# Redis数据类型

## String类型

一、概述：

字符串类型是Redis中最为基础的数据存储类型，它在Redis中是二进制安全的，

这便意味着该类型可以接受任何格式的数据，如JPEG图像数据或Json对象描述信息等。

在Redis中字符串类型的Value最多可以容纳的数据长度是512M。

1. 相关命令列表：

|  |  |  |  |
| --- | --- | --- | --- |
| 命令原型 | 时间复杂度 | 命令描述 | 返回值 |
| APPEND key value | O(1) | 如果该Key已经存在，APPEND命令将参数Value的数据追加到已存在Value的末尾。  如果该Key不存在，APPEND命令将会创建一个新的Key/Value。 | 追加后Value的长度。 |
| DECR key | O(1) | 将指定Key的Value原子性的递减1。如果该Key不存在，其初始值为0，在decr之后其值为-1。  如果Value的值不能转换为整型值，如Hello，该操作将执行失败并返回相应的错误信息。注意：该操作的取值范围是64位有符号整型。 | 递减后的Value值。 |
| INCR key | O(1) | 将指定Key的Value原子性的递增1。如果该Key不存在，其初始值为0，在incr之后其值为1。  如果Value的值不能转换为整型值，如Hello，该操作将执行失败并返回相应的错误信息。注意：该操作的取值范围是64位有符号整型。 | 递增后的Value值。 |
| DECRBY key decrement | O(1) | 将指定Key的Value原子性的减少decrement。如果该Key不存在，其初始值为0，在decrby之后其值为-decrement。  如果Value的值不能转换为整型值，如Hello，该操作将执行失败并返回相应的错误信息。注意：该操作的取值范围是64位有符号整型。 | 减少后的Value值。 |
| INCRBY key increment | O(1) | 将指定Key的Value原子性的增加increment。如果该Key不存在，其初始值为0，在incrby之后其值为increment。  如果Value的值不能转换为整型值，如Hello，该操作将执行失败并返回相应的错误信息。注意：该操作的取值范围是64位有符号整型。 | 增加后的Value值。 |
| GET key | O(1) | 获取指定Key的Value。如果与该Key关联的Value不是string类型，Redis将返回错误信息，因为GET命令只能用于获取string Value。 | 与该Key相关的Value，如果该Key不存在，返回nil。 |
| SET key value | O(1) | 设定该Key持有指定的字符串Value，如果该Key已经存在，则覆盖其原有值。 | 总是返回"OK"。 |
| GETSET key  value | O(1) | 原子性的设置该Key为指定的Value，同时返回该Key的原有值。  和GET命令一样，该命令也只能处理string Value，否则Redis将给出相关的错误信息 | 返回该Key的原有value值，如果该Key之前并不存在，则返回nil。 |
| STRLEN key | O(1) | 返回指定Key的Value字符值长度，如果Value不是string类型，Redis将执行失败并给出相关的错误信息。 | 返回指定Key的Value字符长度，如果该Key不存在，返回0。 |
| SETEX key seconds value | O(1) | 原子性完成两个操作，一是设置该Key的值为指定字符串，同时设置该Key在Redis服务器中的存活时间(秒数)。  该命令主要应用于Redis被当做Cache服务器使用时。 |  |
| SETNX key value | O(1) | 如果指定的Key不存在，则设定该Key持有指定字符串Value，此时其效果等价于SET命令。  相反，如果该Key已经存在，该命令将不做任何操作并返回。 | 1表示设置成功，否则0。 |
| SETRANGE key offset value | O(1) | 替换指定Key的部分字符串值。从offset开始，替换的长度为该命令第三个参数value的字符串长度，  其中如果offset的值大于该Key的原有值Value的字符串长度，Redis将会在Value的后面补齐(offset - strlen(value))数量的0x00，之后再追加新值。  如果该键不存在，该命令会将其原值的长度假设为0，并在其后添补offset个0x00后再追加新值。  鉴于字符串Value的最大长度为512M，因此offset的最大值为536870911。  最后需要注意的是，如果该命令在执行时致使指定Key的原有值长度增加，这将会导致Redis重新分配足够的内存以容纳替换后的全部字符串，  因此就会带来一定的性能折损。 | 修改后的字符串Value长度。 |
| GETRANGE key start end | O(1) | 如果截取的字符串长度很短，我们可以该命令的时间复杂度视为O(1)，否则就是O(N)，这里N表示截取的子字符串长度。  该命令在截取子字符串时，将以闭区间的方式同时包含start(0表示第一个字符)和end所在的字符，  如果end值超过Value的字符长度，该命令将只是截取从start开始之后所有的字符数据。 | 子字符串 |
| SETBIT key offset value | O(1) | 设置在指定Offset上BIT的值，该值只能为1或0，在设定后该命令返回该Offset上原有的BIT值。  如果指定Key不存在，该命令将创建一个新值，并在指定的Offset上设定参数中的BIT值。  如果Offset大于Value的字符长度，Redis将拉长Value值并在指定Offset上设置参数中的BIT值，中间添加的BIT值为0。  最后需要说明的是Offset值必须大于0。 | 在指定Offset上的BIT原有值。 |
| GETBIT key offset | O(1) | 返回在指定Offset上BIT的值，0或1。  如果Offset超过string value的长度，该命令将返回0，所以对于空字符串始终返回0。 | 在指定Offset上的BIT值。 |
| MGET key [key ...] | O(N) | N表示获取Key的数量。返回所有指定Keys的Values，如果其中某个Key不存在，  或者其值不为string类型，该Key的Value将返回nil。 | 返回一组指定Keys的Values的列表。 |
| MSET key value [key value ...] | O(N) | N表示指定Key的数量。该命令原子性的完成参数中所有key/value的设置操作，  其具体行为可以看成是多次迭代执行SET命令。 | 该命令不会失败，始终返回OK。 |
| MSETNX key value [key value ...] | O(N) | N表示指定Key的数量。该命令原子性的完成参数中所有key/value的设置操作，其具体行为可以看成是多次迭代执行SETNX命令。  然而这里需要明确说明的是，如果在这一批Keys中有任意一个Key已经存在了，那么该操作将全部回滚，即所有的修改都不会生效。 | 1表示所有Keys都设置成功，0则表示没有任何Key被修改。 |

## List类型

一、概述：

在Redis中，List类型是按照插入顺序排序的字符串链表。和数据结构中的普通链表一样，我们可以在其头部(left)和尾部(right)添加新的元素。

在插入时，如果该键并不存在，Redis将为该键创建一个新的链表。与此相反，如果链表中所有的元素均被移除，那么该键也将会被从数据库中删除。

List中可以包含的最大元素数量是4294967295。

从元素插入和删除的效率视角来看，如果我们是在链表的两头插入或删除元素，这将会是非常高效的操作，即使链表中已经存储了百万条记录，该操作也可以在常量时间内完成。

然而需要说明的是，如果元素插入或删除操作是作用于链表中间，那将会是非常低效的。相信对于有良好数据结构基础的开发者而言，这一点并不难理解。

二、相关命令列表：

|  |  |  |  |
| --- | --- | --- | --- |
| 命令原型 | 时间复杂度 | 命令描述 | 返回值 |
| LPUSH key value [value ...] | O(1) | 在指定Key所关联的List Value的头部插入参数中给出的所有Values。如果该Key不存在，该命令将在插入之前创建一个与该Key关联的空链表，之后再将数据从链表的头部插入。如果该键的Value不是链表类型，该命令将返回相关的错误信息。 | 插入后链表中元素的数量。 |
| LPUSHX key value | O(1) | 仅有当参数中指定的Key存在时，该命令才会在其所关联的List Value的头部插入参数中给出的Value，否则将不会有任何操作发生。 | 插入后链表中元素的数量。 |
| LRANGE key start stop | O(S+N) | 时间复杂度中的S为start参数表示的偏移量，N表示元素的数量。该命令的参数start和end都是0-based。即0表示链表头部(leftmost)的第一个元素。  其中start的值也可以为负值，-1将表示链表中的最后一个元素，即尾部元素，-2表示倒数第二个并以此类推。该命令在获取元素时，start和end位置上的  元素也会被取出。如果start的值大于链表中元素的数量，空链表将会被返回。如果end的值大于元素的数量，该命令则获取从start(包括start)开始，  链表中剩余的所有元素。 | 返回指定范围内元素的列表。 |
| LPOP key | O(1) | 返回并弹出指定Key关联的链表中的第一个元素，即头部元素。如果该Key不存，返回nil。 | 链表头部的元素。 |
| LLEN key | O(1) | 返回指定Key关联的链表中元素的数量，如果该Key不存在，则返回0。如果与该Key关联的Value的类型不是链表，则返回相关的错误信息 | 链表中元素的数量。 |
| LREM key count value | O(N) | 时间复杂度中N表示链表中元素的数量。在指定Key关联的链表中，删除前count个值等于value的元素。如果count大于0，从头向尾遍历并删除，  如果count小于0，则从尾向头遍历并删除。如果count等于0，则删除链表中所有等于value的元素。如果指定的Key不存在，则直接返回0。 | 返回被删除的元素数量。 |
| LSET key index value | O(N) | 时间复杂度中N表示链表中元素的数量。但是设定头部或尾部的元素时，其时间复杂度为O(1)。设定链表中指定位置的值为新值，其中0表示第一个元素，即头部元素，  -1表示尾部元素。如果索引值Index超出了链表中元素的数量范围，该命令将返回相关的错误信息。 |  |
| LINDEX key index | O(N) | 时间复杂度中N表示在找到该元素时需要遍历的元素数量。对于头部或尾部元素，其时间复杂度为O(1)。该命令将返回链表中指定位置(index)的元素，index是0-based，  表示头部元素，如果index为-1，表示尾部元素。如果与该Key关联的不是链表，该命令将返回相关的错误信息。 | 返回请求的元素，如果index超出范围，则返回nil |
| LTRIM key start stop | O(N) | N表示被删除的元素数量。该命令将仅保留指定范围内的元素，从而保证链接中的元素数量相对恒定。start和stop参数都是0-based，0表示头部元素。和其他命令一样，  start和stop也可以为负值，-1表示尾部元素。如果start大于链表的尾部，或start大于stop，该命令不错报错，而是返回一个空的链表，与此同时该Key也将被删除。  如果stop大于元素的数量，则保留从start开始剩余的所有元素。 |  |
| LINSERT key BEFORE|AFTER pivot value | O(N) | 时间复杂度中N表示在找到该元素pivot之前需要遍历的元素数量。这样意味着如果pivot位于链表的头部或尾部时，该命令的时间复杂度为O(1)。该命令的功能是在  pivot元素的前面或后面插入参数中的元素value。如果Key不存在，该命令将不执行任何操作。如果与Key关联的Value类型不是链表，相关的错误信息将被返回。  成功插入后链表中元素的数量，如果没有找到pivot，返回-1，如果key不存在，返回0。 |  |
| RPUSH key value [value ...] | O(1) | 在指定Key所关联的List Value的尾部插入参数中给出的所有Values。如果该Key不存在，该命令将在插入之前创建一个与该Key关联的空链表，之后再将数据从链表的  尾部插入。如果该键的Value不是链表类型，该命令将返回相关的错误信息。 | 插入后链表中元素的数量。 |
| RPUSHX key value | O(1) | 仅有当参数中指定的Key存在时，该命令才会在其所关联的List Value的尾部插入参数中给出的Value，否则将不会有任何操作发生。 | 插入后链表中元素的数量。 |
| RPOP key | O(1) | 返回并弹出指定Key关联的链表中的最后一个元素，即尾部元素，。如果该Key不存，返回nil。 | 链表尾部的元素。 |
| RPOPLPUSH source destination | O(1) | 原子性的从与source键关联的链表尾部弹出一个元素，同时再将弹出的元素插入到与destination键关联的链表的头部。如果source键不存在，该命令将返回nil，同时  不再做任何其它的操作了。如果source和destination是同一个键，则相当于原子性的将其关联链表中的尾部元素移到该链表的头部。 | 返回弹出和插入的元素。 |

四、链表结构的小技巧：

针对链表结构的Value，Redis在其官方文档中给出了一些实用技巧，如RPOPLPUSH命令，下面给出具体的解释。

Redis链表经常会被用于消息队列的服务，以完成多程序之间的消息交换。假设一个应用程序正在执行LPUSH操作向链表中添加新的元素，我们通常将这样的程序

称之为"生产者(Producer)"，而另外一个应用程序正在执行RPOP操作从链表中取出元素，我们称这样的程序为"消费者(Consumer)"。如果此时，

消费者程序在取出消息元素后立刻崩溃，由于该消息已经被取出且没有被正常处理，那么我们就可以认为该消息已经丢失，由此可能会导致业务数据丢失，

或业务状态的不一致等现象的发生。然而通过使用RPOPLPUSH命令，消费者程序在从主消息队列中取出消息之后再将其插入到备份队列中，

直到消费者程序完成正常的处理逻辑后再将该消息从备份队列中删除。同时我们还可以提供一个守护进程，当发现备份队列中的消息过期时，

可以重新将其再放回到主消息队列中，以便其它的消费者程序继续处理。

## Hash类型

一、概述：

我们可以将Redis中的Hashes类型看成具有String Key和String Value的map容器。所以该类型非常适合于存储值对象的信息。如Username、Password和Age等。

如果Hash中包含很少的字段，那么该类型的数据也将仅占用很少的磁盘空间。每一个Hash可以存储4294967295个键值对。

二、相关命令列表：

|  |  |  |  |
| --- | --- | --- | --- |
| 命令原型 | 时间复杂度 | 命令描述 | 返回值 |
| HSET key field value | O(1) | 为指定的Key设定Field/Value对，如果Key不存在，该命令将创建新Key以参数中的Field/Value对，如果参数中的Field在该Key中已经存在，则用新值覆盖其原有值。 | 1表示新的Field被设置了新值，0表示Field已经存在，用新值覆盖原有值。 |
| HGET key field | O(1) | 返回指定Key中指定Field的关联值。 返回参数中Field的关联值，如果参数中的Key或Field不存，返回nil。 |  |
| HEXISTS key field | O(1) | 判断指定Key中的指定Field是否存在。 1表示存在，0表示参数中的Field或Key不存在。 |  |
| HLEN key | O(1) | 获取该Key所包含的Field的数量。 返回Key包含的Field数量，如果Key不存在，返回0。 |  |
| HDEL key field [field ...] | O(N) | 时间复杂度中的N表示参数中待删除的字段数量。从指定Key的Hashes Value中删除参数中指定的多个字段，如果不存在的字段将被忽略。如果Key不存在，  则将其视为空Hashes，并返回0. | 实际删除的Field数量。 |
| HSETNX key field value | O(1) | 只有当参数中的Key或Field不存在的情况下，为指定的Key设定Field/Value对，否则该命令不会进行任何操作。 | 1表示新的Field被设置了新值，0表示Key或Field已经存在，该命令没有进行任何操作。 |
| HINCRBY key field increment | O(1) | 增加指定Key中指定Field关联的Value的值。如果Key或Field不存在，该命令将会创建一个新Key或新Field，并将其关联的Value初始化为0，  之后再指定数字增加的操作。该命令支持的数字是64位有符号整型，即increment可以负数。 | 返回运算后的值。 |
| HGETALL key | O(N) | 时间复杂度中的N表示Key包含的Field数量。获取该键包含的所有Field/Value。其返回格式为一个Field、一个Value，并以此类推。 | Field/Value的列表。 |
| HKEYS key | O(N) | 时间复杂度中的N表示Key包含的Field数量。返回指定Key的所有Fields名。 | Field的列表。 |
| HVALS key | O(N) | 时间复杂度中的N表示Key包含的Field数量。返回指定Key的所有Values名。 | Value的列表。 |
| HMGET key field [field ...] | O(N) | 时间复杂度中的N表示请求的Field数量。获取和参数中指定Fields关联的一组Values。如果请求的Field不存在，其值返回nil。如果Key不存在，  该命令将其视为空Hash，因此返回一组nil | 返回和请求Fields关联的一组Values，其返回顺序等同于Fields的请求顺序。 |
| HMSET key field value [field value ...] | O(N) | 时间复杂度中的N表示被设置的Field数量。逐对依次设置参数中给出的Field/Value对。如果其中某个Field已经存在，则用新值覆盖原有值。 | 如果Key不存在，则创建新Key，同时设定参数中的Field/Value。 |

## Set类型

一、概述：

在Redis中，我们可以将Set类型看作为没有排序的字符集合，和List类型一样，我们也可以在该类型的数据值上执行添加、删除或判断某一元素是否存在等操作。

需要说明的是，这些操作的时间复杂度为O(1)，即常量时间内完成次操作。Set可包含的最大元素数量是4294967295。

和List类型不同的是，Set集合中不允许出现重复的元素，这一点和C++标准库中的set容器是完全相同的。换句话说，如果多次添加相同元素，Set中将仅保留该元素的

一份拷贝。和List类型相比，Set类型在功能上还存在着一个非常重要的特性，即在服务器端完成多个Sets之间的聚合计算操作，如unions、intersections和differences。

由于这些操作均在服务端完成，因此效率极高，而且也节省了大量的网络IO开销。

二、相关命令列表：

|  |  |  |  |
| --- | --- | --- | --- |
| 命令原型 | 时间复杂度 | 命令描述 | 返回值 |
| SADD key member [member ...] | O(N) | 时间复杂度中的N表示操作的成员数量。如果在插入的过程用，参数中有的成员在Set中已经存在，该成员将被忽略，而其它成员仍将会被正常插入。如果执行该命令之前，  该Key并不存在，该命令将会创建一个新的Set，此后再将参数中的成员陆续插入。如果该Key的Value不是Set类型，该命令将返回相关的错误信息。 | 本次操作实际插入的成员数量。 |
| SCARD key | O(1) | 获取Set中成员的数量。 | 返回Set中成员的数量，如果该Key并不存在，返回0。 |
| SISMEMBER key member | O(1) | 判断参数中指定成员是否已经存在于与Key相关联的Set集合中。 | 1表示已经存在，0表示不存在，或该Key本身并不存在。 |
| SMEMBERS key | O(N) | 时间复杂度中的N表示Set中已经存在的成员数量。获取与该Key关联的Set中所有的成员。 | 返回Set中所有的成员。 |
| SPOP key | O(1) | 随机的移除并返回Set中的某一成员。 由于Set中元素的布局不受外部控制，因此无法像List那样确定哪个元素位于Set的头部或者尾部。 | 返回移除的成员，如果该Key并不存在，则返回nil。 |
| SREM key member [member ...] | O(N) | 时间复杂度中的N表示被删除的成员数量。从与Key关联的Set中删除参数中指定的成员，不存在的参数成员将被忽略，如果该Key并不存在，将视为空Set处理。 | 从Set中实际移除的成员数量，如果没有则返回0。 |
| SRANDMEMBER key | O(1) | 和SPOP一样，随机的返回Set中的一个成员，不同的是该命令并不会删除返回的成员。 返回随机位置的成员，如果Key不存在则返回nil。 |  |
| SMOVE source destination member | O(1) | 原子性的将参数中的成员从source键移入到destination键所关联的Set中。因此在某一时刻，该成员或者出现在source中，或者出现在destination中。  如果该成员在source中并不存在，该命令将不会再执行任何操作并返回0，否则，该成员将从source移入到destination。如果此时该成员已经在destination中存在，  那么该命令仅是将该成员从source中移出。如果和Key关联的Value不是Set，将返回相关的错误信息。 | 1表示正常移动，0表示source中并不包含参数成员。 |
| SDIFF key [key ...] | O(N) | 时间复杂度中的N表示所有Sets中成员的总数量。返回参数中第一个Key所关联的Set和其后所有Keys所关联的Sets中成员的差异。如果Key不存在，则视为空Set。 | 差异结果成员的集合。 |
| SDIFFSTORE destination key [key ...] | O(N) | 该命令和SDIFF命令在功能上完全相同，两者之间唯一的差别是SDIFF返回差异的结果成员，而该命令将差异成员存储在destination关联的Set中。  如果destination键已经存在，该操作将覆盖它的成员。 | 返回差异成员的数量。 |
| SINTER key [key ...] | O(N\*M) | 时间复杂度中的N表示最小Set中元素的数量，M则表示参数中Sets的数量。该命令将返回参数中所有Keys关联的Sets中成员的交集。  因此如果参数中任何一个Key关联的Set为空，或某一Key不存在，那么该命令的结果将为空集。 | 交集结果成员的集合。 |
| SINTERSTORE destination key [key ...] | O(N\*M) | 该命令和SINTER命令在功能上完全相同，两者之间唯一的差别是SINTER返回交集的结果成员，而该命令将交集成员存储在destination关联的Set中。 | 如果destination键已经存在，该操作将覆盖它的成员。返回交集成员的数量。 |
| SUNION key [key ...] | O(N) | 时间复杂度中的N表示所有Sets中成员的总数量。该命令将返回参数中所有Keys关联的Sets中成员的并集。 | 并集结果成员的集合。 |
| SUNIONSTORE destination key [key ...] | O(N) | 该命令和SUNION命令在功能上完全相同，两者之间唯一的差别是SUNION返回并集的结果成员，而该命令将并集成员存储在destination关联的Set中。  如果destination键已经存在，该操作将覆盖它的成员。 | 返回并集成员的数量。 |

四、应用范围：

1). 可以使用Redis的Set数据类型跟踪一些唯一性数据，比如访问某一博客的唯一IP地址信息。对于此场景，我们仅需在每次访问该博客时将访问者的IP存入Redis中，

Set数据类型会自动保证IP地址的唯一性。

2). 充分利用Set类型的服务端聚合操作方便、高效的特性，可以用于维护数据对象之间的关联关系。比如所有购买某一电子设备的客户ID被存储在一个指定的Set中，

而购买另外一种电子产品的客户ID被存储在另外一个Set中，如果此时我们想获取有哪些客户同时购买了这两种商品时，Set的intersections命令就可以充分发挥

它的方便和效率的优势了。

## Sorted-Set类型

一、概述：

Sorted-Sets和Sets类型极为相似，它们都是字符串的集合，都不允许重复的成员出现在一个Set中。它们之间的主要差别是Sorted-Sets中的每一个成员都会有一个

分数(score)与之关联，Redis正是通过分数来为集合中的成员进行从小到大的排序。然而需要额外指出的是，尽管Sorted-Sets中的成员必须是唯一的，但是分数(score)却是可以重复的。

在Sorted-Set中添加、删除或更新一个成员都是非常快速的操作，其时间复杂度为集合中成员数量的对数。由于Sorted-Sets中的成员在集合中的位置是有序的，

因此，即便是访问位于集合中部的成员也仍然是非常高效的。事实上，Redis所具有的这一特征在很多其它类型的数据库中是很难实现的，换句话说，

在该点上要想达到和Redis同样的高效，在其它数据库中进行建模是非常困难的。

二、相关命令列表：

|  |  |  |  |
| --- | --- | --- | --- |
| 命令原型 | 时间复杂度 | 命令描述 | 返回值 |
| ZADD key score member [score] [member] | O(log(N)) | 时间复杂度中的N表示Sorted-Sets中成员的数量。添加参数中指定的所有成员及其分数到指定key的Sorted-Set中，在该命令中我们可以指定多组score/member作为参数。  如果在添加时参数中的某一成员已经存在，该命令将更新此成员的分数为新值，同时再将该成员基于新值重新排序。如果键不存在，该命令将为该键创建一个新的Sorted-Sets Value，并将score/member对插入其中。如果该键已经存在，但是与其关联的Value不是Sorted-Sets类型，相关的错误信息将被返回。 | 本次操作实际插入的成员数量。 |
| ZCARD key | O(1) | 获取与该Key相关联的Sorted-Sets中包含的成员数量。 | 返回Sorted-Sets中的成员数量，如果该Key不存在，返回0。 |
| ZCOUNT key min max | O(log(N)+M) | 该命令用于获取分数(score)在min和max之间的成员数量。针对min和max参数需要额外说明的是，-inf和+inf分别表示Sorted-Sets中分数的最高值和最低值。  缺省情况下，min和max表示的范围是闭区间范围，即min <= score <= max内的成员将被返回。然而我们可以通过在min和max的前面添加"("字符来表示开区间，如(min max表示min < score <= max，而(min (max表示min < score < max | 分数指定范围内成员的数量。 |
| ZINCRBY key increment member | O(log(N)) | 该命令将为指定Key中的指定成员增加指定的分数。如果成员不存在，该命令将添加该成员并假设其初始分数为0，此后再将其分数加上increment。如果Key不存，  该命令将创建该Key及其关联的Sorted-Sets，并包含参数指定的成员，其分数为increment参数。如果与该Key关联的不是Sorted-Sets类型，相关的错误信息将被返回。 | 以字符串形式表示的新分数。 |
| ZRANGE key start stop [WITHSCORES] | O(log(N)+M) | 时间复杂度中的N表示Sorted-Set中成员的数量，M则表示返回的成员数量。该命令返回顺序在参数start和stop指定范围内的成员，这里start和stop参数都是0-based，即0表示第一个成员，-1表示最后一个成员。如果start大于该Sorted-Set中的最大索引值，或start > stop，此时一个空集合将被返回。如果stop大于最大索引值，该命令将返回从start到集合的最后一个成员。如果命令中带有可选参数WITHSCORES选项，该命令在返回的结果中将包含每个成员的分数值，如value1,score1,value2,score2...。 | 返回索引在start和stop之间的成员列表。 |
| ZRANGEBYSCORE key min max [WITHSCORES] [LIMIT offset count] | O(log(N)+M) | 该命令将返回分数在min和max之间的所有成员，即满足表达式min <= score <= max的成员，其中返回的成员是按照其分数从低到高的顺序返回，如果成员具有相同的分数，则按成员的字典顺序返回。可选参数LIMIT用于限制返回成员的数量范围。可选参数offset表示从符合条件的第offset个成员开始返回，同时返回count个成员。可选参数WITHSCORES的含义参照ZRANGE中该选项的说明。最后需要说明的是参数中min和max的规则可参照命令ZCOUNT。 | 返回分数在指定范围内的成员列表。 |
| ZRANK key member | O(log(N)) | 时间复杂度中的N表示Sorted-Set中成员的数量。Sorted-Set中的成员都是按照分数从低到高的顺序存储，该命令将返回参数中指定成员的位置值，其中0表示第一个成员，它是Sorted-Set中分数最低的成员。 如果该成员存在，则返回它的位置索引值。否则返回nil。 |  |
| ZREM key member [member ...] | O(M log(N)) | 时间复杂度中N表示Sorted-Set中成员的数量，M则表示被删除的成员数量。该命令将移除参数中指定的成员，其中不存在的成员将被忽略。如果与该Key关联的Value  不是Sorted-Set，相应的错误信息将被返回。 | 实际被删除的成员数量。 |
| ZREVRANGE key start stop [WITHSCORES] | O(log(N)+M) | 时间复杂度中的N表示Sorted-Set中成员的数量，M则表示返回的成员数量。该命令的功能和ZRANGE基本相同，唯一的差别在于该命令是通过反向排序获取指定位置的成员，即从高到低的顺序。如果成员具有相同的分数，则按降序字典顺序排序。 | 返回指定的成员列表。 |
| ZREVRANK key member | O(log(N)) | 时间复杂度中的N表示Sorted-Set中成员的数量。该命令的功能和ZRANK基本相同，唯一的差别在于该命令获取的索引是从高到低排序后的位置，同样0表示第一个元素，即分数最高的成员。 | 如果该成员存在，则返回它的位置索引值。否则返回nil。 |
| ZSCORE key member | O(1) | 获取指定Key的指定成员的分数。 如果该成员存在，以字符串的形式返回其分数，否则返回nil。 |  |
| ZREVRANGEBYSCORE key max min [WITHSCORES] [LIMIT offset count] | O(log(N)+M) | 时间复杂度中的N表示Sorted-Set中成员的数量，M则表示返回的成员数量。该命令除了排序方式是基于从高到低的分数排序之外，其它功能和参数含义均与  ZRANGEBYSCORE相同。 | 返回分数在指定范围内的成员列表。 |
| ZREMRANGEBYRANK key start stop | O(log(N)+M) | 时间复杂度中的N表示Sorted-Set中成员的数量，M则表示被删除的成员数量。删除索引位置位于start和stop之间的成员，start和stop都是0-based，  即0表示分数最低的成员，-1表示最后一个成员，即分数最高的成员。 | 被删除的成员数量。 |
| ZREMRANGEBYSCORE key min max | O(log(N)+M) | 时间复杂度中的N表示Sorted-Set中成员的数量，M则表示被删除的成员数量。删除分数在min和max之间的所有成员，即满足表达式min <= score <= max的所有成员。对于min和max参数，可以采用开区间的方式表示，具体规则参照ZCOUNT。 | 被删除的成员数量。 |

四、应用范围：

1). 可以用于一个大型在线游戏的积分排行榜。每当玩家的分数发生变化时，可以执行ZADD命令更新玩家的分数，此后再通过ZRANGE命令获取积分TOPTEN的用户信息。

当然我们也可以利用ZRANK命令通过username来获取玩家的排行信息。最后我们将组合使用ZRANGE和ZRANK命令快速的获取和某个玩家积分相近的其他用户的信息。

1. . Sorted-Sets类型还可用于构建索引数据。

# Key操作

一、概述：

在该系列的前几篇博客中，主要讲述的是与Redis数据类型相关的命令，如String、List、Set、Hashes和Sorted-Set。这些命令都具有一个共同点，

即所有的操作都是针对与Key关联的Value的。而该篇博客将主要讲述与Key相关的Redis命令。学习这些命令对于学习Redis是非常重要的基础，也是能够充分挖掘Redis潜力的利器。

二、相关命令列表：

|  |  |  |  |
| --- | --- | --- | --- |
| 命令原型 | 时间复杂度 | 命令描述 | 返回值 |
| KEYS pattern | O(N) | 时间复杂度中的N表示数据库中Key的数量。获取所有匹配pattern参数的Keys。需要说明的是，在我们的正常操作中应该尽量避免对该命令的调用，  因为对于大型数据库而言，该命令是非常耗时的，对Redis服务器的性能打击也是比较大的。pattern支持glob-style的通配符格式，如\*表示任意一个或多个字符，?表示任意字符，[abc]表示方括号中任意一个字母。 | 匹配模式的键列表。 |
| DEL key [key ...] | O(N) | 时间复杂度中的N表示删除的Key数量。从数据库删除中参数中指定的keys，如果指定键不存在，则直接忽略。还需要另行指出的是，如果指定的Key关联的  数据类型不是String类型，而是List、Set、Hashes和Sorted Set等容器类型，该命令删除每个键的时间复杂度为O(M)，其中M表示容器中元素的数量。  而对于String类型的Key，其时间复杂度为O(1) | 实际被删除的Key数量。 |
| EXISTS key | O(1) | 判断指定键是否存在。 | 1表示存在，0表示不在。 |
| MOVE key db | O(1) | 将当前数据库中指定的键Key移动到参数中指定的数据库中。如果该Key在目标数据库中已经存在，或者在当前数据库中并不存在，该命令将不做任何操作并返回0。 | 移动成功返回1，否则0。 |
| RENAME key newkey | O(1) | 为指定指定的键重新命名，如果参数中的两个Keys的命令相同，或者是源Key不存在，该命令都会返回相关的错误信息。 | 如果newKey已经存在，则直接覆盖。 |
| RENAMENX key newkey | O(1) | 如果新值不存在，则将参数中的原值修改为新值。  其它条件和RENAME一致。 | 1表示修改成功，否则0。 |
| PERSIST key | O(1) | 如果Key存在过期时间，该命令会将其过期时间消除，使该Key不再有超时，而是可以持久化存储 | 1表示Key的过期时间被移出，0表示该Key不存在或没有过期时间。 |
| EXPIRE key seconds | O(1) | 该命令为参数中指定的Key设定超时的秒数，在超过该时间后，Key被自动的删除。如果该Key在超时之前被修改，与该键关联的超时将被移除。 | 1表示超时被设置，0则表示Key不存在，或不能被设置。 |
| EXPIREAT key timestamp | O(1) | 该命令的逻辑功能和EXPIRE完全相同，唯一的差别是该命令指定的超时时间是绝对时间，而不是相对时间。该时间参数是Unix timestamp格式的，  即从1970年1月1日开始所流经的秒数。 | 1表示超时被设置，0则表示Key不存在，或不能被设置。 |
| TTL key | O(1) | 获取该键所剩的超时描述。 返回所剩描述，如果该键不存在或没有超时设置，则返回-1。 |  |
| RANDOMKEY | O(1) | 从当前打开的数据库中随机的返回一个Key。 返回的随机键，如果该数据库是空的则返回nil。 |  |
| TYPE key | O(1) | 获取与参数中指定键关联值的类型，该命令将以字符串的格式返回。 返回的字符串为string、list、set、hash和zset，如果key不存在返回none。 |  |

# 事务

一、概述：

和众多其它数据库一样，Redis作为NoSQL数据库也同样提供了事务机制。在Redis中，MULTI/EXEC/DISCARD/WATCH这四个命令是我们实现事务的基石。

相信对有关系型数据库开发经验的开发者而言这一概念并不陌生，即便如此，我们还是会简要的列出Redis中事务的实现特征：

1). 在事务中的所有命令都将会被串行化的顺序执行，事务执行期间，Redis不会再为其它客户端的请求提供任何服务，从而保证了事物中的所有命令被原子的执行。

2). 和关系型数据库中的事务相比，在Redis事务中如果有某一条命令执行失败，其后的命令仍然会被继续执行。

3). 我们可以通过MULTI命令开启一个事务，有关系型数据库开发经验的人可以将其理解为"BEGIN TRANSACTION"语句。在该语句之后执行的命令都将被视为事务之内的操作，最后我们可以通过执行EXEC/DISCARD命令来提交/回滚该事务内的所有操作。这两个Redis命令可被视为等同于关系型数据库中的COMMIT/ROLLBACK语句。

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

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

二、相关命令列表：

|  |  |  |  |
| --- | --- | --- | --- |
| 命令原型 | 时间复杂度 | 命令描述 | 返回值 |
| MULTI |  | 用于标记事务的开始，其后执行的命令都将被存入命令队列，直到执行EXEC时，这些命令才会被原子的执行。 | 始终返回OK |
| EXEC |  | 执行在一个事务内命令队列中的所有命令，同时将当前连接的状态恢复为正常状态，即非事务状态。如果在事务中执行了WATCH命令，那么只有当WATCH所监控的Keys没有被修改的前提下，EXEC命令才能执行事务队列中的所有命令，否则EXEC将放弃当前事务中的所有命令。原子性的返回事务中各条命令的返回结果。如果在事务中使用了WATCH，一旦事务被放弃，EXEC将返回NULL-multi-bulk回复 |  |
| DISCARD |  | 回滚事务队列中的所有命令，同时再将当前连接的状态恢复为正常状态，即非事务状态。如果WATCH命令被使用，该命令将UNWATCH所有的Keys。 | 始终返回OK。 |
| WATCH key [key ...] | O(1) | 在MULTI命令执行之前，可以指定待监控的Keys，然而在执行EXEC之前，如果被监控的Keys发生修改，EXEC将放弃执行该事务队列中的所有命令。 | 始终返回OK。 |
| UNWATCH | O(1) | 取消当前事务中指定监控的Keys，如果执行了EXEC或DISCARD命令，则无需再手工执行该命令了，因为在此之后，事务中所有被监控的Keys都将自动取消。 | 始终返回OK |

四、WATCH命令和基于CAS的乐观锁：

在Redis的事务中，WATCH命令可用于提供CAS(check-and-set)功能。假设我们通过WATCH命令在事务执行之前监控了多个Keys，倘若在WATCH之后有任何Key的值发生了变化，

EXEC命令执行的事务都将被放弃，同时返回Null multi-bulk应答以通知调用者事务执行失败。例如，我们再次假设Redis中并未提供incr命令来完成键值的原子性递增，

如果要实现该功能，我们只能自行编写相应的代码。其伪码如下：

val = GET mykey

val = val + 1

SET mykey $val

以上代码只有在单连接的情况下才可以保证执行结果是正确的，因为如果在同一时刻有多个客户端在同时执行该段代码，那么就会出现多线程程序中经常出现的

一种错误场景--竞态争用(race condition)。比如，客户端A和B都在同一时刻读取了mykey的原有值，假设该值为10，此后两个客户端又均将该值加一后set回Redis服务器，

这样就会导致mykey的结果为11，而不是我们认为的12。为了解决类似的问题，我们需要借助WATCH命令的帮助，见如下代码：

WATCH mykey

val = GET mykey

val = val + 1

MULTI

SET mykey $val

EXEC

和此前代码不同的是，新代码在获取mykey的值之前先通过WATCH命令监控了该键，此后又将set命令包围在事务中，这样就可以有效的保证每个连接在执行EXEC之前，

如果当前连接获取的mykey的值被其它连接的客户端修改，那么当前连接的EXEC命令将执行失败。这样调用者在判断返回值后就可以获悉val是否被重新设置成功。

# 主从复制

一、Redis的Replication：

下面的列表清楚的解释了Redis Replication的特点和优势。

1). 同一个Master可以同步多个Slaves。

2). Slave同样可以接受其它Slaves的连接和同步请求，这样可以有效的分载Master的同步压力。因此我们可以将Redis的Replication架构视为图结构。

3). Master Server是以非阻塞的方式为Slaves提供服务。所以在Master-Slave同步期间，客户端仍然可以提交查询或修改请求。

4). Slave Server同样是以非阻塞的方式完成数据同步。在同步期间，如果有客户端提交查询请求，Redis则返回同步之前的数据。

5). 为了分载Master的读操作压力，Slave服务器可以为客户端提供只读操作的服务，写服务仍然必须由Master来完成。即便如此，系统的伸缩性还是得到了很大的提高。

6). Master可以将数据保存操作交给Slaves完成，从而避免了在Master中要有独立的进程来完成此操作。

二、Replication的工作原理：

在Slave启动并连接到Master之后，它将主动发送一个SYNC命令。此后Master将启动后台存盘进程，同时收集所有接收到的用于修改数据集的命令，在后台进程执行完毕后，Master将传送整个数据库文件到Slave，以完成一次完全同步。而Slave服务器在接收到数据库文件数据之后将其存盘并加载到内存中。此后，Master继续将所有已经收集到的修改命令，和新的修改命令依次传送给Slaves，Slave将在本次执行这些数据修改命令，从而达到最终的数据同步。

如果Master和Slave之间的链接出现断连现象，Slave可以自动重连Master，但是在连接成功之后，一次完全同步将被自动执行。

三、如何配置Replication：

见如下步骤：

1). 同时启动两个Redis服务器，可以考虑在同一台机器上启动两个Redis服务器，分别监听不同的端口，如6379和6380。

2). 在Slave服务器上执行一下命令：

/> redis-cli -p 6380 #这里我们假设Slave的端口号是6380

redis 127.0.0.1:6380> slaveof 127.0.0.1 6379 #我们假设Master和Slave在同一台主机，Master的端口为6379

OK

上面的方式只是保证了在执行slaveof命令之后，redis\_6380成为了redis\_6379的slave，一旦服务(redis\_6380)重新启动之后，他们之间的复制关系将终止。

如果希望长期保证这两个服务器之间的Replication关系，可以在redis\_6380的配置文件中做如下修改：

/> cd /etc/redis #切换Redis服务器配置文件所在的目录。

/> ls

6379.conf 6380.conf

/> vi 6380.conf

将

# slaveof <masterip> <masterport>

改为

slaveof 127.0.0.1 6379

保存退出。

这样就可以保证Redis\_6380服务程序在每次启动后都会主动建立与Redis\_6379的Replication连接了。

# 持久化

一、Redis提供了哪些持久化机制：

1). RDB持久化：

该机制是指在指定的时间间隔内将内存中的数据集快照写入磁盘。

2). AOF持久化:

该机制将以日志的形式记录服务器所处理的每一个写操作，在Redis服务器启动之初会读取该文件来重新构建数据库，以保证启动后数据库中的数据是完整的。

3). 无持久化：

我们可以通过配置的方式禁用Redis服务器的持久化功能，这样我们就可以将Redis视为一个功能加强版的memcached了。

4). 同时应用AOF和RDB。

二、RDB机制的优势和劣势：

RDB存在哪些优势呢？

1). 一旦采用该方式，那么你的整个Redis数据库将只包含一个文件，这对于文件备份而言是非常完美的。比如，你可能打算每个小时归档一次最近24小时的数据，同时还要每天归档一次最近30天的数据。通过这样的备份策略，一旦系统出现灾难性故障，我们可以非常容易的进行恢复。

2). 对于灾难恢复而言，RDB是非常不错的选择。因为我们可以非常轻松的将一个单独的文件压缩后再转移到其它存储介质上。

3). 性能最大化。对于Redis的服务进程而言，在开始持久化时，它唯一需要做的只是fork出子进程，之后再由子进程完成这些持久化的工作，这样就可以极大的避免服务进程执行IO操作了。

4). 相比于AOF机制，如果数据集很大，RDB的启动效率会更高。

RDB又存在哪些劣势呢？

1). 如果你想保证数据的高可用性，即最大限度的避免数据丢失，那么RDB将不是一个很好的选择。因为系统一旦在定时持久化之前出现宕机现象，此前没有来得及写入磁盘的数据都将丢失。

2). 由于RDB是通过fork子进程来协助完成数据持久化工作的，因此，如果当数据集较大时，可能会导致整个服务器停止服务几百毫秒，甚至是1秒钟。

三、AOF机制的优势和劣势：

AOF的优势有哪些呢？

1). 该机制可以带来更高的数据安全性，即数据持久性。Redis中提供了3中同步策略，即每秒同步、每修改同步和不同步。事实上，每秒同步也是异步完成的，其效率也是非常高的，所差的是一旦系统出现宕机现象，那么这一秒钟之内修改的数据将会丢失。而每修改同步，我们可以将其视为同步持久化，即每次发生的数据变化都会被立即记录到磁盘中。可以预见，这种方式在效率上是最低的。至于无同步，无需多言，我想大家都能正确的理解它。

2). 由于该机制对日志文件的写入操作采用的是append模式，因此在写入过程中即使出现宕机现象，也不会破坏日志文件中已经存在的内容。然而如果我们本次操作只是写入了一半数据就出现了系统崩溃问题，不用担心，在Redis下一次启动之前，我们可以通过redis-check-aof工具来帮助我们解决数据一致性的问题。

3). 如果日志过大，Redis可以自动启用rewrite机制。即Redis以append模式不断的将修改数据写入到老的磁盘文件中，同时Redis还会创建一个新的文件用于记录此期间有哪些修改命令被执行。因此在进行rewrite切换时可以更好的保证数据安全性。

4). AOF包含一个格式清晰、易于理解的日志文件用于记录所有的修改操作。事实上，我们也可以通过该文件完成数据的重建。

AOF的劣势有哪些呢？

1). 对于相同数量的数据集而言，AOF文件通常要大于RDB文件。

2). 根据同步策略的不同，AOF在运行效率上往往会慢于RDB。总之，每秒同步策略的效率是比较高的，同步禁用策略的效率和RDB一样高效。

四、其它：

1. Snapshotting:

缺省情况下，Redis会将数据集的快照dump到dump.rdb文件中。此外，我们也可以通过配置文件来修改Redis服务器dump快照的频率，在打开6379.conf文件之后，我们搜索save，可以看到下面的配置信息：

save 900 1 #在900秒(15分钟)之后，如果至少有1个key发生变化，则dump内存快照。

save 300 10 #在300秒(5分钟)之后，如果至少有10个key发生变化，则dump内存快照。

save 60 10000 #在60秒(1分钟)之后，如果至少有10000个key发生变化，则dump内存快照。

2. Dump快照的机制：

1). Redis先fork子进程。

2). 子进程将快照数据写入到临时RDB文件中。

3). 当子进程完成数据写入操作后，再用临时文件替换老的文件。

3. AOF文件：

上面已经多次讲过，RDB的快照定时dump机制无法保证很好的数据持久性。如果我们的应用确实非常关注此点，我们可以考虑使用Redis中的AOF机制。对于Redis服务器而言，其缺省的机制是RDB，如果需要使用AOF，则需要修改配置文件中的以下条目：

将appendonly no改为appendonly yes

从现在起，Redis在每一次接收到数据修改的命令之后，都会将其追加到AOF文件中。在Redis下一次重新启动时，需要加载AOF文件中的信息来构建最新的数据到内存中。

4. AOF的配置：

在Redis的配置文件中存在三种同步方式，它们分别是：

appendfsync always #每次有数据修改发生时都会写入AOF文件。

appendfsync everysec #每秒钟同步一次，该策略为AOF的缺省策略。

appendfsync no #从不同步。高效但是数据不会被持久化。

5. 如何修复坏损的AOF文件：

1). 将现有已经坏损的AOF文件额外拷贝出来一份。

2). 执行"redis-check-aof --fix <filename>"命令来修复坏损的AOF文件。

3). 用修复后的AOF文件重新启动Redis服务器。

6. Redis的数据备份：

在Redis中我们可以通过copy的方式在线备份正在运行的Redis数据文件。这是因为RDB文件一旦被生成之后就不会再被修改。Redis每次都是将最新的数据dump到一个临时文件中，之后在利用rename函数原子性的将临时文件改名为原有的数据文件名。因此我们可以说，在任意时刻copy数据文件都是安全的和一致的。鉴于此，我们就可以通过创建cron job的方式定时备份Redis的数据文件，并将备份文件copy到安全的磁盘介质中。

# 虚拟内存

一、简介：

和大多NoSQL数据库一样，Redis同样遵循了Key/Value数据存储模型。在有些情况下，Redis会将Keys/Values保存在内存中以提高数据查询和数据修改的效率，然而这样的做法并非总是很好的选择。鉴于此，我们可以将之进一步优化，即尽量在内存中只保留Keys的数据，这样可以保证数据检索的效率，而Values数据在很少使用的时候则可以被换出到磁盘。

在实际的应用中，大约只有10%的Keys属于相对比较常用的键，这样Redis就可以通过虚存将其余不常用的Keys和Values换出到磁盘上，而一旦这些被换出的Keys或Values需要被读取时，Redis则将其再次读回到主内存中。

二、应用场景：

对于大多数数据库而言，最为理想的运行方式就是将所有的数据都加载到内存中，而之后的查询操作则可以完全基于内存数据完成。然而在现实中这样的场景却并不普遍，更多的情况则是只有部分数据可以被加载到内存中。

在Redis中，有一个非常重要的概念，即keys一般不会被交换，所以如果你的数据库中有大量的keys，其中每个key仅仅关联很小的value，那么这种场景就不是非常适合使用虚拟内存。如果恰恰相反，数据库中只是包含少量的keys，而每一个key所关联的value却非常大，那么这种场景对于使用虚存就再合适不过了。

在实际的应用中，为了能让虚存更为充分的发挥作用以帮助我们提高系统的运行效率，我们可以将带有很多较小值的Keys合并为带有少量较大值的Keys。其中最主要的方法就是将原有的Key/Value模式改为基于Hash的模式，这样可以让很多原来的Keys成为Hash中的属性。

三、配置：

1). 在配置文件中添加以下配置项，以使当前Redis服务器在启动时打开虚存功能。

vm-enabled yes

2). 在配置文件中设定Redis最大可用的虚存字节数。如果内存中的数据大于该值，则有部分对象被换出到磁盘中，其中被换出对象所占用内存将被释放，直到已用内存小于该值时才停止换出。

vm-max-memory (bytes)

Redis的交换规则是尽量考虑"最老"的数据，即最长时间没有使用的数据将被换出。如果两个对象的age相同，那么Value较大的数据将先被换出。需要注意的是，Redis不会将Keys交换到磁盘，因此如果仅仅keys的数据就已经填满了整个虚存，那么这种数据模型将不适合使用虚存机制，或者是将该值设置的更大，以容纳整个Keys的数据。在实际的应用，如果考虑使用Redis虚拟内存，我们应尽可能的分配更多的内存交给Redis使用，以避免频繁的换入换出。

3). 在配置文件中设定页的数量及每一页所占用的字节数。为了将内存中的数据传送到磁盘上，我们需要使用交换文件。这些文件与数据持久性无关，Redis会在退出前会将它们全部删除。由于对交换文件的访问方式大多为随机访问，因此建议将交换文件存储在固态磁盘上，这样可以大大提高系统的运行效率。

vm-pages 134217728

vm-page-size 32

在上面的配置中，Redis将交换文件划分为vm-pages个页，其中每个页所占用的字节为vm-page-size，那么Redis最终可用的交换文件大小为：vm-pages \* vm-page-size。由于一个value可以存放在一个或多个页上，但是一个页不能持有多个value，鉴于此，我们在设置vm-page-size时需要充分考虑Redis的该特征。

4). 在Redis的配置文件中有一个非常重要的配置参数，即：

vm-max-threads 4

该参数表示Redis在对交换文件执行IO操作时所应用的最大线程数量。通常而言，我们推荐该值等于主机的CPU cores。如果将该值设置为0，那么Redis在与交换文件进行IO交互时，将以同步的方式执行此操作。

对于Redis而言，如果操作交换文件是以同步的方式进行，那么当某一客户端正在访问交换文件中的数据时，其它客户端如果再试图访问交换文件中的数据，该客户端的请求就将被挂起，直到之前的操作结束为止。特别是在相对较慢或较忙的磁盘上读取较大的数据值时，这种阻塞所带来的影响就更为突兀了。然而同步操作也并非一无是处，事实上，从全局执行效率视角来看，同步方式要好于异步方式，毕竟同步方式节省了线程切换、线程间同步，以及线程拉起等操作产生的额外开销。特别是当大部分频繁使用的数据都可以直接从主内存中读取时，同步方式的表现将更为优异。

如果你的现实应用恰恰相反，即有大量的换入换出操作，同时你的系统又有很多的cores，有鉴于此，你又不希望客户端在访问交换文件之前不得不阻塞一小段时间，如果确实是这样，我想异步方式可能更适合于你的系统。至于最终选用哪种配置方式，最好的答案将来自于不断的实验和调优。

# 服务器管理

一、概述：

Redis在设计之初就被定义为长时间不间断运行的服务进程，因此大多数系统配置参数都可以在不重新启动进程的情况下立即生效。即便是将当前的持久化模式从AOF切换到RDB也无需重启。

在Redis中，提供了一组和服务器管理相关的命令，其中就包含和参数设置有关的CONFIG SET/GET command。

二、相关命令列表：

CONFIG GET parameter

主要用于读取服务器的运行时参数，但是并不是所有的配置参数都可以通过该命令进行读取。其中该命令的参数接受glob风格的模式匹配规则，因此如果参数中包含模式元字符，那么所有匹配的参数都将以key/value方式被列出。如果参数是\*，那么该命令支持的所有参数都将被列出。最后需要指出的是，和redis.conf中不同的是，在命令中不能使用数量缩写格式，如GB、KB等，只能使用表示字节数量的整数值。

CONFIG SET parameter value

该命令用于重新配置Redis服务器的运行时参数，在设置成功之后无需重启便可生效。然而并非所有的参数都可以通过该命令进行动态设置，如果需要获悉该命令支持哪些参数，可以查看CONFIG GET \* 命令的执行结果。如果想在一个命令中设置多个同类型参数，如redis.conf配置文件中的save参数：save 900 1/save 300 10。在该命令中我们可以将多个key/value用双引号括起，并用空格符隔开，如：config set save "900 1 300 10"。 OK表示设置成功，否则返回相关的错误信息。

CONFIG RESETSTAT O(1) Reset INFO命令给出的统计数字。 始终返回OK。

DBSIZE 返回当前打开的数据库中Keys的数量。 Key的数量。

FLUSHALL 清空当前服务器管理的数据库中的所有Keys，不仅限于当前打开的数据库。

FLUSHDB 清空当前数据库中的所有Keys。

INFO 获取和服务器运行状况相关的一些列统计数字。

SAVE 设置RDB持久化模式的保存策略。

SHUTDOWN 停止所有的客户端，同时以阻塞的方式执行内存数据持久化。如果AOF模式被启用，则将缓存中的数据flush到AOF文件。退出服务器。

SLAVEOF host port

该命令用于修改SLAVE服务器的复制设置。如果一个Redis服务器已经处于SLAVE状态，SLAVEOF NO ONE命令将关闭当前服务器的被复制状态，与此同时将该服务器切换到MASTER状态。该命令的参数将指定MASTER服务器的监听IP和端口。还有一种情况是，当前服务器已经是另外一台MASTER的SLAVE了，在执行该命令后，当前服务器将终止和之前MASTER之间的复制关系，而将成为新MASTER的SLAVE，之前MASTER中的数据也将被清空，改为新MASTER中的数据。然而如果在当前SLAVE服务器上执行的是SLAVEOF NO ONE命令，那么该服务器只是中断与当前MASTER的复制关系，并升级为独立的MASTER，其中的数据也不会被清空。

SLOWLOG subcommand [argument]

该命令主要用于读取执行时间较长的命令。其中执行时间的评判标准仅为命令本身的执行时间，并不包括网络交互时间。和该命令相关的配置参数主要有两个，第一个就是执行之间的阈值(以微秒为单位)，即执行时间超过该值的命令都会被存入slowlog队列，以供该命令读取。第二个是slowlog队列的长度，如果当前命令在存入之前，该队列中的命令已经等于该参数，在命令进入之前，需要将队列中最老的命令移出队列。这样可以保证该队列所占用的内存总量保持在一个相对恒定的大小。由于slowlog队列不会被持久化到磁盘，因此Redis在收集命令时不会对性能产生很大的影响。通常我们可以将参数"slowlog-log-slower-than"设置为0，以便收集所有命令的执行时间。该命令还包含以下几个子命令：

1). SLOWLOG GET N: 从slowlog队列中读取命令信息，N表示最近N条命令的信息。

2). SLOWLOG LEN：获取slowlog队列的长度。

3). SLOWLOG RESET：清空slowlog中的内容。