Linux IO 复用之 epoll 总结

作者：selfjt
原文链接：https://ld246.com/article/1573026280655
来源网站：链滴

linux IO复用之epoll总结

一、前言

《UNIX网络编程》里并没有提到epoll，不知道为啥，以下的内容是根据linux manual总结的。

二、API介绍

epoll是在linux上提供的实现IO复用的机制。epoll与poll类似，可以同时监听多个描述符；epoll新增边缘触发和水平触发的概念，而且在处理大量描述符时更有优势。

epoll API中核心概念就是epoll实例，它是一个内核内的数据结构，从用户角度来看它可以简单的看包含了两个list：

- interest list（或者叫epoll set）：用户注册的感兴趣的描述符集合
- ready list：就绪的描述符集合，当有IO就绪时内核会自动将就绪的描述符加到ready list中

epoll API包含三个系统调用：

epoll_create

```
int epoll_create(int size);
int epoll_create1(int flags);
```

epoll_create创建一个epoll实例，函数会返回一个指向epoll实例的描述符，在使用完毕后应该调用close关闭epoll实例。size参数类似map的capacity，

标识epoll实例维护的描述符的数量。

epoll_create1与epoll_create相相似，但参数变成了flags，size则被忽略。这里的flags有一个可选

：EPOLL_CLOEXEC，EPOLL_CLOEXEC表示在创建的描述符上设置FD_CLOEXEC标志。

epoll_ctl

```
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);

/* Valid opcodes ( "op" parameter ) to issue to epoll_ctl().  */
#define EPOLL_CTL_ADD 1 /* Add a file decriptor to the interface.  */
#define EPOLL_CTL_DEL 2 /* Remove a file decriptor from the interface.  */
#define EPOLL_CTL_MOD 3 /* Change file decriptor epoll_event structure.  */

typedef union epoll_data {
    void *ptr;
    int fd;
    uint32_t u32;
    uint64_t u64;
};

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

epoll_ctl将描述符和感兴趣的事件注册到epoll实例，这个函数相当于把描述符添加到epoll实例的interest list中。函数操作成功时返回0，否则返回-1并设置errno。

epoll_wait

```
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
```

epoll_wait会阻塞等待IO事件，可以理解为从ready list里获取描述符。函数返回就绪描述的个数，并将就绪的描述符存储到events参数中，

通过timeout可以设置以毫秒为单位的超时时间，-1表示永不超时。

三、边缘触发和水平触发

关于边缘触发和水平触发的介绍有很多，这里就翻译一下man手册里的内容好了。

epoll提供两种触发机制：edge-triggered (ET) 和 level-triggered (LT)，它们的区别可以通过以下例子来说明

- 假设我们已有一个描述符 rfd，我们将从它读取一个pipe输出，我们将其注册到epoll实例中，感趣的事件为可读
- pipe的写入端写入了2KB的数据到pipe
- 进程调用了 epoll_wait，这时rfd会被放到ready list中然后成功返回
- pipe的读取端从pipe读取了1KB的数据
- 进程又一次调用 epoll_wait

如果rfd在注册到epoll实例时使用了EPOLLET选项，那么上述第5步调用epoll_wait可能会发生阻塞，管这时读取缓冲区里仍有可读取的数据；而同时pipe的另一端可能在等待着相应的响应，于是陷入了尽的互相等待。而出现这种现象的原因在于ET仅在描述符发生变化时才会返回事件。在上面的例子当，第2步会产生一个事件，而第3步会消费这个事件。因为第4步没有读取完所有的数据，所以第5步可会陷入无限期的阻塞。

而linux manual建议的边缘触发的使用方式如下：

- 配合非阻塞描述符使用
- 直到每次　read或者write返回EAGAIN时才继续等待下一次事件

与边缘触发不同，当使用水平触发选项时，epoll就相当于poll的升级版， 可以简单地替换poll。

总的来说，ET和LT的区别在于触发事件的条件不同，LT比较符合编程思维（有满足条件的就触发），T触发的条件更苛刻一些（仅在发生变化时才触发），对使用者的要求也更高，理论效率更高。值得提的是java nio的selector会根据操作系统不同采用不同的实现，在linux 2.6及以后的版本中使用的是epoll，采用的是水平触发；而netty中提供的额外的EpollEventLoop则采用了边缘触发。

在监听描述符事件时，同一个描述符上可能会连续发生多个事件，这是用户可以选择设置EPOLLONEHOT选项来通知epoll禁用后续的事件。如果设置了EPOLLONESHOT选项，在事件处理完毕后用户要重新注册事件。这个选项在并发环境更加有用。

当多个进程或者线程同时监听一个epoll实例上的一个描述符时，使用EPOLLET选项可以保证每次事只会通知一个进程或者线程，避免类似“惊群”的问题。

epoll监听的限制

/proc/sys/fs/epoll/max_user_watches 中的配置限制了同一个用户在所有epoll实例中能监听的描符的总数。

四、使用边缘触发的例子

因为水平触发和poll的使用方式区别不大，这里仅展示边缘触发的示例：

```
#define MAX_EVENTS 10
struct epoll_event ev, events[MAX_EVENTS];
int listen_sock, conn_sock, nfds, epollfd;

/* 此处省略调用listen_sock调用socket、bind和listen的过程 */

//创建epoll实例，程序最后应该调用close关闭epollfd
epollfd = epoll_create1(0);
if (epollfd == -1)
{
    perror("epoll_create1");
    exit(EXIT_FAILURE);
}

ev.events = EPOLLIN; //感兴趣的事件为读事件
ev.data.fd = listen_sock; //注册fd为监听套接字

//注册event
if (epoll_ctl(epollfd, EPOLL_CTL_ADD, listen_sock, &ev) == -1)
```

```
    {
        perror("epoll_ctl: listen_sock");
        exit(EXIT_FAILURE);
    }

    for (;;)
    {
        //等待描述符就绪，参数-1表示不超时
        nfds = epoll_wait(epollfd, events, MAX_EVENTS, -1);
        if (nfds == -1)
        {
            perror("epoll_wait");
            exit(EXIT_FAILURE);
        }

        for (n = 0; n < nfds; ++n)
        {
            if (events[n].data.fd == listen_sock)
            {
                //监听套接字就绪，调用accept建立连接
                conn_sock = accept(listen_sock,
                                   (struct sockaddr *)&addr, &addrlen);
                if (conn_sock == -1)
                {
                    perror("accept");
                    exit(EXIT_FAILURE);
                }
                //设置新连接为非阻塞模式（ET下必须设置非阻塞）
                setnonblocking(conn_sock);
                //感兴趣的事件为读事件，同时设置为边缘触发
                ev.events = EPOLLIN | EPOLLET;
                //注册fd为新建立的连接描述符
                ev.data.fd = conn_sock;
                //注册event
                if (epoll_ctl(epollfd, EPOLL_CTL_ADD, conn_sock,
                              &ev) == -1)
                {
                    perror("epoll_ctl: conn_sock");
                    exit(EXIT_FAILURE);
                }
            } else {//新建立的连接就绪
                //do_use_fd应该对fd进行read或者write直到EAGAIN，然后记录当前的read或write进度
等到下次就绪后再继续
                do_use_fd(events[n].data.fd);
            }
        }
    }
```

在边缘触发模式下，如果希望在事件到来时不立刻进行操作，而是等其他条件就绪后再进行read或write，这时可以同时注册EPOLLIN|EPOLLOUT事件以提高性能，而不是反复调用epoll_ctl通过EPOLL_CTL_MOD在EPOLLIN和EPOLLOUT之间来回切换，如果在水平模式下就不能这样做了，因为感兴趣的事一旦就绪的事件就会持续发生，带来不必要的消耗。

五、为什么epoll比poll更快

epoll的介绍里提到epoll比poll更快，根据网上的其他博客总结了以下几点原因：

- 等待描述符就绪时，不需要每次都将描述符集合传递到内核，而是将描述符注册到epoll实例，由epoll实例内部维护全部的描述符集合
- epoll实例内部使用了红黑树和内核cache区维护描述符集合，提高了描述符集合注册和删除操作的率
- epoll内部通过回调机制维护ready list。当有描述符就绪时就将其放到ready list中，调用epoll_wai时只需要判断ready list是否为空即可，如果不为空则将ready list复制到用户空间并清空ready list；则陷入睡眠
- 有描述符就绪时不需要重新遍历所有描述符，epoll会直接返回就绪的描述符集合

这里顺便提一下LT的实现，epoll_wait在返回就绪描述符前会检查描述符的触发类型，如果是水平触并且描述符上有未处理的数据，则会将其加入刚才清空的ready list，这样下次调用epoll_wait时ready ist仍会有该描述符。这也是LT和ET的表现的差别的实际原因。

六、鸣谢

- 感谢原创作者

七、转载

- 转载