<https://mp.weixin.qq.com/s/uPob9_VOWIMN2t_9K92f0g>

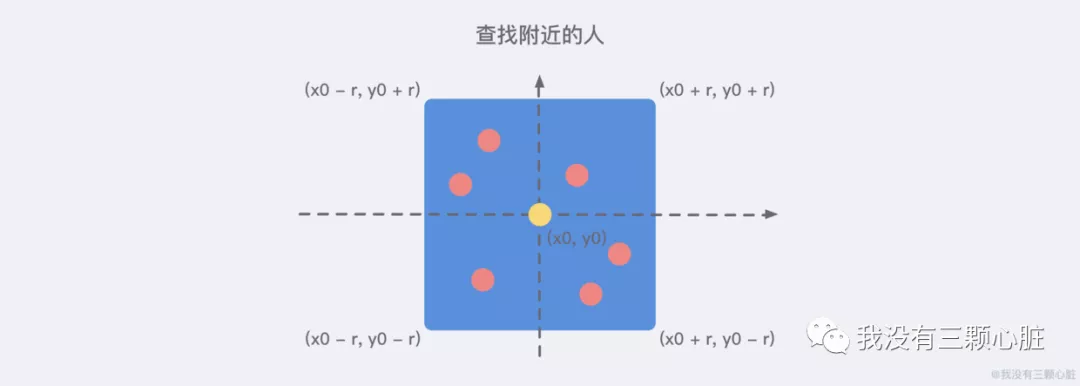
# Redis(6)——GeoHash查找附近的人

像微信 **"附近的人"**，美团 **"附近的餐厅"**，支付宝共享单车 **"附近的车"** 是怎么设计实现的呢？

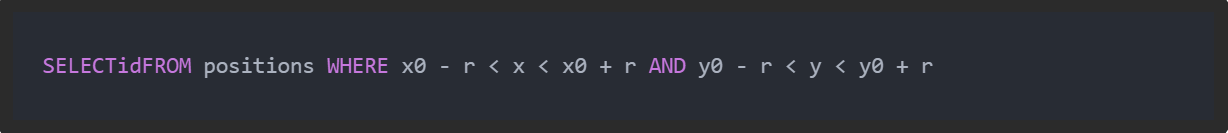
# **一、使用数据库实现查找附近的人**

我们都知道，地球上的任何一个位置都可以使用二维的 **经纬度** 来表示，经度范围 *[-180, 180]*，纬度范围 *[-90, 90]*，纬度正负以赤道为界，北正南负，经度正负以本初子午线 *(英国格林尼治天文台)* 为界，东正西负。比如说，北京人民英雄纪念碑的经纬度坐标就是 *(39.904610, 116.397724)*，都是正数，因为中国位于东北半球。

所以，当我们使用数据库存储了所有人的 **经纬度** 信息之后，我们就可以基于当前的坐标节点，来划分出一个矩形的范围，来得知附近的人，如下图：



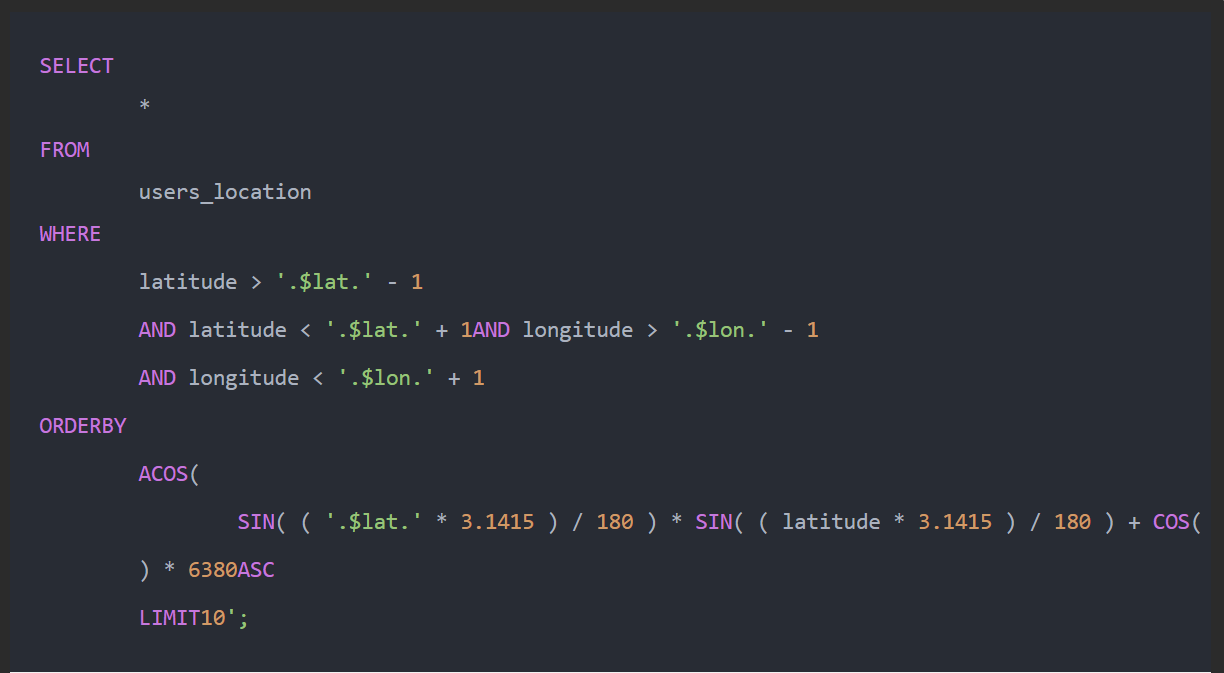
所以，我们很容易写出下列的伪 SQL 语句：



如果我们还想进一步地知道与每个坐标元素的距离并排序的话，就需要一定的计算。

当两个坐标元素的距离不是很远的时候，我们就可以简单利用 **勾股定理** 就能够得出他们之间的 **距离**。不过需要注意的是，地球不是一个标准的球体，**经纬度的密度** 是 **不一样** 的，所以我们使用勾股定理计算平方之后再求和时，需要按照一定的系数 **加权** 再进行求和。当然，如果不准求精确的话，加权也不必了。

参考下方 *参考资料 2* 我们能够差不多能写出如下优化之后的 SQL 语句来：*(仅供参考)*



为了满足高性能的矩形区域算法，数据表也需要把经纬度坐标加上 **双向复合索引 (x, y)**，这样可以满足最大优化查询性能。

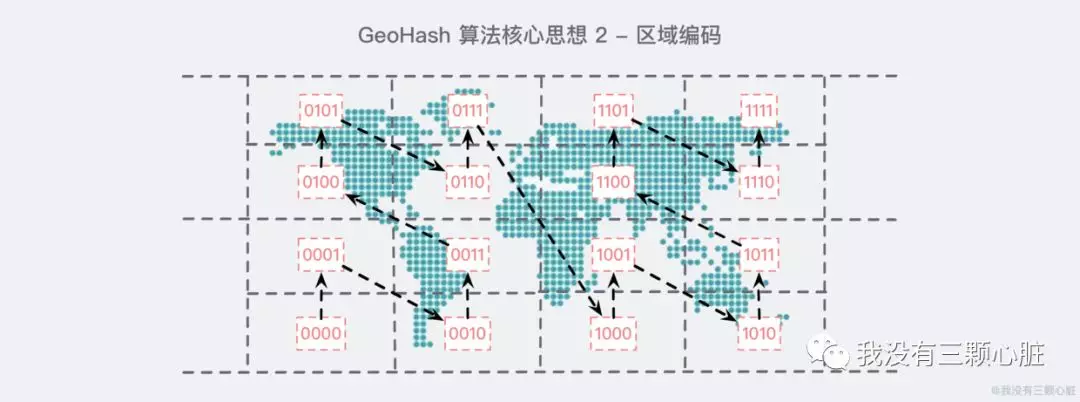
# **二、GeoHash 算法简述**

这是业界比较通用的，用于 **地理位置距离排序** 的一个算法，**Redis** 也采用了这样的算法。GeoHash 算法将 **二维的经纬度** 数据映射到 **一维** 的整数，这样所有的元素都将在挂载到一条线上，距离靠近的二维坐标映射到一维后的点之间距离也会很接近。当我们想要计算 **「附近的人时」**，首先将目标位置映射到这条线上，然后在这个一维的线上获取附近的点就行了。

它的核心思想就是把整个地球看成是一个 **二维的平面**，然后把这个平面不断地等分成一个一个小的方格，**每一个** 坐标元素都位于其中的 **唯一一个方格** 中，等分之后的 **方格越小**，那么坐标也就 **越精确**，类似下图：



经过划分的地球，我们需要对其进行编码：



经过这样顺序的编码之后，如果你仔细观察一会儿，你就会发现一些规律：

* 横着的所有编码中，**第 2 位和第 4 位都是一样的**，例如第一排第一个 0101 和第二个 0111，他们的第 2 位和第 4 位都是 1；
* 竖着的所有编码中，**第 1 位和第 3 位是递增的**，例如第一排第一个 0101，如果单独把第 1 位和第 3 位拎出来的话，那就是 00，同理看第一排第二个 0111，同样的方法第 1 位和第 3 位拎出来是 01，刚好是 00 递增一个；

通过这样的规律我们就把每一个小方块儿进行了一定顺序的编码，这样做的 **好处** 是显而易见的：每一个元素坐标既能够被 **唯一标识** 在这张被编码的地图上，也不至于 **暴露特别的具体的位置**，因为区域是共享的，我可以告诉你我就在公园附近，但是在具体的哪个地方你就无从得知了。

总之，我们通过上面的思想，能够把任意坐标变成一串二进制的编码了，类似于 11010010110001000100 这样 *(注意经度和维度是交替出现的哦..)*，通过这个整数我们就可以还原出元素的坐标，整数越长，还原出来的坐标值的损失程序就越小。对于 **"附近的人"** 这个功能来说，损失的一点经度可以忽略不计。

最后就是一个 Base32 *(0~9, a~z, 去掉 a/i/l/o 四个字母)* 的编码操作，让它变成一个字符串，例如上面那一串儿就变成了 wx4g0ec1。

在 **Redis** 中，经纬度使用 52 位的整数进行编码，放进了 zset 里面，zset 的 value 是元素的 key，score 是 **GeoHash** 的 52 位整数值。zset 的 score 虽然是浮点数，但是对于 52 位的整数值来说，它可以无损存储。

# **三、在 Redis 中使用 Geo**

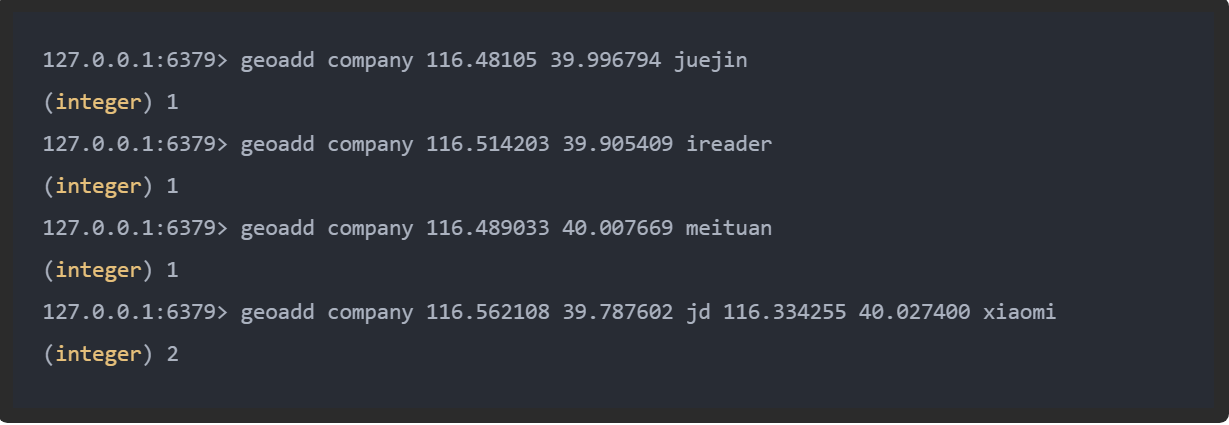
下方内容引自 *参考资料 1 - 《Redis 深度历险》*

在使用 **Redis** 进行 **Geo 查询** 时，我们要时刻想到它的内部结构实际上只是一个 **zset(skiplist)**。通过 zset 的 score 排序就可以得到坐标附近的其他元素 *(实际情况要复杂一些，不过这样理解足够了)*，通过将 score 还原成坐标值就可以得到元素的原始坐标了。

Redis 提供的 Geo 指令只有 6 个，很容易就可以掌握。

## **增加**

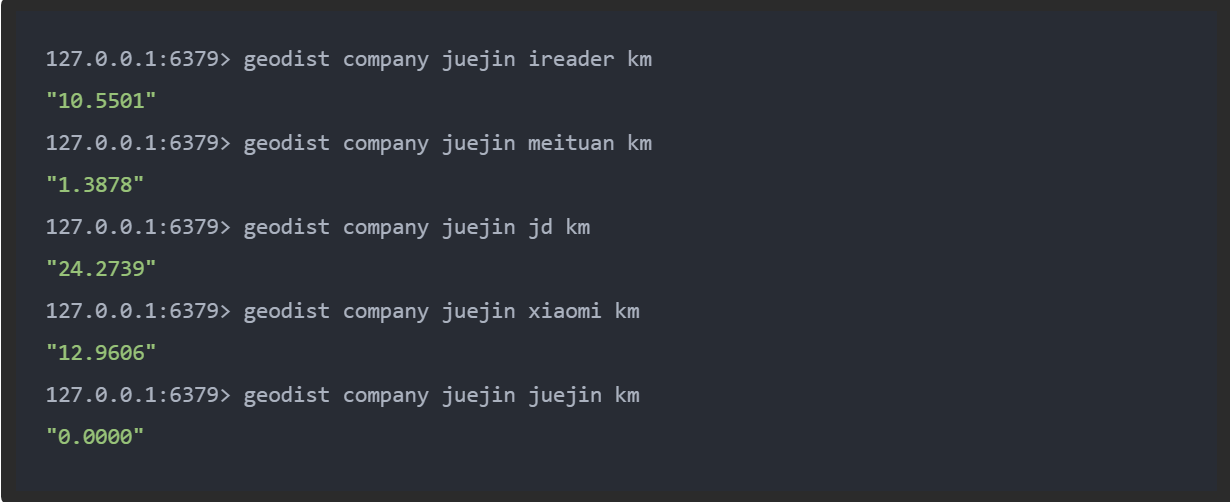
geoadd 指令携带集合名称以及多个经纬度名称三元组，注意这里可以加入多个三元组。



过很奇怪.. Redis 没有直接提供 Geo 的删除指令，但是我们可以通过 zset 相关的指令来操作 Geo 数据，所以元素删除可以使用 zrem 指令即可。

## **距离**

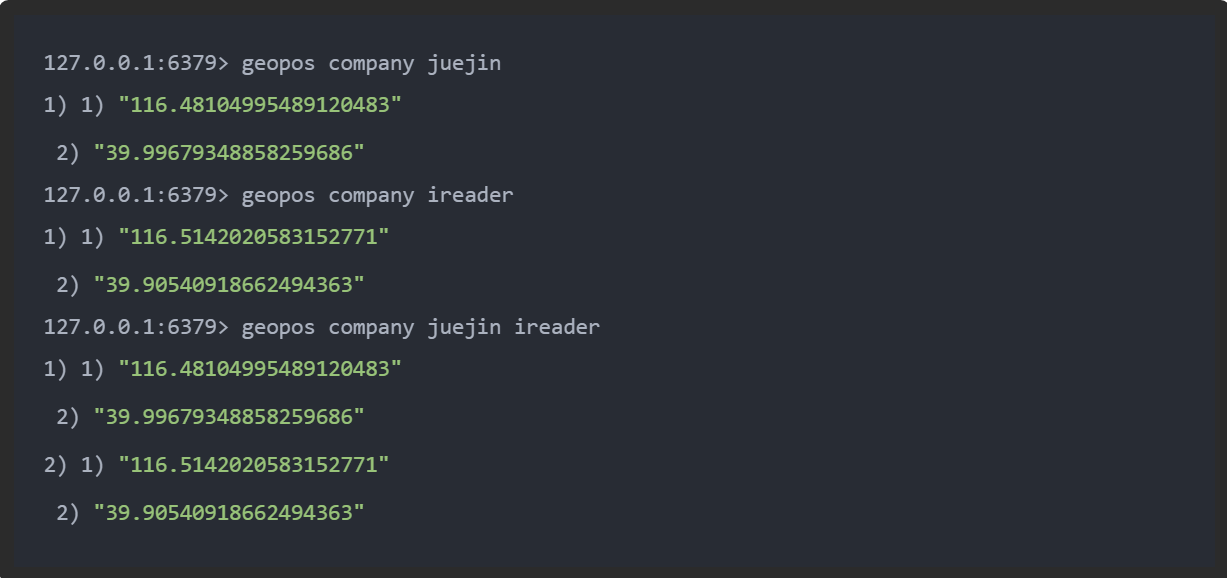
geodist 指令可以用来计算两个元素之间的距离，携带集合名称、2 个名称和距离单位。



我们可以看到掘金离美团最近，因为它们都在望京。距离单位可以是 m、km、ml、ft，分别代表米、千米、英里和尺。

## **获取元素位置**

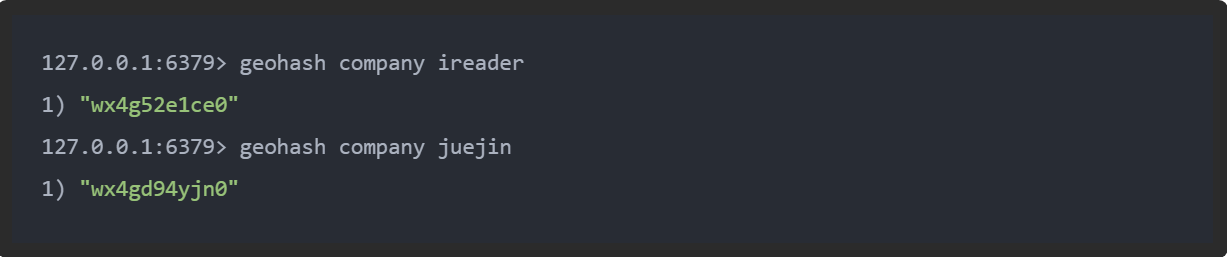
geopos 指令可以获取集合中任意元素的经纬度坐标，可以一次获取多个。



我们观察到获取的经纬度坐标和 geoadd 进去的坐标有轻微的误差，原因是 **Geohash** 对二维坐标进行的一维映射是有损的，通过映射再还原回来的值会出现较小的差别。对于 **「附近的人」** 这种功能来说，这点误差根本不是事。

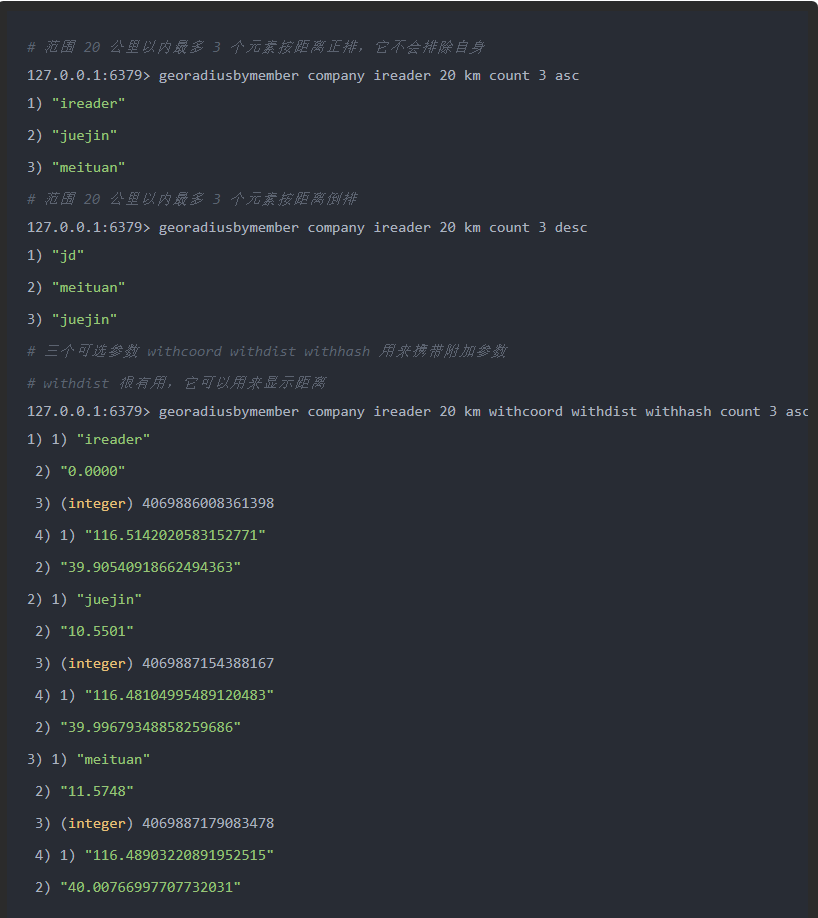
## **获取元素的 hash 值**

geohash 可以获取元素的经纬度编码字符串，上面已经提到，它是 base32 编码。你可以使用这个编码值去 http://geohash.org/${hash} 中进行直接定位，它是 **Geohash** 的标准编码值。

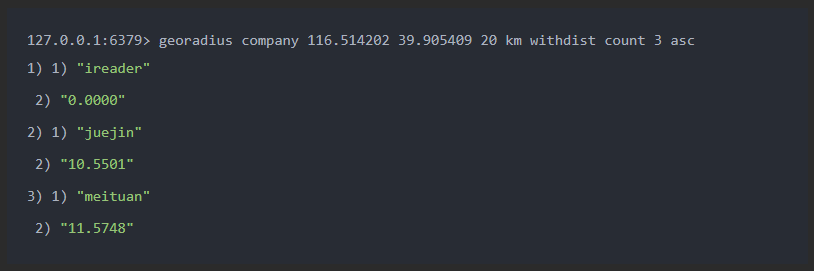


## **附近的公司**

georadiusbymember 指令是最为关键的指令，它可以用来查询指定元素附近的其它元素，它的参数非常复杂。



除了 georadiusbymember 指令根据元素查询附近的元素，**Redis** 还提供了根据坐标值来查询附近的元素，这个指令更加有用，它可以根据用户的定位来计算「附近的车」，「附近的餐馆」等。它的参数和 georadiusbymember 基本一致，除了将目标元素改成经纬度坐标值：



## **注意事项**

在一个地图应用中，车的数据、餐馆的数据、人的数据可能会有百万千万条，如果使用 **Redis** 的 **Geo** 数据结构，它们将 **全部放在一个** zset 集合中。在 **Redis** 的集群环境中，集合可能会从一个节点迁移到另一个节点，如果单个 key 的数据过大，会对集群的迁移工作造成较大的影响，在集群环境中单个 key 对应的数据量不宜超过 1M，否则会导致集群迁移出现卡顿现象，影响线上服务的正常运行。

所以，这里建议 **Geo** 的数据使用 **单独的 Redis 实例部署**，不使用集群环境。

如果数据量过亿甚至更大，就需要对 **Geo** 数据进行拆分，按国家拆分、按省拆分，按市拆分，在人口特大城市甚至可以按区拆分。这样就可以显著降低单个 zset 集合的大小。