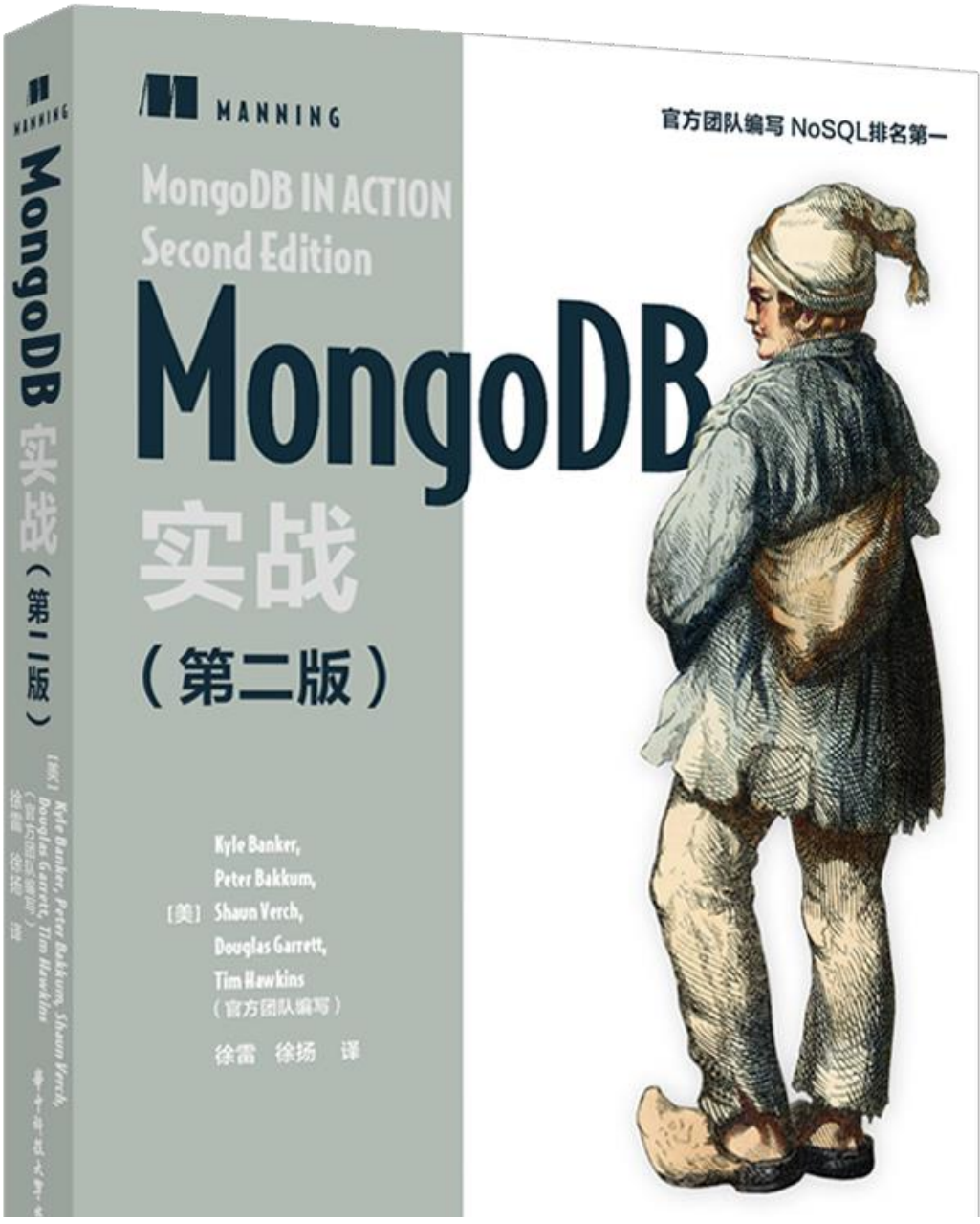
官方团队出品　　NoSQL 排名第一　《Mongodb 实战》第 2 版　互联网架构师必备



第 6 章

聚合查询与统计分析

Aggregation

本章内容

- 电商数据模型上的聚合
- 聚合框架细节
- 性能和限制
- 其他聚合功能

在前面一章我们看了如何使用MongoDB的类JSON查询语言来执行查询操作，比如根据ID、名字或者排序等。本章将会扩展这个主题，包括更多复杂的查询、使用MongoDB的聚合框架。聚合框架是MongoDB的高级查询语言，它允许我们通过转换和合并由多个文档中的数据来生成新的在单个文档里不存在的文档信息。例如，我们可以使用聚合框架根据月份来确定销售、根据产品来确定销售，或者根据用户来确定订单总数。这些对于关系型数据库来说都非常简单，我们可以把MongoDB的聚合框架等价于SQL的`GROUP BY`语句。虽然我们也可以通过MongoDB 的map reduce功能或者使用程序代码计算出这些信息，但是聚合框架允许我们定义一系列文档操作，然后在单个调用里作为数组发送给MongoDB，让我们更容易地完成这些工作。

本章里，我们会展示使用电商数据模型的本书其余部分也会使用的许多例子，然后提供详细的聚合框架操作过程，以及不同的操作选项。在本章结束的时候，我们会有几个聚合框架关键知识点以及如何在电商数据模型里使用它们的对应例子。我们不会介绍聚合电商数据模型操作的细节，因为这就是聚合框架要完成的工作：它提供了比我们期望更多的灵活查询数据的方法。

到目前为止，我们已经设计过数据模型以及支持快速查询的数据库，还有响应式网站性能。聚合框架也可以帮助处理电商网站需要的实时信息，而且还提供了许多想从已有的数据里知道的各种问题的答案，但是这可能需要处理大量的数据。

> **MongoDB 2.6、MongoDB 3.0 中的聚合**
>
> MongoDB聚合框架，第一次在MongoDB 2.2引入，每次新版本发布都会更新。本章包含了MongoDB 2.6的功能，第一次在2014年4月可用；MongoDB 3.0与MongoDB 2.6使用了相同的聚合框架，2.6版本加入了许多更新，以及改善聚合框架功能的操作符。如果你在使用更早期的MongoDB，就应该升级到2.6版本或者以后的版本来运行本章的例子代码。

6.1 聚合框架概览

Aggregation framework overview

为调用聚合框架就要定义一个管道（见图6.1）。聚合管道（aggregation pipeline）里的每一步输出都作为下一步的输入。每一步都在输入文档执行单个操作并生成输出文档。

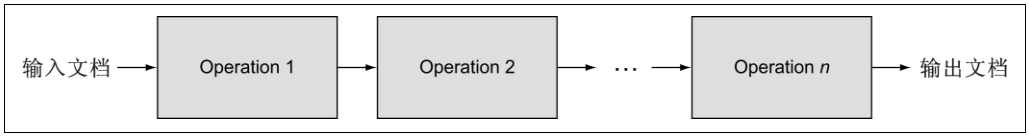


图6.1 聚合管道：一个操作的输出作为下一个操作的输入

聚合管道操作包含下面几个部分：

- `$project`——指定输出文档里的字段（项目化）.。
- `$match`——选择要处理的文档，与`find`()类似。
- `$limit`——限制传递给下一步的文档数量。
- `$skip`——跳过一定数量的文档。
- `$unwind`——扩展数组，为每个数组入口生成一个输出文档。
- `$group`——根据key来分组文档。
- `$sort`——排序文档。
- `$geoNear`——选择某个地理位置附近的文档。
- `$out`——把管道的结果写入某个集合（2.6版新增）。
- `$redact`——控制特定数据的访问（2.6版新增）。

如果阅读过上一章关于构建MongoDB查询的内容，就会发现绝大分部操作符看起来都是相似

的。因为绝大部分聚合框架操作符与MongoDB查询的函数类似。应该确保自己已经很好地理解了5.2节MongoDB查询语言的内容。

这个例子代码定义了一个聚合框架管道，包含一个匹配、组合排序：

```
db.products.aggregate([ {$match: …}, {$group: …}, {$sort: …} ] )
```

这一系列操作如图6.2所示。

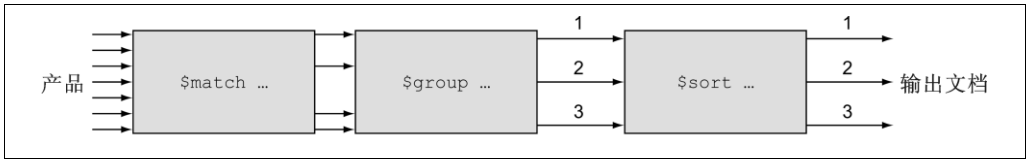


图 6.2　聚合框架管道的例子

如图所示，代码定义了管道，其中

- 整个商品集合传递给`$match`操作，只从输入集合里选择一部分文档。
- `$match`的输出结果传递给`$group`操作符，然后根据`key`来进行排序，提供新的信息，比如求综合和求平均值。
- `$group`的输出结果传递给`$sort`操作符，然后将排序结果返回给最终的结果。

如果对于SQL的`GROUP BY`语句十分熟悉，就会知道它提供了总结信息。表6.1提供了SQL命令和聚合框架操作符对比的详细信息。

表 6.1　SQL 对比聚合框架

SQL 命令	聚合框架操作符
SELECT	$project
	$group functions: $sum, $min, $avg, etc.
FROM	db.collectionName.aggregate(...)
JOIN	$unwind
WHERE	$match
GROUP	BY $group
HAVING	$match

下一节里，我们将会详细了解如何在电商数据模型里使用聚合框架。首先，我们要看一下如何使用聚合框架在Web页面提供商品的摘要信息。然后，我们再看看聚合框架如何应用到其他非Web程序里处理大量的数据，提供有价值的信息，比如找到曼哈顿上城区最高消费者。

6.2 电商聚合例子

E-commerce aggregation example

本节里我们会为电商数据库编写一些聚合查询的例子，回答大家关于使用聚合框架的许多疑问。在继续之前，我们先来复习一下电商网站e-commerce的数据模型。

图6.3所示为电商数据模型。每个大的箱子表示一个数据模型中的集合，包括products、reviews、categories、orders、users。在每个集合里，我们展示了文档的结构，指定任意的数组作为单独的对象集合。例如，在products集合左上角包含商品的信息。对于每个商品，有许多`price_history`对象、许多`category_ids`对象，还有标签。

products 和 reviews之间的线条表示一个商品可以有多个评价，一个评价针对一个商品。我们也可以看到一个评价可能有多个`voter_id`对象关联，表示一个评价可以有多个投票支持。

随着数据模型的增长，这种模型会非常有帮助，因为很难记住多个集合之间的潜在关系，包括每个集合的内部细节。它在帮助确定回答何种问题时也非常有帮助。

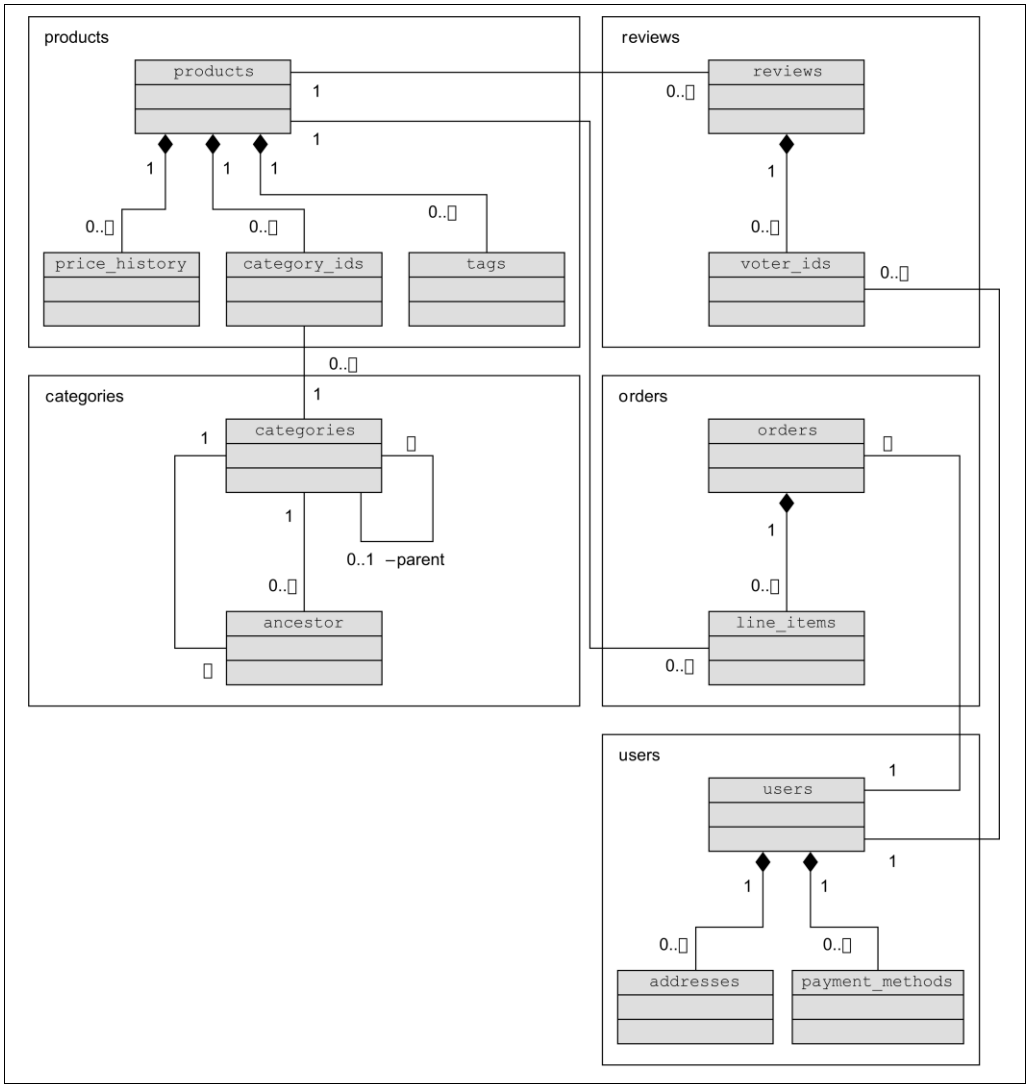


图 6.3　电商集合和关系的数据模型

6.2.1　商品、类别和评价

现在我们来看个如何使用聚合框架来统计商品信息的简单的例子。第5章展示了统计某个商品评价数量的查询方法：

```
product = db.products.findOne({'slug': 'wheelbarrow-9092'})
reviews_count = db.reviews.count({'product_id': product['_id']})
```

我们来看一下如何使用聚合框架。首先，我们来看一下如何统计所有商品的评价总数：

```
db.reviews.aggregate([
  {$group : { _id:'$product_id',
              count:{$sum:1} }}
]);
```

根据 product_id 分组输入文档

计算每个商品的评价数量

这个简单的操作符可以以文档结果的形式返回每个商品的评价信息：

```
{ "_id" : ObjectId("4c4b1476238d3b4dd5003982"), "count" : 2 }
{ "_id" : ObjectId("4c4b1476238d3b4dd5003981"), "count" : 3 }
```

为每个商品输出一个文档

这个例子中，我们有多个文档作为$group操作符的传入参数，但是对于每个_id就只有一个输出文档——此时每个唯一的product_id。$group操作符会为每个输入商品文档添加一个1数字，为了每个product_id的计数统计。结果存放在输出文档的count字段里。

要注意的是，输入文档的字段通常会通过前缀美元符($)指定。在这个例子里，我们定义了_id的值是$product_id，输入文档的product_id字段。

这个例子使用了$sum函数来统计每个商品的评价文档数量，该数量增加1个，则每个product_id的商品评价数量count字段加1。$group操作符支持许多功能，包括求平均值、最小值和最大值以及求和等。

更详细的功能介绍可以在6.3.2小节$group操作符里看到。

接下来，为聚合管道添加一个操作，这样就可以选择唯一的商品来进行计算了：

```
product = db.products.findOne({'slug': 'wheelbarrow-9092'})

ratingSummary = db.reviews.aggregate([
  {$match : { product_id: product['_id']} },
  {$group : { _id:'$product_id',
              count:{$sum:1} }}
]).next();
```

只选择一个商品

返回结果集的第一个文档

这个代码返回了我们感兴趣的文档，并把它赋值给ratingSummary变量。注意，聚合管道里的结果是个游标，指向允许处理任意数量结果、每次一个文档的指针。要查询单个文档，可以使用next()函数来返回游标中的第一个文档。

```
{ "_id" : ObjectId("4c4b1476238d3b4dd5003981"), "count" : 3 }
```

聚合游标: MongoDB 2.6 新增

在MongoDB 2.6之前的版本，聚合管道返回的结果是最大值16MB的单个文档。从MongoDB 2.6以后，我们可以使用游标处理任意大小的结果集。游标是默认的shell命令的

返回值。但是为了避免破坏现有的程序，默认的文档限制仍然是16MB。要在程序里使用游标，我们可以重写默认设置来指定希望的游标。可以参看第6.5节的“聚合游标选项”的内容，了解从聚合管道里返回的游标有什么其他的功能。

传递给$match操作符的参数：{'product_id': product['_id']}应该看起来很熟悉。它们与第5章里使用计算商品评价的数量参数一样。

```
db.reviews.count({'product_id': product['_id']})
```

这些参数在前一章的5.1.1一节里详细介绍了。这里介绍的绝大部分操作符对于$match都可用。

非常重要的是$match要在$group前面。我们不能搞反了顺序，把$match 放到$group后面，返回的结果是一样的。但是这样做会让MongoDB计算所有商品的评价数量，然后抛弃所有的，只保留一个结果。通过$match前置，就可以大大减少$group操作符要处理的文档数量。

既然我们已经获得商品的总评价数量，那么现在就可以看看如何计算商品的平均评分了。这个主题超出了第5章查询语言的内容范畴。

计算评价的平均值

要计算商品评价的平均评分，就需要使用和之前例子一样的管道，添加一个新字段：

```
product = db.products.findOne({'slug': 'wheelbarrow-9092'})

ratingSummary = db.reviews.aggregate([
    {$match : {'product_id': product['_id']}},
    {$group : { _id:'$product_id',
       average:{$avg:'$rating'},
       count: {$sum:1}}}
]).next();
```

计算商品的平均评分

之前的例子返回单个文档，然后复制给变量ratingSummary，文档的内容如下：

```
{
    "_id" : ObjectId("4c4b1476238d3b4dd5003981"),
    "average" : 4.333333333333333,
    "count" : 3
}
```

这个例子使用$avg函数来计算商品的平均评分。

注意，平均评分在$avg函数里使用'$rating'指定。这种做法这里同样也适用，如下代码的字段为$group-id值指定的字段：

```
_id:'$product_id'.
```

细分评分

接下来，我们来扩展一下商品的评分统计信息，展示分类评分信息。大家可能已经在某些购物网站看到过图6.4所示界面。我们可以看到，有5个评分是满分5分，有2个评分是4分，有1个评分是3分。

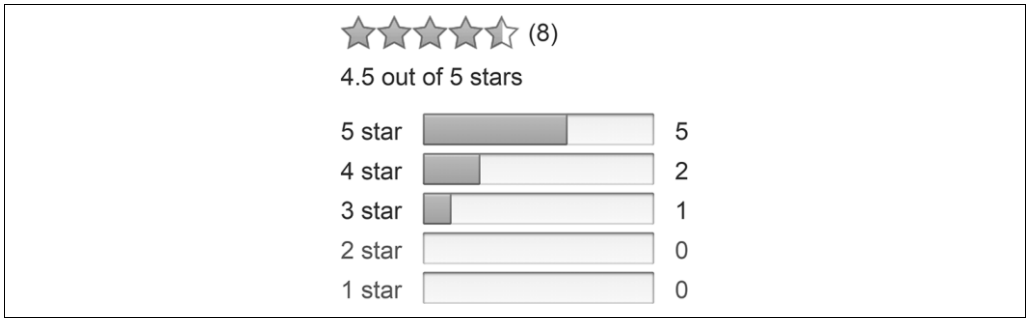


图 6.4　评价统计

使用聚合框架就像用一个命令一样来计算这些信息。这种情况下，首先使用$match来选择要计算的单个商品，和之前的例子一样。接下来，我们进行评分分类并计算每个评分的数量。

下面是实现此功能的聚合命令代码：

```
countsByRating = db.reviews.aggregate([
{$match : {'product_id': product['_id']}},
{$group : { _id:'$rating',
count:{$sum:1}}}
]).toArray();
```

选择商品

根据评分值分组

把结果游标转换为数组

为每个评分统计评价数

如代码所示，我们使用$sum函数计算出了数量；这次，我们为每个评分计算出了单独的数量。另外要注意，聚合调用的结果是个游标，我们已经把它转换为数据，并且赋值给countsByRating变量了。

SQL query 查询

对于熟悉SQL语句的开发者来说，等价的SQL查询代码如下所示：

```
SELECT RATING, COUNT(*) AS COUNT
  FROM REVIEWS
 WHERE PRODUCT_ID = '4c4b1476238d3b4dd5003981'
 GROUP BY RATING
```

聚合调用会生成如下类似的数组：

```
[ { "_id" : 5, "count" : 5 },
  { "_id" : 4, "count" : 2 },
  { "_id" : 3, "count" : 1 } ]
```

连接集合

接下来，假设我们要检查数据库的内容，计算每个类别下的商品数量。记住，商品只有一个主类别。聚合命令如下所示：

```
db.products.aggregate([
    {$group : { _id:'$main_cat_id',
                count:{$sum:1}}}
]);
```

这个命令会生成输出文档的列表。以下就是一个例子：

```
{ "_id" : ObjectId("6a5b1476238d3b4dd5000048"), "count" : 2 }
```

这种结果可能帮助不大，因为我们可能不太想知道 ObjectId ("6a5b1476238d3b4dd5000048") 表示的类别。MongoDB 的一个限制是它不允许在集合中进行关联 Joins 查询。通常我们通过去范式来解决这个问题——让它包含、通过分组或者冗余定义的电商网站期望的字段。例如，在订单集合里，每个文档包含一个商品名字，所以不需要其他调用来获取集合中的商品名字。

但要记住，聚合框架经常用来生成自定义查询的统计报告，这些统计报告也许无法提前获得。我们也许要知道需要多少数据违反范式才不至于复制太多的冗余数据。冗余数据可能会增加数据库的存储空间，使更新复杂化（因为可能需要更新多个文档中的相同信息字段）。

虽然从MongoDB 2.6开始MongoDB不允许自动化连接，但是有多种方式提供等价的SQL连接。一种选择是使用`forEach`函数来处理聚合命令返回的游标，并且使用伪连接（pseudo-join）添加名字。这是例子：

```
db.mainCategorySummary.remove({});

db.products.aggregate([
    {$group : { _id:'$main_cat_id',
                count:{$sum:1}}}
]).forEach(function(doc){
    var category = db.categories.findOne({_id:doc._id});
    if (category !== null) {
        doc.category_name = category.name;
    }
    else {
        doc.category_name = 'not found';
    }
    db.mainCategorySummary.insert(doc);
```

从 mainCategorySummary 集合删除已有文档

读取类别

无法保证类别实际存在

向摘要集合插入组合结果

```
})
```

在这段代码里，假如存在文档，我们首先从mainCategorySummary删除所有的文档。要执行伪连接，就要处理每个结果文档，并且执行findOne()调用来读取类别名称。把类别名字添加到输出结果文档里后，再把结果插入到名为mainCategorySummary的集合里。我们将会在下一章里讲解这个插入函数。

在mainCategorySummary集合上执行find()，返回的结果会包含每个类别信息。下面的findOne()命令展示了第一个结果的信息：

```
> db.mainCategorySummary.findOne();
{
       "_id" : ObjectId("6a5b1476238d3b4dd5000048"),
       "count" : 2,
       "category_name" : "Gardening Tools"
}
```

注意：伪连接很慢

正如之前提到的，MongoDB 2.6以后，聚合管道可以返回游标。但是当使用游标执行这种伪连接（pseudo-join）的时候要格外小心。虽然我们可以处理任意数量的输出文档，为每个文档运行findOne()命令，正如我们读取类别的名字，但是如果执行上百万次还是需要消耗大量的时间的。

$OUT AND 和 $PROJECT

我们很快就会看到一个更快的连接操作符$unwind，但是，我们先应该理解另外2个操作符：$out 和 $project。在前一个例子里，我们把聚合管道的结果集保存到一个名为mainCategorySummary集合里，然后使用代码处理每个文档。最后使用下面的代码保存文档：

```
db.mainCategorySummary.insert(doc);
```

使用$out操作符，可以自动把聚合管道的输出结果保存到集合里。如果集合不存在，则$out操作符将会创建一个集合，或者如果存在就会完全取代现有的集合。此外，如果创建新的集合失败的话，MongoDB不会修改之前的集合。例如，下面的代码将会把聚合管道的集合保存到一个命名的集合里：

mainCategorySummary：

```
db.products.aggregate([
```

```
    {$group : {_id:'$main_cat_id',
               count:{$sum:1}}},
    {$out : 'mainCategorySummary'}
])
```

把管道结果保存到 mainCategorySummary

$project操作符允许我们过滤可以传递给管道下一个阶段的字段。虽然$match允许我们通过限制文档数量传递下一步的数据量，但是$project可以用来限制每个传给下一步文档的大小。限制每个文档的大小可以改善性能，尤其是在处理大文档并且只需要每个文档一部分数据的时候。下面是$project操作符的例子，通过限制输出文档来调整每个产品文档的类别ID:

```
> db.products.aggregate([
... {$project : {category_ids:1}}
... ]);
{ "_id" : ObjectId("4c4b1476238d3b4dd5003981"),
  "category_ids" : [ ObjectId("6a5b1476238d3b4dd5000048"),
                     ObjectId("6a5b1476238d3b4dd5000049") ] }
{ "_id" : ObjectId("4c4b1476238d3b4dd5003982"),
  "category_ids" : [ ObjectId("6a5b1476238d3b4dd5000048"),
                     ObjectId("6a5b1476238d3b4dd5000049") ] }
```

现在我们来看看如何使用$unwind操作符执行快速连接。

使用$UNWIND 更快速连接

接下来我们将会看到聚合框架另外一个强大的功能特性：$unwind操作。这个操作符允许我们扩展数组，为每个输入文档的数组生成一个输出文档。因此，它可以提供另外一种类型的MongoDB连接，这样我们就可以使用每个子文档来连接文档了。

早期，当商品只有一个主类别的时候，要计算每个类别的商品数量。现在假设我们要计算每个类别的商品数量，无论它是不是主类别。回忆一下图6.4展示的数据模型，每个商品都包含一个category_ids数组。$unwind操作符允许我们使用每个数组元素来连接每个商品，为每个商品和category_id生成一个文档。我们可以通过category_id来进行数据统计。聚合命令列表6.1所示。

列表6.1　$unwind使用类别id数组来连接每个商品文档

```
db.products.aggregate([
    {$project : {category_ids:1}},
    {$unwind : '$category_ids'},
    {$group : { _id:'$category_ids',
    count:{$sum:1}}},

    {$out : 'countsByCategory'}
]);
```

只传递类别 ID 给下一步。默认传递_id 特性

为每个 category_ids 中的入口项目创建一个文档

$out 把聚合结果写入到集合 countsByCategory

聚合管道里的第一个操作符$project，限制传递给下一阶段的字段属性，它对于使用$unwind操作符非常重要。使用$unwind将会为数组里的每个元素生成一个输出文档，当想要限制这些输出结果的大小时，$project操作符的作用就体现出来了。如果文档其余的部分很大，而且数组包含了大量的元素，管道里就会有大量的结果集传递给下一步。MongoDB 2.6之前的版本对此会导致出错，但是对于MongoDB 2.6以后的版本，大文档也会降低管道的处理速度。我们也可以使用磁盘来存储输出结果，但这样也会进一步让管道降速。

管道里的最后一个操作符是$out，用于把结果保存到名为countsByCategory的集合里。以下是一个保存输出结果到countsByCategory集合的例子代码：

```
> db.countsByCategory.findOne()
{ "_id" : ObjectId("6a5b1476238d3b4dd5000049"), "count" : 2 }
```

一旦加载了新的集合countsByCategory，如果需要，就可以为集合里的每一行添加类别名称。下一章里，将会展示如何更新集合。

我们已经看到了如何使用聚合框架根据商品和类别来生成各种不同的统计结果。前一节也介绍了两个聚合管道里非常重要的操作符：$group和$unwind。我们已经了解$out操作符了，它可以用来保存聚合查询的结果。现在，我们来看看关于用户和订单分析的有用的总结。我们也会介绍一些聚合功能，并展示这些例子。

6.2.2 用户和订单

编写本书第一版的时候，第一次在MongoDB 2.2引入了聚合框架，还没有正式发布。第一版有两个例子使用了map-reduce函数，通过用户分组评分，根据月份统计销售信息。例子根据用户分组统计了每个用户的评价数量，以及有帮助的投票。本书使用的聚合框架提供了更加简单的、直观的方法：

```
db.reviews.aggregate([
    {$group :
        {_id : '$user_id',
        count : {$sum : 1},
        avg_helpful : {$avg : '$helpful_votes'}}
    }
])
```

调用的返回结果如下所示：

```
{ "_id" : ObjectId("4c4b1476238d3b4dd5000003"),
    "count" : 1, "avg_helpful" : 10 }
{ "_id" : ObjectId("4c4b1476238d3b4dd5000002"),
    "count" : 2, "avg_helpful" : 4 }
```

```
{ "_id" : ObjectId("4c4b1476238d3b4dd5000001"),
   "count" : 2, "avg_helpful" : 5 }
```

根据年、月统计销售信息

下面的例子展示了2010年根据月和年来统计的订单数据。我们可以在6.6.2小节看到如何使用MongoDB `map-reduce`实现类似的功能，它需要18行代码来生成相同的统计数据。下面是使用聚合框架生成的结果：

```
db.orders.aggregate([
    {$match: {purchase_data: {$gte: new Date(2010, 0, 1)}}},
    {$group: {
        _id: {year : {$year :'$purchase_data'},
              month: {$month :'$purchase_data'}},
        count: {$sum:1},
        total: {$sum:'$sub_total'}}},
        {$sort: {_id:-1}}
]);
```

运行这个命令，可以看到结果如下所示：

```
{ "_id" : { "year" : 2014, "month" : 11 },
   "count" : 1, "total" : 4897 }
{ "_id" : { "year" : 2014, "month" : 10 },
   "count" : 2, "total" : 11093 }
{ "_id" : { "year" : 2014, "month" : 9 },
   "count" : 1, "total" : 4897 }
```

在这个例子里，我们使用`$match`操作符只选择2010年1月1日之后的订单。注意，在JavaScript里，1月是从0开始的，所以我们对于日期的限制是`Date(2010,0,1)`以后。匹配函数`$gte`看起来应该很熟悉，因为在5.1.2节中介绍过。

对于`$group`操作符，我们使用年和月组合键来分组订单。虽然在集合里不经常使用组合键，但是在聚合框架里它们非常有用。在这个例子里，组合键由2个字段组成：`year`和`month`。我们也可以使用`$year`和`$month`函数从交易日期里抽取年、月字段。我们要计算订单的数量`$sum:1`，还有计算订单的总和`$sum:$sub_total`.

管道里最后的操作是排序从最近到最老的月份数据结果。对于传递给`$sort`的值也应该不陌生：与MongoDB查询`sort()`函数相同。注意组合键字段的顺序，`_id`会有问题。如果把月份放到年份前面，排序就会先月份然后年份。这看起来很奇怪，除非我们想要确定每月的趋势。

既然我们已经熟悉了聚合框架的基本操作，现在就来看一个更复杂的查询。

找到曼哈顿最好的客户

在5.1.2小节里，我们找到了曼哈顿所有的客户。现在我们来扩展查询，找出曼哈顿消费最高的用户。管道如图6.5所示。注意，$match是管道里的第一步，用来减少管道处理的文档数量。

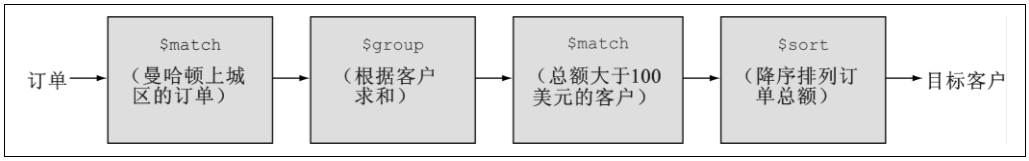


图 6.5　选择目标用户

查询包含以下步骤：

- $match——查找快递到曼哈顿的订单。
- $group——为每个客户求和。
- $match——选择总额超过100美元的客户。
- $sort——按降序排列结果。

我们将使用更简单的开发和测试管道方法。首先，定义每一步使用的参数：

```
upperManhattanOrders = {'shipping_address.zip': {$gte: 10019, $lt: 10040}};

sumByUserId = {_id: '$user_id',
               total: {$sum:'$sub_total'}, };

orderTotalLarge = {total: {$gt:10000}};

sortTotalDesc = {total: -1};
```

这些命令定义了要传递给聚合管道每一步的参数。这样做便于理解管道，因为嵌套的JSON对象难以理解。基于这些定义，整个管道的调用会如下所示：

```
db.orders.aggregate([
    {$match: upperManhattanOrders},
    {$group: sumByUserId},
    {$match: orderTotalLarge},
    {$sort: sortTotalDesc}
]);
```

我们可以单独或者组合测试管道中的每一步。例如，我们来运行管道，统计所有的客户：

```
db.orders.aggregate([
    {$group: sumByUserId},
    {$match: orderTotalLarge},
    {$limit: 10}
]);
```

下面的代码将会展示10个用户的列表:

```
{ "_id" : ObjectId("4c4b1476238d3b4dd5000002"), "total" : 19588 }
```

假设我们决定要保留订单的数量，那么要修改sumByuserId的值：

```
sumByUserId = {_id: '$user_id',
               total: {$sum:'$sub_total'},
               count: {$sum: 1}};
```

返回前面的聚合命令，我们会看到下面的结果：

```
{ "_id" : ObjectId("4c4b1476238d3b4dd5000002"),
   "total" : 19588, "count" : 4 }
```

这样构建聚合管道，可以允许我们简单地开发、迭代以及测试管道，而且也易于理解。一旦对结果满意，就可以添加$out操作符以便把结果保存到新的集合中了，因此，可以通过不同的应用来非常简单地访问这个结果：

```
db.orders.aggregate([
    {$match: upperManhattanOrders},
    {$group: sumByUserId},
    {$match: orderTotalLarge},
    {$sort: sortTotalDesc},
    {$out: 'targetedCustomers'}
]);
```

我们已经学习了聚合框架，它可以帮助我们突破许多数据库设计的限制，允许我们扩展前一章里学习的知识来分析和聚合数据。我们还学习了聚合管道以及管道里核心的操作符，包括$group和$unwind。接下来，我们将会详细看一下每个聚合操作符，并且介绍如何使用它们。正如我们之前提到的，如果读过前一章内容，在这里就会发现很多内容都很熟悉。

6.3 聚合管道操作符

Aggregation pipeline operators

聚合框架支持10个操作符：

- $project——指定要处理的字段。
- $group——根据指定的key进行分组。
- $match——选择要处理的文档，与find(…)类似。
- $limit——限制传递给下一步的文档数量。
- $skip——不传递给下一步的文档数量。

- $unwind——扩展数组，为每个数组元素生成一个输出文档。
- $sort——对文档排序。
- $geoNear——选择地理位置附近的文档。
- $out——把管道的结果输入一个集合里（2.6版本新增的）。
- $redact——控制特定数据的访问（2.6版本新增的）。

下面的小节详细介绍了这些操作符。对于绝大部分应用，$geoNear和$redact操作符用得很少，本章就不做介绍了。可以在http://docs.mongodb.org/manual/reference/operator/aggregation/查看更多详细内容。

6.3.1 $project

$project 操作符包含第5章介绍的所有查询映射功能，而且更多。下面的查询就是基于5.1.2节读取用户的姓和名的例子：

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

返回姓名的投影对象

我们可以使用如下代码实现与前面例子相同的查询条件和映射对象：

```
db.users.aggregate([
    {$match: {username: 'kbanker',
              hashed_password: 'bd1cfa194c3a603e7186780824b04419'}},
    {$project: {first_name:1, last_name:1}}
])
```

返回姓名的投影管道操作符

除了与前面查询映射使用相同的特性，我们还可以使用大量文档的重塑功能。因为文档太多，而且可以用来定义$group操作符的_id字段在6.4节里已有介绍，它主要关注重塑文档。

6.3.2 $group

$group操作符主要用于聚合管道。此操作符可以处理多个文档的聚合数据，提供诸如min、max、average的统计功能。对于熟悉SQL的读者来说，$group操作符等价于SQL的GROUP BY语句。

$group聚合功能的完整列表如表6.2所示。

我们可以告诉$group操作符如何通过定义_id字段来分组文档。然后$group操作符根据指定的_id字段分组输入文档，提供每组文档的聚合信息。下面的例子如6.2.2小节所示，我们可以通过月和年来统计销售信息：

```
> db.orders.aggregate([
...   {$match: {purchase_data: {$gte: new Date(2010, 0, 1)}}},
...   {$group: {
...     _id: {year : {$year :'$purchase_data'},
...           month: {$month :'$purchase_data'}},
...     count: {$sum:1},
...     total: {$sum:'$sub_total'}}},
...   {$sort: {_id:-1}}
... ]);
{ "_id" : { "year" : 2014, "month" : 11 },
  "count" : 1, "total" : 4897 }
{ "_id" : { "year" : 2014, "month" : 8 },
  "count" : 2, "total" : 11093 }
{ "_id" : { "year" : 2014, "month" : 4 },
  "count" : 1, "total" : 4897 }
```

当为分组定义_id字段时，我们可以使用一个或者更多存在的字段，或者可以使用6.4节里介绍的文档重塑函数。这个例子演示了使用2个重塑函数：$year和$month。

只有_id字段可以使用重塑功能（reshaping）。$group输出文档里的其他字段限制使用表6.2所示的$group函数。

表 6.2 **$group** 函数

$group 函数	
$addToSet	为组里唯一的值创建一个数组
$first	组里的第一个值。只有前缀$sort 才有意义
$last	组里最后一个值，只有前缀$sort 才有意义
$max	组里某个字段的最大值
$min	组里某个字段的最小值
$avg	某个字段的平均值

续表

$group 函数	
$push	返回组内所有值的数组。不去除重复值
$sum	求组内所有值的和

这些函数大多都浅显易懂，但这两个不太明显：$push和$addToSet。下面的例子创建了客户端的列表，每个客户端包含一个商品数组，商品数组使用$push函数创建：

```
db.orders.aggregate([
    {$project: {user_id:1, line_items:1}},
    {$unwind: '$line_items'},
    {$group: {_id: {user_id:'$user_id'},
              purchasedItems: {$push: '$line_items'}}}
]).toArray();
```

$push 函数把对象添加到 purchasedItems 数组

前面的例子代码输出的结果如下：

```
[
  {
      "_id" : {
          "user_id" : ObjectId("4c4b1476238d3b4dd5000002")
      },
          "purchasedItems" : [
          {
          "_id" : ObjectId("4c4b1476238d3b4dd5003981"),
          "sku" : "9092",
          "name" : "Extra Large Wheel Barrow",
          "quantity" : 1,
          "pricing" : {
              "retail" : 5897,
              "sale" : 4897
          }
      },
      {
          "_id" : ObjectId("4c4b1476238d3b4dd5003981"),
          "sku" : "9092",
          "name" : "Extra Large Wheel Barrow",
          "quantity" : 1,
          "pricing" : {
              "retail" : 5897,
              "sale" : 4897
          }
      },
  …
```

$addToSet与$push

看到分组函数，你可能好奇$addToSet和$push的区别。集合中的元素必须确保唯一。某个

给定的值不能在集合里出现两次，而且可以通过$addToSet强制实行。而$push操作符没有这个限制，集合中的值可以不唯一。因此，相同的元素可以在$push创建的数组里多次出现。

我们再来继续学习一些看起来很熟悉的操作符。

6.3.3 $match、$sort、$skip、$limit

这4个操作符放在一起介绍，因为它们与第5章里介绍的查询函数相同。使用这些操作符，我们可以选择特定的文档、排序文档、跳过特定数量的文档、限制处理文档的数量。

把它们与第5章里介绍的查询语言对比，会发现它们的参数也一样。以下是5.1.1节里分页查询的例子代码：

```
page_number = 1
product = db.products.findOne({'slug': 'wheelbarrow-9092'})

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

聚合框架等价的查询代码如下所示：

```
reviews2 = db.reviews.aggregate([
    {$match: {'product_id': product['_id']}},
    {$skip : (page_number - 1) * 12},
    {$limit: 12},
    {$sort: {'helpful_votes': -1}}
]).toArray();
```

正如你看到的，两个版本的代码功能和输入参数是一样的。

一个例外之处是find() $where函数，它允许我们使用Javascript表达式来选择文档。$where不能使用聚合框架的$match操作符。

6.3.4 $unwind

我们在6.2.1小节里讨论快速连接的时候已经看到过$unwind。这个操作符通过为数组里的每个元素生成一个输出文档来扩展数组。主文档里的字段、每个数组元素的字段都被放入输出文档里。这个例子显示了$unwind之前和之后商品的类别。

```
> db.products.findOne({},{category_ids:1})
{
```

```
    "_id" : ObjectId("4c4b1476238d3b4dd5003981"),
    "category_ids" : [
        ObjectId("6a5b1476238d3b4dd5000048"),
        ObjectId("6a5b1476238d3b4dd5000049")
    ]
}

> db.products.aggregate([
...     {$project : {category_ids:1}},
...     {$unwind : '$category_ids'},
...     {$limit : 2}
... ]);
{ "_id" : ObjectId("4c4b1476238d3b4dd5003981"),
  "category_ids" : ObjectId("6a5b1476238d3b4dd5000048") }
{ "_id" : ObjectId("4c4b1476238d3b4dd5003981"),
  "category_ids" : ObjectId("6a5b1476238d3b4dd5000049") }
```

现在我们来看一个MongoDB v2.6中的新操作符`$out`。

6.3.5 $out

在6.2.2中，我们创建了一个管道来查询曼哈顿最好的客户。我们这里会再次使用这个例子，但是这次的管道输出文档使用`$out`操作符保存在`targetedCustomers`集合里。`$out`操作符必须是管道里的最后一个操作符：

```
db.orders.aggregate([
    {$match: upperManhattanOrders},
    {$group: sumByUserId},
    {$match: orderTotalLarge},
    {$sort: sortTotalDesc},
    {$out: 'targetedCustomers'}
]);
```

加载的结果数据必须符合集合的约束。例如，所有集合文档必须有唯一的`_id`。如果因为某些原因，管道失败了，无论是在`$out`擦操作之前还是过程中，现有的集合都不能改变。在使用这个方法生成一个等价的`SQL`物化视图时要记住这一点[1]。

MongoDB 的物化视图

绝大部分关系型数据库都提供了物化视图功能。物化视图提供了一种高效和易用的方式来访问提前生成的数据结果。通过提前生成这些信息，我们可以节约大量的生成需要数据的时间，而且也便于应用程序预处理信息。`$out`操作的失败安全性是生成等价物理视图的关键。无论什么原因，如果生成新集合失败，都会保留之前的集合。如果我们期望其他应该

[1]【译者注】SQL Materialized views，物化视图，就是包含查询数据的视图。

也可以独立使用这个信息，则这是个非常重要的特性，就算集合过时也比丢失数据好多了。

我们现在已经看了主要的管道操作符。现在我们回到之前提到的主题上：重塑文档。

6.4 重塑文档

Reshaping documents

MongoDB的聚合管道包含许多可以用来重塑文档的函数，因此可以生成一个包含最初文档没有的字段的新文档。我们通常会与`$project`操作符一起使用这些函数，但是也可以在为`$group`操作符定义`_id`时使用。

最简单的重塑功能就是把一个字段进行重命名，生成一个新字段。我们也可以通过修改或者创建一个新文档来重塑一个文档。例如，回到之前读取用户姓和名字段的例子，如果要创建一个同时包含两个字段first和last名为name的字段，可以使用如下代码实现：

```
db.users.aggregate([
    {$match: {username: 'kbanker'}},
    {$project: {name: {first:'$first_name',
                       last:'$last_name'}}
    }
])
```

运行代码的输出结果如下所示：

```
{ "_id" : ObjectId("4c4b1476238d3b4dd5000001"),
    "name" : { "first" : "Kyle",
               "last" : "Banker" }
  }
```

除了重命名或者重新构建文档字段外，我们也可以使用不同的重塑函数来创建新的字段。重塑函数根据处理数据类型的不同进行分组：字符串、算数运算、日期、逻辑、集合、其他类型。

接下来，我们将会详细看一下每组不同的函数，从执行字符串操作开始。

聚合框架重塑函数

有许多函数——似乎每次发布都会增加函数——允许我们在输入文档中执行许多不同的操作以生成新字段。

本节里我们会看一下各种不同的操作符，同时进行复杂重塑功能的实现。最新的可用函数列表可以参考`MongoDB`官方文档http://docs.mongodb.org/manual/reference/operator/

aggregation/group/。

6.4.1 字符串函数

字符串函数如表6.3所示，允许我们操作字符串。

表 6.3 字符串函数

Strings	
$concat	连接 2 个或者更多字符串为一个字符串
$strcasecmp	大小写敏感的比较，返回数字
$substr	获取字符串的子串
$toLower	转换为小写字符串
$toUpper	转换为大写字符串

这个例子使用了三个函数——$concat、$substr、$toUpper:

```
db.users.aggregate([
    {$match: {username: 'kbanker'}},
    {$project:
        {name: {$concat:['$first_name', ' ', '$last_name']},
        firstInitial: {$substr: ['$first_name',0,1]},
        usernameUpperCase: {$toUpper: '$username'}
        }
    }
])
```

姓名使用空格连接

名字的第一个字符初始化

修改 username 为大写字母

运行代码的结果如下所示：

```
{ "_id" : ObjectId("4c4b1476238d3b4dd5000001"),
  "name" : "Kyle Banker",
  "firstInitial" : "K",
  "usernameUpperCase" : "KBANKER"
}
```

接下来我们来看下算术运算函数。

6.4.2 算术运算函数

算术运算函数包含标准的算术运算操作，如表6.4所示。

表 6.4　算术运算函数

算术运算	
$add	求和
$divide	除法
$mod	求余数
$multiply	乘积
$subtract	减法

通常，算术运算函数允许我们对数字执行基本的计算，例如，加、减、乘、除。

接下来我们来看一些与日期相关的函数。

6.4.3　日期函数

日期函数如表6.5所示，使用现有的日期或者通过计算年、月、日等来创建新字段。

表 6.5　日期函数

日期	
$dayOfYear	一年 365 天中的某一天
$dayOfMonth	一月中的某一天
$dayOfWeek	一周中的某一天，1 表示周日
$year	日期的年份
$month	日期的月份，1～12
$week	一年中的某一周，0～53
$hour	日期中的小时，0～23
$minute	日期中的分钟，0～59
$second	日期中的秒，0～59
$millisecond	日期中的毫秒，0～999

我们已经在6.2.2小节里看过$year和$month的例子，使用它们来根据年份和月份统计销售数据。剩下的日期函数比较简单了，所以我们接下来就详细看一下逻辑函数。

6.4.4　逻辑函数

逻辑函数如表6.6所示，看起来很熟悉。绝大部分与第5章5.2节里总结的查询操作符类似。

表 6.6　逻辑函数

逻辑	
$and true	与操作，如果数组里所有值都为 true，则返回 true
$cmp	如果两个数相等就返回 0
$cond if… then… else	条件逻辑
$eq	两个值是否相等
$gt	值 1 是否大于值 2
$gte	值 1 是否大于等于值 2
$ifNull	把 null 值/表达式转换为特定的值
$lt	值 1 是否小于值 2
$lte	值 1 是否小于等值 2
$ne	值 1 是否不等于值 2
$not.	取反操作
$or	或，如果数组中有一个 true，就返回 true

$cond函数与我们看到的其他函数不同，它允许复杂的操作：如果、然后、其他。这与很多语言里的三元操作符很像。例如x ? y : z，表示如果条件x为true就选择y，否则就选择z。

接下来是集合操作符，它允许我们使用不同的方式来比较集合的值。

6.4.5 集合操作符

集合操作符总结在表6.7中，允许我们比较两个数组的内容。使用集合操作符，可以比较两个数组、判断它们是否完全一样、是否有公共元素、是否有差别元素。如果需要使用这些函数，最简单的方式就是查看MongoDB的官方文档：http://docs.mongodb.org/manual/reference/operator/aggregation-set/。

表 6.7 集合函数

集合	
$setEquals	如果两个集合的元素完全相同，则为 true
$setIntersection	返回两个集合的公共元素
$setDifference	返回第一个集合中与第二个集合不同的元素
$setUnion	合并集合
$setIsSubset	如果第二个集合为第一个集合的子集，则为 true
$anyElementTrue	如果某个集合元素为 true，则为 true
$allElementsTrue true	如果所有集合元素都为 true，则为 true

这是个使用$setUnion函数的例子。假设我们已经有如下商品：

```
{ "_id" : ObjectId("4c4b1476238d3b4dd5003981"),
  "productName" : "Extra Large Wheel Barrow",
  "tags" : [ "tools", "gardening", "soil" ]}
{ "_id" : ObjectId("4c4b1476238d3b4dd5003982"),
  "productName" : "Rubberized Work Glove, Black",
"tags" : [ "gardening" ]}
```

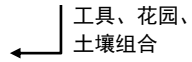
如果合并这些商品的tags，就可以取得名为testSet1的数组，如下所示：

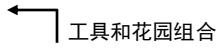
```
  testSet1 = ['tools']

  db.products.aggregate([
    {$project:
       {productName: '$name',
       tags:1,
       setUnion: {$setUnion:['$tags',testSet1]},
       }
     }
  ])
```

结果将包含如下所示的标签：

```
{   "_id" : ObjectId("4c4b1476238d3b4dd5003981"),
    "productName" : "Extra Large Wheel Barrow",
    "tags" : ["tools", "gardening", "soil"],
    "setUnion" : ["gardening","tools","soil"]
}
{   "_id" : ObjectId("4c4b1476238d3b4dd5003982"),
    "productName" : "Rubberized Work Glove, Black",
    "tags" : ["gardening"],
    "setUnion" : ["tools", "gardening"]
}
```





我们已经操作了各种不同的文档重塑函数，但是还有一个类别没有讲到：不出名的小众类别，我们把不属于前面类别的函数放到这里。

6.4.6 其他函数

最后一个分组混合了其他的函数，如表6.8所示。这些函数执行不同的功能，所以我们一起来介绍。$meta函数与文本搜索有关系，本章就不做介绍了。我们可以在第9章里学习更多关于文本搜索的内容。

表 6.8 其他函数

其他函数	
$meta	文本搜索。参考第 9 章
$size	返回数组大小
$map	对数组的每个成员应用表达式
$let	定义表达式内使用的变量
$literal	返回表达式的值，而不评估它

$size 函数返回数组的大小。这个函数非常有用，例如，可用于判断数组是否包含某个元素或者是否为空。$literal函数允许我们避免初始化字段值为0、1、$的问题。

$let函数允许我们使用临时变量，而不需要使用$project步骤。这个函数当我们要执行一系列复杂函数或者计算的时候非常有用。

$map函数允许我们来处理数组，并通过数组的元素执行函数来生成一个新的数组。$map在我们想重塑数组，但是又不想使用$unwind时非常有用。

本节学习完重塑文档的知识后，接下来将要研究一些与性能相关的问题。

6.5 理解聚合管道性能

Understanding aggregation pipeline performance

本节里我们将会学习如何改善聚合管道的性能，理解为什么管道性能会下降，也会学习如何突破中间或者最后输出结果的大小限制、从MongoDB 2.6开始删除的约束。

以下是主要影响聚合管道性能的关键点：

- 尽早在管道里尝试减少文档的数量和大小。
- 索引只能用于`$match`和`$sort`操作，而且可以大大加速查询。
- 在管道使用`$match`和`$sort`之外的操作符后不能使用索引。
- 如果使用分片（分片存储大数据集合），则`$match`和`$project`会在单独的片上执行。一旦使用了其他操作符，其余的管道将会在主要片上执行。

本书全部内容都会鼓励大家尽可能多地使用索引。在第8章索引与查询优化中，我们会详细介绍这个主题。4个关键性能点中有2个都提到了索引，所以希望大家能意识到：索引可以大大加快大集合选择性查询和排序。

有时候，特别是当使用聚合框架的时候要处理大量的数据，此时索引不是恰当的方式。例如，6.2.2小节中通过年份和月份来计算销售数据时。处理大量的数据当然很好，但是用户不愿意长时间等待网页返回结果。当我们必须统计数据——例如，在网页上——通常可以提前生成数据并且使用`$out`存储到集合里。

我们通过聚合框架的`explain()`函数来研究一下如何判断查询是否使用了索引机制。

6.5.1 聚合管道选项

直到现在，我们也只介绍了当传递数组给管道操作时调用`aggregate()`函数。从MongoDB 2.6开始，我们可以为`aggregate()`传递第二个参数来指定聚合调用。选项参数如下：

- `explain()`——运行管道并且只返回管道处理详细信息。
- `allowDiskUse`——使用磁盘存储数据。
- `cursor`——指定初始批处理的大小。

使用如下格式传递这些选项参数：

```
db.collection.aggregate(pipeline,additionalOptions)
```

pipeline就是之前例子里我们看过的管道操作的数组，additionalOptions是一个可选JSON对象，可以传递给aggregate()函数。additionalOptions的参数如下：

```
{explain:true, allowDiskUse:true, cursor: {batchSize: n} }
```

从explain()函数开始，我们来详细看看每个参数选项。

6.5.2 聚合框架的 explain( )函数

MongoDB explain()函数与SQL里的EXPLAIN函数类似，描述查询路径，允许开发者通过确定使用的索引来诊断慢操作。我们在第2章第一次讨论find()查询时介绍了explain()函数。列表6.2是复制列表2.2中的代码，用来说明如何使用索引来改善find()查询函数的性能。

列表6.2 索引查询的explain()函数输出

```
> 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,
```

使用 num_1

返回 4 个文档

```
    "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,
            "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
}
```

只扫描 4 个文档

使用 num_1 索引

聚合框架里使用explain()函数与find()查询里使用explain()函数不同，但是提供了类似的功能。正如我们期望的，对于聚合管道，我们会看到管道里每个操作的输出信息，因为管道里的每一步都是对下一步的调用（参见列表6.3）。

列表6.3 聚合函数的explain( )输出。

```
> countsByRating = db.reviews.aggregate([
... {$match : {'product_id': product['_id']}},
... {$group : { _id:'$rating',
... count:{$sum:1}}}
... ],{explain:true})
{
    "stages" : [
        {
            "$cursor" : {
                "query" : {
                        "product_id" : ObjectId("4c4b1476238d3b4dd5003981")
                },
                "fields" : {
                        "rating" : 1,
                        "_id" : 0
                },
                "plan" : {
                    "cursor" : "BtreeCursor ",
                    "isMultiKey" : false,
                    "scanAndOrder" : false,
                    "indexBounds" : {
                        "product_id" : [
                            [
                                ObjectId("4c4b1476238d3b4dd5003981"),
                                ObjectId("4c4b1476238d3b4dd5003981")
                            ]
                        ]
                    },
                    "allPlans" : [
                        ⋮
                    ]
                }
            }
        },
        {
            "$group" : {
                "_id" : "$rating",
                "count" : {
                        "$sum" : {
                                "$const" : 1
                        }
                }
            }
        }
    ],
    "ok" : 1
}
```

优先匹配$match

Explain 参数为 true

使用 BTreeCursor，基于索引的游标

设置单个文档的范围

虽然这里显示的信息没有列表6.2中显示的find().explain()结果详细，但是它仍然提供一些关键的信息。例如，它显示了某个索引是否被显示，以及索引扫描的范围。这可以告诉我们哪个索引能够限制查询。

聚合框架的 explain()工作过程

explain()是MongoDB 2.6新增的函数。因为之前版本缺少类似find().explain()的功能，所以新增的explain()函数可以提供类似的功能。正如MongoDB在线文档http://docs.mongodb.org/manual/reference/method/db.collection.aggregate/#example-aggregate- method-explain-option介绍的。此函数的输出结果是给人而不是给机器阅读的，输出结果的格式在不同的版本中可能有变化。如果结果与find().explain()一样也不要感到惊讶。find().explain()函数在MongoDB 3.0已经进行了更大改进，包含了比MongoDB 2.6版本里find().explain()函数更多的输出信息，而且可以支持三种模式：queryPlanner、executionStats和allPlansExecution。

现在我们来看看另外一个解决之前限制处理数据大小的参数。

正如你已经知道的，explain()函数输出结果可能与使用的MongoDB服务器有关。

6.5.3 allowDiskUse 选项

如果我们处理超大型集合数据，就会看到下面类似的错误：

```
assert: command failed: {
        "errmsg" : "exception: Exceeded memory limit for $group,
        but didn't allow external sort. Pass allowDiskUse:true to opt in.",
        "code" : 16945,
        "ok" : 0
} : aggregate failed
```

更令人沮丧的是，这个错误可能在很久之后发生，可能已经处理了几百万个文档，但是还是失败了。

发生这种情况的原因是，管道返回了超过MongoDB RAM内存限制的100MB数据。修复错误很简单，正如错误信息里提示的：设置allowDiskUse:true参数就可以了。

我们来看根据月份统计销售数据的例子。因为处理的销售数据量非常大，所以管道需要这个参数：

```
db.orders.aggregate([
```

```
    {$match: {purchase_data: {$gte: new Date(2010, 0, 1)}}},
    {$group: {
        _id: {year : {$year :'$purchase_data'},
              month: {$month :'$purchase_data'}},
        count: {$sum:1},
        total: {$sum:'$sub_total'}}},
    {$sort: {_id:-1}}
], {allowDiskUse:true});
```

优先匹配$match过滤文档

允许MongoDB使用磁盘存储

通常来说，使用allowDiskUse参数可能降低管道的性能，所以只推荐在需要的时候使用。正如前面提到的，我们应该尝试限制管道处理文档的数量和大小，使用$match选择处理的文档，使用$project选择要处理的字段。如果运行大数据的管道遇到这种情况，就使用这个参数会确保安全。

现在，我们来看下聚合管道里的最后一个参数：:cursor。

6.5.4 聚合游标选项

在MongoDB 2.6之前，管道结果的限制是单个文档16 MB。从2.6版本开始，通过Mongo shell访问MongoDB默认返回的是游标。但是，如果从程序里运行管道，为了避免程序崩溃而默认限制不变，则仍然限制返回文档大小是16MB。在程序里，我们可以通过如下的代码来访问数据库，返回游标结果：

```
countsByRating = db.reviews.aggregate([
  {$match : {'product_id': product['_id']}},
  {$group : { _id:'$rating',
              count:{$sum:1}}}
],{cursor:{}})
```

返回一个游标

聚合管道返回的游标支持如下调用：

- cursor.hasNext()——确定结果集是否包含下一个元素。
- cursor.next()——返回结果集的下一个文档。
- cursor.toArray()——以数组返回结果。
- cursor.forEach()——遍历结果集的每一行。
- cursor.map()——遍历结果集的每一行，返回一个结果数组。
- cursor.itcount()——返回结果数量（仅作测试）。
- cursor.pretty()——显示格式化结果的数组。

记住，游标(cursor)允许我们处理大规模数据流(stream)。它允许我们在返回少量文档结果的

时候处理大的结果集，因此可以减少一次性处理数据所需的内存。此外，如果只需要一些文件，游标也可以限制服务端返回的文档数量。使用`toArray()`和`pretty()`就没有这些优势，结果会立即读入内存中。

类似地，`itcount()`会读取所有文档，并且全部发送给客户端，但是通常会抛弃结果，只返回数量。如果程序只需要数量，就可以使用`$group`管道操作符来计算输出文档的数量，而不需要发送每个文档给程序——这是更加高效的处理方式。

6.6 其他聚合功能

Other aggregation capabilities

虽然聚合管道是处理MongoDB聚合查询数据的首选方式，但也还有一些替代方式。有些非常简单，比如`.count()`函数。还有一个更复杂的方式，就是早版本MongoDB的`map-reduce`函数。

让我们先从简单的替代方式开始。

6.6.1 .count( ) 和 .distinct( )

我们在6.2.1一节里已经看到了`.count()`函数。下面是摘录的部分代码：

```
product = db.products.findOne({'slug': 'wheelbarrow-9092'})
reviews_count = db.reviews.count({'product_id': product['_id']})
```

统计商品的评价

现在我们来看看使用`.distinct()`函数的例子。下面代码将返回一串快递订单的邮政编码信息数组，且不重复：

```
db.orders.distinct('shipping_address.zip')
```

`.distinct()`返回结果的大小限制为16 MB，也是MongoDB文档的最大限制。

接下来我们来看一下最早版本MongoDB提供的聚合功能：`map-reduce`。

6.6.2 map-reduce

`map-reduces` 是MongoDB提供灵活聚合功能的首次尝试。使用`map-reduce`，就可以使用

JavaScript定义整个处理流程。这提供了很大的灵活性，但是比聚合框架性能要低得多[1]。此外，编写map-reduce的过程十分复杂，而且比之前构建的聚合框架更加难以理解。我们来看看之前的map-reduce框架应用例子。

注意：map-reduce的详细解释可以参考谷歌最初的论文，关于map-reduce的编程模型地址为
http://static.googleusercontent.com/media/research.google.com/en/us/archive/mapreduce- osdi04.pdf.

在6.2.2小节，我们展示了聚合管道提供的销售统计信息：

```
db.orders.aggregate([
    {"$match": {"purchase_data":{"$gte" : new Date(2010, 0, 1)}}},
    {"$group": {
        "_id": {"year" : {"$year" :"$purchase_data"},
                "month" : {"$month" : "$purchase_data"}},
        "count": {"$sum":1},
        "total": {"$sum":"$sub_total"}}},
    {"$sort": {"_id":-1}}]);
```

我们使用map-reduce来生成相似的结果。第一步，正如名字的含义，就是编写map函数。这个函数会用到集合和处理过程中的每个文档上，有两个目的：第一，定义分组操作的键；第二，打包索引需要计算的数据。要了解这个过程，就来详细看看下面的函数：

```
map = function() {
    var shipping_month = (this.purchase_data.getMonth()+1) +
        '-' + this.purchase_data.getFullYear();

    var tmpItems = 0;
    this.line_items.forEach(function(item) {
        tmpItems += item.quantity;
    });

    emit(shipping_month, {order_total: this.sub_total,
                          items_total: tmpItems});
};
```

首先要知道，在这种情况下变量this引用的是当前迭代的文档——订单。在函数的第一行，我们获取一个值来指定订单创建的月份。然后调用emit()，这是每个map函数必须调用的一个专门的方法。

emit()的第一个参数是分组的关键值，第二个参数是包含值的要处理的文档。此时，我们通

[1]虽然 MongoDB 里改善了 JavaScript 的性能，但是还有一些关键原因，map-reduce 比聚合框架慢的多。这些问题的介绍请参考 William Zola 在 StackOverflow 网站的回答 http://stackoverflow.com/questions/12678631/mapreduce-performance-in-mongodb-2-2-2-4-and-2-6/12680165#12680165。

过月份分组，然后统计每个订单的总额和项目数量。

对应的reduce函数如下：

```
reduce = function(key, values) {
    var result = { order_total: 0, items_total: 0 };
    values.forEach(function(value){
        result.order_total += value.order_total;
        result.items_total += value.items_total;
    });
    return ( result );
};
```

它会为reduce函数传递一个键和一个数组。我们编写reduce函数的工作就是确保这些值按照期望的方式来进行聚合处理，然后返回单个字。因为map-reduce的迭代本性，reduce可能会被多次调用。我们的代码必须处理这种问题。此外，如果map函数只发出一个值，reduce函数就不会被调用。因此，reduce返回的数据结构必须与map函数返回的数据结构一样。

我们来详细看一下map和reduce函数的例子。

添加查询过滤并保存输出结果

shell的map-reduce方法需要一个map和一个reduce函数作为参数。这个例子添加了更多参数。第一个是查询过滤器，用于限制聚合处理过程中的文档数量，从2010开始。第二个参数是输出结果集合的名字。

```
filter = {purchase_data: {$gte: new Date(2010, 0, 1)}};
db.orders.mapReduce(map, reduce, {query: filter, out: 'totals'});
```

过程如图6.6所示，包含以下步骤：

(1) filter 选择特定的订单数据。

(2) map 生成一个键值对，通常一个输入一个输出，但是它也可以不生成或者生成多个结果。

(3) reduce 传递一个key和map生成的值的数组给reduce函数，通常是每个key一个数组，但是可能会使用不同值的数组传递多次相同的key。

图6.6所示中的一个重点就是，如果map函数为一个键生成一个结果，那么reduce步骤就会跳过去不执行。这也是理解为什么不能修改map函数的输出结果的数据结构的关键点，这对于reduce步骤非常重要。

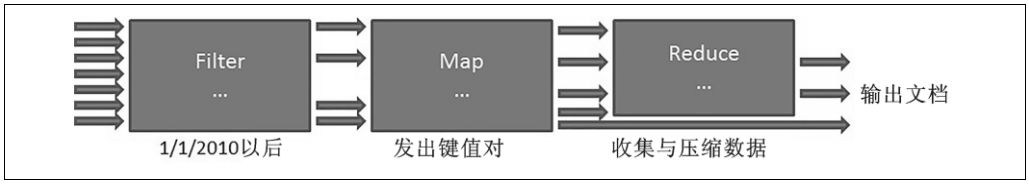


图 6.6　map-reduce 处理过程

在这个例子里，结果存储在名为totals的集合中，我们可以查询这个集合。下面的列表显示了对于此集合的查询结果。_id字段保存的是分组的key，年份和月份还有value字段引用的总计信息。

列表6.4　查询map-reduce输出集合。

```
> db.totals.find()
{ "_id" : "11-2014", "value" : {"order_total" : 4897, "items_total" : 1 } }
{ "_id" : "4-2014", "value" : {"order_total" : 4897, "items_total" : 1 } }
{ "_id" : "8-2014", "value" : {"order_total" : 11093, "items_total" : 4 } }
```

这里的例子应该可以让你对MongoDB的聚合功能有实际的理解。把这个例子和聚合框架的处理过程比较，就会发现map-reduce不再是这种功能推荐的处理方法了。

但是在有些情况下，我们需要map-reduce 提供的JavaScript的灵活性。本书里就不做介绍了，大家可以在MongoDB 网站http://docs.mongodb.org/manual/core/map-reduce/找到map-reduce的例子。

map-reduce——很好的首次尝试

第一届全球MongoDB技术大会于2014年在美国纽约举行，MongoDB数据库的工程师团队展示了不同配置情况下处理几太字节数据的对比测试结果。一个工程师介绍了使用聚合框架而不是map-reduce的测试情况。当问到这个问题时，那个工程师说，map-reduce不再是推荐的处理数据的方式，但是是个“很好的首次尝试”。

虽然map-reduce提供了JavaScript的灵活性，但是它限制了必须是单线程和解释性的模式。聚合框架（aggregation framework），换句话说，是作为原生C++和多线程模式执行的。虽然map-reduce没有被淘汰，但是未来的改进都会在聚合框架上进行。

6.7 总结

Summary

本章包含许多的知识点。$group操作符提供了聚合框架的关键功能：从多个文档生成单个文档数据的功能。还有$unwind 和 $project，聚合框架提供了生成最新统计数据的功能，或者离线处理大量数据并使用$out命令保存到新的集合中的功能。

查询和聚合组成了MongoDB接口的关键部分。所以，一旦我们阅读完本章，就可以测试查询和聚合功能了。如果你还不确定某个查询操作符的组合如何工作，那么MongoDB shell是很好的测试工具。所以，尝试实战一下聚合框架的某些关键功能，比如使用$match操作符选择文档，或者使用$project操作符重建文档。当然还有使用$group进行分组和统计信息。

从现在开始，我们将使用MongoDB的统一查询功能，而且接下来的一章会是很好的实战集合。我们会处理常见文档的CRUD操作：create、read、update、delete。因为在绝大部分更新操作里，查询都扮演了关键的角色，我们可以期待更多查询的详细知识。我们也会学习如何更新文档，特别是高并发大容量更新的数据库，这比大家熟悉的关系型数据库需要更多的功能。