redis持久化之RDB和AOF

## 概述

Redis中数据存储模式有2种：cache-only和persistence：

* cache-only即只做为“缓存”服务，不持久数据，数据在服务终止后将消失，此模式下也将不存在“数据恢复”的手段，是一种安全性低/效率高/容易扩展的方式；
* persistence即为内存中的数据持久备份到磁盘文件，在服务重启后可以恢复，此模式下数据相对安全。

对于persistence持久化存储，Redis提供了两种持久化方法：

* Redis DataBase(简称RDB)
* Append-only file (简称AOF)

除了这两种方法，Redis在早起的版本还存在虚拟内存的方法，现在已经被废弃。

## RDB

当满足条件时，redis单独fork（创建）一个新的线程，会先将内存中的数据写入到一个临时文件中，待持久化过程都结束了，再用这个临时文件替换上次已经持久化好了的文件，整个过程中，主进程是不进行任何IO操作的，确保了极高的性能，此时的主进程还可以进行读写操作。rdb数据持久化的缺点是最后一次持久化的数据可能丢失，当在最后一次持久化的时间截点内还没有持久化，此时机器宕机了或出故障了，那么最后一次的数据就没有持久化到。

Fork:fork的作用是复制一个与当前进程一样的进程，新进程的所有数据（变量、环境变量、程序计数器等）数值都和原进程一致，但是是一个全新的进程，也称为原进程的子进程。

优点：使用单独子进程来进行持久化，主进程不会进行任何IO操作，保证了redis的高性能

缺点：RDB是间隔一段时间进行持久化，如果持久化之间redis发生故障，会发生数据丢失。所以这种方式更适合数据要求不严谨的时候。

这里说的这个执行数据写入到临时文件的时间点是可以通过配置来自己确定的，通过配置redis在n秒内如果超过m个key被修改这执行一次RDB操作。这个操作就类似于在这个时间点来保存一次Redis的所有数据，一次快照数据。所有这个持久化方法也通常叫做snapshots。

RDB默认开启，redis.conf中的具体配置参数如下：

|  |
| --- |
| #dbfilename：持久化数据存储在本地的文件  dbfilename dump.rdb  #dir：持久化数据存储在本地的路径，如果是在/redis/redis-3.0.6/src下启动的redis-cli，则数据会存储在当前src目录下  dir ./  ##snapshot触发的时机，save <seconds> <changes>  ##如下为900秒后，至少有一个变更操作，才会snapshot  ##对于此值的设置，需要谨慎，评估系统的变更操作密集程度  ##可以通过“save “””来关闭snapshot功能  #save时间，以下分别表示更改了1个key时间隔900s进行持久化存储；更改了10个#key300s进行存储；更改10000个key60s进行存储。  save 900 1 #每隔900秒，在这期间至少一个键发生变化时做快照  save 300 10 #每隔300秒，在这期间至少10个键发生变化时做快照  save 60 10000 #每隔60秒，在这期间至少10000个键发生变化时做快照  ##当snapshot时出现错误无法继续时，是否阻塞客户端“变更操作”，“错误”可能因为磁盘已满/磁盘故障/OS级别异常等  stop-writes-on-bgsave-error yes  ##是否启用rdb文件压缩，默认为“yes”，压缩往往意味着“额外的cpu消耗”，同时也意味这较小的文件尺寸以及较短的网络传输时间  rdbcompression yes |

snapshot触发的时机，是有“间隔时间”和“变更次数”共同决定，同时符合2个条件才会触发snapshot,否则“变更次数”会被继续累加到下一个“间隔时间”上。snapshot过程中并不阻塞客户端请求。snapshot首先将数据写入临时文件，当成功结束后，将临时文件重名为dump.rdb。

在RDB方式下，你有两种选择，一种是手动执行持久化数据命令来让redis进行一次数据快照，另一种则是根据你所配置的配置文件的策略，达到策略的某些条件时来自动持久化数据。而手动执行持久化命令，你依然有两种选择，那就是save命令和bgsave命令。

save操作在Redis主线程中工作，因此会阻塞其他请求操作，应该避免使用。bgSave则是调用Fork,产生子进程，父进程继续处理请求，不会导致父进程阻塞。

默认下，持久化到dump.rdb文件，并且在redis重启后，自动读取其中文件，据悉，通常情况下一千万的字符串类型键，1GB的快照文件，同步到内存中的 时间是20-30秒。

## AOF

Append-only file，将“操作 + 数据”以格式化指令的方式追加到操作日志文件的尾部，在append操作返回后(已经写入到文件或者即将写入)，才进行实际的数据变更，“日志文件”保存了历史所有的操作过程；当server需要数据恢复时，可以直接replay此日志文件，即可还原所有的操作过程。AOF相对可靠， AOF文件内容是字符串，非常容易阅读和解析。

优点：可以保持更高的数据完整性，如果设置追加file的时间是1s，如果redis发生故障，最多会丢失1s的数据；且如果日志写入不完整支持redis-check-aof来进行日志修复；AOF文件没被rewrite之前（文件过大时会对命令进行合并重写），可以删除其中的某些命令（比如误操作的flushall）。

缺点：AOF文件比RDB文件大，且恢复速度慢。

我们可以简单的认为AOF就是日志文件，此文件只会记录“变更操作”(例如：set/del等)，如果server中持续的大量变更操作，将会导致AOF文件非常的庞大，意味着server失效后，数据恢复的过程将会很长；事实上，一条数据经过多次变更，将会产生多条AOF记录，其实只要保存当前的状态，历史的操作记录是可以抛弃的；因为AOF持久化模式还伴生了“AOF rewrite”。

AOF的特性决定了它相对比较安全，如果你期望数据更少的丢失，那么可以采用AOF模式。如果AOF文件正在被写入时突然server失效，有可能导致文件的最后一次记录是不完整，你可以通过手工或者程序的方式去检测并修正不完整的记录，以便通过aof文件恢复能够正常；同时需要提醒，如果你的redis持久化手段中有aof，那么在server故障失效后再次启动前，需要检测aof文件的完整性。

AOF默认关闭，开启方法，修改配置文件reds.conf：appendonly yes

|  |
| --- |
| ##此选项为aof功能的开关，默认为“no”，可以通过“yes”来开启aof功能  ##只有在“yes”下，aof重写/文件同步等特性才会生效  appendonly yes  ##指定aof文件名称  appendfilename appendonly.aof  ##指定aof操作中文件同步策略，有三个合法值：always everysec no,默认为everysec  appendfsync everysec  ##在aof-rewrite期间，appendfsync是否暂缓文件同步，"no"表示“不暂缓”，“yes”表示“暂缓”，默认为“no”  no-appendfsync-on-rewrite no  ##aof文件rewrite触发的最小文件尺寸(mb,gb),只有大于此aof文件大于此尺寸是才会触发rewrite，默认“64mb”，建议“512mb”  auto-aof-rewrite-min-size 64mb  ##相对于“上一次”rewrite，本次rewrite触发时aof文件应该增长的百分比。  ##每一次rewrite之后，redis都会记录下此时“新aof”文件的大小(例如A)，那么当aof文件增长到A\*(1 + p)之后  ##触发下一次rewrite，每一次aof记录的添加，都会检测当前aof文件的尺寸。  auto-aof-rewrite-percentage 100 |

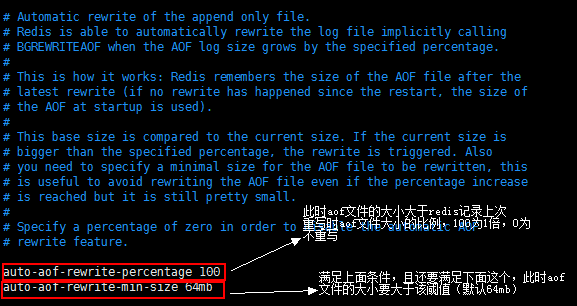
AOF是文件操作，对于变更操作比较密集的server，那么必将造成磁盘IO的负荷加重；此外linux对文件操作采取了“延迟写入”手段，即并非每次write操作都会触发实际磁盘操作，而是进入了buffer中，当buffer数据达到阀值时触发实际写入(也有其他时机)，这是linux对文件系统的优化，但是这却有可能带来隐患，如果buffer没有刷新到磁盘，此时物理机器失效(比如断电)，那么有可能导致最后一条或者多条aof记录的丢失。通过上述配置文件，可以得知redis提供了3中aof记录同步选项：

* always：每一条aof记录都立即同步到文件，这是最安全的方式，也以为更多的磁盘操作和阻塞延迟，是IO开支较大。
* everysec：每秒同步一次，性能和安全都比较中庸的方式，也是redis推荐的方式。如果遇到物理服务器故障，有可能导致最近一秒内aof记录丢失(可能为部分丢失)。
* no：redis并不直接调用文件同步，而是交给操作系统来处理，操作系统可以根据buffer填充情况/通道空闲时间等择机触发同步；这是一种普通的文件操作方式。性能较好，在物理服务器故障时，数据丢失量会因OS配置有关。

其实，我们可以选择的太少，everysec是最佳的选择。如果你非常在意每个数据都极其可靠，建议你选择一款“关系性数据库”吧。

AOF文件会不断增大（大到一定程度就发生rewrite，可以在配置文件中配置），它的大小直接影响“故障恢复”的时间,而且AOF文件中历史操作是可以丢弃的。可以说AOF rewrite操作就是“压缩”AOF文件的过程，当然redis并没有采用“基于原aof文件”来重写的方式，而是采取了类似snapshot的方式：基于copy-on-write，全量遍历内存中数据，然后逐个序列到aof文件中。因此AOF rewrite能够正确反应当前内存数据的状态，这正是我们所需要的；rewrite过程中，对于新的变更操作将仍然被写入到原AOF文件中，同时这些新的变更操作也会被redis收集起来(buffer，copy-on-write方式下，最极端的可能是所有的key都在此期间被修改，将会耗费2倍内存)，当内存数据被全部写入到新的aof文件之后，收集的新的变更操作也将会一并追加到新的aof文件中，此后将会重命名新的aof文件为appendonly.aof,此后所有的操作都将被写入新的aof文件。如果在rewrite过程中，出现故障，将不会影响原AOF文件的正常工作，只有当rewrite完成之后才会切换文件，因为rewrite过程是比较可靠的。

触发rewrite的机制：redis会记录上一次重写时aof文件的大小，默认配置是当aof文件大小是上次rewrite后大小的一倍且文件大于64mb时触发。如果启动redis后没有发生重写的，记录aof文件的大小就启动时加载的aof文件大小。



## 总结

AOF和RDB各有优缺点，这是有它们各自的特点所决定：

* AOF更加安全，可以将数据更加及时的同步到文件中，但是AOF需要较多的磁盘IO开支，AOF文件尺寸较大，文件内容恢复数度相对较慢。
* snapshot，安全性较差，它是“正常时期”数据备份以及master-slave数据同步的最佳手段，文件尺寸较小，恢复数度较快。
* RDB每次在fork子进程来执行RDB快照数据文件生成的时候，如果数据文件特别大，可能会导致对客户端提供的服务暂停数毫秒，或者甚至数秒，因为fork的时候是复制一个与当前进程一样的子进程（变量、环境变量、程序计数器等）。
* AOF日志文件的命令通过非常可读的方式进行记录，这个特性非常适合做灾难性的误删除的紧急恢复。比如某人不小心用flushall命令清空了所有数据，只要这个时候后台rewrite还没有发生，那么就可以立即拷贝AOF文件，将最后一条flushall命令给删了，然后再将该AOF文件放回去，就可以通过恢复机制，自动恢复所有数据。
* AOF开启后，支持的写QPS会比RDB支持的写QPS低，因为AOF一般会配置成每秒fsync一次日志文件，当然，每秒一次fsync，性能也还是很高的（如果fsync的频率过高，性能就会大幅度下降）。比如，如果你要保证一条数据都不丢，也是可以的，AOF的fsync设置成没写入一条数据，fsync一次，那就完蛋了，redis的QPS大降。
* 以前AOF发生过bug，就是通过AOF记录的日志，进行数据恢复的时候，没有恢复一模一样的数据出来。所以说，类似AOF这种较为复杂的基于命令日志/merge/回放的方式，比基于RDB每次持久化一份完整的数据快照文件的方式，更加脆弱一些，容易有bug。不过AOF就是为了避免rewrite过程导致的bug，因此每次rewrite并不是基于旧的指令日志进行merge的，而是基于当时内存中的数据进行指令的重新构建，这样健壮性会好很多。
* AOF唯一的比较大的缺点，其实就是做数据恢复的时候，会比较慢，还有做冷备，定期的备份，不太方便，可能要自己手写复杂的脚本去做，做冷备不太合适。

RDB和AOF可以通过配置文件来指定它们中的一种，或者同时使用它们(不建议同时使用)，或者全部禁用，在架构良好的环境中，master通常使用AOF，slave使用snapshot，主要原因是master需要首先确保数据完整性，它作为数据备份的第一选择；slave提供只读服务(目前slave只能提供读取服务)，它的主要目的就是快速响应客户端read请求；但是如果你的redis运行在网络稳定性差/物理环境糟糕情况下，建议你master和slave均采取AOF，这个在master和slave角色切换时，可以减少“人工数据备份”/“人工引导数据恢复”的时间成本；如果你的环境一切非常良好，且服务需要接收密集性的write操作，那么建议master采取snapshot，而slave采用AOF。

## 附录

<https://blog.csdn.net/canot/article/details/52886923>

<https://www.cnblogs.com/hjy9420/p/6076018.html>

https://www.jianshu.com/p/1f2cffd98e07