除了RDB持久化功能之外，Redis还提供了AOF(Append Only File)持久化功能。与RDB持久化通过保存数据库中的键值对来记录数据库状态不同，AOF持久化是通过保存Redis服务器所执行的写命令来记录数据库状态的。与RDB持久化相比，AOF持久化可能丢失的数据更少，但是AOF持久化可能会降低Redis的性能。

写人AOF文件的所有命令都是以Redis的统一请求协议格式保存的。

在表示Redis服务器的结构体redisServer中，有关AOF的成员如下：

struct redisServer **{**

**...**

*/\* AOF persistence \*/*

int aof\_state**;** */\* REDIS\_AOF\_(ON|OFF|WAIT\_REWRITE) \*/*

int aof\_fsync**;** */\* Kind of fsync() policy \*/*

char **\***aof\_filename**;** */\* Name of the AOF file \*/*

**...**

pid\_t aof\_child\_pid**;** */\* PID if rewriting process \*/*

list **\***aof\_rewrite\_buf\_blocks**;** */\* Hold changes during an AOF rewrite. \*/*

sds aof\_buf**;** */\* AOF buffer, written before entering the event loop \*/*

int aof\_fd**;** */\* File descriptor of currently selected AOF file \*/*

**...**

*/\* AOF pipes used to communicate between parent and child during rewrite. \*/*

int aof\_pipe\_write\_data\_to\_child**;**

int aof\_pipe\_read\_data\_from\_parent**;**

int aof\_pipe\_write\_ack\_to\_parent**;**

int aof\_pipe\_read\_ack\_from\_child**;**

int aof\_pipe\_write\_ack\_to\_child**;**

int aof\_pipe\_read\_ack\_from\_parent**;**

int aof\_stop\_sending\_diff**;** */\* If true stop sending accumulated diffs*

*to child process. \*/*

sds aof\_child\_diff**;** */\* AOF diff accumulator child side. \*/*

**...**

**};**

**一：AOF持久化**

AOF持久化功能的实现可以分为命令追加、文件写人、文件同步(sync)三个步骤。

1：命令追加

开启了AOF快照功能后，当Redis服务器收到客户端命令时，会调用函数feedAppendOnlyFile。该函数按照统一请求协议对命令进行编码，将编码后的内容追加到AOF缓存server.aof\_buf中。feedAppendOnlyFile代码如下：

void feedAppendOnlyFile**(**struct redisCommand **\***cmd**,** int dictid**,** robj **\*\***argv**,** int argc**)** **{**

sds buf **=** sdsempty**();**

robj **\***tmpargv**[**3**];**

*/\* The DB this command was targeting is not the same as the last command*

*\* we appended. To issue a SELECT command is needed. \*/*

**if** **(**dictid **!=** server**.**aof\_selected\_db**)** **{**

char seldb**[**64**];**

snprintf**(**seldb**,sizeof(**seldb**),**"%d"**,**dictid**);**

buf **=** sdscatprintf**(**buf**,**"\*2\r\n$6\r\nSELECT\r\n$%lu\r\n%s\r\n"**,**

**(**unsigned long**)**strlen**(**seldb**),**seldb**);**

server**.**aof\_selected\_db **=** dictid**;**

**}**

**if** **(**cmd**->**proc **==** expireCommand **||** cmd**->**proc **==** pexpireCommand **||**

cmd**->**proc **==** expireatCommand**)** **{**

*/\* Translate EXPIRE/PEXPIRE/EXPIREAT into PEXPIREAT \*/*

buf **=** catAppendOnlyExpireAtCommand**(**buf**,**cmd**,**argv**[**1**],**argv**[**2**]);**

**}** **else** **if** **(**cmd**->**proc **==** setexCommand **||** cmd**->**proc **==** psetexCommand**)** **{**

*/\* Translate SETEX/PSETEX to SET and PEXPIREAT \*/*

tmpargv**[**0**]** **=** createStringObject**(**"SET"**,**3**);**

tmpargv**[**1**]** **=** argv**[**1**];**

tmpargv**[**2**]** **=** argv**[**3**];**

buf **=** catAppendOnlyGenericCommand**(**buf**,**3**,**tmpargv**);**

decrRefCount**(**tmpargv**[**0**]);**

buf **=** catAppendOnlyExpireAtCommand**(**buf**,**cmd**,**argv**[**1**],**argv**[**2**]);**

**}** **else** **{**

*/\* All the other commands don't need translation or need the*

*\* same translation already operated in the command vector*

*\* for the replication itself. \*/*

buf **=** catAppendOnlyGenericCommand**(**buf**,**argc**,**argv**);**

**}**

*/\* Append to the AOF buffer. This will be flushed on disk just before*

*\* of re-entering the event loop, so before the client will get a*

*\* positive reply about the operation performed. \*/*

**if** **(**server**.**aof\_state **==** REDIS\_AOF\_ON**)**

server**.**aof\_buf **=** sdscatlen**(**server**.**aof\_buf**,**buf**,**sdslen**(**buf**));**

*/\* If a background append only file rewriting is in progress we want to*

*\* accumulate the differences between the child DB and the current one*

*\* in a buffer, so that when the child process will do its work we*

*\* can append the differences to the new append only file. \*/*

**if** **(**server**.**aof\_child\_pid **!=** **-**1**)**

aofRewriteBufferAppend**((**unsigned char**\*)**buf**,**sdslen**(**buf**));**

sdsfree**(**buf**);**

**}**

该函数中，首先判断本次命令的数据库索引dictid，是否与上次命令的数据库索引server.aof\_selected\_db相同，如果不同，则编码select命令；

如果命令为EXPIRE、PEXPIRE或者EXPIREAT，则调用catAppendOnlyExpireAtCommand将命令编码为PEXPIREAT命令的格式；

如果命令为setex或psetex，则先调用catAppendOnlyGenericCommand编码SET命令，然后调用catAppendOnlyExpireAtCommand编码PEXPIREAT命令；

其他命令直接用catAppendOnlyGenericCommand对命令进行编码；

如果server.aof\_state为REDIS\_AOF\_ON，则说明开启了AOF功能，将编码后的buf追加到AOF缓存server.aof\_buf中；

另外，如果server.aof\_child\_pid不是-1，说明有子进程在进行AOF重写，则调用aofRewriteBufferAppend将编码后的buf追加到AOF重写缓存server.aof\_rewrite\_buf\_blocks中。

2：文件写人、文件同步

为了提高文件的写入效率，在现代操作系统中，当用户调用write函数将数据写入到文件描述符后，操作系统通常会将写入数据暂时保存在一个内存缓冲区里面，等到缓冲区的空间被填满、或者超过了指定的时限之后，操作系统才真正地将缓冲区中的数据写入到磁盘里面。

这种做法虽然提高了效率，但也为写入数据带来了安全问题，如果计算机发生宕机，那么保存在内存缓冲区里面的写入数据将会丢失。

为此，操作系统提供了fsync同步函数，可以手动让操作系统立即将缓冲区中的数据写入到硬盘里面，从而确保写入数据的安全性。

Redis服务器的主循环中，每隔一段时间就会将AOF缓存server.aof\_buf中的内容写入到AOF文件中。并且根据同步策略的不同，而选择不同的时机进行fsync。同步策略通过配置文件中的appendfsync选项设置，总共有三种同步策略，分别是：

a：appendfsync no

不执行fsync操作，完全交由操作系统进行同步。这种方式是最快的，但也是最不安全的。

b：appendfsync always

每次调用write将AOF缓存server.aof\_buf中的内容写入到AOF文件时，立即调用fsync函数。这种方式是最安全的，却也是最慢的。

c：appendfsync everysec

每隔1秒钟进行一次fsync操作，这是一种对速度和安全性进行折中的方法。如果用户没有设置appendfsync选项的值，则使用everysec作为选项默认值。

将AOF缓存server.aof\_buf中的内容写入到AOF文件中。并且根据同步策略的不同，而选择不同的时机进行fsync。这都是在函数flushAppendOnlyFile中实现的，其代码如下：

void flushAppendOnlyFile**(**int force**)** **{**

ssize\_t nwritten**;**

int sync\_in\_progress **=** 0**;**

mstime\_t latency**;**

**if** **(**sdslen**(**server**.**aof\_buf**)** **==** 0**)** **return;**

**if** **(**server**.**aof\_fsync **==** AOF\_FSYNC\_EVERYSEC**)**

sync\_in\_progress **=** bioPendingJobsOfType**(**REDIS\_BIO\_AOF\_FSYNC**)** **!=** 0**;**

**if** **(**server**.**aof\_fsync **==** AOF\_FSYNC\_EVERYSEC **&&** **!**force**)** **{**

*/\* With this append fsync policy we do background fsyncing.*

*\* If the fsync is still in progress we can try to delay*

*\* the write for a couple of seconds. \*/*

**if** **(**sync\_in\_progress**)** **{**

**if** **(**server**.**aof\_flush\_postponed\_start **==** 0**)** **{**

*/\* No previous write postponing, remember that we are*

*\* postponing the flush and return. \*/*

server**.**aof\_flush\_postponed\_start **=** server**.**unixtime**;**

**return;**

**}** **else** **if** **(**server**.**unixtime **-** server**.**aof\_flush\_postponed\_start **<** 2**)** **{**

*/\* We were already waiting for fsync to finish, but for less*

*\* than two seconds this is still ok. Postpone again. \*/*

**return;**

**}**

*/\* Otherwise fall trough, and go write since we can't wait*

*\* over two seconds. \*/*

server**.**aof\_delayed\_fsync**++;**

redisLog**(**REDIS\_NOTICE**,**"Asynchronous AOF fsync is taking too long (disk is busy?). Writing the AOF buffer without waiting for fsync to complete, this may slow down Redis."**);**

**}**

**}**

*/\* We want to perform a single write. This should be guaranteed atomic*

*\* at least if the filesystem we are writing is a real physical one.*

*\* While this will save us against the server being killed I don't think*

*\* there is much to do about the whole server stopping for power problems*

*\* or alike \*/*

latencyStartMonitor**(**latency**);**

nwritten **=** write**(**server**.**aof\_fd**,**server**.**aof\_buf**,**sdslen**(**server**.**aof\_buf**));**

latencyEndMonitor**(**latency**);**

*/\* We want to capture different events for delayed writes:*

*\* when the delay happens with a pending fsync, or with a saving child*

*\* active, and when the above two conditions are missing.*

*\* We also use an additional event name to save all samples which is*

*\* useful for graphing / monitoring purposes. \*/*

**if** **(**sync\_in\_progress**)** **{**

latencyAddSampleIfNeeded**(**"aof-write-pending-fsync"**,**latency**);**

**}** **else** **if** **(**server**.**aof\_child\_pid **!=** **-**1 **||** server**.**rdb\_child\_pid **!=** **-**1**)** **{**

latencyAddSampleIfNeeded**(**"aof-write-active-child"**,**latency**);**

**}** **else** **{**

latencyAddSampleIfNeeded**(**"aof-write-alone"**,**latency**);**

**}**

latencyAddSampleIfNeeded**(**"aof-write"**,**latency**);**

*/\* We performed the write so reset the postponed flush sentinel to zero. \*/*

server**.**aof\_flush\_postponed\_start **=** 0**;**

**if** **(**nwritten **!=** **(**signed**)**sdslen**(**server**.**aof\_buf**))** **{**

static time\_t last\_write\_error\_log **=** 0**;**

int can\_log **=** 0**;**

*/\* Limit logging rate to 1 line per AOF\_WRITE\_LOG\_ERROR\_RATE seconds. \*/*

**if** **((**server**.**unixtime **-** last\_write\_error\_log**)** **>** AOF\_WRITE\_LOG\_ERROR\_RATE**)** **{**

can\_log **=** 1**;**

last\_write\_error\_log **=** server**.**unixtime**;**

**}**

*/\* Log the AOF write error and record the error code. \*/*

**if** **(**nwritten **==** **-**1**)** **{**

**if** **(**can\_log**)** **{**

redisLog**(**REDIS\_WARNING**,**"Error writing to the AOF file: %s"**,**

strerror**(**errno**));**

server**.**aof\_last\_write\_errno **=** errno**;**

**}**

**}** **else** **{**

**if** **(**can\_log**)** **{**

redisLog**(**REDIS\_WARNING**,**"Short write while writing to "

"the AOF file: (nwritten=%lld, "

"expected=%lld)"**,**

**(**long long**)**nwritten**,**

**(**long long**)**sdslen**(**server**.**aof\_buf**));**

**}**

**if** **(**ftruncate**(**server**.**aof\_fd**,** server**.**aof\_current\_size**)** **==** **-**1**)** **{**

**if** **(**can\_log**)** **{**

redisLog**(**REDIS\_WARNING**,** "Could not remove short write "

"from the append-only file. Redis may refuse "

"to load the AOF the next time it starts. "

"ftruncate: %s"**,** strerror**(**errno**));**

**}**

**}** **else** **{**

*/\* If the ftruncate() succeeded we can set nwritten to*

*\* -1 since there is no longer partial data into the AOF. \*/*

nwritten **=** **-**1**;**

**}**

server**.**aof\_last\_write\_errno **=** ENOSPC**;**

**}**

*/\* Handle the AOF write error. \*/*

**if** **(**server**.**aof\_fsync **==** AOF\_FSYNC\_ALWAYS**)** **{**

*/\* We can't recover when the fsync policy is ALWAYS since the*

*\* reply for the client is already in the output buffers, and we*

*\* have the contract with the user that on acknowledged write data*

*\* is synced on disk. \*/*

redisLog**(**REDIS\_WARNING**,**"Can't recover from AOF write error when the AOF fsync policy is 'always'. Exiting..."**);**

exit**(**1**);**

**}** **else** **{**

*/\* Recover from failed write leaving data into the buffer. However*

*\* set an error to stop accepting writes as long as the error*

*\* condition is not cleared. \*/*

server**.**aof\_last\_write\_status **=** REDIS\_ERR**;**

*/\* Trim the sds buffer if there was a partial write, and there*

*\* was no way to undo it with ftruncate(2). \*/*

**if** **(**nwritten **>** 0**)** **{**

server**.**aof\_current\_size **+=** nwritten**;**

sdsrange**(**server**.**aof\_buf**,**nwritten**,-**1**);**

**}**

**return;** */\* We'll try again on the next call... \*/*

**}**

**}** **else** **{**

*/\* Successful write(2). If AOF was in error state, restore the*

*\* OK state and log the event. \*/*

**if** **(**server**.**aof\_last\_write\_status **==** REDIS\_ERR**)** **{**

redisLog**(**REDIS\_WARNING**,**

"AOF write error looks solved, Redis can write again."**);**

server**.**aof\_last\_write\_status **=** REDIS\_OK**;**

**}**

**}**

server**.**aof\_current\_size **+=** nwritten**;**

*/\* Re-use AOF buffer when it is small enough. The maximum comes from the*

*\* arena size of 4k minus some overhead (but is otherwise arbitrary). \*/*

**if** **((**sdslen**(**server**.**aof\_buf**)+**sdsavail**(**server**.**aof\_buf**))** **<** 4000**)** **{**

sdsclear**(**server**.**aof\_buf**);**

**}** **else** **{**

sdsfree**(**server**.**aof\_buf**);**

server**.**aof\_buf **=** sdsempty**();**

**}**

*/\* Don't fsync if no-appendfsync-on-rewrite is set to yes and there are*

*\* children doing I/O in the background. \*/*

**if** **(**server**.**aof\_no\_fsync\_on\_rewrite **&&**

**(**server**.**aof\_child\_pid **!=** **-**1 **||** server**.**rdb\_child\_pid **!=** **-**1**))**

**return;**

*/\* Perform the fsync if needed. \*/*

**if** **(**server**.**aof\_fsync **==** AOF\_FSYNC\_ALWAYS**)** **{**

*/\* aof\_fsync is defined as fdatasync() for Linux in order to avoid*

*\* flushing metadata. \*/*

latencyStartMonitor**(**latency**);**

aof\_fsync**(**server**.**aof\_fd**);** */\* Let's try to get this data on the disk \*/*

latencyEndMonitor**(**latency**);**

latencyAddSampleIfNeeded**(**"aof-fsync-always"**,**latency**);**

server**.**aof\_last\_fsync **=** server**.**unixtime**;**

**}** **else** **if** **((**server**.**aof\_fsync **==** AOF\_FSYNC\_EVERYSEC **&&**

server**.**unixtime **>** server**.**aof\_last\_fsync**))** **{**

**if** **(!**sync\_in\_progress**)** aof\_background\_fsync**(**server**.**aof\_fd**);**

server**.**aof\_last\_fsync **=** server**.**unixtime**;**

**}**

**}**

如果参数force置为0，并且fsync策略设置为everysec，并且有后台线程尚在进行fsync操作，因为Linux的write操作会被后台的fsync阻塞，因此需要延迟write操作。这种情况下，只需记住尚有缓存需要write，后续在serverCron中再次调用flushAppendOnlyFile函数时再进行write操作。如果force置为1，则不管是否有后台在fsync，都会进行write操作。

首先，如果server.aof\_fsync为AOF\_FSYNC\_EVERYSEC，则查看是否有其他fsync任务正在执行，有则sync\_in\_progress为1，否则sync\_in\_progress为0；

如果server.aof\_fsync为AOF\_FSYNC\_EVERYSEC，并且参数force为0，并且后台有fsync任务正在执行，则需要延迟write操作，延迟策略是：

a：若server.aof\_flush\_postponed\_start为0，说明这是首次推迟write操作，将当前时间戳记录到server.aof\_flush\_postponed\_start中，然后直接返回；

b：若server.unixtime - server.aof\_flush\_postponed\_start < 2，说明上次已经推迟了write操作，但是上次推迟时间距当前时间在2s以内，直接返回；

c：不满足以上条件，说明上次推迟时间已经超过2s，则server.aof\_delayed\_fsync++，并且记录日志，不再等待fsync完成，下面直接开始进行写操作；

接下来进行单次写操作，调用write将server.aof\_buf所有内容写入到server.aof\_fd中。如果写入字节数不等于server.aof\_buf总长度，则根据不同的情况写入不同的错误信息到日志中，并且，如果写入了部分数据，则调用ftruncate将这部分数据删除；

写入失败的情况下，如果server.aof\_fsync为AOF\_FSYNC\_ALWAYS，说明已经向客户端承诺数据必须同步到磁盘中，这种情况下，写入失败直接exit；

如果server.aof\_fsync不是AOF\_FSYNC\_ALWAYS，并且之前ftruncate失败的话，则将写入的字节数增加到当前AOF文件长度server.aof\_current\_size中，然后截取server.aof\_buf为未写入的部分，然后返回，等待下次写入；

写入成功，则将写入字节数增加到server.aof\_current\_size中，然后重置缓存server.aof\_buf；

如果server.aof\_fsync为AOF\_FSYNC\_ALWAYS，则调用aof\_fsync确保数据确实写入磁盘，并且记录延迟时间；

如果server.aof\_fsync为AOF\_FSYNC\_EVERYSEC，并且server.unixtime大于server.aof\_last\_fsync，并且当前没有fsync任务，则调用aof\_background\_fsync增加后台fsync任务；最后更新server.aof\_last\_fsync为server.unixtime

**二：加载AOF文件**

Redis服务器启动时，如果AOF功能开启的话，则需要根据AOF文件的内容恢复到数据中。

Redis加载AOF文件的方式非常巧妙，因为AOF中记录的是统一请求协议格式的客户端命令，因此Redis创建一个不带网络连接的伪客户端，通过伪客户端逐条执行AOF中的命令来恢复数据。主要实现是在函数loadAppendOnlyFile中，代码如下：

int loadAppendOnlyFile**(**char **\***filename**)** **{**

struct redisClient **\***fakeClient**;**

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

struct redis\_stat sb**;**

int old\_aof\_state **=** server**.**aof\_state**;**

long loops **=** 0**;**

off\_t valid\_up\_to **=** 0**;** */\* Offset of the latest well-formed command loaded. \*/*

**if** **(**fp **&&** redis\_fstat**(**fileno**(**fp**),&**sb**)** **!=** **-**1 **&&** sb**.**st\_size **==** 0**)** **{**

server**.**aof\_current\_size **=** 0**;**

fclose**(**fp**);**

**return** REDIS\_ERR**;**

**}**

**if** **(**fp **==** **NULL)** **{**

redisLog**(**REDIS\_WARNING**,**"Fatal error: can't open the append log file for reading: %s"**,**strerror**(**errno**));**

exit**(**1**);**

**}**

*/\* Temporarily disable AOF, to prevent EXEC from feeding a MULTI*

*\* to the same file we're about to read. \*/*

server**.**aof\_state **=** REDIS\_AOF\_OFF**;**

fakeClient **=** createFakeClient**();**

startLoading**(**fp**);**

**while(**1**)** **{**

int argc**,** j**;**

unsigned long len**;**

robj **\*\***argv**;**

char buf**[**128**];**

sds argsds**;**

struct redisCommand **\***cmd**;**

*/\* Serve the clients from time to time \*/*

**if** **(!(**loops**++** **%** 1000**))** **{**

loadingProgress**(**ftello**(**fp**));**

processEventsWhileBlocked**();**

**}**

**if** **(**fgets**(**buf**,sizeof(**buf**),**fp**)** **==** **NULL)** **{**

**if** **(**feof**(**fp**))**

**break;**

**else**

**goto** readerr**;**

**}**

**if** **(**buf**[**0**]** **!=** '\*'**)** **goto** fmterr**;**

**if** **(**buf**[**1**]** **==** '\0'**)** **goto** readerr**;**

argc **=** atoi**(**buf**+**1**);**

**if** **(**argc **<** 1**)** **goto** fmterr**;**

argv **=** zmalloc**(sizeof(**robj**\*)\***argc**);**

fakeClient**->**argc **=** argc**;**

fakeClient**->**argv **=** argv**;**

**for** **(**j **=** 0**;** j **<** argc**;** j**++)** **{**

**if** **(**fgets**(**buf**,sizeof(**buf**),**fp**)** **==** **NULL)** **{**

fakeClient**->**argc **=** j**;** */\* Free up to j-1. \*/*

freeFakeClientArgv**(**fakeClient**);**

**goto** readerr**;**

**}**

**if** **(**buf**[**0**]** **!=** '$'**)** **goto** fmterr**;**

len **=** strtol**(**buf**+**1**,NULL,**10**);**

argsds **=** sdsnewlen**(NULL,**len**);**

**if** **(**len **&&** fread**(**argsds**,**len**,**1**,**fp**)** **==** 0**)** **{**

sdsfree**(**argsds**);**

fakeClient**->**argc **=** j**;** */\* Free up to j-1. \*/*

freeFakeClientArgv**(**fakeClient**);**

**goto** readerr**;**

**}**

argv**[**j**]** **=** createObject**(**REDIS\_STRING**,**argsds**);**

**if** **(**fread**(**buf**,**2**,**1**,**fp**)** **==** 0**)** **{**

fakeClient**->**argc **=** j**+**1**;** */\* Free up to j. \*/*

freeFakeClientArgv**(**fakeClient**);**

**goto** readerr**;** */\* discard CRLF \*/*

**}**

**}**

*/\* Command lookup \*/*

cmd **=** lookupCommand**(**argv**[**0**]->**ptr**);**

**if** **(!**cmd**)** **{**

redisLog**(**REDIS\_WARNING**,**"Unknown command '%s' reading the append only file"**,** **(**char**\*)**argv**[**0**]->**ptr**);**

exit**(**1**);**

**}**

*/\* Run the command in the context of a fake client \*/*

cmd**->**proc**(**fakeClient**);**

*/\* The fake client should not have a reply \*/*

redisAssert**(**fakeClient**->**bufpos **==** 0 **&&** listLength**(**fakeClient**->**reply**)** **==** 0**);**

*/\* The fake client should never get blocked \*/*

redisAssert**((**fakeClient**->**flags **&** REDIS\_BLOCKED**)** **==** 0**);**

*/\* Clean up. Command code may have changed argv/argc so we use the*

*\* argv/argc of the client instead of the local variables. \*/*

freeFakeClientArgv**(**fakeClient**);**

**if** **(**server**.**aof\_load\_truncated**)** valid\_up\_to **=** ftello**(**fp**);**

**}**

*/\* This point can only be reached when EOF is reached without errors.*

*\* If the client is in the middle of a MULTI/EXEC, log error and quit. \*/*

**if** **(**fakeClient**->**flags **&** REDIS\_MULTI**)** **goto** uxeof**;**

loaded\_ok**:** */\* DB loaded, cleanup and return REDIS\_OK to the caller. \*/*

fclose**(**fp**);**

freeFakeClient**(**fakeClient**);**

server**.**aof\_state **=** old\_aof\_state**;**

stopLoading**();**

aofUpdateCurrentSize**();**

server**.**aof\_rewrite\_base\_size **=** server**.**aof\_current\_size**;**

**return** REDIS\_OK**;**

readerr**:** */\* Read error. If feof(fp) is true, fall through to unexpected EOF. \*/*

**if** **(!**feof**(**fp**))** **{**

redisLog**(**REDIS\_WARNING**,**"Unrecoverable error reading the append only file: %s"**,** strerror**(**errno**));**

exit**(**1**);**

**}**

uxeof**:** */\* Unexpected AOF end of file. \*/*

**if** **(**server**.**aof\_load\_truncated**)** **{**

redisLog**(**REDIS\_WARNING**,**"!!! Warning: short read while loading the AOF file !!!"**);**

redisLog**(**REDIS\_WARNING**,**"!!! Truncating the AOF at offset %llu !!!"**,**

**(**unsigned long long**)** valid\_up\_to**);**

**if** **(**valid\_up\_to **==** **-**1 **||** truncate**(**filename**,**valid\_up\_to**)** **==** **-**1**)** **{**

**if** **(**valid\_up\_to **==** **-**1**)** **{**

redisLog**(**REDIS\_WARNING**,**"Last valid command offset is invalid"**);**

**}** **else** **{**

redisLog**(**REDIS\_WARNING**,**"Error truncating the AOF file: %s"**,**

strerror**(**errno**));**

**}**

**}** **else** **{**

*/\* Make sure the AOF file descriptor points to the end of the*

*\* file after the truncate call. \*/*

**if** **(**server**.**aof\_fd **!=** **-**1 **&&** lseek**(**server**.**aof\_fd**,**0**,**SEEK\_END**)** **==** **-**1**)** **{**

redisLog**(**REDIS\_WARNING**,**"Can't seek the end of the AOF file: %s"**,**

strerror**(**errno**));**

**}** **else** **{**

redisLog**(**REDIS\_WARNING**,**

"AOF loaded anyway because aof-load-truncated is enabled"**);**

**goto** loaded\_ok**;**

**}**

**}**

**}**

redisLog**(**REDIS\_WARNING**,**"Unexpected end of file reading the append only file. You can: 1) Make a backup of your AOF file, then use ./redis-check-aof --fix <filename>. 2) Alternatively you can set the 'aof-load-truncated' configuration option to yes and restart the server."**);**

exit**(**1**);**

fmterr**:** */\* Format error. \*/*

redisLog**(**REDIS\_WARNING**,**"Bad file format reading the append only file: make a backup of your AOF file, then use ./redis-check-aof --fix <filename>"**);**

exit**(**1**);**

**}**

函数中，置server.aof\_state为REDIS\_AOF\_OFF，防止向该filename中写入新的AOF数据；因此，Redis在加载AOF文件时，AOF功能是关闭的。

创建伪客户端，读取文件，根据统一请求协议的格式，将AOF文件内容恢复成客户端命令，然后调用lookupCommand查找命令处理函数，然后执行该函数以恢复数据；然后释放客户端命令；如果server.aof\_load\_truncated为True，则记录已读取的字节数到valid\_up\_to中，表示到valid\_up\_to个字节为止，AOF文件还是一切正常的；一直读下去，直到文件末尾，或者格式出错等；

如果读到文件末尾并未出错，则关闭文件，释放伪客户端，恢复状态，调用stopLoading标记停止load过程，调用aofUpdateCurrentSize更新server.aof\_current\_size为AOF文件长度；返回REDIS\_OK。

如果读取文件时发生read错误，若还没读到文件末尾，则直接记录错误日志，然后退出；否则，如果读取中，应该还有数据的时候，却读到了文件末尾，则：

如果server.aof\_load\_truncated为True，则调用truncate将AOF文件截断为valid\_up\_to，如果valid\_up\_to为-1，或者截断失败，则记录错误日志，然后exit退出；否则，截断成功，使描述符server.aof\_fd的状态指向文件末尾，然后当做加载AOF成功处理；

其他情况，一律记录日志，exit退出。

**三：AOF重写**

AOF持久化是通过保存执行的写命令来记录数据库状态的，随着服务器运行，AOF文件中的冗余内容会越来越多，文件的体积也会越来越大。

为了解决AOF文件体积膨胀的问题，Redis提供了AOF文件重写功能。通过该功能，Redis服务器可以创建一个新的AOF文件来替代现有的AOF文件，新旧两个AOF文件所保存的数据库状态相同，但新AOF文件不会包含任何浪费空间的冗余命令，所以新AOF文件的体积通常会比旧AOF文件的体积要小得多。

AOF文件重写并不需要对现有的AOF文件进行任何读取、分析或者写人操作，这个功能是通过读取服务器当前的数据库状态来实现的。

1：后台AOF重写

为了防止重写AOF阻塞服务器，该过程在后台进行的。方法就是Redis服务器fork一个子进程，由子进程进行AOF的重写。任何时刻只能有一个子进程在进行AOF重写。

注意，调用fork时，子进程的内存与父进程（Redis服务器）是一模一样的，因此子进程中的数据库状态也就是服务器此刻的状态。而此时父进程继续接受来自客户端的命令，这就会产生新的数据。

为了使最终的AOF文件与数据库状态尽可能的一致，父进程处理客户端新到来的命令时，会将该命令缓存到server.aof\_rewrite\_buf\_blocks中，并在合适的时机将server.aof\_rewrite\_buf\_blocks中的内容，通过管道发送给子进程。这就是在之前介绍过的命令追加的实现函数feedAppendOnlyFile最后一步所进行的操作。

父进程需要把缓存的新数据发给子进程，这就需要创建一系列用于父子进程间通信的管道，总共有3个管道：

管道1用于父进程向子进程发送缓存的新数据。子进程在重写AOF时，定期从该管道中读取数据并缓存起来，并在最后将缓存的数据写入重写的AOF文件；

管道2负责子进程向父进程发送结束信号。由于父进程在不断的接收客户端命令，但是子进程不能无休止的等待父进程的数据，因此，子进程在遍历完数据库所有数据之后，从管道1中执行一段时间的读取操作后，就会向管道2中发送一个"!"，父进程收到子进程的"!"后，就会置server.aof\_stop\_sending\_diff为1，表示不再向子进程发送缓存的数据了；

管道3负责父进程向子进程发送应答信号。父进程收到子进程的"!"后，会通过该管道也向子进程应答一个"!"，表示已收到了停止信号。

子进程执行重写AOF的过程很简单，就是根据fork时刻的数据库状态，依次轮训Redis的server.dbnum个数据库，遍历每个数据库中的每个键值对数据，进行AOF重写工作。每当重写了10k的数据后，就会从管道1中读取服务器（父进程）缓存的新数据，并缓存到server.aof\_child\_diff中。

子进程遍历完所有数据后，再次从管道1中读取服务器（父进程）缓存的新数据，读取一段时间后，向管道2中发送字符"!"，以使父进程停止发送缓存的新数据；然后从管道3中读取父进程的回应。最后，将server.aof\_child\_diff中的内容写入重写的AOF文件，最终完成了AOF重写的主要过程。

在子进程中调用rewriteAppendOnlyFile函数进行AOF重写，其代码如下：

int rewriteAppendOnlyFile**(**char **\***filename**)** **{**

dictIterator **\***di **=** **NULL;**

dictEntry **\***de**;**

rio aof**;**

FILE **\***fp**;**

char tmpfile**[**256**];**

int j**;**

long long now **=** mstime**();**

char byte**;**

size\_t processed **=** 0**;**

*/\* Note that we have to use a different temp name here compared to the*

*\* one used by rewriteAppendOnlyFileBackground() function. \*/*

snprintf**(**tmpfile**,**256**,**"temp-rewriteaof-%d.aof"**,** **(**int**)** getpid**());**

fp **=** fopen**(**tmpfile**,**"w"**);**

**if** **(!**fp**)** **{**

redisLog**(**REDIS\_WARNING**,** "Opening the temp file for AOF rewrite in rewriteAppendOnlyFile(): %s"**,** strerror**(**errno**));**

**return** REDIS\_ERR**;**

**}**

server**.**aof\_child\_diff **=** sdsempty**();**

rioInitWithFile**(&**aof**,**fp**);**

**if** **(**server**.**aof\_rewrite\_incremental\_fsync**)**

rioSetAutoSync**(&**aof**,**REDIS\_AOF\_AUTOSYNC\_BYTES**);**

**for** **(**j **=** 0**;** j **<** server**.**dbnum**;** j**++)** **{**

char selectcmd**[]** **=** "\*2\r\n$6\r\nSELECT\r\n"**;**

redisDb **\***db **=** server**.**db**+**j**;**

dict **\***d **=** db**->**dict**;**

**if** **(**dictSize**(**d**)** **==** 0**)** **continue;**

di **=** dictGetSafeIterator**(**d**);**

**if** **(!**di**)** **{**

fclose**(**fp**);**

**return** REDIS\_ERR**;**

**}**

*/\* SELECT the new DB \*/*

**if** **(**rioWrite**(&**aof**,**selectcmd**,sizeof(**selectcmd**)-**1**)** **==** 0**)** **goto** werr**;**

**if** **(**rioWriteBulkLongLong**(&**aof**,**j**)** **==** 0**)** **goto** werr**;**

*/\* Iterate this DB writing every entry \*/*

**while((**de **=** dictNext**(**di**))** **!=** **NULL)** **{**

sds keystr**;**

robj key**,** **\***o**;**

long long expiretime**;**

keystr **=** dictGetKey**(**de**);**

o **=** dictGetVal**(**de**);**

initStaticStringObject**(**key**,**keystr**);**

expiretime **=** getExpire**(**db**,&**key**);**

*/\* If this key is already expired skip it \*/*

**if** **(**expiretime **!=** **-**1 **&&** expiretime **<** now**)** **continue;**

*/\* Save the key and associated value \*/*

**if** **(**o**->**type **==** REDIS\_STRING**)** **{**

*/\* Emit a SET command \*/*

char cmd**[]=**"\*3\r\n$3\r\nSET\r\n"**;**

**if** **(**rioWrite**(&**aof**,**cmd**,sizeof(**cmd**)-**1**)** **==** 0**)** **goto** werr**;**

*/\* Key and value \*/*

**if** **(**rioWriteBulkObject**(&**aof**,&**key**)** **==** 0**)** **goto** werr**;**

**if** **(**rioWriteBulkObject**(&**aof**,**o**)** **==** 0**)** **goto** werr**;**

**}** **else** **if** **(**o**->**type **==** REDIS\_LIST**)** **{**

**if** **(**rewriteListObject**(&**aof**,&**key**,**o**)** **==** 0**)** **goto** werr**;**

**}** **else** **if** **(**o**->**type **==** REDIS\_SET**)** **{**

**if** **(**rewriteSetObject**(&**aof**,&**key**,**o**)** **==** 0**)** **goto** werr**;**

**}** **else** **if** **(**o**->**type **==** REDIS\_ZSET**)** **{**

**if** **(**rewriteSortedSetObject**(&**aof**,&**key**,**o**)** **==** 0**)** **goto** werr**;**

**}** **else** **if** **(**o**->**type **==** REDIS\_HASH**)** **{**

**if** **(**rewriteHashObject**(&**aof**,&**key**,**o**)** **==** 0**)** **goto** werr**;**

**}** **else** **{**

redisPanic**(**"Unknown object type"**);**

**}**

*/\* Save the expire time \*/*

**if** **(**expiretime **!=** **-**1**)** **{**

char cmd**[]=**"\*3\r\n$9\r\nPEXPIREAT\r\n"**;**

**if** **(**rioWrite**(&**aof**,**cmd**,sizeof(**cmd**)-**1**)** **==** 0**)** **goto** werr**;**

**if** **(**rioWriteBulkObject**(&**aof**,&**key**)** **==** 0**)** **goto** werr**;**

**if** **(**rioWriteBulkLongLong**(&**aof**,**expiretime**)** **==** 0**)** **goto** werr**;**

**}**

*/\* Read some diff from the parent process from time to time. \*/*

**if** **(**aof**.**processed\_bytes **>** processed**+**1024**\***10**)** **{**

processed **=** aof**.**processed\_bytes**;**

aofReadDiffFromParent**();**

**}**

**}**

dictReleaseIterator**(**di**);**

di **=** **NULL;**

**}**

*/\* Do an initial slow fsync here while the parent is still sending*

*\* data, in order to make the next final fsync faster. \*/*

**if** **(**fflush**(**fp**)** **==** EOF**)** **goto** werr**;**

**if** **(**fsync**(**fileno**(**fp**))** **==** **-**1**)** **goto** werr**;**

*/\* Read again a few times to get more data from the parent.*

*\* We can't read forever (the server may receive data from clients*

*\* faster than it is able to send data to the child), so we try to read*

*\* some more data in a loop as soon as there is a good chance more data*

*\* will come. If it looks like we are wasting time, we abort (this*

*\* happens after 20 ms without new data). \*/*

int nodata **=** 0**;**

mstime\_t start **=** mstime**();**

**while(**mstime**()-**start **<** 1000 **&&** nodata **<** 20**)** **{**

**if** **(**aeWait**(**server**.**aof\_pipe\_read\_data\_from\_parent**,** AE\_READABLE**,** 1**)** **<=** 0**)**

**{**

nodata**++;**

**continue;**

**}**

nodata **=** 0**;** */\* Start counting from zero, we stop on N \*contiguous\**

*timeouts. \*/*

aofReadDiffFromParent**();**

**}**

*/\* Ask the master to stop sending diffs. \*/*

**if** **(**write**(**server**.**aof\_pipe\_write\_ack\_to\_parent**,**"!"**,**1**)** **!=** 1**)** **goto** werr**;**

**if** **(**anetNonBlock**(NULL,**server**.**aof\_pipe\_read\_ack\_from\_parent**)** **!=** ANET\_OK**)**

**goto** werr**;**

*/\* We read the ACK from the server using a 10 seconds timeout. Normally*

*\* it should reply ASAP, but just in case we lose its reply, we are sure*

*\* the child will eventually get terminated. \*/*

**if** **(**syncRead**(**server**.**aof\_pipe\_read\_ack\_from\_parent**,&**byte**,**1**,**5000**)** **!=** 1 **||**

byte **!=** '!'**)** **goto** werr**;**

redisLog**(**REDIS\_NOTICE**,**"Parent agreed to stop sending diffs. Finalizing AOF..."**);**

*/\* Read the final diff if any. \*/*

aofReadDiffFromParent**();**

*/\* Write the received diff to the file. \*/*

redisLog**(**REDIS\_NOTICE**,**

"Concatenating %.2f MB of AOF diff received from parent."**,**

**(**double**)** sdslen**(**server**.**aof\_child\_diff**)** **/** **(**1024**\***1024**));**

**if** **(**rioWrite**(&**aof**,**server**.**aof\_child\_diff**,**sdslen**(**server**.**aof\_child\_diff**))** **==** 0**)**

**goto** werr**;**

*/\* Make sure data will not remain on the OS's output buffers \*/*

**if** **(**fflush**(**fp**)** **==** EOF**)** **goto** werr**;**

**if** **(**fsync**(**fileno**(**fp**))** **==** **-**1**)** **goto** werr**;**

**if** **(**fclose**(**fp**)** **==** EOF**)** **goto** werr**;**

*/\* Use RENAME to make sure the DB file is changed atomically only*

*\* if the generate DB file is ok. \*/*

**if** **(**rename**(**tmpfile**,**filename**)** **==** **-**1**)** **{**

redisLog**(**REDIS\_WARNING**,**"Error moving temp append only file on the final destination: %s"**,** strerror**(**errno**));**

unlink**(**tmpfile**);**

**return** REDIS\_ERR**;**

**}**

redisLog**(**REDIS\_NOTICE**,**"SYNC append only file rewrite performed"**);**

**return** REDIS\_OK**;**

werr**:**

redisLog**(**REDIS\_WARNING**,**"Write error writing append only file on disk: %s"**,** strerror**(**errno**));**

fclose**(**fp**);**

unlink**(**tmpfile**);**

**if** **(**di**)** dictReleaseIterator**(**di**);**

**return** REDIS\_ERR**;**

**}**

首先创建并打开临时文件temp-rewriteaof-<pid>.aof，然后用该文件的文件指针fp初始化rio结构的aof；然后初始化server.aof\_child\_diff，它用于缓存父进程发来的新数据；

如果server.aof\_rewrite\_incremental\_fsync为真，则设置aof的io.file.autosync为32M，也就是每写入文件32M数据，就进行一次fsync操作；

然后，依次轮训Redis的server.dbnum个数据库，开始遍历数据库中的数据，进行AOF重写工作。

首先是将"\*2\r\n$6\r\nSELECT\r\n"以及当前的数据库索引写入aof中；

然后利用字典迭代器di，从数据库的字典中依次取出键key，值对象o，以及超时时间expiretime，如果该key已经超时，则不再写入aof；然后根据值对象的类型调用不同的函数写入aof中：字符串对象，每次写入一个键值对，将命令"set key value"按照统一请求协议的格式写入aof中；列表对象调用rewriteListObject写入；集合对象调用rewriteSetObject写入；有序集合对象调用rewriteSortedSetObject写入；哈希对象调用rewriteHashObject写入。

写入键值对后，如果该键设置了超时时间，则还写入一个PEXPIREAT命令；

每当写入10k的数据后，就调用aofReadDiffFromParent，从管道中读取服务器（父进程）缓存的新数据，追加到server.aof\_child\_diff中；

所有数据库的所有数据都重写完之后，先调用一次fflush和fsync操作，从而使aof文件内容确实写入磁盘。因父进程还在不断的发送新数据，这样可以使后续的fsync操作快一些；

注意，在子进程中可以直接调用fsync，因为它不会阻塞Redis服务器，而在父进程中，调用fsync、unlink等可能阻塞服务器的函数时，需要小心调用，大多是通过后台线程完成的。

接下来，再次调用aofReadDiffFromParent从父进程中读取累积的新数据，因为父进程从客户端接收数据的速度可能大于其向子进程发送数据的速度，所以这里最多耗时1s的时间进行读取，并且如果有20次读取不到数据时，直接就停止该过程；

接下来，向管道server.aof\_pipe\_write\_ack\_to\_parent中发送字符"!"，以使父进程停止发送缓存的新数据；然后从管道server.aof\_pipe\_read\_ack\_from\_parent中，尝试读取父进程的回应"!"，这里读取的超时时间为5s；

然后，最后一次调用aofReadDiffFromParent，读取管道中的剩余数据；并将server.aof\_child\_diff的内容写入到aof中；然后调用fflush和fsync，保证aof文件内容确实写入磁盘；然后fclose(fp)；

最后，将临时文件改名为filename，并返回REDIS\_OK。注意，这里的参数filename，其实也是一个临时文件，其值为temp-rewriteaof-bg-<pid>.aof，子进程之所以将重写的AOF文件记录到临时文件中，是因为此时父进程还在向旧的AOF文件中追加命令。当子进程完成AOF重写之后，父进程就会进行收尾工作，用新的重写AOF文件，替换旧的AOF文件。

2：AOF重写收尾工作

子进程执行完AOF重写后退出，父进程wait得到该子进程的退出状态后，进行AOF重写的收尾工作：

首先将服务器缓存的，剩下的新数据写入该临时文件中，这样该AOF临时文件就完全与当前数据库状态一致了；然后将临时文件改名为配置的AOF文件，并且更改AOF文件描述符，该过程中，为了避免删除操作会阻塞服务器，会使用后台线程进行close操作。

该过程由函数backgroundRewriteDoneHandler实现，代码如下：

void backgroundRewriteDoneHandler**(**int exitcode**,** int bysignal**)** **{**

**if** **(!**bysignal **&&** exitcode **==** 0**)** **{**

int newfd**,** oldfd**;**

char tmpfile**[**256**];**

long long now **=** ustime**();**

mstime\_t latency**;**

redisLog**(**REDIS\_NOTICE**,**

"Background AOF rewrite terminated with success"**);**

*/\* Flush the differences accumulated by the parent to the*

*\* rewritten AOF. \*/*

latencyStartMonitor**(**latency**);**

snprintf**(**tmpfile**,**256**,**"temp-rewriteaof-bg-%d.aof"**,**

**(**int**)**server**.**aof\_child\_pid**);**

newfd **=** open**(**tmpfile**,**O\_WRONLY**|**O\_APPEND**);**

**if** **(**newfd **==** **-**1**)** **{**

redisLog**(**REDIS\_WARNING**,**

"Unable to open the temporary AOF produced by the child: %s"**,** strerror**(**errno**));**

**goto** cleanup**;**

**}**

**if** **(**aofRewriteBufferWrite**(**newfd**)** **==** **-**1**)** **{**

redisLog**(**REDIS\_WARNING**,**

"Error trying to flush the parent diff to the rewritten AOF: %s"**,** strerror**(**errno**));**

close**(**newfd**);**

**goto** cleanup**;**

**}**

latencyEndMonitor**(**latency**);**

latencyAddSampleIfNeeded**(**"aof-rewrite-diff-write"**,**latency**);**

redisLog**(**REDIS\_NOTICE**,**

"Residual parent diff successfully flushed to the rewritten AOF (%.2f MB)"**,** **(**double**)** aofRewriteBufferSize**()** **/** **(**1024**\***1024**));**

*/\* The only remaining thing to do is to rename the temporary file to*

*\* the configured file and switch the file descriptor used to do AOF*

*\* writes. We don't want close(2) or rename(2) calls to block the*

*\* server on old file deletion.*

*\**

*\* There are two possible scenarios:*

*\**

*\* 1) AOF is DISABLED and this was a one time rewrite. The temporary*

*\* file will be renamed to the configured file. When this file already*

*\* exists, it will be unlinked, which may block the server.*

*\**

*\* 2) AOF is ENABLED and the rewritten AOF will immediately start*

*\* receiving writes. After the temporary file is renamed to the*

*\* configured file, the original AOF file descriptor will be closed.*

*\* Since this will be the last reference to that file, closing it*

*\* causes the underlying file to be unlinked, which may block the*

*\* server.*

*\**

*\* To mitigate the blocking effect of the unlink operation (either*

*\* caused by rename(2) in scenario 1, or by close(2) in scenario 2), we*

*\* use a background thread to take care of this. First, we*

*\* make scenario 1 identical to scenario 2 by opening the target file*

*\* when it exists. The unlink operation after the rename(2) will then*

*\* be executed upon calling close(2) for its descriptor. Everything to*

*\* guarantee atomicity for this switch has already happened by then, so*

*\* we don't care what the outcome or duration of that close operation*

*\* is, as long as the file descriptor is released again. \*/*

**if** **(**server**.**aof\_fd **==** **-**1**)** **{**

*/\* AOF disabled \*/*

*/\* Don't care if this fails: oldfd will be -1 and we handle that.*

*\* One notable case of -1 return is if the old file does*

*\* not exist. \*/*

oldfd **=** open**(**server**.**aof\_filename**,**O\_RDONLY**|**O\_NONBLOCK**);**

**}** **else** **{**

*/\* AOF enabled \*/*

oldfd **=** **-**1**;** */\* We'll set this to the current AOF filedes later. \*/*

**}**

*/\* Rename the temporary file. This will not unlink the target file if*

*\* it exists, because we reference it with "oldfd". \*/*

latencyStartMonitor**(**latency**);**

**if** **(**rename**(**tmpfile**,**server**.**aof\_filename**)** **==** **-**1**)** **{**

redisLog**(**REDIS\_WARNING**,**

"Error trying to rename the temporary AOF file: %s"**,** strerror**(**errno**));**

close**(**newfd**);**

**if** **(**oldfd **!=** **-**1**)** close**(**oldfd**);**

**goto** cleanup**;**

**}**

latencyEndMonitor**(**latency**);**

latencyAddSampleIfNeeded**(**"aof-rename"**,**latency**);**

**if** **(**server**.**aof\_fd **==** **-**1**)** **{**

*/\* AOF disabled, we don't need to set the AOF file descriptor*

*\* to this new file, so we can close it. \*/*

close**(**newfd**);**

**}** **else** **{**

*/\* AOF enabled, replace the old fd with the new one. \*/*

oldfd **=** server**.**aof\_fd**;**

server**.**aof\_fd **=** newfd**;**

**if** **(**server**.**aof\_fsync **==** AOF\_FSYNC\_ALWAYS**)**

aof\_fsync**(**newfd**);**

**else** **if** **(**server**.**aof\_fsync **==** AOF\_FSYNC\_EVERYSEC**)**

aof\_background\_fsync**(**newfd**);**

server**.**aof\_selected\_db **=** **-**1**;** */\* Make sure SELECT is re-issued \*/*

aofUpdateCurrentSize**();**

server**.**aof\_rewrite\_base\_size **=** server**.**aof\_current\_size**;**

*/\* Clear regular AOF buffer since its contents was just written to*

*\* the new AOF from the background rewrite buffer. \*/*

sdsfree**(**server**.**aof\_buf**);**

server**.**aof\_buf **=** sdsempty**();**

**}**

server**.**aof\_lastbgrewrite\_status **=** REDIS\_OK**;**

redisLog**(**REDIS\_NOTICE**,** "Background AOF rewrite finished successfully"**);**

*/\* Change state from WAIT\_REWRITE to ON if needed \*/*

**if** **(**server**.**aof\_state **==** REDIS\_AOF\_WAIT\_REWRITE**)**

server**.**aof\_state **=** REDIS\_AOF\_ON**;**

*/\* Asynchronously close the overwritten AOF. \*/*

**if** **(**oldfd **!=** **-**1**)** bioCreateBackgroundJob**(**REDIS\_BIO\_CLOSE\_FILE**,(**void**\*)(**long**)**oldfd**,NULL,NULL);**

redisLog**(**REDIS\_VERBOSE**,**

"Background AOF rewrite signal handler took %lldus"**,** ustime**()-**now**);**

**}** **else** **if** **(!**bysignal **&&** exitcode **!=** 0**)** **{**

server**.**aof\_lastbgrewrite\_status **=** REDIS\_ERR**;**

redisLog**(**REDIS\_WARNING**,**

"Background AOF rewrite terminated with error"**);**

**}** **else** **{**

server**.**aof\_lastbgrewrite\_status **=** REDIS\_ERR**;**

redisLog**(**REDIS\_WARNING**,**

"Background AOF rewrite terminated by signal %d"**,** bysignal**);**

**}**

cleanup**:**

aofClosePipes**();**

aofRewriteBufferReset**();**

aofRemoveTempFile**(**server**.**aof\_child\_pid**);**

server**.**aof\_child\_pid **=** **-**1**;**

server**.**aof\_rewrite\_time\_last **=** time**(NULL)-**server**.**aof\_rewrite\_time\_start**;**

server**.**aof\_rewrite\_time\_start **=** **-**1**;**

*/\* Schedule a new rewrite if we are waiting for it to switch the AOF ON. \*/*

**if** **(**server**.**aof\_state **==** REDIS\_AOF\_WAIT\_REWRITE**)**

server**.**aof\_rewrite\_scheduled **=** 1**;**

**}**

如果子进程执行失败，或者被信号杀死，则标记server.aof\_lastbgrewrite\_status为REDIS\_ERR，然后记录日志错误信息；

如果子进程执行AOF重写成功，则首先打开子进程进行AOF重写的临时文件temp-rewriteaof-bg-<pid>.aof，打开的描述符是newfd；

然后调用aofRewriteBufferWrite，将服务器缓存的剩下的新数据写入该临时文件中，这样该AOF临时文件就完全与当前数据库状态一致了；

接下来要做的就是将临时文件改名为配置的AOF文件，并且更改AOF文件描述符，该过程中要保证close或rename不会阻塞服务器。有以下两种可能的场景：

a：AOF功能被禁用，将临时文件改名为配置的AOF文件，当该文件已经存在时会被删除，删除过程可能阻塞服务器；

b：AOF功能被启用，在将临时文件改名为配置的AOF文件后，会关闭原来的AOF文件描述符，关闭后该文件的描述符引用计数为0，因此会直接删除该文件，这就有可能会阻塞服务器；

为了避免删除操作会阻塞服务器（可能由于场景1的rename，也可能由于场景2的close），这里使用后台线程进行处理。首先通过打开配置的AOF文件，使场景1转换成场景2。rename操作之后的close操作，将会执行unlink操作。

首先，如果server.aof\_fd为-1，说明AOF功能被禁用，尝试打开配置的AOF文件，描述符为oldfd；否则，置oldfd为-1；

然后执行rename操作，将临时文件改名为配置的AOF文件，改名成功后，如果server.aof\_fd为-1，说明AOF功能被禁用，这种情况直接关闭临时文件的描述符newfd；

如果server.aof\_fd不为-1，将AOF文件描述符server.aof\_fd置为newfd，如果server.aof\_fsync为AOF\_FSYNC\_ALWAYS，则直接调用fsync；如果server.aof\_fsync为AOF\_FSYNC\_EVERYSEC，则调用aof\_background\_fsync由后台线程执行fsync操作；

然后，置server.aof\_selected\_db为-1，保证后续添加到AOF中的内容含有SELECT命令；然后调用aofUpdateCurrentSize更新server.aof\_current\_size；然后释放并重置AOF缓存server.aof\_buf；

如果oldfd不是-1，则将关闭原AOF配置文件的任务放入任务队列中，以使后台线程执行，关闭后，原AOF配置文件就会被删除；

最后，执行清理工作，调用aofClosePipes关闭重写AOF时使用的管道；调用aofRewriteBufferReset重置重写AOF缓存；删除重写AOF临时文件；设置server.aof\_child\_pid为-1等；

其他有关AOF功能的代码，参考：

https://github.com/gqtc/redis-3.0.5/blob/master/redis-3.0.5/src/aof.c