1. **Redis简介**

**1.1 概述**

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) |

**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)。

可以通过push,pop操作从链表的头部或者尾部添加删除元素。

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. **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