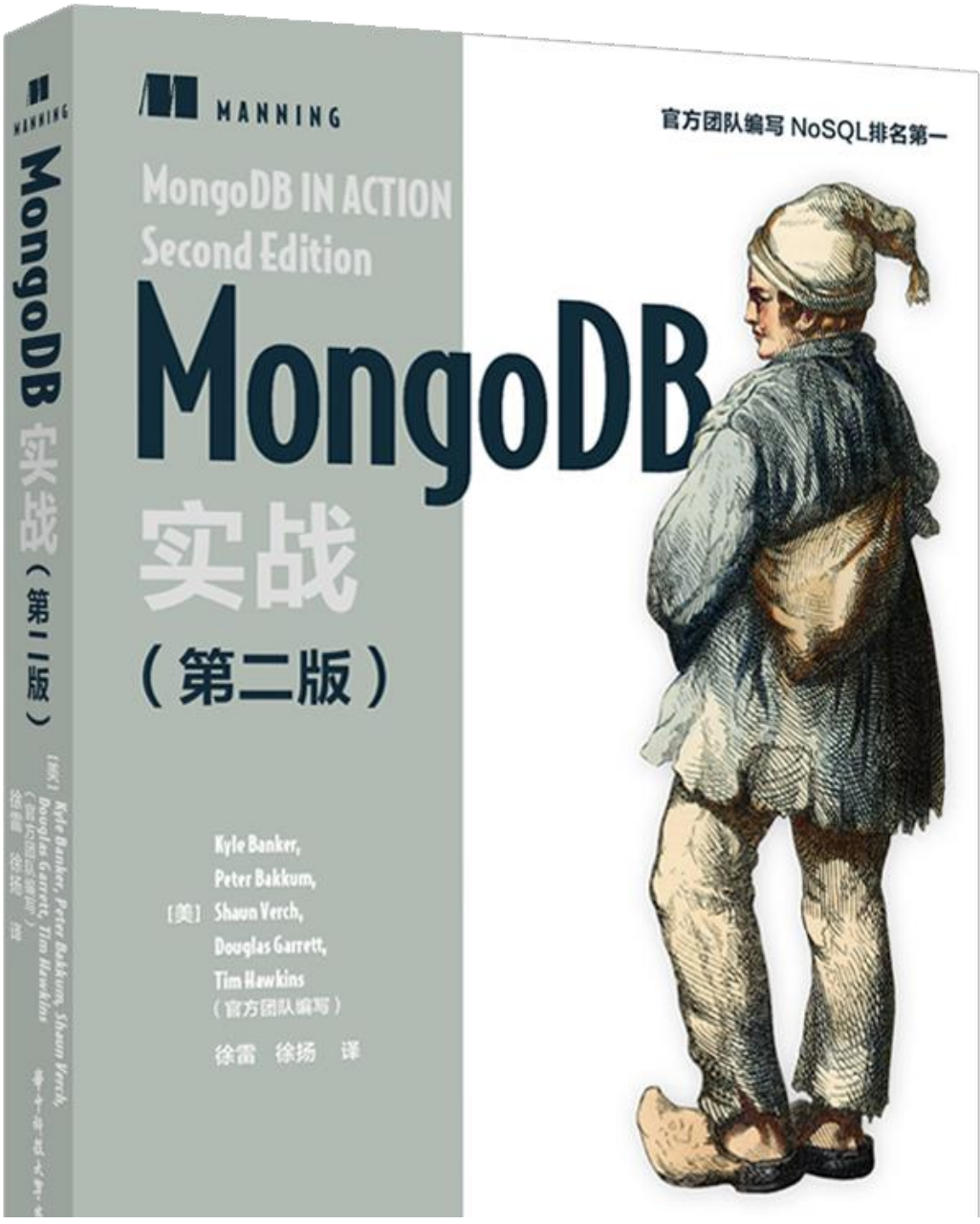
官方团队出品　NoSQL 排名第一 《Mongodb 实战》第 2 版 互联网架构师必备



第 5 章

构建查询

Constructing queries

本章内容

- 查询一种电子商务数据模型
- MongoDB查询语言的详细内容
- 查询选择器及其选项

MongoDB不使用SQL语言，它使用自己的类JSON查询语言来代替这些功能。通过这本书，我们已经对这门语言进行了探索和学习，但现在我们要转向一些耐人寻味的真实例子。在前面的章节我们将会重新引入商务数据模型和目前各种针对它的查询。这些查询包括：_id查找、限定范围、排序和投影。

记住，当我们在学习本章时，MongoDB的查询语言和聚合函数（第6章中讲解）仍然在不断进步，并且每个新版本中都会有新的改进。因此，当Mongodb在持续寻找最佳的方式来完成日常任务的时候，掌握MongoDB查询和聚合并不能涵盖所有问题。本章的例子中，我们将会采用最清晰直接的学习线路。通过本章的学习 ,我们对MongoDB的查询有个良好直观的了解，并且可以把些工具应用到程序schema的设计工作中。

5.1 电子商务查询

E-commerce queries

这一节我们继续探索在上一章中提到的商务数据模型。我们已经为商品，类别，用户，订单和商品评论定义了文档。现在，脑海中应该有了这些文档结构，我们将介绍如何在一个典型的电子商务网站中查询这些实体数据。当然有些查询比较简单。例如，此时，_id查询不应是神秘莫测的。但是我们也会展示一些更复杂的查询模式，包括查询和显示类别的层次结构，

以及提供商品列表视图。此外，我们还会为这些查询寻找可能的索引来提升效率。

5.1.1 产品、类别和评论

大多数电子商务应用至少包含商品和类别这两个基本视图。第一个是商品的首页，此页面会突出显示给出的商品，展示评论，和给出某种意义上的商品类别。第二是商品的列表页，此页面允许用户查看类别层次结构和所选商品类别的视图缩略图。让我们从商品首页开始，两者之中比较简单的方式开始。

设想一下我们商品页面的URL使用了slug关键字（我们在第4章学习的友好链接）。这种情况下，我们使用下面3条查询语句来获取商品页面需要的所有数据。

```
product = db.products.findOne({'slug': 'wheel-barrow-9092'})
db.categories.findOne({'_id': product['main_cat_id']})
db.reviews.find({'product_id': product['_id']})
```

第一个查询语句查找slug为 wheel-barrow-9092的商品。当我们有了商品，我们可以在类别集合使用简单的_Id查询语句在categories集合上查询类别信息。最后，我们可以使用另外一种简单的查询查找关于商品的所有评论。

FINDONE 与 FIND 查询

我们会发现前两种使用 findOne 方法 ,但是最后一个查询使用了 find 方法。所有的 MongoDB 驱动都提供了这两种方法 ,因此这两种方法之间的区别值得探讨。比如在第三章中的讨论 ,find 方法返回一个游标对象，而 findOne 方法返回一个文档。findOne 方法类似下面这样，当使用限制时它也会返回游标：

```
db.products.find({'slug': 'wheel-barrow-9092'}).limit(1)
```

如果我们想查询单个文档，当文档存在时findOne方法会返回文档。如果我们需要返回多个文档，则使用find方法。在程序的某些地方我们可能需要遍历游标。

如果我们使用findOne在数据库中查询匹配多个项目，它会返回在自然排序文档集合中的第一个文档。大多数情况下（普通集合并不总是如此）这个顺序就是文档以被插入到集合的顺序，而对于盖子集合，顺序总是如此。如果我们希望得到多个文档结果，我们尽可能使用find查询或者显式对结果进行排序。

现在再次查看商品查询页面。是否有任何不妥？商品评论查询看起来比较宽松，你是对的。这种查询可以返回指定商品的所有评论，当商品有数百条评论的时候，这样做就不太严谨。

忽略，限制和排序查询选项

大多数程序会实现分页显示评论，为此 MonggoDB 提供了 skip 和 limit 选项。我们可以使用这些选项进行分页查询和显示评论文档：

```
db.reviews.find({'product_id': product['_id']}).skip(0).limit(12)
```

注意，我们通过在find的返回值结果集上调用skip与limit方法来设置这些参数。这与我们通常，甚至其他MongoDB驱动中看到的模式不同，所以可能会造成困惑。它们在查询被调用后出现，但是排序和限制参数发送至查询并有MongoDb服务器处理。这种语法模式被称为方法链（method chaining），旨在更容易地构建查询语句。

如果想要以某个特定顺序显示评论，这意味着我们要对查询结果进行排序。如果要对每个评论收到有效投票进行排序，可以很方便地指定排序条件：

```
db.reviews.find({'product_id': product['_id']}).
                    sort({'helpful_votes': -1}).
                    limit(12)
```

总之，这个查询告诉 MongoDB 返回评论中最有效投票前 12 条，并按降序排列。

现在，现在已经掌握的了 skip, limit, 和 sort 关键字，我们需要决定是否分页放。因此，我们可以发出一个计数查询。然后使用计数查询结果和评论页数结合得到我们需要的分页结果。

这样商品页面的分页查询就完成了：

```
page_number = 1
product = db.products.findOne({'slug': 'wheel-barrow-9092'})
category = db.categories.findOne({'_id': product['main_cat_id']})
reviews_count = db.reviews.count({'product_id': product['_id']})
reviews = db.reviews.find({'product_id': product['_id']}).
                            skip((page_number - 1) * 12).
                            limit(12).
                            sort({'helpful_votes': -1})
```

在JavaScript shell中调用skip，limit和sort的顺序并不重要。

这些查找应该使用索引。我们已经看到slug标签拥有唯一索引，因为它们作为备用主键，并且我们知道所有的_id字段都会自动获取一个标准合集的唯一索引。但是，外键引用字段包含索引也是非常重要的原则。此时，评论合集将会包含user_id和product_id字段。

商品列表页面

商品首页面查询完成以后，现在我们可以转到商品列表页面。这个页面展示选定类别的可浏览的商品列表。商品总类和分类的链接也将展示在页面上。

商品列表页面通过类别定义；因此，页面请求将包含类别的标签slug：

```
page_number = 1
```

```
category = db.categories.findOne({'slug': 'gardening-tools'})
siblings = db.categories.find({'parent_id': category['_id']})
products = db.products.find({'category_id': category['_id']})
                          .skip((page_number - 1) * 12)
                          .limit(12)
                          .sort({'helpful_votes': -1})
```

分类Siblings是不具有相同的总类ID的任何类别，因此分类查询简单明了。因为商品包含了类别ID数组，在一个给定的类别中查找所有商品，这种查询也非常简单。我们可以想出其他替代的排序方法（比如名字，价格等等）。对于这种情况，我们可以更改排序字段。

考虑这些排序是否有效很重要。我们可以依靠索引来处理排序，但是当我们添加了更多的排序选项，索引数目的增加，维护索引工作就变得比较复杂，因为每个索引的写入成本比较高。我们将在第八章中进一步讨论，但是我们根据这些信息这些来权衡利弊。

商品列表页面有个默认显示情况，当访问根类别时，不会显示任何商品。针对查询的类别集合的父 ID 为 null，搜索对应的是根类别：

```
categories = db.categories.find({'parent_id': null})
```

5.1.2 用户和订单

上一节中的查询仅限于_id查找和排序。再看一下用户和订单，我们要深入更多内容，因为我们想生成订单的基本报告。这些查询例子来自于第四章列表4.1（商品）和4.4（用户）。

让我们从简单的开始:用户登录验证。提供用户名和用户密码让用户登录应用。因此，我们希望经常看到下面的查询：

```
db.users.findOne({
    'username': 'kbanker',
    'hashed_password': 'bd1cfa194c3a603e7186780824b04419'})
```

如果用户名存在并且密码正确。我们会得到整个用户文档：否则，不返回任何东西。这种查询时可以接受的，如果我们只想检查用户名是否存在却要返回整个用户文档？我们可以使用投影限制返回字段。

```
db.users.findOne({
  'username': 'kbanker',
  'hashed_password': 'bd1cfa194c3a603e7186780824b04419'},
  {'_id': 1})
```

在 JavaScript shell 中投影需要传递一个额外的参数：想得到的一个哈希字段设置他们的值为1.我们将在第 5.2.2 节中深入探讨投影。如果你已经熟悉 SQL 和 RDBMS，这就是 SELECT* 和 SELECT ID 的不同之处。现在的应答结果只包含了文档的_id 字段：

```
{ "_id": ObjectId("4c4b1476238d3b4dd5000001") }
```

用户部分匹配查询

我们可能想要用其他方式查询用户集合，比如搜索名称。通常希望用单个字段进行查找，如last_ name：

```
db.users.find({'last_name': 'Banker'})
```

这种方法很有效，但是有完全匹配的限制搜索。其一，你可以不知道如何拼写给定的用户名。这种情况下，可以使用部分匹配查询方式。假设我们知道用户的姓由字母Ba开头。MongoDB允许使用正则表达式查询：

```
db.users.find({'last_name': /^Ba/})
```

正则表达式/^Ba/可以读作“行的开头是字母B，紧跟着是字母a”。像这样的前缀搜索可以使用索引，但并不是所有的正则表达式都可以使用索引。

查询特定范围

当对用户进行营销时，最想要的是特定范围内的用户。例如，想查询居住在曼哈顿的用户时，就查询这个范围内用户的邮政编码。

```
db.users.find({'addresses.zip': {'$gt': 10019, '$lt': 10040}})
```

回想一下，每个用户文档包含一个或多个地址的文档。这种查询将匹配用户文档，如果这些地址的邮政编码属于给定的范围。可以使用$gte（大于）和$lt（小于） 运算符来定义范围。为使这种查询高效，我们可能想要在addresses.zip中定义索引。

在下一章中，我们会看到更多查询数据的例子，接下来，我们会深入学习使用MongoDB聚合函数对数据进行查询的知识。

但是掌握了目前为止本章的知识，随后会深入看下MongoDB的查询语言，特别解释一下通用的语法和操作符。

5.2 MongoDB 查询语言

MongoDB’s query language

现在是时候探索MongoDB查询语言精华知识的时候了。我们已经学习了一些真实的查询实例。本节旨在为MongoDB查询功能做个更全面的解释参考。如果是第一次学习关于MongoDB查询的知识，可能更容易理解本节内容。当我们需要为程序编写更高级的复杂查询时，大家

可以重新复习本节内容。

5.2.1 查询条件和选择器

查询条件允许我们使用单个或者多个查询选择器来指定查询结果。MongoDB提供多种可能的选择器。本节进行概述。

选择器匹配

指定查询最简单的一种方式是使用查询器的键值对字面要匹配我们要查找的文档。下面是几个简单的例子代码：

```
db.users.find({'last_name': "Banker"})
db.users.find({'first_name': "Smith", birth_year: 1975})
```

第二种查询读作，“查出first_name为Smith且出生在1975年所有用户。”注意，无论什么时候传递几个键值对，两个条件必须匹配；查询条件功能可以为布尔值AND。如果想要展示一个布尔值OR，参阅下面的章节，布尔值运算符。

在MongoDB中所有的文本字符串匹配都是区分大小写的。如果需要不区分大小写的匹配，考虑使用正则表达式项（本章后面解释，当我们探讨使用i正则表达式标识）或者探讨使用第九章介绍的文本搜索。

范围

我们可能经常需要查询值在一定跨度范围内的文档。在大部分语言中，使用<、<=、> 和 >=。在MongoDB中，我们会得到类似的一组运算符$lt、$lte、$gt和$gte。如我们期望的那样，我们在本书中会经常使用这种运算符。表5.1展示了最常使用的范围查询运算符。

表 5.1 范围查询运算符摘要

运算符	描述
$lt	小于
$gt	大于
$lte	小于等于
$gte	大于等于

初学者经常努力组合这些运算符。一个常见的错误就是重复搜索键值：

```
db.users.find({'birth_year': {'$gte': 1985}, 'birth_year': {'$lte': 2015}})
```

上述查询只会使用最后一个条件。我们可以按照下面的方式编写正确的查询：

```
db.users.find({'birth_year': {'$gte': 1985, '$lte': 2015}})
```

我们也应该了解它们如何处理不同的数据类型。范围查询匹配值仅仅当他们含有相同类型的值才进行比较。例如，假设你有一个下面文档的集合：

```
{ "_id" : ObjectId("4caf82011b0978483ea29ada"), "value" : 97 }
{ "_id" : ObjectId("4caf82031b0978483ea29adb"), "value" : 98 }
{ "_id" : ObjectId("4caf82051b0978483ea29adc"), "value" : 99 }
{ "_id" : ObjectId("4caf820d1b0978483ea29ade"), "value" : "a" }
{ "_id" : ObjectId("4caf820f1b0978483ea29adf"), "value" : "b" }
{ "_id" : ObjectId("4caf82101b0978483ea29ae0"), "value" : "c" }
```

就可以使用下面的查询：

```
db.items.find({'value': {'$gte': 97}})
```

你可能会认为查询应该返回六个文档，因为字符串在数值上等值于整数的97，98和99。但这种情况并非如此。由于MongoDB是无schema的，查询返回的整数结果仅仅是因为提供的标准本身是一个整数。如果你想得到字符串结果，必须使用字符串查询代替：

```
db.items.find({'value': {'$gte': "a"}})
```

我们不需要担心这种类型的限制，只要我们不为同一集合的主键存储多种类型即可。这种一种非常好的通用实践经验，我们应该遵守它。

设置运算符

三种查询运算符—$in, $all, 和$nin—把一个或多个值的列表作为它们的谓语，因此被称为集合运算符。$in返回文档，如果给定的值匹配搜索关键字。我们可以使用这些运算符返回属于某个类别集合的所有商品。表5.2展示了这些集合查询运算符。

表 5.2　集合操作符总结

操作符	描述
$in	如果任意参数在引用集合里，则匹配
$all	如果所有参数在引用集合里且被使用在包含数组的文档中，则匹配
$nin	如果没有参数在引用的集合里，则匹配

如果有下面这些类别IDs列表

```
[
  ObjectId("6a5b1476238d3b4dd5000048"),
  ObjectId("6a5b1476238d3b4dd5000051"),
  ObjectId("6a5b1476238d3b4dd5000057")
```

```
]
```

对应割草机，手工具和工作服类别，要查询找到所有属于这些类别的商品，代码如下所示：

```
db.products.find({
    'main_cat_id': {
      '$in': [
        ObjectId("6a5b1476238d3b4dd5000048"),
        ObjectId("6a5b1476238d3b4dd5000051"),
        ObjectId("6a5b1476238d3b4dd5000057")
      ]
    }
  })
```

另一种思考$in运算符的方式是，把它类比某个单一属性的布尔或OR运算。这种表达方式，前面的查询可以理解为，“找出类别是割草机或手工具或工作服的所有商品。”

注意如果需要通过使用布尔值OR组合多个属性，可以使用$or运算符，下一节中详细叙述。

- `$in`频繁与`ID`列表一起使用
- `$nin`（`not in`）仅返回一个没有给定元素匹配项的文档。我们可以使用`$nin`查找既不是黑色也不是蓝色的所有商品：

  ```
  db.products.find({'details.color': {'$nin': ["black", "blue"]}})
  ```

- `$all`匹配如果每个给定元素匹配搜索关键字。如果想找到所有标签为礼品`gift`和花园`garden`的商品，`$all`是一个很好的选择：

  ```
  db.products.find({'tags': {'$all': ["gift", "garden"]}})
  ```

当然，这种查询的意义仅仅是用标签属性存储一组数组项，比如这样：

```
{
  'name': "Bird Feeder",
  'tags': [ "gift", "birds", "garden" ]
}
```

选择性是一种使用索引缩小查询结果的能力。问题是$ne 和$nin运算符不是选择的。因此，当使用集合运算符，记住$in 和$all可以利用索引，但$nin不能因此需要集合扫描。如果我们使用$nin，尝试使用它结合另一个使用索引的查询项。最好，找到不同的方式来表示这种查询。也可能存储的属性有与$nin查询等价的条件。例如，你通常会如此查询{timeframe: {$nin: ['morning', 'afternoon']}}，其实可以使用更直接的条件来表示如{timeframe: 'evening'}。

布尔运算符

MongoDB 布尔运算符包括`$ne`、`$not`、`$or`、`$and`、`$nor` 和 `$exists`。表5.3总结了布尔运算符。

表 5.3　布尔运算符总结

操作符	描述
$ne	不匹配参数条件
$not	不匹配结果
$or	有一个条件匹配就成立
$nor	所有条件都不匹配
$and	所有条件都匹配
$exists	判断元素是否存在

$ne，不等于运算符，和我们期望的一样。在实践中，使用最佳方式是结合其他的运算符；否则，因为它不会使用索引，会造成性能底下。比如，可以使用$ne查找Acme公司制造的但是没有gardening标签的所有商品：

```
db.products.find({'details.manufacturer': 'Acme', tags: {$ne: "gardening"} })
```

$ne适用于关键字指向的单一值或者数组的情况，如例子匹配的标签tags数组所示。

$ne匹配指定的值，而$not不匹配MongoDB的运算符或者正则表达式的查询结果。大多数查询操作已经支持否定匹配形式($in和$nin, $gt和$lte等)；$not非常有用，因为它可以过滤不符合表达式条件的结果。思考下面的例子：

```
db.users.find({'age': {'$not': {'$lte': 30}}})
```

如我们预期的那样，这种查询返回年龄age大于30岁的用户文档。当然也返回没有age字段的文档，这种情况下，区别于使用$gt运算符。

$or表示逻辑判断两个值使用两个不同的关键字。有个重点指出：如果可能的值的范围是相同的关键字，使用$in替代。查找所有蓝色或者绿色的商品，代码如下：

```
db.products.find({'details.color': {$in: ['blue', 'Green']}})
```

要找到是蓝色或者由Acme公司生产的所有商品，就用到$or：

```
db.products.find({
    '$or': [
      {'details.color': 'blue'},
      {'details.manufacturer': 'Acme'}
    ]
  })
```

$or接受查询选择器数组，每个选择器可以是任意复杂和可能自身包含其他查询运算符的选择器。$nor工作原理和$or大致相同，但逻辑仅当所有查询选择器为假的时候才为真。

与$or一样，$and运算符也接受一组数组查询选择器数组。如果需要MongoDB解释包含多个并且关系的关键字的查询选择器时，仅当没有更简单的方式表示AND且关系时才应该使用$and运算符。例如，假如我们要查找所有标签包含礼品或者节日且或者是园艺或者绿化的商品。表示该查询的方法只有组合两个$in查询语句：

```
db.products.find({
    $and: [
      {
        tags: {$in: ['gift', 'holiday']}
      },
      {
        tags: {$in: ['gardening', 'landscaping']}
      }
    ]
  })
```

查询包含特定关键字的文档

这一节我们讨论学习最后一个运算符$exists。这个运算符是必须的，因为集合不会强制遵守固定的数据架构模式，所以有时候我们需要一种方式查询包含特定关键字的文档。记得我们计划使用每个商品的details属性来存储自定义字段。例如，可能我们存储一个color颜色字段在details属性中。但是如果所有商品的一个子集指定了一个颜色集合，我们可以查询那些不同颜色的商品，如下所示：

```
db.products.find({'details.color': {$exists: false}})
```

相反的查询也是可能的：

```
db.products.find({'details.color': {$exists: true}})
```

这里我们可以检查文档中是否存在该字段。甚至如果字段存在，它还是可以设置为null。取决于我们的数据和查询，我们可能也想筛选这些值。

匹配子文档

本书中的一些电商数据模型实体拥有指向的单一嵌入对象的外键。商品的details属性是个很好的例子。这里是相关文档的一部分数据， JSON格式表示：

```
{
  _id: ObjectId("4c4b1476238d3b4dd5003981"),
  slug: "wheel-barrow-9092",
  sku: "9092",
  details: {
    model_num: 4039283402,
    manufacturer: "Acme",
    manufacturer_id: 432,
```

```
    color: "Green"
  }
}
```

我们查询此类对象使用分离有关关键字a. (dot)。例如，如果想找到Acme公司生产的所有商品，我们可以这样查询：

```
db.products.find({'details.manufacturer': "Acme"});
```

这种查询可以指定任意深度，例如做了修改后的表示形式如下：

```
{
  _id: ObjectId("4c4b1476238d3b4dd5003981"),
  slug: "wheel-barrow-9092",
  sku: "9092",
  details: {
    model_num: 4039283402,
    manufacturer: {
      name: "Acme",
      id: 432
    },
    color: "Green"
  }
}
```

查询选择器中的关键字包含两个点（.）：

```
db.products.find({'details.manufacturer.id': 432});
```

但除了针对单一子文档属性匹配，还可以匹配整个项目。例如，想象我们正在使用MongoDB存储股票市场的职位。为了节约空间，放弃使用标准对象的ID，替代使用股票符号和时间戳组成的复合关键字。这是一个典型例子代表文档：

```
{
  _id: {
  sym: 'GOOG',
    date: 20101005
  }
  open: 40.23,
  high: 45.50,
  low: 38.81,
  close: 41.22
}
```

然后我们可以找到GOOG，2010年10月2号的总结，使用下面_ID查询：

```
db.ticks.find({'_id': {'sym': 'GOOG', 'date': 20101005}})
```

非常重要的是要认识到，像这样查询匹配一个实体对象，将执行严格的逐字节比较工作，这意味着关键字的顺序非常重要。下面的查询不同，也不会匹配相同的文档：

```
db.ticks.find({'_id': {'date': 20101005, 'sym': 'GOOG'}})
```

关键字的顺序通过shell命令在JSON文档保存，这不一定适用于其他语言，更安全的方式是假设不会保留顺序。例如，哈希在Ruby1.8不会保留顺序。要在Ruby1.8中保留关键字顺序，必须使用类的对象BSON::OrderedHash代替：

```
doc = BSON::OrderedHash.new
doc['sym'] = 'GOOG'
doc['date'] = 20101005
@ticks.find(doc)
```

一定要检查使用的语言是否支持有序字典，如果不支持，这种语言的MongoDB驱动程序总是会提供有序的备选方案。

数组

数组给文档模型更多的支持。正如我们在电子商务网站例子中看到的，数组用于存储字符串列表，对象IDs，甚至其他的文档。数组提供了丰富又易于理解的文档；按理说，MongoDB会易于查询和索引数组类型的数据。事实上是：最简单的数组查询看起来与其它查询其他类型的文档一样，比如表5.4所示。

表 5.4　数组操作符

操作符	描述
$elemMatch	如果提供的所有词语在相同的子文档中，则匹配
$size	如果子文档数组大小与提供的文本值相同，则匹配

让我们看看数组的实际操作。再次使用商品标签tags。这些标签表示为一个简单的字符串列表：

```
{
  _id: ObjectId("4c4b1476238d3b4dd5003981"),
  slug: "wheel-barrow-9092",
  sku: "9092",
  tags: ["tools", "equipment", "soil"]
}
```

用标签"soil"查询商品是很简单的，和查询单一文档值的语法相同：

```
db.products.find({tags: "soil"})
```

重要的是，这种查询可以利用索引上的标签字段。如果建立所需索引并使用explain()运行查询，可以看到使用了B-tree游标：

```
db.products.ensureIndex({tags: 1})
```

```
db.products.find({tags: "soil"}).explain()
```

当我们需要对数组查询做更多的控制时，我们可以在特定位置的数组使用点符号进行查询。下面的例子是如何限制上一个查询例子中的第一个商品标签：

```
db.products.find({'tags.0': "soil"})
```

这种查询方式不会有多大的意义。想象一下如果我们处理用户地址，就可能需要表示这些数字的子文档：

```
{
  _id: ObjectId("4c4b1476238d3b4dd5000001")
  username: "kbanker",
  addresses: [
    {
      name: "home",
      street: "588 5th Street",
      city: "Brooklyn",
      state: "NY",
      zip: 11215
    },
    {
      name: "work",
      street: "1 E. 23rd Street",
      city: "New York",
      state "NY",
      zip 10010
    },
  ]
}
```

我们可能规定数组的第0个元素始终是用户的主要配送地址。然而，查找配送地址始终在New York纽约的所有用户，我们可以再次指定第0个位置，并结合一个点打炮目标state字段：

```
db.users.find({'addresses.0.state': "NY"})
```

很容易理解，写生产代码时，不应该指定元素。

很容易忽略此位置并单独指定一个字段。如果列表中的任意地址属于New York纽约，下面的查询就会返回一个用户文档：

```
db.users.find({'addresses.state': "NY"})
```

和以前一样，我们会想要点字段的索引：

```
db.users.ensureIndex({'addresses.state': 1})
```

注意，无论字段指向子文档或者子文档的数组都是使用相同的点符号。点符号是强大的并且一致性令人放心。但对单个或多个自对象数组属性查询时出现歧义。例如，假如我们想获取

家庭住址在纽约的用户列表。你能否想出一种方式表示此查询？

```
db.users.find({'addresses.name': 'home', 'addresses.state': 'NY'})
```

这个查询的问题是字段引用并不局限于一个单一的地址。换句话说，这种查询匹配的地址指定一个是家，一个是纽约New York，但是我们想要的两个属性适应于同一个地址。幸运的是，有个查询运算符可以做到这点。同一个子文档限制多个条件，我们使用$elemMatch运算符。可以合理满足这种查询：

```
db.users.find({
    'addresses': {
      '$elemMatch': {
        'name': 'home',
        'state': 'NY'
    }
  }
})
```

从逻辑上讲，仅当我们需要匹配多个或者更多属性的子文档才使用`$elemMatch`。

通过大小查询数组

剩下的数组运算符只有$size运算符了。这种运算符允许通过自身大小查询数组。例如，如果我们想查找拥有三个地址的所有用户，我们可以使用$size运算符：

```
db.users.find({'addresses': {$size: 3}})
```

像这样写，$size运算符不使用索引和限制来进行准确匹配（我们不能指定sizes的范围）。因此，如果我们需要基于数组的大小执行查询，我们可以在文档自身的属性里缓存size并手动更新它。比如，我们可以考虑添加address_length字段到用户文档。然后我们在这个字段上建立一个索引，发出范围并精确匹配查询我们的需求。一种可能的解决方案是使用聚合框架，我们将在第6章学习。

JavaScript 查询运算符

目前为止如果还不能使用这些工具表示出我们想要的查询，这时候可能需要编写一些JavaScript代码。我们可以使用特定的$where运算符传递JavaScript表达式进行查询，总结：

- `$where` 执行任意JavaScript来选择文档。

在JavaScript范围内，关键字this指向当前文档。让我们实战操作一个例子：

```
db.reviews.find({
    '$where': "function() { return this.helpful_votes > 3; }"
  })
```

还有一种简单表达式的缩写形式：

```
db.reviews.find({'$where': "this.helpful_votes > 3"})
```

这种查询工作，我们可能不会想去使用它，我们可以简便的使用其他查询运算符表达它。问题是JavaScript表达式不能使用索引，并且带来大量开销，因为他们必须在JavaScript解释器的上下文中评估并且是单线程的。基于这些原因，只有当我们使用其他查询运算符不能表示

查询时才应该使用JavaScript查询。如果确实需要使用JavaScript，那么使用JavaScript表达式至少结合一个其他查询运算符。其他查询运算符会消减结果集，减少必须加载到JavaScript上下文的文档数目。让我们看看一个简单的例子，这样做有何意义。

想象一下，我们计算过每个用户的评价可靠性因素。这本质上还是一个整数，当用户的评分增加，结果是更标准化的评价。假如我们想查询特定用户的评论并且返回大于标准评分高于3的。下面的代码展示如何查询：

```
db.reviews.find({
    'user_id': ObjectId("4c4b1476238d3b4dd5000001"),
    '$where': "(this.rating * .92) > 3"
  })
```

这个查询要满足这两条：一是它使用标准查询索引`user_id`字段，二是利用一个绝对超过其他查询运算符能力的JavaScript表达式。记住有时候使用聚合框架更加简单。

除了意识到随之而来的性能损失，也必须注意JavaScript注入攻击的可能性。注入攻击称为可能每当允许用户直接输入代码到JavaScript查询。一个例子就是当用户直接在排序查询中提交了一个用户表单和值。如果用户设置值的属性或者数值，这种查询是不安全的：

```
@users.find({'$where' => "this.#{attribute} == #{value}"})
```

这种情况下，属性的值和数值的值被插入成字符串，然后认定为JavaScript.。这种方式是危险的，因为用户可能会发送包含值中有JavaScript代码，使他们获取其他数据集合。这将导致严重的安全漏洞，恶意用户可能看到其他用户的数据。一般情况，我们总是要假设我们的用户可能会发送恶意数据和相应计划。

正则表达式

在本章开头，我们就接触并使用过查询正则表达式。在那个例子中，我们使用了前缀表达式`/^Ba/`，查找以`Ba`开头的姓，而且，这种查询使用了索引。事实上，还有更多的可能性。`MongoDB`使用Perl兼容正则表达式编译(PCRE; http://mng.bz/hxmh)，对正则表达式提供了全面的支持。

以下是`$regex`运算符的概述。

- `$regex` 匹配元素对应提供给regex（正则表达式）项。

除了刚刚描述的前缀样式查询，正则表达式查询不能使用索引并且执行的时间比大多数选择器更长。我们建议节制性地使用它们。以下代码用于查询出包含最好或者最差词语的用户评

论文本。注意我们要使用i正则表达式标签以指示不区分大小写。

```
db.reviews.find({
    'user_id': ObjectId("4c4b1476238d3b4dd5000001"),
    'text': /best|worst/i
  })
```

使用不区分大小写标签会带来不利后果：它拒绝使用任何索引。在MongoDB中总是区分大小写。如果我们想在大量文档中使用区分大小写搜索，就需要使用提供新建文本搜索功能的2.4版本或者更高版本，或者集成外部的文本搜索引擎。请参阅第9章的MongoDB搜索功能的说明。

如果使用的语言含有自带正则表达式类型，就可以使用自带正则表达式对象执行查询。我们可以像这样在Ruby中表示相同的查询：

```
@reviews.find({
    :user_id => BSON::ObjectId("4c4b1476238d3b4dd5000001"),
    :text => /best|worst/i
  })
```

尽管本地定义了正则表达式，但还是要在MongoDB服务上做出评估。如果从一个本地不支持的正则表达式类型环境中查询，就可以使用特殊的`$regex`和`$options`运算符。通过命令使用这些运算符，我们就可以通过另一种方式表示查询：

```
db.reviews.find({
    'user_id': ObjectId("4c4b1476238d3b4dd5000001"),
    'text': {
      '$regex': "best|worst",
      '$options': "i"}
  })
```

MongoDB是一个区分大小写的系统，当我们使用正则表达式时，除非使用`/i`修饰符（即`/best|worst/i`），否则搜索将与被搜索的字段大小完全匹配。有一点要注意，如果使用了`/i`，那么它会禁止使用索引。如果想做到索引内容不区分大小写搜索文档中的字符串、字段，那么就存储重复字段的内容专为强制小写搜索，或者使用MongoDB文本搜索功能，它可以与其他查询结合且提供了索引不区分大小写的搜索。

其他查询运算符

两个或更多的运算符不容易归类，但它们拥有自己的部分。第一个是$mod，可以允许查询文档匹配一个给定的模运算符；第一个是$type，匹配BSON类型的值。在表5.5中包含两者的详情。

表 5.5　其他运算符总结

运算符	描述
`$mod[(quotient),(result)]`	如果元素除以除数符合结果则匹配
`$type`	如果元素的类型符合指定的 BSON 类型则匹配
`$text`	允许在建立文本索引的字段上执行文本搜索

举个例子：使用下面的查询语句，`$mod`允许查找所有被3整除的订单并汇总：

```
db.orders.find({subtotal: {$mod: [3, 0]}})
```

我们可以看到$mod运算符接受一个含有两个值的数组。第一个是除数，第二个预期的余数。查询技术上读做，“找到当总数除以3返回的余数为0的所有文档”。这是个人为构造的例子，但是演示了这个想法，如果最终使用了$mod运算符，记住它不会使用索引。

第二个运算符`$type`，按其BSON类型值匹配。不建议在同一个集合使用相同的字段存储多个类型，如果出现这种情况，那么有个查询运算符可用来根据类型进行测试。

表5.6所示为MongoDB中用于与每个元素类型相关联的数字类型。其中例子展示出类型的成员如何出现在JavaScript控制台。例如，其他MongoDB驱动程序可能用不同的方式存储等同的`ISODate`对象。

表 5.6　BSON 类型

BSON 类型	$type 数字	示例
Double	1	`123.456`
String (UTF-8)	2	`"Now is the time"`
Object	3	`{ name:"Tim",age:"myob" }`
Array	4	`[123,2345,"string"]`
Binary	5	`BinData(2,"DgAAAEltIHNvbWUgYmluYXJ5")`
ObjectId	7	`ObjectId("4e1bdda65025ea6601560b50")`
Boolean	8	`true`
Date	9	`ISODate("2011-02-24T21:26:00Z")`
Null	10	`null`
Regex	11	`/test/i`
JavaScript	13	`function() {return false;}`
Symbol	14	`Not used; deprecated in the standard`
Scoped JavaScript	15	`function (){return false;}`
32-bit integer	16	`10`
Timestamp	17	`{ "t" : 1371429067,`

		"i" : 0 }
64-bit integer	18	NumberLong(10)
Maxkey	127	{"$maxKey": 1}
Minkey	255	{ "$minKey" : 1}
Maxkey	128	{"maxkey" : { "$maxKey" : 1 }}

在表5.6中有一对元素值得一提。maxkey 和 minkey用于在索引中插入相同的最大或者最小值的虚拟值。这意味着它可以强制文档排序为索引的第一个或最后一个项目。添加一个字段“aardvark”到集合中会强制文档排序在前面，现在已经不需要这么做了。大多数语言的驱动程序具有添加minkey或者maxkey类型的方法。

JavaScript的作用域和JavaScript在表中看起来相同，这仅仅是因为控制台不能展示作用域，这是键值对提供的JavaScript代码片段的字典。作用域指根据上下文执行函数。另一方面，从该函数可以看到变量在作用域字典中的定义和使用它们的执行过程。

最后，符号类型没有代表。因为在大多数语言中它不会被使用，或者仅当语言有独特的类型“keys.”才会被使用。比如，在Ruby中"foo" 和 :foo是不同的，后者：foo是一种符号

Rubyde驱动程序存储任何关键字为符号类型。

BSON符号类型

就查询而言，MongoDB服务对待BSON符号类型和对待字符串一样，只有当文档检索到一个独特的符号类型时，才会映射到相应的语言类型。注意，符号类型在最新的BSON规范中(http://bsonspec.org)被弃用，可能随时被丢弃。不管用什么语言写的数据，可以在任何其他语言使用BSON检索它。

5.2.2 查询选择

所有的查询都需要查询选择器。即使查询为空，查询选择器实际上还是定义了一个空查询。当发出查询时，就有允许进一步限制结果集合的各种查询选项供选择。下面让我们看看这些选项。

映射

我们可以使用映射来选择子集的字段，用来返回每个文档的查询结果合集。尤其在有大文档的情况下，使用映射可以最小化网络延迟和反序列化。对$slice这个唯一运算符，下面进行了总结。

- $slice 选择返回文档的子集。

映射通常定义为返回字段合集：

```
db.users.find({}, {'username': 1})
```

这种查询返回只包含字段username和_id的用户文档，排除其他所有字段这是特殊情况，默认情况下一直包含这2个字段。

在一些情况下，我们可能想指定排除相反的字段。这本书的用户文档包含了送货地址和付款方式，但我们通常不需要这些。要排除它们，就添加这些字段到映射值：

```
db.users.find({}, {'addresses': 0, 'payment_methods': 0})
```

在映射中，尽管_id字段是一种特殊情况，但也是要么包含要么排除。我们可以用同样的方式排除_id字段，通过设置0值到映射文档。

注意，除了包含和排除字段，我们可以将返回值的范围存储到一个数组。例如，我们可能要在商品自身的文档内存储商品的评论。这种情况下，我们坚持对评论分页，这时就可以使用$slice运算符。返回前12条评论或者返回最后5条评论，就可这样使用$slice：

```
db.products.find({}, {'reviews': {$slice: 12}})
db.products.find({}, {'reviews': {$slice: -5}})
```

$slice可以取两个元素数组，其中的值分别表示返回跳过和限制的页数。下面是如何跳过前24条评论并且限制评论为12条的代码：

```
db.products.find({}, {'reviews': {$slice: [24, 12]}})
```

最后注意，使用$slice后不会阻止其他返回字段。如果想要在文档中限制其他字段，就必须明确这样做。例如，这是修改以前的查询，仅返回评论和评分：

```
db.products.find({}, {'reviews': {'$slice': [24, 12]}, 'reviews.rating': 1})
```

排序

当我们刚接触这一章时，我们只对任何查询结果的一个或多个字段按照升序或者降序排序。

简单地通过评分对评论排序，如由最高分到最低分降序排列，像这样：

```
db.reviews.find({}).sort({'rating': -1})
```

当然，也可以按照有益的评论和评分排序：

```
db.reviews.find({}).sort({'helpful_votes':-1, 'rating': -1})
```

对这样的复合排序，顺序无关紧要。请注意，JSON通过命令排序。因为Ruby哈希是无序的，所以我们在Ruby数组中注明数组排序顺序，例如：

```
@reviews.find({}).sort([['helpful_votes', -1], ['rating', -1]])
```

在MongoDB中指定排序非常简单，懂得如何建立索引是提升排序效率的关键。我们将在第8章继续学习，如果现在感觉使用排序很困难，则可以跳到前面学习。

跳过 skip 和限制 limit

大家对于skip和limit的语义大家已经非常熟悉，没有什么神秘之处。这些查询选项总是如我们期望的运行。但我们应注意传递大数值skip跳跃（值大于10000），因为这种查询服务要求扫描的文档数等于skip值。假如，想象我们正在对百万的文档按日期降序进行分页排序，每页10个结果。这意味着查询显示50,000th页面将包含50万skip值，这种效率非常低。更好的策略是完全忽略skip并且将范围添加到查询，指示下一个结果设置的位置。因此，这样的查询

```
db.docs.find({}).skip(500000).limit(10).sort({date: -1})
```

变成这样：

```
previous_page_date = new Date(2013, 05, 05)
db.docs.find({'date': {'$gt': previous_page_date}}).limit(10).sort({'date':
-1})
```

第二种查询浏览项目比第一种少。这里有一个潜在的问题是，如果`date`对每个文档不是唯一的，那么相同的文档可能会显示多次。有多种处理解决方案，把解决方案和练习留给读者。

这儿还有另外一套可以执行MongoDB数据的查询类型:地理空间查询,用于索引和检索地理位置和几何数据，经常用于映射和位置感知应用程序（比如地图类或者和位置相关的应用）。

5.3 总结

Summary

查询是Mongodb的关键功能特性,弥补了MongoDB接口的关键一角。一旦阅读完本章的内容，

鼓励大家把查询机制放到测试程序中进行实战操作。如果我们不能确定特定查询运算符组合如何为我们服务，Mongo Shell就是最好的测试工具。