1. **Redis简介**

**1.1 概述**

开源的、用C语言编写的、支持网络交互的、可基于内存也可持久化的Key-Value数据库。

Redis是一个高性能、基于键值对(Key-Value)的缓存和存储系统；

是一个速度非常快的非关系型数据库,Redis数据库完全在内存中；

支持数据的持久化，使用磁盘仅用于持久性；

不区分命令大小写

用ANSI C编写。

采用了内存中数据集方式。

可每隔一段时间将数据集转存到磁盘上来持久化数据，或在日志尾部追加每一条操作命令。

支持主从复制，且具有非常快速的非阻塞首次同步、网络断开自动重连等功能。

其它特性:check-and-set机制、pub/sub和配置设置等

**1.2 Redis客户端管理软件：**

1、Redis Desktop Manager

2、Redis Client

3、Redis Studio

1. **Redis键命令**

<http://redis.readthedocs.io/en/2.4/index.html>

* key是字符串类型，但不能包括边界字符。

由于key不是binary safe的字符串，所以像"my key"和"mykey\n"这样包含空格和换行的key是不允许的。在redis内部并不限制使用binary字符，这是redis协议限制的。

* "\r\n"在协议格式中会作为特殊字符。redis 1.2以后的协议中部分命令已开始使用新协议

格式了(比如MSET)。总之目前还是把包含边界字符当成非法的key吧，免得被bug纠缠。另外关于key的一个格式约定介绍下，object-type:id:field。比如user:1000:password，blog:xxidxx:title。还有key的长度最好不要太长。道理很明显占内存啊，且查找时候相对短key也更慢。不过也不推荐过短的key，比如u:1000:pwd。显然没上面的user:1000:password可读性好。



|  |  |
| --- | --- |
| **exits** key | [测试](http://lib.csdn.net/base/softwaretest)key是否存在，返回1:存在，0:不存在 |
| **del** key1....keyN | 删除给定key，返回删除key的数目，0:给定key都不存在 |
| type key | 返回给定key的value类型(list set string)。none:不存在key |
| random key | 返回从当前[**数据库**](http://lib.csdn.net/base/mysql)中随机选择的一个key，如当前数据库空,返回空串 |
| rename oldkey newkey | 原子的重命名一个key，如果newkey存在，将会被覆盖，返回1表示成功，0失败。可能是oldkey不存在或者和newkey相同 |
| renamenx oldkey newkey | 同上，但是如果newkey存在返回失败 |
| dbsize | 返回当前数据库的key数量 |
| expire key seconds | 为key指定过期时间，单位秒。返回1成功。 |
| ttl key | 返回设置过过期时间的key的剩余过期秒数 -1表示key不存在或者没有设置过过期时间 |
| select db-index | 将key从当前数据库移动到指定数据库。返回1成功。0 如果key不存在，或者已经在指定数据库中 |
| move key db-index | 将key从当前数据库移动到指定数据库。返回1成功。0 如果key不存在，或者已经在指定数据库中 |
| flushdb | 删除当前数据库中所有key，此方法不会失败。慎用 |
| flushall | 删除所有数据库中的所有key，此方法不会失败。慎用 |

1. **Redis数据类型**

|  |
| --- |
| string 字符串  hash 哈希  list 列表  set 集合  sorted set 有序集合 |

可执行原子操作。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。

**3.1 String 字符串类型：**

* 是redis最基本的类型，且是二进制安全的。

意思是redis的string可包含任何数据。如jpg图片或序列化的对象。

从内部实现来看其实string可看作byte数组，最大上限是1G字节。

* string类型的定义。

|  |
| --- |
| struct sdshdr {  long len;  long free;  char buf[];  } |

buf是个char数组用于存贮实际的字符串内容。

char是一个字节。

len是buf数组的长度，free是数组中剩余可用字节数。

由此可以理解为什么string类型是二进制安全的了。因为它本质上就是个byte数组。当然可以包含任何数据了。

* 另外string类型可被部分命令按int处理。比如incr等命令。

redis的其他类型像list，set，sorted set,hash, 包含的元素与只能是string类型。

如只用string类型，redis就可被看作加上持久化特性的memcached。当然redis对string类型的操作比memcached多很多啊。如下：

|  |  |
| --- | --- |
| set key value | 设置key对应值为string类型的value.1:成功 0:失败 |
| setnx key value | 同上，如key已存在，返回0 。nx 是not exist的意思 |
| get key | 获取key对应的string值,如果key不存在返回nil |
| getset key value | 原子的设置key值,并返回key的旧值。如key不存在返回nil |
| mget key1 key2 ... keyN | 一次获取多个key值，如对应key不存在，则返回nil。 |
| mset key1 value1 ... keyN valueN | 一次设置多个key的值，成功返回1表示所有的值都设置了，失败返回0表示没有任何值被设置 |
| msetnx key1 value1 ... keyN valueN | 同上，但不会覆盖已经存在的key |
| incr key | 对key值做加加操作,并返回新值。注意incr一个不是int的value会返回错误，incr一个不存在的key，则设置key为1 |
| decr key | 减减操作，decr一个不存在key，则设置key为-1 |
| incrby key integer | 同incr,加指定值,key不存在时设置key,并认为原来value= 0 |
| decrby key integer | 同decr，减指定值。decrby完全是为了可读性，我们完全可以通过incrby一个负值来实现同样效果，反之一样。 |
| append key value | 给指定key的字符串值追加value,返回新字符串值的长度。 |
| substr key start end | 返回截取过的key的字符串值,注意并不修改key的值。 下标从0开始 |

* 示例:

|  |
| --- |
| redis 127.0.0.1:6379> set key hello  OK  redis 127.0.0.1:6379> append k ,world  (integer) 11  redis 127.0.0.1:6379> get k  "hello,world"  redis 127.0.0.1:6379> substr k 0 8  "hello,wor"  redis 127.0.0.1:6379> get k  "hello,world"  redis 127.0.0.1:6379> set k1 a  OK  redis 127.0.0.1:6379> set k2 b  OK  redis 127.0.0.1:6379> mget k1 k2 k3  1. "a"  2. "b"  3. (nil) |

* 例子:

|  |
| --- |
| 127.0.0.1:6379> set mynum "2"  OK  127.0.0.1:6379> get mynum  "2"  127.0.0.1:6379> incr mynum  (integer) 3  127.0.0.1:6379> get mynum  "3" |

在遇到数值操作时，redis会将字符串类型转换成数值。

由于INCR等指令本身就具有原子操作特性，所以完全可利用redis的INCR、INCRBY、DECR、DECRBY等指令实现原子计数的效果

假如，在某种场景下有3个客户端同时读取了mynum的值（值为2），然后对其同时进行了加1的操作，那么，最后mynum的值一定是5。

不少网站都利用redis的这个特性来实现业务上的统计计数需求。

**3.2 hash 散列类型：**

* string类型的field和value的映射表。

添加，删除操作都是O(1)（平均）。

特别适合用于存储对象。相较于将对象的每个字段存成单个string类型。将一个对象存储在hash类型中会占用更少内存，且可更方便的存取整个对象。

* 相关命令：

|  |  |
| --- | --- |
| **hset** key field value | 设置hash field为指定值，如key不存在，则先创建 |
| **hget** key field | 获取指定的hash field |
| hmget key filed1....fieldN | 获取全部指定的hash filed |
| hmset key filed1 value1 ... filedN valueN | 同时设置hash的多个field |
| hincrby key field integer | 将指定的hash filed 加上给定值 |
| hexists key field | 测试指定field是否存在 |
| hdel key field | 删除指定的hash field |
| hlen key | 返回指定hash的field数量 |
| hkeys key | 返回hash的所有field |
| hvals key | 返回hash的所有value |
| hgetall | 返回hash的所有filed和value |

散列类型是一个键值数据结构，值只能是字符串，也就是散列数据

类型不支持嵌套其他数据类型。



**3.3 list 列表类型：**

子元素都是string类型的双向链表。

有序的字符串列表，可重复添加数据。

所以[lr]push和[lr]pop命令的[算法](http://lib.csdn.net/base/datastructure)时间复杂度都是O(1)

另外list会记录链表的长度。所以llen操作也是O(1).

链表的最大长度是(2的32次方-1)。

可通过push,pop操作从链表的头部或者尾部添加删除元素。

对于一个具有上百万个元素的lists来说，在头部和尾部插入一个新元素，其时间复杂度是常数级别，比如用LPUSH在10个元素的lists头部插入新元素，和在上千万元素的lists头部插入新元素的速度应该是相同的。

这使得list既可以用作栈，也可用作队列。有意思的是list的pop操作还有阻塞版本的。当我们[lr]pop一个list对象是，如果 list是空，或者不存在，会立即返回nil。但是阻塞版本的b[lr]pop可以则可以阻塞，当然可以加超时时间，超时后也会返回nil。

为什么要阻塞版本的pop呢，主要是为了避免轮询。举个简单的例子如果我们用list来实现一个工作队列。执行任务的thread可以调用阻塞版本的pop去获取任务这样就可以避免轮询去检查是否有任务存在。当任务来时候工作线程可以立即返回，也可以避免轮询带来的延迟。

* list相关命令：

|  |  |
| --- | --- |
| lpush key string | 在key对应list的头部添加字符串元素，返回值为添加新元素后list的元素个数，0表key存在且不是list类型 |
| rpush key string | 同上，在尾部添加 |
| llen key | 返回key对应list的长度，key不存在返回0,如key对应类型不是list返回错误 |
| lrange key start end | 返回指定区间内的元素，下标从0开始，负值表示从后面计算，-1表示倒数第一个元素 ，key不存在返回空列表 |
| ltrim key start end | 截取list,保留指定区间内元素,成功返回1,key不存在返回错误 |
| lset key index value | 设置list中指定下标的元素值,成功返回1,key或下标不存在返回错误 |
| lrem key count value | 从key对应list中删除count个和value相同的元素。  count为0时候删除全部 |
| lpop key | 从list的头部删除元素，并返回删除元素。如key对应list不存在或是空返回nil，如key对应值不是list返回错误 |
| rpop key | 同上，但是从尾部删除 |
| blpop key1...keyN timeout | 从左到右扫描返回对第一个非空list进行lpop操作并返回，比如blpop list1 list2 list3 0 ,如果list不存在list2,list3都是非空则对list2做lpop并返回从list2中删除的元素。如果所有的list都是空或不存在，则会 阻塞timeout秒，timeout为0表示一直阻塞。当阻塞时，如果有client对key1...keyN中的任意key进行push操作，则第一 在这个key上被阻塞的client会立即返回。如果超时发生，则返回nil。有点像unix的select或者poll |
| brpop | 同blpop，一个是从头部删除一个是从尾部删除 |
| rpoplpush srckey destkey | 从srckey对应list的尾部移除元素并添加到destkey对应list的头部,最后返回被移除的元素值，整个操作是原子的.如果srckey是空或者不存在返回nil |

**3.4 set集合类型：**

string类型的无序集合。集合中数据不重复。

set元素最大可包含(2^32 - 1)个元素。

通过hash table实现，所以添加，删除，查找的复杂度都是O(1)。

hash table会随添加或删除自动调整大小。需注意的是调整hash table大小时候需要同步（获取写锁）会阻塞其他读写操作。

除了添加删除，还包含集合的取并集，交集，差集。

通过这些操作可很容易的实现sns中的好友推荐和blog的tag 功能。

* set相关命令：

|  |  |
| --- | --- |
| sadd key member | 添加一个string元素到,key对应的set集合中，成功返回1,如果元素以及在集合中返回0,key对应的set不存在返回错误 |
| srem key member | 从key对应set中移除给定元素，成功返回1，如member在集合中不存在或key不存在返回0，如key对应的不是set类型值返回错误 |
| spop key | 删除并返回key对应set中随机的一个元素,如果set是空或者key不存在返回nil |
| srandmember key | 同spop，随机取set中的一个元素，但是不删除元素 |
| smove srckey dstkey member | 从srckey对应set中移除member并添加到dstkey对应set中，整个操作是原子的。成功返回1,如果member在srckey中不存在返回0，如果key不是set类型返回错误 |
| scard key | 返回set的元素个数，如set是空或key不存在返回0 |
| sismember key member | 判断member是否在set中，存在返回1，0表不存在或key不存在 |
| sinter key1 key2...keyN | 返回所有给定key的交集 |
| sinterstore dstkey key1...keyN | 同sinter，但会同时将交集存到dstkey下 |
| sunion key1 key2...keyN | 返回所有给定key的并集 |
| sunionstore dstkey key1...keyN | 同sunion，并同时保存并集到dstkey下 |
| sdiff key1 key2...keyN | 返回所有给定key的差集 |
| sdiffstore dstkey key1...keyN | 同sdiff，并同时保存差集到dstkey下 |
| smembers key | 返回key对应set的所有元素，结果无序 |
| sadd key member | 添加一个string元素到,key对应的set集合中，成功返回1,如元素已经在集合中返回0,key对应的set不存在返回错误 |

**3.5 有序集合 sorted set：**

有序集合的成员是唯一的,但分数(score)却可重复。

和set一样也是string类型元素的集合，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score。实现是skip list（跳表）和hash table的混合体

当元素被添加到集合中时，一个元素到score的映射被添加到hash table中，所以给定一个元素获取score的开销是O(1)，另一个score到元素的映射被添加到skip list并按照score排序，所以就可有序的获取集合中的元素。

添加，删除操作开销都是O(log(N))和skip list的开销一致，redis的skip list实现用的是双向链表，这样就可逆序从尾部取元素。

最常的使用方式应该是作为索引来使用。

可把要排序的字段作为score存储，对象的id当元素存储。

* sorted set相关命令：

|  |  |
| --- | --- |
| zadd key score member | 添加元素到集合，元素在集合中存在则更新对应score |
| zrem key member | 删除指定元素，1表示成功，如果元素不存在返回0 |
| zincrby key incr member | 增加对应member的score值，然后移动元素并保持skip list保持有序。返回更新后的score值 |
| zrank key member | 返回指定元素在集合中的排名（下标）,集合中元素是按score从小到大排序的 |
| zrevrank key member | 同上,但是集合中元素是按score从大到小排序 |
| zrange key start end | 类似lrange操作从集合中去指定区间的元素。返回的是有序结果 |
| zrevrange key start end | 同上，返回结果是按score逆序的 |
| zrangebyscore key min max | 返回集合中score在给定区间的元素 |
| zcount key min max | 返回集合中score在给定区间的数量 |
| zcard key | 返回集合中元素个数 |
| zscore key element | 返回给定元素对应的score |
| zremrangebyrank key min max | 删除集合中排名在给定区间的元素 |
| zremrangebyscore key min max | 删除集合中scor |
| zadd key score member | 添加元素到集合，元素在集合中存在则更新对应score |
| zrem key member | 删除指定元素，1表示成功，如果元素不存在返回0 |
| zincrby key incr member | 增加对应member的score值，然后移动元素并保持skip list保持有序。返回更新后的score值 |
| zrank key member | 返回指定元素在集合中的排名（下标）,集合中元素是按score从小到大排序的 |
| zrevrank key member | 同上,但是集合中元素是按score从大到小排序 |

1. **Redis键值设计**

<http://blog.csdn.net/freebird_lb/article/details/7733988>

1. **Redis数据存储优化机制**

**5.1 zipmap优化hash:**

将对象存储在hash类型中会占用更少内存，且可更方便的存取整个对象。

省内存的原因是新建一个hash对象时开始是用zipmap来存储的。这个zipmap其实并不是hash table，但是zipmap相比正常的hash实现可节省不少hash本身需要的一些元数据存储开销。

尽管zipmap的添加，删除，查找都是O(n)，但由于一般对象的field数量都不太多。所以使用zipmap也是很快的,就是说添加删除平均还是O(1)。

如field或value的大小超出一定限制后，Redis会在内部自动将zipmap替换成正常的hash实现。这个限制可在配置文件中指定（默认配置在redis根目录下的redis.conf中）：

|  |
| --- |
| hash-max-zipmap-entries 512 #配置字段最多512个  hash-max-zipmap-value 64 #配置value最大为64字节 |

**5.2 ziplist优化list:**

如果redisObject的type成员值是REDIS\_LIST类型的，则当该list的元素个数小于配置值list-max-ziplist-entries且元素值字符串的长度小于配置值list-max-ziplist-value, 则可编码成 REDIS\_ENCODING\_ZIPLIST 类型存储，否则采用 Dict 来存储(Dict实际是Hash Table的一种实现)，list采用ziplist数据结构存储数据，这样做一方面为了节省内存，另一方面这种结构式顺序存储的结构，能够更好利用cpu local和预取策略。

配置如下所示：

|  |
| --- |
| list-max-ziplist-entries 512 #配置元素个数最多512个  list-max-ziplist-value 64 #配置value最大为64字节 |

**5.3 intset优化set:**

当set集合中元素为整数且元素个数小于配置set-max-intset-entries值时，使用intset数据结构存储，否则转化为Dict结构，Dict实际是Hash Table的一种实现，key为元素值，value为NULL，这样即可在O(1)时间内判断集合中是否包含某个元素。

intset中有三种类型数组：int16\_t类型、int32\_t 类型、 int64\_t 类型。

怎么选择数组类型，是根据其保存的值的取值范围来决定的，初始化时是 int16\_t，根据 set 中的最大值在[INT16\_MIN, INT16\_MAX] , [INT32\_MIN, INT32\_MAX], [INT64\_MIN, INT64\_MAX]的取值范围来动态确定整个数组的类型。

例如set一开始是 int16\_t 类型，当一个取值范围在 [INT32\_MIN, INT32\_MAX]的值加入到 set 时，则将保存 set 的数组升级成 int32\_t 的数组。

intset元素限制的配置如下所示：

|  |
| --- |
| set-max-intset-entries 512 #配置元素个数最多512个 |

**5.4 ziplist优化sorted set:**

sorted set也有节约内存的方式，当sorted set元素个数及大小小于一定限制,是用ziplist存储。

配置如下：

|  |
| --- |
| zset-max-ziplist-entries 128 #配置元素个数最多512个  zset-max-ziplist-value 64 #配置value最大为64字节 |

5.小结:

Redis提供了很多关于优化内存的方法，上面这些配置的值都是默认配置，实际要根据我们具体的需求场景来调节，并要做大量的测试，以达到最优的效果。同时必须对Redis这些数据结构有很好的理解。

1. **Redis排序**

Redis支持对list，set和sorted set元素的排序。

排序命令：

|  |
| --- |
| SORT key [BY pattern] [LIMIT start count] [GET pattern] [ASC|DESC] [ALPHA] [STORE dstkey] |

6.1 .SORT key:

对集合自身元素排序并返回排序结果。

例子：

|  |
| --- |
| redis 127.0.0.1:6379> select 2  OK  redis 127.0.0.1:6379[2]> lpush m1 12  (integer) 1  redis 127.0.0.1:6379[2]> lpush m1 11  (integer) 2  redis 127.0.0.1:6379[2]> lpush m1 23  (integer) 3  redis 127.0.0.1:6379[2]> lpush m1 13  (integer) 4  redis 127.0.0.1:6379[2]> sort m1  1) "11"  2) "12"  3) "13"  4) "23" |

6.2 .[ASC|DESC] [ALPHA]:

sort默认排序方式是从小到大排的，当然也可按照逆序或按字符顺序排。

逆序可加上desc选项，想按字母顺序排可加alpha选项，当然alpha可和desc一起用。

按字母顺序排的例子:

|  |
| --- |
| redis 127.0.0.1:6379> lpush mylist baidu  (integer) 1  redis 127.0.0.1:6379> lpush mylist hello  (integer) 2  redis 127.0.0.1:6379> lpush mylist xhan  (integer) 3  redis 127.0.0.1:6379> lpush mylist soso  (integer) 4  redis 127.0.0.1:6379> sort mylist  1. "soso"  2. "xhan"  3. "hello"  4. "baidu"  redis 127.0.0.1:6379> sort mylist alpha  1. "baidu"  2. "hello"  3. "soso"  4. "xhan"  redis 127.0.0.1:6379> sort mylist desc alpha  1. "xhan"  2. "soso"  3. "hello"  4. "baidu" |

6.3 .[BY pattern] [GET pattern] :

除了可按集合元素自身值排序外，还可将集合元素内容按给定pattern组合成新的key，并按照新key中对应的内容进行排序。

下面的例子接着使用第一个例子中的ml集合做演示：

|  |
| --- |
| redis 127.0.0.1:6379> set name11 nihao  OK  redis 127.0.0.1:6379> set name12 wo  OK  redis 127.0.0.1:6379> set name13 shi  OK  redis 127.0.0.1:6379> set name23 lala  OK  redis 127.0.0.1:6379[2]> sort m1 by name\* alpha  1) "23"  2) "11"  3) "13"  4) "12" |

注:如果不加alpha，会报错，提示：(error) err one or more scores can't be converted into double。

\*代表了ml中的元素值，所以这个排序是按照name12 name13 name23name23这四个key对应值排序的，返回的还是排序后ml集合中元素。

上面的例子都是返回的ml集合中的元素。Get操作可将list中的值与pattern组成新的key，用该新key去获取出来的list长度个数的值进行排序（排序是排列的get出的值，而不是list中的值），如果用新组成的key不存在，则返回(nil)。

|  |
| --- |
| redis 127.0.0.1:6379> sort ml by name\* get name\* alpha  1. "lala"  2. "nihao"  3. "shi"  4. "wo" |

这次返回的不在是ml中的元素，而是name12 name13 name23 name23对应的值。

排序是按照name12 name13 name23name23值并根据字母顺序排的。

另外get选项可以有多个。看例子（#特殊符号引用的是原始集合也就是ml）

|  |
| --- |
| redis 127.0.0.1:6379> sort ml by name\* get name\* get # alpha  1. "lala"  2. "23"  3. "nihao"  4. "11"  5. "shi"  6. "13"  7. "wo"  8. "12" |

最后在还有一个引用hash类型字段的特殊字符->，下面是例子

|  |
| --- |
| redis 127.0.0.1:6379> hset user1 name hanjie  (integer) 1  redis 127.0.0.1:6379> hset user11 name hanjie  (integer) 1  redis 127.0.0.1:6379> hset user12 name 86  (integer) 1  redis 127.0.0.1:6379> hset user13 name lxl  (integer) 1  redis 127.0.0.1:6379> sort ml get user\*->name  "hanjie"  "86"  "lxl"  (nil) |

注意当对应的user23不存在时候返回的是nil

6.4 .[LIMIT start count]:

上面例子返回结果都是全部。limit选项可限定返回结果的数量。

|  |
| --- |
| redis 127.0.0.1:6379> sort ml get name\* limit 1 2  1. "wo"  2. "shi" |

start下标从0开始，limit选项意思是从第二个元素开始获取2个。

6.5 .[STORE dstkey]:

如对集合常按照固定模式排序，那么把排序结果缓存起来会减少不少cpu开销。

使用store选项可将排序内容保存到指定key中。

保存的类型是list，下面这个例子我们将排序结果保存到了cl中：

|  |
| --- |
| redis 127.0.0.1:6379> sort ml get name\* limit 1 2 store cl  (integer) 2  redis 127.0.0.1:6379> type cl  list  redis 127.0.0.1:6379> lrange cl 0 -1  1. "wo"  2. "shi" |

6.6.关于排序的一些问题:

* 如有多个redis server，不同的key可能存在于不同的server上。

比如name12 name13 name23 name23，很可能分别在四个不同server上存贮。这种情况会对排序性能造成很大的影响。

redis作者在他的blog上提到了这个问题的解决办法，就是将需要排序的key都放到同一个server上。由于具体决定哪个key存在哪个服务器上一般都是在client端hash办法来做的。可通过只对key部分进行hash。

举个例子假如client如果发现key中包含{}。那么只对key中{}包含的内容进行hash。我们将四个name相关的key，都这样命名{name}12{name}13 {name}23 {name}23，于是client 程序就会把他们都放到同一server上，Redis的Java客户端Jedis已经实现了该功能。

* 还有个问题也比较严重。如要sort的集合非常大的话排序就会消耗很长时间。由于redis

单线程的，所以长时间的排序操作会阻塞其他client的请求。

解决办法是通过主从复制机制将数据复制到多个slave上。然后只在slave上做排序操作。并尽可能的对排序结果缓存。

另外一个方案是就是采用sortedset对需要按某个顺序访问的集合建立索引。

1. **Redis事务**

Redis对事务的支持目前还比较简单。

redis只能保证一个client发起的事务中命令可连续执行，而中间不会插入其他client的命令。

由于redis是单线程来处理所有client的请求的所以做到这点是很容易的。

一般情况下redis在接受到一个client发来的命令后会立即处理并返回处理结果，但当一个client在一个连接中发出multi命令，这个连接会进入事务上下文，该连接后续的命令并不是立即执行，而是先放到一 个队列中。

当从此连接受到exec命令后，redis会顺序执行队列中的所有命令。并将所有命令的运行结果打包到一起返回给client.然后此连接就 结束事务上下文。

例子：（有问题，未同步）

|  |
| --- |
| redis 127.0.0.1:6379> multi  OK  redis 127.0.0.1:6379> incr a  QUEUED  redis 127.0.0.1:6379> incr b  QUEUED  redis 127.0.0.1:6379> exec  1. (integer) 1  2. (integer) 1 |

从这个例子可看到incr a ,incr b命令发出后并没执行而是被放到了队列中。

调用exec后俩个命令被连续的执行，最后返回的是两条命令执行后的结果。

可调用discard命令来取消一个事务。接着上面例子：

|  |
| --- |
| redis 127.0.0.1:6379> multi  OK  redis 127.0.0.1:6379> incr a  QUEUED  redis 127.0.0.1:6379> incr b  QUEUED  redis 127.0.0.1:6379> discard  OK  redis 127.0.0.1:6379> exec  (error) ERR EXEC without MULTI  redis 127.0.0.1:6379> get a  "1"  redis 127.0.0.1:6379> get b  "1" |

可发现这次incr a incr b都没被执行。

discard命令其实就是清空事务的命令队列并退出事务上下文。

虽说redis事务本质上也相当于序列化隔离级别的了。但由于事务上下文的命令只排队并不立即执行，所以事务中的写操作不能依赖事务中的读操作结果。很可能有两个client同时做这个操作，也就是说如我们要实现一个类似incr命令的功能，先get a取出我的值，再将a++，然后set a，期望是加两次a从原来的1变成3. 但是很有可能两个client的get a，取到都是1，造成最终加两次结果却是2。主要问题我们没有对共享资源a的访问进行任何的同步。也就是说redis没提供任何的加锁机制来同步对a的访问。

还好redis 2.1后添加了watch命令，可用来实现乐观锁。

看个正确实现incr命令的例子，只是在前面加了watch a ：

|  |
| --- |
| redis 127.0.0.1:6379> watch a  OK  redis 127.0.0.1:6379> get a  "1"  redis 127.0.0.1:6379> multi  OK  redis 127.0.0.1:6379> set a 2  QUEUED  redis 127.0.0.1:6379> exec  1. OK  redis 127.0.0.1:6379> get a  "2" |

watch 命令会监视给定的key，当exec时候如果监视的key从调用watch后发生过变化，则整个事务会失败。

也可调用watch多次监视多个key.这样就可对指定的key加乐观锁了。

注意watch的key是对整个连接有效的，事务也一样。

如果连接断开，监视和事务都会被自动清除。

当然了 exec、discard、unwatch命令都会清除连接中的所有监视.

redis的事务实现如此简单，当然会存在一些问题。

* 第一个问题是redis只能保证事务的每个命令连续执行，但是如果事务中的一个命令失败了，并不回滚其他命令，比如使用的命令类型不匹配。

|  |
| --- |
| redis 127.0.0.1:6379> set a 5  OK  redis 127.0.0.1:6379> lpush b 5  (integer) 1  redis 127.0.0.1:6379> set c 5  OK  redis 127.0.0.1:6379> multi  OK  redis 127.0.0.1:6379> incr a  QUEUED  redis 127.0.0.1:6379> incr b  QUEUED  redis 127.0.0.1:6379> incr c  QUEUED  redis 127.0.0.1:6379> exec  1. (integer) 6  2. (error) ERR Operation against a key holding the wrong kind of value  3. (integer) 6 |

可看到虽然incr b失败了(b为字符串)，但其他两个命令还是执行了。

* 还有一个问题是当事务的执行过程中，如redis意外挂了。只有部分命令执行了，

后面的也就被丢弃了。

如我们使用的append-only file方式持久化，redis会用单个write操作写入整个事务内容。即使是这种方式还是有可能只部分写入了事务到磁盘。发生部分写入事务的情况下，redis重启时会检测到这种情况，然后失败退出。可以使用redis-check-aof工具进行修复，修复会删除部分写入的事务内容。修复完后就能重启了。

1. **Redis管道（pipeline）**

cs模式的tcp server，使用和http类似的请求响应协议。

一个client可以通过一个socket连接发起多个请求命令。每个请求命令发出后client通常会阻塞并等待redis服务处理，redis处理完后请求命令后会将结果通过响应报文返回给client。基本的通信过程如下：

|  |
| --- |
| Client: INCR X  Server: 1  Client: INCR X  Server: 2  Client: INCR X  Server: 3  Client: INCR X  Server: 4 |

四个命令需8个tcp报文才能完成。

由于通信会有网络延迟,假如从client和server之间的包传输时间需0.125秒。那么上面的四个命令8个报文至少需1秒才能完成。这样即使redis每秒能处理100个命令，而我们的client也只能一秒钟发出四个命令。这显然没有充分利用 redis的处理能力。

除了可利用mget,mset 之类的单条命令处理多个key的命令外，还可利用pipeline的方式从client打包多条命令一起发出，不需等待单条命令的响应返回，而redis服务端会处理完多条命令后会将多条命令的处理结果打包到一起返回给客户端。通信过程如下：

|  |
| --- |
| Client: INCR X  Client: INCR X  Client: INCR X  Client: INCR X  Server: 1  Server: 2  Server: 3  Server: 4 |

假设不会因tcp报文过长而被拆分。可能两个tcp报文就能完成四条命令，client可将四个incr命令放到一个tcp报文一起发送，server则可将四条命令的处理结果放到一个tcp报文返回。

通过pipeline方式当有大批量的操作时。可节省很多原来浪费在网络延迟的时间。

需注意到用 pipeline方式打包命令发送，redis必须在处理完所有命令前先缓存起所有命令的处理结果。打包的命令越多，缓存消耗内存也越多。所以并是不是打包的命令越多越好。具体多少合适需要根据具体情况测试。下面是个jedis客户端使用pipeline的测试：

|  |
| --- |
| package com.jd.redis.client;  import redis.clients.jedis.Jedis;  import redis.clients.jedis.Pipeline;  publicclass PipelineTest {  /\*\*  \* @param args  \*/  publicstaticvoid main(String[] args) {  int count = 1000;  long start = System.currentTimeMillis();  withoutPipeline(count);  long end = System.currentTimeMillis();  System.out.println("withoutPipeline: " + (end-start));  start = System.currentTimeMillis();  usePipeline(count);  end = System.currentTimeMillis();  System.out.println("usePipeline: " + (end-start));  }  privatestaticvoid withoutPipeline(int count){  Jedis jr = null;  try {  jr = new Jedis("10.10.224.44", 6379);  for(int i =0; i<count; i++){  jr.incr("testKey1");  }  } catch (Exception e) {  e.printStackTrace();  }  finally{  if(jr!=null){  jr.disconnect();  }  }  }  privatestaticvoid usePipeline(int count){  Jedis jr = null;  try {  jr = new Jedis("10.10.224.44", 6379);  Pipeline pl = jr.pipelined();  for(int i =0; i<count; i++){  pl.incr("testKey2");  }  pl.sync();  } catch (Exception e) {  e.printStackTrace();  }  finally{  if(jr!=null){  jr.disconnect();  }  }  }  } |

输出：

|  |
| --- |
| withoutPipeline: 11341  usePipeline: 344 |

测试结果还是很明显有较大的差距，所以多次操作用pipeline还是有明显的优势。

1. **Redis发布/订阅**

发布订阅(pub/sub)是一种消息通信模式，主要目的是解耦消息发布者和消息订阅者之间的耦合，这点和设计模式中的观察者模式比较相似。

pub /sub不仅解决发布者和订阅者直接代码级别耦合也解决两者在物理部署上的耦合。

Redis作为一个pub/sub server，在订阅者和发布者之间起到了消息路由的功能。

订阅者可通过subscribe和psubscribe命令向redis server订阅自己感兴趣的消息类型，redis将消息类型称为通道(channel)。

当发布者通过publish命令向redis server发送特定类型的消息时。

订阅该消息类型的全部client都会收到此消息。

这里消息的传递是多对多的。

一个client可以订阅多个 channel,也可以向多个channel发送消息。

这里使用两个不同的client一个是redis自带的redis-cli另一个是用Java版客户端jedis写的。java代码如下：

|  |
| --- |
| package com.jd.redis.client;  import redis.clients.jedis.Jedis;  import redis.clients.jedis.JedisPubSub;  publicclass TestPubSubextends JedisPubSub{  @Override  publicvoid onMessage(String channel, String message) {  System.out.println("onMessage: channel["+channel+"], message["+message+"]");  }  @Override  publicvoid onPMessage(String pattern, String channel, String message) {  System.out.println("onPMessage: channel["+channel+"], message["+message+"]");  }  @Override  publicvoid onSubscribe(String channel,int subscribedChannels) {  System.out.println("onSubscribe: channel["+channel+"],"+  "subscribedChannels["+subscribedChannels+"]");  }  @Override  publicvoid onUnsubscribe(String channel,int subscribedChannels) {  System.out.println("onUnsubscribe: channel["+channel+"], "+  "subscribedChannels["+subscribedChannels+"]");  }  @Override  publicvoid onPUnsubscribe(String pattern,int subscribedChannels) {  System.out.println("onPUnsubscribe: pattern["+pattern+"],"+  "subscribedChannels["+subscribedChannels+"]");  }  @Override  publicvoid onPSubscribe(String pattern,int subscribedChannels) {  System.out.println("onPSubscribe: pattern["+pattern+"], "+  "subscribedChannels["+subscribedChannels+"]");  }  publicstaticvoid main(String[] args) {  Jedis jr = null;  try {  jr = new Jedis("192.168.157.128", 6379, 0);//redis服务地址和端口号  TestPubSub sp = new TestPubSub();  sp.proceed(jr.getClient(),"news.share", "news.blog");  //sp.proceedWithPatterns(jr.getClient(), "news.\*");  } catch (Exception e) {  e.printStackTrace();  }finally{  if(jr!=null){  jr.disconnect();  }  }  }  } |

代码就是用TestPubSub对象来订阅，对象中的那此onXXX方法监听到相应事件

1 首先运行此java程序；

|  |
| --- |
| onSubscribe: channel[news.share], subscribedChannels[1]  onSubscribe: channel[news.blog], subscribedChannels[2] |

2 启动redis-cli

|  |
| --- |
| redis 127.0.0.1:6379> psubscribe news.\*  Reading messages... (press Ctrl-C to quit)  1) "psubscribe"  2) "news.\*"  3) (integer) 1 |

3 再启动一个redis-cli用来发布两条消息：

|  |
| --- |
| redis 127.0.0.1:6379> publish news.share "share a link http://www.google.com"  (integer) 2  redis 127.0.0.1:6379> publish news.blog "I post a blog"  (integer) 2 |

4.查看两个订阅client的输出

此时Java client打印如下内容：

|  |
| --- |
| onMessage: channel[news.share], message[share a link http://www.google.com]  onMessage: channel[news.blog], message[I post a blog] |

另一个redis-cli输出如下：

|  |
| --- |
| 1) "pmessage"  2) "news.\*"  3) "news.share"  4) "share a link http://www.google.com"  1) "pmessage"  2) "news.\*"  3) "news.blog"  4) "I post a blog" |

redis client使用psubscribe订阅了一个使用通配符的通道(\*表示任意字符串)，此订阅会收到所有与news.\*匹配的通道消息。redis-cli打印到控制台的订阅成功消息表示使用psubscribe命令订阅news.\*成功后，连接订阅通道总数为1。

当我们在一个client使用publish向news.share和news.blog通道发出两个消息后。redis返回的(integer) 2表示有两个连接收到了此消息。

需注意当一个连接通过subscribe或psubscribe订阅通道后就进入订阅模式。

在这种模式除了再订阅额外的通道或者用unsubscribe或punsubscribe命令退出订阅模式，就不能再发送其他命令。

另外使用 psubscribe命令订阅多个通配符通道，如果一个消息匹配上了多个通道模式的话，会多次收到同一个消息。

redis的pub/sub还是有点太单薄（实现才用150行代码）。在安全，认证，可靠性这方便都没有太多支持

1. **Redis持久化**

Redis是一个支持持久化的内存数据库，就是说redis需要经常将内存中的数据同步到磁盘来保证持久化。

四种持久化方式：

* Snapshotting（快照） 默认方式
* Append-only file（缩写aof）的方式；
* 虚拟内存方式；
* diskstore方式。

**10.1 Snapshotting**

默认。将内存中数据以快照方式写入到二进制文件中, 默认的文件名为dump.rdb。

可通过配置设置自动做快照持久化的方式。

可配置redis在n秒内如果超过m个key被修改就自动做快照，下面是默认的快照保存配置：

(Redis.conf文件中)

|  |
| --- |
| save 900 1 #900秒内如果超过1个key被修改，则发起快照保存  save 300 10 #300秒内容如超过10个key被修改，则发起快照保存  save 60 10000 |

快照保存过程：

1. redis调用fork,现在有了子进程和父进程。

2. 父进程继续处理client请求，子进程负责将内存内容写入到临时文件。由于os的写时复制机制（copy on write)父子进程会共享相同的物理页面，当父进程处理写请求时os会为父进程要修改的页面创建副本，而不是写共享的页面。所以子进程的地址空间内的数据是fork时刻整个数据库的一个快照。

3. 当子进程将快照写入临时文件完毕后，用临时文件替换原来的快照文件，然后子进程退出（fork一个进程入内在也被复制了，即内存会是原来的两倍）。

client 也可使用save或者bgsave命令通知redis做一次快照持久化。

save操作是在主线程中保存快照的，由于redis是用一个主线程来处理所有 client的请求，这种方式会阻塞所有client请求。所以不推荐使用。

另一点需注意的是，每次快照持久化都是将内存数据完整写入到磁盘一次，并不是增量的只同步脏数据。如果数据量大的话，而且写操作比较多，必然会引起大量的磁盘io操作，可能会严重影响性能。

另外由于快照方式是在一定间隔时间做一次的，所以如果redis意外down掉的话，就会丢失最后一次快照后的所有修改。

如应用要求不能丢失任何修改的话，可采用aof持久化方式。

**10.2 Append-only file**

aof 比快照方式有更好的持久化性，是由于在使用aof持久化方式时，redis会将每一个收到的写命令都通过write函数追加到文件中(默认是appendonly.aof)。

当redis重启时会通过重新执行文件中保存的写命令来在内存中重建整个数据库的内容。

当然由于os会在内核中缓存 write做的修改，所以可能不是立即写到磁盘上。

这样aof方式的持久化也还是有可能会丢失部分修改。

可通过配置文件告诉redis想要通过fsync函数强制os写入到磁盘的时机。

三种方式（默认是：每秒fsync一次）：

|  |
| --- |
| appendonly yes #启用aof持久化方式  # 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 |

aof 方式也同时带来另一个问题。持久化文件会变的越来越大。

例如我们调用incr test命令100次，文件中必须保存全部的100条命令，其实有99条都是多余的。因为要恢复数据库的状态其实文件中保存一条set test 100就够了。

为了压缩aof的持久化文件。redis提供了bgrewriteaof命令。

收到此命令redis将使用与快照类似的方式将内存中的数据以命令的方式保存到临时文件中，最后替换原来的文件。具体过程如下：

1. redis调用fork ，现在有父子两个进程

2. 子进程根据内存中的数据库快照，往临时文件中写入重建数据库状态的命令

3. 父进程继续处理client请求，除了把写命令写入到原来的aof文件中。同时把收到的写命令缓存起来。就能保证如果子进程重写失败的话并不会出问题。

4. 当子进程把快照内容写入以命令方式写到临时文件中后，子进程发信号通知父进程。然后父进程把缓存的写命令也写入到临时文件。

5. 现在父进程可使用临时文件替换老的aof文件，并重命名，后面收到的写命令也开始往新的aof文件中追加。

需注意重写aof文件的操作，并没有读取旧的aof文件，而是将整个内存中的数据库内容用命令的方式重写了一个新的aof文件，这点和快照有点类似。

**10.3 虚拟内存方式（desprecated）**

10.3.1 首先说明：在Redis-2.4后虚拟内存功能已经被deprecated了，原因如下：

|  |
| --- |
| 1）slow restart重启太慢  2）slow saving保存数据太慢  3）slow replication上面两条导致 replication 太慢  4）complex code代码过于复杂 |

redis的虚拟内存与os的虚拟内存不是一码事，但思路和目的相同。

就是暂时把不经常访问的数据从内存交换到磁盘中，从而腾出宝贵的内存空间用于其他需要访问的数据。尤其是对于redis这样的内存数据库，内存总是不够用的。

除了可将数据分割到多个redis server外。

另外的能够提高数据库容量的办法就是使用vm把那些不经常访问的数据交换的磁盘上。如果我们的存储的数据总是有少部分数据被经常访问，大部分数据很少被访问，对于网站来说确实总是只有少量用户经常活跃。当少量数据被经常访问时，使用vm不但能提高单台redis server数据库的容量，而且也不会对性能造成太多影响。

10.3.2 redis没使用os提供的虚拟内存机制，而是自己在用户态实现了自己的虚拟内存机制,作者在自己的blog专门解释了其中原因。

http://antirez.com/post/redis-virtual-memory-story.html

主要的理由有两点：

1. os 的虚拟内存是以4k页面为最小单位进行交换的。

而redis的大多数对象都远小于4k，所以一个os页面上可能有多个redis对象。 另外redis的集合对象类型如list,set可能存在与多个os页面上。

最终可能造成只有10%key被经常访问，但所有os页面都会被os认为是活跃的，这样只有内存真正耗尽时os才会交换页面。

2.相比于os的交换方式。redis可将被交换到磁盘的对象进行压缩,保存到磁盘的对象可以去除指针和对象元数据信息。一般压缩后的对象会比内存中的对象小10倍。这样redis的vm会比os vm能少做很多io操作。

10.3.3下面是vm相关配置：

|  |
| --- |
| slaveof 192.168.1.1 6379 #指定master的ip和端口  vm-enabled yes #开启vm功能  vm-swap-file /tmp/redis.swap #交换出来的value保存的文件路径/tmp/redis.swap  vm-max-memory 1000000 #redis使用的最大内存上限，超过上限后redis开始交换value到  磁盘文件中  vm-page-size 32 #每个页面的大小32个字节  vm-pages 134217728 #最多使用在文件中使用多少页面,交换文件的大小 = vm-page-size \* vm-pages  vm-max-threads 4 #用于执行value对象换入换出的工作线程数量，0表示不使用工作线程（后面介绍) |

redis的vm在设计上为了保证key的查找速度，只会将value交换到swap文件中。

所以如果内存问题是由于太多value很小的key造成的，vm并不能解决。

和os一样redis也是按页面来交换对象的。redis规定同一个页面只能保存一个对象。但是一个对象可以保存在多个页面中。

在redis使用的内存没超过vm-max-memory之前是不会交换任何value的。当超过最大内存限制后，redis会选择较老的对象。如果两个对象一样老会优先交换比较大的对象，精确的公式swappability = age\*log(size\_in\_memory)。对于vm-page-size的设置应该根据自己的应用将页面的大小设置为可以容纳大多数对象的大小。太大了会浪费磁盘空间，太小了会造成交换文件出现碎片。对于交换文件中的每个页面，redis会在内存中对应一个1bit值来记录页面的空闲状态。所以像上面配置中页面数量(vm-pages 134217728 )会占用16M内存用来记录页面空闲状态。

vm-max-threads表示用做交换任务的线程数量。如大于0推荐设为服务器的cpu core数量。如果是0则交换过程在主线程进行。

10.3.5 参数配置讨论完后，来简单介绍下vm是如何工作的：

* 当vm-max-threads设为0时(Blocking VM)
* 换出：

主线程定期检查发现内存超出最大上限后，会直接以阻塞的方式,将选中的对象保存到swap文件中，并释放对象占用的内存,此过程会一直重复直到下面条件满足

1.内存使用降到最大限制以下

2.swap文件满了

3.几乎全部的对象都被交换到磁盘了

* 换入：

当有client请求value被换出的key时。主线程会以阻塞的方式从文件中加载对应的value对象，加载时此时会阻塞所有client。然后处理client的请求

* 当vm-max-threads大于0(Threaded VM)
* 换出：

当主线程检测到使用内存超过最大上限，会将选中的要交换的对象信息放到一个队列中交由工作线程后台处理，主线程会继续处理client请求。

* 换入：

如果有client请求的key被换出了，主线程先阻塞发出命令的client, 然后将加载对象的信息放到一个队列中，让工作线程去加载。加载完毕后工作线程通知主线程。主线程再执行client的命令。这种方式只阻塞请求value被换出key的client

总的来说blocking vm的方式总的性能会好一些，因为不需线程同步，创建线程和恢复被阻塞的client等开销。

但也相应的牺牲了响应性。threaded vm的方式主线程不会阻塞在磁盘io上，所以响应性更好。如果应用不太经常发生换入换出，且也不太在意有点延迟的话则推荐使用blocking vm的方式。

关于redis vm的更详细介绍可以参考下面链接：

http://antirez.com/post/redis-virtual-memory-story.html

http://redis.io/topics/internals-vm

**10.4 diskstore方式**

是作者放弃了虚拟内存方式后选择的一种新的实现方式，也就是传统的B-tree的方式。

具体细节是：

1) 读操作，使用read through以及LRU方式。内存中不存在的数据从磁盘拉取并放入内存，内存中放不下的数据采用LRU淘汰。

2) 写操作，采用另外spawn线程单独处理，写线程通常是异步的，当然也可把cache-flush-delay配置设成0，Redis尽量保证即时写入。

但在很多场合延迟写会有更好的性能，比如一些计数器用Redis存储，在短时间如果某个计数反复被修改，Redis只需要将最终的结果写入磁盘。这种做法作者叫per key persistence。由于写入会按key合并，因此和snapshot还是有差异，disk store并不能保证时间一致性。

由于写操作是单线程，即使cache-flush-delay设成0，多个client同时写则需要排队等待，如果队列容量超过cache-max-memory Redis设计会进入等待状态，造成调用方卡住。

Google Group上有热心网友迅速完成了压力测试，当内存用完之后，set每秒处理速度从25k下降到10k再到后来几乎卡住。 虽然通过增加cache-flush-delay可以提高相同key重复写入性能；通过增加cache-max-memory可以应对临时峰值写入。但是diskstore写入瓶颈最终还是在IO。

3) rdb 和新 diskstore 格式关系

rdb是传统Redis内存方式的存储格式，diskstore是另外一种格式，那两者关系如何？

* 通过BGSAVE可以随时将diskstore格式另存为rdb格式，而且rdb格式还用于Redis复制以及不同存储方式之间的中间格式。
* 通过工具可将rdb格式转换成diskstore格式。

当然，diskstore原理很美好，但是目前还处于alpha版本，也只是一个简单demo，diskstore.c加上注释只有300行，实现的方法就是将每个value作为一个独立文件保存，文件名是key的hash值。因此diskstore需要将来有一个更高效稳定的实现才能用于生产环境。但由于有清晰的接口设计，diskstore.c也很容易换成一种B-Tree的实现。很多开发者也在积极探讨使用bdb或者innodb来替换默认diskstore.c的可行性。

* Diskstore的算法。

其实DiskStore类似于Hash算法，首先通过SHA1算法把Key转化成一个40个字符的Hash值，然后把Hash值的前两位作为一级目录，然后把Hash值的三四位作为二级目录，最后把Hash值作为文件名，类似于“/0b/ee/0beec7b5ea3f0fdbc95d0dd47f3c5bc275da8a33”形式。算法如下：

|  |
| --- |
| dsKeyToPath(key):  char path[1024];  char \*hashKey = sha1(key);  path[0] = hashKey[0];  path[1] = hashKey[1];  path[2] = '/';  path[3] = hashKey[2];  path[4] = hashKey[3];  path[5] = '/';  memcpy(path + 6, hashKey, 40);  return path;  存储算法（如key == apple）：  dsSet(key, value, expireTime):  // d0be2dc421be4fcd0172e5afceea3970e2f3d940  char \*hashKey = sha1(key)  // d0/be/d0be2dc421be4fcd0172e5afceea3970e2f3d940  char \*path = dsKeyToPath(hashKey);  FILE \*fp = fopen(path, "w");  rdbSaveKeyValuePair(fp, key, value, expireTime);  fclose(fp)  获取算法：  dsGet(key):  char \*hashKey = sha1(key);  char \*path = dsKeyToPath(hashKey);  FILE \*fp = fopen(path, "r");  robj \*val = rdbLoadObject(fp);  return val; |

不过DiskStore有个缺点，就是有可能发生两个不同的Key生成一个相同的SHA1 Hash值，这样就有可能出现丢失数据的问题。不过几率较少，所以是可以接受。根据作者的意图，未来可能使用B+tree来替换这种高度依赖文件系统的实现方法。

还可以看：http://www.hoterran.info/redis\_persistence

1. [**Redis主从复制**](http://blog.csdn.net/freebird_lb/article/details/7778989)

Redis主从复制配置和使用都非常简单。

通过主从复制可允许多个slave server拥有和master server相同的数据库副本。

11.1 主从复制特点

|  |
| --- |
| 1.master可有多个slave  2.除了多个slave连到相同的master外，slave也可连接其他slave形成图状结构  3.主从复制不会阻塞master。也就是说当一个或多个slave与master进行初次同步数据时，master可以继续处理client发来的请求。相反slave在初次同步数据时则会阻塞不能处理client的请求。  4.主从复制可用来提高系统的可伸缩性,可用多个slave专门用于client的读请求，比如sort操作可以使用slave来处理。也可以用来做简单的数据冗余  5.可在master禁用数据持久化，只需要注释掉master配置文件中的所有save配置，然后只在slave上配置数据持久化。 |

11.2 主从复制过程

当设置好slave服务器后，slave会建立和master的连接，然后发送sync命令。无论是第一次同步建立的连接还是连接断开后的重新连接，master都会启动(fork)一个后台进程，将数据库快照保存到文件中（fork一个进程入内在也被复制了，即内存会是原来的两倍），同时master主进程会开始收集新的写命令并缓存起来。

后台进程完成写文件后，master就发送文件给slave，slave将文件保存到磁盘上，然后加载到内存恢复数据库快照到slave上。

接着master就会把缓存的命令转发给slave。且后续master收到的写命令都会通过开始建立的连接发送给slave。从master到slave的同步数据的命令和从 client发送的命令使用相同的协议格式。

当master和slave的连接断开时slave可自动重新建立连接。

如master同时收到多个 slave发来的同步连接命令，只会使用启动一个进程来写数据库镜像，然后发送给所有slave。

配置slave服务器只需要在配置文件中加入如下配置：

slaveof 192.168.1.1 6379 #指定master的ip和端口

详细信息还可以看：http://www.hoterran.info/redis\_replication

1. [**Redis分布式**](http://blog.csdn.net/freebird_lb/article/details/7778999)

Redis-2.4.15目前没提供集群功能，redis作者在博客中说将在3.0中实现集群机制。

目前Redis实现集群的方法主要是采用一致性哈稀分片（Shard），将不同的key分配到不同的redis server上，达到横向扩展的目的。

下面来介绍一种比较常用的分布式场景：

在读写操作比较均匀且实时性要求较高，可以用下图的分布式模式：

在读操作远远多于写操作时，可以用下图的分布式模式：

对于一致性哈稀分片的算法，Jedis-2.0.0已经提供了，下面是使用示例代码

（以ShardedJedisPool为例）：

|  |
| --- |
| package com.jd.redis.client;  import Java.util.ArrayList;  import java.util.List;  import redis.clients.jedis.JedisPoolConfig;  import redis.clients.jedis.JedisShardInfo;  import redis.clients.jedis.ShardedJedis;  import redis.clients.jedis.ShardedJedisPool;  import redis.clients.util.Hashing;  import redis.clients.util.Sharded;  publicclass RedisShardPoolTest {  static ShardedJedisPoolpool;  static{  JedisPoolConfig config =new JedisPoolConfig();//Jedis池配置  config.setMaxActive(500);//最大活动的对象个数  config.setMaxIdle(1000 \* 60);//对象最大空闲时间  config.setMaxWait(1000 \* 10);//获取对象时最大等待时间  config.setTestOnBorrow(true);  String hostA = "10.10.224.44";  int portA = 6379;  String hostB = "10.10.224.48";  int portB = 6379;  List<JedisShardInfo> jdsInfoList =new ArrayList<JedisShardInfo>(2);  JedisShardInfo infoA = new JedisShardInfo(hostA, portA);  infoA.setPassword("redis.360buy");  JedisShardInfo infoB = new JedisShardInfo(hostB, portB);  infoB.setPassword("redis.360buy");  jdsInfoList.add(infoA);  jdsInfoList.add(infoB);  pool =new ShardedJedisPool(config, jdsInfoList, Hashing.MURMUR\_HASH,  Sharded.DEFAULT\_KEY\_TAG\_PATTERN);  }  /\*\*  \* @param args  \*/  publicstaticvoid main(String[] args) {  for(int i=0; i<100; i++){  String key = generateKey();  //key += "{aaa}";  ShardedJedis jds = null;  try {  jds = pool.getResource();  System.out.println(key+":"+jds.getShard(key).getClient().getHost());  System.out.println(jds.set(key,"1111111111111111111111111111111"));  } catch (Exception e) {  e.printStackTrace();  }  finally{  pool.returnResource(jds);  }  }  }  privatestaticintindex = 1;  publicstatic String generateKey(){  return String.valueOf(Thread.currentThread().getId())+"\_"+(index++);  }  } |

从运行结果中可看到，不同key被分配到不同的Redis-Server上去了。

实际上，上面的集群模式还存在两个问题：

1.扩容问题：

因为使用了一致性哈稀进行分片，不同的key分布到不同的Redis-Server上，当需要扩容时，需要增加机器到分片列表中，会使得同样的key算出来落到跟原来不同的机器上，如果要取某一个值，会出现取不到的情况，对于这种情况，Redis的作者提出了一种名为Pre-Sharding的方式：

Pre-Sharding方法是将每一个台物理机上，运行多个不同端口的Redis实例，假如有三个物理机，每个物理机运行三个Redis实例，那么分片列表中实际有9个Redis实例，当需要扩容时，增加一台物理机，步骤如下：

|  |
| --- |
| A.在新的物理机上运行Redis-Server；  B.该Redis-Server从属于(slaveof)分片列表中的某一Redis-Server（假设叫RedisA）；  C.等主从复制(Replication)完成后，将客户端分片列表中RedisA的IP和端口改为新物理机  上Redis-Server的IP和端口；  D.停止RedisA。 |

相当于将某一Redis-Server转移到一台新机器上。

Prd-Sharding实际上是一种在线扩容的办法，但还是很依赖Redis本身的复制功能的，如果主库快照数据文件过大，这个复制的过程也会很久，同时会给主库带来压力。

所以做这个拆分的过程最好选择为业务访问低峰时段进行。

http://blog.nosqlfan.com/html/3153.html

2.单点故障问题：

还是用到Redis主从复制的功能，两台物理主机上分别都运行有Redis-Server，其中一个Redis-Server是另一个的从库，采用双机热备技术，客户端通过虚拟IP访问主库的物理IP，当主库宕机时，切换到从库的物理IP。

只是事后修复主库时，应该将之前的从库改为主库（使用命令slaveof no one），主库变为其从库（使命令slaveof IP PORT），才能保证修复期间新增数据的一致性。

补充:

|  |
| --- |
| 热备:与目标设备共同运转,当目标设备发生故障或停机，热备设备立即承担起故障设备的工作任务；  冷备:当目标设备发生故障或停机后，冷备设备才开始由停机等待状态进入启动运转状态，并承担起故障设备的工作任务 |

1. **常见命令**
2. INFO

<http://blog.csdn.net/kexiaoling/article/details/51810919>

查看redis使用情况

1. **优缺点**

**4.1 优点：**

1、丰富的数据结构

2、高速读写

3、操作都是原子性

**4.2 缺点：**

1、单线程

2、耗内存