



Redis 入门

今日内容介绍

- ◆ 通过 Java 程序操作 Redis 不同数据

今日内容学习目标

- ◆ 使用 Redis 命令操作常用数据结构的数据存取
- ◆ 在 Java 程序中通过 Jedis 连接 Redis

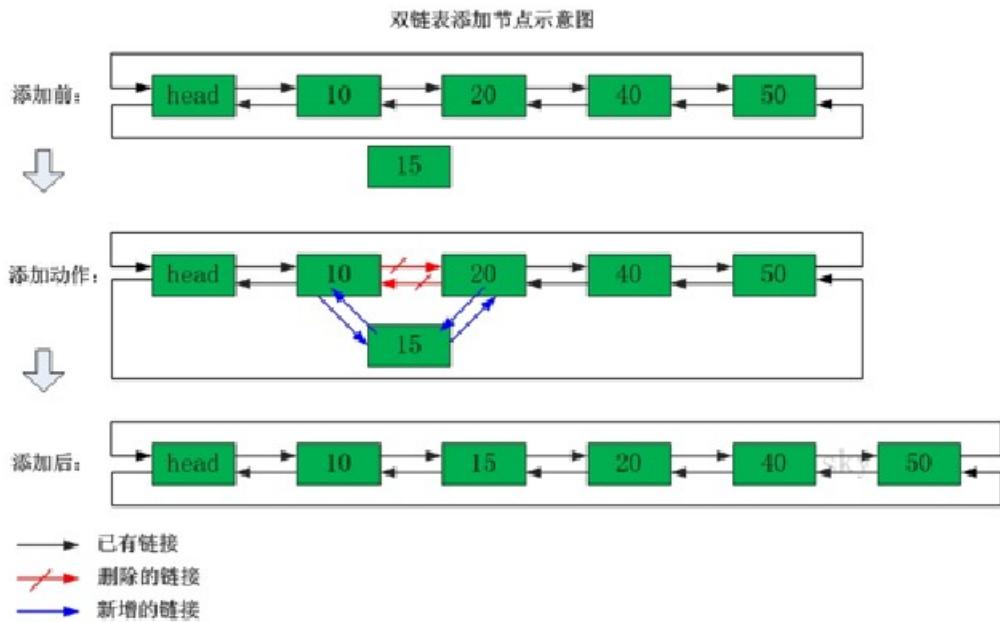
1.1 存储 list

1.1.1 概述

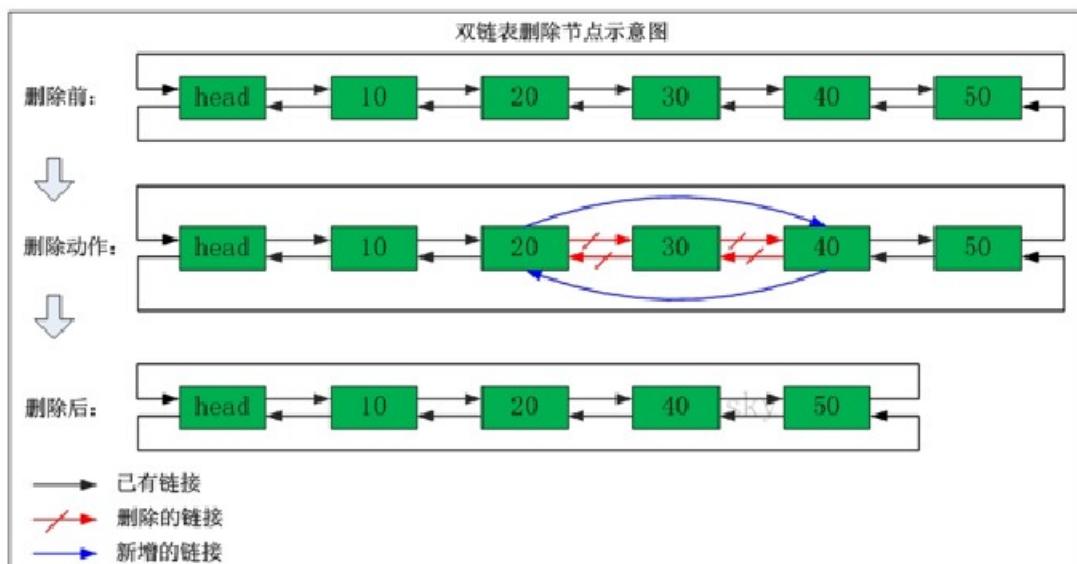
在 Redis 中，List 类型是按照插入顺序排序的字符串链表。和数据结构中的普通链表一样，我们可以在其头部(left)和尾部(right)添加新的元素。在插入时，如果该键并不存在，Redis 将为该键创建一个新的链表。与此相反，如果链表中所有的元素均被移除，那么该键也将会被从数据库中删除。List 中可以包含的最大元素数量是 4294967295。

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。

- 1、**ArrayList** 使用数组方式存储数据，所以根据索引查询数据速度快，而新增或者删除元素时需要设计到位移操作，所以比较慢。
- 2、**LinkedList** 使用双向链接方式存储数据，每个元素都记录前后元素的指针，所以插入、删除数据时只是更改前后元素的指针指向即可，速度非常快，然后通过下标查询元素时需要从头开始索引，所以比较慢。
- 3、双向链表中添加数据



4、双向链表中删除数据



1.1.2 常用命令

1.1.2.1 两端添加

- **lpush key values[value1 value2...]:** 在指定的 **key** 所关联的 **list** 的头部插入所有的 **values**，如果该 **key** 不存在，该命令在插入的之前创建一个与该 **key** 关联的空链表，之后再向该链表的头部插入数据。插入成功，返回元素的个数。



```
127.0.0.1:6379> lpush myList a b c
(integer) 3
127.0.0.1:6379> lpush myList 1 2 3
(integer) 6
```

- **rpush key values[value1、value2...]:** 在该 list 的尾部添加元素。

```
127.0.0.1:6379> rpush myList2 a b c d
(integer) 4
127.0.0.1:6379> rpush myList2 1 2 3
(integer) 7
```

1.1.2.2 查看列表

- **lrange key start end:** 获取链表中从 start 到 end 的元素的值, start、end 从 0 开始计数; 也可为负数, 若为-1 则表示链表尾部的元素, -2 则表示倒数第二个, 依次类推...

```
127.0.0.1:6379> lrange myList 0 5
1) "3"
2) "2"
3) "1"
4) "c"
5) "b"
6) "a"
```

```
127.0.0.1:6379> lrange myList2 0 -1
1) "a"
2) "b"
3) "c"
4) "d"
5) "1"
6) "2"
7) "3"
```

1.1.2.3 两端弹出

- **lpop key:** 返回并弹出指定的 key 关联的链表中的第一个元素, 即头部元素。如果该 key 不存在, 返回 nil; 若 key 存在, 则返回链表的头部元素。

```
127.0.0.1:6379> lpop myList
"3"
127.0.0.1:6379> lrange myList 0 -1
1) "2"
2) "1"
3) "c"
4) "b"
5) "a"
```

- **rpop key:** 从尾部弹出元素。



```
127.0.0.1:6379> rpop myList2
"3"
127.0.0.1:6379> lrange myList2 0 -1
1) "a"
2) "b"
3) "c"
4) "d"
5) "1"
6) "2"
```

1.1.2.4 获取列表中元素的个数

- **llen key:** 返回指定的 **key** 关联的链表中的元素的数量。

```
127.0.0.1:6379> llen myList
(integer) 5
127.0.0.1:6379> llen myList2
(integer) 6
127.0.0.1:6379> llen myList3
(integer) 0
```

1.1.3 扩展命令（了解）

- **lpushx key value:** 仅当参数中指定的 **key** 存在时，向关联的 **list** 的头部插入 **value**。如果不存在，将不进行插入。

```
127.0.0.1:6379> lpushx myList x
(integer) 6
127.0.0.1:6379> lrange myList 0 -1
1) "x"
2) "2"
3) "1"
4) "c"
5) "b"
6) "a"
127.0.0.1:6379> lpushx myList3 x
(integer) 0
```

- **rpushx key value:** 在该 **list** 的尾部添加元素

```
127.0.0.1:6379> rpushx myList2 y
(integer) 7
127.0.0.1:6379> lrange myList2 0 -1
1) "a"
2) "b"
3) "c"
4) "d"
5) "1"
6) "2"
7) "y"
```

- **lrem key count value:** 删除 **count** 个值为 **value** 的元素，如果 **count** 大于 **0**，从头向尾遍历并删除 **count** 个值为 **value** 的元素，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。

0) 初始化数据

```
127.0.0.1:6379> lpush myList3 1 2 3
(integer) 3
127.0.0.1:6379> lpush myList3 1 2 3
(integer) 6
127.0.0.1:6379> lpush myList3 1 2 3
(integer) 9
```

1) 从头删除，2个数字“3”

```
lrem myList3 2 3
```

```
127.0.0.1:6379> lrange myList3 0 -1
1) "3" □
2) "2"
3) "1"
4) "3" □
5) "2"
6) "1"
7) "3"
8) "2"
9) "1"
127.0.0.1:6379> lrem myList3 2 3
(integer) 2
127.0.0.1:6379> lrange myList3 0 -1
1) "2"
2) "1"
3) "2"
4) "1"
5) "3"
6) "2"
7) "1"
```

2) 从尾删除，2个数字“1”

```
lrem myList3 -2 1
```

```
127.0.0.1:6379> lrange myList3 0 -1
1) "2"
2) "1"
3) "2"
4) "1" □
5) "3"
6) "2"
7) "1" □
127.0.0.1:6379> lrem myList3 -2 1
(integer) 2
127.0.0.1:6379> lrange myList3 0 -1
1) "2"
2) "1"
3) "2"
4) "3"
5) "2"
```

3) 删除所有数字“2”

```
lrem myList3 0 2
```

```
127.0.0.1:6379> lrange myList3 0 -1
1) "2"
2) "1"
3) "2"
4) "3"
5) "2"
127.0.0.1:6379> lrem myList3 0 2
(integer) 3
127.0.0.1:6379> lrange myList3 0 -1
1) "1"
2) "3"
```

- **lset key index value:** 设置链表中的 **index** 的脚标的元素值，**0** 代表链表的头元素，**-1** 代表链表的尾元素。操作链表的脚标不存在则抛异常。



```
127.0.0.1:6379> lrange mylist 0 -1
1) "x"
2) "2"
3) "1"
4) "c"
5) "b"
6) "a"
127.0.0.1:6379> lset mylist 3 444
OK
127.0.0.1:6379> lrange mylist 0 -1
1) "x"
2) "2"
3) "1"
4) "444"
5) "b"
6) "a"
```

- **linsert key before|after pivot value:** 在 pivot 元素前或者后插入 value 这个元素。

```
127.0.0.1:6379> lpush myList4 a b c
(integer) 3
127.0.0.1:6379> lpush myList4 a b c
(integer) 6
127.0.0.1:6379> lrange myList4 0 -1
1) "c"
2) "b"
3) "a"
4) "c"
5) "b"
6) "a"           第一个"b"之前
127.0.0.1:6379> linsert myList4 before b 11
(integer) 7
127.0.0.1:6379> lrange myList4 0 -1
1) "c"
2) "11" □
3) "b"
4) "a"
5) "c"
6) "b"
7) "a"           第一个"b"之后
127.0.0.1:6379> linsert myList4 after b 22
(integer) 8
127.0.0.1:6379> lrange myList4 0 -1
1) "c"
2) "11"
3) "b"
4) "22" □
5) "a"
6) "c"
7) "b"
8) "a"
```

- **rpoplpush resource destination:** 将链表中的尾部元素弹出并添加到头部。[循环操作]

```
127.0.0.1:6379> lpush myList5 1 2 3
(integer) 3
127.0.0.1:6379> lpush myList6 a b c
(integer) 3
```

- 1) 将 myList5 右端弹出，压入到 myList6 左边。

```
127.0.0.1:6379> lrange myList5 0 -1
1) "3"
2) "2"
3) "1"
127.0.0.1:6379> lrange myList6 0 -1
1) "c"
2) "b"
3) "a"
127.0.0.1:6379> rpoplpush myList5 myList6
"1"
127.0.0.1:6379> lrange myList5 0 -1
1) "3"
2) "2"
127.0.0.1:6379> lrange myList6 0 -1
1) "1"
2) "c"
3) "b"
4) "a"
```

2) 将 myList6 右端数据弹出，压入到左端

```
127.0.0.1:6379> lrange myList6 0 -1
1) "1"
2) "c"
3) "b"
4) "a" ↗
127.0.0.1:6379> rpoplpush myList6 myList6
"a"
127.0.0.1:6379> lrange myList6 0 -1
1) "a" ↗
2) "1"
3) "c"
4) "b"
```

1.1.4 使用场景

rpoplpush 的使用场景：

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 **LPOP** 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。





1.2 存储 set

1.2.1 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。

1.2.2 常用命令

1.2.2.1 添加/删除元素

- **sadd key values[value1、value2...]**: 向 **set** 中添加数据，如果该 **key** 的值已有则不会重复添加

```
127.0.0.1:6379> sadd myset a b c
(integer) 3
127.0.0.1:6379> sadd myset a
(integer) 0
```

- **srem key members[member1、member2...]**: 删除 **set** 中指定的成员

```
127.0.0.1:6379> sadd myset 1 2 3
(integer) 3
127.0.0.1:6379> srem myset 1 2
(integer) 2
```

1.2.2.2 获得集合中的元素

- **smembers key**: 获取 **set** 中所有的成员

```
127.0.0.1:6379> smembers myset
1) "b"
2) "c"
3) "3"
4) "a"
```

- **sismember key member**: 判断参数中指定的成员是否在该 **set** 中，**1** 表示存在，**0** 表示不存在或

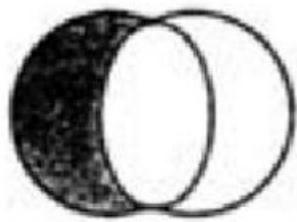


者该 **key** 本身不存在。(无论集合中有多少元素都可以极速的返回结果)

```
127.0.0.1:6379> sismember myset a
(integer) 1
127.0.0.1:6379> sismember myset x
(integer) 0
```

1.2.2.3 集合的差集运算 A-B

- **sdiff key1 key2...:** 返回 **key1** 与 **key2** 中相差的成员，而且与 **key** 的顺序有关。即返回差集。



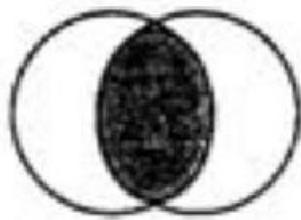
A-B

(属于 A 并且不属于 B 的元素构成的集合)

```
127.0.0.1:6379> sadd mya1 a b c
(integer) 3
127.0.0.1:6379> sadd myb1 a c 1 2
(integer) 4
127.0.0.1:6379> sdiff mya1 myb1
1) "b"
```

1.2.2.4 集合的交集运算 A ∩ B

- **sinter key1 key2 key3...:** 返回交集。



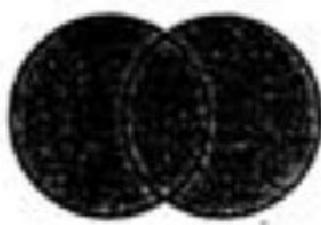
A ∩ B

(属于 A 且属于 B 的元素构成的集合)

```
127.0.0.1:6379> sadd mya2 a b c
(integer) 3
127.0.0.1:6379> sadd myb2 a c 1 2
(integer) 4
127.0.0.1:6379> sinter mya2 myb2
1) "c"
2) "a"
```

1.2.2.5 集合的并集运算 $A \cup B$

- `sunion key1 key2 key3...`: 返回并集。


$$A \cup B$$

(属于 A 或者属于 B 的元素构成的集合)

```
127.0.0.1:6379> sadd mya3 a b c
(integer) 3
127.0.0.1:6379> sadd myb3 a c 1 2
(integer) 4
127.0.0.1:6379> sunion mya3 myb3
1) "a"
2) "b"
3) "c"
4) "2"
5) "1"
```

1.2.3 扩展命令（了解）

- `scard key`: 获取 set 中成员的数量

```
127.0.0.1:6379> smembers myset
1) "b"
2) "c"
3) "3"
4) "a"
127.0.0.1:6379> scard myset
(integer) 4
```

- `srandmember key`: 随机返回 set 中的一个成员



```
127.0.0.1:6379> srandmember myset
"b"
127.0.0.1:6379> srandmember myset
"b"
```

- **sdiffstore destination key1 key2...**: 将 key1、key2 相差的成员存储在 **destination** 上

```
127.0.0.1:6379> sdiff mya1 myb1
1) "b"
127.0.0.1:6379> sdiffstore my1 mya1 myb1
(integer) 1
127.0.0.1:6379> smembers my1
1) "b"
```

- **sinterstore destination key[key...]**: 将返回的交集存储在 **destination** 上

```
127.0.0.1:6379> sinter mya2 myb2
1) "c"
2) "a"
127.0.0.1:6379> sinterstore my2 mya2 myb2
(integer) 2
127.0.0.1:6379> smembers my2
1) "c"
2) "a"
```

- **sunionstore destination key[key...]**: 将返回的并集存储在 **destination** 上

```
127.0.0.1:6379> sunion mya3 myb3
1) "a"
2) "b"
3) "c"
4) "2"
5) "1"
127.0.0.1:6379> sunionstore my3 mya3 myb3
(integer) 5
127.0.0.1:6379> smembers my3
1) "a"
2) "b"
3) "c"
4) "2"
5) "1"
```

1.2.4 使用场景

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。

1.3 存储 sortedset

1.3.1 概述

Sorted-Set 和 **Set** 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-Set** 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-Set** 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-Set** 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仓库中是很难实现的，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。

例如：游戏排名、微博热点话题等使用场景。

1.3.2 常用命令

1.3.2.1 添加元素

- **zadd key score member score2 member2 ...**：将所有成员以及该成员的分数存放到 **sorted-set** 中。如果该元素已经存在则会用新的分数替换原有的分数。返回值是新加入到集合中的元素个数，不包含之前已经存在的元素。

```
127.0.0.1:6379> zadd mysort 70 zhangsan 80 lisi 90 wangwu
(integer) 3
```

```
127.0.0.1:6379> zadd mysort 100 zhangsan 存在替换分数
(integer) 0
127.0.0.1:6379> zadd mysort 50 jack      添加
(integer) 1
```

1.3.2.2 获得元素

- **zscore key member**: 返回指定成员的分数

```
(integer) 1
127.0.0.1:6379> zscore mysort zhangsan
"100"
```

- **zcard key**: 获取集合中的成员数量

```
127.0.0.1:6379> zcard mysort
(integer) 4
```

1.3.2.3 删除元素

- **zrem key member[member...]**: 移除集合中指定的成员，可以指定多个成员。



```
127.0.0.1:6379> zrem mysort jack wangwu
(integer) 2
127.0.0.1:6379> zcard mysort
(integer) 2
```

1.3.2.4 范围查询

- **zrange key start end [withscores]**: 获取集合中脚标为 **start-end** 的成员, [**withscores**]参数表明返回的成员包含其分数。

```
127.0.0.1:6379> zadd mysort 85 jack 95 rose
(integer) 2
127.0.0.1:6379> zrange mysort 0 -1
1) "lisi"
2) "jack"
3) "rose"
4) "zhangsan"
```

```
127.0.0.1:6379> zrange mysort 0 -1 withscores
1) "lisi"
2) "80"
3) "jack"
4) "85"
5) "rose"
6) "95"
7) "zhangsan"
8) "100"
```

- **zrevrange key start stop [withscores]**: 照元素分数从大到小的顺序返回索引从 **start** 到 **stop** 之间的所有元素（包含两端的元素）

```
127.0.0.1:6379> zrevrange mysort 0 -1 withscores
1) "zhangsan"
2) "100"
3) "rose"
4) "95"
5) "jack"
6) "85"
7) "lisi"
8) "80"
```

- **zremrangebyrank key start stop**: 按照排名范围删除元素

```
4) "zhangsan"
127.0.0.1:6379> zremrangebyrank mysort 0 4
(integer) 4
```

- **zremrangebyscore key min max**: 按照分数范围删除元素

```
127.0.0.1:6379> zadd mysort 80 zhangsi 90 lisi 100 wangwu
(integer) 3
127.0.0.1:6379> zremrangebyscore mysort 80 100
(integer) 3
127.0.0.1:6379> zrange mysort 0 -1
(empty list or set)
```



1.3.3 扩展命令（了解）

- **zrangebyscore key min max [withscores] [limit offset count]**: 返回分数在[min,max]的成员并按照分数从低到高排序。[withscores]: 显示分数; [limit offset count]: offset, 表明从脚标为 offset 的元素开始并返回 count 个成员。

```
127.0.0.1:6379> zrangebyscore mysort 0 100 withscores
1) "lisi"
2) "80"
3) "jack"
4) "85"
5) "rose"
6) "95"
7) "zhangsan"
8) "100"
```

```
127.0.0.1:6379> zrangebyscore mysort 0 100 withscores limit 0 2
1) "lisi"
2) "80"
3) "jack"
4) "85"
```

- **zincrby key increment member**: 设置指定成员的增加的分数。返回值是更改后的分数。

```
127.0.0.1:6379> zincrby mysort 3 lisi
"83"
127.0.0.1:6379> zscore mysort lisi
"83"
```

- **zcount key min max**: 获取分数在[min,max]之间的成员

```
127.0.0.1:6379> zcount mysort 80 90
(integer) 2
```

- **zrank key member**: 返回成员在集合中的排名。(从小到大)

```
127.0.0.1:6379> zrank mysort lisi
(integer) 0
127.0.0.1:6379> zrank mysort rose
(integer) 2
```

- **zrevrank key member**: 返回成员在集合中的排名。(从大到小)

```
127.0.0.1:6379> zrevrank mysort rose
(integer) 1
127.0.0.1:6379> zrevrank mysort lisi
(integer) 3
```



1.3.4 使用场景

- 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。
- 2、**Sorted-Set** 类型还可用于构建索引数据。

第2章 keys 的通用操作

- **keys pattern:** 获取所有与 **pattern** 匹配的 **key**，返回所有与该 **key** 匹配的 **keys**。*****表示任意一个或多个字符，**?**表示任意一个字符

```
127.0.0.1:6379> keys *
1) "myset"
2) "mya1"
3) "mylist6"          所有的key
4) "mylist2"
5) "myhash"
6) "my1"
7) "mylist"
8) "mylist5"
9) "name"
10) "my2"
11) "num"
12) "my3"
13) "company"
14) "num4"
15) "num2"
16) "mya3"
17) "myb3"
18) "num3"
19) "num5"
20) "mya2"            以my开头，3长度
21) "myb2"
22) "myb1"
127.0.0.1:6379> keys my?
1) "my1"
2) "my2"
3) "my3"
```

- **del key1 key2...:** 删除指定的 **key**

```
127.0.0.1:6379> del my1 my2 my3
(integer) 3
```

- **exists key:** 判断该 **key** 是否存在，**1** 代表存在，**0** 代表不存在

```
127.0.0.1:6379> exists my1
(integer) 0
127.0.0.1:6379> exists mya1
(integer) 1
```

- **rename key newkey:** 为当前的 **key** 重命名



```
127.0.0.1:6379> get company
"itheima"
127.0.0.1:6379> rename company newcompany
OK
127.0.0.1:6379> get company
(nil)
127.0.0.1:6379> get newcompany
"itheima"
```

- **expire key** : 设置过期时间，单位：秒

```
127.0.0.1:6379> expire newcompany 1000
(integer) 1
```

- **ttl key**: 获取该 key 所剩的超时时间，如果没有设置超时，返回-1。如果返回-2 表示超时不存在。

```
127.0.0.1:6379> ttl newcompany
(integer) -1
```

```
127.0.0.1:6379> ttl newcompany
(integer) 934
```

- **type key**: 获取指定 key 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**。

```
(integer) 1
127.0.0.1:6379> type newcompany
none
127.0.0.1:6379> type name
string
127.0.0.1:6379> type mylist
list
127.0.0.1:6379> type myset
set
127.0.0.1:6379> type myhash
hash
127.0.0.1:6379> type mysort
none
127.0.0.1:6379> zadd mysort 10 lisi
(integer) 1
127.0.0.1:6379> type mysort
zset
```

第3章 jedis 入门

3.1 jedis 介绍

Redis 不仅是使用命令来操作，现在基本上主流的语言都有客户端支持，比如 **java**、**C**、**C#**、**C++**、**php**、**Node.js**、**Go** 等。

在官方网站里列一些 Java 的客户端，有 **Jedis**、**Redisson**、**Jredis**、**JDBC-Redis**、等其中官方推荐使用 **Jedis** 和 **Redisson**。 在企业中用的最多的就是 **Jedis**，下面我们就重点学习下 **Jedis**。

Jedis 同样也是托管在 **github** 上，地址：<https://github.com/xetorthio/jedis>



3.2 Java 连接 Redis

3.2.1 导入 jar 包

```
[ ] commons-pool2-2.3.jar  
[ ] jedis-2.7.0.jar
```

3.2.2 单实例连接

```
@Test  
public void testJedisSingle(){  
    //1 设置ip地址和端口  
    Jedis jedis = new Jedis("192.168.137.128", 6379);  
    //2 设置数据  
    jedis.set("name", "itheima");  
    //3 获得数据  
    String name = jedis.get("name");  
    System.out.println(name);  
    //4 释放资源  
    jedis.close();  
}  
  
@Test  
public void testJedisSingle(){  
    //1 设置 ip 地址和端口  
    Jedis jedis = new Jedis("192.168.137.128", 6379);  
    //2 设置数据  
    jedis.set("name", "itheima");  
    //3 获得数据  
    String name = jedis.get("name");  
    System.out.println(name);  
    //4 释放资源  
    jedis.close();  
}
```

3.2.3 连接超时

- 如果运行上面代码时，抛如下异常

```
redis.clients.jedis.exceptions.JedisConnectionException:  
java.net.SocketTimeoutException: connect timed out
```

- 必须设置 linux 防火墙



```
vim /etc/sysconfig/iptables
-A INPUT -m state --state NEW -m tcp -p tcp --dport 22 -j ACCEPT
-A INPUT -m state --state NEW -m tcp -p tcp --dport 6379 -j ACCEPT
service iptables restart

[root@localhost redis]# service iptables restart
iptables: 将链设置为政策 ACCEPT: filter
iptables: 清除防火墙规则:
iptables: 正在卸载模块:
iptables: 应用防火墙规则:
[确定]
[确定]
[确定]
[确定]
```

3.2.4 连接池连接

```
@Test
public void testJedisPool(){
    //1 获得连接池配置对象，设置配置项
    JedisPoolConfig config = new JedisPoolConfig();
    // 1.1 最大连接数
    config.setMaxTotal(30);
    // 1.2 最大空闲连接数
    config.setMaxIdle(10);

    //2 获得连接池
    JedisPool jedisPool = new JedisPool(config, "192.168.137.128", 6379);

    //3 获得核心对象
    Jedis jedis = null;
    try {
        jedis = jedisPool.getResource();

        //4 设置数据
        jedis.set("name", "itcast");
        //5 获得数据
        String name = jedis.get("name");
        System.out.println(name);

    } catch (Exception e) {
        e.printStackTrace();
    } finally{
        if(jedis != null){
            jedis.close();
        }
        // 虚拟机关闭时，释放pool资源
        if(jedisPool != null){
            jedisPool.close();
        }
    }
}

@Test
public void testJedisPool(){
    //1 获得连接池配置对象，设置配置项
    JedisPoolConfig config = new JedisPoolConfig();
```



```
// 1.1 最大连接数
config.setMaxTotal(30);

// 1.2 最大空闲连接数
config.setMaxIdle(10);

//2 获得连接池
JedisPool jedisPool = new JedisPool(config, "192.168.137.128", 6379);

//3 获得核心对象
Jedis jedis = null;
try {
    jedis = jedisPool.getResource();

    //4 设置数据
    jedis.set("name", "itcast");
    //5 获得数据
    String name = jedis.get("name");
    System.out.println(name);

} catch (Exception e) {
    e.printStackTrace();
} finally{
    if(jedis != null){
        jedis.close();
    }
    // 虚拟机关闭时，释放 pool 资源
    if(jedisPool != null){
        jedisPool.close();
    }
}
}
```

第4章 Redis 特性

4.1 多数据库

4.1.1 概念

一个 Redis 实例可以包括多个数据库，客户端可以指定连接某个 redis 实例的哪个数据库，就好



比一个 mysql 中创建多个数据库，客户端连接时指定连接哪个数据库。

一个 redis 实例最多可提供 16 个数据库，下标从 0 到 15，客户端默认连接第 0 号数据库，也可以通过 select 选择连接哪个数据库，如下连接 1 号库：

```
127.0.0.1:6379> select 1
OK
127.0.0.1:6379[1]> keys *
(empty list or set)
```

连接 0 号数据库：

```
127.0.0.1:6379[1]> select 0
OK
127.0.0.1:6379> keys *
1) "myset"
2) "mya1"
3) "mylist6"
4) "mylist2"
5) "myhash"
6) "mylist"
7) "mylist5"
```

4.1.2 将 newkey 移植到 1 号库

- move newkey 1: 将当前库的 key 移植到 1 号库中

```
127.0.0.1:6379> move myset 1
(integer) 1
127.0.0.1:6379> select 1
OK
127.0.0.1:6379[1]> keys *
1) "myset"
127.0.0.1:6379[1]> type myset
set
```

4.2 服务器命令(自学)

- ping, 测试连接是否存活

```
127.0.0.1:6379> ping
PONG
```

//执行下面命令之前，我们停止 redis 服务器

```
redis 127.0.0.1:6379> ping
Could not connect to Redis at 127.0.0.1:6379: Connection refused
```

- echo, 在命令行打印一些内容

```
127.0.0.1:6379> echo zhangsi
"zhangsi"
```

- select, 选择数据库。Redis 数据库编号从 0~15，可以选择任意一个数据库来进行数据的存取。



```
127.0.0.1:6379> select 1
OK
127.0.0.1:6379[1]> select 16
(error) ERR invalid DB index
```

当选择 16 时，报错，说明没有编号为 16 的这个数据库

- quit，退出连接。

```
#> keys=1, expires=0
127.0.0.1:6379> quit
```

- dbsize，返回当前数据库中 key 的数目。

```
127.0.0.1:6379> dbsize
(integer) 0
127.0.0.1:6379> set name xx
OK
127.0.0.1:6379> dbsize
(integer) 1
```

- info，获取服务器的信息和统计。

```
127.0.0.1:6379> info
# Server
redis_version:3.0.0
redis_git_sha1:00000000
redis_git_dirty:0
redis_build_id:3ee8f8d1a979a04f
redis_mode:standalone
os:Linux 2.6.32-431.el6.i686 i686
arch_bits:32
multiplexing_api:epoll
gcc_version:4.4.7
process_id:5555
run_id:a128768453a78fefbdda5b66bd86
```

- flushdb，删除当前选择数据库中的所有 key。

```
127.0.0.1:6379> dbsize
(integer) 1
127.0.0.1:6379> flushdb
OK
127.0.0.1:6379> dbsize
(integer) 0
```

- flushall，删除所有数据库中的所有 key。



```
127.0.0.1:6379> select 0
OK
127.0.0.1:6379> set name yy
OK
127.0.0.1:6379> dbsize
(integer) 1
127.0.0.1:6379> select 1
OK
127.0.0.1:6379[1]> set pwd zz
OK
127.0.0.1:6379[1]> dbsize
(integer) 1
127.0.0.1:6379[1]> flushall
OK
127.0.0.1:6379[1]> dbsize
(integer) 0
127.0.0.1:6379[1]> select 0
OK
127.0.0.1:6379> dbsize
(integer) 0
```

在本例中我们先查看了一个 1 号数据库中有一个 key，然后我切换到 0 号库执行 flushall 命令，结果 1 号库中的 key 也被清除了，说是此命令工作正常。

4.3 消息订阅与发布

- **subscribe channel:** 订阅频道，例：subscribe mychat，订阅 mychat 这个频道
- **psubscribe channel*:** 批量订阅频道，例：psubscribe s*，订阅以“s”开头的频道
- **publish channel content:** 在指定的频道中发布消息，如 publish mychat ‘today is a newday’
- 步骤 1：在第一个连接中，订阅 mychat 频道。此时如果没有人“发布”消息，当前窗口处于等待状态。

```
[root@localhost redis]# ./bin/redis-cli
127.0.0.1:6379> subscribe mychat
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "mychat"
3) (integer) 1
```

- 步骤 2：在另一个窗口中，在 mychat 频道中，发布消息。

```
127.0.0.1:6379> publish mychat '111'
(integer) 1
127.0.0.1:6379> 当消息发布后，订阅窗口立即获得信息
```



```
[root@localhost redis]# ./bin/redis-cli
127.0.0.1:6379> subscribe mychat
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "mychat"
3) (integer) 1
1) "message"
2) "mychat"
3) "111"
```

- 步骤 3：再第三个窗口，批量订阅以 my 开头所有频道。

```
[root@localhost ~]# /usr/local/redis/bin/redis-cli
127.0.0.1:6379> psubscribe my*
Reading messages... (press Ctrl-C to quit)
1) "psubscribe"
2) "my*"
3) (integer) 1
```

- 步骤 4：在第二个窗口，分别在“mychat”和“mychat2”发布消息

```
127.0.0.1:6379> publish mychat '111'
(integer) 1
127.0.0.1:6379> publish mychat '222'
(integer) 2
127.0.0.1:6379>
```

同时通知“mychat”频道的所有订阅者

```
127.0.0.1:6379> publish mychat '111'
(integer) 1
127.0.0.1:6379> publish mychat '222'
(integer) 2
127.0.0.1:6379> publish mychat '333'
(integer) 2
127.0.0.1:6379> publish mychat2 '444'
(integer) 1
127.0.0.1:6379>
```

4.4 redis 事务

4.4.1 概念

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/这三个命令是我们实现事务的基石。



4.4.2 redis 事务特征

- 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。

4.4.3 命令解释

- **multi**: 开启事务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，类似与关系型数据库中的：**begin transaction**
- **exec**: 提交事务，类似与关系型数据库中的：**commit**
- **discard**: 事务回滚，类似与关系型数据库中的：**rollback**

4.4.4 测试

4.4.4.1 正常执行事务

- 步骤 1：在窗口 1，设置 **num**，并获得数据

```
127.0.0.1:6379> set num 1
OK
127.0.0.1:6379> get num
"1"
```

- 步骤 2：在窗口 2，**num 累加 1**，并获得数据



```
127.0.0.1:6379> incr num
(integer) 2
127.0.0.1:6379> get num
"2"
```

- 步骤 3：在窗口 1，获得数据

```
127.0.0.1:6379> get num
"2"
```

- 步骤 4：在窗口 1，开启事务，多次累加数据。

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379> incr num
QUEUED
127.0.0.1:6379> incr num
QUEUED
```

- 步骤 5：在窗口 2，获得数据

```
127.0.0.1:6379> get num
"2"
```

- 步骤 6：提交事务

```
127.0.0.1:6379> exec
1) (integer) 3
2) (integer) 4
```

```
127.0.0.1:6379> set num 1
OK
127.0.0.1:6379> get num
"1"
127.0.0.1:6379> get num
"2"
127.0.0.1:6379> multi
OK
127.0.0.1:6379> incr num
QUEUED
127.0.0.1:6379> incr num
QUEUED
127.0.0.1:6379> exec
1) (integer) 3
2) (integer) 4
127.0.0.1:6379>
```



4.4.4.2 回滚

```
127.0.0.1:6379> set user jack
OK
127.0.0.1:6379> get user    数据是: jack
"jack"
127.0.0.1:6379> multi      开启事务
OK
127.0.0.1:6379> set user rose  设置新数据
QUEUED
127.0.0.1:6379> discard     回滚事务
OK
127.0.0.1:6379> get user
"jack"                         数据仍是: jack
127.0.0.1:6379>
```

4.4.4.3 失败命令

```
127.0.0.1:6379> set num 10
OK
127.0.0.1:6379> get num          1. 初始数据: 10
"10"
127.0.0.1:6379> multi
OK
127.0.0.1:6379> incrby num 5   2. +5 , num=15
QUEUED
127.0.0.1:6379> incrby num x   3. 累加x, 抛异常
QUEUED
127.0.0.1:6379> incrby num 5   4. +5, num=20
QUEUED
127.0.0.1:6379> exec
1) (integer) 15
2) (error) ERR value is not an integer or out of range
3) (integer) 20
127.0.0.1:6379> get num          5. 当提交事务, 执行所有操作, 如果部分操作异常, 将被忽略
"20"
```

第5章 redis 持久化（了解）

5.1 概述

Redis 的高性能是由于其将所有数据都存储在了内存中，为了使 Redis 在重启之后仍能保证数据不丢失，需要将数据从内存中同步到硬盘中，这一过程就是持久化。

Redis 支持两种方式的持久化，一种是 RDB 方式，一种是 AOF 方式。可以单独使用其中一种或将二者结合使用。

1、RDB 持久化（默认支持，无需配置）

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、redis 可以同时使用 RDB 和 AOF

5.2 RDB

5.2.1 优势

- 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。
- 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
- 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（分叉）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。
- 相比于 AOF 机制，如果数据集很大，RDB 的启动效率会更高。

5.2.2 劣势

- 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。
- 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

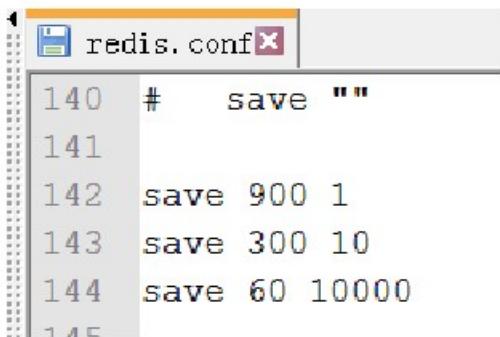
5.2.3 配置说明 Snapshotting

5.2.3.1 快照参数设置

- save 900 1 #每 900 秒(15 分钟)至少有 1 个 key 发生变化，则 dump 内存快照。
- save 300 10 #每 300 秒(5 分钟)至少有 10 个 key 发生变化，则 dump 内存快照
- save 60 10000 #每 60 秒(1 分钟)至少有 10000 个 key 发生变化，则 dump 内存快照

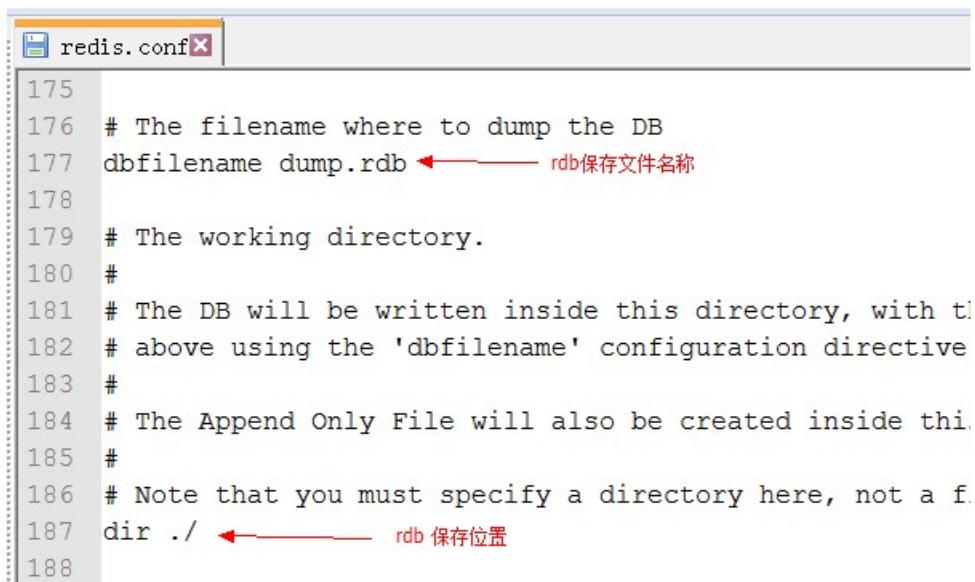


```
[root@localhost ~]# cat /usr/local/redis/redis.conf | grep -i save
# Save the DB on disk:
#   save <seconds> <changes>
#   will save the DB if both the given number of seconds and the given
#   In the example below the behaviour will be to save:
#   Note: you can disable saving completely by commenting out all "save" lines.
#   It is also possible to remove all the previously configured save
#   points by adding a save directive with a single empty string argument
#   save ""
save 900 1
save 300 10
save 60 10000
# (at least one save point) and the latest background save failed.
stop-writes-on-bgsave-error yes
```



```
140 # save ""
141
142 save 900 1
143 save 300 10
144 save 60 10000
145
```

5.2.3.2 保存位置设置



```
175
176 # The filename where to dump the DB
177 dbfilename dump.rdb ← rdb 保存文件名称
178
179 # The working directory.
180 #
181 # The DB will be written inside this directory, with t:
182 # above using the 'dbfilename' configuration directive
183 #
184 # The Append Only File will also be created inside thi:
185 #
186 # Note that you must specify a directory here, not a f:
187 dir ./ ← rdb 保存位置
188
```

```
[root@localhost redis]# ll /usr/local/redis
总用量 52
drwxr-xr-x. 2 root root 4096 7月 13 18:28 bin
-rw-r--r--. 1 root root 38 7月 29 16:08 dump.rdb
-rw-r--r--. 1 root root 41404 7月 13 21:13 redis.conf
```

5.3 AOF

5.3.1 优势

- 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。

5.3.2 劣势

- 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。

5.3.3 配置 AOF

5.3.3.1 配置信息

- **always** #每次有数据修改发生时都会写入 **AOF** 文件
- **everysec** #每秒钟同步一次，该策略为 **AOF** 的缺省策略
- **no** #从不同步。高效但是数据不会被持久化

重写 **AOF**：若不满足重写条件时，可以手动重写，命令：**bgrewriteaof**



```
# AOF and RDB persistence can be enabled at the same time without problems.
# If the AOF is enabled on startup Redis will load the AOF, that is the file
# with the better durability guarantees. AOF在redis启动时被加载，必须保证AOF文件完整性。
#
# Please check http://redis.io/topics/persistence for more information.
appendonly yes
```

```
# The name of the append only file
# appendfilename appendonly.aof
```

redis.conf AOF和RDB 可以同时使用

```
498 # AOF and RDB persistence can be enabled at the same time without problems.
499 # If the AOF is enabled on startup Redis will load the AOF, that is the file
500 # with the better durability guarantees. AOF在redis启动时被加载，必须保证AOF文件完整性。
501 #
502 # Please check http://redis.io/topics/persistence for more information.
503
504 appendonly no ← 默认关闭
505
```

策略的选择：

```
# If unsure, use "everysec".
#
appendfsync always
#appendfsync everysec
# appendfsync no
```

redis.conf

```
532
533 # appendfsync always
534 appendfsync everysec
535 # appendfsync no
```

5.3.3.2 数据恢复演示

- 1、**flushall** 操作 清空数据库
- 2、及时关闭 redis 服务器（防止 **dump.rdb**）。 **shutdown nosave**
- 3、编辑 **aof** 文件，将日志中的 **flushall** 命令删除并重启服务即可

- 步骤 1：开启 **aop**，并设置成总是保存。然后重启 **redis**。

192.168.137.128 (2) ×

```
# Please check http://redis.io/topics/persistence for more information.
appendonly yes
```

192.168.137.128 (2) ×

```
appendfsync always
#appendfsync everysec
```



- 步骤 2：在窗口 1 进行若干操作

```
127.0.0.1:6379> set name jack
OK
127.0.0.1:6379> set num 10
OK
127.0.0.1:6379> set n1 10
OK
127.0.0.1:6379> set n2 10
OK
127.0.0.1:6379> set n3 10
OK
127.0.0.1:6379> keys *
1) "num"
2) "n1"
3) "n3"
4) "name"
5) "n2"
```

- 步骤 3：在窗口 1，清空数据库

```
127.0.0.1:6379> flushall
OK
127.0.0.1:6379> keys *
(empty list or set)
```

- 步骤 4：在窗口 2，关闭 redis

```
[root@localhost redis]# ll
总用量 56
-rw-r--r--. 1 root root 173 7月 29 16:48 appendonly.aof
drwxr-xr-x. 2 root root 4096 7月 13 18:28 bin
-rw-r--r--. 1 root root 38 7月 29 16:47 dump.rdb
-rw-r--r--. 1 root root 41404 7月 29 16:46 redis.conf
[root@localhost redis]# ./bin/redis-cli shutdown
[root@localhost redis]# cat appendonly.aof
```

- 步骤 5：修改“appendonly.aof”文件，将最后的命令“flushall”删除

```
^D
flushall
[root@localhost redis]# vim appendonly.aof
```

- 步骤 6：在窗口 1 启动 redis，然后查询数据库内容

```
^C
[root@localhost redis]# ./bin/redis-server ./redis.conf
[root@localhost redis]# ./bin/redis-cli
127.0.0.1:6379> keys *
1) "num"
2) "name"
3) "n2"
4) "n3"          数据库还原
5) "n1"
```

第6章 redis 使用场景（了解）

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



- (1) 使用 **LPUUSH latest.comments <ID>** 命令，向 **list** 集合中插入数据
- (2) 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
- (3) 然后我们在客户端获取某一页评论时可以用下面的逻辑（伪代码）
伪代码

```
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，比如按项的次数排序，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。

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。没有做不到，只有想不到。

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。



第7章 附

1.1 redis.conf 配置详情

Redis 支持很多的参数，但都有默认值。

参数名称	描述
daemonize	默认情况下，redis 不是在后台运行的，如果需要在后台运行，把该项的值更改为 yes
pidfile	当 Redis 在后台运行的时候，Redis 默认会把 pid 文件放在 /var/run/redis.pid，你可以配置到其他地址。当运行多个 redis 服务时，需要指定不同的 pid 文件和端口
bind	指定 Redis 只接收来自于该 IP 地址的请求，如果不进行设置，那么将处理所有请求，在生产环境中最好设置该项
port	监听端口，默认为 6379
timeout	设置客户端连接时的超时时间，单位为秒。当客户端在这段时间内没有发出任何指令，那么关闭该连接
loglevel	log 等级分为 4 级，debug, verbose, notice, 和 warning。生产环境下一般开启 notice
logfile	配置 log 文件地址，默认使用标准输出，即打印在命令行终端的窗口上
databases	设置数据库的个数，可以使用 SELECT <dbid>命令来切换数据库。默认使用的数据库是 0
save	设置 Redis 进行数据库镜像的频率。 if(在 60 秒之内有 10000 个 keys 发生变化){ 进行镜像备份 } else if(在 300 秒之内有 10 个 keys 发生了变化){ 进行镜像备份 } else if(在 900 秒之内有 1 个 keys 发生了变化){ 进行镜像备份 }
rdbcompression	在进行镜像备份时，是否进行压缩
dbfilename	镜像备份文件的文件名
dir	数据库镜像备份的文件放置的路径。这里的路径跟文件名要分开配置是因为 Redis 在进行备份时，先会将当前数据库的状态写入到一个临时文件中，等备份完成时，再把该临时文件替换为上面所指定的文件，而这里的临时文件和上面所配置的备份文件都会放在这个指定的路径当中
slaveof	设置该数据库为其他数据库的从数据库
masterauth	当主数据库连接需要密码验证时，在这里指定
requirepass	设置客户端连接后进行任何其他指定前需要使用的密码。警告：因为 redis



	速度相当快，所以在一台比较好的服务器下，一个外部的用户可以在一秒钟进行 150K 次的密码尝试，这意味着你需要指定非常非常强大的密码来防止暴力破解。
maxclients	限制同时连接的客户数量。当连接数超过这个值时， redis 将不再接收其他连接请求，客户端尝试连接时将收到 error 信息。
maxmemory	设置 redis 能够使用的最大内存。当内存满了的时候，如果还接收到 set 命令， redis 将先尝试剔除设置过 expire 信息的 key ，而不管该 key 的过期时间还没有到达。在删除时，将按照过期时间进行删除，最早将要被过期的 key 将最先被删除。如果带有 expire 信息的 key 都删光了，那么将返回错误。这样， redis 将不再接收写请求，只接收 get 请求。 maxmemory 的设置比较适合于把 redis 当作于类似 memcached 的缓存来使用。
appendonly	默认情况下， redis 会在后台异步的把数据库镜像备份到磁盘，但是该备份是非常耗时的，而且备份也不能很频繁，如果发生诸如拉闸限电、拔插头等状况，那么将造成比较大范围的数据丢失。所以 redis 提供了另外一种更加高效的数据库备份及灾难恢复方式。开启 append only 模式之后， redis 会把所接收到的每一次写操作请求都追加到 appendonly.aof 文件中，当 redis 重新启动时，会从该文件恢复出之前的状态。但是这样会造成 appendonly.aof 文件过大，所以 redis 还支持了 BGREWRITEAOF 指令，对 appendonly.aof 进行重新整理。所以我认为推荐生产环境下的做法为关闭镜像，开启 appendonly.aof ，同时可以选择在访问较少的时间每天对 appendonly.aof 进行重写一次。
appendfsync	设置对 appendonly.aof 文件进行同步的频率。 always 表示每次有写操作都进行同步， everysec 表示对写操作进行累积，每秒同步一次。这个需要根据实际业务场景进行配置
vm-enabled	是否开启虚拟内存支持。因为 redis 是一个内存数据库，而且当内存满的时候，无法接收新的写请求，所以在 redis 2.0 中，提供了虚拟内存的支持。但是需要注意的是， redis 中，所有的 key 都会放在内存中，在内存不够时，只会把 value 值放入交换区。这样保证了虽然使用虚拟内存，但性能基本不受影响，同时，你需要注意的是你要把 vm-max-memory 设置到足够来放下你的所有的 key
vm-swap-file	设置虚拟内存的交换文件路径
vm-max-memory	这里设置开启虚拟内存之后， redis 将使用的最大物理内存的大小。默认为 0 ， redis 将把他所有的能放到交换文件的都放到交换文件中，以尽量少的使用物理内存。在生产环境下，需要根据实际情况设置该值，最好不要使用默认的 0
vm-page-size	设置虚拟内存的页大小，如果你的 value 值比较大，比如说你要在 value 中放置博客、新闻之类的所有文章内容，就设大一点，如果要放置的都是很小的内容，那就设小一点。
vm-pages	设置交换文件的总的 page 数量，需要注意的是， page table 信息会放在物理内存中，每 8 个 page 就会占据 RAM 中的 1 个 byte 。总的虚拟内存大小 = vm-page-size * vm-pages
vm-max-threads	设置 VM IO 同时使用的线程数量。因为在进行内存交换时，对数据有编码和解码的过程，所以尽管 IO 设备在硬件上本上不能支持很多的并发读写，但是还是如果你所保存的 vlaue 值比较大，将该值设大一些，还是能够提

	升性能的
glueoutputbuf	把小的输出缓存放在一起，以便能够在一个 TCP packet 中为客户端发送多个响应，具体原理和真实效果我不是很清楚。所以根据注释，你不是很确定的时候就设置成 yes
hash-max-zipmap-entries	在 redis 2.0 中引入了 hash 数据结构。当 hash 中包含超过指定元素个数并且最大的元素没有超过临界时， hash 将以一种特殊的编码方式（大大减少内存使用）来存储，这里可以设置这两个临界值
activerehashing	开启之后， redis 将在每 100 毫秒时使用 1 毫秒的 CPU 时间来对 redis 的 hash 表进行重新 hash ，可以降低内存的使用。当你的使用场景中，有非常严格的实时性需要，不能够接受 Redis 时不时的对请求有 2 毫秒的延迟的话，把这项配置为 no 。如果没有这么严格的实时性要求，可以设置为 yes ，以便能够尽可能快的释放内存

7.1 扩展：启动多个 Redis

- 方法 1：启动时指定端口可在一台服务器启动多个 Redis 进程。（多个 Redis 实例）

```
cd /usr/local/redis/bin
./redis-server ./redis.conf --port 6380
```

```
[root@localhost redis]# ./bin/redis-server ./redis.conf --port 6380
[root@localhost redis]# ./bin/redis-server ./redis.conf --port 6381
[root@localhost redis]# ps aux | grep -i redis
root      31579  0.0  0.1  35552  1720 ?        Ssl   21:26   0:00 ./bin/redis-server *:6380
root      31583  0.0  0.1  35552  1720 ?        Ssl   21:26   0:00 ./bin/redis-server *:6381
```

- 方式 2：复制 redis 目录，然后编写 redis.conf 修改端口【推荐使用】

- 步骤 1：拷贝 redis 目录

```
cp -r redis/ redis6380
[root@localhost local]# cp -r redis/ redis6380
[root@localhost local]# ll
总用量 44
drwxr-xr-x. 2 root root 4096 9月 23 2011 bin
drwxr-xr-x. 2 root root 4096 9月 23 2011 etc
drwxr-xr-x. 2 root root 4096 9月 23 2011 games
drwxr-xr-x. 2 root root 4096 9月 23 2011 include
drwxr-xr-x. 2 root root 4096 9月 23 2011 lib
drwxr-xr-x. 2 root root 4096 9月 23 2011 libexec
drwxr-xr-x. 3 root root 4096 7月 13 21:29 redis
drwxr-xr-x. 3 root root 4096 7月 13 21:32 redis6380
drwxr-xr-x. 2 root root 4096 9月 23 2011 sbin
drwxr-xr-x. 5 root root 4096 5月 27 01:12 share
drwxr-xr-x. 2 root root 4096 9月 23 2011 src
[root@localhost local]# ll ./redis6380/
总用量 52
drwxr-xr-x. 2 root root 4096 7月 13 21:32 bin
-rw-r--r--. 1 root root 18 7月 13 21:32 dump.rdb
-rw-r--r--. 1 root root 41404 7月 13 21:32 redis.conf
```

- 步骤 2：修改 redis.conf 文件

```
cd redis6380/
vim redis.conf
```

```
[root@localhost local]# cd redis6380/
[root@localhost redis6380]# vim redis.conf
```



```
43 # Accept connections on the specified port, default is 6379.
44 # If port 0 is specified Redis will not listen on a TCP socket.
45 port 6380
46
```

■ 步骤 3：启动多个 redis

```
cd /usr/local/
./redis/bin/redis-server ./redis/redis.conf
./redis6380/bin/redis-server ./redis6380/redis.conf
ps aux | grep -i redis
```

```
[root@localhost local]# ./redis/bin/redis-server ./redis/redis.conf
[root@localhost local]# ./redis6380/bin/redis-server ./redis6380/redis.conf
[root@localhost local]# ps aux | grep -i redis
root      31709  0.0  0.1  35552  1720 ?        ss1   21:39   0:00 ./redis/bin/redis-server *:6379
root      31713  0.0  0.1  35552  1724 ?        ss1   21:39   0:00 ./redis6380/bin/redis-server *:6380
```

● 关闭指定端口号的 Redis

```
./bin/redis-cli -p 6380 shutdown
```

```
[root@localhost redis]# ./bin/redis-cli shutdown
Could not connect to Redis at 127.0.0.1:6379: Connection refused
[root@localhost redis]# ./bin/redis-cli -p 6380 shutdown
[root@localhost redis]# ./bin/redis-cli -p 6381 shutdown
[root@localhost redis]# ps aux | grep -i redis
root      31614  0.0  0.0  5980    752 pts/2    S+   21:29   0:00 grep -i redis
```

第8章 总结