**Redis**

## 一、简单概述

**1 Redis是一个开源的使用ANSI C语言编写、支持网络、可基于内存亦可持久化的日志型、一个高性能的key-value数据库。**并提供多种语言的API。说到Key-Value数据库NoSQL数据库可以想到MongoDB。和Memcached类似，它支持存储的value类型相对更多，包括string(字符串)、list(链表)、set(集合)、zset(sorted set --有序集合)和hash（哈希类型）。这些数据类型都支持push/pop、add/remove及取交集并集和差集及更丰富的操作，而且这些操作都是原子性的。在此基础上，redis支持各种不同方式的排序。与memcached一样，为了保证效率，数据都是缓存在内存中。区别的是redis会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件，并且在此基础上实现了master-slave(主从)同步。 Redis 是完全开源免费的，遵守BSD协议，是一个高性能的key-value数据库。

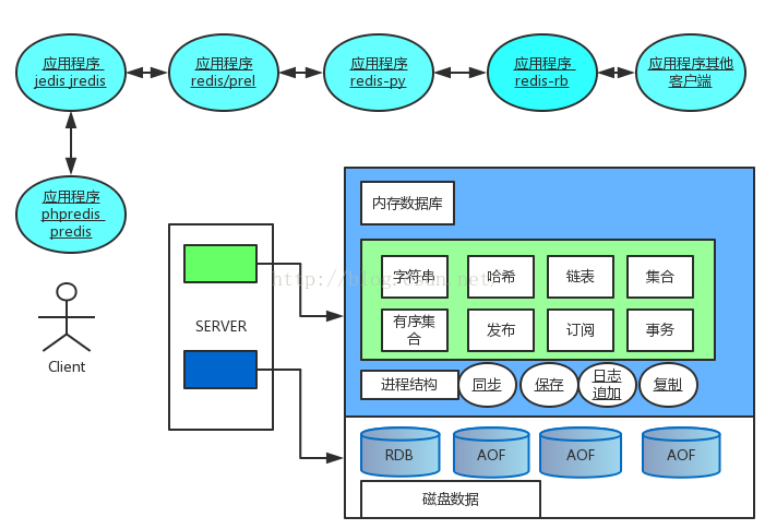


**2 Redis 与其他 key - value 缓存产品有以下三个特点：**

@Redis支持数据的持久化，可以将内存中的数据保存在磁盘中，重启的时候可以再次加载进行使用。

@Redis不仅仅支持简单的key-value类型的数据，同时还提供list，set，zset，hash等数据结构的存储。

@Redis支持数据的备份，即master-slave模式的数据备份。



**3 Redis是一款开源的、高性能的键-值存储（key-value store）。**

它常被称作是一款数据结构服务器（data structure server）。Redis的键值可以包括字符串（strings）类型，同时它还包括哈希（hashes）、列表（lists）、集合（sets）和 有序集合（sorted sets）等数据类型。 对于这些数据类型，你可以执行原子操作。例如：对字符串进行附加操作（append）；递增哈希中的值；向列表中增加元素；计算集合的交集、并集与差集等。为了获得优异的性能，Redis采用了内存中（in-memory）数据集（dataset）的方式。同时，Redis支持数据的持久化，你可以每隔一段时间将数据集转存到磁盘上（snapshot），或者在日志尾部追加每一条操作命令（append only file,aof）。Redis同样支持主从复制（master-slave replication），并且具有非常快速的非阻塞首次同步（ non-blocking first synchronization）、网络断开自动重连等功能。同时Redis还具有其它一些特性，其中包括简单的事物支持、发布订阅 （ pub/sub）、管道（pipeline）和虚拟内存（vm）等 。Redis具有丰富的客户端，支持现阶段流行的大多数编程语言。

**4 Redis配置**

1. . Redis默认不是以守护进程的方式运行，可以通过该配置项修改，使用yes启用守护进程

daemonize no

2）. 当Redis以守护进程方式运行时，Redis默认会把pid写入/var/run/redis.pid文件，可以通过pidfile指定

pidfile /var/run/redis.pid

3）. 指定Redis监听端口，默认端口为6379，作者在自己的一篇博文中解释了为什么选用6379作为默认端口，因为6379在手机按键上MERZ对应的号码，而MERZ取自意大利歌女Alessia Merz的名字

port 6379

4）. 绑定的主机地址

bind 127.0.0.1

5）.当 客户端闲置多长时间后关闭连接，如果指定为0，表示关闭该功能

timeout 300

6）. 指定日志记录级别，Redis总共支持四个级别：debug、verbose、notice、warning，默认为verbose

loglevel verbose

7）. 日志记录方式，默认为标准输出，如果配置Redis为守护进程方式运行，而这里又配置为日志记录方式为标准输出，则日志将会发送给/dev/null

logfile stdout

8）. 设置数据库的数量，默认数据库为0，可以使用SELECT <dbid>命令在连接上指定数据库id

databases 16

9）. 指定在多长时间内，有多少次更新操作，就将数据同步到数据文件，可以多个条件配合

save <seconds> <changes>

Redis默认配置文件中提供了三个条件：

save 900 1

save 300 10

save 60 10000

分别表示900秒（15分钟）内有1个更改，300秒（5分钟）内有10个更改以及60秒内有10000个更改。

10）. 指定存储至本地数据库时是否压缩数据，默认为yes，Redis采用LZF压缩，如果为了节省CPU时间，可以关闭该选项，但会导致数据库文件变的巨大

rdbcompression yes

11）. 指定本地数据库文件名，默认值为dump.rdb

dbfilename dump.rdb

12）. 指定本地数据库存放目录

dir ./

13）. 设置当本机为slav服务时，设置master服务的IP地址及端口，在Redis启动时，它会自动从master进行数据同步

slaveof <masterip> <masterport>

14）. 当master服务设置了密码保护时，slav服务连接master的密码

masterauth <master-password>

15）. 设置Redis连接密码，如果配置了连接密码，客户端在连接Redis时需要通过AUTH <password>命令提供密码，默认关闭

requirepass foobared

16）. 设置同一时间最大客户端连接数，默认无限制，Redis可以同时打开的客户端连接数为Redis进程可以打开的最大文件描述符数，如果设置 maxclients 0，表示不作限制。当客户端连接数到达限制时，Redis会关闭新的连接并向客户端返回max number of clients reached错误信息

maxclients 128

17）. 指定Redis最大内存限制，Redis在启动时会把数据加载到内存中，达到最大内存后，Redis会先尝试清除已到期或即将到期的Key，当此方法处理 后，仍然到达最大内存设置，将无法再进行写入操作，但仍然可以进行读取操作。Redis新的vm机制，会把Key存放内存，Value会存放在swap区

maxmemory <bytes>

18）. 指定是否在每次更新操作后进行日志记录，Redis在默认情况下是异步的把数据写入磁盘，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。默认为no

appendonly no

19）. 指定更新日志文件名，默认为appendonly.aof

appendfilename appendonly.aof

20）. 指定更新日志条件，共有3个可选值：   
 no：表示等操作系统进行数据缓存同步到磁盘（快）   
 always：表示每次更新操作后手动调用fsync()将数据写到磁盘（慢，安全）   
 everysec：表示每秒同步一次（折衷，默认值）

appendfsync everysec

21）.指定是否启用虚拟内存机制，默认值为no，简单的介绍一下，VM机制将数据分页存放，由Redis将访问量较少的页即冷数据swap到磁盘上，访问多的页面由磁盘自动换出到内存中（在后面的文章我会仔细分析Redis的VM机制）

vm-enabled no

22）. 虚拟内存文件路径，默认值为/tmp/redis.swap，不可多个Redis实例共享

vm-swap-file /tmp/redis.swap

23）. 将所有大于vm-max-memory的数据存入虚拟内存,无论vm-max-memory设置多小,所有索引数据都是内存存储的(Redis的索引数据 就是keys),也就是说,当vm-max-memory设置为0的时候,其实是所有value都存在于磁盘。默认值为0

vm-max-memory 0

24）. Redis swap文件分成了很多的page，一个对象可以保存在多个page上面，但一个page上不能被多个对象共享，vm-page-size是要根据存储的 数据大小来设定的，作者建议如果存储很多小对象，page大小最好设置为32或者64bytes；如果存储很大大对象，则可以使用更大的page，如果不 确定，就使用默认值

vm-page-size 32

25）. 设置swap文件中的page数量，由于页表（一种表示页面空闲或使用的bitmap）是在放在内存中的，，在磁盘上每8个pages将消耗1byte的内存。

vm-pages 134217728

26）. 设置访问swap文件的线程数,最好不要超过机器的核数,如果设置为0,那么所有对swap文件的操作都是串行的，可能会造成比较长时间的延迟。默认值为4

vm-max-threads 4

27）. 设置在向客户端应答时，是否把较小的包合并为一个包发送，默认为开启

glueoutputbuf yes

28）. 指定在超过一定的数量或者最大的元素超过某一临界值时，采用一种特殊的哈希算法

hash-max-zipmap-entries 64

hash-max-zipmap-value 512

29）. 指定是否激活重置哈希，默认为开启

activerehashing yes

30）. 指定包含其它的配置文件，可以在同一主机上多个Redis实例之间使用同一份配置文件，而同时各个实例又拥有自己的特定配置文件

## 二、核心技术

**1 redis优势**

**1）速度快**，因为数据存在内存中，类似于HashMap，HashMap的优势就是查找和操作的时间复杂度都是O(1)

**2）持丰富数据类型**，支持string，list，set，sorted set，hash

**3）持事务**，但是对事务的支持不够好，如果在一个事务中执行多少操作，有一个操作失败，不会回滚

**4）富的特性**：可用于缓存，消息，按key设置过期时间，过期后将会自动删除

**2 redis的并发竞争问题如何解决**

**Redis为单进程单线程模式**，采用队列模式将并发访问变为串行访问。Redis本身没有锁的概念，Redis对于多个客户端连接并不存在竞争，但是在Jedis客户端对Redis进行并发访问时会发生连接超时、数据转换错误、阻塞、客户端关闭连接等问题，这些问题均是由于客户端连接混乱造成。对此有2种解决方法：

1）客户端角度，为保证每个客户端间正常有序与Redis进行通信，对连接进行池化，同时对客户端读写Redis操作采用内部锁synchronized

2）服务器角度，利用setnx实现锁。

对于第一种，需要应用程序自己处理资源的同步，可以使用的方法比较通俗，可以使用synchronized也可以使用lock；第二种需要用到Redis的setnx命令，但是需要注意一些问题。

**3 redis事物的了解CAS(check-and-set 操作实现乐观锁 )**

和众多其它数据库一样，Redis作为NoSQL数据库也同样提供了事务机制。在Redis中，MULTI/EXEC/DISCARD/WATCH这四个命令是我们实现事务的基石。相信对有关系型数据库开发经验的开发者而言这一概念并不陌生，即便如此，我们还是会简要的列出Redis中事务的实现特征：

1. 在事务中的所有命令都将会被串行化的顺序执行，事务执行期间，Redis不会再为其它客户端的请求提供任何服务，从而保证了事物中的所有命令被原子的执行
2. 和关系型数据库中的事务相比，在Redis事务中如果有某一条命令执行失败，其后的命令仍然会被继续执行
3. 我们可以通过MULTI命令开启一个事务，有关系型数据库开发经验的人可以将其理解为”BEGIN TRANSACTION”语句。在该语句之后执行的命令都将被视为事务之内的操作，最后我们可以通过执行EXEC/DISCARD命令来提交/回滚该事务内的所有操作。这两个Redis命令可被视为等同于关系型数据库中的COMMIT/ROLLBACK语句。

4） 在事务开启之前，如果客户端与服务器之间出现通讯故障并导致网络断开，其后所有待执行的语句都将不会被服务器执行。然而如果网络中断事件是发生在客户端执行EXEC命令之后，那么该事务中的所有命令都会被服务器执行

5） 当使用Append-Only模式时，Redis会通过调用系统函数write将该事务内的所有写操作在本次调用中全部写入磁盘。然而如果在写入的过程中出现系统崩溃，如电源故障导致的宕机，那么此时也许只有部分数据被写入到磁盘，而另外一部分数据却已经丢失。Redis服务器会在重新启动时执行一系列必要的一致性检测，一旦发现类似问题，就会立即退出并给出相应的错误提示。此时，我们就要充分利用Redis工具包中提供的redis-check-aof工具，该工具可以帮助我们定位到数据不一致的错误，并将已经写入的部分数据进行回滚。修复之后我们就可以再次重新启动Redis服务器了

**4 WATCH命令和基于CAS的乐观锁**

**1）在Redis的事务中，WATCH命令可用于提供CAS(check-and-set)功能。**假设我们通过WATCH命令在事务执行之前监控了多个Keys，倘若在WATCH之后有任何Key的值发生了变化，EXEC命令执行的事务都将被放弃，同时返回Null multi-bulk应答以通知调用者事务执行失败。例如，我们再次假设Redis中并未提供incr命令来完成键值的原子性递增，

如果要实现该功能，我们只能自行编写相应的代码。其伪码如下：

val = GET mykey

val = val + 1

SET mykey $val

**2）以上代码只有在单连接的情况下才可以保证执行结果是正确的**，因为如果在同一时刻有多个客户端在同时执行该段代码，那么就会出现多线程程序中经常出现的一种错误场景–竞态争用(race condition)。比如，客户端A和B都在同一时刻读取了mykey的原有值，假设该值为10，此后两个客户端又均将该值加一后set回Redis服务器，这样就会导致mykey的结果为11，而不是我们认为的12。为了解决类似的问题，我们需要借助WATCH命令的帮助，见如下代码：

WATCH mykey

val = GET mykey

val = val + 1

MULTI

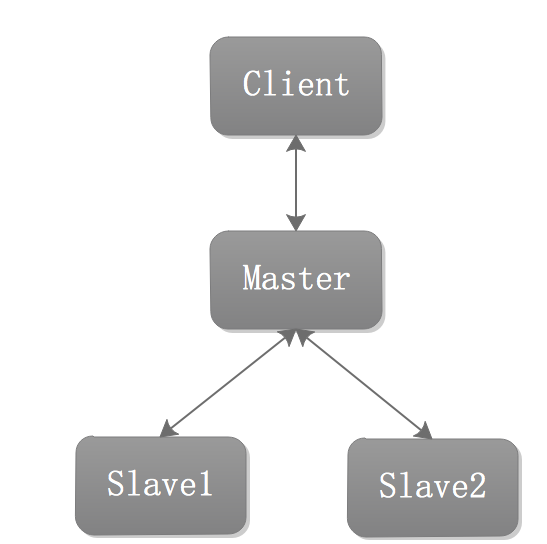
SET mykey $val

EXEC

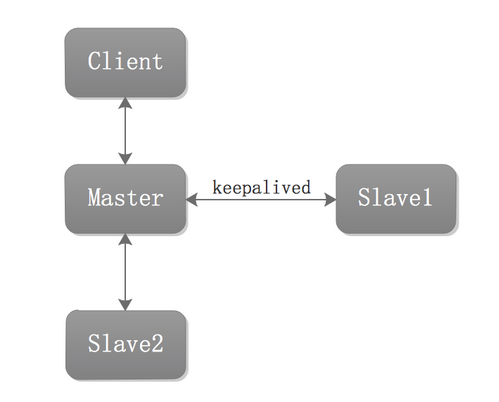
和此前代码不同的是，新代码在获取mykey的值之前先通过WATCH命令监控了该键，此后又将set命令包围在事务中，这样就可以有效的保证每个连接在执行EXEC之前，如果当前连接获取的mykey的值被其它连接的客户端修改，那么当前连接的EXEC命令将执行失败。这样调用者在判断返回值后就可以获悉val是否被重新设置成功。

**5 备份模式和容灾模式**

**（1）备份模式-主从复制**



@常用Redis主从复制，这是最常见的一种架构，一个Master节点，两个Slave节点。客户端写数据的时候是写Master节点，读的时候，是读取两个Slave，这样实现读的扩展，减轻了Master节点读负载。



@这种架构同样是一个Master和两个Slave。不同的是Master和Slave1使用keepalived进行VIP转移。Client连接Master的时候是通过VIP进行连接的。避免了方案一IP更改的情况。

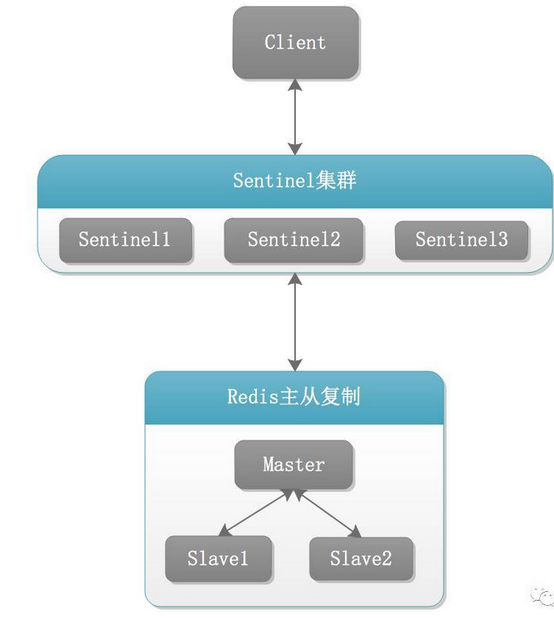
**（2）Redis主从复制优点**

实现了对master数据的备份，一旦master出现故障，slave节点可以提升为新的master，顶替旧的master继续提供服务。实现读扩展。使用主从复制架构， 一般都是为了实现读扩展。Master主要实现写功能， Slave实现读的功能。

**（3）Redis主从复制不足**

当Master出现故障时，Client就与Master端断开连接，无法实现写功能，同时Slave也无法从Master进行复制。当master出现故障后，Client可以连接到Slave1上进行数据操作，但是Slave1就成了一个单点，就出现了经常要避免的单点故障(single point of failure)。

**（4）Redis Sentinel架构**



使用sentinel实现了Redis的高可用，当master出现故障时，完全无需人工干预即可实现故障转移。避免了对业务的影响，提高了运维工作效率。在部署sentinel的时候，建议使用奇数个sentinel节点，最少三个sentinel节点。

**6 redis的缓存失效策略和主键失效机制**

作为缓存系统都要定期清理无效数据，就需要一个主键失效和淘汰策略。在Redis当中，有生存期的key被称为volatile。在创建缓存时，要为给定的key设置生存期，当key过期的时候（生存期为0）它可能会被删除。

**1） 影响生存时间的一些操作**

生存时间可以通过使用 DEL 命令来删除整个 key 来移除，或者被 SET 和 GETSET 命令覆盖原来的数据，也就是说，修改key对应的value和使用另外相同的key和value来覆盖以后，当前数据的生存时间不同。

比如说，对一个 key 执行INCR命令，对一个列表进行LPUSH命令，或者对一个哈希表执行HSET命令，这类操作都不会修改 key 本身的生存时间。另一方面，如果使用RENAME对一个 key 进行改名，那么改名后的 key的生存时间和改名前一样。

RENAME命令的另一种可能是，尝试将一个带生存时间的 key 改名成另一个带生存时间的 another\_key ，这时旧的 another\_key (以及它的生存时间)会被删除，然后旧的 key 会改名为 another\_key ，因此，新的 another\_key 的生存时间也和原本的 key 一样。使用PERSIST命令可以在不删除 key 的情况下，移除 key 的生存时间，让 key 重新成为一个persistent key 。

**2） 如何更新生存时间**

可以对一个已经带有生存时间的 key 执行EXPIRE命令，新指定的生存时间会取代旧的生存时间。过期时间的精度已经被控制在1ms之内，主键失效的时间复杂度是O（1）

EXPIRE和TTL命令搭配使用，TTL可以查看key的当前生存时间。设置成功返回 1；当 key 不存在或者不能为 key 设置生存时间时，返回 0

**3）最大缓存配置**

在 redis 中，允许用户设置最大使用内存大小server.maxmemory默认为0，没有指定最大缓存，如果有新的数据添加，超过最大内存，则会使redis崩溃，所以一定要设置。redis 内存数据集大小上升到一定大小的时候，就会实行数据淘汰策略。

**7 六种数据淘汰策略**

**1） volatile-lru**：从已设置过期时间的数据集（server.db[i].expires）挑选最近最少使用的数据淘汰

**2） volatile-ttl：从**已设置过期时间的数据集（server.db[i].expires）挑选将要过期的数据淘汰

**3） volatile-random：**从已设置过期时间的数据集（server.db[i].expires）任意选择数据淘汰

**4） allkeys-lru：**从数据集（server.db[i].dict）挑选最近最少使用的数据淘汰

**5） allkeys-random：**从数据集（server.db[i].dict）任意选择数据淘汰

**6） no-enviction**（驱逐）禁止驱逐数据

注意这里的6种机制，volatile和allkeys规定了是对已设置过期时间的数据集淘汰数据还是从全部数据集淘汰数据，后面的lru、ttl以及random是三种不同的淘汰策略，再加上一种no-enviction永不回收的策略

**8 使用策略规则：**

1） 如果数据呈现幂律分布，也就是一部分数据访问频率高，一部分数据访问频率低，则使用allkeys-lru

2） 如果数据呈现平等分布，也就是所有的数据访问频率都相同，则使用allkeys-random

**9 三种数据淘汰策略：**

ttl和random比较容易理解，实现也会比较简单。主要是Lru最近最少使用淘汰策略，设计上会对key 按失效时间排序，然后取最先失效的key进行淘汰

**10 与memcached相比：**

1、Redis和Memcache都是将数据存放在内存中，都是内存数据库。不过memcache还可用于缓存其他东西，例如图片、视频等等；

2、Redis不仅仅支持简单的k/v类型的数据，同时还提供list，set，hash等数据结构的存储；

3、[虚拟内存](https://www.baidu.com/s?wd=%E8%99%9A%E6%8B%9F%E5%86%85%E5%AD%98&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHcvrjTdrH00T1Y4rjDYnju-njb1nycLmW-b0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnHbzPHfYP10LPjDznWnkn1T3Ps" \t "https://www.cnblogs.com/timothy-lai/p/_blank)--Redis当物理内存用完时，可以将一些很久没用到的value 交换到磁盘；

4、过期策略--memcache在set时就指定，例如set key1 0 0 8,即永不过期。Redis可以通过例如expire 设定，例如expire name 10；

5、分布式--设定memcache集群，利用magent做一主多从;redis可以做一主多从。都可以一主一从；

6、存储数据安全--memcache挂掉后，数据没了；redis可以定期保存到磁盘（持久化）；

7、[灾难恢复](https://www.baidu.com/s?wd=%E7%81%BE%E9%9A%BE%E6%81%A2%E5%A4%8D&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHcvrjTdrH00T1Y4rjDYnju-njb1nycLmW-b0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnHbzPHfYP10LPjDznWnkn1T3Ps" \t "https://www.cnblogs.com/timothy-lai/p/_blank)--memcache挂掉后，数据不可恢复; redis数据丢失后可以通过aof恢复；

8、Redis支持数据的备份，即master-slave模式的数据备份；

9、应用场景不一样：Redis出来作为NoSQL数据库使用外，还能用做[消息队列](http://www.cnblogs.com/timothy-lai/p/5711840.html" \t "https://www.cnblogs.com/timothy-lai/p/_blank)、数据堆栈和数据缓存等；Memcached适合于缓存SQL语句、数据集、用户临时性数据、延迟查询数据和session等。

**11 Redis支持五种数据类型**

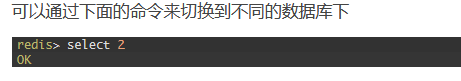
string（字符串）hash（map< string, string>）list（链表）set（没有排序的集合）zset（有序集合）

**12 Redis多数据库操作：**

主要为了防止key冲突问题，默认情况下，客户端连接到数据库0，可通过修改配置文件redis.conf下的参数来进行修改：

databases 16

可通过select <数据库ID>来进行修改，**每个数据库都有自己单独的空间，不用担心key冲突问题。**



**13 相关命令**

**（1）redis启动**

本地启动：redis-cli

远程启动：redis-cli -h host -p port -a password

AUTH password

验证密码是否正确

ECHO message

打印字符串

PING

查看服务是否运行

QUIT

关闭当前连接

**SELECT index**

切换到指定的数据库

**（2）redis keys命令**

DEL key

DUMP key

序列化给定的key并返回序列化的值

EXISTS key

检查给定的key是否存在

EXPIRE key seconds

为key设置过期时间

EXPIRE key timestamp

用时间戳的方式给key设置过期时间

PEXPIRE key milliseconds

设置key的过期时间以毫秒计

**KEYS pattern**

查找所有符合给定模式的key

MOVE key db

将当前数据库的key移动到数据库db当中

PERSIST key

移除key的过期时间，key将持久保存

PTTL key

以毫秒为单位返回key的剩余过期时间

TTL key

以秒为单位，返回给定key的剩余生存时间

RANDOMKEY

从当前数据库中随机返回一个key

RENAME key newkey

修改key的名称

RENAMENX key newkey

仅当newkey不存在时，将key改名为newkey

TYPE key

返回key所存储的值的类型

**（3）reids字符串命令**

SET key value

GET key

GETRANGE key start end

返回key中字符串值的子字符

GETSET key value

将给定key的值设为value，并返回key的旧值

GETBIT KEY OFFSET

对key所储存的字符串值，获取指定偏移量上的位

MGET KEY1 KEY2

获取一个或者多个给定key的值

SETBIT KEY OFFSET VALUE

对key所是存储的字符串值，设置或清除指定偏移量上的位

SETEX key seconds value

将值 value 关联到 key ，并将 key 的过期时间设为 seconds (以秒为单位)。

SETNX key value

只有在 key 不存在时设置 key 的值。

SETRANGE key offset value

用 value 参数覆写给定 key 所储存的字符串值，从偏移量 offset 开始。

STRLEN key

返回 key 所储存的字符串值的长度。

MSET key value [key value ...]

同时设置一个或多个 key-value 对。

MSETNX key value [key value ...]

同时设置一个或多个 key-value 对，当且仅当所有给定 key 都不存在。

PSETEX key milliseconds value

这个命令和 SETEX 命令相似，但它以毫秒为单位设置 key 的生存时间，而不是像 SETEX 命令那样，以秒为单位。

INCR key

将 key 中储存的数字值增一。

INCRBY key increment

将 key 所储存的值加上给定的增量值（increment） 。

INCRBYFLOAT key increment

将 key 所储存的值加上给定的浮点增量值（increment） 。

DECR key

将 key 中储存的数字值减一。

DECRBY key decrement

key 所储存的值减去给定的减量值（decrement） 。

APPEND key value

如果 key 已经存在并且是一个字符串， APPEND 命令将 指定value 追加到改 key 原来的值（value）的末尾。

**（4）Redis hash 命令**

HDEL key field1 [field2]

删除一个或多个哈希表字段

HEXISTS key field

查看哈希表 key 中，指定的字段是否存在。

HGET key field

获取存储在哈希表中指定字段的值。

HGETALL key

获取在哈希表中指定 key 的所有字段和值

HINCRBY key field increment

为哈希表 key 中的指定字段的整数值加上增量 increment 。

HINCRBYFLOAT key field increment

为哈希表 key 中的指定字段的浮点数值加上增量 increment 。

HKEYS key

获取所有哈希表中的字段

HLEN key

获取哈希表中字段的数量

HMGET key field1 [field2]

获取所有给定字段的值

HMSET key field1 value1 [field2 value2 ]

同时将多个 field-value (域-值)对设置到哈希表 key 中。

HSET key field value

将哈希表 key 中的字段 field 的值设为 value 。

HSETNX key field value

只有在字段 field 不存在时，设置哈希表字段的值。

HVALS key

获取哈希表中所有值

HSCAN key cursor [MATCH pattern] [COUNT count]

迭代哈希表中的键值对。

**（5）Redis 列表命令**

BLPOP key1 [key2 ] timeout

移出并获取列表的第一个元素， 如果列表没有元素会阻塞列表直到等待超时或发现可弹出元素为止。

BRPOP key1 [key2 ] timeout

移出并获取列表的最后一个元素， 如果列表没有元素会阻塞列表直到等待超时或发现可弹出元素为止。

BRPOPLPUSH source destination timeout

从列表中弹出一个值，将弹出的元素插入到另外一个列表中并返回它； 如果列表没有元素会阻塞列表直到等待超时或发现可弹出元素为止。

LINDEX key index

通过索引获取列表中的元素

LINSERT key BEFORE|AFTER pivot value

在列表的元素前或者后插入元素

LLEN key

获取列表长度

LPOP key

移出并获取列表的第一个元素

LPUSH key value1 [value2]

将一个或多个值插入到列表头部

LPUSHX key value

将一个值插入到已存在的列表头部

**LRANGE key start stop**

**获取列表指定范围内的元素**

LREM key count value

移除列表元素

LSET key index value

通过索引设置列表元素的值

LTRIM key start stop

对一个列表进行修剪(trim)，就是说，让列表只保留指定区间内的元素，不在指定区间之内的元素都将被删除。

RPOP key

移除并获取列表最后一个元素

RPOPLPUSH source destination

移除列表的最后一个元素，并将该元素添加到另一个列表并返回

RPUSH key value1 [value2]

在列表中添加一个或多个值

RPUSHX key value

为已存在的列表添加值

**（6）Redis 集合命令**

SADD key member1 [member2]

向集合添加一个或多个成员

SCARD key

获取集合的成员数

SDIFF key1 [key2]

返回给定所有集合的差集

SDIFFSTORE destination key1 [key2]

返回给定所有集合的差集并存储在 destination 中

SINTER key1 [key2]

返回给定所有集合的交集

SINTERSTORE destination key1 [key2]

返回给定所有集合的交集并存储在 destination 中

SISMEMBER key member

判断 member 元素是否是集合 key 的成员

SMEMBERS key

返回集合中的所有成员

SMOVE source destination member

将 member 元素从 source 集合移动到 destination 集合

SPOP key

移除并返回集合中的一个随机元素

SRANDMEMBER key [count]

返回集合中一个或多个随机数

SREM key member1 [member2]

移除集合中一个或多个成员

SUNION key1 [key2]

返回所有给定集合的并集

SUNIONSTORE destination key1 [key2]

所有给定集合的并集存储在 destination 集合中

SSCAN key cursor [MATCH pattern] [COUNT count]

迭代集合中的元素

**（7）Redis 有序集合命令**

ZADD key score1 member1 [score2 member2]

向有序集合添加一个或多个成员，或者更新已存在成员的分数

ZCARD key

获取有序集合的成员数

ZCOUNT key min max

计算在有序集合中指定区间分数的成员数

ZINCRBY key increment member

有序集合中对指定成员的分数加上增量 increment

ZINTERSTORE destination numkeys key [key ...]

计算给定的一个或多个有序集的交集并将结果集存储在新的有序集合 key 中

ZLEXCOUNT key min max

在有序集合中计算指定字典区间内成员数量

ZRANGE key start stop [WITHSCORES]

通过索引区间返回有序集合成指定区间内的成员

ZRANGEBYLEX key min max [LIMIT offset count]

通过字典区间返回有序集合的成员

ZRANGEBYSCORE key min max [WITHSCORES] [LIMIT]

通过分数返回有序集合指定区间内的成员

ZRANK key member

返回有序集合中指定成员的索引

ZREM key member [member ...]

移除有序集合中的一个或多个成员

ZREMRANGEBYLEX key min max

移除有序集合中给定的字典区间的所有成员

ZREMRANGEBYRANK key start stop

移除有序集合中给定的排名区间的所有成员

ZREMRANGEBYSCORE key min max

移除有序集合中给定的分数区间的所有成员

ZREVRANGE key start stop [WITHSCORES]

返回有序集中指定区间内的成员，通过索引，分数从高到底

ZREVRANGEBYSCORE key max min [WITHSCORES]

返回有序集中指定分数区间内的成员，分数从高到低排序

ZREVRANK key member

返回有序集合中指定成员的排名，有序集成员按分数值递减(从大到小)排序

ZSCORE key member

返回有序集中，成员的分数值

ZUNIONSTORE destination numkeys key [key ...]

计算给定的一个或多个有序集的并集，并存储在新的 key 中

ZSCAN key cursor [MATCH pattern] [COUNT count]

迭代有序集合中的元素（包括元素成员和元素分值）

**（8）Redis 发布订阅命令**

PSUBSCRIBE pattern [pattern ...]

订阅一个或多个符合给定模式的频道。

PUBSUB subcommand [argument [argument ...]]

查看订阅与发布系统状态。

PUBLISH channel message

将信息发送到指定的频道。

PUNSUBSCRIBE [pattern [pattern ...]]

退订所有给定模式的频道。

SUBSCRIBE channel [channel ...]

订阅给定的一个或多个频道的信息。

UNSUBSCRIBE [channel [channel ...]]

指退订给定的频道。

示例：

redis 127.0.0.1:6379> SUBSCRIBE redisChat

Reading messages... (press Ctrl-C to quit)

1) "subscribe"

2) "redisChat"

3) (integer) 1

现在，我们先重新开启个 redis 客户端，然后在同一个频道 redisChat 发布两次消息，订阅者就能接收到消息。

redis 127.0.0.1:6379> PUBLISH redisChat "Redis is a great caching technique"

# 订阅者的客户端会显示如下消息

1) "message"

2) "redisChat"

3) "Redis is a great caching technique"

**（9）Redis 事务命令**

DISCARD

取消事务，放弃执行事务块内的所有命令。

EXEC

执行所有事务块内的命令。

MULTI

标记一个事务块的开始。

UNWATCH

取消 WATCH 命令对所有 key 的监视。

WATCH key [key ...]

监视一个(或多个) key ，如果在事务执行之前这个(或这些) key 被其他命令所改动，那么事务将被打断。

**（10）Redis 脚本命令**

EVAL script numkeys key [key ...] arg [arg ...]

执行 Lua 脚本。

EVALSHA sha1 numkeys key [key ...] arg [arg ...]

执行 Lua 脚本。

SCRIPT EXISTS script [script ...]

查看指定的脚本是否已经被保存在缓存当中。

SCRIPT FLUSH

从脚本缓存中移除所有脚本。

SCRIPT KILL

杀死当前正在运行的 Lua 脚本。

SCRIPT LOAD script

将脚本 script 添加到脚本缓存中，但并不立即执行这个脚本。

**（11）Redis 服务器命令**

BGREWRITEAOF

异步执行一个 AOF（AppendOnly File） 文件重写操作

BGSAVE

在后台异步保存当前数据库的数据到磁盘

CLIENT KILL [ip:port] [ID client-id]

关闭客户端连接

CLIENT LIST

获取连接到服务器的客户端连接列表

CLIENT GETNAME

获取连接的名称

CLIENT PAUSE timeout

在指定时间内终止运行来自客户端的命令

CLIENT SETNAME connection-name

设置当前连接的名称

CLUSTER SLOTS

获取集群节点的映射数组

COMMAND

获取 Redis 命令详情数组

COMMAND COUNT

获取 Redis 命令总数

COMMAND GETKEYS

获取给定命令的所有键

TIME

返回当前服务器时间

COMMAND INFO command-name [command-name ...]

获取指定 Redis 命令描述的数组

CONFIG GET parameter

获取指定配置参数的值

CONFIG REWRITE

对启动 Redis 服务器时所指定的 redis.conf 配置文件进行改写

CONFIG SET parameter value

修改 redis 配置参数，无需重启

CONFIG RESETSTAT

重置 INFO 命令中的某些统计数据

DBSIZE

返回当前数据库的 key 的数量

DEBUG OBJECT key

获取 key 的调试信息

DEBUG SEGFAULT

让 Redis 服务崩溃

FLUSHALL

删除所有数据库的所有key

FLUSHDB

删除当前数据库的所有key

INFO [section]

获取 Redis 服务器的各种信息和统计数值

LASTSAVE

返回最近一次 Redis 成功将数据保存到磁盘上的时间，以 UNIX 时间戳格式表示

MONITOR

实时打印出 Redis 服务器接收到的命令，调试用

ROLE

返回主从实例所属的角色

SAVE

同步保存数据到硬盘

SHUTDOWN [NOSAVE] [SAVE]

异步保存数据到硬盘，并关闭服务器

SLAVEOF host port

将当前服务器转变为指定服务器的从属服务器(slave server)

SLOWLOG subcommand [argument]

管理 redis 的慢日志

SYNC

用于复制功能(replication)的内部命令

**14 Redis持久化方式**

**（1）RDB-在指定时间间隔内生成数据集的时间点快照。**

@优点：RDB是一个非常紧凑的文件，它保存了Redis在某个时间点上的数据集。非常适合用于进行备份：比如：每小时备份一次RDB文件，遇上问题可随时还原数据集到不同的版本。非常适合灾难恢复：它只有一个文件,并且内容紧凑，可传输到其他地方。RDB在恢复大数据集时的速度比AOF的恢复速度快；RDB可最大化redis性能，父进程保存RDB文件时要做的是fork一个子进程，然后子进程处理接下来的保存操作，父进程无须执行任何磁盘I/O操作。

@缺点：如需要尽量避免在服务器故障时丢失数据，RDB不适合。每隔一段时间保存数据集的状态，一旦故障会丢失这部分数据。每次保存RDB时，redis都要fork()一个子进程，并由子进程进行实际的持久化工作。在数据集庞大时，fork()会很耗时,造成服务器在某毫秒内停止处理客户端。

@redis.conf里rdb相关配置：

save 900 1 # 900秒有1个更改，进行数据同步

save 300 10 # 300秒有10个更改，进行数据同步

save 60 10000 # 60秒有10000更改，进行数据同步

dbfilename dump.rdb # 指定rdb备份数据的文件名

**（2）AOF-记录服务器执行的所有写操作命令,并在服务器启动时，通过重新执行这些命令来还原数据库**

@优点：让redis更耐久，你可以设置不同的 fsync 策略，比如无 fsync ，每秒钟一次 fsync ，或者每次执行写入命令时 fsync 。 AOF 的默认策略为每秒钟 fsync 一次，在这种配置下，Redis 仍然可以保持良好的性能，并且就算发生故障停机，也最多只会丢失一秒钟的数据（ fsync 会在后台线程执行，所以主线程可以继续努力地处理命令请求）AOF 文件是一个只进行追加操作的日志文件（append only log） 因此对 AOF 文件的写入不需要进行 seek ， 即使日志因为某些原因而包含了未写入完整的命令（比如写入时磁盘已满，写入中途停机，等等） redis-check-aof 工具也可以轻易地修复这种问题。Redis 可以在 AOF 文件体积变得过大时，自动地在后台对 AOF 进行重写： 重写后的新 AOF 文件包含了恢复当前数据集所需的最小命令集合。 整个重写操作是绝对安全的，因为 Redis 在创建新 AOF 文件的过程中，会继续将命令追加到现有的 AOF 文件里面，即使重写过程中发生停机，现有的 AOF 文件也不会丢失。 而一旦新 AOF 文件创建完毕，Redis 就会从旧 AOF 文件切换到新 AOF 文件，并开始对新 AOF 文件进行追加操作。AOF 文件有序地保存了对数据库执行的所有写入操作， 这些写入操作以 Redis 协议的格式保存， 因此 AOF 文件的内容非常容易被人读懂， 对文件进行分析（parse）很轻松。 导出（export）AOF 文件也非常简单： 举个例子， 如果你不小心执行了 FLUSHALL 命令， 但只要 AOF 文件未被重写， 那么只要停止服务器， 移除 AOF 文件末尾的 FLUSHALL 命令， 并重启 Redis ， 就可以将数据集恢复到 FLUSHALL 执行之前的状态。

@缺点：对于相同的数据集来说，AOF 文件的体积通常要大于 RDB 文件的体积。根据所使用的 fsync 策略，AOF 的速度可能会慢于 RDB 。 在一般情况下， 每秒 fsync 的性能依然非常高， 而关闭 fsync 可以让 AOF 的速度和 RDB 一样快， 即使在高负荷之下也是如此。

@redis.conf里AOF的相关配置：

　appendonly yes //启用aof持久化方式

　appendfilename appendonly.aof // 更新日志文件名，默认为appendonly.aof

　appendfsync always //每次收到写命令就立即强制写入磁盘，最慢的，但是保证完全的持久化，不推荐使用

　appendfsync everysec //每秒钟强制写入磁盘一次，在性能和持久化方面做了很好的折中，推荐

　appendfsync no //完全依赖os，性能最好,持久化没保证

**15 影响Redis性能的因素**

**\* 网络带宽和延迟。**在执行基准测试前使用ping快速检测客户端和服务器端的延迟是一个良好的做法。对于带宽，比较好的做法是估计Gbits/s的吞吐量和网络的理论带宽值比较。在很多实际的情况，Redis的吞吐量在网络之前会受限于CPU。

**\* CPU也会是一个重要因素。**由于单线程的，Redis受益于快速的含有巨大缓存的CPU。

**\* 内存的速度和容量对于小的对象影响不大。**但对于大于10KB的对象，可能对需要注意。通常购买昂贵的快速内存模块并不是真正的很有效。

**\* Redis在虚拟机上运行慢。**虚拟化对很多普通操作来说代价太高了，Redis并没有增加多少开销在所需的系统调用和网络中断上。

**\* 客户端和服务器在一台机器运行，**对于基准测试TCP/IP回送和UNIX域套接字都可以使用。取决于平台，但UNIX域套接字比TCP/IP回送增加50%的吞吐量。

**\* 当大量使用 pipelining时**，UNIX域套接字获得的性能好处会减少。

**\* 当以太网访问Redis时**，在数据大小小于以太网数据包的大小（大约1500字节）时，聚集命令使用 pipelining会非常有效。

**\* 在多CPU套接字服务器，**Redis的表现变得依赖于NUMA配置和处理位置。

**\* 在高端配置，客户端连接的数量也是一个重要的因素。**基于epool/kqueue模型，Redis的事件循环是相当可伸缩的。

**\* 在高端的配置，**通过调优NIC(s)配置和相关中断可能取得高吞吐量。

**\* 根据平台，Redis编译可以使用不同的内存分配器，**这可能有不同的行为在原始速度,内部和外部的碎片方面

**16 Redis的数据类型**

1. **Redis支持五种数据类型：**string（字符串），hash（哈希），list（列表），set（集合）及zset(sorted set：有序集合)。

| **类型** | **简介** | **特性** | **场景** |
| --- | --- | --- | --- |
| String(字符串) | 二进制安全 | 可以包含任何数据,比如jpg图片或者序列化的对象,一个键最大能存储512M | --- |
| Hash(字典) | 键值对集合,即编程语言中的Map类型 | 适合存储对象,并且可以像数据库中update一个属性一样只修改某一项属性值(Memcached中需要取出整个字符串反序列化成对象修改完再序列化存回去) | 存储、读取、修改用户属性 |
| List(列表) | 链表(双向链表) | 增删快,提供了操作某一段元素的API | 1,最新消息排行等功能(比如朋友圈的时间线) 2,消息队列 |
| Set(集合) | 哈希表实现,元素不重复 | 1,添加、删除,查找的复杂度都是O(1) 2,为集合提供了求交集、并集、差集等操作 | 1,共同好友 2,利用唯一性,统计访问网站的所有独立ip 3,好用推荐时,根据tag求交集,大于某个阈值就可以推荐 |
| Sorted Set(有序集合) | 将Set中的元素增加一个权重参数score,元素按score有序排列 | 数据插入集合时,已经进行天然排序 | 1,排行榜 2,带权重的消息队列 |

**（2）String（字符串）**

string是redis最基本的类型，你可以理解成与Memcached一模一样的类型，一个key对应一个value。

string类型是二进制安全的。意思是redis的string可以包含任何数据。比如jpg图片或者序列化的对象 。

string类型是Redis最基本的数据类型，一个键最大能存储512MB。

实例

redis 127.0.0.1:6379> SET name "runoob"

OK

redis 127.0.0.1:6379> GET name

"runoob"

在以上实例中我们使用了 Redis 的 SET 和 GET 命令。键为 name，对应的值为 runoob。

注意：一个键最大能存储512MB。

**（2）Hash（哈希）**

Redis hash 是一个键值(key=>value)对集合。

Redis hash 是一个 string 类型的 field 和 value 的映射表，hash 特别适合用于存储对象。

实例

redis> HMSET myhash field1 "Hello" field2 "World"

"OK"

redis> HGET myhash field1

"Hello"

redis> HGET myhash field2

"World"

实例中我们使用了 Redis HMSET, HGET 命令，HMSET 设置了两个 field=>value 对, HGET 获取对应 field 对应的 value。

每个 hash 可以存储 232 -1 键值对（40多亿）。

**（3）List（列表）**

Redis 列表是简单的字符串列表，按照插入顺序排序。你可以添加一个元素到列表的头部（左边）或者尾部（右边）。

实例

redis 127.0.0.1:6379> lpush runoob redis

(integer) 1

redis 127.0.0.1:6379> lpush runoob mongodb

(integer) 2

redis 127.0.0.1:6379> lpush runoob rabitmq

(integer) 3

redis 127.0.0.1:6379> lrange runoob 0 10

1) "rabitmq"

2) "mongodb"

3) "redis"

redis 127.0.0.1:6379>

列表最多可存储 232 - 1 元素 (4294967295, 每个列表可存储40多亿)。

**（4）Set（集合）**

Redis的Set是string类型的无序集合。

集合是通过哈希表实现的，所以添加，删除，查找的复杂度都是O(1)。

sadd 命令

添加一个 string 元素到 key 对应的 set 集合中，成功返回1，如果元素已经在集合中返回 0，如果 key 对应的 set 不存在则返回错误。

sadd key member

实例

redis 127.0.0.1:6379> sadd runoob redis

(integer) 1

redis 127.0.0.1:6379> sadd runoob mongodb

(integer) 1

redis 127.0.0.1:6379> sadd runoob rabitmq

(integer) 1

redis 127.0.0.1:6379> sadd runoob rabitmq

(integer) 0

redis 127.0.0.1:6379> smembers runoob

1) "redis"

2) "rabitmq"

3) "mongodb"

注意：以上实例中 rabitmq 添加了两次，但根据集合内元素的唯一性，第二次插入的元素将被忽略。

集合中最大的成员数为 232 - 1(4294967295, 每个集合可存储40多亿个成员)。

**（5）zset(sorted set：有序集合)**

Redis zset 和 set 一样也是string类型元素的集合,且不允许重复的成员。

不同的是每个元素都会关联一个double类型的分数。redis正是通过分数来为集合中的成员进行从小到大的排序。

zset的成员是唯一的,但分数(score)却可以重复。

zadd 命令

添加元素到集合，元素在集合中存在则更新对应score

zadd key score member

实例

redis 127.0.0.1:6379> zadd runoob 0 redis

(integer) 1

redis 127.0.0.1:6379> zadd runoob 0 mongodb

(integer) 1

redis 127.0.0.1:6379> zadd runoob 0 rabitmq

(integer) 1

redis 127.0.0.1:6379> zadd runoob 0 rabitmq

(integer) 0

redis 127.0.0.1:6379> > ZRANGEBYSCORE runoob 0 1000

1) "mongodb"

2) "rabitmq"

3) "redis"

**16 Redis事务**

1. **MULTI、EXEC、DISCARD和WATCH命令是Redis事务功能的基础。**Redis事务允许在一次单独的步骤中执行一组命令，并且可以保证如下两个重要事项：

**1）Redis会将一个事务中的所有命令序列化，然后按顺序执行**。Redis不可能在一个Redis事务的执行过程中插入执行另一个客户端发出的请求。这样便能保证Redis将这些命令作为一个单独的隔离操作执行。

**2） 在一个Redis事务中，Redis要么执行其中的所有命令，要么什么都不执行。**

因此，Redis事务能够保证原子性。EXEC命令会触发执行事务中的所有命令。

因此，当某个客户端正在执行一次事务时，如果它在调用MULTI命令之前就从Redis服务端断开连接，那么就不会执行事务中的任何操作；相反，如果它在调用EXEC命令之后才从Redis服务端断开连接，那么就会执行事务中的所有操作。当Redis使用只增文件（AOF：Append-only File）时，Redis能够确保使用一个单独的write(2)系统调用，这样便能将事务写入磁盘。然而，如果Redis服务器宕机，或者系统管理员以某种方式停止Redis服务进程的运行，那么Redis很有可能只执行了事务中的一部分操作。Redis将会在重新启动时检查上述状态，然后退出运行，并且输出报错信息。使用redis-check-aof工具可以修复上述的只增文件，这个工具将会从上述文件中删除执行不完全的事务，这样Redis服务器才能再次启动。 从2.2版本开始，除了上述两项保证之外，Redis还能够以乐观锁的形式提供更多的保证，这种形式非常类似于“检查再设置”（CAS：Check And Set）操作。本文稍后会对Redis的乐观锁进行描述。

**（2）相关命令**

**1）. MULTI 用于标记事务块的开始。**Redis会将后续的命令逐个放入队列中，然后才能使用EXEC命令原子化地执行这个命令序列。 这个命令的运行格式如下所示：

MULTI 这个命令的返回值是一个简单的字符串，总是OK。

**2）. EXEC 在一个事务中执行所有先前放入队列的命令**，然后恢复正常的连接状态。 当使用WATCH命令时，只有当受监控的键没有被修改时，EXEC命令才会执行事务中的命令，这种方式利用了检查再设置（CAS）的机制。 这个命令的运行格式如下所示：EXEC 这个命令的返回值是一个数组，其中的每个元素分别是原子化事务中的每个命令的返回值。 当使用WATCH命令时，如果事务执行中止，那么EXEC命令就会返回一个Null值。

**3）. DISCARD 清除所有先前在一个事务中放入队列的命**令，然后恢复正常的连接状态。 如果使用了WATCH命令，那么DISCARD命令就会将当前连接监控的所有键取消监控。 这个命令的运行格式如下所示： DISCARD 这个命令的返回值是一个简单的字符串，总是OK。

**4）. WATCH 当某个事务需要按条件执行时**，就要使用这个命令将给定的键设置为受监控的。 这个命令的运行格式如下所示： WATCH key [key ...] 这个命令的返回值是一个简单的字符串，总是OK。 对于每个键来说，时间复杂度总是O(1)。

**5）. UNWATCH 清除所有先前为一个事务监控的键。** 如果你调用了EXEC或DISCARD命令，那么就不需要手动调用UNWATCH命令。 这个命令的运行格式如下所示： UNWATCH 这个命令的返回值是一个简单的字符串，总是OK。 时间复杂度总是O(1)。

1. **使用方法**

使用MULTI命令便可以进入一个Redis事务。这个命令的返回值总是OK。此时，用户可以发出多个Redis命令。Redis会将这些命令放入队列，而不是执行这些命令。一旦调用EXEC命令，那么Redis就会执行事务中的所有命令。 相反，调用DISCARD命令将会清除事务队列，然后退出事务。 以下示例会原子化地递增foo键和bar键的值： <img> 正如从上面的会话所看到的一样，EXEC命令的返回值是一个数组，其中的每个元素都分别是事务中的每个命令的返回值，返回值的顺序和命令的发出顺序是相同的。 当一个Redis连接正处于MULTI请求的上下文中时，通过这个连接发出的所有命令的返回值都是QUEUE字符串（从Redis协议的角度来看，返回值是作为状态回复（Status Reply）来发送的）。当调用EXEC命令时，Redis会简单地调度执行事务队列中的命令。

1. **事务内部的错误**

在一个事务的运行期间，可能会遇到两种类型的命令错误：一个命令可能会在被放入队列时失败。因此，事务有可能在调用EXEC命令之前就发生错误。例如，这个命令可能会有语法错误（参数的数量错误、命令名称错误，等等），或者可能会有某些临界条件（例如：如果使用maxmemory指令，为Redis服务器配置内存限制，那么就可能会有内存溢出条件）。

在调用EXEC命令之后，事务中的某个命令可能会执行失败。例如，我们对某个键执行了错误类型的操作（例如，对一个字符串（String）类型的键执行列表（List）类型的操作）。 可以使用Redis客户端检测第一种类型的错误，在调用EXEC命令之前，这些客户端可以检查被放入队列的命令的返回值：如果命令的返回值是QUEUE字符串，那么就表示已经正确地将这个命令放入队列；否则，Redis将返回一个错误。如果将某个命令放入队列时发生错误，那么大多数客户端将会中止事务，并且丢弃这个事务。 然而，从Redis 2.6.5版本开始，服务器会记住事务积累命令期间发生的错误。

然后，Redis会拒绝执行这个事务，在运行EXEC命令之后，便会返回一个错误消息。最后，Redis会自动丢弃这个事务。 在Redis 2.6.5版本之前，如果发生了上述的错误，那么在客户端调用了EXEC命令之后，Redis还是会运行这个出错的事务，执行已经成功放入事务队列的命令，而不会关心先前发生的错误。从2.6.5版本开始，Redis在遭遇上述错误时，会采用先前描述的新行为，这样便能轻松地混合使用事务和管道。在这种情况下，客户端可以一次性地将整个事务发送至Redis服务器，稍后再一次性地读取所有的返回值。 相反，在调用EXEC命令之后发生的事务错误，Redis不会进行任何特殊处理：在事务运行期间，即使某个命令运行失败，所有其他的命令也将会继续执行。 这种行为在协议层面上更加清晰。

在以下示例中，当事务正在运行时，有一条命令将会执行失败，即使这条命令的语法是正确的： <img> 上述示例的EXEC命令的返回值是批量的字符串，包含两个元素，一个是OK代码，另一个是-ERR错误消息。客户端会根据自身的程序库，选择一种合适的方式，将错误信息提供给用户 需要注意的是，即使某个命令执行失败，事务队列中的所有其他命令仍然会执行 —— Redis不会停止执行事务中的命令。 再看另一个示例，再次使用telnet通信协议，观察命令的语法错误是如何尽快报告给用户的： <img> 这一次，由于INCR命令的语法错误，Redis根本就没有将这个命令放入事务队列。

1. **为什么Redis不支持回滚？**

如果你具备关系型数据库的知识背景，你就会发现一个事实：在事务运行期间，虽然Redis命令可能会执行失败，但是Redis仍然会执行事务中余下的其他命令，而不会执行回滚操作，你可能会觉得这种行为很奇怪。 然而，这种行为也有其合理之处： 只有当被调用的Redis命令有语法错误时，这条命令才会执行失败（在将这个命令放入事务队列期间，Redis能够发现此类问题），或者对某个键执行不符合其数据类型的操作：实际上，这就意味着只有程序错误才会导致Redis命令执行失败，这种错误很有可能在程序开发期间发现，一般很少在生产环境发现。

Redis已经在系统内部进行功能简化，这样可以确保更快的运行速度，因为Redis不需要事务回滚的能力。 对于Redis事务的这种行为，有一个普遍的反对观点，那就是程序有可能会有缺陷（bug）。但是，你应当注意到：事务回滚并不能解决任何程序错误。例如，如果某个查询会将一个键的值递增2，而不是1，或者递增错误的键，那么事务回滚机制是没有办法解决这些程序问题的。请注意，没有人能解决程序员自己的错误，这种错误可能会导致Redis命令执行失败。正因为这些程序错误不大可能会进入生产环境，所以我们在开发Redis时选用更加简单和快速的方法，没有实现错误回滚的功能。

1. **丢弃命令队列 DISCARD命令可以用来中止事务运行。**

在这种情况下，不会执行事务中的任何命令，并且会将Redis连接恢复为正常状态。示例如下所示

**（6）通过CAS操作实现乐观锁 Redis使用WATCH命令实现事务的“检查再设置”（CAS）行为。** 作为WATCH命令的参数的键会受到Redis的监控，Redis能够检测到它们的变化。在执行EXEC命令之前，如果Redis检测到至少有一个键被修改了，那么整个事务便会中止运行，然后EXEC命令会返回一个Null值，提醒用户事务运行失败。 例如，设想我们需要将某个键的值自动递增1（假设Redis没有INCR命令）。 首次尝试的伪码可能如下所示：

val = GET mykey

val = val + 1

SET mykey $val

如果我们只有一个Redis客户端在一段指定的时间之内执行上述伪码的操作，那么这段伪码将能够可靠的工作。如果有多个客户端大约在同一时间尝试递增这个键的值，那么将会产生竞争状态。例如，客户端-A和客户端-B都会读取这个键的旧值（例如：10）。这两个客户端都会将这个键的值递增至11，最后使用SET命令将这个键的新值设置为11。因此，这个键的最终值是11，而不是12。 现在，我们可以使用WATCH命令完美地解决上述的问题，伪码如下所示： WATCH mykey

val = GET mykey

val = val + 1

MULTI

SET mykey $val

EXEC 由上述伪码可知，如果存在竞争状态，并且有另一个客户端在我们调用WATCH命令和EXEC命令之间的时间内修改了val变量的结果，那么事务将会运行失败。 我们只需要重复执行上述伪码的操作，希望此次运行不会再出现竞争状态。这种形式的锁就被称为乐观锁，它是一种非常强大的锁。在许多用例中，多个客户端可能会访问不同的键，因此不太可能发生冲突 —— 也就是说，通常没有必要重复执行上述伪码的操作。

1. **WATCH命令详解**

那么WATCH命令实际做了些什么呢？这个命令会使得EXEC命令在满足某些条件时才会运行事务：我们要求Redis只有在所有受监控的键都没有被修改时，才会执行事务。（但是，相同的客户端可能会在事务内部修改这些键，此时这个事务不会中止运行。）否则，Redis根本就不会进入事务。（注意，如果你使用WATCH命令监控一个易失性的键，然后在你监控这个键之后，Redis再使这个键过期，那么EXEC命令仍然可以正常工作。） WATCH命令可以被调用多次。

简单说来，所有的WATCH命令都会在被调用之时立刻对相应的键进行监控，直到EXEC命令被调用之时为止。你可以在单条的WATCH命令之中，使用任意数量的键作为命令参数。 当调用EXEC命令时，所有的键都会变为未受监控的状态，Redis不会管事务是否被中止。当一个客户单连接被关闭时，所有的键也都会变为未受监控的状态。 你还可以使用UNWATCH命令（不需要任何参数），这样便能清除所有的受监控键。当我们对某些键施加乐观锁之后，这个命令有时会非常有用。因为，我们可能需要运行一个用来修改这些键的事务，但是在读取这些键的当前内容之后，我们可能不打算继续进行操作，此时便可以使用UNWATCH命令，清除所有受监控的键。在运行UNWATCH命令之后，Redis连接便可以再次自由地用于运行新事务。

如何使用WATCH命令实现ZPOP操作呢？ 本文将通过一个示例，说明如何使用WATCH命令创建一个新的原子化操作（Redis并不原生支持这个原子化操作），此处会以实现ZPOP操作为例。这个命令会以一种原子化的方式，从一个有序集合中弹出分数最低的元素。以下源码是最简单的实现方式： WATCH zset

element = ZRANGE zset 0 0

MULTI

ZREM zset element

EXEC 如果伪码中的EXEC命令执行失败（例如，返回Null值），那么我们只需要重复运行这个操作即可。

1. **Redis脚本和事务**

根据定义，Redis脚本也是事务型的。因此，你可以通过Redis事务实现的功能，同样也可以通过Redis脚本来实现，而且通常脚本更简单、更快速。 由于Redis从2.6版本才开始引入脚本特性，而事务特性是很久以前就已经存在的，所以目前的版本才有两个看起来重复的特性。但是，我们不太可能在短时间内移除对事务特性的支持。因为，即使不用求助于Redis脚本，用户仍然能够规避竞争状态，这从语义上来看是适宜的。还有另一个更重要的原因，Redis事务特性的实现复杂度是最小的。 但是，在相当长的一段时间之内，我们不大可能看到整个用户群体都只使用Redis脚本。如果发生这种情况，那么我们可能会废弃，甚至最终移除Redis事务。

**（9）代码实现incr命令：**

package cn.crxy.redis;

import static org.junit.Assert.\*;

import java.util.List;

import org.apache.commons.pool2.impl.GenericObjectPoolConfig;

import org.junit.Test;

import redis.clients.jedis.Jedis;

import redis.clients.jedis.JedisPool;

import redis.clients.jedis.JedisPoolConfig;

import redis.clients.jedis.Transaction;

public class RedisTest {

String ip = "192.168.1.170";

int port = 6379;

Jedis jedis = new Jedis(ip, port);

/\*\*

\* 仅供测试使用，单机无连接池方式

\* @throws Exception

\*/

@Test

public void test1() throws Exception {

//获取到redis服务器的链接

jedis.set("crxy", "wwww");

String value = jedis.get("crxy");

System.out.println(value);

}

/\*\*

\* 单机连接池方式

\* @throws Exception

\*/

@Test

public void test2() throws Exception {

JedisPoolConfig jedisPoolConfig = new JedisPoolConfig();

//指定连接池中最大空闲连接数

jedisPoolConfig.setMaxIdle(10);

//链接池中创建的最大连接数

jedisPoolConfig.setMaxTotal(100);

//设置创建链接的超时时间

jedisPoolConfig.setMaxWaitMillis(2000);

//表示连接池在创建链接的时候会先测试一下链接是否可用，这样可以保证连接池中的链接都可用的。

jedisPoolConfig.setTestOnBorrow(true);

//创建一个jedis连接池

JedisPool jedisPool = new JedisPool(jedisPoolConfig, ip, port);

//从连接池中获取一个链接

Jedis jedis = jedisPool.getResource();

String value = jedis.get("crxy");

System.out.println(value);

//把连接返回给连接池

jedisPool.returnResource(jedis);

} /\*\*

\* 手工实现incr命令

\* @throws Exception

\*/

@Test

public void test4() throws Exception {

//监控指定键的值

jedis.watch("a");

String value = jedis.get("a");

int parseInt = Integer.parseInt(value);

parseInt++;

System.out.println("开始休息");

Thread.currentThread().sleep(5000);

Transaction transaction = jedis.multi();

transaction.set("a", parseInt+"");

List<Object> exec = transaction.exec();//exec执行后，watch的监控状态就取消了

if(exec==null){

System.out.println("a的值被修改了，事务没有执行");

test4();

}else{

System.out.println("事务正常执行了。");

}

}

}

**17 Redis线程模型**

**（1）Redis 基于 Reactor 模式开发了自己的网络事件处理器：** 这个处理器被称为文件事件处理器（file event handler）：文件事件处理器使用 I/O 多路复用（multiplexing）程序来同时监听多个套接字， 并根据套接字目前执行的任务来为套接字关联不同的事件处理器。当被监听的套接字准备好执行连接应答（accept）、读取（read）、写入（write）、关闭（close）等操作时， 与操作相对应的文件事件就会产生， 这时文件事件处理器就会调用套接字之前关联好的事件处理器来处理这些事件。虽然文件事件处理器以单线程方式运行， 但通过使用 I/O 多路复用程序来监听多个套接字， **文件事件处理器既实现了高性能的网络通信模型， 又可以很好地与 redis 服务器中其他同样以单线程方式运行的模块进行对接， 这保持了 Redis 内部单线程设计的简单性。**

**（2）进程，线程**

**@进程：**在计算机发明之初就发现，在输入数据时（I/O速度慢），CPU是空闲的，这样就浪费了CPU资源，为了充分利用CPU资源，发明了进程，在输入程序A的数据时，程序B在占用CPU资源进行计算。

**@线程：**为了减少进程的上下文切换的损耗，满足人机交互的实时性，同时保留进程充分利用CPU资源的优点，出现了线程。

**（3）redis为什么不用多线程（不划算）**

1）纯内存操作；

2）多线程仍然会有上下文切换的损耗，虽然比进程切换损耗小；

3）采用了非阻塞I/O多路复用机制

**（4）单线程模型**

redis中的数据结构并不全是简单的kv，还有list、hash等复杂的结构，这些结构很可能会进行细粒度的操作，比如在很长的列表偶棉添加一个元素，在hash当中或者删除一个对象，这样的一个操作就会添加很多的锁，导致同步的开销大大增加，redis权衡之后选择使用单线程，突出自己功能的灵活性，在单线程基础上任何原子操作都可以无代价的实现，多复杂的数据结构都可以轻松运用。Redis客户端对服务端的每次调用都经历了发送命令、执行命令、返回结果三个过程，其中执行命令阶段，由于redis是单线程来处理命令的，所以每一条到达服务端的命令不会立即执行，所有的命令都会进入一个队列中，然后逐个被执行，并且多个客户端发送的命令的执行顺序是不确定的，但是可以确定的是不会有两个命令被同时执行，不会产生并发问题，这就是redis的单线程基本模型

**（5）单线程模型每秒万级别处理能力的原因：**

1） 纯内存访问，数据存放在内存中，内存的响应时间大约是100ns，这是redis每秒万级别访问的重要基础

2） 非阻塞I/O，redis采用epoll作为I/O多路复用技术的实现，再加上redis自身的事件处理模型将epoll中的连接、读写、关闭都转换为了时间，不在I/O上浪费过多的时间

3） 单线程避免了线程切换和竞态产生的消耗

4） redis采用单线程模型，每条命令执行如果占用大量时间，会造成其他线程的阻塞，对于redis这种高性能服务是致命的，所以redis是面向高速执行的数据

（**6）Redis为什么是单线程的？**

因为CPU不是Redis的瓶颈，Redis的瓶颈最有可能是机器内存或者网络带宽，既然单线程容易实现，而且CPU不会成为瓶颈，那就顺便成章的采用单线程的方案。如果万一CPU成为你的Redis瓶颈了，或者，你就是不想让服务器其他核闲置，那怎么办？那也很简单，你多起几个Redis进程就好了。Redis是keyvalue数据库，又不是关系数据库，数据之间没有约束。只要客户端分清哪些key放在哪个Redis进程上就可以了。redis-cluster可以帮你做的更好。

**18 Redis分页**

**（1）传统分页**

一般分页做缓存都是直接查找出来，按页放到缓存里，但是这种缓存方式有很多缺点。如缓存不能及时更新，一旦数据有变化，所有的之前的分页缓存都失效了。比如像微博这样的场景，微博下面现在有一个顶次数的排序。这个用传统的分页方式很难应对。

**（2）Redis缓存分页**

**1、String:** 主要用于存储字符串，显然不支持分页和排序。

**2、Hash:** 主要用于存储key-value型数据，评论模型中全是key-value型数据，所以在这里Hash无疑会用到。

**3、List:** 主要用于存储一个列表，列表中的每一个元素按元素的插入时的顺序进行保存，如果我们将评论模型按createDate排好序后再插入List中，似乎就能做到排序了，而且再利用List中的**LRANGE key start stop**指令还能做到分页。嗯，到这里List似乎满足了我们分页和排序的要求，但是评论还会被删除，就需要更新Redis中的数据，如果每次删除评论后都将Redis中的数据全部重新写入一次，显然不够优雅，效率也会大打折扣，如果能删除指定的数据无疑会更好，而List中涉及到删除数据的就只有LPOP和RPOP这两条指令，但LPOP和RPOP只能删除列表头和列表尾的数据，不能删除指定位置的数据，所以List也不太适合(转载的时候看了下，是有 LREM命令可以做到删除，但是LRANGE 似乎是一个耗时命令 O(N) )。

**4、Set:** 主要存储无序集合，无序！排除。

**5、SortedSet:** 主要存储有序集合，SortedSet的添加元素指令ZADD key score member [[score,member]…]会给每个添加的元素member绑定一个用于排序的值score，SortedSet就会根据score值的大小对元素进行排序，在这里就可以将createDate当作score用于排序，SortedSet中的指令ZREVRANGE key start stop又可以返回指定区间内的成员，可以用来做分页，SortedSet的指令ZREM key member可以根据key移除指定的成员，能满足删评论的要求，所以，SortedSet在这里是最适合的（时间复杂度O(log(N))）

**我需要用到的数据类型有SortSet和Hash，SortSet用于做分页排序，Hash用于存储具体的键值对数据**

**public** **static** **void** sortedSetPagenation()

{

**for** (**int** i = 1; i <= 100; i += 10)

{

// 初始化CommentId索引 SortSet

HiyaRedis.*zadd*("topicId", i, "commentId" + i);

// 初始化Comment数据 Hash

HiyaRedis.*hset*("Comment\_Key", "commentId" + i, "comment content .......");

}

// 倒序取 从0条开始取 5条 Id 数据

LinkedHashSet<String> sets = HiyaRedis.*zrevrangebyscore*("topicId", "80", "1", 0, 5);

String[] items = **new** String[]{};

System.***out***.println(sets.toString());

// 根据id取comment数据

List<String> list = HiyaRedis.*hmget*("Comment\_Key", sets.toArray(items));

**for** (String str : list)

{

System.***out***.println(str);

}

}

**19 哨兵模式**

**（1）Redis-Sentinel是官方推荐的高可用解决方案**，当redis在做master-slave的高可用方案时，假如master宕机了，redis本身（以及其很多客户端）都没有实现自动进行主备切换，而redis-sentinel本身也是独立运行的进程，可以部署在其他与redis集群可通讯的机器中监控redis集群。

**（2）它的主要功能有一下几点**

1）、不时地监控redis是否按照预期良好地运行;

2）、如果发现某个redis节点运行出现状况，能够通知另外一个进程(例如它的客户端);

3）、能够进行自动切换。当一个master节点不可用时，能够选举出master的多个slave(如果有超过一个slave的话)中的一个来作为新的master,其它的slave节点会将它所追随的master的地址改为被提升为master的slave的新地址。

4）、哨兵为客户端提供服务发现，客户端链接哨兵，哨兵提供当前master的地址然后提供服务，如果出现切换，也就是master挂了，哨兵会提供客户端一个新地址。

哨兵（sentinel）本身也是支持集群的

**哨兵(sentinel) 的一些设计思路和zookeeper非常类似**

**（3）单个哨兵会存在自己挂掉而无法监控整个集群的问题**，所以哨兵也是支持集群的，我们通常用三台哨兵机器来监控一组redis集群。

**（4）环境准备：**

centos7服务器3台,6也可以，没什么区别。

我部署好了三台

redis-1

redis-2

redis-3

清空selinux与iptables

**（5）编译安装redis**

yum install gcc\* tcl -y

mkdir /opt/soft

cd /opt/soft

wget http://download.redis.io/releases/redis-3.2.4.tar.gz

tar xf redis-3.2.4.tar.gz

cd redis-3.2.4

make

mkdir conf

mkdir bin

cp utils/redis\_init\_script bin/redis

cp redis.conf conf/6379.conf

cd ..

mv redis-3.2.4 /opt/redis

cd /opt/redis/bin

sed -i s#CLIEXEC=\/usr\/local\/bin\/redis-cli#CLIEXEC=\/opt\/redis\/src\/redis-cli#g redis

sed -i s#EXEC=\/usr\/local\/bin\/redis-server#EXEC=\/opt\/redis\/src\/redis-server#g redis

sed -i s#CONF=\"\/etc\/redis#CONF=\"\/opt\/redis\/conf#g redis

cd /opt/redis/conf

sed -i s#daemonize\ no#daemonize\ yes#g 6379.conf

**（6）安装完毕后，修改配置文件。**

vim /opt/redis/conf/6379.conf

注释此条

bind 127.0.0.1

protected-mode yes 改为 protected-mode no #关闭安全模式

至此，redis部署完毕。

redis的启动停止脚本在

/opt/redis/bin/redis stop/start

redis的配置文件在

/opt/redis/conf/6379.conf

redis的登陆命令在

/opt/redis/src/redis-cli

**（7）redis配置主从**

启动两台redis

redis-1 10.0.0.10

redis-2 10.0.0.20

若redis-1为主的话，在redis-2的配置文件中配置

slaveof 10.0.0.10 6379

修改完毕后重启redis即可，重启后我们可通过登陆进入redis后info查看主从信息。

**（8）引入哨兵。**

redis-1与redis-2搭建完毕主从后，我们开始引入哨兵。

哨兵是一个单独的程序，所以我们需要单独部署它。

若是在其他机器上部署哨兵，那么请用上面的redis安装脚本重新安装一遍redis。

在这里我已经部署完毕了

redis-1

redis-2

redis-3

增加哨兵的配置文件。三台redis都需要增加，文件内容这三台一样。

vim /opt/redis/conf/sentinel.conf

##sentinel for 10.0.0.10 ,its slave is 10.0.0.20

#master1

port 26383

sentinel monitor master1 10.0.0.10 6379 2

sentinel down-after-milliseconds master1 30000

sentinel failover-timeout master1 900000

sentinel parallel-syncs master1 1

#sentinel auth-pass mymaster 123456　　#如果你的redis集群有密码

**（9）启动哨兵**

三台机器都是一个操作方式。

添加窗口

screen -S sentinel

在新窗口启动哨兵

/opt/redis/src/redis-sentinel /opt/redis/conf/sentinel.conf --protected-mode no

启动后即可看到前台输出信息。

后台挂起这个窗口请按：

Ctrl+a+d

下次返回观看这个窗口请输入

screen -r sentinel

我们这里暂时不挂起窗口，可以观察哨兵监控集群的状态。我们接下来我们进行切换以及增加新的salve节点测试。关掉redis-1并查看哨兵监控的状态，约30秒内，哨兵探测redis-1客观故障后，即会重新选举新的master，重新选举完毕后我们在redis-2中info查看主从状态，会发现redis-2已经被选举为master。重新启动redis-1，并不需要修改配置文件，启动后redis-1自动会与redis-2进行同步。修改redis-3的配置文件，把slaveof指向到redis-2，启动后你会发现哨兵会把redis-3自动添加到集群中。

**20 Redis应用场景**

**1）.取最新N个数据的操作**

比如典型的取你网站的最新文章，通过下面方式，我们可以将最新的5000条评论的ID放在Redis的List集合中，并将超出集合部分从数据库获取

使用LPUSH latest.comments<ID>命令，向list集合中插入数据

插入完成后再用LTRIM latest.comments 0 5000命令使其永远只保存最近5000个ID

然后我们在客户端获取某一页评论时可以用下面的逻辑（伪代码）

FUNCTION get\_latest\_comments(start,num\_items):

id\_list = redis.lrange("latest.comments",start,start+num\_items-1)

IF id\_list.length < num\_items

id\_list = SQL\_DB("SELECT ... ORDER BY time LIMIT ...")

END

RETURN id\_list

END

如果你还有不同的筛选维度，比如某个分类的最新N条，那么你可以再建一个按此分类的List，只存ID的话，Redis是非常高效的。

**2）.排行榜应用，取TOP N操作**

这个需求与上面需求的不同之处在于，前面操作以时间为权重，这个是以某个条件为权重，比如按顶的次数排序，这时候就需要我们的sorted set出马了，将你要排序的值设置成sorted set的score，将具体的数据设置成相应的value，每次只需要执行一条ZADD命令即可。

**3）.需要精准设定过期时间的应用**

比如你可以把上面说到的sorted set的score值设置成过期时间的时间戳，那么就可以简单地通过过期时间排序，定时清除过期数据了，不仅是清除Redis中的过期数据，你完全可以把Redis里这个过期时间当成是对数据库中数据的索引，用Redis来找出哪些数据需要过期删除，然后再精准地从数据库中删除相应的记录。

**4）.计数器应用**

Redis的命令都是原子性的，你可以轻松地利用INCR，DECR命令来构建计数器系统。

**5）.Uniq操作，获取某段时间所有数据排重值**

这个使用Redis的set数据结构最合适了，只需要不断地将数据往set中扔就行了，set意为集合，所以会自动排重。

**6）.实时系统，反垃圾系统**

通过上面说到的set功能，你可以知道一个终端用户是否进行了某个操作，可以找到其操作的集合并进行分析统计对比等。没有做不到，只有想不到。

**7）.Pub/Sub构建实时消息系统**

Redis的Pub/Sub系统可以构建实时的消息系统，比如很多用Pub/Sub构建的实时聊天系统的例子。

**8）.构建队列系统**

使用list可以构建队列系统，使用sorted set甚至可以构建有优先级的队列系统。

**9）.缓存**

这个不必说了，性能优于Memcached（在某些方面，并不是全面优于），数据结构更多样化。

**21 同步原理**

从服务器会向主服务器发出SYNC指令，当主服务器接到此命令后，就会调用BGSAVE指令来创建一个子进程专门进行数据持久化工作，也就是将主服务器的数据写入RDB文件中。在数据持久化期间，主服务器将执行的写指令都缓存在内存中。在BGSAVE指令执行完成后，主服务器会将持久化好的RDB文件发送给从服务器，从服务器接到此文件后会将其存储到磁盘上，然后再将其读取到内存中。这个动作完成后，主服务器会将这段时间缓存的写指令再以redis协议的格式发送给从服务器。

另外，要说的一点是，即使有多个从服务器同时发来SYNC指令，主服务器也只会执行一次BGSAVE，然后把持久化好的RDB文件发给多个下游。在redis2.8版本之前，如果从服务器与主服务器因某些原因断开连接的话，都会进行一次主从之间的全量的数据同步；而在2.8版本之后，redis支持了效率更高的增量同步策略，这大大降低了连接断开的恢复成本。主服务器会在内存中维护一个缓冲区，缓冲区中存储着将要发给从服务器的内容。从服务器在与主服务器出现网络瞬断之后，从服务器会尝试再次与主服务器连接，一旦连接成功，从服务器就会把“希望同步的主服务器ID”和“希望请求的数据的偏移位置（replication offset）”发送出去。主服务器接收到这样的同步请求后，首先会验证主服务器ID是否和自己的ID匹配，其次会检查“请求的偏移位置”是否存在于自己的缓冲区中，如果两者都满足的话，主服务器就会向从服务器发送增量内容。增量同步功能，需要服务器端支持全新的PSYNC指令。这个指令，只有在redis-2.8之后才具有。

**22 Redis 分区**

1. **分区是分割数据到多个Redis实例的处理过程，**因此每个实例只保存key的一个子集。

**（2）分区的优势**

@通过利用多台计算机内存的和值，允许我们构造更大的数据库。

@通过多核和多台计算机，允许我们扩展计算能力；通过多台计算机和网络适配器，允许我们扩展网络带宽。

**（3）分区的不足**

@涉及多个key的操作通常是不被支持的。举例来说，当两个set映射到不同的redis实例上时，你就不能对这两个set执行交集操作。

@涉及多个key的redis事务不能使用。

@当使用分区时，数据处理较为复杂，比如你需要处理多个rdb/aof文件，并且从多个实例和主机备份持久化文件。

@增加或删除容量也比较复杂。redis集群大多数支持在运行时增加、删除节点的透明数据平衡的能力，但是类似于客户端分区、代理等其他系统则不支持这项特性。然而，一种叫做presharding的技术对此是有帮助的。

**（4）分区类型**

Redis 有两种类型分区。 假设有4个Redis实例 R0，R1，R2，R3，和类似user:1，user:2这样的表示用户的多个key，对既定的key有多种不同方式来选择这个key存放在哪个实例中。也就是说，有不同的系统来映射某个key到某个Redis服务。

**1）范围分区**

最简单的分区方式是按范围分区，就是映射一定范围的对象到特定的Redis实例。比如，ID从0到10000的用户会保存到实例R0，ID从10001到 20000的用户会保存到R1，以此类推。这种方式是可行的，并且在实际中使用，不足就是要有一个区间范围到实例的映射表。这个表要被管理，同时还需要各 种对象的映射表，通常对Redis来说并非是好的方法。

**2）哈希分区**

另外一种分区方法是hash分区。这对任何key都适用，也无需是object\_name:这种形式，像下面描述的一样简单：用一个hash函数将key转换为一个数字，比如使用crc32 hash函数。对key foobar执行crc32(foobar)会输出类似93024922的整数。对这个整数取模，将其转化为0-3之间的数字，就可以将这个整数映射到4个Redis实例中的一个了。93024922 % 4 = 2，就是说key foobar应该被存到R2实例中。注意：取模操作是取除的余数，通常在多种编程语言中用%操作符实现。

**23 Pipeline管道**

可以将多次IO往返的时间缩减为一次，前提是pipeline执行的指令之间没有因果相关性。使用redis-benchmark进行压测的时候可以发现影响redis的QPS峰值的一个重要因素是pipeline批次指令的数目。redis的底层通信协议对管道提供了支持。通过管道可以一次性发送多条命令并在执行完后一次性将结果返回，当一组命令中每条命令都不依赖之前命令的执行结果时就可以将这组命令一起通过管道发出。管道通过减少客户端与redis的通信次数来实现降低往返实验累计值的目的。

## 三、JAVA操作API

**1 开始在 Java 中使用 Redis 前，** 我们需要确保已经安装了 redis 服务及 Java redis 驱动且你的机器上能正常使用 Java。 Java的安装配置可以参考我们的 Java开发环境配置 接下来让我们安装 Java redis 驱动：首先你需要下载驱动包 下载 jedis.jar，确保下载最新驱动包。在你的 classpath 中包含该驱动包。

jedis-2.9.0.jar

**2 连接到 redis 服务**

import redis.clients.jedis.Jedis;

public class RedisJava {

public static void main(String[] args) {

//连接本地的 Redis 服务

Jedis jedis = new Jedis("localhost");

System.out.println("连接成功");

//查看服务是否运行

System.out.println("服务正在运行: "+jedis.ping());

}

}

连接成功

服务正在运行: PONG

**2 Redis Java String(字符串) 实例**

import redis.clients.jedis.Jedis;

public class RedisStringJava {

public static void main(String[] args) {

//连接本地的 Redis 服务

Jedis jedis = new Jedis("localhost");

System.out.println("连接成功");

//设置 redis 字符串数据

jedis.set("runoobkey", "www.runoob.com");

// 获取存储的数据并输出

System.out.println("redis 存储的字符串为: "+ jedis.get("runoobkey"));

}

}

连接成功

redis 存储的字符串为: www.runoob.com

**3 Redis Java List(列表) 实例**

import java.util.List;

import redis.clients.jedis.Jedis;

public class RedisListJava {

public static void main(String[] args) {

//连接本地的 Redis 服务

Jedis jedis = new Jedis("localhost");

System.out.println("连接成功");

//存储数据到列表中

jedis.lpush("site-list", "Runoob");

jedis.lpush("site-list", "Google");

jedis.lpush("site-list", "Taobao");

// 获取存储的数据并输出

List<String> list = jedis.lrange("site-list", 0 ,2);

for(int i=0; i<list.size(); i++) {

System.out.println("列表项为: "+list.get(i));

}

}

}

连接成功

列表项为: Taobao

列表项为: Google

列表项为: Runoob

**4 Redis Java Keys 实例**

import java.util.Iterator;

import java.util.Set;

import redis.clients.jedis.Jedis;

public class RedisKeyJava {

public static void main(String[] args) {

//连接本地的 Redis 服务

Jedis jedis = new Jedis("localhost");

System.out.println("连接成功");

// 获取数据并输出

Set<String> keys = jedis.keys("\*");

Iterator<String> it=keys.iterator() ;

while(it.hasNext()){

String key = it.next();

System.out.println(key);

}

}

}

**5 Java连redis集群的例子：**

RedisClusterConnection.java

public class RedisClusterConnection {

private static Logger LOG = LogManager.getLogger(RedisClusterConnection.class);

private static JedisCluster jedisCluster;

private static Pattern p = Pattern.compile("^.+[:]\\d{1,5}\\s\*$");

private static Set<HostAndPort> parseHostAndPort() {

InputStream in = null;

try {

Properties properties = new Properties();

in = ClassLoader.getSystemClassLoader().getResourceAsStream(

"redis-config.properties");

properties.load(in);

Set<HostAndPort> haps = new HashSet<HostAndPort>();

for (Object key : properties.keySet()) {

if (!key.toString().startsWith("redis.address")) {

continue;

}

String val = properties.get(key).toString();

boolean isIpPort = p.matcher(val).matches();

if (!isIpPort) {

throw new IllegalArgumentException("ip 或 port 不合法");

}

String[] ipAndPort = val.split(":");

HostAndPort hap = new HostAndPort(ipAndPort[0],

Integer.parseInt(ipAndPort[1]));

haps.add(hap);

}

return haps;

} catch (InvalidPropertiesFormatException e) {

LOG.error(e);

} catch (IOException e1) {

LOG.error(e1);

} catch (Exception e2) {

LOG.error(e2);

} finally {

if (in != null) {

try {

in.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

return null;

}

public static JedisCluster getInstance() {

Set<HostAndPort> haps = parseHostAndPort();

GenericObjectPoolConfig poolConfig = new GenericObjectPoolConfig();

poolConfig.setMaxTotal(1024);

poolConfig.setMaxIdle(10);

poolConfig.setMaxWaitMillis(1800000L);

poolConfig.setTimeBetweenEvictionRunsMillis(180000L);

poolConfig.setTestOnBorrow(true);

jedisCluster = new JedisCluster(haps, poolConfig);

return jedisCluster;

}

public static void main(String[] args) {

Set<HostAndPort> set = parseHostAndPort();

Iterator<HostAndPort> iterator = set.iterator();

while (iterator.hasNext()) {

HostAndPort hap = iterator.next();

System.out.println(hap.getHost() + ":" + hap.getPort());

}

}

}

6 redis-config.properties

redis.address1=192.168.0.1:6379

redis.address2=192.168.0.1:6380

redis.address3=192.168.0.1:6381

redis.address4=192.168.0.1:6382

redis.address5=192.168.0.1:6383

redis.address6=192.168.0.1:6384

## Redis集群和高可用

**1 使用较多的redis部署方式:**

通过上表比较可知：如果需要完整的分片、复制和高可用特性，在集群节点不多且在使用sentinel这种模式会带来性能瓶颈和资源消耗的情况下，可以选择使用 Redis集群；如果只需要一部分特性（比如只需要分片，但不需要复制和高可用），那么可以选择Redis Sentinel。

**2 单实例模式**

单实例模式是指单台redis完成所有请求任务，因此复用和不具备容错性；同时在单台机器上如果只启用一个redis实例会造成资源浪费 。

**3 Redis集群**

Redis 集群是一个由多个节点组成的分布式服务器群，它具有复制、高可用和分片特性。Redis集群不需要sentinel哨兵也能完成节点移除和故障转移的功能。需要将每个节点设置成集群模式，这种集群模式没有中心节点，多个节点之间存在着网络通信的消耗。多个节点按照分片来处理不同位置的槽，接受到不属于自己的槽操作的命令时会重新发送命令给正确的节点，这中间必然有一定的资源消耗。如同redis主从配置使用sentinel作为代理来处理请求一样。

**4 redis最核心的目标：**

**▪ 性能：**这是Redis赖以生存的看家本领，增加集群功能后当然不能对性能产生太大影响，所以Redis采取了P2P而非Proxy方式、异步复制、客户端重定向等设计，而牺牲了部分的一致性、使用性。

**▪ 水平扩展：**集群的最重要能力当然是扩展，文档中称可以线性扩展到1000节点。

**▪ 可用性：**在Cluster推出之前，可用性要靠Sentinel保证。有了集群之后也自动具有了Sentinel的监控和自动Failover能力。

**5 分布式**

Redis集群是一个由多个Redis服务器组成的分布式网络服务器群，集群中的各个服务器被称为节点（node），这些节点会相互连接并进行通信。分布式的Redis集群没有中心节点，所以用户不必担心某个节点会成为整个集群的性能瓶颈。

**6 复制**

Redis 集群的每个节点都有两种角色可选，一个是主节点（master node），另一个是从节点（slavenode），其中主节点用于储存数据，而从节点则是某个主节点的复制品。当用户需要处理更多读请求的时候，可以添加从节点以扩展系统的读性能。因为Redis集群重用了单机Redis复制特性的代码，所以集群的复制行为和我们之前介绍的单机复制特性的行为是完全一样的。

**7 节点故障检测和自动故障转移**

Redis 集群的主节点内置了类似Redis Sentinel的节点故障检测和自动故障转移功能，当集群中的某个主节点下线时，集群中的其他在线主节点会注意到这一点，并对已下线的主节点进行故障转移。集群进行故障转移的方法和Redis Sentinel进行故障转移的方法基本一样，不同的是，在集群里面，故障转移是由集群中其他在线的主节点负责进行的，所以集群不必另外使用Redis Sentinel 。

**8 分片**

集群使用分片来扩展数据库的容量，并将命令请求的负载交给不同的节点来分担。集群将整个数据库分为 16384 个槽（slot），所有键都属于这 16384 个槽的其中一个，计算键 key属于哪个槽的公式为 slot\_number = crc16(key) % 16384 ，其中 crc16 为 16 位的循环冗余校验和函数。集群中的每个主节点都可以处理 0 个至 16384 个槽，当 16384 个槽都有某个节点在负责处理时，集群进入上线状态，并开始处理客户端发送的数据命令请求。

例如，我们有三个主节点7000、7001 和 7002，那么我们可以：

将槽0至5460指派给节点7000负责处理；

将槽 5461至 10922 指派给节点 7001 负责处理；

将槽 10923至 16383指派给节点 7002 负责处理；

这样就可以将16384个槽平均地指派给三个节点负责处理。

**9 转向**

对于一个被指派了槽的主节点来说，这个主节点只会处理属于指派给自己的槽的命令请求。如果一个节点接收到了与自己处理的槽无关的命令请求，那么节点会向客户端返回一个转向错误（redirection error），告诉客户端，哪个节点负责处理这条命令，之后客户端需要根据错误中包含的地址和端口号重新向正确的节点发送命令请求。

**10 Redis集群客户端**

因为集群功能比起单机功能要复杂得多，所以不同语言的 Redis 客户端通常需要为集群添加特别的支持，或者专门开发一个集群客户端。

目前主要的 Redis 集群客户端（或者说，支持集群功能的 Redis 客户端）有以下这些：

- redis-rb-cluster：antirez 使用 Ruby 编写的 Redis 集群客户端，集群客户端的官方实现。

- predis：Redis 的 PHP 客户端，支持集群功能。

- jedis：Redis 的 JAVA 客户端，支持集群功能。

- StackExchange.Redis：Redis 的 C# 客户端，支持集群功能。

-内置的 redis-cli ：在启动时给定 -c 参数即可进入集群模式，支持部分集群功能。

**11 Redis Sentinel集群**

Sentinel是一个管理redis实例的工具，它可以实现对redis的监控、通知、自动故障转移。sentinel不断地检测redis实例是否可以正常工作，通过API向其他程序报告redis的状态，如果redis master不能工作，则会自动启动故障转移进程，将其中的一个slave提升为master，其他的slave重新设置新的master服务器。

**12 Sentinel主要功能**

**▪ 监控（Monitoring）：**实时监控主服务器和从服务器运行状态。

**▪ 提醒（Notification）：**当被监控的某个Redis服务器出现问题时， Redis Sentinel可以向系统管理员发送通知，也可以通过API向其他程序发送通知。

**▪ 自动故障转移（Automatic failover）：**当一个主服务器不能正常工作时，Sentinel会开始一次自动故障迁移操作，它会将失效主服务器的其中一个从服务器升级为新的主服务器，

并让失效主服务器的其他从服务器改为复制新的主服务器，当客户端试图连接失效的主服务器时集群也会向客户做出正确的应答。

**13 Redis Sentinel备份策略**

Redis提供两种相对有效的备份方法：RDB和AOF。

**（一）RDB持久化设置**

RDB是在某个时间点将内存中的所有数据的快照保存到磁盘上，在数据恢复时，可以恢复备份时间以前的所有数据，但无法恢复备份时间点后面的数据。默认情况下Redis在磁盘上创建二进制格式的命名为dump.rdb的数据快照。可以通过配置文件配置每隔N秒且数据集上至少有M个变化时创建快照、是否对数据进行压缩、快照名称、存放快照的工作目录。

**（二）AOF持久化设置**

AOF是以协议文本的方式，将所有对数据库进行过写入的命令（及其参数）记录到 AOF 文件，以此达到记录数据库状态的目的。

优点是基本可以实现数据无丢失（缓存的数据有可能丢失）

缺点是随着数据量的持续增加，AOF文件也会越来越大。

在保证数据安全的情况下，尽量避免因备份数据消耗过多的Redis资源，采用如下备份策略：主实例：不采用任何备份机制。Slave端：采用AOF（严格数据要求时可同时开启RDB），每天将AOF文件备份至备份服务器。为了最大限度减少主实例的资源干扰，将备份相关全部迁移至Slave端完成。同时这样也有缺点，当主实例挂掉后，应用服务切换至Slave端，此时的Slave端的负载将会很大。目前Redis不支持RDB和AOF参数动态修改，需要重启Redis生效，希望能在新的版本中实现更高效的修改方式。利用快照的持久化方式不是非常可靠，当运行Redis的计算机停止工作、意外掉电、意外杀掉了Redis进程那么最近写入Redis的数据将会丢。对于某些应用这或许不成问题，但对于持久化要求非常高的应用场景快照方式不是理想的选择。AOF文件是一个替代方案，用以最大限度的持久化数据。同样，可以通过配置文件来开闭AOF。

当主实例Redis服务崩溃（包含主机断电、进程消失等），Redis sentinel将Slave切换为读写状态，提供生产服务。通过故障诊断修复主实例，启动后会自动加入Sentinel并从Slave端完成数据同步，但不会切换。当主实例和Slave同时崩溃（如机房断电），启动服务器后，将备份服务器最新的AOF备份拷贝至主实例，启动主实例。一切完成后再启动Slave。

**14 如何在Python 下使用Redis Sentinel？**

首先安装redis-py。

一个简单的测试代码如下，首先获得一个Sentinel对象，然后键入命令vim sentinel.py

执行后成功得到zhanyz这个值，键入python sentinel.py。

上面的master和slave都是标准的建立好连接的StrictRedis实例，slave则是sentinel查询到的第一个可用的slave。

如果正在连接的master不可用时，客户端会先抛出redis.exceptions.ConnectionError异常（此时还未开始failover），然后抛出redis.sentinel.MasterNotFoundError异常（failover进行中），在sentinel正常failover之后，实例正常。

**15 如何在JAVA 下使用Redis Sentinel？**

使用Java操作Redis需要jedis-2.1.0.jar，如果需要使用Redis连接池的话，还需commons-pool-2.4.2.jar。JedisTemplate提供了一个template方法，负责对Jedis连接的获取与归还。

注意：

1)cluster环境下redis的slave不接受任何读写操作，比如sentinel模式下只有slave升级为主实例时才能进行读写操作。

2)client端不支持keys批量操作,不支持select dbNum操作，只有一个db:select 0，这个十分尴尬。

3)JedisCluster 的info()等单机函数无法调用,返回(No way to dispatch this command to Redis Cluster)错误。.

4)JedisCluster 没有针对byte[]的API，需要自己扩展。

5) JedisTemplate中具体代码可以参见附件中JedisTemplate.java



