# 第1篇 基础篇

首先简单介绍了关系型数据库，从关系型数据库所遇到的问题讲到NoSQL的崛起，然后讲述了MongoDB的基础操作和技巧，包括MongoDB的特点。数据类型、扩展性、功能性、管理、使用场合等等。然后更深入的讲解了MongoDB的数据结构、文档、集合、MongoDB shell的使用。还包括文档的增加、修改及删除。

## 第1章 MongoDB 简介

MongoDB是当前NoSQL数据库产品中最热门的一种。它在许多场景下可用于代替传统的关系型数据库或者键/值存储方式，MongoDB使用C++开发。MongoDB的官方网站地址：http://www.MongoDB.org/,读者可以登录此网站获得更详细的信息。

### 1.1 关系型数据库介绍

关系型数据库是建立在关系模型基础上的数据库。关系模型是由关系数据结构、关系操作集合（关系操作是指对关系实施的各种操作，包括选择、投影、连接、并、交、差、增、删、改等，其特点是集合操作方式，也就是操作的对象和结果都是集合）、关系完整性约束（关系完整性包括：实体完整性、参照完整性、用户定义完整性）三部分组成。标准数据库查询语言SQL就是一种基于关系型数据的语言，这种语言执行对关系型数据库中的数据的检索和操作。

关系型数据库中的每个数据库包包含一个或者多个数据表。表是以行和列的形式组织起来的数据集合。

关系模型就是指二维表格模型，因而一个关系型数据库就是由二维表（二维表名就是关系名，表中的第一行通常称为属性名，表中的每一个元组合属性都是不可分的，且元组的次序是无关紧要的。简单来说就是一个只有上下左右，没有前后的平面。如Excel就是一个二维表）及其之间的联系组成的一个数据组织。当前最流行的关系型数据库有：Oracle、DB2、Microsoft SQL Server、Microsoft Access、MySQL等。

### 1.2 关系型数据库面临的问题

1.数据库并发负载高

2.海量数据存储和访问

3.数据库数据越来越大

4.事务管理的负担

5.关系型数据库读/写实时性的忽略

6.多表关联查询被弱化

### 1.3 NoSQL的崛起

关系型数据库在越来越多的应用场景下暴露了这么多难以克服的问题，为了解决这类问题，NoSQL数据库运用而生。

NoSQL是非关系型数据存储的广义定义，次概念在2009年初得到了广泛的认同，也产生了很多种类的NoSQL数据库。

|  |  |  |
| --- | --- | --- |
| 存储类型 | NoSQL产品 | 特性 |
| 文档式存储 | MongoDB | 文档存储一般用类似JSON格式存储，存储的内容是文档类型的，这样也就有机会对某些字段建立索引，实现关系型数据库的某些功能 |
| 列式存储 | Hbase  Cassandra  Hypertable | 按列存储数据，最大的特点是方便存储结构化和半结构化数据，方便做数据压缩，针对某一列或者某几列的查询有非常大的I/O优势 |
| 键值存储 | Redis  TokyoCabinet  Tokyo Tyrant  Flare | 可以通过键快速查询到值。一般来说，存储不管值的格式，照单全收 |
| 对象式存储 | db4o  Versant | 通过类似面向对象语言的语法操作数据库，通过对象的方式存取数据 |
| 图形式存储 | Neo4J  FlockDB | 图形关系的最佳存储方式。使用传统关系型数据库来解决性能低下，而且设计不方便 |
| XML式存储 | Berkeley DB XML  BaseX | 高效地存储XML数据，并支持XML的内部查询语法，如XQuery、Xpath |

相比关系型数据库而言，NoSQL的优势有以下几点：

1.NoSQL数据库种类繁多，他们都有一个共同的特点，就是去掉了关系型数据库的关系型特点。

数据库之间没有关系（面向集合存储，易于存储对象类型数据，存储不需要固定的表结构），这样就非常容易扩展。

2.NoSQL数据库具有非常好的读/写性能，特别是在大数据库量下，读/写性能同样表现得非常优秀。这是因为它没有关系性，数据库结构简单。

3.NoSQL数据库不需要提前为要存储的数据建立字段，一个集合中随时可以存入自定义的数据格式。而在关系型数据库中，增加和删除字段是一件非常麻烦的事情，特别是数据量大的表。而在NoSQL数据库中可以轻而易举地做到。

4.NoSQL数据库可以在不影响性能的情况下，非常方便地实现高可用的架构。例如：Cassandra、Hbase模型，可以复制模型也可以实现高性能。

5.NoSQL数据库的出现，解决了关系型数据库所面临的问题，从而，弥补了关系型数据库的不足。而且还在某些方面极大地节省了开发成本和维护成本。

NoSQL发展至今，出现了好几种菲关系型数据库，本书就以NoSQL中目前最热门的MongoDB为例来进行说明。

### 1.4 MongoDB 是如何解决这些问题的

MongoDB是NoSQL中的一种，当然也具有NoSQL的特性。数据库之间没有关系，所以MongoDB数据库之间是非常容易扩展。

MongoDB中每个数据包包含一个或者多个数集合。集合是以BSON(Binary Serialized Document Format)数据结构组织起来的文档对象集合。

MongoDB数据库没有关系性，数据结构非常简单，所以它的读/写性能非常高。

MongoDB数据库可以处理大量的数据，也可以运行在便宜的PC服务器集群上，PC集群扩充起来非常方便并且成本很低。

MongoDB数据结构非常简单，存储不需要固定的表结构，易于存储对象类型数据，因此就不需要进行表与表之间的关联查询了。

### 1.5 初识MongoDB

MongoDB是一种非常强大、灵活、可扩展性的数据存储方式。MongoDB功能非常丰富，容易上手、便于使用。从而让我们关注编程本身，不再为存储数据而烦恼。

#### 1.5.1 MongoDB的特点

MongoDB最大的特点就是有非常强大的查询语言，其语法与面向对象的查询语法非常相似，可以实现关系型数据库单表查询的大部分功能，也支持对数据建立索引。MongoDB数据库的特点如下：

* 面向集合存储，易于存储对象类型的数据，如一个字段中可以存入一个对象。
* 模式自由，一个集合中可以存储一个键值对的文档，也可以存储多个键值对的文档，还可以存储键不一样的文档。
* 支持动态查询，查询语句使用**JSON**形式作为参数，可以很方便地查询内嵌文档和对象数组。
* 支持完全索引，包含内部对象，可以对内嵌文档创建索引。
* 支持查询。
* 支持复制和故障恢复，**从节点**可以复制**主节点**的数据，主节点的所有对数据的操作都会同步到从节点。从节点的数据跟主节点的数据是完全一样的，以作备份。主节点发生故障之后，从节点可以升级为主节点，也可以通过从节点对故障主节点进行数据恢复。
* 使用高效地二进制数据存储，包括大型对象（如视频），可以将图片文件甚至视频转换成二进制的数据存储到数据库中。
* 自动处理碎片，以支持云计划层次的扩展，可以水平扩展数据库集群，动态添加片（服务器）。
* 支持Python、Ruby、Java、C、C#、Javascript、Perl及C++语言的驱动程序，社区中也提供了对Erlang及.NET等平台的驱动程序。
* 文件存储格式为：BSON(一种JSON 扩展)。
* 可以通过网络访问。

有很多MongoDB常用的客户端GUI管理工具，如:

MongoVUE、RockMongo、MongoHub,

下载路径如下：

<Http://www.mongovue.com/>

<Http://rackmongo.com/>

<Http://github.com/bububa/MongoHub>

#### 1.5.2 数据模型

MongoDB数据库是一个面向集合并模式自由的文档类型数据库。

1. 面向集合

面向集合是指数据被分组存在在数据集中，被称为一个集合（Collencton==table）。集合类似于关系型数据库的表（table）,不同于表的是，集合不需要定义任何模式，集合在存储文档，一个文档类似于关系型数据库的一条记录。

在MongoDB中，要操作一个表并不需要创建它，可以直接往集合中插入数据，如果集合不存在，它会自动创建这个集合。

1. 模式自由

模式自由就是对存储在MongoDB数据库中的数据，我们不需要知道它是什么结构（可以是任务的文档）。

1. 文档型

文档型存储的数据是键值对的集合，键是字符串，值可以是数据类型集合中的任意类型，包括数组和文档。我们把这个数据格式称为BSON（Binary Serialized dOcumentNotation）

#### 1.5.3 扩展性

MongoDB采用面向文档的数据模式，可以自动在多台服务器中直接分割数据。还可以平衡集群的数据和负载，自动重排文档。如果需要扩大容量，只需在集群中添加新的机器，然后数据库会自动出来剩下的事情。

#### 1.5.4 功能性

* 完整的索引支持：可以对文档创建索引，甚至还可以对内嵌文档以及数组创建索引。MongoDB的查询优化器会自动分析查询语句，然后生成一个高效的查询集合。
* 动态查询：MongoDB的查询指令使用JSON形式，使查询表达式变得非常丰富，还可以非常容易地查询文档中的内嵌对象和数组。
* 查询监视：MongoDB提供了很多性能监视工具，用于分析数据库操作的性能。
* 复制及自动故障转移：MongoDB数据库支持服务器之间的复制，**从节点**可以复制**主节点**的数据，主节点的所有对数据的操作都会同步到从节点。从节点的数据跟主节点的数据是完全一样的，以作备份。主节点发生故障之后，从节点可以升级为主节点，也可以通过从节点对故障主节点进行数据恢复。
* 高效地传统存储方式：支持二进制数据及大型对象（如视频），可以将图片文件甚至视频转换成二进制的数据存储到数据库中。
* 自动分片以支持云级别的伸缩性：，可以水平扩展数据库集群，动态添加片（服务器）。

注意：有些关系型数据库常见的功能MongoDB并不具备，如链接（Join）和复杂的多项事务。MongoDB考虑的主要是提高扩展性，而分布式系统上实现这两个功能也的确很难。

#### 1.5.6 便于管理

MongoDB启动服务器之后，我们只需要知道如何在集群中添加新的节点，添加新的节点之后，数据库会自动集成和分配新的节点，如果主机服务器死机，MongoDB会自动切换到备份的服务器上，并将备份服务器升级为活跃的服务器。在分布式环境下，让数据库的管理变得非常方便。

MongoDB的管理理念就是尽可能地让服务器自动配置，让用户能在需要的时候调整设置（不强制）。

#### 1.5.7 使用场合

* 网站数据
* 具有缓存层
* 大尺寸，低价值的数据
* 高伸缩性的场景
* 用于对象及JSON数据的存储

## 第2章 MongoDB基本原理与安装

MongoDB是一个面向集合的、模式自由的文档型数据库。功能非常丰富，它支持的数据结构非常自由，类似于JSON的BJSON格式，可以存储结构非常复杂的数据（如一个字段可以存储一个文档对象等）。MongoDB支持查询语言，其语言类似于面向对象的查询语言。可以完成类似于关系型数据库单表查询的大部分功能，而且还支持对数据建立索引，下面给大家介绍MongoDB的入门知识。

### 2.1 数据库结构

MongoDB中多个文档组成集合，同样多个集合组成一个数据库。一个MongoDB实例可以承载多个数据库，它们之间是完全独立的。每个数据库也有独立的权限限制，在磁盘上不同的数据库 放置在不同的文件中。一个应用的所有数据建议存储在在同一个数据库中。当同一个MongoDB服务器上存放多个应用或者用户数据时，可以使用多个数据库，每个应用对应一个数据库。

1. **数据库的命名**

数据库和集合一样也是通过名字来标识的。数据库的名可以使用满足以下添加任意字符串（UTF-8字符串）来命名：

* 数据库名不能是空字符（“”）。
* 数据库名不能有**’’ (空格).$/\和空格符(\0)。**
* 数据库名应全部小写。
* 数据库名最多**64KB。**

数据库名有这么多限制是因为数据库名最终会变成系统中的文件。MongoDB有些特殊作用的数据库，这些数据库介绍如下：

* **Admin**:从权限角度来看，这是一个“root”数据库。在admin数据库中添加用户会自动继承所有数据库的权限。一些特定的服务器端命令只能通过这个数据库来执行，如列出所有数据库、关闭服务器。
* **Local**:该数据库永远不会被复制，可以用来存储限于本地单台服务器的任意集合。
* **Config**:是MongoDB用于分片设置时，config数据库在内部使用，用于保存分片的相关信息。

1. **体系结构**

MongoDB可以运行于当今流行的每一个平台，是一个可移植的数据库（即所谓的跨平台特性）。MongoDB在不同的平台上略有差别，但是从整体架构看上去，还是一样的，如数据逻辑结构和数据的存储等。

MongoDB服务器启动以后，由实例和数据库组成，一般情况下，MongoDB服务包含有一个实例和多个与之对应的数据库。特殊情况下（如硬件投入成本有限或者特殊的应用要求），运行一个服务器上有多个实例和多个数据库。

MongoDB数据库由一系列的物理文件（如数据文件、日志文件等）的集合或与之对应的逻辑结构（如集合、文档）组成。

1. **数据逻辑结构**

逻辑结构关系对比

* 关系型数据库：

 MySQL数据库(database)、表(table)、记录(rows) 三个层次概念组成。

* 非关系型数据库：

MongoDB 数据库(database)、集合(collection)、文档对象(document)三个层次概念组成。

* MongoDB 里的集合对应于关系型数据库里的表，但是集合中没有列、

行和关系的概念，集合中只有文档，一个文档就相当与一条记录，这体现了模式自由的特点。





1. **数据存储结构**

* MySQL的数据存储结构：

MySQL的每个数据库存放在一个与数据库同名的文件夹中，MySQL

如果使用MyISAM存储引擎，数据库文件类型就包括.frm、.MYD、.MYI。

* MongoDB的数据存储结构：

 MongoDB的默认数据目录是/data/db，它负责存储所有的MongoDB

的数据文件。在MongoDB内部，每个数据库都包含一个.ns文件和一些

数据文件，而且这些数据文件会随着数据量的增加而变得越来越多。

所以如果系统中有一个叫做test的数库，那么构成test这个数据库的

文件就会由test.ns，test.0，test.1等等组成。

****

### 2.2 文档

文档中的“键”类似于关系型数据库中的“字段”，文档中的键是不可以随意定义的，其命名有一定的规则，键的命名规则如下：

* 空格符（\0）在键中表示键的结尾，所以在键中不能包含空格符。
* $ 和 . 有特殊的意义，通常来说就是被保留的，只能在特定的环境下使用，如果使用不当的话，驱动程序会提示。
* 对于下划线“\_”没有严格的要求（下划线“\_”开头的键是保留的）。

除了这些以为，键可以是任意字符串（UTF-8字符）。MongoDB文档中的键值不仅区分类型，也区分大小写。下面两个文档则是不同的：

{“age”: 8}

{“age”: ” 8”}

上述两个文档，一个值是整型，另外一个是字符串型，所以两个文档是不同的文档，一下两个文档也是不同的，一个键是”age”,而另外一个键是”Age”。

{“age”: 8}

{“Age”: 8}

MongoDB在同一个文档中的键是不能重复的，例如，下面这个文档是错误的：

{“age”:” 8 “, “age”:” “8”}

如果想数据库中插入一个文档键重复，则会出现异常。文档是MongoDB中数据的基本单元，也是MongoDB的核心概念。多个“键值对”有序地放置在一起便是文档。

很多情况下，文档都是比较复杂的，经常会包含多个“键值对”，如下文档：

{“name”:”wangjunfeng”,”age”:26,”sex”:”男”,”address”:

{“house\_number”:”602”,”address\_info”:”广东省，广州市，天河区，岑村”}

}

这样的好处就是文档键值对的形成，可以非常容易地增加或者减少字段，下面这样例子告诉我们MongoDB非常重要的概念，文档中的键值对是有序的，如果键/值位置不一样，所以代表的文档也是不一样的。下面两个文档的键/值的位置不同就表示这是两个文档数据：

{“name”:”wangjunfeng”,”age”:26}

{“age”:26,”name”:”wangjunfeng”}

文档中的数据不仅可以是字符串，还可以是其他数据类型，设置可以是文档（内嵌文档）。

### 2.3 集合

MongoDB的文档类类似于关系型数据库表中的行，集合如同关系型数据库中的表。集合中存放的就是文档，每个集合中有N个文档。

#### 2.3.1集合的无模式

集合中的文档各式各样，可以有不同个数键值对的文档，可以有不同键不同类型值的文档，等等，称之为集合的无模式。例如，下面3个文档可以存放在同一个文档中：

1 {“name”:”wangjunfeng”}

2 {“age”:”26}

3 {“name”:”yeyijian”,”age”:25}

上面的3个文档有值的类型不同，键不同，也有键值对个数不同。既然集合中可以放置任意文档，那么为什么还要使用多个集合呢？

* 因为集合可以放置任意文档，就把各式各样的文档混在一个集合中，那么开发中就要确保每次查询只返回需要的文档种类，或者让执行者才查询的应用程序来处理所有不同类型的文档。
* 在同一个集合中查询特定类型的文档速度上不划算，分开多个集合要快得多。
* 把同一类型的文档放在同一个集合中，数据也会更加集中。
* 索引是安装集合来定义的。把同类型的文档放在同一个集合中，可以让索引更加有效。因为在创建索引时，文档会有附加的结构（特殊是唯一索引时）

#### 2.3.2集合的命名

可以通过名字来识别集合。集合可TF以是满足下列条件的任意字符串（UTF-8字符串）来命名：

* 集合名不能使用空字符串””。
* 集合名不能含有空格符**（\0）**，因为空格符表示集合名的结尾。
* 集合名不能以**”system.”**开头，因为这是为系统集合保留的前缀，**system.namespaces**这个集合保存了所有数据集合的信息。
* 用户创建大家的集合不能含有保留字符**$**,这是因为某些系统生成的集合包含该字符，除非要访问这种系统创建的集合，否则不要集合名中出现**$**。

首先要创建一个 **personalinfo**集合。本书的以下例子都会以这个集合为例进行说明。创建**personalinfo**集合的代码如下：

db.createCollection(“personalinfo”)

组织集合一般是使用””字符分开并按照命名空间划分的子集合。例如，一个存储人员的信息应用可以包含这两个集合，分别是personalinfo.baseinfo 和personalinfo.otherinfo。这样使组织结构更加明了，也就是说personalinfo这个集合（这里就不需要了）及其子集合没有任何关系。子集合虽然没有特别之处，但是非常有用，很多MongoDB工具中都包含了子集合。

* GridFS是一种存储大文件的协议，使用子集合来存储元数据，这样就与内容块分开了（关于GridFS内容详见第5章）。
* MongoDB的Web控制台通过子集合的方式将数据组织在DBTOP部分。
* 绝大数驱动程序都提供**语法糖**（在计算机语言中添加某种语法，不但对语法功能没有影响，而且还更方便程序员的使用，这种语法便叫做糖衣语法，也叫做语法糖），为访问指定集合的子集合提供了方便。例如在数据库shell中，db,personalinfo代表personalinfo集合，db.personalinfo.baseinfo 代表personalinfo.baseinfo集合。

### 2.4 MongoDB数据类型

MongoDB的文档使用BSON（Binary JSON）来组织数据，BSON类似于JSON，JSON只是一种简单的表示数据的方式，只包含了6种数据类型（null、布尔、数字、字符串、数组及对象），不能完全满足复杂业务的需要，因此，BSON还提供日期、32位数字、64位数字等类型。以下对mongoDB的数据类型进行简要说明：

**1、 null**

null类型用于表示空值或不存在的字段

如：{“one”:null}

**2、 布尔类型**

布尔类型有两上值，’true’和’false’ 如：{“one”:true}

**3、 32位整数**

mongoDB的控制台使用JS引擎进行输入，而JS仅支持64位浮点数，所以32位整

数将会被自动转义。

**4、 64位整数**

64位整数与32位整数一样，在MongoDB控制台使用时，会转义成64位浮点数。

**5、 64位浮点数**

MongoDB控制台数字的默认类型。

如：{“one”:2.02} {“one”:10}

**6、字符串**

UTF-8字符串都可以表示为字符串类型的数据。

如：{“one”:”Hello World”}

**7、 符号**

在MongoDB控制台中不支持这种类型，将自动转义成

字符串。

**8、 ObjectId 类型**

对象id是文档中唯一的12位的ID

0|1|2|3|4|5|6|7|8|9|10|11

时间戳 |机器 | PID | 计数器

如：ObjectId("4eae239f63520362e051e7fd")

**9、 日期**

日期类型存储是从标准纪元开始的毫秒数，不存储时区

注意：使用的时候要加上new

如：{“one”:new Date()}

**10、正则表达式**

文档键值可以包含正则表达式，其正则表达式采用JS语法来表示。

如：{“one”:/ho/i}

**11、代码**

文档中可以包含JS代码

如：{“one”:function(){/\*………….\*/}}

**12、数组**

文档中键值可以表示为数组，在数组内还可以嵌套数组；

如：{“x”:[“a”,”b”,[“c”,”d”]]}

**13、内嵌文档**

文档可以包含别的文档，也可以作为值嵌入到父文档中。

如：{“x”:{“name”:”Tom”,”age”:20}}

**14、\_id键和ObjectId对象**

MongoDB中存储的文档中必须有一个“**\_id**”键，这个键的值可以是任意类型，也可以自定义，默认为**ObjectId**对象。

**ObjectId**使用12字节存储空间，每个字节存储两位十六进制数，是一个24位的

字符串。看起来这个么长很难处理，关键是要知道**ObjectId**是实际存储数据的

两倍长。

### 2.5 MongoDB的下载和安装

MongoDB安装、配置完后，必须先启动，然后才能使用。MongoDB一般作为网络服务来运行，客户端可以连接到该服务器并执行操作，运行mongod可执行文件启动该服务器。启动服务器之前必须先运行MongoDB数据库安装、配置。

MongoDB服务端可运行在Linux、Windows或OS X平台，支持32位和64位应用，默认端口为27017。推荐运行在64位平台，因为MongoDB

在32位模式运行时支持的最大文件尺寸为2GB。

MongoDB把数据存储在文件中（默认路径为：/data/db），为提高效率使用内存映射文件进行管理。

以上为随便摘的，其实就是非传统的非关系数据库，现在归到文档型数据库分类之中，注意32位操作系统支持的最大文件为2GB，所以做大文件海量储存的朋友要选择64位的系统安装。开始我们的下载安装之路吧。

#### 2.5.1[Mongodb在Windows下安装及配置](http://blog.csdn.net/liuzhoulong/article/details/6124566)

1.下载mongodb的windows版本，有32位和64位版本，根据系统情况下载,下载地址：<http://www.mongodb.org/downloads>

2.解压缩至E:/mongodb即可

3.创建数据库文件的存放位置，比如D:/mongodb/data/db。启动mongodb服务之前需要必须创建数据库文件的存放文件夹，否则命令不会自动创建，而且不能启动成功。默认文件夹路径为c:/data/db.使用系统默认文件夹路径时，启动服务无需加--dbpath 参数说明，但文件夹还要手工创建

4.打开cmd命令行，进入D:/mongodb/bin目录，输入如下的命令启动mongodb服务：

D:/mongodb/bin>mongod.exe --dbpath D:/mongodb/data/db

显示：

Sat Jan 08 18:49:34 MongoDB starting : pid=232 port=27017 dbpath=E:/mongodb/data  
32-bit

\*\* NOTE: when using MongoDB 32 bit, you are limited to about 2 gigabytes of data

\*\* see <http://blog.mongodb.org/post/137788967/32-bit-limitations>

Sat Jan 08 18:49:34 db version v1.6.5, pdfile version 4.5  
Sat Jan 08 18:49:34 git version: 0eb017e9b2828155a67c5612183337b89e12e291  
Sat Jan 08 18:49:34 sys info: windows (5, 1, 2600, 2, 'Service Pack 3') BOOST\_LI  
B\_VERSION=1\_35  
Sat Jan 08 18:49:34 [initandlisten] waiting for connections on port 27017  
Sat Jan 08 18:49:34 [websvr] web admin interface listening on port 28017

表示启动成功，最后两行说明的数据库端口和Web端口，默认分别是27017和28017，在浏览器中打开[http://localhost:28017](http://localhost:28017/)，可以看到其相关的一些信息。

可以通过添加参数--port的方式，来修改数据库端口：D:/mongodb/bin>mongod.exe --port 10001 --dbpath D:/mongodb/data/db

5.再打开一个cmd输入：D:/mongodb/bin>mongo，或者双击mongo.exe，即可进行mongodb的客户端命令操作了,测试下

>// the mongo shell is a javascript shell connected to the db  
> 3+3  
6  
> db  
test  
> // the first write will create the db:  
> db.foo.insert( { a : 1 } )  
> db.foo.find()  
{ \_id : ..., a : 1 }

6.这样每次启动MongoDB很不方便，我们可以像安装的MySQL一样，把它作为Windows服务，这样就方便多了。  
安装MongoDB的windows服务的方法为是在MongoDB安装目录下创建logs目录，然后在CMD命令行输入  
E:/mongodb/bin>mongod --logpath D:/mongodb/logs/mongodb.log --logappend

--dbpath D:/mongodb/data/db --directoryperdb --serviceName MongoDB --install

显示：

all output going to: D:/mongodb/logs/mongodb.log  
Creating service MongoDB.  
Service creation successful.  
Service can be started from the command line via 'net start "MongoDB"'.

表示服务创建成功。

该命令行指定了日志文件：/logs/MongoDB.log，日志是以追加的方式输出的；

数据文件目录：/data/db，并且参数--directoryperdb说明每个DB都会新建一个目录；

Windows服务的名称：MongoDB；

以上的三个参数都是可以根据自己的情况而定的。

最后是安装参数：--install，与之相对的是--remove

7，以后就可以在cmd下用命令net start MongoDB和net stop MongoDB来启动和停止MongoDB了，也可以在本地服务中看到http://hi.csdn.net/attachment/201101/8/0_1294486505WhHp.gif

通过界面来管理该服务。

#### 2.5.2[Mongodb在Linux下安装及配置](http://blog.csdn.net/liuzhoulong/article/details/6125118)

我这里选用centOS作为其运行载体。

1.下载mongodb的linux版本(注意32位和64位的区别)，下载地址：<http://www.mongodb.org/downloads>

2.将下载的mongodb安装文件mongodb-linux-i686-1.6.5.tgz放到/usr/local/下

3.解压tar -zxvf mongodb-linux-i686-1.6.5.tgz

重命名mv mongodb-linux-i686-1.6.5.tgz mongodb

4,创建数据库文件目录，默认为/data/db,我把数据库文件目录放在当前文件夹下，mkdir -p data/db,创建日志目录mkdir log

5.启动服务  
cd /usr/local/mongodb/bin

./mongod -dbpath=../data/db -logpath=../log/mongodb.log

6，启动客户端shell测试

[root@localhost bin]# ./mongo  
MongoDB shell version: 1.6.5  
connecting to: test  
> db.foo.save({a:1})  
> db.foo.find()  
{ "\_id" : ObjectId("4d292a457e289d5d90dc6f33"), "a" : 1 }

7.将mongoDB服务加入随机启动

vi /etc/rc.local

使用vi编辑器打开配置文件，并在其中加入下面一行代码

/usr/local/mongodb/bin/mongod -dbpath=/usr/local/mongodb/data/db --port 27017

-logpath=/usr/local/mongodb/log --logappend

8.连接mongoDB客户端，出现如下信息则启动成功  
./mongodbbin/mongo  
MongoDB shell version: 1.6.5  
connecting to: test  
>

9，脚本启动关闭mongodb

## start-mongod.sh

MONGODIR=/usr/local/mongodb

MONGOD=$MONGODIR/bin/mongod

MONGO=$MONGODIR/bin/mongo

DBDIR=$MONGODIR/data/db

LOGPATH=$MONGODIR/log/mongodb.log

# mongod will print its pid, so store it in out.tmp, then

# print it using awk to mongod.pid

$MONGOD --dbpath $DBDIR --fork --logpath $LOGPATH --logappend > out.tmp

awk 'NR == 2 { print $3 }' < out.tmp > mongod.pid

sleep 3

$MONGO < onstart.js

## in onstart.js

use admin;

db.runCommand("logRotate");

## stop-mongod.sh

kill -15 `cat $mongod.pid`

## 第3章 文档的增加、修改及删除

本章介绍MongoDB数据库的基本操作，操作数据库首先需要插入并保存文档到集合中；然后可以删除、修改集合中的文档；还会介绍数据库的响应、客户端请求和MongoDB数据库连接等功能。

### 3.1 常用命令

在插入数据之前，首先在MongoDB shell中通过db.personalinfo.find()命令查看personalinfo集合的数据，代码如下：

> db.personalinfo.find()

>

查询结果显示，personalinfo集合中没有数据。下面向personalinfo集合中如何插入人员信息数据代码如下:

>personalinfo1={“name”:“wangjunfeng”,“age”:26}

//定义第一个对象，赋值为一个文档

{“name”:“wangjunfeng”,“age”:26}

>personalinfo2={“name”:“yeyijian”,“age”:25}

//定义第二个对象，赋值为一个文档

{“name”:“yeyijian”,“age”:25}

> db.personalinfo.insert(personalinfo1) //将第一个文档插入到集合中

> db.personalinfo.insert(personalinf02) //将第二个文档插入到集合中

> db.personalinfo.find()

{“\_id”:ObjectId(4f661cdf618a714de9afce33),“name”:“wangjunfeng”,“age”:26}

{“\_id”:ObjectId(4f661cdf618a714de9afce34),“name”:“yeyijian”,“age”:5}

以上代码，首先使用Javascript创建两个文档对象，然后通过db.personalinfo.insert()命令将文档对象保存到集合中，最后使用db.personalinfo.find()查询的命令查看集合中的数据。结果显示两个文档对象已经成功地保存带集合中。

**注意：**

* 不需要预先创建一个集合，在第一次插入数据时会自动创建。
* 在文档中其实可以存储任何结构的数据，当然在实际应用存储的还是相同类型文档的集合。这个特性其实可以在应用中很灵活，不需要类似alter table语句来修改数据结构。
* 每次插入数据时集合中都会有一个ID,名为“\_id”

#### 3.1.1 插入的原理和作用

MongoDB在执行插入时，首先，会将插入的数据转换成**BSON**格式；然后，MongoDB数据库会对BSON进行解析，并且检查是否存在“\_id”键（\_id键在插入数据库时MongoDB会自动生成）。MongoDB还对文档的大小做了限制。1.8之前的版本，文档不能超过4MB；1.8之后的版本支持16MB。除了这些之外，没有别的数据校验，就是简单的将文档原样存入数据库中。这样做的坏处是允许插入无限数据；好处就是不会受到注入式攻击，让数据库更加安全。

有很多编程语言是驱动程序在传送数据之前会进行一些数据的有效性校验（如文档是否过长是否包含非UTF-8字符，或者是否使用可未知类型）。如果对驱动的检查不是很清楚，可以启动MongoDB数据库服务器时使用“——objcheckxu”选项，这样启动服务器就会在插入之前校验文档结构的有效性，不过这样会牺牲一些性能。

**注意：**

插入超过4MB(或者1.8版本的16MB)的文档转换成BSON格式之后文档不能存入到数据库。需要查看转换为BSON的大小（以字节为单位），在shell中使用0bject.bsonsize(doc)。

MongoDB在插入时不会执行代码，所以传统的注入式攻击对MongoDB来说是无效的。

#### 3.1.2 批量插入

如果需要插入多个文档，可以使用批量插入。批量插入能传入一个由文档组成的数组给数据库。单个文档保存到数据库的时候，会有一个消息头，由这个消息头来告诉数据库此文档要插入到哪个集合当中去。而批量插入，一次发送数十，数百乃至数千个文档，一次只有一个TCP请求，只有一个消息头，无限一遍一遍地处理每个文档的这种消息，使得插入速度明显提高。

批量插入用于应用程序中，只能插入多个文档到一个集合，而不能用批量插入到多个集合执行操作。

如果只是导入数据，可以使用命令行工具，如“**mongoimport**”。但是如果对导入的数据要做一些小的修整（如转换日期称为日期类型，或者添加自定义的\_id）,则就需要使用批量插入，所以对批量插入对导入数据来说是非常有用的。

### 3.2 删除数据

在删除文档之前，首先通过db.personalinfo.find()命令查看personalinfo集合中有哪些数据，实际代码操作如下：

> db.personalinfo.find()

{“\_id”:ObjectId(4f661cdf618a714de9afce33),“name”:“wangjunfeng”,“age”:26}

{“\_id”:ObjectId(4f661cdf618a714de9afce34),“name”:“yeyijian”,“age”:25}

一共查询到两条数据，下面删除name是yeyijian的记录，实际代码操作如下：

>db.personalinfo.remover({“name”:”yeyijian”}) //删除name是yeyijian的记录

> db.personalinfo.find()

{“\_id”:ObjectId(4f661cdf618a714de9afce33),“name”:

“wangjunfeng”,“age”:26}

**注意：**

Remover()函数可以接收一个查询文档作为可选参数。传入参数之后，只有符合条件的文档才被删除。删除时，如果remover()函数中不传入任何参数，则会删除集合中的所有文档，但不会删除集合本身，原有的索引也会保留。删除数据是永久性的，是不能做撤销，也不能做恢复。

如果需要删除集合中的所有文档，可以直接删除集合（然后重新创建集合，重新创建索引）**drop（）**，速度比使用remover()函数快。但是drop()删除不能有任何条件的限制，整个集合被删除了，而且所有的索引也会被删除。

### 3.3 修改文档

文档存入数据库以后，如果需要对数据进行修改，那么就可以使用update()函数来进行修改它。

Update()函数有两个参数：第一个参数为限定条件，用来查询文档，通过这个条件来找出需要修改的文档。第二个参数为修改器（**Modifier**）文档，用于描述对找到的文档做了哪些修改。

修改操作：属于原子性操作（要么完整的被进行，要么完全不被执行）。如果两个更新同时发生，则先到达服务器的先执行，所以又冲突的更新操作并不会相互干扰。

#### 3.3.1 整个文档替换

最简单的方法就是用一个全新的文档来代替这个文档，这种方法适用于文档模式结构发生了较大的变化。下面是一个代码操作实例：

> db.users.find()

{ "\_id" : 1, "name" : "mongo" }

{ "\_id" : ObjectId("4eb2a199bf10550b2177b6f8"), "age" : 24 }

> db.users.update({"\_id":1},{"name":"nosql"})

> db.users.find()

{ "\_id" : 1, "name" : "nosql" }

{ "\_id" : ObjectId("4eb2a199bf10550b2177b6f8"), "age" : 24 }

以上代码，首先使用db.users.fin()查询的命令查看集合的数据，然后使用db.users.update()函数对修改的的数据进行修改。

#### 3.3.2 使用修改器

通常情况下只会对文档的部分内容进行修改。部分更新可以使用原子的更新修改器，更新修改器是一种特殊的键，用来指定复杂的更新操作（如调整、添加或者删除键），还可以操作数组或者内嵌文档。

**1.“$set”修改器**

“$set”修改器是用来修改指定一个键的值。如果键值不存在，则会自动创建它。这样就可以非常方便添加自己的自定义键。

db.user.update({“\_id”:ObjectId(“..”)},

…{“$set”:{name:”user1”}})

**2. “$inc”修改器**

“$inc”修改器是用来增加已有键的值，或者键不存在时自动创建。对于分析数据、托票、网站访问量统计或者其他有变化数据的地方，使用这个会非常方便。

“$inc”修改器只能操作键的值为数字。

db.user.update({name:”user1”},

…{$inc:{age:1}})

**3.数组修改器：**

1）.$push修改器 $pushAll

向数组末尾加入一个元素，如果没有就创建一个新的数组

db.user.update({name:”user1”},{$push:{hobby:” football”}})

2）.$addToSet修改器

判断一个值不在数组中就插入，存在的话不执行插入

db.user.update({name:”user1”},{$addToSet:{hobby:”sing”}})

相当于：

db.user.update({name:”user1”,hobby:{$ne:”sing”}},

…{$push:{hobby:”sing”}})

3).将$addToSet和$each组合起来，可以添加多个不同的值

db.user.update({“\_id”:ObjectId(”…”)},{$addToSet:{email:

…{$each:[“asee@163.com”,”asee@lampbrother.net”]}}})

这种情况下，使用$ne和$push是无法实现的。

4).$pop修改器

可以从数组的任何一端删除元素

$pop:{key:1}从数组末尾删除一个元素

$pop:{key:-1}从数组开头删除一个元素

5).$pull修改器 $pullAll

基于特定条件删除元素

$pull:{hobby:”football”}

6.数组的定位修改器$

若数组有多个值，我们可以通过下标直接作为键来选择元素

users:[{name:”user1”,},{name:”user2”},{name:”user3”}]

db.c1.update({title:”asee”},{$set:{users.0.name:”user4”}})

这样通过下标取很不方便。

db.c1.update({users.name:”user1”},

…{$set:{users.$.name:”user4”}})

#### upsert和save更新

**1.Upsert更新**

Upsert是一种特殊的更新，要是没有文档符合更新条件，就创建一个新文档，如果找到了匹配的文档，则正常更新。

db.user.update({name:”user1”},{name:”user2”},true)

如果没有以name:”user1”为条件的记录，那么就以name:”user1”这个条件和更新文档name:”user2”为基础创建文档，也就是说，先创建一个name:”user1”的文档，然后再更新name:”user1”为name:”user2”。

又例：

db.math.remove();

db.math.update({count:25},{$inc:{count:3}},true)

先清空文档，创建一个以”count”的值为25的文档，随后将这个值加3，最终得到count为28。

**2.save()函数**

Save()是一个shell函数，可以在文档不存在时进行插入，存在时进行更新。Sava()函数只有一个参数，参数为文档。如果参数的文档中的”\_id”键在集合中存在，save会调用upsert，否则会调用插入，代码如下：

var x = db.user.findOne()

x.num = 42

db.user.save(x)

要是这个文档含有”\_id”键，save会调用upsert，否则会调用插入。

等价于：

db.user.update({\_id:x.\_id},x)

#### 3.3.4 更新多个文档

默认情况下，更新只能对符合匹配条件的第一个文档执行操作。要是有多个文档符合条件，其余的文档不做变化，要是所有匹配到的文档都得到更新，可以设置update的第4个参数为true。

例如：假设要给所有在特定日期过生日的用户一份礼物，就可以使用多文档更新，将gift增加到他们的账号。

db.user.update({birthday:”12/21/2012”},{$set:{gift:”happy birthday”}},false,true)

这样就给生日为2012年12月21日的所有用户文档增加了“gift”键。

想要知道多文档更新到底更新了多少文档，可以运行getLastError命令

db.user.update({birthday:”12/21/2012”},

…{$set:{gift:”happy birthday”}},false,true)

db.runCommand({getLastError:1})

# 第2篇 应用篇

本篇主要讲解了MongoDB数据库的基本应用。注意包括MongoDBD的查询、Capped集合、GridFS存储文件、MapReduce统计等，通过本篇内容的学习，读者可以熟练的使用MongoDB来完成日常的业务需求。

## 第4章 查询

面向文档的NoSQL数据库主要解决的问题不是高性能地并发读/写，而是保证海量数据存储的同时，具备良好地查询性能。MongoDB 最大特点就是支持的查询语言非常强大，其语法类似于面向对象的查询语言，几乎可以实现类似于关系型数据库单表查询的绝大部分功能，而且还支持对数据建立索引。本章注意介绍查询，主要包括以下几个方面：

* 使用find()或者findOne()函数对数据库执行查询。
* 各种条件限制查询，以准确地查询出满足需求的数据。
* 查询指定返回的键。
* 查询给定键在集合中所有不同的值。
* 分组查询。
* 游标。
* 存储过程的介绍。

### 4.1 find介绍

MongoDB中使用find()函数进行查询，默认不带参数时是查询集合中的所有文档。find（）函数的第一个参数指定了要返回那些文档（参数形式为一个文档，就是所谓的查询条件）。

如果不传入参数（不指定查询条件），默认就是{}。例如代码如下：

>db.personalinfo.find()

当在查询条件的文档中添加键值对时，就限定了查找的条件。要查询的字段为键

，值为限定条件，值的类型为字段值的类型。也支持多条件查询。例如代码如下：

>db.personalinfo.find({“age”:26})

#### 4.1.1返回指定的键

但只需要查询文档中的某个字段的值时，该怎么办呢？可以通过find()或者findOne()函数的第二个参数来指定需要返回的字段。这样既节省了数据量的查询，有节省了客户端解码文档的时间和内存的消耗。

例如，指定返回name字段的值，代码如下：

>db.personalinfo.find({},{“name”:1})

{“\_id”:ObjectId(4f661cdf618a714de9afce33),“name”:“wangjunfeng”}

{“\_id”:ObjectId(4f661cdf618a714de9afce34),“name”:“yeyijian”}

以上代码通过设置find()函数的第二个参数来返回指定的字段，上面的例子指定返回name字段的值。

\_id这个键总是被返回，不管有没有指定。

第二个参数也可以用来除去查询结果中的某个键值。例如，当查询人员信息，不希望把年龄信息给其他人看到，那么可以这么做：

>db.personalinfo.find({},{“age”:0})

{“\_id”:ObjectId(4f661cdf618a714de9afce33),“name”:“wangjunfeng”}

{“\_id”:ObjectId(4f661cdf618a714de9afce34),“name”:“yeyijian”}

以上代码通过设置find()函数的第二个参数age为0，表示在该查询结果中不显示该字段。

总是返回的\_id键，也可以用这个参数防止其返回：

>db.personalinfo.find({},{“\_id”:0})

{ “name”:“wangjunfeng”,“age”：26}

{ “name”:“yeyijian”，“age”：25}

#### 4.1.2find查询限制

find()函数查询条件文档中的值必须是常量。但不能引用文档中的其他键的值。

### 4.2 条件操作符

最常用的操作符有<、<=、>、>=，下面的代码是在MongoDB中查询表示方式。

db.collection.find({“field”:{“$gt”:value}}) //大于：field>value

db.collection.find({“field”:{“$lt”:value}}) //小于：field<value

db.collection.find({“field”:{“$gte”:value}}) //大于等于：field>=value

db.collection.find({“field”:{“$lte”:value}}) //小于等于：field<=value

collection代码集合名称，field代码字段名称，value代表需要比较的值。

如果想要同时满足多个条件，则可以这样操作，代码如下：

db.collection.find({“field”:{“$gt”:value1，“$lt”:value2 }}) //value1<field<value2

#### 4.2.1 $all匹配所有

MongoDB中的$all操作符与SQL语法的in类似。但是不同的是，in只需满足()内的某一个值即可，而$all必须满足[]内的所有值。下面介绍$all操作符的用法。

下面集合中的4个文档：

>db.animal.find()

{“\_id”:1,“animal”：[“tiger”,“lion”，“monkey”，“elephant”]}

{“\_id”:2,“animal”：[“tiger”,“lion”，“horse”，“elephant”]}

{“\_id”:3,“animal”：[“tiger”,“lion”，“monkey”，“donkey”]}

{“\_id”:4,“animal”：[“horse”,“lion”，“giraffe”，“tiger”]}

我们需要找出既有tiger又有horse的文档，就可以用到$all来查询，代码如下：

>db.animal.find({“anmal”:{“$all”：[“tiger”,“horse”]}})

{“\_id”:2,“animal”：[“tiger”,“lion”，“horse”，“elephant”]}

{“\_id”:4,“animal”：[“horse”,“lion”，“giraffe”，“tiger”]}

数组元素的顺序无关要紧，查询出的第二个文档的horse在tiger的前面。

**注意：**

如果对一个元素操作，使用$all和不使用$all效果是一样的。例如，

>db.animal.find({“anmal”:{“$all”：[“tiger”]}})

>db.animal.find({“anmal”:“tiger”})

还可以使用数组的下标来定位元素，例如：

>db.animal.find({“anmal.2”:“giraffe”})

{“\_id”:4,“animal”：[“horse”,“lion”，“giraffe”，“tiger”]}

以上代码通过animal的下标来查找文档，下标都是从0 开始的。Animal.2表示animal数组的第三个元素，上面代码表示查询animal数组的第三个元素为giraffe的文档。

#### 4.2.2 $exists判定字段是否存在

如果需要判定一个字段在集合中是否存在，那么就是要$exists()函数来判定是否存在。

例如，查询存在field字段的所有文档，就可以使用以下格式语句：

>db.collection.find({“field”:{$exists：true}})

//collection代表集合名称，field代表需要判定的字段名称

例如，查询所有不存在field字段的所有文档，就可以使用以下格式语句：

>db.collection.find({“field”:{$exists：false}})

//collection代表集合名称，field代表需要判定的字段名称

#### 4.2.3 null值处理

null值的处理稍微有点奇怪，具体看下面的样例数据，代码如下：

> db.personalinfo.find()

{“\_id”:ObjectId(4f661cdf618a714de9afce33),“name”:“wangjunfeng”,“age”:nul}

{“\_id”:ObjectId(4f661cdf618a714de9afce34),“name”:“yeyijian”,“age\_1”:25}

{“\_id”:ObjectId(4f661cdf618a714de9afce35),“name”:“sunqignsong”,“age”:26}

查询age的值为null的文档，代码如下

> db.personalinfo.find(“age”:null)

{“\_id”:ObjectId(4f661cdf618a714de9afce33),“name”:“wangjunfeng”,“age”:nul}

{“\_id”:ObjectId(4f661cdf618a714de9afce34),“name”:“yeyijian”,“age\_1”:25}

以上代码查询age的值为null的文档，但是结果中查询显示了不存在age字段的文档也查询查询出来了。

为什么查询数据不是我们想要的数据呢？

因为null不仅匹配自身，而且还匹配“不存在的”。所有age\_1也查询出来了。

如果只想匹配键值为null的文档，那么嗨需要使用$exist操作符判定键值已经存在：

> db.personalinfo.find(“age”:{“$in”:[null],“$exists”:true)

{“\_id”:ObjectId(4f661cdf618a714de9afce33),“name”:

“wangjunfeng”,“age”:nul}

以上代码，使用了$in和$existe操作符配合查询出了age值为null的文档。

#### 4.2.4 $mod取模运算

$mod是MongoDB中取模运算操作符，例如，需要查询field取模10等于1的文档。代码如下：

>db.collection.find({field:{$mod:[10,1]}})

//代表返回field除以10余1的条件成立的结果

#### 4.2.5 $not取反运算

db.collection.find({“field”:{“$not”:{“$mod”:[5,1]}}})

//代表field除以5不余1的条件成立的结果

#### 4.2.6 $ne不等于

需要查询不等于某个值的文档，就可以使用$ne操作符，例如，需要查询field值不等于N的文档。代码如下：

>db.collection.find({field:{$ne:N}})

#### 4.2.7 $in包含

MongoDB中的$in包含与SQL标准语句的用途是一样的，即要查询的是一系列枚举的范围内。例如，查询field的值N1,N2,N3,…范围内的文档，代码如下：

db.collection.find({“field”:{“$in”: [N1,N2,N3,…]}})

#### 4.2.8 $nin不包含

MongoDB中的$nin包含与SQL标准语句的用途是一样的，即要查询的是一系列枚举的范围外。例如，查询field的值N1,N2,N3,…范围外的文档，代码如下：

db.collection.find({“field”:{“$nin”: [N1,N2,N3,…]}})

#### 4.2.9 $size数组元素个数

$size用来查询指定长度的数组，例如，查询testarray集合中有如下文档：

>db.testaaray.find()

{“\_id”:ObjectId(4f661cdf618a714de9afce36),“arr\_number”:[1,2,3]}

{“\_id”:ObjectId(4f661cdf618a714de9afce36),“arr\_number”:[4,5,6,7]}

{“\_id”:ObjectId(4f661cdf618a714de9afce36),“arr\_number”:[8,9,10]}

{“\_id”:ObjectId(4f661cdf618a714de9afce36),“arr\_number”:[11,12]}

查询arr\_number数组长度为3的文档，代码如下：

>db.testaaray.find({“arr\_number”:{“$size”:3}})

{“\_id”:ObjectId(4f661cdf618a714de9afce36),“arr\_number”:[1,2,3]}

{“\_id”:ObjectId(4f661cdf618a714de9afce36),“arr\_number”:[8,9,10]}

#### 4.2.10 正则表达式匹配

正则表达式可以灵活有效地匹配字符串。其中，查询过程中“/”与关系型数据库中的“%”用法类似，可以代替一个或者对过字符串。

例如，查询name中含有S的文档，在关系型数据库中是“%S%”,而在MongoDB中，代码如下：

> db.personalinfo.find({“name”：/S/})

{“\_id”:ObjectId(4f661cdf618a714de9afce35),“name”:“sunqignsong”,“age”:26}

#### 4.2.11 $slice操作符

$slice返回数组的一个子集合。

例如：假设有一个博客文章的文档，要想返回前10条评论

db.blog.posts.findOne({“title”:”123”},{“comm”:{“$slice”:10}})

例如：假设有一个博客文章的文档，要想返回后10条评论

db.blog.posts.findOne({“title”:”123”},{“comm”:{“$slice”:-10}})

“$slice”:[20,10] //返回从21~30条的评论

#### 4.2.12 Javascript查询和$where查询

$where可以执行任意Javascript作为查询的一部分，这就使得查询集合能做任何事情。例如，查询testnum集合中有如下文档：

>db.testnum.find()

{“\_id”:ObjectId(4f661cdf618a714de9afce37),“num”:1}

{“\_id”:ObjectId(4f661cdf618a714de9afce38),“num”:2}

{“\_id”:ObjectId(4f661cdf618a714de9afce39),“num”:3}

{“\_id”:ObjectId(4f661cdf618a714de9afce40),“num”:4}

{“\_id”:ObjectId(4f661cdf618a714de9afce41),“num”:5}

查询大于3的文档，代码如下：

>db.testnum.find({“num”:{“$gt”:3}})

{“\_id”:ObjectId(4f661cdf618a714de9afce40),“num”:4}

{“\_id”:ObjectId(4f661cdf618a714de9afce41),“num”:5}

>db.testnum.find({“$where”:“this.num>3})”)

{“\_id”:ObjectId(4f661cdf618a714de9afce40),“num”:4}

{“\_id”:ObjectId(4f661cdf618a714de9afce41),“num”:5}

>db.testnum.find({“this.num>3)”})

{“\_id”:ObjectId(4f661cdf618a714de9afce40),“num”:4}

{“\_id”:ObjectId(4f661cdf618a714de9afce41),“num”:5}

>f=function(){return this.a>3;}

Function()

{return this.a>3;

}

>db.testnum.fin(f)

{“\_id”:ObjectId(4f661cdf618a714de9afce40),“num”:4}

{“\_id”:ObjectId(4f661cdf618a714de9afce41),“num”:5}

可以使用一个字符串来指定$where查询，例如，代码如下：

>db.testnum.find({“$where”:“this.x+this.y=9” })

{“\_id”:ObjectId(4f661cdf618a714de9afce40),“x”:4,“y”:5}

{“\_id”:ObjectId(4f661cdf618a714de9afce41),“x”:6,“y”:3}

**或者**

>db.testnum.find({“$where”:“function(){return this.x+this.y=9})”})

{“\_id”:ObjectId(4f661cdf618a714de9afce40),“x”:4,“y”:5}

{“\_id”:ObjectId(4f661cdf618a714de9afce41),“x”:6,“y”:3}

**注意：**

使用$where查询，首先将每个文档从BSON转换成Javascript对象，然后通过$where的表达式来运行，查询速度会相对慢，所以不建议使用$where查询

（**不在非常必要时，避免使用**）。

#### 4.2.13 count查询记录条数

如果需要查询集合中的记录总数，可以使用count()方法来查询。代码格式如下：

>db.collection.find().count()

也可以传递查询，Mongo则会计算查询结果的数量：

db.user.insert({“x”:2})

db.user.count()

db.user.count({“x”:2})

但是，增加查询条件会使得count查询速度变慢。

#### 4.2.14 limit现在返回记录数

最常用的查询就是限制返回结果的记录数量，例如，需要返回N条数据，代码如下：

>db.collection.find().limit(N)

#### 4.2.15 skip限制返回记录的起点

如果需要从第N条开始返回记录，就可以使用skip方式来完成，代码如下：

db.collection.find().skip(N)

#### 4.2.16 sort排序

Sort()用一个对象作为参数：一组键值对，键对应文档的键名（字段名），值代表升序或者降序（**1表示升序排序，-1表示降序排序**）。

例如，下面代码对数据按age进行升序排序，代码如下：

db.personalinfo.find().sort(“age”:1)

也可以对多个键进行排序的，代码如下：

db.personalinfo.find().sort(“name”:1,“age”:-1)

#### 4.2.17 分页查询

可以使用limit()、skip()、sort()三个函数组合进行查询。例如。查询field为value的数据进行分页查询，每页显示N条，按照field1进行升序排序：

//第一页

var page1=db.collection.find({field:value}).limit(N).sort(field:1)

//第二页

var page1=db.collection.find({field:value}).skip((2-1)\*N).limit(N).sort(field:1)

//第三页

var page1=db.collection.find({field:value}).skip((3-1)\*N).limit(N).sort(field:1)

…

//第N1页

var page1=db.collection.find({field:value}).skip((N1-1)\*N).limit(N).sort(field:1)

#### 4.2.18 随机抽取查询

在一个集合中随机抽取一个文档，可以选择一个从0到文档数量之间的随机数进行查询。

例如，下面随机抽取一个文档，代码如下：

>var total=db.personalinfo.cont()

>var random=Math.floor(Math.random()\*tatol)

>db.personalinfo.find().skip(random).limit(1)

{“\_id”:ObjectId(4f661cdf618a714de9afce35),“name”:“sunqignsong”,“age”:26}

如果有抽取随机文档的需求，可以在插入时添加一个键，值为随机数，代码如下：

>db.personalinfo.insert({“name”:“sunqignsong”,“age”:26,“random”：Math.random()})

这样在抽取随机文档时，就非常简单了：

>db.personalinfo.findOne({“random”:Math.random()})

>db.personalinfo.findOne({“random”:{“$gt”:Math.random()}})

### 4.3 distinct找出给定键所有不同的值

Distinct用来找出给定键的所有不同值。使用时必须指定集合和键，代码格式如下：

>db.runCommand({“distinct”:”collection”,”key”:”keyname”})

### 4.4 group分组

Group做聚合，首先选定分组所依据的键，然后MongoDB就会将集合依据选定键的不同分成若干组。然后可以通过聚合每一组内的文档，产生一个结果文档。下面给大家介绍group的具体用户。

测试条件：Windows+MongoDB 1.8.2

先插入测试数据：

for(var i=1; i<20; i++){

var num=i%6;

db.test.insert({\_id:i,name:"user\_"+i,age:num});

}

**1.普通分组查询**

db.test.group({key:{age:true},initial:{num:0},$reduce:function(doc,prev){

prev.num++

}});

db.runCommand({group:

{

ns:"test",

key:{age:true},

initial:{num:0},

$reduce:function(doc,prev){

prev.num++}

}

});

**2.筛选后再分组**

db.test.group({key:{age:true},initial:{num:0},$reduce:function(doc,prev){

prev.num++

},

condition:{age:{$gt:2}}

});

db.runCommand({group:

{

ns:"test",

key:{age:true},

initial:{num:0},

$reduce:function(doc,prev){

prev.num++},

condition:{age:{$gt:2}}

}

});

普通的$where查询：

db.test.find({$where:function(){

return this.age>2;

}

});

group联合$where查询

db.test.group({key:{age:true},initial:{num:0},$reduce:function(doc,prev){

prev.num++

},

condition:{$where:function(){

return this.age>2;

}

}

});

**3.使用函数返回值分组**

//注意，$keyf指定的函数一定要返回一个对象

db.test.group({$keyf:function(doc){return {age:doc.age};},initial:{num:0},$reduce:function(doc,prev){

prev.num++

}

});

db.runCommand({group:

{

ns:"test",

$keyf:function(doc){return {age:doc.age};},

initial:{num:0},

$reduce:function(doc,prev){

prev.num++}

}

});

**4.使用终结器**

db.test.group({$keyf:function(doc){return {age:doc.age};},initial:{num:0},$reduce:function(doc,prev){

prev.num++

},

finalize: function(doc){ doc.count=doc.num;delete doc.num; }

});

db.runCommand({group:

{

ns:"test",

$keyf:function(doc){return {age:doc.age};},

initial:{num:0},

$reduce:function(doc,prev){

prev.num++},

finalize: function(doc){ doc.count=doc.num;delete doc.num; }

}

});

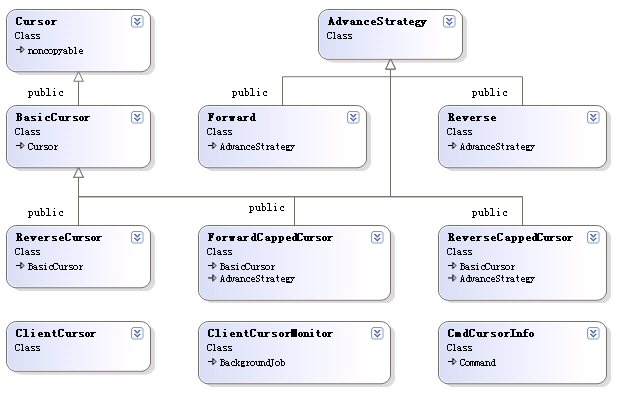
**有关MapReduce**

//首先插入测试数据  
for(var i=1;i<21;i++)  
{  
db.test.insert({\_id:i,name:'mm'+i});  
}  
  
//进行mapreduce  
db.runCommand(  
{  
mapreduce:'test',  
map:function(){emit(this.name.substr(0,3),this);},  
reduce:function(key,vals){return vals[0];}, //注意：vals是一个Object对象而不是数组  
out:'wq'  
});  
  
注意：  
1.mapreduce是根据map函数里调用的emit函数的第一个参数来进行分组的  
2.仅当根据分组键分组后一个键匹配多个文档，才会将key和文档集合交由reduce函数处理。例如：  
db.runCommand(  
{  
mapreduce:'test',  
map:function(){emit(this.name.substr(0,3),this);},  
reduce:function(key,vals){return 'wq';},   
out:'wq'  
});  
执行mapreduce命令后，再查看wq表数据：  
db.wq.find()  
  
{ "\_id" : "mm1", "value" : "wq" }  
{ "\_id" : "mm2", "value" : "wq" }  
{ "\_id" : "mm3", "value" : { "\_id" : 3, "name" : "mm3" } }  
{ "\_id" : "mm4", "value" : { "\_id" : 4, "name" : "mm4" } }  
{ "\_id" : "mm5", "value" : { "\_id" : 5, "name" : "mm5" } }  
{ "\_id" : "mm6", "value" : { "\_id" : 6, "name" : "mm6" } }  
{ "\_id" : "mm7", "value" : { "\_id" : 7, "name" : "mm7" } }  
{ "\_id" : "mm8", "value" : { "\_id" : 8, "name" : "mm8" } }  
{ "\_id" : "mm9", "value" : { "\_id" : 9, "name" : "mm9" } }

### 4.5 游标

[**Mongodb源码分析--游标Cursor**](http://www.cnblogs.com/daizhj/archive/2011/04/15/mongodb_cursor_source_code.html)

在Mongodb中，其提供了类似关系型数据中cursor对象来遍历数据集合，同时mongodb并要根据不同的场景生成不同的游标对象（cursor）,比如顺序遍历游标（basicCursor）,反向游标（reverseCursor), B树索引游标（btreeCursor）等。

下面是其游标体系架构类图(位于cursor.cpp, cursor.h, clientcursor.cpp, clientcursor.h）：  


从该图中，可以看到除了（ClientCursor）之外，其余游标均继承自Cursor这个类（基类），下面我们看一下其具体实现：

[复制代码](javascript:void(0);)

class Cursor : boost::noncopyable//使类和派生类不可复制  
{  
virtual bool ok() = 0;//游标当前指向的对象是否有效  
bool eof() { return !ok(); }//是否已到尾部  
virtual Record\* \_current() = 0;//游标当前指向的记录（记录是组成数据文件的最基本单位）  
virtual BSONObj current() = 0;//游标当前指向的BSONObj对象  
virtual DiskLoc currLoc() = 0;//游标当前指向的DiskLoc  
virtual bool advance() = 0; /\*true=ok,将游标指向到下一条记录所在位置\*/  
virtual BSONObj currKey() const { return BSONObj(); }  
  
/\* 标识游标是否为Tailable类型,该类型支持获取最后一条记录后,不马上关闭游标,以便持续获取后面新添加的记录\*/  
virtual bool tailable() {  
return false;  
}  
//设置游标为Tailable类型  
virtual void setTailable() {}  
.....  
}

[复制代码](javascript:void(0);)

在mongodb中，提供了两种遍历数据集合的方向，分别是“向前”和“倒转”方式，其声明如下：

[复制代码](javascript:void(0);)

class AdvanceStrategy {  
public:  
virtual ~AdvanceStrategy() { }  
virtual DiskLoc next( const DiskLoc &prev ) const = 0;  
};  
  
const AdvanceStrategy \*forward(); //向前  
const AdvanceStrategy \*reverse(); //倒转

[复制代码](javascript:void(0);)

下面是其实现方式如下：

[复制代码](javascript:void(0);)

class Forward : public AdvanceStrategy {  
virtual DiskLoc next( const DiskLoc &prev ) const {  
return prev.rec()->getNext( prev );  
}  
} \_forward;  
  
class Reverse : public AdvanceStrategy {  
virtual DiskLoc next( const DiskLoc &prev ) const {  
return prev.rec()->getPrev( prev );  
}  
} \_reverse;  
  
const AdvanceStrategy \*forward() {  
return &\_forward;  
}  
const AdvanceStrategy \*reverse() {  
return &\_reverse;  
}

[复制代码](javascript:void(0);)

看到这里，我们有必须简要说明一下mongofile文件的结构，见下面说明：

[复制代码](javascript:void(0);)

/\* a datafile - i.e. the "dbname.<#>" files :  
  
----------------------  
DataFileHeader :数据头文件信息，包括版本，文件长度，使用情况等  
----------------------  
Extent (for a particular namespace) 特定namespace下的extent，可理解为数据集合  
Record ： 单条数据记录  
...  
Record (some chained for unused space)  
----------------------  
more Extents... 其它extent  
----------------------  
\*/

[复制代码](javascript:void(0);)

在一个数据库文件中，同一个namespace的extent可以有多个，每一个extent都有一些记录(record)组成，如果访问record，可以使用diskloc加上文件偏移（getOfs：位于diskloc中）获取。  
同时每个extent中包括还包括两个重要属性：

DiskLoc xnext, xprev; /\* next/prev extent for this namespace \*/

它们分别记录了同一namespace下，在extent链表中，当前extent的前或后一个extent的位置信息，上面AdvanceStrategy中的next方法即实现了在两种遍历方向（上面已提到）上，在extent链接中跳转的方式，比如在forward方向：

[复制代码](javascript:void(0);)

inline DiskLoc Record::getNext(const DiskLoc& myLoc) {  
//如果当前 Record的nextOfs偏移不为空，表示在当前extent中还有后续记录可访问  
if ( nextOfs != DiskLoc::NullOfs ) {  
/\* defensive \*/  
if ( nextOfs >= 0 && nextOfs < 10 ) {//是否为已删除的记录  
sayDbContext("Assertion failure - Record::getNext() referencing a deleted record?");  
return DiskLoc();  
}  
return DiskLoc(myLoc.a(), nextOfs);//获取下一条记录  
}  
Extent \*e = myExtent(myLoc);//获取当前记录所属的Extent  
while ( 1 ) {  
if ( e->xnext.isNull() )  
return DiskLoc(); //已到表尾.  
e = e->xnext.ext();//跳转到下一个extent（以便进行next遍历）  
if ( !e->firstRecord.isNull() )  
break;  
// entire extent could be empty, keep looking  
}  
return e->firstRecord;//获取下一个extent中的第一条记录  
}

[复制代码](javascript:void(0);)

在每个extent对象中，其还包括另外两个属性 firstRecord，lastRecord，两者皆为DiskLoc类型，顾名思义，它们分别指向当前extent的第一条和最后一条记录所在位置，这种定义它们是为了后者在extent中进行跳转时使用，当前如果在更加复杂的capped collection情况下，其值在会删除记录等操作时不断更新，比如下面代码：

[复制代码](javascript:void(0);)

//namespace.cpp 文件912行，该方法在删除记录时调用  
void DataFileMgr::\_deleteRecord(NamespaceDetails \*d, const char \*ns, Record \*todelete, const DiskLoc& dl) {  
......  
//extents是一个数据文件区域，该区域有所有记录（records）均属于同一个名空间namespace  
/\* remove ourself from extent pointers \*/  
{  
Extent \*e = getDur().writing( todelete->myExtent(dl) );  
if ( e->firstRecord == dl ) {//如果要删除记录为该extents区域第一条记录时  
if ( todelete->nextOfs == DiskLoc::NullOfs )//且为唯一记录时  
e->firstRecord.Null();//则该空间第一元素为空  
else //将当前空间第一条（有效）记录后移一位  
e->firstRecord.set(dl.a(), todelete->nextOfs);  
}  
if ( e->lastRecord == dl ) {//如果要删除记录为该extents区域最后一条记录时  
if ( todelete->prevOfs == DiskLoc::NullOfs )//如果要删除记录的前一条信息位置为空时  
e->lastRecord.Null();//该空间最后一条记录清空  
else //设置该空间最后一条（有效）记录位置前移一位  
e->lastRecord.set(dl.a(), todelete->prevOfs);  
}  
}  
......  
}

[复制代码](javascript:void(0);)

介绍了cursor基类的定义和遍历方向这两个基本概念后，下面介绍一下在mongodb中，广泛使用的是basicCursor,其定义如下：

[复制代码](javascript:void(0);)

class BasicCursor : public Cursor {  
public:  
BasicCursor(DiskLoc dl, const AdvanceStrategy \*\_s = forward()) : curr(dl), s( \_s ), \_nscanned() {  
incNscanned();  
init();  
}  
BasicCursor(const AdvanceStrategy \*\_s = forward()) : s( \_s ), \_nscanned() {  
init();  
}  
bool ok() { return !curr.isNull(); }  
Record\* \_current() {  
assert( ok() );  
return curr.rec();  
}  
BSONObj current() {  
Record \*r = \_current();  
BSONObj j(r);  
return j;  
}  
virtual DiskLoc currLoc() { return curr; }  
virtual DiskLoc refLoc() { return curr.isNull() ? last : curr; }  
bool advance();  
virtual string toString() { return "BasicCursor"; }  
virtual void setTailable() {  
if ( !curr.isNull() || !last.isNull() )  
tailable\_ = true;  
}  
virtual bool tailable() { return tailable\_; }  
......  
};

[复制代码](javascript:void(0);)

可认看到在其构造函数时，使用了forward方向的遍历方式, 即然定义了Forward方向的游标，mongodb接下来定义了Reverse方向的游标：

[复制代码](javascript:void(0);)

/\* 用于排序 { $natural: -1 } \*/  
class ReverseCursor : public BasicCursor {  
public:  
ReverseCursor(DiskLoc dl) : BasicCursor( dl, reverse() ) { }  
ReverseCursor() : BasicCursor( reverse() ) { }  
virtual string toString() { return "ReverseCursor"; }  
};

[复制代码](javascript:void(0);)

另外为了支持capped collection集合类型（有关capped collection，参见[这篇链接](http://www.mongodb.org/display/DOCS/Capped+Collections)），mongodb分别定义了ForwardCappedCursor和ReverseCappedCursor：

[复制代码](javascript:void(0);)

class ForwardCappedCursor : public BasicCursor, public AdvanceStrategy {  
public:  
ForwardCappedCursor( NamespaceDetails \*nsd = 0, const DiskLoc &startLoc = DiskLoc() );  
virtual string toString() {  
return "ForwardCappedCursor";  
}  
virtual DiskLoc next( const DiskLoc &prev ) const;  
virtual bool capped() const { return true; }  
private:  
NamespaceDetails \*nsd;  
};  
  
class ReverseCappedCursor : public BasicCursor, public AdvanceStrategy {  
public:  
ReverseCappedCursor( NamespaceDetails \*nsd = 0, const DiskLoc &startLoc = DiskLoc() );  
virtual string toString() {  
return "ReverseCappedCursor";  
}  
virtual DiskLoc next( const DiskLoc &prev ) const;  
virtual bool capped() const { return true; }  
private:  
NamespaceDetails \*nsd;  
};

[复制代码](javascript:void(0);)

只不过在ForwardCappedCursor和ReverseCappedCursor中，实现next方法会更复杂一下，因为其要考虑删除的记录不在遍历结果中的情况。相当内容详见cursor.cpp的实现代码:)  
  
介绍游标和mongofile结构之后，我们大体知道了mongodb如果遍历数据文件，另外mongodb使用了b树索引来加快查询效率，因此mongodb也提供了相应的btreeCursor，其主要用于遍历内存中的b树索引。  
除此以外，为了方便client端使用cursor访问数据库，mongodb提供了ClientCursor，其对Cursor进一步封装（详见clientcursor.h）。  
  
下面我们看一下mongodb如果要据查询方式来确定使用那种类型游标的：

[复制代码](javascript:void(0);)

//pdfile.cpp 文件639行，查询从指定记录位置startLoc开始的记录，这里要据不同的条件使用不同的注季  
shared\_ptr<Cursor> DataFileMgr::findAll(const char \*ns, const DiskLoc &startLoc) {  
NamespaceDetails \* d = nsdetails( ns );  
if ( ! d )  
return shared\_ptr<Cursor>(new BasicCursor(DiskLoc()));  
  
DiskLoc loc = d->firstExtent;  
Extent \*e = getExtent(loc);  
  
......  
  
if ( d->capped )  
return shared\_ptr<Cursor>( new ForwardCappedCursor( d , startLoc ) );  
  
if ( !startLoc.isNull() )  
return shared\_ptr<Cursor>(new BasicCursor( startLoc ));  
  
......  
return shared\_ptr<Cursor>(new BasicCursor( e->firstRecord ));  
}

[复制代码](javascript:void(0);)

到这里，可以看了，mongodb在cursor的设计和使用方式上是基于“策略模式”（strategy pattern）的，如下图:



其中cursor就是各种遍历数据集合的策略，而pdfile.cpp就是持有相应cursor的上下文(context) ，该模式也是使用比较广泛的一种设置模式，好处这里就不多说了。  
  
好了，今天的内容到这里就告一段落了，在接下来的文章中，将会介绍mongodb中mmap的使用场景。

### 4.6 存储过程

[**基于Mongodb进行分布式数据存储**](http://www.cnblogs.com/daizhj/archive/2010/09/07/1820528.html)

注：本文是研究Mongodb分布式数据存储的副产品，通过本文的相关步骤可以将一个大表中的数据分布到几个mongo服务器上。

MongoDB的1.6版本中auto-sharding功能基本稳定并可以尝试放到生产环境下使用。因为其是auto-sharding，即mongodb通过mongos(一个自动分片模块，用于构建一个大规模的可扩展的数据库集群,这个集群可以并入动态增加的机器)自动建立一个水平扩展的数据库集群系统，将数据库分表存储在sharding的各个节点上。

一个mongodb集群包括一些shards（包括一些mongod进程），mongos路由进程，一个或多个config服务器

（注：本文的测试用例需求64位的mongo程序,因为我在32位的mongo没成功过）。

下面是一些相关词汇说明:  
**Shards** : 每一个shard包括一个或多个服务和存储数据的mongod进程（mongod是MongoDB数据的核心进程）典型的每个shard开启多个服务来提高服务的可用性。这些服务/mongod进程在shard中组成一个复制集  
  
**Chunks**: Chunk是一个来自特殊集合中的一个数据范围，（collection，minKey，maxKey）描叙一个chunk，它介于minKey和maxKey范围之间。例如chunks 的maxsize大小是100M，如果一个文件达到或超过这个范围时，会被切分到2个新的chunks中。当一个shard的数据过量时，chunks将会被迁移到其他的shards上。同样，chunks也可以迁移到其他的shards上

**Config Servers** : Config服务器存储着集群的metadata信息，包括每个服务器，每个shard的基本信息和chunk信息Config服务器主要存储的是chunk信息。每一个config服务器都复制了完整的chunk信息。  
  
  
接着看一下要配置的测试环境信息：

模拟2个shard服务和一个config服务, 均运行在10.0.4.85机器上，只是端口不同  
Shard1：27020  
Shard2：27021  
Config：27022  
Mongos启动时默认使用的27017端口

在C,D,E磁盘下分别建立如下文件夹：

mongodb\bin

mongodb\db

然后用CMD命令行依次打开相应文件夹下的mongd文件:

c:\mongodb\bin\mongod --dbpath c:\mongodb\db\ --port 27020

d:\mongodb\bin\mongod --dbpath d:\mongodb\db\ --port 27021

e:\mongodb\bin\mongod **--configsvr** --dbpath e:\mongodb\db\ --port 27022 (注:config配置服务器)

启动mongos时，默认开启了27017端口

e:\mongodb\bin\mongos --configdb 10.0.4.85:27022

然后打开mongo:

E:\mongodb\bin>mongo 回车 （有时加端口会造成下面的addshard命令出问题）

> use admin  
switched to db admin  
> db.runCommand( { addshard : "10.0.4.85:27020", allowLocal : 1, maxSize:2 , minKey:1, maxKey:10} )

--添加sharding,maxsize单位是M,此处设置比较小的数值只为演示sharding效果

{ "shardAdded" : "shard0000", "ok" : 1 }  
> db.runCommand( { addshard : "10.0.4.85:27021", allowLocal : 1, minKey:1000} )  
{ "shardAdded" : "shard0001", "ok" : 1 }

注：如果要移除sharding，可用下面写法

db.runCommand( { removeshard : "localhost:10000" } );

> db.runCommand({listshards:1}); 查看shard节点列表

[复制代码](javascript:void(0);)

http://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttp://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

{  
"shards" : [  
{  
"\_id" : "shard0000",  
"host" : "10.0.4.85:27020"  
},  
{  
"\_id" : "shard0001",  
"host" : "10.0.4.85:27021"  
}  
],  
"ok" : 1  
}

[复制代码](javascript:void(0);)

接下来创建相应数据库并设置其"可以sharding",新建自动切片的库user001：

> config = connect("10.0.4.85:27022")  
> config = config.getSisterDB("config")  
> dnt\_mongodb=db.getSisterDB("dnt\_mongodb");  
dnt\_mongodb  
> db.runCommand({enablesharding:"dnt\_mongodb"})  
{ "ok" : 1 }

注：一旦enable了个数据库，mongos将会把数据库里的不同数据集放在不同的分片上。除非数据集被分片(下面会设置)，否则一个数据集的所有数据将放在一个分片上。  
  
> db.printShardingStatus();

[复制代码](javascript:void(0);)

http://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttp://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

--- Sharding Status ---  
sharding version: { "\_id" : 1, "version" : 3 }  
shards:  
{ "\_id" : "shard0000", "host" : "10.0.4.85:27020" }  
{ "\_id" : "shard0001", "host" : "10.0.4.85:27021" }  
databases:  
{ "\_id" : "admin", "partitioned" : false, "primary" : "config" }  
{ "\_id" : "dnt\_mongodb", "partitioned" : true, "primary" : "shard0000" }

[复制代码](javascript:void(0);)

> db.runCommand( { shardcollection : "dnt\_mongodb.posts1", key : {\_id : 1}, unique: true } )

{ "collectionsharded" : "dnt\_mongodb.posts1", "ok" : 1 }

[复制代码](javascript:void(0);)

http://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttp://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

--使用shardcollection 命令分隔数据集，key自动生成 [必须为唯一索引unique index]。   
  
如果要进行GridFS sharding，则需进行如下设置：  
db.runCommand( { shardcollection : "dnt\_mongodb.attach\_gfstream.chunks", key : { files\_id : 1 } } )  
{"ok" : 1} ，更多内容参见http://eshilin.blog.163.com/blog/static/13288033020106215227346/

[复制代码](javascript:void(0);)

> db.printShardingStatus()

[复制代码](javascript:void(0);)

http://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttp://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

--- Sharding Status ---  
sharding version: { "\_id" : 1, "version" : 3 }  
shards:  
{ "\_id" : "shard0000", "host" : "localhost:27020" }  
{ "\_id" : "shard0001", "host" : "localhost:27021" }  
databases:  
{ "\_id" : "admin", "partitioned" : false, "primary" : "config" }  
{ "\_id" : "user001", "partitioned" : true, "primary" : "shard0000" }  
dnt\_mongodb.posts1e chunks:  
{ "name" : { $minKey : 1 } } -->> { "name" : { $maxKey :  
1 } } on : shard0000 { "t" : 1000, "i" : 0

[复制代码](javascript:void(0);)

下面我用一个工具来批量向dnt\_mongodb数据库的 posts1表中导入数据，大约是16万条数据。导入过程中mongos会显示类似如下信息：

[复制代码](javascript:void(0);)

http://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttp://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

Tue Sep 07 12:13:15 [conn14] autosplitting dnt\_mongodb.posts1 size: 47273960 shard: ns:dnt\_mongodb.posts1 at: shard0000:10.0.4.85:27020 lastmod: 1|0 min: { \_id: MinKey } max: { \_id: MaxKey } on: { \_id: 19 }(splitThreshold 47185920)  
Tue Sep 07 12:13:15 [conn14] config change: { \_id: "4\_85-2010-09-07T04:13:15-0", server: "4\_85", time: new Date(1283832795994), what: "split", ns: "dnt\_mongodb.posts1", details: { before: { min: { \_id: MinKey }, max: { \_id: MaxKey } }, left: { min: { \_id: MinKey }, max: { \_id: 19 } }, right: { min: { \_id: 19 }, max: {\_id: MaxKey } } } }  
Tue Sep 07 12:13:16 [conn14] moving chunk (auto): ns:dnt\_mongodb.posts1 at: shard0000:10.0.4.85:27020 lastmod: 1|1 min: { \_id: MinKey } max: { \_id: 19 } to: shard0001:10.0.4.85:27021 #objects: 0  
Tue Sep 07 12:13:16 [conn14] moving chunk ns: dnt\_mongodb.posts1 moving ( ns:dnt\_mongodb.posts1 at: shard0000:10.0.4.85:27020 lastmod: 1|1 min: { \_id: MinKey }max: { \_id: 19 }) shard0000:10.0.4.85:27020 -> shard0001:10.0.4.85:27021  
Tue Sep 07 12:13:23 [WriteBackListener] ~ScopedDBConnection: \_conn != null  
Tue Sep 07 12:13:23 [WriteBackListener] ERROR: splitIfShould failed: ns: dnt\_mongodb.posts1 findOne has stale config  
Tue Sep 07 12:13:28 [WriteBackListener] autosplitting dnt\_mongodb.posts1 size: 54106804 shard: ns:dnt\_mongodb.posts1 at: shard0000:10.0.4.85:27020 lastmod: 2|1min: { \_id: 19 } max: { \_id: MaxKey } on: { \_id: 71452 }(splitThreshold 47185920)  
Tue Sep 07 12:13:28 [WriteBackListener] config change: { \_id: "4\_85-2010-09-07T04:13:28-1", server: "4\_85", time: new Date(1283832808738), what: "split", ns: "dnt\_mongodb.posts1", details: { before: { min: { \_id: 19 }, max: { \_id: MaxKey }}, left: { min: { \_id: 19 }, max: { \_id: 71452 } }, right: { min: { \_id: 71452 }, max: { \_id: MaxKey } } } }

[复制代码](javascript:void(0);)

在完成自动sharding之后，可以使用mongo看一下结果:

> use dnt\_mongodb  
switched to db dnt\_mongodb  
> show collections  
posts1  
system.indexes  
> db.posts1.stats()

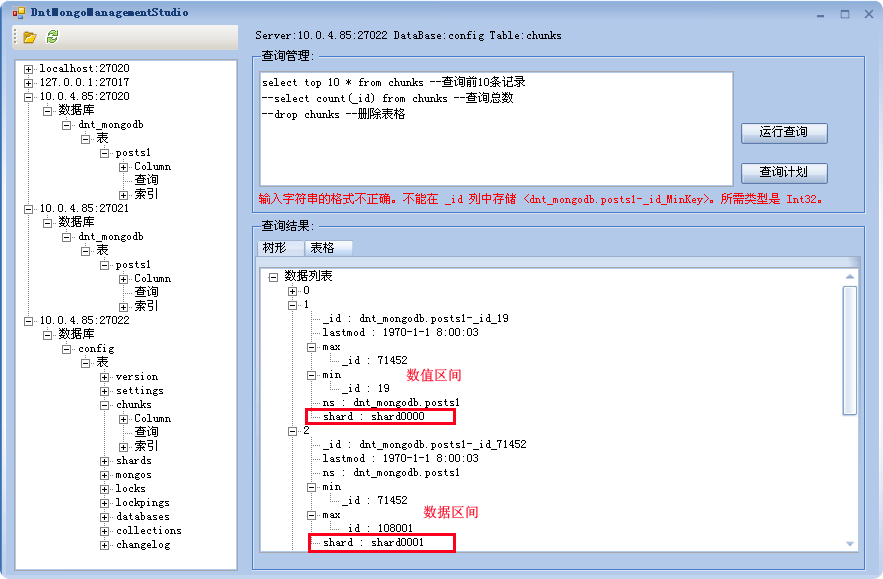
[复制代码](javascript:void(0);)

http://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttp://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

{  
"sharded" : true,  
"ns" : "dnt\_mongodb.posts1",  
"count" : 161531,  
"size" : 195882316,  
"avgObjSize" : 1212.6608267143768,  
"storageSize" : 231467776,  
"nindexes" : 1,  
"nchunks" : 5,  
"shards" : {  
"shard0000" : {  
"ns" : "dnt\_mongodb.posts1",  
"count" : 62434,  
"size" : 54525632,  
"avgObjSize" : 873.3323509626165,  
"storageSize" : 65217024,  
"numExtents" : 10,  
"nindexes" : 1,  
"lastExtentSize" : 17394176,  
"paddingFactor" : 1,  
"flags" : 1,  
"totalIndexSize" : 2179072,  
"indexSizes" : {  
"\_id\_" : 2179072  
},  
"ok" : 1  
},  
"shard0001" : {  
"ns" : "dnt\_mongodb.posts1",  
"count" : 99097,  
"size" : 141356684,  
"avgObjSize" : 1426.4476623913943,  
"storageSize" : 166250752,  
"numExtents" : 12,  
"nindexes" : 1,  
"lastExtentSize" : 37473024,  
"paddingFactor" : 1,  
"flags" : 1,  
"totalIndexSize" : 3424256,  
"indexSizes" : {  
"\_id\_" : 3424256  
},  
"ok" : 1  
}  
},  
"ok" : 1  
}

[复制代码](javascript:void(0);)

通过上面的结果，可以出现16万条记录均分在了两个sharding上，其中shard0000中有62434条，shard0001中有99097条。下面看一下这两个sharding-chunk的分布情况（图中的错误提示‘输入字符串格式不正确’主要因为运行环境与编译程序使用的环境不同，一个是64，一个是32位系统）：



可以看到数据被按区间自动分割开了，有点像sqlserver的数据分区表，只不过这是自动完成的（目前我没找到可以手工指定区间上下限的方式，如有知道的TX可以跟我说一下）。当然在本文中的测试中，共有5个chunk,其中4个位于shard0001，这种情况可以在每次测试过程中会发生变化，包括两个sharding被分配的记录数。另外就是在mongodb移动过程前后会在shard0000上生成一个文件夹，里面包括一些bson文件，名字形如（表格+日期等信息）：

post-cleanup.2010-09-07T04-13-31.1.bson

该文件主要包括一些数据库，表结构及相关记录等信息，我想应该是用于数据恢复备份啥的。

## 第5章 Capped Collection(集合)

### 5.1 Capped Collection的介绍

Capped Collection 是一种特殊的集合，它大小固定，当集合的大小达到指定大小时，新数据覆盖老数据。Capped collections可以按照文档的插入顺序保存到集合中，而且这些文档在磁盘上存放位置也是按照插入顺序来保存的，所以当我们更新Capped collections中文档的时候，更新后的文档不可以超过之前文档的大小，这样话就可以确保所有文档在磁盘上的位置一直保持不变。  
由于Capped collection是按照文档的插入顺序而不是使用索引确定插入位置，这样的话可以提高增添数据的效率。MongoDB的操作日志文件oplog.rs就是利用Capped Collection来实现的。  
除此之外，Capped Collection还有以下的一些特点，首先是不允许删除，但是可以调用drop（）删除集合中的所有行，不许删除的原因也是为了保持每个文档在磁盘上的位置不变。在32位机器上一个capped collection的最大值约482.5M，64位上没有限制系统文件大小限制。不可以对 Capped Collection 进行分片，在 2.2 版本以后，创建的Capped Collection 默认在 \_id 字段上创建索引，而在 2.2 版本或以前没有。   
Capped Collection主要用于存储日志信息和缓存一些少用的文档。

### 5.2 Capped Collection 的应用

**1.创建Capped Collection**

**> db.createCollection("mycoll",{capped:true,size:1024,max:10})**  
**{ "ok" : 1 } > for(var i=1;i<1000;i++) db.mycoll.save({id:i,name:"Hello"}) > db.mycoll.count() 10 > db.mycoll.find() { "\_id" : ObjectId("5274b854f007ff8bb84f6347"), "id" : 990, "name" : "Hello" } { "\_id" : ObjectId("5274b854f007ff8bb84f6348"), "id" : 991, "name" : "Hello" } { "\_id" : ObjectId("5274b854f007ff8bb84f6349"), "id" : 992, "name" : "Hello" } { "\_id" : ObjectId("5274b854f007ff8bb84f634a"), "id" : 993, "name" : "Hello" } { "\_id" : ObjectId("5274b854f007ff8bb84f634b"), "id" : 994, "name" : "Hello" } { "\_id" : ObjectId("5274b854f007ff8bb84f634c"), "id" : 995, "name" : "Hello" } { "\_id" : ObjectId("5274b854f007ff8bb84f634d"), "id" : 996, "name" : "Hello" } { "\_id" : ObjectId("5274b854f007ff8bb84f634e"), "id" : 997, "name" : "Hello" } { "\_id" : ObjectId("5274b854f007ff8bb84f634f"), "id" : 998, "name" : "Hello" } { "\_id" : ObjectId("5274b854f007ff8bb84f6350"), "id" : 999, "name" : "Hello" }**  
**> db.mycoll.isCapped() true >db.mycoll.stats() { "ns" : "test.mycoll", "count" : 0, "size" : 0, "storageSize" : 4096, "numExtents" : 1, "nindexes" : 1, "lastExtentSize" : 4096, "paddingFactor" : 1, "systemFlags" : 1, "userFlags" : 0, "totalIndexSize" : 8176, "indexSizes" : { "\_id\_" : 8176 }, "capped" : true, "max" : NumberLong("9223372036854775807"), "ok" : 1 }**  
  
上面个创建了一个Capped Collection，占用1024个字节，最多10个文档，然后向其中插入了1000条数据通过**db.collection.isCapped()**和 **db.mycoll.stats()**命令可以查看一个集合是否是 Capped Collection 。  
  
**注意：**

1. 创建固定集合不像普通集合，固定集合需要显示的创建使用  
   b.创建固定集合必须指定其大小否则会报错：  
   **> db.createCollection("mycoll",{capped:true})**  
   **{ "errmsg" : "exception: specify size:<n> when capped is true", "code" : 14832, "ok" : 0 }**   
   c.创建集合的时候Max属性是可选的，文档数量限制是在容量没满的时候进行淘汰，要是满了，就根据容量限制来替换数据。例如上面的例子我们设置文档数量限制是10这样的话在没有达到容量上限的时候，最多只能存储10个文档。  
     
   **2.查看索引**  
   **> db.mycoll.getIndexes()[ { "v" : 1, "key" : { "\_id" : 1 }, "ns" : "test.mycoll", "name" : "\_id\_" }]**  
   在2.2版本以后，创建的Capped Collection 默认在 \_id 字段上创建索引。  
   **3.尝试去更新一个文档**  
     
   **> db.mycoll.find({id:9998}) { "\_id" : ObjectId("5274b477f007ff8bb84f5f67"), "id" : 9998, "name" : "aa" }> db.mycoll.update({id:9998},{$set:{name:"China"}})**  
   **> db.mycoll.find({id:9998}) { "\_id" : ObjectId("5274b477f007ff8bb84f5f67"), "id" : 9998, "name" : "China" } >**  
   **> db.mycoll.update({id:9998},{$set:{name:"HeBeiOfChina"}}) failing update: objects in a capped ns cannot grow >**  
   **注意：**  
   **这里成功的把name属性从aa更新为China，但当我们从China更新到HeBeiOfChina时失败了。失败的原因更新后的文档大小值不能超过原有空间，否则更新失败。但是为什么我们更新为China没有报错呢？这里面就涉及到MongDB的Record Padding和内存对齐问题，为了提高数据的插入效率，减少数据的移动带来的性能下降，每条记录都会有不定长度的预留空间。**  
     
   **4.尝试去删除一个文档**  
     
   **> db.mycoll.remove({id:9988})**  
   **can't remove from a capped collection**  
     
   **使用Capped Collection 不能删除一个文档。**  
     
   **5.删除这个集合**  
   **> db.mycoll.drop() true**  
   **6.普通集合转成固定集合**  
   **> db.createCollection("mycoll") { "ok" : 1 } > db.mycoll.isCapped() false >db.runCommand({convertToCapped:'mycoll',size:10000,max:3}) { "ok" : 1 } > db.mycoll.isCapped() true**  
     
   **首先我们创建了一个普通集合，然后使用命令 db.runCommand({convertToCapped:'mycoll',size:10000,max:3})将其转化为一个Capped Collection**

**7.使用validate()工具来查看Collection使用的存储空间。**

**>db.mycappedcoll.validate()**

**Capped Collection**

固定集合(Capped Collection)是性能出色的有着固定大小的集合,以LRU(Least Recently Used最近最少使用)规则和插入顺序进行age-out(老化移出)处理,自动维护集合中对象的插入顺序,在创建时要预先制定大小,如果空间用完,新添加的对象将会取代集合中最旧的对象,永远保持最新的数据。

[复制代码](javascript:void(0);)

查看集合的状态信息

db.cot1.stats(); //查看集合cot1的状态信息

删掉指定集合

db.collection1.drop(); //删除collection1集合

mongoDB中集合创建默认是隐式创建的,可以使用createCollection显示创建集合

db.createCollection("collect");

删掉指定数据库

db.dropDatabase(); //删除当前数据库

查询所有数据库

show dbs //查询所有数据库

显示当前库中所有集合

show tables; 或者 show collections;

[复制代码](javascript:void(0);)

**固定集合的功能特点**

可以插入及更新,但更新不能超出collection的大小,否则更新失败,不允许删除,但是可以调用drop()删除集合中的所有行,但是drop后需要显式地重建集合.在32位机子上一个cappped collection的最大值约为482.5M,64位上只受系统文件大小的限制.

**固定集合属性及用法:**

属性1:对固定集合进行插入速度极快

属性2:按照插入顺序的查询输出速度极快

属性3:能够在插入最新数据时,淘汰最早的数据

用法1:储存日志信息

用法2:缓存一些少量的文档

**创建固定集合**

不像普通集合,固定集合需要显示的创建使用createCollection命令

eg. db.createCollection("collect",{capped:true, size:10000});

创建一个集合为collect的固定集合,大小为10000字节,还可以指定文档个数,加上Max:100属性.

注意:指定文档上限,必须指定大小,文档限制是在容量没满时进行淘汰,要是满了,就根据容量限制来进行淘汰.

[复制代码](javascript:void(0);)

可以再创建capped collection时指定collection中能够存放的最大文档数目,但这是要指定size,因为总是先检查size后检查maxRowNumber.可以使用validate()查看一个collection已经使用了多少空间,从而决定size设为多大.

> db.createCollection("mycappedcoll",{capped:true,size:10000,max:100})

{ "ok" : 1 }

> db.mycappedcoll.validate();

创建collection时还有一个参数”autoIndexID”,值可以为”true”和”false”,决定是否需要在”\_id”上创建索引,例如

> db.createCollection("mycappedcoll",{capped:true,size:10000,max:100,autoIndexId:false})

默认情况下对一般的collection是创建索引的,但不会对capped collection创建.capped collection不能被Shard,这也是有道理的,一个经常被刷新且大小固定的表,做Sharding(分片)也没太大意义.

[复制代码](javascript:void(0);)

**转换集合**

把普通的集合转换成固定集合需要使用convertTocapped命令

db.runCommand( { convertTocapped："test",size:10000 } );

把test普遍集合转换为固定集合,大小为10000字节.

自然排序

固定集合文档按照插入顺序储存的,默认情况下查询就是按照插入顺序返回的,也可以使用$natural调整返回顺序.

db.mycappedcolt.find().sort( {"$natural":1} );

参数1表示默认顺序,-1则相反.

判断集合是否为固定集合

db.colt.isCapped();

查看集合状态信息

db.colt.stats();

**注意事项**

1.可以在创建capped collection时指定collection中能够存放的最大文档数。但这时也要指定size，因为总是先检查size 后检查maxRowNumber。可以使用validate()查看一个collection 已经使用了多少空间，从而决定size设为多大。

db.createCollection("mycoll", {capped:true, size:100000, max:100});

db.mycoll.validate();

max=1时会往collection中存放尽量多的documents。  
 2.上述的 createCollection 函数也可以用来创建一般的 collection，还有一个参数"autoIndexID"，值可以为"true"和"false"来决定是否需要在"\_id"字段上自动创建索引，如： db.createCollection("mycoll", {size:10000000, autoIndexId:false})。 默认情况下对一般的collection是创建索引的，但不会对capped collection创建。

## 第6章 GridFS存储文件

**GridFS：基于MongoDB的分布式文件存储系统**

### 6.1 基本原理

GridFS是MongoDB之上的分布式文件系统，其利用了MongoDB的分布式存储机制并通过MongoDB来存储文件数据和文件元数据，**兼具文档型数据库和文件系统的优势**。GridFS是当前大数据潮流和复杂数据分析需求的产物。

简单地来说，GridFS通过将文件数据和文件元数据保存在MongoDB里来实现文件系统，通过复制(Replication)来应对故障切换、数据集成，还可以用来做读扩展、热备份或者作为离线批处理的数据源，通过分片来实现自动切分数据，实现大数据存储和负载均衡，通过数据库对集合中文档的管理和查询（包括MapReduce）实现轻量级文件系统接口和搜索与分析。

GridFS的一个基本思想就是可以将大文件分成很多块，每一块作为一个单独的文档存储，这样就能存储大文件了。由于MongoDB支持在文档中存储二进制数据，可以最大限度减小块的存储开销。GridFS使用MongoDB的复制、分片等机制来实现分布式文件存储，使用MongoDB进行管理和复杂分析。

GridFS使用两个文档来存储文件，一个用来存储文件本身的块，另外一个用来存储分块的信息和文件的元数据，默认对应的集合分别为fs.chunks和fs.files。

chunks集合：

{

"\_id" : <string>，

"files\_id" : <string>，

"n" : <num>，

"data" : <binary>

}

块集合中文档包含以下属性：

chunks.\_id: 块ID。

chunks.files\_id: 对应files集合中文档的\_id。

chunks.n: 块的编号，由GridFS管理，从0开始。

chunks.data: 文件数据，是 BSON二进制类型。

chunks集合使用files\_id 和 n作为混合索引。 files集合：

{

"\_id" : <ObjectID>，

"length" : <num>，

"chunkSize" : <num>

"uploadDate" : <timestamp>

"md5" : <hash>

"filename" : <string>，

"contentType" : <string>，

"aliases" : <string array>，

"metadata" : <dataObject>，

}

files集合中的文档包含以下属性，应用还可以创建额外任意的属性： files.\_id: 唯一的文件标识，MongoDB的默认值是BSON ObjectID。 files.length: 文件的字节数大小。 files.chunkSize: 每个块的大小，默认为256KB，GridFS根据这个值将文件分成多个块。 files.uploadDate: GridFS第一次存储此文件的时间，类型为ISODate。 files.md5: 文件的md5散列值，是字符串。 files.filename: 可选。人类可读的文件名。 files.contentType: 可选。合法的文件MIME类型。 files.aliases: 可选。别名的字符串数组。 files.metadata: 可选。自定义存储的文件元数据。

可以通过mongofiles工具或者MongoDB驱动程序来使用GridFS。GridFS主要提供5种操作接口：

list: 获取文件列表

get: 获取文件

put: 写入文件

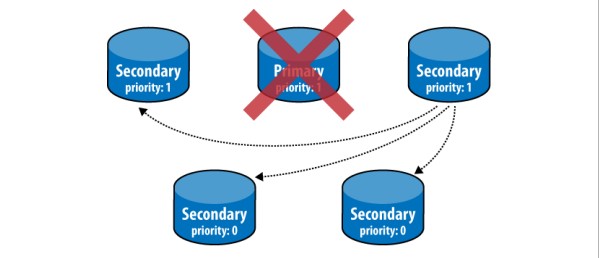
search: 根据文件名搜索文件

delete: 删除文件

### 6.2 主要技术和架构

**(1)复制**

分布式存储系统一个关键的功能就是复制(Replication)，通过复制来应对故障切换、数据集成，还可以用来做读扩展、热备份或者作为离线批处理的数据源。总的来说，MongoDB的复制至少需要两个服务器或者节点。其中一个是注解点，负责处理客户端请求，其他的都是从节点，负责映射主节点的数据。主节点记录在其上执行的所有操作。从节点定期轮询主节点获得这些操作，然后对自己的数据副本执行这些操作。由于和主节点执行了相同的操作，从节点就能保持与主节点的数据同步。



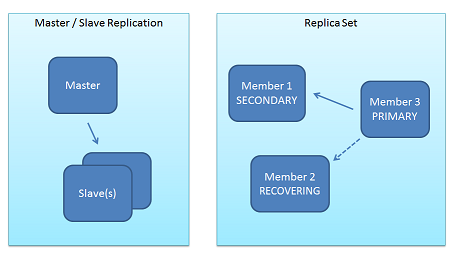
主节点的操作记录被称为oplog，其存储在一个特殊的数据库中，叫做local。oplog就在其中的oplog.$main集合里面。oplog中的每一个文档都代表主节点上执行的一个操作。文档包含的键如下：

ts: 操作的时间戳。时间戳是一种内部类型，用以跟踪操作执行的时间。由4字节的时间戳和4字节的递增计数器构成。

op: 操作类型，只有1字节代码。

ns: 执行操作的命名空间（集合名）。

o: 进一步指定要执行的操作的文档。对插入来说，就是要插入的文档。

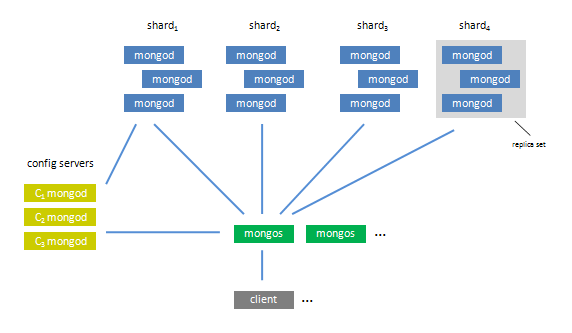


MongoDB有两种复制方式，分别为主从复制和副本集。主从复制是最常用的方式，基本的设置方式就是建立一个主节点和一个或多个从节点，每个从节点要知道主节点的地址。所有的从节点都从主节点复制内容。目前还没有能够从从节点复制的机制（菊花链），原因就是从节点并不保存自己的oplog。副本集中的活跃节点对应主节点，当活跃节点故障时，新的活跃节点由副本集中的大多数选举产生。

从节点的主要作用是作为故障恢复机制，以防主节点数据丢失或者停止服务；其他用途包括：将查询放在从节点上、用从节点做数据处理——以减轻主节点负载、提高系统性能。

**(2)分片**

分片(Sharding)是指将数据拆分，将其分散存放在不同的机器上的过程。将数据分散到不同机器上，不需要功能强大的大型计算机就可以存储更多的数据，处理更大的负载。MongoDB支持自动分片，可以摆脱手动分片的管理困扰。集群自动切分数据，做负载均衡。MongoDB分片的基本思想就是将集合切分成小块。这些块分散到若干片里面，每个片只负责总数据的一部分。应用程序不必知道哪片对应哪些数据，甚至不需要知道数据已经被拆分了，所以在分片之前要运行一个路由进程，该进程为mongos。这个路由器知道所有数据的存放位置，所以应用可以连接它来正常发送请求。对应用来说，它仅知道连接了一个普通的mongod，也就是说分片对于应用来说是透明的。路由器知道数据和片的对应关系，能够转发请求到正确的片上。如果请求有了回应，路由器将其手机起来回送给应用。



设置分片时，需要从集合里面选一个键，用该键的值作为数据划分的依据。这个键称为片键，片键的选择决定了插入操作在片之间的分布。不论片键随机跳跃还是稳定增加，片键的变化至关重要。

**(3)查询和聚合**

因为GridFS文件的元数据存储在files集合中，因此GridFS可以非常方便地进行文件管理，比如根据文件名、上传时间、文件大小或者自定义的文件元数据进行查询，还可以利用MapReduce做复杂数据分析。这是GridFS把传统文件系统和数据库相结合得到的众多好处之一。

### 6.3 对比传统文件系统的优势

分布式：GridFS是基于MongoDB的分布式文件系统，可以直接使用MongoDB Replication和Sharding机制，数据可靠性和水平扩展性都得到保证。GridFS不产生磁盘碎片，因为MongoDB分配数据文件空间时以2GB为一块。

MapReduce： 可以进行复杂管理和查询分析。

索引和缓存：元数据存储在MongoDB中，非常方便索引。并且可以对文件和文件元数据进行索引，能提高系统效率。

checksum：GridFS会为文件产生散列值，可用于校验文件以检查完整性。

开发者友好： 利用Grid可以简化需求，减小开发成本。要是已经用了MongoDB，GridFS就可以不需要使用独立文件存储架构，并且使代码和数据真正分离，方便管理（尤其是对云计算平台）。

其他：GridFS可以避免用于存储用户上传内容的文件系统出现的某些问题。例如，GridFS在同一个目录下放置大量的文件是没有任何问题的。GridFS不产生磁盘碎片，因为MongoDB分配数据文件空间时以2GB为一块。

### 6.4 小结

MongoDB是一个优秀的分布式基于文档的NoSQL数据库，正是基于MongoDB，GridFS才能将数据库与文件系统完美地结合在一起，兼具二者的优势。使用GridFS，开发者能在不怎么损失性能的情况下对系统进行扩展，并且极大地减小了开发成本和系统维护成本。可以预计，GridFS或者类似的分布式文件系统在未来一定会得到广泛地应用。事实上，以CloudFoundry为代表的云平台已经把MongoDB作为首选的数据库和文件管理系统了。

#### ****6.4.1 GridFS****

GridFS是一种在MongoDB中存储大二进制文件的机制,使用GridFS原因有以下几种:

储存巨大的文件,比如视频、高清图片等.利用GridFS可以简化需求.

GridFS会直接利用已经建立的复制或分片机制,故障恢复和扩展都很容易.

GridFS可以避免用户上传内容的文件系统出现问题.

GridFS不产生磁盘碎片

#### ****6.4.2 GridFS存储数据****

GrdiFS使用两个表来存储数据:

files 包含元数据对象

chunks 包含其他一些相关信息的二进制块.

为了使多个GridFS命名为一个单一的数据库,文件和块都有一个前缀,默认情况下,前缀是fs,所以任何默认的GridFS存储将包括命名空间fs.files和fs.chunks。

各种第三方语言可以更改其前缀

#### ****6.4.3 使用GridFS mongofiles****

mongofiles是从命令行操作GridFS的一种工具

四个命令:put(存储)、get(取得/下载)、list(列表)、delete(删除)

[复制代码](javascript:void(0);)

eg. ./mongofiles put testfile.zip

./mongofiles list

./mongofiles get testfile.zip

./mongofiles delete testfile.zip

验证md5

mg5sum testfiles.zip

此时登录mongo,show tables可以看到新增2个文件fs.files和fs.chunks.

查看fs.files中的内容

db.fs.files.find();

fs.files 中存储的是一些基础的元数据信息.

db.fs.chunks.find();

fs.chunks 中存储的是一些实际的内容数据信息.

[复制代码](javascript:void(0);)

## 第7章 MapReduce统计

### 7.1 MapReduce的介绍

在关系型数据库中可以对数据进行分组（group by）,如果在MongoDB中也需要分组怎么办？

MongoDB提供了MapReduce统计，MapReduce相当于SQL中的group by,所以在MongoDB中对数据进行并行统计，可以使用MapReduce统计非常容易的实现。

MapReduce 是目前最流行和被普遍研究的海量数据处理方法。它是Google公司的核心模型，用于大规模数据集(大于1TB）的并行计算。“映射（Map)”与“化简（Reduce)”的概念是它们的主要思想，都是从函数式编程语言借来的。MapReduce将负责的运行于大规模集群上的并行计算过程高度地抽象为两个函数(Map和Reduce),利用一个输入<key,value>集合来产生一个输出地<key,value>对集合，极大地方便了编程地人员，在不了解分布式并行编程地情况下，它们也能将自己地程序运行在分布式系统上。

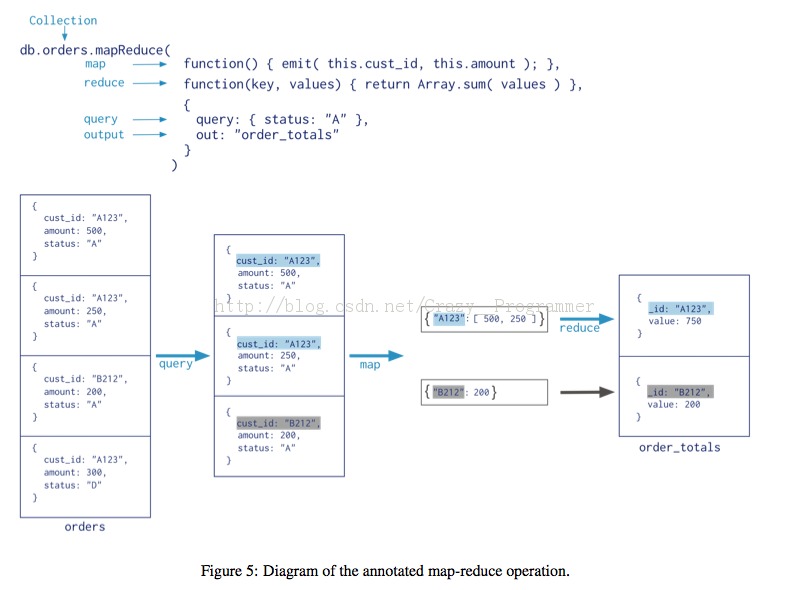
MapReduce在执行时先指定一个Map(映射）函数，把输入<key,value>对映射成一组新的<key,value>对，经过一定处理后交给 Reduce，Reduce对相同key下的所有value处理后再输出<key,value>对作为最终的结果。

MapReduce计算模型非常适合在大量计算机组成的大规模集群上并行运行。在采用Map/Reduce模式的分布式系统上，每一个Map任务和每一个Reduce任务均可以同时运行于一个单独的计算节点上，可想而知其处理效率是很高的。

### 7.2 Map/Reduce过程

MongoDB中的Map/Reduce对于批量处理数据进行聚合操作是非常有用的。在思想上它跟Hadoop一样，从一个单一集合中输入数据，然后将结果输出到一个集合中。通常在使用类似SQL中Group By操作时，Map/Reduce会是一个好的工具。

看下面一个实例：



a.在orders集合上进行MapReduce操作，首先使用query进行“过滤”操作，选择出 status为’A’的所有文档。

b.在选择后的每个文档上执行map操作，在map操作的时候将当前文档的this.cust\_id,this.amount分别作为键值发射出去，经过map操作后，相同键的文档的值被放到一起组成一个数组。

c.如果一个键有多个值的话，进行reduce的操作，在进行reduce 操作的时候将所有的值进行累加

如果一个健只有一个值的话就直接输出到结果集合

d.Reduce完后将结果输出到预先定义好的结果集合中，即order\_totals集合。

### 7.3 Map/Reduce 指令

Map/Rudece是通过数据库指令调用的，指令原型如下：

db.runCommand(

{

mapReduce: <collection]]

>

,

map: <function]]

>

,

reduce: <function]]

>

,

out: <output]]

>

,

query: <document]]

>

,

sort: <document]]

>

,

limit: <number]]

>

,

finalize: <function]]

>

,

scope: <document]]

>

,

jsMode: <boolean]]

>

,

verbose: <boolean]]

>

} )

参数说明：

mapReduce：要执行Map/Reduce集合的名字

map:map 函数 (下面会详细介绍）

reduce:reduce函数(下面会详细介绍）

out:存放结果的集合 (下面会详细介绍）

query:设置查询条件 <可选>

sort: 按某个键来排序 <可选>

limit:指明从集合检索文档个数的最大值 <可选>

finalize:对reduce结果做进一步处理 <可选>

scope:指明通过map/reduce/finalize可以访问到的变量 <可选>

jsMode:指明Map/Reduce执行过程中文档保持JSON状态 <可选>

verbose:提供关于任务执行的统计数据 <可选>

例如：

var mapFunction = function() { ... };

var reduceFunction = function(key, values) { ... };

db.runCommand(

{

mapReduce: 'orders',

map: mapFunction,

reduce: reduceFunction,

out: { merge: 'map\_reduce\_results', db: 'test' },

query: { ord\_date: { $gt: new Date('01/01/2012') } }

} )

### 7.4 MapReduce参数详细说明

对Map/Reduce指令中的几个重要参数做一下详细的说明，以便我们能够写出正确的Map/Reduce程序。主要介绍一下Map、Reduce和Out三个参数使用，它们也是必选的参数。

**1.Map函数**

Map函数通过变量this来检验当前考察的对象。一个Map函数会通过任意多次函数调用emit(key,value)来将数据送入reducer中。大多数情况下，对于每个文档都只会发送一次，但是在有些情况下需要发送多次，例如下面这个Map程序，整个文档被调用的次数取决于items数组中元素的个数.

**function**() {

**this**.items.forEach(**function**(item){ emit(item.sku, 1); }); }

有些情况下也会发送0次，例如：

**function**() {

**if** (**this**.status == 'A')

emit(**this**.cust\_id, 1);

}

每一次发送的数据都被限制到最大文档大小的50%(比如MongoDB1.6.x中是4MB, MngoDB1.8.x中是8MB)。

**2.Reduce函数**

当运行Map/Reduce时，Reduce函数将收到一个发送值构成的数组并且要把它们简化到单个值。因为针对一个键值的Reduce函数可能会被调用到好多次，Reduce函数返回的对象结构要与Map函数发送的值的结构必须时完全相同。我们可以用一个简单的例子来说明这一情况。

假设对一个代表用户评价的文档构成的集合进行迭代，典型的文档内容如下：

｛

name:”jones”,

likes:20,

text:”hello world!”

}

我们想利用Map/Reduce来统计每个用户的评论数，并且计算全部用户评价中“like”的总数。为了达到这个目的，首先写一个如下的Map函数：

func () {

emit( this.username, {count:1,likes:this.likes});

}

这个函数实际上指明了要以username来分组，并且针对count和likes字段进行聚合运算。当Map/Reduce实际运行的时候，一个由用户构成的数组将发送给Reduce函数，这个就是为什么Reduce函数总是用来处理数组。下面就是一个Reduce函数的例子。

func(key,values) {

var result = {count:0, likes:0}

values.forEach(function(value)) {

result.count += value.count;

result.likes += value.likes;

});

return result;

}

**注意：**

a.结果文档和Map函数所传送的数据拥有相同的结构。

即：reduce(key, [ C, reduce(key, [ A, B ]) ] ) == reduce( key, [ C, A, B ] )

这是非常重要的，因为当Reduce函数作用于某个key值时，它并不保证会对这个key值（这里是username)的每一个value进行操作。事实上，Reduce函数不得不运行多次。比如，当处理评论集合时，Map函数可能遇到来自”jones”的10条评论，它会把这些评论传送给Reduce函数，得到如下聚集结果：

｛count:10,likes:247}

然后，Map函数又遇到一个来自”jones”的评论，此时，这值必须被重新考虑来修改聚合结果。如果遇到新的评论为：｛count:1,likes:5}

那么reduce函数将被这样调用：

reduce{“jones”,[{count:10,likes:247},{count:1,likes:5}]}

最后的结果将会是上面两个值的结合：｛count:11,likes:252}

只有你理解对一个key值Reduce函数可能被多次调用，就容易理解为什么这个函数必须饭后一个和Map函数发送的值具有相同的结构了。

b.数组中元素的顺序不能影响结果的输出

reduce( key, [ A, B ] ) == reduce( key, [ B, A ] )

**3.Out函数**

在MongoDB1.8之前的版本，如果你没有指定out的值，那么结果将会被放到一个临时集合中，集合的名字在输出指令中指定，否则，你可以指定一个集合的名字作为out的选项，而结果将会被存储到你指定的集合中。

对于MongoDB1.8以及以后的版本，输出选项改变了。Map/Reduce 不再产生临时集合，你必须为out指定一个值，设置out指令如下：

默认输出第一个集合中，如果该集合不存在的话，会自动的创建

out: <collectionName]]

>

除此之外，也可以加一些选项，这只适用于集合已经存在的情况

out: { <action>: <collectionName]]

>

[, db: <dbName]]

>

]

[, sharded: <**boolean**]]

>

]

[, nonAtomic: <**boolean**]]

>

] }

a.参数Action说明：

Action可以为 replace(默认)、merge、reduce

{replace:”collectionName"}:输出结果将被插入到一个集合中，并且会自动替换掉现有的同名集合。该选项为默认的。

{merge:”collectionName"}:这个选项将会把新的数据连接到旧的输出结合中。换句话说，如果在结果集和旧集合中存在相同键值，那么新的键将会被覆盖掉。

{reduce:”collectionName"}:如果具有某个键值的文档同时存在于结果集和旧集合中，那么一个Reduce操作（利用特定的reduce函数)将作用于这个两个值，并且结果将会被写到输出集合中。如果指定了finalize函数，那么当Reduce结束后它将被执行。

b. 参数db: 指明接收输出结果的数据库名称

out:{replace:”collectionName",db:”otherDB"}

c. 参数shard: {shared:true}:适用于MongoDB1.9及以上的版本。如果设置为true，并且将输出模式设置为输出得到集合中，那么输出的结合的每个文档就将用\_id字符进行切分。

d. 参数 inlne: {inline:1}借助于该选项，将不会创建集合，并且整个Map/Reduce的操作将会在内存中进行。同样，Map/Reduce的结果将会被返回到结果对象中。注意，这个选项只有在结果集的单个文档大小在16MB限制范围内时才有效。

db.users.mapReduce(map,reduce,{out:{inline:1}});

**4 Finalize函数**

finalize函数可能会在Reduce函数结束之后运行，这个函数是可选的，对于很多Map/Reduce任务来说不是必需的。finalize函数接收一个key和一个value，返回一个最终的value.

function finalize(key,value) -> final\_value

针对一个对象你的Reduce函数可能被调用了多次。当最后只需针对一个对象进行一次操作时可以使用finalize函数，比如计算平均值。

**5.jsMode标识**

对于MongoDB2.0及以上的版本，通常Map/Reduce的执行遵循下面两个步骤：

a.从BSON转化为JSON,执行Map过程，将JSON转化为BOSN

b.从BSON转化为JSON,执行Reduce过程，将JSON转化为BSON

因此，需要多次转化格式，但是可以利用临时集合在Map阶段处理很大的数据集。为了节省时间，可以利用{jsMode:ture}使Map/Reduce的执行保持在JSON状态。遵循如下两个步骤：

a.从BSON转化为JSON,执行Map过程

b.执行Reduce过程，从 JSON转化为BSON

这样，执行时间可以大大减小，但需要注意，jsMode 受到JSON堆大小和独立主键最大500KB的限制。因此，对于较大的任务jsMode并不适用，在这种情况下会转变为通常的模式。

### 7.5 MapReduce 在分片集合上的使用

Map-reduce可以在分片集合上使用，而且可以将分片集合作为输入和输出。

当一个分片集合作为其输入的时候，mongos进程会自动的向每个分片来分发map和reduce 作业，然后mongos进程会等待所有分片完成自己的作业。

如果map reduce 的out字段上有sharded值的话，mongodb会将结果集合进行分片，并选择\_id作为其片键。

### 7.6 编写MapReduce程序

所有的map-reduce函数都是用JavaScript书写，然后在mongod实例进程上运行。

Map-Reduce是进行大容量数据处理的范式，对于Map-Reduce操作，MongoDB提供了MapReduce的数据库命令。在进行map-reduce操作的时候，MongoDB会将满足查询条件的文档进行map所定义的操作，map函数会产生( emit)键值型的数据。

如果某个键所对应的值有多个的话，会进行reduce的操作，最后将结果保存到一个集合中。通过定义一个finalize函数可以对reduce的结果做进一步的处理，比如：进行投影或者规范化输出、进一步的计算等。

看下面例子：

1.首先在orders集合中插入5条待检索数据

>db.order.insert({ "cust\_id" : "1", "ord\_date" : new Date("2013-11-13T16:00:00Z"), "status" : "A", "price" : 25, "items" : [ { "sku" : "mmm", "qty" : 5, "price" : 2.5 }, { "sku" : "nnn", "qty" : 5, "price" : 2.5 } ] })

>db.order.find()

{ "\_id" : ObjectId("528312e716b20807b2152db5"), "cust\_id" : "1", "ord\_date" : ISODate("2013-11-13T16:00:00Z"), "status" : "A", "price" : 25, "items" : [ { "sku" : "mmm", "qty" : 5, "price" : 2.5 }, { "sku" : "nnn", "qty" : 5, "price" : 2.5 } ] }

{ "\_id" : ObjectId("528312f716b20807b2152db6"), "cust\_id" : "2", "ord\_date" : ISODate("2013-11-13T16:00:00Z"), "status" : "A", "price" : 25, "items" : [ { "sku" : "mmm", "qty" : 5, "price" : 2.5 }, { "sku" : "nnn", "qty" : 5, "price" : 2.5 } ] }

{ "\_id" : ObjectId("5283130816b20807b2152db7"), "cust\_id" : "3", "ord\_date" : ISODate("2013-11-13T16:00:00Z"), "status" : "A", "price" : 25, "items" : [ { "sku" : "mmm", "qty" : 5, "price" : 2.5 }, { "sku" : "nnn", "qty" : 5, "price" : 2.5 } ] }

{ "\_id" : ObjectId("5283132c16b20807b2152db8"), "cust\_id" : "3", "ord\_date" : ISODate("2013-11-13T16:00:00Z"), "status" : "A", "price" : 30, "items" : [ { "sku" : "mmm", "qty" : 6, "price" : 2.5 }, { "sku" : "nnn", "qty" : 6, "price" : 2.5 } ] }

{ "\_id" : ObjectId("5283134d16b20807b2152db9"), "cust\_id" : "2", "ord\_date" : ISODate("2013-11-13T16:00:00Z"), "status" : "A", "price" : 20, "items" : [ { "sku" : "mmm", "qty" : 4, "price" : 2.5 }, { "sku" : "nnn", "qty" : 4, "price" : 2.5 } ] }

2.统计每个顾客消费总额

a.编写map函数

> var mapFunc = function() {

... emit(this.cust\_id,this.price);

... };

b.编写reduce函数

> var reduceFunc = function(key,values) {

... return Array.sum(values);

... };

c.执行mapreduce

> db.runCommand({mapreduce:'order',map:mapFunc,reduce:reduceFunc,out:{replace:'map\_reduce\_result'}})

{

"result" : "map\_reduce\_result",

"timeMillis" : 35,

"counts" : {

"input" : 5,

"emit" : 5,

"reduce" : 2,

"output" : 3

},

"ok" : 1

}

也可以这样

> db.order.mapReduce(mapFunc,reduceFunc,{out:{replace:'map\_result\_result'}})

{

"result" : "map\_result\_result",

"timeMillis" : 19,

"counts" : {

"input" : 5,

"emit" : 5,

"reduce" : 2,

"output" : 3

},

"ok" : 1,

}

注意：db.XXXmapReduce是对数据库命令db.runCommand的一个包装，前者实际上是调用后者来实现的

d.查看结果

> show collections

map\_reduce\_result

order

system.indexes

> db.map\_reduce\_result.find()

{ "\_id" : "1", "value" : 25 }

{ "\_id" : "2", "value" : 45 }

{ "\_id" : "3", "value" : 55 }

3.计算每种商品的购买总数和平均每次购买数量

a.编写map函数

> var mapFunc = function() {

... for(var idx=0;idx<this.items.length;idx++) {

... var key = this.items[idx].sku;

... var value = {

... count:1,

... qty:this.items[idx].qty

... };

... emit(key,value);

... }};

b.编写reduce函数

> var reduceFunc = function(key,values) {

...

... reduceVal = {count:0,qty:0};

...

... for(var idx=0;idx<values.length;idx++) {

... reduceVal.count += values[idx].count;

... reduceVal.qty += values[idx].qty;

... }

... return reduceVal;

... };

c.编写finalize函数

> var finalizeFunc = function(key,reduceVal) {

… reduce Val.avg = reduceVal.qty/reduceVal.count;

... return reduceVal;

... }

d.执行mapreduce函数

db.order.mapReduce(mapFunc,reduceFunc,

{

out: {merge:"map\_reduce\_result\_2"},

query:{ ord\_date:{$gt:new Date('01/01/2000')}},

finalize:finalizeFunc

}

)

{

"result" : "map\_reduce\_result\_2",

"timeMillis" : 20,

"counts" : {

"input" : 5,

"emit" : 10,

"reduce" : 2,

"output" : 2

},

"ok" : 1,

}

分析：

从执行结果可以发现，有5个文档执行MapReduce的操作，Map操作发生了10次（每个文档产生了两次，印证了上一节所说的一个文档可以进行一次或多次Map操作），Reduce操作发生了2次，最终输出了2个文档。

同样也可以使用runCommand来执行MapReduce操作：

db.runCommand({mapreduce:'order’,

map:mapFunc,

reduce:reduceFunc,

out:{replace:'map\_reduce\_result\_2'},

query:{ord\_date:{$gt:new Date('01/01/2000')}},

finalize:finalizeFunc})

d.查看结果

> db.map\_reduce\_result\_2.find()

{ "\_id" : "mmm", "value" : { "count" : 5, "qty" : 25, "avg" : 5 } }

{ "\_id" : "nnn", "value" : { "count" : 5, "qty" : 25, "avg" : 5 } }

注意：

1.reduce函数返回对象的键值结构必须要和map函数传出对象值的键值类型相同，这样才能保证多次调用Reduce 函数成为可能。否则，会出现奇怪的很难调试的错误。例如在第二个例子中reduce返回的对象的类型为 “ {count:0,qty:0}” ，它和map函数的传出对象的值类型是 “var value = { count:1, qty:this.items[idx].qty }”类型是一致的。

2.使用finalize函数可以进一步对Value的值进行操作,例如可以这样修改上面例子中的finalizeFunc：

> var finalizeFunc = function(key,reduceVal) {

… reduceVal.avg = reduceVal.qty/reduceVal.count;

… reduceVal.tip = “ MapReduce”;

... return reduceVal;

… }

此时的结果显示为：

> db.map\_reduce\_result\_2.find()

{ "\_id" : "mmm", "value" : { "count" : 5, "qty" : 25, "avg" : 5,”tip”: ”MapReduce" } }

{ "\_id" : "nnn", "value" : { "count" : 5, "qty" : 25, "avg" : 5 ,”tip”: "MapReduce"} }

3.进行MapReduce时可以使用qurey对文档进行过滤，只对过滤后的文档进行MapReduce操作，提高MapReduce 执行效率。

4.如果map后的文档中的键只有一个值的时候，不会进行reduce 操作。

5.Reduce 返回的值不能是一个数组，通常是一个对象或者一个数值。

### 7.7 总结

Map-Reduce操作将一个单一集合内文档作为输入，在进行map函数所定义的操作的时候，可以进行任意的排序或者limit文档个数的操作。mapReduce操作 返回的结果可以是一条文档，也可以将其结果写到一个集合里面。 也可以以inline方式返回（结果文档必须要小于BSON文档的大小限制，当前是16M） 如果写到一个集合中，我们就可以在整个集合上进行进一步的聚合操作。

只要理解MapReduce原理，了解MapReduce聚合过程，那么编写MapReduce函数也就没那么困难了。

和聚管道相比，使用map-reduce提供了很大的灵活性，我们可以编写自己的逻辑map和reduce函数，实现特定的功能。但是一般来说，写map-reduce操作 比聚合管道要复杂，而且效率也没有聚合管道高。

## [谷歌技术"三宝"之MapReduce](http://blog.csdn.net/opennaive/article/details/7514146)

1. [MapReduce是干啥的](http://blog.csdn.net/opennaive/article/details/7514146#t0)
2. [例子统计词频](http://blog.csdn.net/opennaive/article/details/7514146#t1)
3. [map函数和reduce函数](http://blog.csdn.net/opennaive/article/details/7514146#t2)
4. [MapReduce是如何工作的](http://blog.csdn.net/opennaive/article/details/7514146#t3)
5. [词频是怎么统计出来的](http://blog.csdn.net/opennaive/article/details/7514146#t4)
6. [用户的权利](http://blog.csdn.net/opennaive/article/details/7514146#t5)
7. [MapReduce的实现](http://blog.csdn.net/opennaive/article/details/7514146#t6)
8. [参考文献](http://blog.csdn.net/opennaive/article/details/7514146#t7)

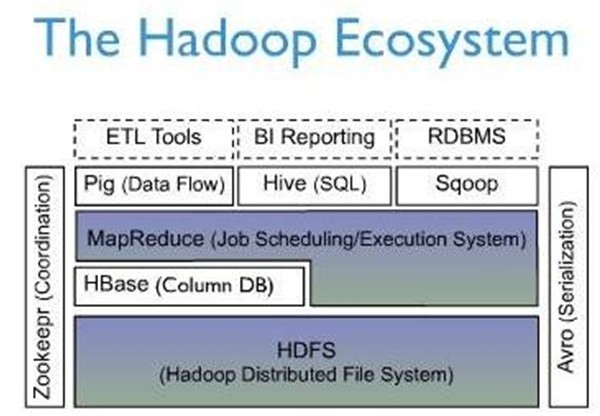
江湖传说永流传：谷歌技术有"三宝"，GFS、MapReduce和大表（BigTable）！

谷歌在03到06年间连续发表了三篇很有影响力的文章，分别是03年SOSP的[GFS](http://blog.csdn.net/opennaive/article/details/7483523)，04年OSDI的MapReduce，和06年OSDI的[BigTable](http://blog.csdn.net/opennaive/article/details/7532589)。SOSP和OSDI都是操作系统领域的顶级会议，在计算机学会推荐会议里属于A类。SOSP在单数年举办，而OSDI在双数年举办。

那么这篇博客就来介绍一下MapReduce。

### 1. MapReduce是干啥的

因为没找到谷歌的示意图，所以我想借用一张Hadoop项目的结构图来说明下MapReduce所处的位置，如下图。



Hadoop实际上就是谷歌三宝的开源实现，Hadoop MapReduce对应Google MapReduce，HBase对应BigTable，HDFS对应GFS。HDFS（或GFS）为上层提供高效的非结构化存储服务，HBase（或BigTable）是提供结构化数据服务的分布式数据库，Hadoop MapReduce（或Google MapReduce）是一种并行计算的编程模型，用于作业调度。

GFS和BigTable已经为我们提供了高性能、高并发的服务，但是并行编程可不是所有程序员都玩得转的活儿，如果我们的应用本身不能并发，那GFS、BigTable也都是没有意义的。MapReduce的伟大之处就在于让不熟悉并行编程的程序员也能充分发挥分布式系统的威力。

简单概括的说，MapReduce是将一个大作业拆分为多个小作业的框架（大作业和小作业应该本质是一样的，只是规模不同），用户需要做的就是决定拆成多少份，以及定义作业本身。

下面用一个贯穿全文的例子来解释MapReduce是如何工作的。

### 2. 例子：统计词频

如果我想统计下过去10年计算机论文出现最多的几个单词，看看大家都在研究些什么，那我收集好论文后，该怎么办呢？

方法一：我可以写一个小程序，把所有论文按顺序遍历一遍，统计每一个遇到的单词的出现次数，最后就可以知道哪几个单词最热门了。

这种方法在数据集比较小时，是非常有效的，而且实现最简单，用来解决这个问题很合适。

方法二：写一个多线程程序，并发遍历论文。

这个问题理论上是可以高度并发的，因为统计一个文件时不会影响统计另一个文件。当我们的机器是多核或者多处理器，方法二肯定比方法一高效。但是写一个多线程程序要比方法一困难多了，我们必须自己同步共享数据，比如要防止两个线程重复统计文件。

方法三：把作业交给多个计算机去完成。

我们可以使用方法一的程序，部署到N台机器上去，然后把论文集分成N份，一台机器跑一个作业。这个方法跑得足够快，但是部署起来很麻烦，我们要人工把程序copy到别的机器，要人工把论文集分开，最痛苦的是还要把N个运行结果进行整合（当然我们也可以再写一个程序）。

方法四：让MapReduce来帮帮我们吧！

MapReduce本质上就是方法三，但是如何拆分文件集，如何copy程序，如何整合结果这些都是框架定义好的。我们只要定义好这个任务（用户程序），其它都交给MapReduce。

在介绍MapReduce如何工作之前，先讲讲两个核心函数map和reduce以及MapReduce的伪代码。

### 3. map函数和reduce函数

map函数和reduce函数是交给用户实现的，这两个函数定义了任务本身。

* map函数：接受一个键值对（key-value pair），产生一组中间键值对。MapReduce框架会将map函数产生的中间键值对里键相同的值传递给一个reduce函数。
* reduce函数：接受一个键，以及相关的一组值，将这组值进行合并产生一组规模更小的值（通常只有一个或零个值）。

统计词频的MapReduce函数的核心代码非常简短，主要就是实现这两个函数。

**[plain]** [view plaincopyprint?](http://blog.csdn.net/opennaive/article/details/7514146)

1. map(String key, String value):
2. // key: document name
3. // value: document contents
4. for each word w in value:
5. EmitIntermediate(w, "1");
6. reduce(String key, Iterator values):
7. // key: a word
8. // values: a list of counts
9. int result = 0;
10. for each v in values:
11. result += ParseInt(v);
12. Emit(AsString(result));

map(String key, String value):

// key: document name

// value: document contents

for each word w in value:

EmitIntermediate(w, "1");

reduce(String key, Iterator values):

// key: a word

// values: a list of counts

int result = 0;

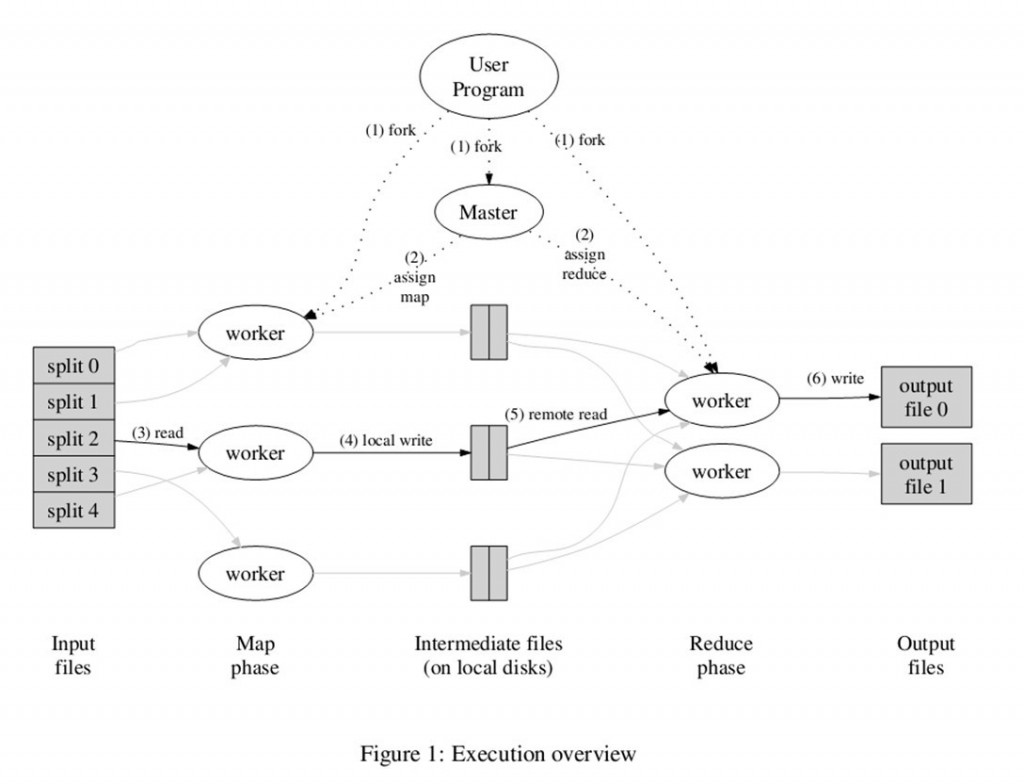
for each v in values:

result += ParseInt(v);

Emit(AsString(result));

在统计词频的例子里，map函数接受的键是文件名，值是文件的内容，map逐个遍历单词，每遇到一个单词w，就产生一个中间键值对<w, "1">，这表示单词w咱又找到了一个；MapReduce将键相同（都是单词w）的键值对传给reduce函数，这样reduce函数接受的键就是单词w，值是一串"1"（最基本的实现是这样，但可以优化），个数等于键为w的键值对的个数，然后将这些“1”累加就得到单词w的出现次数。最后这些单词的出现次数会被写到用户定义的位置，存储在底层的分布式存储系统（GFS或HDFS）。

### 4. MapReduce是如何工作的



上图是论文里给出的流程图。一切都是从最上方的user program开始的，user program链接了MapReduce库，实现了最基本的Map函数和Reduce函数。图中执行的顺序都用数字标记了。

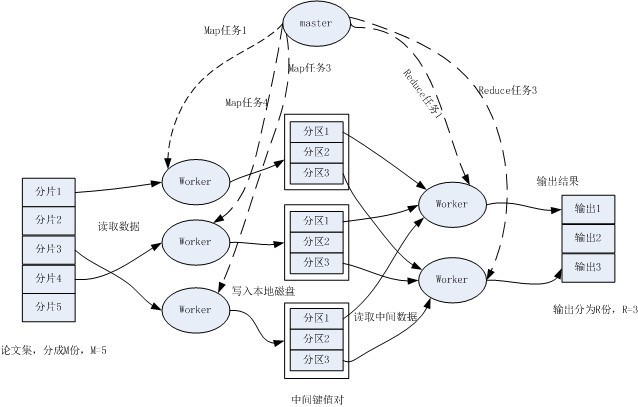
1. MapReduce库先把user program的输入文件划分为M份（M为用户定义），每一份通常有16MB到64MB，如图左方所示分成了split0~4；然后使用fork将用户进程拷贝到集群内其它机器上。
2. user program的副本中有一个称为master，其余称为worker，master是负责调度的，为空闲worker分配作业（Map作业或者Reduce作业），worker的数量也是可以由用户指定的。
3. 被分配了Map作业的worker，开始读取对应分片的输入数据，Map作业数量是由M决定的，和split一一对应；Map作业从输入数据中抽取出键值对，每一个键值对都作为参数传递给map函数，map函数产生的中间键值对被缓存在内存中。
4. 缓存的中间键值对会被定期写入本地磁盘，而且被分为R个区，R的大小是由用户定义的，将来每个区会对应一个Reduce作业；这些中间键值对的位置会被通报给master，master负责将信息转发给Reduce worker。
5. master通知分配了Reduce作业的worker它负责的分区在什么位置（肯定不止一个地方，每个Map作业产生的中间键值对都可能映射到所有R个不同分区），当Reduce worker把所有它负责的中间键值对都读过来后，先对它们进行排序，使得相同键的键值对聚集在一起。因为不同的键可能会映射到同一个分区也就是同一个Reduce作业（谁让分区少呢），所以排序是必须的。
6. reduce worker遍历排序后的中间键值对，对于每个唯一的键，都将键与关联的值传递给reduce函数，reduce函数产生的输出会添加到这个分区的输出文件中。
7. 当所有的Map和Reduce作业都完成了，master唤醒正版的user program，MapReduce函数调用返回user program的代码。

所有执行完毕后，MapReduce输出放在了R个分区的输出文件中（分别对应一个Reduce作业）。用户通常并不需要合并这R个文件，而是将其作为输入交给另一个MapReduce程序处理。整个过程中，输入数据是来自底层分布式文件系统（GFS）的，中间数据是放在本地文件系统的，最终输出数据是写入底层分布式文件系统（GFS）的。而且我们要注意Map/Reduce作业和map/reduce函数的区别：Map作业处理一个输入数据的分片，可能需要调用多次map函数来处理每个输入键值对；Reduce作业处理一个分区的中间键值对，期间要对每个不同的键调用一次reduce函数，Reduce作业最终也对应一个输出文件。

我更喜欢把流程分为三个阶段。第一阶段是准备阶段，包括1、2，主角是MapReduce库，完成拆分作业和拷贝用户程序等任务；第二阶段是运行阶段，包括3、4、5、6，主角是用户定义的map和reduce函数，每个小作业都独立运行着；第三阶段是扫尾阶段，这时作业已经完成，作业结果被放在输出文件里，就看用户想怎么处理这些输出了。

### 5. 词频是怎么统计出来的

结合第四节，我们就可以知道第三节的代码是如何工作的了。假设咱们定义M=5，R=3，并且有6台机器，一台master。



这幅图描述了MapReduce如何处理词频统计。由于map worker数量不够，首先处理了分片1、3、4，并产生中间键值对；当所有中间值都准备好了，Reduce作业就开始读取对应分区，并输出统计结果。

### 6. 用户的权利

用户最主要的任务是实现map和reduce接口，但还有一些有用的接口是向用户开放的。

* an input reader。这个函数会将输入分为M个部分，并且定义了如何从数据中抽取最初的键值对，比如词频的例子中定义文件名和文件内容是键值对。
* a partition function。这个函数用于将map函数产生的中间键值对映射到一个分区里去，最简单的实现就是将键求哈希再对R取模。
* a compare function。这个函数用于Reduce作业排序，这个函数定义了键的大小关系。
* an output writer。负责将结果写入底层分布式文件系统。
* a combiner function。实际就是reduce函数，这是用于前面提到的优化的，比如统计词频时，如果每个<w, "1">要读一次，因为reduce和map通常不在一台机器，非常浪费时间，所以可以在map执行的地方先运行一次combiner，这样reduce只需要读一次<w, "n">了。
* map和reduce函数就不多说了。

### 7. MapReduce的实现

目前MapReduce已经有多种实现，除了谷歌自己的实现外，还有著名的hadoop，区别是谷歌是c++，而hadoop是用java。另外斯坦福大学实现了一个在多核/多处理器、共享内存环境内运行的MapReduce，称为Phoenix（[介绍](http://www.fumin.me/?p=110)），相关的论文发表在07年的HPCA，是当年的最佳论文哦！

### 参考文献

[1] MapReduce : Simplified Data Processing on Large Clusters. In proceedings of OSDI'04.

[2] Wikipedia. <http://en.wikipedia.org/wiki/Mapreduce>

[3] Phoenix. <http://mapreduce.stanford.edu/>

[4] Evaluating MapReduce for Multi-core and Multiprocessor Systems. In proceedings of HPCA'07.

## [谷歌技术"三宝"之BigTable](http://blog.csdn.net/opennaive/article/details/7532589)

1. [什么是BigTable](http://blog.csdn.net/opennaive/article/details/7532589#t0)
2. [BigTable的数据模型](http://blog.csdn.net/opennaive/article/details/7532589#t1)
3. [支撑技术](http://blog.csdn.net/opennaive/article/details/7532589#t2)
4. [Bigtable集群](http://blog.csdn.net/opennaive/article/details/7532589#t3)
5. [片的定位](http://blog.csdn.net/opennaive/article/details/7532589#t4)
6. [片的存储和访问](http://blog.csdn.net/opennaive/article/details/7532589#t5)
7. [BigTable和GFS的关系](http://blog.csdn.net/opennaive/article/details/7532589#t6)
8. [元数据表的结构](http://blog.csdn.net/opennaive/article/details/7532589#t7)
9. [参考文献](http://blog.csdn.net/opennaive/article/details/7532589#t8)

2006年的OSDI有两篇google的论文，分别是BigTable和Chubby。Chubby是一个分布式锁服务，基于Paxos算法；BigTable是一个用于管理结构化数据的分布式存储系统，构建在GFS、Chubby、SSTable等google技术之上。相当多的google应用使用了BigTable，比如Google Earth和Google Analytics，因此它和[GFS](http://blog.csdn.net/opennaive/article/details/7483523)、[MapReduce](http://blog.csdn.net/opennaive/article/details/7514146)并称为谷歌技术"三宝"。

与GFS和MapReduce的论文相比，我觉得BigTable的论文难懂一些。一方面是因为自己对数据库不太了解，另一方面又是因为对数据库的理解局限于关系型数据库。尝试用关系型数据模型去理解BigTable就容易"走火入魔"。在这里推荐一篇文章（需要翻墙）：[Understanding HBase and BigTable](http://jimbojw.com/wiki/index.php?title=Understanding_Hbase_and_BigTable)，相信这篇文章对理解BigTable/HBase的数据模型有很大帮助。

### 1 什么是BigTable

Bigtable是一个为管理大规模结构化数据而设计的分布式存储系统，可以扩展到PB级数据和上千台服务器。很多google的项目使用Bigtable存储数据，这些应用对Bigtable提出了不同的挑战，比如数据规模的要求、延迟的要求。Bigtable能满足这些多变的要求，为这些产品成功地提供了灵活、高性能的存储解决方案。

Bigtable看起来像一个数据库，采用了很多数据库的实现策略。但是Bigtable并不支持完整的关系型数据模型；而是为客户端提供了一种简单的数据模型，客户端可以动态地控制数据的布局和格式，并且利用底层数据存储的局部性特征。Bigtable将数据统统看成无意义的字节串，客户端需要将结构化和非结构化数据串行化再存入Bigtable。

下文对BigTable的数据模型和基本工作原理进行介绍，而各种优化技术（如压缩、Bloom Filter等）不在讨论范围。

### 2 BigTable的数据模型

Bigtable不是关系型数据库，但是却沿用了很多关系型数据库的术语，像table（表）、row（行）、column（列）等。这容易让读者误入歧途，将其与关系型数据库的概念对应起来，从而难以理解论文。[Understanding HBase and BigTable](http://jimbojw.com/wiki/index.php?title=Understanding_Hbase_and_BigTable)是篇很优秀的文章，可以帮助读者从关系型数据模型的思维定势中走出来。

本质上说，Bigtable是一个键值（key-value）映射。按作者的说法，Bigtable是一个稀疏的，分布式的，持久化的，多维的排序映射。

先来看看多维、排序、映射。Bigtable的键有三维，分别是行键（row key）、列键（column key）和时间戳（timestamp），行键和列键都是字节串，时间戳是64位整型；而值是一个字节串。可以用 **(row:string, column:string, time:int64)→string** 来表示一条键值对记录。

行键可以是任意字节串，通常有10-100字节。行的读写都是原子性的。Bigtable按照行键的字典序存储数据。Bigtable的表会根据行键自动划分为片（tablet），片是负载均衡的单元。最初表都只有一个片，但随着表不断增大，片会自动分裂，片的大小控制在100-200MB。行是表的第一级索引，我们可以把该行的列、时间和值看成一个整体，简化为一维键值映射，类似于：

**[javascript]** [view plaincopyprint?](http://blog.csdn.net/opennaive/article/details/7532589)

1. table{
2. "1" : {sth.},//一行
3. "aaaaa" : {sth.},
4. "aaaab" : {sth.},
5. "xyz" : {sth.},
6. "zzzzz" : {sth.}
7. }

table{

"1" : {sth.},//一行

"aaaaa" : {sth.},

"aaaab" : {sth.},

"xyz" : {sth.},

"zzzzz" : {sth.}

}

列是第二级索引，每行拥有的列是不受限制的，可以随时增加减少。为了方便管理，列被分为多个列族（column family，是访问控制的单元），一个列族里的列一般存储相同类型的数据。一行的列族很少变化，但是列族里的列可以随意添加删除。列键按照family:qualifier格式命名的。这次我们将列拿出来，将时间和值看成一个整体，简化为二维键值映射，类似于：

**[javascript]** [view plaincopyprint?](http://blog.csdn.net/opennaive/article/details/7532589)

1. table{
2. // ...
3. "aaaaa" : { //一行
4. "A:foo" : {sth.},//一列
5. "A:bar" : {sth.},//一列
6. "B:" : {sth.} //一列，列族名为B，但是列名是空字串
7. },
8. "aaaab" : { //一行
9. "A:foo" : {sth.},
10. "B:" : {sth.}
11. },
12. // ...
13. }

table{

// ...

"aaaaa" : { //一行

"A:foo" : {sth.},//一列

"A:bar" : {sth.},//一列

"B:" : {sth.} //一列，列族名为B，但是列名是空字串

},

"aaaab" : { //一行

"A:foo" : {sth.},

"B:" : {sth.}

},

// ...

}

或者可以将列族当作一层新的索引，类似于：

**[javascript]** [view plaincopyprint?](http://blog.csdn.net/opennaive/article/details/7532589)

1. table{
2. // ...
3. "aaaaa" : { //一行
4. "A" : { //列族A
5. "foo" : {sth.}, //一列
6. "bar" : {sth.}
7. },
8. "B" : { //列族B
9. "" : {sth.}
10. }
11. },
12. "aaaab" : { //一行
13. "A" : {
14. "foo" : {sth.},
15. },
16. "B" : {
17. "" : "ocean"
18. }
19. },
20. // ...
21. }

table{

// ...

"aaaaa" : { //一行

"A" : { //列族A

"foo" : {sth.}, //一列

"bar" : {sth.}

},

"B" : { //列族B

"" : {sth.}

}

},

"aaaab" : { //一行

"A" : {

"foo" : {sth.},

},

"B" : {

"" : "ocean"

}

},

// ...

}

时间戳是第三级索引。Bigtable允许保存数据的多个版本，版本区分的依据就是时间戳。时间戳可以由Bigtable赋值，代表数据进入Bigtable的准确时间，也可以由客户端赋值。数据的不同版本按照时间戳降序存储，因此先读到的是最新版本的数据。我们加入时间戳后，就得到了Bigtable的完整数据模型，类似于：

**[javascript]** [view plaincopyprint?](http://blog.csdn.net/opennaive/article/details/7532589)

1. table{
2. // ...
3. "aaaaa" : { //一行
4. "A:foo" : { //一列
5. 15 : "y", //一个版本
6. 4 : "m"
7. },
8. "A:bar" : { //一列
9. 15 : "d",
10. },
11. "B:" : { //一列
12. 6 : "w"
13. 3 : "o"
14. 1 : "w"
15. }
16. },
17. // ...
18. }

table{

// ...

"aaaaa" : { //一行

"A:foo" : { //一列

15 : "y", //一个版本

4 : "m"

},

"A:bar" : { //一列

15 : "d",

},

"B:" : { //一列

6 : "w"

3 : "o"

1 : "w"

}

},

// ...

}

查询时，如果只给出行列，那么返回的是最新版本的数据；如果给出了行列时间戳，那么返回的是时间小于或等于时间戳的数据。比如，我们查询"aaaaa"/"A:foo"，返回的值是"y"；查询"aaaaa"/"A:foo"/10，返回的结果就是"m"；查询"aaaaa"/"A:foo"/2，返回的结果是空。

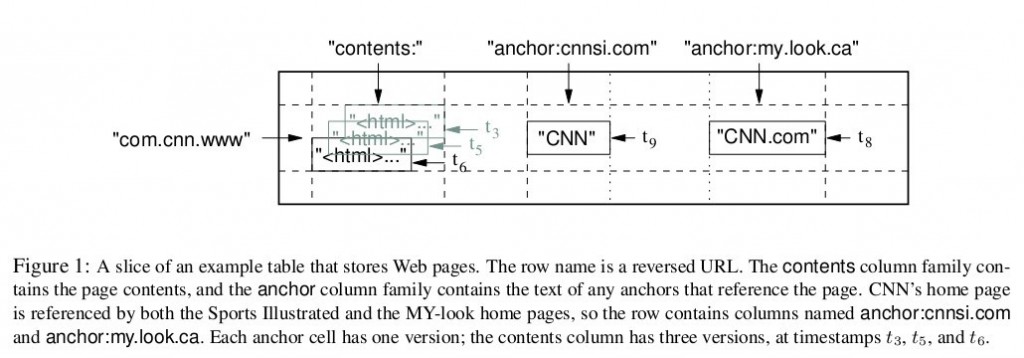


图1是Bigtable论文里给出的例子，Webtable表存储了大量的网页和相关信息。在Webtable，每一行存储一个网页，其反转的url作为行键，比如maps.google.com/index.html的数据存储在键为com.google.maps/index.html的行里，反转的原因是为了让同一个域名下的子域名网页能聚集在一起。图1中的列族"anchor"保存了该网页的引用站点（比如引用了CNN主页的站点），qualifier是引用站点的名称，而数据是链接文本；列族"contents"保存的是网页的内容，这个列族只有一个空列"contents:"。图1中"contents:"列下保存了网页的三个版本，我们可以用("com.cnn.www", "contents:", t5)来找到CNN主页在t5时刻的内容。

再来看看作者说的其它特征：稀疏，分布式，持久化。持久化的意思很简单，Bigtable的数据最终会以文件的形式放到GFS去。Bigtable建立在GFS之上本身就意味着分布式，当然分布式的意义还不仅限于此。稀疏的意思是，一个表里不同的行，列可能完完全全不一样。

### 3 支撑技术

Bigtable依赖于google的几项技术。用GFS来存储日志和数据文件；按SSTable文件格式存储数据；用Chubby管理元数据。

GFS参见[谷歌技术"三宝"之谷歌文件系统](http://blog.csdn.net/opennaive/article/details/7483523)。BigTable的数据和日志都是写入GFS的。

SSTable的全称是Sorted Strings Table，是一种不可修改的有序的键值映射，提供了查询、遍历等功能。每个SSTable由一系列的块（block）组成，Bigtable将块默认设为64KB。在SSTable的尾部存储着块索引，在访问SSTable时，整个索引会被读入内存。BigTable论文没有提到SSTable的具体结构，[LevelDb日知录之四： SSTable文件](http://www.samecity.com/blog/Article.asp?ItemID=100)这篇文章对LevelDb的SSTable格式进行了介绍，因为LevelDB的作者JeffreyDean正是BigTable的设计师，所以极具参考价值。每一个片（tablet）在GFS里都是按照SSTable的格式存储的，每个片可能对应多个SSTable。

Chubby是一种高可用的分布式锁服务，Chubby有五个活跃副本，同时只有一个主副本提供服务，副本之间用Paxos算法维持一致性，Chubby提供了一个命名空间（包括一些目录和文件），每个目录和文件就是一个锁，Chubby的客户端必须和Chubby保持会话，客户端的会话若过期则会丢失所有的锁。关于Chubby的详细信息可以看google的另一篇论文：The Chubby lock service for loosely-coupled distributed systems。Chubby用于片定位，片服务器的状态监控，访问控制列表存储等任务。

### 4 Bigtable集群

Bigtable集群包括三个主要部分：一个供客户端使用的库，一个主服务器（master server），许多片服务器（tablet server）。

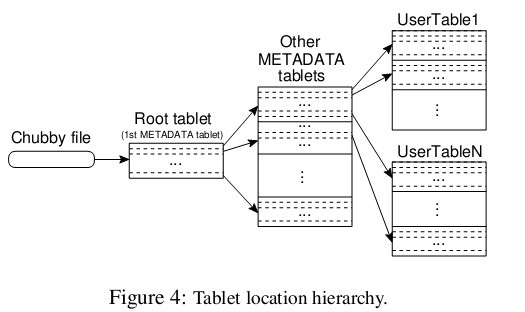
正如数据模型小节所说，Bigtable会将表（table）进行分片，片（tablet）的大小维持在100-200MB范围，一旦超出范围就将分裂成更小的片，或者合并成更大的片。每个片服务器负责一定量的片，处理对其片的读写请求，以及片的分裂或合并。片服务器可以根据负载随时添加和删除。这里片服务器并不真实存储数据，而相当于一个连接Bigtable和GFS的代理，客户端的一些数据操作都通过片服务器代理间接访问GFS。

主服务器负责将片分配给片服务器，监控片服务器的添加和删除，平衡片服务器的负载，处理表和列族的创建等。注意，主服务器不存储任何片，不提供任何数据服务，也不提供片的定位信息。

客户端需要读写数据时，直接与片服务器联系。因为客户端并不需要从主服务器获取片的位置信息，所以大多数客户端从来不需要访问主服务器，主服务器的负载一般很轻。

### 5 片的定位

前面提到主服务器不提供片的位置信息，那么客户端是如何访问片的呢？来看看论文给的示意图，Bigtable使用一个类似B+树的数据结构存储片的位置信息。



首先是第一层，Chubby file。这一层是一个Chubby文件，它保存着root tablet的位置。这个Chubby文件属于Chubby服务的一部分，一旦Chubby不可用，就意味着丢失了root tablet的位置，整个Bigtable也就不可用了。

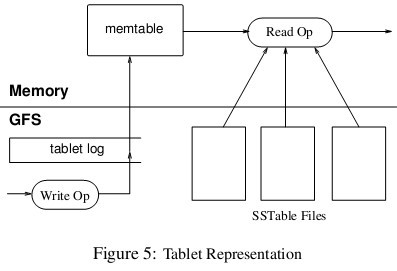
第二层是root tablet。root tablet其实是元数据表（METADATA table）的第一个分片，它保存着元数据表其它片的位置。root tablet很特别，为了保证树的深度不变，root tablet从不分裂。

第三层是其它的元数据片，它们和root tablet一起组成完整的元数据表。每个元数据片都包含了许多用户片的位置信息。

可以看出整个定位系统其实只是两部分，一个Chubby文件，一个元数据表。注意元数据表虽然特殊，但也仍然服从前文的数据模型，每个分片也都是由专门的片服务器负责，这就是不需要主服务器提供位置信息的原因。客户端会缓存片的位置信息，如果在缓存里找不到一个片的位置信息，就需要查找这个三层结构了，包括访问一次Chubby服务，访问两次片服务器。

### 6 片的存储和访问

片的数据最终还是写到GFS里的，片在GFS里的物理形态就是若干个SSTable文件。图5展示了读写操作基本情况。



当片服务器收到一个写请求，片服务器首先检查请求是否合法。如果合法，先将写请求提交到日志去，然后将数据写入内存中的memtable。memtable相当于SSTable的缓存，当memtable成长到一定规模会被冻结，Bigtable随之创建一个新的memtable，并且将冻结的memtable转换为SSTable格式写入GFS，这个操作称为minor compaction。

当片服务器收到一个读请求，同样要检查请求是否合法。如果合法，这个读操作会查看所有SSTable文件和memtable的合并视图，因为SSTable和memtable本身都是已排序的，所以合并相当快。

每一次minor compaction都会产生一个新的SSTable文件，SSTable文件太多读操作的效率就降低了，所以Bigtable定期执行merging compaction操作，将几个SSTable和memtable合并为一个新的SSTable。BigTable还有个更厉害的叫major compaction，它将所有SSTable合并为一个新的SSTable。

遗憾的是，BigTable作者没有介绍memtable和SSTable的详细数据结构。

### 7 BigTable和GFS的关系

集群包括主服务器和片服务器，主服务器负责将片分配给片服务器，而具体的数据服务则全权由片服务器负责。但是不要误以为片服务器真的存储了数据（除了内存中memtable的数据），数据的真实位置只有GFS才知道，主服务器将片分配给片服务器的意思应该是，片服务器获取了片的所有SSTable文件名，片服务器通过一些索引机制可以知道所需要的数据在哪个SSTable文件，然后从GFS中读取SSTable文件的数据，这个SSTable文件可能分布在好几台chunkserver上。

### 8 元数据表的结构

元数据表（METADATA table）是一张特殊的表，它被用于数据的定位以及一些元数据服务，不可谓不重要。但是Bigtable论文里只给出了少量线索，而对表的具体结构没有说明。这里我试图根据论文的一些线索，猜测一下表的结构。首先列出论文中的线索：

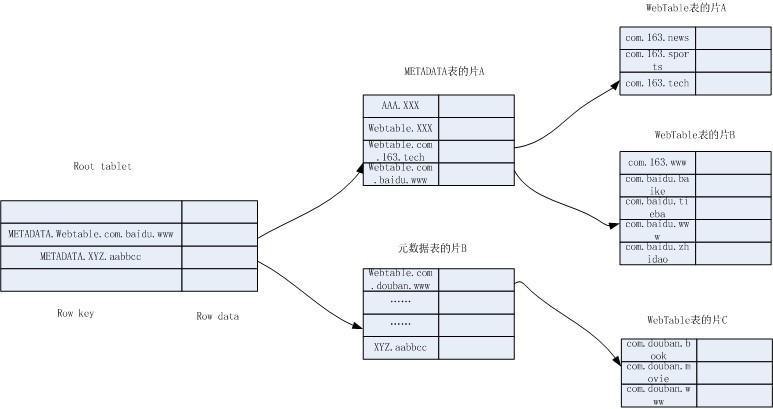
1. The METADATA table stores the location of a tablet under a row key that is an encoding of the tablet's table identifier and its end row.
2. Each METADATA row stores approximately 1KB of data in memory（因为访问量比较大，元数据表是放在内存里的，这个优化在论文的locality groups中提到）.This feature（将locality group放到内存中的特性） is useful for small pieces of data that are accessed frequently: we use it internally for the location column family in the METADATA table.
3. We also store secondary information in the METADATA table, including a log of all events pertaining to each tablet(such as when a server begins  
   serving it).

第一条线索，元数据表的行键是由片所属表名的id和片最后一行编码而成，所以每个片在元数据表中占据一条记录（一行），而且行键既包含了其所属表的信息也包含了其所拥有的行的范围。譬如采取最简单的编码方式，元数据表的行键等于strcat(表名，片最后一行的行键)。

第二点线索，除了知道元数据表的地址部分是常驻内存以外，还可以发现元数据表有一个列族称为location，我们已经知道元数据表每一行代表一个片，那么为什么需要一个列族来存储地址呢？因为每个片都可能由多个SSTable文件组成，列族可以用来存储任意多个SSTable文件的位置。一个合理的假设就是每个SSTable文件的位置信息占据一列，列名为location:filename。当然不一定非得用列键存储完整文件名，更大的可能性是把SSTable文件名存在值里。获取了文件名就可以向GFS索要数据了。

第三个线索告诉我们元数据表不止存储位置信息，也就是说列族不止location，这些数据暂时不是咱们关心的。

通过以上信息，我画了一个简化的Bigtable结构图：



结构图以Webtable表为例，表中存储了网易、百度和豆瓣的几个网页。当我们想查找百度贴吧昨天的网页内容，可以向Bigtable发出查询Webtable表的(com.baidu.tieba, contents:, yesterday)。

假设客户端没有该缓存，那么Bigtable访问root tablet的片服务器，希望得到该网页所属的片的位置信息在哪个元数据片中。使用METADATA.Webtable.com.baidu.tieba为行键在root tablet中查找，定位到最后一个比它大的是METADATA.Webtable.com.baidu.www，于是确定需要的就是元数据表的片A。访问片A的片服务器，继续查找Webtable.com.baidu.tieba，定位到Webtable.com.baidu.www是比它大的，确定需要的是Webtable表的片B。访问片B的片服务器，获得数据。

这里需要注意的是，每个片实际都由若干SSTable文件和memtable组成，而且这些SSTable和memtable都是已排序的。这就导致查找片B时，可能需要将所有SSTable和memtable都查找一遍；另外客户端应该不会直接从元数据表获得SSTable的文件名，而只是获得片属于片服务器的信息，通过片服务器为代理访问SSTable。

### 参考文献

[1] Bigtable: A Distributed Storage System for Structured Data. In proceedings of OSDI'06.

[2] [Understanding HBase and BigTable](http://jimbojw.com/wiki/index.php?title=Understanding_Hbase_and_BigTable).

## [谷歌技术"三宝"之谷歌文件系统](http://blog.csdn.net/opennaive/article/details/7483523)

1. [体系结构](http://blog.csdn.net/opennaive/article/details/7483523#t0)
2. [数据的布局](http://blog.csdn.net/opennaive/article/details/7483523#t1)
3. [元数据服务](http://blog.csdn.net/opennaive/article/details/7483523#t2)
4. [缓存和预取](http://blog.csdn.net/opennaive/article/details/7483523#t3)
5. [出错了肿么办](http://blog.csdn.net/opennaive/article/details/7483523#t4)
   1. [完整性](http://blog.csdn.net/opennaive/article/details/7483523#t5)
   2. [一致性](http://blog.csdn.net/opennaive/article/details/7483523#t6)
   3. [可用性](http://blog.csdn.net/opennaive/article/details/7483523#t7)
6. [测试](http://blog.csdn.net/opennaive/article/details/7483523#t8)
   1. [模拟](http://blog.csdn.net/opennaive/article/details/7483523#t9)
   2. [现实](http://blog.csdn.net/opennaive/article/details/7483523#t10)
7. [胡说八道](http://blog.csdn.net/opennaive/article/details/7483523#t11)
8. [参考文献](http://blog.csdn.net/opennaive/article/details/7483523#t12)

题记：初学分布式文件系统，写篇博客加深点印象。GFS的特点是使用一堆廉价的商用计算机支撑大规模数据处理。

虽然"The Google File System " 是03年发表的老文章了，但现在仍被广泛讨论，其对后来的分布式文件系统设计具有指导意义。然而，作者在设计GFS时，是基于过去很多实验观察的，并提出了很多假设作为前提，这等于给出了一个GFS的应用场景。所以我们自己在设计分布式系统时，一定要注意自己的应用场景是否和GFS相似，不能盲从GFS。

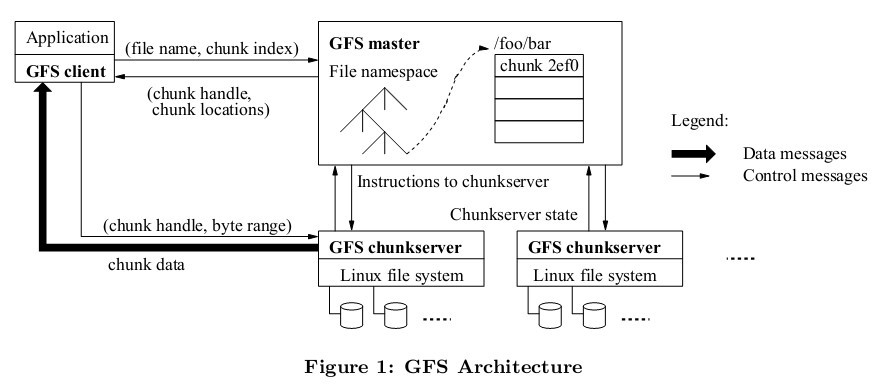
GFS的主要假设如下：

1. GFS的服务器都是普通的商用计算机，并不那么可靠，集群出现结点故障是常态。因此必须时刻监控系统的结点状态，当结点失效时，必须能检测到，并恢复之。
2. 系统存储适当数量的大文件。理想的负载是几百万个文件，文件一般都超过100MB，GB级别以上的文件是很常见的，必须进行有效管理。支持小文件，但不对其进行优化。
3. 负载通常包含两种读：大型的流式读（顺序读），和小型的随机读。前者通常一次读数百KB以上，后者通常在随机位置读几个KB。
4. 负载还包括很多连续的写操作，往文件追加数据（append）。文件很少会被修改，支持随机写操作，但不必进行优化。
5. 系统必须实现良好定义的语义，用于多客户端并发写同一个文件。同步的开销必须保证最小。
6. 高带宽比低延迟更重要，GFS的应用大多需要快速处理大量的数据，很少会严格要求单一操作的响应时间。

从这些假设基本可以看出GFS期望的应用场景应该是大文件，连续读，不修改，高并发。国内的淘宝文件系统（TFS）就不一样，专门为处理小文件进行了优化。

### 1 体系结构

GFS包括一个master结点（元数据服务器），多个chunkserver（数据服务器）和多个client（运行各种应用的客户端）。在可靠性要求不高的场景，client和chunkserver可以位于一个结点。图1是GFS的体系结构示意图，每一结点都是普通的Linux服务器，GFS的工作就是协调成百上千的服务器为各种应用提供服务。



* chunkserver提供存储。GFS会将文件划分为定长数据块，每个数据块都有一个全局唯一不可变的id（chunk\_handle），数据块以普通Linux文件的形式存储在chunkserver上，出于可靠性考虑，每个数据块会存储多个副本，分布在不同chunkserver。
* GFS master就是GFS的元数据服务器，负责维护文件系统的元数据，包括命名空间、访问控制、文件-块映射、块地址等，以及控制系统级活动，如垃圾回收、负载均衡等。
* 应用需要链接client的代码，然后client作为代理与master和chunkserver交互。master会定期与chunkserver交流（心跳），以获取chunkserver的状态并发送指令。

图1还描述了应用读取数据的流程。1.应用指定读取某个文件的某段数据，因为数据块是定长的，client可以计算出这段数据跨越了几个数据块，client将文件名和需要的数据块索引发送给master；2.master根据文件名查找命名空间和文件-块映射表，得到需要的数据块副本所在的地址，将数据块的id和其所有副本的地址反馈给client；3.client选择一个副本，联系chunkserver索取需要的数据；4.chunkserver返回数据给client。

### 2 数据的布局

GFS将文件条带化，按照类似RAID0的形式进行存储，可以提高聚合带宽。事实上，大多数分布式存储系统都会采取这种策略。GFS将文件按固定长度切分为数据块，master在创建一个新数据块时，会给每个数据块分配一个全局唯一且不可变的64位id。每个数据块以Linux文件的形式存储在chunkserver的本地文件系统里。

GFS为数据块设置了一个很大的长度，64MB，这比传统文件系统的块长要大多了。大块长会带来很多好处：1.减少client和master的交互次数，因为读写同一个块只需要一次交互，在GFS假设的顺序读写负载的场景下特别有用；2.同样也减少了client和chunkserver的交互次数，降低TCP/IP连接等网络开销；3.减少了元数据的规模，因此master可以将元数据完全放在内存，这对于集中式元数据模型的GFS尤为重要。

大数据块也有缺点。最大的缺点可能就是内部碎片了，不过考虑到文件一般都相当大，所以碎片也只存在于文件的最后一个数据块。还有一个缺点不是那么容易看出来，由于小文件可能只有少量数据块，极端情况只有一个，那么当这个小文件是热点文件时，存储该文件数据块的chunkserver可能会负载过重。不过正如前面所说，小文件不在GFS的优化范围。

为了提高数据的可靠性和并发性，每一个数据块都有多个副本。当客户端请求一个数据块时，master会将所有副本的地址都通知客户端，客户端再择优（距离最短等）选择一个副本。一个典型的GFS集群可能有数百台服务器，跨越多个子网，因此在考虑副本的放置时，不仅要考虑机器级别的错误，还要考虑整个子网瘫痪了该怎么办。将副本分布到多个子网去，还可以提高系统的聚合带宽。因此创建一个数据块时，主要考虑几个因素：1.优先考虑存储利用率低于平均水平的结点；2.限制单个结点同时创建副本的数量；3.副本尽量跨子网。

### 3 元数据服务

GFS是典型的集中式元数据服务，所有的元数据都存放在一个master结点内。元数据主要包括三种：文件和数据块的命名空间，文件-数据块映射表，数据块的副本位置。所有的元数据都是放在内存里的。

前两种元数据会被持久化到本地磁盘中，以操作日志的形式。操作日志会记录下这两种元数据的每一次关键变化，因此当master宕机，就可以根据日志恢复到某个时间点。日志的意义还在于，它提供了一个时间线，用于定义操作的先后顺序，文件、数据块的版本都依赖于这个时间顺序。

数据块的副本位置则没有持久化，因为动辄数以百计的chunkserver是很容易出错的，因此只有chunkserver对自己存储的数据块有绝对的话语权，而master上的位置信息很容易因为结点失效等原因而过时。取而代之的方法是，master启动时询问每个chunkserver的数据块情况，而且chunkserver在定期的心跳检查中也会汇报自己存储的部分数据块情况。

GFS物理上没有目录结构，也不支持链接操作，使用一张表来映射文件路径名和元数据。

### 4 缓存和预取

GFS的客户端和chunkserver都不会缓存任何数据，这是因为GFS的典型应用是顺序访问大文件，不存在时间局部性。空间局部性虽然存在，但是数据集一般很大，以致没有足够的空间缓存。

我们知道集中式元数据模型的元数据服务器容易成为瓶颈，应该尽量减少客户端与元数据服务器的交互。因此GFS设计了元数据缓存。client需要访问数据时，先询问master数据在哪儿，然后将这个数据地址信息缓存起来，之后client对该数据块的操作都只需直接与chunkserver联系了，当然缓存的时间是有限的，过期作废。

master还会元数据预取。因为空间局部性是存在，master可以将逻辑上连续的几个数据块的地址信息一并发给客户端，客户端缓存这些元数据，以后需要时就可以不用找master的麻烦了。

### 5 出错了肿么办

引用：“We treat component failures as the norm rather than the exception."

分布式系统整体的可靠性是至关重要的。GFS集群使用的都是普通的商用计算机，而且机器的数量众多，设备故障经常出现，如何处理结点失效的问题是GFS最大的挑战。

#### 5.1 完整性

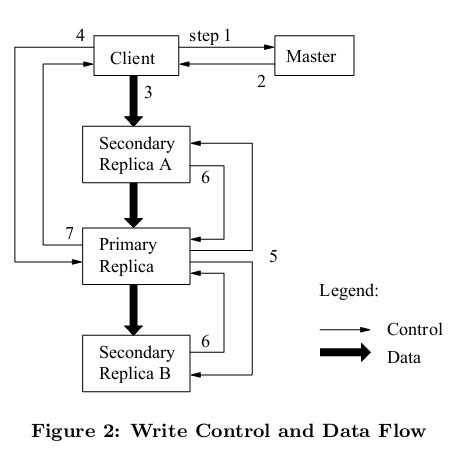
GFS使用数以千计的磁盘，磁盘出错导致数据被破坏时有发生，我们可以用其它副本来恢复数据，但首先必须能检测出错误。chunksever会使用校验和来检测错误数据。每一个块（chunk）都被划分为64KB的单元（block），每个block对应一个32位的校验和。校验和与数据分开存储，内存有一份，然后以日志的形式在磁盘备一份。

chunkserver在发送数据之前会核对数据的校验和，防止错误的数据传播出去。如果校验和与数据不匹配，就返回错误，并且向master反映情况。master会开始克隆副本的操作，完成后就命令该chunkserver删除非法副本。

#### 5.2 一致性

一致性指的是master的元数据和chunkserver的数据是否一致，多个数据块副本之间是否一致，多个客户端看到的数据是否一致。

先来看看元数据一致性。GFS的命名空间操作是原子性的，并且用日志记录下操作顺序。虽然GFS没有目录结构，但是仍然有一颗逻辑的目录树，树的每个结点都有自己的读写锁，每个元数据操作都需要获得一系列的锁，应该是写锁会阻塞其它的锁，而读锁只阻塞写锁而不阻塞读锁。比如/home/user "目录" 正在创建快照，需要获得/home的读锁和/home/user的写锁，这时如果想创建文件/home/user/foo会被阻塞，因为需要获得/home、/home/user的读锁以及/home/user/foo的写锁，快照会阻塞创建操作获取/home/user的读锁。如果是在一个有传统目录树结构的文件系统里，创建一个文件需要修改父目录的数据，因此需要获得父目录的写锁。这种锁机制允许在一个目录里并发修改数据（如并发创建文件等），这在传统文件系统里是不允许的。



再来看看GFS是如何并发写（write）的，GFS必须将对数据块的修改同步到每一个副本。考虑一下多个应用同时修改同一数据块的情况，我们必须为修改操作定义统一的时序，不然多个副本会出现不一致的情况，那么定义时序由谁做呢？还记得前面提到的元数据缓存么，为了减少master的负担，client在获得副本位置后就不再和master交互，所以必然需要选出一个master代理来完成这个任务。事实上GFS采用了租约（lease）的机制，master会将租约授权给某个副本，称为primary，由这个primary来确定数据修改的顺序，其它副本照做就是。

图2是写操作的控制流和数据流：

1. client需要更新一个数据块，询问master谁拥有该数据块的租约（谁是primary）；
2. master将持有租约的primary和其它副本的位置告知client，client缓存之；
3. client向所有副本传输数据，这里副本没有先后顺序，根据网络拓扑情况找出最短路径，数据从client出发沿着路径流向各个chunkserver，这个过程采用流水线（网络和存储并行）。chunkserver将数据放到LRU缓存；
4. 一旦所有的副本都确定接受数据，client向primary发送写请求，primary为这个前面接受到的数据分配序列号（primary为所有的写操作分配连续的序列号表示先后顺序），并且按照顺序执行数据更新；
5. primary将写请求发送给其它副本，每个副本都按照primary确定的顺序执行更新；
6. 其它副本向primary汇报操作情况；
7. primary回复client操作情况，任何副本错误都导致此次请求失败，并且此时副本处于不一致状态（写操作完成情况不一样）。client会尝试几次3到7的步骤，实在不行就只能重头来过了。

如果一个写请求太大了或者跨越了chunk，GFS的client会将其拆分为多个写请求，每个写请求都遵循上述过程，但是可能和其它应用的写操作交叉在一起。所以这些写操作共享的数据区域就可能包含几个写请求的碎片（就是下文提到的undefined状态）。

GFS还提供另一种写操作append record，append只在文件的尾部以record为单位（为了避免内部碎片，record一般不会很大）写入数据。append是原子性的，GFS保证将数据顺序地写到文件尾部至少一次。append record的流程和图2类似，只是在primary有点区别，GFS必须保证一个record存储在一个chunk内，所以当primary判断当前chunk无足够空间时，就通知所有副本将chunk填充，然后汇报失败，client会申请创建新chunk并重新执行一次append record操作。如果该chunk大小合适，primary会将数据写到数据块的尾部，然后通知其它副本将数据写到一样的偏移。任何副本append失败，各个副本会处于不一致状态（完成或未完成），这时primary必然是成功写入的（不然就没有4以后的操作了）。客户端重试append record操作时，因为primary的chunk长度已经变化了，primary就必须在新的偏移写入数据，而其它副本也是照做。这就导致上一个失败的偏移位置，各个副本处于不一致状态，应用必须自己区分record正确与否，我称这为无效数据区。

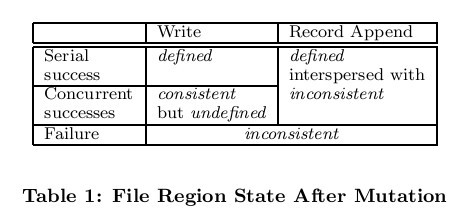


表1说明了GFS的一致性保证，明白write和append操作后就容易理解了。consistent指的是多个副本之间是完全一致的；defined指的是副本数据修改后不仅一致而且client能看到其修改的数据全貌。

* 成功的连续write是已定义的，各个副本数据一致；
* 成功的并发write能保证一致性性，即各个副本是一样的，但数据并不一定如用户所期望，如前所述，多个用户的修改可能交错在一起；
* 失败的write操作，使得副本之间不一致，而且数据undefined，不同client可能看到不同的数据（注意区别defined、undefined数据的方法）；
* 成功的append操作，不管是顺序还是并发都是defined，因为GFS保证了append是原子性的（atomically at least once）。有效数据区确实是defined的，但是失败append操作留下的无效数据区可能会有不一致的情况，所以中间可能零散分布着不一致的数据。

如上所述，在primary的协调下，能保证并发write的一致性。但还有一些可能会导致数据不一致，比如chunkserver宕机错过了数据更新，这时就会出现新旧版本的数据，GFS为每个数据块分配版本来解决这个问题。master每次授权数据块租约给primary之前，都会增加数据块的版本号，并且通知所有副本更新版本号。客户端需要读数据时当然会根据这个版本号来判断副本的数据是否最新。

#### 5.3 可用性

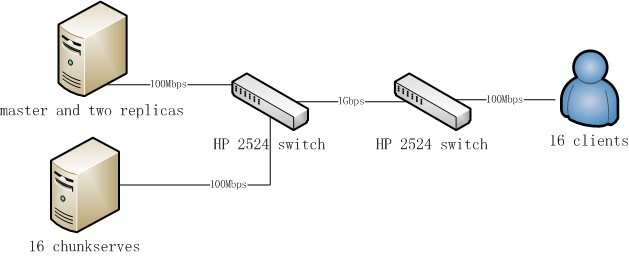
为了保证数据的可用性，GFS为每个数据块存储了多个副本，在数据的布局里有介绍，这里主要关注下元数据的可用性。

GFS是典型的集中式元数据模型，一个元数据服务器承担了巨大的压力，不仅有性能的问题，还有单点故障问题。master为了能快速从故障中恢复过来，采用了log和checkpoint技术，log记录了元数据的每一次变化。用咱们备份的话来说，checkpoint就相当于一次全量备份，log相当于连续数据保护，master宕机后，就先恢复到checkpoint，然后根据log恢复到最新状态。每次创建一个新的checkpoint，log就可以清空，这有效控制了log的大小。

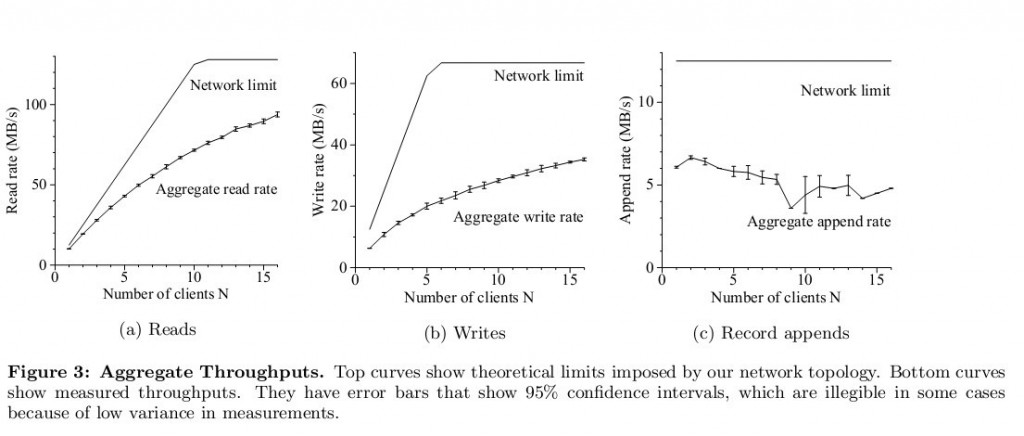
这还不够，如果master完全坏了肿么办？GFS设置了“影子”服务器，master将日志备份到影子上，影子按照日志把元数据与master同步。如果master悲剧了，我们还有影子。平时正常工作时，影子可以分担一部分元数据访问工作，当然不提供直接的写操作。

### 6 测试

#### 6.1 模拟



实验的网络拓扑图大概就是这样。集群包括一个master和它的两个影子，16个chunkserver，16个client，和两个HP交换机。两个交换机之间是1Gbps的链路，结点与交换机的链路是100Mbps。聚合带宽的理论上限是125MB/s，而单个client的理论带宽上限是12.5MB/s。



实验一：N个client同时随机各自读取1GB数据。

图3(a)，x轴是N，y轴是聚合带宽，上面一条线是理论值，下面一条线是实际值。当N=1时，吞吐率是10MB/s，达到了理论值的80%。当N=16时，吞吐率是94MB/s，达到理论值的75%。此时瓶颈可能是在chunkserver，因为client数量很多，同时读一个chunkserver的概率很大。

实验二：N个client同时写N个不同文件，各自连续写1GB。

图3(b)。理论值上限是67MB/s（The limit plateaus at 67 MB/s because we need to write each byte to 3 of the 16 chunkservers,  
each with a 12.5 MB/s input connection），这个理论值我没有看明白是怎么算的。当N=1，吞吐率是6.3MB/s，达到极限的一半，主要的瓶颈是数据在副本之间传递。N=16时，聚合带宽为35MB/s（每个client有2.2MB/s），达到极限的一半，这里chunkserver同时接受多个请求的情况比读更严重，因为得写3个副本。写比期望的要慢。

实验三：N个client同时向一个文件append record。

图3(c)。理论上限值是一台chunkserver的带宽，即12.5MB/s。当N=1时，吞吐率有6.0MB/s。当N=16时，下降到4.8MB/s。这可能是因为拥塞。

#### 6.2 现实

文章里还介绍了两个真实集群的使用情况，cluster A and B。

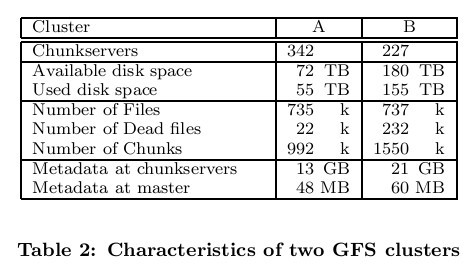
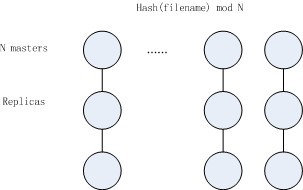


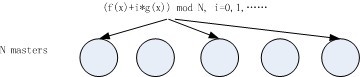
表2是两个集群的情况。A和B都有数百个chunkserver，存储利用率很高，冗余度是3，所以实际存储的数据是18TB和52TB。文件数相当，dead file是需要删除的文件，但还没被垃圾回收（GFS会为删除的文件保留三天才回收空间）。B的块数更多，意味着文件更大，不过A和B的文件平均都只有1到2个块。chunkserver的元数据主要是block（64KB）的校验和（4 Bytes），以及数据块的版本信息。master的元数据（文件和数据块的名字，文件-数据块映射，块位置等）相当小，平均每个文件只有100B元数据，大部分是用于存储文件名。平均下来，每个服务器有50~100MB的元数据。

### 7 胡说八道

画了个全分布式元数据模型。



这估计是最简单的，命名空间被划分为几个区域，master各管各的。为了可靠性，每个master做几个副本。映射方法可以是文件名计算哈希取模，好的哈希函数可以使文件随机分布，负载比较均衡。当然扩展性不是很好，加入新的结点，所有文件得重新映射。而且冗余度只能靠机器堆了，不能软件控制。



又构思了个复杂点的。将文件映射到r(<=N)个master，利用参数r控制冗余度，当r=N时就变成全对等元数据集群。这个模型需要r个不同的哈希函数，为了减少开销，可以用两个函数模拟多个函数。比如有随机性很好的f(x)和g(x)函数，我们用式子f(x)+i\*g(x)来模拟，其中i为非负整数。当我们处理文件x时，就用前面式子求出r个不同的位置（有冲突的概率，实际可能不止计算r次）。

胡说八道，切勿当真；如有雷同，纯属巧合。

### 参考文献：

[1] The Google File System.

# 第3篇 管理篇

本篇主要介绍MongoDB常用的运行维护管理工具、各种命令管理以及访问控制方面的技术；讲解如何管理MongoDB 数据库。包括启动和停止MongoDB、访问控制、命令行操作、监控、数据导出mongoexport、数据导入mongoimprot、数据备份和修复。看完第三篇内容就能够成为MongoDB DBA，因为这一篇包含了MongoDB所需的全部技能。

## 第8章 MongoDB管理

本章主要介绍MongoDB的一些管理操作，主要包括以下操作：

* 启动和停止MongoDB服务。
* 设置MongoDB的访问。
* 命令行操作。
* 对MongoDB的一些进程的控制。
* MongoDB如何进行监控。
* 对MongoDB数据的导入、导出、修复、备份。

### 1.启动和停止MongoDB

执行mongod,启动MongoDB服务器。mongod有很多选项，在命令中执行 mongod --help

主要选项如下：

--dbpath

指定数据目录，默认值是C:\data\db。每个mongod进程都需要独立的数据目录。如果要是有3个mongod

实例，那么必须有3个独立的数据目录。当mongod启动时，会在数据库目录中创建mongod.lock文件

这个文件用于防止其他的mongod纯净使用该数据目录。

--port

指定服务器监听的端口号，默认端口27017.要运行多个mongod进程，则要给每个指定不同的端口号。

--logpath

指定日志的输出路径。如果对文件夹有读写权限，系统会在文件不存在时创建它。它会将已有文件覆盖掉，

清除所有原来的日志记录。如果想要保留原来的日志，需使用--logappend选项。

--config

指定配置文件，加载命令行未指定的各种选项。

### 2.从配置文件启动

MongoDB支持从文件获取配置信息.当需要配置非常多或者要自动化MongoDB的启动时会用到.

指定配置文件可以用-f或--config选项.

如:

mongod --config refactorConfig.txt

refactorConfig.txt内容如下:

#start MongoDB  
port = 10000  
dbpath = "f:\mongo\db"  
logpath = "f:\mongo\log\MongoDB.txt"  
rest = true

配置文件和命令行的功能一样

mongod --dbpath "f:\mongo\db" --logpath "f:\mongo\log\MongoDB.txt" --rest --port 10000

配置文件的特点:

a.以#开头的行是注释

b.指定选项的语法是这种"选项=值"的形式.选项是区分大小写的.

c.命令行如--rest的开关选项,值要设为true

### 3.停止MongoDB

可以使用shutdown命令{"shutdown":1},这个命令要在admin数据库下使用.shell还提供了辅助函数:

use admin

db.shutdownServer()

### 4. 监控

使用管理接口,默认情况下,启动mongod会启动基本的http服务器,该服务的默认端口是28017.可以在浏览器中输入

localhost:28017.有些链接需要在mongod启动时,用--rest选项开启rest支持 才能进去.当开启rest支持后,可以

在mongod启动时使用--nohttpinterface来关闭管理接口.

### 5.serverStatus

要获取运行中的MongoDB服务器统计信息,最基本的工具是serverStatus命令

db.runCommand({"serverStatus":1})

serverStatus返回的键解释:

"globalLock"的值表示全局写入锁占用了服务器多少时间(单位微秒)

"mem"包含服务器内存映射了多少数据,服务器进程的虚拟内存和常驻内存的占用情况(单位MB)

"indexCounters"表示B树在磁盘检索("misses")和内存检索("hits")的次数.如果这个比值开始上升,就要考虑加内存了.

"backgroundFlushing"表示后台做了多少次fsync以及用了多少时间

"opcounters"文档包含了每种主要操作的次数

"asserts"统计了断言的次数

### 6.mongostat

serverStatus虽然强大,但对服务器的监控来说不怎么容易.MongoDB提供了mongostat

mongostat输出一些serverStatus提供的重要信息,它会每秒输出新的一行,比之前看到的静态数据实时性要好.

它输出多个列,分别是 inserts/s commands/s vsize 和 %locked,与serverStatus的数据相对应.

还可以使用第三方插件进行数据库的监控.

### 7.安全和认证

认证的基础知识

每个MongoDB实例中的数据库都可以有很多用户,如果开启了安全性检查,这只有数据库认证用户才能执行读或写操作.

在认证的上下文中,MongoDB会将普通的数据作为admin数据库处理.admin数据库中的用户被称为超级用户(管理员).

在认证后,管理员可以读写所有数据库,执行特定的管理命令,如listDatabases和shutdown.

在开启安全检查前,至少要有个管理员帐号,在shell连接的是没有开启安全检查的服务器

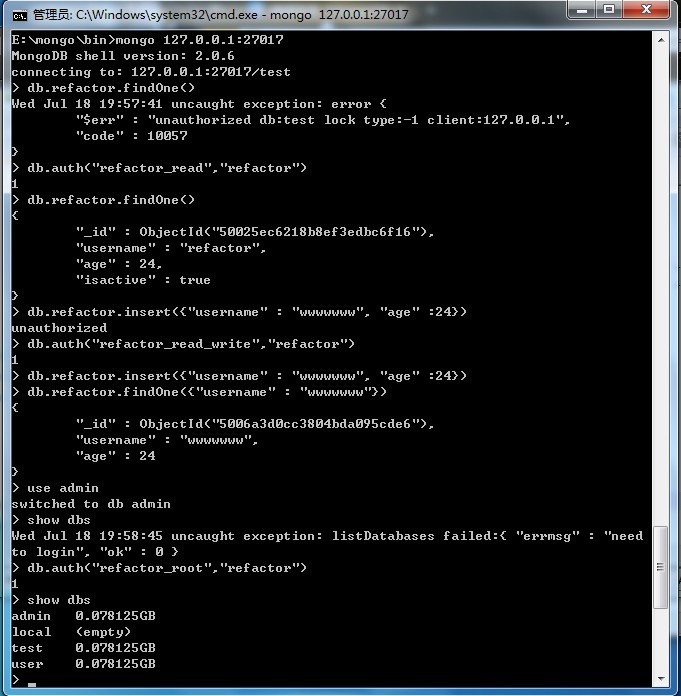


上面添加了管理员refactor\_root,在test数据库添加了两个普通账号,其中一个有只读权限.在shell中创建只读用户只要

在addUser的第三个参数设为true.调用addUser必须有响应数据库的写权限.这里可以对所有数据库调用addUser,

因为还没有开启安全检查.

重启数据库,重启时加入 --auth 命令行选项,开启安全检查



第一次连接时,不能test数据库执行任何操作,作为只读用户认证后,能查找,不能插入数据.能读写用户认证后,能查找和插入

数据,但不能使用show dbs 来列举所有数据库.超级用户认证后,可以为所欲为了.

### 8.认证的工作原理

数据库的用户帐号以文档的形式存储在system.users集合里.文档的结构是

{  
"\_id" : ObjectId("5006a037dff37e149322fd83"),  
"user" : "refactor\_read\_write",  
"readOnly" : false,  
"pwd" : "5a84584ac51d3f702461fce4c46b0d6b"//是根据用户名和密码生成的散列  
}

知道了用户信息是如何存储的以及存储位置后,就可以进行日常的管理工作了.

如删除帐户:

> db.system.users.remove({"user":"refactor\_read"})  
> db.auth("refactor\_read","refactor")  
0

用户认证时,服务器将认证和连接绑定来跟踪认证,也就是说如果驱动程序或是工具使用了连接池或是因故障切换到

另一个节点,所有认证用户必须对每个新连接重新认证.

MongoDB的传输协议是不加密的,如需加密,可以用ssh隧道或者类似的技术做客户端和服务器间的加密.

建议将MongoDB服务器放在防火墙或放在只有应用服务器能访问的网络中.如果MongoDB必须能被外面访问到的话,

建议使用--bindip选项,可以指定mongod绑定到的本地ip地址.如:只能从本机应用服务器访问,可以使用

mongod --bindip localhost

默认情况下MongoDB会开启一个简单的http服务器,便于查看运行,锁,复制等方面的信息,要是不想公开这些信息,可以用

--nohttpinterface来关闭管理接口.

可以用--noscripting完全禁止服务端javascript执行

### 9.备份和修复

MongoDB将所有数据都存放在 数据目录 下,默认目录是C:\data\db\.启动MongoDB的时候可以用--dbpath指定数据目录.

不论数据目录在哪里,它都存放着MongoDB的所有数据.要想备份MongoDB,只要简单的复制数据目录中的所有文件即可.

除非服务器做了完整的fsync,还不允许写入,否则在运行MongoDB时创建数据目录的副本并不安全,这样的备份可能已经

破损了,需要修复.

在运行MongoDB时创建数据目录的副本并不安全,所以就得先把服务器关了,再复制数据目录.但是关闭数据库就要停止业务.

### 10.mongodump和mongorestore

mongodump是一种能在运行时备份的方法.mongodump对运行的MongoDB做查询,然后将所有查到的文档写入磁盘.

因为mongodump是一般的客户端,所以可供运行的MongoDB使用,即便是正在处理其他请求或是执行写入也没有问题.

mongodump使用普通的查询机制,所以产生的备份不一定是服务器数据的实时快照.服务器在备份过程中处理写入时,非常明显.

mongodump备份时的查询会对其他客户端的性能产生影响.

mongodump --help 获得帮助

mongorestore是从备份中恢复数据的工具.

mongorestore获取mongodump 的输出结果,并将备份的数据插入运行的MongoDB实例中.

如:将数据库test备份到backup目录

mongodump -d test -o backup

使用mongorestore 恢复到testNew 数据库  
mongorestore -d testNew --drop backup/test/

-d指定要恢复的数据库.--drop指在恢复前删除集合(若存在),否则数据就会与现有集合数据合并,可能会覆盖一些文档.

可以使用mongorestore --help获得帮助信息

### 11.fsync和锁

虽然使用mongodump和mongorestore能不停机备份,但是却失去了获取实时数据视图的能力.MongoDB的fsync命令

能在MongoDB运行时复制数据目录还不会损坏数据.

fsync命令会强制服务器将所有缓冲区写入磁盘.还可以选择上锁住址对数据库的进一步写入,知道释放锁为止.写入锁是让

fsync在备份时发挥作用的关键.

在shell中,强制执行fsync并获得写入锁:

db.runCommand({"fsync":1,"lock":1})

这时,数据目录的数据就是一致的,且为数据的实时快照.因为上了锁,可以安全的将数据目录副本作为备份.要是数据库运行在

有快照功能的文件系统上时,比如LVM,EBS,这个很有用,因为拍个数据库目录的快照很快.

备份好了,解锁:

db.$cmd.sys.unlock.findOne()

db.currentOp()

运行db.currentOp()是为了确保已经解锁了(初次请求解锁会花点时间)

有了fsync命令,就能非常灵活的备份,不用停掉服务器,也不用牺牲备份的实时性能.要付出的代价就是一些写入操作被

暂时阻塞了.唯一不耽误读写还能保证实时快照的备份方式就是通过从服务器备份.

### 12.从属备份

虽然上面的备份方式很灵活,但都没有从服务器上备份好.当复制的方式运行MongoDB,前面的提到的备份技术就不仅能用在

主服务器上,也可用在从服务器上.从服务器的数据几乎与主服务器同步.因为不太在乎从属服务器的性能或者是能不能读写,

于是就能随意选择上面的3种备份方式:关停,转存或恢复工具或fsync命令.从服务器上备份是MongoDB推荐的备份方式.

### 13.修复

MongoDB的存储方式不能保证磁盘上的数据还能用,因为可能有损毁.MongoDB内置的修复功能会试着恢复损坏的数据文件.

未正常停止MongoDB后应该修复数据库.修复数据库的方式很简单就是 mongod --repair 来启动服务器.

修复数据库的实际过程很简单:将所有的文档导出后马上导入,忽略无效的文档.完成后,会重建索引.数据量大的话,会花很多时间,

因为所有数据都要验证,所有索引都要重建(从MongoDB 1.8 以后版本引入了日志系统,使修复时间打打的缩短).

修复后可能会比修复前少些文档,因为损坏的文档被删除了.

修复数据库还能起到压缩数据的作用.闲置控件(如删除体积较大集合,或删除大量文档后腾出的空间)在修复后会被重新利用.

修复运行中的服务器上的数据库,要在shell用repairDatabases.

use test  
db.repairDatabase()

## 第9章 MongoDB 索引

索引能提高检索数据的速度，你可以想像成在MySQL中创建索引一样，同样索引也是用B-Tree也实现的。

### 1.单列索引

在字段x上创建索引，1 (ascending) or -1 (descending)

> db.data.ensureIndex({x:1})

显示表data里面的所有索引

> db.data.getIndexes()

[

{

"name" : "\_id\_",

"ns" : "recommender.data",

"key" : {

"\_id" : 1

}

},

{

"\_id" : ObjectId("4befb146b0e29ba1ce20e0bb"),

"ns" : "recommender.data",

"key" : {

"x" : 1

},

"name" : "x\_1"

}

]

查找字段x为6的值，此时已经用到索引了

> db.data.find({x:6})

{ "\_id" : ObjectId("4bee804ba23d558eb6687117"), "x" : 6, "name" : "caihuafeng1" }

{ "\_id" : ObjectId("4bee804ba23d558eb6687118"), "x" : 6, "name" : "caihuafeng2" }

{ "\_id" : ObjectId("4bee804ba23d558eb6687119"), "x" : 6, "name" : "caihuafeng3" }

{ "\_id" : ObjectId("4bee804ba23d558eb668711a"), "x" : 6, "name" : "caihuafeng4" }

{ "\_id" : ObjectId("4bee804ba23d558eb668711b"), "x" : 6, "name" : "caihuafeng5" }

{ "\_id" : ObjectId("4bee804ba23d558eb668711c"), "x" : 6, "name" : "caihuafeng6" }

{ "\_id" : ObjectId("4bee804ba23d558eb668711d"), "x" : 6, "name" : "caihuafeng7" }

{ "\_id" : ObjectId("4bee804ba23d558eb668711e"), "x" : 6, "name" : "caihuafeng8" }

{ "\_id" : ObjectId("4bee804ba23d558eb668711f"), "x" : 6, "name" : "caihuafeng9" }

{ "\_id" : ObjectId("4bee804ba23d558eb6687120"), "x" : 6, "name" : "caihuafeng10" }

### 2.默认索引

上述1中db.data.getIndexes()显示出来的一共有2个索引，其中\_id是创建表的时候自动创建的索引，此索引是不能够删除的。

An index is always created on \_id. This index is special and cannot be deleted. The \_id index enforces uniqueness for its keys.

### 3.文档作为索引的键值

#### a.单列索引

MongoDB的官方文档上面是这样说的:

Documents as Keys

Indexed fields may be of any type, including documents:

往数据库recommender的表data中插入三条记录

> db.data.insert({name:"1616",info:{url:"http://www.1616.net/",city:"beijing"}});

> db.data.insert({name:"hao123",info:{url:"http://www.hao123.com/",city:"beijing"}});

> db.data.insert({name:"ll4la",info:{url:"http://www.114la.com/",city:"dongguan"}});

对字段info创建索引

> db.data.ensureIndex({info: 1});

显示表data上的所有索引

> db.data.getIndexes();

[

{

"name" : "\_id\_",

"ns" : "recommender.data",

"key" : {

"\_id" : 1

}

},

{

"\_id" : ObjectId("4befb146b0e29ba1ce20e0bb"),

"ns" : "recommender.data",

"key" : {

"x" : 1

},

"name" : "x\_1"

},

{

"\_id" : ObjectId("4befb76bb0e29ba1ce20e0bf"),

"ns" : "recommender.data",

"key" : {

"info" : 1

},

"name" : "info\_1"

}

]

查找指定的记录，此时会用到索引

> db.data.find({info: {url:"http://www.1616.net/",city:"beijing"}});

{ "\_id" : ObjectId("4befb711b0e29ba1ce20e0bc"), "name" : "1616", "info" : { "url" : "http://www.1616.net/", "city" : "beijing" } }

#### b.组合索引

建立组合索引

> db.data.ensureIndex({"info.url":1, "info.city":1});

> db.data.getIndexes();

[

{

"name" : "\_id\_",

"ns" : "recommender.data",

"key" : {

"\_id" : 1

}

},

{

"\_id" : ObjectId("4befb146b0e29ba1ce20e0bb"),

"ns" : "recommender.data",

"key" : {

"x" : 1

},

"name" : "x\_1"

},

{

"\_id" : ObjectId("4befb76bb0e29ba1ce20e0bf"),

"ns" : "recommender.data",

"key" : {

"info" : 1

},

"name" : "info\_1"

},

{

"\_id" : ObjectId("4befb9d1b0e29ba1ce20e0c0"),

"ns" : "recommender.data",

"key" : {

"info.url" : 1,

"info.city" : 1

},

"name" : "info.url\_1\_info.city\_1"

}

]

下面几个操作均会用到索引

> **db.data.find({"info.url": "http://www.1616.net/", "info.city": "beijing"});**

{ "\_id" : ObjectId("4befb711b0e29ba1ce20e0bc"), "name" : "1616", "info" : { "url" : "http://www.1616.net/", "city" : "beijing" } }

**> db.data.find({"info.url": "http://www.1616.net/"});**

{ "\_id" : ObjectId("4befb711b0e29ba1ce20e0bc"), "name" : "1616", "info" : { "url" : "http://www.1616.net/", "city" : "beijing" } }

1表示升序(asc)，-1表示降序(desc)

**> db.data.find({"info.url": /http:\*/i}).sort({"info.url": 1, "info.city": 1});**

{ "\_id" : ObjectId("4befb740b0e29ba1ce20e0be"), "name" : "ll4la", "info" : { "url" : "http://www.114la.com/", "city" : "dongguan" } }

{ "\_id" : ObjectId("4befb711b0e29ba1ce20e0bc"), "name" : "1616", "info" : { "url" : "http://www.1616.net/", "city" : "beijing" } }

{ "\_id" : ObjectId("4befb723b0e29ba1ce20e0bd"), "name" : "hao123", "info" : { "url" : "http://www.hao123.com/", "city" : "beijing" } }

**> db.data.find({"info.url": /http:\*/i}).sort({"info.url": 1});**

{ "\_id" : ObjectId("4befb740b0e29ba1ce20e0be"), "name" : "ll4la", "info" : { "url" : "http://www.114la.com/", "city" : "dongguan" } }

{ "\_id" : ObjectId("4befb711b0e29ba1ce20e0bc"), "name" : "1616", "info" : { "url" : "http://www.1616.net/", "city" : "beijing" } }

{ "\_id" : ObjectId("4befb723b0e29ba1ce20e0bd"), "name" : "hao123", "info" : { "url" : "http://www.hao123.com/", "city" : "beijing" } }

**> db.data.find({"info.url": /http:\*/i}).sort({"info.url": -1});**

{ "\_id" : ObjectId("4befb723b0e29ba1ce20e0bd"), "name" : "hao123", "info" : { "url" : "http://www.hao123.com/", "city" : "beijing" } }

{ "\_id" : ObjectId("4befb711b0e29ba1ce20e0bc"), "name" : "1616", "info" : { "url" : "http://www.1616.net/", "city" : "beijing" } }

{ "\_id" : ObjectId("4befb740b0e29ba1ce20e0be"), "name" : "ll4la", "info" : { "url" : "http://www.114la.com/", "city" : "dongguan" } }

### 4.组合索引

注意，这里的组合索引与上述3中的b中的组合索引是有点不同的，4里面是对一级字段建立组合索引，而上述3中是对二级字段建立组合索引。

在字段name及info上面创建组合索引

> db.data.ensureIndex({name: 1, info: -1});

当创建组合索引时，字段后面的1表示升序，-1表示降序，是用1还是用-1主要是跟排序的时候或指定范围内查询的时候有关的，具体看下面的英文原文的说明。

When creating an index, the number associated with a key specifies the direction of the index, so it should always be 1 (ascending) or -1 (descending). Direction doesn't matter for single key indexes or for random access retrieval but is important if you are doing sorts or range queries on compound indexes.

显示所有的索引

> db.data.getIndexes();

[

{

"name" : "\_id\_",

"ns" : "recommender.data",

"key" : {

"\_id" : 1

}

},

{

"\_id" : ObjectId("4befb146b0e29ba1ce20e0bb"),

"ns" : "recommender.data",

"key" : {

"x" : 1

},

"name" : "x\_1"

},

{

"\_id" : ObjectId("4befb76bb0e29ba1ce20e0bf"),

"ns" : "recommender.data",

"key" : {

"info" : 1

},

"name" : "info\_1"

},

{

"\_id" : ObjectId("4befb9d1b0e29ba1ce20e0c0"),

"ns" : "recommender.data",

"key" : {

"info.url" : 1,

"info.city" : 1

},

"name" : "info.url\_1\_info.city\_1"

},

{

"\_id" : ObjectId("4befbfcfb0e29ba1ce20e0c1"),

"ns" : "recommender.data",

"key" : {

"name" : 1,

"info" : -1

},

"name" : "name\_1\_info\_-1"

}

]

下面的排序将用到上面的索引

最后一行的"name" : "ll4la"实际上是"name" : "114la"(就是将数字一写成了字母l)，但是我录入的时候写成了"name" : "ll4la"，是我写错了，但是排序的结果是对的。

> db.data.find({"info.url": /http:\*/i}).sort({name:1, info: -1});

{ "\_id" : ObjectId("4befb711b0e29ba1ce20e0bc"), "name" : "1616", "info" : { "url" : "http://www.1616.net/", "city" : "beijing" } }

{ "\_id" : ObjectId("4befb723b0e29ba1ce20e0bd"), "name" : "hao123", "info" : { "url" : "http://www.hao123.com/", "city" : "beijing" } }

{ "\_id" : ObjectId("4befb740b0e29ba1ce20e0be"), "name" : "ll4la", "info" : { "url" : "http://www.114la.com/", "city" : "dongguan" } }

**MongoDB组合索引规则**

If you have a compound index on multiple fields, you can use it to query on the beginning subset of fields. So if you have an index on

a,b,c

you can use it query on

a

a,b

a,b,c

如果用过MySQL的话，看起来是不是很熟悉，原理跟MySQL是一样的。

### 5.唯一索引

往表data中插入一条记录。

> db.data.insert({firstname: "cai", lastname: "huafeng"});

由于表data中只有一记录有字段firstname及lastname，其它的行均没有相应的值，也就是均为null，为null就说明是相同的，而唯一索引是不允许有相同的值的，所以下面创建唯一组合索引时报错了。

所以建立唯一索引时，不管是对单个字段还是多个字段建立索引，则最好每一行均有此字段，否则会报错。

> db.data.find();

{ "\_id" : ObjectId("4bee745a0863b1c233b8b7ea"), "name" : "caihuafeng" }

{ "\_id" : ObjectId("4bee745f0863b1c233b8b7eb"), "website" : "1616.net" }

{ "\_id" : ObjectId("4bee804ba23d558eb6687117"), "x" : 6, "name" : "caihuafeng1" }

{ "\_id" : ObjectId("4bee804ba23d558eb6687118"), "x" : 6, "name" : "caihuafeng2" }

{ "\_id" : ObjectId("4bee804ba23d558eb6687119"), "x" : 6, "name" : "caihuafeng3" }

{ "\_id" : ObjectId("4bee804ba23d558eb668711a"), "x" : 6, "name" : "caihuafeng4" }

{ "\_id" : ObjectId("4bee804ba23d558eb668711b"), "x" : 6, "name" : "caihuafeng5" }

{ "\_id" : ObjectId("4bee804ba23d558eb668711c"), "x" : 6, "name" : "caihuafeng6" }

{ "\_id" : ObjectId("4bee804ba23d558eb668711d"), "x" : 6, "name" : "caihuafeng7" }

{ "\_id" : ObjectId("4bee804ba23d558eb668711e"), "x" : 6, "name" : "caihuafeng8" }

{ "\_id" : ObjectId("4bee804ba23d558eb668711f"), "x" : 6, "name" : "caihuafeng9" }

{ "\_id" : ObjectId("4bee804ba23d558eb6687120"), "x" : 6, "name" : "caihuafeng10" }

{ "\_id" : ObjectId("4befb711b0e29ba1ce20e0bc"), "name" : "1616", "info" : { "url" : "http://www.1616.net/", "city" : "beijing" } }

{ "\_id" : ObjectId("4befb723b0e29ba1ce20e0bd"), "name" : "hao123", "info" : { "url" : "http://www.hao123.com/", "city" : "beijing" } }

{ "\_id" : ObjectId("4befb740b0e29ba1ce20e0be"), "name" : "ll4la", "info" : { "url" : "http://www.114la.com/", "city" : "dongguan" } }

{ "\_id" : ObjectId("4befc51ab0e29ba1ce20e0c2"), "firstname" : "cai", "lastname" : "huafeng" }

> db.data.ensureIndex({firstname: 1, lastname: 1}, {unique: true});

E11000 duplicate key error index: recommender.data.$firstname\_1\_lastname\_1 dup key: { : null, : null }

**下面我们用另外一个表person来进行测试**

> db.person.ensureIndex({firstname:1, lastname: 1},{unique: true});

> db.person.insert({firstname: 'cai', lastname: 'huafeng'});

第二次插入同样值的时候报错了，说明唯一索引生效了，其实跟MySQL里面是一样的。

> db.person.insert({firstname: 'cai', lastname: 'huafeng'});

E11000 duplicate key error index: recommender.person.$firstname\_1\_lastname\_1 dup key: { : "cai", : "huafeng" }

### 6.唯一索引中的重复值处理

删除上述5中的索引，插入两行一样的记录

> db.person.dropIndexes();

{

"nIndexesWas" : 2,

"msg" : "non-\_id indexes dropped for collection",

"ok" : 1

}

> db.person.find();

{ "\_id" : ObjectId("4befcda6b0e29ba1ce20e0cf"), "firstname" : "cai", "lastname" : "huafeng" }

> db.person.insert({firstname: 'cai', lastname: 'huafeng'});

> db.person.find();

{ "\_id" : ObjectId("4befcda6b0e29ba1ce20e0cf"), "firstname" : "cai", "lastname" : "huafeng" }

{ "\_id" : ObjectId("4befcef0b0e29ba1ce20e0d1"), "firstname" : "cai", "lastname" : "huafeng" }

如果现在直接在字段firstname及lastname上面创建唯一组合索引的时候肯定会报错，我们来试一试:

> db.person.ensureIndex({firstname: 1, lastname: 1}, {unique: true});

E11000 duplicate key error index: recommender.person.$firstname\_1\_lastname\_1 dup key: { : "cai", : "huafeng" }

查看表person的索引，我们可以看到，新创建的索引没有生成。

> db.person.getIndexes();

[

{

"name" : "\_id\_",

"ns" : "recommender.person",

"key" : {

"\_id" : 1

}

}

]

可以在第二个json对象加入一项dropDups: true，这样在创建唯一组合索引的时候不会报错，保留文档中第一个重复的值，其它重复的值均删除。

再次测试一下，加入dropDups选项，虽然报错了，但是唯一组合索引已经建立了。

> db.person.ensureIndex({firstname: 1, lastname: 1}, {unique: true, dropDups: true});

E11000 duplicate key error index: recommender.person.$firstname\_1\_lastname\_1 dup key: { : "cai", : "huafeng" }

> db.person.getIndexes();

[

{

"name" : "\_id\_",

"ns" : "recommender.person",

"key" : {

"\_id" : 1

}

},

{

"\_id" : ObjectId("4befcfd9b0e29ba1ce20e0d3"),

"ns" : "recommender.person",

"key" : {

"firstname" : 1,

"lastname" : 1

},

"name" : "firstname\_1\_lastname\_1",

"unique" : true,

"dropDups" : true

}

]

再次查询表person中的记录，发现重复的记录已经自动删除了。

> db.person.find();

{ "\_id" : ObjectId("4befcda6b0e29ba1ce20e0cf"), "firstname" : "cai", "lastname" : "huafeng" }

MongoDB官方文档的说明

A unique index cannot be created on a key that has duplicate values. If you would like to create the index anyway, keeping the first document the database indexes and deleting all subsequent documents that have duplicate values, add the dropDups option.

db.things.ensureIndex({firstname : 1}, {unique : true, dropDups : true})

### 7.删除索引

#### a.删除某个表中的所有索引

To delete all indexes on the specified collection:

db.collection.dropIndexes();

#### b.删除某个表中的单一索引

To delete a single index:

db.collection.dropIndex({x: 1, y: -1})

> db.data.dropIndex({firstname: 1, lastname: 1});

{ "nIndexesWas" : 6, "ok" : 1 }

Running directly as a command without helper:

// note: command was "deleteIndexes", not "dropIndexes", before MongoDB v1.3.2

// remove index with key pattern {y:1} from collection foo

db.runCommand({dropIndexes:'foo', index : {y:1}})

// remove all indexes:

db.runCommand({dropIndexes:'foo', index : '\*'})

> db.person.ensureIndex({firstname: 1, lastname: 1});

> db.runCommand({dropIndexes:'person', index:{firstname:1, lastname:1}});

{ "nIndexesWas" : 2, "ok" : 1 }

### 8.索引管理

通过执行getIndexes()函数，可以获得当前数据集中所有的索引：

> db.users.getIndexes()

[

{

"v" : 1,

"key" : {

"\_id" : 1

},

"ns" : "test.users",

"name" : "\_id\_"

},

{

"v" : 1,

"key" : {

"age" : 1,

"username" : 1

},

"ns" : "test.users",

"name" : "age\_1\_username\_1"

},

{

"v" : 1,

"key" : {

"username" : 1,

"age" : 1

},

"ns" : "test.users",

"name" : "username\_1\_age\_1"

},

{

"v" : 1,

"key" : {

"username" : 1

},

"unique" : true,

"ns" : "test.users",

"name" : "username\_1"

}

]

其中的“name”字段可以用于对索引的删除操作：

> db.users.dropIndex("username\_1\_age\_1")

就将删除{"username" : 1, "age" : 1}这个索引。

Author: [Chenbin](http://heychenbin.github.io/mongodb/mongodb.index.html)

Created: 2013-10-26 Sat 14:12

[Emacs](http://www.gnu.org/software/emacs/) 24.3.1 ([Org](http://orgmode.org) mode 8.2.1)

[Validate](http://validator.w3.org/check?uri=referer)

延伸阅读:

<http://www.mongodb.org/display/DOCS/Indexes#Indexes-DocumentsasKeys>

<http://www.mongodb.org/display/DOCS/min+and+max+Query+Specifiers>

<http://www.mongodb.org/display/DOCS/Advanced+Queries>

## 第10章 MongoDB性能优化

### [mongodb性能优化](http://www.cnblogs.com/dennisit/archive/2013/01/27/2879017.html)

建立索引是优化数据库最直接的手段.遵循以下索引优化原则,可以建立比较高效和合理的索引.

[复制代码](javascript:void(0);)

在索引中包含条件的所有列,可以使用索引形成的屏蔽来拒绝结果集中不合适的行

对于需要排序的引用列,适当地创建索引可以避免排序

考虑到管理上的开销,应避免在索引中使用多于5个的列

对于多列索引,将查询中引用最多的列放在定义的前面

不要在索引中包含经常修改或进行插入、删除的列(主关键字和外来关键字除外)

“$”符号不可以作为索引的首字母,”.”不能在索引名的任何位置出现.

[复制代码](javascript:void(0);)

**索引管理**

1.建立索引的函数:ensureIndex();

[复制代码](javascript:void(0);)

eg.在name上建立索引1(升序)，-1(降序)，默认为升序.

db.person.ensureIndex( { name : 1 } );

当系统已有大量数据时,创建索引非常耗时,需要在后头执行,只需要指定background:true即可.

db.user.ensureIndex( { age : 1 } , { background : true } );

建立索引后,同一条查询语句比较2次扫描的记录条数.

> db.user.find( { name:"user5" } ).explain();

[复制代码](javascript:void(0);)

2.获取集合中的索引信息

db.person.getIndexKeys();

db.person.getIndexes();

3.创建唯一索引

只需要在ensureIndex命令中指定"unique:true"即可创建唯一索引

db.user.ensureIndex( {age:1} , {unique:true} );

创建了唯一索引的字段不允许插入相同的值.

4.创建组合索引

db.user.ensureIndex({“addr.city”:1 , “addr.state”:1}); //在user集合的addr.city和addr.state列上创建组合索引.

5.删除索引

db.user.dropIndexes(); //删除usre表中的除过id外的所有索引

db.user.dropIndex( {age:1} ); //删除user表中的age索引

**性能优化**

explain 查看执行计划

MongoDB提供了一个explain命令让我们获知系统如何处理查询请求,利用explain命令我们可以很好地观察系统

如何使用索引来加快检索,同时可以针对性优化索引.

优化器profile

MongoDB Database Profiler是一种慢查询日志功能,可以作为我们优化数据库的依据.

开启Profiling功能,有2种方式可以控制Profiling的开关盒级别。

启动MonggoDB时加上-profile=级别即可

在客户端调用db.setProfilingLevel(级别)命令来实时配置。

Profiler信息保存在system.profile中.我们可以通过db.getProfilingLevel()命令来获取当前的Profile级别.

profile的级别有3个,分别是0、1、2.默认没有开启

0 - 表示不开启

1 - 表示记录慢命令(默认为>100ms)

2 - 表示记录所有命令

profile在级别1时会记录慢命令,默认值为100ms,设置慢记录的方法有2个

1.通过slowms启动参数配置

2.db.setProfilingLevel(1,设置ms参数)时加上第二个参数.

**性能监控**

通过对数据库的性能监控,能够更好的了解数据库的工作状态,从而进行优化.

几个性能监控的工具:

1.mongosniff

此功能可以从底层监控到底有哪些命令发送给mongoDB去执行,

./mongosniff --source net lo

它是实时动态监视的,需要打开另一个客户端进行命令操作.可以讲这些数据输出到一个日志文件中,

那么就可以保留下所有数据库操作的历史记录,对于后期的性能分析和安全审计等工作将是一个巨大的贡献.

“lo”即Loopback,指IP数据包会送到本机上,通常使用的是127.0.0.1作为回送地址,Loopback通常用于检测网络配置和本地程序之间通信.

2.mongostat

此工具可以快速的查看某组运行中的MongoDB实例的统计信息,也需要在打开另一个客户端进行命令操作

./mongostat

3.db.serverStatus命令

该命令可以查看服务器运行状态,是最基础的查看实例运行状态的命令

4.db.stats命令

MongoDB不但可以查看实例级运行状态信息,而且可以通过运行”db.stats”命令查看特定数据库的详细运行状态,更细粒度地分析故障.

**优化方案**

优化方案1：创建索引

在查询条件的字段上,或者排序条件的字段上创建索引,可以显著提高执行效率.

db.user.ensureIndex( { name:1 } );

优化方案2 : 限定返回结果条数

使用limit()限定返回结果集的大小,可以减少database server的资源消耗,可以减少网络传输数据量.

db.user.find().sort( {age:-1} ).limit(5);

优化方案3:查询使用到的字段,不查询所有字段

db.user.find( {} , {name:1} ).sort( { age:-1} ).limit(5) ;

优化方案4:采用capped collection

capped Collections比普通Collections的读写效率高

优化方案5:采用Profiling

Profiling功能肯定是会影响效率的,但是不太严重,原因是他使用的是system.profile来记录,system.profile是一个capped collection 这种collection在操作上有一些限制和特点,但是效率更高.

转载请注明出处:[<http://www.cnblogs.com/dennisit/archive/2013/01/27/2879017.html>]

## 第11章 MongoDB复制集

### [MongoDB复制集简介（一）--初步认识](http://blog.csdn.net/luohongqing/article/details/12578235)

在NoSQL数据库中，复制功能是作为一项最基本的特性存在的。复制为数据提供了冗余，提高了数据可用性。在多台服务器拥有相同数据集的情况下，复制可以防止因一台机器故障导致数据丢失。如果拥有相同数据集的服务器分布在不同的数据中心，复制还可以提供灾难备份功能。另外，我们还可以将复制功能用于备份、报表、读写分离场景。  
MongoDB作为NoSQL数据库的典型代表，既提供简单的主从式复制，也提供具有灵活的复制集功能。MongoDB复制集具有数据冗余、分离读压力、自动恢复、维护简单、支持灾难恢复等特性。

一、基本概念

主机：从客户端接收所有的写请求，同时在local数据库中oplog.rs文档集中记录所有数据变化日志。一个复制集只能有一个主机。

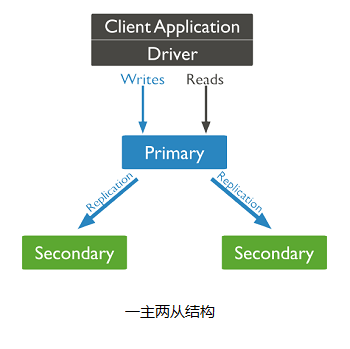
从机：复制主机的日志并应用到本机数据集上，以保证数据与主机同步。当主机不可用时，复制集自动选择一个从机做主机。

仲裁者:不存储数据，不能成为主机，仅仅用于有偶数个复制集，也仅仅在投票产生主机时发挥作用。

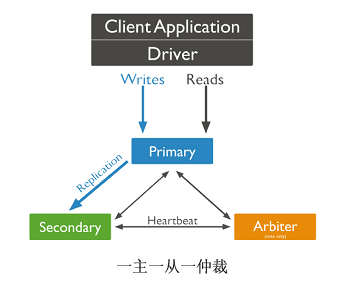
隐藏成员：具有主机数据的完整复制，但是对客户端不可见的成员。隐藏成员优先级为0，不能成为主机，但可以参与投票。

二、复制集常见架构

复制集的架构将直接影响复制集的容量和性能。一个标准的复制集一般为三个成员，具有数据冗余和容错能力，如下图所示。



当然，我们也可以搭建一个只有一个主机、一个从机、一个仲裁者的架构，如下图所示。



考虑复制集中成员个数时，一般需要考虑以下几个因素：

* 成员个数应为奇数，奇数个成员能够保证复制集选出一个主机。如果成员个数为偶数，应需要增加一个仲裁器。
* 需要考虑容错性。容错性是指复制集中最多允许几个成员不可用，但是这时复制集依然能够选出主机。拥有不同数量复制集中具有不同的容错性。

|  |  |  |
| --- | --- | --- |
| 成员个数 | 选举主机需要的个数 | 容错性 |
| 3 | 2 | 1 |
| 4 | 3 | 1 |
| 5 | 3 | 2 |
| 6 | 4 | 2 |

* 为读压力准备专门成员

在考虑成员的分布时，为灾难时能够恢复数据，至少应将一个成员分布在另一个数据中心。当复制集成员分布在多个数据中心时，应保证大多数参与选举主机的成员在一个物理数据中心，以防止网络通讯故障。

三、搭建复制集

1、启动三个节点，复制集名称为test

在shell下执行下面命令启动三个mongod进程

./mongod --dbpath /mongodb/data/db\_1 --port 27117 --replSet test --logpath /mongodb/log/mongod\_1.log --fork  
./mongod --dbpath /mongodb/data/db\_2 --port 27217 --replSet test --logpath /mongodb/log/mongod\_2.log --fork  
./mongod --dbpath /mongodb/data/db\_3 --port 27317 --replSet test --logpath /mongodb/log/mongod\_3.log --fork

使用命令ps -ef|grep mongodb，得到以下结果，确保三个mongod进程已经启动。

root 2311 1 91 05:42 ? 00:02:37 ./mongod --dbpath /mongodb/data/db\_1 --port 27117 --replSet test --logpath /mongodb/log/mongod\_1.log --fork  
root 3244 1 39 05:44 ? 00:00:08 ./mongod --dbpath /mongodb/data/db\_2 --port 27217 --replSet test --logpath /mongodb/log/mongod\_2.log --fork  
root 3368 1 26 05:44 ? 00:00:02 ./mongod --dbpath /mongodb/data/db\_3 --port 27317 --replSet test --logpath /mongodb/log/mongod\_3.log --fork

2、 初始化复制集

使用mongo shell工具连接到端口号27117的mongod：

[root@localhost bin]# ./mongo --port 27117  
MongoDB shell version: 2.5.2  
connecting to: 127.0.0.1:27117/test  
>

执行rs.initiate()初始化复制集：

> rs.initiate()  
{  
"info2" : "no configuration explicitly specified -- making one",  
"me" : "localhost.localdomain:27117",  
"info" : "Config now saved locally. Should come online in about a minute.",  
"ok" : 1  
}  
test:SECONDARY>

我们注意到执行完rs.initiate后，shell的提示符发生了变化，表示当前shell连接的是test的从机。使用rs.conf()查看复制集当前配置：

test:SECONDARY> rs.conf()  
{  
"\_id" : "test",  
"version" : 1,  
"members" : [  
{  
"\_id" : 0,  
"host" : "localhost.localdomain:27117"  
}  
]  
}

根据rs.conf()的输出看出，当前复制集合只有一个成员，我们使用命令将其他两个成员增加进入复制集：

test:PRIMARY> rs.add("localhost.localdomain:27217")  
{ "ok" : 1 }  
test:PRIMARY> rs.add("localhost.localdomain:27317")  
{ "ok" : 1 }  
test:PRIMARY>

我们再使用rs.conf()查看一下复制集的配置，可以看到复制集合的三个成员都已经加入。

test:PRIMARY> rs.conf()  
{  
"\_id" : "test",  
"version" : 3,  
"members" : [  
{  
"\_id" : 0,  
"host" : "localhost.localdomain:27117"  
},  
{  
"\_id" : 1,  
"host" : "localhost.localdomain:27217"  
},  
{  
"\_id" : 2,  
"host" : "localhost.localdomain:27317"  
}  
]  
}

我们使用rs.status()查看一下复制集的状态，可以观察到端口号为27117的mongod进程为主机，其他两个为从机。

test:PRIMARY> rs.status()  
{  
"set" : "test",  
"date" : ISODate("2013-10-12T10:16:57Z"),  
"myState" : 1,  
"members" : [  
{  
"\_id" : 0,  
"name" : "localhost.localdomain:27117",  
"health" : 1,  
"state" : 1,  
"stateStr" : "PRIMARY",  
"uptime" : 1053,  
"optime" : Timestamp(1381572862, 1),  
"optimeDate" : ISODate("2013-10-12T10:14:22Z"),  
"self" : true  
},  
{  
"\_id" : 1,  
"name" : "localhost.localdomain:27217",  
"health" : 1,  
"state" : 2,  
"stateStr" : "SECONDARY",  
"uptime" : 167,  
"optime" : Timestamp(1381572862, 1),  
"optimeDate" : ISODate("2013-10-12T10:14:22Z"),  
"lastHeartbeat" : ISODate("2013-10-12T10:16:56Z"),  
"lastHeartbeatRecv" : ISODate("2013-10-12T10:16:56Z"),  
"pingMs" : 0,  
"syncingTo" : "localhost.localdomain:27117"  
},  
{  
"\_id" : 2,  
"name" : "localhost.localdomain:27317",  
"health" : 1,  
"state" : 2,  
"stateStr" : "SECONDARY",  
"uptime" : 155,  
"optime" : Timestamp(1381572862, 1),  
"optimeDate" : ISODate("2013-10-12T10:14:22Z"),  
"lastHeartbeat" : ISODate("2013-10-12T10:16:57Z"),  
"lastHeartbeatRecv" : ISODate("2013-10-12T10:16:56Z"),  
"pingMs" : 0,  
"syncingTo" : "localhost.localdomain:27117"  
}  
],  
"ok" : 1  
}

3、手工控制切换主机

当我们给不同的复制成员设置上优先级后，mongodb复制集将按照优先级的大小顺序选择主机，优先级数字越大，表示越有可能成为主机。我们将端口号为27317的对应成员的优先级设置为3,端口号为27217对应的成员优先级设置为2，端口号27117的对应成员优先级设置为1。

cfg = rs.conf()

cfg.members[0].priority = 1  
cfg.members[1].priority = 2  
cfg.members[2].priority = 3

rs.reconfig(cfg)

执行rs.status()查看复制集优先级配置变化后的复制集状态，发现端口号为27317的对应成员变为主机了，实现了按照优先级控制主机切换。

rs.status()

{  
"set" : "test",  
"date" : ISODate("2013-10-12T10:38:02Z"),  
"myState" : 2,  
"syncingTo" : "localhost.localdomain:27317",  
"members" : [  
{  
"\_id" : 0,  
"name" : "localhost.localdomain:27117",  
"health" : 1,  
"state" : 2,  
"stateStr" : "SECONDARY",  
"uptime" : 2318,  
"optime" : Timestamp(1381574271, 1),  
"optimeDate" : ISODate("2013-10-12T10:37:51Z"),  
"errmsg" : "syncing to: localhost.localdomain:27317",  
"self" : true  
},  
{  
"\_id" : 1,  
"name" : "localhost.localdomain:27217",  
"health" : 1,  
"state" : 2,  
"stateStr" : "SECONDARY",  
"uptime" : 11,  
"optime" : Timestamp(1381574271, 1),  
"optimeDate" : ISODate("2013-10-12T10:37:51Z"),  
"lastHeartbeat" : ISODate("2013-10-12T10:38:01Z"),  
"lastHeartbeatRecv" : ISODate("2013-10-12T10:38:01Z"),  
"pingMs" : 1,  
"lastHeartbeatMessage" : "syncing to: localhost.localdomain:27317",  
"syncingTo" : "localhost.localdomain:27317"  
},  
{  
"\_id" : 2,  
"name" : "localhost.localdomain:27317",  
"health" : 1,  
"state" : 1,  
"stateStr" : "PRIMARY",  
"uptime" : 10,  
"optime" : Timestamp(1381574271, 1),  
"optimeDate" : ISODate("2013-10-12T10:37:51Z"),  
"lastHeartbeat" : ISODate("2013-10-12T10:38:02Z"),  
"lastHeartbeatRecv" : ISODate("2013-10-12T10:38:01Z"),  
"pingMs" : 0  
}  
],  
"ok" : 1  
}

### [MongoDB复制集简介（二）--内部信息](http://blog.csdn.net/luohongqing/article/details/12778879)

MongoDB复制集简介（二）

MongoDB内部有一个Local数据库，其中存储了与复制有关的内部信息。Local数据库本身对复制不可见，即其中的集合、文档不会被复制到其他机器。

Local数据库与复制集有关的集合如下：

local.system.replset：存储了复制集的配置信息，我们使用rs.conf()查看内容，也可以直接查询该集合。

local.oplog.rs：存储了用于复制的日志信息，该集合是一个固定大小、循环使用的集合。

local.replset.minvalid：存储了用于跟踪复制状态的内部信息  
local.slaves：存储了复制集的成员信息以及最近同步的时间点

test:PRIMARY> use local  
switched to db local

test:PRIMARY> show collections   
oplog.rs  
replset.minvalid  
slaves  
startup\_log  
system.indexes  
system.replset

一、复制集配置

MongoDB复制集配置主要有\_id、version、members和属性settings组成，可以通过rs.conf()查看，也可以直接查询local.system.replset集合。

test:SECONDARY> use local  
switched to db local  
test:SECONDARY> db.system.replset.find()  
  
{ "\_id" : "test",   
"version" : 6,   
"members" : [   
{ "\_id" : 1, "host" : "localhost.localdomain:27217", "priority" : 2 },   
{ "\_id" : 2, "host" : "localhost.localdomain:27317", "priority" : 3 },   
{ "\_id" : 3, "host" : "localhost.localdomain:27117" }  
]   
}

其中，\_id属性表示了复制集的名字，与启动mongod进程时指定的复制集名字相同；members为一个数组，复制集中的每个成员都作为一个文档被包含在这个数组中。members具有以下属性：

{  
\_id : <ordinal>,  
host : hostname<:port>,  
<arbiterOnly : <boolean>,>  
<buildIndexes : <boolean>,>  
<hidden : <boolean>,>  
<priority: <priority>,>  
<tags: { <document> },>  
<slaveDelay : <number>,>  
<votes : <number>>  
}

其中host指出成员所在的主机名和端口号，属性arbiterOnly表示该成员是否仲裁者，默认为false。属性buildIndexes表示mongod进程是否在该成员上构建索引，默认为true。在成员仅仅用于备份或者不接受查询的情况下，忽略索引是有帮助的。属性hidden表示成员是否可见，包括读操作都不能到达该成员，默认为false。属性priority用于选举主机，该值越大表示优先级越高，越有可能成为主机，默认值为1。属性tags用于一组标记，用于描述成员的用途。属性slaveDelay表示该成员延迟几秒从主机复制。votes一般1或0，表示该成员是否参与主机选举。

我们可以直接修改members的成员属性对复制集进行配置，如修改优先级（注意members数组下标从0开始，与\_id不是一个含义）：

cfg = rs.conf()  
cfg.members[0].priority = 0.5  
cfg.members[1].priority = 2  
cfg.members[2].priority = 2  
rs.reconfig(cfg)

复制集配置中settings具有以下属性

settings: {  
<getLastErrorDefaults : <lasterrdefaults>,>  
<chainingAllowed : <boolean>,>  
<getLastErrorModes : <modes>>  
}

其中getLastErrorDefaults用于设置getLastError返回的默认值。chainingAllowed表示该成员是否可以级联从其他从机复制，默认为true。getLastErrorModes描述了名字和成员组合用于应用层使用getLastError方法对是否成功写入数据的确认。

二、复制集日志

MongoDB的复制集日志是通过local数据库下的oplog.rs保存的，oplog.rs是一个固定大小的集合，可以在启动mongodb时通过参数oplogSize指定大小。oplog.rs结构如下：

{ ts : ..., h: ..., v: ..., op: ..., ns: ..., o: ... o2: ... }

其中 ts为时间戳，h为hash值，v为oplog的版本号，op表示操作，可能的取值如下：

"i" ： 表示插入操作  
"u" ：表示更新操作  
"d" ：表示删除操作  
"c" ：表示db cmd，表示运行db.runCommand()  
"n" ：表示无操作，一般用于复制集本身的配置修改或初始化。

oplog.rs中ns表示命名空间，也就是操作的集合名，o表示文档的具体内容。

我们通过修改复制成员的优先级，查看一下oplog.rs的内容。

test:PRIMARY> db.oplog.rs.find().limit(1).sort( { ts : -1 } ).pretty()  
{  
"ts" : Timestamp(1382374260, 1),  
"h" : NumberLong("-2381899609645775674"),  
"v" : 2,  
"op" : "n",  
"ns" : "",  
"o" : {  
"msg" : "Reconfig set",  
"version" : 8  
}  
}

在修改之前，我们看到oplog.rs的最后一条记录内容是一条复制集配置操作，当前复制集版本号为8.

test:PRIMARY>cfg=rs.conf()  
test:PRIMARY>cfg.members[0].priority=3  
test:PRIMARY>cfg.members[1].priority=2  
test:PRIMARY>cfg.members[2].priority=1

test:PRIMARY>rs.reconfig(cfg)

修改完成以后，我们再查看一下oplog.rs的最后两条记录，发现新的配置操作已经记录到oplog.rs中，并且复制集的配置版本已经变成9。

test:SECONDARY> db.oplog.rs.find().limit(2).sort( { ts :-1 } ).pretty()db.oplog.rs.find().limit(2).sort( { ts :-1 } ).pretty()  
{  
"ts" : Timestamp(1382376712, 1),  
"h" : NumberLong("-1575134825584687483"),  
"v" : 2,  
"op" : "n",  
"ns" : "",  
"o" : {  
"msg" : "Reconfig set",  
"version" : 9  
}  
}  
{  
"ts" : Timestamp(1382374260, 1),  
"h" : NumberLong("-2381899609645775674"),  
"v" : 2,  
"op" : "n",  
"ns" : "",  
"o" : {  
"msg" : "Reconfig set",  
"version" : 8  
}  
}

我们测试在主机上插入一条记录，查看oplog的内容：

test:PRIMARY> use test

switched to db test

test:PRIMARY> db.test.insert({empno:100,name:100,address:' it is a test!!!!'})

test:PRIMARY> use local  
switched to db local  
test:PRIMARY> db.oplog.rs.find().limit(1).sort( { ts :-1 } ).pretty()  
{  
"ts" : Timestamp(1382377459, 1),  
"h" : NumberLong("5755918670421899025"),  
"v" : 2,  
"op" : "i",  
"ns" : "test.test",  
"o" : {  
"\_id" : ObjectId("526567f3a1c239e61e9bdda2"),  
"empno" : 100,  
"name" : 100,  
"address" : " it is a test!!!!"  
}  
}

我们发现oplog中已经将插入的文档完整记录下来，并且op类型为i,表示是插入操作，ns为“test.test"表示test库上的test集合。

### [MongoDB复制集简介（三）--复制集中的回滚](http://blog.csdn.net/luohongqing/article/details/13023559)

一、复制集中回滚介绍

事务回滚在关系型数据库中是常见的，在MongoDB复制集中也存在回滚。这里的回滚是指当发生主从失败切换时，前主机重新加入复制集后，将回滚失败之前的写操作。回滚仅仅针对主机失败前，主机已经完成的写操作但从机没有复制的写操作。主机失败后，从机成为主机，由于没有复制刚刚主机的写操作，新的复制集中没有刚刚的数据。这时如果老主机作为新的从机重新加入到复制集，老主机将回滚刚刚未复制的写操作，以与其他成员保持数据一致性，对已经完成复制的写操作，MongoDB不会回滚。MongoDB复制集发生回滚的概率很低，主要发生在出现网络分区的场景下。

当发生回滚时，MongoDB会将回滚数据写成一个BSON文件，放到数据库目录下的一个rollback子目录下。文件名称格式如下：  
  
<database>.<collection>.<timestamp>.bson

DBA需要考虑是放弃这些数据，还是要重新应用这些数据。 如果DBA想应用这些数据，可以手工用bsondump 读取出dump中的内容，也可以 mongorestore 命令在新的主机上应用这些数据。

为避免复制集发生回滚，应用程序可以使用“写确认”方法保证写操作已经传播给复制集中的成员。

二、模拟

下面我们做个实验，模拟一下复制集的回滚。  
我们构建一个具有主机、从机和仲裁者三个成员的复制集test。

在shell下执行下面命令启动三个mongod进程  
  
./mongod --dbpath /mongodb/data/db\_1 --port 27117 --replSet test --logpath /mongodb/log/mongod\_1.log --fork  
./mongod --dbpath /mongodb/data/db\_2 --port 27217 --replSet test --logpath /mongodb/log/mongod\_2.log --fork  
./mongod --dbpath /mongodb/data/db\_3 --port 27317 --replSet test --logpath /mongodb/log/mongod\_3.log --fork

使用mongo shell工具连接到端口号27117的mongod，执行rs.initiate()初始化复制集。

[root@localhost bin]# ./mongo --port 27117  
> rs.initiate()  
{  
"info2" : "no configuration explicitly specified -- making one",  
"me" : "localhost.localdomain:27117",  
"info" : "Config now saved locally. Should come online in about a minute.",  
"ok" : 1  
}

test:OTHER> rs.add("localhost.localdomain:27217")  
{ "ok" : 1 }  
test:PRIMARY> rs.addArb("localhost.localdomain:27317")  
{ "ok" : 1 }

在设置客户端能够从从机读取数据，在每个成员的shell下都执行db.getMongo().setSlaveOk()

在主机上插入数据  
test:PRIMARY> db.test.insert({empno:100,name:100})  
test:PRIMARY> db.test.insert({empno:101,name:101})  
test:PRIMARY> db.test.insert({empno:102,name:102})

在从机上读取，发现3条数据已经插入成功  
test:SECONDARY> use test   
switched to db test  
test:SECONDARY> db.test.find()  
{ "\_id" : ObjectId("526aa7144ff959233867ff95"), "empno" : 100, "name" : 100 }  
{ "\_id" : ObjectId("526aa7194ff959233867ff96"), "empno" : 101, "name" : 101 }  
{ "\_id" : ObjectId("526aa7194ff959233867ff97"), "empno" : 102, "name" : 102 }  
test:SECONDARY>

关闭从机进程，准备模拟从机未复制主机的写操作：  
test:SECONDARY> use admin  
switched to db admin  
test:SECONDARY> db.shutdownServer()  
2013-10-25T13:19:33.447-0400 DBClientCursor::init call() failed  
server should be down...  
2013-10-25T13:19:33.461-0400 trying reconnect to 127.0.0.1:27217  
2013-10-25T13:19:33.464-0400 reconnect 127.0.0.1:27217 failed couldn't connect to server 127.0.0.1:27217  
2013-10-25T13:19:33.470-0400 trying reconnect to 127.0.0.1:27217  
2013-10-25T13:19:33.470-0400 reconnect 127.0.0.1:27217 failed couldn't connect to server 127.0.0.1:27217

在主机上查看复制集状态，确认从机已经不可达到  
test:PRIMARY> rs.status()  
{  
"set" : "test",  
"date" : ISODate("2013-10-25T17:20:50Z"),  
"myState" : 1,  
.......  
{  
"\_id" : 1,  
"name" : "localhost.localdomain:27217",  
"health" : 0,  
"state" : 8,  
"stateStr" : "(not reachable/healthy)",  
"uptime" : 0,  
"optime" : Timestamp(1382721305, 3),  
"optimeDate" : ISODate("2013-10-25T17:15:05Z"),  
"lastHeartbeat" : ISODate("2013-10-25T17:20:49Z"),  
"lastHeartbeatRecv" : ISODate("2013-10-25T17:19:31Z"),  
"pingMs" : 0,  
"syncingTo" : "localhost.localdomain:27117"  
},  
.......  
],  
"ok" : 1  
}  
test:PRIMARY>

主机继续插入数据，并查询确保插入成功  
test:PRIMARY> use test  
switched to db test  
test:PRIMARY> db.test.insert({empno:103,name:103})  
test:PRIMARY> db.test.insert({empno:104,name:104})  
test:PRIMARY> db.test.insert({empno:105,name:105})  
test:PRIMARY> db.test.find()  
{ "\_id" : ObjectId("526aa7144ff959233867ff95"), "empno" : 100, "name" : 100 }  
{ "\_id" : ObjectId("526aa7194ff959233867ff96"), "empno" : 101, "name" : 101 }  
{ "\_id" : ObjectId("526aa7194ff959233867ff97"), "empno" : 102, "name" : 102 }  
{ "\_id" : ObjectId("526aa9204ff959233867ff98"), "empno" : 103, "name" : 103 }  
{ "\_id" : ObjectId("526aa9294ff959233867ff99"), "empno" : 104, "name" : 104 }  
{ "\_id" : ObjectId("526aa9304ff959233867ff9a"), "empno" : 105, "name" : 105 }  
test:PRIMARY>

kill掉主机进程，后启动原从机进程，即端口号为27217的mongod，该进程启动后将成为主机。我们查看test集合，发现empno为103、104、105的数据在新主机中没有  
[root@localhost bin]# ./mongod --dbpath /mongodb/data/db\_2 --port 27217 --replSet test --logpath /mongodb/log/mongod\_2.log --fork  
[root@localhost bin]# ./mongo --port 27217  
test:PRIMARY> use test   
switched to db test  
test:PRIMARY> db.test.find()   
{ "\_id" : ObjectId("526aa7144ff959233867ff95"), "empno" : 100, "name" : 100 }  
{ "\_id" : ObjectId("526aa7194ff959233867ff96"), "empno" : 101, "name" : 101 }  
{ "\_id" : ObjectId("526aa7194ff959233867ff97"), "empno" : 102, "name" : 102 }  
test:PRIMARY>

此时，启动老主机将发生回滚，老主机将对empno为103、104、105的数据进行删除。  
[root@localhost bin]# ./mongod --dbpath /mongodb/data/db\_1 --port 27117 --replSet test --logpath /mongodb/log/mongod\_1.log --fork  
[root@localhost bin]# ./mongo --port 27117  
test:SECONDARY> db.getMongo().setSlaveOk()  
test:SECONDARY> use test  
switched to db test  
test:SECONDARY> db.test.find()  
{ "\_id" : ObjectId("526aa7144ff959233867ff95"), "empno" : 100, "name" : 100 }  
{ "\_id" : ObjectId("526aa7194ff959233867ff96"), "empno" : 101, "name" : 101 }  
{ "\_id" : ObjectId("526aa7194ff959233867ff97"), "empno" : 102, "name" : 102 }  
test:SECONDARY>

我们在/mongodb/log/mongod\_1.log中发现关于rollback的日志：  
2013-10-25T13:33:48.171-0400 [rsBackgroundSync] replSet we are ahead of the sync source, will try to roll back  
2013-10-25T13:33:48.209-0400 [rsBackgroundSync] replSet rollback 0  
2013-10-25T13:33:48.209-0400 [rsBackgroundSync] replSet ROLLBACK  
2013-10-25T13:33:48.209-0400 [rsBackgroundSync] replSet rollback 1  
2013-10-25T13:33:48.209-0400 [rsBackgroundSync] replSet rollback 2 FindCommonPoint  
2013-10-25T13:33:48.211-0400 [rsBackgroundSync] replSet info rollback our last optime: Oct 25 13:24:00:1  
2013-10-25T13:33:48.211-0400 [rsBackgroundSync] replSet info rollback their last optime: Oct 25 13:15:05:3  
2013-10-25T13:33:48.211-0400 [rsBackgroundSync] replSet info rollback diff in end of log times: 535 seconds  
2013-10-25T13:33:48.211-0400 [rsBackgroundSync] replSet rollback found matching events at Oct 25 13:15:05:3  
2013-10-25T13:33:48.211-0400 [rsBackgroundSync] replSet rollback findcommonpoint scanned : 4  
2013-10-25T13:33:48.211-0400 [rsBackgroundSync] replSet replSet rollback 3 fixup  
2013-10-25T13:33:48.214-0400 [rsBackgroundSync] replSet rollback 3.5  
2013-10-25T13:33:48.215-0400 [rsBackgroundSync] replSet rollback 4 n:3  
2013-10-25T13:33:48.215-0400 [rsBackgroundSync] replSet minvalid=Oct 25 13:15:05 526aa719:3  
2013-10-25T13:33:48.217-0400 [rsBackgroundSync] build index local.replset.minvalid { \_id: 1 }  
2013-10-25T13:33:48.219-0400 [rsBackgroundSync] build index done. scanned 0 total records. 0.001 secs  
2013-10-25T13:33:48.219-0400 [rsBackgroundSync] replSet rollback 4.6  
2013-10-25T13:33:48.219-0400 [rsBackgroundSync] replSet rollback 4.7  
2013-10-25T13:33:48.221-0400 [rsBackgroundSync] replSet rollback 5 d:6 u:0  
2013-10-25T13:33:48.221-0400 [rsBackgroundSync] replSet rollback 6  
2013-10-25T13:33:48.222-0400 [rsBackgroundSync] replSet rollback 7  
2013-10-25T13:33:48.222-0400 [rsBackgroundSync] replSet rollback done

在/mongodb/data/db\_1/rollback目录下看到bson文件：  
[root@localhost bin]# ls -al /mongodb/data/db\_1/rollback/  
total 12  
drwxr-xr-x 2 root root 4096 Oct 25 13:33 .  
drwxr-xr-x 4 root root 4096 Oct 25 13:33 ..  
-rw-r--r-- 1 root root 153 Oct 25 13:33 test.test.2013-10-25T17-33-48.0.bson  
[root@localhost bin]#

我们使用工具bsondump将bson转换为json文件，就可以看到empno为103、104、105的回滚数据。  
[root@localhost bin]# ./bsondump /mongodb/data/db\_1/rollback/test.test.2013-10-25T17-33-48.0.bson >test\_rollback.json  
3 objects found  
[root@localhost bin]# more test\_rollback.json  
{ "\_id" : ObjectId( "526aa9204ff959233867ff98" ), "empno" : 103, "name" : 103 }  
{ "\_id" : ObjectId( "526aa9294ff959233867ff99" ), "empno" : 104, "name" : 104 }  
{ "\_id" : ObjectId( "526aa9304ff959233867ff9a" ), "empno" : 105, "name" : 105 }  
[root@localhost bin]#

三、小结

我们通过实验发现，只要在主机插入时，从机没有接收到的数据主从切换后都将被回滚。作为DBA，如果发生了主从切换，需要关注是否发生了回滚。如果有回滚发生，我们需要考虑如何处理回滚数据。

本实验中设置了一个仲裁器，主要是用于保证在老从机关闭时集群还能保证可用，且主机此后插入的数据没有被任何从机复制。

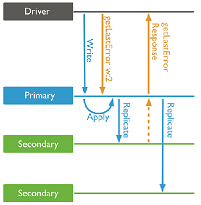
### [MongoDB复制集简介（四）--复制集的写关注](http://blog.csdn.net/luohongqing/article/details/13292363)

一、概念介绍

“Write Concern（写关注）”描述了当系统反馈写成功后MongoDB提供数据持久性保证程度，写关注越强，MongoDB提供的保证程度越高。当写操作具有弱写关注时，写操作会迅速返回，但是写操作可能不具有持久性；具有强写关注的客户端在发出写操作后需要等待MongoDB确认写操作成功。在单机环境下，Journaled为最高写关注级别，表示MongoDB只在写操作已经成功且日志已经写到磁盘后，才确认写成功；Acknowledged为次一等级的写关注，表示MongoDB已经确认收到了正确的写操作，应用可以排除网络故障或者重复键值等错误，这是MongoDB默认的写关注。

在复制集环境下，基本级别的写关注只能影响一台机器上的写操作。应用程序可以通过在getLastError提供w参数的不同配置，将复制集的写关注级别设置为Replica Acknowledged。通过Replica Acknowledged，我们可以保证写操作已经传播到复制集的成员中。

如下图所示，在一个3节点的复制集中，客户端在发出写请求后使用带w：2的getLastError进行写操作确认。w:2表示至少写操作已经传播到了复制集中的两个成员（包括主机），图中在第二成员已经收到主机传播来的日志且已经获得应用后对getLastError做出反馈，不需要等待日志传播到第三成员。如果我们将w的值设置超过了当前拥有数据拷贝的复制集成员个数，会导致getLastError被阻塞，直到其他数据成员变得可用满足了w的设置要求。当然，我们可以对getLastError设置一个超时阈值，以防止getLastError一直被阻塞。



在复制集配置中，有两个设置和复制集的写关注有很大关系，一个为成员的tags，另一个为复制集的getLastErrorModes设置。通过修改这两个设置，我们可以定制自己的复制集写关注，当然tags也可以用于设置读喜好，这里就不涉及了。当我们定制写关注模式时，需要注意的是该模式需要指定一个域名和一个值，如果该值为n，则要求至少有n个成员反馈接受了写操作且这n个成员对应tag的名字与模式指定的域名相同，但是每个tag的值不同，这时才能确认写成功。

二、应用举例

1、假设一个具有5个成员的复制集，每个复制成员的tags依次如下：

cfg=rs.conf()

cfg.members[0].tags={ "use": "reporting" }  
cfg.members[1].tags={ "use": "backup" }  
cfg.members[2].tags={ "use": "application" }  
cfg.members[3].tags={ "use": "application" }  
cfg.members[4].tags={ "use": "application" }

cfg.settings = { getLastErrorModes: { use2: { "use": 2 } } }

上面的getLastErrorModes要求至少有两个不同值的“use” tag成员反馈，才给应用反馈写结果，比如第一、第二个成员反馈写操作，主机就可以给应用反馈写结果。如果第三、第四个成员反馈写操作，还不能给应用反馈，因为他们两个的"use“tag值相同，必须要等到第一或者第二个成员反馈写操作后，才能给应用反馈。

2、假设一个具有三个成员的复制集，每个复制成员的tags依次如下：

{ "disk": "ssd" }  
{ "disk": "ssd" }  
{ "disk": "ssd" }

我们可以将getLastErrorModes设置如下：

cfg.settings = { getLastErrorModes: { disk2: { "disk": 2 } } }

getLastErrorModes配置要求至少具有两个不同的disk tag值的成员反馈才向应用反馈写结果，而实际成员的tag配置完全相同，不可能具有两个不同值的成员反馈，因此这种配置将会失败，如下所示。

test:PRIMARY> rs.reconfig(cfg)  
{  
"errmsg" : "exception: mode { disk: 2.0 } requires 2 tagged with disk, but only 1 with this tag were found",  
"code" : 14831,  
"ok" : 0  
}

3、假设一个具有三个成员的复制集，每个复制成员的tags依次如下：

{ "disk": "ssd" }  
{ "disk": "san" }  
{ "disk": "spinning" }

根据上面的tags，如果我们想配置成在写操作传播到SAN后才反馈成功，这种配置方案在现有tags下无法实现。如果我们想实现上述写关注策略，需要额外的tags才能实现，如下：

cfg=rs.conf()  
cfg.members[0].tags={ "disk": "ssd" }  
cfg.members[1].tags={ "disk": "san","disksan":"san" }  
cfg.members[2].tags={ "disk": "spinning" }  
cfg.settings = { getLastErrorModes: { disksan1: { "disksan": 1 } } }

rs.reconfig(cfg)

上述配置成功后，我们可以看到复制集的配置如下，端口号27217对应的成员具有两个tag，而其他成员只有一个tag，我们可以使用disksan的tag来控制写操作必须传播到27217对应的成员

test:PRIMARY> rs.conf()  
{  
"\_id" : "test",  
"version" : 19,  
"members" : [  
{  
"\_id" : 0,  
"host" : "localhost.localdomain:27117",  
"tags" : {  
"disk" : "ssd"  
}  
},  
{  
"\_id" : 1,  
"host" : "localhost.localdomain:27217",  
"tags" : {  
"disk" : "san",  
"disksan" : "san"  
}  
},  
{  
"\_id" : 2,  
"host" : "localhost.localdomain:27317",  
"tags" : {  
"disk" : "spinning"  
}  
}  
],  
"settings" : {  
"getLastErrorModes" : {  
"disksan1" : {  
"disksan" : 1  
}  
}  
}  
}

在端口号为27217的成员可用的情况下，我们使用上述写关注模式，得到以下结果，表示写已经成功。

test:PRIMARY> use test

test:PRIMARY> db.test.insert({empno:100,name:100})  
test:PRIMARY> db.runCommand( { getLastError: 1, w: "disksan1" } )  
{  
"n" : 0,  
"lastOp" : Timestamp(1382978081, 1),  
"connectionId" : 957,  
"writtenTo" : [  
{  
"\_id" : 1,  
"host" : "localhost.localdomain:27217",  
"tags" : {  
"disk" : "san",  
"disksan" : "san"  
}  
},  
{  
"\_id" : 0,  
"host" : "localhost.localdomain:27117",  
"tags" : {  
"dis" : "ssd"  
}  
},  
{  
"\_id" : 2,  
"host" : "localhost.localdomain:27317",  
"tags" : {  
"disk" : "spinning"  
}  
}  
],  
"wtime" : 0,  
"err" : null,  
"ok" : 1  
}

如果端口号为27217的成员不可用了，此事还使用该写关注模式，会发生什么？

我们通过kill进程的方法模拟端口号为27217的成员不可用，当该mongd进程被kill掉以后，复制集的状态如下：

test:SECONDARY> rs.status()  
{  
"set" : "test",  
"date" : ISODate("2013-10-28T16:40:08Z"),  
"myState" : 2,  
"syncingTo" : "localhost.localdomain:27317",  
"members" : [  
{  
"\_id" : 0,  
"name" : "localhost.localdomain:27117",  
"health" : 1,  
"state" : 2,  
"stateStr" : "SECONDARY",  
"uptime" : 1274,  
"optime" : Timestamp(1382978081, 1),  
"optimeDate" : ISODate("2013-10-28T16:34:41Z"),  
"errmsg" : "syncing to: localhost.localdomain:27317",  
"self" : true  
},  
{  
"\_id" : 1,  
"name" : "localhost.localdomain:27217",  
"health" : 0,  
"state" : 8,  
"stateStr" : "(not reachable/healthy)",  
"uptime" : 0,  
"optime" : Timestamp(1382978081, 1),  
"optimeDate" : ISODate("2013-10-28T16:34:41Z"),  
"lastHeartbeat" : ISODate("2013-10-28T16:40:06Z"),  
"lastHeartbeatRecv" : ISODate("2013-10-28T16:39:56Z"),  
"pingMs" : 0  
},  
{  
"\_id" : 2,  
"name" : "localhost.localdomain:27317",  
"health" : 1,  
"state" : 1,  
"stateStr" : "PRIMARY",  
"uptime" : 1017,  
"optime" : Timestamp(1382978081, 1),  
"optimeDate" : ISODate("2013-10-28T16:34:41Z"),  
"lastHeartbeat" : ISODate("2013-10-28T16:40:07Z"),  
"lastHeartbeatRecv" : ISODate("2013-10-28T16:40:08Z"),  
"pingMs" : 0  
}  
],  
"ok" : 1  
}

我们使用mongo工具连接到主机，执行插入操作后再调用getLastError，并增加超时10秒参数，以防止getLastError被阻塞。执行结果显示getLastError发生超时，表示设置的写关注模式未成功，这是由于具有disksan的tag成员已经不可用，写操作不可能传播到该成员上。但是这时在主机上查询empno为109的文档时，还是能够显示查询成功的，表明主机上已经写入该数据。

test:PRIMARY> db.test.insert({empno:109,name:109})  
test:PRIMARY> db.runCommand( { getLastError: 1, w: "disksan1",wtimeout:10000 } )  
{  
"n" : 0,  
"lastOp" : Timestamp(1382978744, 1),  
"connectionId" : 964,  
"wtimeout" : true,  
"waited" : 10001,  
"writtenTo" : [  
{  
"\_id" : 2,  
"host" : "localhost.localdomain:27317",  
"tags" : {  
"disk" : "spinning"  
}  
},  
{  
"\_id" : 0,  
"host" : "localhost.localdomain:27117",  
"tags" : {  
"disk" : "ssd"  
}  
}  
],  
"err" : "timeout",  
"ok" : 1  
}  
test:PRIMARY> db.test.find({empno:109}))  
{ "\_id" : ObjectId("526e94b82e59202b33f85752"), "empno" : 109, "name" : 109 }

## 第12章 MongoDB shareing分片

### [MongoDB分片介绍（一）—基本概念](http://blog.csdn.net/luohongqing/article/details/14450771)

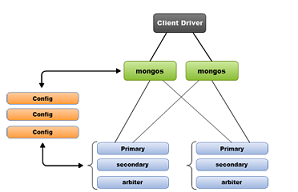
一、简介

拥有大量数据和高吞吐量应用的数据库将挑战单个服务器的容量。大量的查询将会耗尽单台机器的CPU，大量的数据将超出单台机器的存储空间，活动数据集合将超出单台机器的内存容量导致磁盘IO压力巨大。为解决上述问题，存在两种思路，一种为水平扩展，一种为垂直扩展。

垂直扩展的基本思路是增加单台机器的性能，包括CPU扩容、内存扩容、存储空间扩容。通过增加容量进行扩展存在着明显的限制，拥有大量CPU和内存的服务器过于昂贵，远远超过那些小型系统。并且，单台机器能够扩展的容量总是存在限制的，不能做到无限扩容。

水平扩展将数据打散并分布到多台物理机器上（称为多个分片），每个分片都是一套独立的数据库。无数个分片组成一个大的、单一对外的数据库系统。分片技术主要用于解决大容量数据和高吞吐应用带来的扩展难题。分片技术减少了每个分片需要处理操作个数，当集群增大时，每个新增的分片都会分担一部分操作压力，这样集群的数据容量和处理能力得到了水平扩展。分片技术同时减少了每台机器存储的数据量，当集群规模增大时，整个数据库能够容纳数据量也随之增大。

MongoDB通过分片技术支持水平扩展，MongoDB分片集群中有三个组件：Shards、Query Routers、Config servers。Shards存储实际的用户数据，为提高可用性和一致性，在生产环境下，每个分片可以是一个复制集。Query Routers是通过Mongos实例实现，它与客户应用程序接口并将应用程序的数据请求转发到对应的分片。Mongos实例在获得各个分片返回结果后，再拼装成最终结果集返回给应用程序。一个应用程序可以有多个Query Routers，以分担客户端请求的压力。Config servers存储了分片集群的元数据，包含了集群用户数据和分片之间的映射关系，Query Routers将使用这些元处理将用户请求引导到合适的分片上。一个MongoDB分片集群典型部署如下，包含三个Config servers、两个Query Routers和若干个分片。

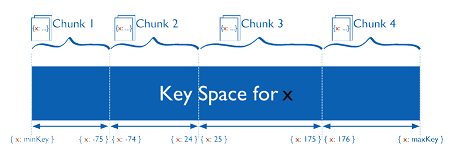


二、数据分片过程

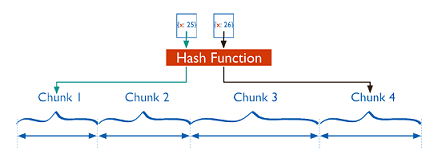
1. 分片键

当客户端应用程序通过mongos向数据库插入一条数据时，碰到的第一个问题是应该将输入插入到那个分片中。MongoDB是通过分片键实现对分片的选择的，即通过集合字段中个别字段的值确定数据应该插入到那个分片。分片键必须是一个索引或者是组合索引的一个字段，MongMongoDB支持基于范围的分片和基于HASH的分片两种方法。

对基于范围的分片，MongoDB根据分区键值范围把输入分成各个区间。假设是一个数据型键值取值范围从负无穷到正无穷，类似一个横坐标，MongoDB就把这个坐标线分成若干段，每段线表示一个区间，相互之间不重叠。



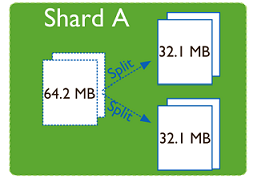
对基于HASH的分片，MongoDB首先计算分区键值的HASH值，然后用这些HASH值去创建Chunk。由于键值相邻的记录HASH不同，因此它们会落到不同的Chunk中，这一点保证了集合在分片集群上的随机分布。



2、分裂与平衡

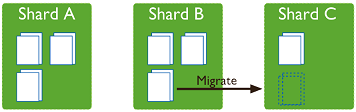
当新数据不断插入到MongoDB的时候，一个个Chunk是如何产生的呢？MongoDB是如何保证各个分片之间的Chunk数量平衡的呢？这里就要提到两个后台进程：splitting和balancer。

Splitting是一个用于防止一个Chunk过大的后台进程，当Chunk增大到超过设定的值时，MongoDB将会把这这个Chunk分裂成两个Chunk。插入、更新操作都可能引起Chunk分裂。在分裂过程中，MongoDB只涉及一些元数据的修改，不需要在分片之间移动数据。



当某个集合的各分片的数据分布不平衡时，balancer后台进程将会从拥有最多Chunk的分片将Chunk迁移到拥有最少Chunk的分片上，直到各个分片上的数据达到平衡。比如一个集合在分片1上有100个Chunk，分片2上有50个Chunk，balancer进程将会从分片1上迁移25个Chunk到分片2上，以达到两个分片的数据分布平衡。Chunk的移动都是在后台进行的，如果在移动过程中有数据请求，这些请求将会仍然转发到原始分片上。

Chunk迁移过程分为三步：首先目标分片接受来自源分片的集合数据，然后目标分片应用在迁移过程中捕获的对正在迁移的Chunk的修改，最后目标分片更新Config Server上的元数据。如果迁移过程失败，那么后台进程将会退出迁移过程，继续将Chunk留在源分片。迁移成功后，MongoDB才会删除源分片上的Chunk。



三、元数据

Config Server中保存了关于分片的所有元数据，这些元数据反映了分片的状态信息和组织信息。元数据包含了分片列表、Chunk列表以及分片的配置参数设置等，都存放在Config Server 中的一个名称为Config的数据库中。我们不建议大家手工直接维护Config数据库中的数据，应通过分片相关的命令来维护这些数据。

mongos实例在启动或者重启时将会从Config Server中读取元数据并保存在自己的内存中。当管理员执行命令修改Config Server中的元数据或者Chunk完成迁移后时，mongos实例同时也会在内存中更新相关数据。

在生产环境下，MongoDB建议应同时启动三个Config Server，以提高Config Server的整体可用性。当其中的一个或者两个Config Server不可用时，集群的整个元数据将变成只读状态，这时应用依然能够存取业务数据，但是分裂、平衡等影响元数据的操作将不能进行，直到三个Config Server都可用。

### [MongoDB分片介绍（二）—分片键的选择](http://blog.csdn.net/luohongqing/article/details/15335997)

分类： [mongodb](http://blog.csdn.net/luohongqing/article/category/1644627)2013-11-12 17:2622人阅读[评论](http://blog.csdn.net/luohongqing/article/details/15335997#comments)(0)[收藏](javascript:void(0);)[举报](http://blog.csdn.net/luohongqing/article/details/15335997#report)

[mongodb](http://www.csdn.net/tag/mongodb)[Sharding](http://www.csdn.net/tag/Sharding)[Chunk 集群](http://www.csdn.net/tag/Chunk%20%e9%9b%86%e7%be%a4)

分片键决定了MongoDB集群如何对数据进行分布，mongos实例如何将操作请求转发给集群，因此分片键的选择将直接影响MongoDB的数据库性能。

一、分片键的影响

1. Write Scaling（写扩展）

某些集群键可以利用集群提供的写能力，而另外一些集群键则不能。我们以集合的默认对象ID即\_ID字段为例，看分片键对写操作的影响。MongoDB为每个新创建的文档产生一个唯一的对象ID，然而该值的大部分表示的是时间戳，这意味则对象ID的值将以一个规律的、可预测的方式生成。当然这些对象ID的值是具有很高的基（即拥有大量不同的值），但是由于这些值是线性增长的，导致了插入操作将会将数据存储到同一个Chunk中，当然也会在同一个分片中。这样该分片的写能力将决定整个集群的写能力。如果我们很少有插入操作，主要是更新操作的话，那么具有线性增长特性的分片键将不会阻碍数据的写性能。因此选择一个具有大量基的同时能够将写操作分布到整个集群的分片键是一个良好的选择。MongDB 2.4版本提供了基于HASH的分区，能够将写操作随机的分布到整个集群。

2. Query Isolation（查询隔离）

mongos实例为应用程序提供接口与分片集群进行交互，它隔离了复杂的集群数据分区问题。在集群环境下，使用分片键且只有一个分区参与的查询将是最快的查询。那些不包含分片键的查询将查询所有的分片，mongos实例将操作转发给所有分片后将会等待所有分片返回结果后才向应用程序返回，因此这些查询将会是耗时的查询。如果我们查询条件包含分区键或者一部分分区键，mongos实例将会把查询请求转发给一个或者几个分片，这将会提高查询的性能。因此我们需要考虑：

* 应用中那些字段将经常作为查询条件
* 应用中那些查询的性能最重要

二、选择分片键的考虑因素

分片键的选择是否合适，对数据库集群的性能、容量以及功能都有巨大的影响。分片键选择的合理性依赖于数据的模式以及我们查询、写数据的方式。下面几条都是选择分片键的潜在考虑因素。

1. 选择容易分开的字段为分片键

易分散的分片键可以方便MongoDB把数据分布到各个分片，只有有限取值的分片键将导致Chunk难以分裂。比如，性别字段就不是一个良好的候选分片键，它的不同值太少，只有两个。而邮政编码就是一个比较好的候选分片键，在中国它的取值是每个乡镇一个，可以达到上万个；身份证号就是一个更好的候选分片键了，每个人一个，可以达到上亿的不同值。

2. 选择具有随机性字段为分片键

有的字段虽然具有大量的不同取值，但是这些字段在插入时不具有随机性，而具有规律性。这种规律性使得插入时一个分片成为热点，其他分片不能负担写压力，从而产生性能瓶颈。比如上面提到的\_ID字段，虽然它具有很大的基，但是它不具有随机性，每次插入时都会插入到同一个Chunk上；而身份证号码则同时拥有很大的基同时有具有随机性。

3. 选择经常作为查询条件字段为分片键

这类分片键可以使得查询时mongos仅仅将查询发送给特定的mongod实例，不需要等待多个实例返回数据后再进行合并。这一点与上面一条存在冲突，一个具有高度随机性的字段将难以实现查询时仅仅涉及特定的分片，因为其数据一定是打散后分布在多个分片的。

从MongoDB 2.4版本支持Hash分片键看，数据分布的均衡性优先级高于查询的分区过滤。

4.考虑使用组合分片键

有时候集合中的现有字段单独作为分片键不合适，把现有一个字段的计算值或者直接使用组合键作为分片键会产生一个较为理想的结果。

三、基的重要性

我们上面提到，对数据库集群来说，数据分布的均衡性优先级很高，而字段的基直接决定该字段是否合适作为分片键。基是指一个字段不同取值的个数，如性别字段的基为2，在中国省字段的基为31个，而身份证字段的基为13亿个。分片键的基决定了系统将数据分片到各机器的能力。当分片键具有较少的基时，所有所有的键值相同导致MongoDB不能分裂Chunk。迁移这些不可分裂的Chunk将更加耗时，也即使迁移后也难以保证数据在各个分片上的平衡。Chunk数量被基约束住后，我们就不能利用MongoDB集群特性为集合部署更多的机器。

### [MongoDB分片介绍（三）—分布式集群部署及维护](http://blog.csdn.net/luohongqing/article/details/16113735)

我们提到过，MongoDB分片需要三个角色才能完成，一个是Config Server作为元数据存储服务器，一个是Mongos实例作为查询路由转发器，最后是MongoD实例用于存储实际用户数据。

一、启动实例

Config Server其实也是一个MongoDB实例，只不过我们通过启动时增加--configsvr选项将其功能专注于集群元数据的处理，每个Config Server都有一套完整且独立的集群元数据。在实际生产环境下，我们需要部署三个Config Server在不同的机器上，以提高可用性。在测试环境下，我们可以将三个Config Server部署在一台机器上，或者只部署一个Config Server，默认情况下Config Server的端口号为27019

[root@localhost bin]# mkdir -p /mongodb/data/configdb1  
[root@localhost bin]# mkdir -p /mongodb/data/configdb2  
[root@localhost bin]# mkdir -p /mongodb/data/configdb3

我们给每个Config Server单独设置一个配置文件，每个配置文件的configsvr选项都设置为true。由于在同一台机器上运行，我们将各Config Server的端口号依次为27119、27219、27319，数据库目录依次为configdb1、configdb2、configdb3，其中configsvr1如下：

fork = true  
bind\_ip = 127.0.0.1  
port = 27119  
quiet = true  
dbpath = /mongodb/data/configdb1  
logpath = /mongodb/log/mongod\_configdb1.log  
logappend = true  
journal = true  
configsvr=true  
smallfiles = true  
oplogSize = 128

[root@localhost bin]#./mongod --config /mongodb/mongodb.conf1

[root@localhost bin]#./mongod --config /mongodb/mongodb.conf2

[root@localhost bin]#./mongod --config /mongodb/mongodb.conf3

完成对Config Server的启动后，我们就可以启动mongos实例。mongos实例是一个轻量级的实例，不需要存储数据的目录，其默认端口号为27017。在启动mongos实例时，我们需要指定ConfigServer的IP和端口号。为了避免因Config Server IP地址变化导致MongoDB实例和Mongos实例重新启动，因此最好给每个Config Server设置一个DNS逻辑名，Mongos实例启动时直接使用Config Server的DNS逻辑名和端口号。我们给mongos实例设置的配置文件如下：

fork = true  
bind\_ip = 127.0.0.1  
port = 27017  
quiet = true  
configdb= 127.0.0.1:27119,127.0.0.1:27219,127.0.0.1:27319  
logpath = /mongodb/log/mongos.log

下面启动mongos实例和sharding实例，sharding实例启动和普通Mongod实例基本相同，只是多设置一个shardsvr为TRUE即可，带上该参数后mongod实例的默认端口号为27018。我们拟启动两个sharding实例，端口号依次为27118、27218。

[root@localhost bin]# ./mongos --config /mongodb/mongodb.s

[root@localhost bin]# ./mongod --config /mongodb/mongodb.sh1

[root@localhost bin]# ./mongod --config /mongodb/mongodb.sh2

二、初始化sharding

1、连接mongos实例，将两个sharding实例增加到集群中

[root@localhost bin]# ./mongo --port 27017  
MongoDB shell version: 2.5.2  
connecting to: 127.0.0.1:27017/test  
mongos> sh.addShard( "127.0.0.1:27118" )sh.addShard( "127.0.0.1:27118" )  
{ "shardAdded" : "shard0000", "ok" : 1 }  
mongos> sh.addShard( "127.0.0.1:27218" )sh.addShard( "127.0.0.1:27218" )  
{ "shardAdded" : "shard0001", "ok" : 1 }

2、使能test数据库以支持集群

mongos> sh.enableSharding("test")  
{ "ok" : 1 }

3、使能people集合以支持分片

mongos> sh.shardCollection("test.people", {user\_id:1})  
{ "collectionsharded" : "test.people", "ok" : 1 }

三、观察集群状态

我们可以使用sh.status()查看集群的状态。下面的命令结果显示，在集群中有两个分片，两个数据库中只有Test数据库支持分片，其主分片是shard0000。主分片是指在分片开始之前，集合数据所在的位置。

mongos> sh.status()  
--- Sharding Status ---   
sharding version: {  
"\_id" : 1,  
"version" : 3,  
"minCompatibleVersion" : 3,  
"currentVersion" : 4,  
"clusterId" : ObjectId("5284bb99c8b3dd65fd2c3ea6")  
}  
shards:  
{ "\_id" : "shard0000", "host" : "127.0.0.1:27118" }  
{ "\_id" : "shard0001", "host" : "127.0.0.1:27218" }  
databases:  
{ "\_id" : "admin", "partitioned" : false, "primary" : "config" }  
{ "\_id" : "test", "partitioned" : true, "primary" : "shard0000" }  
test.people  
shard key: { "user\_id" : 1 }  
chunks:  
shard0000 1  
{ "user\_id" : { "$minKey" : 1 } } -->> { "user\_id" : { "$maxKey" : 1 } } on : shard0000 Timestamp(1, 0)

mongos>

由于people表中还没有数据，上面集群状态显示people集合只有一个Chunk。下面我们插入1万数据，观察一下Chunk分裂的过程。

use test

people = ["Marc", "Bill", "George", "Eliot", "Matt", "Trey", "Tracy", "Greg", "Steve", "Kristina"]

for(var i=0; i<10000; i++){  
name = people[Math.floor(Math.random()\*people.length)];  
user\_id = i;  
boolean = [true, false][Math.floor(Math.random()\*2)];  
added\_at = new Date();  
number = Math.floor(Math.random()\*10001);  
db.people.save({"name":name, "user\_id":user\_id, "boolean":boolean,"added\_at":added\_at,"number":number });  
}

在数据完成插入以后，我们再执行sh.status()，发现目前已经people集合拥有三个Chunk，shard0000拥有2个chunk，shard0001拥有1一个chunk。shard0000上,user\_id从最小值到0为一个chunk，0到5739为另外一个chunk；shard0001，user\_id从5739到最大值为一个chunk。

..............................

test.people  
shard key: { "user\_id" : 1 }  
chunks:  
shard0000 2  
shard0001 1  
{ "user\_id" : { "$minKey" : 1 } } -->> { "user\_id" : 0 } on : shard0000 Timestamp(2, 1)  
{ "user\_id" : 0 } -->> { "user\_id" : 5739 } on : shard0000 Timestamp(1, 3)  
{ "user\_id" : 5739 } -->> { "user\_id" : { "$maxKey" : 1 } } on : shard0001 Timestamp(2, 0)

我们继续向people集合插入数据，user\_id从1万到100万，合计99万条记录。在插入完成后发现chunk分裂成了8个，shard0000和shard0001上各4个Chunk。

..................................

test.people  
shard key: { "user\_id" : 1 }  
chunks:  
shard0001 4  
shard0000 4  
{ "user\_id" : { "$minKey" : 1 } } -->> { "user\_id" : 0 } on : shard0001 Timestamp(8, 0)  
{ "user\_id" : 0 } -->> { "user\_id" : 5739 } on : shard0000 Timestamp(8, 1)  
{ "user\_id" : 5739 } -->> { "user\_id" : 188952 } on : shard0001 Timestamp(7, 1)  
{ "user\_id" : 188952 } -->> { "user\_id" : 372157 } on : shard0000 Timestamp(4, 2)  
{ "user\_id" : 372157 } -->> { "user\_id" : 555370 } on : shard0001 Timestamp(5, 2)  
{ "user\_id" : 555370 } -->> { "user\_id" : 746356 } on : shard0001 Timestamp(6, 2)  
{ "user\_id" : 746356 } -->> { "user\_id" : 929568 } on : shard0000 Timestamp(7, 2)  
{ "user\_id" : 929568 } -->> { "user\_id" : { "$maxKey" : 1 } } on : shard0000 Timestamp(7, 3)

四、集群维护

1、迁移Config Server

一般集群拥有三个Config Server，当某一台出现故障时，我们可以进行在线迁移。在迁移过程中，如果不需要更改Config Server的主机名称，则步骤如下：

* 使用sh.setBalancerState(false)暂时停止后台的Balance活动，并使用sh.getBalancerState()确认
* 关闭拟迁移的Config Server，此时集群的配置数据变得只读，所有Chunk的分裂和平衡活动都被禁止，但是用户正常的数据插入是被允许的。
* 更改DNS解析条目，将原主机名称指向新的IP地址。
* 将老Config Server的DB目录复制到新Config Server的DB目录下。
* 在新机器上启动Config Server实例，并使用sh.setBalancerState(true)启动Chunk平衡。

如果涉及到的Config Server的主机名称发生变化，则需要一定的停机重启时间，步骤如下：

* 使用sh.setBalancerState(false)暂时停止后台的Balance活动，并使用sh.getBalancerState()确认
* 关闭拟迁移的Config Server，此时集群的配置数据变得只读，所有Chunk的分裂和平衡活动都被禁止，但是用户正常的数据插入是被允许的。
* 将老Config Server的DB目录复制到新Config Server的DB目录下。
* 在新机器上启动Config Server实例。
* 关闭现有的实例，包括Config Server实例，mongos实例和存放实际用户数据的集群实例
* 启动所有集群实例
* 更新mongos实例的configdb配置参数，并重新启动mongos实例
* 使用sh.setBalancerState(true)启动Chunk平衡

从上述过程可以看出，如果需要更改configdb的配置需要对整个集群进行停机，因此我们在规划时应尽可能采用DNS方式解析Config Server以提高集群的可用性。

2. 维护集群实例

需要将增加新集群实例到集群中时，我们需要首先启动拟新增加的集群实例，然后在mongo shell环境下执行sh.addShard()命令即可。

当我们需要从集群中删除一个集群实例时，一般步骤如下：

* 使用sh.setBalancerState(TRUE)启动Chunk平衡
* 找到准备删除的shard名字，例如我们拟删除mongodb0分片
* 执行命令removeShard，此命令将从该分片中移出所有Chunk到其他分片，如使用db.runCommand( { removeShard: "mongodb0" } )删除mongodb0分片。
* 使用removeShard不断观察迁移的进展情况，直到返回"remove shard completed successfully"消息。
* 使用sh.status()命令查看是否有文档是以该分片为主分片的，movePrimary命令迁移主分片，如db.runCommand( { movePrimary: "products", to: "mongodb1" })将products文档的主分片设置为mongodb1.

### MongoDB之Shard初步认识

[MongoDB](http://www.oschina.net/search?scope=blog&q=MongoDB)[Shard](http://www.oschina.net/search?scope=blog&q=Shard)

**准备工作**

1、 什么是shard

副本集实现了网站的安全备份和故障的无缝转移，但是并不能实现数据的大容量存储，MongoDB实现的是分布式部署，把数据保存到其他机器上。实现这一过程的就是分片。

2、 什么时候需要分片

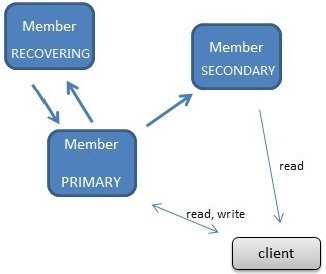
a) 用光了当前机器的磁盘空间

b) 单个的Mongod已经无法提供你要的写入性能了

c) 需要把大部分数据驻留在内存中籍此来提供更好的性能

3、 分片需要的三个部分

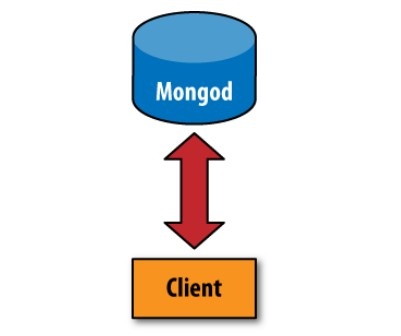
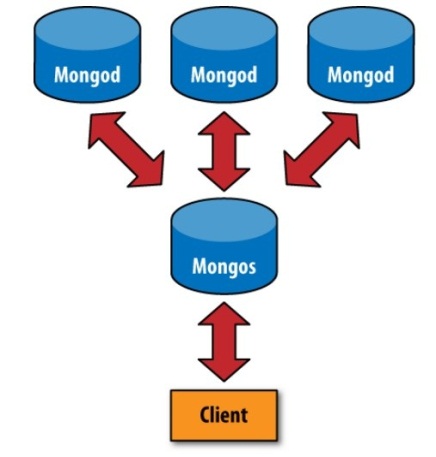
a) shard服务器(Shard Server)。Shard服务器是存储实际数据的分片，每个Shard可以是一个mongod实例，也可以是一组mongod实例构成的Replica Sets。MongoDB官方建议每个Shard为一组Replica Sets（结合了Master/Slave模式与自动数据容错、自动恢复的综合体，结构如下图）。

[](http://static.oschina.net/uploads/space/2012/1120/125252_twDT_264813.jpg)

b) 配置服务器(Config Server)。它存储了所有Shard节点的配置信息，每个chunk的Shard key范围，chunk在各Shard的分布情况以及集群中所有DB和collection的Shard配置信息。

c) 路由进程(Route Process)。一个前端路由，客户端由此接入，首先询问配置服务器需要到那个Shard上查询或保存记录，然后连接相应Shard执行操作，最后将结果返回客户端。（类似于网络中路由器的功能）

4、 未分片和分片Client反问mongod 的区别如下图。

[](http://static.oschina.net/uploads/space/2012/1120/125420_GawI_264813.jpg)[](http://static.oschina.net/uploads/space/2012/1120/125443_GyQX_264813.jpg)

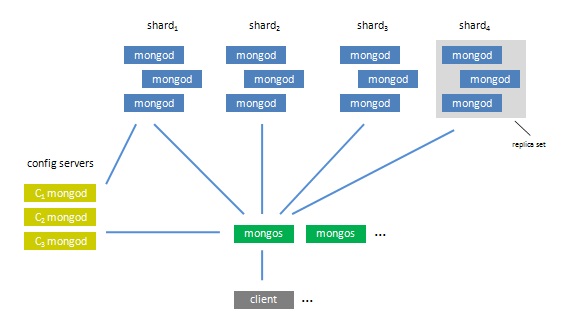
未分片客户端的连接 分片后客户端的连接

5、 Shard Keys

MongoDB主要根据Shard Keys来划分数据，Shard Keys可以由文档的一个或者多个物理键值组成，对于分片Key的选定直接决定了集群中数据分布是否均衡、集群性能是否合理。

配置命令（参考《MongoDB:The Definiive Guide》,以及官网上的《MongoDB Manual》, <http://docs.mongodb.org/manual/MongoDB-Manual.pdf>）

**一． 整体架构**

[](http://static.oschina.net/uploads/space/2012/1120/130258_YuEb_264813.jpg)

说明：

1. 由于config servers非常重要，建议使用3个config servers，但是可以只是用一个config servers。

2. 一个较好的Auto Sharding解决方案是将每个一shard（分片）定为一个Replica Sets（复制集）。一个Replica Sets由若干个mongod instance组成，在这个集合中，所有的instance的数据相同，这使得即使有某一台机子当掉了，其它机子还是可以正常运行，而且这部分的控制是由Mongo自动完成的，因而尽可能地减少了因当机而产生的错误及人工处理的部分。而Sharding是可以将庞大的数据库拆分为几个部分分别发放到每一个shard，一来降低了单一一台服务器的压力，同时通过减少潜在的损失比例来提高效率。也就是说一个基于Replica Sets的Auto Sharding结构，可以把一个完整而且庞大的数据库根据个人定制，拆分到若干个服务器集合，每个服务器集合中的服务器群又相互保持数据同步，所以除非一个服务器集合中的所有服务器都当掉了，否则某台或几台的当机对数据库的影响是微忽其微的。

3. 一个完整的Auto Sharding功能的实现需要用到mongod和mongos，其中mongos作为真正的应用接口，数据的输入输出都应经过它。然后还需要一个或多个config server（建议是三个），它是mongod，但它不会用来存储应用程序的数据库，通俗的来说而是存放了这整个结构的配置属性，mongos会从config server中读取配置来进行工作。最后是真正会存储数据的mongod们，它们按组分为若干个Replica Sets，用来存放mongos拆分下来的各个sharding。

4. Config服务器存储着集群的metadata信息，包括每个服务器，每个shard的基本信息和chunk（后面介绍）信息Config服务器主要存储的是chunk信息。每一个config服务器都复制了完整的chunk信息。

5. Set up a Sharded Cluster（一开始就允许shard）的步骤。详细命令介绍参考  
（<http://docs.mongodb.org/manual/administration/sharding/#list-databases-with-sharding-enabled>）

a) Create data directories for each of the three config server instances.

b) Start the three config server instances.

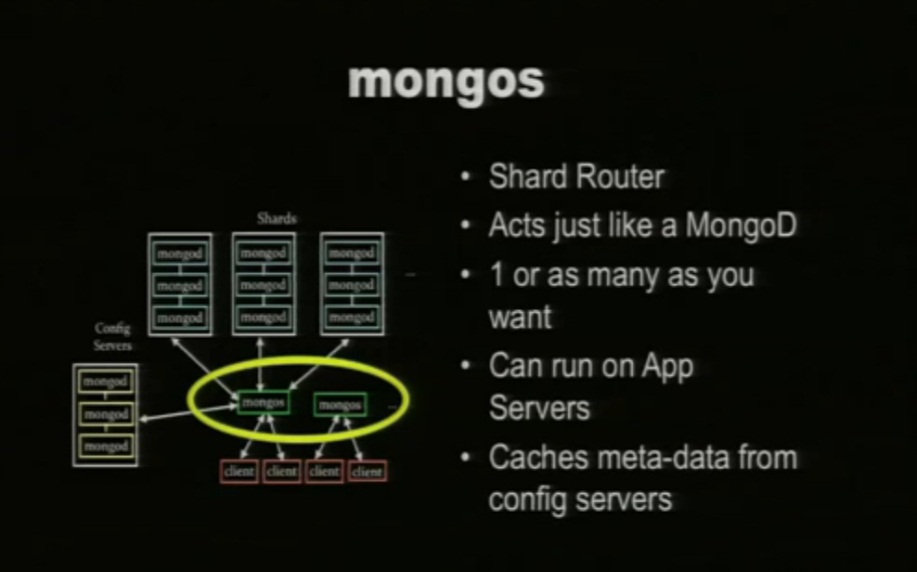
c) Start a mongos instance.

d) Connect to one of the mongos instances.

e) Add shards to the cluster.

f) Enable sharding for each database you want to shard.**OR** Enable sharding on a per-collection basis.

**二． Mongos**

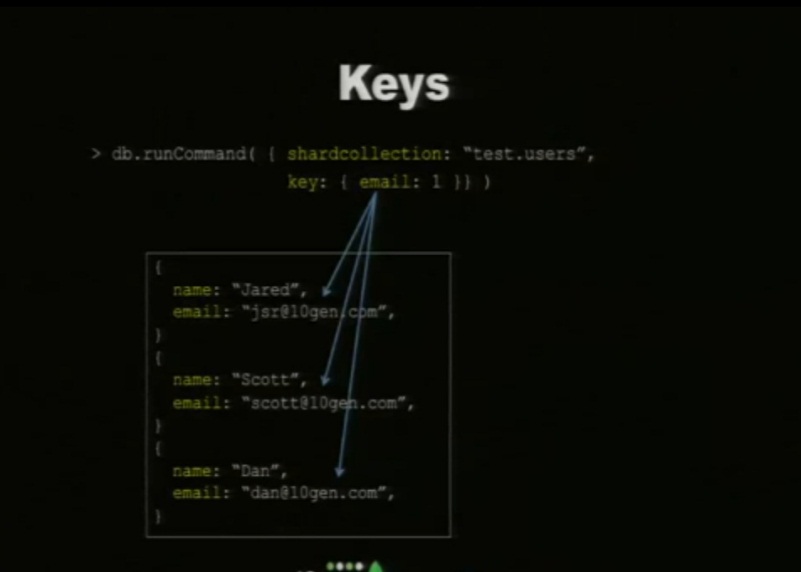
[](http://static.oschina.net/uploads/space/2012/1120/130333_uUZw_264813.jpg)

说明：

1. mongos作为真正的应用接口，数据的输入输出都应经过它。

2. 另外，mongos还有平衡shard的功能（后面介绍）；

**三． shardKey**

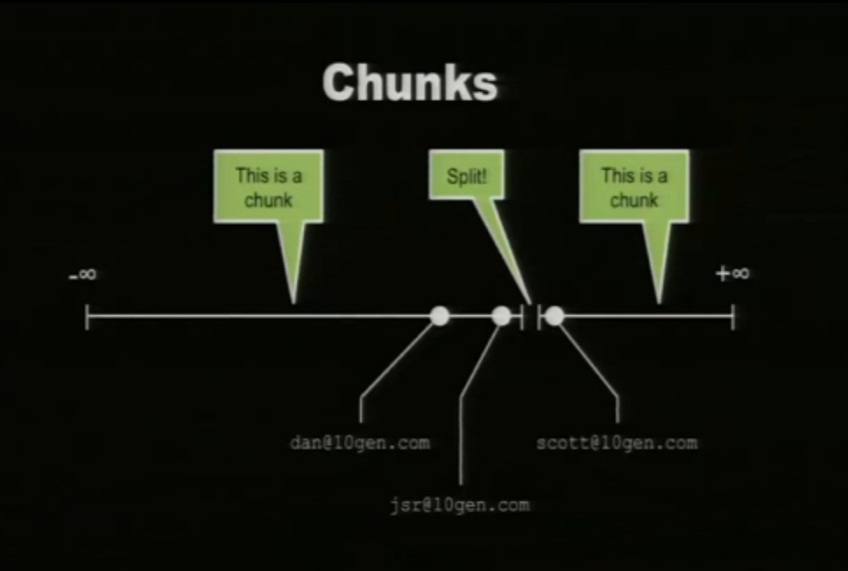
[](http://static.oschina.net/uploads/space/2012/1120/130422_0cKc_264813.jpg)

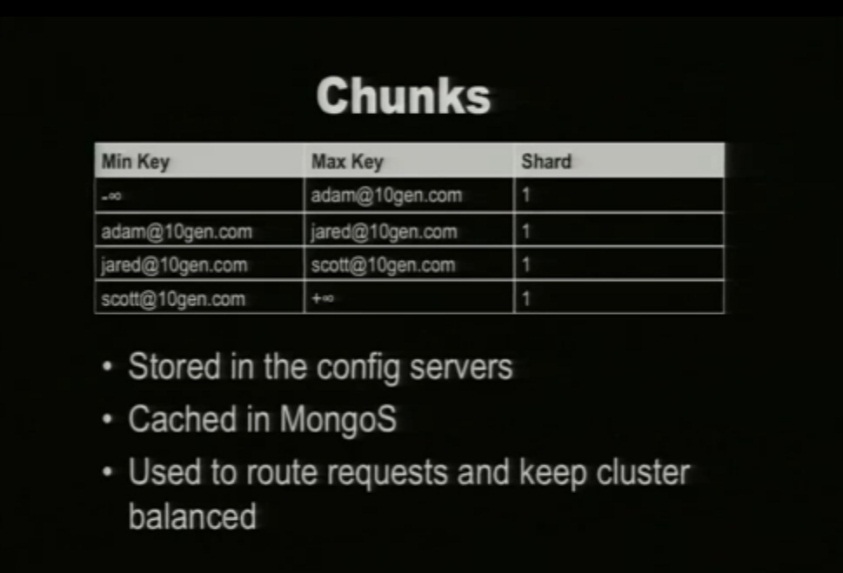
说明：

1. 命令：{ shardCollection: "<db>.<collection>", key: { "<shardkey>": 1 } }

2. MongoDB主要根据Shard Keys来划分数据，Shard Keys可以由文档的一个或者多个物理键值组成，对于分片Key的选定直接决定了集群中数据分布是否均衡、集群性能是否合理。

**四． Chunks**

[](http://static.oschina.net/uploads/space/2012/1120/130502_ZzLC_264813.jpg)

[](http://static.oschina.net/uploads/space/2012/1120/130527_J5ip_264813.jpg)

说明：

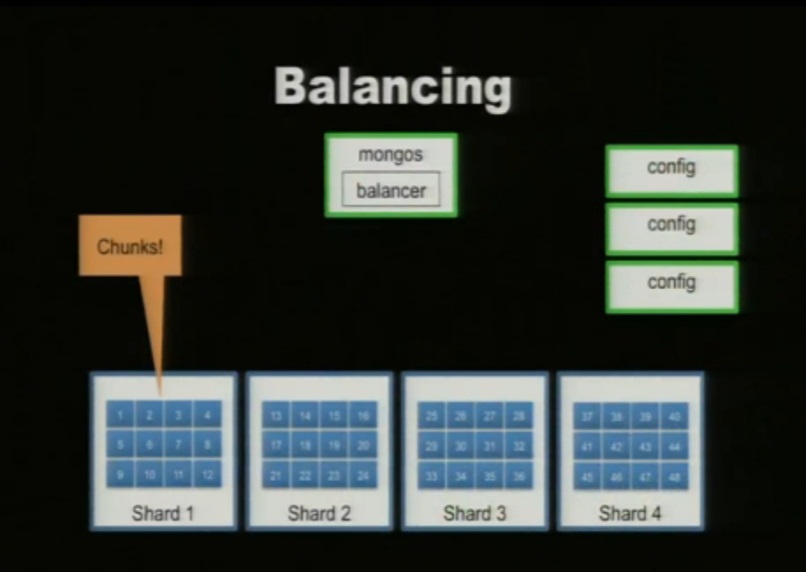
1. Chunk是一个来自特殊集合中的一个数据范围，（collection，minKey，maxKey）描叙一个chunk，它介于minKey和maxKey范围之间。例如chunks 的maxsize（默认值是64M）大小是100M，如果一个文件达到或超过这个范围时，会被切分到2个新的chunks中。当一个shard的数据过量时，chunks将会被迁移到其他的shards上。同样，chunks也可以迁移到其他的shards上。

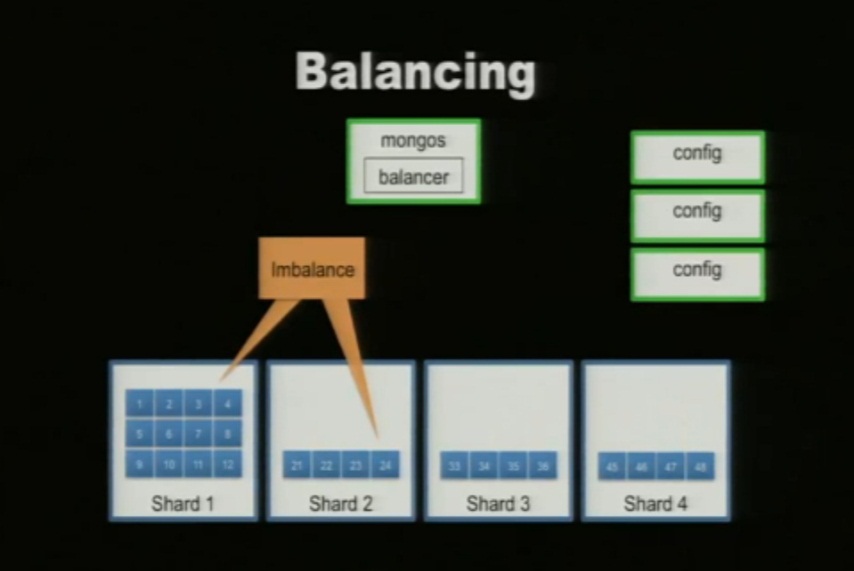
2. Chunk包括了Shard Key取值在minKey和maxKey之间一组文档集合，但Chunk并不存放实际数据。整个MongoDB的chunk元信息存放在config数据库的chunks Colecttion中。

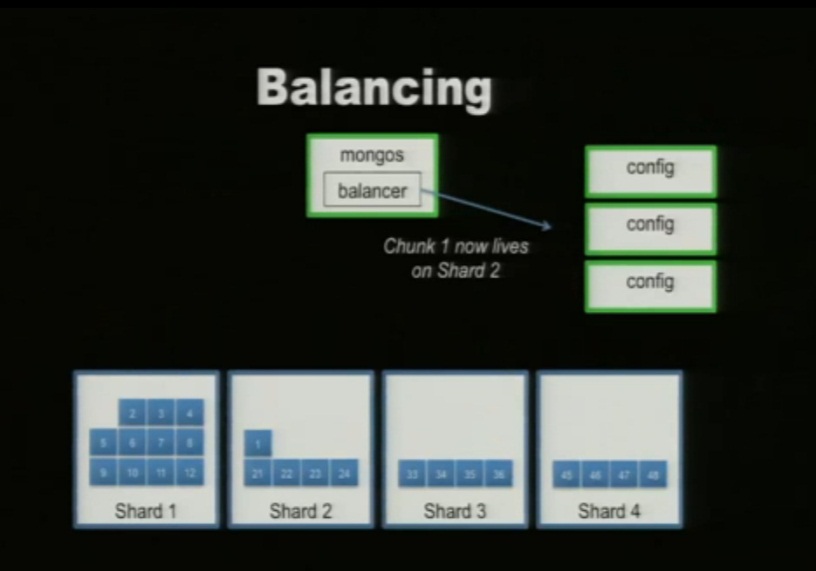
3. 当Chunk的大小达到maxsizeMongoDB会根据minKey和maxKey的一个中间值将一个Chunk分裂（Split）为两个Chunk，分裂完毕后，MongoDB会根据各个Shard的负载情况，决定是否将新Chunk移动到其他Shard。

4. 常用命令参考  
（<http://docs.mongodb.org/manual/administration/sharding/#list-databases-with-sharding-enabled>）

**五． Balance Shard**

**[](http://static.oschina.net/uploads/space/2012/1120/130549_UTVr_264813.jpg)**

**[](http://static.oschina.net/uploads/space/2012/1120/130617_ocvK_264813.jpg)**

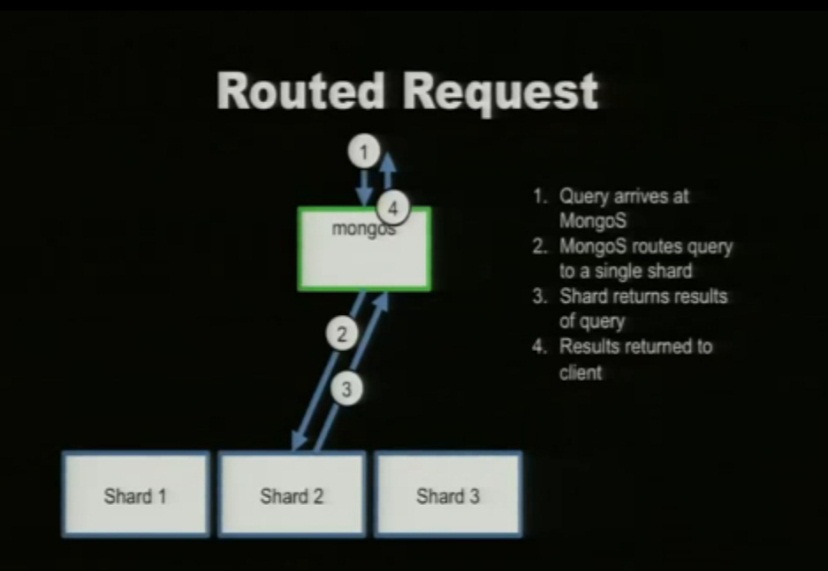
**[](http://static.oschina.net/uploads/space/2012/1120/130639_3BnP_264813.jpg)**

说明：

1. shard don’t a range of chunks but a block of chunks which have a range of data by shardKey.

2. > db.runCommand( { addshard : "10.0.4.85:27020", allowLocal : 1, maxSize:2 , minKey:1, maxKey:10} ) (网上有人测试出shard1中数据超过10M也没有分片，暂时感觉还有些问题，找了很多资料也没有找到)

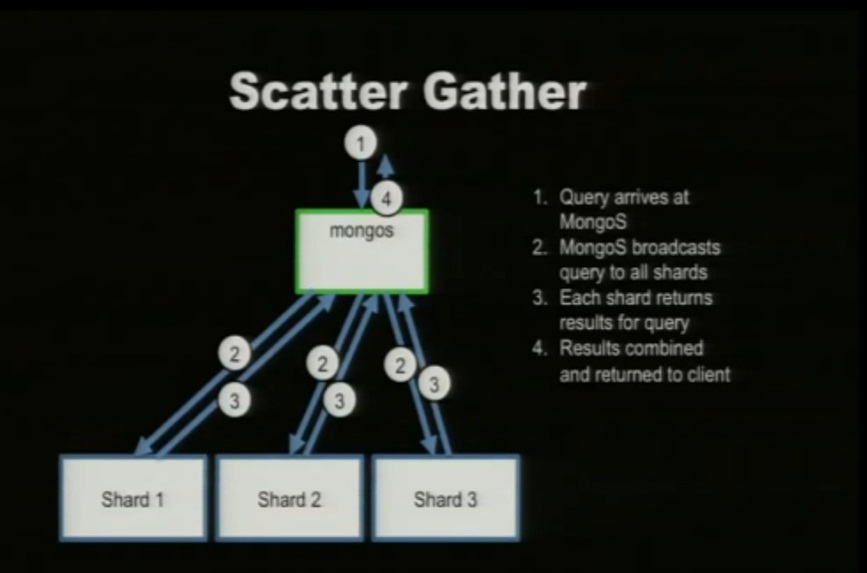
**六． Routed Request**

**[](http://static.oschina.net/uploads/space/2012/1120/130720_oS4A_264813.jpg)**

说明：

1. 命令到达mongos，mongos判断出了query指向某个shard，将命令直接指向该shard并得到results并返回给用户。

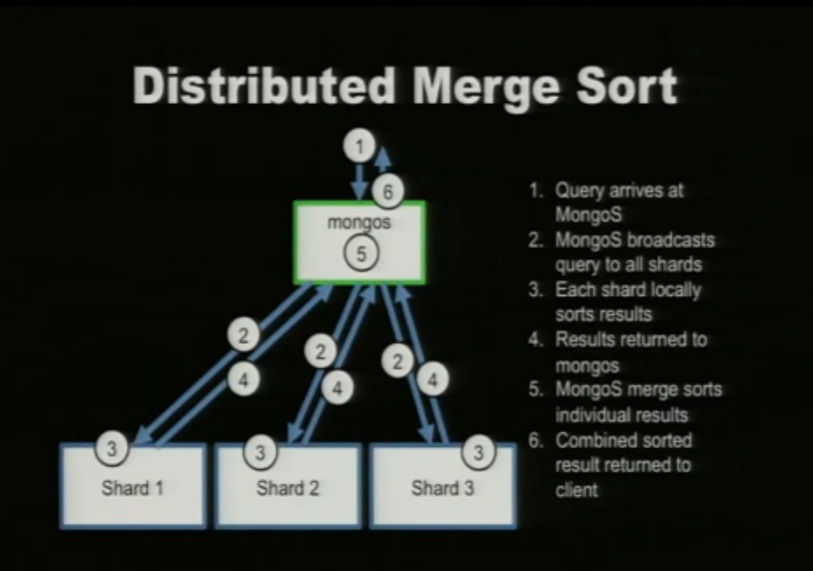
**七． Scatter Gather**

**[](http://static.oschina.net/uploads/space/2012/1120/130828_uTvT_264813.jpg)**

说明：

1. 命令到达mongos，比如说查询所有住在CA的说有用户，mongos判断出需要查询所有的shards，便将这个命令分散传达个每个shard，每个shard都得到相关的结果（并按照默认的shardKey进行排序）返回给mongos，mongos将所有shard返回的结果进行merge（也按照shardKey进行排序）返回给用户。

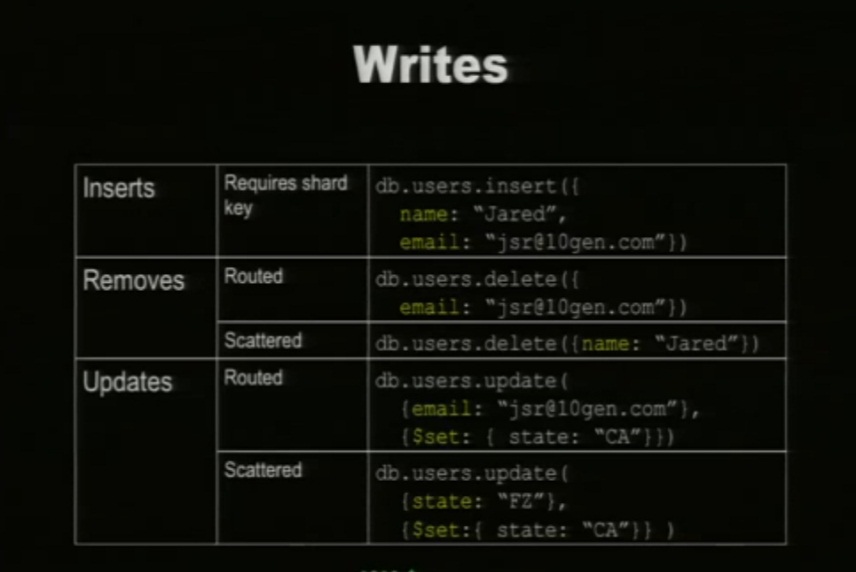
**八． Distributed Merge Sort**

**[](http://static.oschina.net/uploads/space/2012/1120/130848_m7c3_264813.jpg)**

说明：

1. 命令到达mongos，比如说查询所有住在CA的说有用户并按照用户名排序，mongos判断出需要查询所有的shards，便将这个命令分散传达个每个shard，每个shard都得到相关的结果并按照用户名排序返回给mongos，mongos将所有shard返回的结果进行merge（按照用户名排序）返回给用户。

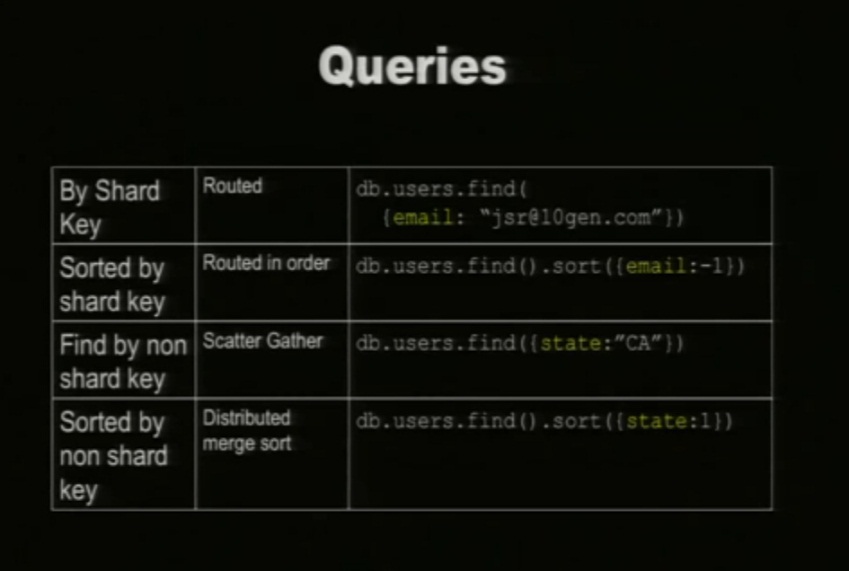
**九． 操作命令**

**[](http://static.oschina.net/uploads/space/2012/1120/130916_0Fmm_264813.jpg)**

说明：

1. 当插入一个document时不含有shardKey的话会出现error。(Insert into it the documents which don’t have shardKey, it doesn‘t work).

**十． 查询命令**

**[](http://static.oschina.net/uploads/space/2012/1120/130937_vg8w_264813.jpg)**

**以上的图片均来源于互联网和视频（http://www.mongodb.org/display/DOCS/Sharding），本人刚刚开始了解MangoDB，有错误的地方希望大家多多指正！**

## 第8章 MongoDB管理

本章主要介绍MongoDB的一些管理操作，主要包括以下操作：

* 启动和停止MongoDB服务。
* 设置MongoDB的访问。
* 命令行操作。
* 对MongoDB的一些进程的控制。
* MongoDB如何进行监控。
* 对MongoDB数据的导入、导出、修复、备份。

### 8.1 启动和停止MongoDB

MongoDB环境搭建完成（MongoDB安装、配置完）之后，必须先启动数据库服务，然后才能使用它。

下面介绍三种启动的实例。

#### 8.1.1 使用命令行启动

MongoDB在启动时没有设置存储数据的目录，默认会将/data/db（windows系统默认的存储数据的目录为c:/data/db）目录作为存储数据的目录。启动时没有设置端口号时，**默认端口号为：27017，默认的Http端口为：28017.**如果想自定义存储数据的目录，可以在启动数据库服务时指它—dbpath参数（--dbpath=/data/newdb）。

使用命令行启动，首先需要创建存储数据的目录/data/db，代码如下：

[root@mongodb1 /]$ mkdir –p /data/db

[root@mongodb1 /]$

存储数据的目录创建好了之后，就可以启动服务了，代码如下：

[root@mongodb1 /home/mo ngo/mongodb-2.0.2/bin]$ ./mongd

./mongod –help for help and startup options

当输入上述命令行，出现27017端口和28017端口时，说明服务已经启动成功。下面就可以直接在shell中连接数据库，代码 如下：

[root@mongodb1 /home/mo ngo/mongodb-2.0.2/bin]$ ./mongd

MongoDB shell version:2.0.2

Connecting to :test

>

以上代码使用**./mongod命令**连接数据库，结果显示连接数据库成功。

直接执行./mongod命令，默认连接到端口27017，存储数据目录为data/db的数据库，默认的数据库为：test数据库。如果需要连接到其他数据库，可以在连接数据库时指定所需要连接的数据库名称。代码如下：

[root@mongodb1 /home/mo ngo/mongodb-2.0.2/bin]$ ./mongd 192.168.1.50:27017/admim

MongoDB shell version:2.0.2

Connecting to :192.168.1.50:27017/admin

>

代码说明：

* 192.168.1.50：连接数据库所做的IP地址。
* 27017：数据库服务端口。
* Admin：数据库名称（可以为任何数据库名称。存在的可以直接连接，不存在的则创建此数据并连接）以上代码使用./mongod命令连接数据库，并且制定所需要连接到的数据库名称。上述例子连接到了admin数据库中。

默认的Http端口为28017，即可以通过这个端口在浏览器中访问数据库，访问地址如下：

<http://192.168.1.50:28017/>

#### 8.1.2 配置文件

如果想要系统运行地非常稳定，那么在数据库实例启动时需要加很多参数。这样就会在数据库实例启动时在mongod命令后面加很多参数，非常不好管理和维护。而且每次重启计算机之后都要执行一次这个命令，非常麻烦。那么该怎么办？MongoDB和MySQL一样，同样也支持读取配置文件的方式启动数据库实例。下面介绍如何使用读取配置文件的方式启动数据库实例。

首先需要创建配置文件，并编辑配置文件，将启动MongoDB数据库实例所需要的参数都配置到这个配置文件中，具体代码如下：

[root@mongodb1 /home/mo ngo/mongodb-2.0.2/bin]$ vi /ect/mongodb.config

[root@mongodb1 /home/mo ngo/mongodb-2.0.2/bin]$ cat /ect/mongodb.config

dbpath=/data/db/

[root@mongodb1 /home/mo ngo/mongodb-2.0.2/bin]$

以上代码创建并编辑了mongodb.config配置文件，将dbpath=/data/db/参数配置到这个配置文件中。

下面通过配置文件的方式启动启动数据库实例，启动时只需要加上**—f参数**，并且指向配置文件即可。代码如下：

[root@mongodb1 /home/mo ngo/mongodb-2.0.2/bin]$ ./mongod -f /ect/mongodb.config

#### 8.1.3 Daemon方式启动

上面的两种启动方式都是在前台启动MongoDB数据库服务，当计算机重新启动或者不小心关掉了启动MongoDB数据库服务的session窗口时，MongoDB的服务也会随即停止。这样非常不安全，所以MongoDB提供了一种从后台启动的方式（Daemon方式）启动数据库服务。这种启动方式只需要加上一个**—fork参数**即可。如果使用了—fork参数就必须指定存储日志的文件（即启动—logpath参数）这是使用Daemon方式启动MongoDB数据库服务强制要求的。

具体操作代码如下：

[root@mongodb1 /home/mo ngo/mongodb-2.0.2/bin]$ mond –-dbpath=/data/db –-

logpath=/data/log/mongodb50.log –-fork

[root@mongodb1 /home/mo ngo/mongodb-2.0.2/bin]$ forked process:16527

All output gong to:/data/log/mongodb50.log

[root@mongodb1 /home/mo ngo/mongodb-2.0.2/bin]$

以上代码就是通过Daemon方式启动MongoDB数据库服务。

#### 8.1.4 mongod参数说明

使用mongod启动MongoDB数据库服务时可以设置非常多的参数，在不指定参数的情况下，都是默认的值。默认情况下，最重要的参数就是存储数据的目录，并且需要确保默认的data/db/存储数据的目录存在而且有访问的权限，否则启动服务后又会自动关闭。所以在默认的情况下，只要确认dbpath参数就可以启动MongoDB数据库服务。

mongod 的主要参数及含义如下：

* dbpath：数据文件存放路径，每个数据库都会在其中创建一个子目录，用于防止同一个实例多次运行的mongod.lock也保存在此目录中。
* logpath：日志文件，设置了此参数，所以的日志都会打印到此参数指定的文件中。
* logappend：日志采用追加模式（默认是覆写模式）。
* bin\_ip：对外服务的绑定IP，一般设置为空，及绑定在本机所有可用的IP上，如有需要可以单独指定。
* port：对外服务的端口。Web管理端口在这个port的基础上+1000。
* fork：以后台Daemon形式运行服务。
* journal：开启日志功能，通过保存操作日志来降低单击故障的恢复时间，在1.8版本后正式加入，取代1.75版本中的dur参数。
* syncdelay：系统同步刷新新磁盘的时间，单位为秒，默认为60秒。
* directoryperdb：每个db存放在单独的目录中，建议设置该参数。与MySQL的独立表空间类似。
* maxConns：最大的连接数。
* repairpath：执行repair时的临时目录。如果没有开启journal，出现异常死机后重启，必须执行repair操作。
* 其他的参数可以参考一下说明：

**基本配置**  
--------------------------------------------------------------------------------

|  |  |
| --- | --- |
| --quiet | # 安静输出 |
| --port arg | # 指定服务端口号，默认端口27017 |
| --bind\_ip arg | # 绑定服务IP，若绑定127.0.0.1，则只能本机访问，不指定默认本地所有IP |
| --logpath arg | # 指定MongoDB日志文件，注意是指定文件不是目录 |
| --logappend | # 使用追加的方式写日志 |
| --pidfilepath arg | # PID File 的完整路径，如果没有设置，则没有PID文件 |
| --keyFile arg | # 集群的私钥的完整路径，只对于Replica Set 架构有效 |
| --unixSocketPrefix arg | # UNIX域套接字替代目录,(默认为 /tmp) |
| --fork | # 以守护进程的方式运行MongoDB，创建服务器进程 |
| --auth | # 启用验证 |
| --cpu | # 定期显示CPU的CPU利用率和iowait |
| --dbpath arg | # 指定数据库路径 |
| --diaglog arg | # diaglog选项 0=off 1=W 2=R 3=both 7=W+some reads |
| --directoryperdb | # 设置每个数据库将被保存在一个单独的目录 |
| --journal | # 启用日志选项，MongoDB的数据操作将会写入到journal文件夹的文件里 |
| --journalOptions arg | # 启用日志诊断选项 |
| --ipv6 | # 启用IPv6选项 |
| --jsonp | # 允许JSONP形式通过HTTP访问（有安全影响） |
| --maxConns arg | # 最大同时连接数 默认2000 |
| --noauth | # 不启用验证 |
| --nohttpinterface | # 关闭http接口，默认关闭27018端口访问 |
| --noprealloc | # 禁用数据文件预分配(往往影响性能) |
| --noscripting | # 禁用脚本引擎 |
| --notablescan | # 不允许表扫描 |
| --nounixsocket | # 禁用Unix套接字监听 |
| --nssize arg (=16) | # 设置信数据库.ns文件大小(MB) |
| --objcheck | # 在收到客户数据,检查的有效性， |
| --profile arg | # 档案参数 0=off 1=slow, 2=all |
| --quota | # 限制每个数据库的文件数，设置默认为8 |
| --quotaFiles arg | # number of files allower per db, requires --quota |
| --rest | # 开启简单的rest API |
| --repair | # 修复所有数据库run repair on all dbs |
| --repairpath arg | # 修复库生成的文件的目录,默认为目录名称dbpath |
| --slowms arg (=100) | # value of slow for profile and console log |
| --smallfiles | # 使用较小的默认文件 |
| --syncdelay arg (=60) | # 数据写入磁盘的时间秒数(0=never,不推荐) |
| --sysinfo | # 打印一些诊断系统信息 |
| --upgrade | # 如果需要升级数据库 |

\* Replicaton 参数  
  
--------------------------------------------------------------------------------

|  |  |
| --- | --- |
| --fastsync | # 从一个dbpath里启用从库复制服务，该dbpath的数据库是主库的快照，可用于快速启用同步 |
| --autoresync | # 如果从库与主库同步数据差得多，自动重新同步， |
| --oplogSize arg | # 设置oplog的大小(MB) |

\* 主/从参数  
--------------------------------------------------------------------------------

|  |  |
| --- | --- |
| --master | # 主库模式 |
| --slave | # 从库模式 |
| --source arg | # 从库 端口号 |
| --only arg | # 指定单一的数据库复制 |
| --slavedelay arg | # 设置从库同步主库的延迟时间 |

\* Replica set(副本集)选项：  
--------------------------------------------------------------------------------

|  |  |
| --- | --- |
| --replSet arg | # 设置副本集名称 |

\* Sharding(分片)选项  
--------------------------------------------------------------------------------

|  |  |
| --- | --- |
| --configsvr | # 声明这是一个集群的config服务,默认端口27019，默认目录/data/configdb |
| --shardsvr | # 声明这是一个集群的分片,默认端口27018 |
| --noMoveParanoia | # 关闭偏执为moveChunk数据保存 |

# 上述参数都可以写入 mongod.conf 配置文档里，例如：  
dbpath = /data/mongodb  
logpath = /data/mongodb/mongodb.log  
logappend = true  
port = 27017  
fork = true  
auth = true  
e.g：./mongod -shardsvr -replSet shard1 -port 16161 -dbpath /data/mongodb/data/shard1a -oplogSize 100 -logpath /data/mongodb/logs/shard1a.log -logappend -fork -rest

更多

执行mongod –help可以获取大多数参数的解释。一些隐含的参数只能通过代码来获得。

在命令行操作，当参数非常多的时候，可以将参数配置到一个配置文件中，然后通过—config参数来应用此配置文件。这样就可以比较好的管理这些参数了，具体的代码如下：

[root@mongodb1 /home/mo ngo/mongodb-2.0.2/bin]$ .mongod --config /ect/mongodb.config

#### 8.1.5 停止数据库

MongoDB提供了多种停止数据库服务的方式。如Control-C、发送shutdownServer()指令及发送UNIX系统中断信号等。下面来介绍几种命令的用法。

1.Control-C

如果数据库服务处于启动状态下，可以直接通过Control-C方式去停止MongoDB实例。

具体代码如下：

$ mongo

>Control-C

2.shutdownServer()指令

如果是处于连接状态，可以切换到admin数据，然后可以直接通过在admin库中发送

db.shutdownServer()指令来停止MongoDB数据库实例。具体代码如下：

[root@mongodb1 /home/mo ngo/mongodb-2.0.2/bin]$ ./mongd

MongoDB shell version:2.0.2

Connecting to :test

>user admin

Switched to db admin

> db.shutdownServer()

3.UNIX系统指令

使用UNIX系统指令停止数据库实例，必须先找到相应的实例进程，可以通过发送Kill -2 PID或者Kill -15 PID来停止进程。具体代码如下：

$ ps aux|grep mongod   
$ kill -2 pid

以上代码首先通过ps aux|grep mongod命令查询到mongod的进程，然后发

送kill -2 pid 指令来停止mongod的进程。

**注意**：不要使用Kill -9 pid来杀死MongoDB进程，这样可能会导致MongoDB数据被破坏。

### 8.2 访问控制

官方手册中启动 MongoDB  服务时没有任何参数，一旦客户端连接后可以对数据库任意操作，而且可以远程访问数据库，所以推荐开发阶段可以不设置任何参数，但对于生产环境还

是要仔细考虑一下安全方面的因素，而提高 MongoDB  数据库安全有几个方面：

 绑定IP内网地址访问MongoDB服务

 设置监听端口

 使用用户名和口令登录

#### 8.2.1 绑定IP 内网地址访问MongoDB服务

MongoDB 可以限制只允许某一特定 IP 来访问，只要在启动时加一个参数 bind\_ip 即可，如下:

服务端限制只有192.168.1.103这个IP可以访问MongoDB 服务

[root@localhost bin]# ./mongod --bind\_ip 192.168.1.103

#### 8.2.2 设置监听端口

官方默认的监听端口是27017，为了安全起见，一般都会修改这个监听端口，避免恶意的连

接尝试，具体如下：

将服务端监听端口修改为28018

[root@localhost bin]# ./mongod --bind\_ip 192.168.1.103 --port 28018

#### 8.2.3 使用用户名和口令登录

MongoDB默认的启动是不验证用户名和密码的,启动MongoDB后,可以直接用MongoDB连接上来,对所有的库具有root权限。所以启动的时候指定参数,可以阻止客户端的访问和连接。

先启用系统的登录验证模块,  只需在启动时指定 auth  参数即可，如

[root@localhost bin]# ./mongod --auth

##### 1. 建立系统root帐号

在admin库中新添一个用户root:

[root@localhost bin]# ./mongo

MongoDB shell version: 1.8.1

connecting to: test

> db.addUser("root","111")

{

        "user" : "root",

        "readOnly" : false,

        "pwd" : "e54950178e2fa777b1d174e9b106b6ab"

}

> db.auth("root","111")

本地客户端连接，指定用户，结果如下:

 [root@localhost bin]# ./mongo -u root -p

MongoDB shell version: 1.8.1

Enter password:

connecting to: test

> show collections;

system.indexes

system.users

##### 2. 建立指定权限用户

MongoDB 也支持为某个特定的数据库来设置用户，如我们为 test 库设一个只读的用户user\_reader:

[root@localhost bin]# ./mongo -u root -p

MongoDB shell version: 1.8.1

Enter password:

connecting to: test

> show collections;

system.indexes

system.users

> use test

switched to db test

> db.addUser("user\_reader", "user\_pwd", true)

{

        "user" : "user\_reader",

        "readOnly" : true,

        "pwd" : "0809760bb61ee027199e513c5ecdedc6"

}

### 8.3 命令行操作

MongoDB不仅可以交互，还可以执行指定的javascript文件，执行指定命令的片段。有了这个特性，就可以通过MongoDB与Linux Shell完美结合。完成大部分日常管理和维护工作。

#### 8.3.1 通过eval参数执行指定语句

例如，需要查询test库的t1表中的记录数有多少，常用方法如下:

|  |
| --- |
| [root@localhost bin]# ./mongo test  MongoDB shell version: 1.8.1  connecting to: test  > db.t1.count()  7  > |

通过命令行eval参数直接执行语句:

|  |
| --- |
| [root@localhost bin]# ./mongo test --eval "printjson(db.t1.count())"  MongoDB shell version: 1.8.1  connecting to: test  7 |

#### 8.3.2执行指定文件中的内容

如果涉及到很多的操作后，才能得到结果，那么用eval的方式来做的话是不可能完成的，那么更灵活的执行指定文件的方式就派上用场了。例如我们仍然要查看test库t1表中的记录数：

t1\_count.js就是我们要执行的文件，里面的内容如下

|  |
| --- |
| [root@localhost bin]# cat t1\_count.js  var totalcount = db.t1.count();  printjson('Total count of t1 is :  ' + totalcount);  printjson('-----------------------'); |

下面我们将执行这个文件

|  |
| --- |
| [root@localhost bin]# ./mongo t1\_count.js  MongoDB shell version: 1.8.1  connecting to: test  "Total count of t1 is :  7"  "-----------------------"  [root@localhost bin]# |

大家可以看到最终得到t1表的记录数 7，那么一些不必要的说明性文字我们要是不希望出现该怎么办呢?

|  |
| --- |
| [root@localhost bin]# ./mongo --quiet  t1\_count.js  "Total count of t1 is :  7"  "-----------------------"  [root@localhost bin]# |

通过指定quiet参数，即可以将一些登录信息屏蔽掉，这样可以让结果更清晰。

### 8.4 进程控制

我们经常需要解决一些数据库的性能问题，可以通过查看有哪些进程，看有没有异常的进程，如果有异常进程，就把他停止。进程的控制MongoDB又是怎么处理的呢？下面介绍一下MongoDB中是如何对进程进行控制的。

#### 8.4.1 查看活动进程

查看活动进程，便于了解系统正在做什么，以便做下一步判断

|  |
| --- |
| > db.currentOp();  > // 等同于: db.$cmd.sys.inprog.findOne()  { inprog: [ { "opid" : 18 , "op" : "query" , "ns" : "mydb.votes" ,                "query" : "{ score : 1.0 }" , "inLock" : 1 }            ]  } |

**字段说明:**

l  Opid: 操作进程号

l  Op: 操作类型(查询，更新等)

l  Ns: 命名空间, 指操作的是哪个对象

l  Query: 如果操作类型是查询的话，这里将显示具体的查询内容

l  lockType: 锁的类型，指明是读锁还是写锁

#### 8.4.2 结束进程

如果某个异常是由于某个进程产生的，那么一般DBA都会毫不留情的杀掉这个罪魁祸首的进程，下面将是这操作

|  |
| --- |
| > db.killOp(1234/\*opid\*/)  > // 等同于: db.$cmd.sys.killop.findOne({op:1234}) |

|  |
| --- |
| 注意:  不要kill内部发起的操作，比如说replica set发起的sync操作等 |

### 8.5 监控

MongoDB管理员很重要的工作就是监控系统的状态和性能。下面介绍几种监控方式。

#### 8.5.1 使用管理接口

默认情况下,启动mongod会启动基本的http服务器,该服务的默认端口是28017.可以在浏览器中输入

localhost:28017.有些链接需要在mongod启动时,用--rest选项开启rest支持 才能进去.当开启rest支持后,可以

在mongod启动时使用--nohttpinterface来关闭管理接口.

#### 8.5.2 serverStatus

要获取运行中的MongoDB服务器统计信息,最基本的工具是serverStatus命令

db.runCommand({"serverStatus":1})

serverStatus返回的键解释:

"globalLock"的值表示全局写入锁占用了服务器多少时间(单位微秒)

"mem"包含服务器内存映射了多少数据,服务器进程的虚拟内存和常驻内存的占用情况(单位MB)

"indexCounters"表示B树在磁盘检索("misses")和内存检索("hits")的次数.如果这个比值开始上升,就要考虑加内存了.

"backgroundFlushing"表示后台做了多少次fsync以及用了多少时间

"opcounters"文档包含了每种主要操作的次数

"asserts"统计了断言的次数

#### 8.5.3 mongostat

serverStatus虽然强大,但对服务器的监控来说不怎么容易.MongoDB提供了mongostat

mongostat输出一些serverStatus提供的重要信息,它会每秒输出新的一行,比之前看到的静态数据实时性要好.

它输出多个列,分别是 inserts/s commands/s vsize 和 %locked,与serverStatus的数据相对应.

#### 8.5.4 第三方插件

还可以使用第三方插件进行数据库的监控，如Nagios、GangLia、Cacti.可以登录到<http://dochub.mongodb.org/core/monitoring>查看与监控工具相关的文档。

### 8.6 数据导出 mongoexport

作为DBA，经常会碰到导入导出数据的需求，下面就介绍实用工具mongoexport和mongoimport的使用方法，望你会有所收获。

假设库里有一张user表，里面有2条记录，我们要将它导出

|  |
| --- |
| > use my\_mongodb  switched to db my\_mongodb  > db.user.find();  { "\_id" : ObjectId("4f81a4a1779282ca68fd8a5a"), "uid" : 2, "username" : "Jerry", "age" : 100 }  { "\_id" : ObjectId("4f844d1847d25a9ce5f120c4"), "uid" : 1, "username" : "Tom", "age" : 25 }  > |

#### 8.6.1常用导出方法

|  |
| --- |
| [root@localhost bin]# ./mongoexport -d my\_mongodb -c user -o user.dat  connected to: 127.0.0.1  exported 2 records  [root@localhost bin]# cat user.dat  { "\_id" : { "$oid" : "4f81a4a1779282ca68fd8a5a" }, "uid" : 2, "username" : "Jerry", "age" : 100 }  { "\_id" : { "$oid" : "4f844d1847d25a9ce5f120c4" }, "uid" : 1, "username" : "Tom", "age" : 25 }  [root@localhost bin]# |

**参数说明**:

l -d 指明使用的库, 本例中为” my\_mongodb”

l -c 指明要导出的表, 本例中为”user”

l -o指明要导出的文件名, 本例中为”user.dat”

从上面可以看到导出的方式使用的是JSON的样式

#### 8.6.2导出CSV格式的文件

|  |
| --- |
| [root@localhost bin]# ./mongoexport -d my\_mongodb -c user--csv -f uid,username,age -o user\_csv.dat  connected to: 127.0.0.1  exported 2 records  [root@localhost bin]# cat user\_csv.dat  uid,username,age  2,"Jerry",100  1,"Tom",25  [root@localhost bin]# |

**参数说明:**

l -csv 指要要导出为csv格式

l -f 指明需要导出哪些例

更详细的用法可以 mongoexport –help来查看

### 8.7 数据导入mongoimport

在上例中我们讨论的是导出工具的使用，那么本节将讨论如何向表中导入数据

#### 8.7.1 导入JSON数据

我们先将表user删除掉，以便演示效果

|  |
| --- |
| > db.user.drop();  true  > show collections;  system.indexes  > |

然后导入数据

|  |
| --- |
| [root@localhost bin]# ./mongoimport -d my\_mongodb -c user user.dat  connected to: 127.0.0.1  imported 2 objects  [root@localhost bin]# |

可以看到导入数据的时候会隐式创建表结构

#### 8.7.2 导入CSV数据

我们先将表user删除掉，以便演示效果

|  |
| --- |
| > db.user.drop();  true  > show collections;  system.indexes  > |

然后导入数据

|  |
| --- |
| [root@localhost bin]# ./mongoimport -d my\_mongodb -c user --type csv --headerline --file user\_csv.dat  connected to: 127.0.0.1  imported 3 objects  [root@localhost bin]# |

**参数说明:**

l  -type 指明要导入的文件格式

l  -headerline 批明不导入第一行，因为第一行是列名

l  -file  指明要导入的文件路径

|  |
| --- |
| 注意:  CSV格式良好，主流数据库都支持导出为CSV的格式，所以这种格式非常利于异构数据迁移 |

### 8.8 数据备份mongodump

可以用mongodump来做MongoDB的库或表级别的备份，下面举例说明:

备份my\_mongodb数据库

|  |
| --- |
| [root@localhost bin]# ./mongodump -d my\_mongodb  connected to: 127.0.0.1  DATABASE: my\_mongodb     to     dump/my\_mongodb          my\_mongodb.system.indexes to dump/my\_mongodb/system.indexes.bson                   1 objects          my\_mongodb.user to dump/my\_mongodb/user.bson                   2 objects  [root@localhost bin]# ll  总计 67648  -rwxr-xr-x 1 root root 7508756 2011-04-06 bsondump  drwxr-xr-x 3 root root    4096 04-10 23:54 dump  -rwxr-xr-x 1 root root 2978016 2011-04-06 mongo |

此时会在当前目录下创建一个dump目录，用于存放备份出来的文件

也可以指定备份存放的目录，

|  |
| --- |
| [root@localhost bin]#./mongodump -d my\_mongodb -o my\_mongodb\_dump  connected to: 127.0.0.1  DATABASE: my\_mongodb     to     my\_mongodb\_dump/my\_mongodb          my\_mongodb.system.indexes to my\_mongodb\_dump/my\_mongodb/system.indexes.bson                   1 objects          my\_mongodb.user to my\_mongodb\_dump/my\_mongodb/user.bson                   2 objects  [root@localhost bin]# |

这个例子中将备份的文件存在了当前目录下的my\_mongodb\_dump目录下

### 8.9 数据恢复mongorestore

由于刚刚已经做了备份，所以我们先将库my\_mongodb删除掉

|  |
| --- |
| > use my\_mongodb  switched to db my\_mongodb  > db.dropDatabase()  { "dropped" : "my\_mongodb", "ok" : 1 }  > show dbs  admin   (empty)  local   (empty)  test    (empty)  > |

接下来我们进行数据库恢复

|  |
| --- |
| [root@localhost bin]# ./mongorestore -d my\_mongodb my\_mongodb\_dump/\*  connected to: 127.0.0.1  Wed Apr 11 00:03:03 my\_mongodb\_dump/my\_mongodb/user.bson  Wed Apr 11 00:03:03      going into namespace [my\_mongodb.user]  Wed Apr 11 00:03:03      2 objects found  Wed Apr 11 00:03:03 my\_mongodb\_dump/my\_mongodb/system.indexes.bson  Wed Apr 11 00:03:03      going into namespace [my\_mongodb.system.indexes]  Wed Apr 11 00:03:03 { name: "\_id\_", ns: "my\_mongodb.user", key: { \_id: 1 }, v: 0 }  Wed Apr 11 00:03:03      1 objects found  [root@localhost bin]# |

经验证数据库又回来了，其实要是想恢复库，也大可不必先删除my\_mongodb库，只要指明 –drop参数，就可以在恢复的时候先删除表然后再向表中插入数据的

## [【MongoDB for Java】Java操作MongoDB](http://www.cnblogs.com/hoojo/archive/2011/06/02/2068665.html)

上一篇文章: <http://www.cnblogs.com/hoojo/archive/2011/06/01/2066426.html>介绍到了在MongoDB的控制台完成MongoDB的数据操作，通过前一篇文章我们对MongoDB有了全面的认识和理解。现在我们就用Java来操作MongoDB的数据。

开发环境：

System：Windows

IDE：eclipse、MyEclipse 8

Database：mongoDB

开发依赖库：

JavaEE5、mongo-2.5.3.jar、junit-4.8.2.jar

Email：hoojo\_@126.com

Blog：<http://blog.csdn.net/IBM_hoojo>

<http://hoojo.cnblogs.com/>

#### ****一、准备工作****

1、 首先，下载mongoDB对Java支持的驱动包

驱动包下载地址：<https://github.com/mongodb/mongo-java-driver/downloads>

mongoDB对Java的相关支持、技术：<http://www.mongodb.org/display/DOCS/Java+Language+Center>

驱动源码下载：<https://download.github.com/mongodb-mongo-java-driver-r2.6.1-7-g6037357.zip>

在线查看源码：<https://github.com/mongodb/mongo-java-driver>

2、 下面建立一个JavaProject工程，导入下载下来的驱动包。即可在Java中使用mongoDB，目录如下：

[](http://images.cnblogs.com/cnblogs_com/hoojo/201106/201106021635553981.jpg)

#### ****二、Java操作MongoDB示例****

在本示例之前你需要启动mongod.exe的服务，启动后，下面的程序才能顺利执行；

1、 建立SimpleTest.java，完成简单的mongoDB数据库操作

Mongo mongo = new Mongo();

这样就创建了一个MongoDB的数据库连接对象，它默认连接到当前机器的localhost地址，端口是27017。

DB db = mongo.getDB(“test”);

这样就获得了一个test的数据库，如果mongoDB中没有创建这个数据库也是可以正常运行的。如果你[读过上一篇文章](http://www.cnblogs.com/hoojo/archive/2011/06/01/2066426.html)就知道，mongoDB可以在没有创建这个数据库的情况下，完成数据的添加操作。当添加的时候，没有这个库，mongoDB会自动创建当前数据库。

得到了db，下一步我们要获取一个“聚集集合DBCollection”，通过db对象的getCollection方法来完成。

DBCollection users = db.getCollection("users");

这样就获得了一个DBCollection，它相当于我们数据库的“表”。

查询所有数据

DBCursor cur = users.find();

while (cur.hasNext()) {

System.out.println(cur.next());

}

完整源码

package com.hoo.test;

import java.net.UnknownHostException;

import com.mongodb.DB;

import com.mongodb.DBCollection;

import com.mongodb.DBCursor;

import com.mongodb.Mongo;

import com.mongodb.MongoException;

import com.mongodb.util.JSON;

/\*\*

\* <b>function:</b>MongoDB 简单示例

\* @author hoojo

\* @createDate 2011-5-24 下午02:42:29

\* @file SimpleTest.java

\* @package com.hoo.test

\* @project MongoDB

\* @blog http://blog.csdn.net/IBM\_hoojo

\* @email hoojo\_@126.com

\* @version 1.0

\*/

public class SimpleTest {

public static void main(String[] args) throws UnknownHostException, MongoException {

Mongo mg = new Mongo();

//查询所有的Database

for (String name : mg.getDatabaseNames()) {

System.out.println("dbName: " + name);

}

DB db = mg.getDB("test");

//查询所有的聚集集合

for (String name : db.getCollectionNames()) {

System.out.println("collectionName: " + name);

}

DBCollection users = db.getCollection("users");

//查询所有的数据

DBCursor cur = users.find();

while (cur.hasNext()) {

System.out.println(cur.next());

}

System.out.println(cur.count());

System.out.println(cur.getCursorId());

System.out.println(JSON.serialize(cur));

}

}

2、 完成CRUD操作，首先建立一个MongoDB4CRUDTest.java，基本测试代码如下：

package com.hoo.test;

import java.net.UnknownHostException;

import java.util.ArrayList;

import java.util.List;

import org.bson.types.ObjectId;

import org.junit.After;

import org.junit.Before;

import org.junit.Test;

import com.mongodb.BasicDBObject;

import com.mongodb.Bytes;

import com.mongodb.DB;

import com.mongodb.DBCollection;

import com.mongodb.DBCursor;

import com.mongodb.DBObject;

import com.mongodb.Mongo;

import com.mongodb.MongoException;

import com.mongodb.QueryOperators;

import com.mongodb.util.JSON;

/\*\*

\* <b>function:</b>实现MongoDB的CRUD操作

\* @author hoojo

\* @createDate 2011-6-2 下午03:21:23

\* @file MongoDB4CRUDTest.java

\* @package com.hoo.test

\* @project MongoDB

\* @blog http://blog.csdn.net/IBM\_hoojo

\* @email hoojo\_@126.com

\* @version 1.0

\*/

public class MongoDB4CRUDTest {

private Mongo mg = null;

private DB db;

private DBCollection users;

@Before

public void init() {

try {

mg = new Mongo();

//mg = new Mongo("localhost", 27017);

} catch (UnknownHostException e) {

e.printStackTrace();

} catch (MongoException e) {

e.printStackTrace();

}

//获取temp DB；如果默认没有创建，mongodb会自动创建

db = mg.getDB("temp");

//获取users DBCollection；如果默认没有创建，mongodb会自动创建

users = db.getCollection("users");

}

@After

public void destory() {

if (mg != null)

mg.close();

mg = null;

db = null;

users = null;

System.gc();

}

public void print(Object o) {

System.out.println(o);

}

}

3、 添加操作

在添加操作之前，我们需要写个查询方法，来查询所有的数据。代码如下：

/\*\*

\* <b>function:</b> 查询所有数据

\* @author hoojo

\* @createDate 2011-6-2 下午03:22:40

\*/

private void queryAll() {

print("查询users的所有数据：");

//db游标

DBCursor cur = users.find();

while (cur.hasNext()) {

print(cur.next());

}

}

@Test

public void add() {

//先查询所有数据

queryAll();

print("count: " + users.count());

DBObject user = new BasicDBObject();

user.put("name", "hoojo");

user.put("age", 24);

//users.save(user)保存，getN()获取影响行数

//print(users.save(user).getN());

//扩展字段，随意添加字段，不影响现有数据

user.put("sex", "男");

print(users.save(user).getN());

//添加多条数据，传递Array对象

print(users.insert(user, new BasicDBObject("name", "tom")).getN());

//添加List集合

List<DBObject> list = new ArrayList<DBObject>();

list.add(user);

DBObject user2 = new BasicDBObject("name", "lucy");

user.put("age", 22);

list.add(user2);

//添加List集合

print(users.insert(list).getN());

//查询下数据，看看是否添加成功

print("count: " + users.count());

queryAll();

}

4、 删除数据

@Test

public void remove() {

queryAll();

print("删除id = 4de73f7acd812d61b4626a77：" + users.remove(new BasicDBObject("\_id", new ObjectId("4de73f7acd812d61b4626a77"))).getN());

print("remove age >= 24: " + users.remove(new BasicDBObject("age", new BasicDBObject("$gte", 24))).getN());

}

5、 修改数据

@Test

public void modify() {

print("修改：" + users.update(new BasicDBObject("\_id", new ObjectId("4dde25d06be7c53ffbd70906")), new BasicDBObject("age", 99)).getN());

print("修改：" + users.update(

new BasicDBObject("\_id", new ObjectId("4dde2b06feb038463ff09042")),

new BasicDBObject("age", 121),

true,//如果数据库不存在，是否添加

false//多条修改

).getN());

print("修改：" + users.update(

new BasicDBObject("name", "haha"),

new BasicDBObject("name", "dingding"),

true,//如果数据库不存在，是否添加

true//false只修改第一天，true如果有多条就不修改

).getN());

//当数据库不存在就不修改、不添加数据，当多条数据就不修改

//print("修改多条：" + coll.updateMulti(new BasicDBObject("\_id", new ObjectId("4dde23616be7c19df07db42c")), new BasicDBObject("name", "199")));

}

6、 查询数据

@Test

public void query() {

//查询所有

//queryAll();

//查询id = 4de73f7acd812d61b4626a77

print("find id = 4de73f7acd812d61b4626a77: " + users.find(new BasicDBObject("\_id", new ObjectId("4de73f7acd812d61b4626a77"))).toArray());

//查询age = 24

print("find age = 24: " + users.find(new BasicDBObject("age", 24)).toArray());

//查询age >= 24

print("find age >= 24: " + users.find(new BasicDBObject("age", new BasicDBObject("$gte", 24))).toArray());

print("find age <= 24: " + users.find(new BasicDBObject("age", new BasicDBObject("$lte", 24))).toArray());

print("查询age!=25：" + users.find(new BasicDBObject("age", new BasicDBObject("$ne", 25))).toArray());

print("查询age in 25/26/27：" + users.find(new BasicDBObject("age", new BasicDBObject(QueryOperators.IN, new int[] { 25, 26, 27 }))).toArray());

print("查询age not in 25/26/27：" + users.find(new BasicDBObject("age", new BasicDBObject(QueryOperators.NIN, new int[] { 25, 26, 27 }))).toArray());

print("查询age exists 排序：" + users.find(new BasicDBObject("age", new BasicDBObject(QueryOperators.EXISTS, true))).toArray());

print("只查询age属性：" + users.find(null, new BasicDBObject("age", true)).toArray());

print("只查属性：" + users.find(null, new BasicDBObject("age", true), 0, 2).toArray());

print("只查属性：" + users.find(null, new BasicDBObject("age", true), 0, 2, Bytes.QUERYOPTION\_NOTIMEOUT).toArray());

//只查询一条数据，多条去第一条

print("findOne: " + users.findOne());

print("findOne: " + users.findOne(new BasicDBObject("age", 26)));

print("findOne: " + users.findOne(new BasicDBObject("age", 26), new BasicDBObject("name", true)));

//查询修改、删除

print("findAndRemove 查询age=25的数据，并且删除: " + users.findAndRemove(new BasicDBObject("age", 25)));

//查询age=26的数据，并且修改name的值为Abc

print("findAndModify: " + users.findAndModify(new BasicDBObject("age", 26), new BasicDBObject("name", "Abc")));

print("findAndModify: " + users.findAndModify(

new BasicDBObject("age", 28), //查询age=28的数据

new BasicDBObject("name", true), //查询name属性

new BasicDBObject("age", true), //按照age排序

false, //是否删除，true表示删除

new BasicDBObject("name", "Abc"), //修改的值，将name修改成Abc

true,

true));

queryAll();

}

mongoDB不支持联合查询、子查询，这需要我们自己在程序中完成。将查询的结果集在Java查询中进行需要的过滤即可。

7、 其他操作

public void testOthers() {

DBObject user = new BasicDBObject();

user.put("name", "hoojo");

user.put("age", 24);

//JSON 对象转换

print("serialize: " + JSON.serialize(user));

//反序列化

print("parse: " + JSON.parse("{ \"name\" : \"hoojo\" , \"age\" : 24}"));

print("判断temp Collection是否存在: " + db.collectionExists("temp"));

//如果不存在就创建

if (!db.collectionExists("temp")) {

DBObject options = new BasicDBObject();

options.put("size", 20);

options.put("capped", 20);

options.put("max", 20);

print(db.createCollection("account", options));

}

//设置db为只读

db.setReadOnly(true);

//只读不能写入数据

db.getCollection("test").save(user);

}

好了，这里基本上就介绍这么多Java操作MongoDB的方法。

## [用C#操作Mongodb(c#mongodb驱动)](http://blog.csdn.net/zzq900503/article/details/8599681)

分类： [mongodb](http://blog.csdn.net/q383965374/article/category/1327953)2013-02-26 13:53 644人阅读 [评论](http://blog.csdn.net/zzq900503/article/details/8599681#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/zzq900503/article/details/8599681#report)

MongoDB的C#驱动基于.Net 3.5的 必要用.net 3.5以上的框架

1.对数据库的操作主要还是增删改查

2.了解c#操作mapreduce的语法

我们先布置驱动环境，再通过实例来了解一下相关操作语法。

方法一：驱动工程直接下载：

https://github.com/mongodb/mongo-csharp-driver/downloads

下载驱动。驱动有两个文件

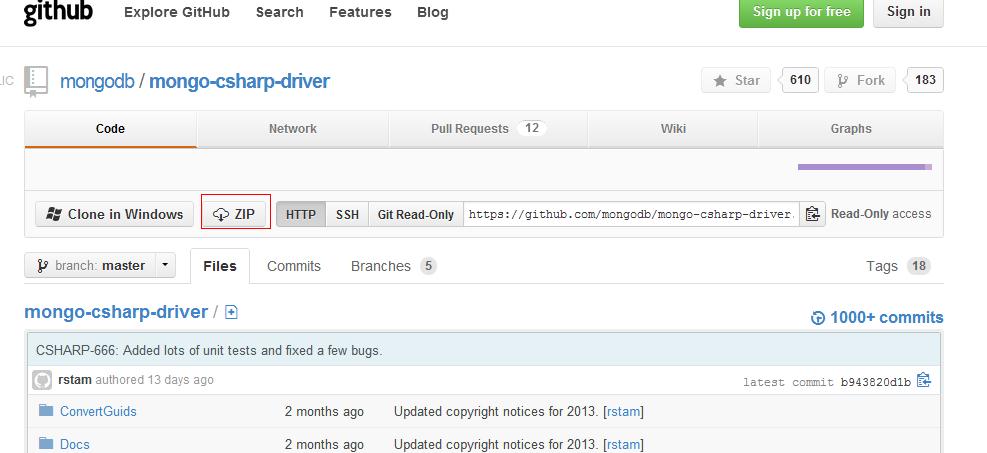
1. MongoDB.Bson.dll
2. MongoDB.Driver.dll

可以直接下载这两个驱动，或者遵守下载源码进行编译生成。下载的源码可以看些test例子

方法二：驱动源码下载后编译得到驱动工程

地址：

https://github.com/mongodb/mongo-csharp-driver



下载的是源代码，进行编译后就可以得到MongoDB.Bson.dll，MongoDB.Driver.dll这两个驱动。

驱动说明：

http://docs.mongodb.org/ecosystem/tutorial/use-csharp-driver/

驱动说明中文翻译：

http://www.bwxxkj.com/a/jishuzhongxin/shujukukaifa/2013/0109/160953.html

驱动API说明：

http://api.mongodb.org/csharp/current/

下面通过实例操作来熟悉一下：

引入驱动：

在新建的c＃工程中添加这两个dll文件并引用它们，并且应用如下定名空间

至少要引用如下定名空间

using MongoDB.Bson;

using MongoDB.Driver;

用得比较多的命名空间

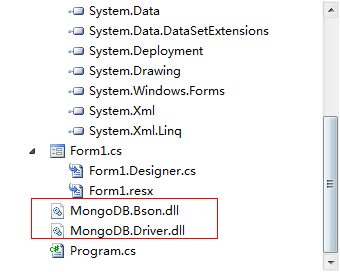
**[csharp]** [view plaincopy](http://blog.csdn.net/zzq900503/article/details/8599681)

1. **using** MongoDB.Driver.Builders;
3. **using** MongoDB.Driver.GridFS;
5. **using** MongoDB.Driver.Linq;

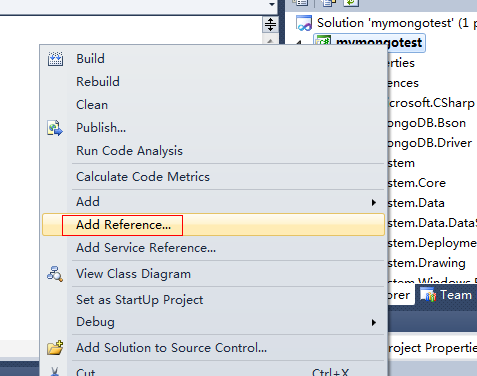
可能用到的命名空间

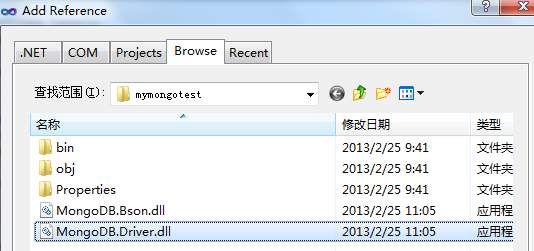
**[csharp]** [view plaincopy](http://blog.csdn.net/zzq900503/article/details/8599681)

1. **using** MongoDB.Bson.IO;
3. **using** MongoDB.Bson.Serialization;
5. **using** MongoDB.Bson.Serialization.Attributes;
7. **using** MongoDB.Bson.Serialization.Conventions;
9. **using** MongoDB.Bson.Serialization.IdGenerators;
11. **using** MongoDB.Bson.Serialization.Options;
13. **using** MongoDB.Bson.Serialization.Serializers;
15. **using** MongoDB.Driver.Wrappers;

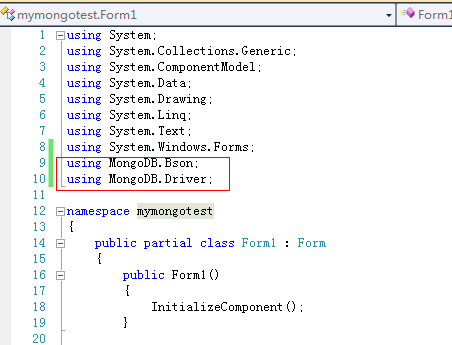
把这两文件复制进工程中  


引用它们

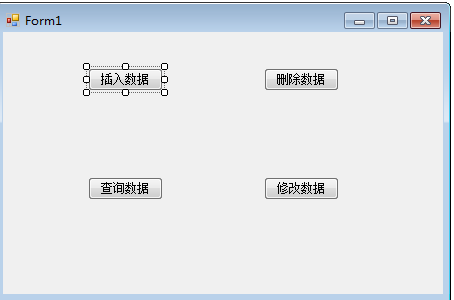




添加命名空间



好 可以用了 下面开始进行操作



对mongodb的数据进行操作有两种方法：

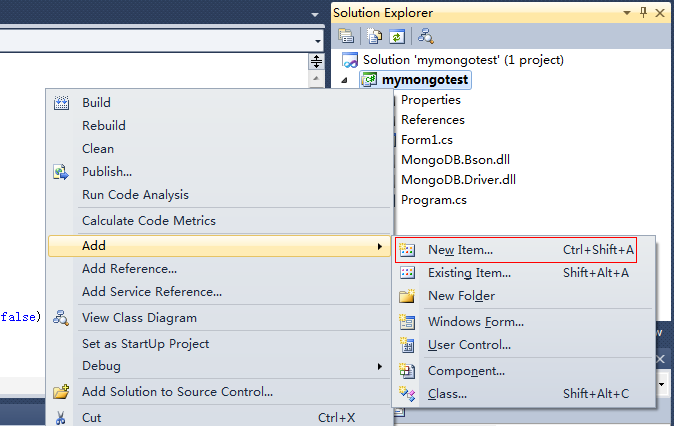
1.先构造好类的结构，用这个结构操作数据。

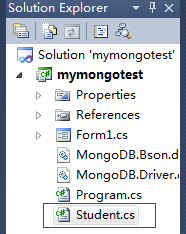
2.直接操作数据。

插入数据：

方法一：

增加一个Student类，构建student的构造。





类的代码如下：

**[csharp]** [view plaincopy](http://blog.csdn.net/zzq900503/article/details/8599681)

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
6. **namespace** mymongotest
7. {
8. **class**  Student\_Insert
9. {

12. **public** **string** 姓名{**get**;**set**;}
14. **public** **string** 学号{**get**;**set**;}
16. **public** Information[] 个人信息{**get**;**set**;}




22. }
23. **class** Information
24. {
26. **public** **string** Name { **get**; **set**; }
28. **public** **string** Value { **get**; **set**; }
30. }
32. }

引用命名空间：

**[csharp]** [view plaincopy](http://blog.csdn.net/zzq900503/article/details/8599681)

1. **using** MongoDB.Bson.Serialization;
2. **using** MongoDB.Driver.Builders;

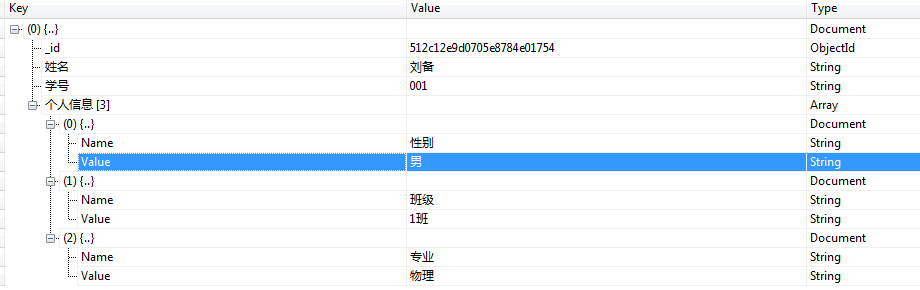
插入按钮的代码：

**[csharp]** [view plaincopy](http://blog.csdn.net/zzq900503/article/details/8599681)

1. **private** **void** button1\_Click(**object** sender, EventArgs e)
2. {
3. MongoClient client;
4. MongoServer server;
5. MongoDatabase database;
7. var conStr = "mongodb://192.168.0.188";
8. client = **new** MongoClient(conStr);
10. server = client.GetServer();
11. database = server.GetDatabase("test");
12. var coll = database.GetCollection("student");
14. var sInsert = @"{""姓名"":""刘备"",""学号"":""001"",
15. ""个人信息"":[
16. {""Name"":""性别"",""Value"":""男""},
17. {""Name"":""班级"",""Value"":""1班""},
18. {""Name"":""专业"",""Value"":""物理""}
19. ]
20. }";
21. var bd = BsonSerializer.Deserialize<Student\_Insert>(sInsert);
22. coll.Insert(bd);
24. MessageBox.Show("入库结束");
25. }

运行，点击插入按钮

用MongoVue查看结果



插入成功

方法二：

不构造类结构，直接创建文档插入数据。可以根据你的需要构造成不同的格式。

这里 我们构造跟方法一一样的格式：

**[csharp]** [view plaincopy](http://blog.csdn.net/zzq900503/article/details/8599681)

1. MongoClient client;
2. MongoServer server;
3. MongoDatabase database;

6. var conStr = "mongodb://192.168.0.188";
7. client = **new** MongoClient(conStr);

10. server = client.GetServer();
11. database = server.GetDatabase("test");
12. var coll = database.GetCollection("student");





19. BsonDocument bd = **new** BsonDocument();
20. bd.Add(**new** BsonElement("姓名", "赵云"));
21. bd.Add(**new** BsonElement("学号", "002"));
22. //个人信息格式构造一： bd.Add(new BsonDocument{{"个人信息",new BsonDocument{{"性别","男"},{"班级","1班"},{"专业","数学"}}}});

25. /\* 个人信息格式构造二：
27. BsonArray ba = new BsonArray();
29. BsonDocument information = new BsonDocument();

32. information.Add(new  BsonElement("性别", "男"));
33. information.Add(new  BsonElement("班级", "002"));
34. information.Add(new  BsonElement("专业", "数学"));



39. ba.Add(information);
40. bd.Add(new BsonDocument("个人信息", ba));
42. \*/





49. //个人信息格式3
50. //BsonArray ba = new BsonArray();



55. //ba.Add(new BsonDocument("性别", "男"));
56. //ba.Add(new BsonDocument("班级", "002"));
57. //ba.Add(new BsonDocument("专业", "数学"));

60. //bd.Add(new BsonDocument("个人信息", ba));

63. //个人信息格式4
64. BsonArray ba = **new** BsonArray();



69. ba.Add(**new** BsonDocument { { "Name", "性别" }, { "Value", "男" } });
70. ba.Add(**new** BsonDocument { { "Name", "班级" }, { "Value", "2班" } });
71. ba.Add(**new** BsonDocument { { "Name", "专业" }, { "Value", "物理" } });

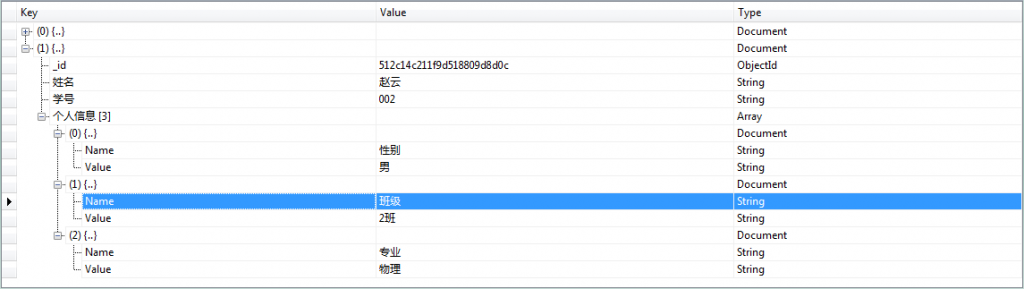
74. bd.Add(**new** BsonDocument("个人信息", ba));



79. coll.Insert(bd);

82. MessageBox.Show("入库结束");

结果如下：



插入成功

查询数据：

在已建的类Student代码里添加类结构：Student\_Query

添加后类Student.cs的代码为：

**[csharp]** [view plaincopy](http://blog.csdn.net/zzq900503/article/details/8599681)

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
6. **namespace** mymongotest
7. {
8. **class** Student\_Insert
9. {


13. **public** **string** 姓名{**get**;**set**;}
15. **public** **string** 学号{**get**;**set**;}
17. **public** Information[] 个人信息{**get**;**set**;}




23. }
24. **class** Information
25. {
27. **public** **string** Name { **get**; **set**; }
29. **public** **string** Value { **get**; **set**; }



34. }

37. **class** Student\_Query
38. {

41. **public** Object \_id { **get**; **set**; }
42. **public** **string** 姓名 { **get**; **set**; }
44. **public** **string** 学号 { **get**; **set**; }
46. **public** Information[] 个人信息 { **get**; **set**; }




52. }
54. }

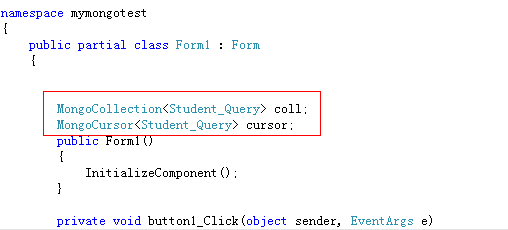
情景一：通过索引值取数据----通常用于取上一条，下一条数据。(用到类结构)

在窗体下声明集合coll和浮标cucor:

**[csharp]** [view plaincopy](http://blog.csdn.net/zzq900503/article/details/8599681)

1. MongoCollection<Student\_Query> coll;
2. MongoCursor<Student\_Query> cursor;

声明位置如图：

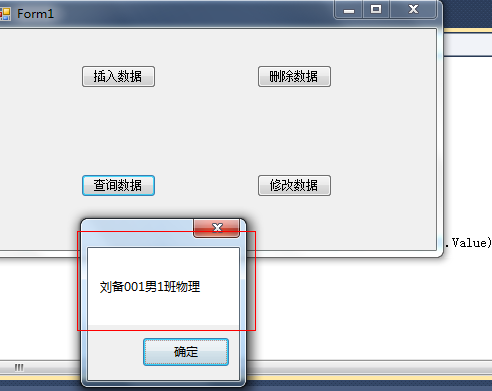


查询按钮的代码：

**[csharp]** [view plaincopy](http://blog.csdn.net/zzq900503/article/details/8599681)

1. **private** **void** button3\_Click(**object** sender, EventArgs e)
2. {
3. MongoClient client;
4. MongoServer server;
5. MongoDatabase database;
7. var conStr = "mongodb://192.168.0.188";
8. client = **new** MongoClient(conStr);
10. server = client.GetServer();
11. database = server.GetDatabase("test");
12. coll = database.GetCollection< Student\_Query>("student");
13. cursor = coll.FindAll();
14. var ex = cursor.ElementAt< Student\_Query>(0);//取第一个数据
15. **string** sResult = "";
16. sResult = **string**.Format("{0}{1}{2}{3}{4}", ex.姓名, ex.学号, ex.个人信息[0].Value, ex.个人信息[1].Value, ex.个人信息[2].Value);
18. MessageBox.Show(sResult);
19. }

结果：



情景二：用条件查询

情景二方法一：（用类结构）

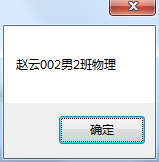
查询按钮代码如下：

**[csharp]** [view plaincopy](http://blog.csdn.net/zzq900503/article/details/8599681)

1. MongoClient client;
2. MongoServer server;
3. MongoDatabase database;
5. var conStr = "mongodb://192.168.0.188";
6. client = **new** MongoClient(conStr);
8. server = client.GetServer();
9. database = server.GetDatabase("test");
10. coll = database.GetCollection<Student\_Query>("student");

13. // var bs = coll.FindOne(Query.EQ("\_id", new ObjectId("512c14c211f9d518809d8d0c")));
14. var bs = coll.FindOne(Query.EQ("姓名", "赵云"));
16. **string** sResult = "";
17. sResult = **string**.Format("{0}{1}{2}{3}{4}", bs.姓名, bs.学号, bs.个人信息[0].Value, bs.个人信息[1].Value, bs.个人信息[2].Value);
19. MessageBox.Show(sResult);

结果：



情景二方法二：（不用类结构）

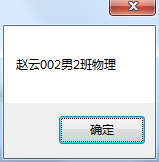
查询按钮代码如下：

**[csharp]** [view plaincopy](http://blog.csdn.net/zzq900503/article/details/8599681)

1. MongoClient client;
2. MongoServer server;
3. MongoDatabase database;
5. var conStr = "mongodb://192.168.0.188";
6. client = **new** MongoClient(conStr);
8. server = client.GetServer();
9. database = server.GetDatabase("test");
10. var  coll = database.GetCollection("student");

13. // var bs = coll.FindOne(Query.EQ("\_id", new ObjectId("512c14c211f9d518809d8d0c")));
14. var bs = coll.FindOne(Query.EQ("姓名", "赵云"));
16. **string** sResult = "";
17. sResult = **string**.Format("{0}{1}{2}{3}{4}", bs.AsBsonDocument["姓名"].ToString(), bs.AsBsonDocument["学号"].ToString(), bs.AsBsonDocument["个人信息"].AsBsonArray[0].AsBsonDocument["Value"].ToString(), bs.AsBsonDocument["个人信息"].AsBsonArray[1].AsBsonDocument["Value"].ToString(), bs.AsBsonDocument["个人信息"].AsBsonArray[2].AsBsonDocument["Value"].ToString());
19. MessageBox.Show(sResult);

结果：



查询常用语句：

求文档长度

**[csharp]** [view plaincopy](http://blog.csdn.net/zzq900503/article/details/8599681)

1. var bs = coll.FindAll();
2. **string** i=  bs.Count().ToString();

求文档中数组长度

**[csharp]** [view plaincopy](http://blog.csdn.net/zzq900503/article/details/8599681)

1. var bs = coll.FindOne(Query.EQ("姓名", "赵云"));
3. **string** i = bs.AsBsonDocument["个人信息"].AsBsonArray.Count().ToString();

文档中是否包含某个元素：

**[csharp]** [view plaincopy](http://blog.csdn.net/zzq900503/article/details/8599681)

1. **if** (bs.AsBsonDocument.Contains("姓名"))

遍历一个文档中的所有属性：

**[csharp]** [view plaincopy](http://blog.csdn.net/zzq900503/article/details/8599681)

1. var bs = coll\_fieldConnect.FindOne(Query.EQ("id", process\_id));
2. StringBuilder sb = **new** StringBuilder();
3. **foreach** (BsonElement d **in** bs)
4. {



9. sb.AppendFormat("{0}:\t{1}\n", d.Name, d.Value);
11. }
12. richTextBox1.Text = sb.ToString();

修改数据：

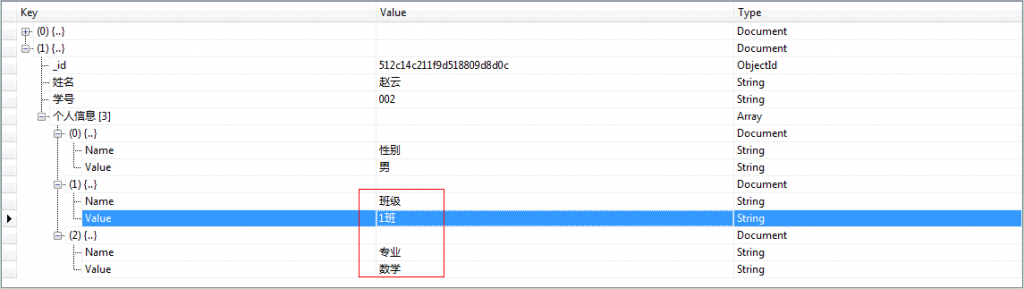
修改按钮代码如下：

**[csharp]** [view plaincopy](http://blog.csdn.net/zzq900503/article/details/8599681)

1. <span style="color:#000099;">  </span>          MongoClient client;
2. MongoServer server;
3. MongoDatabase database;
5. var conStr = "mongodb://192.168.0.188";
6. client = **new** MongoClient(conStr);
8. server = client.GetServer();
9. database = server.GetDatabase("test");
10. var coll = database.GetCollection("student");
12. BsonDocument bd = **new** BsonDocument();
13. bd.Add(**new** BsonElement("姓名", "赵云"));
14. bd.Add(**new** BsonElement("学号", "002"));
15. BsonArray ba = **new** BsonArray();

18. ba.Add(**new** BsonDocument { { "Name", "性别" }, { "Value", "男" } });
19. ba.Add(**new** BsonDocument { { "Name", "班级" }, { "Value", "1班" } });
20. ba.Add(**new** BsonDocument { { "Name", "专业" }, { "Value", "数学" } });
22. bd.Add(**new** BsonDocument("个人信息", ba));
24. IMongoQuery query = Query.EQ("学号", "002");
26. coll.Update(query, **new** UpdateDocument(bd));  MessageBox.Show("修改结束");

结果如下： 转专业班级成功



删除数据：

删除按妞的代码如下：

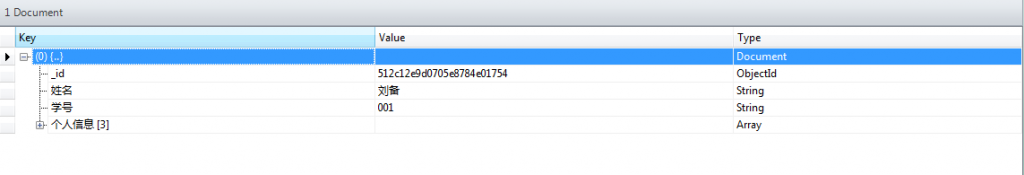
**[csharp]** [view plaincopy](http://blog.csdn.net/zzq900503/article/details/8599681)

1. **private** **void** button2\_Click(**object** sender, EventArgs e)
2. {
3. MongoClient client;
4. MongoServer server;
5. MongoDatabase database;
7. var conStr = "mongodb://192.168.0.188";
8. client = **new** MongoClient(conStr);
10. server = client.GetServer();
11. database = server.GetDatabase("test");
12. var coll = database.GetCollection("student");



17. IMongoQuery query = Query.EQ("学号", "002");
19. coll.Remove(query);
20. MessageBox.Show("删除结束");
22. }

结果：



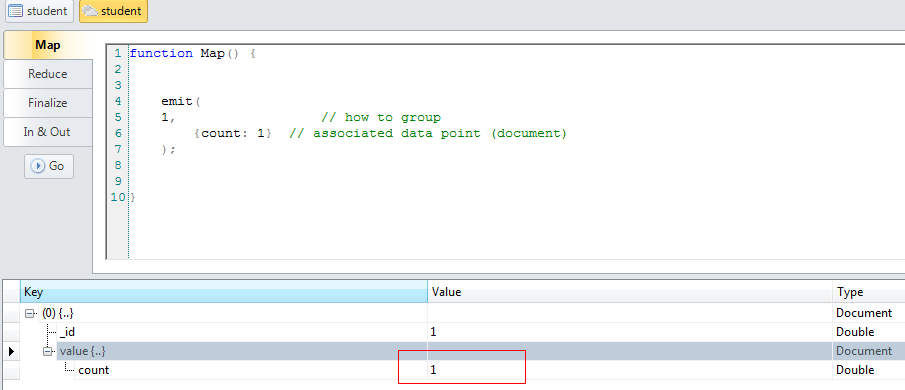
删除成功

2.对mapreduce的操作语法

我们先在mongovue中测试好要用的mapreduce

这里我们写一个统计文档数的mapreduce，详见[mongodb mapreduce分析](http://blog.csdn.net/q383965374/article/details/8510030)

得到结果为 文档数为1



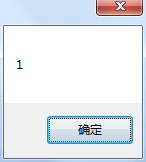
用c#实现这个功能 按钮代码为：

**[csharp]** [view plaincopy](http://blog.csdn.net/zzq900503/article/details/8599681)

1. **private** **void** button5\_Click(**object** sender, EventArgs e)
2. {
3. MongoClient client;
4. MongoServer server;
5. MongoDatabase database;
7. var conStr = "mongodb://192.168.0.188";
8. client = **new** MongoClient(conStr);
10. server = client.GetServer();
11. database = server.GetDatabase("test");
12. var coll = database.GetCollection("student");
14. BsonJavaScript map = **new** BsonJavaScript(@"
15. function map() {
16. emit(
17. 1,
18. {count: 1}  );
19. }");
20. BsonJavaScript reduce = **new** BsonJavaScript(@"function(key,values){
21. var total=0;
22. values.forEach(function(val) {
23. total   +=  val.count;
25. });
26. **return** total;
27. };");


31. var  result = coll.MapReduce(map, reduce);
32. **string** count = "";
33. **foreach** (var s **in** result.InlineResults)
34. {
35. count = s.AsBsonDocument["value"].AsBsonDocument["count"].ToString();
36. }
38. MessageBox.Show(count);
39. }

结果：



## [java操作mongodb](http://blog.csdn.net/zzq900503/article/details/12051373)

首先下载驱动:

<http://download.csdn.net/detail/q383965374/6209691>

官网下载驱动:

<https://github.com/mongodb/mongo-java-driver/downloads>

驱动源码下载:

<https://codeload.github.com/mongodb/mongo-java-driver/zip/master>

在线查看源码:

<https://github.com/mongodb/mongo-java-driver>

对mongodb的操作主要还是 增删改查

本文的内容:

1.引用驱动

2.添加命名空间

3.连接数据库

4.插入

5.查询

6.修改

7.删除

8.增删改查完整代码

9.可能用到的用法

10.mapreduce的用法完整代码

1.引用驱动

把下载的驱动jar放到工程中 引用:



2.添加命名空间

使用mongoDB需要导入以下类，当然不是全部需要，用到的类就导入。(也可以用到的时候再导入)

**[java]** [view plaincopy](http://blog.csdn.net/zzq900503/article/details/12051373)

1. **import** com.mongodb.Mongo;
2. **import** com.mongodb.DB;
3. **import** com.mongodb.DBCollection;
4. **import** com.mongodb.BasicDBObject;
5. **import** com.mongodb.DBObject;
6. **import** com.mongodb.DBCursor;
7. **import** com.mongodb.ObjectId;

3.连接数据库

**[java]** [view plaincopy](http://blog.csdn.net/zzq900503/article/details/12051373)

1. Mongo mg=**new** Mongo("192.168.0.121",27017);
2. DB db = mg.getDB("test"); //相当于库名
3. DBCollection coll = db.getCollection("Student");//相当于表名

4.插入

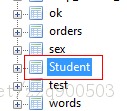
**[java]** [view plaincopy](http://blog.csdn.net/zzq900503/article/details/12051373)

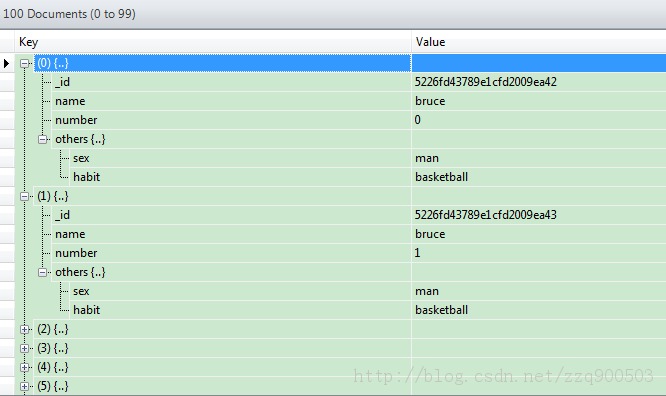
1. //插入数据(批量插入)
2. insert(coll);

**[java]** [view plaincopy](http://blog.csdn.net/zzq900503/article/details/12051373)

1. **private** **static** **void** insert(DBCollection coll) {
2. List<DBObject> datas = **new** ArrayList<DBObject>();
3. **for** (**int** i=0; i < 100; i++) {
4. BasicDBObject bo = **new** BasicDBObject();
5. BasicDBObject xx = **new** BasicDBObject();
6. xx.put("sex", "man");
7. xx.put("habit", "basketball");
9. bo.put("name", "bruce");
10. bo.append("number", i);
12. bo.put("others", xx);
14. datas.add(bo);
15. }
16. coll.insert(datas);
18. System.out.print("插入ok");
19. }

结果:



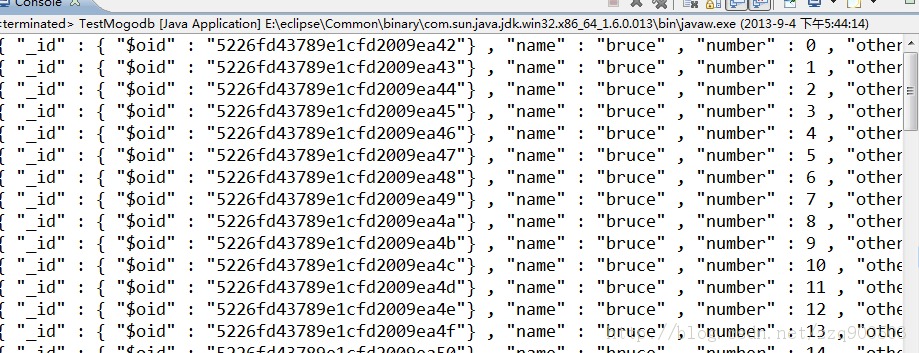


5.查询

**[java]** [view plaincopy](http://blog.csdn.net/zzq900503/article/details/12051373)

1. //查询
2. //查询所有
3. queryAll(coll);
5. //查询一条
6. //      System.out.println(coll.findOne());
7. //加条件查询方法一
8. //      System.out.println(coll.findOne(new BasicDBObject("number",4)));
9. //加条件查询方法二
10. //      BasicDBObject condition = new BasicDBObject();
11. //      condition.put("number",4);
12. //      DBCursor cur = coll.find(condition);
13. //      while(cur.hasNext()) {
14. //      System.out.println(cur.next());
15. //      }

结果:

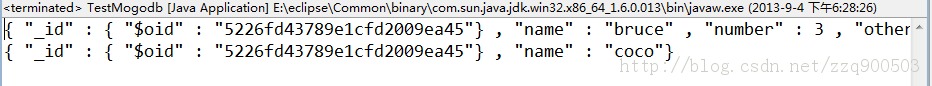


6.修改

**[java]** [view plaincopy](http://blog.csdn.net/zzq900503/article/details/12051373)

1. //修改数据
2. System.out.println(coll.findOne(**new** BasicDBObject("\_id",**new** ObjectId("5226fd43789e1cfd2009ea45"))));
4. coll.update(**new** BasicDBObject("\_id",**new** ObjectId("5226fd43789e1cfd2009ea45")),**new** BasicDBObject("name","coco"));
6. coll.update(**new** BasicDBObject("\_id",**new** ObjectId("5226fd43789e1cfd2009ea45")),**new** BasicDBObject("name","coco"),**true**,**true**);
7. //第一个true代表没有该条数据时进行添加
8. //第二个true表示存在多条时只修改第一条，有多条时不修改.为false时,修改多条
9. System.out.println(coll.findOne(**new** BasicDBObject("\_id",**new** ObjectId("5226fd43789e1cfd2009ea45"))));

结果:

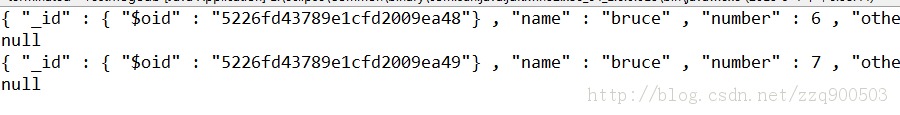


7.删除

**[java]** [view plaincopy](http://blog.csdn.net/zzq900503/article/details/12051373)

1. //删除
2. System.out.println(coll.findOne(**new** BasicDBObject("number",6)));
3. coll.findAndRemove(**new** BasicDBObject("number",6));
4. System.out.println(coll.findOne(**new** BasicDBObject("number",6)));
6. System.out.println(coll.findOne(**new** BasicDBObject("number",7)));
7. coll.remove(**new** BasicDBObject("number",7));
8. System.out.println(coll.findOne(**new** BasicDBObject("number",7)));

结果:



8.增删改查完整代码

**[java]** [view plaincopy](http://blog.csdn.net/zzq900503/article/details/12051373)

1. **package** Test;
3. **import** java.net.UnknownHostException;
4. **import** java.util.ArrayList;
5. **import** java.util.List;
7. **import** org.bson.types.ObjectId;
9. **import** com.mongodb.BasicDBObject;
10. **import** com.mongodb.DB;
11. **import** com.mongodb.DBCollection;
12. **import** com.mongodb.DBCursor;
13. **import** com.mongodb.DBObject;
14. **import** com.mongodb.Mongo;
15. **import** com.mongodb.MongoException;
17. **public** **class** TestMogodb {
19. **public** **static** **void** main(String[] args) **throws** UnknownHostException,MongoException {
20. Mongo mg=**new** Mongo("192.168.0.121",27017);
21. DB db = mg.getDB("test"); //相当于库名
22. DBCollection coll = db.getCollection("Student");//相当于表名

25. //插入数据(批量插入)
26. //      insert(coll);
28. //查询
29. //查询所有
30. //      queryAll(coll);
32. //查询一条
33. //      System.out.println(coll.findOne());
34. //加条件查询方法一
35. //      System.out.println(coll.findOne(new BasicDBObject("number",4)));
36. //加条件查询方法二
37. //      BasicDBObject condition = new BasicDBObject();
38. //      condition.put("number",4);
39. //      DBCursor cur = coll.find(condition);
40. //      while(cur.hasNext()) {
41. //      System.out.println(cur.next());
42. //      }
44. //      //修改数据
45. //      System.out.println(coll.findOne(new BasicDBObject("\_id",new ObjectId("5226fd43789e1cfd2009ea45"))));
46. //
47. //      coll.update(new BasicDBObject("\_id",new ObjectId("5226fd43789e1cfd2009ea45")),new BasicDBObject("name","coco"));
48. //
49. //      //coll.update(new BasicDBObject("\_id",new ObjectId("5226fd43789e1cfd2009ea45")),new BasicDBObject("name","coco"),true,true);
50. //      //第一个true代表没有该条数据时进行添加
51. //      //第二个true表示存在多条时只修改第一条，有多条时不修改.为false时,修改多条
52. //      System.out.println(coll.findOne(new BasicDBObject("\_id",new ObjectId("5226fd43789e1cfd2009ea45"))));
54. //删除
55. System.out.println(coll.findOne(**new** BasicDBObject("number",6)));
56. coll.findAndRemove(**new** BasicDBObject("number",6));
57. System.out.println(coll.findOne(**new** BasicDBObject("number",6)));
59. System.out.println(coll.findOne(**new** BasicDBObject("number",7)));
60. coll.remove(**new** BasicDBObject("number",7));
61. System.out.println(coll.findOne(**new** BasicDBObject("number",7)));
63. }
65. **private** **static** **void** queryAll(DBCollection coll) {
66. DBCursor cur = coll.find();
67. **while**(cur.hasNext()) {
68. System.out.println(cur.next());
69. }
70. }
72. **private** **static** **void** insert(DBCollection coll) {
73. List<DBObject> datas = **new** ArrayList<DBObject>();
74. **for** (**int** i=0; i < 100; i++) {
75. BasicDBObject bo = **new** BasicDBObject();
76. BasicDBObject xx = **new** BasicDBObject();
77. xx.put("sex", "man");
78. xx.put("habit", "basketball");
80. bo.put("name", "bruce");
81. bo.append("number", i);
83. bo.put("others", xx);
85. datas.add(bo);
86. }
87. coll.insert(datas);
89. System.out.print("插入ok");
90. }
92. }

9.可能用到的用法

类转换  
当把一个类对象存到mongoDB后，从mongoDB取出来时使用setObjectClass()将其转换回原来的类。  
public class Tweet implements DBObject {  
  
  
}  
Tweet myTweet = new Tweet();  
myTweet.put("user", "bruce");  
myTweet.put("message", "fun");  
myTweet.put("date", new Date());  
collection.insert(myTweet);  
//转换  
collection.setObjectClass(Tweet);  
Tweet myTweet = (Tweet)collection.findOne();  
  
  
默认ID  
当保存的对象没有设置ID时，mongoDB会默认给该条记录设置一个ID（"\_id"）。  
当然你也可以设置自己指定的ID，如：(在mongoDB中执行用db.users.save({\_id:1,name:'bruce'});)  
BasicDBObject bo = new BasicDBObject();  
bo.put('\_id', 1);  
bo.put('name', 'bruce');  
collection.insert(bo);  
  
  
权限  
判断是否有mongoDB的访问权限，有就返回true，否则返回false。  
boolean auth = db.authenticate(myUserName, myPassword);  
  
  
查看mongoDB数据库列表  
Mongo m = new Mongo();  
for (String s : m.getDatabaseNames()) {  
System.out.println(s);  
}  
  
  
  
  
查看当前库下所有的表名，等于在mongoDB中执行show tables;  
Set<String> colls = db.getCollectionNames();  
for (String s : colls) {  
System.out.println(s);  
}  
  
  
查看一个表的索引  
List<DBObject> list = coll.getIndexInfo();  
for (DBObject o : list) {  
System.out.println(o);  
}  
  
  
删除一个数据库  
Mongo m = new Mongo();  
m.dropDatabase("myDatabaseName");  
  
  
建立mongoDB的链接  
Mongo m = new Mongo("localhost", 27017);  
DB db = m.getDB("myDatabaseName"); //相当于库名  
DBCollection coll = db.getCollection("myUsersTable")；//相当于表名  
  
  
#查询数据  
查询第一条记录  
DBObject firstDoc = coll.findOne();  
findOne()返回一个记录，而find()返回的是DBCursor游标对象。  
  
  
查询全部数据  
DBCursor cur = coll.find();  
while(cur.hasNext()) {  
System.out.println(cur.next());  
}  
  
  
查询记录数量  
coll.find().count();  
coll.find(new BasicDBObject("age", 26)).count();  
  
  
设置条件查询  
BasicDBObject condition = new BasicDBObject();  
condition.put("name", "bruce");  
condition.put("age", 26);  
coll.find(condition);  
  
  
查询部分数据块  
DBCursor cursor = coll.find().skip(0).limit(10);  
while(cursor.hasNext()) {  
System.out.println(cursor.next());  
}  
  
  
比较查询(age > 50)  
BasicDBObject condition = new BasicDBObject();  
condition.put("age", new BasicDBObject("$gt", 50));  
coll.find(condition);  
比较符  
"$gt"： 大于  
"$gte"：大于等于  
"$lt"： 小于  
"$lte"：小于等于  
"$in"： 包含  
//以下条件查询20<age<=30  
condition.put("age", new BasicDBObject("$gt", 20).append("$lte", 30));  
  
  
#插入数据  
批量插入  
List datas = new ArrayList();  
for (int i=0; i < 100; i++) {  
BasicDBObject bo = new BasicDBObject();  
bo.put("name", "bruce");  
bo.append("age", i);  
datas.add(bo);  
}  
coll.insert(datas);  
  
  
正则表达式  
查询所有名字匹配 /joh?n/i 的记录  
Pattern pattern = Pattern.compile("joh?n", CASE\_INSENSITIVE);  
BasicDBObject query = new BasicDBObject("name", pattern);  
DBCursor cursor = coll.find(query);  
-----------------------------------------------------------------  
  
  
模糊查询:  
db.analytics.find({referer\_url: /http:.\*/i});  
db.analytics.find({start\_time:{$gte:new Date(2010,5,28),$lt:new Date

(2010,5,29)},referer\_url:/http:.\*/}).count()

正则表达式  
查询所有名字匹配 /joh?n/i 的记录  
Pattern pattern = Pattern.compile("joh?n", CASE\_INSENSITIVE);  
BasicDBObject query = new BasicDBObject("name", pattern);  
DBCursor cursor = coll.find(query);  
condition.put("name", "bruce");  
condition.put("age", 26);  
coll.find(condition);  
  
  
查询部分数据块  
DBCursor cursor = coll.find().skip(0).limit(10);  
while(cursor.hasNext()) {  
System.out.println(cursor.next());  
}  
  
  
比较查询(age > 50)  
BasicDBObject condition = new BasicDBObject();  
condition.put("age", new BasicDBObject("$gt", 50));  
coll.find(condition);  
比较符  
"$gt"： 大于  
"$gte"：大于等于  
"$lt"： 小于  
"$lte"：小于等于  
"$in"： 包含  
//以下条件查询20<age<=30  
condition.put("age", new BasicDBObject("$gt", 20).append("$lte", 30));  
  
  
#插入数据  
批量插入  
List datas = new ArrayList();  
for (int i=0; i < 100; i++) {  
BasicDBObject bo = new BasicDBObject();  
bo.put("name", "bruce");  
bo.append("age", i);  
datas.add(bo);  
}  
coll.insert(datas);  
  
  
正则表达式  
查询所有名字匹配 /joh?n/i 的记录  
Pattern pattern = Pattern.compile("joh?n", CASE\_INSENSITIVE);  
BasicDBObject query = new BasicDBObject("name", pattern);  
DBCursor cursor = coll.find(query);  
--------  
日期：  
db.cpc\_common.cpc\_click\_log.find({date\_created:{$gte:new Date(2010,5,16)}});  
  
  
---------------------------  
建立collection db.createCollection("mycoll", {size:10000000});  
----------  
索引：db.mycollection.ensureIndex( {active:1} );

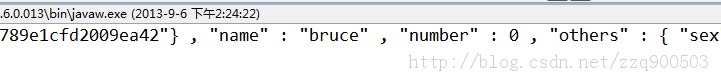
10.mapreduce的用法完整代码

**[java]** [view plaincopy](http://blog.csdn.net/zzq900503/article/details/12051373)

1. **package** Test;
3. **import** com.mongodb.BasicDBObject;
4. **import** com.mongodb.DB;
5. **import** com.mongodb.DBCollection;
6. **import** com.mongodb.DBObject;
7. **import** com.mongodb.MapReduceCommand;
8. **import** com.mongodb.MapReduceOutput;
9. **import** com.mongodb.Mongo;
11. **public** **class** catchMongo{
13. /\*\*
14. \* @param args
15. \*/
16. **public** **static** **void** main(String[] args) {
18. Mongo mongo;
20. **try** {
21. mongo = **new** Mongo("192.168.0.121", 27017);
22. DB db = mongo.getDB("test");
24. DBCollection books = db.getCollection("Student");

27. String map = "function Map(){if(this.number=='0'){emit('result',this);}}";
29. String reduce = "function Reduce(key, values) {return values[0];}";
31. MapReduceCommand cmd = **new** MapReduceCommand(books, map, reduce,
32. **null**, MapReduceCommand.OutputType.INLINE, **null**);
34. MapReduceOutput out = books.mapReduce(cmd);
36. **for** (DBObject o : out.results()) {
37. System.out.println(o.toString());
38. }
39. } **catch** (Exception e) {
40. // TODO Auto-generated catch block
41. e.printStackTrace();
42. }
43. }
44. }

结果：



参考资料：

<http://blog.csdn.net/cdhongwan/article/details/7685288>

<http://www.cnblogs.com/oubo/archive/2012/02/21/2394663.html>

<http://blog.csdn.net/zzq900503/article/details/12051373>

<http://www.cnblogs.com/hoojo/archive/2011/06/02/2068665.html>

<http://www.open-open.com/doc/view/3ef25572e1cd4732a4c1d2d05f2e5e7c>

<http://blog.csdn.net/cdhongwan/article/category/1168386>