# 网络编程第3天

## 内容回顾

当read读文件描述符为非阻塞状态的时候, 若对方没有发送数据, 会立刻返回, errno设置为EAGAIN, 这个错误我们要忽略.

## 学习目标

* 熟练掌握TCP状态转换图
* 熟练掌握端口复用的方法
* 了解半关闭的概念和实现方式
* 了解多路IO转接模型
* 熟练掌握select函数的使用
* 熟练使用fd\_set相关函数的使用
* 能够编写select多路IO转接模型的代码

## TCP状态转换图



了解TCP状态转换图可以帮助开发人员查找问题.

说明: 上图中粗线表示主动方, 虚线表示被动方, 细线部分表示一些特殊情况, 了解即可, 不必深入研究.

对于建立连接的过程客户端属于主动方, 服务端属于被动接受方(图的上半部分)

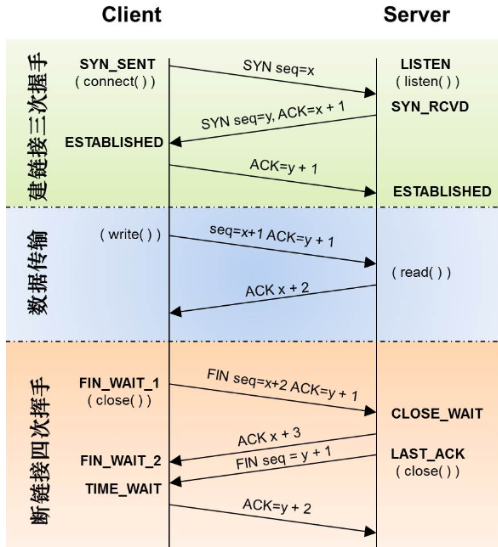
而对于关闭(图的下半部分), 服务端和客户端都可以先进行关闭.

处于ESTABLISHED状态的时候就可以收发数据了, 双方在通信过程当中一直处于ESTABLISHED状态, 数据传输期间没有状态的变化.

TIME\_WAIT状态一定是出现在主动关闭的一方.

主动关闭的Socket端会进入TIME\_WAIT状态，并且持续2MSL时间长度，MSL就是maximum segment lifetime(最大分节生命期），这是一个IP数据包能在互联网上生存的最长时间，超过这个时间将在网络中消失。

使用netstat -anp可以查看连接状态



注:数据传输的时候带了一个字节的数据, 所以server发送给client的ACK=x+2

为什么需要2MSL?

原因之一: 让四次挥手的过程更可靠, 确保最后一个发送给对方的ACK到达;

若对方没有收到ACK应答, 对方会再次发送FIN请求关闭, 此时在2MS时间内被动关闭方仍然可以发送ACK给对方.

原因之二: 为了保证在2MS时间内, 不能启动相同的SOCKET-PAIR.

TIME\_WAIT一定是出现在主动关闭的一方, 也就是说2MS是针对主动关 闭一方来说的;由于TCP有可能存在丢包重传, 丢包重传若发给了已经断 开连接之后相同的socket-pair(该连接是新建的, 与原来的socket-pair完 全相同, 双方使用的是相同的IP和端口), 这样会对之后的连接造成困扰, 严重可能引起程序异常.

如何避免问题2呢??

--很多操作系统实现的时候, 只要端口被占用, 服务就不能启动.

测试: 启动服务端和客户端, 然后先关闭服务端, 再次启动服务端, 此时服务端报错: bind error: Address already in use; 若是先关闭的客户端, 再关闭的服务端, 此时启动服务端就不会报这个错误.

socket-pair的概念: 客户端与服务端连接其实是一个连接对, 可以通过使用netstat -anp | grep 端口号 进行查看.

## 端口复用

解决端口复用的问题: bind error: Address already in use, 发生这种情况是在服务端主动关闭连接以后, 接着立刻启动就会报这种错误.

setsockopt函数

int setsockopt(int sockfd, int level, int optname, const void \*optval, socklen\_t optlen);

setsockopt(lfd, SOL\_SOCKET, SO\_REUSEADDR, &opt, sizeof(int));

setsockopt(lfd, SOL\_SOCKET, SO\_REUSEPORT, &opt, sizeof(int));

函数说明可参看<<UNIX环境高级编程>>

由于错误是bind函数报出来的, 该函数调用要放在bind之前, socket之后调用.

## 半关闭状态

半关闭的概念:

如果一方close, 另一方没有close, 则认为是半关闭状态, 处于半关闭状态的 时候, 可以接收数据, 但是不能发送数据. 相当于把文件描述符的写缓冲区 操作关闭了.

注意: 半关闭一定是出现在主动关闭的一方.

shutdown函数

长连接和端连接的概念:

连接建立之后一直不关闭为长连接;

连接收发数据完毕之后就关闭为短连接;

shutdown和close的区别:

shutdown能够把文件描述符上的读或者写操作关闭, 而close关闭文件描述 符只是将连接的引用计数的值减1, 当减到0就真正关闭文件描述符了.

如: 调用dup函数或者dup2函数可以复制一个文件描述符, close其中一个并 不影响另一个文件描述符, 而shutdown就不同了, 一旦shutdown了其中一 个文件描述符, 对所有的文件描述符都有影响 .

## 心跳包

如何检查与对方的网络连接是否正常??

一般心跳包用于长连接.

方法1

keepAlive = 1;

setsockopt(listenfd, SOL\_SOCKET, SO\_KEEPALIVE, (void\*)&keepAlive, sizeof(keepAlive));

由于不能实时的检测网络情况, 一般不用这种方法

方法2: 在应用程序中自己定义心跳包, 使用灵活, 能实时把控.

到此为止, 概念相关的东西就讲完毕了.

## 高并发服务器模型--select

继续研究高并发服务器的问题.

多路IO技术: select, 同时监听多个文件描述符, 将监控的操作交给内核去处理,

数据类型fd\_set: 文件描述符集合--本质是位图(关于集合可联想一个信号集sigset\_t)

int select(int nfds, fd\_set \* readfds, fd\_set \*writefds, fd\_set \*exceptfds, struct timeval \*timeout);

函数介绍: 委托内核监控该文件描述符对应的读,写或者错误事件的发生.

参数说明:

nfds: 最大的文件描述符+1

readfds: 读集合, 是一个传入传出参数

传入: 指的是告诉内核哪些文件描述符需要监控

传出: 指的是内核告诉应用程序哪些文件描述符发生了变化

writefds: 写文件描述符集合(传入传出参数)

execptfds: 异常文件描述符集合(传入传出参数)

timeout:

NULL--表示永久阻塞, 直到有事件发生

0 --表示不阻塞, 立刻返回, 不管是否有监控的事件发生

>0--到指定事件或者有事件发生了就返回

返回值: 成功返回发生变化的文件描述符的个数

失败返回-1, 并设置errno值.

/usr/include/x86\_64-linux-gnu/sys/select.h和

/usr/include/x86\_64-linux-gnu/bits/select.h

从上面的文件中可以看出, 这几个宏本质上还是位操作.

void FD\_CLR(int fd, fd\_set \*set);

将fd从set集合中清除.

int FD\_ISSET(int fd, fd\_set \*set);

功能描述: 判断fd是否在集合中

返回值: 如果fd在set集合中, 返回1, 否则返回0.

void FD\_SET(int fd, fd\_set \*set);

将fd设置到set集合中.

void FD\_ZERO(fd\_set \*set);

初始化set集合.

调用select函数其实就是委托内核帮我们去检测哪些文件描述符有可读数据,可写,错误发生;

代码思路:

代码的具体实现: 编写代码并进行测试.

可以使用发生事件的总数进行控制, 减少循环次数

调用select函数涉及到了用户空间和内核空间的数值交互过程.

事件一共包括两部分, 一类是新连接事件, 一类是有数据可读的事件

问题分析: select函数的readfds是一个传出传入参数

测试和总结select用法

关于select的思考:

问题: 如果有效的文件描述符比较少, 会使循环的次数太多.

解决办法: 可以将有效的文件描述符放到一个数组当中, 这样遍历效率就高了.

select优点:

1 一个进程可以支持多个客户端

2 select支持跨平台

select缺点:

1 代码编写困难

2 会涉及到用户区到内核区的来回拷贝

3 当客户端多个连接, 但少数活跃的情况, select效率较低

例如: 作为极端的一种情况, 3-1023文件描述符全部打开, 但是只有1023有发送数据, select就显得效率低下

4 最大支持1024个客户端连接

select最大支持1024个客户端连接不是有文件描述符表最多可以支持1024个文件描述符限制的, 而是由FD\_SETSIZE=1024限制的.

FD\_SETSIZE=1024 fd\_set使用了该宏, 当然可以修改内核, 然后再重新编译内核, 一般不建议这么做.

作业:

编写代码, 让select监控标准输入, 监控网络, 如果标准输入有数据就写入网络, 如果网络有数据就读出网络数据, 然后打印到标准输出.

注意: select不仅可以监控socket文件描述符, 也可以监视标准输入.

预习内容:

poll epoll epoll反应堆

[POSIX](https://baike.baidu.com/item/POSIX" \t "https://baike.baidu.com/item/POSIX/_blank)表示[可移植操作系统接口](https://baike.baidu.com/item/%E5%8F%AF%E7%A7%BB%E6%A4%8D%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F%E6%8E%A5%E5%8F%A3" \t "https://baike.baidu.com/item/POSIX/_blank)（Portable Operating System Interface of UNIX，缩写为 POSIX ），POSIX标准定义了操作系统应该为应用程序提供的接口标准.

关于fd\_set类型的底层定义:

/usr/include/x86\_64-linux-gnu/sys/select.h和

/usr/include/x86\_64-linux-gnu/bits/select.h

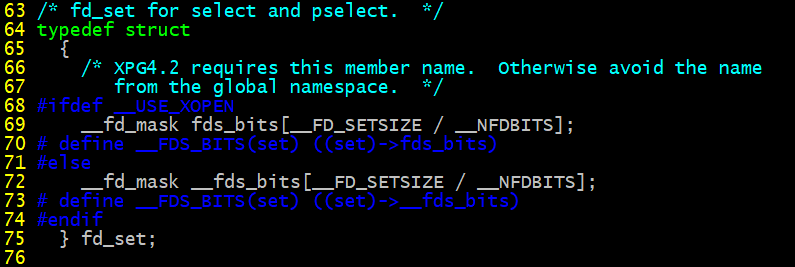
在/usr/include/x86\_64-linux-gnu/sys/select.h文件中:





\_\_NFDBITS计算出来的值是: 8\*8=64





上面是在头文件中一步一步跟踪的定义, 最简单的方法就是使用预处理将头文件和宏全部替换掉, 直接就可以看到最终的定义了.

如: gcc -E select.c -o select.i

打开select.i后

typedef struct

{

\_\_fd\_mask \_\_fds\_bits[1024 / (8 \* (int) sizeof (\_\_fd\_mask))];

} fd\_set;

进一步转换后:

typedef struct

{

long int \_\_fds\_bits[1024/(8\*8))];

//long int \_\_fds\_bits[16];

}

这个数组一共占用: 8 \* 16 \* 8 = 1024, 也就是说fd\_set这个文件描述符表中一共有1024个bit位, 每个bit位只有0和1两种值, 1表示有, 0表示没有.