Redis提供的持久化机制

Redis是一种面向“key-value”类型数据的分布式NoSQL数据库系统，具有高性能、持久存储、适应高并发应用场景等优势。它虽然起步较晚，但发展却十分迅速。

近日，Redis的作者在博客中写到，他看到的所有针对Redis的讨论中，对Redis持久化的误解是最大的，于是他写了一篇长文来对Redis的持久化进行了系统性的论述。
文章主要包含三个方面：Redis持久化是如何工作的、这一性能是否可靠以及和其它类型的数据库比较。以下为文章内容：

一、Redis持久化是如何工作的？

什么是持久化？简单来讲就是将数据放到断电后数据不会丢失的设备中，也就是我们通常理解的硬盘上。
首先我们来看一下数据库在进行写操作时到底做了哪些事，主要有下面五个过程：

- 客户端向服务端发送写操作（数据在客户端的内存中）。
- 数据库服务端接收到写请求的数据（数据在服务端的内存中）。
- 服务端调用write这个系统调用，将数据往磁盘上写（数据在系统内存的缓冲区中）。
- 操作系统将缓冲区中的数据转移到磁盘控制器上（数据在磁盘缓存中）。
- 磁盘控制器将数据写到磁盘的物理介质中（数据真正落到磁盘上）。

故障分析

写操作大致有上面5个流程，下面我们结合上面的5个流程看一下各种级别的故障：

- 当数据库系统故障时，这时候系统内核还是完好的。那么此时只要我们执行完了第3步，那么数据就是安全的，因为后续操作系统会来完成后面几步，保证数据最终会落到磁盘上。
- 当系统断电时，这时候上面5项中提到的所有缓存都会失效，并且数据库和操作系统都会停止工作。所以只有当数据在完成第5步后，才能保证在断电后数据不丢失。

通过上面5步的了解，可能我们会希望搞清下面一些问题：

- 数据库多长时间调用一次write，将数据写到内核缓冲区？

- 内核多长时间会将系统缓冲区中的数据写到磁盘控制器？
- 磁盘控制器又在什么时候把缓存中的数据写到物理介质上？

对于第一个问题，通常数据库层面会进行全面控制。

而对第二个问题，操作系统有其默认的策略，但是我们也可以通过POSIX API提供的fsync系列命令强制操作系统将数据从内核区写到磁盘控制器上。

对于第三个问题，好像数据库已经无法触及，但实际上，大多数情况下磁盘缓存是被设置关闭的，或者是只开启为读缓存，也就是说写操作不会进行缓存，直接写到磁盘。

建议的做法是仅仅当你的磁盘设备有备用电池时才开启写缓存。

数据损坏

所谓数据损坏，就是数据无法恢复，上面我们讲的都是如何保证数据是确实写到磁盘上去，但是写到磁盘上可能并不意味着数据不会损坏。比如我们可能一次写请求会进行两次不同的写操作，当意外发生时，可能会导致一次写操作安全完成，但是另一次还没有进行。如果数据库的数据文件结构组织不合理，可能就会导致数据完全不能恢复的状况出现。

这里通常也有三种策略来组织数据，以防止数据文件损坏到无法恢复的情况：

- 第一种是最粗糙的处理，就是不通过数据的组织形式保证数据的可恢复性。而是通过配置数据同步备份的方式，在数据文件损坏后通过数据备份来进行恢复。实际上MongoDB在不开启操作日志，通过配置Replica Sets时就是这种情况。
- 另一种是在上面基础上添加一个操作日志，每次操作时记一下操作的行为，这样我们可以通过操作日志来进行数据恢复。因为操作日志是顺序追加的方式写的，所以不会出现操作日志也无法恢复的情况。这也类似于MongoDB开启了操作日志的情况。
- 更保险的做法是数据库不进行旧数据的修改，只是以追加方式去完成写操作，这样数据本身就是一份日志，这样就永远不会出现数据无法恢复的情况了。实际上CouchDB就是此做法的优秀范例。

二 、Redis提供了RDB持久化和AOF持久化

RDB机制的优势和略施

RDB持久化是指在指定的时间间隔内将内存中的数据集快照写入磁盘。

也是默认的持久化方式，这种方式是就是将内存中数据以快照的方式写入到二进制文件中,默认的文件名为dump.rdb。

可以通过配置设置自动做快照持久化的方式。我们可以配置redis在n秒内如果超过m个key被修改就自动做快照，下面是默认的快照保存配置

```
save 900 1    #900秒内如果超过1个key被修改，则发起快照保存
save 300 10   #300秒内容如超过10个key被修改，则发起快照保存
save 60 10000
```

RDB文件保存过程

- redis调用fork,现在有了子进程和父进程。
- 父进程继续处理client请求，子进程负责将内存内容写入到临时文件。由于os的写时复制机制（copy on write)父子进程会共享相同的物理页面，当父进程处理写请求时os会为父进程要修改的页面创建副本，而不是写共享的页面。所以子进程的地址空间内的数 据是fork时刻整个数据库的一个快照。
- 当子进程将快照写入临时文件完毕后，用临时文件替换原来的快照文件，然后子进程退出。

client 也可以使用save或者bgsave命令通知redis做一次快照持久化。save操作是在主线程中保存快照的，由于redis是用一个主线程来处理所有 client的请求，这种方式会阻塞所有client请求。所以不推荐使用。

另一点需要注意的是，每次快照持久化都是将内存数据完整写入到磁盘一次，并不 是增量的只同步脏数据。如果数据量大的话，而且写操作比较多，必然会引起大量的磁盘io操作，可能会严重影响性能。

优势

- 一旦采用该方式，那么你的整个Redis数据库将只包含一个文件，这样非常方便进行备份。比如你可能打算没1天归档一些数据。
- 方便备份，我们可以很容易的将一个一个RDB文件移动到其他的存储介质上
- RDB 在恢复大数据集时的速度比 AOF 的恢复速度要快。
- RDB 可以最大化 Redis 的性能：父进程在保存 RDB 文件时唯一要做的就是 fork 出一个子进程，然后这个子进程就会处理接下来的所有保存工作，父进程无须执行任何磁盘 I/O 操作。

劣势

- 如果你需要尽量避免在服务器故障时丢失数据，那么 RDB 不适合你。 虽然 Redis 允许你设置不同的保存点（save point）来控制保存 RDB 文件的频率， 但是， 因为RDB 文件需要保存整个数据集的状态， 所以它并不是一个轻松的操作。 因此你可能会至少 5 分钟才保存一次 RDB 文件。 在这种情况下， 一旦发生故障停机， 你就可能会丢失好几分钟的数据。

- 每次保存 RDB 的时候，Redis 都要 fork() 出一个子进程，并由子进程来进行实际的持久化工作。 在数据集比较庞大时， fork() 可能会非常耗时，造成服务器在某某毫秒内停止处理客户端； 如果数据集非常巨大，并且 CPU 时间非常紧张的话，那么这种停止时间甚至可能会长达整整一秒。 虽然 AOF 重写也需要进行 fork() ，但无论 AOF 重写的执行间隔有多长，数据的耐久性都不会有任何损失。

AOF文件保存过程

redis会将每一个收到的写命令都通过write函数追加到文件中(默认是 appendonly.aof)。当redis重启时会通过重新执行文件中保存的写命令来在内存中重建整个数据库的内容。当然由于os会在内核中缓存 write做的修改，所以可能不是立即写到磁盘上。这样aof方式的持久化也还是有可能会丢失部分修改。不过我们可以通过配置文件告诉redis我们想要 通过fsync函数强制os写入到磁盘的时机。有三种方式如下（默认是：每秒fsync一次）

```
appendonly yes              //启用aof持久化方式
# appendfsync always     //每次收到写命令就立即强制写入磁盘，最慢的，但是保证完全的持久化，不推荐使用
appendfsync everysec     //每秒钟强制写入磁盘一次，在性能和持久化方面做了很好的折中，推荐
# appendfsync no    //完全依赖os，性能最好,持久化没保证
```

aof 的方式也同时带来了另一个问题。持久化文件会变的越来越大。例如我们调用incr test命令100次，文件中必须保存全部的100条命令，其实有99条都是多余的。因为要恢复数据库的状态其实文件中保存一条set test 100就够了。

为了压缩aof的持久化文件。redis提供了bgrewriteaof命令。收到此命令redis将使用与快照类似的方式将内存中的数据 以命令的方式保存到临时文件中，最后替换原来的文件。具体过程如下

- redis调用fork ，现在有父子两个进程
- 子进程根据内存中的数据库快照，往临时文件中写入重建数据库状态的命令
- 父进程继续处理client请求，除了把写命令写入到原来的aof文件中。同时把收到的写命令缓存起来。这样就能保证如果子进程重写失败的话并不会出问题。
- 当子进程把快照内容写入已命令方式写到临时文件中后，子进程发信号通知父进程。然后父进程把缓存的写命令也写入到临时文件。
- 现在父进程可以使用临时文件替换老的aof文件，并重命名，后面收到的写命令也开始往新的aof文件中追加。

需要注意到是重写aof文件的操作，并没有读取旧的aof文件，而是将整个内存中的数据库内容用命令的方式重写了一个新的aof文件,这点和快照有点类似。

优势

- 使用 AOF 持久化会让 Redis 变得非常耐久（much more durable）：你可以设置不同的 fsync 策略，比如无 fsync ，每秒钟一次 fsync ，或者每次执行写入命令时 fsync 。 AOF 的默认策略为每秒钟 fsync 一次，在这种配置下，Redis 仍然可以保持良好的性能，并且就算发生故障停机，也最多只会丢失一秒钟的数据（ fsync 会在后台线程执行，所以主线程可以继续努力地处理命令请求）。
- AOF 文件是一个只进行追加操作的日志文件（append only log）， 因此对 AOF 文件的写入不需要进行 seek ， 即使日志因为某些原因而包含了未写入完整的命令（比如写入时磁盘已满，写入中途停机，等等）， redis-check-aof 工具也可以轻易地修复这种问题。

Redis 可以在 AOF 文件体积变得过大时，自动地在后台对 AOF 进行重写： 重写后的新 AOF 文件包含了恢复当前数据集所需的最小命令集合。 整个重写操作是绝对安全的，因为 Redis 在创建新 AOF 文件的过程中，会继续将命令追加到现有的 AOF 文件里面，即使重写过程中发生停机，现有的 AOF 文件也不会丢失。 而一旦新 AOF 文件创建完毕，Redis 就会从旧 AOF 文件切换到新 AOF 文件，并开始对新 AOF 文件进行追加操作。

- AOF 文件有序地保存了对数据库执行的所有写入操作， 这些写入操作以 Redis 协议的格式保存， 因此 AOF 文件的内容非常容易被人读懂， 对文件进行分析（parse）也很轻松。 导出（export） AOF 文件也非常简单： 举个例子， 如果你不小心执行了 FLUSHALL 命令， 但只要 AOF 文件未被重写， 那么只要停止服务器， 移除 AOF 文件末尾的 FLUSHALL 命令， 并重启 Redis ， 就可以将数据集恢复到 FLUSHALL 执行之前的状态。

劣势

- 对于相同的数据集来说，AOF 文件的体积通常要大于 RDB 文件的体积。
- 根据所使用的 fsync 策略，AOF 的速度可能会慢于 RDB 。 在一般情况下， 每秒 fsync 的性能依然非常高， 而关闭 fsync 可以让 AOF 的速度和 RDB 一样快， 即使在高负荷之下也是如此。 不过在处理巨大的写入载入时，RDB 可以提供更有保证的最大延迟时间（latency）。
- AOF 在过去曾经发生过这样的 bug ： 因为个别命令的原因，导致 AOF 文件在重新载入时，无法将数据集恢复成保存时的原样。（举个例子，阻塞命令 BRPOPLPUSH 就曾经引起过这样的 bug 。）测试套件里为这种情况添加了测试：它们会自动生成随机的、复杂的数据集， 并通过重新载入这些数据来确保一切正常。 虽然这种 bug 在 AOF 文件中并不常见， 但是对比来说， RDB 几乎是不可能出现这种 bug 的。

抉择

一般来说， 如果想达到足以媲美 PostgreSQL 的数据安全性， 你应该同时使用两种持久化功能。

如果你非常关心你的数据， 但仍然可以承受数分钟以内的数据丢失， 那么你可以只使用 RDB 持久化。
其余情况我个人喜好选择AOF

三、

1. Snapshotting:
缺省情况下，Redis会将数据集的快照dump到dump.rdb文件中。此外，我们也可以通过配置文件来修改Redis服务器dump快照的频率，在打开6379.conf文件之后，我们搜索save，可以看到下面的配置信息：
save 900 1 #在900秒(15分钟)之后，如果至少有1个key发生变化，则dump内存快照。
save 300 10 #在300秒(5分钟)之后，如果至少有10个key发生变化，则dump内存快照。
save 60 10000 #在60秒(1分钟)之后，如果至少有10000个key发生变化，则dump内存快照。

2. Dump快照的机制：
1). Redis先fork子进程。
2). 子进程将快照数据写入到临时RDB文件中。
3). 当子进程完成数据写入操作后，再用临时文件替换老的文件。

5.4.3. AOF文件：
上面已经多次讲过，RDB的快照定时dump机制无法保证很好的数据持久性。如果我们的应用确实非常关注此点，我们可以考虑使用Redis中的AOF机制。对于Redis服务器而言，其缺省的机制是RDB，如果需要使用AOF，则需要修改配置文件中的以下条目：
将appendonly no改为appendonly yes
从现在起，Redis在每一次接收到数据修改的命令之后，都会将其追加到AOF文件中。在Redis下一次重新启动时，需要加载AOF文件中的信息来构建最新的数据到内存中。

5.4.5. AOF的配置：
在Redis的配置文件中存在三种同步方式，它们分别是：
appendfsync always #每次有数据修改发生时都会写入AOF文件。
appendfsync everysec #每秒钟同步一次，该策略为AOF的缺省策略。
appendfsync no #从不同步。高效但是数据不会被持久化。

5.4.6. 如何修复坏损的AOF文件：
1). 将现有已经坏损的AOF文件额外拷贝出来一份。

2). 执行"redis-check-aof --fix <filename>"命令来修复坏损的AOF文件。
3). 用修复后的AOF文件重新启动Redis服务器。

5.4.7. Redis的数据备份：
在Redis中我们可以通过copy的方式在线备份正在运行的Redis数据文件。这是因为RDB文件一旦被生成之后就不会再被修改。Redis每次都是将最新的数据dump到一个临时文件中，之后在利用rename函数原子性的将临时文件改名为原有的数据文件名。因此我们可以说，在任意时刻copy数据文件都是安全的和一致的。鉴于此，我们就可以通过创建cron job的方式定时备份Redis的数据文件，并将备份文件copy到安全的磁盘介质中。

5.5、立即写入

```
// 立即保存，同步保存
public static void syncSave() throws Exception{
 Jedis jedis=new Jedis("127.0.0.1",6379);
 for (int i = 0; i <1000; i++) {
 jedis.set("key"+i, "Hello"+i);
 System.out.println("设置key"+i+"的数据到redis");
 Thread.sleep(2);
 }
 // 执行保存，会在服务器下生成一个dump.rdb数据库文件
  jedis.save();
  jedis.close();
  System.out.println("写入完成");
}
```

运行结果：

```
设置key990的数据到redis
设置key991的数据到redis
设置key992的数据到redis
设置key993的数据到redis
设置key994的数据到redis
设置key995的数据到redis
设置key996的数据到redis
设置key997的数据到redis
设置key998的数据到redis
设置key999的数据到redis
写入完成
```

这里的save方法是同步的，没有写入完成前不执行后面的代码。

5.6、异步写入

```
// 异步保存
public static void asyncSave() throws Exception{
 Jedis jedis=new Jedis("127.0.0.1",6379);
 for (int i = 0; i <1000; i++) {
 jedis.set("key"+i, "Hello"+i);
 System.out.println("设置key"+i+"的数据到redis");
 Thread.sleep(2);
 }
 // 执行异步保存，会在服务器下生成一个dump.rdb数据库文件
  jedis.bgsave();
  jedis.close();
  System.out.println("写入完成");
}
```

如果数据量非常大，要保存的内容很多，建议使用bgsave，如果内容少则可以使用save方法。关于各方式的比较源自网友的博客。

1、Redis的第一个持久化策略：RDB快照

Redis支持将当前数据的快照存成一个数据文件的持久化机制。而一个持续写入的数据库如何生成快照呢。Redis借助了fork命令的copy on write机制。在生成快照时，将当前进程fork出一个子进程，然后在子进程中循环所有的数据，将数据写成为RDB文件。

我们可以通过Redis的save指令来配置RDB快照生成的时机，比如你可以配置当10分钟以内有100次写入就生成快照，也可以配置当1小时内有1000次写入就生成快照，也可以多个规则一起实施。这些规则的定义就在Redis的配置文件中，你也可以通过Redis的CONFIG SET命令在Redis运行时设置规则，不需要重启Redis。

Redis的RDB文件不会坏掉，因为其写操作是在一个新进程中进行的，当生成一个新的RDB文件时，Redis生成的子进程会先将数据写到一个临时文件中，然后通过原子性rename系统调用将临时文件重命名为RDB文件，这样在任何时候出现故障，Redis的RDB文件都总是可用的。

同时，Redis的RDB文件也是Redis主从同步内部实现中的一环。

但是，我们可以很明显的看到，**RDB有它的不足，就是一旦数据库出现问题，那么我们的RDB文件中保存的数据并不是全新的**，从上次RDB文件生成到 Redis停机这段时间的数据全部丢掉了。在某些业务下，这是可以忍受的，我们也推荐这些业务使用RDB的方式进行持久化，因为开启RDB的代价并不高。 但是对于另外一些对数据安全性要求极高的应用，无法容忍数据丢失的应用，RDB就无能为力了，所以Redis引入了另一个重要的持久化机制：AOF日志。

2、Redis的第二个持久化策略：AOF日志

AOF日志的全称是Append Only File，从名字上我们就能看出来，它是一个追加写入的日志文件。与一般数据库不同的是，**AOF文件是可识别的纯文本，它的内容就是一个个的Redis标准命令**。比如我们进行如下实验，使用Redis2.6 版本，在启动命令参数中设置开启AOF功能：

```
./redis-server --appendonly yes
```

然后我们执行如下的命令：

```
redis 127.0.0.1:6379> set key1 Hello
OK
redis 127.0.0.1:6379> append key1 " World!"
(integer) 12
redis 127.0.0.1:6379> del key1
(integer) 1
redis 127.0.0.1:6379> del non_existing_key
(integer) 0
```

这时我们查看AOF日志文件，就会得到如下内容：

```
$ cat appendonly.aof
*2
$6
SELECT
$1
0
*3
$3
set
$4
key1
$5
Hello
```

```
*3
$6
append
$4
key1
$7
 World!
*2
$3
del
$4
key1
```

可以看到，写操作都生成了一条相应的命令作为日志。**其中值得注意的是最后一个del命令，它并没有被记录在AOF日志中，这是因为Redis判断出 这个命令不会对当前数据集做出修改**。所以不需要记录这个无用的写命令。另外AOF日志也不是完全按客户端的请求来生成日志的，比如命令 INCRBYFLOAT 在记AOF日志时就被记成一条SET记录，因为浮点数操作可能在不同的系统上会不同，所以为了避免同一份日志在不同的系统上生成不同的数据集，所以这里只将操作后的结果通过SET来记录。

AOF重写

你可以会想，每一条写命令都生成一条日志，那么AOF文件是不是会很大？答案是肯定的，AOF文件会越来越大，**所以Redis又提供了一个功能，叫做AOF rewrite**。其功能就是重新生成一份AOF文件，新的AOF文件中一条记录的操作只会有一次，而不像一份老文件那样，可能记录了对同一个值的多次操作。其生成过程和RDB类似，也是fork一个进程，直接遍历数据，写入新的AOF临时文件。在写入新文件的过程中，所有的写操作日志还是会写到原来老的 AOF文件中，同时还会记录在内存缓冲区中。当重完操作完成后，会将所有缓冲区中的日志一次性写入到临时文件中。然后调用原子性的rename命令用新的 AOF文件取代老的AOF文件。

二、Redis持久化性能是否可靠？

从上面的流程我们能够看到，RDB是顺序IO操作，性能很高。而同时在通过RDB文件进行数据库恢复的时候，也是顺序的读取数据加载到内存中。所以也不会造成磁盘的随机读取错误。

而AOF是一个写文件操作，其目的是将操作日志写到磁盘上，所以它也同样会遇到我们上面说的写操作的5个流程。那么写AOF的操作安全性又有多高呢？实际上这是可以设置的，**在**

Redis中对AOF调用write写入后，何时再调用fsync将其写到磁盘上，通过appendfsync选项来控制，下面appendfsync的三个设置项，安全强度逐渐变强。

1、appendfsync no

当设置appendfsync为no的时候，Redis不会主动调用fsync去将AOF日志内容同步到磁盘，所以这一切就完全依赖于操作系统的调试了。**对大多数Linux操作系统，是每30秒进行一次fsync，将缓冲区中的数据写到磁盘上。**

2、appendfsync everysec

当设置appendfsync为everysec的时候，Redis会默认每隔一秒进行一次fsync调用，将缓冲区中的数据写到磁盘。但是当这一 次的fsync调用时长超过1秒时。Redis会采取延迟fsync的策略，再等一秒钟。也就是在两秒后再进行fsync，这一次的fsync就不管会执行多长时间都会进行。这时候由于在fsync时文件描述符会被阻塞，所以当前的写操作就会阻塞。

所以，**结论就是：在绝大多数情况下，Redis会每隔一秒进行一次fsync。在最坏的情况下，两秒钟会进行一次fsync操作。**

这一操作在大多数数据库系统中被称为group commit，就是组合多次写操作的数据，一次性将日志写到磁盘。

3、appednfsync always

当设置appendfsync为always时，每一次写操作都会调用一次fsync，这时数据是最安全的，当然，由于每次都会执行fsync，所以其性能也会受到影响。

对于pipelining有什么不同？

对于pipelining的操作，其具体过程是客户端一次性发送N个命令，然后等待这N个命令的返回结果被一起返回。通过采用pipilining 就意味着放弃了对每一个命令的返回值确认。由于在这种情况下，N个命令是在同一个执行过程中执行的。所以当设置appendfsync为everysec 时，可能会有一些偏差，因为这N个命令可能执行时间超过1秒甚至2秒。但是可以保证的是，最长时间不会超过这N个命令的执行时间和。

三、和其它数据库的比较

上面操作系统层面的数据安全我们已经讲了很多，其实，不同的数据库在实现上都大同小异。总之，最后的结论就是，在Redis开启AOF的情况下，其单机数据安全性并不比这些成熟的SQL数据库弱。

在数据导入方面的比较

这些持久化的数据有什么用，当然是用于重启后的数据恢复。Redis是一个内存数据库，无论是RDB还是AOF，都只是其保证数据恢复的措施。所以 Redis在利用RDB和AOF进行恢复的时候，都会读取RDB或AOF文件，重新加载到内存中。相对于MySQL等数据库的启动时间来说，会长很多，因为MySQL本来是不需要将数据加载到内存中的。

但是相对来说，MySQL启动后提供服务时，其被访问的热数据也会慢慢加载到内存中，通常我们称之为预热，而在预热完成前，其性能都不会太高。而Redis的好处是一次性将数据加载到内存中，一次性预热。这样只要Redis启动完成，那么其提供服务的速度都是非常快的。

而在利用RDB和利用AOF启动上，其启动时间有一些差别。RDB的启动时间会更短，原因有两个，一是RDB文件中每一条数据只有一条记录，不会像 AOF日志那样可能有一条数据的多次操作记录。所以每条数据只需要写一次就行了。另一个原因是RDB文件的存储格式和Redis数据在内存中的编码格式是一致的，不需要再进行数据编码工作。在CPU消耗上要远小于AOF日志的加载。