11-Redis事件驱动框架（下）：Redis有哪些事件？

你好，我是蒋德钧。

在第9讲中，我给你介绍了Linux提供的三种IO多路复用机制，分别是select、poll和epoll，这是Redis实现事件驱动框架的操作系统层支撑技术。

紧接着在上节课，我带你学习了Redis事件驱动框架的基本工作机制，其中介绍了事件驱动框架基于的Reactor模型，并以IO事件中的客户端连接事件为例，给你介绍了**框架运行的基本流程**：从server初始化时调用aeCreateFileEvent函数注册监听事件，到server初始化完成后调用aeMain函数，而aeMain函数循环执行aeProceeEvent函数，来捕获和处理客户端请求触发的事件。

但是在上节课当中，我们主要关注的是框架基本流程，所以到这里，你或许仍然存有一些疑问，比如说：

- Redis事件驱动框架监听的IO事件，除了上节课介绍的客户端连接以外，还有没有其他事件？而除了IO事件以外，框架还会监听其他事件么？
- 这些事件的创建和处理又分别对应了Redis源码中的哪些具体操作？

今天这节课，我就来给你介绍下Redis事件驱动框架中的两大类事件类型：**IO事件和时间事件，以及它们相应的处理机制。**

事实上，了解和学习这部分内容，一方面可以帮助我们更加全面地掌握，Redis事件驱动框架是如何以事件形式，处理server运行过程中面临的请求操作和多种任务的。比如，正常的客户端读写请求是以什么事件、由哪个函数进行处理，以及后台快照任务又是如何及时启动的。

因为事件驱动框架是Redis server运行后的核心循环流程，了解它何时用什么函数处理哪种事件，对我们排查server运行过程中遇到的问题，是很有帮助的。

另一方面，我们还可以学习到如何在一个框架中，同时处理IO事件和时间事件。我们平时开发服务器端程序，经常需要处理周期性任务，而Redis关于两类事件的处理实现，就给了我们一个不错的参考。

好，为了对这两类事件有个相对全面的了解，接下来，我们先从事件驱动框架循环流程的数据结构及其初始化开始学起，因为这里面就包含了针对这两类事件的数据结构定义和初始化操作。

aeEventLoop结构体与初始化

首先，我们来看下Redis事件驱动框架循环流程对应的数据结构aeEventLoop。这个结构体是在事件驱动框架代码ae.h中定义的，记录了框架循环运行过程中的信息，其中，就包含了记录两类事件的变量，分别是：

- **aeFileEvent类型的指针*events，表示IO事件**。之所以类型名称为aeFileEvent，是因为所有的IO事件都会用文件描述符进行标识；
- **aeTimeEvent类型的指针*timeEventHead，表示时间事件**，即按一定时间周期触发的事件。

此外，aeEventLoop结构体中还有一个**aeFiredEvent类型的指针*fired**，这个并不是一类专门的事件类型，它只是用来记录已触发事件对应的文件描述符信息。

下面的代码显示了Redis中事件循环的结构体定义，你可以看下。

```
typedef struct aeEventLoop {
    …
    aeFileEvent *events; //IO事件数组
    aeFiredEvent *fired; //已触发事件数组
    aeTimeEvent *timeEventHead; //记录时间事件的链表头
    …
    void *apidata; //和API调用接口相关的数据
    aeBeforeSleepProc *beforesleep; //进入事件循环流程前执行的函数
    aeBeforeSleepProc *aftersleep;  //退出事件循环流程后执行的函数
} aeEventLoop;
```

了解了aeEventLoop结构体后，我们再来看下，这个结构体是如何初始化的，这其中就包括了IO事件数组和时间事件链表的初始化。

aeCreateEventLoop函数的初始化操作

因为Redis server在完成初始化后，就要开始运行事件驱动框架的循环流程，所以，aeEventLoop结构体在server.c的initServer函数中，就通过调用**aeCreateEventLoop函数**进行初始化了。这个函数的参数只有一个，是setsize。

下面的代码展示了initServer函数中对aeCreateEventLoop函数的调用。

```
initServer() {
…
//调用aeCreateEventLoop函数创建aeEventLoop结构体，并赋值给server结构的el变量
server.el = aeCreateEventLoop(server.maxclients+CONFIG_FDSET_INCR);
…
}
```

从这里我们可以看到**参数setsize**的大小，其实是由server结构的maxclients变量和宏定义CONFIG_FDSET_INCR共同决定的。其中，maxclients变量的值大小，可以在Redis的配置文件redis.conf中进行定义，默认值是1000。而宏定义CONFIG_FDSET_INCR的大小，等于宏定义CONFIG_MIN_RESERVED_FDS的值再加上96，如下所示，这里的两个宏定义都是在server.h文件中定义的。

```
#define CONFIG_MIN_RESERVED_FDS 32
#define CONFIG_FDSET_INCR (CONFIG_MIN_RESERVED_FDS+96)
```

好了，到这里，你可能有疑问了：aeCreateEventLoop函数的参数setsize，设置为最大客户端数量加上一个宏定义值，可是**这个参数有什么用呢**？这就和aeCreateEventLoop函数具体执行的初始化操作有关了。

接下来，我们就来看下aeCreateEventLoop函数执行的操作，大致可以分成以下三个步骤。

第一步，aeCreateEventLoop函数会创建一个aeEventLoop结构体类型的变量eventLoop。然后，该函数会给eventLoop的成员变量分配内存空间，比如，按照传入的参数setsize，给IO事件数组和已触发事件数组分配相应的内存空间。此外，该函数还会给eventLoop的成员变量赋初始值。

第二步，aeCreateEventLoop函数会调用aeApiCreate函数。aeApiCreate函数封装了操作系统提供的IO多路复用函数，假设Redis运行在Linux操作系统上，并且IO多路复用机制是epoll，那么此时，aeApiCreate函数就会调用epoll_create创建epoll实例，同时会创建epoll_event结构的数组，数组大小等于参数setsize。

这里你需要注意，aeApiCreate函数是把创建的epoll实例描述符和epoll_event数组，保存在了aeApiState结构体类型的变量state，如下所示：

```
typedef struct aeApiState {  //aeApiState结构体定义
    int epfd;   //epoll实例的描述符
    struct epoll_event *events;   //epoll_event结构体数组，记录监听事件
} aeApiState;

static int aeApiCreate(aeEventLoop *eventLoop) {
    aeApiState *state = zmalloc(sizeof(aeApiState));
    ...
    //将epoll_event数组保存在aeApiState结构体变量state中
    state->events = zmalloc(sizeof(struct epoll_event)*eventLoop->setsize);
    ...
    //将epoll实例描述符保存在aeApiState结构体变量state中
    state->epfd = epoll_create(1024);
```

紧接着，aeApiCreate函数把state变量赋值给eventLoop中的apidata。这样一来，eventLoop结构体中就有了epoll实例和epoll_event数组的信息，这样就可以用来基于epoll创建和处理事件了。我一会儿还会给你具体介绍。

```
eventLoop->apidata = state;
```

第三步，aeCreateEventLoop函数会把所有网络IO事件对应文件描述符的掩码，初始化为AE_NONE，表示暂时不对任何事件进行监听。

我把aeCreateEventLoop函数的主要部分代码放在这里，你可以看下。

```
aeEventLoop *aeCreateEventLoop(int setsize) {
    aeEventLoop *eventLoop;
    int i;

    //给eventLoop变量分配内存空间
 if ((eventLoop = zmalloc(sizeof(*eventLoop))) == NULL) goto err;
 //给IO事件、已触发事件分配内存空间
    eventLoop->events = zmalloc(sizeof(aeFileEvent)*setsize);
    eventLoop->fired = zmalloc(sizeof(aeFiredEvent)*setsize);
```

```
    …
    eventLoop->setsize = setsize;
    eventLoop->lastTime = time(NULL);
    //设置时间事件的链表头为NULL
    eventLoop->timeEventHead = NULL;
 …
 //调用aeApiCreate函数，去实际调用操作系统提供的IO多路复用函数
 if (aeApiCreate(eventLoop) == -1) goto err;

    //将所有网络IO事件对应文件描述符的掩码设置为AE_NONE
    for (i = 0; i < setsize; i++)
        eventLoop->events[i].mask = AE_NONE;
    return eventLoop;

    //初始化失败后的处理逻辑，
    err:
    …
}
```

好，那么从aeCreateEventLoop函数的执行流程中，我们其实可以看到以下**两个关键点**：

- 事件驱动框架监听的IO事件数组大小就等于参数setsize，这样决定了和Redis server连接的客户端数量。所以，当你遇到客户端连接Redis时报错“max number of clients reached”，你就可以去redis.conf文件修改maxclients配置项，以扩充框架能监听的客户端数量。
- 当使用Linux系统的epoll机制时，框架循环流程初始化操作，会通过aeApiCreate函数创建epoll_event结构数组，并调用epoll_create函数创建epoll实例，这都是使用epoll机制的准备工作要求，你也可以再回顾下第9讲中对epoll使用的介绍。

到这里，框架就可以创建和处理具体的IO事件和时间事件了。所以接下来，我们就先来了解下IO事件及其处理机制。

IO事件处理

事实上，Redis的IO事件主要包括三类，分别是可读事件、可写事件和屏障事件。

其中，可读事件和可写事件其实比较好理解，也就是对应于Redis实例，我们可以**从客户端读取数据或是向客户端写入数据**。而屏障事件的主要作用是用来**反转事件的处理顺序**。比如在默认情况下，Redis会先给客户端返回结果，但是如果面临需要把数据尽快写入磁盘的情况，Redis就会用到屏障事件，把写数据和回复客户端的顺序做下调整，先把数据落盘，再给客户端回复。

我在上节课给你介绍过，在Redis源码中，IO事件的数据结构是aeFileEvent结构体，IO事件的创建是通过aeCreateFileEvent函数来完成的。下面的代码展示了aeFileEvent结构体的定义，你可以再回顾下：

```
typedef struct aeFileEvent {
    int mask; //掩码标记，包括可读事件、可写事件和屏障事件
    aeFileProc *rfileProc;   //处理可读事件的回调函数
    aeFileProc *wfileProc;   //处理可写事件的回调函数
    void *clientData;  //私有数据
} aeFileEvent;
```

而对于aeCreateFileEvent函数来说，在上节课我们已经了解了它是通过aeApiAddEvent函数来完成事件注册的。那么接下来，我们再从代码级别看下它是如何执行的，这可以帮助我们更加透彻地理解，事件驱动框架对IO事件监听是如何基于epoll机制对应封装的。

IO事件创建

首先，我们来看aeCreateFileEvent函数的原型定义，如下所示：

```
int aeCreateFileEvent(aeEventLoop *eventLoop, int fd, int mask, aeFileProc *proc, void *clientData)
```

这个函数的参数有5个，分别是循环流程结构体*eventLoop、IO事件对应的文件描述符fd、事件类型掩码mask、事件处理回调函数*proc，以及事件私有数据*clientData。

因为循环流程结构体*eventLoop中有IO事件数组，这个数组的元素是aeFileEvent类型，所以，每个数组元素都对应记录了一个文件描述符（比如一个套接字）相关联的监听事件类型和回调函数。

aeCreateFileEvent函数会先根据传入的文件描述符fd，在eventLoop的IO事件数组中，获取该描述符关联的IO事件指针变量*fe，如下所示：

```
aeFileEvent *fe = &eventLoop->events[fd];
```

紧接着，aeCreateFileEvent函数会调用aeApiAddEvent函数，添加要监听的事件：

```
if (aeApiAddEvent(eventLoop, fd, mask) == -1)
   return AE_ERR;
```

aeApiAddEvent函数实际上会调用操作系统提供的IO多路复用函数，来完成事件的添加。我们还是假设Redis实例运行在使用epoll机制的Linux上，那么aeApiAddEvent函数就会调用epoll_ctl函数，添加要监听的事件。我在第9讲中其实已经给你介绍过epoll_ctl函数，这个函数会接收4个参数，分别是：

- epoll实例；
- 要执行的操作类型（是添加还是修改）；
- 要监听的文件描述符；
- epoll_event类型变量。

那么，**这个调用过程是如何准备epoll_ctl函数需要的参数，从而完成执行的呢？**

首先，epoll实例是我刚才给你介绍的aeCreateEventLoop函数，它是通过调用aeApiCreate函数来创建的，保存在了eventLoop结构体的apidata变量中，类型是aeApiState。所以，aeApiAddEvent函数会先获取该

变量，如下所示：

```
static int aeApiAddEvent(aeEventLoop *eventLoop, int fd, int mask) {
    //从eventLoop结构体中获取aeApiState变量，里面保存了epoll实例
 aeApiState *state = eventLoop->apidata;
    ...
 }
```

其次，对于要执行的操作类型的设置，aeApiAddEvent函数会根据传入的文件描述符fd，在eventLoop结构体中IO事件数组中查找该fd。因为IO事件数组的每个元素，都对应了一个文件描述符，而该数组初始化时，每个元素的值都设置为了AE_NONE。

所以，如果要监听的文件描述符fd在数组中的类型不是AE_NONE，则表明该描述符已做过设置，那么操作类型就是修改操作，对应epoll机制中的宏定义EPOLL_CTL_MOD。否则，操作类型就是添加操作，对应epoll机制中的宏定义EPOLL_CTL_ADD。这部分代码如下所示：

```
//如果文件描述符fd对应的IO事件已存在，则操作类型为修改，否则为添加
 int op = eventLoop->events[fd].mask == AE_NONE ?
            EPOLL_CTL_ADD : EPOLL_CTL_MOD;
```

第三，epoll_ctl函数需要的监听文件描述符，就是aeApiAddEvent函数接收到的参数fd。

最后，epoll_ctl函数还需要一个epoll_event类型变量，因此aeApiAddEvent函数在调用epoll_ctl函数前，会新创建epoll_event类型**变量ee。**然后，aeApiAddEvent函数会设置变量ee中的监听事件类型和监听文件描述符。

aeApiAddEvent函数的参数mask，表示的是要监听的事件类型掩码。所以，aeApiAddEvent函数会根据掩码值是可读（AE_READABLE）或可写（AE_WRITABLE）事件，来设置ee监听的事件类型是EPOLLIN还是EPOLLOUT。这样一来，Redis事件驱动框架中的读写事件就能够和epoll机制中的读写事件对应上来。下面的代码展示了这部分逻辑，你可以看下。

```
…
struct epoll_event ee = {0}; //创建epoll_event类型变量
…
//将可读或可写IO事件类型转换为epoll监听的类型EPOLLIN或EPOLLOUT
if (mask & AE_READABLE) ee.events |= EPOLLIN;
if (mask & AE_WRITABLE) ee.events |= EPOLLOUT;
ee.data.fd = fd;  //将要监听的文件描述符赋值给ee
…
```

好了，到这里，aeApiAddEvent函数就准备好了epoll实例、操作类型、监听文件描述符以及epoll_event类型变量，然后，它就会调用epoll_ctl开始实际创建监听事件了，如下所示：

```
static int aeApiAddEvent(aeEventLoop *eventLoop, int fd, int mask) {
...
//调用epoll_ctl实际创建监听事件
if (epoll_ctl(state->epfd,op,fd,&ee) == -1) return -1;
return 0;
}
```

了解了这些代码后，我们可以学习到事件驱动框架是如何基于epoll，封装实现了IO事件的创建。那么，在Redis server启动运行后，最开始监听的IO事件是可读事件，对应于客户端的连接请求。具体是initServer函数调用了aeCreateFileEvent函数，创建可读事件，并设置回调函数为acceptTcpHandler，用来处理客户端连接。这部分内容，你也可以再回顾下第10讲。

接下来，我们再来看下一旦有了客户端连接请求后，IO事件具体是如何处理的呢？

读事件处理

当Redis server接收到客户端的连接请求时，就会使用注册好的**acceptTcpHandler函数**进行处理。

acceptTcpHandler函数是在networking.c文件中，它会接受客户端连接，并创建已连接套接字cfd。然后，acceptCommonHandler函数（在networking.c文件中）会被调用，同时，刚刚创建的已连接套接字cfd会作为参数，传递给acceptCommonHandler函数。

acceptCommonHandler函数会调用createClient函数（在networking.c文件中）创建客户端。而在createClient函数中，我们就会看到，aeCreateFileEvent函数被再次调用了。

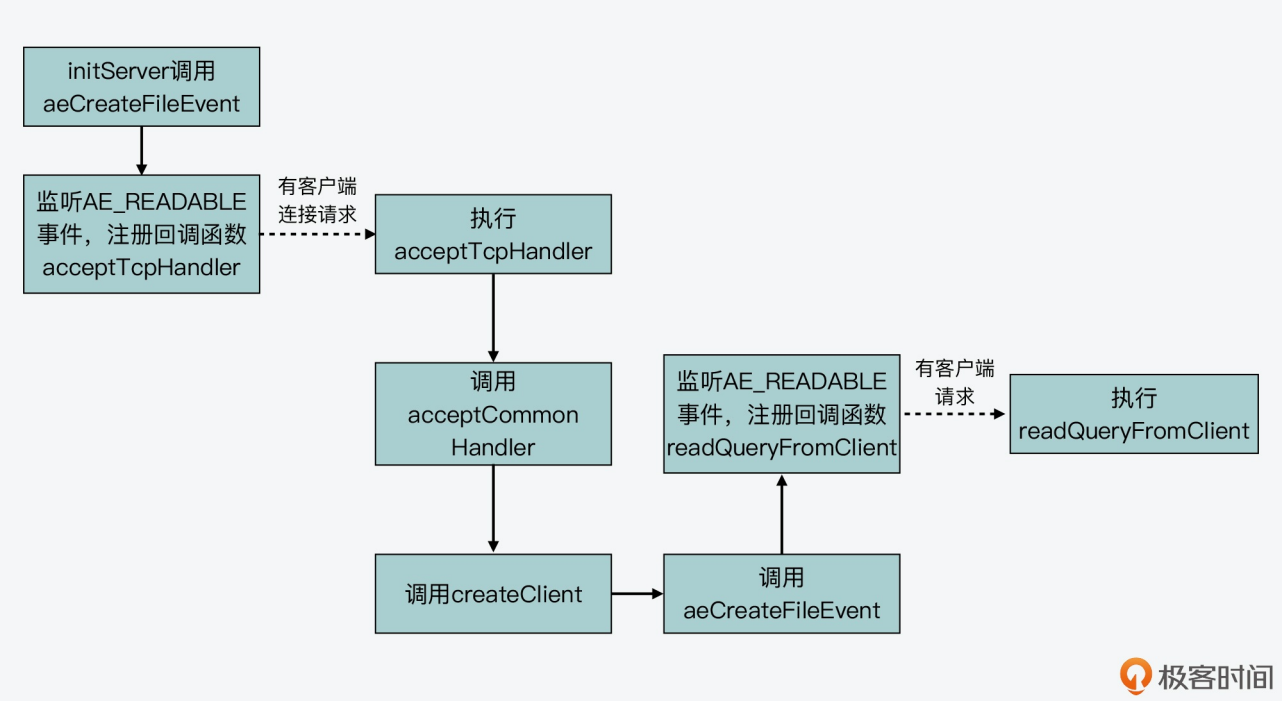
此时，aeCreateFileEvent函数会针对已连接套接字上，创建监听事件，类型为AE_READABLE，回调函数是readQueryFromClient（在networking.c文件中）。

好了，到这里，事件驱动框架就增加了对一个客户端已连接套接字的监听。一旦客户端有请求发送到server，框架就会回调readQueryFromClient函数处理请求。这样一来，客户端请求就能通过事件驱动框架进行处理了。

下面代码展示了createClient函数调用aeCreateFileEvent的过程，你可以看下。

```
client *createClient(int fd) {
…
if (fd != -1) {
        …
        //调用aeCreateFileEvent，监听读事件，对应客户端读写请求，使用readQueryFromclient回调函数处理
        if (aeCreateFileEvent(server.el,fd,AE_READABLE,
            readQueryFromClient, c) == AE_ERR)
        {
            close(fd);
            zfree(c);
            return NULL;
        } }
…
}
```

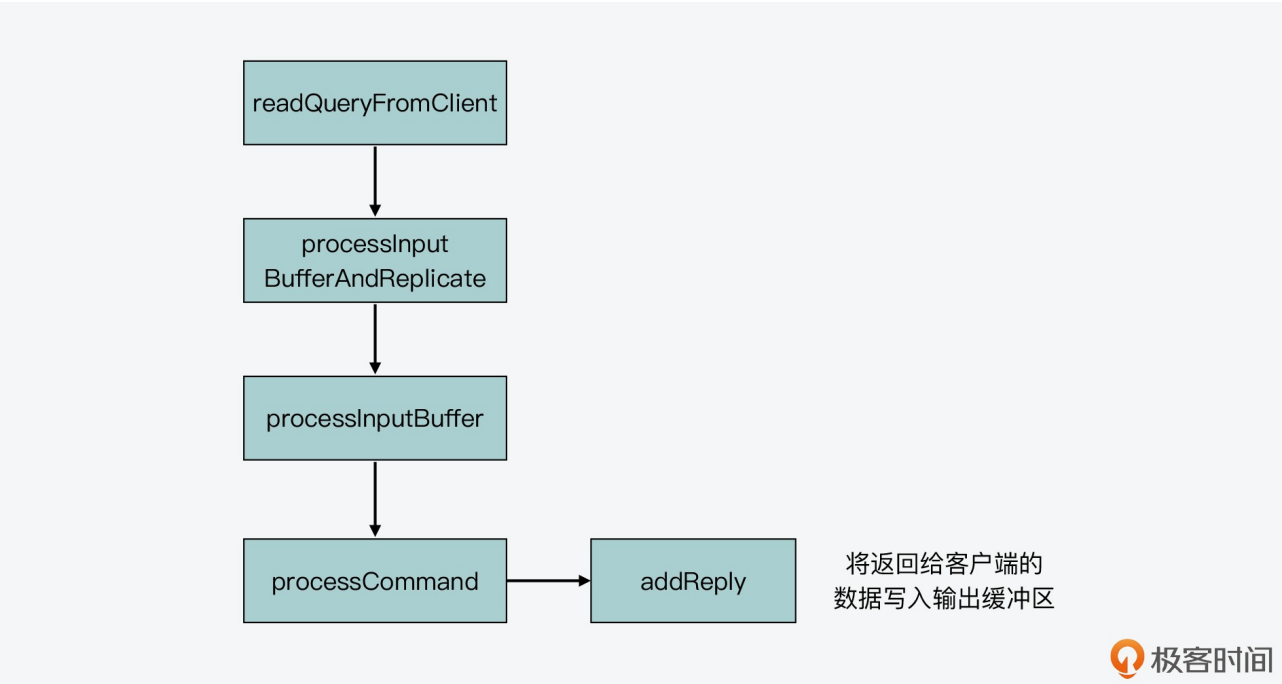
为了便于你掌握从监听客户端连接请求到监听客户端常规读写请求的事件创建过程，我画了下面这张图，你可以看下。



了解了事件驱动框架中的读事件处理之后，我们再来看下写事件的处理。

写事件处理

Redis实例在收到客户端请求后，会在处理客户端命令后，将要返回的数据写入客户端输出缓冲区。下图就展示了这个过程的函数调用逻辑：

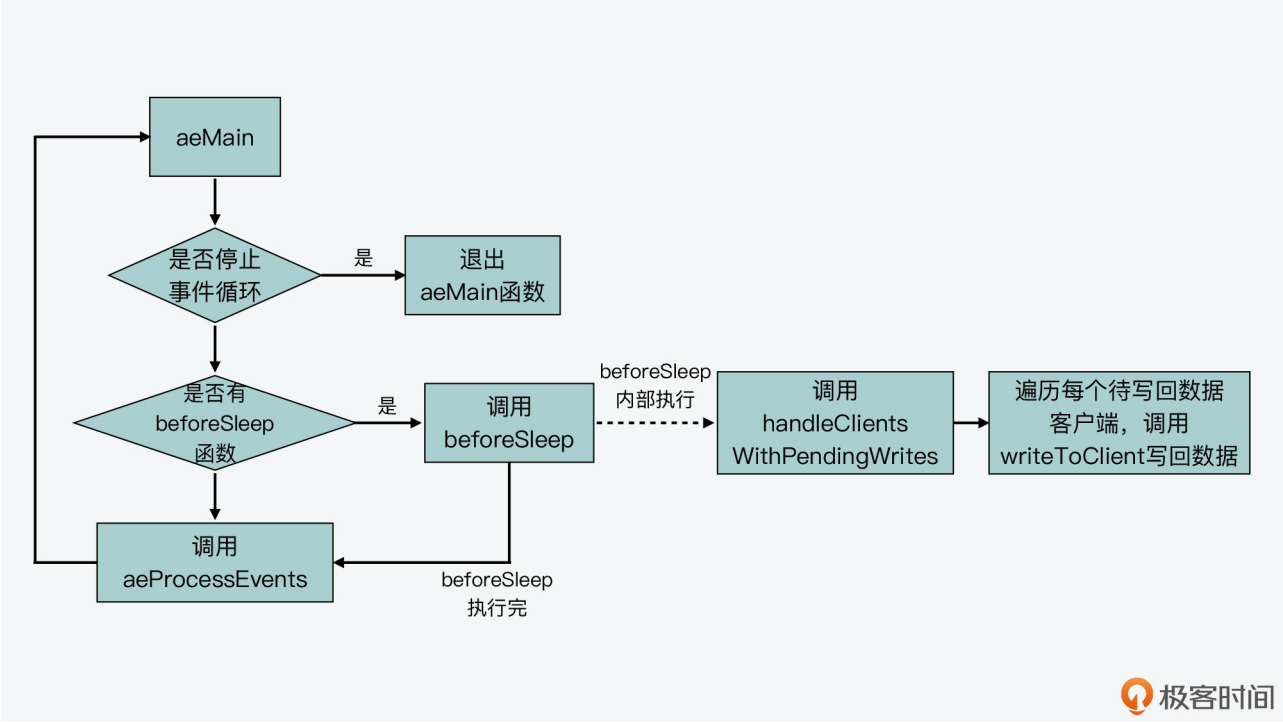


而在Redis事件驱动框架每次循环进入事件处理函数前，也就是在框架主函数aeMain中调用aeProcessEvents，来处理监听到的已触发事件或是到时的时间事件之前，都会调用server.c文件中的**beforeSleep函数**，进行一些任务处理，这其中就包括了调用handleClientsWithPendingWrites函数，它会将Redis sever客户端缓冲区中的数据写回客户端。

下面给出的代码是事件驱动框架的主函数aeMain。在该函数每次调用aeProcessEvents函数前，就会调用beforeSleep函数，你可以看下。

```
void aeMain(aeEventLoop *eventLoop) {
    eventLoop->stop = 0;
 while (!eventLoop->stop) {
     //如果beforeSleep函数不为空，则调用beforeSleep函数
        if (eventLoop->beforesleep != NULL)
            eventLoop->beforesleep(eventLoop);
        //调用完beforeSleep函数，再处理事件
        aeProcessEvents(eventLoop, AE_ALL_EVENTS|AE_CALL_AFTER_SLEEP);
    }
}
```

这里你要知道，beforeSleep函数调用的handleClientsWithPendingWrites函数，会遍历每一个待写回数据的客户端，然后调用writeToClient函数，将客户端输出缓冲区中的数据写回。下面这张图展示了这个流程，你可以看下。



不过，如果输出缓冲区的数据还没有写完，此时，handleClientsWithPendingWrites函数就会**调用aeCreateFileEvent函数，创建可写事件**，并设置回调函数sendReplyToClient。sendReplyToClient函数里面会调用writeToClient函数写回数据。

下面的代码展示了handleClientsWithPendingWrite函数的基本流程，你可以看下。

```
int handleClientsWithPendingWrites(void) {
    listIter li;
 listNode *ln;
 …
    //获取待写回的客户端列表
 listRewind(server.clients_pending_write,&li);
 //遍历每一个待写回的客户端
```

```
 while((ln = listNext(&li))) {
    client *c = listNodeValue(ln);
    …
    //调用writeToClient将当前客户端的输出缓冲区数据写回
    if (writeToClient(c->fd,c,0) == C_ERR) continue;
    //如果还有待写回数据
    if (clientHasPendingReplies(c)) {
              int ae_flags = AE_WRITABLE;
              //创建可写事件的监听，以及设置回调函数
               if (aeCreateFileEvent(server.el, c->fd, ae_flags,
                  sendReplyToClient, c) == AE_ERR)
              {
                     …
              }
   } }
}
```

好了，我们刚才了解的是读写事件对应的回调处理函数。实际上，为了能及时处理这些事件，Redis事件驱动框架的aeMain函数还会循环**调用aeProcessEvents函数，来检测已触发的事件，并调用相应的回调函数进行处理。**

从aeProcessEvents函数的代码中，我们可以看到该函数会调用aeApiPoll函数，查询监听的文件描述符中，有哪些已经就绪。一旦有描述符就绪，aeProcessEvents函数就会根据事件的可读或可写类型，调用相应的回调函数进行处理。aeProcessEvents函数调用的基本流程如下所示：

```
int aeProcessEvents(aeEventLoop *eventLoop, int flags){
…
//调用aeApiPoll获取就绪的描述符
numevents = aeApiPoll(eventLoop, tvp);
…
for (j = 0; j < numevents; j++) {
 aeFileEvent *fe = &eventLoop->events[eventLoop->fired[j].fd];
 …
    //如果触发的是可读事件，调用事件注册时设置的读事件回调处理函数
 if (!invert && fe->mask & mask & AE_READABLE) {
       fe->rfileProc(eventLoop,fd,fe->clientData,mask);
                fired++;
 }
    //如果触发的是可写事件，调用事件注册时设置的写事件回调处理函数
 if (fe->mask & mask & AE_WRITABLE) {
                if (!fired || fe->wfileProc != fe->rfileProc) {
                    fe->wfileProc(eventLoop,fd,fe->clientData,mask);
                    fired++;
                }
             }
 …
 } }
 …
}
```

到这里，我们就了解了IO事件的创建函数aeCreateFileEvent，以及在处理客户端请求时对应的读写事件和它们的处理函数。那么接下来，我们再来看看事件驱动框架中的时间事件是怎么创建和处理的。

时间事件处理

其实，相比于IO事件有可读、可写、屏障类型，以及不同类型IO事件有不同回调函数来说，时间事件的处理就比较简单了。下面，我们就来分别学习下它的定义、创建、回调函数和触发处理。

时间事件定义

首先，我们来看下时间事件的结构体定义，代码如下所示：

```
typedef struct aeTimeEvent {
    long long id; //时间事件ID
    long when_sec; //事件到达的秒级时间戳
    long when_ms; //事件到达的毫秒级时间戳
    aeTimeProc *timeProc; //时间事件触发后的处理函数
    aeEventFinalizerProc *finalizerProc;  //事件结束后的处理函数
    void *clientData; //事件相关的私有数据
    struct aeTimeEvent *prev;  //时间事件链表的前向指针
    struct aeTimeEvent *next;  //时间事件链表的后向指针
} aeTimeEvent;
```

时间事件结构体中主要的变量，包括以秒记录和以毫秒记录的时间事件触发时的时间戳when_sec和when_ms，以及时间事件触发后的处理函数*timeProc。另外，在时间事件的结构体中，还包含了前向和后向指针*prev和*next，这表明**时间事件是以链表的形式组织起来的**。

在了解了时间事件结构体的定义以后，我们接着来看下，时间事件是如何创建的。

时间事件创建

与IO事件创建使用aeCreateFileEvent函数类似，**时间事件的创建函数是aeCreateTimeEvent函数**。这个函数的原型定义如下所示：

```
long long aeCreateTimeEvent(aeEventLoop *eventLoop, long long milliseconds, aeTimeProc *proc, void *clientD
```

在它的参数中，有两个需要我们重点了解下，以便于我们理解时间事件的处理。一个是**milliseconds**，这是所创建时间事件的触发时间距离当前时间的时长，是用毫秒表示的。另一个是***proc**，这是所创建时间事件触发后的回调函数。

aeCreateTimeEvent函数的执行逻辑不复杂，主要就是创建一个时间事件的**变量te**，对它进行初始化，并把它插入到框架循环流程结构体eventLoop中的时间事件链表中。在这个过程中，aeCreateTimeEvent函数会**调用aeAddMillisecondsToNow函数**，根据传入的milliseconds参数，计算所创建时间事件具体的触发时间戳，并赋值给te。

实际上，Redis server在初始化时，除了创建监听的IO事件外，也会调用aeCreateTimeEvent函数创建时间事件。下面代码显示了initServer函数对aeCreateTimeEvent函数的调用：

```
initServer() {
…
//创建时间事件
if (aeCreateTimeEvent(server.el, 1, serverCron, NULL, NULL) == AE_ERR){
… //报错信息
}
}
```

从代码中，我们可以看到，**时间事件触发后的回调函数是serverCron**。所以接下来，我们就来了解下serverCron函数。

时间事件回调函数

serverCron函数是在server.c文件中实现的。**一方面**，它会顺序调用一些函数，来实现时间事件被触发后，执行一些后台任务。比如，serverCron函数会检查是否有进程结束信号，若有就执行server关闭操作。serverCron会调用databaseCron函数，处理过期key或进行rehash等。你可以参考下面给出的代码：

```
...
//如果收到进程结束信号，则执行server关闭操作
 if (server.shutdown_asap) {
        if (prepareForShutdown(SHUTDOWN_NOFLAGS) == C_OK) exit(0);
        ...
 }
...
clientCron();  //执行客户端的异步操作
databaseCron(); //执行数据库的后台操作
...
```

另一方面，serverCron函数还会以不同的频率周期性执行一些任务，这是通过执行宏run_with_period来实现的。

run_with_period宏定义如下，该宏定义会根据Redis实例配置文件redis.conf中定义的hz值，来判断参数_ms_表示的时间戳是否到达。一旦到达，serverCron就可以执行相应的任务了。

```
#define run_with_period(_ms_) if ((_ms_ <= 1000/server.hz) || !(server.cronloops%((_ms_)/(1000/server.hz))))
```

比如，serverCron函数中会以1秒1次的频率，检查AOF文件是否有写错误。如果有的话，serverCron就会调用flushAppendOnlyFile函数，再次刷回AOF文件的缓存数据。下面的代码展示了这一周期性任务：

```
serverCron() {
   …
   //每1秒执行1次，检查AOF是否有写错误
   run_with_period(1000) {
        if (server.aof_last_write_status == C_ERR)
            flushAppendOnlyFile(0);
```

```
    }
   …
}
```

如果你想了解更多的周期性任务，可以再详细阅读下serverCron函数中，以run_with_period宏定义包含的代码块。

好了，了解了时间事件触发后的回调函数serverCron，我们最后来看下，时间事件是如何触发处理的。

时间事件的触发处理

其实，时间事件的检测触发比较简单，事件驱动框架的aeMain函数会循环调用aeProcessEvents函数，来处理各种事件。而aeProcessEvents函数在执行流程的最后，会**调用processTimeEvents函数处理相应到时的任务**。

```
aeProcessEvents(){
…
//检测时间事件是否触发
if (flags & AE_TIME_EVENTS)
        processed += processTimeEvents(eventLoop);
…
}
```

那么，具体到proecessTimeEvent函数来说，它的基本流程就是从时间事件链表上逐一取出每一个事件，然后根据当前时间判断该事件的触发时间戳是否已满足。如果已满足，那么就调用该事件对应的回调函数进行处理。这样一来，周期性任务就能在不断循环执行的aeProcessEvents函数中，得到执行了。

下面的代码显示了processTimeEvents函数的基本流程，你可以再看下。

```
static int processTimeEvents(aeEventLoop *eventLoop) {
...
te = eventLoop->timeEventHead;  //从时间事件链表中取出事件
while(te) {
   ...
  aeGetTime(&now_sec, &now_ms);  //获取当前时间
  if (now_sec > te->when_sec || (now_sec == te->when_sec && now_ms >= te->when_ms))   //如果当前时间已经满足当前
  {
     ...
    retval = te->timeProc(eventLoop, id, te->clientData); //调用注册的回调函数处理
    ...
  }
  te = te->next;   //获取下一个时间事件
  ...
}
```

小结

这节课，我给你介绍了Redis事件驱动框架中的两类事件：IO事件和时间事件。

对于IO事件来说，它可以进一步分成可读、可写和屏障事件。因为可读、可写事件在Redis和客户端通信处理请求过程中使用广泛，所以今天我们重点学习了这两种IO事件。当Redis server创建Socket后，就会注册可读事件，并使用acceptTCPHandler回调函数处理客户端的连接请求。

当server和客户端完成连接建立后，server会在已连接套接字上监听可读事件，并使用readQueryFromClient函数处理客户端读写请求。这里，你需要再注意下，**无论客户端发送的请求是读或写操作，对于server来说，都是要读取客户端的请求并解析处理**。所以，server在客户端的已连接套接字上注册的是可读事件。

而当实例需要向客户端写回数据时，实例会在事件驱动框架中注册可写事件，并使用sendReplyToClient作为回调函数，将缓冲区中数据写回客户端。我总结了一张表格，以便你再回顾下IO事件和相应套接字、回调函数的对应关系。

注册事件类型	对应套接字	回调函数	用途
AE_READABLE	监听套接字	acceptTCPHandler	接受客户端连接请求，和客户端建立连接
AE_READABLE	已连接套接字	readQueryFromClient	接收并处理客户端读写请求
AE_WRITABLE	已连接套接字	sendReplyToClient	向客户端返回数据

然后，对于时间事件来说，它主要是用于在事件驱动框架中注册一些周期性执行的任务，以便Redis server进行后台处理。时间事件的回调函数是serverCron函数，你可以做进一步阅读了解其中的具体任务。

好了，从第9讲开始，我用了3节课，向你介绍Redis事件驱动框架的运行机制，本质上来说，事件驱动框架是基于操作系统提供的IO多路复用机制进行了封装，并加上了时间事件的处理。这是一个非常经典的事件框架实现，我希望你可以学习并掌握好它，然后用在你自己的系统开发中。

每课一问

已知，Redis事件驱动框架的aeApiCreate、aeApiAddEvent等等这些函数，是对操作系统提供的IO多路复用函数进行了封装，具体的IO多路复用函数分别是在ae_epoll.c，ae_evport.c，ae_kqueue.c，ae_select.c四个代码文件中定义的。那么你知道，Redis在调用aeApiCreate，aeApiAddEvent这些函数时，是根据什么条件来决定，具体调用哪个文件中的IO多路复用函数的吗？

精选留言：

- Kaito 2021-08-19 01:54:46
  1、Redis 事件循环主要处理两类事件：文件事件、时间事件

  - 文件事件包括：client 发起新连接、client 向 server 写数据、server 向 client 响应数据

- 时间事件：Redis 的各种定时任务（主线程中执行）

2、Redis 在启动时，会创建 aeEventLoop，初始化 epoll 对象，监听端口，之后会注册文件事件、时间事件：

- 文件事件：把 listen socket fd 注册到 epoll 中，回调函数是 acceptTcpHandler（新连接事件）
- 时间事件：把 serverCron 函数注册到 aeEventLoop 中，并指定执行频率

3、Redis Server 启动后，会启动一个死循环，持续处理事件（ae.c 的 aeProcessEvents 函数）

4、有文件事件（网络 IO），则优先处理。例如，client 到 server 的新连接，会调用 acceptTcpHandler 函数，之后会注册读事件 readQueryFromClient 函数，client 发给 server 的数据，都会在这个函数处理，这个函数会解析 client 的数据，找到对应的 cmd 函数执行

5、cmd 逻辑执行完成后，server 需要写回数据给 client，会先把响应数据写到对应 client 的 内存 buffer 中，在下一次处理 IO 事件之前，Redis 会把每个 client 的 buffer 数据写到 client 的 socket 中，给 client 响应

6、如果响应给 client 的数据过多，则会分多次发送，待发送的数据会暂存到 buffer，然后会向 epoll 注册回调函数 sendReplyToClient，待 socket 可写时，继续调用回调函数向 client 写回剩余数据

7、在这个死循环中处理每次事件时，都会先检查一下，时间事件是否需要执行，因为之前已经注册好了时间事件的回调函数 + 执行频率，所以在执行 aeApiPoll 时，timeout 就是定时任务的周期，这样即使没有 IO 事件，epoll_wait 也可以正常返回，此时就可以执行一次定时任务 serverCron 函数，这样就可以在一个线程中就完成 IO 事件 + 定时任务的处理

课后题：Redis 在调用 aeApiCreate、aeApiAddEvent 这些函数时，是根据什么条件来决定，具体调用哪个文件中的 IO 多路复用函数的？

在 ae.c 中，根据不同平台，首先定义好了要导入的封装好的 IO 多路复用函数，每个平台对应的文件中都定义了 aeApiCreate、aeApiAddEvent 这类函数，在执行时就会执行对应平台的函数逻辑。

```
// ae.c
#ifdef HAVE_EVPORT
#include "ae_evport.c" // Solaris
#else
#ifdef HAVE_EPOLL
#include "ae_epoll.c" // Linux
#else
#ifdef HAVE_KQUEUE
#include "ae_kqueue.c" // MacOS
#else
#include "ae_select.c" // Windows
#endif
#endif
#endif
```

[4赞]

- Darren 2021-08-19 00:17:42
  通过不同的操作系统，include不同的头文件。

在ae.c中
/* Include the best multiplexing layer supported by this system.
* The following should be ordered by performances, descending. */
#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

在config.h中，根据操作系统，设置相关的值

/* Test for polling API */
#ifdef __linux__
#define HAVE_EPOLL 1
#endif

#if (defined(__APPLE__) && defined(MAC_OS_X_VERSION_10_6)) || defined(__FreeBSD__) || defined(__OpenBSD__) || defined (__NetBSD__)
#define HAVE_KQUEUE 1
#endif

#ifdef __sun
#include <sys/feature_tests.h>
#ifdef _DTRACE_VERSION
#define HAVE_EVPORT 1
#endif
#endif [3赞]

- 徐志超-Klaus 2021-08-20 16:57:19
  请问这个课程能开个微信讨论群吗？这样不仅能大大提高学习氛围和学习效率，还能提高宣传力度。不过还是先问问老师意见？

- 曾轼麟 2021-08-20 13:55:04
  感谢老师的文章，一样首先回答老师的问题：Redis事件驱动框架是如何区分文件的？
  答：
  首先，观察可以发现这几份文件都是命名了相同的代码名字和结构体，例如都有aeApiState结构体，都有aeApiCreate，aeApiResize，aeApiAddEvent，aeApiPoll等方法，那么这里就可以发现其实无论我们使用那一份文件，都不会对调用方造成影响，其次我们观察发现在ae.c文件头部中有这样一段代码：

  #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
```

这段代码大致就是根据当前操作系统类型来决定使用那一份头文件，而对操作系统识别在config.h这份文件中，主要划分了（HAVE_EVPORT，HAVE_EPOLL，HAVE_KQUEUE，和默认ae_select.c）这四种类型，所以大致逻辑如下：
1、config.h中确定当前操作系统类型并打上标记
2、通过标记选择对应的IO复用文件ae_xxxx.c
3、编译期间使用对应的文件进行编译
4、通过统一实现的方法调用例如：aeApiPoll

所以从这里也能看出，如果我们想强行选择select这种方式进行IO多路复用，可以直接修改ae.c上面的这段宏定义

最后顺便拓展一下本篇文章外的内容，本篇文章除了老师提到的事件驱动框架的核心方法外，老师还提到了几个方法，我个人觉得也比较重要可以深度阅读一下其在Redis项目中出现的位置和实现：
1、readQueryFromClient
2、beforeSleep
3、handleClientsWithPendingWrites
4、writeToClient

原因：
1、readQueryFromClient主要是和querybuf打交道（对应读事件），里面涉及到RESP编解码可以深度阅读一下（个人发现Redis处理粘包拆包的方式很独特）。
2、beforeSleep和afterSleep是搭配的（在aeMain的大循环中每次都被调用），通过这两个钩子函数Redis实现了很多主干路的功能，和一些分治思想的功能。
3、handleClientsWithPendingWrites可以关注一下其调用方handleClientsWithPendingWritesUsingThreads函数，在这两个方法中都要调用writeToClient方法（对应写事件），而在他们中通过一种巧妙的方式实现了Redis6 IO多线程的方案（可以先预习），handleClientsWithPendingWrites是在只有一个线程或者禁用IO线程的时候使用的。

- 一步 2021-08-20 10:33:27
事件循环结构体中的 aeEventLoop 的 io 事件 aeFileEvent *events 中的 文件描述符 是怎么和外面传入的对应上的?

我看 创建监听连接事件的时候 ， 直接就把文件描述符传入进入了 server.ipfd[j] 这样怎么保证一定能找到对应的上?

- 徐志超-Klaus 2021-08-19 11:04:14
刚买课程我就来这里评论了，希望被作者翻牌。我在C语言这块迷路了，我自学看完了《C语言程序设计现代方法第2版》，然后我发现看别人的代码还是有很多不懂的，比如有些关键字书本里没提到，比如我

还有些疑问C语言的并发编程怎么写，也不知道接下来该上哪里查资料，因为又要学其他东西，导致我又放弃了一个月的C语言。我现在该怎么办

- Milittle 2021-08-19 10:31:59
  总结一下ae处理框架：
  处理IO事件的一些感悟：
  1. 创建eventloop（这个地方还没有建立socket服务器监听）(initServer->aeCreateEventLoop)
  2. 然后接着使用listenToPort将所有需要监听的端口进行socket建立以及bind，然后listen。（initServer->listenToPort->anetTcpServer->_anetTcpServer->anetListen）将普通的监听fd放在server.ipfd，将tls的监听fd放在server.tlsfd
  2. 有了这个监听的fd，就得注册到IO多路复用的事件集合中，让后续client可以连接，（initServer->createSocketAcceptHandler(记住这个地方传入了回调函数acceptTcpHandler，当有客户端连接以后，调的就是它)->aeCreateFileEvent->aeApiAddEvent(底层添加事件的调用)）。
  3. 把这个监听的scoket放进去，我们就等着连接事件的到来，main函数已经启动的时候调用了aeMain函数，一直在循环监听（main->aeMain->aeProcessEvents->aeApiPoll->(这里如果有事件，就会调用事件回调，rfileProc or wfileProc， 这两个函数是FilteEvent的回调函数注册，刚才的acceptTcpHandler就注册在这个上面了)）当我们收到客户端监听事件以后，开始在accpetTcpHandler处理任务了。
  4. （acceptTcpHandler->anetTcpAccept(这个函数玩命把这个监听的socket accept一下，然后就返回了我们想要的conn fd叫cfd)->acceptCommonHandler）。
  5. 第四步，最后一个函数就开始处理我们这个链接的cfd的事件注册，重点在调用这个函数createClient，从这函数瞅了半天，发现没有createFileEvent，原来在connSetReadHandler这里，老牛逼了，这些读写事件都被提前注册在conn这个实例里面了，叫type的这个变量。到了connSetReadHandler这个函数里面，那就是该注册连接的cfd了，然后你看到，这个函数传入了readQueryFromClient（这个回调，意思是说，如果这个fd被触发了，那么我们就调它么？显然你从conn中得知，不是的，很遗憾，那么是啥呢，你得乖乖看CT_Socket这个变量，这个玩意就是初始化给了conn的type，然后connSetReadHandler这个函数调用的就是connSocketSetReadHandler它，隐藏在CT_Socket这里面，妈呀，挺绕的。），那么接下来我们看看connSocketSetReadHandler这个函数里面到底干了个啥，不就是注册个fd的事件么，至于这么麻烦么，进去一看，发现真有aeCreateFileEvent，果然，不负真心，上面说的传入的readQueryFromClient，不是fd被触发以后的回调，是因为aeCreateFileEvent这个里面传入的是connSocketEventHandler这个回调，也就是说，我们在后续创建conn以后的读写事件都是在connSocketEventHandler这个函数调用的。
  6. 那么我们假设现在已经有了一个连接的事件，客户端发来一条消息，那么就会触发这个cfd，进而回调在这个函数上，进去探探，搞了半天，里面就调了一个conn的读事件，和写事件（还有那个屏障，写法也是牛逼，这个函数可以理解为proxy，触发的是上层注册进来的读事件或者写事件，就是第五步，注册在conn里面的read_handler or write_handler），还得看函数readQueryFromClient，将数据读取，然后放在client里面，然后processInputBuffer处理实际的命令请求，processCommandAndResetClient->processCommand(各种reject不符合条件的命令)->call(核心了)。
  7. 重点来了，c->cmd->proc(c); call的一行调用，看struct redisCommand的定义，注册在server.c的redisCommandTable（查找命令：c->cmd = c->lastcmd = lookupCommand(c->argv[0]->ptr); 在processCommand这个函数的某一行）。
  8. 我们把它串起来了。
  备注：本人走读的代码都是最新版本，望知。细节慢慢在补充吧。

- Milittle 2021-08-19 09:28:39
  由宏定义去确定的 ecport->epoll->kqueue->select

  性能逐级递减。这个包含在ae.c头部包含c文件的