使用Go与redis构建有趣的应用

黄健宏

本文分为4个部分，第一部分是介绍redis的功能、应用以及数据结构是怎样的。第二部分是开始使用redis构建锁。第三个是使用redis构建在线用户统计器。第四个是使用redis构建自动补完程序。
首先介绍一下redis的特点，redis具有多种不同的数据结构可用，包括字符串、散列、列表、有序集合、位图（bitmap）、Hyperloglog、地理坐标（GEO）等。它还有内存储存和基于多路服用的事件响应系统，确保了命令请求的执行速度。第三个是它具有丰富的附加功能，如事务、lua脚本，键过期机制（定期让键自动删除）、键淘汰机制，多种持久化方式（AOF、RDB、RDB + AOF混合）等。另外它还有强大的多机功能支持，前几年redis还是2点多版本的时候，人们经常会说是它是玩具数据库，因为缺少多机的支持，最近这几个版本已经慢慢加入了多机功能，比如说把主从提供高可用。最新的4.0版本它还提供了一个模块的扩展系统，把redis当做一个内存的平台，它提供了一些接口，用户可以通过接口去使用redis的功能，在原来的基础上提供更多新的功能，让你对它进行开发。

图 1

redis有多种不同的数据结构。首先是字符串，有点类似C语言的字符串，但是在C语言的基础上增加了一些功能，譬如说长度记录，就是获取一个C字符串的长度是线性的，redis增加了一个字符串长度记录功能，让客户可以以常数复杂度去获取长度，不需要整个遍历一遍，直接访问就可以。还有一些二进制安全，可以直接在数据库里面储存二进制数据，不一定是字符串，比如说一些压缩文件都可以。然后它还有内存预分配系统，当你要频繁的修复字符串，它可以通过内存预分配策略减少内存重分配次数，提高性能。

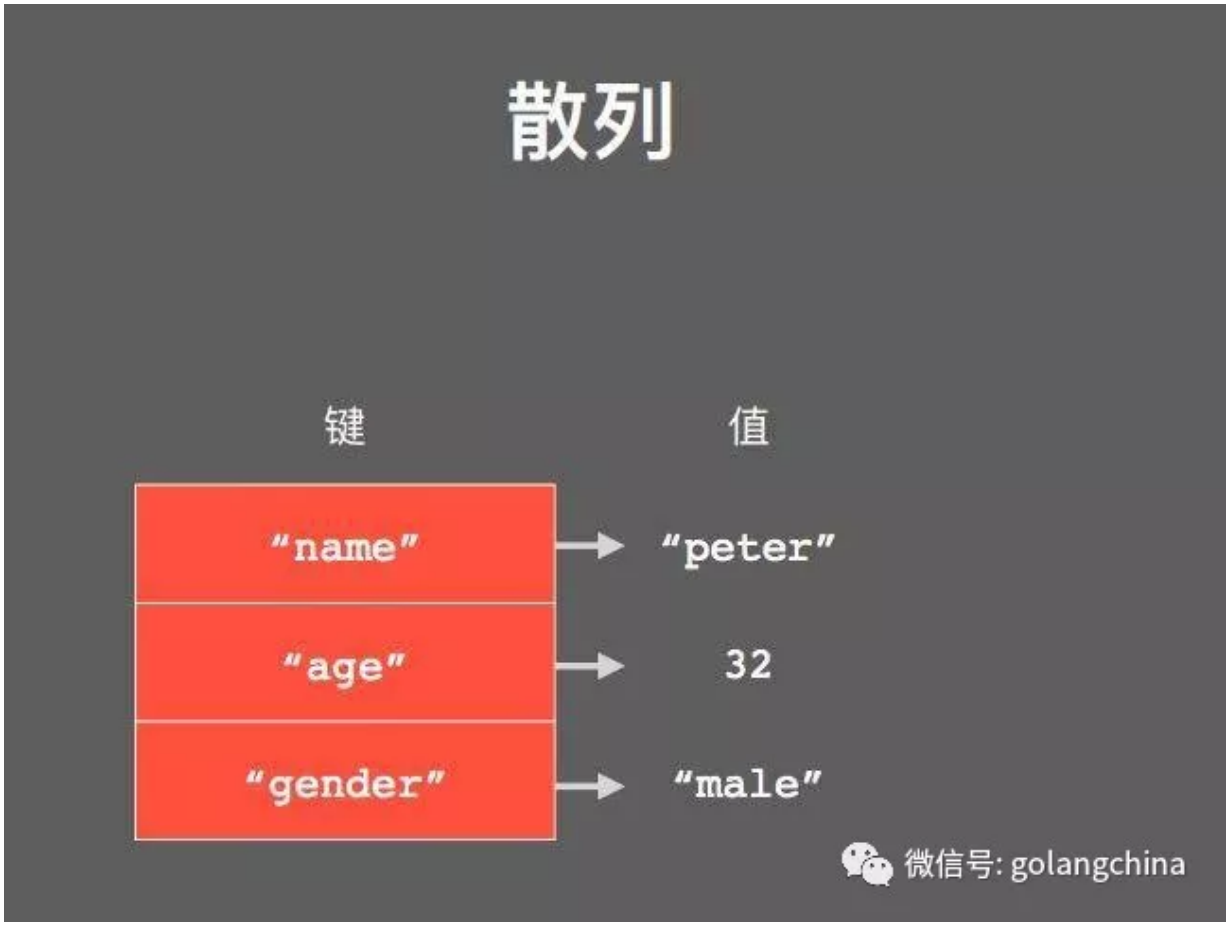

图 2

第二个散列，散列和GO语言里面的map很像，一个键映射到另外一个值，每个键都各不同，获取单个键的复杂度为常数，如果有需要可以一次把键值对全部获取出来，但是性能会比较差。

图 3

还有列表，列表有点像GO的序列，底层是用双端链表来实现的的，所以你可以对它的头或者尾进行操作，都是常数复杂度，但是如果你要增加或者截断这个列表或者是对它进行遍历的话就比较慢。它跟之前的哈希表不同，它允许有重复的元素。

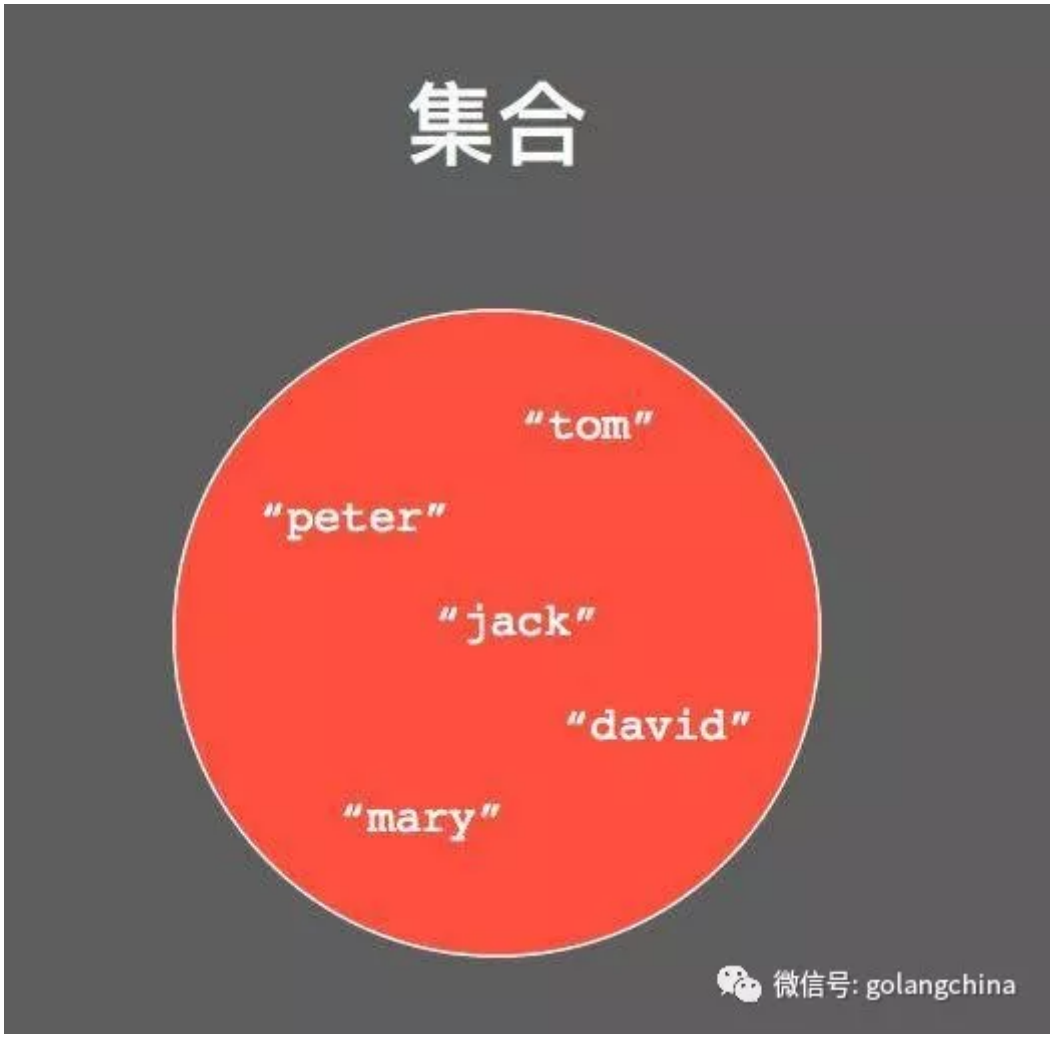

图 4

接下来是集合，集合会以无序的方式储存多个各不相同的元素，针对单个元素的复杂度的操作都为常数，速度很快，可以把它跟其他集合，做一些交集、并集计算。

图 5

有序集合比较少见，集合里每个元素都由一个分值和成员组成，每个成员是按照分值的大小有序的排序。用户可以按照这个分值有序的获取，譬如说你获取分值最少的元素或者获取分值最大的两个元素，或者你直接按照成员获取前两个和后两个都可以。有序集合的底层是使用跳跃表来实现的，所以获取单个元素会有复杂度。

图 6

位图是由一连串二进制位组成的数组，数组有索引，你可以根据索引对二进制位进行操作。可单独设置指定的位，可获取指定范围的多个位又或者对它们计数。

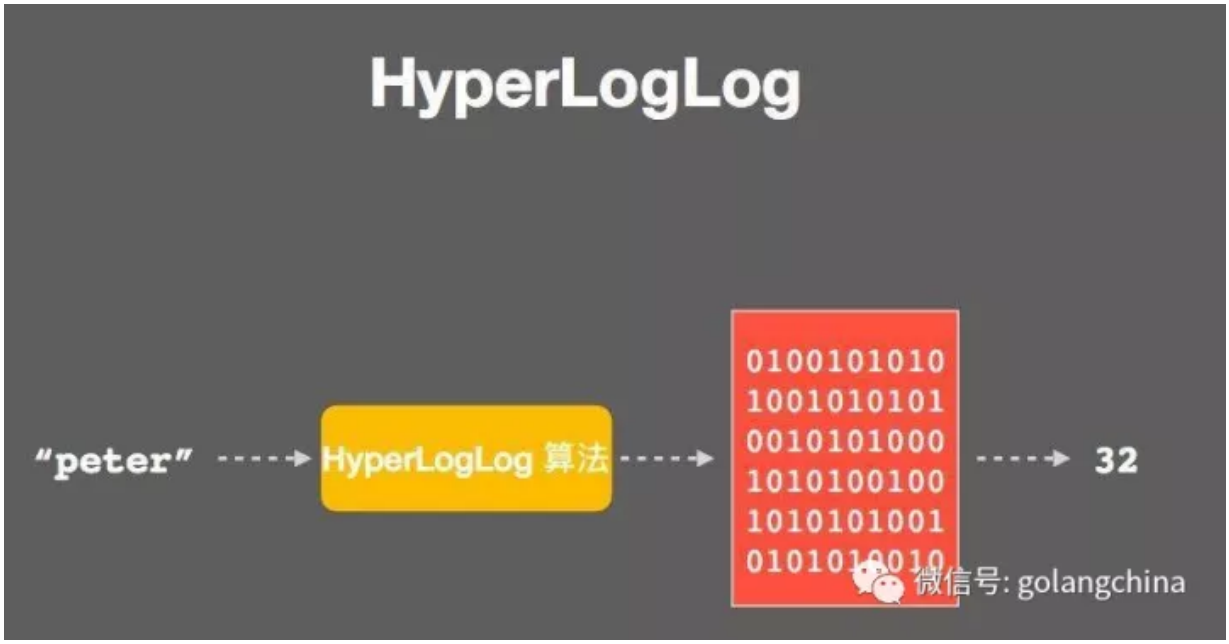

图 7

Hyperloglog比较复杂，它有一个算法，对一系列二进制位进行计算，可以算出计数值，一个特点是就算你往Hyperloglog里面添加上亿个元素，它对上亿个元素进行计数，占用的内存都是固定的12GB，后面我们可以看到如何使用Hyperloglog减少对内存的使用。

图 8

最后一个地理位置，你给一个经纬度和位置名就可以储存到redis里面，或者对它进行范围计算，或者说计算出在这个地方100公里或者5公里之内有什么东西，写LBS应用的朋友应该对这个会感兴趣。

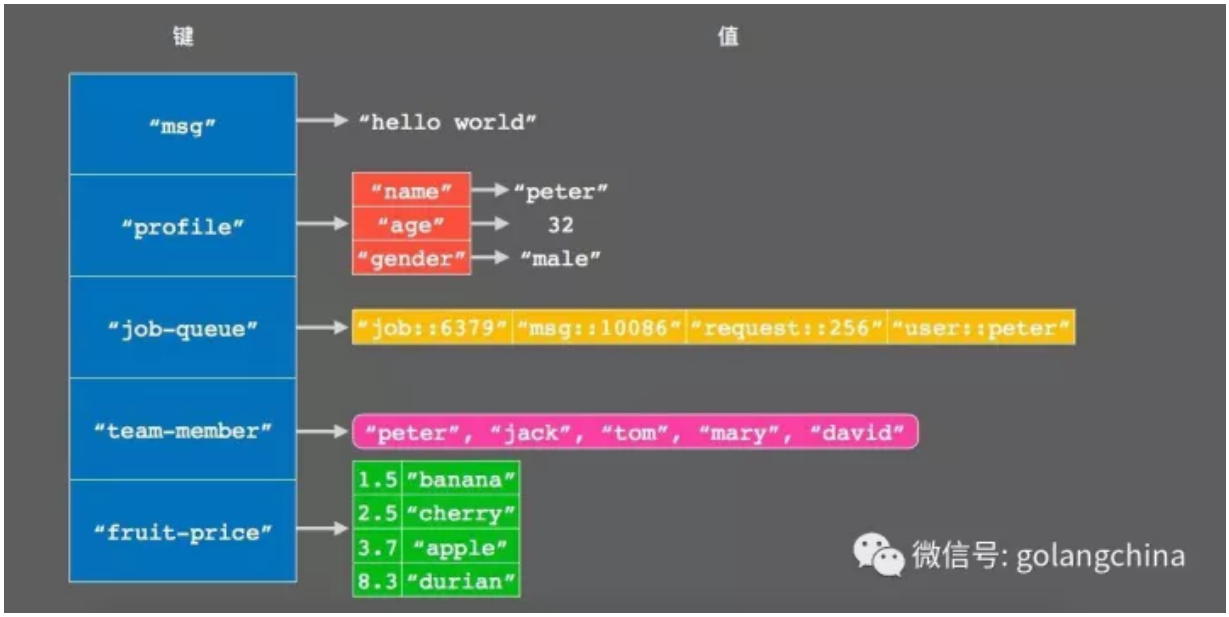

图 9

这个就是redis数据库典型的样子，redis是键值对数据库，它的键都是字符串，它的值可以是我们刚才介绍过的数据结构的任意一种。

图 10

我们可以通过命令去操作这些键值对，就好像我们用SQL语言操作关系数据库一样。首先要有个命令，命令你要执行什么工作，第二个就有一个键，键就是要有一个你要操作的对象，然后你要有任意多个参数，因为非常简单，所以有时候经常都是1、2个参数就可以做到很多东西，有的比较复杂的命令还会有选项，选项还会有值。

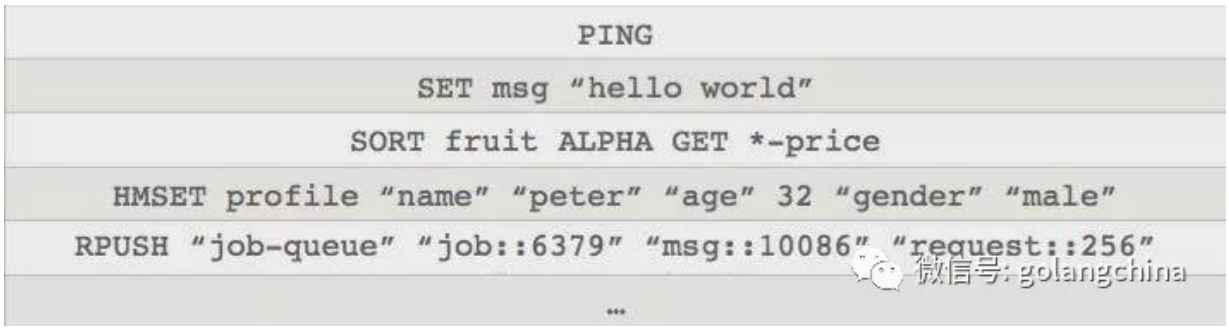

图 11

这里是一些命令的示例，比如说ping,如果一切正常服务器就会给你返回一个ping，就是以一个键的形式进行排序,第三个是对一些哈希散列进行设置，设置多个键值对，最后一个RPUSH可以把后面的元素都归到一个列表里面。

回到我们今天的主题GO，要在GO使用redis，我们需要通过客户端，客户端官方推荐两个radix和redigo，今天我们会使用的是radix和我们的老朋友GOget。这里是连接客户端的示例代码，用dial方法去连接数据库，通过cmd方法去执行ping命令并获取回复，最后我们用str方法过将回复转换成字符串，然后打印出来。

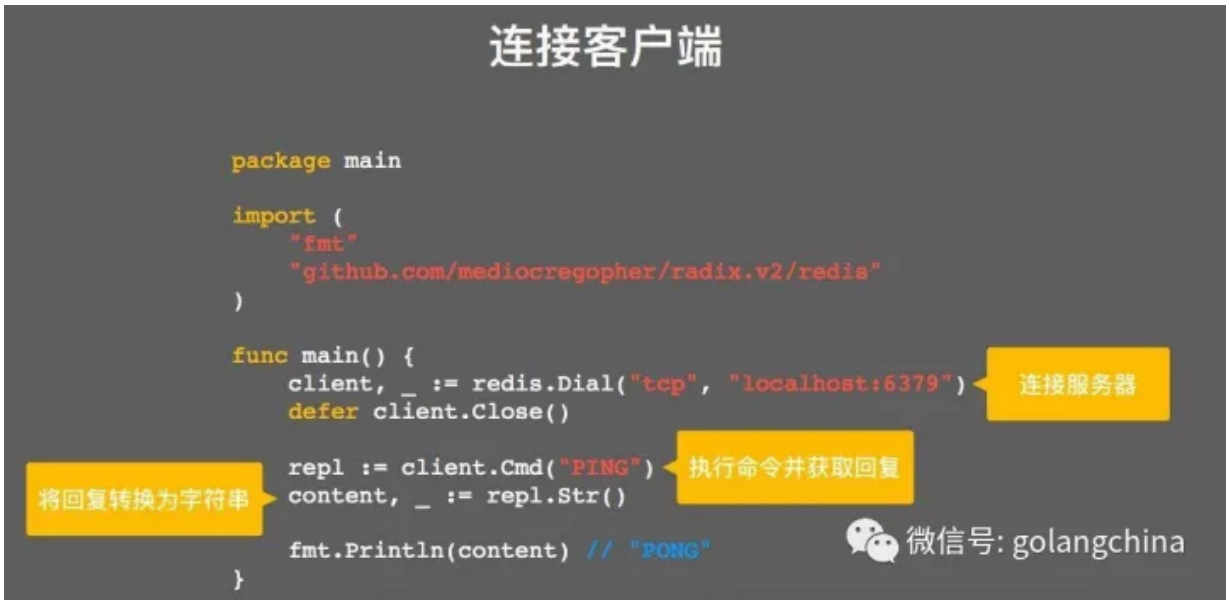

图 12

来到我们第一个应用实例——锁。锁是一种同步机制，它可以保证一项资源在任何时候只能被一个进程使用，如果有其它进程想使用这个资源就必须等待，直到正在使用资源的进程放弃使用权为止。一般一个锁都有获取和释放这2个操作，获取操作就是获取资源的使用权，在任何时候，获取这个资源的安全，只能有一个进程去获得锁，当一个锁被获取的时候其他尝试获取锁的其它进程都会失败。还有是一个释放操作，获得的锁进程释放后，其他人就可以再次使用资源。

方法一是使用字符串结构去实现锁，具体的方法是把一个字符串用做锁，如果这个字符串键有值，那么就说明锁被获取了，如果键没有值的话就是没有被获取。

下面是需要用到的一些命令：

- GET key：获取字符键 key 的值,如果该键尚未有值,那么返回 个空值(nil)
- SET key value： 将字符 键 key 的值设置为 value ,如果键已经有值,那么默认使 新值去覆盖旧值
- DEL key：删除给定的键 key

我们首先使用GET方法，获取键的值，并把这个值转换为字符串，然后用if方法去检查有没有值，如果没有值的话就返回一个空的字符串，确认没有值就调用set方法进行设置，就是给它加锁。这里展示的代码就是为了节约时间，我们就把错误的回复处理掉了。

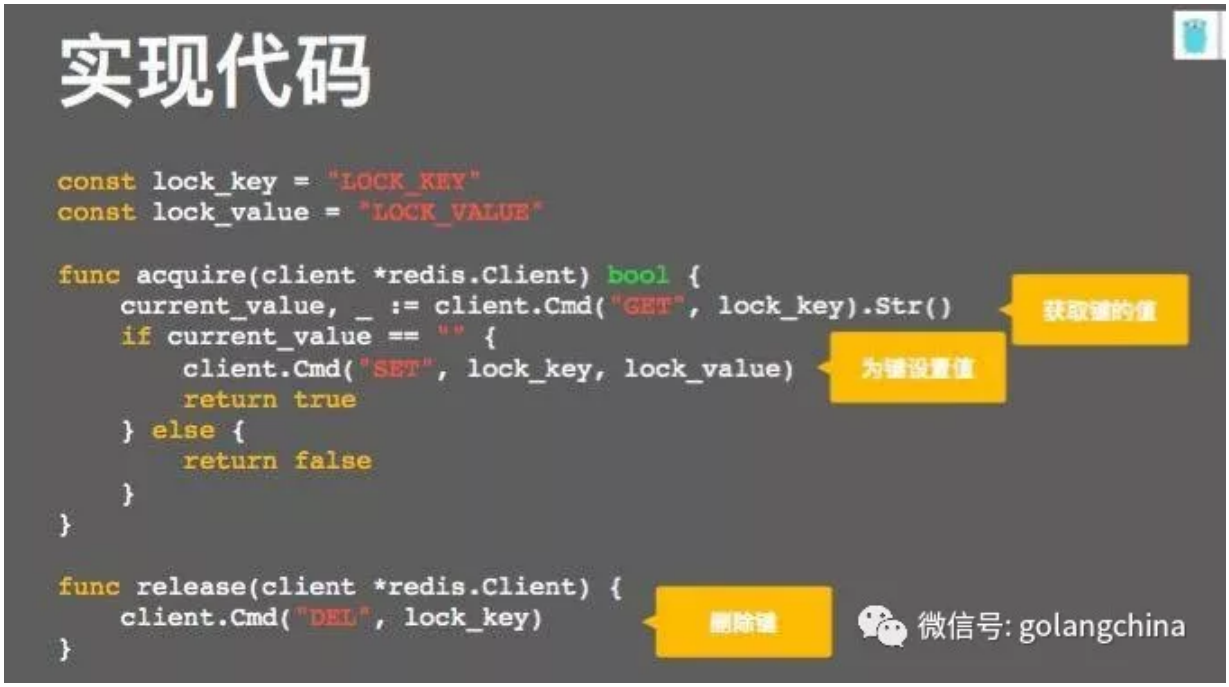

图 13

这个方法虽然能够成功，但是它有一个竞争条件，在执行get命令之后和执行set方法之前，这里有个中间时段，其他客户端就可能抢先对键进行了设置，这时候就会产生一个竞争条件，假设现在有两个客户端，他们一起去执行刚才的acquire方法，他们同时调用get，都获得了那个键没有值的共识，客户端2因为执行的快一点就执行了set方法进行了加锁，然后成功获得了锁，这个时候客户端1姗姗来迟，因为前面是同时的执行get方法，大家都以为键没有加锁，所以客户端1就会继续执行这个set方法，对锁进行加锁，这时候就会出现两个客户端同时出现成功获取锁的情况,就会出错。

为了解决这个问题我们需要使用redis的事务特性去保证加锁操作的安全性。这里是一个redis的非事务命令的演示，一般来说很多数据库都一样的，如果设置一般命令的话会一个接一个，事务也是命令的一种特殊的命令，一个事务里面会包含多种命令，这样就可以保证安全性。

现在看一下安全的锁需要用到什么命令？第一个是WATCH，监视给定的键，如果被监视的键在事务执行之前已经被其它客户端抢先修改的话，执行命令的客户端提交的命令就会被拒绝。第二个是MULTI命令，它会开启一个事务，在执行这个命令之后，客户端发送的所有操作命令都会被进入到事务队列里等待。最后一个EXEC，是尝试执行事务，成功时将返回一个由多个命令回复组成的队列给客户端，失败则返回nil。

图 14

这个是实现代码，首先用WATCH命令去监测一个键，然后再调用get方法，判断这个键没有值的话就调用MULTI开始一个事务，然后把EXEC放到队列里面，这是最关键的一步，如果事务成功执行的话，这个回复就不会是空的，如果返回空，就说明lock_key就已经被其它客户端抢先修改了，然后我们就根据这个事物是否执行成功来判断加锁是否成功。

但是使用事务也会带来代价，它会使代码复杂化，我们的加锁程序本身也不到5行代码，现在加到10行，代码的量加到了一倍，虽然保证了安全性，但是代码复杂了。考虑到这种情况，redis提供了一种新的功能——带NX选项的SET命令。当我们使用带NX选项的SET命令时，只有在键key不存在的情况下才会对它进行设置，如果键已经有值，就会放弃对它进行设置代码，并返回nil表示设置失败。NX选项的作用就相当于把刚才这一段会引起竞争条件的代码放到服务器里面执行了，这样就保证了执行操作的安全性。

图 15

这个就是我们的NX选项，很简单，比我们最初的那一版加锁还要简单，就直接一个SET命令，但是有一个NX选项，在lock_key没有值的情况下对它进行设置，通过这个检查回复是否回空的情况下就知道加锁成功了。

图 16

看一下第二个应用示例——在线用户统计器，比如可以统计网站有多少用户，有多少人看直播等。实现这种功能的第一个方法就是使用集合，当一个用户上线的时候我们就把用户名添加在在线用户集合里面。

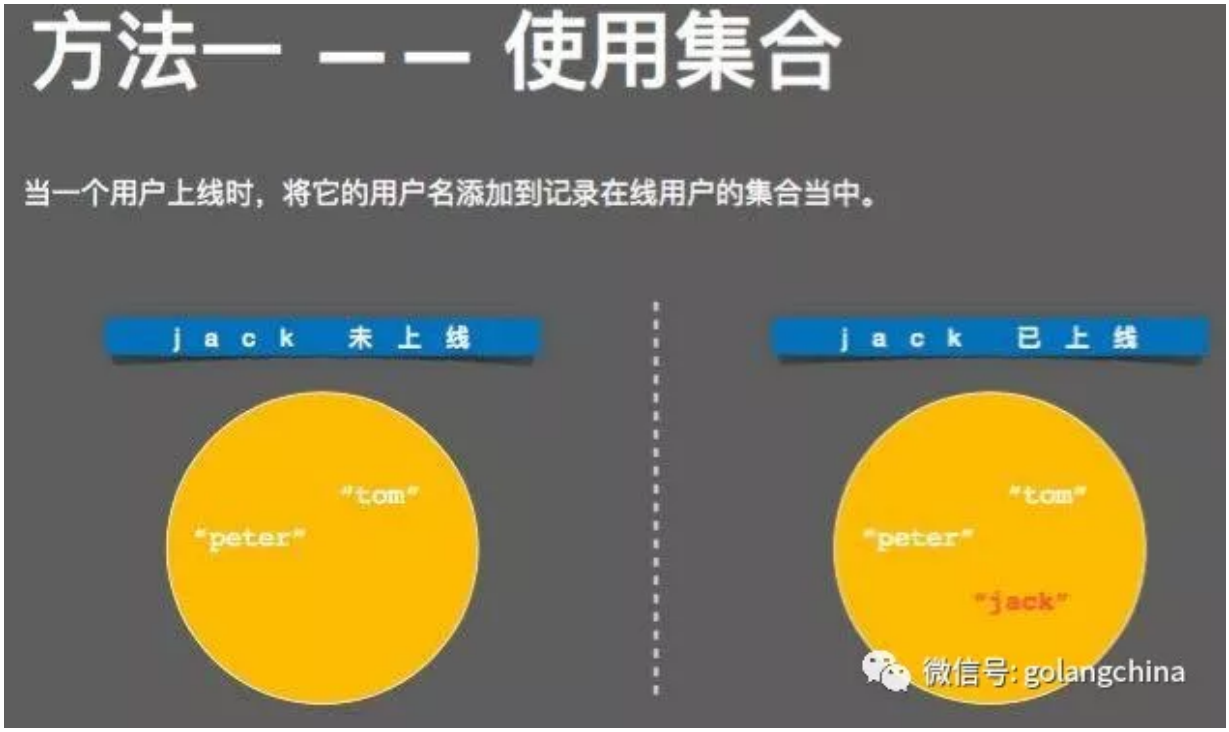

图 17

需要用到的命令，SADD set element [element ...]，可以将给定的元素添加到集合当中。SCARD set 可以获取集合的基数 ，即集合包含的元素数量，也即当前有多少用户。 SISMEMBER set element 可以检查给定的元素是否存在集合里面，应用起来比如可以用这个命令去检查某个用户是否在线等。

图 18

实现代码也很简单，当一个用户在线的时候我们就接受用户名，然后调入SADD命令，如果要统计多少用户在线的话，就获取集合有多少个元素的基数，最后我们要检查一个用户是否在线的话，会给定元素存在的时候返回1，我们就看返回值数给1就可能知道用户是否在线。

使用集合统计在线用户有一个很严重的问题，集合的体积将随着元素的增加而增加，假设每个用户平均名字是10字节，那么拥有100万用户的网站每天需要使用10MB内存去储存，拥有一千万用户的网站每天需要使用100MB 内存去储存。如果把这些信息储存1年，拥有100万用户的网站每年需要为此使用3.65GB内存，拥有1000万用户的网站每年需要为此付出36.5G内存。统计如此小的一个功能却要花费如何的的内存将会非常不值得，这还不包括一些额外的开销。

方法二是使用位图，为每一个用户创建一个ID，当一个用户上线后，就用他的ID作为索引，假设现在有一个用户peter，我们给他映射一个ID 10086，然后根据这个ID把这个位图里面索引为10086的值设置为1，值为1的用户就是在线，值为0的就是不在线。

这里同样需要用到3个命令：

- SETBIT bitmap index value ：将位图指定索引上的二进制位设置为给定的值
- GETBIT bitmap index ：获取位图指定索引上的二进制位
- BITCOUNT bitmap ：统计位图中值为 1 的二进制位的数量

实现代码如下， setbit接受用户的ID，把二进制位设为1。统计值为1的二进制位数量，如果要检查用户是否在线，就先根据用户的ID然后检查位图里面的二进制位的值是否为1，1就是在线，0就是不在线。

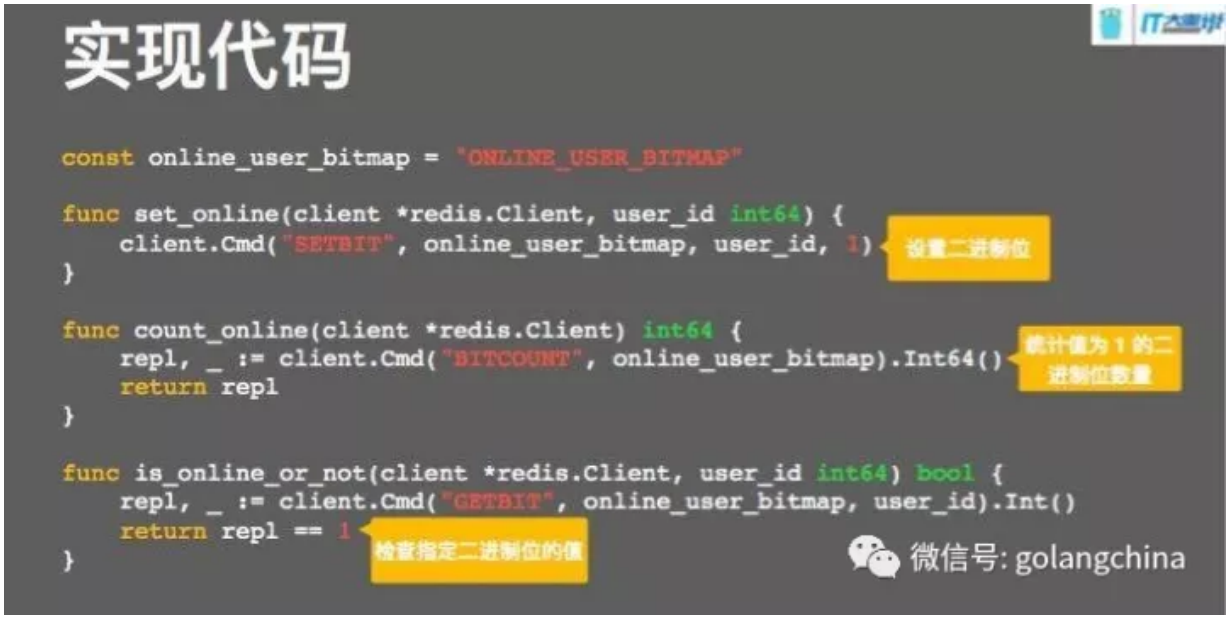

图 19

跟刚才的集合相比，虽然位图的体积仍然会随着用户数量的增多而变大，但因为记录每个用户所需的内存数量从原来的平均10字节变成了1位，所以将节约大量的内存，把几十G的占用降为了几百MB。

我们要继续进行优化就得到了方法三——使用Hyperloglog。当一个用户上线时，我们就使用Hyperloglog对他进行计数。假设现在有一个用户jack，我们通过Hyperloglog算法对他进行计数，然后把这个计数反映到Hyperloglog里面，如果这个元素之前没有被Hyperloglog计数过的话，你新添加在Hyperloglog里面就会对自己的计数进行加1。如果jack已经存在，它的计数值就不会加1。

Hyperloglog需要用到两个方法，第一个是PFADD hll element [element ...] ，对给定的元素进行计数。第二个PFCOUNT hll 是获取Hyperloglog的近似基数，也即是基数的估算值。使用Hyperloglog有一个缺陷，因为Hyperloglog是一个概率算法，它只能给出一个估算的值。比如你有1000个用户进行计数，可能只会返回你970或者是980个，它没有办法给出一个准确的计数值，只能给出一个近似，好处就是无论我们对多少用户进行计数，单个Hyperloglog都只占12KB，下面是实现代码。

图 20

三种实现的内容消耗对比如下图所示：

图 21

最后一个示例应用——自动补全。我们在很多无论是桌面应用，譬如说网页浏览器、搜索引擎、twitter，当你输入一些东西的时候，譬如说你在浏览器里面输入go然后会自动帮你补全go相关的一些命令。我们分析一下自动补全的原理，当我们输入的时候会返回一个补全的补全的结果，这个结果还带有一个权重值，排在越前面的值权重值越高。为了实现这个自动补全程序我们需要构建一个权重表，对redis来说储存这样的权重最合适的就是有序集合。

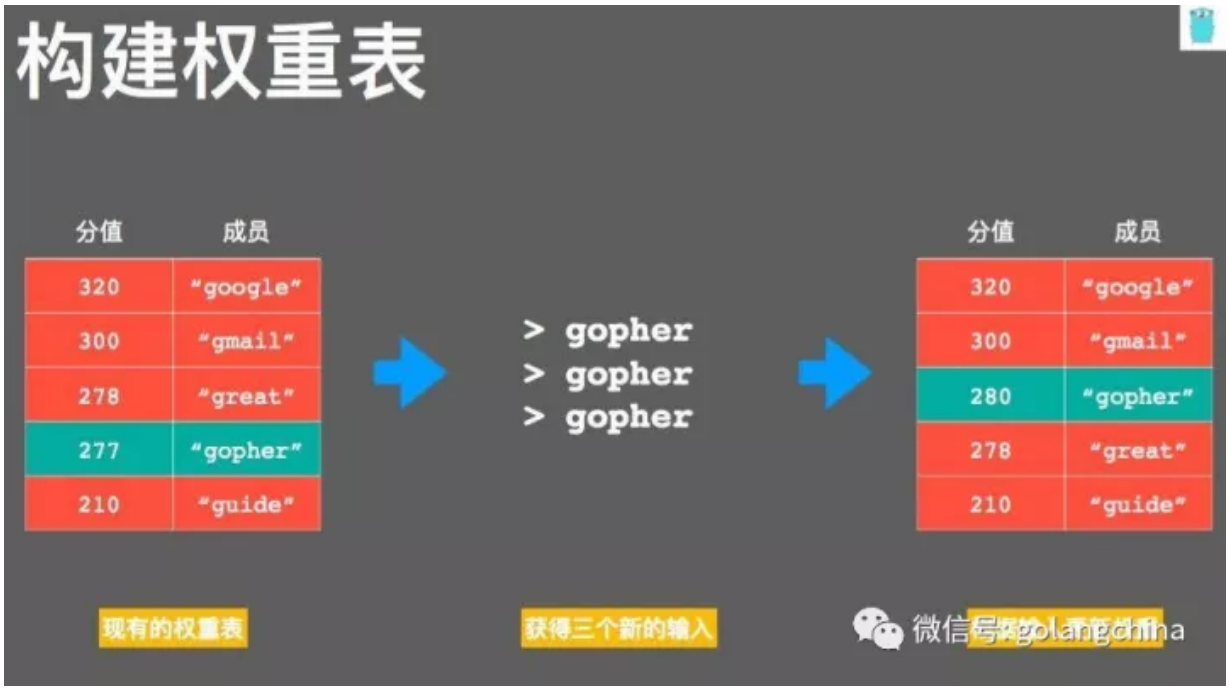

图 22

那么该如何去构建一个权重表呢？构建权重表有很多种方式，比如你可以用很多复杂的算法计算出每个候选结构的权重，也有一些简单的方法，比如最简单的是你根据用户的输入，譬如用户输入gmail，然后你就对用户进行计数，如果输入google又进行一次计数，然后根据客户输入的计数构建一个权重表。假设我们有一个权重表，

gopher的权重值为277，如果连续输入3次gopher，就会对gopher的权重值进行加三操作，提高到280。所以我们构建了一个权重表，但是我们无法对它进行补全。如果我们要补全的话，你输入G的时候会联想GOPHER，输入go的时候也会联想gopher, 我们要枚举出这样的一个排列，对每个排列都建一个权重表，譬如我们要在权重表里面添加一个gopher项，对于每一个排列都要为gopher添加一个权重项。每当用户输入一个的gopher时候，对每个排列对应的权重表我们在权重表里面找到gopher然后再添加过去。

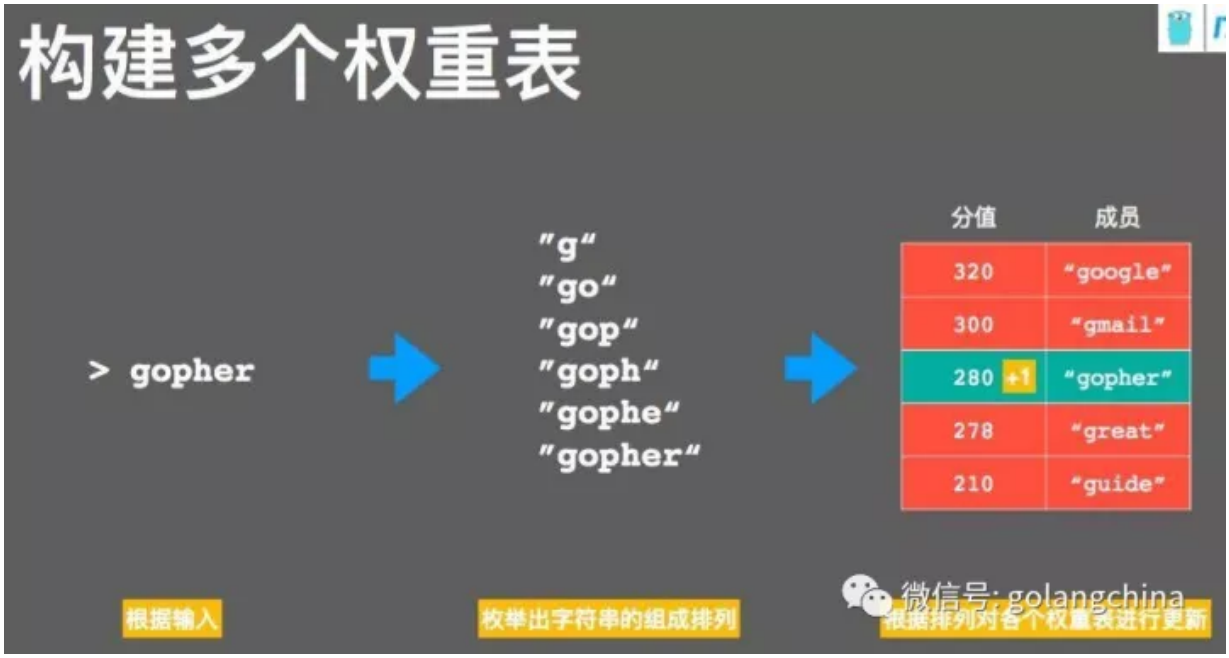

图 23

补全的过程，当我们输入G的时候就根据G查找对应的权重表，可以看到gopher排到第三位，它的分值比google和gmail要低，这个表里面全部是G开头的候选结果。当我们输入go的时候，就开始联想，它会继续寻找一个go开头的成员。可以看到根据我们联想的结果，gopher不断的在上升，就出现在用户上面，当我们输入GOP的时候，会根据GOP找到我们的gopher。

图 24

实现我们的自动补全需要用到两个命令，第一个ZINCRBY zset increment member是对给定成员的分值执行自增操作；第二个ZREVRANGE zset start end [WITHSCORES]是按照分值从大到小的顺序，从有序集合里面获取指定索引范围内的成员。因为我们的权重是从大到小排列的，我们就先获取权重最高的值就会显示出来,实现代码如下所示。

图 25

我们来总结一下：首先GO和redis都是很简单强大的工具，组合起来可以轻松解决很多过去 常难以实现或者需要很多代码才能实现的特性，比如自动补全，如果你不是用GO和redis一起做，那今天的PPT数量页数可能会增加三倍。第二点是在构建程序的时候一定要确保程序的安全性和正确性，虽然保证这两点常常会使得程序变得复杂，但有时候本身也有鱼与熊掌兼得的方案。譬如说前面说的在线统计，就可以有很多方法去实现，但是只有对它足够熟悉，才能找到最优方法。最后一点是不同方法实现的效率和功能通常也会有所不同，我们需要根据自身的情况进行选择，不要盲目相信所谓的最优解。