redis-epoll源码分析

1 aeApiState结构体

```
typedef struct aeApiState {
    int epfd;
    struct epoll_event *events;
} aeApiState;
```

The letters `ae` in `aeEventLoop` refer to "Asynchronous Events"

> In Redis, aeApiState is a structure that represents the platform-specific state of the event loop abstraction layer used by Redis. The aeApiState structure is defined in the "src/ae.h" header file and is used by Redis to provide a layer of abstraction between the event loop implementation and the platform-specific I/O multiplexing mechanism.

1. `epfd`: An integer representing the file descriptor of the epoll instance used by the event loop for monitoring file descriptor events. On platforms other than Linux, this field may refer to a different I/O multiplexing mechanism.
2. `events`: A pointer to an array of epoll events used by the event loop to store active file descriptor events.

2 aeEventLoop

```
typedef struct aeEventLoop {
    int maxfd;                // 监听描述符的最大值
    int setsize;              // 可监听的最大描述符数
    long long timeEventNextId; // 下一个时间事件ID
    time_t lastTime;          // 上一次执行时间事件的时间
    aeFileEvent *events;      // 监听事件的数组
    aeFiredEvent *fired;      // 已触发的事件数组
    aeTimeEvent *timeEventHead; // 时间事件链表
    int stop;                 // 是否停止事件循环
    void *apidata;            // 抽象IO复用接口数据结构
    aeBeforeSleepProc *beforesleep; // 事件循环前回调函数
    aeBeforeSleepProc *aftersleep;  // 事件循环后回调函数
    int flag
} aeEventLoop;

```

1. `events`：指向 `aeFileEvent` 结构的数组的指针，表示已在事件循环中注册的文件事件。每个 `aeFileEvent` 结构包含一个文件描述符、已注册的事件掩码（可读、可写或两者兼有）以及在事件发生时执行的回调函数。
2. `fired`：指向 `aeFiredEvent` 结构的数组的指针，表示已发生并准备处理的文件事件。每个 `aeFiredEvent` 结构包含文件描述符、已发生的事件掩码以及对应的 `aeFileEvent` 结构的引用。
3. `timeEventHead`：指向 `aeTimeEvent` 结构链表头的指针，表示已在事件循环中注册的时间事件。每个 `aeTimeEvent` 结构包含一个在时间事件发生时执行的回调函数和一个以毫秒为单位的时间间隔。
4. `stop`：标志，指示事件循环是否应停止运行。当创建事件循环时，此标志设置为 0，可以将其设置为 1 以停止事件循环的运行。
5. `maxfd`：表示在事件循环中注册的最大文件描述符值的整数。此值用于在轮询文件事件时优化性能。
6. `beforeSleep`：指向回调函数的指针，该回调函数在事件循环进入休眠等待事件发生之前执行。可以使用此函数执行任何必要的清理或准备任务。
7. `apiData`：指向事件循环实现所需的平台特定数据的指针。可以使用此指针存储事件循环机制所需的任何其他信息。

Redis 使用这些字段来管理事件循环的状态和已注册的文件和时间事件。当事件发生时，事件循环会使用相应的 `aeFiredEvent` 结构更新 `fired` 数组，然后 Redis 可以执行适当的回调函数。`timeEventHead` 链表用于跟踪时间事件，并在适当的时间间隔执行相应的回调函数。

3 aeApiCreate

```
static int aeApiCreate(aeEventLoop *eventLoop) {
    aeApiState *state = zmalloc(sizeof(aeApiState));

    if (!state) return -1;
    state->events = zmalloc(sizeof(struct epoll_event)*eventLoop->setsize);
    if (!state->events) {
        zfree(state);
        return -1;
    }
    state->epfd = epoll_create(1024); /* 1024 is just a hint for the kernel */
    if (state->epfd == -1) {
        zfree(state->events);
        zfree(state);
        return -1;
```

```
    }
    anetCloexec(state->epfd);
    eventLoop->apidata = state;
    return 0;
}

```

> `aeApiCreate` 函数的作用是创建一个与操作系统的 I/O 多路复用机制相关联的数据结构，以便于 Redis 事件循环可以使用该机制进行事件的监视和处理。
>
> 在 Redis 中，`aeApiCreate` 函数的实现会使用操作系统提供的 `epoll_create` 系统调用创建一个 epoll 实例。该 epoll 实例可以用于监视 Redis 事件循环中所注册的文件描述符的状态，并在这些文件描述符变为可读或可写时通知 Redis 事件循环，以便于 Redis 事件循环可以及时处理这些事件。
>
> `aeApiCreate` 函数还会为 epoll 实例分配一些内存，用于存储操作系统通知 Redis 事件循环的事件信息。这些信息会被存储在 `aeApiState` 结构体的 `events` 成员中，由 Redis 事件循环使用。
>
> 最后，`aeApiCreate` 函数会将创建好的 `aeApiState` 结构体保存到 Redis 事件循环的 `apidata` 成员中，以便于 Redis 事件循环在后续的事件处理中可以使用该结构体。

4 anetCloexex

`anetCloexec` 函数是一个网络编程辅助函数，用于设置文件描述符的 close-on-exec 标志，以防止在调用 `exec` 系列函数时继承该文件描述符。该函数的定义在 Redis 源代码中的 `anet.c` 文件中，其代码如下：

```
/* Enable the FD_CLOEXEC on the given fd to avoid fd leaks.
 * This function should be invoked for fd's on specific places
 * where fork + execve system calls are called. */
int anetCloexec(int fd) {
    int r;
    int flags;

    do {
        r = fcntl(fd, F_GETFD);
    } while (r == -1 && errno == EINTR);

    if (r == -1 || (r & FD_CLOEXEC))
        return r;

    flags = r | FD_CLOEXEC;
```

```
16
17      do {
18          r = fcntl(fd, F_SETFD, flags);
19      } while (r == -1 && errno == EINTR);
20
21      return r;
22  }
23
```

`anetCloexec` 函数是一个网络编程辅助函数，用于设置文件描述符的 close-on-exec 标志，以防止在调用 `exec` 系列函数时继承该文件描述符。该函数的定义在 Redis 源代码中的 `anet.c` 文件中，其代码如下：

```
1  cCopy codestatic int anetCloexec(int fd) {
2      int flags;
3
4      if ((flags = fcntl(fd, F_GETFD)) == -1) {
5          return -1;
6      }
7
8      if (fcntl(fd, F_SETFD, flags | FD_CLOEXEC) == -1) {
9          return -1;
10     }
11
12     return 0;
13 }
```

`anetCloexec` 函数首先使用 `fcntl` 函数获取文件描述符 `fd` 的文件描述符标志，将其保存在变量 `flags` 中。然后，该函数将 `FD_CLOEXEC` 标志设置到 `flags` 变量中，并使用 `fcntl` 函数将更新后的标志写回到文件描述符 `fd` 中。如果任何一个 `fcntl` 调用失败，`anetCloexec` 函数会返回 -1。

`FD_CLOEXEC` 是一个文件描述符标志，用于指示在调用 `exec` 系列函数时自动关闭该文件描述符。如果一个进程在调用 `exec` 系列函数之前没有显示地关闭某个文件描述符，那么这个文件描述符会被复制到新进程中。如果在新进程中不需要该文件描述符，那么就需要手动关闭该文件描述符，否则会出现文件描述符泄漏的问题。通过设置 `FD_CLOEXEC` 标志，可以使得在调用 `exec` 系列函数时自动关闭该文件描述符，从而避免文件描述符泄漏的问题。

5 aeApiResize

```
static int aeApiResize(aeEventLoop *eventLoop, int setsize) {
    aeApiState *state = eventLoop->apidata;

    state->events = zrealloc(state->events, sizeof(struct epoll_event)*setsize);
    return 0;
}
```

`aeApiResize` 函数的作用是重新调整文件事件处理器的内部数据结构，以容纳更多的文件描述符。具体来说，它会重新分配 `state->events` 数组的内存空间，以适应新的 `setsize` 大小。其中，`state->events` 数组保存由 epoll 文件事件处理器返回的事件集合。当 Redis 调用 `aeCreateFileEvent` 函数时，它会向 epoll 文件事件处理器注册指定文件描述符上的事件，并将该事件保存到 `state->events` 数组中。在事件处理器处理完事件后，Redis 可以通过 `state->events` 数组获取发生事件的文件描述符和事件类型。

需要注意的是，Redis 7.2 中的 `aeApiResize` 函数只会重新分配 `state->events` 数组的内存空间，而不会重新分配 `eventLoop->fired` 数组的内存空间。这与 Redis 6.0 中的实现不同。

6 aeApiFree

`aeApiFree` 函数是 Redis 文件事件处理器底层实现相关的接口函数之一，用于释放 epoll 文件事件处理器相关的状态。具体来说，它会释放 `aeApiState` 结构体中保存的 epoll 文件描述符集合以及事件集合，并将 `aeApiState` 结构体本身所占用的内存空间释放。

```
static void aeApiFree(aeEventLoop *eventLoop) {
    aeApiState *state = eventLoop->apidata;

    close(state->epfd);
    zfree(state->events);
    zfree(state);
}
```

该函数首先从事件循环结构体 `eventLoop` 中获取 `aeApiState` 结构体指针 `state`，然后依次释放 `state->events` 数组的内存空间、关闭 epoll 文件描述符集合 `state->epfd`，最后释放 `aeApiState` 结构体指针 `state` 所占用的内存空间。

需要注意的是，`aeApiFree` 函数只是负责释放 epoll 文件事件处理器相关的状态，并不负责释放其他文件事件处理器相关的状态。如果 Redis 使用其他类型的文件事件处理器（如 kqueue、evport 等），则需要编写相应的 `aeApiFree` 函数来释放相关的状态。

7 aeApiAddEvent

`aeApiAddEvent` 函数是 Redis 文件事件处理器底层实现相关的接口函数之一，用于向 epoll 文件事件处理器中添加一个文件事件。具体来说，它会将指定的文件描述符和事件类型添加到 epoll 文件描述符集合中，并将对应的事件结构体添加到事件集合中。

```
#define AE_NONE 0       /* No events registered. */
#define AE_READABLE 1   /* Fire when descriptor is readable. */
#define AE_WRITABLE 2   /* Fire when descriptor is writable. */
typedef union epoll_data
{
  void *ptr;
  int fd;
  uint32_t u32;
  uint64_t u64;
} epoll_data_t;

struct epoll_event
{
  uint32_t events;  /* Epoll events */
  epoll_data_t data;    /* User data variable */
} __EPOLL_PACKED;

```

```
static int aeApiAddEvent(aeEventLoop *eventLoop, int fd, int mask) {
    aeApiState *state = eventLoop->apidata;
    struct epoll_event ee = {0}; /* avoid valgrind warning */
    /* If the fd was already monitored for some event, we need a MOD
     * operation. Otherwise we need an ADD operation. */
    int op = eventLoop->events[fd].mask == AE_NONE ?
            EPOLL_CTL_ADD : EPOLL_CTL_MOD;

    ee.events = 0;
    mask |= eventLoop->events[fd].mask; /* Merge old events */
```

```
    if (mask & AE_READABLE) ee.events |= EPOLLIN;//读
    if (mask & AE_WRITABLE) ee.events |= EPOLLOUT;//写
    ee.data.fd = fd;
    if (epoll_ctl(state->epfd,op,fd,&ee) == -1) return -1;
    return 0;
}
```

该函数首先从事件循环结构体 `eventLoop` 中获取 `aeApiState` 结构体指针 `state`，然后根据指定的文件描述符 `fd` 和事件类型 `mask` 初始化一个 `struct epoll_event` 结构体 `ee`。接着，它会判断当前文件描述符是否已经被添加到 epoll 文件描述符集合中。如果是，则采用 `EPOLL_CTL_MOD` 操作进行修改，否则采用 `EPOLL_CTL_ADD` 操作进行添加。然后，它会将指定的事件类型 `mask` 合并到已有的事件类型中，并根据读写事件类型设置 `ee.events` 的值。最后，它会将 `ee` 作为参数调用 `epoll_ctl` 函数将指定的文件描述符和事件类型添加到 epoll 文件描述符集合中，并将对应的事件结构体添加到事件集合中。

需要注意的是，`aeApiAddEvent` 函数只是负责向 epoll 文件事件处理器中添加一个文件事件，而不是修改文件事件。如果需要修改文件事件，应该使用 `aeApiModEvent` 函数。

8 aeApiDelEvent

`aeApiDelEvent`函数是用于从epoll实例中删除事件的函数，其函数定义如下：

```
static void aeApiDelEvent(aeEventLoop *eventLoop, int fd, int
delmask) {
    aeApiState *state = eventLoop->apidata;
    struct epoll_event ee = {0}; /* avoid valgrind warning */
    int mask = eventLoop->events[fd].mask & (~delmask);

    ee.events = 0;
    if (mask & AE_READABLE) ee.events |= EPOLLIN;
    if (mask & AE_WRITABLE) ee.events |= EPOLLOUT;
    ee.data.fd = fd;
    if (mask != AE_NONE) {
        epoll_ctl(state->epfd,EPOLL_CTL_MOD,fd,&ee);
    } else {
        /* Note, Kernel < 2.6.9 requires a non null event
pointer even for
         * EPOLL_CTL_DEL. */
        epoll_ctl(state->epfd,EPOLL_CTL_DEL,fd,&ee);
    }
}

```

该函数首先从`eventLoop`中获取到`state`，然后根据传入的`fd`和`mask`确定需要删除的事件类型。接着，函数将根据被删除事件之后剩下的事件类型进行对应的`epoll_ctl`操作：

- 如果事件集合中还有其他事件，那么需要对事件进行修改（`EPOLL_CTL_MOD`）操作，否则需要将事件从epoll实例中删除（`EPOLL_CTL_DEL`）。

该函数的返回值为0表示操作成功，否则表示操作失败。

9 aeApiPoll

`aeApiPoll`是Redis事件驱动机制中与`epoll_wait`函数相对应的函数。它会阻塞等待事件的发生，并将就绪的事件放入`eventLoop`中。该函数的定义如下：

```
static int aeApiPoll(aeEventLoop *eventLoop, struct timeval *tvp) {
    aeApiState *state = eventLoop->apidata;
    int retval, numevents = 0;

    retval = epoll_wait(state->epfd,state->events,eventLoop->setsize,
            tvp ? (tvp->tv_sec*1000 + (tvp->tv_usec + 999)/1000) : -1);
    if (retval > 0) {
        int j;

        numevents = retval;
        for (j = 0; j < numevents; j++) {
            int mask = 0;
            struct epoll_event *e = state->events+j;

            if (e->events & EPOLLIN) mask |= AE_READABLE;
            if (e->events & EPOLLOUT) mask |= AE_WRITABLE;
            if (e->events & EPOLLERR) mask |= AE_WRITABLE|AE_READABLE;
            if (e->events & EPOLLHUP) mask |= AE_WRITABLE|AE_READABLE;
            eventLoop->fired[j].fd = e->data.fd;
            eventLoop->fired[j].mask = mask;
        }
    } else if (retval == -1 && errno != EINTR) {
        panic("aeApiPoll: epoll_wait, %s", strerror(errno));
    }

```

```
26      return numevents;
27  }
28
```

`aeApiPoll`函数中的主要逻辑是通过`epoll_wait`函数进行阻塞等待事件的发生，并将就绪的事件放入`state->events`中。该函数返回已经就绪的事件数量，事件的数量通过`numevents`变量保存。

接下来，`aeApiPoll`函数遍历`state->events`数组，并将每一个就绪的事件都添加到`eventLoop->fired`数组中。在添加的过程中，根据每个事件的类型，设置对应事件的掩码。

最后，`aeApiPoll`函数返回已经就绪的事件数量。这些就绪事件的掩码已经设置好，可以在之后的处理中使用。

10 *aeApiName

```
1 static char *aeApiName(void) {
2     return "epoll";
3 }
```

`aeApiName`是 Redis 源代码中的函数指针，指向当前 Redis 正在使用的 I/O 多路复用 API 的名称。它用于打印与 I/O 多路复用相关的调试信息和错误消息，并根据平台和可用 API 在编译时设置为适当的函数。例如，在支持 epoll 的 Linux 系统上，`aeApiName`指向一个返回字符串“epoll”的函数。在 macOS 系统上，它指向一个返回字符串“kqueue”的函数。

11 函数指针类型介绍

函数指针类型的定义是一种 C 语言中的语法结构，它通过`typedef`关键字将一个函数的类型定义成一个新的类型，使得我们可以使用这个新类型来定义变量、参数、返回值等。

一般来说，函数指针类型的定义形如：

```
1 typedef 返回值类型 (*函数指针类型名)(参数类型列表);
```

其中，`返回值类型`表示函数的返回值类型，`函数指针类型名`表示定义的新类型的名称，`参数类型列表`表示函数的参数类型列表，多个参数类型之间用逗号`,`隔开。在定义函数指针类型时，需要使用一对圆括号`()`将整个函数指针类型的定义括起来，以确保`*`操作符作用于函数指针类型名上，而不是作用于返回值类型上。

例如，以下代码定义了一个函数指针类型`func_ptr`，它指向一个参数为整数类型，返回值为浮点数类型的函数：

```
typedef float (*func_ptr)(int);
```

有了这个函数指针类型，我们就可以声明一个指向这种类型函数的指针变量：

```
func_ptr p;
```

或者将这个函数指针类型作为某个函数的参数类型或返回值类型：

```
float foo(func_ptr f) { ... }
func_ptr bar() { ... }
```

因为下一张我们需要用到这个我在这说明一下,例如：

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

`typedef void aeFileProc(struct aeEventLoop *eventLoop, int fd, void *clientData, int mask);` 是一个 C 语言中的函数指针类型定义。它定义了一个函数指针类型 `aeFileProc`，该函数指针可以指向一个参数为 `aeEventLoop *`、`int`、`void *` 和 `int` 类型的函数，并且该函数的返回类型为 `void`。在 Redis 中，这个函数指针类型主要用于表示事件处理器函数的类型，例如 `aeCreateFileEvent` 函数的第三个参数就是一个 `aeFileProc` 类型的函数指针。使用这种方式定义函数指针类型可以使代码更加简洁、清晰，并且易于维护和扩展。