Redis相关问题

记录一下Redis的学习

SDS 简单动态字符串

redis没有直接使用C语言的传统字符串表示，而是自己构建了(simple dynamic string)的抽象类型，并且广泛运用在redis的代码当中。

传统的c字符串只会在redis的代码中充当字符串字面量使用，也就是类似于打印日志时 log("xxxxx")这样使用。

sds.h/sdshdr 定义了sds的结构

```
struct sdshdr {
    //记录buf数组中所使用的字节数量
    //等于sds所保存的字符串的长度
    int len;
    //记录buf数组中为使用的字节数量
    int free;
    //字节数组 用于保存字符串
    char buf[];
}
```

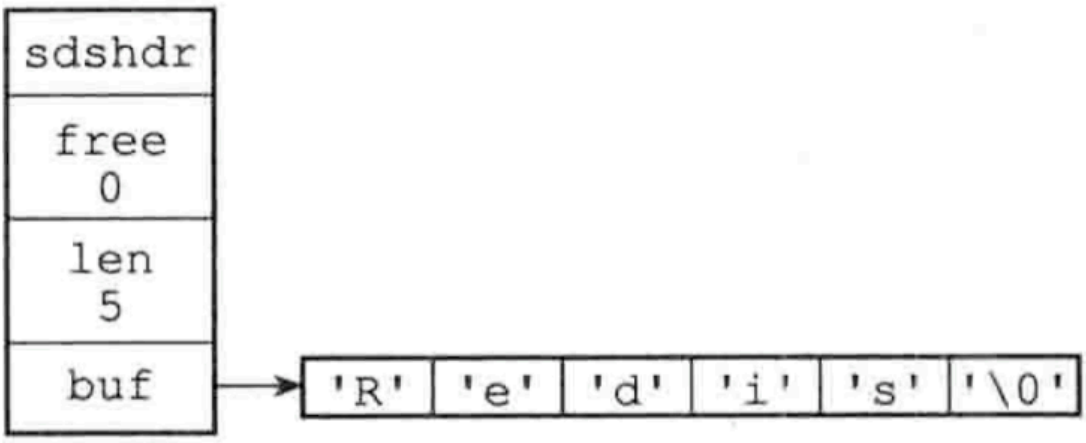


图 2-1　SDS 示例

并且还沿用了c字符串的以空字符串'\0'结尾，这样可以重用一部分c字符串函数库里面的函数。

C字符串	SDS
获取字符串长度的复杂度为O(n)	获取字符串长度的复杂度为O(1)
API是不安全的，可能造成缓冲区溢出	API是安全的，不会造成缓冲区溢出
修改字符串长度N次必然要执行N次内存重分配	修改字符串长度N次最多执行N次内存重分配
只能保存文本数据	可以保存文本或者二进制数据

对象的类型与编码

redis内置有5种对象：`字符串，列表，哈希，集合，有序集合`。而redis中也自己实现了许多的数据结构例如:`SDS，双端链表，字典，跳表，压缩列表，整数集合，哈希表`等等，这里不会讨论如何实现这些数据结构，但是redis是用这些实现的数据结构来实现它的5种内置对象的，每种对象都用到了至少一种我们刚才介绍的数据结构。

针对不同的场景，我们可以为对象设置多种不同的数据结构实现，可以优化对象在不同场景下的使用效率。

对象

类型常量	对象
REDIS_STRING	字符串对象
REDIS_LIST	列表对象
REDIS_HASH	哈希对象
REDIS_SET	集合对象
REDIS_ZSET	有序集合的对象

编码

编码常量	数据结构
REDIS_ENCODING_INT	long类型的整数
REDIS_ENCODING_EMBSTR	embstr编码的简单动态字符串
REDIS_ENCODING_RAW	简单动态字符串
REDIS_ENCODING_HT	字典
REDIS_ENCODING_LINKEDLIST	有序集合的对象
REDIS_ENCODING_ZIPLIST	压缩列表
REDIS_ENCODING_INTSET	整数集合
REDIS_ENCODING_SKIPLIST	跳表，字典

对象与编码的关系

类型	编码	对象
REDIS_STRING	REDIS_ENCODING_INT	使用整数实现的字符串对象
REDIS_STRING	REDIS_ENCODING_EMBSTR	使用embstr编码的动态字符串实现的字符串对象
REDIS_STRING	REDIS_ENCODING_RAW	使用简单动态字符串实现的字符串对象
REDIS_LIST	REDIS_ENCODING_ZIPLIST	使用压缩列表实现的列表对象
REDIS_LIST	REDIS_ENCODING_LINKEDLIST	使用双端列表实现的列表对象
REDIS_HASH	REDIS_ENCODING_ZIPLIST	使用压缩列表实现的哈希对象
REDIS_HASH	REDIS_ENCODING_HT	使用字典实现的哈希对象
REDIS_SET	REDIS_ENCODING_INTSET	使用整数集合实现的集合对象
REDIS_SET	REDIS_ENCODING_HT	使用字典实现的集合对象
REDIS_ZSET	REDIS_ENCODING_ZIPLIST	使用压缩列表实现的有序集合对象
REDIS_ZSET	REDIS_ENCODING_SKIPLIST	使用跳表和字典实现的有序集合对象

谨慎处理多数据库程序

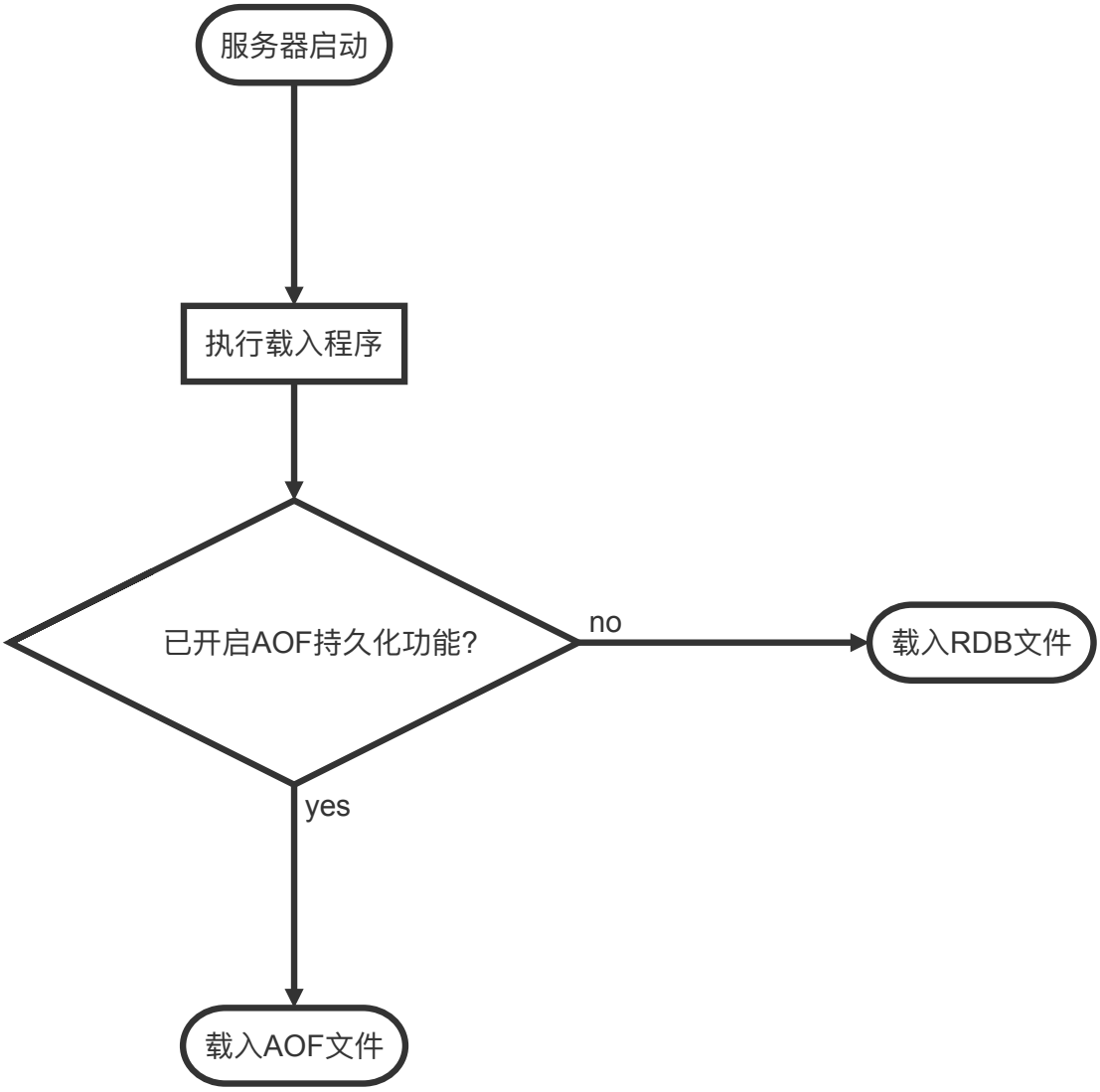
到目前为止，Redis仍然没有可以返回客户端目标数据库的命令，虽然redis-cli客户端会在输入符旁边提示当前所使用的目标数据库，但在其他的redis-sdk中并没有继承，所以为了避免对数据库进行误操作，最好先执行下select命令。

RDB和AOF

rdb和aof 都是redis提供的用于持久化的功能。

RDB持久化保存数据库状态的方法是将数据编码后保存在RDB文件当中，而AOF则是记录执行的`SET,SADD,RPUSH`三个命令保存到AOF文件当中。

两种恢复手段的载入判断流程。



SAVE 和 BGSAVE

这两个命令都用来生成RDB文件，她们主要的区别如下：

SAVE 命令会阻塞Redis 服务器进程，直到RDB文件创建完毕，在此期间，redis-server不能处理任何命令请求。

BGSAVE 命令会派生出一个子进程，由它来负责创建RDB文件，服务器进程继续进行命令请求。

伪代码：

```
def save():
   rdbSave()

def bg_save():
   pid = fork() //创建子进程

   if pid == 0
     rdbSave()
     signal_parent() //告诉父进程
   elif pid > 0 //父进程继续处理命令请求，并通过轮训等待子进程的信号
     handle_request_and_wait_signal()
   else:
     //处理出错情况
     handle_fork_error()
```

与生成rdb文件不同，rdb的载入工作是服务器启动时自动执行的，所以redis并没有专门用于载入rdb文件的命令。

由于BGSAVE命令的保存工作是由子进程执行的，所以在子进程创建RDB文件的过程中，Redis服务器仍然可以处理客户端的命令请求，但是在此期间服务器处理`SAVE,BGSAVE,BGREWRITEAOF`三个命令的方式会和平时有所不同。

- 在BGSAVE命令执行期间，客户端发送的SAVE命令会被服务器拒绝，此举是防止父进程和子进程同时执行rdbSave函数调用，防止产生竞争条件。
- 其次客户端发送BGSAVE也会被拒绝，理由与拒绝SAVE命令一样
- BGREWRITEAOF 和 BGSAVE两个命令不能同时执行
    - 如果BGSAVE命令正在执行，那么BGREWRITEAOF会被延迟到BGSAVE命令执行完成之后执行
    - 如果BGREWRITEAOF命令正在执行，那么BGSAVE会被服务器拒绝
        - 原因是这两个命令的实际工作都是子进程执行，所以没有冲突和竞争，但是这两个子进程同时执行大量的磁盘写入，会大大的降低性能。

AOF持久化的实现

AOF持久化功能的实现可以分为命令追加(append)，文件写入，文件同步(sync) 三个步骤。

命令追加

当AOF功能正处在打开状态时，客户端发送一条写入命令，服务器执行完之后，会以协议格式将这条命令追加到aof_buf缓冲区末尾

```
struct redisServer {
    // ....
    // AOF 缓冲区
    sds aof_buf;
    // ....
}
```

这就是AOF持久化命令追加步骤的实现原理。

AOF文件的写入与同步

Redis的服务器进程就是一个事件循环(loop),这个循环中的文件事件负责接收客户端的命令请求，以及向客户端发送命令回复。那么如果打开了AOF功能，则会将命令尾加到aof_buf缓冲区中，所以在事件结束前都会调用flushAppendOnlyFile函数来考虑是否要将缓冲区里的内容写入和保存到AOF文件当中。

伪代码:

```
def eventLoop():
    while True:
      //处理文件事件，接收命令请求以及发送命令回复
      processFileEvents();
      //处理时间事件
      processTimeEvents();
      //考虑是否将aof_buf中的内容写入AOF缓冲区
      flushAppendOnlyFile();
```

`flushAppendOnlyFile`这个函数的行为由服务器配置的`appendfsync`选项的值来决定

- always
  - 将aof_buf缓冲区中的所有内容写入并同步到AOF文件
- everysec
  - 将oaf_buf缓冲区中的所有内容写入到AOF文件，如果上次同步AOF文件的时间距离现在超过了1秒钟，那么再次对AOF文件进行同步，并且这个同步操作是由一个线程专门负责的
- no
  - 将aof_buf缓冲区中的所有内容写入到AOF文件，但并不对AOF文件进行同步，何时同步由操作系统来决定。

AOF 持久化的效率和安全性

服务器配置 appendfsync 选项的值直接决定 AOF 持久化功能的效率和安全性。

- 当 appendfsync 的值为 always 时，服务器在每个事件循环都要将 aof_buf 缓冲区中的所有内容写入到 AOF 文件，并且同步 AOF 文件，所以 always 的效率是 appendfsync 选项三个值当中最慢的一个，但从安全性来说，always 也是最安全的，因为即使出现故障停机，AOF 持久化也只会丢失一个事件循环中所产生的命令数据。
- 当 appendfsync 的值为 everysec 时，服务器在每个事件循环都要将 aof_buf 缓冲区中的所有内容写入到 AOF 文件，并且每隔一秒就要在子线程中对 AOF 文件进行一次同步。从效率上来讲，everysec 模式足够快，并且就算出现故障停机，数据库也只丢失一秒钟的命令数据。
- 当 appendfsync 的值为 no 时，服务器在每个事件循环都要将 aof_buf 缓冲区中的所有内容写入到 AOF 文件，至于何时对 AOF 文件进行同步，则由操作系统控制。因为处于 no 模式下的 flushAppendOnlyFile 调用无须执行同步操作，所以该模式下的 AOF 文件写入速度总是最快的，不过因为这种模式会在系统缓存中积累一段时间的写入数据，所以该模式的单次同步时长通常是三种模式中时间最长的。从平摊操作的角度来看，no 模式和 everysec 模式的效率类似，当出现故障停机时，使用 no 模式的服务器将丢失上次同步 AOF 文件之后的所有写命令数据。

AOF文件重写的实现

为了解决AOF文件体积膨胀的问题，Redis提供了AOF文件重写功能，新生成一个AOF文件来替代现有的AOF文件，新旧两个文件所保存的数据库状态相同，但新文件不会包含任何冗余命令,所以新AOF文件的体积会比旧的文件小。

redis的重写aof算法非常的聪明。

直接读取key的值，获取最新的key当前的值,然后用一条命令就可以做为这个key的当前状态。

伪代码:

```
def aof_rewrite(new_aof_file_name):
    # 创建新的AOF文件
    f = create_file(new_aof_file_name)

    # 遍历数据库
    for db in redisServer.db:
```

```
        # 忽略空数据库
        if db.is_empty(): continue

        # 显示指定数据库
        f.write_command("SELECT "+ db.id)

        for key in db:
         # 忽略已过期的key
         if key.is_expired(): continue
          # 根据key的类型对key进行重新
          switch(key.type):
              case String:
                rewrite_string(key) #根据key获取到所有的value 然后拼成写入命令即可
              case List:
                rewrite_list(key)
              case Hash:
                rewrite_hash(key)
              case Set:
                rewrite_set(key)
              case SortedSet:
                rewrite_sorted_set(key)
           if key.have_expire_time()
              rewrite_expire_time(key)
      #写入完毕，关闭文件
      f.close()
```

ps:在实际中，重写程序在处理列表，哈希表，集合，有序集合这四种带有多个元素的键时，会先检查键所包含的元素数量，如果元素的数量超过了redis.h/REDIS_AOF_REWRITE_ITEMS_PER_CMD 常量的值，那么重写程序将使用多条命令来记录键的值，而不是单单一条命令。在redis 2.9 版本中这个常量的值为64。

AOF后台重写

因为redis是使用单线程来处理请求命令，为了不阻塞主进程,所以AOF重写的工作会起一个子进程来进行。

但这样做的同时会导致一个问题，如果子进程在进行重写的同时，主进程继续处理命令请求，而新的命令可能会对现在的数据库状态进行修改，从而使得重写前后的文件保存的数据库状态不一致。

表 11-2　AOF 文件重写时的服务器进程和子进程

时间	服务器进程	子进程
T1	执行命令 SET k1 v1	
T2	执行命令 SET k1 v2	
T3	执行命令 SET k1 v3	
T4	创建子进程，执行 AOF 文件重写	开始 AOF 文件重写
T5	执行命令 SET k2 10086	执行重写操作
T6	执行命令 SET k3 12345	执行重写操作
T7	执行命令 SET k4 22222	完成 AOF 文件重写

为了解决这个问题，redis服务器设置了一个AOF重写缓冲区，这个缓冲区在服务器创建子进程之后开始使用，当执行完一个写命令之后，他会同时将这个写命令发送给AOF缓冲区和AOF重写缓冲区，这样子进程开始后，服务器执行的所有写命令都会被记录到AOF重写缓冲区里面，这样就能解决上面这个问题了。

在整个过程中只有重写完成后的信号处理函数会对主进程造成阻塞，其他时候都不会造成阻塞。

这就是**AOF后台重写，也就是BGREWRITEAOF命令**的实现原理。

事件

> redis服务器于客户端或者其他redis服务器通信是基于socket套接字,并且是事件驱动的。

redis需要处理以下两种事件:

1. 文件事件(file event)

   文件事件就是redis socket通信的抽象，我认为就是数据交换格式。通过监听各种文件事件来完成一系列的网络通信操作。

2. 时间事件(time event)

   redis服务器中有一系列的操作需要在指定时间执行,例如(serverCron函数)时间事件就是这类定时操作的抽象。

文件事件

redis基于Reactor模式开发了自己的网络事件处理器:这个处理器被称作文件处理器。

使用I/O多路复用来监听多个套接字,然后根据套接字当前执行的任务来关联不同的事件处理器

当套接字准备应答(accpet),读取(read),写入(write),关闭(close)时，相应的事件就会产生，然后文件处理器就会调用关联好的事件处理器来处理这些事件。

通过I/O多路复用,虽然是文件事件处理器虽然是单线程，但是却实现了高性能的网络通信模型，又方便与redis中其他的单线程模块进行对接，保持了redis内部单线程简单性。

serverCron函数

redis服务器中的serverCron函数默认每隔100ms执行一次，这个函数负责管理服务器的资源，并保持服务器自身的良好运转。

更新服务器时间缓存

redis服务器中有很多的功能需要获取系统时间，例如记录日志，设置键过期时间等，而每次获取系统的当前时间都需要执行一次系统调用，为了减少调用次数，服务器的结构体里面`unixtime`和`mstime`属性被当作当前时间的缓存。

```
struct redisServer {
  //...
  time_t unixtime; //保存秒级的当前时间戳
  long long mstime;//保存毫秒级的当前时间戳
}
```

serverCron每一百毫秒执行一次，所以这两个属性的精度不高，所以会在打印日志等需要精度不高的服务才会使用，对于为键设置过期时间等高精度功能来说，还是会去执行系统调用。

更新服务器内存峰值

服务器状态里面`stat_paek_memory`属性记录了服务器的内存峰值大小：

```
struct redisServer {

  //...
  size_t stat_peak_memory;
}
```

每次serverCron函数执行时，程序都会查看服务器当前使用的内存数量并且与`stat_peak_memory`进行比较，如果大于这个值则更新。

管理客户端资源

`serverCron`函数每次执行都会调用`clientsCron`函数，这个函数会对一定数量的客户端进行以下两个检查：

- 如果客户端与服务器之间的连接已经超时，那么程序释放这个客户端。
- 如果客户端在上一次执行命令请求后，输入缓冲区的大小超过了一定的长度，那么程序会释放客户端当前的输入缓冲区，并重新创建一个默认的输入缓冲区，从而防止客户的输入缓冲区消耗过多的内存。

管理数据库资源

`serverCron`函数每次执行都会调用`databaseCron`函数，这个函数会对服务器中的一部分数据库进行检查，删除其中过期的键，并在有需要的时候，对字典进行收缩操作

执行被延迟的BGREWRITEAOF

如果BGSAVE命令正在执行，那么BGREWRITEAOF的执行时间会被延迟到BGSAVE命令执行完成之后。

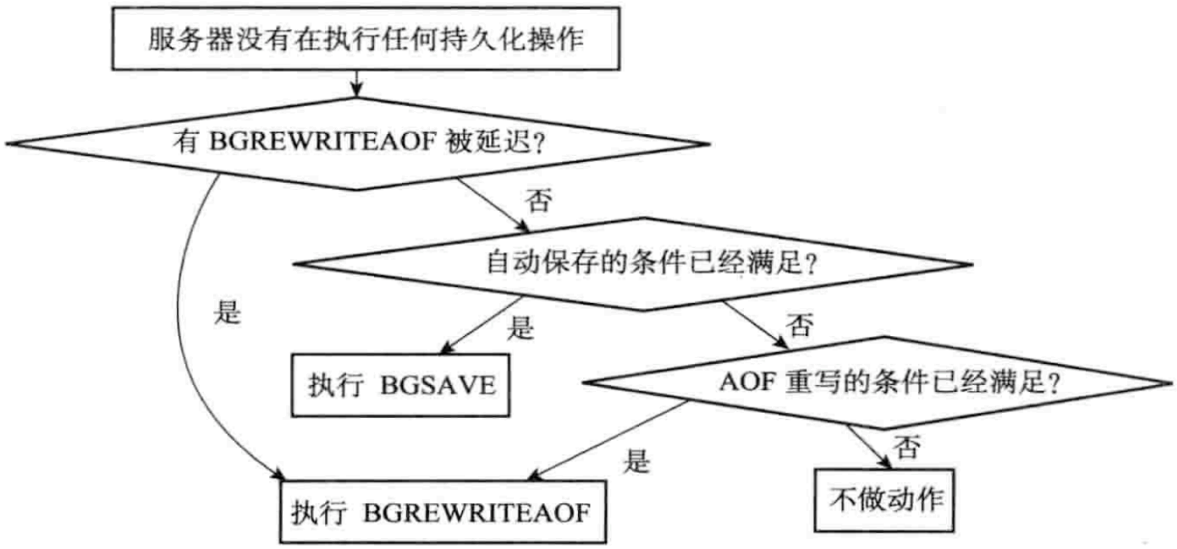
redisServer的结构体中维护一个参数

```
struct redisServer {
  int aof_rewrite_scheduled;//如果值为1，那么表示有 BGREWRITEAOF 命令被延迟了
}
```

每次serverCorn函数执行的时候，都会检查`BIGSAVE`或者`BIGREWRITEAOF`命令是否正在执行，如果都没有执行，并且aof_rewrite_scheduled属性的值为1，那么服务器就会执行被推迟的BGREWRITEAOF命令。

检查持久化操作的运行状态

用流程图来表示这个检查过程



复制

在redis里面可以通过`slaveof`命令来让一个redis服务器已复制另一个服务器，也就是主从配置

主从服务器双方的数据库将保存相同的数据

redis2.8以前

redis复制功能分为同步(sync)和命令传播(command propagate) 两个操作

- 同步操作用于将从服务器的数据库状态更新至主服务器当前所处的数据库状态
- 命令传播操作则用于在主服务器的数据库状态被修改，导致主从服务器的数据库状态出现不一致时，让主从服务器重新回到一致状态

同步

当一个服务器使用slaveof成为另一个服务器的slave时，他就需要发送同步操作给主人，步骤如下：

1. 从服务器向主服务器发送sync命令
2. 收到sync命令的主服务器执行BGSAVE命令，在后台生成rdb文件，并使用一个缓冲区记录从现在开始执行的所有命令。
3. 当主服务器的BGSAVE命令执行完毕时，主服务器会将生成的RDB文件发送至从服务器，从服务器接收并载入这个rdb文件，将自己的数据库状态更新至主服务器执行BGSAVE命令时的数据库状态。
4. 主服务器将记录在缓冲区里面的所有写命令发送给从服务器，从服务器执行这些写命令，将自己的

数据库状态更新至主服务器当前的状态

命令传播

在同步操作完成后，主从双方的数据库状态不是一成不变的，。每当主服务器执行客户端的写入命令时，双方的状态就有可能不一致。

因此，需要主服务器对从服务器进行命令传播操作:**主服务器会将自己执行的写命令，也即是造成主服务器不一致的那条命令，发送给从服务器执行，当从服务器执行完后，双方再次回到一致状态。**

旧版复制功能的缺陷

复制又分两种情况

- 初次复制，就是服务器刚开始成为slave时候。旧版复制功能能够很好的完成任务
- 断线后重新复制，处于命令传播阶段的主从服务器因为网络原因或者其他原因中断了复制，但从服务器通过自动重连重新连接上了主服务器，并继续复制主服务器，这里效率却非常低

因为断线重新复制，可能主服务器只写入了或者更新了少量的数据，而却重新进行了sync的操作，这样效率是非常低的，因为sync需要大量的磁盘IO和网络IO

redis2.8版本之后

为了解决旧版复制功能在处理断线重复情况时的低效问题，2.8版本后使用`PSYNC`命令来代替`SYNC`命令来执行复制时的同步操作。

`PSYNC`命令具有完全重同步(full resynchronization) 和部分重同步(partial resynchronization)两种模式

- 完整重同步用于处理初次复制情况：这中模式的执行步骤和SYNC命令的执行步骤是一样的。
- 部分重同步则用于处理断线后重新复制的情况：当断线重连时，如果条件允许，主服务器可以将主服务器连接断开期间执行的写命令发送给从服务器，从服务器只要接收并执行这些写命令，就可以统一主从数据库状态

部分重同步的实现

部分重同步功能由以下三个部分构成:

- 主服务器的复制偏移量(replication offset) 和从服务器的复制偏移量
- 主服务器的复制积压缓冲区(replication backlog)
- 服务器的运行ID(run ID)