Table of Contents

用户空间与内核空间

现在操作系统都是采用虚拟存储器，那么对32位操作系统而言，它的寻址空间（虚拟存储空间）为4G（2的32次方）。操作系统的核心是内核，独立于普通的应用程序，可以访问受保护的内存空间，也有访问底层硬件设备的所有权限。为了保证用户进程不能直接操作内核（kernel），保证内核的安全，操作系统将虚拟空间划分为两部分，一部分为内核空间，一部分为用户空间。针对linux操作系统而言，将最高的1G字节（从虚拟地址0xC0000000到0xFFFFFFFF），供内核使用，称为内核空间，而将较低的3G字节（从虚拟地址0x00000000到0xBFFFFFFF），供各个进程使用，称为用户空间。

进程切换

为了控制进程的执行，内核必须有能力挂起正在CPU上运行的进程，并恢复以前挂起的某个进程的执行。这种行为被称为进程切换。因此可以说，任何进程都是在操作系统内核的支持下运行的，是与内核紧密相关的。

从一个进程的运行转到另一个进程上运行，这个过程中经过下面这些变化：

- 保存处理机上下文，包括程序计数器和其他寄存器。
- 更新PCB信息。
- 把进程的PCB移入相应的队列，如就绪、在某事件阻塞等队列。 选择另一个进程执行，并更新其PCB。
- 更新内存管理的数据结构。

- 恢复处理机上下文。

进程的阻塞

正在执行的进程，由于期待的某些事件未发生，如请求系统资源失败、等待某种操作的完成、新数据尚未到达或无新工作做等，则由系统自动执行阻塞原语(Block)，使自己由运行状态变为阻塞状态。可见，进程的阻塞是进程自身的一种主动行为，也因此只有处于运行态的进程（获得CPU），才可能将其转为阻塞状态。当进程进入阻塞状态，是不占用CPU资源的。

文件描述符

文件描述符（File descriptor）是计算机科学中的一个术语，是一个用于表述指向文件的引用的抽象化概念。

文件描述符在形式上是一个非负整数。实际上，它是一个索引值，指向内核为每一个进程所维护的该进程打开文件的记录表。当程序打开一个现有文件或者创建一个新文件时，内核向进程返回一个文件描述符。在程序设计中，一些涉及底层的程序编写往往会围绕着文件描述符展开。但是文件描述符这一概念往往只适用于UNIX、Linux这样的操作系统。

缓存 IO

缓存 IO 又被称作标准 IO，大多数文件系统的默认 IO 操作都是缓存 IO。在 Linux 的缓存 IO 机制中，操作系统会将 IO 的数据缓存在文件系统的页缓存（ page cache ）中，也就是说，数据会先被拷贝到操作系统内核的缓冲区中，然后才会从操作系统内核的缓冲区拷贝到应用程序的地址空间。

缓存 IO 的缺点：

数据在传输过程中需要在应用程序地址空间和内核进行多次数据拷贝操作，这些数据拷贝操作所带来的 CPU 以及内存开销是非常大的。

IO模型介绍

作者：cooffeelis 链接：https://www.jianshu.com/p/511b9cffbdac 來源：简书 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

常用的5种IO模型: blocking IO nonblocking IO IO multiplexing signal driven IO asynchronous IO

再说一下IO发生时涉及的对象和步骤:

对于一个network IO (这里我们以read举例)，它会涉及到两个系统对象：

- 一个是调用这个IO的process (or thread)
- 一个就是系统内核(kernel)

当一个read操作发生时，它会经历两个阶段：

- 等待数据准备,比如accept(), recv()等待数据 (Waiting for the data to be ready)
- 将数据从内核拷贝到进程中, 比如 accept()接受到请求,recv()接收连接发送的数据后需要复制到内核,再从内核复制到进程用户空间 (Copying the data from the kernel to the process)

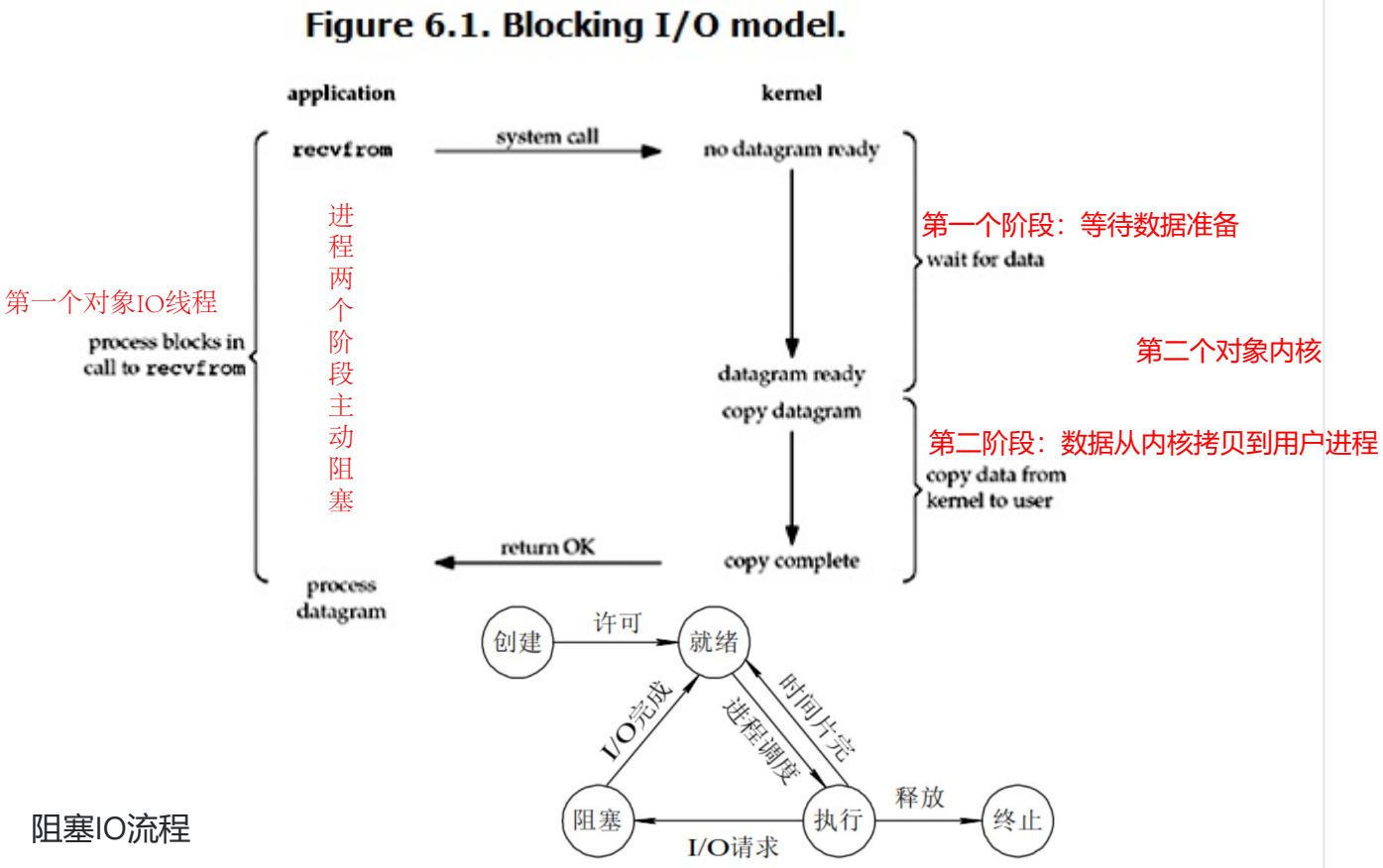
对于socket流而言,数据的流向经历两个阶段：

- 第一步通常涉及等待网络上的数据分组到达，然后被复制到内核的某个缓冲区。
- 第二步把数据从内核缓冲区复制到应用进程缓冲区。

记住这两点很重要，因为这些IO Model的区别就是在两个阶段上各有不同的情况。

阻塞 I/O（blocking IO)

在linux中，默认情况下所有的socket都是blocking，一个典型的读操作流程大概是这样：



阻塞IO流程

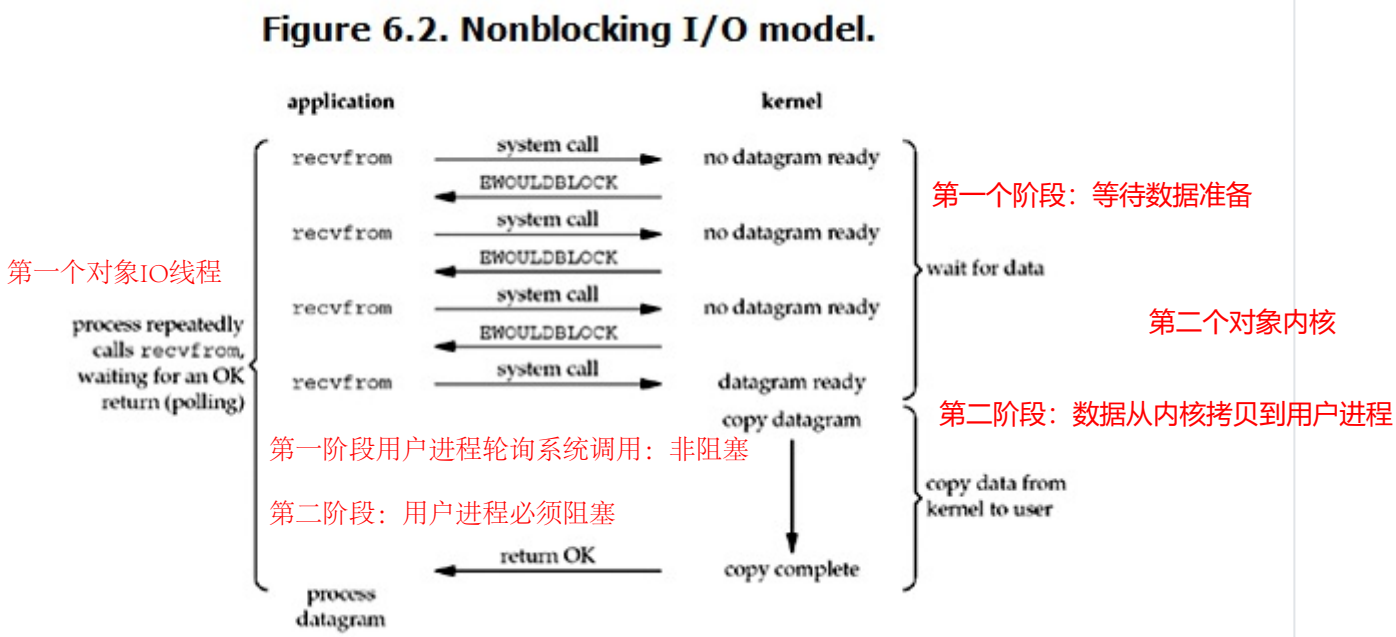
当用户进程调用了recvfrom这个系统调用，kernel就开始了IO的第一个阶段：准备数据（对于网络IO来说，很多时候数据在一开始还没有到达。比如，还没有收到一个完整的UDP包。这个时候kernel就要等待足够的数据到来）。这个过程需要等待，也就是说数据被拷贝到**操作系统内核的缓冲区**中是需要一个过程的。而在用户进程这边，整个进程会被阻塞（当然，是进程自己选择的阻塞）。当kernel一直等到数据准备好了，它就会**将数据从kernel中拷贝到用户内存**，然后kernel返回结果，用户进程才解除block的状态，重新运行起来。

> 所以，blocking IO的特点就是在IO执行的两个阶段都被block了。

非阻塞 I/O（nonblocking IO）

linux下，可以通过设置socket使其变为non-blocking。当对一个non-blocking socket执行读操作时，流程是这个样子：

非阻塞IO只是在第一个等待数据准备阶段不阻塞，第二个阶段数据拷贝仍然需要阻塞



非阻塞 I/O 流程

当用户进程发出read操作时，如果kernel中的数据还没有准备好，**那么它并不会block用户进程，而是立刻返回一个error。**从用户进程角度讲 ，它发起一个read操作后，并不需要等待，而是马上就得到了一个结果。用户进程判断结果是一个error时，它就知道数据还没有准备好，于是它可以再次发送read操作。一旦kernel中的数据准备好了，并且又再次收到了用户进程的system call，那么它马上就将数据拷贝到了用户内存，然后返回。

> 所以，nonblocking IO的特点是用户进程需要不断的主动询问kernel数据好了没有。

***值得注意的是,此时的非阻塞IO只是应用到等待数据上,当真正有数据到达执行recvfrom的时候,还是同步阻塞IO来的, 从图中的copy data from kernel to user可以看出 ***

I/O 多路复用（ IO multiplexing）

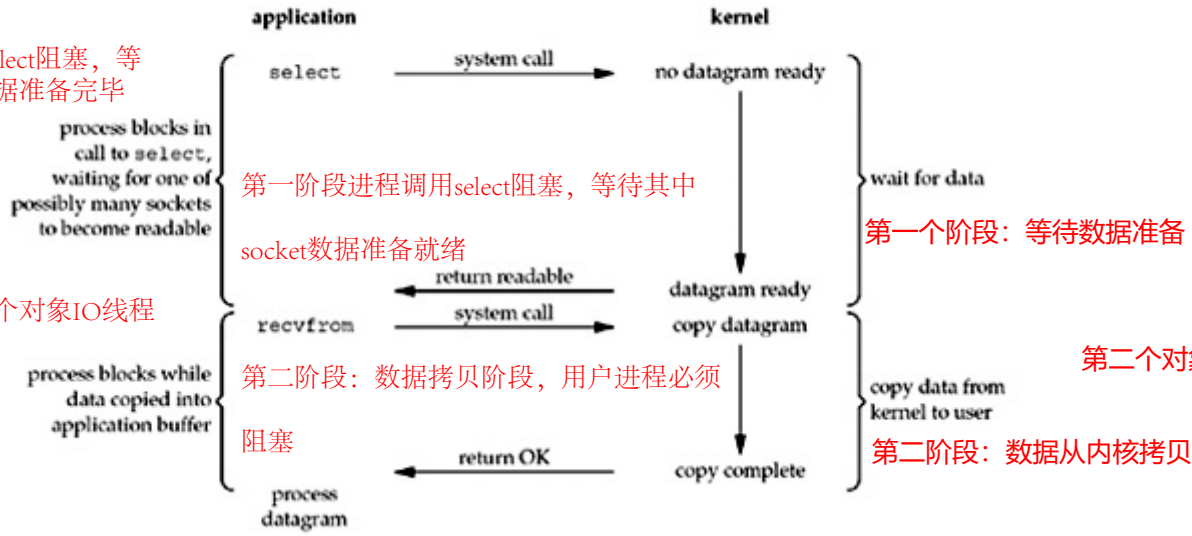
多个网络IO请求，复用同一个线程或用户进程

事件驱动的IO

IO multiplexing就是我们说的select，poll，epoll，有些地方也称这种IO方式为event driven IO。select/epoll的好处就在于单个process就可以同时处理多个网络连接的IO。它的基本原理就是select，poll，epoll这个function会不断的轮询所负责的所有socket，当某个socket有数据到达了，就通知用户进程。

Figure 6.3. I/O multiplexing model.

一个进程调用select阻塞，等待其中socket数据准备完毕



I/O 多路复用流程

这个图和blocking IO的图其实并没有太大的不同，事实上，还更差一些。因为这里需要使用两个system call (select 和 recvfrom)，而blocking IO只调用了一个system call (recvfrom)。但是，用select的优势在于单个线程可以同时处理多个connection。

> 所以，如果处理的连接数不是很高的话，使用select/epoll的web server不一定比使用multi-threading + blocking IO的web server性能更好，可能延迟还更大。select/epoll的优势并不是对于单个连接能处理得更快，而是在于能处理更多的连接。）

IO复用的实现方式目前主要有select、poll和epoll。

select和poll的原理基本相同：select和poll相同点

- 注册待侦听的fd(这里的fd创建时最好使用非阻塞)
- 每次调用都去检查这些fd的状态，当有一个或者多个fd就绪的时候返回
- 返回结果中包括已就绪和未就绪的fd

select、poll不同点

相比select，poll解决了单个进程能够打开的文件描述符数量有限制这个问题：select受限于FD_SIZE的限制，如果修改则需要修改这个宏重新编译内核；而poll通过一个pollfd数组向内核传递需要关注的事件，避开了文件描述符数量限制。

select、poll一样的缺点

此外，select和poll共同具有的一个很大的缺点就是包含大量fd的数组被整体复制于用户态和内核态地址空间之间，开销会随着fd数量增多而线性增大。

select和poll就类似于上面说的就餐方式。但当你每次都去询问时，老板会把所有你点的饭菜都轮询一遍再告诉你情况，当大量饭菜很长时间都不能准备好的情况下是很低效的。于是，老板有些不耐烦了，就让厨师每做好一个菜就通知他。这样每次你再去问的时候，他会直接把已经准备好的菜告诉你，你再去端。这就是事件驱动IO就绪通知的方式-epoll。

epoll优点

epoll的出现，解决了select、poll的缺点：

- 基于事件驱动的方式，避免了每次都要把所有fd都扫描一遍。只扫描wait就绪的事件
- epoll_wait只返回就绪的fd。
- epoll使用mmap内存映射技术避免了内存复制的开销。内存区域映射到内核空间，对于内核空间，用户空间两者之间需要大量数据传输等操作的话效率是非常高
- epoll的fd数量上限是操作系统的最大文件句柄数目,这个数目一般和内存有关，通常远大于1024。

目前，epoll是Linux2.6下最高效的IO复用方式，也是Nginx、Node的IO实现方式。而在freeBSD下，kqueue是另一种类似于epoll的IO复用方式。

此外，对于IO复用还有一个水平触发和边缘触发的概念：

- 水平触发：当就绪的fd未被用户进程处理后，下一次查询依旧会返回，这是select和poll的触发方式。
- 边缘触发：无论就绪的fd是否被处理，下一次不再返回。理论上性能更高，但是实现相当复杂，并且任何意外的丢失事件都会造成请求处理错误。epoll默认使用水平触发，通过相应选项可以使用边缘触发。

> 点评：**I/O 多路复用的特点是通过一种机制一个进程能同时等待多个文件描述符*，**而这些文件描述符（套接字描述符）其中的任意一个进入读就绪状态，select()函数就可以返回。 ***所以, IO多路复用，本质上不会有并发的功能，因为任何时候还是只有一个进程或线程进行工作，它之所以能提高效率是因为select\epoll 把进来的socket放到他们的`监视`列表里面，当任何socket有可读可写数据立马处理，那如果select\epoll 手里同时检测着很多socket， 一有动静马上返回给进程处理，总比一个一个socket过来,阻塞等待,处理高效率。*** 当然也可以多线程/多进程方式，一个连接过来开一个进程/线程处理，这样消耗的内存和进程切换页会耗掉更多的系统资源。 所以我们可以结合IO多路复用和多进程/多线程 来高性能并发，IO复用负责提高接受socket的通知效率，收到请求后，交给进程池/线程池来处理逻辑。

信号驱动

上文的就餐方式还是需要你每次都去问一下饭菜状况。于是，你再次不耐烦了，就跟老板说，哪个饭菜好了就通知我一声吧。然后就自己坐在桌子那里干自己的事情。更甚者，你可以把手机号留给老板，自己出门，等饭菜好了直接发条短信给你。这就类似信号驱动的IO模型。

流程如下：

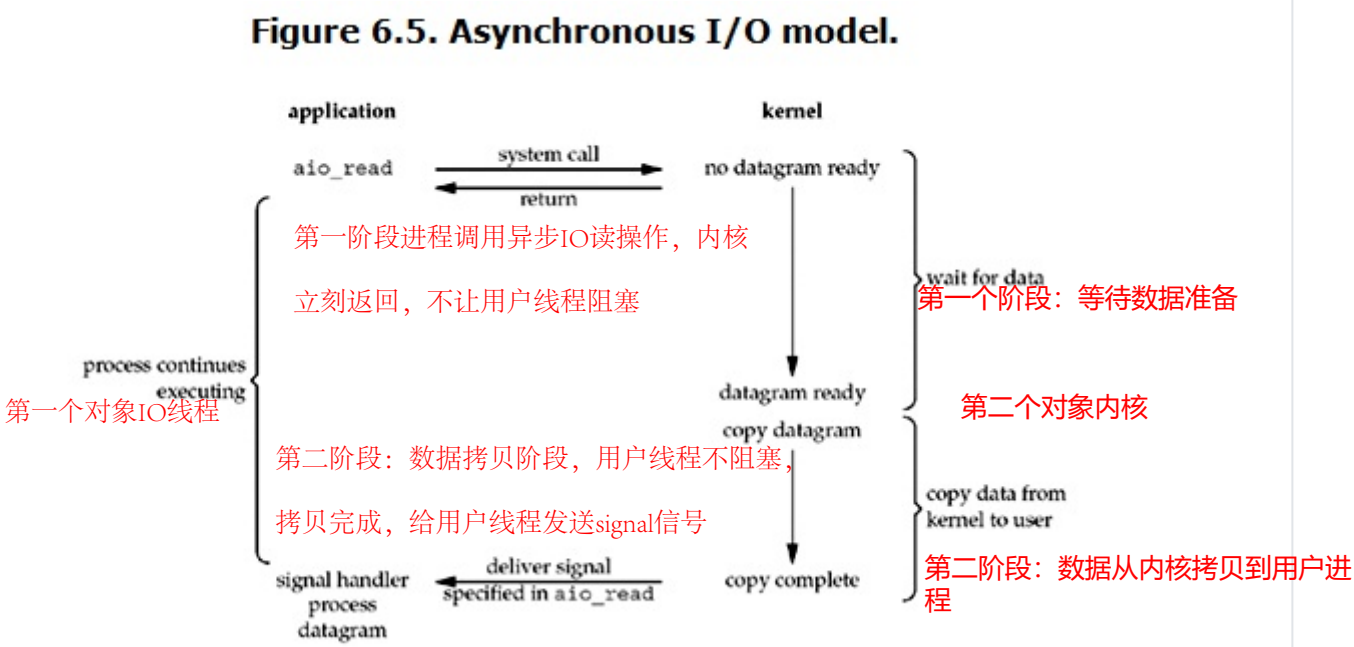
- 开启套接字信号驱动IO功能
- 系统调用sigaction执行信号处理函数（非阻塞，立刻返回）
- 数据就绪，生成sigio信号，通过信号回调通知应用来读取数据。

> 此种io方式存在的一个很大的问题：Linux中信号队列是有限制的，如果超过这个数字问题就无法读取数据。

异步非阻塞

异步 I/O（asynchronous IO）

linux下的asynchronous IO其实用得很少。先看一下它的流程：



异步IO 流程

用户进程发起read操作之后，立刻就可以开始去做其它的事。而另一方面，从kernel的角度，当它受到一个asynchronous read之后，首先它会立刻返回，所以不会对用户进程产生任何block。然后，kernel会等待数据准备完成，然后将数据拷贝到用户内存，当这一切都完成之后，kernel会给用户进程发送一个signal，告诉它read操作完成了。

阻塞IO,非阻塞IO 与 同步IO, 异步IO的区别和联系

阻塞IO VS 非阻塞IO：

概念： 阻塞和非阻塞关注的是**程序在等待调用结果（消息，返回值）时的状态**. 阻塞调用是指调用结果返回之前，当前线程会被挂起。调用线程只有在得到结果之后才会返回。非阻塞调用指在不能立刻得到结果之前，该调用不会阻塞当前线程。

例子：你打电话问书店老板有没有《分布式系统》这本书，你如果是阻塞式调用，你会一直把自己“挂起”，直到得到这本书有没有的结果，如果是非阻塞式调用，你不管老板有没有告诉你，你自己先一边去玩了， 当然你也要偶尔过几分钟check一下老板有没有返回结果。在这里阻塞与非阻塞与是否同步异步无关。跟老板通过什么方式回答你结果无关。

分析： 阻塞IO会一直block住对应的进程直到操作完成，而非阻塞IO在kernel还准备数据的情况下会立刻返回。

同步IO VS 异步IO：

概念： 同步与异步同步和异步关注的是___消息通信机制 ___(synchronous communication/ asynchronous communication)所谓同步，就是在发出一个_调用_时，在没有得到结果之前，该_调用_就不返回。但是一旦调用返回，就得到返回值了。换句话说，就是由_调用者_主动等待这个_调用_的结果。而异步则是相反，_调用_在发出之后，这个调用就直接返回了，所以没有返回结果。换句话说，当一个异步过程调用发出后，调用者不会立刻得到结果。而是在_调用_发出后，_被调用者_通过状态、通知来通知调用者，或通过回调函数处理这个调用。

典型的异步编程模型比如Node.js举个通俗的例子：你打电话问书店老板有没有《分布式系统》这本书，如果是同步通信机制，书店老板会说，你稍等，”我查一下"，然后开始查啊查，等查好了（可能是5秒，也可能是一天）告诉你结果（返回结果）。而异步通信机制，书店老板直接告诉你我查一下啊，查好了打电话给你，然后直接挂电话了（不返回结果）。然后查好了，他会主动打电话给你。在这里老板通过”回电”这种方式来回调。

分析： 在说明同步IO和异步IO的区别之前，需要先给出两者的定义。Stevens给出的定义（其实是POSIX的定义）是这样子的：

> A synchronous I/O operation causes the requesting process to be blocked until that I/O operation completes; An asynchronous I/O operation does not cause the requesting process to be blocked;

两者的区别就在于同步IO做”IO operation”的时候会将process阻塞。按照这个定义，之前所述的***阻塞IO,非阻塞IO ， IO复用都属于同步IO。*** 有人可能会说，非阻塞IO 并没有被block啊。这里有个非常”狡猾”的地方，定义中所指的”IO operation”是指真实的IO操作，就是例子中的recvfrom这个system call。非阻塞IO在执行recvfrom这个system call的时候，如果kernel的数据没有准备好，这时候不会block进程。但是，当kernel中数据准备好的时候，recvfrom会将数据从kernel拷贝到用户内存中，这个时候进程是被block了，在这段时间内，进程是被block的。

而异步IO则不一样，当进程发起IO 操作之后，就直接返回再也不理睬了，直到kernel发送一个信号，告诉进程说IO完成。在这整个过程中，进程完全没有被block。

IO模型的形象举例

最后，再举几个不是很恰当的例子来说明这四个IO Model: 有A，B，C，D四个人在钓鱼： A用的是最老式的鱼竿，所以呢，得一直守着，等到鱼上钩了再拉杆； B的鱼竿有个功能，能够显示是否有鱼上钩，所以呢，B就和旁边的MM聊天，隔会再看看有没有鱼上钩，有的话就迅速拉杆； C用的鱼竿和B差不多，但他想了一个好办法，就是同时放好几根鱼竿，然后守在旁边，一旦有显示说鱼上钩了，它就将对应的鱼竿拉起来； D是个有钱人，干脆雇了一个人帮他钓鱼，一旦那个人把鱼钓上来了，就给D发个短信。

Select/Poll/Epoll 轮询机制

select，poll，epoll本质上都是同步I/O，因为他们都需要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是阻塞的 Select/Poll/Epoll 都是IO复用的实现方式， 上面说了使用IO复用，会把socket设置成non-blocking，然后放进Select/Poll/Epoll 各自的监视列表里面，那么，他们的对socket是否有数据到达的监视机制分别是怎样的？效率又如何？我们应该使用哪种方式实现IO复用比较好？下面列出他们各自的实现方式，效率，优缺点：

（1）select，poll实现需要自己不断轮询所有fd集合，直到设备就绪，期间可能要睡眠和唤醒多次交替。而epoll其实也需要调用epoll_wait不断轮询就绪链表，期间也可能多次睡眠和唤醒交替，但是它是设备就绪时，调用回调函数，把就绪fd放入就绪链表中，并唤醒在epoll_wait中进入睡眠的进程。虽然都要睡眠和交替，但是select和poll在“醒着”的时候要遍历整个fd集合，而epoll在“醒着”的时候只要判断一下就绪链表是否为空就行了，这节省了大量的CPU时间。这就是回调机制带来的性能提升。

（2）select，poll每次调用都要把fd集合从用户态往内核态拷贝一次，并且要把current往设备等待队列中挂一次，而epoll只要一次拷贝，而且把current往等待队列上挂也只挂一次（在epoll_wait的开始，注意这里的等待队列并不是设备等待队列，只是一个epoll内部定义的等待队列）。这也能节省不少的开销。

Java网络编程模型

上文讲述了UNIX环境的五种IO模型。基于这五种模型，在Java中，随着NIO和NIO2.0(AIO)的引入，一般具有以下几种网络编程模型：

- BIO
- NIO
- AIO

BIO

BIO是一个典型的网络编程模型，是通常我们实现一个服务端程序的过程，步骤如下：

- 主线程accept请求阻塞
- 请求到达，创建新的线程来处理这个套接字，完成对客户端的响应。

- 主线程继续accept下一个请求

这种模型有一个很大的问题是：当客户端连接增多时，服务端创建的线程也会暴涨，系统性能会急剧下降。因此，在此模型的基础上，类似于 tomcat的bio connector，采用的是线程池来避免对于每一个客户端都创建一个线程。有些地方把这种方式叫做伪异步IO(把请求抛到线程池中异步等待处理)。

NIO

JDK1.4开始引入了NIO类库，这里的NIO指的是New IO，主要是使用Selector多路复用器来实现。Selector在Linux等主流操作系统上是通过epoll实现的。

NIO的实现流程，类似于select：

- 创建ServerSocketChannel监听客户端连接并绑定监听端口，设置为非阻塞模式。
- 创建Reactor线程，创建多路复用器(Selector)并启动线程。
- 将ServerSocketChannel注册到Reactor线程的Selector上。监听accept事件。
- Selector在线程run方法中无线循环轮询准备就绪的Key。
- Selector监听到新的客户端接入，处理新的请求，完成tcp三次握手，建立物理连接。
- 将新的客户端连接注册到Selector上，监听读操作。读取客户端发送的网络消息。
- 客户端发送的数据就绪则读取客户端请求，进行处理。

相比BIO，NIO的编程非常复杂。

AIO

JDK1.7引入NIO2.0，提供了异步文件通道和异步套接字通道的实现。其底层在windows上是通过IOCP，在Linux上是通过epoll来实现的(LinuxAsynchronousChannelProvider.java,UnixAsynchronousServerSocketChannelImpl.java)。

- 创建AsynchronousServerSocketChannel，绑定监听端口
- 调用AsynchronousServerSocketChannel的accpet方法，传入自己实现的CompletionHandler。包括上一步，都是非阻塞的
- 连接传入，回调CompletionHandler的completed方法，在里面，调用AsynchronousSocketChannel的read方法，传入负责处理数据的CompletionHandler。
- 数据就绪，触发负责处理数据的CompletionHandler的completed方法。继续做下一步处理即可。
- 写入操作类似，也需要传入CompletionHandler。

其编程模型相比NIO有了不少的简化。

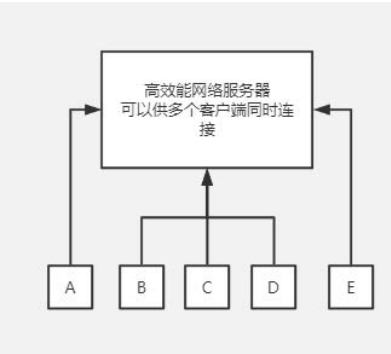
对比

.	同步阻塞IO	伪异步IO	NIO	AIO
客户端数目： IO线程	1 : 1	m : n	m : 1	m : 0
IO模型	同步阻塞IO	同步阻塞IO	同步非阻塞IO	异步非阻塞IO
吞吐量	低	中	高	高
编程复杂度	简单	简单	非常复杂	复杂

Linux的select源码解析

1. 场景概述

一个用户客户端，发送多个网络IO请求



2 .单线程在BIO情况下存在的问题

A,B,C,D,E由同一个客户端，单个线程按照先后顺序，向网络服务器发送网络IO请求。如果A请求的网络IO过程中，数据迟迟不能到达，导致后A请求阻塞，后面的B,C,D,E请求也会被阻塞。问题：多单线程下个网络IO请求阻塞严重。

3. 多线程解决单线程存在的问题

同一个客户端，开启五个线程分别发送网络IO请求。
优点：解决了单线程下一个网络IO阻塞，导致后面的网路IO都阻塞的问题
缺点：每个网络IO都开启一个线程，因此需要CPU频繁切换线程，线程的切换成本很高，因此多线程也不是解决BIO的最好方式

4. 线程池解决多线程存在的问题
同一个客户端使用线程池发送网络IO请求

优点：比单线程下网络IO阻塞情况降低，比多线程下线程个数降低
缺点：线程池仍然属于多线程，cpu切换成本依然很高，并且线程池中某个线程网络IO阻塞，导致该线程不能被后面的网络IO请求使用。

总结：上述问题主要原因是BIO，阻塞时IO请求，一旦请求的数据不能到达，当前线程由运行状态切换到阻塞状态，在此阻塞期间，该线程不能做其余的事情。可以理解BIO是重量级的阻塞IO

NIO非阻塞式IO解决BIO存在的阻塞问题

5. 如果是你自己如何设计一个非阻塞高性能服务请求？

回到原来的单线程多个网络IO请求例子。

```
while(true){ 使用单个线程死循环监听
   for(遍历五个IO 从A到E){ 用户态线程监听数据是否准备就绪
         if(X数据准备完毕)
            执行数据处理
         }

}
```

优点：解决了原始单线程下一个网络IO请求，可能导致后面的IO阻塞，并且只使用单个线程，没有cpu切换的开销
缺点：用户态线程死循环监听数据是否准备就绪，cpu利用率不高，并且很多是否cpu都是无效工作。

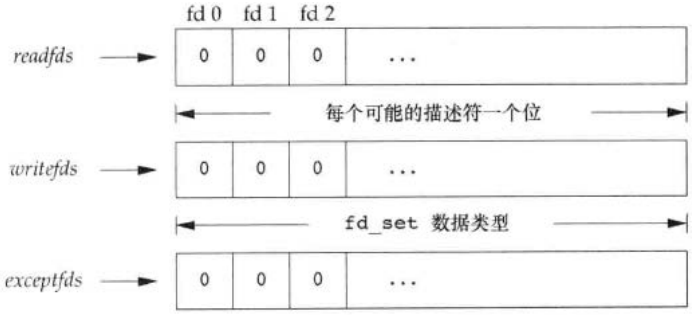
Linux的select实现NIO的多路复用

文件描述符请查看：背景介绍中的文件描述符

fd代表文件描述符数值，fdset代表文件描述符的集合，fdset是一个位图结构，类似于OS内存分页中位图,000010010000这种结构。

```
int FD_ISSET(int fd, fd_set *fdset);  //返回值：若fd在文件描述符集中，返回非0值；否则，返回0
void FD_CLR(int fd, fd_set *fdset);   //清除最后一位
void FD_SET(int fd, fd_set *fdset);   //开启描述符中的一位
void FD_ZERO(fd_set *fdset);          //所有描述符位置位0
```

```
#include <sys/select.h>
int select(int maxfpd1, fdset *read_fds, fdset *write_fds, fdset *exception_fds, struct timeval *restrict tvpr);
//返回值为可以操作的文件描述符的数量。
```



读操作的文件描述符，哪一位为1代表对应的文件有读取需求

图 14-15　对 select 指定读、写和异常条件描述符

maxfdp1 最大的文件描述符编号+1，最大为1024。通过指定我们关注的最大的描述符，内核只需要在此范围内搜索打开的位。比如select返回后，for i in maxfdp:只在maxfdp1之内进行搜索对应的文件。

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
memset(&addr, 0, sizeof (addr));
addr.sin_family = AF_INET;
addr.sin_port = htons(2000);
addr.sin_addr.s_addr = INADDR_ANY;
bind(sockfd,(struct sockaddr*)&addr ,sizeof(addr));
listen (sockfd, 5);

for (i=0;i<5;i++)
{
  memset(&client, 0, sizeof (client));
  addrlen = sizeof(client);
  fds[i] = accept(sockfd,(struct sockaddr*)&client, &addrlen);
  if(fds[i] > max)
      max = fds[i];
}
```

上半部分主要做了两件事：
1.创建了socket客户端
2.创建了5个文件描述符，并把这五个文件描述符放入到了数组中

创建五个文件描述符，放到数组内

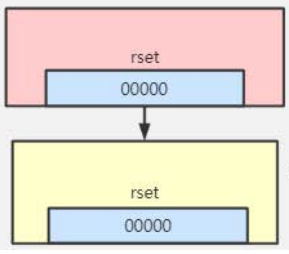
```
while(1){ 死循环遍历
  FD_ZERO(&rset);
  for (i = 0; i< 5; i++ ) {
      FD_SET(fds[i],&rset);
  }

  puts("round again");
  select(max+1, &rset, NULL, NULL, NULL);

  for(i=0;i<5;i++) {
      if (FD_ISSET(fds[i], &rset)){
          memset(buffer,0,MAXBUF);
          read(fds[i], buffer, MAXBUF);
          puts(buffer);
      }
  }
```

设置可读的fd_set集合，每次便利都要设置

执行select监听所有的读文件事件，一旦有一个文件就绪，就返回

循环遍历被内核置位的fd_set集合，就可以找到对应的文件描述符数组

select方法的参数
1. 读文件描述符集合、写文件描述符集合、异常描述符集合、超时时间
2. 我们更关心读文件操作，所以其他可以置为NULL
其中第二个参数&rset是一个bitmap用于表征哪些文件描述符是被启用的或者说被监听的
3. 这个bitmap大小为1024位，



1. 将rset从用户态拷贝到内核态
由内核态直接判断文件描述符是否有数据的操作，这样效率更高
2. 以前判断时需要由用户态切换到内核态所以效率低

FD_ZERO(rset) 每次select调用前都需要清空（FD_ZERO）传入的fd事件集合,因为select之后内核修改了rset集合，因此rset不能重用

分析select函数的执行流程：
1. select是一个阻塞函数，当没有数据时，会一直阻塞在select那一行。　　rset是fd_set型变量，读取操作
2. 当有数据时会将rset中对应的那一位置为1
3. select函数返回，不再阻塞
4. 遍历文件描述符数组，判断哪个fd被置位了
5. 读取数据，然后处理

select函数的缺点
1. bitmap默认大小为1024，虽然可以调整但还是有限度的　bitmap就是df_set集合，位图
2. rset每次循环都必须重新置位为0，不可重复使用 因为内核已经修改rset为就绪的读取文件描述符
3. 尽管将rset从用户态拷贝到内核态 由内核态判断是否有数据，但是还是有拷贝的开销
4. 当有数据时select就会返回，但是select函数并不知道哪个文件描述符有数据了，后面还需要再次对文件描述符数组进行遍历。效率比较低 循环遍历max+1个位置

Linux的poll实现NIO的多路复用

select解决了我们自己设计的while循环服务器什么缺点？

1. 之前自己的while循环判断IO是否准备完毕由用户态线程判断，select使用了内核判断哪些IO请求数据准备完毕。整体来说select还算优秀，从九十年到到现在应用场景很多。

```
for (i=0;i<5;i++)
{
  memset(&client, 0, sizeof (client));
  addrlen = sizeof(client);
  pollfds[i].fd = accept(sockfd,(struct sockaddr*)&client, &addrlen);
  pollfds[i].events = POLLIN;
}
sleep(1);
while(1){
  puts("round again");
  poll(pollfds, 5, 50000);

  for(i=0;i<5;i++) {
      if (pollfds[i].revents & POLLIN){
          pollfds[i].revents = 0;
          memset(buffer,0,MAXBUF);
          read(pollfds[i].fd, buffer, MAXBUF);
          puts(buffer);
      }
  }
}
```

初始化需要监听的文件和事件结构体数组。pollfds

仍然是死循环

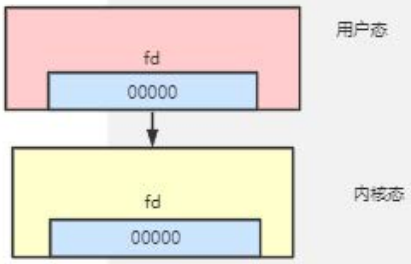
循环遍历结构体数组

```
struct pollfd {
      int fd;
      short events;
      short revents;
};
```

1. fd：文件描述符
2. events：在意的事件是什么，如果在于读就是POLLIN，如果在意写就是POLLOUT
3. revents:对events的回馈，开始时为0，**当有数据可读时就置为POLLIN**，类似于上面的rset

poll的参数：
1. 自定义的结构体数组
2. 数组的长度
3. 超时时间

poolfds对应的revent如果被改变，说明该文件的IO数据已经准备完毕，执行数据处理操作。



poll的执行流程：
1. 将五个fd从用户态拷贝到内核态
2. poll为阻塞方法，执行poll方法，如果有数据会将fd对应的revents置为POLLIN
3. poll方法返回
4. 循环遍历，查找哪个fd被置位为POLLIN了
5. 将revents重置为0 便于复用
6. 对置位的fd进行读取和处理

poolfds结构体数组，revent被内核修改即该文件IO数据准备就绪

解决的问题：优点：和select不同点
1. 解决了bitmap大小限制 polllfds结构体数组可以远大于1024
2. 解决了rset不可重用的情况

后面由于二者原理相同，所以没能解决
select中rset被内核全部重置，poll中pollfds结构体数组内核只重置了revents元素，其余没有改动

poll缺点：和select相同点
1. 结构体数据pollfds和select中fd_set结构体一样，需要拷贝到内核。数据拷贝
2. select和poll一样，返回时候需要全部遍历监听的文件，效率低。全部遍历，

Linux的epoll实现NIO的多路复用

epoll是在2.6内核中提出的，是之前的select和poll的增强版本。相对于select和poll来说，epoll更加灵活，没有文件描述符个数限制。epoll使用一个文件描述符管理多个描述符，将用户关系的文件描述符的事件存放到内核的一个事件表中，这样在用户空间和内核空间的copy只需一次。即只需要向内核的内存中的事件表修改一次。视频推荐：https://www.bilibili.com/video/BV1Ka4y177gs?from=search&seid=17334354775071952235

epoll接口
epoll操作过程需要三个接口，分别如下：

```
#include <sys/epoll.h>
int epoll_create(int size);

int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);

int epoll_wait(int epfd, struct epoll_event * events, int maxevents, int timeout);
```

(1) epoll_create 函数是一个系统函数，函数将在内核空间内开辟一块新的空间，可以理解为epoll结构空间，返回值为epoll的文件描述符编号，方便后续操作使用。

(2) epoll_ctl 是 epoll的事件注册函数，epoll与select不同，select函数是调用时指定需要监听的描述符和事件，epoll先将用户感兴趣的描述符事件注册到epoll空间内，此函数是非阻塞函数，作用仅仅是增删改epoll空间内的描述符信息。

参数一：epfd，很简单，epoll结构的进程fd编号，函数将依靠该编号找到对应的epoll结构。

参数二：op，表示当前请求类型，由三个宏定义
(EPOLL_CTL_ADD：注册新的fd到epfd中）、（EPOLL_CTL_MOD：修改已经注册的fd的监听事件）、（EPOLL_CTL_DEL：从epfd中删除一个fd）

参数三：fd，需要监听的文件描述符。一般指socket_fd。 文件描述符fd和感兴趣的事件注册到内核的epoll_event中

参数四：event，告诉内核对该fd资源感兴趣的事件。

struct epoll_event结构如下：

```
struct epoll_event {
    __uint32_t events;  /* Epoll events */
    epoll_data_t data;  /* User data variable */
};
```

events可以是以下几个宏的集合：
EPOLLIN、EPOLLOUT、EPOLLPRI、EPOLLERR、EPOLLHUP（挂断）、EPOLLET（边缘触发）、EPOLLONESHOT（只监听一次，事件触发后自动清除该fd，从epoll列表）

(3) epoll_wait 等待事件的产生，类似于select()调用。根据参数timeout，来决定是否阻塞。

参数一：epfd，指定感兴趣的epoll事件列表。

参数二：*events，是一个指针，必须指向一个epoll_event结构数组，当函数返回时，内核会把就绪状态的数据拷贝到该数组中！

参数三：maxevents，标明参数二 epoll_event数组最多能接收的数据量，即本次操作最多能获取多少就绪数据。

参数四：timeout，单位为毫秒。
0：表示立即返回，非阻塞调用。
-1：阻塞调用，直到有用户感兴趣的事件就绪为止。
>0：阻塞调用，阻塞指定时间内如果有事件就绪则提前返回，否则等待指定时间后返回。

返回值：本次就绪的fd个数。

工作模式 文件描述符的操作
epoll对文件描述符的操作有两种模式：LT（水平触发）和ET（边缘触发）。LT模式是默认模式，LT模式与ET模式的区别如下
本次wait得到的事件，可以不处理，下次wait可以接着处理
LT（水平触发）：事件就绪后，用户可以选择处理或者不处理，如果用户本次未处理，那么下次调用epoll_wait时仍然会将未处理的事件打包给你。
ET（边缘触发）：事件就绪后，用户必须处理，因为内核不给你兜底了，内核把就绪的事件打包给你后，就把对应的就绪事件清理掉了。
本次wait得到的事件，必须处理，否则下次就没有该事件
ET模式在很大程度上减少了epoll事件被重复触发的次数，因此效率要比LT模式高。

epoll源码解析

```
1  int epfd = epoll_create(10);
2  ...
3  ...
4  struct epoll_event events[5];
5  for(i=0; i<5; i++){
6      static struct epoll_event ev;
7      memset(&client, 0, sizeof(client));
8      addrlen = sizeof(client);
9      ev.data.fd = accept(sockfd, (struct sockaddr*)&client, &addrlen);
10      ev.events = EPOLLIN;
11      epoll_ctl(epfd, EPOLL_CTL_ADD, ev.data.fd, &ev);
12 }
13
14 while(1) {
15     prints("round again");
16     nfds = epoll_wait(epfd, events, 5, 10000);
17
18     for(i=0; i<nfds; i++) {
19         memset(buffer, 0, MAXBUF);
20         read(events[i].data.fd, buffer, MAXBUF);
21         prints(buffer);
```

1、内核开辟空间，用于存储监听事件列表和就绪的事件列表，返回的是该“空间”的文件描述符epfd，linux万物皆文件

创建一个感兴趣的事件 event

初始化感兴趣的事件events和文件描述符fd

2. 事件和对应的文件描述符注册到内核开辟的eventpoll中（根据epfd文件描述符）

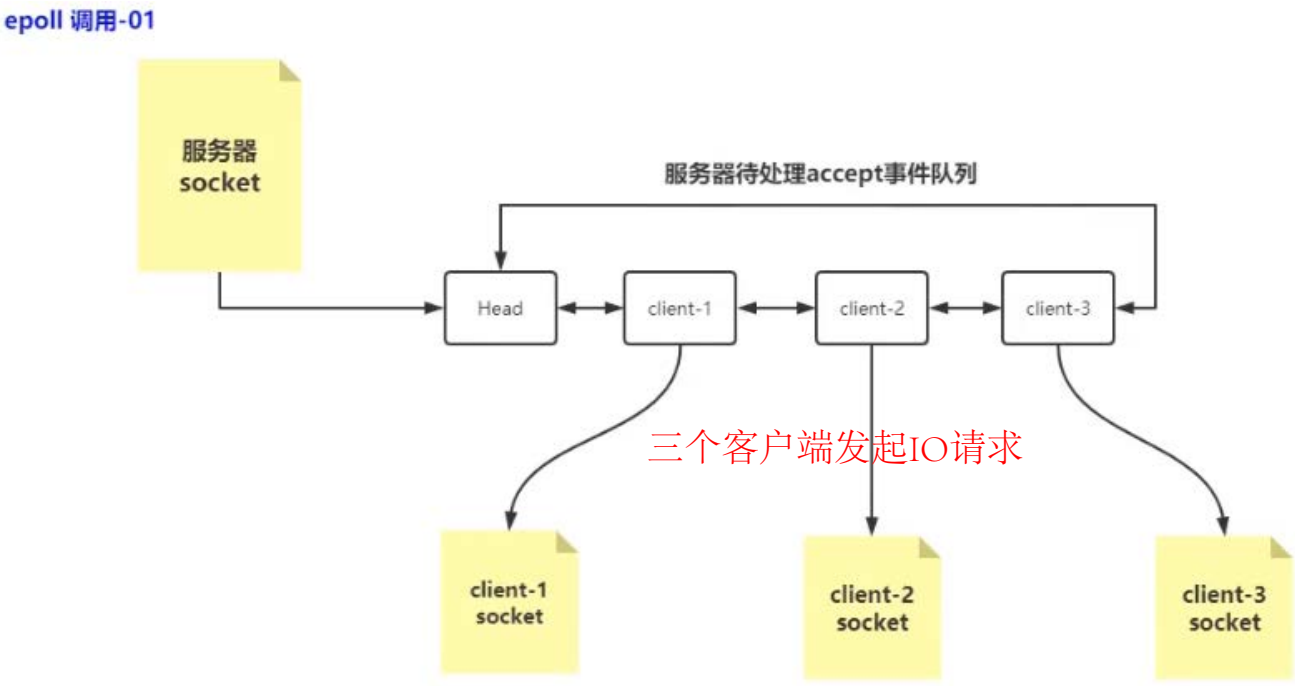
单个线程死循环监听就绪事件

3、监听内核epfd的eventpoll中的就绪队列事件，就绪的事件copy到外部的events数组中。

4、返回就绪队列中事件

获取该事件对应的文件描述符和处理数据buffer

Epoll原理解析过程



epoll 调用-02

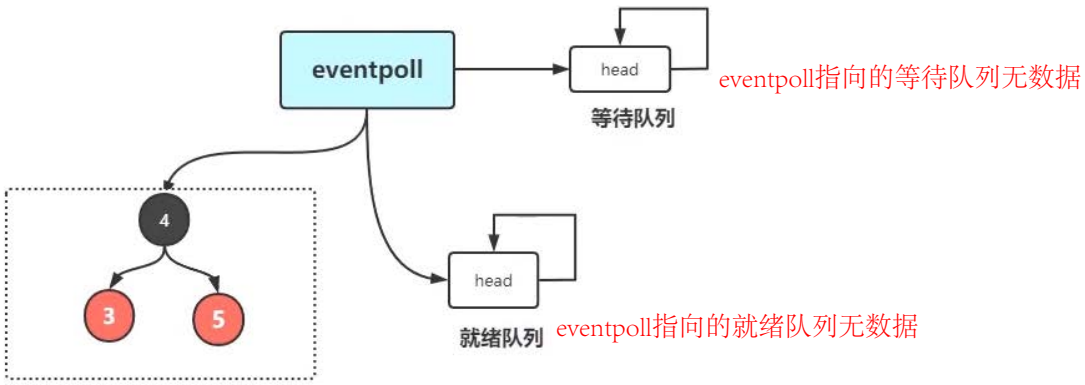
代码执行完：epoll_create(...) 函数

进程A用户态堆栈：
....
fds = [3, 4, 5]客户端三个socket对应的fd
epfd = 6　　eventpoll的fd

内核开辟空间eventpoll对应的文件描述符fd=6

epoll 调用-03

epoll_ctl 将进程关心的套接字事件 加入到eventpoll内

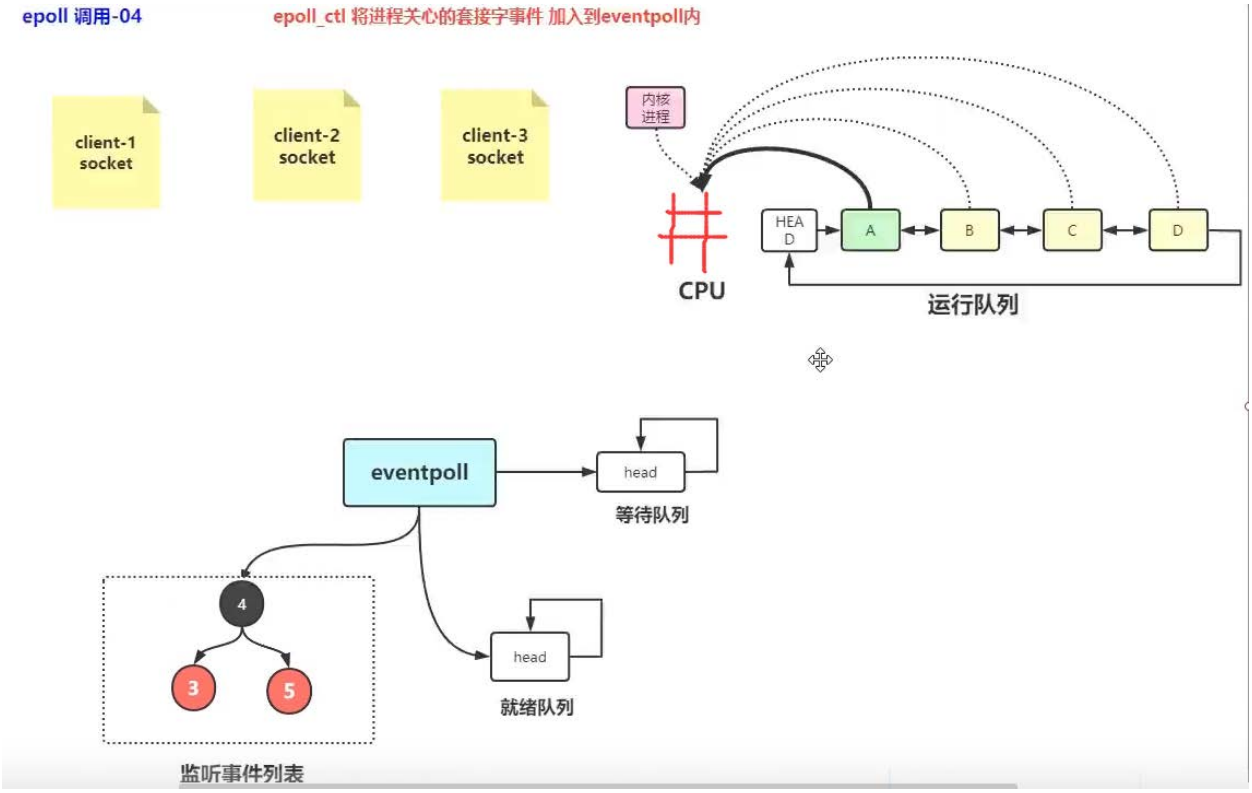


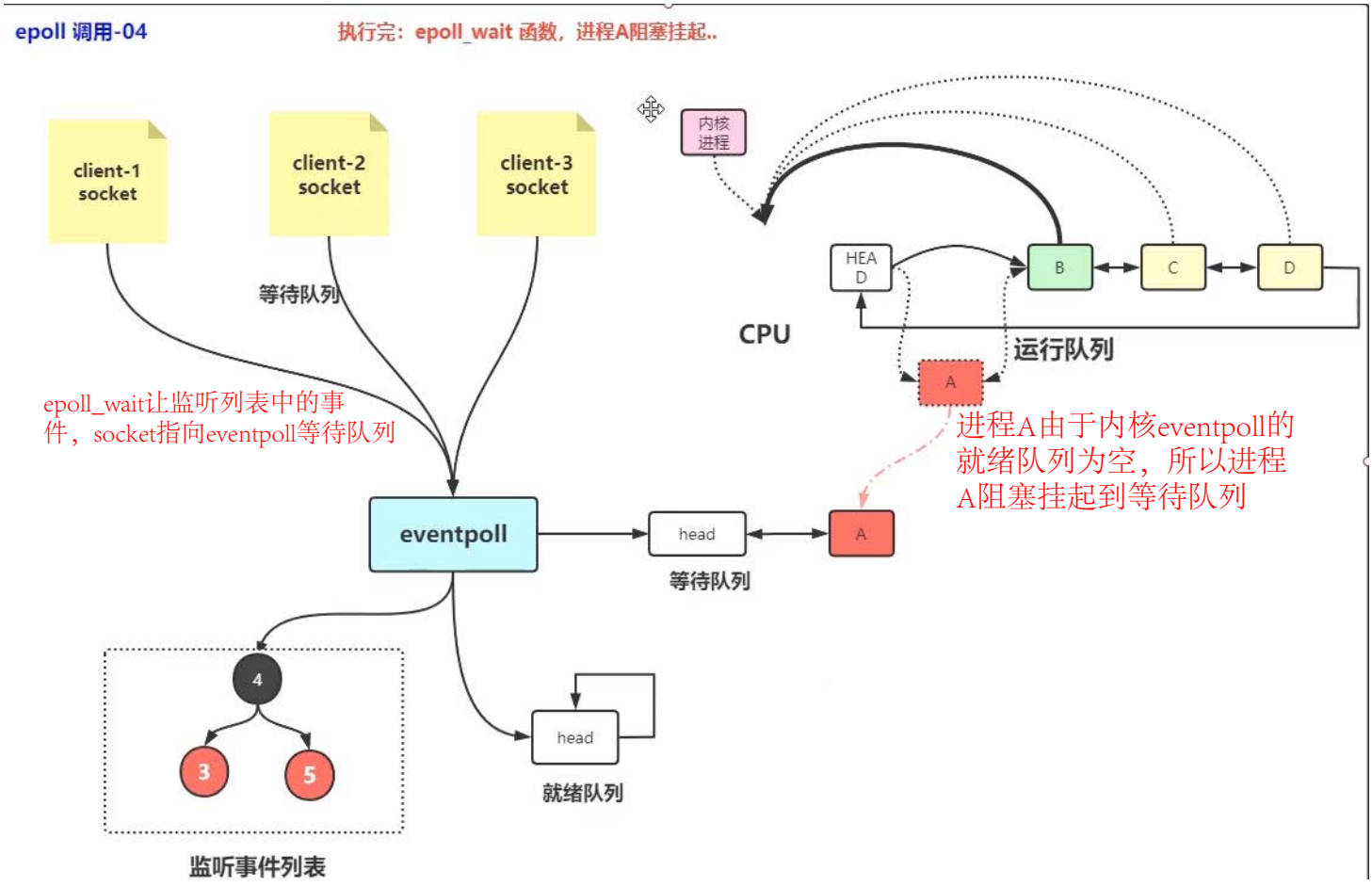
eventpoll指向的等待队列无数据

eventpoll指向的就绪队列无数据

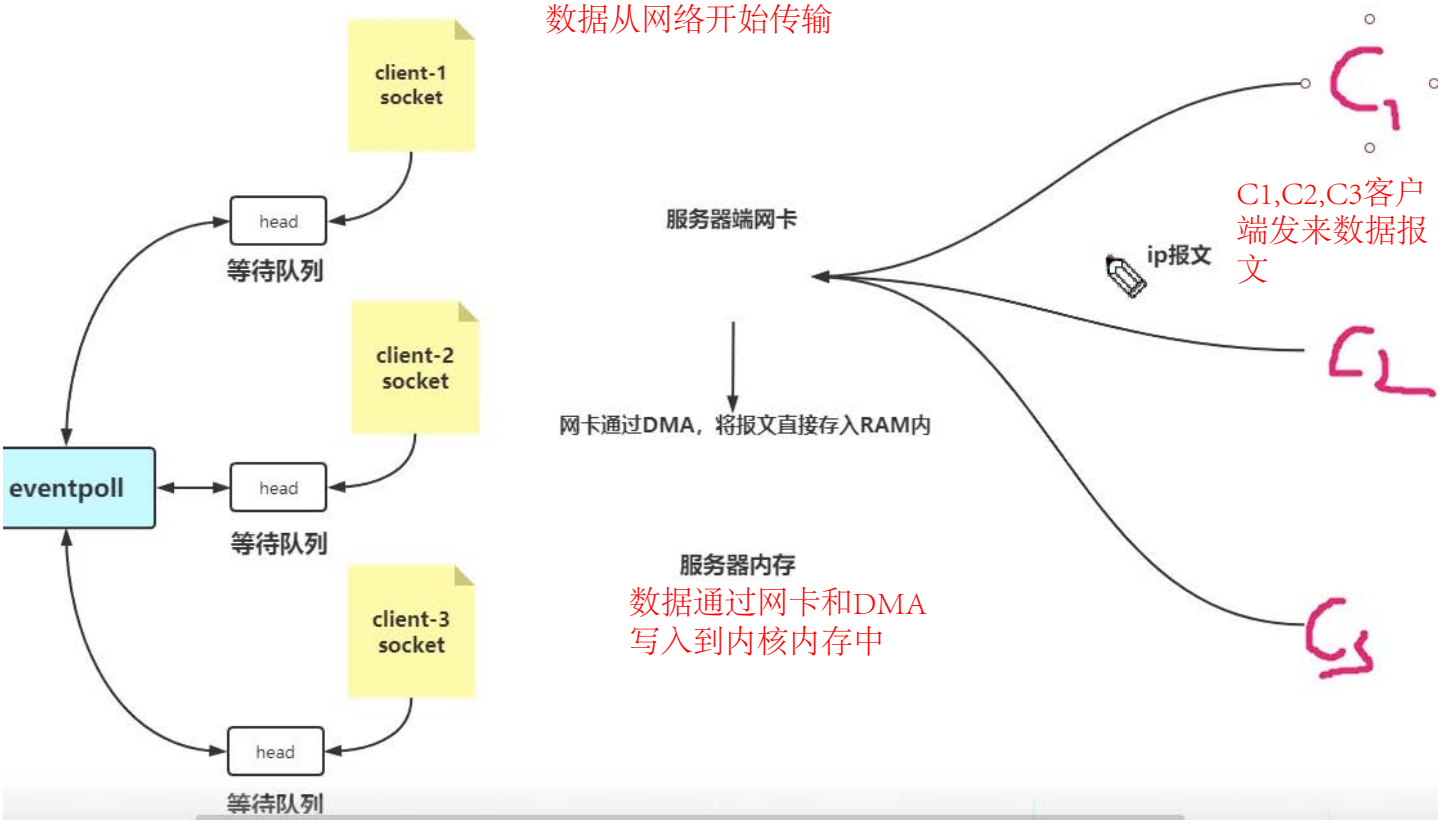
监听事件列表

关心的event事件通过epoll_ctl
注册到红黑树监听列表中

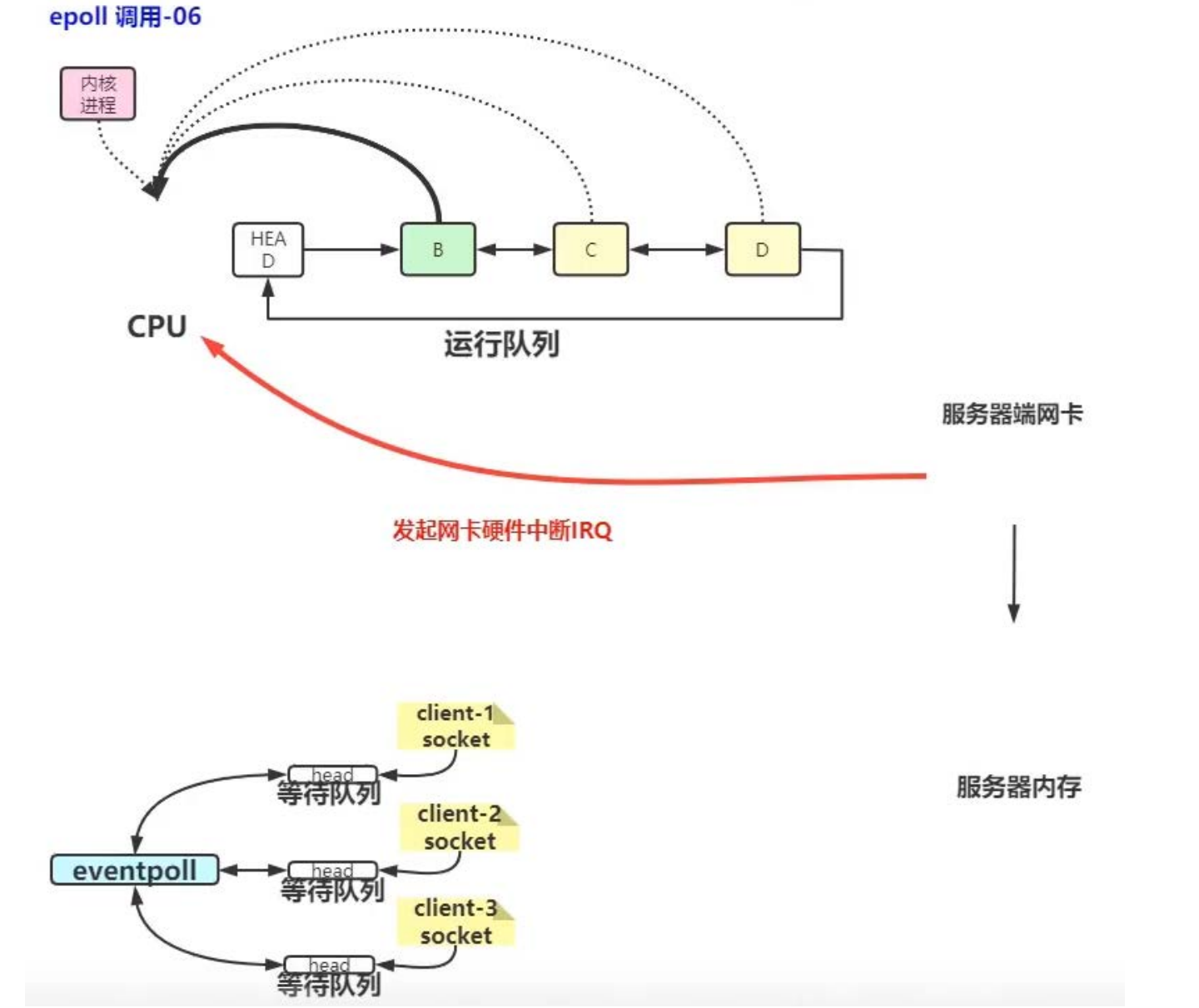

epoll 调用-04
epoll_ctl 将进程关心的套接字事件 加入到eventpoll内
client-1 socket
client-2 socket
client-3 socket
内核进程
HEAD
A
B
C
D
CPU
运行队列
eventpoll
head
等待队列
4
3
5
head
就绪队列
监听事件列表


epoll 调用-04
执行完：epoll_wait 函数，进程A阻塞挂起..
client-1 socket
client-2 socket
client-3 socket
内核进程
等待队列
HEAD
B
C
D
CPU
运行队列
A
epoll_wait让监听列表中的事件，socket指向eventpoll等待队列
进程A由于内核eventpoll的就绪队列为空，所以进程A阻塞挂起到等待队列
eventpoll
head
A
等待队列
4
3
5
head
就绪队列
监听事件列表
监听文件对应的感兴趣的事件所需要的数据还未准备好

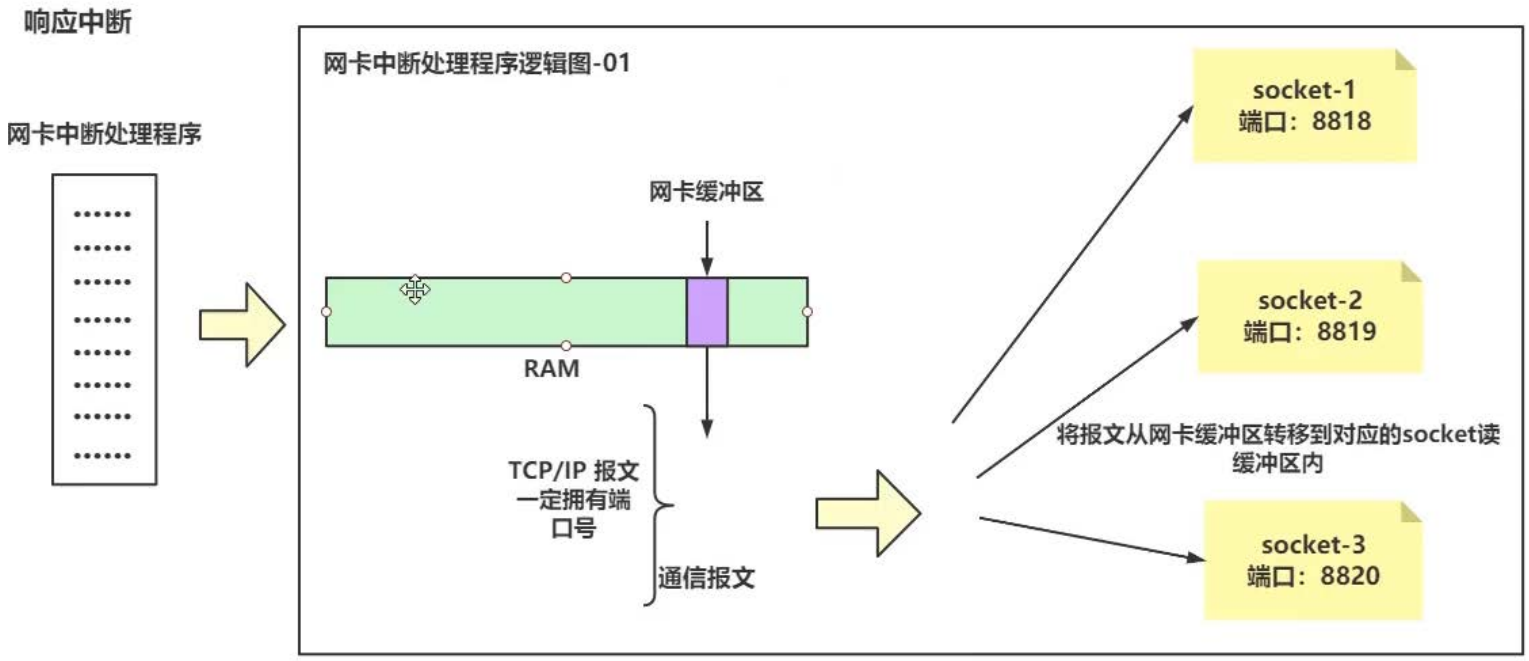
epoll 调用-05

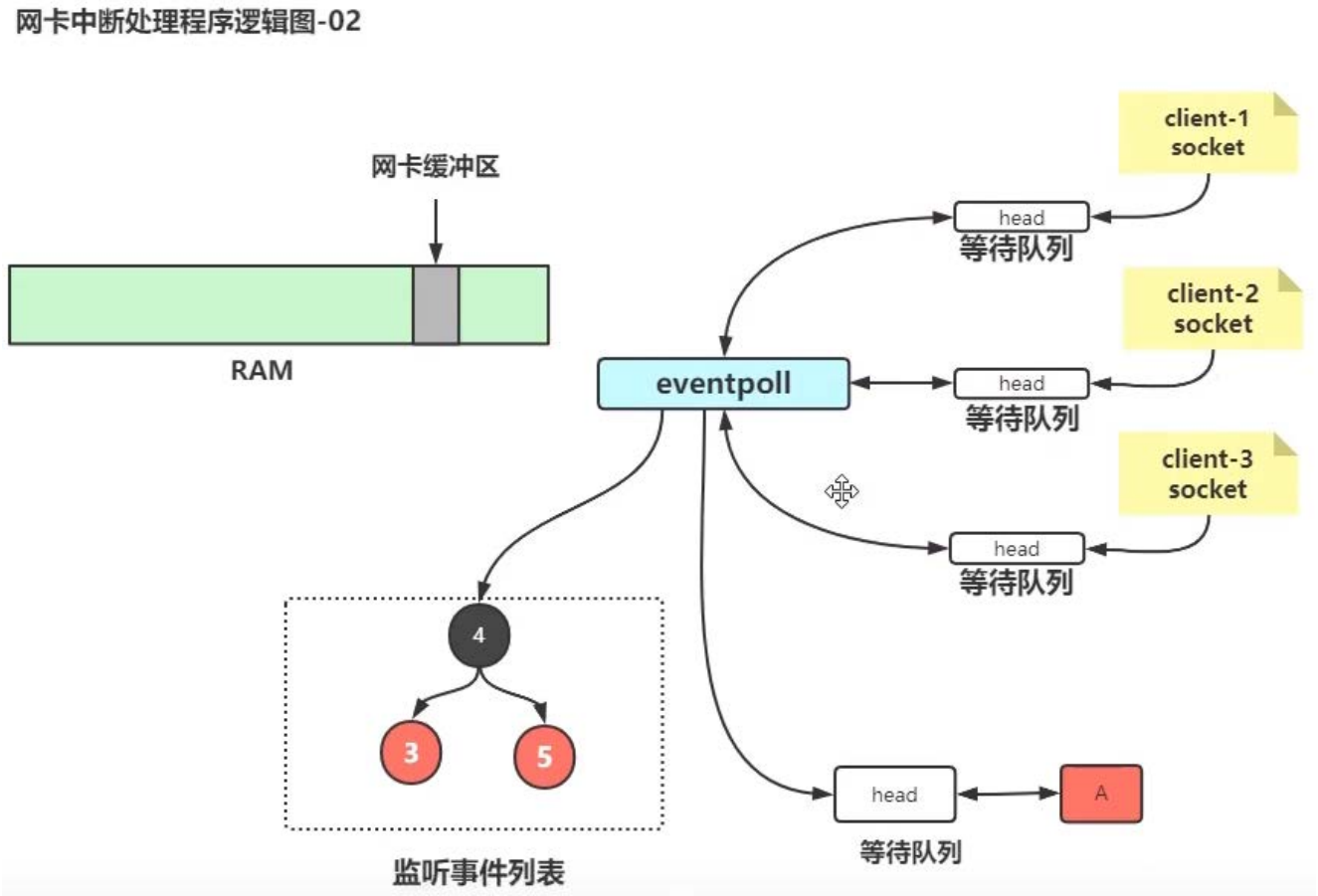


epoll 调用-06

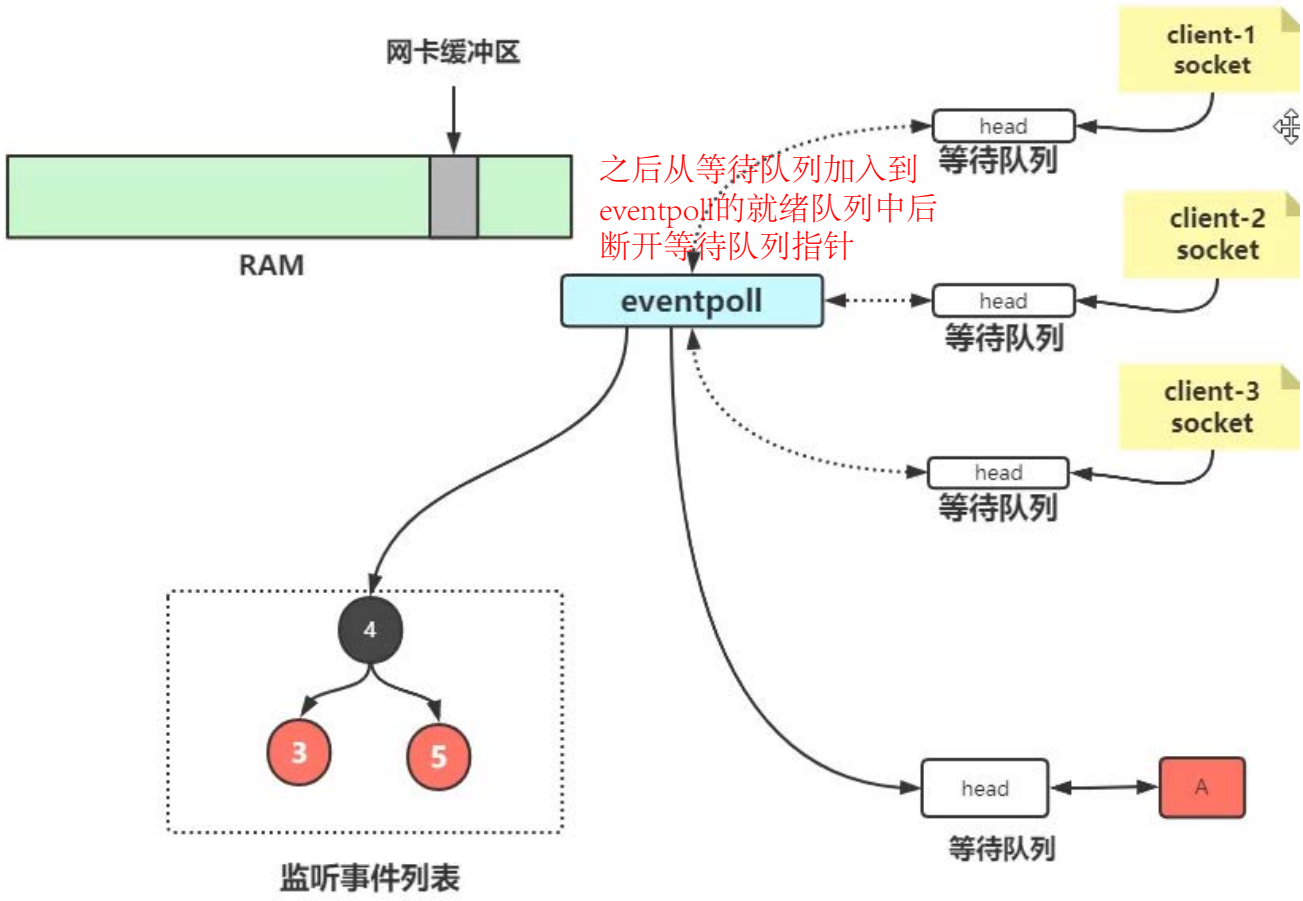


cpu响应网卡的内核中断，并且把网卡中数据写入到内存RAM中

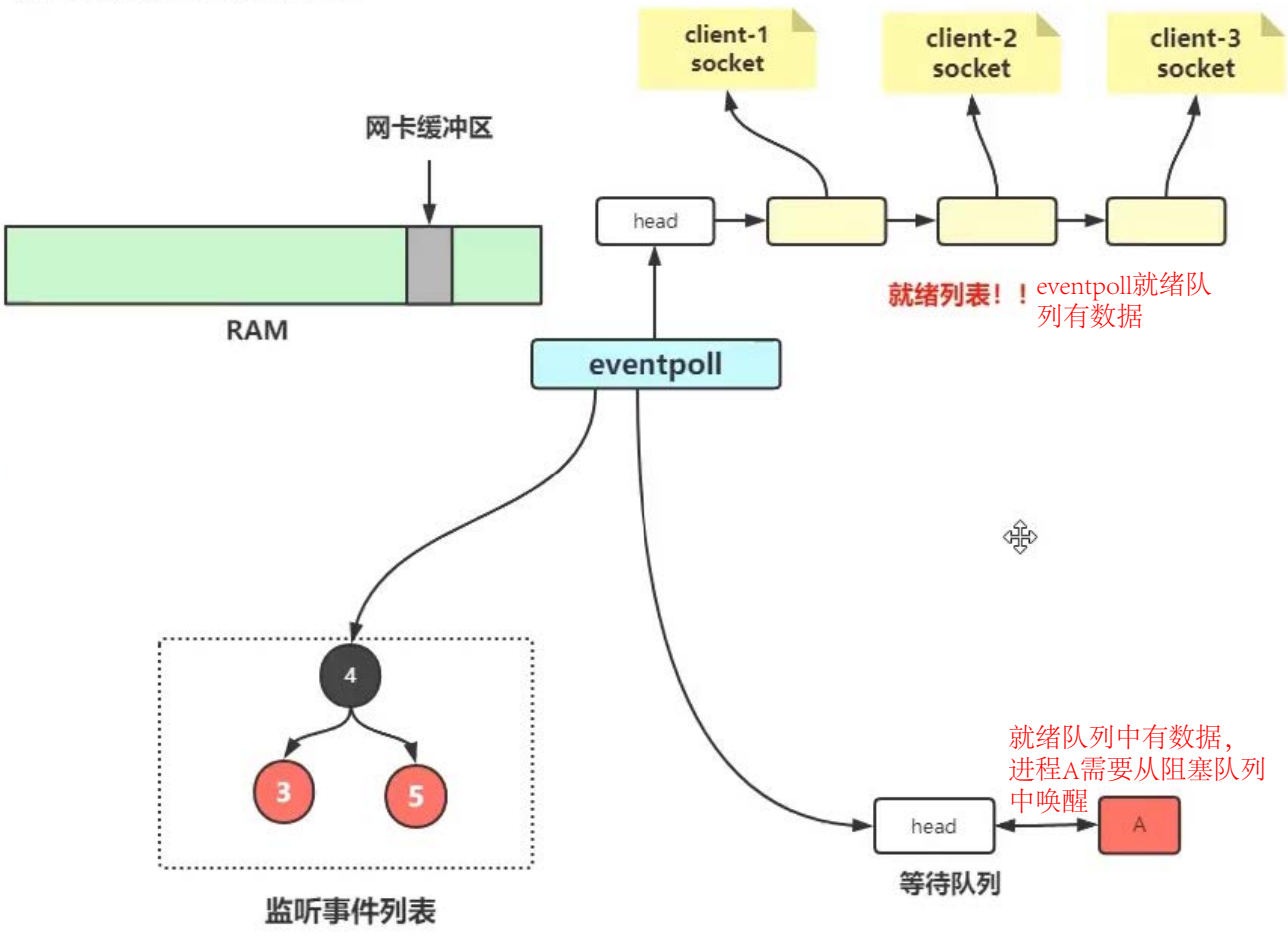


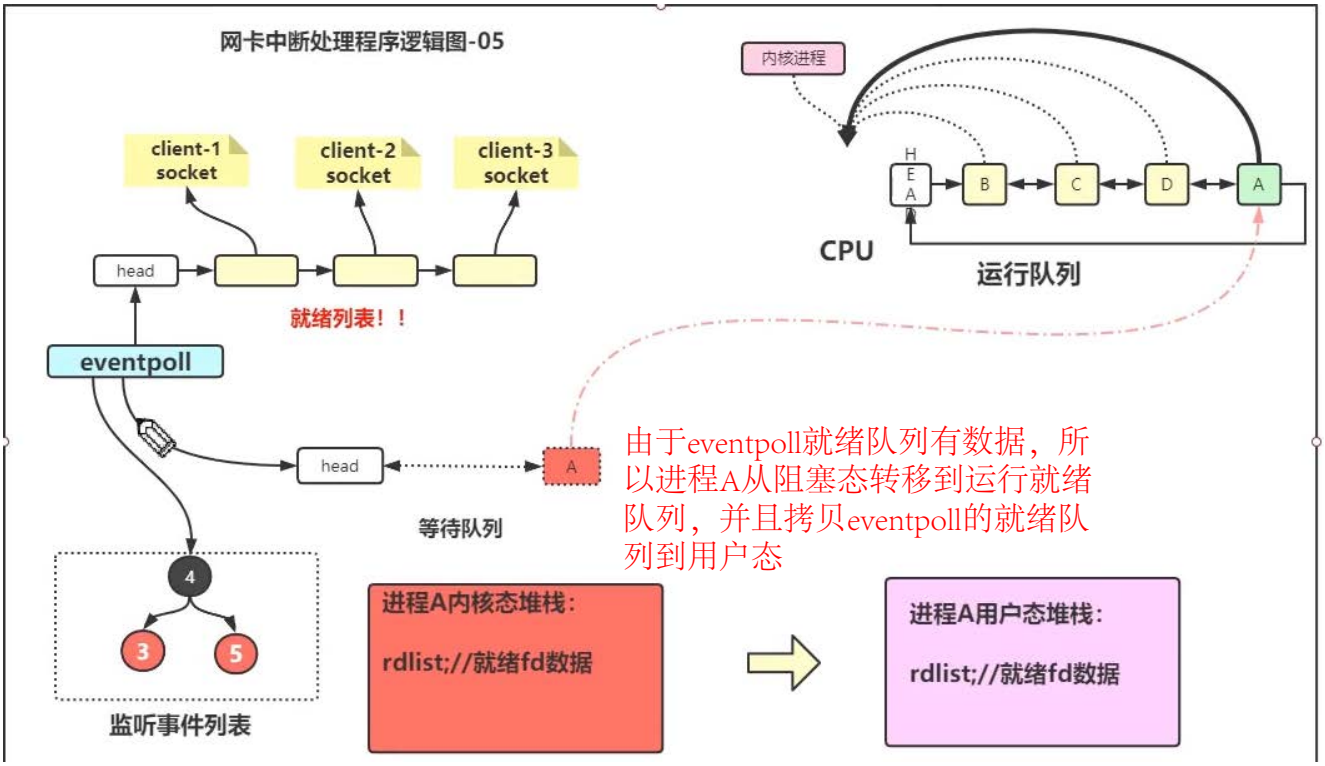


网卡中断处理程序逻辑图-03



网卡中断处理程序逻辑图-04




网卡中断处理程序逻辑图-05
内核进程
client-1 socket
client-2 socket
client-3 socket
head
就绪列表！！
eventpoll
head
A
等待队列
4
3
5
监听事件列表
H
E
A
B
C
D
A
CPU
运行队列
由于eventpoll就绪队列有数据，所以进程A从阻塞态转移到运行就绪队列，并且拷贝eventpoll的就绪队列到用户态
进程A内核态堆栈：
rdlist;//就绪fd数据
进程A用户态堆栈：
rdlist;//就绪fd数据