IO 复用

一、为什么使用 IO 复用

在操作系统中，当有多个网络连接同时进行访问时，根据操作系统网络模型：

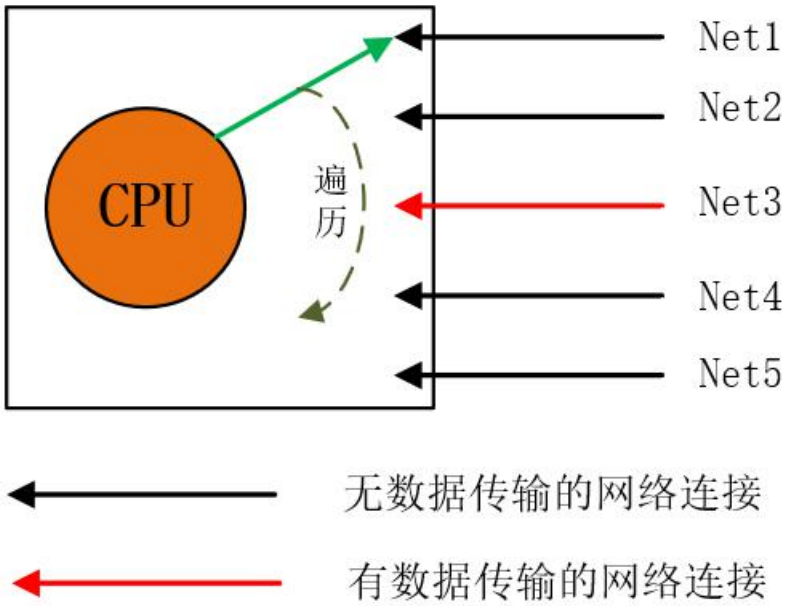
1. BIO/同步阻塞 IO/Blocking IO

BIO 对于多个同时访问的网络链接有两种处理方式：多线程异步处理和单线程同步阻塞等待处理，但是他们有如下缺点：

1) 多线程异步处理：

多线程异步处理需要 CPU 在多个线程间来回切换，以确定各个线程是否有读写操作，并处理这些需要读写的数据；

当网络连接数量特别多，但是传输数据的线程又很少时，就会造成 CPU 性能的浪费，使系统效率大大降低。

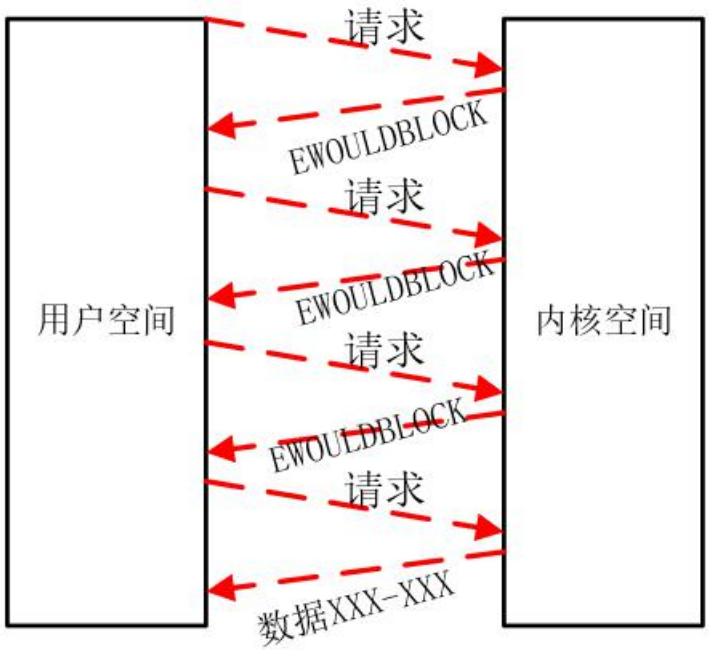


2) 单线程同步等待处理

单线程同步等待处理会造成等待时间过长，处理网络连接效率低下的问题；

2. NIO/非阻塞 IO/Non-Blocking I/O

1) NIO 则需要用户空间不停的去询问内核空间数据是否准备好，当线程过多，需要准备的数据量过大，就会造成线程的询问次数过多，从而浪费 CPU 性能，使 CPU 的有效利用率大大下降



3. 所以就需要一种高效的处理网络连接的方式，即 IO 复用

二、基础知识

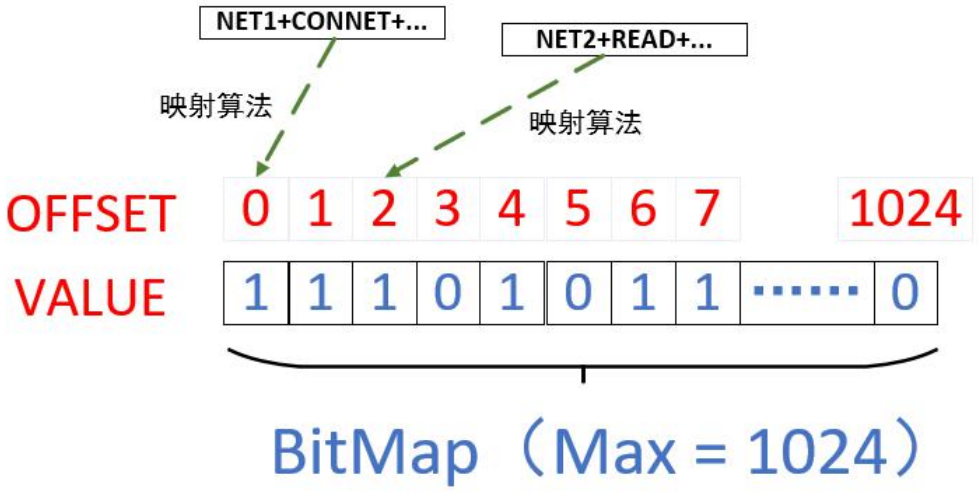
1. FD （file descriptor）

文件描述符，Linux 系统中用于指向资源的索引号；Linux 系统，”一切皆文件”,一切资源都可以通过文件的形式访问和管理。内核通过 FD 来访问和管理资源。

2. BitMap (位图)

具体可以查看 redis 笔记

位图是由 0 和 1 两个值组成的数组，在 IO 复用的 select 中，位图的最大长度为 1024，每一个位图的下标表示一个网络事件 FD，这些网络事件 FD 可以通过 hash 等算法，取到 1024 以内的数字进行映射



三、IO 复用 之 Select

```
/**
 * 获取就绪事件
 *
 * @param nfds       3个监听集合的文件描述符最大值+1
 * @param readfds    要监听的可读文件描述符集合
 * @param writefds   要监听的可写文件描述符集合
 * @param exceptfds 要监听的异常文件描述符集合
 * @param timeval    本次调用的超时时间
 * @return 大于0：已就绪的文件描述符数；等于0：超时；小于：出错
 */
int select(int nfds,
           fd_set *readfds,
           fd_set *writefds,
           fd_set *exceptfds,
           struct timeval *timeout);
```

1. 介绍

   Select 函数有以下四个参数：

   ① 要监听的网络可读事件 FD，文件描述符的集合，用位图来表示（最大长度 1024）

   ② 要监听的网络可写事件 FD，文件描述符的集合，用位图来表示（最大长度 1024）

   ③ 要监听的网络异常事件 FD，文件描述符的集合，用位图来表示（最大长度 1024）

   ④ 上述三个监听事件文件描述符的最大值+1，即三个位图的最大有效长度

   ⑤ 超时时间

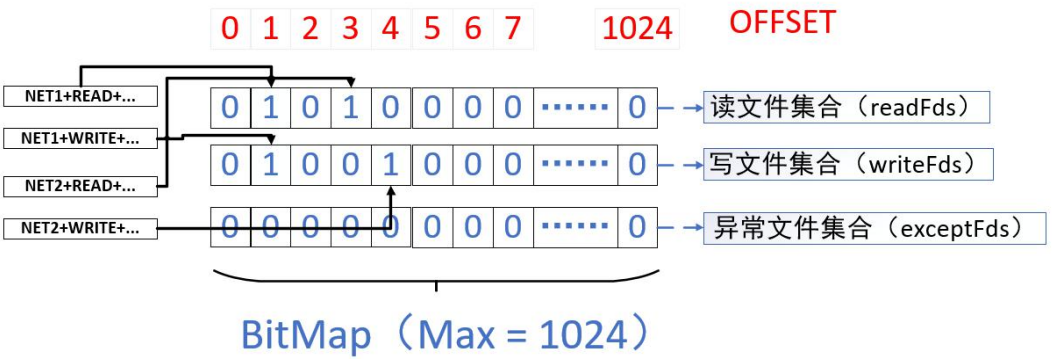
2. 常用的监测事件（了解）

监测事件	含义
OP_CONNECT	经过三次握手 变为可连接状态
OP_ACCEPT	在可连接的基础上　做好缓冲池等准备工作 变为数据传输就绪状态
OP_WRITE	可读状态
OP_READ	可写状态

3. Select IO 复用的过程介绍

   当 IO 复用模型为 select 时，网络连接会发生如下过程：

(1) 系统会获取到要监测的网络事件 FD 的描述符，根据其类型，放置到对应的位图中；将 FD 描述符的值对应位图下标，置位 1，表示监测该事件（位图最大长度为 1024）

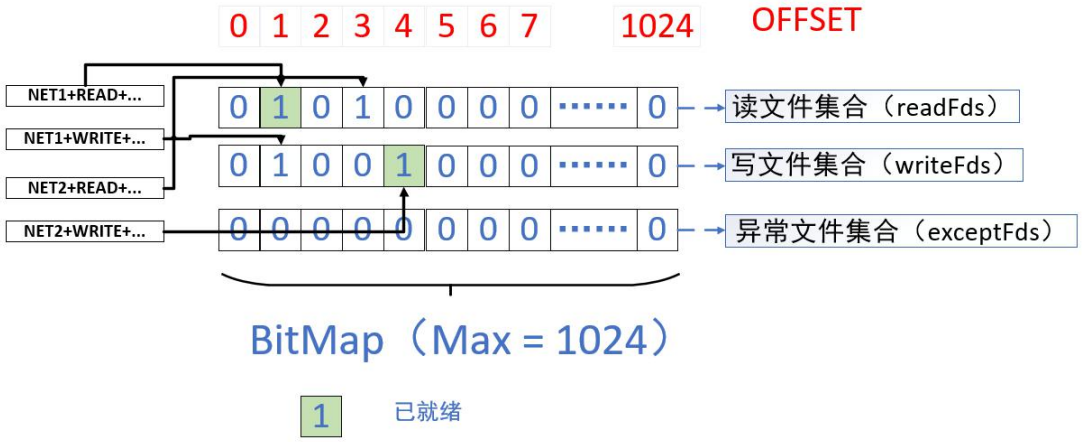


(2) 然后会将这 3 个位图,连同其他两个参数，传入到内核空间，位图采用拷贝的方式

(3) 在将这 3 个需要监测的网络事件的位图拷贝至内核空间后，CPU 会进行一次遍历，观察哪个网络事件已经就绪；

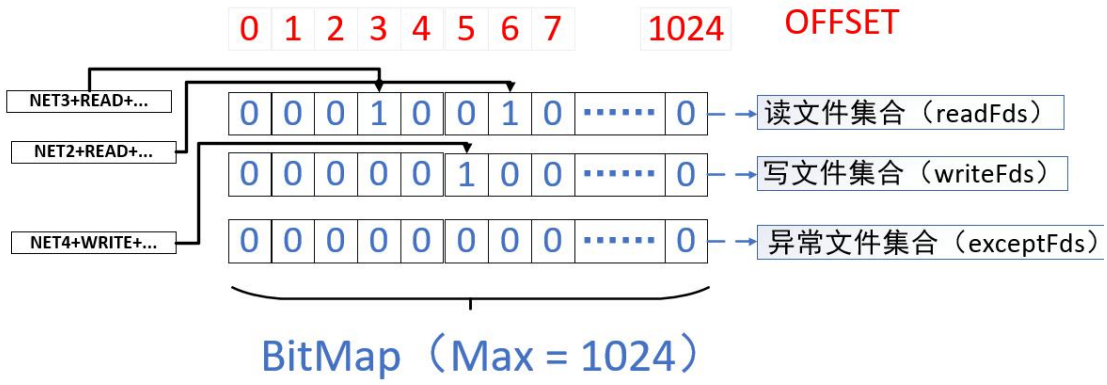
(4) 如果在遍历时，发现了已经就绪的网络事件

① 系统会在已经就绪的网络事件的位图上做已就绪的标记，然后返回给用户空间已经就绪的网络事件数量；



② 然后用户空间会得到已就绪的网络事件数量，并且因为位图在内核空间已经被标记（传的是指针，内核空间改变了位图，用户空间的位图也会改变），用户空间就会拿着已就绪的网络事件数量，以及被标记的已就绪的网络事件位图，遍历出已就绪的网络事件，并对已就绪的网络事件进行读写操作；

③ 如果还要进行下一次监测事件，就需要重置三个位图，然后再次送至内核空间进行监测



(5) 如果在遍历时，未发现已经就绪的网络事件

① 进程将会阻塞起来；

② 当有网络数据到达网卡后，系统会使用 DMA 技术，将网卡的数据拷贝至内存中；

③ 拷贝完后，会发送中断信号给 CPU，通知 CPU 有信息网络数据到达；

④ CPU 接收到中断信号后进行响应中断，会将刚才拷贝至内存的数据包进行解析，根据数据包中的 IP、端口等 socket 信息，将该数据包拷贝至对应的 socket 接收对列中；

⑤ 在拷贝至 socket 接收对列后，CPU 会检测该 socket 的等待对列是否有进程正在阻塞等待；

⑥ 如果有阻塞等待，则会唤醒该进程，然后重新遍历一遍该位图，重新走第（3）步（即标记就绪，返回等）；

(6) 总结：

优点：

① 可以批量的进行网络事件的监测，且更加高效的利用 CPU；

缺点：

① 有监测数量限制，每个位图最大长度 1024；
② 每次调用都需要将 FD 集合从用户态拷贝至内核态
③ 函数返回的是网络事件就绪的数量，需要遍历三个位图才能确定哪些网络事件就绪
④ 入参的 3 个位图每次调用都需要重置

四、IO 复用之 poll

poll

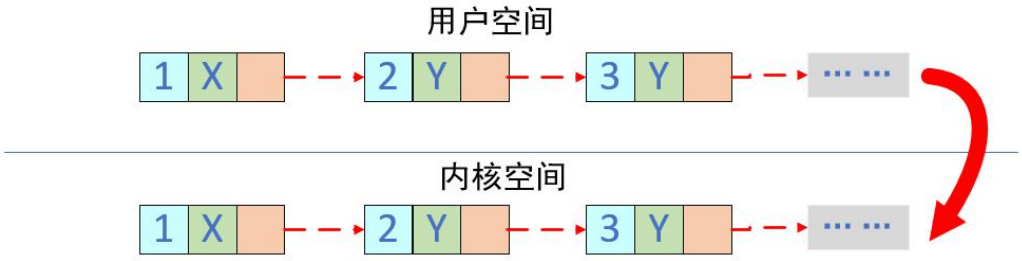
```
/**
 * 获取就绪事件
 *
 * @param pollfd  要监听的文件描述符集合
 * @param nfds    文件描述符数量
 * @param timeout 本次调用的超时时间
 * @return 大于0：已就绪的文件描述符数；等于0：超时；小于：出错
 */
int poll(struct pollfd *fds,
         unsigned int nfds,
         int timeout);
```

```
struct pollfd {
    int fd;          // 监听的文件描述符
    short events;    // 监听的事件
    short revents;   // 就绪的事件
}
```

1. 介绍

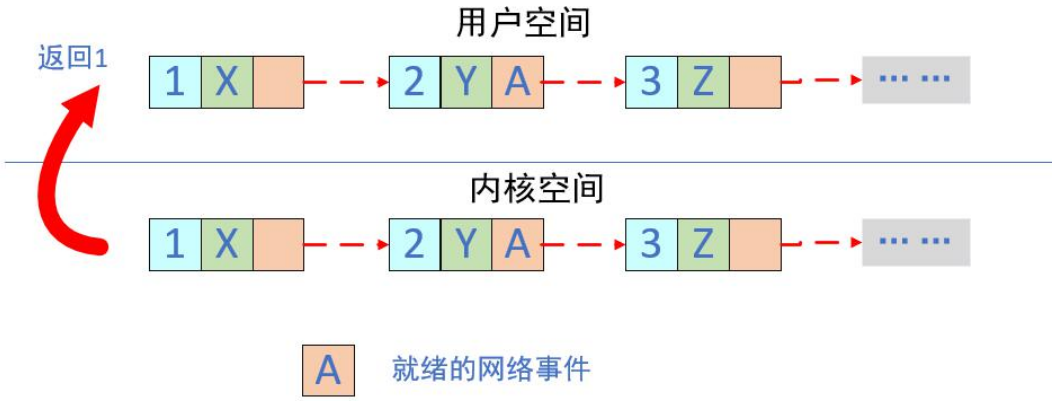
Poll 其实就是对 select 的改进，底层原理以及过程几乎和 select 一致，只是使用了链表的形式代替了 select 中的位图，使监测网络事件的数量突破了 1024 的限制，以及不用每次入参时都需要重置要监测的网络事件。

2. 参数介绍（了解）
   ① 监听的网络事件链表，无数量限制（链表中有监听的事件和就绪的事件）
   ② 监测的网络事件数量
   ③ 超时时间
3. 过程
   (1) 用户空间会将要监听的网络事件以及文件描述符放入到链表中
   (2) 然后会将这个链表,连同其他两个参数，传入到内核空间



(3) 在将需要监测的网络事件的链表拷贝至内核空间后，CPU 会进行一次遍历，观察哪个网络事件已经就绪；
(4) 如果在遍历时，发现了已经就绪的网络事件

① 系统会在已经就绪的网络事件的链表节点上填写已就绪的事件，然后返回给用户空间已经就绪的网络事件数量；



② 然后用户空间会得到已就绪的网络事件数量，并且因为链表在内核空间已经被填写就绪事件（传的是指针，内核空间改变了位图，用户空间的位图也会改变），用户空间就会拿着已就绪的网络事件数量，以及链表，遍历出已就绪的网络事件，并对已就绪的网络事件进行读写操作；

③ 如果还要进行下一次监测事件，直接将链表送至内核空间进行监测；

(5) 如果在遍历时，未发现已经就绪的网络事件

① 进程将会阻塞起来；

② 当有网络数据到达网卡后，系统会使用 DMA 技术，将网卡的数据拷贝至内存中；

③ 拷贝完后，会发送中断信号给 CPU，通知 CPU 有信息网络数据到达；

④ CPU 接收到中断信号后进行响应中断，会将刚才拷贝至内存的数据包进行解析，根据数据包中的 IP、端口等 socket 信息，将该数据包拷贝至对应的 socket 接收对列中；

⑤ 在拷贝至 socket 接收对列后，CPU 会检测该 socket 的等待对列是否有进程正在阻塞等待；

⑥ 如果有阻塞等待，则会唤醒该进程，然后重新遍历一遍该链表，重新走第（3）步；

五、IO 复用之 epoll

1. 基本知识介绍

(1) 回调函数

关于回调函数的介绍，见本人 CSDN：

https://blog.csdn.net/qq_23095607/article/details/138451911

(2) epoll 的相关源码及详细介绍

相关的底层源码以及详细介绍，见本人 CSDN：

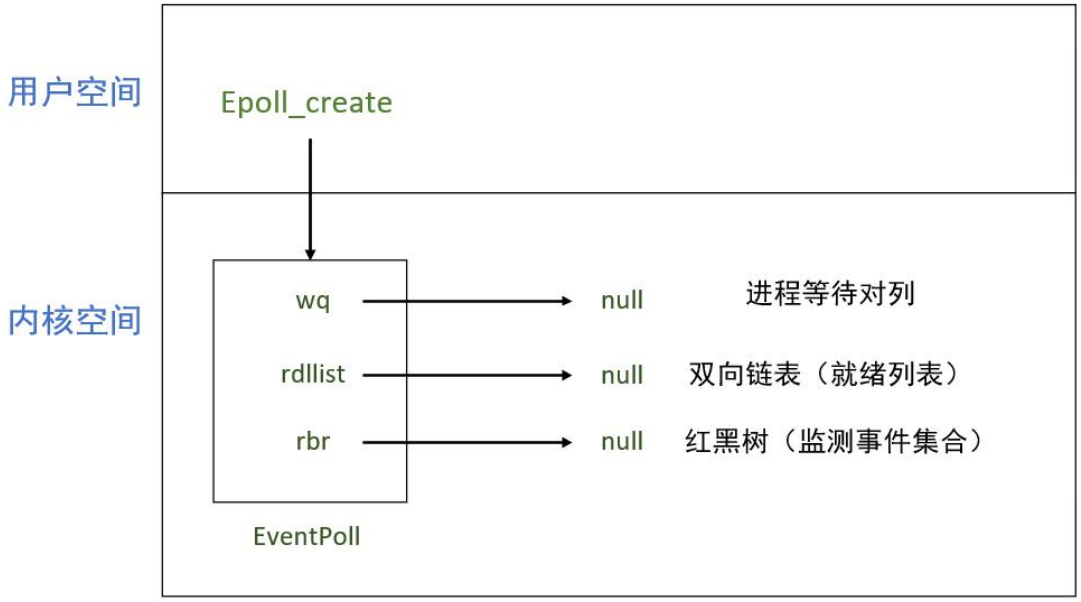
https://blog.csdn.net/qq_23095607/article/details/138571429

2. 相关原理介绍

Epoll 是仅适用于 Linux 内核，较为成熟的，可用于网络交互并发高的场景，是现阶段 CPU 利用率最高的网络模型；

当我们进行网络连接的时候，IO 复用模型-EPOLL 模型，会发生如下过程：

(1) 首先，内核空间会使用 epoll_create 创建一个 event_poll 对象，相当于一个容器，用来存储各个阶段产生的对象，具体结构如下：



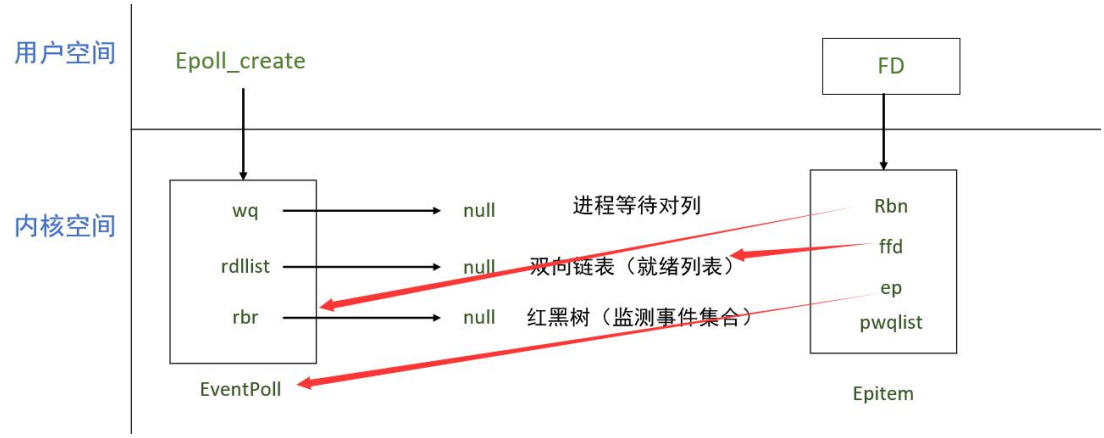
Eventpoll 包含以下三个对象：

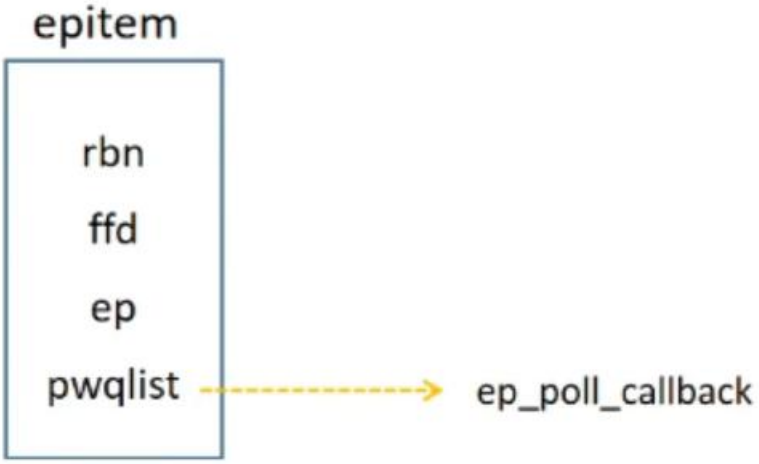
① 红黑树：用于保存管理要监测的网络事件；

② 就绪列表：结构为双向链表，用于保存已就绪的网络事件

③ 进程等待对列：用于管理未找到就绪网络事件，而被阻塞的进程

(2) 创建完存储容器后，就会执行 epoll_ctl，epoll_ctl 会将要监测的网络事件，从用户空间拷贝至内核空间，并在内核空间包装为 epitem 结构（结构如下）





其中：

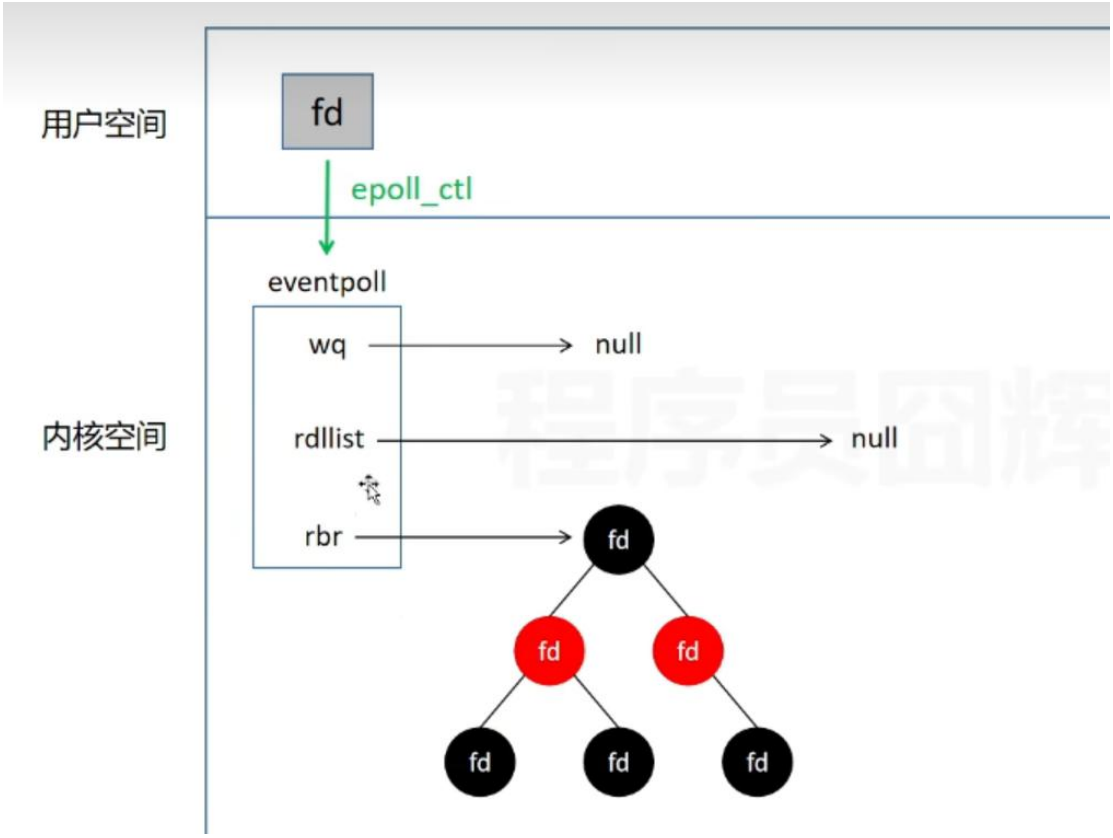
① Rbn: 表示一个红黑树节点

② FFD: 表示就绪列表，双向链表的一个节点

③ EP：event_poll 的 FD 描述符

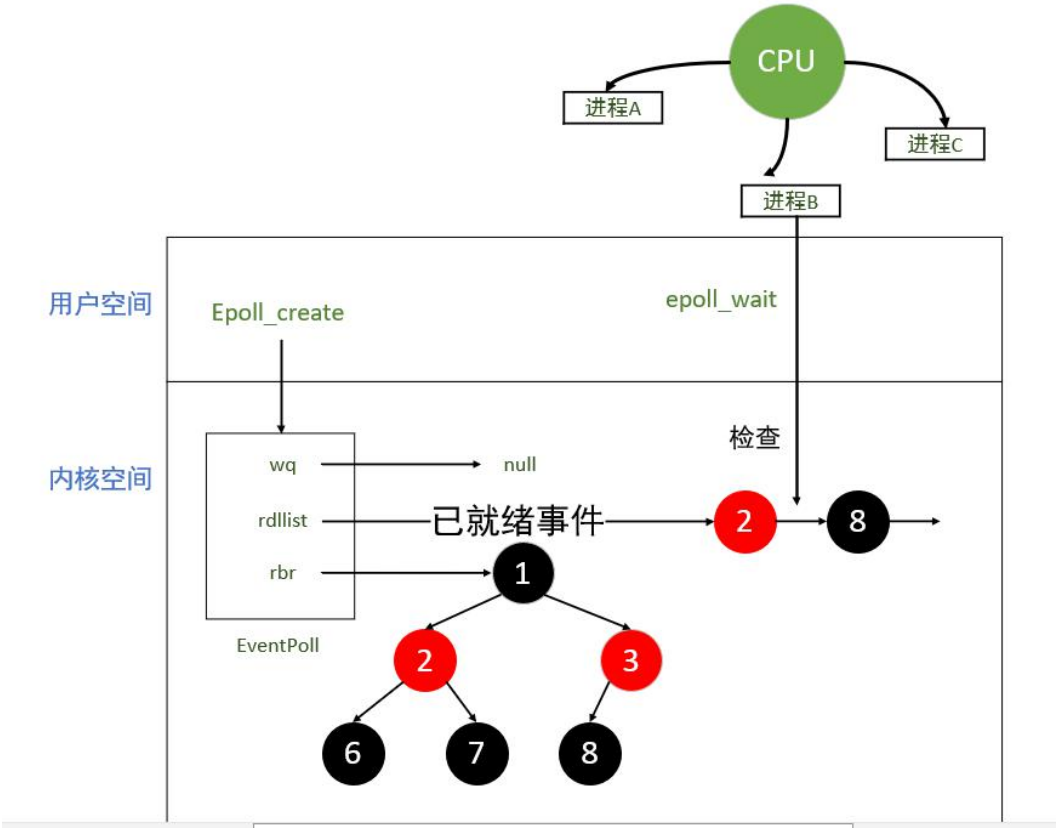
④ Pwqlist：Linux 系统有一个回调函数 ep_poll_callback，pwqlist 可以注册在该回调函数中，系统触发该回调函数，就会将红黑树上，该 FD 对应的 epitem 结构，放置于就绪队列中；

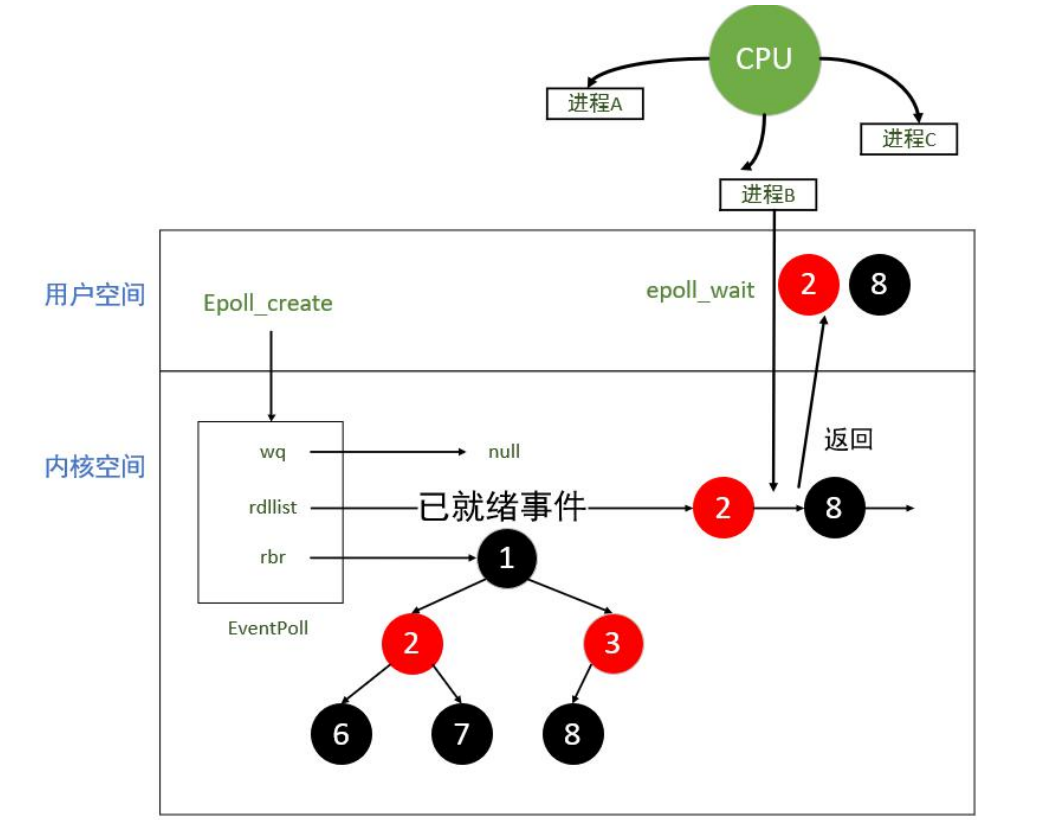
创建完 epitem 后，如果是增加监测网络事件，就会将该 epitem 添加到红黑树上；如果是删除监测事件，就会将该 epitem 从红黑树上删掉；



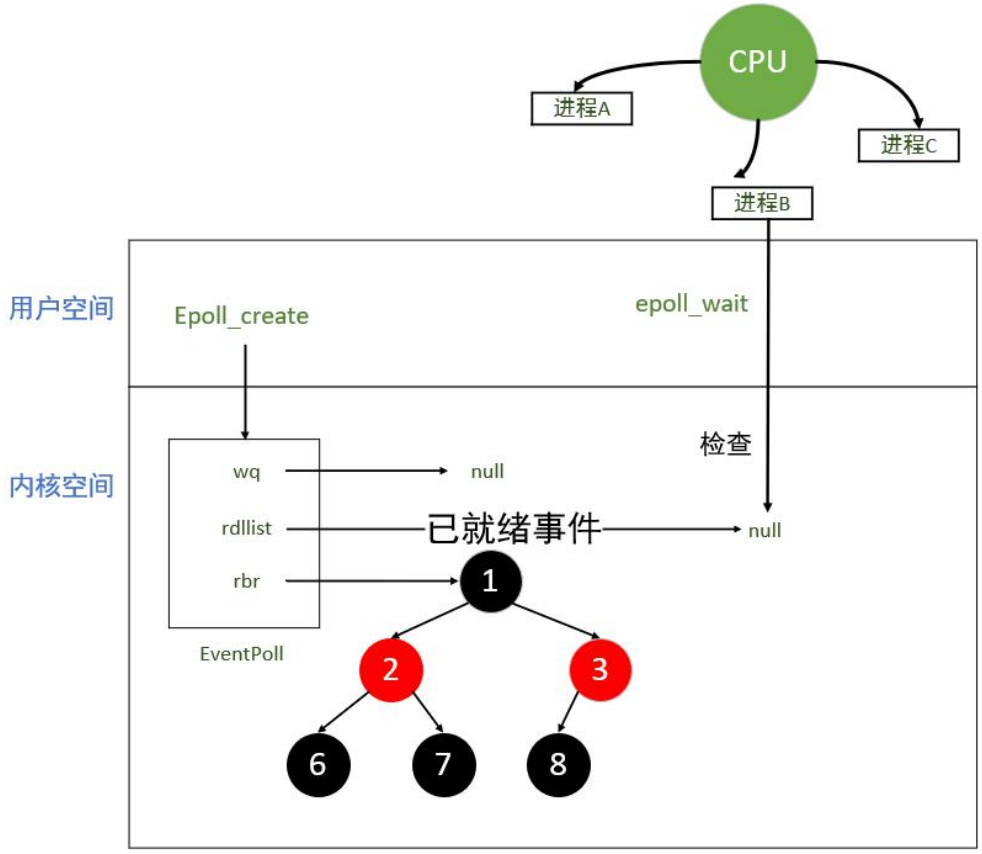
(3) 在经历了创建阶段，注册阶段，就到了检测阶段。系统会执行 epoll_wait 函数，来进行检测；

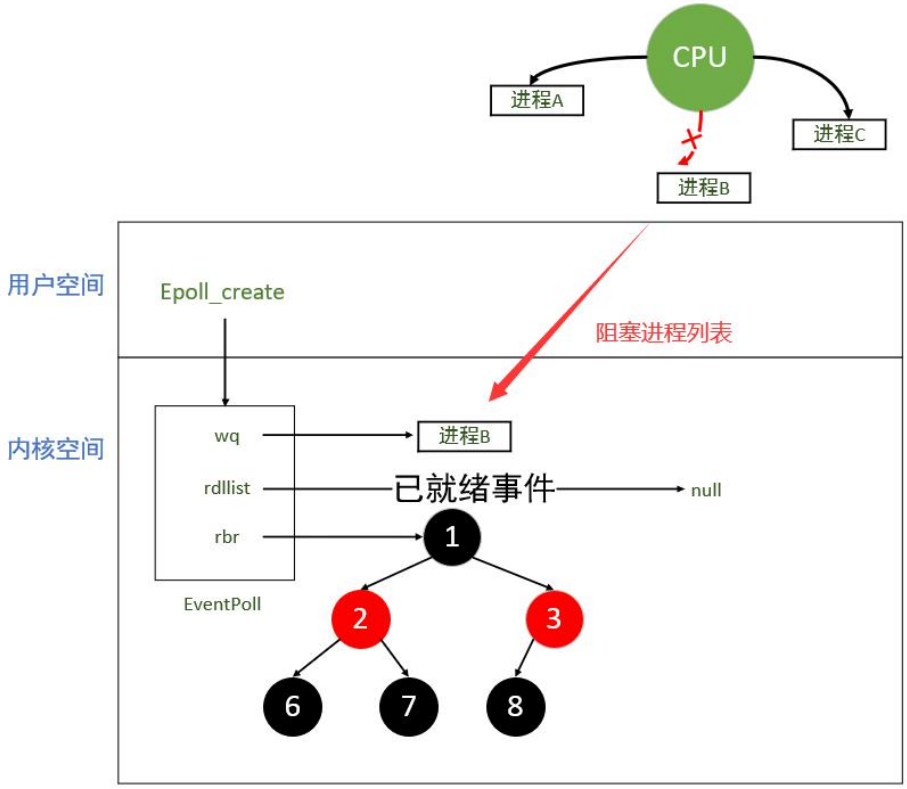
① epoll_wait 函数会先检测就绪队列是否有数据，如果有数据，epoll_wait 函数就会将就绪队列中的 FD 进行返回，然后用户空间就会得到已就绪的网络事件，并且读取已就绪网络事件的数据;



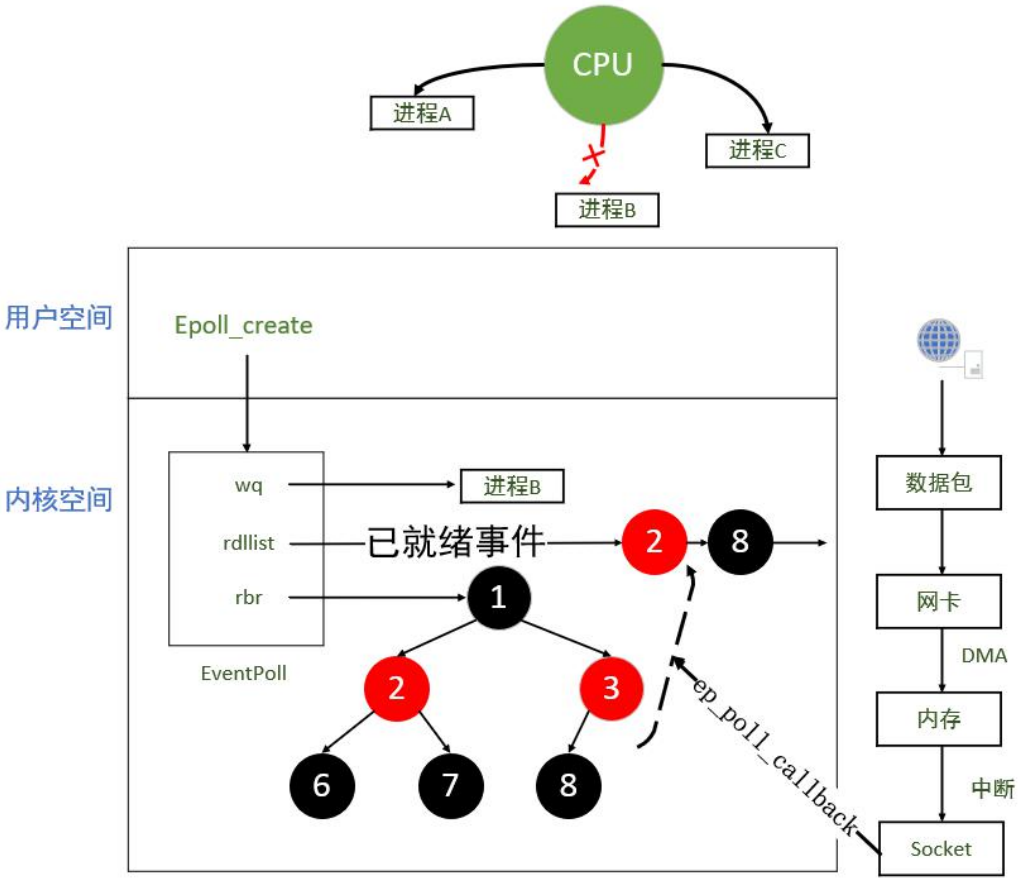


② 如果就绪队列中没有数据，系统就会将该 epoll_wait 进程进行阻塞，将该进程放置于阻塞队列中，等待发生如下过程：

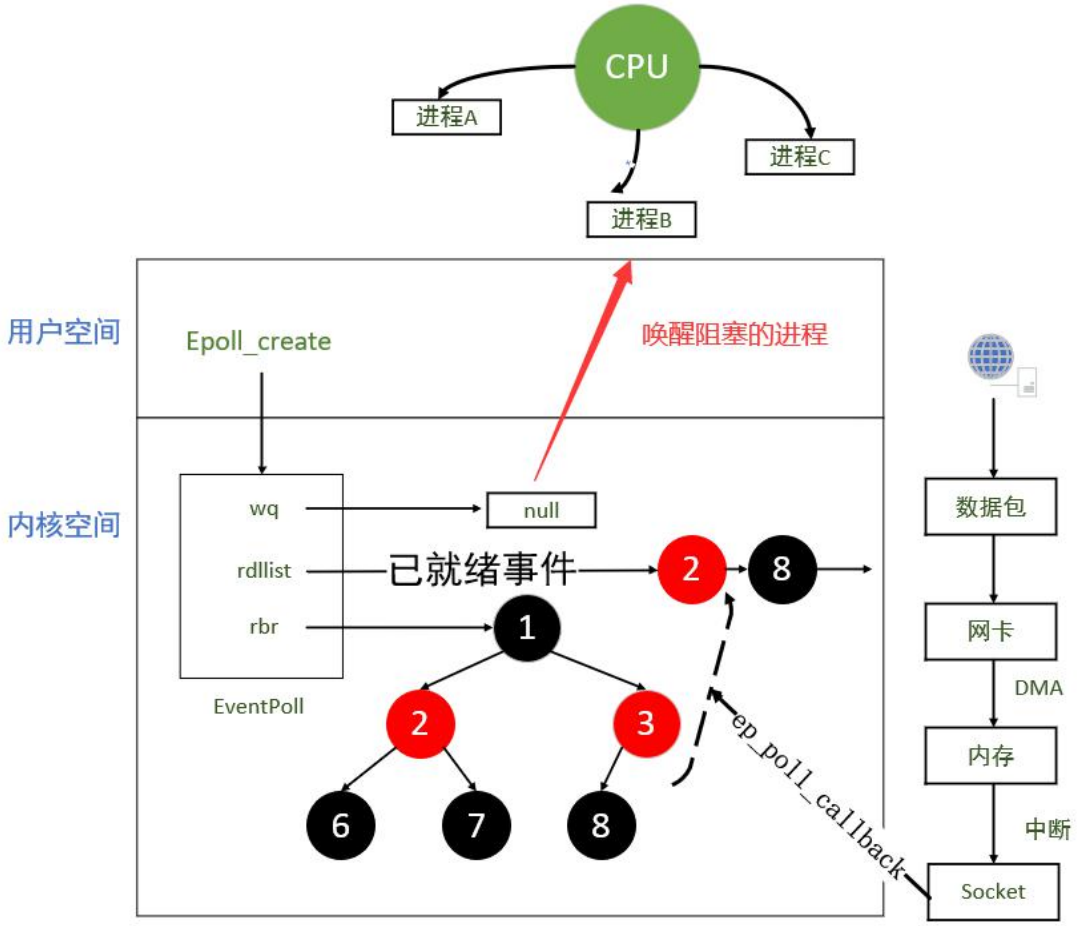




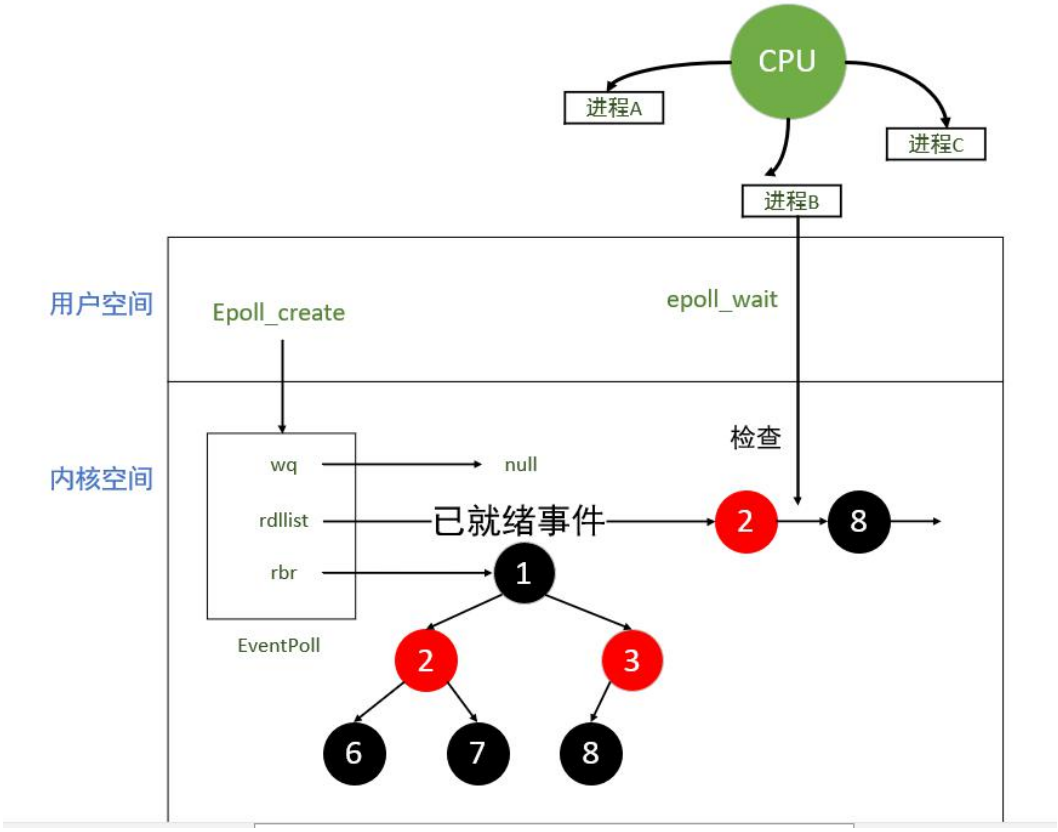
a. 当有网络数据发送过来的时候，系统会使用 DMA 技术，将网卡的数据拷贝至内存中；
b. 然后系统会给 CPU 发送中断信号，让 CPU 空出时间来处理这些数据；
c. CPU 会根据数据包中的 IP 和端口号，找到对应地 socket 队列，将数据包放到 socket 队列中
d. 并执行回调函数 ep_poll_callback，将 epoll_ctl 阶段注册在回调函数中的，并且在红黑树中的 epitem 节点，放置到就绪队列中；

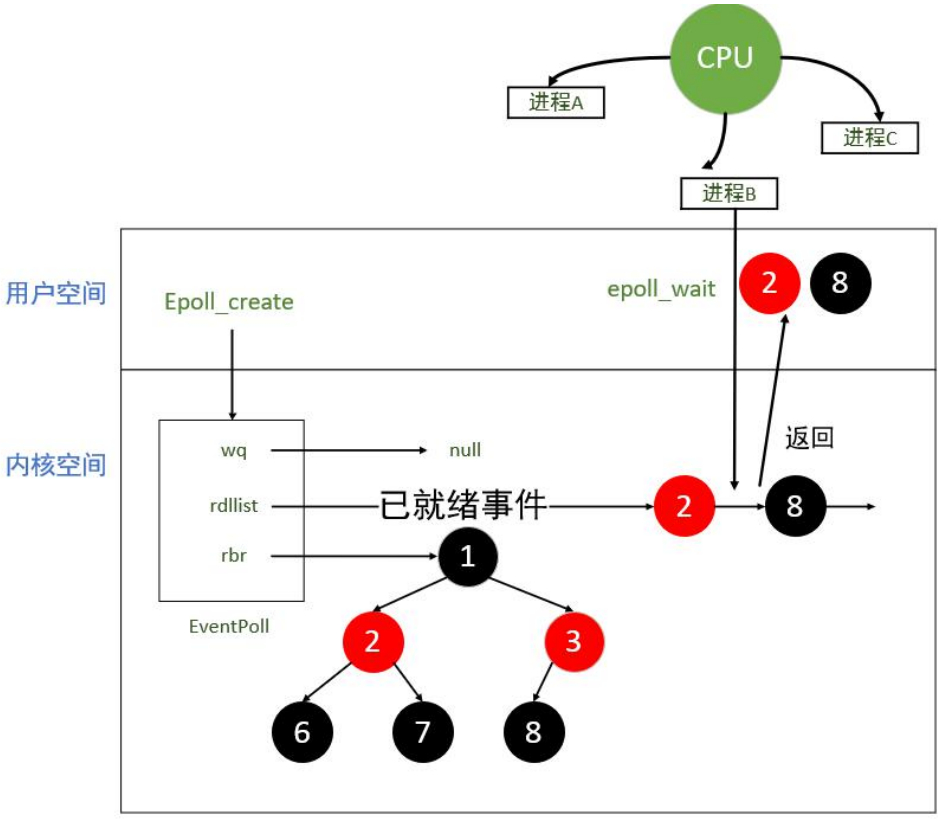


e. 然后唤醒阻塞队列中被阻塞的执行 epoll_wait 的进程，使之能够被 CPU 执行得到；



f. CPU 遍历到这个被唤醒的 epoll_wait 进程,执行 epoll_wait 函数，检测就绪队列是否有数据，如果有数据，epoll_wait 函数就会将就绪队列中的 FD 进行返回，然后用户空间就会得到已就绪的网络事件，并且读取已就绪网络事件的数据;





g. 需要注意的是 Epoll 中并没有使用 MMAP 技术

3. Epoll 总结

(1) 高效处理高并发下的大量连接，同时有非常优异的性能

(2) 优点：

a. 监听的网络事件数量，没有 1024 等的数量限制

b. 每次返回具体就绪的网络事件，不需要遍历

(3) 缺点：

A. 跨平台不够好，只支持 linux

B. Select,poll 轻量级，epoll 为重量级，移植性较差

C. 当监听网络事件比较少的情况下，select、poll 会更优

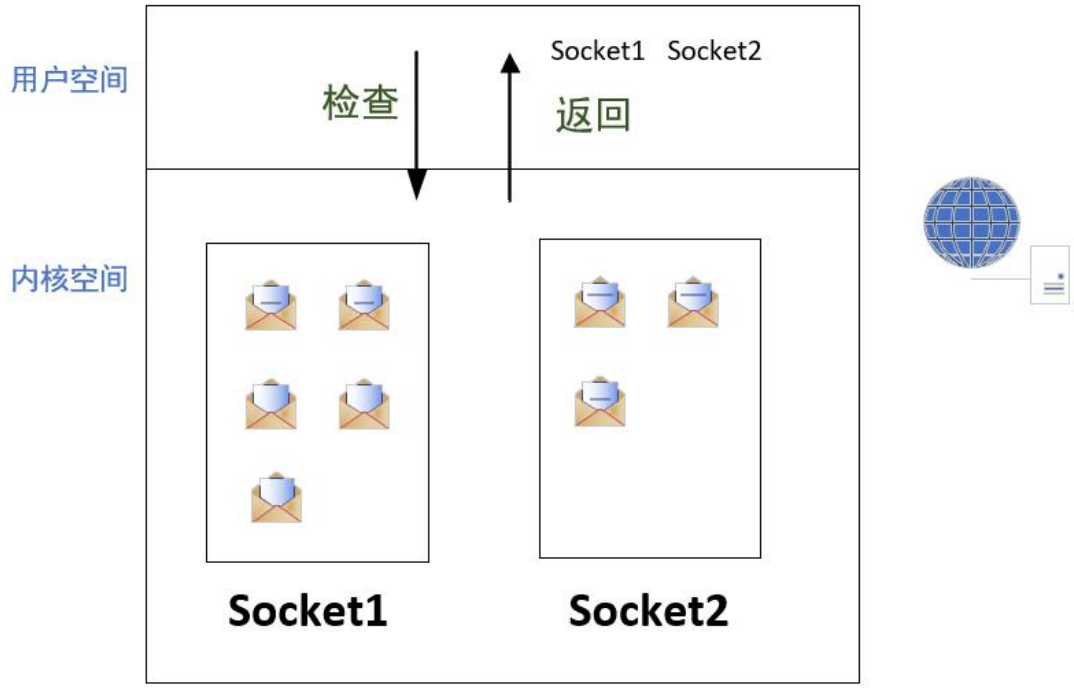
六、IO 复用的就绪事件触发机制

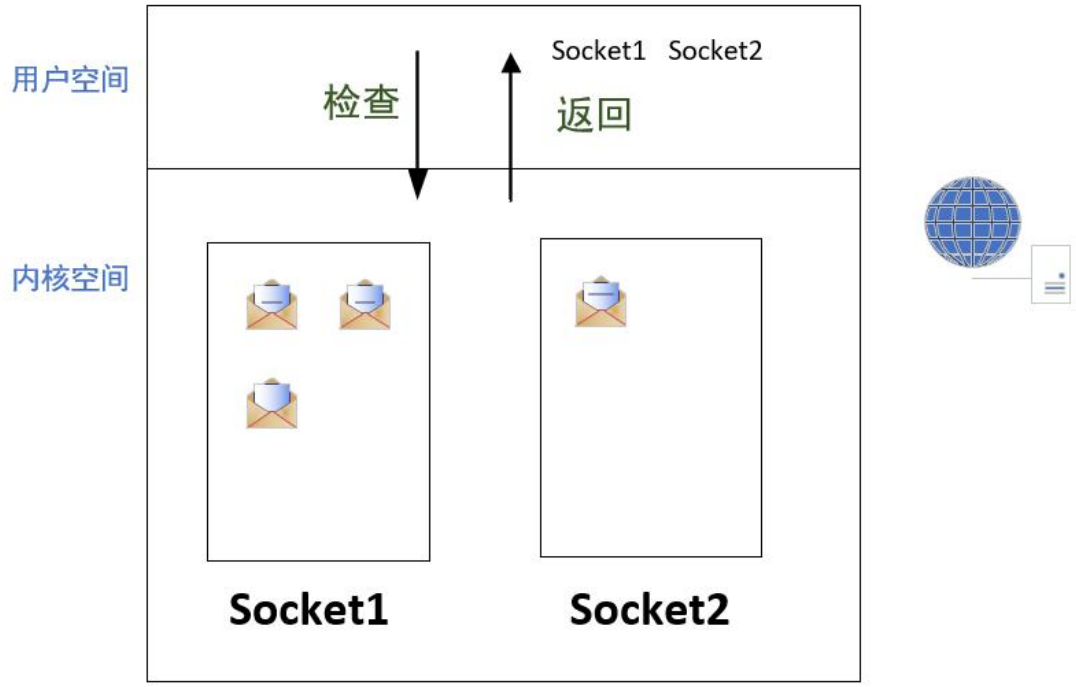
1. 简述

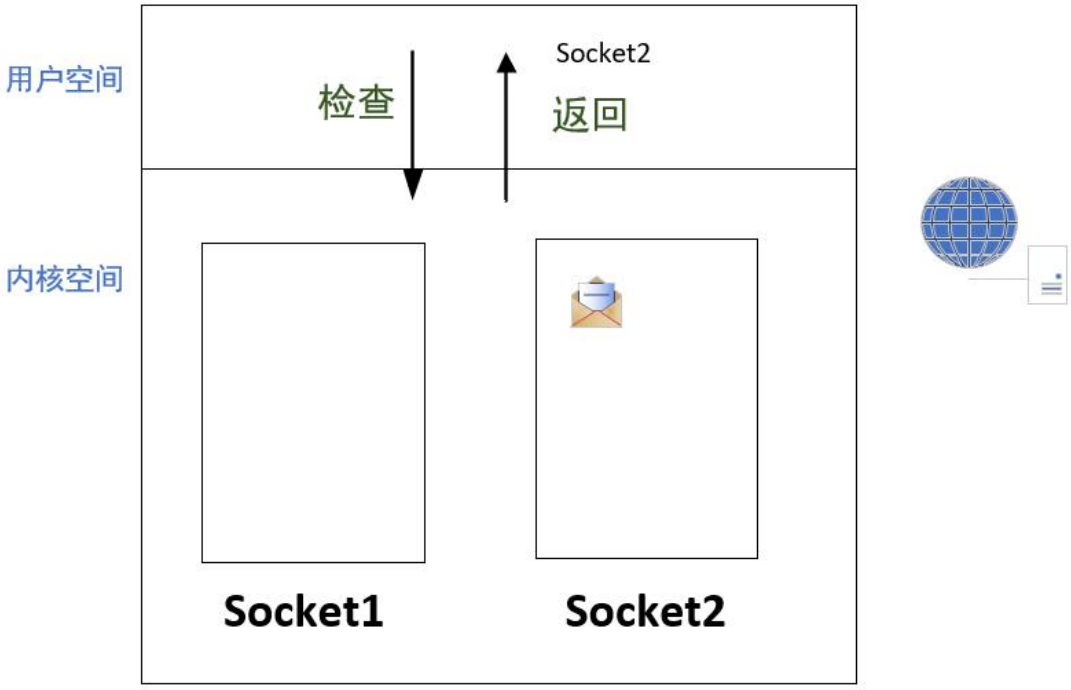
上述的三种 IO 多路复用，对于就绪事件的检测，操作系统分为两种方式:水平触发和边缘触发

2. 水平触发（三种 IO 复用的默认触发方式）

在内核空间，就绪的 socket 队列中的数据，只要未被用户空间获取完毕，用户空间每次进行已就绪队列检查时，都会返回这些有数据的队列；

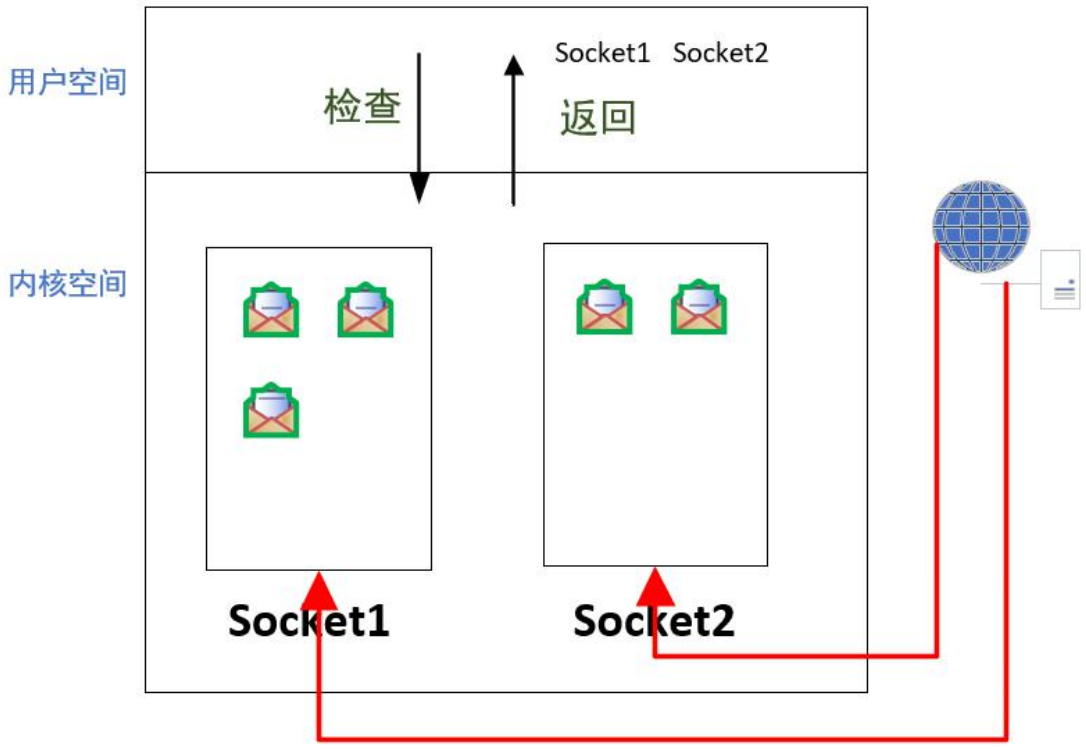


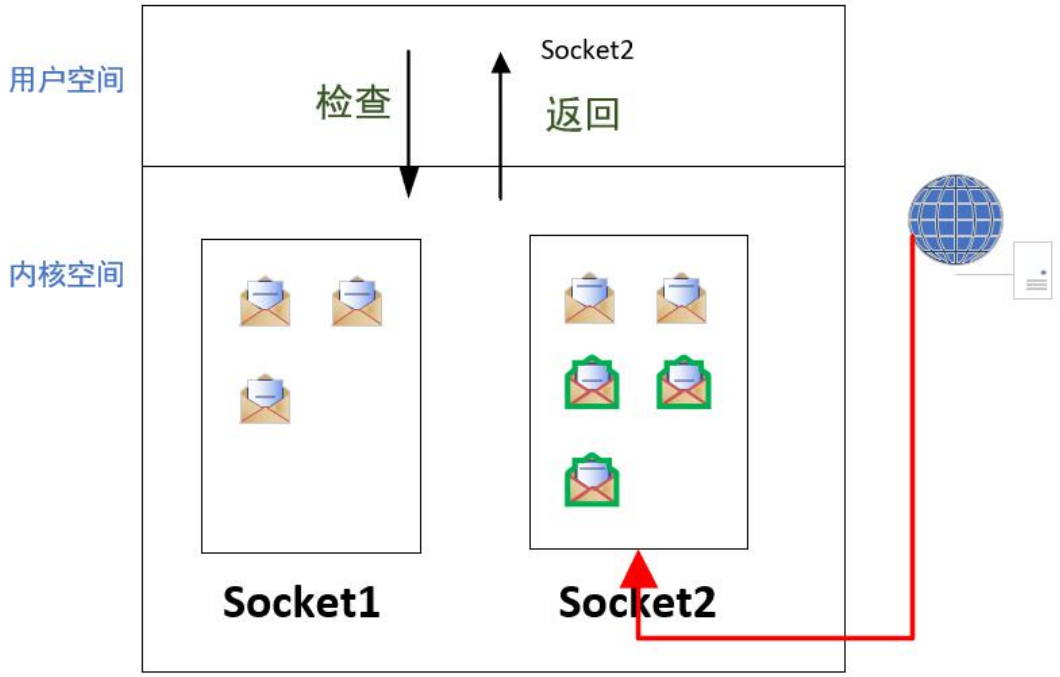




3. 边缘触发

在内核空间，监测的 socket 队列，只有在有新数据从网卡到达 socket 队列时，注意只有在有新数据到达时，用户空间进行就绪队列检查时，才会返回该就绪的 socket 队列；





七、IO 复用的总结

1. IO 复用其实就是利用网卡的 DMA 技术，以及 CPU 的中断信号，替代了 NIO 频繁的轮询遍历，使得 CPU 的每一个时间片做无用功的比例大大减少，提高了 CPU 的利用率
2. IO 复用的存在并不是为了提高单个网络 IO 的执行效率，而是为了使系统能够同时容纳更多的网络连接，同时提高 CPU 的利用率