从 4 个方面分析 epoll 的实现原理

Linux爱好者 2022-09-06 20:50 Posted on 浙江

前言

本文以四个方面介绍epoll的实现原理：

1. epoll的数据结构；
2. 协议栈如何与epoll通信；
3. epoll线程安全如何加锁；
4. ET与LT的实现。

epoll的数据结构

多种数据结构进行决策

epoll至少需要两个集合

1. 所有fd的总集
2. 就绪fd的集合

那么这个总集选用什么数据结构存储呢？我们知道，一个fd，其底层对应一个TCB。那么也就是说key=fd,val=TCB,是一个典型的kv型数据结构，对于kv型数据结构我们可以使用以下三种进行

存储。

1. hash
2. 红黑树
3. b/b+tree

如果使用hash进行存储，其优点是查询速度很快，O(1)。但是在我们调用epoll_create()的时候，hash底层的数组创建多大合适呢？如果我们有百万的fd，那么这个数组越大越好，如果我们仅仅十几个fd需要管理，在创建数组的时候，太大的空间就很浪费。而这个fd我们又不能预先知道有多少，所以hash是不合适的。

b/b+tree是多叉树，一个结点可以存多个key，主要是用于降低层高，用于磁盘索引的，所以在我们这个内存场景下也是不适合的。

在内存索引的场景下我们一般使用红黑树来作为首选的数据结构，首先红黑树的查找速度很快,O(log(N))。其次在调用epoll_create()的时候，只需要创建一个红黑树树根即可，无需浪费额外的空间。

那么就绪集合用什么数据结构呢，首先就绪集合不是以查找为主的，就绪集合的作用是将里面的元素拷贝给用户进行处理，所以集合里的元素没有优先级，那么就可以采用线性的数据结构，使用队列来存储，先进先出，先就绪的先处理。

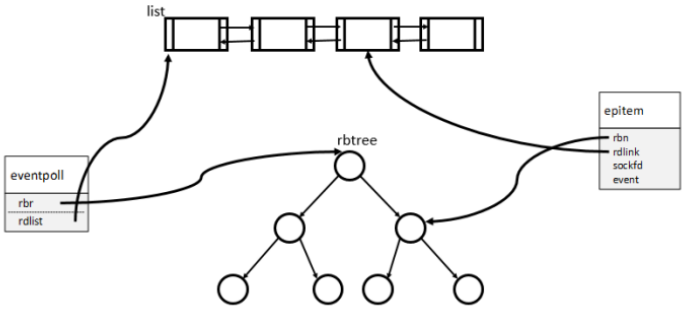
1. 所有fd的总集 -----> 红黑树
2. 就绪fd的集合 -----> 队列

红黑树和就绪队列的关系

红黑树的结点和就绪队列的结点的同一个节点，所谓的加入就绪队列，就是将结点的前后指针联系到一起。所以就绪了不是将红黑树结点delete掉然后加入队列。他们是同一个结点，不需要delete。

```
struct epitem {
    RB_ENTRY(epitem) rbn;
    LIST_ENTRY(epitem) rdlink;
    int rdy; //exist in list

    int sockfd;
    struct epoll_event event;
};

struct eventpoll {
    ep_rb_tree rbr;
    int rbcnt;
```

```
    int rbcnt;

    LIST_HEAD( ,epitem) rdlist;
    int rdnum;

    int waiting;

    pthread_mutex_t mtx; //rbtree update
    pthread_spinlock_t lock; //rdlist update

    pthread_cond_t cond; //block for event
    pthread_mutex_t cdmtx; //mutex for cond

};
```

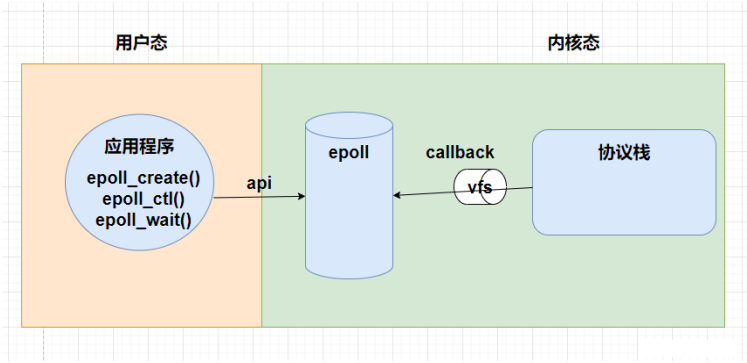


协议栈如何与epoll模块通信

epoll的工作环境

应用程序只能通过三个api接口来操作epoll。当一个io准备就绪的时候，epoll是怎么知道io准备就绪了呢？是由协议栈将数据解析出来触发回调通知epoll的。

也就是说可以把epoll的工作环境看出三部分，左边应用程序的api，中间的epoll，右边是协议栈的回调(协议栈当然不能直接操作epoll，中间的vfs在此不是重点，就直接省略vfs这一层了)。



协议栈触发回调通知epoll的时机

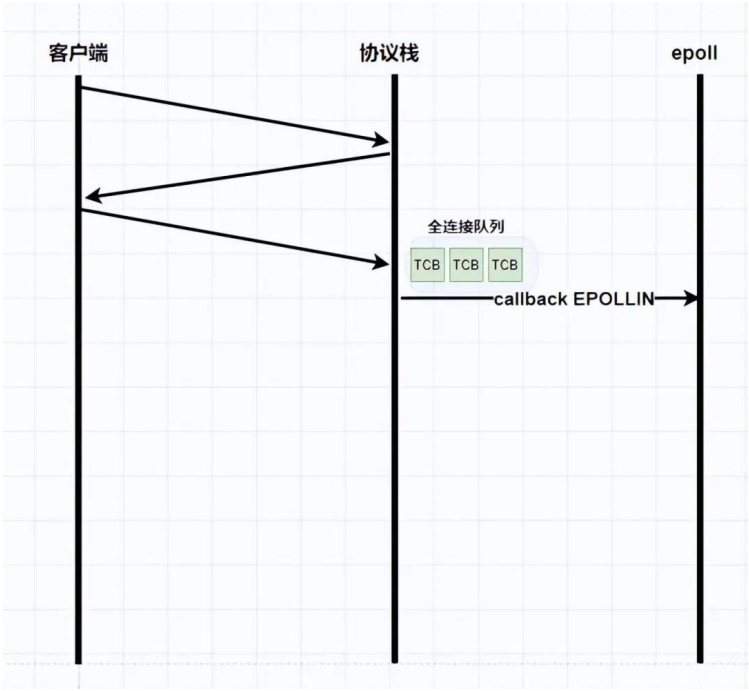
socket有两类，一类是监听listenfd，一类是客户端clientfd。对于sockfd而言，我们一般比较关注EPOLLIN和EPOLLOUT这两个事件，所以如果是listenfd，我们通常的做法就是accept。对于clientfd来说，如果可读我们就recv，如果可写我们就send。

协议栈将数据解析出来触发回调通知epoll。epoll是怎么知道哪个io就绪了呢？我们从ip头可以解析出源ip，目的ip和协议，从tcp头可以解析出源端口和目的端口，此时五元组就凑齐了。
socket fd --- < 源IP地址 , 源端口 , 目的IP地址 , 目的端口 , 协议 > 一个fd就是一个五元组，知道了fd，我们就能从红黑树中找到对应的结点。

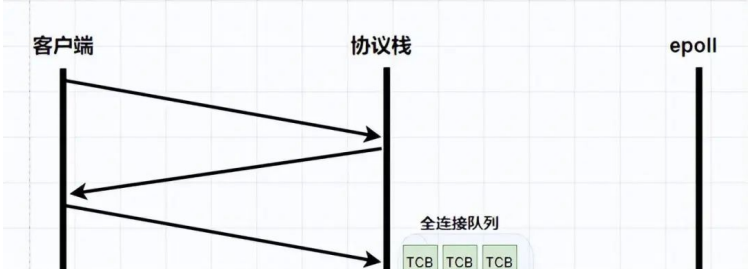
那么这个回调函数做什么事情呢？我们传入fd和具体事件这两个参数，然后做下面两个操作

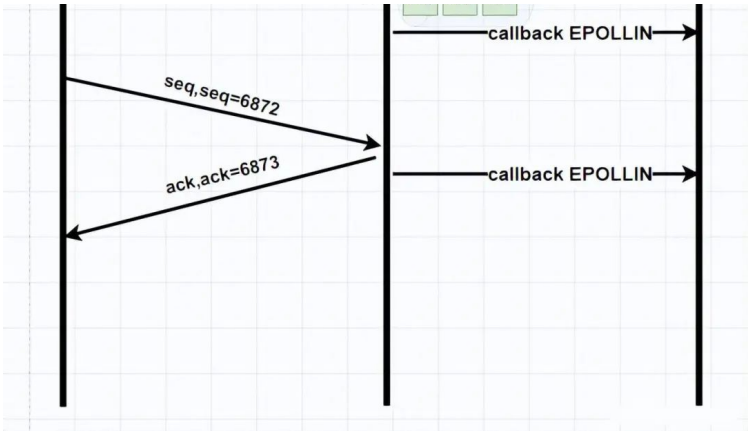
```
1. 通过fd找到对应的结点
2. 把结点加入到就绪队列
```

1. 协议栈中，在三次握手完成之后，会往全连接队列中添加一个TCB结点，然后触发一个回调函数，通知到epoll里面有个EPOLLIN事件

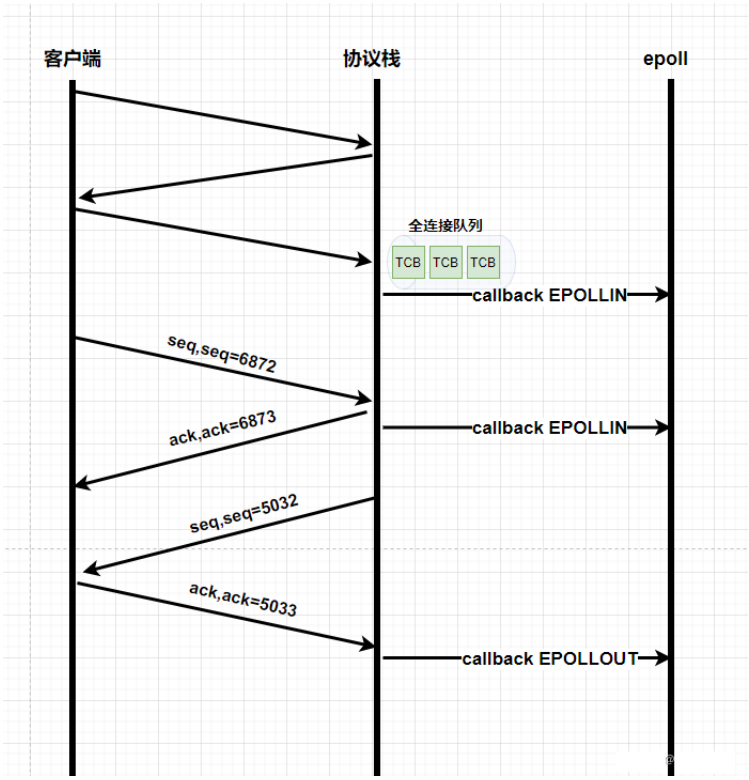


2. 客户端发送一个数据包，协议栈接收后回复ACK，之后触发一个回调函数，通知到epoll里面有个EPOLLIN事件

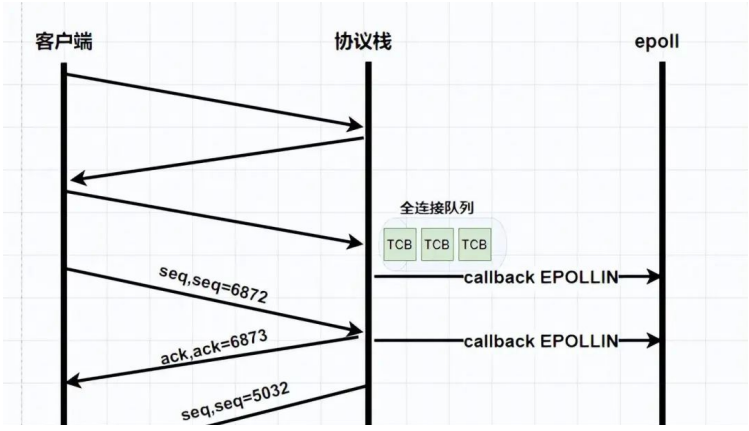


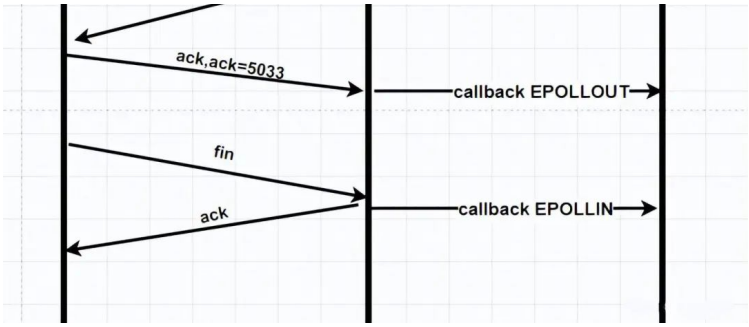


3. 每个连接的TCB里面都有一个sendbuf，在对端接收到数据并返回ACK以后，sendbuf就可以将这部分确认接收的数据清空，此时sendbuf里面就有剩余空间，此时触发一个回调函数，通知到epoll里面有个EPOLLOUT事件

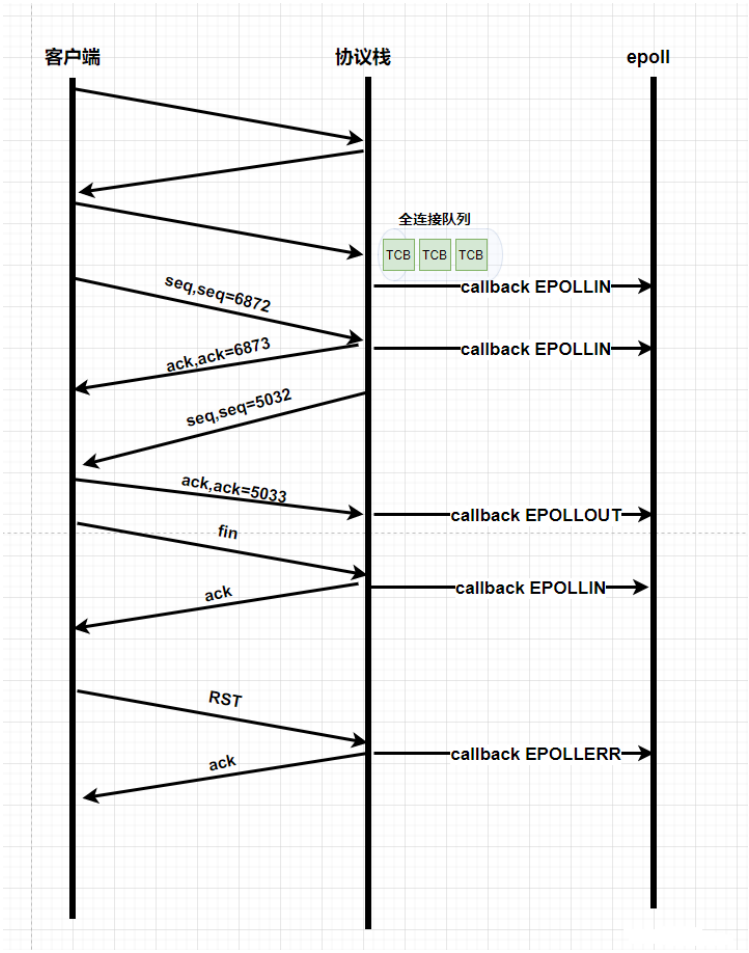


4. 当对端发送close，在接收到fin后回复ACK，此时会调用回调函数，通知到epoll有个EPOLLIN事件





5. 当接收到rst标志位的时候，回复ack之后也会触发回调函数，通知epoll有一个EPOLLERR事件



通知的时机总结

一个有5个通知的地方

1. 三次握手完成之后
2. 接收数据回复ACK之后
3. 发送数据收到ACK之后
4. 接收FIN回复ACK之后
5. 接收RST回复ACK之后

从回调机制看epoll 与 select/poll的区别

由于select和poll没有本质的区别，所以下面统一称为poll。

```
//  poll跟select类似，其实poll就是把select三个文件描述符集合变成一个集合了。
int select(int nfds, fd_set * readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

我们看到每次调用poll，都需要把总集fds拷贝到内核态，检测完之后，再有内核态拷贝的用户态，这就是poll。而epoll不是这样，epoll只要有新的io就调用epoll_ctl()加入到红黑树里面，一旦有触发就用epoll_wait()将有事件的结点带出来，可以看到他们的第一个区别：poll总是拷贝总集，如果有100w个fd，只有两三个就绪呢？这会造成大量资源浪费；而epoll总是将需要拷贝的东西进行拷贝，没有浪费。

第二个区别：我们从上面知道了epoll的事件都是由协议栈进行回调然后加入到就绪队列的，而poll呢？内核如何检测poll的io是否就绪？只能通过遍历的方法判断，所以poll检测io通过遍历的方法也是比较慢的。

所以两者的区别：

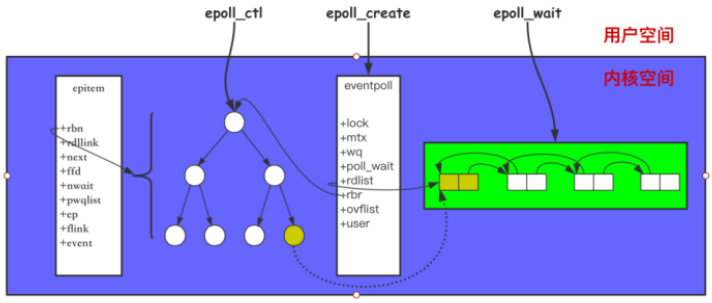
1. select/poll需要把总集copy到内核，而epoll不用
2. 实现原理上面，select/poll 需要循环遍历总集是否有就绪，而epoll是那个结点就绪了就加入就绪队列里面。

注意：poll不一定就比epoll慢，在io量小的情况下，poll是比epoll快的，而在大io量下，epoll绝对是有主导地位的。至于有多少个io才算多，其实也很难说，一般认为500或者1024为分界点（我乱说的）。

epoll线程安全如何加锁

3个api做什么事情

1. epoll_create() ===》创建红黑树的根节点
2. epoll_ctl() ===》add,del,mod 增加、删除、修改结点
3. epoll_wait() ===》把就绪队列的结点copy到用户态放到events里面，跟recv函数很像



分析加锁

如果有3个线程同时操作epoll，有哪些地方需要加锁？我们用户层面一共就只有3个api可以使用：

1. 如果同时调用 epoll_create() ，那就是创建三颗红黑树，没有涉及到资源竞争，没有关系。
2. 如果同时调用 epoll_ctl() ，对同一颗红黑树进行，增删改，这就涉及到资源竞争需要加锁了，此时我们对整棵树进行加锁。
3. 如果同时调用epoll_wait() ，其操作的是就绪队列，所以需要对就绪队列进行加锁。

我们要扣住epoll的工作环境，在应用程序调用 epoll_ctl() ，协议栈会不会有回调操作红黑树结点？调用epoll_wait() copy出来的时候，协议栈会不会操作操作红黑树结点加入就绪队列？综上所述：

```
epoll_ctl() 对红黑树加锁
epoll_wait()对就绪队列加锁
回调函数()    对红黑树加锁,对就绪队列加锁
```

那么红黑树加什么锁，就绪队列加什么锁呢？

对于红黑树这种节点比较多的时候，采用互斥锁来加锁。就绪队列就跟生产者消费者一样，结点是从协议栈回调函数来生产的，消费是epoll_wait()来消费。那么对于队列而言，用自旋锁（对于队列而言，插入删除比较简单，cpu自旋等待比让出的成本更低，所以用自旋锁）。

ET与LT如何实现

- ET边沿触发，只触发一次
- LT水平触发，如果没有读完就一直触发

代码如何实现ET和LT的效果呢？水平触发和边沿触发不是故意设计出来的，这是自然而然，水到渠成的功能。水平触发和边沿触发代码只需要改一点点就能实现。

从协议栈检测到接收数据，就调用一次回调，这就是ET，接收到数据，调用一次回调。而LT水平触发，检测到recvbuf里面有数据就调用回调。所以ET和LT就是在使用回调的次数上面的差异。

那么具体如何实现呢？协议栈流程里面触发回调，是天然的符合ET只触发一次的。那么如果是LT，在recv之后，如果缓冲区还有数据那么加入到就绪队列。那么如果是LT，在send之后，如果缓冲区还有空间那么加入到就绪队列。那么这样就能实现LT了。

```
ssize_t nty_recv(int sockid, char *buf, size_t len, int flags) {
    ...
    int event_remaining = 0;
    if (socket->epoll & NTY_EPOLLIN) {
        if (!(socket->epoll & NTY_EPOLLET) && rcv->recvbuf->merged_len > 0) {
            event_remaining = 1;
        }
    }
    if (event_remaining) {
        if (socket->epoll) {
            nty_epoll_add_event(tcp->ep, USR_SHADOW_EVENT_QUEUE, socket, NTY_EPOLLIN);
        }
    }
    ...
}
```

```
ssize_t nty_send(int sockid, const char *buf, size_t len) {
    ...
    if (snd->snd_wnd > 0) {
        if ((socket->epoll & NTY_EPOLLOUT) && !(socket->epoll & NTY_EPOLLET)) {
            nty_epoll_add_event(tcp->ep, USR_SHADOW_EVENT_QUEUE, socket, NTY_EPOLLOUT);
        }
    }
    ...
}
```

> 作者：cheems~
> https://blog.csdn.net/qq_42956653/article/details/125941045

- EOF -