Hiredis中的异步API函数需要与事件库（libevent, libev, ev）一起工作。因为事件循环的机制，异步环境中的命令是自动管道化的。因为命令是异步发送的，因此发送命令时，必要情况下，需要提供一个回调函数，以便在收到命令回复时调用该函数。

异步API涉及到的函数分别是：

redisAsyncContext **\***redisAsyncConnect**(**const char **\***ip**,** int port**);**

int redisAsyncCommand**(**redisAsyncContext **\***ac**,** redisCallbackFn **\***fn**,** void **\***privdata**,** const char **\***format**,** **...);**

int redisAsyncCommandArgv**(**redisAsyncContext **\***ac**,** redisCallbackFn **\***fn**,** void **\***privdata**,** int argc**,** const char **\*\***argv**,** const size\_t **\***argvlen**);**

void redisAsyncDisconnect**(**redisAsyncContext **\***ac**);**

以上函数分别对应TCP建链、发送命令和TCP断链。

**一：异步上下文**

类似于同步操作API，异步操作API中也有一个上下文结构redisAsyncContext，用于维护异步链接中的各种状态。

redisAsyncContext结构在同步上下文结构redisContext的基础上，增加了一些异步属性，它的定义如下：

**typedef** struct redisAsyncContext **{**

*/\* Hold the regular context, so it can be realloc'ed. \*/*

redisContext c**;**

*/\* Setup error flags so they can be used directly. \*/*

int err**;**

char **\***errstr**;**

*/\* Not used by hiredis \*/*

void **\***data**;**

*/\* Event library data and hooks \*/*

struct **{**

void **\***data**;**

*/\* Hooks that are called when the library expects to start*

*\* reading/writing. These functions should be idempotent. \*/*

void **(\***addRead**)(**void **\***privdata**);**

void **(\***delRead**)(**void **\***privdata**);**

void **(\***addWrite**)(**void **\***privdata**);**

void **(\***delWrite**)(**void **\***privdata**);**

void **(\***cleanup**)(**void **\***privdata**);**

**}** ev**;**

*/\* Called when either the connection is terminated due to an error or per*

*\* user request. The status is set accordingly (REDIS\_OK, REDIS\_ERR). \*/*

redisDisconnectCallback **\***onDisconnect**;**

*/\* Called when the first write event was received. \*/*

redisConnectCallback **\***onConnect**;**

*/\* Regular command callbacks \*/*

redisCallbackList replies**;**

*/\* Subscription callbacks \*/*

struct **{**

redisCallbackList invalid**;**

struct dict **\***channels**;**

struct dict **\***patterns**;**

**}** sub**;**

**}** redisAsyncContext**;**

该结构的第一个属性就是同步上下文结构redisContext c，剩下的就是一些异步属性：

结构体ev中包含了，当Hiredis异步API与事件库（libev, libevent, Redis ev）一起工作时，用于注册和删除读写事件的函数；

回调函数onDisconnect，表示断链时会调用的函数，该属性可以通过redisAsyncSetDisconnectCallback函数设置；

回调函数onConnect，表示TCP建链成功或失败之后会调用的函数，该属性可以通过redisAsyncSetConnectCallback函数设置；

replies属性是一个redisCallbackList结构，也就是由回调结构redisCallback组成的单链表。当发送普通命令时，会依次将该命令对应的回调结构追加到链表中，当Redis服务器回复普通命令时，会依次调用该链表中的每个redisCallback结构中的回调函数；

结构体sub用于处理订阅模式，其中的字典channels，以频道名为key，以回调结构redisCallback为value。当客户端使用Hiredis异步API发送”subscribe”命令后，服务器产生回复时，就会根据回复信息中的频道名查询字典channels，找到对应的回调结构，调用其中的回调函数。字典patterns与channels类似，只不过它用于”psubscirbe”命令，其中的key是频道名模式；回调链表invalid，当客户端处于订阅模式下，服务器发来了意想不到的回复时，会依次调用该链表中，每个回调结构中的回调函数。

**二：建链**

函数redisAsyncConnect执行异步操作中的TCP建链。

redisAsyncContext **\***redisAsyncConnect**(**const char **\***ip**,** int port**)** **{**

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

redisAsyncContext **\***ac**;**

c **=** redisConnectNonBlock**(**ip**,**port**);**

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

**return** **NULL;**

ac **=** redisAsyncInitialize**(**c**);**

**if** **(**ac **==** **NULL)** **{**

redisFree**(**c**);**

**return** **NULL;**

**}**

\_\_redisAsyncCopyError**(**ac**);**

**return** ac**;**

**}**

该函数首先根据ip和port，调用redisConnectNonBlock函数向Redis服务器发起非阻塞的建链操作，然后调用redisAsyncInitialize函数创建异步上下文结构redisAsyncContext。

redisAsyncSetConnectCallback函数用于设置异步上下文中的建链回调函数。其代码如下：

int redisAsyncSetConnectCallback**(**redisAsyncContext **\***ac**,** redisConnectCallback **\***fn**)** **{**

**if** **(**ac**->**onConnect **==** **NULL)** **{**

ac**->**onConnect **=** fn**;**

*/\* The common way to detect an established connection is to wait for*

*\* the first write event to be fired. This assumes the related event*

*\* library functions are already set. \*/*

\_EL\_ADD\_WRITE**(**ac**);**

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

**}**

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

**}**

该函数中，如果之前已经设置过建链回调函数了，则直接返回REDIS\_ERR。

该函数除了设置异步上下文中的建链回调函数之外，还会调用\_EL\_ADD\_WRITE，注册可写事件。对于使用Redis的ae事件库的客户端来说，该宏定义实际上就是调用redisAeAddWrite函数：

static void redisAeAddWrite**(**void **\***privdata**)** **{**

redisAeEvents **\***e **=** **(**redisAeEvents**\*)**privdata**;**

aeEventLoop **\***loop **=** e**->**loop**;**

**if** **(!**e**->**writing**)** **{**

e**->**writing **=** 1**;**

aeCreateFileEvent**(**loop**,**e**->**fd**,**AE\_WRITABLE**,**redisAeWriteEvent**,**e**);**

**}**

**}**

可写事件的回调函数是redisAeWriteEvent，该函数调用redisAsyncHandleWrite实现。redisAsyncHandleWrite中，处理建链的代码如下：

void redisAsyncHandleWrite**(**redisAsyncContext **\***ac**)** **{**

redisContext **\***c **=** **&(**ac**->**c**);**

int done **=** 0**;**

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

*/\* Abort connect was not successful. \*/*

**if** **(**\_\_redisAsyncHandleConnect**(**ac**)** **!=** REDIS\_OK**)**

**return;**

*/\* Try again later when the context is still not connected. \*/*

**if** **(!(**c**->**flags **&** REDIS\_CONNECTED**))**

**return;**

**}**

**. . .**

**}**

在该函数中，如果上下文标志位中还没有设置REDIS\_CONNECTED标记，说明目前还没有检测是否建链成功，因此调用\_\_redisAsyncHandleConnect，判断建链是否成功，如果建链成功，则会在异步上下文的标志位中增加REDIS\_CONNECTED标记，如果还没有建链成功，则直接返回。

\_\_redisAsyncHandleConnect的代码如下：

static int \_\_redisAsyncHandleConnect**(**redisAsyncContext **\***ac**)** **{**

redisContext **\***c **=** **&(**ac**->**c**);**

**if** **(**redisCheckSocketError**(**c**)** **==** REDIS\_ERR**)** **{**

*/\* Try again later when connect(2) is still in progress. \*/*

**if** **(**errno **==** EINPROGRESS**)**

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

**if** **(**ac**->**onConnect**)** ac**->**onConnect**(**ac**,**REDIS\_ERR**);**

\_\_redisAsyncDisconnect**(**ac**);**

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

**}**

*/\* Mark context as connected. \*/*

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

**if** **(**ac**->**onConnect**)** ac**->**onConnect**(**ac**,**REDIS\_OK**);**

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

**}**

该函数中，首先调用redisCheckSocketError判断当前TCP是否建链成功，如果该函数返回REDIS\_ERR，在errno为EINPROGRESS的情况下，说明TCP尚在建链中，这种情况直接返回REDIS\_OK，等待下次处理；其他情况说明建链失败，以REDIS\_ERR为参数，调用异步上下文中的建链回调函数，然后调用\_\_redisAsyncDisconnect做清理工作，最后返回REDIS\_ERR；

如果redisCheckSocketError函数返回REDIS\_OK，则将REDIS\_CONNECTED标记增加到上下文标志位中，并以REDIS\_OK为参数调用异步上下文中的建链回调函数；最后返回REDIS\_OK；

**二：发送命令，接收回复**

redisAsyncCommand函数，是异步API中用于向Redis发送命令的函数。该函数与同步API中发送命令的函数redisCommand类似，同样支持printf式的可变参数。该函数的原型如下：

int redisAsyncCommand**(**redisAsyncContext **\***ac**,** redisCallbackFn **\***fn**,** void **\***privdata**,** const char **\***format**,** **...);**

这里的fn和privdata分别表示收到命令回复后要调用的回调函数及其参数。因为Redis是单线程处理命令，因此当客户端使用异步API与事件库的结合之后，命令就自动的管道化了。也就是客户端在单线程模式下，发送命令的顺序和接收回复的顺序是一致的。因此，当发送命令时，就会将回调函数fn和参数privdata封装成回调结构redisCallback，并将该结构记录到单链表或者字典中。当收到回复后，就会依次得到链表或者字典中的redisCallback结构，调用其中的回调函数。

redisAsyncCommand函数主要是调用redisvAsyncCommand实现，而redisvAsyncCommand函数又是通过调用redisvFormatCommand和\_\_redisAsyncCommand函数实现的。

redisvFormatCommand，解析用户输入的命令，转换成统一请求协议格式的字符串cmd，然后调用\_\_redisAsyncCommand函数，将cmd发送给Redis，并且记录相应的回调函数。

\_\_redisAsyncCommand函数的代码如下：

static int \_\_redisAsyncCommand**(**redisAsyncContext **\***ac**,** redisCallbackFn **\***fn**,** void **\***privdata**,** char **\***cmd**,** size\_t len**)** **{**

redisContext **\***c **=** **&(**ac**->**c**);**

redisCallback cb**;**

int pvariant**,** hasnext**;**

char **\***cstr**,** **\***astr**;**

size\_t clen**,** alen**;**

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

sds sname**;**

*/\* Don't accept new commands when the connection is about to be closed. \*/*

**if** **(**c**->**flags **&** **(**REDIS\_DISCONNECTING **|** REDIS\_FREEING**))** **return** REDIS\_ERR**;**

*/\* Setup callback \*/*

cb**.**fn **=** fn**;**

cb**.**privdata **=** privdata**;**

*/\* Find out which command will be appended. \*/*

p **=** nextArgument**(**cmd**,&**cstr**,&**clen**);**

assert**(**p **!=** **NULL);**

hasnext **=** **(**p**[**0**]** **==** '$'**);**

pvariant **=** **(**tolower**(**cstr**[**0**])** **==** 'p'**)** **?** 1 **:** 0**;**

cstr **+=** pvariant**;**

clen **-=** pvariant**;**

**if** **(**hasnext **&&** strncasecmp**(**cstr**,**"subscribe\r\n"**,**11**)** **==** 0**)** **{**

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

*/\* Add every channel/pattern to the list of subscription callbacks. \*/*

**while** **((**p **=** nextArgument**(**p**,&**astr**,&**alen**))** **!=** **NULL)** **{**

sname **=** sdsnewlen**(**astr**,**alen**);**

**if** **(**pvariant**)**

dictReplace**(**ac**->**sub**.**patterns**,**sname**,&**cb**);**

**else**

dictReplace**(**ac**->**sub**.**channels**,**sname**,&**cb**);**

**}**

**}** **else** **if** **(**strncasecmp**(**cstr**,**"unsubscribe\r\n"**,**13**)** **==** 0**)** **{**

*/\* It is only useful to call (P)UNSUBSCRIBE when the context is*

*\* subscribed to one or more channels or patterns. \*/*

**if** **(!(**c**->**flags **&** REDIS\_SUBSCRIBED**))** **return** REDIS\_ERR**;**

*/\* (P)UNSUBSCRIBE does not have its own response: every channel or*

*\* pattern that is unsubscribed will receive a message. This means we*

*\* should not append a callback function for this command. \*/*

**}** **else** **if(**strncasecmp**(**cstr**,**"monitor\r\n"**,**9**)** **==** 0**)** **{**

*/\* Set monitor flag and push callback \*/*

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

\_\_redisPushCallback**(&**ac**->**replies**,&**cb**);**

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

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

*/\* This will likely result in an error reply, but it needs to be*

*\* received and passed to the callback. \*/*

\_\_redisPushCallback**(&**ac**->**sub**.**invalid**,&**cb**);**

**else**

\_\_redisPushCallback**(&**ac**->**replies**,&**cb**);**

**}**

\_\_redisAppendCommand**(**c**,**cmd**,**len**);**

*/\* Always schedule a write when the write buffer is non-empty \*/*

\_EL\_ADD\_WRITE**(**ac**);**

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

**}**

在函数中，首先将回调函数fn，以及用户提供的该回调函数的私有参数privdata，封装到redisCallback回调结构的cb中。当然，用户如果没有提供回调函数和参数，则cb中相应的属性为NULL。

然后解析用户输入命令，根据不同的命令，将回调函数追加到不同的链表或字典中：

如果用户输入命令为"subscribe"或者"psubscribe"，首先将REDIS\_SUBSCRIBED标记增加到上下文标志中，表示当前客户端进入订阅模式；

然后循环解析命令中的后续参数，这些参数表示订阅的频道名（"subscribe"），或者订阅的频道名的匹配模式（"psubscribe"）。以这些频道名或匹配模式为key，以回调结构cb为value，插入到异步上下文的字典ac->sub.patterns或ac->sub.channels中。

如果用户输入命令为" unsubscribe"，这种情况无需记录回调函数。但是该命令只有客户端处于订阅模式下才有效，否则直接返回REDIS\_ERR；

如果用户输入命令为"monitor"，则将REDIS\_MONITORING标记增加到上下文标志位中，表示客户端进入monitor模式，然后调用\_\_redisPushCallback，将回调结构cb追加到上下文的回调链表ac->replies中；

如果用户输入的是其他命令，则若当前客户端处于订阅模式，因处于订阅模式中，客户端只能发送”subscribe/psubscribe/unsubscribe/punsubscribe”命令，走到这一步，说明客户端发送了其他命令，因此将回调结构cb追加到链表ac->sub.invalid中；

其他情况，将回调结构cb追加到链表ac->replies中；

记录完回调函数之后，剩下的，就是调用\_\_redisAppendCommand，将cmd追加到上下文的输出缓存中。

然后调用\_EL\_ADD\_WRITE，注册可写事件。对于使用Redis的ae事件库的客户端来说，该宏定义实际上就是调用redisAeAddWrite函数，可写事件的回调函数是redisAeWriteEvent，该函数调用redisAsyncHandleWrite实现。

redisAsyncHandleWrite函数的全部代码如下：

void redisAsyncHandleWrite**(**redisAsyncContext **\***ac**)** **{**

redisContext **\***c **=** **&(**ac**->**c**);**

int done **=** 0**;**

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

*/\* Abort connect was not successful. \*/*

**if** **(**\_\_redisAsyncHandleConnect**(**ac**)** **!=** REDIS\_OK**)**

**return;**

*/\* Try again later when the context is still not connected. \*/*

**if** **(!(**c**->**flags **&** REDIS\_CONNECTED**))**

**return;**

**}**

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

\_\_redisAsyncDisconnect**(**ac**);**

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

*/\* Continue writing when not done, stop writing otherwise \*/*

**if** **(!**done**)**

\_EL\_ADD\_WRITE**(**ac**);**

**else**

\_EL\_DEL\_WRITE**(**ac**);**

*/\* Always schedule reads after writes \*/*

\_EL\_ADD\_READ**(**ac**);**

**}**

**}**

首先处理建链尚未成功的情况，之前已经讲过，不在赘述。

建链成功之后，调用redisBufferWrite，将上下文中输出缓存的内容通过socket描述符发送出去。

全部发送成功之后，调用\_EL\_DEL\_WRITE，删除注册的可写事件。对于使用Redis的ae事件库的客户端来说，这里就是调用redisAeDelWrite函数，删除注册的可写事件。

然后，调用\_EL\_ADD\_READ，注册可读事件。对于使用Redis的ae事件库的客户端来说，这里就是调用redisAeAddRead函数，注册可读事件。事件回调函数为redisAeReadEvent。该回调函数主要是调用redisAsyncHandleRead实现。

redisAsyncHandleRead函数的代码如下：

void redisAsyncHandleRead**(**redisAsyncContext **\***ac**)** **{**

redisContext **\***c **=** **&(**ac**->**c**);**

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

*/\* Abort connect was not successful. \*/*

**if** **(**\_\_redisAsyncHandleConnect**(**ac**)** **!=** REDIS\_OK**)**

**return;**

*/\* Try again later when the context is still not connected. \*/*

**if** **(!(**c**->**flags **&** REDIS\_CONNECTED**))**

**return;**

**}**

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

\_\_redisAsyncDisconnect**(**ac**);**

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

*/\* Always re-schedule reads \*/*

\_EL\_ADD\_READ**(**ac**);**

redisProcessCallbacks**(**ac**);**

**}**

**}**

该函数中，首先处理未建链的情况，与redisAsyncHandleWrite中的处理方式一致，不在赘述。

建链成功之后，首先调用redisBufferRead，从socket中读取数据，并追加到解析器的输入缓存中，这在同步操作API中已讲过，不再赘述。

读取成功之后，调用redisProcessCallbacks函数进行处理。该函数就是根据回复信息找到相应的回调结构，然后调用其中的回调函数。redisProcessCallbacks函数的代码如下：

void redisProcessCallbacks**(**redisAsyncContext **\***ac**)** **{**

redisContext **\***c **=** **&(**ac**->**c**);**

redisCallback cb **=** **{NULL,** **NULL,** **NULL};**

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

int status**;**

**while((**status **=** redisGetReply**(**c**,&**reply**))** **==** REDIS\_OK**)** **{**

**if** **(**reply **==** **NULL)** **{**

*/\* When the connection is being disconnected and there are*

*\* no more replies, this is the cue to really disconnect. \*/*

**if** **(**c**->**flags **&** REDIS\_DISCONNECTING **&&** sdslen**(**c**->**obuf**)** **==** 0**)** **{**

\_\_redisAsyncDisconnect**(**ac**);**

**return;**

**}**

*/\* If monitor mode, repush callback \*/*

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

\_\_redisPushCallback**(&**ac**->**replies**,&**cb**);**

**}**

*/\* When the connection is not being disconnected, simply stop*

*\* trying to get replies and wait for the next loop tick. \*/*

**break;**

**}**

*/\* Even if the context is subscribed, pending regular callbacks will*

*\* get a reply before pub/sub messages arrive. \*/*

**if** **(**\_\_redisShiftCallback**(&**ac**->**replies**,&**cb**)** **!=** REDIS\_OK**)** **{**

*/\**

*\* A spontaneous reply in a not-subscribed context can be the error*

*\* reply that is sent when a new connection exceeds the maximum*

*\* number of allowed connections on the server side.*

*\**

*\* This is seen as an error instead of a regular reply because the*

*\* server closes the connection after sending it.*

*\**

*\* To prevent the error from being overwritten by an EOF error the*

*\* connection is closed here. See issue #43.*

*\**

*\* Another possibility is that the server is loading its dataset.*

*\* In this case we also want to close the connection, and have the*

*\* user wait until the server is ready to take our request.*

*\*/*

**if** **(((**redisReply**\*)**reply**)->**type **==** REDIS\_REPLY\_ERROR**)** **{**

c**->**err **=** REDIS\_ERR\_OTHER**;**

snprintf**(**c**->**errstr**,sizeof(**c**->**errstr**),**"%s"**,((**redisReply**\*)**reply**)->**str**);**

c**->**reader**->**fn**->**freeObject**(**reply**);**

\_\_redisAsyncDisconnect**(**ac**);**

**return;**

**}**

*/\* No more regular callbacks and no errors, the context \*must\* be subscribed or monitoring. \*/*

assert**((**c**->**flags **&** REDIS\_SUBSCRIBED **||** c**->**flags **&** REDIS\_MONITORING**));**

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

\_\_redisGetSubscribeCallback**(**ac**,**reply**,&**cb**);**

**}**

**if** **(**cb**.**fn **!=** **NULL)** **{**

\_\_redisRunCallback**(**ac**,&**cb**,**reply**);**

c**->**reader**->**fn**->**freeObject**(**reply**);**

*/\* Proceed with free'ing when redisAsyncFree() was called. \*/*

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

\_\_redisAsyncFree**(**ac**);**

**return;**

**}**

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

*/\* No callback for this reply. This can either be a NULL callback,*

*\* or there were no callbacks to begin with. Either way, don't*

*\* abort with an error, but simply ignore it because the client*

*\* doesn't know what the server will spit out over the wire. \*/*

c**->**reader**->**fn**->**freeObject**(**reply**);**

**}**

**}**

*/\* Disconnect when there was an error reading the reply \*/*

**if** **(**status **!=** REDIS\_OK**)**

\_\_redisAsyncDisconnect**(**ac**);**

**}**

该函数循环调用redisGetReply，将解析器中输入缓存中的内容，组织成redisReply结构树，树的根节点通过参数reply返回。

在循环中，如果取得的reply为NULL，说明输入缓存已空，这种情况下，如果当前上下文标志位中设置了REDIS\_DISCONNECTING，说明之前某个命令的回调函数中，调用了redisAsyncDisconnect函数设置了该标记，因此在输出缓存为空，并且输入缓存也为空（reply为NULL）的条件下，调用\_\_redisAsyncDisconnect开始执行断链操作，释放清理内存，最后返回。

如果取得的reply为NULL，并且当前处于监控模式下，则将上次取出的回调结构cb，重新插入到链表ac->replies中。最后退出循环。

如果取得的reply非空，则首先调用\_\_redisShiftCallback，尝试从链表ac->replies中取出第一个回调结构cb。

如果链表ac->replies已空，这种情况下，客户端要么是处于订阅模式下，要么就是服务器主动向客户端发送了某个错误信息，比如该客户端向服务器建链，服务器中已经超过了最大的客户端数，或者是服务器正在加载转储数据，而向客户端返回一个错误信息。

如果回复类型为REDIS\_REPLY\_ERROR，则调用\_\_redisAsyncDisconnect断链；如果回复类型不是REDIS\_REPLY\_ERROR，则当前客户端只能处于订阅模式或者监控模式，如果当前处于订阅模式下，则调用\_\_redisGetSubscribeCallback，根据reply，从相应的字典中取出回调结构cb；

取得回调结构cb之后，只要其中的回调函数不为空，就调用\_\_redisRunCallback函数，调用其中的回调函数；对于回调函数为空的回调结构，直接释放reply即可。

\_\_redisGetSubscribeCallback函数根据回复信息，在字典结构中找到对应的回调结构并返回该结构。它的代码如下：

static int \_\_redisGetSubscribeCallback**(**redisAsyncContext **\***ac**,** redisReply **\***reply**,** redisCallback **\***dstcb**)** **{**

redisContext **\***c **=** **&(**ac**->**c**);**

dict **\***callbacks**;**

dictEntry **\***de**;**

int pvariant**;**

char **\***stype**;**

sds sname**;**

*/\* Custom reply functions are not supported for pub/sub. This will fail*

*\* very hard when they are used... \*/*

**if** **(**reply**->**type **==** REDIS\_REPLY\_ARRAY**)** **{**

assert**(**reply**->**elements **>=** 2**);**

assert**(**reply**->**element**[**0**]->**type **==** REDIS\_REPLY\_STRING**);**

stype **=** reply**->**element**[**0**]->**str**;**

pvariant **=** **(**tolower**(**stype**[**0**])** **==** 'p'**)** **?** 1 **:** 0**;**

**if** **(**pvariant**)**

callbacks **=** ac**->**sub**.**patterns**;**

**else**

callbacks **=** ac**->**sub**.**channels**;**

*/\* Locate the right callback \*/*

assert**(**reply**->**element**[**1**]->**type **==** REDIS\_REPLY\_STRING**);**

sname **=** sdsnewlen**(**reply**->**element**[**1**]->**str**,**reply**->**element**[**1**]->**len**);**

de **=** dictFind**(**callbacks**,**sname**);**

**if** **(**de **!=** **NULL)** **{**

memcpy**(**dstcb**,**dictGetEntryVal**(**de**),sizeof(\***dstcb**));**

*/\* If this is an unsubscribe message, remove it. \*/*

**if** **(**strcasecmp**(**stype**+**pvariant**,**"unsubscribe"**)** **==** 0**)** **{**

dictDelete**(**callbacks**,**sname**);**

*/\* If this was the last unsubscribe message, revert to*

*\* non-subscribe mode. \*/*

assert**(**reply**->**element**[**2**]->**type **==** REDIS\_REPLY\_INTEGER**);**

**if** **(**reply**->**element**[**2**]->**integer **==** 0**)**

c**->**flags **&=** **~**REDIS\_SUBSCRIBED**;**

**}**

**}**

sdsfree**(**sname**);**

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

*/\* Shift callback for invalid commands. \*/*

\_\_redisShiftCallback**(&**ac**->**sub**.**invalid**,**dstcb**);**

**}**

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

**}**

正常情况下，处于订阅模式下的客户端，接收到的消息类型应该是REDIS\_REPLY\_ARRAY类型，比如：

1**)** "message"

2**)** "channel1"

3**)** "hi

1**)** "pmessage"

2**)** "channel.?\*"

3**)** "channel.1"

4**)** "this is channel.1"

根据回复信息第一行的首字节是否为”p”，找到不同的字典结构callbacks。然后根据reply->element[1]的内容，也就是频道名或者频道名模式，从字典中找到相应的回调结构。

如果Redis回复的信息是"unsubscribe"，则从字典中删除相应的回调结构，此时reply->element[2]中的信息应该是个整数，表示当前客户端目前还订阅了多少频道，如果该值为0，表示客户端已经从最后一个频道中退订了，因此将REDIS\_SUBSCRIBED标记从标志位c->flags中删除，表示该客户端退出订阅模式；

如果Redis的回复信息不是REDIS\_REPLY\_ARRAY类型，说明发生了异常，此时从链表ac->sub.invalid中取出下一个回调结构即可。

**三：断链**

客户端可以通过调用redisAsyncDisconnect函数主动断链。该函数的代码如下：

void redisAsyncDisconnect**(**redisAsyncContext **\***ac**)** **{**

redisContext **\***c **=** **&(**ac**->**c**);**

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

**if** **(!(**c**->**flags **&** REDIS\_IN\_CALLBACK**)** **&&** ac**->**replies**.**head **==** **NULL)**

\_\_redisAsyncDisconnect**(**ac**);**

**}**

一般情况下，该函数是在某个命令回调函数中被调用。当调用该函数时，并不一定会立即进行断链操作，该函数将REDIS\_DISCONNECTING标记增加到上下文的标志位中。只有当输出缓存中所有命令都发送完毕，并且收到他们的回复，调用了回调函数之后，才会真正的执行断链操作，这是在函数redisProcessCallbacks中处理的。

设置了REDIS\_DISCONNECTING标记后，在\_\_redisAsyncCommand函数中，会直接返回REDIS\_ERR，表示不再发送新的命令。

真正的断链操作由函数\_\_redisAsyncDisconnect实现。

当客户与服务器之间的交互过程中发生了错误，或者是服务器主动断链时，就会调用\_\_redisAsyncDisconnect进入断链流程。该函数代码如下：

static void \_\_redisAsyncDisconnect**(**redisAsyncContext **\***ac**)** **{**

redisContext **\***c **=** **&(**ac**->**c**);**

*/\* Make sure error is accessible if there is any \*/*

\_\_redisAsyncCopyError**(**ac**);**

**if** **(**ac**->**err **==** 0**)** **{**

*/\* For clean disconnects, there should be no pending callbacks. \*/*

assert**(**\_\_redisShiftCallback**(&**ac**->**replies**,NULL)** **==** REDIS\_ERR**);**

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

*/\* Disconnection is caused by an error, make sure that pending*

*\* callbacks cannot call new commands. \*/*

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

**}**

*/\* For non-clean disconnects, \_\_redisAsyncFree() will execute pending*

*\* callbacks with a NULL-reply. \*/*

\_\_redisAsyncFree**(**ac**);**

**}**

首先调用\_\_redisAsyncCopyError，得到异步上下文中的err，如果err为0，则说明是客户端主动断链，这种情况下，链表ac->replies应该为空才对；否则，将上下文标志位中增加REDIS\_DISCONNECTING标记，表明这是由于错误引起的断链，设置该标记后，不再发送新的命令给Redis。

最终调用\_\_redisAsyncFree函数，进行最后的清理。在\_\_redisAsyncFree函数中，会议NULL为reply，调用所有异步上下文中尚存的回调函数。然后调用断链回调函数，最后调用redisFree关闭socket描述符，清理释放空间。