宕机了，Redis数据丢了怎么办？

苏三说技术 2021-12-19 19:27

以下文章来源于码猿技术专栏 ，作者不才陈某

码猿技术专栏

前蚂蚁P8，纯粹的技术人，专注于Java后端技术分享，只写外面看不到的干货，你想要的都在这里.....

目录

前言

Redis 作为内存型的数据库，虽然很快，依然有着很大的隐患，一旦**「服务器宕机」**重启，内存中数据还会存在吗？

很容易想到的一个方案是从后台数据恢复这些数据，如果数据量很小，这倒是一个可行的方案。但是如果数据量过大，频繁的从后台数据库访问数据，压力很大；另外一方面恢复数据的时间极慢。

对于 Redis 来说，实现数据的持久化和快速恢复是至关重要。

今天这篇文章就来介绍一下 Redis 持久化的两种机制 AOF 日志、 RDB 快照。

什么是 AOF 日志?

AOF（Append Only File）日志称之为**「写后日志」**，即是命令先执行完成，把数据写入内存，然后才会记录日志。

AOF 日志（文本形式）会将收到每一条的命令且执行成功的命令以一定的格式写入到文本中（追加的方式）。

「写后日志有什么好处呢？」 如下：

1. 对于写前日志无论命令是否执行成功都会被记录，但是 Redis 的写后日志则只有命令执行成功才会被写入日志，避免了日志中存在错误命令；
2. 同时由于是命令执行成功之后才会写入日志，因此不会阻塞当前命令的执行。

但是 AOF 日志也有**「潜在的风险」**，分析如下：

1. 由于是写后日志，如果在命令执行成功之后，在日志未写入磁盘之前服务器突然宕机，那重启恢复数据的时候，这部分的数据肯定在日志文件中不存在了，那么将会丢失。（无法通过后台数据库恢复的情况下）
2. 虽然不会阻塞当前命令的执行，由于记录日志也是在主线程中（Redis 是单线程），如果日志写入磁盘的时候突然阻塞了，肯定会影响下一个命令的执行。

为了解决上面的风险， AOF 日志提供了三种回写策略。

三种写回策略

AOF 机制提供了三种回写策略，这些都在 appendfsync 配置，如下：

1. Always （同步写回）：命令执行完成，立马同步的将日志写入磁盘
2. Everysec （每秒写回）：命令执行完成后，先将日志写入 AOF 文件的内存缓冲区，每隔一秒把缓冲区中内容写入磁盘。
3. No (操作系统控制的写回)：每个写命令执行完，只是先把日志写到 AOF 文件的内存缓冲区，由操作系统决定何时将缓冲区内容写回磁盘。

其实这三中写回策略都无法解决主线程的阻塞和数据丢失的问题，分析如下：

1. 同步写回：基本不丢失数据，但是每步操作都会有一个慢速的落盘操作，不可避免的影响主线程性能。
2. 每秒写回：采用一秒写一次到 AOF 日志文件中，但是一旦宕机还是会丢失一秒的数据。
3. 操作系统控制的写回：在写完缓冲区之后则会写入磁盘，但是数据始终在缓冲区的时间内一旦宕机，数据还是会丢失。

以上三种策略优缺点总结如下表：

策略	优点	缺点

策略	优点	缺点
Always	可靠性高，数据基本不丢失	每个写命令都要落盘，性能影响较大
Everysec	性能适中	宕机时丢失一秒数据
No	性能好	宕机时丢失数据较多

日志文件太大怎么办？

随着数据量的增大，AOF日志文件难免会很大，这样将会导致写入和恢复数据都将变得非常慢。此时AOF提供了一种**「重写机制」**解决这一问题。

> 重写机制理解起来很简单，即是 `Redis` 会创建一个新的 `AOF` 日志文件，将每个键值对最终的值用一条命令写入日志文件中。

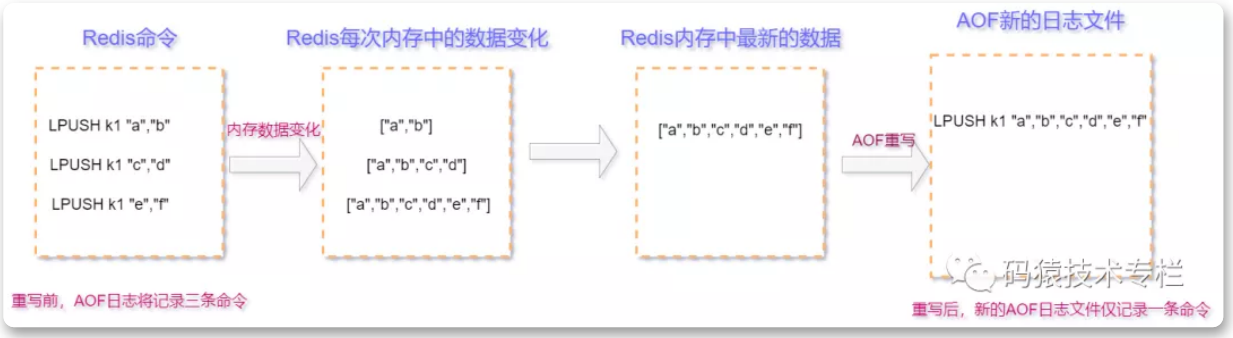
比如读取了键值对 `key1:value1` ，重写机制会在新的AOF日志文件中记录如下一条命令：

```
set key1 value1
```

其实即是记录多次修改的最终的值记录在新的AOF日志文件中，这样当恢复数据时可直接执行该命令。

「为什么重写机制能够缩小文件呢？」 当一个键值被多次修改后， `AOF` 日志文件中将会记录多次修改键值的命令，重写机制是根据这个键值最新状态为它生成**「写入」**命令，这样旧文件中的**「多条」**命令在重写后的新日志中变成了**「一条」**命令。

作者画了一张重写流程图，仅供参考，如下：



重写机制流程

AOF重写会阻塞主线程吗？

AOF重写虽然能够缩减日志文件的大小，达到减少日志记录和数据恢复的时间，但是在数据量非常的大情况下把整个数据库重写后的日志写入磁盘是一个非常耗时的过程，难道不会阻塞主线程吗？

「答案是：不会阻塞主线程」；因为AOF重写过程是由后台子进程 `bgrewriteaof` 来完成的，这也是为了避免阻塞主线程，导致数据库性能下降。

其实重写的过程分为两个阶段：**「一个拷贝，两处日志」**。

「一个拷贝」：指每次执行重写时，主线程都 `fork` 一个子线程 `bgrewriteaof` ，主线程会把内存数据拷贝一份到子线程，此时子线程中包含了数据库的最新数据。然后子线程就能在不影响主线程的情况下进行AOF重写了。

「两处日志」是什么？如下：

1. 第一处日志：子线程重写并未阻塞主线程，此时主线程仍然会处理请求，此时的AOF日志仍然正在记录着，这样即使宕机了，数据也是齐全的。第一处日志即是值主线程正在使用的日志。
2. 第二处日志：指新的AOF重写日志；重写过程中的操作也会被写到重写日志缓冲区，这样重写日志也不会丢失最新的操作。等到拷贝数据的所有操作记录重写完成后，重写日志记录的这些最新操作也会写入新的 AOF 文件，以保证数据库最新状态的记录。此时，我们就可以用新的 AOF 文件替代旧文件了。

> **「总结」**： `Redis` 在进行 `AOF` 重写时，会 `fork` 一个子线程（不会阻塞主线程）并进行内存拷贝用于重写，然后使用两个日志保证重写过程中，新写入的数据不会丢失。

AOF的缺点

虽说进行了日志重写后，AOF日志文件会缩减很多，但是在数据恢复过程中仍然是一条命令一条命令（由于单线程，只能顺序执行）的执行恢复数据，这个恢复的过程非常缓慢。

总结

AOF这种通过逐一记录操作命令的日志方式，提供了三种写回策略保证数据的可靠性，分别是 `Always` 、 `Everysec` 和 `No` ，这三种策略在可靠性上是从高到低，而在性能上则是从低到高。

为了避免日志文件过大，Redis提供了重写的机制，每次重写都fork一个子线程，拷贝内存数据进行重写，将多条命令缩减成一条生成键值对的命令，最终重写的日志作为新的日志。

什么是RDB?

RDB (Redis DataBase)是另外一种持久化方式：内存快照。

> RDB 记录的是**「某一个时刻」**的内存数据，并不是操作命令。

这种方式类似于拍照，只保留某一时刻的形象。内存快照是将某一时刻的状态以文件的形式写入磁盘。这样即使宕机了，数据也不会丢失，这个快照文件就称为 RDB 文件。

> 由于记录的是某个时刻的内存数据，数据恢复非常快的，不需要像AOF日志逐一执行记录的命令。

给哪些数据做快照？

为了保证数据的可靠性，Redis执行的**「全量快照」**，也就是把内存中的所有数据都写到磁盘中。

随着数据量的增大，一次性把全部数据都写到磁盘中势必会造成线程阻塞，这就关系到Redis的性能了。

针对线程阻塞的问题Redis提供了两个命令，如下：

1. save：在主线程中执行，会导致主线程阻塞。
2. bgsave：fork 一个子进程，专门用于写入 RDB 文件，避免了主线程的阻塞，这是Redis的默认配置。

这样就可以使用 bgsave 命令执行全量快照，既可以保证数据的可靠性也避免了主线程的阻塞。

快照时能够修改数据吗？

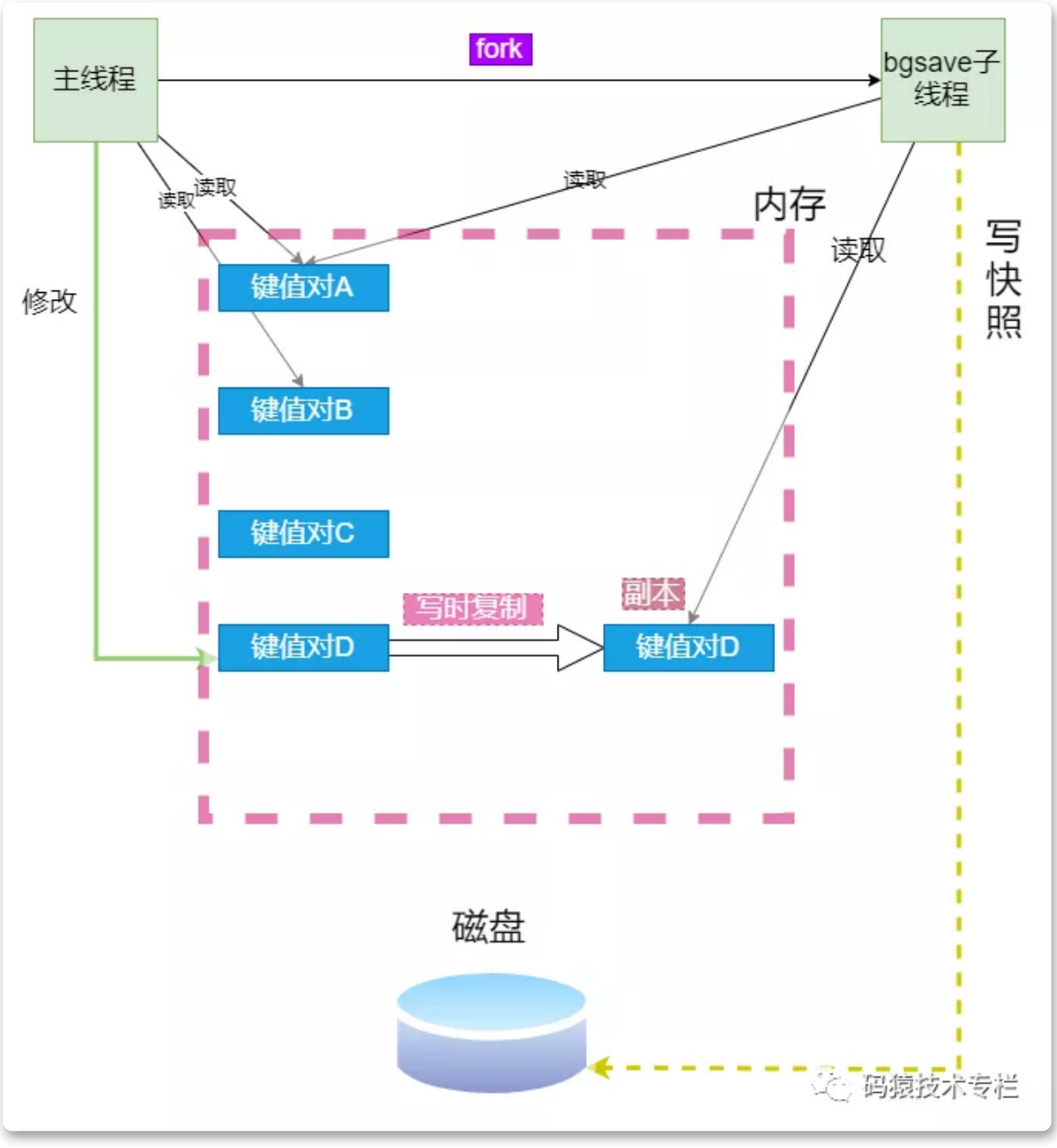
子线程执行全量快照的同时，主线程仍然在接受着请求，读数据肯定没有问题，但是如果个修改了数据，如何能够保证快照的完整性呢？

「举个栗子」：我在 T 时刻进行全量快照，假设数据量有 8G，写入磁盘的过程至少需要 20S，在这 20S 的时间内，一旦内存中的数据发生了修改，则快照的完整性就破坏了。

但是如果在快照时不能修改数据，则对Redis的性能有巨大的影响，对于这个问题，Redis是如何解决的呢？

> Redis 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。

其实很简单， bgsave 命令会 fork 一个子线程，这个子线程共享所有内存的数据，子线程会读取主线程内存中的数据，将他们写入 RDB 文件。



写时复制保证数据可修改

如上图，对于 键值对A 的读取并不会影响子线程，但是如果主线程一旦修改内存中一块数据（例如 键值对D），这块数据将会被复制一个副本，然后 bgsave 子线程会将其写入 RDB 文件。

多久做一次快照？

快照只是记录某一时刻的数据，一旦时间隔离很久，则服务器一旦宕机，则会丢失那段时间的数据。

比如在 `T1` 时间做了一次快照，在 `T1+t` 时又做了一次快照，如果在 `t` 这个时间段内服务器突然宕机了，则快照中只保存了 `T1` 时刻的快照，在 `t` 时间段内的数据修改未被记录（丢失）。如下图：

t时刻宕机，未执行快照

从上图明显可以看出，**「RDB 并不是一个完美的日志记录方案」**，只有让 `t` 时间逐渐缩小，才能保证丢失的数据缩小。

「那么问题来了，时间能够缩短 1秒 吗？」 即是每秒执行一次快照。

> 全量快照是记录某一个时刻的**「全部」**内存数据，每秒执行一次的对Redis性能影响巨大，于是**「增量快照」**就出来了。

增量快照

「增量快照是指做了一次全量快照之后，后续的快照只对修改的数据进行快照记录」，这样可以避免每次都全量快照的开销。

增量快照的前提是Redis能够记住修改的数据，这个功能其实开销也是巨大的，需要保存完整的键值对，这对内存的消耗是巨大的。

>

> 为了解决这个问题，Redis使用了 AOF 和 RDB 混合使用的方式。

AOF和RDB混合使用

这个概念是在 Redis4.0 提出的，简单的说就是**「内存快照以一定的频率执行，比如1小时一次，在两次快照之间，使用AOF日志记录这期间的所有命令操作。」**

> 混合使用的方式使得内存快照不必频繁的执行，并且AOF记录的也不是全部的操作命令，而是两次快照之间的操作命令，不会出现AOF日志文件过大的情况了，避免了AOF重写的开销了。

这个方案既能够用到的RDB的快速恢复的好处，又能享受都只记录操作命令的简单优势，强烈建议使用。

总结

RDB 内存快照记录的是某一个时刻的内存数据，因此能够快速恢复； AOF 和 RDB 混合使用能够使得宕机后数据快速恢复，又能够避免 AOF 日志文件过大。

总结

本文介绍了两种数据恢复和持久化的方案，分别是 AOF 和 RDB 。

AOF 介绍了什么？如下：

1. AOF 是写后日志，通过记录操作命令持久化数据。
2. 由于 AOF 是在命令执行之后记录日志，如果在写入磁盘之前服务器宕机，则会丢失数据；如果写入磁盘的时候突然阻塞，则会阻塞主线程；为了解决以上问题，AOF机制提供了三种写回的策略，每种策略都有不同的优缺点。
3. AOF 日志文件过大怎么办？ AOF 通过 fork 一个子线程重写一个新的日志文件（共享主线程的内存，记录最新数据的写入命令），同时子线程重写，避免阻塞主线程。

RDB 介绍了什么？如下：

1. RDB 是内存快照，记录某一个时刻的内存数据，而不是操作命令。
2. Redis 提供了两个命令，分别是 save 、 bgsave 来执行全量快照，这两个命令的区别则是 save 是在主线程执行，势必会阻塞主线程， bgsave 是在 fork 一个子线程，共享内存。
3. RDB通过操作系统的**「写时复制技术」**，能够保证在执行快照的同时主线程能够修改快照。

4. 由于两次快照之间是存在间隔的，一旦服务器宕机，则会丢失两次间隔时刻的数据， `Redis4.0` 开始使用 `AOF` 日志记录两次快照之间执行的命令（ `AOF` 和 `RDB` 混合使用）。