面试分享专题

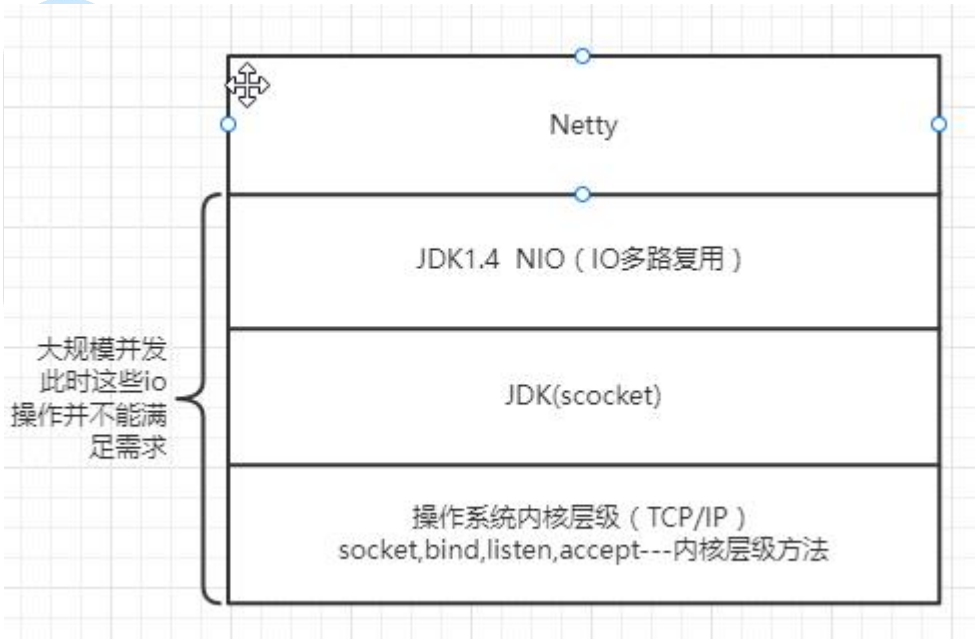
本节课主题：

1）、从操作系统层面分析 BIO
2）、从操作系统层面分析 NIO
3）、从操作内核的角度分析 IO 多路复用
4）、零拷贝知识

1 阻塞式 IO

1.1 IO 层次结构

从技术的发展角度看 io 层次结构： 内核层级， JDK（socket）, JDK NIO , Netty



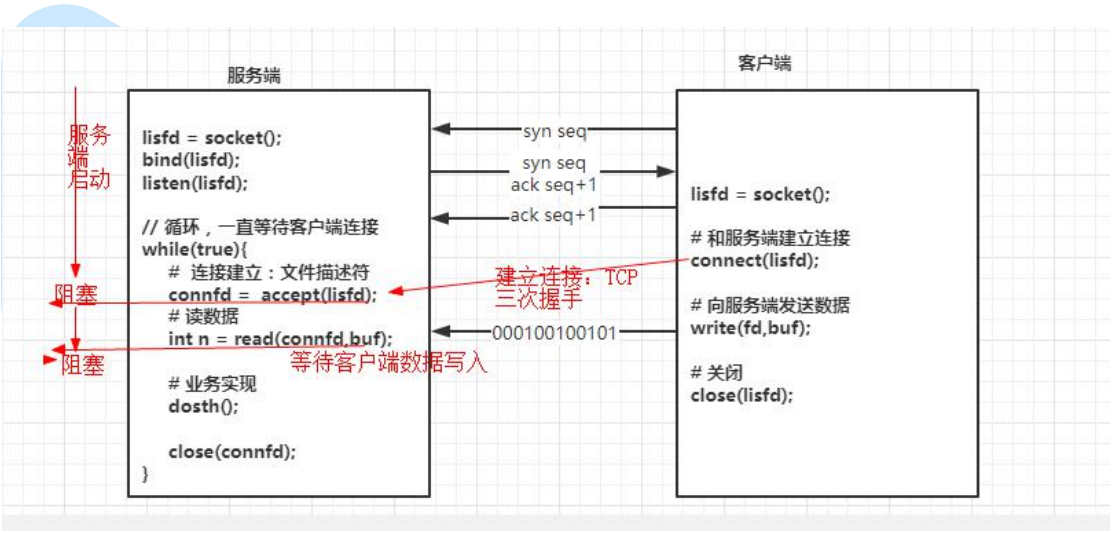
1.2 内核函数

这些函数都是 linux kernel 内核中函数，用于实现相关 io 的操作；

```
内核函数：
# 打开一个网络通信端口
lisfd = socket() ;
# 绑定
bind(lisfd);
# 监听
listen(lisfd)
# 等待客户端建立连接
accept(fd)
#读函数
read()
# 写操作
write
# 客户端和服务建立连接函数
connect(host,port)

# io多路复用函数
select
epoll_create
epoll_ctl
epoll_wait
```

1.3 IO 连接过程

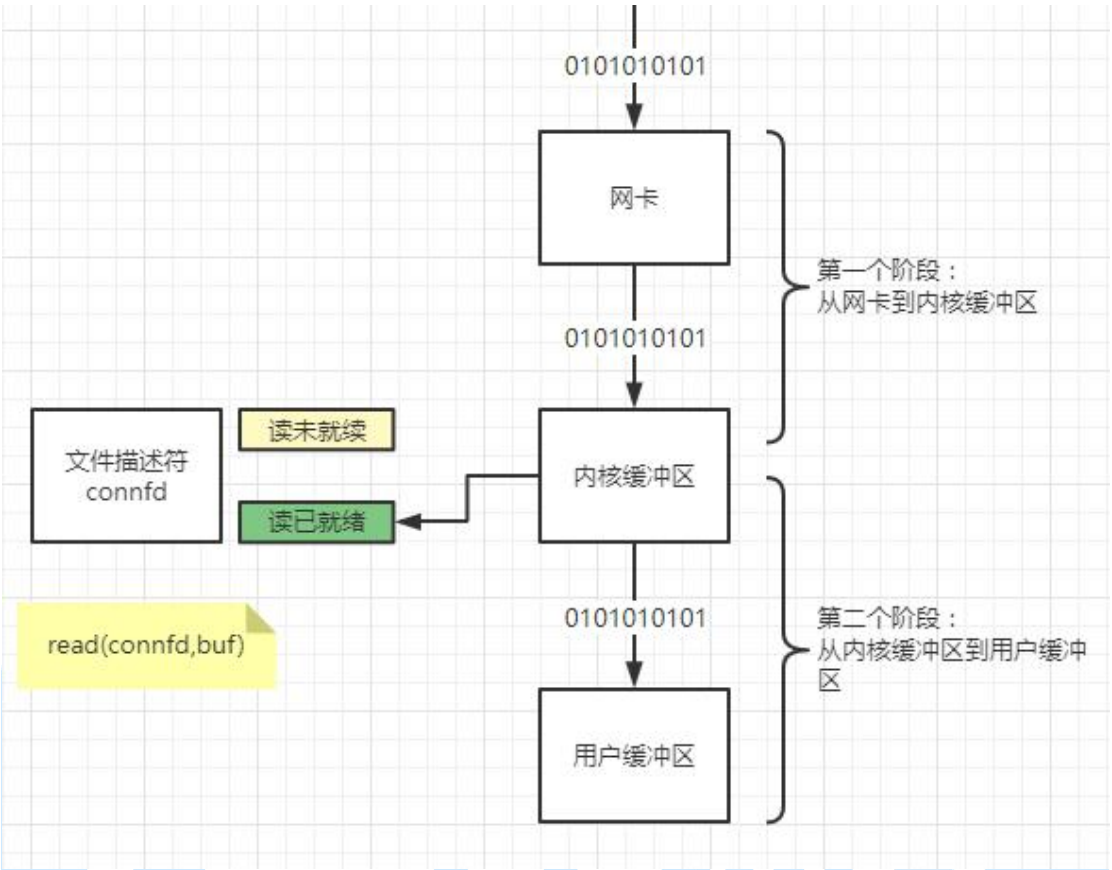


实现 io 的过程，发现 io 阻塞的位置：

1）第一个位置：accept 等待客户端建立连接的位置

2）第二个位置：read 等待客户端写入数据（如果客户端一直不写数据，服务端将会一直阻塞，其他连接连接也不能完成数据处理）

1.4 Read 内核



从内核来看，数据读操作其实阻塞在 2 个阶段：

1）、数据从网卡拷贝到内核缓冲区这个阶段

2）、文件描述符变成读已就绪，read 方法就可以阻塞的方式从内核态把数据读到用户缓冲区；

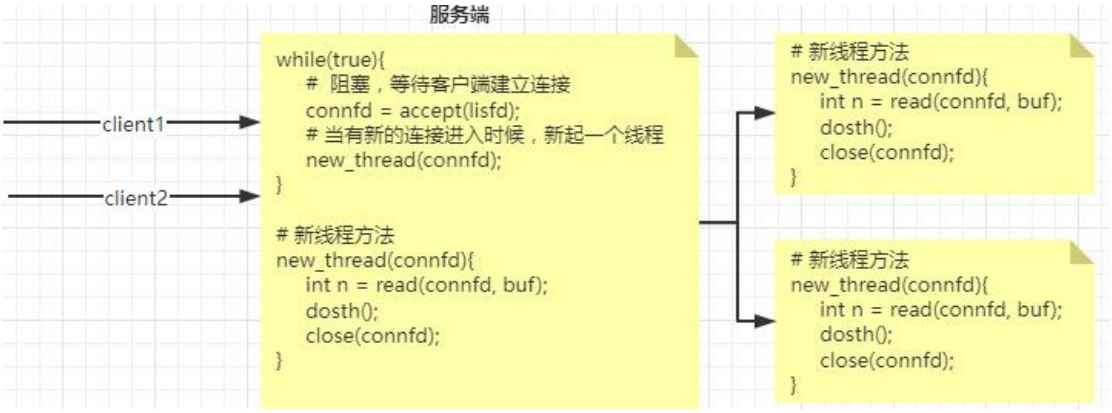
2 NIO

阻塞式 io 性能非常差，服务端每次只允许一个客户端建立建立，支持并发能力非常差，另外：read 读数据的时候，如果客户端一直不发数据，导致其他客户端也无法进一步处理；

因此必须对这个阻塞式 io 做出一些优化，提升 io 的性能，你会如何考虑 io 呢？？？

2.1 多线程改造

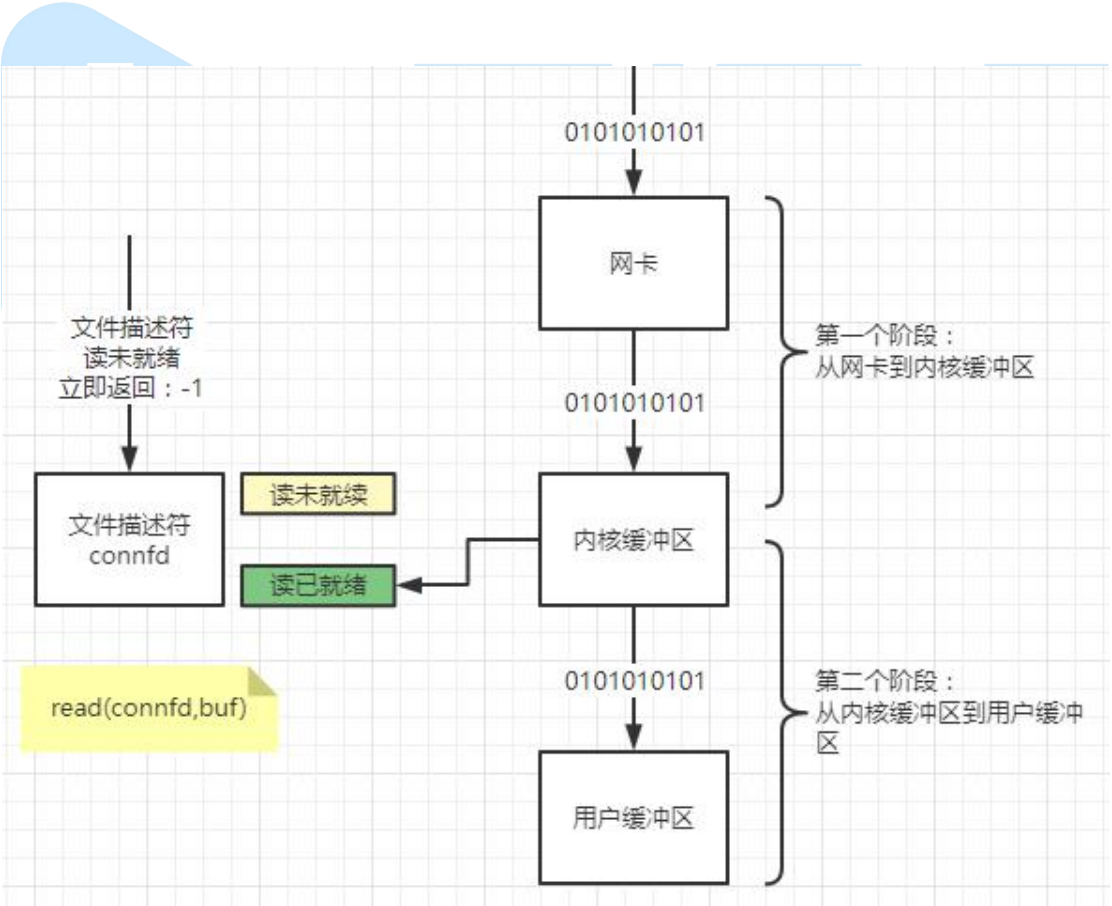
经过改造后，服务的 io 处理方式初步具有的并发的处理能力，不会向上面 bio **卡在 read 方法位置；**



上述改造不是非阻塞 io，因为 Read 方法其实还是阻塞的；只是对这个代码做了多线程改造，使得可以子在服务端使用多线程的方式实现数据的读；

因此上述的改造方式还不是真正的非阻塞的 IO；只不过是我们在代码层面（用户层面）自己实现的功能；我们应该把这个过程交给操作系统内核函数去实现，然后在代码层面实现调用即可；让操作系统提供一个非阻塞的 read 方法；

2.2 NIO 内核



在内核函数 Read 方法实现非阻塞 IO 的时候，当数据没有到达内核缓冲区，此时客户端读取数据返回-1，而不是阻塞的等待；从而从内核层面实现了非阻塞 io；

温馨提示： 非阻塞 IO 在数据从网卡拷贝到内核缓冲区这个阶段是非阻塞的，但是当数据到达内核缓冲区后，此时 read 方法还是需要阻塞，从内核缓冲区读取数据到用户缓冲区；

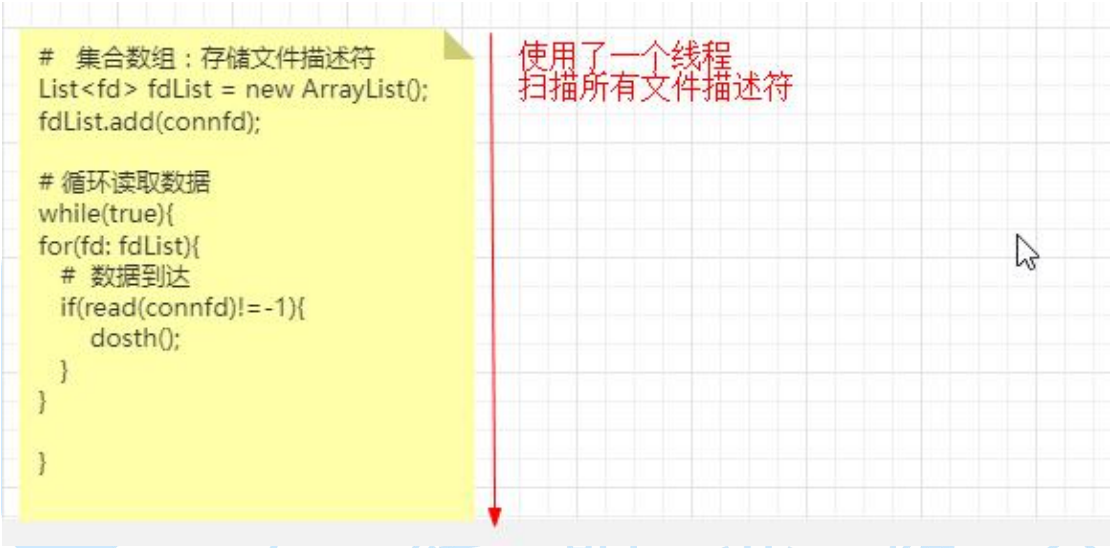
3 IO 多路复用

思考：BIO,NIO 存在什么问题？？

BIO,NIO 在处理客户端连接时候，会为每一个客户端建立一个连接（创建一个线程），很容易服务器线程资源的耗尽，从而导致整个性能上不去；

3.1 一个线程

解决方案：每一个连接都有一个文件描述符，把每一个连接的文件描述符放入一个数组；然后去不断遍历这个数组，查看文件描述的状态，如果数据到达，就进行处理；

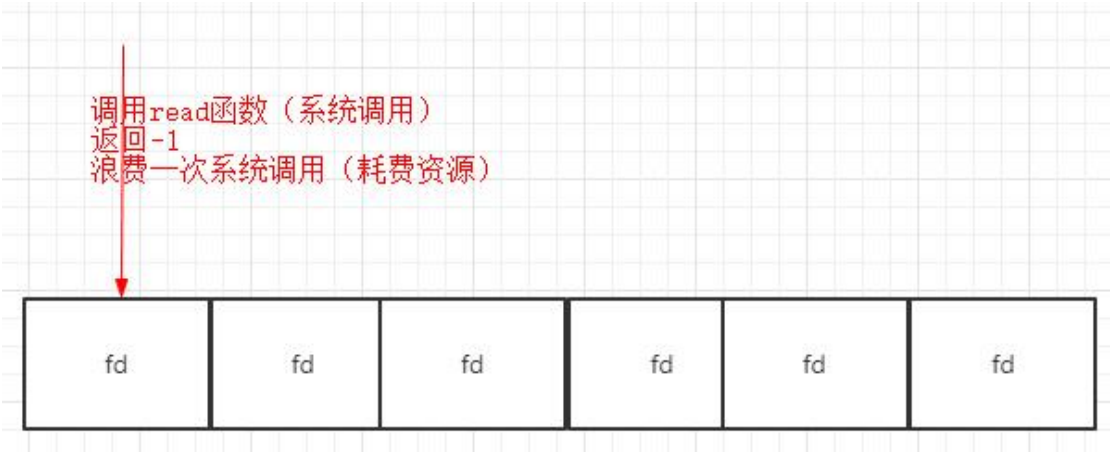



```
#  集合数组：存储文件描述符
List<fd> fdList = new ArrayList();
fdList.add(connfd);

# 循环读取数据
while(true){
for(fd: fdList){
  #  数据到达
  if(read(connfd)!=-1){
     dosth();
  }
}

}
```

上述思想：就是 IO 多路复用思想，一个线程处理多个客户端 io 连接，一个线程循环处理所有的 io 操作；

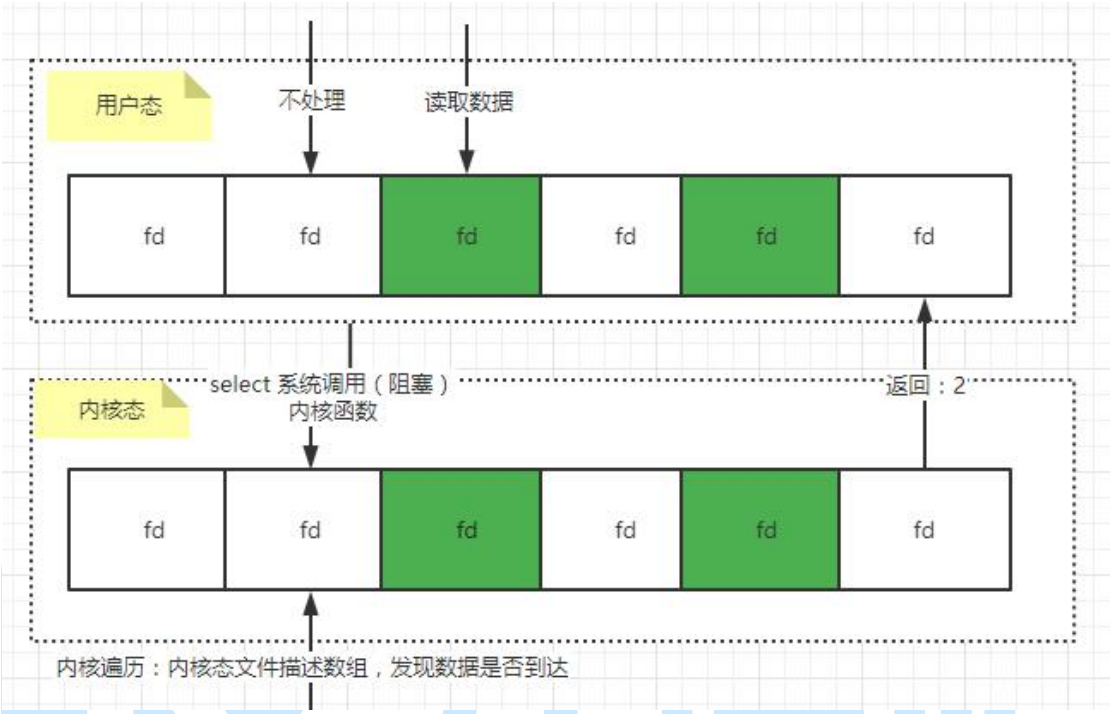


把文件描述符放入数组中，不管这个文件描述的状态是什么，每次都去循环，每次都会发生系统内核函数的调用，存在大量的资源浪费；

解决方案：把循环遍历的工作交给操作系统去完成，就不会发生大量的系统调用；

3.2 Select 函数

Select 函数就是操作系统提供内核函数，通过这个函数就可以实现把文件描述数组交给操作系统内核去遍历；以此来确定那个文件可以读写；



Select 方法

```
int select(
    int nfds,
    fd_set *readfds,   #  读文件描述符集合
    fd_set *writefds,   #  写文件描述符集合
    fd_set *exceptfds,   #  异常文件描述符集合
    struct timeval *timeout); #  定时阻塞监控
```

存在问题：

1）、select 函数调用需要从用户态把 fd 数组拷贝一份到内核，高并发场景下，这样的拷贝资源消耗是非常大的；

2）、select 在内核层遍历 fd 数组，是个同步的过程，这里也是非常耗时的，在高并发场景下，这个对于性能来说是不太友好的；

3）、select 函数仅仅返回的 fd 文件描述符个数，以及标识；具有是那个文件描述符有数据，还需要用户态自己的去再扫描一遍；

3.3 Poll

Poll 内核函数和 select 函数是一样的，唯一区别：

1）select 函数监听的文件描述符：1024 个

2）poll 函数没有限制监听的文件描述符的个数；

3.4 Epoll

存在问题：
1）、select 函数调用需要从用户态把 fd 数组拷贝一份到内核，高并发场景下，这样的拷贝资源消耗是非常大的；（优化为：不拷贝）
2）、select 在内核层遍历 fd 数组，是个同步的过程，这里也是非常耗时的，在高并发场景下，这个对于性能来说是不太友好的；（内核层优化异步事件通知模式）
3）、select 函数仅仅返回的 fd 文件描述符个数，以及标识；具有是那个文件描述符有数据，还需要用户态自己的去再扫描一遍；（优化为：返回为具体的文件描述符）

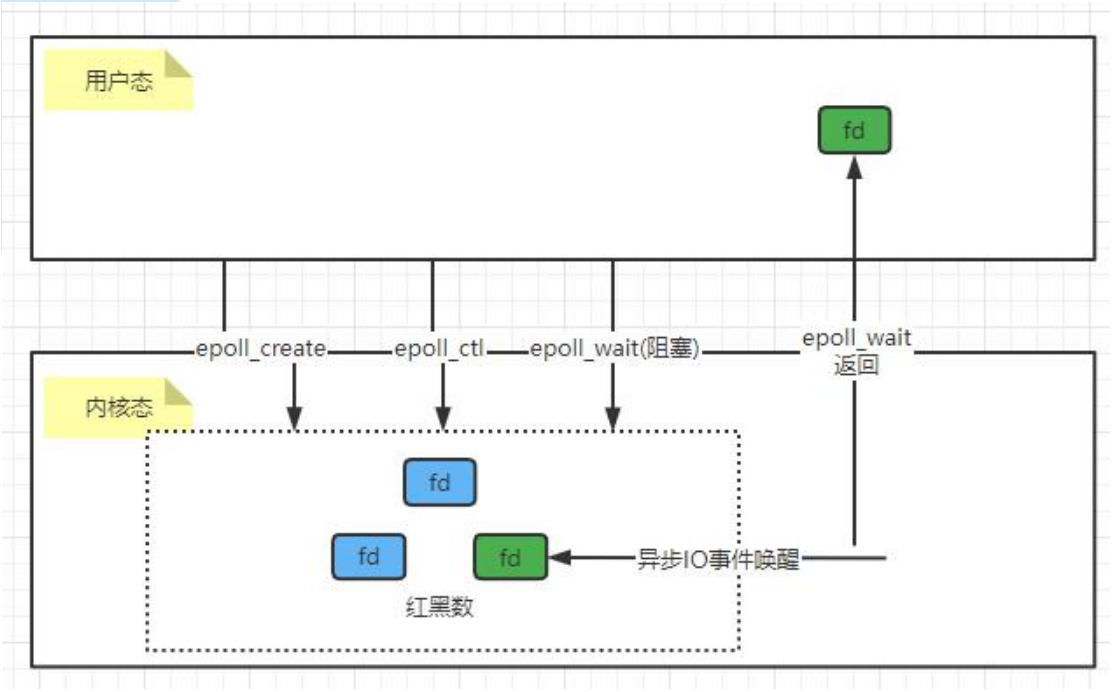
Epoll io 模型提供 3 个函数实现 io 多路复用：

1）、epoll_create：创建一个 epoll 句柄

2）、epoll_ctl：向内核添加，修改，删除要监控的文件描述符

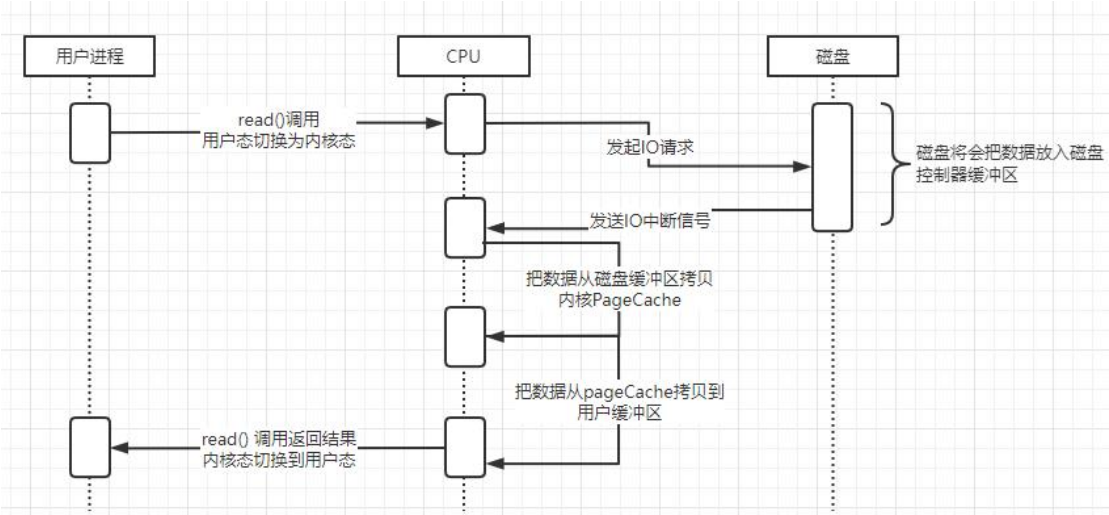
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);

3）、epoll_wait：发起类似 select 调用



4 零拷贝

4.1 传统 io 过程



上述 io 过程中，使用 io 搬运数据，都需要 cpu 参与大量的搬运数据的工作，如果此时有大量的数据需要搬运，cpu 肯定忙不过来；

4.2 DMA