MongoDB

--- 老孙

1、简介

1.1 行业应用场景

- 游戏场景，使用 MongoDB 存储游戏用户信息，用户的装备、积分等直接以内嵌文档的形式存储，方便查询、更新
- 物流场景，使用 MongoDB 存储订单信息，订单状态在运送过程中会不断更新，以 MongoDB 内嵌数组的形式来存储，一次查询就能将订单所有的变更读取出来。
- 社交场景，使用 MongoDB 存储存储用户信息，以及用户发表的朋友圈信息，通过地理位置索引实现附近的人、地点等功能
- 物联网场景，使用 MongoDB 存储所有接入的智能设备信息，以及设备汇报的日志信息，并对这些信息进行多维度的分析
- 视频直播，使用 MongoDB 存储用户信息、礼物信息等

1.2 性能优越

- 在使用场合下，千万级别的文档对象，近10G的数据，对有索引的ID的查询不会比mysql慢
- 而对非索引字段的查询，则是全面胜出。

- mysql实际无法胜任大数据量下任意字段的查询，而mongodb的查询性能实在牛x。
- MySQL、MongoDB简单的性能测试，网上看到一组测试数据，分享给大家。
    - 测试环境：Windows 10、内存8G、CPU i5 3.30GHZ。均无索引
    - 测试语言：Python
    - 链接工具：pymysql、pymongo
    - MySQL && Mongo 测试数据统计

	提交次数	单次提交个数	MySQL运行时间（s）	Mongo运行时间（s）
1	1000	10000	3912	1622.02
2	100	100	30	1.61
3	100	100	5.77	1.60
4	10	25	2.35	1.56
5	10	25	7.42	1.60
6	10000	1	298.07	5.29
7	10000	1	496.18	5.29

1.3 MongoDB的优点

1.3.1 弱一致性（最终一致），更能保证用户的访问速度

- 当我们说外部一致性时，是针对分布式系统所说的CAP理论中的一致性，简单来说就是如何使得多台机器的副本保持一致，实际上Mongodb只能做到最终一致性，总会有“不一致时间窗口”，这是由于Mongodb在更新操作的时候，需要将同样的更新复制到副本节点当中，而这段时间无法保证reader读到的一定是最新数据，只能保证返回目前大多数节点的所持有的数据，

而不一定是最新的数据（比如，只有primary节点更新完成，其它所有secondary节点都还没有更新完成）。

1.3.2 文档结构的存储方式，能够更便捷的获取数据

- 对于一个层级式的数据结构来说，如果要将这样的数据使用扁平式的，表状的结构来保存数据，这无论是在查询还是获取数据时都十分困难。
- 举例1：就拿一个“字典项”来说，虽然并不十分复杂，但还是会关系到“定义”、“词性”、“发音”或是“引用”等内容。大部分工程师会将这种模型使用关系型数据库 中的主键和外键表现出来，但把它看作一个“文档”而不是“一系列有关系的表”岂不更好？使用“dictionary.definition.partOfSpeech='noun'”来查询也比表之间一系列复杂（往往代价也很高）的连接查询方便 且快速。
- 举例2：在一个关系型数据库中，一篇博客（包含文章内容、评论、评论的投票）会被打散在多张数据表中。在MongoDB中，能用一个文档来表示一篇博客， 评论与投票作为文档数组，放在正文主文档中。这样数据更易于管理，消除了传统关系型数据库中影响性能和水平扩展性的“JOIN”操作。

```
db.blogposts.save({
    title : "My First Post",
    author: {name : "Jane", id :1},
    comments : [{ by: "Abe", text: "First" },{ by
: "Ada", text : "Good post" }]
});

db.blogposts.find( { "author.name" : "Jane" } );

db.blogposts.findOne({
    title : "My First Post",
    "author.name": "Jane",
```

```
    comments : [{ by: "Abe", text: "First" },{ by
: "Ada", text : "Good post" } ]
});

db.blogposts.find( { "comments.by" : "Ada" } );
```

- 举例3：MongoDB是一个面向文档的数据库，目前由10gen开发并维护，它的功能丰富，齐全，完全可以替代MySQL。在使用MongoDB做产品原型的过程中，我们总结了MonogDB的一些亮点：
  - 使用JSON风格语法，易于掌握理解：MongoDB使用JSON的变种BSON作为内部存储的格式和语法。针对MongoDB的操作都使用JSON风格语法，客户端提交或接收的数据都使用JSON形式来展现。相对于SQL来说，更加直观，容易理解和掌握。
  - Schema-less，支持嵌入子文档：MongoDB是一个Schema-free的文档数据库。
    - 一个数据库可以有多个Collection，每 个Collection是Documents的集合。Collection和Document和传统数据库的Table和Row并不对等。无需事先定Collection，随时可以创建。
  - Collection中可以包含具有不同schema的文档记录。 这意味着，你上一条记录中的文档有3个属性，而下一条记录的文档可以有10个属 性，属性的类型既可以是基本的数据类型（如数字、字符串、日期等），也可以是数组或者散列，甚至还可以是一个子文档（embed document）。这 样，可以实现逆规范化（denormalizing）的数据模型，提高查询的速度。

1.3.3 内置GridFS，支持大容量的存储

- GridFS是一个出色的分布式文件系统，可以支持海量的数据存储。
- 内置了GridFS了MongoDB，能够满足对大数据集的快速范围查询。

1.3.4 内置Sharding

- 提供基于Range的Auto Sharding机制：一个collection可按照记录的范围，分成若干个段，切分到不同的Shard上。
- Shards可以和复制结合，配合Replica sets能够实现Sharding+fail-over，不同的Shard之间可以负载均衡。
- 查询是对 客户端是透明的。客户端执行查询，统计，MapReduce等操作，这些会被MongoDB自动路由到后端的数据节点。这让我们关注于自己的业务，适当的 时候可以无痛的升级。MongoDB的Sharding设计能力最大可支持约20 petabytes(PB)，足以支撑一般应用。
- 保证MongoDB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，避免了“sharding”操作的复杂性和成本。

1.3.5 第三方支持丰富

- (这是与其他的NoSQL相比，MongoDB也具有的优势)
- 现在网络上的很多NoSQL开源数据库完全属于社区型的，没有官方支持，给使用者带来了很大的风险。
    - 而开源文档数据库MongoDB背后有商业公司10gen为其提供供商业培训和支持。
    - 而且MongoDB社区非常活跃，很多开发框架都迅速提供了对MongDB的支持。

- 不少知名大公司和网站也在生产环境中使用MongoDB，越来越多的创新型企业转而使用MongoDB作为和Django，RoR来搭配的技术方案。

1.4 MongoDB的缺点

1.4.1 事务支持不友好

- 所以事务要求严格的系统（如果银行系统）肯定不能用它。(这点和优点①是对应的)

1.4.2 占用空间过大

- 关于其原因，在官方的FAQ中，提到有如下几个方面：
  - 1、空间的预分配：为避免形成过多的硬盘碎片，mongodb每次空间不足时都会申请生成一大块的硬盘空间，而且申请的量从64M、128M、256M那 样的指数递增，直到2G为单个文件的最大体积。随着数据量的增加，你可以在其数据目录里看到这些整块生成容量不断递增的文件。
  - 2、字段名所占用的空间：为了保持每个记录内的结构信息用于查询，mongodb需要把每个字段的key-value都以BSON的形式存储，如果 value域相对于key域并不大，比如存放数值型的数据，则数据的overhead是最大的。一种减少空间占用的方法是把字段名尽量取短一些，这样占用 空间就小了，但这就要求在易读性与空间占用上作为权衡了。我曾想过把字段名做个index，每个字段名用一个字节表示，这样就不用担心字段名取多长 了。但作者的担忧也不无道理，这种索引方式需要每次查询得到结果后把索引值跟原值作一

个替换，再发送到客户端，这个替换也是挺耗费时间的。现在的实现算是拿空间来换取时间吧。

- 3、删除记录不释放空间：这很容易理解，为避免记录删除后的数据的大规模挪动，原记录空间不删除，只标记“已删除”即可，以后还可以重复利用。
- 4、可以定期运行db.repairDatabase()来整理记录，但这个过程会比较缓慢

2、安装与配置

2.1 安装

- 官网： www.mongodb.org
- 下载社区版 MongoDB 4.1.3

```
[root@A opt]# wget
https://fastdl.MongoDB.org/linux/MongoDB-linux-
x86_64-4.1.3.tgz
```

- 将压缩包解压即可

```
[root@A opt]# tar -zxvf mongodb-linux-x86_64-
4.1.3.tgz
```

- 启动

```
[root@A mongodb-linux-x86_64-4.1.3]# ./bin/mongod
此方式报错，因为mongo启动需要加载默认的数据目录和日志文件，当前并没有为此创建
我们把各种参数都放在配置文件中，并在启动前创建好各种目录
[root@A opt]# mkdir -p /data/mongo/
```

- 配置文件样例

在mongo的根目录下创建配置文件mongo.conf

```
[root@A mongodb-linux-x86_64-4.1.3]# vim
mongo.conf
```

```
dbpath=/data/mongo/
port=27017
bind_ip=0.0.0.0
fork=true
logpath=/data/mongo/MongoDB.log
logappend=true
auth=false
```

参数	说明
dbpath	数据库目录，默认/data/db
port	监听的端口，默认27017
bind_ip	监听IP地址，默认全部可以访问
fork	是否已后台启动的方式登陆
logpath	日志路径
logappend	是否追加日志
auth	是开启用户密码登陆
config	指定配置文件

- 指定配置文件方式的启动

```
[root@A /]# ./bin/mongod -f mongo.conf
```

- 使用mongo shell进入mongo

```
[root@A /]# ./bin/mongo
```

- 指定主机和端口的方式进入mongo：指定ip和端口

`./bin/mongo --host=主机IP --port=端口`

```
[root@A mongodb-linux-x86_64-4.1.3]# ./bin/mongo --host 192.168.204.141:27017
```

- ip默认，指定端口

```
[root@A mongodb-linux-x86_64-4.1.3]# ./bin/mongo --port 27017
```

2.2 Mongodb GUI工具

2.2.1 MongoDB Compass Community

- MongoDB Compass Community由MongoDB开发人员开发，这意味着更高的可靠性和兼容性。
- 它为MongoDB提供GUI mongodb工具，以探索数据库交互,具有完整的CRUD功能并提供可视方案。
- 借助内置模式可视化，用户可以分析文档并显示丰富的结构。
- 为了监控服务器的负载，它提供了数据库操作的实时统计信息。
- 就像MongoDB一样，Compass也有两个版本，一个是Enterprise（付费），社区可以免费使用。适用于Linux，Mac或Windows。

2.2.2 NoSQLBooster（mongobooster）

- NoSQLBooster是MongoDB CLI界面中非常流行的GUI工具。它正式名称为MongoBooster。
- NoSQLBooster是一个跨平台，它带有一堆mongodb工具来管理数据库和监控服务器。
- 这个Mongodb工具包括服务器监控工具，Visual Explain Plan，查询构建器，SQL查询，ES2017语法支持等等.....

- 它有免费，个人和商业版本，当然，免费版本有一些功能限制。NoSQLBooster也可用于Windows，MacOS和Linux。

3、数据库基本操作

- 查看数据库

```
show dbs;
```

- 切换数据库 如果没有对应的数据库则创建（新建的库默认不显示，必须要插入一个文档后才会显示）

```
use 数据库名;

db.laosun.insert({"did":1,"dname":"开发部","loc":"A区1座"});
```

- 创建集合

```
db.createCollection("集合名")
```

- 查看集合（下面两条命令结果一样）

```
show tables;
show collections;
```

- 删除集合

```
db.集合名.drop();
```

- 删除数据库

```
db.dropDatabase();
```

3.1 添加

- 插入一条数据,自动创建dept集合(集合中的数据格式,就是json格式)

```
db.dept.insert(
    {
       "did":1,
       "dname":"开发部",
       "loc":"A区1座"
    });
```

- 查看集合中的所有数据

```
db.dept.find();
```

- 插入数据

```
var dataVar = {
    "did":2,
    "dname":"市场部",
    "loc":"深圳",
    "count":20
}
db.dept.insert(dataVar);
```

- 数据随便定义,所以没法查看集合结构的方法,没有固定结构

```
var dataVar = {
    "did":2,
    "dname":"市场部"
}
db.dept.insert(dataVar);
```

- MongoDB每一行记录都有_id,这个_id是由 **时间戳** + **机器码** + **进程的pid** + **计数器**

- 这个_id值是**绝对不会重复**的.这个ID信息是MongoDB为用户服务的.不能被改变
- 插入数组数据

```
db.dept.insert([
    {"did":3,"dname":"人事部"},
    {"did":4,"dname":"保洁部"}
]);

//插入100条数据
for(var i = 1; i<100;i++){
    db.dept.insert({"did":i,"dname":"测试"+i+"部"});
}
```

MongoDB是部分显示数据.

3.2 删除

- remove( ) 有两个可选项
  - 满足条件的删除
  - 是否只删除一条,true或1.就是删除一条
- 删除指定的某个数据

```
db.dept.remove({
    "_id" : ObjectId("6114d0d2430f6b4dcc876c41")
});
```

- 删除名称中含有"测试"的数据,默认会将匹配的数据全部删除

```
db.dept.remove({
    "dname" : /测试/
});
```

- 删除名称中含有"测试"的数据,要求只删除一条

```
db.dept.remove({
    "dname" : /测试/
},true);
```

- 清空集合中的全部数据

```
db.dept.remove({});
```

- 删除集合

```
db.dept.drop();
```

- 查看当前所在的数据库

```
db;
```

- 删除当前所在的数据库

```
db.dropDatabase();
```

3.3 修改

- 关系型数据库的数据更新,对于Mongo而言,最好的办法不是修改,而是删掉,重新添加.

3.3.1 函数

- update语法:
  - db.集合.update(**更新条件** , **新对象数据** , upsert , multi);
  - upsert:函数的动作,修改还是添加
    - false:如果没有匹配的数据,啥也不干
    - true:如果没有匹配的数据,执行增添操作
  - multi:如果条件匹配多个结果

    - false:匹配项只修改一行
    - true:匹配项全部修改
- 修改数据根据_Id

```
// 插入一条
var dataVar = {
    "id":1,
    "name":"吕布",
    "age":29,
    "address":"灵界",
};
db.student.insert(dataVar);

// 修改
var dataNew = {
    "id":1,
    "name":"吕小布",
    "age":29,
    "address":"灵幻界",
};
db.student.update({
    "_id": ObjectId("6114df1d430f6b4dcc876c42"),
},dataNew);
```

- 修改年龄27岁的人的地址为"大东北"

```
db.student.update(
    {"age":27},
    {"$set":{"address":"大东北"}},
    false,
    false
);
```

- 修改99岁的名字为"上仙"，如果没有99岁的人，则添加本条数据

```
db.student.update(
    {"age":99} ,
    {"$set":{"name":"上仙"}},
    true,
    false
);
```

- 只修改 27岁的**第一条** 地址为“东北三省”

```
db.student.update(
    {"age":27},
    {"$set":{"address":"东北三省"}},
    true,
    false
);
```

- 修改**全部** 27岁的地址为“东北三省”

```
db.student.update(
    {"age":27},
    {"$set":{"address":"东北三省"}},
    true,
    true
);
```

3.3.2 修改器

- 初始化一条测试数据：

```
db.student.insert({
    "name":"地三鲜",
    "age":88,
    "score":99
});
```

1. **$inc**主要针对数字字段,增加某个数字字段的值
   - 语法 : {"$inc":{"成员":增减量}
     // 修改年龄88岁的人,年龄+2岁,评分-10分

```
db.student.update(
    {"age":88},
    {
        "$inc":{
            "age":2,
            "score":-10
        }
    }
);
```

2. **$set**对内容重新设定
   - 语法 : {"$set":{"成员":新的内容}
     // 将90岁的人,评分设置为90分

```
db.student.update(
    {"age":90},
    {
        "$set":{
            "score":100
        }
    }
);
```

3. **$unset**删除某个成员
   - 语法 : {"$unset":{"成员":1}
     // 删除"地三鲜"的年龄和分数信息

```
db.student.update(
    {"name":"地三鲜"},
    {
        "$unset":{
            "age":1,
            "score":1
        }
    }
);
```

4. **$push**将指定内容追加到指定成员(基本上是数组)
    - 语法 : {"$push":{"成员":value}
      // 给"地三鲜"添加一门"毛概" (此时地三鲜没有课程信息)

```
db.student.update(
    {"name":"地三鲜"},
  {
        "$push":{
          "subject":"毛概"
        }
    }
);
```

// 给"地三鲜"追加一门"邓论"

```
db.student.update(
    {"name":"地三鲜"},
    {
        "$push":{
            "subject":"邓论"
        }
    }
);
```

5. **$pushAll**追加数组的方式已经在高版本中淘汰了，可以使用$each添加一个数组到属性值中

- // 添加一道菜”毛血旺”,在多追加两种配菜”["豆腐","油菜"]”

```
db.student.insert({"name":"毛血旺"});
db.student.update(
    {"name":"毛血旺"},
    {
        "$push":{
            "cai":{
                "$each":["豆腐","鸭血","豆芽"]
            }
        }
    }
);
```

6. **$addToSet**追加到数据内容,先判断是否存在,如果存在,不做任何修改,如果不存在,则会追到到数组内容中
   - 语法 : {"$addToSet":{"成员":内容}
     // 向”毛血旺”多加一道配菜”豆腐”

```
db.student.update(
    {"name":"毛血旺"},
    {
      "$addToSet":{
            "cai":"豆腐"
       }
    }
);
// 因为上一次操作,毛血旺配菜的数组中已经存在豆腐了,所以没变化
```

```
// 再追加油菜,而油菜是不存在的,所以油菜被追加进去了
db.student.update(
    {"name":"毛血旺"},
    {
        "$addToSet":{
            "cai":"油菜"
        }
    }
);
```

7. **$pop**删除数组内的数据
   - 语法 : {"$pop":{"成员":数组}

   // 删除地三鲜的第一门课程

```
db.student.update(
    {"name":"地三鲜"},
  {
        "$pop":{
          "subject":-1
        }
    }
);
```

   // 删除地三鲜的最后一门课程

```
db.student.update(
    {"name":"地三鲜"},
    {
        "$pop":{
            "subject":1
        }
    }
);
```

8. **$pull**从数组中删除指定数据 , 存在,删除!不存在,没变化

- 语法 : {"$pull":{"成员":数组}
  // 删除毛血旺配菜中的鲍鱼,不存在鲍鱼,所有没变化.

```
db.student.update(
    {"name":"毛血旺"},
  {
        "$pull":{
          "cai":"鲍鱼"
        }
    }
);
```

  // 删除毛血旺配菜中的油菜,油菜存在,所有删除.

```
db.student.update(
    {"name":"毛血旺"},
    {
        "$pull":{
            "cai":"油菜"
        }
    }
);
```

9. **$pullAll**一次性删除多个内容
   - 语法 : {"$pullAll":{"成员":[“值1”,”值2”,...]}
     // 删除孙大仙的毛概和计算机两门课

```
db.student.insert({
    "name":"孙大仙",
    "subject":["毛概","邓论","计算机"]
});

db.student.update(
    {"name":"孙大仙"},
    {
        "$pullAll":{
            "subject":["毛概","计算机"]
        }
    }
);
```

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"}
      // 将毛血旺的 “cai”改成”配菜”

```
db.student.update(
    {"name":"毛血旺"},
{
        "$rename":{
            "cai":"配菜"
        }
    }
);
```

3.4 查询

语法:db.dept.find({查询条件});

- 查看一条数据

```
db.dept.insert({
    "did":1,
    "dname":"开发部",
    "address":"北京海淀"
});
db.dept.insert({
    "did":2,
    "dname":"市场部",
    "address":"上海虹桥"
});

db.dept.findOne();
```

- 查询did=1的数据

```
db.dept.find({
    "did":1
});
```

- 投影查询（查询部分属性，控制属性的显示）did=1的数据 0:不显示 1显示

  投射里 除了_id以外，要么全是1，要么全是0，否则就报错："Projection cannot have a mix of inclusion and exclusion."

```
db.dept.find(
    {"did":1},
    {"dname":1}
);

db.dept.find(
    {"did":1},
    {"dname":1,"address":1}
);
```

- 漂亮的显示,列太少看不出来效果.不妨自己动手,列多一些

在命令行模式下，会格式化bson

```
db.dept.find(
    {"did":1},
    {"dname":1,"address":1}
).pretty();
```

- findOne()自带格式化功能.也是列多才有效果

```
db.dept.findOne();
```

3.4.1 关系运算

操作	条件格式	例子	RDBMS中的条件
等于	{key:value}	db.col.find({字段名:值}).pretty()	where 字段名=值
大于	{key:{$gt:value}}	db.col.find({字段名:{$gt:值}}).pretty()	where 字段名>值
小于	{key:{$lt:value}}	db.col.find({字段名:{$lt:值}}).pretty()	where 字段名<值
大于等于	{key:{$gte:value}}	db.col.find({字段名:{$gte:值}}).pretty()	where 字段名>=值
小于等于	{key:{$lte:value}}	db.col.find({字段名:{$lte:值}}).pretty()	where 字段名<=值
不等于	{key:{$ne:value}}	db.col.find({字段名:{$ne:值}}).pretty()	where 字段名!=值

- 初始化数据

```
db.student.drop();
db.student.insert({"name":"张
三","sex":"男","age":19,"address":"高碑店"});
db.student.insert({"name":"李
四","sex":"女","age":18,"address":"海淀区"});
db.student.insert({"name":"王
五","sex":"女","age":21,"address":"东城区"});
db.student.insert({"name":"赵
六","sex":"女","age":21,"address":"高碑店"});
db.student.insert({"name":"孙
七","sex":"男","age":22,"address":"东城区"});
db.student.insert({"name":"田
八","sex":"女","age":22,"address":"王府井"});
db.student.insert({"name":"钱
九","sex":"男","age":23,"address":"东城区"});
db.student.insert({"name":"周
十","sex":"男","age":17,"address":"高碑店"});
```

- 查询女同学

```
db.student.find({
    "sex":"女"
});
```

- 查询年龄大于18岁的学生(json格式中套用json格式)

```
db.student.find({
    "age": {
        "$gt": 18
    }
});

// 18岁以上的女同学
db.student.find({
    "age": {
        "$gt": 18
    },
    "sex":"女"
});
```

3.4.2 逻辑运算

- 与$and
- 或$or
- 非$not/$nor
- 查询年龄在18到20之间的同学

```
db.student.find({
    "age": {
        "$gte": 18,
        "$lte": 20
    },
});
```

- 18岁以上,或者性别是女的同学

```
db.student.find({
  "$or":[
      {"age":{"$gt":18}},
      {"sex":{"$eq":"女"}}
    ]
});
```

- 22岁以下,性别是男的学生($nor指的是对$or操作取反)

```
db.student.find({
  "$nor":[
      {"age":{"$gt":22}},
      {"sex":{"$eq":"女"}}
    ]
});
```

3.4.3 求模

求模(求余数)

- 年龄正好是2的倍数.也就是age%2 == 0

```
db.student.find({
    "age":{
        "$mod":[2,0]
    }
});
```

- age%20==1

```
db.student.find({
    "age":{
        "$mod":[2,1]
    }
});
```

3.4.4 范围

- 只要是数据库,必然会存在这些操作
  - 在范围内 $in
  - 不在范围内 $nin
- 姓名是张三,李四,王五的范围查询

```
db.student.find({
    "name":{
        "$in":["张三","李四","王五"]
    }
});
```

- 范围取反

```
db.student.find({
    "name":{
        "$nin":["张三","李四","王五"]
    }
});
```

3.4.5 数组查询

- 初始化数据

```
db.student.insert({"name":"老孙 -
A","sex":"男","age":17,"address":"中关村1",
"subject":["语文","数学","英语"]});
db.student.insert({"name":"老孙 -
B","sex":"女","age":18,"address":"中关村2",
"subject":["语文","数学","英语","物理"]});
db.student.insert({"name":"老孙 -
C","sex":"男","age":19,"address":"中关村3",
"subject":["语文","数学","英语","物理","化学"]});
db.student.insert({"name":"老孙 -
D","sex":"女","age":20,"address":"中关村4",
"subject":["物理","化学"]});
db.student.insert({"name":"老孙 -
E","sex":"男","age":21,"address":"中关村5",
"subject":["语文","数学","化学"]});
```

- 查询同时参加 语文 和 数学 课程的学生

```
db.student.find({
    "subject" : {
        "$all" : ["语文","数学"]
    }
});
```

- 查询地址是"中关村3"的学生的另一种写法

```
db.student.find({
    "address" : {
        "$all" : ["中关村3"]
    }
});
```

现在,集合中保存的信息是数组信息,那么我们就可以使用索引查询了.key.index

- 查询第二门(index=1)课程是数学的学生

```
db.student.find({
    "subject.1" : "数学"
});
```

- 查询只参加两门课程的学生

```
db.student.find({
    "subject" : {
        "$size": 2
    }
});
```

- 查询年龄19岁,但是课程只返回**前两门**课程,使用$slice关键字

```
db.student.find(
    {"age":19},
    {
    "subject" : {
        "$slice":2
    }
});
```

- 返回**后两门**课程

```
db.student.find(
    {"age":19},
    {
    "subject" : {
        "$slice":-2
    }
});
```

- 返回中间的课程 索引1开始（包含索引）,返回2门课

```
db.student.find(
    {"age":19},
    {
    "subject" : {
        "$slice": [1,2]
    }
});
```

3.4.6 嵌套集合

- 初始化数据,养宠物

```
db.student.insert({
    "name":"孙大仙",
    "sex":"男",
    "age":31,
    "address":"天宫院",
    "subject":["java","mongo","html"],
    "pets":[
        {
            "name":"大G",
            "age":3,
            "brand":"狗"
        },
        {
            "name":"大咪",
            "age":4,
            "brand":"猫"
        }
    ]
});

db.student.insert({"name":"乔
丹","sex":"男","age":47,"address":"芝加哥","subject":
["数学","英语"],
```

```
"pets":[
    {"name":"杰瑞","age":2,"brand":"鼠"},
    {"name":"山姆","age":3,"brand":"猫"}
]});

db.student.insert({"name":"kobe","sex":"男","age":40
,"address":"洛杉矶","subject":["英语","篮球"],
"pets":[
    {"name":"克里斯提娜","age":2,"brand":"金丝雀"},
    {"name":"雪瑞","age":12,"brand":"马"}
]});
```

- 查询20岁以上并且养猫的人

```
db.student.find(
    {
        "age": {
            "$gte":20
        },
        "pets" : {
            "$elemMatch" : {
                "brand":"猫"
            }
        }
    }
);
```

3.4.7 字段是否存在判断

- 查询养宠物的学生(不养的就是false)

```
db.student.find({
    "pets":{
        "$exists" : true
    }
});
```

3.4.8 条件过滤where

- 年龄40岁以上的学生

```
db.student.find({
    "$where":"this.age > 40"
});
```

```
db.student.find({
    $where: "this.age > 40"    // $where 没加引号也可以
});
```

```
db.student.find("this.age>=40");   // booster虽然报红色浪线，但并不影响执行
```

- 查询30到40岁之间的同学

```
db.student.find({
    $and:[
        {$where:"this.age > 30"},
        {$where:"this.age < 40"}
    ]
});
```

- 这样的写法,可以实现数据的查询.
- 但是最大的缺点是将MongoDB中的BSON数据转成了Javascript语法结构循环验证

- 这样的话,不方便使用数据库的索引机制.并不推荐使用
- 因为MongoDB中索引的效果，对查询的提升是相当明显的.

3.4.9 正则运算

想模糊查询,必须使用正则.

- 姓名包含孙字的学生(不要加双引号)

```
db.student.find({
    "name" : {
        "$regex" : /孙/
    }
});
```

- 以下用简略的写法

```
db.student.find({
    "name" : /孙/
});
```

- 姓名中包含字母a,忽略大小写

```
db.student.find({
    "name" : /a/i
});
```

- 数组查询也同理,查询学习课程有化字的

```
db.student.find({
    "subject" : /化/
});
```

3.4.10 排序

- 年龄**升序**排列

```
db.student.find().sort({
    "age" : 1
});
```

- 年龄**降序**排列

```
db.student.find().sort({
    "age" : -1
});
```

自然排序:数据保存的先后顺序

- 最新添加的靠前

```
db.student.find().sort({
    "$natural" : -1
});
```

3.4.11 分页

- skip(n)：跨过多少行 `(页码-1) * 每页数量`
- limit(n)：每页数量

// 年龄排序:第1页

```
db.student.find().skip(0).limit(5).sort({
    "age":1
});
```

// 年龄排序:第2页

```
db.student.find().skip(5).limit(5).sort({
    "age":1
});
```

3.5 游标

- 所谓的游标,就是让数据一行行的操作.类似与resultSet数据结构.
- 使用find()函数返回游标 `var you = db.student.find();`
- 要想操作返回的游标,我们有两个函数可以使用
    - hasNext() 判断是否有下一行数据
    - next() 取出当前数据

- 循环取值

```
var you = db.student.find();
while(you.hasNext()){
    print(you.next().name);
}
```

- 按照json格式打印

```
var you = db.student.find();
while(you.hasNext()){
    printjson(you.next());
}
```

4、索引

- 与以往的数据库一样.加快检索性能.

- 索引分为两种,一种自动创建的.一种手动创建的.
- 准备一个简单的集合:

```
db.person.insert({"name":"吕
布","sex":"男","age":19,"address":"高碑店"});
db.person.insert({"name":"赵
云","sex":"女","age":18,"address":"海淀区"});
db.person.insert({"name":"典
韦","sex":"女","age":21,"address":"东城区"});
db.person.insert({"name":"关
羽","sex":"女","age":21,"address":"高碑店"});
db.person.insert({"name":"马
超","sex":"男","age":22,"address":"东城区"});
db.person.insert({"name":"张
飞","sex":"女","age":22,"address":"王府井"});
db.person.insert({"name":"黄
忠","sex":"男","age":23,"address":"东城区"});
db.person.insert({"name":"夏侯
惇","sex":"男","age":17,"address":"高碑店"});
```

// 查看当前集合下的索引

```
db.person.getIndexes();
```

结果:

Key	Value	Type
(1)	{ v : 2, key : { _id : 1 }, name : "_id_", ns : "test.student" }	Object
v	2 版本	Int32
key	{ _id : 1 }	Object
_id	1 升序	Double
name	_id_ 索引名	String
ns	test.student 所属数据库的集合	String

v:版本 1:升序 name:索引名 ns:所属数据库的集合

- 创建一个索引：语法：`db.person.ensureIndex({列:1});`
- 为age列创建一个降序索引

```
db.person.ensureIndex({
    "age" : -1
});
```

再次查看：索引的名字是自动命名的.

- 删除一个索引

```
db.person.dropIndex({
    "age" : -1
});
```

- 删除全部索引,默认的索引不能删除

```
db.person.dropIndexes();
```

4.1 唯一索引

用在某一个字段,让该字段的内容不能重复

- 让name不能重复,添加重复的name数据,报错

```
db.user1.insert({
    "name":"a1"
});

db.user1.find();

db.user1.ensureIndex(
    {
        "name":1
    },
    {
        "unique":true
    }
```

```
);
```

4.2 过期索引

程序会出现若干秒之后，信息删除。

这个间隔时间大多数情况并不是很准确。

- 设置10秒后,索引过期

```
db.phones.ensureIndex(
    {"time":1},
    {"expireAfterSeconds":10}
);
```

- 添加数据

```
db.phones.insert({"num":110,"time": ISODate("2020-01-01T22:55:13.369Z")});
db.phones.insert({"num":119,"time": ISODate("2020-01-02T22:55:13.369Z")});
db.phones.insert({"num":120,"time": ISODate("2020-01-03T22:55:13.369Z")});
db.phones.insert({"num":114,"time": new Date()});
```

10秒(时间不准确,应该说一段时间后)后,再次查询.数据全都消失了...

1、存储在过期索引字段的值必须是指定的时间类型
　　说明：比如是ISODate或者ISODate数组，不能使用时间戳，否者不能被自动删除
　　比如> db.phones.insert({time:1})，这种是不能被删除的
　　过期索引中最好的时间类型是ISODate（0时区的时间），0时区到Date会有一定的换算程序，所以用Date删除的时间偏差更大（但会被删除）
2、如果指定了ISODate()数组，则按照最小的时间进行删除。
3、过期索引不能是复合索引，因为我们不能指定两个过期时间索引
4、删除时间不是精确的
　　说明：删除过程是由后台程序每60s跑一次，而且删除也需要一些时间，索引存在误差

4.3 全文索引

在以往的应用中,经常会用到模糊查询,而模糊查询往往并不是很准确,因为他只能查询A列或者B列.

全文索引就来解决这个问题

- 创建一个新的集合

```
db.news.insert({"title":"NoSQL","nei":"MongoDB"});
db.news.insert({"title":"js java","nei":"前端技术"});
db.news.insert({"title":"编程语言","nei":"java"});
db.news.insert({"title":"java","nei":"好语言"});
db.news.insert({"title":"java","nei":"java"});
```

- 设置全文索引

```
db.news.ensureIndex({
    "title":"text",
    "nei":"text"
});
```

- 进行模糊查询
- 全文检索$text判断符，数据查询$search运算符
- 查询指定关键字: {"$search" : "关键字"}
  - 查询多个关键字(或关系): `{"$search" : "关键字1 关键字2 ..."}` 空格
  - 查询多个关键字(与关系): `{"$search" : "\"关键字\""}` 双引转义
- 查询单个内容

```
db.news.find({
    "$text":{
        "$search":"java"
    }
});
```

- 查询java或js的新闻

```
db.news.find({
    "$text":{
        "$search":"java js"
    }
});
```

- 查询同时包含java和js的新闻

```
db.news.find({
    "$text":{
        "$search":"\"java\" \"js\""     // 两个并且的关键字之间的空格有没有都可以
    }
});
```

- 查询包含java,但是不包含js的内容

```
db.news.find({
    "$text":{
        "$search":"java -js"
    }
});
```

MongoDB的特色：在进行全文检索的时候，还可以使用相似度打分的机制来检验查询的结果.

- 给相似度打分

```
db.news.find(
    {
        "$text":{
            "$search":"java"
        }
    },
    {
       "score":{
           "$meta":"textScore"
       }
    }
);
```

- 分值越大，越相似。也可以进行打分的排序（降序）

```
db.news.find(
    {
        "$text":{
            "$search":"java"
        }
    },
    {
       "score":{
           "$meta":"textScore"
```

```
        }
    }
).sort({
    "score":{
            "$meta":"textScore"
    }
});
```

- 为所有字段设置全文索引(先删掉全部索引,再创建)

```
db.news.dropIndexes();
db.news.ensureIndex({"$**":"text"});
```

虽然很简单.但是尽量别用,因为"慢"

4.4 地理信息索引

- 2DSphere：球面索引
- 2D索引：摇一摇，大众点评，美团都是基于2D索引的，保存的信息都是坐标，坐标都是经纬度.
- 初始化店铺集合

```
db.shops.insert({'loc':[10,10]});
db.shops.insert({'loc':[11,11]});
db.shops.insert({'loc':[13,12]});
db.shops.insert({'loc':[50,14]});
db.shops.insert({'loc':[66,66]});
db.shops.insert({'loc':[110,119]});
db.shops.insert({'loc':[93,24]});
db.shops.insert({'loc':[99,54]});
db.shops.insert({'loc':[77,7]});
```

- 创建2D索引

```
db.shops.ensureIndex({
    "loc":"2d"
});
```

- 2D索引创建完成之后,我们就可以实现坐标的查询了,有两种方式:
  - $near : 查询距离某个点距离最近的坐标点;
  - $geoWithin : 查询某个形状内的坐标点;
- 假设我的坐标是[11,11],查询离我最近的点

```
db.shops.find({
    "loc":{
        "$near":[11,11]
    }
});
```

结果会将集合中前100个点都返回,可是太多了.我们要设置范围

- 设置查询范围,5个点内的

```
db.shops.find({
    "loc":{
        "$near":[11,11],
        "$maxDistance":5
    }
});
```

虽然支持最大的距离.但是不支持最小距离，但我们可以设置形状内查询

- 矩形范围 $box : {"$geoWithin" : {"$box" : [[x1,y1] , [x2,y2]]}

```
db.shops.find({
    "loc":{
        "$geowithin":{
            "$box":[[9,9],[11,11]]
        }
    }
});
```

- 圆形范围 $center : {"$geoWithin" : {"$center" : [[x1,y1] , r]}

```
db.shops.find({
    "loc":{
        "$geowithin":{
            "$center":[[9,9],2]
        }
    }
});
```

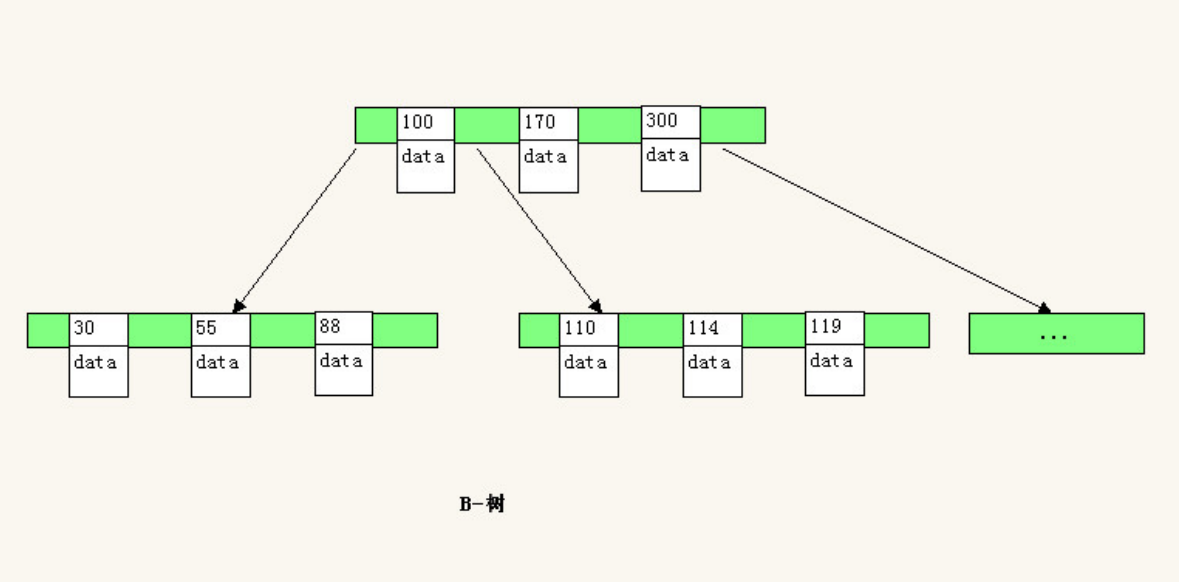
- 多边形范围 $polygon : {"$polygon" : [[x1,y1] , [x2,y2] , [x3,y3] , ...]}

```
db.shops.find({
    "loc":{
        "$geowithin":{
            "$polygon":[[9,9],[12,15],[66,7]]
        }
    }
});
```

4.5 索引底层实现原理分析

- MongoDB 是文档型的数据库，是一种 NOSQL，它使用BSON格式保存数据。

- 比如之前我们的表可能有用户表、订单表等等，还要建立他们之间的外键关联关系。但是BSON就不一样了，这种形式更简单，通俗易懂。
- MongoDB使用B-树（B树），所有节点都有Data域，只要找到指定索引就可以进行访问，无疑单次查询平均快于Mysql。
- Mysql作为一个关系型数据库，数据的关联性是非常强的，区间访问是常态，B+树由于数据全部存储在叶子节点，并且通过指针串在一起，这样就很容易的进行区间遍历甚至全部遍历。
- B-树是一种自平衡的搜索树，形式很简单：



B-树

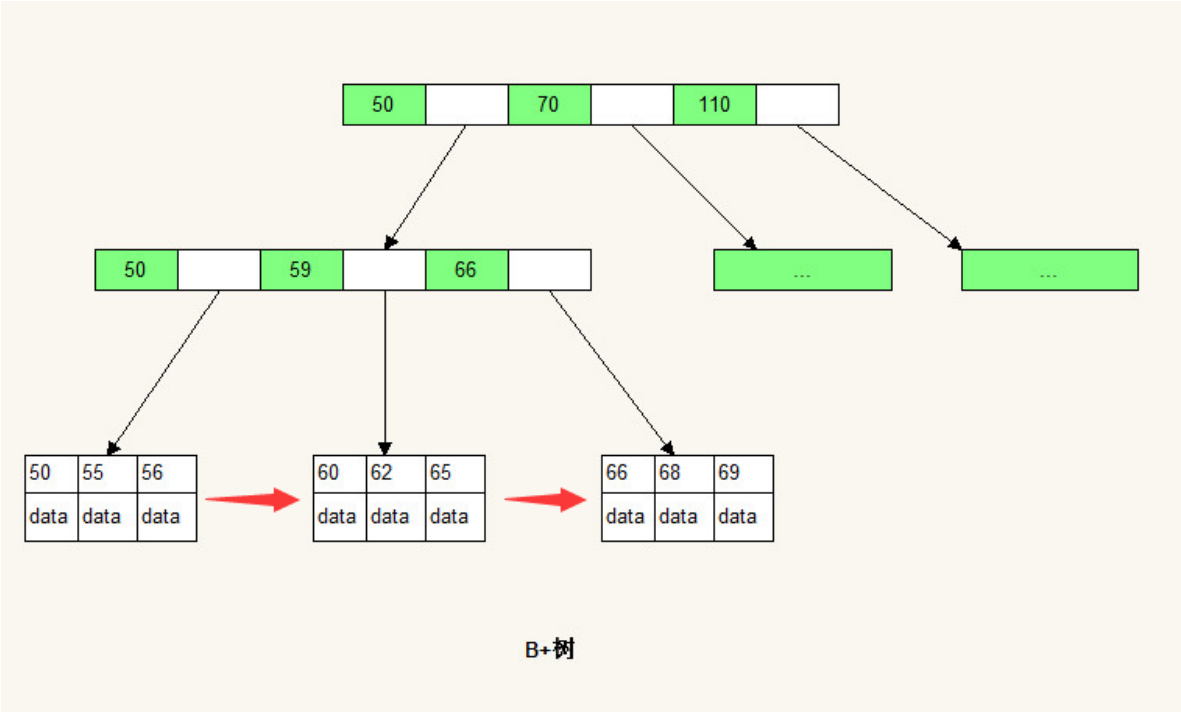
- B-树的特点:

  （1）多路，非二叉树

  （2）每个节点既保存索引，又保存数据

  （3）搜索时相当于二分查找

- B+树是B-树的变种



B+树

- B+树的特点:

  （1）多路非二叉

  （2）只有叶子节点保存数据

  （3）搜索时相当于二分查找

  （4）增加了相邻接点的指向指针。

- 从上面我们可以看出最核心的区别主要有俩，

  一个是数据的保存位置：B树保存在所有的节点中，B+树保存在叶子节点

  一个是相邻节点的指向：B树叶子节点之间没有指针，B+树有

  就是这俩造成他们的差别：

  （1）B+树查询时间复杂度固定是 O(logn)，B-树查询复杂度最好是 O(1)

  （2）B+树相邻接点的指针可以大大增加区间遍历性，而B-树每个节点 key 和 data 在一起，遍历不方便。

  （3）B+树更适合外部存储，也就是磁盘存储。由于内节点无 data 域，每个节点能索引的范围更大更精确

（4）注意这个区别相当重要，是基于（1）（2）（3）的，B-树每个节点即保存数据又保存索引，所以磁盘IO的次数很少，B+树只有叶子节点保存，磁盘IO多，但是区间访问比较好

总结：

1、区间访问是关系型数据库的法宝，所以要使用B+树

2、单一查询mongo用的最多，所以使用B-树

- 为什么关系型数据库遍历访问较多？而mongo单一访问较多呢？
  - mysql经常做变关联查询，而单表查询是不常见的，商品->订单->库存->用户等等多级关联，是家常便饭
  - mongo虽然有也lookup类似join的关联查询，但基本不用。在MongoDB中，根本不推荐这么设计数据关联。反之，如果你非要设计成mysql的数据关联，那么，你就用错mongo了，还是放弃mongo，选择mysql吧。
  - 用mongo如何设计数据关联呢 ？看下面的代码：

```
db.class.insert(
    {
        "id":1,
        "className":"一年一班",
        "students":[
            {"name":"吕布", "gender":"男"},
            {"name":"貂蝉", "gender":"女"}
        ]
    },
    {
        "id":2,
        "className":"一年二班",
        "students":[
            {"name":"韩立", "gender":"男"},
```

```
            {"name":"南宫婉", "gender":"女"}
        ]
    },
);
```

4.7 索引的限制

1. 如果MongoDB的索引项超过索引限制，即1024 bytes，将不会创建该索引（创建失败），并且插入数据和更新数据会报错
2. 针对分片的collections，当数据迁移时，如果数据块中包含索引属性超过了索引限制数据块的迁移将会失败
3. 一个collections最多能够有64个索引
4. 针对索引的全名，包含命名空间和“.”分隔符,如:`<database>.<collection name>.$<index name>`,最多不超过128 characters
5. 针对复合索引,包含的索引属性不能够超过31个属性
6. 查询不能够同时使用文本索引和地理空间索引(Queries cannot use both text and Geospatial Indexes)
7. 包含2d sphere属性的索引，只能够针对地理空间属性

5、聚合(重点)

- MongoDB的产生,是依靠着大数据时代的到来.
- 所谓的大数据,其实就进行数据的抓取,收集,汇总,这样就产生信息的统计操作,这就叫做”聚合”,也就是”分组统计”

5.1 集合的数据量

- 集合中数据量

```
db.student.count();
```

- 模糊查询之后的数据量,包含”孙”

```
db.student.count({"name":/孙/i});
```

查询全部也是模糊查询的一种.只不过是没有条件的模糊查询.
没有条件的查询,永远比条件查询快很多.

5.2 消除重复数据

- 沿用了oracle的关键字. Distinct
- 没有直接的函数支持.所以我们只能用,最原始的runCommand()
- 查询所有name,消除掉重复的name

```
db.runCommand({
    "distinct":"student",
    "key":"name"
});
```

5.3 MapReduce

Map：数据映射（数据取出/获取/来源）

Reduce：归约（数据处理）

可以理解为数据聚集一起，统计，找出需要的各种结果。可以百度百科“MapReduce”，了解更深层次的内容

更简单的理解：select xxxx + group by

- 初始化集合数据,员工集合

```
db.emps.insert({"name":"张
三","sex":"男","age":19,"job":"程序
猿","salary":5000});
db.emps.insert({"name":"李
四","sex":"女","age":18,"job":"美工","salary":6000});
db.emps.insert({"name":"王
五","sex":"女","age":21,"job":"测试","salary":7000});
db.emps.insert({"name":"赵
六","sex":"女","age":21,"job":"程序
猿","salary":5500});
db.emps.insert({"name":"孙
七","sex":"男","age":22,"job":"测试","salary":8000});
db.emps.insert({"name":"田
八","sex":"女","age":22,"job":"程序
猿","salary":3000});
db.emps.insert({"name":"钱
九","sex":"男","age":23,"job":"美工","salary":4500});
db.emps.insert({"name":"周
十","sex":"男","age":17,"job":"程序
猿","salary":9000});
```

需求：按照职位分组,取出每个职位的人名

1. 编写分组定义

```
var jobMap = function(){
    emit(this.job , this.name); // 按照job分组,取出
name
}
```

第一组{key:"程序猿",values:[name,name,...]}

2. 编写reduce操作

```
var jobReduce = function(key,values){
    return {job:key,names:values};
}
```

3. mapReduce

```
db.runCommand({
    "mapreduce":"emps",          // 操作emps集合
    "map":jobMap,                // 定义map
    "reduce":jobReduce,          // 定义reduce
    "out":"emps_job_names"       // 将统计的结果输出到
emps_job_names集合中
});
```

4. 执行之后,所有数据都保存在一个新的集合emps_job_names中了.

```
db.emps_job_names.find();
```

- 案例：统计出男女的人数，平均工资，最高工资，最低工资，员工姓名
  (sql中几乎写不出来,只能存储过程,或重名调用,Mongo来搞定)

```
var sexMap = function(){
    //定义分组的条件,以及每个集合要取出的内容
    emit(this.sex,{
        "ccount":1,
        "csal":this.salary,
        "cavg":this.salary,
        "cmax":this.salary,
        "cmin":this.salary,
        "cname":this.name
    });
}
```

```
var sexReduce = function(key , values){
    var total = 0; //总数
    var sum = 0; // 工资和
    var max = values[0].cmax; // 默认第一个为最高工资
    var min = values[0].cmin; // 默认第一个为最低工资
    var names = new Array(); // 定义姓名数组
    for(var i in values){
        total += values[i].ccount; // 人数++
        sum += values[i].csal; // 累加工资和

        if(max < values[i].cmax){
            max = values[i].cmax;  //最高工资
        }

        if(min > values[i].cmin){
            min = values[i].cmin;  //最低工资
        }

        names[i] = values[i].cname; // 保存姓名
    }

    var avg = (sum/total).toFixed(2); // 平均工资

    return {
```

```
        "count":total,
        "avg":avg,
        "sum":sum,
        "max":max,
        "min":min,
        "names":names
    };
}


db.runCommand({
    "mapreduce":"emps",
    "map":sexMap,
    "reduce":sexReduce,
    "out":"emps_info"
});

db.emps_info.find();
```

5.4 聚合框架

- MapReduce虽然强大，但是写起来的复杂度也是相当的高
- Mongo2.x之后,提供了聚合框架. 函数 : aggregate()

5.4.1 $group

分组

- 每个职位的人数 $sum每次累加1

```
db.emps.aggregate({
    "$group":{
        "_id":"$job",
        "job_count":{
            "$sum":1
        }
    }
});
```

- 每个职位的总工资

```
db.emps.aggregate({
    "$group":{
        "_id":"$job",
        "job_salary_sum":{
            "$sum":"$salary"
        }
    }
});
```

- 每个职位的总工资，平均工资，最高工资，最低工资

```
db.emps.aggregate({
    "$group":{
        "_id":"$job",
        "sum":{
            "$sum":"$salary"
        },
        "avg":{
            "$avg":"$salary"
        },
        "max":{
            "$max":"$salary"
        },
        "min":{
            "$min":"$salary"
        },
```

```
    }
});
```

- 每个职位的工资数据（使用$push，以数组的形式显示每个职工的工资）

```
db.emps.aggregate({
    "$group":{
        "_id":"$job",
        "salary":{
            "$push":"$salary"
        }
    }
});
```

- 每个职位的员工姓名

```
db.emps.aggregate({
    "$group":{
        "_id":"$job",
        "xingming":{
            "$push":"$name"
        }
    }
});
```

使用$push,可以数组显示.但是会出现重复.我们使用一个关键字来取消重复

- 取消员工姓名的重复

```
db.emps.aggregate({
    "$group":{
        "_id":"$job",
        "xingming":{
            "$addToSet":"$name"
        }
    }
});
```

5.4.2 $project

数据列的显示规则,投影查询

1|true 显示

0|false不显示

- 不显示_id,显示name

```
db.emps.aggregate({
    "$project":{
        "_id":0,
        "name":1
    }
});
```

- 起别名显示

```
db.emps.aggregate({
    "$project":{
        "_id":0,
        "姓名":"$name",
        "职位":"$job",
        "工资":"$salary"
    }
});
```

- 聚合管道运算符中文文档：https://www.docs4dev.com/docs/zh/mongodb/v3.6/reference/reference-operator-aggregation-abs.html
- 支持四则运算
  - 加 $add
  - 减 $subtract
  - 乘 $multiply
  - 除 $divide
  - 求模 $mod

  // 求年薪

```
db.emps.aggregate({
    "$project":{
        "_id":0,
        "姓名":"$name",
        "职位":"$job",
        "工资":"$salary",
        "年薪":{
            "$multiply":["$salary",12]
        }
    }
});
```

- 支持关系运算
  - 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

- 支持逻辑运算
  - 与 $and
  - 或 $or
  - 非 $not
- 字符串操作
  - 连接 $concat
  - 截取 $substr
  - 小写 $toLower
  - 大写 $toUpper
  - 忽略大小写比较 $strcasecmp

  // 找出工资大于等于8000的员工姓名,工资

```
db.emps.aggregate({
    "$project":{
        "_id":0,
        "姓名":"$name",
        "职位":"$job",
        "工资":"$salary",
        "money":{
            "$gte":["$salary",8000]
        }
    }
});
```

- 查询职位是"程序猿"
  正常比较

```
db.emps.aggregate({
    "$project":{
        "_id":0,
        "姓名":"$name",
        "职位":"$job",
        "is程序猿":{
            "$eq":["$job","程序猿"]
        }
    }
});
```

- 转成大写比较

```
db.emps.aggregate({
    "$project":{
        "_id":0,
        "姓名":"$name",
        "职位":"$job",
        "is程序猿":{
            "$eq":["$job",{"$toUpper":"程序猿"}]
        }
    }
});
```

- 忽略大小写比较

```
db.emps.aggregate({
    "$project":{
        "_id":0,
        "姓名":"$name",
        "职位":"$job",
        "is程序猿":{
            "$strcasecmp":["$job","程序猿"]
        }
    }
});
```

```
// 0 : 相等
// 1 : unicode靠后,例如，在unicode编码中，m>c，意味着m的编码要在c编码的后面
// -1 : unicode靠前
```

- 截取字符串第一个字
  - 运算符使用 UTF-8 编码字节的索引，其中每个代码点或字符都可以使用一到四个字节进行编码。
  - US-ASCII （英文字母）字符使用一个字节编码
  - 带有变音符号的字符和其他拉丁字母字符(即英语字母之外的拉丁字符)使用两个字节进行编码。
  - 中文，日文和韩 Literals 符通常需要三个字节，而其他 Unicode 平面(表情符号，math 符号等)则需要四个字节。

```
db.emps.aggregate({
    "$project":{
        "_id":0,
        "姓名":"$name",
        "职位":"$job",
        "姓氏":{
            "$substrBytes":["$name",0,3]
        }
    }
});
```

5.4.3 $sort

- 1：年龄升序

```
db.emps.aggregate({
    "$sort":{
        "age":1
    }
});
```

- -1 : 工资降序

```
db.emps.aggregate({
    "$sort":{
        "salary":-1
    }
});
```

- 年龄升序，如果年龄相等，则工资降序排列

```
db.emps.aggregate({
    "$sort":{
        "age":1,
        "salary":-1
    }
});
```

5.4.4 $分页处理

$limit : 取出个数
$skip : 跨过个数

- 取出3个值

```
db.emps.aggregate(
    {
        "$project":{
            "_id":0,
            "name":1,
```

```
            "age":1,
            "job":1
        }
    },
    {
        "$limit":3

    }
);
```

- 跨过3行数据 (先跨,再取)

```
db.emps.aggregate(
    {
        "$project":{
            "_id":0,
            "name":1,
            "age":1,
            "job":1
        }
    },
    {
        "$skip":3

    },
    {
        "$limit":3

    }
);
// $skip一定要在$limit前面，先跨过，再取值，否则查询不到数据
```

5.4.5 $unwind

查询数据的时候，会返回数组信息，数组不方便浏览

我们要将数组变成**独立的**字符串.

- 初始数据

```
db.dept.insert({"name":"技术部","业务":["研发","培训","维护"]});
db.dept.insert({"name":"人事部","业务":["招聘","调岗","发工资"]});
db.dept.insert({"name":"市场部","业务":["销售","渠道","开拓"]});
```

- 转换

```
db.dept.aggregate([{"$unwind":"$业务"}]);
```

5.4.6 $out

将查询结果输出到指定集合中.类似mapreduce

- 将投影结果,输出到一个集合中

```
db.emps.aggregate(
    {
        "$project":{
            "_id":0,
            "name":1,
            "job":1
        }
    },
    {
        "$skip":3
    },
    {
        "$limit":3
    },
    {
```

```
        "$out":"emps_test"
    }
);

db.emps_test.find();
```

5.4.7 $geoNear

得到附近的坐标点

帮助文档：https://docs.mongodb.com/manual/reference/operator/aggregation/geoNear/#mongodb-pipeline-pipe.-geoNear

- 初始化数据

```
db.shops.drop();
db.shops.insert({'loc':[10,10]});
db.shops.insert({'loc':[11,11]});
db.shops.insert({'loc':[13,12]});
db.shops.insert({'loc':[50,14]});
db.shops.insert({'loc':[66,66]});
db.shops.insert({'loc':[110,119]});
db.shops.insert({'loc':[93,24]});
db.shops.insert({'loc':[99,54]});
db.shops.insert({'loc':[77,7]});
```

- 得到附近的坐标点
  // 必须先建立2D索引.否则查询报错

```
db.shops.ensureIndex({
    "loc":"2d"
});

db.shops.aggregate({
```

```
    "$geoNear":{
        "near":[11,11],     // 坐标点
        "distanceField":"loc",    // 操作的字段
        "maxDistance":2,    // 最远距离
        "spherical":true     // 按照球面查询
    }
},{
    "$limit":5     // 返回坐标的数量
});
```

5.4.8 $lookup

- 多表关联（3.2版本新增）主要功能：
  - 是将每个输入待处理的文档，经过$lookup 阶段的处理，输出的新文档中会包含一个新生成的数组列（户名可根据需要命名新key的名字 ）。
  - 数组列存放的数据 是 来自 被Join 集合的适配文档，如果没有，集合为空（即 为[]）

基本语法：

```
{
    $lookup:
    {
        from: <collection to join> 从表,
        localField: <field from the input documents>
主表主键,
        foreignField: <field from the documents of
the "from" collection> 从表主键,
        as: <output array field> 输出文档的新增值命名
    }
}
```

- 以上的语法介绍有些枯燥，不易理解，我们直接干案例。
  订单集合，测试数据 如下：

```
db.orders.insert([
    { "order_id" : 1, "pname" : "可乐", "price" :
12, "count" : 2 },
    { "order_id" : 2, "pname" : "薯片", "price" :
20, "count" : 1 },
    { "order_id" : 3 }
]);
```

- 库存集合 ，测试数据 如下：

```
db.stock.insert([
    { "s_id" : 1, "sku" : "可乐", "description":
"product 1", "instock" : 120 },
    { "s_id" : 2, "sku" : "面包", "description":
"product 2", "instock" : 80 },
    { "s_id" : 3, "sku" : "香肠", "description":
"product 3", "instock" : 60 },
    { "s_id" : 4, "sku" : "薯片", "description":
"product 4", "instock" : 70 },
    { "s_id" : 5, "sku" : null,  "description":
"Incomplete" },
    { "s_id" : 6 }
]);
```

- 此集合中的 sku 等同于 订单 集合中的 pname。
- 在这种模式设计下，如果要查询订单表对应商品的库存情况，
  应如何写代码呢？
- 很明显这需要两个集合Join。场景简单，不做赘述，直送答案。
  如下：

```
db.orders.aggregate({
    "$lookup":{
        from: "stock",    // 从表
        localField: "pname", // 主表主键
        foreignField: "sku",  // 从表主键
        as: "shop_stock"  // 新生的关联字段
    }
});
```

上述过程，其实和关系型数据库中的左外连接（left）查询非常相像。

5.5 固定集合

- 规定集合大小，如果保存内容超过了集合的长度，那么会采用LRU的算法(最近最少使用的原则)，将最早的数据移除，来保存新的数据. 就是名将韩信管理粮仓的政策"推陈出新".
- 创建集合必须明确创建一个空集合
- 创建一个集合

```
db.createCollection("depts",{
    "capped":true,  // true:固定集合
    "size":1024,    // 集合空间容量 1024字节
    "max":5         // 最多有5条记录
});
```

- 初始化数据

```
db.depts.insert({"name":"开发部-1","loc":"北京"});
db.depts.insert({"name":"开发部-2","loc":"北京"});
db.depts.insert({"name":"开发部-3","loc":"北京"});
db.depts.insert({"name":"开发部-4","loc":"北京"});
db.depts.insert({"name":"开发部-5","loc":"北京"});
```

- 再加入一条.

```
db.depts.insert({"name":"开发部-6","loc":"北京"});
```

6、Java操作

这里介绍spring提供对MongoDB操作的工具类的使用

- pom

```
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-mongodb</artifactId>
    <version>2.0.9.RELEASE</version>
</dependency>
```

- spring配置如下：在配置文件中配置 MongoTemplate
  （ resources下创建application.xml ）

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"

xmlns:mongo="http://www.springframework.org/schema/d
ata/mongo"

xmlns:context="http://www.springframework.org/schema
/context"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
```

```
http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context.xsd

http://www.springframework.org/schema/data/mongo

http://www.springframework.org/schema/data/mongo/spring-mongo.xsd">

    <mongo:db-factory id="mongoDbFactory" client-uri="mongodb://192.168.204.141:27017/test"/>

    <bean id="mongoTemplate" class="org.springframework.data.mongodb.core.MongoTemplate">
        <constructor-arg name="mongoDbFactory" ref="mongoDbFactory"></constructor-arg>
    </bean>

    <context:component-scan base-package="com.lagou"></context:component-scan>

</beans>
```

- 注入 MongoTemplate 完成增删改查

```
@Autowired
protected MongoTemplate mongoTemplate;
```

```
package com.lagou.entity;
```

```
import
org.springframework.data.mongodb.core.mapping.Docume
nt;
import java.io.Serializable;

@Document(collection = "emps")
public class Emp implements Serializable {
    private String _id;
    private String name;
    private String sex;
    private double age;
    private String job;
    private double salary;
    // get和set，构造省略
```

- 常用方法

```
mongoTemplate.findAll(Student.class) // 查询Student文
档的全部数据
mongoTemplate.findById(<id>, Student.class) // 查询
Student文档id为id的数据
mongoTemplate.find(query, Student.class) // 根据
query内的查询条件查询
mongoTemplate.upsert(query, update, Student.class)
// 修改
mongoTemplate.remove(query, Student.class) // 删除
mongoTemplate.insert(student) // 新增
```

- Query对象
  - 创建一个query对象（用来封装所有条件对象），再创建一个criteria对象（用来构建条件）
  - 精准条件：`criteria.and("key").is("条件")` 相当于 where id = 1
  - 模糊条件：`criteria.and("key").regex("条件")`
  - 封装条件：`query.addCriteria(criteria)`

- 大于（创建新的criteria）：`Criteria gt = Criteria.where("key").gt("条件")`
- 小于（创建新的criteria）：`Criteria lt = Criteria.where("key").lt("条件")`
- `Query.addCriteria(new Criteria().andOperator(gt,lt));`
- 一个query中只能有一个andOperator()。其参数也可以是Criteria数组。
- 排序 ：`query.with(new Sort(Sort.Direction.ASC, "age"). and(new Sort(Sort.Direction.DESC, "date")))`

```
package com.lagou.dao;

import com.lagou.entity.Emp;
import com.mongodb.client.result.DeleteResult;
import com.mongodb.client.result.UpdateResult;
import
org.springframework.beans.factory.annotation.Autowir
ed;
import
org.springframework.data.mongodb.core.MongoTemplate;
import
org.springframework.data.mongodb.core.query.Criteria
;
import
org.springframework.data.mongodb.core.query.Query;
import
org.springframework.data.mongodb.core.query.Update;
import org.springframework.stereotype.Repository;
import org.springframework.util.StringUtils;
```

```java
import java.util.List;
import java.util.regex.Pattern;

/**
 * @Author: GuoAn.Sun
 * @Description:
 */
@Repository("empDao")
public class EmpDaoImpl implements EmpDao {

    @Autowired
    private MongoTemplate mongoTemplate;

    // 添加
    public void save(Emp emp) {
        mongoTemplate.insert(emp);
    }

    // 删除
    public void delete(String id) {
        Query query = new
Query(Criteria.where("_id").is(id));
        DeleteResult result =
mongoTemplate.remove(query, Emp.class);
        long l = result.getDeletedCount();
        if(l>0){
            System.out.println("删除成功");
        }
    }

    // 修改
    public void update(Emp emp) {
        // 查询被修改的对象
        Query query = new
Query(Criteria.where("_id").is(emp.get_id()));
        // 设置全新的数据对象
        Update update = new Update();
```

```
        update.set("name", emp.getName());
        update.set("age", emp.getAge());
        update.set("sex", emp.getSex());
        update.set("job", emp.getJob());
        update.set("salary", emp.getSalary());

        UpdateResult result =
mongoTemplate.upsert(query, update, Emp.class);
        long i = result.getModifiedCount();
        if(i > 0){
            System.out.println("修改成功");
        }
    }

    // 分页查询（模糊）
    // 模糊查询以 【^】开始 以【$】结束 【.*】相当于Mysql中
的%
    public List<Emp> findListPage(Integer pageIndex,
Integer pageSize, String name) {
        Query query = new Query();

        if(!StringUtils.isEmpty(name)){
            // 使用正则拼装模糊查询的条件
            String regex = String.format("%s%s%s",
"^.*", name, ".*$");

            // 构建正则条件对象,
Pattern.CASE_INSENSITIVE忽略正则中的大小写
            Pattern pattern =
Pattern.compile(regex,Pattern.CASE_INSENSITIVE);


 query.addCriteria(Criteria.where("name").regex(patt
ern) );
        }

        // 获取总数量
```

```
        long count = mongoTemplate.count(query,
Emp.class);
        System.out.println("总数量: " + count);

        // 分页查询
        List<Emp> emps =
mongoTemplate.find(query.skip( (pageIndex-
1)*pageSize).limit(pageSize), Emp.class);
        return emps;
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        ClassPathXmlApplicationContext
applicationContext  =
                new
ClassPathXmlApplicationContext("application.xml");
        EmpDao dao =
applicationContext.getBean("empDao",EmpDao.class);

        //dao.save(new Emp("1","1",1,"1",1));

        //dao.delete("611f2d2c03b8fe4bd8022749");

        /*
        Emp emp = new
Emp("611f2dba03b8fe3ac8dde3b7","貂蝉","女",19,"美
女",7250);
        dao.update(emp);
        */

        //List<Emp> emps = dao.findListPage(1, 5,
null);
```

```
        List<Emp> emps = dao.findListPage(1, 5,
"张");
        System.out.println(emps);

    }
}
```