  登录

首页 专栏 专家 热文

shuningzhang的专栏 .

[转]linux内核内存管理(zone\_dma zone\_normal zone\_highmem)

2014-7-20阅读196 评论0

Linux 操作系统和驱动程序运行在内核空间，应用程序运行在用户空间，两者不能简单地使用指针传递数据，因为Linux使用的虚拟内存机制，用户空间的数据可能被换出，当内核空间使用用户空间指针时，对应的数据可能不在内存中。

   Linux内核地址空间划分

通常32位Linux内核地址空间划分0~3G为用户空间，3~4G为内核空间。注意这里是32位内核地址空间划分，64位内核地址空间划分是不同的。

1、x86的物理地址空间布局：

wKiom1Nm\_tiyhQPXAACWQfYUvVU353.jpg

  物理地址空间的顶部以下一段空间，被PCI设备的I/O内存映射占据，它们的大小和布局由PCI规范所决定。640K~1M这段地址空间被BIOS和VGA适配器所占据。

  Linux系统在初始化时，会根据实际的物理内存的大小，为每个物理页面创建一个page对象，所有的page对象构成一个mem\_map数组。

进一步，针对不同的用途，Linux内核将所有的物理页面划分到3类内存管理区中，如图，分别为ZONE\_DMA，ZONE\_NORMAL，ZONE\_HIGHMEM。

  ZONE\_DMA的范围是0~16M，该区域的物理页面专门供I/O设备的DMA使用。之所以需要单独管理DMA的物理页面，是因为DMA使用物理地址访问内存，不经过MMU，并且需要连续的缓冲区，所以为了能够提供物理上连续的缓冲区，必须从物理地址空间专门划分一段区域用于DMA。

  ZONE\_NORMAL的范围是16M~896M，该区域的物理页面是内核能够直接使用的。

  ZONE\_HIGHMEM的范围是896M~结束，该区域即为高端内存，内核不能直接使用。

2、linux虚拟地址内核空间分布

wKiom1Nm\_tezL0\_TAACljE7bXGU278.jpg

  在kernel image下面有16M的内核空间用于DMA操作。位于内核空间高端的128M地址主要由3部分组成，分别为vmalloc area，持久化内核映射区，临时内核映射区。

  由于ZONE\_NORMAL和内核线性空间存在直接映射关系，所以内核会将频繁使用的数据如kernel代码、GDT、IDT、PGD、mem\_map数组等放在ZONE\_NORMAL里。而将用户数据、页表(PT)等不常用数据放在ZONE\_ HIGHMEM里，只在要访问这些数据时才建立映射关系(kmap())。比如，当内核要访问I/O设备存储空间时，就使用ioremap()将位于物理地址高端的mmio区内存映射到内核空间的vmalloc area中，在使用完之后便断开映射关系。

3、linux虚拟地址用户空间分布

wKiom1NnADvC07kmAACCVbbosf0493.jpg

  用户进程的代码区一般从虚拟地址空间的0x08048000开始，这是为了便于检查空指针。代码区之上便是数据区，未初始化数据区，堆区，栈区，以及参数、全局环境变量。

4、linux虚拟地址与物理地址映射的关系

wKioL1Nm\_qiCvmYxAADRFZV\_1jQ205.jpg

  Linux将4G的线性地址空间分为2部分，0~3G为user space，3G~4G为kernel space。

  由于开启了分页机制，内核想要访问物理地址空间的话，必须先建立映射关系，然后通过虚拟地址来访问。为了能够访问所有的物理地址空间，就要将全部物理地址空间映射到1G的内核线性空间中，这显然不可能。于是，内核将0~896M的物理地址空间一对一映射到自己的线性地址空间中，这样它便可以随时访问ZONE\_DMA和ZONE\_NORMAL里的物理页面；此时内核剩下的128M线性地址空间不足以完全映射所有的ZONE\_HIGHMEM，Linux采取了动态映射的方法，即按需的将ZONE\_HIGHMEM里的物理页面映射到kernel space的最后128M线性地址空间里，使用完之后释放映射关系，以供其它物理页面映射。虽然这样存在效率的问题，但是内核毕竟可以正常的访问所有的物理地址空间了。

5、buddyinfo的理解

cat /proc/buddyinfo 显示如下：

Node 0, zone      DMA       0      4      5      4      4      3 ...

Node 0, zone   Normal      1      0      0      1    101     8 ...

Node 0, zone  HighMem    2      0      0      1      1       0 ...

其中，Node表示在NUMA环境下的节点号，这里只有一个节点0；zone表示每一个节点下的区域，一般有DMA、Normal和HignMem三个区域；后面的列表示，伙伴系统中每一个order对应的空闲页面块。例如，对于zone DMA的第二列（从0开始算起），空闲页面数为5\*2^4，可用内存为5\*2^4\*PAGE\_SIZE。

计算方法就是：

                     当前列的数字\*2^列数\*PAGE\_SIZE 其中列数是从0开始计算的，即第一列是 当前列的数字\*2^0\*PAGE\_SIZE

常见问题：

1、用户空间（进程）是否有高端内存概念？

用户进程没有高端内存概念。只有在内核空间才存在高端内存。用户进程最多只可以访问3G物理内存，而内核进程可以访问所有物理内存。

2、64位内核中有高端内存吗？

目前现实中，64位Linux内核不存在高端内存，因为64位内核可以支持超过512GB内存。若机器安装的物理内存超过内核地址空间范围，就会存在高端内存。

3、用户进程能访问多少物理内存？内核代码能访问多少物理内存？

32位系统用户进程最大可以访问3GB，内核代码可以访问所有物理内存。

64位系统用户进程最大可以访问超过512GB，内核代码可以访问所有物理内存。

4、高端内存和物理地址、逻辑地址、线性地址的关系？

高端内存只和逻辑地址有关系，和逻辑地址、物理地址没有直接关系。

5、为什么不把所有的地址空间都分配给内核？

若把所有地址空间都给内存，那么用户进程怎么使用内存？怎么保证内核使用内存和用户进程不起冲突？

EPOLL：

什么是epoll

epoll是什么？按照man手册的说法：是为处理大批量句柄而作了改进的poll。当然，这不是2.6内核才有的，它是在2.5.44内核中被引进的(epoll(4) is a new API introduced in Linux kernel 2.5.44)，它几乎具备了之前所说的一切优点，被公认为Linux2.6下性能最好的多路I/O就绪通知方法。

epoll的相关系统调用

epoll只有epoll\_create,epoll\_ctl,epoll\_wait 3个系统调用。

1. int epoll\_create(int size);

创建一个epoll的句柄。自从linux2.6.8之后，size参数是被忽略的。需要注意的是，当创建好epoll句柄后，它就是会占用一个fd值，在linux下如果查看/proc/进程id/fd/，是能够看到这个fd的，所以在使用完epoll后，必须调用close()关闭，否则可能导致fd被耗尽。

2. int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event);

epoll的事件注册函数，它不同于select()是在监听事件时告诉内核要监听什么类型的事件，而是在这里先注册要监听的事件类型。

第一个参数是epoll\_create()的返回值。

第二个参数表示动作，用三个宏来表示：

EPOLL\_CTL\_ADD：注册新的fd到epfd中；

EPOLL\_CTL\_MOD：修改已经注册的fd的监听事件；

EPOLL\_CTL\_DEL：从epfd中删除一个fd；

第三个参数是需要监听的fd。

第四个参数是告诉内核需要监听什么事，struct epoll\_event结构如下：

[cpp] view plaincopyprint?

01.//保存触发事件的某个文件描述符相关的数据（与具体使用方式有关）

02.

03.typedef union epoll\_data {

04.    void \*ptr;

05.    int fd;

06.    \_\_uint32\_t u32;

07.    \_\_uint64\_t u64;

08.} epoll\_data\_t;

09. //感兴趣的事件和被触发的事件

10.struct epoll\_event {

11.    \_\_uint32\_t events; /\* Epoll events \*/

12.    epoll\_data\_t data; /\* User data variable \*/

13.};

events可以是以下几个宏的集合：

EPOLLIN ：表示对应的文件描述符可以读（包括对端SOCKET正常关闭）；

EPOLLOUT：表示对应的文件描述符可以写；

EPOLLPRI：表示对应的文件描述符有紧急的数据可读（这里应该表示有带外数据到来）；

EPOLLERR：表示对应的文件描述符发生错误；

EPOLLHUP：表示对应的文件描述符被挂断；

EPOLLET： 将EPOLL设为边缘触发(Edge Triggered)模式，这是相对于水平触发(Level Triggered)来说的。

EPOLLONESHOT：只监听一次事件，当监听完这次事件之后，如果还需要继续监听这个socket的话，需要再次把这个socket加入到EPOLL队列里

3. int epoll\_wait(int epfd, struct epoll\_event \* events, int maxevents, int timeout);

收集在epoll监控的事件中已经发送的事件。参数events是分配好的epoll\_event结构体数组，epoll将会把发生的事件赋值到events数组中（events不可以是空指针，内核只负责把数据复制到这个events数组中，不会去帮助我们在用户态中分配内存）。maxevents告之内核这个events有多大，这个 maxevents的值不能大于创建epoll\_create()时的size，参数timeout是超时时间（毫秒，0会立即返回，-1将不确定，也有说法说是永久阻塞）。如果函数调用成功，返回对应I/O上已准备好的文件描述符数目，如返回0表示已超时。

epoll工作原理

epoll同样只告知那些就绪的文件描述符，而且当我们调用epoll\_wait()获得就绪文件描述符时，返回的不是实际的描述符，而是一个代表就绪描述符数量的值，你只需要去epoll指定的一个数组中依次取得相应数量的文件描述符即可，这里也使用了内存映射（mmap）技术，这样便彻底省掉了这些文件描述符在系统调用时复制的开销。

另一个本质的改进在于epoll采用基于事件的就绪通知方式。在select/poll中，进程只有在调用一定的方法后，内核才对所有监视的文件描述符进行扫描，而epoll事先通过epoll\_ctl()来注册一个文件描述符，一旦基于某个文件描述符就绪时，内核会采用类似callback的回调机制，迅速激活这个文件描述符，当进程调用epoll\_wait()时便得到通知。

Epoll的2种工作方式-水平触发（LT）和边缘触发（ET）

假如有这样一个例子：

1. 我们已经把一个用来从管道中读取数据的文件句柄(RFD)添加到epoll描述符

2. 这个时候从管道的另一端被写入了2KB的数据

3. 调用epoll\_wait(2)，并且它会返回RFD，说明它已经准备好读取操作

4. 然后我们读取了1KB的数据

5. 调用epoll\_wait(2)......

Edge Triggered 工作模式：

如果我们在第1步将RFD添加到epoll描述符的时候使用了EPOLLET标志，那么在第5步调用epoll\_wait(2)之后将有可能会挂起，因为剩余的数据还存在于文件的输入缓冲区内，而且数据发出端还在等待一个针对已经发出数据的反馈信息。只有在监视的文件句柄上发生了某个事件的时候 ET 工作模式才会汇报事件。因此在第5步的时候，调用者可能会放弃等待仍在存在于文件输入缓冲区内的剩余数据。在上面的例子中，会有一个事件产生在RFD句柄上，因为在第2步执行了一个写操作，然后，事件将会在第3步被销毁。因为第4步的读取操作没有读空文件输入缓冲区内的数据，因此我们在第5步调用 epoll\_wait(2)完成后，是否挂起是不确定的。epoll工作在ET模式的时候，必须使用非阻塞套接口，以避免由于一个文件句柄的阻塞读/阻塞写操作把处理多个文件描述符的任务饿死。最好以下面的方式调用ET模式的epoll接口，在后面会介绍避免可能的缺陷。

   i    基于非阻塞文件句柄

   ii   只有当read(2)或者write(2)返回EAGAIN时才需要挂起，等待。但这并不是说每次read()时都需要循环读，直到读到产生一个EAGAIN才认为此次事件处理完成，当read()返回的读到的数据长度小于请求的数据长度时，就可以确定此时缓冲中已没有数据了，也就可以认为此事读事件已处理完成。

Level Triggered 工作模式

相反的，以LT方式调用epoll接口的时候，它就相当于一个速度比较快的poll(2)，并且无论后面的数据是否被使用，因此他们具有同样的职能。因为即使使用ET模式的epoll，在收到多个chunk的数据的时候仍然会产生多个事件。调用者可以设定EPOLLONESHOT标志，在 epoll\_wait(2)收到事件后epoll会与事件关联的文件句柄从epoll描述符中禁止掉。因此当EPOLLONESHOT设定后，使用带有 EPOLL\_CTL\_MOD标志的epoll\_ctl(2)处理文件句柄就成为调用者必须作的事情。

LT(level triggered)是epoll缺省的工作方式，并且同时支持block和no-block socket.在这种做法中，内核告诉你一个文件描述符是否就绪了，然后你可以对这个就绪的fd进行IO操作。如果你不作任何操作，内核还是会继续通知你 的，所以，这种模式编程出错误可能性要小一点。传统的select/poll都是这种模型的代表．

ET (edge-triggered)是高速工作方式，只支持no-block socket，它效率要比LT更高。ET与LT的区别在于，当一个新的事件到来时，ET模式下当然可以从epoll\_wait调用中获取到这个事件，可是如果这次没有把这个事件对应的套接字缓冲区处理完，在这个套接字中没有新的事件再次到来时，在ET模式下是无法再次从epoll\_wait调用中获取这个事件的。而LT模式正好相反，只要一个事件对应的套接字缓冲区还有数据，就总能从epoll\_wait中获取这个事件。

因此，LT模式下开发基于epoll的应用要简单些，不太容易出错。而在ET模式下事件发生时，如果没有彻底地将缓冲区数据处理完，则会导致缓冲区中的用户请求得不到响应。

图示说明：

Nginx默认采用ET模式来使用epoll。

epoll的优点：

1.支持一个进程打开大数目的socket描述符(FD)

    select 最不能忍受的是一个进程所打开的FD是有一定限制的，由FD\_SETSIZE设置，默认值是2048。对于那些需要支持的上万连接数目的IM服务器来说显然太少了。这时候你一是可以选择修改这个宏然后重新编译内核，不过资料也同时指出这样会带来网络效率的下降，二是可以选择多进程的解决方案(传统的 Apache方案)，不过虽然linux上面创建进程的代价比较小，但仍旧是不可忽视的，加上进程间数据同步远比不上线程间同步的高效，所以也不是一种完美的方案。不过 epoll则没有这个限制，它所支持的FD上限是最大可以打开文件的数目，这个数字一般远大于2048,举个例子,在1GB内存的机器上大约是10万左右，具体数目可以cat /proc/sys/fs/file-max察看,一般来说这个数目和系统内存关系很大。

2.IO效率不随FD数目增加而线性下降

    传统的select/poll另一个致命弱点就是当你拥有一个很大的socket集合，不过由于网络延时，任一时间只有部分的socket是"活跃"的，但是select/poll每次调用都会线性扫描全部的集合，导致效率呈现线性下降。但是epoll不存在这个问题，它只会对"活跃"的socket进行操作---这是因为在内核实现中epoll是根据每个fd上面的callback函数实现的。那么，只有"活跃"的socket才会主动的去调用 callback函数，其他idle状态socket则不会，在这点上，epoll实现了一个"伪"AIO，因为这时候推动力在os内核。在一些 benchmark中，如果所有的socket基本上都是活跃的---比如一个高速LAN环境，epoll并不比select/poll有什么效率，相反，如果过多使用epoll\_ctl,效率相比还有稍微的下降。但是一旦使用idle connections模拟WAN环境,epoll的效率就远在select/poll之上了。

3.使用mmap加速内核与用户空间的消息传递

    这点实际上涉及到epoll的具体实现了。无论是select,poll还是epoll都需要内核把FD消息通知给用户空间，如何避免不必要的内存拷贝就很重要，在这点上，epoll是通过内核于用户空间mmap同一块内存实现的。而如果你想我一样从2.5内核就关注epoll的话，一定不会忘记手工 mmap这一步的。

4.内核微调

这一点其实不算epoll的优点了，而是整个linux平台的优点。也许你可以怀疑linux平台，但是你无法回避linux平台赋予你微调内核的能力。比如，内核TCP/IP协议栈使用内存池管理sk\_buff结构，那么可以在运行时期动态调整这个内存pool(skb\_head\_pool)的大小--- 通过echo XXXX>/proc/sys/net/core/hot\_list\_length完成。再比如listen函数的第2个参数(TCP完成3次握手的数据包队列长度)，也可以根据你平台内存大小动态调整。更甚至在一个数据包面数目巨大但同时每个数据包本身大小却很小的特殊系统上尝试最新的NAPI网卡驱动架构。

linux下epoll如何实现高效处理百万句柄的

开发高性能网络程序时，windows开发者们言必称iocp，linux开发者们则言必称epoll。大家都明白epoll是一种IO多路复用技术，可以非常高效的处理数以百万计的socket句柄，比起以前的select和poll效率高大发了。我们用起epoll来都感觉挺爽，确实快，那么，它到底为什么可以高速处理这么多并发连接呢？

使用起来很清晰，首先要调用epoll\_create建立一个epoll对象。参数size是内核保证能够正确处理的最大句柄数，多于这个最大数时内核可不保证效果。

epoll\_ctl可以操作上面建立的epoll，例如，将刚建立的socket加入到epoll中让其监控，或者把 epoll正在监控的某个socket句柄移出epoll，不再监控它等等。

epoll\_wait在调用时，在给定的timeout时间内，当在监控的所有句柄中有事件发生时，就返回用户态的进程。

从上面的调用方式就可以看到epoll比select/poll的优越之处：因为后者每次调用时都要传递你所要监控的所有socket给select/poll系统调用，这意味着需要将用户态的socket列表copy到内核态，如果以万计的句柄会导致每次都要copy几十几百KB的内存到内核态，非常低效。而我们调用epoll\_wait时就相当于以往调用select/poll，但是这时却不用传递socket句柄给内核，因为内核已经在epoll\_ctl中拿到了要监控的句柄列表。

所以，实际上在你调用epoll\_create后，内核就已经在内核态开始准备帮你存储要监控的句柄了，每次调用epoll\_ctl只是在往内核的数据结构里塞入新的socket句柄。

当一个进程调用epoll\_creaqte方法时，Linux内核会创建一个eventpoll结构体，这个结构体中有两个成员与epoll的使用方式密切相关：

[cpp] view plaincopyprint?

01./\*

02.

03. 171 \* This structure is stored inside the "private\_data" member of the file

04.

05. 172 \* structure and represents the main data structure for the eventpoll

06.

07. 173 \* interface.

08.

09. 174 \*/

10.

11. 175struct eventpoll {

12.

13. 176        /\* Protect the access to this structure \*/

14.

15. 177        spinlock\_t lock;

16.

17. 178

18.

19. 179        /\*

20.

21. 180         \* This mutex is used to ensure that files are not removed

22.

23. 181         \* while epoll is using them. This is held during the event

24.

25. 182         \* collection loop, the file cleanup path, the epoll file exit

26.

27. 183         \* code and the ctl operations.

28.

29. 184         \*/

30.

31. 185        struct mutex mtx;

32.

33. 186

34.

35. 187        /\* Wait queue used by sys\_epoll\_wait() \*/

36.

37. 188        wait\_queue\_head\_t wq;

38.

39. 189

40.

41. 190        /\* Wait queue used by file->poll() \*/

42.

43. 191        wait\_queue\_head\_t poll\_wait;

44.

45. 192

46.

47. 193        /\* List of ready file descriptors \*/

48.

49. 194        struct list\_head rdllist;

50.

51. 195

52.

53. 196        /\* RB tree root used to store monitored fd structs \*/

54.

55. 197        struct rb\_root rbr;//红黑树根节点，这棵树存储着所有添加到epoll中的事件，也就是这个epoll监控的事件

56. 198

57. 199        /\*

58. 200         \* This is a single linked list that chains all the "struct epitem" that

59. 201         \* happened while transferring ready events to userspace w/out

60. 202         \* holding ->lock.

61. 203         \*/

62. 204        struct epitem \*ovflist;

63. 205

64. 206        /\* wakeup\_source used when ep\_scan\_ready\_list is running \*/

65. 207        struct wakeup\_source \*ws;

66. 208

67. 209        /\* The user that created the eventpoll descriptor \*/

68. 210        struct user\_struct \*user;

69. 211

70. 212        struct file \*file;

71. 213

72. 214        /\* used to optimize loop detection check \*/

73. 215        int visited;

74. 216        struct list\_head visited\_list\_link;//双向链表中保存着将要通过epoll\_wait返回给用户的、满足条件的事件

75. 217};

每一个epoll对象都有一个独立的eventpoll结构体，这个结构体会在内核空间中创造独立的内存，用于存储使用epoll\_ctl方法向epoll对象中添加进来的事件。这样，重复的事件就可以通过红黑树而高效的识别出来。

在epoll中，对于每一个事件都会建立一个epitem结构体：

[cpp] view plaincopyprint?

01./\*

02. 130 \* Each file descriptor added to the eventpoll interface will

03. 131 \* have an entry of this type linked to the "rbr" RB tree.

04. 132 \* Avoid increasing the size of this struct, there can be many thousands

05. 133 \* of these on a server and we do not want this to take another cache line.

06. 134 \*/

07. 135struct epitem {

08. 136        /\* RB tree node used to link this structure to the eventpoll RB tree \*/

09. 137        struct rb\_node rbn;

10. 138

11. 139        /\* List header used to link this structure to the eventpoll ready list \*/

12. 140        struct list\_head rdllink;

13. 141

14. 142        /\*

15. 143         \* Works together "struct eventpoll"->ovflist in keeping the

16. 144         \* single linked chain of items.

17. 145         \*/

18. 146        struct epitem \*next;

19. 147

20. 148        /\* The file descriptor information this item refers to \*/

21. 149        struct epoll\_filefd ffd;

22. 150

23. 151        /\* Number of active wait queue attached to poll operations \*/

24. 152        int nwait;

25. 153

26. 154        /\* List containing poll wait queues \*/

27. 155        struct list\_head pwqlist;

28. 156

29. 157        /\* The "container" of this item \*/

30. 158        struct eventpoll \*ep;

31. 159

32. 160        /\* List header used to link this item to the "struct file" items list \*/

33. 161        struct list\_head fllink;

34. 162

35. 163        /\* wakeup\_source used when EPOLLWAKEUP is set \*/

36. 164        struct wakeup\_source \_\_rcu \*ws;

37. 165

38. 166        /\* The structure that describe the interested events and the source fd \*/

39. 167        struct epoll\_event event;

40. 168};

此外，epoll还维护了一个双链表，用户存储发生的事件。当epoll\_wait调用时，仅仅观察这个list链表里有没有数据即eptime项即可。有数据就返回，没有数据就sleep，等到timeout时间到后即使链表没数据也返回。所以，epoll\_wait非常高效。

而且，通常情况下即使我们要监控百万计的句柄，大多一次也只返回很少量的准备就绪句柄而已，所以，epoll\_wait仅需要从内核态copy少量的句柄到用户态而已，如何能不高效？！

那么，这个准备就绪list链表是怎么维护的呢？当我们执行epoll\_ctl时，除了把socket放到epoll文件系统里file对象对应的红黑树上之外，还会给内核中断处理程序注册一个回调函数，告诉内核，如果这个句柄的中断到了，就把它放到准备就绪list链表里。所以，当一个socket上有数据到了，内核在把网卡上的数据copy到内核中后就来把socket插入到准备就绪链表里了。

如此，一颗红黑树，一张准备就绪句柄链表，少量的内核cache，就帮我们解决了大并发下的socket处理问题。执行epoll\_create时，创建了红黑树和就绪链表，执行epoll\_ctl时，如果增加socket句柄，则检查在红黑树中是否存在，存在立即返回，不存在则添加到树干上，然后向内核注册回调函数，用于当中断事件来临时向准备就绪链表中插入数据。执行epoll\_wait时立刻返回准备就绪链表里的数据即可。

epoll的使用方法

那么究竟如何来使用epoll呢？其实非常简单。

通过在包含一个头文件#include <sys/epoll.h> 以及几个简单的API将可以大大的提高你的网络服务器的支持人数。

首先通过create\_epoll(int maxfds)来创建一个epoll的句柄。这个函数会返回一个新的epoll句柄，之后的所有操作将通过这个句柄来进行操作。在用完之后，记得用close()来关闭这个创建出来的epoll句柄。

之后在你的网络主循环里面，每一帧的调用epoll\_wait(int epfd, epoll\_event events, int max events, int timeout)来查询所有的网络接口，看哪一个可以读，哪一个可以写了。基本的语法为：

nfds = epoll\_wait(kdpfd, events, maxevents, -1);

其中kdpfd为用epoll\_create创建之后的句柄，events是一个epoll\_event\*的指针，当epoll\_wait这个函数操作成功之后，epoll\_events里面将储存所有的读写事件。max\_events是当前需要监听的所有socket句柄数。最后一个timeout是 epoll\_wait的超时，为0的时候表示马上返回，为-1的时候表示一直等下去，直到有事件返回，为任意正整数的时候表示等这么长的时间，如果一直没有事件，则返回。一般如果网络主循环是单独的线程的话，可以用-1来等，这样可以保证一些效率，如果是和主逻辑在同一个线程的话，则可以用0来保证主循环的效率。

epoll\_wait返回之后应该是一个循环，遍历所有的事件。

几乎所有的epoll程序都使用下面的框架：

[cpp] view plaincopyprint?

01.for( ; ; )

02.   {

03.       nfds = epoll\_wait(epfd,events,20,500);

04.       for(i=0;i<nfds;++i)

05.       {

06.           if(events[i].data.fd==listenfd) //有新的连接

07.           {

08.               connfd = accept(listenfd,(sockaddr \*)&clientaddr, &clilen); //accept这个连接

09.               ev.data.fd=connfd;

10.               ev.events=EPOLLIN|EPOLLET;

11.               epoll\_ctl(epfd,EPOLL\_CTL\_ADD,connfd,&ev); //将新的fd添加到epoll的监听队列中

12.           }

13.

14.           else if( events[i].events&EPOLLIN ) //接收到数据，读socket

15.           {

16.               n = read(sockfd, line, MAXLINE)) < 0    //读

17.               ev.data.ptr = md;     //md为自定义类型，添加数据

18.               ev.events=EPOLLOUT|EPOLLET;

19.               epoll\_ctl(epfd,EPOLL\_CTL\_MOD,sockfd,&ev);//修改标识符，等待下一个循环时发送数据，异步处理的精髓

20.           }

21.           else if(events[i].events&EPOLLOUT) //有数据待发送，写socket

22.           {

23.               struct myepoll\_data\* md = (myepoll\_data\*)events[i].data.ptr;    //取数据

24.               sockfd = md->fd;

25.               send( sockfd, md->ptr, strlen((char\*)md->ptr), 0 );        //发送数据

26.               ev.data.fd=sockfd;

27.               ev.events=EPOLLIN|EPOLLET;

28.               epoll\_ctl(epfd,EPOLL\_CTL\_MOD,sockfd,&ev); //修改标识符，等待下一个循环时接收数据

29.           }

30.           else

31.           {

32.               //其他的处理

33.           }

34.       }

35.   }