Nginx 的文件异步 I/O

Nginx 中的文件异步 I/O 采用的不是 glibc 库提供的基于多线程实现的异步 I/O，而是由 Linux 内核实现的。Linux 内核提供了 5 个系统调用来完成文件操作的异步 I/O 功能，列举如下：

方法名	参数含义	执行意义
int io_setup(unsigned nr_events, aio_context_t *ctxp)	nr_events 表示需要可以处理的事件的最小个数，ctxp 是文件异步 I/O 的上下文描述符指针	初始化文件异步 I/O 的上下文，返回 0 表示成功。
int io_destory(aio_context_t ctx)	ctx 是文件异步 I/O 的上下文描述符	销毁文件异步 I/O 的上下文，返回 0 表示成功。
int io_submit(aio_context_t ctx, long nr, struct iocb *cbp[])	ctx 是文件异步 I/O 的上下文描述符，nr 是一次提交的事件个数，cbp 是提交的事件数组中的首个元素地址	提交文件异步 I/O 操作。返回值表示成功提交的事件个数
int io_cancel(aio_context_t ctx, struct iocb *iocb, struct io_event *result)	ctx 表示文件异步 I/O 的上下文描述符，iocb 是要取消的异步 I/O 操作，而 result 表示这个操作的执行结果	取消之前使用 io_submit 提交的一个文件异步 I/O 操作。返回 0 表示成功。
int io_getevents(aio_context_t ctx, long min_nr,long_nr, struct io_event *events, struct timespec *timeout)	ctx 表示文件异步 I/O 的上下文描述符，获取的已完成事件个数范围是[min_nr,nr]，events 是执行完成的事件数组，timeout 是超时时间，也就是获取 min_nr 个时间前的等待时间。	从已完成的文件异步 I/O 操作队列中读取操作。

文件异步 I/O 中有一个核心结构体--struct iocb，其定义如下：

```
struct iocb {
    /*存储业务需要的指针，与 io_getevents 方法返回的 io_event 结构体的 data 成员一致*/
    u_int64_t aio_data;
    u_int32_t PADDED(aio_key, aio_reserved1);
    u_int16_t aio_lio_opcode;   //操作码
    int16_t aio_reqprio;   //请求的优先级
    u_int32_t aio_fildes;   //异步 I/O 操作的文件描述符
    u_int64_t aio_buf;   //读/写对应的用户态缓冲区
    u_int64_t aio_nbytes;   //读/写操作的字节长度
    int64_t aio_offset;   //读/写操作对应的文件中的偏移量
    u_int64_t aio_reserved2;
    /*
     *设置为 IOCB_FLAG_RESFD,表示当有异步 I/O 请求完成时让内核使用 eventfd 进行通
     *知，这是与 epoll 配合使用的关键
     */
    u_int32_t aio_flags;
    /*当 aio_flags 为 IOCB_FLAG_RESFD 时，用于事件通知的 eventfd 句柄*/
    u_int32_t aio_resfd;
}
```

上述的 struct iocb 结构体中，aio_flags 和 aio_read 两个结构体成员正是 Nginx

中将文件异步 I/O、eventfd 以及 epoll 机制结合起来一起使用的关键，三者的结合也使得 Nginx 中的文件异步 I/O 同网络事件的处理一样高效。另外有一点需要说明的就是 Nginx 中目前只使用了异步 I/O 中的读操作，即 struct iocb 结构体中的成员 aio_lio_opcode 的值为 IO_CMD_PREAD，因为文件的异步 I/O 不支持缓存操作，而正常写文件的操作往往是写入内存中就返回，而如果使用异步 I/O 方式写入的话反而会使得速度下降。

struct iocb 用在提交和取消异步 I/O 事件中，而通过 io_getevents 获取已经完成的 I/O 事件时则用到的是另一个十分重要的结构体—struct io_event，其定义如下：

```
struct io_event {
        uint64_t data;    //与提交事件时的 iocb 结构体的 aio_data 成员一致
        uint64_t obj;    //指向提交事件时对应的 iocb 结构体
        int64_t res;    //异步 I/O 操作的结果，res 大于等于 0 表示成功，小于 0 失败
        int64_t res2;    //保留字段
}
```

在 Nginx 中，主要用到的字段就是 data 和 res。其中 data 中保存的是文件异步 I/O 事件对象，res 就是保存异步 I/O 的结果。

在简单了解了 Linux 内核提供的异步 I/O 系统调用及其在 Nginx 中涉及到的相关知识后，再来讲述下 eventfd 系统调用，因为这是 Nginx 将异步 I/O 事件集成到 epoll 中的一个桥梁。为什么这么说呢？通过上面对异步 I/O 结构体 struct iocb 结构体的分析，我们知道，当成员 aio_flags 设置为 IOCB_FLAG_RESFD 时，表明使用 eventfd 句柄来进行异步 I/O 事件的完成通知。正是这个 eventfd，让其可以使用 epoll 机制来对其进行监控，从而间接对异步 I/O 事件完成进行监控，保证了事件驱动模块对网络事件和文件异步 I/O 事件的处理接口保持一致。

eventfd 系统调用原型如下：

```
int eventfd(unsigned int initval, int flags);
```

eventfd 系统调用常用与进程之间通信或者用于内核与应用程序之间通信。在 Nginx 中正是利用了内核会在异步 I/O 事件完成时通过 eventfd 通知 Nginx 来完成对异步 I/O 事件的间接监控。

在简单介绍完 Linux 内核提供的异步 I/O 接口以及 eventfd 系统调用后，接下来开始分析文件异步 I/O 事件是如何在 ngx_epoll_module 中实现的。下面是涉及到的主要全局变量，可以分为两部分：

1)、系统调用相关：

int ngx_eventfd = -1; 这个是用于通知异步 I/O 事件完成的描述符，在 Nginx 中它会赋值给 struct iocb 结构体中的 aio_resfd 成员，也是 epoll 监控的描述符。

aio_context_t ngx_aio_ctx = 0; 这个就是异步 I/O 接口会使用到的异步 I/O 上下文，并且需要经过 io_setup 初始化后才能使用。

2)、与网络事件处理兼容相关。

static ngx_event_t ngx_eventfd_event; 这个就是 eventfd 描述符对应的读事件对象。因为文件异步 I/O 事件完成后，内核通知应用程序 eventfd 有可读事件(EPOLLIN)发生。然后应用程序就会调用读事件回调函数进行处理。

static ngx_connection_t ngx_eventfd_conn; 这个就是 eventfd 描述符对应的连接对象。

为什么要称它们是与网络事件处理兼容呢？回想下 Nginx 在处理网络事件的

时候会为 socket 获取一个连接对象，然后设置连接对象 ngx_connection_t 的 fd 成员为 socket 描述符，接着设置连接的读事件和写事件，并设置对应的事件回调函数，最后将读/写事件（或整个连接）加入到 epoll 中监控。对应地处理文件异步 I/O 事件时，首先是让 I/O 事件的完成通知用 eventfd 来完成，然后设置 eventfd 的读事件及其处理函数，再用一个连接对象来保存 eventfd 和读事件，并将 eventfd 加入到 epoll 监控。这样就保证了 Nginx 内核可以像处理网络事件一样处理文件异步 I/O 事件。但是 Nginx 内核处理文件异步 I/O 事件又有其特别的地方。因为当 epoll 中监控到 eventfd 有读事件完成时，只可以说明 Linux 内核通知 Nginx 有文件异步 I/O 事件完成了，此时 Nginx 还并不知道有哪些或有几个异步 I/O 事件完成了，可以这么理解，eventfd 仅仅是 Linux 内核用来通知 Nginx 有异步 I/O 事件完成了。那 Nginx 又是如何获取完成的异步 I/O 事件的呢，这就是 eventfd 描述符关联的读事件回调函数所需要完成的工作了，这个后面进行详细说明。

现假设有一个模块需要读取磁盘中的文件，那么如果 Nginx 启动了文件异步 I/O 处理的话，那么这个读盘的操作会被 Nginx 作为一个异步 I/O 事件来处理。

因为要将这个读盘事件以异步 I/O 方式来处理，那么首先就需要初始化一个异步 I/O 上下文，在 Nginx 中代码如下：

```
static void
ngx_epoll_aio_init(ngx_cycle_t *cycle, ngx_epoll_conf_t *epcf)
{
    int    n;
    struct epoll_event    ee;

#if (NGX_HAVE_SYS_EVENTFD_H)
    ngx_eventfd = eventfd(0, 0); //调用 eventfd()系统调用可以创建一个 efd 描述符
#else
    ngx_eventfd = syscall(SYS_eventfd, 0);
#endif
    ……
    n = 1;
    /*设置 ngx_eventfd 为非阻塞*/
    if (ioctl(ngx_eventfd, FIONBIO, &n) == -1) {
        ……
    }
    /*
     *初始化文件异步 io 上下文，aio_requests 表示至少可以处理的异步文件 io 事件
     *个数
     */
    if (io_setup(epcf->aio_requests, &ngx_aio_ctx) == -1) {
        ……
    }
    /*设置异步 io 完成时通知的事件*/
     /* ngx_event_t->data 成员通常就是事件对应的连接对象*/
    ngx_eventfd_event.data = &ngx_eventfd_conn;
    ngx_eventfd_event.handler = ngx_epoll_eventfd_handler;
```

```
        ngx_eventfd_event.log = cycle->log;
        ngx_eventfd_event.active = 1;   //active 为 1(下面会加入到 epoll 中监控)
        ngx_eventfd_conn.fd = ngx_eventfd;
        ngx_eventfd_conn.read = &ngx_eventfd_event;
        /*文件异步 io 对应的连接对象读事件为 ngx_eventfd_event*/
        ngx_eventfd_conn.log = cycle->log;

        ee.events = EPOLLIN|EPOLLET;    //监控 eventfd 读事件并设置为 ET 模式

        /*
        *ngx_eventfd 被监控到有读事件发生时，会利用 ee.data.ptr 获取对应的连接对象,
        *详见 ngx_epoll_process_events()
        */
        ee.data.ptr = &ngx_eventfd_conn;

        /*
        * 将异步文件 io 的通知的描述符加入到 epoll 监控中，因为在 ngx_file_aio_read( )
        *函数中将 struct iocb 结构体的 aio_flags
        * 成员赋值为 IOCB_FLAG_RESFD，他会告诉内核当有异步 io 请求处理完成时使用
        *eventfd 描述符通知应用程序，这使得异步 io、
        * eventfd 和 epoll 可以结合起来使用。另外，将将 struct iocb 结构体的 aio_resfd 设
        *置为 ngx_eventfd，那么当有异步 io 事件
        * 完成时，epoll 就会收到 ngx_eventfd 描述符的读事件，然后
        *ngx_epoll_process_events()中会调用其读事件回调函数，即
        * ngx_epoll_eventfd_handler 处理内核的通知。
        */
        if (epoll_ctl(ep, EPOLL_CTL_ADD, ngx_eventfd, &ee) != -1) {
            return;
        }
        ……
    }
```

通过调用 ngx_epoll_aio_init 方法，Nginx 就将异步 I/O 以 eventfd 为桥梁与 epoll 结合起来了。

初始化完异步 I/O 上下文后，模块就可以提交文件异步 I/O 事件了。在此之前需要再了解下 Nginx 封装的一个异步 I/O 事件的对象，如下：

```
    struct ngx_event_aio_s {
        void    *data;
        /*由业务模块实现，用于在异步 I/O 事件完成后进行业务相关的处理*/
        ngx_event_handler_pt    handler;
        ngx_file_t      *file;   //文件异步 I/O 涉及的文件对象
        ……
        ngx_fd_t      fd;   //异步 I/O 将要操作的文件描述符
        ……
        /*aiocb 就是 struct iocb 类型的，异步 I/O 事件控制块*/
```

```
    ngx_aiocb_t                     aiocb;
    ngx_event_t                     event;    //异步 I/O 对应的事件对象
};
```

那 Nginx 中是如何处理异步 I/O 事件的提交的呢？其代码实现如下：

```
ssize_t
ngx_file_aio_read(ngx_file_t *file, u_char *buf, size_t size, off_t offset,
    ngx_pool_t *pool)
{
    ngx_err_t             err;
    struct iocb           *piocb[1];
    ngx_event_t           *ev;
    ngx_event_aio_t       *aio;
   ……
    /*
     * ngx_event_aio_t 封装的异步 io 对象，如果 file->aio 为空，需要初始化
     * file->aio
     */
    if (file->aio == NULL && ngx_file_aio_init(file, pool) != NGX_OK) {
        return NGX_ERROR;
    }
    aio = file->aio;
    ev = &aio->event;
     ……
    /*提交异步事件之前要初始化结构体 struct iocb*/
    ngx_memzero(&aio->aiocb, sizeof(struct iocb));

    /*
     *  将 struct iocb 的 aio_data 成员赋值为异步 io 的事件对象，下面提交异步 I/O 事
     *  件之后，等该事件完成，在通过 io_getevents()
     *  获取到事件后，对应的 struct io_event 结构体中的 data 成员就会指向这个事件。
     * struct iocb 的 aio_data 成员和 struct io_event 的 data 成员指向的是同一个东西
     */
    aio->aiocb.aio_data = (uint64_t) (uintptr_t) ev;
    aio->aiocb.aio_lio_opcode = IOCB_CMD_PREAD;
    aio->aiocb.aio_fildes = file->fd;
    aio->aiocb.aio_buf = (uint64_t) (uintptr_t) buf;
    aio->aiocb.aio_nbytes = size;
    aio->aiocb.aio_offset = offset;
   /*
    *设置 IOCB_FLAG_RESFD 当内核有异步 io 请求处理完时通过 eventfd 通知应用程序
     */
    aio->aiocb.aio_flags = IOCB_FLAG_RESFD;
    aio->aiocb.aio_resfd = ngx_eventfd;    //这个就是 eventfd 描述符
```

```
        /*
        * 将异步 I/O 对应的事件处理函数设置为 ngx_file_aio_event_handler。当
        *io_getevents()函数中获取到该异步 io 事件时，会调用该回调函数，在 Nginx 中并
        *不是直接调用，而是先将其加入到
        * ngx_posted_event 队列，等遍历完所有完成的异步 io 事件后，再依次调用所有
        *事件的回调函数
        */
        ev->handler = ngx_file_aio_event_handler;

        piocb[0] = &aio->aiocb;

        /*
        * 将该异步 io 请求加入到异步 io 上下文中，等待 io 完成，内核会通过 eventfd 通
        * 知应用程序
        */
        if (io_submit(ngx_aio_ctx, 1, piocb) == 1) {
            ev->active = 1;
            ev->ready = 0;
            ev->complete = 0;

            return NGX_AGAIN;
        }
        ……
}
```

在模块提交了文件异步 I/O 事件后，在事件完成之后，Linux 就会触发 eventfd 的读事件来告诉 Nginx 异步 I/O 事件完成了，我们知道，当 epoll 监控到 eventfd 有事件发生时，在 ngx_epoll_process_events()函数中会通过 epoll_wait 取出该事件，然后通过 struct epoll_event 结构体中的 data.ptr 成员获取 eventfd 对应的连接对象(在上面有介绍)，并调用连接对象中的读事件处理函数 ngx_epoll_eventfd_handler()，而 Nginx 正是通过这个读事件处理函数来获取真正完成的文件异步 I/O 事件，这个读事件处理函数正是在 ngx_epoll_aio_init()函数中进行注册的。该函数的实现如下：

```
static void
ngx_epoll_eventfd_handler(ngx_event_t *ev)
{
    int                 n, events;
    long                i;
    uint64_t            ready;
    ngx_err_t           err;
    ngx_event_t         *e;
    ngx_event_aio_t     *aio;
    struct io_event     event[64];    //一次性最多处理 64 个异步 io 事件
    struct timespec     ts;
    /*
```

```
*通过 read 获取已经完成的事件数，并设置到 ready 中，注意这里的 ready 可以大于 64
*/
n = read(ngx_eventfd, &ready, 8);
……
ts.tv_sec = 0;
ts.tv_nsec = 0;

while (ready) {
      /*
      *从已完成的异步 io 队列中读取已完成的事件，返回值代表获取的事件个数
      */
      events = io_getevents(ngx_aio_ctx, 1, 64, event, &ts);
     ……
      if (events > 0) {
            ready -= events;    //计算剩余已完成的异步 io 事件
            for (i = 0; i < events; i++) {
                  /*
                   * data 成员指向这个异步 io 事件对应着的实际事件，这个与 struct iocb
                   *  结构体中的 aio_data 成员是一致的。
                   * struct iocb 控制块中的 aio_data 成员被赋予对应 io 事件对象是在函
                   *  数 ngx_file_aio_read()中实现的。
                   */
                  e = (ngx_event_t *) (uintptr_t) event[i].data;
                  ……
                  /*
                   *  异步 io 事件 ngx_event_t->data 成员指向的就是 ngx_event_aio_t 对
                   *象，这个在 ngx_file_aio_init()函数中可以看到
                   */
                  aio = e->data;
                  aio->res = event[i].res;    //res 成员代表的是异步 io 事件执行的结果
                 /*将异步 io 事件加入到 ngx_posted_events 普通读写事件队列中*/
                  ngx_post_event(e, &ngx_posted_events);
            }
            continue;
      }

      if (events == 0) {
            return;
      }
      /* events == -1 */
      ngx_log_error(NGX_LOG_ALERT, ev->log, ngx_errno,
                              "io_getevents() failed");
      return;
}
```

}

一般来说业务模块在文件异步 I/O 事件完成后都需要在进行一些和业务相关的处理，那 Nginx 又是怎么实现的呢？当然是通过注册回调函数的方法来实现的。在通过 ngx_epoll_eventfd_handler()回调函数获取到已经完成的文件异步 I/O 事件并加入到 ngx_posted_events 队列中后，执行 ngx_posted_events 队列中的事件时就会回调异步 I/O 事件完成的回调函数 ngx_file_aio_event_handler(在 ngx_file_aio_read 注册)，然后在该函数中调用业务模块(提交文件异步 I/O 事件的模块)实现的回调方法，这个方法一般都是在业务模块提交异步 I/O 事件前注册到上面介绍的 ngx_event_aio_t 的 handler 成员中。其中异步 I/O 事件完成后的回调函数实现如下：

```
static void
ngx_file_aio_event_handler(ngx_event_t *ev)
{
    ngx_event_aio_t    *aio;

    /*
     * 获取事件对应的 data 对象，即 ngx_event_aio_t，这个在 ngx_file_aio_init()函数中初
     * 始化的
     */
    aio = ev->data;
    ……
    /*
     * 这个回调是由真正的业务模块实现的，举个例子如果是 http cache 模块，则会在
     * ngx_http_file_cache_aio_read()函数中
     * 调用完 ngx_file_aio_read()后设置为 ngx_http_cache_aio_event_handler()进行业务逻
     * 辑的处理，为什么要在调用完
     *   ngx_file_aio_read()之后再设置呢，因为可能业务模块一开始并没有为 ngx_file_t 对
     * 象设置 ngx_event_aio_t 对象，而是在
     * ngx_file_aio_read()中调用 ngx_file_aio_init()进行初始化的。
     */
    aio->handler(ev);
}
```

到这里 Nginx 中设计到的文件异步 I/O、eventfd 和 epoll 的配合使用就介绍完了。