Redis 数据结构和操作

redis 不只是一个简单的键(key)-值(value)数据库，实际上它是一个数据结构服务器，支持各种类型的值。也就是说，在传统的键-值数据库中，你把字符串键与字符串值联系起来，而在 redis，值不仅限于一个简单的字符串，还可以是更复杂的数据结构。下面列出了所有 redis 支持的数据结构，下文会分别对这些结构进行介绍：

- 二进制安全字符串
- 队列(lists)：基于插入顺序有序存储的字符串元素集合。主要是链式的 list。
- 集(sets)：元素唯一的、无序的字符串元素集合。
- 有序集(sorted sets)：与 sets 相似，但是每个字符串元素都与一个被称为分数(score)的浮点数相关联。和 sets 不同的是，元素能够基于分数排序，因此可以检索某个范围内的元素（比如你可以查询前 10 个或后 10 个）。
- 哈希(hashes)：由域(fields)和值之间关系组成的映射。域和值都是字符串。这和 Ruby 或 Python 的哈希非常相似。
- 位数组（位图 bitmaps）：可以通过特殊命令，像处理位图一样地处理字符串：设置和清除某一位，统计被置 1 的位数，找到第一个被设置或没有被设置的位等。
- HyperLogLogs：这是一种概率数据结构，用于估算集的势。不要被吓到了，没那么难。本文将在下文中 HyperLogLog 章节介绍。

遇到问题的时候，理解数据结构是怎么工作的以及怎么被使用的并不是那么微不足道的事情。因此，这篇文档是一个关于 Redis 数据类型和它们常用模式的速成教材。

这里所有的例子，我们都使用 redis 客户端（redis-cli）。相对于 redis 服务器来说，这是一个简单方便的命令行控制台。

redis 的键

redis 的键是二进制安全【1】的，也说是说，你可以使用任意的二进制序列作为键，比如字符串”foo”或一个 JPEG 文件的内容。

空串也是一个有效的键。

一些关于键的其它规则：

- 太长的键不推荐。例如长度为 1024 字节的键并不好，不管是从内存角度，还是从查询键的角度。因为从数据集中查询键需要多次的键匹配步骤。即使手边的任务就是要判断一个很大的值是否存在，采用某种手段对它做 hash 是个好主意，尤其是从内存和带宽的角度去考虑。
- 太短的键通常也不推荐。如果你把键“user:1000:followers”写成“u1000flw”可能会有点问题。因为前者可读性更好，而只需要多花费一点点的空间。短的键显然占的花费的空间会小一点，因此你需要找到平衡点。
- 尽量坚持模式。例如”object-type:id”是推荐的，就像”user:1000”。点和短线常用于多个单词的场景，比如”comment:1234:reply.to”

  或”comment:1234:reply-to”。
- 键的大小不能超过 512MB。

Redis 中的字符串

Redis 中的字符串类型是可以与键关联的最简单的类型。它中 Memcached 中唯一的数据类型，也是 Redis 新手最常用的类型。

由于 Redis 的键都是字符串，那么把使用字符串为值，也就是字符串到字符串的映射。字符串数据类型可以用于许多场景，比如缓存 HTML 片段或页面。

让我们用 redis 客户端尝试一些字符串类型的使用吧（本文所有的例子都在 redis 客户端执行）。

```
> set mykey somevalue
OK
> get mykey
"somevalue"
```

正如你所看到的，**GET** 和 **SET** 命令用于设置或获取一个字符串值。需要注意的是，如果键已经存在，**SET** 会覆盖它的值，即使与这个键相关联的不是字符串类型的值。**SET** 相当于赋值。

值可以是任意类型的字符串（包含二进制数据），你也可以使用一个 jpeg 图像。值在大小不能大于 512MB。

SET 命令配上一些额外的参数，可以实现一些有趣的功能。例如，我可以要求如果键已经存在，SET 就会失败，或者相反，键已经存在时 SET 才会成功。

```
> set mykey newval nx
(nil)
> set mykey newval xx
OK
```

虽然字符串是最基础的数据类型，你仍可以对它执行一些有趣的操作，比如原子性的自增：

```
> set counter 100
OK
> incr counter
(integer) 101
```

```
> incr counter

(integer) 102

> incrby counter 50

(integer) 152
```

INCR 命令把字符串解析成一个整数，然后+1，把得到的结果作为一个新值存进去。还有其它相似的命令：**INCRBY**, **DECR**, **DECRBY**。从命令的实现原理上讲，这几个命令是相同的，只是有一点细微的差别。

为什么说 **INCR** 是原子性的呢？因为即使是多个客户端对同一个键使用 **INCR** 命令，也不会形成竞争条件。举个例子，像这样的情况是不会发生的：客户端 1 读取到键是 10，客户端 2 也读到键值是 10，它们同时对它执行自增命令，最终得到的值是 11。实际上，最终的得到的值是 12，因为当一个客户端对键值做读-自增-写的过程中，其它的客户是不能同时执行这个过程的。

有许多用于操作字符串的命令，例如 **GETSET** 命令，它给键设置一个新值，并返回旧值。比如你有一个系统，每当有一个新的访问者登陆你的网站时，使用 **INCR** 对一个键值自增。你可能想要统计每个小时的信息，却又不希望丢失每次自增操作。你可以使用 **GETSET** 命令，设置一个新值“0”，同时读取旧值。

redis 支持通过一条命令同时设置或读取多个键，这对于减少延时很有用。这就是 **MSET** 命令和 **MGET** 命令：

```
> mset a 10 b 20 c 30

OK

> mget a b c

1) "10"
```

```
2) "20"
3) "30"
```

使用 **MGET** 时，redis 返回包含多个值的数组。

更改或查询键空间

【2】

有些命令并没有指定特定的类型，但在与键空间的交互有非常有用，因此可以用于任意类型的键。

举个例子，**EXISTS** 命令返回 1 或者 0，用于表示某个给定的键在数据库中是否存在。**DEL** 命令删除键以及它对应的值而不管是什么值。

```
> set mykey hello
OK
> exists mykey
(integer) 1
> del mykey
(integer) 1
> exists mykey
(integer) 0
```

DEL 返回 1 还是 0 取决于键是（键存在）否（键不存在）被删除掉了。

有许多键空间相关的命令，但以上这两个命令和 TYPE 命令是最基本的。**TYPE** 命令的作用是返回这个键的值的类型。

```
> set mykey x
OK
> type mykey
```

```
string

> del mykey

(integer) 1

> type mykey

none
```

键的生命周期

在介绍更多更复杂的数据结构之间，我们先讨论另一个与值类型无关的特性，那就是 redis 的期限（redis expires）。最基本的，你可以给键设置一个超时时间，就是这个键的生存周期。当生存周期过去了，键会被自动销毁，就好像被用户执行过 **DEL** 一样。

一些关于 redis 期限的快速信息：

- 生存周期可以设置的时间单位从秒级到毫秒级。
- 生存周期的时间精度都是 1 毫秒。
- 关于生存周期的数据有多份且存在硬盘上，基于 Redis 服务器停止了，时间仍在流逝，这意味着 redis 存储的是 key 到期的时间。

设置生存周期是件琐碎的事情：

```
> set key some-value

OK

> expire key 5

(integer) 1

> get key (immediately)

"some-value"

> get key (after some time)
```

```
(nil)
```

键在两次调用之间消失了，这是因为第二次调用的延迟了超过 5 秒的时间。在上面的例子中，我们使用 **EXPIRE** 命令设置生命周期（它也可以用于为一个已经设置过生命周期的键重新设置生命周期，**PERSIST** 命令可以用于移除键的命令周期，使它能够长期存在）。我们还可以使用 redis 命令在创建键的同时设置生命周期。比如使用带参数的 **SET** 命令：

```
> set key 100 ex 10
OK
> ttl key
(integer) 9
```

上面这个例子中创建了一个键，它的值是字符串 100，生命周期是 10 秒。后面的 TTL 命令用于查看键的剩余时间。

如果要以毫秒为单位设置或查询键的生命周期，请查询 **PEXPIRE** 命令和 **PTTL** 命令，以及 **SET** 命令的参数列表。

redis 中的列表(lists)

要解释列表数据类型，最好先从一点理论开始。因为列表这个术语常被信息技术人员错误地使用。例如“python 列表”，并不像它的命令所提示的（链表），而是数组（实际上与 Ruby 中的数组是同一个数据类型）。

从广义上讲，列表只是元素的有序序列：10，20，1，2，3 是一个列表。但是用数组实现的列表和用链表实现的列表，它们的属性有很大的不同。

redis 的列表都是用链表的方式实现的。也就是说，即使列表中有数百万个元素，增加一个新元素到列表头部或尾部操作的执行时间是常数时间。使用 **LPUSH** 命令把一个新元素增加

到一个拥有 10 个元素的列表的头部，或是增加到一个拥有一千万个元素的列表的头部，其速度是一样的。

缺点是什么呢？通过索引访问一个元素的操作，在数组实现的列表中非常快（常数时间），但在链表实现的列表中不是那么快（与找到元素对应下标的速度成比例）。

redis 选择用链表实现列表，因为对于一个数据库来说，快速地向一个很大的列表新增元素是非常重要的。另一个使用链表的强大优势，你稍后将会看到，能够在常数时间内得到一个固定长度的 redis 列表。

快速地读取很大一堆元素的中间元素也是重要的，这时可以使用另一种数据结构，称为有序集（sorted sets）。本文后面会讲到有序集。

regis 列表第一步

LPUSH 命令把一个新的元素加到列表的左边（头部），而 **RPUSH** 命令把一个新的元素加到列表的右边（尾部）。**LRANGE** 命令从列表中提取某个范围内的元素

```
> rpush mylist A
(integer) 1
> rpush mylist B
(integer) 2
> lpush mylist first
(integer) 3
> lrange mylist 0 -1
1) "first"
2) "A"
3) "B"
```

注意，**LRANGE** 命令需要输入两个下标，即范围的第一个元素下标和最后一个元素下标。两个下标都可以是负的，意思是从尾部开始数：因此-1 是最后一个元素，-2 是倒数第二个元素，等。

正如你所见，**RPUSH** 把元素加到列表右边，**LPUSH** 把元素加到列表左边。所有命令的参数都是可变的，即你可以随意地把多个增加元素入列表的命令放到一次调用中：

```
> rpush mylist 1 2 3 4 5 "foo bar"
(integer) 9
> lrange mylist 0 -1
1) "first"
2) "A"
3) "B"
4) "1"
5) "2"
6) "3"
7) "4"
8) "5"
9) "foo bar"
```

Redis 中定义了一个重要的操作就是删除元素。删除命令可以同时从列表中检索和删除元素。你可以从左边或者右边删除元素，和从两边增加元素的方法类似：

```
> rpush mylist a b c
(integer) 3
> rpop mylist
"c"
```

```
> rpop mylist

"b"

> rpop mylist

"a"
```

我们增加和删除的三个元素，因此最后列表是空的，没有元素可以删除。如果我们尝试继续删除元素，会得到这样的结果：

```
> rpop mylist

(nil)
```

redis 返回空值说明列表中没有元素了。

列表的常见用例

列表可以用于完成多种任务，以下是两个非常有代表性的用例：

- 记住用户发布到社交网络的最新更新。
- 使用消费者-生产者模型进行进程间通信，生产生把表项（items）放进列表中，消费者（通常是工作者）消费这些 items 并执行一些行为。redis 针对这种用例有一些特殊的列表命令，既可靠又高效。

例如非常有名的 Ruby 库 **resque** 和 **sidekip**，在底层都使用了 Redis 列表来实现后台作业。

著名的社交网络 Twitter 使用 Redis 列表来获取用户发布的最新的消息。

为了一步一步地描述一个常见用例，假设要在你的主页上展示社交网络上最新分享的照片并且加速访问。

- 每当一个用户发布了一张新的照片，我们使用 **LPUSH** 命令把它的 ID 加入到列表中。
- 当用户访问这个主页，我们使用 LRANGE 0 9 获取最新加入的 10 个表项。

限制列表

很多情况下我们只想要使用列表来存储最新的几条表项，例如社交网络更新、日志或者其它。

Redis 允许我们使用列表作为一个固定集合，使用 **LTRIM** 命令，只记录最新的 N 条记录，而丢弃所有更早的记录。

LTRIM 命令和 **LRANGE** 命令相似，但不像 **LRANGE** 一样显示特定范围的元素，而是用这个范围内的值重新设置列表。所有范围外的元素都被删除了。

用个例子来说明这一点：

```
> rpush mylist 1 2 3 4 5
(integer) 5
> ltrim mylist 0 2
OK
> lrange mylist 0 -1
1) "1"
2) "2"
3) "3"
```

上面的 **LTRIM** 命令告诉 Redis 只取得列表中下标为 0 到 2 的元素，其它的都要丢弃。这就是一种简单有用的模式成为了可能：列表增加(push)元素操作+列表提取(trim)元素操作=增加一个元素同时删除一个元素使得列表元素总数有限：

```
LPUSH mylist <some element>
LTRIM mylist 0 999
```

上面的操作结合增加一个元素但只是存在 1000 个最新的元素在列表中。通过 **LRANGE** 你可以获取最新的表项而不需要记住旧的数据。

注意：由于理论上 **LRANGE** 是O（N）命令，读取从头开始或从尾开始的小范围数据是常数时间的操作。

列表中是阻塞型操作

列表的一些特性使它适合现实队列（queues），也通常作为进程间通信系统的一个基础组件：阻塞式操作。

假设你通过一个进程把元素增加到列表中，使用另一个进程对这些元素做些实际的操作。这是通常的生产者/消费者基础，你可以用下面这种简单的方法实现：

- 生产者调用 LPUSH，把元素加入列表
- 消费者调用 RPOP，把元素从列表中取出或处理

有没有可能出现这种情况，列表是空的，没有什么东西可以处理，因此 RPOP 返回 NULL。

这种情况下，消费者不得不等一会再尝试 RPOP。这就叫轮询。这并不是一个好方法，因为它有以下缺点：

1. 要求 redis 和客户端执行没有意义的命令（当列表为空是所有的请求都不会执行实际工作，只是返回 NULL）
2. 工作者在收到 NULL 之后加入一个延时，让它等待一些时间。如果让延时小一点，在两次调用 **RPOP** 之间的等待时间会比较短，这成为第一个问题的放大-调用 Redis 更加没有意义

因此 Redis 实现了命令 **BRPOP** 和 **BLPOP**，它是 **RPOP** 和 **LPOP** 的带阻塞功能的版本：当列表为空时，它们会等到一个新的元素加入到列表时，或者用户定义的等待时间到了时，才会返回。

这是 **BRPOP** 调用的一个例子，我们可以在工作者进程使用它：

```
> brpop tasks 5
```

```
1) "tasks"
2) "do_something"
```

它的意思是：等待列表中的元素，如果 5 秒还没有可用的元素。

注意，如果使用 0 作为超时时间，将会永远等待，你也可以定义多个列表而不只是一个，

这样就会同时等待多个列表，当任意一个列表收到一个元素时就会收到通知。

一些关于 **BRPOP** 需要注意的事情：

1. 客户端是按顺序被服务的：第一个等待某个列表的客户端，当列表被另一个客户端增加一个元素时，它会第一个处理。
2. 返回值与 RPOP 的不同：只得到两个元素的包含键名的数组，因为 BRPOP 和 BLPOP 因为等待多个列表而阻塞。
3. 如果时间超时了，就会返回 NULL

还有更多你应该知道的关于列表和阻塞操作的东西。我们建议你阅读以下材料：

- 可以使用 **RPOPLPUSH** 创建更安全的队列或旋转队列。
- 这个命令有一个阻塞参数，即 **BRPOPLPUSH**

自动创建和移除键

到目前为止我们的例子还没有涉及到这些情景，在增加一个元素之间创建一个空的列表，或者当一个列表没有元素时把它移除。redis 有责任删除变为空的列表，或当我们试图增加元素时创建空列表。例如 **LPUSH**

这不仅适用于列表，它可以应用于所有包含多个元素的 Redis 数据结构-集、有序集和哈希。基本上讲，我们把它的行为总结为三个规则：

1. 当我们把一个元素增加到一个集合类数据类型时，如果这个键不存在，在增加前会创建一个空的集合类数据类型。

2. 我们从一个集合类数据类型中移除一个元素时，如果值保持为空，键就会被自动删除
3. 调用一个只读命令例如 **LLEN**（返回列表的长度），或者一个移除元素的写命令但键为空，结果不会改变。[3]

规则 1 举例：

```
> del mylist

(integer) 1

> lpush mylist 1 2 3

(integer) 3
```

然而，我们不能对一个已经存在的键执行与它类型不同的操作：

```
> set foo bar

OK

> lpush foo 1 2 3

(error) WRONGTYPE Operation against a key holding the wrong
 kind of value

> type foo

string
```

规则 2 举例：

```
> lpush mylist 1 2 3

(integer) 3

> exists mylist

(integer) 1

> lpop mylist
```

```
"3"

> lpop mylist

"2"

> lpop mylist

"1"

> exists mylist

(integer) 0
```

当所有元素被取出，这个键就不存在了。

规则 3 举例：

```
> del mylist

(integer) 0

> llen mylist

(integer) 0

> lpop mylist

(nil)
```

redis 中的哈希（hashed）

redis 的哈希和我们所认识的“哈希”非常相似，是域-值对。

```
> hmset user:1000 username antirez birthyear 1977 verified
1

OK

> hget user:1000 username

"antirez"
```

```
> hget user:1000 birthyear
"1977"
> hgetall user:1000
1) "username"
2) "antirez"
3) "birthyear"
4) "1977"
5) "verified"
6) "1"
```

hash 表示对象(object)非常方便。实际上，可以放入一个 hash 的域的数量没有限制（不考虑可用内存），因此你可以在你的应用中用许多不同的方式使用哈希。

HMSET 命令为 hash 设置多个域，而 **HGET** 获取某一个域。**HMGET** 和 **HGET** 相似，但它返回由值组成的数组。

```
> hmget user:1000 username birthyear no-such-field
1) "antirez"
2) "1977"
3) (nil)
```

还有一些命令可以对单个的域执行操作，例如 **HINCRBY**：

```
> hincrby user:1000 birthyear 10
(integer) 1987
> hincrby user:1000 birthyear 10
(integer) 1997
```

你可以查看这篇文档《hash 命令全列》

把小的哈希（少量的元素，较小的值）用特殊的编码方式存放在内存中并不是什么难事，因此它们的空间效率非常高。

redis 的集（sets）

Redis 的集是字符串无序的集合。**SADD** 向集中增加一些元素。对于集合还有很多其它的操作，例如测试某个给定的元素是否存在，多个集合之间求交集、合集或者差集，等。

```
> sadd myset 1 2 3

(integer) 3

> smembers myset

1. 3

2. 1

3. 2
```

在这个例子中，我向 myset 中增加了三个元素，并让 redis 返回所有的元素。正如你所看到的，它们是无序的。每次调用，redis 都可能以任何顺序返回元素，因此在这里，用户不能对元素的顺序有要求。

redis 提供测试成员的命令。这个给定的元素是否存在？

```
> sismember myset 3

(integer) 1

> sismember myset 30

(integer) 0
```

“3”是这个集中的一员，而“30”不是。

集善于表现对象之间的关系。例如我们可以很容易使用集实现标签(tags)。处理这个问题的

一个简单的模型就是把所有要打标签的对象设置一个集。集包含相关对象的标签的 ID。

假设我们想要为新闻加标签。ID 为 1000 的新闻被打上 1，2，5 和 77 这几个标签，我们可以用一个集将这些标签 ID 与新闻关联起来：

```
> sadd news:1000:tags 1 2 5 77
(integer) 4
```

然而有时我会想要相反的关系：列表中的所有新闻都被打上一个给定的标签：

```
> sadd tag:1:news 1000
(integer) 1
> sadd tag:2:news 1000
(integer) 1
> sadd tag:5:news 1000
(integer) 1
> sadd tag:77:news 1000
(integer) 1
```

要获取一个对象的所有标签是很麻烦的。

```
> smembers news:1000:tags
1. 5
2. 1
3. 77
4. 2
```

注意：在这个例子中我们假设你还有另一个数据结构，例如 redis 的哈希，用于标签 ID 到标签名的映射。

如果使用正确的 redis 命令，可以通过并不繁琐却简单的方式去实现。例如我们可能想到同时拥有 1，2，10 和 27 标签的所有对象。我们可以使用 **SINTER** 命令执行不同集之间的求交运算。我们可以这么用：

```
> sinter tag:1:news tag:2:news tag:10:news tag:27:news
... results here ...
```

求交集运算不是唯一可以执行的操作，你还可以执行求并集运算、求差集运算、提取任意一个元素等。

提取一个元素的命令是 **SOP**，它对于模拟某些问题很方便。例如要实现一个基于网页的扑克牌游戏，你可能会把你的牌(deck)做成一个集。假设我们使用一个字符前缀来表示 C（梅花）、D（方块）、H（红心）、S（黑桃）：

```
>  sadd deck C1 C2 C3 C4 C5 C6 C7 C8 C9 C10 CJ CQ CK
   D1 D2 D3 D4 D5 D6 D7 D8 D9 D10 DJ DQ DK H1 H2 H3
   H4 H5 H6 H7 H8 H9 H10 HJ HQ HK S1 S2 S3 S4 S5 S6
   S7 S8 S9 S10 SJ SQ SK
   (integer) 52
```

现在我们给每个玩家提供 5 张牌。**SPOP** 命令移除一个随机的元素，并把它返回给客户端，因此在这个用例中是最好的操作。

然后如果我们直接对 deck 调用它，下一轮游戏我们需要把再次填写所有的牌，这还不够理想。因此在开始之前，先把集中存储的 deck 键做一个备份到 game 中。使用 **SUNIONSTORE** 来实现，把结果存到另一个集中。这个命令通常是对多个集做求并集运行的。对一个集求并集运算就是它自己，因此可以用于复制：

```
> sunionstore game:1:deck deck
```

```
(integer) 52
```

现在我已经准备好为第一个玩家发五张牌了。

```
> spop game:1:deck
"C6"
> spop game:1:deck
"CQ"
> spop game:1:deck
"D1"
> spop game:1:deck
"CJ"
> spop game:1:deck
"SJ"
```

一对 J，不太好。。。

现在是时候介绍集中元素个数的命令了。在集理论中，元素个数常被为集的势，因此这个命令是 **SCARD**。

```
> scard game:1:deck
(integer) 47
```

计算公式：52-5=47

如果你只是想得到一个随机的元素但不把它从集中删除，**SRANDMEMBER** 命令适合这个任务。它还可以提供返回重复元素或非重要元素的功能。

Redis 的有序集

有序集像一种将集和哈希混合的数据类型。像集一样，有序集由唯一的不重复的字符串元素组成。因此某种意义上说，有序集也是一个集。

集中的元素是无序的，而有序集中的元素都基于一个相关联的浮点值排序。这个浮点值称为分数(score)（每个元素都映射到一个值，因此和哈希相似）。

此外，有序集中的元素是按顺序取的（它们不是按照要求排序的，排序是这个数据结构用于表现有序集的一个特性【4】）。它们按照下面的规则排序：

- 假设 A 和 B 是分值不同的两个元素，如果 A 的分数>B 的分数，则 A>B
- 假设 A 和 B 的分值相同，如果字符串 A 的字典序大于字符串 B 的字典序，则 A>B。

  A 和 B 两个字符串不可能相同，因为有序集的元素是唯一的。

我们从一个简单的例子开始，向有序集增加一些黑客的名字，将它们的出生年份作为“分数”。

```
> zadd hackers 1940 "Alan Kay"
(integer) 1
> zadd hackers 1957 "Sophie Wilson"
(integer 1)
> zadd hackers 1953 "Richard Stallman"
(integer) 1
> zadd hackers 1949 "Anita Borg"
(integer) 1
> zadd hackers 1965 "Yukihiro Matsumoto"
(integer) 1
> zadd hackers 1914 "Hedy Lamarr"
(integer) 1
```

```
> zadd hackers 1916 "Claude Shannon"
(integer) 1
> zadd hackers 1969 "Linus Torvalds"
(integer) 1
> zadd hackers 1912 "Alan Turing"
(integer) 1
```

正如你所见，**ZADD** 和 **SADD** 相似，但是需要一个额外的参数（位置在要加的元素之前），这就是分数。**ZADD** 也是参数可变的，你可以随意地定义多个“分数-值”对，虽然上面的例子没有这么写。

要求有序集中的黑客名单按他们的出生年份排序是没有意义的，因为它们已经是这样的了。实现细节：有序集是基于一个双端口数据结构实现的，包含一个跳跃表和一个哈希表。因此增加一个元素的执行时间是 O(log(N))。这很好，当我们请求有序的元素时不需要其它的工作，它们已经是排序的了：

```
> zrange hackers 0 -1
1) "Alan Turing"
2) "Hedy Lamarr"
3) "Claude Shannon"
4) "Alan Kay"
5) "Anita Borg"
6) "Richard Stallman"
7) "Sophie Wilson"
8) "Yukihiro Matsumoto"
9) "Linus Torvalds"
```

注意：0 和-1 的意思是从下标为 0 的元素开始到最后一个元素（这里的-1 和 LRANGE 命令中的-1 一样）。

如果想要反向排序，从最年轻到最老呢？使用 **ZREVERANGE** 代替 **ZRANGE**。

```
> zrevrange hackers 0 -1
1) "Linus Torvalds"
2) "Yukihiro Matsumoto"
3) "Sophie Wilson"
4) "Richard Stallman"
5) "Anita Borg"
6) "Alan Kay"
7) "Claude Shannon"
8) "Hedy Lamarr"
9) "Alan Turing"
```

也可以同时返回分数，使用 **WITHSCORES** 参数：

```
> zrange hackers 0 -1 withscores
1) "Alan Turing"
2) "1912"
3) "Hedy Lamarr"
4) "1914"
5) "Claude Shannon"
6) "1916"
7) "Alan Kay"
8) "1940"
```

```
9) "Anita Borg"
10) "1949"
11) "Richard Stallman"
12) "1953"
13) "Sophie Wilson"
14) "1957"
15) "Yukihiro Matsumoto"
16) "1965"
17) "Linus Torvalds"
18) "1969"
```

基于范围的操作

有序集的功能强大远不止这些。它还可以基于范围操作。我们要取得所有出生年份早于（包括）1950 年的人，就使用 ZRANGEBYSCORE 命令来实现：

```
> zrangebyscore hackers -inf 1950
1) "Alan Turing"
2) "Hedy Lamarr"
3) "Claude Shannon"
4) "Alan Kay"
5) "Anita Borg"
```

我们请求 Redis 返回所有分数在无限到 1950（两边都是闭区间）之间的元素。

也可以移除某个范围内的元素。我们要从有序集中移除所有出生年份在 1940 和 1960 之间的黑客：

```
> zremrangebyscore hackers 1940 1960
```

```
(integer) 4
```

ZREMRANGEBYSCORE 命令的名字也许不是很好，但是真的很有用，它返回被移除的元素的个数。

另一个为有序集元素定义的非常有用的操作是获取排名（get-rank）操作。可以询问一个元素在它的有序集中的位置。

```
> zrank hackers "Anita Borg"
(integer) 4
```

ZREVRANK 命令也可以获取排名，不过元素是逆序排序的。

字典序的分数

最近的 Redis 2.8 版本引入了一个新特性，假设有序集中所有元素的分数相同（使用 C 语言的 memcmp 函数来比较元素，这样保证每个 redis 实例都会返回相同的结果）的情况下，允许按照字典序获得范围。【5】

针对字典序范围操作的主要命令是 **ZRANGEBYLEX**、**ZREVRANGEBYLEX**、**ZREMRANGEBYLEX** 和 **ZLEXCOUNT**。

举个例子，我们再次把所有著名黑客加入到列表中，但这一次所有元素的分数都是 0：

```
> zadd hackers 0 "Alan Kay" 0 "Sophie Wilson" 0 "Richard Stallman" 0
  "Anita Borg" 0 "Yukihiro Matsumoto" 0 "Hedy Lamarr" 0 "Claude Shannon"
  0 "Linus Torvalds" 0 "Alan Turing"
```

基于有序集的排序规则，它们是字典序排序的：

```
> zrange hackers 0 -1
```

```
1) "Alan Kay"
2) "Alan Turing"
3) "Anita Borg"
4) "Claude Shannon"
5) "Hedy Lamarr"
6) "Linus Torvalds"
7) "Richard Stallman"
8) "Sophie Wilson"
9) "Yukihiro Matsumoto"
```

我们可以使用 ZRANGEBYLEX 命令请求字典序的范围：

```
> zrangebylex hackers [B [P
1) "Claude Shannon"
2) "Hedy Lamarr"
3) "Linus Torvalds"
```

范围是开区间还是闭区间都可以（由第一个字符决定），字符串正无穷和负无穷分别通过+和-来定义。更多信息请查看文档。

这个特性很重要，它使得我们使用有序集作为一个通常索引。举个例子，如果你想要使用一个 128 位的无符号整数来为元素索引，你所要做的只是把元素加入到一个有序集并设置一个相同的分数（比如 0）以及一个由 128 位数值组成的 8 字节前缀。由于数值是大端的，字典序的顺序（原始字节序）实际上是数字序的，你可以在 128 位空间请求范围，以前缀降序返回元素的值。

如果你想查看这个特性更严谨的演示，请查看 Redis autocomplete demo.

更新分数：排行榜

这是在切换到下一个话题之前最后一个关于有序集的点。有序集的分数可以随时被更新。只需要对一个在有序集中已经存在的元素执行 **ZADD** 就可以在 O(log(N))更新它的分数（和位置）。同样的，当会经常更新时使用有序集非常合适。

这个特性的一个通常用例是排行榜。典型的应用是 Facebook 的一个游戏，你可以使用户基于它们的高分排序，增加获取排名的操作，在排行榜上显示前 N 个用户及用户排名（例：你是第 4932 好的分数）

位图

位图其实不是一个真正的数据类型，只是在这种字符串类型上有一系列基于位的操作。由于字符串是二进制安全的，最大长度是 512MB，因此可以设置多达 2^32 种不同的位串。

位操作分为两类，一种是针对某一个位的常数时间操作，如果设置某个位为 1 或 0，或获取某个位的值。另一种是对所有位的操作，例如计算一个给定范围内被设置为 1 的位的个数（如人数统计）。

位图一个最大的优势就是存储信息时非常少空间。例如一个系统里面每个用户用一个不同的递增的用户 ID 表示，可以记录 40 亿用户的某个信息（这个用户是否想要接收邮件）只需要 512M 的内存。

通过 **SETBIT** 命令和 **GETBIT** 命令设置或获取位：

```
> setbit key 10 1
(integer) 1
> getbit key 10
(integer) 1
> getbit key 11
(integer) 0
```

SETBIT 命令把第一个参数作为位的序号，第二个参数作为这个位要设置的值，只能是 1 或 0。如果地址位大于当前字符串的长度，这个命令会自动扩充字符串。

GETBIT 只返回位于某个位置的值。超过长度的位（地址位大于字符串的长度）将必然得到 0。

这三个命令是对所有位的操作：

1. **BITOP**：执行不同字符串之间的位操作。包括 AND、OR、XOR 和 NOT。
2. **BITCOUNT**：执行统计操作，返回位集中 1 的个数
3. **BITPOS**：找到第一个被设置为 0 或 1 的位置

BITPOS 和 **BITCOUNT** 都可以接受位的地址范围作为参数，这样就不对对整个字符串做操作。下面是一个关于 **BITCOUNT** 的简单例子：

```
> setbit key 0 1
(integer) 0
> setbit key 100 1
(integer) 0
> bitcount key
(integer) 2
```

位图的通常用法：

- 各种实时分析
- 存储要求空间时间高效的与对象 ID 关联的二进制信息。

假如你想知道你的网页用户中每天访问最长的时间【6】，你从开始记录天数，你把你的网页公开化的第一天开始，计数为 0，然后每当有用户用户网页，使用 **SETBIT** 设置一位。位的下标可以简单的认为是当前系统时间，减去第一天得到偏移，然后除以 3600*24。

使用这种方式，对于每一个用户，你有一个包含每天访问信息的字符串来表示。使用 **BITCOUNT** 命令可以得到某个用户访问网页的天数。而使用多个 **BITOPs** 调用，可能简单地获取和分析位图，就很容易地计算出 longest streak。

为了共享数据或避免对一个很大的键操作，bitmap 可以分裂成多个键。要把一个位图分裂成多个键而不是全部设置到一个键里面去，一个比较麻烦的方法就是让每个键存储 M 位，键名为键的数量/M，第 N 个位在这个键里的地址是数的位置%M。

HyperLogLogs

HyperLogLogs 是一个概率性的数据结构，用于计算特别的东西（技术上常用于估算一个集的势）。通过计算一个特别的表项需要使用相对于要计算的表项本身来说很大的内存，因为需要记住你已经见过的元素，以免重复计算。然后有一系列算法可以按精度使用内存：在 redis 的实现中，你最终得到一个标准错误的估计测量结果的可能性少于 1%【7】。这个算法的神奇之处在于你再大需要相对于要计算的表项本身来说很大的内存空间，可能是只使用一个常数数量的空间。最坏情况下 12K 字节，如果元素少的话，这个空间会更小。

Redis 中的 HLL，从技术上讲，它是一个完全不同的数据结构，但像字符串一样编码，因此你可以使用 **GET** 来串行化一个 HLL，使用 **SET** 并行到服务器。

从概念上讲，HLL 的接口使用集来完成相同的工作。你会使用 SADD 把每一个观测值写入集中，使用 **SCARD** 来查询集中的元素个数。因为 **SADD** 不会重复添加一个已存在的元素，因此集中的元素是唯一的。

但是你不能把一个表项真正地添加入一个 HLL，因为数据结构只包含并不真正存在的元素的状态，API 是一样的：

- 每当你看见一个新的元素，使用 **PFADD** 计数增加
- 每当你想要恢复当前的近似值，使用 **PFCOUNT**。【8】

```
> pfadd hll a b c d
(integer) 1
> pfcount hll
(integer) 4
```

使用这种数据结构的一个例子统计在一个调整中用户每天执行的请求数。

Redis 还可以对 HLL 执行求并集操作，请查询完整文件获取更多信息。

其它显著的特性

关于 redis 的接口，还有其它一些重要的信息不能放在这个文档中，但是非常值得引起你的注意：

- 可以递增地迭代键空间
- 可以在服务器端运行 LUA 脚本获取潜在因素和带宽
- redis 还是一个发布-订阅型服务器