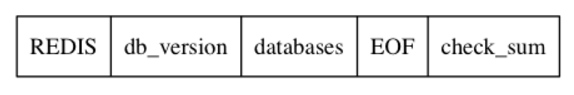
Redis的RDB持久化的相关功能主要是在src/rdb.c中实现的。RDB文件是具有一定编码格式的数据文件，因此src/rdb.c中大部分代码都是处理数据格式的问题。

**一：RDB文件格式**



上图就是一个完整RDB文件的格式。

RDB文件的最开头是REDIS部分，这个部分的长度为5字节，保存着"REDIS"五个字符。通过这个字符串，程序可以在载人文件时，快速检查所载人的文件是否RDB文件。

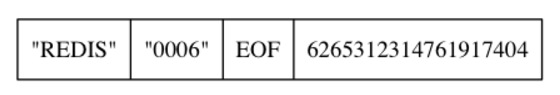
db\_version长度为4字节，它是一个字符串表示的整数，这个整数记录了RDB文件的版本号。比如，”0006”就代表RDB文件的版本为第6版。Redis3.0.5使用的是第6版，因此本文只介绍第6版RDB文件的结构。

databases部分包含着零个或任意多个数据库。也就是保存着Redis服务器中所有数据库中的键值对数据。如果Redis服务器中的所有数据库都是空的，那这个部分也为空的，长度 为0字节。根据数据库所保存键值对的数量、类型和内容不同，这个部分的长度也会有所不同。

EOF部分是一个1字节长度的常量，这个常量标志着RDB文件正文内容的结束，当载入程序遇到这个值的时候，就表明所有数据库的所有键值对都已经载人完毕了。

check\_sum是一个8字节长的无符号整数，保存着一个校验和。该校验和是对RED1S，db\_version，databases，EOF四个部分的内容计算得到的。服务器在载人RDB文件时，会对载入的数据重新计算校验和，然后与check\_sum所记录的校验和进行对比，以此来检查RDB文件是否出错或者损坏。

下图就是一个databases部分为空的RDB文件：



1：databases部分

databases部分可以保存任意多个非空数据库。每个非空数据库都保存为SELECTDB，db\_index，key\_value\_pairs三个部分。

SELECTDB是一个长度为1字节的常量，当载入程序读到这个值时，它知道接下来要读人的将是一个数据库索引db\_index。

db\_index是一个表示数据库索引号的整数值，根据索引号的大小，这个部分的长度可以编码为1字节、2字节或5字节。当读人db\_index部分之后，就切换到相应的数据库上，准备将之后的key\_value\_pairs载入到该数据库中。

key\_value\_pairs部分保存了数据库中的所有键值对数据，如果键值对带有过期时间，那么过期时间也会和键值对保存在一起。根据键值对的数量、类型、内容以及是否有过期时间等条件的不同，key\_value\_pairs部分的长度也会有所不同。

下图展示了一个包含0号数据库和3号数据库的完整RDB文件：



2：key\_value\_pairs部分

key\_value\_pairs 部分保存了数据库中所有的键值对数据，如果键值对带有过期时间的话， 那么过期时间也会被保存在内。

不带过期时间的键值对由TYPE， key和 value 三部分组成。TYPE记录了 value 的类型，代表了值对象的类型及其底层编码。长度为 1 字节，值可以是以下常量中的一个：

#define REDIS\_RDB\_TYPE\_STRING 0

#define REDIS\_RDB\_TYPE\_LIST 1

#define REDIS\_RDB\_TYPE\_SET 2

#define REDIS\_RDB\_TYPE\_ZSET 3

#define REDIS\_RDB\_TYPE\_HASH 4

#define REDIS\_RDB\_TYPE\_HASH\_ZIPMAP 9

#define REDIS\_RDB\_TYPE\_LIST\_ZIPLIST 10

#define REDIS\_RDB\_TYPE\_SET\_INTSET 11

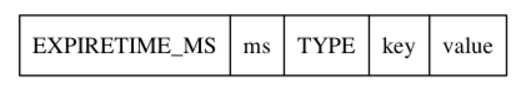
#define REDIS\_RDB\_TYPE\_ZSET\_ZIPLIST 12

#define REDIS\_RDB\_TYPE\_HASH\_ZIPLIST 13

key和value分别保存了键对象和值对象。因键对象总是一个字符串，根据其内容以及长度，key可以有不同的编码和长度。

根据值对象中编码和内容长度的不同，value的结构和长度也会有所不同。

带有过期时间的键值对在RDB文件中的结构如下图所示。



EXPIRETIME\_MS 是长度为1字节的常量，它告知读入程序，接下来要读入的将是一个以毫秒为单位的过期时间。

ms 是一个 8 字节长的带符号整数，记录着一个以毫秒为单位的UNIX时间戳，这个时间戳就是键值对的过期时间。

剩下的TYPE，key和value三个部分与不带过期时间的键值对意义相同。

4：TYPE编码

TYPE常量记录了值对象的类型和编码，TYPE的编码规则如下：

如果值是字符串对象，则TYPE为REDIS\_RDB\_TYPE\_STRING；

列表对象编码为REDIS\_ENCODING\_ZIPLIST时，TYPE为REDIS\_RDB\_TYPE\_LIST\_ZIPLIST；列表对象编码为REDIS\_ENCODING\_LINKEDLIST时，TYPE为REDIS\_RDB\_TYPE\_LIST；

集合对象编码为REDIS\_ENCODING\_INTSET时，TYPE为REDIS\_RDB\_TYPE\_SET\_INTSET；集合对象编码为REDIS\_ENCODING\_HT时，TYPE为REDIS\_RDB\_TYPE\_SET；

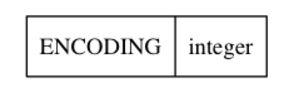
有序集合对象编码为REDIS\_ENCODING\_ZIPLIST时，TYPE为REDIS\_RDB\_TYPE\_ZSET\_ZIPLIST；有序集合对象编码为REDIS\_ENCODING\_SKIPLIST时，TYPE为REDIS\_RDB\_TYPE\_ZSET；

哈希对象编码为REDIS\_ENCODING\_ZIPLIST时，TYPE为REDIS\_RDB\_TYPE\_HASH\_ZIPLIST；哈希对象编码为REDIS\_ENCODING\_HT时，TYPE为REDIS\_RDB\_TYPE\_HASH；

5：key

key记录了键值对中的键。因键总是一个字符串，根据字符串的形式和长度不同，key也有不同的形式。

如果键字符串长度小于等于11，并且是一个整数型字符串，比如”123”, “-151541”等，则将字符串转换为整数，然后以ENCODING和integer的形式保存：



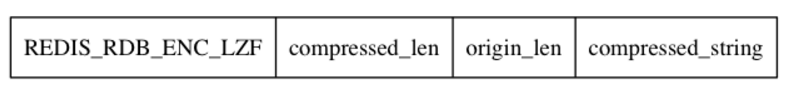
ENCODING是长度为1字节的编码，integer是具体的整数值。根据integer范围的不同，ENCODING的值也不同，规则如下：

如果integer在范围[-128,127]内，则ENCODING的二进制形式为11000000，integer长度为1字节；

如果integer在范围[-32768,32767]内，则ENCODING的二进制形式为11000001，integer长度为2字节；

如果integer在范围[-2147483648,2147483647]内，则ENCODING的二进制形式为11000010，integer长度为4字节；

如果字符串不满足上面的条件，如果Redis开启了压缩功能，并且字符串长度大于20字节，则字符串需要压缩保存，以下面的格式保存：



REDIS\_RDB\_ENC\_LZF 是长度为1字节的常量，表明这是压缩字符串。其值的二进制形式为11000011；

compressed\_len是压缩后的字符串长度；origin\_len是压缩前的字符串长度；

compressed\_string是压缩后的字符串。

如果未开启压缩功能，或者字符串长度小于等于20字节，则以len+string的格式保存，其中len是字符串的长度，string是字符串：

6：value

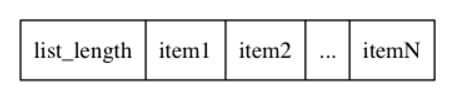
value 部分保存了一个值对象，每个值对象的类型和编码由 TYPE 记录。

a：字符串对象

 TYPE 的值为 REDIS\_RDB\_TYPE\_STRING，则value保存的是一个字符串对象。保存的格式与key的规则一样，不再赘述。

b：列表对象

TYPE值为REDIS\_RDB\_TYPE\_LIST，则value 保存的是一个 REDIS\_ENCODING\_LINKEDLIST 编码的列表对象，RDB文件保存这种对象的结构如下图所示：

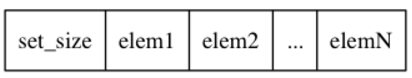


list\_length 记录了列表的长度，也就是列表中的元素个数。接下来以 item 开头的部分代表列表的元素，因为每个列表项都是一个字符串对象，因此保存的规则与key相同。

如果TYPE值为REDIS\_RDB\_TYPE\_LIST\_ZIPLIST，则value 保存的是一个 REDIS\_ENCODING\_ZIPLIST编码的列表对象，这种编码的列表对象底层是连续的内存块，RDB文件保存这种类型时，直接将其当做字符串对象处理，因此保存的规则与key相同。

c：集合对象

TYPE 的值为REDIS\_RDB\_TYPE\_SET，则value 保存的是一个 REDIS\_ENCODING\_HT 编码的集合对象，RDB文件保存这种对象的结构如下图所示：

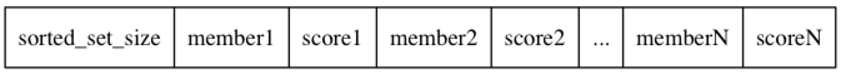


set\_size记录了集合中的元素个数。接下来以 elem开头的部分代表集合的元素，因为每个集合元素都是一个字符串对象，因此保存的规则与key相同。

如果TYPE值为REDIS\_RDB\_TYPE\_SET\_INTSET，则value 保存的是一个 REDIS\_ENCODING\_INTSET编码的集合对象，这种编码的集合对象底层是连续的内存块，RDB文件保存这种类型时，直接将其当做字符串对象处理，因此保存的规则与key相同。

d：有序集合对象

TYPE 的值为REDIS\_RDB\_TYPE\_ZSET，则 value 保存的是一个 REDIS\_ENCODING\_SKIPLIST 编码的有序集合对象，RDB文件保存这种对象的结构如下图所示：

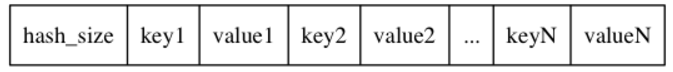


sorted\_set\_size 记录了有序集合的大小，也就是这个有序集合保存了多少元素。接下来是每个元素的成员和分值部分，成员是一个字符串对象，因此保存的规则与key相同。分值是一个 double 类型的浮点数，保存到RDB文件中时，会先将分值转换成字符串对象，因此保存的规则与key相同。

如果TYPE值为REDIS\_RDB\_TYPE\_ZSET\_ZIPLIST，则value 保存的是一个 REDIS\_ENCODING\_ZIPLIST编码的有序集合对象，这种编码的有序集合对象底层是连续的内存块，RDB文件保存这种类型时，直接将其当做字符串对象处理，因此保存的规则与key相同。

e：哈希对象

TYPE 的值为 REDIS\_RDB\_TYPE\_HASH，则value 保存的就是一个 REDIS\_ENCODING\_HT 编码的哈希对象，RDB文件保存这种对象的结构如下图所示：



hash\_size 记录了哈希表的大小，也就是这个哈希表保存了多少键值对。剩下的就是键值对了，键值对的键和值都是字符串对象，因此保存的规则与key相同。

如果TYPE值为REDIS\_RDB\_TYPE\_HASH\_ZIPLIST，则value 保存的是一个 REDIS\_ENCODING\_ZIPLIST编码的哈希对象，这种编码的哈希对象底层是连续的内存块，RDB文件保存这种类型时，直接将其当做字符串对象处理，因此保存的规则与key相同。

**二：代码实现**

1：保存数据库的实现

保存数据库到RDB文件的操作，是由函数rdbSaveRio实现的，它的代码如下：

int rdbSaveRio**(**rio **\***rdb**,** int **\***error**)** **{**

dictIterator **\***di **=** **NULL;**

dictEntry **\***de**;**

char magic**[**10**];**

int j**;**

long long now **=** mstime**();**

uint64\_t cksum**;**

**if** **(**server**.**rdb\_checksum**)**

rdb**->**update\_cksum **=** rioGenericUpdateChecksum**;**

snprintf**(**magic**,sizeof(**magic**),**"REDIS%04d"**,**REDIS\_RDB\_VERSION**);**

**if** **(**rdbWriteRaw**(**rdb**,**magic**,**9**)** **==** **-**1**)** **goto** werr**;**

**for** **(**j **=** 0**;** j **<** server**.**dbnum**;** j**++)** **{**

redisDb **\***db **=** server**.**db**+**j**;**

dict **\***d **=** db**->**dict**;**

**if** **(**dictSize**(**d**)** **==** 0**)** **continue;**

di **=** dictGetSafeIterator**(**d**);**

**if** **(!**di**)** **return** REDIS\_ERR**;**

*/\* Write the SELECT DB opcode \*/*

**if** **(**rdbSaveType**(**rdb**,**REDIS\_RDB\_OPCODE\_SELECTDB**)** **==** **-**1**)** **goto** werr**;**

**if** **(**rdbSaveLen**(**rdb**,**j**)** **==** **-**1**)** **goto** werr**;**

*/\* Iterate this DB writing every entry \*/*

**while((**de **=** dictNext**(**di**))** **!=** **NULL)** **{**

sds keystr **=** dictGetKey**(**de**);**

robj key**,** **\***o **=** dictGetVal**(**de**);**

long long expire**;**

initStaticStringObject**(**key**,**keystr**);**

expire **=** getExpire**(**db**,&**key**);**

**if** **(**rdbSaveKeyValuePair**(**rdb**,&**key**,**o**,**expire**,**now**)** **==** **-**1**)** **goto** werr**;**

**}**

dictReleaseIterator**(**di**);**

**}**

di **=** **NULL;** */\* So that we don't release it again on error. \*/*

*/\* EOF opcode \*/*

**if** **(**rdbSaveType**(**rdb**,**REDIS\_RDB\_OPCODE\_EOF**)** **==** **-**1**)** **goto** werr**;**

*/\* CRC64 checksum. It will be zero if checksum computation is disabled, the*

*\* loading code skips the check in this case. \*/*

cksum **=** rdb**->**cksum**;**

memrev64ifbe**(&**cksum**);**

**if** **(**rioWrite**(**rdb**,&**cksum**,**8**)** **==** 0**)** **goto** werr**;**

**return** REDIS\_OK**;**

werr**:**

**if** **(**error**)** **\***error **=** errno**;**

**if** **(**di**)** dictReleaseIterator**(**di**);**

**return** REDIS\_ERR**;**

**}**

首先，如果配置文件中的rdbchecksum选项为"yes"的话，则server.rdb\_checksum为1，因此设置rdb->update\_cksum为rioGenericUpdateChecksum；表明使用该函数作为计算校验码的函数；

然后，构造RDB文件的文件头"REDIS0006"，其中"0006"是RDB文件的版本，目前是6，构造完文件头之后，调用rdbWriteRaw写入到rdb中；

然后，针对Redis中的每一个数据库，只要该数据库不为空，就创建一个轮训数据库字典的安全迭代器di；

然后，首先将常量REDIS\_RDB\_OPCODE\_SELECTDB写入rdb中，再将当前的数据库索引j写入到rdb中；

然后，利用迭代器di，轮训数据库字典中每一个字典项，取出其中的键keystr，值对象o以及键的超时时间expire（如果有的话），因为数据库中保存键时是直接保存的原始字符串，因此需要将keystr转换成字符串对象key，然后调用rdbSaveKeyValuePair将key、o以及expire写入到rdb中；

处理完所有的键值对后，将常量REDIS\_RDB\_OPCODE\_EOF写入rdb中；

最后，因每次向rdb写入数据时，同时会计算当前内容的校验码，并将其记录到rdb->cksum中，因此，将当前所有数据的校验码cksum，转换成小端模式后，写入到rdb中；

2：SAVE命令的实现

执行SAVE命令时，会阻塞当前Redis服务器，此时客户端无法进行操作，该命令主要是通过saveCommand实现的，而该函数又主要是调用rdbSave实现：

void saveCommand**(**redisClient **\***c**)** **{**

**if** **(**server**.**rdb\_child\_pid **!=** **-**1**)** **{**

addReplyError**(**c**,**"Background save already in progress"**);**

**return;**

**}**

**if** **(**rdbSave**(**server**.**rdb\_filename**)** **==** REDIS\_OK**)** **{**

addReply**(**c**,**shared**.**ok**);**

**}** **else** **{**

addReply**(**c**,**shared**.**err**);**

**}**

**}**

在函数saveCommand中，如果server.rdb\_child\_pid不是-1，则说明已经有子进程开始进行SAVE过程了，则直接反馈"Background save already in progress"给客户端；

然后调用rdbSave，将数据记录到server.rdb\_filename中，成功则反馈shared.ok，失败反馈shared.err。

函数rdbSave的代码如下：

int rdbSave**(**char **\***filename**)** **{**

char tmpfile**[**256**];**

FILE **\***fp**;**

rio rdb**;**

int error**;**

snprintf**(**tmpfile**,**256**,**"temp-%d.rdb"**,** **(**int**)** getpid**());**

fp **=** fopen**(**tmpfile**,**"w"**);**

**if** **(!**fp**)** **{**

redisLog**(**REDIS\_WARNING**,** "Failed opening .rdb for saving: %s"**,**

strerror**(**errno**));**

**return** REDIS\_ERR**;**

**}**

rioInitWithFile**(&**rdb**,**fp**);**

**if** **(**rdbSaveRio**(&**rdb**,&**error**)** **==** REDIS\_ERR**)** **{**

errno **=** error**;**

**goto** werr**;**

**}**

*/\* Make sure data will not remain on the OS's output buffers \*/*

**if** **(**fflush**(**fp**)** **==** EOF**)** **goto** werr**;**

**if** **(**fsync**(**fileno**(**fp**))** **==** **-**1**)** **goto** werr**;**

**if** **(**fclose**(**fp**)** **==** EOF**)** **goto** werr**;**

*/\* Use RENAME to make sure the DB file is changed atomically only*

*\* if the generate DB file is ok. \*/*

**if** **(**rename**(**tmpfile**,**filename**)** **==** **-**1**)** **{**

redisLog**(**REDIS\_WARNING**,**"Error moving temp DB file on the final destination: %s"**,** strerror**(**errno**));**

unlink**(**tmpfile**);**

**return** REDIS\_ERR**;**

**}**

redisLog**(**REDIS\_NOTICE**,**"DB saved on disk"**);**

server**.**dirty **=** 0**;**

server**.**lastsave **=** time**(NULL);**

server**.**lastbgsave\_status **=** REDIS\_OK**;**

**return** REDIS\_OK**;**

werr**:**

redisLog**(**REDIS\_WARNING**,**"Write error saving DB on disk: %s"**,** strerror**(**errno**));**

fclose**(**fp**);**

unlink**(**tmpfile**);**

**return** REDIS\_ERR**;**

**}**

在该函数中，首先在当前目录创建临时文件temp-<pid>.rdb，其中<pid>就是当前进程的PID。然后使用该临时文件的文件指针fp初始化rio结构rdb，该结构是Redis中用于IO操作的数据结构，主要是封装了read和write操作。

然后调用rdbSaveRio，将Redis所有数据写入rdb中，也就是写入上面的临时文件中；之后调用fflush，fsync和fclose，保证数据已经写入到硬盘上，并且关闭临时文件；

然后将该临时文件改名为filename； 然后更新server中RDB相关的属性：

server**.**dirty **=** 0**;**

server**.**lastsave **=** time**(NULL);**

server**.**lastbgsave\_status **=** REDIS\_OK**;**

server.dirty计数器记录距离上一次成功执行SAYE命令或者BGSAYE命令之后，服务器 对数据库状态（所有数据库）进行了多少次修改（包括写人、删除、更新等操作)；

server.lastsave属性是记录了服务器上一次成功执行SAYE命令或BGSAYE命令的时间。

配置文件中，设置的Redis服务器自动快照的条件，就是根据这两个值进行判断的。

3：BGSAVE命令的实现

BGSAVE命令可以在后台异步地进行快照操作，快照的同时服务器还可以继续响应来自客户端的请求。该命令主要是通过bgsaveCommand实现的，而该函数又主要是调用rdbSaveBackground实现：

void bgsaveCommand**(**redisClient **\***c**)** **{**

**if** **(**server**.**rdb\_child\_pid **!=** **-**1**)** **{**

addReplyError**(**c**,**"Background save already in progress"**);**

**}** **else** **if** **(**server**.**aof\_child\_pid **!=** **-**1**)** **{**

addReplyError**(**c**,**"Can't BGSAVE while AOF log rewriting is in progress"**);**

**}** **else** **if** **(**rdbSaveBackground**(**server**.**rdb\_filename**)** **==** REDIS\_OK**)** **{**

addReplyStatus**(**c**,**"Background saving started"**);**

**}** **else** **{**

addReply**(**c**,**shared**.**err**);**

**}**

**}**

在函数bgsaveCommand中，如果server.rdb\_child\_pid不是-1，则说明已经有进程开始进行SAVE过程了，则直接反馈"Background save already in progress"给客户端；

如果server.aof\_child\_pid不是-1，则说明已经有进程开始进行重写AOF文件的过程了，为了避免性能问题，则直接反馈"Can't BGSAVE while AOF log rewriting is in progress"给客户端；

然后调用rdbSaveBackground，将数据记录到server.rdb\_filename中，成功则反馈shared.ok，失败反馈shared.err；

rdbSaveBackground的代码如下：

int rdbSaveBackground**(**char **\***filename**)** **{**

pid\_t childpid**;**

long long start**;**

**if** **(**server**.**rdb\_child\_pid **!=** **-**1**)** **return** REDIS\_ERR**;**

server**.**dirty\_before\_bgsave **=** server**.**dirty**;**

server**.**lastbgsave\_try **=** time**(NULL);**

start **=** ustime**();**

**if** **((**childpid **=** fork**())** **==** 0**)** **{**

int retval**;**

*/\* Child \*/*

closeListeningSockets**(**0**);**

redisSetProcTitle**(**"redis-rdb-bgsave"**);**

retval **=** rdbSave**(**filename**);**

**if** **(**retval **==** REDIS\_OK**)** **{**

size\_t private\_dirty **=** zmalloc\_get\_private\_dirty**();**

**if** **(**private\_dirty**)** **{**

redisLog**(**REDIS\_NOTICE**,**

"RDB: %zu MB of memory used by copy-on-write"**,**

private\_dirty**/(**1024**\***1024**));**

**}**

**}**

exitFromChild**((**retval **==** REDIS\_OK**)** **?** 0 **:** 1**);**

**}** **else** **{**

*/\* Parent \*/*

server**.**stat\_fork\_time **=** ustime**()-**start**;**

server**.**stat\_fork\_rate **=** **(**double**)** zmalloc\_used\_memory**()** **\*** 1000000 **/** server**.**stat\_fork\_time **/** **(**1024**\***1024**\***1024**);** */\* GB per second. \*/*

latencyAddSampleIfNeeded**(**"fork"**,**server**.**stat\_fork\_time**/**1000**);**

**if** **(**childpid **==** **-**1**)** **{**

server**.**lastbgsave\_status **=** REDIS\_ERR**;**

redisLog**(**REDIS\_WARNING**,**"Can't save in background: fork: %s"**,**

strerror**(**errno**));**

**return** REDIS\_ERR**;**

**}**

redisLog**(**REDIS\_NOTICE**,**"Background saving started by pid %d"**,**childpid**);**

server**.**rdb\_save\_time\_start **=** time**(NULL);**

server**.**rdb\_child\_pid **=** childpid**;**

server**.**rdb\_child\_type **=** REDIS\_RDB\_CHILD\_TYPE\_DISK**;**

updateDictResizePolicy**();**

**return** REDIS\_OK**;**

**}**

**return** REDIS\_OK**;** */\* unreached \*/*

**}**

在该函数中，首先如果server.rdb\_child\_pid不为-1，说明当前已经在后台保存Redis数据了，这种情况直接返回REDIS\_ERR；

然后保存当前有关RDB的状态：

server**.**dirty\_before\_bgsave **=** server**.**dirty**;**

server**.**lastbgsave\_try **=** time**(NULL);**

server.dirty\_before\_bgsave用于执行完后，恢复server.dirty；server.lastbgsave\_try用于记录BGSAVE上一次的执行时间，以便决定何时自动执行下一次BGSAVE操作；

调用fork创建子进程，在子进程中，首先调用closeListeningSockets，关闭不必要的描述符；然后调用redisSetProcTitle然后调用rdbSave保存数据到filename中。

注意，调用fork时，子进程的内存与父进程（Redis服务器）是一模一样的，因此子进程保存的数据库也就是fork时刻的状态。而此时父进程继续接受来自客户端的命令，这就会产生新的数据，新的数据并未追加到RDB中。AOF持久化可以做到这点。因此AOF持久化丢失的数据会更少。

如果rdbSave执行成功，则调用zmalloc\_get\_private\_dirty，从文件/proc/self/smaps中获取当前进程的Private\_Dirty值，也就是用于写时复制的内存，将其记录到日志中；然后子进程退出。

调用fork后， 在父进程中，首先计算执行fork系统调用的执行时间，记录到server.stat\_fork\_time中；然后根据当前使用的内存总量，得到server.stat\_fork\_rate（单位为GB/s），然后调用latencyAddSampleIfNeeded，根据fork执行时间是否超过阈值，记录到server.latency\_events中；以上信息主要用于Redis的延迟分析。

如果fork调用失败，则记录错误信息到日志，并且返回REDIS\_ERR； 否则，更新以下信息：

server**.**rdb\_save\_time\_start **=** time**(NULL);**

server**.**rdb\_child\_pid **=** childpid**;**

server**.**rdb\_child\_type **=** REDIS\_RDB\_CHILD\_TYPE\_DISK**;**

然后调用updateDictResizePolicy，禁止Redis中的字典数据结构rehash（并非完全禁止，字典哈希表负载率大于500%时，依然进行rehash）；最后返回REDIS\_OK。

4：加载RDB文件

当Redis服务器启动时，会查找是否存在RDB文件，如果存在，则将RDB文件加载到Redis中。加载RDB文件的操作主要是通过rdbLoad实现的，代码如下：

int rdbLoad**(**char **\***filename**)** **{**

uint32\_t dbid**;**

int type**,** rdbver**;**

redisDb **\***db **=** server**.**db**+**0**;**

char buf**[**1024**];**

long long expiretime**,** now **=** mstime**();**

FILE **\***fp**;**

rio rdb**;**

**if** **((**fp **=** fopen**(**filename**,**"r"**))** **==** **NULL)** **return** REDIS\_ERR**;**

rioInitWithFile**(&**rdb**,**fp**);**

rdb**.**update\_cksum **=** rdbLoadProgressCallback**;**

rdb**.**max\_processing\_chunk **=** server**.**loading\_process\_events\_interval\_bytes**;**

**if** **(**rioRead**(&**rdb**,**buf**,**9**)** **==** 0**)** **goto** eoferr**;**

buf**[**9**]** **=** '\0'**;**

**if** **(**memcmp**(**buf**,**"REDIS"**,**5**)** **!=** 0**)** **{**

fclose**(**fp**);**

redisLog**(**REDIS\_WARNING**,**"Wrong signature trying to load DB from file"**);**

errno **=** EINVAL**;**

**return** REDIS\_ERR**;**

**}**

rdbver **=** atoi**(**buf**+**5**);**

**if** **(**rdbver **<** 1 **||** rdbver **>** REDIS\_RDB\_VERSION**)** **{**

fclose**(**fp**);**

redisLog**(**REDIS\_WARNING**,**"Can't handle RDB format version %d"**,**rdbver**);**

errno **=** EINVAL**;**

**return** REDIS\_ERR**;**

**}**

startLoading**(**fp**);**

**while(**1**)** **{**

robj **\***key**,** **\***val**;**

expiretime **=** **-**1**;**

*/\* Read type. \*/*

**if** **((**type **=** rdbLoadType**(&**rdb**))** **==** **-**1**)** **goto** eoferr**;**

**if** **(**type **==** REDIS\_RDB\_OPCODE\_EXPIRETIME**)** **{**

**if** **((**expiretime **=** rdbLoadTime**(&**rdb**))** **==** **-**1**)** **goto** eoferr**;**

*/\* We read the time so we need to read the object type again. \*/*

**if** **((**type **=** rdbLoadType**(&**rdb**))** **==** **-**1**)** **goto** eoferr**;**

*/\* the EXPIRETIME opcode specifies time in seconds, so convert*

*\* into milliseconds. \*/*

expiretime **\*=** 1000**;**

**}** **else** **if** **(**type **==** REDIS\_RDB\_OPCODE\_EXPIRETIME\_MS**)** **{**

*/\* Milliseconds precision expire times introduced with RDB*

*\* version 3. \*/*

**if** **((**expiretime **=** rdbLoadMillisecondTime**(&**rdb**))** **==** **-**1**)** **goto** eoferr**;**

*/\* We read the time so we need to read the object type again. \*/*

**if** **((**type **=** rdbLoadType**(&**rdb**))** **==** **-**1**)** **goto** eoferr**;**

**}**

**if** **(**type **==** REDIS\_RDB\_OPCODE\_EOF**)**

**break;**

*/\* Handle SELECT DB opcode as a special case \*/*

**if** **(**type **==** REDIS\_RDB\_OPCODE\_SELECTDB**)** **{**

**if** **((**dbid **=** rdbLoadLen**(&**rdb**,NULL))** **==** REDIS\_RDB\_LENERR**)**

**goto** eoferr**;**

**if** **(**dbid **>=** **(**unsigned**)**server**.**dbnum**)** **{**

redisLog**(**REDIS\_WARNING**,**"FATAL: Data file was created with a Redis server configured to handle more than %d databases. Exiting\n"**,** server**.**dbnum**);**

exit**(**1**);**

**}**

db **=** server**.**db**+**dbid**;**

**continue;**

**}**

*/\* Read key \*/*

**if** **((**key **=** rdbLoadStringObject**(&**rdb**))** **==** **NULL)** **goto** eoferr**;**

*/\* Read value \*/*

**if** **((**val **=** rdbLoadObject**(**type**,&**rdb**))** **==** **NULL)** **goto** eoferr**;**

*/\* Check if the key already expired. This function is used when loading*

*\* an RDB file from disk, either at startup, or when an RDB was*

*\* received from the master. In the latter case, the master is*

*\* responsible for key expiry. If we would expire keys here, the*

*\* snapshot taken by the master may not be reflected on the slave. \*/*

**if** **(**server**.**masterhost **==** **NULL** **&&** expiretime **!=** **-**1 **&&** expiretime **<** now**)** **{**

decrRefCount**(**key**);**

decrRefCount**(**val**);**

**continue;**

**}**

*/\* Add the new object in the hash table \*/*

dbAdd**(**db**,**key**,**val**);**

*/\* Set the expire time if needed \*/*

**if** **(**expiretime **!=** **-**1**)** setExpire**(**db**,**key**,**expiretime**);**

decrRefCount**(**key**);**

**}**

*/\* Verify the checksum if RDB version is >= 5 \*/*

**if** **(**rdbver **>=** 5 **&&** server**.**rdb\_checksum**)** **{**

uint64\_t cksum**,** expected **=** rdb**.**cksum**;**

**if** **(**rioRead**(&**rdb**,&**cksum**,**8**)** **==** 0**)** **goto** eoferr**;**

memrev64ifbe**(&**cksum**);**

**if** **(**cksum **==** 0**)** **{**

redisLog**(**REDIS\_WARNING**,**"RDB file was saved with checksum disabled: no check performed."**);**

**}** **else** **if** **(**cksum **!=** expected**)** **{**

redisLog**(**REDIS\_WARNING**,**"Wrong RDB checksum. Aborting now."**);**

exit**(**1**);**

**}**

**}**

fclose**(**fp**);**

stopLoading**();**

**return** REDIS\_OK**;**

eoferr**:** */\* unexpected end of file is handled here with a fatal exit \*/*

redisLog**(**REDIS\_WARNING**,**"Short read or OOM loading DB. Unrecoverable error, aborting now."**);**

exit**(**1**);**

**return** REDIS\_ERR**;** */\* Just to avoid warning \*/*

**}**

该函数中，首先打开filename，用该文件初始化rdb；然后置rdb.update\_cksum为rdbLoadProgressCallback，该函数用于每次读取文件中数据时计算其校验码，以及处理事件等；然后置rdb.max\_processing\_chunk为server.loading\_process\_events\_interval\_bytes，该值表示是一次read操作读取的最大字节数；

开始从rdb中读取9个字节，判断前5个字节是否是"REDIS"，不是直接报错退出；将后4个字节的版本号转换成整数rdbver，如果rdbver小于1，或者大于6，则报错退出；

然后调用startLoading标记开始加载过程，该函数记录load开始的时间，要load的总字节数，以及置server.loading为1表明开始load等；

接下来，开始从rdb中读取数据。首先调用rdbLoadType读取1字节的type，如果type值为REDIS\_RDB\_OPCODE\_EXPIRETIME，则接着调用rdbLoadTime读取键的超时时间(秒)，并将其转换为毫秒单位；如果type值为REDIS\_RDB\_OPCODE\_EXPIRETIME\_MS，则调用rdbLoadMillisecondTime读取键的超时时间(毫秒)，然后接着读1字节的type；

如果type值为REDIS\_RDB\_OPCODE\_EOF，则直接退出循环；

如果type值为REDIS\_RDB\_OPCODE\_SELECTDB，则调用rdbLoadLen得到数据库索引，然后判断索引是否有效，无效直接报错退出；索引有效，则切换到相应的数据库，然后接着读取；

调用rdbLoadStringObject从rdb中读取出键对象key，然后调用rdbLoadObject从rdb中读取值对象val；如果当前是主节点，则判断该键是否超时，若是则直接抛弃；如果是从节点，则不判断键是否超时；

调用dbAdd将key和val添加到数据库的字典中；如果键设置了超时时间，则调用setExpire设置该键的超时时间；

如果RDB版本号rdbver大于等于5并且server.rdb\_checksum为真，则需要比对校验码，首先从rdb中读取校验码，然后跟当前计算的校验码expected比较，不匹配则报错退出；

最后，关闭filename，调用stopLoading置server.loading为0表示load过程结束，然后返回REDIS\_OK；如果以上过程有错误发生，则记录错误之后，程序直接退出。

其他相关RDB的代码，参考：

https://github.com/gqtc/redis-3.0.5/blob/master/redis-3.0.5/src/rdb.c

http://redis.io/topics/latency.