NoSQL 数据库
—MongoDB 和 Redis

目 录

NoSQL数据库

—MongoDB和Redis

1NoSQL简述

CAP（Consistency，Availabiity，Partition tolerance)理论告诉我们，一个分布式系统不可能满足一致性，可用性和分区容错性这三个需求，最多只能同时满足两个。关系型数据库通过把更新操作写到事务型日志里实现了部分耐用性，但带来的是写性能的下降 。MongoDB 等 NoSQL 数据库背后蕴涵的哲学是不同的平台应该使用不同类型的数据库，MongoDB 通过降低一些特性来达到性能的提高，这在很多大型站点中是可行的。因为 MongoDB 是非原子性的，所以如果如果应用需要事务，还是需要选择 MySQL 等关系数据库。

NoSQL 数据库，顾名思义就是打破了传统关系型数据库的范式约束。很多 NoSQL 数据库从数据存储的角度看也不是关系型数据库，而是 key-value 数据格式的 hash 数据库。由于放弃了关系数据库强大的 SQL 查询语言和事务一致性以及范式约束，NoSQL 数据库在很大程度上解决了传统关系型数据库面临的诸多挑战。

在社区中，NoSQL 是指“not only sql”，其特点是非关系型，分布式，开源，可水平扩展，模式自由，支持 replication，简单的 API，最终一致性（相对于即时一致性，最终一致性允许有一个“不一致性窗口”，但能保证最终的客户都能看到最新的值）。

2MongoDB简介

mongo 取自“humongous”（海量的），是开源的文档数据库——nosql 数据库的一种。

MongoDB 是一种面向集合（collection）的，模式自由的文档（document）数据库。

面向集合是说数据被分成集合的形式，每个集合在数据库中有惟一的名称，集合可以包含不限数目的文档。除了模式不是预先定义好的，集合与 RDBMS 中的表概念类似，虽然二者并不是完全对等。数据库和集合的创建是“lazy”的，即只有在第一个 document 被插入时集合和数据库才真正创建——这时在磁盘的文件系统里才能看见。

模式自由是说数据库不需要知道存放在集合中的文档的结构，完全可以在同一个集合中存放不同结构的文档，支持嵌入子文档。

文档类似于 RDBMS 中的记录，以 BSON 的格式保存。BSON 是 Binary JSON 的简称，是对 JSON-like 文档的二进制编码序列化。像 JSON（JavaScript Object Notation）一样

BSON 支持在对象和数组内嵌入其它的对象和数组。有些数据类型在 JSON 里不能表示，但可以在 BSON 里表示，如 Date 类型和 BinData（二进制数据），Python 原生的类型都可以表示。与 Protocal Buffers（Google 开发的用以处理对索引服务器请求/应答的协议）相比，BSON 模式更自由，所以更灵活，但这样也使得每个文档都要保存字段名，所以空间压缩上不如 Protocol Buffers。

BSON 第一眼看上去像 BLOB，但 MongoDB 理解 BSON 的内部机制，所以 MongoDB 可以深入 BSON 对象的内部，即使是嵌套的对象，这样 MongoDB 就可以在顶层和嵌套的 BSON 对象上建立索引来应对各种查询了。

MongoDB 可运行在 Linux、Windows 和 OS X 平台，支持 32 位和 64 位应用，默认端口为 27017。推荐运行在 64 位平台，因为 MongoDB 为了提高性能使用了内存映射文件进行数据管理，而在 32 位模式运行时支持的最大文件为 2GB。

MongoDB 查询速度比 MySQL 要快，因为它 cache 了尽可能多的数据到 RAM 中，即使是 non-cached 数据也非常快。当前 MongoDB 官方支持的客户端 API 语言就多达 8 种（C|C++|Java|Javascript|Perl|PHP|Python|Ruby），社区开发的客户端 API 还有 Erlang、Go、Haskell.....

3 术语介绍

数据库、集合、文档

每个 MongoDB 服务器可以有多个数据库，每个数据库都有可选的安全认证。数据库包括一个或多个集合，集合以命名空间的形式组织在一起，用“.”隔开（类似于 JAVA/Python 里面的包），比如集合 blog.posts 和 blog.authors 都处于"blog"下，不会与 bbs.authors 有名称上的冲突。集合里的数据由多个 BSON 格式的文档对象组成，document 的命名有一些限定，如字段名不能以"$"开头，不能有"."，名称"_id"被保留为主键。

如果插入的文档没有提供“_id”字段，数据库会为文档自动生成一个 ObjectId 对象作为“_id”的值插入到集合中。字段“_id”的值可以是任意类型，只要能够保证惟一性。BSON ObjectID 是一个 12 字节的值，包括 4 字节的时间戳，3 字节的机器号，2 字节的进程 id 以及 3 字节的自增计数。建议用户还是使用有意义的“_id”值。

MongoDb 相比于传统的 SQL 关系型数据库，最大的不同在于它们的模式设计（Schema Design）上的差别，正是由于这一层次的差别衍生出其它各方面的不同。

如果将关系数据库简单理解为由数据库、表（table）、记录（record）三个层次概念组成，而在构建一个关系型数据库的时候，工作重点和难点都在数据库表的划分与组织上。一般而言，为了平衡提高存取效率与减少数据冗余之间的矛盾，设计的数据库表都会尽量满足所谓的第三范式。相应的，可以认为 MongoDb 由数据库、集合（collection）、文档对象（Document-oriented、BSON）三个层次组成。MongoDb 里的 collection 可以理解为关系型数

据库里的表，虽然二者并不完全对等。当然，不要期望 collection 会满足所谓的第三范式，因为它们根本就不在同一个概念讨论范围之内。类似于表由多条记录组成，集合也包含多个文档对象，虽然说一般情况下，同一个集合内的文档对象具有相同的格式定义，但这并不是必须的，即 MongoDb 的数据模式是自由的（schema-free、模式自由、无模式），collection 中可以包含具有不同 schema 的文档记录，支持嵌入子文档。

4MongoDB 资源消耗

考虑到性能的原因，mongo 做了很多预分配，包括提前在文件系统中为每个数据库分配逐渐增长大小的文件集。这样可以有效地避免潜在的文件系统碎片，使数据库操作更高效。

一个数据库的文件集从序号 0 开始分配，0，1...，大小依次是 64M，128M，256M，512M，1G，2G，然后就是一直 2G 的创建下去（32 位系统最大到 512M）。所以如果上一个文件是 1G，而数据量刚好超过 1G，则下一个文件（大小为 2G)则可能有超过 90%都是空的。

如果想使磁盘利用更有效率，下面是一些解决方法：

1. 只建立一个数据库，这样最多只会浪费 2G。

2. 每个文档使用自建的“_id”值而不要使用默认的 ObjectId 对象。

3. 由于每个 document 的每个字段名都会存放，所以如果字段名越长，document 的数据占用就会越大，因此把字段名缩短会大大降低数据的占用量。如把“timeAdded”改为“tA”。

Mongo 使用内存映射文件来访问数据，在执行插入等操作时，观察 mongod 进程的内存占用时会发现量很大，当使用内存映射文件时是正常的。并且映射数据的大小只出现在虚拟内存那一列，常驻内存量才反应出有多少数据 cached 在内存中。

【按照 mongodb 官方的说法，mongodb 完全由系统内核进行内存管理，会尽可能的占用系统空闲内存，用 free 可以看到，大部分内存都是作为 io cache 被占用的，而这部分内存是可以释放出来给应用使用的。】

5 交互式 shell

mongo 类似于 MySQL 中的 mysql 进程，但功能远比 mysql 强大，它可以使用 JavaScript 语法的命令从交互式 shell 中直接操作数据库。如查看数据库中的内容，使用游标循环查看查询结果，创建索引，更改及删除数据等数据库管理功能。下面是一个在 mongo 中使用游标的例子：

```
> for(var cur = db.posts.find(); cur.hasNext();) {
... print(tojson(cur.next()));
```

... }
输出：

```
{
    "_id" : ObjectId("4bb311164a4a1b0d84000000"),
    "date" : "Wed Mar 31 17:05:23 2010",
    "content" : "blablablabla",
    "author" : "navygong",
    "title" : "the first blog"
}
```

...其它的 documents。

6 一般功能

6.1 插入

客户端把数据序列化为 BSON 格式传给 DB 后被存储在磁盘上，在读取时数据库几乎不做什么改动直接把对象返回给客户端，由 client 完成 unserialized。如：

```
> doc = {'author': 'joe', 'created': new Date('2010, 6, 21'), 'title':'Yet another blog post', 'text': 'Here is the text...', 'tags': ['example', 'joe'], 'comments': [{'author': 'jim', 'comment': 'I disgree'}, {'author': 'navy', 'comment': 'Good post'}], '_id': 'test_id'}
> db.posts.insert(doc)
```

6.2 查询

基本上你能想到的查询种类 MongoDB 都支持，如等值匹配，<，<=，>， >=，$ne，$in，$mod，$all，$size[1]，$exists，$type[2]，正则表达式匹配，全文搜索，.....。还有 distinct()，sort()，count()，skip()[3]，group()[4]，.....。这里列表的查询中很多用法都和一般的 RDBMS 不同。

[1] 匹配一个有 size 个元素的数组。如 db.things.find({a: {$size: 1}})能够匹配文档{a: ["foo"]}。

[2] 根据类型匹配。db.things.find({a : {$type : 16}})能够匹配所有 a 为 int 类型的文档。BSON 协议中规定了各种类型对应的枚举值。

[3] 指定跳过多少个文档后开始返回结果，可以用在分页中。如：db.students.find().skip((pageNumber-1)*nPerPage).limit(nPerPage).forEach(function(student)

{ print(student.name + "<p>"); })。

[4] 在 sharded MongoDB 配置环境中应该应该使用 map/reduce 来代替 group()。

6.3 删除

可以像查询一样指定条件来删除特定的文档。

6.4 索引

可以像在一般的 RDBMS 中一样使用索引 。提供了建立（一般 、惟一 、组合）索引 、删除索引、重建索引等各种方法。索引信息保存在集合“system.indexes”中。

6.5map/reduce

MongoDB 提供了 map/reduce 方法来进行数据的批处理及聚集操作。和 Hadoop 的使用类似，从集合中接收输入，结果输出到另一个集合。如果你需要使用 group，map/reduce 会是个不错的选择。但 MongoDB 中的索引和标准查询不是使用 map/reduce，而是与 MySQL 相似。

7 模式设计

Mongo 式的模式设计

使用 Mongo 有很多种方式，你本能上可能会像使用关系型数据库一样去使用。当然这样也可以工作得很好，但却没能发挥出 Mongo 的真正威力。Monog 是专门设计为富对象模型（rich object model）使用的。

例如：如果你建立了一个简单的在线商店并且把产品信息存储在关系型数据库中，那你可能会有两个像这样的表：

item

title	price	sku

item_features

sku	feature_name	feature_value

你进行了范式处理因为不同的物品有不同的特征，这样你不用建立一个包含所有特征的表了。在 Mongo 中你也可以像上面那样建立两个集合，但像下面这样存储每种物品会更有效。

```
item : {
      "title" : <title> ,
      "price" : <price> ,
      "sku" : <sku> ,
      "features" : {
        "optical zoom" : <value> ,
        ...
      }
}
```

因为只要查询一个集合就能取得一件物品的所有信息，而这些信息都保存在磁盘上同一个地方，因此大大提高了查询的速度。如果你想插入或更新一种特征，如 db.items.update({ sku : 123 } , { "$set" : { "features.zoom" : "5" } })，也不必在磁盘上移动整个对象，因为 Mongo 为每个对象在磁盘上预留了空间来适应对象的增长。

8 嵌入与引用

以一实例来说，假设需要设计一个小型数据库来存储“学生、地址、科目、成绩”这些信息，那么关系型数据库的设计如图 1 所示，而 key-value 型数据库的设计则可能如图 2 所示。

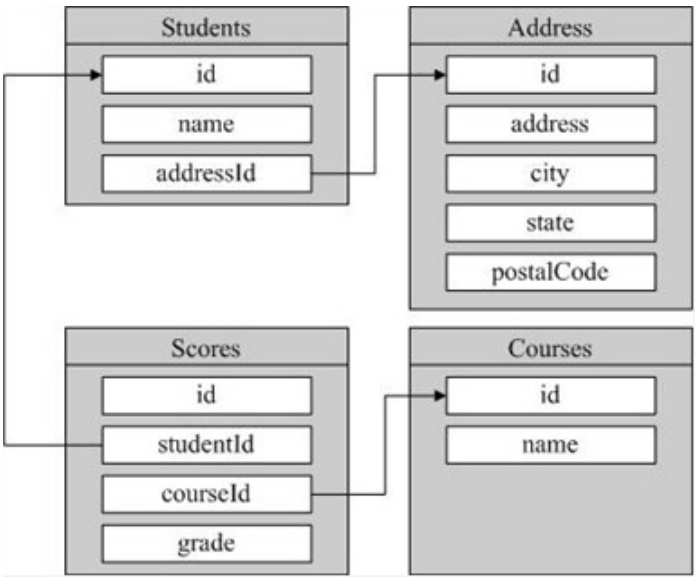


图 1 关系型的数据库设计

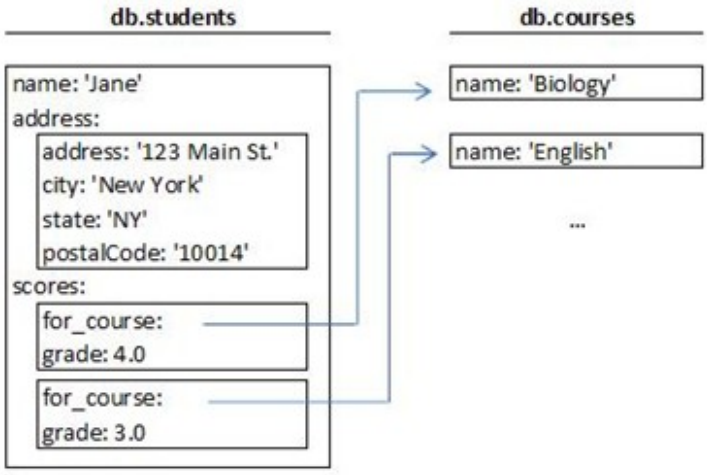


图 2 key-value 型的数据库设计

对比图 1 和图 2，在关系型的数据库设计里划分出了 4 个表，而在 key-value 型的数据库设计里却只有两个集合。如果说集合与表一一对应的话，那么图 2 中应该也有 4 个集合才对，把本应该是集合的 address 和 scores 直接合入了集合 students 中，原因在于在 key-value 型的数据库里，数据模式是自由的。

以 scores 来说，在关系型的数据库设计中将其单独成一个表是因为 student 与 score 是一对多的关系，如果将 score 合入 student 表，那么就必须预留最多可能的字段，这会存在浪费，并且当以后新增一门课程时扩展困难，因此一般都会将 score 表单独出来。而对于 key-value 型的数据库就不同了，其 scores 字段就是一个 BSON，该 BSON 可以只有一个 for_course，也可以有任意多个 for_course，其固有的模式自由特性使得它可以将 score 包含在内而无需另建一个 score 集合。

对于与 student 为一对一关系的 address 表也可以直接合入 student，无需担心 address 的扩展性，当以后需要给 address 新增一个 province 字段，直接在数据插入时加上这个值即可。

当然，对于与 student 成多对多关系 course 表，为了减少数据冗余，可以将 course 建立为一个集合，同关系型的数据库设计中类似。

students 文档中嵌入了 address 文档和 scores 文档，scores 文档的“for_course”字段的值是指向 courses 集合的文档的引用。如果是关系型数据库，需要把“scores”作为一个单独的表，然后在 students 表中建立一个指向“scores”的外键。所以 Mongo 模式设计中的一个关键问题就是“是值得为这个对象新建一个集合呢，还是把这个对象嵌入到其它的集合中”。在关系型数据库中为了范式的要求，每个子项都要建一个单独的表，但在 Mongo 中使用嵌入式对象更有效，所以你应该给出不使用嵌入式对象而单独建一个集合的理由。

为什么说引用要慢些呢，以上面的 students 集合为例，比如执行：

```
print( student.scores[0].for_course.name );
```

如果这是第一次访问 scores[0]，那些客户端必须执行：

```
student.scores[0].for_course = db.courses.findOne({_id:_course_id_to_find_}); //伪代码
```

所以每一次遍历引用都要对数据库进行一次这样的查询，即使所有的数据都在内存中。

再考虑到从客户端到服务器端的种种延迟，这个时间也不会低。

有一些规则可以决定该用嵌入还是引用：

1. 第一个类对象，也就是处于顶层的，往往应该有自己的集合。
2. 排列项详情对象应该用嵌入。
3. 处于被包含关系的应该用嵌入。
4. 多对多的关系通常应该用引用。
5. 数据量小的集合可以放心地做成一个单独的集合，因为整个集合可以很快地 cached。
6. 要想获得嵌入式对象的系统级视图会更困难一些。如上面的“Scores”如果不做成嵌入式对象可以更容易地查询出分数排名前 100 的学生。
7. 如果嵌入的是大对象，需要留意到 BSON 对象的 4M 大小限定（后面会讲到）。
8. 如果性能是关键就用嵌入。

下面是一些示例：

1. Customer/Order/Order Line-Item

cutomers 和 orders 应该做成一个集合，line-items 应该以数组的形式嵌入在 order 中。

2. 博客系统

posts 应该是一个集合；author 可以是一个单独的集合，如果只需记录作者的 email 地址也可以以字段的方式存在于 posts 中；comments 应该做成嵌入的对象。

9GridFS

GridFS 是 MongoDB 中用来存储大文件而定义的一种文件系统。MongoDB 默认是用 BSON 格式来对数据进行存储和网络传输。但由于 BSON 文档对象在 MongoDB 中最大为 4MB，无法存储大的对象。即使没有大小限制，BSON 也无法满足对大数据集的快速范围查询，所以 MongoDB 引进了 GridFS。

9.1GridFS 表示的对象信息

1. 文件对象（类 GridFSFile 的对象）的元数据信息。结构如下

```
{
 "_id" : <unspecified>,              // unique ID for this file
 "filename" : data_string,            // human name for the file
 "contentType" : data_string,         // valid mime type for the object
 "length" : data_number,             // size of the file in bytes
 "chunkSize" : data_number,          // size of each of the chunks.  Default is 256k
 "uploadDate" : data_date,           // date when object first stored
```

"aliases" : data_array of data_string, // optional array of alias strings

"metadata" : data_object, // anything the user wants to store

"md5" : data_string //result of running "filemd5" command on the file's chunks

}

如下是 put 进去的一个文件例子：

{

_id: ObjId(4bbdf6200459d967be9d8e98),

filename: "/home/hjgong/source_file/wnwb.svg",

length: 7429,

chunkSize: 262144,

uploadDate: new Date(1270740513127),

md5: "ccd93f05e5b9912c26e68e9955bbf8b9"

}

2. 数据的二进制块以及一些统计信息。结构如下：

{

"_id": <unspecified>, // object id of the chunk in the _chunks collection

"files_id": <unspecified>, // _id value of the owning {{files}} collection entry

"n": data_number, // "chunk number" - starting with 0

"data": data_binary (type 0x02), // binary data for chunk

}

因此使用 GridFS 可以储存富媒体文件，同时存入任意的附加信息,因为这些信息实际上也是一个普通的 collection。以前，如果要存储一个附件，通常的做法是，在主数据库中存放文件的属性同时记录文件的 path，当查询某个文件时，需要首先查询数据库，获得该文件的 path，然后从存储系统中获得相应的文件。在使用 GridFS 时则非常简单，可以直接将这些信息直接存储到文件中。比如下面的 Java 代码，将文件 file（file 可以是图片、音频、视频等文件）储存到 db 中：

```
public void saveFile(File file, String filename) {
    GridFS f = new GridFS(db);
    try {
        GridFSFile mongofile = f.createFile(file);
        mongofile.put("filename", filename);
        mongofile.put("uploadDate", new Date());
        mongofile.put("contentType", "AVI");
        mongofile.save();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

其中该方法的第一个参数的类型还可以是 InputStream，byte[]，从而实现多个重载的方法。

9.2GridFS 管理

MongoDB 提供的工具 mongofiles 可以从命令行操作 GridFS。如：

./mongofiles -host localhost:1727 -u navygong -p 111 put ~/source_file/wnwb.svg

每种语言提供的 MongoDB 客户端 API 都提供了一套方法，可以像操作普通文件一样对 GridFS 文件进行操作，包括 read()，write()，tell()，seek()等。

10Replication（复制）

Mongo 提供了两种方式的复制：简单的 master-slave 配置及 replica pair 的概念。

如果安全认证被 enable，不管哪种 replicate 方式，都要在 master/slave 中创建一个能为各个 database 识别的用户名/密码。认证步骤如下：

slave 先在 local.system.users 里查找一个名为"repl"的用户，找到后用它去认证 master。如果"repl"用户没有找到，则使用 local.system.users 中的第一个用户去认证。local 数据库和 admin 数据库一样，local 中的用户可以访问整个 db server。

10.1master-slave 模式

一个 server 可以同时为 master 和 slave。一个 slave 可以有多个 master，这种方式并不推荐，因为可能会产生不可预期的结果。

在该模式中，一般是在两个不同的机器上各部署一个 MongDB 实例，一个为 master，另一作为 slave。将 MongoDB 作为 master 启动，只需要在命令行输入：

```
./mongod --master
```

然后主服务进程将会在数据库中创建一个集合 local.oplog.$main，该 collection 主要记录了事务日志，即需要在 slave 执行的操作。

而将 MongoDB 作为 slave 启动，只需要在命令行输入：

```
./mongod --slave --source <masterhostname>[:<port>]
```

port 不指定时即使用默认端口，masterhostname 是 master 的 IP 或 master 机器的 FQDN。

其他配置选项：

--autoresync：自动 sync，但在 10 分钟内最多只会进行一次。

--oplogSize：指定 master 上用于存放更改的数据量，如果不指定，在 32 位机上最少为 50M，在 64 位机上最少为 1G，最大为磁盘空间的 5%。

10.2replica pairs 模式

以这种方式启动后，数据库会自动协商谁是 master 谁是 slave。一旦一个数据库服务器断电，另一个会自动接管，并从那一刻起起为 master。万一另一个将来也出错了，那么 master 状态将会转回给第一个服务器。以这种复制方式启动本地 MongoDB 的命令如下：

```
./mongod --pairwith <remoteserver>  --arbiter <arbiterserver>
```

其中 remoteserver 是 pair 里的另一个 server，arbiterserver 是一个起仲裁作用的 Mongo 数据库服务器，用来协商 pair 中哪一个是 master。arbiter 运行在第三个机器上，利用"平分决胜制"决定在 pair 中的两台机器不能联系上对方时让哪一个做 master，一般是能同 arbiter 通话的那台机器做 master。如果不加--arbiter 选项，出现网络问题时两台机器都作为 master。命令 db.$cmd.findOne({ismaster:1})可以检查当前哪一个 database 是 master。

pair 中的两台机器只能满足最终一致性。当 replica pair 中的一台机器完全挂掉时，需要用一台新的来代替。如(n1, n2)中的 n2 挂掉，这时用 n3 来代替 n2。步骤如下：

1. 告诉 n1 用 n3 来代替 n2：db.$cmd.findOne({replacepeer:1});
2. 重启 n1 让它同 n3 对话：./mongod --pairwith n3 --arbiter <arbiterserver>
3. 启动 n3：./mongod --pairwith n1 --arbiter <arbiterserver>。

在 n3 的数据没有同步到 n1 前 n3 还不能做 master，这个过程长短由数据量的多少决定。

10.3 受限的 master-master 复制

Mongo 不支持完全的 master-master 复制，通常情况下不推荐使用 master-master 模式，但在一些特定的情况下 master-master 也可用。master-master 也只支持最终一致性。配置 master-master 只需运行 mongod 时同时加上--master 选项和--slave 选项。如下：

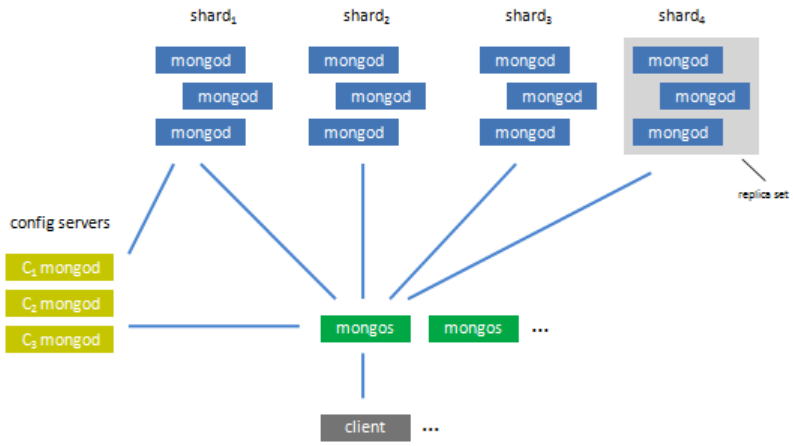
```
$ nohup mongod --dbpath /data1/db --port 27017 --master --slave --source localhost:27018 > /tmp/dblog1 &
$ nohup mongod --dbpath /data2/db --port 27018 --master --slave --source localhost:27017 > /tmp/dblog2 &
```

这种模式对插入、查询及根据_id 进行的删除操作都是安全的，但对同一对象的并发更新无法进行。

11Sharding（分片）

11.1sharding 介绍

MongoDB 包括一个自动分片的的模块（“mongos”），从而可以构建一个大的水平可扩展的数据库集群，可以动态地添加和移走机器。如下是一个数据库集群的示意图：



mongod：数据库服务器进程，类似于 mysqld。

shards：每个 shard 有一个或多个 mongod，通常是一个 master，多个 slave 组成 replication。数据由集合按一个预定的顺序划分，某一个范围的数据被放到一个特定的 shard 中，这样可以通过 shard 的 key 进行有效的范围查询。

shard keys：用于划分集合，格式类似于索引的定义，也是把一个或多个字段作为 key，以 key 来分布数据。如：｛name : 1 （1 代表升序，-1 代表降序）｝、｛ _id : 1 ｝、｛ lastname : 1, firstname : 1 ｝、｛ tag : 1, timestamp : -1 ｝。如果有 100 万人同名，可能还需要划分，因为放到一个块里太大了，这时定义的 shar key 不能只有一个 name 字段了。划分能够保证相邻的数据存储在一个 server（当然也在相同的块上）。

chunks：是一个集合里某一范围的数据，(collection, minkey, maxkey)描述了一个 chunk。块的大小有限定，当块里的数据超过最大值，块会一分为二。如果一个 shard 里的数据过多（添加 shard 时，可以指定这个 shard 上可以存放的最大数据量 maxSize），就会有块迁移到其它的 shard。同样，当添加新的 server 时，为了平衡各个 server 的负载，也会迁移 chunk 过去。

config server（配置服务器）：存储了集群的元信息，包括每一个 shard、一个 shard 里的 server、以及每一个 chunk 的基本信息。其中主要是 chunk 的信息，每个 config server 中都有一份所有 chunk 信息的完全拷贝。使用两阶段提交协议来保证配置信息在 config server 间的一致。mongos：可以认为是一个“数据库路由器”，用以协调集群的各个部分，使它们看起

来像一个系统。mongos 没有固定的状态，可以在 server 需要的时候运行。mongos 启动后会从 config server 里取出元信息，然后接收客户请求，把请求路由到合适的 server，得到结果后送回客户。一个系统可以有多个 mongos 例程，每个例程都需要内存来存储元信息。例程间不需协同工作，每个 mongos 只需要协同 shard servers 和 config servers 工作即可。当然 shard servers 间也会彼此对话，也会同 config servers 对话。

11.2sharding 的配置和管理

mongod 的启动选项中也包含了与 sharding 相关的参数，如--shardsvr（声明这是一个 shard db），--configsvr（声明这是一个 config db）。mongos 的启动选项--configdb 指定 config server 的位置。下面的链接地址是一个简单的 sharding 配置例子：http://www.mongodb.org/display/DOCS/A+Sample+Configuration+Session。

像安全和认证一样，如果要 sharding，先要允许一个数据库 sharding，然后要指定数据库里集合的分片方式，这些都有相应的命令可以完成。

12Java API 简介

要使用 Java 操作 MongoDB，在官网上下载 jar 包，目前最新的版本是：mongo-2.0.jar。首先介绍一下比较常用的几个类：

Mongo：连接服务器，执行一些数据库操作的选项，如新建立一个数据库等；

DB：对应一个数据库，可以用来建立集合等操作；

DBCollection：对应一个集合（类似表），可能是我们用得最多的，可以添加删除记录等；

DBObject 接口和 BasicDBObject 对象：表示一个具体的记录，BasicDBObject 实现了 DBObject，因为是 key-value 的数据结构，所以用起来其实和 HashMap 是基本一致的；

DBCursor：用来遍历取得的数据，实现了 Iterable 和 Iterator。

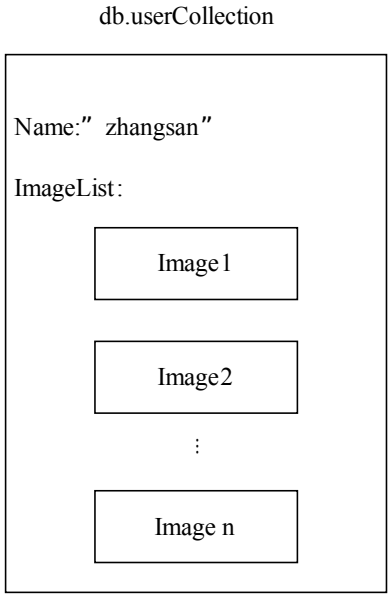
下面以一段简单的例子说明：

```
import com.mongodb.*;

public class Main {
    public static void main(String[] args) {
        try {
            Mongo m = new Mongo("127.0.0.1");
            // 选择数据库，如果没有这个数据库的话，会自动建立
            DB db = m.getDB("mongo");
            // 列出所有数据库名，不过发现，如果数据库里面是没有数据的话，并不会被列出来
            System.out.println("数据库列表: " + m.getDatabaseNames());
            // 建立一个集合，和数据库一样，如果没有，会自动建立
            DBCollection col = db.getCollection("col");
            // 列出所有集合名，和数据库一样，如果集合里面是没有数据的话，并不会被列出来
            System.out.println("当前数据库下的集合列表: " + db.getCollectionNames());
            // 建立一个数据项，重复执行会多次添加一样的数据
            BasicDBObject val = new BasicDBObject();
            val.put("name", "张三");
            col.save(val);
            // 保存的数据为
            // { "_id" : "4b8de2f3053068a371e870c0" , "name" : "张三"}
            // _id是系统自动帮加上的，全局唯一
            // 多次加入一样的数据时，_id都是不一样的
            // 取得所有数据并打印出来
            DBCursor ite = col.find();
            while (ite.hasNext()) {
                System.out.println(ite.next());
            }
        } catch (Exception error) {
            error.printStackTrace();
        }
    }
}
```

13MongoDB 实例分析

下面通过一个实例说明如何用 MongoDB 作为数据库。该实例中有一个 user 实体，包含一个 name 属性，每个 user 对应一到多个图片 image。按照关系型数据库设计，可以设计一个 user 表和一个 image 表，其中 image 表中有一个关联到 user 表的外键。如果将这两个表对应为两个 collection，即 image 对应的 collection 中的每一个 document 都有一个 key，其 value 是该 image 关联的 user。但为了体现 MongoDB 的效率，即 MongoDB 是 schema-free 的，而且支持嵌入子文档，因此在实现时，将一个 user 发布的 image 作为该 user 的子文档嵌入其中，这样只需要定义一个 collection，即 userCollection。如下图所示：



对于图片等文件，可以存储在文件系统中，也可以存储在数据库中。因此下面分两种情况实现。

13.1 图片保存在文件系统中

这种情况下，图片实体中需要记录图片的路径 uri，因此 Image 类的定义如下：

```
import com.mongodb.ReflectionDBObject;
public class Image extends ReflectionDBObject {
    private String uri;
    public String getUri() {
        return uri;
    }
    public void setUri(String uri) {
        this.uri = uri;
    }
}
```

因为在 MongoDB 中，当保存的对象没有设置 ID 时，mongoDB 会默认给该条记录设置一个 ID（"_id"），因此在类中没有定义 id 属性（下同）。

因为一个 user 对应多个 image，所以在 user 实体中需要记录对应的 image。如下：

```
import java.util.ArrayList;
import java.util.List;
import com.mongodb.ReflectionDBObject;
//插入DBCollection中的java对象必需实现DBObject接口，
//ReflectionDBObject实现了DBObject接口，在内部是采用反射来完成的
public class User extends ReflectionDBObject {
    private String name;
    List<Image> imageList = new ArrayList<Image>();

    public String getName() {
        return name;
    }

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

    public List<Image> getImageList() {
        return imageList;
    }

    public void setImageList(List<Image> imageList) {
        this.imageList = imageList;
    }

}
```

在 main 函数中实现如下功能：首先定义一个 user（假设 id 为 1），其对应 3 张图片，然后将该 user 插入 userCollection 中。然后，通过查询查找到该 user（根据 id），再发布第 4 张图片，更新该 user，然后打印出其信息。部分代码如下：

```
Mongo mongo = new Mongo();
//DB类用来表示一个数据库，如果数据库不存在则创建一个
DB db = mongo.getDB("test");
//DBCollection类用来存放对象，若不存在则创建一个
DBCollection coll = db.getCollection("userCollection");
DBCursor cur = null;
List<Image> imgList = new ArrayList<Image>();

//顶级class
User user = new User();
user.set_id(1);
user.setName("zhangsan");

for (int i = 1; i <= 3; i++) {//3张图片
    Image image = new Image();
    image.set_id(i);
    image.setUri("file/image/" + i + ".jpg");
    imgList.add(image);
}
user.setImageList(imgList);
coll.insert(user);//将user插入collection中
out("添加3幅图片后：");
cur = coll.find();
while (cur.hasNext()) {
    out(cur.next());//打印出user的信息
}
```

```
BasicDBObject query = new BasicDBObject();
query.put("_id", 1);
cur = coll.find(query);//查询id为1的user
BasicDBObject user1 = null;
while (cur.hasNext()) {
    user1 = (BasicDBObject) cur.next();
}

if (user1 != null) {
    List<Image> list = new ArrayList<Image>();
    list = (List) user1.get("ImageList");
    Image image = new Image();
    image.set_id(4);//插入第4幅图片
    image.setUri("file/image/4.jpg");
    list.add(image);
    user.setImageList(imgList);
    coll.save(user1); //保存更新至user1
}

out("添加第4幅图片后：");
cur = coll.find();
while (cur.hasNext()) {
    out(cur.next());//打印出插入第4幅图片后user的信息
}
```

程序运行后，在控制台打印出的信息如下：

添加3幅图片后：

```
{ "_id" : 1 , "ImageList" : [ { "Uri" : "file/image/1.jpg" , "_id" : 1} , { "Uri" :
"file/image/2.jpg" , "_id" : 2} , { "Uri" : "file/image/3.jpg" , "_id" : 3}] , "Name" :
"zhangsan"}
```

添加第4幅图片后：

```
{ "_id" : 1 , "ImageList" : [ { "Uri" : "file/image/1.jpg" , "_id" : 1} , { "Uri" :
"file/image/2.jpg" , "_id" : 2} , { "Uri" : "file/image/3.jpg" , "_id" : 3} , { "Uri" :
"file/image/4.jpg" , "_id" : 4}] , "Name" : "zhangsan"}
```

从该结果容易看出，用户 user 有两个属性“_id”和“Name”，而且 ImageList 作为其子文档（数组）嵌入其中，该数组中是 3 个图片，每个图片仍然是 bson 格式。

13.2 图片保存在数据库中

这种情况下，图片实体只需要存储文件名即可，因此 Image2 类的定义如下：

```
public class Image2 extends ReflectionDBObject {
    private String fileName;

    public String getFileName() {
        return fileName;
    }
    public void setFileName(String fileName) {
        this.fileName = fileName;
    }
}
```

User2 类和上面类似，如下所示：

```
//插入DBCollection中的java对象必需实现DBObject接口，
//ReflectionDBObject实现了DBObject接口，在内部是采用反射来完成的
public class User2 extends ReflectionDBObject {

    private String name;
    List<Image2> imageFileList = new ArrayList<Image2>();

    public String getName() {
        return name;
    }

    public void setName(String name) {
```

实现了类 MongoTest2，其功能仍然是一个 user 对应 3 个图片，存入数据库中后，通过查询得到该 user 后，再插入第 4 幅图片，然后打印出信息。同时为了演示文件的查询，对存入 MongoDB 中的图片进行了查询并打印出其部分元数据信息。部分代码如下所示：

```
static Mongo mongo = null;
static DB db = null;
static String basepath = MongoTest2.class.getResource("/").getPath();

public MongoTest2() {
    try {
        mongo = new Mongo();
    } catch (UnknownHostException e) {
        e.printStackTrace();
    } catch (MongoException e) {
        e.printStackTrace();
    }
    db = mongo.getDB("test");
}

public static void out(Object o) {
    System.out.println(o);
}

public static void out() {
    System.out.println();
}
```

```
//DBCollection类用来存放对象，若不存在则创建一个
DBCollection coll = db.getCollection("useCollection2");
DBCursor cur = null;
List<Image2> imgFileList = new ArrayList<Image2>();

//顶级class
User2 user = new User2();
user.set_id(1);
user.setName("zhangsan");

for (int i = 1; i <= 3; i++) {
    Image2 image = new Image2();
    image.set_id(i);
    image.setFileName(i + ".jpg");
    String filePath = basepath + "file/" + i + ".jpg";
    File file = new File(filePath);
    if (file.exists()) {
        System.out.println("文件" + i + "存在");
        try {
            saveFile(file, i, i + ".jpg","jpg");//将文件保存到数据库中
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    imgFileList.add(image);
}
user.setImageFileList(imgFileList);
coll.insert(user);
out("添加3幅图片后：");
cur = coll.find();
while (cur.hasNext()) {
    out(cur.next());
}
```

```
    BasicDBObject query = new BasicDBObject();
    query.put("_id", 1);
    cur = coll.find(query);//查找id为1的用户
    BasicDBObject user1 = null;
    while (cur.hasNext()) {
        user1 = (BasicDBObject) cur.next();
    }

    if (user1 != null) {
        List<Image2> list = new ArrayList<Image2>();
        list = (List) user1.get("ImageFileList");
        Image2 image = new Image2();
        image.set_id(4);//插入第4幅图片
        image.setFileName("4.jpg");
        String filePath = basepath + "file/4.jpg";
        File file = new File(filePath);
        if (file.exists()) {
            System.out.println("文件4存在");
            try {
                saveFile(file, 4, "4.jpg","jpg");//将文件保存到数据库中
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
        list.add(image);
        user.setImageFileList(imgFileList);
        coll.save(user1); //保存更新至user1
    }

    out("添加第4幅图片后：");
    cur = coll.find();
    while (cur.hasNext()) {
        out(cur.next());
    }

    out("查找数据库中的图片文件，并打印出图片信息：");
    GridFS f = new GridFS(db);
    List<GridFSDBFile> flist = f.find("1.jpg");//根据文件名查找文件
    GridFSDBFile img = flist.get(0);
    out("图片1的id：" + img.getId());
    out("图片1的文件名：" + img.getFilename());
    out("图片1的上传时间：" + img.getUploadDate());
    out("图片1的类型：" + img.getContentType());

public static void saveFile(File file, int id, String filename ,String type) {
    GridFS f = new GridFS(db);
    try {
        GridFSFile mongofile = f.createFile(file);
        //设置元数据信息
        mongofile.put("_id", id);
        mongofile.put("filename", filename);
        mongofile.put("uploadDate", new Date());
        mongofile.put("contentType", type);
        mongofile.save();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

运行程序，控制台打印出的结果如下：

```
文件1存在
文件2存在
文件3存在
添加3幅图片后：
{ "_id" : 1 , "ImageFileList" : [ { "FileName" : "1.jpg" , "_id" : 1} , { "FileName" : "2.jpg"
, "_id" : 2} , { "FileName" : "3.jpg" , "_id" : 3}] , "Name" : "zhangsan"}

文件4存在
添加第4幅图片后：
{ "_id" : 1 , "ImageFileList" : [ { "FileName" : "1.jpg" , "_id" : 1} , { "FileName" : "2.jpg"
, "_id" : 2} , { "FileName" : "3.jpg" , "_id" : 3} , { "FileName" : "4.jpg" , "_id" : 4}] ,
"Name" : "zhangsan"}

查找数据库中的图片文件，并打印出图片信息：
图片1的id：1
图片1的文件名：1.jpg
图片1的上传时间：Wed Jun 23 11:23:46 CST 2010
图片1的类型：jpg
```

14MongoDB 常用 API 总结

- 类转换

当把一个类对象存到 mongoDB 后，从 mongoDB 取出来时使用 setObjectClass()将其转换回原来的类。

```
public class Tweet implements DBObject {
   /* ... */
}
Tweet myTweet = new Tweet();
myTweet.put("user", "bruce");
myTweet.put("message", "fun");
myTweet.put("date", new Date());
collection.insert(myTweet);
//转换
collection.setObjectClass(Tweet.class);
Tweet myTweet = (Tweet)collection.findOne();
```

- 默认 **ID**

当保存的对象没有设置 ID 时，mongoDB 会默认给该条记录设置一个 ID（"_id"）。

当然你也可以设置自己指定的 ID，如：(在 mongoDB 中执行用 db.users.save({_id:1,name:'bruce'});)

```
BasicDBObject bo = new BasicDBObject();
```

```
bo.put('_id', 1);
bo.put('name', 'bruce');
collection.insert(bo);
```

- 权限

判断是否有 mongoDB 的访问权限，有就返回 true，否则返回 false。

```
boolean auth = db.authenticate(myUserName, myPassword);
```

- 查看 **mongoDB** 数据库列表

```
Mongo m = new Mongo();
for (String s : m.getDatabaseNames()) {
System.out.println(s);
}
```

- 查看当前库下所有的表名，等于在 **mongoDB** 中执行 **show tables;**

```
Set<String> colls = db.getCollectionNames();
for (String s : colls) {
System.out.println(s);
}
```

- 查看一个表的索引

```
List<DBObject> list = coll.getIndexInfo();
for (DBObject o : list) {
System.out.println(o);
}
```

- 删除一个数据库

```
Mongo m = new Mongo();
m.dropDatabase("myDatabaseName");
```

- 建立 **mongoDB** 的链接

```
Mongo m = new Mongo("localhost", 27017); //有多个重载方法，可根据需要选择
DB db = m.getDB("myDatabaseName"); //相当于库名
DBCollection coll = db.getCollection("myUsersTable"); //相当于表名
```

查询数据

- 查询第一条记录

DBObject firstDoc = coll.findOne();

findOne()返回一个记录，而 find()返回的是 DBCursor 游标对象。

- 查询全部数据

```
DBCursor cur = coll.find();
while(cur.hasNext()) {
System.out.println(cur.next());
}
```

- 查询记录数量

```
coll.find().count();
coll.find(new BasicDBObject("age", 26)).count();
```

- 条件查询

```
BasicDBObject condition = new BasicDBObject();
condition.put("name", "bruce");
condition.put("age", 26);
coll.find(condition);
```

- 查询部分数据块

```
DBCursor cursor = coll.find().skip(0).limit(10);
while(cursor.hasNext()) {
System.out.println(cursor.next());
}
```

- 比较查询**(age > 50)**

```
BasicDBObject condition = new BasicDBObject();
condition.put("age", new BasicDBObject("$gt", 50));
coll.find(condition);
```

比较符

"$gt"： 大于

"$gte"：大于等于

"$lt"： 小于

"$lte"：小于等于

"$in"： 包含

//以下条件查询 20<age<=30

condition.put("age", new BasicDBObject("$gt", 20).append("$lte", 30));

插入数据

- 批量插入

```
List datas = new ArrayList();
for (int i=0; i < 100; i++) {
BasicDBObject bo = new BasicDBObject();
bo.put("name", "bruce");
bo.append("age", i);
datas.add(bo);
}
coll.insert(datas);
```

又如：

```
DBCollection coll = db.getCollection("testCollection");
for(int i=1; i<=100; i++) {//插入 100 条记录
   User user = new User();
   user.setName("user_"+i);
   user.setPoint(i);
   coll.insert(user);
}
```

- 正则表达式

查询所有名字匹配 /joh?n/i 的记录

```
Pattern pattern = Pattern.compile("joh?n", CASE_INSENSITIVE);
BasicDBObject query = new BasicDBObject("name", pattern);
DBCursor cursor = coll.find(query);
```

15Redis 简介

Redis 是一个 key-value 类型的内存数据库，每一个 key 都与一个 value 关联，使得 Redis 与其他 key-value 数据库不同是因为在 Redis 中的每一个 value 都有一个类型

（type），目前在 Redis 中支持 5 中数据类型：String 、List 、Set 、ZSet 和 Hash 。每一种类型决定了可以赋予其上的操作（这些操作成为命令 command）。比如你可以使用 LPUSH 或 RPUSH 命令在 O(1)时间对一个 list 添加一个元素，然后你可以使用 LRANGE 命令得到 list 中的一部分元素或使用 LTRIM 对该 list 进行 trim 操作。集合 set 操作也是很灵活的，你可以从 set（无序的 String 的集合）中 add 或 remove 元素，还可以进行交集 、合集和差集运算 。每一个 command 都是服务端自动的操作。

Redis 性能上和 memcached 一样快但提供了更多的特性。和 memcached 一样，Redis 支持对 key 设置失效时间，因此当设定的时间过后会被自动删除。

15.1Redis 特性

- 速度快

  Redis 使用标准 C 编写实现，而且将所有数据加载到内存中，所以速度非常快。官方提供的数据表明，在一个普通的 Linux 机器上，Redis 读写速度分别达到 81000/s 和 110000/s。
- 持久化

  由于所有数据保持在内存中（2.0 版本开始可以只将部分数据的 value 放在内存，见“虚拟内存”），所以对数据的更新将异步地保存到磁盘上，Redis 提供了一些策略来保存数据，比如根据时间或更新次数。
- 数据结构

  可以将 Redis 看做“数据结构服务器”。目前，Redis 支持 5 种数据结构。
- 自动操作

  Redis 对不同数据类型的操作是自动的，因此设置或增加 key 值，从一个集合中增加或删除一个元素都能安全的操作。
- 支持多种语言

  Redis 支持多种语言，诸如 Ruby, Python, Twisted Python, PHP, Erlang, Tcl, Perl, Lua, Java, Scala, Clojure 等 。对 Java 的支持，包括两个 。一是 JDBC-Redis，是使用 JDBC 连接 Redis 数据库的驱动 。这个项目的目标并不是完全实现 JDBC 规范，因为 Redis 不是关系型数据库，但给 Java 开发人员提供了一个像操作关系数据库一样操作 Redis 数据库的接口。

  另一个是 JRedis，是使用 Redis 作为数据库开发 Java 程序的开发包，主要提供了对数据结构的操作。
- 主-从复制

  Redis 支持简单而快速的主-从复制。官方提供了一个数据，Slave 在 21 秒即完成了对 Amazon 网站 10G key set 的复制。

- Sharding
  很容易将数据分布到多个 Redis 实例中，但这主要看该语言是否支持。目前支持 Sharding 功能的语言只有 PHP、Ruby 和 Scala。

16Redis 数据类型

官方文档上，将 Redis 成为一个“数据结构服务器”（data structures server）是有一定道理的。Redis 的所有功能就是以其固有的几种数据结构保存，并提供用户操作这几种结构的接口。可以对比在其他语言中那些固有数据类型及其操作。
Redis 目前提供四种数据类型：string、list、set 和 zset（sorted set）。

16.1String 类型

String 是最简单的类型，一个 key 对应一个 value。Redis String 是安全的，String 类型的数据最大 1G。String 类型的值可以被视作 integer，从而可以让“INCR”命令族操作，这种情况下，该 integer 的值限制在 64 位有符号数。
在 list、set 和 zset 中包含的独立的元素类型都是 Redis String 类型。
在 Redis 中，String 类型由 sds.c 库定义，它被封装成 Redis 对象。和 Java 中的对象一样，Redis 对象也是使用“引用”，因此当一个 Redis String 被多次使用时，Redis 会尽量使用同一个 String 对象而不是多次分配。
从 Redis 1.1 版开始，String 对象可以编码成数字，因此这样可以节省内存空间。

16.2List 类型

链表类型，主要功能是 push、pop、获取一个范围的所有值等。其中的 key 可以理解为链表的名字。在 Redis 中，list 就是 Redis String 的列表，按照插入顺序排序。比如使用 LPUSH 命令在 list 头插入一个元素，使用 RPUSH 命令在 list 的尾插入一个元素。当这两个命令之一作用于一个空的 key 时，一个新的 list 就创建出来了。比如：

```
LPUSH mylist a   # now the list is "a"
LPUSH mylist b   # now the list is "b","a"
RPUSH mylist c   # now the list is "b","a","c" (RPUSH was used this time)
```

最终在 mylist 中存储的元素为：”b”,”a”,”c”。
List 的最大长度是 2^32-1 个元素。

16.3Set 类型

集合，和数学中的集合概念相似。操作中的 key 理解为集合的名字。在 Redis 中，set 就是 Redis String 的无序集合，不允许有重复元素。对 set 操作的 command 一般都有返回值标识所操作的元素是否已经存在。比如 SADD 命令是往 set 中插入一个元素，如果 set 中已存在该元素，则命令返回 0，否则返回 1。
Set 的最大元素数是 2^32-1。
Redis 中对 set 的操作还有交集、并集、差集等。

16.4ZSet 类型

Zset 是 set 的一个升级版本，在 set 的基础上增加了一个顺序属性，这一属性在添加修改元素时可以指定，每次指定后 zset 会自动安装指定值重新调整顺序。可以理解为一张表，一列存 value，一列存顺序。操作中的 key 理解为 zset 的名字。
比如 ZADD 命令是向 zset 中添加一个新元素，对该元素要指定一个 score。如果对已经在 zset 中存在的元素施加 ZADD 命令同时指定了不同的 score，则该元素的 score 将更新同时该元素将被移动到合适的位置以使集合保持有序。
使用 ZRANGE 命令可以得到 zset 的一部分元素，当然也可以使用命令 ZRANGEBYSCORE，根据 score 得到或删除一部分元素。
Zset 的最大元素数是 2^32-1。
对于已经有序的 zset，仍然可以使用 SORT 命令，通过指定 ASC|DESC 参数对其进行排序。

16.5Hash 类型

Redis Hash 类型对数据域和值提供了映射，这一结构很方便表示对象。此外，在 Hash 中可以只保存有限的几个“域”，而不是将所有的“域”作为 key，这可以节省内存。这一特性的体现可以参考下文的“虚拟内存”部分。

17All data in memory, but saved on disk

Redis 在内存中加载并维护整个数据集，但这些数据是持久化的，因为它们在同一时间被保持在磁盘上，所以当 Redis 服务重启时，这些数据能够重新被加载到内存中。

Redis 支持两种数据持久化的方法。一种称为 snapsshotting，在这种模式下，Redis 异步地将数据 dump 到磁盘上。Redis 还可以通过配置，根据更新操作次数或间隔时间，将数据 dump 到磁盘。比如你可以设定发生 1000 个更新操作时，或距上次转储最多 60s 就要将数据 dump 到磁盘。

因为数据是异步 dump 的，所以当系统崩溃时，就会出现错误。因为，Redis 提供了另外一种模式，更安全的持久化模式，称为 Append Only File，当出现修改数据命令的地方，这些命令就要写到“append only file”—ASAP。当服务重启时，这些命令会重新执行（replay）从而在内存中重新构建数据。

Redis 的存储可以分为：内存存储、磁盘存储和 log 文件三部分，配置文件（redis.conf）中有三个参数对其进行配置。

save <seconds> <changes>：save 配置，指出在多长时间，有多少次更新操作，就将数据同步到数据文件。在默认的配置文件中设置就设置了三个条件。

appendonly yes/no：appendonly 配置，指出是否在每次更新操作后进行日志记录，如果不开启，可能会在断电时导致一段时间内的数据丢失。因为 redis 本身同步数据文件是按上面的 save 条件来同步的，所以有的数据会在一段时间内只存在于内存中。

appendfsync no/always/everysec：appendfsync 配置，no 表示等操作系统进行数据缓存同步到磁盘，always 表示每次更新操作后手动调用 fsync()将数据写到磁盘，everysec 表示每秒同步一次。

18Redis 的 Master-Slave 模式

Redis 中配置 Master-Slave 模式很简单，二者会自动同步。配置方法是在从机的配置文件中指定 slaveof 参数为主机的 ip 和 port 即可，比如：**slaveof 192.168.2.201 6379**。从原理上来说，是从机请求主机的方式，按照 tokyotyrant 的做法，是可以实现主-主，主-从等灵活形式的，而 redis 只能支持主-从，不能支持主-主。Redis 中的复制具有以下特点：

- 一个 master 可以有多个 slave。
- 一个 slave 可以接收其他 slave 的链接，即不仅仅能连接一个 master 所属的 slaves，而且能连接到 slave 的 slave，从而形成了“图”结构。
- 在复制进行时，master 是“非阻塞”的。即一个或多个 slave 对 master 进行第一次同步时，master 仍然可以提供查询服务，而对于 slave，当复制进行时则是“阻塞”的，期间 slave 不能响应查询。
- 复制可以用于提高数据的扩展性（比如多个 slave 用于只读查询），或者仅用于数据备份。
- 使用复制可以避免在 master 存储数据，通过修改 redis.conf 文件（将“save”的 al 注释），这样数据存储只在 slave 端进行。

slave 连接 master 并发出 SYNC 命令开始复制或当连接关闭时与 master 同步数据 。master 在“后台”进行存储操作，同时会监听所有对数据的更新操作。当存储操作完成后，master 开始向 slave 传送数据并将数据保存到磁盘，加载到内存。这一过程中，master 将会向 slave 发送所有对数据有更新操作的命令 。通过 telnet 可以实验这一过程 。通过连接到 Redis 端口并且发出 SYNC 命令，在 telnet 的 session 中，可以看到数据包的传送，以及发送给 master 的命令都将重新发送给 slave。

当 master-slave 连接断掉时，slave 能自动重新建立连接。当一个 master 并发地收到多个 slave 数据同步请求时，master 也能自动处理。

19Redis 虚拟内存管理

在 Redis2.0 开始（目前最新版本）第一次提出了 Virtual Memory（VM）的特性。Redis 是内存数据库，因此通常情况下 Redis 会将所有的 key 和 value 都放在内存中，但有时这并不是最好的选择，为了查询速度，可以将所有的 key 放在内存中，而 values 可以放在磁盘上，当用到时再交换到内存。

比如你的数据有 100000 个 key 都放在了内存中，而只有其中 10%的 key 被经常访问，那么可进行 VM 配置的 Redis 会尝试将不经常访问的 key 关联的 value 放到磁盘上，当这些 value 被请求时才会交换到内存中。

什么时候使用 VM 配置是需要考虑的问题。因为 Redis 是以磁盘为后备的内存数据库，在大多数情况下，只有 RAM 足够大时才使用 Redis。但实际情况是，总会出现内存不够的情况：

- 有“倾向”的数据访问 。只有很小比例的 key 被经常访问（比如一个网站的在线用户），但同时在内存中却有所有的数据。
- 不考虑访问情况以及大的 value，内存不能加载所有数据 。这种情况下，Redis 可以视作 on-disk DB，即所有的 key 在内存中，而访问 value 时要访问慢速的磁盘。

需要注意的一点是，Redis 不能交换 key。即如果是因为数据的 key 太大，value 太小而造成内存不够，VM 是不能解决这种情况的。当出现这种情况时，有时可以使用 Hash 结构，将“many keys with small values”的问题转换成“less keys but with very values”的问题（对 key 进行 hash，从而对数据进行“分组”）。

启用 VM 功能，只需要在 redis.conf 文件中使 vm-enabled 的值为 yes。

20Redis 实例分析

仍然以用户和图片的例子进行说明 。假设将图片存储在文件系统中，即类 Image 中只保存图

片的 uri。类 Image 和类 User 的代码参见 MongoDB 部分的例子。下面给出测试类的关键代码：

```
// 创建jredis客户端。基本上操作都是这个接口提供的
JRedis jredis = new JRedisClient("192.168.2.204 ", 6379);
jredis.ping();
User user = new User();
user.setName("zhangsan");
List<Image> imgList = new ArrayList<Image>();
for (int i = 1; i <= 3; i++) {
    Image img = new Image();
    img.setUri("image uri " + i);
    imgList.add(img);
}
user.setImageList(imgList);
jredis.sadd("userSet", user); //将user插入userSet中
out("插入数据完毕");

List<User> members = DefaultCodec.decode(jredis.smembers("userSet"));
for (User u : members) {
    out(u.getName());
    out("该用户图片数："+u.getImageList().size());
}

jredis.quit();
```

在实现中，将类 User 的对象实例存放在一个 set 中，所以利用了 SADD 命令，在 Java 中即调用 JRedis 对象的 sadd 方法。smembers 方法将返回指定 key（这里是 userSet）的集合中的所有 value，这样就可以对每一个元素进行操作。

21Redis 命令总结

Redis 提供了丰富的命令（command）对数据库和各种数据类型进行操作，这些 command 可以在 Linux 终端使用。在编程时，比如使用 Redis 的 Java 语言包，这些命令都有对应的方法，比如上面例子中使用的 sadd 方法，就是对集合操作中的 SADD 命令。下面将 Redis 提供的命令做一总结。

21.1 连接操作相关的命令

- **quit**：关闭连接（connection）
- **auth**：简单密码认证

21.2 对 value 操作的命令

- **exists(*key*)**：确认一个 key 是否存在
- **del(*key*)**：删除一个 key
- **type(*key*)**：返回值的类型

- **keys(*pattern*)**：返回满足给定 pattern 的所有 key
- **randomkey**：随机返回 key 空间的一个 key
- **rename(*oldname*, *newname*)**：将 key 由 oldname 重命名为 newname，若 newname 存在则删除 newname 表示的 key
- **dbsize**：返回当前数据库中 key 的数目
- **expire**：设定一个 key 的活动时间（s）
- **ttl**：获得一个 key 的活动时间
- **select(*index*)**：按索引查询
- **move(*key*, *dbindex*)**：将当前数据库中的 key 转移到有 dbindex 索引的数据库
- **flushdb**：删除当前选择数据库中的所有 key
- **flushall**：删除所有数据库中的所有 key

21.3 对 String 操作的命令

- **set(*key, value*)**：给数据库中名称为 key 的 string 赋予值 value
- **get(*key*)**：返回数据库中名称为 key 的 string 的 value
- **getset(*key, value*)**：给名称为 key 的 string 赋予上一次的 value
- **mget(*key1, key2,…, key N*)**：返回库中多个 string（它们的名称为 key1，key2…）的 value
- **setnx(*key, value*)**：如果不存在名称为 key 的 string，则向库中添加 string，名称为 key，值为 value
- **setex(*key*, *time*, *value*)**：向库中添加 string（名称为 key，值为 value）同时，设定过期时间 time
- **mset(*key1, value1, key2, value2,…key N, value N*)**：同时给多个 string 赋值，名称为 key *i* 的 string 赋值 value *i*
- **msetnx(*key1, value1, key2, value2,…key N, value N*)**：如果所有名称为 key *i* 的 string 都不存在，则向库中添加 string，名称 key *i* 赋值为 value *i*
- **incr(*key*)**：名称为 key 的 string 增 1 操作
- **incrby(*key, integer*)**：名称为 key 的 string 增加 integer
- **decr(*key*)**：名称为 key 的 string 减 1 操作
- **decrby(*key, integer*)**：名称为 key 的 string 减少 integer
- **append(*key, value*)**：名称为 key 的 string 的值附加 value
- **substr(*key, start, end*)**：返回名称为 key 的 string 的 value 的子串

21.4 对 List 操作的命令

- **rpush(*key, value*)**：在名称为 key 的 list 尾添加一个值为 value 的元素
- **lpush(*key, value*)**：在名称为 key 的 list 头添加一个值为 value 的 元素
- **llen(*key*)**：返回名称为 key 的 list 的长度
- **lrange(*key, start, end*)**：返回名称为 key 的 list 中 start 至 end 之间的元素（下标从 0 开始，下同）
- **ltrim(*key, start, end*)**：截取名称为 key 的 list，保留 start 至 end 之间的元素
- **lindex(*key, index*)**：返回名称为 key 的 list 中 index 位置的元素
- **lset(*key, index, value*)**：给名称为 key 的 list 中 index 位置的元素赋值为 value
- **lrem(*key, count, value*)**：删除 count 个名称为 key 的 list 中值为 value 的元素。count 为 0，删除所有值为 value 的元素，count>0 从头至尾删除 count 个值为 value 的元素，count<0 从尾到头删除|count|个值为 value 的元素。
- **lpop(*key*)**：返回并删除名称为 key 的 list 中的首元素
- **rpop(*key*)**：返回并删除名称为 key 的 list 中的尾元素
- **blpop(*key1, key2,… key N, timeout*)**：lpop 命令的 block 版本。即当 timeout 为 0 时，若遇到名称为 key *i* 的 list 不存在或该 list 为空，则命令结束。如果 timeout>0，则遇到上述情况时，等待 timeout 秒，如果问题没有解决，则对 key *i+1* 开始的 list 执行 pop 操作。
- **brpop(*key1, key2,… key N, timeout*)**：rpop 的 block 版本。参考上一命令。
- **rpoplpush(*srckey, dstkey*)**：返回并删除名称为 srckey 的 list 的尾元素，并将该元素添加到名称为 dstkey 的 list 的头部

21.5 对 Set 操作的命令

- **sadd(*key, member*)**：向名称为 key 的 set 中添加元素 member
- **srem(*key, member*)**：删除名称为 key 的 set 中的元素 member
- **spop(*key*)**：随机返回并删除名称为 key 的 set 中一个元素
- **smove(*srckey, dstkey, member*)**：将 member 元素从名称为 srckey 的集合移到名称为 dstkey 的集合
- **scard(*key*)**：返回名称为 key 的 set 的基数
- **sismember(*key, member*)**：测试 member 是否是名称为 key 的 set 的元素
- **sinter(*key1, key2,…key N*)**：求交集
- **sinterstore(*dstkey, key1, key2,…key N*)**：求交集并将交集保存到 dstkey 的集合
- **sunion(*key1, key2,…key N*)**：求并集
- **sunionstore(*dstkey, key1, key2,…key N*)**：求并集并将并集保存到 dstkey 的集合

- **sdiff(*key1, key2,…key N*)**：求差集
- **sdiffstore(*dstkey, key1, key2,…key N*)**：求差集并将差集保存到 dstkey 的集合
- **smembers(*key*)**：返回名称为 key 的 set 的所有元素
- **srandmember(*key*)**：随机返回名称为 key 的 set 的一个元素

21.6 对 zset（sorted set）操作的命令

- **zadd(*key, score, member*)**：向名称为 key 的 zset 中添加元素 member，score 用于排序。如果该元素已经存在，则根据 score 更新该元素的顺序。
- **zrem(*key, member*)**：删除名称为 key 的 zset 中的元素 member
- **zincrby(*key, increment, member*)**：如果在名称为 key 的 zset 中已经存在元素 member，则该元素的 score 增加 increment；否则向集合中添加该元素，其 score 的值为 increment
- **zrank(*key, member*)**：返回名称为 key 的 zset（元素已按 score 从小到大排序）中 member 元素的 rank（即 index，从 0 开始），若没有 member 元素，返回“nil”
- **zrevrank(*key, member*)**：返回名称为 key 的 zset（元素已按 score 从大到小排序）中 member 元素的 rank（即 index，从 0 开始），若没有 member 元素，返回“nil”
- **zrange(*key, start, end*)**：返回名称为 key 的 zset（元素已按 score 从小到大排序）中的 index 从 start 到 end 的所有元素
- **zrevrange(*key, start, end*)**：返回名称为 key 的 zset（元素已按 score 从大到小排序）中的 index 从 start 到 end 的所有元素
- **zrangebyscore(*key, min, max*)**：返回名称为 key 的 zset 中 score >= min 且 score <= max 的所有元素
- **zcard(*key*)**：返回名称为 key 的 zset 的基数
- **zscore(*key, element*)**：返回名称为 key 的 zset 中元素 element 的 score
- **zremrangebyrank(*key, min, max*)**：删除名称为 key 的 zset 中 rank >= min 且 rank <= max 的所有元素
- **zremrangebyscore(*key, min, max*)**：删除名称为 key 的 zset 中 score >= min 且 score <= max 的所有元素
- **zunionstore / zinterstore(*dstkeyN, key1,…,keyN*, WEIGHTS *w1,…wN,* AGGREGATE SUM|MIN|MAX)**：对 N 个 zset 求并集和交集，并将最后的集合保存在 dstkeyN 中。对于集合中每一个元素的 score，在进行 AGGREGATE 运算前，都要乘以对于的 WEIGHT 参数。如果没有提供 WEIGHT，默认为 1。默认的 AGGREGATE 是 SUM，即结果集合中元素的 score 是所有集合对应元素进行 SUM 运算的值，而 MIN 和 MAX 是指，结果集合中元素的 score 是所有集合对应元素中最小值和最大值。

21.7 对 Hash 操作的命令

- **hset(*key, field, value*)**：向名称为 key 的 hash 中添加元素 field<—>value
- **hget(*key, field*)**：返回名称为 key 的 hash 中 field 对应的 value
- **hmget(*key, field1, …,field N*)**：返回名称为 key 的 hash 中 field *i* 对应的 value
- **hmset(*key, field1, value1,…,field N, value N*)**：向名称为 key 的 hash 中添加元素 field *i*<—>value *i*
- **hincrby(*key, field, integer*)**：将名称为 key 的 hash 中 field 的 value 增加 integer
- **hexists(*key, field*)**：名称为 key 的 hash 中是否存在键为 field 的域
- **hdel(*key, field*)**：删除名称为 key 的 hash 中键为 field 的域
- **hlen(*key*)**：返回名称为 key 的 hash 中元素个数
- **hkeys(*key*)**：返回名称为 key 的 hash 中所有键
- **hvals(*key*)**：返回名称为 key 的 hash 中所有键对应的 value
- **hgetall(*key*)**：返回名称为 key 的 hash 中所有的键（field）及其对应的 value

21.8 持久化

- **save**：将数据同步保存到磁盘
- **bgsave**：将数据异步保存到磁盘
- **lastsave**：返回上次成功将数据保存到磁盘的 Unix 时戳
- **shundown**：将数据同步保存到磁盘，然后关闭服务

21.9 远程服务控制

- **info**：提供服务器的信息和统计
- **monitor**：实时转储收到的请求
- **slaveof**：改变复制策略设置
- **config**：在运行时配置 Redis 服务器