众所周知，Linux下的多路复用函数select采用描述符集表示处理的描述符。描述符集的大小就是它所能处理的最大描述符限制。通常情况下该值为1024，等同于每个进程所能打开的描述符个数。

增大描述符集大小的唯一方法是先增大FD\_SETSIZE的值，然后重新编译内核，不重新编译内核而改变其值时不够的。

在阅读Libev源码时，发现它实现了一种突破这种限制的方法。该方法本质上而言，就是自定义fd\_set结构，以及FD\_SET， FD\_CLR，FD\_ISSET宏。

首先看一下Linux中原fd\_set结构的实现细节，我的虚拟机的系统版本是Ubuntu 14.10，内核为3.16.0-23-generic。在其中的/usr/include/i386-linux-gnu/目录下的<sys/select.h>、<bits/select.h>、<bits/typesizes.h>文件中找到了fd\_set结构的实现细节，代码如下：

//<bits/typesizes.h>

#define \_\_FD\_SETSIZE 1024

//<sys/select.h>

**typedef** long int \_\_fd\_mask**;**

**typedef** \_\_fd\_mask fd\_mask**;**

#define FD\_SETSIZE \_\_FD\_SETSIZE

#define \_\_NFDBITS (8 \* (int) sizeof (\_\_fd\_mask))

#define NFDBITS \_\_NFDBITS

#define \_\_FD\_ELT(d) ((d) / \_\_NFDBITS)

#define \_\_FD\_MASK(d) ((\_\_fd\_mask) 1 << ((d) % \_\_NFDBITS))

**typedef** struct

**{**

/\* XPG4.2 requires this member name. Otherwise avoid the name

from the global namespace. \*/

#ifdef \_\_USE\_XOPEN

\_\_fd\_mask fds\_bits**[**\_\_FD\_SETSIZE **/** \_\_NFDBITS**];**

# define \_\_FDS\_BITS(set) ((set)->fds\_bits)

#else

\_\_fd\_mask \_\_fds\_bits**[**\_\_FD\_SETSIZE **/** \_\_NFDBITS**];**

# define \_\_FDS\_BITS(set) ((set)->\_\_fds\_bits)

#endif

**}** fd\_set**;**

//<bits/select.h>

#define \_\_FD\_SET(d, set) \

((void) (\_\_FDS\_BITS (set)[\_\_FD\_ELT (d)] |= \_\_FD\_MASK (d)))

#define \_\_FD\_CLR(d, set) \

((void) (\_\_FDS\_BITS (set)[\_\_FD\_ELT (d)] &= ~\_\_FD\_MASK (d)))

#define \_\_FD\_ISSET(d, set) \

((\_\_FDS\_BITS (set)[\_\_FD\_ELT (d)] & \_\_FD\_MASK (d)) != 0)

#if defined \_\_GNUC\_\_ && \_\_GNUC\_\_ >= 2

# define \_\_FD\_ZERO(fdsp) \

do { \

int \_\_d0, \_\_d1; \

\_\_asm\_\_ \_\_volatile\_\_ ("cld; rep; " \_\_FD\_ZERO\_STOS \

: "=c" (\_\_d0), "=D" (\_\_d1) \

: "a" (0), "0" (sizeof (fd\_set) \

/ sizeof (\_\_fd\_mask)), \

"1" (&\_\_FDS\_BITS (fdsp)[0]) \

: "memory"); \

} while (0)

#else /\* ! GNU CC \*/

# define \_\_FD\_ZERO(set) \

do { \

unsigned int \_\_i; \

fd\_set \*\_\_arr = (set); \

for (\_\_i = 0; \_\_i < sizeof (fd\_set) / sizeof (\_\_fd\_mask); ++\_\_i) \

\_\_FDS\_BITS (\_\_arr)[\_\_i] = 0; \

} while (0)

#endif /\* GNU CC \*/

//<sys/select.h>

#define FD\_SET(fd, fdsetp) \_\_FD\_SET (fd, fdsetp)

#define FD\_CLR(fd, fdsetp) \_\_FD\_CLR (fd, fdsetp)

#define FD\_ISSET(fd, fdsetp) \_\_FD\_ISSET (fd, fdsetp)

#define FD\_ZERO(fdsetp) \_\_FD\_ZERO (fdsetp)

代码比较简单，就不一一分析了，总结一下就是：fd\_set是个结构体，它内部包含一个long int类型的数组，数组长度为\_\_FD\_SETSIZE / \_\_NFDBITS。也就是说该数组一共包含\_\_FD\_SETSIZE个位。每一位就代表一个描述符。

FD\_SET就是将该数组相应的位置1，FD\_CLR就是将该数组相应的位置0，FD\_ISSET就是判断某位是否为1.

以上就是fd\_set的具体实现。Libev采用的方法是动态申请fd\_set的空间，并实现相应的宏。下面是模仿Libev的代码，实现的一个简单的echo服务器：

# define NFDBYTES (NFDBITS / 8)

#define IFERR(res, msg) \

if(res < 0) \

{ \

perror(#msg " error"); \

return; \

}

#define MYFDSET(fd) \

word = fd / NFDBITS; \

mask = 1UL << (fd % NFDBITS); \

if(word+1 > fdsize) \

{ \

roset = realloc(roset, (word+1) \* NFDBYTES); \

rset = realloc(rset, (word+1) \* NFDBYTES); \

for(; fdsize < word+1; fdsize++) \

{ \

((fd\_mask \*)rset)[fdsize] = ((fd\_mask \*)roset)[fdsize] = 0; \

} \

} \

((fd\_mask \*)roset)[word] |= mask;

#define MYFDCLR(fd) \

word = fd / NFDBITS; \

mask = 1UL << (fd % NFDBITS); \

((fd\_mask \*)roset)[word] &= ~mask;

void echoserver**()**

**{**

int listenfd **=** **-**1**,** confd **=** **-**1**;**

int res **=** **-**1**;**

int val **=** 1**;**

int i**;**

char rbuf**[**1024**];**

char sbuf**[**1024**];**

void **\***roset **=** **NULL;**

void **\***rset **=** **NULL;**

int fdsize **=** 0**;**

int word**;**

fd\_mask mask**;**

struct sockaddr\_in serveraddr**;**

int addrlen **=** **sizeof(**serveraddr**);**

listenfd **=** socket**(**AF\_INET**,** SOCK\_STREAM**,** 0**);**

IFERR**(**listenfd**,** socket**);**

serveraddr**.**sin\_family **=** AF\_INET**;**

serveraddr**.**sin\_port **=** htons**(**8898**);**

serveraddr**.**sin\_addr**.**s\_addr **=** htonl**(**INADDR\_ANY**);**

res **=** setsockopt**(**listenfd**,** SOL\_SOCKET**,** SO\_REUSEADDR**,** **(**void **\*)&**val**,** **sizeof(**val**));**

IFERR**(**res**,** setsockopt**);**

res **=** bind**(**listenfd**,** **(**struct sockaddr **\*)&**serveraddr**,** addrlen**);**

IFERR**(**res**,** bind**);**

**MYFDSET(listenfd);**

res **=** listen**(**listenfd**,** 5**);**

IFERR**(**res**,** "listen error"**)**

**for(;;)**

**{**

memcpy**(**rset**,** roset**,** fdsize**\***NFDBYTES**);**

**res = select(fdsize \* NFDBITS, (fd\_set \*)rset, NULL, NULL, NULL);**

IFERR**(**res**,** select**)**

**for(**i **=** fdsize**-**1**;** i **>=** 0**;** i**--)**

**{**

**if(((**fd\_mask **\*)**rset**)[**i**]** **==** 0**)** **continue;**

int bit **=** 0**;**

fd\_mask bitmask **=** 0**;**

int fd **=** 0**;**

**for(**bit **=** NFDBITS**-**1**;** bit **>=** 0**;** bit**--)**

**{**

bitmask **=** 1UL **<<** bit**;**

**if(((**fd\_mask **\*)**rset**)[**i**]** **&** bitmask**)**

**{**

fd **=** i**\***NFDBITS **+** bit**;**

**if(**fd **==** listenfd**)**

**{**

confd **=** accept**(**listenfd**,** **NULL,** **NULL);**

IFERR**(**confd**,** accept**)**

**MYFDSET(confd)**

**}**

**else**

**{**

bzero**(**rbuf**,** 1024**);**

res **=** read**(**fd**,** rbuf**,** 1024**);**

**if(**res **<=** 0**)**

**{**

**if(**res **<** 0**)**perror**(**"read error"**);**

**else** printf**(**"client over\n"**);**

**MYFDCLR(fd);**

close**(**fd**);**

**}**

**else**

**{**

snprintf**(**sbuf**,** 1024**,** "server echo: %s"**,** rbuf**);**

write**(**fd **,**sbuf**,** strlen**(**sbuf**));**

**}**

**}**

**}**

**}**

**}**

**}**

**}**

该代码中，rset和roset是表示描述符集的动态数组，fdsize \* NFDBITS就是该描述符集所能表示的最大位数。当需要添加新的描述符时，如果空间不够，则调用realloc动态增加。并置相应的位为1。除了动态增长之外，MYFDSET和MYFDCLR实现思路与原来的FD\_SET和FD\_CLR一样的。

下面是简单的测试代码，用python实现的TCP客户端：

**from** socket **import** **\***

**import** sys

**from** time **import** sleep

serverAddr **=** **(**'localhost'**,** 8898**)**

clientlist **=** **[]**

errlist **=** **[]**

clientcnts **=** int**(**sys**.**argv**[**1**])**

**for** i **in** xrange**(**clientcnts**):**

clientlist**.**insert**(**i**,** socket**(**AF\_INET**,** SOCK\_STREAM**))**

**for** index**,** client **in** enumerate**(**clientlist**):**

**try:**

client**.**connect**(**serverAddr**)**

**except** Exception **as** e**:**

**print** 'exception catched : ' **,**e

errlist**.**append**(**client**)**

**for** i **in** errlist**:**

clientlist**.**remove**(**i**)**

**for** index**,** client **in** enumerate**(**clientlist**):**

data **=** "this is %d client"**%**index

client**.**send**(**data**)**

data **=** client**.**recv**(**100**)**

**print** 'recieve : '**,** data

sleep**(**10**)**

**for** index**,** client **in** enumerate**(**clientlist**):**

client**.**close**()**

测试时，运行如下命令即可，3000表示并发量。

# python tcpclient.py 3000

当然，在运行服务端和客户端之前，需要增加每个进程所能打开的描述符的限制，否则的话，客户端会出现错误：socket.error: [Errno 24] Too many open files，服务端会出现错误：accept error: Too many open files。 增加限制的命令如下：

# ulimit -n 65535

总结： 虽然select的最大描述符限制是可以突破的，但是与poll或者epoll相比，select在处理大量描述符时依然性能有限。select的优势在于它的跨平台特性；以及在处理少量描述符时，它可能是最快的；而且如果多数的描述符都处于活跃状态的话，它的性能也会非常好。

参考：

http://pod.tst.eu/http://cvs.schmorp.de/libev/ev.pod

<http://codemacro.com/2014/06/01/select-limit/>