MongoDB 简明教程

1.MongDB 简介

MongoDB 是专为可扩展性，高性能和高可用性而设计的数据库。它可以从单服务器部署扩展到大型、复杂的多数据中心架构。利用内存计算的优势，MongoDB 能够提供高性能的数据读写操作。MongoDB 的本地复制和自动故障转移功能使的应用程序具有企业级的可靠性和操作灵活性。

1.1 文档型数据库

简而言之，MongoDB 是一个免费开源跨平台的 NoSQL 数据库，与关系型数据库不同，MongoDB 的数据以类似于 JSON 格式的二进制文档存储：

```
{
        name:  "wangjie",
        age:  22,
}
```

文档型的数据存储方式有几个重要好处：

1. 文档的数据类型可以对应到语言的数据类型，如数组类型（Array）和对象类型（Object）；
2. 文档可以嵌套，有时关系型数据库涉及几个表的操作，在 MongoDB 中一次就能完成，可以减少昂贵的连接花销；
3. 文档不对数据结构加以限制，不同的数据结构可以存储在同一张表；
4. MongoDB 的文档数据模型和索引系统能有效提升数据库性能；
5. 复制集功能提供数据冗余，自动化容灾容错，提升数据库可用性；

6. 分片技术能够分散单服务器的读写压力，提高并发能力，提升数据库的可拓展性；

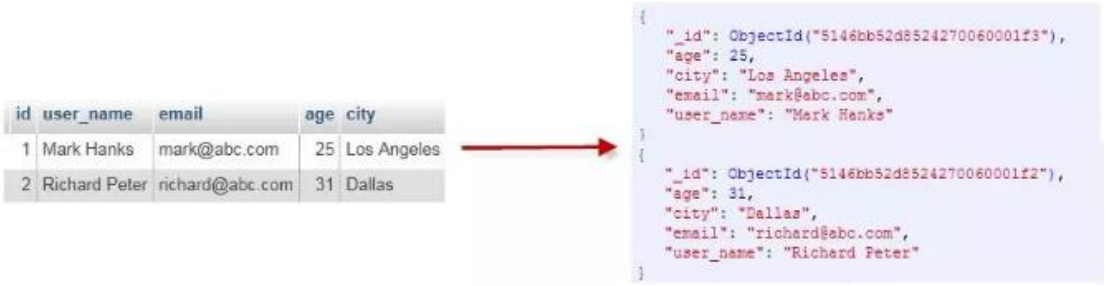
7. MongoDB 高性能，高可用性、可扩展性等特点。

1.2 MongoDB 基础概念

可以使用我们熟悉的 MySQL 数据库来加以对比：

MySQL 基础概念	MongoDB 对应概念
数据库（database）	容器（database）
表（table）	集合（collection）
行（row）	文档（document）
列（column）	域（filed）
索引（index）	索引（index）

)的图来更加形象生动的说明一下：



这很容易理解，但是问题在于：我们为什么要引入新的概念呢？（也就是为什么我们要把“表”替换成“集合”，“行”替换成“文档”，“列”替换成“域”呢？）原因在于，其实在 MySQL 这样的典型关系型数据中，我们是在定义表

的时候定义列的，但是由于上述文档型数据库的特点，它允许文档的数据类型可以对应到语言的数据类型，所以我们是在定义文档的时候才会定义域的。

也就是说，集合中的每个文档都可以有独立的域。因此，虽说集合相对于表来说是一个简化了的容器，而文档则包含了比行要多得多的信息。

2 搭建环境

怎么样都好，搭建好环境就行，这里以 OS 环境为例，可以使用 OSX 的 brew 安装 mongodb：

```
brew install mongodb
```

在运行之前我们需要创建一个数据库存储目录 /data/db：

```
sudo mkdir -p /data/db
```

然后启动 mongodb，默认数据库目录即为 /data/db（如果不是，可以使用 --dbpath 指令来指定）：

```
sudo mongd
```

过一会儿就能看到的 mongodb 运行起来的提示：

```
2019-04-24T18:18:25.866+0800 I RECOVERY [initandlisten] WiredTiger recoveryTimestamp. Ts: Timestamp(0, 0)
2019-04-24T18:18:25.911+0800 I CONTROL  [initandlisten]
2019-04-24T18:18:25.911+0800 I CONTROL  [initandlisten] ** WARNING: Access control is not enabled for the d
atabase.
2019-04-24T18:18:25.911+0800 I CONTROL  [initandlisten] **          Read and write access to data and confi
guration is unrestricted.
2019-04-24T18:18:25.911+0800 I CONTROL  [initandlisten] ** WARNING: You are running this process as the roo
t user, which is not recommended.
2019-04-24T18:18:25.911+0800 I CONTROL  [initandlisten]
2019-04-24T18:18:25.911+0800 I CONTROL  [initandlisten] ** WARNING: This server is bound to localhost.
2019-04-24T18:18:25.911+0800 I CONTROL  [initandlisten] **          Remote systems will be unable to connec
t to this server.
2019-04-24T18:18:25.911+0800 I CONTROL  [initandlisten] **          Start the server with --bind_ip <addres
s> to specify which IP
2019-04-24T18:18:25.911+0800 I CONTROL  [initandlisten] **          addresses it should serve responses fro
m, or with --bind_ip_all to
2019-04-24T18:18:25.911+0800 I CONTROL  [initandlisten] **          bind to all interfaces. If this behavio
r is desired, start the
2019-04-24T18:18:25.911+0800 I CONTROL  [initandlisten] **          server with --bind_ip 127.0.0.1 to disa
ble this warning.
2019-04-24T18:18:25.911+0800 I CONTROL  [initandlisten]
2019-04-24T18:18:25.961+0800 I FTDC     [initandlisten] Initializing full-time diagnostic data capture with
```

3 基于 Shell 的 CRUD

3.1 连接实例

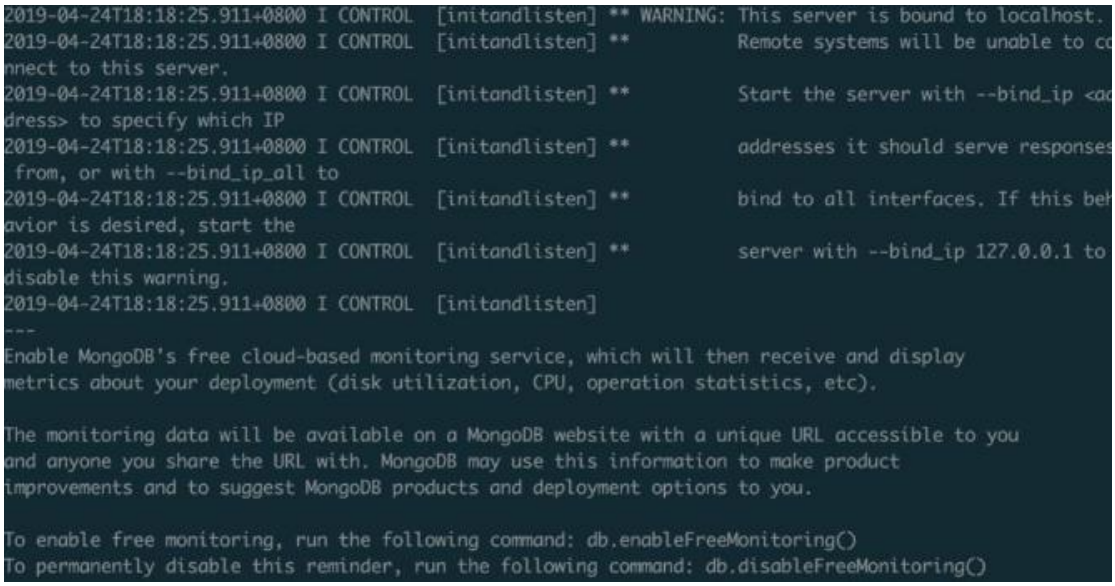
通过上面的步骤我们在系统里运行了一个 mongodb 实例，接下来通过 mongo 命令来连接它：

mongo [options] [db address] [file names]

由于上面运行的 mongodb 运行在 27017 端口，并且灭有启动安全模式，所以我们也不需要输入用户名和密码就可以直接连接：

mongo 127.0.0.1:27017

或者通过 --host 和 --port 选项指定主机和端口。一切顺利的话，就进入了 mongoDBshell，shell 会报出一连串权限警告，不过不用担心，这并不会影响之后的操作。在添加授权用户和开启认证后，这些警告会自动消失。



3.2 CRUD 操作

在进行增删改查操作之前，我们需要先了解一下常用的 shell 命令：

- db 显示当前所在数据库，默认为 test

- show dbs 列出可用数据库
- show tables show collections 列出数据库中可用集合
- use 用于切换数据库

mongoDB 预设有两个数据库，admin 和 local，admin 用来存放系统数据，local 用来存放该实例数据，在副本集中，一个实例的 local 数据库对于其它实例是不可见的。使用 use 命令切换数据库：

```
> use admin
> use local
> use newDatabase
```

可以 use 一个不存在的数据库，当存入新数据时，mongoDB 会创建这个数据库：

```
> use newDatabase
> db.newCollection.insert({x:1})
WriteResult({ "nInserted" : 1 })
```

以上命令向数据库中插入一个文档，返回 1 表示插入成功，mongoDB 自动创建 newCollection 集合和数据库 newDatabase。下面将对增查改删操作进行一个简单的演示。

3.2.1 创建（Create）

MongoDB 提供 insert 方法创建新文档：

- db.collection.inserOne() 插入单个文档
  WriteResult({ "nInserted" : 1 })
- db.collection.inserMany() 插入多个文档
- db.collection.insert() 插入单条或多条文档

我们接着在刚才新创建的 newDatabase 下面新增数据吧：

```
db.newCollection.insert({name:"wmyskxz",age:22})
```

根据以往经验应该会觉得蛮奇怪的，因为之前在这个集合中插入的数据格式是 {x:1} 的，而这里新增的数据格式确

是 {name:"wmyskxz",age:22} 这个样子的。还记得吗，文档型数据库的与传统型的关系型数据的区别就是在这里！

并且要注意,age:22 和 age:"22" 是不一样的哦,前者插入的是一个数值,而后者是字符串，我们可以通过 db.newCollection.find() 命令查看到刚刚插入的文档：

```
> db.newCollection.find()
{ "_id" : ObjectId("5cc1026533907ae66490e46c"), "x" : 1 }
{ "_id" : ObjectId("5cc102fb33907ae66490e46d"), "name" : "wm
yskxz", "age" : 22 }
```

这里有一个神奇的返回，那就是多了一个叫做 _id 的东西，这是 MongoDB 为自动添加的字段，也可以自己生成。大部分情况下还是会让 MongoDB 为我们生成，而且默认情况下，该字段是被加上了索引的。

3.2.2 查找（Read）

MongoDB 提供 find 方法查找文档，**第一个参数为查询条件：**

```
> db.newCollection.find() # 查找所有文档
{ "_id" : ObjectId("5cc1026533907ae66490e46c"), "x" : 1 }
{ "_id" : ObjectId("5cc102fb33907ae66490e46d"), "name" : "wm
yskxz", "age" : 22 }
> db.newCollection.find({name:"wmyskxz"}) # 查
找 name 为 wmyskxz 的文档
{ "_id" : ObjectId("5cc102fb33907ae66490e46d"), "name" : "wm
yskxz", "age" : 22 }
> db.newCollection.find({age:{$gt:20}}) # 查找 age 大
于 20 的文档
{ "_id" : ObjectId("5cc102fb33907ae66490e46d"), "name" : "wm
yskxz", "age" : 22 }
```

上述代码中的$gt 对应于大于号>的转义。

第二个参数可以传入投影文档映射数据：

```
> db.newCollection.find({age:{$gt:20}},{name:1})
{ "_id" : ObjectId("5cc102fb33907ae66490e46d"), "name" : "wmyskxz" }
```

上述命令将查找 age 大于 20 的文档，返回 name 字段，排除其他字段。投影文档中字段为 1 或其他真值表示包含，0 或假值表示排除，可以设置多个字段位为 1 或 0，但不能混合使用。

为了测试，我们为这个集合弄了一些奇奇怪怪的数据：

```
> db.newCollection.find()
{ "_id" : ObjectId("5cc1026533907ae66490e46c"), "x" : 1 }
{ "_id" : ObjectId("5cc102fb33907ae66490e46d"), "name" : "wmyskxz", "age" : 22 }
{ "_id" : ObjectId("5cc108fb33907ae66490e46e"), "name" : "wmyskxz-test", "age" : 22, "x" : 1, "y" : 30 }
```

然后再来测试：

```
> db.newCollection.find({age:{$gt:20}},{name:1,x:1})
{ "_id" : ObjectId("5cc102fb33907ae66490e46d"), "name" : "wmyskxz" }
{ "_id" : ObjectId("5cc108fb33907ae66490e46e"), "name" : "wmyskxz-test", "x" : 1 }
> db.newCollection.find({age:{$gt:20}},{name:0,x:0})
{ "_id" : ObjectId("5cc102fb33907ae66490e46d"), "age" : 22 }
{ "_id" : ObjectId("5cc108fb33907ae66490e46e"), "age" : 22, "y" : 30 }
> db.newCollection.find({age:{$gt:20}},{name:0,x:1})
Error: error: {
        "ok" : 0,
        "errmsg" : "Projection cannot have a mix of inclusion and exclusion.",
        "code" : 2,
        "codeName" : "BadValue"
}
```

从上面的命令我们就可以把我们的一些想法和上面的结论得以验证，perfect!

除此之外，还可以通过 count、skip、limit 等指针（Cursor）方法，改变文档查询的执行方式：

```
> db.newCollection.find().count()
3
> db.newCollection.find().skip(1).limit(10).sort({age:1})
{ "_id" : ObjectId("5cc102fb33907ae66490e46d"), "name" : "wmyskxz", "age" : 22 }
{ "_id" : ObjectId("5cc108fb33907ae66490e46e"), "name" : "wmyskxz-test", "age" : 22, "x" : 1, "y" : 30 }
```

上述查找命令跳过 1 个文档，限制输出 10 个，以 age 子段正序排序（大于 0 为正序，小于 0 位反序）输出结果。最后，可以使用 Cursor 方法中的 pretty 方法，提升查询文档的易读性，特别是在查看嵌套的文档和配置文件的时候：

```
> db.newCollection.find().pretty()
{ "_id" : ObjectId("5cc1026533907ae66490e46c"), "x" : 1 }
{
        "_id" : ObjectId("5cc102fb33907ae66490e46d"),
        "name" : "wmyskxz",
        "age" : 22
}
{
        "_id" : ObjectId("5cc108fb33907ae66490e46e"),
        "name" : "wmyskxz-test",
        "age" : 22,
        "x" : 1,
        "y" : 30
}
```

3.2.3 更新（Update）

MongoDB 提供 update 方法更新文档：

- db.collection.updateOne() 更新最多一个符合条件的文档
- db.collection.updateMany() 更新所有符合条件的文档
- db.collection.replaceOne() 替代最多一个符合条件的文档
- db.collection.update() 默认更新一个文档，可配置 multi 参数，跟新多个文档

以 update() 方法为例。其格式：

```
> db.collection.update(
        <query>,
        <update>,
        {
                upsert: <boolean>,
                multi: <boolean>
        }
)
```

各参数意义：

- query 为查询条件
- update 为修改的文档
- upsert 为真，查询为空时插入文档
- multi 为真，更新所有符合条件的文档

下面我们测试把 name 字段为 wmyskxz 的文档更新一下试试：

```
> db.newCollection.update({name:"wmyskxz"},{name:"wmyskxz",age:30})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1
 })
```

要注意的是，如果更新文档只传入 age 字段，那么文档会被更新为{age:30}，而不是{name:"wmyskxz", age:30}。要避免文档被覆盖，需要用到 $set 指令，$set 仅替换或添加指定字段：

```
> db.newCollection.update({name:"wmyskxz"},{$set:{age:30}})
```

如果要在查询的文档不存在的时候插入文档，要把 upsert 参数设置真值：

```
> db.newCollection.update({name:"wmyskxz11"},{$set:{age:30}},{upsert:tr
ue})
```

update 方法默认情况只更新一个文档，如果要更新符合条件的所有文档，要把 multi 设为真值，并使用 $set 指令：

```
> db.newCollection.update({age:{$gt:20}},{$set:{test:"A"}},{multi:true})
WriteResult({ "nMatched" : 3, "nUpserted" : 0, "nModified" : 3
 })
> db.newCollection.find()
{ "_id" : ObjectId("5cc1026533907ae66490e46c"), "x" : 1 }
```

```
{ "_id" : ObjectId("5cc102fb33907ae66490e46d"), "name" : "wmyskxz", "age" : 30, "test" : "A" }
{ "_id" : ObjectId("5cc108fb33907ae66490e46e"), "name" : "wmyskxz-test", "age" : 22, "x" : 1, "y" : 30, "test" : "A" }
{ "_id" : ObjectId("5cc110148d0a578f03d43e81"), "name" : "wmyskxz11", "age" : 30, "test" : "A" }
```

3.2.4 删除（Delete）

MongoDB 提供了 **delete** 方法删除文档：

- db.collection.deleteOne() 删除最多一个符合条件的文档
- db.collection.deleteMany() 删除所有符合条件的文档
- db.collection.remove() 删除一个或多个文档

以 remove 方法为例：

```
> db.newCollection.remove({name:"wmyskxz11"})
> db.newCollection.remove({age:{$gt:20}},{justOne:true})
> db.newCollection.find()
{ "_id" : ObjectId("5cc1026533907ae66490e46c"), "x" : 1 }
{ "_id" : ObjectId("5cc108fb33907ae66490e46e"), "name" : "wmyskxz-test", "age" : 22, "x" : 1, "y" : 30, "test" : "A" }
```

MongoDB 提供了 drop 方法删除集合，返回 true 表面删除集合成功：

```
> db.newCollection.drop()
```

3.2.5 小结

相比传统关系型数据库，MongoDB 的 CURD 操作更像是编写程序，更符合开发人员的直觉，不过 MongoDB 同样也支持 SQL 语言。MongoDB 的 CURD 引擎配合索引技术、数据聚合技术和 JavaScript 引擎，赋予 MongoDB 用户更强大的操纵数据的能力。

4 MongoDB 数据模型的一些讨论

这是一个抽象的话题，与大多数 NoSQL 方案相比,在建模方面,面向文档的数据库算是和关系数据库相差最小的。这些差别是很小，但是并不是说不重要。

4.1 没有连接（Join）

要接受的第一个也是最基本的一个差别，就是 **MongoDB 没有连接（join）**。我不知道 MongoDB 不支持某些类型连接句法的具体原因，但是我知道一般而言人们认为连接是不可扩展的。也就是说，**一旦开始横向分割数据，最终不可避免的就是在客户端（应用程序服务器）使用连接**。且不论 MongoDB 为什么不支持连接，事实是数据是有关系的，可是 MongoDB 不支持连接。（译者：这里的关系指的是不同的数据之间是有关联的，对于没有关系的数据，就完全不需要连接。）

为了在没有连接的 MongoDB 中生存下去，在没有其他帮助的情况下，我们必须在自己的应用程序中实现连接。

基本上我们需要**用第二次查询**去找到相关的数据。找到并组织这些数据相当于在关系数据库中声明一个外来的键。现在先别管什么独角兽了，我们来看看我们的员工。首先我们创建一个员工的数据（这次我告诉具体的_id 值，这样我们的例子就是一样的了）：

```
db.employees.insert({_id: ObjectId("4d85c7039ab0fd70a117d730"), name: 'Leto'})
```

然后我们再加入几个员工并把 Leto 设成他们的老板：

```
db.employees.insert({_id: ObjectId("4d85c7039ab0fd70a117d731"), name: 'Duncan', manager: ObjectId("4d85c7039ab0fd70a117d730")});
db.employees.insert({_id: ObjectId("4d85c7039ab0fd70a117d732"), name: 'Moneo', manager: ObjectId("4d85c7039ab0fd70a117d730")});
```

（有必要再强调一下，_id 可以是任何的唯一的值。在实际工作中很可能会用到 ObjectId， 所以我们在这里也使用它）

显然，要找到 Leto 的所有员工，只要执行：

```
db.employees.find({manager: ObjectId("4d85c7039ab0fd70a117d730")})
```

没什么了不起的。在最糟糕的情况下，为弥补连接的缺失需要做的只是再多查询一次而已，该查询很可能是经过索引了的。

4.1.1 数组和嵌入文档（Embedded Documents）

MongoDB 没有连接并不意味着它没有其他的优势。还记得我们曾说过 MongoDB 支持数组并把它当成文档中的一级对象吗？当处理多对一或是多对多关系的时候，这一特性就显得非常好用了。用一个简单的例子来说明，如果一个员工有两个经理，我们可以把这个关系储存在一个数组当中：

```
({name: 'Siona', manager: [ObjectId("4d85c7039ab0fd70a117d730"), Obj
ectId("4d85c7039ab0fd70a117d732")] })
```

需要注意的是，在这种情况下，有些文档中的 manager 可能是一个向量，而其他的却是数组。在两种情况下，前面的 find 还是一样可以工作：

```
db.employees.find({manager: ObjectId("4d85c7039ab0fd70a117d730")})
```

很快就会发现数组中的值比起多对多的连接表（join-table）来说要更容易处理。

除了数组，MongoDB 还支持嵌入文档。尝试插入含有内嵌文档的文档，像这样：

```
db.employees.insert({_id: ObjectId("4d85c7039ab0fd70a117d734"), name:
 'Ghanima', family: {mother: 'Chani', father: 'Paul', brother: O
bjectId("4d85c7039ab0fd70a117d730")}})
```

也许会这样想，确实也可以这样做：嵌入文档可以用‘.’符号来查询：

```
db.employees.find({'family.mother': 'Chani'})
```

就这样，我们简要地介绍了嵌入文档适用的场合以及应该怎样使用它。

4.1.2 DBRef

MongoDB 支持一个叫做 DBRef 的功能，许多 MongoDB 的驱动都提供对这一功能的支持。当驱动遇到一个 DBRef 时它会把当中引用的文档读取出来。DBRef 包含了所引用的文档的 ID 和所在的集合。它通常专门用于这样的场合：相同集合中的文档需要引用另外一个集合中的不同文档。例如，文档 1 的 DBRef 可能指向 managers 中的文档，而文档 2 中的 DBRef 可能指向 employees 中的文档。

4.1.3 范规范化（Denormalization）

代替连接的另一种方法就是反规范化数据。在过去，反规范化是为性能敏感代码所设，或者是需要数据快照（例如审计日志）的时候才应用的。然而，随着 NoSQL 的日渐普及，有许多这样的数据库并不提供连接操作，于是作为规范建模的一部分，反规范化就越来越常见了。这样说并不是说就需要为每个文档中的每一条信息创建副本。与此相反，与其在设计的时候被复制数据的担忧牵着走，还不如按照不同的信息应该归属于相应的文档这一思路来对数据建模。

比如说，假设在编写一个论坛的应用程序。把一个 user 和一篇 post 关联起来的传统方法是在 posts 中加入一个 userid 的列。这样的模型中，如果要显示 posts 就不得不读取（连接）users。一种简单可行的替代方案就是直接把 name 和 userid 存储在 post 中。甚至可以用嵌入文档来实现，比如说 user: {id: ObjectId('Something'), name: 'Leto'}。当然，如果允许用户更改他们的用户名，那么每当有用户名修改的时候，就需要去更新所有的文档了（这需要一个额外的查询）。

对一些人来说改用这种方法并非易事。甚至在一些情况下根本行不通。不过别不敢去尝试这种方法：有时候它不仅可行，而且就是正确的方法。

4.1.4 应该选择哪一种？

当处理一对多或是多对多问题的时候，采用 id 数组往往都是正确的策略。可以这么说，DBRef 并不是那么常用，虽然完全可以试着采用这项技术。这使得新手们在面临选择嵌入文档还是手工引用（manual reference）时犹豫不决。

首先，要知道目前一个单独的文档的大小限制是 4MB，虽然已经比较大了。了解了这个限制可以为如何使用文档提供一些思路。目前看来多数的开发者还是大量地依赖手工引用来维护数据的关系。嵌入文档经常被使用，but mostly for small pieces of data which we want to always pull with the parent document。一个真实的例子，我把 accounts 文档嵌入存储在用户的文档中，就像这样：

```
db.users.insert({name: 'leto', email: 'leto@dune.gov', account: {allowed_gholas: 5, spice_ration: 10}})
```

这不是说就应该低估嵌入文档的作用，也不是说应该把它当成是鲜少用到的工具并直接忽略。将数据模型直接映射到目标对象上可以使问题变得更加简单，也往往因此而不再需要连接操作。当知道 MongoDB 允许对嵌入文档的域进行查询并做索引后，这个说法就尤其显得正确了。

4.2 集合：少一些还是多一些？

既然集合不强制使用模式，那么就完全有可能用一个单一的集合以及一个不匹配的文档构建一个系统。以我所见过的情况，大部分的 MongoDB 系统都像在关系数据库中所见到的那样布局。换句话说，如果在关系数据库中会用表，那么很有可能在 MongoDB 中就要用集合（多对多连接表在这里是一个不可忽视的例外）

当把嵌入文档引进来的时候，讨论就会变得更加有意思了。最常见的例子就是博客系统。是应该分别维护 posts 和 comments 两个集合，还是在每个 post 中嵌入一个 comments 数组？暂且不考虑那个 4MB 的限制（哈姆雷特所有的评论也不超过 200KB，谁的博客会比他更受欢迎？），大多数的开发者还是倾向于把数据划分开。因为这样既简洁又明确。

没有什么硬性的规定（呃，除了 4MB 的限制）。做了不同的尝试之后就可以凭感觉知道怎样做是对的了。

总结

已经对 MongoDB 有了一个基本的了解和入门，但是要运用在实际的项目中仍然有许多实践需要自己去完成

ougd . ai . wangjie