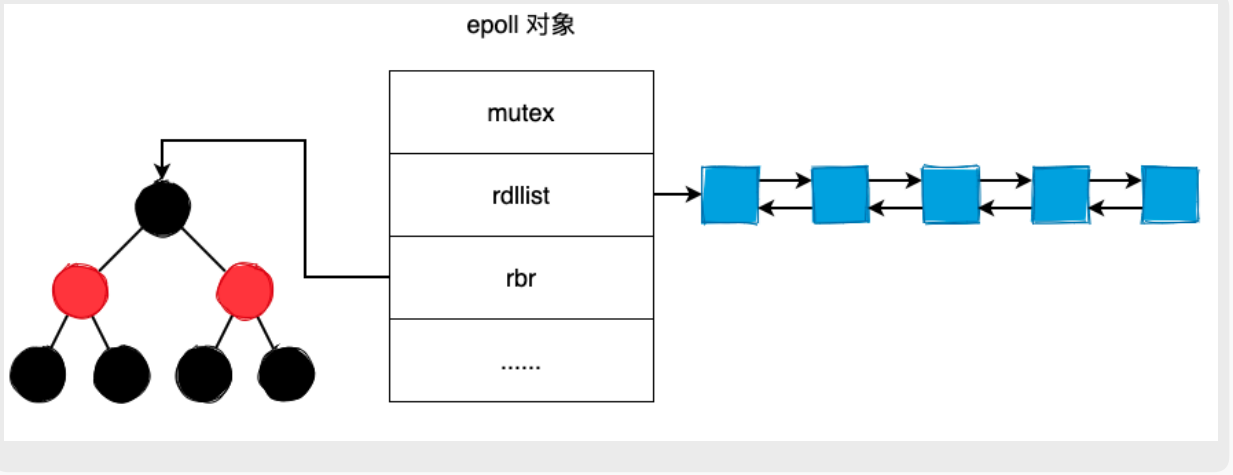
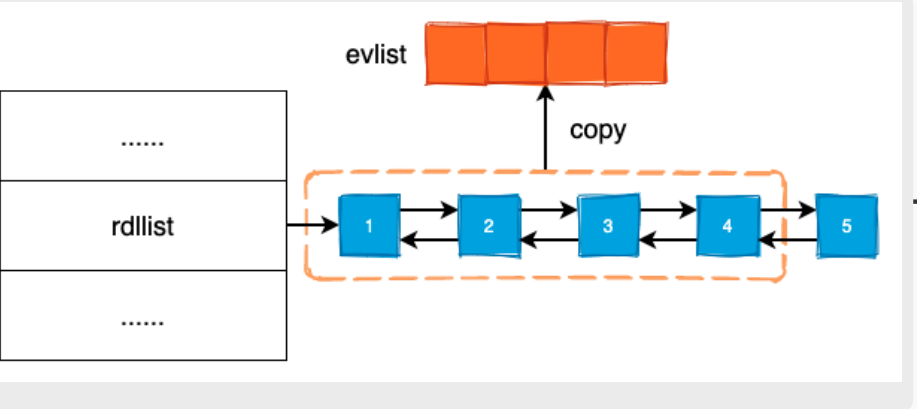


epoll

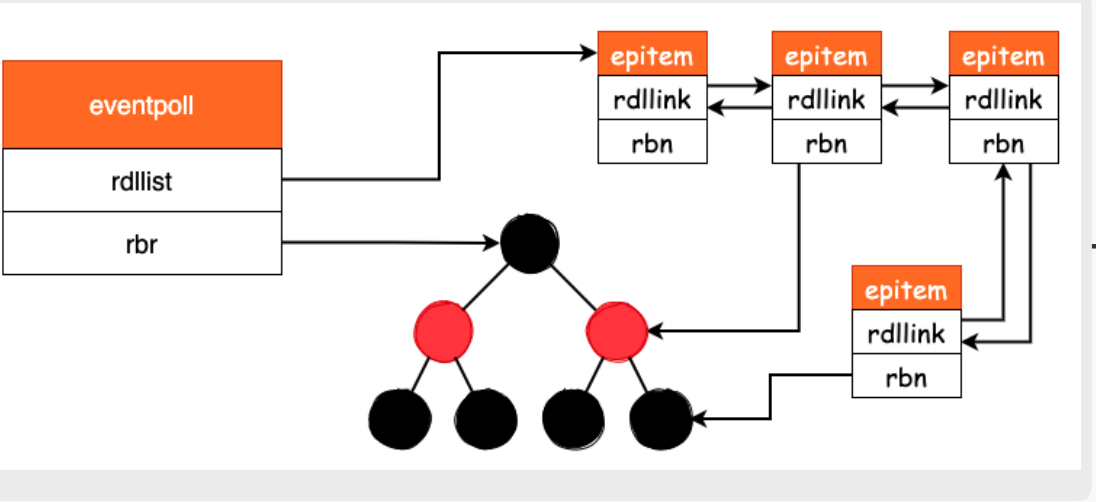
基本概念

- 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 **struct**，我们将在后续描述
 - 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（通常），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
 - struct epoll_event *event** — 于下方详述
 - 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
 - 
 - 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，同时也能在 **O(n)** 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

边缘触发与水平触发

- 边缘触发 (ET, Edge Trigger)**
 - 边缘触发的意思就是当可读/可写事件发生时，内核只会通知一次，后续不再通知
 - 我们以 **recv()** 为例来描述边缘触发模式
 - 假设我们在读取接收缓冲区数据时，每次就取 2 字节的数据，并且我们假设接收到的任何一个 TCP 包的数据均为 20 字节
 - 当有新的 TCP 包进入接收缓冲区时，内核就会将 **epitem** 移入双向链表中，并解除 **epoll_wait()** 的阻塞（如果被阻塞的话），我们一般把这个过程称之为通知
 - 当我们使用 **recv()** 仅接收 2 字节之后，位于双向链表中的 **epitem** 将会被移除，直到下一个 TCP 包到达。如果说没有下一个包了，那么我们将永远都不可能获取剩下的 18 字节数据
 - 所以，在边缘触发模式下，我们在处理可读事件时，必须要尽可能多的读取缓冲区中的数据，直到没有其它数据为止。同时，也正因为内核仅通知一次的特性，使得边缘触发模式的执行效率非常之高
- 水平触发 (LT, Level Trigger)**
 - 水平触发的意思就是只要有可读/可写事件发生，并且应用程序没有处理时，那么就会一直在双向链表中
 - 还是上面儿的例子，在水平触发模型下，**recv()** 读了 2 字节之后，还剩 18 个字节，此时该 TCP 连接依然是可读的。所以下一次的 **epoll_wait()** 也会返回该 **event**，应用程序可以继续读取剩下的字节
 - 正因为内核会多次的把同一个事件"重复"地通知，所以水平触发模式又称为慢速模式，效率要比边缘触发模式低。但是胜在稳妥，OS 会为我们兜底。