图解 epoll 是如何工作的及epoll实现原理

Hu先生的Linux
Linux服务器开发技术群720209036

本文包含以下内容：

- epoll是如何工作的

本文不包含以下内容：

- epoll 的用法
- epoll 的缺陷

epoll实现原理由视频讲解：

C/C++ Linux服务器开发高级架构学习视频点击：C/C++Linux服务器开发/Linux后台架构师-学习视频教程

epoll原理剖析以及reactor模型应用

基于linux epoll网络编程细节处理

> 我实在非常喜欢像epoll这样使用方便、原理不深却有大用处的东西，即使它可能已经比较老了

select 和 poll 的缺点

epoll 对于动辄需要处理上万连接的网络服务应用的意义可以说是革命性的。对于普通的本地应用，select 和 poll可能就很好用了，但对于像C10K这类高并发的网络场景，select 和 poll就捉襟见肘了。

看看他们的**API**

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);

int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

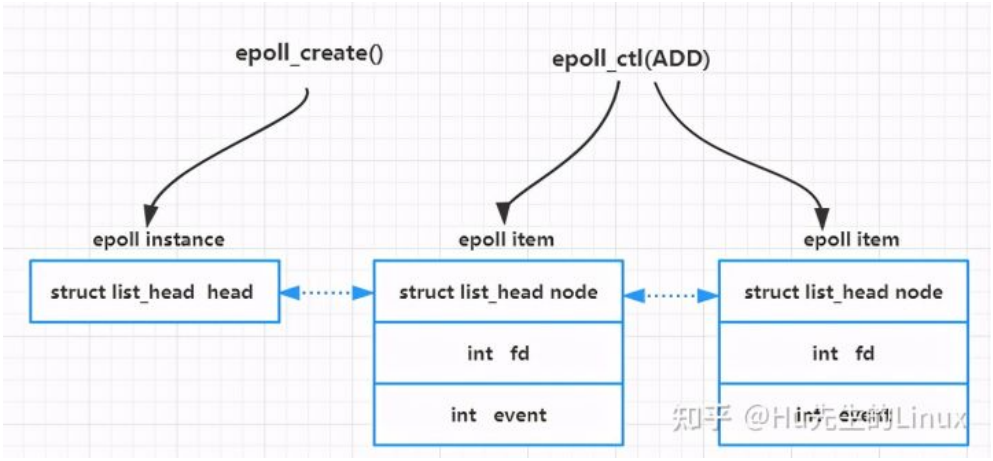
它们有一个共同点，用户需要将**监控**的文件描述符集合打包当做参数传入，每次调用时，这个集合都会从用户空间**拷贝**到内核空间，这么做的原因是内核对这个集合是无记忆的。对于绝大部分应用，这是一种十足的浪费，因为应用需要监控的描述符在大部分时间内基本都是不变的, 也许会有变化, 但都不大.

epoll 对此的改进

1. 描述符添加 --- 内核可以记下用户关心哪些文件的哪些事件.
2. 事件发生 --- 内核可以记下哪些文件的哪些事件真正发生了, 当用户前来获取时, 能把结果提供给用户.

描述符添加

既然要有记忆, 那么理所当然的内核需要需要一个数据结构来记, 这个数据结构简单点就像下面这个图中的epoll_instance, 它有一个链表头，链表上的元素epoll_item就是用户添加上去的， 每一项都记录了描述符fd和感兴趣的事件组合event

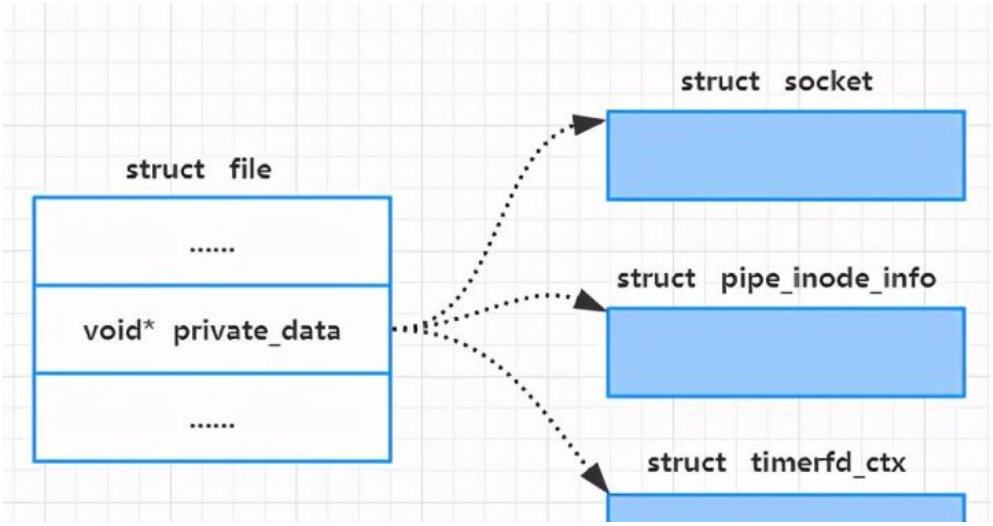


事件发生

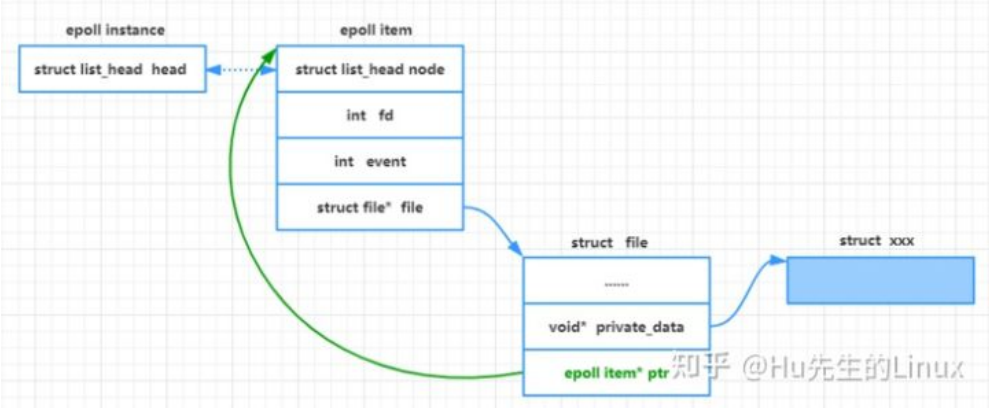
事件有多种类型, 其中POLLIN表示的**可读**事件是用户使用的最多的。比如:

- 当一个 TCP 的socket收到报文，它会变得可读；
- 当一个pipe受到对端发送的数据，它会变得可读;
- 当一个timerfd对应的定时器超时，它会变得可读;

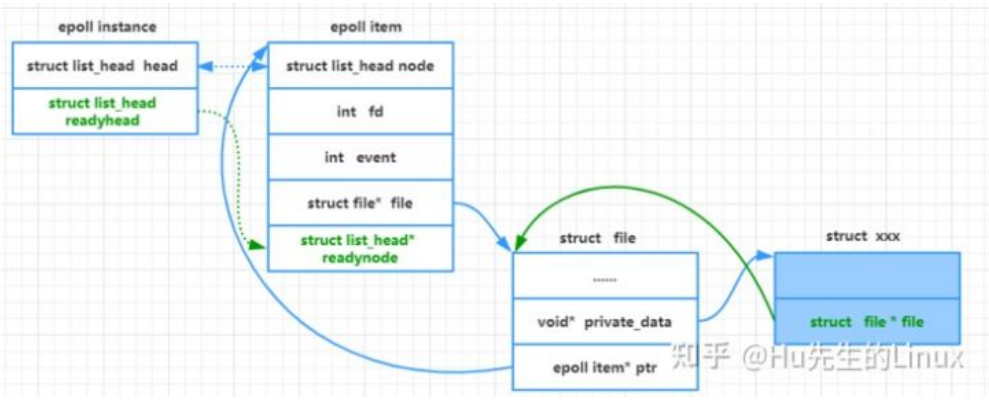
那么现在需要将这些**可读**事件和前面的epoll_instance关联起来。linux中，每一个文件描述符在内核都有一个struct file结构对应, 这个struct file有一个private_data指针，根据文件的实际类型，它们指向不同的数据结构。



那么我能想到的最方便的做法就是epoll_item中增加一个指向struct file的指针，在struct file中增加一个指回epoll item的指针。



为了能记录有事件发生的文件，我们还需要在epoll_instance中增加一个就绪链表readylist，在private_data指针指向的各种数据结构中增加一个指针回指到 struct file，在epoll item中增加一个挂接点字段，当一个文件可读时，就把它对应的epoll item挂接到epoll_instance



在这之后，用户通过系统调用下来读取readylist就可以知道哪些文件就绪了。

好了，以上纯属我个人一拍脑袋想到的epoll大概的工作方式，其中一定包含不少缺陷。

不过真实的epoll的实现思想上与上面也差不多，下面来说一下

创建 epoll 实例

如同上面的epoll_instance，内核需要一个数据结构保存记录用户的注册项，这个结构在内核中就是struct eventpoll， 当用户使用epoll_create(2)或者epoll_create1(2)时，内核fs/eventpoll.c实际就会创建一个这样的结构.

```
error = ep_alloc(&ep);
```

这个结构中比较重要的部分就是几个链表了，不过实例刚创建时它们都是空的，后续可以看到它们的作用

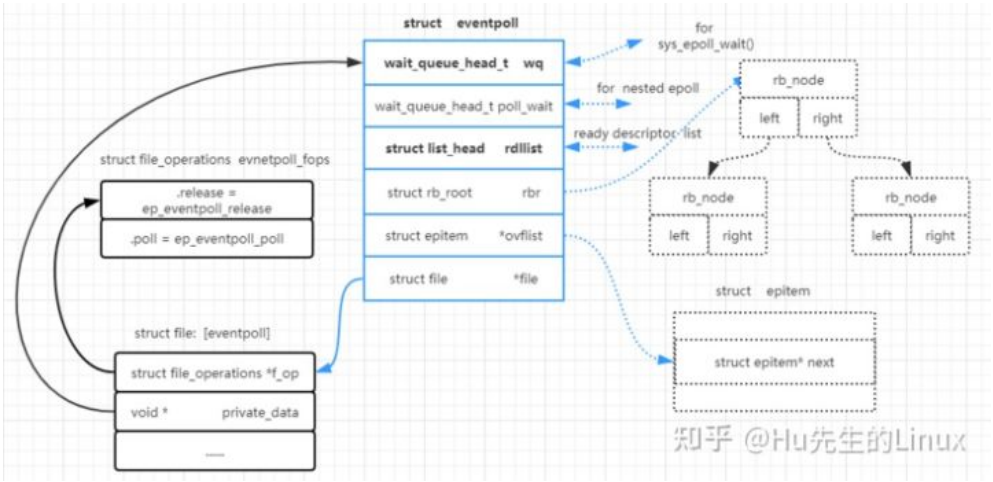
epoll_create()最终会向用户返回一个文件描述符，用来方便用户之后操作该 **epoll 实例**，所以在创建**epoll 实例**之后，内核就会分配一个文件描述符fd和对应的struct file结构

```
fd = get_unused_fd_flags(O_RDWR | (flags & O_CLOEXEC));

file = anon_inode_getfile("[eventpoll]", &eventpoll_fops, ep,
                          O_RDWR | (flags & O_CLOEXEC));
```

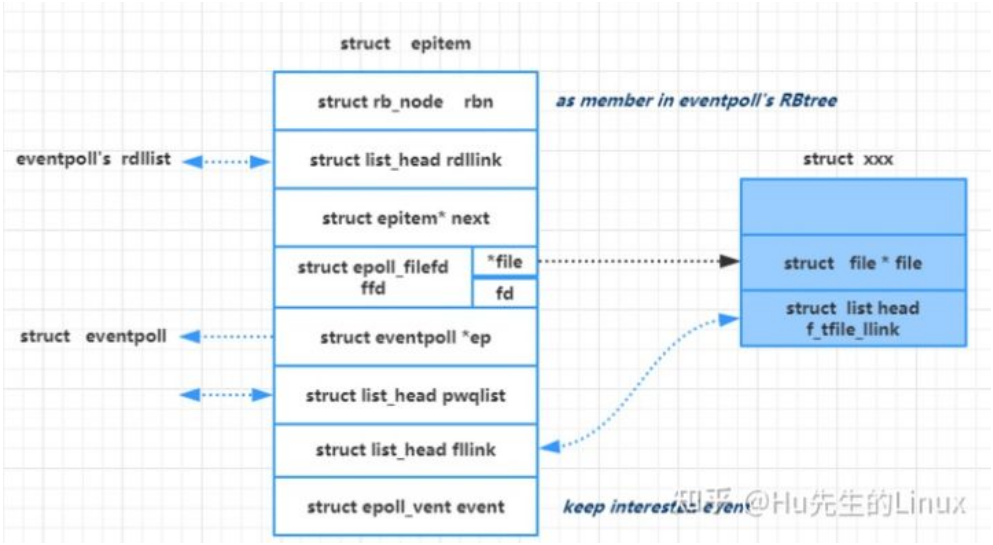
最后就是把它们和刚才的**epoll 实例** 关联起来，然后向用户返回fd

```
ep->file = file;

fd_install(fd, file);

return fd;
```

完成后，**epoll 实例** 就成这样了。



用户可以通过 epoll_ctl(2)向 **epoll 实例** 添加要监控的描述符和感兴趣的事件。如同前面的epoll item，内核实际创建的是一个叫struct epitem的结构作为注册表项。如下图所示



为了在描述符很多时的也能有较高的搜索效率, **epoll 实例** 以红黑树的形式来组织每个struct epitem (取代上面例子中链表)。struct epitem结构中ffd是用来记录关联文件的字段, 同时它也作为该表项添加到红黑树上的**Key**；

rdllink的作用是当fd对应的文件准备好 (关心的事件发生) 时，内核会将它作为挂载点挂接到**epoll 实例**中ep->rdllist链表上
fllink的作用是作为挂载点挂接到fd对应的文件的file->f_tfile_llink链表上，一般这个链表最多只有一个元素，除非发生了dup。
pwqlist是一个链表头，用来连接 poll wait queue。虽然它是链表，但其实链表上最多只会再挂接一个元素。

创建struct epitem的代码在fs/evnetpoll.c的ep_insert()中

```
if (!(epi = kmem_cache_alloc(epi_cache, GFP_KERNEL)))
    return -ENOMEM;
```

之后会进行各个字段初始化

```
INIT_LIST_HEAD(&epi->rdllink);
INIT_LIST_HEAD(&epi->fllink);
INIT_LIST_HEAD(&epi->pwqlist);
epi->ep = ep;
ep_set_ffd(&epi->ffd, tfile, fd);
epi->event = *event;
epi->nwait = 0;
epi->next = EP_UNACTIVE_PTR;
```

然后是设置局部变量epq

```
struct ep_pqueue epq;
```

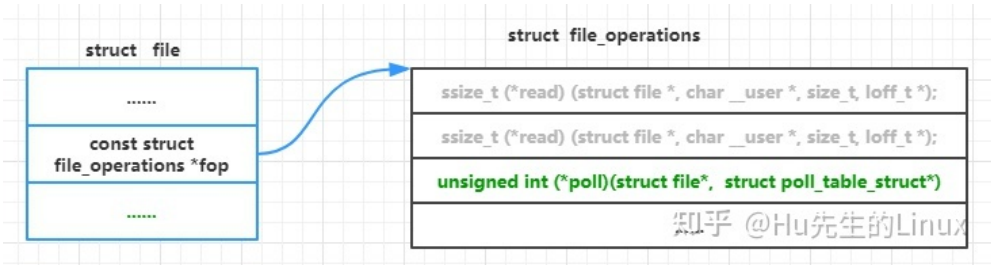
epq的数据结构是struct ep_pqueue, 它是poll table的一层包装 (加了一个struct epitem* 的指针)

```
struct  ep_pqueue{
    poll_table pt;
    struct epitem* epi;
}
```

poll table包含一个函数和一个事件掩码

```
typedef void (*poll_queue_proc)(struct file *, wait_queue_head_t *, struct poll_table_

typedef struct poll_table_struct {
    poll_queue_proc _qproc;
    unsigned long _key;
}poll_table;
```

这个poll table用在哪里呢？答案是, 用在了struct file_operations的poll操作 (这和本文开始说的select`poll`不是一个东西)




```
struct file_operations {

   unsigned int (*poll)(struct file*,  struct poll_table_struct*);

}
```

不同的文件有不同poll实现方式, 但一般它们的实现方式差不多是下面这种形式

```
static unsigned int XXXX_poll(struct file *file, poll_table *wait)
{
    私有数据 = file->private_data;
    unsigned int events = 0;

    poll_wait(file, &私有数据->wqh, wait);

    if (文件可读了)
        events |= POLLIN;

    return events;
}
```

1. 将**XXX**放到文件私有数据的等待队列上 (一般file->private_data中都有一个等待队列头 wait_queue_head_t wqh), 至于**XXX**是啥, 各种类型文件实现各异, 取决于poll_table参数
2. 查询是否真的有事件了, 若有则返回.

> 有兴趣的读者可以 timerfd_poll() 或者 pipe_poll() 它们的实现

poll_wait的实现很简单, 就是调用poll_table中设置的函数, 将文件私有的等待队列当作了参数.

```
static inline void poll_wait(struct file * filp, wait_queue_head_t * wait_address, pol
{
    if (p && p->_qproc && wait_address)
        p->_qproc(filp, wait_address, p);
}
```

回到 ep_insert()

所以这里设置的poll_table就是ep_ptable_queue_proc().

然后

```
revents = ep_item_poll(epi, &epq.pt)
```

看其实现可以看到, 其实就是主动去调用文件的poll函数. 这里以 TCP socket文件为例好了 (毕竟网络应用是最广泛的)

```
unsigned int tcp_poll(struct file *file, struct socket *sock, poll_table *wait)
{
    sock_poll_wait(file, sk_sleep(sk), wait);

}
```

可以看到, 最终还是调用到了poll_wait(), 所以注册的ep_ptable_queue_proc()会执行

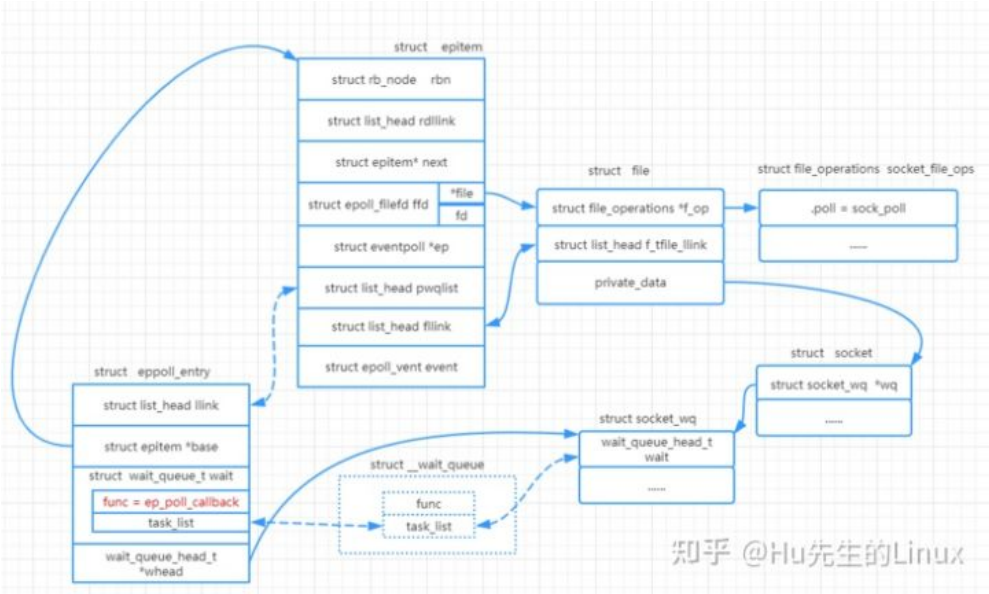
```
struct epitem *epi = ep_item_from_epqueue(pt);
    struct eppoll_entry *pwq;

    pwq = kmem_cache_alloc(pwq_cache, GFP_KERNEL)
```

这里面, 又分配了一个struct eppoll_entry结构. 其实它和struct epitem 结构是一一对应的.

随后就是一些初始化

```
init_waitqueue_func_entry(&pwq->wait, ep_poll_callback);
    pwq->whead = whead;
    pwq->base = epi;

    add_wait_queue(whead, &pwq->wait)
    list_add_tail(&pwq->llink, &epi->pwqlist);
    epi->nwait++;
```

现在, struct epitem 和struct eppoll_entry的关系就像下面这样



文件可读之后

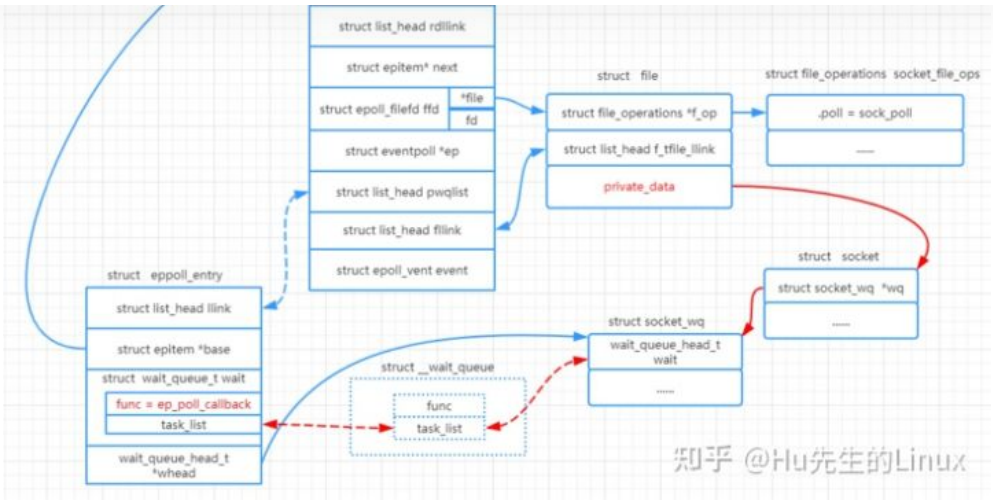
对于 TCP socket, 当收到对端报文后, 最初设置的sk->sk_data_ready函数将被调用

```
void sock_init_data(struct socket *sock, struct sock *sk)
{

     sk->sk_data_ready   =    sock_def_readable;

}
```

经过层层调用, 最终会调用到 __wake_up_common 这里面会遍历挂在socket.wq上的等待队列上的函数

```
static void __wake_up_common(wait_queue_head_t *q, unsigned int mode,
            int nr_exclusive, int wake_flags, void *key)
{
    wait_queue_t *curr, *next;

    list_for_each_entry_safe(curr, next, &q->task_list, task_list) {
        unsigned flags = curr->flags;

        if (curr->func(curr, mode, wake_flags, key) &&
                (flags & WQ_FLAG_EXCLUSIVE) && !--nr_exclusive)
            break;
    }
}
```

于是, 顺着图中的这条红色轨迹, 就会调用到我们设置的ep_poll_callback, 那么接下来就是要让epoll实例能够知有文件已经可读了



先从入参中取出当前表项epi和ep

```
struct epitem *epi = ep_item_from_wait(wait);
    struct eventpoll *ep = epi->ep;
```

再把epi挂到ep的就绪队列

```
if (!ep_is_linked(&epi->rdllink)) {
        list_add_tail(&epi->rdllink, &ep->rdllist)
    }
```

接着唤醒阻塞在 (如果有) 该epoll实例的用户.

```
waitqueue_active(&ep->wq)
```

用户获取事件

谁有可能阻塞在epoll实例的等待队列上呢? 当然就是使用epoll_wait来从epoll实例获取发生了感兴趣事件的的描述符的用户.
epoll_wait会调用到ep_poll()函数.

```
if (!ep_events_available(ep)) {

        init_waitqueue_entry(&wait, current);
        __add_wait_queue_exclusive(&ep->wq, &wait);
```

如果没有事件, 我们就将自己挂在epoll实例的等待队列上然后睡去.....
如果有事件, 那么我们就要将事件返回给用户

```
ep_send_events(ep, events, maxevents)
```