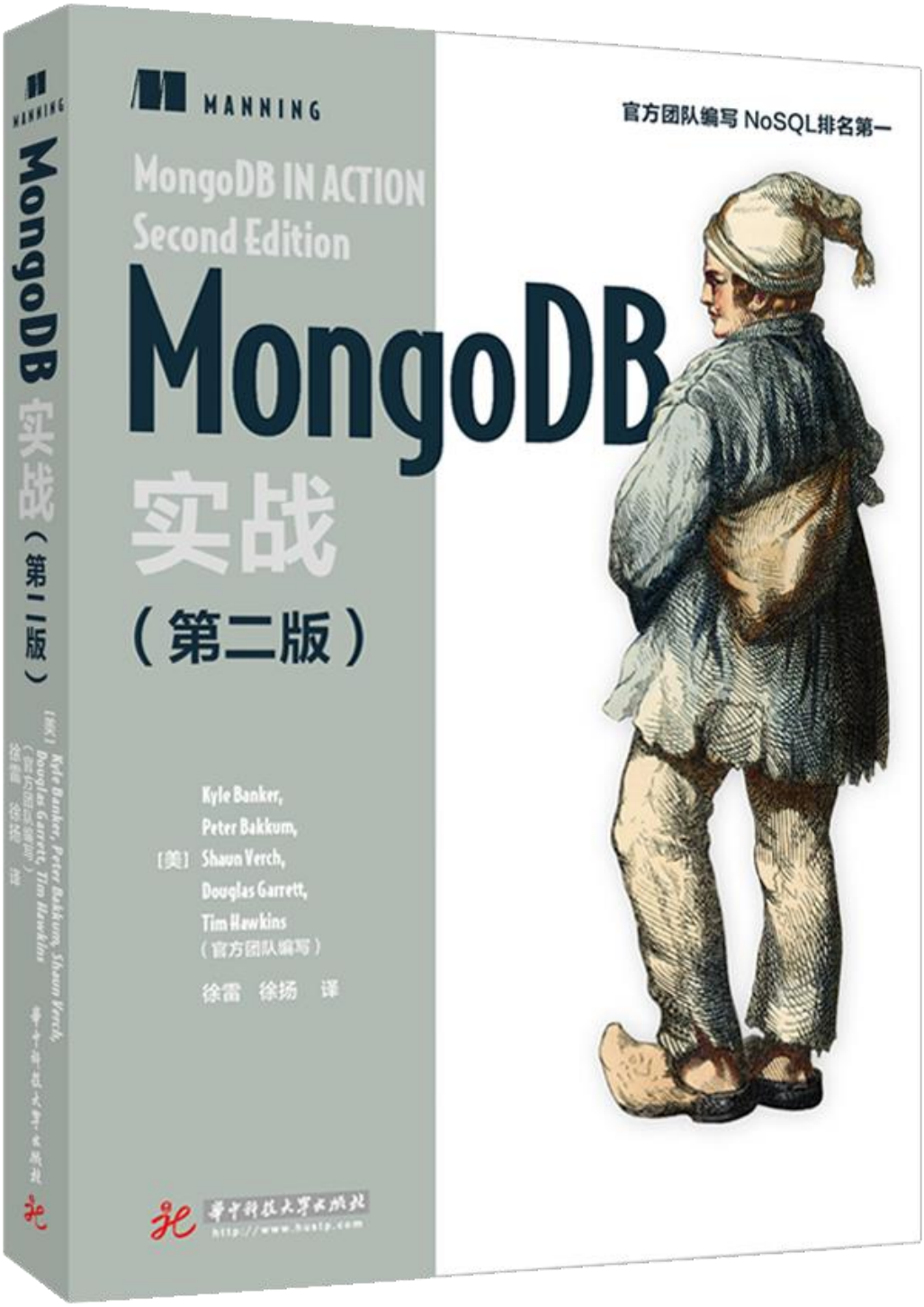
MANNING
官方团队编写 NoSQL排名第一
MongoDB IN ACTION
Second Edition
MongoDB
实战
（第二版）
Kyle Banker,
Peter Bakkum,
[美] Shaun Verch,
Douglas Garrett,
Tim Hawkins
（官方团队编写）
徐雷 徐扬 译
MongoDB 实战（第二版）
华中科技大学出版社
http://www.hustp.com

第 2 章

通过JavaScript shell操作MongoDB

MongoDB through the JavaScript shell

本章内容

- MongoDB shell里使用CRUD操作
- 构建索引和使用explain()
- 掌握基本管理
- 获取帮助

前一章提到了MongoDB运行的经验。如果想学习更多关于实战操作的介绍，本章内容就是。使用MongoDB shell，本章通过一系列实际操作来教会大家基本的数据库知识。你将会学习如何新增、读取、更新和删除(CRUD)文档，在这个过程中熟悉MongoDB查询语言。此外，我们还会初步学习数据库索引，以及如何使用它们来优化查询。然后我们将会学习一些基本的管理命令，建议一些使用MongoDB shell获取帮助的方式。可以把本章作为已经介绍的概念的详细阐述和MongoDB shell操作实战。

MongoDB shell是实验数据库的帮助工具，可以运行ad-hoc查询、管理MongoDB实例。当使用MongoDB开发应用时，我们会使用语言驱动而不是shell，但是shell可以用来做测试和重构这些查询。任意的MongoDB查询都可以在shell里运行。

如果你是MongoDB shell新手，也不要担心，记住它提供了你期望的所有功能；它允许你检查和操作数据，以及管理数据库服务器。MongoDB shell在查询语言方面不同于其他数据库，它没有采用标准的SQL查询语言，我们可以使用Javascript语言和简单的API操作数据库。这意味着你可以在shell里编写脚本与MongoDB数据库交互。如果不熟悉Javascript，也没有关系，只需要简单的语法就可以使用shell的功能，本章里所有的例子都会详细解释。MongoDB shell里提供的API与各个语言驱动里提供的接口一样，所以shell里编写的查询代码很容易移植到自

己的应用程序代码里。

如果跟着例子进行操作，就会收获很大、受益匪浅，但是必须先安装MongoDB。安装步骤可以在附录A里找到。

2.1 深入 MongoDB shell

Diving into the MongoDB shell

用MongoDB JavaScript shell操作数据库非常简单，而且能够对于文档、集合和数据库查询语言有个实际的感知。下面对MongoDB进行详细介绍。

我们将会从获取和运行shell开始，然后学习用JavaScript如何表示文档，学习如何插入这些文档到MongoDB集合中。要检验这些插入命令，就需要实际查询集合数据、更新操作。最后，通过学习删除数据和集合来完成CRUD操作的练习。

2.1.1 启动 shell

参考附录A，可以很快在本机安装一个可以工作的MongoDB服务和一个运行的`mongod`实例。通过运行`mongo`执行文件启动MongoDB shell：

```
mongo
```

如果shell程序启动成功，你的屏幕看起来就会如图2.1所示。shell顶部显示的是运行的MongoDB服务器的版本信息，后面是选择连接的当前数据库信息。

图 2.1 启动 MongoDB JavaScript shell

如果你熟悉JavaScript，就可以开始输入代码，然后探索学习shell。或者，看看如何运行MongoDB数据库的第一个操作命令。

2.1.2 数据库、集合和文档

正如我们知道的，MongoDB把数据存储在文档中，数据可以以JSON (JavaScript Object Notation)格式输出。你可能喜欢在不同的地方存储不同类型的文档，比如users 和 orders。这意味着MongoDB需要一种方式来分类文档，这与关系型数据库RDBMS中的表类似。在MongoDB数据库中，我们称之为集合（collection）。

MongoDB把集合分别存储在不同的数据库中。与传统的SQL数据库不同，MongoDB的数据库只区分集合的命名空间。要查询MongoDB数据库，需要知道存储文档数据的数据库和集合的名字。如果开始没有指定数据库，shell会选择默认的test数据库。为了与后面的练习例子统一，我们切换到tutorial数据库：

```
> use tutorial
switched to db tutorial
```

你会看到一个切换数据库成功的提示信息。

为什么MongoDB有数据库和集合？答案取决于MongoDB如何在磁盘上写数据。数据库中有的集合都分组保存在相同的文件中，所以从内存的角度说，这是合理的，保证相关的结合在同一个库里。你可能还希望不同的应用访问同一个集合（多租户），而且保持数据组织性很有帮助，以备未来需求。

创建数据库和集合

你可能好奇，怎样才能切换到tutorial数据库而不需要显示创建它。其实创建数据库不是必须的。只有在第一次插入数据库和集合时才会创建。这个行为符合MongoDB动态操作数据的模式。正如文档的数据结构不需要提前定义一样，单个的数据库和集合也可以在运行时创建。

这样可以简化并加速开发过程。也就是说，你不用担心意外创建数据库或者集合，绝大部分驱动都会阻止这种事情发生。

现在是时候创建第一个文档了。因为使用JavaScript shell，文档使用JSON格式指定，所以简单的文档定义如下所示：

```
{username: "smith"}
```

这个文档包含了一个简单的key和value，用来存储smith的用户名。

2.1.3 插入和查询

要插入文档，就需要选择一个目标集合。我们恰如其分地选择users集合。下面是插入代码：

```
> db.users.insert({username: "smith"})
WriteResult({ "nInserted" : 1 })
```

注意：例子中，我们选择MongoDB shell命令开头带个>符号，这样可以区分输出结果。

你会注意到，在输入这个命令后会出现一些延迟。此刻，既没有在磁盘上创建tutorial数据库，也没有创建users集合。延迟的原因是要为二者分配初始化文件。

如果插入成功，就已经成功保存了第一个文档。在默认的MongoDB数据库配置里，确保已插入这个数据，即使你关闭shell或者重启机器。你可以使用查询来查看新文档数据：

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

因为数据是users集合的一部分，所以重新打开shell，运行查询，就会显示如下的结果。应答消息如下所示：

```
{ "_id" : ObjectId("552e458158cd52bcb257c324"), "username" : "smith" }
```

MONGODB 的_ID 字段

注意到_id字段已经默认添加到文档里。我们可以把_id字段的值作为文档的主键。每个MongoDB文档都需要一个_id，而且如果创建文档时没有_id，就会专门创建一个MongoDB ObjectID添加到文档里。出现在控制台中的ObjectID与代码里列举的不同，但是在集合中_id值的作用是唯一的，这是基本要求。可以在文档里插入自己的_id，ObjectID是MongoDB默认的。

下一章会介绍更多的关于ObjectID的知识。我们选择继续添加第二个用户user到集合中：

```
> db.users.insert({username: "jones"})
WriteResult({ "nInserted" : 1 })
```

现在集合中有2个文档了。继续通过运行count来验证下结果：

```
> db.users.count()
```

传递查询条件

现在集合里既然有1个以上文档了，我们来看下更复杂的查询。和前面一样，我们先来查询集合里所有的文档：

```
> db.users.find()
{ "_id" : ObjectId("552e458158cd52bcb257c324"), "username" : "smith" }
{ "_id" : ObjectId("552e542a58cd52bcb257c325"), "username" : "jones" }
```

我们也可以给find方法传递简单的查询选择权。查询选择器是用来匹配集合中文档的。要查询集合中username 为jones的数据，可以传递简单的条件，语句如下：

```
> db.users.find({username: "jones"})
{ "_id" : ObjectId("552e542a58cd52bcb257c325"), "username" : "jones" }
```

这个查询条件{username: "jones"}会返回索引名字为jones的文档数据,它会迭代所有的文档。

注意，调用find方法若不传递参数，就等价于传递空条件。也就是说，db.users.find()和db.users.find({})的效果一样。

当然也可以在查询语句中指定多个字段，这样隐式创建AND语句。例如，可以使用下面的选择器查询：

```
> db.users.find({
... _id: ObjectId("552e458158cd52bcb257c324"),
... username: "smith"
... })
{ "_id" : ObjectId("552e458158cd52bcb257c324"), "username" : "smith" }
```

程序中的三个点是MongoDB shell自动添加的，表示这是单行命令。

查询条件会返回等价的文档。条件会使用AND，也就是按并且关系进行查询，索引必须匹配_id和username 字段。

也可以使用MongoDB的$and操作符。之前的查询语句等价修改为

```
> db.users.find({ $and: [
... { _id: ObjectId("552e458158cd52bcb257c324") },
... { username: "smith" }
... ] })
{ "_id" : ObjectId("552e458158cd52bcb257c324"), "username" : "smith" }
```

使用OR查询文档的语法类似：只需要把$or操作符替换一下就可以了。思考下面的语句：

```
> db.users.find({ $or: [
... { username: "smith" },
... { username: "jones" }
... ]})
{ "_id" : ObjectId("552e458158cd52bcb257c324"), "username" : "smith" }
{ "_id" : ObjectId("552e542a58cd52bcb257c325"), "username" : "jones" }
```

这个查询条件会返回smith和jones的文档，因为我们要找的是名字等于smith或jones的数据。

这个例子和之前的不同，因为它不是简单地插入或查找一个数据文档，而是查询文档本身。

在MongoDB里，使用文档来表示命令的做法很普遍，如果你习惯关系型数据库，则可能会感到吃惊。这种做法的一个好处就是，它非常容易在应用中构建自己的查询，因为它们是文档而不是SQL字符串。

我们已经看了基本的创建和读取数据的操作。现在是时候来看一下如何更新数据了。

2.1.4 更新文档

所有的更新至少需要2个参数。第一个指定要更新的文档，第二个定义要如何修改此文档。第一个例子演示了如何修改单个文档，第二个例子演示了如何修改多个文档，甚至如本节的末尾部分一样是集合中的所有文档。但是要记住，默认`update()`只更新一个文档。

通常有两种类型的更新操作，用于不同的属性和使用场景。其中一个类型的更新是在一个文档或者多个文档上修改，另外一个是使用新文档取代旧的文档。

从下面的例子我们来看一下简单的文档：

```
> db.users.find({username: "smith"})
{ "_id" : ObjectId("552e458158cd52bcb257c324"), "username" : "smith" }
```

更新操作符

第一种类型的更新需要传递一个文档参数，还有一些操作符作为第二个参数。本节里，我们先来看下如何使用`$set`操作符，它可以为单个字段设置特定的值。

假设用户smith要增加自己的居住地国家，我们可以使用如下命令更新：

```
> db.users.update({username: "smith"}, {$set: {country: "Canada"}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

这个命令会告诉MongoDB，找到一个用户名为`smith`的文档，然后把`country`属性设置为`Canada`。我们可以在服务器返回的消息里看到更新结果。如果查询被修改的数据，就可以看到该文档已经被更新：

```
> db.users.find({username: "smith"})
{ "_id" : ObjectId("552e458158cd52bcb257c324"), "username" : "smith",
     "country" : "Canada" }
```

替换更新

另外一个更新文档的方式就是替换文档，而不是更新某个字段。当使用`$set`操作符的时候这一点容易混淆。思考下面不同的更新命令：

```
> db.users.update({username: "smith"}, {country: "Canada"})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

这个例子里，文档被替换为只包含country字段的文档。username字段被删除，因为它只是用来匹配文档，第二个参数用来更新替换。当我们在使用这种更新时应该多加注意。新文档的查询如下所示：

```
> db.users.find({country: "Canada"})
{ "_id" : ObjectId("552e458158cd52bcb257c324"), "country" : "Canada" }
```

_id相同，但是数据已经被替换为新的文档了。当确定是新增或者修改数据而不是替换整个文档时，就使用$set操作符。

把用户名username重新添加到记录里：

```
> db.users.update({country: "Canada"}, {$set: {username: "smith"}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.users.find({country: "Canada"})
{ "_id" : ObjectId("552e458158cd52bcb257c324"), "country" : "Canada",
    "username" : "smith" }
```

如果以后不想要country字段了，使用$unset操作符删除即可：

```
> db.users.update({username: "smith"}, {$unset: {country: 1}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.users.find({username: "smith"})
{ "_id" : ObjectId("552e458158cd52bcb257c324"), "username" : "smith" }
```

更新复杂数据

我们来丰富一下以上例子。正如在第1章里看到的，我们使用文档来标识数据。文档可以包含复杂的数据。假设除了保存配置信息，用户还可以保存自己喜欢的东西。符合要求的文档格式可能看起来如下：

```
{
  username: "smith",
  favorites: {
    cities: ["Chicago", "Cheyenne"],
    movies: ["Casablanca", "For a Few Dollars More", "The Sting"]
  }
}
```

favorites键指向了一个包含2个键的新对象，它包含喜欢的城市和电影。假设你已经弄懂了，就可以把之前smith的文档修改为这个格式了。此时应该想起了$set操作符了：

```
> db.users.update( {username: "smith"},
...   {
...     $set: {
...       favorites: {
...         cities: ["Chicago", "Cheyenne"],
...         movies: ["Casablanca", "For a Few Dollars More", "The Sting"]
...     }
...   }
... })
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

请注意，间距缩进不是必须的，但是这样可以避免错误，文档的可读性更强。

下面我们来用相似的方式修改jones数据。此时，我们只能添加一些喜欢的电影：

```
> db.users.update( {username: "jones"},
...   {
...     $set: {
...       favorites: {
...         movies: ["Casablanca", "Rocky"]
...       }
...     }
...   })
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

如果输入错误，则可以使用向上的方向键回到前一条语句。

选择查询users集合来确保两个更新成功：

```
> > db.users.find().pretty()
{
    "_id" : ObjectId("552e458158cd52bcb257c324"),
    "username" : "smith",
    "favorites" : {
        "cities" : [
            "Chicago",
            "Cheyenne"
        ],
        "movies" : [
            "Casablanca",
            "For a Few Dollars More",
            "The Sting"
        ]
```

```
    }
}
{
    "_id" : ObjectId("552e542a58cd52bcb257c325"),
    "username" : "jones",
    "favorites" : {
        "movies" : [
            "Casablanca",
            "Rocky"
        ]
    }
}
```

严格来说，`find()`命令文档返回一个cursor游标。因此，要访问文档，就需要迭代游标。`find()`命令自动返回`20`个文档——如果可用——就会迭代游标20次。

有了这些例子文档，我们可以开始体验MongoDB查询语言的强大之处了。特别是，查询引擎可以深入内嵌对象的内部，以及根据数据元素进行匹配，这些都是非常有用的功能。注意，我们是怎样通过给`find`操作附加`pretty`操作来获取从服务器端返回的良好格式的结构。严格来说，`pretty()`实际上就是`cursor.pretty()`，它可以配置游标以容易阅读的方式来显示结果。

我们可以在本次查询里看到这两个概念的演示例子，找出所有喜欢电影Casablanca（卡萨布兰卡）的用户：

```
> db.users.find({"favorites.movies": "Casablanca"})
```

favorites和movies之间的原点会告诉查询引擎来搜索一个键名为`favorites`的对象，内部键名为`movies`作为新的匹配条件。因此，这个查询返回2个用户文档，如果数组中的元素匹配了最初查询，数组上的查询将会匹配。

假设你知道任意喜欢Casablanca的用户也会喜欢电影The Maltese Falcon，而且想要用更细数据库来反映这个事实，那么就要看更复杂的查询。如何在MongoDB更新命令里实现呢？

高级更新

你可能想到再次使用`$set`操作符，但是这样做需要重新编写并发送整个电影数组。因为我们想要做的就是给列表添加元素，最好还是使用`$push`或者`$addToSet`。这2个命令都是往数组中添加数据，但是第二个是唯一的，阻止了重复的数据。

这就是你要的更新语句:

```
> db.users.update( {"favorites.movies": "Casablanca"},
...     {$addToSet: {"favorites.movies": "The Maltese Falcon"} },
```

```
...           false,
...           true )
WriteResult({ "nMatched" : 2, "nUpserted" : 0, "nModified" : 2 })
```

这个代码容易理解。第一个参数是查询条件，匹配电影列表中包含Casablanca的用户。第二个参数使用`$addToSet`添加The Maltese Falcon到列表中。

第三个参数`false`，控制是否允许upsert。这个命令告诉更新操作，当一个文档不存在的时候是否插入它，这取决于更新操作是操作符更新还是替换更新。

第四个参数`true`，表示是否是多个更新。默认情况下，MongoDB更新只针对第一个匹配文档。如果想更新所有匹配的文档，就必须显示指定这个参数。如果你想对`smith`和`jones`都更新数据，那么这个参数是必须的。

我们将会详细介绍更新，但是在继续学习之前先练习一下这些例子。

2.1.5 删除数据

现在我们已经学习了通过MongoDB Shell 创建、读取和更新数据的基本操作。最后我们来学习下最简单的操作：删除数据。

如果没有参数，删除操作将会清空集合里的所有文档。如果要清空`foo`集合里的所有文档内存，可以输入以下命令：

```
> db.foo.remove()
```

通常我们只需要删除集合中某个文档，因此，我们要传递查询选择器给`remove()`方法。如果要删除所有喜欢Cheyenne城市的用户，则可以这样编写简单的表达式：

```
> db.users.remove({"favorites.cities": "Cheyenne"})
WriteResult({ "nRemoved" : 1 })
```

注意：`remove()`操作不会删除集合，它只会删除集合中的某个文档。我们可以把它和SQL中的DELETE命令进行类比。

如果要删除集合及其附带的索引数据，可以使用`drop()`方法：

```
> db.users.drop()
```

创建、读取、更新和删除都是任意数据库的基本操作。如果你已经用过，那么可以直接在MongoDB里进行CRUD的操作练习。下一节里，我们将会学习如何通过辅助索引来增强查询、修改和删除。

2.1.6 shell 的其他特性

你可能已经注意到了，shell可以很方便地做很多与MongoDB有关的工作。我们可以使用上下方向键快速切换之前的命令，使特定的输入自动完成，比如集合名字。自动完成特性使用tab键来自动完成或者列举完成的可能情况[1]。

[1]完整的快捷键列表，请参考 http://docs.mongodb.org/v3.0/reference/program/mongo/#mongo-keyboard- shortcuts。

我们也可以通过输入help来获取更多关于shell的帮助信息：

```
> help
```

许多函数也可以打印帮助信息，用来提示如何使用函数。试一下：

```
> db.help()
DB methods:
    db.adminCommand(nameOrDocument) - switches to 'admin' db, and runs
    command [ just calls db.runCommand(...) ]
    db.auth(username, password)
    db.cloneDatabase(fromhost)
    db.commandHelp(name) returns the help for the command
    db.copyDatabase(fromdb, todb, fromhost)
⋮
```

查询的帮助通过一个不同的函数`explain`来提供。我们会在后面详细讲解。在启动MongoDB shell时还有许多参数选项可用。要显示这个列表，可以在启动MongoDB shell时指定help标签：

```
$ mongo --help
```

你不需要担心怎样使用这些功能特性，我们还没有详细介绍shell呢！但是它确实可以提供我们需要的帮助信息。

2.2 使用索引创建和查询

Creating and querying with indexes

通过创建索引来改善查询性能是常见的做法。幸运的是，MongoDB的索引可以方便地从shell创建。如果你还不了解数据库索引，则本节可以帮你弄清楚这些概念；如果你应经使用过索引，就会发现创建索引非常简单。使用`explain()`方法监控查询。

2.2.1 创建大集合

索引只有在集合包含许多文档的时候才有意义。我们先往`numbers`集合里添加`20000`个文档。因为MongoDB shell也是个Javascript解析器，代码实现起来非常简单：

```
> for(i = 0; i < 20000; i++) {
    db.numbers.save({num: i});
  }
WriteResult({ "nInserted" : 1 })
```

因为文档很多，所以如果花费了几秒钟也不要惊讶。一旦返回，我们可以运行一些查询语句来检验所有显示的文档：

```
> db.numbers.count()
20000
```

```
> db.numbers.find()
{ "_id": ObjectId("4bfbf132dba1aa7c30ac830a"), "num": 0 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac830b"), "num": 1 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac830c"), "num": 2 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac830d"), "num": 3 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac830e"), "num": 4 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac830f"), "num": 5 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac8310"), "num": 6 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac8311"), "num": 7 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac8312"), "num": 8 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac8313"), "num": 9 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac8314"), "num": 10 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac8315"), "num": 11 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac8316"), "num": 12 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac8317"), "num": 13 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac8318"), "num": 14 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac8319"), "num": 15 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac831a"), "num": 16 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac831b"), "num": 17 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac831c"), "num": 18 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac831d"), "num": 19 }
Type "it" for more
```

`count()`命令显示已经插入了20000个文档。下一个查询显示了前20条结果（shell里的结果可能不太一样）。

我们可以使用`it`命令来显示其余的结果：

```
> it
{ "_id": ObjectId("4bfbf132dba1aa7c30ac831e"), "num": 20 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac831f"), "num": 21 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac8320"), "num": 22 }
⋮
```

`it`命令告诉shell返回下一个结果集[1]。

既然已经有了大量的测试数据，我们来尝试一些新的查询。假设你已经熟悉了MongoDB的查询引擎，则匹配文档属性`num`的查询语句如下：

```
> db.numbers.find({num: 500})
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac84fe"), "num" : 500 }
```

范围查询

更有意思的是，你可以使用`$gt`和`$lt`运算符来进行范围查询。这两个符号表示大于和小于。假如你要查询所有`num`值大于19995的所有文档，则语句如下：

```
> db.numbers.find( {num: {"$gt": 19995 }} )
```

[1]你可能好奇背后发生的事情。所有的查询都创建一个游标，以便于在结果集上迭代。这是使用 shell 时候看不到的工作，所以此时没有必要讨论太详细。如果你等不及学习游标及其特性，那可以直接阅读第 3 章和第 4 章的内容。

```
{ "_id" : ObjectId("552e660b58cd52bcb2581142"), "num" : 19996 }
{ "_id" : ObjectId("552e660b58cd52bcb2581143"), "num" : 19997 }
{ "_id" : ObjectId("552e660b58cd52bcb2581144"), "num" : 19998 }
{ "_id" : ObjectId("552e660b58cd52bcb2581145"), "num" : 19999 }
```

可以结合使用两个运算符，设置上下限：

```
> db.numbers.find( {num: {"$gt": 20, "$lt": 25 }} )
{ "_id" : ObjectId("552e660558cd52bcb257c33b"), "num" : 21 }
{ "_id" : ObjectId("552e660558cd52bcb257c33c"), "num" : 22 }
{ "_id" : ObjectId("552e660558cd52bcb257c33d"), "num" : 23 }
{ "_id" : ObjectId("552e660558cd52bcb257c33e"), "num" : 24 }
```

可以使用简单的JSON文档查看，你也可以用与SQL一样的方式来指定范围。`$gt`和`$lt`是MongoDB查询语言里两个主要的运算符。`$gte`表示大于等于，`$lte`表示小于等于，而`$ne`表示不等于。我们会在后续的章节里看到更多的运算符。

当然，除非这种查询非常高效，否则也没有价值。下一节，我们会思考如何通过MongoDB的索引功能来提升查询效率。

2.2.2 索引和 explain( )

如果你使用过关系型数据库，则可能对于SQL的`EXPLAIN`非常熟悉了，它是一个调试和优化查询的好工具。当所有数据库接收到查询后，它必须弄清楚如何执行查询；这就称为查询计划。`EXPLAIN`描述了查询路径并且允许开发者通过确定查询使用的索引来诊断慢的查询语句。查询通常可以有多种方式执行，但有时候并非我们期望的方式[1]。MongoDB 有自己的`EXPLAIN`版本，它可以提供相同的功能。

要了解其工作原理，我们来分析一下刚才已经使用的查询语句。在系统里运行下面的命令：

```
> db.numbers.find({num: {"$gt": 19995}}).explain("executionStats")
```

结果应该与列表2.1里显示的类似。"`execution-Stats`"关键字是MongoDB 3.0新增的，请求不同的模式并输出更详细的信息。

列表2.1 未使用索引的典型的explain("executionStats")输出结果

```
{
   "queryPlanner" : {
      "plannerVersion" : 1,
      "namespace" : "tutorial.numbers",
      "indexFilterSet" : false,
      "parsedQuery" : {
         "num" : {
```

[1]【译者注】SQL Server 有执行计划，可视化工具，可以优化分析 SQL 语句的性能。

```
                "$gt" : 19995
            }
        },
        "winningPlan" : {
            "stage" : "COLLSCAN",
            "filter" : {
                "num" : {
                    "$gt" : 19995
                }
            },
            "direction" : "forward"
        },
        "rejectedPlans" : [ ]
    },
    "executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 4,
        "executionTimeMillis" : 8,
        "totalKeysExamined" : 0,
        "totalDocsExamined" : 20000,
        "executionStages" : {
            "stage" : "COLLSCAN",
            "filter" : {
                "num" : {
                    "$gt" : 19995
                }
            },
            "nReturned" : 4,
            "executionTimeMillisEstimate" : 0,
            "works" : 20002,
            "advanced" : 4,
            "needTime" : 19997,
            "needFetch" : 0,
            "saveState" : 156,
            "restoreState" : 156,
            "isEOF" : 1,
            "invalidates" : 0,
            "direction" : "forward",
            "docsExamined" : 20000
        }
    },
    "serverInfo" : {
        "host" : "rMacBook.local",
        "port" : 27017,
        "version" : "3.0.6",
        "gitVersion" : "nogitversion"
    },
    "ok" : 1
}
```

看到explain()的输出结果[1]，你可能感到惊讶，居然查询引擎会扫描整个集合（即20000个文档）(docsExamined)，而只返回了4个结果。totalKeysExamined字段显示了整个扫描的索引

[1]这些例子里我们插入“hostname”作为机器主机名。在你的机器上可能是 localhost，或者机器名或者名字加上.local。不要担心与我们的结果不同，这和平台的版本还有 MongoDB 版本有关系。

数量，它的值是0。

扫描文档的数量和低效查询的数量居然差距这么大。现实的情况是集合和文档可能更大，实际查询消耗的时间可能会更多，超过8 ms（不同的机器可能不太一样）。

集合需要的就是一个索引。我们可以使用createIndex()方法为num 键创建索引。可以输入以下命令：

```
> db.numbers.createIndex({num: 1})
{
    "createdCollectionAutomatically" : false,
    "numIndexesBefore" : 1,
    "numIndexesAfter" : 2,
    "ok" : 1
}
```

在MongoDB 3.0以后的版本，`createIndex()`方法取代了`ensureIndex()`方法。如果你使用的是旧版本的MongoDB，则可以继续使用`ensureIndex()`而不用`createIndex()`。在MongoDB 3中，`ensureIndex()`依然可用，它是`createIndex()`的别称。

对于其他的MongoDB操作，比如查询和更新，要传递文档给createIndex()方法来确定索引的键。此时{num: 1}文档表示应该为numbers集合中的所有文档的num键建立升序索引。

我们可以通过`getIndexes()`方法来检验索引是否创建成功：

```
> db.numbers.getIndexes()
[
    {
        "v" : 1,
        "key" : {
            "_id" : 1
        },
            "name" : "_id_",
            "ns" : "tutorial.numbers"
        },
        {
            "v" : 1,
            "key" : {
            "num" : 1
        },
        "name" : "num_1",
        "ns" : "tutorial.numbers"
    }
]
```

集合现在有2个索引。第一个是标准的`_id`索引，自动为每个集合构建的；第二个是我们自己创建的`num`索引。这些索引的名字分别叫`_id_` 和 `num_1`。如果没有设置名字，MongoDB会自动创建一个有意义的名字给它们。

如果使用explain()进行查询，可以看到应答时间有很大改变。具体如列表2.2所示。

列表2.2　explain()输出的索引查询

```
> db.numbers.find({num: {"$gt": 19995 }}).explain("executionStats")
{
    "queryPlanner" : {
        "plannerVersion" : 1,
        "namespace" : "tutorial.numbers",
        "indexFilterSet" : false,
        "parsedQuery" : {
            "num" : {
                "$gt" : 19995
            }
        },
        "winningPlan" : {
            "stage" : "FETCH",
            "inputStage" : {
                "stage" : "IXSCAN",
                "keyPattern" : {
                    "num" : 1
                },
                "indexName" : "num_1",
                "isMultiKey" : false,
                "direction" : "forward",
                "indexBounds" : {
                    "num" : [
                    "(19995.0, inf.0]"
                ]
            }
        }
    },
    "rejectedPlans" : [ ]
},
    "executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 4,
        "executionTimeMillis" : 0,
        "totalKeysExamined" : 4,
        "totalDocsExamined" : 4,
        "executionStages" : {
            "stage" : "FETCH",
            "nReturned" : 4,
            "executionTimeMillisEstimate" : 0,
            "works" : 5,
            "advanced" : 4,
            "needTime" : 0,
            "needFetch" : 0,
            "saveState" : 0,
            "restoreState" : 0,
            "isEOF" : 1,
            "invalidates" : 0,
            "docsExamined" : 4,
            "alreadyHasObj" : 0,
            "inputStage" : {
                "stage" : "IXSCAN",
                "nReturned" : 4,
```

使用 num_1 索引

返回 4 个文档

只扫描 4 个文档

更快

```
"executionTimeMillisEstimate" : 0,
"works" : 4,
"advanced" : 4,
"needTime" : 0,
"needFetch" : 0,
"saveState" : 0,
"restoreState" : 0,
"isEOF" : 1,
```

```
            "invalidates" : 0,
            "keyPattern" : {
               "num" : 1
            },
            "indexName" : "num_1",
            "isMultiKey" : false,
            "direction" : "forward",
            "indexBounds" : {
               "num" : [
                  "(19995.0, inf.0]"
               ]
            },
            "keysExamined" : 4,
            "dupsTested" : 0,
            "dupsDropped" : 0,
            "seenInvalidated" : 0,
            "matchTested" : 0
         }
      }
   },
   "serverInfo" : {
      "host" : "rMacBook.local",
      "port" : 27017,
      "version" : "3.0.6",
      "gitVersion" : "nogitversion"
   },
   "ok" : 1
}
```

使用 num_1 索引

现在查询使用了num 键的num_1索引，它只扫描了与查询有关的4个文档。整个查询消耗的总时间从8 ms 降低到0 ms!

索引并非没有成本，它们会占用空间，而且会让插入成本稍微提升，对于查询优化来说这是必备的工具。如果对这个例子理解有些疑问，可以阅读本书第8章，第8章专门深入讲解了索引和查询优化机制。接下来我们要学习关于MongoDB实例的管理命令。我们也会学习如何从shell里获取帮助，这有助于我们掌握各种不同的shell命令。

2.3 基本管理

Basic administration

本章承诺要介绍通过JavaScript shell 管理MongoDB。我们已经学习了基本的数据操作和索引。这里，我们将会介绍获取mongod进程信息的方法。例如，你可能想知道不同的集合占用的存储空间，或者集合中定义了多少索引。命令返回的信息可以帮助我们来诊断性能问题并且跟踪分析数据。

我们将了解一下MongoDB的命令接口。最特殊的就是 ,在MongoDB上执行的是非CRUD操作，

从服务器状态检查到数据文件完整性验证，都使用数据库命令来实现。我们将会解释MongoDB上下文里的命令，以及展示它使用起来有多么简单。最后，最好要知道去哪里获取帮助。我们后面将会告诉大家在shell里如何获取深入学习MongoDB的更多帮助。

2.3.1 获取数据库信息

通常，我们想知道安装MongoDB的服务器上存在什么集合和数据库。

幸运的是，MongoDB shell提供了许多命令，包括一些语法糖，用来获取系统的信息。

`show dbs`打印系统中所有的数据库列表信息：

```
> show dbs
admin (empty)
local 0.078GB
tutorial 0.078GB
```

`show collections`展示了当前数据库里所有的集合[1]。如果选择的还是`tutorial`数据库，就会看到这个集合。后面我们还会继续使用这个集合。

```
> show collections
numbers
system.indexes
users
```

有个可能你不认识的集合`system.indexes`。这个是每个数据库都存在的特殊集合。`system.indexes`中的每个入口都定义了一个数据库索引，我们可以使用`getIndexes()`方法来查看，正如之前我们看到的一样。但是MongoDB 3.0放弃了对于`system.indexes`集合的访问，我们可以使用`createIndexes`或者`listIndexes`替换。`getIndexes()` JavaScript方法可以被`db.runCommand( {"listIndexes": "numbers"} )` shell命令取代。

对于低级别数据库和集合分析，`stats()`方法非常有用。当我们在数据库对象上运行此方法时，就会获取下面的结果：

```
> db.stats()
{
    "db" : "tutorial",
    "collections" : 4,
    "objects" : 20010,
    "avgObjSize" : 48.0223888055972,
    "dataSize" : 960928,
```

[1]这个结果里显示的有些信息只有在复杂的调试或者优化时才会有用。但是我们至少也可以了解某个集合以及索引占用的空间。

```
    "storageSize" : 2818048,
    "numExtents" : 8,
    "indexes" : 3,
    "indexSize" : 1177344,
    "fileSize" : 67108864,
    "nsSizeMB" : 16,
    "extentFreeList" : {
        "num" : 0,
        "totalSize" : 0
    },
    "dataFileVersion" : {
        "major" : 4,
        "minor" : 5
    },
    "ok" : 1
}
```

我们也可以在单个集合上执行stats()：

```
> db.numbers.stats()
{
    "ns" : "tutorial.numbers",
    "count" : 20000,
    "size" : 960064,
    "avgObjSize" : 48,
    "storageSize" : 2793472,
    "numExtents" : 5,
    "nindexes" : 2,
    "lastExtentSize" : 2097152,
    "paddingFactor" : 1,
    "paddingFactorNote" : "paddingFactor is unused and unmaintained in 3.0.
    It remains hard coded to 1.0 for compatibility only.",
    "systemFlags" : 1,
    "userFlags" : 1,
    "totalIndexSize" : 1169168,
    "indexSizes" : {
        "_id_" : 654080,
        "num_1" : 515088
    },
    "ok" : 1
}
```

2.3.2 命令如何执行

MongoDB特定的操作集合——与本章目前为止介绍的insert、update、remove、query命令不同——称为数据库命令。数据库命令通常是管理性的，正如刚才介绍的stats()方法，但是也可以控制MongoDB的核心功能，比如更新数据。

无论提供什么功能，所有的数据库命令实现都有个共同点，就是它们在一个叫做$cmd的虚拟集合上实现查询。

要理解这句话，我们来快速看个例子，回忆下我们如何调用stats()命令：

```
> db.stats()
```

stats()方法包装了shell命令的方法调用。

可以尝试输入下面等价的操作：

```
> db.runCommand( {dbstats: 1} )
```

执行后的结果与使用stats()方法显示的一样。注意，这个命令也是使用{dbstats: 1}定义的。通常，我们可以通过给runCommand()方法传递参数来调用任意命令。

以下是说明如何运行集合stats命令：

```
> db.runCommand( {collstats: "numbers"} )
```

输出的结果看起来很熟悉。

要掌握数据库命令的内部原理，我们还要看看runCommand()方法是怎么工作的。这个也不难，因为MongoDB shell将会在无括号调用的时候打印任意方法的实现。

如不使用这种运行命令的方式：

```
> db.runCommand()
```

我们可以执行无括号的版本，来看看内部机制：

```
> db.runCommand
  function ( obj, extra ){
    if ( typeof( obj ) == "string" ){
        var n = {};
        n[obj] = 1;
        obj = n;
        if ( extra && typeof( extra ) == "object" ) {
            for ( var x in extra ) {
                n[x] = extra[x];
            }
        }
    }
    return this.getCollection( "$cmd" ).findOne( obj );
  }
```

最后一行就是在$cmd集合上的查询。定义在数据库特殊命令集合上进行查询，$cmd，查询选择器定义了命令本身。这就是背后原理。你能设想一个手动运行统计命令的方法吗？其实非常简单。

```
> db.$cmd.findOne( {collstats: "numbers"} );
```

使用runCommand帮助方法可以更简单，但是通常最好要理解底层的机制。

2.4 获取帮助

Getting help

目前为止，使用MongoDB shell实验数据操作和管理数据库的价值是显而易见的。但是因为可能大家会在shell花费很多时间，所以要知道如何获取帮助。

内置的帮助命令是首选方式。用db.help()也可以打印通常数据库使用的操作方法。通过运行db.numbers.help()，我们可以找到一个相似的方法列表。

也有tab键只能完成功能。开始时输入任意方法的首字母，然后按tab键两次，就会看到所有匹配的方法。下面是所有以get开头的集合操作方法：

```
> db.numbers.get
db.numbers.getCollection( db.numbers.getIndexes(
db.numbers.getShardDistribution(
db.numbers.getDB( db.numbers.getIndices(
db.numbers.getShardVersion(
db.numbers.getDiskStorageStats( db.numbers.getMongo(
db.numbers.getSlaveOk(
db.numbers.getFullName( db.numbers.getName(
db.numbers.getSplitKeysForChunks(
db.numbers.getIndexKeys( db.numbers.getPagesInRAM(
db.numbers.getWriteConcern(
db.numbers.getIndexSpecs( db.numbers.getPlanCache(
db.numbers.getIndexStats( db.numbers.getQueryOptions(
```

官方的MongoDB手册是最好的资源，我们可以在http://docs.mongodb.org找到。它包括教程和参考资料，而且更新到最新的MongoDB版本。文档还包括每个语言的MongoDB驱动实现，比如Ruby驱动，这些驱动在我们使用应用程序访问MongoDB的时候是必须用到的。

如果你还想学习更多内容，而且喜欢使用JavaScript，shell可以让你查看任意的方法实现。例如，假设你想知道save()到底是如何工作的。虽然我们可以通过查看MongoDB源码来达到目的，但是还有更简单的方法，就输入方法名，且不需要括号。下面是大家正常执行save()方法的命令：

```
> db.numbers.save({num: 123123123});
```

以下就是如何来查看实现代码：

```
> db.numbers.save
function ( obj , opts ){
    if ( obj == null )
```

```
        throw "can't save a null";

    if ( typeof( obj ) == "number" || typeof( obj) == "string" )
        throw "can't save a number or string"

    if ( typeof( obj._id ) == "undefined" ){
    obj._id = new ObjectId();
        return this.insert( obj , opts );
    }
    else {
        return this.update( { _id : obj._id } , obj , Object.merge({
    upsert:true }, opts));
    }
}
```

仔细阅读代码，你就会看到`save()`方法只是对`insert()`和`update()`方法的包装。在检查`obj`参数的类型后，如果要保存的对象没有`_id`字段，就会添加字段，然后调用`insert()`，否则就执行更新操作。

检查shell方法的实现非常方便。记住这个技巧，后面扩展学习MongoDB shell的时候非常有用。

2.5 总结

Summary

现在我们已经实际看到了文档的数据模型，而且我们也演示了数据模型上的不同MongoDB操作。我们已经学习了如何创建索引，以及通过`explain()`基于索引的性能优化。此外，我们已经了解如何查询系统上集合和数据库的信息，也知道了聪明的`$cmd`集合，需要帮助时也知道找到合适的方式。

我们学习了许多MongoDB shell的命令，但是这替代不了我们构建真实应用的经验。这也是我们为什么要在下一章里从实验操作转换到真实的数据库开发。我们会学习驱动是如何工作的，然后使用Ruby驱动来构建一个简单的应用，使用真实的数据来操作MongoDB数据库[1]。

[1]【译者注】所有语言的官方驱动都可以在官方网站下载，也可以通过包管理工具搜索 MongoDB。无论是 Node.js 还是 Java、C#都可以找到。对于别的语言的例子，大家也可以自己动手实践，参考官方文档即可：https://docs.mongodb.com/ecosystem/drivers/。我写了 Java 和 C#的例子程序，包括 Helper 工具类。中国 MongoDB 学习交流群 511943641