MongoDB中的基本概念及原理

MongoDB介绍

官网地址：https://www.mongodb.com/

MongoDB 是一个基于【分布式文件存储】的数据库，它属于NoSQL数据库。由 C++ 语言编写。旨在为 WEB 应用提供【可扩展】的【高性能】数据存储解决方案。

MongoDB是一个介于**非系数据库**和非关系数据库之间的产品，是非关系数据库当中功能最丰富，最像关系数据库的。它支持的数据结构非常松散，是类似**json**的**bson**格式，因此可以存储比较复杂的数据类型。Mongo最大的特点是它支持的查询语言非常强大，其语法有点类似于面向对象的查询语言，几乎可以实现类似关系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**。

```
NoSQL分类：键值型（key-value）、文档型(document)

MongoDB就是文档型NoSQL数据库，它文档中的数据是以类似JSON的BSON格式进行存储的。我们拿JSON去理解，JSON中的数据，都是key-value，key一般都是String类型的，而value就多种多样了。只有value的类型，后续有专门的讲解。记住value中可以再存储一个文档。
```

MongoDB概念解析

不管我们学习什么数据库都应该学习其中的基础概念，在mongodb中基本的概念是文档、集合、数据库，下面我们挨个介绍。

下表将帮助您更容易理解Mongo中的一些概念：

SQL术语/概念	MongoDB术语/概念	解释/说明
database	database	数据库
table	collection	数据库表/集合
row	document	数据记录行/文档
column	field	数据字段/域
index	index	索引
table joins		表连接,MongoDB不支持
primary key	primary key	主键,MongoDB自动将_id字段设置为主键

通过下图实例，我们也可以更直观的了解Mongo中的一些概念：

id	user_name	email	age	city
1	Mark Hanks	mark@abc.com	25	Los Angeles
2	Richard Peter	richard@abc.com	31	Dallas


```
{
    "_id": ObjectId("5146bb52d8524270060001f3"),
    "age": 25,
    "city": "Los Angeles",
    "email": "mark@abc.com",
    "user_name": "Mark Hanks"
}
{
    "_id": ObjectId("5146bb52d8524270060001f2"),
    "age": 31,
    "city": "Dallas",
    "email": "richard@abc.com",
    "user_name": "Richard Peter"
}
```


数据库

一个mongodb中可以建立多个数据库。

MongoDB的默认数据库为"db"，该数据库存储在data目录中（**安装时，可以默认，可以指定，但是必须该目录是存在的**）。

MongoDB的单个实例可以容纳多个独立的数据库，每一个都有自己的集合和权限，不同的数据库也放置在不同的文件中。

"show dbs" 命令可以显示所有数据的列表。

```
$ ./mongo
MongoDB shell version: 3.0.6
connecting to: test
> show dbs
local  0.078GB
test   0.078GB
>
```

执行 **"db"** 命令可以显示当前数据库对象或集合。

```
$ ./mongo
MongoDB shell version: 3.0.6
connecting to: test
> db
test
>
```

运行"**use**"命令，可以连接到一个指定的数据库。

```
> use local
switched to db local
> db
local
>
```

以上实例命令中，"local" 是你要链接的数据库。

在下一个章节我们将详细讲解MongoDB中命令的使用。

数据库也通过名字来标识。数据库名可以是满足以下条件的任意UTF-8字符串。

- 不能是空字符串（""）。
- 不得含有' '（空格)、.、$、/、\和\0 (空字符)。
- 应全部小写。
- 最多64字节。

有一些数据库名是保留的，可以直接访问这些有特殊作用的数据库。

- **admin**： 从权限的角度来看，这是"root"数据库。要是将一个用户添加到这个数据库，这个用户自动继承所有数据库的权限。一些特定的服务器端命令也只能从这个数据库运行，比如列出所有的数据库或者关闭服务器。
- **local:** 这个数据永远不会被复制，可以用来存储限于本地单台服务器的任意集合
- **config**: 当Mongo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。

文档

文档是一组键值(key-value)对(即 BSON)。**MongoDB 的文档不需要设置相同的字段，并且相同的字段不需要相同的数据类型，这与关系型数据库有很大的区别，也是 MongoDB 非常突出的特点。**

一个简单的文档例子如下：

```
{"site":"www.kaikeba.com", "name":"开课吧"}
```

下表列出了 RDBMS 与 MongoDB 对应的术语：

RDBMS	MongoDB
数据库	数据库
表格	集合
行	文档
列	字段
表联合	嵌入文档
主键	主键 (MongoDB 提供了 key 为 _id)
数据库服务和客户端	
Mysqld/Oracle	mongod
mysql/sqlplus	mongo

需要注意的是：

1. 文档中的键/值对是有序的。
2. 文档中的值不仅可以是在双引号里面的字符串，还可以是其他几种数据类型（甚至可以是整个嵌入的文档)。
3. MongoDB区分类型和大小写。
4. MongoDB的文档不能有重复的键。

5. 文档的键是字符串。除了少数例外情况，键可以使用任意UTF-8字符。

文档键命名规范：

- 键不能含有\0 (空字符)。这个字符用来表示键的结尾。
- .和$有特别的意义，只有在特定环境下才能使用。
- 以下划线"_"开头的键是保留的(不是严格要求的)。

集合

集合就是 MongoDB 文档组，类似于 RDBMS （关系数据库管理系统：Relational Database Management System)中的表格。

集合存在于数据库中，集合没有固定的结构，这意味着你在对集合可以插入不同格式和类型的数据，但通常情况下我们插入集合的数据都会有一定的关联性。

比如，我们可以将以下不同数据结构的文档插入到集合中：

```
{"site":"www.baidu.com"}
{"site":"www.google.com","name":"Google"}
{"site":"www.kaikeba.com","name":"开课吧","num":5}
```

当第一个文档插入时，集合就会被创建。

```
一个collection（集合）中的所有field（域）是collection（集合）中所有document（文档）中包含的field（域）的并集。
```

合法的集合名

- 集合名不能是空字符串""。
- 集合名不能含有\0字符（空字符)，这个字符表示集合名的结尾。
- 集合名不能以"system."开头，这是为系统集合保留的前缀。
- 用户创建的集合名字不能含有保留字符。有些驱动程序的确支持在集合名里面包含，这是因为某些系统生成的集合中包含该字符。除非你要访问这种系统创建的集合，否则千万不要在名字里出现$。

如下实例：

```
db.col.findOne()
```

capped collections

Capped collections 就是固定大小的collection。

它有很高的性能以及队列过期的特性(过期按照插入的顺序). 有点和 "RRD" 概念类似。

Capped collections 是高性能自动的维护对象的插入顺序。它非常适合类似记录日志的功能和标准的 collection 不同，你必须要显式的创建一个capped collection，指定一个 collection 的大小，单位是字节。collection 的数据存储空间值提前分配的。

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。

要注意的是指定的存储大小包含了数据库的头信息。

```
db.createCollection("mycoll", {capped:true, size:100000})
```

- 在 capped collection 中，你能添加新的对象。
- 能进行更新，然而，对象不会增加存储空间。如果增加，更新就会失败 。
- 使用 Capped Collection 不能删除一个文档，可以使用 drop() 方法删除 collection 所有的行。
- 删除之后，你必须显式的重新创建这个 collection。
- 在32bit机器中，capped collection 最大存储为 1e9(1X10的9次方)个字节。

元数据

数据库的信息是存储在集合中。它们使用了系统的命名空间：

```
dbname.system.*
```

在MongoDB数据库中名字空间 .system.* 是包含多种系统信息的特殊集合(Collection)，如下:

集合命名空间	描述
dbname.system.namespaces	列出所有名字空间。
dbname.system.indexes	列出所有索引。
dbname.system.profile	包含数据库概要(profile)信息。
dbname.system.users	列出所有可访问数据库的用户。
dbname.local.sources	包含复制对端（slave）的服务器信息和状态。

对于修改系统集合中的对象有如下限制。

在{{system.indexes}}插入数据，可以创建索引。但除此之外该表信息是不可变的(特殊的drop index命令将自动更新相关信息)。

{{system.users}}是可修改的。 {{system.profile}}是可删除的。

MongoDB 数据类型

下表为MongoDB中常用的几种数据类型:

数据类型	说明	解释	举例
String	字符串	UTF-8 编码的字符串才是合法的。	{“v”:“kkb”}
Integer	整型数值	根据你所采用的服务器，可分为 32 位或 64 位。	{“v”:1}
Boolean	布尔值	用于存储布尔值（真/假）。	{“v”:true}
Double	双精度浮点值	用于存储浮点值	{“v”:3.14}
ObjectID	对象ID	用于创建文档的ID	{“__id”:ObjectId(“123123”)}
Array	数组	用于将数组或列表或多个值存储为一个键	{“arr”:[“a”,“b”]}
Timestamp	时间戳	从开始纪元开始的毫秒数	
Object	内嵌文档	文档可以作为文档中某个key的value	{“o”:{“foo”:“bar”}}
Null	空值	表示空值或者未定义的对象	{“v”:null}
Date	日期	日期时间，用Unix日期格式来存储当前日期或时间。	{“date”:new Date()}
Regular	正则表达式	文档中可以包含正则表达式，遵循JS语法	{“v”:/kkb/i}
Code	代码	可以包含JS代码	{“x”:function(){}}
File	文件	1、二进制转码(Base64)后存储 (<16M) 2、GridFS(>16M)	GridFS 用两个集合来存储一个文件：fs.files与fs.chunks

下面说明下几种重要的数据类型。

ObjectId

ObjectId 类似唯一主键，可以很快的去生成和排序，包含 12 bytes，含义是：

- 前 4 个字节表示创建 **unix** 时间戳,格林尼治时间 **UTC** 时间，比北京时间晚了 8 个小时
- 接下来的 3 个字节是机器标识码
- 紧接的两个字节由进程 id 组成 PID
- 最后三个字节是随机数

MongoDB 中存储的文档必须有一个 _id 键。这个键的值可以是任何类型的，默认是个 ObjectId 对象

由于 ObjectId 中保存了创建的时间戳，所以你不需要为你的文档保存时间戳字段，你可以通过 getTimestamp 函数来获取文档的创建时间:

```
> var newObject = ObjectId()
> newObject.getTimestamp()
ISODate("2017-11-25T07:21:10Z")
```

ObjectId 转为字符串

```
> newObject.str
5a1919e63df83ce79df8b38f
```

字符串

BSON 字符串都是 UTF-8 编码。

时间戳

BSON 有一个特殊的时间戳类型用于 MongoDB 内部使用，与普通的 日期 类型不相关。 时间戳值是一个 64 位的值。其中:

- 前32位是一个 time_t 值（与Unix新纪元相差的秒数）
- 后32位是在某秒中操作的一个递增的 序数

在单个 mongod 实例中，时间戳值通常是唯一的。

在复制集中， oplog 有一个 ts 字段。这个字段中的值使用BSON时间戳表示了操作时间。

> BSON 时间戳类型主要用于 MongoDB 内部使用。在大多数情况下的应用开发中，你可以使用 BSON 日期类型。

日期

表示当前距离 Unix新纪元（1970年1月1日）的毫秒数。日期类型是有符号的, 负数表示 1970 年之前的日期。

```
> var mydate1 = new Date()      //格林尼治时间
> mydate1
ISODate("2018-03-04T14:58:51.233Z")
> typeof mydate1
object
> var mydate2 = ISODate() //格林尼治时间
> mydate2
ISODate("2018-03-04T15:00:45.479Z")
> typeof mydate2
object
```

这样创建的时间是日期类型，可以使用 JS 中的 Date 类型的方法。

返回一个时间类型的字符串:

```
> var mydate1str = mydate1.toString()
> mydate1str
Sun Mar 04 2018 14:58:51 GMT+0000 (UTC)
> typeof mydate1str
string
```

或者

```
> Date()
Sun Mar 04 2018 1
```

SQL术语/概念	MongoDB术语/概念	解释/说明
database	database	数据库
table	collection	数据库表/集合
row	document	数据记录行/文档
column	field	数据字段/域
index	index	索引
table joins		表连接,MongoDB不支持
primary key	primary key	主键,MongoDB自动将_id字段设置为主键

总结起来，RDBMS和NoSQL的区别在于三点：事务、表关联、表结构约束

RDBMS	NoSQL
有事务	无事务（性能提升）
有表关联	数据和数据之间没有关系
有表结构约束	key-value结构，没有约束

举例：

RDBMS中的User表（字段先要预定义）：

```
id              name            address

自增主键、int     非空、字符串      允许空、字符串
```

MongoDB中的User文档（字段不需要预定义）：

```
文档对象1:
{
    “_id”:ObjectID(),
    "name":"James",
```

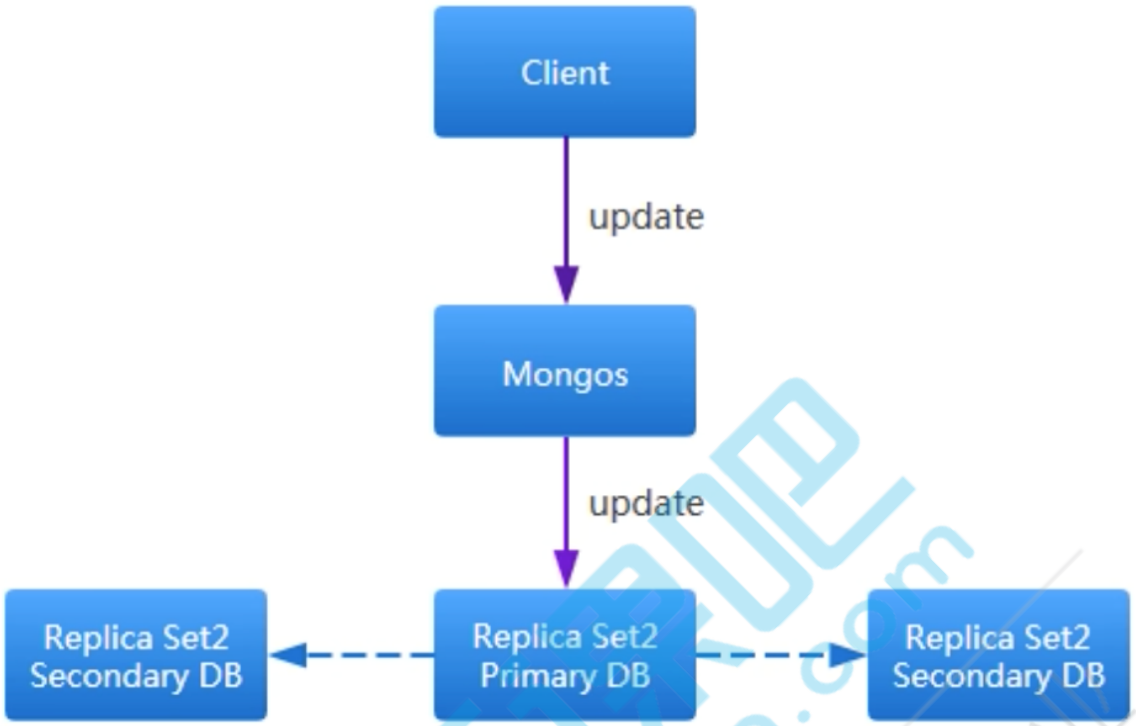
```
    "age":18
}

插入之后，产生了三个列：
id  name    age

文档对象1：
{
    “_id”:ObjectID(),
    "name":"xiaoqiao",
    "address":“my house”
}

插入之后，产生了三个列：
id  name    address

但是会和上面的列计算出一个并集，最后会有4个俩
id  name    age     address
```

图解MongoDB底层原理

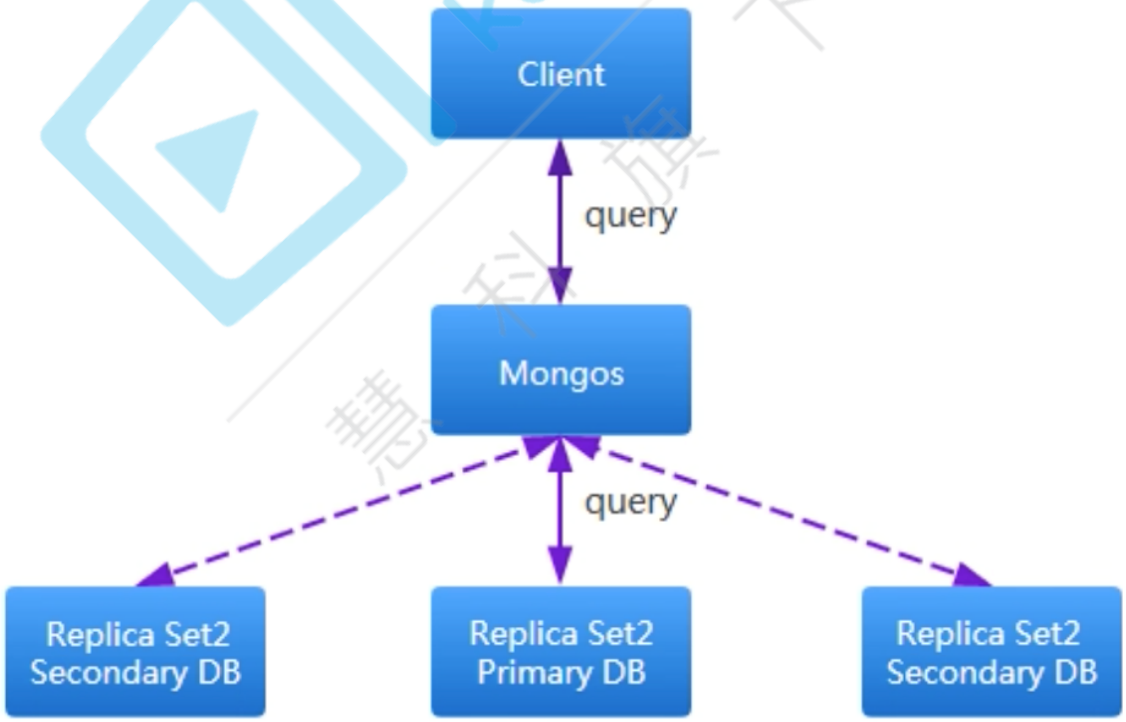
Mongodb的部署方案有**单机部署、主从部署、副本集（主备）部署、分片部署、副本集与分片混合部署。**

副本集集群

对于副本集集群，又有主和从两种角色，写数据和读数据也是不同，写数据的过程是只写到主结点中，由主 结点以异步的方式同步到从结点中：



而读数据则只要从任一结点中读取，具体到哪个结点读取是可以指定的：



副本集与分片混合部署

Mongodb的集群部署方案有三类角色：**实际数据存储节点，配置文件存储节点和路由接入节点。**

- **实际数据存储节点**的作用就是存储数据，
- **路由接入节点**的作用是在分片的情况下起到负载均衡的作用。
- **存储配置存储节点**的作用其实存储的是片键与chunk 以及chunk 与server 的映射关系，用上面的数据表 示的配置结点存储的数据模型如下表：

map1

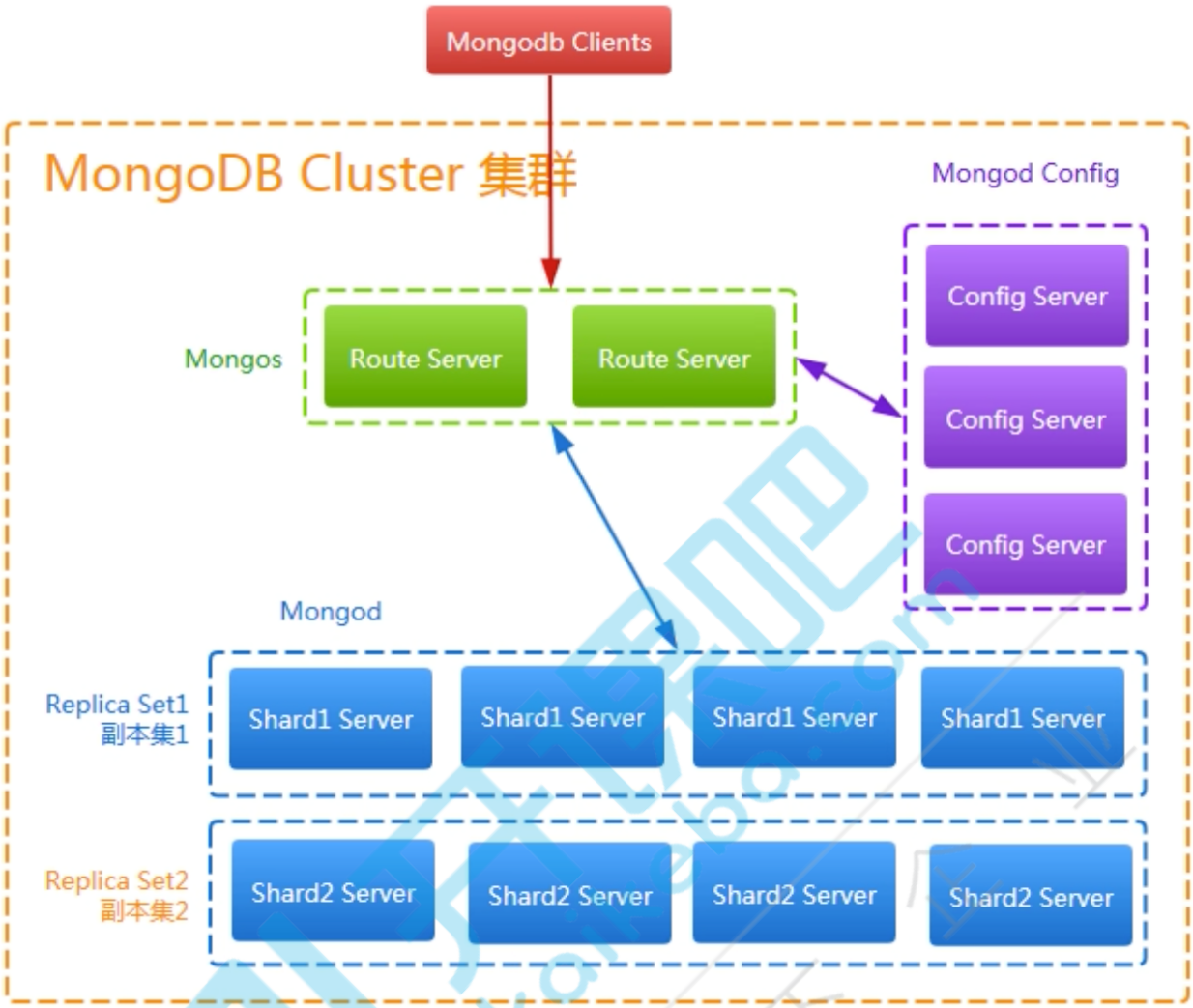
key range	chunk
[0,10}	chunk1
[10,20}	chunk2
[20,30}	chunk3
[30,40}	chunk4
[40,50}	chunk5

map2

chunk	shard
chunk1	shard1
chunk2	shard2
chunk3	shard3
chunk4	shard4
chunk5	shard5

MongoDB的客户端直接与路由节点相连，从配置节点上查询数据，根据查询结果到实际的存储节点上查询和存储数据。

副本集与分片混合部署方式如图：

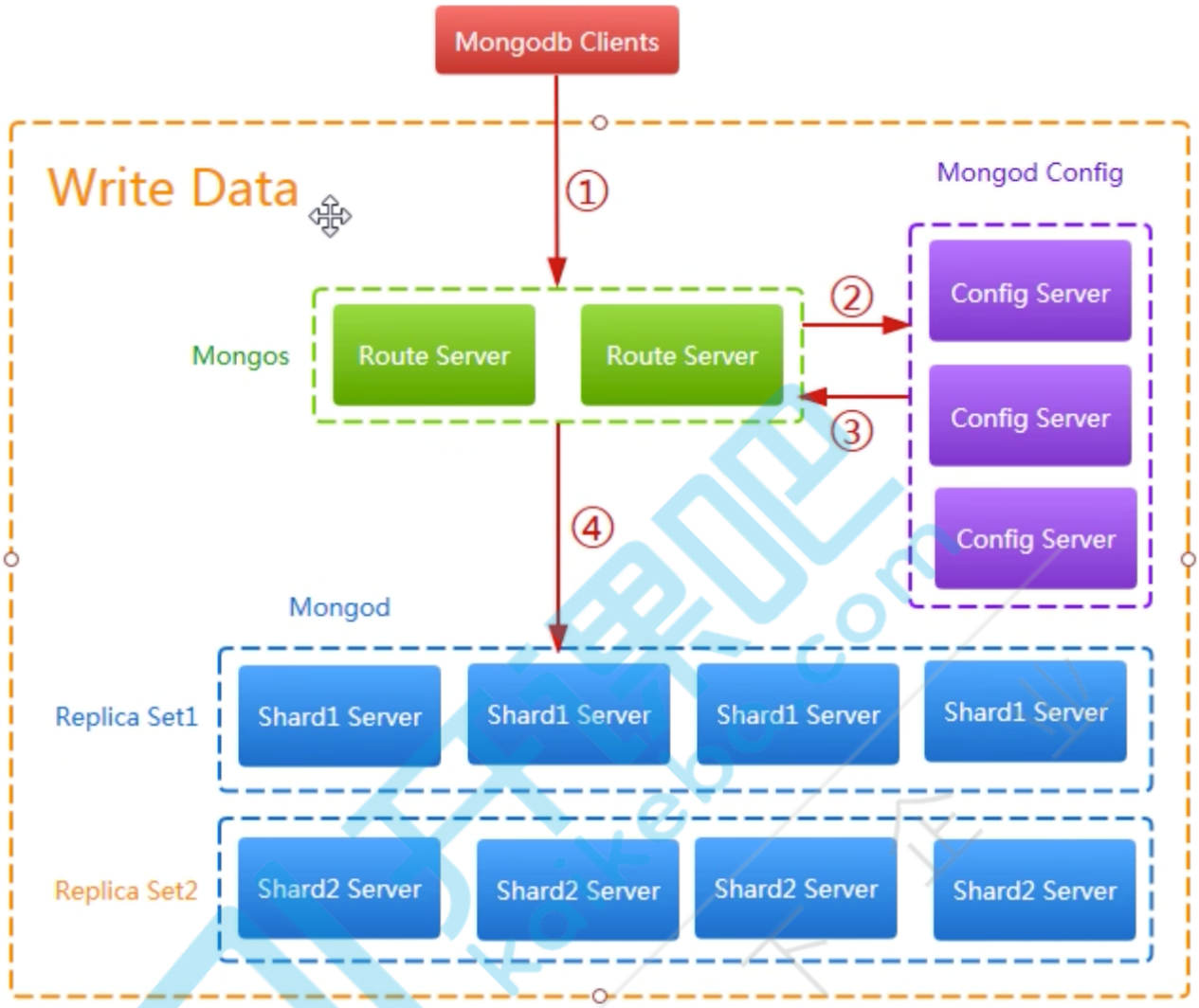


相同的副本集中的节点存储的数据是一样的，副本集中的节点是分为主节点、从节点、仲裁节点（非必须）三种角色。【这种设计方案的目的，主要是为了高性能、高可用、数据备份。】

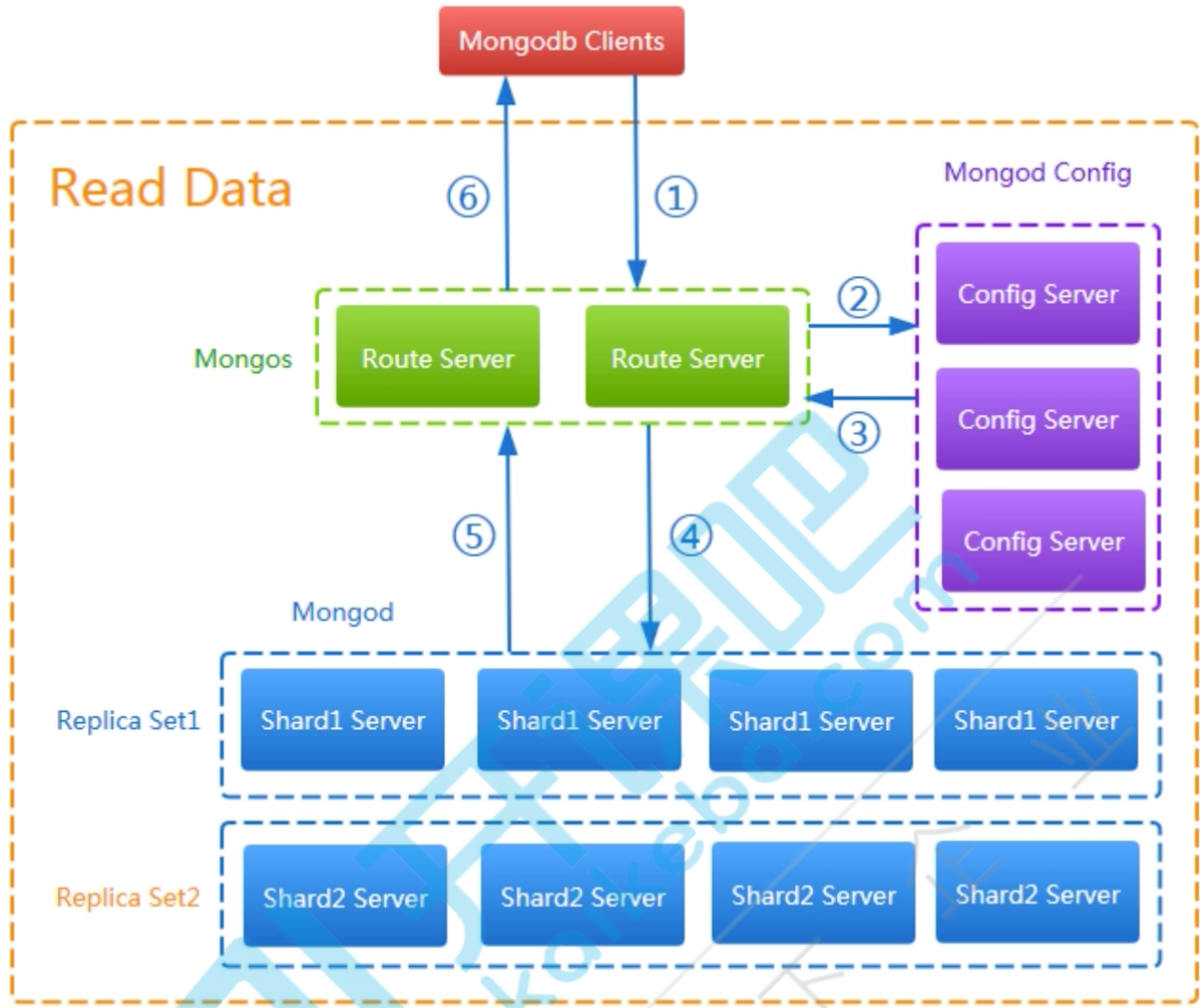
不同的副本集中的节点存储的数据是不一样，【这种设计方案，主要是为了解决高扩展问题，理论上是可以无限扩展的。】

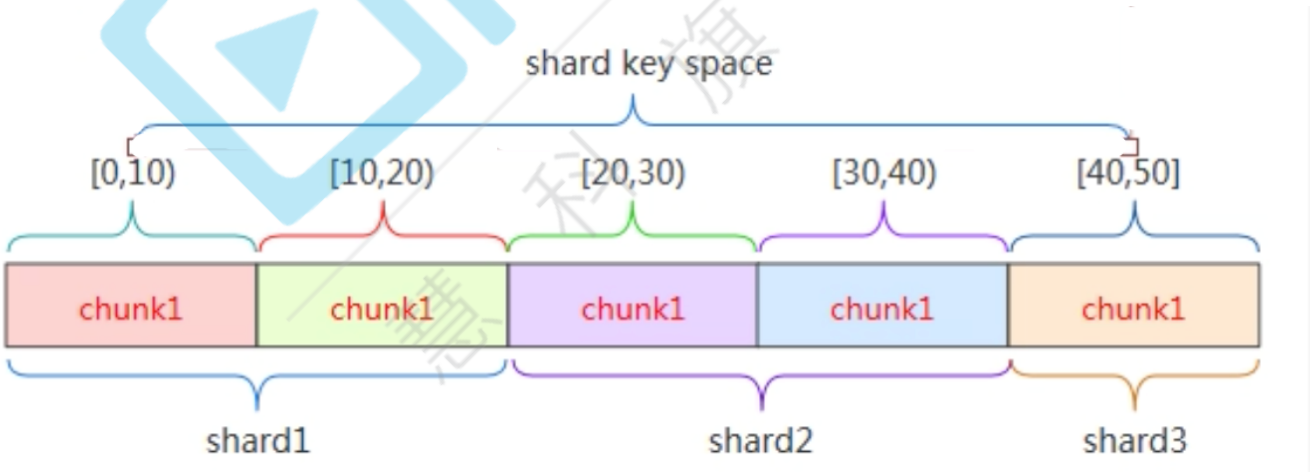
每一个副本集可以看成一个shard（分片），多个副本集共同组成一个逻辑上的大数据节点。通过对shard上面进行逻辑分块chunk（块），每个块都有自己存储的数据范围，所以说客户端请求存储数据的时候，会去读取config server中的映射信息，找到对应的chunk（块）存储数据。

混合部署方式下向MongoDB写数据的流程如图：



混合部署方式下读MongoDB里的数据流程如图： 按条件查询 查的就是片键 （建立索引）





MongoDB的应用场景和不适用场景

适用场景

更高的写入负载

默认情况下，MongoDB更侧重高数据写入性能，而非事务安全，MongoDB很适合业务系统中有大量"低价值"数据的场景。但是应当避免在高事务安全性的系统中使用MongoDB，除非能从架构设计上保证事务安全。

高可用性

MongoDB的复副集(Master-Slave)配置非常简洁方便，此外，MongoDB可以快速响应的处理单节点故障，自动、安全的完成故障转移。这些特性使得MongoDB能在一个相对不稳定（如云主机）的环境中，保持高可用性。

数据量很大或者未来会变得很大

依赖数据库(MySQL)自身的特性，完成数据的扩展是较困难的事，**在MySQL中，当一个单表达到5-10GB时会出现明显的性能降级**，此时需要通过数据的水平和垂直拆分、库的拆分完成扩展，使用MySQL通常需要借助驱动层或代理层完成这类需求。而MongoDB内建了多种数据分片的特性，可以很好的适应大数据量的需求。

基于位置的数据查询

MongoDB支持二维空间索引，因此可以快速及精确的从指定位置获取数据。

表结构不明确，且数据在不断变大

在一些传统RDBMS中，增加一个字段会锁住整个数据库/表，或者在执行一个重负载的请求时会明显造成其它请求的性能降级。通常发生在数据表大于1G的时候（当大于1TB时更甚）。 因MongoDB是文档型数据库，为非结构货的文档增加一个新字段是很快速的操作，并且不会影响到已有数据。另外一个好处当业务数据发生变化时，是将不在需要由DBA修改表结构。

没有DBA支持

如果没有专职的DBA，并且准备不使用标准的关系型思想（结构化、连接等）来处理数据，那么MongoDB将会是你的首选。MongoDB对于对像数据的存储非常方便，类可以直接序列化成JSON存储到MongoDB中。 但是需要先了解一些最佳实践，避免当数据变大后，由于文档设计问题而造成的性能缺陷。

不适用场景

在某些场景下，MongoDB作为一个非关系型数据库有其局限性。MongoDB不支持事务操作，所以需要用到事务的应用建议不用MongoDB，另外MongoDB目前不支持join操作，需要复杂查询的应用也不建议使用MongoDB。

MongoDB安装和常用命令

安装

MongoDB 提供了 linux 各发行版本 64 位的安装包，你可以在官网下载安装包。

下载地址：https://www.mongodb.com/download-center#community

下载完安装包，并解压 **tgz**（以下演示的是 64 位 Linux上的安装） 。

```
wget https://fastdl.mongodb.org/linux/mongodb-linux-x86_64-rhel70-3.6.11.tgz
tar -xf mongodb-linux-x86_64-rhel70-3.6.11.tgz
mv mongodb-linux-x86_64-rhel70-3.6.11 mongodb
cd mongodb/bin

# 一定注意要创建数据目录，默认路径/data/db
mkdir -p /data/db
```

配置环境变量（**可以不配置**）：

vim /etc/profile

```
export PATH=$JAVA_HOME/bin:/root/mongodb/bin:$PATH
```

source /etc/profile

启动

通过配置文件方式启动

```
mongod --config 配置文件路径
mongod -f 配置文件路径
```

说明：

-f是--config的缩写

配置文件，需要手动创建，参考以下配置即可。

创建mongodb.cfg配置文件

```
#数据库文件位置
dbpath=/root/mongodb/singleton/data
#日志文件位置
logpath=/root/mongodb/singleton/logs/mongodb.log
# 以追加方式写入日志
logappend=true
# 是否以守护进程方式运行
fork=true
#绑定客户端访问的ip 0.0.0.0 不绑定ip
bind_ip=192.168.10.135
# 默认27017
port=27017
```

创建数据和日志目录：

```
mkdir /root/mongodb/singleton/data -p
mkdir /root/mongodb/singleton/logs -p
```

通过配置文件方式启动：

```
mongod -f /root/mongodb/sinleton/mongodb.cfg
```

连接客户端

```
mongo 192.168.10.135:27017
```

常用客户端

- 自带命令行客户端
- Navicat for mongodb
- MongoVUE
- Studio 3T
- Robo 3T
- RockMongo

常用命令

创建数据库

语法

MongoDB 创建数据库的语法格式如下：

```
use DATABASE_NAME
```

如果数据库不存在，则创建数据库，否则切换到指定数据库。

实例

以下实例我们创建了数据库 kkb:

```
> use kkb
switched to db kkb
> db
kkb
>
```

如果你想查看所有数据库，可以使用 **show dbs** 命令：

```
> show dbs
admin   0.000GB
local   0.000GB
>
```

可以看到，我们刚创建的数据库 kkb 并不在数据库的列表中， 要显示它，我们需要向 kkb 数据库插入一些数据。

```
> db.mycollection.insert({"name":"开课吧"})
WriteResult({ "nInserted" : 1 })
> show dbs
local   0.078GB
kkb     0.078GB
test    0.078GB
>
```

删除数据库

语法

MongoDB 删除数据库的语法格式如下：

```
db.dropDatabase()
```

删除当前数据库，默认为 test，你可以使用 db 命令查看当前数据库名。

实例

以下实例我们删除了数据库 kkb。

首先，查看所有数据库：

```
> show dbs
local   0.078GB
kkb     0.078GB
test    0.078GB
```

接下来我们切换到数据库 kkb：

```
> use kkb
switched to db kkb
>
```

执行删除命令：

```
> db.dropDatabase()
{ "dropped" : "kkb", "ok" : 1 }
```

最后，我们再通过 show dbs 命令数据库是否删除成功：

```
> show dbs
local  0.078GB
test   0.078GB
>
```

创建集合

MongoDB 中使用 **createCollection()** 方法来创建集合。

语法格式：

```
db.createCollection(name, options)
```

参数说明：

- name: 要创建的集合名称
- options: 可选参数, 指定有关内存大小及索引的选项

options 可以是如下参数：

字段	类型	描述
capped	布尔	（可选）如果为 true，则创建固定集合。固定集合是指有着固定大小的集合，当达到最大值时，它会自动覆盖最早的文档。 **当该值为 true 时，必须指定 size 参数。**
autoIndexId	布尔	（可选）如为 true，自动在 _id 字段创建索引。默认为 false。
size	数值	（可选）为固定集合指定一个最大值（以字节计）。 **如果 capped 为 true，也需要指定该字段。**
max	数值	（可选）指定固定集合中包含文档的最大数量。

在插入文档时，MongoDB 首先检查固定集合的 size 字段，然后检查 max 字段。

实例

在 test 数据库中创建 kkb 集合：

```
> use test
switched to db test
> db.createCollection("mycollection")
{ "ok" : 1 }
>
```

如果要查看已有集合，可以使用 show collections 命令：

```
> show collections
mycollection
system.indexes
```

删除集合

集合删除语法格式如下：

```
db.collection_name.drop()
```

以下实例删除了 kkb 数据库中的集合 site：

```
> use kkb
switched to db kkb
> show tables
site
> db.site.drop()
true
> show tables
>
```

插入文档

MongoDB 使用 insert() 或 save() 方法向集合中插入文档，语法如下：

```
db.COLLECTION_NAME.insert(document)
```

以下文档可以存储在 MongoDB 的 kkb 数据库 的 mycollection 集合中：

```
>db.mycollection.insert({title: 'MongoDB 教程',
    description: 'MongoDB 是一个 Nosql 数据库',
    by: '开课吧',
    url: 'http://www.kaikeba.com',
    tags: ['mongodb', 'database', 'NoSQL'],
    likes: 100
})


>var document = {title: 'MongoDB 教程',
    description: 'MongoDB 是一个 Nosql 数据库',
    by: '开课吧',
    url: 'http://www.kaikeba.com',
    tags: ['mongodb', 'database', 'NoSQL'],
    likes: 100
}
>db.mycollection.insert(document)
```

删除文档

MongoDB remove()函数是用来移除集合中的数据。在执行remove()函数前先执行find()命令来判断执行的条件是否正确，这是一个比较好的习惯。

remove() 方法的基本语法格式如下所示：

```
db.collection.remove(
   <query>,
   <justOne>
)
```

如果你的 MongoDB 是 2.6 版本以后的，语法格式如下：

```
db.collection.remove(
   <query>,
   {
     justOne: <boolean>,
     writeConcern: <document>
   }
)
```

参数说明：

- **query**：（可选）删除的文档的条件。
- **justOne**：（可选）如果设为 true 或 1，则只删除一个文档，如果不设置该参数，或使用默认值 false，则删除所有匹配条件的文档。
- **writeConcern**：（可选）抛出异常的级别。

以下文档我们执行**两次插入**操作：

```
>db.mycollection.insert({title: 'MongoDB 教程',
    description: 'MongoDB 是一个 Nosql 数据库',
    by: '开课吧',
    url: 'http://www.kaikeba.com',
    tags: ['mongodb', 'database', 'NoSQL'],
    likes: 100
})
```

使用 find() 函数查询数据：

```
> db.mycollection.find()
{ "_id" : ObjectId("56066169ade2f21f36b03137"), "title" : "MongoDB 教程", "description" :
"MongoDB 是一个 Nosql 数据库", "by" : "开课吧", "url" : "http://www.kaikeba.com", "tags" : [
"mongodb", "database", "NoSQL" ], "likes" : 100 }
{ "_id" : ObjectId("5606616dade2f21f36b03138"), "title" : "MongoDB 教程", "description" :
"MongoDB 是一个 Nosql 数据库", "by" : "开课吧", "url" : "http://www.kaikeba.com", "tags" : [
"mongodb", "database", "NoSQL" ], "likes" : 100 }
```

接下来我们移除 title 为 'MongoDB 教程' 的文档：

```
 >db.mycollection.remove({'title':'MongoDB 教程'})
 WriteResult({ "nRemoved" : 2 })           # 删除了两条数据
 >db.mycollection.find()
 ……                                        # 没有数据
```

如果你只想删除第一条找到的记录可以设置 justOne 为 1，如下所示：

```
>db.COLLECTION_NAME.remove(DELETION_CRITERIA,1)
```

如果你想删除所有数据，可以使用以下方式（类似常规 SQL 的 truncate 命令）：

```
>db.mycollection.remove({})
>db.mycollection.find()
>
```

查询文档

MongoDB 查询数据的语法格式如下：

```
db.collection.find(query, projection)
```

- **query** ： 可选，使用查询操作符指定查询条件
- **projection** ： 可选，使用投影操作符指定返回的键。查询时返回文档中所有键值， 只需省略该参数即可（默认省略）。

如果你需要以易读的方式来读取数据，可以使用 pretty() 方法，语法格式如下：

```
>db.mycollection.find().pretty()
```

- pretty() 方法以格式化的方式来显示所有文档。

以下实例我们查询了集合 mycollection 中的数据：

```
> db.mycollection.find().pretty()
{
        "_id" : ObjectId("56063f17ade2f21f36b03133"),
        "title" : "MongoDB 教程",
        "description" : "MongoDB 是一个 Nosql 数据库",
        "by" : "开课吧",
        "url" : "http://www.kaikeba.com",
        "tags" : [
                "mongodb",
                "database",
                "NoSQL"
        ],
        "likes" : 100
}
```

limit与skip方法

limit() 方法

如果你需要在MongoDB中读取指定数量的数据记录，可以使用MongoDB的Limit方法，limit()方法接受一个数字参数，该参数指定从MongoDB中读取的记录条数。

语法

limit()方法基本语法如下所示：

```
>db.COLLECTION_NAME.find().limit(NUMBER)
```

实例

集合 mycollection 中的数据如下：

```
{ "_id" : ObjectId("56066542ade2f21f36b0313a"), "title" : "PHP 教程", "description" : "PHP 是一种创建动态交互性站点的强有力的服务器端脚本语言。", "by" : "开课吧", "url" : "http://www.kaikeba.com", "tags" : [ "php" ], "likes" : 200 }
{ "_id" : ObjectId("56066549ade2f21f36b0313b"), "title" : "Java 教程", "description" : "Java 是由Sun Microsystems公司于1995年5月推出的高级程序设计语言。", "by" : "开课吧", "url" : "http://www.kaikeba.com", "tags" : [ "java" ], "likes" : 150 }
{ "_id" : ObjectId("5606654fade2f21f36b0313c"), "title" : "MongoDB 教程", "description" : "MongoDB 是一个 Nosql 数据库", "by" : "开课吧", "url" : "http://www.kaikeba.com", "tags" : [ "mongodb" ], "likes" : 100 }
```

以下实例为显示查询文档中的两条记录：

```
> db.col.find({},{"title":1,_id:0}).limit(2)
{ "title" : "PHP 教程" }
{ "title" : "Java 教程" }
>
```

注：如果你们没有指定limit()方法中的参数则显示集合中的所有数据。

---

skip() 方法

我们除了可以使用limit()方法来读取指定数量的数据外，还可以使用skip()方法来跳过指定数量的数据，skip方法同样接受一个数字参数作为跳过的记录条数。

语法

skip() 方法脚本语法格式如下：

```
>db.COLLECTION_NAME.find().limit(NUMBER).skip(NUMBER)
```

实例

以下实例只会显示第二条文档数据

```
>db.col.find({},{"title":1,_id:0}).limit(1).skip(1)
{ "title" : "Java 教程" }
>
```

注:skip()方法默认参数为 0 。

文档排序

在 MongoDB 中使用 sort() 方法对数据（文档）进行排序，sort() 方法可以通过参数指定排序的字段，并使用 1 和 -1 来指定排序的方式，**其中 1 为升序排列，而 -1 是用于降序排列。**

sort()方法基本语法如下所示：

```
>db.COLLECTION_NAME.find().sort({KEY:1})
```

mycollection集合中的数据如下：

```
{ "_id" : ObjectId("56066542ade2f21f36b0313a"), "title" : "PHP 教程", "description" : "PHP 是一种创建动态交互性站点的强有力的服务器端脚本语言。", "by" : "开课吧", "url" : "http://www.kaikeba.com", "tags" : [ "php" ], "likes" : 200 }
{ "_id" : ObjectId("56066549ade2f21f36b0313b"), "title" : "Java 教程", "description" : "Java是由Sun Microsystems公司于1995年5月推出的高级程序设计语言。", "by" : "开课吧", "url" : "http://www.kaikeba.com", "tags" : [ "java" ], "likes" : 150 }
{ "_id" : ObjectId("5606654fade2f21f36b0313c"), "title" : "MongoDB 教程", "description" : "MongoDB 是一个 Nosql 数据库", "by" : "开课吧", "url" : "http://www.kaikeba.com", "tags" : [ "mongodb" ], "likes" : 100 }
```

以下实例演示了 mycollection 集合中的数据按字段 likes 的降序排列：

```
>db.mycollection.find({},{"title":1,_id:0}).sort({"likes":-1})
{ "title" : "PHP 教程" }
{ "title" : "Java 教程" }
{ "title" : "MongoDB 教程" }
>
```

MongoDB 索引

索引通常能够极大的提高查询的效率，如果没有索引，MongoDB在读取数据时必须扫描集合中的每个文件并选取那些符合查询条件的记录。

这种扫描全集合的查询效率是非常低的，特别在处理大量的数据时，查询可以要花费几十秒甚至几分钟，这对网站的性能是非常致命的。

索引是特殊的数据结构，索引存储在一个易于遍历读取的数据集合中，索引是对数据库表中一列或多列的值进行排序的一种结构

创建索引

MongoDB使用 createIndex() 方法来创建索引。

> 注意在 3.0.0 版本前创建索引方法为 db.collection.ensureIndex()，之后的版本使用了 db.collection.createIndex() 方法，ensureIndex() 还能用，但只是 createIndex() 的别名。

语法

createIndex()方法基本语法格式如下所示：

```
>db.collection.createIndex(keys, options)
```

语法中 Key 值为你要创建的索引字段，1 为指定按升序创建索引，如果你想按降序来创建索引指定为 -1 即可。

实例1

```
>db.col.createIndex({"title":1})
>
```

createIndex() 方法中你也可以设置使用多个字段创建索引（关系型数据库中称作复合索引）。

```
>db.col.createIndex({"title":1,"description":-1})
>
```

createIndex() 接收可选参数，可选参数列表如下：

Parameter	Type	Description
background	Boolean	建索引过程会阻塞其它数据库操作，background可指定以后台方式创建索引，即增加 "background" 可选参数。 "background" 默认值为**false**。
unique	Boolean	建立的索引是否唯一。指定为true创建唯一索引。默认值为**false**.
name	string	索引的名称。如果未指定，MongoDB的通过连接索引的字段名和排序顺序生成一个索引名称。
dropDups	Boolean	**3.0+版本已废弃。**在建立唯一索引时是否删除重复记录,指定 true 创建唯一索引。默认值为 **false**.
sparse	Boolean	对文档中不存在的字段数据不启用索引；这个参数需要特别注意，如果设置为true的话，在索引字段中不会查询出不包含对应字段的文档.。默认值为 **false**.
expireAfterSeconds	integer	指定一个以秒为单位的数值，完成 TTL设定，设定集合的生存时间。
v	index version	索引的版本号。默认的索引版本取决于mongod创建索引时运行的版本。
weights	document	索引权重值，数值在 1 到 99,999 之间，表示该索引相对于其他索引字段的得分权重。
default_language	string	对于文本索引，该参数决定了停用词及词干和词器的规则的列表。 默认为英语
language_override	string	对于文本索引，该参数指定了包含在文档中的字段名，语言覆盖默认的language，默认值为 language.

实例2

在后台创建索引：

```
db.values.createIndex({open: 1, close: 1}, {background: true})
```

通过在创建索引时加 background:true 的选项，让创建工作在后台执行

查看集合索引

```
db.col.getIndexes()
```

查看集合索引大小

```
db.col.totalIndexSize()
```

删除集合所有索引

```
db.col.dropIndexes()
```

删除集合指定索引

```
db.col.dropIndex("索引名称")
```

聚合查询

MongoDB中聚合(aggregate)主要用于处理数据(诸如统计平均值,求和等)，并返回计算后的数据结果。有点类似sql语句中的 count(*)。利用Aggregate聚合管道可以完成。MongoDB的聚合管道将MongoDB文档在一个管道处理完毕后将结果传递给下一个管道处理。管道操作是可以重复的。

表达式：处理输入文档并输出。表达式是无状态的，只能用于计算当前聚合管道的文档，不能处理其它的文档。

基本语法为：

```
>db.COLLECTION_NAME.aggregate(AGGREGATE_OPERATION)
```

为了便于理解，先将常见的mongo的聚合操作和mysql的查询做下类比：

表达式	SQL操作	描述
$group	group by	分组
$sum	count()、sum()	计算总和。
$avg	avg()	计算平均值
$min	min()	获取集合中所有文档对应值得最小值。
$max	max()	获取集合中所有文档对应值得最大值。
$match	where、having	查询条件
$sort	order by	排序
$limit	limit	取条数
$project	select	选择
$lookup （v3.2 新增）	join	连接

下面以一个案例来演示：

集合中的数据如下

```
> db.LIST1.aggregate([])
{ "_id" : ObjectId("5cceaf55e7a4cf28dc5854cf"), "name" : "zhaoyun1", "age" : 1, "city" :
"BJ" }
{ "_id" : ObjectId("5cceaf55e7a4cf28dc5854d0"), "name" : "zhaoyun2", "age" : 2, "city" :
"BJ" }
{ "_id" : ObjectId("5cceaf55e7a4cf28dc5854d1"), "name" : "zhaoyun3", "age" : 3, "city" :
"BJ" }
{ "_id" : ObjectId("5cceaf55e7a4cf28dc5854d2"), "name" : "zhaoyun4", "age" : 4, "city" :
"BJ" }
{ "_id" : ObjectId("5cceaf55e7a4cf28dc5854d3"), "name" : "zhaoyun5", "age" : 5, "city" :
"BJ" }
{ "_id" : ObjectId("5cceaf55e7a4cf28dc5854d4"), "name" : "zhaoyun6", "age" : 6, "city" :
"BJ" }
{ "_id" : ObjectId("5cceaf55e7a4cf28dc5854d5"), "name" : "zhaoyun7", "age" : 7, "city" :
"TJ" }
{ "_id" : ObjectId("5cceaf55e7a4cf28dc5854d6"), "name" : "zhaoyun8", "age" : 8, "city" :
"TJ" }
{ "_id" : ObjectId("5cceaf55e7a4cf28dc5854d7"), "name" : "zhaoyun9", "age" : 9, "city" :
"TJ" }
{ "_id" : ObjectId("5cceaf55e7a4cf28dc5854d8"), "name" : "zhaoyun10", "age" : 10, "city" :
"TJ"}
```

$group

按照城市分组对年龄进行求和

```
> db.LIST1.aggregate([{$group : {_id : "$city", snum : {$sum : "$age"}}}])
{ "_id" : "TJ", "snum" : 34 }
{ "_id" : "BJ", "snum" : 21 }
```

$sum

按照城市分组对年龄进行求和

```
> db.LIST1.aggregate([{$group : {_id : "$city", snum : {$sum : "$age"}}}])
{ "_id" : "TJ", "snum" : 34 }
{ "_id" : "BJ", "snum" : 21 }
```

按照城市分组求总个数

```
> db.LIST1.aggregate([{$group : {_id : "$city", sc : {$sum : 1}}}])
{ "_id" : "TJ", "sc" : 4 }
{ "_id" : "BJ", "sc" : 6 }
```

$avg

按照城市分组对年龄进行求平均

```
>  db.LIST1.aggregate([{$group : {_id : "$city", avg : {$avg : "$age"}}}])
{ "_id" : "TJ", "avg" : 8.5 }
{ "_id" : "BJ", "avg" : 3.5 }
```

$min

按照城市分组获得每组最小年龄

```
> db.LIST1.aggregate([{$group : {_id : "$city", min : {$min : "$age"}}}])
{ "_id" : "TJ", "min" : 7 }
{ "_id" : "BJ", "min" : 1 }
```

$max

按照城市分组获得每组最大年龄

```
> db.LIST1.aggregate([{$group : {_id : "$city", max : {$max : "$age"}}}])
{ "_id" : "TJ", "max" : 10 }
{ "_id" : "BJ", "max" : 6 }
```

$match

取年龄大于3并且小于10的记录再按照城市分组求个数和

```
> db.LIST1.aggregate( [{ $match : { age : { $gt : 3, $lte : 10 } } },{ $group: { _id:
"$city",count: { $sum: 1 } } }] );
{ "_id" : "TJ", "count" : 4 }
{ "_id" : "BJ", "count" : 3 }
```

$sort

按照年龄的倒序显示

```
> db.LIST1.aggregate( [{ $sort:{ age :-1}}] )
{ "_id" : ObjectId("5cceaf55e7a4cf28dc5854d8"), "name" : "zhaoyun10", "age" : 10, "city" :
"TJ"}
{ "_id" : ObjectId("5cceaf55e7a4cf28dc5854d7"), "name" : "zhaoyun9", "age" : 9, "city" :
"TJ" }
{ "_id" : ObjectId("5cceaf55e7a4cf28dc5854d6"), "name" : "zhaoyun8", "age" : 8, "city" :
"TJ" }
{ "_id" : ObjectId("5cceaf55e7a4cf28dc5854d5"), "name" : "zhaoyun7", "age" : 7, "city" :
"TJ" }
{ "_id" : ObjectId("5cceaf55e7a4cf28dc5854d4"), "name" : "zhaoyun6", "age" : 6, "city" :
"BJ" }
{ "_id" : ObjectId("5cceaf55e7a4cf28dc5854d3"), "name" : "zhaoyun5", "age" : 5, "city" :
"BJ" }
{ "_id" : ObjectId("5cceaf55e7a4cf28dc5854d2"), "name" : "zhaoyun4", "age" : 4, "city" :
"BJ" }
{ "_id" : ObjectId("5cceaf55e7a4cf28dc5854d1"), "name" : "zhaoyun3", "age" : 3, "city" :
"BJ" }
{ "_id" : ObjectId("5cceaf55e7a4cf28dc5854d0"), "name" : "zhaoyun2", "age" : 2, "city" :
"BJ" }
{ "_id" : ObjectId("5cceaf55e7a4cf28dc5854cf"), "name" : "zhaoyun1", "age" : 1, "city" :
"BJ" }
```

$limit

取得前5条数据

```
> db.LIST1.aggregate( { $limit:5} )
{ "_id" : ObjectId("5cceaf55e7a4cf28dc5854cf"), "name" : "zhaoyun1", "age" : 1, "city" :
"BJ" }
{ "_id" : ObjectId("5cceaf55e7a4cf28dc5854d0"), "name" : "zhaoyun2", "age" : 2, "city" :
"BJ" }
{ "_id" : ObjectId("5cceaf55e7a4cf28dc5854d1"), "name" : "zhaoyun3", "age" : 3, "city" :
"BJ" }
{ "_id" : ObjectId("5cceaf55e7a4cf28dc5854d2"), "name" : "zhaoyun4", "age" : 4, "city" :
"BJ" }
{ "_id" : ObjectId("5cceaf55e7a4cf28dc5854d3"), "name" : "zhaoyun5", "age" : 5, "city" :
"BJ" }
```

$project

不显示_id字段 显示 name和city字段

```
> db.LIST1.aggregate( [ {$project :{_id:0,name:1,city:1}   }  ] )
{ "name" : "zhaoyun1", "city" : "BJ" }
{ "name" : "zhaoyun2", "city" : "BJ" }
{ "name" : "zhaoyun3", "city" : "BJ" }
{ "name" : "zhaoyun4", "city" : "BJ" }
{ "name" : "zhaoyun5", "city" : "BJ" }
{ "name" : "zhaoyun6", "city" : "BJ" }
{ "name" : "zhaoyun7", "city" : "TJ" }
{ "name" : "zhaoyun8", "city" : "TJ" }
{ "name" : "zhaoyun9", "city" : "TJ" }
{ "name" : "zhaoyun10", "city" : "TJ" }
```

$lookup

用于多文档关联，类似于SQL的多表关联

建立新的集合

```
> db.sex.find()
{ "_id" : 1, "name" : "male" }
{ "_id" : 2, "name" : "female" }
```

给LIST1新增字段sex

```
> db.LIST1.update({},{$set:{sex:1}},{multi:1})
WriteResult({ "nMatched" : 10, "nUpserted" : 0, "nModified" : 10 })
> db.LIST1.find()
{ "_id" : ObjectId("5cceaf55e7a4cf28dc5854cf"), "name" : "zhaoyun1", "age" : 1, "city" :
"BJ", "sex" : 1 }
{ "_id" : ObjectId("5cceaf55e7a4cf28dc5854d0"), "name" : "zhaoyun2", "age" : 2, "city" :
"BJ", "sex" : 1 }
{ "_id" : ObjectId("5cceaf55e7a4cf28dc5854d1"), "name" : "zhaoyun3", "age" : 3, "city" :
"BJ", "sex" : 1 }
{ "_id" : ObjectId("5cceaf55e7a4cf28dc5854d2"), "name" : "zhaoyun4", "age" : 4, "city" :
"BJ", "sex" : 1 }
{ "_id" : ObjectId("5cceaf55e7a4cf28dc5854d3"), "name" : "zhaoyun5", "age" : 5, "city" :
"BJ", "sex" : 1 }
{ "_id" : ObjectId("5cceaf55e7a4cf28dc5854d4"), "name" : "zhaoyun6", "age" : 6, "city" :
"BJ", "sex" : 1 }
{ "_id" : ObjectId("5cceaf55e7a4cf28dc5854d5"), "name" : "zhaoyun7", "age" : 7, "city" :
"TJ", "sex" : 1 }
{ "_id" : ObjectId("5cceaf55e7a4cf28dc5854d6"), "name" : "zhaoyun8", "age" : 8, "city" :
"TJ", "sex" : 1 }
{ "_id" : ObjectId("5cceaf55e7a4cf28dc5854d7"), "name" : "zhaoyun9", "age" : 9, "city" :
"TJ", "sex" : 1 }
{ "_id" : ObjectId("5cceaf55e7a4cf28dc5854d8"), "name" : "zhaoyun10", "age" : 10, "city" :
"TJ", "sex" : 1 }
```

将两个文档关联

```
> db.LIST1.aggregate([{$lookup:{from:"sex",localField:"sex",foreignField:"_id",as:"docs"}},
{$project:{"_id":0}}])
{ "name" : "zhaoyun1", "age" : 1, "city" : "BJ", "sex" : 1, "docs" : [ { "_id" : 1, "name"
: "male" } ] }
{ "name" : "zhaoyun2", "age" : 2, "city" : "BJ", "sex" : 1, "docs" : [ { "_id" : 1, "name"
: "male" } ] }
{ "name" : "zhaoyun3", "age" : 3, "city" : "BJ", "sex" : 1, "docs" : [ { "_id" : 1, "name"
: "male" } ] }
{ "name" : "zhaoyun4", "age" : 4, "city" : "BJ", "sex" : 1, "docs" : [ { "_id" : 1, "name"
: "male" } ] }
{ "name" : "zhaoyun5", "age" : 5, "city" : "BJ", "sex" : 1, "docs" : [ { "_id" : 1, "name"
: "male" } ] }
{ "name" : "zhaoyun6", "age" : 6, "city" : "BJ", "sex" : 1, "docs" : [ { "_id" : 1, "name"
: "male" } ] }
{ "name" : "zhaoyun7", "age" : 7, "city" : "TJ", "sex" : 1, "docs" : [ { "_id" : 1, "name"
: "male" } ] }
{ "name" : "zhaoyun8", "age" : 8, "city" : "TJ", "sex" : 1, "docs" : [ { "_id" : 1, "name"
: "male" } ] }
{ "name" : "zhaoyun9", "age" : 9, "city" : "TJ", "sex" : 1, "docs" : [ { "_id" : 1, "name"
: "male" } ] }
{ "name" : "zhaoyun10", "age" : 10, "city" : "TJ", "sex" : 1, "docs" : [ { "_id" : 1,
"name" : "male" } ] }
```

LIST1中的数据字段sex关联sex中的数据字段_id，在LIST1中显示sex对应的中文，显示在docs中，不显示LIST1中的__id

下面简单介绍一些$lookup中的参数：

from：需要关联的表【sex】

localField:【LIST1】表需要关联的键。

foreignField：【sex】的matching key 主键

as：对应的外键集合的数据，【因为是一对多的】

MongoDB Java客户端

- marphia：基于mongodb的ORM框架
- spring-data-mongodb：spring提供的mongodb客户端
- mongodb-java-driver：官方驱动包

mongodb-java-API

POM依赖

```
<dependency>
            <groupId>org.mongodb</groupId>
            <artifactId>mongo-java-driver</artifactId>
            <version>3.10.1</version>
        </dependency>
```

测试代码

```
public class MongodbDemo {

    private MongoClient client;

    @Before
    public void init() {
        client = new MongoClient("192.168.10.135",27017);
    }

    @Test
    public void connectDB() {
        // 连接数据库，你需要指定数据库名称，如果指定的数据库不存在，mongo会自动创建数据库。
        MongoDatabase database = client.getDatabase("kkb");
        System.out.println("connect successful----"+database.getName());
    }
    @Test
    public void createCollection() {
        MongoDatabase database = client.getDatabase("kkb");
        database.createCollection("mycollection");
        System.out.println("集合创建成功");
    }
    @Test
    public void getCollection() {
        MongoDatabase database = client.getDatabase("kkb");
        MongoCollection<Document> collection = database.getCollection("mycollection");
        System.out.println("获取集合成功: "+collection.getNamespace());
        MongoIterable<String> collectionNames = database.listCollectionNames();
        for (String string : collectionNames) {
            System.out.println("collection name : "+ string);
        }
    }
    @Test
    public void insertDocument() {
        MongoDatabase database = client.getDatabase("kkb");
        MongoCollection<Document> collection = database.getCollection("mycollection");
        Document document = new Document("name", "James");
        document.append("age", 34);
        document.append("sex", "男");
        Document document2 = new Document("name", "wade");
        document2.append("age", 36);
        document2.append("sex", "男");
        Document document3 = new Document("name", "cp3");
```

```
        document3.append("age", 32);
        document3.append("sex", "男");

        List<Document> documents = new ArrayList<>();
        documents.add(document);
        documents.add(document2);
        documents.add(document3);

        collection.insertMany(documents);
        System.out.println("文档插入成功");
    }
    @Test
    public void findDocuments() {
        MongoDatabase database = client.getDatabase("kkb");
        MongoCollection<Document> collection = database.getCollection("mycollection");

        FindIterable<Document> iterable = collection.find();
        for (Document document : iterable) {
            System.out.println(document.toJson());
        }

//      MongoCursor<Document> mongoCursor = iterable.iterator();
//      while (mongoCursor.hasNext()) {
//          Document document = mongoCursor.next();
//          System.out.println(document.toJson());
//      }
    }


    @Test
    public void updateDocuments() {
        MongoDatabase database = client.getDatabase("kkb");
        MongoCollection<Document> collection = database.getCollection("mycollection");

        collection.updateMany(Filters.eq("age",18), new Document("$set", new
Document("age", 20)));

        FindIterable<Document> iterable = collection.find();
        for (Document document : iterable) {
            System.out.println(document.toJson());
        }

    }
    @Test
    public void deleteDocuments() {
        MongoDatabase database = client.getDatabase("kkb");
        MongoCollection<Document> collection = database.getCollection("mycollection");

        // 删除符合条件的第一个文档
        collection.deleteOne(Filters.eq("age",20));
        collection.deleteOne(Filters.eq("name","James"));
        // 删除符合条件的所有文档
        collection.deleteMany(Filters.eq("age",20));
```

```
        FindIterable<Document> iterable = collection.find();
        for (Document document : iterable) {
            System.out.println(document.toJson());
        }

    }
    /*
     * 查询
     */
    @Test
    public void testQuery1() {

        MongoCollection<Document> collection = db.getCollection("LIST1");

        //获取第一条记录
        Document first = collection.find().first();

        //根据key值获得数据
        System.out.println(first.get("name"));

        //查询指定字段
        Document qdoc=new Document();
        qdoc.append("_id", 0); //0是不包含字段
        qdoc.append("name", 1);//1是包含字段

        FindIterable<Document> findIterable = collection.find()
                .projection(qdoc);
        for (Document document : findIterable) {
            System.out.println(document.toJson());
        }

    }
    @Test
    public void testQuery2() {

        MongoCollection<Document> collection = db.getCollection("LIST1");

        //按指定字段排序
        Document sdoc=new Document();
        //按年龄的倒序
        sdoc.append("age", -1);
        FindIterable<Document> findIterable = collection.find()
                .sort(sdoc);
        for (Document document : findIterable) {
            System.out.println(document.toJson());
        }

    }

}
```

Filters

利用Filters可以进行条件过滤，需要引入com.mongodb.client.model.Filters类，比如：eq、ne、gt、lt、gte、lte、in、nin、and、or、not、nor、exists、type、mod、regex、text、where、all、elemMatch、size、bitsAllClear、bitsAllSet、bitsAnyClear、bitsAnySet、geoWithin、geoWithin、geoWithinBox、geoWithinPolygon、geoWithinCenter、geoWithinCenterSphere、geoIntersects、near、nearSphere 等。

主要介绍几个常用的过滤操作。

```
@Test
public void testFilters1() {
        MongoCollection<Document> collection = db.getCollection("LIST1");
        //获得name等于zhaoyun1的记录
        FindIterable<Document> findIterable =collection.find(Filters.eq("name",
"zhaoyun1"));
        for (Document document : findIterable) {
            System.out.println(document.toJson());
        }
}

@Test
public void testFilters2() {
        MongoCollection<Document> collection = db.getCollection("LIST1");
        //获得age大于3的记录
        FindIterable<Document> findIterable =collection.find(Filters.gt("age", 3));
        for (Document document : findIterable) {
            System.out.println(document.toJson());
        }
}
@Test
    public void testFilters3() {
        MongoCollection<Document> collection = db.getCollection("LIST1");
        //获得age大于3的记录 或者 name=zhaoyun1
        FindIterable<Document> findIterable =collection.find(Filters.or(Filters.eq("name",
"zhaoyun1"),Filters.gt("age", 3)));
        for (Document document : findIterable) {
            System.out.println(document.toJson());
        }

    }
```

BasicDBObject

利用BasicDBObject可以实现模糊查询和分页查询

```
@Test
public void testLike() {
        MongoCollection<Document> collection = db.getCollection("LIST1");
         //利用正则 模糊匹配 包含zhaoyun1
        Pattern pattern = Pattern.compile("^.*zhaoyun1.*$", Pattern.CASE_INSENSITIVE);
        BasicDBObject query = new BasicDBObject();
        //name包含zhaoyun1
        query.put("name",pattern);
        FindIterable<Document> findIterable = collection
```

```
                .find(query);
        for (Document document : findIterable) {
            System.out.println(document.toJson());
        }
}

@Test
public void testPage() {
        MongoCollection<Document> collection = db.getCollection("LIST1");
         //利用正则 模糊匹配 name包含zhaoyun1
        Pattern pattern = Pattern.compile("^.*zhaoyun.*$", Pattern.CASE_INSENSITIVE);
        //查询条件
        BasicDBObject query = new BasicDBObject();
        query.put("name",pattern);
        //排序
        BasicDBObject sort = new BasicDBObject();
        //按年龄倒序
        sort.put("age", -1);
        //获得总条数
        System.out.println("共计: "+collection.count()+"条记录");
        /*
         * 取出第1至3条记录
         * query : 查询条件
         * sort: 排序
         * skip : 起始数 从0开始
         * limit:取几条
         */
        FindIterable<Document> findIterable = collection
                .find(query).sort(sort).skip(0).limit(3);
        for (Document document : findIterable) {
            System.out.println(document.toJson());
        }
    }
```

spring-data-mongod

pom依赖

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.kkb</groupId>
    <artifactId>mongodb-demo</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <dependencies>
        <dependency>
            <groupId>org.mongodb</groupId>
```

```
            <artifactId>mongo-java-driver</artifactId>
            <version>3.10.1</version>
        </dependency>

        <dependency>
            <groupId>org.springframework.data</groupId>
            <artifactId>spring-data-mongodb</artifactId>
            <version>2.1.1.RELEASE</version>
        </dependency>

        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-test</artifactId>
            <version>5.0.7.RELEASE</version>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.8.0</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                    <encoding>UTF-8</encoding>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

spring配置文件

```
<?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/data/mongo"
    xmlns:repository="http://www.springframework.org/schema/data/repository"
    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
```

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

    <!-- 加载mongodb的配置属性文件 -->
    <context:property-placeholder location="classpath:mongodb.properties" />

    <!-- 扫描持久层 -->
    <context:component-scan base-package="com.kkb.mongodb.springdata" />

    <!-- 配置mongodb客户端连接服务器的相关信息 -->
    <mongo:mongo-client host="${mongo.host}" port="${mongo.port}" id="mongo">
        <mongo:client-options
            write-concern="${mongo.writeconcern}"
            connect-timeout="${mongo.connectTimeout}"
            socket-keep-alive="${mongo.socketKeepAlive}" />
    </mongo:mongo-client>

    <!-- mongo:db-factory dbname="database" mongo-ref="mongo" / -->
    <!--这里的dbname就是自己的数据库/collection的名字 -->
    <mongo:db-factory id="mongoDbFactory" dbname="kkb" mongo-ref="mongo" />

    <!-- 配置mongodb的模板类: MongoTemplate -->
    <bean id="mongoTemplate" class="org.springframework.data.mongodb.core.MongoTemplate">
        <constructor-arg name="mongoDbFactory" ref="mongoDbFactory" />
    </bean>

</beans>
```

perperties文件

```
mongo.host=192.168.10.135
mongo.port=27017
mongo.connectionsPerHost=8
mongo.threadsAllowedToBlockForConnectionMultiplier=4
#连接超时时间
mongo.connectTimeout=1000
#等待时间
mongo.maxWaitTime=1500
mongo.autoConnectRetry=true
mongo.socketKeepAlive=true
#socket超时时间
mongo.socketTimeout=1500
mongo.slaveOK=true
mongo.writeconcern=safe
```

po类

```
public class User {
    private String name;
    private int age;
    private String sex;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public String getSex() {
        return sex;
    }

    public void setSex(String sex) {
        this.sex = sex;
    }

    @Override
    public String toString() {
        return "NBAPlayer [name=" + name + ", age=" + age + ", sex=" + sex + "]";
    }

}
```

dao接口和实现类

```
public interface UserDao {
    /**
     * 根据条件查询
     */
    public List<User> find(Query query);

    /**
     * 根据条件查询一个
     */
    public User findOne(Query query);

    /**
     * 插入
     */
```

```
    public void save(User entity);

    /**
     * 根据条件 更新
     */
    public UpdateResult update(Query query, Update update);

    /**
     * 获得所有该类型记录,并且指定了集合名(表的意思)
     */
    public List<User> findAll(String collectionName);

    /**
     * 根据条件 获得总数
     */
    public long count(Query query,String collectionName);

    /**
     * 根据条件 删除
     *
     * @param query
     */
    public void remove(Query query);
}
```

```
@Repository
public class UserDaoImpl implements UserDao {

    @Resource
    private MongoTemplate template;

    @Override
    public List<User> find(Query query) {
        return template.find(query, User.class);
    }

    @Override
    public User findOne(Query query) {
        return template.findOne(query, User.class);
    }

    @Override
    public void save(User entity) {
        template.save(entity);
    }

    @Override
    public UpdateResult update(Query query, Update update) {
        return template.updateMulti(query, update, User.class);
    }
```

```
    @Override
    public List<User> findAll(String collectionName) {
        return template.findAll(User.class, collectionName);
    }

    @Override
    public long count(Query query,String collectionName) {
        return template.count(query, collectionName);
    }

    @Override
    public void remove(Query query) {
        template.remove(query,"user");
    }

}
```

测试类

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = "classpath:spring-mongodb.xml")
public class SpringDataMongodbTest {

    @Resource
    private UserDao userDao;

    @Test
    public void test1() {
        User user = new User();
        user.setName("科比");
        user.setAge(26);

        userDao.save(user);

        User user2 = new User();
        user2.setName("詹姆斯");
        user2.setAge(33);

        userDao.save(user2);

        User user3 = new User();
        user3.setName("韦德");
        user3.setAge(36);

        userDao.save(user3);

        User user4 = new User();
        user4.setName("罗斯");
        user4.setAge(30);

        userDao.save(user4);
```

```
        System.out.println("插入成功! ");
    }

    @Test
    public void test2() {
        Query query = new Query(Criteria.where("name").is("科比"));
        List<User> list = userDao.find(query);
        for (User user : list) {
            System.out.println(user);
        }
        System.out.println("======================");
        List<User> list2 = userDao.find(new Query());
        for (User user : list2) {
            System.out.println(user);
        }

        System.out.println("======================");
        User user = (User) userDao.findOne(query);
        System.out.println(user);
    }

    @Test
    public void test3() {
        Query query = new Query(Criteria.where("name").is("韦德"));
        Update update = new Update();
        update.set("sex", "纯爷们");
        UpdateResult updateResult = userDao.update(query, update);
        System.out.println(updateResult.getUpsertedId());
    }

    @Test
    public void test4() {
        List<User> findAll = userDao.findAll("user");
        for (User user : findAll) {
            System.out.println(user);
        }
    }

    @Test
    public void test5() {
        Query query = new Query();
        long count = userDao.count(query,"user");
        System.out.println("总条数: " + count);
    }

    @Test
    public void test6() {
        Query query = new Query(Criteria.where("name").is("科比"));
        userDao.remove(query);
    }

}
```

开课吧
kaikeba.com
慧科旗下企业