知识点1【epoll的概述】

1、epoll的概述

2、epoll的特点

知识点2【epoll的API】(了解)

1、epoll操作过程需要4个函数接口

2、epoll_create函数 创建树

功能：

参数：

返回值：

3、epoll_ctl函数 上树、下树

功能：

参数：

epfd: epoll 专用的文件描述符，epoll_create()的返回值

op: 表示动作，用三个宏来表示：

struct epoll_event中events 可以是一下几个宏的集合

返回值：

4、epoll_wait函数 监听

功能：

参数：

返回值：

5、close函数 关闭epoll专用的文件描述符

6、epoll对于文件描述操作的两种模式

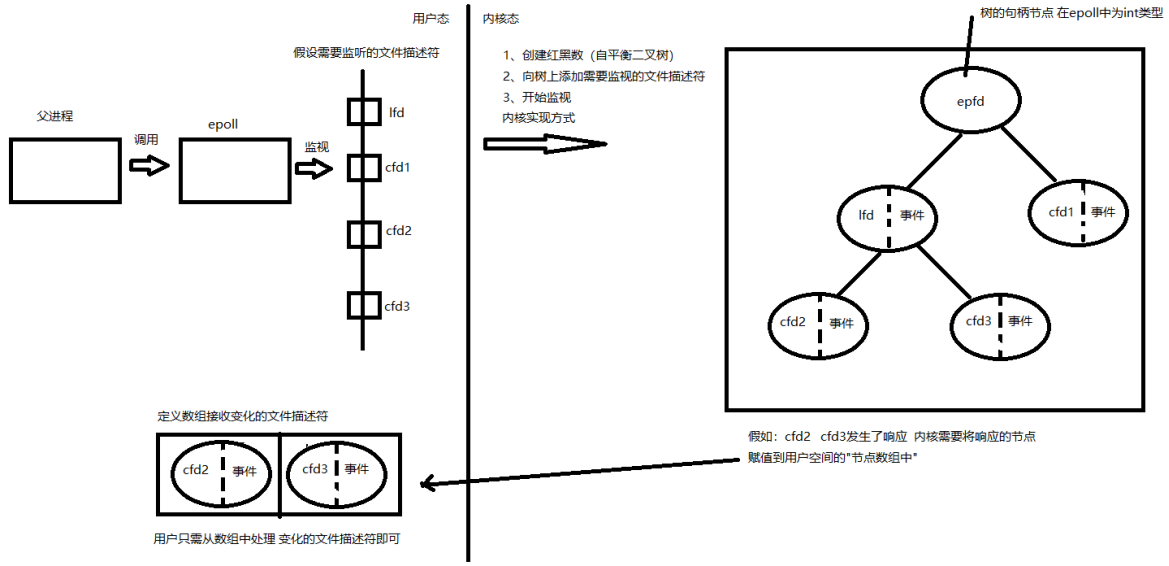
LT(电平触发)和ET(边沿触发)的区别

知识点3【epoll的tcp echo 并发服务器】(了解)

知识点1【epoll的概述】

1、epoll的概述

epoll 是在 2.6 内核中提出的，是之前的 select() 和 poll() 的增强版本。相对于 select() 和 poll() 来说，epoll 更加灵活，没有描述符限制。epoll 使用一个文件描述符管理多个描述符，将用户关系的文件描述符的事件存放到内核的一个事件表中，这样在用户空间和内核空间的 copy 只需一次。



2、epoll的特点

1、没有文件描述符的限制

2、需要监听的描述符已上树不需要再次上树

3、监视到文件描述符变化后返回的是变化的文件描述符

知识点2【epoll的API】(了解)

1、epoll操作过程需要4个函数接口

```
#include <sys/epoll.h>
int epoll_create(int size);
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
int close(int epfd);
```

2、epoll_create函数 创建树

```
int epoll_create(int size);
```

功能：

该函数生成一个epoll专用的文件描述符（其余的接口函数一般都用使用这个专用的文件描述符）

参数：

size: 用来告诉内核这个监听的数目一共有多大，参数 size 并不是限制了 epoll 所能监听的描述符最大个数，只是对内核初始分配内部数据结构的一个建议。自从 linux 2.6.8 之后，size 参数是被忽略的，也就是说可以填只有大于 0 的任意值。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

返回值：

成功：epoll的专用的文件描述符

失败：-1

3、epoll_ctl函数 上树、下树

```
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
```

功能：

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

参数：

epfd: epoll 专用的文件描述符，epoll_create()的返回值

op: 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；

fd: 需要监听的文件描述符

event: 告诉内核要监听什么事件，struct epoll_event 结构如下：

```
// 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）
typedef union epoll_data {
 void *ptr;
 int fd;
 __uint32_t u32;
 __uint64_t u64;
} epoll_data_t;
```

```
8
9  // 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  };
```

struct epoll_event中events 可以是一下几个宏的集合

```
1  EPOLLIN ：表示对应的文件描述符可以读（包括对端 SOCKET 正常关闭）；
2  EPOLLOUT：表示对应的文件描述符可以写；
3  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）；
4  EPOLLERR：表示对应的文件描述符发生错误；
5  EPOLLHUP：表示对应的文件描述符被挂断；
6  EPOLLET ：将 EPOLL 设为边缘触发(Edge Triggered)模式，
7  这是相对于水平触发(Level Triggered)来说的。
8  EPOLLONESHOT：只监听一次事件，当监听完这次事件之后，
9   如果还需要继续监听这个 socket 的话，
10   需要再次把这个 socket 加入到 EPOLL 队列里
```

返回值：

成功：0

失败：-1

4、epoll_wait函数 监听

```
1  int epoll_wait(int epfd, struct epoll_event *events, int maxevents,\
2   int timeout);
```

功能：

等待事件的产生，收集在 epoll 监控的事件中已经发送的事件，类似于 select() 调用

参数：

epfd: epoll 专用的文件描述符，epoll_create()的返回值

events: 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: maxevents 告之内核这个 events 有多大 。

timeout: 超时时间，单位为毫秒，为 -1 时，函数为阻塞

返回值：

成功：返回需要处理的事件数目，如返回 0 表示已超时。

失败：-1

5、close函数 关闭epoll专用的文件描述符

```
int close(int epfd)
```

在用完之后，记得用close()来关闭这个创建出来的epoll句柄

6、epoll对于文件描述操作的两种模式

LT（level trigger）电平触发和 ET（edge trigger）边沿触发，默认为LT模式

LT(电平触发)和ET(边沿触发)的区别

LT 模式：当 epoll_wait 检测到描述符事件发生并将此事件通知应用程序，应用程序可以不立即处理该事件。下次调用 epoll_wait 时，会再次响应应用程序并通知此事件。

ET 模式：当 epoll_wait 检测到描述符事件发生并将此事件通知应用程序，应用程序必须立即处理该事件。如果不处理，下次调用 epoll_wait 时，不会再次响应应用程序并通知此事件

ET 模式在很大程度上减少了 epoll 事件被重复触发的次数，因此效率要比 LT 模式高。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

知识点3【epoll的tcp echo 并发服务器】(了解)

```
#include <stdio.h>
#include "wrap.h"
#include <sys/epoll.h>
int main(int argc, char const *argv[])
{
```

```
6       //创建监听套接字 绑定 端口复用
7       int lfd = tcp4bind(8000, NULL);
8
9       //服务器监听
10       Listen(lfd, 128);
11
12       //创建树
13       int epfd = epoll_create(1);
14
15       //将lfd添加上树
16       struct epoll_event ev, evs[1024];
17       ev.events = EPOLLIN;//读事件
18       ev.data.fd = lfd;
19       epoll_ctl(epfd, EPOLL_CTL_ADD, lfd, &ev);
20
21       //循环监视 树中的文件描述符
22       while (1)
23       {
24           int n = epoll_wait(epfd, evs, 1024, -1);
25           if(n < 0)
26           {
27               perr_exit("epoll");
28           }
29           else if(n>=0)//有文件描述符准备完毕
30           {
31               int i = 0;
32               for (i = 0; i < n; i++)
33               {
34                   int fd = evs[i].data.fd;
35                   //如果是lfd变化，并且是读事件发生 就处理
36                   if( (fd ==lfd) && (evs[i].events&EPOLLIN) )
37                   {
38                       struct sockaddr_in cliaddr;
39                       socklen_t len = sizeof(cliaddr);
40                       char ip[16]="";
41                       int cfd = Accept(lfd,
(struct sockaddr*)&cliaddr,&len);
42                       printf("client ip=%s port=%d\n",
43                               inet_ntop(AF_INET,&cliaddr.sin_addr.s_addr,i
p,16),
44                               ntohs(cliaddr.sin_port));
```

```
45                      //将cfd上树
46                      ev.data.fd = cfd;
47                      ev.events = EPOLLIN;
48                      epoll_ctl(epfd,EPOLL_CTL_ADD,cfd,&ev);
49                  }
50                  else if(evs[i].events&EPOLLIN)
51                  {
52                      //与客户端连接的套接字准备就绪
53                      char buf[1500]="";
54                      int count = Read(fd,buf,sizeof(buf));
55                      if(count <= 0)
56                      {
57                          printf("error or client close\n");
58                          close(fd);
59                          //套接字关闭需要下树
60                          epoll_ctl(epfd,EPOLL_CTL_DEL,fd,&evs[i]);
61                      }
62                      else
63                      {
64                          printf("%s\n",buf);
65                          Write(fd,buf,count);
66                      }
67                  }
68              }
69
70          }
71      }
72
73      close(epfd);
74      return 0;
75  }
```