Redis开发运维实践指南

Pengcheng Huang

Published
with GitBook

目錄

Redis开发运维实践指南

本手册是我在一家中国大陆的中型商业银行做大数据系统工程师中进行的总结归纳，包含开发和运维的各方面的使用、应用场景和最佳实践，以及各个高可用架构的搭建和测试。

下载和线上阅读最新版请访问：https://www.gitbook.com/book/gnuhpc/redis-all-about/details

我会定期持续更新此指南，以供各位了解认识redis。

其中会有引用的各种文献和图片，我会尽可能的标注引用，如果您发现您的文字或者图片未标注引用，请fork本手册的github并pull request： https://github.com/gnuhpc/All-About-Redis

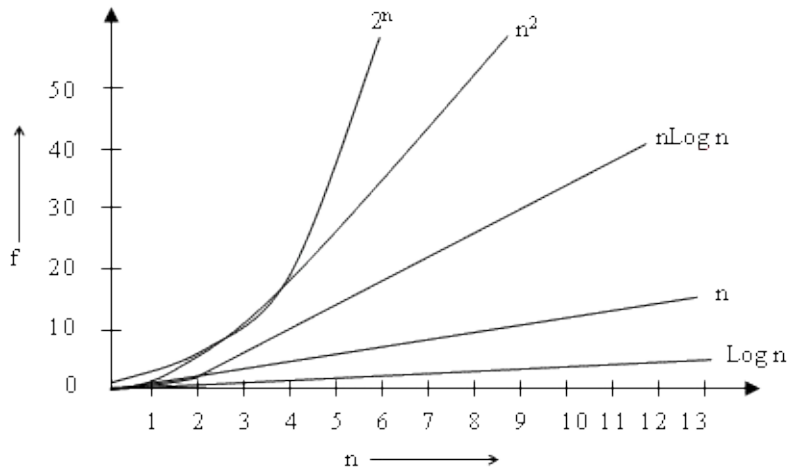
有不正确的地方，请通过gnuhpc@weibo或者huangpengcheng#cmbc.com.cn (:s/#/@)联系我。 — Pengcheng Huang

1. 简述

Redis为一个运行在内存中的数据结构服务器（data structures server）。Redis使用的是单进程（除持久化时），所以在配置时，一个实例只会用到一个CPU。本手册分为三部分，第一部分从开发角度介绍了redis开发中使用API、场景和生产设计规范最佳实践，第二部分从运维角度介绍了如何运维redis，其间的常见操作和最佳实践等,第三部分是从高可用和集群方面介绍了redis相关的集群架构、搭建和测试。

2. 数据操作

熟悉每个数据操作前一定要明白每个操作都是代价，以时间复杂度和对应查询集或者结果集大小为衡量。时间复杂度收敛状况如下：



2.1. key操作

redis 的key操作是涉及范围最广的操作 。

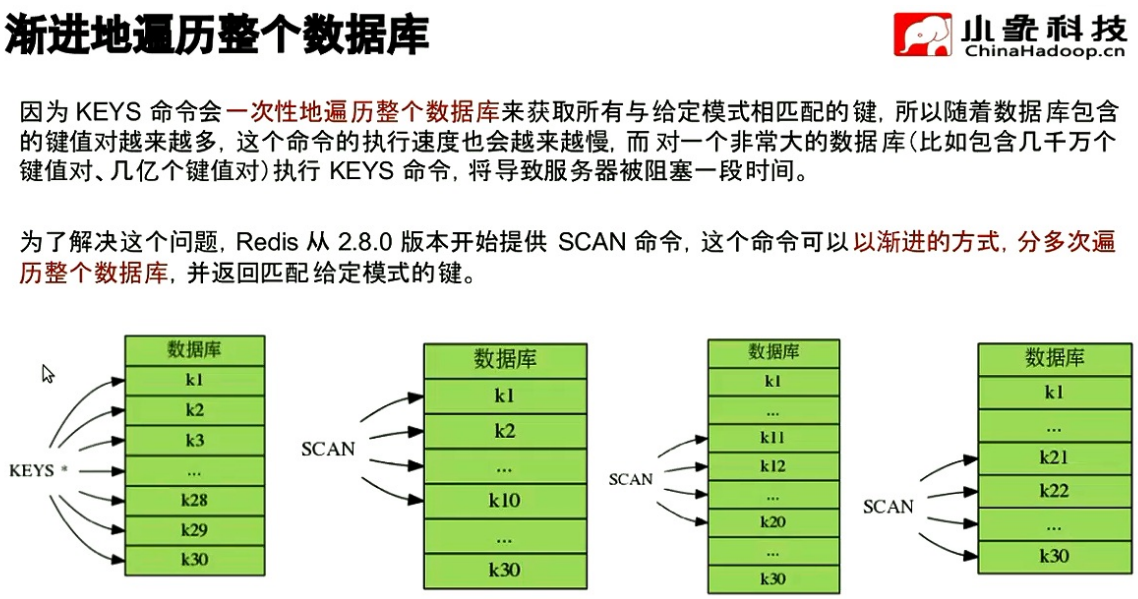
2.1.1 列出key

```
keys  *user*
keys  *
```

有3个通配符 *, ? ,[]

- *: 通配任意多个字符
- ?: 通配单个字符
- []: 通配括号内的某1个字符

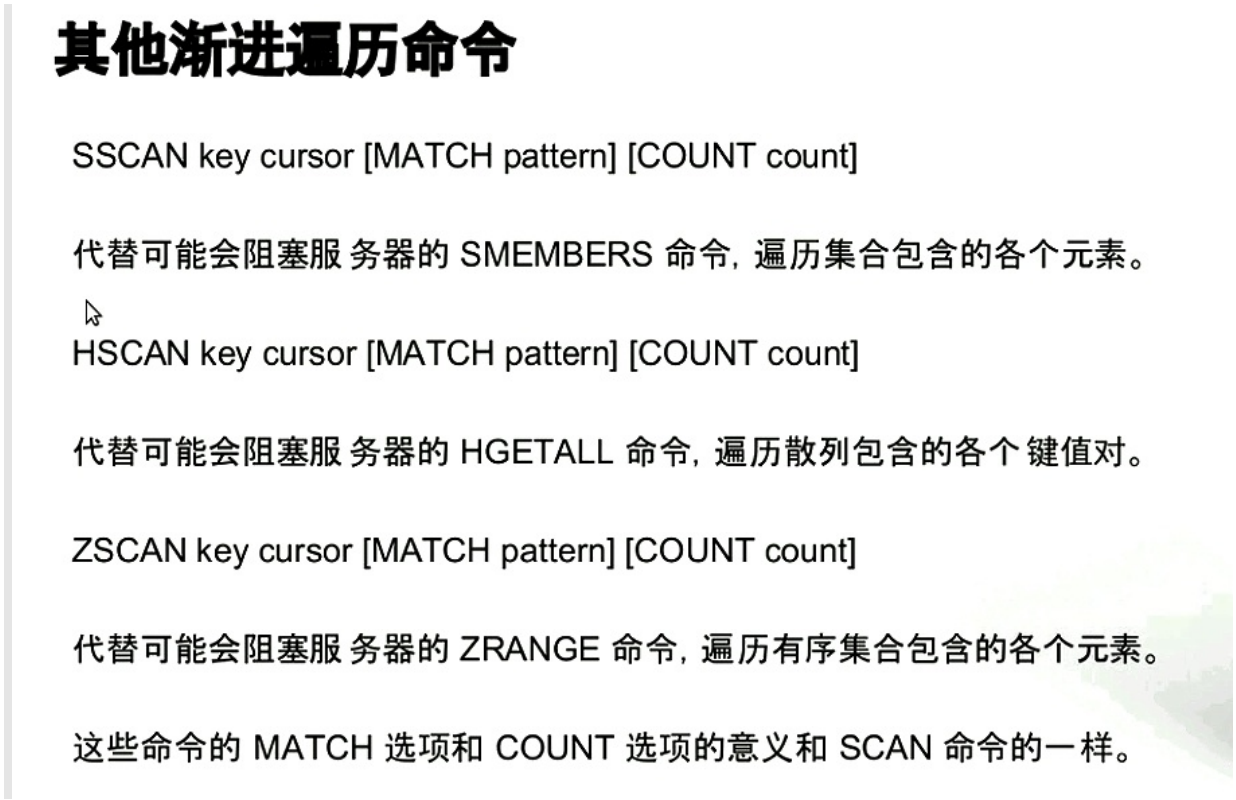
注：生产已经禁止。更安全的做法是采用scan，原理和操作如下：



针对Keys的改进，支持分页查询Key。在迭代过程中，Keys有增删时不会要锁定写操作，数据集完整度不做任何保证，同一条key可能会被返回多次.

	KEYS	SCAN
处理方式	一次性遍历整个数据库并返回所有结果。	每次遍历数据库中的一部分键，并返回一部分结果。
是否会阻塞服务器？	可能会	不会
是否会出现重复值？	不会	可能会
复杂度	O(N)，N 为数据库包含的键值对数量。	每次执行的复杂度为 O(1)，遍历整个数据库的总复杂度为 O(N)，N 为数据库包含的键值对数量。

对于其他危险的命令，新版本也进行了替代：



redis-cli下的扫描:

```
redis-cli --scan --pattern 'chenqun_*'
```

这是用scan命令扫描redis中的key，--pattern选项指定扫描的key的pattern。相比keys *pattern* 模式,不会长时间阻塞redis而导致其他客户端的命令请求一直处于阻塞状态。

> 本页中采用了小象学院的两张片子，版权归 http://www.chinahadoop.cn/ 所有

2.1.2 测试指定key是否存在

```
exits key
```

返回1表示存在，0不存在

2.1.3 删除给定key

```
del key1 key2 ....keyN
```

返回1表示存在，0不存在

2.1.4 返回给定key的value类型

```
type key
```

返回 none 表示不存在key。string字符类型，list 链表类型 set 无序集合类型...

2.1.5 返回从当前数据库中随机选择的一个key

```
randomkey
```

如果当前数据库是空的，返回空串

2.1.6 原子的重命名一个key

```
rename oldkey newkey
```

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

2.1.7 Key的超时设置处理

```
expire key seconds
```

单位是秒。返回1成功，0表示key已经设置过过期时间或者不存在。 如果想消除超时则使用persist key。如果希望采用绝对超时，则使用expireat命令。

```
ttl key
```

返回设置过过期时间的key的剩余过期秒数 -1表示没有设置过过期时间，对于不存在的key,返回-2。

```
pexpire key 毫秒数
```

设置生命周期。

```
pttl  key
```

以毫秒返回生命周期。

> 注意：
>
> 当client主动访问key会先对key进行超时判断，过时的key会立刻删除。
>
> 如果clien永远都不再get那条key呢？它会在Master的后台，每秒10次的执行如下操作：随机选取100个key校验是否过期，如果有25个以上的key过期了，立刻额外随机选取下100个key(不计算在10次之内)。可见，如果过期的key不多，它最多每秒回收200条左右，如果有超过25%的key过期了，它就会做得更多，但只要key不被主动get，它占用的内存什么时候最终被清理掉只有天知道。
>
> 在主从复制环境中，由于上述原因存在已经过期但是没有删除的key，在主snapshot时并不包含这些key，因此在slave环境中我们往往看到dbsize较master是更小的。

最大字符串为512M，但是大字符串非常不建议。

2.2.1 设置key对应的值为string类型的value

```
set key value [ex 秒数] / [px 毫秒数]  [nx] /[xx]
```

返回1表示成功，0失败

> 注: 如果ex,px同时写,以后面的有效期为准

```
setnx key value
```

仅当key不存在时才Set，如果key已经存在，返回0 。nx 是not exist的意思。

> 应用场景：用来选举Master或做分布式锁：所有Client不断尝试使用SetNx master myName抢注Master，成功的那位不断使用Expire刷新它的过期时间。如果Master倒掉了key就会失效，剩下的节点又会发生新一轮抢夺。

```
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，但是不会覆盖已经存在的key

SET 命令还支持可选的 NX 选项和 XX 选项，例如：SET nx-str "this will fail" XX

- 如果给定了 NX 选项，那么命令仅在键 key 不存在的情况下，才进行设置操作；如果键 key 已经存在，那么 SET ... NX 命令不做动作（不会覆盖旧值）。
- 如果给定了 XX 选项，那么命令仅在键 key 已经存在的情况下，才进行设置操作；如果键 key 不存在，那么 SET ... XX 命令不做动作（一定会覆盖旧值）。在给定 NX 选项和 XX 选项的情况下，SET 命令在设置成功时返回 OK ，设置失败时返回 nil 。

2.2.2 获取key对应的string值

```
get key
```

如果key不存在返回nil

```
getset key value
```

原子的设置key的值，并返回key的旧值。如果key不存在返回nil。应用场景：设置新值，返回旧值，配合setnx可实现分布式锁。

分布式锁的思路：注意该思路要保证多台Client服务器的NTP一致。

1. C3发送SETNX lock.foo 想要获得锁，由于C0还持有锁，所以Redis返回给C3一个0
2. C3发送GET lock.foo 以检查锁是否超时了，如果没超时，则等待或重试。
3. 反之，如果已超时，C3通过下面的操作来尝试获得锁：
4. GETSET lock.foo
5. 通过GETSET，C3拿到的时间戳如果仍然是超时的，那就说明，C3如愿以偿拿到锁了。
6. 如果在C3之前，有个叫C4的客户端比C3快一步执行了上面的操作，那么C3拿到的时间戳是个未超时的值，这时，C3没有如期获得锁，需要再次等待或重试。留意一下，尽管C3没拿到锁，但它改写了C4设置的锁的超时值，不过这一点非常微小的误差带来的影响可以忽略不计。

伪代码为：

```
# get lock
lock = 0
while lock != 1:
    timestamp = current Unix time + lock timeout + 1
    lock = SETNX lock.foo timestamp
    if lock == 1 or (now() > (GET lock.foo) and now() > (GETSET lock.foo timestamp)):
        break;
    else:
        sleep(10ms)

# do your job
do_job()

# release
if now() < GET lock.foo:
    DEL lock.foo
```

以上是一个单Server 的分布式锁思路，官网上还介绍了另一个单机使用超时方式进行的思路，和这个基本一致，并且在同一个文档中介绍了一个名为redlock的多Server容错型分布式锁的算法，同时列出了多语言的实现。这个算法的优势在于几个服务器可以有少量的时间差，不要求严格时间一致。

也可以设计一个按小时计算的计数器，可以用GetSet获取计数并重置为0。

```
mget key1 key2 ... keyN
```

一次获取多个key的值，如果对应key不存在，则对应返回nil

```
incr key
```

对key的值做加加操作,并返回新的值。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。范围为64有符号，-9223372036854775808~9223372036854775807。

```
decr key
```

同上，但是做的是减减操作，decr一个不存在key，则设置key为-1

```
incrby key integer
```

同incr，加指定值，key不存在时候会设置key，并认为原来的value是 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。

```
incrbyfloat key floatnumber
```

针对浮点数

浮点数的自增和自减

INCRBYFLOAT key increment

为字符串键 key 储存的值加上浮点数增量 increment，命令返回操作执行之后，键 key 的值。

没有相应的 DECRBYFLOAT，但可以通过给定负值来达到 DECRBYFLOAT 的效果。

复杂度为 O(1)。

```
redis> SET num 10
OK
redis> INCRBYFLOAT num 3.14
"13.14"
redis> INCRBYFLOAT num -2.04    # 通过传递负值来达到做减法的效果
"11.1"
```

哪些可以被操作呢？

值	能否执行数字值命令？	原因
10086	可以	值可以被解释为整数
3.14	可以	值可以被解释为浮点数
+123	可以	值可以被解释为整数
12345678912345678912345678	不可以	值太大，没办法使用 64 位整数来储存
2.0e7	不可以	Redis 不解释以科学记数法表示的浮点数
123ABC	不可以	值包含文字
ABC	不可以	值为文字

这个操作的应用场景：计数器

2.2.4 追加字符串

```
append key value
```

返回新字符串值的长度。

2.2.5 截取字符串

```
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。下标是从0开始的

2.2.6 改写字符串

```
SETRANGE key offset value
```

用value 参数覆写(overwrite)给定key 所储存的字符串值，从偏移量offset 开始。 不存在的key当作空白字符串处理。可以用作append：

```
127.0.0.1:6379> set foo bar
OK
127.0.0.1:6379> setrange foo 3 barla
(integer) 8
127.0.0.1:6379> get foo
"barbarla"
```

> 注意: 如果偏移量>字符长度, 该字符自动补0x00，注意它不会报错

```
127.0.0.1:6379> set foo bar
OK
127.0.0.1:6379> setrange foo 3 a
(integer) 4
127.0.0.1:6379> get foo
"bara"
127.0.0.1:6379> setrange foo 6 a
(integer) 7
127.0.0.1:6379> get foo
"bara\x00\x00a"
127.0.0.1:6379>
```

2.2.7 返回子字符串

GETRANGE key start end

返回key 中字符串值的子字符串，字符串的截取范围由start 和end 两个偏移量决定(包括start 和end 在内)。可以使用负值，字符串右面下标是从-1开始的。

注意返回值处理：
1: start>=length, 则返回空字符串 2: stop>=length,则截取至字符结尾 3: 如果start 所处位置在stop右边, 返回空字符串

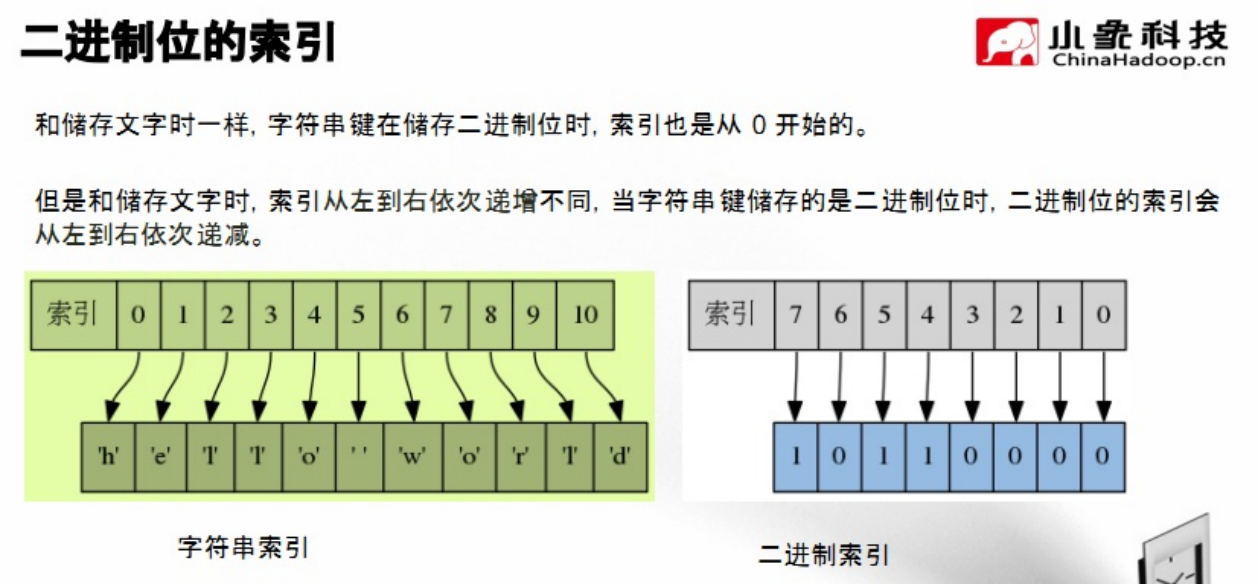
```
$ redis-cli --raw    # 在 redis-cli 中使用中文时，必须打开 --raw 选项，才能正常显示中文

redis> SET msg "世界你好"    # 设置四个中文字符
OK

redis> GET msg    # 储存中文没有问题
世界你好

redis> STRLEN msg    # 这里 STRLEN 显示了“世界你好”的字节长度为 12 字节
12                   # 但我们真正想知道的是 msg 键里面包含多少个字符
```

2.2.9 取指定key的value值的长度

```
strlen
```

2.2.10 位操作

注意：位操作中的位置是反过来的，**offset**过大,则会在中间填充**0**，比如 **SETBIT bit 0 1**，此时**bit**为**10000000**，此时再进行**SETBIT bit 7 1**，此时**bit**为**10000001**。**offset**最大**2^32-1**。

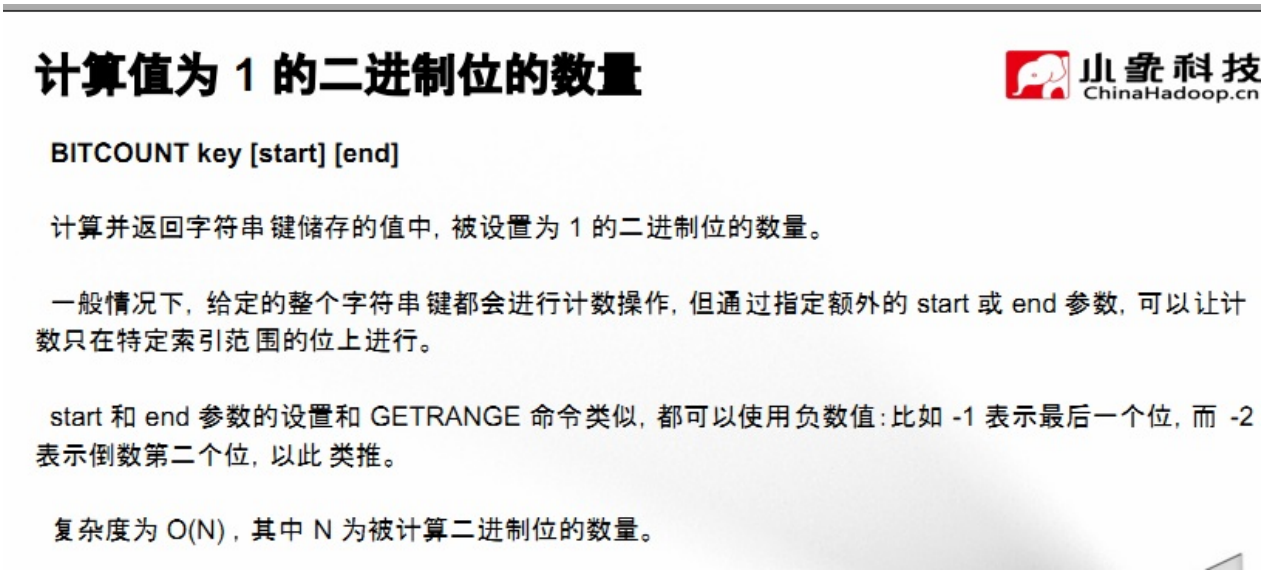



```
GETBIT key offset / SETBIT key offset value
```

设置某个索引的位为0/1

```
bitcount
```

对位进行统计



```
bitop
```

对1个或多个key对应的值进行AND/OR/XOR/NOT操作

注意:

> 1.bitop操作避免阻塞应尽量移到slave上操作. 2.对于NOT操作, key不能多个

2.3.1 添加元素

```
lpush key string
```

在key对应list的头部添加字符串元素，返回1表示成功，0表示key存在且不是list类型。注意：江湖规矩一般从左端Push，右端Pop，即LPush/RPop。

```
lpushx
```

也是将一个或者多个value插入到key列表的表头，但是如果key不存在，那么就什么都不在，返回一个false【rpushx也是同样】

```
rpush key string
```

同上，在尾部添加

```
linsert
```

在key对应list的特定位置之前或之后添加字符串元素 , 例如

```
redis 127.0.0.1:6379linsert mylist3 before "world" "there"
```

2.3.2 查看列表长度

```
llen key
```

返回key对应list的长度，key不存在返回0,如果key对应类型不是list返回错误

2.3.3 查看元素

```
lindex
```

返回名称为key的list中index位置的元素，例如：

```
redis 127.0.0.1:6379> lindex mylist5 0
```

2.3.4 查看一段列表

```
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。特殊的，

```
lrange key 0 -1
```

返回所有数据。

2.3.5 截取list

```
ltrim key start end
```

保留指定区间内元素，成功返回1，key不存在返回错误。O(N)操作。

> 注意：N是被移除的元素的个数，不是列表长度。

2.3.6 删除元素

```
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，count为正，则删除匹配count个元素，如果为负数，则是从右侧扫描删除匹配count个元素。复杂度是O(N)，N是List长度，因为List的值不唯一，所以要遍历全部元素，而Set只要O(log(N))。

```
lpop key
```

从list的头部删除元素，并返回删除元素。如果key对应list不存在或者是空返回nil，如果key对应值不是list返回错误。

```
rpop
```

同上，但是从尾部删除。

2.3.7 设置list中指定下标的元素值

```
lset key index value
```

成功返回1，key或者下标不存在返回错误

2.3.8 阻塞队列

```
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（返回键和值）。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。

> 注意：不要采用其作为ajax的服务端推送，因为连接有限，遇到问题连接直接打满。

BLPOP/BRPOP 的先到先服务原则 如果有多个客户端同时因为某个列表而被阻塞，那么当有新值被推入到这个列表时，服务器会按照先到先服务（first in first service）原则，优先向最早被阻塞的客户端返回新值。举个例子，假设列表 lst 为空，那么当客户端 X 执行命令 BLPOP lst timeout 时，客户端 X 将被阻塞。在此之后，客户端 Y 也执行命令 BLPOP lst timeout ，也因此被阻塞。如果这时，客户端 Z 执行命令 RPUSH lst "hello" ，将值 "hello" 推入列表 lst ，那么这个 "hello" 将被返回给客户端 X ，而不是客户端 Y ，因为客户端 X 的被阻塞时间要早于客户端 Y 的被阻塞时间。

rpoplpush/brpoplpush：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，注意这是唯一一个操作两个列表的操作，用于两个队列交换消息。

应用场景：task + bak 双链表完成工作任务转交的安全队列，保证原子性。 业务逻辑: 1: Rpoplpush task bak 2: 接收返回值,并做业务处理 3: 完成时用LREM消掉。如不成功或者如果集群管理(如zookeeper)发现worker已经挂掉,下次从bak表里取任务

另一个应用场景是循环链表： 127.0.0.1:6379> lrange list 0 -1 1) "c" 2) "b" 3) "a" 127.0.0.1:6379> rpoplpush list list "a" 127.0.0.1:6379> lrange list 0 -1 1) "a" 2) "c" 3) "b"

2.4.1 添加元素

```
sadd key member
```

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

2.4.2 移除元素

```
srem key member
```

成功返回1，如果member在集合中不存在或者key不存在返回0，如果key对应的不是set类型的值返回错误

2.4.3 删除并返回元素

```
spop key
```

如果set是空或者key不存在返回nil

2.4.4 随机返回一个元素

```
srandmember key
```

同spop，随机取set中的一个元素，但是不删除元素

2.4.5 集合间移动元素

```
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

2.4.6 查看集合大小

```
scard key
```

如果set是空或者key不存在返回0

2.4.7 判断member是否在set中

```
sismember key member
```

存在返回1，0表示不存在或者key不存在

2.4.8 集合交集

```
sinter key1 key2...keyN
```

返回所有给定key的交集

```
sinterstore dstkey key1...keyN
```

同sinter，但是会同时将交集存到dstkey下

2.4.9 集合并集

```
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

2.4.10 集合差集

```
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

2.4.11 获取所有元素

```
smembers key
```

返回key对应set的所有元素，结果是无序的，集合元素很多时会阻塞，生产上禁用！

Sorted Set的实现是hash table(element->score, 用于实现ZScore及判断element是否在集合内)，和skip list(score->element,按score排序)的混合体。 skip list有点像平衡二叉树那样，不同范围的score被分成一层一层，每层是一个按score排序的链表。

ZAdd/ZRem是O(log(N))，ZRangeByScore/ZRemRangeByScore是O(log(N)+M)，N是Set大小，M是结果/操作元素的个数。可见，原本可能很大的N被很关键的Log了一下，1000万大小的Set，复杂度也只是几十不到。当然，如果一次命中很多元素M很大那谁也没办法了。

2.5.1 添加元素

```
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。

2.5.2 删除元素

```
zrem key member
```

1表示成功，如果元素不存在返回0

```
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score在给定区间的元素

2.5.3 增加score

```
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，可以为负数递减

2.5.4 获取排名

```
zrank key member
```

返回指定元素在集合中的排名（下标，注意不是分数）,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

2.5.5 获取排行榜

```
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,如果需要得分则加上withscores

> 注：index从start到end的所有元素

2.5.6 返回给定分数区间的元素

```
zrangebyscore key min max
```

可以指定inf为无穷

2.5.7 返回集合中score在给定区间的数量

```
zcount key min max
```

2.5.8 返回集合中元素个数

```
zcard key
```

2.5.9 返回给定元素对应的score

```
zscore key element
```

2.5.10 评分的聚合

```
ZUNIONSTORE destination numkeys key [key ...] [WEIGHTS weight] [AGGREGATE SUM|MIN|MAX]
```

例如：

```
127.0.0.1:6379> zrangebyscore votes -inf inf withscores
1) "sina"
2) "1"
3) "google"
4) "5"
5) "baidu"
6) "10"
127.0.0.1:6379> zrangebyscore visits -inf inf withscores
1) "baidu"
2) "1"
3) "google"
4) "5"
5) "sina"
6) "10"
127.0.0.1:6379> zunionstore award 2 visits votes weights 1 2 aggregate sum
(integer) 3
127.0.0.1:6379> zrangebyscore award -inf inf withscores
1) "sina"
2) "12"
3) "google"
4) "15"
5) "baidu"
6) "21"
```

一个小技巧是如果需要对评分进行倍加，则使用如下的方法：

```
127.0.0.1:6379>zrangebyscore visits -inf inf withscores
1) "baidu"
2) "1"
3) "google"
4) "5"
5) "sina"
6) "10"
127.0.0.1:6379>zunionstore visits 1 visits weights 2
(integer) 3
127.0.0.1:6379>zrangebyscore visits -inf inf withscores
1) "baidu"
2) "2"
3) "google"
4) "10"
5) "sina"
6) "20"
```

3. 专题功能

3.1 排序

redis支持对list，set和sorted set元素的排序。排序命令是sort 完整的命令格式如下：

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

复杂度为O(N+M*log(M))。(N是集合大小，M 为返回元素的数量)

说明：

1. [ASC|DESC] [ALPHA]: sort默认的排序方式（asc）是从小到大排的,当然也可以按照逆序或者按字符顺序排。
2. [BY pattern] : 除了可以按集合元素自身值排序外，还可以将集合元素内容按照给定pattern组合成新的key，并按照新key中对应的内容进行排序。例如：
3. 127.0.0.1:6379sort watch:leto by severtity:* desc
4. [GET pattern]：可以通过get选项去获取指定pattern作为新key对应的值，get选项可以有多个。例如：127.0.0.1:6379sort watch:leto by severtity: *get severtity:*。 对于Hash的引用，采用->，例如：sort watch:leto get # get bug:*->priority。
5. [LIMIT start count] 限定返回结果的数量。
6. [STORE dstkey] 把排序结果缓存起来

调用一次 SORT 命令可以给定多个 GET 选项。

```
redis> SADD names "peter" "tom" "jack"
(integer) 3

redis> MSET peter-name "Peter Hanson" jack-name "Jack Sparrow" tom-name "Thomas Edward"
OK

redis> MSET peter-id 10086 jack-id 255255 tom-id 123321
OK
```

```
redis> SORT names ALPHA GET # GET *-id GET *-name
 1) "jack"
 2) "255255"
 3) "Jack Sparrow"
 4) "peter"
 5) "10086"
 6) "Peter Hanson"
 7) "tom"
 8) "123321"
 9) "Thomas Edward"
```

当给定 GET # 时，命令会返回被排序的值本身。

```
redis> SORT team-member-ids BY *-KPI GET # GET *-name GET *-KPI
# 通过 KPI 值，对团队中各个成员的 ID 进行排序
# 然后根据排序结果，依次获取成员的 ID、名字和 KPI 值

redis> SORT names ALPHA DESC GET # GET *-id GET *-name LIMIT 0 5 STORE profiles
# 对 names 键的值进行文字形式的降序排序，
# 根据排序后的前五个值，获取值本身、及其对应的 ID 和名字
# 然后将获取到的这些信息储存到 profiles 键里面
```

3.2 事务

用Multi(Start Transaction)、Exec(Commit)、Discard(Rollback)实现。 在事务提交前，不会执行任何指令，只会把它们存到一个队列里，不影响其他客户端的操作。在事务提交时，批量执行所有指令。 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，但是当一个client在一个连接中发出multi命令后，这个连接会进入一个事务上下文，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会顺序的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。

Redis还提供了一个Watch功能，你可以对一个key进行Watch，然后再执行Transactions，在这过程中，如果这个Watched的值进行了修改，那么这个Transactions会发现并拒绝执行。

使用discard命令来取消一个事务。

注意：redis只能保证事务的每个命令连续执行（因为是单线程架构，在执行完事务内所有指令前是不可能再去同时执行其他客户端的请求的，也因此就不存在"事务内的查询要看到事务里的更新，在事务外查询不能看到"这个让人万分头痛的问题），但是如果事务中的一个命令失败了，并不回滚其他命令。另外，一个十分罕见的问题是当事务的执行过程中，如果redis意外的挂了。只有部分命令执行了，后面的也就被丢弃了。注意，如果是笔误，语法出现错误，则整个事务都无法执行。

当然如果我们使用的append-only file方式持久化，redis会用单个write操作写入整个事务内容。即是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。修复完后就能够重新启动了。

3.3 流水线

利用流水线（pipeline）的方式从client打包多条命令一起发出，不需要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：

```
cat data.txt | redis-cli –pipe
```

在选择开源redis开发库时需要着重注意是否支持pipeline，常见的jedis可以支持。

3.4 发布订阅

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。订阅者可以通过subscribe和psubscribe命令向redis server订阅自己感兴趣的消息类型，redis将消息类型称为频道(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。

```
终端A：
127.0.0.1:6379> subscribe comments
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "comments"
3) (integer) 1
终端B:
127.0.0.1:6379> subscribe comments
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "comments"
3) (integer) 1
终端C：
127.0.0.1:6379> publish comments good!
(integer) 2
终端A:
127.0.0.1:6379> subscribe comments
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "comments"
3) (integer) 1
1) "message"
2) "comments"
3) "good!"
终端B：
127.0.0.1:6379> subscribe comments
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "comments"
3) (integer) 1
1) "message"
2) "comments"
3) "good!"
```

可以使用psubscribe通过通配符进行多个channel的订阅

4. 开发设计规范

4.1 Key设计

key的一个格式约定：`object-type:id:field`。用":"分隔域，用"."作为单词间的连接，如"`comment:12345:reply.to`"。不推荐含义不清的key和特别长的key。

一般的设计方法如下： 1: 把表名转换为key前缀 如, tag: 2: 第2段放置用于区分区key的字段--对应mysql中的主键的列名,如userid 3: 第3段放置主键值,如2,3,4...., a , b ,c 4: 第4段,写要存储的列名

例如用户表 user, 转换为key-value存储：

userid	username	password	email
9	lisi	1111111	lisi@163.com

```
set user:userid:9:username lisi
set user:userid:9:password 111111
set user:userid:9:email   lisi@163.com
```

例如，查看某个用户的所有信息为：

```
keys user:userid:9*
```

如果另一个列也常常被用来查找，比如username，则也要相应的生成一条按照该列为主的key-value，例如：

```
user:username:lisi:uid 9
```

此时相当于RDBMS中在username上加索引，我们可以根据`username:lisi:uid`,查出userid=9，再查`user:9:password/email`...

4.2 超时设置

从业务需求逻辑和内存的角度，尽可能的设置key存活时间。

4.3 数据异常处理

程序应该处理如果redis数据丢失时的清理redis内存和重新加载的过程。

4.4 内存考虑

1. 只要有可能的话，就尽量使用散列键而不是字符串键来储存键值对数据，因为散列键管理方便、能够避免键名冲突、并且还能够节约内存。

   具体实例： 节约内存：Instagram的Redis实践 blog.nosqlfan.com/html/3379.html

2. 如果将redis作为cache进行频繁读写和超时删除等，此时应该避免设置较大的k-v，因为这样会导致redis的 内存碎片增加，导致rss占用较大，最后被操作系统OOM killer干掉。一个很具体的issue例子请见：https://github.com/antirez/redis/issues/2136

3. 如果采用序列化考虑通用性，请采用json相关的库进行处理，如果对内存大小和速度都很关注的，推荐使用messagepack进行序列化和反序列化

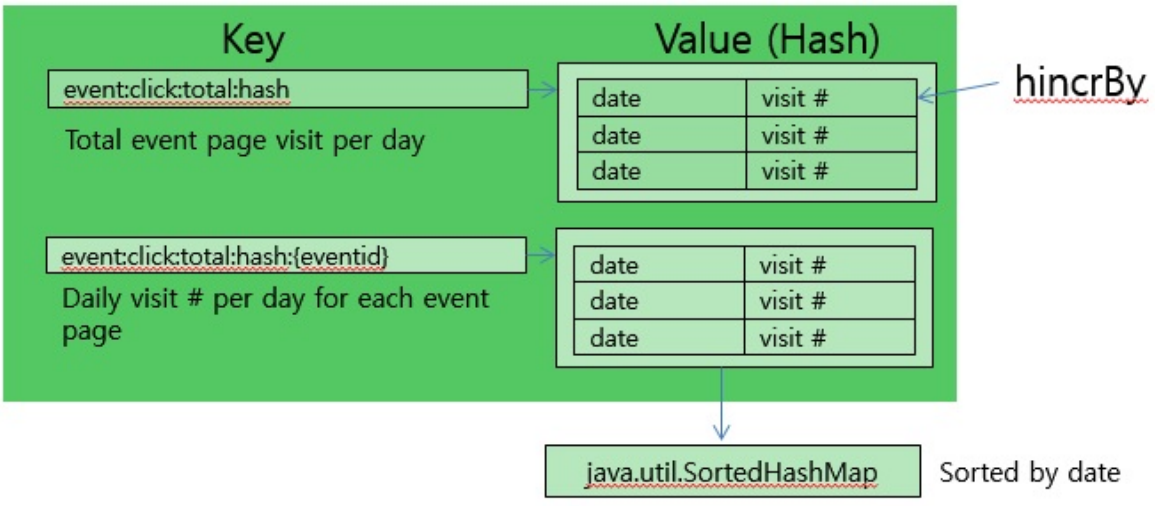
4. 如果需要计数器，请将计数器的key通过天或者小时分割，比如下边的设计:

event:click:total	visit #
event:click:{event page# id}	visit #
event:click:{event page# id}	visit #

   需要修改为：

event:click:daily:total:{date}	visit #
event:click:daily:{date}:{event page# id}	visit #
event:click:daily:{date}:{event page# id}	visit #

   更好的一个设计是采用hash：



5. 各种数据结构及其占用内存的benchmark测试

set个数	每个**set**的元素总数	内存占用	**Key**大小	**Value**大小
100	100	1.88M	7	36
100	1000	10.75M	7	36
100	10000	111.12M	7	36
1000	100	11.59M	8	36
1000	1000	100.35M	8	36
1000	10000	1.08G	8	36
10000	100	108.71M	9	36
10000	1000	996.23M	9	36

zset个数	每个**zset**的元素总数	内存占用	**Key**大小	**Value**大小
100	100	1.62M	8	49
100	1000	15.91M	8	49
100	10000	162.06M	8	49
1000	100	8.71M	9	49
1000	1000	151.87M	9	49
1000	10000	1.58G	9	49
10000	100	79.83M	10	49
10000	1000	1.48G	10	49

hash个数	每个**hash**的元素总数	内存占用	**Key**大小	**Value**大小
100	100	1.63M	8	49
100	1000	6.29M	8	49
100	10000	156.91M	8	49
1000	100	8.71M	9	49
1000	1000	55.59M	9	49
1000	10000	1.52G	9	49
10000	100	79.83M	10	49
10000	1000	548.58M	10	49

list个数	每个list的元素总数	内存占用	Key大小	Value大小
100	100	1.23M	8	36
100	1000	10.00M	8	36
100	10000	92.40M	8	36
1000	100	4.83M	9	36
1000	1000	92.52M	9	36
1000	10000	916.47M	9	36
10000	100	40.76M	10	36
10000	1000	917.69M	10	36

string个数	内存占用	Key大小	Value大小
100	846.79K	13	36
1000	966.29K	13	36
10000	2.16M	13	36
100000	130.88M	13	36

4.5 延迟考虑

1. 尽可能使用批量操作：

- mget、hmget而不是get和hget，对于set也是如此。
- lpush向一个list一次性导入多个元素，而不用lset一个个添加
- LRANGE 一次取出一个范围的元素，也不用LINDEX一个个取出

2. 尽可能的把redis和APP SERVER部署在一个网段甚至一台机器。

3. 对于数据量较大的集合，不要轻易进行删除操作，这样会阻塞服务器，一般采用重命名+批量删除的策略：

排序集合：

```
# Rename the key
newkey = "gc:hashes:" + redis.INCR("gc:index")
redis.RENAME("my.zset.key", newkey)

# Delete members from the sorted set in batche of 100s
while redis.ZCARD(newkey) > 0
  redis.ZREMRANGEBYRANK(newkey, 0, 99)
end
```

集合：

```
# Rename the key
newkey = "gc:hashes:" + redis.INCR("gc:index")
redis.RENAME("my.set.key", newkey)

# Delete members from the set in batches of 100
cursor = 0
loop
  cursor, members = redis.SSCAN(newkey, cursor, "COUNT", 100)
  if size of members > 0
redis.SREM(newkey, members)
  end
  if cursor == 0
break
  end
end
```

列表：

```
# Rename the key
newkey = "gc:hashes:" + redis.INCR("gc:index")
redis.RENAME("my.list.key", newkey)

# Trim off elements in batche of 100s
while redis.LLEN(newkey) > 0
  redis.LTRIM(newkey, 0, -99)
end
```

Hash：

```
# Rename the key
newkey = "gc:hashes:" + redis.INCR( "gc:index" )
redis.RENAME("my.hash.key", newkey)

# Delete fields from the hash in batche of 100s
cursor = 0
loop
  cursor, hash_keys = redis.HSCAN(newkey, cursor, "COUNT", 100)
  if hash_keys count > 0
redis.HDEL(newkey, hash_keys)
  end
  if cursor == 0
break
  end
end
```

4. 尽可能使用不要超过1M大小的kv。

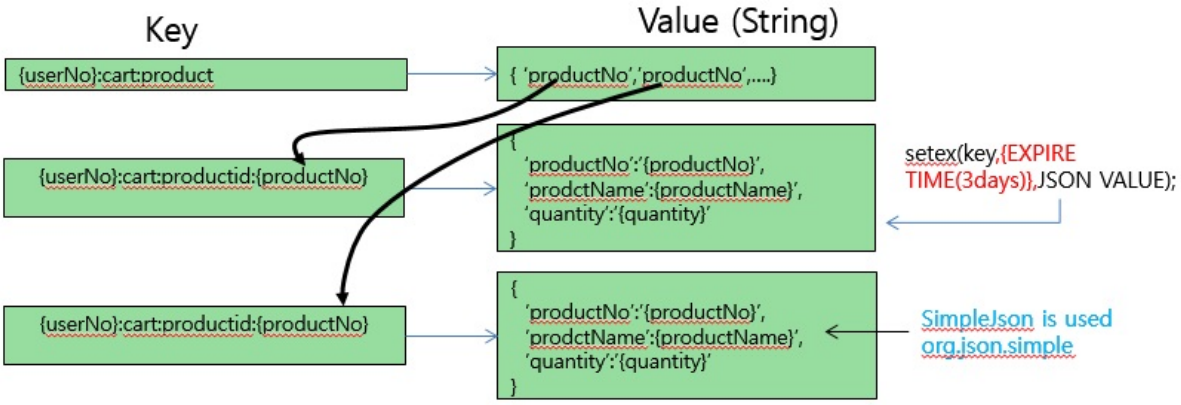
5. 减少对大数据集的高时间复杂度的操作：根据复杂度计算，如下命令可以优化:

命令	时间复杂度	描述	提升性能的建议
ZINTERSTORE	O(N*K)+O(M*log(M))	统计多个有序集合的交集，并存储结果	减少集合的和(或)结果的数量
SINTERSTORE	O(N * M)	统计多个集合的交集，并存储结果	尽可能减少集合的数量(大小)
SINTER	O(N * M)	返回多个集合的交集	尽可能减少集合的数量(大小)
MIGRATE	O(N)	从一个实例上传输key到另一个	减少对象值的数量和(或)大小
DUMP	O(N*M)	返回指定key的序列化结果	减少对象值的数量和(或)大小
ZREM	O(M*log(N))	从有序集合中移除一个或多个成员	减少移除成员的数量和(或)有序集合的大小
ZUNIONSTORE	O(N)+O(M log(M))	统计多个有序集合并集，并存储结果	减少总求并集的集合的总数量和(或)减少结果的数量
SORT	O(N+M*log(M))	排序list、set、sortedset的成员	减少排序的数量和(或)返回成员的数量
SDIFFSTORE	O(N)	统计多个set的差集并存储结果	减少总求差集的集合的数量
SDIFF	O(N)	返回多个set的差集	减少总求差集的集合的数量
SUNION	O(N)	统计多个set的并集	减少总求并集的集合的数量
LSET	O(N)	设置list中某个成员的值	减少list数据结构的长度
LREM	O(N)	从list中移除一定数量成员	减少list数据结构的长度
LRANGE	O(S+N)	返回list中指定区间的集合	减少偏移量和(或)区间的数量

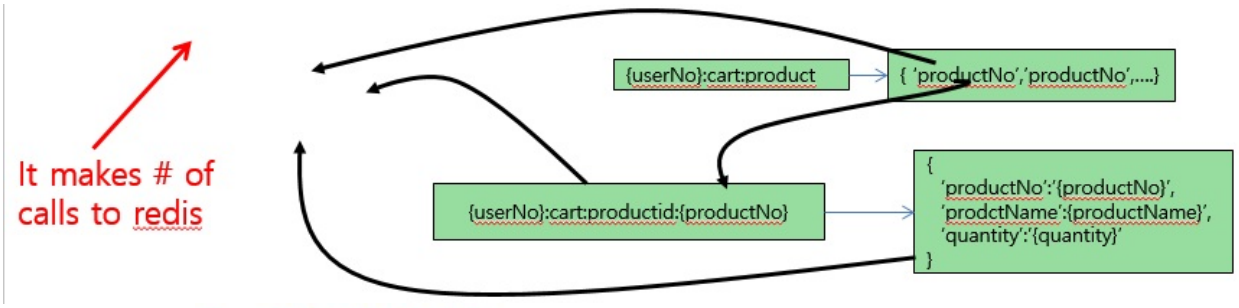
6. 尽可能使用**pipeline**操作：一次性的发送命令比一个个发要减少网络延迟和单个处理开销。一个性能测试结果为（注意并不是**pipeline**越大效率越高，注意最后一个测试结果）：

```
logger@BIGD1TMP:~> redis-benchmark -q -r 100000 -n 1000000 -c 50
PING_INLINE: 90155.07 requests per second
PING_BULK: 92302.02 requests per second
SET: 85070.18 requests per second
GET: 86184.61 requests per second

logger@BIGD1TMP:~> redis-benchmark -q -r 100000 -n 1000000 -c 50 -P 10
PING_INLINE: 558035.69 requests per second
PING_BULK: 668002.69 requests per second
SET: 275027.50 requests per second
GET: 376647.84 requests per second

logger@BIGD1TMP:~> redis-benchmark -q -r 100000 -n 1000000 -c 50 -P 20
PING_INLINE: 705716.25 requests per second
PING_BULK: 869565.25 requests per second
SET: 343406.59 requests per second
GET: 459347.72 requests per second

logger@BIGD1TMP:~> redis-benchmark -q -r 100000 -n 1000000 -c 50 -P 50
PING_INLINE: 940733.81 requests per second
PING_BULK: 1317523.00 requests per second
SET: 380807.31 requests per second
GET: 523834.47 requests per second

logger@BIGD1TMP:~> redis-benchmark -q -r 100000 -n 1000000 -c 50 -P 100
PING_INLINE: 999000.94 requests per second
PING_BULK: 1440922.12 requests per second
SET: 386996.88 requests per second
GET: 602046.94 requests per second

logger@BIGD1TMP:~> redis-benchmark -q -r 100000 -n 1000000 -c 50 -P 200
PING_INLINE: 1078748.62 requests per second
PING_BULK: 1381215.50 requests per second
SET: 379218.81 requests per second
GET: 537634.38 requests per second
```

一个场景是一个购物车的设计，一般的设计思路是：



在获取购物车内部货品时，不使用pipeline会很低效：




```
p = redis.pipelined()
getProductList()
for(productsNo){
  p.get(product)
}
List<Object> redisResult = p.syncAndReturnAll();
for(item:redisResult){
   json.add(item)
}
```

可以修改为：

7. 如果出现频繁对**string**进行**append**操作，则请使用**list**进行**push**操作，取出时使用**pop**。这样避免**string**频繁分配内存导致的延时。

8. 如果要**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。然后我们只在**slave**上做排序操作。把可能的对排序结果缓存。另外就是一个方案是就是采用**sorted set**对需要按某个顺序访问的集合建立索引。

4.6 典型使用场景参考

下面是Redis适用的一些场景:

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

```
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。

```
127.0.0.1:6379> zdd topapp 1 weixin
(error) ERR unknown command 'zdd'
127.0.0.1:6379> zadd topapp 1 weixin
(integer) 1
127.0.0.1:6379> zadd topapp 1 QQ
(integer) 1
127.0.0.1:6379> zadd topapp 2 meituan
(integer) 1
127.0.0.1:6379> zincrby topapp 1 meituan
"3"
127.0.0.1:6379> zincrby topapp 1 QQ
"2"
127.0.0.1:6379> zrank topapp QQ
(integer) 1
3) "meituan"
127.0.0.1:6379> zrank topapp weixin
(integer) 0
127.0.0.1:6379> zrange topapp 0 -1
1) "weixin"
2) "QQ"
```

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。

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

性能优于Memcached，数据结构更多样化。

10.使用setbit进行统计计数

```
#!/usr/bin/python
import redis
from bitarray import bitarray
from datetime import date

r=redis.Redis(host='localhost', port=6379, db=0)
today=date.today().strftime('%Y-%m-%d')

def bitcount(n):
len(bin(n)-2)

def setup():
r.delete('user:'+today)
r.setbit('user:'+today,100,0)

def setuniquser(uid):
r.setbit('user:'+today,uid,1)

def countuniquser():
a = bitarray()
a.frombytes(r.get('user:'+today),)
print a
print a.count()


if __name__=='__main__':
setup()
setuniquser(uid=0)
countuniquser()
```

11.维护好友关系

使用set进行是否为好友关系，共同好友等操作

12.使用 Redis 实现自动补全功能

使用有序集合保存输入结果：

```
ZADD word:a 0 apple 0 application 0 acfun 0 adobe
ZADD word:ap 0 apple 0 application
ZADD word:app 0 apple 0 application
ZADD word:appl 0 apple 0 application
ZADD word:apple 0 apple
ZADD word:appli 0 application
```

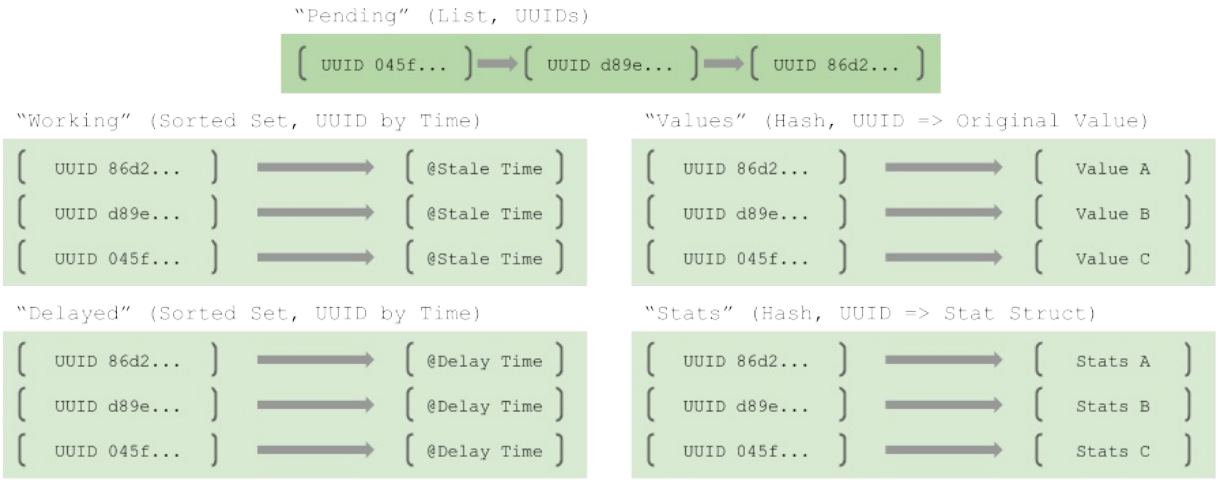
再使用一个有序集合保存热度：

```
ZADD word_scores 100 apple 80 adobe 70 application 60 acfun
```

取结果时采用交集操作：

```
ZINTERSTORE word_result 2 word_scores word:a WEIGHTS 1 1
ZRANGE word_result 0 -1 withscores
```

13. 可靠队列设计



• UUIDs as Surrogate Keys Our strategy spreads information about the state of an item in the queue across a number of Redis data structures, requiring the use of a per-item surrogate key to tie them together. The UUID is a good choice here because 1) they are quick to generate, and 2) can be generated by the clients in a decentralized manner. • Pending List The Pending List holds the generated UUIDs for the items that have been enqueued(), and are ready to be processed. It is a RedisList, presenting the generated UUIDs in FIFO order. • Values Hash The Values Hash holds the actual items that have been enqueued. It is a Redis Hash, mapping the generated UUID to the binary form of the the

item. This is the only representation of the original item that will appear in any of the data structures. • Stats Hash The Stats Hash records some timestamps and counts for each of the items. Specifically: • enqueue time • last dequeue time • dequeue count • last requeue time • last requeue count. It is a Redis Hash, mapping the generated UUID to a custom data structure that holds this data in a packed, binary form. Why keep stats on a per-item basis? We find it really useful for debugging (e.g. do we have a bad apple item that is being continuously requeued?), and for understanding how far behind we are if queues start to back up. Furthermore, the cost is only ~40 bytes per item, much smaller than our typical queued items. • Working Set The Working Set holds the set of UUIDs that have been dequeued(), and are currently being processed. It is a Redis Sorted Set, and scores each of the UUIDs by a pre-calculated, millisecond timestamp. Any object that has exceeded its assigned timeout is considered abandoned, and is available to be reclaimed. • Delay Set The Delay Set holds the set of UUIDs that have been requeued() with a per-item deferment. It is a Redis Sorted Set, and scores each of the UUIDs by a pre-calculated, millisecond timestamp. Once the deferment timestamp has expired, the item will be returned to the Pending List. Why support per-item deferment? We have a number of use cases where we might want to backoff specific pieces of work — maybe an underlying resource is too busy — without backing off the entire queue. Per-item deferment lets us say, “requeue this item, but don’t make it available for dequeue for another n seconds.”

5. 上线部署规划

2.1 内存规划

一定要设置最大内存maxmemory参数，否则物理内存用爆了就会大量使用Swap，写RDB文件时的速度很慢。注意这个参数指的是info中的used_memory，在一些不利于jmalloc的时候，内存碎片会很大。

多留55%内存是最安全的。重写AOF文件和RDB文件的进程(即使不做持久化，复制到Slave的时候也要写RDB)会fork出一条新进程来，采用了操作系统的Copy-On-Write策略(子进程与父进程共享Page。如果父进程的Page-每页4K有修改，父进程自己创建那个Page的副本，不会影响到子进程)。

另外，需要考虑内存碎片，假设碎片为1.2，则如果机器为64G，那么64*45%/1.2 = 24G作为maxmemory是比较安全的规划。

留意Console打出来的报告，如"RDB: 1215 MB of memory used by copy-on-write"。在系统极度繁忙时，如果父进程的所有Page在子进程写RDB过程中都被修改过了，就需要两倍内存。

按照Redis启动时的提醒，设置 vm.overcommit_memory = 1 ，使得fork()一条10G的进程时，因为COW策略而不一定需要有10G的free memory。

当最大内存到达时，按照配置的Policy进行处理， 默认策略为volatile-lru，对设置了expire time的key进行LRU清除(不是按实际expire time)。如果没有数据设置了expire time或者policy为noeviction，则直接报错，但此时系统仍支持get之类的读操作。 另外还有几种policy，比如volatile-ttl按最接近expire time的，allkeys-lru对所有key都做LRU。注意在一般的缓存系统中，如果没有设置超时时间，则lru的策略需要设置为allkeys-lru，并且应用需要做好未命中的异常处理。特殊的，当redis当做DB时，请使用noneviction策略，但是需要对系统内存监控加强粒度。

2.2 网卡RPS设置

RPS就是让网卡使用多核CPU的。传统方法就是网卡多队列（RSS，需要硬件和驱动支持），RPS则是在系统层实现了分发和均衡。如果对redis网络处理能力要求高或者在生产上发现cpu0的，可以在OS层面打开这个内核功能。

设置脚本：

```
#!/bin/bash
# Enable RPS (Receive Packet Steering)

rfc=32768
cc=$(grep -c processor /proc/cpuinfo)
rsfe=$(echo $cc*$rfc | bc)
sysctl -w net.core.rps_sock_flow_entries=$rsfe
for fileRps in $(ls /sys/class/net/eth*/queues/rx-*/rps_cpus)
do
echo fff > $fileRps
done

for fileRfc in $(ls /sys/class/net/eth*/queues/rx-*/rps_flow_cnt)
do
echo $rfc > $fileRfc
done

tail /sys/class/net/eth*/queues/rx-*/{rps_cpus,rps_flow_cnt}
```

2.3 服务器部署位置

尽可能把client和server部署在同一台机器上，比如都部署在app server，或者一个网段中，减少网络延迟对于redis的影响。

2.4 持久化设置

RDB和AOF两者毫无关系，完全独立运行，如果使用了AOF，重启时只会从AOF文件载入数据，不会再管RDB文件。在配置上有三种选择：不持久化，RDB，RDB+AOF。官方不推荐只开启AOF（因为恢复太慢另外如果aof引擎有bug），除非明显的读多写少的应用。 开启AOF时应当关闭AOF自动rewrite，并在crontab中启动在业务低峰时段进行的bgrewrite。 如果在一台机器上部署多个redis实例，则关闭RDB和AOF的自动保存（save "", auto-aof-rewrite-percentage 0），通过crontab定时调用保存：

```
m h * * * redis-cli -p <port> BGSAVE
m h */4 * * redis-cli -p <port> BGREWRITEAOF
```

2.5 多实例配置

如果一台机器上防止多个redis实例，为了防止上下文切换导致的开销，可以采用taskset。taskset是LINUX提供的一个命令(ubuntu系统可能需要自行安装，schedutils package)。他可以让某个程序运行在某个（或）某些CPU上。

1）显示进程运行的CPU （6137为redis-server的进程号）

```
[redis@hadoop1 ~]$ taskset  -p 6137
pid 6137's current affinity mask: f
```

显示结果的f实际上是二进制4个低位均为1的bitmask，每一个1对应于1个CPU，表示该进程在4个CPU上运行

2）指定进程运行在某个特定的CPU上

```
[redis@hadoop1 ~]$ taskset -pc 3 6137
pid 6137's current affinity list: 0-3
pid 6137's new affinity list: 3
```

注：3表示CPU将只会运行在第4个CPU上（从0开始计数）。

3）进程启动时指定CPU

```
taskset -c 1 ./redis-server ../redis.conf
```

参数：OPTIONS -p, --pid operate on an existing PID and not launch a new task

-c, --cpu-list specify a numerical list of processors instead of a bitmask. The list may contain multiple items, separated by comma, and ranges. For example, 0,5,7,9-11.

2.6 具体设置参数

详见安装包conf目录中的各个配置文件和上线前检查表格。

6. 常见运维操作

3.1 启动

3.1.1 启动redis

```
$ redis-server redis.conf
```

常见选项： ./redis-server (run the server with default conf) ./redis-server /etc/redis/6379.conf ./redis-server --port 7777 ./redis-server --port 7777 --slaveof 127.0.0.1 8888 ./redis-server /etc/myredis.conf --loglevel verbose

3.1.2 启动redis-sentinel

```
    ./redis-server /etc/sentinel.conf –sentinel
    ./redis-sentinel /etc/sentinel.conf
```

部署后可以使用sstart对redis 和sentinel进行拉起，使用sctl进行supervisorctl的控制。（两个alias）

3.2 停止

```
redis-cli shutdown
```

sentinel方法一样，只是需要执行sentinel的连接端口

> 注意：正确关闭服务器方式是redis-cli shutdown 或者 kill，都会graceful shutdown，保证写RDB文件以及将AOF文件fsync到磁盘，不会丢失数据。 如果是粗暴的Ctrl+C，或者kill -9 就可能丢失。

3.3 查看和修改配置

查看：

```
config get ：获取服务器配置信息。
redis 127.0.0.1:6379> config get dir
config get *：查看所有配置
```

修改：

```
临时设置：config set
永久设置：config rewrite，将目前服务器的参数配置写入redis conf.
```

3.4 批量执行操作

使用telnet也可以连接redis-server。并且在脚本中使用nc命令进行redis操作也是很有效的：

```
gnuhpc@gnuhpc:~$ (echo -en "ping\r\nset key abc\r\nget key\r\n";sleep 1) | nc 127.0.0.1 6
+PONG
+OK
$3
abc
```

另一个方式是使用pipeline：

```
在一个脚本中批量执行多个写入操作:
先把插入操作放入操作文本insert.dat：
set a b
set 1 2
set h w
set f u
然后执行命令:cat insert.bat | ./redis-cli --pipe, 或者如下脚本：
#!/bin/sh
host=$1
port=$;
password=$3
cat insert.dat | ./redis-cli -h $host -p $port -a $password --pipe
```

3.5 选择数据库

```
select db-index
```

默认连接的数据库所有是0,默认数据库数是16个。返回1表示成功，0失败

3.6 清空数据库

```
flushdb
```

删除当前选择数据库中的所有 key。生产上已经禁止。

```
flushall
```

删除所有的数据库。生产上已经禁止。

3.7 重命名命令

```
rename-command
```

例如：rename-command FLUSHALL ""。必须重启。

3.8 执行lua脚本

```
--eval
```

例如： redis-cli --eval myscript.lua key1 key2 , arg1 arg2 arg3

3.9 设置密码

```
config set requirepass [passw0rd]
```

3.10 验证密码

```
auth passw0rd
```

3.11 性能测试命令

```
redis-benchmark -q -r 100000 -n 100000 -c 50
```

比如：开100条线程(默认50)，SET 1千万次(key在0-1千万间随机)，key长21字节，value长256字节的数据。-r指的是使用随机key的范围。

```
redis-benchmark -t SET -c 100 -n 10000000 -r 10000000 -d 256
```

也可以直接执行lua脚本模拟客户端

```
redis-benchmark -n 100000 -q script load "redis.call('set','foo','bar')"
```

注意：Redis-Benchmark的测试结果提供了一个保证你的 Redis-Server 不会运行在非正常状态下的基准点，但是你永远不要把它作为一个真实的“压力测试”。压力测试需要反应出应用的运行方式，并且需要一个尽可能的和生产相似的环境。

3.12 Redis-cli命令行其他操作

1. echo ：在命令行打印一些内容

redis 127.0.0.1:6379> echo HongWan "HongWan"

2. quit ：退出连接。

redis 127.0.0.1:6379> quit

3. -x选项从标准输入（stdin）读取最后一个参数。 比如从管道中读取输入：

echo -en "chen.qun" | redis-cli -x set name

4. -r -i

-r 选项重复执行一个命令指定的次数。 -i 设置命令执行的间隔。 比如查看redis每秒执行的commands（qps） redis-cli -r 100 -i 1 info stats | grep instantaneous_ops_per_sec

5. -c：开启reidis cluster模式，连接redis cluster节点时候使用。

6. --rdb：获取指定redis实例的rdb文件,保存到本地。

redis-cli -h 192.168.44.16 -p 6379 --rdb 6379.rdb

7. --slave

模拟slave从master上接收到的commands。slave上接收到的commands都是update操作，记录数据的更新行为。

8. --pipe

这个一个非常有用的参数。发送原始的redis protocl格式数据到服务器端执行。比如下面的形式的数据（linux服务器上需要用unix2dos转化成dos文件）。 linux下默认的换行是\n,windows系统的换行符是\r\n，redis使用的是\r\n. echo -en '*3\r\n$3\r\nSET\r\n$3\r\nkey\r\n$5\r\nvalue\r\n' | redis-cli --pipe

6.13 持久化与备份恢复

3.13.1 RDB相关操作

BGSAVE：后台子进程进行RDB持久化 SAVE：主进程进行RDB，生产环境千万别用，服务器将无法响应任何操作。 LASTSAVE： 返回上一次成功SAVE的Unix时间

```
redis-cli config set save ""
```

动态关闭RDB

3.13.2 AOF相关操作

```
BGREWRITEAOF
```

在后台执行一个 AOF文件重写操作

3.13.3 备份

对于RDB和AOF，都是直接拷贝文件即可，可以设定crontab进行定时备份： cp /var/lib/redis/dump.rdb /somewhere/safe/dump.$(date +%Y%m%d%H%M).rdb

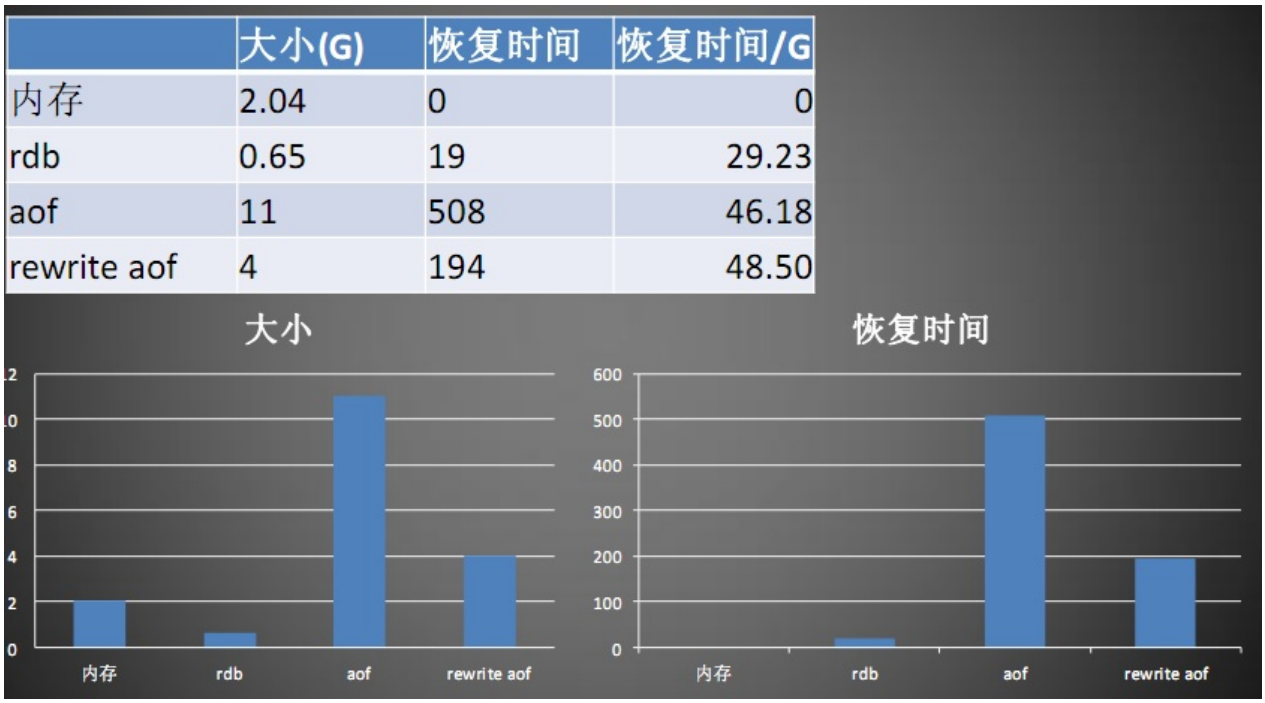
3.13.4 恢复

如果只使用了RDB，则首先将redis-server停掉，删除dump.rdb，最后将备份的dump.rdb文件拷贝回data目录并修改相关属主保证其属主和redis-server启动用户一致，然后启动redis-server。

如果是RDB+AOF的持久化，只需要将aof文件放入data目录，启动redis-server，查看是否恢复，如无法恢复则应该将aof关闭后重启，redis就会从rdb进行恢复了，随后调用命令BGREWRITEAOF进行AOF文件写入，在info的aof_rewrite_in_progress为0后一个新的aof文件就生成了，此时再将配置文件的aof打开，再次重启redis-server就可以恢复了。注意先不要将dump.rdb放入data目录，否则会因为aof文件万一不可用，则rdb也不会被恢复进内存，此时如果有新的请求进来后则原先的rdb文件被重写。

恢复速度参见新浪的测试结果：

	大小(G)	恢复时间	恢复时间/G
内存	2.04	0	0
rdb	0.65	19	29.23
aof	11	508	46.18
rewrite aof	4	194	48.50



这个结果是可信的，在一台SSD、4个CPU的虚拟机上测试为28.3G/s.

检查修复AOF文件：

```
redis-check-aof data/appendonly.aof
```

7. 数据迁移

4.1 将key从当前数据库移动到指定数据库

```
move key db-index
```

返回1成功。0 如果key不存在，或者已经在指定数据库中

8. 数据迁移

8.1 一般处理流程

6.1.1 探测服务是否可用

```
127.0.0.1:6379> ping
```

返回PONG说明正常

6.1.2 探测服务延迟

redis-cli --latency 显示的单位是milliseconds，一个千兆网延迟在0.2ms上就有问题，作为参考，百兆网一般的平均为0.08ms。

6.1.3 监控正在请求执行的命令 在cli下执行monitor，生产环境慎用。

6.1.4 查看统计信息

```
Mrds:6379> info
```

在cli下执行info。

```
info Replication
```

只看其中一部分。

```
config resetstat
```

重新统计。

```
# Server
redis_version:2.8.19 ###redis版本号
redis_git_sha1:00000000  ###git SHA1
redis_git_dirty:0   ###git dirty flag
redis_build_id:78796c63e58b72dc
redis_mode:standalone   ###redis运行模式
os:Linux 2.6.32-431.el6.x86_64 x86_64   ###os版本号
arch_bits:64  ###64位架构
multiplexing_api:epoll  ###调用epoll算法
gcc_version:4.4.7   ###gcc版本号
process_id:25899   ###服务器进程PID
run_id:eae356ac1098c13b68f2b00fd7e1c9f93b1c6a2c   ###Redis的随机标识符(用于sentinel和集群)
tcp_port:6379   ###Redis监听的端口号
uptime_in_seconds:6419 ###Redis运行时长(s为单位)
uptime_in_days:0  ###Redis运行时长(天为单位)
hz:10
lru_clock:10737922  ###以分钟为单位的自增时钟,用于LRU管理
config_file:/etc/redis/redis.conf   ###redis配置文件

# Clients
connected_clients:1   ###已连接客户端的数量（不包括通过从属服务器连接的客户端）这个参数也要一定关注，有
client_longest_output_list:0   ###当前连接的客户端中最长的输出列表
client_biggest_input_buf:0   ###当前连接的客户端中最大的。输出缓存
blocked_clients:0  ###正在等待阻塞命令（BLPOP、BRPOP、BRPOPLPUSH）的客户端的数量 需监控

# Memory
used_memory:2281560   ###由 Redis 分配器分配的内存总量，以字节（byte）为单位
used_memory_human:2.18M   ###以更友好的格式输出redis占用的内存
used_memory_rss:2699264   ###从操作系统的角度，返回 Redis 已分配的内存总量（俗称常驻集大小）。这个值
used_memory_peak:22141272  ### Redis 的内存消耗峰值（以字节为单位）
used_memory_peak_human:21.12M  ###以更友好的格式输出redis峰值内存占用
used_memory_lua:35840  ###LUA引擎所使用的内存大小
mem_fragmentation_ratio:1.18  ###   =used_memory_rss /used_memory 这两个参数都包含保存用户k-v
mem_allocator:jemalloc-3.6.0
```

```
# Persistence
loading:0  ###记录服务器是否正在载入持久化文件，1为正在加载
rdb_changes_since_last_save:0   ###距离最近一次成功创建持久化文件之后，产生了多少次修改数据集的操作
rdb_bgsave_in_progress:0   ###记录了服务器是否正在创建 RDB 文件，1为正在进行
rdb_last_save_time:1420023749  ###最近一次成功创建 RDB 文件的 UNIX 时间戳
rdb_last_bgsave_status:ok   ###最近一次创建 RDB 文件的结果是成功还是失败
rdb_last_bgsave_time_sec:0  ###最近一次创建 RDB 文件耗费的秒数
rdb_current_bgsave_time_sec:-1  ###如果服务器正在创建 RDB 文件，那么这个域记录的就是当前的创建操作已
aof_enabled:1   ###AOF 是否处于打开状态，1为启用
aof_rewrite_in_progress:0   ###服务器是否正在创建 AOF 文件
aof_rewrite_scheduled:0   ###RDB 文件创建完毕之后，是否需要执行预约的 AOF 重写操作（因为在RDB时aof
aof_last_rewrite_time_sec:-1  ###最近一次创建 AOF 文件耗费的时长
aof_current_rewrite_time_sec:-1  ###如果服务器正在创建 AOF 文件，那么这个域记录的就是当前的创建操作
aof_last_bgrewrite_status:ok  ###最近一次创建 AOF 文件的结果是成功还是失败
aof_last_write_status:ok
aof_current_size:176265  ###AOF 文件目前的大小
aof_base_size:176265  ###服务器启动时或者 AOF 重写最近一次执行之后，AOF 文件的大小
aof_pending_rewrite:0  ###是否有 AOF 重写操作在等待 RDB 文件创建完毕之后执行
aof_buffer_length:0   ###AOF 缓冲区的大小
aof_rewrite_buffer_length:0  ###AOF 重写缓冲区的大小
aof_pending_bio_fsync:0  ###后台 I/O 队列里面，等待执行的 fsync 调用数量
aof_delayed_fsync:0###被延迟的 fsync 调用数量
loading_start_time:1441769386   loading启动时间戳
loading_total_bytes:1787767808   loading需要加载数据量
loading_loaded_bytes:1587418182  已经加载的数据量
loading_loaded_perc:88.79 加载百分比
loading_eta_seconds:7   剩余时间

# Stats
total_connections_received:8466  ###服务器已接受的连接请求数量
total_commands_processed:900668   ###服务器已执行的命令数量，这个数值需要持续监控，如果在一段时间内
instantaneous_ops_per_sec:1   ###服务器每秒钟执行的命令数量
total_net_input_bytes:82724170
total_net_output_bytes:39509080
instantaneous_input_kbps:0.07
instantaneous_output_kbps:0.02
rejected_connections:0  ###因为最大客户端数量限制而被拒绝的连接请求数量
sync_full:2
sync_partial_ok:0
sync_partial_err:0
expired_keys:0   ###因为过期而被自动删除的数据库键数量
evicted_keys:0   ###因为最大内存容量限制而被驱逐（evict）的键数量。这个数值如果不是0则说明maxmemory有
keyspace_hits:0  ###查找数据库键成功的次数。可以计算命中率
keyspace_misses:500000   ###查找数据库键失败的次数。
pubsub_channels:0  ###目前被订阅的频道数量
pubsub_patterns:0  ###目前被订阅的模式数量
latest_fork_usec:402  ###最近一次 fork() 操作耗费的毫秒数

# Replication
role:master   ###如果当前服务器没有在复制任何其他服务器，那么这个域的值就是 master ；否则的话，这个域的
connected_slaves:2   ###2个slaves
slave0:ip=192.168.65.130,port=6379,state=online,offset=1639,lag=1
```

```
slave1:ip=192.168.65.129,port=6379,state=online,offset=1639,lag=0
master_repl_offset:1639
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:2
repl_backlog_histlen:1638

# CPU
used_cpu_sys:41.87  ###Redis 服务器耗费的系统 CPU
used_cpu_user:17.82  ###Redis 服务器耗费的用户 CPU
used_cpu_sys_children:0.01  ###后台进程耗费的系统 CPU
used_cpu_user_children:0.01  ###后台进程耗费的用户 CPU

# Keyspace
db0:keys=3101,expires=0,avg_ttl=0   ###keyspace 部分记录了数据库相关的统计信息，比如数据库的键数量
```

6.1.5 获取慢查询

```
SLOWLOG GET 10
```

结果为查询ID、发生时间、运行时长和原命令 默认10毫秒，默认只保留最后的128条。单线程的模型下，一个请求占掉10毫秒是件大事情，注意设置和显示的单位为微秒，注意这个时间是不包含网络延迟的。

```
slowlog get
```

获取慢查询日志

```
slowlog len
```

获取慢查询日志条数

```
slowlog reset
```

清空慢查询

6.1.6 查看客户端

```
client list
```

列出所有连接

```
client kill
```

杀死某个连接， 例如 `CLIENT KILL 127.0.0.1:43501`

```
client getname #
```

获取连接的名称 默认nil

```
client setname "名称"
```

设置连接名称,便于调试

6.1.7 查看日志

日志位置在/redis/log下，redis.log为redis主日志，sentinel.log为sentinel监控日志。

6.2 并发延迟检查

top看到单个CPU 100%时，就是垂直扩展的时候了。如果需要让CPU使用率最大化，可以配置Redis实例数对应CPU数, Redis实例数对应端口数(8核Cpu, 8个实例, 8个端口), 以提高并发。单机测试时, 单条数据在200字节, 测试的结果为8~9万tps。（未实测）。 另外，对于命令的复杂度一定要关注。

6.2.1 检查CPU情况

```
mpstat -P ALL 1
```

6.2.2 检查系统情况

着重检查探测服务延迟、 监控正在请求执行的命令、获取慢查询

6.2.3 检查连接数

```
total_connections_received
```

如果过高请修改应用不要频繁关闭连接redis

6.2.4 检查持久化

RDB的时间：`latest_fork_usec:936` 上次导出rdb快照,持久化花费，微秒。 检查是否有人使用了SAVE。

6.2.5 检查命令执行情况

```
INFO commandstats
```

查看命令执行了多少次，执行命令所耗费的毫秒数(每个命令的总时间和平均时间)

8.3 内存检查

6.3.1 系统内存查看

script/下的memstat.sh或者ps_mem.py都可以查看系统的内存情况，两个工具都需要root权限。

6.3.2 系统swap内存查看

```
#!/bin/bash
# Get current swap usage for all running processes
# Erik Ljungstrom 27/05/2011
# Modified by Mikko Rantalainen 2012-08-09
# Pipe the output to "sort -nk3" to get sorted output
# Modified by Marc Methot 2014-09-18
# removed the need for sudo

SUM=0
OVERALL=0
for DIR in `find /proc/ -maxdepth 1 -type d -regex "^/proc/[0-9]+"`
do
    PID=`echo $DIR | cut -d / -f 3`
    PROGNAME=`ps -p $PID -o comm --no-headers`
for SWAP in `grep VmSwap $DIR/status 2>/dev/null | awk '{ print $2 }'`
do
    let SUM=$SUM+$SWAP
done
if (( $SUM > 0 )); then
    echo "PID=$PID swapped $SUM KB ($PROGNAME)"
fi
let OVERALL=$OVERALL+$SUM
SUM=0
done
echo "Overall swap used: $OVERALL KB"
```

6.3.3 info查看内存

`used_memory:859192` 数据结构的空间 `used_memory_rss:7634944` 实占空间 `mem_fragmentation_ratio:8.89` 前2者的比例,1.N为佳,如果此值过大,说明redis的内存的碎片化严重,可以导出再导入一次.

6.3.4 dump.rdb文件成生内存报告（rdb-tool）

```
# rdb -c memory ./dump.rdb > redis_memory_report.csv
# sort -t, -k4nr redis_memory_report.csv
```

6.3.5 query在线分析

```
redis-cli MONITOR | head -n 5000 | ./redis-faina.py
```

6.3.6 内存抽样分析

```
/redis/script/redis-sampler.rb 127.0.0.1 6379 0 10000
/redis/script/redis-audit.rb  127.0.0.1 6379 0 10000
```

6.3.7 统计生产上比较大的key

```
./redis-cli --bigkeys
```

对redis中的key进行采样，寻找较大的keys。是用的是scan方式，不用担心会阻塞redis很长时间不能处理其他的请求。执行的结果可以用于分析redis的内存的只用状态，每种类型key的平均大小。

6.3.8 查看key内部结构和编码等信息

```
debug object
```

查看一个key内部信息，比如refcount、encoding、serializedlength等，结果如下 Value at:0x7f21b9479850 refcount:1 encoding:raw serializedlength:6 lru:8462202 lru_seconds_idle:215

6.3.9 Rss增加，内存碎片增加

此时可以选择时间进行redis服务器的重新启动，并且注意在rss突然降低观察是否swap被使用，以确定并非是因为swap而导致的rss降低。

一个典型的例子是：http://grokbase.com/t/gg/redis-db/14ag5n9qhv/redis-memory-fragmentation-ratio-reached-5000

9. 测试方法

7.1 模拟oom

```
redis-cli debug oom
```

redis直接退出。

7.2 模拟宕机

```
redis-cli debug segfault
```

7.3 模拟hang

```
redis-cli -p 6379 DEBUG sleep 30
```

7.4 快速产生测试数据

```
debug populate
```

测试利器，快速产生大量的key

```
127.0.0.1:6379> debug populate 10000
OK
127.0.0.1:6379> dbsize
(integer) 10000
```

7.5 模拟RDB load情形

```
debug reload
```

save当前的rdb文件，并清空当前数据库，重新加载rdb，加载与启动时加载类似，加载过程中只能服务部分只读请求（比如info、ping等）： rdbSave(); emptyDb(); rdbLoad();

7.6 模拟AOF加载情形

```
debug loadaof
```

清空当前数据库，重新从aof文件里加载数据库 emptyDb(); loadAppendOnlyFile();