Redis 在新浪微博中的应用

**Redis简介**

**1. 支持5种数据结构**

支持strings, hashes, lists, sets, sorted sets  
string是很好的存储方式，用来做计数存储。sets用于建立索引库非常棒；

**2. K-V 存储 vs K-V 缓存**

新浪微博目前使用的98%都是持久化的应用，2%的是缓存，用到了600+服务器  
Redis中持久化的应用和非持久化的方式不会差别很大：  
非持久化的为8-9万tps，那么持久化在7-8万tps左右；  
当使用持久化时，需要考虑到持久化和写性能的配比，也就是要考虑redis使用的内存大小和硬盘写的速率的比例计算；

**3. 社区活跃**

Redis目前有3万多行代码, 代码写的精简，有很多巧妙的实现，作者有技术洁癖  
Redis的社区活跃度很高，这是衡量开源软件质量的重要指标，开源软件的初期一般都没有商业技术服务支持，如果没有活跃社区做支撑，一旦发生问题都无处求救；

**Redis基本原理**

redis持久化(aof) append online file：  
写log(aof), 到一定程度再和内存合并. 追加再追加, 顺序写磁盘, 对性能影响非常小

**1. 单实例单进程**

Redis使用的是单进程，所以在配置时，一个实例只会用到一个CPU；  
在配置时，如果需要让CPU使用率最大化，可以配置Redis实例数对应CPU数, Redis实例数对应端口数(8核Cpu, 8个实例, 8个端口), 以提高并发:  
单机测试时, 单条数据在200字节, 测试的结果为8~9万tps；

**2. Replication**

过程: 数据写到master-->master存储到slave的rdb中-->slave加载rdb到内存。  
存储点(save point): 当网络中断了, 连上之后, 继续传.  
Master-slave下第一次同步是全传，后面是增量同步；、

**3. 数据一致性**

长期运行后多个结点之间存在不一致的可能性；  
开发两个工具程序：  
1.对于数据量大的数据，会周期性的全量检查；  
2.实时的检查增量数据，是否具有一致性；

对于主库未及时同步从库导致的不一致，称之为延时问题；  
对于一致性要求不是那么严格的场景，我们只需要要保证最终一致性即可；  
对于延时问题，需要根据业务场景特点分析，从应用层面增加策略来解决这个问题；  
例如：  
1.新注册的用户，必须先查询主库；  
2.注册成功之后，需要等待3s之后跳转，后台此时就是在做数据同步。

**新浪Redis使用历程**

2009年, 使用memcache(用于非持久化内容), memcacheDB(用于持久化+计数),  
memcacheDB是新浪在memcache的基础上，使用BerkeleyDB作为数据持久化的存储实现；

**1. 面临的问题**

* 数据结构(Data Structure)需求越来越多, 但memcache中没有, 影响开发效率
* 性能需求, 随着读操作的量的上升需要解决，经历的过程有:  
  数据库读写分离(M/S)-->数据库使用多个Slave-->增加Cache (memcache)-->转到Redis
* 解决写的问题：  
  水平拆分，对表的拆分，将有的用户放在这个表，有的用户放在另外一个表；
* 可靠性需求  
  Cache的"雪崩"问题让人纠结  
  Cache面临着快速恢复的挑战
* 开发成本需求  
  Cache和DB的一致性维护成本越来越高(先清理DB, 再清理缓存, 不行啊, 太慢了!)  
  开发需要跟上不断涌入的产品需求  
  硬件成本最贵的就是数据库层面的机器，基本上比前端的机器要贵几倍，主要是IO密集型，很耗硬件；
* 维护性复杂  
  一致性维护成本越来越高；  
  BerkeleyDB使用B树，会一直写新的，内部不会有文件重新组织；这样会导致文件越来越大；大的时候需要进行文件归档，归档的操作要定期做；  
  这样，就需要有一定的down time；

基于以上考虑， 选择了Redis

**2. 寻找开源软件的方式及评判标准**

* 对于开源软件，首先看其能做什么，但更多的需要关注它不能做什么，它会有什么问题？
* 上升到一定规模后，可能会出现什么问题，是否能接受？
* google code上, 国外论坛找材料(国内比国外技术水平滞后5年）
* 观察作者个人的代码水平

**Redis应用场景**

**1. 业务使用方式**

* hash sets: 关注列表, 粉丝列表, 双向关注列表(key-value(field), 排序)
* string(counter): 微博数, 粉丝数, ...(避免了select count(\*) from ...)
* sort sets(自动排序): TopN, 热门微博等, 自动排序
* lists(queue): push/sub提醒,...

上述四种, 从精细化控制方面，hash sets和string(counter)推荐使用, sort sets和lists(queue)不推荐使用  
还可通过二次开发，进行精简。比如: 存储字符改为存储整形, 16亿数据, 只需要16G内存  
存储类型保存在3种以内，建议不要超过3种；

将memcache +myaql 替换为Redis：  
Redis作为存储并提供查询，后台不再使用mysql，解决数据多份之间的一致性问题；

**2. 对大数据表的存储**

（eg：140字微博的存储）  
一个库就存唯一性id和140个字；  
另一个库存id和用户名，发布日期、点击数等信息，用来计算、排序等，等计算出最后需要展示的数据时再到第一个库中提取微博内容；

改进的3个步骤:  
1）发现现有系统存在问题;  
2）发现了新东西, 怎么看怎么好, 全面转向新东西;  
3）理性回归, 判断哪些适合新东西, 哪些不适合, 不合适的回迁到老系统

**3. 一些技巧**

* 很多应用, 可以承受数据库连接失败, 但不能承受处理慢
* 一份数据, 多份索引(针对不同的查询场景)
* 解决IO瓶颈的唯一途径: 用内存
* 在数据量变化不大的情况下，优先选用Redis

**遇到的问题及解决办法**

(注意: 都是量特别大时候会出现的, 量小了怎么都好说)

**1.Problem: Replication中断后, 重发-->网络突发流量**

Solution: 重写Replication代码, rdb+aof(滚动)

**2.Problem: 容量问题**

Solution: 容量规划和M/S的sharding功能(share nothing, 抽象出来的数据对象之间的关联数据很小)  
增加一些配置, 分流, 比如: 1,2,3,4, 机器1处理%2=1的, 机器2处理%2=0的.  
低于内存的1/2使用量, 否则就扩容（建议Redis实例使用的数据，最大不要超过内存的80%）  
我们线上96G/128G内存服务器不建议单实例容量大于20/30G。  
微博应用中单表数据最高的有2T的数据，不过应用起来已经有些力不从心；  
每个的端口不要超过20G；测试磁盘做save所需要的时间，需要多长时间能够全部写入；内存越大，写的时间也就越长；  
单实例内存容量较大后，直接带来的问题就是故障恢复或者Rebuild从库的时候时间较长，对于普通硬盘的加载速度而言，我们的经验一般是redis加载1G需要1分钟；（加载的速度依赖于数据量的大小和数据的复杂度）  
Redis rewrite aof和save rdb时，将会带来非常大且长的系统压力，并占用额外内存，很可能导致系统内存不足等严重影响性能的线上故障。

reblance: 现有数据按照上述配置重新分发。  
后面使用中间层，路由HA；  
注：目前官方也正在做这个事，Redis Cluster，解决HA问题；

**3. Problem: bgsave or bgwriteaof的冰晶问题**

Solution: 磁盘性能规划和限制写入的速度, 比如: 规定磁盘以200M/s的速度写入, 细水长流, 即使到来大量数据. 但是要注意写入速度要满足两个客观限制:  
符合磁盘速度  
符合时间限制(保证在高峰到来之前, 就得写完)

**4.Problem: 运维问题**

1）Inner Crontab: 把Crontab迁移到Redis内部, 减少迁移时候的压力  
本机多端口避免同时做 - 能做到  
同一业务多端口(分布在多机上), 避免同时做 - 做不到  
2）动态升级: 先加载.so文件, 再管理配置, 切换到新代码上(Config set命令)  
把对redis改进的东西都打包成lib.so文件，这样能够支持动态升级  
自己改的时候要考虑社区的升级。当社区有新的版本，有很好用的新功能时，要能很容易的与我们改进后的版本很好的merge；  
升级的前提条件: 模块化, 以模块为单位升级  
加载时间取决于两个方面: 数据大小, 数据结构复杂度. 一般, 40G数据耗时40分钟  
分布式系统的两个核心问题: A.路由问题 B.HA问题

3）危险命令的处理: 比如: fresh all删除全部数据, 得进行控制  
运维不能只讲数据备份，还得考虑数据恢复所需要的时间；  
增加权限认证(管理员才有权限)eg：flashall 权限认证，得有密码才能做；  
当然，高速数据交互一般都不会在每次都进行权限认证，通用的处理策略是第一次认证，后期都不用再认证；  
控制hash策略(没有key, 就找不到value; 不知道hash策略, 就无法得到key)

4）Config Dump:  
内存中的配置项动态修改过, 按照一定策略写入到磁盘中(Redis已支持)  
5）bgsave带来aof写入很慢:  
fdatasync在做bgsave时, 不做sync aof(会有数据出入)  
6）成本问题: (22T内存, 有10T用来计数)  
Redisscounter(16亿数据占用16G内存) - 全部变为整型存储, 其余(字符串等)全不要  
Redis+SSD(counterService计数服务)  
顺序自增, table按照顺序写, 写满10个table就自动落地(到SSD)  
存储分级: 内存分配问题, 10K和100K写到一块, 会有碎片. Sina已经优化到浪费只占5%以内(已经很好了!)

**5.Problem: 分布式问题**

1.Config Server: 命名空间, 特别大的告诉访问, 都不适合用代理, 因为代理降低速度, 但是, Sina用了(单机多端口, Redis Cluster, sentinel)  
Config Server放到Zookeeper上  
最前面是命名服务，后面跟的是无状态的twmemproxy（twitter的改进的，用C写的） ,后面才是redis；  
2.twmemproxy  
应用不必关心连接失败, 由代理负责重连  
把Hash算法放到代理商  
代理后边的升级, 前端不关心, 解决了HA的问题  
无状态, 多台代理无所谓  
3.AS --> Proxy -->Redis  
4.Sina的Redis都是单机版, 而Redis-Cluster交互过于复杂，没有使用  
做HA的话，一定要配合监控来做，如果挂了之后，后续该如何做；  
并不是追求单机性能，而是集群的吞吐量，从而可以支持无线扩展；

**经验总结**

* 提前做好数据量的规划, 减少sharding(互联网公司一般以年为单位)
* 只存精细化数据(内存很金贵!)
* 存储用户维度的数据  
  对象维度的数据要有生命周期  
  特别是数据量特别大的时候，就很有必要来进行划分了；
* 暴露服务的常见过程: IP-->负载均衡-->域名-->命名服务(一张表: 名字+资源(IP+端口))
* 对于硬件消耗，IO、网络和CPU相比，Redis最消耗的是CPU，复杂的数据类型必定带来CPU消耗；
* 新浪微博响应时间超时目前设置为5s；（返回很慢的记录key，需记录下来分析，慢日志）；
* 备份的数据要定期要跑一下生产的数据；用来检查备份数据的有效性；
* slave挂多了肯定会对master造成比较的影响；新浪微博目前使用的M/S是一拖一，主要用来做容灾；  
  同步时，是fork出一个单独进程来和slave进行同步；不会占用查询的进程；
* 升级到2.6.30以后的linux内核；  
  在2.6.30以上对软中断的问题处理的很好，性能提升效果明显，差不多有15%到30%的差距；
* redis不用读写分离，每个请求都是单线程，为什么要进行读写分离。