# 18-如何生成和解读RDB文件？

你好，我是蒋德钧。

从今天这节课开始，我们又将进入一个新的模块，也就是可靠性保证模块。在这个模块中，我会先带你了解Redis数据持久化的实现，其中包括Redis内存快照RDB文件的生成方法，以及AOF日志的记录与重写。了解了这部分内容，可以让你掌握RDB文件的格式，学习到如何制作数据库镜像，并且你也会进一步掌握AOF日志重写对Redis性能的影响。

然后，我还会围绕Redis主从集群的复制过程、哨兵工作机制和故障切换这三个方面，来给你介绍它们的代码实现。因为我们知道，主从复制是分布式数据系统保证可靠性的一个重要机制，而Redis就给我们提供了非常经典的实现，所以通过学习这部分内容，你就可以掌握到在数据同步实现过程中的一些关键操作和注意事项，以免踩坑。

好，那么今天这节课，我们就先从RDB文件的生成开始学起。下面呢，我先带你来了解下RDB创建的入口函数，以及调用这些函数的地方。

## RDB创建的入口函数和触发时机

Redis源码中用来创建RDB文件的函数有三个，它们都是在rdb.c文件中实现的，接下来我就带你具体了解下。

- **rdbSave函数**

这是Redis server在本地磁盘创建RDB文件的入口函数。它对应了Redis的save命令，会在save命令的实现函数saveCommand（在rdb.c文件中）中被调用。而rdbSave函数最终会调用rdbSaveRio函数（在rdb.c文件中）来实际创建RDB文件。rdbSaveRio函数的执行逻辑就体现了RDB文件的格式和生成过程，我稍后向你介绍。

- **rdbSaveBackground函数**

这是Redis server使用后台子进程方式，在本地磁盘创建RDB文件的入口函数。它对应了Redis的bgsave命令，会在bgsave命令的实现函数bgsaveCommand（在rdb.c文件中）中被调用。这个函数会调用fork创建一个子进程，让子进程调用rdbSave函数来继续创建RDB文件，而父进程，也就是主线程本身可以继续处理客户端请求。

下面的代码展示了rdbSaveBackground函数创建子进程的过程，你可以看下。我在第12讲中也向你介绍过fork的使用，你可以再回顾下。

```
int rdbSaveBackground(char *filename, rdbSaveInfo *rsi) {
...
if ((childpid = fork()) == 0) {  //子进程的代码执行分支
   ...
   retval = rdbSave(filename,rsi);  //调用rdbSave函数创建RDB文件
   ...
   exitFromChild((retval == C_OK) ? 0 : 1);  //子进程退出
} else {
   ...  //父进程代码执行分支
}
```

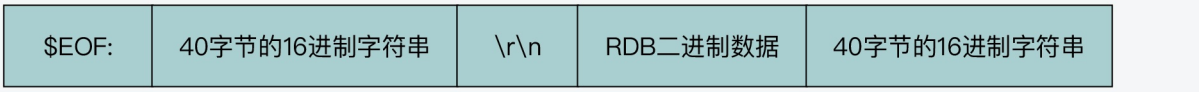
```
}
```

- **rdbSaveToSlavesSockets函数**

这是Redis server在采用不落盘方式传输RDB文件进行主从复制时，创建RDB文件的入口函数。它会被startBgsaveForReplication函数调用（在replication.c文件中）。而startBgsaveForReplication函数会被replication.c文件中的syncCommand函数和replicationCron函数调用，这对应了Redis server执行主从复制命令，以及周期性检测主从复制状态时触发RDB生成。

和rdbSaveBackground函数类似，rdbSaveToSlavesSockets函数也是通过fork创建子进程，让子进程生成RDB。不过和rdbSaveBackground函数不同的是，rdbSaveToSlavesSockets函数是通过网络**以字节流的形式，直接发送RDB文件的二进制数据给从节点**。

而为了让从节点能够识别用来同步数据的RDB内容，rdbSaveToSlavesSockets函数调用**rdbSaveRioWithEOFMark函数**（在rdb.c文件中），在RDB二进制数据的前后加上了标识字符串，如下图所示：






以下代码也展示了rdbSaveRioWithEOFMark函数的基本执行逻辑。你可以看到，它除了写入前后标识字符串之外，还是会调用rdbSaveRio函数实际生成RDB内容。

```
int rdbSaveRioWithEOFMark(rio *rdb, int *error, rdbSaveInfo *rsi) {
...
getRandomHexChars(eofmark,RDB_EOF_MARK_SIZE); //随机生成40字节的16进制字符串，保存在eofmark中，宏定义RDB_EOF_MARK_
if (rioWrite(rdb,"$EOF:",5) == 0) goto werr;  //写入$EOF
if (rioWrite(rdb,eofmark,RDB_EOF_MARK_SIZE) == 0) goto werr; //写入40字节的16进制字符串eofmark
if (rioWrite(rdb,"\r\n",2) == 0) goto werr; //写入\r\n
if (rdbSaveRio(rdb,error,RDB_SAVE_NONE,rsi) == C_ERR) goto werr; //生成RDB内容
if (rioWrite(rdb,eofmark,RDB_EOF_MARK_SIZE) == 0) goto werr; //再次写入40字节的16进制字符串eofmark
...
}
```

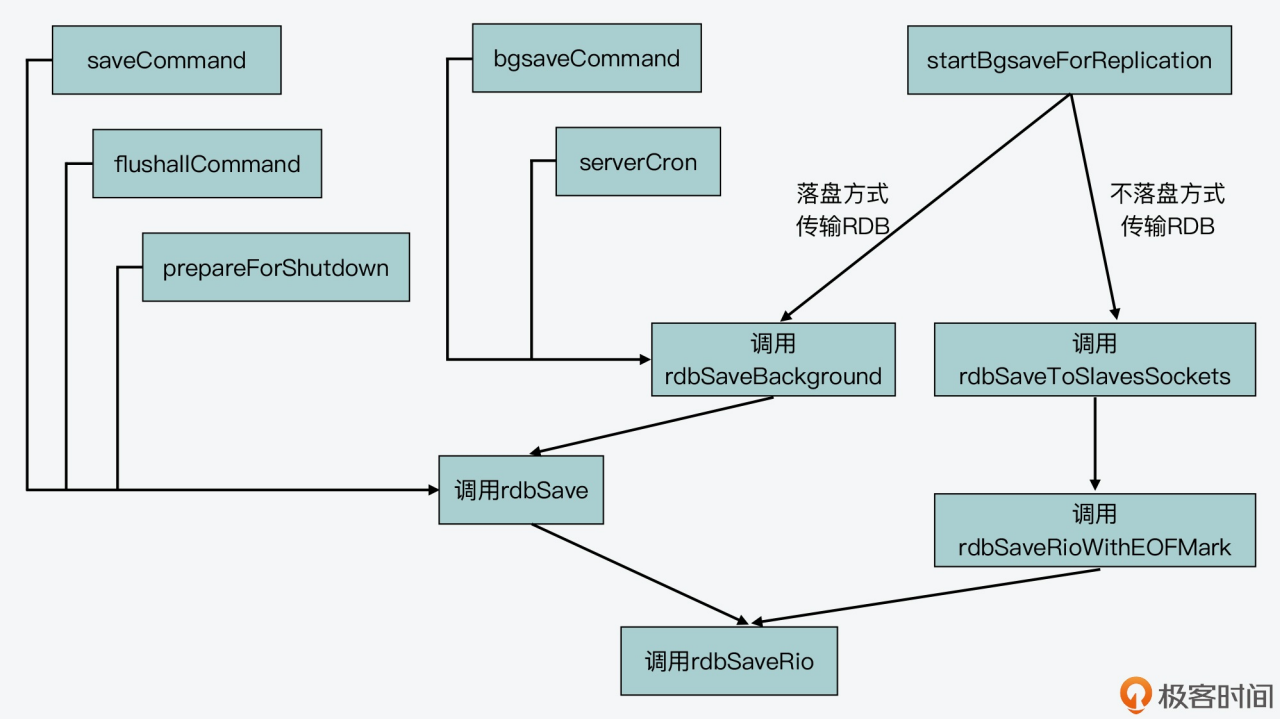
好了，了解了RDB文件创建的三个入口函数后，我们也看到了，RDB文件创建的三个时机，分别是save命令执行、bgsave命令执行以及主从复制。那么，**除了这三个时机外，在Redis源码中，还有哪些地方会触发RDB文件创建呢？**

实际上，因为rdbSaveToSlavesSockets函数只会在主从复制时调用，所以，我们只要通过在Redis源码中查找**rdbSave、rdbSaveBackground**这两个函数，就可以了解触发RDB文件创建的其他时机。

那么经过查找，我们可以发现在Redis源码中，rdbSave还会在**flushallCommand函数**（在db.c文件中）、**prepareForShutdown函数**（在server.c文件中）中被调用。这也就是说，Redis在执行flushall命令以及正常关闭时，会创建RDB文件。

对于rdbSaveBackground函数来说，它除了在执行bgsave命令时被调用，当主从复制采用落盘文件方式传输RDB时，它也会被startBgsaveForReplication函数调用。此外，Redis server运行时的周期性执行函数serverCron（在server.c文件中），也会调用rdbSaveBackground函数来创建RDB文件。

为了便于你掌握RDB文件创建的整体情况，我画了下面这张图，展示了Redis源码中创建RDB文件的函数调用关系，你可以看下。




好了，到这里，你可以看到，实际最终生成RDB文件的函数是rdbSaveRio。所以接下来，我们就来看看rdbSaveRio函数的执行过程。同时，我还会给你介绍RDB文件的格式是如何组织的。

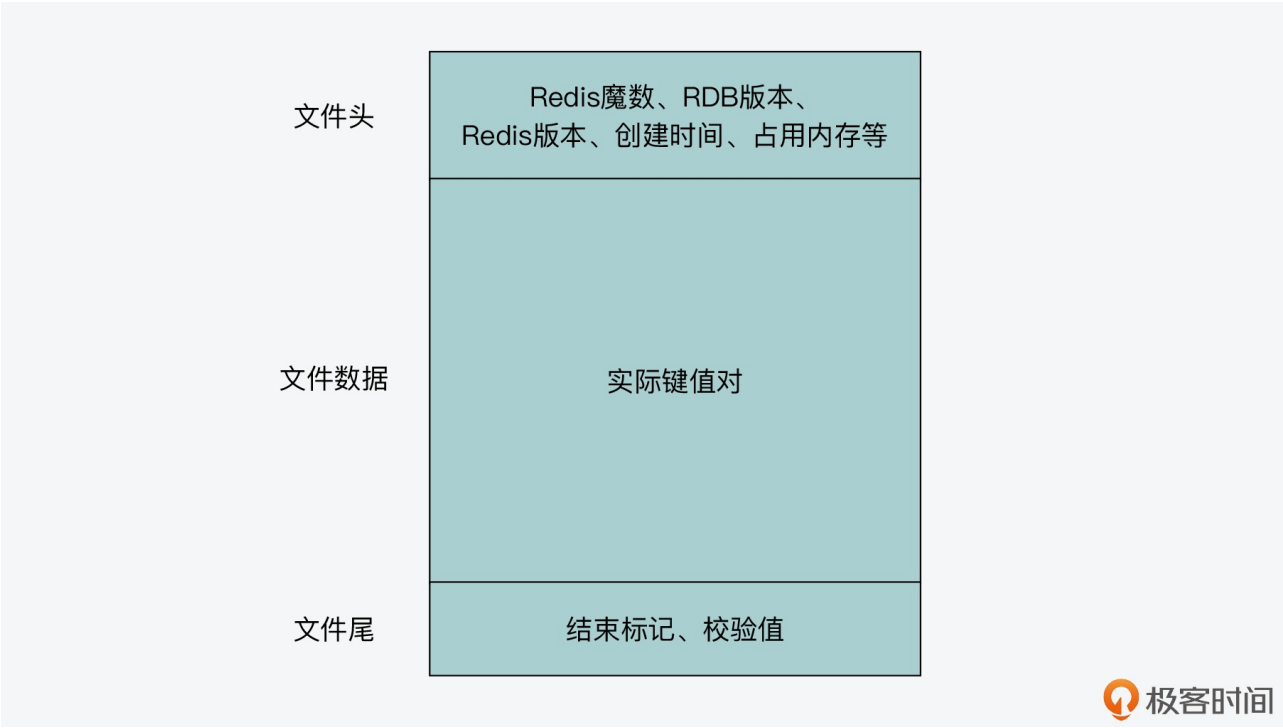
## RDB文件是如何生成的?

不过在了解rdbSaveRio函数具体是如何生成RDB文件之前，你还需要先了解下RDB文件的基本组成部分。这样，你就可以按照RDB文件的组成部分，依次了解rdbSaveRio函数的执行逻辑了。

那么，一个RDB文件主要是由三个部分组成的。

- **文件头**：这部分内容保存了Redis的魔数、RDB版本、Redis版本、RDB文件创建时间、键值对占用的内存大小等信息。
- **文件数据部分**：这部分保存了Redi数据库实际的所有键值对。
- **文件尾**：这部分保存了RDB文件的结束标识符，以及整个文件的校验值。这个校验值用来在Redis server加载RDB文件后，检查文件是否被篡改过。

下图就展示了RDB文件的组成，你可以看下。




好，接下来，我们就来看看rdbSaveRio函数是如何生成RDB文件中的每一部分的。这里，为了方便你理解RDB文件格式以及文件内容，你可以先按照如下步骤准备一个RDB文件。

第一步，在你电脑上Redis的目录下，启动一个用来测试的Redis server，可以执行如下命令：

```
./redis-server
```

第二步，执行flushall命令，清空当前的数据库：

```
./redis-cli flushall
```

第三步，使用redis-cli登录刚启动的Redis server，执行set命令插入一个String类型的键值对，再执行hmset命令插入一个Hash类型的键值对。执行save命令，将当前数据库内容保存到RDB文件中。这个过程如下所示：

```
127.0.0.1:6379>set hello redis
OK
127.0.0.1:6379>hmset userinfo uid 1 name zs age 32
OK
127.0.0.1:6379> save
OK
```

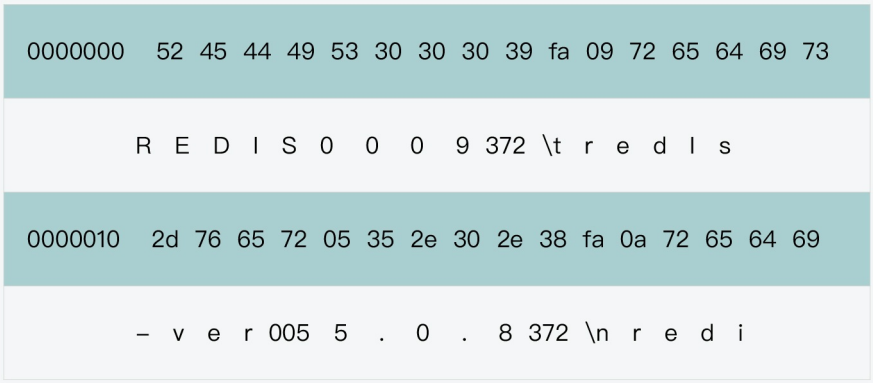
好了，到这里，你就可以在刚才执行redis-cli命令的目录下，找见刚生成的RDB文件，文件名应该是dump.rdb。

不过，因为RDB文件实际是一个二进制数据组成的文件，所以如果你使用一般的文本编辑软件，比如Linux系统上的Vim，在打开RDB文件时，你会看到文件中都是乱码。所以这里，我给你提供一个小工具，如果你想查看RDB文件中二进制数据和对应的ASCII字符，你可以使用**Linux上的od命令**，这个命令可以用不同进制的方式展示数据，并显示对应的ASCII字符。

比如，你可以执行如下的命令，读取dump.rdb文件，并用十六进制展示文件内容，同时文件中每个字节对应的ASCII字符也会被对应显示出来。

```
od -A x -t x1c -v dump.rdb
```

以下代码展示的就是我用od命令，查看刚才生成的dump.rdb文件后，输出的从文件头开始的部分内容。你可以看到这四行结果中，第一和第三行是用十六进制显示的dump.rdb文件的字节内容，这里每两个十六进制数对应了一个字节。而第二和第四行是od命令生成的每个字节所对应的ASCII字符。




这也就是说，在刚才生成的RDB文件中，如果想要转换成ASCII字符，它的文件头内容其实就已经包含了REDIS的字符串和一些数字，而这正是RDB文件头包含的内容。

那么下面，我们就来看看RDB文件的文件头是如何生成的。

## 生成文件头

就像刚才给你介绍的，RDB文件头的内容首先是**魔数**，这对应记录了RDB文件的版本。在rdbSaveRio函数中，魔数是通过snprintf函数生成的，它的具体内容是字符串“REDIS”，再加上RDB版本的宏定义RDB_VERSION（在rdb.h文件中，值为9）。然后，rdbSaveRio函数会调用rdbWriteRaw函数（在rdb.c文件中），将魔数写入RDB文件，如下所示：

```
snprintf(magic,sizeof(magic),"REDIS%04d",RDB_VERSION);  //生成魔数magic
if (rdbWriteRaw(rdb,magic,9) == -1) goto werr;  //将magic写入RDB文件
```

刚才用来写入魔数的**rdbWriteRaw函数**，它实际会调用rioWrite函数（在rdb.h文件中）来完成写入。而rioWrite函数是RDB文件内容的最终写入函数，它负责根据要写入数据的长度，把待写入缓冲区中的内容写

入RDB。这里，**你需要注意的是**，RDB文件生成过程中，会有不同的函数负责写入不同部分的内容，不过这些函数最终都还是调用rioWrite函数，来完成数据的实际写入的。

好了，当在RDB文件头中写入魔数后，rdbSaveRio函数紧接着会调用**rdbSaveInfoAuxFields函数**，将和Redis server相关的一些属性信息写入RDB文件头，如下所示：

```
if (rdbSaveInfoAuxFields(rdb,flags,rsi) == -1) goto werr; //写入属性信息
```

rdbSaveInfoAuxFields函数是在rdb.c文件中实现的，它会使用键值对的形式，在RDB文件头中记录Redis server的属性信息。下表中列出了RDB文件头记录的一些主要信息，以及它们对应的键和值，你可以看下。

| 属性 | 键 | 值 |
| --- | --- | --- |
| Redis版本信息 | redis-ver | 宏定义REDIS_VERSION |
| Redis server运行平台的架构信息 | redis-bits | 32或64，根据指针大小定 |
| RDB文件的创建时间 | ctime | 调用time获取当前时间 |
| Redis server已使用的内存量 | used-mem | 调用zmalloc_used_memory获取当前内存使用量 |



那么，当属性值为字符串时，rdbSaveInfoAuxFields函数会调用**rdbSaveAuxFieldStrStr函数**写入属性信息；而当属性值为整数时，rdbSaveInfoAuxFields函数会调用**rdbSaveAuxFieldStrInt函数**写入属性信息，如下所示：

```
if (rdbSaveAuxFieldStrStr(rdb,"redis-ver",REDIS_VERSION) == -1) return -1;
if (rdbSaveAuxFieldStrInt(rdb,"redis-bits",redis_bits) == -1) return -1;
if (rdbSaveAuxFieldStrInt(rdb,"ctime",time(NULL)) == -1) return -1;
if (rdbSaveAuxFieldStrInt(rdb,"used-mem",zmalloc_used_memory()) == -1) return -1;
```

这里，无论是rdbSaveAuxFieldStrStr函数还是rdbSaveAuxFieldStrInt函数，**它们都会调用rdbSaveAuxField函数来写入属性值**。rdbSaveAuxField函数是在rdb.c文件中实现的，它会分三步来完成一个属性信息的写入。

**第一步**，它调用rdbSaveType函数写入一个操作码。这个操作码的目的，是用来在RDB文件中标识接下来的内容是什么。当写入属性信息时，这个操作码对应了宏定义RDB_OPCODE_AUX（在rdb.h文件中），值为250，对应的十六进制值为FA。这样一来，就方便我们解析RDB文件了。比如，在读取RDB文件时，如果程序读取到FA这个字节，那么，这就表明接下来的内容是一个属性信息。

这里，**你需要注意的是**，RDB文件使用了多个操作码，来标识文件中的不同内容。它们都是在rdb.h文件中

定义的，下面的代码中展示了部分操作码，你可以看下。

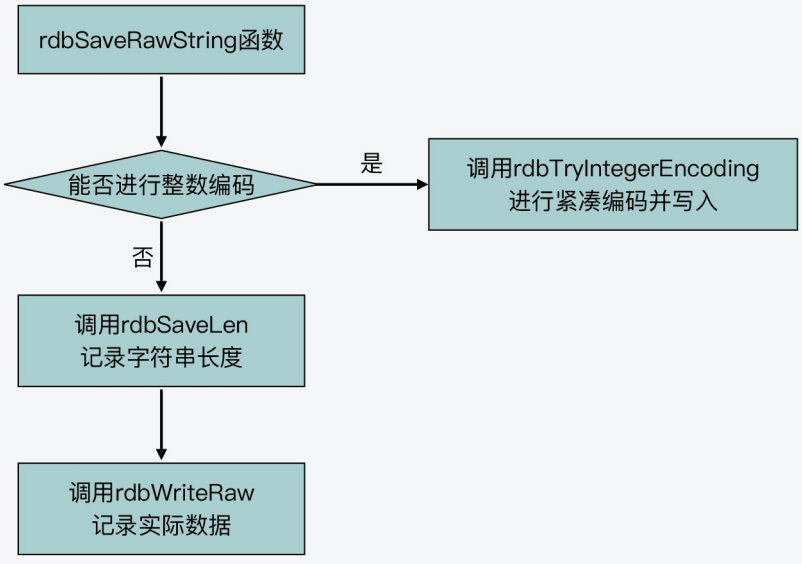
```
#define RDB_OPCODE_IDLE       248   //标识LRU空闲时间
#define RDB_OPCODE_FREQ       249   //标识LFU访问频率信息
#define RDB_OPCODE_AUX        250   //标识RDB文件头的属性信息
#define RDB_OPCODE_EXPIRETIME_MS 252    //标识以毫秒记录的过期时间
#define RDB_OPCODE_SELECTDB   254   //标识文件中后续键值对所属的数据库编号
#define RDB_OPCODE_EOF        255   //标识RDB文件结束，用在文件尾
```

**第二步**，rdbSaveAuxField函数调用rdbSaveRawString函数（在rdb.c文件中）写入属性信息的键，而键通常是一个字符串。**rdbSaveRawString函数**是用来写入字符串的通用函数，它会先记录字符串长度，然后再记录实际字符串，如下图所示。这个长度信息是为了解析RDB文件时，程序可以基于它知道当前读取的字符串应该读取多少个字节。




不过，为了节省RDB文件消耗的空间，如果字符串中记录的实际是一个整数，rdbSaveRawString函数还会**调用rdbTryIntegerEncoding函数**（在rdb.c文件中），尝试用**紧凑结构**对字符串进行编码。具体做法你可以进一步阅读rdbTryIntegerEncoding函数。

下图展示了rdbSaveRawString函数的基本执行逻辑，你可以看下。其中，它调用rdbSaveLen函数写入字符串长度，调用rdbWriteRaw函数写入实际数据。



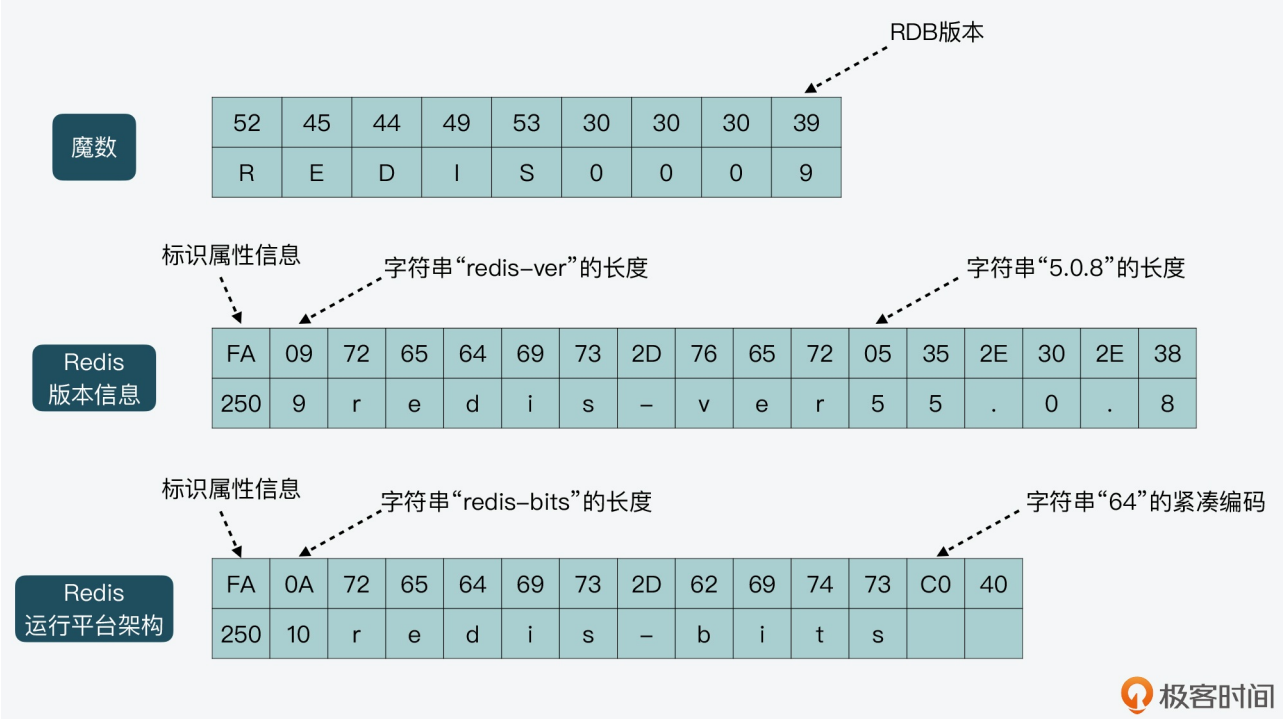



**第三步**，rdbSaveAuxField函数就需要写入属性信息的值了。因为属性信息的值通常也是字符串，所以和第二步写入属性信息的键类似，rdbSaveAuxField函数会调用rdbSaveRawString函数来写入属性信息的值。

下面的代码展示了rdbSaveAuxField函数的执行整体过程，你可以再回顾下。

```
ssize_t rdbSaveAuxField(rio *rdb, void *key, size_t keylen, void *val, size_t vallen) {
    ssize_t ret, len = 0;
    //写入操作码
    if ((ret = rdbSaveType(rdb,RDB_OPCODE_AUX)) == -1) return -1;
    len += ret;
    //写入属性信息中的键
    if ((ret = rdbSaveRawString(rdb,key,keylen)) == -1) return -1;
    len += ret;
    //写入属性信息中的值
    if ((ret = rdbSaveRawString(rdb,val,vallen)) == -1) return -1;
    len += ret;
    return len;
}
```

到这里，RDB文件头的内容已经写完了。我把刚才创建的RDB文件头的部分内容，画在了下图当中，并且标识了十六进制对应的ASCII字符以及一些关键信息，你可以结合图例来理解刚才介绍的代码。




这样接下来，rdbSaveRio函数就要开始写入实际的键值对了，这也是文件中实际记录数据的部分。下面，我们就来具体看下。

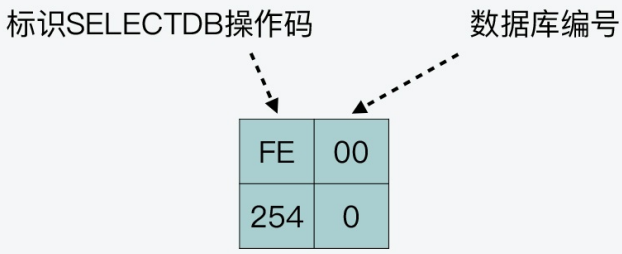
### 生成文件数据部分

因为Redis server上的键值对可能被保存在不同的数据库中，所以，**rdbSaveRio函数会执行一个循环，遍历每个数据库，将其中的键值对写入RDB文件**。

在这个循环流程中，rdbSaveRio函数会先将**SELECTDB操作码**和对应的数据库编号写入RDB文件，这样一来，程序在解析RDB文件时，就可以知道接下来的键值对是属于哪个数据库的了。这个过程如下所示：

```
...
for (j = 0; j < server.dbnum; j++) { //循环遍历每一个数据库
...
//写入SELECTDB操作码
if (rdbSaveType(rdb,RDB_OPCODE_SELECTDB) == -1) goto werr;
if (rdbSaveLen(rdb,j) == -1) goto werr;  //写入当前数据库编号j
...
```

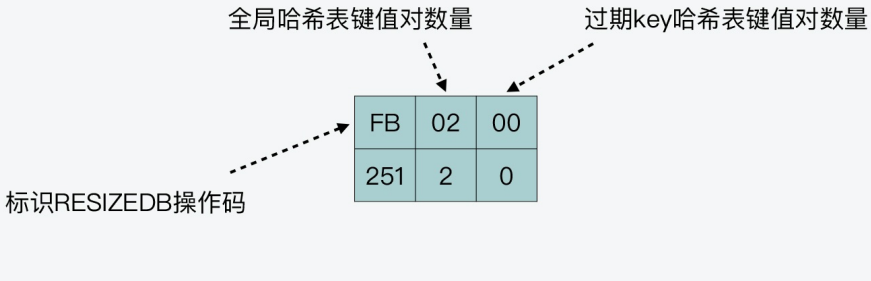
下图展示了刚才我创建的RDB文件中SELECTDB操作码的信息，你可以看到，数据库编号为0。




紧接着，rdbSaveRio函数会写入**RESIZEDB操作码**，用来标识全局哈希表和过期key哈希表中键值对数量的记录，这个过程的执行代码如下所示：

```
...
db_size = dictSize(db->dict);   //获取全局哈希表大小
expires_size = dictSize(db->expires);  //获取过期key哈希表的大小
if (rdbSaveType(rdb,RDB_OPCODE_RESIZEDB) == -1) goto werr;  //写入RESIZEDB操作码
if (rdbSaveLen(rdb,db_size) == -1) goto werr;  //写入全局哈希表大小
if (rdbSaveLen(rdb,expires_size) == -1) goto werr; //写入过期key哈希表大小
...
```

我也把刚才创建的RDB文件中，RESIZEDB操作码的内容画在了下图中，你可以看下。




你可以看到，在RESIZEDB操作码后，紧接着记录的是全局哈希表中的键值对，它的数量是2，然后是过期key哈希表中的键值对，其数量为0。我们刚才在生成RDB文件前，只插入了两个键值对，所以，RDB文件中记录的信息和我们刚才的操作结果是一致的。

好了，在记录完这些信息后，rdbSaveRio函数会接着**执行一个循环流程**，在该流程中，rdbSaveRio函数会取出当前数据库中的每一个键值对，并调用rdbSaveKeyValuePair函数（在rdb.c文件中），将它写入RDB文件。这个基本的循环流程如下所示：

```
 while((de = dictNext(di)) != NULL) {  //读取数据库中的每一个键值对
    sds keystr = dictGetKey(de);  //获取键值对的key
    robj key, *o = dictGetVal(de);  //获取键值对的value
    initStaticStringObject(key,keystr);  //为key生成String对象
    expire = getExpire(db,&key);  //获取键值对的过期时间
    //把key和value写入RDB文件
    if (rdbSaveKeyValuePair(rdb,&key,o,expire) == -1) goto werr;
    ...
}
```

这里，**rdbSaveKeyValuePair函数**主要是负责将键值对实际写入RDB文件。它会先将键值对的过期时间、LRU空闲时间或是LFU访问频率写入RDB文件。在写入这些信息时，rdbSaveKeyValuePair函数都会先调用rdbSaveType函数，写入标识这些信息的操作码，你可以看下下面的代码。

```
if (expiretime != -1) {
    //写入过期时间操作码标识
   if (rdbSaveType(rdb,RDB_OPCODE_EXPIRETIME_MS) == -1) return -1;
   if (rdbSaveMillisecondTime(rdb,expiretime) == -1) return -1;
}
if (savelru) {
   ...
   //写入LRU空闲时间操作码标识
   if (rdbSaveType(rdb,RDB_OPCODE_IDLE) == -1) return -1;
   if (rdbSaveLen(rdb,idletime) == -1) return -1;
}
if (savelfu) {
   ...
   //写入LFU访问频率操作码标识
   if (rdbSaveType(rdb,RDB_OPCODE_FREQ) == -1) return -1;
   if (rdbWriteRaw(rdb,buf,1) == -1) return -1;
}
```

好了，到这里，rdbSaveKeyValuePair函数就要开始实际写入键值对了。为了便于解析RDB文件时恢复键值对，rdbSaveKeyValuePair函数会先调用rdbSaveObjectType函数，写入键值对的类型标识；然后调用rdbSaveStringObject写入键值对的key；最后，它会调用rdbSaveObject函数写入键值对的value。这个过程如下所示，这几个函数都是在rdb.c文件中实现的：

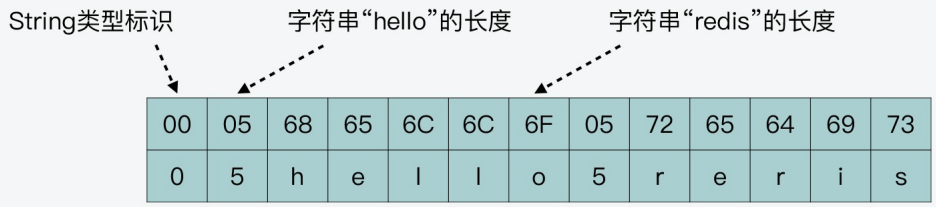
```
if (rdbSaveObjectType(rdb,val) == -1) return -1;  //写入键值对的类型标识
if (rdbSaveStringObject(rdb,key) == -1) return -1; //写入键值对的key
if (rdbSaveObject(rdb,val,key) == -1) return -1; //写入键值对的value
```

这里，你需要注意的是，**rdbSaveObjectType函数会根据键值对的value类型，来决定写入到RDB中的键**

**值对类型标识**，这些类型标识在rdb.h文件中有对应的宏定义。比如，我在刚才创建RDB文件前，写入的键值对分别是String类型和Hash类型，而Hash类型因为它包含的元素个数不多，所以默认采用ziplist数据结构来保存。这两个类型标识对应的数值如下所示：

```
#define RDB_TYPE_STRING   0
#define RDB_TYPE_HASH_ZIPLIST  13
```

我把刚才写入的String类型键值对“hello”“redis”在RDB文件中对应的记录内容，画在了下图中，你可以看下。




你可以看到，这个键值对的开头类型标识就是0，和刚才介绍的RDB_TYPE_STRING宏定义的值是一致的。而紧接着的key和value，它们都会先记录长度信息，然后才记录实际内容。

因为键值对的key都是String类型，所以rdbSaveKeyValuePair函数就用rdbSaveStringObject函数来写入了。而键值对的value有不同的类型，所以，rdbSaveObject函数会根据value的类型，执行不同的代码分支，将value底层数据结构中的内容写入RDB。

好了，到这里，我们就了解了rdbSaveKeyValuePair函数是如何将键值对写入RDB文件中的了。在这个过程中，除了键值对类型、键值对的key和value会被记录以外，键值对的过期时间、LRU空闲时间或是LFU访问频率也都会记录到RDB文件中。这就生成RDB文件的数据部分。

最后，我们再来看下RDB文件尾的生成。

## 生成文件尾

当所有键值对都写入RDB文件后，**rdbSaveRio函数**就可以写入文件尾内容了。文件尾的内容比较简单，主要包括两个部分，一个是RDB文件结束的操作码标识，另一个是RDB文件的校验值。

rdbSaveRio函数会先调用rdbSaveType函数，写入文件结束操作码RDB_OPCODE_EOF，然后调用rioWrite写入检验值，如下所示：

```
...
//写入结束操作码
if (rdbSaveType(rdb,RDB_OPCODE_EOF) == -1) goto werr;

//写入校验值
cksum = rdb->cksum;
```

```
memrev64ifbe(&cksum);
if (rioWrite(rdb,&cksum,8) == 0) goto werr;
...
```

下图展示了我刚才生成的RDB文件的文件尾，你可以看下。


这样，我们也就整体了解了RDB文件从文件头、文件数据部分再到文件尾的整个生成过程了。

## 小结

今天这节课，我给你介绍了Redis内存快照文件RDB的生成。你要知道，创建RDB文件的三个入口函数分别是rdbSave、rdbSaveBackground、rdbSaveToSlavesSockets，它们在Redis源码中被调用的地方，也就是触发RDB文件生成的时机。

另外，你也要重点关注RDB文件的基本组成，并且也要结合rdbSaveRio函数的执行流程，来掌握RDB文件头、文件数据部分和文件尾这三个部分的生成。我总结了以下两点，方便你对RDB文件结构和内容有个整体把握：

- RDB文件使用多种操作码来标识Redis不同的属性信息，以及使用类型码来标识不同value类型；
- RDB文件内容是自包含的，也就是说，无论是属性信息还是键值对，RDB文件都会按照类型、长度、实际数据的格式来记录，这样方便程序对RDB文件的解析。

最后，我也想再说一下，RDB文件包含了Redis数据库某一时刻的所有键值对，以及这些键值对的类型、大小、过期时间等信息。当你了解了RDB文件的格式和生成方法后，其实你就可以根据需求，开发解析RDB文件的程序或是加载RDB文件的程序了。

比如，你可以在RDB文件中查找内存空间消耗大的键值对，也就是在优化Redis性能时通常需要查找的bigkey；你也可以分析不同类型键值对的数量、空间占用等分布情况，来了解业务数据的特点；你还可以自行加载RDB文件，用于测试或故障排查。

当然，这里我也再给你一个小提示，就是在你实际开发RDB文件分析工具之前，可以看下redis-rdb-tools这个工具，它能够帮助你分析RDB文件中的内容。而如果它还不能满足你的定制化需求，你就可以用上这节课学习的内容，来开发自己的RDB分析工具了。

## 每课一问

你能在serverCron函数中，查找到rdbSaveBackground函数一共会被调用执行几次吗？这又分别对应了什么场景呢？

## 精选留言：

- Kaito 2021-09-07 00:47:39

1、RDB 文件是 Redis 的数据快照，以「二进制」格式存储，相比 AOF 文件更小，写盘和加载时间更短

2、RDB 在执行 SAVE / BGSAVE 命令、定时 BGSAVE、主从复制时产生

3、RDB 文件包含文件头、数据部分、文件尾

4、文件头主要包括 Redis 的魔数、RDB 版本、Redis 版本、RDB 创建时间、键值对占用的内存大小等信息

5、文件数据部分包括整个 Redis 数据库中存储的所有键值对信息

- 数据库信息：db 编号、db 中 key 的数量、过期 key 的数量、键值数据
- 键值数据：过期标识、时间戳（绝对时间）、键值对类型、key 长度、key、value 长度、value

6、文件尾保存了 RDB 的结束标记、文件校验值

7、RDB 存储的数据，为了压缩体积，还做了很多优化:

- 变长编码存储键值对数据
- 用操作码标识不同的内容
- 可整数编码的内容使用整数类型紧凑编码

课后题：在 serverCron 函数中，rdbSaveBackground 函数一共会被调用执行几次？这又分别对应了什么场景？

在 serverCron 函数中 rdbSaveBackground 会被调用 2 次。

一次是满足配置的定时 RDB 条件后（save <seconds> <changes>），触发子进程生成 RDB。

另一次是客户端执行了 BGSAVE 命令，Redis 会先设置 server.rdb_bgsave_scheduled = 1，之后 serverCron 函数判断这个变量为 1，也会触发子进程生成 RDB。 [3赞]

- 曾轼麟 2021-09-08 11:52:24

先回答老师的问题：serverCron函数中，查找到 rdbSaveBackground 函数一共会被调用执行几次？

答：包含直接或者间接，一共调用了4次（不知道还有没有漏的）

1、【直接调用】：server.c(1296行)
如果达到了更改量阈值，等待秒数阈值，延时失败重试达到时间，会进行一次调用。

2、【直接调用】：server.c(1369行)
bgsave因为AOF重写的原有被迫推迟，所以在最后需要重新调用。

3、【间接调用】：replicationCron(1340行) -> startBgsaveForReplication -> rdbSaveBackground
主从复制定时任务，通过startBgsaveForReplication，触发的RDB文件保存。

4、【间接调用】：backgroundSaveDoneHandler(1261行) -> backgroundSaveDoneHandlerDisk/backgroundSaveDoneHandlerSocket -> updateSlavesWaitingBgsave -> startBgsaveForReplication -> rdb

SaveBackground
如果当前的进程角度是rdb_child_pid子进程，在结束bgsave后可能有机器在等待RDB文件，那么会调用updateSlavesWaitingBgsave，从而间接的可能调用startBgsaveForReplication函数

补充总结：
本期老师主要介绍了Redis的持久化做法和RDB文件的编码方式，包括文件头部的编码方式，文件的键值对写入的编码方式，还有写入的触发时机等等，也方便我们日后自行解析RDB文件。

此外在本次源码中多次出现了RIO的标识，这里解释一下，RIO其实是unix下的一款IO包，起本质是封装了操作系统I/O，能通过缓冲区的方式调用操作系统I/O去对文件进行读写，此外Redis在保存RDB文件也使用了一些技巧，例如在rdbSave函数中，文件是先写入tmpfile（临时文件）的，最后通过rename的方式修改文件名字来替换掉整个文件，这是安全的文件写入方式，如果在写入期间掉电也并不会导致旧RDB文件损坏，但是也证明在磁盘预留上是需要双倍空间的。 [2赞]