高级I/O模型(1)

基本IO概念

- 阻塞与非阻塞
  - 在一些常见I/O系统调用中，如read，accept，当数据未准备完全时，进程将陷入阻塞状态，内核调度程序转而调度其它进程 ⊙ 从内核的角度上来看，这一行为非常合理，毕竟内核的主要工作之一就是充分利用硬件，使得在一段时间内能够运行更多的进程
  - 当然，I/O系统调用大多也提供了非阻塞模式的调用，即系统调用要么立刻完成，要么在数据未准备充分的情况下返回错误
    - 当返回错误时，应用程序通常有两个选择
      - 高频率轮询 ⊙ 过于浪费CPU这一宝贵的资源
      - 低频率轮询 ⊙ 无法第一时间获取到IO事件，中间会有延迟
- 同步与异步
  - 同步是指当进行IO系统调用时，必须等待IO操作的结果，才能继续执行
  - 异步是指当进行IO系统调用时，无需等待IO操作的结果，程序可转而执行其它系统调用，如多线程，多进程方式或者协程

IO模型概览

- 在一般的应用程序中，通常只会涉及到少量的文件描述符读写操作，在这类程序中，IO不是主要的性能瓶颈，所以采用同步阻塞的IO方式即能满足需求 ⊙ 但是在诸如Web服务器、数据库系统等应用程序而言，通常会同时处理成千上万的IO操作，此时必须要采用其它IO模型
- 常见I/O模型
  - 多进程或多线程 ⊙ 采用阻塞的方式进行IO调用，但是执行IO的主体位于另外的进程或者线程中
    - 多进程执行阻塞的I/O调用过于奢侈，原因在于创建以及运行一个进程的代价过于昂贵，也不可能创建出几万个I/O进程 ⊙ 进程上下文切换代价不菲，单个进程所占用系统资源不菲
    - 多线程执行阻塞I/O相比与多进程来说对资源占用更少，线程切换也更加快速，但是与进程存在着同样的限制: 无法创建出几万个线程
  - I/O多路复用 ⊙ 原理在于将所有需要处理的文件描述符置于一个容器中，同时检查这些文件描述符并找出能够执行I/O操作的文件描述符，并返回给用户 ⊙ 诸如Redis、Nginx等高性能网络工具均采用epoll作为Socket IO模型
  - 信号驱动I/O ⊙ 当某个文件描述符可读或者可写时，内核向请求数据的进程发送SIGIO信号
  - 异步I/O ⊙ 当进程发起异步I/O调用后立即返回，无需等待数据准备以及数据在内核与用户空间的拷贝。当IO操作完全完成时，内核将通知进程，此时，数据已经位于用户空间中 ⊙ glibc目前采用多线程的方式实现异步I/O
- 水平触发与边缘触发
  - 水平触发通知 ⊙ 若文件描述符上可以非阻塞地执行I/O系统调用，就认为它已经就绪
  - 边缘触发通知 ⊙ 如果文件描述符自上次状态检查以来有了新的I/O活动，则触发通知
- 各IO模型与触发机制支持 ⊙

I/O模式	水平触发	边缘触发
select, poll	○	
信号驱动I/O		○
epoll	○	○
异步I/O	○	

⊙ epoll是唯一支持水平触发与边缘触发的I/O模型，但是受限于Linux平台

select

- *原型: int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);* ⊙ 返回已就绪的文件描述符个数，0表示超时，-1表示调用错误
- 文件描述符集合: fd_set
  - 和信号集一样，不同的Linux版本可能采用不同的实现，通常是位掩码，也可能是数组，所以使用fd_set类型进行封装
  - 基本操作
    - void FD_ZERO(fd_set *fdset); ⊙ 必须使用FD_ZERO初始化一个描述符集，如同sigemptyset一样
    - void FD_SET(int fd, fd_set *fdset); ⊙ 将文件描述符fd添加至fdset所指向的集合中
    - void FD_CLR(int fd, fd_set *fdset); ⊙ 将文件描述符fd从fdset所指向的集合中移除
    - int FD_ISSET(int fd, fd_set *fdset); ⊙ 判断文件秒描述符fd是否是fdset中的成员，返回0或者1
      - 当select调用返回时，可用此函数判断描述符fd是否准备就绪
  - fd_set存在最大容量限制 ⊙ 文件描述符集合有一个最大容量限制，由常量FD_SETSIZE决定，通常为1024 ⊙ glibc定义这个值的目的在于告知用户: 使用select()同时检查的文件描述符数量最大值为1024，若多于该值，请使用其它模型，如epoll
- 最大文件描述符号: nfds ⊙ nfds是一个很有意思的参数，参数值必须设置为比3个文件描述符集合中所包含的最大文件描述符还要大1 ⊙ 假设在3个文件描述符集合中，最大的文件描述符号为2659，则nfds的值应为2660
- 返回值 ⊙ 当3个文件描述符集合内没有可读、可写或者是异常文件描述符时，select()调用将一直阻塞，直到至少一个文件描述符变为就绪态
  - 需要注意的是，调用将会返回已经就绪的文件描述符个数，但是具体是哪些文件描述符，在哪些文件描述符集合中，需要用户迭代判断
  - 对所有已经打开的文件描述符数组进行迭代，并使用FD_ISSET判断是否有可读、可写或者是异常事件
  - readfds、writefds等文件描述符集合，既是传入所需要监控的文件描述符，同时也是当文件描述符就绪时的结果表示
- select()调用最早出现在BSD系统的套接字API中，而不是文件读取中。事实上，对一个文件来说，文件描述符总是返回已准备就绪: 文件读写一定会在有限时间内返回

poll

- 系统调用poll()执行的任务与select()非常相似，两者间主要的区别就在于如何指定待检查的文件描述符
- *原型: int poll(struct pollfd fds[], nfds_t nfds, int timeout);* ⊙ 返回已就绪的文件描述符个数，0表示超时，-1表示调用错误
- 结构 pollfd ⊙

```
strutc poolfd {
    int   fd;         /* 文件描述符 */
    short events;     /* 指定fd感兴趣的事件 */
    short revents;    /* poll()返回时，表示fd上实际发生的事件 */
}
```

  - POLLIN ⊙ 可读
  - POLLOUT ⊙ 可写
- 参数timeout
  - 等于-1 ⊙ poll()调用会一值阻塞，直到fds中有一个文件描述符达到就绪态
  - 等于0 ⊙ poll()调用不会阻塞，只是执行一次检查，看看哪些文件描述符处于就绪状态
  - 大于0 ⊙ poll()至多阻塞timeout毫秒
- 文件描述符何时就绪问题
  - 文件I/O
    - 尽管当内核cache没有用户所需要的数据时，将由DMA对磁盘发出数据读取操作，此时内核将调度其它进程，发起IO将会阻塞。但是，文件I/O操作将会在有限的时间内返回，不会永久的阻塞
    - 所以，不管是poll还是select，对硬盘文件的描述符检查总是返回就绪
  - 管道、套接字I/O ⊙ 当对socket描述符进行读操作时，如果网络的发送方未发送任何数据，那么可能会永久阻塞。当写入数据时，若发送缓冲区已满，且不能及时的将缓冲区内数据清空的话，写入操作也有可能长时间阻塞

虽然表现都是阻塞，但是仍有差别