Redis基础、高级特性与性能调优

本文将从Redis的基本特性入手，通过讲述Redis的数据结构和主要命令对Redis的基本能力进行直观介绍。之后概览Redis提供的高级能力，并在部署、维护、性能调优等多个方面进行更深入的介绍和指导。本文适合使用Redis的普通开发人员，以及对Redis进行选型、架构设计和性能调优的架构设计人员。

目录

概述

Redis是一个开源的，基于内存的结构化数据存储媒介，可以作为数据库、缓存服务或消息服务使用。

Redis支持多种数据结构，包括字符串、哈希表、链表、集合、有序集合、位图、Hyperloglogs等。

Redis具备LRU淘汰、事务实现、以及不同级别的硬盘持久化等能力，并且支持副本集和通过Redis Sentinel实现的高可用方案，同时还支持通过Redis Cluster实现的数据自动分片能力。

Redis的主要功能都基于单线程模型实现，也就是说Redis使用一个线程来服务所有的客户端请求，同时Redis采用了非阻塞式IO，并精细地优化各种命令的算法时间复杂度，这些信息意味着：

- Redis是线程安全的（因为只有一个线程），其所有操作都是原子的，不会因并发产生数据异常
- Redis的速度非常快（因为使用非阻塞式IO，且大部分命令的算法时间复杂度都是O(1)）
- 使用高耗时的Redis命令是很危险的，会占用唯一的一个线程的大量处理时间，导致所有的请求都被拖慢。（例如时间复杂度为O(N)的KEYS命令，严格禁止在生产环境中使用）

Redis的数据结构和相关常用命令

本节中将介绍Redis支持的主要数据结构，以及相关的常用Redis命令。本节只对Redis命令进行扼要的介绍，且只列出了较常用的命令。如果想要了解完整的Redis命令集，或了解某个命令的详细使用方法，请参考官方文档：https://redis.io/commands

Key

Redis采用Key-Value型的基本数据结构，任何二进制序列都可以作为Redis的Key使用（例如普通的字符串或一张JPEG图片）

关于Key的一些注意事项：

- 不要使用过长的Key。例如使用一个1024字节的key就不是一个好主意，不仅会消耗更多的内存，还会导致查找的效率降低
- Key短到缺失了可读性也是不好的，例如"u1000flw"比起"user:1000:followers"来说，节省了寥寥的存储空间，却引发了可读性和可维护性上的麻烦

- 最好使用统一的规范来设计Key，比如"object-type:id:attr"，以这一规范设计出的Key可能是"user:1000"或"comment:1234:reply-to"
- Redis允许的最大Key长度是512MB（对Value的长度限制也是512MB）

String

String是Redis的基础数据类型，Redis没有Int、Float、Boolean等数据类型的概念，所有的基本类型在Redis中都以String体现。
与String相关的常用命令：

- **SET**：为一个key设置value，可以配合EX/PX参数指定key的有效期，通过NX/XX参数针对key是否存在的情况进行区别操作，时间复杂度O(1)
- **GET**：获取某个key对应的value，时间复杂度O(1)
- **GETSET**：为一个key设置value，并返回该key的原value，时间复杂度O(1)
- **MSET**：为多个key设置value，时间复杂度O(N)
- **MSETNX**：同MSET，如果指定的key中有任意一个已存在，则不进行任何操作，时间复杂度O(N)
- **MGET**：获取多个key对应的value，时间复杂度O(N)

上文提到过，Redis的基本数据类型只有String，但Redis可以把String作为整型或浮点型数字来使用，主要体现在INCR、DECR类的命令上：

- **INCR**：将key对应的value值自增1，并返回自增后的值。只对可以转换为整型的String数据起作用。时间复杂度O(1)

- **INCRBY**：将key对应的value值自增指定的整型数值，并返回自增后的值。只对可以转换为整型的String数据起作用。时间复杂度O(1)
- **DECR/DECRBY**：同INCR/INCRBY，自增改为自减。

INCR/DECR系列命令要求操作的value类型为String，并可以转换为64位带符号的整型数字，否则会返回错误。
也就是说，进行INCR/DECR系列命令的value，必须在[-2^63 ~ 2^63 - 1]范围内。
前文提到过，Redis采用单线程模型，天然是线程安全的，这使得INCR/DECR命令可以非常便利的实现高并发场景下的精确控制。

例1：库存控制

在高并发场景下实现库存余量的精准校验，确保不出现超卖的情况。
设置库存总量：

```
SET inv:remain "100"
```

库存扣减+余量校验：

```
DECR inv:remain
```

当DECR命令返回值大于等于0时，说明库存余量校验通过，如果返回小于0的值，则说明库存已耗尽。
假设同时有300个并发请求进行库存扣减，Redis能够确保这300个请求分别得到99到-200的返回值，每个请求得到的返回值都是唯一的，绝对不会找出现两个请求得到一样的返回值的情况。

例2：自增序列生成

实现类似于RDBMS的Sequence功能，生成一系列唯一的序列号
设置序列起始值：

```
SET sequence "10000"
```

获取一个序列值：

```
INCR sequence
```

直接将返回值作为序列使用即可。

获取一批（如100个）序列值：

```
INCRBY sequence 100
```

假设返回值为N，那么[N - 99 ~ N]的数值都是可用的序列值。

当多个客户端同时向Redis申请自增序列时，Redis能够确保每个客户端得到的序列值或序列范围都是全局唯一的，绝对不会出现不同客户端得到了重复的序列值的情况。

List

Redis的List是链表型的数据结构，可以使用LPUSH/RPUSH/LPOP/RPOP等命令在List的两端执行插入元素和弹出元素的操作。虽然List也支持在特定index上插入和读取元素的功能，但其时间复杂度较高（O(N)），应小心使用。

与List相关的常用命令：

- **LPUSH**：向指定List的左侧（即头部）插入1个或多个元素，返回插入后的List长度。时间复杂度O(N)，N为插入元素的数量
- **RPUSH**：同LPUSH，向指定List的右侧（即尾部）插入1或多个元素
- **LPOP**：从指定List的左侧（即头部）移除一个元素并返回，时间复杂度O(1)
- **RPOP**：同LPOP，从指定List的右侧（即尾部）移除1个元素并返回
- **LPUSHX/RPUSHX**：与LPUSH/RPUSH类似，区别在于，LPUSHX/RPUSHX操作的key如果不存在，则不会进行任何操作

- **LLEN**：返回指定List的长度，时间复杂度O(1)
- **LRANGE**：返回指定List中指定范围的元素（双端包含，即LRANGE key 0 10会返回11个元素），时间复杂度O(N)。应尽可能控制一次获取的元素数量，一次获取过大范围的List元素会导致延迟，同时对长度不可预知的List，避免使用LRANGE key 0 -1这样的完整遍历操作。

应谨慎使用的List相关命令：

- **LINDEX**：返回指定List指定index上的元素，如果index越界，返回nil。index数值是回环的，即-1代表List最后一个位置，-2代表List倒数第二个位置。时间复杂度O(N)
- **LSET**：将指定List指定index上的元素设置为value，如果index越界则返回错误，时间复杂度O(N)，如果操作的是头/尾部的元素，则时间复杂度为O(1)
- **LINSERT**：向指定List中指定元素之前/之后插入一个新元素，并返回操作后的List长度。如果指定的元素不存在，返回-1。如果指定key不存在，不会进行任何操作，时间复杂度O(N)

由于Redis的List是链表结构的，上述的三个命令的算法效率较低，需要对List进行遍历，命令的耗时无法预估，在List长度大的情况下耗时会明显增加，应谨慎使用。
换句话说，Redis的List实际是设计来用于实现队列，而不是用于实现类似ArrayList这样的列表的。如果你不是想要实现一个双端出入的队列，那么请尽量不要使用Redis的List数据结构。
为了更好支持队列的特性，Redis还提供了一系列阻塞式的操作命令，如BLPOP/BRPOP等，能够实现类似于BlockingQueue的能力，即在List为空时，阻塞该连接，直到List中有对象可以出队时再返回。针对

阻塞类的命令，此处不做详细探讨，请参考官方文档（https://redis.io/topics/data-types-intro ）中"Blocking operations on lists"一节。

Hash

Hash即哈希表，Redis的Hash和传统的哈希表一样，是一种field-value型的数据结构，可以理解成将HashMap搬入Redis。

Hash非常适合用于表现对象类型的数据，用Hash中的field对应对象的field即可。

Hash的优点包括：

- 可以实现二元查找，如"查找ID为1000的用户的年龄"
- 比起将整个对象序列化后作为String存储的方法，Hash能够有效地减少网络传输的消耗
- 当使用Hash维护一个集合时，提供了比List效率高得多的随机访问命令

与Hash相关的常用命令：

- **HSET**：将key对应的Hash中的field设置为value。如果该Hash不存在，会自动创建一个。时间复杂度O(1)
- **HGET**：返回指定Hash中field字段的值，时间复杂度O(1)
- **HMSET/HMGET**：同HSET和HGET，可以批量操作同一个key下的多个field，时间复杂度：O(N)，N为一次操作的field数量
- **HSETNX**：同HSET，但如field已经存在，HSETNX不会进行任何操作，时间复杂度O(1)
- **HEXISTS**：判断指定Hash中field是否存在，存在返回1，不存在返回0，时间复杂度O(1)

- **HDEL**：删除指定Hash中的field（1个或多个），时间复杂度：O(N)，N为操作的field数量
- **HINCRBY**：同INCRBY命令，对指定Hash中的一个field进行INCRBY，时间复杂度O(1)

应谨慎使用的Hash相关命令：

- **HGETALL**：返回指定Hash中所有的field-value对。返回结果为数组，数组中field和value交替出现。时间复杂度O(N)
- **HKEYS/HVALS**：返回指定Hash中所有的field/value，时间复杂度O(N)

上述三个命令都会对Hash进行完整遍历，Hash中的field数量与命令的耗时线性相关，对于尺寸不可预知的Hash，应严格避免使用上面三个命令，而改为使用HSCAN命令进行游标式的遍历，具体请见 https://redis.io/commands/scan

Set

Redis Set是无序的，不可重复的String集合。

与Set相关的常用命令：

- **SADD**：向指定Set中添加1个或多个member，如果指定Set不存在，会自动创建一个。时间复杂度O(N)，N为添加的member个数
- **SREM**：从指定Set中移除1个或多个member，时间复杂度O(N)，N为移除的member个数
- **SRANDMEMBER**：从指定Set中随机返回1个或多个member，时间复杂度O(N)，N为返回的member个数

- **SPOP**：从指定Set中随机移除并返回count个member，时间复杂度O(N)，N为移除的member个数
- **SCARD**：返回指定Set中的member个数，时间复杂度O(1)
- **SISMEMBER**：判断指定的value是否存在于指定Set中，时间复杂度O(1)
- **SMOVE**：将指定member从一个Set移至另一个Set

慎用的Set相关命令：

- **SMEMBERS**：返回指定Hash中所有的member，时间复杂度O(N)
- **SUNION/SUNIONSTORE**：计算多个Set的并集并返回/存储至另一个Set中，时间复杂度O(N)，N为参与计算的所有集合的总member数
- **SINTER/SINTERSTORE**：计算多个Set的交集并返回/存储至另一个Set中，时间复杂度O(N)，N为参与计算的所有集合的总member数
- **SDIFF/SDIFFSTORE**：计算1个Set与1或多个Set的差集并返回/存储至另一个Set中，时间复杂度O(N)，N为参与计算的所有集合的总member数

上述几个命令涉及的计算量大，应谨慎使用，特别是在参与计算的Set尺寸不可知的情况下，应严格避免使用。可以考虑通过SSCAN命令遍历获取相关Set的全部member（具体请见 https://redis.io/commands/scan ），如果需要做并集/交集/差集计算，可以在客户端进行，或在不服务实时查询请求的Slave上进行。

Sorted Set

Redis Sorted Set是有序的、不可重复的String集合。Sorted Set中的每个元素都需要指派一个分数(score)，Sorted Set会根据score对元素进行升序排序。如果多个member拥有相同的score，则以字典序进行升序排序。

Sorted Set非常适合用于实现排名。

Sorted Set的主要命令：

- **ZADD**：向指定Sorted Set中添加1个或多个member，时间复杂度O(Mlog(N))，M为添加的member数量，N为Sorted Set中的member数量
- **ZREM**：从指定Sorted Set中删除1个或多个member，时间复杂度O(Mlog(N))，M为删除的member数量，N为Sorted Set中的member数量
- **ZCOUNT**：返回指定Sorted Set中指定score范围内的member数量，时间复杂度：O(log(N))
- **ZCARD**：返回指定Sorted Set中的member数量，时间复杂度O(1)
- **ZSCORE**：返回指定Sorted Set中指定member的score，时间复杂度O(1)
- **ZRANK/ZREVRANK**：返回指定member在Sorted Set中的排名，ZRANK返回按升序排序的排名，ZREVRANK则返回按降序排序的排名。时间复杂度O(log(N))
- **ZINCRBY**：同INCRBY，对指定Sorted Set中的指定member的score进行自增，时间复杂度O(log(N))

慎用的Sorted Set相关命令：

- **ZRANGE/ZREVRANGE**：返回指定Sorted Set中指定排名范围内的所有member，ZRANGE为按score升序排序，ZREVRANGE为按score降序排序，时间复杂度O(log(N)+M)，M为本次返回的member数
- **ZRANGEBYSCORE/ZREVRANGEBYSCORE**：返回指定Sorted Set中指定score范围内的所有member，返回结果以升序/降序排序，min和max可以指定为-inf和+inf，代表返回所有的member。时间复杂度O(log(N)+M)
- **ZREMRANGEBYRANK/ZREMRANGEBYSCORE**：移除Sorted Set中指定排名范围/指定score范围内的所有member。时间复杂度O(log(N)+M)

上述几个命令，应尽量避免传递[0 -1]或[-inf +inf]这样的参数，来对Sorted Set做一次性的完整遍历，特别是在Sorted Set的尺寸不可预知的情况下。可以通过ZSCAN命令来进行游标式的遍历（具体请见 https://redis.io/commands/scan），或通过LIMIT参数来限制返回member的数量（适用于ZRANGEBYSCORE和ZREVRANGEBYSCORE命令），以实现游标式的遍历。

Bitmap和HyperLogLog

Redis的这两种数据结构相较之前的并不常用，在本文中只做简要介绍，如想要详细了解这两种数据结构与其相关的命令，请参考官方文档 https://redis.io/topics/data-types-intro 中的相关章节

Bitmap在Redis中不是一种实际的数据类型，而是一种将String作为Bitmap使用的方法。可以理解为将String转换为bit数组。使用Bitmap来存储true/false类型的简单数据极为节省空间。

HyperLogLogs是一种主要用于数量统计的数据结构，它和Set类似，维护一个不可重复的String集合，但是HyperLogLogs并不维护具体的

member内容，只维护member的个数。也就是说，HyperLogLogs只能用于计算一个集合中不重复的元素数量，所以它比Set要节省很多内存空间。

其他常用命令

- **EXISTS**：判断指定的key是否存在，返回1代表存在，0代表不存在，时间复杂度O(1)
- **DEL**：删除指定的key及其对应的value，时间复杂度O(N)，N为删除的key数量
- **EXPIRE/PEXPIRE**：为一个key设置有效期，单位为秒或毫秒，时间复杂度O(1)
- **TTL/PTTL**：返回一个key剩余的有效时间，单位为秒或毫秒，时间复杂度O(1)
- **RENAME/RENAMENX**：将key重命名为newkey。使用RENAME时，如果newkey已经存在，其值会被覆盖；使用RENAMENX时，如果newkey已经存在，则不会进行任何操作，时间复杂度O(1)
- **TYPE**：返回指定key的类型，string, list, set, zset, hash。时间复杂度O(1)
- **CONFIG GET**：获得Redis某配置项的当前值，可以使用*通配符，时间复杂度O(1)
- **CONFIG SET**：为Redis某个配置项设置新值，时间复杂度O(1)
- **CONFIG REWRITE**：让Redis重新加载redis.conf中的配置

数据持久化

Redis提供了将数据定期自动持久化至硬盘的能力，包括RDB和AOF两种方案，两种方案分别有其长处和短板，可以配合起来同时运行，确保数据的稳定性。

必须使用数据持久化吗？

Redis的数据持久化机制是可以关闭的。如果你只把Redis作为缓存服务使用，Redis中存储的所有数据都不是该数据的主体而仅仅是同步过来的备份，那么可以关闭Redis的数据持久化机制。

但通常来说，仍然建议至少开启RDB方式的数据持久化，因为：

- RDB方式的持久化几乎不损耗Redis本身的性能，在进行RDB持久化时，Redis主进程唯一需要做的事情就是fork出一个子进程，所有持久化工作都由子进程完成
- Redis无论因为什么原因crash掉之后，重启时能够自动恢复到上一次RDB快照中记录的数据。这省去了手工从其他数据源（如DB）同步数据的过程，而且要比其他任何的数据恢复方式都要快
- 现在硬盘那么大，真的不缺那一点地方

RDB

采用RDB持久方式，Redis会定期保存数据快照至一个rbd文件中，并在启动时自动加载rdb文件，恢复之前保存的数据。可以在配置文件中配置Redis进行快照保存的时机：

```
save [seconds] [changes]
```

意为在[seconds]秒内如果发生了[changes]次数据修改，则进行一次RDB快照保存，例如

```
save 60 100
```

会让Redis每60秒检查一次数据变更情况，如果发生了100次或以上的数据变更，则进行RDB快照保存。

可以配置多条save指令，让Redis执行多级的快照保存策略。

Redis默认开启RDB快照，默认的RDB策略如下：

```
save 900 1
save 300 10
save 60 10000
```

也可以通过**BGSAVE**命令手工触发RDB快照保存。

RDB的优点：

- 对性能影响最小。如前文所述，Redis在保存RDB快照时会fork出子进程进行，几乎不影响Redis处理客户端请求的效率。
- 每次快照会生成一个完整的数据快照文件，所以可以辅以其他手段保存多个时间点的快照（例如把每天0点的快照备份至其他存储媒介中），作为非常可靠的灾难恢复手段。
- 使用RDB文件进行数据恢复比使用AOF要快很多。

RDB的缺点：

- 快照是定期生成的，所以在Redis crash时或多或少会丢失一部分数据。
- 如果数据集非常大且CPU不够强（比如单核CPU），Redis在fork子进程时可能会消耗相对较长的时间（长至1秒），影响这期间的客户端请求。

AOF

采用AOF持久方式时，Redis会把每一个写请求都记录在一个日志文件里。在Redis重启时，会把AOF文件中记录的所有写操作顺序执行一遍，确保数据恢复到最新。

AOF默认是关闭的，如要开启，进行如下配置：

```
appendonly yes
```

AOF提供了三种fsync配置，always/everysec/no，通过配置项[appendfsync]指定：

- appendfsync no：不进行fsync，将flush文件的时机交给OS决定，速度最快
- appendfsync always：每写入一条日志就进行一次fsync操作，数据安全性最高，但速度最慢
- appendfsync everysec：折中的做法，交由后台线程每秒fsync一次

随着AOF不断地记录写操作日志，必定会出现一些无用的日志，例如某个时间点执行了命令**SET key1 "abc"**，在之后某个时间点又执行了**SET key1 "bcd"**，那么第一条命令很显然是没有用的。大量的无用日志会让AOF文件过大，也会让数据恢复的时间过长。
所以Redis提供了AOF rewrite功能，可以重写AOF文件，只保留能够把数据恢复到最新状态的最小写操作集。
AOF rewrite可以通过**BGREWRITEAOF**命令触发，也可以配置Redis定期自动进行：

```
auto-aof-rewrite-percentage 100
auto-aof-rewrite-min-size 64mb
```

上面两行配置的含义是，Redis在每次AOF rewrite时，会记录完成rewrite后的AOF日志大小，当AOF日志大小在该基础上增长了100%后，自动进行AOF rewrite。同时如果增长的大小没有达到64mb，则不会进行rewrite。
AOF的优点：

- 最安全，在启用appendfsync always时，任何已写入的数据都不会丢失，使用在启用appendfsync everysec也至多只会丢失1秒的数据。
- AOF文件在发生断电等问题时也不会损坏，即使出现了某条日志只写入了一半的情况，也可以使用redis-check-aof工具轻松修复。
- AOF文件易读，可修改，在进行了某些错误的数据清除操作后，只要AOF文件没有rewrite，就可以把AOF文件备份出来，把错误的命令删除，然后恢复数据。

AOF的缺点：

- AOF文件通常比RDB文件更大
- 性能消耗比RDB高
- 数据恢复速度比RDB慢

内存管理与数据淘汰机制

最大内存设置

默认情况下，在32位OS中，Redis最大使用3GB的内存，在64位OS中则没有限制。

在使用Redis时，应该对数据占用的最大空间有一个基本准确的预估，并为Redis设定最大使用的内存。否则在64位OS中Redis会无限制地占用内存（当物理内存被占满后会使用swap空间），容易引发各种各样的问题。

通过如下配置控制Redis使用的最大内存：

```
maxmemory 100mb
```

在内存占用达到了maxmemory后，再向Redis写入数据时，Redis会：

- 根据配置的数据淘汰策略尝试淘汰数据，释放空间
- 如果没有数据可以淘汰，或者没有配置数据淘汰策略，那么Redis会对所有写请求返回错误，但读请求仍然可以正常执行

在为Redis设置maxmemory时，需要注意：

- 如果采用了Redis的主从同步，主节点向从节点同步数据时，会占用掉一部分内存空间，如果maxmemory过于接近主机的可用内存，导致数据同步时内存不足。所以设置的maxmemory不要过于接近主机可用的内存，留出一部分预留用作主从同步。

数据淘汰机制

Redis提供了5种数据淘汰策略：

- volatile-lru：使用LRU算法进行数据淘汰（淘汰上次使用时间最早的，且使用次数最少的key），只淘汰设定了有效期的key
- allkeys-lru：使用LRU算法进行数据淘汰，所有的key都可以被淘汰
- volatile-random：随机淘汰数据，只淘汰设定了有效期的key
- allkeys-random：随机淘汰数据，所有的key都可以被淘汰
- volatile-ttl：淘汰剩余有效期最短的key

最好为Redis指定一种有效的数据淘汰策略以配合maxmemory设置，避免在内存使用满后发生写入失败的情况。

一般来说，推荐使用的策略是volatile-lru，并辨识Redis中保存的数据的重要性。对于那些重要的，绝对不能丢弃的数据（如配置类数据等），应不设置有效期，这样Redis就永远不会淘汰这些数据。对于那些相对不是那么重要的，并且能够热加载的数据（比如缓存最近登录的

用户信息，当在Redis中找不到时，程序会去DB中读取），可以设置上有效期，这样在内存不够时Redis就会淘汰这部分数据。

配置方法：

```
maxmemory-policy volatile-lru    #默认是noeviction，即不进行数据淘汰
```

Pipelining

Pipelining

Redis提供许多批量操作的命令，如MSET/MGET/HMSET/HMGET等等，这些命令存在的意义是减少维护网络连接和传输数据所消耗的资源和时间。

例如连续使用5次SET命令设置5个不同的key，比起使用一次MSET命令设置5个不同的key，效果是一样的，但前者会消耗更多的RTT(Round Trip Time)时长，永远应优先使用后者。

然而，如果客户端要连续执行的多次操作无法通过Redis命令组合在一起，例如：

```
SET a "abc"
INCR b
HSET c name "hi"
```

此时便可以使用Redis提供的pipelining功能来实现在一次交互中执行多条命令。

使用pipelining时，只需要从客户端一次向Redis发送多条命令（以\r\n）分隔，Redis就会依次执行这些命令，并且把每个命令的返回按顺序组装在一起一次返回，比如：

```
$ (printf "PING\r\nPING\r\nPING\r\n"; sleep 1) | nc localhost 6379
+PONG
```

```
+PONG
+PONG
```

大部分的Redis客户端都对Pipelining提供支持，所以开发者通常并不需要自己手工拼装命令列表。

Pipelining的局限性

Pipelining只能用于执行**连续且无相关性**的命令，当某个命令的生成需要依赖于前一个命令的返回时，就无法使用Pipelining了。

通过Scripting功能，可以规避这一局限性

事务与Scripting

Pipelining能够让Redis在一次交互中处理多条命令，然而在一些场景下，我们可能需要在此基础上确保这一组命令是连续执行的。

比如获取当前累计的PV数并将其清0

```
> GET vCount
12384
> SET vCount 0
OK
```

如果在GET和SET命令之间插进来一个INCR vCount，就会使客户端拿到的vCount不准确。

Redis的事务可以确保复数命令执行时的原子性。也就是说Redis能够保证：一个事务中的一组命令是绝对连续执行的，在这些命令执行完成之前，绝对不会有来自于其他连接的其他命令插进去执行。

通过MULTI和EXEC命令来把这两个命令加入一个事务中：

```
> MULTI
OK
> GET vCount
QUEUED
```

```
> SET vCount 0
QUEUED
> EXEC
1) 12384
2) OK
```

Redis在接收到MULTI命令后便会开启一个事务，这之后的所有读写命令都会保存在队列中但并不执行，直到接收到EXEC命令后，Redis会把队列中的所有命令连续顺序执行，并以数组形式返回每个命令的返回结果。

可以使用DISCARD命令放弃当前的事务，将保存的命令队列清空。

需要注意的是，**Redis事务不支持回滚**：

如果一个事务中的命令出现了语法错误，大部分客户端驱动会返回错误，2.6.5版本以上的Redis也会在执行EXEC时检查队列中的命令是否存在语法错误，如果存在，则会自动放弃事务并返回错误。

但如果一个事务中的命令有非语法类的错误（比如对String执行HSET操作），无论客户端驱动还是Redis都无法在真正执行这条命令之前发现，所以事务中的所有命令仍然会被依次执行。在这种情况下，会出现一个事务中部分命令成功部分命令失败的情况，然而与RDBMS不同，Redis不提供事务回滚的功能，所以只能通过其他方法进行数据的回滚。

通过事务实现CAS

Redis提供了WATCH命令与事务搭配使用，实现CAS乐观锁的机制。

假设要实现将某个商品的状态改为已售：

```
if(exec(HGET stock:1001 state) == "in stock")
    exec(HSET stock:1001 state "sold");
```

这一伪代码执行时，无法确保并发安全性，有可能多个客户端都获取到了"in stock"的状态，导致一个库存被售卖多次。

使用WATCH命令和事务可以解决这一问题：

```
exec(WATCH stock:1001);
if(exec(HGET stock:1001 state) == "in stock") {
    exec(MULTI);
    exec(HSET stock:1001 state "sold");
    exec(EXEC);
}
```

WATCH的机制是：在事务EXEC命令执行时，Redis会检查被WATCH的key，只有被WATCH的key从WATCH起始时至今没有发生过变更，EXEC才会被执行。如果WATCH的key在WATCH命令到EXEC命令之间发生过变化，则EXEC命令会返回失败。

Scripting

通过EVAL与EVALSHA命令，可以让Redis执行LUA脚本。这就类似于RDBMS的存储过程一样，可以把客户端与Redis之间密集的读/写交互放在服务端进行，避免过多的数据交互，提升性能。

Scripting功能是作为事务功能的替代者诞生的，事务提供的所有能力Scripting都可以做到。Redis官方推荐使用LUA Script来代替事务，前者的效率和便利性都超过了事务。

关于Scripting的具体使用，本文不做详细介绍，请参考官方文档 https://redis.io/commands/eval

Redis性能调优

尽管Redis是一个非常快速的内存数据存储媒介，也并不代表Redis不会产生性能问题。

前文中提到过，Redis采用单线程模型，所有的命令都是由一个线程串行执行的，所以当某个命令执行耗时较长时，会拖慢其后的所有命令，

这使得Redis对每个任务的执行效率更加敏感。
针对Redis的性能优化，主要从下面几个层面入手：

- 最初的也是最重要的，确保没有让Redis执行耗时长的命令
- 使用pipelining将连续执行的命令组合执行
- 操作系统的Transparent huge pages功能必须关闭：

```
echo never > /sys/kernel/mm/transparent_hugepage/enabled
```

- 如果在虚拟机中运行Redis，可能天然就有虚拟机环境带来的固有延迟。可以通过./redis-cli --intrinsic-latency 100命令查看固有延迟。同时如果对Redis的性能有较高要求的话，应尽可能在物理机上直接部署Redis。
- 检查数据持久化策略
- 考虑引入读写分离机制

长耗时命令
Redis绝大多数读写命令的时间复杂度都在O(1)到O(N)之间，在文本和官方文档中均对每个命令的时间复杂度有说明。
通常来说，O(1)的命令是安全的，O(N)命令在使用时需要注意，如果N的数量级不可预知，则应避免使用。例如对一个field数未知的Hash数据执行HGETALL/HKEYS/HVALS命令，通常来说这些命令执行的很快，但如果这个Hash中的field数量极多，耗时就会成倍增长。
又如使用SUNION对两个Set执行Union操作，或使用SORT对List/Set执行排序操作等时，都应该严加注意。
避免在使用这些O(N)命令时发生问题主要有几个办法：

- 不要把List当做列表使用，仅当做队列来使用
- 通过机制严格控制Hash、Set、Sorted Set的大小

- 可能的话，将排序、并集、交集等操作放在客户端执行
- 绝对禁止使用KEYS命令
- 避免一次性遍历集合类型的所有成员，而应使用SCAN类的命令进行分批的，游标式的遍历

Redis提供了SCAN命令，可以对Redis中存储的所有key进行游标式的遍历，避免使用KEYS命令带来的性能问题。同时还有SSCAN/HSCAN/ZSCAN等命令，分别用于对Set/Hash/Sorted Set中的元素进行游标式遍历。SCAN类命令的使用请参考官方文档：https://redis.io/commands/scan

Redis提供了Slow Log功能，可以自动记录耗时较长的命令。相关的配置参数有两个：

```
slowlog-log-slower-than xxxms  #执行时间慢于xxx毫秒的命令计入Slow Log
slowlog-max-len xxx  #Slow Log的长度，即最大纪录多少条Slow Log
```

使用**SLOWLOG GET [number]**命令，可以输出最近进入Slow Log的number条命令。

使用**SLOWLOG RESET**命令，可以重置Slow Log

网络引发的延迟

- 尽可能使用长连接或连接池，避免频繁创建销毁连接
- 客户端进行的批量数据操作，应使用Pipeline特性在一次交互中完成。具体请参照本文的Pipelining章节

数据持久化引发的延迟

Redis的数据持久化工作本身就会带来延迟，需要根据数据的安全级别和性能要求制定合理的持久化策略：

- AOF + fsync always的设置虽然能够绝对确保数据安全，但每个操作都会触发一次fsync，会对Redis的性能有比较明显的影响
- AOF + fsync every second是比较好的折中方案，每秒fsync一次
- AOF + fsync never会提供AOF持久化方案下的最优性能
- 使用RDB持久化通常会提供比使用AOF更高的性能，但需要注意RDB的策略配置
- 每一次RDB快照和AOF Rewrite都需要Redis主进程进行fork操作。fork操作本身可能会产生较高的耗时，与CPU和Redis占用的内存大小有关。根据具体的情况合理配置RDB快照和AOF Rewrite时机，避免过于频繁的fork带来的延迟

Redis在fork子进程时需要将内存分页表拷贝至子进程，以占用了24GB内存的Redis实例为例，共需要拷贝24GB / 4kB * 8 = 48MB的数据。在使用单Xeon 2.27Ghz的物理机上，这一fork操作耗时216ms。

可以通过**INFO**命令返回的latest_fork_usec字段查看上一次fork操作的耗时（微秒）

Swap引发的延迟

当Linux将Redis所用的内存分页移至swap空间时，将会阻塞Redis进程，导致Redis出现不正常的延迟。Swap通常在物理内存不足或一些进程在进行大量I/O操作时发生，应尽可能避免上述两种情况的出现。/proc/<pid>/smaps文件中会保存进程的swap记录，通过查看这个文件，能够判断Redis的延迟是否由Swap产生。如果这个文件中记录了较大的Swap size，则说明延迟很有可能是Swap造成的。

数据淘汰引发的延迟

当同一秒内有大量key过期时，也会引发Redis的延迟。在使用时应尽量将key的失效时间错开。

引入读写分离机制

Redis的主从复制能力可以实现一主多从的多节点架构，在这一架构下，主节点接收所有写请求，并将数据同步给多个从节点。

在这一基础上，我们可以让从节点提供对实时性要求不高的读请求服务，以减小主节点的压力。

尤其是针对一些使用了长耗时命令的统计类任务，完全可以指定在一个或多个从节点上执行，避免这些长耗时命令影响其他请求的响应。

关于读写分离的具体说明，请参见后续章节

主从复制与集群分片

主从复制

Redis支持一主多从的主从复制架构。一个Master实例负责处理所有的写请求，Master将写操作同步至所有Slave。

借助Redis的主从复制，可以实现读写分离和高可用：

- 实时性要求不是特别高的读请求，可以在Slave上完成，提升效率。特别是一些周期性执行的统计任务，这些任务可能需要执行一些长耗时的Redis命令，可以专门规划出1个或几个Slave用于服务这些统计任务
- 借助Redis Sentinel可以实现高可用，当Master crash后，Redis Sentinel能够自动将一个Slave晋升为Master，继续提供服务

启用主从复制非常简单，只需要配置多个Redis实例，在作为Slave的Redis实例中配置：

```
slaveof 192.168.1.1 6379  #指定Master的IP和端口
```

当Slave启动后，会从Master进行一次冷启动数据同步，由Master触发BGSAVE生成RDB文件推送给Slave进行导入，导入完成后Master再将增量数据通过Redis Protocol同步给Slave。之后主从之间的数据便一直以Redis Protocol进行同步

使用Sentinel做自动failover

Redis的主从复制功能本身只是做数据同步，并不提供监控和自动failover能力，要通过主从复制功能来实现Redis的高可用，还需要引入一个组件：Redis Sentinel

Redis Sentinel是Redis官方开发的监控组件，可以监控Redis实例的状态，通过Master节点自动发现Slave节点，并在监测到Master节点失效时选举出一个新的Master，并向所有Redis实例推送新的主从配置。

Redis Sentinel需要至少部署3个实例才能形成选举关系。

关键配置：

```
sentinel monitor mymaster 127.0.0.1 6379 2  #Master实例的IP、端口，以及选举需要的赞成票数
sentinel down-after-milliseconds mymaster 60000  #多长时间没有响应视为Master失效
sentinel failover-timeout mymaster 180000  #两次failover尝试间的间隔时长
sentinel parallel-syncs mymaster 1  #如果有多个Slave，可以通过此配置指定同时从新Master进行数据同步的Slave数，避免所有Slave同时进行数据同步导致查询服务也不可用
```

另外需要注意的是，Redis Sentinel实现的自动failover不是在同一个IP和端口上完成的，也就是说自动failover产生的新Master提供服务的

IP和端口与之前的Master是不一样的，所以要实现HA，还要求客户端必须支持Sentinel，能够与Sentinel交互获得新Master的信息才行。

集群分片

为何要做集群分片：

- Redis中存储的数据量大，一台主机的物理内存已经无法容纳
- Redis的写请求并发量大，一个Redis实例以无法承载

当上述两个问题出现时，就必须要对Redis进行分片了。

Redis的分片方案有很多种，例如很多Redis的客户端都自行实现了分片功能，也有向Twemproxy这样的以代理方式实现的Redis分片方案。然而首选的方案还应该是Redis官方在3.0版本中推出的Redis Cluster分片方案。

本文不会对Redis Cluster的具体安装和部署细节进行介绍，重点介绍Redis Cluster带来的好处与弊端。

Redis Cluster的能力

- 能够自动将数据分散在多个节点上
- 当访问的key不在当前分片上时，能够自动将请求转发至正确的分片
- 当集群中部分节点失效时仍能提供服务

其中第三点是基于主从复制来实现的，Redis Cluster的每个数据分片都采用了主从复制的结构，原理和前文所述的主从复制完全一致，唯一的区别是省去了Redis Sentinel这一额外的组件，由Redis Cluster负责进行一个分片内部的节点监控和自动failover。

Redis Cluster分片原理

Redis Cluster中共有16384个hash slot，Redis会计算每个key的CRC16，将结果与16384取模，来决定该key存储在哪一个hash slot中，同时需要指定Redis Cluster中每个数据分片负责的Slot数。Slot的分配在任何时间点都可以进行重新分配。

客户端在对key进行读写操作时，可以连接Cluster中的任意一个分片，如果操作的key不在此分片负责的Slot范围内，Redis Cluster会自动将请求重定向到正确的分片上。

hash tags

在基础的分片原则上，Redis还支持hash tags功能，以hash tags要求的格式明明的key，将会确保进入同一个Slot中。例如：{uiv}user:1000和{uiv}user:1001拥有同样的hash tag {uiv}，会保存在同一个Slot中。

使用Redis Cluster时，pipelining、事务和LUA Script功能涉及的key必须在同一个数据分片上，否则将会返回错误。如要在Redis Cluster中使用上述功能，就必须通过hash tags来确保一个pipeline或一个事务中操作的所有key都位于同一个Slot中。

> 有一些客户端（如Redisson）实现了集群化的pipelining操作，可以自动将一个pipeline里的命令按key所在的分片进行分组，分别发到不同的分片上执行。但是Redis不支持跨分片的事务，事务和LUA Script还是必须遵循所有key在一个分片上的规则要求。

主从复制 vs 集群分片

在设计软件架构时，要如何在主从复制和集群分片两种部署方案中取舍呢？

从各个方面看，Redis Cluster都是优于主从复制的方案

- Redis Cluster能够解决单节点上数据量过大的问题
- Redis Cluster能够解决单节点访问压力过大的问题

- Redis Cluster包含了主从复制的能力

那是不是代表Redis Cluster永远是优于主从复制的选择呢？
并不是。
软件架构永远不是越复杂越好，复杂的架构在带来显著好处的同时，一定也会带来相应的弊端。采用Redis Cluster的弊端包括：

- 维护难度增加。在使用Redis Cluster时，需要维护的Redis实例数倍增，需要监控的主机数量也相应增加，数据备份/持久化的复杂度也会增加。同时在进行分片的增减操作时，还需要进行reshard操作，远比主从模式下增加一个Slave的复杂度要高。
- 客户端资源消耗增加。当客户端使用连接池时，需要为每一个数据分片维护一个连接池，客户端同时需要保持的连接数成倍增多，加大了客户端本身和操作系统资源的消耗。
- 性能优化难度增加。你可能需要在多个分片上查看Slow Log和Swap日志才能定位性能问题。
- 事务和LUA Script的使用成本增加。在Redis Cluster中使用事务和LUA Script特性有严格的限制条件，事务和Script中操作的key必须位于同一个分片上，这就使得在开发时必须对相应场景下涉及的key进行额外的规划和规范要求。如果应用的场景中大量涉及事务和Script的使用，如何在保证这两个功能的正常运作前提下把数据平均分到多个数据分片中就会成为难点。

所以说，在主从复制和集群分片两个方案中做出选择时，应该从应用软件的功能特性、数据和访问量级、未来发展规划等方面综合考虑，只在**确实有必要**引入数据分片时再使用Redis Cluster。
下面是一些建议：

1. 需要在Redis中存储的数据有多大？未来2年内可能发展为多大？这些数据是否都需要长期保存？是否可以使用LRU算法进行非热点数据的淘汰？综合考虑前面几个因素，评估出Redis需要使用的物理内存。
2. 用于部署Redis的主机物理内存有多大？有多少可以分配给Redis使用？对比(1)中的内存需求评估，是否足够用？
3. Redis面临的并发写压力会有多大？在不使用pipelining时，Redis的写性能可以超过10万次/秒（更多的benchmark可以参考 https://redis.io/topics/benchmarks ）
4. 在使用Redis时，是否会使用到pipelining和事务功能？使用的场景多不多？

综合上面几点考虑，如果单台主机的可用物理内存完全足以支撑对Redis的容量需求，且Redis面临的并发写压力距离Benchmark值还尚有距离，建议采用主从复制的架构，可以省去很多不必要的麻烦。同时，如果应用中大量使用pipelining和事务，也建议尽可能选择主从复制架构，可以减少设计和开发时的复杂度。