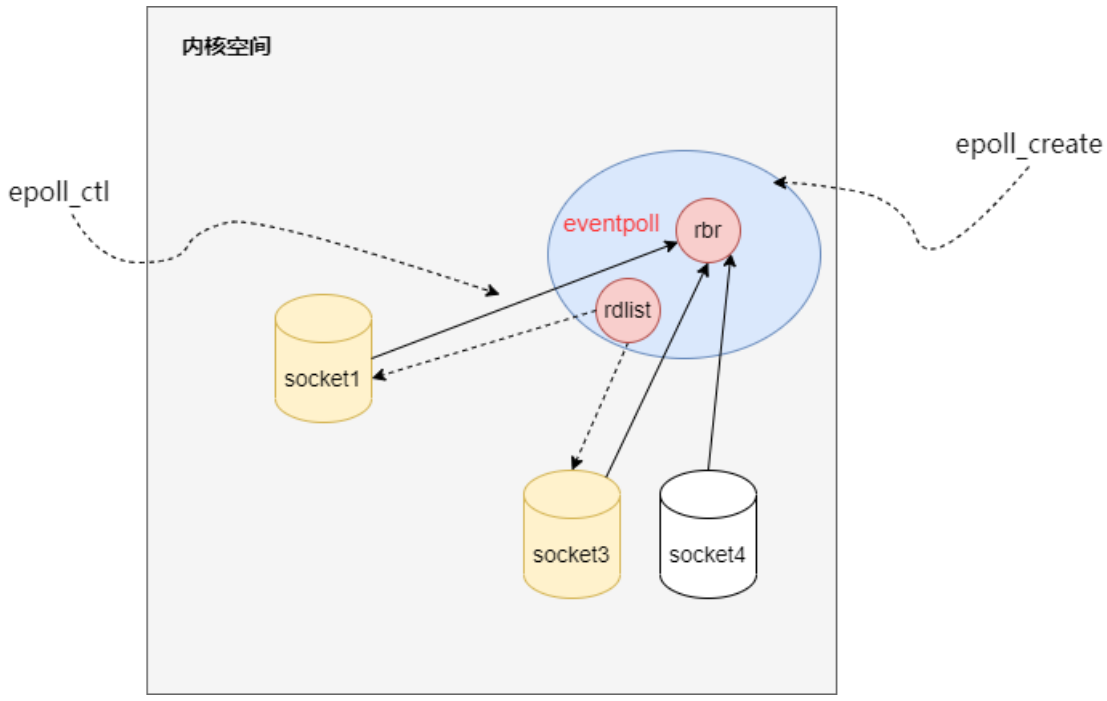
epoll

概述

epoll 全称 eventpoll，是 linux 内核实现IO多路复用的一个实现。其可被用于代替 POSIX select 和 poll 系统调用，并且在处理大量请求时有很好的性能。

原理

通过epoll_creat会在内核空间中创建一个eventpoll对象。创建epoll对象后，可以用epoll_ctl添加或删除所要监听的socket，这些套接字被放在一个名为rbr的红黑树中。在执行epoll_ctl时，除了把socket放到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，当socket收到数据后，中断程序会中断eventpoll，把收到数据的套接字放到rdlist链表里，让rdlist引用这些套接字，如下图所示。



eventpoll 结构

```
struct eventpoll {
    /*
     * This mutex is used to ensure that files are not removed
     * while epoll is using them. This is held during the event
     * collection loop, the file cleanup path, the epoll file exit
```

```
     * code and the ctl operations.
     */
    struct mutex mtx;

    /* Wait queue used by sys_epoll_wait() */
    wait_queue_head_t wq;

    /* Wait queue used by file->poll() */
    wait_queue_head_t poll_wait;

    /* List of ready file descriptors */
    struct list_head rdllist;

    /* Lock which protects rdllist and ovflist */
    rwlock_t lock;

    /* RB tree root used to store monitored fd structs */
    struct rb_root_cached rbr;

    /*
     * This is a single linked list that chains all the "struct
epitem" that
     * happened while transferring ready events to userspace w/out
     * holding ->lock.
     */
    struct epitem *ovflist;

    /* wakeup_source used when ep_scan_ready_list is running */
    struct wakeup_source *ws;

    /* The user that created the eventpoll descriptor */
    struct user_struct *user;

    struct file *file;

    /* used to optimize loop detection check */
    u64 gen;
    struct hlist_head refs;

#ifdef CONFIG_NET_RX_BUSY_POLL
    /* used to track busy poll napi_id */
    unsigned int napi_id;
```

```
#endif

#ifdef CONFIG_DEBUG_LOCK_ALLOC
    /* tracks wakeup nests for lockdep validation */
    u8 nests;
#endif
};
```

触发模式

- 水平触发（LT）：只要套接字可读/可写，epoll_wait都会将描述符返回；
- 边缘触发（ET）：当套接字的缓冲状态发生变化时返回。即只有当socket由不可写到可写或由不可读到可读，才会返回。如果这次没有把数据全部读写完 (如读写缓冲区太小)，那么下次调用 epoll_wait () 时，它不会通知你，直到该文件描述符上出现第二次可读写事件才会通知你，这次通知的内容包括上次未取完的数据。

两者比较

- LT模式同时支持block和no-block socket，ET模式只支持no-block socket；
- 采用边缘触发模式有可能造成饥饿问题；

相关接口

epoll_creat 函数

```
/**
*description: 创建一个epoll的句柄
*param size: 指定监听的数目（从linux2.6.8开始，该参数被忽略，其值大于0就行）
*return: 成功则该函数会返回一个文件描述符，并占用一个fd值，失败返回-1
*/
int epoll_create(int size);
```

epoll_ctl 函数

```
/**
*description:用于注册事件
*param epfd: epoll_create()创建的epoll文件描述符
*param op: 操作动作
*param fd: 需要监听的描述符
*param event: 需要监听的事件集合
*return :成功返回0，失败返回-1
*/
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
```

op的动作由以下宏指定：

- 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；

epoll_event 结构如下：

```
//用户自定义数据
typedef union epoll_data
{
    void        *ptr;
    int          fd;
    __uint32_t   u32;
    __uint64_t   u64;
} epoll_data_t;

struct epoll_event {
    __uint32_t events; /* Epoll events */
    epoll_data_t data; /* User data variable */
};
```

events是以下几个宏的集合:

- EPOLLIN //表示对应的文件描述符可以读（输入数据）
- EPOLLOUT //表示对应的文件描述符可以写（输出数据）
- EPOLLPRI //表示对应的文件描述符有紧急的数据可读
- 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（默认为水平触发方式）。

- 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。

epoll_wait 函数

```
/**
*description: 该函数用于等待epollevent中的fd集合的就绪事件
*param epfd: epoll_create()创建的epoll文件描述符
*param enents: 需要监听的事件集合
*param maxevents: 最多maxevents数量的事件集合会被返回
*param timeout: 超时时间，单位为毫秒；指定为-1没有超时时间，指定为0则立即返回并返回0
*return: 就绪事件的个数,出现错误则返回-1
*/
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
```

测试

server

```
//server.h
#ifndef SERVER_H
#define SERVER_H

class Server
{
    private:
    char *ip="127.0.0.1";
    int port=37992;

    public:
    Server();
};
#endif

//server.cpp
#include <iostream>
#include <string.h>
#include "Server.h"
#include <sys/socket.h>
```

```cpp
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <sys/epoll.h>

using namespace std;

Server::Server()
{
    int socket_fd, conn_fd;
    sockaddr_in addr;

    bzero(&addr,sizeof(addr));
    addr.sin_family=AF_INET;
    addr.sin_port=htons(port);

    if (inet_pton(AF_INET, ip, &addr.sin_addr) == -1)
    {
        cout << "inet_pton error." << endl;
        return;
    }

    //创建套接字
    if ((socket_fd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        cout << "create socket_fd error." << endl;
        return;
    }

    //绑定地址信息
    if ((bind(socket_fd, (sockaddr *)&addr, sizeof(addr))) == -1)
    {
        cout << "bind error." << endl;
        return;
    }

    //监听套接字
    if (listen(socket_fd, 20) == -1)
    {
        cout << "listen error." << endl;
        return;
    }
```

```
    int epoll_fd;
    epoll_event event;
    event.events = EPOLLIN|EPOLLET;
    event.data.fd = socket_fd;

    //创建epoll
    if ((epoll_fd = epoll_create(10)) == -1)
    {
        cout << "create epoll error." << endl;
        return;
    }

    //添加fd
    if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, socket_fd, &event) ==
-1)
    {
        cout << "epoll_ctl error." << endl;
        return;
    }

    epoll_event events[5];
    while (true)
    {
        int ret_num = epoll_wait(epoll_fd, events, 5, -1);
        if (ret_num == -1)
        {
            cout << "epoll_wait error." << endl;
            return;
        }
        for (int i = 0; i < ret_num; i++)
        {
            if (events[i].data.fd == socket_fd)
            {
                if ((conn_fd = accept(socket_fd, NULL, NULL)) ==
-1)
                {
                    cout << "accept error." << endl;
                }
                event.data.fd = conn_fd;
                if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, conn_fd,
&event) == -1)
```

```
                {
                    cout << "epoll_ctl error." << endl;
                    return;
                }
            }
            else if(events[i].events)
            {
                char buffer[100];
                bzero(buffer,sizeof(buffer));
                conn_fd=events[i].data.fd;
                int n = recv(conn_fd, buffer, sizeof(buffer), 0);

                if(n>0)
                {
                    cout<< buffer << endl;
                }
                else
                {
                    epoll_ctl(epoll_fd, EPOLL_CTL_DEL,
events[i].data.fd, &event);
                }

            }
        }
    }
    close(socket_fd);
}
```

cient

```
//cient.h
#ifndef CIENT_H
#define CIENT_H
class Cient
{
    private:
    char *ip="127.0.0.1";
    int port=37992;

    public:
```

```
    void sendData();
};
#endif

//cient.cpp
#include<iostream>
#include<string.h>
#include"Cient.h"
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<unistd.h>
#include<string>
#include<sys/wait.h>

using namespace std;

#define MAX_MESSAGE_LENGTH

void Cient::sendData()
{
    int socket_fd,bind_ret;
    string buffer_in;

    //地址信息初始化
    sockaddr_in addr;
    bzero(&addr,sizeof(addr));
    addr.sin_family=AF_INET;
    addr.sin_port=htons(port);
    if(inet_pton(AF_INET,ip,&addr.sin_addr)==-1)
    {
        cout<<"inet_pton error."<<endl;
        return;
    }

    //创建TCP套接字
    if((socket_fd=socket(AF_INET,SOCK_STREAM,0))==-1)
    {
        cout<<"create socket error."<<endl;
        return ;
    }
```

```
    //连接服务器
    if(connect(socket_fd,(sockaddr*)&addr,sizeof(addr))==-1)
    {
        cout<<"connect error."<<endl;
        cout<<errno<<endl;
        return ;
    }
    buffer_in="hello world.";
    //发送数据
    if(send(socket_fd,
(char*)buffer_in.data(),buffer_in.size(),0)==-1)
    {
        cout<<"send error."<<endl;
        return ;
    }
    close(socket_fd);
}
```