# 20-AOF重写（下）：重写时的新写操作记录在哪里？

你好，我是蒋德钧。

在上节课中，我给你介绍了AOF重写过程，其中我带你重点了解了AOF重写的触发时机，以及AOF重写的基本执行流程。现在你已经知道，AOF重写是通过重写子进程来完成的。

但是在上节课的最后，我也提到了AOF重写时，主进程仍然在接收客户端写操作，那么这些新写操作会记录到AOF重写日志中吗？如果需要记录的话，重写子进程又是通过什么方式向主进程获取这些写操作的呢？

今天这节课，我就来带你了解下AOF重写过程中所使用的管道机制，以及主进程和重写子进程的交互过程。这样一方面，你就可以了解AOF重写日志包含的写操作的完整程度，当你要使用AOF日志恢复Redis数据库时，就知道AOF能恢复到的程度是怎样的。另一方面，因为AOF重写子进程就是通过操作系统提供的管道机制，来和Redis主进程交互的，所以学完这节课之后，你就可以掌握管道技术，从而用来实现进程间的通信。

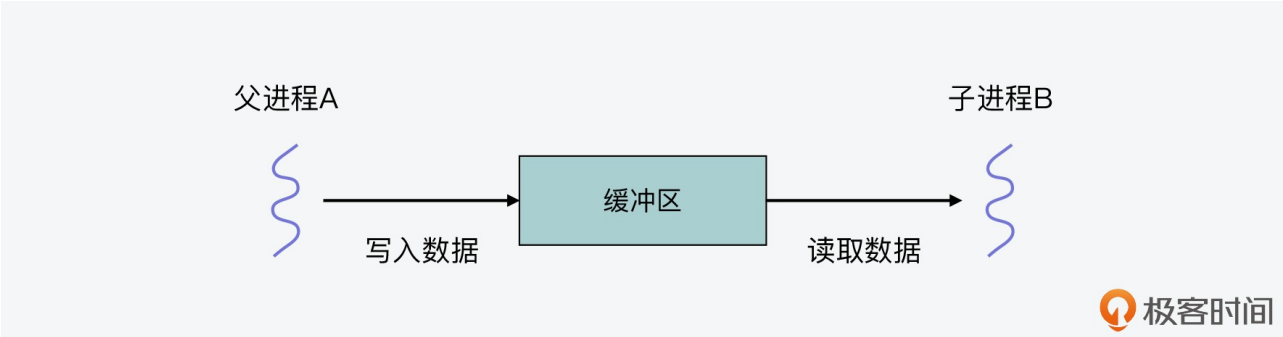
好了，接下来，我们就先来了解下管道机制。

## 如何使用管道进行父子进程间通信？

首先要知道，当进程A通过调用fork函数创建一个子进程B，然后进程A和B要进行通信时，我们通常都需要依赖操作系统提供的通信机制，而**管道**（pipe）就是一种用于父子进程间通信的常用机制。

具体来说，管道机制在操作系统内核中创建了一块缓冲区，父进程A可以打开管道，并往这块缓冲区中写入数据。同时，子进程B也可以打开管道，从这块缓冲区中读取数据。这里，你需要注意的是，进程每次往管道中写入数据时，只能追加写到缓冲区中当前数据所在的尾部，而进程每次从管道中读取数据时，只能从缓冲区的头部读取数据。其实，管道创建的这块缓冲区就像一个先进先出的队列一样，写数据的进程写到队列尾部，而读数据的进程则从队列头读取。

下图就展示了两个进程使用管道进行数据通信的过程，你可以看下。




好了，了解了管道的基本功能后，我们再来看下使用管道时需要注意的一个关键点。管道中的数据在一个时刻只能向一个方向流动，这也就是说，如果父进程A往管道中写入了数据，那么此时子进程B只能从管道中读取数据。类似的，如果子进程B往管道中写入了数据，那么此时父进程A只能从管道中读取数据。如果父子进程间需要同时进行数据传输通信，那么，我们就需要创建两个管道了。

接下来，我们来看下怎么用代码实现管道通信。这就和操作系统提供的管道的系统调用pipe有关了，pipe的函数原型如下所示：

```
int pipe(int pipefd[2]);
```

你可以看到，pipe的参数是一个数组pipefd，表示的是管道的文件描述符。这是因为进程在往管道中写入或读取数据时，其实是使用write或read函数的，而write和read函数需要通过文件描述符才能进行写数据和读数据操作。

数组pipefd有两个元素pipefd[0]和pipefd[1]，分别对应了管道的读描述和写描述符，这也就是说，当进程需要从管道中读数据时，就需要用到pipefd[0]，而往管道中写入数据时，就使用pipefd[1]。

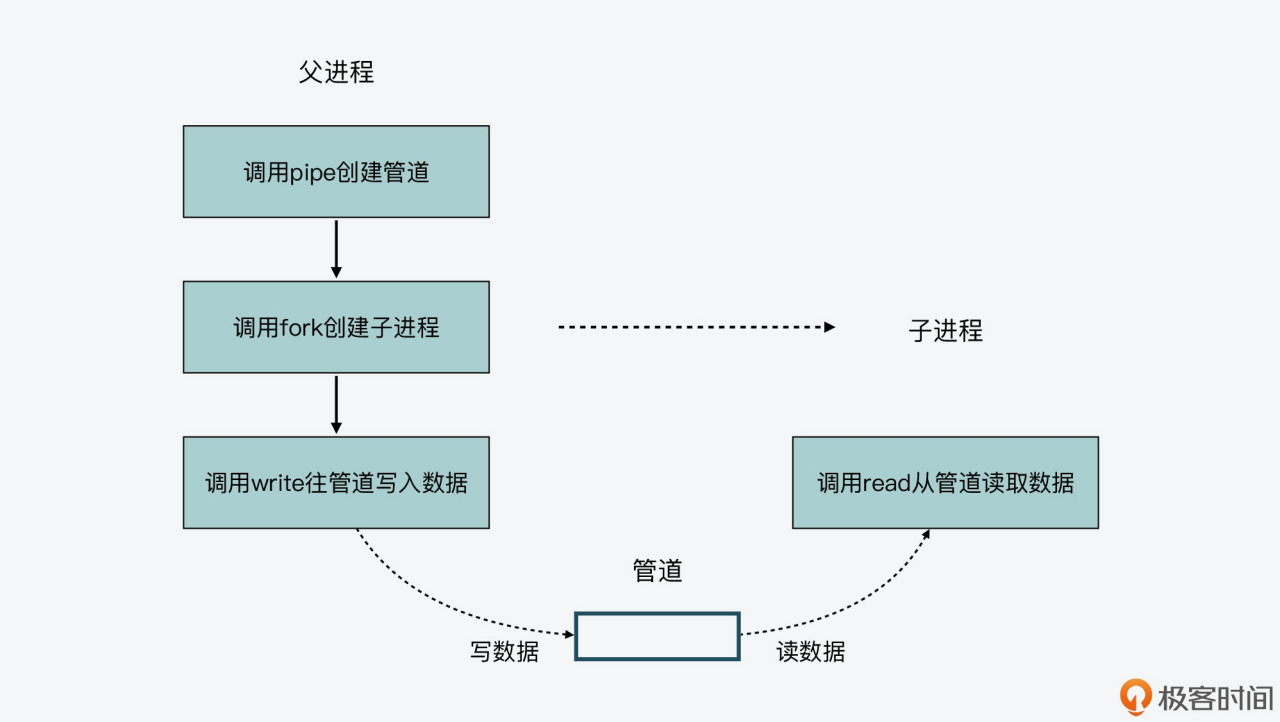
我写了一份示例代码，展示了父子进程如何使用管道通信，你可以看下。

```
int main()
{
    int fd[2], nr = 0, nw = 0;
    char buf[128];
    pipe(fd);
    pid = fork();

 if(pid == 0) {
     //子进程调用read从fd[0]描述符中读取数据
        printf("child process wait for message\n");
        nr = read(fds[0], buf, sizeof(buf))
        printf("child process receive %s\n", buf);
 }else{
      //父进程调用write往fd[1]描述符中写入数据
        printf("parent process send message\n");
        strcpy(buf, "Hello from parent");
        nw = write(fd[1], buf, sizeof(buf));
        printf("parent process send %d bytes to child.\n", nw);
    }
    return 0;
}
```

从代码中，你可以看到，在父子进程进行管道通信前，我们需要在代码中定义用于保存读写描述符的数组fd，然后调用pipe系统创建管道，并把数组fd作为参数传给pipe函数。紧接着，在父进程的代码中，父进程调用write函数往管道文件描述符fd[1]中写入数据，另一方面，子进程调用read函数从管道文件描述符fd[0]中读取数据。

这里，为了便于你理解，我也画了一张图，你可以参考。




好了，现在你就了解了如何使用管道来进行父子进程的通信了。那么下面，我们就来看下在AOF重写过程中，重写子进程是如何用管道和主进程（也就是它的父进程）进行通信的。

## AOF重写子进程如何使用管道和父进程交互？

首先，我们来看下，AOF重写过程中创建了几个管道。这是AOF重写函数rewriteAppendOnlyFileBackground在执行过程中，通过调用aofCreatePipes函数来完成的，如下所示：

```
int rewriteAppendOnlyFileBackground(void) {
…
if (aofCreatePipes() != C_OK) return C_ERR;
…
}
```

aofCreatePipes函数是在aof.c文件中实现的，它的逻辑比较简单，可以分成三步。

**第一步**，aofCreatePipes函数创建了包含6个文件描述符元素的**数组fds**。就像我刚才给你介绍的，每一个管道会对应两个文件描述符，所以，数组fds其实对应了AOF重写过程中要用到的三个管道。紧接着，aofCreatePipes函数就调用pipe系统调用函数，分别创建三个管道。

这部分的代码如下所示，你可以看下。

```
int aofCreatePipes(void) {
    int fds[6] = {-1, -1, -1, -1, -1, -1};
    int j;
    if (pipe(fds) == -1) goto error; /* parent -> children data. */
    if (pipe(fds+2) == -1) goto error; /* children -> parent ack. */
 if (pipe(fds+4) == -1) goto error;
 …}
}
```

**第二步**，aofCreatePipes函数会调用anetNonBlock函数（在anet.c文件中），将fds数组的第一和第二个描述符（fds[0]和fds[1]）对应的管道设置为非阻塞。然后，aofCreatePipes函数调用aeCreateFileEvent函数在数组fds的第三个描述符(fds[2])上注册了读事件的监听，对应的回调函数是aofChildPipeReadable。aofChildPipeReadable函数也是在aof.c文件中实现的，我稍后会给你具体介绍。

```
int aofCreatePipes(void) {
…
if (anetNonBlock(NULL,fds[0]) != ANET_OK) goto error;
if (anetNonBlock(NULL,fds[1]) != ANET_OK) goto error;
if (aeCreateFileEvent(server.el, fds[2], AE_READABLE, aofChildPipeReadable, NULL) == AE_ERR) goto error;
…
}
```

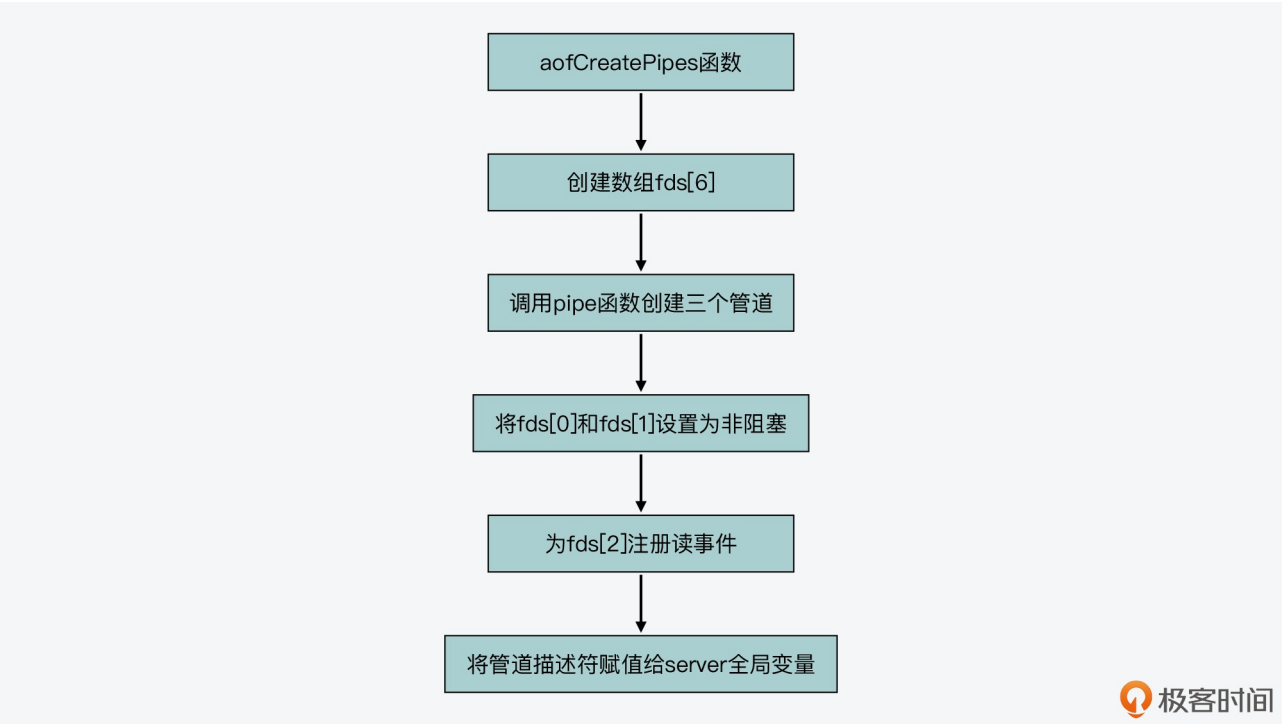
在完成了管道创建、管道设置和读事件注册后，最后一步，aofCreatePipes函数将数组fds中的六个文件描述符分别复制给server变量的成员变量，如下所示：

```
int aofCreatePipes(void) {
…
server.aof_pipe_write_data_to_child = fds[1];
server.aof_pipe_read_data_from_parent = fds[0];
server.aof_pipe_write_ack_to_parent = fds[3];
server.aof_pipe_read_ack_from_child = fds[2];
server.aof_pipe_write_ack_to_child = fds[5];
server.aof_pipe_read_ack_from_parent = fds[4];
…
}
```

在这一步中，我们就可以从server变量的成员变量名中，看到aofCreatePipes函数创建的三个管道，以及它们各自的用途。

- **fds[0]和fds[1]**：对应了主进程和重写子进程间用于传递操作命令的管道，它们分别对应读描述符和写描述符；
- **fds[2]和fds[3]**：对应了重写子进程向父进程发送ACK信息的管道，它们分别对应读描述符和写描述符；
- **fds[4]和fds[5]**：对应了父进程向重写子进程发送ACK信息的管道，它们分别对应读描述符和写描述符。

下图也展示了aofCreatePipes函数的基本执行流程，你可以再回顾下。




好了，了解了AOF重写过程中的管道个数和用途后，我们来看下这些管道具体是如何使用的。

### 操作命令传输管道的使用

当AOF重写子进程执行时，主进程还会继续接收和处理客户端写请求。这些写操作会被主进程正常写入AOF日志文件，这个过程是由feedAppendOnlyFile函数（在aof.c文件中）来完成。

feedAppendOnlyFile函数在执行的最后一步，会判断当前是否有AOF重写子进程在运行。如果有的话，那么，它会调用aofRewriteBufferAppend函数（在aof.c文件中），如下所示：

```
if (server.aof_child_pid != -1)
        aofRewriteBufferAppend((unsigned char*)buf,sdslen(buf));
```

aofRewriteBufferAppend函数的作用是将参数buf，追加写到全局变量server的aof_rewrite_buf_blocks这个列表中。

这里，你需要注意的是，参数buf是一个字节数组，feedAppendOnlyFile函数将主进程收到的命令操作写入到buf中。而aof_rewrite_buf_blocks列表中的每个元素是aofrwblock结构体类型，这个结构体中包括了一个字节数组，大小是AOF_RW_BUF_BLOCK_SIZE，默认值是10MB。此外，aofrwblock结构体还记录了字节数组已经使用的空间和剩余可用的空间。

以下代码展示了aofrwblock结构体的定义，你可以看下。

```
typedef struct aofrwblock {
    unsigned long used, free; //buf数组已用空间和剩余可用空间
    char buf[AOF_RW_BUF_BLOCK_SIZE]; //宏定义AOF_RW_BUF_BLOCK_SIZE默认为10MB
} aofrwblock;
```

这样一来，aofrwblock结构体就相当于是一个10MB的数据块，记录了AOF重写期间主进程收到的命令，而aof_rewrite_buf_blocks列表负责将这些数据块连接起来。当aofRewriteBufferAppend函数执行时，它会从aof_rewrite_buf_blocks列表中取出一个aofrwblock类型的数据块，用来记录命令操作。

当然，如果当前数据块中的空间不够保存参数buf中记录的命令操作，那么aofRewriteBufferAppend函数就会再分配一个aofrwblock数据块。

好了，当aofRewriteBufferAppend函数将命令操作记录到aof_rewrite_buf_blocks列表中之后，它会检查aof_pipe_write_data_to_child管道描述符上是否注册了写事件，这个管道描述符就对应了我刚才向你介绍的fds[1]。

如果没有注册写事件，那么aofRewriteBufferAppend函数就会调用aeCreateFileEvent函数，注册一个写事件，这个写事件会监听aof_pipe_write_data_to_child这个管道描述符，也就是主进程和重写子进程间的操作命令传输管道。

当这个管道可以写入数据时，写事件对应的回调函数aofChildWriteDiffData（在aof.c文件中）会被调用执行。这个过程你可以看下下面的代码。

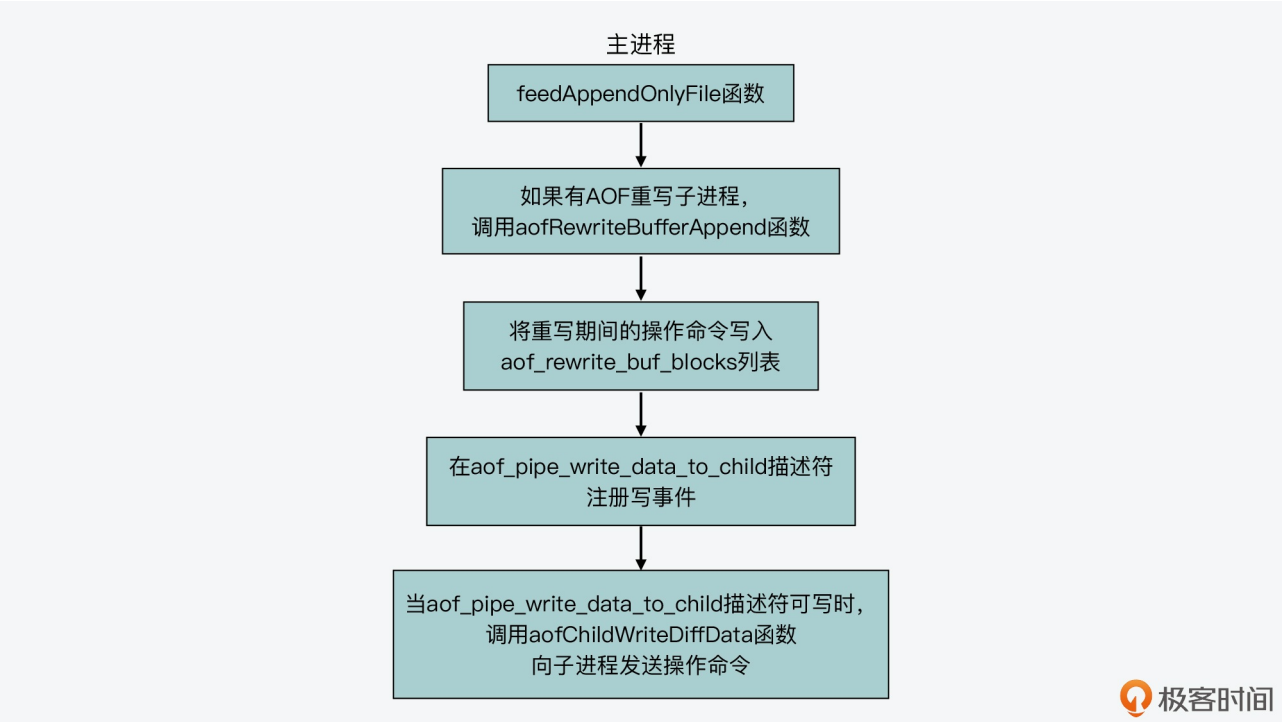
```
void aofRewriteBufferAppend(unsigned char *s, unsigned long len) {
...
//检查aof_pipe_write_data_to_child描述符上是否有事件
if (aeGetFileEvents(server.el,server.aof_pipe_write_data_to_child) == 0) {
     //如果没有注册事件，那么注册一个写事件，回调函数是aofChildWriteDiffData
     aeCreateFileEvent(server.el, server.aof_pipe_write_data_to_child,
            AE_WRITABLE, aofChildWriteDiffData, NULL);
}
...}
```

刚才介绍的写事件回调函数aofChildWriteDiffData，它的主要作用是从aof_rewrite_buf_blocks列表中逐个取出数据块，然后通过aof_pipe_write_data_to_child管道描述符，将数据块中的命令操作通过管道发给重写子进程，这个过程如下所示：

```
void aofChildWriteDiffData(aeEventLoop *el, int fd, void *privdata, int mask) {
...
while(1) {
   //从aof_rewrite_buf_blocks列表中取出数据块
   ln = listFirst(server.aof_rewrite_buf_blocks);
   block = ln ? ln->value : NULL;
   if (block->used > 0) {
      //调用write将数据块写入主进程和重写子进程间的管道
      nwritten = write(server.aof_pipe_write_data_to_child,
                           block->buf,block->used);
      if (nwritten <= 0) return;
            ...
        }
 ...}}
```

好了，到这里，你就了解了主进程其实是在正常记录AOF日志时，将收到的命令操作写入aof_rewrite_buf_blocks列表中的数据块，然后再通过aofChildWriteDiffData函数将记录的命令操作通过主进程和重写子进程间的管道发给子进程。

下图就展示了这个过程，你可以看下。




接下来，我们来看下重写子进程是如何从管道中读取父进程发送的命令操作的，这是由aofReadDiffFromParent函数（在aof.c文件中）来完成的。这个函数使用一个64KB大小的缓冲区，然后调用read函数，读取父进程和重写子进程间的操作命令传输管道中的数据。下面的代码展示了aofReadDiffFromParent函数的基本执行流程，你可以看下。

```
ssize_t aofReadDiffFromParent(void) {
    char buf[65536]; //管道默认的缓冲区大小
    ssize_t nread, total = 0;
    //调用read函数从aof_pipe_read_data_from_parent中读取数据
    while ((nread =
      read(server.aof_pipe_read_data_from_parent,buf,sizeof(buf))) > 0) {
        server.aof_child_diff = sdscatlen(server.aof_child_diff,buf,nread);
        total += nread;
    }
    return total;
}
```

从代码中，你可以看到，aofReadDiffFromParent函数通过aof_pipe_read_data_from_parent描述符读取数据。然后，它将读取的操作命令追加到全局变量server的aof_child_diff字符串中。而在AOF重写函数rewriteAppendOnlyFile的执行过程最后，aof_child_diff字符串会被写入AOF重写日志文件，以便我们使用AOF重写日志时，能尽可能地恢复重写期间收到的操作。

下面的代码展示aof_child_diff字符串写入重写日志文件的过程，你可以看下。

```
int rewriteAppendOnlyFile(char *filename) {
...
//将aof_child_diff中累积的操作命令写入AOF重写日志文件
if (rioWrite(&aof,server.aof_child_diff,sdslen(server.aof_child_diff)) == 0)
        goto werr;
...
}
```

好了，到这里，我们了解了，aofReadDiffFromParent函数实现了重写子进程向主进程读取操作命令。那么，我们再看下，aofReadDiffFromParent函数会在哪里被调用，也就是重写子进程会在什么时候从管道中读取主进程收到的操作。

其实，aofReadDiffFromParent函数一共会被以下三个函数调用。

- **rewriteAppendOnlyFileRio函数**：这个函数是由重写子进程执行的，它负责遍历Redis每个数据库，生成AOF重写日志，在这个过程中，它会不时地调用aofReadDiffFromParent函数；
- **rewriteAppendOnlyFile函数**：这个函数是重写日志的主体函数，也是由重写子进程执行的。它本身会调用rewriteAppendOnlyFileRio函数。此外，它在调用完rewriteAppendOnlyFileRio函数后，还会调用aofReadDiffFromParent函数多次，尽可能多地读取主进程在重写日志期间收到的操作命令；
- **rdbSaveRio函数**：这个函数是创建RDB文件的主体函数。当我们使用AOF和RDB混合持久化机制时，这个函数也会调用aofReadDiffFromParent函数。

从这里，我们可以看到，Redis源码在实现AOF重写过程中，其实会多次让重写子进程向主进程读取新收到的操作命令，这也是为了让重写日志尽可能多地记录最新的操作，提供更加完整的操作记录。

最后，我们再来看下重写子进程和主进程间用来传递ACK信息的两个管道的使用。

### ACK管道的使用

刚才我向你介绍主进程调用aofCreatePipes函数创建管道时，我们就了解到了，主进程会在aof_pipe_read_ack_from_child管道描述符上注册读事件。这个描述符对应了重写子进程向主进程发送ACK信息的管道。同时，这个描述符是一个读描述符，表示主进程从管道中读取ACK信息。

其实，重写子进程在执行rewriteAppendOnlyFile函数时，这个函数在完成日志重写，以及多次向父进程读取操作命令后，就会调用write函数，向aof_pipe_write_ack_to_parent描述符对应的管道中写入“！”，这就是重写子进程向主进程发送ACK信号，让主进程停止发送收到的新写操作。这个过程如下所示：

```
int rewriteAppendOnlyFile(char *filename) {
...
if (write(server.aof_pipe_write_ack_to_parent,"!",1) != 1) goto werr;
...}
```

一旦重写子进程向主进程发送ACK信息的管道中有了数据，aof_pipe_read_ack_from_child管道描述符上注册的读事件就被触发，也就是说，这个管道中有数据可以读取了。那么，aof_pipe_read_ack_from_child管道描述符上注册的回调函数aofChildPipeReadable（在aof.c文件中）就

会执行。

这个函数会判断从aof_pipe_read_ack_from_child管道描述符读取的数据是否是“！”，如果是的话，那它就会调用write函数往aof_pipe_write_ack_to_child管道描述符上写入“！”，表示主进程已经收到重写子进程发送的ACK信息，同时它给重写子进程回复一个ACK信息。这个过程如下所示：

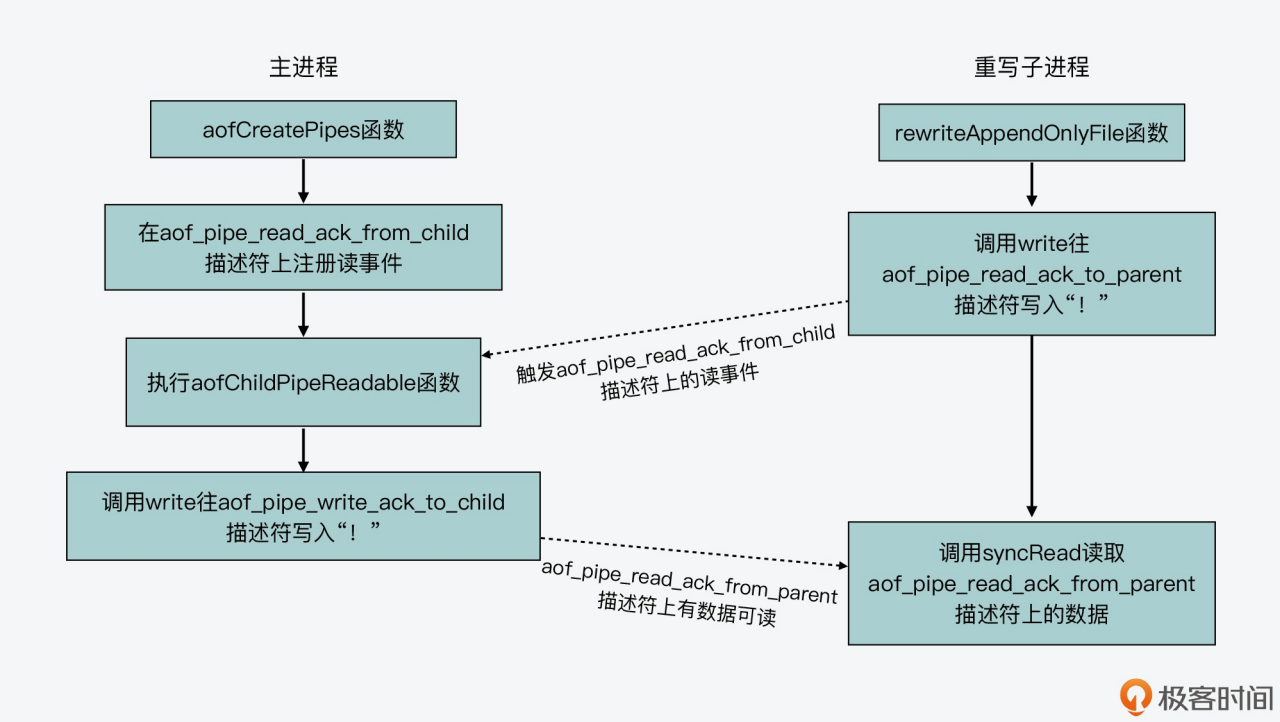
```
void aofChildPipeReadable(aeEventLoop *el, int fd, void *privdata, int mask) {
...
if (read(fd,&byte,1) == 1 && byte == '!') {
   ...
   if (write(server.aof_pipe_write_ack_to_child,"!",1) != 1) { ...}
}
...
}
```

好了，到这里，我们就了解了，重写子进程在完成日志重写后，是先给主进程发送ACK信息。然后主进程在aof_pipe_read_ack_from_child描述符上监听读事件发生，并调用aofChildPipeReadable函数向子进程发送ACK信息。

最后，重写子进程执行的rewriteAppendOnlyFile函数，它会调用syncRead函数，从aof_pipe_read_ack_from_parent管道描述符上读取主进程发送给它的ACK信息，如下所示：

```
int rewriteAppendOnlyFile(char *filename) {
...
if (syncRead(server.aof_pipe_read_ack_from_parent,&byte,1,5000) != 1  || byte != '!') goto werr
...
}
```

下图也展示了ACK管道的使用过程，你可以再回顾下。




这样一来，重写子进程和主进程之间就通过两个ACK管道，相互确认重写过程结束了。

## 小结

今天这节课，我给你介绍了AOF重写过程中，主进程和重写子进程间的管道通信。我先是带你了解了管道机制的使用。

然后，我们学习了主进程和重写子进程使用管道通信的过程。在这个过程中，AOF重写子进程和主进程使用了一个操作命令传输管道和两个ACK信息发送管道。操作命令传输管道用于主进程写入收到的新操作命令，以及用于重写子进程读取操作命令，而ACK信息发送管道是在重写结束时，重写子进程和主进程用来相互确认重写过程的结束。最后，重写子进程进一步将收到的操作命令记录到重写日志文件中。

这样一来，AOF重写过程中主进程收到的新写操作，并不会被遗漏。一方面，这些新写操作会被记录在正常的AOF日志中，另一方面，主进程会将新写操作缓存在aof_rewrite_buf_blocks数据块列表中，并通过管道发送给重写子进程。这就能尽可能保证重写日志具有最新、最完整的写操作了。

最后，我也再提醒你一下，今天这节课我们学习的管道其实属于匿名管道，是用在父子进程间进行通信的。如果你在实际开发中，要在非父子进程的两个进程间进行通信，那么你就需要用到命名管道了。而命名管道会以一个文件的形式保存在文件系统中，并会有相应的路径和文件名。因此，非父子进程的两个进程通过命名管道的路径和文件名，就可以打开管道进行通信了。

## 每课一问

今天这节课，我给你介绍了重写子进程和主进程间进行操作命令传输、ACK信息传递用的三个管道。那么，你在Redis源码中还能找见其他使用管道的地方吗？

## 精选留言：

- Kaito 2021-09-11 02:28:46
  1、AOF 重写是在子进程中执行，但在此期间父进程还会接收写操作，为了保证新的 AOF 文件数据更完整，所以父进程需要把在这期间的写操作缓存下来，然后发给子进程，让子进程追加到 AOF 文件中

2、因为需要父子进程传输数据，所以需要用到操作系统提供的进程间通信机制，这里 Redis 用的是「管道」，管道只能是一个进程写，另一个进程读，特点是单向传输

3、AOF 重写时，父子进程用了 3 个管道，分别传输不同类别的数据：

- 父进程传输数据给子进程的管道：发送 AOF 重写期间新的写操作
- 子进程完成重写后通知父进程的管道：让父进程停止发送新的写操作
- 父进程确认收到子进程通知的管道：父进程通知子进程已收到通知

4、AOF 重写的完整流程是：父进程 fork 出子进程，子进程迭代实例所有数据，写到一个临时 AOF 文件，在写文件期间，父进程收到新的写操作，会先缓存到 buf 中，之后父进程把 buf 中的数据，通过管道发给子进程，子进程写完 AOF 文件后，会从管道中读取这些命令，再追加到 AOF 文件中，最后 rename 这个临时 AOF 文件为新文件，替换旧的 AOF 文件，重写结束

课后题：Redis 中其它使用管道的地方还有哪些?

在源码中搜索 pipe 函数，能看到 server.child_info_pipe 和 server.module_blocked_pipe 也使用了管道。

其中 child_info_pipe 管道如下：

```
/* Pipe and data structures for child -> parent info sharing. */
int child_info_pipe[2]; /* Pipe used to write the child_info_data. */
struct {
int process_type; /* AOF or RDB child? */
size_t cow_size; /* Copy on write size. */
unsigned long long magic; /* Magic value to make sure data is valid. */
} child_info_data;
```

从注释能看出，子进程在生成 RDB 或 AOF 重写完成后，子进程通知父进程在这期间，父进程「写时复制」了多少内存，父进程把这个数据记录到 server 的 stat_rdb_cow_bytes / stat_aof_cow_bytes 下（childinfo.c 的 receiveChildInfo 函数），以便客户端可以查询到最后一次 RDB 和 AOF 重写期间写时复制时，新申请的内存大小。

而 module_blocked_pipe 管道主要服务于 Redis module。

```
/* Pipe used to awake the event loop if a client blocked on a module command needs to be processed. */
int module_blocked_pipe[2];
```

看注释是指，如果被 module 命令阻塞的客户端需要处理，则会唤醒事件循环开始处理。
[1赞]

- 可怜大灰狼 2021-09-11 06:59:35
回答问题。
1.rdbSaveBackground。2.rdbSaveToSlavesSockets，diskless模式下直接把rdb通过socket发送给slave。3.moduleInitModulesSystem，和第三方子模块通信。4.linuxMadvFreeForkBugCheck，检查MADV_FREE在arm64 Linux内核上bug，具体见https://github.com/redis/redis/commit/b02780c41dbc5b28d265b5cf141c03c1a7383ef9。