复习文件描述符相关内容！！！！

1. select()函数

select函数用于在非阻塞中，当一个套接字或一组套接字有信号时通知你，系统提供select函数来实现多路复用输入/输出模型，原型：

int select(int maxfd,fd\_set \*rdset,fd\_set \*wrset,fd\_set \*exset,struct timeval \*timeout);

所在的头文件为：#include <sys/time.h> 和#include <unistd.h>

参数maxfd是需要监视的最大的文件描述符值+1；rdset,wrset,exset分别对应于需要检测的可读文件描述符的集合，可写文件描述符的集 合及异常文件描述符的集合。struct timeval结构用于描述一段时间长度，如果在这个时间内，需要监视的描述符没有事件发生则函数返回，返回值为0。

在这些参数中有一个类似于结构体的东西，fd\_set，这是一组文件描述字(fd)的集合，它用一位来表示一个fd。

对于fd\_set类型通过下面四个宏来操作：  
    FD\_ZERO(fd\_set \*fdset) 将指定的文件描述符集清空，在对文件描述符集合进行设置前，必须对其进行初始化，如果不清空，由于在系统分配内存空间后，通常并不作清空处理，所以结果是不可知的。  
    FD\_SET(fd\_set \*fdset) 用于在文件描述符集合中增加一个新的文件描述符。  
    FD\_CLR(fd\_set \*fdset) 用于在文件描述符集合中删除一个文件描述符。  
    FD\_ISSET(int fd,fd\_set \*fdset) 用于测试指定的文件描述符是否在该集合中。

UNIX系统通常会在头文件<sys/select.h>中定义常量FD\_SETSIZE，它是数据类型fd\_set的描述字数量，其值通常是1024，这样就能表示小于1024的fd。

研究了一番关于fd\_set的信息之后，再回到对select函数的理解上来吧：

功能：测试指定的fd可读？可写？有异常条件待处理？  
    readset 用来检查可读性的一组文件描述字。  
    writeset 用来检查可写性的一组文件描述字。  
    exceptset用来检查是否有异常条件出现的文件描述字。(注：不包括错误)  
    timeout  用于描述一段时间长度，如果在这个时间内，需要监视的描述符没有事件发生则函数返回，返回值为0。

    对于select函数的功能简单的说就是对文件fd做一个测试。测试结果有三种可能：  
    1.timeout=NULL（阻塞：select将一直被阻塞，直到某个文件描述符上发生了事件）  
    2.timeout所指向的结构设为非零时间（等待固定时间：如果在指定的时间段里有事件发生或者时间耗尽，函数均返回）  
    3.timeout所指向的结构，时间设为0（非阻塞：仅检测描述符集合的状态，然后立即返回，并不等待外部事件的发生）

返回值：返回对应位为1的fd的总数。注意啦：只有那些可读，可写以及有异常条件待处理的fd位为1。否则为0。

代码分析：

socket s;

fd\_set read\_set;

while(1)

｛

FD\_ZERO(&read\_set);//将你的套节字集合清空

FD\_SET(s, &read\_set);//加入你感兴趣的套节字到集合,这里是一个读数据的套节字s

select(0,&set,NULL,NULL,0);//检查套节字是否可读,0非阻塞直接返回

if(FD\_ISSET(s, &set) //检查s是否在这个集合里面,

{ recv(s,...);}

………

}

理解select模型的关键在于理解fd\_set,为说明方便，取fd\_set长度为1字节，fd\_set中的每一bit可以对应一个文件描述符fd。则1字节长的fd\_set最大可以对应8个fd。  
   （1）执行fd\_set set; FD\_ZERO(&set);则set用位表示0000,0000。  
   （2）若fd＝5,执行FD\_SET(fd,&set);后set变为0001,0000(第5位置为1)

   （3）若再加入fd＝2，fd=1,则set变为0001,0011  
   （4）执行select(6,&set,0,0,0)非阻塞等待  
   （5）若fd=1,fd=2上都发生可读事件，则select返回，此时set变为0000,0011。注意：没有事件发生的fd=5被清空。

基于上面的讨论，可以轻松得出select模型的特点：  
   （1)可监控的文件描述符个数取决与sizeof(fd\_set)的值。

   （2）可以有效突破select可监控的文件描述符上限。  
   （3）将fd加入select监控集的同时，还要再使用一个数据结构array存放到select监控集中的fd，一是用于再select 返回后，array作为源数据和fd\_set进行FD\_ISSET判断。二是select返回后会把以前加入的但并无事件发生的fd清空，则每次开始 select前都要重新从array取得fd逐一加入（FD\_ZERO最先），扫描array的同时取得fd最大值maxfd，用于select的第一个 参数。  
   （4）可见select模型必须在select前循环array（加fd，取maxfd），select返回后循环array（FD\_ISSET判断是否有时间发生）。

使用select函数的过程一般是：  
    先调用宏FD\_ZERO将指定的fd\_set清零，然后调用宏FD\_SET将需要测试的fd加入fd\_set，接着调用函数select测试fd\_set中的所有fd，最后用宏FD\_ISSET检查某个fd在函数select调用后，相应位是否仍然为1。

1. epoll()函数

在linux 没有实现epoll事件驱动机制之前，我们一般选择用select或者poll等IO多路复用的方法来实现并发服务程序。

**select的缺点：**

1. 单个进程能够监视的文件描述符的数量存在最大限制，通常是1024，当然可以更改数量，但由于select采用轮询的方式扫描文件描述符，文件描述符数量越多，性能越差；
2. (在linux内核头文件中，有这样的定义：#define \_\_FD\_SETSIZE    1024)内核 / 用户空间内存拷贝问题，select需要复制大量的句柄数据结构，产生巨大的开销；
3. elect返回的是含有整个句柄的数组，应用程序需要遍历整个数组才能发现哪些句柄发生了事件；
4. select的触发方式是水平触发，应用程序如果没有完成对一个已经就绪的文件描述符进行IO操作，那么之后每次select调用还是会将这些文件描述符通知进程。

拿select模型为例，假设我们的服务器需要支持100万的并发连接，则在\_\_FD\_SETSIZE 为1024的情况下，则我们至少需要开辟1k个进程才能实现100万的并发连接。除了进程间上下文切换的时间消耗外，从内核/用户空间大量的无脑内存拷贝、数组轮询等，是系统难以承受的。因此，基于select模型的服务器程序，要达到10万级别的并发访问，是一个很难完成的任务。

因此，该epoll上场了。

由于epoll的实现机制与select/poll机制完全不同，上面所说的 select的缺点在epoll上不复存在。

设想一下如下场景：有100万个客户端同时与一个服务器进程保持着TCP连接。而每一时刻，通常只有几百上千个TCP连接是活跃的(事实上大部分场景都是这种情况)。如何实现这样的高并发？

在select/poll时代，服务器进程每次都把这100万个连接告诉操作系统(从用户态复制句柄数据结构到内核态)，让操作系统内核去查询这些套接字上是否有事件发生，轮询完后，再将句柄数据复制到用户态，让服务器应用程序轮询处理已发生的网络事件，这一过程资源消耗较大，因此，select/poll一般只能处理几千的并发连接。

epoll的设计和实现与select完全不同。epoll通过在Linux内核中申请一个简易的文件系统(文件系统一般用什么数据结构实现？B+树)。把原先的select/poll调用分成了3个部分：

1）调用epoll\_create()建立一个epoll对象(在epoll文件系统中为这个句柄对象分配资源)

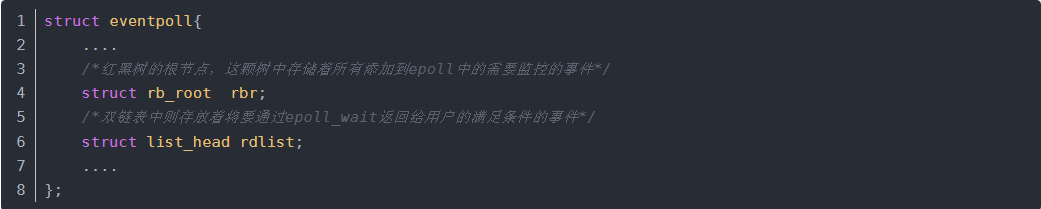
2）调用epoll\_ctl向epoll对象中添加这100万个连接的套接字

3）调用epoll\_wait收集发生的事件的连接

如此一来，要实现上面说是的场景，只需要在进程启动时建立一个epoll对象，然后在需要的时候向这个epoll对象中添加或者删除连接。同时，epoll\_wait的效率也非常高，因为调用epoll\_wait时，并没有一股脑的向操作系统复制这100万个连接的句柄数据，内核也不需要去遍历全部的连接。

**下面来看看Linux内核具体的epoll机制实现思路。**

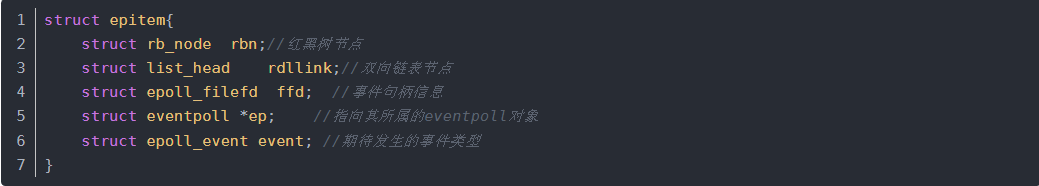
当某一进程调用epoll\_create方法时，Linux内核会创建一个eventpoll结构体，这个结构体中有两个成员与epoll的使用方式密切相关。eventpoll结构体如下所示：



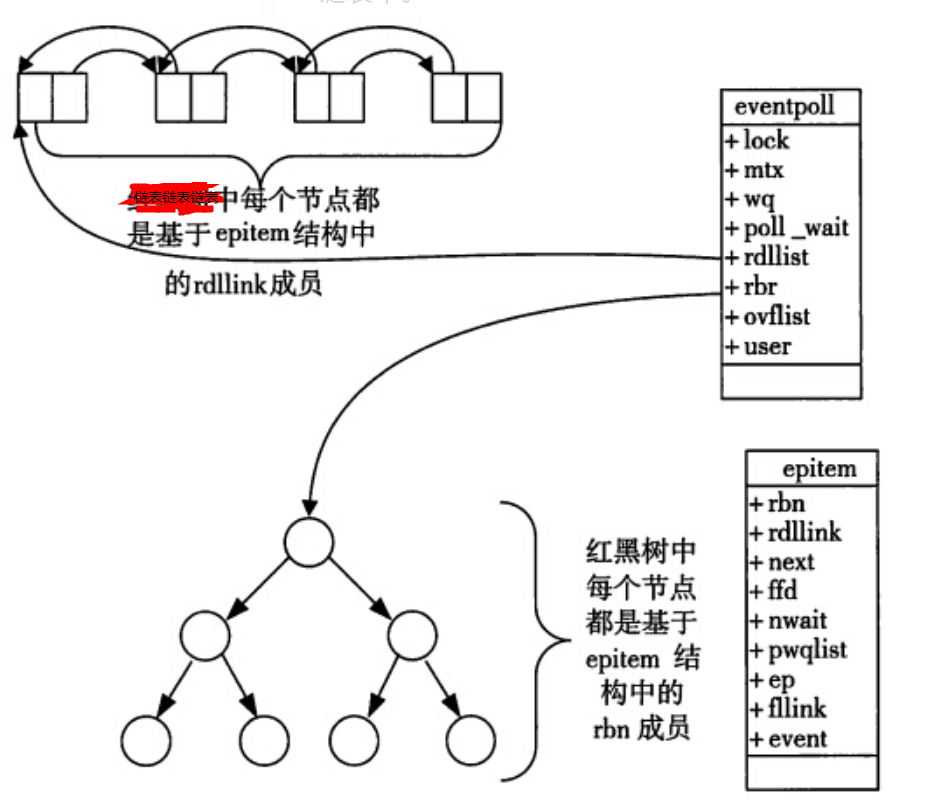
每一个epoll对象都有一个独立的eventpoll结构体，用于存放通过epoll\_ctl方法向epoll对象中添加进来的事件。这些事件都会挂载在红黑树中，如此，重复添加的事件就可以通过红黑树而高效的识别出来(红黑树的插入时间效率是lgn，其中n为树的高度)。

而所有添加到epoll中的事件都会与设备(网卡)驱动程序建立回调关系，也就是说，当相应的事件发生时会调用这个回调方法。这个回调方法在内核中叫ep\_poll\_callback,它会将发生的事件添加到rdlist双链表中。

在epoll中，对于每一个事件，都会建立一个epitem结构体，如下所示：



当调用epoll\_wait检查是否有事件发生时，只需要检查eventpoll对象中的rdlist双链表中是否有epitem元素即可。如果rdlist不为空，则把发生的事件复制到用户态，同时将事件数量返回给用户。



从上面的讲解可知：通过红黑树和双链表数据结构，并结合回调机制，造就了epoll的高效。

OK，讲解完了Epoll的机理，我们便能很容易掌握epoll的用法了。一句话描述就是：三步曲。

第一步：epoll\_create()系统调用。此调用返回一个句柄，之后所有的使用都依靠这个句柄来标识。

第二步：epoll\_ctl()系统调用。通过此调用向epoll对象中添加、删除、修改感兴趣的事件，返回0标识成功，返回-1表示失败。

第三部：epoll\_wait()系统调用。通过此调用收集收集在epoll监控中已经发生的事件。

三、select与epoll介绍和对比

https://blog.csdn.net/chenlycly/article/details/52191660