《MongoDB 最佳实践 – 持续更新版》

前言

作为MongoDB的一名方案架构师，我的大部分时间都是在和MongoDB的客户和用户交互。在这里，我希望通过一个不断更新的活文章的方式来为大家收集整理一下MongoDB开发及维护时候值得了解或者遵从的一些最佳实践。我非常真切地希望您也可以参与进来，共同维护这个文档，让更多的用户受惠(可以通过文末微信号联系我)

本文包括以下几个方面：

如果您正在打算上线或者已经上线基于MongoDB的核心业务应用，请和MongoDB 大中华区团队联络(team-china@mongodb.com)以获得来自官方权威顾问团队的专业建议。

关于安全

为MongoDB集群启用认证鉴权

MongoDB服务器在默认安装下不启用鉴权。这意味着每个人都可以直接连接到mongod实例并执行任意数据库操作。建议按照文档启用鉴权 http://docs.mongoing.com/manual-zh/tutorial/enable-authentication.html

为不同用户分配不同的角色权限

MongoDB支持按角色定义的权限系统。你应该基于“最少权限”准则，显式的为用户分配仅需要的相应权限。

使用中央鉴权服务器

尽可能使用LDAP、Kerbero之类的中央鉴权服务器，并使用强口令策略。

为需要访问MongoDB的应用服务器创建白名单（防火墙配置）

如果你的服务器有多个网卡，建议只在内网的IP上监听服务。

对敏感数据使用加密引擎

MongoDB企业版支持存储加密，对涉及到客户的敏感数据应该使用加密引擎来保护数据。

关于部署

至少使用3个数据节点的复制集

MongoDB的建议最小部署是3个数据节点构成的复制集。复制集可以提供以下优点：

- 系统99.999% 高可用
- 自动故障切换
- 数据冗余
- 容灾部署
- 读写分离

不用太早分片

分片可以用来扩展你系统的读写能力，但是分片也会带来不少新的挑战比如说管理上的复杂度，成本的增加，选择合适片键的挑战性等等。一般来说，你应该先穷尽了其他的性能调优的选项以后才开始考虑分片，比如说，索引优化，模式优化，代码优化，硬件资源优化，IO优化等。

选择合适的分片数

分片的一些触发条件为：

- 数据总量太大，无法在一台服务器上管理
- 并发量太高，一台服务器无法及时处理
- 磁盘IO压力太大
- 单机系统内存不够大，无法装下热数据
- 服务器网卡处理能力达到瓶颈
- 多地部署情况下希望支持本地化读写

取决于你分片的触发条件，你可以按照总的需求 然后除以每一台服务器的能力来确定所需的分片数。

为每个分片部署足够的复制集成员

分片之间的数据互相不复制。每个分片的数据必须在分片内保证高可用。因此，对每一个分片MongoDB要求至少部署3个数据节点来保证该分片在绝大部分时间都不会因为主节点宕机而造成数据不可用。

选择合适的片键

在分片场景下， 最重要的一个考量是选择合适的片键。选择片键需要考虑到应用的读写模式。通常来说一个片键要么是对写操作优化，要么是对读操作优化。要根据哪种操作更加频繁而进行相应的权衡。

- 片键值应该具有很高的基数，或者说，这个片键在集合内有很多不同的值，例如*id就是一个基数很高的片键因为*id值不会重复
- 片键一般不应该是持续增长的，比如说timestamp就是个持续增长的片键。此类片键容易造成热分片现象，即新的写入集中到某一个分片上
- 好的片键应该会让查询定向到某一个（或几个）分片上从而提高查询效率。一般来说这个意味着片键应该包括最常用查询用到的字段
- 好的片键应该足够分散，让新的插入可以分布到多个分片上从而提高并发写入率。
- 可以使用几个字段的组合来组成片键，以达到几个不同的目的（基数，分散性，及查询定向等）

关于系统

使用SSD 或RAID10 来提高存储IOPS能力

MongoDB是一个高性能高并发的数据库，其大部分的IO操作为随机更新。一般来说本机自带的SSD是最佳的存储方案。如果使用普通的硬盘，建议使用RAID10条带化来提高IO通道的并发能力。

为Data和Journal/log使用单独的物理卷

MongoDB很多的性能瓶颈和IO相关。建议为日志盘（Journal和系统日志）单独设定一个物理卷，减少对数据盘IO的资源占用。

系统日志可以直接在命令行或者配置文件参数内指定。Journal日志不支持直接指定到另外的目录，可以通过对Journal目录创建symbol link的方式来解决。

使用XFS 文件系统

MongoDB在WiredTiger存储引擎下建议使用XFS文件系统。Ext4最为常见，但是由于ext文件系统的内部journal和WiredTiger有所冲突，所以在IO压力较大情况下表现不佳。

WiredTiger下谨慎使用超大缓存

WiredTiger 对写操作的落盘是异步发生的。默认是60秒做一次checkpoint。做checkpoint需要对内存内所有脏数据遍历以便整理然后把这些数据写入硬盘。如果缓存超大（如大于128G），那么这个checkpoint时间就需要较长时间。在checkpoint期间数据写入性能会受到影响。目前建议实际缓存设置在64GB或以下。

关闭 Transparent Huge Pages

Transparent Huge Pages (THP) 是Linux的一种内存管理优化手段，通过使用更大的内存页来减少Translation Lookaside Buffer(TLB)的额外开销。 MongoDB数据库大部分是比较分散的小量数据读写，THP对MongoDB这种工况会有负面的影响所以建议关闭。

http://docs.mongoing.com/manual-zh/tutorial/transparent-huge-pages.html

启用Log Rotation

防止MongoDB 的log文件无限增大，占用太多磁盘空间。好的实践是启用log rotation并及时清理历史日志文件。

http://docs.mongoing.com/manual-zh/tutorial/rotate-log-files.html

分配足够的Oplog空间

足够的Oplog空间可以保证有足够的时间让你从头恢复一个从节点，或者对从节点执行一些比较耗时的维护操作。假设你最长的下线维护操作需要H小时，那么你的Oplog 一般至少要保证可以保存 H *2 或者 H*3 小时的oplog。

如果你的MongoDB部署的时候未设置正确的Oplog 大小，可以参照下述链接来调整：

http://docs.mongoing.com/manual-zh/tutorial/change-oplog-size.html

关闭数据库文件的 atime

禁止系统对文件的访问时间更新会有效提高文件读取的性能。这个可以通过在 /etc/fstab 文件中增加 noatime 参数来实现。例如:

```
/dev/xvdb /data ext4 noatime 0 0
```

修改完文件后重新 mount就可以:

```
# mount -o remount /data
```

提高默认文件描述符和进程/线程数限制

Linux默认的文件描述符数和最大进程数对于MongoDB来说一般会太低。建议把这个数值设为64000。因为MongoDB服务器对每一个数据库文件以及每一个客户端连接都需要用到一个文件描述符。如果这个数字太小的话在大规模并发操作情况下可能会出错或无法响应。 你可以通过以下命令来修改这些值：

```
ulimit -n 64000
ulimit -u 64000
```

禁止 NUMA

在一个使用NUMA技术的多处理器Linux 系统上，你应该禁止NUMA的使用。MongoDB在NUMA环境下运行性能有时候会可能变慢，特别是在进程负载很高的情况下。

预读值(readahead)设置

预读值是文件操作系统的一个优化手段，大致就是在程序请求读取一个页面的时候，文件系统会同时读取下面的几个页面并返回。这原因是因为很多时候IO最费时的磁盘寻道。通过预读，系统可以提前把紧接着的数据同时返回。假设程序是在做一个连续读的操作，那么这样可以节省很多磁盘寻道时间。

MongoDB很多时候会做随机访问。对于随机访问，这个预读值应该设置的较小为好.一般来说32是一个不错的选择。

你可以使用下述命令来显示当前系统的预读值：

```
sudo blockdev --report
```

要更改预读值，可以用以下命令：

```
sudo blockdev --setra 32
```

把 换成合适的存储设备。

使用NTP时间服务器

在使用MongoDB复制集或者分片集群的时候，注意一定要使用NTP时间服务器。这样可以保证MongoDB集群成原则之间正确同步。

关于监控及备份

对重要的数据库指标进行监控及告警

关键的指标包括：

```
- Disk Space 磁盘空间
- CPU
- RAM 使用率
- Ops Counter 增删改查
- Replication Lag 复制延迟
- Connections 连接数
- Oplog Window
```

对慢查询日志进行监控

默认情况下MongoDB会在日志文件中(mongod.log)记录超过100ms的数据库操作。

关于索引

为你的每一个查询建立合适的索引

这个是针对于数据量较大比如说超过几十上百万（文档数目）数量级的集合。如果没有索引MongoDB需要把所有的Document从盘上读到内存，这会对MongoDB服务器造成较大的压力并影响到其他请求的执行。

创建合适的组合索引，不要依赖于交叉索引

如果你的查询会使用到多个字段，MongoDB有两个索引技术可以使用：交叉索引和组合索引。交叉索引就是针对每个字段单独建立一个单字段索引，然后在查询执行时候使用相应的单字段索引进行索引交叉而得到查询结果。交叉索引目前触发率较低，所以如果你有一个多字段查询的时候，建议使用组合索引能够保证索引正常的使用。

例如，如果应用需要查找所有年龄小于30岁的深圳市马拉松运动员：

```
db.athelets.find({sport: "marathon", location: "sz", age: {$lt: 30}})
```

那么你可能需要这样的一个索引：

```
db.athelets.ensureIndex({sport:1, location:1, age:1});
```

组合索引字段顺序：匹配条件在前，范围条件在后（Equality First, Range After）

以上文为例子，在创建组合索引时如果条件有匹配和范围之分，那么匹配条件（sport: “marathon”) 应该在组合索引的前面。范围条件(age: <30)字段应该放在组合索引的后面。

尽可能使用覆盖索引（Covered Index）

有些时候你的查询只需要返回很少甚至只是一个字段，例如，希望查找所有虹桥机场出发的所有航班的目的地。已有的索引是：

```
{origin: 1, dest: 1}
```

如果正常的查询会是这样（只需要返回目的地机场）：

```
db.flights.find({origin:"hongqiao"}, {dest:1});
```

这样的查询默认会包含_id 字段，所以需要扫描匹配的文档并取回结果。相反，如果使用这个查询语句：

```
db.flights.find({origin:"hongqiao"}, {_id:0, dest:1});
```

MongoDB则可以直接从索引中取得所有需要返回的值，而无需扫描实际文档（文档可能需要从硬盘里调入到内存）

建索引要在后台运行

在对一个集合创建索引时，该集合所在的数据库将不接受其他读写操作。对数据量的集合建索引，建议使用后台运行选项 {background: true}

关于开发

模式设计

不要按照关系型来设计表结构

MongoDB可以让你像关系型数据库一样设计表结构，但是它不支持外键，也不支持复杂的Join！如果你的程序发现有大量实用JOIN的地方，那你的设计可能需要重新来过。参照以下相关模式设计建议。

数据库集合（collection）的数量不宜太多

MongoDB的模式设计基于灵活丰富的JSON文档模式。在很多情况下，一个MongoDB应用的数据库内的集合（表）的数量应该远远小于使用关系数据库的同类型应用。MongoDB表设计不遵从第三范式。MongoDB的数据模型非常接近于对象模型，所以基本上就是按照主要的Domain object的数量来建相应的集合。根据经验，一般小型应用的集合数量通常在几个之内，中大型的应用会在10多个或者最多几十个。

不要害怕数据冗余

MongoDB模式设计不能按照第三范式，很多时候允许数据在多个文档中重复，比如说，在每一个员工的文档中重复他的部门名字，就是一个可以接受的做法。如果部门名字改了，可以执行一个update({},{}, {multi:true}) 的多文档更新来一次性把部门名字更新掉。

适合和不适合冗余的数据类型

一般来说，如果某个字段的数据值经常会变，则不太适合被大量冗余到别的文档或者别的集合里面去。举例来说，如果我们是在做一些股票类型资产管理， 可能有很多人都购买了Apple的股票，但是如果把经常变动的股价冗余到客户的文档里，由于股票价格变动频繁，会导致有大量的更新操作。从另外一个角度来说，如果是一些不经常变的字段，如客户的姓名，地址，部门等，则可以尽管进行冗余shi'yang

对 1：N（一些）的关系使用全部内嵌

对于一对多的关系，如一个人有几个联系方式，一本书有10几个章节，等等，建议使用内嵌方式，把N的数据以数组形式来描述，如：

```
> db.person.findOne()
{
  user_id: 'tjworks',
  name: 'TJ Tang',
  contact : [
     { type: 'mobile', number: '1856783691' },
     { type: 'wechat', number: 'tjtang826'}
  ]
}
```

对 1: NN (很多) 的关系使用ID内嵌

有些时候这个一对多的多端数量较大， 比如说，一个部门内有多少员工。在华为一个三级部门可能有数千员工，这个时候如果把所有员工信息直接内嵌到部门内肯定不是个好的选择，有可能会超出16MB的文档限制。这个时候可以采用引用ID的方式：

```
> db.departments.findOne()
{
    name : 'Enterprise BG',
    president: 'Zhang San',
    employees : [     // array of references to Employee colletion
        ObjectID('AAAA'),
        ObjectID('F17C'),
        ObjectID('D2AA'),
        // etc
    ]
}
```

如果需要查询部门下员工相关信息，你可以使用$lookup聚合操作符来把员工信息进行关联并返回。

对 1: NNN (很多) 的关系使用

如果一对多情况下，这个多端数量无限大并会频繁增长，比如说，一个测量仪的每分钟读数，一年下来有几十万条，这个时候即使是把ID放到数组里都会管理不便，这个时候就应该把多端的数据创建一个集合，并在那个集合的文档里加入对主文档的连接引用，如：

```
    > db.sensors.findOne()
    {
        _id : ObjectID('AAAB'),
        name : 'engine temperature',
        vin : '4GD93039GI239',
        engine_id: '20394802',
        manuafacture: 'First Motor',
        production_date: '2014-02-01'
        ...
    }

    >db.readings.findOne()
    {
        time : ISODate("2014-03-28T09:42:41.382Z"),
        sensor: ObjectID('AAAB'),
        reading: 67.4
    }
```

把二进制大文件和元数据分集合存放

如果你有需要把PDF文件，图片，甚至小视频等二进制文件需要管理，建议使用MongoDB 的GridFS API 或者自己手动分集合来分开管理二进制数据和元数据。

经常更新的数据不要放在嵌套数组内

数组是用来表达 1对多关系的利器，但是MongoDB对嵌套的数组内元素缺乏直接更新能力。比如说：

```
{
    name: "Annice",
    courses: [
        { name: "English", score: 97 },
        { name: "Math", score: 89 },
        { name: "Physics", score: 95 }
    ]
}
```

这样设计没有嵌套数组，我们可以直接对 Math的score 修改为99：

```
db.students.update({name: "Annice",  "courses.name":"Math"}, {$set:{"courses.$.score": 99 }})
```

注意数组定位符 的用法， 表示当前匹配的第一个数组元素的在数组内的索引。

但是下面这种情况就涉及到了数组嵌套：

```
    {
        name: "Annice",
        courses: [
            { name: "Math", scores: [
                                {term: 1, score: 80} ,
                                {term: 2, score: 90}
                            ]
             },
            { name: "Physics", score: 95 }
        ]
    }
```

这个时候如果你想对Math course的term 1的Score进行修改，你就需要把 scores 这个数组整个调到内存然后在代码里对这个嵌套数组的元素进行修改。这是因为MongoDB的数组定位符 $ 只对第一层数组有效。

当然，如果你的模型不需要修改嵌套的数组内元素，那么这条就不适用。

程序配置

设定合适的MongoDB连接池大小 （Connections Per Host）

Java驱动的默认连接池大小是100。建议按照应用的实际情况做调整。对压力较小的应用可以适当调小减少对应用服务器的资源占用。

正确使用写关注设置（Write Concern）

MongoDB的建议最小部署是一个复制集，包含3个数据节点。默认情况下应用的写操作（更新，插入或者删除）在主节点上完成后就会立即返回。写操作则通过OPLOG方式在后台异步方式复制到其他节点。在极端情况下，这些写操作可能还未在复制到从节点的时候主节点就出现宕机。这个时候发生主备节点切换，原主节点的写操作会被回滚到文件而对应用不可见。为防止这种情况出现，MongoDB建议对重要的数据使用 {w: “marjority”} 的选项。{w: “majority”} 可以保证数据在复制到多数节点后才返回成功结果。使用该机制可以有效防止数据回滚的发生。

另外你可以使用 {j:1} （可以和 w:”majrotiy” 结合使用） 来指定数据必须在写入WAL日志之后才向应用返回成功确认。这个会导致写入性能有所下降，但是对于重要的数据可以考虑使用。

正确使用读选项设置（Read Preference）

MongoDB由于是一个分布式系统，一份数据会在多个节点上进行复制。从哪个节点上读数据，要根据应用读数据的需求而定。以下是集中可以配置的读选项：

- primary: 默认，在主节点上读数据
- priaryPreferred: 先从主节点上读，如果为成功再到任意一台从节点上读
- secondary： 在从节点上读数据（当有多台节点的时候，随机的使用某一台从节点）
- secondaryPreferred： 首先从从节点上读，如果从节点由于某种原因不能提供服务，则从主节点上进行读
- nearest： 从距离最近的节点来读。距离由ping操作的时间来决定。

除第一个选项之外，其他读选项都存在读到的数据不是最新的可能。原因是数据的复制是后台异步完成的。

不要实例化多个MongoClient

MongoClient是个线程安全的类，自带线程池。通常在一个JVM内不要实例化多个MongoClient实例，避免连接数过多和资源的不必要浪费。

对写操作使用Retry机制

MongoDB使用复制集技术可以实现99.999%的高可用。当一台主节点不能写入时，系统会自动故障转移到另一台节点。转移可能会耗时几秒钟，在这期间应用应该捕获相应的Exception并执行重试操作。重试应该有backoff机制，例如，分别在1s，2s，4s，8s等时候进行重试。

避免使用太长的字段名

MongoDB 没有表结构定义。每个文档的结构由每个文档内部的字段决定。所有字段名会在每个文档内重复。使用太长的字段名字会导致对内存、网络带宽更多的需求。（由于压缩技术，长字段名对硬盘上的存储不会有太多占用）

使用有规律的命名方式

如： School， Course, StudentRecord 或者：school, course, stuent_record

正确使用更新语句

不要把MongoDB和普通的键值型数据库（KV）视为等同。MongoDB支持和关系型数据库update语句类似的in place update。你只需要在update语句中指定需要更新的字段，而不是整个文档对象。

举例来说，加入我想把用户的名字从TJ改为Tang Jianfa.

不建议的做法：

```
user = db.users.findOne({_id: 101});
user.name="Tang Jianfa"
db.users.save(user);
```

建议的做法：

```
user = db.users.findOne({_id: 101});
// do certain things
db.users.update({_id:101}, {$set: {name: "Tang Jianfa"}});
```

使用投射 （projection）来减少返回的内容

MongoDB 支持类似于SQL语句里面的select，可以对返回的字段进行过滤。使用Projection可以减少返回的内容，降低网络传输的量和代码中转化成对象所需的时间。

使用TTL来自动删除过期的数据

很多时候我们用MongoDB来存储一些时效性的数据，如7天的监控数据。与其自己写个后台脚本定期清理过期数据，你可以使用TTL索引来让MongoDB自动删除过期数据：

```
db.data.ensureIndex({create_time:1}, {expireAfterSeconds: 7*24*3600})
```

使用execute命令来实现upsert

有些时候你不知道一条文档数据是否已经在库里存在。这个时候你要么先查询一下，要么就是使用upsert语句。在SpringData下面upsert语句需要你把每个字段的值都在upsert语句中格式化出来。字段多的时候未免有些繁琐。SpringData MongoDB里面的MongoTemplate有个execute方法可以用来实现一个DB调用，也不用繁琐的把所有字段罗列出来的例子。

```
    public boolean persistEmployee(Employee employee) throws Exception {

        BasicDBObject dbObject = new BasicDBObject();
        mongoTemplate.getConverter().write(employee, dbObject);
        mongoTemplate.execute(Employee.class, new CollectionCallback<Object>() {
            public Object doInCollection(DBCollection collection) throws MongoException,
DataAccessException {
                collection.update(new
Query(Criteria.where("name").is(employee.getName())).getQueryObject(),
                        dbObject,
                        true,  // means upsert - true
                        false  // multi update - false
                );
                return null;
            }
        });
        return true;
    }
```

删除SpringData MongoDB下面的_class 字段

SpringData MongoDB默认会在MongoDB文档中添加一个_class字段，里面保存的是fully qualified class name，如”com.mongodb.examples.Customer”。对于有些小文档来说，这个字段可能会占据不小一部分的存储空间。如果你不希望SpringData 自动加入这个字段，你可以：

1） 自定义MongoTypeMapper

```
@Bean
public MongoTemplate mongoTemplate() throws UnknownHostException {
    MappingMongoConverter mappingMongoConverter =  new MappingMongoConverter(new
DefaultDbRefResolver
            (mongoDbFactory()), new
            MongoMappingContext());
    mappingMongoConverter.setTypeMapper(new DefaultMongoTypeMapper(null));
    return new MongoTemplate(mongoDbFactory(), mappingMongoConverter );
}
```

2) 在使用find语句时，显式地指定类的名字/类型：

```
    MongoTemplate.find(new Query(), Inventory.class))
```