

## 第二部分 MongoDB 电商开发实战

---

# Application development in MongoDB

本书的第二部分会详细介绍MongoDB的文档数据模型、查询语言和CRUD（新增、读取、更新和删除）操作。

我们会通过逐步实现一个电商数据模型和实现必要的CRUD操作来说明这些问题。每章都会使用自上而下的方式来阐述主题，首先是通过例子介绍电商应用的领域，然后逐步丰富细节。在第一次阅读本书的时候，可能你想只看例子部分的内容，细节留着以后阅读。

第4章，我们会学习一些schema架构设计原则，然后为商品、类别、用户、订单和商品评价构建一个基本的电商数据模型。最后我们会学习MongoDB如何在数据库、集合和文档级别组织数据。本章包含BSON的核心数据类型。

第5章涵盖了MongoDB的查询语言。我们会学习如何基于上一章介绍的数据模型使用常见的查询。在要点小节里，我们会看到查询操作符语义的详细介绍。

第6章是关于聚合概念的内容。我们会介绍如何实现一些简单的分组，并且会深入介绍MongoDB的聚合框架。

在介绍MongoDB的更新和删除操作时，第7章通过介绍电子商务数据模型的设计原理来完善本部分的知识点。我们还会深入学习如何维护类别层级和如何管理仓库事务。此外，还详细介绍强大的更新操作符findAndModify命令。

( MongoDB中国学习交流群 二维码 )





# 面向文档的数据

## Document-oriented data

### 本章内容

- schema设计
- 电子商务数据模型
- 数据库、集合和文档

本章将会详细看一下面向文档的数据模型以及如何在MongoDB数据库、集合和文档级别里组织这些数据。我们将从一个简单的、通用的案例讨论开始，介绍如何使用MongoDB来设计schema。

记住，MongoDB本身并不强制使用schema，但是每个应用都需要一些如何存储数据的基本内部标准。本章的第二部分会探讨设计原则，检查MongoDB的电商网站schema设计。这样，我们将会看到RDBMS schema和MongoDB的不同，也会学习一些典型的实体关系，比如如何在MongoDB里体现一对多和多对多关系。电商网站的schema展示的知识也是本章后续内容查询、聚合以及更新知识的基础。

因为文档是MongoDB的原材料，所以我们会在本章最后部分详细讲解大家在思考schema的时候可能遇到的问题。这里会更深入地讨论数据库集合和文档知识。如果阅读到结尾，你可能会明白很多特性以及MongoDB文档的限制。大家在阅读本章最后一节的时候可能会发现许多宝藏，因为这里包含了大家使用MongoDB开发系统过程中遇到的许多典型的问题及其答案。

## 4.1 schema 设计原则

### Principles of schema design

数据库schema设计是基于数据库特性、数据属性和应用系统选择最好的数据表示形式的过程。关系型数据库schema设计的原则已经建立了。对于RDBMS数据库，我们只需要遵守数据库设计范式<sup>[1]</sup>即可，它可以帮助我们确保通用查询以及数据一致性。此外，这些原则模式避免开发人员思考如何建模，比如一对多以及多对多的关系。但是schema设计即使对于数据库也并非是一门精确的科学。应用功能和性能是schema设计最重要的考虑因素，所以每个规则都有例外。

如果你来自RDBMS世界，可能会对于MongoDB缺少强制的schema设计原则感到困惑。优秀的实践原则已经出现，但是还有很多好方法可以处理数据集建模问题。本节的原则可以驱动schema的设计，但是现实中这些原则也是弹性的，大家可以灵活选择。

当使用数据库系统建模时，可以先思考以下问题：

- 应用访问的模式是什么？你需要分解需求，不仅仅是落实schema设计，还有选择采用什么数据库。记住，MongoDB并非适用于所有的应用。理解应用的访问模式是目前为止schema设计最重要的方面。

应用程序的特征很容易要求schema严格遵守数据建模的原则，导致你在决定理想的数据模型之前必须问许多问题：读/写的比率是多少？查询是不很简单？查询一个key还是更复杂的key？是否需要聚合查询？数据量是多少？

- 数据的基本单位是什么？在RDBMS里，我们有列和行的表。在键值数据库里，我们有键指向不同的值。在MongoDB里，数据的基本单位是BSON文档。
- 数据库的功能是什么？一旦明白基本的数据类型，就知道如何来操作它了。RDBMS功能ad hoc查询以及连接通常写入SQL，而简单的键值存储允许通过key获取数据。MongoDB也允许ad hoc查询，但是不支持join连接查询。

数据库更新数据的方式也不同。RDBMS允许使用SQL进行复杂的更新，可以在事务里包含多个更新并支持原子性和回滚。MongoDB不支持事务，但是它支持另外一个原子更新操作，可以更新复杂结构的文档数据。使用键值库，你可以更新一个值，但是每次更新都意味着完全替换一个值。

- 如何记录生成好的唯一ID或者主键？虽然也有例外，但是无论什么数据库系统，许多

---

<sup>[1]</sup>简单理解范式的方式是，信息不会存储2次以上。因此，实体的一对多关系通常会分割为2个表。

schema都有记录的唯一key。选择key的策略会影响如何访问和存储数据。如果你正在设计user集合，例如，使用任意值、名字、username或者社会安全号作为主键，那么结果证明在数据集里姓名和社会安全账号都不是唯一的。

MongoDB选择\_id字段里存储的值作为主键。这个自动生成的默认值不错，当然并非适用于所有的情况。你要在多个机器上分片存储数据时，这非常重要，因为它决定了数据文档存储在哪个地方。我们会在12章里详细讨论这个问题。

最好的schema设计通常是深入了解使用的数据库、了解应用系统的需求，以及具有丰富的经验之后的产物。好的schema通常需要试验和迭代，比如当应用伸缩时，或者性能考虑变化时。

当学习新知识的时候不要担心会修改schema，因为几乎不可能在实现代码之前把所有问题都了解清楚了。本章的例子用来帮助大家开发好的MongoDB schema设计。学习了这些例子后，我们就可以开始为我们的应用系统设计最好的schema了。

## 4.2 设计电商网站数据模型

### Designing an e-commerce data model

第3章提供的Twitter应用程序例子主要是演示了基本的MongoDB特性，但是不需要关注schema设计。我们在后续的章节里，会学习更多的电商平台领域知识。电子商务的优势是包含大量的、熟悉的数据建模模式，而且，很容易在RDBMS关系型数据库里进行products、categories、product reviews、orders等建模。这个可以让例子更容易理解，因为我们可以与之前先入为主的schema设计规则进行对比。

电商网站使用RDBMS数据库有几个原因。首先，电商网站通常需要事务，这是关系型数据库的强项。其次，需要复杂数据模型和复杂查询的领域最适合采用关系型数据库。

下面的例子可以说明第二个观点。

开发完整的电商后台并非本书的主要内容。相反，我们将会处理一些常见的、有用的电商实体，比如产品和客户评价，以及如何在MongoDB里建模。尤其是，我们将会看到一些产品、类别、用户、订单、商品评价。对于每个实体，我们将会展示例子文档。然后，将会展示一些数据库特性、完善文档的结构。

对于许多开发者来说，数据模型和对象映射一样重要，因此他们可能会使用一些ORM框架，必须是Java的Hibernate或者Ruby的ActiveRecord。这些框架可以和RDBMS关系型数据库高

效地结合，但是无法与MongoDB一起使用。这是因为文档本身已经是类对象表示形式了。其次是MongoDB驱动的原因，它已经提供了对MongoDB的高级别抽象了。毫无疑问，我们只使用驱动接口来构建MongoDB应用。

对象映射器可以通过验证、类型检查和模型关联来提供值，而且都有标准的框架，比如Ruby on Rails。对象映射器也在程序员和数据库之间引入了新的复杂性，隐藏了重要的查询特性。当使用对象映射器时，就应该评估一下它的利与弊。许多优秀的应用，有的使用了对象映射器，有的没有使用对象映射器<sup>[1]</sup>。我们在本书的例子中没有使用对象映射器，而且我们推荐第一次使用MongoDB的时候不要使用它。

## 4.2.1 schema 基础知识

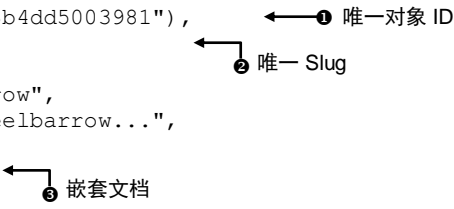
商品和类别是所有电商网站的主要部分。对商品，可以在RDBMS里进行范式模型建模，需要许多表，有基本的商品信息表，比如名字和SKU，而且还有其它表，比如快递信息和价格历史。多表schema可以使用RDBMS的多表关联。

MongoDB建模商品表时简单得多。因为集合不强制使用schema，任意文档都可以满足商品动态字段的需求。通过在文档里定义数组就可以在MongoDB集合里实现RDBMS的多表关联关系。

列表4.1展示了一个园艺店商品的具体文档例子。建议在使用`db.products.insert(yourVariable)`保存文档到数据库之前，把值传递给参数变量。

列表4.1 商品文档的例子

```
{
  _id: ObjectId("4c4b1476238d3b4dd5003981"),
  slug: "wheelbarrow-9092",
  sku: "9092",
  name: "Extra Large Wheelbarrow",
  description: "Heavy duty wheelbarrow...",
  details: {
    weight: 47,
    weight_units: "lbs",
    model_num: 4039283402,
    manufacturer: "Acme",
    color: "Green"
  },
  total_reviews: 4,
  average_review: 4.5,
  pricing: {
    retail: 589700,
```



<sup>[1]</sup>要找到合适的对象映射器，可以参考 [mongodb.org](http://mongodb.org) 网站。



```

    sale: 489700,
  },
  price_history: [
    {
      retail: 529700,
      sale: 429700,
      start: new Date(2010, 4, 1),
      end: new Date(2010, 4, 8)
    },
    {
      retail: 529700,
      sale: 529700,
      start: new Date(2010, 4, 9),
      end: new Date(2010, 4, 16)
    },
  ],
  primary_category: ObjectId("6a5b1476238d3b4dd5000048"),
  category_ids: [
    ObjectId("6a5b1476238d3b4dd5000048"),
    ObjectId("6a5b1476238d3b4dd5000049")
  ],
  main_cat_id: ObjectId("6a5b1476238d3b4dd5000048"),
  tags: ["tools", "gardening", "soil"],
}

```



文档包含了基本的name、sku、description字段，还有标准的MongoDB object ID字段①。我们会在下一节里讨论文档的其他方面的问题。

## 唯一的 URL Slug

此外，我们还定义了 URL slug ②，wheelbarrow-9092，提供一个有意义的URL结构，利于SEO。MongoDB用户有时候会抱怨URL里无意义的object ID值。

显然，我们不喜欢如下的这种URL结构：

`http://mygardensite.org/products/4c4b1476238d3b4dd5003981`

有意义的URL结构是更好的选择：

`http://mygardensite.org/products/wheelbarrow-9092`

这种用户友好的永久链接通常叫做slug。我们通常推荐为文档创建一个slug字段来构建有意义的URL。这种字段通常唯一索引，以加速查询和确保唯一。我们也可以在\_id 里存储slug，用做主键。这个例子我们就不选择它来演示唯一索引了，每个方式都可以接受。假设我们要在products集合里存储商品文档信息，就可以创建如下的唯一索引。

```
db.products.createIndex({slug: 1}, {unique: true})
```

如果slug创建唯一索引，则插入重复值会抛出异常。这时就可以尝试不同的slug值。假设花园

商店有多个独轮手推车在销售，那么当销售新的独轮车的时候，就需要创建一个新商品的唯一slug。

下面是执行插入操作的Ruby代码：

```
@products.insert_one({
  :name => "Extra Large Wheelbarrow",
  :sku => "9092",
  :slug => "wheelbarrow-9092"})
```

除非指定，否则驱动会自动确保不会抛出错误。如果插入成功就不会有错，你也可以知道自己插入的是唯一的slug。但是如果抛出异常，那么代码就需要尝试新的slug值。你可以在7.3.2节里查看捕获和优雅地处理异常的方式。

## 内嵌文档

假设有个key叫`details`❶，指向一个子文档，它包含商品的详细信息。这个key与`_id`字段不同，它允许我们在一个现有的文档里进行查询。我们可以指定重量、高度单位，还有制造商的型号。我们也可以存储其他的查询字段。例如，如果销售的是种子，可能会包含产量和收获时间；如果卖割草机，可能要包含马力、燃油和保养信息。`details`字段为这种动态属性数据提供了良好的扩展点。

我们也可以在同一个文档里存储商品的当前和过去的价格。`pricing`键指向的对象包含零售和批发价格。`price_history`相反，引用了一个完整的数组价格选项。存储的文档副本很像版本的控制技巧。

接下来是一个商品tag名字的数组。我们可以看到与第1章类似的例子。因为我们可以为数组key建立索引，这是最好和最简单的存储相关标签的方式而且同时可以保证高效的查询效率。

## 一对多关系

关系怎么处理？通常需要关联其他集合中存储的文档。从把商品和类别关联起来开始❷。首先要定义商品的类别，把商品区分开，假设有单独的类别集合。然后我们需要一个商品和类别的关系❸。这是一对多的关系，因为商品只有一个类别，但是类别可以有多个商品。

## 多对多关系

我们也想把商品加入它关联的类别里，而不是主要的类别里。这时，商品和类别的关系就是多对多关系。RDMBS中，我们可以使用交叉表来表示多对多的关系。交叉表把多对多的关系存

储在单个表里。使用SQL join语句就可以查询与商品相关的所有类别，反之亦然。

MongoDB 不支持join连接，所以需要不同的多对多策略。我们定义了一个字段叫做 `category_ids`❹，包含对象ID数组。每个对象ID都作为指向类别文档的指针。

## 关系结构

列表4.2展示了相同的类别文档。我们可以把它赋值给一个变量，然后调用 `db.categories.insert(newCategory)` 方法插入数据库。在以后的查询里可以继续使用而不需要再次输入。

列表4.2 类别文档。

```
{
  _id: ObjectId("6a5b1476238d3b4dd5000048"),
  slug: "gardening-tools",
  name: "Gardening Tools",
  description: "Gardening gadgets galore!",
  parent_id: ObjectId("55804822812cb336b78728f9"),
  ancestors: [
    {
      name: "Home",
      _id: ObjectId("558048f0812cb336b78728fa"),
      slug: "home"
    },
    {
      name: "Outdoors",
      _id: ObjectId("55804822812cb336b78728f9"),
      slug: "outdoors"
    }
  ]
}
```

如果回到商品文档，仔细看下 `category_ids` 里的对象ID，就会发现商品关联的是 Gardening Tools 类别。商品文档里的 `category_ids` 数组键允许我们查询多对多的关系。例如，查询 Gardening Tools 类别下的所有商品，代码如下：

```
db.products.find({category_ids: ObjectId('6a5b1476238d3b4dd5000048')})
```

要在商品目录里查询所有的类别，可以使用 `$in` 操作符：

```
db.categories.find({_id: {$in: product['category_ids']}})
```

之前的命令假设 `product` 变量已经提前定义了：

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

你会注意到类别文档里的 `_id`、`slug`、`name`、`description` 字段。非常简单，但是父文档

数组就没有这么简单。为什么要存储如此大量的父文档的类别冗余数据呢？

类别通常都是分级的，而且数据库有多重方式表示它。例如，假设“Home”是商品的类别，“Outdoors”是子类别，“Gardening Tools”又是其子类别。由于MongoDB不支持连接，因此我们违反范式来在每个子文档里存储父类别的名字，这意味着数据是重复存储的。如此，当查询“Gardening Products”类别时，就不需要执行额外的查询来获取父类Outdoors 和 Home 的名字和URL。

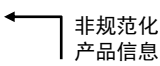
有些开发者可能感觉这种级别的去范式设计是无法接受的。但是此刻最好的schema设计就是根据实际的应用程序的设计要求来确定，而并非死背教条理论。当你在下一章里看到更多查询和更新此结构的例子时，许多概念会更加清晰。

## 4.2.2 用户和订单

如果你关注如何建模用户和订单表，就会看到另外一个公共关系：一对多。这时，每个用户可以有多个订单。在RDBMS数据库中，我们会在orders表中使用外键（这里约定习惯类似）。看看下面的代码列表4.3。

列表4.3 电商订单，包含商品、价格和快递地址

```
{
  _id: ObjectId("6a5b1476238d3b4dd5000048"),
  user_id: ObjectId("4c4b1476238d3b4dd5000001"),
  state: "CART",
  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
      }
    }
  ],
  shipping_address: {
```



非规范化  
产品信息

```

    street: "588 5th Street",
    city: "Brooklyn",
    state: "NY",
    zip: 11215
  },
  sub_total: 6196
}

```

非规范化销  
 售价总和

第二个订单字段`user_id`存储用户的`_id`。它可以高效地指向购买商品的用户信息，我们会在列表4.4里介绍。

```
db.orders.find({user_id: user['_id']})
```

查询特定订单的用户也非常简单：

```
db.users.findOne({_id: order['user_id']})
```

这种方式使用对象ID作为引用，就可以很方便地在用户和订单之间构建一对多关系。

## 基于文档思考

我们看一下订单对象的其他方面。通常，订单文档包含商品和传递地址。这些属性，在关系型数据库范式设计模型中分别存储在不同的表中。这里的商品信息使用子文档数组存储，每个子文档表示一个购物车里的商品信息。快递地址只指向一个地址字段的对象。

这种表示有几个优势。首先，是人类思想的胜利。一份完整订单包含商品、快递地址和最终的支付信息，可以封装到单个实体里。当查询数据库时，可以使用单个查询返回整个订单对象。而且，商品交易信息高效地存储在订单文档里。最后，正如将在接下来两章里看到的，你可以方便地执行查询和修改订单文档的操作。

用户文档（列表4.4）展示了相似的模式，因为它也存储了包含`payment_methods`的地址数组文档。正如商品上的`slug`字段，它智能地存储了`username`字段的唯一索引。

列表4.4 带有地址和支付方式的用户文档。

```

{
  _id: ObjectId("4c4b1476238d3b4dd5000001"),
  username: "kbanker",
  email: "kylebanker@gmail.com",
  first_name: "Kyle",
  last_name: "Banker",
  hashed_password: "bd1cfa194c3a603e7186780824b04419",
  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
    }
],
payment_methods: [
    {
        name: "VISA",
        payment_token: "43f6ba1dfda6b8106dc7"
    }
]
}

```

## 4.2.3 评价

我们使用商品评价来结束例子数据模型，如列表4.5所示。

每个商品可以有多个评价，而且我们可以通过在评价里存储`product_id`来实现一对多关系。

列表4.5 商品评价的文档

```

{
  _id: ObjectId("4c4b1476238d3b4dd5000041"),
  product_id: ObjectId("4c4b1476238d3b4dd5003981"),
  date: new Date(2010, 5, 7),
  title: "Amazing",
  text: "Has a squeaky wheel, but still a darn good wheelbarrow.",
  rating: 4,
  user_id: ObjectId("4c4b1476238d3b4dd5000042"),
  username: "dgreenthumb",
  helpful_votes: 3,
  voter_ids: [
    ObjectId("4c4b1476238d3b4dd5000033"),
    ObjectId("7a4f0376238d3b4dd5000003"),
    ObjectId("92c21476238d3b4dd5000032")
  ]
}

```

其余的字段基本是自我描述，易于理解。我们存储了评价的日期、标题和文本；用户提供的评分；用户ID。你可能会好奇为什么要存储`username`。如果使用RDBMS，就会使用`username`来关联`users`表。因为MongoDB不支持join连接，我们可以使用两种方式：根据`user`集合的每个评价进行查询或者接受去范式。根据每个评价进行查询也许没有必要，会导致不必要的成本，尤其是当`username`经常修改的时候。所以，我们这里选择优化查询而不是去范式。

另外值得一提的是，决定在评价文档里保存投票信息。对于用户来说可以选择支持某个评价。

这里我们选择在每个评价里保存评论用户的ID。这可以阻止用户多次投票，而且这可以帮助我们查询所有投票的用户。我们缓存了所有有帮助的投票数量，这可以让我们基于投票对于有帮助的评论进行排序。缓存非常有用，因为MongoDB不允许我们查询文档里数组的大小。通过投票来排序评价，例如，投票数组大小缓存在`helpful_votes`字段里，这是非常有帮助的。

至此，我们已经介绍了基本的电商数据模型。我们也已经看了基本的子文档、数组、一对多和多对多关系，以及如何使用去范式来作为工具简化查询。如果你是第一次看到MongoDB数据模型，理解这个模型则可能需要思想大转换。在接下来的几章里介绍这些问题——从添加唯一投票到修改订单，到智能查询商品。

## 4.3 核心概念：数据库、集合、文档

### Nuts and bolts: On databases, collections, and documents

我们现在从电商网站数据库例子里抽身出来，休息一会，来学习使用MongoDB数据库、集合和文档这些核心概念。这些知识涉及定义、专用特性、极端案例。如果你对于MongoDB分配文件的底层原理，如文档里严格允许的数据类型或者使用封顶集合的优势感兴趣，那就继续阅读下去。

### 4.3.1 数据库

数据库是集合和索引的命名空间和物理分组。本节里，我们将会讨论创建和删除数据库的细节，深入探讨MongoDB底层如何为单个的数据库分配空间。

#### 管理数据库

MongoDB没有显式的创建数据库的方式。相反，会在第一次写入数据的时候创建数据库。看看下面的Ruby代码：

```
connection = Mongo::Client.new( [ '127.0.0.1:27017' ], :database => 'garden' )
db = connection.database
```

回忆一下，当启动的时候，JavaScript shell执行这个连接，然后允许选择数据库：

```
use garden
```

假设数据库还不存在，在执行上面的代码后还没有创建在磁盘上。那么我们要做的就是创

建一个`Mongo::DB`的实例对象，表示一个MongoDB数据库。只有当我们向集合中写入数据的时候，才会创建数据库文件。继续Ruby代码，

```
products = db['products']
products.insert_one({:name => "Extra Large Wheelbarrow"})
```

当我们在集合上调用`insert_one`时，驱动会告诉MongoDB把商品信息插入`garden.products`集合。如果集合不存在，就创建它。这里还包括在磁盘上创建`garden`数据库。

通过调用下面的代码来删除集合中的所有数据：

```
products.find({}).delete_many
```

这个代码会删除所有匹配过滤器`{}`的文档，这就是集合中的所有文档。这个命令不会删除集合，它只会清空集合。要删除集合，就需要使用如下`drop`方法：

```
products.drop
```

要删除数据库，这意味着要丢弃所有的集合。我们可以通过专门的命令完成。可以使用下面的Ruby代码来删除`garden`数据库：

```
db.drop
```

在MongoDB shell里，使用JavaScript运行`dropDatabase()`方法：

```
use garden
db.dropDatabase();
```

删除数据库的时候要格外小心，因为没有办法回滚操作，它会从磁盘上删除数据库文件。

## 数据库文件和分配

当创建数据库的时候，MongoDB会在磁盘上分配一系列数据库文件集合，包括所有的集合、索引，还有其他元数据。数据库文件存储在启动`mongod`时`dbpath`参数指定的目录文件夹里。不指定参数时，`mongod`会在`/data/db`<sup>[1]</sup>文件夹里存储数据。我们来看看此文件夹下创建`garden`数据库后的样子：

```
$ cd /data/db
$ ls -lah
drwxr-xr-x 81 pbakkum admin 2.7K Jul 1 10:42 .
drwxr-xr-x 5 root admin 170B Sep 19 2012 ..
-rw----- 1 pbakkum admin 64M Jul 1 10:43 garden.0
```

---

<sup>[1]</sup>在Windows上，是`c:\data\db`。如果使用包管理器安装了MongoDB，则有可能在别的地方。例如，在OS X使用Homebrew把数据文件放置到`/usr/local/var/mongodb`里。



```
-rw----- 1 pbakkum admin 128M Jul 1 10:42 garden.1
-rw----- 1 pbakkum admin 16M Jul 1 10:43 garden.ns
-rwxr-xr-x 1 pbakkum admin 3B Jul 1 08:31 mongod.lock
```

这些文件取决于我们创建的数据库和数据库配置，可能在不同的机器上看起来不一样。首先注意mongod.lock文件，它可以存储服务器进程ID。不要删除或者修改锁文件，除非你正在从宕机事故中恢复数据。如果要启动mongod，就会得到锁文件的错误消息，导致“不干净关机”，而且我们可能必须启动恢复进程。我们会在第11章里深入讨论。

所有的数据库文件都前缀了属于自己的数据库名字。garden.ns是第一个生成的文件。文件扩展名是ns，表示命名空间。数据库中每个集合和索引的元数据有自己的命名空间文件，它的组织形式是哈希表。默认情况下.ns文件固定大小为16MB，它大约允许存储26000个数据。鉴于元数据的大小，这意味着数据库中索引和集合的数目的总和不能超过26000。实际上，不会有这么多索引和集合，但是如果真的需要比这个还多，则我们可以在启动mongod时通过-nssize修改。

处理创建命名空间文件，MongoDB还为集合和索引在文件里分配空间，从0开始以递增证书结束。仔细查看列出的目录，我们可以看到2个核心数据文件，即64 MB garden.0和128 MB garden.1。初始文件的大小可能会吓到新用户。但是MongoDB喜欢以这种预分配空间的方式来确保足够多的数据可以持续存储在文件中。这种方式下，当用户查询和更新数据时，这些操作会在临近的区域执行而不是跨磁盘进行。

随着持续添加数据到数据库，MongoDB会持续分配更多的数据文件。每个新数据文件会是前一个数据文件的两倍大小，直到最大的文件达到2GB的上限为止。从此，后续文件都是2GB。因此，garden.2是256 MB，garden.3是512 MB，以此类推。这里假设数据大小是按照一定的速度增长的，那么数据文件应该持续增长地分配，这是一个常见的分配策略。当然，其后果就是实际分配的空间比使用空间大得多<sup>[1]</sup>。

我们可以在JavaScript shell使用stats命令来查询使用的空间和分配的空间：

```
> db.stats()
{
  "db" : "garden",
  "collections" : 3,
  "objects" : 5,
  "avgObjSize" : 49.6,
  "dataSize" : 248,
  "storageSize" : 12288,
  "numExtents" : 3,
  "indexes" : 1,
```

---

<sup>[1]</sup>这在空间有限的环境里会带来问题。此时，应该使用-noprealloc和-smallfiles的组合配置。

```

    "indexSize" : 8176,
    "fileSize" : 201326592,
    "nsSizeMB" : 16,
    "dataFileVersion" : {
      "major" : 4,
      "minor" : 5
    },
    "ok" : 1
  }

```

在这个例子里，`fileSize`字段表示为此数据库分配的总文件空间。这里很简单就是garden数据库2个文件(garden).0 和 garden.1的总空间。`dataSize`和`storageSize`的差别很小。前者是数据库里实际BSON数据的大小，后者包括额外的为集合增长预留的空间，还有未删除的空间<sup>[1]</sup>。最后，`indexSize`值展示了数据库索引的总空间。

非常重要的是要注意总索引的大小；数据库性能只有在所有使用的索引都加载到内存里才是最好的。我们将会在第8章和第12章详细解释解决性能问题的技巧。

当我们部署MongoDB时这些都意味着什么？实际上，我们应该使用这些信息来帮助计划需要多少磁盘空间和RAM才能运行MongoDB。我们应该为预期的数据预留足够的磁盘空间，包括一些安全空间。磁盘空间通常很便宜，所以会分配比实际需要多得多的空间。

预估需要的RAM有点困难。我们喜欢有足够的RAM来满足工作数据集合在内存中的需要。工作数据集合 ( working set ) 是应用经常访问的数据。在电商网站例子中，可能要访问我们介绍的集合，比如经常在程序运行的时候访问商品和类别集合。这些集合，它们自己的开销和索引的大小应该装入内存，否则就会经常访问磁盘，性能就会受到影响。

这可能是最常见的MongoDB性能问题。也许有其他集合，我们只偶尔需要访问，例如audit审计集合，我们可以把它从工作集合中排除。通常，提前计划足够的内存为正常的应用程序操作集合所用。

## 4.3.2 集合

集合是结构或概念上相似的文档的容器。这里，我们将更加详细地介绍创建和删除集合，然后介绍MongoDB专门的盖子集合 ( capped collections，固定数目 )。我们看看例子，展示服务器内核是如何使用集合的。

---

<sup>[1]</sup>从技术上来说，集合是在每个数据文件中分配的，这个块叫做扩展(extents)。`storageSize` 是集合扩展分配空间的总空间。`capped collections` 不能翻译为“固定集合”，因为有歧义，是“固定数量”还是“固定容量”。所以取名“盖子集合”

## 管理集合

正如我们在前面一节看到的，创建集合是隐式的，只有插入文档时才会创建。但是因为有多重集合类型存在，所以MongoDB也提供了一个创建集合的命令。可以在JavaScript shell里使用：

```
db.createCollection("users")
```

创建标准集合时，可以通过参数指定预分配空间的字节大小。虽然这种做法通常没有必要，但还是可以在JavaScript shell中操作：

```
db.createCollection("users", {size: 20000})
```

集合名字可能包含数字、字母或者圆点字符，但是最好由字母或数字开头。从内部来说，集合名字是通过其命名空间名字来区分的，它包含所属的数据库名字。因此，商品集合技术上的指的是`garden.products`。完全限定的集合名字不能超过128个字符。

圆点符号有时候可以用作虚拟的命名空间。例如，可以使用下面的方式表示一系列集合：

```
products.categories  
products.images  
products.reviews
```

记住，这只是个组织原则。数据库带原点的集合名字与其他集合是一样的。

集合也可以重命名。例如，可以在shell里使用`renameCollection`方法重命名`products`商品集合：

```
db.products.renameCollection("store_products")
```

## 盖子集合

除了目前为止创建的标准集合外，还可以创建盖子集合（capped collection，也指有数量上限的集合，好像盖了盖子一样）。盖子集合最初是为高性能日志场景设计的。它与标准的集合不同，因为有固定的文档数量。这意味着一旦盖子集合达到最大上限，后续的插入将会覆盖最先插入的文档数据。

当只有最近的数据有价值时，这个设计也避免了用户手动修改集合数据。

要理解如何使用盖子集合，先要假设我们要记录网站用户的行为。这些行为包括查看商品、添加购物车、结账和交易。我们可以编写脚本来模拟日志用户行为到盖子集合里。在这个过程中，我们将会看到一些有意思的集合属性。列表4.6就是例子代码。

列表4.6 模拟日志用户行为到盖子集合里

```
require 'mongo'

VIEW_PRODUCT = 0 # action type constants
ADD_TO_CART = 1
CHECKOUT = 2
PURCHASE = 3

client = Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'garden')
client[:user_actions].drop
actions = client[:user_actions, :capped => true, :size => 16384]
actions.create

500.times do |n| # loop 500 times, using n as the iterator
  doc = {
    :username => "kbanker",
    :action_code => rand(4), # random value between 0 and 3, inclusive
    :time => Time.now.utc,
    :n => n
  }
  actions.insert_one(doc)
end
```

① 类型

garden.user\_action 集合

例子文档

首先，我们创建一个16KB的集合叫做`user_actions`，使用的是`client`创建<sup>[1]</sup>。接下来，我们插入500个日志文档①。每个文档包含一个`username`、行为代码（0到3的随机数代表）、时间戳，还包括一个自增的整数`n`，这样我们就可以区分哪个文档过时了。现在我们可以从`shell`里查询集合了：

```
> use garden
> db.user_actions.count();
160
```

虽然我们插入了500个文档，但只有160个文档保存在集合中<sup>[2]</sup>。如果查询集合，就会看到原因了：

```
db.user_actions.find().pretty();
{
  "_id" : ObjectId("51d1c69878b10e1a0e000040"),
  "username" : "kbanker",
  "action_code" : 3,
  "time" : ISODate("2013-07-01T18:12:40.443Z"),
  "n" : 340
}
{
  "_id" : ObjectId("51d1c69878b10e1a0e000041"),
  "username" : "kbanker",
  "action_code" : 2,
  "time" : ISODate("2013-07-01T18:12:40.444Z"),
  "n" : 341
}
```

<sup>[1]</sup>等价的创建命令可以从 `shell` 里使用 `db.createCollection("user_actions",{capped: true, size: 16384})`。

<sup>[2]</sup>数字可能不同，这和 `MongoDB` 版本有关系；值得注意的是部分就是少于这个数字的文档被插入集合中了。

```

}
{
  "_id" : ObjectId("51dlc69878b10e1a0e000042"),
  "username" : "kbanker",
  "action_code" : 2,
  "time" : ISODate("2013-07-01T18:12:40.445Z"),
  "n" : 342
}
:

```

文档按照插入的顺序返回。查看n的值可知，集合中最老的值是n等于340，这意味着从0~339已经老化了。因为这个盖子集合最大是16 384 kB，只能包含160个文档，每个文档大概102B。我们会在下一节里确认这个假设。尝试去为文档添加一个字段以增加平均的文档大小，从而减少总的文档数量。

除了大小限制以外，MongoDB允许为盖子集合指定最大文档数量的max参数。这允许对存储文档的总数进行细粒度控制。记住，大小配置具备优先权。创建集合的方式如下所示：

```

> db.createCollection("users.actions",
  {capped: true, size: 16384, max: 100})

```

盖子集合不允许正常集合的所有操作。例如，不允许从盖子集合里删除单个文档，而且也不能增加文档的大小。盖子集合最初是为日志设计的，所以没有必要实现删除和更新文档操作。

## TTL 集合

MongoDB也允许在特定的时间后废弃文档数据，有时候叫做生存时间time-to-live (TTL)集合，这个功能实际上是通过一个特殊的索引实现的。创建TTL索引的方式如下：

```

> db.reviews.createIndex({time_field: 1}, {expireAfterSeconds: 3600})

```

这个命令会在 ( time\_field ) 字段上创建索引。这个字段会定期检查时间戳，与当前的时间值比较。如果time\_field与当前时间值的差距大于expireAfterSeconds设置，文档就会自动被删除。这个例子里，评价文档会在一个小时后删除。

使用TTL索引，假设我们已经在time\_field里存储了时间戳。

以下是存储时间戳的例子代码：

```

> db.reviews.insert({
  time_field: new Date(),
  ...
})

```

这个代码在插入时为time\_field设置时间值。我们也可以插入其他的时间值，比如将来的

时刻。记住，TTL索引只会比较索引值和当前值的差别，比较当前的时间间隔与`expireAfterSeconds`值的大小。因此，如果我们在字段里设置将来的值，那么它只会等到未来的某个时刻超过了设定的`expireAfterSeconds`的长度。这个功能可以管理文档的生命周期。

TTL索引还有几个限制。我们不能在`_id`字段建立TTL索引，或者在其他已经建立索引的字段建立TTL索引。我们也不能在盖子集合里使用TTL索引，因为它不支持删除单个文档。最后，虽然我们可以在索引字段里有一组时间戳，但也不能组合TTL索引。这种情况下，TTL属性只会使用集合中最早的时间戳。

实际上，你可能发现自己不会要用到TTL集合，其实它在某些常见下是非常有用的，所以最好记住它。

## 系统集合

MongoDB的部分设计依赖于内部集合的使用。有两个特殊的集合：`system.namespaces` 和 `system.indexes`。我们可以在前者里查询当前数据库定义的所有命名空间：

```
> db.system.namespaces.find();
{ "name" : "garden.system.indexes" }
{ "name" : "garden.products.$_id_" }
{ "name" : "garden.products" }
{ "name" : "garden.user_actions.$_id_" }
{ "name" : "garden.user_actions", "options" : { "create" : "user_actions",
" capped" : true, "size" : 1024 } }
```

`system.indexes`存储了当前数据库里定义的所有索引。为了查看当前`garden`数据库定义的索引，可以插入这个集合：

```
> db.system.indexes.find();
{ "v" : 1, "key" : { "_id" : 1 }, "ns" : "garden.products", "name" : "_id_" }
{ "v" : 1, "key" : { "_id" : 1 }, "ns" : "garden.user_actions", "name" :
"_id_" }
{ "v" : 1, "key" : { "time_field" : 1 }, "name" : "time_field_1", "ns" :
"garden.reviews", "expireAfterSeconds" : 3600 }
```

`system.namespaces` 和 `system.indexes` 都是标准的集合，调试的时候非常有用。MongoDB为了复制功能也使用了盖子集合，复制功能可以保证多个`mongod`数据库之间的数据同步。每个主从复制集在特殊的盖子集合`oplog.rs`里记录了所有的写操作日志。从节点从这个集合里依次读取数据，应用到自己的数据库中。我们会在第10章里详细讨论这个问题。

### 4.3.3 文档和插入

我们将在本章里详细介绍文档和插入的细节内容。

#### 文档序列化、类型和限制

所有的文档在发送给MongoDB之前都序列化为BSON格式，以后再从BSON反序列化。驱动程序会处理底层的数据类型转换工作。绝大部分驱动都提供了从BSON序列化的简单接口，当读/写文档的时候会自动完成，我们并不需要担心这个过程。这里的例子仅仅是用于演示。

在之前的盖子集合里，可以假设文档大约是102B。我们也可以通过Ruby 驱动的BSON serializer序列化器来验证这个假设：

```
doc = {
  :_id => BSON::ObjectId.new,
  :username => "kbanker",
  :action_code => rand(5),
  :time => Time.now.utc,
  :n => 1
}
bson = doc.to_bson
puts "Document #{doc.inspect} takes up #{bson.length} bytes as BSON"
```

这个序列化方法会返回一个字节数组。如果运行这个代码，就会获取一个82B的对象，它和预估的差距不大。82B和102B的差别在于正常的集合和文档的其他开销。MongoDB为集合分配了特定的空间，一定要用于存储元数据。此外，在正常的集合（非盖子集合）里，更新文档可能导致空间增长，需要移动新的地址，并且留出空白区<sup>[1]</sup>。这个特性导致数据大小和MongoDB占用的磁盘空间之间有差别。

使用StringIO帮助类反序列化BSON非常简单。尝试运行Ruby代码来验证它的工作：

```
string_io = StringIO.new(bson)
deserialized_doc = String.from_bson(string_io)
puts "Here's our document deserialized from BSON:"
puts deserialized_doc.inspect
```

注意，不能序列化任意哈希数据结构。要做到无错序列化，key名字就必须有效，而且每个值必须可以转换为BSON类型。有效的key名字由最大255B长度的字符串组成。字符串可以由任意合法的ASCII字符组成，但有3个例外：不能由\$开始、不能包含圆点、除了在最后以外不能包含null字节。当使用Ruby编程时，可以使用符号作为哈希key，但是它们也会在序列化时转

---

<sup>[1]</sup>更多细节可以查看配置指令的填充因子。填充因子确保为文档增长预留一定的空间。填充因子从1开始，所以，第一次插入时没有额外分配空间。

换为等价的字符串。

这看起来有些奇怪，其实是因为key名也会存储在文档里，会影响数据的大小。这与RDBMS关系型数据库相反，它们的列名与行名是分离的。当使用BSON时，如果使用dob代替date\_of\_birth，则每个文档就可以节约10B的空间。看起来不多，但一旦有10亿个文档，就可以节约10GB的空间。当然也不是说尽量使用不合理、短小的key名字，要看情况而定。但是如果有大量的数据，那么合适的key名字确实是可以节约空间的。

除了有效的key名字外，文档也必须包含可以序列化为BSON的值。我们可以在<http://bsonspec.org>查看BSON类型的表，带有例子和注释。我们这里只介绍一些重要的数据类型。

## 字符串

所有的字符串必须使用UTF-8编码。虽然UTF-8已经很快成为字符编码的标准，但是还有许多时候在使用旧的编码方式。用户在使用旧的编码导入数据到MongoDB时会遇到问题。解决办法通常是先转换为UTF-8，再进行插入，或者直接把文本存储为BSON二进制类型<sup>[1]</sup>。

## 数字

BSON指定了三种数据类型：double、int、long。这意味着BSON可以编码任意IEEE浮点值以及任意8B长度的有符号整数。在动态语言比如Ruby和Python里序列化整数时，驱动会自动确定是否编码为int或者long int。事实上，只有一种情况需要显示指定类型：当我们通过Javascript shell插入数据类型时。Javascript，不幸的是，只支持一种数据类型叫做Number，它等价于IEEE 754 Double。所以，如果希望从shell保存一种数据类型，作为整数形式，我们需要显示指定类型，使用NumberLong()或NumberInt()。看看以下例子：

```
db.numbers.save({n: 5});
db.numbers.save({n: NumberLong(5)});
```

我们已经为numbers集合保存了2个文档，虽然它们的值是相等的：第一个以double保存，第二个以long int保存。查询n为5的文档，会返回所有的数据：

```
> db.numbers.find({n: 5});
{ "_id" : ObjectId("4c581c98d5bbeb2365a838f9"), "n" : 5 }
{ "_id" : ObjectId("4c581c9bd5bbeb2365a838fa"), "n" : NumberLong( 5 ) }
```

我们可以看到第2个值标记为long int。另外一种方式是使用专门的\$type操作符查询BSON类

---

<sup>[1]</sup>如果你不熟悉字符编码，可以去阅读 Joel Spolsky 著名的文章(<http://mng.bz/LVO6>)。



型。每个BSON类型通过一个从1开始的整数标识。如果查看BSON的文档<http://bsonspec.org>，会看到double是类型1，而64b的整数类型是18。因此，可以通过类型来查询集合：

```
> db.numbers.find({n: {$type: 1}});
{ "_id" : ObjectId("4c581c98d5bbeb2365a838f9"), "n" : 5 }
> db.numbers.find({n: {$type: 18}});
{ "_id" : ObjectId("4c581c9bd5bbeb2365a838fa"), "n" : NumberLong( 5 ) }
```

这个验证了存储数据的差别。我们可能从来不会在生产环境里使用\$type操作符，但是正如这里看到的，它确实是很棒的调试工具。

BSON数值类型最常见的问题就是缺少小数位支持。这意味着如果我们计划在MongoDB里存储货币数据值，就需要使用整数类型，并且单位是分。

## 时间

BSON的实际类型用来存储临时值。从Unix纪元计时开始时间值使用64b整数的毫秒值表示，负数表示之前的时间<sup>[1]</sup>。

使用要注意一下几点。首先，如果要在JavaScript里创建日期，记住JavaScript的月份是从0开始的。这意味着new Date(2011, 5, 11)表示2011年6月11日。其次，如果使用Ruby驱动存储类似数据，BSON序列化器会期望一个UTC格式的Ruby Time。

因此，我们不能使用日期类型表示时间区间，因为BSON时间不能编码此时间数据。

## 虚拟类型

如果非要用特定的时区来存储时间，怎么办？基本的BSON类型不能满足需求时，虽然没有创建自定义BSON类型的方法，但可以使用不同的基元BSON类型在子文档里创建自定义虚拟类型。例如，如果要存储带有时区的时间，则可以使用如下的文档结构。Ruby代码如下：

```
{
  time_with_zone: {
    time: new Date(),
    zone: "EST"
  }
}
```

编写处理这种组合文档类型的应用代码其实不难，这也是实际项目中处理问题的方式。例如，用Mongo-Mapper、Ruby编写的MongoDB对象映射器允许我们通过定义to\_mongo和from\_mongo方法来处理自定义数据类型。

---

<sup>[1]</sup> UNIX 纪元从 1970 年 1 月 1 日午夜开始计算，UTC 时间。我们会在 3.2.1 节简要介绍这个概念。

## 文档的限制

在MongoDB 2.0和以后的版本里，BSON文档大小限制为16MB<sup>[1]</sup>。限制它的原因有2个。第一，阻止开发者创建无意义的数据模型。虽然差的数据模型仍然可能出现，但16MB的限制不鼓励使用超过这个限制的文档。如果要存储超过16MB的文档，就可以考虑把文档存储为更小的文档。或者考虑MongoDB文档是否是一个合适的存储库——或许它更适合管理小文件。

第二，16MB的限制与性能相关。在服务器段，查询大的文档时需要在发送给客户端之前把文档拷贝到缓存里。这个拷贝工作非常昂贵，特别是当客户端不需要整个文档时<sup>[2]</sup>。此外，一旦发送数据，这就是跨网络传输数据的工作，而且客户端驱动要反序列化它，成本很高，尤其是处理上吉字节的大文件数据时。

MongoDB文档的嵌套深度最大值限制是100。嵌套就是文档里包含新的文档。使用深嵌套文档——例如，如果要序列化一个树形结构到MongoDB文档中——其结果就是查询和访问非常困难，还可能会导致其他问题。这种类型的数据结构通常都通过递归函数访问，对深入嵌套的文档处理会导致堆栈溢出。

如果你遇到了文档大小和嵌套的限制，最好把它们分割开，修改数据模型，或者使用其他的集合进行存储。如果要存储大的二进制对象，比如图片或者视频，这就是完全不同的情况。请参考附录C对于大二进制对象的处理技术。

## 大量插入

只要验证问答有效，插入过程就十分简单。关于插入文档最有价值的细节，包括对象ID生成、如何在网络层实现以及检查异常，在第3章里都做了介绍。但是一个重要的特性——大量插入，仍然值得在这里详细讨论。

所有的驱动几乎都可以一次插入多个文档。这对于插入大量数据非常方便，尤其是从另外一个数据库里导入大量数据时。这是一个简单的Ruby大量插入的例子：

```
docs = [ # define an array of documents
  { :username => 'kbanker' },
  { :username => 'pbakkum' },
  { :username => 'sverch' }
```

---

<sup>[1]</sup>不同版本的限制可能不同。要查询当前服务器版本的限制可以在当前 shell 里运行 `db.isMaster()`，查询 `maxBsonObjectSize` 值。如果没有找到这个字段，限制就是 4MB（使用的是很老版本的 MongoDB）。我们可以在这里看到更多的信息 <http://docs.mongodb.org/manual/reference/limits>。

<sup>[2]</sup>正如将会在下一章看到的，我们可以指定文档的某个字段在返回查询里来限制应答的大小。如果经常这么做，可以考虑重新评估数据模型。

```
]
@col = @db['test_bulk_insert']
@ids = @col.insert_many(docs) # pass the entire array to insert
puts "Here are the ids from the bulk insert: #{@ids.inspect}"
```

大量插入返回对象的ID数组，而不是单个ID。这是MongoDB驱动的标准。

大量插入对于性能非常有帮助。这意味着只需要一个连接就可以插入大量数据，而不需要大量独立地来回往返。这个方法有个限制，所以，如果要插入100万个文档，我们就必须把它们分割为多个大量插入的文档组<sup>[1]</sup>。

用户经常询问到底多大的插入是理想的，答案取决于许多因素。理想的数字可能从10到200。这里实践是检验真理的唯一标准。数据库设置的插入操作上限是16MB。经验证明，最有效的大量插入在这个限制下都可以正常工作。

## 4.4 总结

### Summary

本章介绍了许多深入的内容，祝贺你坚持学习到这个深度！

我们从schema设计的理论讨论开始，然后实战设计了电商网站的数据模型。这给了我们一个机会来看看生产环境下的文档是什么样子，这应该激发我们去思考传统关系型数据库RDBMS和MongoDB之间的schema更具体的差别。

本章结束部分我们又深入了解了数据库、文档和集合的知识；我们可能后面会继续复习，把本节作为参考知识。我们已经解释了MongoDB的雏形，但是还没有开始移动数据。下一章里会继续深入学习ad hoc查询的强大威力。

---

<sup>[1]</sup>批量插入的限制是 16MB。

