linux Socket-应用编程-专题讲座

written by 王保明

## 5Socket Api编程进价2

### 服务器端多进程并发子进程退出问题

|  |
| --- |
| **TCP回射客户/服务器模型总结** |
| **TCP是个流协议**   * TCP是基于字节流传输的，只维护发送出去多少，确认了多少，没有维护消息与消息之间的边界，因而可能导致粘包问题。 * 粘包问题解决方法是在应用层维护消息边界。 |
|  |

**僵进程与SIGCHLD信号**

|  |
| --- |
| **服务器端避免僵尸进程的方法：**  **1）通过忽略SIGCHLD信号，解决僵尸进程**   * signal(SIGCHLD, SIG\_IGN) |
| **2）通过wait方法，解决僵尸进程**   * signal(SIGCHLD, handle\_sigchld); * wait(NULL) |
| **3）通过waitpid方法，解决僵尸进程**   * signal(SIGCHLD, handle\_sigchld); * wait(-1, NULL, WNOHANG) |
|  |
|  |

## 6Socket Api 与TCP/IP的11种状态

### TCP/IP协议的11种状态

|  |
| --- |
|  |
| 理解0：什么是主动套接字，什么是被动套接字？  理解1:为什么TCP/IP要三次握手，和四次断开？  理解2：客户端状态向前推进过程，服务器端状态向前推进过程  理解3：执行主动关闭的那一端，进入TIME\_WAIT状态  理解4：TIME\_WAIT 时间是多长2MSL （2倍的最大生命期时间）  原因：（ACK y+1）如果发送失败可以重发。  服务器端处于closed状态，不等于客户端也处于closed状态。。  理解5：图上几种状态，还有一种CLOSING状态  两端同时关闭 将产生closing状态，最后双方都进入TIME\_WAIT状态。 |
|  |
| 实验：   1. 关闭服务方子进程，观察TCP/IP状态 2. 关闭客户端，观察TCP/IP状态。 |

### SIGPIPE

|  |
| --- |
| 如果对方socket已关闭，对等方再发写数据，则会产生SIGPIPE信号   * SIGPIPE信号会让进程终止（man 7 signal，阅读SIGPIPE默认ACT） * 往一个已经接收FIN的套接中写是允许的，接收到FIN仅仅代表对方不再发送数据。 * 在收到RST段之后，如果再调用write就会产生SIGPIPE信号，对于这个信号的处理我们通常忽略即可。   signal(SIGPIPE, SIG\_IGN); |
|  |
| 结论：对SIGPIPE处理方法：1）忽略该信号即可signal(SIGPIPE, SIG\_IGN);  2）捕捉。改变默认行为。 |
| TCP/IP 的RST段重置  1）服务器端启动、客户端启动  2）服务器端先kill与客户端通讯的子进程，服务器端会给客户端发送FIN分节  此时：只代表服务器端不发送数据了，不能代表客户端不能往套接字中写数据。  3）如果子进程此时写数据给服务器端（解除屏幕阻塞，输入字符aaaa），  将要导致TCP/IP协议重置，产生RST段；产生SIGPIPE信号。。  4）所以，一般情况下，需要我们处理SIGPIPE信号，忽略即可。 |

### close与shutdown区别

|  |
| --- |
| * close终止了数据传送的两个方向。 * shutdown可以有选择的终止某个方向的数据传送或者终止数据传送的两个方向。 * shutdown how=1就可以保证对等方接收到一个EOF字符，而不管其他进程是否已经打开了套接字。而close不能保证，直到套接字引用计数减为0时才发送。也就是说直到所有的进程都关闭了套接字。 |
| 思考1  客户端向服务器发送：FIN(close) E D C B A，  问：服务器还能收到数据吗？服务器还可以向客户端回报文吗？  客户端想在关闭之后，仍然能接收到回射服务器应答（shutdown）。  思考2  父进程中close(conn);会不会向客户端发送FIN报文段那？  文件的引用计数-1，当减少为0，才会发送引用计数。  思考3：  客户端//shutdown(sock, SHUT\_WR);只关闭了写； |

## 7五种I/O模型

|  |
| --- |
| **阻塞I/O**    说明1：当上层应用app1调用recv系统调用时，如果对等方没有发送数据（缓冲区没有数据），上层应用app1将阻塞（默认行为，被linux内核阻塞）；  说明2：当对等方发送了数据，linux内核recv端缓冲区，有数据后，内核会把数据copy给用户空间。然后上层应用app1解除阻塞，执行下一步操作。 |
| **非阻塞I/O**    说明1： 上层应用程序app2将套接字设置成非阻塞模式。  说明2： 上层应用程序app2轮询调用recv函数，接受数据。若缓冲区没有数据，上层程序app2不会阻塞，recv返回值为-1，错误码是EWOULDBLOCK。  说明3：上层应用程序不断轮询有没有数据到来。会造成上层应用忙等待。大量消耗CPU。很少直接用。应用范围小，一般和selectIO复用配合使用。 |
|  |
| **I/O复用**    说明1： 上层应用程序app3调用select机制（该机制有linux内核支持，避免了app3忙等待。），进行轮询文件描述符的状态变化。  说明2：当select管理的文件描述符没有数据（或者状态没有变化时），上层应用程序app3也会阻塞。  说明3：好处select机制可以管理多个文件描述符  说明4：select可以看成一个管理者，用select来管理多个IO。  一旦检测到的一个I/O或者多个IO，有我们感兴事件，发生，select函数将返回，返回值为检测到的事件个数。进而可以利用select相关api函数，操作具体事件。  说明5：select函数可以设置等待时间，避免了上层应用程序app3，长期僵死。  说明6: 和阻塞IO模型相比，selectI/O复用模型相当于提前阻塞了。等到有数据到来时，再调用recv就不会发生阻塞。 |
| **信号驱动I/O**    说明1： 上层应用程序app4建立SIGIO信号处理程序。当缓冲区有数据到来，内核会发送信号告诉上层应用程序app4。  说明2：上层应用程序app4接收到信号后，调用recv函数，因缓冲区有数据，recv函数一般不会阻塞。  说明3：这种用于模型用的比较少，属于典型的“拉模式”。即：上层应用app4,需要调用recv函数把数据拉进来。 |
| **异步I/O**    说明1：上层应用程序app5调用aio\_read函数，同时提交一个应用层的缓冲区buf；调用完毕后，不会阻塞。上层应用程序app5可以继续其他任务。  说明2：当tcpip协议缓冲区有数据时，linux主动的把内核数据copy到用户空间。然后再给上层应用app5发送信号；告诉app5数据有了，赶快处理吧！  说明3：典型的“推模式”  说明4： 效率最高的一种形式，上层应用程序app5有异步处理的能力（在linux内核的支持下，言外之意：处理其他任务的同时，也可支持IO通讯）。异步I/O指的是什么？  上层应用程序app5，在也可以干别的活的时，可以接收数据（接受异步通信事件。===）异步命令来源）。与信号驱动IO模型，上层应用程序app5不需要调用recv函数。 |
| **结论：IO复用和异步IO是重点。** |
|  |
|  |

## 8 I/O复用模型

### select基本概念

|  |
| --- |
| 其他重要概念补充   * 阻塞IO   + 数据没有准备好, 读操作就会阻塞   + 数据不能立即被收时, 写操作就会阻塞   + 打开文件时阻塞, 直到某些条件发生 * 非阻塞IO   + 立即返回, 并用错误值来表示当前的状态 * 指定非阻塞方式   + 打开时指定O\_NONBLOCK 标志   + 使用fcntl 打开或关闭非阻塞方式 * 网络编程时, 使用非阻塞, 用轮询方式发送 * 使用多线程可以避免使用非阻塞IO, 但是同步开销较大 |
| 多路IO   * 当程序需要同时从两个输入读数据时 * 使用多进程/多线程, 同步复杂, 进程线程开销 * 使用非阻塞IO, 交替轮询 * 通过信号使用异步IO, 无法判断哪个IO完成 * 多路IO: 把关心的IO放入一个列表, 调用多路函数 * 多路IO函数阻塞, 直到有一个IO数据准备好后返回 * 返回后告诉调用者哪个描述符准备好了 |
| select()实现说明   * 调用select时通过参数告诉内核用户感兴趣的IO描述符 * 关心的IO状态: 输入,输出或错误 * 调用者等待时间 * 返回之后内核告诉调用者多个描述符准备好了 * 哪些描述符发生了变化 * 调用返回后对准备好的描述符调用读写操作 * 不关心的描述符集合传NULL |
|  |
|  |
| **select()**  /\* According to POSIX 1003.1-2001 \*/  #include <sys/select.h>  /\* According to earlier standards \*/  #include <sys/time.h>  #include <sys/types.h>  #include <unistd.h>  int **select**(int n, fd\_set \*readfds, fd\_set \*writefds, fd\_set \*exceptfds, struct timeval \*timeout);  如果成功，返回所有sets中描述符的个数；如果超时，返回0；如果出错，返回-1。 |
| * 监视readfds来查看是否read的时候会被堵塞，注意，即便到了end-of-file，fd也是可读的。 * 监视writefds看写的时候会不会被堵塞。 * 监视exceptfd是否出现了异常。主要用来读取OOB数据，异常并不是指出错。 * 注意当一个套接口出错时，它会变得既可读又可写。 * 如果有了状态改变，会将其他fd清零，只有那些发生改变了的fd保持置位，以用来指示set中的哪一个改变了状态。 * 参数n是所有set里所有fd里，具有最大值的那个fd的值加1 |
| **fd\_set**  四个宏用来对fd\_set进行操作：  FD\_CLR(int fd, fd\_set \*set);  FD\_ISSET(int fd, fd\_set \*set);  FD\_SET(int fd, fd\_set \*set);  FD\_ZERO(fd\_set \*set);   * FD\_ZERO用来清空set； * FD\_SET和FD\_CLR用来对某个set添加和删除一个fd； * FD\_ISSET用来指示一个fd是不是一个set的一部分。他很有用，用来看select后哪一个fd可用了。 |
| **time\_out**   * timeout是从调用开始到select返回前，会经历的最大等待时间。 * 两种特殊情况：如果为值为0，会立刻返回。 * 如果timeout是NULL,会阻塞式等待。   struct timeval {  long tv\_sec; /\* seconds \*/  long tv\_usec; /\* microseconds \*/  };   * 一些调用使用3个空的set, n为zero, 一个非空的timeout来达到较为精确的sleep. * Linux中, select函数改变了timeout值，用来指示还剩下的时间，但很多实现并不改timeout。 * 为了较好的可移植性，timeout在循环中需要被重新赋初值。 * timeout== NULL   + 无限等待   + 被信号打断时返回1, errno 设置成 EINTR * timeout->tv\_sec == 0 && tvptr->tv\_usec == 0   + 不等待立即返回 * timeout->tv\_sec != 0 || tvptr->tv\_usec != 0   + 等待特定时间长度, 超时返回0 |
| **select示例**  man 手册 |

select实现原理说明

|  |
| --- |
| * fd\_set是一个位向量, 每位表示一个描述符      * int FD\_ISSET(int fd, fd\_set \*fdset);   + 测试某个描述符是否在集合内 * void FD\_CLR(int fd, fd\_set \*fdset);   + 从集合内把一个描述符移除 * void FD\_SET(int fd, fd\_set \*fdset);   + 把一个描述符加入集合 * void FD\_ZERO(fd\_set \*fdset);   + 清空描述符集合 |
| * 可以把同一个描述符同时放取读和写集合 * 当读和写者准备好时, 返回值的计数分别加1次 * 普通文件的三种状态总是返回准备好的状态 * 是否阻塞式IO不会影响select的结果 * 如果一个描述符到了文件结尾,select返回的状态是准备好 * 对一个准备好的描述符, 读出长度是0表示到达结尾 |
|  |

### 简单select语法

### 用select完善echoclient

### 用select完善echoserver

### 套接字IO超时设置方法

|  |
| --- |
| **套接字I/O超时设置方法**   * **alarm** * **套接字选项**    + **SO\_SNDTIMEO**   + **SO\_RCVTIMEO** * **select** |
| **闹钟方法alarm**  方案1  void handler(int sig)  {  return 0; //闹钟仅用来做打断read操作，无其他用途  }  //设置一个闹钟，5秒钟以后打断read函数，达到超时的目的  SIGALRM  signal(SIGALRM, handle);  alarm(5)  int ret = read(fd, buf, sizeof(buf));  if (ret==-1 && errno=EINTR)  {  errno = ETIMEDOUT;  }  else if (ret >=0)  {  alarm(0)';  } |
| **套接字选项**  SO\_SNDTIME 发送的时间  SO\_RCVTIME 接受的时间  setsockopt (sock, SOL\_SOCKET, SO\_RCVTIMEO, 5);  if (ret==-1 && errno=EWOULDBLOCK)  {  errno = ETIMEDOUT;  }  缺点：不易移植，有些内核没有实现。 |
| **select**   * **用select实现超时**   + **read\_timeout函数封装**   + **write\_timeout函数封装**   + **accept\_timeout函数封装**   + **connect\_timeout函数封装** |

## 9socket性能测试