社交网站数据库技术分析_知识库_博客园

文 / 丁艺明

传统互联网正在迈向一个全新的时代——社交服务网（Social Networking Service）时代，从“人与机器”的时代迈向“人与人”的时代。互联网社交服务网站的发展验证了“六度分隔理论”（Six Degrees of Separation），即“人际关系脉络方面你必然可以通过不超出六位中间人间接与世上任意先生女士相识”。个体的社交圈会不断地扩大和重叠并最终形成大的社交网络。无论是国外的Facebook、MySpace、Twitter，还是国内的开心网、人人网等都一头扎进社交网，因为他们认定社交网必然掀起新一轮的互联网革命。

社交网的一个显著特点是支持巨大用户数，例如Facebook支持超过3亿的用户，其数据中心运行着超过万台的服务器，为遍布全球的用户提供信息通讯服务。另外，任何两个社交网用户都可能交互，也就是必须支持任何两个数据库用户的数据关联操作。这对于服务端的数据库管理提出了极大的挑战。

关系数据库与 NoSQL 数据库

关系数据库使用者遵循一些数据库范式，这些范式是数据库设计中的一系列原理和技术，目的是为了减少数据库中数据冗余和增进数据的一致性。结构化查询语言SQL大量使用多表连接操作，SQL的通用性可以为使用者带来很多方便。

随着越来越多大规模工作负荷应用的发行，对可伸缩性的需求，可能会变得非常迅速和无比庞大。

关系数据库的确能伸缩自如，但通常只能在单台服务器节点上进行。例如采用表分区技术，一个表格可以由多个物理文件组成，虽然表格的容量增大了，但该表格仍然只能由一数据库引擎管理；另外增加一物理文件时，表格Schema得做改动，也就是还不能支持动态扩容。

一旦单节点的能力抵达上限，就得通过多服务器节点来扩展来分发负载。这时关系数据库的复杂性就开始影响其潜在的扩展规模了。RDBMS支持分区视图（Partition View） 技术，也就是支持联合数据库（Federated Database）（如图1）。一个分区视图可以由多个分布在不同数据库节点服务器上的表格组合而成，数据库用户看到的只是该视图，不必关心物理表格。通过数据水平分割技术，分区视图把负载分担到多个数据库节点服务器上。扩容时，该方法除了需改动视图定义外，分区视图成为分布式数据库系统的中心，存在单点故障问题。另外，跨数据库节点之间多表格间连接操作的支持出现极大困难。

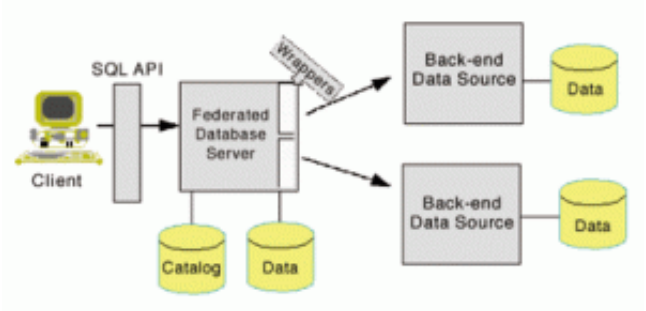

图1 IBM联邦数据库的体系结构

当试图扩展数据库系统到成百上千个节点时，将导致不堪复杂性之重负，这一特点使得RDBMS在大型分布式系统平台市场里的生存能力被大幅削减。

为了能向客户提供一个伸缩自如的空间存放应用数据，供应商实际上只有一种真正的选择——实现一种新型的专注于可扩展性的数据库系统，而牺牲掉关系数据库所带来的其他好处。NoSQL是非关系型数据存储的广义定义，它打破了长久以来关系型数据库与ACID理论大一统的局面。NoSQL数据存储不需要固定的表结构，通常也不存在连接操作，在超大型数据存取上具备关系型数据库无法比拟的性能优势。该术语在2009 年初得到了广泛认同，其中Key-Value数据模型是解决大型数据库系统扩充问题的一种可行的解决方案。

Berkeley DB Key-Value数据模型

Berkeley DB是一种支持Key-Value数据模型的嵌入式数据库存储引擎。它不支持Client/Server网络访问方式，程序通过进程内的API访问数据库，不支持SQL或者其他数据库查询语言，不支持表结构和数据列。访问数据库的程序自主决定数据如何储存在记录里，一条记录由一个称为键（Key）的数据块和一个称为值（Value）的数据块组成。Berkeley DB不对记录里的数据进行任何包装。应用程序可通过回调函数来定义不同键之间的大小关系，记录和它的键都可以达到4GB的长度。尽管架构简单，Berkeley DB却支持很多高级的数据库特性，比如ACID 数据库事务处理、细粒度锁、 XA接口、热备份以及同步复制。Berkley DB为不同用户提供多种功能集（Feature Set）：支持单个写线程的数据存储（Data Store）；支持多并发写线程的并发数据存储（Concurrent Data Store）；支持ACID和灾难恢复的事务数据存储（Transactional Data Store）；通过复制支持容错的高可靠数据存储（High Availability）。

关系数据库系统由存储引擎和关系引擎两个独立部分组成。存储引擎负责记录存储、索引和事务处理，关系引擎负责基于存储引擎提供的服务，分析SQL、制定查询执行计划等。Berkeley DB是一种存储引擎。例如MySQL数据库可采用MyISAM、InnoDB、Berkeley DB等存储引擎，如图2所示。

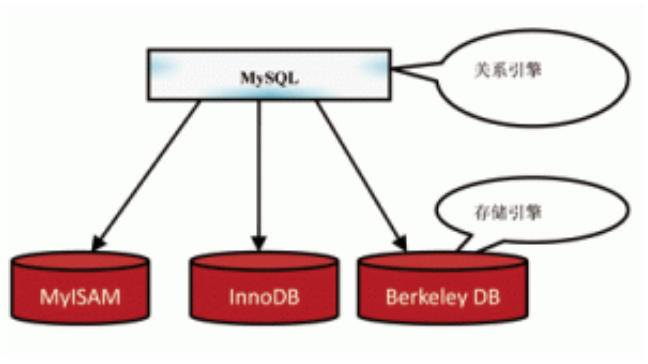

图2 MySQL使用的不同的存储引擎

Berkeley DB支持平衡树（BTree）、哈希（Hash）、队列（Queue）和记录（Record）等数据集存储和索引方式，还支持根据Key-Value中的Key创建集群索引（Clustered Index）。这样记录集的物理次序就根据Key值大小来排列。如果要查询结果记录集的键值为给定的一个范围，该特性对于支持这种类型的快速查询起了很大作用。Berkeley DB的一个Key-Value记录集称为一个数据库，会存储在一个单独文件中。Berkeley DB通过创建辅助数据库（Secondary Database），允许对记录集建立非集群索引（Non-Clustered Index）。非集群索引适用于快速查询结果为一条记录，该记录的键值为给定的一个值。例如社交网用户数据集：

User <UID, First_Name, Last_Name, Icon, E-mail>

如果以UID作为主数据库的键，其他字段作为主数据库的值，可再创建一个辅助数据库，以E-mail作为辅助数据库的键，辅助数据库的值为E-mail所对应的UID，也就是指向主数据库记录的指针。若在一个Key-Value数据库查询，一般可根据查询条件创建成一个键值，引擎返回一个游标（Cursor），该游标指向等于或大于该键值的结果数据集。

不难看出Berkeley DB使用的索引技术与SQL Server、Oracle等高端数据库系统是一样的。RDBMS中经常使用的表格连接操作，在Berkeley DB中不再支持，需要应用程序去实现两个数据集的连接操作。这是Key-Value数据模型与关系数据模型典型的区别。

Berkeley DB除了作为MySQL的存储引擎之外，还应用在OpenLDAP、MemCached等知名软件。

与Berkeley DB类似的数据库引擎还有Tokyo Cabinet/ Tyrant等。

社交网数据库系统Cassandra

以Facebook用户数据集为例，不可能把3亿条数据集存放在同一个表格、文件或由同一台计算机处理，这要求系统能支持数据分区，把数据集分割在多台节点计算机中，每台计算机分担一部分负载，当用户增加到一定程度时，系统能允许加入新的节点计算机，并且尽可能地减少数据迁移。

2007年10月30日，Amazon的CTO Werner Vogels发表了一篇文章，讨论了一种基于Key-Value数据模型的存储系统Dynamo。该系统支撑了不少Amazon的面向电子商务等关键性应用，它采用的存储引擎是 Berkeley DB 事务数据存储（Transactional Data Store）。Dynamo系统主要为存储1M左右甚至更小的内容，如购物车、推荐列表等。Dynamo设计上有下面一些特点。

- 通过数据分区复制来支持高可靠性与高可伸缩性。
- 始终可写。
- 一致性与写入速度的折中，不要求同步写入所有副本。
- 对称，完全去中心化，人工管理工作很小。

Cassandra DB最初由Facebook开发，后来转变成为开源项目。它是一个为网络社交云计算设计的数据库，主要特点是它不再是一个数据库，而是由一堆数据库节点共同构成的一个分布式网络服务，对Cassandra的一个写操作会被复制到其他节点上去，对Cassandra的读操作也会被路由到某个节点上面去读取。对于Cassandra群集来说，扩展性能是比较简单的事情，只管在群集里面添加节点就可以了。

Cassandra的用户包括Facebook、Twitter和Digg等。Digg工程副总裁John Quinn说：“Cassandra采用完全分散的模式，每个节点都一样，不会出现单点故障。它的容错率也非常高，数据可以被复制到数据中心的多个节点中。它还非常具有弹性，随着新设备的加入，其读写吞吐量将呈线性增加。”

Cassandra以Amazon专有的完全分布式的Dynamo为基础，结合了Google BigTable基于列族（Column Family）的数据模型。P2P去中心化的存储。很多方面都可以称之为Dynamo 2.0。

图3为Cassandra、Dynamo、Key-Value之间的关系及在社交网上的应用。箭头表示依赖关系。

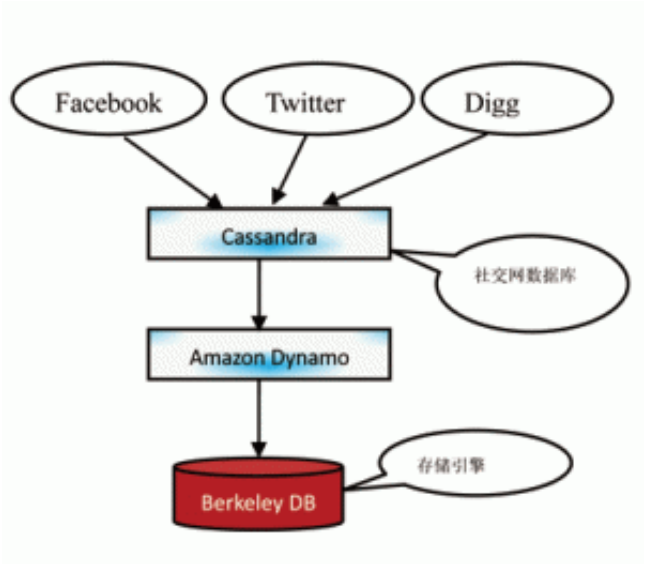

图3 Cassandra、Dynamo、Key-Value关系图

分布式存储系统Dynamo

Dynamo采用Consistent Hashing算法来实现数据分区。

Consistent Hashing基本原理是：首先求出服务器节点的哈希值，并将其配置到0～2^32的圆上。然后用同样的方法求出存储数据的键的哈希值，并映射到圆上。再从数据映射到的位置开始顺时针查找，将数据保存到找到的第一个服务器上。如果超过2^32仍然找不到服务器，就会保存到第一台服务器上。如图4所示。

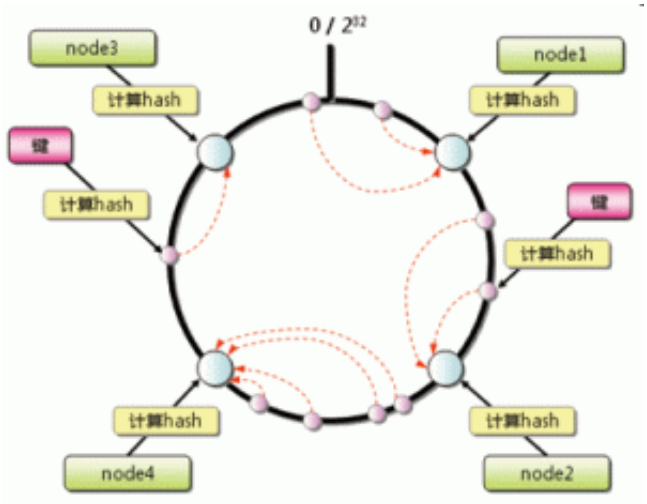

图4 数据分割到4个节点数据库

如果添加一台服务器。只有在圆上，增加服务器的地点逆时针方向的第一台服务器上的部分数据需要迁移到新的节点数据库。如图5所示。

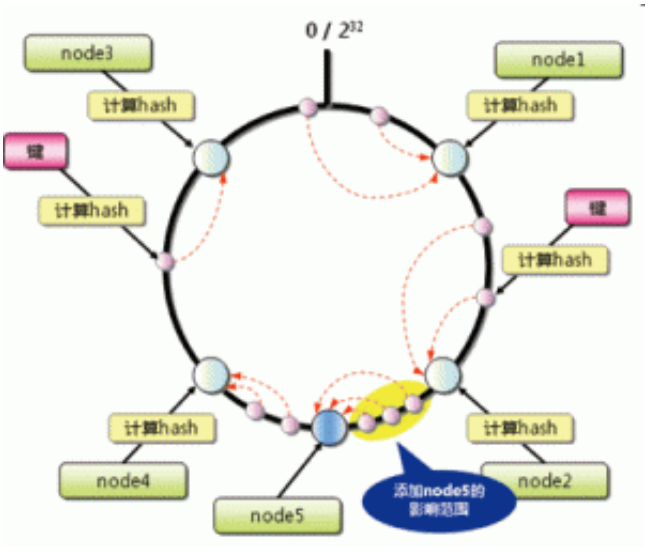

图5 添加Node5后需要迁移的数据

数据分区后，数据块被复制到N个节点上。复制时因为更新产生的一致性问题的维护采取类似拜占庭容错Quorum协议（Byzantine Fault-tolerance Quorum）的机制以及去中心化的复制同步协议。当一个存储节点被认为是拜占庭节点时，它的行为可能任意偏移，表现在：拒绝响应请求、发送错误消息、存储错误信息。Quorum协议中除了N之外还有两个关键参数：R与W。R代表一次成功的读取操作中最小参与节点数量，W代表一次成功的写操作中最小参与节点数量。R和W直接影响性能、一致性。R 和 W 值过小则影响一致性，过大则影响效率，这两个值要平衡。如果W设置为1，则一个实例中只要有一个节点可写就写成功，不会影响写效率；如果R设置为1，只要有一个节点可读，就读成功，不会影响读效率。

Facebook数据库查询语言：FQL

Facebook为开发者提供一套和SQL风格一致的数据库查询语言，称为Facebook Query Language（FQL）。FQL是一种基于列的数据查询语言。提供丰富的条件查询，甚至包括子查询。

例如，以下FQL查询已安装Facebook应用程序的用户$app_user的好友ID集合：

```
SELECT uid FROM user WHERE is_app_user = 1 AND
uid IN （SELECT uid2 FROM friend WHERE uid1 = $app_user）
```

与SQL重要区别是FQL不支持：

- 多表连接：JOIN操作
- 分组： GROUP BY操作
- 排序：ORDER BY操作

随着技术发展，一部分基于列结构的NoSQL数据库开始支持分租、排序等复杂数据统计分析功能。

举例：查询好友信息

Facebook应用程序从以下两个数据集中查找一用户的好友数据集信息：

User <UID,First_Name, Last_Name, Icon>

Friend_List <UID, Friend_UID>

注Friend_UID是一指向User（UID）的外键。RDBMS应用程序可使用数据集连接操作实现：

```
SELECT f.UID, u.Friend_UID, u.First_Name, u.Last_Name, u.Icon
FROM Friend_List f, User u
WHERE f.Friend_UID = u.UID AND
f.UID=@Input_UID
```

在社交网数据库系统中，由于User数据分布在多台服务器中，其连接操作和外键约束实际上不能支持。

在Facebook中查找一用户的好友信息，得分A、B两步操作实现：

A步

```
SELECT Friend_UID
INTO @Out_Record_Set
FROM Friend_List f
WHERE f.UID=@Input_UID
```

B步

```
FOR EACH (Friend_UID in @Out_Record_Set)
SELECT u.Friend_UID, u.First_Name, u.Last_Name, u.Icon
FROM User u
WHERE u.UID = Friend_UID
```

No-SQL: Not Only SQL

对于那些关系复杂的数据处理和分析统计，SQL值得花钱。但当数据库结构非常简单时，SQL可能没有太大用处。如果能用普通文件存储代替数据库系统的话，优选普通文件存储。

对于社交网，能够不受限制的扩展比更丰富的功能更加重要。建立大规模社交网成本的压力让很多社交网开发人员努力去寻找更高性价比的解决方案。研究表明基于普通廉价硬件的分布式存储解决方案比现在的高端数据库更加可靠。支持SQL的RDBMS不能解决所有问题的时候，NoSQL不是简单的No SQL，其本质是Non-Relational，这时候NoSQL也就成为Not Only SQL。

作者简介：

丁艺明，供职于提供计算机安全服务的趋势科技Data Center部门，从事计算机安全、SaaS软件开发，数据库设计。

0
0

原文参见：http://www.mongodb.org/display/DOCS/Tutorial

译文链接：http://chenxiaoyu.org/blog/archives/242

启动数据库

下载 MongoDB, 解压后并启动:

```
$ bin/mongod
```

MongoDB 默认存储数据目录为 /data/db/ (或者 c:\data\db), 当然你也可以修改成不同目录, 只需要指定 –dbpath 参数:

```
$ bin/mongod --dbpath /path/to/my/data/dir
```

获取数据库连接

现在我们就可以使用自带的shell工具来操作数据库了. (我们也可以使用各种编程语言的驱动来使用MongoDB, 自带的shell工具可以方便我们管理数据库)

启动 MongoDB JavaScript 工具:

```
$ bin/mongo
```

默认 shell 连接的是本机localhost 上面的 test库, 会看到:

```
MongoDB shell version: 0.9.8
url: test
connecting to: test
type "help" for help
>
```

“connecting to:” 这个会显示你正在使用的数据库的名称. 想换数据库的话可以:

```
> use mydb
```

可以输入 help 来查看所有的命令.

插入数据到集合

下面我们来建立一个test的集合并写入一些数据. 建立两个对象, j 和 t , 并保存到集合中去.
在例子里 ‘>’ 来表示是 shell 输入提示

```
>j = { name : "mongo" };
{"name" : "mongo"}
>t = { x : 3};
```

```
{ "x" : 3 }
> db.things.save(j);
> db.things.save(t);
> db.things.find();
{ "name" : "mongo" , "_id" : ObjectId( "497cf60751712cf7758fbdbb") }
{ "x" : 3 , "_id" : ObjectId( "497cf61651712cf7758fbdbc") }
>
```

有几点需要注意下 :

- 不需要预先建立一个集合. 在第一次插入数据时候会自动建立.
- 在例子其实可以存储任何结构的数据, 当然在实际应用我们存储的还是相同元素的集合. 这个特性其实可以在应用里很灵活, 你不需要类似alter table 来修改你的数据结构
- 每次插入数据时候对象都会有一个ID, 名字叫 _id.
- 当你运行不同的例子, 你的对象ID值都是不同的.

下面再加点数据:

```
> for( var i=1; i<10; i++ ) db.things.save( { x:4, j:i } ); > db.things.find();
{ "name" : "mongo" , "_id" : ObjectId( "497cf60751712cf7758fbdbb") }
{ "x" : 3 , "_id" : ObjectId( "497cf61651712cf7758fbdbc") }
{ "x" : 4 , "j" : 1 , "_id" : ObjectId( "497cf87151712cf7758fbdbd") }
{ "x" : 4 , "j" : 2 , "_id" : ObjectId( "497cf87151712cf7758fbdbe") }
{ "x" : 4 , "j" : 3 , "_id" : ObjectId( "497cf87151712cf7758fbdbf") }
{ "x" : 4 , "j" : 4 , "_id" : ObjectId( "497cf87151712cf7758fbdc0") }
{ "x" : 4 , "j" : 5 , "_id" : ObjectId( "497cf87151712cf7758fbdc1") }
{ "x" : 4 , "j" : 6 , "_id" : ObjectId( "497cf87151712cf7758fbdc2") }
{ "x" : 4 , "j" : 7 , "_id" : ObjectId( "497cf87151712cf7758fbdc3") }
{ "x" : 4 , "j" : 8 , "_id" : ObjectId( "497cf87151712cf7758fbdc4") }
```

请注意下, 这里循环次数是10, 但是只显示到8, 还有2条数据没有显示.

如果想继续查询下面的数据只需要使用 it 命令, 就会继续下面的数据:

```
{ "x" : 4 , "j" : 7 , "_id" : ObjectId( "497cf87151712cf7758fbdc3") }
{ "x" : 4 , "j" : 8 , "_id" : ObjectId( "497cf87151712cf7758fbdc4") }
```

继续

```
> it
{ "x" : 4 , "j" : 9 , "_id" : ObjectId( "497cf87151712cf7758fbdc5") }
{ "x" : 4 , "j" : 10 , "_id" : ObjectId( "497cf87151712cf7758fbdc6") }
```

从技术上讲 find() 返回一个游标对象. 但在上面的例子里, 并没有拿到一个游标的变量. 所以 shell 自

动遍历游标, 返回一个初始化的set, 并允许我们继续用 it 迭代输出.

当然我们也可以直接用游标来输出, 不过这个是下一部分的内容了.

查询数据

在没有深入查询之前, 我们先看看怎么从一个查询中返回一个游标对象. 可以简单的通过 find() 来查询, 他返回一个任意结构的集合. 如果实现特定的查询稍后讲解.

实现上面同样的查询, 然后通过 while 来输出:

```
> var cursor = db.things.find();
> while (cursor.hasNext()) { print(tojson(cursor.next())); }
{ "name" : "mongo", "_id" : ObjectId("497cf60751712cf7758fbdbb") }
{ "x" : 3, "_id" : ObjectId("497cf61651712cf7758fbdbc") }
{ "x" : 4, "j" : 1, "_id" : ObjectId("497cf87151712cf7758fbdbd") }
{ "x" : 4, "j" : 2, "_id" : ObjectId("497cf87151712cf7758fbdbe") }
{ "x" : 4, "j" : 3, "_id" : ObjectId("497cf87151712cf7758fbdbf") }
{ "x" : 4, "j" : 4, "_id" : ObjectId("497cf87151712cf7758fbdc0") }
{ "x" : 4, "j" : 5, "_id" : ObjectId("497cf87151712cf7758fbdc1") }
{ "x" : 4, "j" : 6, "_id" : ObjectId("497cf87151712cf7758fbdc2") }
{ "x" : 4, "j" : 7, "_id" : ObjectId("497cf87151712cf7758fbdc3") }
{ "x" : 4, "j" : 8, "_id" : ObjectId("497cf87151712cf7758fbdc4") }
{ "x" : 4, "j" : 9, "_id" : ObjectId("497cf87151712cf7758fbdc5") }
>
```

上面的例子显示了游标风格的迭代输出. hasNext() 函数告诉我们是否还有数据, 如果有则可以调用 next() 函数. 这里我们也用了自带的 tojson() 方法返回一个标准的 JSON 格式数据.

当我们使用的是 JavaScript shell, 可以用到JS的特性, forEach 就可以输出游标了. 下面的例子就是使用 forEach() 来循环输出:

forEach() 必须定义一个函数供每个游标元素调用.

```
> db.things.find().forEach( function(x) { print(tojson(x));});
{ "name" : "mongo", "_id" : ObjectId("497cf60751712cf7758fbdbb") }
{ "x" : 3, "_id" : ObjectId("497cf61651712cf7758fbdbc") }
{ "x" : 4, "j" : 1, "_id" : ObjectId("497cf87151712cf7758fbdbd") }
{ "x" : 4, "j" : 2, "_id" : ObjectId("497cf87151712cf7758fbdbe") }
{ "x" : 4, "j" : 3, "_id" : ObjectId("497cf87151712cf7758fbdbf") }
{ "x" : 4, "j" : 4, "_id" : ObjectId("497cf87151712cf7758fbdc0") }
{ "x" : 4, "j" : 5, "_id" : ObjectId("497cf87151712cf7758fbdc1") }
{ "x" : 4, "j" : 6, "_id" : ObjectId("497cf87151712cf7758fbdc2") }
{ "x" : 4, "j" : 7, "_id" : ObjectId("497cf87151712cf7758fbdc3") }
{ "x" : 4, "j" : 8, "_id" : ObjectId("497cf87151712cf7758fbdc4") }
{ "x" : 4, "j" : 9, "_id" : ObjectId("497cf87151712cf7758fbdc5") }
>
```

在 mongo shell 里, 我们也可以把游标当作数组来用 :

```
> var cursor = db.things.find();
> print(tojson(cursor[4]));
{"x" : 4, "j" : 3, "_id" : ObjectId("497cf87151712cf7758fbdbf")}
```

使用游标时候请注意占用内存的问题, 特别是很大的游标对象, 有可能会内存溢出. 所以应该用迭代的方式来输出.

下面的示例则是把游标转换成真实的数组类型:

```
> var arr = db.things.find().toArray();
> arr[5];
{"x" : 4, "j" : 4, "_id" : ObjectId("497cf87151712cf7758fbdc0")}
```

请注意这些特性只是在 mongo shell 里使用, 而不是所有的其他应用程序驱动都支持.

MongoDB 游标对象不是没有快照 – 如果有其他用户在集合里第一次或者最后一次调用 next(), 你可以得不到游标里的数据. 所以要明确的锁定你要查询的游标.

指定条件的查询

到这里我们已经知道怎么从游标里实现一个查询并返回数据对象, 下面就来看看怎么根据指定的条件来查询.

下面的示例就是说明如何执行一个类似SQL的查询, 并演示了怎么在 MongoDB 里实现. 这是在 MongoDB shell 里查询, 当然你也可以用其他的应用驱动或者语言来实现:

```
SELECT * FROM things WHERE name="mongo"
```

```
>db.things.find({name:"mongo"}).forEach(function(x) { print(tojson(x));});
{"name" : "mongo", "_id" : ObjectId("497cf60751712cf7758fbdbb")}
>SELECT * FROM things WHERE x=4> db.things.find({x:4}).forEach(function(x) { print(tojson(x));});
{"x" : 4, "j" : 1, "_id" : ObjectId("497cf87151712cf7758fbdbd")}
{"x" : 4, "j" : 2, "_id" : ObjectId("497cf87151712cf7758fbdbe")}
{"x" : 4, "j" : 3, "_id" : ObjectId("497cf87151712cf7758fbdbf")}
{"x" : 4, "j" : 4, "_id" : ObjectId("497cf87151712cf7758fbdc0")}
{"x" : 4, "j" : 5, "_id" : ObjectId("497cf87151712cf7758fbdc1")}
{"x" : 4, "j" : 6, "_id" : ObjectId("497cf87151712cf7758fbdc2")}
{"x" : 4, "j" : 7, "_id" : ObjectId("497cf87151712cf7758fbdc3")}
{"x" : 4, "j" : 8, "_id" : ObjectId("497cf87151712cf7758fbdc4")}
{"x" : 4, "j" : 9, "_id" : ObjectId("497cf87151712cf7758fbdc5")}
>
```

查询条件是 { a:A, b:B, ... } 类似 “where a==A and b==B and ...”, 更多的查询方式可以参考 Mongo 开发

教程部分.

上面显示的是所有的元素, 当然我们也可以返回特定的元素, 类似于返回表里某字段的值, 只需要在 find({x:4}) 里指定元素的名字, 比如 j:

```
SELECT j FROM things WHERE x=4> db.things.find( { x: 4} , { j: true} ) .forEach( function( x)  {
print( tojson( x) ) ; } ) ;
{ "j" : 1 , "_id" : ObjectId( "497cf87151712cf7758fbdbd") }
{ "j" : 2 , "_id" : ObjectId( "497cf87151712cf7758fbdbe") }
{ "j" : 3 , "_id" : ObjectId( "497cf87151712cf7758fbdbf ") }
{ "j" : 4 , "_id" : ObjectId( "497cf87151712cf7758fbdc0") }
{ "j" : 5 , "_id" : ObjectId( "497cf87151712cf7758fbdc1") }
{ "j" : 6 , "_id" : ObjectId( "497cf87151712cf7758fbdc2") }
{ "j" : 7 , "_id" : ObjectId( "497cf87151712cf7758fbdc3") }
{ "j" : 8 , "_id" : ObjectId( "497cf87151712cf7758fbdc4") }
{ "j" : 9 , "_id" : ObjectId( "497cf87151712cf7758fbdc5") }
>
```

请注意 “_id” 元素会一直被返回.

findOne() – 语法糖

为了方便, mongo shell (其他驱动) 避免游标的可能带来的开销, 提供一个findOne() 函数. 这个函数和 find() 参数一样, 不过他返回游标里第一条数据, 或者返回 null 空数据库.

作为一个例子, name==’mongo’ 可以用很多方法来实现, 可以用 next() 来循环游标(需要校验是否为 null), 或者当做数组返回第一个元素.

但是用 findOne() 方法则更简单和高效:

```
> var mongo = db.things.findOne( { name: "mongo"} ) ;
> print( tojson( mongo) ) ;
{ "name" : "mongo" , "_id" : ObjectId( "497cf60751712cf7758fbdbb") }
>
```

findOne 方法更跟 find({name:”mongo”}).limit(1) 一样.

limit() 查询

你可以需要限制结果集的长度, 可以调用 limit 方法.

这是强烈推荐高性能的原因, 通过限制条数来减少网络传输, 例如:

```
> db.things.find( ) .limit( 3) ;
in cursor for : DBQuery : example.things - >
{ "name" : "mongo" , "_id" : ObjectId( "497cf60751712cf7758fbdbb") }
{ "x" : 3 , "_id" : ObjectId( "497cf61651712cf7758fbdbc") }
```

```
{ "x" : 4 , "j" : 1 , "_id" : ObjectId( "497cf87151712cf7758fbdbd") }
>
```

更多帮助

除非了一般的 help 之外, 你还可以查询 help 数据库和db.whatever 来查询具体的说明.

如果你对一个函数要做什么, 你可以不输入 {{()}} 这些结束的括号则可以输出实现的源码, 例如:

```
> db.foo.insert
function ( obj, _allow_dot) {
    if ( ! obj) {
        throw "no object passed to insert! ";
    }
    if ( ! _allow_dot) {
        this._validateForStorage( obj) ;
    }
    return this._mongo.insert( this._fullName, obj) ;
}
```

mongo 是一个完整的 JavaScript shell程序, 所以在 shell 里完全可以私用JS的方法、类、语法. 此外, MongoDB 定义 很多自己的类和全局变量 (比如 db). 这里可以查看完整的API说明 http://api.mongodb.org/js/.

接下来

看完这篇教程后下一步则看MongoDB更详细的文档 http://www.mongodb.org/display/DOCS/Manual

0

0