redis持久化

2019-11-10 10:04:17

Redis的持久化方式分为RDB和AOF两种方式。这篇笔记记录了RDB和AOF的原理和常用配置，以及在不同场景下的最佳配置，重点在AOF的三种缓存方式，以及两种可能阻塞的情况（AOF第二种保存模式SAVE阻塞以及AOF重写时的阻塞）。

# redis持久化数据

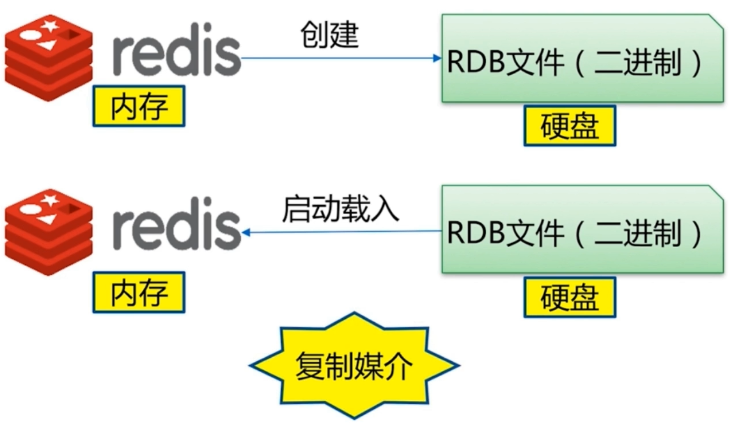
持久化方式有：

* 快照：
  + MySQL Dump
  + Redis RDB
* 写日志：
  + MySQL Binglog
  + Hbase HLog
  + Redis AOF

# RDB持久化方式

## 什么是RDB

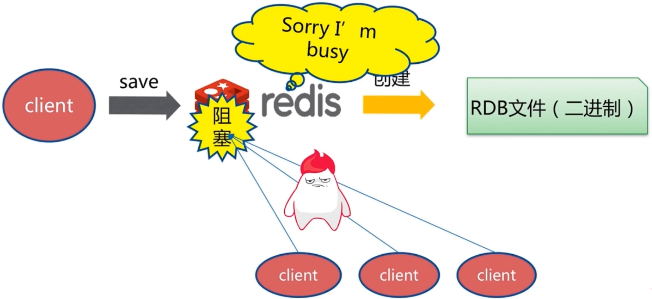
1. RDB是Redis内存到硬盘的快照，用于持久化
2. save通常会阻塞Redis
3. bgsave不会阻塞Redis，但是会for新进程
4. save自动配置满足任一就会被执行
5. 有些触发机制不容忽视



## 触发机制-只要三种方式

### save(同步)

使用save命令，就会开始以同步命令的方式同步文件



**文件策略**

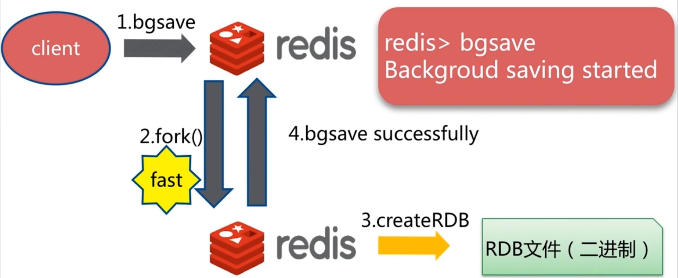
生成一个新的文件，替换老的RDB文件



### bgsave(异步)

使用linux的fork()函数，生成一个redis进程的子进程，生成RDB文件

注意：如果fork()函数执行的慢，依然会阻塞redis，一般fork函数足够快，所以redis会正常的响应客户端



**文件策略**

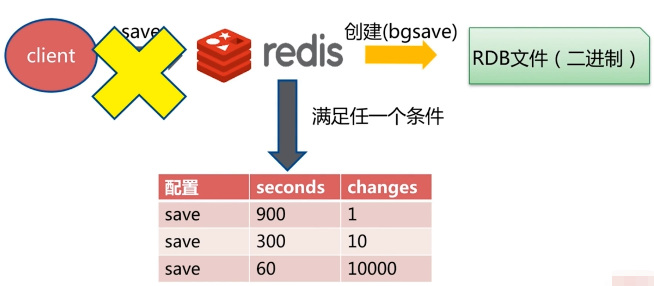
和save相同

### save和bgsave比较

|  |  |  |
| --- | --- | --- |
| 命令 | save | bgsave |
| IO类型 | 同步 | 异步 |
| 是否阻塞 | 是 | 是（阻塞发生在fork） |
| 复杂度 | O(n) | O(n) |
| 优点 | 不会消耗额外内存 | 不阻塞客户端命令 |
| 缺点 | 阻塞客户端命令 | 需要fork,消耗内存 |

### 自动

Redis提供了RDB的save配置，其实就是内部执行了bgsave



**默认配置**

60秒内change了10000条数据以上，三个条件满足一个就会触发bgsave

stop-writes-on-bgsave-error:bgsave发生异常，停止写入

rdbcompression:是否压缩

rdbchecksum:是否采用校验和的方式

save 900 1  
save 300 10  
save 60 10000  
dbfilename dump.rdb  
dir ./  
stop-writes-on-bgsave-error yes  
rdbcompression yes  
rdbchecksum yes

最佳配置

dbfilename dump-${port}.rdb  
dir /bigdiskpath  
stop-writes-on-bgsave-error yes  
rdbcompression yes  
rdbchecksum yes

## 触发机制-不容忽略方式

会触发生成RDB文件的方式

1. 全量复制：主从重复的时候
2. debug reload：不会将内存清空的重启
3. shutdown：执行shutdown

示例：

#端口号  
port 6379  
#是否以守护进程的方式执行  
daemonize yes  
#生成pid的文件名  
pidfile /var/run/redis\_6379.pid  
#日志  
logfile "6379.log"  
#日志级别  
loglevel notice  
  
#RDB配置 注释掉默认配置  
#save 900 1  
#save 300 10  
#save 60 10000  
  
#使用的是默认配置，  
stop-writes-on-bgsave-error yes  
rdbcompression yes  
rdbchecksum yes  
  
#修改rdb文件  
dbfilename dump\_6379.rdb  
#修改rdb保存目录  
dir ./

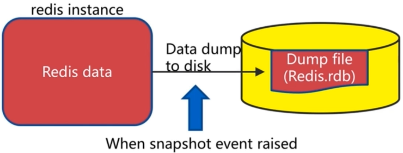
## AOF持久化方式

AOF和RDB区别

https://blog.csdn.net/qq\_42695926/article/details/83069308

### RDB现存问题

1. 耗时、耗性能

* redis将内存的数据dump到硬盘当中，生成rdb文件
* 
  + O(n)数据：耗时
  + fork()：消耗内存，copy-on-write策略
  + Disk I/O：IO性能

1. 不可控、丢失数据

|  |  |
| --- | --- |
| * 时间戳 | * save |
| * T1 | * 执行多个写命令 |
| * T2 | * 满足RDB自动的创建的条件 |
| * T3 | * 再次执行多个写命令 |
| * T4 | * 宕机（**在这里就会发生数据丢失**） |

### 什么是AOF

AOF后台执行的方式和RDB有类似的地方，fork一个子进程，主进程仍进行服务，子进程执行AOF持久化，数据被dump到磁盘上。与RDB不同的是，后台子进程持久化过程中，主进程会记录期间的所有数据变更（主进程还在服务），并存储在server.aof*rewrite*buf\_blocks中；后台子进程结束后，Redis更新缓存到AOF文件中，是RDB持久化所不具备的。

更新缓存可以存储在 server.aof*buf 中，你可以把它理解为一个小型临时中转站，所有累积的更新缓存都会先放入这里，它会在特定时机写入文件或者插入到server.aof*-rewrite*buf*blocks 下链表（下面会详述）；server.aof*buf 中的数据在 propagrate() 添加，在涉及数据更新的地方都会调用propagrate() 以累积变更。更新缓存也可以存储在 server.aof*-rewrite*buf*blocks，这是一个元素类型为 struct aofrwblock 的链表，你可以把它理解为一个仓库，当后台有AOF 子进程的时候，会将累积的更新缓存（在 server.aof\_buf 中）插入到链表中，而当 AOF 子进程结束，它会被整个写入到文件。两者是有关联的。

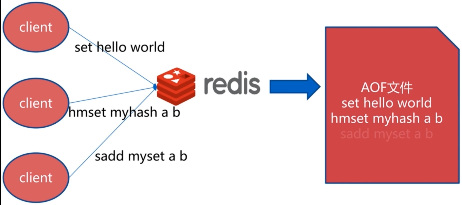
这里的意图即是不用每次出现数据变更的时候都触发一个写操作，可以将写操作先缓存到内存中，待到合适的时机写入到磁盘，如此避免频繁的写操作。当然，完全可以实现让数据变更及时更新到磁盘中。两种做法的好坏就是一种博弈了。

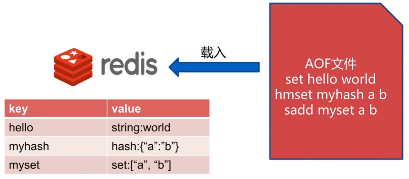
这里有两篇文章，一篇讲AOF的write和save

https://blog.csdn.net/luolaifa000/article/details/84178289

一篇通过源码角度看redis AOF

https://www.cnblogs.com/williamjie/p/9546663.html





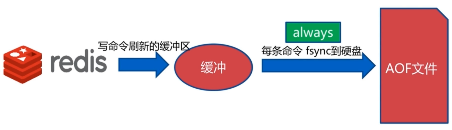
### AOF三种策略(SAVE的三种策略)

AOF操作主要有SAVE和WRITE

https://www.jianshu.com/p/1e34fdc51e3b

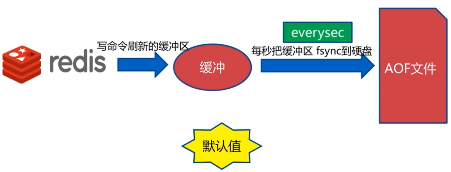
always

每条命令都写入到硬盘当中



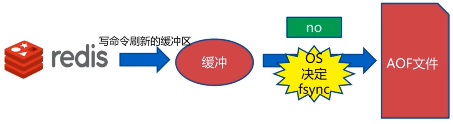
everysec

每秒把数据刷新到硬盘（是配置的默认值）



no

由系统决定什么时候刷新



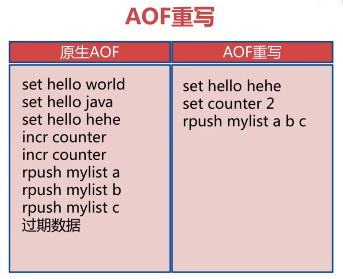
always、everysec、no

|  |  |  |  |
| --- | --- | --- | --- |
| 命令 | always | everysec | no |
| 优点 | 不丢失数据 | 每秒一次fsync丢1秒数据 | 不用管 |
| 缺点 | IO开销较大，一般的sata盘只有几百TPS | 丢1秒数据 | 不可控 |

### AOF重写

AOP重写的好处

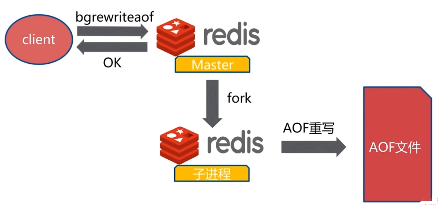
* 减少磁盘用量
* 加快恢复速度



### AOP重写实现的两种方式

#### **bgrewriteaof**

（这里就是往硬盘里重写数据，上面的图只是为了演示）



#### **AOF重写配置**

实际上也是执行了bgrewriteaof

配置

#AOF文件重写需要的尺寸  
auto-aof-rewrite-min-size  
#AOF文件增长率  
auto-aof-rewrite-percentage

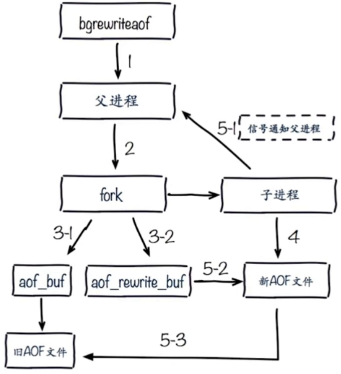
统计

#AOF当前尺寸（单位：字节）  
aof\_current\_size  
#AOF上次启动和重写的尺寸（单位：字节）  
aof\_base\_size

自动触发时机（以下两个条件同时满足）

#当前尺寸大于最小尺寸 aof*current*size > auto-aof-rewrite-min-size #增长率 当前尺寸-上一次重写(或者重启)的尺寸/上一次重写(或者重启)的尺寸 > aof*current*size - aof*base*size/aof*base*size > auto-aof-rewrite-percentage

### AOF重写流程



### AOF配置

#将AOF功能打开  
appendonly yes  
  
#AOF的文件名  
appendfilename "appendonly-${port}.aof"  
  
#AOF同步的三种策略  
appendfsync everysec  
  
#保存rdb aof 和日志的目录  
dir /bigdiskpath  
  
#AOF重写的时候，是否要做AOF的apend操作(这里配置的是不进行次操作)，就是在AOF重写的时候（比较消耗性能），这段时间是否继续写AOF，如果AOF重写失败，  
no-appendfsync-on-rewrite yes  
  
#增长率  
auto-aof-rewrite-percentage 100  
  
#尺寸  
auto-aof-rewrite-min-size 64mb

no-appendfsync-on-rewrite解释：

bgrewriteaof机制，在一个子进程中进行aof的重写，从而不阻塞主进程对其余命令的处理，同时解决了aof文件过大问题。

现在问题出现了，**同时在执行bgrewriteaof操作和主进程写aof文件的操作，两者都会操作磁盘，而bgrewriteaof往往会涉及大量磁盘操作，这样就会造成主进程在写aof文件的时候出现阻塞的情形**，现在no-appendfsync-on-rewrite参数出场了。如果该参数设置为no，是最安全的方式，不会丢失数据，但是要忍受阻塞的问题。如果设置为yes呢？这就相当于将appendfsync设置为no，这说明并没有执行磁盘操作，只是写入了缓冲区，因此这样并不会造成阻塞（因为没有竞争磁盘），但是如果这个时候redis挂掉，就会丢失数据。丢失多少数据呢？在linux的操作系统的默认设置下，最多会丢失30s的数据。

问：我对aof重写的理解： no-appendfsync-on-rewrite=yes，会让redis在进行aof重写时，不阻塞主进程对客户端的请求。 某时刻T1触发了重写，redis fork出一条子进程，将数据以写操作命令的形式写到新的tmp.aof文件，期间T2时刻，客户端发送了一条写操作请求SET1，此时主进程应该是

1. 把SET1加入到原来的aof文件
2. 把SET1写到重写缓存

时刻T3结束重写，然后主进程将重写缓存中的写操作SET1加到新的tmp.aof文件中，最后替换掉原aof文件。

假如T1-T3时间段内redis意外宕机，即使重写缓存里的SET1没有添加到tmp.aof文件中，重启的时候，也是通过原有aof文件（包括SET1操作）来恢复数据，不会导致**意料之外**的数据丢失。

答：aof 和 rdb 是两个机制, 没有什么关系 如果 no-appendfsync-on-rewrite=yes, 这个时候主线程的 set 操作会被阻塞掉, 由于没有新的值写入 redis, 所有就没有这个时候数据丢失的可能. 一旦 tmp.aof 重写成功, 就不会有数据丢失. 如果 no-appendfsync-on-rewrite=no, 这个时候主线程的 set 操作不会阻塞, 就会有新值写入 redis, 但是这部分记录不会同步到硬盘上, 就会有数据丢失的问题可能. 一旦 tmp.aof 重写成功就发生故障, 就会产生数据丢失.

### AOF试验

#查看AOF功能是否打开，  
config get appendonly  
config set appendonly  
#将 appendonly 的修改写入到 redis.conf 中  
config rewrite  
  
#这时候应该就生成了aof文件  
#开始aof重写数据  
bgrewriteaof

一篇AOF持久化的文章（详细讲解AOF过程\*\*）

https://blog.csdn.net/yangyutong0506/article/details/46880773

## RDB和AOF的抉择

### RDB和AOF比较

|  |  |  |
| --- | --- | --- |
| 命令 | RDB | AOF |
| 启动优先级 | 低 | 高 |
| 体积 | 小 | 大 |
| 恢复速度 | 快 | 慢 |
| 数据安全性 | 丢数据 | 根据策略决定 |
| 轻重 | 重 | 轻 |

### RDB最佳策略

1. 关
2. 集中管理
3. 主从，从开

### AOF最佳策略

1. “开”：缓存和存储
2. AOF重写集中管理
3. everysec(每秒)

### 最佳策略

1. 小分片(maxmemory)
2. 缓存或者存储
3. 监控（硬盘、内存、负载、网络）
4. 足够的内存

## 开发运维常见问题

### fork操作

1. 同步操作

* for操作只是做内存页的拷贝，而不是内存的拷贝，但是如果fork操作阻塞了，会阻塞redis主线程

1. 与内存量息息相关：内存越大，耗时越长（与机器类型有关）
2. info:latest*fork*userc(查看上一次执行fork时使用的微秒数)

#### 改善fork

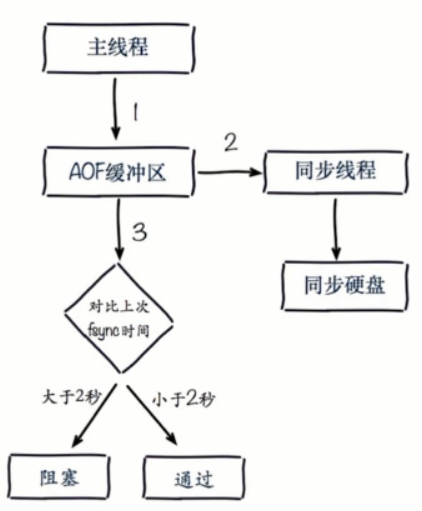
1. 优先使用物理机或者高效支持fork操作的虚拟化技术
2. 控制Redis实例最大可用内存：maxmemory
3. 合理配置Linux内存分配策略：vm.overcommit\_memory=1(默认是0)
4. 降低fork频率：例如放款AOF重写的自动触发机制，不必要的全量复制

### 子进程的开销和优化

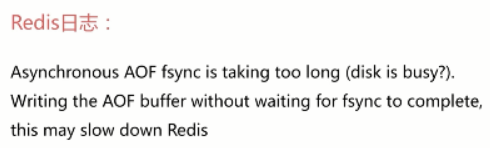
1. CPU
   * 开销：RDB和AOF文件生成（主要体现在文件的重写bgsave、bgrewriteaof将内存的数据写到硬盘），属于CPU密集型
   * 优化：不做CPU绑定，不和CPU密集型部署
2. 内存
   * 开销：fork内存开销，copy-on-write（linux父子进程会共享相同的物理内存文件，父进程有写请求的时候，数据会被复制，从而使各个进程拥有各自的拷贝，这时候才会消耗内存）。
   * 优化：echo never > /sys/kernel/mm/transparent\_hugepage/enabled (不允许单机多实例的时候产生大量的重写，然后在主进程写入量比较少的时候去做一个bgsave或者aof重写，再就是这个配置，linux在2.6.38版本里面增加了THP的特性，支持大的内存页的分配，由原来的4k变成2m)
3. 硬盘
   * 开销：AOF和RDB文件写入，可以结合iostat,iotop分析
   * 不要和高硬盘负载服务部署在一起：存储服务、消息队列等
   * no-appendfsync-on-rewrite = yes (禁止AOF重写时追加的操作)
   * 根据写入量决定磁盘类型 ：例如ssd
   * 单机多实例持久化文件目录可以考虑分盘

### AOF追加阻塞

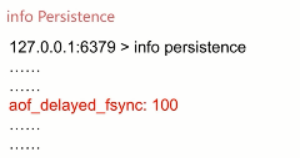
原理看一下下面追加的AOF保存



从

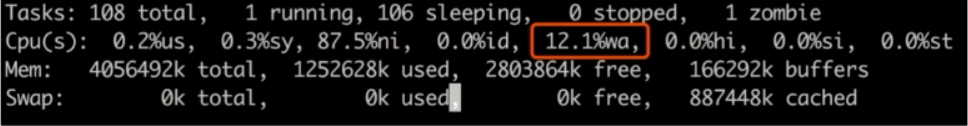


和info persistence命令（每发生一次，就会加1）



都可以看到

还可以通过top命令



### 单机多实例部署

# AOF保存

## AOF文件写入和保存

每当服务器常规任务函数被执行、 或者事件处理器被执行时， aof.c/flushAppendOnlyFile 函数都会被调用， 这个函数执行以下两个工作：

1. WRITE：根据条件，将 aof\_buf 中的缓存写入到 AOF 文件。
2. SAVE：根据条件，调用 fsync 或 fdatasync 函数，将 AOF 文件保存到磁盘中。

两个步骤都需要根据一定的条件来执行， 而这些条件由 AOF 所使用的保存模式来决定， 以下小节就来介绍 AOF 所使用的三种保存模式， 以及在这些模式下， 步骤 WRITE 和 SAVE 的调用条件。

## AOF保存模式

Redis目前支持三种AOF保存模式，它们分别是：

AOF*FSYNC*NO: 不保存

AOF*FSYNC*EVERYSEC: 每一秒保存一次

AOF*FSYNC*ALWAYS: 每执行一个命令保存一次

### 不保存

在这种模式下， 每次调用 flushAppendOnlyFile 函数， WRITE 都会被执行， 但 SAVE 会被略过。

在这种模式下， SAVE 只会在以下任意一种情况中被执行：

* Redis 被关闭
* AOF 功能被关闭
* 系统的写缓存被刷新（可能是缓存已经被写满，或者定期保存操作被执行）

这三种情况下的 SAVE 操作都会引起 Redis 主进程阻塞。

### 每一秒钟保存一次

在这种模式中， SAVE 原则上每隔一秒钟就会执行一次， 因为 SAVE 操作是由后台子线程调用的， 所以它不会引起服务器主进程阻塞。

注意， 在上一句的说明里面使用了词语“原则上”， 在实际运行中， 程序在这种模式下对 fsync 或 fdatasync 的调用并不是每秒一次， 它和调用 flushAppendOnlyFile 函数时 Redis 所处的状态有关。

每当 flushAppendOnlyFile 函数被调用时， 可能会出现以下四种情况：

* 子进程正在执行SAVE，并且：
  1. 这个SAVE的执行时间未超过2秒，那么程序直接返回，并不执行WRITE或新的SAVE
  2. **这个SAVE已经执行超过了2秒，那么程序执行WRITE，单不执行新的SAVE。注意，因为这时的WRITE的写入必须等待子线程先完成（旧的）SAVE，因此这里WRITE会比平时阻塞更长时间**
* 子进程没有在执行SAVE，并且：
  1. 上次成功执行 SAVE 距今不超过 1 秒，那么程序执行 WRITE ，但不执行 SAVE 。
  2. 次成功执行 SAVE 距今已经超过 1 秒，那么程序执行 WRITE 和 SAVE 。

如下图：

根据以上说明可以知道， 在“每一秒钟保存一次”模式下， 如果在情况 1 中发生故障停机， 那么用户最多损失小于 2 秒内所产生的所有数据。

如果在情况 2 中发生故障停机， 那么用户损失的数据是可以超过 2 秒的。

Redis 官网上所说的， AOF 在“每一秒钟保存一次”时发生故障， 只丢失 1 秒钟数据的说法， 实际上并不准确。

### 每执行一个命令保存一次

在这种模式下，每次执行完一个命令之后， WRITE 和 SAVE 都会被执行。

另外，因为 SAVE 是由 Redis 主进程执行的，所以在 SAVE 执行期间，主进程会被阻塞，不能接受命令请求。

## AOF 保存模式对性能和安全性的影响

对于三种 AOF 保存模式， 它们对服务器主进程的阻塞情况如下：

1. 不保存（AOF*FSYNC*NO）：写入和保存都由主进程执行，两个操作都会阻塞主进程。
2. 每一秒钟保存一次（AOF*FSYNC*EVERYSEC）：写入操作由主进程执行，阻塞主进程。**保存操作由子线程执行，不直接阻塞主进程，但保存操作完成的快慢会影响写入操作的阻塞时长。（上面flushAppendOnlyFile 第二种情况）**
3. 每执行一个命令保存一次（AOF*FSYNC*ALWAYS）：和模式 1 一样。

模式 1 的保存操作只会在AOF 关闭或 Redis 关闭时执行， 或者由操作系统触发， 在一般情况下， 这种模式只需要为写入阻塞， 因此它的写入性能要比后面两种模式要高， 当然， 这种性能的提高是以降低安全性为代价的： 在这种模式下， 如果运行的中途发生停机， 那么丢失数据的数量由操作系统的缓存冲洗策略决定。

模式 2 在性能方面要优于模式 3 ， 并且在通常情况下， 这种模式最多丢失不多于 2 秒的数据， 所以它的安全性要高于模式 1 ， 这是一种兼顾性能和安全性的保存方案。

模式 3 的安全性是最高的， 但性能也是最差的， 因为服务器必须阻塞直到命令信息被写入并保存到磁盘之后， 才能继续处理请求。

综合起来，三种 AOF 模式的操作特性可以总结如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 模式 | WRITE 是否阻塞？ | SAVE 是否阻塞？ | 停机时丢失的数据量 |
| AOF\_FSYNC\_NO | 阻塞 | 阻塞 | 操作系统最后一次对 AOF 文件触发 SAVE 操作之后的数据。 |
| AOF\_FSYNC\_EVERYSEC | 阻塞 | 不阻塞 | 一般情况下不超过 2 秒钟的数据。 |
| AOF\_FSYNC\_ALWAYS | 阻塞 | 阻塞 | 最多只丢失一个命令的数据。 |