**[Redis几个认识误区](http://timyang.net/data/redis-misunderstanding/" \o "Permanent Link to Redis几个认识误区)**

Saturday, Dec 4th, 2010 by Tim | Tags: [key value store](http://timyang.net/tag/key-value-store/), [redis](http://timyang.net/tag/redis/)

前几天微博发生了一起大的[系统故障](http://tech.sina.com.cn/i/2010-12-01/15324930344.shtml)，很多技术的朋友都比较关心，其中的原因不会超出James Hamilton在On Designing and Deploying Internet-Scale Service(1)概括的那几个范围，James第一条经验“Design for failure”是所有互联网架构成功的一个关键。互联网系统的工程理论其实非常简单，James paper中内容几乎称不上理论，而是多条实践经验分享，每个公司对这些经验的理解及执行力决定了架构成败。

题外话说完，最近又研究了[Redis](http://code.google.com/p/redis/)。去年曾做过一个[MemcacheDB, Tokyo Tyrant, Redis performance test](http://timyang.net/data/mcdb-tt-redis/)，到目前为止，这个benchmark结果依然有效。这1年我们经历了很多眼花缭乱的key value存储产品的诱惑，从Cassandra的淡出(Twitter暂停在主业务使用)到HBase的兴起(Facebook新的邮箱业务选用HBase(2))，当再回头再去看Redis，发现这个只有1万多行源代码的程序充满了神奇及大量未经挖掘的特性。Redis性能惊人，国内前十大网站的子产品估计用1台Redis就可以满足存储及Cache的需求。除了性能印象之外，业界其实普遍对Redis的认识存在一定误区。本文提出一些观点供大家探讨。

**1. Redis是什么**

这个问题的结果影响了我们怎么用Redis。如果你认为Redis是一个key value store, 那可能会用它来代替MySQL；如果认为它是一个可以持久化的cache, 可能只是它保存一些频繁访问的临时数据。Redis是**REmote DIctionary Server**的缩写，在Redis在官方网站的的副标题是A persistent key-value database with built-in net interface written in ANSI-C for Posix systems，这个定义偏向key value store。还有一些看法则认为Redis是一个memory database，因为它的高性能都是基于内存操作的基础。另外一些人则认为Redis是一个data structure server，因为Redis支持复杂的数据特性，比如List, Set等。对Redis的作用的不同解读决定了你对Redis的使用方式。

互联网数据目前基本使用两种方式来存储，关系数据库或者key value。但是这些互联网业务本身并不属于这两种数据类型，比如用户在社会化平台中的关系，它是一个list，如果要用关系数据库存储就需要转换成一种多行记录的形式，这种形式存在很多冗余数据，每一行需要存储一些重复信息。如果用key value存储则修改和删除比较麻烦，需要将全部数据读出再写入。Redis在内存中设计了各种数据类型，让业务能够高速原子的访问这些数据结构，并且不需要关心持久存储的问题，从架构上解决了前面两种存储需要走一些弯路的问题。

**2. Redis不可能比Memcache快**

很多开发者都认为Redis不可能比Memcached快，Memcached完全基于内存，而Redis具有持久化保存特性，即使是异步的，Redis也不可能比Memcached快。但是测试结果基本是Redis占绝对优势。一直在思考这个原因，目前想到的原因有这几方面。

* [Libevent](http://monkey.org/~provos/libevent/)。和Memcached不同，Redis并没有选择libevent。Libevent为了迎合通用性造成代码庞大(目前Redis代码还不到libevent的1/3)及牺牲了在特定平台的不少性能。Redis用libevent中两个文件修改实现了自己的epoll event loop(4)。业界不少开发者也建议Redis使用另外一个libevent高性能替代libev，但是作者还是坚持Redis应该小巧并去依赖的思路。一个印象深刻的细节是编译Redis之前并不需要执行./configure。
* CAS问题。CAS是Memcached中比较方便的一种防止竞争修改资源的方法。CAS实现需要为每个cache key设置一个隐藏的cas token，cas相当value版本号，每次set会token需要递增，因此带来CPU和内存的双重开销，虽然这些开销很小，但是到单机10G+ cache以及QPS上万之后这些开销就会给双方相对带来一些细微性能差别(5)。

**3. 单台Redis的存放数据必须比物理内存小**

Redis的数据全部放在内存带来了高速的性能，但是也带来一些不合理之处。比如一个中型网站有100万注册用户，如果这些资料要用Redis来存储，内存的容量必须能够容纳这100万用户。但是业务实际情况是100万用户只有5万活跃用户，1周来访问过1次的也只有15万用户，因此全部100万用户的数据都放在内存有不合理之处，RAM需要为冷数据买单。

这跟操作系统非常相似，操作系统所有应用访问的数据都在内存，但是如果物理内存容纳不下新的数据，操作系统会智能将部分长期没有访问的数据交换到磁盘，为新的应用留出空间。现代操作系统给应用提供的并不是物理内存，而是虚拟内存(Virtual Memory)的概念。

基于相同的考虑，Redis 2.0也增加了VM特性。让Redis数据容量突破了物理内存的限制。并实现了数据冷热分离。

**4. Redis的VM实现是重复造轮子**

Redis的VM依照之前的epoll实现思路依旧是自己实现。但是在前面操作系统的介绍提到OS也可以自动帮程序实现冷热数据分离，Redis只需要OS申请一块大内存，OS会自动将热数据放入物理内存，冷数据交换到硬盘，另外一个知名的“理解了现代操作系统(3)”的Varnish就是这样实现，也取得了非常成功的效果。

作者antirez在解释为什么要自己实现VM中提到几个原因(6)。主要OS的VM换入换出是基于Page概念，比如OS VM1个Page是4K, 4K中只要还有一个元素即使只有1个字节被访问，这个页也不会被SWAP, 换入也同样道理，读到一个字节可能会换入4K无用的内存。而Redis自己实现则可以达到控制换入的粒度。另外访问操作系统SWAP内存区域时block进程，也是导致Redis要自己实现VM原因之一。

**5. 用get/set方式使用Redis**

作为一个key value存在，很多开发者自然的使用set/get方式来使用Redis，实际上这并不是最优化的使用方法。尤其在未启用VM情况下，Redis全部数据需要放入内存，节约内存尤其重要。

假如一个key-value单元需要最小占用512字节，即使只存一个字节也占了512字节。这时候就有一个设计模式，可以把key复用，几个key-value放入一个key中，value再作为一个set存入，这样同样512字节就会存放10-100倍的容量。

这就是为了节约内存，建议使用hashset而不是set/get的方式来使用Redis，详细方法见参考文献(7)。

**6. 使用aof代替snapshot**

Redis有两种存储方式，默认是snapshot方式，实现方法是定时将内存的快照(snapshot)持久化到硬盘，这种方法缺点是持久化之后如果出现crash则会丢失一段数据。因此在完美主义者的推动下作者增加了aof方式。aof即append only mode，在写入内存数据的同时将操作命令保存到日志文件，在一个并发更改上万的系统中，命令日志是一个非常庞大的数据，管理维护成本非常高，恢复重建时间会非常长，这样导致失去aof高可用性本意。另外更重要的是Redis是一个内存数据结构模型，所有的优势都是建立在对内存复杂数据结构高效的原子操作上，这样就看出aof是一个非常不协调的部分。

其实aof目的主要是数据可靠性及高可用性，在Redis中有另外一种方法来达到目的：Replication。由于Redis的高性能，复制基本没有延迟。这样达到了防止单点故障及实现了高可用。

**小结**

要想成功使用一种产品，我们需要深入了解它的特性。Redis性能突出，如果能够熟练的驾驭，对国内很多大型应用具有很大帮助。希望更多同行加入到Redis使用及代码研究行列。

### Redis与Memcached性能对比

虽然Memcached完全基于内存，而Redis具有持久化保存特性而且还是异步的，但经过测试Redis还是比Memcached快。  
1） Libevent为了通用性使代码很庞大  
2） CAS问题。CAS是Memcached中比较方便的一种防止竞争修改资源的方法。CAS实现需要为每个cache key设置一个隐藏的cas token，cas相当value版本号，每次set会token需要递增，因此带来CPU和内存的双重开销，虽然这些开销很小，但是到单机10G+ cache以及QPS上万之后这些开销就会给双方相对带来一些细微性能差别  
Redis 可以通过虚拟内存的方式利用硬盘虚拟化出一块内存，将使用率不是太高的数据放入虚拟内存。  
为了节约内存，建议使用hashset而不是set/get的方式来使用Redis（hashset可以把key复用，几个key-value放入一个key中，value再作为一个set存入）  
3） Memcached使用libevent库，而Redis则原生的使用了epoll，kqueue等多个异步通信模型。使用snapshot(定时将内存的快照(snapshot)持久化到硬盘)代替aof (写入内存数据的同时将操作命令保存到日志文件)

### 特性

1） 数据类型：Redis可以存set,list,String,hash,sorted set等多种数据类型的数据

2） 排序： SORT key [asc] [alpha] [limit start count]  
Lpush key1 12  
Lpush key1 14  
Sort key1  
3） 计数： 有了原子递增，你可以放心的加上各种计数，用getset重置，或者是让他们过期  
1. INCR user:<id> EXPIRE  
2. User:<id> 60  
可以计算出最近用户在各邮件停顿不超过60秒的邮件浏览量，当计数达到一个值时可以显示邮件的具体信息从而可以分析他想要找哪方面的信息。  
实时分析正在发生的情况  
统计在某段特定时间里有多少特定用户访问了某个特定资源，比如我想要知道某些特征用户或IP地址，他们有多少访问了某个邮件及访问的时间长度。比如线次我获得一次新邮件浏览时可以使用此命令:  
SADD email:day1:<email\_id><user\_id> day1也可用time()-(time()600\*24) 替代  
特定用户的数量： SCARD email:day1:<email\_id>  
测试某个特定用户是否访问了这个邮件： SISMEMBER email:day1:<email\_id>

4） 事务：  
Redis只能保证一个client 发起的事务中的命令可以连续的执行，中间不会插入其他client的命令，因为redis是单线程来处理所有client的请求所以很容易做到这一点。一般是发来一个命令后立即处理并返回结果，但如果client发出multi命令后此连接会进入一个事务上下文，该连接后续命令会放到一个队列中，当发出exec命令后再顺序执行命令。  
Ex: multi (OK) incr a (QUEUED) incr b (QUEUED) exec (1.(integer) 1 2.(integer) 2)

乐观锁：Watch 命令会监视给定的key,当exec 时候如果监视的key从调用watch后发生过变化，则整个事务会失败  
Watch a (OK) incr a (QUEUED) incr b (QUEUED) exec (nil)

5） 支持发布订阅： 订阅者可以通过subscribe和psubscribe命令向redis订阅自己感兴趣的消息类型  
持久化 (Snapshotting & Append-only file aof)  
主从复制： 多个slave连到master, slave互连.  
1) 主从复制不会阻塞master  
2) 提高系统伸缩性，slave分别负责读请求，排序，数据冗余  
3) 禁掉master上的数据持久化(save配置信息)，转到某个slave来进行。  
过程：   
1) master启动一个后台进程，将数据库快照保存到文件中，收集新的写命令并缓存，批量发送给slave  
2) slave将文件保存到磁盘上然后加载到内存恢复快照到slave上  
3) master把缓存的命令转发给slave. (写一个文件将其发送给所有slave)

6） 虚拟内存：  
将不经常访问的数据放到交换分区中