Redis中，处理网络IO时，采用的是事件驱动机制。但它没有使用libevent或者libev这样的库，而是自己实现了一个非常简单明了的事件驱动库ae\_event，主要代码仅仅400行左右。

没有选择libevent或libev的原因大概在于，这些库为了迎合通用性造成代码庞大，而且其中的很多功能，比如监控子进程，复杂的定时器等，这些都不是Redis所需要的。

Redis中的事件驱动库只关注网络IO，以及定时器。该事件库处理下面两类事件：

a：文件事件(file event)：用于处理Redis服务器和客户端之间的网络IO。

b：时间事件(time eveat)：Redis服务器中的一些操作（比如serverCron函数）需要在给定的时间点执行，而时间事件就是处理这类定时操作的。

事件驱动库的代码主要是在src/ae.c中实现的。

**一：文件事件**

Redis基于Reactor模式开发了自己的网络事件处理器，也就是文件事件处理器。文件事件处理器使用IO多路复用技术，同时监听多个套接字，并为套接字关联不同的事件处理函数。当套接字的可读或者可写事件触发时，就会调用相应的事件处理函数。

Redis使用的IO多路复用技术主要有：select、epoll、evport和kqueue等。每个IO多路复用函数库在Redis源码中都对应一个单独的文件，比如ae\_select.c，ae\_epoll.c， ae\_kqueue.c等。

这些多路复用技术，根据不同的操作系统，Redis按照一定的优先级，选择其中的一种使用。在ae.c中，是这样实现的：

#ifdef HAVE\_EVPORT

#include "ae\_evport.c"

#else

#ifdef HAVE\_EPOLL

#include "ae\_epoll.c"

#else

#ifdef HAVE\_KQUEUE

#include "ae\_kqueue.c"

#else

#include "ae\_select.c"

#endif

#endif

#endif

注意这里是include的.c文件，因此，使用哪种多路复用技术，是在编译阶段就决定了的。

文件事件由结构体aeFileEvent表示，它的定义如下：

/\* File event structure \*/

**typedef** struct aeFileEvent **{**

int mask**;** */\* one of AE\_(READABLE|WRITABLE) \*/*

aeFileProc **\***rfileProc**;**

aeFileProc **\***wfileProc**;**

void **\***clientData**;**

**}** aeFileEvent**;**

其中mask表示描述符注册的事件，可以是AE\_READABLE，AE\_WRITABLE或者是AE\_READABLE|AE\_WRITABLE。

rfileProc和wfileProc分别表示可读和可写事件的回调函数。

clientData是用户提供的数据，在调用回调函数时被当做参数。注意，该数据是可读和可写事件共用的。

**二：时间事件**

Redis的时间事件主要有一次性事件和周期性事件两种。一次性时间事件仅触发一次，而周期性事件每隔一段时间就触发一次。

时间事件由aeTimeEvent结构体表示，它的定义如下：

/\* Time event structure \*/

**typedef** struct aeTimeEvent **{**

long long id**;** */\* time event identifier. \*/*

long when\_sec**;** */\* seconds \*/*

long when\_ms**;** */\* milliseconds \*/*

aeTimeProc **\***timeProc**;**

aeEventFinalizerProc **\***finalizerProc**;**

void **\***clientData**;**

struct aeTimeEvent **\***next**;**

**}** aeTimeEvent**;**

id用于标识时间事件，id号按照从小到大的顺序递增，新时间事件的id号比旧时间事件的id号要大；

when\_sec和when\_ms表示时间事件的下次触发时间，实际上就是一个Unix时间戳，when\_sec记录它的秒数，when\_ms记录它的毫秒数。因此触发时间是一个绝对值，而非相对值；

timeProc是时间事件处理器，也就是时间事件触发时的回调函数；

finalizerProc是删除该时间事件时要调用的函数；

clientData是用户提供的数据，在调用timeProc和finalizerProc时，作为参数；

所有的时间事件aeTimeEvent结构被组织成一个链表，next指针就执行链表中，当前aeTimeEvent结构的后继结点。

aeTimeEvent结构链表是一个无序链表，也就是说它并不按照事件的触发时间而排序。每当创建一个新的时间事件aeTimeEvent结构时，该结构就插入链表的头部。因此，当监控时间事件时，需要遍历整个链表，查找所有已到达的时间事件，并调用相应的事件处理器。

在目前版本中，正常模式下的Redis服务器只使用serverCron一个时间事件，而在benchmark模式下，服务器也只使用两个时间事件。因此，时间事件链表的这种设计虽然简单粗暴，但是也能满足性能需求。

**三：事件循环结构**

在事件驱动的实现中，需要有一个事件循环结构来监控调度所有的事件，比如Libevent库中的event\_base，libev中的ev\_loop等。

在Redis中的事件驱动库中，事件循环结构是由aeEventLoop结构体实现的，aeEventLoop结构是Redis中事件驱动机制的主要数据结构。它的定义如下：

**typedef** struct aeEventLoop **{**

int maxfd**;** */\* highest file descriptor currently registered \*/*

int setsize**;** */\* max number of file descriptors tracked \*/*

long long timeEventNextId**;**

time\_t lastTime**;** */\* Used to detect system clock skew \*/*

aeFileEvent **\***events**;** */\* Registered events \*/*

aeFiredEvent **\***fired**;** */\* Fired events \*/*

aeTimeEvent **\***timeEventHead**;**

int stop**;**

void **\***apidata**;** */\* This is used for polling API specific data \*/*

aeBeforeSleepProc **\***beforesleep**;**

**}** aeEventLoop**;**

events是aeFileEvent结构的数组，每个aeFileEvent结构表示一个注册的文件事件。events数组以描述符的值为下标。

fired是aeFiredEvent结构的数组，aeFiredEvent结构表示一个触发的文件事件。结构中包含了描述符，以及其上已经触发的事件。该数组不是以描述符的值为下标，而是依次保存所有触发的文件事件。当处理事件时，轮训fired数组中的每个元素，然后依次处理。

setsize表示eventLoop->events和eventLoop->fired数组的大小。因此，setsize - 1就表示所能处理的最大的描述符的值。

lastTime：为了处理时间事件而记录的Unix时间戳，主要为了在系统时间被调整时能够尽快的处理时间事件；

timeEventHead：时间事件aeTimeEvent结构组成的链表的头指针；

timeEventNextId：下个时间事件的ID，该ID依次递增，因此当前时间事件的最大ID为timeEventNextId-1；

stop：是否停止事件监控；

maxfd：当前处理的最大的描述符的值，主要是在select中使用；

beforesleep：每次监控事件触发之前，需要调用的函数；

apidata表示具体的底层多路复用所使用的数据结构，比如对于select来说，该结构中保存了读写描述符数组；对于epoll来说，该结构中保存了epoll描述符，以及epoll\_event结构数组；

**四：监控调度时间事件**

监控调度时间事件是由函数processTimeEvents实现的，它的代码如下：

static int processTimeEvents**(**aeEventLoop **\***eventLoop**)** **{**

int processed **=** 0**;**

aeTimeEvent **\***te**;**

long long maxId**;**

time\_t now **=** time**(NULL);**

*/\* If the system clock is moved to the future, and then set back to the*

*\* right value, time events may be delayed in a random way. Often this*

*\* means that scheduled operations will not be performed soon enough.*

*\**

*\* Here we try to detect system clock skews, and force all the time*

*\* events to be processed ASAP when this happens: the idea is that*

*\* processing events earlier is less dangerous than delaying them*

*\* indefinitely, and practice suggests it is. \*/*

**if** **(**now **<** eventLoop**->**lastTime**)** **{**

te **=** eventLoop**->**timeEventHead**;**

**while(**te**)** **{**

te**->**when\_sec **=** 0**;**

te **=** te**->**next**;**

**}**

**}**

eventLoop**->**lastTime **=** now**;**

te **=** eventLoop**->**timeEventHead**;**

maxId **=** eventLoop**->**timeEventNextId**-**1**;**

**while(**te**)** **{**

long now\_sec**,** now\_ms**;**

long long id**;**

**if** **(**te**->**id **>** maxId**)** **{**

te **=** te**->**next**;**

**continue;**

**}**

aeGetTime**(&**now\_sec**,** **&**now\_ms**);**

**if** **(**now\_sec **>** te**->**when\_sec **||**

**(**now\_sec **==** te**->**when\_sec **&&** now\_ms **>=** te**->**when\_ms**))**

**{**

int retval**;**

id **=** te**->**id**;**

retval **=** te**->**timeProc**(**eventLoop**,** id**,** te**->**clientData**);**

processed**++;**

*/\* After an event is processed our time event list may*

*\* no longer be the same, so we restart from head.*

*\* Still we make sure to don't process events registered*

*\* by event handlers itself in order to don't loop forever.*

*\* To do so we saved the max ID we want to handle.*

*\**

*\* FUTURE OPTIMIZATIONS:*

*\* Note that this is NOT great algorithmically. Redis uses*

*\* a single time event so it's not a problem but the right*

*\* way to do this is to add the new elements on head, and*

*\* to flag deleted elements in a special way for later*

*\* deletion (putting references to the nodes to delete into*

*\* another linked list). \*/*

**if** **(**retval **!=** AE\_NOMORE**)** **{**

aeAddMillisecondsToNow**(**retval**,&**te**->**when\_sec**,&**te**->**when\_ms**);**

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

aeDeleteTimeEvent**(**eventLoop**,** id**);**

**}**

te **=** eventLoop**->**timeEventHead**;**

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

te **=** te**->**next**;**

**}**

**}**

**return** processed**;**

**}**

首先判断系统时间是否被调整了。将当前时间now，与上次记录的时间戳eventLoop->lastTime相比较，如果now小于eventLoop->lastTime，说明系统时间被调整到过去了，比如由201603312030调整到了201603312000了，这种情况下，直接将所有事件的触发时间的秒数清0，这意味着所有的时间事件都会立即触发。之所以这么做，是因为提前处理比延后处理的危险性要小；

然后更新eventLoop->lastTime为now；

接下来，先记录当前的maxId。之所以这么做，是因为有时间事件触发后，要重新回到链表头结点开始处理。而在时间事件的触发回调函数中，有可能注册了新的时间事件，成为新的链表头结点，这就可能导致会无限处理下去。为了防止这种情况发生，记录当前的maxId，只处理当前的时间事件；

轮训链表eventLoop->timeEventHead，针对其中的每一个事件节点te，如果te的id大于maxId，说明该事件，是在之前已经触发的时间事件的回调函数中注册的，不处理这样的事件，直接处理下一个；

然后得到当前时间，判断当前时间是否已经超过了te的触发时间，若是，说明该事件需要触发，调用触发回调函数te->timeProc，该函数的返回值为retval；

如果retval是AE\_NOMORE，说明触发的时间事件是一次性事件，直接从链表中删除；否则，说明该事件是周期性事件，将其触发时间更改为当前时间加上retval；

事件触发后，链表已经被修改了，要重新回到链表头结点开始处理。因为Redis中只有一个时间事件，因此采用了这种简单粗暴的算法，更好的处理方式是处理完当前事件后，标记该节点需要删除（比如在另一个链表中保存该节点的指针），然后接着处理下一个节点，所有节点处理完之后，将标记为删除的节点统一删除即可。

最后返回触发的事件总数。

**五：监控调度所有事件**

监控调度所有事件是由函数aeProcessEvents实现的，它的代码如下：

int aeProcessEvents**(**aeEventLoop **\***eventLoop**,** int flags**)**

**{**

int processed **=** 0**,** numevents**;**

*/\* Nothing to do? return ASAP \*/*

**if** **(!(**flags **&** AE\_TIME\_EVENTS**)** **&&** **!(**flags **&** AE\_FILE\_EVENTS**))** **return** 0**;**

*/\* Note that we want call select() even if there are no*

*\* file events to process as long as we want to process time*

*\* events, in order to sleep until the next time event is ready*

*\* to fire. \*/*

**if** **(**eventLoop**->**maxfd **!=** **-**1 **||**

**((**flags **&** AE\_TIME\_EVENTS**)** **&&** **!(**flags **&** AE\_DONT\_WAIT**)))** **{**

int j**;**

aeTimeEvent **\***shortest **=** **NULL;**

struct timeval tv**,** **\***tvp**;**

**if** **(**flags **&** AE\_TIME\_EVENTS **&&** **!(**flags **&** AE\_DONT\_WAIT**))**

shortest **=** aeSearchNearestTimer**(**eventLoop**);**

**if** **(**shortest**)** **{**

long now\_sec**,** now\_ms**;**

*/\* Calculate the time missing for the nearest*

*\* timer to fire. \*/*

aeGetTime**(&**now\_sec**,** **&**now\_ms**);**

tvp **=** **&**tv**;**

tvp**->**tv\_sec **=** shortest**->**when\_sec **-** now\_sec**;**

**if** **(**shortest**->**when\_ms **<** now\_ms**)** **{**

tvp**->**tv\_usec **=** **((**shortest**->**when\_ms**+**1000**)** **-** now\_ms**)\***1000**;**

tvp**->**tv\_sec **--;**

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

tvp**->**tv\_usec **=** **(**shortest**->**when\_ms **-** now\_ms**)\***1000**;**

**}**

**if** **(**tvp**->**tv\_sec **<** 0**)** tvp**->**tv\_sec **=** 0**;**

**if** **(**tvp**->**tv\_usec **<** 0**)** tvp**->**tv\_usec **=** 0**;**

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

*/\* If we have to check for events but need to return*

*\* ASAP because of AE\_DONT\_WAIT we need to set the timeout*

*\* to zero \*/*

**if** **(**flags **&** AE\_DONT\_WAIT**)** **{**

tv**.**tv\_sec **=** tv**.**tv\_usec **=** 0**;**

tvp **=** **&**tv**;**

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

*/\* Otherwise we can block \*/*

tvp **=** **NULL;** */\* wait forever \*/*

**}**

**}**

numevents **=** aeApiPoll**(**eventLoop**,** tvp**);**

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

aeFileEvent **\***fe **=** **&**eventLoop**->**events**[**eventLoop**->**fired**[**j**].**fd**];**

int mask **=** eventLoop**->**fired**[**j**].**mask**;**

int fd **=** eventLoop**->**fired**[**j**].**fd**;**

int rfired **=** 0**;**

*/\* note the fe->mask & mask & ... code: maybe an already processed*

*\* event removed an element that fired and we still didn't*

*\* processed, so we check if the event is still valid. \*/*

**if** **(**fe**->**mask **&** mask **&** AE\_READABLE**)** **{**

rfired **=** 1**;**

fe**->**rfileProc**(**eventLoop**,**fd**,**fe**->**clientData**,**mask**);**

**}**

**if** **(**fe**->**mask **&** mask **&** AE\_WRITABLE**)** **{**

**if** **(!**rfired **||** fe**->**wfileProc **!=** fe**->**rfileProc**)**

fe**->**wfileProc**(**eventLoop**,**fd**,**fe**->**clientData**,**mask**);**

**}**

processed**++;**

**}**

**}**

*/\* Check time events \*/*

**if** **(**flags **&** AE\_TIME\_EVENTS**)**

processed **+=** processTimeEvents**(**eventLoop**);**

**return** processed**;** */\* return the number of processed file/time events \*/*

**}**

根据flags处理不同的事件：

如果flags为0，则该函数直接返回；

如果flags中设置了AE\_ALL\_EVENTS，则处理所有的文件事件和时间事件；

如果flags中设置了AE\_FILE\_EVENTS，则处理所有的文件事件；

如果flags中设置了AE\_TIME\_EVENTS，则处理所有的时间事件；

如果flags中设置了AE\_DONT\_WAIT，则调用多路复用函数时，不会阻塞等

待事件的触发，将所有已触发的事件处理完后立即返回。

目前在Redis中，调用aeProcessEvents时设置的flags只有AE\_ALL\_EVENTS和

AE\_FILE\_EVENTS|AE\_DONT\_WAIT两种。

函数中，首先如果flags中既没有设置AE\_TIME\_EVENTS，也没有设置AE\_FILE\_EVENTS，则该函数直接返回0.

接下来，如果已经注册过文件事件，或者需要处理时间事件且不是AE\_DONT\_WAIT，则需要调用底层多路复用函数aeApiPoll。因此需要计算调用aeApiPoll函数时，最长阻塞时间tvp，该值是由最早要触发的时间事件（如果有的话）决定的。

如果需要处理时间事件且不是AE\_DONT\_WAIT，这种情况下，不管有没有文件事件，都要阻塞一段时间，阻塞的时间根据shortest得到，shortest是通过调用aeSearchNearestTimer得到的最早要触发的时间事件。得到shortest后，计算得出其触发时间距离当前时间的差值，该差值就是阻塞时间tvp；

否则，如果注册过文件事件，并且flags中设置了AE\_DONT\_WAIT，则将tvp中的值设置为0，表示完全不阻塞；

如果注册过文件事件，但是flags中没有设置AE\_DONT\_WAIT，则将tvp置为NULL，表示一直阻塞，直到有文件事件触发；

得到最长阻塞时间tvp之后，以tvp为参数调用aeApiPoll等待文件事件的触发。该函数由不同的底层多路复用函数实现，最终都返回触发的文件事件总数numevents，并将触发的事件和描述符，依次记录到eventLoop->fired中；

接下来，依次轮训eventLoop->fired中的前numevents个元素，调用相应的事件回调函数。注意，如果一个套接字又可读又可写的话，那么服务器将先处理可读事件，然后在处理可写事件。

触发的文件事件是依次处理的，如果某个文件事件的处理时间过长，就会影响到下一个事件的处理。在事件驱动的实现中，要由用户保证事件回调函数能够快速返回，而不阻塞。

注意，有这样一种情况，比如描述符3和4都有事件触发了，在3的事件回调函数中，调用aeDeleteFileEvent将4的注册事件删除了。这样在处理描述符4时，就不应该再次调用4的回调函数了。所以，每次调用事件回调函数之前，都判断该描述符上的注册事件是否还有效。而且如果可读和可写事件的回调函数相同的话，只能调用一次该函数。

处理完文件事件之后（或者没有文件事件，而仅仅阻塞了tvp的时间），如果flags中设置了AE\_TIME\_EVENTS，则调用processTimeEvents处理时间事件，因已经阻塞了tvp的时间，因此此时肯定有触发的时间事件。最后，返回所有触发的事件总数。

因为时间事件在文件事件之后处理，并且事件之间不会出现抢占，所以时间事件的实际处理时间，通常会比时间事件设定的到达时间稍晚一些。

再次强调一点：对文件事件和时间事件的处理都是同步、有序、原子地执行的，服务器不会中途中断事件处理，也不会对事件进行抢占。因此，不管是文件事件的回调函数，还是时间事件的回调函数，都需要尽可地减少程序的阻塞时间，从而降低造成事件饥饿的可能性。比如，在命令回复回调函数中，将一个命令回复写入到客户端套接字时，如果写人字节数超过了一个预设常量的话，命令回复函数就会主动用break跳出写人循环，将余下的数据留到下次再写。另外，时间事件也会将非常耗时的持久化操作放到子线程或者子进程执行。

**六：事件循环监控**

事件循环监控是由函数aeMain实现的，它的代码如下：

void aeMain**(**aeEventLoop **\***eventLoop**)** **{**

eventLoop**->**stop **=** 0**;**

**while** **(!**eventLoop**->**stop**)** **{**

**if** **(**eventLoop**->**beforesleep **!=** **NULL)**

eventLoop**->**beforesleep**(**eventLoop**);**

aeProcessEvents**(**eventLoop**,** AE\_ALL\_EVENTS**);**

**}**

**}**

只要eventLoop->stop不为1，则持续调用aeProcessEvents监控调度所有事件的触发。正常情况下，在Redis服务器中，eventLoop->stop永远不可能为1。

在Redis服务器的主函数中，所有初始化工作完成之后，就会调用该函数，监控所有事件的触发。

**七：例子：ECHO服务器**

下面是使用Redis的事件驱动库，实现的一个简单echo服务器：

#define SERVER\_PORT 9998

**typedef** struct

**{**

char clientaddr**[**INET\_ADDRSTRLEN**];**

int port**;**

char buf**[**1024**];**

**}**Userbuf**;**

void setunblock**(**int fd**)**

**{**

int flags**;**

**if** **((**flags **=** fcntl**(**fd**,** F\_GETFL**))** **==** **-**1**)**

**{**

perror**(**"fcntl(F\_GETFL) error"**);**

**return;**

**}**

flags **|=** O\_NONBLOCK**;**

**if** **(**fcntl**(**fd**,** F\_SETFL**,** flags**)** **==** **-**1**)**

**{**

perror**(**"fcntl(F\_SETFL) error"**);**

**return;**

**}**

**return;**

**}**

void acceptfun**(**struct aeEventLoop **\***eventLoop**,** int fd**,** void **\***clientData**,** int mask**)**

**{**

int acceptfd **=** **-**1**;**

struct sockaddr\_in cliaddr**;**

socklen\_t addrlen **=** **sizeof(**cliaddr**);**

acceptfd **=** accept**(**fd**,** **(**struct sockaddr **\*)&**cliaddr**,** **&**addrlen**);**

**if** **(**acceptfd **<** 0**)**

**{**

perror**(**"accept error\n"**);**

**return;**

**}**

Userbuf **\***usrbuf **=** calloc**(**1**,** **sizeof(**Userbuf**));**

printf**(**"calloc %p\n"**,** usrbuf**);**

inet\_ntop**(**AF\_INET**,** **&**cliaddr**.**sin\_addr**,** usrbuf**->**clientaddr**,** INET\_ADDRSTRLEN**),**

usrbuf**->**port **=** ntohs**(**cliaddr**.**sin\_port**);**

printf**(**"\naccept from <%s:%d>\n"**,** usrbuf**->**clientaddr**,** usrbuf**->**port**);**

setunblock**(**acceptfd**);**

**if** **(**aeCreateFileEvent**(**eventLoop**,** acceptfd**,** AE\_READABLE**,** readfun**,** usrbuf**)** **!=** AE\_OK**)**

**{**

perror**(**"aeCreateFileEvent error"**);**

close**(**acceptfd**);**

printf**(**"free %p\n"**,** usrbuf**);**

free**(**usrbuf**);**

**return;**

**}**

**return;**

**}**

void readfun**(**struct aeEventLoop **\***eventLoop**,** int fd**,** void **\***clientData**,** int mask**)**

**{**

char readbuf**[**1024**]** **=** **{};**

int len **=** **-**1**;**

Userbuf **\***usrbuf **=** **(**Userbuf **\*)**clientData**;**

**if** **((**len **=** read**(**fd**,** readbuf**,** 1024**))** **>** 0**)**

**{**

printf**(**"read from <%s:%d>: %s\n"**,** usrbuf**->**clientaddr**,** usrbuf**->**port**,** readbuf**);**

memcpy**(**usrbuf**->**buf**,** readbuf**,** 1024**);**

**if** **(**aeCreateFileEvent**(**eventLoop**,** fd**,** AE\_WRITABLE**,** writefun**,** clientData**)** **!=** AE\_OK**)**

**{**

printf**(**"aeCreateFileEvent error\n"**);**

**goto** END**;**

**}**

**else**

**return;**

**}**

**else** **if** **(**len **==** 0**)**

**{**

printf**(**"close link from %s\n"**,** usrbuf**->**buf**);**

**goto** END**;**

**}**

**else**

**{**

printf**(**"read error from %s\n"**,** usrbuf**->**buf**);**

**goto** END**;**

**}**

END**:**

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

aeDeleteFileEvent**(**eventLoop**,** fd**,** AE\_READABLE**);**

aeDeleteFileEvent**(**eventLoop**,** fd**,** AE\_WRITABLE**);**

printf**(**"free %p\n"**,** clientData**);**

free**(**clientData**);**

**return;**

**}**

void writefun**(**struct aeEventLoop **\***eventLoop**,** int fd**,** void **\***clientData**,** int mask**)**

**{**

int len **=** 0**;**

char **\***buf **=** **((**Userbuf **\*)**clientData**)->**buf**;**

len **=** strlen**(**buf**);**

printf**(**"write to client: %s\n"**,** buf**);**

**if(**write**(**fd**,** buf**,** len**)** **!=** len**)**

**{**

perror**(**"write error"**);**

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

aeDeleteFileEvent**(**eventLoop**,** fd**,** AE\_READABLE**);**

aeDeleteFileEvent**(**eventLoop**,** fd**,** AE\_WRITABLE**);**

printf**(**"free %p\n"**,** clientData**);**

free**(**clientData**);**

**}**

aeDeleteFileEvent**(**eventLoop**,** fd**,** AE\_WRITABLE**);**

**}**

int main**()**

**{**

int listenfd**;**

aeEventLoop **\***eventloop **=** **NULL;**

struct sockaddr\_in seraddr**;**

listenfd **=** socket**(**AF\_INET**,** SOCK\_STREAM**,** 0**);**

**if** **(**listenfd **<** 0**)**

**{**

perror**(**"socket error"**);**

**return** **-**1**;**

**}**

seraddr**.**sin\_family **=** AF\_INET**;**

seraddr**.**sin\_addr**.**s\_addr **=** htonl**(**INADDR\_ANY**);**

seraddr**.**sin\_port **=** htons**(**SERVER\_PORT**);**

**if** **(**bind**(**listenfd**,** **(**struct sockaddr **\*)&**seraddr**,** **sizeof(**seraddr**))** **<** 0**)**

**{**

perror**(**"bind error"**);**

close**(**listenfd**);**

**return** **-**1**;**

**}**

**if** **(**listen**(**listenfd**,** 5**)** **<** 0**)**

**{**

perror**(**"listen error"**);**

close**(**listenfd**);**

**return** **-**1**;**

**}**

eventloop **=** aeCreateEventLoop**(**1024**);**

**if** **(**eventloop **==** **NULL)**

**{**

printf**(**"aeCreateEventLoop error\n"**);**

close**(**listenfd**);**

**return** **-**1**;**

**}**

**if** **(**aeCreateFileEvent**(**eventloop**,** listenfd**,** AE\_READABLE**,** acceptfun**,** **NULL)** **!=** AE\_OK**)**

**{**

perror**(**"aeCreateFileEvent error"**);**

close**(**listenfd**);**

aeDeleteEventLoop**(**eventloop**);**

**return** **-**1**;**

**}**

aeMain**(**eventloop**);**

**return** 0**;**

**}**

这里要注意的是，对于同一个acceptfd，调用aeCreateFileEvent函数，分别注册可读事件和可写事件时，其clientData是共享的。如果在注册可写事件时，修改了clientData，则可读事件的clientData也相应改变，这是因为一个描述符只有一个aeFileEvent结构。

客户端的代码根据Webbench改写，具体代码见：

https://github.com/gqtc/redis-3.0.5/blob/master/redis-3.0.5/tests/hhunittest/test\_ae\_client.c

其他有关事件驱动的代码实现，可以参考：

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

https://github.com/gqtc/redis-3.0.5/blob/master/redis-3.0.5/src/ae\_epoll.c

https://github.com/gqtc/redis-3.0.5/blob/master/redis-3.0.5/src/ae\_select.c