加餐4-RDB和AOF文件损坏了咋办？

你好，我是蒋德钧。今天的加餐课程，我来和你聊聊Redis对损坏的RDB和AOF文件的处理方法。

我们知道，Redis为了提升可靠性，可以使用AOF记录操作日志，或者使用RDB保存数据库镜像。AOF文件的记录和RDB文件的保存都涉及写盘操作，但是，如果在写盘过程中发生了错误，就会导致AOF或RDB文件不完整。而Redis使用不完整的AOF或RDB文件，是无法恢复数据库的。那么在这种情况下，我们该怎么处理呢？

实际上，Redis为了应对这个问题，就专门实现了针对AOF和RDB文件的完整性检测工具，也就是redis-check-aof和redis-check-rdb两个命令。今天这节课，我就来给你介绍下这两个命令的实现以及它们的作用。学完这节课后，如果你再遇到无法使用AOF或RDB文件恢复Redis数据库时，你就可以试试这两个命令。

接下来，我们先来看下AOF文件的检测和修复。

AOF文件检测与修复

要想掌握AOF文件的检测和修复，我们首先需要了解下，AOF文件的内容格式是怎样的。

AOF文件的内容格式

AOF文件记录的是Redis server运行时收到的操作命令。当Redis server往AOF文件中写入命令时，它会按照RESP 2协议的格式来记录每一条命令。当然，如果你使用了Redis 6.0版本，那么Redis会采用RESP 3协议。我在第一季的时候，曾经给你介绍过Redis客户端和server之间进行交互的RESP 2协议，你可以再去回顾下。

这里我们就简单来说下，RESP 2协议会为每条命令或每个数据进行编码，在每个编码结果后面增加一个**换行符“\r\n”**，表示一次编码结束。一条命令通常会包含命令名称和命令参数，RESP 2协议会使用数组编码类型来对命令进行编码。比如，当我们在AOF文件中记录一条SET course redis-code命令时，Redis会分别为命令本身SET、key和value的内容course和redis-code进行编码，如下所示：

```
*3
$3
SET
$6
course
$10
redis-code
```

注意，编码结果中以“*”或是“$”开头的内容很重要，因为“*”开头的编码内容中的数值，表示了一条命令操作包含的参数个数。当然，命令本身也被算为是一个参数，记录在了这里的数值当中。而“$”开头的编码内容中的数字，表示紧接着的字符串的长度。

在刚才介绍的例子中，“*3”表示接下来的命令有三个参数，这其实就对应了SET命令本身，键值对的key“course”和value“redis-code”。而“$3”表示接下来的字符串长度是3，这就对应了SET命令这个字符串本身的长度是3。

好了，了解了AOF文件中命令的记录方式之后，我们再来学习AOF文件的检测就比较简单了。这是因为RESP 2协议会将命令参数个数、字符串长度这些信息，通过编码也记录到AOF文件中，redis-check-aof命令在检测AOF文件时，就可以利用这些信息来判断一个命令是否完整记录了。

AOF文件的检测过程

AOF文件的检测是在redis-check-aof.c文件中实现的，这个文件的入口函数是**redis_check_aof_main**。在这个函数中，我们可以看到AOF检测的主要逻辑，简单来说可以分成三步。

首先，redis_check_aof_main会调用fopen函数打开AOF文件，并调用redis_fstat函数获取AOF文件的大小size。

其次，redis_check_aof_main会调用process函数，实际检测AOF文件。process函数会读取AOF文件中的每一行，并检测是否正确，我一会儿会给你具体介绍这个函数。这里你要知道，process函数在检测AOF文件时，如果发现有不正确或是不完整的命令操作，它就会停止执行，并且返回已经检测为正确的AOF文件位置。

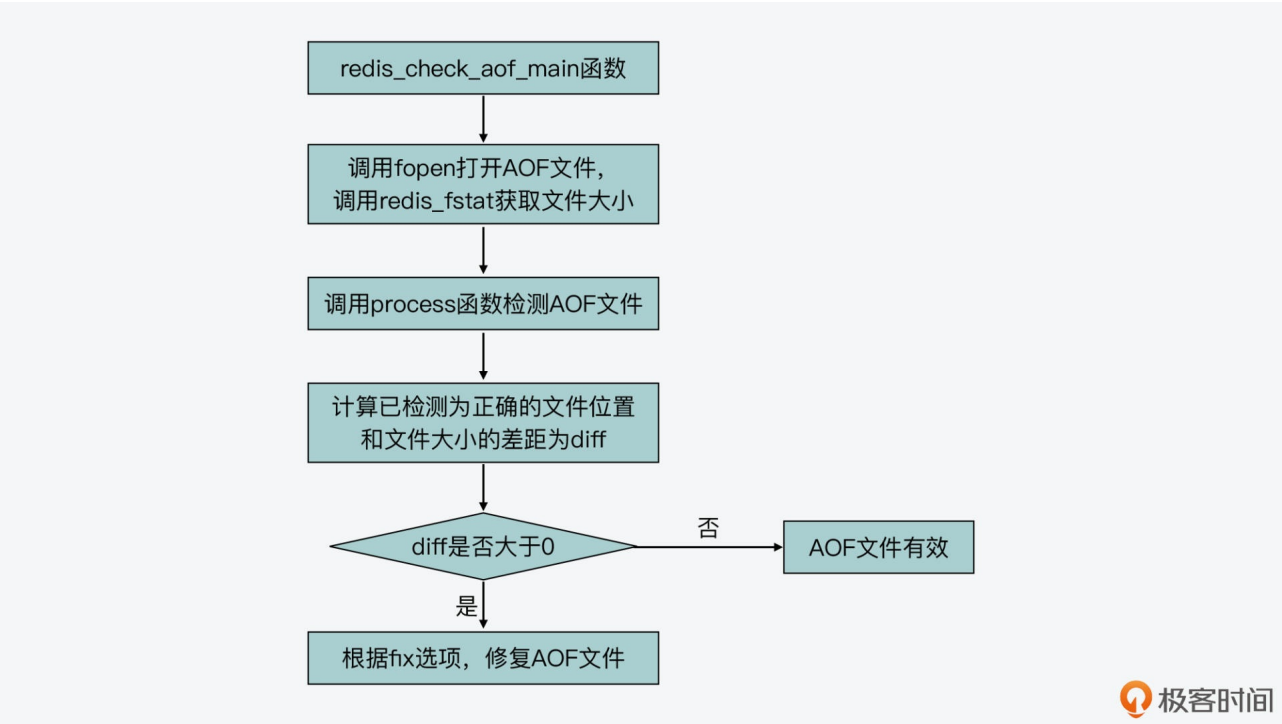
最后，redis_check_aof_main会根据process函数的返回值，来判断已经检测为正确的文件大小是否等于AOF文件本身大小。如果不等于的话，redis_check_aof_main函数会根据redis-check-aof命令执行时的fix选项，来决定是否进行修复。

下面的代码就展示了刚才介绍的AOF文件检测的基本逻辑，你可以看下。

```
//调用fopen打开AOF文件
FILE *fp = fopen(filename,"r+");

//调用redis_fstat获取文件大小size
struct redis_stat sb;
if (redis_fstat(fileno(fp),&sb) == -1) { …}
off_t size = sb.st_size;
…
off_t pos = process(fp); //调用process检测AOF文件，返回AOF文件中已检测为正确的位置
off_t diff = size-pos;  //获取已检测正确的文件位置和文件大小的差距
printf("AOF analyzed: size=%lld, ok_up_to=%lld, diff=%lld\n", (long long) size, (long long) pos, (long long
if (diff > 0) { …} //如果检测正确的文件位置还小于文件大小，可以进行修复
else {
   printf("AOF is valid\n");
}
```

你也可以参考这里我给出的示意图：



OK，了解了AOF文件检测的基本逻辑后，我们再来看下刚才介绍的process函数。

process函数的主体逻辑，其实就是**执行一个while(1)的循环流程**。在这个循环中，process函数首先会调用readArgc函数，来获取每条命令的参数个数，如下所示：

```
int readArgc(FILE *fp, long *target) {
    return readLong(fp,'*',target);
}
```

readArgc函数会进一步调用readLong函数，而readLong函数是用来读取“*”“$”开头的这类编码结果的。readLong函数的原型如下所示，它的参数fp是指向AOF文件的指针，参数prefix是编码结果的开头前缀，比如“*”或“$”，而参数target则用来保存编码结果中的数值。

```
int readLong(FILE *fp, char prefix, long *target)
```

这样一来，process函数通过调用readArgc就能获得命令参数个数了。不过，**如果readLong函数从文件中读到的编码结果前缀，和传入的参数prefix不一致，那么它就会报错**，从而让process函数检测到AOF文件中不正确的命令。

然后，process函数会根据命令参数的个数执行一个for循环，调用readString函数逐一读取命令的参数。而readString函数会先调用readLong函数，读取以“$”开头的编码结果，这也是让readString函数获得要读取的字符串的长度。

紧接着，readString函数就会调用readBytes函数，从AOF文件中读取相应长度的字节。不过，**如果此时readBytes函数读取的字节长度，和readLong获得的字符串长度不一致，那么就表明AOF文件不完整**，process函数也会停止读取命令参数，并进行报错。

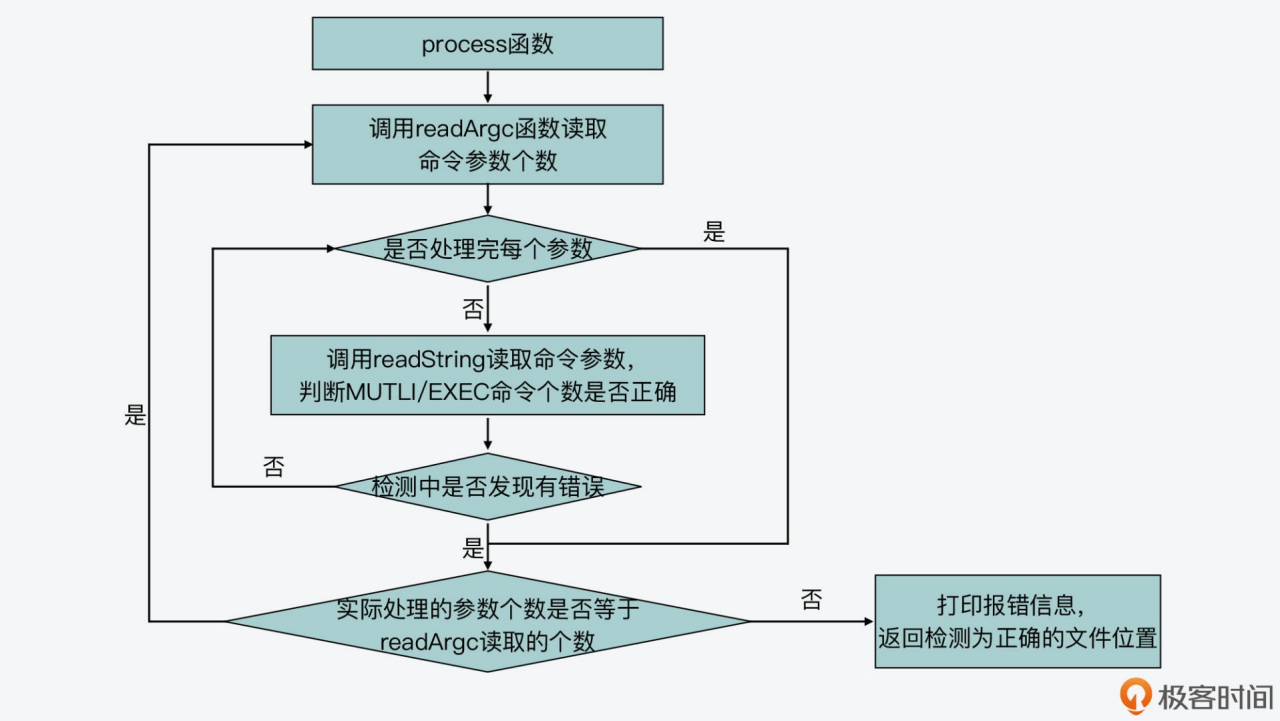
下面的代码展示了readBytes函数的主要逻辑，你可以看下。

```
int readBytes(FILE *fp, char *target, long length) {
    …
    real = fread(target,1,length,fp);  //从AOF文件中读取length长度的字节
    if (real != length) {  //如果实际读取的字节不等于length，则报错
        ERROR("Expected to read %ld bytes, got %ld bytes",length,real);
        return 0;
 }
 …}
```

这里你需要注意的是，如果process函数实际读取的命令参数个数，小于readArgc函数获取的参数个数，那么process函数也会停止继续检测AOF文件。

最后，process函数会打印检测AOF文件过程中遇到的错误，并返回已经检测正确的文件位置。

现在，我们就了解了AOF文件检测过程中的主要函数process，你也可以看下这里我给出的示意图。



那么，就像我刚才给你介绍的，process函数返回已经检测的文件位置后，redis_check_aof_main函数会判断是否已经完成整个AOF文件的检查。如果没有的话，那么就说明在检测AOF文件的过程中，发生了某些错误，此时redis_check_aof_main函数会判断redis-check-aof命令执行时，**是否带了“–fix”选项**。

而如果有这一选项，redis_check_aof_main函数就会开始执行修复操作。接下来，我们就来看下AOF文件的修复。

AOF文件的修复

AOF文件的修复其实实现得很简单，它就是从AOF文件已经检测正确的位置开始，调用**ftruncate函数**执行截断操作。

这也就是说，redis-check-aof命令在对AOF文件进行修复时，一旦检测到有不完整或是不正确的操作命令时，它就只保留了从AOF文件开头到出现不完整，或是不正确的操作命令位置之间的内容，而不完整或是不正确的操作命令，以及其后续的内容就被直接删除了。

所以，我也给你一个**小建议**，当你使用redis-check-aof命令修复AOF文件时，最好是把原来的AOF文件备份一份，以免出现修复后原始AOF文件被截断，而带来的操作命令缺失问题。

下面的代码展示了redis_check_aof_main函数中对AOF文件进行修复的基本逻辑，你可以看下。

```
if (fix) {
   …
   if (ftruncate(fileno(fp), pos) == -1) {
      printf("Failed to truncate AOF\n");
      exit(1);
    } else {
      printf("Successfully truncated AOF\n");
 }
```

好了，到这里，你就了解了AOF文件的检测和修复。接下来，我们再来看下RDB文件的检测。

RDB文件检测

RDB文件检测是在redis-check-rdb.c文件中实现的，这个文件的入口函数是**redis_check_rdb_main**。它会调用redis_check_rdb函数，来完成检测。

这里你要知道，和AOF文件用RESP 2协议格式记录操作命令不一样，RDB文件是Redis数据库内存中的内容在某一时刻的快照，它包括了文件头、键值对数据部分和文件尾三个部分。而且，**RDB文件是通过一定的编码来记录各种属性信息和键值对的**。我在第18讲中给你介绍过RDB文件的组成和编码方式，建议你可以再去回顾下，毕竟只有理解了RDB文件的组成和编码，你才能更好地理解redis_check_rdb函数是如何对RDB文件进行检测的。

其实，redis_check_rdb函数检测RDB文件的逻辑比较简单，就是根据RDB文件的组成，逐一检测文件头的魔数、文件头中的属性信息、文件中的键值对数据，以及最后的文件尾校验和。下面，我们就来具体看下。

首先，redis_check_rdb函数**先读取RDB文件头的9个字节**，这是对应RDB文件的魔数，其中包含了“REDIS”字符串和RDB的版本号。redis_check_rdb函数会检测“REDIS”字符串和版本号是否正确，如下所示：

```
int redis_check_rdb(char *rdbfilename, FILE *fp) {
…
if (rioRead(&rdb,buf,9) == 0) goto eoferr;  //读取文件的头9个字节
buf[9] = '\0';
if (memcmp(buf,"REDIS",5) != 0) { //将前5个字节和“REDIS”比较，如果有误则报错
   rdbCheckError("Wrong signature trying to load DB from file");
   goto err;
}
rdbver = atoi(buf+5); //读取版本号
if (rdbver < 1 || rdbver > RDB_VERSION) { //如果有误，则报错
```

```
    rdbCheckError("Wrong signature trying to load DB from file");
    goto err;
}
…}
```

然后，redis_check_rdb函数会**执行一个while(1)循环流程**，在这个循环中，它会按照RDB文件的格式依次读取其中的内容。我在第18讲中也给你介绍过，RDB文件在文件头魔数后记录的内容，包括了Redis的属性、数据库编号、全局哈希表的键值对数量等信息。而在实际记录每个键值对之前，RDB文件还要记录键值对的过期时间、LRU或LFU等信息，这些信息都是按照操作码和实际内容来组织的。

比如，RDB文件中要记录Redis的版本号，它就会用十六进制的FA表示操作码，表明接下来的内容就是Redis版本号；当它要记录使用的数据库时，它会使用十六进制的FE操作码来表示。

好了，了解了RDB文件的这种内容组织方式后，我们接着来看下**redis_check_rdb函数**，它在循环流程中，就是先调用rdbLoadType函数读取操作码，然后，使用多个条件分支来匹配读取的操作码，并根据操作码含义读取相应的操作码内容。

以下代码就展示了这部分的操作码读取和分支判断逻辑，而对于每个分支中读取操作码的具体实现，你可以去仔细阅读一下源码。

```
if ((type = rdbLoadType(&rdb)) == -1) goto eoferr;
if (type == RDB_OPCODE_EXPIRETIME) { …} //表示过期时间的操作码
else if (type == RDB_OPCODE_EXPIRETIME_MS) {…} //表示毫秒记录的过期时间操的作码
else if (type == RDB_OPCODE_FREQ) {…} //表示LFU访问频率的操作码
else if (type == RDB_OPCODE_IDLE) {…} //表示LRU空闲时间的操作码
else if (type == RDB_OPCODE_EOF) {…} //表示RDB文件结束的操作码
else if (type == RDB_OPCODE_SELECTDB) {…} //表示数据库选择的操作码
else if (type == RDB_OPCODE_RESIZEDB) {…} //表示全局哈希表键值对数量的操作码
else if (type == RDB_OPCODE_AUX) {…} //表示Redis属性的操作码
else {…} //无法识别的操作码
```

这样，在判断完操作码之后，redis_check_rdb函数就会调用rdbLoadStringObject函数，读取键值对的key，以及调用rdbLoadObject函数读取键值对的value。

最后，当读取完所有的键值对后，redis_check_rdb函数就会读取文件尾的校验和信息，然后验证校验和是否正确。

到这里，整个RDB文件的检测就完成了。在这个过程中，如果redis_check_rdb函数发现RDB文件有错误，就会将错误在文件中出现的位置、当前的检测操作、检测的键值对的key等信息打印出来，以便我们自行进一步检查。

小结

今天这节课，我带你了解了检测AOF文件和RDB文件正确性和完整性的两个命令，redis-check-aof和redis-check-rdb，以及它们各自的实现过程。

对于**redis-check-aof命令**来说，它是根据AOF文件中记录的操作命令格式，逐一读取命令，并根据命令参数个数、参数字符串长度等信息，进行命令正确性和完整性的判断。

对于**redis-check-rdb命令**来说，它的实现逻辑较为简单，也就是按照RDB文件的组织格式，依次读取RDB文件头、数据部分和文件尾，并在读取过程中判断内容是否正确，并进行报错。

事实上，Redis server在运行时，遇到故障而导致AOF文件或RDB文件没有记录完整，这种情况有时是不可避免的。当了解了redis-check-aof命令的实现后，我们就知道它可以提供出现错误或不完整命令的文件位置，并且，它本身提供了修复功能，可以从出现错误的文件位置处截断后续的文件内容。不过，如果我们不想通过截断来修复AOF文件的话，也可以尝试人工修补。

而在了解了redis-check-rdb命令的实现后，我们知道它可以发现RDB文件的问题所在。不过，redis-check-rdb命令目前并没有提供修复功能。所以如果我们需要修复的话，就只能通过人工自己来修复了。

每课一问

redis_check_aof_main函数是检测AOF文件的入口函数，但是它还会调用检测RDB文件的入口函数redis_check_rdb_main，那么你能找到这部分代码，并通过阅读说说它的作用吗？

精选留言：

- Ethan New 2021-10-05 12:22:57
  redis 4.0后提供了aof rewrite的功能，重写后的aof文件既有RDB格式的数据也有AOF格式的命令，redis_check_aof_main调用redis_check_rdb_main就是为了检测文件中RDB格式的数据。