Redis的sentinel模式使用了Hiredis代码，Hiredis是Redis数据库一个轻量级的C语言客户端库。它实现的向Redis发送命令的API函数redisCommand，使用方法类似于printf。因此只要熟悉redis命令，就可以很容易的使用该函数将redis命令字符串，转换成统一请求协议格式之后，发送给Redis服务器。

Hiredis库包含三类API：同步操作API、异步操作API和回复解析API。本文主要介绍同步操作API和回复解析API，下一篇介绍异步操作API。

**一：同步操作API**

所谓的同步操作，就是以阻塞的方式向Redis服务器建链，发送命令，接收命令回复。使用同步操作API，主要涉及以下三个API函数：

redisContext \*redisConnect(const char \*ip, int port);

void \*redisCommand(redisContext \*c, const char \*format, ...);

void freeReplyObject(void \*reply);

1：TCP建链

redisConnect函数创建一个上下文结构redisContext，并向Redis服务器发起TCP建链。该函数是同步建链API，因此该函数返回后，要么TCP已经建链成功了，要么建链期间发生了错误，可以通过检查redisContext结构的err和errstr属性得到错误类型和错误类型。

redisConnect的代码较简单，如下：

redisContext **\***redisConnect**(**const char **\***ip**,** int port**)** **{**

redisContext **\***c**;**

c **=** redisContextInit**();**

**if** **(**c **==** **NULL)**

**return** **NULL;**

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

redisContextConnectTcp**(**c**,**ip**,**port**,NULL);**

**return** c**;**

**}**

redisContext上下文结构用于保存所有与Redis服务器连接的状态。比如socket描述符，输出缓存，回复解析器等。该结构的定义如下：

**typedef** struct redisContext **{**

int err**;** */\* Error flags, 0 when there is no error \*/*

char errstr**[**128**];** */\* String representation of error when applicable \*/*

int fd**;**

int flags**;**

char **\***obuf**;** */\* Write buffer \*/*

redisReader **\***reader**;** */\* Protocol reader \*/*

**}** redisContext**;**

属性err为非0时，表示与Redis服务器的连接发生了错误，属性errstr就包含一个描述该错误的字符串。因此，每次与Redis进行交互之后，就需要检查该属性判断是否发生了错误，一旦有错误发生，则立即终止与Redis的链接。

属性fd就是与Redis服务器链接的socket描述符；flags表示客户端标志位，表示客户端当前的状态；

obuf就是输出缓存，当用户调用redisCommand向Redis发送命令时，命令字符串首先就是追加到该输出缓存中；

reader是一个回复解析器，后续在“回复解析API”中会详细介绍。

2：发送命令，接收回复

用户可以调用redisCommand函数向Redis服务器发送命令，该函数返回Redis的回复信息。该函数的原型如下：

void **\***redisCommand**(**redisContext **\***c**,** const char **\***format**,** **...)**

该函数类似于printf，支持不定参数，使用起来很方便，比如：

reply = redisCommand(context, "SET foo bar");

redisCommand函数返回NULL表示发生了错误，可以通过检查redisContext结构中的err得到错误类型；如果执行成功，则返回值是一个指向redisReply结构的指针，其中包含了Redis的回复信息。

可以在格式字符串中使用”%s”，表示在命令中插入一个字符串，此时使用strlen判断字符串的长度：

reply = redisCommand(context, "SET foo %s", value);

如果需要在命令中传递二进制安全的字符串，可以使用”%b”，此时需要一个size\_t类型的参数表示该字符串的长度：

reply = redisCommand(context, "SET foo %b", value, (size\_t) valuelen);

redisCommand主要是通过redisvCommand实现的，而redisvCommand主要是通过redisvAppendCommand和\_\_redisBlockForReply两个函数实现。它们的代码如下：

void **\***redisvCommand**(**redisContext **\***c**,** const char **\***format**,** va\_list ap**)** **{**

**if** **(**redisvAppendCommand**(**c**,**format**,**ap**)** **!=** REDIS\_OK**)**

**return** **NULL;**

**return** \_\_redisBlockForReply**(**c**);**

**}**

void **\***redisCommand**(**redisContext **\***c**,** const char **\***format**,** **...)** **{**

va\_list ap**;**

void **\***reply **=** **NULL;**

va\_start**(**ap**,**format**);**

reply **=** redisvCommand**(**c**,**format**,**ap**);**

va\_end**(**ap**);**

**return** reply**;**

**}**

redisvAppendCommand函数作用是解析用户的输入，并将用户输入的命令字符串转换成Redis统一请求协议的格式，存储到上下文结构redisContext中的输出缓存obuf中，它的代码如下：

int redisvAppendCommand**(**redisContext **\***c**,** const char **\***format**,** va\_list ap**)** **{**

char **\***cmd**;**

int len**;**

len **=** redisvFormatCommand**(&**cmd**,**format**,**ap**);**

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

\_\_redisSetError**(**c**,**REDIS\_ERR\_OOM**,**"Out of memory"**);**

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

**}**

**if** **(**\_\_redisAppendCommand**(**c**,**cmd**,**len**)** **!=** REDIS\_OK**)** **{**

free**(**cmd**);**

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

**}**

free**(**cmd**);**

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

**}**

int \_\_redisAppendCommand**(**redisContext **\***c**,** const char **\***cmd**,** size\_t len**)** **{**

sds newbuf**;**

newbuf **=** sdscatlen**(**c**->**obuf**,**cmd**,**len**);**

**if** **(**newbuf **==** **NULL)** **{**

\_\_redisSetError**(**c**,**REDIS\_ERR\_OOM**,**"Out of memory"**);**

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

**}**

c**->**obuf **=** newbuf**;**

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

**}**

redisvAppendCommand 首先调用redisvFormatCommand函数，用于解析用户输入的命令字符串，并将字符串转换成协议格式之后，保存在cmd中。然后调用\_\_redisAppendCommand函数，将cmd追加到输出缓存c->obuf中。代码较简单，不在赘述。

在redisvCommand函数中，调用redisvAppendCommand之后，接下来就是调用\_\_redisBlockForReply函数，将输出缓存中的内容发送给Redis服务器，并读取Redis的回复，并解析之。

\_\_redisBlockForReply函数主要是通过redisGetReply实现的，它们的代码如下：

static void **\***\_\_redisBlockForReply**(**redisContext **\***c**)** **{**

void **\***reply**;**

**if** **(**c**->**flags **&** REDIS\_BLOCK**)** **{**

**if** **(**redisGetReply**(**c**,&**reply**)** **!=** REDIS\_OK**)**

**return** **NULL;**

**return** reply**;**

**}**

**return** **NULL;**

**}**

int redisGetReply**(**redisContext **\***c**,** void **\*\***reply**)** **{**

int wdone **=** 0**;**

void **\***aux **=** **NULL;**

*/\* Try to read pending replies \*/*

**if** **(**redisGetReplyFromReader**(**c**,&**aux**)** **==** REDIS\_ERR**)**

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

*/\* For the blocking context, flush output buffer and read reply \*/*

**if** **(**aux **==** **NULL** **&&** c**->**flags **&** REDIS\_BLOCK**)** **{**

*/\* Write until done \*/*

**do** **{**

**if** **(**redisBufferWrite**(**c**,&**wdone**)** **==** REDIS\_ERR**)**

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

**}** **while** **(!**wdone**);**

*/\* Read until there is a reply \*/*

**do** **{**

**if** **(**redisBufferRead**(**c**)** **==** REDIS\_ERR**)**

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

**if** **(**redisGetReplyFromReader**(**c**,&**aux**)** **==** REDIS\_ERR**)**

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

**}** **while** **(**aux **==** **NULL);**

**}**

*/\* Set reply object \*/*

**if** **(**reply **!=** **NULL)** **\***reply **=** aux**;**

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

**}**

在redisGetReply中，首先是循环调用redisBufferWrite，将输出缓存c->obuf中的所有内容发送给Redis。然后循环调用redisBufferRead，读取Redis的回复，并调用函数redisGetReplyFromReader对回复信息进行解析。

redisBufferRead函数的代码如下：

int redisBufferRead**(**redisContext **\***c**)** **{**

char buf**[**1024**\***16**];**

int nread**;**

*/\* Return early when the context has seen an error. \*/*

**if** **(**c**->**err**)**

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

nread **=** read**(**c**->**fd**,**buf**,sizeof(**buf**));**

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

**if** **((**errno **==** EAGAIN **&&** **!(**c**->**flags **&** REDIS\_BLOCK**))** **||** **(**errno **==** EINTR**))** **{**

*/\* Try again later \*/*

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

\_\_redisSetError**(**c**,**REDIS\_ERR\_IO**,NULL);**

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

**}**

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

\_\_redisSetError**(**c**,**REDIS\_ERR\_EOF**,**"Server closed the connection"**);**

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

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

**if** **(**redisReaderFeed**(**c**->**reader**,**buf**,**nread**)** **!=** REDIS\_OK**)** **{**

\_\_redisSetError**(**c**,**c**->**reader**->**err**,**c**->**reader**->**errstr**);**

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

**}**

**}**

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

**}**

该函数主要是从socket中读取数据到buf中，然后通过函数redisReaderFeed，将buf内容追加到解析器的输入缓存中。redisReaderFeed函数属于回复解析API函数。

**二：回复解析API**

回复解析API的函数主要有下面几个：

redisReader **\***redisReaderCreate**(**void**);**

void redisReaderFree**(**redisReader **\***reader**);**

int redisReaderFeed**(**redisReader **\***reader**,** const char **\***buf**,** size\_t len**);**

int redisReaderGetReply**(**redisReader **\***reader**,** void **\*\***reply**);**

1：输入缓存

解析器结构redisReader，是回复解析API最主要的数据结构。它的部分定义如下：

*/\* State for the protocol parser \*/*

**typedef** struct redisReader **{**

int err**;** */\* Error flags, 0 when there is no error \*/*

char errstr**[**128**];** */\* String representation of error when applicable \*/*

char **\***buf**;** */\* Read buffer \*/*

size\_t pos**;** */\* Buffer cursor \*/*

size\_t len**;** */\* Buffer length \*/*

size\_t maxbuf**;** */\* Max length of unused buffer \*/*

...

**}** redisReader**;**

其中的err和errstr属性与redisContext结构中的err和errstr属性作用一样，都是用于保存错误类型和错误信息的；

buf属性就是输入缓存，redisReaderFeed函数将读取到的Redis回复信息都存储到该缓存中，该缓存根据回复信息可以动态扩容。len表示当前缓存的容量；pos表示缓存当前的读取索引，每次读取输入缓存时，都是从reader->buf + reader->pos处开始读取，读取数据之后，会增加pos的值；

maxbuf属性表示输入缓存所允许的最大闲置空间。为了节省内存空间，当buf为空，并且当前buf的闲置空间大于reader->maxbuf时，就会释放r->buf，重新为其申请空间。该属性的默认值为16K。如果置为0，表示无此限制。

redisReaderFeed就是将从socket读取的Redis回复信息，追加到输入缓存的函数。其代码如下：

int redisReaderFeed**(**redisReader **\***r**,** const char **\***buf**,** size\_t len**)** **{**

sds newbuf**;**

*/\* Return early when this reader is in an erroneous state. \*/*

**if** **(**r**->**err**)**

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

*/\* Copy the provided buffer. \*/*

**if** **(**buf **!=** **NULL** **&&** len **>=** 1**)** **{**

*/\* Destroy internal buffer when it is empty and is quite large. \*/*

**if** **(**r**->**len **==** 0 **&&** r**->**maxbuf **!=** 0 **&&** sdsavail**(**r**->**buf**)** **>** r**->**maxbuf**)** **{**

sdsfree**(**r**->**buf**);**

r**->**buf **=** sdsempty**();**

r**->**pos **=** 0**;**

*/\* r->buf should not be NULL since we just free'd a larger one. \*/*

assert**(**r**->**buf **!=** **NULL);**

**}**

newbuf **=** sdscatlen**(**r**->**buf**,**buf**,**len**);**

**if** **(**newbuf **==** **NULL)** **{**

\_\_redisReaderSetErrorOOM**(**r**);**

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

**}**

r**->**buf **=** newbuf**;**

r**->**len **=** sdslen**(**r**->**buf**);**

**}**

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

**}**

2：解析

在redisGetReply函数中，将Redis的回复信息追加到解析器输入缓存之后，接下来就会调用函数redisGetReplyFromReader对解析器的输入缓存中的消息进行解析，解析的内容以redisReply结构进行组织。

如果回复信息是嵌套的话，则形成一颗以redisReply结构为节点的多叉树；如果回复信息只是基本信息的话，则该树仅仅包含一个根节点。redisCommand函数就是返回一个指向redisReply结构树根节点的指针。redisReply结构树的宽度没有限制，但是深度的最大值为7，也就是仅允许最多7层嵌套。

首先看一下redisReply结构的定义如下：

/\* This is the reply object returned by redisCommand() \*/

**typedef** struct redisReply **{**

int type**;**

long long integer**;**

int len**;**

char **\***str**;**

size\_t elements**;**

struct redisReply **\*\***element**;**

**}** redisReply**;**

该结构中的type成员表示Redis回复信息的类型，可以有下面几种类型：

REDIS\_REPLY\_STATUS：状态回复，状态信息以'+'开头。str属性保存Redis回复的状态信息字符串，该字符串的长度保存在len属性中。

REDIS\_REPLY\_ERROR：错误回复，错误信息以'-'开头。str属性保存Redis回复的错误信息字符串，该字符串的长度保存在len属性中。

REDIS\_REPLY\_INTEGER：整数回复，整数信息以':'开头。integer 属性保存Redis回复的整数值。

REDIS\_REPLY\_STRING：单行字符串回复，这种信息以'$'开头。str属性保存Redis回复的字符串信息，该字符串的长度保存在len属性中。

REDIS\_REPLY\_NIL：Redis回复”nil”。

以上的类型可以称为基本类型。

REDIS\_REPLY\_ARRAY：数组回复，也就是嵌套回复，数组信息以'\*'开头，后接数组元素个数。数组中的元素可以是以上所有基本类型，也可以是REDIS\_REPLY\_ARRAY类型，也就是数组嵌套数组。

数组元素的个数保存在elements属性中，数组元素也以redisReply结构表示，指向数组元素的指针保存在element指针数组中，也就是说，指针数组element中保存了所有孩子节点的指针。

经过回复解析API函数redisReaderGetReply的解析之后，最终形成的redisReply结构树，非叶子节点只能是REDIS\_REPLY\_ARRAY类型，叶子节点的类型只能是基本类型。

处理数组信息的代码较复杂，以一个例子说明。假设Redis的回复信息是："\*3\r\n\*3\r\n:11\r\n:12\r\n:13\r\n\*3\r\n:21\r\n:22\r\n:23\r\n:31\r\n"。

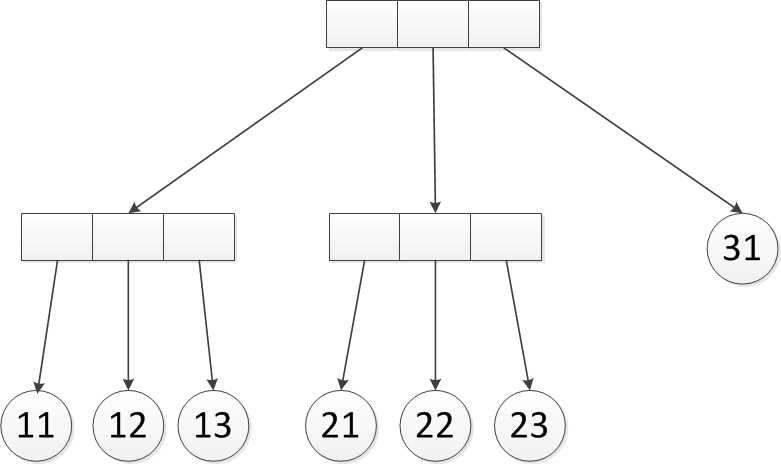
分析该字符串，第一个字符为”\*”，表明这是一条数组回复，后面的3表示数组元素的个数，因此，最终形成的树，根节点有3个孩子节点。

接下来就是根节点各个孩子节点的信息。第一个孩子节点首字符还是”\*”，说明该孩子节点又是一个数组信息，它也有3个孩子。接下来就是3个孩子信息，也就是3个整数：11，12和13。

接下来是根节点第二个孩子节点的信息。首字符还是”\*”，说明该孩子节点也是一个数组，它也有3个孩子，分别是整数：21，22和23。

接下来是根节点最后一个孩子的信息，首字符是”:”，说明该孩子节点是一个整数，整数值为31。

根据以上的分析，最终形成的树如下图：



上图中，非叶子节点都是REDIS\_REPLY\_ARRAY类型的redisReply结构，叶子节点是REDIS\_REPLY\_INTEGER类型的redisReply结构。

回复解析API函数redisReaderGetReply的作用，就是解析回复信息，最终形成一颗这样的redisReply结构树。

在回复解析API的代码中，使用redisReadTask任务结构解析回复信息，构建每个redisReply结构节点，填充到树中合适的位置。

redisReadTask结构包含解析器结构redisReader中，redisReader结构剩下的定义如下：

**typedef** struct redisReader **{**

...

redisReadTask rstack**[**9**];**

int ridx**;** */\* Index of current read task \*/*

void **\***reply**;** */\* Temporary reply pointer \*/*

redisReplyObjectFunctions **\***fn**;**

void **\***privdata**;**

**}** redisReader**;**

在redisReader结构中，redisReadTask结构数组rstack大小为9。rstack[0]用于处理redisReply结构树中的根节点；rstack[1]表示处理redisReply结构树中的第一层子节点，以此类推。

ridx属性表示当前正在处理第几层子节点；fn属性是一个redisReplyObjectFunctions结构体，该结构中包含了用于生成各种类型redisReply结构的函数；

reply属性指向redisReply结构树中的根节点

构建每个redisReply结构节点的redisReadTask结构定义如下：

**typedef** struct redisReadTask **{**

int type**;**

int elements**;** */\* number of elements in multibulk container \*/*

int idx**;** */\* index in parent (array) object \*/*

void **\***obj**;** */\* holds user-generated value for a read task \*/*

struct redisReadTask **\***parent**;** */\* parent task \*/*

void **\***privdata**;** */\* user-settable arbitrary field \*/*

**}** redisReadTask**;**

type表示该redisReadTask结构当前处理的信息类型，与其当前构建的redisReply结构节点中的type一致；

elements表示当前构建的REDIS\_REPLY\_ARRAY 类型的redisReply结构节点中，包含的子节点数目。也就是redisReply结构节点中，数组element中的元素个数；idx表示当前构建的redisReply结构节点，在其父节点element数组中的索引；obj就是指向当前正在构建的REDIS\_REPLY\_ARRAY 类型的redisReply结构节点，parent表示正在处理当前节点的父节点的redisReadTask结构。

回复解析API函数redisReaderGetReply的代码如下：

int redisReaderGetReply**(**redisReader **\***r**,** void **\*\***reply**)** **{**

*/\* Default target pointer to NULL. \*/*

**if** **(**reply **!=** **NULL)**

**\***reply **=** **NULL;**

*/\* Return early when this reader is in an erroneous state. \*/*

**if** **(**r**->**err**)**

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

*/\* When the buffer is empty, there will never be a reply. \*/*

**if** **(**r**->**len **==** 0**)**

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

*/\* Set first item to process when the stack is empty. \*/*

**if** **(**r**->**ridx **==** **-**1**)** **{**

r**->**rstack**[**0**].**type **=** **-**1**;**

r**->**rstack**[**0**].**elements **=** **-**1**;**

r**->**rstack**[**0**].**idx **=** **-**1**;**

r**->**rstack**[**0**].**obj **=** **NULL;**

r**->**rstack**[**0**].**parent **=** **NULL;**

r**->**rstack**[**0**].**privdata **=** r**->**privdata**;**

r**->**ridx **=** 0**;**

**}**

*/\* Process items in reply. \*/*

**while** **(**r**->**ridx **>=** 0**)**

**if** **(**processItem**(**r**)** **!=** REDIS\_OK**)**

**break;**

*/\* Return ASAP when an error occurred. \*/*

**if** **(**r**->**err**)**

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

*/\* Discard part of the buffer when we've consumed at least 1k, to avoid*

*\* doing unnecessary calls to memmove() in sds.c. \*/*

**if** **(**r**->**pos **>=** 1024**)** **{**

sdsrange**(**r**->**buf**,**r**->**pos**,-**1**);**

r**->**pos **=** 0**;**

r**->**len **=** sdslen**(**r**->**buf**);**

**}**

*/\* Emit a reply when there is one. \*/*

**if** **(**r**->**ridx **==** **-**1**)** **{**

**if** **(**reply **!=** **NULL)**

**\***reply **=** r**->**reply**;**

r**->**reply **=** **NULL;**

**}**

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

**}**

首先，将r->ridx置为0，然后初始化r->rstack[0]，表示接下来开始构建根节点。

接下来的语句，就是循环调用processItem函数，直到r->ridx再次等于-1。循环调用processItem函数的过程，就是以深度优先的顺序构建redisReply结构树的过程。

processItem函数的代码如下：

static int processItem**(**redisReader **\***r**)** **{**

redisReadTask **\***cur **=** **&(**r**->**rstack**[**r**->**ridx**]);**

char **\***p**;**

*/\* check if we need to read type \*/*

**if** **(**cur**->**type **<** 0**)** **{**

**if** **((**p **=** readBytes**(**r**,**1**))** **!=** **NULL)** **{**

**switch** **(**p**[**0**])** **{**

**case** '-'**:**

cur**->**type **=** REDIS\_REPLY\_ERROR**;**

**break;**

**case** '+'**:**

cur**->**type **=** REDIS\_REPLY\_STATUS**;**

**break;**

**case** ':'**:**

cur**->**type **=** REDIS\_REPLY\_INTEGER**;**

**break;**

**case** '$'**:**

cur**->**type **=** REDIS\_REPLY\_STRING**;**

**break;**

**case** '\*'**:**

cur**->**type **=** REDIS\_REPLY\_ARRAY**;**

**break;**

**default:**

\_\_redisReaderSetErrorProtocolByte**(**r**,\***p**);**

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

**}**

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

*/\* could not consume 1 byte \*/*

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

**}**

**}**

*/\* process typed item \*/*

**switch(**cur**->**type**)** **{**

**case** REDIS\_REPLY\_ERROR**:**

**case** REDIS\_REPLY\_STATUS**:**

**case** REDIS\_REPLY\_INTEGER**:**

**return** processLineItem**(**r**);**

**case** REDIS\_REPLY\_STRING**:**

**return** processBulkItem**(**r**);**

**case** REDIS\_REPLY\_ARRAY**:**

**return** processMultiBulkItem**(**r**);**

**default:**

assert**(NULL);**

**return** REDIS\_ERR**;** */\* Avoid warning. \*/*

**}**

**}**

首先得到构建当前节点的redisReadTask结构cur，然后从输入缓存中读取首个字符，以判断接下来的回复信息的类型，赋值到cur->type中。

得到类型信息之后，就调用不同的函数处理不同的类型。首先看一下处理数组类型的函数processMultiBulkItem的实现：

static int processMultiBulkItem**(**redisReader **\***r**)** **{**

redisReadTask **\***cur **=** **&(**r**->**rstack**[**r**->**ridx**]);**

void **\***obj**;**

char **\***p**;**

long elements**;**

int root **=** 0**;**

*/\* Set error for nested multi bulks with depth > 7 \*/*

**if** **(**r**->**ridx **==** 8**)** **{**

\_\_redisReaderSetError**(**r**,**REDIS\_ERR\_PROTOCOL**,**

"No support for nested multi bulk replies with depth > 7"**);**

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

**}**

**if** **((**p **=** readLine**(**r**,NULL))** **!=** **NULL)** **{**

elements **=** readLongLong**(**p**);**

root **=** **(**r**->**ridx **==** 0**);**

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

**if** **(**r**->**fn **&&** r**->**fn**->**createNil**)**

obj **=** r**->**fn**->**createNil**(**cur**);**

**else**

obj **=** **(**void**\*)**REDIS\_REPLY\_NIL**;**

**if** **(**obj **==** **NULL)** **{**

\_\_redisReaderSetErrorOOM**(**r**);**

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

**}**

moveToNextTask**(**r**);**

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

**if** **(**r**->**fn **&&** r**->**fn**->**createArray**)**

obj **=** r**->**fn**->**createArray**(**cur**,**elements**);**

**else**

obj **=** **(**void**\*)**REDIS\_REPLY\_ARRAY**;**

**if** **(**obj **==** **NULL)** **{**

\_\_redisReaderSetErrorOOM**(**r**);**

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

**}**

*/\* Modify task stack when there are more than 0 elements. \*/*

**if** **(**elements **>** 0**)** **{**

cur**->**elements **=** elements**;**

cur**->**obj **=** obj**;**

r**->**ridx**++;**

r**->**rstack**[**r**->**ridx**].**type **=** **-**1**;**

r**->**rstack**[**r**->**ridx**].**elements **=** **-**1**;**

r**->**rstack**[**r**->**ridx**].**idx **=** 0**;**

r**->**rstack**[**r**->**ridx**].**obj **=** **NULL;**

r**->**rstack**[**r**->**ridx**].**parent **=** cur**;**

r**->**rstack**[**r**->**ridx**].**privdata **=** r**->**privdata**;**

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

moveToNextTask**(**r**);**

**}**

**}**

*/\* Set reply if this is the root object. \*/*

**if** **(**root**)** r**->**reply **=** obj**;**

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

**}**

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

**}**

首先得到构建当前节点的redisReadTask结构cur，然后调用readLine函数，从输入缓存中读取一行信息（”\r\n”之前的内容），并解析出当前节点中包含的元素个数elements。

如果elements不是-1，说明正确解析到了数组元素个数，接下来调用r->fn->createArray创建一个数组类型的redisReply结构节点。然后将创建的redisReply结构信息记录到cur中：将元素个数记录到cur->elements中，将创建的redisReply记录到cur->obj中：

cur**->**elements **=** elements**;**

cur**->**obj **=** obj**;**

数组类型的redisReply结构节点创建完成后，接下来就是开始构建其各个子节点。首先就是将r->ridx加1，并初始化r->rstack[r->ridx]结构，注意这里置r->rstack[r->ridx].idx为0.表示接下来首先构建第一个子节点。

下面是创建数组类型redisReply结构的函数createArrayObject的代码：

static void **\***createArrayObject**(**const redisReadTask **\***task**,** int elements**)** **{**

redisReply **\***r**,** **\***parent**;**

r **=** createReplyObject**(**REDIS\_REPLY\_ARRAY**);**

**if** **(**r **==** **NULL)**

**return** **NULL;**

**if** **(**elements **>** 0**)** **{**

r**->**element **=** calloc**(**elements**,sizeof(**redisReply**\*));**

**if** **(**r**->**element **==** **NULL)** **{**

freeReplyObject**(**r**);**

**return** **NULL;**

**}**

**}**

r**->**elements **=** elements**;**

**if** **(**task**->**parent**)** **{**

parent **=** task**->**parent**->**obj**;**

assert**(**parent**->**type **==** REDIS\_REPLY\_ARRAY**);**

parent**->**element**[**task**->**idx**]** **=** r**;**

**}**

**return** r**;**

**}**

该函数中，重点是要理解下面的代码：

**if** **(**task**->**parent**)** **{**

parent **=** task**->**parent**->**obj**;**

assert**(**parent**->**type **==** REDIS\_REPLY\_ARRAY**);**

parent**->**element**[**task**->**idx**]** **=** r**;**

**}**

这段代码的作用，就是将指向新创建的redisReply结构节点的指针r，存放到其父节点的element数组中，存放索引就是task->idx。

如果task->parent不为NULL，说明当前新建的redisReply结构节点具有父节点，根据当前task得到该父节点redisReply结构parent。然后将当前节点放到存储到父节点的element数组中的task-idx索引处。

接下来是moveToNextTask函数的实现，该函数的主要作用，实际上是变更属性r->ridx和cur->idx。说白了，就是为下一个要创建的节点，找到合适的位置。代码如下：

static void moveToNextTask**(**redisReader **\***r**)** **{**

redisReadTask **\***cur**,** **\***prv**;**

**while** **(**r**->**ridx **>=** 0**)** **{**

*/\* Return a.s.a.p. when the stack is now empty. \*/*

**if** **(**r**->**ridx **==** 0**)** **{**

r**->**ridx**--;**

**return;**

**}**

cur **=** **&(**r**->**rstack**[**r**->**ridx**]);**

prv **=** **&(**r**->**rstack**[**r**->**ridx**-**1**]);**

assert**(**prv**->**type **==** REDIS\_REPLY\_ARRAY**);**

**if** **(**cur**->**idx **==** prv**->**elements**-**1**)** **{**

r**->**ridx**--;**

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

*/\* Reset the type because the next item can be anything \*/*

assert**(**cur**->**idx **<** prv**->**elements**);**

cur**->**type **=** **-**1**;**

cur**->**elements **=** **-**1**;**

cur**->**idx**++;**

**return;**

**}**

**}**

**}**

在while循环中，首先得到处理当前节点的redisReadTask结构cur，然后是正处理该节点父节点的redisReadTask结构prv。

cur->idx记录了当前处理的节点，其在父节点中的element数组中的索引，也就是当前节点是父节点的第几个孩子。

prv->elements表示当前节点的父节点，共有几个孩子。

因此，如果cur->idx小于prv->elements的话，则接下来，cur结构要开始构建当前节点的下一个兄弟节点了，因此将cur->idx加1。

如果cur->idx等于prv->elements的话，说明当前节点，已经是其父节点最后一个孩子节点了。接下来，就开始构建当前节点的叔叔结点了（父节点的兄弟节点），因此r->ridx--，表示回溯。上移一层，将父结点变成当前节点，然后接着判断新的cur点在其父节点中是否是最后一个孩子，若是，则接着回溯，否则开始构建其兄弟节点。

如果当前节点已经是根节点了（r->ridx == 0），因为根节点没有兄弟节点，因此将r->ridx置为-1后，直接返回。

构建好一颗redisReply结构树之后，如果需要释放它，可以通过API函数freeReplyObject实现，代码如下：

void freeReplyObject**(**void **\***reply**)** **{**

redisReply **\***r **=** reply**;**

size\_t j**;**

**switch(**r**->**type**)** **{**

**case** REDIS\_REPLY\_INTEGER**:**

**break;** */\* Nothing to free \*/*

**case** REDIS\_REPLY\_ARRAY**:**

**if** **(**r**->**element **!=** **NULL)** **{**

**for** **(**j **=** 0**;** j **<** r**->**elements**;** j**++)**

**if** **(**r**->**element**[**j**]** **!=** **NULL)**

freeReplyObject**(**r**->**element**[**j**]);**

free**(**r**->**element**);**

**}**

**break;**

**case** REDIS\_REPLY\_ERROR**:**

**case** REDIS\_REPLY\_STATUS**:**

**case** REDIS\_REPLY\_STRING**:**

**if** **(**r**->**str **!=** **NULL)**

free**(**r**->**str**);**

**break;**

**}**

free**(**r**);**

**}**

标准的深度优先顺序进行释放。不再赘述。

以上就是回复解析API的主要工作流程。构建redisReply结构树和redisReadTask结构的作用比较晦涩，但是却是一个很好的构建多叉树的例子。学习代码时，脑子中跟着代码逐步建立这颗树就好理解了。

参考：

http://blog.csdn.net/it\_small\_farmer/article/details/41726293