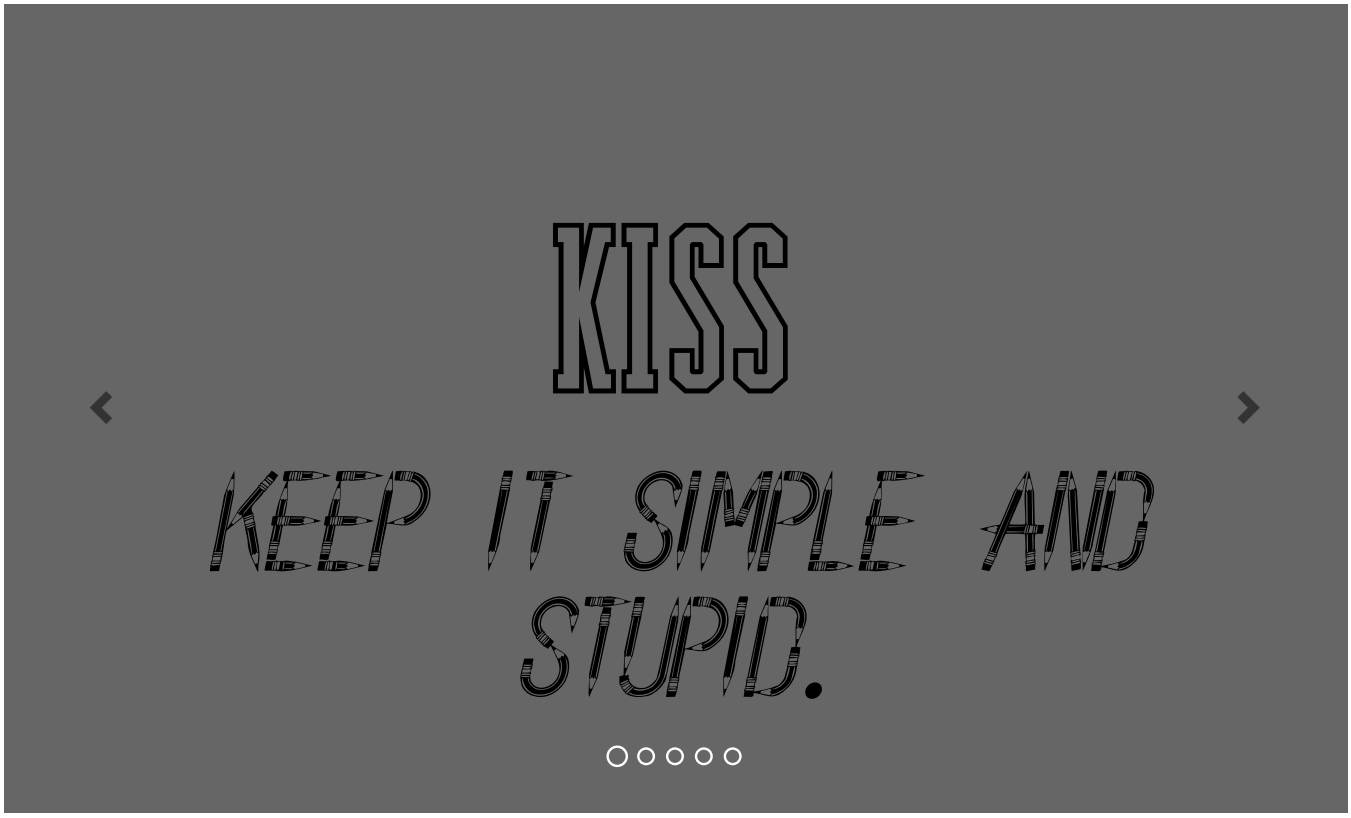

Linux C 网络编程

2017-08-21 Monday program , linux

简单介绍下，在 Linux C 中进行网络编程时常用到的一些技巧。

结构体

在网络编程时，可以看到多种结构体，例如 `struct sockaddr` 、 `struct` ，这些结构体的大小一致，可以直接用来相互转换。

一般来说， `struct sockaddr` 是通用的 socket 地址，其定义如下：

```
struct sockaddr {
        unsigned short sa_family;
        char sa_data[14];
};

struct sockaddr_in {
        short int            sin_family;
        unsigned short int   sin_port;
        struct in_addr       sin_addr;
        unsigned char        sin_zero[8];
};
```

其中 `struct in_addr` 就是 32 位 IP 地址。

```
struct in_addr {
        unsigned long s_addr;
};
```

在使用的时候，一般使用 `struct sockaddr_in` 作为函数 (如 `bind()`) 的参数传入，使用时再转换为 `struct sockaddr` 即可，毕竟都是 16 个字符长。

使用示例如下：

```
int sockfd;
struct sockaddr_in addr;

addr.sin_family = AF_INET;
addr.sin_port = htons(MYPORT);
addr.sin_addr.s_addr = inet_addr("192.168.0.1");
bzero(&(addr.sin_zero), 8);

sockfd = socket(AF_INET, SOCK_STREAM, 0);
bind(sockfd, (struct sockaddr *)&addr, sizeof(struct sockaddr));
```

getaddrinfo

`getaddrinfo()` 函数的前身是做 DNS 解析的 `gethostbyname()` 函数，现在通过 `getaddrinfo()` 可以做 DNS 解析以及 Service Name 查询，如下是其声明：

```
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>

int getaddrinfo(const char *node, const char *service,
        const struct addrinfo *hints,
        struct addrinfo **res);
void freeaddrinfo(struct addrinfo *res);
const char *gai_strerror(int errcode);
```

其中 node 可以是域名、IP，如 `"www.example.com"`；而 service 可以是 `"http"` 或者端口号，其定义在 `/etc/services` 文件中。

使用场景

通常有两种使用方式：A) 建立 Server 监听本机所有的 IP 地址；B) 作为客户端链接到服务器，需要解析服务器的地址信息。

```
int status;
struct addrinfo hints; /* 指定入参配置 */
struct addrinfo *res;  /* 返回结果 */

memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_UNSPEC;     /* 返回IPv4以及IPV6,也可以指定 AF_INET 或 AF_INET6 */
hints.ai_socktype = SOCK_STREAM; /* 使用TCP协议 */
hints.ai_flags = AI_PASSIVE;     /* 返回结果中会填写本地的IP地址 */

if ((status = getaddrinfo(NULL, "3490", &hints, &servinfo)) != 0) {
        fprintf(stderr, "getaddrinfo error: %s\n", gai_strerror(status));
        exit(1);
}
```

其中 `flags` 参数设置为 `AI_PASSIVE` 表示获取本地 IP 地址，这样在调用函数时，第一个参数可以指定为 `NULL` 。

关于客户端的使用可以直接参考如下的示例。

```
#define _POSIX_SOURCE

#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>

#include <netdb.h>
#include <arpa/inet.h>
#include <netinet/in.h>

int main(int argc, char **argv)
{
        int status;
        struct addrinfo hints, *res, *this;
        char ipaddr[INET6_ADDRSTRLEN];

        if (argc != 2) {
                fprintf(stderr, "usage: showip hostname\n");
                return 1;
        }

        memset(&hints, 0, sizeof hints);
        hints.ai_family = AF_UNSPEC;     /* AF_INET(IPv4) AF_INET6(IPv6) */
        hints.ai_socktype = SOCK_STREAM; /* TCP stream sockets */

        if ((status = getaddrinfo(argv[1], NULL, &hints, &res))) {
                fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(status));
                return 2;
        }

        printf("IP addresses for %s:\n\n", argv[1]);

        for(this = res; this != NULL; this = this->ai_next) {
                void *addr;
                char *ipver;

                if (this->ai_family == AF_INET) { /* IPv4 */
                        struct sockaddr_in *ipv4;

                        ipv4 = (struct sockaddr_in *)this->ai_addr;
                        addr = &(ipv4->sin_addr);
                        ipver = "IPv4";
                } else { /* IPv6 */
                        struct sockaddr_in6 *ipv6;

                        ipv6 = (struct sockaddr_in6 *)this->ai_addr;
                        addr = &(ipv6->sin6_addr);
                        ipver = "IPv6";
                }

                /* convert the IP to a string and print it */
                inet_ntop(this->ai_family, addr, ipaddr, sizeof(ipaddr));
                printf("%s:\t%s\n", ipver, ipaddr);
        }

        return 0;
}
```

IP 地址

IP 有两种表达方式，分别为点分十进制(Numbers and Dots Notation) 以及整形(Binary Data) 。

对于 IPv4 实际上是一个 32bit 的整形，假设地址为 `A.B.C.D` ，那么对应的整形是 `A<<24 + B<<16 + C<<8 + D` 。

在 MySQL 中，可以通过 `SELECT INET_ATON('192.168.1.38');` 函数将点分格式转化为整形，然后再通过 `SELECT INET_NTOA(3232235814);` 执行反向转换。

IPv6

IPv6 采用的是 128 位，通常以 16 位为一组，每组之间以冒号 ':' 分割，总共分为 8 组，例如 `2001:0db8:85a3:08d3:1319:8a2e:0370:7344` 。

C 语言使用

对于 C 语言中的地址转换函数，也就是 BSD 网络软件包，可通过 `inet_addr()`、 `inet_aton()` 和 `inet_ntoa()` 三个函数用于二进制地址格式与点分十进制之间的相互转换，但是仅仅适用于 IPv4。

另外，两个新函数 `inet_ntop()` 和 `inet_pton()` 具有相似的功能，字母 p 代表 presentation，字母 n 代表 numeric，并且同时支持 IPv4 和 IPv6 。

```
#include <sys/socket.h>

//----- 将点分地址转换成网络字节序的IP地址
in_addr_t inet_addr(const char *strptr); /* INADDR_NONE: ERROR */
int inet_aton(const char *strptr, struct in_addr *addrptr); /* 1:OK, 0:ERROR */
int inet_pton(int family, const char *strptr, void *addrptr); /* 1:OK, 0:INVALID, -1:FAILED *

//----- 将点分地址转换成主机字节序的IP地址
in_addr_t inet_network(const char *cp);

//----- 网络字节序IP转化点分十进制
char* inet_ntoa(struct in_addr inaddr);
const char* inet_ntop(int family, const void *addrptr, char *strptr, size_t len); /* NULL: ER
```

`inet_addr()` 与 `inet_aton()` 不同在于其返回值为转换后的 32 位网络字节序二进制值，不过这样会存在一个问题，返回的有效 IP 地址应该为 `0.0.0.0` 到 `255.255.255.255` ，如果函数出错，返回常量值 `INADDR_NONE` (一般为 `0xFFFFFFFF`)，也就是 `255.255.255.255` (IPv4 的有限广播地址) 。

对于 `inet_ntop()` 和 `inet_pton()` 两个函数，family 参数可以是 `AF_INET` 或者 `AF_INET6` ，如果不是这两个会返回错误，且将 `errno` 置为 `EAFNOSUPPORT` 。

缓冲区的大小在 `<netinet/in.h>` 中定义：

```
#define INET_ADDRSTRLEN  16    /* for IPv4 dotted-decimal */
#define INET6_ADDRSTRLEN 46    /* for IPv6 hex string */
```

如果缓冲区无法容纳表达格式结果 (包括空字符)，则返回一个空指针，并置 errno 为 `ENOSPC` 。

```
#include <unistd.h>
#include <string.h>

#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main(void)
{
        char ip[] = "255.0.0.1";

        in_addr_t rc;  /* 一般通过 typedef uint32_t in_addr_t; 定义 */
        int status;
        struct in_addr addr;

        rc = inet_addr(ip); /* 返回网络字节序 */
        if (rc == 0xffffffff) {
                fprintf(stderr, "Format error '%s'\n", ip);
                return -1;
        }
        fprintf(stdout, "inet_addr() ==> 0x%x\n", rc);

        rc = inet_network(ip); /* 返回主机字节序 */
        if (rc == 0xffffffff) {
                fprintf(stderr, "Format error '%s'\n", ip);
                return -1;
        }
        fprintf(stdout, "inet_network() ==> 0x%x\n", rc);

        status = inet_aton(ip, &addr);
        if (status == 0) {
                fprintf(stderr, "Format error '%s'\n", ip);
                return -1;
        }
        fprintf(stdout, "inet_aton() rc(%d) ==> 0x%x\n", status, addr.s_addr);

        /* Support IPv4(AF_INET) and IPv6(AF_INET6) */
        status = inet_pton(AF_INET, ip, &addr.s_addr);
        if (status == 0) {
                fprintf(stderr, "Format error '%s'\n", ip);
                return -1;
        }
        fprintf(stdout, "inet_aton() rc(%d) ==> 0x%x\n", status, addr.s_addr);

        char str[INET_ADDRSTRLEN];
        if(inet_ntop(AF_INET, &addr.s_addr, str, sizeof(str)) == NULL) {
                fprintf(stderr, "Format error 0x%x\n", addr.s_addr);
                return -1;
        }
        fprintf(stdout, "inet_network() ==> %s\n", str);

        return 0;
}
```

获取地址

`getpeername()` 用于获取与某个套接字关联的对端地址， `accept()` 在接收连接的时候也会获取对端的地址， `getsockname()` 用于获取本地地址。

```
// Server
#include <stdio.h>
#include <unistd.h>
#include <arpa/inet.h>

int main()
{
        int svrfd, clifd;
        struct sockaddr_in addr;

        svrfd = socket(AF_INET, SOCK_STREAM, 0);
        addr.sin_family = AF_INET;
        addr.sin_addr.s_addr =htonl(INADDR_ANY);
        addr.sin_port = htons(8888);

        bind(svrfd, (const struct sockaddr *)&addr, sizeof(struct sockaddr_in));
        listen(svrfd, 5);

        struct sockaddr_in addrcli, peeraddr;
        socklen_t len = sizeof(struct sockaddr_in);
        clifd = accept(svrfd, (struct sockaddr *)&addrcli, &len);

        printf("Client #%d ip=%s port=%d\n", clifd,
                inet_ntoa(addrcli.sin_addr), ntohs(addrcli.sin_port));

        len = sizeof(struct sockaddr_in);
        getpeername(clifd, (struct sockaddr *)&peeraddr, &len);
        printf("Peer #%d ip=%s port=%d\n", clifd,
                inet_ntoa(peeraddr.sin_addr), ntohs(peeraddr.sin_port));

        getchar();
        close(svrfd);
        close(clifd);
        return 0;
}
```

```
// Client
#include <stdio.h>
#include <unistd.h>
#include <arpa/inet.h>

int main()
{
        int rc;
        int sockfd;

        struct sockaddr_in addr;
        addr.sin_addr.s_addr = inet_addr("127.0.0.1");
        addr.sin_family = AF_INET;
        addr.sin_port = htons(8888);

        printf("Server ip=%s port=%d\n",
                inet_ntoa(addr.sin_addr), ntohs(addr.sin_port));

        sockfd = socket(AF_INET, SOCK_STREAM, 0);
        rc = connect(sockfd, (const struct sockaddr *)&addr, sizeof(struct sockaddr_in));

        struct sockaddr_in svraddr;
        socklen_t len = sizeof(struct sockaddr_in);
        getsockname(sockfd,(struct sockaddr *)&svraddr, &len);
        printf("Local #%d ip=%s port=%d\n", sockfd,
                inet_ntoa(svraddr.sin_addr), ntohs(svraddr.sin_port));

        getchar();
        close(sockfd);
        return 0;
}
```

直接编译运行，然后通过 `netstat -atunp | grep 8888` 查看。

非阻塞链接

对于面向连接的 socket 类型，如 SOCK_STREAM，在通过 `connect()` 函数建立链接时，对于 TCP 需要三次握手过程。

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
```

正常返回 0 ，失败返回 -1 并设置 errno 标示失败的原因，常见原因是主机不可达或超时。

对于 TCP 套接字，在通过 connect() 函数建立链接时需要经过三次握手，Linux 内核默认的超时时间是 75s ，如果是阻塞模式，那么 connect() 会等到链接建立成功或者超时失败，这也就导致如果服务异常，每个客户端都要等待 75s 才可能退出。

使用非阻塞 connect 时需要注意的问题是：

1. 很可能 调用 connect 时会立即建立连接（比如，客户端和服务端在同一台机子上），必须处理这种情况。
2. POSIX 定义了两个与非阻塞相关的内容：
    - 成功建立连接时，socket 描述字变为可写，报错可以通过 getsockopt() 获取。
    - 连接建立失败时，socket 描述字既可读又可写同时报错，可以优先检查报错退出。

```
#include <sys/epoll.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/select.h>
#include <fcntl.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include <errno.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

int select_version(int sockfd)
{
        fd_set rset, wset;

        FD_ZERO(&rset);
        FD_ZERO(&wset);
        FD_SET(sockfd, &rset);
        FD_SET(sockfd, &wset);

        struct timeval tval;
        tval.tv_sec = 0;
        tval.tv_usec = 300 * 1000;

        int rdynum;
        rdynum = select(sockfd + 1, &rset, &wset, NULL, &tval);
        if (rdynum < 0) {
                fprintf(stderr, "Select error, %s\n", strerror(errno));
                return -1;
        } else if (rdynum == 0) {
                fprintf(stderr, "Select timeout\n");
                return -1;
        }

        if (FD_ISSET(sockfd, &rset)) {
                fprintf(stderr, "read \n");
        }

        if (FD_ISSET(sockfd, &wset)) {
                fprintf(stderr, "write \n");
        }

        return 0;
}

int epoll_version(int sockfd)
{
        int ep, i, evtnum;

        ep = epoll_create(1024); /* ignore 1024 since Linux 2.6.8 */
        if (ep < 0) {
                fprintf(stderr, "Create epoll error, %s\n", strerror(errno));
                return -1;
        }

        struct epoll_event event;
        event.events = EPOLLIN | EPOLLOUT | EPOLLET;
        event.data.fd = sockfd;
        epoll_ctl(ep, EPOLL_CTL_ADD, sockfd, &event);


        int err = 0;
        int errlen = sizeof(err);

        struct epoll_event events[64];
        evtnum = epoll_wait(ep, events, sizeof(events), 60 * 1000);
```

```
        for (i = 0; i < evtnum; i++) {
                int fd = events[i].data.fd;
                if (events[i].events & EPOLLERR) {
                        fprintf(stderr, "error\n");

                        if (getsockopt(fd, SOL_SOCKET, SO_ERROR, &err, &errlen) == -1) {
                                fprintf(stderr, "getsockopt(SO_ERROR): %s", strerror(errno));
                                close(fd);
                                //return -1;
                        }

                        if (err) {
                                fprintf(stderr, "%s\n", strerror(err));
                                close(fd);
                                //return -1;
                        }
                }

                if (events[i].events & EPOLLIN) {
                        fprintf(stderr, "in\n");
                }

                if (events[i].events & EPOLLOUT) {
                        fprintf(stderr, "out\n");
                }
        }

        return 0;
}


int main(int argc, char** argv)
{
        struct sockaddr_in svraddr;
        int sockfd, rc;


        memset(&svraddr, 0, sizeof(svraddr));
        svraddr.sin_family = AF_INET;
        svraddr.sin_port = htons(8009);
        inet_aton("127.0.0.1", &svraddr.sin_addr.s_addr);

        if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
                perror("Create socket fail");
                return -1;
        }

        if (fcntl(sockfd, F_SETFL, fcntl(sockfd, F_GETFL, 0) | O_NONBLOCK) < 0) {
                perror("Set socket O_NONBLOCK fail");
                return -1;
        }

        rc = connect(sockfd, (struct sockaddr *)&svraddr, sizeof(struct sockaddr_in));
        if (rc < 0 && errno != EINPROGRESS) {
                perror("Connect remote server fail");
                return -1;
        }

        //select_version(&c_fd);
        epoll_version(sockfd);

        close(sockfd);

        return 0;
}
```

← Older (/post/hash-functions-introduce.html)

Newer → (/post/zabbix-monitor-introduce.html)

如果喜欢这里的文章，而且又不差钱的话，欢迎打赏个早餐 ^_^

Like

Issue Page (undefined)

Error: Comments Not Initialized

(https://github.com/login/oauth/authorize?scope=public_repo&redirect_uri=https%3A%2F%2Fjin-yang.github.io%2Fpost%2Fprogram-c-network.html&client_id=6d89d48ce689192bf95d&client_secret=c9a720aafb8e3084e3feb46cadee80b03cdc792f)

Login (https://github.com/login/oauth/authorize?scope=public_repo&redirect_uri=https%3A%2F%2Fjin-yang.github.io%2Fpost%2Fprogram-c-network.html&client_id=6d89d48ce689192bf95d&client_secret=c9a720aafb8e3084e3feb46cadee80b03cdc792f) with GitHub

Write Preview

Leave a comment

Styling with Markdown is supported (https://guides.github.com/features/mastering-markdown/)

Comment

Powered by Gitment (https://github.com/imsun/gitment)