Redis源代码分析(下)

文 / 阮若夷

Redis是一个开源Key-Value内存数据库，以支持丰富的数据结构著称，支持主从复制、持久化等高可用特性，可以和程序无缝结合，本文继续上期分析Redis的源代码方法，帮助大家了解这一款数据库的工作机理。

服务端处理过程

图1涵盖了接收request、处理请求、调用函数、发送reply的过程。

我们先从Redis的网络事件库开始。Redis并没有使用libevent等第三方网络框架，而是自己实现一个reactor的事件分离器，封装了epoll、select、kqueue，代码精练，只有1KB多的代码。

开启监听后，会给监听fd注册一个文件事件，当fd有数据可读状况下，执行acceptTcpHandler函数。

客户端发起连接时，从epoll_wait返回，由于监听的fd处于可读状况，调用先前注册过的acceptTcpHandler函数，产生一个连接fd，创建一个redisClient对象，并为接下来的读取数据注册一个readQueryFromClient的只读事件，把这些都存储在aeEventLoop里的events里，fd就是数组的index，clientData就是redisClient，rfileProc保存函数。处理完监听fd后，就处理这个连接fd，执行刚才注册的readQueryFromClient函数。

readQueryFromClient先从连接fd中读取数据，先存储在c→querybuf里（networking.c 823）。接下来函数processInputBuffer来解析querybuf，上面说过如果是telnet发送的裸协议数据是没有*打头的表示参数个数的辅助信息，针对telnet的数据跳到processInlineBuffer函数，而其他则通过函数processMultibulkBuffer。这两个函数的作用一样，解析c→querybuf的字符串，分解成多参数到c→argc和c→argv里面，argc表示参数的个数，argv是个redis_object的指针数组。每个指针指向一个redis_object，object的ptr里存储具体的内容，对“get a”的请求转化后，argc就是2，argv就是：

```
(gdb) p (char*)(*c->argv[0])->ptr
$28 = 0x80ea5ec "get"
(gdb) p (char*)(*c->argv[1])->ptr
$26 = 0x80e9fc4 "a"
```

协议解析后就执行命令。processCommand首先调用lookupCommand找到get对应的函数。上面讲过从redisServer→commands这个hash表查找对应函数。这里就查找到的是getGenericCommand（t_string.c 62行）。

getGenericCommand的工作原理非常简单，根据redisServer→db这个hash table查询“a”这个key，并通过applyReply函数把需要返回的数据存储在c→buf上，字符串的长度为c→bufpos。然后需要把数据写入到连接fd中，所以为这个fd添加一个可写的函数sendReplyToClient。

等到事件分离器开始处理可写事件时，就会调用sendReplyToClient函数，同样redisClient对象从文件事件的clientData里获得。sendReplyToClient函数从c→buf上把数据写到连接的fd里，发送的长度记录在c→sendlen，直到

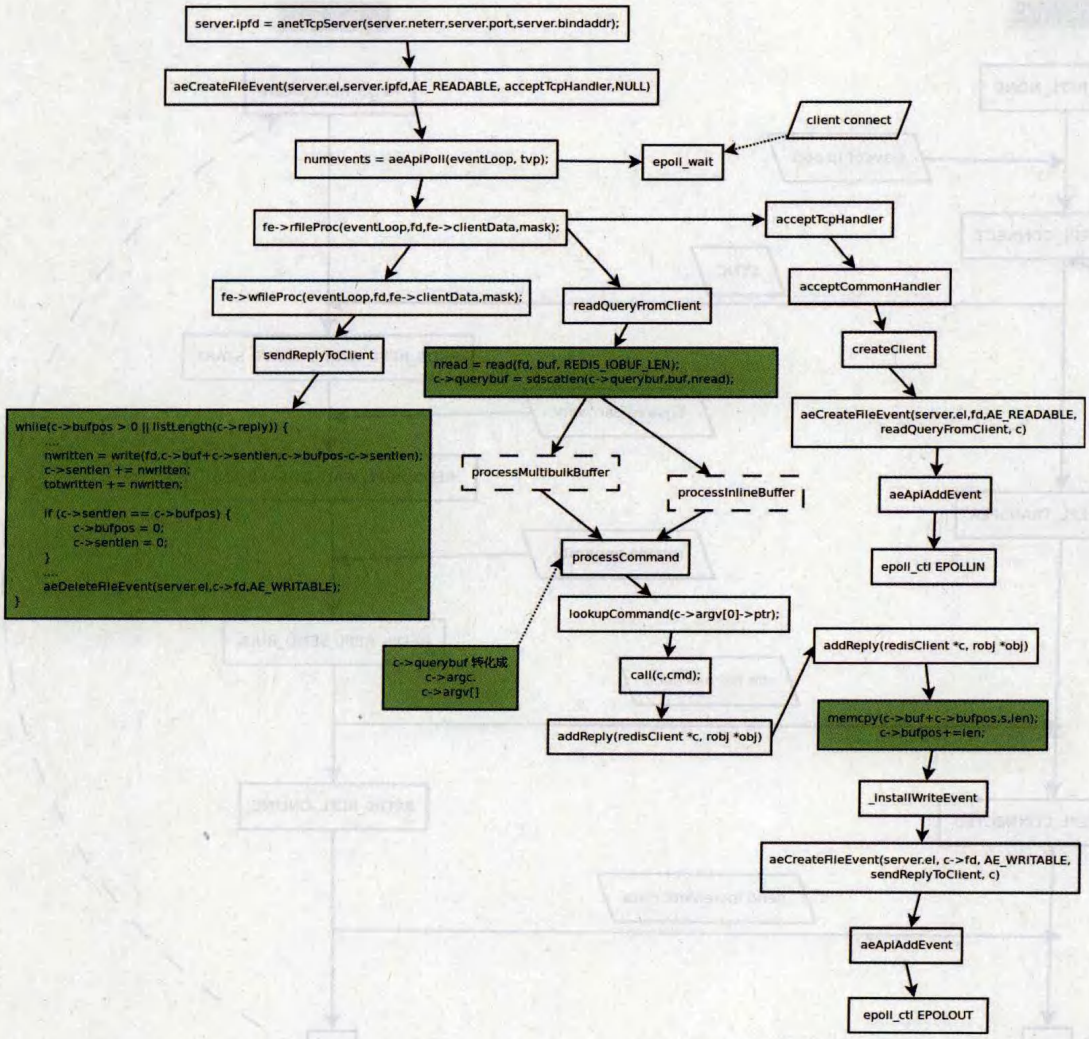

图1 接收request、处理请求、调用函数、发送reply的过程

c→bufpos==c→sendlen就可以认为buf内的数据都已经发送出去。

回复协议

下面介绍一下Redis服务端的reply协议。reply协议分很多种，有bulk replies、err message、integer reply、status reply、Multi-bulk replies。

■ bulk replies

是以$打头消息体，格式$值长度\r\n值\r\n，一般的get命令返回的结果就是这种个格式。

```
redis>get aaa
$3\r\n
bbb\r\n
```

对应的的处理函数addReplyBulk：

addReplyBulkLen(c,obj);

addReply(c,obj);

addReply(c,shared.crlf);

■ error message

以-ERR打头的消息体，后面跟着出错的信息，以\r\n结尾，针对命令出错。

```
redis>d
-ERR unknown command 'd'\r\n
```

处理的函数是addReplyError：

addReplyString(c,"-ERR ",5);

addReplyString(c,s,len);

addReplyString(c,"\r\n",2);

■ integer reply

以:打头，后面跟着数字和\r\n：

```
redis>incr a
:2\r\n
```

处理函数是：

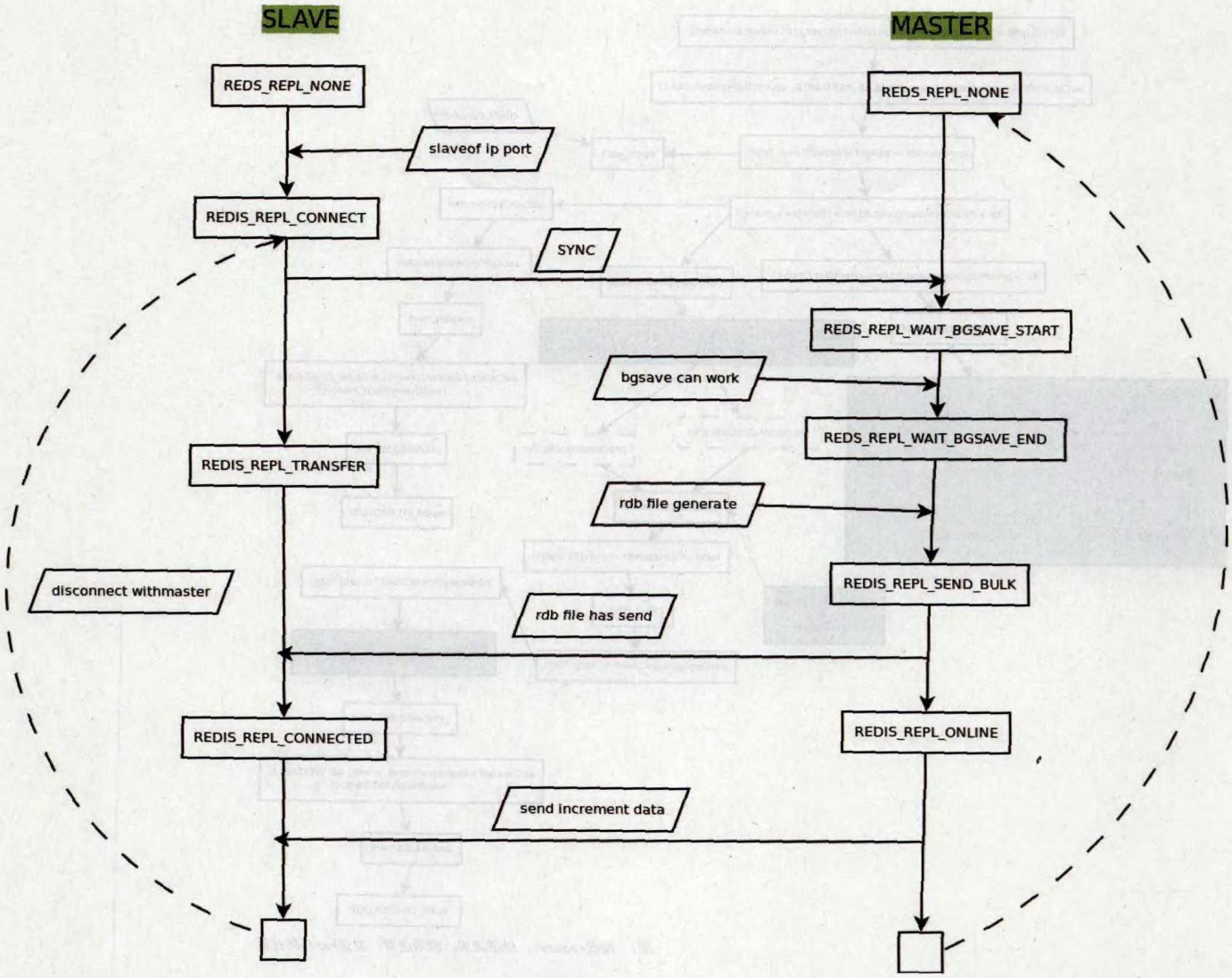

图2 Redis的复制原理

```
addReply(c,shared.colon);
addReply(c,o);
addReply(c,shared.crlf);
```

■ status reply

以+打头，后面直接跟状态内容和\r\n：

```
redis>ping
+PONG\r\n
```

这里要注意reply的内容经过协议加工后，都会先保存在c→buf里，c→bufpos表示buf的长度。待到事件分离器转到写出操作（sendReplyToClient）的时候，就把c→buf的内容写入到fd里，c→sentlen表示写出长度。当c→sentlen=c→bufpos才算写完。

■ Multi-bulk replies

lrange、hgetall这类函数通常需要返回多个值，这种消息结构与请求的格式一模一样。相关的函数是setDeferredMultiBulkLength。临时数据存储在链表c→reply里，处理方式同其他的协议格式。

复制

Redis复制的原理和使用都非常简单（如图2）。只需要在Slave端键入：

```
slaveof masterip port
```

就开启了从Master到Slave的复制。一个Master可以有多个Slave，Master有变化时会主动把命令传播给每个Slave。Slave同时可以作为其他Slave的Master，前提条件是这个Slave已经处

于稳定状态（REDIS_REPL_CONNECTED）。Slave在复制的开始阶段处于阻塞状态（sync_readline），无法对外提供服务。

取消复制，从Slave状态的转换回Master状态，切断与原Master的数据同步。

```
slaveof no one
```

Slave端接收到客户端的slaveof masterip port命令之后，调度slaveofCommand保存的masterip、port，修改server.replstate为REDIS_REPL_CONNECT，然后返回给客户端，这个复制的行为是异步的。

Slave端主线程在时间事件serverConn（redis.c 518行）里执行replicationCron（redis.c 646行）开始与Master的连接。syncWithMaster函数与Master的通信。经过校验之后（如果需要），会发送一个SYNC command给Master端，然后打开一个临时文件用于接收接下来Master发过来的rdb文件数据。再添加一个文件事件注册readSyncBulkPayload函数，这个就是接下来用于接收rdb文件的数据的函数，然后修改状态为REDIS_REPL_TRANSFER。

Master接收到SYNC command后，跳转到syncCommand函数（replication.c 556行）。syncCommand会调度rdbSaveBackground函数，启动一个子进程做一个全库的快照，并把状态改为REDIS_REPL_WAIT_BGSAVE_END。Master的主线程的serverCron会检查这个持久化的子进程是否退出。

```
if ((pid = wait3(&statloc,WNOHANG,NULL))
!= 0) {
        if (pid == server.bgsavechildpid)
{
                backgroundSaveDoneHandler
(statloc);
}
```

如果bgsave子进程正常退出，会调用backgroundSaveDoneHandler函数继续复制的工作，该函数打开刚刚产生的rdb文件。然后注册一个sendBulkToSlave函数用于发送rdb文件，状态切换至REDIS_REPL_SEND_BULK。sendBulkToSlave作用就是根据上面打开的rdb文件，读取并发送到slave端，当文件全部发送完毕之后修改状态为REDIS_REPL_ONLINE。

我们回到Slave，上面讲到Slave通过readSyncBulkPayload接收rdb数据，接收完整个rdb文件后，会清空整个数据库emptyDb()（replication.c 374）。然后就通过rdbLoad函数装载接收到的rdb文件，这样Slave和Master数据就一致了，然后把状态修改为REDIS_REPL_CONNECTED。

接下来就是Master和Slave之间增量传递的增量数据，另外Slave和Master在应用层有心跳检测（replication.c 543）和超时退出（replication.c 511）。

持久化

Redis有全量（save/bgsave）和增量（aof）的持久化命令。

全量持久化

遍历所有的RedisDB，读取每个bucket里链表的Key和Value并写入dump.rdb文件（rdb.c 405）。save命令直接调度rdbSave函数，这会阻塞主线程的工作，通常我们使用bgsave。

bgsave命令调度rdbSaveBackground函数启动了一个子进程然后调度了rdbSave函数，子进程的退出状态由serverCron的backgroundSaveDoneHandler来判断，这个在前面复制章节已经提及。

除了直接的save、bgsave命令之外，还有几个地方还调用到rdbSaveBackground和rdbSave函数。

shutdown：Redis关闭调度的prepareFor-Shutdown会做一次持久化工作，保证重启后数据依然存在，会调用rdbSave()。

flushallCommand：清空Redis数据后，如果不做立即执行一个rdbSave()，生成一个空的快照出现crash后，可能会载入含有老数据的快照。

```
void flushallCommand(redisClient *c) {
touchWatchedKeysOnFlush(-1);
server.dirty += emptyDb();      //
清空数据
addReply(c,shared.ok);
if (server.bgsavechildpid != -1) {
        kill(server.
bgsavechildpid,SIGKILL);
        rdbRemoveTempFile(server.
bgsavechildpid);
}
rdbSave(server.dbfilename);     //
没有数据的dump.db
```

```
        server.dirty++;
}
```

sync：当Master接收到Slave发来的该命令时，会执行rdbSaveBackground，这个以前也有提过。

数据发生变化：在多少秒内出现了多少次变化则触发一次bgsave，这个可以在conf里配置。

```
for (j = 0; j < server.saveparamslen; j++)
{
        struct saveparam *sp = server.
saveparams+j;
        if (server.dirty >= sp->changes &&
now-server.lastsave > sp->seconds) {
                rdbSaveBackground(server.
dbfilename);
                break;
    }
}
```

增量持久化

aof原理有点类似redo log。每次执行命令后如出现数据变化，会调用feedAppendOnlyFile，把数据写到server.aofbuf里。

```
void call(redisClient *c, struct
redisCommand *cmd) {
        long long dirty;
        dirty = server.dirty;
        cmd->proc(c);          //执行命令
        dirty = server.dirty-dirty;
        if (server.appendonly && dirty)
                feedAppendOnlyFile(cmd,c-
>db->id,c->argv,c->argc);
```

待到下次循环的before_sleep函数会通过flushAppendOnlyFile函数把server.aofbuf里的数据write到append file里。在redis.conf里配置每次write到append file后从page cache刷新到disk的规律。

```
#appendfsync always
appendfsync everysec
#appendfsync no
```

该参数的原理MySQL的innodb_flush_log_at_trx_commit一样，是个比较影响I/O的一个参数，需要在高性能和不丢数据之间做平衡。软件的优化就是平衡的过程，没有银弹。

为什么会先写到server.aofbuf，而不是直接写到aof文件内，这是一个优化的过程，合并多次的小I/O成一次大的连续I/O。

先写到server.aofbuf，然后再写到数据文件，过程中如果crash会不会丢数据呢？

答案是不会，我们先看看网络事件库如何处理读写事件：

```
if (fe->mask & mask & AE_READABLE) {
        rfired = 1;
        fe->rfileProc(eventLoop,fd,fe-
>clientData,mask);
}
if (fe->mask & mask & AE_WRITABLE) {
if(!rfired||fe->wfileProc!=fe-
>rfileProc)
fe->wfileProc(eventLoop,fd,fe-
>clientData,mask);
}
```

rfired变量决定了在一次文件事件循环内，如果对于某个fd触发了可读事件的函数，不会再次触发写事件。我们来看函数执行的步骤：

```
readQueryFromClient()
call()
feedAppendOnlyFile()
因为rfired原因退出本次循环
下一次循环
beforeSleep()-->flushAppendOnlyFile()
aeMain()--->sendReplyToClient()
```

只有执行完了flush之后才会通知客户端数据写成功了，所以如果在feed和flush之间crash，客户接会因为进程退出接受到一个fin包，也就是一个错误的返回，所以数据并没有丢，只是执行出了错。

redis crash后，重启除了利用rdb重新载入数据外，还会读取append file（redis.c 1561）加载镜像之后的数据。

激活aof，可以在redis.conf配置文件里设置：

```
appendonly yes
```

也可以通过config命令在运行态启动aof：

```
cofig set appendonly yes
```

aof最大的问题就是随着时间append file会变得很大，所以我们需要bgrewriteaof命令重新整理文件，只保留最新的K-V数据，会调用rewriteAppendOnlyFile这个函数，该函数与rdbSave和类似。保存全库的K-V数据。

在作快照的过程中，K-V的变化是先写到aofbuf里。如果存在bgrewritechildpid进程，变化数据还要写到server.bgrewritebuf里（aof.c 177行）。等子进程完成快照退出之时，再由backgroundRewriteDoneHandler函数合并bgrewritebuf和全镜像两部分数据（aof.c 673行）。合并后的aof文件才是最新的全库的镜像数据。

阮若夷

目前就职于阿里云计算运维部，是阿里云关系型数据库云服务（RDS）和鹰眼监控的开发者。

博客：www.hoterran.info

微博：weibo.com/hoterran

责任编辑：高松（gaosong@csdn.net）