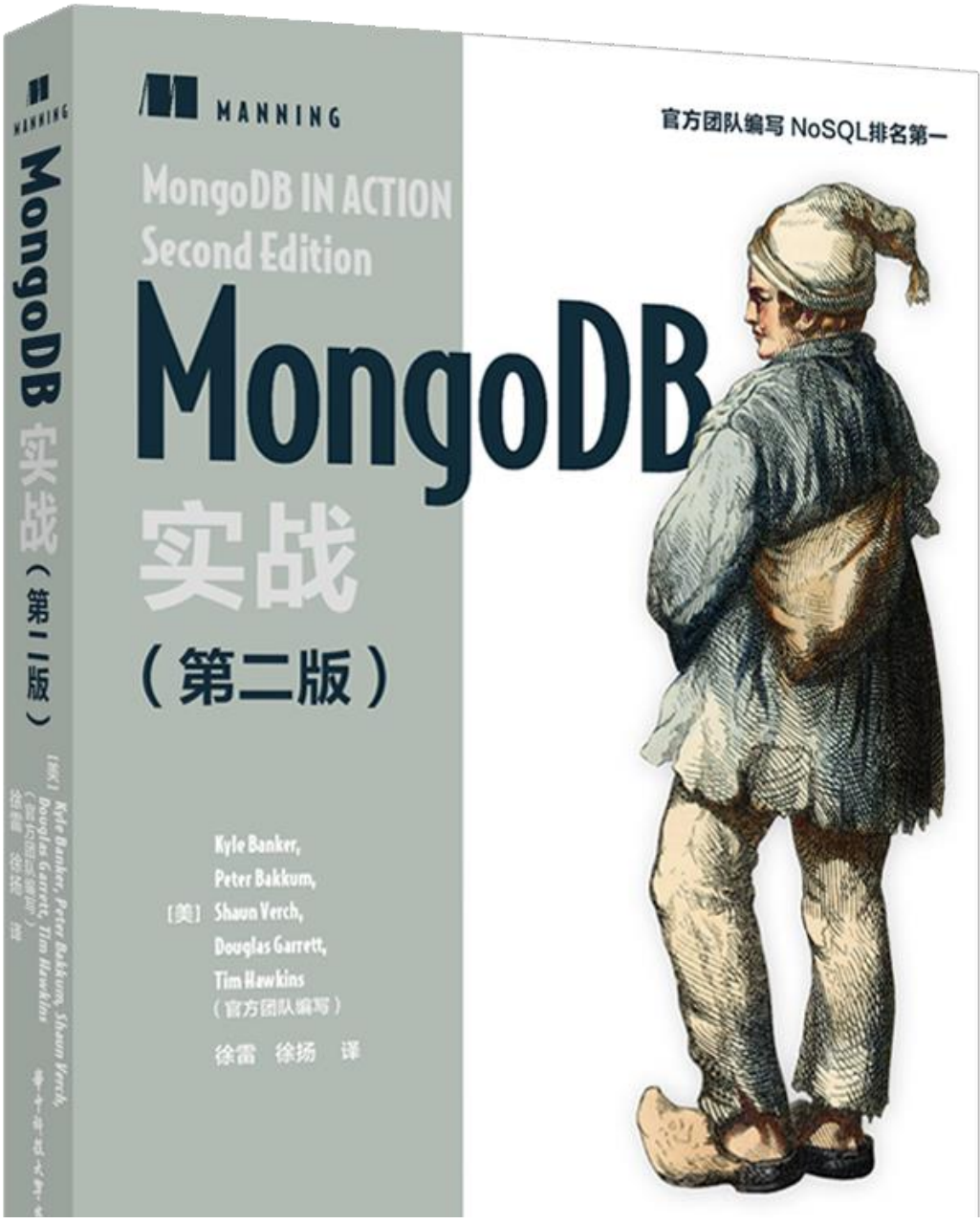
官方团队出品　　NoSQL 排名第一　《Mongodb 实战》第 2 版　互联网架构师必备



第 7 章

更新、原子操作和删除

Updates,atomic operations and deletes

本章内容

- 更新文档
- 自动处理文档
- 复杂更新的实际例子
- 使用更新操作符
- 删除文档

更新就是写入数据到现有的文档中。高效的做法需要完整理解文档结构的类型以及MongoDB支持的查询表达式。在过去的两章里我们已经学习了电商的数据模型，明白了schema数据模型设计和查询方式。在我们的更新学习中我们将会使用这些知识。

特别是，我们将会深入讲解为什么我们要使用非范式来设计类型的层次结构，以及MongoDB如何更新，让此数据结构看起来合理。

我们将会看一下库存管理，并且解决其中的一些并发问题。大家会学习一些新的更新操作符，包括一些原子更新的技巧，试验`findAndModify`命令。在这种情况下，MongoDB 的原子性指的是查找并更新一个文档，确保不会被其他线程干扰。在许多列子之后，会有一节内容专门总结更新操作符，它会扩展所有的例子来允许我们通过参数控制如何更新数据。我们还会讨论如何删除MongoDB中的数据 ,最后来深入讲解一些MongoDB高并发和优化的知识。

本章里的绝大部分内容都使用JavaScript shell编写。本节里我们会讨论原子性文档处理工作，可能到时候需要许多应用级别的逻辑代码实现，所以我们将会使用Ruby来编写。

在本章结束时，你会看到看到完整的MongoDB CRUD操作，并且可以熟练掌握最大化利用MongoDB数据库接口和数据模型优势的设计方法。

7.1 文档更新概要

A brief tour of document updates

更新MongoDB的数据库有两种实现方式。我们既可以完整替换现有的文档，也可以使用更新操作符来修改文档里的某个字段。在更详细的例子开始之前，我们先从简单的demo开始介绍这两种方式。然后介绍一下选择其中一种最优方式的理由。

大家回忆一下第4章里的user文档。这个文档包括用户的姓名、email邮箱，还有快递地址。

这是一个简单的例子：

```
{
  _id: ObjectId("4c4b1476238d3b4dd5000001"),
  username: "kbanker",
  email: "kylebanker@gmail.com",
  first_name: "Kyle",
  last_name: "Banker",
  hashed_password: "bd1cfa194c3a603e7186780824b04419",
  addresses: [
    {
      name: "work",
      street: "1 E. 23rd Street",
      city: "New York",
      state: "NY",
      zip: 10010
    }
  ]
}
```

毫无疑问，我们会经常更新电子邮件地址，所以现在就从它开始做吧。

请注意下，`ObjectId`的值可能不太一样，但确保使用有效的值。如果需要，就手动添加文档，这样可以帮助熟悉本章的命令。或者也可以使用下面的方法来查询有效的文档，获取它的`ObjectId`，在其他地方使用：

```
doc = db.users.findOne({username: "kbanker"})
user_id = doc._id
```

7.1.1 通过替换修改

要替换文档，首先要查询文档，在客户端修改，然后修改文档。下面是在JavaScript shell里使用命令的代码：

```
user_id = ObjectId("4c4b1476238d3b4dd5003981")
```

```
doc = db.users.findOne({_id: user_id})
doc['email'] = 'mongodb-user@mongodb.com'
print('updating ' + user_id)
db.users.update({_id: user_id}, doc)
```

有了用户_id，首先可以查询到文档数据，接下来可以本地修改文档。这种情况下，修改email字段，然后传递修改过的文档给update方法。最后一行意思是“使用给定的_id在users集合里查找文档”。要记住的是，更新操作会取代整个文档，这也就是要先获取文档数据的原因。如果多个用户更新同一个文档，则只会保存最后更新的数据。

7.1.2 通过操作符修改

上面展示了如何通过替换文档修改数据，现在我们来看看如何通过操作符来修改数据：

```
user_id = ObjectId("4c4b1476238d3b4dd5000001")
db.users.update({_id: user_id},
    {$set: {email: 'mongodb-user2@mongodb.com'}})
```

这个例子使用的$set是几个特殊的更新操作符之一，用于修改单个文档中的email地址。这种情况下，更新请求更具目标性：找到文档，然后修改email地址为mongodbuser2@mongodb.com。

Syntax note 语法提示：更新与查询

MongoDB的新手区分更新和查询语法可能有点困难。目标更新通常使用更新操作符，而且操作符通常使用动词构造（set、push等）。以$addToSet为例：

```
db.products.update({}, {$addToSet: {tags: 'Green'}})
```

如果为update添加查询选择器，注意查询选择器语义上是类似的（小于、等于等），跟随在查询字段名字的后面(这个例子里的price)：

```
db.products.update({price: {$lte: 10}},
{$addToSet: {tags: 'cheap'}})
```

最后一个查询例子只更新价格小于等于10的数据，然后添加“cheap（便宜的）”标签。

更新操作符使用前缀表示符，而查询操作符使用中缀表示符，这意味着$addToSet在更新操作符里是第一个，而$lte在查询操作符之内。

7.1.3 比较两个方法

另外一个例子怎么样？这次我们要增加商品的评价。以下是使用替换方法实现此功能需求的：

```
product_id = ObjectId("4c4b1476238d3b4dd5003982")
doc = db.products.findOne({_id: product_id})
doc['total_reviews'] += 1 // add 1 to the value in total_reviews
db.products.update({_id: product_id}, doc)
```

这里是目标方法：

```
db.products.update({_id: product_id}, {$inc: {total_reviews: 1}})
```

替换方法也和前面一样，从服务器获取用户文档→修改→然后重新发回给服务器。这些代码与之前更新用户email地址的代码类似。与之相反，目标更新方法使用了不同的操作符`$inc`来增加`total_reviews`字段的值。

7.1.4 决定：替换与操作符

既然我们已经看了许多更新的实战例子，能否给出一个选择更优方法的理由呢？哪个方法更直观？为什么某一个比另一个的性能更好?当多线程同时更新的时候会有什么问题？彼此之间是否隔离？

替换是更通用的做法。想象一下通过应用HTML表单来更新用户user数据。使用文档替换时，数据从表单提交，一旦验证，就可以传递给MongoDB；不管哪个字段被更新，代码执行的更新是相同的。例如，要构建一个MongoDB对象的映射器来支持更新，就可通过替换实现更新，这可能默认是合理的[1]。

但是目标更新通常可以获得更好的性能，因为，不需要往返服务器来获取并修改文档数据。

而且，更重要的是文档更新通常很小。如果通过替换更新，每个文档平均200KB大小，则每次更新要接受和发送200 KB数据！

记住第5章我们讲过的，使用投影来获取文档的部分字段。如果要替换文档而不丢失数据，这并非理想的选择。与之前的例子使用`$set`和`$push`更新的方法正好相反，不论文档原始的大小，这些文档更新可能每次小于100B。因此，频繁使用目标更新意味着可以在序列化和传输数据上花费更少的时间。

此外，目标操作允许原子更新文档。例如，如果通过替换更新增加计数器，就可能会出现问题。如果读/写之间的时间发生变化修改呢？唯一方式是使用乐观锁（optimistic locking）来实现原子更新。使用目标更新，可以使用`$inc`来原子性修改计数器。这意味着，即使大并发

[1]绝大部分 MongoDB 对象映射器都使用了这个策略，容易理解其原因。如果用户可以建模任意复杂的实体，然后通过替换来更新会比使用特定的更新操作符来更新简单得多。

更新，每个$inc都会隔离操作，要么成功要么失败[1]。

[1]MongoDB 文档使用了原子更新这个词语来表示目标更新。这个新的术语是为了澄清 atomic 一词。事实上，所有发送给服务器的更新都是原子性的，基于每个文档进行隔离。更新操作符被原子性调用，因为这可以让查询和更新操作在单个操作内完成。

乐观锁定模式

乐观锁定模式（optimistic locking中文有的翻译为：乐观锁，容易歧义，实际是一种并发锁定策略）或者乐观并发控制（optimistic concurrency control），是一种确保干净更新数据但是不需要锁定数据的技术。理解这个概念最好的例子就是wiki。可能同时有多个用户同时更新wiki页面，但是又绝不希望有用户更新已经过时的页面信息。因此，可以使用乐观锁定协议。当用户尝试保存修改的时候，可以包含一个尝试更新的时间戳。如果时间戳比最新保存的版本时间旧，就不允许更新。如果没有用户保存任何编辑页面，就允许更新。这个策略允许多个用户同时编辑，它比另外一种需要用户锁定页面的并发策略更好。

使用悲观锁定模式（pessimistic locking），记录会在事务里第一次访问时被锁定，直到事务结束，在此期间无法进行其他事务访问。

现在既然已经掌握了可用的更新方法，那么我们就可以来学习下一节介绍的策略了。会返回到电商数据模型，回答一些关于生产环境下操作数据更加困难的问题。

7.2 电商数据模型更新

E-commerce updates

更新MongoDB文档字段的例子很容易实现，但是对产品数据模型和真实的应用程序，复杂度会大大增加，字段的更新可能无法用一行代码实现。

下面的小节里，我们将会使用前两章提供的电商数据模型来实战练习更新操作。将会发现特定的更新非常直观，没有这么复杂。从总体上讲，我们会更好地理解第4章介绍的schema数据定义架构以及MongoDB更新语言的功能和局限性。

7.2.1 商品和目录

这里我们会看几个针对更新的例子，首先我们来看一下如何计算商品的平均评分，然后更复杂的任务是维护类别层次。

平均商品评分

商品数据模型适用于大量更新的策略。假设管理员提供了编辑商品信息的接口，那么最简单

的更新办法就是获取商品文档信息，然后与用户的编辑信息合并，使用替换策略更新文档。有时候我们可能只需要更新几个字段，而此时适用于目标更新（targeted update）方式，比如更新商品的评分。因为用户需要对商品评价进行排序，所以需要在商品里存储评分的值，当用户评分或者删除评分时再更新这个值。

这里是使用JavaScript更新操作方式的代码之一：

```
product_id = ObjectId("4c4b1476238d3b4dd5003981")
count = 0
total = 0
db.reviews.find({product_id: product_id}, {rating: 4}).forEach(
  function(review) {
    total += review.rating
    count++
  })
average = total / count
db.products.update({_id: product_id},
  {$set: {total_reviews: count, average_review: average}})
```

这段代码从每个商品的评价中计算商品评价，然后计算出平均评分。我们也可以迭代计算所有的评分。这需要调用另外的函数`count`。有了总的评分和平均评分，代码就可以使用`$set`来进行目标更新操作了。

如果不想硬编码`ObjectId`,可以使用如下代码来找到某个特定的`ObjectId`,然后再使用它：

```
product_id = db.products.findOne({sku: '9092'}, {'_id': 1})
```

对性能敏感的用户，可能害怕每次更新重新聚合计算商品的评价。这个很大程度上取决于读/写的比例，其实查看商品评价的人数会远远超过更新的人数，所以当写入的时候重新计算是必要的。这里提供的方法虽然保守，但是绝大部分情况下还可以接受，当然其他策略也是可能的。例如，我们也可以在商品文档里存储额外的字段来保存评分信息，这样可以逐步计算平均评分。在插入新的评价后，我们可以先手动查询当前总的评分和平均评分，然后计算新的平均值，进而使用选择器更新文档。代码如下：

```
db.products.update({_id: product_id},
  {
    $set: {
      average_review: average,
      ratings_total: total
    },
    $inc: {
      total_reviews: 1
    }
  })
```

这个例子使用了`$inc`操作符，它可以基于给定的值进行增加运算——这里是每次加1。

只有根据系统的测试标准，使用代表性数据测试，才能判断这些做法是否合适。这个例子展示了MongoDB提供的一个有效方式。应用需求将会帮助你来确定哪个方式是最佳的。

类别层次

大部分数据库都没有好的办法表示类别层级。MongoDB也不例外，虽然它的文档结构可以一定程度上解决这些问题。文档鼓励读优化策略，因为每个类别可能都包含父类别的信息，这些都属于非范式设计。这里有个变态的要求就是保持父类别列表的更新。我们来看一下具体如何实现这个需求。

首先，需要一个通用的更新祖先级别列表的方法，对于任意的类别都适用。这里是一个可行的解决方案：

```
var generate_ancestors = function(_id, parent_id) {
  ancestor_list = []
  var cursor = db.categories.find({_id: parent_id})
  while(cursor.size() > 0) {
    parent = cursor.next()
    ancestor_list.push(parent)
    parent_id = parent.parent_id
    cursor = db.categories.find({_id: parent_id})
  }
  db.categories.update({_id: _id}, {$set: {ancestors: ancestor_list}})
}
```

这个方法通过回溯遍历类别层级，依次查询每个节点的父类ID字段`parent_id`，直到到达根节点(`parent_id`为`null`)。此时，它构建了一个顺序的类别列表，存储在`ancestor_list`数组里。最后，它使用`$set`来更新类别的`ancestors`字段。

既然我们已经编写了基本的代码块，现在就来看看插入新类别的代码。

假设有如图7.1所示的简单类别结构。

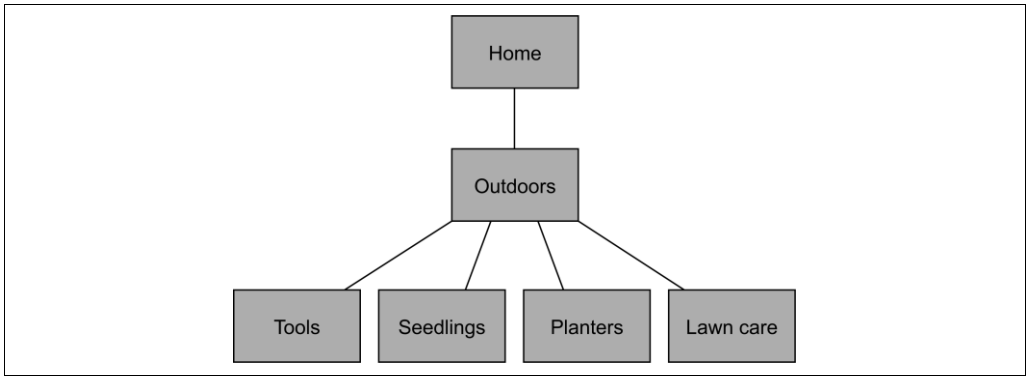


图 7.1　初始化类别层级

假设要添加一个新的类别Gardening（花园）放到Home（主页）类别下。首先，插入新文档，

然后运行新方法来生成它的父类别：

```
parent_id = ObjectId("8b87fb1476238d3b4dd50003")
category = {
  parent_id: parent_id,
  slug: "gardening",
  name: "Gardening",
  description: "All gardening implements, tools, seeds, and soil."
}
```

```
db.categories.save(category)
generate_ancestors(category._id, parent_id)
```

注意：`save()`把创建的ID放入最初的文档里。这个ID用来调用`generate_ancestors()`方法。图7.2所示为更新后的树形结构。

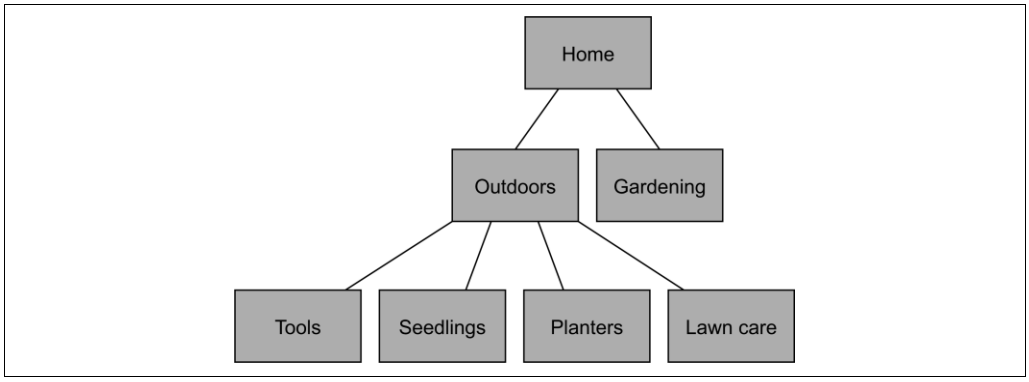


图 7.2　添加 Gardening 类

这非常简单。如果要把Outdoors类别插入到Gardening类下面呢？这就有点复杂了，因为它会修改父类别的列表。我们可以修改Outdoors类的`parent_id`为Gardening类的`_id`值就可以了。这样就不会太困难，因为我们有`outdoors_id`和`gardening_id`可用。

```
db.categories.update({_id: outdoors_id}, {$set: {parent_id: gardening_id}})
```

因为我们已经高效地移动了Outdoors类别，所以Outdoors子类别的上级列表信息都无效了。可以使用Outdoors来查询所有的子类别，然后进行修改更新父类别的操作，这样就可维护类别层级的信息。

MongoDB强大的查询功能让这个需求变得小菜一碟，代码很简单：

```
db.categories.find({'ancestors.id': outdoors_id}).forEach(
  function(category) {
    generate_ancestors(category._id, outdoors_id)
  })
```

这就是如何更新类别的`parent_id`字段，新的类别结构如图7.3所示。

如果要更新类别名字，怎么办？如果要修改Outdoors为The Great Outdoors，就必须修改其他类别的父类别列表中所有出现Outdoors的地方。你可能会想，“这就是违反范式设计数据库带来的后果”！但是，如果知道下面的真相，就可能会感觉好点，因为不需要重新计算任意的上级类别列表来执行这个更新。下面是实现代码：

```
doc = db.categories.findOne({_id: outdoors_id})
```

```
doc.name = "The Great Outdoors"
db.categories.update({_id: outdoors_id}, doc)
db.categories.update(
  {'ancestors._id': outdoors_id},
  {$set: {'ancestors.$': doc}},
  {multi: true})
```

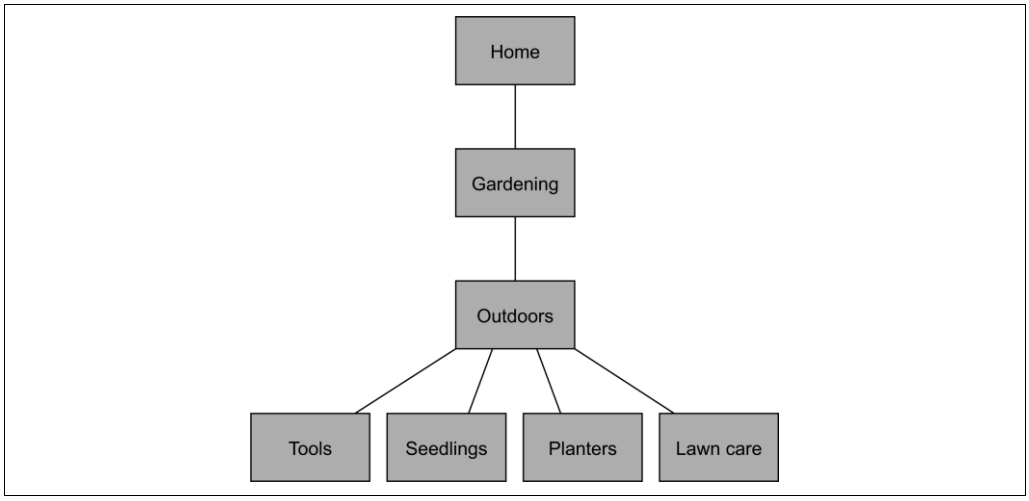


图 7.3　类别树的最终状态

首先要找到Outdoors文档，修改name字段，然后通过替换进行修改。现在，使用更新过的Outdoors文档来替换它出现过的所有列表中的上级类别信息。multi参数{multi: true}比较易于理解，它允许对所有匹配选择器的文档执行更新——如果没有{multi: true}，则只会更新第一个匹配的文档。这里我们想要更新每个包含Outdoors为上级类别的列表。

位置操作符更加微妙。假设我们不知道Outdoors类别出现在哪个类的父类列表中。这就需要我们动态定位Outdoors类别在列表中的位置。输入位置定位符。这个符号($ 在ancestors.$中)使用自己来替代匹配选择器数组元素的索引，启用更新。

下面是这个技巧的又一个例子。假设想要修改用户的地址address信息(7.1节里的例子文档)，已经标记为“work”。我们可以使用如下的代码来解决这个问题：

```
db.users.update({
      _id: ObjectId("4c4b1476238d3b4dd5000001"),
      'addresses.name': 'work'},
      {$set: {'addresses.$.street': '155 E 31st St.'}})
```

因为需要更新数组中的单个文档，所以希望保留位置操作符。通常，这个方法适用于更新类别层级信息。无论什么时候处理文档数组，我们都可以实现定位更新。

7.2.2 评价

并非所有的评价都一样，因此允许用户去投票。这些投票是原始的，可以表示那些评价是有帮助的。我们已经建模了评价，它们可以用来缓存有帮助的投票数量，并保存一个投票人的ID列表。每个相关的评价文档的结构如下所示：

```
{
  helpful_votes: 3,
  voter_ids: [
    ObjectId("4c4b1476238d3b4dd5000041"),
    ObjectId("7a4f0376238d3b4dd5000003"),
    ObjectId("92c21476238d3b4dd5000032")
  ]
}
```

我们可以使用目标更新（targeted update）来记录用户的投票。策略就是使用`$push`操作符来添加投票者的`ID`到列表中，并使用`$inc`增加投票的数量，实现代码都在一个JavaScript控制台更新操作里：

```
db.reviews.update({_id: ObjectId("4c4b1476238d3b4dd5000041")}, {
    $push: {
      voter_ids: ObjectId("4c4b1476238d3b4dd5000001")
    },
    $inc: {
      helpful_votes: 1
    }
  })
```

这个代码基本正确了，但是我们要确保只有没有投过票的用户才能投票，所以需要修改查询选择器，只有`voter_ids`数组不包含用户的`ID`才能投票。这个使用`$ne`查询操作符很容易实现：

```
query_selector = {
  _id: ObjectId("4c4b1476238d3b4dd5000041"),
  voter_ids: {
    $ne: ObjectId("4c4b1476238d3b4dd5000001")
  }
}
db.reviews.update(query_selector, {
    $push: {
      voter_ids: ObjectId("4c4b1476238d3b4dd5000001")
    },
    $inc : {
      helpful_votes: 1
    }
  })
```

这个例子专门演示了MongoDB强大的更新机制，以及它如何与面向文档的schema一起使用。投票，在这个例子里是原子性和高效的。更新是原子性的，因为查询和修改都在单个操作里

完成。原子性确保，即使在高并发环境里，它都可以确保每个用户不会投票超过1次。这个高效性依赖于在同一个请求中对于投票资格的检验，以及更新计数器和投票者列表。

现在，如果你最终使用这个技术来记录投票，特别重要的是其他针对这个评价的更新操作也要是目标性的——替换更新可能导致数据的不一致性。想象一下，用户更新了它们评价的内容，而这次更新是通过替换完成的。当通过替换更新时，首先要查询要更新的文档数据。但是在查询和更新的操作之间，可能有不同的用户来投票修改这个评价文档的数据。这叫做竞争条件（race condition）。这个事件导致的结果如图7.4所示。

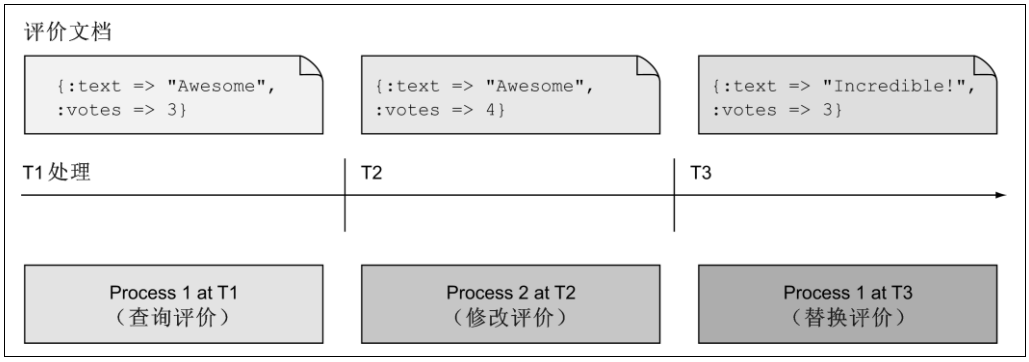


图 7.4　当通过目标和替换更新来并发更新评价时，数据可能会丢失

很明显，这个发生在T3的文档替换会覆盖T2的更新数据。可以通过使用乐观锁技术来避免这种事情发生，但是这样做需要额外的应用代码来实现乐观锁，而且确保所有的更新都是目标性的，在这个例子中更容易实现。

7.2.3　订单

我们在评价里看到更新操作的原子性和高效性都可以应用到Orders订单。特别是，我们将要看到MongoDB调用需要使用目标更新来实现`add_to_cart`函数。这是个三步过程。首先，需要构造订单商品数组里的商品文档。其次，要使用目标更新，表明这是一个upsert——如果更新的文档不存在，就创建它（在下一节里会详细介绍upsert）。最后，如果订单不存在，upsert将会创建新的订单对象，在初始和后续添加到购物车[1]的过程中实现无缝处理。

我们来构建一个例子文档，然后添加到购物车里：

[1]我们使用了可以互换的词汇购物车（shopping cart）和订单（order），因为它们都使用了相同的文档表示。它们是由文档的 state 字段区分的（CART 状态表示购物车）。

```
cart_item = {
  _id: ObjectId("4c4b1476238d3b4dd5003981"),
  slug: "wheel-barrow-9092",
  sku: "9092",
  name: "Extra Large Wheel Barrow",
  pricing: {
    retail: 5897,
    sale: 4897
  }
}
```

我们很可能通过查询products集合来构建这个文档，然后抽取需要的字段来保存到订单中。对于这个商品，_id、sku、slug、name、price字段应该足够了。接下来，使用参数{upsert: true}确保有个状态为“CART”的客户订单。

这个操作也会使用$inc操作符增加订单的商品数量sub_total：

```
selector = {
  user_id: ObjectId("4c4b1476238d3b4dd5000001"),
  state: 'CART'
}
update = {
  $inc: {
    sub_total: cart_item['pricing']['sale']
  }
}
db.orders.update(selector, update, {upsert: true})
```

通过初始化 upsert 来创建订单文档

为解析一下代码，我们分别构建了查询器和更新文档。更新文档通过商品价格增加了订单的总额。当然，用户第一次执行add_to_cart函数时，购物车是不存在的。这也是为什么我们使用upsert。用upsert将会创建由查询器定位的文档。因此初始化的upsert将会生成一个如下的订单文档：

```
{
  user_id: ObjectId("4c4b1476238d3b4dd5000001"),
  state: 'CART',
  subtotal: 9794
}
```

然后执行order的更新操作把商品添加进来。如果商品不在订单上，就新增一个：

```
selector = {user_id: ObjectId("4c4b1476238d3b4dd5000001"),
    state: 'CART',
    'line_items._id':
       {'$ne': cart_item._id}
    }

update = {'$push': {'line_items': cart_item}}
```

```
db.orders.update(selector, update)
```

更新数量

接下来，我们使用另外一个目标更新来确保商品数量的正确性。我们需要这个更新操作来处理特殊的情况，比如用户点击“Add to Cart”时，商品已经添加到购物车里了。此时，之前的更新不会再添加新商品到购物车，但是需要调整订单里商品的数量：

```
selector = {
  user_id: ObjectId("4c4b1476238d3b4dd5000001"),
  state: 'CART',
  'line_items._id': ObjectId("4c4b1476238d3b4dd5003981")
}
update = {
  $inc: {
    'line_items.$.quantity': 1
  }
}
db.orders.update(selector, update)
```

我们使用`$inc`操作符来更新单个商品的数量。使用之前介绍的位置操作符`$`更新也非常方便。因此，在用户两次点击独轮车商品的Add to Cart按钮后，购物车的结果如下所示：

```
{
  user_id: ObjectId("4c4b1476238d3b4dd5000001"),
  state: 'CART',
  line_items: [
    {
      _id: ObjectId("4c4b1476238d3b4dd5003981"),
      quantity: 2,
      slug: "wheel-barrow-9092",
      sku: "9092",
      name: "Extra Large Wheel Barrow",
      pricing: {
        retail: 5897,
        sale: 4897
      }
    }
  ],
  subtotal: 9794
}
```

应该有2辆独轮车在购物车里，商品数量应该正确地显示了2个。

要完整实现购物车还需要很多其他的操作。绝大部分操作，比如从购物车删除商品或者清空购物车时都可以使用一个或者多个目标实现。如果还不明显，接下来的内容会介绍每个查询操作符，让你理解得更清晰。实际的订单处理过程可以通过一系列状态和逻辑组合来进行处理。

我们将会在下一节里进行演示，那里我们会介绍文档的原子处理以及findAndModify命令。

7.3 原子文档处理

Atomic document processing

不想错过的一个工具就是MongoDB的findAndModify命令[1]。

这个命令允许我们在同一个往返过程中原子更新文档并返回它。原子更新就是一个不会被其他更新中断或者与其他操作交互的操作。如果用户在我们找到这个文档后修改之前尝试修改此文档呢？这个查找不会成功。原子更新会阻止这个情况，所有其他操作必须等待原子更新完成才行。

每个MongoDB更新都是原子性的，但是与findAndModify不同的是它会自动返回文档给你。为什么这非常有用呢？因为当你要获取并更新一个文档（或更新并获取它）时，可能有另外一个MongoDB用户修改这个文档，虽然更新是原子性的，但也不可能知道我们更新文档的真实状态（更新前或者后），除非使用了findAndModify。其他选择是使用7.1节提到的乐观锁机制，这需要额外的应用逻辑来实现。

原子更新非常重要是因为它支持许多功能。例如，可以使用findAndModify来构建工作队列和状态机，然后使用这些原始语句来构建事务语义，这极大地扩展了MongoDB的应用范围。使用事务性特性，我们可以使用MongoDB开发完整的电商网站——不仅仅是商品内容，还包括结账机制以及库存管理。

为了演示，我们使用了两个findAndModify命令的例子。首先，我们看一下如何处理基本的购物车状态转换。然后我们会看一个关于库存管理的稍微复杂的例子。

7.3.1 订单状态转换

所有的状态转换包含2个部分：查询、确保有效的初始状态和一个影响状态修改的更新。我们跳过一些订单处理的步骤，假设用户要点击支付（Pay Now）按钮来授权交易。如果要在应用端同步授权信用卡支付，就需要以下4步：

(1) 授权用户查看结算页面。

[1]这个命令会根据环境变化。shell 帮助类调用使用了驼峰命名法则 db.orders.findAndModify，Ruby 使用了下画线：find_and_modify。核心服务器会识别 findandmodify。如果要手动调用命令可以使用最终的格式。

(2) 在授权过程中购物车内容不变。

(3) 授权过程中的任何错误都会导致购物车返回之前的状态。

(4) 如果信用卡授权成功，支付信息会提交给订单，然后订单状态转换为PRE-SHIPPING（准备快递）。

我们使用了状态转换，如图7.5所示。

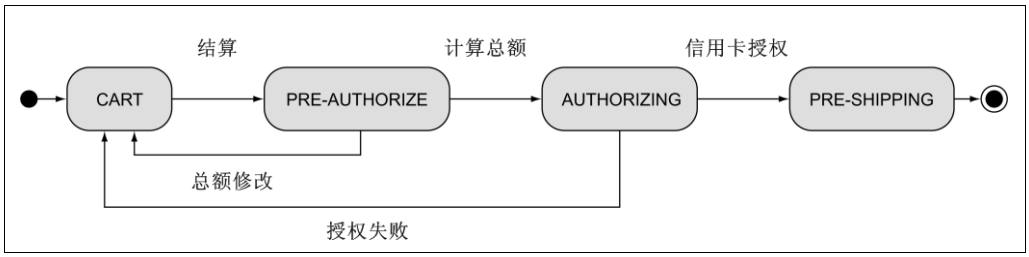


图 7.5　订单状态转换

准备结算订单

第一步要进入PRE-AUTHORIZE（预授权）状态。我们使用`findAndModify`来查找用户当前的订单对象，并确保对象在CART状态：

```
newDoc = db.orders.findAndModify({
    query: {
      user_id: ObjectId("4c4b1476238d3b4dd5000001"),
      state: 'CART'
    },
    update: {
      $set: {
        state: 'PRE-AUTHORIZE'
      }
    },
    'new': true
  })
```

如果成功，`findAndModify`会返回修改过的订单对象给`newDoc`[1]。一旦订单是PRE-AUTHORIZE（预备授权）状态，用户就不能编辑购物车内容。这是因为所有的更新都必须确保CART购物车的状态。`findAndModify`非常有用，能让我们知道当修改状态为PRE-AUTHORIZE时文档的准确状态。如果另外一个线程也在尝试修改用户的结算流程，会发生什么问题？

[1]默认情况下，`findAndModify`命令会返回文档更新之前的状态。要返回修改后的文档，就必须制定`'new': true`参数。正如这个例子一样。

检查订单和授权

现在，在预授权状态下我们获取了返回的订单对象，并且重新计算各个商铺的总和。一旦计算完成，就可以使用findAndModify把文档状态转换为AUTHORIZING。当然，新的总额必须与旧的总额匹配才行。以下就是使用findAndModify的例子代码：

```
oldDoc = db.orders.findAndModify({
    query: {
      user_id: ObjectId("4c4b1476238d3b4dd5000001"),
      total: 99000,
      state: "PRE-AUTHORIZE"
    },
    update: {
      '$set': {
        state: "AUTHORIZING"
      }
    }
  })
```

如果第二个findAndModify失败，就必须返回订单的状态为CART并且返回更新后的总额给用户。但是如果成功，授权的总额必须与展示给用户的总额相同。这意味着接下来可以继续实际的信用卡授权API的调用。因此，应用现在对用户信用卡发起了授权请求。如果信用卡授权失败，就会记录失败，并且返回订单的状态为CART。

完成订单

如果授权成功，就要把授权信息写入订单，然后转换到下一状态。下面的策略是调用findAndModify做了同样的工作。这里的例子使用文档来表示授权接受信息，它被附加到最初的订单中：

```
auth_doc = {
  ts: new Date(),
  cc: 3432003948293040,
  id: 29238382910293984483949348,
  gateway: "Authorize.net"
}
db.orders.findAndModify({
    query: {
      user_id: ObjectId("4c4b1476238d3b4dd5000001"),
      state: "AUTHORIZING"
    },
    update: {
      $set: {
        state: "PRE-SHIPPING",
        authorization: auth_doc
      }
    }
  })
```

非常重要的一点就是要知道MongoDB的功能使用了这个事务性过程。它具备原子性修改任意文档的能力，还可以确保单一连接中读取的一致性。最后，文档结构自身允许这些操作适应MongoDB提供的单个文档的原子性。在这个例子中，这个结构允许我们把商品列表、价格、用户资料保存到单个文档里，确保我们只需要在单个文档上操作就可以了，提升了伸缩性。

这应该会让你印象深刻。同时它也可能让你好奇，MongoDB里能否支持多个对象（multi-object）的事务操作？

7.3.2 库存管理

并非每一个点上的网站都需要严格的库存管理。大部分商品都有足够的时间来补足库存而不需要理会实际的库存。在这样的情况下，管理库存就非常随意，只需要管理预期就可以了，只要库存很少的时候调整一下快递预期。

一个商品属于一类的情况带来了另外的挑战。设想一下我们正在销售音乐会门票，并且带有固定的座位或者手工艺术品。这些商品无法重复，用户通常需要确保他们可以买到自己选择的商品。这里我们介绍一个使用MongoDB的解决方案。这将会展示`findAndModify`命令的性能，以及表明使用这个文档模型是非常明智的选择；也会展示如何在多个文档中实现事务性语义。虽然我们只会看到此过程中的一些关键的MongoDB调用，但是完整的`InventoryFetcher`类的代码也包含在本书中。

建模库存的最佳理解方式就是通过真实的商店例子。如果你去过花园商店，就可以看到和感觉到实际的物理库存：铁锹、钉耙和剪刀放在货架上。如果选择了铁锹并放入购物车，那么顾客可选的铁锹数量就会少一把。作为推论，2个顾客不能同时选择相同的铁锹放到购物车里。我们可以使用这个简单的原则来建模。对于每个仓库中的物品，集合中都存储一个文档。如果有10把铁锹，在数据库里就有10个有关铁锹的文档。每个库存项目都会通过sku连接到一个商品上，每个项目可以有4个状态：`AVAILABLE` (0), `IN_CART` (1)，`PRE_ORDER` (2)，`PURCHASED` (3)。

这里的代码插入铁锹、钉耙和剪刀作为可用的3个状态。例子代码使用Ruby编写，因此事务需要更多的逻辑，所以看看具体如何使用的例子会更有帮助：

```
3.times do
  $inventory.insert_one({:sku => 'shovel', :state => AVAILABLE})
  $inventory.insert_one({:sku => 'rake', :state => AVAILABLE})
  $inventory.insert_one({:sku => 'clippers', :state => AVAILABLE})
end
```

我们将会使用专门的库存提取的类来处理库存管理。首先我们来看一下这个类如何工作，然后会详细看一下它的实现代码。

库存提取器

用这个库存提取器可以向购物车里添加任意商品的集合。这里我们创建了一个新的订单对象和一个新的库存提取器。然后让提取器添加3把铁锹和一些剪刀到订单里，通过订单ID识别订单，用另外2个文档指定商品和数量，传递给方法`add_to_cart`。此提取器隐藏了操作的复杂性，它可以一次性修改2个集合：

```
$order_id = BSON::ObjectId('561297c5530a69dbc9000000')
$orders.insert_one({
    :_id => $order_id,
    :username => 'kbanker',
    :item_ids => []
  })

@fetcher = InventoryFetcher.new({
    :orders => $orders,
    :inventory => $inventory
  })

  @fetcher.add_to_cart(@order_id,
    [
      {:sku => "shovel", :quantity => 3},
      {:sku => "clippers", :quantity => 1}
    ])

    $orders.find({"_id" => $order_id}).each do |order|
    puts "\nHere's the order:"
    p order
end
```

如果添加项目到购物车里，`add_to_cart`方法将会抛出一个异常。如果成功，订单应该如下所示：

```
{
  "_id" => BSON::ObjectId('4cdf3668238d3b6e3200000a'),
  "username" => "kbanker",
  "item_ids" => [
    BSON::ObjectId('4cdf3668238d3b6e32000001'),
    BSON::ObjectId('4cdf3668238d3b6e32000004'),
    BSON::ObjectId('4cdf3668238d3b6e32000007'),
    BSON::ObjectId('4cdf3668238d3b6e32000009')
     ]
  }
```

每个物理库存项目的`_id`将会存储到单独的订单文档里。人们可以如下方式来查询每个项目：

```
puts "\nHere's each item:"
```

```
order['item_ids'].each do |item_id|
  item = @inventory.find({"_id" => item_id}).each do |myitem|
    p myitem
  end
end
```

单独看每个商品项目，可以看到每个都有一个状态1，对应的状态是IN_CART。应该注意到每个条目记录了状态修改的最后时间。我们可以在后面使用这个时间戳来过期已经长时间存储在购物车里的物品。例如，我们可能给用户15分钟来结算订单，购物车的商品可以存放15分钟：

```
{
  "_id" => BSON::ObjectId('4cdf3668238d3b6e32000001'),
  "sku"=>"shovel",
  "state"=>1,
  "ts"=>"Sun Nov 14 01:07:52 UTC 2010"
}
{
  "_id"=>BSON::ObjectId('4cdf3668238d3b6e32000004'),
  "sku"=>"shovel",
  "state"=>1,
  "ts"=>"Sun Nov 14 01:07:52 UTC 2010"
}
{
  "_id"=>BSON::ObjectId('4cdf3668238d3b6e32000007'),
  "sku"=>"shovel",
  "state"=>1,
  "ts"=>"Sun Nov 14 01:07:52 UTC 2010"
}
```

库存管理

如果InventoryFetcher的API合理，就应该对于如何实现库存管理有了初步的想法。不出意外，findAndModify命令会出现在核心代码中。InventoryFetcher的源码出现在本书的源代码中。我们不会在本书里列出每一行代码，但是我们将会重点介绍其中的3个方法。

首先，当要传递列表项目给购物车时，fetcher获取器会尝试把每个商品的状态AVAILABLE转换为IN_CART。如果任意一个操作失败（某一个添加失败），整个操作就会回滚。来看一下我们之前调用的方法add_to_cart：

```
def add_to_cart(order_id, *items)
  item_selectors = []
  items.each do |item|
    item[:quantity].times do
      item_selectors << {:sku => item[:sku]}
    end
  end
  transition_state(order_id, item_selectors,
      {:from => AVAILABLE, :to => IN_CART})
```

```
end
```

*items语法允许用户传递任意数量的对象，它被放置到一个名为items的数组里。这个方法工作不多，它把商品添加到购物车里，并且扩展数量以保证每个要添加到购物车的商品项目都有一个物理商品对应一个选择器。例如，这个文档表示我们要添加2把铁锹到购物车里：

```
{:sku => "shovel", :quantity => 2}
```

变成这个:

```
[{:sku => "shovel"}, {:sku => "shovel"}]
```

每个要添加到购物车的商品都有单独的查询选择器。因此，这个方法传递商品选择器的数组给transition_state方法。例如，之前的代码指定了状态应该从AVAILABLE转换到IN_CART：

```
def transition_state(order_id, selectors, opts={})
  items_transitioned = []
  begin # 使用 begin/end 块可以为选择器进行错误恢复
    query = selector.merge({:state => opts[:from]})
    physical_item = @inventory.find_and_modify({
        :query => query,
        :update => {
          '$set' => {
            :state => opts[:to],    # 目标状态
            :ts => Time.now.utc     # 获取当前客户端时间
          }
        }
      })
      if physical_item.nil?
        raise InventoryFetchFailure
      end

      items_transitioned << physical_item['_id'] #  项目加入数组中
      @orders.update_one({:_id => order_id}, {
          '$push' => {
            :item_ids => physical_item['_id']
          }
        })
    end # 结束循环

  rescue Mongo::OperationFailure, InventoryFetchFailure
    rollback(order_id, items_transitioned, opts[:from], opts[:to])
    raise InventoryFetchFailure, "Failed to add #{selector[:sku]}"
  end

  return items_transitioned.size
end
```

要进行状态转换，每个选择器获取一个额外的条件{:state => AVAILABLE}，同时选择器

发送给findAndModify，如果匹配，就修改商品时间戳和新状态。这个方法保存转换状态的项目，并且使用新的商品ID更新订单。

优雅的失败

如果findAndModify命令失败，并返回nil，就抛出一个InventoryFetchFailure异常。如果是因为网络问题失败，就抛出Mongo::OperationFailure异常。在两种情况下，都可以通过回滚目前的转换状态并抛出InventoryFetchFailure异常，它包含无法添加的商品SKU。我们也可以在应用代码里优雅地处理这个异常。

现在剩下的全部工作就是检查回滚代码了：

```
def rollback(order_id, item_ids, old_state, new_state)
  @orders.update_one({"_id" => order_id},
                     {"$pullAll" => {:item_ids => item_ids}})
  item_ids.each do |id|
    @inventory. find_one_and_update({
        :query => {
          "_id" => id,
          :state => new_state
        }
      },
      {
        :update => {
          "$set" => {
            :state => old_state,
            :ts => Time.now.utc
          }
        }
      }
    })
  end
end
```

我们使用$pullAll操作符来删除所有添加到订单item_ids数组的ID。然后迭代ID列表，转换每个商品的状态为旧的状态。$pullAll操作符与其他更新操作符一样会在7.4.2小节里详细介绍。

transition_state方法可以用作其他方法的基础，其他方法可以继续转换这些商品的状态。把它集成到订单转换系统中应该不困难，这里就留给读者作为练习作业了。

这实现代码里忽略的一个情况就是，库存商品无法恢复到它们的初始状态。如果Ruby驱动无法与MongoDB通信，或者在完成之前过程运行了回滚挂起，这就会让购物车商品留在IN_CART状态，但是订单集合不会包含库存。此种情况下，管理交易变得困难。这些都会最终被修复，但是，通过购物车超时可以删除这些超时的商品。

你可能会怀疑这个系统作为生产环境的健壮性。这个问题不能轻易回答，除非了解更多的特殊性，但是可以肯定的是，MongoDB提供了足够的功能来确保一个可用的事务性交易解决方案。MongoDB从未打算支持多个集合的事务，但是它允许用户使用`find_one_and_update`和乐观并发控制来模拟这个行为。如果发现经常要管理事务，则最好考虑使用不同的schema或者数据库。并非所有的应用都适合MongoDB，但是如果仔细规划schema就可以避免这种事务的需求。

7.4 核心要点：MongoDB 更新与删除

Nuts and bolts: MongoDB updates and deletes

要理解MongoDB更新机制，就需要完整理解MongoDB的文档模型和查询语言，之前章节的例子就是最好的帮助资料。正如本章开始承诺的，我们将会深入谈到一些本质问题。这会涉及每个MongoDB更新接口的功能总结，我们也会介绍几个性能问题的注意事项。为了简洁起见，后面的例子我们大部分使用JavaScript代码编码。

7.4.1 更新类型与参数选项

正如我们之前例子展示的，MongoDB支持目标更新和替换更新。前者通过使用一个或多个操作符定义；后者通过使用文档来替换匹配查询器的文档实现。

注意，如果更新文档非常含糊，那么更新可能失败。这是使用MongoDB的常见问题，而且是非常容易犯的错误。这里，我们已经集合使用了更新操作符`$addToSet`，使用了替换语义`{name:"Pitchfork"}`：

```
db.products.update_one({}, {name: "Pitchfork", $addToSet: {tags: 'cheap'}})
```

如果要修改文档的名字，就必须使用`$set`操作符：

```
db.products.update_one({},
{$set: {name: "Pitchfork"}, $addToSet: {tags: 'cheap'}})
```

多文档更新

默认情况下，只会更新匹配查询器的第一个文档。要更新所有的匹配文档，就需要显示指定多文档更新模式。在shell里，可以通过添加参数`multi: true`来实现。以下是如何在商品集合里添加`cheap`标签到所有的文档中：

```
db.products.update({}, {$addToSet: {tags: 'cheap'}}, {multi: true})
```

在文档级别更新是原子性的,这意味着一条更新10个文档的语句可能在更新3个文档后由于某些原因失败。应用程序必须根据自己的策略来处理这些失败。

使用Ruby驱动(以及其他驱动),可以使用与以下相似的方式来处理多个文档更新:

```
@products.update_one({},
    {'$addToSet' => {'tags' => 'cheap'}},
    {:multi => true})
```

UPSERTS

有个问题非常常见,就是当文档存在时更新,文档不存在时插入数据。我们可以使用upsert来处理这种常见的问题。如果查询选择器匹配,更新就正常执行。如果没有匹配的文档,就会插入新的文档。新文档的字段是查询选择器和目标更新文档的逻辑合并[1]。

下面是shell使用upsert的简单例子,设置`upsert: true`参数来允许upsert:

```
db.products.update({slug: 'hammer'},
                   {$addToSet: {tags: 'cheap'}}, {upsert: true})
```

以下是等价的Ruby实现upsert的代码:

```
@products.update_one({'slug' => 'hammer'},
  {'$addToSet' => {'tags' => 'cheap'}}, {:upsert => true})
```

正如你所期望的,upserts一次可以插入或者更新一个文档。当需要更新原子性和不确定文档是否存在时,upserts非常有用。对于特别的例子,参考7.2.3,它描述了添加商品到购物车的过程。

7.4.2 更新操作符

MongoDB支持一系列更新操作。这里我们介绍一下每个操作符的简单使用例子。

标准操作符

第一个操作符集合是最通用的,每个都可以用来处理任意数据类型。

$inc

可以使用`$inc`操作符来增加或者减少一个数值:

[1]注意:upsert 无法与替换更新文档模式一起工作。

```
db.products.update({slug: "shovel"}, {$inc: {review_count: 1}})
db.users.update({username: "moe"}, {$inc: {password_retries: -1}})
```

也可以使用$inc来添加或者减去任意的数值：

```
db.readings.update({_id: 324}, {$inc: {temp: 2.7435}})
```

$inc的高效就如其便捷性一样。因为它很少用于修改文档的大小，$inc通常发生在磁盘上，因此只影响指定的值[1]。

之前的观点只适用于MMAPv1存储引擎。WiredTiger存储引擎的工作方式不同，它使用先写事务日志的方式，与检查点一起使用以确保数据的一致性。

正如往购物车里添加商品的演示代码一样，$inc与upserts一起使用。例如，可以修改之前的更新代码为upsert，如下所示：

```
db.readings.update({_id: 324}, {$inc: {temp: 2.7435}}, {upsert: true})
```

如果_id为324的文档不存在，就会创建一个新的文档，_id是324，temp的增加值为2.7435。

$set 和$unset

如果要设置某个文档里的特定key值，就可以使用$set。

我们可以为key设置任意有效的BSON类型值。这意味着下面的代码都是有效的：

```
db.readings.update({_id: 324}, {$set: {temp: 97.6}})
db.readings.update({_id: 325}, {$set: {temp: {f: 212, c: 100}}})
db.readings.update({_id: 326}, {$set: {temps: [97.6, 98.4, 99.1]}})
```

如果key键已经存在，它的值就会被重写；否则，创建新的key值。

$unset会从文档中删除提供的key。以下是从文档里删除temp键的例子代码：

```
db.readings.update({_id: 324}, {$unset: {temp: 1}})
```

我们也可以在嵌入式文档和数组里使用$unset。两种情况下，都使用了原点符号来指定内部对象。如果集合里存在以下2个文档：

```
{_id: 325, 'temp': {f: 212, c: 100}}
{_id: 326, temps: [97.6, 98.4, 99.1]}
```

就可以在第一个文档里删除华氏摄氏度（Fahrenheit）以及第二个文档里的“0(zeroth)”元素，

[1]当数值类型修改时根据这个规则会抛出异常。如果用$inc 把 32 位整数转换为 64 位整数，那么整个 BSON 文档都会被替换重写。

如下所示：

```
db.readings.update({_id: 325}, {$unset: {'temp.f': 1}})
db.readings.update({_id: 326}, {$pop: {temps: -1}})
```

原点访问子文档或者数组元素也可以用到$set中。

数组中使用$unset

注意，在单个数组元素上使用$unset可能无法像我们期望的那样准确地工作。它仅仅会设置元素的值为null而不是删除元素。要完全删除数组元素，可以看一下$pull和$pop操作符：

```
db.readings.update({_id: 325}, {$unset: {'temp.f': 1}})
db.readings.update({_id: 326}, {$unset: {'temps.0': 1}})
```

$rename

如果要修改key的名字，可以使用$rename：

```
db.readings.update({_id: 324}, {$rename: {'temp': 'temperature'}})
```

也可以修改子文档的名字：

```
db.readings.update({_id: 325}, {$rename: {'temp.f': 'temp.fahrenheit'}})
```

$SetOnInsert

在upsert中，有时候要注意，不能重写某些数据。这时，若只想新增的数据就会非常有用，而不会修改数据。这就是setOnInsert操作符的由来：

```
db.products.update({slug: 'hammer'}, {
    $inc: {
      quantity: 1
    },
    $setOnInsert: {
      state: 'AVAILABLE'
    }
  }, {upsert: true})
```

我们要增加某个库存项目的数量而不受状态影响，状态的默认值是'AVAILABLE'。如果执行插入，qty的值是1，状态会设置为默认值。如果执行更新，就只会插入qty。MongoDB 2.4 增加的$setOnInsert操作符就是用于处理这种问题的。

数组更新操作符

数组在MongoDB文档模型中非常重要。自然而然地，MongoDB提供了许多操作数组的操作符。

$push、$pushAll 和 $each

如果要在数组后面追加值，$push是最好的选择。默认情况下，它会在数组尾部添加一个单独的元素。例如，在商品标签里添加一个新的标签，代码非常简单：

```
db.products.update({slug: 'shovel'}, {$push: {tags: 'tools'}})
```

如果要在一次更新里添加多个更新，则可以组合使用$each与$push操作符：

```
db.products.update({slug: 'shovel'},
  {$push: {tags: {$each: ['tools', 'dirt', 'garden']}}})
```

注意，可以向数组里添加任意类型的值，而不仅仅是增加同类型值。例如，7.3.2小节，可以添加一个商品到购物车数组里。

MongoDB 2.4版本之前，使用$pushAll操作符来添加多个值到数组上。这个方法在2.4及其以后的版本里仍然可以使用，但是已经是过时的方法，因为$pushAll可能在以后的版本中完全删除。$pushAll操作符的使用如下所示：

```
db.products.update({slug: 'shovel'},
  {$pushAll: {'tags': ['tools', 'dirt', 'garden']}})
```

$slice

$slice操作符是在MongoDB 2.4里添加的，其目的是方便管理经常更新的数组。当向数组添加值但是不想数组太大的时候，这个操作符非常有用。它必须与$push、$each操作符一起使用，允许用来剪短数组的大小、删除旧的值。传递给$slice的参数必须是小于或者等于0。这个参数的值是数组里允许的项目数量乘以 - 1。

对这个语义有点疑惑，所以来看一下具体的例子。假设要更新下面的文档：

```
{
  _id: 326,
  temps: [92, 93, 94]
}
```

则可以使用如下的命令来更新文档：

```
db.temps.update({_id: 326}, {
    $push: {
```

```
      temps: {
        $each: [95, 96],
        $slice: -4
      }
    }
  })
```

这是完美的语法。这里传递了 - 4给$slice操作符。在更新后，文档的数据如下：

```
{
  _id: 326,
  temps: [93, 94, 95, 96]
}
```

在推送数据到数组后，从开头删除了值，只有4个数留下。如果传递-1给$slice操作符，结果数组是[96]。如果传递0，结果就是[]，为空数组。注意，从MongoDB 2.6开始也可以传递正整数了。如果传递给$slice的是正整数，它就会从数组尾部开始删除元素。

在之前的例子中，如果使用了$slice: 4，结果将是

```
temps: [92, 93, 94, 95].
```

$sort

与$slice很像，MongoDB 2.4新增了$sort操作符，帮助更新数组。当使用$push和$slice时，有时候要先排序文档再删除它们。思考下面的文档：

```
{
  _id: 300,
  temps: [
    { day: 6, temp: 90 },
    { day: 5, temp: 95 }
  ]
}
```

我们有个数组子文档。当添加子文档到数组中并分割它时，首先要确定按照日期排序，保留更高的day值。我们可以使用如下的更新实现这个目标：

```
db.temps.update({_id: 300}, {
    $push: {
      temps: {
        $each: [
          { day: 7, temp: 92 }
        ],
        $slice: -2,
        $sort: {
          day: 1
        }
      }
    }
  })
```

当运行更新时，首先根据day排序temps数组，这样最小的值在开头部分。然后就可以把数组分割为两个部分了。结果是保留两个更高day值的子文档：

```
{
  _id: 300,
  temps: [
    { day: 6, temp: 90 },
    { day: 7, temp: 92 }
  ]
}
```

这里使用$sort操作符需要$push、$each、$slice。虽然很有用，但是这种处理的是非常少见的情况，我们可能不会经常使用$sort更新。

$addToSet 和 $each

使用$addToSet也会往数组后面添加值，但是它更加特殊：它只会添加向数组里添加不存在的值。因此，如果铁锹已经存在于tool标签中了，则下面的代码就不会更新文档：

```
db.products.update({slug: 'shovel'}, {$addToSet: {'tags': 'tools'}})
```

如果需要一次添加多个值到数组里，就必须组合使用$each 和$addToSet操作符。这是例子代码：

```
db.products.update({slug: 'shovel'},
  {$addToSet: {tags: {$each: ['tools', 'dirt', 'steel']}}})
```

只有不存在于tags的$each值才会被追加。注意，$each只能和$addToSet、$push操作符一起使用。

$pop

从数组中删除元素的最基本的方式就使用$pop操作符。

如果$push追加了一个项目到数组中，$pop会删除最后一个推进的项目。虽然它经常与$push运算符搭配使用，但是我们也可以单独使用$pop运算符。如果数组标签tags包含值['tools', 'dirt', 'garden', 'steel']，下面的$pop操作会删除steel标签：

```
db.products.update({slug: 'shovel'}, {$pop: {'tags': 1}})
```

与$unset类似，$pop的语法是{$pop: {'elementToRemove': 1}}。但是与$unset不同，$pop接受第二个可能的值–1来删除数组的第一个元素。

下面是如何从数组中删除tools标签的代码：

```
db.products.update({slug: 'shovel'}, {$pop: {'tags': -1}})
```

有一点可能比较遗憾的是，无法返回$pop从数组里删除的元素。因此，$pop的名字和实际的栈操作运算的结果不太一样。

$bit

如果使用过按位运算的操作符，就会发现自己可能需要在更新操作里使用相同的计算。按位运算经常在单个的二进制级别来执行逻辑运算。常见的例子是（特别是C语言）使用按位运算操作来传递标志位。换句话说，如果一个整数二进制的第4位是1，就可以使用某些条件，进而执行代码。

通常有更聪明和适合的方法来解决这个问题，但是这种存储会保持最小尺寸，并满足现存系统的工作。MongoDB提供了$bit操作符来进行按位运算，或（OR）和与（AND）在更新中都可以使用。

我们来看一下在MongoDB存储位敏感的数据，并且更新它们。UNIX文件权限通常这样存储。如果在UNIX里运行ls -l命令，就会看到drwxr-xr-x。第一个标志d表示文件是个目录，r表示文件权限，w表示写权限，x表示执行权限。这些标志有3块，分别控制不同的用户权限、用户组、每个人。因此，这个例子假设每个用户有所有的权限，但是其他只有读取和执行的权限。

有时候权限使用单个数字表示，使用二进制数据的位表示权限。x用1表示，w用2表示，r用4表示。因此你可以7来表示二进制格式的111，或者rwx。也可以使用5来表示二进制格式的101，或者使用r-x。也可以使用3来表示二进制格式的011，或-wx。

我们来使用这些特性在MongoDB里存储一个变量。从下面这个文档开始：

```
{
  _id: 16,
  permissions: 4
}
```

4此时可以表示二进制格式的100，或者r--。也可以使用按位运算符OR来添加权限：

```
db.permissions.update({_id: 16}, {$bit: {permissions: {or: NumberInt(2)}}})
```

在JavaScript shell里，必须使用NumberInt()，因为它默认使用了double类型。结果文档保护了一个二进制格式的100和010 或OR运算，结果是110，也就是十进制的6：

```
{
  _id: 16,
  permissions: 6
```

```
}
```

对于二进制运算，我们可以使用AND代替OR。这又是一个少见的情况，不会经常使用，但是在某些特定情况下非常有用。

$pull 和$pullall

$pull是$pop的复杂形势。使用$pull，可以通过值精确指定要删除的元素。回到tags的例子，如果需要删除dirt标签，不需要知道它在数组中的位置，只需要告诉$pull操作符要删除哪个值即可：

```
db.products.update({slug: 'shovel'}, {$pull: {tags: 'dirt'}})
```

$pullAll与$pushAll的工作非常类似，允许提供要删除值的列表。

要同时删除dirt和garden标签，可以使用$pullAll，代码如下：

```
db.products.update({slug: 'shovel'},
  {$pullAll: {'tags': ['dirt', 'garden']}})
```

$pull的一个强大特性就是可以传递查询作为参数来选择要拉取的元素。思考下面的文档：

```
{_id: 326, temps: [97.6, 98.4, 100.5, 99.1, 101.2]}
```

假设要删除温度大于100的值，查询文档如下：

```
db.readings.update({_id: 326}, {$pull: {temps: {$gt: 100}}})
```

修改后的文档结构如下：

```
{_id: 326, temps: [97.6, 98.4, 99.1]}
```

定位更新

MongoDB里经常会使用子文档来建模数据，但是这些子文档操作起来并不方便，直到有了定位操作符。

定位操作符允许我们通过原点选择器来定位要更新的元素。例如，有如下的订单文档：

```
{
  _id: ObjectId("6a5b1476238d3b4dd5000048"),
  line_items: [
    {
      _id: ObjectId("4c4b1476238d3b4dd5003981"),
      sku: "9092",
      name: "Extra Large Wheelbarrow",
      quantity: 1,
      pricing: {
        retail: 5897,
        sale: 4897
      }
    },
    {
      _id: ObjectId("4c4b1476238d3b4dd5003982"),
      sku: "10027",
      name: "Rubberized Work Glove, Black",
      quantity: 2,
      pricing: {
        retail: 1499,
        sale: 1299
      }
    }
  ]
}
```

若想设置第二行项目的数量为5，使用的SKU为10027，但我们不知道`line_items`数组保存在哪里，也不知道它是否存在。我们可以使用简单的查询器和位置操作符来解决上面这些问题：

```
query = {
  _id: ObjectId("6a5b1476238d3b4dd5000048"),
  'line_items.sku': "10027"
}
update = {
  $set: {
    'line_items.$.quantity': 5
  }
}
db.orders.update(query, update)
```

位置操作符`$`就是我们在`line_items.$.quantity`字符串里看到的。如果查询选择器匹配，

则SKU为10027的文档索引会内部替换位置操作符，从而更新正确的文档。

如果数据模型包含子文档，就会发现在执行细微的文档更新时位置操作符非常有用。

7.4.3 findAndModify 命令

本章介绍了许多使用findAndModify命令的例子，现在只剩下在JavaScript shell里使用这些操作参数了。这里是使用findAndModify的简单例子：

```
doc = db.orders.findAndModify({
    query: {
      user_id: ObjectId("4c4b1476238d3b4dd5000001"),
    },
    update: {
      $set: {
        state: "AUTHORIZING"
      }
    }
  })
```

有许多参数可以控制此命令的功能。下面的参数是query、update、remove需要的：

- query——查询选择器，默认为{}。
- update——指定更新的文档，默认是{}。
- remove——布尔值，如果为true，则返回删除的对象。默认是false。
- new——布尔值，如果为true，则返回修改后的文档。默认是false，意味着返回最初的文档。
- sort——指定排序的方向。使用findAndModify一次就修改一个文档，sort参数可以帮助控制要处理哪个文档。例如，可以根据{created_at: -1}排序来处理最近创建的文档。
- fields——如果只要返回部分字段，就使用这个参数来指定它们。这在处理大文档时非常有用。这些指定可以在任意查询里指定。可以查看第5章的例子。
- upsert——布尔值，为true时，findAndModify作为upsert操作。如果文档不存在就创建它。注意，如果要返回新创建的文档，就需要指定{new: true}。

7.4.4 删除

删除文档也会有许多挑战。我们可以从集合中删除整个文档，也可以通过传递查询器给

remove方法来删除部分文档。删除所有的评价文档数据则非常简单：

```
db.reviews.remove({})
```

但是通常我们可能要删除某个用户的评价：

```
db.reviews.remove({user_id: ObjectId('4c4b1476238d3b4dd5000001')})
```

所有调用remove方法都会使用查询选择器来指定要删除的文档。但是我们可能还会对并发和原子性有些疑问。将在下一节里解答这些问题。

7.4.5 并发、原子性和隔离

很重要的一点就是要理解MongoDB里并发的工作原理。在MongoDB 2.2之前，锁策略非常简单，单个读/写锁驻留在整个MongoDB实例中。这意味着，在任意时刻，MongoDB只允许一个写或者多个读（不是2个）操作。在MongoDB 2.2版本里改成了数据库级别的锁，意味着语义上在数据库级别而不是整个MongoDB实例级别使用锁； 数据库可以有一个写者或者多个读取者。在MongoDB 3.0版本中，WiredTiger存储引擎工作在集合级别，提供了更加强大的文档级别的锁。其他的存储引擎可能提供了类似的特性。

锁特性听起来比实际更糟糕，那是因为针对锁很少有优化措施。其中之一就是数据库在内存里保留文档的内部映射。对于非RAM里的读/写，数据库都会屈服于其他操作，直到文档加入内存。

第二个优化就是针对写锁。如果某个写操作需要长时间来完成，那么所有其他的读和写操作都会被阻塞。所有的插入、更新和删除都需要写入锁。插入很少需要很长时间。但是更新会影响整个集合，还有删除也会影响许多文档，可能需要很长时间才能完成。当前的解决方案允许这种长时间运行的操作为其他读/写让路。当一个操作屈服时，它会暂停，并释放锁，后面再重新启动。

尽管对于锁机制进行了优化，但是MongoDB在大量读/写的情况下还是会影响性能。一种好而简单的避免问题的方法就是把高并发的集合保存到单独的数据库里，特别是使用MMAPv1存储引擎时。但是正如之前提到的，这种情况在MongoDB 3.0好多了，因为WiredTiger存储引擎工作在集合级别，而不是数据库级别。

当更新和删除文档时，这种退让机制可能喜忧参半。设想我们要在其他操作发生之前更新或者删除所有的文档。此时，我们可以使用专门的参数$isolated来保持操作独立，不会让路。可以为查询选择器添加$isolated操作符：

```
db.reviews.remove({user_id: ObjectId('4c4b1476238d3b4dd5000001'),
  $isolated: true})
```

同样的代码可以应用到多个操作中。它会强制独立隔离完成多个更新。

```
db.reviews.update({$isolated: true}, {$set: {rating: 0}}, {multi: true})
```

这个更新会把每个评价设置为0。这个操作是隔离进行的，操作不会屈服于其他操作，确保系统的一致性视图。

注意，如果使用`$isolated`半路失败了，就不会有显示的回滚操作。部分文档已经更新而其他部分还是原来的值。MongoDB 2.2之前，`$isolated`操作符叫做`$atomic`，这个名字已经过时，因为这些操作失败在于是非原子性的。`$isolated`操作符无法在分片集合中使用，意味着我们使用的时候必须注意这个问题。

7.4.6 更新性能注意事项

下面的问题只适用于MMAPv1存储引擎，它是当前默认的存储引擎。第10章讲解了WiredTiger引擎，它与MMAPv1的工作方式不同，更加高效。如果对WiredTiger感兴趣，现在就可以阅读第10章的内容！

经验表明，理解MongoDB如何更新磁盘上的文档数据可以帮助我们优化性能。首先要理解的事情就是更新发生的程度。理想情况下，更新会影响磁盘上BSON文档的最小部分，因为这样是最高效的。但是这种情况不会经常发生。

更新磁盘上的文档有三种方式。第一种更新是最高效的，当只更新文档里的单个值并且BSON文档大小不会改变时才发生。这通常发生在`$inc`操作符中。因为`$inc`只增加整数，磁盘上的文件大小不会发生变化。如果整数代表`int`，那么在磁盘上只占用4个字节，长整数和double类型会占用8个字节。修改这些数值不会改变存储空间的大小，只需要把文档的只重新写入磁盘即可。

第二种更新是修改文档的大小和数据结构。BSON文档表示为字节数组，前4个字节表示文档的大小。因此使用`$push`操作符修改文档，既增加文档的大小又修改结构。这需要在磁盘上重写整个文档。虽然这样还不是特别低效率，但是值得我们注意。如果有大型文档——假设4MB——我们要在此文档里添加数组，那么在服务端会是很大的工作。这意味着如果要做许多的更新操作，最好尽量保持文档较小。

第三种更新是重写一个文档。如果文档扩大，但是不能满足现在的磁盘空间，则不仅仅需要

重写，还需要移动到新的空间里。这种移动操作如果经常发生，则会非常昂贵。MongoDB会通过动态调整集合分配预留空间的填充因子（padding factor）来优化这个问题。这意味着当在一个集合中有许多的需要文档迁移更新时，内部预留空间的填充因子会增加。填充因子会乘以每个插入文档的大小来获取额外的空间。这也许可以减少未来文档重新迁移的数量。此外，MongoDB 3.0使用了2的幂来作为MMAPv1存储引擎默认的记录空间分配大小。

要看下某个集合的填充因子，运行下面的命令即可：

```
db.tweets.stats()
{
  "ns" : "twitter.tweets",
  "count" : 53641,
  "size" : 85794884,
  "avgObjSize" : 1599.4273783113663,
  "storageSize" : 100375552,
  "numExtents" : 12,
  "nindexes" : 3,
```

```
  "lastExtentSize" : 21368832,
  "paddingFactor" : 1.2,
  "flags" : 0,
  "totalIndexSize" : 7946240,
  "indexSizes" : {
    "_id_" : 2236416,
    "user.friends_count_1" : 1564672,
    "user.screen_name_1_user.created_at_-1" : 4145152
  },
  "ok" : 1
}
```

这个推文tweets的集合的填充因子是1.2，这表示当插入100个字节的文档时，MongoDB会在磁盘上分配120个字节。默认的填充因子是1，表示不会分配额外的空间。

现在，简要强调下，当数据超过RAM大小或总需要极大地写入负载时，这里提到的问题特别对于部署情况要格外注意。此时,重写或者移动文档都会产生很高的成本。随着伸缩MongoDB应用，仔细思考以下最佳的更新方式，比如用`$inc`来避免这些开销。

7.5 复习更新操作符

Reviewing update operators

表7.1 列举了本章之前讨论的更新操作符。

表 7.1 操作符

操作符	
$inc	根据给定的值增加字段
$set	设置字段为给定的值
$unset	取消设置字段
$rename	重命名字段为给定的值
$setOnInsert	在 upsert 中，只在插入时设置字段
$bit	只执行按位更新字段
数组操作符	
$	根据查询选择器定位要更新的子文档
$push	添加值到数组中
$pushAll	添加数组到一个数组中。过期，被$each 取代
$addToSet	添加值到数组中，重复了也不处理
$pop	从数组中删除第一个或者最后一个值
$pull	从数组中删除匹配查询条件的值
$pullAll	从数组中删除多个值

续表

数组运算符修饰符	
$each	与$push 和$addToSet 一起使用来操作多个值
$slice	与$push 和$each 一起使用来缩小更新后数组的大小
$sort	与$push、$each、$slice 一起来排序数组中的子文档
隔离运算符	
$isolated	隔离其他操作，不允许其他操作交叉更新多个文档

7.6 总结

Summary

本章我们介绍了许多内容。各种不同的更新方式，这些操作展现出的强大功能都是可以保证的。事实上，MongoDB更新语言与查询语言一样复杂。我们可以像更新简单文档一样更新复杂的文档。当需要的时候，也可以单独原子地更新某个文档，与findAndModify一起使用，可以构建事务性工作流。

如果你已经学习完本章，感觉自己可以独立应用这些例子了，那么现在你就在成为MongoDB大师的路上了。