[1、MongoDB简介（了解） 2](#_Toc1461001523)

[2、MongoDB的安装与配置（重点） 3](#_Toc8564226)

[3、MongoDB的基础操作（重点） 5](#_Toc1308564639)

[4、数据操作（重点） 7](#_Toc779509275)

[4.1、数据增加 7](#_Toc286337822)

[4.2、数据查询 8](#_Toc2077876027)

[4.2.1、关系运算 9](#_Toc1872066807)

[4.2.2、逻辑运算 10](#_Toc1940577982)

[4.2.3、求模 10](#_Toc235174715)

[4.2.4、范围查询 11](#_Toc1426442819)

[4.2.5、数组查询 11](#_Toc1016478973)

[4.2.6、嵌套集合运算 12](#_Toc1185098008)

[4.2.7、判断某个字段是否存在 13](#_Toc803491952)

[4.2.8、条件过滤 13](#_Toc1637612004)

[4.2.9、正则运算 14](#_Toc21978924)

[4.2.10、数据排序 15](#_Toc560340356)

[4.2.11、数据分页显示 15](#_Toc530395095)

[4.3、数据更新操作 16](#_Toc29797263)

[4.3.1、函数的基本使用 16](#_Toc846877274)

[4.3.2、修改器 17](#_Toc1904125448)

[4.3.3、删除数据 19](#_Toc20203223)

[5、游标（重点） 19](#_Toc209313107)

[6、索引（重点） 20](#_Toc1855903796)

[6.1、唯一索引 22](#_Toc1159190402)

[6.2、过期索引 23](#_Toc343498816)

[6.3、全文索引 23](#_Toc852690164)

[6.4、地理信息索引 25](#_Toc75583823)

[7、聚合（重点） 26](#_Toc334971714)

[7.1、取得集合的数据量 26](#_Toc525472554)

[7.2、消除重复数据 26](#_Toc1709288337)

[7.3、group操作 27](#_Toc1548877534)

[7.4、MapReduce 27](#_Toc1986474077)

[7.5、聚合框架（核心） 29](#_Toc1717852564)

[7.5.1、$group 30](#_Toc709958525)

[7.5.2、$project（投影） 31](#_Toc618499705)

[7.5.3、$match 33](#_Toc2004190386)

[7.5.4、$sort 34](#_Toc640350904)

[7.5.5、分页处理：$limit、$skip 34](#_Toc343082864)

[7.5.6、$unwind 35](#_Toc1797284721)

[7.5.7、$geoNear 35](#_Toc875525619)

[7.5.8、$out 36](#_Toc1769525684)

[8、深入操作 36](#_Toc666280046)

[8.1、固定集合 36](#_Toc2060623627)

[8.2、GridFS 37](#_Toc425533988)

[9、用户管理 38](#_Toc156408402)

[总结 41](#_Toc2082602551)

# 1、MongoDB简介（了解）

MongoDB数据库是一种NOSQL数据库，NOSQL数据库不是这几年才有的，从数据库的初期发展就已经存在有了NOSQL数据库。数据库之中支持的SQL语句是由IBM开发出来的，并且最早就应用在了Oracle数据库，但是SQL语句的使用并不麻烦，就是几个简单的单词：SELECT、FROM、WHERE、GROUP BY、HAVING、ORDER BY。但是在这个时候有人开始反感于编写SQL操作。于是有一些人就开始提出一个理论——不要去使用SQL，于是最早的NoSQL概念产生了。可是后来的发展产生了一点变化，在90年代到2010年之间，世界上最流行的数据库依然是关系型数据库，并且围绕着关系型数据库开发出了大量的程序应用。后来又随着移动技术（云计算、大数据）的发展，很多公司并不愿意去使用大型的厂商数据库—— Oracle、DB2，因为这些人已经习惯于使用MySQL数据库了。这些人发现在大数据以及云计算的环境下，数据存储受到了很大的挑战，那么后来就开始重新进行了NoSQL数据库的开发，但是经过长期的开发发现，NoSQL数据库依然不可能离开传统的关系型数据（NoSQL、Not OnlySQL）。

实际上在现实的开发之中一直存在一种神奇的问题：

·数据表→ JDBC读取→ POJO（VO、PO）→控制层转化为JSON数据→客户端。

可是这样的转换实在是太麻烦了，那么最好的做法是，直接有一个数据库就存放有要显示的JSON数据该有多好，那么就省略所有需要进行转换的过程。所以在实际的开发之中，往往除了关系型数据库之外还要提供有一个NoSQL数据库，其中NoSQL数据库负责数据的读取，因为直接保存的就是JSON（前提：MongoDB中的数据是排列好的组合数据）。

例如：现在要求显示出每个雇员的编号、姓名、职位、部门名称、部门位置、工资等级。传统的关系型数据库之中一定要存放大量的冗余数据，不合理。而有了NoSQL数据库之中，可以直接在业务层里面将数据交给NoSQL数据库保存，按照指定的结构进行存储。

在MongoDB数据库之中与Oracle数据库有如下的概念对应：

在整个行业之中，MongoDB数据库是发展最好的一个NoSQL数据库，因为它与Node.JS捆绑在一起了，也就是如果你要从事Node.JS的开发，那么一定要使用MongoDB，而Node.JS（基于JavaScript）在国内最成功的应用—— taobao。

MongoDB之所以能够更好的发展也取决于：面向集合的存储过程、模式自由（无模式）、方便的进行数据的存储扩充、支持索引、支持短暂数据保留、具备完整的数据库状态监控、基于BSON应用。

# 2、MongoDB的安装与配置（重点）

如果要想得到MongoDB数据库只需要登陆（**www.mongodb.org**）上就可以直接下载可用版本，最新的版本是0.x，但是这个版本变为了windows安装版（只是把原来的解压缩版变为了安装拷贝）。

将下载下来的MongoDB数据库直接进行安装。安装的时候需要选择好对应的操作系统，本次使用的是64位安装版。

随后将MongonDB数据库保存在E盘目录下。

此时MongoDB安装完成之后，严格来讲并不能够在windows下使用，需要为它配置path环境属性，配置目录：“E:\MongoDB\bin”。

如果要想正常启动MongoDB数据库，那么必须建立一个文件夹，这个文件夹将保存所有的数据库的信息，现在将在MongoDB文件夹之中建立一个db的目录，并且在此目录下保存所有的数据文件。

MongoDB数据库的启动需要使用mongod.exe命令完成，启动的时候可以设置端口号，也可以不设置端口号。

·不设置端口号启动MongoDB服务：

·设置端口号启动

如果日后需要通过程序访问数据库的话，那么一定要设置端口号。

当MongoDB服务启动之后，可以使用mongo命令连接数据库。

**范例：**查询所有的数据库

此时只存在有一个local的本地数据库，不过这个数据库不使用。

虽然以上的代码实现了数据库的启动与连接操作，但是从严格意义上来讲，以上的代码没有任何的用处，因为从实际的开发来讲，在MongoDB启动的时候需要设置一些相应参数：端口号、是否启用用户验证、数据文件的位置等等。

**范例：**在E:\MongoDB目录下建立一个文件“mongodb.conf”

·同时建立了一个保存日志信息的文件：log\mongodb.log；



·随后重新启动MongonDB数据库服务：



**范例：**连接数据库

此时服务器已经存在有指定的端口号了。



使用端口号启动在日后的程序开发部分是非常有用处的。

# 3、MongoDB的基础操作（重点）

在MongoDB数据库里面是存在有数据库的概念，但是没有模式（所有的信息都是按照文档保存的），保存数据的结构就是JSON结构，只不过在进行一些数据处理的时候才会使用到MongoDB自己的一些操作符。

1、 使用mldn数据库：

|  |
| --- |
| user mldn |

实际上这个时候并不会创建数据库，只有在数据库里面保存集合数据之后才能够真正创建数据库。

1. 创建一个集合——创建一个emp集合

|  |
| --- |
| db.createCollection(’emp’) |

这个时候mldn数据库才会真正的存在。

1. 但是很多的时候如果按照以上的代码形式进行会觉得你不正常，因为正常人使用MongonDB数据库集合操作的时候都是直接向里面保存一个数据。

|  |
| --- |
| db.dept.insert({deptno:10,dname:’财务部’,loc:’北京‘}) |

4、 查看所有集合：

|  |
| --- |
| show collections |

发现dept集合自动创建了。

5、 查看emp表的数据

·语法：db.集合名称.find({若干条件})

|  |
| --- |
| db.dept.find() |

 从传统的数据表来看（集合就相当于表的结构），表的结构一旦定义就必须按照其定义的要求进行内容的编写，但是MongoDB不一样，它可以自己随意扩充数据。

1. 增加不规则的数据

|  |
| --- |
| var deptData = {  'deptno':20,  'dname': '研发部',  'loc': '深圳',  'count': 20,  'avg': 8000.0  }  db.dept.insert(deptData) |
| var deptData ={  'deptno': 30,  'dname':'市场部'  }  db.dept.insert(deptData) |

此时dept集合的内容可以由用户随便去定义，完全不用考虑其它的结构，那么实际上就必须明确一点了，在MongoDB数据库之中是绝对不可能存在有查看集合结构的操作。

7、 关于ID的问题

在MongoDB集合中的每一行记录都会自动的生成一个“"\_id" :ObjectId("55949a13eecd74894d19d8dc")”数据，这个数据组成是：“时间戳 + 机器码 + PID + 计数器”，这个ID的信息是MongonDB数据自己为用户生成的。

8、 查看单独的一个文档信息

|  |
| --- |
| db.dept.findOne() |

9、 删除数据

|  |
| --- |
| db.dept.remove({\_id:ObjectId(‘\*\*\*’)}) |

10、更新数据

|  |
| --- |
| var deptData = {  'deptno': 50,  'dname': '乞讨部',  'loc': '家里蹲',  'count':60,  'avg': 9000.0  }  db.dept.update({'\_id': ObjectId('5b0e0eb53755df645754198c')},deptData) |

11、删除集合

·语法：db.集合名称.drop()

|  |
| --- |
| db.dept.drop() |

12、删除数据库（删除当前所在的数据库）

|  |
| --- |
| db.dropDatabase() |

删除数据库的是删除当前所在的数据库，必须先切换到数据库后才可以删除。

# 4、数据操作（重点）

只要是数据库那么就绝对离不开最为核心的功能：CRUD，所以在MongoDB里面对于数据的操作也是有支持的，但是需要提醒的是，除了增加之外，其它的都很麻烦。

## 4.1、数据增加

使用“db.集合.insert()”可以实现数据的增加操作。

**范例：**增加一个简单数据

|  |
| --- |
| db.infos.insert({url:’www.duodainfo.cn’}) |

**范例：**保存数组

|  |
| --- |
| db.infos.insert([{url:’www.duodainfo.cn’},{url:’www.taobao.com’}]) |

如果要想保存多个数据，那么就使用数组。

**范例：**保存10000个数据？

|  |
| --- |
| For(var x = 0; x< 1000;x++){  db.infos.insert({url:’www.duodainfo.cn-’+x})  } |

如果数据保存很多的情况下，列表时不会全部列出，它只会列出部分的内容。

## 4.2、数据查询

任何的数据库之中，数据的查询操作都是最为麻烦的，而在MongoDB数据库里面，对于查询的支持非常到位，包含有关系运算、逻辑运算、数组运算、正则运算等等。

首先对于数据的查询操作核心的语法：“db.集合名称.find({查询条件} [,{设置显示的字段}])”。

**范例：**最简单的用法就是直接使用find()函数完成查询

|  |
| --- |
| db.infos.find() |

**范例：**希望查询出url为“www.mldn.cn”的数据

|  |
| --- |
| db.infos.find({url:’www.duodainfo.cn’}) |

发现在进行数据查询的时候也是按照JSON的形式设置的相等关系。它的整个开发之中都不可能离开JSON数据。

对于设置的显示字段严格来讲就称为数据的**投影**操作，如果不需要显示的字段设置“0”，而需要显示的字段设置“1”。

**范例：**不想显示“\_id”

|  |
| --- |
| db.infos.find({url:’www.duodainfo.cn’},{\_id:0}) |
| Db.infos.find({url:’www.duodainfo.cn’},{\_id:0.url:1}) |

大部分的情况下，这种投影操作的意义不大。同时对于数据的查询也可以使用“**pretty()**”函数进行漂亮显示。

**范例：**漂亮显示

|  |
| --- |
| db.infos.find({url:’www.duodainfo.cn’}).pretty() |

数据列多的时候一定可以看出华丽的显示效果。

**范例：**查询单个数据

|  |
| --- |
| db.infos.findOne() |

查询出一条数据并可以实现格式化的输出效果

### 4.2.1、关系运算

在MongoDB里面支持的关系查询操作：

大于（$gt）、

小于（$lt）、

大于等于（$gte）、

小于等于（$lte）、

不等于（$ne）、

等于（key:value、$eq）。

**范例：**定义一个学生信息集合

|  |
| --- |
| db.students.insert({'name':'张三','sex':'男','age':19,'score': 89,'address': '海淀区'})  db.students.insert({'name':'李四','sex':'女','age':20,'score': 100,'address': '朝阳区'})  db.students.insert({'name':'王五','sex':'男','age':22,'score': 50,'address': '西城区'})  db.students.insert({'name':'赵六','sex':'女','age':21,'score': 60,'address': '东城区'})  db.students.insert({'name':'孙七','sex':'男','age':19,'score': 70,'address': '海淀区'})  db.students.insert({'name':'王八','sex':'女','age':23,'score': 90,'address': '海淀区'})  db.students.insert({'name':'刘九','sex':'女','age':35,'score': 56,'address': '朝阳区'})  db.students.insert({'name':'钱十','sex':'男','age':27,'score': 89,'address': '海淀区'}) |

**范例：**查询姓名是张三的学生信息

|  |
| --- |
| db.students.find({name:’张三’}).pretty() |

**范例：**查询性别是男的学生信息

|  |
| --- |
| db.students.find({sex:’男’}).pretty() |

**范例：**查询年龄大于19岁的学生

|  |
| --- |
| db.students.find({age:{$gt:19}}).pretty() |

**范例：**查询成绩大于等于60分的学生

|  |
| --- |
| db.students.find({score:{$gte:60}}).pretty() |

**范例：**查询姓名不是王五的信息

|  |
| --- |
| db.students.find({name:{$ne:’王五’}}).pretty() |

此时与之前最大的区别就在于，在一个JSON结构里面需要定义其它的JSON结构，并且这种风格在日后通过程序进行操作的时候依然如此。

### 4.2.2、逻辑运算

逻辑运算主要就是三种类型：与（$and）、或（$or）、非（$not、$nor）。

**范例：**查询年龄在19 ~ 20岁的学生信息

|  |
| --- |
| db.students.find({age:{$gte:19,$lte:20}}).pretty() |

在进行逻辑运算的时候“and”的连接是最容易的，因为只需要利用“,”分割若干个条件就可以了。

**范例：**查询年龄不是19岁的学生

|  |
| --- |
| db.students.find({age:{$ne:19}}).pretty() |

**范例：**查询年龄大于19岁，或者成绩大于90分的学生信息

|  |
| --- |
| db.students.find({$or:[  {age:{$gte:19}},  {score:{$gt:90}}  ]}).pretty() |

**范例：**也可以进行或的求反操作

|  |
| --- |
| db.students.find({&nor:[  {age:{$gt:19}},  {score:{$gt:90}}  ]}).pretty() |

针对于或的操作可以实现一个求反的功能。

在这几个逻辑运算之中，与的连接最简单的，而或的连接需要为数据设置数组的过滤条件。

### 4.2.3、求模

模的运算使用“$mod”来完成，语法“{**$mod : [除数,余数]**}”。

**范例：**求模

|  |
| --- |
| db.students.find({age:{$mod:[20,1]}}).pretty() |

利用求模计算可以编写一些数学的计算公式。

### 4.2.4、范围查询

只要是数据库，必须存在有“$in”（在范围之中）、“$nin”（不在范围之中）。

**范例：**查询姓名是“张三”、“李四”、“王五”的信息

|  |
| --- |
| db.students.find({name:{$in:[‘张三’,’李四’,’王五’]}}}).pretty() |

**范例：**不在范围

|  |
| --- |
| db.students.find({name:{$nin:[‘张三’,’李四’,’王五’]}}}).pretty() |

在实际的工作之中，范围的操作很重要。

### 4.2.5、数组查询

首先在mongoDB里面是支持数组保存的，一旦支持了数组保存，就需要针对于数组的数据进行匹配。

**范例：**保存一部分数组内容

|  |
| --- |
| db.students.insert({name:'谷大神-A',sex:'男',age:19,score:89,address:'海淀区',course:['语文','数学','英语','音乐','政治']})  db.students.insert({name:'谷大神-B',sex:'男',age:19,score:89,address:'海淀区',course:['语文','数学']})  db.students.insert({name:'谷大神-C',sex:'男',age:19,score:89,address:'海淀区',course:['语文','数学','英语']})  db.students.insert({name:'谷大神-D',sex:'男',age:19,score:89,address:'海淀区',course:['英语','音乐','政治']})  db.students.insert({name:'谷大神-E',sex:'男',age:19,score:89,address:'海淀区',course:['语文','政治']}) |

此时的数据包含有数组内容，而后需要针对于数组数据进行判断，可以使用几个运算符：$all、$size、$slice、$elemMatch。

**范例：**查询同时参加语文和数学课程的学生

·现在两个数组内容都需要保存，所以使用“**{"$all",[内容1,内容2,..]}**”

|  |
| --- |
| db.students.find({cource:[$all:[‘语文’,‘数学’]}).pretty() |

现在所有显示的学生信息里面包含语文和数学的内容，而如果差一个内容的不会显示。

虽然“$all”计算可以用于数组上，但是也可以用于一个数据的匹配上。

**范例：**查询学生地址是“海淀区”的信息

|  |
| --- |
| db.students.find({address: [$all:[‘海淀区’]}).pretty() |

 既然在集合里面现在保存的是数组信息，那么数组就可以利用索引操作，使用“key.index”的方式来定义索引。

**范例：**查询数组中第二个内容（index= 1，索引下标从0开始）为数学的信息

|  |
| --- |
| db.students.find({‘cource.1’: ‘数学’}).pretty() |

**范例：**要求查询出只参加两门课程的学生

·使用“$size”来进行数量的控制。

|  |
| --- |
| db.students.find({‘cource’: {$size:2}}).pretty() |

发现在进行数据查询的时候只要是内容复合条件，数组的内容就全部显示出来了，但是现在希望可以控制数组的返回的数量，那么可以使用“$slice”进行控制。

**范例：**返回年龄为19岁所有学生的信息，但是要求只显示两门参加课程

|  |
| --- |
| db.students.find({age: 19},{cource: {$slice: 2}}).pretty() |

现在只取得了前两门的信息，那么也可以设置负数表示取出后两门的信息。

|  |
| --- |
| db.students.find({age: 19},{cource: {$slice: -2}}).pretty() |

或者只是取出中间部分的信息。

|  |
| --- |
| db.students.find({age: 19},{cource: {$slice: [1,2]}}).pretty() |

在此时设置的两个数据里面第一个数据表示跳过的数据量，而第二个数据表示返回的数量。

### 4.2.6、嵌套集合运算

在MongoDB数据里面每一个集合数据可以继续保存其它的集合数据，例如：有些学生需要保存家长信息。

**范例：**增加数据

|  |
| --- |
| db.students.insert(  {name:'A',sex:'男',age:19,score:89,address:'海淀区',  course:['语文','数学','英语','音乐','政治'],  parents:[  {name:'A(father)',age:50,job:'工人'},  {name:'A(mother)',age:50,job:'职员'}]})  db.students.insert(  {name:'B',sex:'男',age:19,score:89,address:'海淀区',  course:['语文','数学'],  parents:[  {name:'B(father)',age:50,job:'处长'},  {name:'B(mother)',age:50,job:'局长'}]})  db.students.insert(  {name:'C',sex:'男',age:19,score:89,address:'海淀区',  course:['语文','数学','英语'],  parents:[  {name:'C(father)',age:50,job:'工人'},  {name:'C(mother)',age:50,job:'局长'}]}) |

此时给出的内容是嵌套的集合，而这种集合的数据的判断只能够通过“**$elemMatch**”完成。

**范例：**查询出父母有人是局长的信息

|  |
| --- |
| db.students.find({parents:{$elemMatch:{job:'局长'}}}).pretty() |

由于这种查询的时候条件比较麻烦，所以如果可能，尽量别搞这么复杂的数据结构组成。

### 4.2.7、判断某个字段是否存在

使用“$exists”可以判断某个字段是否存在，如果设置为true表示存在，如果设置为false就表示不存在。

**范例：**查询具有parents成员的数据

|  |
| --- |
| db.students.find({parents:{$exists:true}}).pretty() |

**范例：**查询不具有course成员的数据

|  |
| --- |
| db.students.find({course:{$exists:false}}).pretty() |

可以利用此类查询来进行一些不需要的数据的过滤。

### 4.2.8、条件过滤

实际上习惯于传统关系型数据库开发的我们对于数据的筛选，可能首先想到的一定是where子句，所以在MongoDB里面也提供有“$where”。

**范例：**使用where进行数据查询

|  |
| --- |
| db.students.find({course:{$exists:false}}).pretty() |
| db.students.find('this.age>20') |

对于“$where”是可以简化的，但是这类的操作是属于进行每一行的信息判断，实际上对于数据量较大的情况并不方便使用。实际上以上的代码严格来讲是属于编写一个操作的函数。

|  |  |
| --- | --- |
| db.students.find(function(){  return this.age>20  }).pretty() | db.students.find({$where:function(){  return this.age>20  }}).pretty() |

 以上只是查询了一个判断，如果要想实现多个条件的判断，那么就需要使用and连接。

|  |
| --- |
| db.students.find({$and:[  {$where:'this.age>19'},  {$where:'this.age<21'}]}).pretty() |

虽然这种形式的操作可以实现数据查询，但是最大的缺点是将在MongoDB里面保存的BSON数据变为了JavaScript的语法结构，这样的方式不方便使用数据库索引机制。

### 4.2.9、正则运算

如果要想实现模糊查询，那么必须使用正则表达式，而且正则表达式使用的是语言Perl兼容的正则表达式的形式。如果要想实现正则使用，则按照如下的定义格式：

·基础语法：{key : 正则标记}；

·完整语法：{key : {"$regex" : 正则标记 , "$options" : 选项}}。

|- 对于options主要是设置正则的信息查询的标记：

|- “i”：忽略字母大小写；

|- “m”多行查找；

|- “x”：空白字符串除了被转义的或在字符类中意外的完全被忽略；

|- “s”：匹配所有的字符（圆点、“.”），包括换行内容。

|- 需要注意的是，如果是直接使用（javascript）那么只能够使用i和m，而“x”和“s”必须使用“$regex”。

**范例：**查询以“谷”开头的姓名

|  |
| --- |
| db.students.find({name:/谷/}).pretty() |

**范例：**查询姓名有字母A

|  |
| --- |
| db.students.find({name: /a/i}).pretty() |
| db.students.find({name:{$regex:/a/i}}).pretty() |

如果要执行模糊查询的操作，严格来讲只需要编写一个关键字就够了。

正则操作之中除了可以查询出单个字段的内容之外，也可以进行数组数据的查询。

**范例：**查询数组数据

|  |
| --- |
| db.students.find({course:/语?/}).pretty() |
| db.students.find({course:/语/}).pretty() |

 MongoDB中的正则符号和之前Java正则是有一些小小差别，不建议使用以前的一些标记，正则就将其应用在模糊数据查询上。

### 4.2.10、数据排序

在MongoDB里面数据的排序操作使用“sort()”函数，在进行排序的时候可以有两个顺序：升序（1）、降序（-1）。

**范例：**数据排序

|  |
| --- |
| db.students.find().sort({score:-1}).pretty() |

但是在进行排序的过程里面有一种方式称为自然排序，按照数据保存的先后顺序排序，使用“$natural”表示。

**范例：**自然排序

|  |
| --- |
| db.students.find().sort({$natural:1}).pretty() |

在MongoDB数据库里面排序的操作相比较传统关系型数据库的设置要简单。

### 4.2.11、数据分页显示

在MongoDB里面的数据分页显示也是符合于大数据要求的操作函数：

· skip(n)：表示跨过多少数据行；

· limit(n)：取出的数据行的个数限制。

**范例：**分页显示（第一页，skip(0)、limit(5)）

|  |
| --- |
| db.students.find().skip(0).limit(5).sort({age:-1}).pretty() |

**范例：**分页显示（第二页，skip(5)、limit(5)）

|  |
| --- |
| db.students.find().skip(5).limit(5).sort({age:-1}).pretty() |

这两个分页的控制操作，就是在以后只要是存在有大数据的信息情况下都会使用它。

## 4.3、数据更新操作

对于MongoDB而言，数据的更新基本上是一件很麻烦的事情，如果在实际的工作之中，真的具有此类的操作支持，那么最好的做法，在MongoDB里面对于数据的更新操作提供了两类函数：save()、update()。

### 4.3.1、函数的基本使用

如果要修改数据最直接的使用函数就是update()函数，但是这个函数的语法要求很麻烦：

·语法：db.集合.update(更新条件 , 新的对象数据（更新操作符） , **upsert** , **multi**)；

|- upsert：如果要更新的数据不存在，则增加一条新的内容（true为增加、false为不增加）；

|- multi：表示是否只更新满足条件的第一行记录，如果设置为false，只更新第一条，如果是true全更新。

**范例：**更新存在的数据——将年龄是19岁的人的成绩都更新为100分（此时会返回多条数据）

·只更新第一条数据：

|  |
| --- |
| db.students.update({age:19},{$set:{score:100}},false,false) |

·所有满足条件的数据都更新

|  |
| --- |
| db.students.update({age:19},{$set:{score:100}},false,true) |

**范例：**更新不存在的数据

|  |
| --- |
| db.students.update({age:30},{$set:{name:’nobody’’}},true,false) |

由于没有年龄是30岁的学生信息，所以此时相当于进行了数据的创建。

那么除了update()函数之外，还提供有一个save()函数，这个函数的功能与更新不存在的内容相似。

**范例：**使用save()操作

|  |
| --- |
| db.students.save({\_id:ObjectId('5b0e4813a5c331b522c20ba5'),age:50}) |

由于此时对应的id数据存在了，所以就变为了更新操作。但是如果要保存的数据不存在（不能保存有“\_id”），那么就变为了增加操作。save()和update()均能更新能保存，但是常用upodate

### 4.3.2、修改器

对MongoDB数据库而言，数据的修改会牵扯到内容的变更、结构的变更（包含有数组），所以在进行MongoDB设计的时候就提供有一系列的修改器的应用，那么像之前使用的“$set”就是一个修改器。

1、 $inc：主要针对于一个数字字段，增加某个数字字段的数据内容；

·语法：{"$inc" : {"成员" : 内容}}

**范例：**将所有年龄为19岁的学生成绩一律减少30分，年龄增加1

|  |
| --- |
| db.students.updateMany({age:19},{$inc:{score:-30,age:1}}) |
| db.students.update({age:19},{$inc:{score:-30,age:1}},false,true) |

2、 $set：进行内容的重新设置；

·语法：{"$set" : {"成员" : "新内容"}}；

**范例：**将年龄是20岁的人的成绩修改为89

|  |
| --- |
| db.students.update({age:20},{$set:{score:89}}) |

3、 **$unset**：删除某个成员的内容；

·语法：{"$unset" : {"成员" : 1}}

**范例：**删除“张三”的年龄与成绩信息

|  |
| --- |
| db.students.update({name:’张三’},{$unset:{age:1,score:1}}) |

执行之后指定的成员内容就消失了。

4、 **$push**：相当于将内容追加到指定的成员之中（基本上是数组）；

·语法：${"$push" : {成员 : value}}

**范例：**向“李四”添加课程信息（此时张三信息下没有course信息）

|  |
| --- |
| db.students.update({name:’李四’},{$push:{course:’语文’}}) |

**范例：**向“谷大神 - E”里面的课程追加一个“美术”

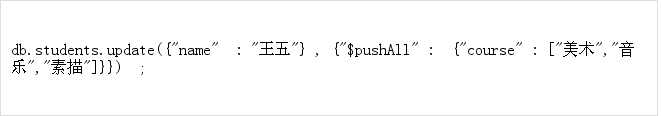
|  |
| --- |
| db.students.update({name:’谷大神 - E’},{$push:{course:’美术’}}) |

就是进行数组数据的添加操作使用的，如果没有数组则进行一个新的数组的创建，如果有则进行内容的追加。添加多个内容，要用$each

5、 ~~$pushAll：与“$push”是类似的，可以一次追加多个内容到数组里面；~~(2.4版本后废除)

~~·语法：${"$pushAll" : {成员 : 数组内容}}~~

**范例：**向“王五”的信息里面添加多个课程内容



6、 **$addToSet**：向数组里面增加一个新的内容，只有这个内容不存在的时候才会增加；

·语法：{"$addToSet" : {成员 : 内容}}

**范例：**向王五的信息增加新的内容

|  |
| --- |
| db.students.update({name:’王五’},{$addToSet:{course:’舞蹈’}}) |

此时会判断要增加的内容在数组里面是否已经存在了，如果不存在则向数组之中追加内容，如果存在了则不做任何的修改操作。

7、 **$pop**：删除数组内的数据；

·语法：{"$pop" : {成员 : 内容}}，内容如果设置为-1表示删除第一个，如果是1表示删除最后一个；

**范例：**删除王五的第一个课程

|  |
| --- |
| db.students.update({name:'王五'},{$pop:{course:-1}}) |

**范例：**删除王五的最后一个课程

|  |
| --- |
| db.students.update({name:'王五'},{$pop:{course:1}}) |

8、 $pull：从数组内删除一个指定内容的数据

·语法：{"$pull" : {成员 : 数据}}，进行数据比对的，如果是此数据则删除；

**范例：**删除王五学生的音乐课程信息

|  |
| --- |
| db.students.update({name:'王五'},{$pull:{course:’音乐’}}) |

9、 $pullAll：一次性删除多个内容；

·语法：{"$pullAll" : {成员 : [数据, 数据,...]}}

**范例：**删除“A”中的三门课程

|  |
| --- |
| db.students.update({name:’A'},{$pullAll:{course:[‘语文’,’数学’,’英语’]}}) |

10、 $rename：为成员名称重命名；

·语法：{"$rename" : {旧的成员名称 : 新的成员名称}}；

**范例：**将“张三”name成员名称修改为“姓名”

|  |
| --- |
| db.students.update({name:'张三'},{$rename:{name:'姓名'}}) |

在整个MongoDB数据库里面，提供的修改器的支持很到位。

### 4.3.3、删除数据

在MongoDB里面数据的删除实际上并不复杂，只需要使用“remove()”函数就可以了。

但是在这个函数是有两个可选项：

·删除条件：满足条件的数据被删除；

·是否只删除一个数据，如果设置为true或者是1表示只删除一个。

**范例：**清空infos集合中的内容

|  |
| --- |
| db.infos.remove({}) |

**范例：**删除所有姓名里面带有“谷”的信息，默认情况下会全部删除

|  |
| --- |
| db.students.remove({name: /谷/}) |

**范例：**删除姓名带有“高”的信息，要求只删除一个

|  |
| --- |
| db.students.remove({name: /高/},true) |

删除操作里面依然需要使用限定查询的相关操作内容。

# 5、游标（重点）

所谓的游标就是指的数据可以一行行的进行操作，非常类似于ResultSet数据处理。在MongoDB数据库里面对于游标的控制非常的简单，只需要使用find()函数就可以返回游标了。对于返回的游标如果要想进行操作，使用两个函数：

·判断是否有下一行数据：hasNext()；

·取出当前数据：next()；

|  |
| --- |
| var cursor = db.students.find()  cursor.hasNext()  cursor.next() |

以上是游标的操作形式，但是实际上不可能这么去用，因为必须利用循环才能够输出内容。

**范例：**编写具体的操作代码

|  |
| --- |
| var cursor = db.students.find()  while(cursor.hasNext()){  print(cursor.next().name)  } |

相当于每一个数据都单独拿出来进行逐行的控制。当游标数据取出来之后，实际上每行数据返回的都是一个Object型的内容，那么如果需要将数据按照json的形式出现，则可以使用printjson()函数完成。

|  |
| --- |
| var cursor = db.students.find()  while(cursor.hasNext()){  printjson(cursor.next())  } |

在所有的已知数据库，只有MongoDB的游标操作是最简单，最直观的。

# 6、索引（重点）

在任何的数据库之中，索引都是一种提升数据库检索性能的手段，这一点在MongoDB数据库之中同样是存在的，在MongoDB数据库里面依然会存在有两种的索引创建：是自动创建的，另外一种索引是手工创建的。

**范例：**重新准备一个新的简单集合

|  |
| --- |
| db.students.drop()  db.students.insert({'name':'张三','sex':'男','age':19,'score': 89,'address': '海淀区'})  db.students.insert({'name':'李四','sex':'女','age':20,'score': 100,'address': '朝阳区'})  db.students.insert({'name':'王五','sex':'男','age':22,'score': 50,'address': '西城区'})  db.students.insert({'name':'赵六','sex':'女','age':21,'score': 60,'address': '东城区'})  db.students.insert({'name':'孙七','sex':'男','age':19,'score': 70,'address': '海淀区'})  db.students.insert({'name':'王八','sex':'女','age':23,'score': 90,'address': '海淀区'})  db.students.insert({'name':'刘九','sex':'女','age':35,'score': 56,'address': '朝阳区'})  db.students.insert({'name':'钱十','sex':'男','age':27,'score': 89,'address': '海淀区'}) |

此时在students集合上并没有去设置任何的索引，那么下面通过getIndexes()函数来观察在students集合里面已经存在的索引内容。

**范例：**查询默认状态下的students集合的索引内容

|  |
| --- |
| db.students.getIndexes() |

现在发现会存在有一个“\_id”列的索引内容。但是如果要想创建自己的索引，则可以使用如下的语法：

·索引创建：db.集合名称.ensureIndex({列 : 1})

|- 设置的1表示索引将按照升序的方式进行排列，如果使用降序设置“-1”。

**范例：**创建一个索引，在age字段上设置一个降序索引；

|  |
| --- |
| db.students.ensureIndex({‘age’:-1}) |

此时并没有设置索引的名字，所以名字是自动命名的。命名规范：“字段名称\_索引的排序模式”。

**范例：**针对于当前的age字段上的索引做一个分析

|  |
| --- |
| db.students.find({'age':19}).explain() |

此时的查询使用了索引的技术，但是下面再来观察一个查询，不使用索引字段。

**范例：**针对于score字段上设置查询

|  |
| --- |
| db.students.find({score:{$gt:60}}).explain() |

此时在score字段上并没有设置索引，所以当前的索引形式就变为了全集合扫描的模式。

但是如果说，现在换一种形式，年龄和成绩一起执行判断查询？

|  |
| --- |
| db.students.find({$or:[{age:{$gt:19}},{score:{$gt:60}}]}).explain() |

这个时候虽然age字段上存在有索引，但是一个明显的问题是，由于score字段上没有索引，所以依然使用的是全表扫描操作，那么为了解决此时的问题，可以使用一个复合索引。

|  |
| --- |
| db.students.ensureIndex({age:1,score:-1},{name:'age\_1\_score\_1\_index'}) |

**范例：**默认使用索引

|  |
| --- |
| db.students.find({age:19,score:89}).explain() |

但是如果换到了条件之中：

|  |
| --- |
| db.students.find({$or:[{age:{$gt:19}},{score:{$gt:60}}]}).explain() |

 现在发现并没有使用索引，所以这个时候看能否强制使用一次索引。hint()函数为强制使用索引操作。

**范例：**强制使用索引

|  |
| --- |
| db.students.find({$or:[{age:{$gt:19}},{score:{$gt:60}}]}).hint({age:1,score:-1}).explain() |

如果正常来讲，这个代码根本就不可能调用默认的索引执行，但是我们觉得不好，所以需要使用hint()函数来强制MongoDB告诉你必须使用一次索引，由于此时在age和score两个字段上已经设置了复合索引，那么现在使用的就是默认的复合索引。

但是如果在一个集合里面设置了过多的索引，实际上会导致性能下降，那么就可以删除索引。

**范例：**删除一个索引

|  |
| --- |
| db.students.dropIndex({age:1}) |

可是如果只是一个一个删除索引也会很麻烦，所以提供有删除全部索引的操作。

**范例：**删除全部索引

|  |
| --- |
| db.students.dropIndexes() |

所谓的删除全部索引指的就是非“\_id”的索引，所有的自定义索引。

## 6.1、唯一索引

唯一索引的主要目的是用在某一个字段上，使该字段的内容不重复。

**范例：**创建唯一索引

|  |
| --- |
| db.students.ensureIndex({name:1},{unique:true}) |

在name字段上的内容绝对不允许重复。

**范例：**在students集合里面增加重复的数据

|  |
| --- |
| db.students.insert({name:'张三',sex:'男',age:21,score:100}) |

此时除了name字段上的内容之外，发现所有的数据都不一样，但是由于在name字段上设置了唯一索引，所以整个程序里面如果增加了重复内容，那么会出现以下的错误提示信息：

|  |
| --- |
| E11000 duplicate key error collection: test.students index: name\_1 dup key |

 唯一索引能够保证指定字段上的数据不重复。

## 6.2、过期索引

在一些程序站点会出现若干秒之后信息被删除的情况，例如：手机信息验证码，那么在MongoDB里面就可以轻松的实现过期索引，但是这个时间往往不怎么准确。必须要有时间，否则没法实现过期

**范例：**设置过期索引

|  |
| --- |
| db.phones.ensureIndex({time:1},{expireAfterSeconds:10}) |

设置索引在10秒后过期。

**范例：**在一个phones集合里面设置过期索引

如果要想实现过期索引，需要保存一个时间信息；

|  |
| --- |
| db.phones.insert({tel:110,code:110,time:new Date()})  db.phones.insert({tel:111,code:111,time:new Date()})  db.phones.insert({tel:112,code:112,time:new Date()})  db.phones.insert({tel:113,code:113,time:new Date()})  db.phones.insert({tel:114,code:114,time:new Date()})  db.phones.insert({tel:115,code:115,time:new Date()}) |

等到10秒以后（永远不会那么准确）所保存的数据就会消失。这样的特性在进行一些临时数据保存的时候非常有帮助，最早如果没有MongoDB这种特性，而只是使用最简单的关系型数据库开发，那么是非常麻烦的。

## 6.3、全文索引

在一些信息管理平台上经常需要进行信息模糊查询，最早的时候是利用了某个字段上实现的模糊查询，但是这个时候返回的信息并不会很准确，因为只能够查A字段或者是B字段，而在MongoDB里面实现了非常简单的全文检索。

**范例：**定义一个新的集合

|  |
| --- |
| db.news.insert({'title':'mldn mldnjava lxh gyh',content:'gyh'})  db.news.insert({'title':'mldn mldnjava lxh',content:'java'})  db.news.insert({'title':'gyh',content:'sfq'})  db.news.insert({'title':'gyh',content:'gry'})  db.news.insert({'title':'sfq',content:'gry'}) |

**范例：**设置全文检索

|  |
| --- |
| db.news.ensureIndex({title:'text',content:'text'}) |

**范例：**实现数据的模糊查询

如果要想表示出全文检索，则使用“**$text**”判断符，而要想进行数据的查询则使用“**$search**”运算符：

|- 查询指定关键字：{"$search" : "查询关键字"}；

|- 查询多个关键字（或关系）：{"$search" : "查询关键字查询关键字 ..."}；

|- 查询多个关键字（与关系）：{"$search" : "\"查询关键字\" \"查询关键字\"..."}

|- 查询多个关键字（排除某一个）：{"$search" : "查询关键字查询关键字 ... -排除关键字"}

**范例：**查询单个内容

|  |
| --- |
| db.news.find({$text:{$search:'gry'}}) |

**范例：**包含有“gry”与“sfq”的信息

|  |
| --- |
| db.news.find({$text:{$search:'gry gyh'}}) |

**范例：**同时包含有“mldn”与“lxh”的内容

|  |
| --- |
| db.news.find({$text:{$search:'\'mldn\' \'lxh\''}}) |

**范例：**包含有“mldn”但是没有“gyh”的内容

|  |
| --- |
| db.news.find({$text:{$search:'\'mldn\' \'lxh\' -gyh'}}) |

但是在进行全文检索操作的时候还可以使用相似度的打分来判断检索成果。

**范例：**为结果打分

|  |
| --- |
| db.news.find({$text:{$search:'gyh'}},{‘score’:{$meta:'textScore'}}) |
| db.news.find({$text:{$search:'gyh'}},{‘score’:{$meta:'textScore'}}).sort({'score':{$meta:'textScore'}}) |

按照打分的成绩进行排列，实际上就可以实现更加准确的信息搜索。分数越高越好

但是在这里面还有一个小问题，如果一个集合的字段太多了，那么每一个字段都分别设置全文检索麻烦点，所以简单一些，可以为所有的字段设置全文检索。

**范例：**为所有字段设置全文检索

|  |
| --- |
| db.news.ensureIndex({'$\*\*':'text'}) |

这是一种最简单的设置全文索引的方式，但是尽可能别用，一个字：慢。

## 6.4、地理信息索引

地理信息索引分为两类：2D平面索引，另外就是2DSphere球面索引。在2D索引里面基本上能够保存的信息都是坐标，而且坐标保存的就是经纬度坐标。

**范例：**定义一个商铺的集合

|  |
| --- |
| db.shop.insert({loc:[10,10]})  db.shop.insert({loc:[11,10]})  db.shop.insert({loc:[10,11]})  db.shop.insert({loc:[12,15]})  db.shop.insert({loc:[16,17]})  db.shop.insert({loc:[90,90]}) |

**范例：**为shop的集合定义2D索引

|  |
| --- |
| db.shop.ensureIndex({loc:'2d'}) |

这个时候shop集合就可以实现坐标位置的查询了，而要进行查询有两种查询方式：

·“$near”查询，查询距离某个点最近的坐标点；

·“$geoWithin”查询：查询某个形状内的点；

**范例：**假设我的现在的坐标是：[11,11]

|  |
| --- |
| db.shop.find({loc:{$near:[11,11]}}) |

但是如果执行了以上的查询，实际上会将数据集合里面的前100个点的信息都返回来了，可是太远了，设置一个距离范围—— 5个点内的。

**范例：**设置查询距离范围

|  |
| --- |
| db.shop.find({loc:{$near:[11,11],$maxDistance:5}}) |

但是需要注意一点，在2D索引里面虽然支持最大距离，但是不支持最小距离。

但是也可以设置一个查询的范围，使用“$geoWithin”查询，而可以设置的范围：

·矩形范围（$box）：{"$box":[[x1,y1],[x2,y2]]}；

·圆形范围（$center）：{"$center" : [[x1,y1],r]}；

·多边型（$polygon）：{"$polygon" :[[x1,y1],[x2,y2][x3,y3] , ...]}

**范例：**查询矩形

|  |
| --- |
| db.shop.find({loc:{$geoWithin:{$box:[[9,9],[12,12]]}}}) |

**范例：**查询圆形

|  |
| --- |
| db.shop.find({loc:{$geoWithin:{$center:[[10,10],2]}}}) |

在MongoDB数据库里面，除了一些支持的操作函数之外，还有一个重要的命令：**runCommand()**，这个函数可以执行所有的特定的MongoDB命令（最底层的函数）。

**范例：**利用runCommand()实现信息查询

|  |
| --- |
| db.shop.runCommand({'geoNear':'shop',near:[10,10],maxDistance:5,num:3}) |

这类的命令可以说是MongoDB之中最为基础的命令。

# 7、聚合（重点）

MongoDB的产生背景是在大数据环境，所谓的大数据实际上也就是进行的信息收集汇总。那么就必须存在有信息的统计操作，而这样的统计操作就称为聚合（直白：分组统计就是一种聚合操作）。

## 7.1、取得集合的数据量

对于集合的数据量而言，在MongoDB里面直接使用count()函数就可以完成了。

**范例：**统计students表中的数据量

|  |
| --- |
| db.students.count() |

**范例：**模糊查询

|  |
| --- |
| db.students.find({name:/张/i}) |

在进行信息查询的时候，不设置条件永远要比设置条件的查询快很多，也就是说在之前的代码编写里面不管是查询全部还是模糊查询，实际上最终都使用的模糊查询一种（没有设置关键字）。

## 7.2、消除重复数据

在学习SQL的时候对于重复的数据可以使用“DISTINCT”，那么这一操作在MongoDB之中依然支持。

**范例：**查询所有name的信息

·本次的操作没有直接的函数支持，只能够利用runCommand()函数。

|  |
| --- |
| db.students.runCommand({**'distinct'**:'students','key':'name'}) |

此时实现了对于name数据的重复值的筛选。

## 7.3、group操作

使用“group”操作可以实现数据的分组操作，在MongoDB里面会将集合依据指定的key的不同进行分组操作，并且每一个组都会产生一个处理的文档结果。

**范例：**查询所有年龄大于等于19岁的学生信息，并且按照年龄分组（runCommand是底层的方法，了解就行）

|  |
| --- |
| db.runCommand({'group':{  'ns':'students',  'key':{age:true},  'initial': {count: 0},  'condition': {age:{$gte:19}},  '$reduce':function(doc,prev){  prev.count++  }  }}) |
| db.students.group({  key:{sex:1},  cond:{age:{$gte:19}},  reduce:function(cur,result){result.count++},  initial:{count:0}  }) |

以上的操作代码里面实现的就属于一种MapReduce，但是这样只是根据传统的数据库的设计思路，实现了一个所谓的分组操作，但是这个分组操作的最终结果是有限的。

## 7.4、MapReduce

MapReduce是整个大数据的精髓所在（实际中别用），所谓的MapReduce就是分为两步处理数据：

· Map：将数据分别取出；

· Reduce：负责数据的最后的处理。

可是要想在MongoDB里面实现MapReduce处理，那么复杂度是相当高的。

**范例：**建立一组雇员数据

|  |
| --- |
| db.emps.insert({name:'zhangsan',age:30,sex:'man',job:'ClERK',salary:1000})  db.emps.insert({name:'lisi',age:28,sex:'woman',job:'ClERK',salary:5000})  db.emps.insert({name:'wangwu',age:26,sex:'man',job:'MANAGER',salary:6000})  db.emps.insert({name:'zhaoliu',age:32,sex:'woman',job:'MANAGER',salary:7000})  db.emps.insert({name:'sunqi',age:31,sex:'man',job:'ClERK',salary:2000})  db.emps.insert({name:'wangba',age:35,sex:'woman',job:'PRESIDENT',salary:9000}) |

使用MapReduce操作最终会将处理结果保存在一个单独的集合里面，而最终的处理效果如下。

**范例：**按照职位分组，取得每个职位的人名

·编写分组的定义：

|  |
| --- |
| var jobMapFun = function(){  emit(this.job,this.name)  } |
| 第一组：{key:’CLERK’,values:[姓名,姓名...]} |

·第二步：编写reduce操作；

|  |
| --- |
| var jobReduceFun = function(key,values){  return {job:key,names:values}  } |

·第三步：针对于MapReduce处理完成的数据实际上也可以执行一个最后处理。

|  |
| --- |
| var jobFinalizeFun = function(key,values) {  if(key === 'PRESIDENT'){  return {job:key,names:values,info:'the boss'}  }  return {job:key,names:values}  } |

·进行操作的整合：

|  |
| --- |
| db.runCommand({  mapreduce:'emps',  map:jobMapFun,  reduce:jobReduceFun,  out:’t\_job\_emp’,  finalize:jobFinalizeFun  }) |

现在执行之后，所有的处理结果都保存在了“t\_job\_emp”集合里面。

|  |
| --- |
| db.t\_job\_emp.find().pretty() |

**范例：**统计出各性别的人数、平均工资、最低工资、雇员姓名

|  |
| --- |
| var sexMapFun = function(){  // 定义好分组的条件，以及每个集合要取出的内容  emit(this.sex,{ccount:1,csal:this.salary,cmax:this.salary,cmin:this.salary,cname:this.name})  }  var sexReduceFun = function(key,values){  var total = 0 //统计  var sum = 0 // 计算总工资  var max = values[0] //假设第一个数据是最高工资  var min = values[0] //假设第一个数据是最低工资  var names = new Array() //定义数组内容  for(var x in values) { // 表示循环取出里面的内容  total += values[x].ccount // 人数增加  sum += values[x].csal // 循环取出所有的工资，并且累加  if(max < values[x]){  max = values[x]  }  if(min > values[x]){  min = values[x]  }  names[x] = values[x].cname // 保存姓名  }  var avg = (sum/total).toFixed(2)  // 返回数据的处理结果  return {count:total,avg:avg,sum:sum,max:max,min:min,names:names}  }  db.runCommand({  mapreduce:'emps',  map:sexMapFun,  reduce:sexReduceFun,  out:'t\_sex\_emp'  }) |

虽然大数据的时代提供有最强悍的MapReduce支持，但是从现实的开发来讲，真的不可能使用起来。

## 7.5、聚合框架（核心）

MapReduce功能强大，但是它的复杂度和功能一样强大，那么很多时候我们需要MapReduce的功能，可是又不想把代码写的太复杂，所以从Mongo 2.x版本之后开始引入了聚合框架并且提供了聚合函数：**aggregate()**。

### 7.5.1、$group

group主要进行分组的数据操作。

**范例：**实现聚合查询的功能——求出每个职位的雇员人数

|  |
| --- |
| db.emps.aggregate([{$group:{\_id:'$job',job\_count:{$sum:1}}}]) |

这样的操作更加符合于传统的groupby子句的操作使用。

**范例：**求出每个职位的总工资

|  |
| --- |
| db.emps.aggregate([{$group:{\_id:'$job',job\_sql:{$sum:'$salary'}}}]) |

在整个聚合框架里面如果要引用每行的数据使用：“**$字段名称**”。

**范例：**计算出每个职位的平均工资

|  |
| --- |
| db.emps.aggregate([{$group:{\_id:'$job',job\_avg:{$avg:'$salary'}}}]) |

**范例：**求出最高与最低工资

|  |
| --- |
| db.emps.aggregate([{$group:{  \_id:'$job',  job\_max:{$min:'$salary'},  job\_min:{$min:'$salary'}}}]) |

以上的几个与SQL类似的操作计算就成功的实现了。

**范例：**计算出每个职位的工资数据（数组显示）

|  |
| --- |
| db.emps.aggregate([{$group:  {\_id:'$job',  job\_sal:{$push:'$salary'}}}]) |

**范例：**求出每个职位的人员

|  |
| --- |
| db.emps.aggregate([{$group:  {\_id:'$job',  job\_sal:{$push:'$name'}}}]) |

使用“$push”的确可以将数据变为数组进行保存，但是有一个问题出现了，重复的内容也会进行保存，那么在MongoDB里面提供有取消重复的设置。

**范例：**取消重复的数据

|  |
| --- |
| db.emps.aggregate([{$group:  {\_id:'$job',  job\_sal:{$addToSet:'$name'}}}]) |

**范例：**保存第一个内容

|  |
| --- |
| db.emps.aggregate([{$group:  {\_id:'$job',  job\_sal:{$first:'$name'}}}]) |

**范例：**保存最后一个内容

|  |
| --- |
| db.emps.aggregate([{$group:  {\_id:'$job',  job\_sal:{$last:'$name'}}}]) |

虽然可以方便的实现分组处理，但是有一点需要注意，所有的分组数据是无序的，并且都是在内存之中完成的，所以不可能支持大数据量。

### 7.5.2、$project（投影）

可以利用“$project”来控制数据列的显示规则，那么可以执行的规则如下：

|- 普通列（{成员 : 1 | true}）：表示要显示的内容；

|- “\_id”列（{"\_id" : 0 | false}）：表示“\_id”列是否显示；

|- 条件过滤列（{成员 : 表达式}）：满足表达式之后的数据可以进行显示。

**范例：**只显示name、job列，不显示“\_id”列

|  |
| --- |
| db.emps.aggregate([  {$project:{\_id:0,name:1}}  ]) |

此时只有设置进去的列才可以被显示出来，而其它的列不能够被显示出来。实际上这就属于数据库的投影机制。

实际上在进行数据投影的过程里面也支持四则运算：**加法（“$add”）、减法（“$subtract”）、乘法（“$multiply”）、除法（“$divide”）、求模（$mod）**。

**范例：**观察四则运算

|  |
| --- |
| db.emps.aggregate([  {$project:{  \_id:0,  name:1,  job:1,  salary:1}}  ]) |
| db.emps.aggregate([  {$project:{  \_id:0,  name:1,  job:1,  salary:{'年薪':{$multiply:['$salary',12]}}}}  ]) |

除了四则运算之外也支持如下的各种运算符：

·关系运算：大小比较（“$cmp”）、等于（“$eq”）、大于（“$gt”）、大于等于（“$gte”）、小于（“$lt”）、小于等于（“$lte”）、不等于（“$ne”）、判断NULL（“$ifNull”），这些返回的结果都是布尔型数据；

·逻辑运算：与（“$and”）、或（“$or”）、非（“$not”）；

·字符串操作：连接（“$concat”）、截取（“$substr”）、转小写（“$toLower”）、转大写（“$toUpper”）、不区分大小写比较（“$strcasecmp”）。

**范例：**找出所有工资大于等于2000的雇员姓名、年龄、工资

|  |
| --- |
| db.emps.aggregate([  {$project:{  \_id:0,  name:1,  job:1,  'sal':'$salary',  salary:{$gte:['$salary',2000]}}}  ]) |

**范例：**查询职位是manager的信息

· MongoDB中的数据是区分大小写的；

|  |
| --- |
| db.emps.aggregate([  {$project:{  \_id:0,  name:1,  '职位':'$job',  job:{$eq:['$job',’manager’]}}}  ]) |
| db.emps.aggregate([  {$project:{  \_id:0,  name:1,  '职位':'$job',  job:{$eq:['$job',{$toUpper:’manager’}]}}}  ]) |
| db.emps.aggregate([  {$project:{  \_id:0,  name:1,  '职位':'$job',  job:{$strcasecmp:['$job',’manager’]}}}  ]) |

**范例：**使用字符串截取

|  |
| --- |
| db.emps.aggregate([  {$project:{  \_id:0,  name:1,  '职位':'$job',  job:{'前三位':{$substr:['$job',0,3]}}}}  ]) |

利用“$project”实现的投影操作功能相当强大，所有可以出现的操作几乎都能够使用。

### 7.5.3、$match

“$match”是一个滤波操作，就是进行WHERE的过滤。

**范例：**查询工资在2000 ~5000的雇员

|  |
| --- |
| db.emps.aggregate([  {$match:{salary:{$gte:2000,$lte:5000}}}  ]) |

这个时候实现的代码严格来讲只是相当于“SELECT \* FROM 表 WHERE 条件”，属于所有的内容都进行了查询。

**范例：**控制投影操作

|  |
| --- |
| db.emps.aggregate([  {$match:{salary:{$gte:2000,$lte:5000}}},  {$project:{\_id:0,name:1,salary:1}}  ]) |

此时相当于实现了“SELECT字段 FROM ... WHERE”语句结构。

**范例：**继续分组

|  |
| --- |
| db.emps.aggregate([  {$match:{salary:{$gte:1000,$lte:10000}}},  {$project:{\_id:0,name:1,job:1,salary:1}},  {$group:{\_id:’$job’,count:{$sum:1},avg:{$avg:’$salary’}}}  ]) |

通过一系列的演示可以总结一点：

·“$project”：相当于SELECT子句；

·“$match”：相当于WHERE子句；

·“$group”：相当于GROUP BY子句。

### 7.5.4、$sort

使用“$sort”可以实现排序，设置1表示升序，设置-1表示降序。

**范例：**实现排序

|  |
| --- |
| db.emps.aggregate([{$sort:{age:1,salary:-1}}]) |

**范例：**将所有的操作一起使用

|  |
| --- |
| db.emps.aggregate([  {$match:{salary:{$gte:1000,$lte:10000}}},  {$project:{\_id:0,name:1,job:1,salary:1}},  {$group:{\_id:’$job’,count:{$sum:1},avg:{$avg:’$salary’}}},  {$sort:{count:-1}}  ]) |

此时实现了降序排序，使用的是生成定义的别名。

### 7.5.5、分页处理：$limit、$skip

“$limit”：负责数据的取出个数；

“$skip”：数据的跨过个数。

**范例：**使用“$limit”设置取出的个数

|  |
| --- |
| db.emps.aggregate([  {$project:{\_id:0,name:1,salary:1,job:1}},  {$limit:5}  ]) |

**范例：**跨过三行数据

|  |
| --- |
| db.emps.aggregate([  {$project:{\_id:0,name:1,salary:1,job:1}},  {$skip:3},  {$limit:5}  ]) |

**范例：**综合应用

|  |
| --- |
| db.emps.aggregate([  {$match:{salary:{$gte:1000,$lte:10000}}},  {$project:{\_id:0,name:1,job:1,salary:1}},  {$group:{\_id:’$job’,count:{$sum:1},avg:{$avg:’$salary’}}},  {$sort:{count:-1}},  {$skip:1},  {$limit:1}  ]) |

 只能够说现在的查询可以在实际的开发之中使用了。

### 7.5.6、$unwind

在查询数据的时候经常会返回数组信息，但是数组并不方便信息的浏览，所以提供有“$unwind”可以将数组数据变为独立的字符串内容。

**范例：**添加一些信息

|  |
| --- |
| db.depts.insert({title:'技术部',bus:['研发','生产','培训']})  db.depts.insert({title:'财务',bus:['工资','税收']}) |

**范例：**将信息进行转化

|  |
| --- |
| db.depts.aggregate([  {$project:{\_id:0}},  {$unwind:'$bus'}  ]) |

此时相当于将数组中的数据变为了单行的数据。

### 7.5.7、$geoNear

使用“$geoNear”可以得到附近的坐标点。（了解）

**范例：**准备测试数据

|  |
| --- |
| db.shop.drop()  db.shop.insert({loc:[10,10]})  db.shop.insert({loc:[11,10]})  db.shop.insert({loc:[10,11]})  db.shop.insert({loc:[12,15]})  db.shop.insert({loc:[16,17]})  db.shop.insert({loc:[60,60]})  db.shop.insert({loc:[110,100]}) |
| db.shops.ensureIndex(loc:’2d’) |

**范例：**设置查询

|  |
| --- |
| db.shop.aggregate([  {$geoNear:{  'near':[11,12],  'distanceField':’loc’,  'maxDistance':1,  'num':2,  'spherical':true}}  ]) |

地理信息的检索必须存在有索引的支持。

### 7.5.8、$out

“$out”：利用此操作可以将查询结果输出到指定的集合里面。

**范例：**将投影的结果输出到集合里

|  |
| --- |
| db.emps.aggregate([  {$project:{\_id:0}},  {$out:'emp\_infos'}  ]) |

这类的操作就相当于实现了最早的数据表的复制操作。

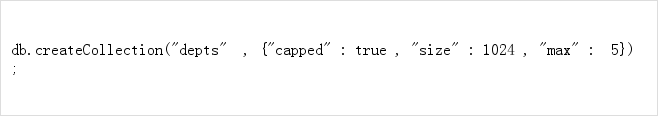
# 8、深入操作

## 8.1、固定集合

所谓的固定集合指的是规定集合大小，如果要保存的内容已经超过了集合的长度，那么会采用LRU的算法（最近最少使用原则）将最早的数据移出，从而保存新的数据。

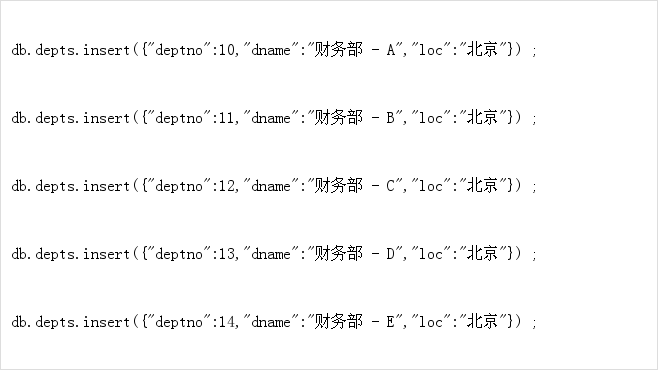
默认情况下一个集合可以使用createCollection()函数创建，或者使用增加数据后自动创建，但是如果要想使用固定集合，就必须明确的创建一个空集合。

**范例：**创建一个空集合（固定集合）

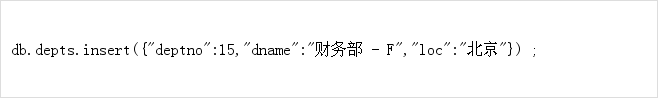


其中“capped :true”表示为一个固定集合，而“size:1024”指的是集合所占的空间容量（字节）、“max : 5”最多只能够有五条记录。

**范例：**向集合里面保存5条数据



此时已经达到了集合的上限，那么继续保存新的内容。



此时最早保留的数据已经消失了。实际上这种操作跟缓存机制是非常相似的，例如，在baidu上经常会出现一些搜索的关键词（热门），这些词都是会被不断替换的。

## 8.2、GridFS

在MongoDB里面支持大数据的存储（例如：图片、音乐、各种二进制数据），但是这个做法需要用户自己进行处理了，使用“mogofiles”命令完成。

1、 利用命令行进入到所在的路径下；

2、 将文件保存到文件库之中；



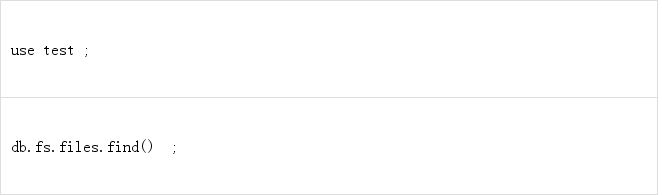
此时会向数据库里面写入要保存的二进制数据；

3、 查看保存的文件；



4、 在MongoDB里面有一个fs系统集合，这个集合默认保存在了test数据库下；

**范例：**查看保存的信息



5、 删除文件



等于在mongoDB里面支持二进制数据的保存，但是存在的意义不大。

# 9、用户管理

在MongoDB里面默认情况下只要是进行连接都可以不使用用户名与密码，因为要想让其起作用，则必须具备以下两个条件：

·条件一：服务器启动的时候打开授权认证；

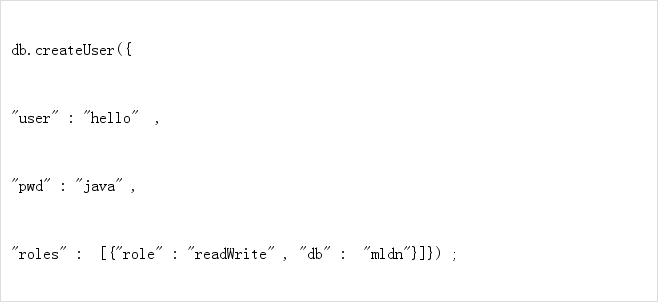
·条件二：需要配置用户名和密码。

但是需要明确的是，如果要想配置用户名和密码一定是针对于一个数据库的，例如现在要创建的是mldn数据库的用户，那么就必须首先切换到mldn数据库上。



**范例：**执行用户的创建（hello、java）

·任何的用户都必须具备有一个自己的操作角色，对于角色最基础的角色：read、readWrite。



那么现在就表示已经成功的创建了hello用户。那么如果要想让此用户名起作用，则必须以授权的方式打开MongoDB的服务，修改MongoDB的启动文件。

**范例：**新的启动配置文件



此时增加了一个验证的启动模式，发现依然可以在不输入用户名和密码的前提下进行登录，并且也可以直接进行数据库的切换操作。但是在使用数据库集合的时候出现了错误提示。

**范例：**登录数据库的时候使用用户名与密码



此时就表示成功的实现了用户的登录操作。

**范例：**修改密码



如果要修改密码，那么就请关闭授权登录。

# 总结

MongoDB实际上是作为一个附属数据库存在，只有Node.JS把它作为正室，但是除了Node.JS之外，Mongo就是一个不能够单独使用的数据库，都需要与传统的关系型数据库匹配在一起使用。