方便、更直观的方式来操作 MongoDB，就要用到数据库可视化工具。

MongoDB 的可视化工具有很多，在这里笔者推荐使用 Robo 3T（官方网站为 https://robomongo.org/）。

在官方网站首页单击 Download Robo 3T 后，选择对应自己操作系统的版本并安装。

启动 Robo 3T，会弹出对话框让用户选择 MongoDB 服务。单击 create 按钮新建一个连接，默认状态下的地址就是 localhost:27017 号端口，单击 Save 按钮保存即可。

在命令行工具中输入

```
$ mongod
```

启动 MongoDB 的服务，不要关闭该窗口。在保持该服务开启的状态下，在 Robo 3T 中单击 Connect 按钮连接到刚才创建的连接。在左侧的树结构中找到 test→Collections→users，双击打开，就可以看到刚才插入的数据了。如图 5.3 所示，共有三种数据显示模式：树模式、表模式和文本（JSON）模式，这样可以让数据一览无余、一目了然。

图 5.3　Robo 3T 显示数据的三种模式

5.1.6 小结

和传统的 SQL 关系型数据库相比，MongoDB 作为非关系型数据库的代表，具有其独到的优势。

也正因为 MongoDB 属于非关系型数据库，使其在概念上与传统的关系型数据库略有不同：传统数据库中的表在 MongoDB 中被称为集合，行则被称为文档。

此外，还学习了使用 mongod 命令开启数据库服务，以及用 mongo 命令开启 MongoDB Shell 交互界面。在交互界面中，又学习了如何使用各种命令操控数据的增、查、改、删。

用一条条命令修改、查询数据毕竟不够直观，可以换用数据库可视化工具 Robo 3T 对 MongoDB 进行更直观的查询和操作。


视频讲解

5.2 数据建模工具 Mongoose

在 5.1 节学习的 MongoDB，也是在本书中，为了实现数据的持久化存储，所采用的数据库技术。在此基础上，本节将学习 Mongoose。所谓 Mongoose，是一个在 Node.js 环境下针对 MongoDB 进行便捷操作的模型工具。使用 Mongoose，就可以像操作 JavaScript 对象一样对数据进行创建、读取、更新、删除。

5.2.1 简介与安装

如前所述，MongoDB 是一个非关系型数据库。但在很多建模场合，用关系型数据库的思想进行设计更符合人的认知模式。通过使用 Mongoose，就可以以关系型的方式建模，同时又保留 MongoDB 非关系型的优点，可谓取长补短、珠联璧合。

另外，Mongoose 还封装了很多方法，大大简化了对 MongoDB 的操作。比如在 5.1.4 节所学的创建、读取、更新、删除的方法，在使用 Mongoose 后，就可以变得更加便捷、更加易操作。

使用 Mongoose 的两个基本前提是已经安装好了 Node.js 环境（包含 NPM，参见 4.1.1 节）和 MongoDB（参见 5.1.2 节）。

接下来按照 4.2.1 节所述的方法初始化一个名为 BBS 的项目，这也将成为贯穿本书所有后续章节的项目。

初始化完成后，在命令行工具中进入项目根目录，并执行如下命令安装第三方库 Mongoose：

```
$ cnpm install mongoose -- save
```

打开 package.json 会发现 dependencies 下多了 Mongoose，说明安装成功。

接下来，结合 BBS 项目一起了解几个 Mongoose 的基本概念。

5.2.2 图式

Mongoose 的一个基本概念就是图式。所谓图式（schema），就是通过规定实体属性及其对应数据类型，创建出数据库模型的基本骨架。图式本身并不具备操作数据库的能力，仅

仅是定义被操作集合字段的步骤。

Mongoose 图式可以简单书写为：

```
new mongoose.Schema({
    字段: 数据类型,
    字段: 数据类型,
    …
})
```

也可以写作：

```
new mongoose.Schema({
    字段: 数据类型,
    字段: {type:数据类型, 其他选项},
    …
})
```

对字段做进一步规定，常用的选项有 required(是否强制用户输入)、unique(是否不允许重复)、default(用户不输入时的默认值是什么)和 trim(要不要去掉两头的空格)。

图式中可使用的数据类型主要有 String(字符串)、Number(数字)、Date(日期)、Boolean(布尔值)、ObjectId(文档 ID)和 Array(数组)，更详细的类型可参考官方文档 https://mongoosejs.com/docs/schematypes.html。

在 3.1.3 节设计的 E-R 图实际上已经规定好了全部字段。在项目根目录下新建一个名为 model 的文件夹，并根据 E-R 图，在 model 中新建三个文件：User.js、Thread.js 和 Comment.js。

分别书写三个实体的图式，编辑 User.js 为如下内容：

```
const mongoose = require('mongoose');
const UserSchema = new mongoose.Schema({
    username: {                          //用户名
        type: String,
        unique: true,
        required: true,
        trim: true,
    },
    password: {                          //密码
        type: String,
        required: true,
        trim: true,
    },
    avatar: String,                      //头像
    description: {                       //个人描述
        type: String,
        trim: true,
    },
    token: String,                       //身份凭证
});
```

类似地，在 Thread.js 中写入如下代码：

```
const mongoose = require('mongoose');

const ThreadSchema = new mongoose.Schema({
    title: {                                    //帖子标题
        type: String,
        required: true,
        trim: true,
    },
    content: {                                  //帖子内容
        type: String,
        required: true,
        trim: true,
    },
    posttime: {                                 //帖子发布时间
        type: Date,
        default: Date.now(),                    //发布时间默认为用户提交时的当前时间
    },
});
```

同样，在 Comment.js 中写入：

```
const mongoose = require('mongoose');

const CommentSchema = new mongoose.Schema({
    content: {                                  //回复内容
        type: String,
        required: true,
        trim: true,
    },
    posttime: {                                 //回复时间
        type: Date,
        default: Date.now(),                    //回复时间默认为用户提交时的当前时间
    },
});
```

在 E-R 图中，除了每个实体的固有属性，还包括了实体之间的 1∶*N* 关系，为了体现出这些关系，还需要在图式中添加数据类型为 ObjectId 的字段，用来存储其他实体的 ID。在三个图式中分别追加如下代码：

```
const Thread = require('./Thread');

const UserSchema = new mongoose.Schema({
    ...
    threads: [{                                 //该用户发布过的帖子
        type: mongoose.SchemaTypes.ObjectId,
        ref: 'Thread',                          //关联到 Thread 模型
    }],
});
```

```
const User = require('./User');

const ThreadSchema = new mongoose.Schema({
    ...
    author: {                                    //该帖子的发布者
        type: mongoose.SchemaTypes.ObjectId,
        required: true,
        ref: 'User',                             //关联到 User 模型
    },
});
```

```
const User = require('./User');

const CommentSchema = new mongoose.Schema({
    ...
    author: {                                    //该回复的发布者
        type: mongoose.SchemaTypes.ObjectId,
        required: true,
        ref: 'User',                             //关联到 User 模型
    },
    target: {                                    //该回复的目标帖子
        type: mongoose.SchemaTypes.ObjectId,
        required: true,
        ref: 'Thread',                           //关联到 Thread 模型
    },
});
```

其中，threads(用户发布过的帖子)之所以要用中括号[]括起来，是因为一个“用户”对应多个“发布过的帖子”，即一对多关系，需要使用数组来表示。如此一来，图式与图式之间建立起了关系，也就完成了三个图式的全部声明工作。

5.2.3 模型

Mongoose 的另一个重要概念是模型(model)，它是连接图式与 MongoDB 集合的纽带，负责实际对数据库进行读写操作。

具体的书写格式为：

```
mongoose.model(模型名, 图式);
```

在 User.js、Thread.js 和 Comment.js 的图式下面分别追加各自的模型，并提供导出接口以供其他文件引用，代码如下：

```
...
const User = mongoose.model('User', UserSchema);
module.exports = User;
```

```
...
const Thread = mongoose.model('Thread', ThreadSchema);
module.exports = Thread;
```

```
...
const Comment = mongoose.model('Comment', CommentSchema);
module.exports = Comment;
```

在模型被实际引用并使用后，Mongoose 会根据模型名自动创建相应的数据集合，非常省心。这也是使用 Mongoose 的主要原因之一：从对数据库的繁重操作中解脱出来，转而集中精力在数据模型本身上。

关于 Mongoose 的这个集合自动创建机制也有必要补充一下。以存储用户信息的模型为例，刚才将其命名为 User（即 mongoose.model 的第一个参数），在日后实际使用该模型插入数据时，Mongoose 就会根据这个模型名（User）自动在数据库中建立相关集合：首先会将模型名转为小写；其次，还会给模型名加上 s 表示复数。于是，User 模型也就对应了数据库中的 users 集合。该机制如图 5.4 所示。

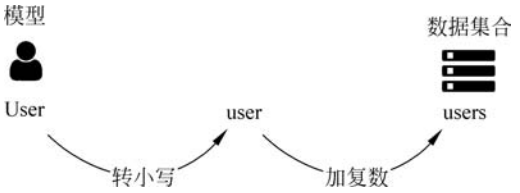

图 5.4　Mongoose 模型-集合创建机制

5.2.4　连接

与面向对象编程（Object Oriented Programming，OOP）的思想一样，Mongoose 的模型也是为了趋近人对世界的认知方式而设计出的更便于操作数据的中介手段。因此，在实际使用这些 Mongoose 模型之前，还需要先连接到数据库。

在项目根目录（即 BBS 文件夹）下创建一个名为 index.js 的文件，其内容如下：

```
const mongoose = require('mongoose');

const DB = 'mongodb://localhost:27017/bbs'; //定义数据库地址

mongoose.connect(DB, (err) => {
    if (err) throw err;
    console.log('已连接到数据库...');
});
```

代码中（err）=>{...}是一个回调函数（请参见 2.4.1 节中对回调函数的说明），Mongoose 首先尝试连接给定的数据库地址 DB，然后执行这个回调函数。在回调函数中先加了一句判断语句：如果在连接时出现错误（即 err），则用 throw 抛出，终止程序运行；否则

往下继续执行，显示成功连接数据库。

在命令行工具下先用如下命令开启 MongoDB 服务：

```
$ mongod
```

不关闭窗口，保持服务开启的状态下新开一个命令行工具窗口，在项目根目录 BBS 下执行：

```
$ node index.js
```

如果显示“已连接到数据库...”字样，则说明数据库连接成功。

虽然数据库连接成功，但同时也会如下看到三条警告：

```
DeprecationWarning: current URL string parser is deprecated, and will be removed in a future
version. To use the new parser, pass option { useNewUrlParser: true } to MongoClient.connect.
DeprecationWarning: current Server Discovery and Monitoring engine is deprecated, and will be
removed in a future version. To use the new Server Discover and Monitoring engine, pass option {
useUnifiedTopology: true } to the MongoClient constructor.
DeprecationWarning: collection.ensureIndex is deprecated. Use createIndexes instead.
```

这说明目前的写法将不再适用于未来版本。要消除这三条警告，可将原来的代码：

```
mongoose.connect(DB, (err) => {
    if (err) throw err;
    console.log('已连接到数据库...');
});
```

改为

```
mongoose.connect(DB,
    { useNewUrlParser: true, useCreateIndex:true,useUnifiedTopology:true },
    (err) => {
        if (err) throw err;
        console.log('已连接到数据库...');
    }
);
```

如此一来，即便日后 MongoDB 更新到新版本，也不必担心数据库连接会失效。

5.2.5 Mongoose 的 CRUD

在 5.1.4 节中，学习了 MongoDB 的 CRUD，接下来，将学习基于 Mongoose 的 CRUD。借助于 Mongoose 的模型机制，可以让数据库操作变得更加便捷、更加直观。

1. 创建

继续编辑 5.2.4 节的 index.js 文件，在顶部新增如下代码，引用此前定义的三个模型：User(用户)、Thread(帖子)和 Comment(回复)。

```
…
const User = require('./model/User');
const Thread = require('./model/Thread');
const Comment = require('./model/Comment');
…
```

在回调函数中添加如下语句：

```
…
    console.log('已连接到数据库…');
    const user = new User({
        username: 'Test',
        password: '123456',
    });                                    //将 User 模型实例化
    user.save();                           //保存数据
…
```

在命令行工具中执行 index.js。再用 Robo 3T 打开 bbs 数据库，双击 users 表（集合），就可以看到刚才新增的数据。实例化与.save()的并用让操作更加直观且符合人的认知，即先将模型 User 具体化为一个实际用户，再将这个实际用户的数据存储到数据库。

如果新增帖子时还要记录用户的 ID，可将上面代码修改为：

```
…
    console.log('已连接到数据库…');
    const user = new User({
        username: 'Wu',
        password: '123456',
    });                                    //将 User 模型实例化，同时 user 也有了_id 属性
    const thread = new Thread({
        title: '新年快乐',
        content: '祝大家新年快乐!',
        author: user,                      //将 user 的_id 作为帖子的 author 字段
    });                                    //将 Thread 模型实例化
    user.threads.push(thread);             //将 thread 的_id 写入到用户发帖字段的数组中
    user.save();                           //保存该用户到数据库
    thread.save();                         //保存该帖子到数据库
…
```

后再执行，在 Robo 3T 左侧树状图中的 bbs 数据库上右击，选择 Refresh 刷新数据，就会发现多出了 threads 集合，可双击打开查看。其中，author 字段存储的就是新用户 Wu 的 ID，而 posttime 因为在声明模型时就设了 default，显示的是插入数据时的格林威治时间（中国时区需要加 8 小时）。另外，users 集合中的 threads 数组也多出了新增的帖子。

2. 读取

读取方面，只需要用

```
模型名.findOne(检索条件).exec(回调函数)
```

就可以。比如将上节代码修改为如下代码后执行：

```
…
    console.log('已连接到数据库…');
    User.findOne({
        username: 'Wu',
    }).exec((err, user) => {
        console.log(user);
    });
…
```

可以看到刚才插入的用户名为 Wu 的这条数据。若想获取多条数据，也可以用 find() 取代 findOne()，返回所有符合条件的数据的数组。

通过 user.threads 的确可以获取该用户的所有发帖，但只可见帖子的 ID。若要在查询一个用户的同时，也获取其所有发帖的实际内容，可以先用迭代出数组中每个帖子的 ID，然后再用 findOne()一条条查询，但这样写的话，不仅麻烦，还大大降低查询效率。其实更简便的做法是插入一个 populate()，写法如下：

```
…
    console.log('已连接到数据库…');
    User.findOne({
        username: 'Wu',
    })
    .populate('threads', 'title posttime')
    .exec((err, user) => {
        console.log(user);
    });
…
```

.populate()中第一个参数表示查询 User 模型的 threads 字段，第二个参数表示只需获取关联帖子的 title 和 posttime 信息。早在定义用户图式(UserSchema)的 threads 字段时，就已经将其 ref 设置为了“Thread”，因此 Mongoose 会自动到相对应集合(threads)去提取信息。

也可以用同样的方法获取某条帖子的作者信息。

3. 更新

基于 Mongoose 的数据更新非常直观，根据条件找到数据后，一一修改其属性值并使用.save()方法保存即可。比如修改用户 Wu 的个人描述为：

```
…
    console.log('已连接到数据库…');
    User.findOne({
        username: 'Wu',
    }).exec((err, user) => {
        const wu = user;                          //将找到的 user 对象设为 wu
        wu.description = 'Happy new year!!';      //修改 wu 的个人描述
        wu.save();                                //将修改后的 user 对象保存到数据库
    });
…
```

在 Robo 3T 中刷新并查看，发现 Wu 的个人描述变成了“Happy new year!!”。

4. 删除

删除也很简单，使用.remove()就可以了。但如果被删除数据和其他集合有关联，则需要考虑周全才可以将数据彻底删除干净，比如：

```
...
    console.log('已连接到数据库...');
    Thread.findOne({ title: '新年快乐' }).exec((err, thread) => {
        const thd = thread;
        User.findById(thd.author).exec((err, user) => {
            const wu = user;
            wu.threads.remove(thd);              //从用户帖子列表中移除该帖 ID
            thd.remove();                        //从数据库中删除该帖
            wu.save();                           //保存用户删除该帖后的状态
        });
    });
...
```

5.2.6 小结

本节在 MongoDB 的基础上，进一步学习了 Mongoose。Mongoose 是在 Node.js 环境下对 MongoDB 进行便捷操作的模型工具。

Mongoose 不仅大大简化了对 MongoDB 的操作，还能够用关系型数据库的思想对数据进行建模，这样既保留了非关系型数据库的优点，又吸收了关系型数据库的长处。

Mongoose 有两个重要概念：图式（schema）和模型（model）。图式规定字段及其数据类型，模型则是图式与数据库之间的桥梁。

模型可被其他文件引用，在实例化之后可以以非常方便且直观的方式对数据对象进行创建、读取、更新、删除。