

# 细谈Select,Poll,Epoll



六尺帐篷 (/u/f8e9b1c246f1)

2017.07.31 11:43 字数 2310 阅读 159 评论 0 喜欢 9

(/u/f8e9b1c246f1)

编辑文章 (/writer#/notebooks/14906121/notes/15205576)

- 阻塞 io 模型 blocking IO
- 非阻塞 io 模型 nonblocking IO
- io多路复用模型 IO multiplexing
- 细谈 io 多路复用技术 select 和poll
- 细谈事件驱动—epoll
- 总结

操作系统在处理io的时候，主要有两个阶段：

- 等待数据传到io设备
- io设备将数据复制到user space

我们一般将上述过程简化理解为：

- 等到数据传到kernel内核space
- kernel内核区域将数据复制到user space（理解为进程或者线程的缓冲区）

而根据这两个阶段而不同的操作方法，就会产生多种io模型，本文只讨论select，poll，epoll，所以只引出三种io模型。

## 阻塞 io 模型 blocking IO

最常用的也就是阻塞io模型。默认情况下，所有文件操作都是阻塞的。我们以套接字接口为例来讲解此模型，在进程空间调用recvfrom，其系统调用知道数据包到达并且被复制到进程缓冲中或者发生错误时才会返回，在此期间会一直阻塞，所以进程在调用recvfrom开始到它返回的整段时间都是阻塞的，因此称之为阻塞io模型。

注意：**在阻塞状态下，程序是不会浪费CPU的**，cpu只是不执行io操作了，还会去做别的。



Figure 6.1. Blocking I/O model.

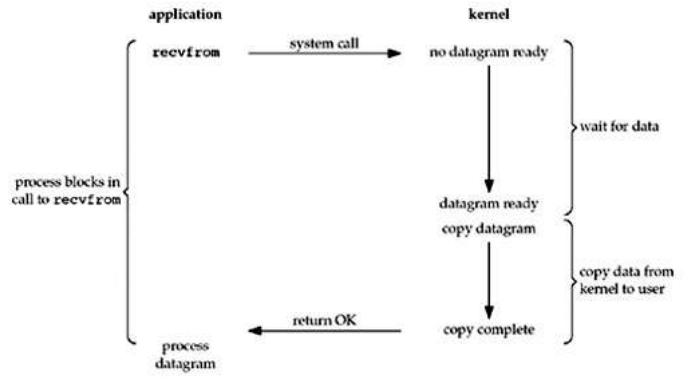


image.png

应用层有数据过来，会调用recvfrom方法，但是这个时候应用层的数据还没复制到kernel中，将应用层数据复制到kernel这个阶段是需要时间的，所以recvfrom方法会阻塞，当内核中的数据准备好之后，recvfrom方法还会不会返回，而是会发起一个系统调用将kernel中的数据复制到进程的缓冲区中，也就是user space，当这个工作完成之后，recvfrom才会返回并解除程序的阻塞。

所以我们总结可以发现，主要就是上面两个阶段

- 应用层数据到kernel
- kernel复制到user space

阻塞io模型就是将这两个过程合并在一起，一起阻塞。而非阻塞模型则是将第一个过程的阻塞变成非阻塞，第二个阶段是系统调用，是必须阻塞的，所以非阻塞模型也是同步的，因为它们在kernel里的数据准备好之后，进行系统调用，将数据拷贝到进程缓冲区中。

非阻塞 io 模型 nonblocking IO

就是对于第一个阶段，也就是应用层数据到kernel的过程中，recvfrom会轮询检查，如果kernel数据没有准备好，就返回一个EWOULDBLOCK错误。不断的轮询检查，直到发现kernel中的数据准备好了，就返回，然后进行系统调用，将数据从kernel拷贝到进程缓冲区中。有点类似busy-waiting的方法。

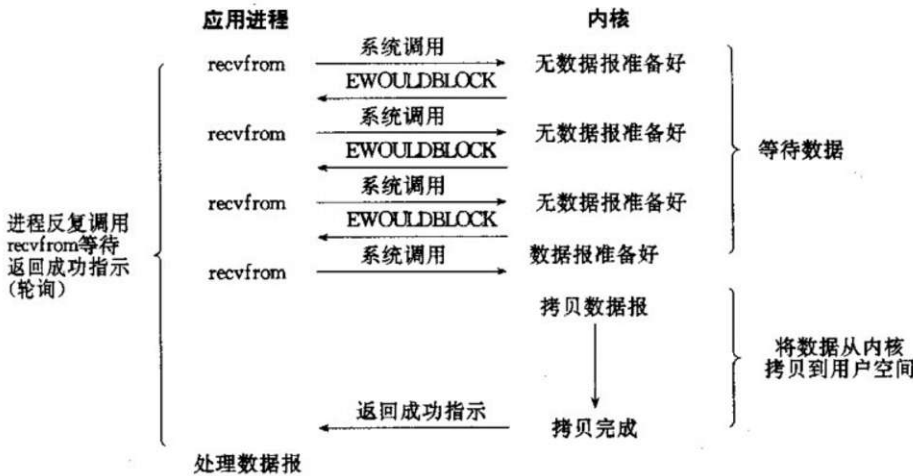


image.png

io多路复用模型 IO multiplexing

- 目的：因为阻塞模型在没有收到数据的时候就会阻塞卡住，如果一次需要接受多个 socket fd的时候，就会导致必须处理完前面的fd，才能处理后面的fd，即使可能后面的fd比前面的fd还要先准备好，所以这样就会造成客户端的严重延迟。为了处理多个请求，我们自然先想到用多线程来处理多个socket fd，但是这样又会启动大量的线程，造成资源的浪费，所以这个时候就出现了io多路复用技术。就是用个进程来处理多个fd的请求。
- 应用：适用于针对大量的io请求的情况，对于服务器必须在同时处理来自客户端的大量的io操作的时候，就非常适合

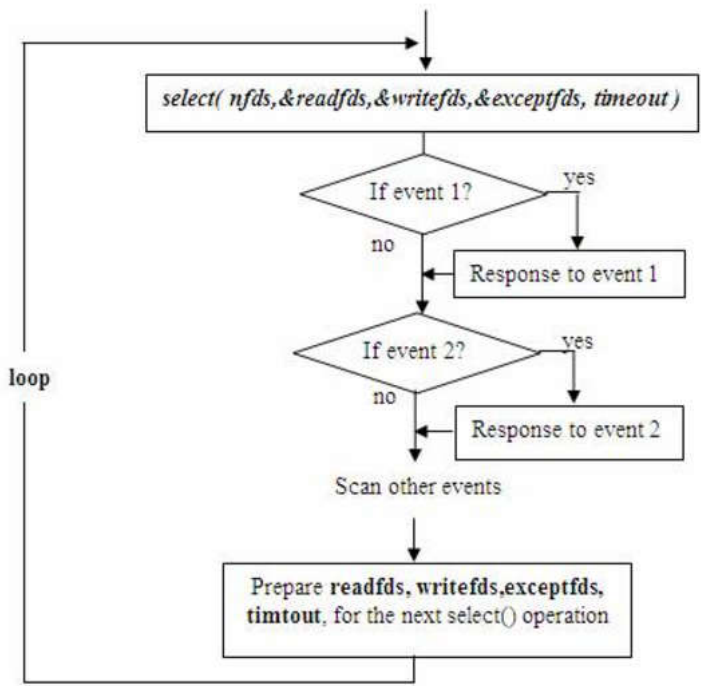


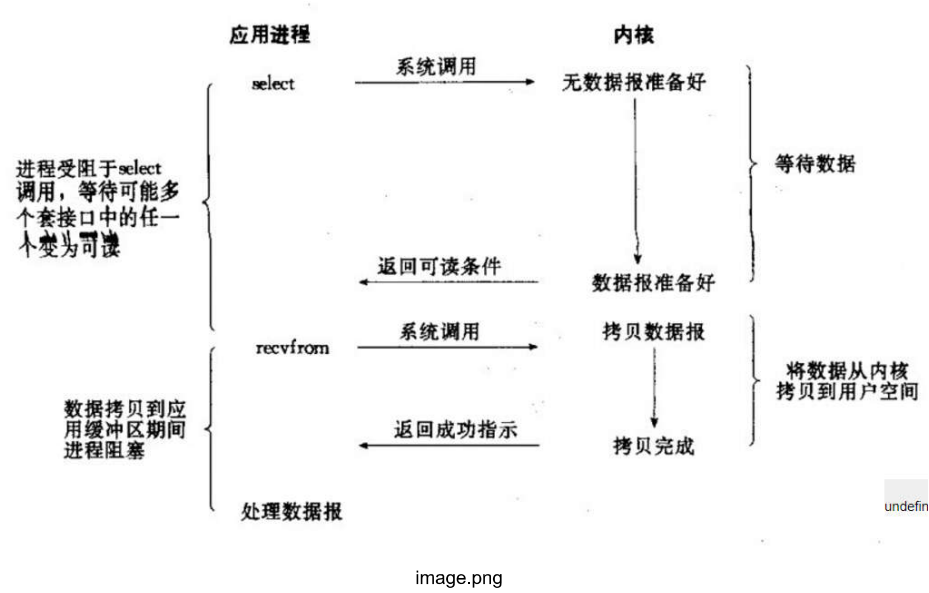
image.png

## 细谈 io 多路复用技术 select 和poll

### select

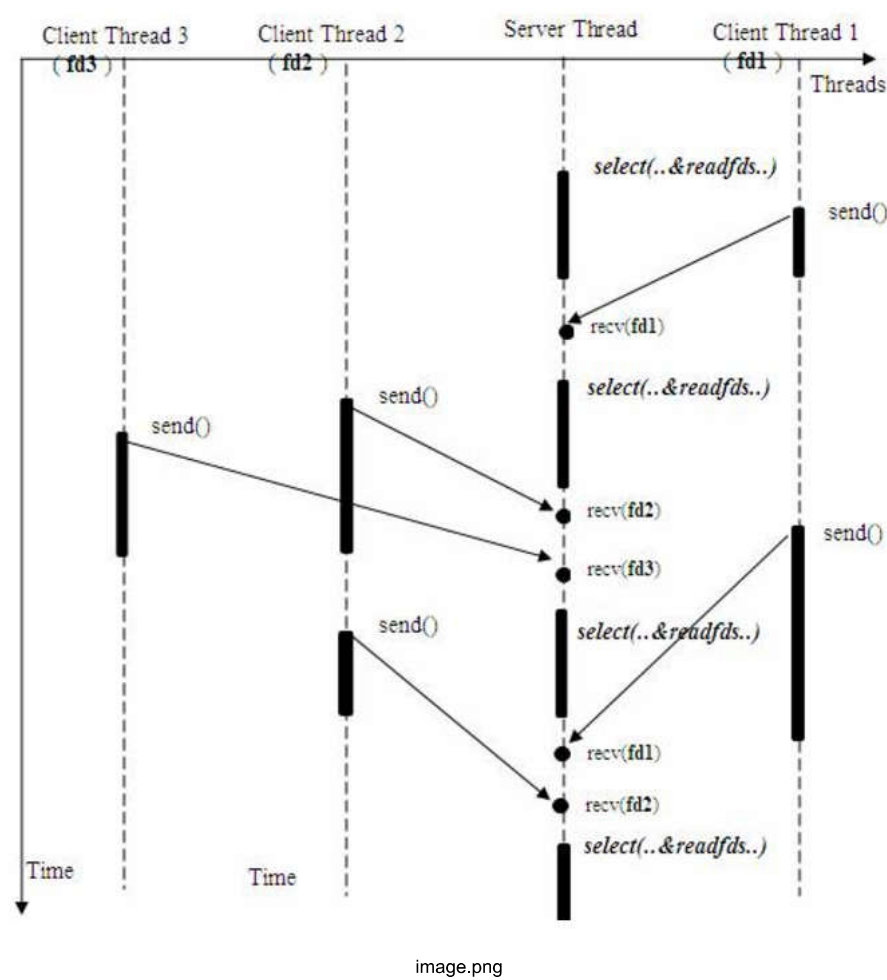
select的工作流程：  
单个进程就可以同时处理多个网络连接的io请求（同时阻塞多个io操作）。基本原理就是程序呼叫select，然后整个程序就阻塞了，这时候，kernel就会轮询检查所有select负责的fd，当找到一个client中的数据准备好了，select就会返回，这个时候程序就会系统调用，将数据从kernel复制到进程缓冲区。





下图为select同时从多个客户端接受数据的过程

虽然服务器进程会被select阻塞，但是select会利用内核不断轮询监听其他客户端的io操作是否完成。



### Poll介绍

poll的原理与select非常相似，差别如下：

- 描述fd集合的方式不同，poll使用 `pollfd` 结构而不是select结构 `fd_set` 结构，所以poll是链式的，没有最大连接数的限制

- poll有一个特点是水平触发，也就是通知程序fd就绪后，这次没有被处理，那么下次poll的时候会再次通知同个fd已经就绪。

## select缺点

- 根据fd\_size的定义，它的大小为32个整数大小（32位机器为32\*32，所有共有1024bits可以记录fd），每个fd一个bit，所以最大只能同时处理1024个fd
- 每次要判断【有哪些event发生】这件事的成本很高，因为select（polling也是）采取主动轮询机制

1.每一次呼叫 select( ) 都需要先从 user space把 FD\_SET复制到 kernel（约线性时间成本）

为什么 select 不能像epoll一样，只做一次复制就好呢？

每一次呼叫 select()前，FD\_SET都可能更动，而 epoll 提供了共享记忆存储结构，所以不需要有 kernel 與 user之间的数据沟通

2.然后kernel还要轮询每个fd，约线性时间

- 假设现实中，有1百万个客户端同时与一个服务器保持着tcp连接，而每一个时刻，通常只有几百上千个tcp连接是活跃的，这时候我们仍然使用select/poll机制，kernel必须在搜寻完100万个fd之后，才能找到其中状态是active的，这样资源消耗大而且效率低下。

对于select和poll的上述缺点，就引进了一种新的技术，epoll技术

## 细谈事件驱动--epoll

epoll 提供了三个函数：

- int epoll\_create(int size);  
建立一个 epoll 对象，并传回它的id
- int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event);  
事件注册函数，将需要监听的事件和需要监听的fd交给epoll对象
- int epoll\_wait(int epfd, struct epoll\_event \*events, int maxevents, int timeout);  
等待注册的事件被触发或者timeout发生

epoll解决的问题：

- epoll没有fd数量限制  
epoll没有这个限制，我们知道每个epoll监听一个fd，所以最大数量与能打开的fd数量有关，一个g的内存的机器上，能打开10万个左右
- epoll不需要每次都从user space 将fd set复制到内核kernel  
epoll在用epoll\_ctl函数进行事件注册的时候，已经将fd复制到内核中，所以不需要每次都重新复制一次
- select 和 poll 都是主動輪詢機制，需要拜訪每一個 FD；  
epoll是被动触发方式，给fd注册了相应事件的时候，我们为每一个fd指定了一个回调函数，当数据准备好之后，就会把就绪的fd加入一个就绪的队列中，epoll\_wait的工作方式实际上就是在这个就绪队列中查看有没有就绪的fd，如果有，就唤醒就绪队列上的等待者，然后调用回调函数。



- 虽然epoll。poll。epoll都需要查看是否有fd就绪，但是epoll之所以是被动触发，就在于它只要去查找就绪队列中有没有fd，就绪的fd是主动加到队列中，epoll不需要一个个轮询确认。  
换一句话讲，就是select和poll只能通知有fd已经就绪了，但不能知道究竟是哪个fd就绪，所以select和poll就要去主动轮询一遍找到就绪的fd。而epoll则是不但可以知道有fd可以就绪，而且还具体可以知道就绪fd的编号，所以直接找到就可以，不用轮询。

总结

- select, poll是为了解决同时大量IO的情况（尤其网络服务器），但是随着连接数越多，性能越差
- epoll是select和poll的改进方案，在 linux 上可以取代 select 和 poll，可以处理大量连接的性能问题

Java NIO (/nb/14906121)


© 著作权归作者所有



六尺帐篷 (/u/f8e9b1c246f1)

写了 245418 字，被 15240 人关注，获得了 1429 个喜欢 (/u/f8e9b1c246f1)

喜欢 9





更多分享


(<http://cwb.assets.jianshu.io/notes/images/1520557>)

被以下专题收入，发现更多相似内容

投稿管理

- + 收入我的专题
- 

Android知识 (/c/3fde3b545a35?utm\_source=desktop&utm\_medium=notes-included-collection)
- 

Android... (/c/5139d555c94d?utm\_source=desktop&utm\_medium=notes-included-collection)
- 

程序员 (/c/NEt52a?utm\_source=desktop&utm\_medium=notes-included-collection)





