[Redis 持久化，写入磁盘的方式](https://www.cnblogs.com/datang6777/p/7056272.html)

[Redis](http://lib.csdn.net/base/redis)是一个支持持久化的内存[数据库](http://lib.csdn.net/base/mysql%22%20%5Co%20%22MySQL%E7%9F%A5%E8%AF%86%E5%BA%93%22%20%5Ct%20%22_blank)=>也就是说[redis](http://lib.csdn.net/base/redis%22%20%5Co%20%22Redis%E7%9F%A5%E8%AF%86%E5%BA%93%22%20%5Ct%20%22_blank)需要经常将内存中的数据同步到磁盘来保证持久化。

redis支持四种持久化方式，

一是 Snapshotting（快照）也是默认方式；

二是Append-only file（缩写aof）的方式；

三是虚拟内存方式；

四是diskstore方式。

**一）Snapshotting**

       快照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 |

快照保存过程：

       1. redis调用fork,现在有了子进程和父进程。

       2. 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。
       3. 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（fork一个进程入内在也被复制了，即内存会是原来的两倍）。

       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。**       另外由于快照方式是在一定间隔时间做一次的，所以如果redis意外down掉的话，就会丢失最后一次快照后的所有修改。如果应用要求不能丢失任何修改的话，可以采用aof持久化方式。下面介绍：

**（二）Append-only file**

　　aof 比快照方式有更好的持久化性，是由于在使用**aof持久化方式时，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，不推荐使用**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：

       1.  redis调用fork ，现在有父子两个进程
       2. 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
       3. 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。
       4. 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。
       5. 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。

**（三）虚拟内存方式（desprecated）**

首先说明：在Redis-2.4后虚拟内存功能已经被deprecated了，原因如下：

1）slow restart重启太慢

2）slow saving保存数据太慢

3）slow replication上面两条导致 replication 太慢

4）complex code代码过于复杂

下面还是介绍一下redis的虚拟内存。

redis的虚拟内存与os的虚拟内存不是一码事，但是思路和目的都是相同的。就是暂时把不经常访问的数据从内存交换到磁盘中，从而腾出宝贵的内存空间用于其他需要访问的数据。尤其是对于redis这样的内存数据库，内存总是不够用的。除了可以将数据分割到多个redis server外。另外的能够提高数据库容量的办法就是使用vm把那些不经常访问的数据交换的磁盘上。如果我们的存储的数据总是有少部分数据被经常访问，大部分数据很少被访问，对于网站来说确实总是只有少量用户经常活跃。当少量数据被经常访问时，使用vm不但能提高单台redis server数据库的容量，而且也不会对性能造成太多影响。

        redis没有使用os提供的虚拟内存机制而是自己在用户态实现了自己的虚拟内存机制,作者在自己的blog专门解释了其中原因。

<http://antirez.com/post/redis-virtual-memory-story.html>
主要的理由有两点：
       1. os 的虚拟内存是已4k页面为最小单位进行交换的。而redis的大多数对象都远小于4k，所以一个os页面上可能有多个redis对象。另外redis的集合对象类型如list,set可能存在与多个os页面上。最终可能造成只有10%key被经常访问，但是所有os页面都会被os认为是活跃的，这样只有内存真正耗尽时os才会交换页面。
       2.相比于os的交换方式。redis可以将被交换到磁盘的对象进行压缩,保存到磁盘的对象可以去除指针和对象元数据信息。一般压缩后的对象会比内存中的对象小10倍。这样redis的vm会比os vm能少做很多io操作。

       下面是vm相关配置：

|  |
| --- |
| slaveof 192.168.1.1 6379  #指定master的ip和端口vm-enabled yes          #开启vm功能vm-swap-file /tmp/redis.swap   #交换出来的value保存的文件路径/tmp/redis.swapvm-max-memory 1000000  #redis使用的最大内存上限，超过上限后redis开始交换value到磁盘文件中vm-page-size 32        #每个页面的大小32个字节vm-pages 134217728     #最多使用在文件中使用多少页面,交换文件的大小 = vm-page-size \* vm-pagesvm-max-threads 4       #用于执行value对象换入换出的工作线程数量，0表示不使用工作线程（后面介绍) |

       redis的vm在设计上为了保证key的查找速度，只会将value交换到swap文件中。所以如果是内存问题是由于太多value很小的key造成的，那么vm并不能解决。和os一样redis也是按页面来交换对象的。redis规定同一个页面只能保存一个对象。但是一个对象可以保存在多个页面中。

在redis使用的内存没超过vm-max-memory之前是不会交换任何value的。当超过最大内存限制后，redis会选择较老的对象。如果两个对象一样老会优先交换比较大的对象，精确的公式swappability = age\*log(size\_in\_memory)。对于vm-page-size的设置应该根据自己的应用将页面的大小设置为可以容纳大多数对象的大小。太大了会浪费磁盘空间，太小了会造成交换文件出现碎片。对于交换文件中的每个页面，redis会在内存中对应一个1bit值来记录页面的空闲状态。所以像上面配置中页面数量(vm-pages 134217728 )会占用16M内存用来记录页面空闲状态。vm-max-threads表示用做交换任务的线程数量。如果大于0推荐设为服务器的cpu core的数量。如果是0则交换过程在主线程进行。

参数配置讨论完后，在来简单介绍下vm是如何工作的：
**当vm-max-threads设为0时(Blocking VM)**

**换出：**
       主线程定期检查发现内存超出最大上限后，会直接已阻塞的方式,将选中的对象保存到swap文件中，并释放对象占用的内存,此过程会一直重复直到下面条件满足
       1.内存使用降到最大限制以下
       2.swap文件满了
       3.几乎全部的对象都被交换到磁盘了
**换入：**
              当有client请求value被换出的key时。主线程会以阻塞的方式从文件中加载对应的value对象，加载时此时会阻塞所有client。然后处理client的请求

**当vm-max-threads大于0(Threaded VM)**
**换出：**
       当主线程检测到使用内存超过最大上限，会将选中的要交换的对象信息放到一个队列中交由工作线程后台处理，主线程会继续处理client请求。
**换入：**
       如果有client请求的key被换出了，主线程先阻塞发出命令的client,然后将加载对象的信息放到一个队列中，让工作线程去加载。加载完毕后工作线程通知主线程。主线程再执行client的命令。这种方式只阻塞请求value被换出key的client

       总的来说blocking vm的方式总的性能会好一些，因为不需要线程同步，创建线程和恢复被阻塞的client等开销。但是也相应的牺牲了响应性。threaded vm的方式主线程不会阻塞在磁盘io上，所以响应性更好。如果我们的应用不太经常发生换入换出，而且也不太在意有点延迟的话则推荐使用blocking vm的方式。

关于redis vm的更详细介绍可以参考下面链接：
       <http://antirez.com/post/redis-virtual-memory-story.html>
       <http://redis.io/topics/internals-vm>

**（四）diskstore方式**

diskstore方式是作者放弃了虚拟内存方式后选择的一种新的实现方式，也就是传统的B-tree的方式。具体细节是：

1) 读操作，使用read through以及LRU方式。内存中不存在的数据从磁盘拉取并放入内存，内存中放不下的数据采用LRU淘汰。

2) 写操作，采用另外spawn一个线程单独处理，写线程通常是异步的，当然也可以把cache-flush-delay配置设成0，Redis尽量保证即时写入。但是在很多场合延迟写会有更好的性能，比如一些计数器用Redis存储，在短时间如果某个计数反复被修改，Redis只需要将最终的结果写入磁盘。这种做法作者叫per key persistence。由于写入会按key合并，因此和snapshot还是有差异，disk store并不能保证时间一致性。

由于写操作是单线程，即使cache-flush-delay设成0，多个client同时写则需要排队等待，如果队列容量超过cache-max-memory Redis设计会进入等待状态，造成调用方卡住。

Google Group上有热心网友迅速完成了压力[测试](http://lib.csdn.net/base/softwaretest)，当内存用完之后，set每秒处理速度从25k下降到10k再到后来几乎卡住。 虽然通过增加cache-flush-delay可以提高相同key重复写入性能；通过增加cache-max-memory可以应对临时峰值写入。但是diskstore写入瓶颈最终还是在IO。

3) rdb 和新 diskstore 格式关系
rdb是传统Redis内存方式的存储格式，diskstore是另外一种格式，那两者关系如何？

·         通过BGSAVE可以随时将diskstore格式另存为rdb格式，而且rdb格式还用于Redis复制以及不同存储方式之间的中间格式。

·         通过工具可以将rdb格式转换成diskstore格式。

当然，diskstore原理很美好，但是目前还处于alpha版本，也只是一个简单demo，diskstore.c加上注释只有300行，实现的方法就是将每个value作为一个独立文件保存，文件名是key的hash值。因此diskstore需要将来有一个更高效稳定的实现才能用于生产环境。但由于有清晰的接口设计，diskstore.c也很容易换成一种B-Tree的实现。很多开发者也在积极探讨使用bdb或者innodb来替换默认diskstore.c的可行性。

下面介绍一下Diskstore的[算法](http://lib.csdn.net/base/datastructure%22%20%5Co%20%22%E7%AE%97%E6%B3%95%E4%B8%8E%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84%E7%9F%A5%E8%AF%86%E5%BA%93%22%20%5Ct%20%22_blank)。

其实DiskStore类似于Hash算法，首先通过SHA1算法把Key转化成一个40个字符的Hash值，然后把Hash值的前两位作为一级目录，然后把Hash值的三四位作为二级目录，最后把Hash值作为文件名，类似于“/0b/ee/0beec7b5ea3f0fdbc95d0dd47f3c5bc275da8a33”形式。算法如下：

dsKeyToPath(key):

char path[1024];

char \*hashKey = sha1(key);

path[0] = hashKey[0];

path[1] = hashKey[1];

path[2] = '/';

path[3] = hashKey[2];

path[4] = hashKey[3];

path[5] = '/';

memcpy(path + 6, hashKey, 40);

return path;

存储算法（如key == apple）：

dsSet(key, value, expireTime):

    // d0be2dc421be4fcd0172e5afceea3970e2f3d940

char \*hashKey = sha1(key);

// d0/be/d0be2dc421be4fcd0172e5afceea3970e2f3d940

char \*path = dsKeyToPath(hashKey);

FILE \*fp = fopen(path, "w");

rdbSaveKeyValuePair(fp, key, value, expireTime);

fclose(fp)

获取算法：

dsGet(key):

char \*hashKey = sha1(key);

char \*path = dsKeyToPath(hashKey);

FILE \*fp = fopen(path, "r");

robj \*val = rdbLoadObject(fp);

return val;

不过DiskStore有个缺点，就是有可能发生两个不同的Key生成一个相同的SHA1 Hash值，这样就有可能出现丢失数据的问题。不过这种情况发生的几率比较少，所以是可以接受的。根据作者的意图，未来可能使用B+tree来替换这种高度依赖文件系统的实现方法