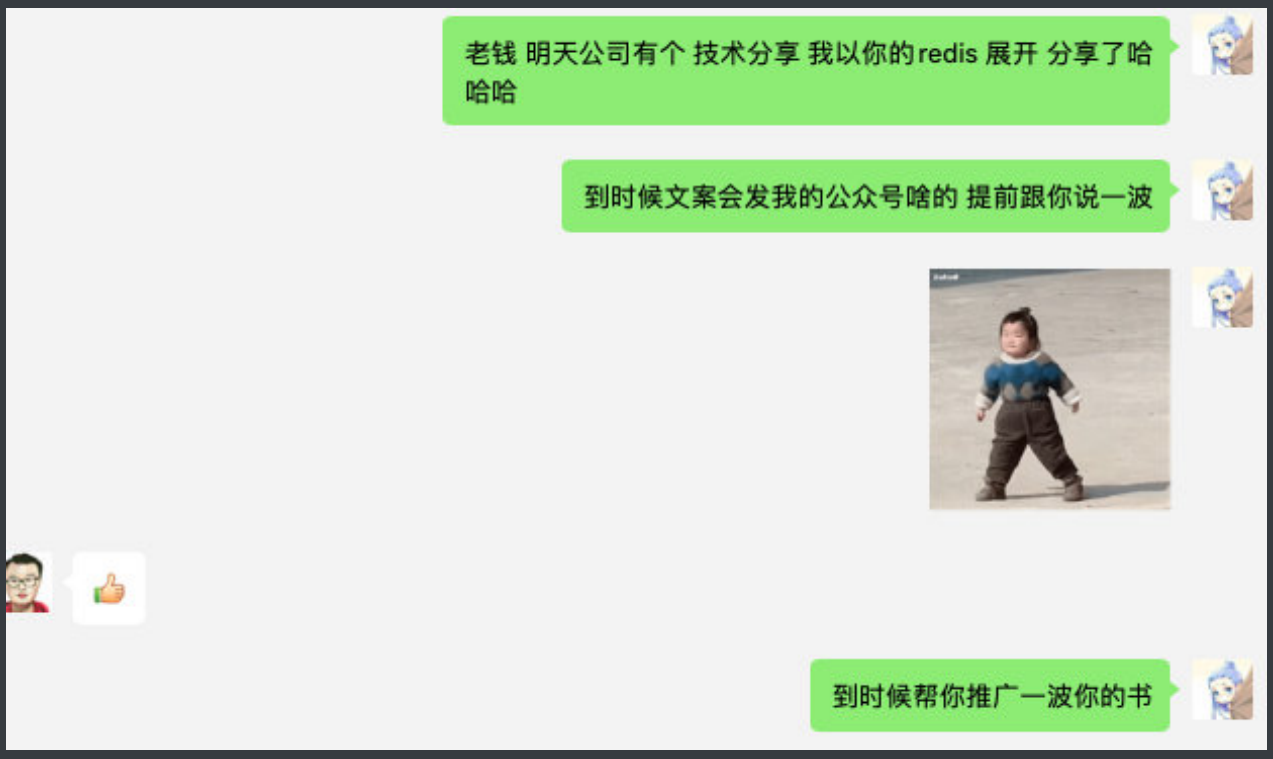
本期呢是我在蘑菇街算法工程团队做技术分享的一个文字版本，我后面有机会做一下视频的线上版本。

其实在做这个分享的时候我很纠结，我都不知道当时为啥自己选了Redis，这个我是知道比较多点，但是作为技术分享就不是很适合了，大家都知道Redis是啥怎么用，还不如去学一点冷门的分享一下。

所以准备阶段也有点小纠结，好在我认识Redis的大佬，**老钱（前掌阅技术专家，现字节跳动技术专家，《Redis深度历险》作者）**，我就有了一点思路，按照他的一个技术体系，再结合一些实战去做分享，**本文主要内容来自他的书籍**，因为我发现他都写了，我没必要做重复的事情。



因为丙之前是做电商的活动嘛，缓存接触也是很多的，缓存的问题也多场景也是相当丰富，所以分享下来大家倒也get了一些点，最后还算一次不算失败的分享吧。

正文

大家都知道Redis主要作用就是缓存，知道他的五种基础类型，那你们知道他的一些高级用法么？或者他基础数据类型底层的，SDS动态字符，链表，字典，跳跃表么？以及相关的底层衍生比如字典的渐进式hash，集合的升级降级么？

大家可能知道，可能不知道，没关系，不知道的课后看看，因为上面的我都不分享，我主要是分享一下，redis在我们实际开发中的一些高级用法，和一些稀少的用法。

这些用法大家不一定都用得到，但是了解了，哪天你有这样的场景的时候你至少有个印象，你要是都不知道，那你谷歌都不知道怎么谷歌对吧？

注：本文的主要思路是我工作实践以及老钱的《Redis深度历险》中来的，大家都去安排一下哈哈，这本书有一说一瑕疵还是有，但是不影响书本整体内容的质量，很多都是老钱工作多年的经验之谈。

Redis可以做什么？

正常大家知道的就是缓存，又或者分布式情况的分布式锁，还有么？

那丙不是一直都是写博客嘛，我就说一下博客哪些场景会用到redis：

1. 记录帖子的点赞数、评论数和点击数 (hash)。
2. 记录用户的帖子 ID 列表 (排序)，便于快速显示用户的帖子列表 (zset)。
3. 记录帖子的标题、摘要、作者和封面信息，用于列表页展示 (hash)。
4. 记录帖子的点赞用户 ID 列表，评论 ID 列表，用于显示和去重计数 (zset)。
5. 缓存近期热帖内容 (帖子内容空间占用比较大)，减少数据库压力 (hash)。
6. 记录帖子的相关文章 ID，根据内容推荐相关帖子 (list)。
7. 如果帖子 ID 是整数自增的，可以使用 Redis 来分配帖子 ID(计数器)。
8. 收藏集和帖子之间的关系 (zset)。
9. 记录热榜帖子 ID 列表，总热榜和分类热榜 (zset)。
10. 缓存用户行为历史，进行恶意行为过滤 (zset,hash)。
11. 数据推送去重Bloom filter
12. pv，uv统计

大家是不是从未发现Redis居然有这么多的用法？

既然是一次分享嘛，那我们就在看正式内容之前看一个有趣的东西。

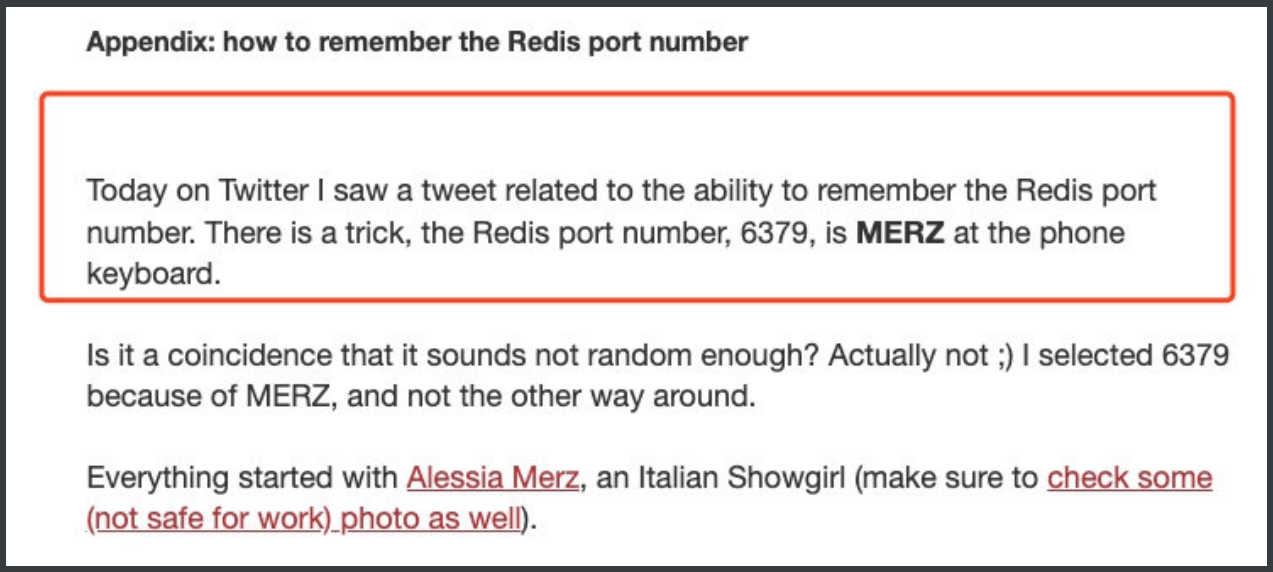
大家知道Redis端口为啥是6379么？

Redis 由意大利人 Salvatore Sanfilippo（网名 Antirez） 开发，Antirez 不仅帅的不像实力派，也非常有趣。

他出生在非英语系国家，英语能力长期以来是一个短板，他曾经专门为自己蹩脚的英语能力写过一篇博文《英语伤痛 15 年》，用自己的成长经历来鼓励那些非英语系的技术开发者们努力攻克英语难关。

我们都知道 Redis 的默认端口是 6379，这个端口号也不是随机选的，而是由手机键盘字母「MERZ」的位置决定的。

「MERZ」在 Antirez 的朋友圈语言中是「愚蠢」的代名词，它源于意大利广告女郎「Alessia Merz」在电视节目上说了一堆愚蠢的话 Antirez 今年已经四十岁了，依旧在孜孜不倦地写代码，为 Redis 的开源事业持续贡献力量。



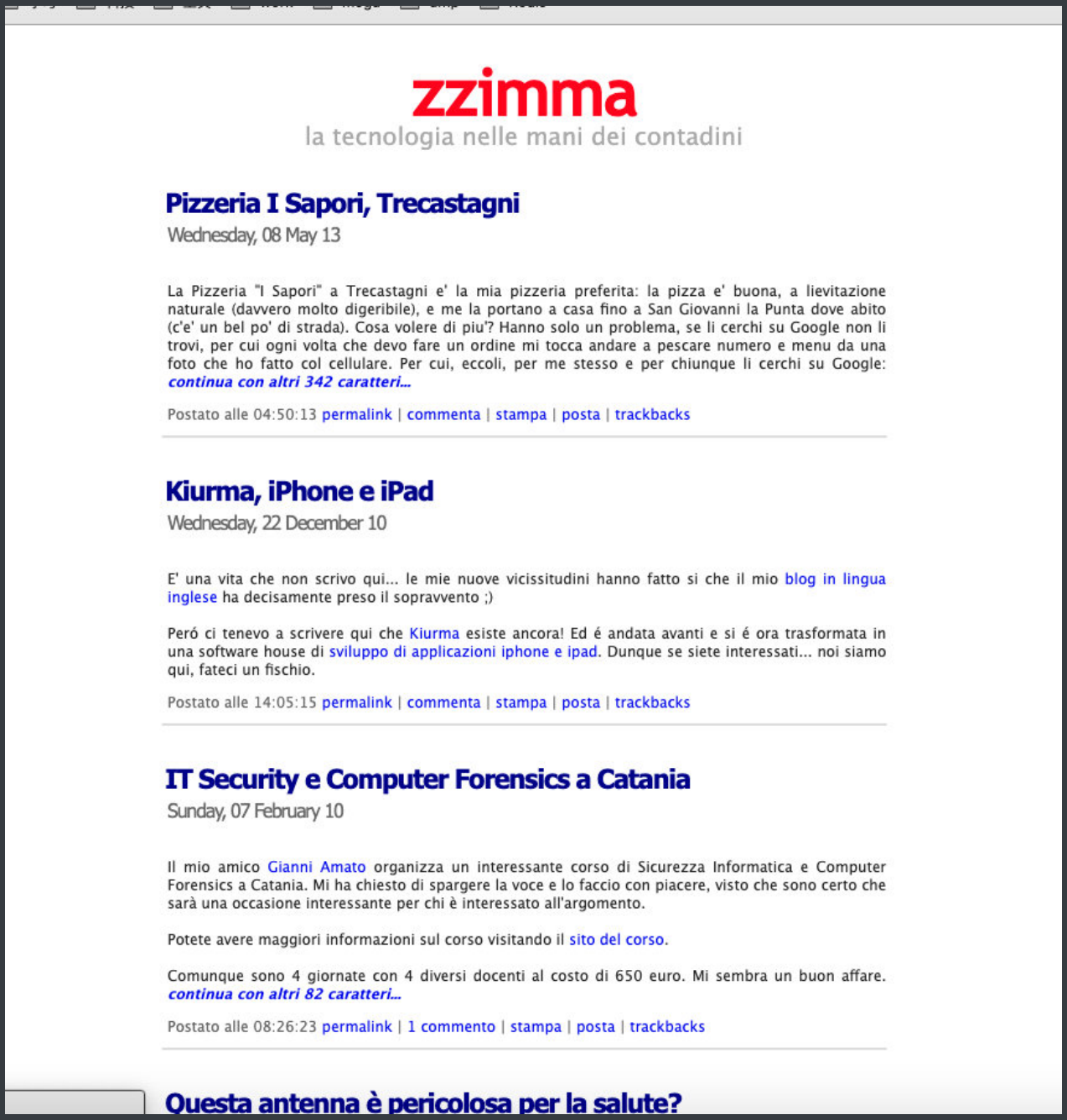
Appendix: how to remember the Redis port number

Today on Twitter I saw a tweet related to the ability to remember the Redis port number. There is a trick, the Redis port number, 6379, is **MERZ** at the phone keyboard.

Is it a coincidence that it sounds not random enough? Actually not ;) I selected 6379 because of MERZ, and not the other way around.

Everything started with Alessia Merz, an Italian Showgirl (make sure to check some (not safe for work) photo as well).

他的博客地址：http://oldblog.antirez.com/post/redis-as-LRU-cache.html 包括上面的英语伤痛15年也是有的，他是意大利人嘛，所以英文的博客不是那么多，不过他最近更新的大部分都是英文的了。



好了闲聊的东西都聊完了，Redis的爸爸我们也认识了，也知道6379的端口含义来源了，那我们就开始聊他的一些应用吧，其实这里面提到的redis技术栈几乎我都写过了，大家不知道能不能发现哈哈。

分布式锁

分布式锁本质上要实现的目标就是在 Redis 里面占一个“茅坑”，当别的进程也要来占时，发现已经有人蹲在那里了，就只好放弃或者稍后再试。

占坑一般是使用 setnx(set if not exists) 指令，只允许被一个客户端占坑，先来先占， 用完了，再调用 del 指令释放茅坑。

```
setnx aobing
expire aobing
del aobing
```

但是引入后会有问题，原子性，怎么解决，就比如setnx成功，设置失效时间expire的时候失败，怎么办?

当时出现了贼多的第三方插件，作者为了解决这个乱象，就在Redis 2.8 版本中作者加入了 set 指令的扩展参数:

```
set  aobing ture  ex 5 nx
del  aobing
```

但是这样还是有问题超时问题，可重入问题等等，这个时候，第三方的一些插件就横空出世了，Redission ，Jedis，他们的底层我就不过多描述了，都是通过lua脚本去保证的，大致逻辑跟我们代码实现是差不多的。

就比如去删除的时候，去校验是否当前线程锁定的，就把比较和删除这样一些动作都放到一起了:

```
# delifequals
if redis.call("get",KEYS[1]) == ARGV[1] then
    return redis.call("del",KEYS[1])
else
    return 0
end
```

延时队列

我们平时习惯于使用 Rabbitmq 、RocketMQ和 Kafka 作为消息队列中间件，来给应用程序之间增加异步消息传递功能。这两个中间件都是专业的消息队列中间件，特性之多超出了大多数人的理解能力。

使用过 Rabbitmq 的同学知道它使用起来有多复杂，发消息之前要创建 Exchange，再创建 Queue，还要将 Queue 和 Exchange 通过某种规则绑定起来，发消息的时候要指定 routing-key，还要控制头部信息。消费者在消费消息之前也要进行上面一系列的繁琐过程。

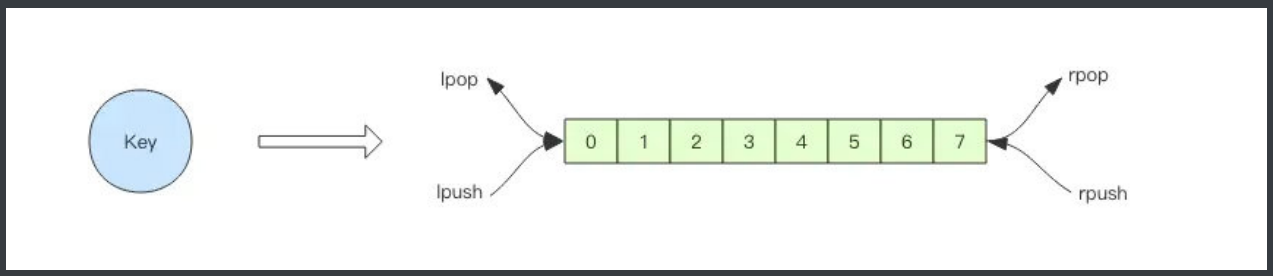
但是绝大多数情况下，虽然我们的消息队列只有一组消费者，但还是需要经历上面这些繁琐的过程。

有了 Redis，它就可以让我们解脱出来，对于那些只有一组消费者的消息队列，使用 Redis 就可以非常轻松的搞定。

Redis 的消息队列不是专业的消息队列，它没有非常多的高级特性，没有 ack 保证，如果对消息的可靠性有着极致的追求，那么它就不适合使用。

```
> rpush notify-queue apple banana pear
(integer) 3
> llen notify-queue
(integer) 3
```

```
> lpop notify-queue
"apple"
> llen notify-queue
(integer) 2
> lpop notify-queue
"banana"
> llen notify-queue
(integer) 1
> lpop notify-queue
"pear"
> llen notify-queue
(integer) 0
> lpop notify-queue
(nil)
```



但是这样有问题大家发现没有？队列会空是吧，那怎么解决呢?

客户端是通过队列的 pop 操作来获取消息，然后进行处理，处理完了再接着获取消息，再进行处理。

如此循环往复，这便是作为队列消费者的客户端的生命周期。

可是如果队列空了，客户端就会陷入 pop 的死循环，不停地 pop，没有数据，接着再 pop，又没有数据，这就是浪费生命的空轮询。

空轮询不但拉高了客户端的 CPU，redis 的 QPS 也会被拉高，如果这样空轮询的客户端有几十来个，Redis 的慢查询可能会显著增多。

解决方式很简单，让线程睡一秒 Thread.sleep(1000)

Redis在我开发的一些简易后台我确实有用到，因为我觉得没必要接入消息队列中间件，大家平时开发小系统可以试试。

位图bitmap

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。

这个类型不仅仅可以用来让我们改二进制改字符串值，最经典的就是用户连续签到。

key 可以设置为 前缀:用户id:年月 譬如 setbit sign:123:1909 0 1

代表用户ID=123签到，签到的时间是19年9月份，0代表该月第一天，1代表签到了

第二天没有签到，无需处理，系统默认为0

第三天签到 setbit sign:123:1909 2 1

可以查看一下目前的签到情况，显示第一天和第三天签到了，前8天目前共签到了2天

```
127.0.0.1:6379> setbit sign:123:1909 0 1
0
127.0.0.1:6379> setbit sign:123:1909 2 1
0
127.0.0.1:6379> getbit sign:123:1909 0
1
127.0.0.1:6379> getbit sign:123:1909 1
0
127.0.0.1:6379> getbit sign:123:1909 2
1
127.0.0.1:6379> getbit sign:123:1909 3
0
127.0.0.1:6379> bitcount sign:123:1909 0 0
2
```

HyperLogLog

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。

你也许已经想到了一个简单的方案，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。没错，这是一个非常简单的方案。

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？其实老板需要的数据又不需要太精确，105w 和 106w 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?

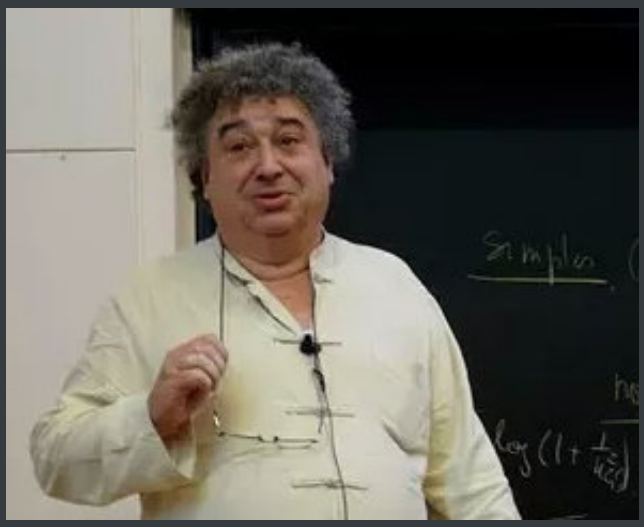
HyperLogLog 提供了两个指令 pfadd 和 pfcount，根据字面意义很好理解，一个是增加计数，一个是获取计数。

pfadd 用法和 set 集合的 sadd 是一样的，来一个用户 ID，就将用户 ID 塞进去就是，pfcount 和 scard 用法是一样的，直接获取计数值。

```
127.0.0.1:6379> pfadd codehole user1
(integer) 1
127.0.0.1:6379> pfcount codehole
(integer) 1
127.0.0.1:6379> pfadd codehole user2
(integer) 1
127.0.0.1:6379> pfcount codehole
(integer) 2
127.0.0.1:6379> pfadd codehole user3
(integer) 1
127.0.0.1:6379> pfcount codehole
(integer) 3
127.0.0.1:6379> pfadd codehole user4
(integer) 1
127.0.0.1:6379> pfcount codehole
(integer) 4
```

```
127.0.0.1:6379> pfadd codehole user5
(integer) 1
127.0.0.1:6379> pfcount codehole
(integer) 5
127.0.0.1:6379> pfadd codehole user6
(integer) 1
127.0.0.1:6379> pfcount codehole
(integer) 6
127.0.0.1:6379> pfadd codehole user7 user8 user9 user10
(integer) 1
127.0.0.1:6379> pfcount codehole
(integer) 10
```

pfadd 这个 pf 是什么意思?



它是 HyperLogLog 这个数据结构的发明人 Philippe Flajolet 的首字母缩写，老师觉得他发型很酷，看起来是个佛系教授。

他底层有点复杂，他是怎么做到这么小的结构，存储这么多数据的？也是很取巧大家有空可以看下我之前的文章。

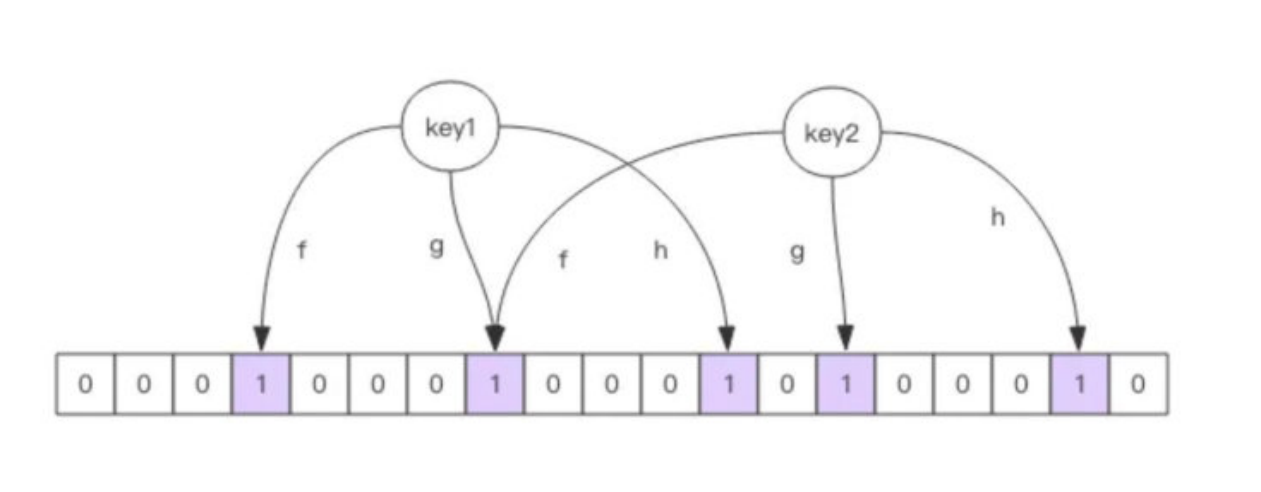
布隆过滤器

HyperLogLog 数据结构来进行估数，它非常有价值，可以解决很多精确度不高的统计需求。

但是如果我们想知道某一个值是不是已经在 HyperLogLog 结构里面了，它就无能为力了，它只提供了 pfadd 和 pfcount 方法，没有提供 pfcontains 这种方法。

讲个使用场景，比如我们在使用新闻客户端看新闻时，它会给我们不停地推荐新的内容，它每次推荐时要去重，去掉那些已经看过的内容。问题来了，新闻客户端推荐系统如何实现推送去重的?

你会想到服务器记录了用户看过的所有历史记录，当推荐系统推荐新闻时会从每个用户的历史记录里进行筛选，过滤掉那些已经存在的记录。问题是当用户量很大，每个用户看过的新闻又很多的情况下，这种方式，推荐系统的去重工作在性能上跟的上么？




```
127.0.0.1:6379> bf.add codehole user1
(integer) 1
127.0.0.1:6379> bf.add codehole user2
(integer) 1
127.0.0.1:6379> bf.add codehole user3
(integer) 1
127.0.0.1:6379> bf.exists codehole user1
(integer) 1
127.0.0.1:6379> bf.exists codehole user2
(integer) 1
127.0.0.1:6379> bf.exists codehole user3
(integer) 1
127.0.0.1:6379> bf.exists codehole user4
(integer) 0
127.0.0.1:6379> bf.madd codehole user4 user5 user6
1) (integer) 1
2) (integer) 1
3) (integer) 1
127.0.0.1:6379> bf.mexists codehole user4 user5 user6 user7
1) (integer) 1
2) (integer) 1
3) (integer) 1
4) (integer) 0
```

布隆过滤器的initial_size估计的过大，会浪费存储空间，估计的过小，就会影响准确率，用户在使用之前一定要尽可能地精确估计好元素数量，还需要加上一定的冗余空间以避免实际元素可能会意外高出估计值很多。

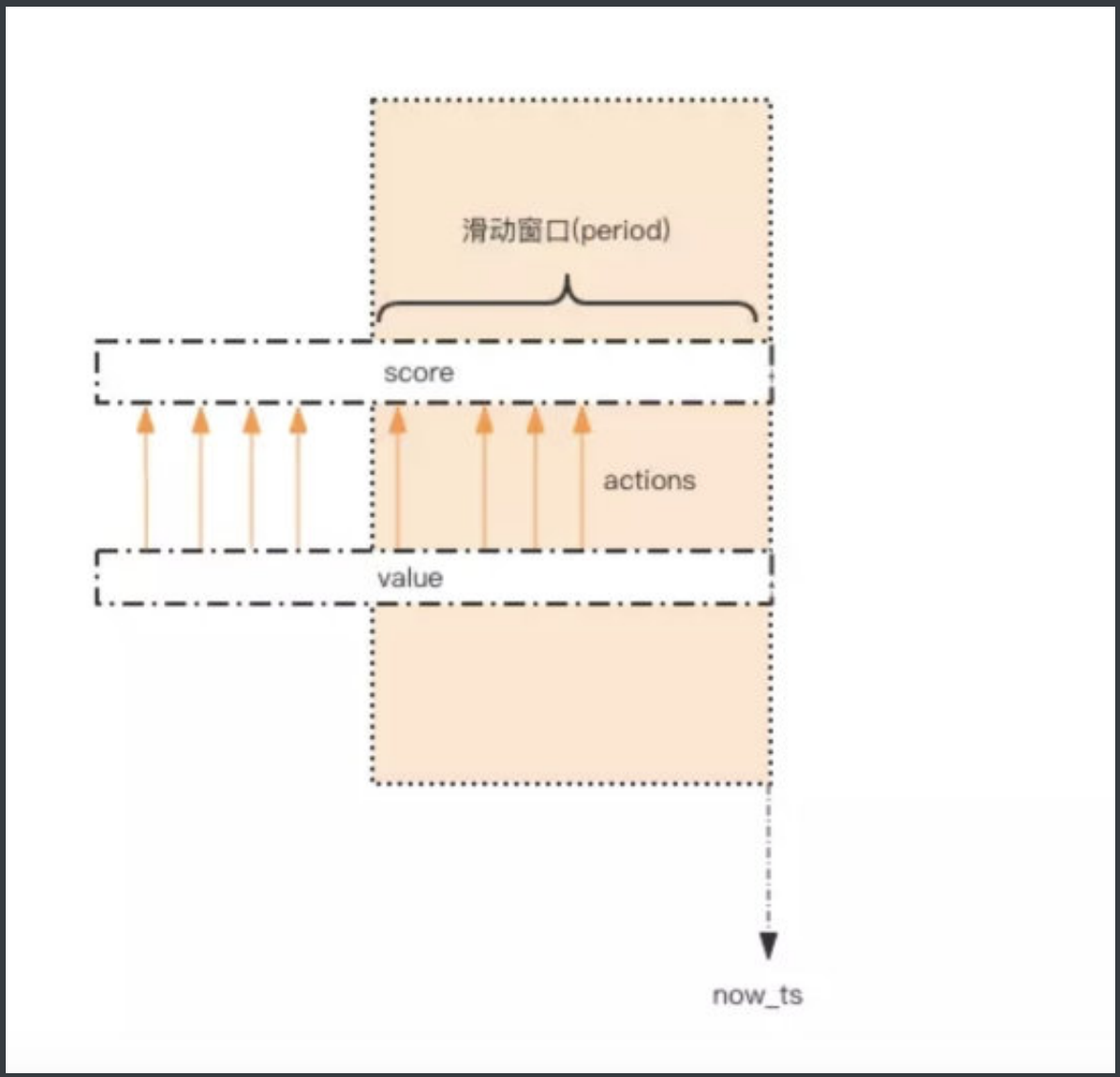
布隆过滤器的error_rate越小，需要的存储空间就越大，对于不需要过于精确的场合，error_rate设置稍大一点也无伤大雅。比如在新闻去重上而言，误判率高一点只会让小部分文章不能让合适的人看到，文章的整体阅读量不会因为这点误判率就带来巨大的改变。

在爬虫系统中，我们需要对 URL 进行去重，已经爬过的网页就可以不用爬了。但是 URL 太多了，几千万几个亿，如果用一个集合装下这些 URL 地址那是非常浪费空间的。这时候就可以考虑使用布隆过滤器。它可以大幅降低去重存储消耗，只不过也会使得爬虫系统错过少量的页面。

布隆过滤器在 NoSQL 数据库领域使用非常广泛，我们平时用到的 HBase、Cassandra 还有 LevelDB、RocksDB 内部都有布隆过滤器结构，布隆过滤器可以显著降低数据库的 IO 请求数量。当用户来查询某个 row 时，可以先通过内存中的布隆过滤器过滤掉大量不存在的 row 请求，然后再去磁盘进行查询。

邮箱系统的垃圾邮件过滤功能也普遍用到了布隆过滤器，因为用了这个过滤器，所以平时也会遇到某些正常的邮件被放进了垃圾邮件目录中，这个就是误判所致，概率很低。

他其实还有很多用法，我没怎么讲，比如限流，附近的人GeoHash等等，我们公司的附近的人也是用它实现的。



遇到过的坑

sync的时候遇到了bgsave 这个时候cpu飙升，因为bgsave之后会做一个emptyDB这个时候做bgsave cow的机制就没了会重新加载整个RDB 然后就swap了。

还有当写的tps较高，slave在加载rdb文件，slave会被阻塞，无法执行master同步过来的命令，复制积压缓冲区数据被冲掉。

有个参数client-output-buffer-limit slave 256mb 64mb 60（大于256mb断开,或者连续60秒超过64mb断开） 当加载完数据之后，slave向master发送 sync 命令，由于复制积压数据被冲掉了，slave会再次请求同步数据。

如此反复，导致无限同步。

这中间的名词我就不过多解释了，大家应该都是有redis基础的。

总结

以上就是本次的分享内容了，谢谢大家，大家看了之后不知道有没有一些收获，个人觉得很多点大家应该还是有点小收获的，是不是都没有想到一个Redis花样能这么多？

以前都只是get set用到它的hash kv结构就没了，现在甚至能用来做签到什么的了对吧，其实文案很急最近618比较忙，我在只能这样了哈哈哈哈。

学习一门技术就是这样，不要局限于他的表面，你深入了解就发现，后面是个庞大的家族体系，你和大神的差距可能也没那么大，不过技术的时间，只有日积月累这一条路。

成长路上，脚踏实地的代码量没有捷径。

最后再次感谢老钱，他的《Redis深度历险》和黄健宏老师的《Redis设计与实现》这两本书结合来看，你会在Redis上有很深的理解的，真心推荐。

我是敖丙，一个在互联网苟且偷生的工具人。

你知道的越多，你不知道的越多，人才们的 **【三连】** 就是丙丙创作的最大动力，我们下期见！

注：如果本篇博客有任何错误和建议，欢迎人才们留言，**你快说句话啊**！

> 文章持续更新，可以微信搜索「 **敖丙** 」第一时间阅读，回复【**资料**】有我准备的一线大厂面试资料和简历模板，本文 **GitHub** https://github.com/JavaFamily 已经收录，有大厂面试完整考点，欢迎Star。