

## 05-内存快照：宕机后，Redis如何实现快速恢复？

你好，我是蒋德钧。

上节课，我们学习了Redis避免数据丢失的AOF方法。这个方法的好处，是每次执行只需要记录操作命令，需要持久化的数据量不大。一般而言，只要你采用的不是always的持久化策略，就不会对性能造成太大影响。

但是，也正因为记录的是操作命令，而不是实际的数据，所以，用AOF方法进行故障恢复的时候，需要逐一把操作日志都执行一遍。如果操作日志非常多，Redis就会恢复得很缓慢，影响到正常使用。这当然不是理想的结果。那么，还有没有既可以保证可靠性，还能在宕机时实现快速恢复的其他方法呢？

当然有了，这就是我们今天要一起学习的另一种持久化方法：**内存快照**。所谓内存快照，就是指内存中的数据在某一个时刻的状态记录。这就类似于照片，当你给朋友拍照时，一张照片就能把朋友一瞬间的形象完全记下来。

对Redis来说，它实现类似照片记录效果的方式，就是把某一时刻的状态以文件的形式写到磁盘上，也就是快照。这样一来，即使宕机，快照文件也不会丢失，数据的可靠性也就得到了保证。这个快照文件就称为RDB文件，其中，RDB就是Redis DataBase的缩写。

和AOF相比，RDB记录的是某一时刻的数据，并不是操作，所以，在做数据恢复时，我们可以直接把RDB文件读入内存，很快地完成恢复。听起来好像很不错，但内存快照也并不是最优选项。为什么这么说呢？

我们还要考虑两个关键问题：

- 对哪些数据做快照？这关系到快照的执行效率问题；
- 做快照时，数据还能被增删改吗？这关系到Redis是否被阻塞，能否同时正常处理请求。

这么说可能你还不太好理解，我还是拿拍照片来举例子。我们在拍照时，通常要关注两个问题：

- 如何取景？也就是说，我们打算把哪些人、哪些物拍到照片中；
- 在按快门前，要记着提醒朋友不要乱动，否则拍出来的照片就模糊了。

你看，这两个问题是不是非常重要呢？那么，接下来，我们就来具体地聊一聊。先说“取景”问题，也就是我们对哪些数据做快照。

### 给哪些内存数据做快照？

Redis的数据都在内存中，为了提供所有数据的可靠性保证，它执行的是**全量快照**，也就是说，把内存中的所有数据都记录到磁盘中，这就类似于给100个人拍合影，把每一个人都拍进照片里。这样做的好处是，一次性记录了所有数据，一个都不少。

当你给一个人拍照时，只用协调一个人就够了，但是，拍100人的大合影，却需要协调100个人的位置、状态，等等，这当然会更费时费力。同样，给内存的全量数据做快照，把它们全部写入磁盘也会花费很多时间。而且，全量数据越多，RDB文件就越大，往磁盘上写数据的时间开销就越大。

对于Redis而言，它的单线程模型就决定了，我们要尽量避免所有会阻塞主线程的操作，所以，针对任何操作，我们都会提一个灵魂之问：“它会阻塞主线程吗？”RDB文件的生成是否会阻塞主线程，这就关系到是否会降低Redis的性能。

Redis提供了两个命令来生成RDB文件，分别是save和bgsave。

- save：在主线程中执行，会导致阻塞；
- bgsave：创建一个子进程，专门用于写入RDB文件，避免了主线程的阻塞，这也是Redis RDB文件生成的默认配置。

好了，这个时候，我们就可以通过bgsave命令来执行全量快照，这既提供了数据的可靠性保证，也避免了对Redis的性能影响。

接下来，我们要关注的问题就是，在对内存数据做快照时，这些数据还能“动”吗？也就是说，这些数据还能被修改吗？这个问题非常重要，这是因为，如果数据能被修改，那就意味着Redis还能正常处理写操作。否则，所有写操作都得等到快照完了才能执行，性能一下子就降低了。

## 快照时数据能修改吗？

在给别人拍照时，一旦对方动了，那么这张照片就拍糊了，我们就需要重拍，所以我们当然希望对方保持不动。对于内存快照而言，我们也不希望数据“动”。

举个例子。我们在时刻t给内存做快照，假设内存数据量是4GB，磁盘的写入带宽是0.2GB/s，简单来说，至少需要20s ( $4/0.2 = 20$ ) 才能做完。如果在时刻t+5s时，一个还没有被写入磁盘的内存数据A，被修改成了A'，那么就会破坏快照的完整性，因为A'不是时刻t时的状态。因此，和拍照类似，我们在做快照时也不希望数据“动”，也就是不能被修改。

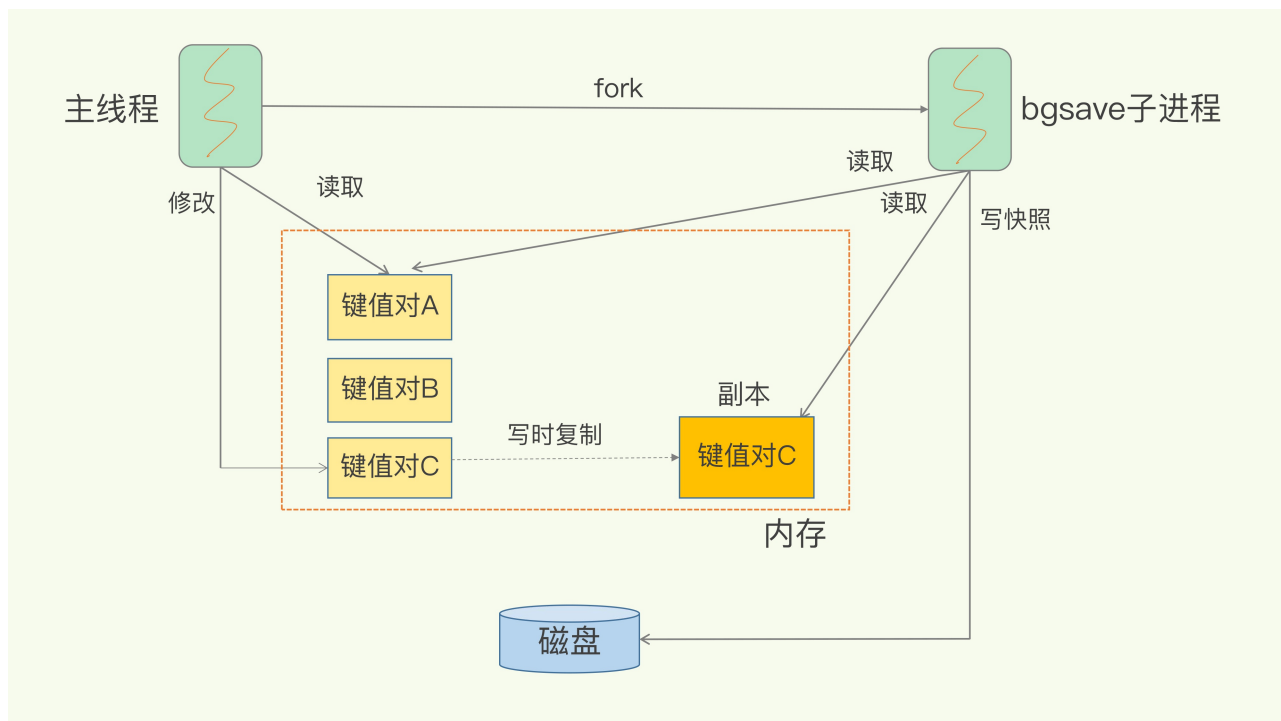
但是，如果快照执行期间数据不能被修改，是会有潜在问题的。对于刚刚的例子来说，在做快照的20s时间里，如果这4GB的数据都不能被修改，Redis就不能处理对这些数据的写操作，那无疑就会给业务服务造成巨大的影响。

你可能会想到，可以用bgsave避免阻塞啊。这里我就要说到一个常见的误区了，**避免阻塞和正常处理写操作并不是一回事**。此时，主线程的确没有阻塞，可以正常接收请求，但是，为了保证快照完整性，它只能处理读操作，因为不能修改正在执行快照的数据。

为了快照而暂停写操作，肯定是不能接受的。所以这个时候，Redis就会借助操作系统提供的写时复制技术（Copy-On-Write, COW），在执行快照的同时，正常处理写操作。

简单来说，bgsave子进程是由主线程fork生成的，可以共享主线程的所有内存数据。bgsave子进程运行后，开始读取主线程的内存数据，并把它们写入RDB文件。

此时，如果主线程对这些数据也都是读操作（例如图中的键值对A），那么，主线程和bgsave子进程相互不影响。但是，如果主线程要修改一块数据（例如图中的键值对C），那么，这块数据就会被复制一份，生成该数据的副本。然后，bgsave子进程会把这个副本数据写入RDB文件，而在这个过程中，主线程仍然可以直接修改原来的数据。



这既保证了快照的完整性，也允许主线程同时对数据进行修改，避免了对正常业务的影响。

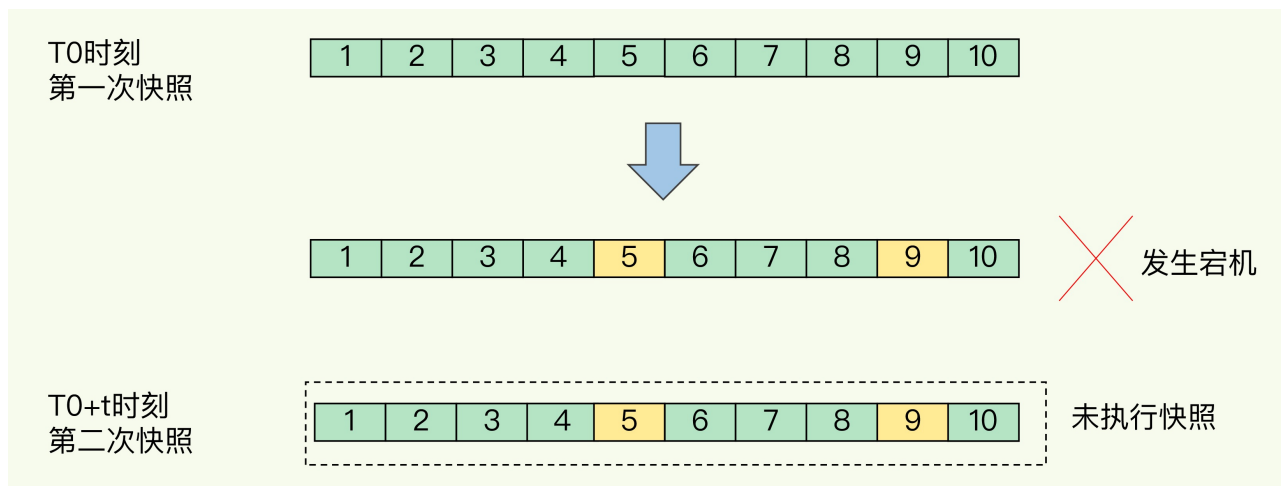
到这里，我们就解决了对“哪些数据做快照”以及“做快照时数据能否修改”这两大问题：Redis会使用bgsave对当前内存中的所有数据做快照，这个操作是子进程在后台完成的，这就允许主线程同时可以修改数据。

现在，我们再来看另一个问题：多久做一次快照？我们在拍照的时候，还有项技术叫“连拍”，可以记录人或物连续多个瞬间的状态。那么，快照也适合“连拍”吗？

## 可以每秒做一次快照吗？

对于快照来说，所谓“连拍”就是指连续地做快照。这样一来，快照的间隔时间变得很短，即使某一时刻发生宕机了，因为上一时刻快照刚执行，丢失的数据也不会太多。但是，这其中的快照间隔时间就很关键了。

如下图所示，我们先在 $T_0$ 时刻做了一次快照，然后又在 $T_0+t$ 时刻做了一次快照，在这期间，数据块5和9被修改了。如果在 $t$ 这段时间内，机器宕机了，那么，只能按照 $T_0$ 时刻的快照进行恢复。此时，数据块5和9的修改值因为没有快照记录，就无法恢复了。



所以，要想尽可能恢复数据， $t$ 值就要尽可能小， $t$ 越小，就越像“连拍”。那么， $t$ 值可以小到什么程度

呢，比如说是不是可以每秒做一次快照？毕竟，每次快照都是由bgsave子进程在后台执行，也不会阻塞主线程。

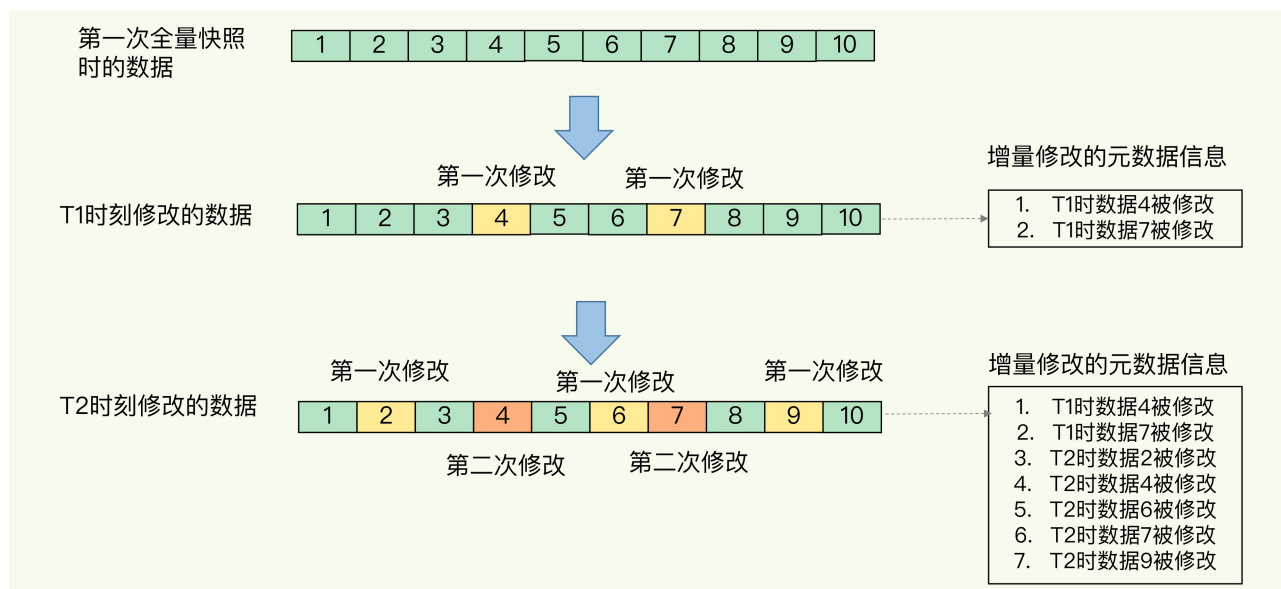
这种想法其实是错误的。虽然bgsave执行时不阻塞主线程，但是，**如果频繁地执行全量快照，也会带来两方面的开销。**

一方面，频繁将全量数据写入磁盘，会给磁盘带来很大压力，多个快照竞争有限的磁盘带宽，前一个快照还没有做完，后一个又开始做了，容易造成恶性循环。

另一方面，bgsave子进程需要通过fork操作从主线程创建出来。虽然，子进程在创建后不会再阻塞主线程，但是，fork这个创建过程本身会阻塞主线程，而且主线程的内存越大，阻塞时间越长。如果频繁fork出bgsave子进程，这就会频繁阻塞主线程了。那么，有什么其他好方法吗？

此时，我们可以做增量快照，所谓增量快照，就是指，做了一次全量快照后，后续的快照只对修改的数据进行快照记录，这样可以避免每次全量快照的开销。

在第一次做完全量快照后，T1和T2时刻如果再做快照，我们只需要将被修改的数据写入快照文件就行。但是，这么做的前提是，**我们需要记住哪些数据被修改了**。你可不要小瞧这个“记住”功能，它需要我们使用额外的元数据信息去记录哪些数据被修改了，这会带来额外的空间开销问题。如下图所示：



如果我们对每一个键值对的修改，都做个记录，那么，如果有1万个被修改的键值对，我们就需要有1万条额外的记录。而且，有的时候，键值对非常小，比如只有32字节，而记录它被修改的元数据信息，可能就需要8字节，这样的画，为了“记住”修改，引入的额外空间开销比较大。这对于内存资源宝贵的Redis来说，有些得不偿失。

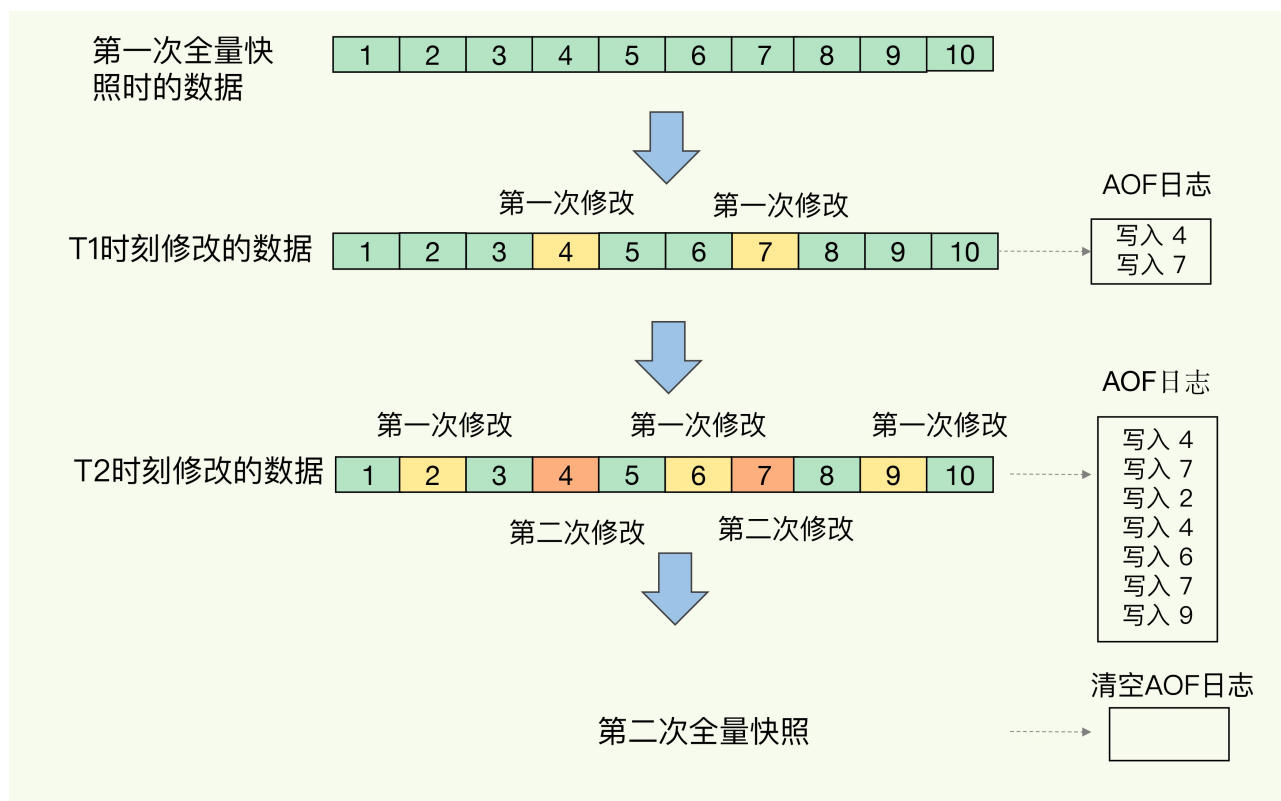
到这里，你可以发现，虽然跟AOF相比，快照的恢复速度快，但是，快照的频率不好把握，如果频率太低，两次快照间一旦宕机，就可能有比较多的数据丢失。如果频率太高，又会产生额外开销，那么，还有什么方法既能利用RDB的快速恢复，又能以较小的开销做到尽量少丢数据呢？

Redis 4.0中提出了一个**混合使用AOF日志和内存快照**的方法。简单来说，内存快照以一定的频率执行，在两次快照之间，使用AOF日志记录这期间的所有命令操作。

这样一来，快照不用很频繁地执行，这就避免了频繁fork对主线程的影响。而且，AOF日志也只用记录两次

快照间的操作，也就是说，不需要记录所有操作了，因此，就不会出现文件过大的情况了，也可以避免重写开销。

如下图所示，T1和T2时刻的修改，用AOF日志记录，等到第二次做全量快照时，就可以清空AOF日志，因为此时的修改都已经记录到快照中了，恢复时就不再用日志了。



这个方法既能享受到RDB文件快速恢复的好处，又能享受到AOF只记录操作命令的简单优势，颇有点“鱼和熊掌可以兼得”的感觉，建议你在实践中用起来。

## 小结

这节课，我们学习了Redis用于避免数据丢失的内存快照方法。这个方法的优势在于，可以快速恢复数据库，也就是只需要把RDB文件直接读入内存，这就避免了AOF需要顺序、逐一重新执行操作命令带来的低效性能问题。

不过，内存快照也有它的局限性。它拍的是一张内存的“大合影”，不可避免地会耗时耗力。虽然，Redis设计了bgsave和写时复制方式，尽可能减少了内存快照对正常读写的影响，但是，频繁快照仍然是不太能接受的。而混合使用RDB和AOF，正好可以取两者之长，避两者之短，以较小的性能开销保证数据可靠性和性能。

最后，关于AOF和RDB的选择问题，我想再给你提三点建议：

- 数据不能丢失时，内存快照和AOF的混合使用是一个很好的选择；
- 如果允许分钟级别的数据丢失，可以只使用RDB；
- 如果只用AOF，优先使用everysec的配置选项，因为它在可靠性和性能之间取了一个平衡。

## 每课一问

到这里，关于持久化我们就讲完了，这块儿内容是熟练掌握Redis的基础，建议你一定好好学习下这两节课。如果你觉得有收获，希望你能帮我分享给更多的人，帮助更多人解决持久化的问题。

- Kaito 2020-08-14 01:23:00

[illegible][illegible]

äö`redis4.0ä»¥â%ö❖ï¼(Eredis AOFçš,,é ± ❖â + ™æœ°â^ ¶ æ~ æ£ ± ä»æ · ´â❖^ï¼^è€❖â, ^ä, Šä  
 ,€š,ë~¾ä²·ç❖❖è~ ´è; ±ï¼%ï¼¼£ä½ + æ~`äö`redis4.0ä»¥â%ö❖❖ï¼(Eredisçš,, AOF é ± ❖â + ™çš,,æ  
 — ¶ â™â±±c> ´æ❖¥æšš RDB çš,,â + ⋯â®¹â + ™â°^° AOF æ- ± ä» ¶ â¼äâ¶ ´ï¼£â° + â❖❖é ± ❖çš,,ä»¥æ





l,æœæ~˘ā+™æ “ä½œā°±āœā^¶  
āè¿ç˘˘ā^æ~æœä¹¿ÿ¥é “ä»ā “é+Æçÿ¥é “é” ºā€¼ā¹Cçš,ā%~æœ-çš,āœ°āœçš  
”

- é,ææ—¶ā^» 2020-08-14 09:34:20  
è˘·é—ºèœä, ^¼Æāœ˘æ·ā^ä½¿ç” “aofā” Æā+˘˘ā~ā¿ç˘˘ S çš,æ¹æ³·æ—¶ā€™āœæ  
ä^°āœ˘äšā˘˘é+ä¿ç˘˘ S çš,æ—¶ā€™¼Ææ,˘ç©°AOFæ—¥ā¿—äœ,æ~˘āœ˘˘˘é+ä¿ç  
˘˘ S ā%œ,˘ç©°AOFæ—¥ā¿—è¿æ~˘ā˘˘˘é+ä¿ç˘˘ S āœ,˘ç©°AOFæ—¥ā¿—ā ‘¼ÿ  
ç²—ç·¥çš,æf³¼Æāœ˘äšā˘˘é+ä¿ç˘˘ S æ—¶ā€™ā° S ç” ÿæ-°çš,AOFæ—¥ā¿—¼Æā|,æœ  
æ,˘ç©°AOFæ—¥ā¿—çš,è~¼Æä¼šæššæ-°çš,æ—¥ā¿—æ,˘æ%....

- è,±ç¼°çš,é+é©— 2020-08-14 09:17:36  
èœä, ^¼Æä½ è˘¿çš,ä,»ç¿ç˘˘˘forkāç¿ç˘˘˘ä¼ä¼šé»ā¿¼ÆèœÆä,” ä,»ç¿ç˘˘˘ā+˘˘ā~è¶šā  
S é~»ā¿è¶šā˘˘äœ,èœä, ^ä~ä»¥ā+äæ·±ā˘˘¥è²è S £ä, €ä,˘forkāç¿ç˘˘˘çš,è¿ç˘˘˘ä~¼ÿ

- ç S æµ 2020-08-14 08:50:28  
èœä, ^æ˘ ‘æœ%ā+ä, ºé—ºéœ~æf³é—º¼š  
1äœä˘äšā¿ç˘˘ S æ—¶¼Æç¬ä, €æ¬ä½¿ç” “çš,æ~˘ā˘˘˘é+ä¿ç˘˘ S ¼Æä¹ä^ä½¿ç  
” “çš,éf½æ~˘äœé+ä¿ç˘˘ S ¼Æé,£ä¹è¿™ä, ºæ~Redisäšā¿ç˘˘ S æ—¶é~ºçš,æœ°ā^  
¶è¿æ~˘éœœè|æ˘ ‘ä¬ä, »äš˘é˘ç½º¼ÿ  
2äœæ—ç,¶ saveä ‘½ä»ä¼šä»ä, »ç¿ç˘˘˘é~»ā¿¼Æé,£ä, ºä»ä¹^Redisè¿è~ºç½ºè¿™ä, €  
ä, ºéœ%è¿¼ÿè¿™ä, ºèºç½ºæ~˘æœ%ä»ä¹^ä˘˘¶ä¬çš,ä¼~ç,¹ä’ Æä½¿ç” “æ°æ™~ä—¼ÿ  
3äœä˘äšāœæä¹˘äœæ—¶è¥äœæ—¶ā¼ä~ä°+AOFä’ ÆRDB¼^ä, æ~æ·ä^  
æ˘¼ä¼¿¼%¼¼Æè¿™æ·äšçš,è~æ~ä^éœ,çš,æœä¹˘äœæ—éœ%æ©ä—¼ÿä¼šä¹^Redisè  
¿ç˘˘˘ä° S ç” ÿä»ä¹^ä½±ä “¼ÿ

- æ—äœäœä, % 2020-08-14 08:07:10  
ç°æä·æ¥è~¼Æbgsave äè¿ç˘˘˘æ~ç” ±ä, »ç¿ç˘˘˘forkç” ÿæ^çš,¼Æä~ä»¥ā˘˘±ä°ä  
, »ç¿ç˘˘˘çš,æ%œæ%ā+˘˘ā~æ·æ°æœ,bgsave äè¿ç˘˘˘è¿è¿æä¿¼Æä¼ä S è˘»ä~ä, »ç  
°¿ç˘˘˘çš,ā+˘˘ā~æ·æ°æº¼Æä¹¶äššä°fä¬ā+™ä˘˘¥RDB æ¬ä»¶ä»¶,¶äœ,

èœä, ^¼Æè¿™é+ææ~ä, »ç¿ç˘˘˘è¿æ~ä, »è¿ç˘˘˘¼ÿ

ä½œèœ˘˘ä^æ 2020-08-14 08:43:37

ä, ºRedisæ~ä·ç¿ç˘˘˘ä¿¼Ææ%ä»¥Redisä, »è¿ç˘˘˘ä¼ä°±æ~ä, €ä, ºç¿ç˘˘˘¼Ææ˘ ‘ä  
, èè~ä¼ÿ S ºä, ºä, »ç¿ç˘˘˘ä°+äœ,

- æ¼«æ¥œœä° ‘ç¬ 2020-08-14 07:06:06  
æ˘ ‘æf³æœä, €ä, ºä, ºé—ºéœ~¼Ææ˘ ‘ä½œä, ºä^ä|èœ˘æf³é—º¼Æä|,æœœredisæœäšjæ  
Æ,ä°+¼Æä±ä½æœ%ä»ä¹^ç” “¼ÿèf½æœœä¿çš,ä%œœä, æ~æœœäšjè¿~ä~æ˘  
ä—¼ÿèš¼é “æœœäšjæÆ,ä°+ä¼šè+äš˘æœ%èµ·æœœäšj¼ÿè+äš˘è¿~äÿä—¼ÿ

- learn more 2020-08-14 03:51:23  
ä, ºredisæ~ä·ç¿ç˘˘˘çš,¼Ææ%ä»¥äœæ, ā¿ç” “¼Æä, €ä, ºæ% S è¿æä ‘½ä»ä¼ä, €  
ä, ºç” “æ¥bgsave¼Æä, ä¼šé~»ā¿ä, »ç¿ç˘˘˘¼ä+˘˘ā~4g¼Æä°é™˘˘äç” “2gä·|ä³¼  
Æäæā+˘˘ā~æ~ä, €ä, ºä, ‘ç·Æç,¼¼Ææ~” ā|, fork äè¿ä¿çš,æ—¶ā€™ä³¼^æœ%äèf½  
èœ—ä°½æœ%œœ%çš,ā+˘˘ā~¼Æäšä, šè~ä+™æ~” ä¼æ~ä+™ä¿è~ä° ‘¼Æé,£ä¹è¹æ¹æ°  
save çš,è˘ç½ºä³¼^æœ%äèf½ä+°ç°éœ ‘ç¹çš, bgsave¼Æææ—¶ā+˘˘ā~ä°+æ^ä, º  
ç “¶éœ˘¼ç£ç˘˘ç˘°é—è¶šā¿¼Ææ%ä»¥ä, ä¼šä½±ä “äœ,



ä, ¢ÿŸé “æœ%æ²;æœ%â ^ + æ¢¢æ£ç;®ǧŸ~,

- ä°¢â®‡â¢2B 2020-08-14 00:28:53  
â¢šRDBæœŸé— ‘æ~^â +™æ— ¶â¢â^ ¶ çš,, 2GBçš,,æ · °æ¢® 80%éf½æ~^â +™è^ · æ±, ä¹Ÿâ°±  
æ~^â ¢ æ|,è|¢â¢â^ ¶ â ‡ °æ¢¥1.6GBæ · °æ¢®i¼Œâš ä , Šæœ¬è°æ · °æ¢®2GB i¼Œâ ·²ç»¢è¾  
¾â^°3.6GBi¼Œâ»æ¢%æ “¢ä½œç³»ç»Ÿæœ¬è°«çš,,â + …â~â¢ ç” ¨ æœ°â™¨ æ%œâ%©â + …  
â~â ·²ç»¢ä , ¢âšä° + â®¹æ~ “â¢ ‘ç” ŸOOM