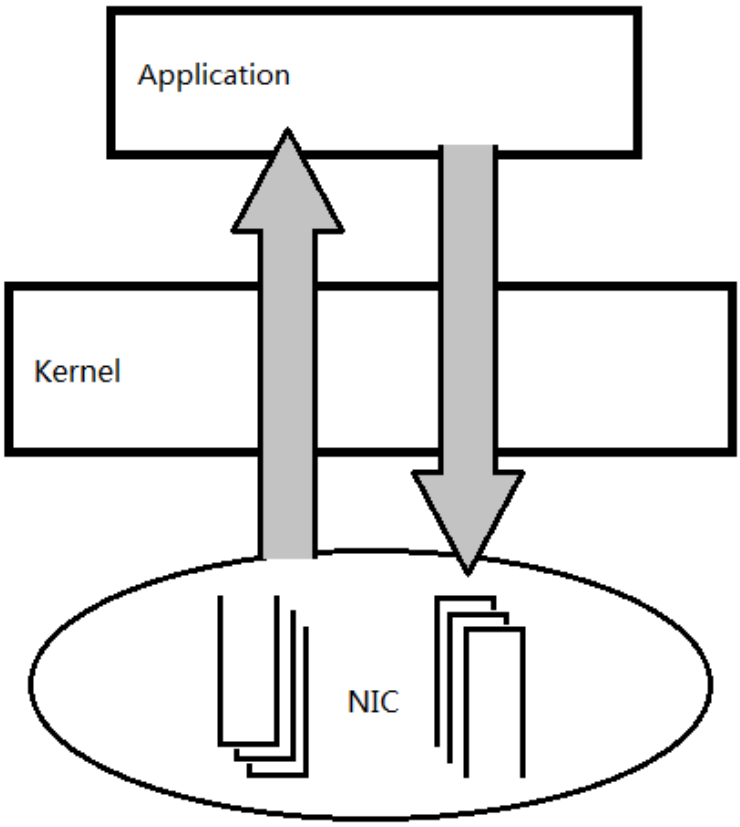
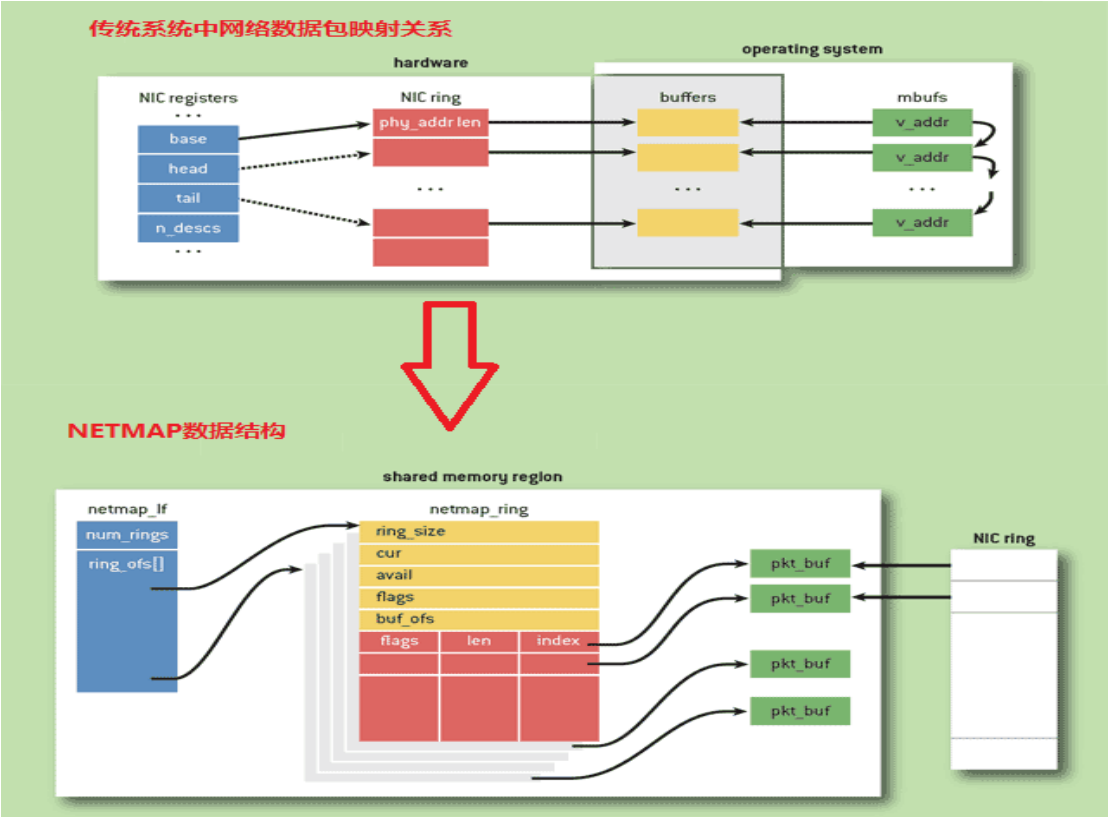
用户态协议栈之 TCP/IP 的设计

1. Netmap 简介

Netmap 是一个高性能收发原始数据包的框架，由 Luigi Rizzo 等人开发完成，其包含了内核模块以及用户态库函数。其目标是，不修改现有操作系统软件以及不需要特殊硬件支持，实现用户态和网卡之间数据包的高性能传递。其原理图如下，数据包不经过操作系统内核进行处理，用户空间程序收发数据包时，直接与网卡进行通信。
代码位置：https://github.com/luigirizzo/netmap



1. 数据结构



在 Netmap 框架下，内核拥有数据包池，发送环接收环上的数据包不需要动态申请，有数据到达网卡时，当有数据到达后，直接从数据包池中取出一个数据包，然后将数据放入此数据包中，再将数据包的描述符放入接收环中。内核中的数据包池，通过 mmap 技术映射到用户空间。用户态程序最终通过 netmap_if 获取接收发送环 netmap_ring，进行数据包的获取发送。

2. 特点总结

（1）性能高：数据包不走传统协议栈，不需要层层解析，用户态直接与网卡的接受环和发送环交互。性能高的具体原因有一下三个：

（a） 系统调用以及处理数据包的时间花费少

（b） 不需要进行数据包的内存分配：采用数据包池，当有数据到达后，直接从数据包池中取出一个数据包，然后将数据放入此数据包中，再将数据包的描述符放入接收环中。

（c） 数据拷贝次数少：内核中的数据包采用 mmap 技术映射到用户态。所以数据包在到达用户态时，不需要进行数据包的拷贝。

（2） 稳定性高：有关网卡寄存器数据的维护都是在内核模块进行，用户不会直接操作寄存器。所以在用户态操作时，不会导致操作系统崩溃

（3） 亲和性：可采用了 CPU 亲和性，实现 CPU 和网卡绑定，提高性能。

（4） 易用性好：API 操作简单，用户态只需要调用 ioctl 函数即可完成数据包收发工作

（5） 与硬件解耦：不依赖硬件，只需要对网卡驱动程序稍微做点修改就可以使用此框架（几十行行），传统网卡驱动将数据包传递给操作系统内核中协议栈，而修改后的数据包直接放入 Netmap_ring 供用户使用。

2. Netmap API 介绍

1. 简要说明

1.netmap API 主要为两个头文件 netmap.h 和 netmap_user.h ，当解压下载好的 netmap 程序后，在./netmap/sys/net/目录下，本文主要对这两个头文件进行分析。

2.我们从 netmap_user.h 头文件开始看起。

2. likely()和 unlikely()

这两个宏定义是对编译器做优化的，并不会对变量做什么改变。后面看到这两个宏的调用自动忽略就好了。

```
#ifndef likely
#define likely(x)       __builtin_expect(!!(x), 1)
#define unlikely(x)     __builtin_expect(!!(x), 0)
#endif /* likely and unlikely */
```

3. netmap.h 头文件

1.netmap.h 被 netmap_user.h 调用，里面定义了一些宏和几个主要的结构体，如 nmreq{}, netmap_if{}, netmap_ring{}, netmap_slot{}。

2.一个网卡(或者网络接口)只有一个 netmap_if{}结构，在使用 mmap()申请的共享内存中，通过 netmap_if{}结构可以访问到任何一个发送/接收环(也就是 netmap_ring{}结构，一个 netmap_if{}可以对应多发送/接收环,这应该和物理硬件有关 ，我在虚拟机下只有一对环，在真实主机上有两队环)。

3.找到 netmap_ring{}的地址后，我们就可以找到环中每一个 buffer 的地址(buffer 里面存储的是将要发送/接收的数据包)。后面会讲解这是如何实现的。

4.通过一个 nifp 是如何访问到多个收/发环的，通过一个 ring 如何找到多个不同的 buffer 地址的，其实都是通过存储这些结构体相邻的后面一部分空间实现。(申请共享内存的时候，这些均已被设计好)

4. 几个重要的宏定义

1._NETMAP_OFFSET

```
#define _NETMAP_OFFSET(type, ptr, offset) \
    ((type)(void *)((char *)(ptr) + (offset)))
```

解释：该宏定义的作用是将 ptr 指针(强转成 char *类型)向右偏移 offset 个字节，再将其转化为指定的类型 type。

2.NETMAP_IF

```
#define NETMAP_IF(_base, _ofs)     _NETMAP_OFFSET(struct netmap_if *, _base, _ofs)
```

解释：该宏定义将_base 指针向右偏移_ofs 个字节后，强转为 netmap_if *类型返回。在 nemap 中通过此宏得到 d->nifp 的地址。

3.NETMAP_TXRING

```
#define NETMAP_TXRING(nifp, index) _NETMAP_OFFSET(struct netmap_ring *, \
    nifp, (nifp)->ring_ofs[index] )
```

解释：1.通过该宏定义，可以找到 nifp 的第 index 个发送环的地址(index 是从 0 开始的)，ring_ofs[index]为偏移量，由内核生成。

2.其中，我们注意到 struct netmap_if{}最后面只定义了 const ssize_t ring_ofs[0]，实际上其它的 netmap 环的偏移量都写在了该结构体后面的内存地址里面，直接访问就可以了。

4.NETMAP_RXRING

```
#define NETMAP_RXRING(nifp, index) _NETMAP_OFFSET(struct netmap_ring *,    \
    nifp, (nifp)->ring_ofs[index + (nifp)->ni_tx_rings + 1] )
```

解释：通过该宏定义，可以找到 nifp 的第 index 个接收环的地址，其中(nifp)->ring_ofs[]里面的下标为 index+(nifp)->ni_tx_rings+1，正好与发送环的偏移量区间隔开 1 个。(我想这应该是作者特意设计的)

5.NETMAP_BUF

```
#define NETMAP_BUF(ring, index)                \
    ((char *)(ring) + (ring)->buf_ofs +
((index)*(ring)->nr_buf_size))
```

解释：1.通过该宏定义，可以找到 ring 这个环的第 index 个 buffer 的地址(buffer 里面存的就是我们接收/将发送的完整数据包)，每个 buffer 占的长度是 2048 字节(在(ring)->nr_buf_size 也给出了)。

2.其中(ring) ->buf_ofs 是固定的偏移量，不同的环这个值不相同，但所有的(char *)(ring)+(ring)->buf_ofs 会指向同一个地址，也就是存放 buffer 的连续内存的开始地址(d->buf_start 会指向该地址)。

6. NETMAP_BUF_IDX

```
#define NETMAP_BUF_IDX(ring, buf)          \
    ( ((char *)(buf) - ((char *)(ring) + (ring)->buf_ofs) ) / \
        (ring)->nr_buf_size )
```

解释：在讲 NETMAP_BUF 的时候我们说(char *)(ring) + (ring)->buf_ofs)总会指向存放 buffer 的起始位置(无论是哪一个环)，在这段内存中将第一个 buffer 下标标记为 0 的话，NETMAP_BUF_IDX 计算的恰好是指针 buf 所指 buffer 的下标。

上面几个宏一时没弄懂也没关系，下面调用的时候还会提的。

5. nm_open 函数

1.调用 nm_open 函数时，如：nmr = nm_open("netmap:eth0", NULL, 0, NULL); nm_open()会对传递的 ifname 指针里面的字符串进行分析，提取出网络接口名。

2.nm_open() 会对 struct nm_desc *d 申请内存空间，并 通过 d->fd = open(NETMAP_DEVICE_NAME, O_RDWR);打开一个特殊的设备/dev/netmap 来创建文件描述符 d->fd。

3.通过 ioctl(d->fd, NIOCREGIF, &d->req)语句，将 d->fd 绑定到一个特殊的接口，并对 d->req 结构体里面的成员做初始化，包括 **a.**在共享内存区域中 nifp 的偏移，**b.**共享区域的大小 nr_memsize，**c.**tx/rx 环的大小 nr_tx_slots/nr_rx_slots(大小为 256)，**d.**tx/rx 环的数量 nr_tx_rings、nr_rx_rings(视硬件性能而定)等。

4.接着在 if ((!(new_flags & NM_OPEN_NO_MMAP) || parent) && nm_mmap(d, parent))语句中调用 nm_mmap 函数，继续给 d 指针指向的内存赋值。

6. nm_mmap 函数

nm_mmap()源码：

```
static int nm_mmap(struct nm_desc *d, const struct nm_desc *parent)
{
//XXX TODO: check if mmap is already done

    if (IS_NETMAP_DESC(parent) && parent->mem && parent->req.nr_arg2
== d->req.nr_arg2)
    {
        /* do not mmap, inherit from parent */
        D("do not mmap, inherit from parent");
        d->memsize = parent->memsize;
        d->mem = parent->mem;
    } else
    {
        /* XXX TODO: 检查如果想申请的内存太大 (or there is overflow)
*/
        d->memsize = d->req.nr_memsize;       /* 将需要申请的内存大小
赋值给 d->memsize */
        d->mem = mmap(0, d->memsize, PROT_WRITE | PROT_READ,
MAP_SHARED, d->fd, 0);  /* 申请共享内存 */
        if (d->mem == MAP_FAILED)
        {
            goto fail;
        }
        d->done_mmap = 1;
    }
    {
```

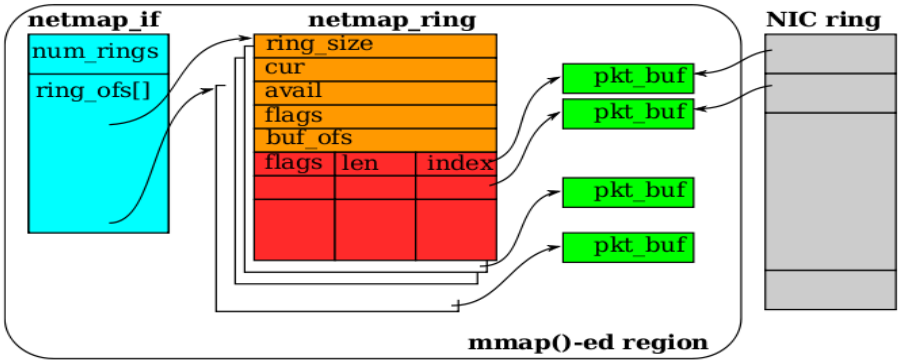
```
        struct netmap_if *nifp = NETMAP_IF(d->mem, d->req.nr_offset);
/*通过 d->req.nr_offset 这个偏移量的到 nifp 的地址，NETMAP_IF 前面说过
*/
        int i;
        /*
         *for(i=0; i<=2; i++)
         *    printf("ring_ofs[%d]:0x%x\n",i,nifp->ring_ofs[i]);   //
这里是我自己加的，为了手动计算收/发环的偏移量
         */
        struct netmap_ring *r = NETMAP_RXRING(nifp,); //对 nifp，找接
收包的环 r，因为 index 为 0，所以省略了

        *(struct netmap_if **) (uintptr_t) &(d->nifp) = nifp;   //对
d->nifp 赋值，虽然 d->nifp 使用 const 定义的，但对其取地址再强值类型转
换后，依然可以对其指向的空间进行操作
        *(struct netmap_ring **) (uintptr_t) &d->some_ring = r; //同
理，对 d->some_ring 进行赋值，此处指向了第一个接受(rx)环。
        //printf("buf_ofs:0x%x\n", (u_int)r->buf_ofs);
        *(void **) (uintptr_t) &d->buf_start = NETMAP_BUF(r, 0);//计
算第一个 buffer 的地址，并存入 d->buf_start 指针中
        *(void **) (uintptr_t) &d->buf_end = (char *) d->mem +
d->memsize; //计算共享区间的最后一个地址，赋值给 d->buf_end
    }

    return 0;

    fail: return EINVAL;
}
```

其中：

1.nifp 为申请的共享内存首地址 d->mem 向右偏移 d->req.nr_offset(该值在调用前面的 ioctl()时得到)得到。并且一个网络接口(网卡)只对应一个 nifp。(使用宏 NETMAP_IF 计算)

2.得到的 nifp 的地址，nifp 结构体里最后定义的 ring_ofs[0]以及接下来内存中的 ring_ofs[1]，ring_ofs[2]...，这些内存中存储的是访问每一个环(tx or rx ring)的偏移量，通过这个偏移量我们可以得到每一个环的地址(使用宏 NETMAP_RXRING/NETMAP_TXRING 进行计算)。

3.得到每个收/发环的地址了，netmap_ring 结构体最后面有一个 struct netmap_slot slot[0];，通过 slot[0],后面内存的 slot[1],slot[2],slot[3]...,取出里面的偏移量就可以得到每一个 buffer(也叫数据包槽)的地址了(使用宏 NETMAP_BUF 计算得到)。 到这里，netmap 如何访问到内存槽中的每一个 buffer 的，我们都知道了。实际上 netmap 运行的数据结构就和下图描述的一样：



4.在 struct nm_desc 中，nifp，some_ring，buf_start，buf_end 等指针都定义为 const 的，但我们通过对其取地址再强转指针的方式去往这些指针指向的内存中赋值。

注:在 nm_mmap()中使用 mmap()申请共享的时候，这些数据结构里数据的设计是内核模块就已写好了的，我们在这里其实是在做验证。

7. nm_nextpkt 函数

1.nm_nextpkt()是用来接收网卡上到来的数据包的函数。

2.nm_nextpkt()会将所有 rx 环都检查一遍，当发现有一个 rx 环有需要接收的数据包时，得到这个数据包的地址，并返回。所以 nm_nextpkt()每次只能取一个数据包。

nm_nextpkt()源代码：

```
static u_char *nm_nextpkt(struct nm_desc *d, struct nm_pkthdr *hdr)
{
    int ri = d->cur_rx_ring;  //当前的接收环的编号
    do
    {
        /* compute current ring to use */
        struct netmap_ring *ring = NETMAP_RXRING(d->nifp, ri);  //得到当前 rx 环的地址
        if (!nm_ring_empty(ring))  //判断环里是否有新到的包
        {
            u_int i = ring->cur; //当前该访问哪个槽(buffer)了
            u_int idx = ring->slot[i].buf_idx; //得到第 i 个 buffer 的下标
            //printf("%d\n", idx);
            u_char *buf = (u_char *) NETMAP_BUF(ring, idx);  //得到存有到来数据包的地址

            // __builtin_prefetch(buf);
            hdr->ts = ring->ts;
            hdr->len = hdr->caplen = ring->slot[i].len;
            ring->cur = nm_ring_next(ring, i);  //ring->cur 向后移动一位
```

```
            /* we could postpone advancing head if we want
             * to hold the buffer. This can be supported in
             * the future.
             */
            ring->head = ring->cur;
            d->cur_rx_ring = ri; //将当前环(d->cur_rx_ring)指向第 ri 个(因为可能有多个环)。
            return buf;  //将数据包地址返回
        }
        ri++;
        if (ri > d->last_rx_ring)  //如果 ri 超过了 rx 环的数量，则再从第一个 rx 环开始检测是否有包到来。
            ri = d->first_rx_ring;
    } while (ri != d->cur_rx_ring);
    return NULL; /* 什么也没发现 */
}
```

8. nm_inject 函数

1.nm_inject()是用来往共享内存中写入待发送的数据包数据的。数据包经共享内存拷贝到网卡，然后发送出去。所以 nm_inject()是用来发包的。

2.nm_inject()也会查找所有的发送环(tx 环)，找到一个可以发送的槽，就将数据包写入并返回，所以每次函数调用也只能发送一个包。

源代码：

```
static int nm_inject(struct nm_desc *d, const void *buf, size_t size)
{
    u_int c, n = d->last_tx_ring - d->first_tx_ring + 1;

    for (c = 0; c < n; c++)
    {
        /*   计算当前的环去使用(compute current ring to use) */
        struct netmap_ring *ring;
        uint32_t i, idx;
        uint32_t ri = d->cur_tx_ring + c;  //该访问第几个 tx 环了

        if (ri > d->last_tx_ring)  //当超过访问的 tx 环的下标范围时，从头开始访问
            ri = d->first_tx_ring;
        ring = NETMAP_TXRING(d->nifp, ri);  //得到当前 tx 环的地址
        if (nm_ring_empty(ring))        //如果当前 tx 环是满的(ring->cur=ring->tail 表示没地方存数据包了)，就跳过
```

```
        {
            continue;
        }
        i = ring->cur; //当前要往哪个槽(槽指向 buffer)中写入数据
        idx = ring->slot[i].buf_idx; //得到这个槽相对于 buffer 起始地
址(d->buf_start)的下标编号
        ring->slot[i].len = size;  //size 为待发送数据包的长度
        nm_pkt_copy(buf, NETMAP_BUF(ring, idx), size); //将 buf 里存的
数据包拷贝给 ring 这个环的第 i 个槽
        d->cur_tx_ring = ri;
        ring->head = ring->cur = nm_ring_next(ring, i); //将 head 和
cur 指向下一个槽
        return size;
    }
    return 0; /* 失败 */
}
```

9. nm_close 函数

1.nm_close 函数就是回收动态内存，回收共享内存，关闭文件描述符什么的了。

源代码：

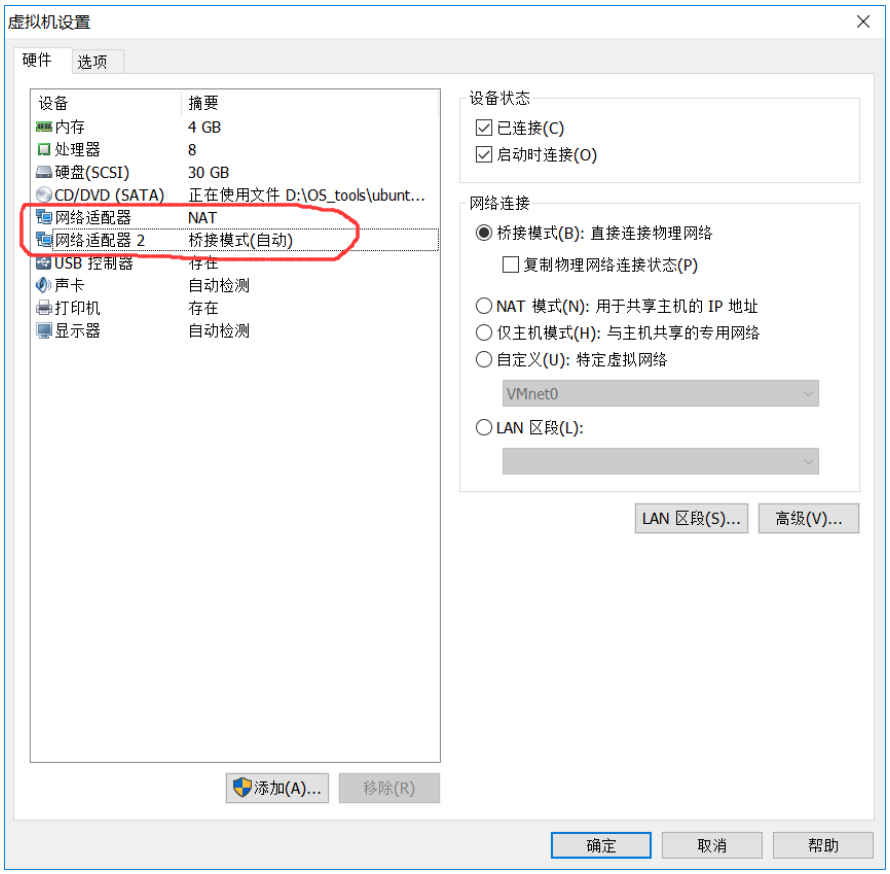
```
static int nm_close(struct nm_desc *d)
{
    /*
     * ugly trick to avoid unused warnings
     */
    static void *__xxzt[] __attribute__ ((unused)) =
    { (void *) nm_open, (void *) nm_inject, (void *) nm_dispatch,
(void *) nm_nextpkt };

    if (d == NULL || d->self != d)
        return EINVAL;
    if (d->done_mmap && d->mem)
        munmap(d->mem, d->memsize);  //释放申请的共享内存
    if (d->fd != -1)
    {
        close(d->fd); //关闭文件描述符
    }

    bzero(d, sizeof(*d));  //将 d 指向的空间全部置 0
    free(d);    //释放指针 d 指向的空间
    return 0;
```

```
}
```

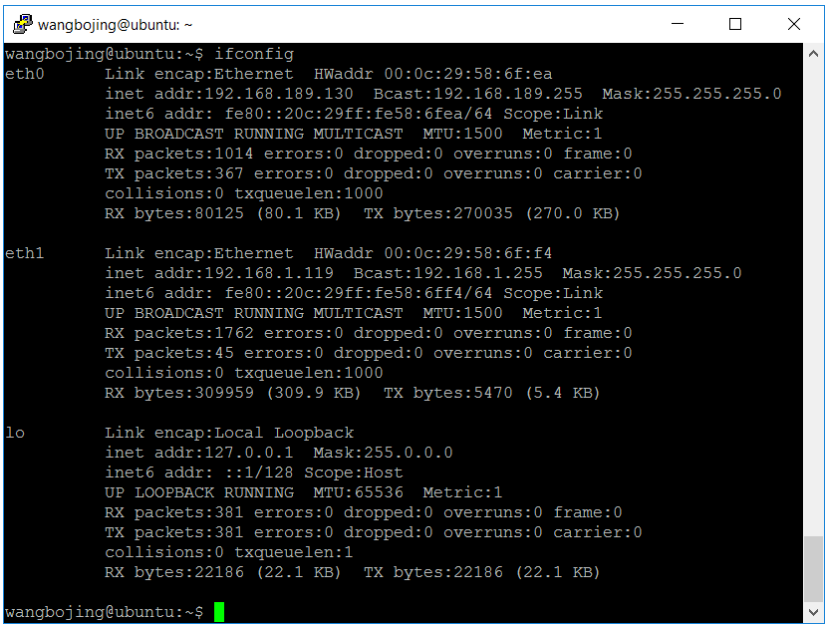
3. NtyTCP 安装

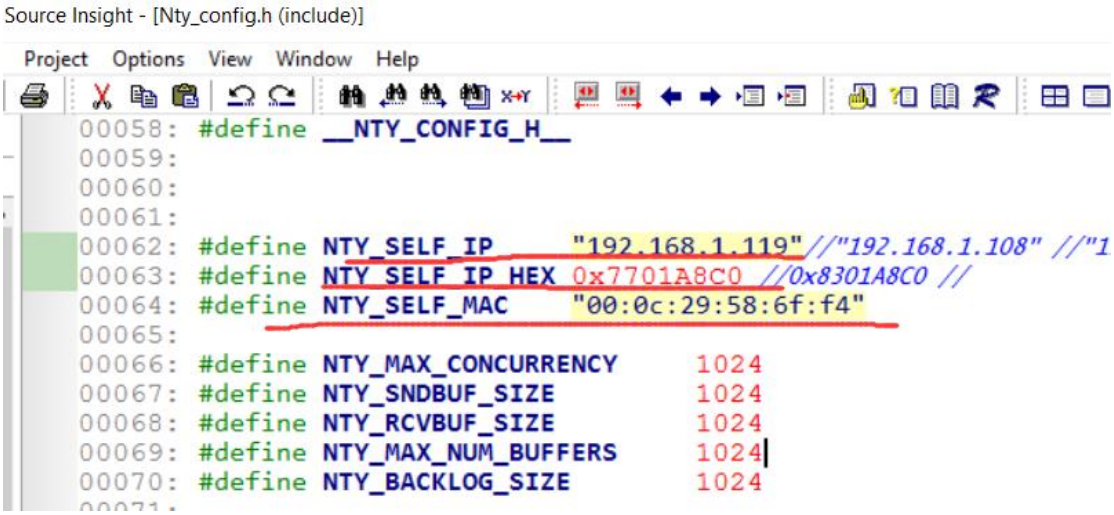
VMWare 编译与调试

1. 添加两个网络适配器



系统启动后，





添加绑定的网卡的 IP 地址，十六进制 IP 地址，网卡相应的 MAC 地址。

代码地址

https://github.com/wangbojing/NtyTcp.git

环境编译，下面以 ubuntu server 版本为例。

先安装 netmap

Ubuntu 14.04

https://github.com/wangbojing/netmap.git

Ubuntu 16.04

https://github.com/luigirizzo/netmap.git

```
# ./configure
# make
# make install
```

进入 ntytcp 的目录

直接 make

compile:

$ sudo apt-get install libhugetlbfs-dev

$ make

update NtyTcp/include/nty_config.h

#define NTY_SELF_IP "192.168.0.106" //your ip

#define NTY_SELF_IP_HEX 0x6A00A8C0 //your ip hex.

#define NTY_SELF_MAC "00:0c:29:58:6f:f4" //your mac

block server run:

$ ./bin/nty_example_block_server

epoll server run:

$ ./bin/nty_example_epoll_rb_server

if you discover bug to sending email to 1989wangbojing@163.com.

also, want to be an NtyTcper, so you can sent email to 1989wangbojing@163.com .

4. C10M 的问题

截至目前，40gpbs、32-cores、256G RAM 的 X86 服务器在 Newegg 网站上的报价是几千美元。实际上以这样的硬件配置来看，它完全可以处理 1000 万个以上的并发连接，如果它们不能，那是因为你选择了错误的软件，而不是底层硬件的问题。

可以预见在接下来的 10 年里，因为 IPv6 协议下每个服务器的潜在连接数都是数以百万级的，单机服务器处理数百万的并发连接（甚至千万）并非不可能，但我们需要重新审视目前主流 OS 针对网络编程这一块的具体技术实现。

1、解决 C10M 问题并非不可能

很多人会想当然的认为，要实现 C10M（即单机千万）并发连接和处理能力，是不可能的。不过事实并非如此，现在系统已经在用你可能不熟悉甚至激进的方式支持千万级别的并发连接。

要知道它是如何做到的，我们首先要了解 Errata Security 的 CEO Robert Graham，以及他在 Shmoocon 2013 大会上的“天方夜谈”视频记录： C10M Defending The Internet At Scale（此为 Yutube 视频，你懂的）。

Robert 用一种我以前从未听说的方式来很巧妙地解释了这个问题。他首先介绍了一点有关 Unix 的历史，Unix 的设计初衷并不是一般的服务器操作系统，而是电话网络的控制系统。由于是实际传送数据的电话网络，所以在控制层和数据层之间有明确的界限。问题是我们现在根本不应该使用 Unix 服务器作为数据层的一部分。正如设计只运行一个应用程序的服务器内核，肯定和设计多用户的服务器内核是不同的。

Robert Graham 的结论是：OS 的内核不是解决 C10M 问题的办法，恰恰相反 OS 的内核正是导致 C10M 问题的关键所在。

这也就意味着：
不要让 OS 内核执行所有繁重的任务：将数据包处理、内存管理、处理器调度等任务从内核转移到应用程序高效地完成，让诸如 Linux 这样的 OS 只处理控制层，数据层完全交给应用程序来处理。
最终就是要设计这样一个系统，该系统可以处理千万级别的并发连接，它在 200 个时钟周期内处理数据包，在 14 万个时钟周期内处理应用程序逻辑。由于一次主存储器访问就要花费 300 个时钟周期，所以这是最大限度的减少代码和缓存丢失的关键。
面向数据层的系统可以每秒处理 1 千万个数据包，面向控制层的系统，每秒只能处理 1 百万个数据包。这似乎很极端，请记住一句老话：可扩展性是专业化的，为了做好一些事情，你不能把性能问题外包给操作系统来解决，你必须自己做。

2、回顾一下 C10K 问题

10 年前，开发人员处理 C10K 可扩展性问题时，尽量避免服务器处理超过 1 万个的并发连接。通过改进操作系统内核以及用事件驱动服务器（典型技术实现如：Nginx 和 Node）代替线程服务器（典型代表：Apache），使得这个问题已经被解决。人们用十年的时间从 Apache 转移到可扩展服务器，在近几年，可扩展服务器的采用率增长得更快了。

以传统网络编程模型作为代表的 Apache 为例，我们来看看它在 C10K 问题上的局限表现在哪些方面，并针对性的讨论对应的解决方法。Apache 的问题在于服务器的性能会随着连接数的增多而变差，实际上性能和可扩展性并不是一回事。当人们谈论规模时，他们往往是在谈论性能，但是规模和性能是不同的，比如 Apache。持续几秒的短期连接：比如快速事务，如果每秒处理 1000 个事务，只能有约 1000 个并发连接到服务器。如果事务延长到 10 秒，要维持每秒 1000 个事务则必须打开 1 万个并发连接。这种情况下：尽管你不顾 DoS 攻击，Apache 也会性能陡降，同时大量的下载操作也会使 Apache 崩溃。

如果每秒处理的连接从 5 千增加到 1 万，你会怎么做？比方说，你升级硬件并且提高处理器速度到原来的 2 倍。到底发生了什么？你得到两倍的性能，但你没有得到两倍的处理规模。每秒处理的连接可能只达到了 6000。你继续提高速度，情况也没有改善。甚至 16 倍的性能时，仍然不能处理 1 万个并发连接。所以说性能和可扩展性是不一样的。

问题在于 Apache 会创建一个 CGI 进程，然后关闭，这个步骤并没有扩展。为什么呢？内核使用的 O(N^2)算法使服务器无法处理 1 万个并发连接。

OS 内核中的两个基本问题：

- 连接数=线程数/进程数：当一个数据包进来，内核会遍历其所有进程以决定由哪个进程来处理这个数据包。
- 连接数=选择数/轮询次数（单线程）：同样的可扩展性问题，每个包都要走一遭列表上所有的 socket。

通过上述针对 Apache 所表现出的问题，实际上彻底解决并发性能问题的解决方法的根本就是改进 OS 内核使其在常数时间内查找，使线程切换时间与线程数量无关，使用一个新的可扩展 epoll()/IOCompletionPort 常数时间去做 socket 查询。

因为线程调度并没有得到扩展，所以服务器大规模对 socket 使用 epoll 方法，这样就导致需要使用异步编程模式，而这些编程模式正是 Nginx 和 Node 类型服务器具有的。所以当从 Apache 迁移到 Nginx 和 Node 类型服务器时，即使在一个配置较低的服务器上增加连接数，性能也不会突降。所以在处理 C10K 连接时，一台笔记本电脑的速度甚至超过了 16 核的服务器。这也是前一个 10 年解决 C10K 问题的普遍方法。

3、实现 C10M 意味着什么？

实现 10M（即 1 千万）的并发连接挑战意味着什么：

- **1 千万的并发连接数**；
- **100 万个连接/秒**：每个连接以这个速率持续约 10 秒；
- **10GB/秒的连接**：快速连接到互联网；
- **1 千万个数据包/秒**：据估计目前的服务器每秒处理 50K 数据包，以后会更多；
- **10 微秒的延迟**：可扩展服务器也许可以处理这个规模（但延迟可能会飙升）；
- **10 微秒的抖动**：限制最大延迟；
- **并发 10 核技术**：软件应支持更多核的服务器（通常情况下，软件能轻松扩展到四核，服务器可以扩展到更多核，因此需要重写软件，以支持更多核的服务器）。

4、为什么说实现 C10M 的挑战不在硬件而在软件？

1. 理由概述

硬件不是 10M 问题的性能瓶颈所在处，真正的问题出在软件上，尤其是*nux 操作系统。理由如下面这几点：

首先：最初的设计是让 Unix 成为一个电话网络的控制系统，而不是成为一个服务器操作系统。对于控制系统而言，针对的主要目标是用户和任务，而并没有针对作为协助功能的数据处理做特别设计，也就是既没有所谓的快速路径、慢速路径，也没有各种数据服务处理的优先级差别。

其次：传统的 CPU，因为只有一个核，操作系统代码以多线程或多任务的形式来提升整体性能。而现在，4 核、8 核、32 核、64 核和 100 核，都已经是真实存在的 CPU 芯片，如何提高多核的性能可扩展性，是一个必须面对的问题。比如让同一任务分割在多个核心上执行，以避免 CPU 的空闲浪费，当然，这里面要解决的技术点有任务分割、任务同步和异步等。

再次：核心缓存大小与内存速度是一个关键问题。现在，内存已经变得非常的便宜，随便一台普通的笔记本电脑，内存至少也就是 4G 以上，高端服务器的内存上 24G 那是相当的平常。但是，内存的访问速度仍然很慢，CPU 访问一次内存需要约 60~100 纳秒，相比很久以前的内存访问速度，这基本没有增长多少。对于在一个带有 1GHZ 主频 CPU 的电脑硬件

里，如果要实现 10M 性能，那么平均每一个包只有 100 纳秒，如果存在两次 CPU 访问内存，那么 10M 性能就达不到了。核心缓存，也就是 CPU L1/L2/LL Cache，虽然访问速度会快些，但大小仍然不够，我之前接触到的高端至强，LLC 容量大小貌似也就是 12M。

2. 解决思路

解决这些问题的关键在于如何将功能逻辑做好恰当的划分，比如专门负责控制逻辑的控制面和专门负责数据逻辑的数据面。数据面专门负责数据的处理，属于资源消耗的主要因素，压力巨大，而相比如此，控制面只负责一些偶尔才有非业务逻辑，比如与外部用户的交互、信息的统计等等。我之前接触过几种网络数据处理框架，比如 Intel 的 DPDK、6wind、windriver，它们都针对 Linux 系统做了特别的补充设计，增加了数据面、快速路径等等特性，其性能的提升自然是相当巨大。

看一下这些高性能框架的共同特点：

- **数据包直接传递到业务逻辑：**
  而不是经过 Linux 内核协议栈。这是很明显的事情，因为我们知道，Linux 协议栈是复杂和繁琐的，数据包经过它无非会导致性能的巨大下降，并且会占用大量的内存资源，之前有同事测试过，Linux 内核要吃掉 2.5KB 内存/socket。我研究过很长一段时间的 DPDK 源码，其提供的 82576 和 82599 网卡驱动就直接运行在应用层，将接管网卡收到的数据包直接传递到应用层的业务逻辑里进行处理，而无需经过 Linux 内核协议栈。当然，发往本服务器的非业务逻辑数据包还是要经过 Linux 内核协议栈的，比如用户的 SSH 远程登录操作连接等。
- **多线程的核间绑定：**
  一个具有 8 核心的设备，一般会有 1 个控制面线程和 7 个或 8 个数据面线程，每一个线程绑定到一个处理核心（其中可能会存在一个控制面线程和一个数据面线程都绑定到同一个处理核心的情况）。这样做的好处是最大化核心 CACHE 利用、实现无锁设计、避免进程切换消耗等等。
- **内存是另外一个核心要素：**
  常见的内存池设计必须在这里得以切实应用。有几个考虑点，首先，可以在 Linux 系统启动时把业务所需内存直接预留出来，脱离 Linux 内核的管理。其次，Linux 一般采用 4K 每页，而我们可以采用更大内存分页，比如 2M，这样能在一定程度上减少地址转换等的性能消耗。

3. 关于 Intel 的 DPDK 框架/ Netmap 开源框架

随着网络技术的不断创新和市场的发展，越来越多的网络设备基础架构开始向基于通用处理器平台的架构方向融合，期望用更低的成本和更短的产品开发周期来提供多样的网络单元和丰富的功能，如应用处理、控制处理、包处理、信号处理等。为了适应这一新的产业趋势，Intel 推出了基于 Intel x86 架构 DPDK (Data Plane Development Kit，数据平面开发套件) 实现了高效灵活的包处理解决方案。经过近 6 年的发展，DPDK 已经发展成支持多种高性能网卡和多通用处理器平台的开源软件工具包。

5、解决 C10M 问题的思路总结

综上所述，解决 C10M 问题的关键主要是从下面几个方面入手：

网卡问题

网卡问题：通过内核工作效率不高
解决方案：使用自己的驱动程序并管理它们，使适配器远离操作系统。

CPU 问题

CPU 问题：使用传统的内核方法来协调你的应用程序是行不通的。
解决方案：Linux 管理前两个 CPU，你的应用程序管理其余的 CPU，中断只发生在你允许的 CPU 上。

内存问题

内存问题：内存需要特别关注，以求高效。
解决方案：在系统启动时就分配大部分内存给你管理的大内存页。
以 Linux 为例，解决的思咯就是将控制层交给 Linux，应用程序管理数据。应用程序与内核之间没有交互、没有线程调度、没有系统调用、没有中断，什么都没有。 然而，你有的是在 Linux 上运行的代码，你可以正常调试，这不是某种怪异的硬件系统，需要特定的工程师。你需要定制的硬件在数据层提升性能，但是必须是在你熟悉的编程和开发环境上进行。