### IO 复用

1. 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：

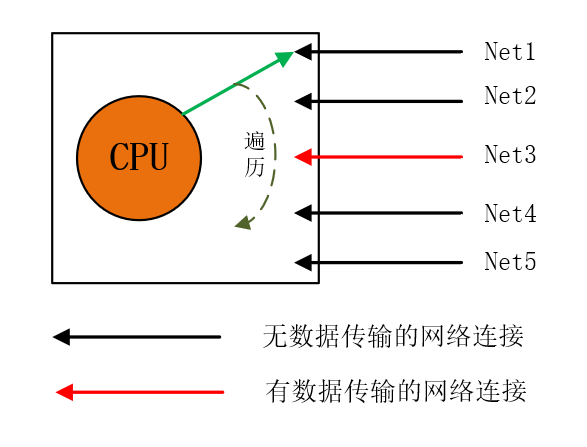
1. BIO/同步阻塞IO/Blocking IO

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；

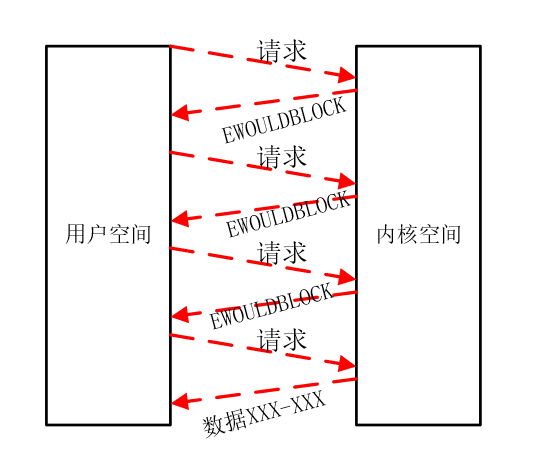
当网络连接数量特别多，但是传输数据的线程又很少时，就会造成CPU性能的浪费，使系统效率大大降低。



1. 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；

1. NIO/非阻塞IO/Non-Blocking I/O
2. 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



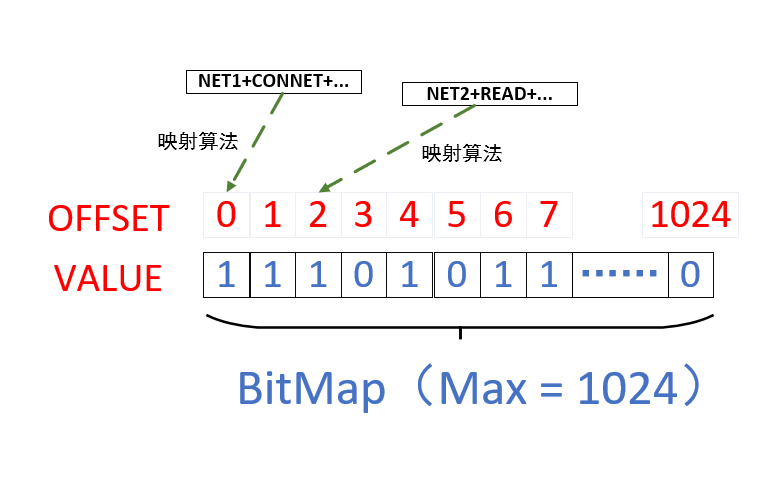
1. 所以就需要一种高效的处理网络连接的方式，即IO复用
2. 基础知识
3. FD （file descriptor）

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。

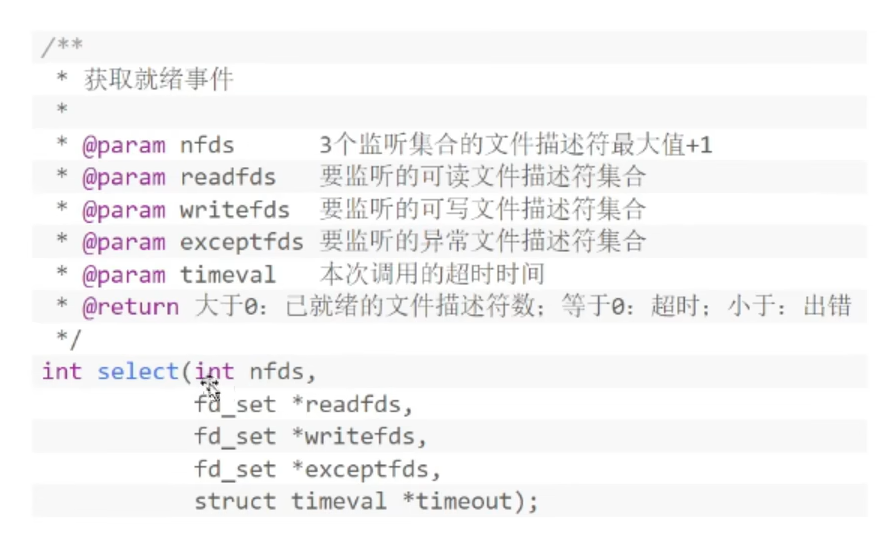
1. BitMap (位图)

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



1. IO复用 之 Select



1. 介绍

Select函数有以下四个参数：

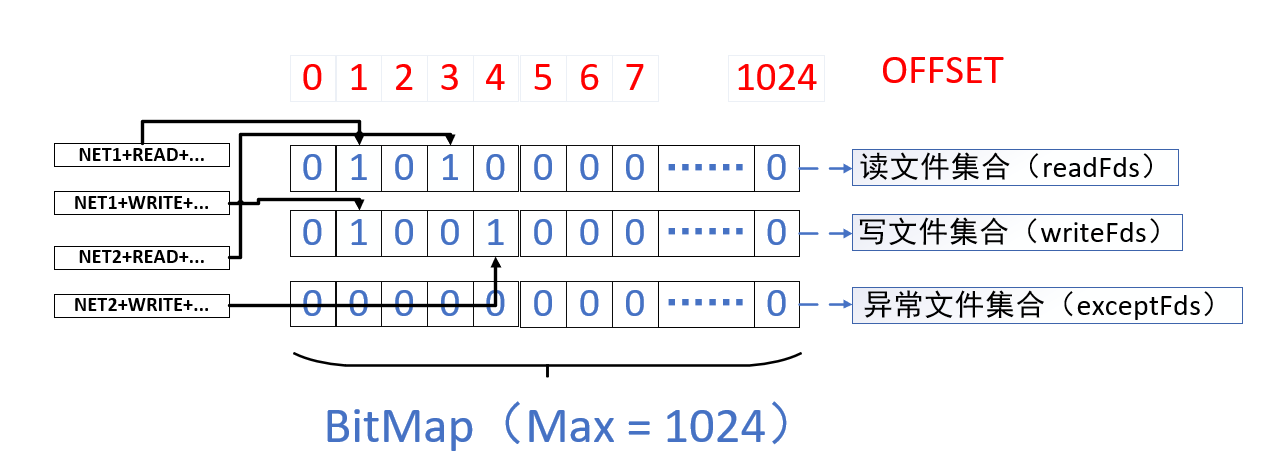
1. 要监听的网络可读事件FD，文件描述符的集合，用位图来表示（最大长度1024）
2. 要监听的网络可写事件FD，文件描述符的集合，用位图来表示（最大长度1024）
3. 要监听的网络异常事件FD，文件描述符的集合，用位图来表示（最大长度1024）
4. 上述三个监听事件文件描述符的最大值+1，即三个位图的最大有效长度
5. 超时时间
6. 常用的监测事件（了解）

|  |  |
| --- | --- |
| 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 |

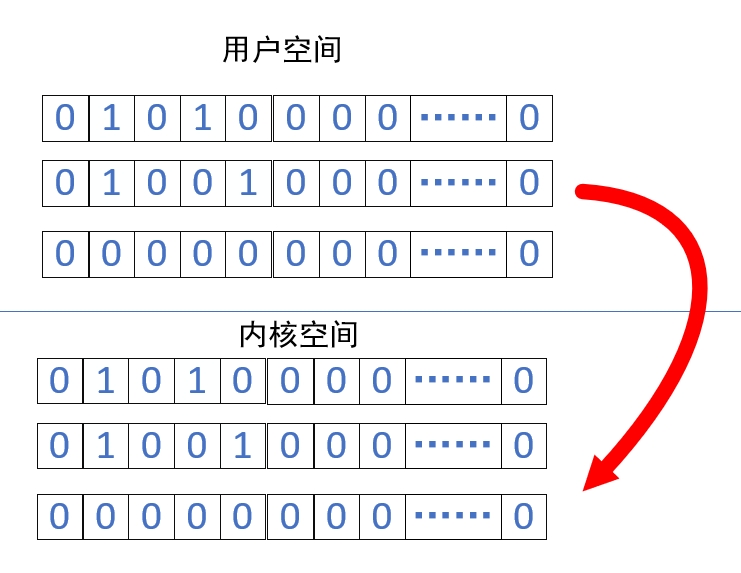
1. Select IO复用的过程介绍

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）



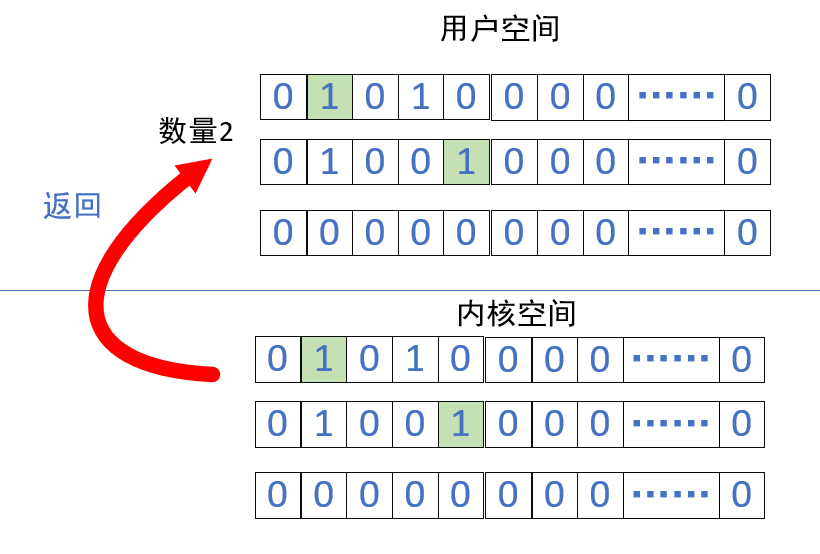
1. 然后会将这3个位图,连同其他两个参数，传入到内核空间，位图采用拷贝的方式



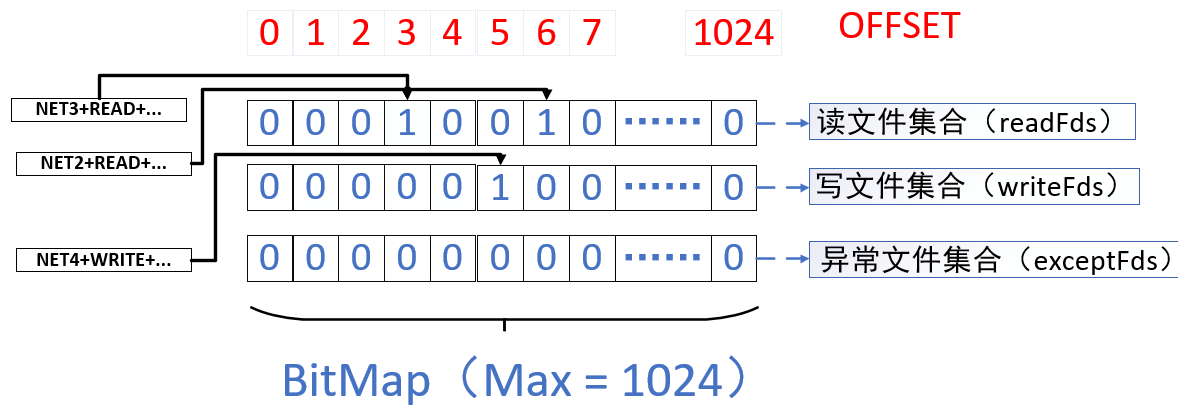
1. 在将这3个需要监测的网络事件的位图拷贝至内核空间后，CPU会进行一次遍历，观察哪个网络事件已经就绪；
2. 如果在遍历时，发现了已经就绪的网络事件
3. 系统会在已经就绪的网络事件的位图上做已就绪的标记，然后返回给用户空间已经就绪的网络事件数量；

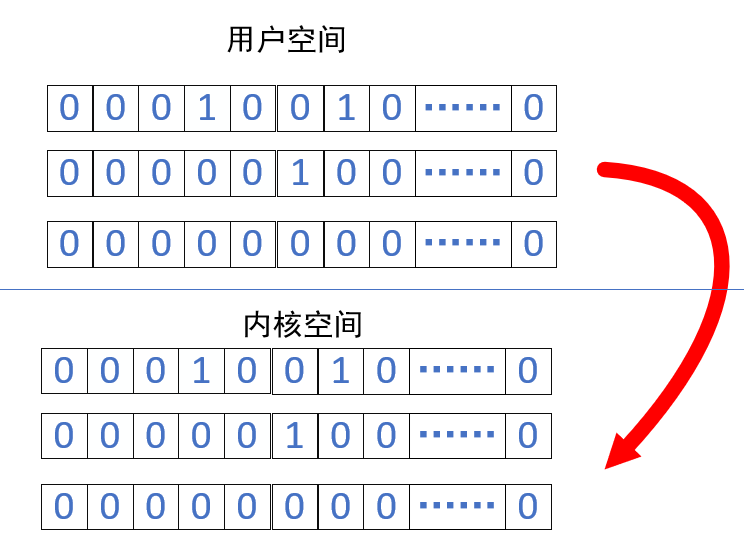


1. 然后用户空间会得到已就绪的网络事件数量，并且因为位图在内核空间已经被标记（传的是指针，内核空间改变了位图，用户空间的位图也会改变），用户空间就会拿着已就绪的网络事件数量，以及被标记的已就绪的网络事件位图，遍历出已就绪的网络事件，并对已就绪的网络事件进行读写操作；



1. 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





1. 如果在遍历时，未发现已经就绪的网络事件
2. 进程将会阻塞起来；
3. 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；
4. 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；
5. 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；
6. 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；
7. 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）；
8. 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：

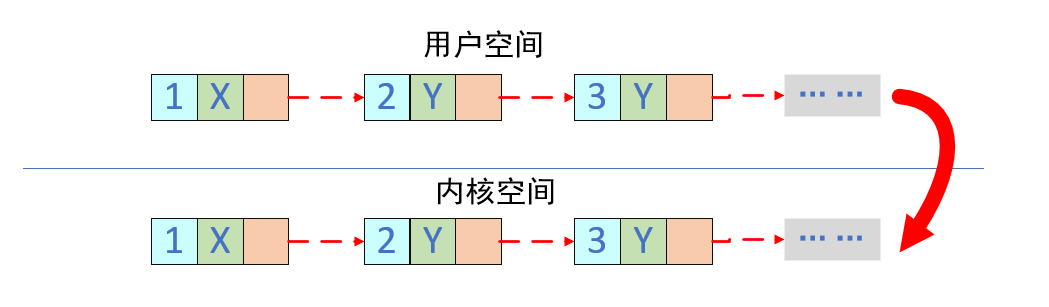
1. 有监测数量限制，每个位图最大长度1024；
2. 每次调用都需要将FD集合从用户态拷贝至内核态
3. 函数返回的是网络事件就绪的数量，需要遍历三个位图才能确定哪些网络事件就绪
4. 入参的3个位图每次调用都需要重置
5. IO 复用之poll



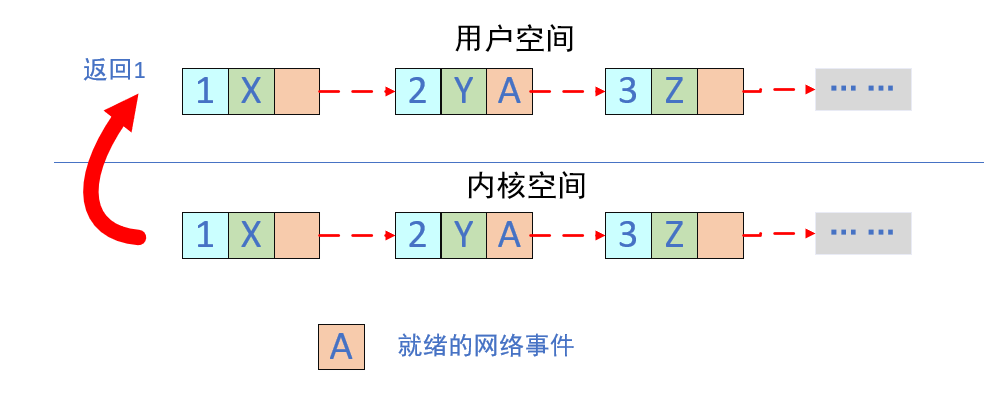
1. 介绍

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。

1. 参数介绍（了解）
2. 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）
3. 监测的网络事件数量
4. 超时时间
5. 过程
6. 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
7. 然后会将这个链表,连同其他两个参数，传入到内核空间



1. 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；
2. 如果在遍历时，发现了已经就绪的网络事件
3. 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；



1. 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；
2. 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；
3. 如果在遍历时，未发现已经就绪的网络事件
4. 进程将会阻塞起来；
5. 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；
6. 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；
7. 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；
8. 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；
9. 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；
10. IO复用之epoll
11. 基本知识介绍
12. 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

1. 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：

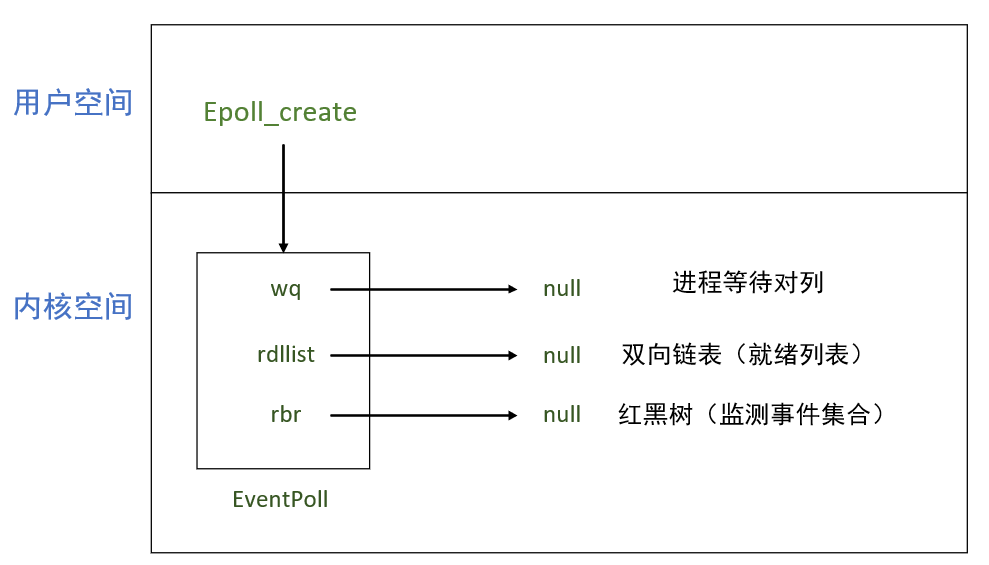
https://blog.csdn.net/qq\_23095607/article/details/138571429

1. 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；

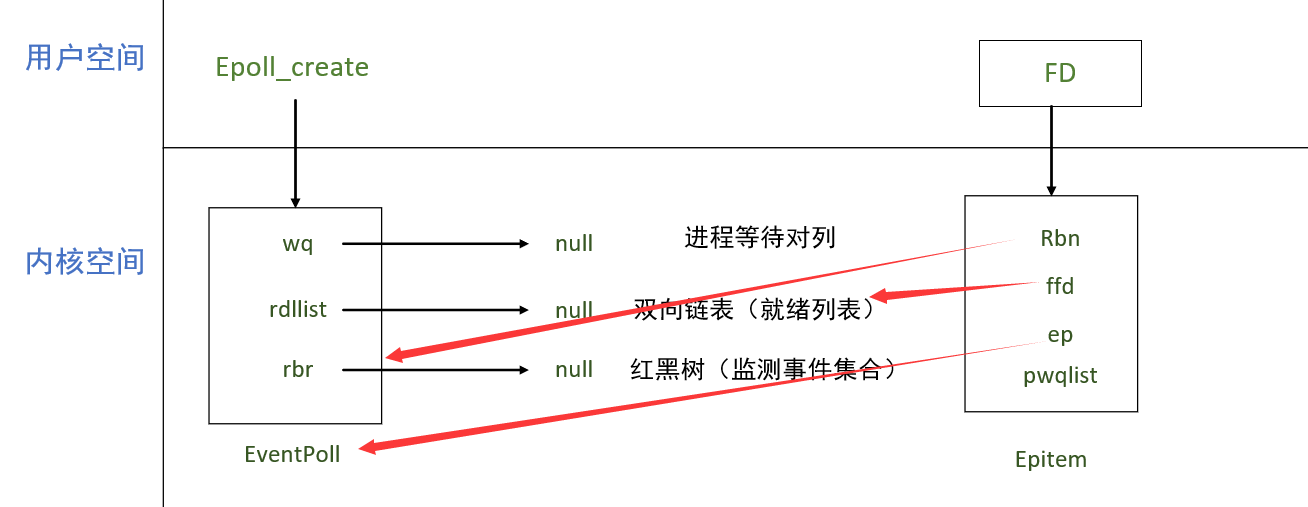
当我们进行网络连接的时候，IO复用模型-EPOLL模型，会发生如下过程：

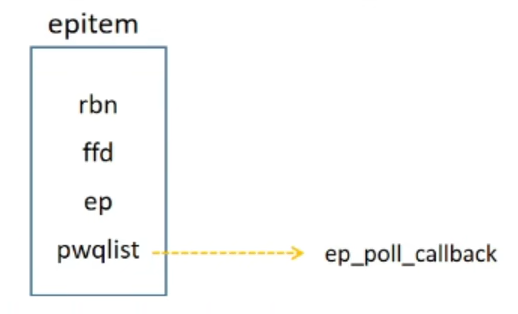
1. 首先，内核空间会使用epoll\_create创建一个event\_poll对象，相当于一个容器，用来存储各个阶段产生的对象，具体结构如下：



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
4. 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）

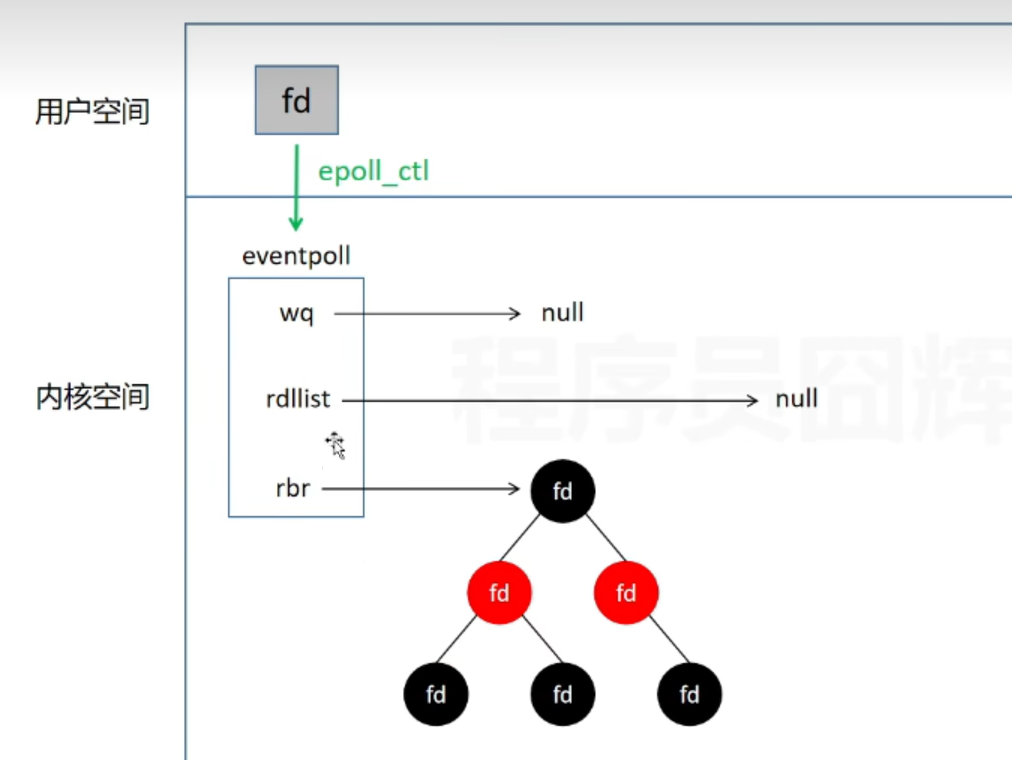




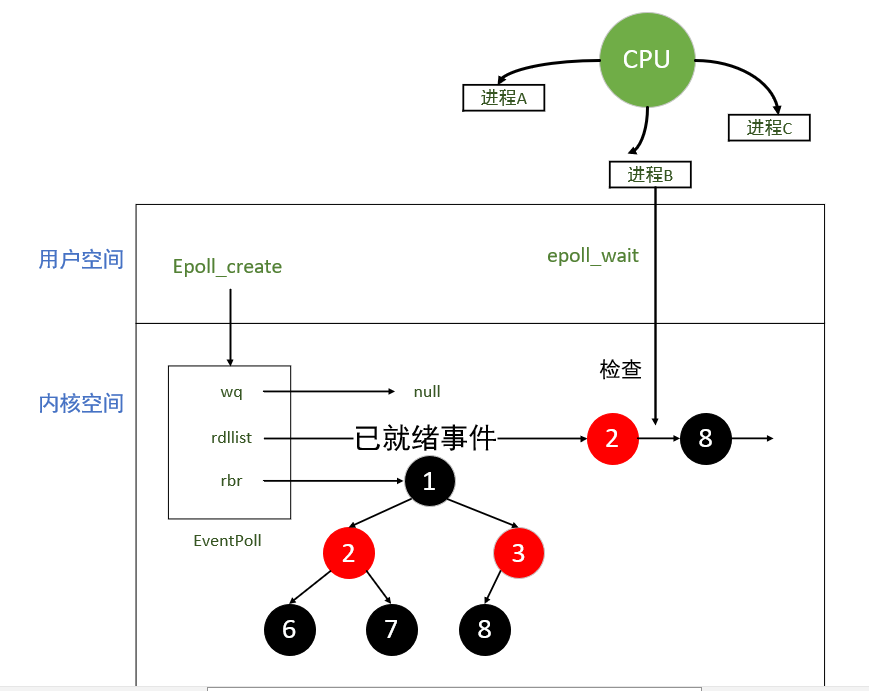
其中：

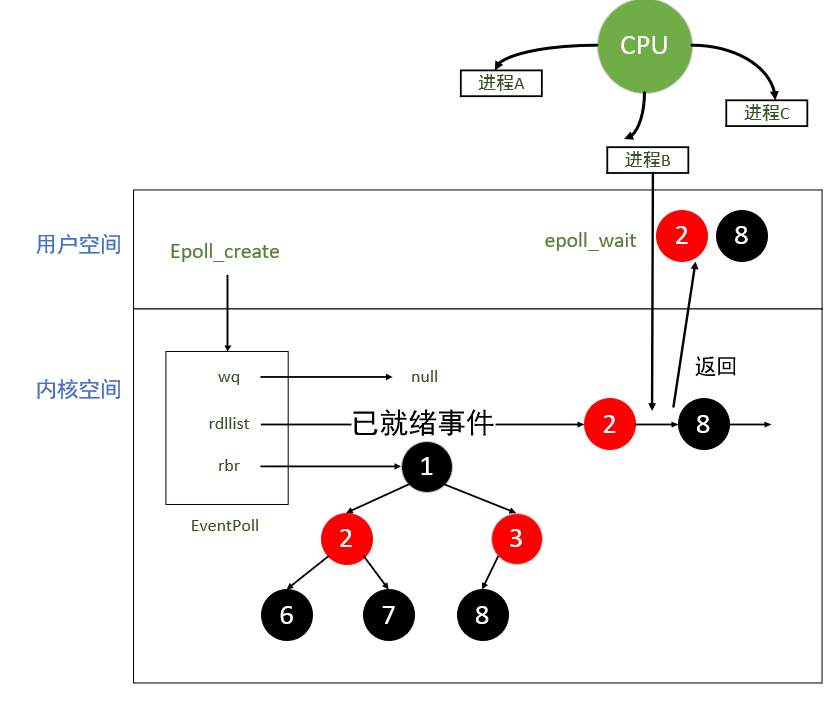
1. Rbn: 表示一个红黑树节点
2. FFD: 表示就绪列表，双向链表的一个节点
3. EP：event\_poll的FD描述符
4. Pwqlist：Linux系统有一个回调函数ep\_poll\_callback，pwqlist可以注册在该回调函数中，系统触发该回调函数，就会将红黑树上，该FD对应的epitem结构，放置于就绪队列中；

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；

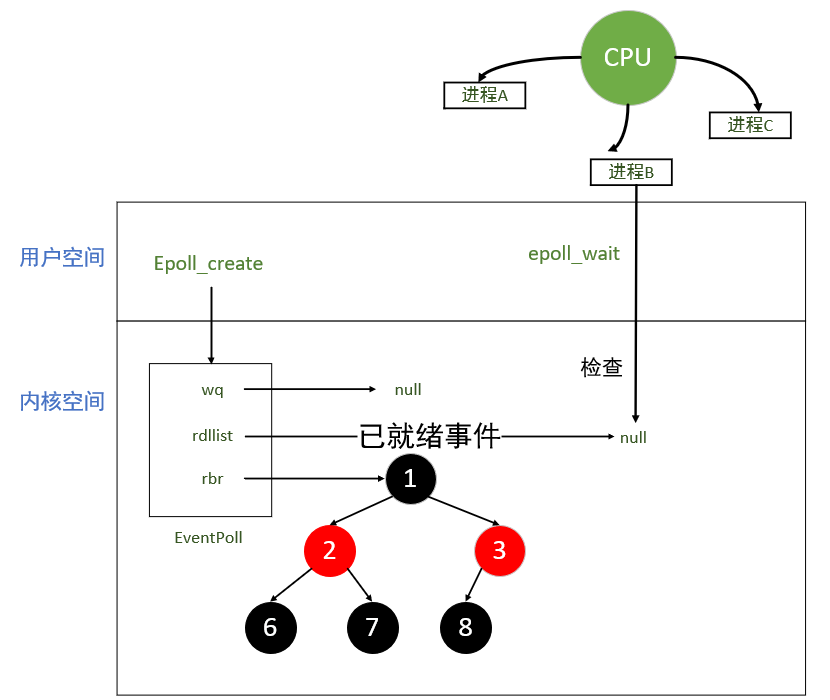


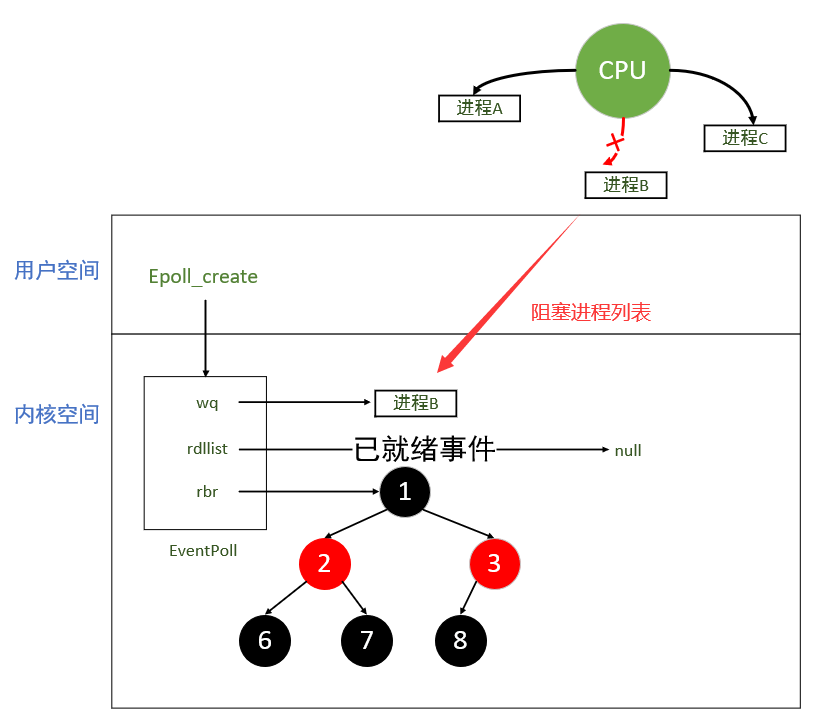
1. 在经历了创建阶段，注册阶段，就到了检测阶段。系统会执行epoll\_wait函数，来进行检测；
2. epoll\_wait函数会先检测就绪队列是否有数据，如果有数据，epoll\_wait函数就会将就绪队列中的FD进行返回，然后用户空间就会得到已就绪的网络事件，并且读取已就绪网络事件的数据;



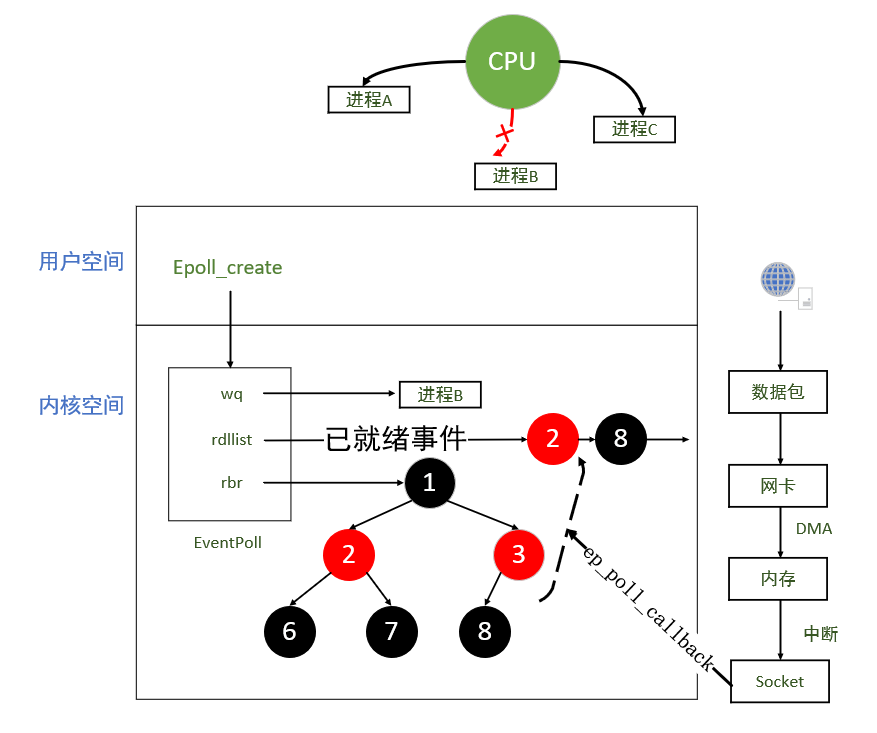


1. 如果就绪队列中没有数据，系统就会将该epoll\_wait进程进行阻塞，将该进程放置于阻塞队列中，等待发生如下过程：

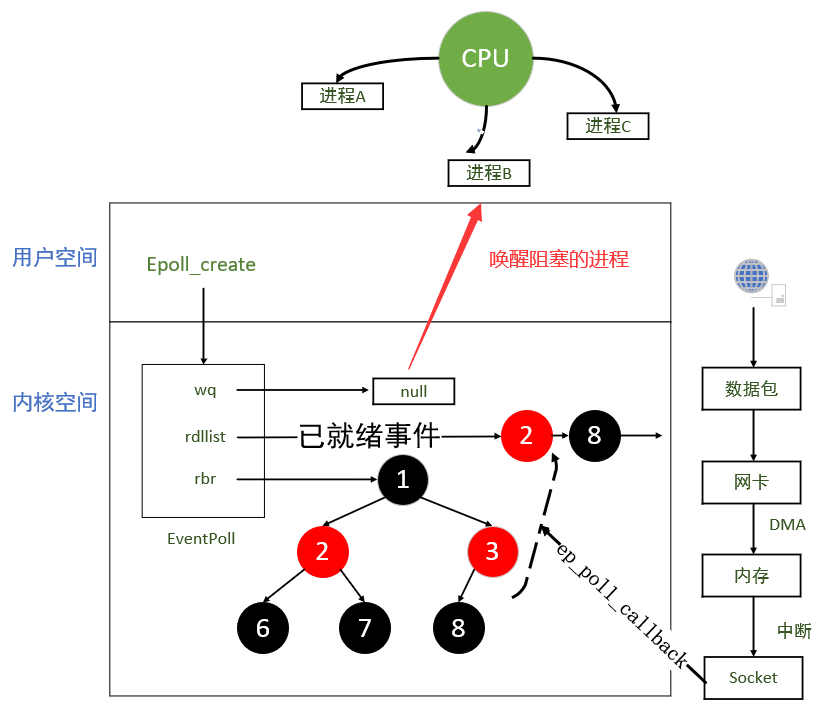




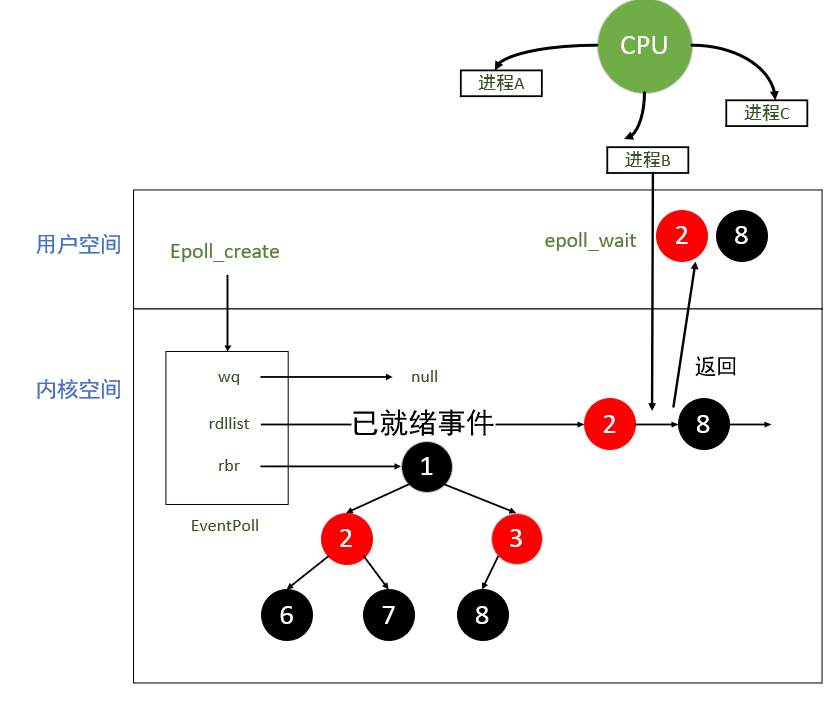
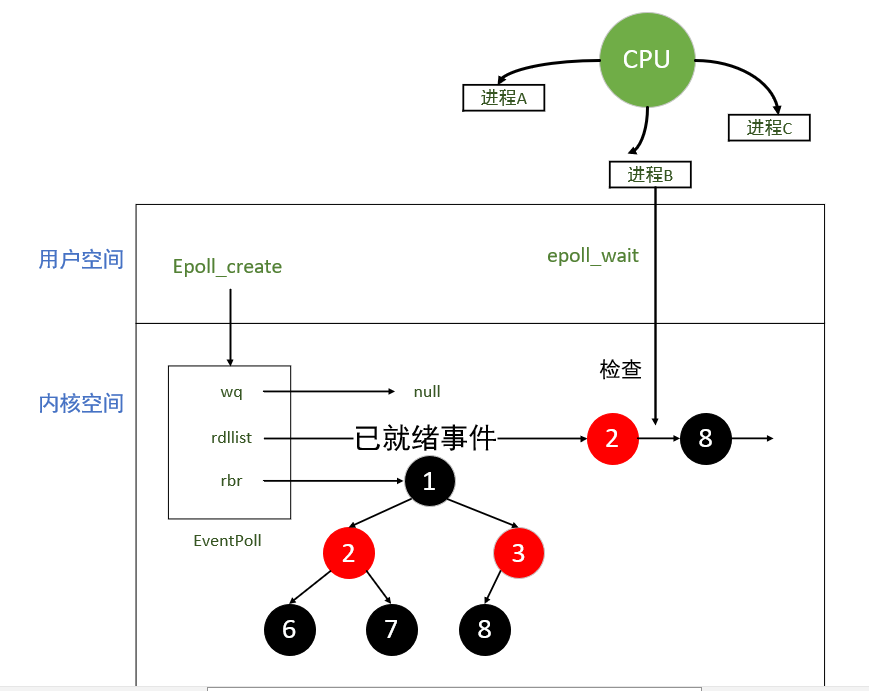
1. 当有网络数据发送过来的时候，系统会使用DMA技术，将网卡的数据拷贝至内存中；
2. 然后系统会给CPU发送中断信号，让CPU空出时间来处理这些数据；
3. CPU会根据数据包中的IP和端口号，找到对应地socket队列，将数据包放到socket队列中
4. 并执行回调函数ep\_poll\_callback，将epoll\_ctl阶段注册在回调函数中的，并且在红黑树中的epitem节点，放置到就绪队列中；



1. 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；



1. 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;

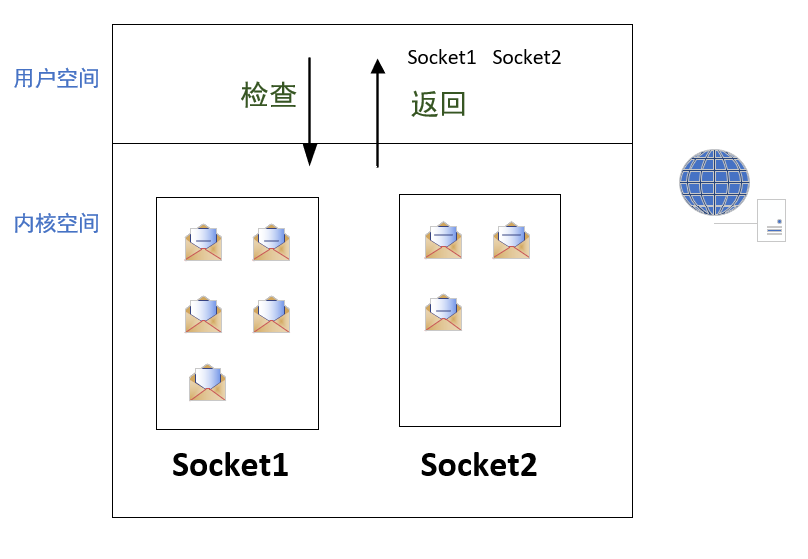


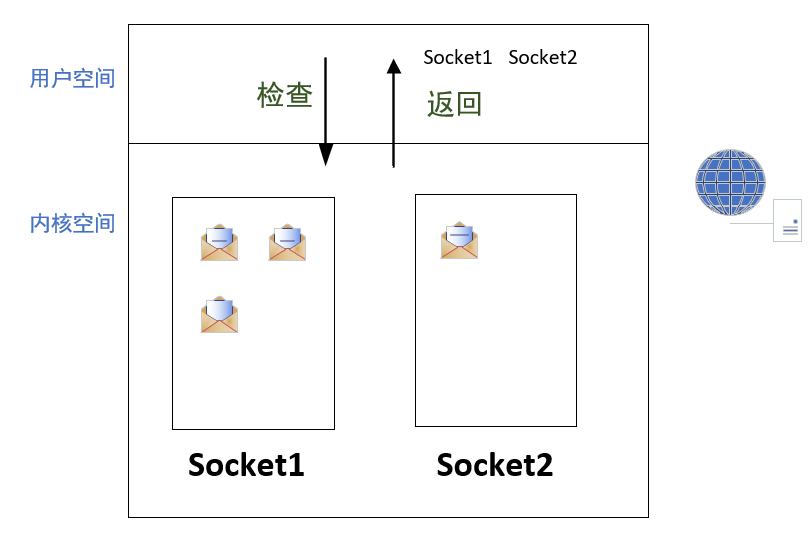
1. 需要注意的是Epoll中并没有使用MMAP技术
2. Epoll总结
3. 高效处理高并发下的大量连接，同时有非常优异的性能
4. 优点：
5. 监听的网络事件数量，没有1024等的数量限制
6. 每次返回具体就绪的网络事件，不需要遍历
7. 缺点：
8. 跨平台不够好，只支持linux
9. Select,poll 轻量级，epoll为重量级，移植性较差
10. 当监听网络事件比较少的情况下，select、poll会更优
11. IO复用的就绪事件触发机制
12. 简述

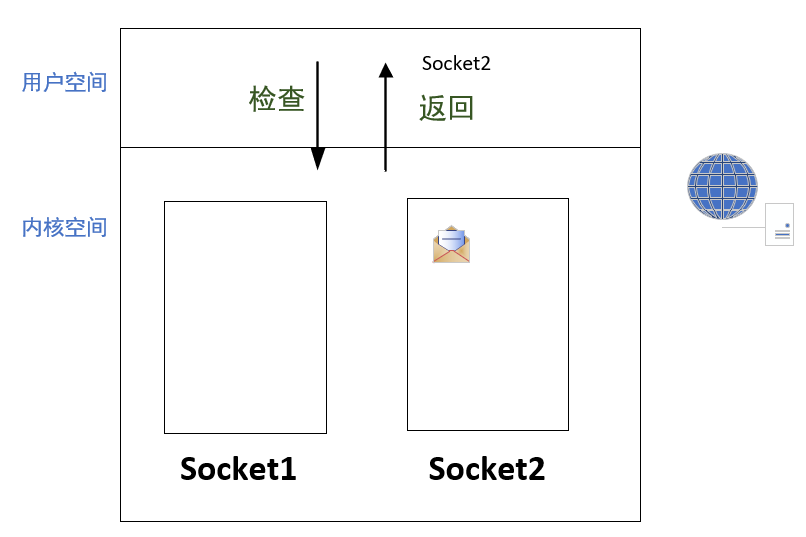
上述的三种IO多路复用，对于就绪事件的检测，操作系统分为两种方式:水平触发和边缘触发

1. 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；

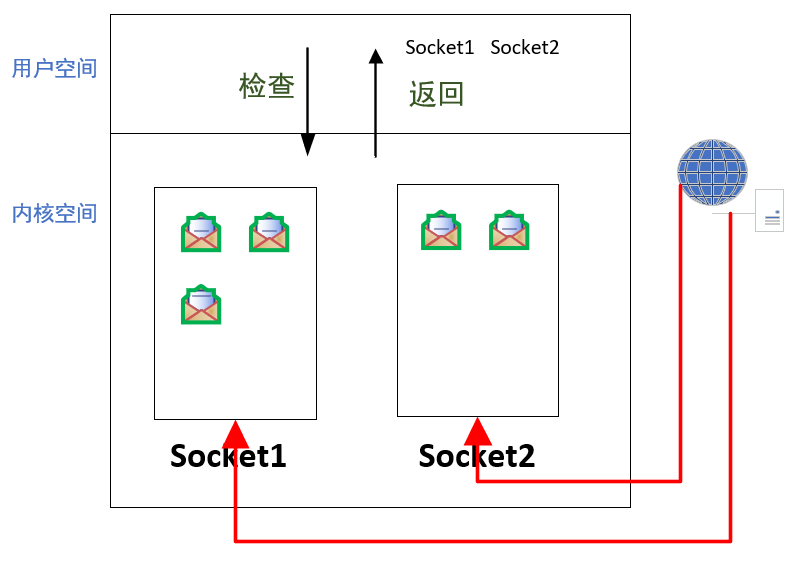


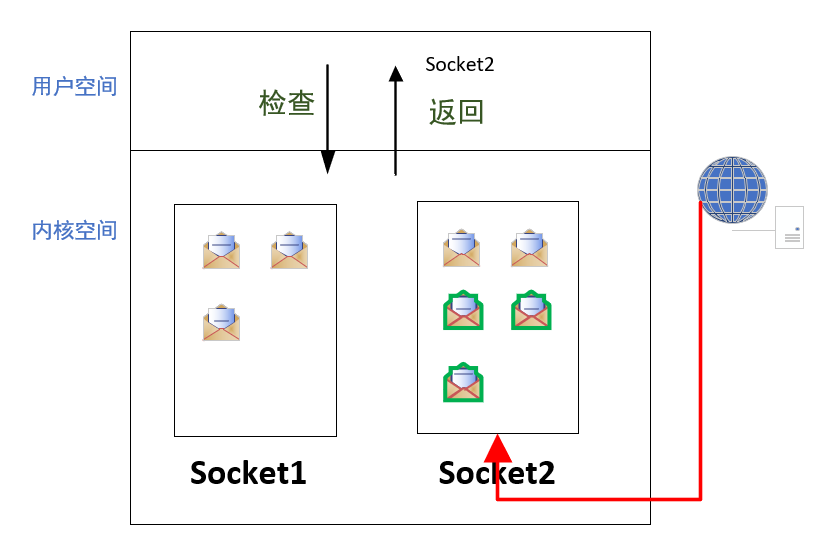




1. 边缘触发

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；





1. IO复用的总结
2. 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
3. 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