

AOF持久化及AOF重写的配置：

默认AOF方式是关闭的，如下图：

```
##### APPEND ONLY MODE #####

# By default Redis asynchronously dumps the dataset on disk. This mode is
# good enough in many applications, but an issue with the Redis process or
# a power outage may result into a few minutes of writes lost (depending on
# the configured save points).
#
# The Append Only File is an alternative persistence mode that provides
# much better durability. For instance using the default data fsync policy
# (see later in the config file) Redis can lose just one second of writes in a
# dramatic event like a server power outage, or a single write if something
# wrong with the Redis process itself happens, but the operating system is
# still running correctly.
#
# AOF and RDB persistence can be enabled at the same time without problems.
# If the AOF is enabled on startup Redis will load the AOF, that is the file
# with the better durability guarantees.
#
# Please check http://redis.io/topics/persistence for more information.

appendonly no

# The name of the append only file (default: "appendonly.aof")

appendfilename "appendonly.aof"
```

如果要开启的话，就是把no改写成yes。如下图：

```
# Please check http://redis.io/topics/persistence for more information.

appendonly yes
```

默认文件名称appendonly.aof，你也可以修改文件名。默认保存目录同样也是配置文件中dir配置项中的设置，它和RDB共用一个目录。如下图：

```
[root@Redis01 6379]# ls
appendonly.aof dump.rdb
[root@Redis01 6379]#
```

默认同步策略是每秒，如下图：

```
# If unsure, use "everysec".

# appendfsync always
appendfsync everysec
# appendfsync no
```

我们对数据库做一些操作然后查看一下appendonly.aof文件内容

```
[root@Redis01 6379]# cat appendonly.aof
*2
$6
SELECT
$1
0
*3
$3
set
$4
name
$3
```

```
aaa
[root@Redis01 6379]#
```

它会记录所有写操作内容。

*2	表示2个参数
\$6	表示第一个参数长度为6
SELECT	第一个参数
\$1	第二个参数长度为1
0	第二个参数

AOF重写策略

```
# If you have latency problems turn this to "yes". Otherwise leave it as
# "no" that is the safest pick from the point of view of durability.
no-appendfsync-on-rewrite no

# Automatic rewrite of the append only file.
# Redis is able to automatically rewrite the log file implicitly calling
# BGREWRITEAOF when the AOF log size grows by the specified percentage.
#
# This is how it works: Redis remembers the size of the AOF file after the
# latest rewrite (if no rewrite has happened since the restart, the size of
# the AOF at startup is used).
#
# This base size is compared to the current size. If the current size is
# bigger than the specified percentage, the rewrite is triggered. Also
# you need to specify a minimal size for the AOF file to be rewritten, this
# is useful to avoid rewriting the AOF file even if the percentage increase
# is reached but it is still pretty small.
#
# Specify a percentage of zero in order to disable the automatic AOF
# rewrite feature.
auto-aof-rewrite-percentage 100
auto-aof-rewrite-min-size 64mb

# An AOF file may be found to be truncated at the end during the Redis
```


AOF持久化实现原理:

当AOF持久化开启后，当对数据库进行一次更新操作后，更新命令就会被追加到aof_buf缓冲区的末尾，然后由缓冲区写入到AOF文件。

AOF文件中记录的内容就是对数据更新操作的指令。这个文件本身就是以文本来记录的，如下图：

```
[root@Redis01 6379]# cat appendonly.aof
*2
$6
SELECT
$1
0
*3
$3
set
```

```
$4
name
$3
aaa
[root@Redis01 6379]#
```




当需要恢复数据的时候，通过执行AOF文件中记录的更新指令，就可以完成。人为的看里面的指令，然后手动敲命令也可以完成。

AOF重写实现原理：

因为AOF持久化是通过记录命令的方式来保存数据库状态的，随着时间的推移AOF文件肯定会逐渐增大，如果不加以控制会对AOF持久化性能以及数据恢复造成影响。下面举例来更加形象的说明重写的必要：

我们以一个压缩列表为例

```
127.0.0.1:6379>
127.0.0.1:6379> rpush list A B
(integer) 2
127.0.0.1:6379> rpush list C
(integer) 3
127.0.0.1:6379> rpush list D
(integer) 4
127.0.0.1:6379> lpop list
"A"
127.0.0.1:6379> rpush list E F
(integer) 5
127.0.0.1:6379> lrange list 0 -1
1) "B"
2) "C"
3) "D"
4) "E"
5) "F"
127.0.0.1:6379>
```



根据AOF的原理，那么上面红色方框中的5条命令都要追加到AOF文件中，其实我们看到最后list的状态就是BCDEF值。也就是说为例实现最后的状态，需要追加5条命令。所以在大量内存读写的业务里AOF文件增长的很快，为例解决这个问题，Redis提供了AOF重写功能。

AOF重写就是创建一个新的AOF文件来替换现有的AOF文件，实际上AOF重写并不对现有的旧AOF文件进行操作。

以上面例子来说，当进行重写的时候直接从数据库里去获取list的最新状态，然后在新的AOF文件中直接写一条rpushlist B C D E F命令，从而避免写5条的操作，这样AOF文件的增长速度就会降低，同时容量也不会特别大。

AOF重写程序aof_rewrite函数去完成创建新的AOF文件的任务，但是该函数并不会由Redis主进程去直接调用，而是使用子进程后台去执行（BGREWRITEAOF，该命令其实就是执行aof_rewrite，只不过是子进程去调用的），这时主进程就不会被阻塞，那么就可以在执行重写的过程中父进程可以继续对外提供响应。整个过程如下：

- 当重写被触发时父进程调用一个函数，该函数创建一个子进程用于执行BGREWRITEAOF，该子进程创建一个临时文件，然后父进程继续对外提供读写服务
- 子进程遍历数据库，将每个键值的最新状态输出到临时文件中，在BGREWRITEAOF过程中，父进程把所有对数据库的更新命令同时写入到AOF缓冲区和AOF重写缓冲区（aof_rewrite_buf_blocks），AOF缓冲区（aof_buf）会继续同步到现有AOF文件中（一般情况下在AOF重写期间不建议把AOF缓冲区的内容同步到现有的AOF文件中，这会降低性能，默认为NO）
- AOF重写完成后子进程通知父进程，父进程调用信号处理函数
- 信号处理函数会阻塞父进程对外提供读写操作（时间很短，不阻塞就又会又会出现数据不一致的情况），然后将AOF重写缓冲区的内容写入到新的AOF文件中，最后用新的AOF文件替换现有AOF文件（更名操作）

APPENDFSYNC选项说明：

参数	说明
always	将aof_buf缓冲区中的所有内容写入并同步到AOF文件中，立即执行write数据的安全性最高，但是执行最慢，如果出现故障只会丢失一个事件循环
everysec	将aof_buf缓冲区的所有内容写入到AOF文件，如果上次同步AOF的时间行同步，每隔一秒执行一次write()和fsync()系统调用。数据安全性居中，数据。
no	将aof_buf缓冲区的所有内容写入到AOF文件，但是何时同步由操作系统调用。写入动作效率高，但是不执行同步，但是单次同步消耗时间最长失上一次同步之后的所有数据。

这里要特别说明一下Linux系统的文件写入和同步原理，为什么要说这个，因为不解释一下这个过程，你就很难理解APPENDFSYNC选项中的no参数，如果把Always理解为总是、一直或者实时；而把everysec理解为每秒的话，那no的含义难道是不执行AOF文件同步吗？如果不同步文件，那开启AOF持久化干嘛呢？

在Redis调用appendfsync函数的时候，其实是先调用一个write()函数，然后再调用sync()或者fsync()函数（对于任何程序来说只要想把数据写入磁盘其过程都一样，有些也有例外）。

用户空间：常规进程所在区域，用户发起的，此区域的代码不能直接访问硬件

内核空间：操作系统所在区域，能和设备控制器通讯

当调用了write()函数时，该函数一旦返回正常值，我们可能就认为数据已经写入到了

磁盘，但实际上，操作系统在实现磁盘文件的IO时，为了保证IO的效率，会在内存中使用一段专门的地址空间，该空间叫做内核空间，而内核空间之内又会有一段是用作IO的数据缓冲区（这个缓冲区和之前说的aof_buf缓冲区不是一个概念，虽然都在内存中），write()函数的作用就是把数据写入到内核空间的IO缓冲区中。



内核空间的IO缓冲区也有一定大小，当该缓冲区没有写满时或者没有到一个同步周期时，会持续的把write()函数传递的数据写入到该缓冲区中，而当该缓冲区写满或者到了一个同步周期，则会把该缓冲区的内容提交到输出队列，当需要数据到达队列队首的时候，开始执行真正的磁盘IO操作，把数据写入磁盘（这里虽然用来写入磁盘，但是真正的动作不是移动而是复制，复制完成之后，内核空间的IO缓冲区才会释放该数据占用的空间）。这种方式叫做延迟写入。

所以这就会出现一个问题，当调用了write()函数后并不等于数据真的保存到了磁盘，但是这里又会有一个错觉，就是你再次请求该文件的时候，可以显示你最后一次更新的内容，其实这个内容并不是从磁盘上读取过来的，而是从用户空间的缓冲区读取的。接着刚才提到的问题，如果数据在内核空间的IO缓冲区内，而此时操作系统出现故障、断电等异常情况就会造成数据丢失。

为了解决数据丢失问题，Unix系统提供了sync、fsync和fdatasync三个函数。

函数	功能
sync	函数返回0表示成功，该函数负责把所有内核空间中IO缓冲区I队列，然后就返回，它并不等待所有磁盘IO操作完成。所以即等于成功保存到磁盘了。
fsync	函数返回0表示成功，与sync不同，它只会对指定文件描述符的文件相连的所有修改过的数据传送到磁盘上，并且等待磁盘IC数返回0时，才真正表示成功保存到磁盘。数据库会在调用了v
fdatasync	它与fsync类似，它只影响文件数据部分，不涉及数据属性，比fsync它需要较少的写磁盘操作。