目　录

致谢

当前文档 《MongoDB入门指南》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-05-09。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常生活、工作和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址：http://www.bookstack.cn/books/Getting-Started-with-MongoDB

书栈官网：http://www.bookstack.cn

书栈开源：https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

README

Getting-Started-with-MongoDB

《MongoDB入门指南》是一个快速入门MongoDB的教程，它以MongoDB的3.0版本进行说明。本教程安装的是MongoDB Windows 64位版本，目的只是为了让读者快速的入门MongoDB，快速理解和操作MongoDB。在开发或生产中强烈要求使用Linux版本。

本教程是基础入门级别的，只包含MongoDB非常基础的CURD操作和基本概念，适合第一次接触MongoDB的人员阅读。

本教程不涉及MongoDB复制集，分片集群，分布式文件存储，监控与管理等内容。以上知识请参考MongoDB官方手册。

开始阅读

来源(书栈小编注)

> *https://github.com/jockchou/Getting-Started-with-MongoDB*

简介

MongoDB简介

MongoDB是一个开源的文档类型数据库，它具有高性能，高可用，可自动收缩的特性。MongoDB能够避免传统的ORM映射从而有助于开发。

文档

在MongoDB中，一行纪录就是一个文档，它是一个由键值对构成的数据结构，MongoDB文档与JSON对象类似。键的值可以包含其他的文档，数组，文档数组。

```
{
   "_id" : ObjectId("54c955492b7c8eb21818bd09"),
   "address" : {
      "street" : "2 Avenue",
      "zipcode" : "10075",
      "building" : "1480",
      "coord" : [ -73.9557413, 40.7720266 ],
   },
   "borough" : "Manhattan",
   "cuisine" : "Italian",
   "grades" : [
      {
         "date" : ISODate("2014-10-01T00:00:00Z"),
         "grade" : "A",
         "score" : 11
      },
      {
         "date" : ISODate("2014-01-16T00:00:00Z"),
         "grade" : "B",
         "score" : 17
      }
   ],
   "name" : "Vella",
   "restaurant_id" : "41704620"
}
```

集合

MongoDB在集合中存储文档。集合类似于关系数据库中的表。然而，与表不同的是集合不要求它里面的文档具有相同的结构。

在MongoDB中，存储在集合中的文档必然有一个唯一的_id字段作为主键。

安装

安装MongoDB

MongoDB能够运行在多种平台，并支持32位和64的构架。

由于本教程是在Windows上开展，所以只讲Windows上的安装。其他平台参考MongoDB官方手册。

在Windows上安装MongoDB

MongoDB2.2版本之前不支持Windows XP，本教程使用的版本是最新的3.0的版本。为了方便操作和理解，所以选择在Windows讲解，生产环境请使用Linux版本。

MongoDB支持32位和64的CPU构架。32位版本只是为了适应老的操作系统，用于开发学习，它有很多局限性，仅支持数据库少于2G的系统。64位版本还有一个Legacy版本，它不包括最近的性能优化，不建议使用。

在这里我们直接下载这个64位版本（MongoDB for Windows 64-bit）。安装过程非常简单，跟安装其他软件一样，一直下一步就行了。比如我的机器上安装到了 `C:\mongodb` ，在安装目录下面有一个 `bin` 目录。这个目录包含了MongoDB所有的命令和工具集合，把它配置到环境变量PATH中。如果你选择其他目录安装，请确保路径上没有空格，不然到时候会有很多坑。

设置MongoDB环境

MongoDB需要一个目录来保存数据，默认的数据目录是 `\data\db` 。在我的机器上使用下面的目录作为数据目录。

```
1. C:\data\mongo
```

你可以在启动MongoDB的时候为它指定一个数据目录。例如我们用如下命令启动MongoDB：

```
1. C:\mongodb\bin\mongod.exe --dbpath C:\data\mongo
```

数据目录不应该包含空格，否则要用 mongod.exe –dbpath “C:\data\mongo”。这些启动参数都是可以放到配置文件当中的，启动的时候指定配置文件。由于配置文件的参数比较多，我们这里暂时不需要理解那么多，先不使用。

```
1. mongod.exe --config /etc/mongod.conf
```

运行MongoDB

启动MongoDB，使用 `mongod.exe` 命令，例如：

```
1. C:\mongodb\bin\mongod.exe --dbpath C:\data\mongo
```

以上命令用来启动MongoDB服务主进程，并指定数据目录。执行完此命令后，在控制台会打印一系列的启动信息，包括MongoDB的版本，是否根据journal日志执行recovery，进程的信号，操作系统的信息等乖。最后一行会提示你启动成功，监听了27017端口，等待连接消息。

连接MongoDB

使用 `mongo.exe` 命令连接。打开另一个命令行窗口，输入如下命令：

```
1. C:\mongodb\bin\mongo.exe
```

执行些命令后，就能连接上MongoDB服务。由于没有配置任何其他端口，也没有配置权限认证，所以一切都是默认的本地连接，相当简单。连接成功后，执行help命令，看看有什么内容吧。

导入数据

导入数据

本教程使用test数据库和restaurants集合为例进行讲解。下面是restaurants的一个文档结构示例：

```
{
  "address": {
     "building": "1007",
     "coord": [ -73.856077, 40.848447 ],
     "street": "Morris Park Ave",
     "zipcode": "10462"
  },
  "borough": "Bronx",
  "cuisine": "Bakery",
  "grades": [
     { "date": { "$date": 1393804800000 }, "grade": "A", "score": 2 },
     { "date": { "$date": 1378857600000 }, "grade": "A", "score": 6 },
     { "date": { "$date": 1358985600000 }, "grade": "A", "score": 10 },
     { "date": { "$date": 1322006400000 }, "grade": "A", "score": 9 },
     { "date": { "$date": 1299715200000 }, "grade": "B", "score": 14 }
  ],
  "name": "Morris Park Bake Shop",
  "restaurant_id": "30075445"
}
```

导入例子数据

在进行操作之前，我们需要例子数据，在这里下载数据文件dataset.json。

导入数据到集合

在命令行中执行 `mongoimport` 命令将上面下载的数据文件中的数据导入到 `test` 数据库的 `restaurants` 集合中。如果此集合已经存在，下面的操作会先删除。

```
1. mongoimport --db test --collection restaurants --drop --file C:\data\dataset.json
```

`mongoimport` 命令连接到本机运行的 `mongod` 实例，如果要把数据导到不同主机，不同端口的实例，可以指定主机和端口，使用参数 `--host` 和 `--port` 。

数据导入后，你可以用 `mongo` 命令连接到实例，使用 `show dbs` ， `use test` ， `show collections` 和 `db.restaurants.find()` 命令查看导入的数据。

命令行

命令行

Mongo命令行是一个跟MongodDB服务交互的JavaScript接口工具，它是MongoDB封装的一个组件。你可以使用这个命令行工具查询，更新数据，执行一些管理操作。

运行命令行

安装并启动MongoDB后，就可以连接 `mongo` 命令行到MongoDB实例了。先确认MongoDB实例已经运行，然后才可以启动 `mongo` 命令行连接。

打开一个命令行窗口，执行如下命令即可：

```
1. mongo
```

请确认你已经配置了环境变量，在Windows上你也可以加上后缀.exe：

```
1. mongo.exe
```

如果没有配置环境变量，你要指定命令的全路径。

```
1. C:\mongodb\bin\mongo.exe
```

当运行 `mongo` 命令，不指定任何参数的时候，它默认是连接到本机localhost的27017端口。详细的参数参考MongoDB Shell

Mongo命令行Help命令

在mongo命令行中输出 `help` 将会列出所有可用的命令以及描述。

```
1. > help
```

mongo命令行还提供了跟Linux一样的自动完成和提示功能。并且可以使用上下箭头翻动历史命令。

插入数据

使用mongo命令行插入数据

概述

在MongoDB中，你可以使用 `insert()` 方法插入一个文档到MongoDB集合中，如果此集合不存在，MongoDB会自动为你创建。

插入文档

先用mongo命令行连接到一个MongoDB实例，转到test数据库。

```
use test
```

插入一个文档到restaurants集中，如果restaurants集合不存在，这个操作会先创建一个restaurants集合。

```
db.restaurants.insert(
   {
      "address" : {
         "street" : "2 Avenue",
         "zipcode" : "10075",
         "building" : "1480",
         "coord" : [ -73.9557413, 40.7720266 ],
      },
      "borough" : "Manhattan",
      "cuisine" : "Italian",
      "grades" : [
         {
            "date" : ISODate("2014-10-01T00:00:00Z"),
            "grade" : "A",
            "score" : 11
         },
         {
            "date" : ISODate("2014-01-16T00:00:00Z"),
```

```
            "grade" : "B",
            "score" : 17
        }
    ],
    "name" : "Vella",
    "restaurant_id" : "41704620"
}
)
```

可以看到，命令行的执行，其实就是javascript函数的调用。函数调用后返回一个 `WriteResult` 对象，它包含操作的返回状态信息。

如果插入的文档不包含 `_id` 字段，mongo命令行会自动加上这个字段到文档中，并且这个字段的值是根据ObjectId生成。

查询数据

使用mongo命令行查询数据

概述

使用 `find()` 方法在MongoDB集合中查询数据。MongoDB所有的查询范围都是单个集合的。也就是说MongoDB不能跨集合查询数据。

查询可以返回集合中的所有文档，或者仅仅返回指定过滤条件的文档。你可以指定一个过滤条件或才一个判断条件作为参数传递给 `find()` 方法。

`find()` 方法在一个游标中返回所有的结果集，通过游标的迭代可以输出所有文档。

查询集合中的所有文档

查询集合中的所有文档，直接调用集合的 `find()` 方法，不需要指定条件。如下命令查询 `restaurants` 中的所有文档。

```
1. db.restaurants.find()
```

返回的结果集包含 `restaurants` 所有的文档。

指定”等于”条件

查询条件如果是某个字段上的“等于”匹配的话，具有如下格式：

```
1. { <field1>: <value1>, <field2>: <value2>, ... }
```

如果是文档中的顶级字段，并不是内嵌的，也不是数组的话，你可以使用引号括住字段名，或者不使用引号。

如果就文内嵌字段，或者是数组，使用“.”号访问字段。而且必要使用相号括住整个字段名。

根据顶级字段查询

下面的命令查询所有 `borough` 字段值为”Manhattan”的文档。

```
1. db.restaurants.find( { "borough": "Manhattan" } )
```

查询的结果集中仅包含匹配的文档。

根据数组中的字段查询

在restaurants集合中，grades数组包含了内嵌文档作为它的元素。使用“.”号可以在内嵌文档中的某个字段上指定一个条件。同样，需要用引号括住有点号的引用。如下命令查询grades包括一个内嵌文档，它的grade字段的值为’B’的所有文档。

```
1. db.restaurants.find({"grades.grade": "B"})
```

指定操作条件查询

MongoDB提供了一些操作用来指定查询条件，比如比较操作。一些操作是除此之外的，比如 `$or` 和 `$and` 条件操作。使用操作的查询条件的格式如下：

```
1. { <field1>: { <operator1>: <value1> } }
```

大于操作($gt)

查询所有grades数组的内嵌文档中score字段的值大于30的文档。

```
1. db.restaurants.find( { "grades.score": { $gt: 30 } } )
```

小于操作($lt)

```
db.restaurants.find( { "grades.score": { $lt: 10 } } )
```

逻辑AND

你可以使用逻辑AND用于查询条件之间，使用逗号隔开。

```
db.restaurants.find( { "cuisine": "Italian", "address.zipcode": "10075" } )
```

逻辑OR

你可以为多个查询条件中使用逻辑OR，使用$or查询操作。

```
db.restaurants.find(
    { $or: [ { "cuisine": "Italian" }, { "address.zipcode": "10075" } ] }
)
```

当然，$and也可以使用上面的语法。

排序查询结果

指定查询结果排序方式的就是在查询后追加一个 `sort()` 方法调用。传递给此方法一个文档，包含指定排序字段和排序类型。1表示长充，-1表示降序。

```
db.restaurants.find().sort( { "borough": 1, "address.zipcode": 1 } )
```

如上命令，先按borough字段升序排列，再按address.zipcode升序排。

更新数据

使用mongo命令行更新数据

概述

使用 `update()` 方法更新文档。这个方法接收以下参数：

- 一个方档匹配的过滤器，用于过滤要更新的文档
- 一个用来执行修改操作的更新文档
- 一个可选的参数

指定过滤器和指定查询的时候是一样的。 `update()` 方法默认只更新单个文档，使用 `multi` 可选参数指定更新所有匹配的文档。

不能更新文档的 `_id` 字段。

更新指定字段

要改变某个字段的值，MongoDB提供了更新操作，比如 `$set` 用来修改值。如果字段不存在， `$set` 会创建这个字段。

更新顶级字段

下面的操作更新 `name` 字段值为”Juni”的第一个文档，使用 `$set` 操作更新cuisine字段，使用 `$currentDate` 操作更新lastModified字段。

```
1. db.restaurants.update(
2.     { "name" : "Juni" },
3.     {
4.       $set: { "cuisine": "American (New)" },
```

```
      $currentDate: { "lastModified": true }
    }
)
```

更新操作会返回一个 `WriteResult` 对象，它包含更新操作返回的一些状态信息。

更新内嵌文档字段

更新内嵌文档的字段，需要使用“.”号。如下所示：

```
db.restaurants.update(
  { "restaurant_id" : "41156888" },
  { $set: { "address.street": "East 31st Street" } }
)
```

更新多个文档

默认地， `update()` 方法只更新一个文档。如果要更新多个文档，需要指定 `multi` 可选参数。

```
db.restaurants.update(
  { "address.zipcode": "10016", cuisine: "Other" },
  {
    $set: { cuisine: "Category To Be Determined" },
    $currentDate: { "lastModified": true }
  },
  { multi: true}
)
```

替换文档

要替换一个文档，只需要把一个新的文档传递给 `update()` 的第二个参数，并且不需要包含 `_id` 字段。如果包含 `_id` 字段，只保证跟原文档是同一个值。用于替换的文档可以跟原文档具有完全不同的字段。

```
db.restaurants.update(
   { "restaurant_id" : "41704620" },
   {
     "name" : "Vella 2",
     "address" : {
              "coord" : [ -73.9557413, 40.7720266 ],
              "building" : "1480",
```

```
                "street" : "2 Avenue",
                "zipcode" : "10075"
      }
    }
)
```

删除数据

使用mongo命令行删除数据

概述

使用 `remove()` 方法从集合中删除文档。这个方法需要一个条件文档用来决定哪些文档将被删除。

删除匹配的所有文档

下面的操作将删除指定条件匹配的所有文件：

```
1. db.restaurants.remove( { "borough": "Manhattan" } )
```

删除操作返回一个 `WriteResult` 对象，它包含了操作的状态信息。 `nRemoved` 字段值表示被删除的文档数量。

使用justOne可选参数

默认地， `remove()` 方法将删除匹配指定条件的所有文档。使用justOne可选参数可以限制删除操作只删除一条。

```
1. db.restaurants.remove( { "borough": "Queens" }, { justOne: true } )
```

操作成功将返回如下的 `WriteResult` 对象。

```
1. WriteResult({ "nRemoved" : 1 })
```

`nRemoved` 字段值表示删除的文档数量。

删除所有文档

删除一个集合中的所有文档，传递一个空的条件文档即可。

```
1. db.restaurants.remove( { } )
```

删除一个集合

删除所有的操作仅仅是删除集合中的全部文档。集合本身和集合的索引并不会被删除。直接删除集合包括索引，也许比删除一个集合中的所有文档更高效。需要的时候重新创建集合并构建索引。使用 `drop()` 方法删除一个集合，包括所有索引。

```
1. db.restaurants.drop()
```

删除集合如果成功，此操作将返回true。如果被删除的集合不存在，将返回false。

在MongoDB中，”写”操作是文档级别的原子操作。如果一个删除操作要删除集合中的多个文档，这个操作会和其他写操作交错。具体请参考MongoDB手册中Atomicity。

数据聚集

使用mongo命令行进行数据聚合

概述

MongoDB可以执行数据聚合，比如按指定Key分组，计算总数，求不同分组的值。

使用 `aggregate()` 方法执行一个基于步骤的聚合操作（类似于Linux管道）。 `aggregate()` 接收一个步骤数组成为它的参数，每个步骤描述对数据处理的操作。

```
db.collection.aggregate( [ <stage1>, <stage2>, ... ] )
```

按字段分组并计算总数

使用$group管理操作符进行分组操作。在$group操作符中，使用 `_id` 来说明分组的key。$gropu管理操作使用$+字段名的方式来访问分组Key的。可以在每个分组管理操作中进行分组计算。下面的例子把restaurants集合按borough字段分组，并使用$sum操作符计算每个分组的文档数。

```
db.restaurants.aggregate(
    [
      { $group: { "_id": "$borough", "count": { $sum: 1 } } }
    ]
);
```

结果集包含以下文档：

```
{ "_id" : "Staten Island", "count" : 969 }
{ "_id" : "Brooklyn", "count" : 6086 }
{ "_id" : "Manhattan", "count" : 10259 }
{ "_id" : "Queens", "count" : 5656 }
{ "_id" : "Bronx", "count" : 2338 }
{ "_id" : "Missing", "count" : 51 }
```

`_id` 字段包含了不同的borough值，它也是分组参照的Key值。

过滤并分组文档

使用 `$match` 管道操作符过滤文档。 `$match` 使用的是MongoDB查询语法。下面的管道使用 `$macth` 查询borough字段值为"Queens"并且cuisine字段值为"Brazilian"的所有文档。然后 `$group` 分组管理操作符把匹配的所有文档按address.zipcode字段每组，并且使用 `$sum` 计算器计算总数。

```
db.restaurants.aggregate(
   [
     { $match: { "borough": "Queens", "cuisine": "Brazilian" } },
     { $group: { "_id": "$address.zipcode" , "count": { $sum: 1 } } }
   ]
);
```

结果集包含的文档如下：

```
{ "_id" : "11368", "count" : 1 }
{ "_id" : "11106", "count" : 3 }
{ "_id" : "11377", "count" : 1 }
{ "_id" : "11103", "count" : 1 }
{ "_id" : "11101", "count" : 2 }
```

`_id` 字段包含不同的zipcode的值。它是分组的Key。

Journaling日志

Journaling日志机制

运行MongoDB如果开启了journaling日志功能，MongoDB先在内存保存写操作，并记录journaling日志到磁盘，然后才会把数据改变刷入到磁盘上的数据文件。为了保证journal日志文件的一致性，写日志是一个原子操作。本文将讨论MongoDB中journaling日志的实现机制。

Journal日志文件

如果开启了journal日志功能，MongoDB会在数据目录下创建一个 `journal` 文件夹，用来存放预写重放日志。同时这个目录也会有一个 `last-sequence-number` 文件。如果MongoDB安全关闭的话，会自动删除此目录下的所有文件，如果是崩溃导致的关闭，不会删除日志文件。在MongoDB进程重启的过程中，journal日志文件用于自动修复数据到一个一致性的状态。

journal日志文件是一种往文件尾不停追加内容的文件，它命名以 `j._` 开头，后面接一个数字（从0开始）作为序列号。如果文件超过1G大小，MongoDB会新建一个journal文件 `j._1` 。只要MongoDB把特定日志中的所有写操作刷入到磁盘数据文件，将会删除此日志文件。因为数据已经持久化，不再需要用它来重放恢复数据了。journal日志文件一般情况下只会生成两三个，除非你每秒有大量的写操作发生。

如果你需要的话，你可以使用 `storage.smallFiles` 参数来配置journal日志文件的大小。比如配置为 `128M` 。

Journaling机制的存储视图

Journaling功能用到了MongoDB存储层数据集内部的两个视图。

`shared` 视图保存数据修改操作，用于刷入到磁盘数据文件。 `shared` 视图是MongoDB中唯一访问磁盘数据文件的视图。 `mongod` 进程请求操作系统把磁盘数据文件映射到虚拟内存的 `shared` 视图。操作系统只是映射数据与内存关系，并不马上加载数据到内存。当查询需要的时候，才会加载数据到内存，即按需加载。

`private` 视图存储用于查询操作的数据。同时 `private` 视图也是MongoDB执行写操作的第一个地方。一旦journal日志提交完成，MongoDB会复制 `private` 视图中的改变到 `shared` 视图，再通过 `shared` 视图将数据刷入到磁盘数据文件。

`journal` 视图是一个用来保证新的写操作的磁盘视图。当MongoDB在 `private` 视图执行完写操作后，在数据刷入磁盘之前，会先记录 `journal` 日志。 `journal` 日志保证了持久性。如果 `mongod` 实例在数据刷入磁盘之前崩溃，重启过程中 `journal` 日志会重放并写入 `shared` 视图，最终刷入磁盘持久化。

Journaling如何纪录写操作

MongoDB采用 `group commits` 方式将写操作批量复制到 `journal` 日志文件中。 `group commits` 提交方式能够最小化journal日志机制对性能的影响。因此 `group commits` 方式在提交过程中必须阻塞所有写入。 `commitIntervalMs` 参数可以用于配置日志提交的频率，默认是100ms。

Journaling存储以下原始操作:

- 文档插入或更新
- 索引修改
- 命名空间文件元数据的修改
- 创建和者删除数据库或关联的数据文件

当发生写操作，MongoDB首先写入数据到内存中的 `private` 视图，然后批量复制写操作到 `journal` 日志文件。写个 `journal` 日志实体来用描述写操作改变数据文件的哪些字节。

MongoDB接下来执行 `journal` 的写操作到 `shared` 视图。此时， `shared` 视图与磁盘数据文件不一样。

默认每60s钟，MongoDB请求操作系统将 `shared` 视图刷入到磁盘。使数据文件更新到最新的写入状态。如果系统内存资源不足的时候，操作系统会选择以更高的频率刷入 `shared` 视图到磁盘。

MongoDB刷入数据文件完成后，会通知 `journal` 日志已经刷入。一旦 `journal` 日志文件只包含全部刷入的写操作，不再用于恢复，MongoDB会将它删除或者作为一个新的日志文件再次使用。

作为journaling机制的一部分，MongoDB会例行性请求操作系统重新将 `shared` 视图映射到 `private` 视图，为了节省物理内存。一旦发生重映射，操作系统能够识别到可以在 `private` 视图和 `shared` 视图共享的内存页映射。

小结

Journaling是MongoDB中非常重要的一项功能，类似于关系数据库中的事务日志。Journaling能够使MongoDB数据库由于意外故障后快速恢复。MongoDB2.0版本后默认开启了Journaling日志功能， `mongod` 实例每次启动时都会检查 `journal` 日志文件看是否需要恢复。由于提交 `journal` 日

志会产生写入阻塞，所以它对写入的操作有性能影响，但对于读没有影响。在生产环境中开启Journaling是很有必要的。

原子性和事务

MongoDB原子性和事务

在MongoDB中，写操作的原子性是在 `document` 级别上的，即使修改的是文档中的内嵌部分，写锁的级别也是 `document` 上。

当一个写操作要修改多个文档，每个文档的修改是原子性的。整个的写操作并不是原子性的，它可能和其他写操作产生交织。然而你可以使用 `$isolated` 隔离操作符来限制写操作，让它不与其他写操作交织。 不隔离性能更高，但是会产生数据的不确定性，隔离写操作，事务性更好。MongoDB把这个级别完全由用户控制。

隔离写操作

MongoDB使用 `$isolated` 操作符来隔离写操作。如果一个写操作要更新多个文档，它能防止其他进程与本次写操作交错。直到这个写操作完成，其他进程才能写。

但是， `$isolated` 算不上一个事务，如果在写的过程中发生错误，MongoDB并不会回滚已经写的数据。 `$isolated` 也不能在分片集群上工作。

特性:

- 隔离不支持分片集群
- 不支持“all-or-nothing”特性
- MongoDB2.2版本后 `$isolated` 被替换成 `$atomic`

类事务语法

MongoDB并不支持关系型数据库中的那种事务特性，为了性能着想，它把这个特性交给程序员去实现。这就是MongoDB官方所讲的Two Phase Commits两阶段提交。这个技术虽然在一定程度上能保证数据最终的一致性，但是应用程序还是可能会读到提交或者回滚过程中的中间数据。对于这个技术如果有兴趣可以读一读原文。

并发控制

并发控制允许多个应用层程序同时访问数据库，而不引起数据不一致或冲突。

MongoDB中提到两种技术来解决这个问题。第一种是唯一索引，第二种是叫 `Update if Current` 。

用唯一索引来防止多个进程重复插入或者更新导致的重复的值。
`Update if Current` 意思是说在更新数据的时候，在更新条件里给定一个期望的值（这个值是先查询出来的），用来防止在更新之前其他进程已经将此值更新。看一个例子：

```
var myDocument = db.products.findOne( { sku: "abc123" } );

if ( myDocument ) {
   var oldQuantity = myDocument.quantity;
   var oldReordered = myDocument.reordered;

   var results = db.products.update(
      {
        _id: myDocument._id,
        quantity: oldQuantity,
        reordered: oldReordered
      },
      {
        $inc: { quantity: 50 },
        $set: { reordered: true }
      }
   )

   if ( results.hasWriteError() ) {
      print( "unexpected error updating document: " + tojson(results) );
   }
   else if ( results.nMatched === 0 ) {
      print( "No matching document for " +
             "{ _id: "+ myDocument._id.toString() +
             ", quantity: " + oldQuantity +
             ", reordered: " + oldReordered
             + " } "
      );
   }
}
```

同样的，在findAndModify()函数中：

```
db.people.findAndModify({
    query: { name: "Andy" },
    sort: { rating: 1 },
    update: { $inc: { score: 1 } },
    upsert: true
})
```

如果有多个进程同时调用此函数，这些进程都完成了查询阶段，如果 `name` 字段上没有唯一索引，upsert阶段的操作，多个进程可能都会执行。导致写入重复的文档。