redis-ae

1 ae介绍

`ae` 是 Redis 事件处理器的核心模块，全称为 Asynchronous Events（异步事件）。它是 Redis 用来处理各种异步 I/O 事件（如网络 I/O、定时器等）的**统一接口**，支持多种不同的 I/O 多路复用技术，包括 epoll、kqueue、select 等，并提供了一些高层次的事件处理接口，如文件事件、定时器事件等，让 Redis 能够方便地进行事件的等待、分发和处理。`ae` **实现了 I/O 多路复用的底层细节，并将其抽象成通用的事件处理接口，使得 Redis 的业务代码可以更加简单和易于维护。**

2 aeFileEvent

`aeFileEvent`是一个文件事件结构体，用于保存文件事件相关的信息，包括文件描述符、监听事件类型和处理该事件的函数指针等。它的定义如下：

```
/* File event structure */
typedef struct aeFileEvent {
    int mask; /* one of AE_(READABLE|WRITABLE|BARRIER) */
    aeFileProc *rfileProc;
    aeFileProc *wfileProc;
    void *clientData;
} aeFileEvent;
```

其中，`mask`表示该事件的类型掩码，可能是以下值的任意组合：

- `AE_READABLE`：表示可读事件。
- `AE_WRITABLE`：表示可写事件。
- `AE_BARRIER`：表示本次事件处理会涉及到大量计算或IO操作，需要防止其他同样类型的事件插入到当前事件之间。

`rfileProc`和`wfileProc`是对应读写事件处理函数的函数指针，类型为`aeFileProc`，定义如下：

```
typedef void aeFileProc(struct aeEventLoop *eventLoop, int fd,
void *clientData, int mask);
```

最后，`clientData`是一个指针，指向和该事件相关的客户端数据。

3 aeTimeEvent

```
/* Time event structure */
typedef struct aeTimeEvent {
    long long id; /* time event identifier. */
    monotime when;
    aeTimeProc *timeProc;
    aeEventFinalizerProc *finalizerProc;
    void *clientData;
    struct aeTimeEvent *prev;
    struct aeTimeEvent *next;
    int refcount; /* refcount to prevent timer events from being
            * freed in recursive time event calls. */
} aeTimeEvent;
```

`aeTimeEvent`是Redis事件处理框架中的一种类型，它用于实现基于时间的事件。

一个`aeTimeEvent`结构体包含以下字段：

- id：时间事件的唯一标识符，是一个长整型数值。
- when：时间事件的执行时间，以纳秒为单位。
- timeProc：时间事件的处理函数指针，当事件达到执行时间时会调用该函数。
- finalizerProc：时间事件的终结函数指针，当事件被删除时会调用该函数。
- clientData：指向与时间事件相关的客户端数据的指针。
- prev：指向前一个时间事件结构体的指针。
- next：指向下一个时间事件结构体的指针。
- refcount：用于计数，防止在递归的时间事件调用中释放定时器事件。

Redis中的时间事件用于在指定时间执行某些操作，常见的应用场景包括延时任务、定时任务等。当一个时间事件被添加到Redis的事件循环中时，事件循环会按照时间顺序排列这些事件，并在事件达到执行时间时执行时间事件的处理函数。时间事件可以通过调用aeCreateTimeEvent函数来创建。

`aeTimeEvent`通常用于定时任务，例如周期性的定时器、延迟任务等。在事件处理框架启动时，它会将`aeTimeEvent`加入到一个最小堆中，以便事件处理框架可以及时地发现并处理这些事件。当时间达到`aeTimeEvent`中指定的执行时间时，事件处理框架会调用`timeProc`函数，处理该事件。在处理完事件后，如果指定了`finalizerProc`函数，事件处理框架还会调用该函数。最后，事件处理框架将从最小堆中删除该事件。

```
typedef void aeFileProc(struct aeEventLoop *eventLoop, int fd,
void *clientData, int mask);
typedef void aeEventFinalizerProc(struct aeEventLoop
*eventLoop, void *clientData);
```

4 aeFiredEvent

```
/* A fired event */
typedef struct aeFiredEvent {
    int fd;
    int mask;
} aeFiredEvent;

```

`aeFiredEvent` 是一个结构体，表示触发了的事件。它包含两个整型成员变量 `fd` 和 `mask`，分别表示触发事件的文件描述符和对应的事件类型掩码。在调用 `aeApiPoll` 函数时，内部会填充触发的事件到一个数组中，每个元素就是一个 `aeFiredEvent` 结构体。应用程序通过遍历这个数组，可以得到触发了哪些事件。

5 ae.c宏开关

```
#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
```

这段代码是Redis中的事件处理模块的实现方式。Redis使用了一种跨平台的事件处理模块，通过封装底层的事件机制（如epoll、select、kqueue等），来实现跨平台的事件处理。其中，`ae_evport.c`是基于Solaris的event port实现，`ae_epoll.c`是基于Linux的epoll实现，`ae_kqueue.c`是基于BSD的kqueue实现，`ae_select.c`是基于select实现。这段代码会根据编译时的配置，选择合适的事件处理实现，保证Redis可以在不同的平台上运行。

Redis会在编译时根据系统支持的I/O多路复用机制，选择使用哪一个实现。在Redis源代码中，会先根据编译选项判断系统支持的I/O多路复用机制，然后根据不同的机制分别包含对应的文件，比如`ae_select.c`、`ae_epoll.c`、`ae_kqueue.c`等。在这些文件中，会定义`aeApiCreate`等函数来实现对应的I/O多路复用机制，而Redis的核心事件处理器`ae.c`中，则只会调用`aeApiCreate`等函数，不需要关心具体是哪个机制的实现。因此，Redis可以根据不同的编译选项，编译出适配不同系统的二进制文件

6 aeCreateEventLoop

`aeCreateEventLoop`是创建一个事件循环的函数，它会分配一个`aeEventLoop`结构体，并初始化这个结构体中的成员变量。函数定义如下：

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

    monotonicInit();    /* just in case the calling app didn't initialize */
    /*monotonicInit()函数的作用是初始化一个用于记录时间的结构体，以便后续可以通过该结构体获取当前的时间戳。这里的注释是为了防止调用该函数的应用程序未初始化该结构体而导致错误*/
    if ((eventLoop = zmalloc(sizeof(*eventLoop))) == NULL) goto err;
    eventLoop->events = zmalloc(sizeof(aeFileEvent)*setsize);
    eventLoop->fired = zmalloc(sizeof(aeFiredEvent)*setsize);
    if (eventLoop->events == NULL || eventLoop->fired == NULL) goto err;
    eventLoop->setsize = setsize;
    eventLoop->timeEventHead = NULL;
    eventLoop->timeEventNextId = 0;
    eventLoop->stop = 0;
    eventLoop->maxfd = -1;
    eventLoop->beforesleep = NULL;
    eventLoop->aftersleep = NULL;
    eventLoop->flags = 0;
    if (aeApiCreate(eventLoop) == -1) goto err;
    /* Events with mask == AE_NONE are not set. So let's initialize the
     * vector with it. */
    for (i = 0; i < setsize; i++)
        eventLoop->events[i].mask = AE_NONE;
    return eventLoop;

```

```
err:
    if (eventLoop) {
        zfree(eventLoop->events);
        zfree(eventLoop->fired);
        zfree(eventLoop);
    }
    return NULL;
}

```

这个函数会传入一个`setsize`参数，表示这个事件循环中最大同时监听的文件描述符数。函数会首先分配一个`aeEventLoop`结构体，同时初始化其中的成员变量，比如时间事件列表的头指针`timeEventHead`和下一个时间事件的ID`timeEventNextId`。然后，调用`aeApiCreate`函数创建事件接口的状态，创建成功后，初始化事件数组中所有的文件事件掩码为`AE_NONE`，最后返回这个初始化好的事件循环结构体。

在`aeCreateEventLoop`函数中，会使用`zmalloc`函数申请内存，这个函数实际上就是Redis自己封装的`malloc`函数，用于内存申请，可以参考`zmalloc.c`文件中的实现。此外，在创建完成事件循环结构体后，如果创建接口状态失败，则会释放先前分配的内存，并返回`NULL`，表示创建失败。

7 aeResizeSetSize

```
int aeResizeSetSize(aeEventLoop *eventLoop, int setsize) {
    int i;

    if (setsize == eventLoop->setsize) return AE_OK;
    if (eventLoop->maxfd >= setsize) return AE_ERR;
    if (aeApiResize(eventLoop,setsize) == -1) return AE_ERR;

    eventLoop->events = zrealloc(eventLoop->events,sizeof(aeFileEvent)*setsize);
    eventLoop->fired = zrealloc(eventLoop->fired,sizeof(aeFiredEvent)*setsize);
    eventLoop->setsize = setsize;

    /* Make sure that if we created new slots, they are initialized with
     * an AE_NONE mask. */
    for (i = eventLoop->maxfd+1; i < setsize; i++)
        eventLoop->events[i].mask = AE_NONE;
    return AE_OK;
}
```

`aeResizeSetSize` 是一个 Redis 事件循环中用来调整底层实现的接口集合大小的函数。在 Redis 事件循环中，事件的底层实现接口集合通过 `aeApiCreate` 函数创建。当 Redis 事件循环中需要调整接口集合大小时，就会调用 `aeResizeSetSize` 函数。

具体来说，`aeResizeSetSize` 函数的作用是将事件接口集合的大小调整为 `setsize`。如果需要增大接口集合的大小，`aeResizeSetSize` 会分配一块新的内存来存储更多的事件接口。如果需要缩小接口集合的大小，`aeResizeSetSize` 会释放多余的事件接口占用的内存。

在 Redis 中，事件接口的底层实现由编译时配置的不同选项决定。具体而言，如果编译时选择了 `HAVE_EPOLL`，则 Redis 会使用 epoll 来实现事件接口；如果选择了 `HAVE_KQUEUE`，则 Redis 会使用 kqueue；如果两者都没选，Redis 会使用 select 来实现事件接口。而 `aeResizeSetSize` 函数中所调用的事件接口函数，实际上是由对应的底层实现提供的。例如，在 epoll 实现中，调用 `aeResizeSetSize` 函数时实际上是调用了 `aeApiResize` 函数。

8 aeDeleteEventLoop

```
void aeDeleteEventLoop(aeEventLoop *eventLoop) {
    aeApiFree(eventLoop);
    zfree(eventLoop->events);
    zfree(eventLoop->fired);

    /* Free the time events list. */
    aeTimeEvent *next_te, *te = eventLoop->timeEventHead;
    while (te) {
        next_te = te->next;
        zfree(te);
        te = next_te;
    }
    zfree(eventLoop);
}

```

该函数是用于删除整个事件循环和相应的内存资源的。它首先调用`aeApiFree`函数来释放底层事件处理机制（例如 epoll、select 等）所使用的数据结构的内存，然后释放事件集合和已触发事件集合的内存，最后释放时间事件链表中所有的时间事件节点的内存，最后再释放事件循环本身所使用的内存。

9 aeCreateFileEvent

`aeCreateFileEvent` 函数用于向 `eventLoop` 注册文件事件，当文件描述符变得可读、可写或发生错误时，将触发对应的事件处理函数。

函数原型如下：

```
int aeCreateFileEvent(aeEventLoop *eventLoop, int fd, int
mask,
        aeFileProc *proc, void *clientData)
{
    if (fd >= eventLoop->setsize) {
        errno = ERANGE;
        return AE_ERR;
    }
    aeFileEvent *fe = &eventLoop->events[fd];

    if (aeApiAddEvent(eventLoop, fd, mask) == -1)
        return AE_ERR;
    fe->mask |= mask;
    if (mask & AE_READABLE) fe->rfileProc = proc;
    if (mask & AE_WRITABLE) fe->wfileProc = proc;
    fe->clientData = clientData;
    if (fd > eventLoop->maxfd)
        eventLoop->maxfd = fd;
    return AE_OK;
}
```

参数解释：

- `eventLoop`：事件循环结构体指针。
- `fd`：要监听的文件描述符。
- `mask`：要监听的事件类型，可以是 `AE_READABLE`、`AE_WRITABLE` 或它们的按位或。
- `proc`：事件处理函数指针，当文件描述符变得可读、可写或发生错误时，该函数将被调用。
- `clientData`：客户端数据指针，会作为参数传递给事件处理函数。

函数返回值为 `AE_OK` 表示成功，返回 `AE_ERR` 表示失败。

10 aeDeleteFileEvent

```
void aeDeleteFileEvent(aeEventLoop *eventLoop, int fd, int
mask)
{
    if (fd >= eventLoop->setsize) return;
    aeFileEvent *fe = &eventLoop->events[fd];
    if (fe->mask == AE_NONE) return;

    /* We want to always remove AE_BARRIER if set when
AE_WRITABLE
     * is removed. */
    if (mask & AE_WRITABLE) mask |= AE_BARRIER;

    aeApiDelEvent(eventLoop, fd, mask);
    fe->mask = fe->mask & (~mask);
    if (fd == eventLoop->maxfd && fe->mask == AE_NONE) {
        /* Update the max fd */
        int j;

        for (j = eventLoop->maxfd-1; j >= 0; j--)
            if (eventLoop->events[j].mask != AE_NONE) break;
        eventLoop->maxfd = j;
    }
}
```

参数说明：

- `eventLoop`：事件循环。
- `fd`：文件描述符。
- `mask`：需要删除的事件掩码。可以是 AE_READABLE 或者 AE_WRITABLE。

函数实现：

- 首先根据文件描述符 `fd` 获取文件事件结构体 `fe`。
- 如果需要删除的事件掩码 `mask` 是 AE_READABLE，就从监听可读事件的文件事件结构体列表中删除 `fe`，否则从监听可写事件的文件事件结构体列表中删除 `fe`。
- 如果文件事件结构体 `fe` 的监听事件列表 `mask` 变成了 AE_NONE，说明该文件描述符已经没有任何事件要监听，就删除该文件描述符的所有监听事件，即从事件驱动库中删除该文件描述符的所有监听事件。

`aeDeleteFileEvent`函数则用于将给定的文件描述符（`fd`）及其对应的事件掩码（`mask`），从事件循环（`eventLoop`）中删除。在内部实现中，`aeDeleteFileEvent`函数会首先判断该文件描述符是否已经在事件循环中被监听过，如果存在，则会将其从事件循环中删除。如果文件描述符对应的事件处理器已经存在，该函数还会将其处理器函数和客户端数据清除。

11 aeGetFileClientData

```
void *aeGetFileClientData(aeEventLoop *eventLoop, int fd) {
    if (fd >= eventLoop->setsize) return NULL;
    aeFileEvent *fe = &eventLoop->events[fd];
    if (fe->mask == AE_NONE) return NULL;

    return fe->clientData;
}
```

aeGetFileClientData函数用于获取与文件事件关联的客户端数据。该函数使用aeEventLoop指针、文件描述符和事件掩码作为参数，并返回与事件关联的客户端数据。

客户端数据是在使用aeCreateFileEvent函数创建文件事件时设置的。它是指向特定于事件的数据的指针，并由事件处理函数用于执行其操作。客户端数据可以是用户想要关联事件的任何内容。

当事件处理函数需要访问与事件关联的客户端数据时，aeGetFileClientData函数非常有用。例如，如果事件处理函数是在套接字上有数据可用时调用的回调函数，则客户端数据可以是指向包含有关套接字或正在读取或写入的数据的信息的结构体的指针。

总之，aeGetFileClientData检索与文件事件关联的客户端数据，并将其返回给调用者。

12 aeCreateTimeEvent

```
long long aeCreateTimeEvent(aeEventLoop *eventLoop, long long
milliseconds,
        aeTimeProc *proc, void *clientData,
        aeEventFinalizerProc *finalizerProc)
{
    long long id = eventLoop->timeEventNextId++;
    aeTimeEvent *te;

    te = zmalloc(sizeof(*te));
    if (te == NULL) return AE_ERR;
```

```
    te->id = id;
    te->when = getMonotonicUs() + milliseconds * 1000;
    te->timeProc = proc;
    te->finalizerProc = finalizerProc;
    te->clientData = clientData;
    te->prev = NULL;
    te->next = eventLoop->timeEventHead;
    te->refcount = 0;
    if (te->next)
        te->next->prev = te;
    eventLoop->timeEventHead = te;
    return id;
}
```

aeCreateTimeEvent函数用于在事件循环中创建一个时间事件。该函数接受一个aeEventLoop指针、一个毫秒级别的时间戳、一个回调函数指针、一个回调函数的参数指针和一个可选的释放函数指针作为参数。

时间事件是在指定的时间戳之后执行的函数，类似于定时器。回调函数指针是要在事件到达时调用的函数。回调函数的参数指针是一个指向传递给回调函数的数据的指针。释放函数指针是可选的，如果提供，将在时间事件被删除时调用以释放任何为事件分配的资源。

时间事件的id是自动生成的，并且在事件循环中必须是唯一的。时间事件可以添加到事件循环中，也可以从事件循环中删除。

总之，aeCreateTimeEvent函数用于在事件循环中创建一个时间事件，使用户可以在指定的时间戳之后执行一个函数，并在事件循环中管理它。

13 aeDeleteTimeEvent

```
int aeDeleteTimeEvent(aeEventLoop *eventLoop, long long id)
{
    aeTimeEvent *te = eventLoop->timeEventHead;
    while(te) {
        if (te->id == id) {
            te->id = AE_DELETED_EVENT_ID;
            return AE_OK;
        }
        te = te->next;
    }
    return AE_ERR; /* NO event with the specified ID found */
}
```

aeDeleteTimeEvent函数用于从事件循环中删除指定的时间事件。它接收一个指向事件循环的指针和一个指向要删除的时间事件的指针作为参数。它首先从事件循环中移除该时间事件，然后释放该时间事件所占用的内存。

在Redis中，时间事件是通过调用aeCreateTimeEvent函数创建的。时间事件的创建和删除通常在一起使用，以确保时间事件不再被调度并且不占用内存。当不再需要时间事件时，应使用aeDeleteTimeEvent函数将其从事件循环中删除。

需要注意的是，在递归调用时间事件处理函数时，必须对时间事件进行引用计数。这是因为，在时间事件处理函数中删除时间事件会导致内存泄漏或未定义的行为。因此，aeCreateTimeEvent会在创建时间事件时增加时间事件的引用计数，并在删除时间事件时减少引用计数。只有当时间事件的引用计数为零时，才会真正地将其从事件循环中删除。

总之，aeDeleteTimeEvent函数用于从事件循环中删除时间事件并释放其占用的内存，通常与aeCreateTimeEvent函数一起使用。它还确保递归调用时间事件处理函数时不会发生内存泄漏或未定义的行为。

14 usUntilEarliestTimer

```
static int64_t usUntilEarliestTimer(aeEventLoop *eventLoop) {
    aeTimeEvent *te = eventLoop->timeEventHead;
    if (te == NULL) return -1;

    aeTimeEvent *earliest = NULL;
    while (te) {
        if (!earliest || te->when < earliest->when)
            earliest = te;
        te = te->next;
    }

    monotime now = getMonotonicUs();
    return (now >= earliest->when) ? 0 : earliest->when - now;
}
```

usUntilEarliestTime函数的作用是计算事件循环中最早的时间事件所需的等待时间（以微秒为单位）。这个函数会遍历所有的时间事件，找到最早的那个事件的时间戳（when字段），并计算当前时间到该时间戳之间的时间差，作为返回值返回。

如果事件循环中没有任何时间事件，则该函数返回默认的等待时间，即给定的默认等待时间或100毫秒，以较小的那个为准。如果时间事件已经过期，那么函数会返回0，表示无需等待，直接处理过期事件。

15 processTimeEvents

```
static int processTimeEvents(aeEventLoop *eventLoop) {
    int processed = 0;
    aeTimeEvent *te;
    long long maxId;

    te = eventLoop->timeEventHead;
    maxId = eventLoop->timeEventNextId-1;
    monotime now = getMonotonicUs();
    while(te) {
        long long id;

        /* Remove events scheduled for deletion. */
        if (te->id == AE_DELETED_EVENT_ID) {
            aeTimeEvent *next = te->next;
            /* If a reference exists for this timer event,
             * don't free it. This is currently incremented
             * for recursive timerProc calls */
            if (te->refcount) {
                te = next;
                continue;
            }
            if (te->prev)
                te->prev->next = te->next;
            else
                eventLoop->timeEventHead = te->next;
            if (te->next)
                te->next->prev = te->prev;
            if (te->finalizerProc) {
                te->finalizerProc(eventLoop, te->clientData);
                now = getMonotonicUs();
            }
            zfree(te);
            te = next;
            continue;
        }

        /* Make sure we don't process time events created by
time events in
         * this iteration. Note that this check is currently
useless: we always
         * add new timers on the head, however if we change
the implementation
```

```
40          * detail, this check may be useful again: we keep it
here for future
41          * defense. */
42         if (te->id > maxId) {
43             te = te->next;
44             continue;
45         }
46 
47         if (te->when <= now) {
48             int retval;
49 
50             id = te->id;
51             te->refcount++;
52             retval = te->timeProc(eventLoop, id, te-
>clientData);
53             te->refcount--;
54             processed++;
55             now = getMonotonicUs();
56             if (retval != AE_NOMORE) {
57                 te->when = now + retval * 1000;
58             } else {
59                 te->id = AE_DELETED_EVENT_ID;
60             }
61         }
62         te = te->next;
63     }
64     return processed;
65 }
```

`processTimeEvent` 是一个处理时间事件的函数，它在事件循环中被调用，用于检查当前是否有到期的时间事件，并调用它们的处理函数。

具体来说，`processTimeEvent` 函数首先获取当前的 Unix 时间戳，然后遍历整个时间事件链表，查找到期的时间事件，并将其放入到一个数组中，同时更新事件循环中最近的一个时间事件。

接着，`processTimeEvent` 函数遍历刚刚放入数组中的时间事件，依次调用它们的处理函数，并将处理函数的返回值存储到另一个数组中。如果时间事件需要重复执行，`processTimeEvent` 函数会将它们再次添加到事件循环中。

最后，`processTimeEvent` 函数根据处理函数返回的值，更新事件循环中最近的一个时间事件，并返回处理的时间事件的数量。如果没有到期的时间事件，返回 0。

总之，`processTimeEvent`函数用于处理时间事件，并根据处理函数的返回值更新事件循环中最近的一个时间事件。

16 aeProcessEvents

```
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 to 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;
        struct timeval tv, *tvp;
        int64_t usUntilTimer = -1;

        if (flags & AE_TIME_EVENTS && !(flags & AE_DONT_WAIT))
            usUntilTimer = usUntilEarliestTimer(eventLoop);

        if (usUntilTimer >= 0) {
            tv.tv_sec = usUntilTimer / 1000000;
            tv.tv_usec = usUntilTimer % 1000000;
            tvp = &tv;
        } 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 {
```

```
33                  /* Otherwise we can block */
34                  tvp = NULL; /* wait forever */
35              }
36          }
37
38          if (eventLoop->flags & AE_DONT_WAIT) {
39              tv.tv_sec = tv.tv_usec = 0;
40              tvp = &tv;
41          }
42
43          if (eventLoop->beforesleep != NULL && flags &
AE_CALL_BEFORE_SLEEP)
44              eventLoop->beforesleep(eventLoop);
45
46          /* Call the multiplexing API, will return only on
timeout or when
47           * some event fires. */
48          numevents = aeApiPoll(eventLoop, tvp);
49
50          /* After sleep callback. */
51          if (eventLoop->aftersleep != NULL && flags &
AE_CALL_AFTER_SLEEP)
52              eventLoop->aftersleep(eventLoop);
53
54          for (j = 0; j < numevents; j++) {
55              int fd = eventLoop->fired[j].fd;
56              aeFileEvent *fe = &eventLoop->events[fd];
57              int mask = eventLoop->fired[j].mask;
58              int fired = 0; /* Number of events fired for
current fd. */
59
60              /* Normally we execute the readable event first,
and the writable
61               * event later. This is useful as sometimes we
may be able
62               * to serve the reply of a query immediately
after processing the
63               * query.
64               *
65               * However if AE_BARRIER is set in the mask, our
application is
66               * asking us to do the reverse: never fire the
writable event
67               * after the readable. In such a case, we invert
the calls.
```

```
            * This is useful when, for instance, we want to
do things
             * in the beforeSleep() hook, like fsyncing a
file to disk,
             * before replying to a client. */
            int invert = fe->mask & AE_BARRIER;

            /* 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.
             *
             * Fire the readable event if the call sequence
is not
             * inverted. */
            if (!invert && fe->mask & mask & AE_READABLE) {
                fe->rfileProc(eventLoop,fd,fe-
>clientData,mask);
                fired++;
                fe = &eventLoop->events[fd]; /* Refresh in
case of resize. */
            }

            /* Fire the writable event. */
            if (fe->mask & mask & AE_WRITABLE) {
                if (!fired || fe->wfileProc != fe->rfileProc)
{
                    fe->wfileProc(eventLoop,fd,fe-
>clientData,mask);
                    fired++;
                }
            }

            /* If we have to invert the call, fire the
readable event now
             * after the writable one. */
            if (invert) {
                fe = &eventLoop->events[fd]; /* Refresh in
case of resize. */
                if ((fe->mask & mask & AE_READABLE) &&
                    (!fired || fe->wfileProc != fe-
>rfileProc))
                {
```

```
100                     fe->rfileProc(eventLoop,fd,fe->clientData,mask);
101                     fired++;
102                 }
103             }
104 
105             processed++;
106         }
107     }
108     /* Check time events */
109     if (flags & AE_TIME_EVENTS)
110         processed += processTimeEvents(eventLoop);
111 
112     return processed; /* return the number of processed file/time events */
113 }
114 
```

aeProcessEvents函数是事件循环的核心函数，用于处理事件的等待、分发和执行。它首先会根据timeout参数计算出最长等待时间，然后调用底层实现的等待函数（如select、epoll_wait等）等待事件的发生。当有事件发生时，它会根据事件类型（文件事件或时间事件）调用相应的处理函数。文件事件处理函数会检查文件事件的类型，并根据事件类型调用相应的处理函数（读、写、错误等）。时间事件处理函数会遍历时间事件链表，并根据事件到期时间调用相应的处理函数。处理完所有的事件后，它会重新计算最近一个时间事件到期的时间，并返回等待的时间。

总之，aeProcessEvents函数是事件循环的核心，负责等待、分发和执行事件。它调用底层的等待函数等待事件的发生，然后根据事件类型调用相应的处理函数。处理完所有的事件后，它会重新计算下一次等待的时间，并返回等待的时间。

17 aeWait

```
1  int aeWait(int fd, int mask, long long milliseconds) {
2      struct pollfd pfd;
3      int retmask = 0, retval;
4  
5      memset(&pfd, 0, sizeof(pfd));
6      pfd.fd = fd;
7      if (mask & AE_READABLE) pfd.events |= POLLIN;
8      if (mask & AE_WRITABLE) pfd.events |= POLLOUT;
9  
10     if ((retval = poll(&pfd, 1, milliseconds))== 1) {
11         if (pfd.revents & POLLIN) retmask |= AE_READABLE;
12         if (pfd.revents & POLLOUT) retmask |= AE_WRITABLE;
```

```
        if (pfd.revents & POLLERR) retmask |= AE_WRITABLE;
        if (pfd.revents & POLLHUP) retmask |= AE_WRITABLE;
        return retmask;
    } else {
        return retval;
    }
}

void aeMain(aeEventLoop *eventLoop) {
    eventLoop->stop = 0;
    while (!eventLoop->stop) {
        aeProcessEvents(eventLoop, AE_ALL_EVENTS|
                                   AE_CALL_BEFORE_SLEEP|
                                   AE_CALL_AFTER_SLEEP);
    }
}

```

函数aeWait用于等待事件的发生，它会阻塞当前进程直到有事件被触发或达到指定的超时时间。当事件被触发时，aeWait会返回已触发事件的数量。

aeWait的参数是一个指向事件循环结构体的指针，以及等待的超时时间（以毫秒为单位）。如果超时时间为NULL，则aeWait会一直阻塞直到事件被触发。

aeWait会使用事件处理机制提供的API来等待事件的发生。它会根据操作系统支持的机制使用epoll、select、kqueue等机制进行等待。

在等待事件的过程中，aeWait会根据事件循环结构体中设置的时间事件列表的最小超时时间来调整等待的超时时间。当有时间事件的超时时间到达时，aeWait会调用相应的处理函数来处理时间事件。

总之，aeWait是事件循环的核心函数，它通过调用底层的API等待事件的发生，同时处理时间事件。