epoll

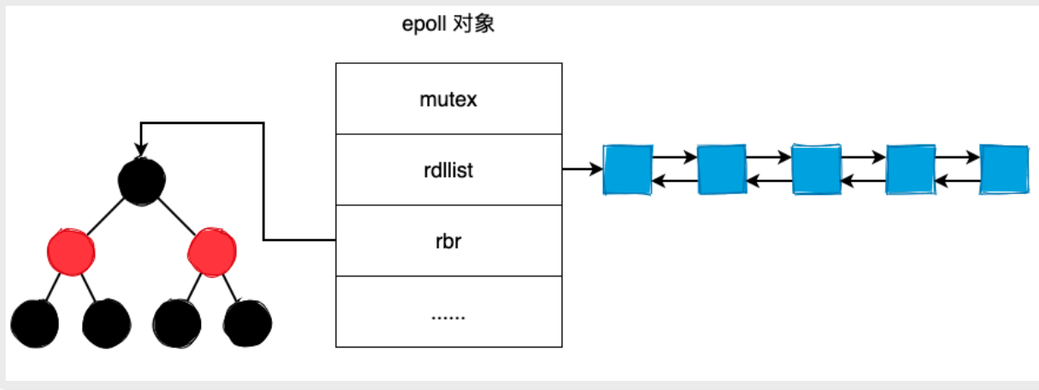
基本概念

- epoll 允许在多个非阻塞的 socket 描述符上等待可读、可写事件，本质上是一个事件驱动模型
- 简单来说，假设我们当前的 server 有 10 万个 TCP 连接，在这些 TCP 连接这种，能够读/写数据的连接并不是 10 万，可能只有 5000，或者更少，这是因为用户不可能实时活跃
- 如果说我们能够直接找出这 5000 个活跃的连接进行处理的话，那么系统效率将得到巨大的提升，epoll 的本质作用就是在这 10 万个连接中找到这 5000 个活跃连接

基本原理

epoll_create()

- **int epoll_create(int size);**
  - 从 Linux 2.6.8 版以来， size 参数被忽略不用，因此我们随便传一个正整数即可
  - 函数返回的一个 epoll 句柄，把它当作是我们创建的 epoll 实例 ID 即可
- **epoll_create() 系统调用将创建一个 epoll 对象，该对象有两个最为核心的结构体成员: 红黑树以及双向链表，epoll 对象中会保存 RBTree 的 root 节点，以及双向链表的 head 节点，初始化时两者均为 NULL**



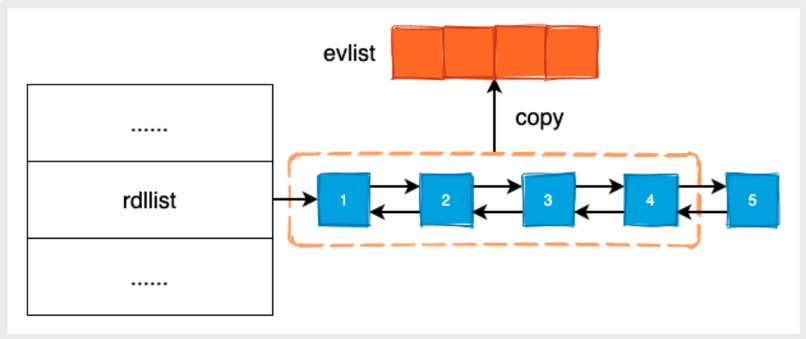
- 其中 RBTree 的节点就表示一个一个的事件，而双向链表中的节点则表示已经就绪的事件

epoll_ctl()

- **int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);**
- **系统调用 epoll_ctl() 能够修改由文件描述符 epfd 所代表的 epoll 实例中的兴趣列表，更进一步地，epoll_ctl() 其实就是对 epoll 实例中的 RBTree 进行节点的添加、修改和删除**
- 形参
  - int epfd
    - 该参数自然不用多说，指定我们到底要操作哪个 epoll 实例，也就是 epoll_create() 的返回值
  - int op
    - 该参数指明我们要对 epoll 实例做何种操作，添加，或者更新，或者是删除
    - EPOLL_CTL_ADD
      - 将文件描述符 fd 添加至 epoll 实例中，这里的 fd 大部分情况下都是 socket 描述符
      - **本质上就是在 RBTree 中新增一个节点，其 key 为我们指定的文件描述符 fd，value 则是一个名为 epitem 的对象**
      - **除此以外，epoll 最为精髓的地方就在于此时会添加等待事件到 scoket 的等待队列中，并设置回调函数为 ep_poll_callback。当 socket 中有对应事件发生时，OS 将调用该回调函数。而这个回调函数所做的事情，其实就是将 RBTree 中的节点扔到 rdllist 就绪队列中，表示事件已经就绪**
      - 因此我们可以看到 epoll 的本质还是回调，并且由于在 socket 实现代码中添加对应的入口，所以 epoll 的移植性不高
    - EPOLL_CTL_MOD — 修改描述符 fd 上设定的事件
    - EPOLL_CTL_DEL — 将文件描述符 fd 从 RBTree 中移除
  - int fd — socket 文件描述符，也可以是 POSIX 消息队列、inotify 实例、管道或者是 FIFO 描述符，唯独不能是普通文件或者是目录的文件描述符。因为在 Linux 看到，相比于 socket、命名管道，文件 I/O 是"快速 I/O"，即要么成功，要么失败，不可能永久阻塞
  - struct epoll_event *event — 篇幅有限，于下篇描述
- epoll_ctl() 的源码可在 linux/fs/eventpoll.c 中查看，对应于函数 ep_insert()、ep_modify() 以及 ep_remove()

epoll_wait()

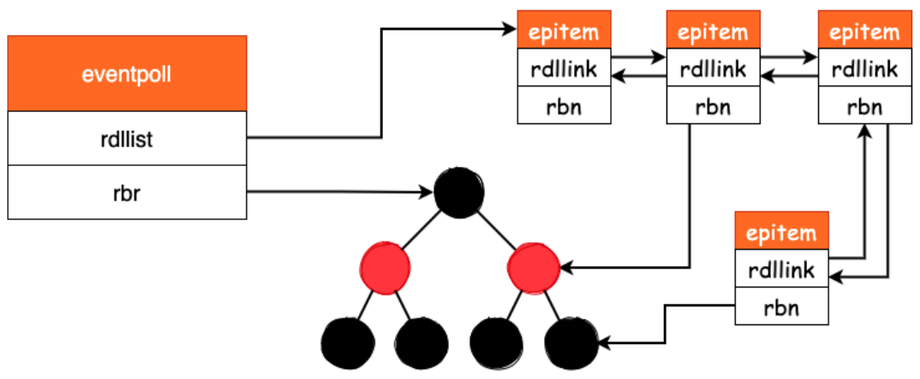
- **int epoll_wait(int epfd, struct epoll_event *evlist, int maxevents, int timeout);**
- 系统调用 epoll_wait() 返回 epoll 实例中处于就绪态的文件描述符信息。单个 epoll_wait() 调用能返回多个就绪态文件描述符的信息，这些描述符将被保存到形参 evlist 中，也就是说，evlist 应该是一个数组
- **前面我们提到了 epoll 实例中的 rdllist 保存了已经就绪的事件，或者说可读/可写的文件描述符。那么，epoll_wait() 做的事情就是将双向链表中的节点复制到用户提供的数组中，并将已复制的节点从双向链表中移除**



- 虽然 rdllist 中有 5 个已就绪节点，但是 evlist 只有 4 个位置，所以只能移动 4 个节点出去
- rdllist 中的内容由内核进行维护
- epoll_wait() 调用将返回已就绪的文件描述符个数，其值可能会小于 maxevents，但一定不会大于它。maxevents 则表示我们一次 epoll_wait() 调用最多从 rdllist 复制多少个节点到 evlist 中

节点详情

- 在前面我们提到了 epoll 实例中有一棵红黑树保存全部事件，还有一个双向链表来保存就绪事件，在画图的时候将其分开了，但实际上它们是共享节点的
- **也就是说，对于节点 epi 来说，它有可能既是 RBTree 上的节点，也有可能是 rdllist 中的节点**



- 也就是说，epitem 即是红黑树的节点，同时也是双向链表的节点
- 这样一来，我们既能够通过 fd 利用红黑树的特性，在 O(logn) 的平均时间复杂度内找到对应的事件，同时也能够在 O(K) 的时间复杂度内找到所有已准备就绪的事件
- 同时，我们并没有使用额外的空间来存储已就绪的事件

流程梳理

1. 首先调用 epoll_create() 创建 epoll 实例，此时内部的 RBTree、双向链表均为空
2. 而后，调用 epoll_ctl() 并将 EPOLL_CTL_ADD 传入，将 socketfd、事件等信息注册至 epoll 中
   1. 首先，为 epitem 分配空间
   2. 添加等待事件到 socket 的等待队列中，这个等待队列是 Linux TCP/IP 实现的一部分，并添加回调函数 ep_poll_callback
   3. 将 epitem 插入至红黑树中，并且以 socketfd 为 key，使得我们能够在 O(logn) 的时间复杂度查找到 socketfd 对应的节点
3. 当 socketfd 上有可读、可写事件发生时，内核将调用先前注册的回调函数，也就是 ep_poll_callback。该函数做的事情就是将 epitem 节点添加至 rdllist 双向链表中，表示事件已就绪
4. 当我们调用 epoll_wait() 时，该函数会将 rdllist 中的数据拷贝至我们传入的 evlist 中，并从双向链表中移除该节点。若此时 rdllist 为空，那么 epoll_wait() 调用将一直阻塞，直到所管理的 scoketfd 上有事件发生为止