Redis高级数据结构HyperLogLog

HyperLogLog(Hyper['haɪpə(r)])并不是一种新的数据结构(实际类型为字符串类型)，而是一种基数算法，通过HyperLogLog可以利用极小的内存空间完成独立总数的统计，数据集可以是IP、Email、ID等。

如果你负责开发维护一个大型的网站，有一天产品经理要网站每个网页每天的 UV 数据，然后让你来开发这个统计模块，你会如何实现？

如果统计 PV 那非常好办，给每个网页一个独立的 Redis 计数器就可以了，这个计数器的 key 后缀加上当天的日期。这样来一个请求，incrby 一次，最终就可以统计出所有的 PV 数据。

但是 UV 不一样，它要去重，同一个用户一天之内的多次访问请求只能计数一次。这就要求每一个网页请求都需要带上用户的 ID，无论是登陆用户还是未登陆用户都需要一个唯一 ID 来标识。

一个简单的方案，那就是为每一个页面一个独立的 set 集合来存储所有当天访问过此页面的用户 ID。当一个请求过来时，我们使用 sadd 将用户 ID 塞进去就可以了。通过 scard 可以取出这个集合的大小，这个数字就是这个页面的 UV 数据。

但是，如果你的页面访问量非常大，比如一个爆款页面几千万的 UV，你需要一个很大的 set 集合来统计，这就非常浪费空间。如果这样的页面很多，那所需要的存储空间是惊人的。为这样一个去重功能就耗费这样多的存储空间，值得么？其实需要的数据又不需要太精确，1050w 和 1060w 这两个数字对于老板们来说并没有多大区别，So，有没有更好的解决方案呢？

这就是HyperLogLog 的用武之地，Redis 提供了 HyperLogLog 数据结构就是用来解决这种统计问题的。HyperLogLog 提供不精确的去重计数方案，虽然不精确但是也不是非常不精确，Redis官方给出标准误差是 0.81%，这样的精确度已经可以满足上面的 UV 统计需求了。

操作命令

HyperLogLog提供了3个命令: pfadd、pfcount、pfmerge。

例如08-15的访问用户是u1、u2、u3、u4，08-16的访问用户是u-4、u-5、u-6、u-7

pfadd

pfadd key element [element …]

pfadd用于向HyperLogLog 添加元素,如果添加成功返回1:

pfadd 08-15:u:id "u1" "u2" "u3" "u4"

pfcount

pfcount key [key …]

pfcount用于计算一个或多个HyperLogLog的独立总数，例如08-15:u:id的独立总数为4:

pfcount 08-15:u:id

如果此时向插入u1、u2、u3、u90，结果是5:

pfadd 08-15:u:id "u1" "u2" "u3" "u90"

pfcount 08-15:u:id

如果我们继续往里面插入数据，比如插入100万条用户记录。内存增加非常少，但是pfcount 的统计结果会出现误差。

以使用集合类型和 HperLogLog统计百万级用户访问次数的占用空间对比：

数据类型 1天 1个月 1年

集合类型 80M 2.4G 28G

HyperLogLog 15k 450k 5M

可以看到，HyperLogLog内存占用量小得惊人，但是用如此小空间来估算如此巨大的数据，必然不是100%的正确，其中一定存在误差率。前面说过，Redis官方给出的数字是0.81%的失误率。

pfmerge

pfmerge destkey sourcekey [sourcekey ...]

pfmerge可以求出多个HyperLogLog的并集并赋值给destkey，请自行测试。

原理概述

基本原理

HyperLogLog基于概率论中伯努利试验并结合了极大似然估算方法，并做了分桶优化。

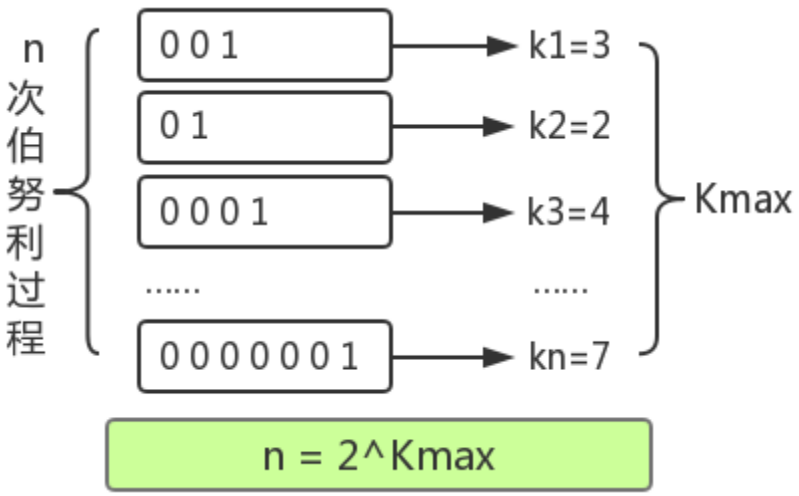
实际上目前还没有发现更好的在大数据场景中准确计算基数的高效算法，因此在不追求绝对准确的情况下，使用概率算法算是一个不错的解决方案。概率算法不直接存储数据集合本身，通过一定的概率统计方法预估值，这种方法可以大大节省内存，同时保证误差控制在一定范围内。目前用于基数计数的概率算法包括:

Linear Counting(LC)：早期的基数估计算法，LC在空间复杂度方面并不算优秀；

LogLog Counting(LLC)：LogLog Counting相比于LC更加节省内存，空间复杂度更低；

HyperLogLog Counting(HLL)：HyperLogLog Counting是基于LLC的优化和改进，在同样空间复杂度情况下，能够比LLC的基数估计误差更小。

举个例子来理解HyperLogLog 算法，有一天Fox老师和Mark老师玩抛硬币的游戏，规则是Mark老师负责抛硬币，每次抛的硬币可能正面，可能反面，每当抛到正面为一回合，Mark老师可以自己决定进行几个回合。最后需要告诉Fox老师最长的那个回合抛了多少次以后出现了正面，再由Fox老师来猜Mark老师一共进行了几个回合。



进行了n次，比如上图：

第一次：抛了3次才出现正面，此时 k=3，n=1

第二次试验：抛了2次才出现正面，此时 k=2，n=2

第三次试验：抛了4次才出现正面，此时 k=4，n=3

.....

第n 次试验：抛了7次才出现正面，此时我们估算，k=7，n=n

k是每回合抛到1（硬币的正面）所用的次数，我们已知的是最大的k值，也就是Mark老师告诉Fox老师的数，可以用k_max表示。由于每次抛硬币的结果只有0和1两种情况，因此，能够推测出k_max在任意回合出现

的概率 ，并由kmax结合极大似然估算的方法推测出n的次数n = 2^(k_max) 。概率学把这种问题叫做伯努利实验。

现在Mark老师已经完成了n个回合，并且告诉Fox老师最长的一次抛了4次，Fox老师此时也胸有成竹，马上说出他的答案16，最后的结果是：Mark老师只抛了3回合，

这三个回合中k_max=4，放到公式中，Fox老师算出n=2^4，于是推测Mark老师抛了16个回合，但是Fox老师输了，要负责买奶茶一个星期。

所以这种预估方法存在较大误差，为了改善误差情况，HLL中引入分桶平均的概念。

同样举抛硬币的例子，如果只有一组抛硬币实验，显然根据公式推导得到的实验次数的估计误差较大；如果100个组同时进行抛硬币实验，样本数变大，受运气影响的概率就很低了，每组分别进行多次抛硬币实验，并上报各自实验过程中抛到正面的抛掷次数的最大值，就能根据100组的平均值预估整体的实验次数了。

分桶平均的基本原理是将统计数据划分为m个桶，每个桶分别统计各自的k_max，并能得到各自的基数预估值，最终对这些基数预估值求平均得到整体的基数估计值。LLC中使用几何平均数预估整体的基数值，但是当统计数据量较小时误差较大；HLL在LLC基础上做了改进，采用调和平均数过滤掉不健康的统计值。

什么叫调和平均数呢？举个例子

求平均工资：A的是1000/月，B的30000/月。采用平均数的方式就是： (1000 + 30000) / 2 = 15500

采用调和平均数的方式就是： 2/(1/1000 + 1/30000) ≈ 1935.484

可见调和平均数比平均数的好处就是不容易受到大的数值的影响，比平均数的效果是要更好的。

结合实例理解原理

现在我们和前面的业务场景进行挂钩：统计网页每天的 UV 数据。

1.转为比特串

通过hash函数，将数据转为比特串，例如输入5，便转为：101，字符串也是一样。为什么要这样转化呢？

是因为要和抛硬币对应上，比特串中，0 代表了反面，1 代表了正面，如果一个数据最终被转化了10010000，那么从右往左，从低位往高位看，我们可以认为，首次出现 1 的时候，就是正面。

那么基于上面的估算结论，我们可以通过多次抛硬币实验的最大抛到正面的次数来预估总共进行了多少次实验，同样也就可以根据存入数据中，转化后的出现了 1 的最大的位置 k_max 来估算存入了多少数据。

2.分桶

分桶就是分多少轮。抽象到计算机存储中去，存储的是一个长度为 L 的位(bit)大数组 S ，将 S 平均分为 m 组，这个 m 组，就是对应多少轮，然后每组所占有的比特个数是平均的，设为 P。容易得出下面的关系：

L = S.length

L = m * p

以 K 为单位，S 占用的内存 = L / 8 / 1024

3、对应

假设访问用户 id 为：idn ， n->0,1,2,3....

在这个统计问题中，不同的用户 id 标识了一个用户，那么我们可以把用户的 id 作为被hash的输入。即：

hash(id) = 比特串

不同的用户 id，拥有不同的比特串。每一个比特串，也必然会至少出现一次 1 的位置。我们类比每一个比特串为一次伯努利试验。

现在要分轮，也就是分桶。所以我们可以设定，每个比特串的前多少位转为10进制后，其值就对应于所在桶的标号。假设比特串的低两位用来计算桶下标志，总共有4个桶，此时有一个用户的id的比特串是：1001011000011。它的所在桶下标为：1*2^1 + 1*2^0 = 3，处于第3个桶，即第3轮中。

上面例子中，计算出桶号后，剩下的比特串是：10010110000，从低位到高位看，第一次出现 1 的位置是 5 。也就是说，此时第3个桶中，k_max = 5。5 对应的二进制是：101，将 101 存入第3个桶。

模仿上面的流程，多个不同的用户 id，就被分散到不同的桶中去了，且每个桶有其 k_max。然后当要统计出某个页面有多少用户点击量的时候，就是一次估算。最终结合所有桶中的 k_max，代入估算公式，便能得出估算值。

Redis 中的 HyperLogLog 实现

Redis的实现中，HyperLogLog 占据12KB(占用内存为=16834 * 6 / 8 / 1024 = 12K)的大小，共设有16384 个桶，即：2^14 = 16384，每个桶有 6 位，每个桶可以表达的最大数字是：25+24+...+1 = 63 ，二进制为： 111 111 。

对于命令：pfadd key value

在存入时，value 会被 hash 成 64 位，即 64 bit 的比特字符串，前 14 位用来分桶，剩下50位用来记录第一个1出现的位置。

之所以选 14位 来表达桶编号是因为分了 16384 个桶，而 2^14 = 16384，刚好地，最大的时候可以把桶利用完，不造成浪费。假设一个字符串的前 14 位是：00 0000 0000 0010（从右往左看），其十进制值为 2。那么 value 对应转化后的值放到编号为 2 的桶。

index 的转化规则：

首先因为完整的 value 比特字符串是 64 位形式，减去 14 后，剩下 50 位，假设极端情况，出现 1 的位置，是在第 50 位，即位置是 50。此时 index = 50。此时先将 index 转为 2 进制，它是：110010 。

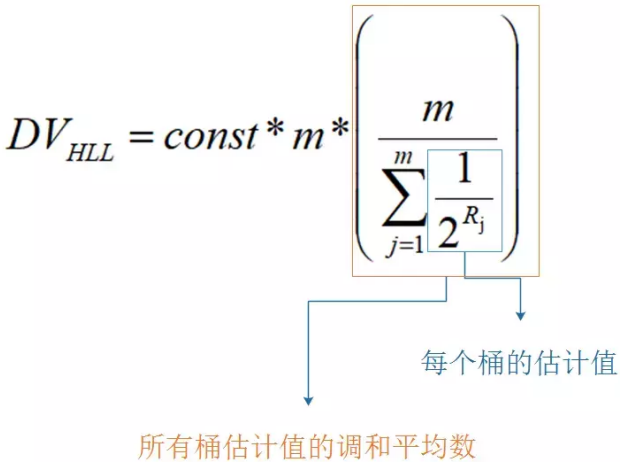
因为16384 个桶中，每个桶是 6 bit 组成的。于是 110010 就被设置到了第 2 号桶中去了。请注意，50 已经是最坏的情况，且它都被容纳进去了。那么其他的不用想也肯定能被容纳进去。

因为 fpadd 的 key 可以设置多个 value。例如下面的例子：

```
pfadd lgh golang
pfadd lgh python
pfadd lgh java
```

根据上面的做法，不同的 value，会被设置到不同桶中去，如果出现了在同一个桶的，即前 14 位值是一样的，但是后面出现 1 的位置不一样。那么比较原来的 index 是否比新 index 大。是，则替换。否，则不变。

最终地，一个 key 所对应的 16384 个桶都设置了很多的 value 了，每个桶有一个k_max。此时调用 pfcount 时，按照调和平均数进行估算，同时加以偏差修正，便可以计算出 key 的设置了多少次 value，也就是统计值，具体的估算公式如下：



value 被转为 64 位的比特串，最终被按照上面的做法记录到每个桶中去。64 位转为十进制就是：2^64，HyperLogLog 仅用了：16384 * 6 /8 / 1024 =12K 存储空间就能统计多达 2^64 个数。

同时，在具体的算法实现上，HLL还有一个分阶段偏差修正算法。我们就不做更深入的了解了。

事务

Redis事务

大家应该对事务比较了解，简单地说，事务表示一组动作，要么全部执行，要么全部不执行。例如在社交网站上用户A关注了用户B，那么需要在用户A的关注表中加入用户B，并且在用户B的粉丝表中添加用户A，这两个行为要么全部执行，要么全部不执行,否则会出现数据不一致的情况。

Redis提供了简单的事务功能，将一组需要一起执行的命令放到multi和exec两个命令之间。multi(['mʌlti]) 命令代表事务开始，exec(美[ɪg'zek])命令代表事务结束，如果要停止事务的执行，可以使用discard命令代替exec命令即可。

它们之间的命令是原子顺序执行的,例如下面操作实现了上述用户关注问题。

```
127.0.0.1:6880> multi
OK
127.0.0.1:6880(TX)> sadd u:a:follow ub
QUEUED
127.0.0.1:6880(TX)> sadd u:b:fans ua
QUEUED
```

可以看到sadd命令此时的返回结果是QUEUED，代表命令并没有真正执行，而是暂时保存在Redis中的一个缓存队列（所以discard也只是丢弃这个缓存队列中的未执行命令，并不会回滚已经操作过的数据，这一点要和关系型数据库的Rollback操作区分开）。如果此时另一个客户端执行sismember u:a:follow ub返回结果应该为0。

```
127.0.0.1:6880> sismember u:a:follow ub
(integer) 0
```

只有当exec执行后，用户A关注用户B的行为才算完成，如下所示exec返回的两个结果对应sadd命令。

```
127.0.0.1:6880> multi
OK
127.0.0.1:6880(TX)> sadd u:a:follow ub
QUEUED
127.0.0.1:6880(TX)> sadd u:b:fans ua
QUEUED
127.0.0.1:6880(TX)> exec
1) (integer) 1
2) (integer) 1
```

另一个客户端：

```
127.0.0.1:6880> sismember u:a:follow ub
(integer) 0
127.0.0.1:6880> sismember u:a:follow ub
(integer) 1
```

如果事务中的命令出现错误,Redis 的处理机制也不尽相同。

1、命令错误

例如下面操作错将set写成了sett，属于语法错误，会造成整个事务无法执行，key和counter的值未发生变化:

```
127.0.0.1:6880> set txkey hello
OK
127.0.0.1:6880> set txcount 100
OK
127.0.0.1:6880> mget txkey txcount
1) "hello"
2) "100"
127.0.0.1:6880> multi
OK
127.0.0.1:6880(TX)> sett txkey world
(error) ERR unknown command `sett`, with args beginning with: `txkey`, `world`,
127.0.0.1:6880(TX)> incr txcount
QUEUED
127.0.0.1:6880(TX)> exec
(error) EXECABORT Transaction discarded because of previous errors.
127.0.0.1:6880> mget txkey txcount
1) "hello"
2) "100"
```

2.运行时错误

例如用户B在添加粉丝列表时，误把sadd命令(针对集合)写成了zadd命令(针对有序集合)，这种就是运行时命令，因为语法是正确的:

```
127.0.0.1:6880> multi
OK
127.0.0.1:6880(TX)> sadd u:c:follow ub
QUEUED
127.0.0.1:6880(TX)> zadd u:b:fans 1 uc
QUEUED
127.0.0.1:6880(TX)> exec
1) (integer) 1
2) (error) WRONGTYPE Operation against a key holding the wrong kind of value
127.0.0.1:6880> sismember u:c:follow ub
(integer) 1
127.0.0.1:6880>
```

可以看到Redis并不支持回滚功能，sadd u:c:follow ub命令已经执行成功,开发人员需要自己修复这类问题。

有些应用场景需要在事务之前，确保事务中的key没有被其他客户端修改过，才执行事务，否则不执行(类似乐观锁)。Redis 提供了watch命令来解决这类问题。

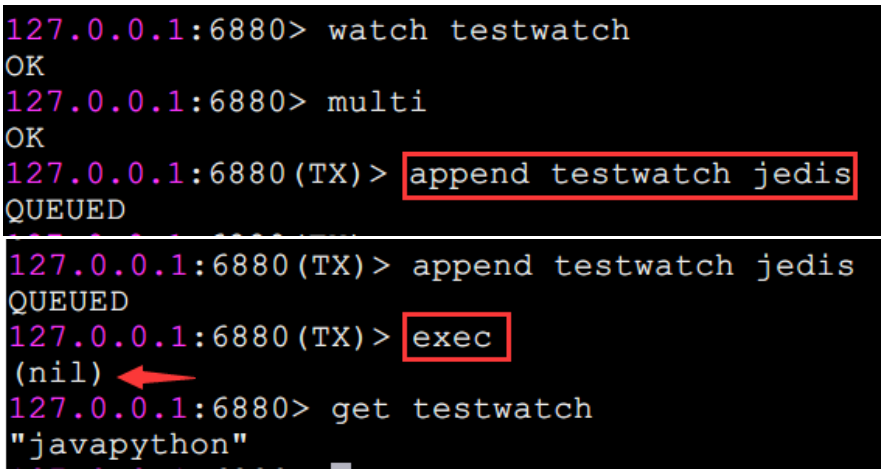
客户端1：

```
127.0.0.1:6880> set testwatch java
OK
127.0.0.1:6880> watch testwatch
OK
127.0.0.1:6880> multi
OK
127.0.0.1:6880(TX)>
```

客户端2：

```
127.0.0.1:6880> append testwatch python
(integer) 10
127.0.0.1:6880>
```

客户端1继续：



可以看到“客户端-1”在执行multi之前执行了watch命令，“客户端-2”在“客户端-1”执行exec之前修改了key值，造成客户端-1事务没有执行(exec结果为nil)。

Redis客户端中的事务使用代码参见：

cn.tuling.redis.adv.RedisTransaction

Pipeline和事务的区别

简单来说，

1、pipeline是客户端的行为，对于服务器来说是透明的，可以认为服务器无法区分客户端发送来的查询命令是以普通命令的形式还是以pipeline的形式发送到服务器的；

2而事务则是实现在服务器端的行为，用户执行MULTI命令时，服务器会将对应这个用户的客户端对象设置为一个特殊的状态，在这个状态下后续用户执行的查询命令不会被真的执行，而是被服务器缓存起来，直到用户执行EXEC命令为止，服务器会将这个用户对应的客户端对象中缓存的命令按照提交的顺序依次执行。

3、应用pipeline可以提服务器的吞吐能力，并提高Redis处理查询请求的能力。

但是这里存在一个问题，当通过pipeline提交的查询命令数据较少，可以被内核缓冲区所容纳时，Redis可以保证这些命令执行的原子性。然而一旦数据量过大，超过了内核缓冲区的接收大小，那么命令的执行将会被打断，原子性也就无法得到保证。因此pipeline只是一种提升服务器吞吐能力的机制，如果想要命令以事务的方式原子性的被执行，还是需要事务机制，或者使用更高级的脚本功能以及模块功能。

4、可以将事务和pipeline结合起来使用，减少事务的命令在网络上的传输时间，将多次网络IO缩减为一次网络IO。

Redis提供了简单的事务，之所以说它简单，主要是因为它不支持事务中的回滚特性,同时无法实现命令之间的逻辑关系计算，当然也体现了Redis 的“keep it simple”的特性。

Redis 7.0前瞻

2022年2月初，Redis 7.0 迎来了首个候选发布（RC）版本。这款内存键值数据库迎来了“重大的性能优化”和其它功能改进，性能优化包括降低写入时复制内存的开销、提升内存效率，改进 fsync 来避免大量的磁盘写入和优化延迟表现。

Redis 7.0-rc1 的其它一些变动，包括将“Redis 函数”作为新的服务器端脚本功能，细粒度 / 基于键的权限、改进子命令处理 / Lua 脚本 / 各种新命令。

此外也提供了一些安全改进。

我们从分析 Redis 主从复制中的内存消耗过多和堵塞问题，以及 Redis 7.0（尚未发布）的共享复制缓冲区方案是如何解决这些问题的。

Redis 主从复制原理

我们先简单回顾一下 Redis 主从复制的基本原理。Redis 的主从复制主要分为两种情况：

全量同步

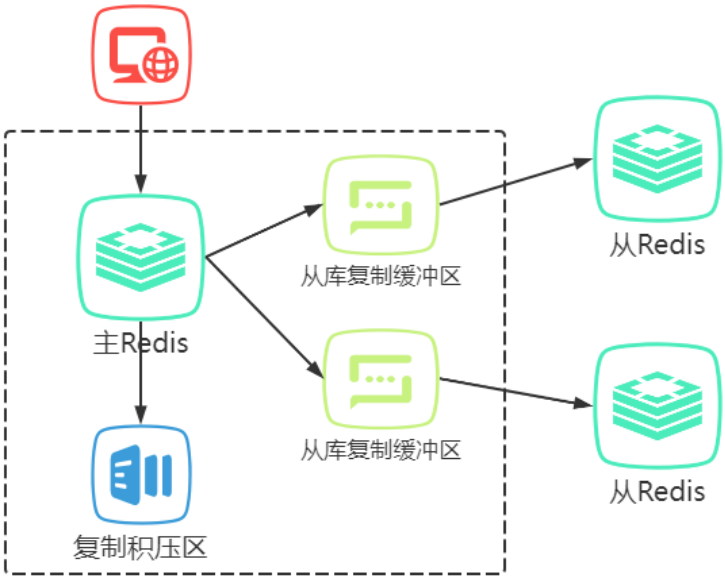
主库通过 fork 子进程产生内存快照，然后将数据序列化为 RDB 格式同步到从库，使从库的数据与主库某一时刻的数据一致。

命令传播

当从库与主库完成全量同步后，进入命令传播阶段，主库将变更数据的命令发送到从库，从库将执行相应命令，使从库与主库数据持续保持一致。

Redis 复制缓存区相关问题分析

多从库时主库内存占用过多



如上图所示，对于 Redis 主库，当用户的写请求到达时，主库会将变更命令分别写入所有从库复制缓冲区（OutputBuffer），以及复制积压区（ReplicationBacklog）。全量同步时依然会执行该逻辑，所以在全量同步阶段经常会触发 client-output-buffer-limit，主库断开与从库的连接，导致主从同步失败，甚至出现循环持续失败的情况。

该实现一个明显的问题是内存占用过多，所有从库的连接在主库上是独立的，也就是说每个从库 OutputBuffer 占用的内存空间也是独立的，那么主从复制消耗的内存就是所有从库缓冲区内存大小之和。如果我们设定从库的 client-output-buffer-limit 为 1GB，如果有三个从库，则在主库上可能会消耗 3GB 的内存用于主从复制。另外，真实环境中从库的数量不是确定的，这也导致 Redis 实例的内存消耗不可控。

OutputBuffer 拷贝和释放的堵塞问题

Redis 为了提升多从库全量复制的效率和减少 fork 产生 RDB 的次数，会尽可能的让多个从库共用一个 RDB，从代码(replication.c)上看：

```
/* To attach this slave, we check that it has at least all the
 * capabilities of the slave that triggered the current BGSAVE. */
if (ln && ((c->slave_capa & slave->slave_capa) == slave->slave_capa)) {
    /* Perfect, the server is already registering differences for
     * another slave. Set the right state, and copy the buffer.
     * We don't copy buffer if clients don't want. */
    if (!(c->flags & CLIENT_REPL_RDBONLY)) copyClientOutputBuffer(c,slave);
    replicationSetupSlaveForFullResync(c,slave->psync_initial_offset);
    serverLog(LL_NOTICE,"Waiting for end of BGSAVE for SYNC");
} else {
    /* No way, we need to wait for the next BGSAVE in order to
     * register differences. */
    serverLog(LL_NOTICE,"Can't attach the replica to the current BGSAVE. Wa
}
```

当已经有一个从库触发 RDB BGSAVE 时，后续需要全量同步的从库会共享这次 BGSAVE 的 RDB，为了从库复制数据的完整性，会将之前从库的 OutputBuffer 拷贝到请求全量同步从库的 OutputBuffer 中。

其中的copyClientOutputBuffer 可能存在堵塞问题，因为 OutputBuffer 链表上的数据可达数百 MB 甚至数 GB 之多，对其拷贝可能使用百毫秒甚至秒级的时间，而且该堵塞问题没法通过日志或者 latency 观察到，但对Redis性能影响却很大。

同样地，当 OutputBuffer 大小触发 limit 限制时，Redis 就是关闭该从库链接，而在释放 OutputBuffer 时，也需要释放数百 MB 甚至数 GB 的数据，其耗时对 Redis 而言也很长。

ReplicationBacklog 的限制

我们知道复制积压缓冲区 ReplicationBacklog 是 Redis 实现部分重同步的基础，如果从库可以进行增量同步，则主库会从 ReplicationBacklog 中拷贝从库缺失的数据到其 OutputBuffer。拷贝的数据量最大当然是 ReplicationBacklog 的大小，为了避免拷贝数据过多的问题，通常不会让该值过大，一般百兆左右。但在大容量实例中，为了避免由于主从网络中断导致的全量同步，又希望该值大一些，这就存在矛盾了。

而且如果重新设置 ReplicationBacklog 大小时，会导致 ReplicationBacklog 中的内容全部清空，所以如果在变更该配置期间发生主从断链重连，则很有可能导致全量同步。

Redis7.0共享复制缓存区的设计与实现

简述

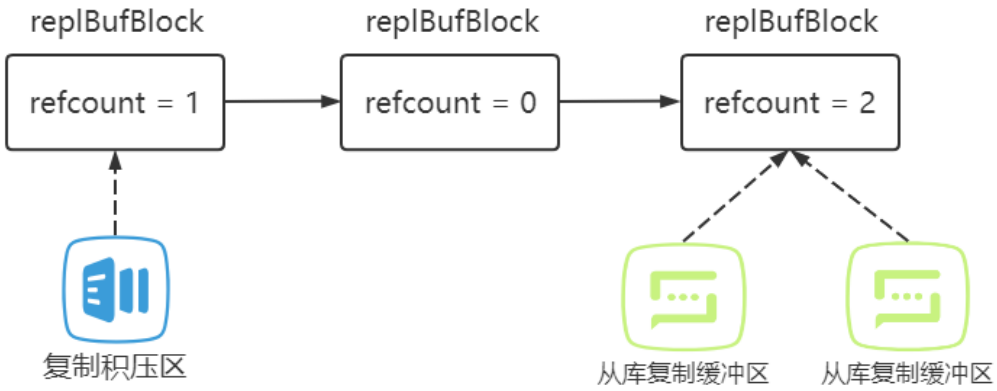
每个从库在主库上单独拥有自己的 OutputBuffer，但其存储的内容却是一样的，一个最直观的想法就是主库在命令传播时，将这些命令放在一个全局的复制数据缓冲区中，多个从库共享这份数据，不同的从库对引用复制数据缓冲区中不同的内容，这就是『共享复制缓存区』方案的核心思想。实际上，复制积压缓冲区（ReplicationBacklog）中的内容与从库 OutputBuffer 中的数据也是一样的，所以该方案中，ReplicationBacklog 和从库一样共享一份复制缓冲区的数据，也避免了 ReplicationBacklog 的内存开销。

『共享复制缓存区』方案中复制缓冲区（ReplicationBuffer）的表示采用链表的表示方法，将 ReplicationBuffer 数据切割为多个 16KB 的数据块（replBufBlock），然后使用链表来维护起来。为了维护不同从库的对 ReplicationBuffer 的使用信息，在 replBufBlock 中存在字段：

refcount：block 的引用计数

id：block 的唯一标识，单调递增的数值

repl_offset：block 开始的复制偏移



ReplicationBuffer 由多个 replBufBlock 组成链表，当 复制积压区 或从库对某个 block 使用时，便对正在使用的 replBufBlock 增加引用计数，上图中可以看到，复制积压区正在使用的 replBufBlock refcount 是 1，从库 A 和 B 正在使用的 replBufBlock refcount 是 2。当从库使用完当前的 replBufBlock（已经将数据发送给从库）时，就会对其 refcount 减 1 而且移动到下一个 replBufBlock，并对其 refcount 加 1。

堵塞问题和限制问题的解决

多从库消耗内存过多的问题通过共享复制缓存区方案得到了解决，对于OutputBuffer 拷贝和释放的堵塞问题和 ReplicationBacklog 的限制问题是否解决了呢？

首先来看 OutputBuffer 拷贝和释放的堵塞问题问题， 这个问题很好解决，因为ReplicationBuffer 是个链表实现，当前从库的 OutputBuffer 只需要维护共享 ReplicationBuffer 的引用信息即可。所以无需进行数据深拷贝，只需要更新引用信息，即对正在使用的 replBufBlock refcount 加 1，这仅仅是一条简单的赋值操作，非常轻量。OutputBuffer 释放问题呢？在当前的方案中释放从库 OutputBuffer 就变成了对其正在使用的 replBufBlock refcount 减 1，也是一条赋值操作，不会有任何阻塞。

对于ReplicationBacklog 的限制问题也很容易解决了，因为 ReplicatonBacklog 也只是记录了对 ReplicationBuffer 的引用信息，对 ReplicatonBacklog 的拷贝也仅仅成了找到正确的 replBufBlock，然后对其 refcount 加 1。这样的话就不用担心 ReplicatonBacklog 过大导致的拷贝堵塞问题。而且对 ReplicatonBacklog 大小的变更也仅仅是配置的变更，不会清掉数据。

ReplicationBuffer 的裁剪和释放

ReplicationBuffer 不可能无限增长，Redis 有相应的逻辑对其进行裁剪，简单来说，Redis 会从头访问 replBufBlock 链表，如果发现 replBufBlock refcount 为 0，则会释放它，直到迭代到第一个 replBufBlock refcount 不为 0 才停止。所以想要释放 ReplicationBuffer，只需要减少相应 ReplBufBlock 的 refcount，会减少 refcount 的主要情况有：

1、当从库使用完当前的 replBufBlock 会对其 refcount 减 1；

2、当从库断开链接时会对正在引用的 replBufBlock refcount 减 1，无论是因为超过 client-output-buffer-limit 导致的断开还是网络原因导致的断开；

3、当 ReplicationBacklog 引用的 replBufBlock 数据量超过设置的该值大小时，会对正在引用的 replBufBlock refcount 减 1，以尝试释放内存；

不过当一个从库引用的 replBufBlock 过多，它断开时释放的 replBufBlock 可能很多，也可能造成堵塞问题，所以Redis7里会限制一次释放的个数，未及时释放的内存在系统的定时任务中渐进式释放。

数据结构的选择

当从库尝试与主库进行增量重同步时，会发送自己的 repl_offset，主库在每个 replBufBlock 中记录了该其第一个字节对应的 repl_offset，但如何高效地从数万个 replBufBlock 的链表中找到特定的那个？

从链表的性质我们知道，链表只能直接从头到位遍历链表查找对应的 replBufBlock ，这个操作必然会耗费较多时间而堵塞服务。有什么改进的思路？可以额外使用一个链表用于索引固定区间间隔的 replBufBlock，每 1000 个 replBufBlock 记录一个索引信息，当查找 repl_offset 时，会先从索引链表中查起，然后再查找 replBufBlock 链表，这个就类似于跳表的查找实现。Redis 的 zset就是跳表的实现：

在极端场景下可能会查找超过千次，有 10 毫秒以上的延迟，所以Redis 7没有使用这种数据结构。

最终使用 rax 树实现了对 replBufBlock 固定区间间隔的索引，每 64 个记录一个索引点。一方面，rax 索引占用的内存较少；另一方面，查询效率也是非常高，理论上查找比较次数不会超过 100，耗时在 1 毫秒以内。

rax树

Redis中还有其他地方使用了Rax树，比如我们前面学习过的streams 这个类型里面的 consumer group(消费者组) 的名称还有和Redis集群名称存储。

RAX叫做基数树（前缀压缩树），就是有相同前缀的字符串，其前缀可以作为一个公共的父节点，什么又叫前缀树？

Trie树

即字典树，也有的称为前缀树，是一种树形结构。广泛应用于统计和排序大量的字符串（但不仅限于字符串），所以经常被搜索引擎系统用于文本词频统计。它的优点是最大限度地减少无谓的字符串比较，查询效率比较高。

Trie的核心思想是空间换时间，利用字符串的公共前缀来降低查询时间的开销以达到提高效率的目的。

先看一下几个场景问题：

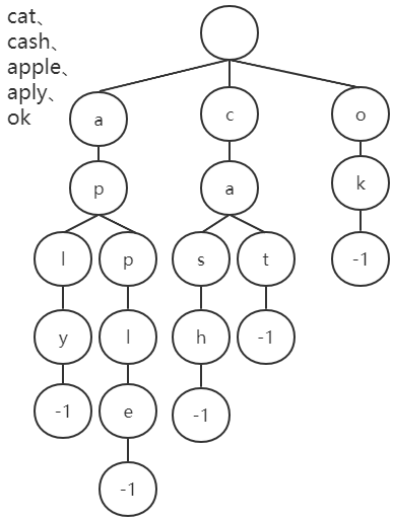
1.我们输入n个单词，每次查询一个单词，需要回答出这个单词是否在之前输入的n单词中出现过。

答：当然是用map来实现。

2.我们输入n个单词，每次查询一个单词的前缀，需要回答出这个前缀是之前输入的n单词中多少个单词的前缀？

答：还是可以用map做，把输入n个单词中的每一个单词的前缀分别存入map中，然后计数，这样的话复杂度会非常的高。若有n个单词，平均每个单词的长度为c，那么复杂度就会达到nc。

因此我们需要更加高效的数据结构，这时候就是Trie树的用武之地了。现在我们通过例子来理解什么是Trie树。现在我们对cat、cash、apple、aply、ok这几个单词建立一颗Trie树。



从图中可以看出：

1.每一个节点代表一个字符

2.有相同前缀的单词在树中就有公共的前缀节点。

3.整棵树的根节点是空的。

4.每个节点结束的时候用一个特殊的标记来表示，这里我们用-1来表示结束，从根节点到-1所经过的所有的节点对应一个英文单词。

5.查询和插入的时间复杂度为O(k)，k为字符串长度，当然如果大量字符串没有共同前缀时还是很耗内存的。

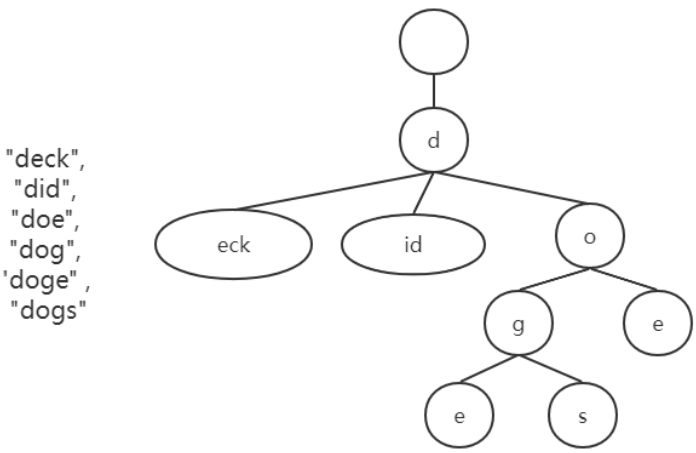
所以，总的来说，Trie树把很多的公共前缀独立出来共享了。这样避免了很多重复的存储。想想字典集的方式，一个个的key被单独的存储，即使他们都有公共的前缀也要单独存储。相比字典集的方式，Trie树显然节省更多的空间。

Trie树其实依然比较浪费空间，比如我们前面所说的“然如果大量字符串没有共同前缀时”。

比如这个字符串列表："deck", "did", "doe", "dog", "doge" , "dogs"。"deck"这一个分支，有没有必要一直往下来拆分吗？还是"did"，存在着一样的问题。像这样的不可分叉的单支分支，其实完全可以合并，也就是压缩。

Radix树：压缩后的Trie树

所以Radix树就是压缩后的Trie树，因此也叫压缩Trie树。比如上面的字符串列表完全可以这样存储：



同时在具体存储上，Radix树的处理是以bit（或二进制数字）来读取的。一次被对比r个bit。

比如"dog", "doge" , "dogs"，按照人类可读的形式，dog是dogs和doge的子串。

但是如果按照计算机的二进制比对：

dog: 01100100 01101111 01100111

doge: 01100100 01101111 01100111 01100101

dogs: 01100100 01101111 01100111 01110011

可以发现dog和doge是在第二十五位的时候不一样的。dogs和doge是在第二十八位不一样的，按照位的比对的结果，doge是dogs二进制子串，这样在存储时可以进一步压缩空间。

本文档分享地址

https://note.youdao.com/s/UQck7LmD