扩展类型之GEO

说明

```
什么是GEO？你会点外卖，你会网上叫车吗，会使用附近的人，摇一摇这样的功能吗？
那么是怎么通过我们的距离，去定位自己和别人，自己和车子的距离的呢？
怎么知道这个餐馆距离我们最近，怎么知道这个车子离我们最近呢？ _ 业内大部分都是使用GEOHash的方式。
Redis基于这种场景的广度，也推出了这样一种类型。（Redis 3.2 版本以后开始支持）

GEO，可以将用户给定的地理位置信息储存起来。
名字取自业界通用的地理位置距离排序算法GeoHash，
将二维的经纬度数据映射到一维的整数，也就是挂载到一条线上，
方便计算两点之间的距离。
实际的内部结构是zset（GEO是对zset的拓展）。

存储的是经纬度的信息 —— 转化为字符串，前缀匹配的越多，距离越相近。
```

命令

```
* 在开始之前，先搜索几个经维度数据
  * ccut  （经度，纬度）——> （125.39，43.99）
  * ccut0 （经度，纬度）——> （125.28，43.85）

1. geoadd geoKeyName longitude latitude member [longitude latitude member ...] ：设置geoKeyName对应的经度，纬度，名称，支持查找多个。
  如：
    127.0.0.1:6379> geoadd geo1 125.39 43.99 ccut  返回(integer) 1成功
    127.0.0.1:6379> geoadd geo1 125.28 43.85 ccut0 返回(integer) 1成功
* 由于geo的底层是zset，所以我们可以使用zset的命令进行一些查询支持。
  如：
         127.0.0.1:6379> zrange geo1 0 -1 withscores
         1) "ccut0"
         2) "4266334758699953"
         3) "ccut"
         4) "4266521082223420"
2. geodist geoKeyName member1 member2 [unit] ：求两个地点之间的相对距离,unit指定单位，可以设置为m（米）/km（千米）/mi（英里）/mt（英尺）
  如：
    127.0.0.1:6379> geodist geo1 ccut ccut0 km
    "17.8927"
3. geopos geoKeyName member [member ...] ：返回member的经纬度，出现少许误差，可以接受
  如：
     127.0.0.1:6379> geopos geo1 ccut
        1) 1) "125.39000183343887329"
           2) "43.99000080716865568"
4. geohash geoKeyName member [member...] ：对member执行geohash算法，得到hash字符串
  如：
```

```
     127.0.0.1:6379> geohash geo1 ccut
        1) "wzc4m24hqf0"

* 再增加几个地点的经纬度信息：
  * tian ：(经度, 纬度) -> (116.2706, 37.69234)
  * wei ：(经度, 纬度) -> (118.24239, 33.96271)
127.0.0.1:6379> geoadd geo1 116.2706 37.69234 tian 118.24239 33.96271 wei 返回：(integer) 2
127.0.0.1:6379> geodist geo1 wei tian km  公里数："451.3042"

5. georadius geoKeyName longitude latitude radius m|km|ft|mi [withcoord] [withdist] [withhash]：在geoKeyName中，距离(longitude, latitude)在以radius为半径的范围内的地点名称
  如：以修正大厦(125.261292, 43.794174)为圆心，20km为半径做圆，在该区域内的地点查询：
        127.0.0.1:6379> georadius geo1 125.261292 43.794174 40 km withcoord withdist
        1) 1) "ccut0"
           2) "6.3883"
           3) 1) "125.27999907732009888"
              2) "43.85000055337488334"
        2) 1) "ccut"
           2) "24.1008"
           3) 1) "125.39000183343887329"
              2) "43.99000080716865568"
   * 这就是附近的xx的实现方式。

---
命令总结：
* 基础操作 ：geoadd / geopos / geodist
* 获取定位： gethash (拿到结果去geohash.org网站查询)
* 查询附近： georadius / georadiusbymember
```

原理

```
映射算法，将地球看成一个二维平面，划分成一系列正方形的方格，
所有地图坐标都被放置于唯一的方格中，然后进行整数编码(如切蛋糕法)，编码越接近的方格距离越近。
```

扩展类型之位图

说明

话说我们在平时的开发中，有一些对于boolean类型的存储需求，比如说，要记录用户在一年之中的签到次数，如果签了是1，没签是0，那要记录365个，如果使用最普通的key-value，那么每个用户都要记录365个，这个数量是非常庞大的。
但实际上，我们只需要记录他每一天的状态就可以了，类似于这种，只需要记录在某一个节点的状态的时候，我们可以使用“位图”去记录，也就是使用字符数组去记录，这个字符数组，本质上就是一个一个boolean值,true或false。而这种字符数组对应的其实也是字符串的一系列操作。
我们可以通过“零存零取，整存整取，整存零取”等等操作， 位图和字符串将会有一个相互关联的密切关系。

BitMap 就是一个byte数组，元素中每一个 bit 位用来表示元素在当前节点对应的状态，
实际上底层也是通过对字符串的操作来实现，对应开发中boolean类型数据的存取。

命令

```
* 选取 “小写字母m” 二进制“ 0110 1101 ”
* 我们来存储这个小写字母，并返回，通过：setbit bitKeyName offset value
  * 如，存储m(5 个 1)：
        127.0.0.1:6379> setbit m 1 1
        (integer) 0
        127.0.0.1:6379> setbit m 2 1
        (integer) 0
        127.0.0.1:6379> setbit m 4 1
        (integer) 0
        127.0.0.1:6379> setbit m 5 1
        (integer) 0
        127.0.0.1:6379> setbit m 7 1
        (integer) 0
        127.0.0.1:6379> get m
        "m"
        * get m : 零存整取，就拿到m的值了。
        * get bitKeyName key offset : 零存零取
          127.0.0.1:6379> getbit m 7
            (integer) 1
        * “小写字母n” 二进制“ 0110 1110 ”
          127.0.0.1:6379> set n n
          OK
          127.0.0.1:6379> getbit n 1
          (integer) 1
          * 我们整存n，零取 1号位置的数：整存零取
* 其实“位图”本质上是帮我们节省存储空间的，因为我们如果使用最普通的字符串来存，可能消耗巨大，存储空间惊人。如果我们只是记录有或没有，签到或没有签到，存在或没存在这种状态的时候，我们用“位图”来记录是非常的便捷的。

* bitcount bitKeyName [start end] : 统计“位图”中有几个 1 . [start end]表示字符的个数
  比如：
        127.0.0.1:6379> bitcount n
        (integer) 5
        127.0.0.1:6379> set key1 "hello"
        OK
```

```
        127.0.0.1:6379> bitcount key1 0 0  ( 0 0 表示第一个字符即 h 即统计h的二进制1的个数)
        (integer) 3
        127.0.0.1:6379> bitcount key1 0 1  ( 0 1 表示前两个字符即 he 即统计he的二进制1的个数)
        (integer) 7
---
命令汇总:
* 基础操作 ： setbit key offset value  (offset 必须是数字，代表数组下标，value 只能是0或者1，代表布尔型)
* CRUD操作：setbit / getbit
* 统计和查找操作： bitcount / bitpos
* 批量操作：bitfield (三个子指令 get set incrby)
```

原理

```
位数组是自动扩展的，可以直接得到字符串的ascii码，是为整存零取，也可以零存零取或零存整取。
如果对应的字节是不可打印字符，会显示该字符的十六进制。
```

扩展类型之HyperLogLog

说明

```
Redis的基数统计，这个结构可以非常省内存的去统计各种计数。它是一个基于基数估算的算法，但并不绝对准确，标准误差是 0.81% 。HyperLogLog数据结构的发明人是Philippe Flajolet，pf是名字首字母缩写，所以我们这个命令的开头也是pf。
它能解决什么问题呢？ 这时候你开发一个网站，产品经历找你要这个网站上的UV和PV
  * UV，独立访客，是说如果可以区分的话，同一个用户不论这一天访问了多少次网站，都记为1次在UV里，也就是一天的活跃人数，就是我们的UV。
  * PV，页面访问数，不管多少人来访问这个页面，只要访问一次就记录一次，这是一个访问量或者说点击量的统计。
  * 我们这个页面一天有多少人访问，说的就是UV；我们这个页面一天有多少次访问，说的就是PV。

现在我要统计，这个网站，一天有多少人访问，我们就可以使用这个HyperLogLog

我们发现，对于UV的统计，和PV的不同之处在于，PV只要有访问就加加，最终的结果在24点的时候就生成了，但是UV是需要去重的，每一次过来的时候，都要判断我今天有没有对它进行过统计，如果没有统计，如果统计不重复统计。

如果没有HyperLogLog，让我们去实现，我们很容易就想到了set，将用户的ID扔进set,自动去重，最后统计一下set的长度就可以了。但是这样有个问题，就需要记录用户的唯一值，set的大小有可能非常的大，像淘宝百度这样体量的网站，再用set去记录UV，显然不可行。这个时候我们就可以使用这个HyperLogLog。
```

命令

```
pfadd pfKeyName element [element ...] : 将element存入pfKeyName,
通过pfcount pfKeyName统计访问次数
  127.0.0.1:6379> pfadd uv u1
    (integer) 1
    127.0.0.1:6379> pfcount uv
    (integer) 1
    127.0.0.1:6379> pfadd uv u2
    (integer) 1
    127.0.0.1:6379> pfcount uv
    (integer) 2
    127.0.0.1:6379> pfadd uv u3
    (integer) 1
    127.0.0.1:6379> pfcount uv
    (integer) 3
    127.0.0.1:6379> pfcount uv
    (integer) 3
   * 我们看，显然，当已经统计过了，再次统计的时候不会重复的统计。
   * 当数据量比较小的时候，还是比较精准的，但是数据量大了的时候，就会显示出标准误差了。
---
命令汇总:
* 计数操作 ： pfadd、pfcount
* 累加操作:  pfmerge + destkey sourcekey [sourcekey ...]
```

原理

```
HyperLogLog最大占用12KB的存储空间。
当计数比较小时，使用稀疏矩阵存储，占用空间很小，
在变大到超过阈值时，会转变成稠密矩阵，占用12KB。
算法：给定一系列的随机整数，记录低位连续0位的最大长度k，通过k可以估算出随机数的数量N。
```