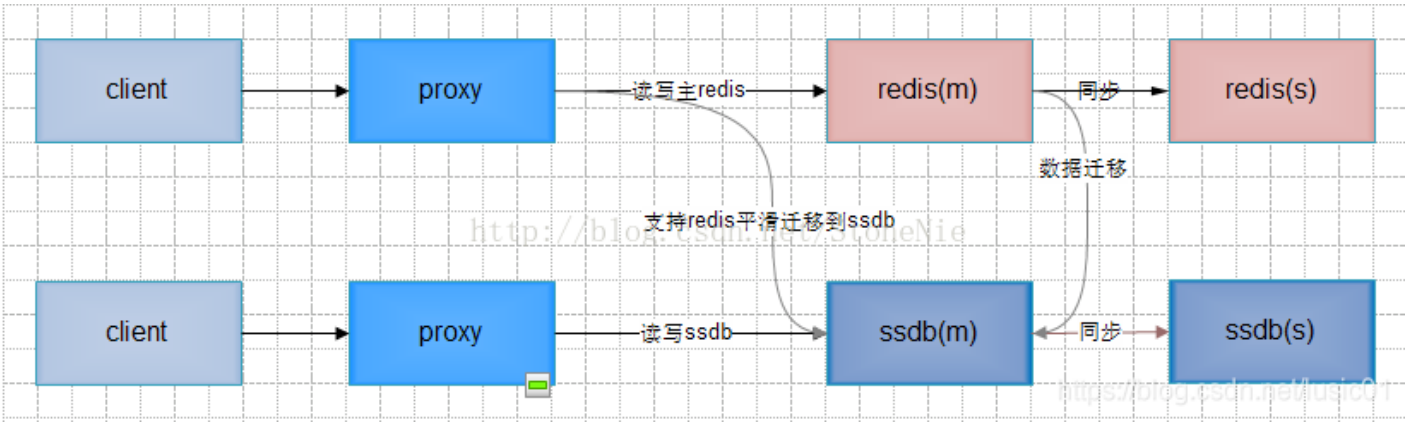
一，当前KV数据库从存储介质可以分为两种模式：

1，一种是以内存为主持久化为辅，如memcache(无持久化)，redis等----------侧重高性能

2，一种是以持久化为主内存为辅，如ssdb(基于leveldb/rocksdb存储引擎)-------侧重大容量

冷热分离方案主要基于redis或者基于redis协议及命令实现

方案一 改造redis，是它支持冷热分离



client----------->proxy-------读写主redis------->redis(m)----同步---->redis(s)

client----------->proxy-----读写ssdb---------->ssdb(m)<----同步------>ssdb(s)

支持redis平滑迁移到ssdb

实现描述：

可以使用开源的rocksdb或imdb引擎读写落地数据

写操作全部记录在内存，不同步写磁盘

常驻写子进程定时将内存中的数据写到磁盘

内存中标记不存在的key,如果一个key在磁盘上不存在，则在标记之后不用再去磁盘查看这个值是否存在

读操作先读内存，如内存中不存在且key未被标识磁盘不存在，则由读子进程从磁盘读并写回到redis（key不存在才写回）。之后子进程通知主进程再次读取，此过程会阻塞主进程上单个连接的处理。

优点：真正意义上实现单机redis的冷热分离。redis和落地数据在同一台机器，容易保证数据一致性。

缺点：基于redis做二次开发，后续不方便升级redis

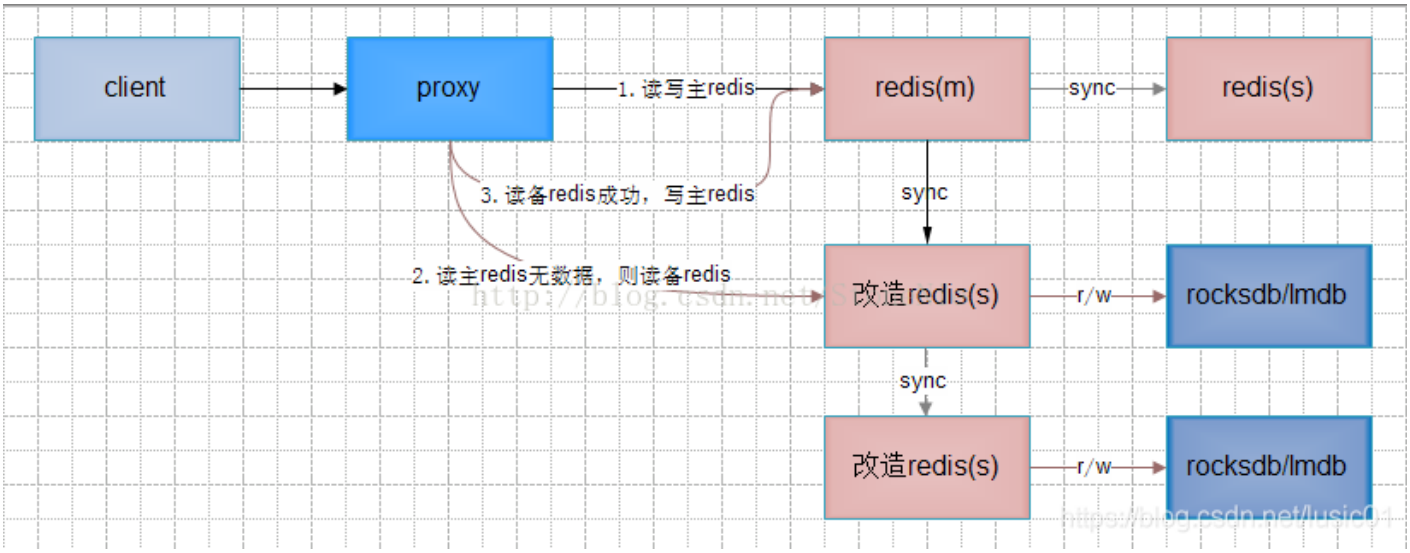
分析：

redis定位是内存KV数据库，只支持所有数据存放在内存，持久化只是数据安全性的一种保障方式。

redis2.0加入支持VM，VM机制即虚拟内存机制，参考操作系统的虚拟内存机制实现，把不经常访问的数据从内存交换到磁盘中。但由于VM机制在某些情况下会导致redis重启，保存和同步数据太慢，2.4版后就不再支持了

Jimdb S是京东基于redis2.8实现的KV数据库，用SSD(固态硬盘)持久化数据。使用jimdb S可以保存全量数据，把缓存+数据库的两层架构用一层架构取代。写操作先写内存，再异步写cycledb。读操作如果数据不在内存，则创建后台任务读cycledb。这个后台任务的作用是预热，读到数据后并不把数据载入内存，执行完成后由前台主线程再次读取，这次在内存中读不到则直接读取cycledb并载入内存。

方案二 改造备redis和proxy,备redis落地数据



client------>proxy-----1,读写主redis------>redis(m)----sync---->redis(s)

(sync)改造redis(s)----r/w--->rocksdb/imdb

(sync)改造redis(s)----r/w--->rocksdb/imdb

2,读主redis无数据，则读备redis　3，读备redis成功，写主redis

实现描述

写操作时proxy正常写主redis，由改造备redis写rocksdb

读操作时proxy先正常读主redis，如无数据，则读改造备redis；改造备redis在内存中读不到数据则读rocksdb,proxy从改造备redis读到数据再写主redis

优点：写操作和当前流程完全一样；读操作和当前迁移流程中rrw流程基本一致，可以复用。不影响纯内存的原生redis使用，风险可控

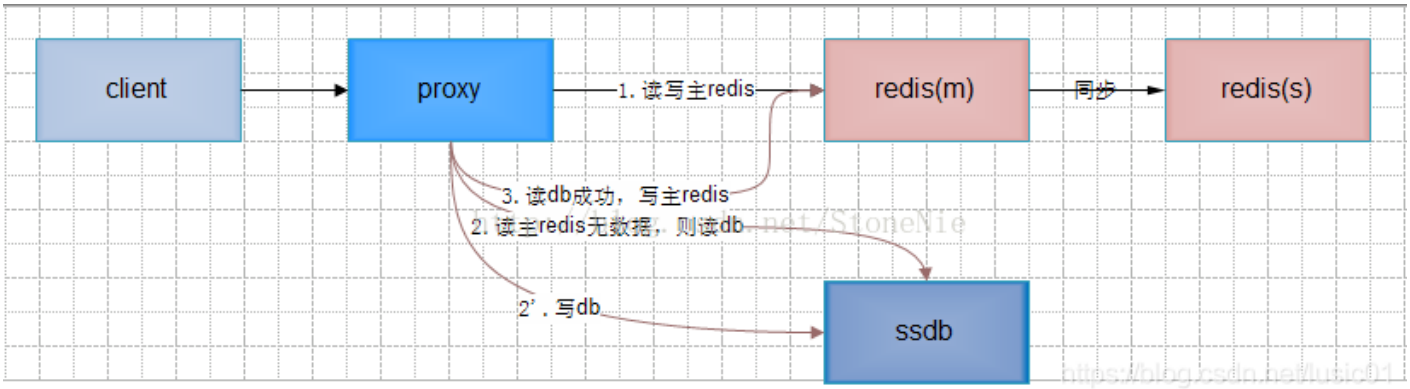
缺点：proxy和redis均需修改。在原有一主一备redis基础上需要增加改造备redis部署。

分析：

最大特点是不影响纯内存的原生redis使用，且proxy改动较小

可以视情况选择部署一个或两个改造备redis。只部署一个改造备redis时落地数据是单点，可用于数据丢失不重要或后端另存有全量数据的场景。部署两个改造备redis可以避免单点，因为是链式同步，对主redis几乎无影响。

方案三 ： 改造proxy，使用ssdb落地数据



实现描述：

写操作时proxy先正常写主redis，再同步或异步写ssdb

读操作时proxy先正常读redis，如redis无数据，则读ssdb;读ssdb成功，再写主redis

优点：只改proxy,redis无须改动

缺点：proxy实现较复杂，redis和ssdb的数据一致性不好保证。因为ssdb基于leveldb/rocksdb实现，在读操作且redis中无数据且ssdb内存中无数据时，可能极大影响性能。

方案四 提供ssdb,业务选择接入redis或ssdb

原

关于redis的冷热数据分离

2017年01月24日 16:02:14 stonenie 阅读数：10428 标签： redis 冷热分离 kv数据库 c 更多

个人分类： 缓存

一、概述

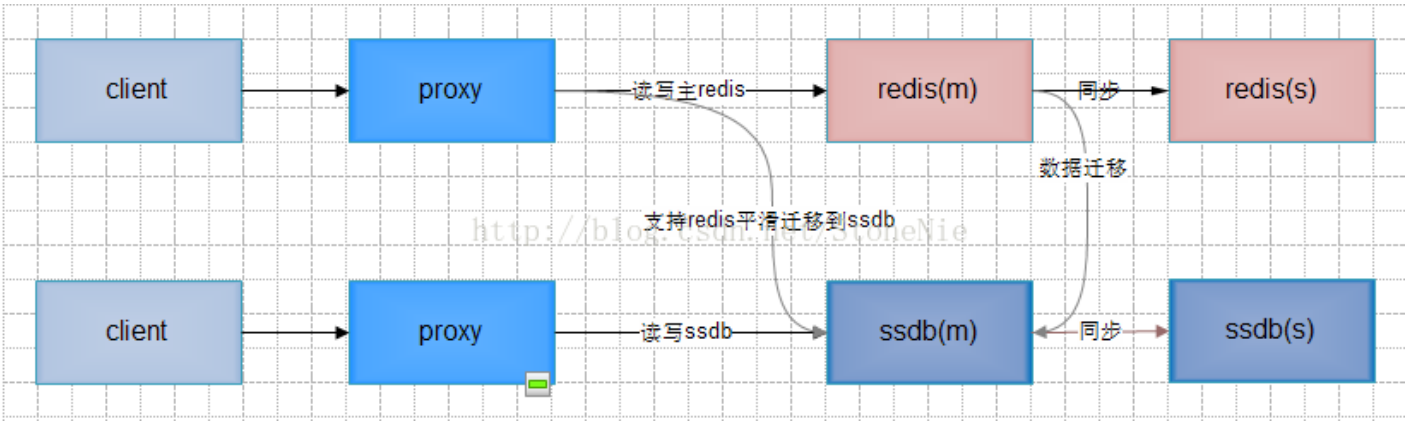
当前KV数据库从存储介质可以分为两种模式，一种是以内存为主持久化为辅，如memcache（无持久化）、redis等；一种是以持久化为主内存为辅，如ssdb（基于leveldb/rocksdb存储引擎）。这两种模式代表了两种不同的选择策略和哲学，适应不同的业务场景。简单地说，以内存为主的模式侧重高性能，信奉“内存是新的硬盘”的哲学；以持久化为主的模式则侧重大容量，兼顾性能。

对于以持久化为主的模式，因其天然支持大容量数据的快速读写，实现冷热数据分离是相对比较容易的，当前用到的数据就认为是热数据，只要把热数据保留于指定大小的内存即可。而对于以内存为主的模式，即使有持久化，也只是顺序写的持久化，在需要读硬盘时不能做到快速读取。因此这种模式要实现冷热分离，需要准确区分冷热数据、精心设计落地策略并保证可以快速读取。

这里冷热分离方案主要基于redis或者基于redis协议及命令实现。

二、方案汇总

2.1 方案一 改造redis，使之支持冷热分离



l 实现描述：

ü 可以使用开源的rocksdb或lmdb引擎读写落地数据；

ü 写操作全部记录在内存，不同步写磁盘；

ü 常驻写子进程定时将内存中的数据写到磁盘；

ü 内存中标记不存在的key，如果一个key在磁盘上不存在，则在标记之后不用再去磁盘查看这个值是否存在；

ü 读操作先读内存，如内存中不存在且key未被标识磁盘不存在，则由读子进程从磁盘读并写回到redis（key不存在才写回）。之后子进程通知主进程再次读取，此过程会阻塞主进程上单个连接的处理。

l 优点：真正意义上实现单机redis的冷热分离。Redis和落地数据在同一台机器，容易保证数据一致性。

l 缺点：实现较复杂。因为是基于redis做二次开发，后续不方便升级redis，不过单机redis已经非常稳定，后续升级可能性较小。

l 分析：

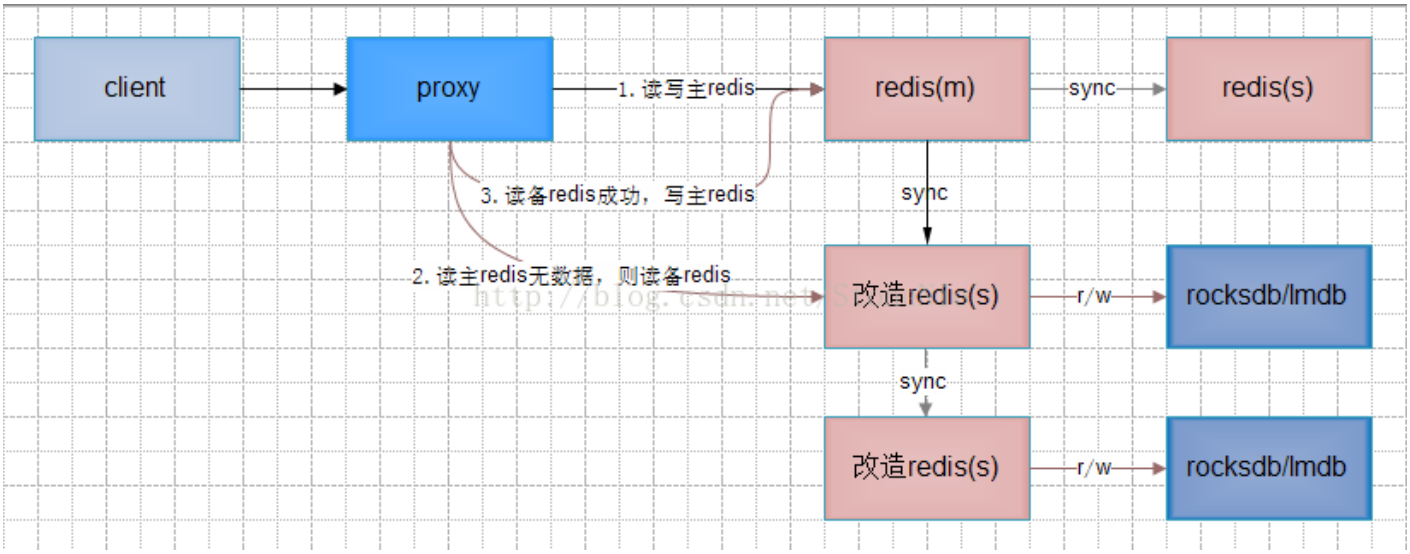
redis定位是内存KV数据库，只支持所有数据存放在内存，持久化只是数据安全性的一种保障方式。基于redis做冷热分离的例子有两个，一个是v2.0-2.4版的原生redis，一个是jimdb S，这两个做得都不成功。

redis 2.0版加入支持VM机制，VM机制即虚拟内存机制，参考操作系统的虚拟内存机制实现，暂时把不经常访问的数据从内存交换到磁盘中，需要时再从磁盘交换回内存，可以实现冷热数据分离。但由于VM机制在某些情况下会导致redis重启、保存和同步数据等太慢及代码复杂，所以2.4版后就不再支持了。

Jimdb S是京东云平台基于redis2.8实现的KV数据库，用SSD持久化数据。使用Jimdb S可以保存全量数据，把缓存+数据库的两层架构用一层架构取代。写操作时先写内存，再异步写cycledb。读操作如数据不在内存，则创建后台任务读cycledb。这个后台任务的作用是预热，读到数据后并不把结果载入内存，执行完成后由前台主线程再次读取，这次在内存中读不到则直接读取cycledb并载入内存。目前了解到的情况是使用不广泛，而且即将下线。主要原因是性能不如纯内存的redis，但不知道是否还有其它缺陷。

从以上redis删除VM机制和jimdb S的实践情况看，直接改造redis做冷热分离可能并不是一个很好的发展方向，即使做出来也很可能是一个平庸的产品。最根本的原因是纯内存的redis性能更好，而用户对性能的期望是没有最好，只有更好。随着内存越来越大、越来越便宜，更多的数据可以直接放到内存，会进一步导致冷热分离成为一个鸡肋功能。另外一个原因是实现冷热分离会导致redis代码复杂性增加不少，不利于后续的维护。

方案二 改造备redis和proxy，备redis落地数据



l 实现描述：

ü 写操作时proxy正常写主redis，由改造备redis写rocksdb；

ü 读操作时proxy先正常读主redis，如无数据，则读改造备redis；改造备redis在内存中读不到数据则读rocksdb，proxy从改造备redis读到数据再写主redis。

l 优点：写操作和当前流程完全一样；读操作和当前迁移流程中rrw流程基本一致，可以复用。不影响纯内存的原生redis使用，风险可控。

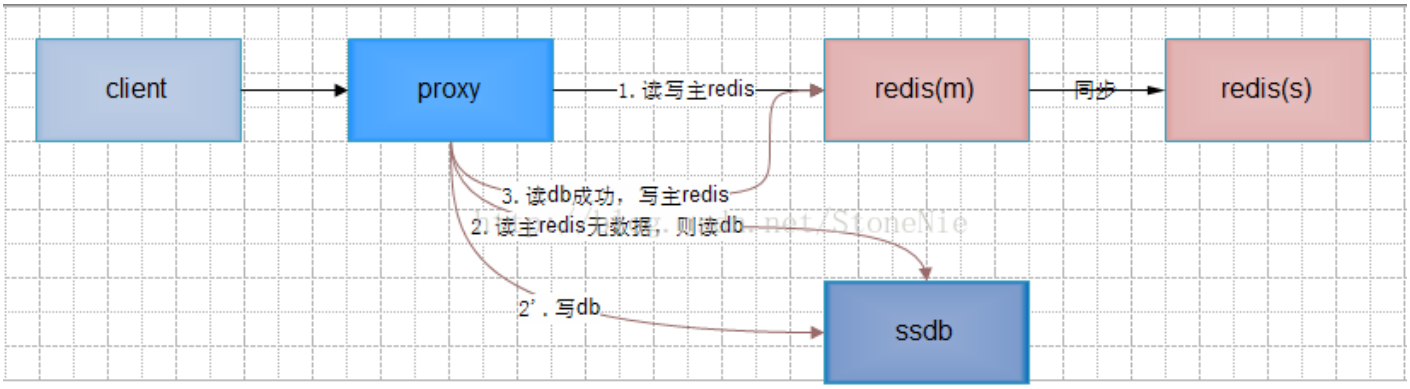
l 缺点：proxy和redis均需修改。在原有一主一备redis基础上需要增加改造备redis部署。

l 分析：

最大特点是不影响纯内存的原生redis使用，且proxy改动较小。

可以视情况选择部署一个或两个改造备redis。只部署一个改造备redis时落地数据是单点，可用于数据丢失不重要或后端另存有全量数据的场景。部署两个改造备redis可以避免单点，因为是链式同步，对主redis几乎无影响。

方案三 改造proxy，使用ssdb落地数据



l 实现描述：

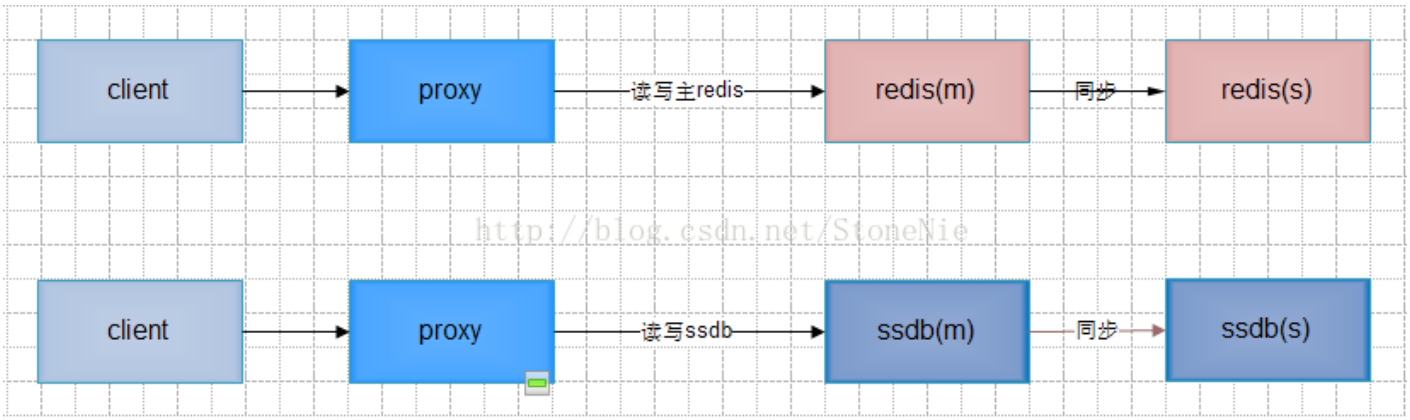
ü 写操作时proxy先正常写主redis，再同步或异步写ssdb；

ü 读操作时proxy先正常读redis，如redis无数据，则读ssdb；读ssdb成功，再写主redis。

l 优点：只改proxy，redis无须改动。

l 缺点：proxy实现较复杂，redis和ssdb的数据一致性不好保证。因为ssdb基于leveldb/rocksdb实现，在读操作且redis中无数据且ssdb内存中无数据时，可能极大影响性能。

方案四 提供ssdb，业务选择接入redis或ssdb

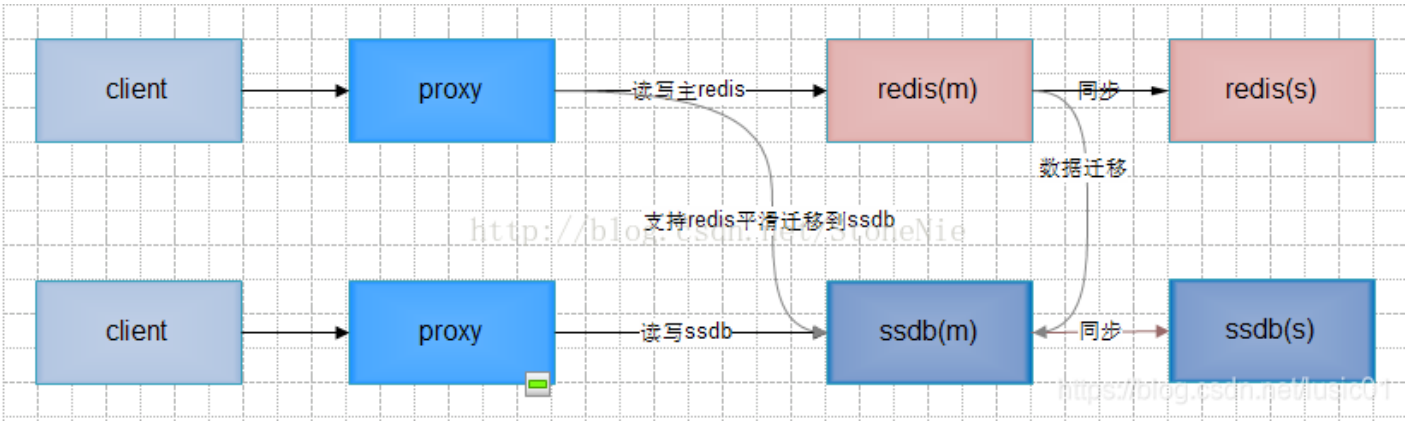


实现描述：redis和ssdb独立两套系统，类似之前腾讯提供CMEM和TSSD两套系统。业务开发根据业务特点决定使用哪一套系统。client，proxy可以复用，等同可以选择使用redis存储引擎或leveldb/rocksdb存储引擎。

优点：无须开发，只需引入ssdb系统即可

缺点：业务开发可能没办法一开始确定使用哪一套系统。需要维护和运维两套系统

方案五 提供ssdb,业务初始化接入redis，可选择平滑迁入ssdb



实现描述：类似方案四，但可选择从redis平滑迁入ssdb

优点：只需开发proxy支持迁入ssdb系统即可

缺点：需要维护和运维两套系统

三：通用问题

1，读操作且redis中无数据的性能问题

不管是直接基于leveldb/rocksdb做数据落地，还是使用ssdb,都会碰到读操作且redis中无数据的性能问题，因此此时需要先读取redis，redis中读不到再一个level一个level去读磁盘文件，这种情况的性能可想而知不会太好

2，redis的淘汰

redis区分冷热数据都是设定redis的maxmemory,然后进行lru淘汰使内存中只保留热数据，而redis的lru淘汰只是从随机选的一些key选出最符合lru规则的一个key进行淘汰，即只是一种近似淘汰，所以不能很好地区分冷热数据。因此有可能出现被lru淘汰的key实际并不是冷数据，这样下次读取时会因为redis中已无数据而去磁盘读，出现一些性能问题。

3，写操作先写内存还是先写磁盘

先写内存，此时如果系统奔溃，内存中的数据还没来得及dump到磁盘，会丢失数据。先写磁盘，再写内存，则即使系统崩溃，不会造成数据的丢失，但可能导致磁盘和内存数据的不一致，为了避免这种不一致，又得先删除内存中的key，再写磁盘，再写内存，影响性能。总的来说，对于以内存为主的KV数据库，优先选择先写内存。

SSD应用风险

为了提高读写磁盘的性能，需要使用SSD。而SSD本身存在一些问题：

毛刺问题：同时读写SSD盘时，读SSD盘有可能会耗时数秒。被挂住的几率为万分之一；

坏盘问题：SSD坏盘几率比普通sas硬盘要高

坏块问题：SSD盘中可能存在某个块可以写入，但是读不出来，此时这个块的数据将会丢失

（

毛刺问题：

最近收到一封邮件，对于线上磁盘使用出现毛刺的分析。

话说某天，磁盘使用的监控出现多个毛刺，每秒读的大小为20M/S,造成的影响就是响应时间慢，用户体验不好。于是开始了分析的过程，通过查看毛刺出现点的web server日志，发现这个时候多是在处理查询的请求，于是开始进一步探索这个时候的查询都分别是对哪些表，什么动作行为的查询？查看得知，此时最多的查询请求是对我们item表--记录当前资源信息的表的查询，于是，想到，对表的查询，速度慢，原因是啥呢？索引？数据量大？分区，分表不合理？于是，对该表的信息进行查询，发现该表当前存在13W条左右的数据，占据的磁盘大小为4G+，相当于每一条数据占据的大小为32K，那么做一次全表扫描（全表扫描是数据库服务器用来搜寻表的每一条记录的过程，直到所有符合给定条件的记录返回为止。），，，，，，，，这个动作造成的磁盘的IO的确很惊人。通过对DB此时查询执行的sql语句进行分析，进行查询的条件多没建索引，于是尝试建立索引（考虑了该类索引对应的字段内容是不是会经常进行修改，字段类型是不是适合建立索引等）。这些动作做完之后，回归该问题，发现比之前有所改进，但是毛刺还是不理想，于是开始分析，发现我们的环境里面还存在一个情况，就是虽然我们做了上面的操作：合理的建立了索引、将查询的数据分批次执行（不将13w的数据一次性全表扫描），但是环境里面出现多次对item表操作的动作，为什么呢？还是查看日志，发现我们环境里有一台机器不知道为什么，每半分钟就从item表里面拉5次数据，之前这个机器是用来做缓存的，可能当时没有考虑到这样频繁拉到带来的影响，现在已经有了新的替代方案，做了nginx的代理缓存和web缓存，这一层对于db数据的缓存也用了相应的技术解决，于是申请停掉该机器的服务，再次回归，问题回归合理值。读从20M/s回归到0.2M/S。

）

五、附录

附各C/C++持久化KV数据库简介和分析：

l Leveldb

LevelDb是google开源的能够处理十亿级别规模KV型数据持久性存储的C++程序库。LevleDb在存储数据时，是根据记录的key值有序存储。LevelDb的写操作要大大快于读操作，而顺序读写操作则大大快于随机读写操作。

level一个写操作仅涉及一次磁盘文件追加写和内存SkipList插入操作，因此leveldb的写操作非常高效。

由于 LevelDB 在某一层查找不存在的数据时, 会继续在下一层进行查找, 所以对于不存在的数据的查找会速度非常慢. 所以, 需要结合 Bloom Filter, 利用 Bloom Filter 能快速地判定”不存在”的特点。

l Rocksdb

Facebook 维护的一个活跃的 LevelDB 的分支。RocksDB 在 LevelDB 上做了很多的改进，比如多线程 Compactor、分层自定义压缩、多 MemTable 等。另外 RocksDB 对外暴露了很多配置项，可以根据不同业务的形态进行调优。

l Lmdb

LMDB 选择在内存映像文件 (mmap) 实现 B+Tree，而且同时使用了 Copy-On-Write 实现了 MVCC 实现并发事务无锁读的能力，对于高并发读的场景比较友好；同时因为使用的是 mmap 所以拥有跨进程读取的能力。

l Ssdb

基于leveldb/rocksdb存储引擎实现，加入网络支持，兼容redis协议和redis数据类型（不支持set集合），支持主从复制和负载均衡。SSDB/LevelDB 在进行数据库整理(Compaction)操作时, 磁盘io高，持续的时间一般随着数据变大而变长, 一般只持续数秒。不能指定执行compaction的时间。有些redis命令不支持，有些支持的redis命令可能不完全和redis一致。

l Mangodb

分布式文档数据库。MongoDB的最大卖点是不需构建非主键索引也能执行很多查询。

l Fatcache

SSD上实现的memcached，内存中保存索引数据，机器重启索引数据会丢失。假如只需要支持string类型数据落地，可使用代替ssdb。

l Tair

淘宝开源的分布式KV数据库。抽象存储层的架构设计使Tair很容易接入新的存储引擎，当前支持的存储引擎有非持久化的MDB(自主研发的类memcache）/RDB（抽离Redis的存储部分），持久化的LDB（接入LevelDB）。

l TTC

非严格意义上的KV数据库。支持无数据源和持久数据源两种工作模式。无数据源模式就是一个简单的基于共享内存的cache服务。持久数据源后接mysql，写操作先写db，再写内存；读操作先读内存，内存中不存在再去读db，读db成功再添加到内存。所以TTC本质上是一个带缓存功能的mysql数

据库，可以想见其读写性能肯定不如纯内存数据库，也没有leveldb高效写的特点，注定是一个平庸的产品，再加上使用繁琐，早已被淘汰。

l CMEM

分布式KV内存数据库，支持主从同步和平滑迁移，数据可落地，高性能。仅做KV缓存还不错，但支持数据类型和命令不如redis丰富。

===

===

==

Redis是单线程，但redis为什么这么快

缓存雪崩，缓存穿透，缓存预热，缓存更新，缓存降级

((

一：缓存雪崩：如果缓存集中在一段时间内失效，发生大量的缓存穿透，所有的查询都落在数据库上，造成缓存雪崩

（大多数系统设计者考虑用加锁或者队列的方式保证缓存的单线程（进程）写，从而避免失效时大量的并发请求落到底层存储系统上。）

解决方法：

A，加锁排队。限流----限流算法 1.计数 2.滑动窗口 3令牌桶Token Bucket 4漏桶leaky bucket

在缓存失效后，通过加锁 或者队列来控制读数据库写缓存的线程数量。比如对某个key只允许一个线程查询数据和写数据，其他线程等待。

业界比较常用的做法，是使用mutex。简单来说，就是在缓存失效的时候（判断拿出来的值为空），不是立即去load db,而是先使用缓存工具的某些带成功操作返回值的操作（比如redis的SETNX（set if not exists）或者Memcache的ADD）去set一个mutex key，当操作返回成功时，再进行load db的操作并回设缓存；否则，就重试整个get缓存的方法。

B：数据预热：可以通过缓存reload机制（清空缓存），预先去更新缓存，再即使发生大并发访问前手动触发加载缓存不同的key，设置不同的过期时间，让缓存失效的时间点尽量均匀

C：做二级缓存，或者双缓存策略

A1为原始缓存，A2为拷贝缓存，A1失效时，可以访问A2，A1缓存失效时间设置为短期，A2设置为长期。

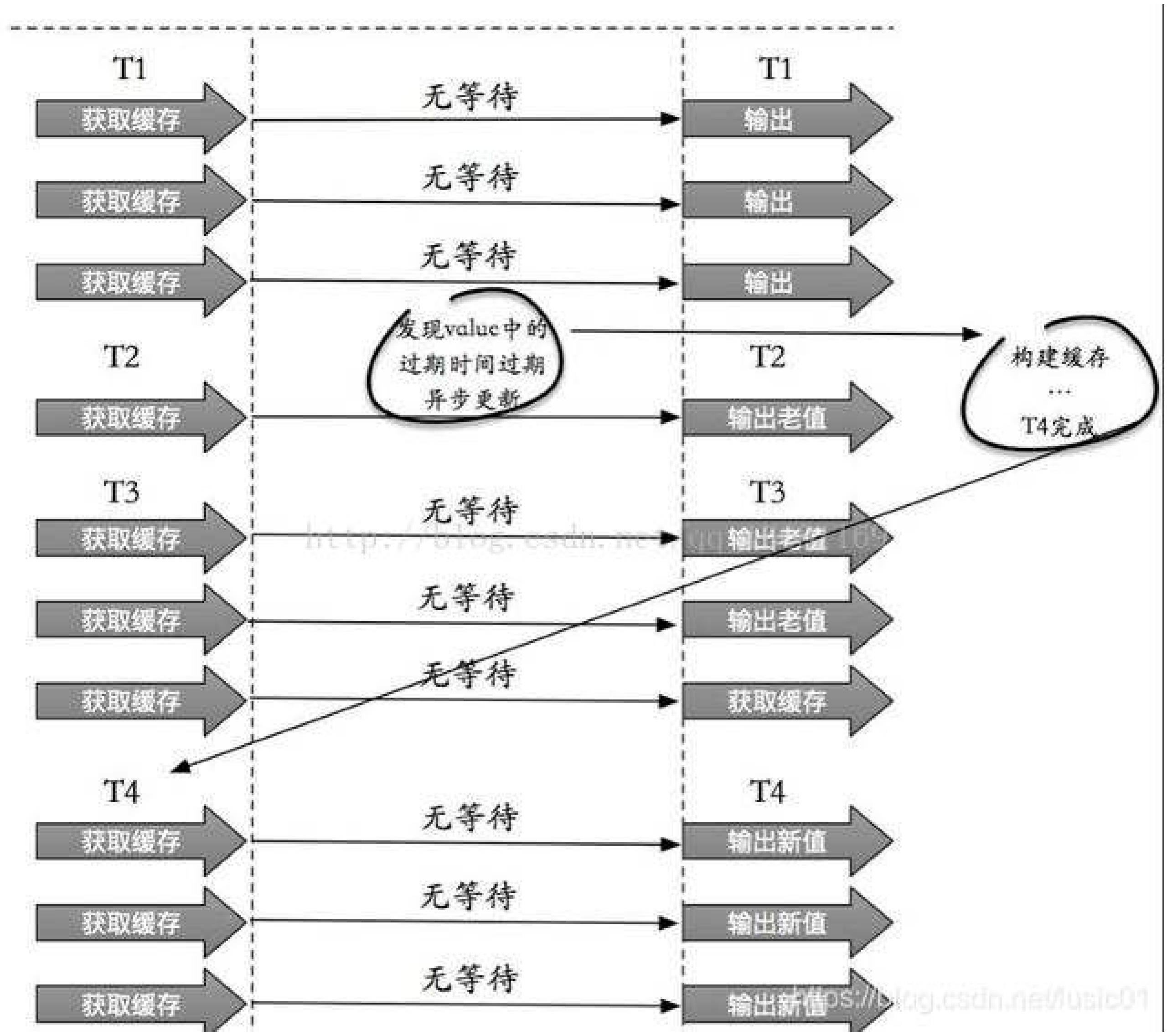
D：缓存永远不过期：

这里的“”永远不过期“”包含两层意思：

1）从缓存上看，确实没有设置过期时间，这就保证了，不会出现热点key过期问题，也就是“”物理“”不过期

2）从功能上看，如果不过期，那不就成静态的了吗？所以我们把过期时间存在key对应的value里，如果发现要过期了，通过一个后台的异步线程进行缓存的构建，也就是“逻辑”过期.

从实战看，这种方法对于性能非常友好，唯一不足的就是构建缓存时候，其余线程（非构建缓存的线程）可能访问的是老数据，但是对于一般的互联网功能来说这个还是可以忍受的。



二，缓存穿透：是指查询一个一定不存在的数据，由于缓存是不命中时需要从数据库查询，查不到数据则不写入缓存，这将导致这个不存在的数据每次请求都要到数据库去查询，造成缓存穿透。

)))00

redis是一个开源的内存中的数据结构存储系统，它可以用作：数据库、缓存和消息中间件

它支持多种类型的数据结构，如字符串（String），散列（Hash），列表（List），集合（Set），有序集合（Sorted Set或者是ZSet）与范围查询，Bitmaps，Hyperloglogs 和地理空间（Geospatial）索引半径查询。其中常见的数据结构类型有：String、List、Set、Hash、ZSet这5种。

(((

原文链接：
http://rainybowe.com/blog/2017/07/13/%E7%A5%9E%E5%A5%87%E7%9A%84HyperLogLog%E7%AE%97%E6%B3%95/index.html?utm_source=tuicool&utm_medium=referral

神奇的HyperLogLog算法

基数计数基本概念

基数计数(cardinality counting)通常用来统计一个集合中不重复的元素个数，例如统计某个网站的UV(**unique visitor** 网站独立访客-------**独立IP：**是指独立用户/独立访客。----------**PV(访问量)：**即Page View, 即页面浏览量或点击量)，或者用户搜索网站的关键词数量。数据分析、网络监控及数据库优化等领域都会涉及到基数计数的需求。 要实现基数计数，最简单的做法是记录集合中所有不重复的元素集合S_uSu，当新来一个元素x_ixi，若S_uSu中不包含元素x_ixi，则将x_ixi加入S_uSu，否则不加入，计数值就是S_uSu的元素数量。这种做法存在两个问题：

1. 当统计的数据量变大时，相应的存储内存也会线性增长
2. 当集合S_uSu变大，判断其是否包含新加入元素x_ixi的成本变大

大数据量背景下，要实现基数计数，首先需要确定存储统计数据的方案，以及如何根据存储的数据计算基数值；另外还有一些场景下需要融合多个独立统计的基数值，例如对一个网站分别统计了三天的UV，现在需要知道这三天的UV总量是多少，怎么融合多个统计值。

基数计数方法

B树

B树最大的优势是插入和查找效率很高，如果用B树存储要统计的数据，可以快速判断新来的数据是否已经存在，并快速将元素插入B树。要计算基数值，只需要计算B树的节点个数。 将B树结构维护到内存中，可以快速统计和计算，但依然存在问题，B树结构只是加快了查找和插入效率，并没有节省存储内存。例如要同时统计几万个链接的UV，每个链接的访问量都很大，如果把这些数据都维护到内存中，实在是够呛。

bitmap

bitmap可以理解为通过一个bit数组来存储特定数据的一种数据结构，每一个bit位都能独立包含信息，bit是数据的最小存储单位，因此能大量节省空间，也可以将整个bit数据一次性load到内存计算。 如果定义一个很大的bit数组，基数统计中每一个元素对应到bit数组的其中一位，例如bit数组 0011010010011101001代表实际数组[2,3,5,8][2,3,5,8]。新加入一个元素，只需要将已有的bit数组和新加入的数字做按位或 (or)(or)计算。bitmap中1的数量就是集合的基数值。

bitmap有一个很明显的优势是可以轻松合并多个统计结果，只需要对多个结果求异或就可以。也可以大大减少存储内存，可以做个简单的计算，如果要统计1亿个数据的基数值，大约需要内存：
100000000/8/1024/1024 \approx≈ 12M
如果用32bit的int代表每个统计数据，大约需要内存：
32*100000000/8/1024/1024 \approx≈ 381M

bitmap对于内存的节约量是显而易见的，但还是不够。统计一个对象的基数值需要12M，如果统计10000个对象，就需要将近120G了，同样不能广泛用于大数据场景。

概率算法

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基数值，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，实际上LC的空间复杂度与上文中简单bitmap方法是一样的（但是有个常数项级别的降低），都是O(N_{max})O(Nmax)；
- LogLog Counting(LLC)：LogLog Counting相比于LC更加节省内存，空间复杂度只有O(log_2(log_2(N_{max})))O(log2(log2(Nmax)))
- 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。

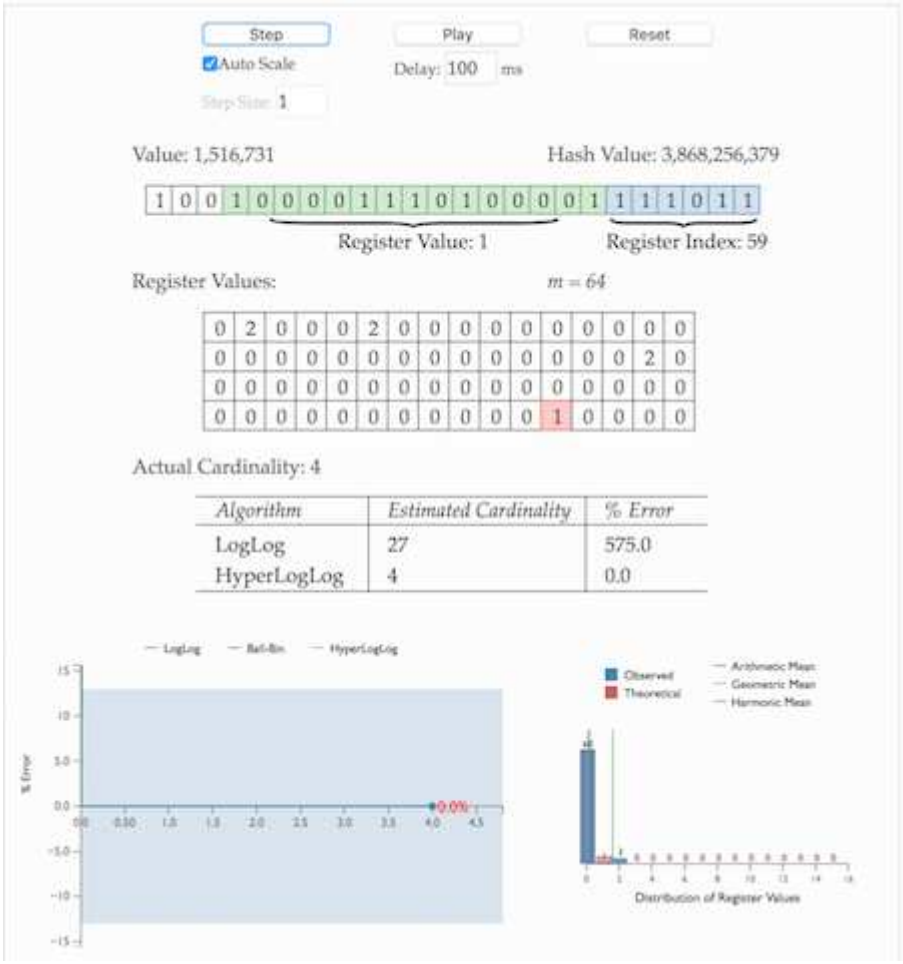
下面将着重讲HLL的原理和计算过程。

HyperLogLog的惊人表现

上面我们计算过用bitmap存储1一亿个统计数据大概需要12M内存；而在HLL中，只需要不到1K内存就能做到；redis中实现的HyperLogLog，只需要12K内存，在标准误差0.81%的前提下，能够统计2^{64}264个数据。首先容我感叹一下数学的强大和魅力，那么概率算法是怎样做到如此节省内存的，又是怎样控制误差的呢?

首先简单展示一下HLL的基本做法，HLL中实际存储的是一个长度为mm的大数组SS，将待统计的数据集合划分成mm组，每组根据算法记录一个统计值存入数组中。数组的大小mm由算法实现方自己确定，redis中这个数组的大小是16834，mm越大，基数统计的误差越小，但需要的内存空间也越大。

这里有个HLL demo可以看一下HLL到底是怎么做到这种超乎想象的事情的。



1. 通过hash函数计算输入值对应的比特串
2. 比特串的低 t(t=log_2^m)t(t=log2m)位对应的数字用来找到数组SS中对应的位置 ii
3. t+1t+1位开始找到第一个1出现的位置 kk，将 kk 记入数组S_iSi位置
4. 基于数组SS记录的所有数据的统计值，计算整体的基数值，计算公式可以简单表示为：\hat{n}=f(S)n^=f(S)

看到这里心里应该有无数个问号，这样真的就能统计到上亿条数据的基数了吗？我总结一下，先抛出三个疑问：

1. 为什么要记录第一个1出现的位置?
2. 为什么要有分桶数组 SS ？
3. 通过分桶数组 SS 计算基数的公式是什么?

hyperloglog原理理解

举一个我们最熟悉的抛硬币例子，出现正反面的概率都是1/2，一直抛硬币直到出现正面，记录下投掷次数kk，将这种抛硬币多次直到出现正面的过程记为一次伯努利过程，对于nn次伯努利过程，我们会得到nn个出现正面的投掷次数值k_1k1,k_2k2……k_nkn，其中最大值记为k_{max}kmax，那么可以得到下面结论：

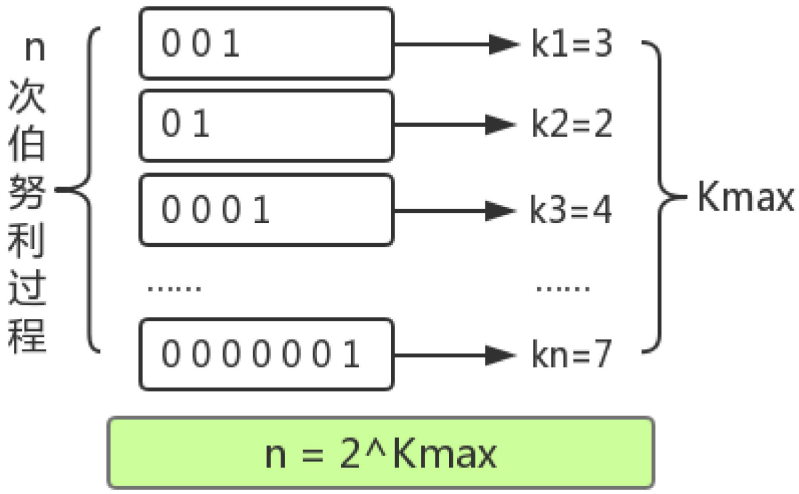
1. nn次伯努利过程的投掷次数都不大于k_{max}kmax
2. nn次伯努利过程，至少有一次投掷次数等于k_{max}kmax

对于第一个结论，nn次伯努利过程的抛掷次数都不大于k_{max}kmax的概率用数学公式表示为：
P_n(X \le k_{max})=(1-1/2^{k_{max}})^nPn(X≤kmax)=(1−1/2kmax)n

第二个结论至少有一次等于k_{max}kmax的概率用数学公式表示为：
P_n(X \ge k_{max})=1-(1-1/2^{k_{max}-1})^nPn(X≥kmax)=1−(1−1/2kmax−1)n

当n\ll 2^{k_{max}}n≪2kmax时，P_n(X \ge k_{max})\approx0Pn(X≥kmax)≈0，即当nn远小于2^{k_{max}}2kmax时，上述第一条结论不成立；
当n\gg 2^{k_{max}}n≫2kmax时，P_n(X \le k_{max})\approx0Pn(X≤kmax)≈0，即当nn远大于2^{k_{max}}2kmax时，上述第二条结论不成立。 因此，我们似乎就可以用2^{k_{max}}2kmax的值来估计nn的大小。

以上结论可以总结为：进行了nn次进行抛硬币实验，每次分别记录下第一次抛到正面的抛掷次数kk，那么可以用n次实验中最大的抛掷次数k_{max}kmax来预估实验组数量nn： \hat{n} =



2^{k_{max}}n^=2kmax

可以通过一组小实验验证一下这种估计方法是否基本合理。

回到基数统计的问题，我们需要统计一组数据中不重复元素的个数，集合中每个元素的经过hash函数后可以表示成0和1构成的二进制数串，一个二进制串可以类比为一次抛硬币实验，1是抛到正面，0是反面。二进制串中从低位开始第一个1出现的位置可以理解为抛硬币试验中第一次出现正面的抛掷次数kk，那么基于上面的结论，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，同样可以可以通过第一个1出现位置的最大值k_{max}kmax来预估总共有多少个不同的数字（整体基数）。

这种通过局部信息预估整体数据流特性的方法似乎有些超出我们的基本认知，需要用概率和统计的方法才能推导和验证这种关联关系。HyperLogLog核心在于观察集合中每个数字对应的比特串，通过统计和记录比特串中最大的出现1的位置来估计集合整体的基数，可以大大减少内存耗费。

现在回到第二节中关于HyperLogLog的第一个疑问，为什么要统计hash值中第一个1出现的位置？第一个1出现的位置可以类比为抛硬币实验中第一次抛到正面的抛掷次数，根据抛硬币实验的结论，

记录每个数据的第一个出现的位置kk，就可以通过其中最大值{k_{max}}kmax推导出数据集合的基数：\hat{n} = 2^{k_{max}}n^=2kmax。

hyperloglog算法讲解

分桶平均

HLL的基本思想是利用集合中数字的比特串第一个1出现位置的最大值来预估整体基数，但是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，运气较好，第一次实验过程就抛了10次才第一次抛到正面，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，同时运气这么好的概率就很低了，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mm个桶，每个桶分别统计各自的{k_{max}}kmax并能得到各自的基数预估值 \hat{n}n^ ，最终对这些 \hat{n}n^ 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，调和平均数的优点是可以过滤掉不健康的统计值，具体的计算公式为：

回到第二节中关于HLL的第二个疑问，为什么要有分桶数组 ？分桶数组是为了消减因偶然性带来的误差，提高预估的准确性。那么分桶数组的大小怎么确定呢?
这是由算法实现方自己设定的，例如上面HLL demo中，设定统计数组的大小，如果函数得到的比特串是32位，需要其中6()位定位分桶数组中的桶的位置，还剩下26位（需要记录的出现1的位置的最大值是26），那么数组中每个桶需要5()位记录1第一次出现的位置，整个统计数组需要花费的内存为：

也就是用32bit的内存能够统计的基数数量为。

偏差修正

上述经过分桶平均后的估计量看似已经很不错了，不过通过数学分析可以知道这并不是基数n的无偏估计。因此需要修正成无偏估计。这部分的具体数学分析在“Loglog Counting of Large Cardinalities”中。

其中系数由统计数组的大小 决定，具体的公式为：

根据论文中分析结论，HLL与LLC一样是渐进无偏估计，渐进标准误差表示为：

因此，统计数组大小 越大，基数统计的标准误差越小，但需要的存储空间也越大，在 的情况下，HLL的标准误差为1.1%。

虽然调和平均数能够适当修正算法误差，但作者给出一种分阶段修正算法。当HLL算法开始统计数据时，统计数组中大部分位置都是空数据，并且需要一段时间才能填满数组，这种阶段引入一种小

范围修正方法；当HLL算法中统计数组已满的时候，需要统计的数据基数很大，这时候hash空间会出现很多碰撞情况，这种阶段引入一种大范围修正方法。最终算法用伪代码可以表示为如下。

```
m = 2^b # with b in [4...16]

if m == 16:
alpha = 0.673
elif m == 32:
alpha = 0.697
elif m == 64:
alpha = 0.709
else:
alpha = 0.7213/(1 + 1.079/m)

registers = [0]*m # initialize m registers to 0

##############################################################################
##
# Construct the HLL structure
for h in hashed(data):
register_index = 1 + get_register_index( h,b ) # binary address of the
rightmost b bits
run_length = run_of_zeros( h,b ) # length of the run of zeroes starting
at bit b+1
registers[ register_index ] = max( registers[ register_index ],
run_length )

```

```
21. ################################################################################
    #
22. # Determine the cardinality
23. DV_est = alpha * m^2 * 1/sum( 2^ -register ) # the DV estimate
24.
25. if DV_est < 5/2 * m: # small range correction
26. V = count_of_zero_registers( registers ) # the number of registers equal
    to zero
27. if V == 0: # if none of the registers are empty, use the HLL estimate
28. DV = DV_est
29. else:
30. DV = m * log(m/V) # i.e. balls and bins correction
31.
32. if DV_est <= ( 1/30 * 2^32 ): # intermediate range, no correction
33. DV = DV_est
34. if DV_est > ( 1/30 * 2^32 ): # large range correction
35. DV = -2^32 * log( 1 - DV_est/2^32)
```

redis中hyperloglog实现

redis正是基于以上的HLL算法实现的HyperLogLog结构，用于统计一组数据集合中不重复的数据个数。 redis中统计数组大小设置为，hash函数生成64位bit数组，其中 位用来找到统计数组的位置，剩下50位用来记录第一个1出现的位置，最大位置为50，需要 位记录。

那么统计数组需要的最大内存大小为： 基数估计的标准误差为。可以学习一下redis中HyperLogLog的源码实现。

参考阅读

Redis new data structure: the HyperLogLog

HyperLogLog — Cornerstone of a Big Data Infrastructure

解读Cardinality Estimation算法（第四部分：HyperLogLog Counting及Adaptive Counting）

redis内置了复制，LUA脚本，LRU驱动事件(LRU eviction)，事务(Transaction)和不同级别的磁盘持久化，并通过redis哨兵（sentinel）和自动分区（cluster）提供高可用性（high availablity）

redis也提供了持久化的选项，这些选项可以让用户将自己的数据保存到磁盘上面进行存储。根据实际情况，可以每隔一定时间将数据集导出到磁盘（快照），或者追加到命令日志中（AOF只追加文件），

Redis提供的持久化机制（RDB和AOF）

Redis提供的持久化机制

Redis是一种面向“key-value”类型数据的分布式NoSQL数据库系统，具有高性能、持久存储、适应高并发应用场景等优势。它虽然起步较晚，但发展却十分迅速。

近日，Redis的作者在博客中写到，**他看到的所有针对Redis的讨论中，对Redis持久化的误解是最大的**，于是他写了一篇长文来对Redis的持久化进行了系统性的论述。

文章主要包含三个方面：**Redis持久化是如何工作的、这一性能是否可靠以及和其它类型的数据库比较**。以下为文章内容：

一、Redis持久化是如何工作的?

什么是持久化？简单来讲就是将数据放到断电后数据不会丢失的设备中，也就是我们通常理解的硬盘上。

首先我们来看一下数据库在进行写操作时到底做了哪些事，主要有下面五个过程：

- 客户端向服务端发送写操作（数据在客户端的内存中）。
- 数据库服务端接收到写请求的数据（数据在服务端的内存中）。
- 服务端调用write这个系统调用，将数据往磁盘上写（数据在系统内存的缓冲区中）。
- 操作系统将缓冲区中的数据转移到磁盘控制器上（数据在磁盘缓存中）。
- 磁盘控制器将数据写到磁盘的物理介质中（数据真正落到磁盘上）。

故障分析

写操作大致有上面5个流程，**下面我们结合上面的5个流程看一下各种级别的故障**：

- 当数据库系统故障时，这时候系统内核还是完好的。那么此时只要我们执行完了第3步，那么数据就是安全的，因为后续操作系统会来完成后面几步，保证数据最终会落到磁盘上。
- 当系统断电时，这时候上面5项中提到的所有缓存都会失效，并且数据库和操作系统都会停止工作。**所以只有当数据在完成第5步后，才能保证在断电后数据不丢失。**

通过上面5步的了解，可能我们会希望搞清下面一些问题：

- 数据库多长时间调用一次write，将数据写到内核缓冲区？
- 内核多长时间会将系统缓冲区中的数据写到磁盘控制器？
- 磁盘控制器又在什么时候把缓存中的数据写到物理介质上？

对于第一个问题，通常数据库层面会进行全面控制。

而对第二个问题，操作系统有其默认的策略，但是我们也可以通过POSIX API提供的fsync系列命令强制操作系统将数据从内核区写到磁盘控制器上。

对于第三个问题，好像数据库已经无法触及，但实际上，大多数情况下磁盘缓存是被设置关闭的，或者是只开启为读缓存，也就是说写操作不会进行缓存，直接写到磁盘。

建议的做法是仅仅当你的磁盘设备有备用电池时才开启写缓存。

数据损坏

所谓数据损坏，就是数据无法恢复，上面我们讲的都是如何保证数据是确实写到磁盘上去，但是写到磁盘上可能并不意味着数据不会损坏。比如我们可能一次写请求会进行两次不同的写操作，当意外发生时，可能会导致一次写操作安全完成，但是另一次还没有进行。如果数据库的数据文件结构组织不合理，可能就会导致数据完全不能恢复的状况出现。

这里通常也有三种策略来组织数据，以防止数据文件损坏到无法恢复的情况：

-

- 第一种是最粗糙的处理，就是不通过数据的组织形式保证数据的可恢复性。而是通过配置数据同步备份的方式，在数据文件损坏后通过数据备份来进行恢复。实际上MongoDB在不开启操作日志，通过配置Replica Sets时就是这种情况。
- 另一种是在上面基础上添加一个操作日志，每次操作时记一下操作的行为，这样我们可以通过操作日志来进行数据恢复。因为操作日志是顺序追加的方式写的，所以不会出现操作日志也无法恢复的情况。这也类似于MongoDB开启了操作日志的情况。
- 更保险的做法是数据库不进行旧数据的修改，只是以追加方式去完成写操作，这样数据本身就是一份日志，这样就永远不会出现数据无法恢复的情况了。实际上CouchDB就是此做法的优秀范例。

二 、Redis提供了RDB持久化和AOF持久化

RDB机制的优势和略施

RDB持久化是指在指定的时间间隔内将内存中的数据集快照写入磁盘。

也是默认的持久化方式，这种方式是就是将内存中数据以快照的方式写入到二进制文件中,默认的文件名为dump.rdb。

可以通过配置设置自动做快照持久化的方式。我们可以配置redis在n秒内如果超过m个key被修改就自动做快照，下面是默认的快照保存配置

```
save 900 1      #900秒内如果超过1个key被修改，则发起快照保存
save 300 10     #300秒内容如超过10个key被修改，则发起快照保存
save 60 10000
```

RDB文件保存过程

- redis调用fork,现在有了子进程和父进程。
- 父进程继续处理client请求，子进程负责将内存内容写入到临时文件。由于os的写时复制机制（copy on write)父子进程会共享相同的物理页面，当父进程处理写请求时os会为父进程要修改的页面创建副本，而不是写共享的页面。所以子进程的地址空间内的数 据是fork时刻整个数据库的一个快照。
- 当子进程将快照写入临时文件完毕后，用临时文件替换原来的快照文件，然后子进程退出。

client 也可以使用save或者bgsave命令通知redis做一次快照持久化。save操作是在主线程中保存快照的，由于redis是用一个主线程来处理所有 client的请求，这种方式会阻塞所有client请求。所以不推荐使用。

另一点需要注意的是，每次快照持久化都是将内存数据完整写入到磁盘一次，并不 是增量的只同步脏数据。如果数据量大的话，而且写操作比较多，必然会引起大量的磁盘io操作，可能会严重影响性

能。

优势

- 一旦采用该方式，那么你的整个Redis数据库将只包含一个文件，这样非常方便进行备份。比如你可能打算没1天归档一些数据。
- 方便备份，我们可以很容易的将一个一个RDB文件移动到其他的存储介质上
- RDB 在恢复大数据集时的速度比 AOF 的恢复速度要快。
- RDB 可以最大化 Redis 的性能：父进程在保存 RDB 文件时唯一要做的就是 fork 出一个子进程，然后这个子进程就会处理接下来的所有保存工作，父进程无须执行任何磁盘 I/O 操作。

劣势

- 如果你需要尽量避免在服务器故障时丢失数据，那么 RDB 不适合你。 虽然 Redis 允许你设置不同的保存点（save point）来控制保存 RDB 文件的频率， 但是， 因为RDB 文件需要保存整个数据集的状态， 所以它并不是一个轻松的操作。 因此你可能会至少 5 分钟才保存一次 RDB 文件。 在这种情况下， 一旦发生故障停机， 你就可能会丢失好几分钟的数据。
- 每次保存 RDB 的时候，Redis 都要 fork() 出一个子进程，并由子进程来进行实际的持久化工作。 在数据集比较庞大时， fork() 可能会非常耗时，造成服务器在某某毫秒内停止处理客户端； 如果数据集非常巨大，并且 CPU 时间非常紧张的话，那么这种停止时间甚至可能会长达整整一秒。 虽然 AOF 重写也需要进行 fork() ，但无论 AOF 重写的执行间隔有多长，数据的耐久性都不会有任何损失。

AOF文件保存过程

redis会将每一个收到的写命令都通过write函数追加到文件中(默认是 appendonly.aof)。

当redis重启时会通过重新执行文件中保存的写命令来在内存中重建整个数据库的内容。当然由于os会在内核中缓存 write做的修改，所以可能不是立即写到磁盘上。这样aof方式的持久化也还是有可能会丢失部分修改。不过我们可以通过配置文件告诉redis我们想要 通过fsync函数强制os写入到磁盘的时机。有三种方式如下（默认是：每秒fsync一次）

```
appendonly yes               //启用aof持久化方式

# appendfsync always       //每次收到写命令就立即强制写入磁盘，最慢的，但是保证完全的持久化，

appendfsync everysec      //每秒钟强制写入磁盘一次，在性能和持久化方面做了很好的折中，推荐
# appendfsync no     //完全依赖os，性能最好,持久化没保证
```

aof 的方式也同时带来了另一个问题。持久化文件会变的越来越大。例如我们调用incr test命令100次，文件中必须保存全部的100条命令，其实有99条都是多余的。因为要恢复数据库的状态其实文件中保存一条set test 100就够了。

为了压缩aof的持久化文件。redis提供了bgrewriteaof命令。收到此命令redis将使用与快照类似的方式将内存中的数据 以命令的方式保存到临时文件中，最后替换原来的文件。具体过程如下

- redis调用fork ，现在有父子两个进程
- 子进程根据内存中的数据库快照，往临时文件中写入重建数据库状态的命令
- 父进程继续处理client请求，除了把写命令写入到原来的aof文件中。同时把收到的写命令缓存起来。这样就能保证如果子进程重写失败的话并不会出问题。
- 当子进程把快照内容写入已命令方式写到临时文件中后，子进程发信号通知父进程。然后父进程把缓存的写命令也写入到临时文件。
- 现在父进程可以使用临时文件替换老的aof文件，并重命名，后面收到的写命令也开始往新的aof文件中追加。

需要注意到是重写aof文件的操作，并没有读取旧的aof文件，而是将整个内存中的数据库内容用命令的方式重写了一个新的aof文件,这点和快照有点类似。

优势

- 使用 AOF 持久化会让 Redis 变得非常耐久（much more durable）：你可以设置不同的 fsync 策略，比如无 fsync ，每秒钟一次 fsync ，或者每次执行写入命令时 fsync 。 AOF 的默认策略为每秒钟 fsync 一次，在这种配置下，Redis 仍然可以保持良好的性能，并且就算发生故障停机，也最多只会丢失一秒钟的数据（ fsync 会在后台线程执行，所以主线程可以继续努力地处理命令请求）。

- AOF 文件是一个只进行追加操作的日志文件（append only log）， 因此对 AOF 文件的写入不需要进行 seek ， 即使日志因为某些原因而包含了未写入完整的命令（比如写入时磁盘已满，写入中途停机，等等）， redis-check-aof 工具也可以轻易地修复这种问题。
  Redis 可以在 AOF 文件体积变得过大时，自动地在后台对 AOF 进行重写： 重写后的新 AOF 文件包含了恢复当前数据集所需的最小命令集合。 整个重写操作是绝对安全的，因为 Redis 在创建新 AOF 文件的过程中，会继续将命令追加到现有的 AOF 文件里面，即使重写过程中发生停机，现有的 AOF 文件也不会丢失。 而一旦新 AOF 文件创建完毕，Redis 就会从旧 AOF 文件切换到新 AOF 文件，并开始对新 AOF 文件进行追加操作。

- AOF 文件有序地保存了对数据库执行的所有写入操作， 这些写入操作以 Redis 协议的格式保存， 因此 AOF 文件的内容非常容易被人读懂， 对文件进行分析（parse）也很轻松。 导出（export） AOF 文件也非常简单： 举个例子， 如果你不小心执行了 FLUSHALL 命令， 但只要 AOF 文件未被重写， 那么只要停止服务器， 移除 AOF 文件末尾的 FLUSHALL 命令， 并重启 Redis ， 就可以将数据集恢复到 FLUSHALL 执行之前的状态。

劣势

- 对于相同的数据集来说，AOF 文件的体积通常要大于 RDB 文件的体积。
- 根据所使用的 fsync 策略，AOF 的速度可能会慢于 RDB 。 在一般情况下， 每秒 fsync 的性能依然非常高， 而关闭 fsync 可以让 AOF 的速度和 RDB 一样快， 即使在高负荷之下也是如此。 不过在处理巨大的写入载入时，RDB 可以提供更有保证的最大延迟时间（latency）。
- AOF 在过去曾经发生过这样的 bug： 因为个别命令的原因，导致 AOF 文件在重新载入时，无法将数据集恢复成保存时的原样。 （举个例子，阻塞命令 BRPOPLPUSH 就曾经引起过这样的 bug 。） 测试套件里为这种情况添加了测试：它们会自动生成随机的、复杂的数据集， 并通过重新载入这些数据来确保一切正常。 虽然这种 bug 在 AOF 文件中并不常见， 但是对比来说， RDB 几乎是不可能出现这种 bug 的。

抉择

一般来说， 如果想达到足以媲美 PostgreSQL 的数据安全性， 你应该同时使用两种持久化功能。

如果你非常关心你的数据， 但仍然可以承受数分钟以内的数据丢失， 那么你可以只使用 RDB 持久化。

其余情况我个人喜好选择AOF

三

1. Snapshotting:
缺省情况下，Redis会将数据集的快照dump到dump.rdb文件中。此外，我们也可以通过配置文件来修改Redis服务器dump快照的频率，在打开6379.conf文件之后，我们搜索save，可以看到下面的配置信息：
save 900 1 #在900秒(15分钟)之后，如果至少有1个key发生变化，则dump内存快照。
save 300 10 #在300秒(5分钟)之后，如果至少有10个key发生变化，则dump内存快照。
save 60 10000 #在60秒(1分钟)之后，如果至少有10000个key发生变化，则dump内存快照。

2. Dump快照的机制：
1). Redis先fork子进程。
2). 子进程将快照数据写入到临时RDB文件中。
3). 当子进程完成数据写入操作后，再用临时文件替换老的文件。

5.4.3. AOF文件：
上面已经多次讲过，RDB的快照定时dump机制无法保证很好的数据持久性。如果我们的应用确实非常关注此点，我们可以考虑使用Redis中的AOF机制。对于Redis服务器而言，其缺省的机制是

RDB，如果需要使用AOF，则需要修改配置文件中的以下条目：
将appendonly no改为appendonly yes
从现在起，Redis在每一次接收到数据修改的命令之后，都会将其追加到AOF文件中。在Redis下一次重新启动时，需要加载AOF文件中的信息来构建最新的数据到内存中。

5.4.5. AOF的配置：
在Redis的配置文件中存在三种同步方式，它们分别是：
appendfsync always #每次有数据修改发生时都会写入AOF文件。
appendfsync everysec #每秒钟同步一次，该策略为AOF的缺省策略。
appendfsync no #从不同步。高效但是数据不会被持久化。

5.4.6. 如何修复坏损的AOF文件：
1). 将现有已经坏损的AOF文件额外拷贝出来一份。
2). 执行"redis-check-aof --fix <filename>"命令来修复坏损的AOF文件。
3). 用修复后的AOF文件重新启动Redis服务器。

5.4.7. Redis的数据备份：
在Redis中我们可以通过copy的方式在线备份正在运行的Redis数据文件。这是因为RDB文件一旦被生成之后就不会再被修改。Redis每次都是将最新的数据dump到一个临时文件中，之后在利用rename函数原子性的将临时文件改名为原有的数据文件名。因此我们可以说，在任意时刻copy数据文件都是安全的和一致的。鉴于此，我们就可以通过创建cron job的方式定时备份Redis的数据文件，并将备份文件copy到安全的磁盘介质中。

5.5、立即写入

```
//立即保存，同步保存
    public static void syncSave() throws Exception{
        Jedis jedis=new Jedis("127.0.0.1",6379);
        for (int i = 0; i <1000; i++) {
            jedis.set("key"+i, "Hello"+i);
            System.out.println("设置key"+i+"的数据到redis");
            Thread.sleep(2);
        }
        //执行保存，会在服务器下生成一个dump.rdb数据库文件
        jedis.save();
        jedis.close();
        System.out.println("写入完成");
    }
```

运行结果：

```
设置key990的数据到redis
设置key991的数据到redis
设置key992的数据到redis
设置key993的数据到redis
设置key994的数据到redis
设置key995的数据到redis
设置key996的数据到redis
设置key997的数据到redis
设置key998的数据到redis
设置key999的数据到redis
写入完成
```

这里的save方法是同步的，没有写入完成前不执行后面的代码。

5.6、异步写入

```
//异步保存
public static void asyncSave() throws Exception{
    Jedis jedis=new Jedis("127.0.0.1",6379);
    for (int i = 0; i <1000; i++) {
        jedis.set("key"+i, "Hello"+i);
        System.out.println("设置key"+i+"的数据到redis");
        Thread.sleep(2);
    }
    //执行异步保存，会在服务器下生成一个dump.rdb数据库文件
    jedis.bgsave();
    jedis.close();
    System.out.println("写入完成");
}
```

如果数据量非常大，要保存的内容很多，建议使用bgsave，如果内容少则可以使用save方法。关于各方式的比较源自网友的博客。

1、Redis的第一个持久化策略：RDB快照

Redis支持将当前数据的快照存成一个数据文件的持久化机制。而一个持续写入的数据库如何生成快照呢。Redis借助了fork命令的copy on write机制。在生成快照时，将当前进程fork出一个子进程，然后在子进程中循环所有的数据，将数据写成为RDB文件。

我们可以通过Redis的save指令来配置RDB快照生成的时机，比如你可以配置当10分钟以内有100次写入就生成快照，也可以配置当1小时内有1000次写入就生成快照，也可以多个规则一起实施。这些规则的定义就在Redis的配置文件中，你也可以通过Redis的CONFIG SET命令在Redis运行时设置规则，不需要重启Redis。

Redis的RDB文件不会坏掉，因为其写操作是在一个新进程中进行的，当生成一个新的RDB文件时，Redis生成的子进程会先将数据写到一个临时文件中，然后通过原子性rename系统调用将临时文件重命名为RDB文件，这样在任何时候出现故障，Redis的RDB文件都总是可用的。

同时，Redis的RDB文件也是Redis主从同步内部实现中的一环。

但是，我们可以很明显的看到，**RDB有它的不足，就是一旦数据库出现问题，那么我们的RDB文件中保存的数据并不是全新的**，从上次RDB文件生成到 Redis停机这段时间的数据全部丢掉了。在某些业务下，这是可以忍受的，我们也推荐这些业务使用RDB的方式进行持久化，因为开启RDB的代价并不高。 但是对于另外一些对数据安全性要求极高的应用，无法容忍数据丢失的应用，RDB就无能为力了，所以Redis引入了另一个重要的持久化机制：AOF日志。

2、Redis的第二个持久化策略：AOF日志

AOF日志的全称是Append Only File，从名字上我们就能看出来，它是一个追加写入的日志文件。与一般数据库不同的是，**AOF文件是可识别的纯文本，它的内容就是一个个的Redis标准命令。**比如我们进行如下实验，使用Redis2.6 版本，在启动命令参数中设置开启AOF功能：

```
./redis-server --appendonly yes
```

然后我们执行如下的命令：

```
redis 127.0.0.1:6379>  set key1 Hello
OK
redis 127.0.0.1:6379> append key1  " World!"
(integer) 12
redis 127.0.0.1:6379> del key1
(integer) 1
```

```
redis 127.0.0.1:6379> del non_existing_key
(integer) 0
```

这时我们查看AOF日志文件，就会得到如下内容：

```
$ cat appendonly.aof
*2
$6
SELECT
$1
0
*3
$3
set
$4
key1
$5
Hello
*3
$6
append
$4
key1
$7
 World!
*2
$3
del
$4
key1
```

可以看到，写操作都生成了一条相应的命令作为日志。**其中值得注意的是最后一个del命令，它并没有被记录在AOF日志中，这是因为Redis判断出 这个命令不会对当前数据集做出修改。**所以不需要记录这个无用的写命令。另外AOF日志也不是完全按客户端的请求来生成日志的，比如命令INCRBYFLOAT 在记AOF日志时就被记成一条SET记录，因为浮点数操作可能在不同的系统上会不同，所以为了避免同一份日志在不同的系统上生成不同的数据集，所以这里只将操作后的结果通过SET来记录。

AOF重写

你可以会想，每一条写命令都生成一条日志，那么AOF文件是不是会很大？答案是肯定的，AOF文件会越来越大，**所以Redis又提供了一个功能，叫做AOF rewrite。**其功能就是重新生成一份AOF文

件，新的AOF文件中一条记录的操作只会有一次，而不像一份老文件那样，可能记录了对同一个值的多次操作。其生成过程和RDB类似，也是fork一个进程，直接遍历数据，写入新的AOF临时文件。在写入新文件的过程中，所有的写操作日志还是会写到原来老的 AOF文件中，同时还会记录在内存缓冲区中。当重完操作完成后，会将所有缓冲区中的日志一次性写入到临时文件中。然后调用原子性的rename命令用新的 AOF文件取代老的AOF文件。

二、Redis持久化性能是否可靠?

从上面的流程我们能够看到，RDB是顺序IO操作，性能很高。而同时在通过RDB文件进行数据库恢复的时候，也是顺序的读取数据加载到内存中。所以也不会造成磁盘的随机读取错误。

而AOF是一个写文件操作，其目的是将操作日志写到磁盘上，所以它也同样会遇到我们上面说的写操作的5个流程。那么写AOF的操作安全性又有多高呢？实际上这是可以设置的，**在Redis中对AOF调用write写入后，何时再调用fsync将其写到磁盘上，通过appendfsync选项来控制，下面appendfsync的三个设置项，安全强度逐渐变强。**

1、appendfsync no

当设置appendfsync为no的时候，Redis不会主动调用fsync去将AOF日志内容同步到磁盘，所以这一切就完全依赖于操作系统的调试了。**对大多数Linux操作系统，是每30秒进行一次fsync，将缓冲区中的数据写到磁盘上。**

2、appendfsync everysec

当设置appendfsync为everysec的时候，Redis会默认每隔一秒进行一次fsync调用，将缓冲区中的数据写到磁盘。但是当这一 次的fsync调用时长超过1秒时。Redis会采取延迟fsync的策略，再等一秒钟。也就是在两秒后再进行fsync，这一次的fsync就不管会执行多长时间都会进行。这时候由于在fsync时文件描述符会被阻塞，所以当前的写操作就会阻塞。

所以，**结论就是：在绝大多数情况下，Redis会每隔一秒进行一次fsync。在最坏的情况下，两秒钟会进行一次fsync操作。**

这一操作在大多数数据库系统中被称为group commit，就是组合多次写操作的数据，一次性将日志写到磁盘。

3、appednfsync always

当设置appendfsync为always时，每一次写操作都会调用一次fsync，这时数据是最安全的，当然，由于每次都会执行fsync，所以其性能也会受到影响。

对于pipelining有什么不同?

对于pipelining的操作，其具体过程是客户端一次性发送N个命令，然后等待这N个命令的返回结果被一起返回。通过采用pipilining 就意味着放弃了对每一个命令的返回值确认。由于在这种情况下，N个命令是在同一个执行过程中执行的。所以当设置appendfsync为everysec 时，可能会有一些偏差，因为这N个命令可能执行时间超过1秒甚至2秒。但是可以保证的是，最长时间不会超过这N个命令的执行时间和。

三、和其它数据库的比较

上面操作系统层面的数据安全我们已经讲了很多，其实，不同的数据库在实现上都大同小异。总之，最后的结论就是，**在Redis开启AOF的情况下，其单机数据安全性并不比这些成熟的SQL数据库弱。**

在数据导入方面的比较

这些持久化的数据有什么用，当然是用于重启后的数据恢复。Redis是一个内存数据库，无论是RDB还是AOF，都只是其保证数据恢复的措施。所以 Redis在利用RDB和AOF进行恢复的时候，都会读取RDB或AOF文件，重新加载到内存中。相对于MySQL等数据库的启动时间来说，会长很多，因为MySQL本来是不需要将数据加载到内存中的。

但是相对来说，MySQL启动后提供服务时，其被访问的热数据也会慢慢加载到内存中，通常我们称之为预热，而在预热完成前，其性能都不会太高。**而Redis的好处是一次性将数据加载到内存中，一次性预热。**这样只要Redis启动完成，那么其提供服务的速度都是非常快的。

而在利用RDB和利用AOF启动上，其启动时间有一些差别。RDB的启动时间会更短，原因有两个，一是RDB文件中每一条数据只有一条记录，不会像 AOF日志那样可能有一条数据的多次操作记录。所以每条数据只需要写一次就行了。另一个原因是RDB文件的存储格式和Redis数据在内存中的编码格式是一致的，不需要再进行数据编码工作。在CPU消耗上要远小于AOF日志的加载。