应用 3：节衣缩食 —— 位图

在我们平时开发过程中，会有一些 bool 型数据需要存取，比如用户一年的签到记录，签了是 1，没签是 0，要记录 365 天。如果使用普通的 key/value，每个用户要记录 365 个，当用户上亿的时候，需要的存储空间是惊人的。

为了解决这个问题，Redis 提供了位图数据结构，这样每天的签到记录只占据一个位，365 天就是 365 个位，46 个字节 (一个稍长一点的字符串) 就可以完全容纳下，这就大大节约了存储空间。

位图不是特殊的数据结构，它的内容其实就是普通的字符串，也就是 byte 数组。我们可以使用普通的 get/set 直接获取和设置整个位图的内容，也可以使用位图操作 getbit/setbit 等将 byte 数组看成「位数组」来处理。

当我们要统计月活的时候，因为需要去重，需要使用 set 来记录所有活跃用户的 id，这非常浪费内存。这时就可以考虑使用位图来标记用户的活跃状态。每个用户会都在这个位图的一个确定位置上，0 表示不活跃，1 表示活跃。然后到月底遍历一次位图就可以得到月度活跃用户数。不过这个方法也是有条件的，那就是 userid 是整数连续的，并且活跃占比较高，否则可能得不偿失。

本节略显枯燥，如果读者看的有点蒙，这是正常现象，读者可以跳过阅读下一节。以老钱的经验，在面试中有 Redis 位图使用经验的同学很少，如果你对 Redis 的位图有所了解，它将会是你的面试加分项。

基本使用

Redis 的位数组是自动扩展，如果设置了某个偏移位置超出了现有的内容范围，就会自动将位数组进行零扩充。

接下来我们使用位操作将字符串设置为 hello (不是直接使用 set 指令)，首先我们需要得到 hello 的 ASCII 码，用 Python 命令行可以很方便地得到每个字符的 ASCII 码的二进制值。

```
>>> bin(ord('h'))
'0b1101000'    # 高位 -> 低位
>>> bin(ord('e'))
'0b1100101'
>>> bin(ord('l'))
'0b1101100'
>>> bin(ord('l'))
'0b1101100'
>>> bin(ord('o'))
'0b1101111'
```

接下来我们使用 redis-cli 设置第一个字符，也就是位数组的前 8 位，我们只需要设置值为 1 的位，如上图所示，h 字符只有 1/2/4 位需要设置，e 字符只有 9/10/13/15 位需要设置。值得注意的是位数组的顺序和字符的位顺序是相反的。

```
127.0.0.1:6379> setbit s 1 1
(integer) 0
127.0.0.1:6379> setbit s 2 1
(integer) 0
127.0.0.1:6379> setbit s 4 1
(integer) 0
127.0.0.1:6379> setbit s 9 1
(integer) 0
127.0.0.1:6379> setbit s 10 1
(integer) 0
127.0.0.1:6379> setbit s 13 1
(integer) 0
127.0.0.1:6379> setbit s 15 1
(integer) 0
127.0.0.1:6379> get s
"he"
```

上面这个例子可以理解为「零存整取」，同样我们还也可以「零存零取」，「整存零取」。「零存」就是使用 setbit 对位值进行逐个设置，「整存」就是使用字符串一次性填充所有位数组，覆盖掉旧值。

零存零取

```
127.0.0.1:6379> setbit w 1 1
(integer) 0
127.0.0.1:6379> setbit w 2 1
(integer) 0
127.0.0.1:6379> setbit w 4 1
(integer) 0
127.0.0.1:6379> getbit w 1  # 获取某个具体位置的值
0/1
(integer) 1
127.0.0.1:6379> getbit w 2
(integer) 1
127.0.0.1:6379> getbit w 4
(integer) 1
127.0.0.1:6379> getbit w 5
(integer) 0
```

整存零取

```
127.0.0.1:6379> set w h  # 整存
(integer) 0
127.0.0.1:6379> getbit w 1
(integer) 1
127.0.0.1:6379> getbit w 2
(integer) 1
127.0.0.1:6379> getbit w 4
(integer) 1
127.0.0.1:6379> getbit w 5
(integer) 0
```

如果对应位的字节是不可打印字符，redis-cli 会显示该字符的 16 进制形式。

```
127.0.0.1:6379> setbit x 0 1
(integer) 0
127.0.0.1:6379> setbit x 1 1
(integer) 0
127.0.0.1:6379> get x
"\xc0"
```

统计和查找

Redis 提供了位图统计指令 bitcount 和位图查找指令 bitpos，bitcount 用来统计指定位置范围内 1 的个数，bitpos 用来查找指定范围内出现的第一个 0 或 1。

比如我们可以通过 bitcount 统计用户一共签到了多少天，通过 bitpos 指令查找用户从哪一天开始第一次签到。如果指定了范围参数[start，end]，就可以统计在某个时间范围内用户签到了多少天，用户自某天以后的哪天开始签到。

遗憾的是， start 和 end 参数是字节索引，也就是说指定的位范围必须是 8 的倍数，而不能任意指定。这很奇怪，我表示不是很能理解 Antirez 为什么要这样设计。因为这个设计，我们无法直接计算某个月内用户签到了多少天，而必须要将这个月所覆盖的字节内容全部取出来 (getrange 可以取出字符串的子串) 然后在内存里进行统计，这个非常繁琐。

接下来我们简单试用一下 bitcount 指令和 bitpos 指令:

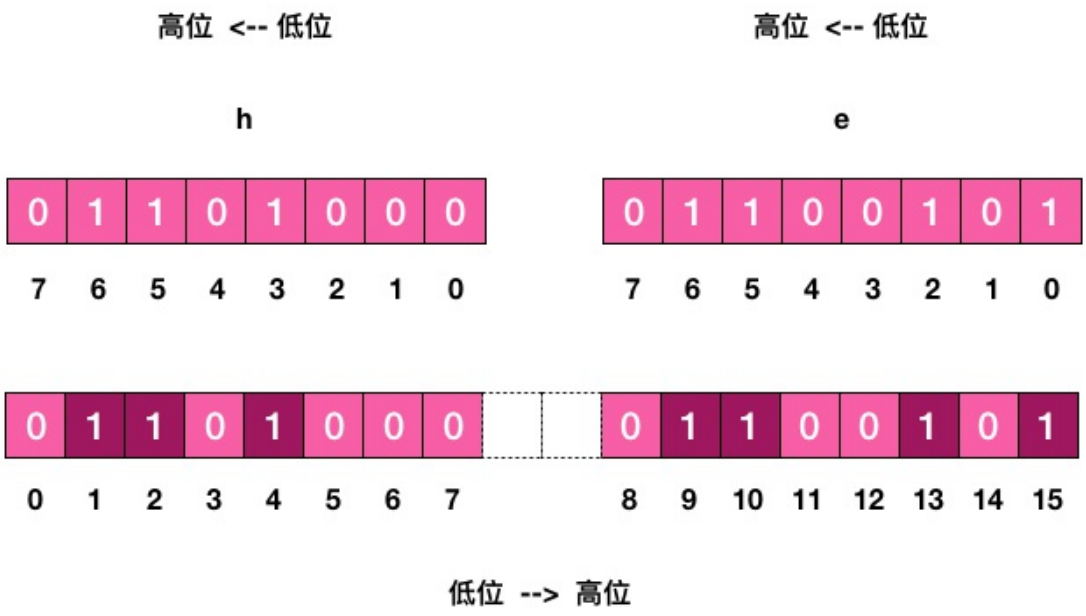
```
127.0.0.1:6379> set w hello
OK
127.0.0.1:6379> bitcount w
(integer) 21
127.0.0.1:6379> bitcount w 0 0  # 第一个字符中 1 的位数
(integer) 3
127.0.0.1:6379> bitcount w 0 1  # 前两个字符中 1 的位数
(integer) 7
127.0.0.1:6379> bitpos w 0  # 第一个 0 位
(integer) 0
127.0.0.1:6379> bitpos w 1  # 第一个 1 位
(integer) 1
127.0.0.1:6379> bitpos w 1 1 1  # 从第二个字符算起，第一个 1 位
(integer) 9
127.0.0.1:6379> bitpos w 1 2 2  # 从第三个字符算起，第一个 1 位
(integer) 17
```

魔术指令 bitfield

前文我们设置 (setbit) 和获取 (getbit) 指定位的值都是单个位的，如果要一次操作多个位，就必须使用管道来处理。

不过 Redis 的 3.2 版本以后新增了一个功能强大的指令，有了这条指令，不用管道也可以一次进行多个位的操作。

bitfield 有三个子指令，分别是 get/set/incrby，它们都可以对指定位片段进行读写，但是最多只能处理 64 个连续的位，如果超过 64 位，就得使用多个子指令，bitfield 可以一次执行多个子指令。



接下来我们对照着上面的图看个简单的例子:

```
127.0.0.1:6379> set w hello
OK
127.0.0.1:6379> bitfield w get u4 0  # 从第一个位开始取 4 个位，结果是无符号数 (u)
(integer) 6
127.0.0.1:6379> bitfield w get u3 2  # 从第三个位开始取 3 个位，结果是无符号数 (u)
(integer) 5
127.0.0.1:6379> bitfield w get i4 0  # 从第一个位开始取 4 个位，结果是有符号数 (i)
1) (integer) 6
127.0.0.1:6379> bitfield w get i3 2  # 从第三个位开始取 3 个位，结果是有符号数 (i)
1) (integer) -3
```

所谓有符号数是指获取的位数组中第一个位是符号位，剩下的才是值。如果第一位是 1，那就是负数。无符号数表示非负数，没有符号位，获取的位数组全部都是值。有符号数最多可以获取 64 位，无符

号数只能获取 63 位 (因为 Redis 协议中的 integer 是有符号数，最大 64 位，不能传递 64 位无符号值)。如果超出位数限制，Redis 就会告诉你参数错误。

接下来我们一次执行多个子指令:

```
127.0.0.1:6379> bitfield w get u4 0 get u3 2 get
i4 0 get i3 2
1) (integer) 6
2) (integer) 5
3) (integer) 6
4) (integer) -3
```

wow，很魔法有没有!

然后我们使用 set 子指令将第二个字符 e 改成 a，a 的 ASCII 码是 97，返回旧值。

```
127.0.0.1:6379> bitfield w set u8 8 97  # 从第 9
个位开始，将接下来的 8 个位用无符号数 97 替换
1) (integer) 101
127.0.0.1:6379> get w
"hallo"
```

再看第三个子指令 incrby，它用来对指定范围的位进行自增操作。既然提到自增，就有可能出现溢出。如果增加了正数，会出现上溢，如果增加的是负数，就会出现下溢出。Redis 默认的处理是折返。如果出现了溢出，就将溢出的符号位丢掉。如果是 8 位无符号数 255，加 1 后就会溢出，会全部变零。如果是 8 位有符号数 127，加 1 后就会溢出变成 -128。

接下来我们实践一下这个子指令 incrby :

```
127.0.0.1:6379> set w hello
OK
127.0.0.1:6379> bitfield w incrby u4 2 1  # 从第三个位开始，对接下来的 4 位无符号数 +1
1) (integer) 11
127.0.0.1:6379> bitfield w incrby u4 2 1
1) (integer) 12
127.0.0.1:6379> bitfield w incrby u4 2 1
1) (integer) 13
127.0.0.1:6379> bitfield w incrby u4 2 1
1) (integer) 14
127.0.0.1:6379> bitfield w incrby u4 2 1
1) (integer) 15
127.0.0.1:6379> bitfield w incrby u4 2 1  # 溢出折返了
1) (integer) 0
```

bitfield 指令提供了溢出策略子指令 overflow，用户可以选择溢出行为，默认是折返 (wrap)，还可以选择失败 (fail) 报错不执行，以及饱和截断 (sat)，超过了范围就停留在最大最小值。overflow 指令只影响接下来的第一条指令，这条指令执行完后溢出策略会变成默认值折返 (wrap)。

接下来我们分别试试这两个策略的行为

饱和截断 SAT

```
127.0.0.1:6379> set w hello
OK
127.0.0.1:6379> bitfield w overflow sat incrby u4
2 1
1) (integer) 11
127.0.0.1:6379> bitfield w overflow sat incrby u4
2 1
1) (integer) 12
127.0.0.1:6379> bitfield w overflow sat incrby u4
2 1
1) (integer) 13
127.0.0.1:6379> bitfield w overflow sat incrby u4
2 1
1) (integer) 14
127.0.0.1:6379> bitfield w overflow sat incrby u4
2 1
1) (integer) 15
127.0.0.1:6379> bitfield w overflow sat incrby u4
2 1  # 保持最大值
1) (integer) 15
```

失败不执行 FAIL

```
127.0.0.1:6379> set w hello
OK
127.0.0.1:6379> bitfield w overflow fail incrby
u4 2 1
1) (integer) 11
127.0.0.1:6379> bitfield w overflow fail incrby
u4 2 1
1) (integer) 12
127.0.0.1:6379> bitfield w overflow fail incrby
u4 2 1
1) (integer) 13
127.0.0.1:6379> bitfield w overflow fail incrby
u4 2 1
1) (integer) 14
127.0.0.1:6379> bitfield w overflow fail incrby
u4 2 1
1) (integer) 15
127.0.0.1:6379> bitfield w overflow fail incrby
u4 2 1  # 不执行
1) (nil)
```

思考 & 作业

1. 文中我们使用位操作设置了 he 两个字符，请读者将完整的 hello 单词中 5 个字符都使用位操作设置一下。
2. bitfield 可以同时混合执行多个 set/get/incrby 子指令，请读者尝试完成。