Redis服务器是典型的一对多服务器程序，通过使用由IO多路复用技术实现的文件事件处理器，Redis服务器使用单线程单进程的方式来处理命令请求，并与多个客户端进行网络通信。

Redis客户端与服务器之间通过TCP协议进行通信。TCP协议是一种流式协议，数据以字节流的形式进行传递，没有固有的"报文"或"报文边界"的概念，如果需要设置边界，需要应用层自行处理。

因此，Redis客户端与服务器之间的交互数据，都按照Redis自定义的统一请求协议的格式进行编码。使用这种协议，每条命令之间都有了“边界”。

举个例子，如果客户端要向服务器发送以下命令请求：

SET msg “hello world”

那么客户端实际发送的数据是：

\*3\r\n$3\r\nSET\r\n$3\r\nmsg\r\n$11\r\nhello world\r\n

服务器收到这样的数据时，就可以通过解析”\*3”得到该命令有3个参数，第一个参数长度为3，值为”SET”，也就是要执行的命令；第二个参数长度为3，值为”msg”；第三个参数长度为11，值为”hello world”。

这样就得到了一条完整的命令，解析并处理该命令后，接着解析下一条命令。

**一：客户端结构redisClient**

对于每个与服务器进行连接的客户端，服务器都为这些客户端建立了相应的redisClient结构，该结构体定义在redis.h中，它的定义如下（有省略）：

**typedef** struct redisClient **{**

uint64\_t id**;** */\* Client incremental unique ID. \*/*

int fd**;**

redisDb **\***db**;**

int dictid**;**

robj **\***name**;** */\* As set by CLIENT SETNAME \*/*

sds querybuf**;**

size\_t querybuf\_peak**;** */\* Recent (100ms or more) peak of querybuf size \*/*

int argc**;**

robj **\*\***argv**;**

struct redisCommand **\***cmd**,** **\***lastcmd**;**

int reqtype**;**

int multibulklen**;** */\* number of multi bulk arguments left to read \*/*

long bulklen**;** */\* length of bulk argument in multi bulk request \*/*

list **\***reply**;**

unsigned long reply\_bytes**;** */\* Tot bytes of objects in reply list \*/*

**...**

int flags**;** */\* REDIS\_SLAVE | REDIS\_MONITOR | REDIS\_MULTI ... \*/*

int authenticated**;** */\* when requirepass is non-NULL \*/*

**...**

*/\* Response buffer \*/*

int bufpos**;**

char buf**[**REDIS\_REPLY\_CHUNK\_BYTES**];**

**}** redisClient**;**

这个结构保存了客户端当前的状态信息，以及执行相关功能时需要用到的数据结构，比如：客户端的socket描述符(fd)，指向客户端正在使用的数据库的指针(db)，客户端的名字(name)，客户端的标志值(flags)，客户端输入缓存(querybuf)，客户端当前要执行的命令参数(argv)，以及参数个数(argc)，以及客户端的输出缓存(buf和reply)等。

这些属性的具体意义会在下面的章节中介绍。

**二：初始化（创建监听端口、注册建连事件）**

在Redis服务器的初始化函数initserver中，调用aeCreateEventLoop创建了Redis服务器中唯一的事件循环结构(aeEventLoop)：server.e1。server.e1是全局性的，Redis服务器中所有的事件都注册在该结构上。

默认情况下，Redis服务器监听本地所有网络接口上的连接(0.0.0.0)。可以在配置文件中，通过"bind"选项设置监听的地址，其后跟一个或多个空格分隔的IP地址，比如：

bind 192.168.1.100 10.0.0.1

Redis将这些地址保存在server.bindaddr中，IP地址总数为server.bindaddr\_count。

在initserver函数中，调用listenToPort，根据这些监听地址，调用socket、bind和listen创建监听socket描述符。

/\* Open the TCP listening socket for the user commands. \*/

**if** **(**server**.**port **!=** 0 **&&**

listenToPort**(**server**.**port**,** server**.**ipfd**, &**server**.**ipfd\_count**)** **==** REDIS\_ERR**)**

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

创建好的监听描述符保存在描述符数组server.ipfd中，最后创建的监听描述符的总数为server.ipfd\_count。server.ipfd数组为固定大小：REDIS\_BINDADDR\_MAX(16)，因此最多只支持16个监听地址。

然后，针对每个监听描述符，调用aeCreateFileEvent，注册其上的可读事件，回调函数为acceptTcpHandler：

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

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

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

**{**

redisPanic**(**

"Unrecoverable error creating server.ipfd file event."**);**

**}**

**}**

Redis服务器收到客户端的TCP连接后，就会调用acceptTcpHandler函数进行处理。acceptTcpHandler函数的代码如下：

void acceptTcpHandler**(**aeEventLoop **\***el**,** int fd**,** void **\***privdata**,** int mask**)** **{**

int cport**,** cfd**,** max **=** MAX\_ACCEPTS\_PER\_CALL**;**

char cip**[**REDIS\_IP\_STR\_LEN**];**

REDIS\_NOTUSED**(**el**);**

REDIS\_NOTUSED**(**mask**);**

REDIS\_NOTUSED**(**privdata**);**

**while(**max**--)** **{**

cfd **=** anetTcpAccept**(**server**.**neterr**,** fd**,** cip**,** **sizeof(**cip**),** **&**cport**);**

**if** **(**cfd **==** ANET\_ERR**)** **{**

**if** **(**errno **!=** EWOULDBLOCK**)**

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

"Accepting client connection: %s"**,** server**.**neterr**);**

**return;**

**}**

redisLog**(**REDIS\_VERBOSE**,**"Accepted %s:%d"**,** cip**,** cport**);**

acceptCommonHandler**(**cfd**,**0**);**

**}**

**}**

该函数每次最多处理MAX\_ACCEPTS\_PER\_CALL(1000)个连接，如果还有其他连接，则等到下次调用acceptTcpHandler时再处理，这样做的原因是为了保证该函数的执行时间不会过长，以免影响后续事件的处理。

针对每个连接，调用anetTcpAccept函数进行accept，并将客户端地址记录到cip以及cport中；

建链后的socket描述符为cfd，根据该值调用acceptCommonHandler，该函数中，调用createClient创建一个redisClient结构，并注册socket描述符上的可读事件，回调函数为readQueryFromClient。最后将该redisClient结构存储到全局客户端列表server.clients中；

**if** **(**aeCreateFileEvent**(**server**.**el**,**fd**,**AE\_READABLE**,**

readQueryFromClient**,** c**)** **==** AE\_ERR**)**

**{**

close**(**fd**);**

zfree**(**c**);**

**return** **NULL;**

**}**

**三：接收客户端请求，解析并处理请求**

1：接收数据

Redis服务器收到客户端的请求数据后，就会触发socket描述符上的可读事件，从而调用其回调函数readQueryFromClient。

在readQueryFromClient中，调用read读取客户端的请求，并缓存到redisClient结构中的输入缓存querybuf中，该输入缓存会根据接收到的数据长度动态扩容。接下来对收到的请求数据进行解析，并执行相应的命令处理函数。

readQueryFromClient函数代码如下：

void readQueryFromClient**(**aeEventLoop **\***el**,** int fd**,** void **\***privdata**,** int mask**)** **{**

redisClient **\***c **=** **(**redisClient**\*)** privdata**;**

int nread**,** readlen**;**

size\_t qblen**;**

REDIS\_NOTUSED**(**el**);**

REDIS\_NOTUSED**(**mask**);**

server**.**current\_client **=** c**;**

readlen **=** REDIS\_IOBUF\_LEN**;**

*/\* If this is a multi bulk request, and we are processing a bulk reply*

*\* that is large enough, try to maximize the probability that the query*

*\* buffer contains exactly the SDS string representing the object, even*

*\* at the risk of requiring more read(2) calls. This way the function*

*\* processMultiBulkBuffer() can avoid copying buffers to create the*

*\* Redis Object representing the argument. \*/*

**if** **(**c**->**reqtype **==** REDIS\_REQ\_MULTIBULK **&&** c**->**multibulklen **&&** c**->**bulklen **!=** **-**1

**&&** c**->**bulklen **>=** REDIS\_MBULK\_BIG\_ARG**)**

**{**

int remaining **=** **(**unsigned**)(**c**->**bulklen**+**2**)-**sdslen**(**c**->**querybuf**);**

**if** **(**remaining **<** readlen**)** readlen **=** remaining**;**

**}**

qblen **=** sdslen**(**c**->**querybuf**);**

**if** **(**c**->**querybuf\_peak **<** qblen**)** c**->**querybuf\_peak **=** qblen**;**

c**->**querybuf **=** sdsMakeRoomFor**(**c**->**querybuf**,** readlen**);**

nread **=** read**(**fd**,** c**->**querybuf**+**qblen**,** readlen**);**

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

**if** **(**errno **==** EAGAIN**)** **{**

nread **=** 0**;**

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

redisLog**(**REDIS\_VERBOSE**,** "Reading from client: %s"**,**strerror**(**errno**));**

freeClient**(**c**);**

**return;**

**}**

**}** **else** **if** **(**nread **==** 0**)** **{**

redisLog**(**REDIS\_VERBOSE**,** "Client closed connection"**);**

freeClient**(**c**);**

**return;**

**}**

**if** **(**nread**)** **{**

sdsIncrLen**(**c**->**querybuf**,**nread**);**

c**->**lastinteraction **=** server**.**unixtime**;**

**if** **(**c**->**flags **&** REDIS\_MASTER**)** c**->**reploff **+=** nread**;**

server**.**stat\_net\_input\_bytes **+=** nread**;**

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

server**.**current\_client **=** **NULL;**

**return;**

**}**

**if** **(**sdslen**(**c**->**querybuf**)** **>** server**.**client\_max\_querybuf\_len**)** **{**

sds ci **=** catClientInfoString**(**sdsempty**(),**c**),** bytes **=** sdsempty**();**

bytes **=** sdscatrepr**(**bytes**,**c**->**querybuf**,**64**);**

redisLog**(**REDIS\_WARNING**,**"Closing client that reached max query buffer length: %s (qbuf initial bytes: %s)"**,** ci**,** bytes**);**

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

sdsfree**(**bytes**);**

freeClient**(**c**);**

**return;**

**}**

processInputBuffer**(**c**);**

server**.**current\_client **=** **NULL;**

**}**

该函数中，首先设置每次read读取的最大字节数readlen为REDIS\_IOBUF\_LEN(16k)。然后得到输入缓存c->querybuf当前长度qblen，也就是已接收到的客户端请求数据的长度。根据qblen更新c->querybuf\_peak的值，该属性记录了输入缓存c->querybuf的最大长度。

接下来为c->querybuf扩容，使其能容纳readlen个字节；然后就调用read，最多读取readlen个字节。读取的内容追加到c->querybuf尾部。

如果read返回值nread为-1，若errno等于EAGAIN，说明暂无数据，置nread为0；否则记录错误信息到日志，释放客户端结构redisClient，并关闭链接，然后直接返回；

如果read返回0，说明客户端关闭连接，此时记录信息到日志，释放客户端结构redisClient，并关闭链接，然后直接返回；

read返回非0，说明读取到了数据。判断当前输入缓存c->querybuf的长度是否大于阈值server.client\_max\_querybuf\_len(1G)。若超过阈值，则记录当前客户端信息，以及c->querybuf中前64个字节到日志中，然后释放客户端结构redisClient，并关闭链接，然后直接返回；

最后，调用processInputBuffer解析收到的数据，并在读取到完整的一条命令请求之后，执行相应的命令处理函数。

2：解析处理客户端命令

Redis服务器收到客户端的请求数据后，调用processInputBuffer函数解析输入缓存redisClient->querybuf中的数据。在得到一条完整的命令请求数据后，就调用processCommand函数处理执行相应的命令。

processInputBuffer的代码如下：

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

*/\* Keep processing while there is something in the input buffer \*/*

**while(**sdslen**(**c**->**querybuf**))** **{**

*/\* Return if clients are paused. \*/*

**if** **(!(**c**->**flags **&** REDIS\_SLAVE**)** **&&** clientsArePaused**())** **return;**

*/\* Immediately abort if the client is in the middle of something. \*/*

**if** **(**c**->**flags **&** REDIS\_BLOCKED**)** **return;**

*/\* REDIS\_CLOSE\_AFTER\_REPLY closes the connection once the reply is*

*\* written to the client. Make sure to not let the reply grow after*

*\* this flag has been set (i.e. don't process more commands). \*/*

**if** **(**c**->**flags **&** REDIS\_CLOSE\_AFTER\_REPLY**)** **return;**

*/\* Determine request type when unknown. \*/*

**if** **(!**c**->**reqtype**)** **{**

**if** **(**c**->**querybuf**[**0**]** **==** '\*'**)** **{**

c**->**reqtype **=** REDIS\_REQ\_MULTIBULK**;**

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

c**->**reqtype **=** REDIS\_REQ\_INLINE**;**

**}**

**}**

**if** **(**c**->**reqtype **==** REDIS\_REQ\_INLINE**)** **{**

**if** **(**processInlineBuffer**(**c**)** **!=** REDIS\_OK**)** **break;**

**}** **else** **if** **(**c**->**reqtype **==** REDIS\_REQ\_MULTIBULK**)** **{**

**if** **(**processMultibulkBuffer**(**c**)** **!=** REDIS\_OK**)** **break;**

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

redisPanic**(**"Unknown request type"**);**

**}**

*/\* Multibulk processing could see a <= 0 length. \*/*

**if** **(**c**->**argc **==** 0**)** **{**

resetClient**(**c**);**

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

*/\* Only reset the client when the command was executed. \*/*

**if** **(**processCommand**(**c**)** **==** REDIS\_OK**)**

resetClient**(**c**);**

**}**

**}**

**}**

该函数中，只要c->querybuf不为空，就一直循环处理。在该循环中：

首先，根据客户端的当前状态标志c->flags，判断是否需要继续解析处理，比如：

如果当前客户端不是SLAVE节点，并且客户端处于阻塞状态，则直接返回；

如果客户端标志c->flags包含REDIS\_BLOCKED，则直接返回；

如果客户端标志c->flags包含REDIS\_CLOSE\_AFTER\_REPLY，则直接返回。该标志表明发生了异常，服务器不再需要处理客户端请求，在回复客户端错误消息后直接关闭链接。

接下来，如果c->reqtype为0，说明刚要开始处理一条请求（第一次处理c->querybuf中的数据，或刚处理完一条完整的命令请求）。如果数据c->querybuf的首字节为'\*'，说明该请求会跨越多行（包含多个”\r\n”），则置c->reqtype为EDIS\_REQ\_MULTIBULK；否则说明该请求为单行请求，置c->reqtype为REDIS\_REQ\_INLINE；

如果c->reqtype为REDIS\_REQ\_INLINE，则调用processInlineBuffer解析单行请求，如果c->reqtype为EDIS\_REQ\_MULTIBULK，则调用processMultibulkBuffer解析多行请求。这两个函数的返回值如果不是REDIS\_OK，则说明尚未收到一条完整的请求，需要退出循环，函数返回后接着读取剩余的数据；

如果这两个函数返回为REDIS\_OK，则说明已经收到并解析好了一条完整的请求，命令的参数已经分解到数组c->argv中，c->argc表示参数个数。

如果c->argc为0，则无需处理，直接调用resetClient重置客户端状态，也就是释放c->argv数组中的元素，置c->argc、c->reqtype和c->multibulklen为0，置c->bulklen为-1等。然后接着处理c->querybuf中剩下的内容；

如果c->argc非0，则调用processCommand处理该命令，调用相应的命令处理函数。处理成功后，调用resetClient重置客户端状态。然后接着处理c->querybuf中剩下的内容。

函数processInlineBuffer和processMultibulkBuffer分别解析客户端的单行请求和多行请求。这两个函数返回REDIS\_OK，说明已经收到并解析好了一条完整的请求，命令的参数已经分解到数组c->argv中，c->argc表示参数个数。

如果这俩函数返回REDIS\_ERR，要么说明收到的客户端命令请求尚不完整，这其实不是错误，这种情况下函数返回后，服务器需要继续接收客户端请求；要么说明客户端发来的请求不符合统一请求协议的格式要求，这种情况下调用setProtocolError删除c->querybuf相应的内容，并且将客户端的标志位c->flags增加REDIS\_CLOSE\_AFTER\_REPLY标记，从而在回复客户端错误信息后直接关闭连接。

processMultibulkBuffer函数要比processInlineBuffer稍微复杂一些，直接看一下processMultibulkBuffer的实现：

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

char **\***newline **=** **NULL;**

int pos **=** 0**,** ok**;**

long long ll**;**

**if** **(**c**->**multibulklen **==** 0**)** **{**

*/\* The client should have been reset \*/*

redisAssertWithInfo**(**c**,NULL,**c**->**argc **==** 0**);**

*/\* Multi bulk length cannot be read without a \r\n \*/*

newline **=** strchr**(**c**->**querybuf**,**'\r'**);**

**if** **(**newline **==** **NULL)** **{**

**if** **(**sdslen**(**c**->**querybuf**)** **>** REDIS\_INLINE\_MAX\_SIZE**)** **{**

addReplyError**(**c**,**"Protocol error: too big mbulk count string"**);**

setProtocolError**(**c**,**0**);**

**}**

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

**}**

*/\* Buffer should also contain \n \*/*

**if** **(**newline**-(**c**->**querybuf**)** **>** **((**signed**)**sdslen**(**c**->**querybuf**)-**2**))**

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

*/\* We know for sure there is a whole line since newline != NULL,*

*\* so go ahead and find out the multi bulk length. \*/*

redisAssertWithInfo**(**c**,NULL,**c**->**querybuf**[**0**]** **==** '\*'**);**

ok **=** string2ll**(**c**->**querybuf**+**1**,**newline**-(**c**->**querybuf**+**1**),&**ll**);**

**if** **(!**ok **||** ll **>** 1024**\***1024**)** **{**

addReplyError**(**c**,**"Protocol error: invalid multibulk length"**);**

setProtocolError**(**c**,**pos**);**

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

**}**

pos **=** **(**newline**-**c**->**querybuf**)+**2**;**

**if** **(**ll **<=** 0**)** **{**

sdsrange**(**c**->**querybuf**,**pos**,-**1**);**

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

**}**

c**->**multibulklen **=** ll**;**

*/\* Setup argv array on client structure \*/*

**if** **(**c**->**argv**)** zfree**(**c**->**argv**);**

c**->**argv **=** zmalloc**(sizeof(**robj**\*)\***c**->**multibulklen**);**

**}**

redisAssertWithInfo**(**c**,NULL,**c**->**multibulklen **>** 0**);**

**while(**c**->**multibulklen**)** **{**

*/\* Read bulk length if unknown \*/*

**if** **(**c**->**bulklen **==** **-**1**)** **{**

newline **=** strchr**(**c**->**querybuf**+**pos**,**'\r'**);**

**if** **(**newline **==** **NULL)** **{**

**if** **(**sdslen**(**c**->**querybuf**)** **>** REDIS\_INLINE\_MAX\_SIZE**)** **{**

addReplyError**(**c**,**

"Protocol error: too big bulk count string"**);**

setProtocolError**(**c**,**0**);**

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

**}**

**break;**

**}**

*/\* Buffer should also contain \n \*/*

**if** **(**newline**-(**c**->**querybuf**)** **>** **((**signed**)**sdslen**(**c**->**querybuf**)-**2**))**

**break;**

**if** **(**c**->**querybuf**[**pos**]** **!=** '$'**)** **{**

addReplyErrorFormat**(**c**,**

"Protocol error: expected '$', got '%c'"**,**

c**->**querybuf**[**pos**]);**

setProtocolError**(**c**,**pos**);**

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

**}**

ok **=** string2ll**(**c**->**querybuf**+**pos**+**1**,**newline**-(**c**->**querybuf**+**pos**+**1**),&**ll**);**

**if** **(!**ok **||** ll **<** 0 **||** ll **>** 512**\***1024**\***1024**)** **{**

addReplyError**(**c**,**"Protocol error: invalid bulk length"**);**

setProtocolError**(**c**,**pos**);**

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

**}**

pos **+=** newline**-(**c**->**querybuf**+**pos**)+**2**;**

**if** **(**ll **>=** REDIS\_MBULK\_BIG\_ARG**)** **{**

size\_t qblen**;**

*/\* If we are going to read a large object from network*

*\* try to make it likely that it will start at c->querybuf*

*\* boundary so that we can optimize object creation*

*\* avoiding a large copy of data. \*/*

sdsrange**(**c**->**querybuf**,**pos**,-**1**);**

pos **=** 0**;**

qblen **=** sdslen**(**c**->**querybuf**);**

*/\* Hint the sds library about the amount of bytes this string is*

*\* going to contain. \*/*

**if** **(**qblen **<** **(**size\_t**)**ll**+**2**)**

c**->**querybuf **=** sdsMakeRoomFor**(**c**->**querybuf**,**ll**+**2**-**qblen**);**

**}**

c**->**bulklen **=** ll**;**

**}**

*/\* Read bulk argument \*/*

**if** **(**sdslen**(**c**->**querybuf**)-**pos **<** **(**unsigned**)(**c**->**bulklen**+**2**))** **{**

*/\* Not enough data (+2 == trailing \r\n) \*/*

**break;**

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

*/\* Optimization: if the buffer contains JUST our bulk element*

*\* instead of creating a new object by \*copying\* the sds we*

*\* just use the current sds string. \*/*

**if** **(**pos **==** 0 **&&**

c**->**bulklen **>=** REDIS\_MBULK\_BIG\_ARG **&&**

**(**signed**)** sdslen**(**c**->**querybuf**)** **==** c**->**bulklen**+**2**)**

**{**

c**->**argv**[**c**->**argc**++]** **=** createObject**(**REDIS\_STRING**,**c**->**querybuf**);**

sdsIncrLen**(**c**->**querybuf**,-**2**);** */\* remove CRLF \*/*

c**->**querybuf **=** sdsempty**();**

*/\* Assume that if we saw a fat argument we'll see another one*

*\* likely... \*/*

c**->**querybuf **=** sdsMakeRoomFor**(**c**->**querybuf**,**c**->**bulklen**+**2**);**

pos **=** 0**;**

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

c**->**argv**[**c**->**argc**++]** **=**

createStringObject**(**c**->**querybuf**+**pos**,**c**->**bulklen**);**

pos **+=** c**->**bulklen**+**2**;**

**}**

c**->**bulklen **=** **-**1**;**

c**->**multibulklen**--;**

**}**

**}**

*/\* Trim to pos \*/*

**if** **(**pos**)** sdsrange**(**c**->**querybuf**,**pos**,-**1**);**

*/\* We're done when c->multibulk == 0 \*/*

**if** **(**c**->**multibulklen **==** 0**)** **return** REDIS\_OK**;**

*/\* Still not read to process the command \*/*

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

**}**

redisClient结构中的multibulklen属性，记录正在解析的一条完整的命令请求中，尚未处理的命令参数的个数。如果c->multibulklen为0，说明当前要解析的是命令请求的开头，格式为"\*<n>\r\n"。

这种情况下，首先找到c->querybuf中的第一个'\r'的位置newline，如果c->querybuf中找不到'\r'，说明收到的客户端的请求尚不完整，直接返回REDIS\_ERR。并且如果c->querybuf目前长度超过64k的话，则反馈给客户端错误信息："Protocol error: too big mbulk count string"，然后调用setProtocolError为客户端标志位c->flags增加REDIS\_CLOSE\_AFTER\_REPLY标记；直接返回REDIS\_ERR；

然后如果(newline-(c->querybuf))大于((signed)sdslen(c->querybuf)-2)，说明收到的客户端请求尚不完整（缺少'\n'），直接返回REDIS\_ERR；

接下来就开始解析该行，该行内容的正确格式是"\*<n>\r\n"，其中<n>是一个表明接下来包含多少个字符串的整数。调用string2ll解析得到其中的整数ll，如果解析失败，或者ll大于1M，则反馈给客户端信息"Protocol error: invalid multibulk length"，然后，调用setProtocolError为客户端标志位c->flags增加REDIS\_CLOSE\_AFTER\_REPLY标记，返回REDIS\_ERR；

然后使pos记为c->querybuf下一行首地址的索引；

如果ll小于等于0，则直接清除c->querybuf中刚刚解析的行，直接返回REDIS\_OK；然后将ll赋值到c->multibulklen中。然后根据c->multibulklen的值申请数组c->argv的空间，其数组长度就是c->multibulklen。

得到c->multibulklen的值后，接下来开始依次处理命令请求中的每一个字符串行：

redisClient结构中的bulklen属性，记录接下来要解析的命令请求行中，包含的字符串的长度。如果c->bulklen为-1，说明当前要解析的，是字符串的长度行，格式为"$<n>\r\n"。

这种情况下，处理过程与c->multibulklen为0时的解析过程类似，不在赘述。解析完后，下一行中包含的字符串长度存储在ll中，ll最大为512M，否则反馈给客户端错误信息："Protocol error: invalid bulk length"，并且调用setProtocolError为客户端标志位c->flags增加REDIS\_CLOSE\_AFTER\_REPLY标记，返回REDIS\_ERR；

然后使pos记为c->querybuf下一行首地址的索引；

如果字符串长度ll大于等于32k，为了后续创建字符串对象时避免复制大块内存，直接使用c->querybuf创建字符串对象。因此直接将c->querybuf中pos之前的内容删除，置pos为0，并且必要情况下为c->querybuf扩容。最后将ll赋值到c->bulklen中；

接下来开始解析c->querybuf中的字符串行，格式为"xxxx\r\n"；

如果(sdslen(c->querybuf)-pos)小于((unsigned)(c->bulklen+2))，说明收到的客户端请求中，字符串行尚不完整，直接退出循环，返回REDIS\_ERR；

否则，如果同时满足以下三个条件：

pos == 0；

c->bulklen >= REDIS\_MBULK\_BIG\_ARG；

(signed) sdslen(c->querybuf) == c->bulklen+2)；

说明，当前c->querybuf中，不多不少正好包含的是一个大于32k的大字符串行，这种情况下，为了避免拷贝大块内存，直接使用c->querybuf创建字符串对象，并存储到c->argv中；然后重新创建c->querybuf，并为其扩容为c->bulklen+2，这样可以容纳在后续遇到的大字符串（Assume that if we saw a fat argument we'll see another one likely...）；

如果不满足上面的条件，则创建字符串对象，将c->querybuf+pos的内容复制到该字符串对象中；

处理完一个完整的字符串行后，重置c->bulklen为-1，并且c->multibulklen--；然后循环处理下一个字符串行；

跳出循环后，首先删除已解析的内容，如果c->multibulklen为0，说明已经完整的收到并解析了客户端的一个跨多行的命令请求，返回REDIS\_OK，表示可以开始处理该命令了；否则，返回REDIS\_ERR，继续接收客户端请求；

processInlineBuffer函数的实现要简单很多，不再赘述。

**四：回复客户端**

服务器执行完相应的命令处理函数之后，就会调用addReply类的函数将要回复给客户端的信息写入客户端输出缓存。这些函数包括addReply，addReplySds，addReplyError，addReplyStatus等。

这些函数首先都会调用prepareClientToWrite函数，注册socket描述符上的可写事件，然后将回复信息写入到客户端输出缓存中。

redisClient结构中有两种客户端输出缓存，一种是静态大小的数组(buf)，一种是动态大小的列表(reply)。追加回复信息时，首先尝试将信息追加到数组buf中，如果其空间不足，则将信息在追加到reply中。比如addReplyString的代码如下：

void addReplyString**(**redisClient **\***c**,** char **\***s**,** size\_t len**)** **{**

**if** **(**prepareClientToWrite**(**c**)** **!=** REDIS\_OK**)** **return;**

**if** **(**\_addReplyToBuffer**(**c**,**s**,**len**)** **!=** REDIS\_OK**)**

\_addReplyStringToList**(**c**,**s**,**len**);**

**}**

调用函数\_addReplyToBuffer向c->buf中添加数据，如果该函数返回REDIS\_ERR，说明添加失败，则调用\_addReplyStringToList，将数据添加到c->reply中。其他addReply类的函数也是类似的处理，不再赘述。

每次向客户端输出缓存追加新数据之前，都要调用函数prepareClientToWrite。 因Redis中不同类型的客户端需要不同的处理：有些客户端（比如加载AOF文件时的伪客户端）无需追加新数据，这种情况下，该函数直接返回REDIS\_ERR；有些客户端（比如Lua客户端）需要追加新数据，但无需注册socket描述符上的可写事件；有些客户端（普通客户端）需要追加数据，并注册socket描述符上的可写事件；

因此，调用prepareClientToWrite函数返回REDIS\_ERR，则表示无需向输出缓存追加新数据，只有返回REDIS\_OK时才需要向输出缓存中追加新数据。

prepareClientToWrite函数的代码如下：

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

*/\* If it's the Lua client we always return ok without installing any*

*\* handler since there is no socket at all. \*/*

**if** **(**c**->**flags **&** REDIS\_LUA\_CLIENT**)** **return** REDIS\_OK**;**

*/\* Masters don't receive replies, unless REDIS\_MASTER\_FORCE\_REPLY flag*

*\* is set. \*/*

**if** **((**c**->**flags **&** REDIS\_MASTER**)** **&&**

**!(**c**->**flags **&** REDIS\_MASTER\_FORCE\_REPLY**))** **return** REDIS\_ERR**;**

**if** **(**c**->**fd **<=** 0**)** **return** REDIS\_ERR**;** */\* Fake client for AOF loading. \*/*

*/\* Only install the handler if not already installed and, in case of*

*\* slaves, if the client can actually receive writes. \*/*

**if** **(**c**->**bufpos **==** 0 **&&** listLength**(**c**->**reply**)** **==** 0 **&&**

**(**c**->**replstate **==** REDIS\_REPL\_NONE **||**

**(**c**->**replstate **==** REDIS\_REPL\_ONLINE **&&** **!**c**->**repl\_put\_online\_on\_ack**)))**

**{**

*/\* Try to install the write handler. \*/*

**if** **(**aeCreateFileEvent**(**server**.**el**,** c**->**fd**,** AE\_WRITABLE**,**

sendReplyToClient**,** c**)** **==** AE\_ERR**)**

**{**

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

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

**}**

**}**

*/\* Authorize the caller to queue in the output buffer of this client. \*/*

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

**}**

如果当前客户端是Lua客户端，直接返回REDIS\_OK，而无需注册socket描述符上的可写事件，因为根本没有socket描述符；

如果客户端为Master节点，除非设置REDIS\_MASTER\_FORCE\_REPLY标志，否则这种客户端不接收回复，因此直接返回REDIS\_ERR；

如果客户端的socket描述符小于等于0，说明是加载AOF文件时的伪客户端，直接返回REDIS\_ERR；

如果是普通客户端，或者是在从节点需要接收数据时，如果此前从未注册过socket上的可写事件，则调用aeCreateFileEvent注册socket描述符c->fd上的可写事件，事件回调函数为sendReplyToClient；最后直接返回REDIS\_OK；

当TCP输出缓冲区有一定剩余空间时，socket描述符上的可写事件就会触发，从而调用事件回调函数sendReplyToClient。该函数调用write，将输出缓存中的数据发送出去。函数的代码如下：

void sendReplyToClient**(**aeEventLoop **\***el**,** int fd**,** void **\***privdata**,** int mask**)** **{**

redisClient **\***c **=** privdata**;**

int nwritten **=** 0**,** totwritten **=** 0**,** objlen**;**

size\_t objmem**;**

robj **\***o**;**

REDIS\_NOTUSED**(**el**);**

REDIS\_NOTUSED**(**mask**);**

**while(**c**->**bufpos **>** 0 **||** listLength**(**c**->**reply**))** **{**

**if** **(**c**->**bufpos **>** 0**)** **{**

nwritten **=** write**(**fd**,**c**->**buf**+**c**->**sentlen**,**c**->**bufpos**-**c**->**sentlen**);**

**if** **(**nwritten **<=** 0**)** **break;**

c**->**sentlen **+=** nwritten**;**

totwritten **+=** nwritten**;**

*/\* If the buffer was sent, set bufpos to zero to continue with*

*\* the remainder of the reply. \*/*

**if** **(**c**->**sentlen **==** c**->**bufpos**)** **{**

c**->**bufpos **=** 0**;**

c**->**sentlen **=** 0**;**

**}**

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

o **=** listNodeValue**(**listFirst**(**c**->**reply**));**

objlen **=** sdslen**(**o**->**ptr**);**

objmem **=** getStringObjectSdsUsedMemory**(**o**);**

**if** **(**objlen **==** 0**)** **{**

listDelNode**(**c**->**reply**,**listFirst**(**c**->**reply**));**

c**->**reply\_bytes **-=** objmem**;**

**continue;**

**}**

nwritten **=** write**(**fd**,** **((**char**\*)**o**->**ptr**)+**c**->**sentlen**,**objlen**-**c**->**sentlen**);**

**if** **(**nwritten **<=** 0**)** **break;**

c**->**sentlen **+=** nwritten**;**

totwritten **+=** nwritten**;**

*/\* If we fully sent the object on head go to the next one \*/*

**if** **(**c**->**sentlen **==** objlen**)** **{**

listDelNode**(**c**->**reply**,**listFirst**(**c**->**reply**));**

c**->**sentlen **=** 0**;**

c**->**reply\_bytes **-=** objmem**;**

**}**

**}**

*/\* Note that we avoid to send more than REDIS\_MAX\_WRITE\_PER\_EVENT*

*\* bytes, in a single threaded server it's a good idea to serve*

*\* other clients as well, even if a very large request comes from*

*\* super fast link that is always able to accept data (in real world*

*\* scenario think about 'KEYS \*' against the loopback interface).*

*\**

*\* However if we are over the maxmemory limit we ignore that and*

*\* just deliver as much data as it is possible to deliver. \*/*

server**.**stat\_net\_output\_bytes **+=** totwritten**;**

**if** **(**totwritten **>** REDIS\_MAX\_WRITE\_PER\_EVENT **&&**

**(**server**.**maxmemory **==** 0 **||**

zmalloc\_used\_memory**()** **<** server**.**maxmemory**))** **break;**

**}**

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

**if** **(**errno **==** EAGAIN**)** **{**

nwritten **=** 0**;**

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

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

"Error writing to client: %s"**,** strerror**(**errno**));**

freeClient**(**c**);**

**return;**

**}**

**}**

**if** **(**totwritten **>** 0**)** **{**

*/\* For clients representing masters we don't count sending data*

*\* as an interaction, since we always send REPLCONF ACK commands*

*\* that take some time to just fill the socket output buffer.*

*\* We just rely on data / pings received for timeout detection. \*/*

**if** **(!(**c**->**flags **&** REDIS\_MASTER**))** c**->**lastinteraction **=** server**.**unixtime**;**

**}**

**if** **(**c**->**bufpos **==** 0 **&&** listLength**(**c**->**reply**)** **==** 0**)** **{**

c**->**sentlen **=** 0**;**

aeDeleteFileEvent**(**server**.**el**,**c**->**fd**,**AE\_WRITABLE**);**

*/\* Close connection after entire reply has been sent. \*/*

**if** **(**c**->**flags **&** REDIS\_CLOSE\_AFTER\_REPLY**)** freeClient**(**c**);**

**}**

**}**

当追加要发送的数据到输出缓存时，首先尝试将其添加到c->buf中；如果c->buf空间不足，则追加到c->reply中。如果使用的是c->buf，则c->bufpos表示其中缓存的数据总量，c->sentlen表示其中已发送的数据量；如果使用的是c->reply，则c->reply\_bytes表示列表c->reply中，保存的所有sds字符串占用的内存总字节数，c->sentlen表示列表中的正在发送数据的单块缓存元素中，已发送的数据量。

函数中的totwritten表示本函数当前已发送的数据量；

在函数中，如果c->bufpos大于0，或者listLength(c->reply)大于0，说明缓存中有数据要发送，进入循环，调用write发送数据，write返回值nwritten小于等于0时，要么是TCP输出缓存无空间，要么是发生了错误，因此直接跳出循环。

在循环中：如果c->bufpos大于0，说明使用的缓存是c->buf。因此调用write，将c->buf中的剩余数据（c->bufpos - c->sentlen个字节）发送出去。如果write返回值nwritten小于等于0时，直接跳出循环；否则，将nwritten增加到c->sentlen和totwritten中，继续下一轮循环写入。如果c->buf中的数据已全部发送出去，则重置c->bufpos和c->sentlen为0，表示清空缓存c->buf；

否则的话，表示使用的缓存是列表c->reply。得到其头结点中保存的字符串对象o，然后得到该字符串的长度objlen，以及该字符串占用的内存objmem。接着调用write，将o->ptr中未发送的数据（objlen - c->sentlen个字节）全部发送出去。如果write返回值nwritten小于等于0时，直接跳出循环；否则，将nwritten增加到c->sentlen和totwritten中，继续下一轮循环写入。如果c->sentlen等于objlen，说明当前节点的数据已经全部发送完成，直接删除该节点，并重置c->sentlen为0，并从c->reply\_bytes中减去objmem；

接下来，将本次已发送的字节数totwritten加到server.stat\_net\_output\_bytes中。

因本函数是可写事件的回调函数，为了避免该函数执行时间过长，而影响其他事件的处理。因此这里限制该函数最大发送的字节数为REDIS\_MAX\_WRITE\_PER\_EVENT(64k)，一旦已发送的字节数totwritten超过了该值，并且在没设置最大内存限制，或者尚未超过设置的最大内存限制的条件下，直接退出循环，停止发送。

退出循环后，如果write出错，并且errno为EAGAIN，说明TCP输出缓存无空间了，这种情况不是错误，直接置nwritten = 0即可；否则需要记录错误日志，并且调用freeClient释放redisClient，关闭与客户端的连接；

最后，如果缓存中所有的数据都已经发送完成，则置c->sentlen为0，并且删除socket描述符c->fd上的可写事件；如果客户端标志c->flags中设置了REDIS\_CLOSE\_AFTER\_REPLY，则调用freeClient释放redisClient，关闭与客户端的连接。

其他相关代码，可以参考：

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