Redis集群是Redis提供的分布式数据库方案，通过分片来进行数据共享，并提供复制和故障转移功能。

**一：初始化**

1：数据结构

在源码中，通过server.cluster记录整个集群当前的状态，比如集群中的所有节点；集群目前的状态，比如是上线还是下线；集群当前的纪元等等。该属性是一个clusterState类型的结构体。该结构体的定义如下：

**typedef** struct clusterState **{**

clusterNode **\***myself**;** */\* This node \*/*

**...**

int state**;** */\* REDIS\_CLUSTER\_OK, REDIS\_CLUSTER\_FAIL, ... \*/*

int size**;** */\* Num of master nodes with at least one slot \*/*

dict **\***nodes**;** */\* Hash table of name -> clusterNode structures \*/*

**...**

clusterNode **\***slots**[**REDIS\_CLUSTER\_SLOTS**];**

zskiplist **\***slots\_to\_keys**;**

**...**

**}** clusterState**;**

myself指向当前Redis实例所表示的节点；state表示集群状态；字典nodes中记录了，包括自己在内的所有集群节点，该字典以节点名为key，以结构体clusterNode为value。其他属性与具体的流程相关，后续在介绍集群各种流程时会介绍。

集群中的节点是由clusterNode表示的，该结构体的定义如下：

**typedef** struct clusterNode **{**

mstime\_t ctime**;** */\* Node object creation time. \*/*

char name**[**REDIS\_CLUSTER\_NAMELEN**];** */\* Node name, hex string, sha1-size \*/*

int flags**;** */\* REDIS\_NODE\_... \*/*

**...**

mstime\_t ping\_sent**;** */\* Unix time we sent latest ping \*/*

mstime\_t pong\_received**;** */\* Unix time we received the pong \*/*

**...**

char ip**[**REDIS\_IP\_STR\_LEN**];** */\* Latest known IP address of this node \*/*

int port**;** */\* Latest known port of this node \*/*

clusterLink **\***link**;** */\* TCP/IP link with this node \*/*

list **\***fail\_reports**;** */\* List of nodes signaling this as failing \*/*

**}** clusterNode**;**

该结构体记录了节点的状态和属性。ctime表示节点的创建时间；name表示节点名，每个节点都有一个40字节长的随机字符串作为名字，该名字同时也作为该节点在字典server.cluster->nodes中的key；flags表示节点的类型和状态，比如节点是否下线，是主节点还是从节点等，都记录在标志位flags中；ip和port表示该节点的地址属性；link表示当前节点与该节点间的TCP连接，该结构中包含socket描述符、输入缓冲区和输出缓冲区等属性。在link所表示的TCP连接中，当前节点为客户端，clusterNode所表示的节点为服务端。其他属性与具体的流程相关，后续在介绍集群各种流程时会介绍。

2：初始化

Redis实例启动时，根据配置文件中的"cluster-enabled"选项，决定该Redis实例是否处于集群模式。如果该选项值为”yes”，则Redis实例中的server.cluster\_enabled被置为1，表示当前处于集群模式。

在集群模式下，Redis实例启动时，首先会调用clusterInit函数，初始化集群需要使用的结构，并创建监听端口。该函数的代码如下：

void clusterInit**(**void**)** **{**

int saveconf **=** 0**;**

server**.**cluster **=** zmalloc**(sizeof(**clusterState**));**

server**.**cluster**->**myself **=** **NULL;**

server**.**cluster**->**currentEpoch **=** 0**;**

server**.**cluster**->**state **=** REDIS\_CLUSTER\_FAIL**;**

server**.**cluster**->**size **=** 1**;**

server**.**cluster**->**todo\_before\_sleep **=** 0**;**

server**.**cluster**->**nodes **=** dictCreate**(&**clusterNodesDictType**,NULL);**

server**.**cluster**->**nodes\_black\_list **=**

dictCreate**(&**clusterNodesBlackListDictType**,NULL);**

server**.**cluster**->**failover\_auth\_time **=** 0**;**

server**.**cluster**->**failover\_auth\_count **=** 0**;**

server**.**cluster**->**failover\_auth\_rank **=** 0**;**

server**.**cluster**->**failover\_auth\_epoch **=** 0**;**

server**.**cluster**->**cant\_failover\_reason **=** REDIS\_CLUSTER\_CANT\_FAILOVER\_NONE**;**

server**.**cluster**->**lastVoteEpoch **=** 0**;**

server**.**cluster**->**stats\_bus\_messages\_sent **=** 0**;**

server**.**cluster**->**stats\_bus\_messages\_received **=** 0**;**

memset**(**server**.**cluster**->**slots**,**0**,** **sizeof(**server**.**cluster**->**slots**));**

clusterCloseAllSlots**();**

*/\* Lock the cluster config file to make sure every node uses*

*\* its own nodes.conf. \*/*

**if** **(**clusterLockConfig**(**server**.**cluster\_configfile**)** **==** REDIS\_ERR**)**

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

*/\* Load or create a new nodes configuration. \*/*

**if** **(**clusterLoadConfig**(**server**.**cluster\_configfile**)** **==** REDIS\_ERR**)** **{**

*/\* No configuration found. We will just use the random name provided*

*\* by the createClusterNode() function. \*/*

myself **=** server**.**cluster**->**myself **=**

createClusterNode**(NULL,**REDIS\_NODE\_MYSELF**|**REDIS\_NODE\_MASTER**);**

redisLog**(**REDIS\_NOTICE**,**"No cluster configuration found, I'm %.40s"**,**

myself**->**name**);**

clusterAddNode**(**myself**);**

saveconf **=** 1**;**

**}**

**if** **(**saveconf**)** clusterSaveConfigOrDie**(**1**);**

*/\* We need a listening TCP port for our cluster messaging needs. \*/*

server**.**cfd\_count **=** 0**;**

*/\* Port sanity check II*

*\* The other handshake port check is triggered too late to stop*

*\* us from trying to use a too-high cluster port number. \*/*

**if** **(**server**.**port **>** **(**65535**-**REDIS\_CLUSTER\_PORT\_INCR**))** **{**

redisLog**(**REDIS\_WARNING**,** "Redis port number too high. "

"Cluster communication port is 10,000 port "

"numbers higher than your Redis port. "

"Your Redis port number must be "

"lower than 55535."**);**

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

**}**

**if** **(**listenToPort**(**server**.**port**+**REDIS\_CLUSTER\_PORT\_INCR**,**

server**.**cfd**,&**server**.**cfd\_count**)** **==** REDIS\_ERR**)**

**{**

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

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

int j**;**

**for** **(**j **=** 0**;** j **<** server**.**cfd\_count**;** j**++)** **{**

**if** **(**aeCreateFileEvent**(**server**.**el**,** server**.**cfd**[**j**],** AE\_READABLE**,**

clusterAcceptHandler**,** **NULL)** **==** AE\_ERR**)**

redisPanic**(**"Unrecoverable error creating Redis Cluster "

"file event."**);**

**}**

**}**

*/\* The slots -> keys map is a sorted set. Init it. \*/*

server**.**cluster**->**slots\_to\_keys **=** zslCreate**();**

*/\* Set myself->port to my listening port, we'll just need to discover*

*\* the IP address via MEET messages. \*/*

myself**->**port **=** server**.**port**;**

server**.**cluster**->**mf\_end **=** 0**;**

resetManualFailover**();**

**}**

在该函数中，首先初始化clusterState结构类型server.cluster中的各个属性；

如果在Redis配置文件中指定了"cluster-config-file"选项的值，则用server.cluster\_configfile属性记录该选项值，表示集群配置文件。接下来，就根据配置文件的内容，初始化server.cluster中的各个属性；

如果加载集群配置文件失败（或者配置文件不存在），则以REDIS\_NODE\_MYSELF和REDIS\_NODE\_MASTER为标记，创建一个clusterNode结构表示自己本身，置为主节点，并设置自己的名字为一个40字节的随机串；然后将该节点添加到server.cluster->nodes中；

接下来，调用listenToPort函数，在集群监端口上创建socket描述符进行监听。该集群监听端口是在Redis监听端口基础上加10000，比如如果Redis监听客户端的端口为6379，则集群监听端口就是16379，该监听端口用于接收其他集群节点的TCP建链，集群中的每个节点，都会与其他节点进行建链，因此整个集群就形成了一个强连通网状图；

然后注册监听端口上的可读事件，事件回调函数为clusterAcceptHandler。

当当前节点收到其他集群节点发来的TCP建链请求之后，就会调用clusterAcceptHandler函数accept连接。在clusterAcceptHandler函数中，对于每个已经accept的链接，都会创建一个clusterLink结构表示该链接，并注册socket描述符上的可读事件，事件回调函数为clusterReadHandler。

**二：集群节点间的握手**

1：CLUSTER MEET命令

Redis实例以集群模式启动之后，此时，在它的视角中，当前集群只有他自己一个节点。如何认识集群中的其他节点呢，这就需要客户端发送”CLUSTER MEET”命令。

客户端向集群节点A发送命令” CLUSTER MEET nodeB\_ip nodeB\_port”， 其中的nodeB\_ip和nodeB\_port，表示节点B的ip和port。节点A收到客户端发来的该命令后，调用clusterCommand函数处理。这部分的代码如下：

**if** **(!**strcasecmp**(**c**->**argv**[**1**]->**ptr**,**"meet"**)** **&&** c**->**argc **==** 4**)** **{**

long long port**;**

**if** **(**getLongLongFromObject**(**c**->**argv**[**3**],** **&**port**)** **!=** REDIS\_OK**)** **{**

addReplyErrorFormat**(**c**,**"Invalid TCP port specified: %s"**,**

**(**char**\*)**c**->**argv**[**3**]->**ptr**);**

**return;**

**}**

**if** **(**clusterStartHandshake**(**c**->**argv**[**2**]->**ptr**,**port**)** **==** 0 **&&**

errno **==** EINVAL**)**

**{**

addReplyErrorFormat**(**c**,**"Invalid node address specified: %s:%s"**,**

**(**char**\*)**c**->**argv**[**2**]->**ptr**,** **(**char**\*)**c**->**argv**[**3**]->**ptr**);**

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

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

**}**

**}**

以命令中的ip和port为参数，调用clusterStartHandshake函数，节点A开始向节点B进行握手。

在clusterStartHandshake函数中，会以REDIS\_NODE\_HANDSHAKE|REDIS\_NODE\_MEET为标志，创建一个clusterNode结构表示节点B，该结构的ip和port属性分别置为节点B的ip和port，并将该节点插入到字典server.cluster->nodes中。 这部分的代码如下：

*/\* Add the node with a random address (NULL as first argument to*

*\* createClusterNode()). Everything will be fixed during the*

*\* handshake. \*/*

n **=** createClusterNode**(NULL,**REDIS\_NODE\_HANDSHAKE**|**REDIS\_NODE\_MEET**);**

memcpy**(**n**->**ip**,**norm\_ip**,sizeof(**n**->**ip**));**

n**->**port **=** port**;**

clusterAddNode**(**n**);**

注意，因为此时A还不知道节点B的名字，因此以NULL为参数调用函数createClusterNode，该函数中，会暂时以一个随机串当做B的名字，后续交互过程中，节点B会在PONG包中发来自己的名字。

2：TCP建链

在集群定时器函数clusterCron中，会轮训字典server.cluster->nodes中的每一个节点node，一旦发现node->link为NULL，就表示尚未向该节点建链（或是之前的连接已断开）。因此，开始向其集群端口发起TCP建链，这部分代码如下：

**if** **(**node**->**link **==** **NULL)** **{**

int fd**;**

mstime\_t old\_ping\_sent**;**

clusterLink **\***link**;**

fd **=** anetTcpNonBlockBindConnect**(**server**.**neterr**,** node**->**ip**,**

node**->**port**+**REDIS\_CLUSTER\_PORT\_INCR**,** REDIS\_BIND\_ADDR**);**

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

*/\* We got a synchronous error from connect before*

*\* clusterSendPing() had a chance to be called.*

*\* If node->ping\_sent is zero, failure detection can't work,*

*\* so we claim we actually sent a ping now (that will*

*\* be really sent as soon as the link is obtained). \*/*

**if** **(**node**->**ping\_sent **==** 0**)** node**->**ping\_sent **=** mstime**();**

redisLog**(**REDIS\_DEBUG**,** "Unable to connect to "

"Cluster Node [%s]:%d -> %s"**,** node**->**ip**,**

node**->**port**+**REDIS\_CLUSTER\_PORT\_INCR**,**

server**.**neterr**);**

**continue;**

**}**

link **=** createClusterLink**(**node**);**

link**->**fd **=** fd**;**

node**->**link **=** link**;**

aeCreateFileEvent**(**server**.**el**,**link**->**fd**,**AE\_READABLE**,**

clusterReadHandler**,**link**);**

*/\* Queue a PING in the new connection ASAP: this is crucial*

*\* to avoid false positives in failure detection.*

*\**

*\* If the node is flagged as MEET, we send a MEET message instead*

*\* of a PING one, to force the receiver to add us in its node*

*\* table. \*/*

old\_ping\_sent **=** node**->**ping\_sent**;**

clusterSendPing**(**link**,** node**->**flags **&** REDIS\_NODE\_MEET **?**

CLUSTERMSG\_TYPE\_MEET **:** CLUSTERMSG\_TYPE\_PING**);**

**if** **(**old\_ping\_sent**)** **{**

*/\* If there was an active ping before the link was*

*\* disconnected, we want to restore the ping time, otherwise*

*\* replaced by the clusterSendPing() call. \*/*

node**->**ping\_sent **=** old\_ping\_sent**;**

**}**

*/\* We can clear the flag after the first packet is sent.*

*\* If we'll never receive a PONG, we'll never send new packets*

*\* to this node. Instead after the PONG is received and we*

*\* are no longer in meet/handshake status, we want to send*

*\* normal PING packets. \*/*

node**->**flags **&=** **~**REDIS\_NODE\_MEET**;**

redisLog**(**REDIS\_DEBUG**,**"Connecting with Node %.40s at %s:%d"**,**

node**->**name**,** node**->**ip**,** node**->**port**+**REDIS\_CLUSTER\_PORT\_INCR**);**

**}**

当前节点A调用anetTcpNonBlockBindConnect函数，开始向节点B发起非阻塞的TCP建链，然后调用createClusterLink，创建clusterLink结构link，在这种连接中，节点B为服务端，当前节点为客户端；然后注册link->fd上的可读事件，事件回调函数为clusterReadHandler；

然后根据节点标志位中是否有REDIS\_NODE\_MEET标记，向该节点发送MEET包或者PING包；最后清除节点标志位中的REDIS\_NODE\_MEET标记。（该非阻塞的建链过程，没有判断建链成功或失败的步骤，只要可写事件触发，直接发送MEET或PING包，如果发送成功，则说明之前建链成功了，如果发送失败，则说明建链失败，会直接释放该链接）。

节点B在集群端口上收到其他集群节点发来的消息之后，触发其监听端口上的可读事件，事件回调函数clusterReadHandler中，调用read读取其他节点发来的数据。当收齐一个包的所有数据后，调用clusterProcessPacket函数处理该包。

在clusterProcessPacke函数中，首先尝试在server.cluster->nodes字典中，以发送者的名字为key寻找发送者节点sender，因为此时节点B对于节点A一无所知，自然找不到对应的节点。

如果找不到发送者节点，并且收到的报文为MEET报文，则以REDIS\_NODE\_HANDSHAKE为标志，创建一个clusterNode结构表示节点A，该结构的ip和port分别置为节点A的ip和port，并将该节点插入到字典server.cluster->nodes中。并回复PONG包给节点A。这部分的代码如下：

**if** **(**type **==** CLUSTERMSG\_TYPE\_PING **||** type **==** CLUSTERMSG\_TYPE\_MEET**)** **{**

redisLog**(**REDIS\_DEBUG**,**"Ping packet received: %p"**,** **(**void**\*)**link**->**node**);**

**...**

*/\* Add this node if it is new for us and the msg type is MEET.*

*\* In this stage we don't try to add the node with the right*

*\* flags, slaveof pointer, and so forth, as this details will be*

*\* resolved when we'll receive PONGs from the node. \*/*

**if** **(!**sender **&&** type **==** CLUSTERMSG\_TYPE\_MEET**)** **{**

clusterNode **\***node**;**

node **=** createClusterNode**(NULL,**REDIS\_NODE\_HANDSHAKE**);**

nodeIp2String**(**node**->**ip**,**link**);**

node**->**port **=** ntohs**(**hdr**->**port**);**

clusterAddNode**(**node**);**

clusterDoBeforeSleep**(**CLUSTER\_TODO\_SAVE\_CONFIG**);**

**}**

**...**

*/\* Anyway reply with a PONG \*/*

clusterSendPing**(**link**,**CLUSTERMSG\_TYPE\_PONG**);**

**}**

注意，节点B这里调用createClusterNode函数创建clusterNode结构表示A节点时，也是以NULL为参数创建的，因此B不会设置A的名字，同样以一个随机串当做其名字，后续在节点B向节点A握手时，节点A会在PONG包中发来自己的名字。

节点A在集群端口上收到节点B发来的PONG回复包之后，触发其监听端口上的可读事件，调用回调函数clusterReadHandler，同样也调用clusterProcessPacket函数处理该包。

同样的，也是在server.cluster->nodes字典中，以包中发送者的名字为key寻找匹配的节点。因为此时A尚不知道B的名字，因此还找不到对应的sender。

此时在A中，节点B尚处于REDIS\_NODE\_HANDSHAKE状态，因此，利用PONG包中B的名字更新节点B中的name属性，并清除节点B标志位中的REDIS\_NODE\_HANDSHAKE标记。并根据节点B在PONG包中填写的角色信息，将REDIS\_NODE\_MASTER或REDIS\_NODE\_SLAVE标记增加到B节点中的标志位中。这部分的代码如下：

**if** **(**type **==** CLUSTERMSG\_TYPE\_PING **||** type **==** CLUSTERMSG\_TYPE\_PONG **||**

type **==** CLUSTERMSG\_TYPE\_MEET**)**

**{**

redisLog**(**REDIS\_DEBUG**,**"%s packet received: %p"**,**

type **==** CLUSTERMSG\_TYPE\_PING **?** "ping" **:** "pong"**,**

**(**void**\*)**link**->**node**);**

**if** **(**link**->**node**)** **{**

**if** **(**nodeInHandshake**(**link**->**node**))** **{**

*/\* If we already have this node, try to change the*

*\* IP/port of the node with the new one. \*/*

**if** **(**sender**)** **{**

**...**

**}**

*/\* First thing to do is replacing the random name with the*

*\* right node name if this was a handshake stage. \*/*

clusterRenameNode**(**link**->**node**,** hdr**->**sender**);**

redisLog**(**REDIS\_DEBUG**,**"Handshake with node %.40s completed."**,**

link**->**node**->**name**);**

link**->**node**->**flags **&=** **~**REDIS\_NODE\_HANDSHAKE**;**

link**->**node**->**flags **|=** flags**&(**REDIS\_NODE\_MASTER**|**REDIS\_NODE\_SLAVE**);**

clusterDoBeforeSleep**(**CLUSTER\_TODO\_SAVE\_CONFIG**);**

**}**

**}**

**}**

至此，节点A向节点B的握手算是完成了。

在节点B中，收到A发来的MEET包后，也创建了相应的节点，并插入到server.cluster->nodes中。因此在节点B的clusterCron中，也会向A发起TCP建链。并且在建链成功之后，向该节点发送PING包，表示B开始向A发起握手过程。

A收到B发来的PING包后，会回复一个PONG包。在B中，类似的，也调用clusterProcessPacket函数进行处理。同样也在server.cluster->nodes字典中，以发送者的名字寻找匹配的节点。因为之前B没有设置A的名字，因此还找不到对应的sender。

此时在B中，节点A尚处于REDIS\_NODE\_HANDSHAKE状态，因此，利用PONG包中A的名字更新节点A中的name属性，并清除节点A标志位中的REDIS\_NODE\_HANDSHAKE标记。并根据节点A在PONG包中填写的角色信息，将REDIS\_NODE\_MASTER或REDIS\_NODE\_SLAVE标记增加到A节点中的标志位中。

至此，节点B向节点A的握手也算是完成了。节点A和B它们算是相互认识了。

**三：Gossip**

这里还有一个问题，如果集群中共有N个节点的话，当有新节点加入进来时，难道对于其中的每个节点，都需要发送一次”CLUSTER MEET”命令，该节点才能被集群中的其他节点所认识吗？当然不会这么做，只要通过Gossip协议，只需向集群中的任一节点发送命令，新结点就能加入到集群中，被其他所有节点所认识。

Gossip是分布式系统中被广泛使用的协议，其主要用于实现分布式节点之间的信息交换。Gossip算法如其名，灵感来自于办公室八卦，只要一个人八卦一下，在有限的时间内所有的人都会知道该八卦的信息，也就是所谓的”一传十，十传百”。这种方式也与病毒传播类似，因此Gossip有众多的别名“闲话算法”、“疫情传播算法”、“病毒感染算法”、“谣言传播算法”。

Gossip的特点是：在一个有界网络中，每个节点都随机地与其他节点通信，经过一番杂乱无章的通信，最终所有节点的状态都会达成一致。每个节点可能知道所有其他节点，也可能仅知道几个邻居节点，只要这些节可以通过网络连通，最终他们的状态都是一致的，当然这也是疫情传播的特点。

Gossip是一个最终一致性算法。虽然无法保证在某个时刻所有节点状态一致，但可以保证在”最终“所有节点一致，”最终“是一个现实中存在，但理论上无法证明的时间点。但Gossip的缺点也很明显，冗余通信会对网路带宽、CPU资源造成很大的负载。

具体到Redis集群中而言，Redis集群中的每个节点，每隔一段时间就会向其他节点发送心跳包，心跳包中除了包含自己的信息之外，还会包含若干我认识的其他节点的信息，这就是所谓的gossip部分。

节点收到心跳包后，会检查其中是否包含自己所不认识的节点，若有，就会向该节点发起握手流程。

举个例子，如果集群中，有A、B、C、D四个节点，A和B相互认识，C和D相互认识，此时只要客户端向A发送” CLUSTER MEET nodeC\_ip nodeC\_port”命令，则A在向节点C发送MEET包时，该MEET包中还会带有节点B的信息，C收到该MEET包后，不但认识了A节点，也会认识B节点。同样，C后续在向A和B发送PING包时，该PING包中也会带有节点D的信息，这样A和B也就认识了D节点。因此，经过一段时间之后，A、B、C、D四个节点就相互认识了。

在源码中，调用clusterSendPing函数向其他集群节点发送心跳包或MEET包，心跳包可以是PING、PONG包。PING、PONG和MEET包，三种包的格式是一样的，只是通过包头中的type属性来区分不同的包。该函数的源码如下，其中参数type指明了包的类型；link表示发送报文的TCP连接：

void clusterSendPing**(**clusterLink **\***link**,** int type**)** **{**

unsigned char **\***buf**;**

clusterMsg **\***hdr**;**

int gossipcount **=** 0**;** */\* Number of gossip sections added so far. \*/*

int wanted**;** */\* Number of gossip sections we want to append if possible. \*/*

int totlen**;** */\* Total packet length. \*/*

*/\* freshnodes is the max number of nodes we can hope to append at all:*

*\* nodes available minus two (ourself and the node we are sending the*

*\* message to). However practically there may be less valid nodes since*

*\* nodes in handshake state, disconnected, are not considered. \*/*

int freshnodes **=** dictSize**(**server**.**cluster**->**nodes**)-**2**;**

*/\* How many gossip sections we want to add? 1/10 of the number of nodes*

*\* and anyway at least 3. Why 1/10?*

*\**

*\* If we have N masters, with N/10 entries, and we consider that in*

*\* node\_timeout we exchange with each other node at least 4 packets*

*\* (we ping in the worst case in node\_timeout/2 time, and we also*

*\* receive two pings from the host), we have a total of 8 packets*

*\* in the node\_timeout\*2 falure reports validity time. So we have*

*\* that, for a single PFAIL node, we can expect to receive the following*

*\* number of failure reports (in the specified window of time):*

*\**

*\* PROB \* GOSSIP\_ENTRIES\_PER\_PACKET \* TOTAL\_PACKETS:*

*\**

*\* PROB = probability of being featured in a single gossip entry,*

*\* which is 1 / NUM\_OF\_NODES.*

*\* ENTRIES = 10.*

*\* TOTAL\_PACKETS = 2 \* 4 \* NUM\_OF\_MASTERS.*

*\**

*\* If we assume we have just masters (so num of nodes and num of masters*

*\* is the same), with 1/10 we always get over the majority, and specifically*

*\* 80% of the number of nodes, to account for many masters failing at the*

*\* same time.*

*\**

*\* Since we have non-voting slaves that lower the probability of an entry*

*\* to feature our node, we set the number of entires per packet as*

*\* 10% of the total nodes we have. \*/*

wanted **=** floor**(**dictSize**(**server**.**cluster**->**nodes**)/**10**);**

**if** **(**wanted **<** 3**)** wanted **=** 3**;**

**if** **(**wanted **>** freshnodes**)** wanted **=** freshnodes**;**

*/\* Compute the maxium totlen to allocate our buffer. We'll fix the totlen*

*\* later according to the number of gossip sections we really were able*

*\* to put inside the packet. \*/*

totlen **=** **sizeof(**clusterMsg**)-sizeof(**union clusterMsgData**);**

totlen **+=** **(sizeof(**clusterMsgDataGossip**)\***wanted**);**

*/\* Note: clusterBuildMessageHdr() expects the buffer to be always at least*

*\* sizeof(clusterMsg) or more. \*/*

**if** **(**totlen **<** **(**int**)sizeof(**clusterMsg**))** totlen **=** **sizeof(**clusterMsg**);**

buf **=** zcalloc**(**totlen**);**

hdr **=** **(**clusterMsg**\*)** buf**;**

*/\* Populate the header. \*/*

**if** **(**link**->**node **&&** type **==** CLUSTERMSG\_TYPE\_PING**)**

link**->**node**->**ping\_sent **=** mstime**();**

clusterBuildMessageHdr**(**hdr**,**type**);**

*/\* Populate the gossip fields \*/*

int maxiterations **=** wanted**\***3**;**

**while(**freshnodes **>** 0 **&&** gossipcount **<** wanted **&&** maxiterations**--)** **{**

dictEntry **\***de **=** dictGetRandomKey**(**server**.**cluster**->**nodes**);**

clusterNode **\***this **=** dictGetVal**(**de**);**

clusterMsgDataGossip **\***gossip**;**

int j**;**

*/\* Don't include this node: the whole packet header is about us*

*\* already, so we just gossip about other nodes. \*/*

**if** **(**this **==** myself**)** **continue;**

*/\* Give a bias to FAIL/PFAIL nodes. \*/*

**if** **(**maxiterations **>** wanted**\***2 **&&**

**!(**this**->**flags **&** **(**REDIS\_NODE\_PFAIL**|**REDIS\_NODE\_FAIL**)))**

**continue;**

*/\* In the gossip section don't include:*

*\* 1) Nodes in HANDSHAKE state.*

*\* 3) Nodes with the NOADDR flag set.*

*\* 4) Disconnected nodes if they don't have configured slots.*

*\*/*

**if** **(**this**->**flags **&** **(**REDIS\_NODE\_HANDSHAKE**|**REDIS\_NODE\_NOADDR**)** **||**

**(**this**->**link **==** **NULL** **&&** this**->**numslots **==** 0**))**

**{**

freshnodes**--;** */\* Tecnically not correct, but saves CPU. \*/*

**continue;**

**}**

*/\* Check if we already added this node \*/*

**for** **(**j **=** 0**;** j **<** gossipcount**;** j**++)** **{**

**if** **(**memcmp**(**hdr**->**data**.**ping**.**gossip**[**j**].**nodename**,**this**->**name**,**

REDIS\_CLUSTER\_NAMELEN**)** **==** 0**)** **break;**

**}**

**if** **(**j **!=** gossipcount**)** **continue;**

*/\* Add it \*/*

freshnodes**--;**

gossip **=** **&(**hdr**->**data**.**ping**.**gossip**[**gossipcount**]);**

memcpy**(**gossip**->**nodename**,**this**->**name**,**REDIS\_CLUSTER\_NAMELEN**);**

gossip**->**ping\_sent **=** htonl**(**this**->**ping\_sent**);**

gossip**->**pong\_received **=** htonl**(**this**->**pong\_received**);**

memcpy**(**gossip**->**ip**,**this**->**ip**,sizeof(**this**->**ip**));**

gossip**->**port **=** htons**(**this**->**port**);**

gossip**->**flags **=** htons**(**this**->**flags**);**

gossip**->**notused1 **=** 0**;**

gossip**->**notused2 **=** 0**;**

gossipcount**++;**

**}**

*/\* Ready to send... fix the totlen fiend and queue the message in the*

*\* output buffer. \*/*

totlen **=** **sizeof(**clusterMsg**)-sizeof(**union clusterMsgData**);**

totlen **+=** **(sizeof(**clusterMsgDataGossip**)\***gossipcount**);**

hdr**->**count **=** htons**(**gossipcount**);**

hdr**->**totlen **=** htonl**(**totlen**);**

clusterSendMessage**(**link**,**buf**,**totlen**);**

zfree**(**buf**);**

**}**

包中不仅包含了当前节点的信息，还会包含本节点所记录的其他集群节点的信息，这就是所谓的gossip部分。接收者就是通过包中的gossip部分，认识其他集群节点，更新其他节点状态的。

这就面临一个问题，包中需要包含多少个节点信息呢？Redis目前是这样规定的：gossip部分的节点数应该是所有节点数的1/10，但是最少应该包含3个节点信息。之所以在gossip部分需要包含所有节点数的1/10，是为了能够在下线检测时间，也就是2倍的node\_timeout时间内，如果有节点下线的话，能够收到大部分集群节点发来的，关于该节点的下线报告；

1/10这个数是这样来的：如果共有N个集群节点，在超时时间node\_timeout内，当前节点最少会收到其他任一节点发来的4个心跳包：因节点最长经过node\_timeout/2时间，就会其他节点发送一次PING包。节点收到PING包后，会回复PONG包。因此，在node\_timeout时间内，当前节点会收到节点A发来的两个PING包，并且会收到节点A发来的，对于我发过去的PING包的回复包，也就是2个PONG包。因此，在下线监测时间node\_timeout\*2内，会收到其他任一集群节点发来的8个心跳包。因此，当前节点总共可以收到8\*N个心跳包，每个心跳包中，包含下线节点信息的概率是1/10，因此，收到下线报告的期望值就是8\*N\*(1/10)，也就是N\*80%，因此，这意味着可以收到大部分节点发来的下线报告。

变量freshnodes表示gossip部分可以包含节点数的最大值，该值是集群节点总数减去2，这个2，包含当前节点自己，以及接收者节点；

变量wanted，就表示gossip部分需要包含的实际节点数，也就是总节点数的1/10；

接下来计算发送报文占用的总内存空间totlen，并且为报文申请内存；

如果发送的PING包的话，还需要更新接收节点的ping\_sent属性；

接下来，调用clusterBuildMessageHdr，构建包头信息，包头中主要是当前节点本身的信息；

接下来开始在循环中，填充包的gossip部分，注意最大的循环遍历次数为3\*wanted。在循环中：

首先从字典server.cluster->nodes中随机取得一个节点；

如果该节点就是当前节点本身，则直接过滤；

如果当前遍历次数已经超过了2\*wanted，并且该节点没有标志为下线或疑似下线，则直接过滤。这么做是为了尽可能的在心跳包中包含下线节点的信息；

如果该节点处于握手或者NOADDR状态，或者当前节点与该节点没有建链并且该节点没有配置槽位，则直接过滤；

接下来，查看该节点是否已经添加到gossip部分了，若是，则直接过滤；剩下的，就是将该节点信息添加到gossip部分中；

心跳包构建完成之后，修正包的长度信息totlen，并将gossip部分的节点数，以及包的总长度，填充到包头中；最后，调用clusterSendMessage函数将包发送出去；

当当前节点收到其他节点发来的PING、PONG或MEET包后，调用clusterProcessPacket处理这种类型的包时，会调用clusterProcessGossipSection函数处理包中的gossip部分。在该函数中，针对包中gossip部分中的每个节点，如果当前节点已认识该节点，则利用其中的节点信息更新节点状态，如果还不认识该节点，就会向该节点发起握手流程。

clusterProcessGossipSection函数的代码如下：

void clusterProcessGossipSection**(**clusterMsg **\***hdr**,** clusterLink **\***link**)** **{**

uint16\_t count **=** ntohs**(**hdr**->**count**);**

clusterMsgDataGossip **\***g **=** **(**clusterMsgDataGossip**\*)** hdr**->**data**.**ping**.**gossip**;**

clusterNode **\***sender **=** link**->**node **?** link**->**node **:** clusterLookupNode**(**hdr**->**sender**);**

**while(**count**--)** **{**

uint16\_t flags **=** ntohs**(**g**->**flags**);**

clusterNode **\***node**;**

sds ci**;**

ci **=** representRedisNodeFlags**(**sdsempty**(),** flags**);**

redisLog**(**REDIS\_DEBUG**,**"GOSSIP %.40s %s:%d %s"**,**

g**->**nodename**,**

g**->**ip**,**

ntohs**(**g**->**port**),**

ci**);**

sdsfree**(**ci**);**

*/\* Update our state accordingly to the gossip sections \*/*

node **=** clusterLookupNode**(**g**->**nodename**);**

**if** **(**node**)** **{**

*/\* We already know this node.*

*Handle failure reports, only when the sender is a master. \*/*

**if** **(**sender **&&** nodeIsMaster**(**sender**)** **&&** node **!=** myself**)** **{**

**if** **(**flags **&** **(**REDIS\_NODE\_FAIL**|**REDIS\_NODE\_PFAIL**))** **{**

**if** **(**clusterNodeAddFailureReport**(**node**,**sender**))** **{**

redisLog**(**REDIS\_VERBOSE**,**

"Node %.40s reported node %.40s as not reachable."**,**

sender**->**name**,** node**->**name**);**

**}**

markNodeAsFailingIfNeeded**(**node**);**

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

**if** **(**clusterNodeDelFailureReport**(**node**,**sender**))** **{**

redisLog**(**REDIS\_VERBOSE**,**

"Node %.40s reported node %.40s is back online."**,**

sender**->**name**,** node**->**name**);**

**}**

**}**

**}**

*/\* If we already know this node, but it is not reachable, and*

*\* we see a different address in the gossip section, start an*

*\* handshake with the (possibly) new address: this will result*

*\* into a node address update if the handshake will be*

*\* successful. \*/*

**if** **(**node**->**flags **&** **(**REDIS\_NODE\_FAIL**|**REDIS\_NODE\_PFAIL**)** **&&**

**(**strcasecmp**(**node**->**ip**,**g**->**ip**)** **||** node**->**port **!=** ntohs**(**g**->**port**)))**

**{**

clusterStartHandshake**(**g**->**ip**,**ntohs**(**g**->**port**));**

**}**

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

*/\* If it's not in NOADDR state and we don't have it, we*

*\* start a handshake process against this IP/PORT pairs.*

*\**

*\* Note that we require that the sender of this gossip message*

*\* is a well known node in our cluster, otherwise we risk*

*\* joining another cluster. \*/*

**if** **(**sender **&&**

**!(**flags **&** REDIS\_NODE\_NOADDR**)** **&&**

**!**clusterBlacklistExists**(**g**->**nodename**))**

**{**

clusterStartHandshake**(**g**->**ip**,**ntohs**(**g**->**port**));**

**}**

**}**

*/\* Next node \*/*

g**++;**

**}**

**}**

首先得到sender：如果当前节点是作为客户端，收到了服务端的回复，则sender就是服务端节点；否则，就根据包中的发送者信息，在字典server.cluster->nodes中寻找相应的服务端节点，找不到则sender为NULL；

接下来，就是在循环中依次处理gossip部分中每一个节点信息：首先将节点A的信息记录日志；

然后根据节点名，在字典中server.cluster->nodes中寻找该节点，如果能找到该节点node，则这里主要是下线检测的流程，会在下一节中介绍，这里暂时略过。

如果没有找到node节点的信息，并且有sender信息（也就是sender已经是集群中一个可信的节点了），并且节点标志位中没有REDIS\_NODE\_NOADDR标记，并且该节点不在黑名单中，这说明node节点是集群中的新节点，因此调用clusterStartHandshake函数开始向该节点发起握手流程；

**四：心跳消息和下线检测**

1：心跳消息

集群中的每个节点，每隔一段时间就会向其他节点发送PING包，节点收到PING包之后，就会回复PONG包。PING包和PONG包具有相同的格式，通过包头的type字段区分类型。因此，将PING和PONG包都称为心跳包。

节点发送PING包的策略是：节点每隔1秒钟，就会从字典server.cluster->nodes中，随机挑选一个节点向其发送PING包。而且，还会轮训字典中的所有节点，如果已经超过 NODE\_TIMEOUT/2的时间，没有向该节点发送过PING包了，则会立即向该节点发送PING包。

节点发送PING包和收到PONG包时，会更新两个时间属性：ping\_sent和pong\_received。节点根据这两个属性判断是否需要向其他节点发送PING，以及其他节点是否下线。这两个属性的更新策略是：

node->ping\_sent：创建节点时，该属性置为0，当向node节点发送PING包后，该属性置为当时时间，当收到node节点对于PING的回复PONG包之后，该属性重置为0；

node->pong\_received：创建节点时，该属性置为0，向node节点发送PING包，当收到node节点对于PING的回复PONG包之后，该属性置为当时时间；

发送PING包的逻辑是在集群定时器函数clusterCron中处理的，这部分的代码如下：

void clusterCron**(**void**)** **{**

**...**

*/\* Ping some random node 1 time every 10 iterations, so that we usually ping*

*\* one random node every second. \*/*

**if** **(!(**iteration **%** 10**))** **{**

int j**;**

*/\* Check a few random nodes and ping the one with the oldest*

*\* pong\_received time. \*/*

**for** **(**j **=** 0**;** j **<** 5**;** j**++)** **{**

de **=** dictGetRandomKey**(**server**.**cluster**->**nodes**);**

clusterNode **\***this **=** dictGetVal**(**de**);**

*/\* Don't ping nodes disconnected or with a ping currently active. \*/*

**if** **(**this**->**link **==** **NULL** **||** this**->**ping\_sent **!=** 0**)** **continue;**

**if** **(**this**->**flags **&** **(**REDIS\_NODE\_MYSELF**|**REDIS\_NODE\_HANDSHAKE**))**

**continue;**

**if** **(**min\_pong\_node **==** **NULL** **||** min\_pong **>** this**->**pong\_received**)** **{**

min\_pong\_node **=** this**;**

min\_pong **=** this**->**pong\_received**;**

**}**

**}**

**if** **(**min\_pong\_node**)** **{**

redisLog**(**REDIS\_DEBUG**,**"Pinging node %.40s"**,** min\_pong\_node**->**name**);**

clusterSendPing**(**min\_pong\_node**->**link**,** CLUSTERMSG\_TYPE\_PING**);**

**}**

**}**

**...**

di **=** dictGetSafeIterator**(**server**.**cluster**->**nodes**);**

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

clusterNode **\***node **=** dictGetVal**(**de**);**

now **=** mstime**();** */\* Use an updated time at every iteration. \*/*

mstime\_t delay**;**

**if** **(**node**->**flags **&**

**(**REDIS\_NODE\_MYSELF**|**REDIS\_NODE\_NOADDR**|**REDIS\_NODE\_HANDSHAKE**))**

**continue;**

**...**

*/\* If we have currently no active ping in this instance, and the*

*\* received PONG is older than half the cluster timeout, send*

*\* a new ping now, to ensure all the nodes are pinged without*

*\* a too big delay. \*/*

**if** **(**node**->**link **&&**

node**->**ping\_sent **==** 0 **&&**

**(**now **-** node**->**pong\_received**)** **>** server**.**cluster\_node\_timeout**/**2**)**

**{**

clusterSendPing**(**node**->**link**,** CLUSTERMSG\_TYPE\_PING**);**

**continue;**

**}**

**...**

**}**

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

**...**

**}**

函数中的iteration是个静态变量，表示调用clusterCron函数的次数。因为该函数每隔100ms调用一次，因此该变量被10整除意味着1s的间隔时间。因此，每隔1s，就从字典server.cluster->nodes中随机挑选5个节点，这5个节点满足以下条件：连接正常，上一次向其发送的PING包已经收到了回复的PONG包；该节点不是我自己，也不处于握手状态。

然后，从这5个随机节点中，挑选出最早收到PONG回复的那个节点，向其发送PING包。

接下来，轮训字典server.cluster->nodes，只要其中的节点不是我自己，没有处于REDIS\_NODE\_NOADDR或者握手状态，就对该node节点做相应的处理：

如果与node的连接正常，并且上一次发送的PING包已经收到了相应的回复PONG包，并且距离收到该PONG包已经超过了server.cluster\_node\_timeout/2的时间，则直接向该节点发送PING包；

这种发送PING包的策略，如果NODE\_TIMEOUT被置为一个较小值，而总结点数较大时，集群内发送心跳包的总数会是比较大的。因为只要当前节点已经超过 NODE\_TIMEOUT/2的时间没有向某个节点没有发送过PING包了，则会立即向其发送PING包。比如，如果当前集群中有100个节点，而NODE\_TIMEOUT设置为60秒，则每个节点每隔30秒，就会向其他99个节点发送PING包，也就是说，每个节点平均每一秒就会发送3.3个PING包，100个节点，每秒就会发送330个PING包。

尽管可以降低发包数，但是目前尚未有关于带宽问题的报告，因此目前还是采用这种方法来发送心跳包。

2：下线检测

Redis集群节点是通过某个节点是否能及时回复PING包来判断该节点是否下线的。这里的下线包括两种状态：疑似下线(PFAIL)和下线(FAIL)。

如果当前节点已经长时间没有收到节点A对于PING包的回复了，就会将节点A标记为疑似下线。因此所谓疑似下线，就是仅从当前节点的视角来看，节点A已经不可达了。但是节点A是否真正的下线了，还需要征求其他节点的意见。

节点间交互的心跳包中，在其gossip部分会带有节点的状态信息，如果当前节点在收到的其他节点发来的心跳包中，有大多数节点都把节点A标记为PFAIL了，则当前节点就会认为节点A确实下线了，就将其标记为FAIL，表示该节点A确实下线。一旦将A标记为FAIL后，当前节点就会立即通过FAIL包，将节点A下线的消息广播给其他所有节点，这样最终所有节点都会标记节点A为FAIL状态了。

疑似下线和下线，比较类似于哨兵中的主观下线和客观下线。

如果节点已经超过server.cluster\_node\_timeout的时间没有回复当前节点的PING包了，则当前节点就会将该节点标记为疑似下线。这部分逻辑是在定时器函数clusterCron中处理的，这部分的代码如下：

void clusterCron**(**void**)** **{**

**...**

di **=** dictGetSafeIterator**(**server**.**cluster**->**nodes**);**

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

clusterNode **\***node **=** dictGetVal**(**de**);**

now **=** mstime**();** */\* Use an updated time at every iteration. \*/*

mstime\_t delay**;**

**if** **(**node**->**flags **&**

**(**REDIS\_NODE\_MYSELF**|**REDIS\_NODE\_NOADDR**|**REDIS\_NODE\_HANDSHAKE**))**

**continue;**

**...**

*/\* If we are waiting for the PONG more than half the cluster*

*\* timeout, reconnect the link: maybe there is a connection*

*\* issue even if the node is alive. \*/*

**if** **(**node**->**link **&&** */\* is connected \*/*

now **-** node**->**link**->**ctime **>**

server**.**cluster\_node\_timeout **&&** */\* was not already reconnected \*/*

node**->**ping\_sent **&&** */\* we already sent a ping \*/*

node**->**pong\_received **<** node**->**ping\_sent **&&** */\* still waiting pong \*/*

*/\* and we are waiting for the pong more than timeout/2 \*/*

now **-** node**->**ping\_sent **>** server**.**cluster\_node\_timeout**/**2**)**

**{**

*/\* Disconnect the link, it will be reconnected automatically. \*/*

freeClusterLink**(**node**->**link**);**

**}**

**...**

*/\* Check only if we have an active ping for this instance. \*/*

**if** **(**node**->**ping\_sent **==** 0**)** **continue;**

*/\* Compute the delay of the PONG. Note that if we already received*

*\* the PONG, then node->ping\_sent is zero, so can't reach this*

*\* code at all. \*/*

delay **=** now **-** node**->**ping\_sent**;**

**if** **(**delay **>** server**.**cluster\_node\_timeout**)** **{**

*/\* Timeout reached. Set the node as possibly failing if it is*

*\* not already in this state. \*/*

**if** **(!(**node**->**flags **&** **(**REDIS\_NODE\_PFAIL**|**REDIS\_NODE\_FAIL**)))** **{**

redisLog**(**REDIS\_DEBUG**,**"\*\*\* NODE %.40s possibly failing"**,**

node**->**name**);**

node**->**flags **|=** REDIS\_NODE\_PFAIL**;**

update\_state **=** 1**;**

**}**

**}**

**}**

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

**...**

**}**

在轮训字典server.cluster->nodes的过程中，只要其中的节点不是我自己，没有处于REDIS\_NODE\_NOADDR或者握手状态，就对该node节点做相应的处理：

如果与node节点的连接正常，并且建链时间已经超过了server.cluster\_node\_timeout，并且最近一次向该node节点发送的PING包，还没有收到回复的PONG包，并且距离最近一次向其发送PING包，已经超过了server.cluster\_node\_timeout/2，则直接释放该连接。这样下一次调用clusterCron时会重新向该节点建链，这是因为虽然网络暂时有问题，但是该node节点可能还是正常的，这么做可以避免因暂时的网咯问题，就标记该node节点下线；

如果距离上次向node发送PING包，已经超过了server.cluster\_node\_timeout的时间，则只要该node节点尚未被标记为PFAIL或FAIL，则将其标记为PFAIL，因此该节点目前处于疑似下线的状态；

一旦当前节点A将节点B标记为PFAIL之后，则当前节点A发出去的心跳包中，在gossip部分就可能会带有节点B的信息。其他节点C收到节点A的心跳包后，解析其中的gossip部分，发现B节点被A节点标记为PFAIL了，则就会将一个包含A节点的下线报告结构体clusterNodeFailReport插入到列表B->fail\_reports中。

clusterNodeFailReport结构体的定义如下：

**typedef** struct clusterNodeFailReport **{**

struct clusterNode **\***node**;** */\* Node reporting the failure condition. \*/*

mstime\_t time**;** */\* Time of the last report from this node. \*/*

**}** clusterNodeFailReport**;**

该结构体中，包含发送下线报告的节点node，以及最近一次该节点发来下线报告的时间戳。

在节点结构体clusterNode中，有一个下线报告列表fail\_reports，列表中的每个元素都是一个clusterNodeFailReport结构，该列表记录了将该节点B标记为疑似下线的所有其他节点。因此节点C收到节点A对于节点B的下线报告后，就会将包含A节点的下线报告结构体clusterNodeFailReport插入到列表B->fail\_reports中。

节点C每收到一次对于B节点的下线报告，就会统计列表B->fail\_reports中，报告时间在2倍server.cluster\_node\_timeout内的元素个数，若元素个数已经超过了集群节点的一半，则节点C就可以将节点B标记为下线（FAIL）了。

这部分的处理逻辑是在clusterProcessGossipSection函数中实现的。该函数的代码如下：

void clusterProcessGossipSection**(**clusterMsg **\***hdr**,** clusterLink **\***link**)** **{**

uint16\_t count **=** ntohs**(**hdr**->**count**);**

clusterMsgDataGossip **\***g **=** **(**clusterMsgDataGossip**\*)** hdr**->**data**.**ping**.**gossip**;**

clusterNode **\***sender **=** link**->**node **?** link**->**node **:** clusterLookupNode**(**hdr**->**sender**);**

**while(**count**--)** **{**

uint16\_t flags **=** ntohs**(**g**->**flags**);**

clusterNode **\***node**;**

sds ci**;**

ci **=** representRedisNodeFlags**(**sdsempty**(),** flags**);**

redisLog**(**REDIS\_DEBUG**,**"GOSSIP %.40s %s:%d %s"**,**

g**->**nodename**,**

g**->**ip**,**

ntohs**(**g**->**port**),**

ci**);**

sdsfree**(**ci**);**

*/\* Update our state accordingly to the gossip sections \*/*

node **=** clusterLookupNode**(**g**->**nodename**);**

**if** **(**node**)** **{**

*/\* We already know this node.*

*Handle failure reports, only when the sender is a master. \*/*

**if** **(**sender **&&** nodeIsMaster**(**sender**)** **&&** node **!=** myself**)** **{**

**if** **(**flags **&** **(**REDIS\_NODE\_FAIL**|**REDIS\_NODE\_PFAIL**))** **{**

**if** **(**clusterNodeAddFailureReport**(**node**,**sender**))** **{**

redisLog**(**REDIS\_VERBOSE**,**

"Node %.40s reported node %.40s as not reachable."**,**

sender**->**name**,** node**->**name**);**

**}**

markNodeAsFailingIfNeeded**(**node**);**

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

**if** **(**clusterNodeDelFailureReport**(**node**,**sender**))** **{**

redisLog**(**REDIS\_VERBOSE**,**

"Node %.40s reported node %.40s is back online."**,**

sender**->**name**,** node**->**name**);**

**}**

**}**

**}**

*/\* If we already know this node, but it is not reachable, and*

*\* we see a different address in the gossip section, start an*

*\* handshake with the (possibly) new address: this will result*

*\* into a node address update if the handshake will be*

*\* successful. \*/*

**if** **(**node**->**flags **&** **(**REDIS\_NODE\_FAIL**|**REDIS\_NODE\_PFAIL**)** **&&**

**(**strcasecmp**(**node**->**ip**,**g**->**ip**)** **||** node**->**port **!=** ntohs**(**g**->**port**)))**

**{**

clusterStartHandshake**(**g**->**ip**,**ntohs**(**g**->**port**));**

**}**

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

*/\* If it's not in NOADDR state and we don't have it, we*

*\* start a handshake process against this IP/PORT pairs.*

*\**

*\* Note that we require that the sender of this gossip message*

*\* is a well known node in our cluster, otherwise we risk*

*\* joining another cluster. \*/*

**if** **(**sender **&&**

**!(**flags **&** REDIS\_NODE\_NOADDR**)** **&&**

**!**clusterBlacklistExists**(**g**->**nodename**))**

**{**

clusterStartHandshake**(**g**->**ip**,**ntohs**(**g**->**port**));**

**}**

**}**

*/\* Next node \*/*

g**++;**

**}**

**}**

首先得到sender：如果当前节点是作为客户端，收到了服务端的回复，则sender就是服务端节点；否则，就根据包中的发送者信息，在字典server.cluster->nodes中寻找相应的节点，找不到则sender为NULL；

接下来，就是在循环中依次处理gossip部分中每一个节点信息：首先将节点A的信息记录日志；

然后根据节点名，在字典中server.cluster->nodes中寻找该节点，如果能找到该节点node，并且sender不为NULL，并且sender为主节点，并且节点node不是我，则如果包中标记该节点node为FAIL或者PFAIL，则调用clusterNodeAddFailureReport，将sender节点的下线报告，追加到列表node->fail\_reports中。然后调用markNodeAsFailingIfNeeded函数，在条件满足的情况下，将node标注为FAIL，并向其他所有节点广播发送FAIL包，以便能尽快通知其他节点。

如果包中没有标注该节点为FAIL或PFAIL，则调用clusterNodeDelFailureReport，清除列表node->fail\_reports中的sender节点的下线报告（如果有的话）；

接下来，如果node节点已经被当前节点标注为PFAIL或者FAIL了，并且包中对于该节点的地址信息与当前节点记录的不一致，则可能该节点有了新的地址，因此调用clusterStartHandshake函数，开始向新地址发起握手流程；

剩下的是处理新结点的部分，之前已经解析过了，不再赘述。

markNodeAsFailingIfNeeded函数的代码如下：

void markNodeAsFailingIfNeeded**(**clusterNode **\***node**)** **{**

int failures**;**

int needed\_quorum **=** **(**server**.**cluster**->**size **/** 2**)** **+** 1**;**

**if** **(!**nodeTimedOut**(**node**))** **return;** */\* We can reach it. \*/*

**if** **(**nodeFailed**(**node**))** **return;** */\* Already FAILing. \*/*

failures **=** clusterNodeFailureReportsCount**(**node**);**

*/\* Also count myself as a voter if I'm a master. \*/*

**if** **(**nodeIsMaster**(**myself**))** failures**++;**

**if** **(**failures **<** needed\_quorum**)** **return;** */\* No weak agreement from masters. \*/*

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

"Marking node %.40s as failing (quorum reached)."**,** node**->**name**);**

*/\* Mark the node as failing. \*/*

node**->**flags **&=** **~**REDIS\_NODE\_PFAIL**;**

node**->**flags **|=** REDIS\_NODE\_FAIL**;**

node**->**fail\_time **=** mstime**();**

*/\* Broadcast the failing node name to everybody, forcing all the other*

*\* reachable nodes to flag the node as FAIL. \*/*

**if** **(**nodeIsMaster**(**myself**))** clusterSendFail**(**node**->**name**);**

clusterDoBeforeSleep**(**CLUSTER\_TODO\_UPDATE\_STATE**|**CLUSTER\_TODO\_SAVE\_CONFIG**);**

**}**

本函数用于在条件满足的情况下，将节点node标记为下线(FAIL)状态。这里的条件是指：

node节点已经被当前节点标记为疑似下线了(PFAIL)；

在node节点的下线报告列表node->fail\_reports中，在2倍server.cluster\_node\_timeout的时间段内，有超过一半的节点都将node节点标记为PFAIL或FAIL了；

在函数中，如果node节点未被当前节点标记为PFAIL，则直接返回；如果node节点已经被标记为FAIL状态了，则直接返回；

然后调用clusterNodeFailureReportsCount统计下线报告列表node->fail\_reports中的元素个数failures。在clusterNodeFailureReportsCount中，会首先清除那些发来下线报告的时间已经超过2倍server.cluster\_node\_timeout的所有节点；

如果当前节点是主节点，则增加failures的值，因为当前节点也已把node节点标记为PFAIL了；

如果failures的值，没有超过所有节点数的一半，则直接返回；

接下来就是将node节点标记为FAIL状态了：首先清除node标志位中的REDIS\_NODE\_PFAIL标记，然后将REDIS\_NODE\_FAIL增加到node标志位中，更新node->fail\_time为当前时间；如果当前节点为主节点，则调用clusterSendFail向起他节点广播FAIL包，FAIL包中除了包头以外，就仅包含下线节点的名字nodename；

其他节点收到FAIL包后，在包处理函数clusterProcessPacket中，立即将该节点标记为下线(FAIL)，不管它之前是否已经将该节点标记为PFAIL了。这部分的代码如下：

**else** **if** **(**type **==** CLUSTERMSG\_TYPE\_FAIL**)** **{**

clusterNode **\***failing**;**

**if** **(**sender**)** **{**

failing **=** clusterLookupNode**(**hdr**->**data**.**fail**.**about**.**nodename**);**

**if** **(**failing **&&**

**!(**failing**->**flags **&** **(**REDIS\_NODE\_FAIL**|**REDIS\_NODE\_MYSELF**)))**

**{**

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

"FAIL message received from %.40s about %.40s"**,**

hdr**->**sender**,** hdr**->**data**.**fail**.**about**.**nodename**);**

failing**->**flags **|=** REDIS\_NODE\_FAIL**;**

failing**->**fail\_time **=** mstime**();**

failing**->**flags **&=** **~**REDIS\_NODE\_PFAIL**;**

clusterDoBeforeSleep**(**CLUSTER\_TODO\_SAVE\_CONFIG**|**

CLUSTER\_TODO\_UPDATE\_STATE**);**

**}**

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

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

"Ignoring FAIL message from unknown node %.40s about %.40s"**,**

hdr**->**sender**,** hdr**->**data**.**fail**.**about**.**nodename**);**

**}**

**}**

如果sender不为NULL，说明发送者是可信的。因此根据包中的节点名，从字典server.cluster->nodes中寻找对应的failing节点。如果能找到该failing节点，并且该节点尚未被标记为FAIL，并且该节点也不是当前节点本身，则将该failing节点标记为FAIL：

将REDIS\_NODE\_FAIL标记增加到节点标志位中，更新failing->fail\_time为当前时间；将标记REDIS\_NODE\_PFAIL从标志位中清除；

如果sender为NULL，则说明当前节点还不认识发送者，因此不做任何处理；

参考：

<http://redis.io/topics/cluster-spec#nodes-handshake>

<http://redis.io/topics/cluster-spec#fault-tolerance>