网络通信

概念补充

事件处理的设计模式

> 一般地,I/O多路复用机制都依赖于一个事件多路分离器(Event Demultiplexer)。分离器对象可将来自事件源的I/O事件分离出来，并分发到对应的read/write事件处理器(Event Handler)。开发人员预先注册需要处理的事件及其事件处理器（或回调函数）；事件分离器负责将请求事件传递给事件处理器。

- proactor
  异步IO，处理器--或者兼任处理器的事件分离器，只负责发起异步读写操作。IO操作本身由操作系统来完成
- reactor
  同步IO，事件分离器负责等待文件描述符或socket为读写操作准备就绪，然后将就绪事件传递给对应的处理器，最后由处理器负责完成实际的读写工作

Reactor框架中用户定义的操作是在实际操作之前调用的。比如你定义了操作是要向一个SOCKET写数据，那么当该SOCKET可以接收数据的时候，你的操作就会被调用；而Proactor框架中用户定义的操作是在实际操作之后调用的。比如你定义了一个操作要显示从SOCKET中读入的数据，那么当读操作完成以后，你的操作才会被调用。

IO模式

> Linux IO模式及 select、poll、epoll详解
> 以下为简要介绍，详细内容参照上连接的博客，博客底部的参考文献有更细节的描述和样例代码

I/O多路复用就是通过一种机制，一个进程可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。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
  select 函数监视的文件描述符分3类，分别是writefds、readfds、和exceptfds。调用后select函数会阻塞，直到有描述副就绪（有数据 可读、可写、或者有except），或者超时（timeout指定等待时间，如果立即返回设为null即可），函数返回。当select函数返回后，可以 通过遍历fdset，来找到就绪的描述符。
  select目前几乎在所有的平台上支持，其良好跨平台支持也是它的一个优点。select的一个缺点在于单个进程能够监视的文件描述符的数量存在最大限制，在Linux上一般为1024，可以通过修改宏定义甚至重新编译内核的方式提升这一限制，但是这样也会造成效率的降低。
- poll
  不同与select使用三个位图来表示三个fdset的方式，poll使用一个 pollfd的指针实现。
  pollfd结构包含了要监视的event和发生的event，不再使用select“参数-值”传递的方式。同时，pollfd并没有最大数量限制（但是数量过大后性能也是会下降）。 和select函数一样，poll返回后，需要轮询pollfd来获取就绪的描述符。
- epoll （MySQL连接处理的源码中有使用该模式）
  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，没有描述符限制。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。

> 高性能网络编程(二)：上一个10年，著名的C10K并发连接问题末尾介绍了从select到epoll再到libevent的原因

网络库对比

现有网络库包括：boost的asio、libevent、libev等等，经过调研，libevent主要为支持linux平台设计，使用较为简单，社区较活跃度较高，一直保持版本更新，且被Chromium、Memcached、AliSQL等大型软件使用，稳定性较好。因此，本项目中计划采用libevent作为网络通信库。
libevent还有一个优势在于其只提供了网络API的封装，多余的线程调度等功能需要自己实现，这一设计有利于未来在保持上层调用不变的情况，替换libevent为自己实现的RDMA网络库（保持与libevent相同的API）。

libevent

> 官网
>
> libevent book

进程间通信

> 样例代码可见《unix环境高级编程》

管道

局限性

1. 历史上是半双工（数据只能在一个方向上流动）的，部分系统支持全双工
2. 只能在具有公共祖先的两个进程之间使用

函数

```
int pipe(int fd[2]);
```

工作方式

pipe会返回两个文件描述符，fd[0]用于读，fd[1]用于写；

1. 先调用pipe再调用fork就会形成父子进程的fd[0]和fd[1]通向内核中的同一个管道；
2. 此时关闭父进程的fd[1]和子进程的fd[0]，即可形成由子进程向父进程发送数据的管道（反向与此相似，关闭子进程的写与父进程的读；
3. 然后便可以对文件描述符调用read和write进行管道通信。

> popen与pclose是对管道执行shell命令的一个封装，与本项目关联不大，感兴趣可以自行查阅资料

FIFO（命名管道）

通过FIFIO，不相关的进程也可以交换数据

工作方式

调用mkfifo建立FIFO类型的文件，创建FIFO文件后需要用open函数打开它，open时FIFO的非阻塞标志决定函数是否需要阻塞等待管道“另一头“的读或写进程打开它（“管道通畅”？）。一个给定的FIFO有多个写进程是常见，这意味着，如果不希望多个进程的写数据交叉，则必须考虑原子操作。

用途

1. shell命令使用FIFO将数据从一条管道传送到另一条，无需创建中间文件；
2. 客户端-服务端进程中，使用FIFO作为客户端与服务端之间的数据传递工具。（无法判断客户端是否崩溃，即没有超时机制）

Posix IPC

> 《UNIX网络编程卷2：进程间通信（第2版）》

进程通信方式对比

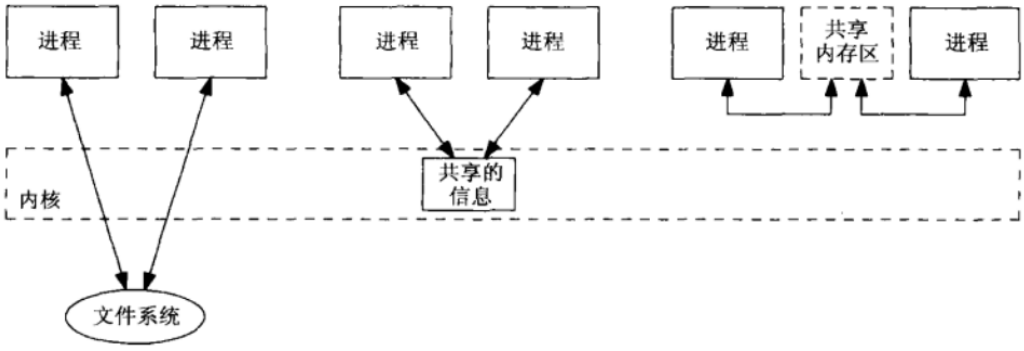

图1-1　Unix进程间共享信息的三种方式

1. 两个进程通过对文件系统上一个文件的读写完成通信；
2. 两个进程共享驻留于内核中的信息，管道、信号量、消息队列都属于此列；
3. 两个进程通过共享的内存区域进行通信，不需要内核参与

> 注：消息队里、信号量和共享内存分为system V与posix两种标准，实现上有所不同，但概念比较接近，因此在此只介绍posix标准的IPC

IPC名字

posix IPC使用名字进行标识，这个名字可能是文件系统中的一个路径名，也可能不是。路径名的命名规则根据系统的不同实现有所区别，请在使用时基于使用的系统查询资料。书

中给出的建议使用#define定义名字，便于在不同系统下修改。

说　明	mq_open	sem_open	shm_open
只读	O_RDONLY		O_RDONLY
只写	O_WRONLY		
读写	O_RDWR		O_RDWR
若不存在则创建	O_CREAT	O_CREAT	O_CREAT
排他性创建	O_EXCL	O_EXCL	O_EXCL
非阻塞模式	O_NONBLOCK		
若已存在则截短			O_TRUNC

图2-3　打开或创建Posix IPC对象所用的各种*oflag*常值

图2-3中mq指消息队列，sem指信号量，shm指共享内存。图2-3中三个函数负责打开或者创建一个IPC对象，消息队列可以通过任意一种方式打开，信号量的打开不指定任何打开模式，共享内存不能通过只读模式打开。

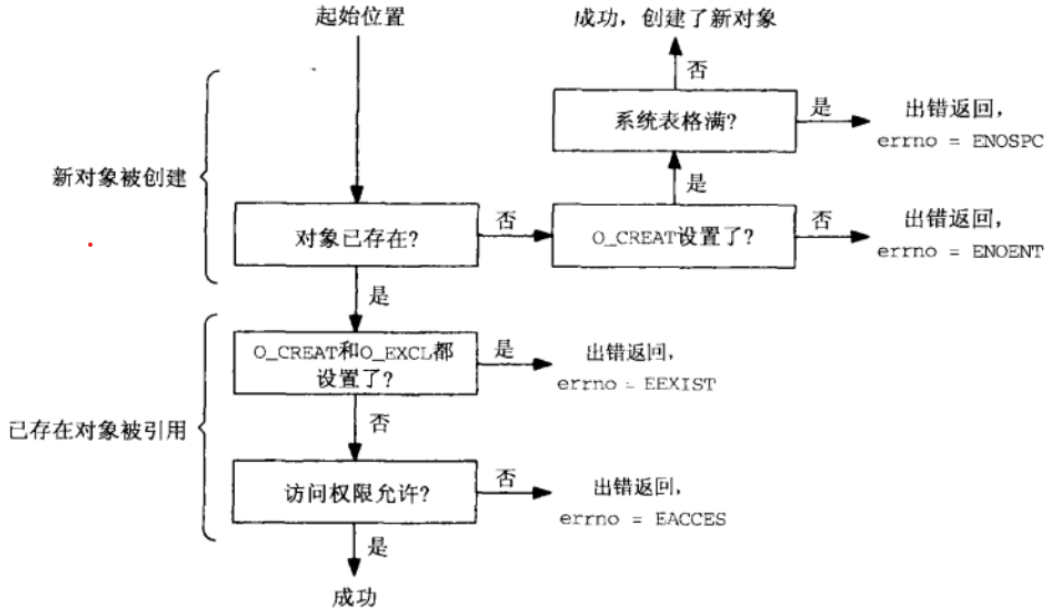

图2-5　打开或创建一个IPC对象的逻辑

图2-5说明了一个IPC对象被打开或创建的逻辑

IPC对象的持续性

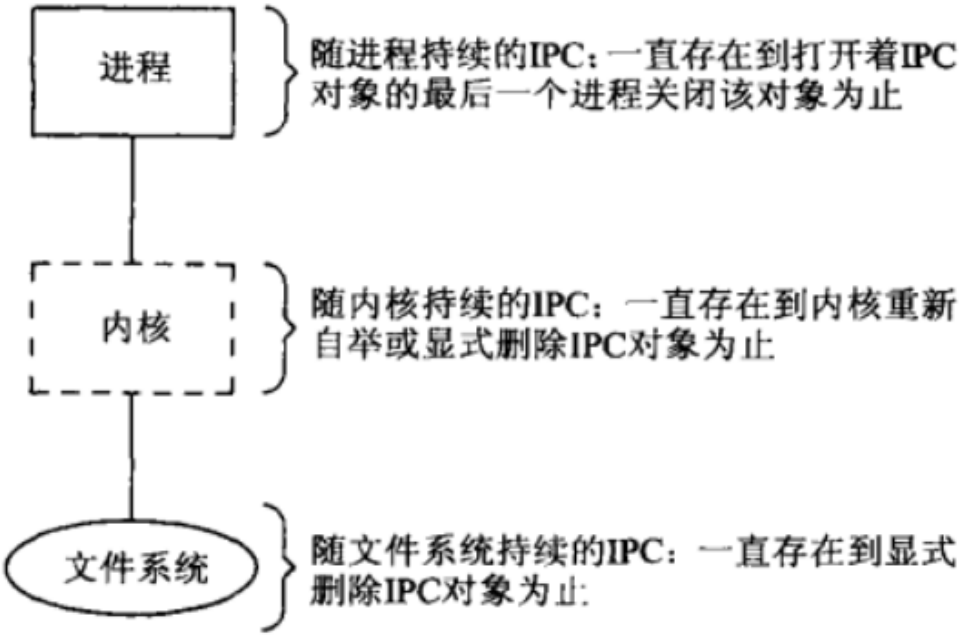

图1-2　IPC对象的持续性

IPC对象的持续性分为图1-2中的三种，管道和FIFO属于进程持续性，system V IPC属于内核持续性，而posix IPC至少是内核持续性，在某些实现里，它也可能是文件系统持续性。进程持续性指使用的进程关闭后，管道中的数据即被丢弃。内核持续性则是在重启前一直保留。文件系统持续性则在内核重启的情况下也可以保留数据（但据说性能会较差）。

消息队列

消息队列可认为是一个消息链表，一个进程往队列放入消息，另一个进程从队列中读取消息。不同的是，消息队列允许进程往队列中写入消息而不用考虑队列的另一头是否有进程等待消息的到达。而管道和FIFO认为除非读者已经存在，否则先有写入者是没有意义的。

与system V的区别：

- posix消息队列的读总是返回最高优先级的消息，而system V可以指定优先级；
- 当向空队列放入一个消息时，posix允许产生一个信号或启动一个线程，而system V不支持类似的提醒机制。

信号量

> 信号量分为三种，由于不经过内核的共享内存IPC效率更高，因此我们重点讨论共享内存上的信号量，另外两种仅提供简述。

信号量是一种用于不同进程间或是不同线程间同步的手段。

system V 信号量

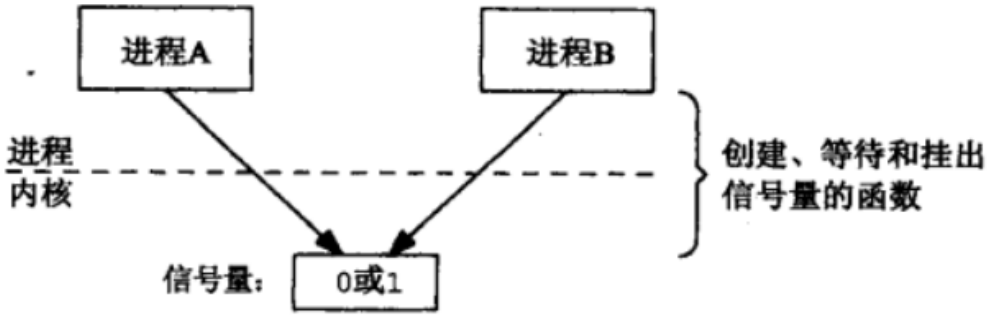

图10-1　由两个进程使用的一个二值信号量

图10-1是system V信号量，由内核维护。

posix 有名信号量

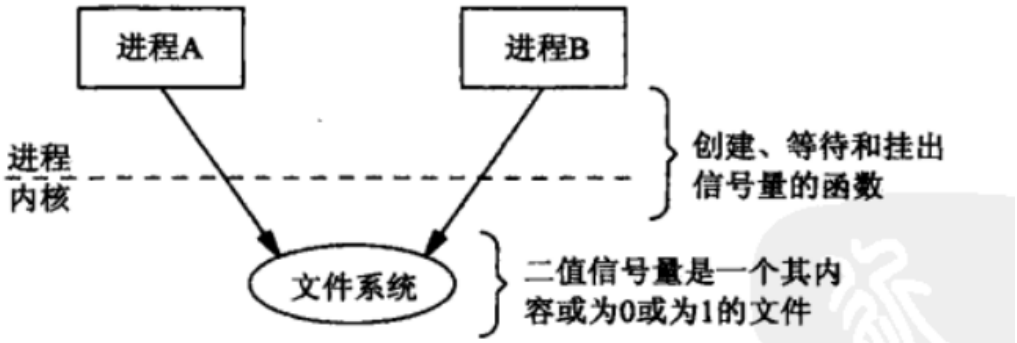

图10-2　由两个进程使用的一个Posix有名二值信号量

图10-2是posix有名信号量，不必在内核中维护，可能与文件系统中的路径名对应。

图中还显示了一个进程在某个信号量上可以执行的三种操作：

1. 创建（create）：指定初始值为0或1；
2. 等待（wait）：测试信号量的值，值小于等于0，则等待（阻塞），若值大于0，则减1；
3. 挂出（post）：将信号量的值加1。

二值信号量可以用于互斥目的。

posix 无名信号量 （基于内存的信号量）

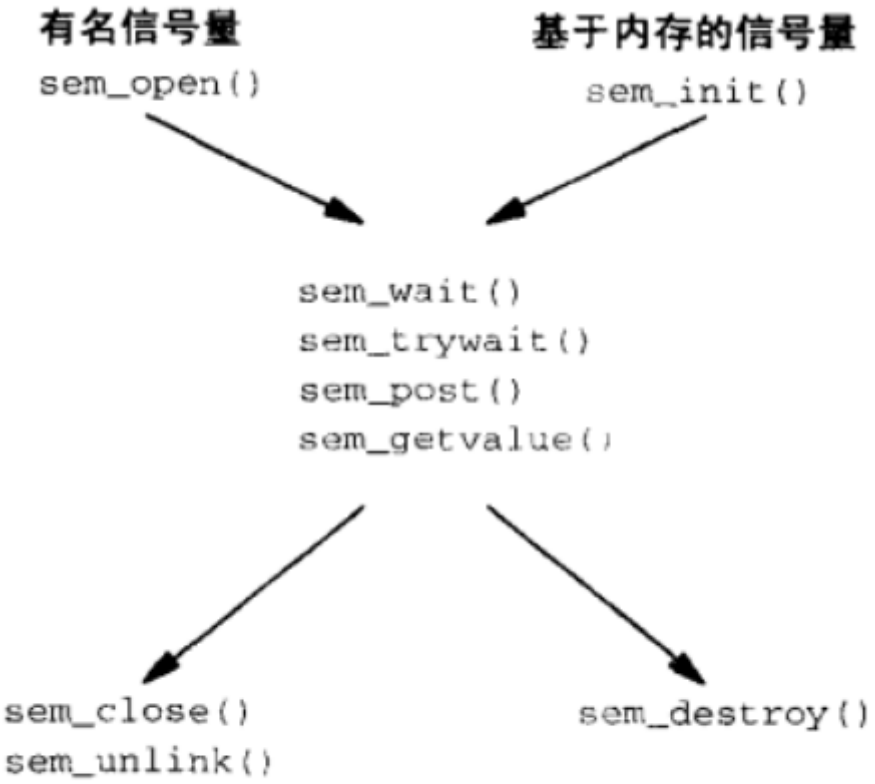

图10-6　用于Posix信号量的函数调用

图10-6是有名信号量与基于内存的信号量的函数调用区别

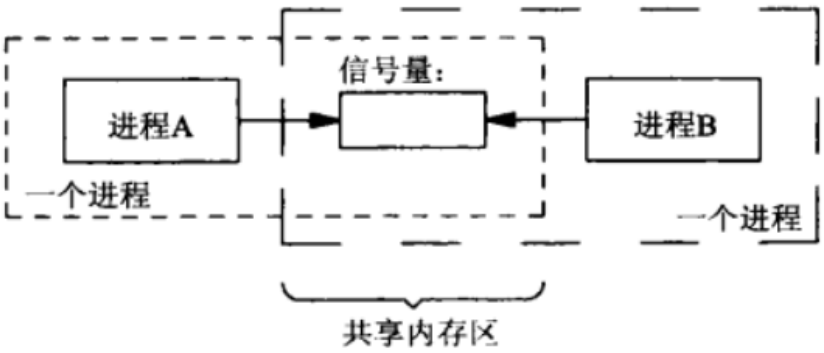

图10-8　由两个进程共享、处于共享内存区中的基于内存的信号量

进程间共享信号量

- 对于有名信号量，只要指定名字，不同进程就可以访问到同一个信号量
- 共享基于内存的信号量，则需要信号量本身驻留在共享内存里

共享内存区

共享内存区是可用IPC形式中最快的。

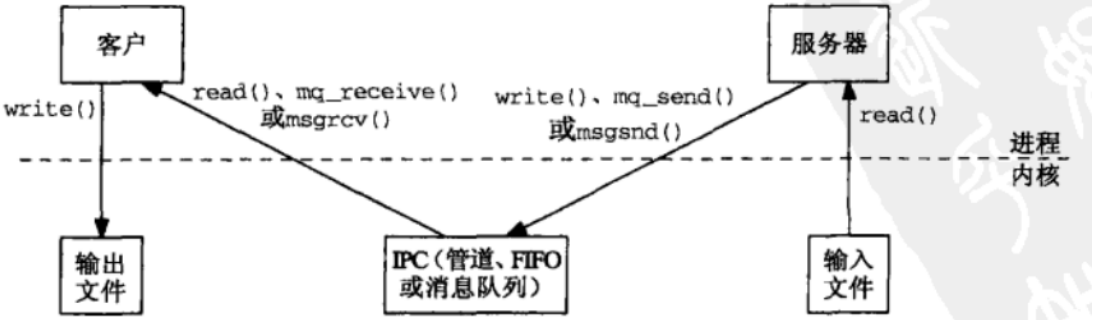

图12-1　从服务器到客户的文件数据流

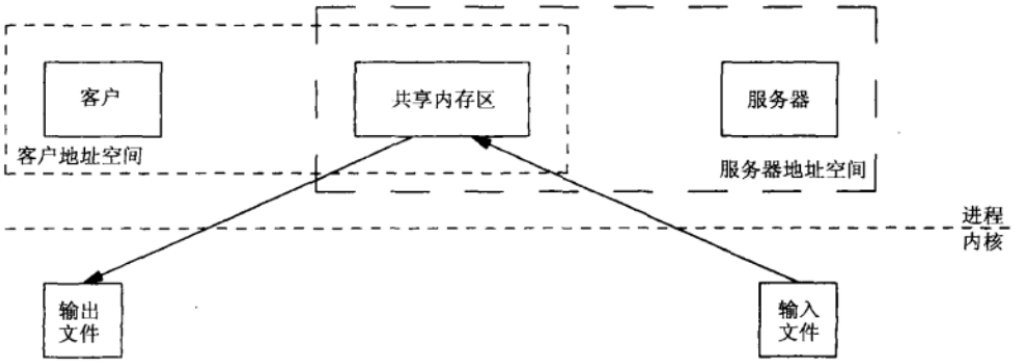

图12-2　使用共享内存区将文件数据从服务器复制到客户

图12-1与图12-2可以看出，不使用共享内存的IPC方式需要四次内存复制（内核和用户态之间的复制），而使用共享内存区仅需要两次

工作方式

1. 使用mmap映射一片共享的内存区域，内核的虚拟内存算法保持内存映射文件与内存映射区的同步；

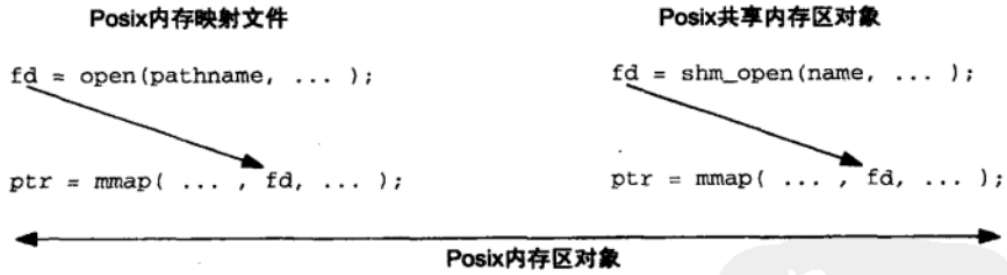

图13-1　Posix内存区对象：内存映射文件和共享内存区对象

2. 无亲缘关系的进程间的共享内存区有两种方案：内存映射文件与共享内存区对象。
   - 内存映射文件即使用open打开一个文件，获得文件描述符fd后，使用mmap将文件映射到当前进程的内存区域中；
   - 共享内存区对象是指使用shm_open打开一个posix IPC名字，返回描述符后由mmap映射到当前进程的地址空间。

> 基于客户端-服务器模型的共享内存区对象使用样例代码见《UNIX网络编程卷2：进程间通信（第2版）》13.6