本文主要讲解主节点部分重同步的实现，以及主从复制中的其他功能。本文是Redis主从复制机制的最后一篇文章。

主节点在收到从节点发来的PSYNC命令之前，主节点的部分重同步流程，与完全重同步流程是一样的。在收到PSYNC命令后，主节点调用masterTryPartialResynchronization函数，尝试进行部分重同步。

首先看一下部分重同步的实现原理，然后在看具体的实现。

**一：部分重同步原理**

Redis实现的部分重同步功能，依赖于以下三个属性：

a：主节点的复制偏移量和从节点的复制偏移量；

b：主节点的复制积压队列( replication backlog )；

c：Redis实例的运行ID；

主节点维持一个积压队列。当它收到客户端发来的命令请求时，除了将该命令请求缓存到从节点的输出缓存，还会将命令追加到积压队列中。

积压队列中的每个字节，都有一个全局性的偏移量。主节点维持一个计数器作为复制偏移量，当主节点回复从节点”+FULLRESYNC  <runid>  <offset>”信息时，其中的offset就是当前主节点的复制偏移量的值。当从节点收到该消息后，保存<runid>，取出<offset>作为自己的复制偏移量的初始值。

当主节点收到客户端发来的，长度为len的命令请求之后，就会将len增加到复制偏移量上。然后将该命令请求追加到积压队列中，并且发给每个从节点。从节点收到主节点发来的命令之后，同样会将命令长度len增加到自己的复制偏移量上，这就保证了主从节点上复制偏移量的一致性，也就是数据库状态的一致性。

积压队列是一个空间有限的循环队列，随着命令的追加，不断覆盖之前的命令，积压队列中累积的命令偏移量范围也在不断发生变化。

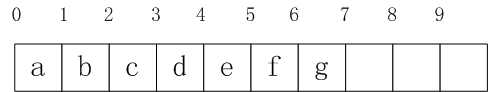
当从节点断链一段时间，然后重连主节点时，向主节点发来”PSYNC <runid> <offset>”命令。其中的<runid>就是断链前保存的主节点运行ID，<offset>就是自己的复制偏移量加1，表示需要接收的下一条命令首字节的偏移量。

主节点收到该”PSYNC”消息后，首先判断<runid>是否与自己的运行ID匹配，如果不匹配，则不能执行部分重同步；然后判断偏移量<offset>是否还在积压队列中累积的命令范围内，如果在，则说明可以进行部分重同步。

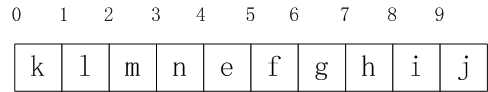
要理解部分重同步，必须理解积压队列的实现。

**二：积压队列的实现**

Redis中的积压队列server.repl\_backlog，是一个固定大小的循环队列。所谓循环队列，举个简单的例子，假设server.repl\_backlog的大小为10个字节，则向其中插入数据”abcdefg”之后，该积压队列的内容如下：



现在插入数据”hijklmn”，则积压队列的内容如下：



也就是说，插入数据时，一旦到达了积压队列的尾部，则重新从头部开始插入，覆盖最早插入的内容。

要理解积压队列，关键在于理解下面的，有关积压队列的属性：

server.master\_repl\_offset：一个全局性的计数器。该属性只有存在积压队列的情况下才会增加计数。当存在积压队列时，每次收到客户端发来的，长度为len的请求命令时，就会将server.master\_repl\_offset增加len。

该属性也就是所谓的主节点上的复制偏移量。当从节点发来PSYNC命令后，主节点回复从节点"+FULLRESYNC <runid> <offset>"消息时，其中的offset就是取的主节点当时的server.master\_repl\_offset的值。这样当从节点收到该消息后，将该值保存在复制偏移量server.master->reploff中。

进入命令传播阶段后，每当主节点收到客户端的命令请求，则将命令的长度增加到server.master\_repl\_offset上，然后将命令传播给从节点，从节点收到后，也会将命令长度加到server.master->reploff上，从而保证了主节点上的复制偏移量server.master\_repl\_offset和从节点上的复制偏移量server.master->reploff的一致性。

需要注意的，server.master\_repl\_offset的值并不是严格的从0开始增加的。它只是一个计数器，只要能保证主从节点上的复制偏移量一致即可。比如如果它的初始值为10，发送给从节点后，从节点保存的复制偏移量初始值也为10，当新的命令来临时，主从节点上的复制偏移量都会相应增加该命令的长度，因此这并不影响主从节点上偏移量的一致性。

server.repl\_backlog\_size：积压队列server.repl\_backlog的总容量。

server.repl\_backlog\_idx：在积压队列server.repl\_backlog中，每次写入新数据时的起始索引，是一个相对于server.repl\_backlog的索引。当server.repl\_backlog\_idx 等于server.repl\_backlog的长度server.repl\_backlog\_size时，置其值为0，表示从头开始。

以上面那个积压队列为例，server.repl\_backlog\_idx的初始值为0，插入”abcdefg”之后，该值变为7；插入”hijklmn”之后，该值变为4。

server.repl\_backlog\_histlen：积压队列server.repl\_backlog中，当前累积的数据量的大小。该值不会超过积压队列的总容量server.repl\_backlog\_size。

server.repl\_backlog\_off：在积压队列中，最早保存的命令的首字节，在全局范围内（而非积压队列内）的偏移量。在累积命令流时，下列等式恒成立：

server.master\_repl\_offset - server.repl\_backlog\_off + 1 = server.repl\_backlog\_histlen。

还是以上面那个积压队列为例：如果在插入”abcdefg”之前，server.master\_repl\_offset的初始值为2，则插入”abcdefg”之后，积压队列中当前的数据量，也就是属性server.repl\_backlog\_histlen的值为7。属性server.master\_repl\_offset的值变为9，此时命令的首字节为”a”，它在全局的偏移量就是3。满足上面的等式。

在插入”hijklmn”之后，积压队列中当前的数据量，也就是属性server.repl\_backlog\_histlen的值为10。属性server.master\_repl\_offset的值变为16。此时最早保存的命令首字节为”e”，它在全局的偏移量是7，满足上面的等式。

根据上面的等式，主节点的积压队列中累积的命令流，首字节和尾字节在全局范围内的偏移量分别是server.repl\_backlog\_off和server.master\_repl\_offset。

当从节点断链重连后，向主节点发送”PSYNC <runid> <offset>”消息，其中的<offset>表示需要接收的下一条命令首字节的偏移量。也就是server.master->reploff + 1。

主节点判断<offset>的值，如果该值在下面的范围内，就表示可以进行部分重同步：

[server.repl\_backlog\_off, server.repl\_backlog\_off + server.repl\_backlog\_histlen]。如果<offset>的值为server.repl\_backlog\_off + server.repl\_backlog\_histlen，也就是server.master\_repl\_offset + 1，说明从节点断链期间，主节点没有收到过新的命令请求。

**三：部分重同步**

**1：**masterTryPartialResynchronization函数

主节点收到从节点的”PSYNC <runid> <offset>”消息后，调用函数masterTryPartialResynchronization尝试进行部分重同步。该函数的代码如下：

int masterTryPartialResynchronization**(**redisClient **\***c**)** **{**

long long psync\_offset**,** psync\_len**;**

char **\***master\_runid **=** c**->**argv**[**1**]->**ptr**;**

char buf**[**128**];**

int buflen**;**

*/\* Is the runid of this master the same advertised by the wannabe slave*

*\* via PSYNC? If runid changed this master is a different instance and*

*\* there is no way to continue. \*/*

**if** **(**strcasecmp**(**master\_runid**,** server**.**runid**))** **{**

*/\* Run id "?" is used by slaves that want to force a full resync. \*/*

**if** **(**master\_runid**[**0**]** **!=** '?'**)** **{**

redisLog**(**REDIS\_NOTICE**,**"Partial resynchronization not accepted: "

"Runid mismatch (Client asked for runid '%s', my runid is '%s')"**,**

master\_runid**,** server**.**runid**);**

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

redisLog**(**REDIS\_NOTICE**,**"Full resync requested by slave %s"**,**

replicationGetSlaveName**(**c**));**

**}**

**goto** need\_full\_resync**;**

**}**

*/\* We still have the data our slave is asking for? \*/*

**if** **(**getLongLongFromObjectOrReply**(**c**,**c**->**argv**[**2**],&**psync\_offset**,NULL)** **!=**

REDIS\_OK**)** **goto** need\_full\_resync**;**

**if** **(!**server**.**repl\_backlog **||**

psync\_offset **<** server**.**repl\_backlog\_off **||**

psync\_offset **>** **(**server**.**repl\_backlog\_off **+** server**.**repl\_backlog\_histlen**))**

**{**

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

"Unable to partial resync with slave %s for lack of backlog (Slave request was: %lld)."**,** replicationGetSlaveName**(**c**),** psync\_offset**);**

**if** **(**psync\_offset **>** server**.**master\_repl\_offset**)** **{**

redisLog**(**REDIS\_WARNING**,**

"Warning: slave %s tried to PSYNC with an offset that is greater than the master replication offset."**,** replicationGetSlaveName**(**c**));**

**}**

**goto** need\_full\_resync**;**

**}**

*/\* If we reached this point, we are able to perform a partial resync:*

*\* 1) Set client state to make it a slave.*

*\* 2) Inform the client we can continue with +CONTINUE*

*\* 3) Send the backlog data (from the offset to the end) to the slave. \*/*

c**->**flags **|=** REDIS\_SLAVE**;**

c**->**replstate **=** REDIS\_REPL\_ONLINE**;**

c**->**repl\_ack\_time **=** server**.**unixtime**;**

c**->**repl\_put\_online\_on\_ack **=** 0**;**

listAddNodeTail**(**server**.**slaves**,**c**);**

*/\* We can't use the connection buffers since they are used to accumulate*

*\* new commands at this stage. But we are sure the socket send buffer is*

*\* empty so this write will never fail actually. \*/*

buflen **=** snprintf**(**buf**,sizeof(**buf**),**"+CONTINUE\r\n"**);**

**if** **(**write**(**c**->**fd**,**buf**,**buflen**)** **!=** buflen**)** **{**

freeClientAsync**(**c**);**

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

**}**

psync\_len **=** addReplyReplicationBacklog**(**c**,**psync\_offset**);**

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

"Partial resynchronization request from %s accepted. Sending %lld bytes of backlog starting from offset %lld."**,**

replicationGetSlaveName**(**c**),**

psync\_len**,** psync\_offset**);**

*/\* Note that we don't need to set the selected DB at server.slaveseldb*

*\* to -1 to force the master to emit SELECT, since the slave already*

*\* has this state from the previous connection with the master. \*/*

refreshGoodSlavesCount**();**

**return** REDIS\_OK**;** */\* The caller can return, no full resync needed. \*/*

need\_full\_resync**:**

*/\* We need a full resync for some reason... Note that we can't*

*\* reply to PSYNC right now if a full SYNC is needed. The reply*

*\* must include the master offset at the time the RDB file we transfer*

*\* is generated, so we need to delay the reply to that moment. \*/*

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

**}**

该函数返回REDIS\_ERR表示不能进行部分重同步；返回REDIS\_OK表示可以进行部分重同步。

首先比对"PSYNC"命令参数中的运行ID和本身的ID号是否匹配，如果不匹配，则需要进行完全重同步，因此直接返回REDIS\_ERR即可；

然后取出"PSYNC"命令参数中的从节点复制偏移到psync\_offset中，该值表示从节点需要接收的下一条命令首字节的偏移量。接下来根据积压队列的状态判断是否可以进行部分重同步，判断的条件上一节中已经讲过了，不再赘述。

经过上面的检查后，说明可以进行部分重同步了。因此：首先将REDIS\_SLAVE标记增加到客户端标志位中；然后将从节点客户端的复制状态置为REDIS\_REPL\_ONLINE，并且将c->repl\_put\_online\_on\_ack置为0。这点很重要，因为只有当c->replstate为REDIS\_REPL\_ONLINE，并且c->repl\_put\_online\_on\_ack为0时，在函数prepareClientToWrite中，才为socket描述符注册可写事件，这样才能将输出缓存中的内容发送给从节点客户端；

接下来，直接向客户端的socket描述符上输出"+CONTINUE\r\n"命令，这里不能用输出缓存，因为输出缓存只能用于累积命令流。之前主节点向从节点发送的信息很少，因此内核的输出缓存中应该会有空间，因此这里直接的write操作一般不会出错；

接下来，调用addReplyReplicationBacklog，将积压队列中psync\_offset之后的数据复制到客户端输出缓存中，注意这里不需要设置server.slaveseldb为-1，因为从节点是接着上次连接进行的；

最后，调用refreshGoodSlavesCount，更新当前状态正常的从节点数量；

2：addReplyReplicationBacklog函数

主节点确认可以为从节点进行部分重同步时，首先就是调用addReplyReplicationBacklog函数，将积压队列中，全局偏移量为offset的字节，到尾字节之间的所有内容，追加到从节点客户端的输出缓存中。该函数的代码如下：

long long addReplyReplicationBacklog**(**redisClient **\***c**,** long long offset**)** **{**

long long j**,** skip**,** len**;**

redisLog**(**REDIS\_DEBUG**,** "[PSYNC] Slave request offset: %lld"**,** offset**);**

**if** **(**server**.**repl\_backlog\_histlen **==** 0**)** **{**

redisLog**(**REDIS\_DEBUG**,** "[PSYNC] Backlog history len is zero"**);**

**return** 0**;**

**}**

redisLog**(**REDIS\_DEBUG**,** "[PSYNC] Backlog size: %lld"**,**

server**.**repl\_backlog\_size**);**

redisLog**(**REDIS\_DEBUG**,** "[PSYNC] First byte: %lld"**,**

server**.**repl\_backlog\_off**);**

redisLog**(**REDIS\_DEBUG**,** "[PSYNC] History len: %lld"**,**

server**.**repl\_backlog\_histlen**);**

redisLog**(**REDIS\_DEBUG**,** "[PSYNC] Current index: %lld"**,**

server**.**repl\_backlog\_idx**);**

*/\* Compute the amount of bytes we need to discard. \*/*

skip **=** offset **-** server**.**repl\_backlog\_off**;**

redisLog**(**REDIS\_DEBUG**,** "[PSYNC] Skipping: %lld"**,** skip**);**

*/\* Point j to the oldest byte, that is actaully our*

*\* server.repl\_backlog\_off byte. \*/*

j **=** **(**server**.**repl\_backlog\_idx **+**

**(**server**.**repl\_backlog\_size**-**server**.**repl\_backlog\_histlen**))** **%**

server**.**repl\_backlog\_size**;**

redisLog**(**REDIS\_DEBUG**,** "[PSYNC] Index of first byte: %lld"**,** j**);**

*/\* Discard the amount of data to seek to the specified 'offset'. \*/*

j **=** **(**j **+** skip**)** **%** server**.**repl\_backlog\_size**;**

*/\* Feed slave with data. Since it is a circular buffer we have to*

*\* split the reply in two parts if we are cross-boundary. \*/*

len **=** server**.**repl\_backlog\_histlen **-** skip**;**

redisLog**(**REDIS\_DEBUG**,** "[PSYNC] Reply total length: %lld"**,** len**);**

**while(**len**)** **{**

long long thislen **=**

**((**server**.**repl\_backlog\_size **-** j**)** **<** len**)** **?**

**(**server**.**repl\_backlog\_size **-** j**)** **:** len**;**

redisLog**(**REDIS\_DEBUG**,** "[PSYNC] addReply() length: %lld"**,** thislen**);**

addReplySds**(**c**,**sdsnewlen**(**server**.**repl\_backlog **+** j**,** thislen**));**

len **-=** thislen**;**

j **=** 0**;**

**}**

**return** server**.**repl\_backlog\_histlen **-** skip**;**

**}**

在该函数中，首先计算需要在积压队列中跳过的字节数skip，offset为从节点所需数据的首字节的全局偏移量，server.repl\_backlog\_off表示积压队列中最早累积的命令首字节的全局偏移量，因此skip等于offset - server.repl\_backlog\_off；

接下来，计算积压队列中，最早累积的命令首字节，在积压队列中的索引j，server.repl\_backlog\_idx-1表示积压队列中，命令尾字节在积压队列中的索引，server.repl\_backlog\_size表示积压队列的总容量，server.repl\_backlog\_histlen表示积压队列中累积的命令的大小，因此得到j的值为： (server.repl\_backlog\_idx+(server.repl\_backlog\_size-server.repl\_backlog\_histlen))%server.repl\_backlog\_size;

接下来，将j置为需要数据首字节相对于积压队列中的索引；然后计算总共需要复制的字节数len；然后就是将数据循环追加到从节点客户端的输出缓存中（追加之前，已经在函数syncCommand保证该输出缓存为空）；

3：feedReplicationBacklog函数

主节点收到客户端发来的命令请求后，除了需要将命令累积到从节点的输出缓存中，还需要将该命令追加到积压队列中。feedReplicationBacklog函数就是用于实现将命令追加到积压队列中的函数。

它的代码如下：

void feedReplicationBacklog**(**void **\***ptr**,** size\_t len**)** **{**

unsigned char **\***p **=** ptr**;**

server**.**master\_repl\_offset **+=** len**;**

*/\* This is a circular buffer, so write as much data we can at every*

*\* iteration and rewind the "idx" index if we reach the limit. \*/*

**while(**len**)** **{**

size\_t thislen **=** server**.**repl\_backlog\_size **-** server**.**repl\_backlog\_idx**;**

**if** **(**thislen **>** len**)** thislen **=** len**;**

memcpy**(**server**.**repl\_backlog**+**server**.**repl\_backlog\_idx**,**p**,**thislen**);**

server**.**repl\_backlog\_idx **+=** thislen**;**

**if** **(**server**.**repl\_backlog\_idx **==** server**.**repl\_backlog\_size**)**

server**.**repl\_backlog\_idx **=** 0**;**

len **-=** thislen**;**

p **+=** thislen**;**

server**.**repl\_backlog\_histlen **+=** thislen**;**

**}**

**if** **(**server**.**repl\_backlog\_histlen **>** server**.**repl\_backlog\_size**)**

server**.**repl\_backlog\_histlen **=** server**.**repl\_backlog\_size**;**

*/\* Set the offset of the first byte we have in the backlog. \*/*

server**.**repl\_backlog\_off **=** server**.**master\_repl\_offset **-**

server**.**repl\_backlog\_histlen **+** 1**;**

**}**

函数中，首先将len增加到主节点复制偏移量server.master\_repl\_offset中；

然后进入循环，将ptr追加到积压队列中，在循环中：

首先计算本次追加的数据量thislen。server.repl\_backlog\_size表示积压队列的总容量，server.repl\_backlog\_idx-1表示积压队列中，累积的命令尾字节在积压队列中的索引，因此thislen等于server.repl\_backlog\_size-server.repl\_backlog\_idx，表示在积压队列的尾部之前，还可以追加多少字节。如果thislen大于len，则调整其值；

然后将p中的thislen个字节，复制到首地址为server.repl\_backlog+server.repl\_backlog\_idx的内存中；

接下来更新server.repl\_backlog\_idx的值，如果其值等于积压队列的总容量，表示已经到达积压队列的尾部，因此下一次添加数据时，需要重新从头部开始，因此置server.repl\_backlog\_idx为0；

然后更新len和p；

最后更新server.repl\_backlog\_histlen的值；该值表示积压队列中累积的命令总量；

server.repl\_backlog\_histlen的值最大不能超过积压队列的总容量，因此将所有数据追加到积压队列后，如果其值已经大于总容量server.repl\_backlog\_size，则重新置其值为server.repl\_backlog\_size；

最后，更新server.repl\_backlog\_off的值，使其满足等式：

server.repl\_backlog\_histlen=server.master\_repl\_offset-server.repl\_backlog\_off+1;

**四：定时监测函数replicationCron**

主从节点为了探测网络是连通的，每隔一段时间，都会向对方发送一定的心跳信息。

之前在《[Resis主从复制之从节点流程](http://blog.csdn.net/gqtcgq/article/details/51172085)》介绍过，从节点在接受完RDB数据之后，清空本身数据库时，以及加载RDB数据时，都会时不时的向主节点发送一个换行符”\n”（通过回调函数replicationSendNewlineToMaster实现）；而且，当从节点本身的复制状态变为REDIS\_REPL\_CONNECTED之后，每隔1秒钟就会向主节点发送一个"REPLCONF ACK <offset>"命令。以上的”\n”和"REPLCONF”命令都是从节点向主节点发送的心跳消息。

主节点每隔一段时间，也会向从节点发送”PING”命令，以及换行符”\n”。这是主节点向从节点发送的心跳消息。

主从节点收到对方发来的消息后，都会更新一个时间戳。双方都会定时检查各自时间戳的最后更新时间。这样，当主从节点间长时间没有交互时，说明网络出现了问题，主从双方都可以探测到该问题，从而断开连接；

以上这些探测功能就是在定时执行的函数replicationCron中实现的，该函数每隔1秒钟调用一次。该函数的代码如下：

void replicationCron**(**void**)** **{**

static long long replication\_cron\_loops **=** 0**;**

*/\* Non blocking connection timeout? \*/*

**if** **(**server**.**masterhost **&&**

**(**server**.**repl\_state **==** REDIS\_REPL\_CONNECTING **||**

slaveIsInHandshakeState**())** **&&**

**(**time**(NULL)-**server**.**repl\_transfer\_lastio**)** **>** server**.**repl\_timeout**)**

**{**

redisLog**(**REDIS\_WARNING**,**"Timeout connecting to the MASTER..."**);**

undoConnectWithMaster**();**

**}**

*/\* Bulk transfer I/O timeout? \*/*

**if** **(**server**.**masterhost **&&** server**.**repl\_state **==** REDIS\_REPL\_TRANSFER **&&**

**(**time**(NULL)-**server**.**repl\_transfer\_lastio**)** **>** server**.**repl\_timeout**)**

**{**

redisLog**(**REDIS\_WARNING**,**"Timeout receiving bulk data from MASTER... If the problem persists try to set the 'repl-timeout' parameter in redis.conf to a larger value."**);**

replicationAbortSyncTransfer**();**

**}**

*/\* Timed out master when we are an already connected slave? \*/*

**if** **(**server**.**masterhost **&&** server**.**repl\_state **==** REDIS\_REPL\_CONNECTED **&&**

**(**time**(NULL)-**server**.**master**->**lastinteraction**)** **>** server**.**repl\_timeout**)**

**{**

redisLog**(**REDIS\_WARNING**,**"MASTER timeout: no data nor PING received..."**);**

freeClient**(**server**.**master**);**

**}**

*/\* Check if we should connect to a MASTER \*/*

**if** **(**server**.**repl\_state **==** REDIS\_REPL\_CONNECT**)** **{**

redisLog**(**REDIS\_NOTICE**,**"Connecting to MASTER %s:%d"**,**

server**.**masterhost**,** server**.**masterport**);**

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

redisLog**(**REDIS\_NOTICE**,**"MASTER <-> SLAVE sync started"**);**

**}**

**}**

*/\* Send ACK to master from time to time.*

*\* Note that we do not send periodic acks to masters that don't*

*\* support PSYNC and replication offsets. \*/*

**if** **(**server**.**masterhost **&&** server**.**master **&&**

**!(**server**.**master**->**flags **&** REDIS\_PRE\_PSYNC**))**

replicationSendAck**();**

*/\* If we have attached slaves, PING them from time to time.*

*\* So slaves can implement an explicit timeout to masters, and will*

*\* be able to detect a link disconnection even if the TCP connection*

*\* will not actually go down. \*/*

listIter li**;**

listNode **\***ln**;**

robj **\***ping\_argv**[**1**];**

*/\* First, send PING according to ping\_slave\_period. \*/*

**if** **((**replication\_cron\_loops **%** server**.**repl\_ping\_slave\_period**)** **==** 0**)** **{**

ping\_argv**[**0**]** **=** createStringObject**(**"PING"**,**4**);**

replicationFeedSlaves**(**server**.**slaves**,** server**.**slaveseldb**,**

ping\_argv**,** 1**);**

decrRefCount**(**ping\_argv**[**0**]);**

**}**

*/\* Second, send a newline to all the slaves in pre-synchronization*

*\* stage, that is, slaves waiting for the master to create the RDB file.*

*\* The newline will be ignored by the slave but will refresh the*

*\* last-io timer preventing a timeout. In this case we ignore the*

*\* ping period and refresh the connection once per second since certain*

*\* timeouts are set at a few seconds (example: PSYNC response). \*/*

listRewind**(**server**.**slaves**,&**li**);**

**while((**ln **=** listNext**(&**li**)))** **{**

redisClient **\***slave **=** ln**->**value**;**

**if** **(**slave**->**replstate **==** REDIS\_REPL\_WAIT\_BGSAVE\_START **||**

**(**slave**->**replstate **==** REDIS\_REPL\_WAIT\_BGSAVE\_END **&&**

server**.**rdb\_child\_type **!=** REDIS\_RDB\_CHILD\_TYPE\_SOCKET**))**

**{**

**if** **(**write**(**slave**->**fd**,** "\n"**,** 1**)** **==** **-**1**)** **{**

*/\* Don't worry, it's just a ping. \*/*

**}**

**}**

**}**

*/\* Disconnect timedout slaves. \*/*

**if** **(**listLength**(**server**.**slaves**))** **{**

listIter li**;**

listNode **\***ln**;**

listRewind**(**server**.**slaves**,&**li**);**

**while((**ln **=** listNext**(&**li**)))** **{**

redisClient **\***slave **=** ln**->**value**;**

**if** **(**slave**->**replstate **!=** REDIS\_REPL\_ONLINE**)** **continue;**

**if** **(**slave**->**flags **&** REDIS\_PRE\_PSYNC**)** **continue;**

**if** **((**server**.**unixtime **-** slave**->**repl\_ack\_time**)** **>** server**.**repl\_timeout**)**

**{**

redisLog**(**REDIS\_WARNING**,** "Disconnecting timedout slave: %s"**,**

replicationGetSlaveName**(**slave**));**

freeClient**(**slave**);**

**}**

**}**

**}**

*/\* If we have no attached slaves and there is a replication backlog*

*\* using memory, free it after some (configured) time. \*/*

**if** **(**listLength**(**server**.**slaves**)** **==** 0 **&&** server**.**repl\_backlog\_time\_limit **&&**

server**.**repl\_backlog**)**

**{**

time\_t idle **=** server**.**unixtime **-** server**.**repl\_no\_slaves\_since**;**

**if** **(**idle **>** server**.**repl\_backlog\_time\_limit**)** **{**

freeReplicationBacklog**();**

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

"Replication backlog freed after %d seconds "

"without connected slaves."**,**

**(**int**)** server**.**repl\_backlog\_time\_limit**);**

**}**

**}**

*/\* If AOF is disabled and we no longer have attached slaves, we can*

*\* free our Replication Script Cache as there is no need to propagate*

*\* EVALSHA at all. \*/*

**if** **(**listLength**(**server**.**slaves**)** **==** 0 **&&**

server**.**aof\_state **==** REDIS\_AOF\_OFF **&&**

listLength**(**server**.**repl\_scriptcache\_fifo**)** **!=** 0**)**

**{**

replicationScriptCacheFlush**();**

**}**

*/\* If we are using diskless replication and there are slaves waiting*

*\* in WAIT\_BGSAVE\_START state, check if enough seconds elapsed and*

*\* start a BGSAVE.*

*\**

*\* This code is also useful to trigger a BGSAVE if the diskless*

*\* replication was turned off with CONFIG SET, while there were already*

*\* slaves in WAIT\_BGSAVE\_START state. \*/*

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

time\_t idle**,** max\_idle **=** 0**;**

int slaves\_waiting **=** 0**;**

int mincapa **=** **-**1**;**

listNode **\***ln**;**

listIter li**;**

listRewind**(**server**.**slaves**,&**li**);**

**while((**ln **=** listNext**(&**li**)))** **{**

redisClient **\***slave **=** ln**->**value**;**

**if** **(**slave**->**replstate **==** REDIS\_REPL\_WAIT\_BGSAVE\_START**)** **{**

idle **=** server**.**unixtime **-** slave**->**lastinteraction**;**

**if** **(**idle **>** max\_idle**)** max\_idle **=** idle**;**

slaves\_waiting**++;**

mincapa **=** **(**mincapa **==** **-**1**)** **?** slave**->**slave\_capa **:**

**(**mincapa **&** slave**->**slave\_capa**);**

**}**

**}**

**if** **(**slaves\_waiting **&&** max\_idle **>** server**.**repl\_diskless\_sync\_delay**)** **{**

*/\* Start a BGSAVE. Usually with socket target, or with disk target*

*\* if there was a recent socket -> disk config change. \*/*

startBgsaveForReplication**(**mincapa**);**

**}**

**}**

*/\* Refresh the number of slaves with lag <= min-slaves-max-lag. \*/*

refreshGoodSlavesCount**();**

replication\_cron\_loops**++;** */\* Incremented with frequency 1 HZ. \*/*

**}**

server.repl\_timeout属性是用户在配置文件中配置的"repl-timeout"选项的值，表示主从复制期间最大的超时时间，默认为60秒；

从从节点向主节点建链开始，到读取完主节点发来的RDB数据为止，也就是复制状态从REDIS\_REPL\_CONNECTING到REDIS\_REPL\_TRANSFER期间，每当从节点读取到主节点发来的信息后，都会更新server.repl\_transfer\_lastio属性为当时的Unix时间戳；

当从节点处于REDIS\_REPL\_CONNECTING状态或者握手状态时，并且最后一次更新server.repl\_transfer\_lastio的时间已经超过了最大超时时间，则调用函数undoConnectWithMaster，断开与主节点间的连接；

当从节点处于REDIS\_REPL\_TRANSFER状态（接收RDB数据），并且最后一次更新server.repl\_transfer\_lastio的时间已经超过了最大超时时间，则调用函数replicationAbortSyncTransfer，终止本次复制过程；

在读取客户端发来的消息的函数readQueryFromClient中，每次从socket描述符上读取到数据后，就会更新客户端结构中的lastinteraction属性。

因此，当从节点处于REDIS\_REPL\_CONNECTED状态时（命令传播阶段），如果最后一次更新server.master->lastinteractio的时间已经超过了最大超时时间，则调用函数freeClient，断开与主节点间的连接；

以上就是从节点探测网络是否连通的方法；

如果当前从节点的复制状态为REDIS\_REPL\_CONNECT，则调用connectWithMaster开始向主节点发起建链请求。从节点收到客户端发来的”SLAVEOF”命令，或从节点实例启动，从配置文件中读取到了"slaveof"选项后，就将复制状态置为REDIS\_REPL\_CONNECT，而在此处开始向主节点发起TCP建链；

如果当前从节点的server.master属性已配置好，说明该从节点已处于REDIS\_REPL\_CONNECTED状态，并且主节点支持PSYNC命令的情况下，调用函数replicationSendAck向主节点发送"REPLCONF ACK <offset>"消息，这就是从节点向主节点发送心跳消息；

主节点每隔一定时间也会向从节点发送心跳消息，以使从节点可以更新属性server.repl\_transfer\_lastio的值。

首先是每隔server.repl\_ping\_slave\_period秒，向从节点输出缓存以及积压队列中追加"PING"命令；

然后就是轮训列表server.slaves，对于处于REDIS\_REPL\_WAIT\_BGSAVE\_START状态的从节点，或者处于REDIS\_REPL\_WAIT\_BGSAVE\_END状态的从节点，且当目前是无硬盘复制的RDB转储时，直接调用write向从节点发送一个换行符；

当主节点将从节点的复制状态置为REDIS\_REPL\_ONLINE后，每当收到从节点发来的换行符"\n"（从节点加载RDB数据时发送）或者"REPLCONF ACK <offset>"信息时，就会更新该从节点客户端的repl\_ack\_time属性。

因此，主节点轮训server.slaves列表，如果其中的某个从节点的repl\_ack\_time属性的最近一次的更新时间，已经超过了最大超时时间，则调用函数freeClient，断开与从节点间的连接；

以上就是主节点探测网络是否连通的方法；

在freeClient函数中，每当释放了一个从节点客户端后，都会判断列表server.slaves当前长度，如果其长度为0，说明该主节点已经没有连接的从节点了，因此就会设置属性server.repl\_no\_slaves\_since为当时的时间戳；

server.repl\_backlog\_time\_limit属性值表示当主节点没有从节点连接时，积压队列最长的存活时间，该值默认为1个小时。

因此，如果主节点当前已没有从节点连接，并且配置了server.repl\_backlog\_time\_limit属性值，并且积压队列还存在的情况下，则判断属性server.repl\_no\_slaves\_since最近一次更新时间是否已经超过配置的server.repl\_backlog\_time\_limit属性值，若已超过，则调用freeReplicationBacklog释放积压队列；

如果主节点当前已没有从节点连接，并且Redis实例关闭了AOF功能，并且列表server.repl\_scriptcache\_fifo的长度非0，则调用函数replicationScriptCacheFlush；

之前在函数syncCommand中介绍过，如果当前没有进行RDB数据转储，则当支持无硬盘复制的RDB数据的从节点的"PSYNC"命令到来时，并非立即启动BGSAVE操作，而是等待一段时间再开始。这是因为无硬盘复制的RDB数据无法复用，Redis通过这种方式来等待更多的从节点到来，从而减少执行BGSAVE操作的次数；

配置文件中"repl-diskless-sync-delay"选项的值，记录在server.repl\_diskless\_sync\_delay中，该值就是主节点等待的最大时间。

因此，轮训列表server.slaves，针对其中处于REDIS\_REPL\_WAIT\_BGSAVE\_START状态的从节点，得到这些从节点的空闲时间的最大值max\_idle，以及能力的最小值mincapa；

轮训完之后，如果max\_idle大于选项server.repl\_diskless\_sync\_delay的值，则以参数mincapa调用函数startBgsaveForReplication，开始BGSAVE操作；

最后，调用refreshGoodSlavesCount，更新当前状态正常的从节点数量。

**五：min-slaves选项**

Redis主节点可以配置"min-slaves-to-write"和"min-slaves-max-lag"两个选项用于防止主节点在不安全的情况下执行写命令。

这两个选项的意义在于：如果从节点与主节点的最后交互时间，距离当前时间小于"min-slaves-max-lag"的值，则认为该从节点状态是连接的。主节点定时计算当前状态为连接的从节点数目，如果该数目小于"min-slaves-to-write"的值，则主节点拒绝执行写数据库的命令。

计算当前状态为连接的从节点数目，是通过函数refreshGoodSlavesCount实现的。该函数会在定时函数replicationCron中调用，也就是每隔1秒就会调用一次。该函数的代码如下：

void refreshGoodSlavesCount**(**void**)** **{**

listIter li**;**

listNode **\***ln**;**

int good **=** 0**;**

**if** **(!**server**.**repl\_min\_slaves\_to\_write **||**

**!**server**.**repl\_min\_slaves\_max\_lag**)** **return;**

listRewind**(**server**.**slaves**,&**li**);**

**while((**ln **=** listNext**(&**li**)))** **{**

redisClient **\***slave **=** ln**->**value**;**

time\_t lag **=** server**.**unixtime **-** slave**->**repl\_ack\_time**;**

**if** **(**slave**->**replstate **==** REDIS\_REPL\_ONLINE **&&**

lag **<=** server**.**repl\_min\_slaves\_max\_lag**)** good**++;**

**}**

server**.**repl\_good\_slaves\_count **=** good**;**

**}**

从节点的复制状态为REDIS\_REPL\_ONLINE之后，主节点收到从节点发来的”REPLCONF ACK <offset>”命令时，就会更新该从节点客户端repl\_ack\_time属性，以此属性判断从节点与主节点的最后交互时间。

该函数中，如果没有配置server.repl\_min\_slaves\_to\_write或者server.repl\_min\_slaves\_max\_lag，则直接返回；

然后轮训列表server.slaves，针对其中的每个从节点客户端，得到其slave->repl\_ack\_time属性与当前时间的差值，如果该差值小于等于server.repl\_min\_slaves\_max\_lag的值，则说明该从节点状态良好，计数器加1。

最后将状态良好的从节点数目更新到server.repl\_good\_slaves\_count中。

在处理客户端命令的函数processCommand中，有下面的代码：

/\* Don't accept write commands if there are not enough good slaves and

\* user configured the min-slaves-to-write option. \*/

**if** **(**server**.**masterhost **==** **NULL** **&&**

server**.**repl\_min\_slaves\_to\_write **&&**

server**.**repl\_min\_slaves\_max\_lag **&&**

c**->**cmd**->**flags **&** REDIS\_CMD\_WRITE **&&**

server**.**repl\_good\_slaves\_count **<** server**.**repl\_min\_slaves\_to\_write**)**

**{**

flagTransaction**(**c**);**

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

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

**}**

因此，只要当前要执行的是写数据库命令，而且server.repl\_good\_slaves\_count的值小于server.repl\_min\_slaves\_to\_write的值，则会回复客户端错误信息，并直接返回而不再处理。

**六：WAIT命令**

WAIT命令是自Redis3.0.0版本开始引入的。客户端发送”WAIT <numslaves>  <timeout>”命令后，会被阻塞。直到以下的两个条件之一发生：

a：在WAIT命令之前的写数据库命令，都已经发给从库，并且至少<numslaves>个从库确认收到了；

b：超时时间 <timeout>（毫秒）到时；

WAIT命令返回时，不管是正常返回，还是超时返回，返回的结果都是已经确认收到WAIT之前的写命令的从节点个数。

注意，如果WATI命令在MULTI事务中执行的，那该命令会立即返回已经确认的从节点个数。

如果timeout置为0，则表示永久等待；

主节点收到客户端发来的WAIT命令后，调用waitCommand函数处理。该函数的代码如下：

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

mstime\_t timeout**;**

long numreplicas**,** ackreplicas**;**

long long offset **=** c**->**woff**;**

*/\* Argument parsing. \*/*

**if** **(**getLongFromObjectOrReply**(**c**,**c**->**argv**[**1**],&**numreplicas**,NULL)** **!=** REDIS\_OK**)**

**return;**

**if** **(**getTimeoutFromObjectOrReply**(**c**,**c**->**argv**[**2**],&**timeout**,**UNIT\_MILLISECONDS**)**

**!=** REDIS\_OK**)** **return;**

*/\* First try without blocking at all. \*/*

ackreplicas **=** replicationCountAcksByOffset**(**c**->**woff**);**

**if** **(**ackreplicas **>=** numreplicas **||** c**->**flags **&** REDIS\_MULTI**)** **{**

addReplyLongLong**(**c**,**ackreplicas**);**

**return;**

**}**

*/\* Otherwise block the client and put it into our list of clients*

*\* waiting for ack from slaves. \*/*

c**->**bpop**.**timeout **=** timeout**;**

c**->**bpop**.**reploffset **=** offset**;**

c**->**bpop**.**numreplicas **=** numreplicas**;**

listAddNodeTail**(**server**.**clients\_waiting\_acks**,**c**);**

blockClient**(**c**,**REDIS\_BLOCKED\_WAIT**);**

*/\* Make sure that the server will send an ACK request to all the slaves*

*\* before returning to the event loop. \*/*

replicationRequestAckFromSlaves**();**

**}**

每当客户端的命令被处理后（在processCommand中，调用call函数之后），都会更新c->woff的值为当时的复制偏移量server.master\_repl\_offset，因此，只要从节点客户端的slave->repl\_ack\_off属性大于该值，就说明该从节点已经确认了WAIT之前的写命令；

函数中，首先从命令参数中取出numreplicas和timeout；注意，命令参数中的<timeout>是个相对毫秒值，比如3000等；而这里取出的timeout，会被转换为绝对时间戳；

接着调用replicationCountAcksByOffset函数，得到当前已经发送来确认的从节点个数ackreplicas；如果ackreplicas大于等于numreplicas，或者当前客户端正在执行MULTI事务处理，则立即返回给客户端ackreplicas信息，并返回；

其他情况下，将WAIT命令参数，以及offset记录到c->bpop中。然后将该客户端追加到列表server.clients\_waiting\_acks中；并调用函数blockClient，将客户端标志位阻塞的；

最后，调用replicationRequestAckFromSlaves，置标志位server.get\_ack\_from\_slaves为1：

c**->**bpop**.**timeout **=** timeout**;**

c**->**bpop**.**reploffset **=** offset**;**

c**->**bpop**.**numreplicas **=** numreplicas**;**

listAddNodeTail**(**server**.**clients\_waiting\_acks**,**c**);**

blockClient**(**c**,**REDIS\_BLOCKED\_WAIT**);**

*/\* Make sure that the server will send an ACK request to all the slaves*

*\* before returning to the event loop. \*/*

replicationRequestAckFromSlaves**();**

1：超时检查

在定时执行的函数clientsCronHandleTimeout中，会检查客户端的c->bpop.timeout属性，一旦客户端的bpop.timeout属性小于当前时间戳，说明该客户端的WAIT超时时间到时了，因此会调用replyToBlockedClientTimedOut，向客户端返回当前已发来WAIT确认的从节点个数，并调用unblockClient解除该客户端的阻塞。

2：确认检查

将server.get\_ack\_from\_slaves属性置为1后，在每次事件处理函数aeProcessEvents调用之前，都会调用的beforeSleep函数中，判断该属性为1后，就会调用函数replicationFeedSlaves，向所有从节点的输出缓存中，以及积压队列中，追加命令"REPLCONF GETACK \*"，也就是相当于向从节点发送该命令，然后置server.get\_ack\_from\_slaves为0。从节点收到该命令后，就会向主节点返回"REPLCONF GETACK <offset>"命令。

beforeSleep中，该部分代码如下：

/\* Send all the slaves an ACK request if at least one client blocked

\* during the previous event loop iteration. \*/

**if** **(**server**.**get\_ack\_from\_slaves**)** **{**

robj **\***argv**[**3**];**

argv**[**0**]** **=** createStringObject**(**"REPLCONF"**,**8**);**

argv**[**1**]** **=** createStringObject**(**"GETACK"**,**6**);**

argv**[**2**]** **=** createStringObject**(**"\*"**,**1**);** */\* Not used argument. \*/*

replicationFeedSlaves**(**server**.**slaves**,** server**.**slaveseldb**,** argv**,** 3**);**

decrRefCount**(**argv**[**0**]);**

decrRefCount**(**argv**[**1**]);**

decrRefCount**(**argv**[**2**]);**

server**.**get\_ack\_from\_slaves **=** 0**;**

**}**

*/\* Unblock all the clients blocked for synchronous replication*

*\* in WAIT. \*/*

**if** **(**listLength**(**server**.**clients\_waiting\_acks**))**

processClientsWaitingReplicas**();**

*/\* Try to process pending commands for clients that were just unblocked. \*/*

**if** **(**listLength**(**server**.**unblocked\_clients**))**

processUnblockedClients**();**

在beforeSleep中，只要列表server.clients\_waiting\_acks不为空，就调用函数processClientsWaitingReplicas，找到哪些因WAIT而阻塞的客户端可以解除阻塞了。函数processClientsWaitingReplicas的代码如下：

void processClientsWaitingReplicas**(**void**)** **{**

long long last\_offset **=** 0**;**

int last\_numreplicas **=** 0**;**

listIter li**;**

listNode **\***ln**;**

listRewind**(**server**.**clients\_waiting\_acks**,&**li**);**

**while((**ln **=** listNext**(&**li**)))** **{**

redisClient **\***c **=** ln**->**value**;**

*/\* Every time we find a client that is satisfied for a given*

*\* offset and number of replicas, we remember it so the next client*

*\* may be unblocked without calling replicationCountAcksByOffset()*

*\* if the requested offset / replicas were equal or less. \*/*

**if** **(**last\_offset **&&** last\_offset **>** c**->**bpop**.**reploffset **&&**

last\_numreplicas **>** c**->**bpop**.**numreplicas**)**

**{**

unblockClient**(**c**);**

addReplyLongLong**(**c**,**last\_numreplicas**);**

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

int numreplicas **=** replicationCountAcksByOffset**(**c**->**bpop**.**reploffset**);**

**if** **(**numreplicas **>=** c**->**bpop**.**numreplicas**)** **{**

last\_offset **=** c**->**bpop**.**reploffset**;**

last\_numreplicas **=** numreplicas**;**

unblockClient**(**c**);**

addReplyLongLong**(**c**,**numreplicas**);**

**}**

**}**

**}**

**}**

轮训列表server.clients\_waiting\_acks，针对其中的每一个客户端：

调用replicationCountAcksByOffset函数，得到当前复制偏移量大于c->bpop.reploffset的从节点个数numreplicas，如果numreplicas大于该客户端的c->bpop.numreplicas属性，说明该客户端的WAIT命令可以接触阻塞了，因此调用unblockClient解除该客户端的阻塞，并回复给该客户端numreplicas信息；

并且，将numreplicas记录到last\_numreplicas中，将刚接触阻塞的客户端的c->bpop.reploffset属性记录到last\_offset中。这样，当后续轮训其他客户端时，只要last\_numreplicas大于该客户端的c->bpop.numreplicas，并且last\_offset大于客户端的c->bpop.reploffset，说明该客户端也满足解除WAIT阻塞的条件，因此可以无需调用replicationCountAcksByOffset函数，而直接调用unblockClient解除该客户端的阻塞，并回复给该客户端numreplicas信息。

主从复制相关的代码注释，可以参考：

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