理解了实现再谈网络性能

开发内功修炼之网络篇

张彦飞

最后更新时间：2022-02-01
添加网络包发送过程分析

公众号：「开发内功修炼」

了解你的每一比特、用好你的每一纳秒！

公众号

作者微信

Github：https://github.com/yanfeizhang/coder-kung-fu

技术群：加作者微信 zhangyanfei748527，我来拉大家！

引 言

作者简介

大家好，我是张彦飞，大家都喜欢称呼我飞哥。我于 2010 年 7 月从西北大学计算机学院硕士毕业，之后一直混迹于帝都。刚毕业时因为户口的问题去了一家广播电视软件龙头公司，但按捺不住对互联网的向往，在 2011 年就跳槽去了腾讯。后来随着腾讯和搜狗的战略联姻，被老板连人带业务一并注入到搜狗。现在在搜狗任专家开发工程师，负责 PC 浏览器、搜狗手机助手等产品的后端技术。

我在这十多年的工作中有不少的技术思考，因此创办了「开发内功修炼」技术号，分别在微信公众号和知乎两个平台发布。这本电子书是我之前所发布的网络相关文章的一个总结，在网络文章的基础上理顺了整本电子书的架构，也添加了一些网络开发优化建议、实际问题分析等章节。

马上 35 岁，我该不该焦虑

之前网络在爆炒一篇标题为《互联网不需要中年人》，疯狂渲染 35 岁的码农的前程问题，制造焦虑。本来我觉得这个事情应该只是媒体博眼球的一个炒作而已。不过恰恰最近面试了有 70 多人，其中有很多工作 7, 8 年以上的的同学。

这些人里基本上可以非常明确地划分成两类。第一类是虽然工作了 7, 8 年以上了，但是所有的经验都集中在业务层。换句话说，并不是有 7 - 8 年经验，而是工作了 7 - 8 年而已。稍微深入问一点性能相关的问题都没有好的思路，技术能力并没有随着工作年限的增长而增长。也确实为这种类型的同学的前途感到担忧。另外一类同学都是除了业务的开发以外，还会额外抽时间注重自己的技术积累，成长的就比较好，对于他们我觉得别说 35 岁了，45 岁也仍然会是团队的中流砥柱。

对于第一类同学来讲，技术成长过慢也不完全是他们自己的原因。现在国内的开发环境中都存在着一个普遍的矛盾，那就是排期过满的需求开发和需要大把时间去磨练技术的矛盾。 排期过满导致开发同学平时绝大部分的时间都是在处理各种各样的业务逻辑和 bug 。很多同学的时间全部被这种需求吃掉了，导致没有想法或精力去提升自己的底层技术能力，导致技术 level 并没有随着工作年限而同比增长。

udp_rcv 平时大家都是用各种语言进行业务逻辑的代码编写，无论你用的是 PHP、Go、还是 Java，都是属于应用层的范畴。但是应用层是建立在物理层和内核层之上的。如果遇到一些性能问题的时候，很可能需要你对底层有足够深的理解，才能够有效优化性能或排除线上故障。内核和物理层的知识往往又比较难吃透，不投入足够多的时间是难有大的突破。

我把在应用层的技术能力称之为外功，把 Linux 内核、设备物理结构称之为内功。我想大家的外功能力已经都足够优秀了，但是一部分同学的内功能力却严重不足。

我的技术号的名称之所以命名为开发者内功修炼，就是想把我在内核、物理层上的一些理解思考总结分享给大家，帮助这些同学提升自己的技术木桶短板。

在本书中，我会从应用层的问题出发，深入到 Linux 中进行分析。帮助大家提升对底层的理解，加强你的技术深度，也为你的中年焦虑症送上一剂良药。

目前业界技术文章的问题

相信很多同学也都保持着在互联网上进行学习的习惯，我但我觉得目前大多数的技术类文章还是太偏向于各种花拳绣腿了。

高性能的 PHP 异步网络通信引擎 swoole 的作者韩天峰说过，很多程序员职业规划的文章，上来就是 Linux、PHP、MySQL、Nginx、Redis、Memcache、jQuery 这些，然后就直接上手搭环境、做项目，中级就是学习各种 PHP 框架和类库，高级阶段就是 MySQL 优化、PHP 内核与扩展、架构设计这些了。这些文章都存在一个严重的缺陷，不重视基础。就好比练武功，只求速成，不修炼内功和心法，只练各种招式，这样练出来的高手能高到哪里去?

事实上，码农圈里的牛人比如韩天峰，比如 PHP7 作者鸟哥，他们都具备非常非常扎实的基础，他们之所以牛，并不是他们掌握的花拳绣腿的功夫多，而是内功非常非常深厚。举个小例子，鸟哥在 PHP7 中把 HashTable 结构体从 72 字节压缩到了 56 字节，表面看起来不大的优化，实际上是成倍的性能提升。因为 CPU 在向内存要数据的时候是以 Cache Line 为单位进行的，一个 Cache Line 是 64 字节。56 字节可以一次请求搞定，而原来的 72 字节则需要两次。另外就是 L1/L2/L3 的命中率也会提升很多，这个对性能的帮助更大。

内功它不帮助你掌握最新的开发语言，也不教会你时髦的框架，也不会带你走进火热的人工智能。但是我相信它是你成为大牛的必经之路。我简单列一下锻炼内功的好处：

1）内功方面的技术生命周期长。Linux 操作系统 1991 年就发布了，现在还是发展的如日中天。对于作者 Linus，我觉得他也有年龄焦虑。但他可能焦虑的找不到接班人。反观应用层的一些技术尤其是很多的框架，生命周期能超过十年我就已经觉得他很牛叉了。如果你的精力全部押宝在这些生命周期很短的技术上，你说能不焦虑吗！所以我觉得戒掉浮躁，踏踏实实练好内功打好是你对抗中年焦虑的解药之一。

2）内功深厚的人理解新技术非常快。拿我自己来举两个小例子吧。我其实没怎么去翻过 kafka 的源码。但是当我研究完了内核是如何读取文件的、内核处理网络包的整体过程后，就秒懂了 kafka 在网络这块为啥性能表现很突出了。另外一个是当我理解了 epoll 的内部实现以后，回头再看 Golang 的 net 包，才切切实实看懂了地球上绝顶精妙的对网络 IO 的封装。所以 Linux 内核你真的弄懂了的话，再看应用层的各种新技术就犹如带了透视镜一般，直接看到骨骼。

3）内核提供了优秀系统设计的实例。Linux 作为一个千锤百炼的系统，其中蕴含了大量的世界顶级的设计和实现方案。平时我们在自己的业务开发中，在编码之前也需要先进行设计。比如我在刚工作的时候负责的一个数据采集任务调度，其中的实现就是部分参考了操作系统进程调度方案。再比如如何在管理海量的连接的情况下仍然能高效发现某一条连接上的 IO 事件，epoll 内部的红黑树 + 队列的组合可以提供给你一个很好的参考。这种例子还有很多很多，总之如果能将 Linux 的某些优秀实现，搬到你的系统中会极大提升你的项目的实现水平。

时髦的东西终究会过时，但扎实的内功能力将会伴随你的一生。只有具备了深厚的内功底蕴，你才能在你的发展道路上走的更稳、走的更远。

飞哥深入研究网络的出发点

其实我在刚工作前几年对计算机网络的理解也是不深的。

有人说，学习网络就是在学习各种协议，这种说法其实误导了很多的人。提到计算机网络的知识点，你肯定也首先想到的是 OSI 七层模型、IP、TCP、UDP，HTTP 等等。关于 TCP 再多一点你也会想到三次握手、四次挥手、滑动窗口、流量控制。关于 HTTP 协议就是报文格式、GET/POST，状态码、Cookie/Session 等等。

但是我的这些知识却不能帮我清除我在工作中的如下几个疑惑。

1）有一次我们运维找过来，说某某几台线上机器上出现了 3 万多个 TIME_WAIT，不行了得赶紧处理哈。后来他帮我们打开了 tcp_tw_reuse 和 tcp_tw_recycle，先把问题处理掉了。但是我的思考却并没有停止，**一条 TIME_WAIT 到底会有哪些开销**。端口占用导致新连接无法建立？还是会过多消耗机器上的内存？ 3 万条 TIME_WAIT 究竟该算是 warning 还是 error?

2）另外一次是当时公司还没有建立 redis 平台之前，我们业务自己维护了一组 redis server。为了节约握手开销，我们的业务进程对 redis 开启了长连接。这样一个 redis 实例上最终就出现了 6000 条的连接。虽然每条连接上大部分时间都是空闲的，但是我却在思考**一条空闲的连接究竟会有哪些开销**？这 6000 条连接会不会把服务器搞坏？ 当然最终观测的结果是没啥问题，但是对于细节原理我仍然吃的不是很准。

3）另外一次也是我们业务要把短连接优化成长连接。但是这次涉及到了要访问公司的 Mysql 平台。当时我们公司的 Mysql 需要为每一个 ip 申请一个并发数。因为我们当时使用的是 php-fpm，没有连接池的概念。所以我们有多少个 fpm 进程，就得申请多大的并发数，我们当时申请了 200 个。 然后工程部的同学就过来 PK 了，你们这单机 200 个并发不行，太高了。虽然我最终给他举了上面 redis 的例子，最终他同意了。虽然勉强说服了他，但是我仍然吃不准空闲的 tcp 连接到底开销在哪儿？

经过以上各种实践问题的磨炼，我发现了问题的关键所在。那就是**我们大家之前对于计算机网络的知识太多都聚集于协议层面了，太过于偏向理论。而对于网络在机器是如何被实现的，开销如何，却理解太少太少了。 网络是如何使用 cpu的，如何使用内存的，这些知识感觉在整个业界说很匮乏也不为过。**

举个小例子，按理说现在整个业界都在讲高并发，那一台服务器到底能支持多少条 TCP 连接，这算是高并发里很基础的问题了吧。但是当我产生以上疑问以后，在网上、在各种经典书里，都根本搜不到能让我满意的答案。很多文章都是含含糊糊讲了半天 ，讨论了半天的 CxxxK。但看完还是不知道最多能支持多少，更别说具体点的细节了，比如一条 TCP 到底需要消耗多大内存。

所以，飞哥下定决心，要在实现层面把计算机网络扒个底儿朝天，把网络的 cpu 开销，内存开销要彻底搞搞清楚！

开发内功修炼网络篇就这样诞生了！

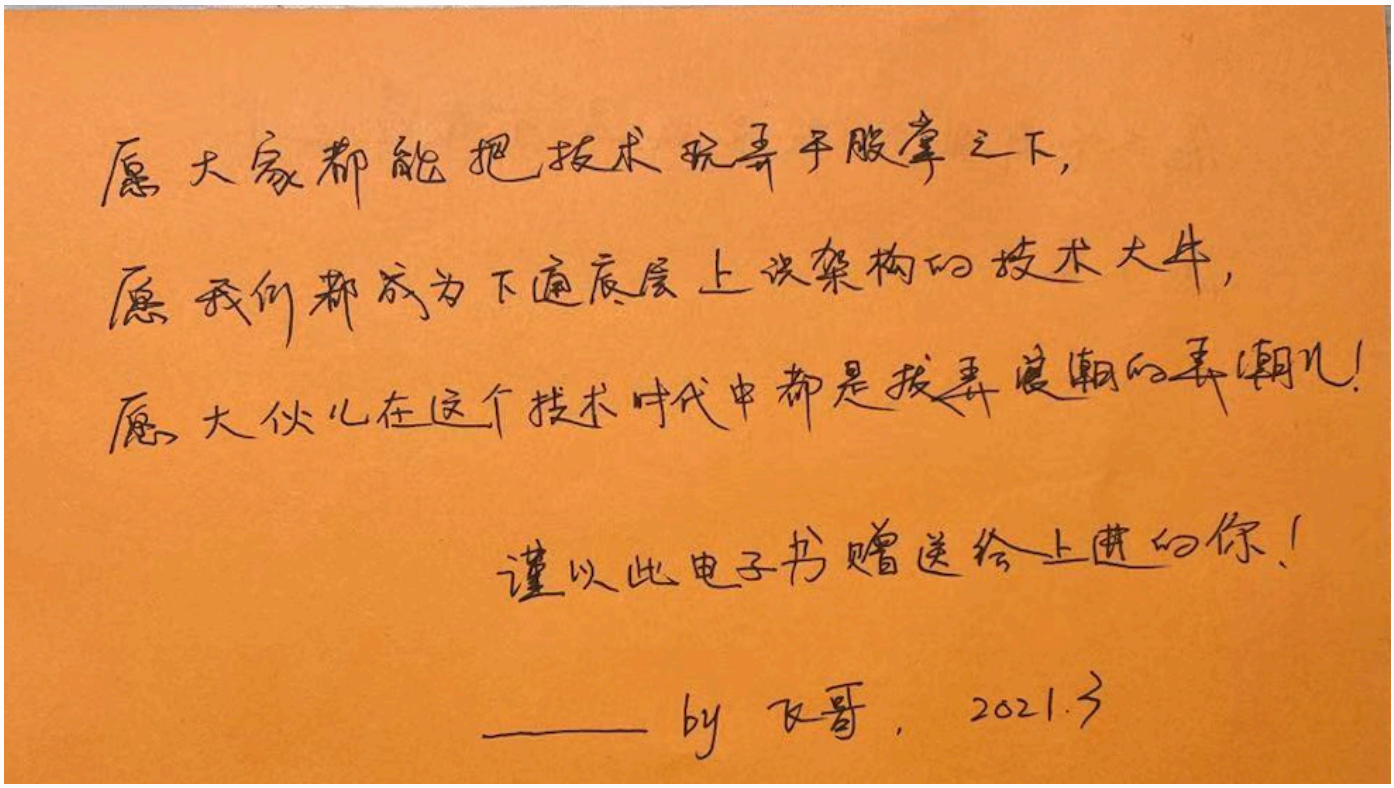
适用读者

本书并不是一本计算机网络的入门书，需要你具备起码的计算机网络知识。它适用于以下这些同学

1）有几年开发工作经验，但对网络开销把握不准的开发同学
2）想做网络性能优化，但却只会压测，没有成体系的理论指导的同学
3）维护各种高并发服务器的运维同学
4）不满足于只学习网络协议，也想理解它是咋实现的同学

其它说明

本电子书内容可能比较滞后，想关注飞哥的最新思考还请关注公众号「开发内功修炼」。另外由于本人才疏学浅，难免会有疏漏。如您发现内容中有描述不正确的地方，欢迎添加飞哥本人微信 (zhangyanfei748527) 指正！



公众号：「开发内功修炼」

了解你的每一比特、 用好你的每一纳秒！

公众号

作者微信

一、内核是如何接收网络包的

今天开始我们讨论 Linux 里最重要的一个模块-网络模块。让我们从一段最简单的代码开始思考。为了简单起见，我们用简单的 udp 来举例，如下：

```
int main(){
    int serverSocketFd = socket(AF_INET, SOCK_DGRAM, 0);
    bind(serverSocketFd, ...);
    recvfrom(serverSocketFd, ...);
    ......
}
```

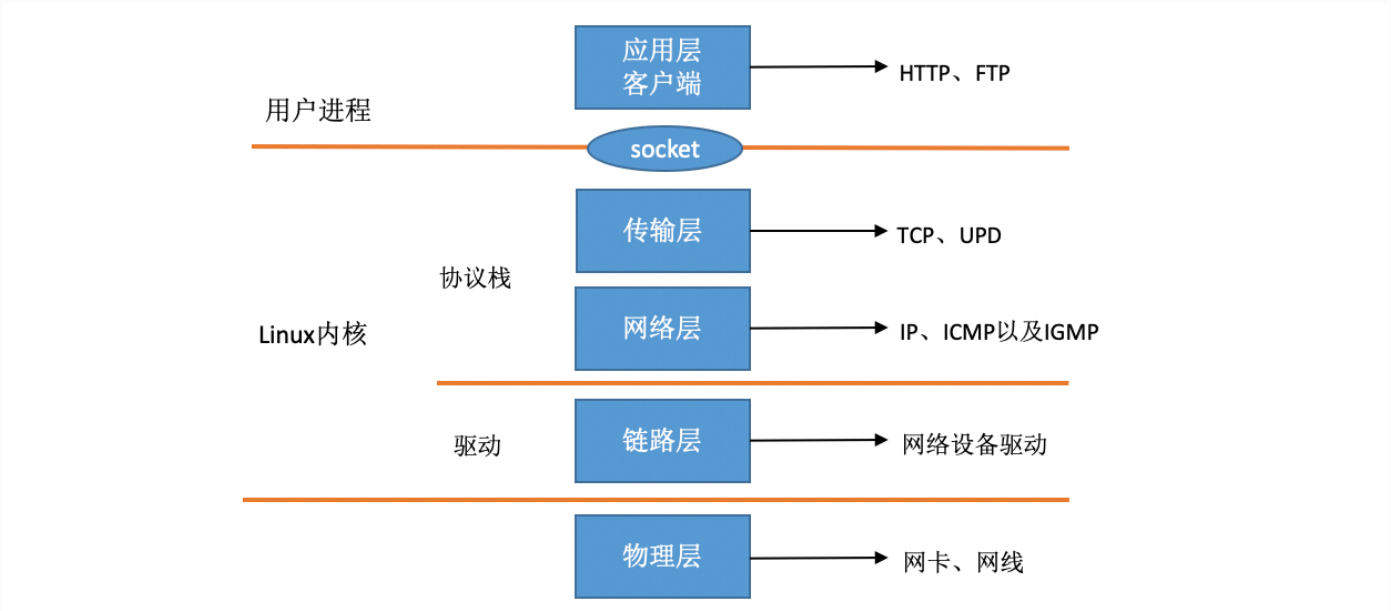
上面代码是非常简单的一段 udp server 接收收据的逻辑。 当在开发视角看的时候，只要客户端有对应的数据发送过来，服务器端执行 `recv_from` 后就能收到它，并把它打印出来。我们现在想知道的是，Linux 是如何接收一个网络包的呢?

我们为什么要了解这么底层呢？如果你负责的应用不是高并发的，流量也不大，确实没有必要往下看。如果你负责的是为百万，千万甚至过亿用户提供的服务，深入理解 Linux 系统内部是如何实现的，以及各个部分之间是如何交互对你的工作将会有非常大的帮助。本文基于 Linux 3.10，源代码参见 https://mirrors.edge.kernel.org/pub/linux/kernel/v3.x/，网卡驱动采用Intel的igb网卡举例。

1.1 Linux网络收包总览

在 TCP/IP 网络分层模型里，整个协议栈被分成了物理层、链路层、网络层，传输层和应用层。物理层对应的是网卡和网线，应用层对应的是我们常见的 Nginx，FTP 等等各种应用。Linux 实现的是链路层、网络层和传输层这三层。

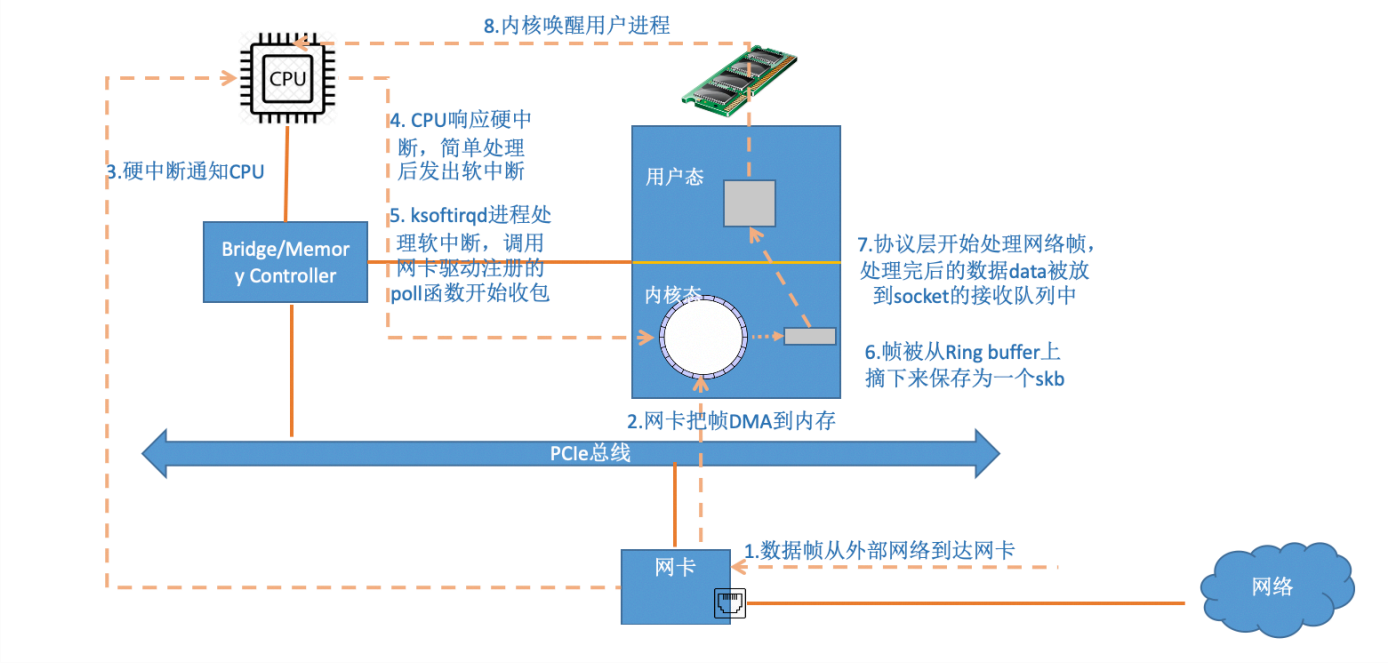
在 Linux 内核实现中，链路层协议靠网卡驱动来实现，内核协议栈来实现网络层和传输层。内核对更上层的应用层提供 socket 接口来供用户进程访问。我们用 Linux 的视角来看到的 TCP/IP 网络分层模型应该是下面这个样子的。



在 Linux 的源代码中，网络设备驱动对应的逻辑位于 `driver/net/ethernet` ，其中 intel 系列网卡的驱动在 `driver/net/ethernet/intel` 目录下。协议栈模块代码位于 `kernel` 和 `net` 目录。

内核和网络设备驱动是通过中断的方式来处理的。当设备上有数据到达的时候，会给 CPU 的相关引脚上触发一个电压变化，以通知 CPU 来处理数据。对于网络模块来说，由于处理过程比较复杂和耗时，如果在中断函数中完成所有的处理，将会导致中断处理函数（优先级过高）将过度占据 CPU ，将导致 CPU 无法响应其它设备，例如鼠标和键盘的消息。因此Linux中断处理函数是分上半部和下半部的。上半部是只进行最简单的工作，快速处理然后释放 CPU ，接着 CPU 就可以允许其它中断进来。剩下将绝大部分的工作都放到下半部中，可以慢慢从容处理。2.4 以后的内核版本采用的下半部实现方式是软中断，由 ksoftirqd 内核线程全权处理。和硬中断不同的是，硬中断是通过给 CPU 物理引脚施加电压变化，而软中断是通过给内存中的一个变量的二进制值以通知软中断处理程序。

好了，大概了解了网卡驱动、硬中断、软中断和ksoftirqd线程之后，我们在这几个概念的基础上给出一个内核收包的路径示意:



当网卡上收到数据以后，Linux 中第一个工作的模块是网络驱动。 网络驱动会以 DMA 的方式把网卡上收到的帧写到内存里。再向 CPU 发起一个中断，以通知 CPU 有数据到达。第二，当 CPU 收到中断请求后，会去调用网络驱动注册的中断处理函数。 网卡的中断处理函数并不做过多工作，发出软中断请求，然后尽快释放 CPU。ksoftirqd 检测到有软中断请求到达，调用 poll 开始轮询收包，收到后交由各级协议栈处理。对于 udp 包来说，会被放到用户

socket 的接收队列中。

我们从上面这张图中已经从整体上把握到了 Linux 对数据包的处理过程。但是要想了解更多网络模块工作的细节，我们还得往下看。

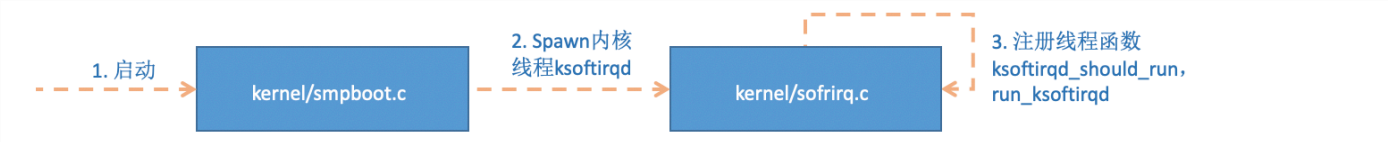
1.2 Linux 启动

Linux 驱动，内核协议栈等等模块在具备接收网卡数据包之前，要做很多的准备工作才行。比如要提前创建好ksoftirqd内核线程，要注册好各个协议对应的处理函数，网卡设备子系统要提前初始化好，网卡要启动好。只有这些都Ready之后，我们才能真正开始接收数据包。那么我们现在来看看这些准备工作都是怎么做的。

创建ksoftirqd内核进程

Linux 的软中断都是在专门的内核线程（ksoftirqd）中进行的，因此我们非常有必要看一下这些进程是怎么初始化的，这样我们才能在后面更准确地了解收包过程。该进程数量不是 1 个，而是 N 个，其中 N 等于你的机器的核数。

系统初始化的时候在 kernel/smpboot.c中调用了 smpboot_register_percpu_thread， 该函数进一步会执行到 spawn_ksoftirqd（位于kernel/softirq.c）来创建出 softirqd 进程。



相关代码如下：

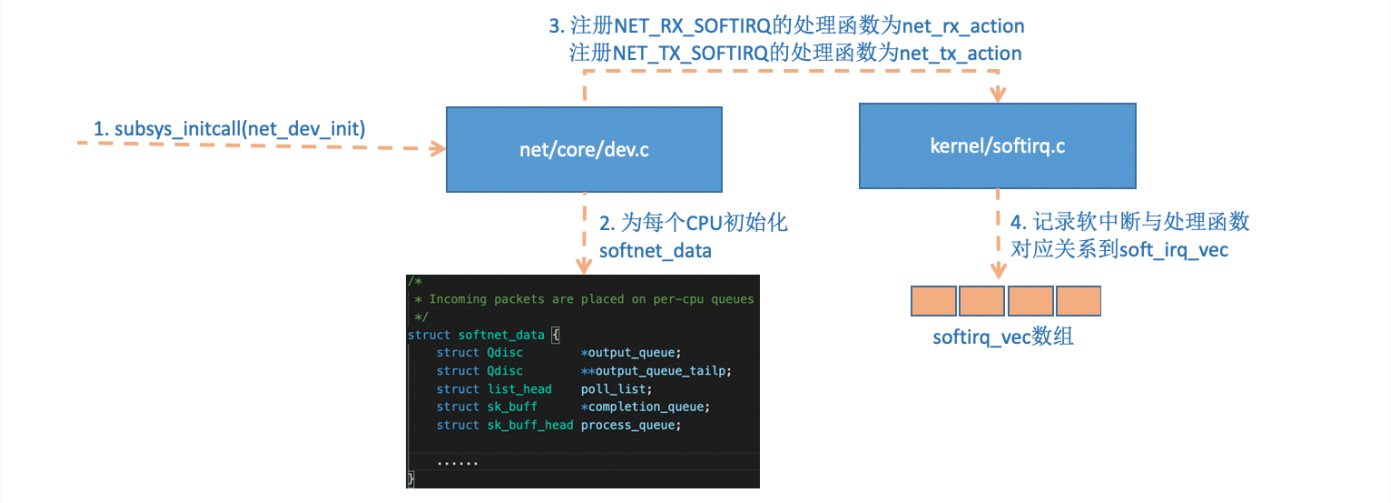
```
//file: kernel/softirq.c
static struct smp_hotplug_thread softirq_threads = {
    .store              = &ksoftirqd,
    .thread_should_run  = ksoftirqd_should_run,
    .thread_fn          = run_ksoftirqd,
    .thread_comm        = "ksoftirqd/%u",
};
```

```
static __init int spawn_ksoftirqd(void)
{
    register_cpu_notifier(&cpu_nfb);

    BUG_ON(smpboot_register_percpu_thread(&softirq_threads));

    return 0;
}
early_initcall(spawn_ksoftirqd);
```

当 ksoftirqd 被创建出来以后，它就会进入自己的线程循环函数 ksoftirqd_should_run和 run_ksoftirqd 了。不停地判断有没有软中断需要被处理。这里需要注意的一点是，软中断不仅仅只有网络软中断，还有其它类型。

```
//file: include/linux/interrupt.h
enum
{
    HI_SOFTIRQ=0,
    TIMER_SOFTIRQ,
    NET_TX_SOFTIRQ,
    NET_RX_SOFTIRQ,
    BLOCK_SOFTIRQ,
    BLOCK_IOPOLL_SOFTIRQ,
    TASKLET_SOFTIRQ,
    SCHED_SOFTIRQ,
    HRTIMER_SOFTIRQ,
    RCU_SOFTIRQ,    /* Preferable RCU should always be the last
softirq */

    NR_SOFTIRQS
};
```

网络子系统初始化



linux 内核通过调用 `subsys_initcall` 来初始化各个子系统，在源代码目录里你可以 grep 出许多对这个函数的调用。这里我们要说的是网络子系统的初始化，会执行到 `net_dev_init` 函数。

```
//file: net/core/dev.c
static int __init net_dev_init(void)
{
    ......

    for_each_possible_cpu(i) {
        struct softnet_data *sd = &per_cpu(softnet_data, i);

        memset(sd, 0, sizeof(*sd));
        skb_queue_head_init(&sd->input_pkt_queue);
        skb_queue_head_init(&sd->process_queue);
        sd->completion_queue = NULL;
        INIT_LIST_HEAD(&sd->poll_list);

        ......
    }

    ......

    open_softirq(NET_TX_SOFTIRQ, net_tx_action);
    open_softirq(NET_RX_SOFTIRQ, net_rx_action);
}
subsys_initcall(net_dev_init);
```

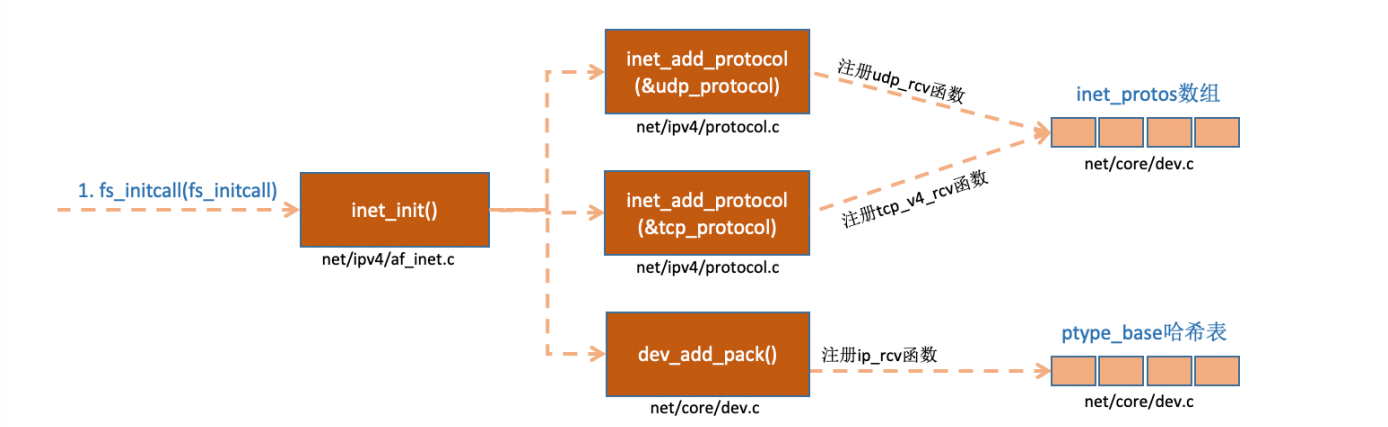
在这个函数里，会为每个 CPU 都申请一个 `softnet_data` 数据结构，在这个数据结构里的 `poll_list` 是等待驱动程序将其 poll 函数注册进来，稍后网卡驱动初始化的时候我们可以看到这一过程。

另外 open_softirq 注册了每一种软中断都注册一个处理函数。 NET_TX_SOFTIRQ 的处理函数为 net_tx_action，NET_RX_SOFTIRQ 的为 net_rx_action。继续跟踪 `open_softirq` 后发现这个注册的方式是记录在 `softirq_vec` 变量里的。后面 ksoftirqd 线程收到软中断的时候，也会使用这个变量来找到每一种软中断对应的处理函数。

```
//file: kernel/softirq.c
void open_softirq(int nr, void (*action)(struct softirq_action
*))
{
    softirq_vec[nr].action = action;
}
```

协议栈注册

内核实现了网络层的 ip 协议，也实现了传输层的 tcp 协议和 udp 协议。 这些协议对应的实现函数分别是 ip_rcv(), tcp_v4_rcv()和 udp_rcv()。和我们平时写代码的方式不一样的是，内核是通过注册的方式来实现的。 Linux 内核中的 `fs_initcall` 和 `subsys_initcall` 类似，也是初始化模块的入口。 `fs_initcall` 调用 `inet_init` 后开始网络协议栈注册。 通过 `inet_init`，将这些函数注册到了 inet_protos 和 ptype_base 数据结构中了。如下图:



相关代码如下

```
//file: net/ipv4/af_inet.c
static struct packet_type ip_packet_type __read_mostly = {
    .type = cpu_to_be16(ETH_P_IP),
    .func = ip_rcv,
};

static const struct net_protocol udp_protocol = {
    .handler =  udp_rcv,
    .err_handler =  udp_err,
    .no_policy =    1,
    .netns_ok = 1,
};

static const struct net_protocol tcp_protocol = {
    .early_demux    =   tcp_v4_early_demux,
    .handler    =   tcp_v4_rcv,
    .err_handler    =   tcp_v4_err,
    .no_policy  =   1,
    .netns_ok   =   1,
};

static int __init inet_init(void)
{
    ......

    if (inet_add_protocol(&icmp_protocol, IPPROTO_ICMP) < 0)
        pr_crit("%s: Cannot add ICMP protocol\n", __func__);
    if (inet_add_protocol(&udp_protocol, IPPROTO_UDP) < 0)
        pr_crit("%s: Cannot add UDP protocol\n", __func__);
    if (inet_add_protocol(&tcp_protocol, IPPROTO_TCP) < 0)
        pr_crit("%s: Cannot add TCP protocol\n", __func__);

    ......

    dev_add_pack(&ip_packet_type);
}
```

上面的代码中我们可以看到，udp_protocol 结构体中的 handler 是 udp_rcv，tcp_protocol 结构体中的 handler 是tcp_v4_rcv，通过 inet_add_protocol 被初始化了进来。

```
int inet_add_protocol(const struct net_protocol *prot, unsigned
char protocol)
{
    if (!prot->netns_ok) {
        pr_err("Protocol %u is not namespace aware, cannot
register.\n",
            protocol);
        return -EINVAL;
    }

    return !cmpxchg((const struct net_protocol
**)&inet_protos[protocol],
            NULL, prot) ? 0 : -1;
}
```

`inet_add_protocol` 函数将 tcp 和 udp 对应的处理函数都注册到了 inet_protos 数组中了。再看 `dev_add_pack(&ip_packet_type);` 这一行，ip_packet_type 结构体中的 type 是协议名，func 是 ip_rcv 函数，在dev_add_pack 中会被注册到 ptype_base 哈希表中。

```
//file: net/core/dev.c
void dev_add_pack(struct packet_type *pt)
{
    struct list_head *head = ptype_head(pt);
    ......
}

static inline struct list_head *ptype_head(const struct
packet_type *pt)
{
    if (pt->type == htons(ETH_P_ALL))
        return &ptype_all;
    else
        return &ptype_base[ntohs(pt->type) & PTYPE_HASH_MASK];
}
```

这里我们需要记住 **inet_protos 记录着 udp，tcp 的处理函数地址，ptype_base 存储着 ip_rcv() 函数的处理地址**。后面我们会看到软中断中会通过 ptype_base 找到 ip_rcv 函数地址，进而将 ip 包正确地送到 ip_rcv() 中执行。在 ip_rcv 中将会通过 inet_protos 找到 tcp 或者 udp 的处理函数，再而把包转发给 udp_rcv() 或 tcp_v4_rcv() 函数。

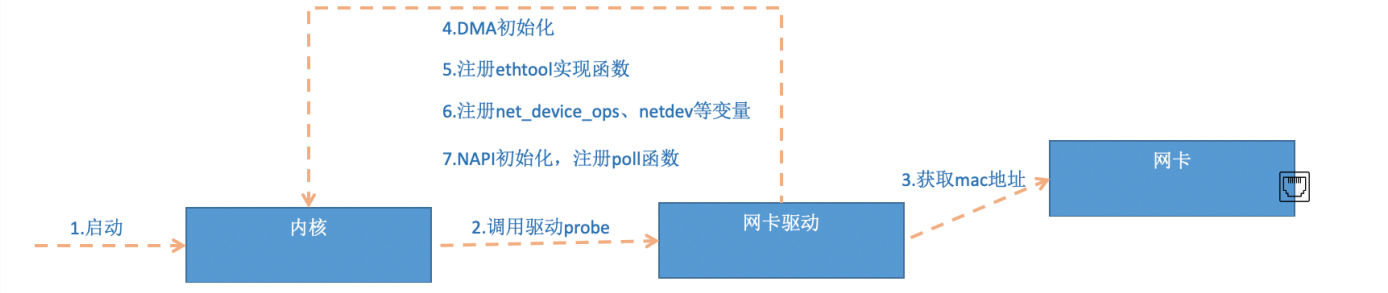
扩展一下，如果看一下 ip_rcv 和 udp_rcv 等函数的代码能看到很多协议的处理过程。例如，ip_rcv 中会处理 netfilter 和 iptable 过滤，如果你有很多或者很复杂的 netfilter 或 iptables 规则，这些规则都是在软中断的上下文中执行的，会加大网络延迟。再例如，udp_rcv 中会判断 socket 接收队列是否满了。对应的相关内核参数是 net.core.rmem_max 和net.core.rmem_default。如果有兴趣，建议大家好好读一下 `inet_init` 这个函数的代码。

网卡驱动初始化

每一个驱动程序（不仅仅只是网卡驱动）会使用 module_init 向内核注册一个初始化函数，当驱动被加载时，内核会调用这个函数。比如igb网卡驱动的代码位于 `drivers/net/ethernet/intel/igb/igb_main.c`

```
//file: drivers/net/ethernet/intel/igb/igb_main.c
static struct pci_driver igb_driver = {
    .name     = igb_driver_name,
    .id_table = igb_pci_tbl,
    .probe    = igb_probe,
    .remove   = igb_remove,
    ......
};

static int __init igb_init_module(void)
{
    ......
    ret = pci_register_driver(&igb_driver);
    return ret;
}
```

驱动的 `pci_register_driver` 调用完成后，Linux 内核就知道了该驱动的相关信息，比如 igb 网卡驱动的 `igb_driver_name` 和 `igb_probe` 函数地址等等。当网卡设备被识别以后，内核会调用其驱动的 probe 方法（igb_driver 的 probe 方法是 igb_probe）。驱动 probe 方法执行的目的就是让设备 ready，对于 igb 网卡，其 `igb_probe` 位于 drivers/net/ethernet/intel/igb/igb_main.c 下。主要执行的操作如下：



第 5 步中我们看到，网卡驱动实现了 ethtool 所需要的接口，也在这里注册完成函数地址的注册。当 ethtool 发起一个系统调用之后，内核会找到对应操作的回调函数。对于 igb 网卡来说，其实现函数都在 drivers/net/ethernet/intel/igb/igb_ethtool.c 下。 相信你这次能彻底理解 ethtool 的工作原理了吧？ 这个命令之所以能查看网卡收发包统计、能修改网卡自适应模式、能调整 RX 队列的数量和大小，是因为 ethtool 命令最终调用到了网卡驱动的相应方法，而不是 ethtool 本身有这个超能力。

第6步注册的 igb_netdev_ops 中包含的是 igb_open 等函数，该函数在网卡被启动的时候会被调用。

```
//file: drivers/net/ethernet/intel/igb/igb_main.c
......
static const struct net_device_ops igb_netdev_ops = {
  .ndo_open               = igb_open,
  .ndo_stop               = igb_close,
  .ndo_start_xmit         = igb_xmit_frame,
  .ndo_get_stats64        = igb_get_stats64,
  .ndo_set_rx_mode        = igb_set_rx_mode,
  .ndo_set_mac_address    = igb_set_mac,
  .ndo_change_mtu         = igb_change_mtu,
  .ndo_do_ioctl           = igb_ioctl,......
```

第 7 步中，在 igb_probe 初始化过程中，还调用到了 `igb_alloc_q_vector` 。他注册了一个 NAPI 机制所必须的 poll 函数，对于 igb 网卡驱动来说，这个函数就是 igb_poll ,如下代码所示。

```
static int igb_alloc_q_vector(struct igb_adapter *adapter,
                  int v_count, int v_idx,
                  int txr_count, int txr_idx,
                  int rxr_count, int rxr_idx)
{

    ......
    /* initialize NAPI */
    netif_napi_add(adapter->netdev, &q_vector->napi,
                igb_poll, 64);

}
```

启动网卡

当上面的初始化都完成以后，就可以启动网卡了。回忆前面网卡驱动初始化时，我们提到了驱动向内核注册了 structure net_device_ops 变量，它包含着网卡启用、发包、设置mac地址等回调函数（函数指针）。当启用一个网卡时（例如，通过 ifconfig eth0 up），net_device_ops 中的 igb_open 方法会被调用。它通常会做以下事情：


```
//file: drivers/net/ethernet/intel/igb/igb_main.c
static int __igb_open(struct net_device *netdev, bool resuming)
{
    /* allocate transmit descriptors */
    err = igb_setup_all_tx_resources(adapter);
```

```
    /* allocate receive descriptors */
    err = igb_setup_all_rx_resources(adapter);

    /* 注册中断处理函数 */
    err = igb_request_irq(adapter);
    if (err)
        goto err_req_irq;

    /* 启用NAPI */
    for (i = 0; i < adapter->num_q_vectors; i++)
        napi_enable(&(adapter->q_vector[i]->napi));

    ......
}
```

在上面 `__igb_open` 函数调用了 igb_setup_all_tx_resources ,和 igb_setup_all_rx_resources。在 `igb_setup_all_rx_resources` 这一步操作中，分配了 RingBuffer，并建立内存和Rx队列的映射关系。（Rx Tx 队列的数量和大小可以通过 ethtool 进行配置）。我们再接着看中断函数注册 `igb_request_irq`：

```
static int igb_request_irq(struct igb_adapter *adapter)
{
    if (adapter->msix_entries) {
        err = igb_request_msix(adapter);
        if (!err)
            goto request_done;
        ......
    }
}

static int igb_request_msix(struct igb_adapter *adapter)
{
    ......
    for (i = 0; i < adapter->num_q_vectors; i++) {
        ...
        err = request_irq(adapter->msix_entries[vector].vector,
                  igb_msix_ring, 0, q_vector->name,
    }
```

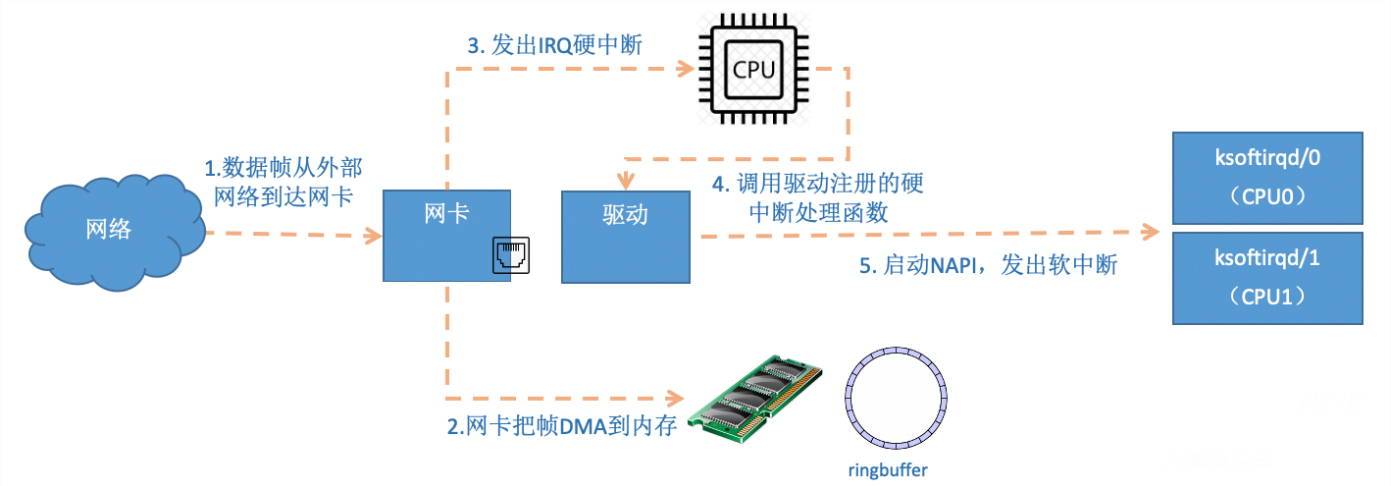
在上面的代码中跟踪函数调用， `__igb_open` => `igb_request_irq` => `igb_request_msix`，在 `igb_request_msix` 中我们看到了，对于多队列的网卡，为每一个队列都注册了中断，其对应的中断处理函数是 igb_msix_ring（该函数也在 drivers/net/ethernet/intel/igb/igb_main.c 下）。 我们也可以看到，msix 方式下，每个 RX 队列有独立的 MSI-X 中断，从网卡硬件中断的层面就可以设置让收到的包被不同的 CPU 处理。（可以通过 irqbalance ，或者修改 /proc/irq/IRQ_NUMBER/smp_affinity 能够修改和CPU的绑定行为）。

当做好以上准备工作以后，就可以开门迎客（数据包）了！

1.3 迎接数据的到来

硬中断处理

首先当数据帧从网线到达网卡上的时候，第一站是网卡的接收队列。网卡在分配给自己的 RingBuffer 中寻找可用的内存位置，找到后 DMA 引擎会把数据 DMA 到网卡之前关联的内存里，这个时候 CPU 都是无感的。当 DMA 操作完成以后，网卡会向 CPU 发起一个硬中断，通知 CPU 有数据到达。



> 注意：当RingBuffer满的时候，新来的数据包将给丢弃。ifconfig查看网卡的时候，可以里面有个overruns，表示因为环形队列满被丢弃的包。如果发现有丢包，可能需要通过ethtool命令来加大环形队列的长度。

在启动网卡一节，我们说到了网卡的硬中断注册的处理函数是igb_msix_ring。

```
//file: drivers/net/ethernet/intel/igb/igb_main.c
static irqreturn_t igb_msix_ring(int irq, void *data)
{
    struct igb_q_vector *q_vector = data;

    /* Write the ITR value calculated from the previous
interrupt. */
    igb_write_itr(q_vector);

    napi_schedule(&q_vector->napi);

    return IRQ_HANDLED;
}
```

`igb_write_itr` 只是记录一下硬件中断频率（据说目的是在减少对 CPU 的中断频率时用到）。顺着 napi_schedule 调用一路跟踪下去，`__napi_schedule` => `____napi_schedule`

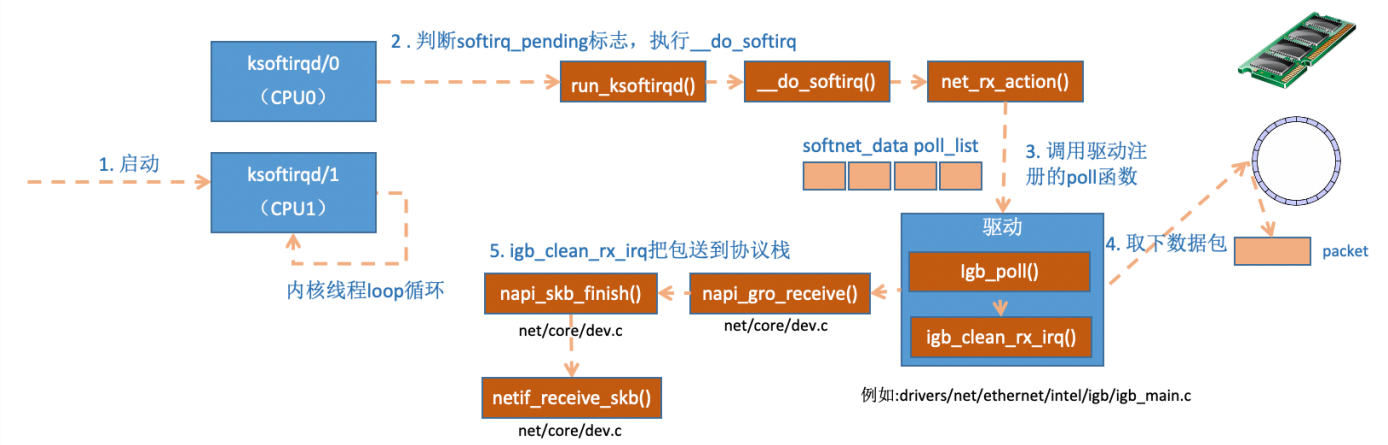
```
/* Called with irq disabled */
static inline void ____napi_schedule(struct softnet_data *sd,
                      struct napi_struct *napi)
{
    list_add_tail(&napi->poll_list, &sd->poll_list);
    __raise_softirq_irqoff(NET_RX_SOFTIRQ);
}
```

这里我们看到，`list_add_tail` 修改了 CPU 变量 softnet_data 里的 poll_list，将驱动 napi_struct 传过来的 poll_list 添加了进来。
其中 softnet_data 中的 poll_list 是一个双向列表，其中的设备都带有输入帧等着被处理。紧接着 `__raise_softirq_irqoff` 触发了一个软中断 NET_RX_SOFTIRQ，这个所谓的触发过程只是对一个变量进行了一次或运算而已。

```
void __raise_softirq_irqoff(unsigned int nr)
{
    trace_softirq_raise(nr);
    or_softirq_pending(1UL << nr);
}
```

我们说过，Linux 在硬中断里只完成简单必要的工作，剩下的大部分的处理都是转交给软中断的。通过上面代码可以看到，硬中断处理过程真的是非常短。只是记录了一个寄存器，修改了一下下 CPU 的 poll_list，然后发出个软中断。就这么简单，硬中断工作就算是完成了。

ksoftirqd 内核线程处理软中断



内核线程初始化的时候，我们介绍了 ksoftirqd 中两个线程函数 `ksoftirqd_should_run` 和 `run_ksoftirqd` 。其中 `ksoftirqd_should_run` 代码如下：

```
static int ksoftirqd_should_run(unsigned int cpu)
{
    return local_softirq_pending();
}

#define local_softirq_pending() \
    __IRQ_STAT(smp_processor_id(), __softirq_pending)
```

这里看到和硬中断中调用了同一个函数 `local_softirq_pending` 。使用方式不同的是硬中断位置是为了写入标记，这里仅仅只是读取。如果硬中断中设置了 `NET_RX_SOFTIRQ` ,这里自然能读取的到。接下来会真正进入线程函数中 `run_ksoftirqd` 处理：

```
static void run_ksoftirqd(unsigned int cpu)
{
    local_irq_disable();
    if (local_softirq_pending()) {
        __do_softirq();
        rcu_note_context_switch(cpu);
        local_irq_enable();
        cond_resched();
        return;
    }
    local_irq_enable();
}
```

在 `__do_softirq` 中，判断根据当前 CPU 的软中断类型，调用其注册的 action 方法。

```
asmlinkage void __do_softirq(void)
{
    do {
        if (pending & 1) {
            unsigned int vec_nr = h - softirq_vec;
            int prev_count = preempt_count();

            ...
            trace_softirq_entry(vec_nr);
            h->action(h);
            trace_softirq_exit(vec_nr);
            ...
        }
        h++;
        pending >>= 1;
    } while (pending);
}
```

这里需要注意一个细节，硬中断中设置软中断标记，和 ksoftirq 的判断是否有软中断到达，都是基于 smp_processor_id() 的。这意味着只要硬中断在哪个 CPU 上被响应，那么软中断也是在这个 CPU 上处理的。所以说，如果你发现你的 Linux 软中断 CPU 消耗都集中在一个核上的话，做法是要把调整硬中断的 CPU 亲和性，来将硬中断打散到不同的 CPU 核上去。

我们再来把精力集中到这个核心函数 `net_rx_action` 上来。

```
//file:net/core/dev.c
static void net_rx_action(struct softirq_action *h)
{
    struct softnet_data *sd = &__get_cpu_var(softnet_data);
    unsigned long time_limit = jiffies + 2;
    int budget = netdev_budget;
    void *have;

    local_irq_disable();

    while (!list_empty(&sd->poll_list)) {
        ......
```

```
        n = list_first_entry(&sd->poll_list, struct
napi_struct, poll_list);

        work = 0;
        if (test_bit(NAPI_STATE_SCHED, &n->state)) {
            work = n->poll(n, weight);
            trace_napi_poll(n);
        }

        budget -= work;net_rx_action
    }
}
```

> 有同学问在硬中断中添加设备到 poll_list，会不会重复添加呢？答案是不会的，在软中断处理函数 net_rx_action 这里一进来就调用 local_irq_disable 把所有的硬中断都给关了，不会让硬中断重复添加 poll_list 的机会。在硬中断的处理函数中本身也有类似的判断机制，打磨了几十年的内核考虑在细节考虑上还是很完善的。

函数开头的 time_limit 和 budget 是用来控制 net_rx_action 函数主动退出的，目的是保证网络包的接收不霸占 CPU 不放。 等下次网卡再有硬中断过来的时候再处理剩下的接收数据包。其中 budget 可以通过内核参数调整。 这个函数中剩下的核心逻辑是获取到当前 CPU 变量 softnet_data，对其 poll_list 进行遍历，然后执行到网卡驱动注册到的 poll 函数。对于 igb 网卡来说，就是 igb 驱动里的 `igb_poll` 函数了。

```
/**
 *  igb_poll - NAPI Rx polling callback
 *  @napi: napi polling structure
 *  @budget: count of how many packets we should handle
 **/
static int igb_poll(struct napi_struct *napi, int budget)
{
    ...
    if (q_vector->tx.ring)
        clean_complete = igb_clean_tx_irq(q_vector);

    if (q_vector->rx.ring)
        clean_complete &= igb_clean_rx_irq(q_vector, budget);
    ...
}
```

在读取操作中，`igb_poll` 的重点工作是对 `igb_clean_rx_irq` 的调用。

```
static bool igb_clean_rx_irq(struct igb_q_vector *q_vector,
const int budget)
{
    ...

    do {

        /* retrieve a buffer from the ring */
        skb = igb_fetch_rx_buffer(rx_ring, rx_desc, skb);

        /* fetch next buffer in frame if non-eop */
        if (igb_is_non_eop(rx_ring, rx_desc))
            continue;
        }

        /* verify the packet layout is correct */
        if (igb_cleanup_headers(rx_ring, rx_desc, skb)) {
            skb = NULL;
            continue;
        }

        /* populate checksum, timestamp, VLAN, and protocol */
        igb_process_skb_fields(rx_ring, rx_desc, skb);

        napi_gro_receive(&q_vector->napi, skb);
}
```

`igb_fetch_rx_buffer` 和 `igb_is_non_eop` 的作用就是把数据帧从 RingBuffer 上取下来。为什么需要两个函数呢？因为有可能帧要占多个 RingBuffer，所以是在一个循环中获取的，直到帧尾部。获取下来的一个数据帧用一个 sk_buff 来表示。收取完数据以后，对其进行一些校验，然后开始设置 sbk 变量的 timestamp, VLAN id, protocol 等字段。接下来进入到 napi_gro_receive 中:

```
//file: net/core/dev.c
gro_result_t napi_gro_receive(struct napi_struct *napi, struct
sk_buff *skb)
{
    skb_gro_reset_offset(skb);

    return napi_skb_finish(dev_gro_receive(napi, skb), skb);
}
```

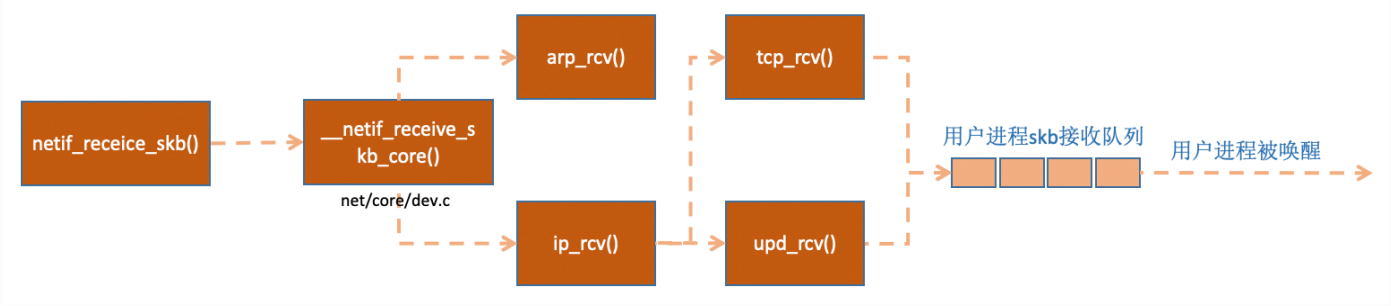
`dev_gro_receive` 这个函数代表的是网卡 GRO 特性，可以简单理解成把相关的小包合并成一个大包就行，目的是减少传送给网络栈的包数，这有助于减少 CPU 的使用量。我们暂且忽略，直接看 `napi_skb_finish` ，这个函数主要就是调用了 `netif_receive_skb` 。

```
//file: net/core/dev.c
static gro_result_t napi_skb_finish(gro_result_t ret, struct
sk_buff *skb)
{
    switch (ret) {
    case GRO_NORMAL:
        if (netif_receive_skb(skb))
            ret = GRO_DROP;
        break;
    ......
}
```

在 `netif_receive_skb` 中，数据包将被送到协议栈中。

网络协议栈处理

`netif_receive_skb` 函数会根据包的协议，假如是 udp 包，会将包依次送到 ip_rcv()，udp_rcv() 协议处理函数中进行处理。




```
//file: net/core/dev.c
int netif_receive_skb(struct sk_buff *skb)
{
    //RPS处理逻辑，先忽略
    ......

    return __netif_receive_skb(skb);
}

static int __netif_receive_skb(struct sk_buff *skb)
{
    ......
    ret = __netif_receive_skb_core(skb, false);
}

static int __netif_receive_skb_core(struct sk_buff *skb, bool
pfmemalloc)
{
    ......

    //pcap逻辑，这里会将数据送入抓包点。tcpdump就是从这个入口获取包的
    list_for_each_entry_rcu(ptype, &ptype_all, list) {
        if (!ptype->dev || ptype->dev == skb->dev) {
            if (pt_prev)
                ret = deliver_skb(skb, pt_prev, orig_dev);
            pt_prev = ptype;
        }
    }

    ......

    list_for_each_entry_rcu(ptype,
            &ptype_base[ntohs(type) & PTYPE_HASH_MASK], list) {
        if (ptype->type == type &&
```

```
            (ptype->dev == null_or_dev || ptype->dev == skb->dev ||
             ptype->dev == orig_dev)) {
            if (pt_prev)
                ret = deliver_skb(skb, pt_prev, orig_dev);
            pt_prev = ptype;
        }
    }
}
```

在 `__netif_receive_skb_core` 中，我看着了原来经常使用的 tcpdump 的抓包点，很是激动，看来读一遍源代码时间真的没白浪费。接着 `__netif_receive_skb_core` 取出 protocol，它会从数据包中取出协议信息，然后遍历注册在这个协议上的回调函数列表。`ptype_base` 是一个 hash table，在协议注册小节我们提到过。ip_rcv 函数地址就是存在这个 hash table 中的。

```
//file: net/core/dev.c
static inline int deliver_skb(struct sk_buff *skb,
                  struct packet_type *pt_prev,
                  struct net_device *orig_dev)
{
    ......
    return pt_prev->func(skb, skb->dev, pt_prev, orig_dev);
}
```

`pt_prev->func` 这一行就调用到了协议层注册的处理函数了。对于 ip 包来讲，就会进入到 `ip_rcv` （如果是arp包的话，会进入到arp_rcv）。

IP 协议层处理

我们再来大致看一下 linux 在 ip 协议层都做了什么，包又是怎么样进一步被送到 udp 或 tcp 协议处理函数中的。

```
//file: net/ipv4/ip_input.c
int ip_rcv(struct sk_buff *skb, struct net_device *dev, struct
packet_type *pt, struct net_device *orig_dev)
{
    ......

    return NF_HOOK(NFPROTO_IPV4, NF_INET_PRE_ROUTING, skb, dev,
NULL,
                   ip_rcv_finish);
}
```

这里 `NF_HOOK` 是一个钩子函数，当执行完注册的钩子后就会执行到最后一个参数指向的函数 `ip_rcv_finish` 。

```
static int ip_rcv_finish(struct sk_buff *skb)
{
    ......

    if (!skb_dst(skb)) {
        int err = ip_route_input_noref(skb, iph->daddr, iph-
>saddr,
                                       iph->tos, skb->dev);
        ...
    }

    ......

    return dst_input(skb);
}
```

跟踪 `ip_route_input_noref` 后看到它又调用了 `ip_route_input_mc` 。 在 `ip_route_input_mc` 中，函数 `ip_local_deliver` 被赋值给了 `dst.input` ，如下：

```
//file: net/ipv4/route.c
static int ip_route_input_mc(struct sk_buff *skb, __be32 daddr,
__be32 saddr,
                 u8 tos, struct net_device *dev, int our)
{
    if (our) {
        rth->dst.input= ip_local_deliver;
        rth->rt_flags |= RTCF_LOCAL;
    }
}
```

所以回到 `ip_rcv_finish` 中的 `return dst_input(skb)`。

```
/* Input packet from network to transport.  */
static inline int dst_input(struct sk_buff *skb)
{
    return skb_dst(skb)->input(skb);
}
```

`skb_dst(skb)->input` 调用的 input 方法就是路由子系统赋的 ip_local_deliver。

```
//file: net/ipv4/ip_input.c
int ip_local_deliver(struct sk_buff *skb)
{
    /*
     *  Reassemble IP fragments.
     */

    if (ip_is_fragment(ip_hdr(skb))) {
        if (ip_defrag(skb, IP_DEFRAG_LOCAL_DELIVER))
            return 0;
    }

    return NF_HOOK(NFPROTO_IPV4, NF_INET_LOCAL_IN, skb, skb-
>dev, NULL,
                ip_local_deliver_finish);
}
```

```
static int ip_local_deliver_finish(struct sk_buff *skb)
{
    ......

    int protocol = ip_hdr(skb)->protocol;
    const struct net_protocol *ipprot;

    ipprot = rcu_dereference(inet_protos[protocol]);
    if (ipprot != NULL) {
        ret = ipprot->handler(skb);
    }
}
```

如协议注册小节看到 inet_protos 中保存着 tcp_v4_rcv() 和 udp_rcv() 的函数地址。这里将会根据包中的协议类型选择进行分发,在这里 skb 包将会进一步被派送到更上层的协议中，udp 和 tcp。

1.4 总结

网络模块是 Linux 内核中最复杂的模块了，看起来一个简简单单的收包过程就涉及到许多内核组件之间的交互，如网卡驱动、协议栈，内核 ksoftirqd 线程等。
看起来很复杂，本文想通过源码 + 图示的方式，尽量以容易理解的方式来将内核收包过程讲清楚。现在让我们再串一串整个收包过程。

当用户执行完 `recvfrom` 调用后，用户进程就通过系统调用进行到内核态工作了。如果接收队列没有数据，进程就进入睡眠状态被操作系统挂起。这块相对比较简单，剩下大部分的戏份都是由 Linux 内核其它模块来表演了。

首先在开始收包之前，Linux 要做许多的准备工作：

- 创建ksoftirqd线程，为它设置好它自己的线程函数，后面就指望着它来处理软中断呢。
- 协议栈注册，linux要实现许多协议，比如arp，icmp，ip，udp，tcp，每一个协议都会将自己的处理函数注册一下，方便包来了迅速找到对应的处理函数
- 网卡驱动初始化，每个驱动都有一个初始化函数，内核会让驱动也初始化一下。在这个初始化过程中，把自己的DMA准备好，把NAPI的poll函数地址告诉内核
- 启动网卡，分配RX，TX队列，注册中断对应的处理函数

以上是内核准备收包之前的重要工作，当上面都 ready 之后，就可以打开硬中断，等待数据包的到来了。

当数据到到来了以后，第一个迎接它的是网卡（我去，这不是废话么）：

- 网卡将数据帧 DMA 到内存的 RingBuffer 中，然后向 CPU 发起中断通知
- CPU 响应中断请求，调用网卡启动时注册的中断处理函数
- 中断处理函数几乎没干啥，就发起了软中断请求
- 内核线程 ksoftirqd 线程发现有软中断请求到来，先关闭硬中断
- ksoftirqd 线程开始调用驱动的 poll 函数收包
- poll 函数将收到的包送到协议栈注册的 ip_rcv 函数中
- ip_rcv 函数再将包送到 udp_rcv 函数中（对于 tcp 包就送到 tcp_rcv ）

二、内核是如何与进程协作的

2.1 UDP 下用户进程如何和内核协同?

2.1.1 UDP 协议处理

在协议注册小节的时候我们说过，udp协议的处理函数是 `udp_rcv` 。

```
//file: net/ipv4/udp.c
int udp_rcv(struct sk_buff *skb)
{
    return __udp4_lib_rcv(skb, &udp_table, IPPROTO_UDP);
}

int __udp4_lib_rcv(struct sk_buff *skb, struct udp_table
*udptable,
           int proto)
{
    sk = __udp4_lib_lookup_skb(skb, uh->source, uh->dest,
udptable);

    if (sk != NULL) {
        int ret = udp_queue_rcv_skb(sk, skb
    }

    icmp_send(skb, ICMP_DEST_UNREACH, ICMP_PORT_UNREACH, 0);
}
```

`__udp4_lib_lookup_skb` 是根据 skb 来寻找对应的socket，当找到以后将数据包放到 socket 的缓存队列里。如果没有找到，则发送一个目标不可达的 icmp 包。

```
//file: net/ipv4/udp.c
int udp_queue_rcv_skb(struct sock *sk, struct sk_buff *skb)
{
    ......

    if (sk_rcvqueues_full(sk, skb, sk->sk_rcvbuf))
        goto drop;

        rc = 0;

    ipv4_pktinfo_prepare(skb);
    bh_lock_sock(sk);
    if (!sock_owned_by_user(sk))
        rc = __udp_queue_rcv_skb(sk, skb);
    else if (sk_add_backlog(sk, skb, sk->sk_rcvbuf)) {
        bh_unlock_sock(sk);
        goto drop;
    }
    bh_unlock_sock(sk);

    return rc;
}
```

sock_owned_by_user 判断的是用户是不是正在这个 socket 上进行系统调用（socket 被占用），如果没有，那就可以直接放到 socket 的接收队列中。如果有，那就通过 `sk_add_backlog` 把数据包添加到 backlog 队列。 当用户释放的 socket 的时候，内核会检查 backlog 队列，如果有数据再移动到接收队列中。

`sk_rcvqueues_full` 接收队列如果满了的话，将直接把包丢弃。接收队列大小受内核参数 net.core.rmem_max 和 net.core.rmem_default 影响。

2.1.2 recvfrom 系统调用实现

花开两朵，各表一枝。 上面我们说完了整个 Linux 内核对数据包的接收和处理过程，最后把数据包放到 socket 的接收队列中了。那么我们再回头看用户进程调用 `recvfrom` 后是发生了什么。 我们在代码里调用的 `recvfrom` 是一个 glibc 的库函数，该函数在执行后会将用户进行陷入到内核态，进入到 Linux 实现的系统调用 `sys_recvfrom` 。在理解Linux对 `sys_recvfrom` 之前，我们先来简单看一下 `socket` 这个核心数据结构。这个数据结构太大了，我们只把对和我们今天主题相关的内容画出来，如下：

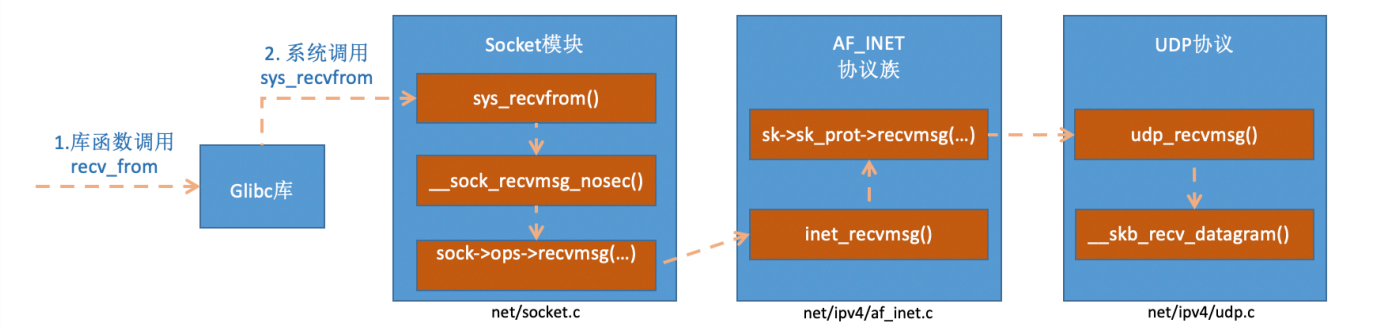
`socket` 数据结构中的 `const struct proto_ops` 对应的是协议的方法集合。每个协议都会实现不同的方法集，对于IPv4 Internet 协议族来说,每种协议都有对应的处理方法，如下。对于 udp 来说，是通过 `inet_dgram_ops` 来定义的，其中注册了 `inet_recvmsg` 方法。

```
//file: net/ipv4/af_inet.c
const struct proto_ops inet_stream_ops = {
    ......
    .recvmsg        = inet_recvmsg,
    .mmap           = sock_no_mmap,
    ......
}
const struct proto_ops inet_dgram_ops = {
    ......
    .sendmsg        = inet_sendmsg,
    .recvmsg        = inet_recvmsg,
    ......
}
```

`socket` 数据结构中的另一个数据结构 `struct sock *sk` 是一个非常大，非常重要的子结构体。其中的 `sk_prot` 又定义了二级处理函数。对于udp协议来说，会被设置成 udp 协议实现的方法集 `udp_prot` 。

```
//file: net/ipv4/udp.c
struct proto udp_prot = {
    .name           = "UDP",
    .owner          = THIS_MODULE,
    .close          = udp_lib_close,
    .connect        = ip4_datagram_connect,
    ......
    .sendmsg        = udp_sendmsg,
    .recvmsg        = udp_recvmsg,
    .sendpage       = udp_sendpage,
    ......
}
```

看完了 `socket` 变量之后，我们再来看 `sys_recvfrom` 的实现过程。



在 `inet_recvmsg` 调用了 `sk->sk_prot->recvmsg` 。

```
//file: net/ipv4/af_inet.c
int inet_recvmsg(struct kiocb *iocb, struct socket *sock,
struct msghdr *msg,
         size_t size, int flags)
{
    ......
    err = sk->sk_prot->recvmsg(iocb, sk, msg, size, flags &
MSG_DONTWAIT,
                    flags & ~MSG_DONTWAIT, &addr_len);
    if (err >= 0)
        msg->msg_namelen = addr_len;
    return err;
}
```

```
//file: net/core/datagram.c:EXPORT_SYMBOL(__skb_recv_datagram);
struct sk_buff *__skb_recv_datagram(struct sock *sk, unsigned
int flags,
                    int *peeked, int *off, int *err)
{
    ......
    do {
        struct sk_buff_head *queue = &sk->sk_receive_queue;
        skb_queue_walk(queue, skb) {
            ......
        }

        /* User doesn't want to wait */
        error = -EAGAIN;
        if (!timeo)
            goto no_packet;
    } while (!wait_for_more_packets(sk, err, &timeo, last));
}
```

终于我们找到了我们想要看的重点，在上面我们看到了所谓的读取过程，就是访问 `sk->sk_receive_queue` 。如果没有数据，且用户也允许等待，则将调用 wait_for_more_packets() 执行等待操作，它加入会让用户进程进入睡眠状态。

具体是怎么进入睡眠状态的，让我们下一小节里和 TCP 一起来看，睡眠是相同的。

2.2 TCP 下用户进程如何和内核协同?

在网络开发模型中，有一种非常易于开发同学使用方式，那就是同步阻塞的网络 IO（在 Java 中习惯叫 BIO）。

例如我们想请求服务器上的一段数据，那么 C 语言的一段代码 demo 大概是下面这样：

```
int main()
{
  int sk = socket(AF_INET, SOCK_STREAM, 0);
  connect(sk, ...)
  recv(sk, ...)
}
```

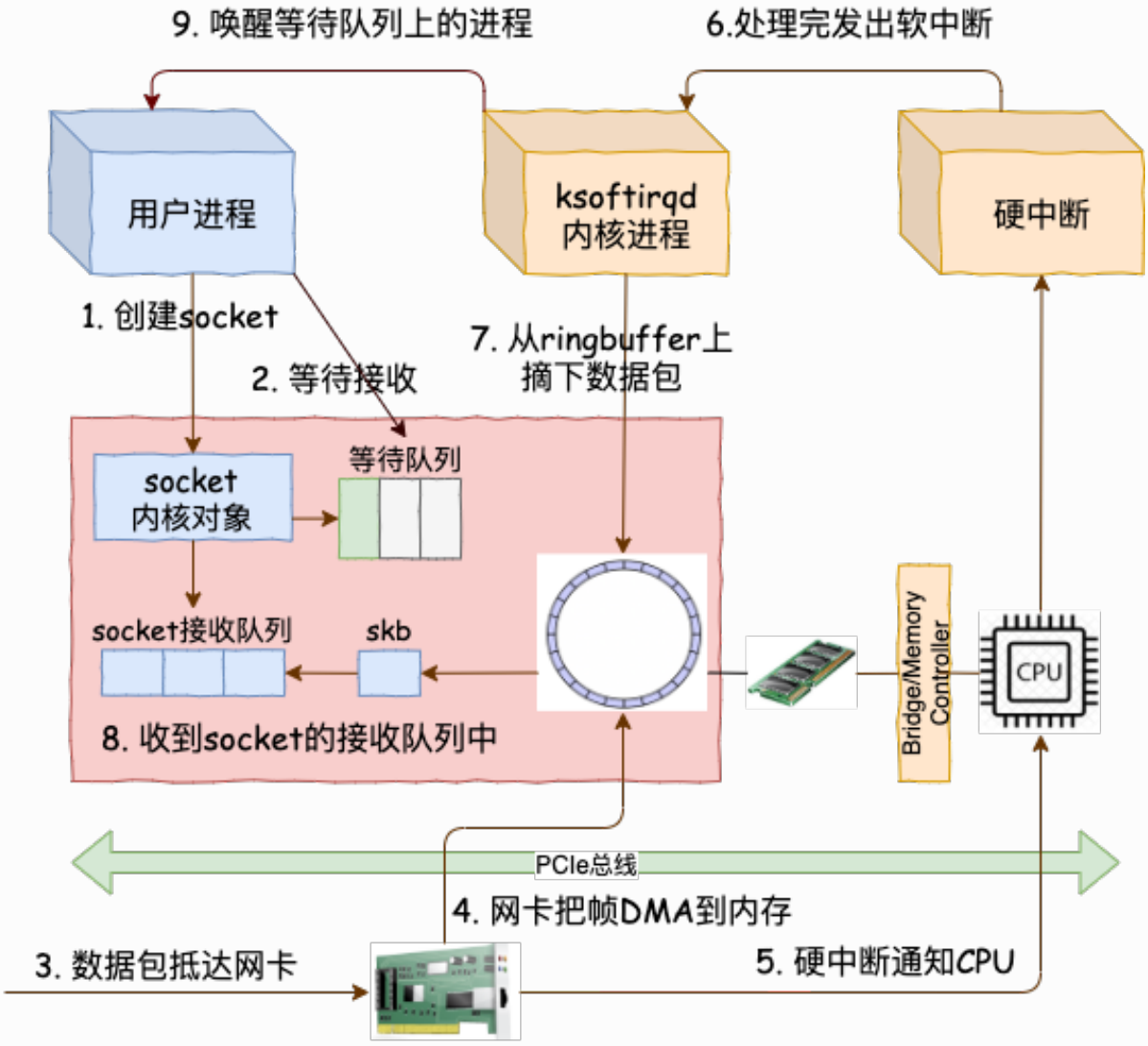
但是在高并发的服务器开发中，这种网络 IO 的性能奇差。因为

- 1.进程在 recv 的时候大概率会被阻塞掉，导致一次进程切换
- 2.当连接上数据就绪的时候进程又会被唤醒，又是一次进程切换
- 3.一个进程同时只能等待一条连接，如果有很多并发，则需要很多进程

如果用一句话来概括，那就是：**同步阻塞网络 IO 是高性能网络开发路上的绊脚石！** 俗话说得好，知己知彼方能百战百胜。所以我们今天先不讲优化，只深入分析同步阻塞网络 IO 的内部实现。

在上面的 demo 中虽然只是简单的两三行代码，但实际上用户进程和内核配合做了非常多的工作。先是用户进程发起创建 socket 的指令，然后切换到内核态完成了内核对象的初始化。接下来 Linux 在数据包的接收上，是硬中断和 ksoftirqd 进程在进行处理。当 ksoftirqd 进程处理完以后，再通知到相关的用户进程。

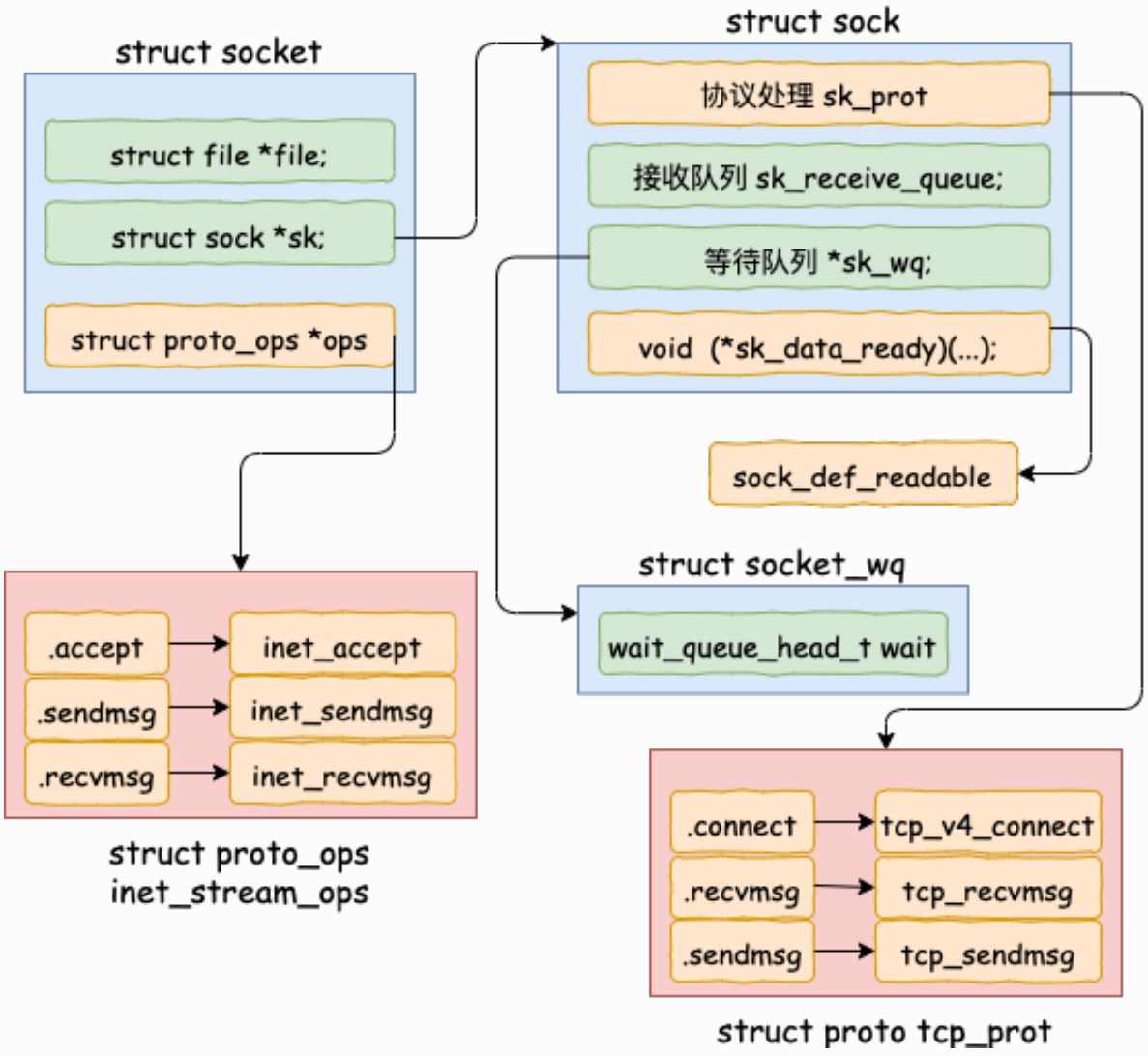
从用户进程创建 socket，到一个网络包抵达网卡到被用户进程接收到，总体上的流程图如下：



我们今天用图解加源码分析的方式来详细拆解一下上面的每一个步骤，来看一下在内核里是它们是怎么实现的。阅读完本文，你将深刻地理解在同步阻塞的网络 IO 性能低下的原因！

2.2.1 socket 的创建

开篇源码中的 socket 函数调用执行完以后，内核在内部创建了一系列的 socket 相关的内核对象（是的，不是只有一个）。它们互相之间的关系如图。当然了，这个对象比图示的还要更复杂。我只在图中把和今天的主题相关的内容展现了出来。



我们来翻翻源码，看下上面的结构是如何被创造出来的。

```
//file:net/socket.c
SYSCALL_DEFINE3(socket, int, family, int, type, int, protocol)
{
  ......
  retval = sock_create(family, type, protocol, &sock);
}
```

sock_create 是创建 socket 的主要位置。其中 sock_create 又调用了 __sock_create。

```
//file:net/socket.c
int __sock_create(struct net *net, int family, int type, int
protocol,
        struct socket **res, int kern)
{
  struct socket *sock;
```

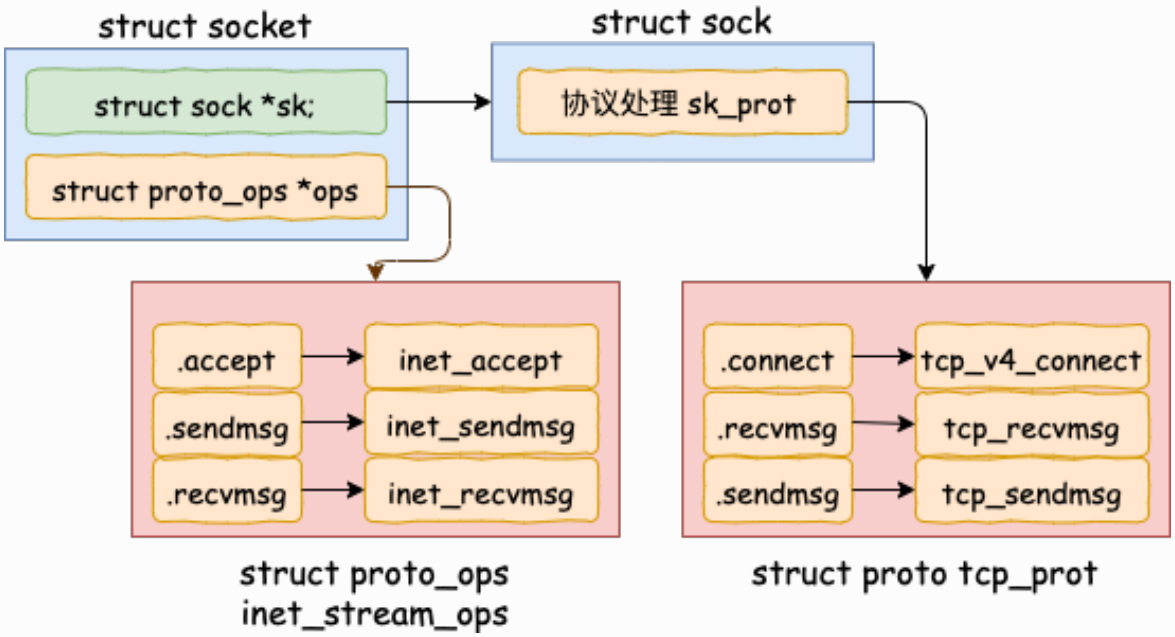
```
  const struct net_proto_family *pf;

  ......

  //分配 socket 对象
  sock = sock_alloc();

  //获得每个协议族的操作表
  pf = rcu_dereference(net_families[family]);

  //调用每个协议族的创建函数， 对于 AF_INET 对应的是
  err = pf->create(net, sock, protocol, kern);
}
```

在 __sock_create 里，首先调用 sock_alloc 来分配一个 struct sock 对象。 接着在获取协议族的操作函数表，并调用其 create 方法。 对于 AF_INET 协议族来说，执行到的是 inet_create 方法。

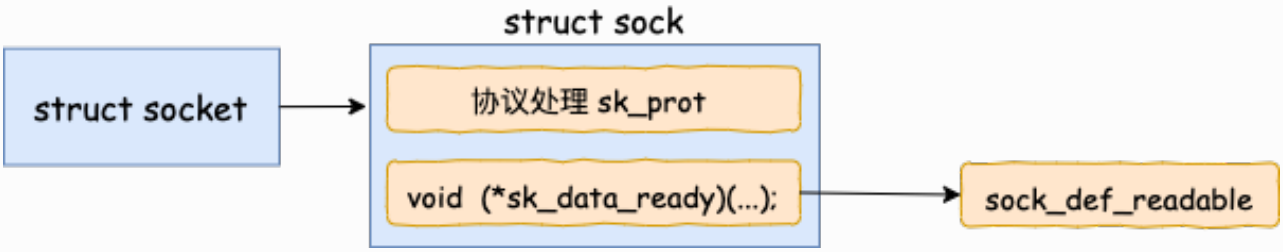
```
//file:net/ipv4/af_inet.c
tatic int inet_create(struct net *net, struct socket *sock, int
protocol,
            int kern)
{
  struct sock *sk;

  //查找对应的协议，对于TCP SOCK_STREAM 就是获取到了
  //static struct inet_protosw inetsw_array[] =
    //{
  //    {
  //       .type =       SOCK_STREAM,
  //       .protocol =   IPPROTO_TCP,
  //       .prot =       &tcp_prot,
  //       .ops =        &inet_stream_ops,
  //       .no_check =   0,
  //       .flags =      INET_PROTOSW_PERMANENT |
  //                INET_PROTOSW_ICSK,
  //    },
  //}
    list_for_each_entry_rcu(answer, &inetsw[sock->type], list)
{
```

```
    //将 inet_stream_ops 赋到 socket->ops 上
    sock->ops = answer->ops;

    //获得 tcp_prot
    answer_prot = answer->prot;

    //分配 sock 对象，  并把 tcp_prot 赋到 sock->sk_prot 上
    sk = sk_alloc(net, PF_INET, GFP_KERNEL, answer_prot);

    //对 sock 对象进行初始化
    sock_init_data(sock, sk);
}
```

在 inet_create 中，根据类型 SOCK_STREAM 查找到对于 tcp 定义的操作方法实现集合 inet_stream_ops 和 tcp_prot。并把它们分别设置到 socket->ops 和 sock->sk_prot 上。



我们再往下看到了 sock_init_data。 在这个方法中将 sock 中的 sk_data_ready 函数指针进行了初始化，设置为默认 sock_def_readable()。



```
//file: net/core/sock.c
void sock_init_data(struct socket *sock, struct sock *sk)
{
    sk->sk_data_ready   =   sock_def_readable;
    sk->sk_write_space  =   sock_def_write_space;
    sk->sk_error_report =   sock_def_error_report;
}
```

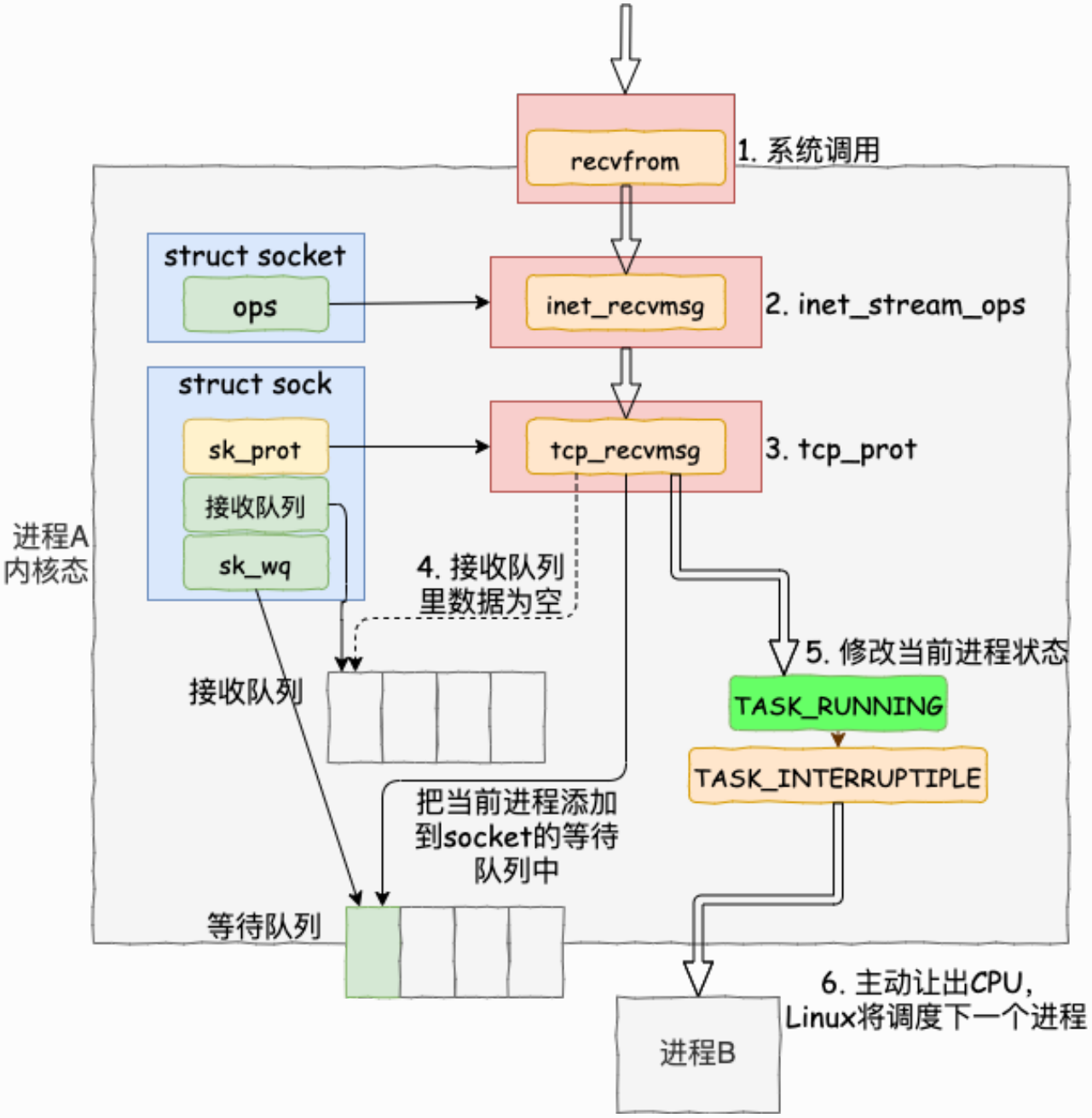
当软中断上收到数据包时会通过调用 sk_data_ready 函数指针（实际被设置成了 sock_def_readable()） 来唤醒在 sock 上等待的进程。这个咱们后面介绍软中断的时候再说，这里记住这个就行了。

至此，一个 tcp对象，确切地说是 AF_INET 协议族下 SOCK_STREAM对象就算是创建完成了。**这里花费了一次 socket 系统调用的开销。**

2.2.2 等待接收消息

接着我们来看 recv 函数依赖的底层实现。首先通过 strace 命令跟踪，可以看到 clib 库函数 recv 会执行到 recvfrom 系统调用。

进入系统调用后，用户进程就进入到了内核态，通过执行一系列的内核协议层函数，然后到 socket 对象的接收队列中查看是否有数据，没有的话就把自己添加到 socket 对应的等待队列里。最后让出CPU，操作系统会选择下一个就绪状态的进程来执行。 整个流程图如下：



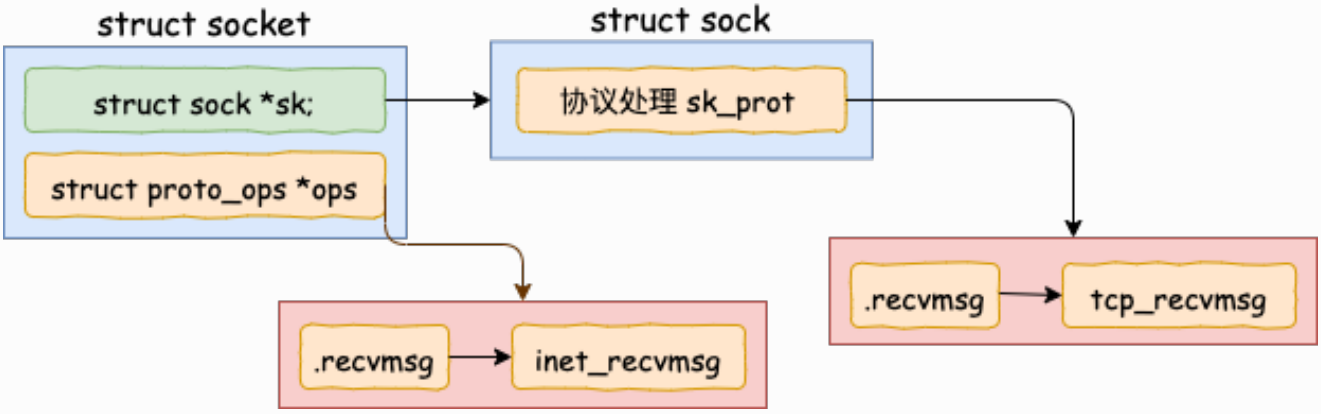
看完了整个流程图，接下来让我们根据源码来看更详细的细节。其中**我们今天要关注的重点是 recvfrom 最后是怎么把自己的进程给阻塞掉的**（假如我们没有使用 O_NONBLOCK 标记）。

```
//file: net/socket.c
SYSCALL_DEFINE6(recvfrom, int, fd, void __user *, ubuf, size_t,
size,
    unsigned int, flags, struct sockaddr __user *, addr,
    int __user *, addr_len)
{
  struct socket *sock;

  //根据用户传入的 fd 找到 socket 对象
  sock = sockfd_lookup_light(fd, &err, &fput_needed);
  ......
  err = sock_recvmsg(sock, &msg, size, flags);
  ......
}
```

sock_recvmsg ==> __sock_recvmsg => __sock_recvmsg_nosec

```
static inline int __sock_recvmsg_nosec(struct kiocb *iocb,
struct socket *sock,
                struct msghdr *msg, size_t size, int flags)
{
  ......
  return sock->ops->recvmsg(iocb, sock, msg, size, flags);
}
```

调用 socket 对象 ops 里的 recvmsg， 回忆我们上面的 socket 对象图，从图中可以看到 recvmsg 指向的是 inet_recvmsg 方法。

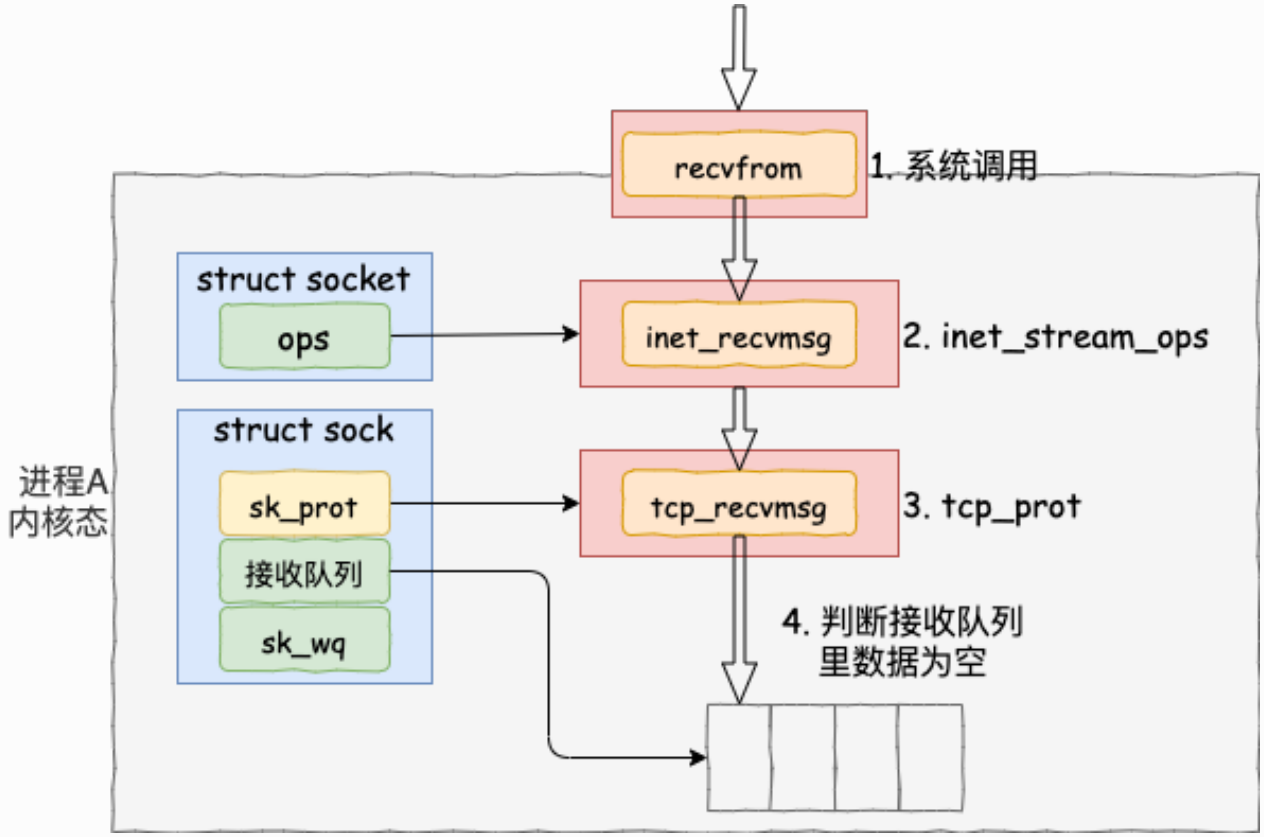


```
//file: net/ipv4/af_inet.c
int inet_recvmsg(struct kiocb *iocb, struct socket *sock,
struct msghdr *msg,
     size_t size, int flags)
{
  ...

  err = sk->sk_prot->recvmsg(iocb, sk, msg, size, flags &
MSG_DONTWAIT,
            flags & ~MSG_DONTWAIT, &addr_len);
```

这里又遇到一个函数指针，这次调用的是 socket 对象里的 sk_prot 下面的 recvmsg方法。同上，得出这个 recvmsg 方法对应的是 tcp_recvmsg 方法。

```
//file: net/ipv4/tcp.c
int tcp_recvmsg(struct kiocb *iocb, struct sock *sk, struct
msghdr *msg,
    size_t len, int nonblock, int flags, int *addr_len)
{
  int copied = 0;
  ...
  do {
    //遍历接收队列接收数据
    skb_queue_walk(&sk->sk_receive_queue, skb) {
      ...
    }
    ...
  }

  if (copied >= target) {
    release_sock(sk);
    lock_sock(sk);
  } else //没有收到足够数据，启用 sk_wait_data 阻塞当前进程
    sk_wait_data(sk, &timeo);
}
```

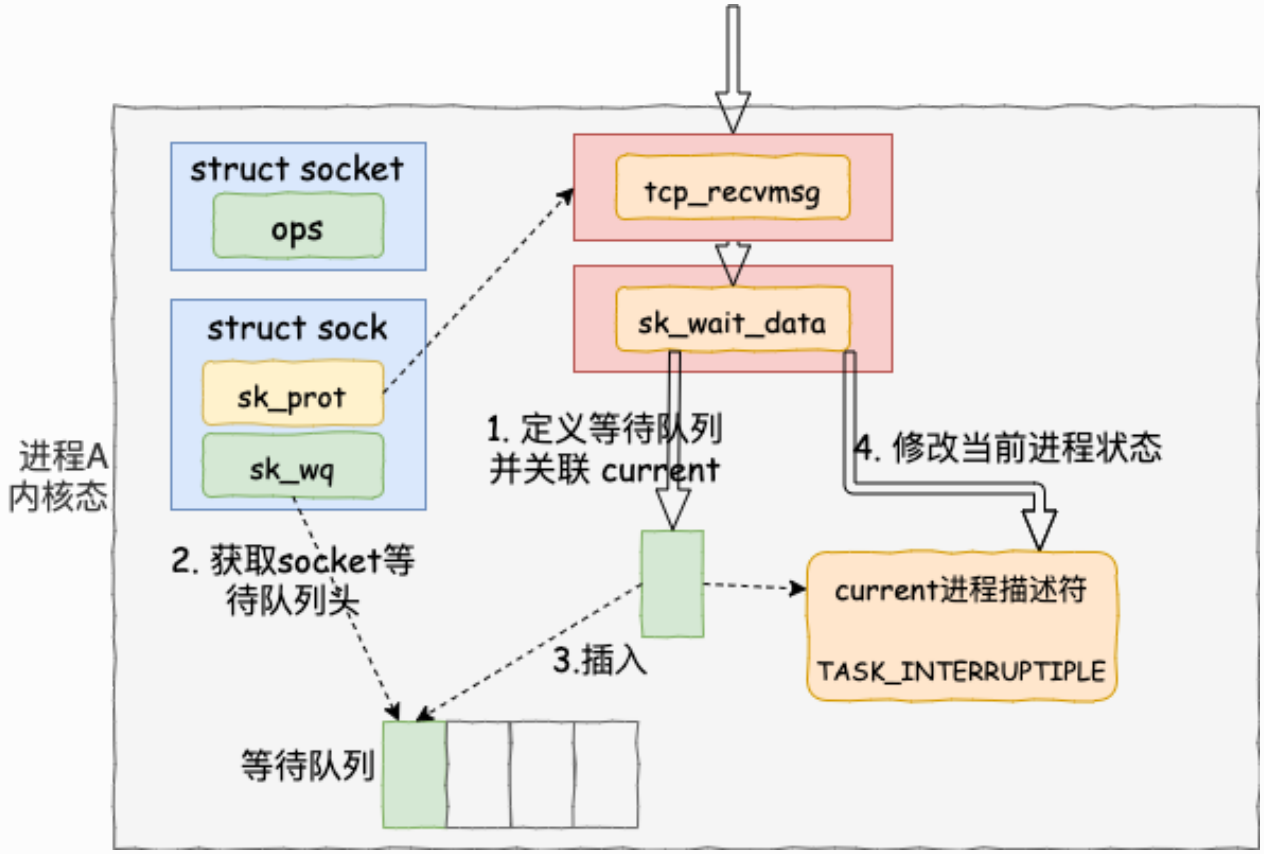
终于看到了我们想要看的东西，skb_queue_walk 是在访问 sock 对象下面的接收队列了。



如果没有收到数据，或者收到不足够多，则调用 sk_wait_data 把当前进程阻塞掉。

```
//file: net/core/sock.c
int sk_wait_data(struct sock *sk, long *timeo)
{
  //当前进程(current)关联到所定义的等待队列项上
  DEFINE_WAIT(wait);

  // 调用 sk_sleep 获取 sock 对象下的 wait
  // 并准备挂起，将进程状态设置为可打断 INTERRUPTIBLE
  prepare_to_wait(sk_sleep(sk), &wait, TASK_INTERRUPTIBLE);
  set_bit(SOCK_ASYNC_WAITDATA, &sk->sk_socket->flags);

  // 通过调用schedule_timeout让出CPU，然后进行睡眠
  rc = sk_wait_event(sk, timeo, !skb_queue_empty(&sk-
>sk_receive_queue));
  ...
```

我们再来详细看下 sk_wait_data 是怎么把当前进程给阻塞掉的。



首先在 DEFINE_WAIT 宏下，定义了一个等待队列项 wait。 在这个新的等待队列项上，注册了回调函数 autoremove_wake_function，并把当前进程描述符 current 关联到其 .private 成员上。

```
//file: include/linux/wait.h
#define DEFINE_WAIT(name) DEFINE_WAIT_FUNC(name,
autoremove_wake_function)

#define DEFINE_WAIT_FUNC(name, function)        \
  wait_queue_t name = {             \
    .private  = current,          \
    .func   = function,         \
    .task_list  = LIST_HEAD_INIT((name).task_list), \
  }
```

紧接着在 sk_wait_data 中 调用 sk_sleep 获取 sock 对象下的等待队列列表头 wait_queue_head_t。sk_sleep 源代码如下：

```
//file: include/net/sock.h
static inline wait_queue_head_t *sk_sleep(struct sock *sk)
{
  BUILD_BUG_ON(offsetof(struct socket_wq, wait) != 0);
  return &rcu_dereference_raw(sk->sk_wq)->wait;
}
```

接着调用 prepare_to_wait 来把新定义的等待队列项 wait 插入到 sock 对象的等待队列下。

```
//file: kernel/wait.c
void
prepare_to_wait(wait_queue_head_t *q, wait_queue_t *wait, int
state)
{
  unsigned long flags;

  wait->flags &= ~WQ_FLAG_EXCLUSIVE;
  spin_lock_irqsave(&q->lock, flags);
  if (list_empty(&wait->task_list))
    __add_wait_queue(q, wait);
  set_current_state(state);
  spin_unlock_irqrestore(&q->lock, flags);
}
```

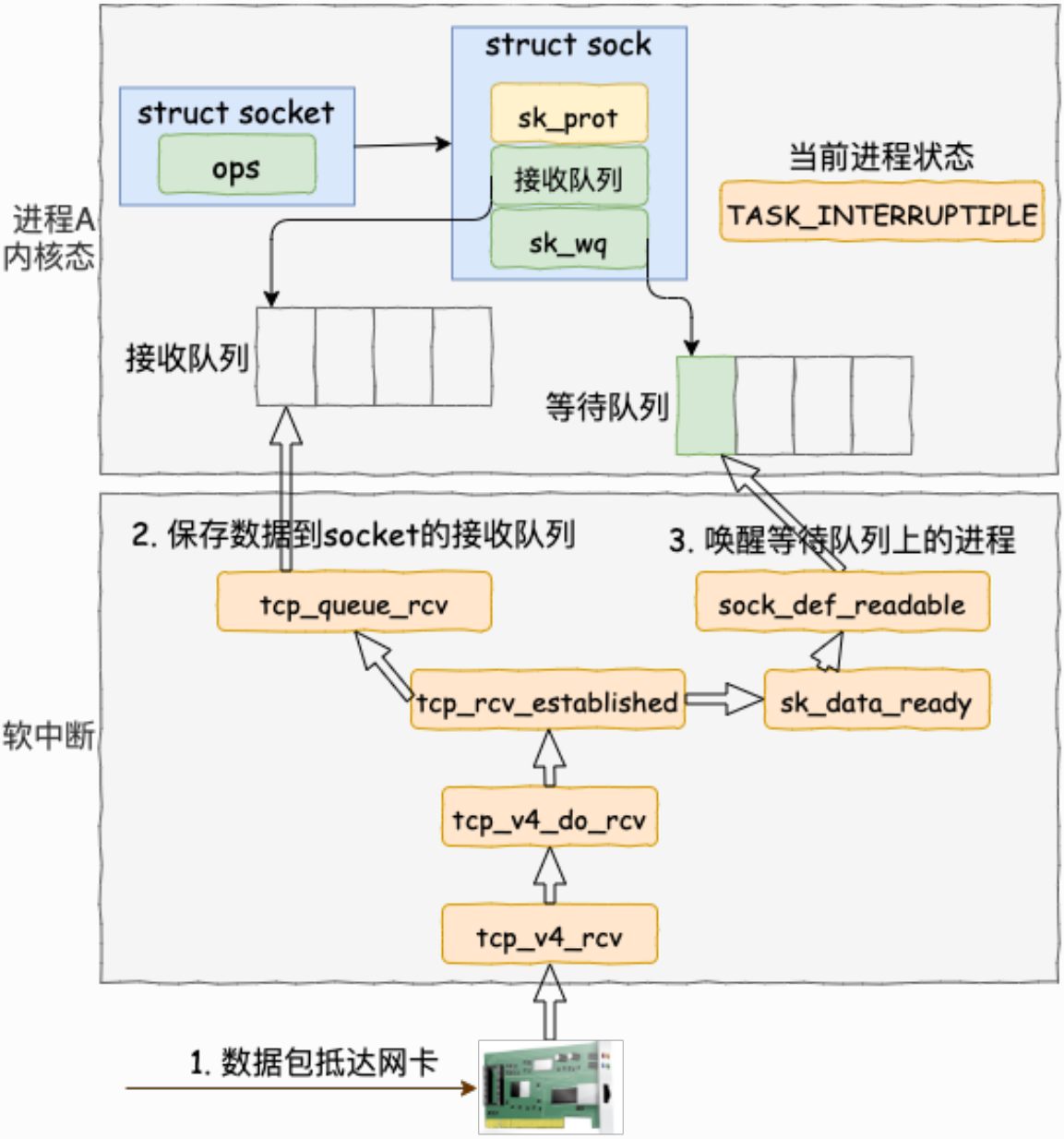
这样后面当内核收完数据产生就绪时间的时候，就可以查找 socket 等待队列上的等待项，进而就可以找到回调函数和在等待该 socket 就绪事件的进程了。

最后再调用 sk_wait_event 让出 CPU，进程将进入睡眠状态，**这会导致一次进程上下文的开销**。

接下来的小节里我们将能看到进程是如何被唤醒的了。

2.2.3 软中断模块

接着我们再转换一下视角，来看负责接收和处理数据包的软中断这边。前文看到了关于网络包到网卡后是怎么被网卡接收，最后在交由软中断处理的。我们今天直接从 tcp 协议的接收函数 tcp_v4_rcv 看起。



软中断（也就是 Linux 里的 ksoftirqd 进程）里收到数据包以后，发现是 tcp 的包的话就会执行到 tcp_v4_rcv 函数。接着走，如果是 ESTABLISH 状态下的数据包，则最终会把数据拆出来放到对应 socket 的接收队列中。然后调用 sk_data_ready 来唤醒用户进程。

我们看更详细一点的代码：

```
// file: net/ipv4/tcp_ipv4.c
int tcp_v4_rcv(struct sk_buff *skb)
{
  ......
  th = tcp_hdr(skb); //获取tcp header
  iph = ip_hdr(skb); //获取ip header

  //根据数据包 header 中的 ip、端口信息查找到对应的socket
```

```
  sk = __inet_lookup_skb(&tcp_hashinfo, skb, th->source, th-
>dest);
  ......

  //socket 未被用户锁定
  if (!sock_owned_by_user(sk)) {
    {
      if (!tcp_prequeue(sk, skb))
        ret = tcp_v4_do_rcv(sk, skb);
    }
  }
}
```

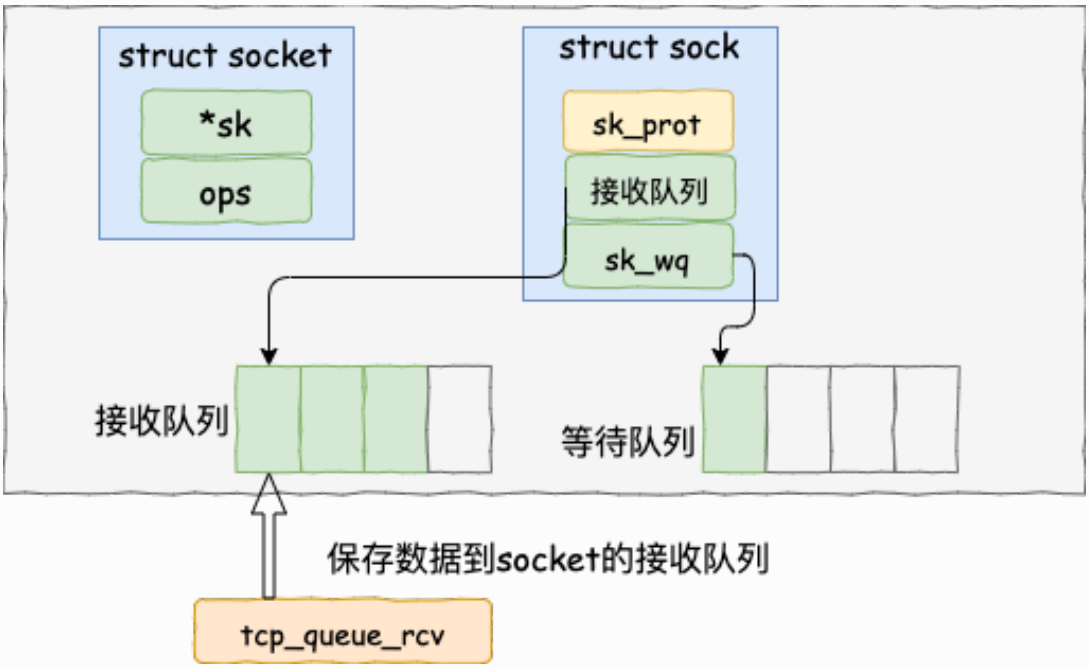
在 tcp_v4_rcv 中首先根据收到的网络包的 header 里的 source 和 dest 信息来在本机上查询对应的 socket。找到以后，我们直接进入接收的主体函数 tcp_v4_do_rcv 来看。

```
//file: net/ipv4/tcp_ipv4.c
int tcp_v4_do_rcv(struct sock *sk, struct sk_buff *skb)
{
  if (sk->sk_state == TCP_ESTABLISHED) {

    //执行连接状态下的数据处理
    if (tcp_rcv_established(sk, skb, tcp_hdr(skb), skb->len)) {
      rsk = sk;
      goto reset;
    }
    return 0;
  }

  //其它非 ESTABLISH 状态的数据包处理
  ......
}
```

我们假设处理的是 ESTABLISH 状态下的包，这样就又进入 tcp_rcv_established 函数中进行处理。

```
//file: net/ipv4/tcp_input.c
int tcp_rcv_established(struct sock *sk, struct sk_buff *skb,
      const struct tcphdr *th, unsigned int len)
{
  ......

  //接收数据到队列中
  eaten = tcp_queue_rcv(sk, skb, tcp_header_len,
                &fragstolen);

  //数据 ready，唤醒 socket 上阻塞掉的进程
  sk->sk_data_ready(sk, 0);
```

在 tcp_rcv_established 中通过调用 tcp_queue_rcv 函数中完成了将接收数据放到 socket 的接收队列上。



如下源码所示

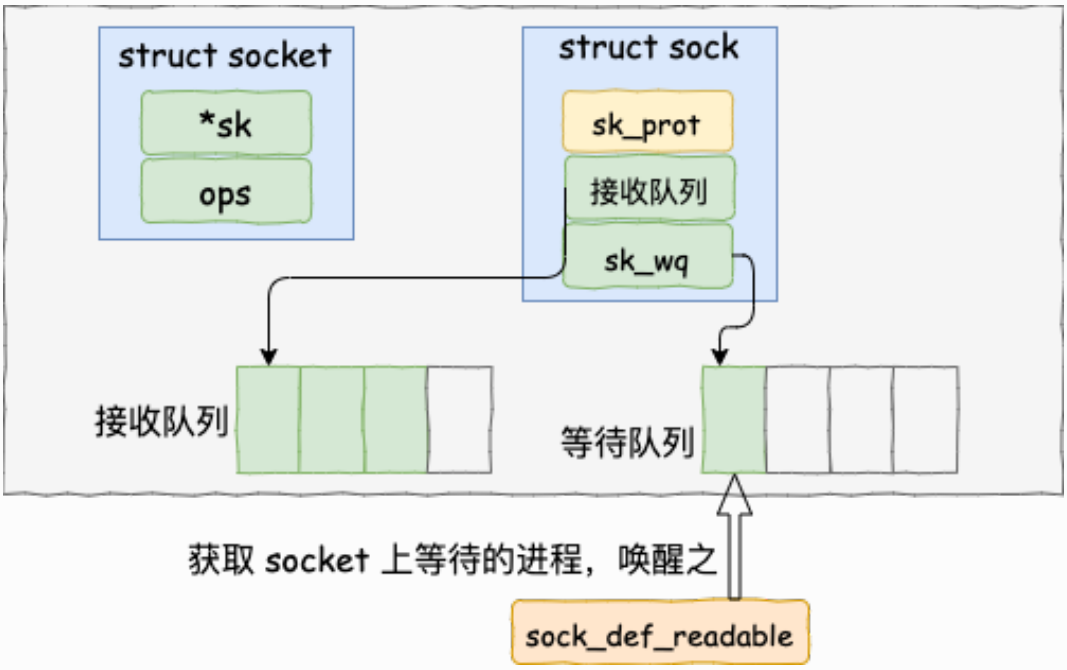
```
//file: net/ipv4/tcp_input.c
static int __must_check tcp_queue_rcv(struct sock *sk, struct
sk_buff *skb, int hdrlen,
      bool *fragstolen)
{
  //把接收到的数据放到 socket 的接收队列的尾部
  if (!eaten) {
    __skb_queue_tail(&sk->sk_receive_queue, skb);
    skb_set_owner_r(skb, sk);
  }
  return eaten;
}
```

调用 tcp_queue_rcv 接收完成之后，接着再调用 sk_data_ready 来唤醒在socket上等待的用户进程。 这又是一个函数指针。回想上面我们在 创建 socket 流程里执行到的 sock_init_data 函数，在这个函数里已经把 sk_data_ready 设置成 sock_def_readable 函数了（可以ctrl + f 搜索前文）。它是默认的数据就绪处理函数。

```
//file: net/core/sock.c
static void sock_def_readable(struct sock *sk, int len)
{
  struct socket_wq *wq;

  rcu_read_lock();
  wq = rcu_dereference(sk->sk_wq);

  //有进程在此 socket 的等待队列
  if (wq_has_sleeper(wq))
    //唤醒等待队列上的进程
    wake_up_interruptible_sync_poll(&wq->wait, POLLIN | POLLPRI
|
            POLLRDNORM | POLLRDBAND);
  sk_wake_async(sk, SOCK_WAKE_WAITD, POLL_IN);
  rcu_read_unlock();
}
```

在 sock_def_readable 中再一次访问到了 sock->sk_wq 下的wait。回忆下我们前面调用 recvfrom 执行的最后，通过 DEFINE_WAIT(wait) 将当前进程关联的等待队列添加到 sock->sk_wq 下的 wait 里了。

那接下来就是调用 wake_up_interruptible_sync_poll 来唤醒在 socket 上因为等待数据而被阻塞掉的进程了。




```
//file: include/linux/wait.h
#define wake_up_interruptible_sync_poll(x, m)        \
  __wake_up_sync_key((x), TASK_INTERRUPTIBLE, 1, (void *) (m))
```

```
//file: kernel/sched/core.c
void __wake_up_sync_key(wait_queue_head_t *q, unsigned int
mode,
      int nr_exclusive, void *key)
{
  unsigned long flags;
  int wake_flags = WF_SYNC;

  if (unlikely(!q))
    return;
```

```
    if (unlikely(!nr_exclusive))
      wake_flags = 0;

    spin_lock_irqsave(&q->lock, flags);
    __wake_up_common(q, mode, nr_exclusive, wake_flags, key);
    spin_unlock_irqrestore(&q->lock, flags);
}
```

__wake_up_common 实现唤醒。这里注意下， 该函数调用是参数 nr_exclusive 传入的是 1，这里指的是**即使是有多个进程都阻塞在同一个 socket 上，也只唤醒 1 个进程。其作用是为了避免惊群。**

```
//file: kernel/sched/core.c
static void __wake_up_common(wait_queue_head_t *q, unsigned int
mode,
      int nr_exclusive, int wake_flags, void *key)
{
  wait_queue_t *curr, *next;

  list_for_each_entry_safe(curr, next, &q->task_list,
task_list) {
    unsigned flags = curr->flags;

    if (curr->func(curr, mode, wake_flags, key) &&
        (flags & WQ_FLAG_EXCLUSIVE) && !--nr_exclusive)
      break;
  }
}
```

在 __wake_up_common 中找出一个等待队列项 curr，然后调用其 curr->func。回忆我们前面在 recv 函数执行的时候，使用 DEFINE_WAIT() 定义等待队列项的细节，内核把 curr->func 设置成了 autoremove_wake_function。

```
//file: include/linux/wait.h
#define DEFINE_WAIT(name) DEFINE_WAIT_FUNC(name,
autoremove_wake_function)

#define DEFINE_WAIT_FUNC(name, function)        \
  wait_queue_t name = {             \
    .private  = current,          \
    .func   = function,         \
    .task_list  = LIST_HEAD_INIT((name).task_list), \
  }
```

在 autoremove_wake_function 中，调用了 default_wake_function。

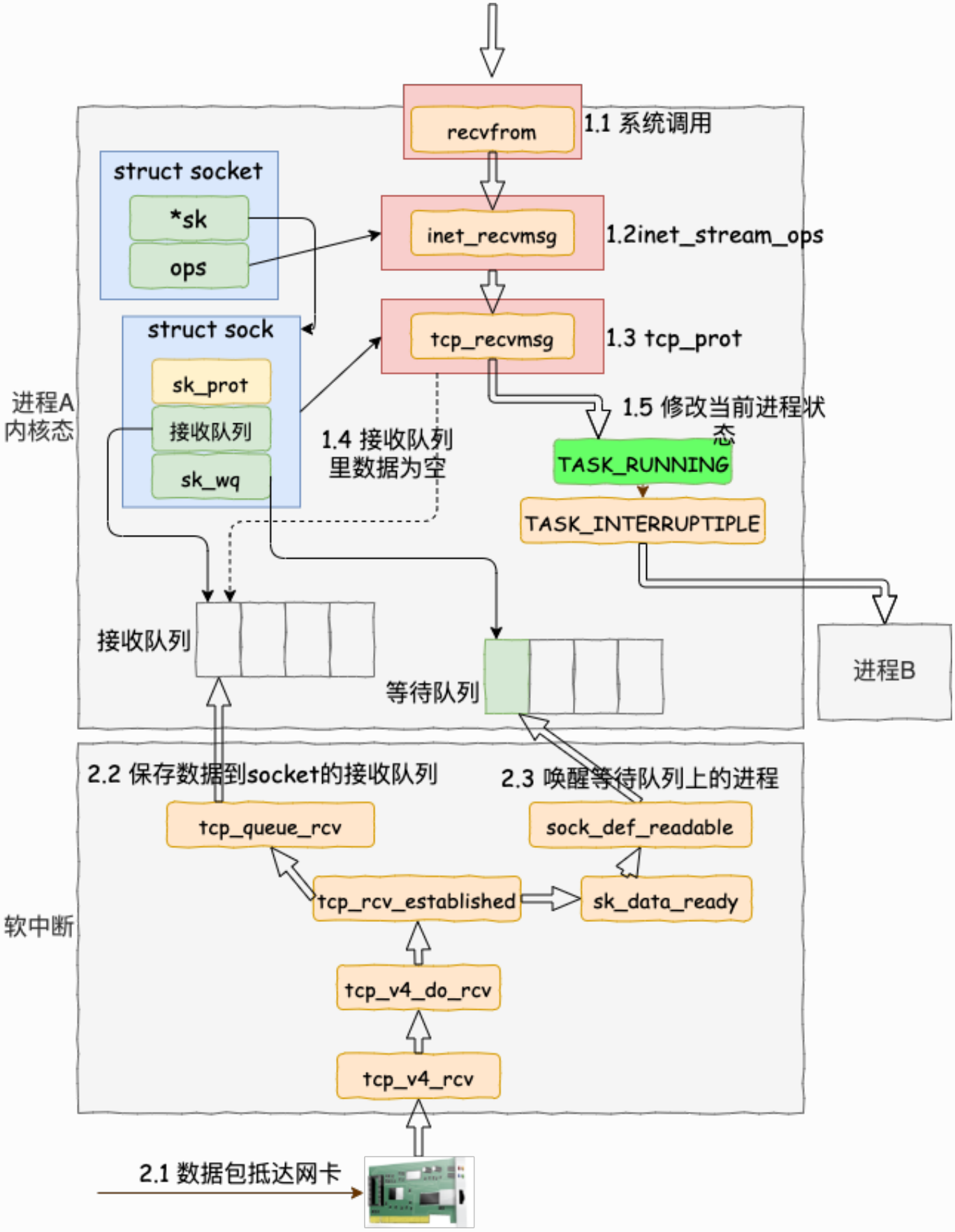
```
//file: kernel/sched/core.c
int default_wake_function(wait_queue_t *curr, unsigned mode,
int wake_flags,
        void *key)
{
  return try_to_wake_up(curr->private, mode, wake_flags);
}
```

调用 try_to_wake_up 时传入的 task_struct 是 curr->private。这个就是当时因为等待而被阻塞的进程项。 当这个函数执行完的时候，**在 socket 上等待而被阻塞的进程就被推入到可运行队列里了，这又将是一次进程上下文切换的开销。**

2.2.4 小结

好了，我们把上面的流程总结一下。 内核在通知网络包的运行环境分两部分：

- 第一部分是我们自己代码所在的进程，我们调用的 socket() 函数会进入内核态创建必要内核对象。 recv() 函数在进入内核态以后负责查看接收队列，以及在没有数据可处理的时候把当前进程阻塞掉，让出 CPU。
- 第二部分是硬中断、软中断上下文（系统进程 ksoftirqd）。在这些组件中，将包处理完后会放到 socket 的接收队列中。然后再根据 socket 内核对象找到其等待队列中正在因为等待而被阻塞掉的进程，然后把它唤醒。


recvfrom
1.1 系统调用
struct socket
*sk
ops
inet_recvmsg
1.2inet_stream_ops
struct sock
sk_prot
接收队列
sk_wq
tcp_recvmsg
1.3 tcp_prot
进程A
内核态
1.4 接收队列
里数据为空
1.5 修改当前进程状态
TASK_RUNNING
TASK_INTERRUPTIPLE
接收队列
等待队列
进程B
2.2 保存数据到socket的接收队列
2.3 唤醒等待队列上的进程
tcp_queue_rcv
sock_def_readable
tcp_rcv_established
sk_data_ready
软中断
tcp_v4_do_rcv
tcp_v4_rcv
2.1 数据包抵达网卡

每次一个进程专门为了等一个 socket 上的数据就得被从 CPU 上拿下来。然后再换上另一个进程。等到数据 ready 了，睡眠的进程又会被唤醒。总共两次进程上下文切换开销，根据之前的测试来看，每一次切换大约是 3-5 us(微秒)左右。 如果是网络 IO 密集型的应用的话，CPU 就不停地做进程切换这种无用功。

在服务端角色上，这种模式完全没办法使用。因为这种简单模型里的 socket 和进程是一对一的。我们现在要在单台机器上承载成千上万，甚至十几、上百万的用户连接请求。如果用上面的方式，那就得为每个用户请求都创建一个进程。 相信你在无论多原始的服务器网络编程里，都没见过有人这么干吧。

如果让我给它起一个名字的话，它就叫**单路不复用**（飞哥自创名词）。那么有没有更高效的网络 IO 模型呢？ 当然有，那就是你所熟知的 select、poll 和 epoll了。 下次飞哥再开始拆解 epoll 的实现源码，敬请期待！

这种模式在客户端角色上，现在还存在使用的情形。因为你的进程可能确实得等 Mysql 的数据返回成功之后，才能渲染页面返回给用户，否则啥也干不了。

> 注意一下，我说的是角色，不是具体的机器。 例如对于你的 php/java/golang 接口机，你接收用户请求的时候，你是服务端角色。但当你再请求 redis 的时候，就变为客户端角色了。

不过现在有一些封装的很好的网络框架例如 Sogou Workflow，Golang 的 net 包等在网络客户端角色上也早已摒弃了这种低效的模式！

2.3 IO 多路复用 EPOLL 内部实现

进程在 Linux 上是一个开销不小的家伙，先不说创建，光是上下文切换一次就得几个微秒。所以为了高效地对海量用户提供服务，必须要让一个进程能同时处理很多个 tcp 连接才行。现在假设一个进程保持了 10000 条连接，那么如何发现哪条连接上有数据可读了、哪条连接可写了？

我们当然可以采用循环遍历的方式来发现 IO 事件，但这种方式太低级了。我们希望有一种更高效的机制，在很多连接中的某条上有 IO 事件发生的时候直接快速把它找出来。其实这个事情 Linux 操作系统已经替我们都做好了，它就是我们所熟知的 **IO 多路复用**机制。 这里的复用指的就是对进程的复用。

在 Linux 上多路复用方案有 select、poll、epoll。 它们三个中 epoll 的性能表现是最优秀的，能支持的并发量也最大。所以我们今天把 epoll 作为要拆解的对象，深入揭秘内核是如何实现多路的 IO 管理的。

为了方便讨论，我们举一个使用了 epoll 的简单示例（只是个例子，实践中不这么写）：

```
int main(){
    listen(lfd, ...);

    cfd1 = accept(...);
    cfd2 = accept(...);
    efd = epoll_create(...);

    epoll_ctl(efd, EPOLL_CTL_ADD, cfd1, ...);
    epoll_ctl(efd, EPOLL_CTL_ADD, cfd2, ...);
    epoll_wait(efd, ...)
}
```

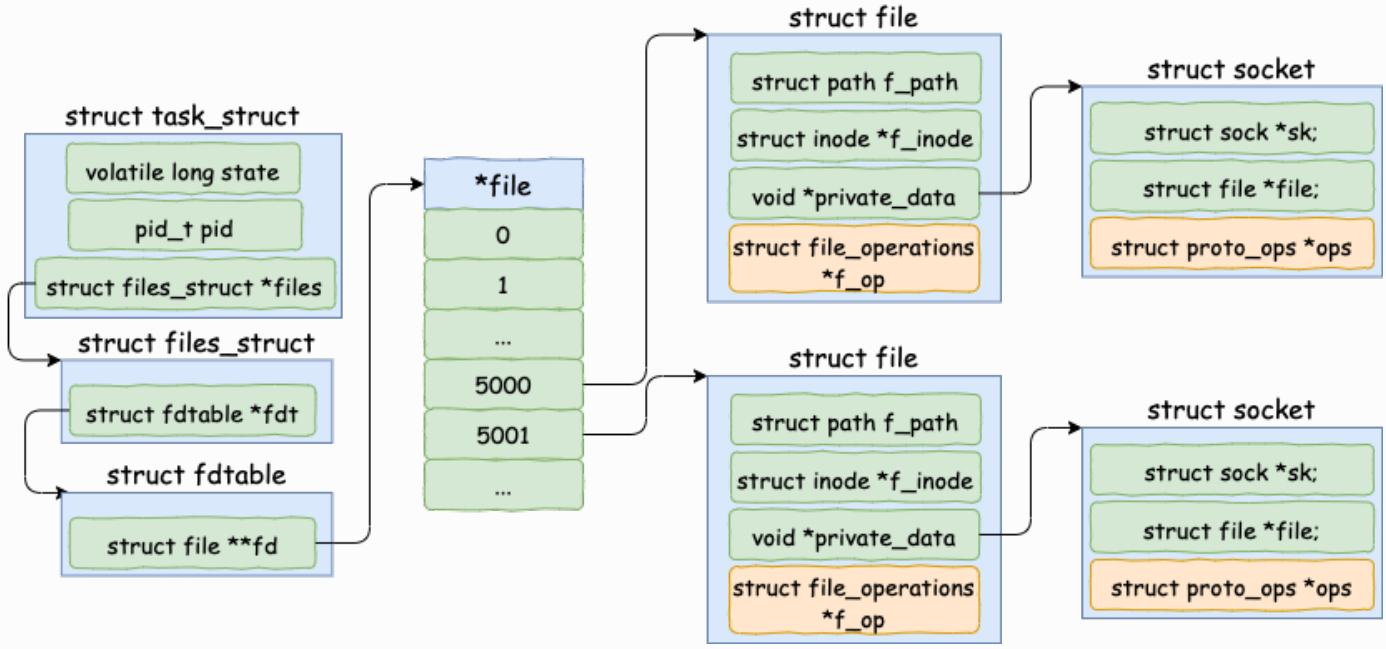
其中和 epoll 相关的函数是如下三个：

- epoll_create：创建一个 epoll 对象
- epoll_ctl：向 epoll 对象中添加要管理的连接
- epoll_wait：等待其管理的连接上的 IO 事件

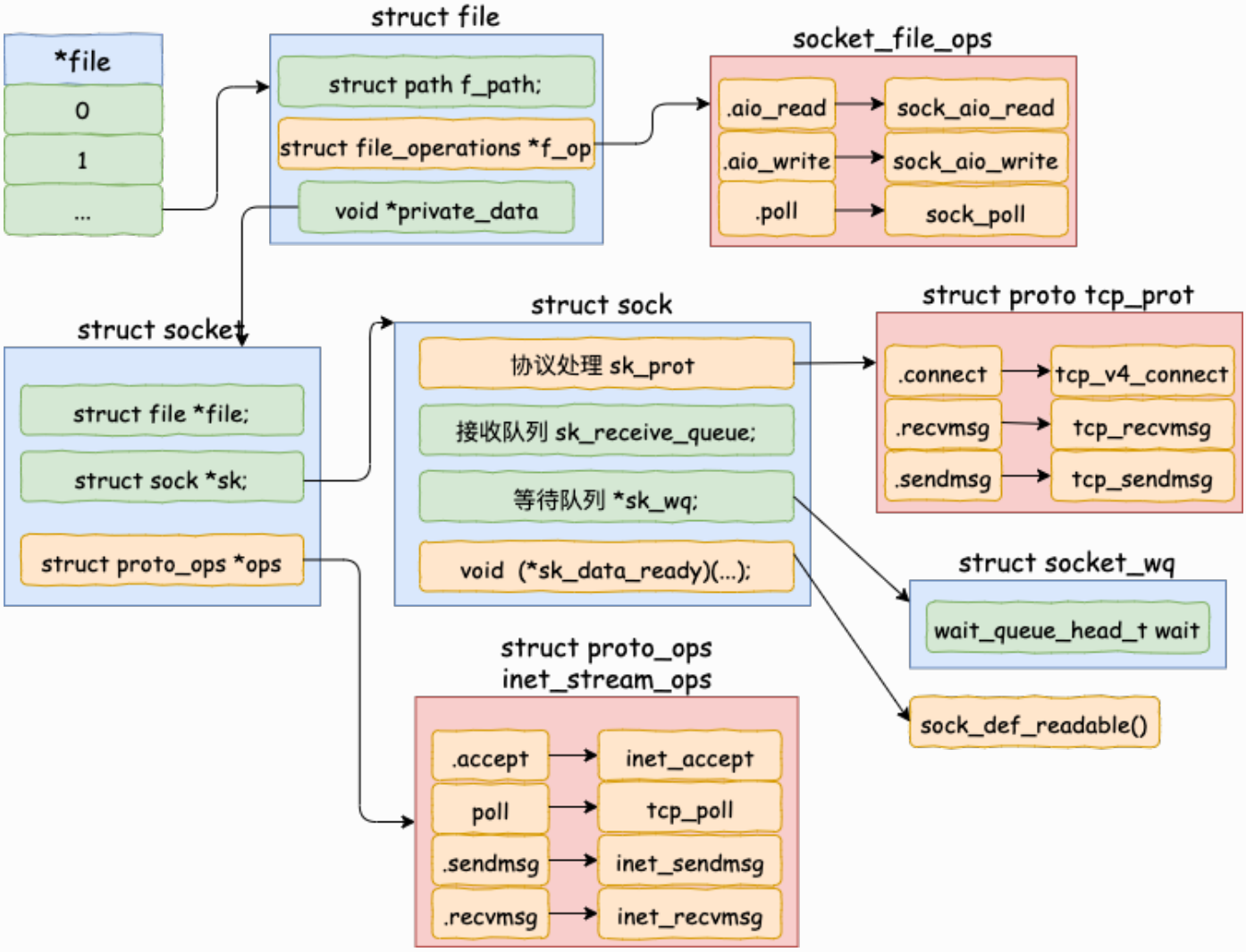
借助这个 demo，我们来展开对 epoll 原理的深度拆解。相信等你理解了这篇文章以后，你对 epoll 的驾驭能力将变得炉火纯青！！

2.3.1 accept 创建新 socket

我们直接从服务器端的 accept 讲起。当 accept 之后，进程会创建一个新的 socket 出来，专门用于和对应的客户端通信，然后把它放到当前进程的打开文件列表中。



其中一条连接的 socket 内核对象更为具体一点的结构图如下。



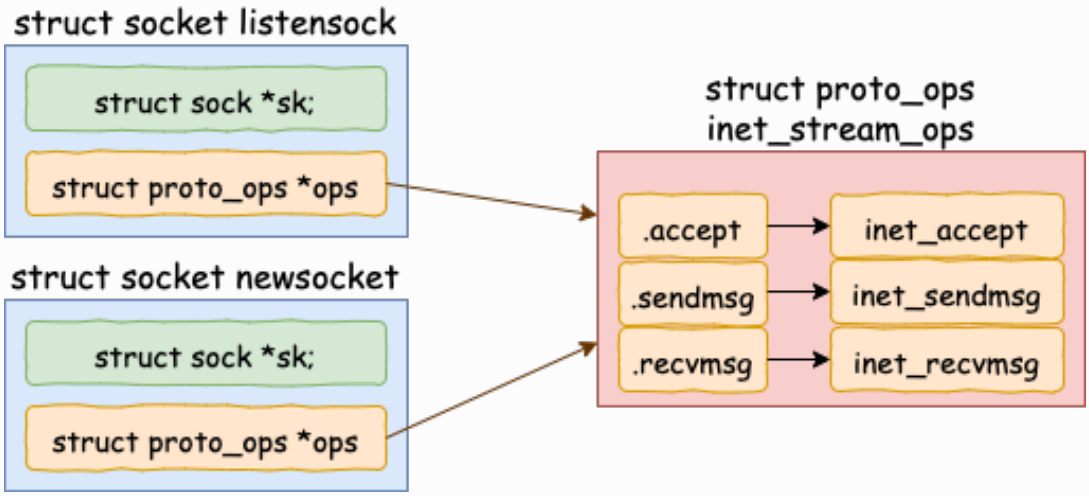
接下来我们来看一下接收连接时 socket 内核对象的创建源码。accept 的系统调用代码位于源文件 net/socket.c 下。

```
//file: net/socket.c
SYSCALL_DEFINE4(accept4, int, fd, struct sockaddr __user *,
upeer_sockaddr,
        int __user *, upeer_addrlen, int, flags)
{
    struct socket *sock, *newsock;

    //根据 fd 查找到监听的 socket
    sock = sockfd_lookup_light(fd, &err, &fput_needed);

    //1.1 申请并初始化新的 socket
    newsock = sock_alloc();
    newsock->type = sock->type;
    newsock->ops = sock->ops;
```

```
    //1.2 申请新的 file 对象，并设置到新 socket 上
    newfile = sock_alloc_file(newsock, flags, sock->sk->sk_prot_creator->name);
    ......

    //1.3 接收连接
    err = sock->ops->accept(sock, newsock, sock->file->f_flags);

    //1.4 添加新文件到当前进程的打开文件列表
    fd_install(newfd, newfile);
```

初始化 struct socket 对象

在上述的源码中，首先是调用 sock_alloc 申请一个 struct socket 对象出来。然后接着把 listen 状态的 socket 对象上的协议操作函数集合 ops 赋值给新的 socket。（对于所有的 AF_INET 协议族下的 socket 来说，它们的 ops 方法都是一样的，所以这里可以直接复制过来）

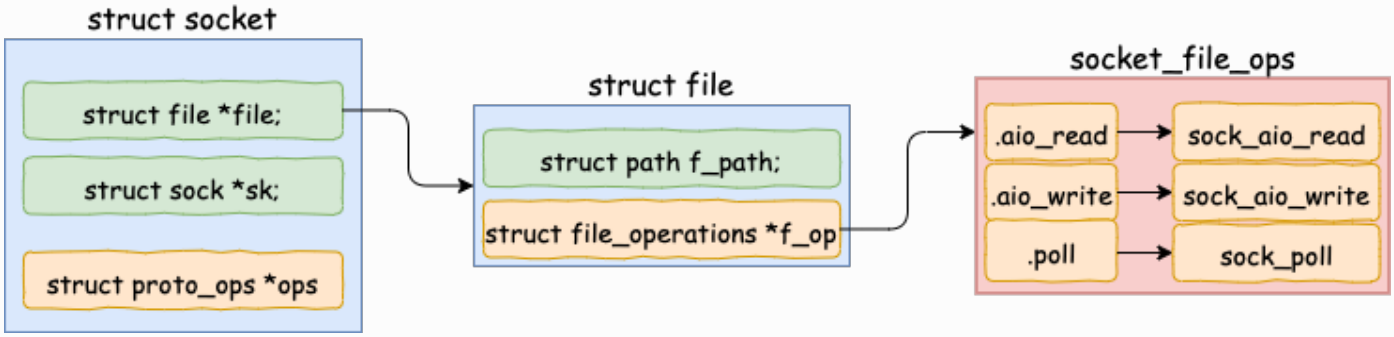


其中 inet_stream_ops 的定义如下

```
//file: net/ipv4/af_inet.c
const struct proto_ops inet_stream_ops = {
    ...
    .accept         = inet_accept,
    .listen         = inet_listen,
    .sendmsg        = inet_sendmsg,
    .recvmsg        = inet_recvmsg,
    ...
}
```

为新 socket 对象申请 file

struct socket 对象中有一个重要的成员 -- file 内核对象指针。这个指针初始化的时候是空的。 在 accept 方法里会调用 sock_alloc_file 来申请内存并初始化。 然后将新 file 对象设置到 sock->file 上。



来看 sock_alloc_file 的实现过程：

```
struct file *sock_alloc_file(struct socket *sock, int flags,
    const char *dname)
{
    struct file *file;
    file = alloc_file(&path, FMODE_READ | FMODE_WRITE,
            &socket_file_ops);
    ......
    sock->file = file;
}
```

sock_alloc_file 又会接着调用到 alloc_file。注意在 alloc_file 方法中，把 socket_file_ops 函数集合一并赋到了新 file->f_op 里了。

```
//file: fs/file_table.c
struct file *alloc_file(struct path *path, fmode_t mode,
        const struct file_operations *fop)
{
    struct file *file;
    file->f_op = fop;
    ......
}
```

socket_file_ops 的具体定义如下：

```
//file: net/socket.c
static const struct file_operations socket_file_ops = {
    ...
    .aio_read   = sock_aio_read,
    .aio_write  = sock_aio_write,
    .poll       = sock_poll,
    .release    = sock_close,
    ...
};
```

这里看到，在accept里创建的新 socket 里的 file->f_op->poll 函数指向的是 sock_poll。接下来我们会调用到它，后面我们再说。

> 其实 file 对象内部也有一个 socket 指针，指向 socket 对象。

接收连接

在 socket 内核对象中除了 file 对象指针以外，有一个核心成员 sock。

```
//file: include/linux/net.h
struct socket {
    struct file     *file;
    struct sock     *sk;
}
```

这个 struct sock 数据结构非常大，是 socket 的核心内核对象。发送队列、接收队列、等待队列等核心数据结构都位于此。其定义位置文件 include/net/sock.h，由于太长就不展示了。

在 accept 的源码中：

```
//file: net/socket.c
SYSCALL_DEFINE4(accept4, ...)
    ...
    //1.3 接收连接
    err = sock->ops->accept(sock, newsock, sock->file->f_flags);
}
```

`sock->ops->accept` 对应的方法是 inet_accept。它执行的时候会从握手队列里直接获取创建好的 sock。 sock 对象的完整创建过程涉及到三次握手，比较复杂，不展开了说了。咱们只看 struct sock 初始化过程中用到的一个函数：

```
void sock_init_data(struct socket *sock, struct sock *sk)
{
    sk->sk_wq    =    NULL;
    sk->sk_data_ready    =    sock_def_readable;
}
```

在这里把 sock 对象的 sk_data_ready 函数指针设置为 sock_def_readable。这个这里先记住就行了，后面会用到。

添加新文件到当前进程的打开文件列表

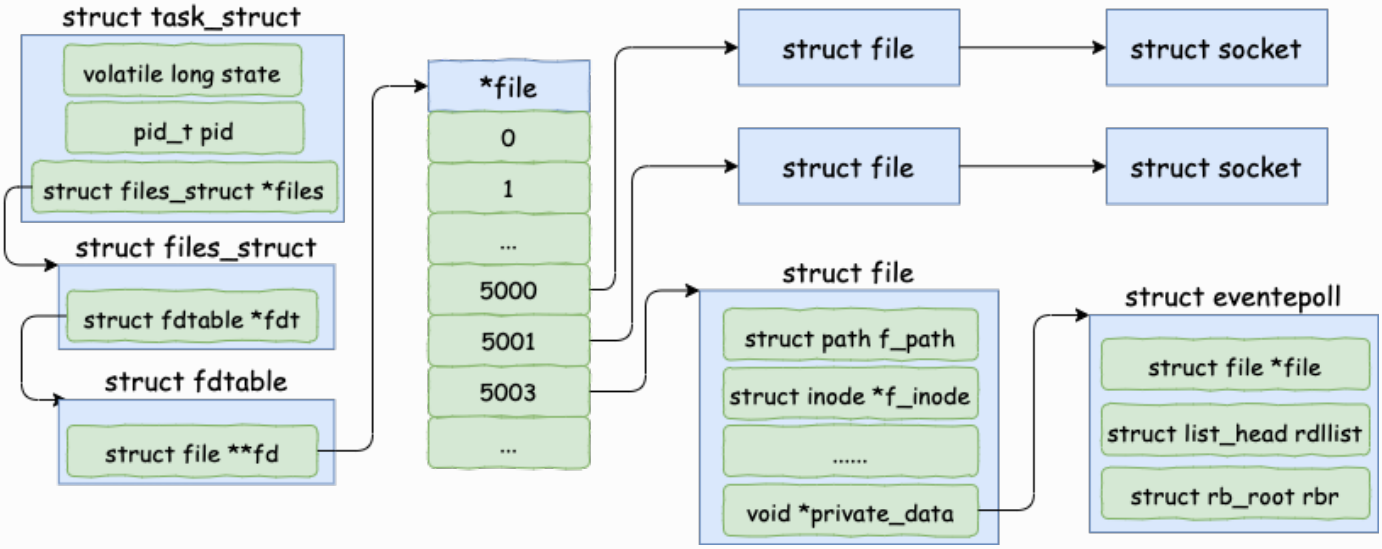
当 file、socket、sock 等关键内核对象创建完毕以后，剩下要做的一件事情就是把它挂到当前进程的打开文件列表中就行了。

```
//file: fs/file.c
void fd_install(unsigned int fd, struct file *file)
{
    __fd_install(current->files, fd, file);
}
```

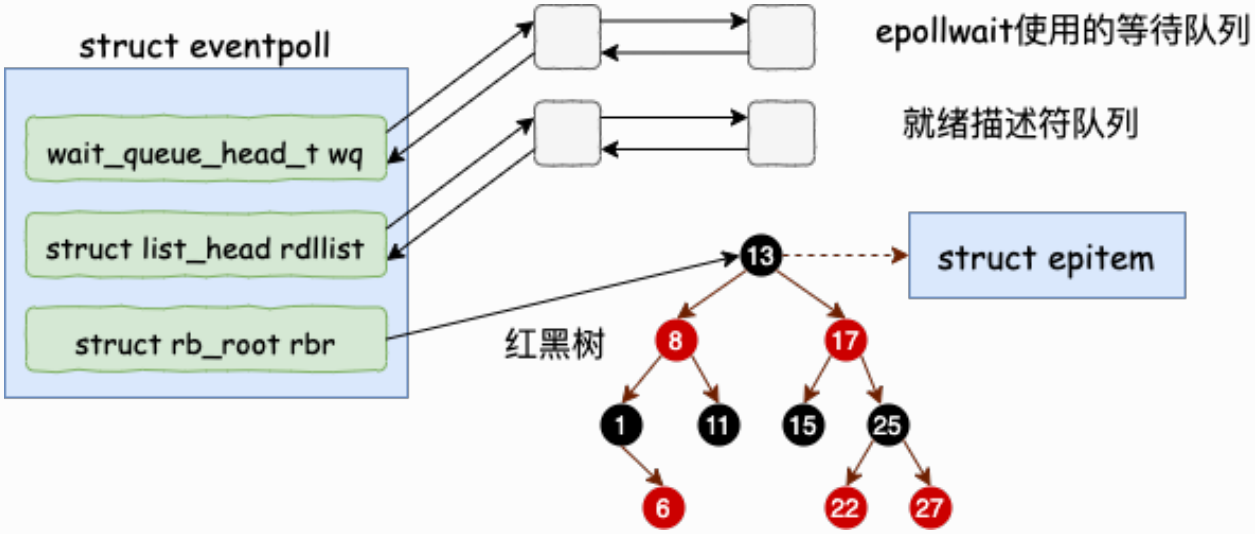
```
void __fd_install(struct files_struct *files, unsigned int fd,
        struct file *file)
{
    ...
    fdt = files_fdtable(files);
    BUG_ON(fdt->fd[fd] != NULL);
    rcu_assign_pointer(fdt->fd[fd], file);
}
```

2.3.2 epoll_create 实现

在用户进程调用 epoll_create 时，内核会创建一个 struct eventpoll 的内核对象。并同样把它关联到当前进程的已打开文件列表中。



对于 struct eventpoll 对象，更详细的结构如下（同样只列出和今天主题相关的成员）。



epoll_create 的源代码相对比较简单。在 fs/eventpoll.c 下

```
// file: fs/eventpoll.c
SYSCALL_DEFINE1(epoll_create1, int, flags)
{
    struct eventpoll *ep = NULL;

    //创建一个 eventpoll 对象
    error = ep_alloc(&ep);
}
```

struct eventpoll 的定义也在这个源文件中。

```
// file: fs/eventpoll.c
struct eventpoll {

    //sys_epoll_wait用到的等待队列
    wait_queue_head_t wq;

    //接收就绪的描述符都会放到这里
    struct list_head rdllist;

    //每个epoll对象中都有一颗红黑树
    struct rb_root rbr;

    ......
}
```

eventpoll 这个结构体中的几个成员的含义如下：

- **wq：** 等待队列链表。软中断数据就绪的时候会通过 wq 来找到阻塞在 epoll 对象上的用户进程。
- **rbr：** 一棵红黑树。为了支持对海量连接的高效查找、插入和删除，eventpoll 内部使用了一棵红黑树。通过这棵树来管理用户进程下添加进来的所有 socket 连接。
- **rdllist：** 就绪的描述符的链表。当有的连接就绪的时候，内核会把就绪的连接放到 rdllist 链表里。这样应用进程只需要判断链表就能找出就绪进程，而不用去遍历整棵树。

当然这个结构被申请完之后，需要做一点点的初始化工作，这都在 ep_alloc 中完成。

```
//file: fs/eventpoll.c
static int ep_alloc(struct eventpoll **pep)
{
    struct eventpoll *ep;

    //申请 epollevent 内存
    ep = kzalloc(sizeof(*ep), GFP_KERNEL);

    //初始化等待队列头
    init_waitqueue_head(&ep->wq);

    //初始化就绪列表
    INIT_LIST_HEAD(&ep->rdllist);

    //初始化红黑树指针
    ep->rbr = RB_ROOT;

    ......
}
```

说到这儿，这些成员其实只是刚被定义或初始化了，还都没有被使用。它们会在下面被用到。

2.3.3 epoll_ctl 添加 socket

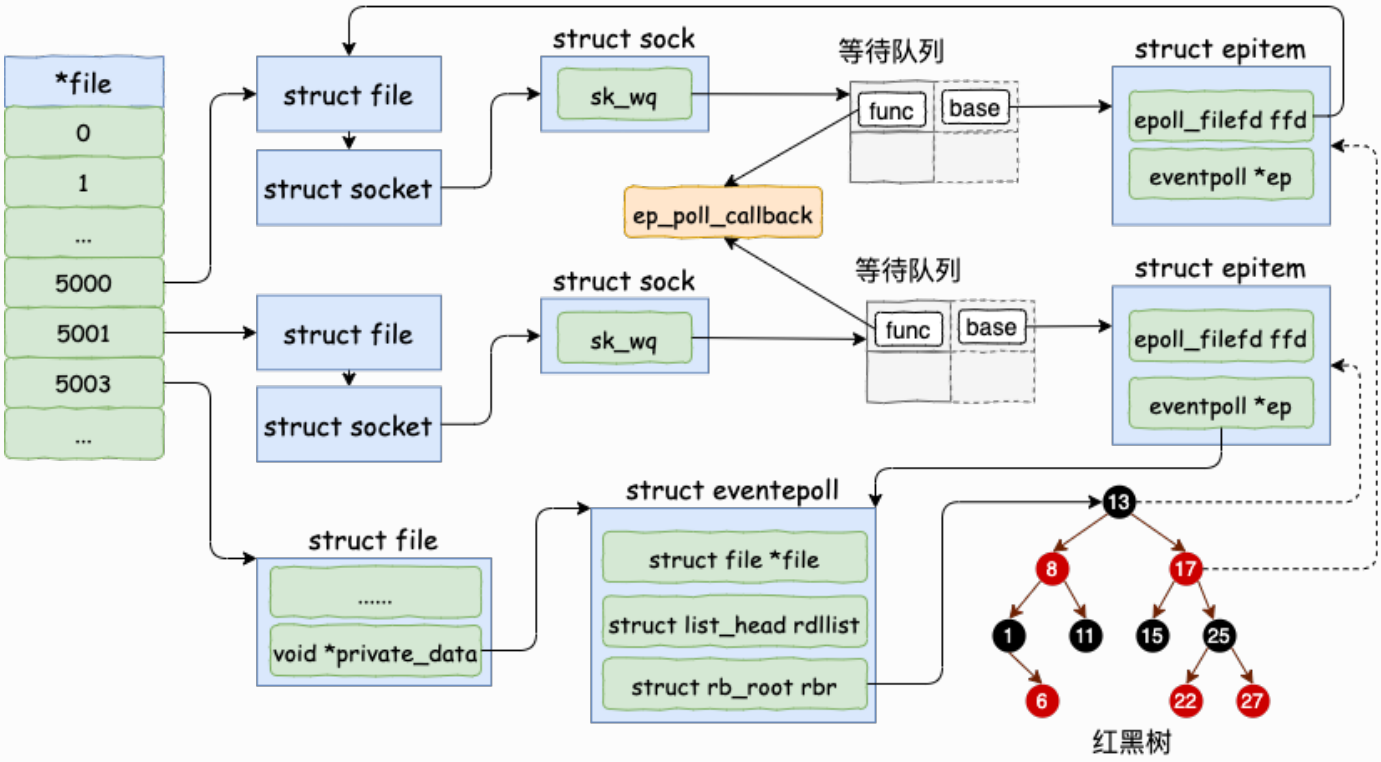
理解这一步是理解整个 epoll 的关键。

为了简单，我们只考虑使用 EPOLL_CTL_ADD 添加 socket，先忽略删除和更新。

假设我们现在和客户端们的多个连接的 socket 都创建好了，也创建好了 epoll 内核对象。在使用 epoll_ctl 注册每一个 socket 的时候，内核会做如下三件事情

- 1.分配一个红黑树节点对象 epitem，
- 2.添加等待事件到 socket 的等待队列中，其回调函数是 ep_poll_callback
- 3.将 epitem 插入到 epoll 对象的红黑树里

通过 epoll_ctl 添加两个 socket 以后，这些内核数据结构最终在进程中的关系图大致如下：



我们来详细看看 socket 是如何添加到 epoll 对象里的，找到 epoll_ctl 的源码。

```
// file: fs/eventpoll.c
SYSCALL_DEFINE4(epoll_ctl, int, epfd, int, op, int, fd,
        struct epoll_event __user *, event)
{
    struct eventpoll *ep;
    struct file *file, *tfile;
```

```
    //根据 epfd 找到 eventpoll 内核对象
    file = fget(epfd);
    ep = file->private_data;

    //根据 socket 句柄号， 找到其 file 内核对象
    tfile = fget(fd);

    switch (op) {
    case EPOLL_CTL_ADD:
        if (!epi) {
            epds.events |= POLLERR | POLLHUP;
            error = ep_insert(ep, &epds, tfile, fd);
        } else
            error = -EEXIST;
        clear_tfile_check_list();
        break;
}
```

在 epoll_ctl 中首先根据传入 fd 找到 eventpoll、socket相关的内核对象 。对于 EPOLL_CTL_ADD 操作来说，会然后执行到 ep_insert 函数。所有的注册都是在这个函数中完成的。

```
//file: fs/eventpoll.c
static int ep_insert(struct eventpoll *ep,
                struct epoll_event *event,
                struct file *tfile, int fd)
{
    //3.1 分配并初始化 epitem
    //分配一个epi对象
    struct epitem *epi;
    if (!(epi = kmem_cache_alloc(epi_cache, GFP_KERNEL)))
        return -ENOMEM;

    //对分配的epi进行初始化
    //epi->ffd中存了句柄号和struct file对象地址
    INIT_LIST_HEAD(&epi->pwqlist);
    epi->ep = ep;
    ep_set_ffd(&epi->ffd, tfile, fd);

    //3.2 设置 socket 等待队列
    //定义并初始化 ep_pqueue 对象
```

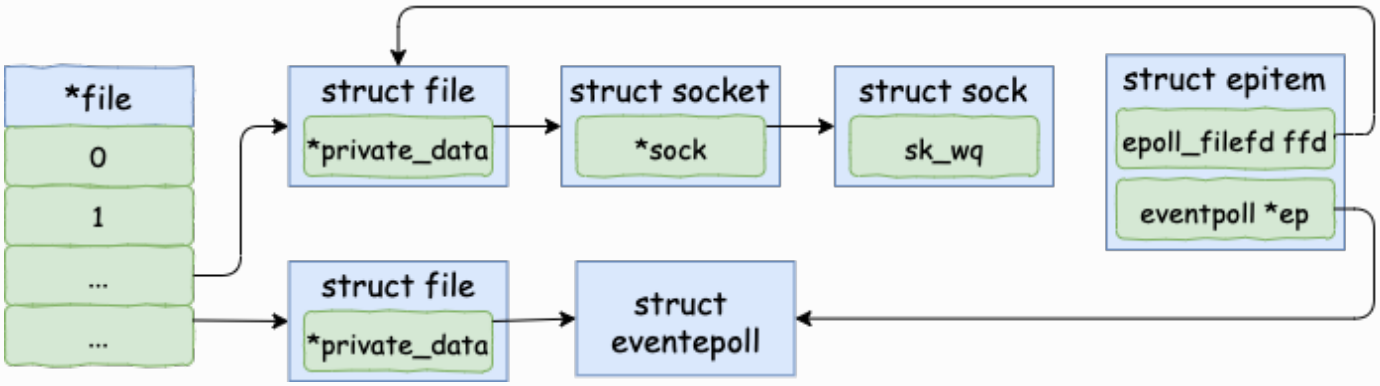
```
    struct ep_pqueue epq;
    epq.epi = epi;
    init_poll_funcptr(&epq.pt, ep_ptable_queue_proc);

    //调用 ep_ptable_queue_proc 注册回调函数
    //实际注入的函数为 ep_poll_callback
    revents = ep_item_poll(epi, &epq.pt);

    ......
    //3.3 将epi插入到 eventpoll 对象中的红黑树中
    ep_rbtree_insert(ep, epi);
    ......
}
```

分配并初始化 epitem

对于每一个 socket，调用 epoll_ctl 的时候，都会为之分配一个 epitem。 该结构的主要数据如下：

```
//file: fs/eventpoll.c
struct epitem {

    //红黑树节点
    struct rb_node rbn;

    //socket文件描述符信息
    struct epoll_filefd ffd;

    //所归属的 eventpoll 对象
    struct eventpoll *ep;

    //等待队列
    struct list_head pwqlist;
}
```

对 epitem 进行了一些初始化，首先在 `epi->ep = ep` 这行代码中将其 ep 指针指向 eventpoll 对象。另外用要添加的 socket 的 file、fd 来填充 epitem->ffd。

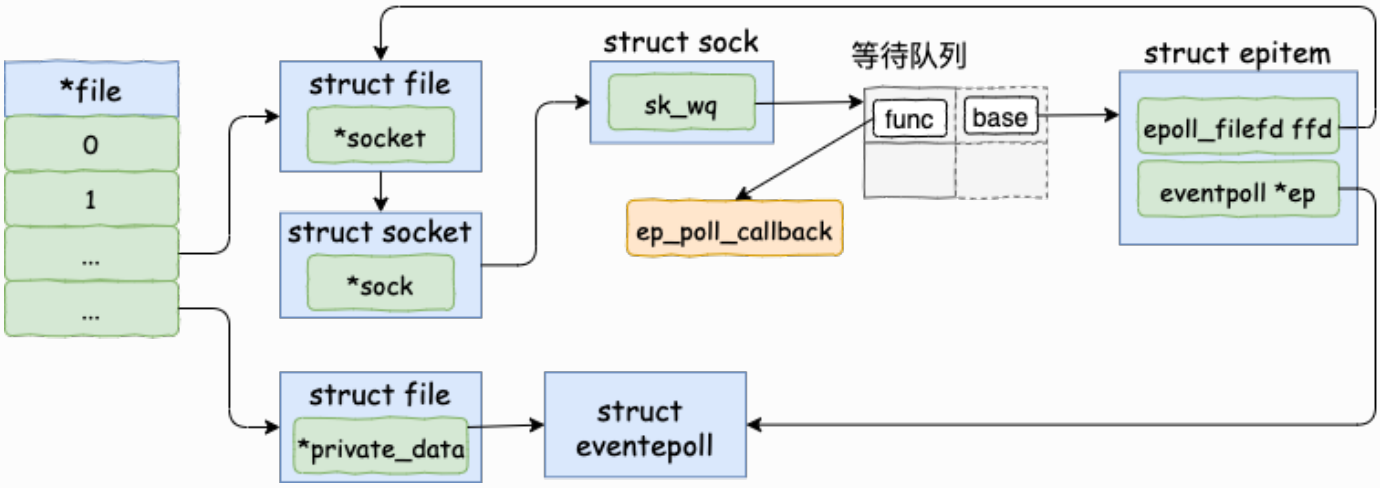


其中使用到的 ep_set_ffd 函数如下。

```
static inline void ep_set_ffd(struct epoll_filefd *ffd,
                              struct file *file, int fd)
{
    ffd->file = file;
    ffd->fd = fd;
}
```

设置 socket 等待队列

在创建 epitem 并初始化之后，ep_insert 中第二件事情就是设置 socket 对象上的等待任务队列。并把函数 fs/eventpoll.c 文件下的 ep_poll_callback 设置为数据就绪时候的回调函数。



这一块的源代码稍微有点绕，没有耐心的话直接跳到下面的加粗字体来看。首先来看 ep_item_poll。

```
static inline unsigned int ep_item_poll(struct epitem *epi,
poll_table *pt)
{
    pt->_key = epi->event.events;

    return epi->ffd.file->f_op->poll(epi->ffd.file, pt) & epi-
>event.events;
}
```

看，这里调用到了 socket 下的 file->f_op->poll。通过上面第一节的 socket 的结构图，我们知道这个函数实际上是 sock_poll。

```
/* No kernel lock held - perfect */
static unsigned int sock_poll(struct file *file, poll_table
*wait)
{
    ...
    return sock->ops->poll(file, sock, wait);
}
```

同样回看第一节里的 socket 的结构图，sock->ops->poll 其实指向的是 tcp_poll。

```
//file: net/ipv4/tcp.c
unsigned int tcp_poll(struct file *file, struct socket *sock,
poll_table *wait)
{
    struct sock *sk = sock->sk;

    sock_poll_wait(file, sk_sleep(sk), wait);
}
```

在 sock_poll_wait 的第二个参数传参前，先调用了 sk_sleep 函数。**在这个函数里它获取了 sock 对象下的等待队列列表头 wait_queue_head_t，待会等待队列项就插入这里**。这里稍微注意下，是 socket 的等待队列，不是 epoll 对象的。来看 sk_sleep 源码：

```
//file: include/net/sock.h
static inline wait_queue_head_t *sk_sleep(struct sock *sk)
{
    BUILD_BUG_ON(offsetof(struct socket_wq, wait) != 0);
    return &rcu_dereference_raw(sk->sk_wq)->wait;
}
```

接着真正进入 sock_poll_wait。

```
static inline void sock_poll_wait(struct file *filp,
        wait_queue_head_t *wait_address, poll_table *p)
{
    poll_wait(filp, wait_address, p);
}
```

```
static inline void poll_wait(struct file * filp,
wait_queue_head_t * wait_address, poll_table *p)
{
    if (p && p->_qproc && wait_address)
        p->_qproc(filp, wait_address, p);
}
```

这里的 qproc 是个函数指针，它在前面的 init_poll_funcptr 调用时被设置成了 ep_ptable_queue_proc 函数。

```
static int ep_insert(...)
{
    ...
    init_poll_funcptr(&epq.pt, ep_ptable_queue_proc);
    ...
}
```

```
//file: include/linux/poll.h
static inline void init_poll_funcptr(poll_table *pt,
    poll_queue_proc qproc)
{
    pt->_qproc = qproc;
    pt->_key   = ~0UL; /* all events enabled */
}
```

敲黑板！！！注意，废了半天的劲，终于到了重点了！ 在 ep_ptable_queue_proc 函数中，新建了一个等待队列项，并注册其回调函数为 ep_poll_callback 函数。然后再将这个等待项添加到 socket 的等待队列中。

```
//file: fs/eventpoll.c
static void ep_ptable_queue_proc(struct file *file,
wait_queue_head_t *whead,
                 poll_table *pt)
{
    struct eppoll_entry *pwq;
    f (epi->nwait >= 0 && (pwq = kmem_cache_alloc(pwq_cache,
GFP_KERNEL))) {
                //初始化回调方法
                init_waitqueue_func_entry(&pwq->wait,
ep_poll_callback);

                //将ep_poll_callback放入socket的等待队列whead（注意
不是epoll的等待队列）
                add_wait_queue(whead, &pwq->wait);

        }
```

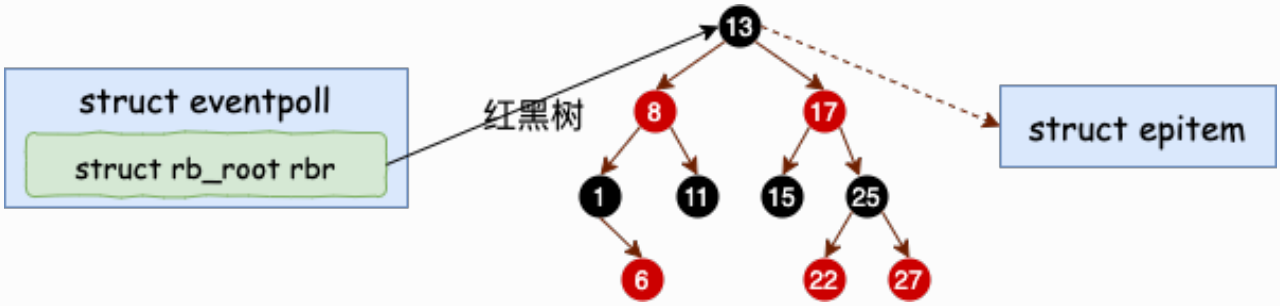
在前文 深入理解高性能网络开发路上的绊脚石 - 同步阻塞网络 IO 里阻塞式的系统调用 recvfrom 里，由于需要在数据就绪的时候唤醒用户进程，所以等待对象项的 private (这个变量名起的也是醉了) 会设置成当前用户进程描述符 current。 而我们今天的 socket 是交给 epoll 来管理的，不需要在一个 socket 就绪的时候就唤醒进程，所以这里的 `q->private` 没有啥卵用就设置成了 NULL。

```
//file:include/linux/wait.h
static inline void init_waitqueue_func_entry(
    wait_queue_t *q, wait_queue_func_t func)
{
    q->flags = 0;
    q->private = NULL;

    //ep_poll_callback 注册到 wait_queue_t对象上
    //有数据到达的时候调用 q->func
    q->func = func;
}
```

如上，等待队列项中仅仅只设置了回调函数 q->func 为 ep_poll_callback。在后面的第 5 节数据来啦中我们将看到，软中断将数据收到 socket 的接收队列后，会通过注册的这个 ep_poll_callback 函数来回调，进而通知到 epoll 对象。

插入红黑树

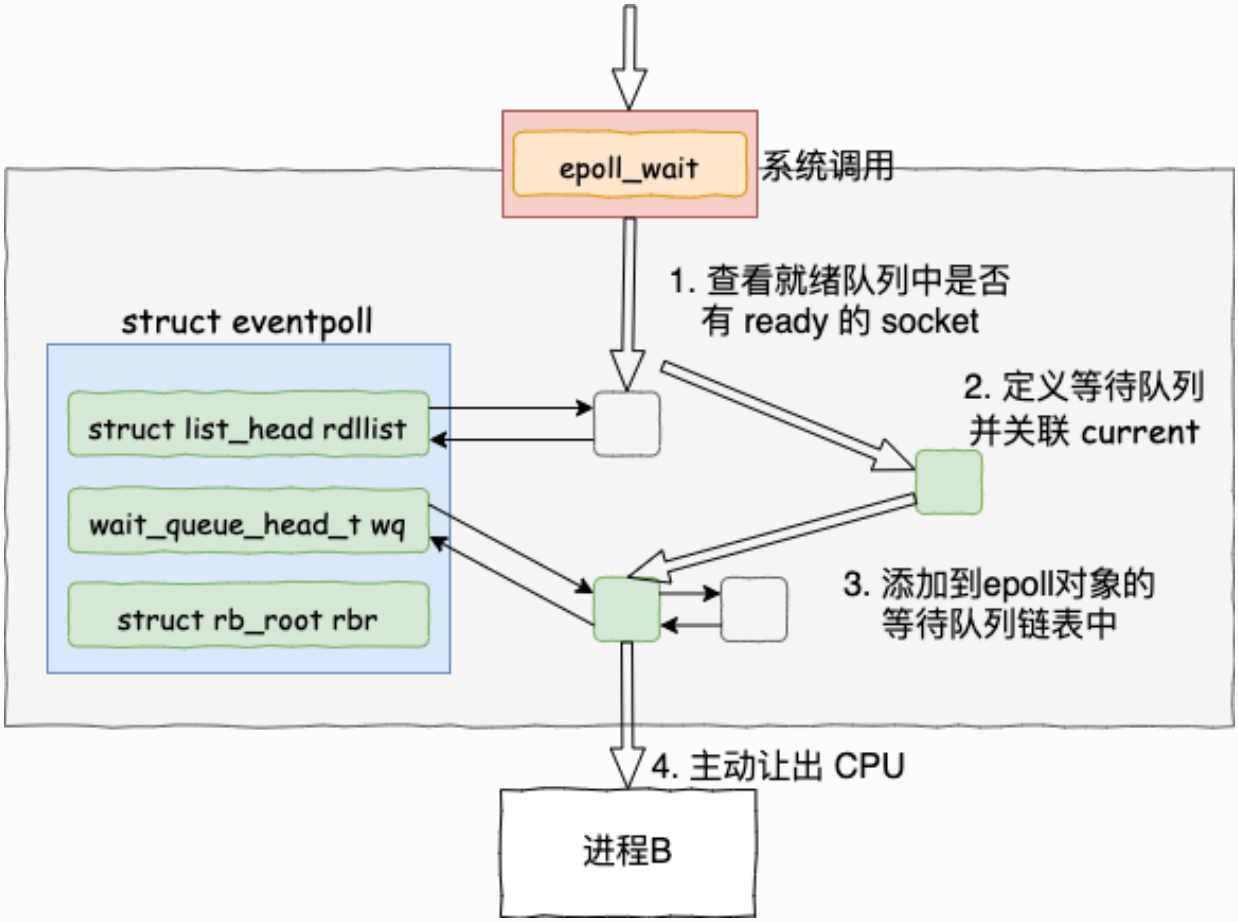
分配完 epitem 对象后，紧接着并把它插入到红黑树中。一个插入了一些 socket 描述符的 epoll 里的红黑树的示意图如下：



这里我们再聊聊为啥要用红黑树，很多人说是因为效率高。其实我觉得这个解释不够全面，要说查找效率树哪能比的上 HASHTABLE。我个人认为觉得更为合理的一个解释是为了让 epoll 在查找效率、插入效率、内存开销等等多个方面比较均衡，最后发现最适合这个需求的数据结构是红黑树。

2.3.4 epoll_wait 等待接收

epoll_wait 做的事情不复杂，当它被调用时它观察 eventpoll->rdllist 链表里有没有数据即可。有数据就返回，没有数据就创建一个等待队列项，将其添加到 eventpoll 的等待队列上，然后把自己阻塞掉就完事。



> 注意：epoll_ctl 添加 socket 时也创建了等待队列项。不同的是这里的等待队列项是挂在 epoll 对象上的，而前者是挂在 socket 对象上的。

其源代码如下：

```
//file: fs/eventpoll.c
SYSCALL_DEFINE4(epoll_wait, int, epfd, struct epoll_event
__user *, events,
        int, maxevents, int, timeout)
{
    ...
    error = ep_poll(ep, events, maxevents, timeout);
}

static int ep_poll(struct eventpoll *ep, struct epoll_event
__user *events,
             int maxevents, long timeout)
{
    wait_queue_t wait;
    ......
```

```
fetch_events:
    //4.1 判断就绪队列上有没有事件就绪
    if (!ep_events_available(ep)) {

        //4.2 定义等待事件并关联当前进程
        init_waitqueue_entry(&wait, current);

        //4.3 把新 waitqueue 添加到 epoll->wq 链表里
        __add_wait_queue_exclusive(&ep->wq, &wait);

        for (;;) {
            ...
            //4.4 让出CPU 主动进入睡眠状态
            if (!schedule_hrtimeout_range(to, slack,
HRTIMER_MODE_ABS))
                timed_out = 1;
            ...
}
```

判断就绪队列上有没有事件就绪

首先调用 ep_events_available 来判断就绪链表中是否有可处理的事件。

```
//file: fs/eventpoll.c
static inline int ep_events_available(struct eventpoll *ep)
{
    return !list_empty(&ep->rdllist) || ep->ovflist !=
EP_UNACTIVE_PTR;
}
```

定义等待事件并关联当前进程

假设确实没有就绪的连接，那接着会进入 init_waitqueue_entry 中定义等待任务，并把 current （当前进程）添加到 waitqueue 上。

> 是的，当没有 IO 事件的时候， epoll 也是会阻塞掉当前进程。这个是合理的，因为没有事情可做了占着 CPU 也没啥意义。网上的很多文章有个很不好的习惯，讨论阻塞、非阻塞等概念的时候都不说主语。这会导致你看的云里雾里。拿 epoll 来说，epoll 本身是阻塞的，但一般会把 socket 设置成非阻塞。只有说了主语，这些概念才有意义。

```
//file: include/linux/wait.h
static inline void init_waitqueue_entry(wait_queue_t *q, struct
task_struct *p)
{
    q->flags = 0;
    q->private = p;
    q->func = default_wake_function;
}
```

注意这里的回调函数名称是 default_wake_function。 后续在第 5 节数据来啦时将会调用到该函数。

添加到等待队列

```
static inline void __add_wait_queue_exclusive(wait_queue_head_t
*q,
                                wait_queue_t *wait)
{
    wait->flags |= WQ_FLAG_EXCLUSIVE;
    __add_wait_queue(q, wait);
}
```

在这里，把上一小节定义的等待事件添加到了 epoll 对象的等待队列中。

让出CPU 主动进入睡眠状态

通过 set_current_state 把当前进程设置为可打断。 调用 schedule_hrtimeout_range 让出 CPU，主动进入睡眠状态

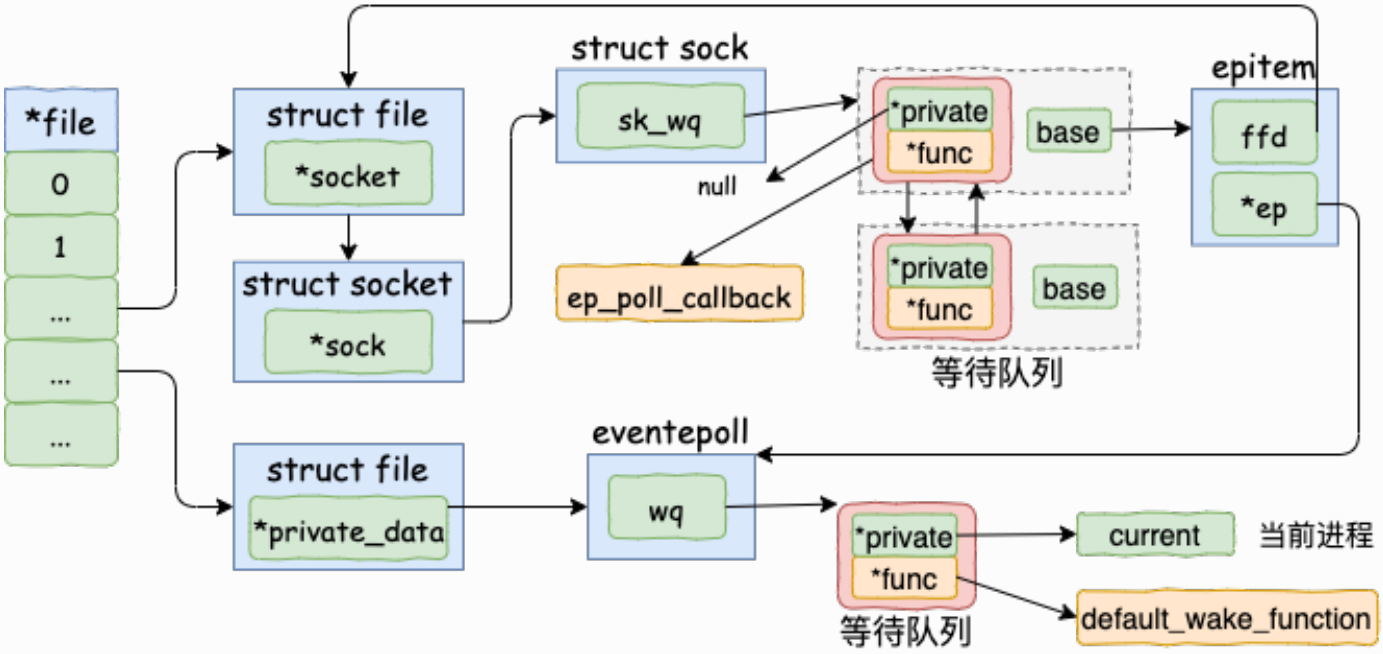
```
//file: kernel/hrtimer.c
int __sched schedule_hrtimeout_range(ktime_t *expires,
    unsigned long delta, const enum hrtimer_mode mode)
{
    return schedule_hrtimeout_range_clock(
            expires, delta, mode, CLOCK_MONOTONIC);
}

int __sched schedule_hrtimeout_range_clock(...)
{
    schedule();
    ...
}
```

在 schedule 中选择下一个进程调度

```
//file: kernel/sched/core.c
static void __sched __schedule(void)
{
    next = pick_next_task(rq);
    ...
    context_switch(rq, prev, next);
}
```

2.3.5 数据来啦

在前面 epoll_ctl 执行的时候，内核为每一个 socket 上都添加了一个等待队列项。 在 epoll_wait 运行完的时候，又在 event poll 对象上添加了等待队列元素。在讨论数据开始接收之前，我们把这些队列项的内容再稍微总结一下。



- socket->sock->sk_data_ready 设置的就绪处理函数是 sock_def_readable
- 在 socket 的等待队列项中，其回调函数是 ep_poll_callback。另外其 private 没有用了，指向的是空指针 null。
- 在 eventpoll 的等待队列项中，回调函数是 default_wake_function。其 private 指向的是等待该事件的用户进程。

在这一小节里，我们将看到软中断是怎么样在数据处理完之后依次进入各个回调函数，最后通知到用户进程的。

接收数据到任务队列

关于软中断是怎么处理网络帧，为了避免篇幅过于臃肿，这里不再介绍。感兴趣的可以看文章 《图解Linux网络包接收过程》。 我们今天直接从 tcp 协议栈的处理入口函数 tcp_v4_rcv 开始说起。

```
// file: net/ipv4/tcp_ipv4.c
int tcp_v4_rcv(struct sk_buff *skb)
{
    ......
    th = tcp_hdr(skb); //获取tcp header
    iph = ip_hdr(skb); //获取ip header

    //根据数据包 header 中的 ip、端口信息查找到对应的socket
```

```
    sk = __inet_lookup_skb(&tcp_hashinfo, skb, th->source, th-
>dest);
    ......

    //socket 未被用户锁定
    if (!sock_owned_by_user(sk)) {
        {
            if (!tcp_prequeue(sk, skb))
                ret = tcp_v4_do_rcv(sk, skb);
        }
    }
}
```

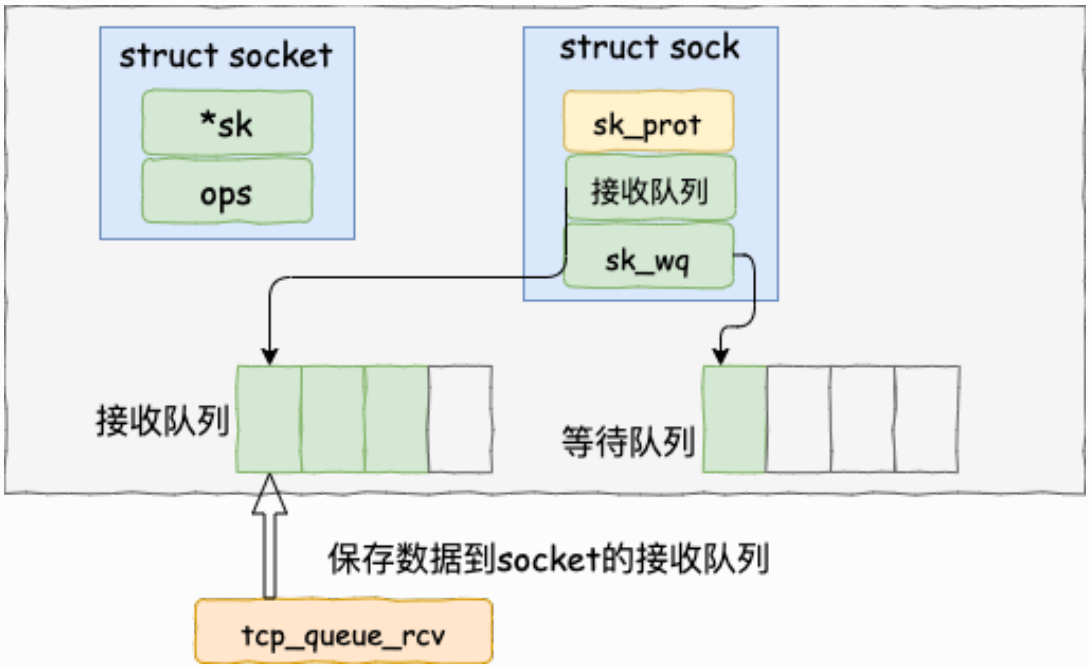
在 tcp_v4_rcv 中首先根据收到的网络包的 header 里的 source 和 dest 信息来在本机上查询对应的 socket。找到以后，我们直接进入接收的主体函数 tcp_v4_do_rcv 来看。

```
//file: net/ipv4/tcp_ipv4.c
int tcp_v4_do_rcv(struct sock *sk, struct sk_buff *skb)
{
    if (sk->sk_state == TCP_ESTABLISHED) {

        //执行连接状态下的数据处理
        if (tcp_rcv_established(sk, skb, tcp_hdr(skb), skb-
>len)) {
            rsk = sk;
            goto reset;
        }
        return 0;
    }

    //其它非 ESTABLISH 状态的数据包处理
    ......
}
```

我们假设处理的是 ESTABLISH 状态下的包，这样就又进入 tcp_rcv_established 函数中进行处理。

```
//file: net/ipv4/tcp_input.c
int tcp_rcv_established(struct sock *sk, struct sk_buff *skb,
            const struct tcphdr *th, unsigned int len)
{
    ......

    //接收数据到队列中
    eaten = tcp_queue_rcv(sk, skb, tcp_header_len,
                                    &fragstolen);

    //数据 ready，唤醒 socket 上阻塞掉的进程
    sk->sk_data_ready(sk, 0);
```

在 tcp_rcv_established 中通过调用 tcp_queue_rcv 函数中完成了将接收数据放到 socket 的接收队列上。



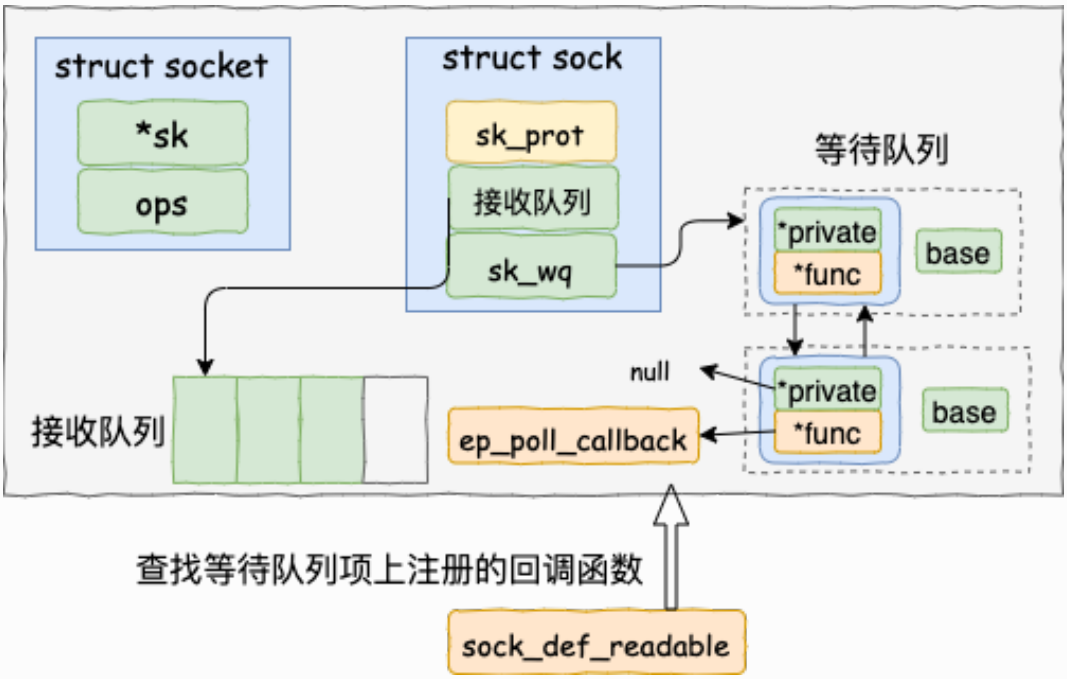
如下源码所示

```
//file: net/ipv4/tcp_input.c
static int __must_check tcp_queue_rcv(struct sock *sk, struct
sk_buff *skb, int hdrlen,
            bool *fragstolen)
{
    //把接收到的数据放到 socket 的接收队列的尾部
    if (!eaten) {
        __skb_queue_tail(&sk->sk_receive_queue, skb);
        skb_set_owner_r(skb, sk);
    }
    return eaten;
}
```

查找就绪回调函数

调用 tcp_queue_rcv 接收完成之后，接着再调用 sk_data_ready 来唤醒在 socket上等待的用户进程。 这又是一个函数指针。 回想上面第一节我们在 accept 函数创建 socket 流程里提到的 sock_init_data 函数，在这个函数里已经把 sk_data_ready 设置成 sock_def_readable 函数了。它是默认的数据就绪处理函数。

当 socket 上数据就绪时候，内核将以 sock_def_readable 这个函数为入口，找到 epoll_ctl 添加 socket 时在其上设置的回调函数 ep_poll_callback。



我们来详细看下细节：

```
//file: net/core/sock.c
static void sock_def_readable(struct sock *sk, int len)
{
    struct socket_wq *wq;

    rcu_read_lock();
    wq = rcu_dereference(sk->sk_wq);

    //这个名字起的不好，并不是有阻塞的进程，
    //而是判断等待队列不为空
    if (wq_has_sleeper(wq))
        //执行等待队列项上的回调函数
        wake_up_interruptible_sync_poll(&wq->wait, POLLIN |
POLLPRI |
                        POLLRDNORM | POLLRDBAND);
    sk_wake_async(sk, SOCK_WAKE_WAITD, POLL_IN);
    rcu_read_unlock();
}
```

这里的函数名其实都有迷惑人的地方。

- wq_has_sleeper，对于简单的 recvfrom 系统调用来说，确实是判断是否有进程阻塞。但是对于 epoll 下的 socket 只是判断等待队列不为空，不一定有进程阻塞的。
- wake_up_interruptible_sync_poll，只是会进入到 socket 等待队列项上设置的回调函数，并不一定有唤醒进程的操作。

那接下来就是我们重点看 wake_up_interruptible_sync_poll 。

我们看一下内核是怎么找到等待队列项里注册的回调函数的。

```
//file: include/linux/wait.h
#define wake_up_interruptible_sync_poll(x, m)        \
    __wake_up_sync_key((x), TASK_INTERRUPTIBLE, 1, (void *)
(m))
```

```
//file: kernel/sched/core.c
void __wake_up_sync_key(wait_queue_head_t *q, unsigned int
mode,
            int nr_exclusive, void *key)
{
    ...
    __wake_up_common(q, mode, nr_exclusive, wake_flags, key);
}
```

接着进入 __wake_up_common

```
static void __wake_up_common(wait_queue_head_t *q, unsigned int
mode,
            int nr_exclusive, int wake_flags, void *key)
{
    wait_queue_t *curr, *next;

    list_for_each_entry_safe(curr, next, &q->task_list,
task_list) {
        unsigned flags = curr->flags;

        if (curr->func(curr, mode, wake_flags, key) &&
                (flags & WQ_FLAG_EXCLUSIVE) && !--nr_exclusive)
            break;
    }
}
```

在 __wake_up_common 中，选出等待队列里注册某个元素 curr， 回调其 curr->func。 回忆我们 ep_insert 调用的时候，把这个 func 设置成 ep_poll_callback 了。

执行 socket 就绪回调函数

在上一小节找到了 socket 等待队列项里注册的函数 ep_poll_callback，软中断接着就会调用它。

```
//file: fs/eventpoll.c
static int ep_poll_callback(wait_queue_t *wait, unsigned mode,
int sync, void *key)
```

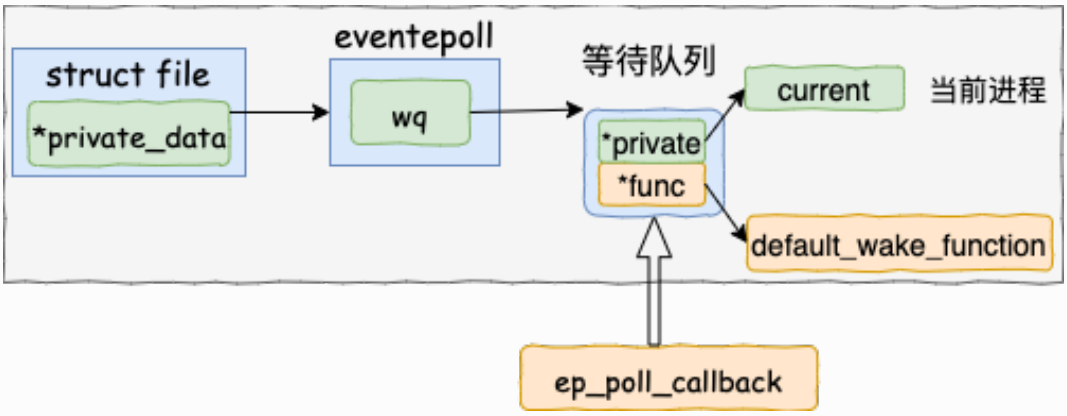
```
{
    //获取 wait 对应的 epitem
    struct epitem *epi = ep_item_from_wait(wait);

    //获取 epitem 对应的 eventpoll 结构体
    struct eventpoll *ep = epi->ep;

    //1. 将当前epitem 添加到 eventpoll 的就绪队列中
    list_add_tail(&epi->rdllink, &ep->rdllist);

    //2. 查看 eventpoll 的等待队列上是否有在等待
    if (waitqueue_active(&ep->wq))
        wake_up_locked(&ep->wq);
```

在 ep_poll_callback 根据等待任务队列项上的额外的 base 指针可以找到 epitem， 进而也可以找到 eventpoll对象。

首先它做的第一件事就是**把自己的 epitem 添加到 epoll 的就绪队列中**。

接着它又会查看 eventpoll 对象上的等待队列里是否有等待项（epoll_wait 执行的时候会设置）。

如果没执行软中断的事情就做完了。如果有等待项，那就查找到等待项里设置的回调函数。



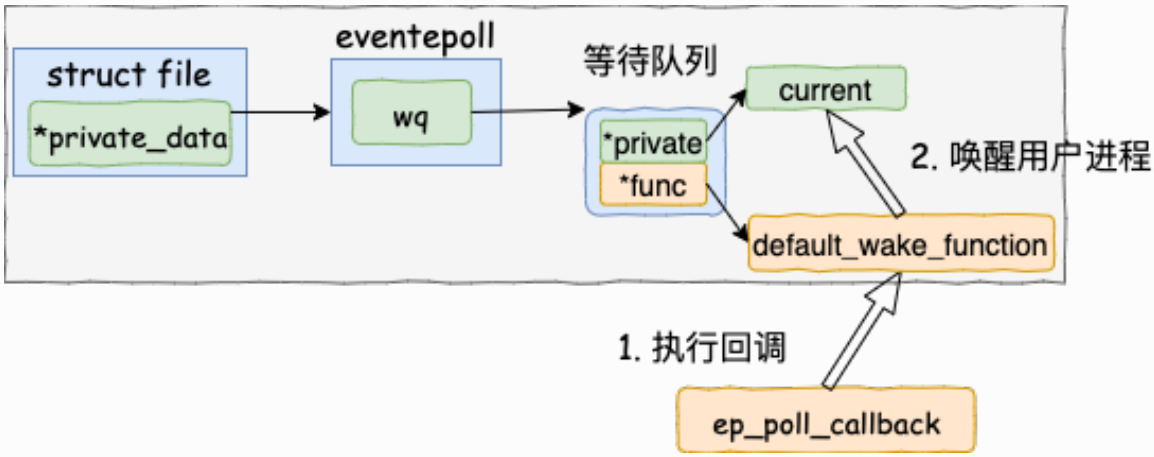
调用 wake_up_locked() => __wake_up_locked() => __wake_up_common。

```
static void __wake_up_common(wait_queue_head_t *q, unsigned int
mode,
            int nr_exclusive, int wake_flags, void *key)
{
    wait_queue_t *curr, *next;

    list_for_each_entry_safe(curr, next, &q->task_list,
task_list) {
        unsigned flags = curr->flags;

        if (curr->func(curr, mode, wake_flags, key) &&
                (flags & WQ_FLAG_EXCLUSIVE) && !--nr_exclusive)
            break;
    }
}
```

在 __wake_up_common里， 调用 curr->func。 这里的 func 是在 epoll_wait 是传入的 default_wake_function 函数。

执行 epoll 就绪通知

在default_wake_function 中找到等待队列项里的进程描述符，然后唤醒之。



源代码如下：

```
//file:kernel/sched/core.c
int default_wake_function(wait_queue_t *curr, unsigned mode,
int wake_flags,
                void *key)
{
    return try_to_wake_up(curr->private, mode, wake_flags);
}
```

等待队列项 curr->private 指针是在 epoll 对象上等待而被阻塞掉的进程。

将epoll_wait进程推入可运行队列，等待内核重新调度进程。然后epoll_wait对应的这个进程重新运行后，就从 schedule 恢复

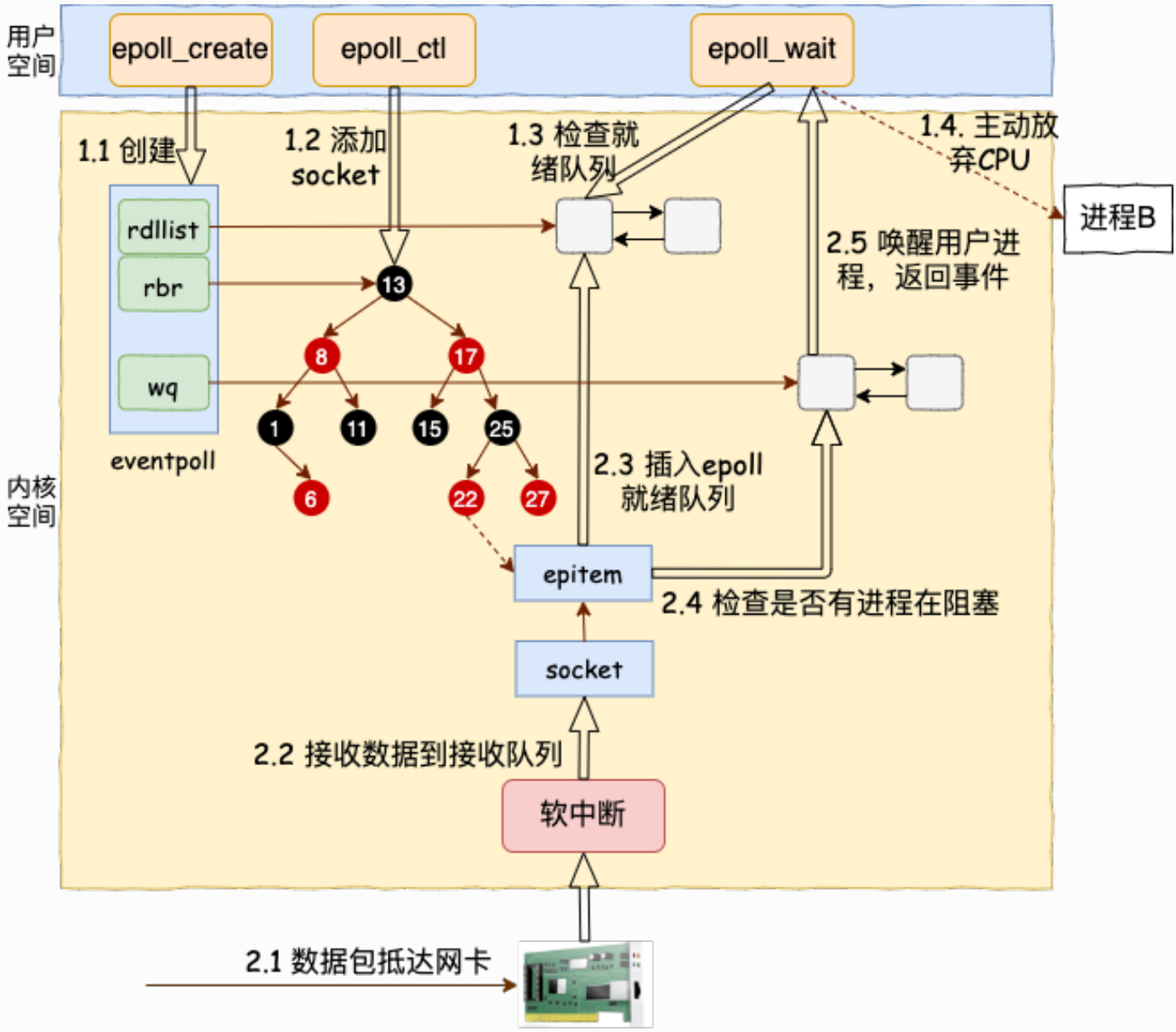
当进程醒来后，继续从 epoll_wait 时暂停的代码继续执行。把 rdlist 中就绪的事件返回给用户进程

```
//file: fs/eventpoll.c
static int ep_poll(struct eventpoll *ep, struct epoll_event
__user *events,
             int maxevents, long timeout)
{

    ......
    __remove_wait_queue(&ep->wq, &wait);

    set_current_state(TASK_RUNNING);
    }
check_events:
    //返回就绪事件给用户进程
    ep_send_events(ep, events, maxevents))
}
```

从用户角度来看，epoll_wait 只是多等了一会儿而已，但执行流程还是顺序的。

2.3.6 总结

我们来用一幅图总结一下 epoll 的整个工作路程。



其中软中断回调的时候回调函数也整理一下：

sock_def_readable： sock 对象初始化时设置的

=> ep_poll_callback : epoll_ctl 时添加到 socket 上的

=> default_wake_function: epoll_wait 是设置到 epoll上的

总结下，epoll 相关的函数里内核运行环境分两部分：

- 用户进程内核态。进行调用 epoll_wait 等函数时会将进程陷入内核态来执行。这部分代码负责查看接收队列，以及负责把当前进程阻塞掉，让出 CPU。
- 硬软中断上下文。在这些组件中，将包从网卡接收过来进行处理，然后放到 socket 的

接收队列。对于 epoll 来说，再找到 socket 关联的 epitem，并把它添加到 epoll 对象的就绪链表中。 这个时候再捎带检查一下 epoll 上是否有被阻塞的进程，如果有唤醒之。

为了介绍到每个细节，本文涉及到的流程比较多，把阻塞都介绍进来了。

但其实**在实践中，只要活儿足够的多，epoll_wait 根本都不会让进程阻塞**。用户进程会一直干活，一直干活，直到 epoll_wait 里实在没活儿可干的时候才主动让出 CPU。这就是 epoll 高效的地方所在！

2.4 网络包处理之 CPU 开销汇总

了解完了整个 TCP 协议下网络包的处理过程，现在我们总结一下其 CPU 开销都有哪些

1. **系统态 CPU 开销**：当用户进程调用 socket、connect、recvfrom 等等函数的时候，都会将用户进程陷入到内核态。在内核态里的 CPU 开销就是进程里的 sy 列。如果你的应用程序是网络 IO 密集型的，你的 sy 中可能大部分都是因为网络造成成。
2. **硬中断、软中断开销**：内核收到包以后会在硬中断、软中断上下文中进行内核相关工作的处理。硬中断由于逻辑简单，一般不会成为系统的瓶颈。这也就是你查看 cpu 开销是，hi 这一列大部分时候都是 0.0 的原因。但软中断中确确实实要执行好多好多的操作，包括协议栈的处理。在网络 IO 密集型应用中，si 会吃掉不少 CPU。
3. **进程上下文切换**：用户进程和内核在配合的时候有可能会引起进程的阻塞与唤醒。尤其是对于使用 recvfrom 系统调用的，每一次的等待都会造成进程被阻塞。当数据来临的时候，又会被唤醒。在网络请求量大的时候，会消耗非常多 CPU 周期来执行进程的上下文切换。在 epoll 中改善了很多，但当没有 IO 事件可做的时候，进程同样要被阻塞掉。

三、内核是如何发送网络包的

半年前我以源码的方式描述了网络包的接收过程。之后不断有粉丝提醒我，飞哥飞哥，你还没聊发送过程呢。好，安排！

在开始今天的文章之前，我先来请大家思考几个小问题。

- 问1：我们在查看内核发送数据消耗的 CPU 时，是应该看 sy 还是 si？
- 问2：为什么你服务器上的 /proc/softirqs 里 NET_RX 要比 NET_TX 大的多的多？
- 问3：发送网络数据的时候都涉及到哪些内存拷贝操作？

这些问题虽然在线上经常看到，但我们似乎很少去深究。如果真的能透彻地把这些问题理解到位，我们对性能的掌控能力将会变得更强。

带着这三个问题，我们开始今天对 Linux 内核网络发送过程的深度剖析。还是按照我们之前的传统，先从一段简单的代码作为切入。如下代码是一个典型服务器程序的典型的缩微代码：

```
int main(){
  fd = socket(AF_INET, SOCK_STREAM, 0);
  bind(fd, ...);
  listen(fd, ...);

  cfd = accept(fd, ...);

  // 接收用户请求
  read(cfd, ...);

  // 用户请求处理
  dosometing();

  // 给用户返回结果
  send(cfd, buf, sizeof(buf), 0);
}
```

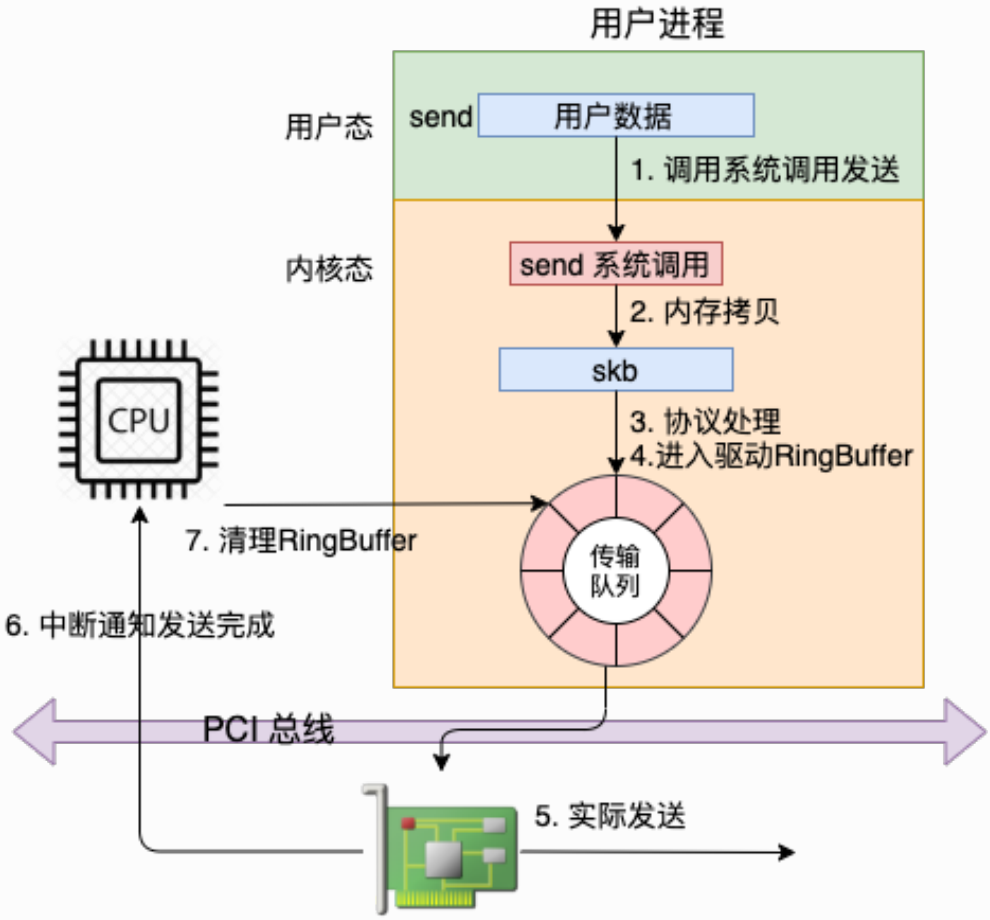
今天我们来讨论上述代码中，调用 send 之后内核是怎么样把数据包发送出去的。本文基于 Linux 3.10，网卡驱动采用Intel的igb网卡举例。

预警：本文共有两万六千字，20 多张图，长文慎入！

3.1 LINUX 网络发送过程总览

飞哥觉得看 Linux 源码最重要的是得有整体上的把握，而不是一开始就陷入各种细节。

我这里先给大家准备了一个总的流程图，简单阐述下 send 发送了的数据是如何一步一步被发送到网卡的。



在这幅图中，我们看到用户数据被拷贝到内核态，然后经过协议栈处理后进入到了 RingBuffer 中。随后网卡驱动真正将数据发送了出去。当发送完成的时候，是通过硬中断来通知 CPU，然后清理 RingBuffer。

因为文章后面要进入源码，所以我们再从源码的角度给出一个流程图。

应用层

```c
int main(){
    // 给用户返回结果
    send(cfd, buf, sizeof(buf), 0);
}
```

系统调用

```c
//file: net/socket.c
SYSCALL_DEFINE6(sendto, int, fd, ...)
{
    //构造 msghdr 并赋值
    struct msghdr msg;
    ......

    //发送数据
    sock_sendmsg(sock, &msg, len);
```

```
}
```

```
//file: net/socket.c
static inline int __sock_sendmsg_nosec(...)
{
    return sock->ops->sendmsg(iocb, sock, msg, size);
}
```

协议栈

```
//file: net/ipv4/af_inet.c
int inet_sendmsg(......)
{
    return sk->sk_prot->sendmsg(iocb, sk, msg, size);
}
```

传输层

```
//file: net/ipv4/tcp.c
int tcp_sendmsg(...)
{
    ...
}

//file: net/ipv4/tcp_output.c
static int tcp_transmit_skb(......)
{
    //封装TCP头
    th = tcp_hdr(skb);
    th->source      = inet->inet_sport;
    th->dest        = inet->inet_dport;

    //调用网络层发送接口
    err = icsk->icsk_af_ops->queue_xmit(skb);
}
```

网络层

```
//file: net/ipv4/ip_output.c
int ip_queue_xmit(struct sk_buff *skb, struct flowi *fl)
{
    res = ip_local_out(skb);
}

//file: net/ipv4/ip_output.c
static inline int ip_finish_output2(struct sk_buff *skb)
{
    //继续向下层传递
    int res = dst_neigh_output(dst, neigh, skb);
}
```

链路层

```
//file: include/net/dst.h
static inline int dst_neigh_output(...)
{
    ......
    return neigh_hh_output(hh, skb);
}

//file: include/net/neighbour.h
static inline int neigh_hh_output(...)
{
    ......
    skb_push(skb, hh_len);
    return dev_queue_xmit(skb);
}
```

网络设备子系统

```
//file: net/core/dev.c
int dev_queue_xmit(struct sk_buff *skb)
{
    //选择发送队列并获取 qdisc
    txq = netdev_pick_tx(dev, skb);
    q = rcu_dereference_bh(txq->qdisc);

    //则调用__dev_xmit_skb 继续发送
    rc = __dev_xmit_skb(skb, q, dev, txq);
}
```

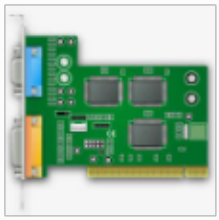
```
//file: net/core/dev.c
int dev_hard_start_xmit(...)
{
    //获取设备的回调函数集合 ops
    const struct net_device_ops *ops = dev->netdev_ops;

    //调用驱动里的发送回调函数 ndo_start_xmit 将数据包传给网卡设备
    skb_len = skb->len;
    rc = ops->ndo_start_xmit(skb, dev);
}
```

驱动程序

```
//file: drivers/net/ethernet/intel/igb/igb_main.c
static netdev_tx_t igb_xmit_frame(...)
{
    return igb_xmit_frame_ring(skb, ...);
}

netdev_tx_t igb_xmit_frame_ring(...)
{
    //获取TX Queue 中下一个可用缓冲区信息
    first = &tx_ring->tx_buffer_info[tx_ring->next_to_use];
    first->skb = skb;
    first->bytecount = skb->len;

    //igb_tx_map 函数准备给设备发送的数据。
```

```
    igb_tx_map(tx_ring, first, hdr_len);
}
```

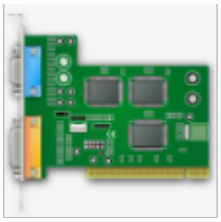
硬件



虽然数据这时已经发送完毕，但是其实还有一件重要的事情没有做，那就是释放缓存队列等内存。

那内核是如何知道什么时候才能释放内存的呢，当然是等网络发送完毕之后。网卡在发送完毕的时候，会给 CPU 发送一个硬中断来通知 CPU。更完整的流程看图：

硬件



硬中断

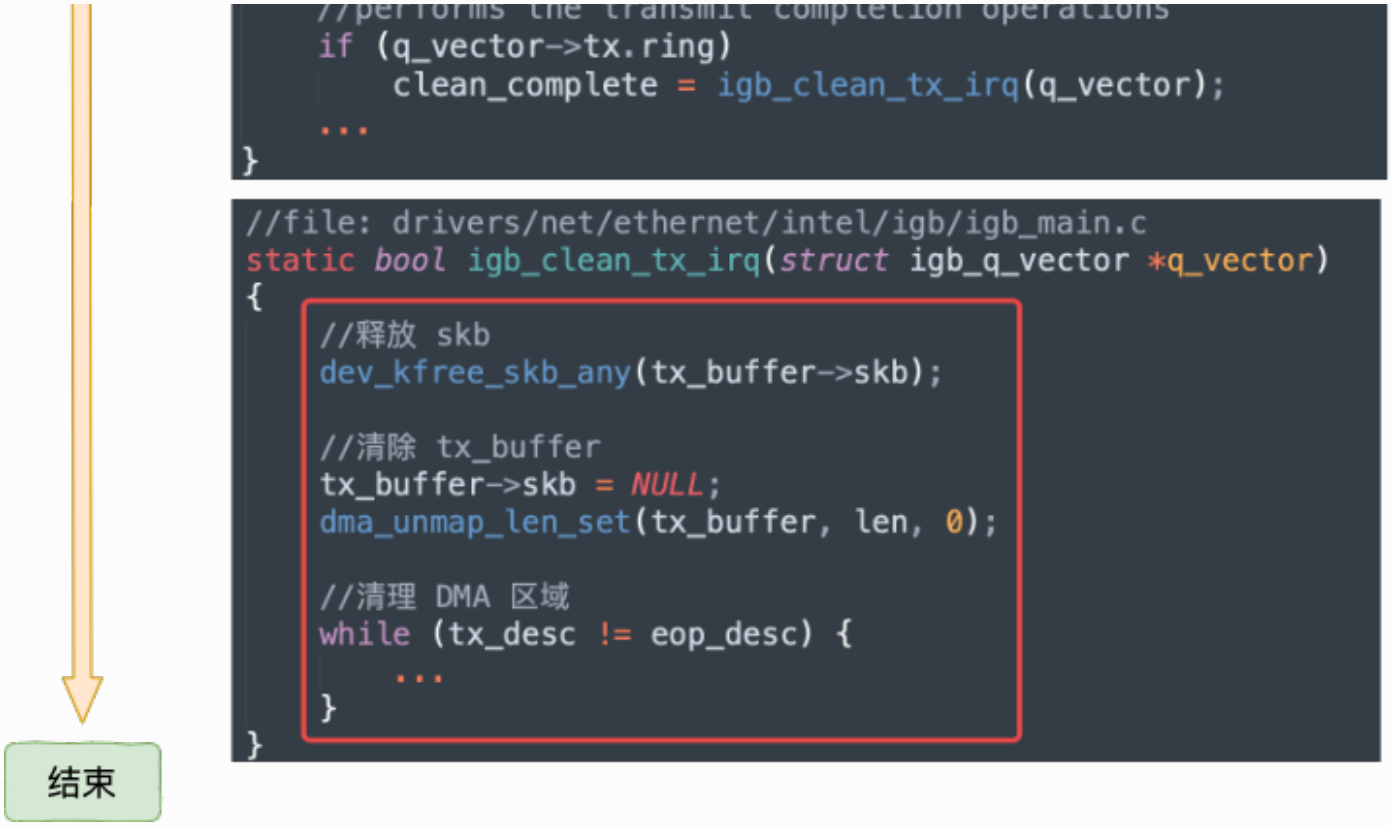
```
//file: drivers/net/ethernet/intel/igb/igb_main.c
static irqreturn_t igb_msix_ring(int irq, void *data)
{
    napi_schedule(&q_vector->napi);
}

static inline void ____napi_schedule(...)
{
    list_add_tail(&napi->poll_list, &sd->poll_list);
    __raise_softirq_irqoff(NET_RX_SOFTIRQ);
}
```

软中断

```
static void net_rx_action(struct softirq_action *h)
{
    while (!list_empty(&sd->poll_list)) {
        ......
        n = list_first_entry(&sd->poll_list, ...);
        work = n->poll(n, weight);
    }
}
```

驱动

```
//file: drivers/net/ethernet/intel/igb/igb_main.c
static int igb_poll(struct napi_struct *napi, int budget)
{
    //performs the transmit completion operations
```



注意，我们今天的主题虽然是发送数据，但是硬中断最终触发的软中断却是 NET_RX_SOFTIRQ，而并不是 NET_TX_SOFTIRQ！！！ (T 是 transmit 的缩写，R 表示 receive)

意不意外，惊不惊喜？？？

所以这就是开篇问题 1 的一部分的原因（注意，这只是一部分原因）。

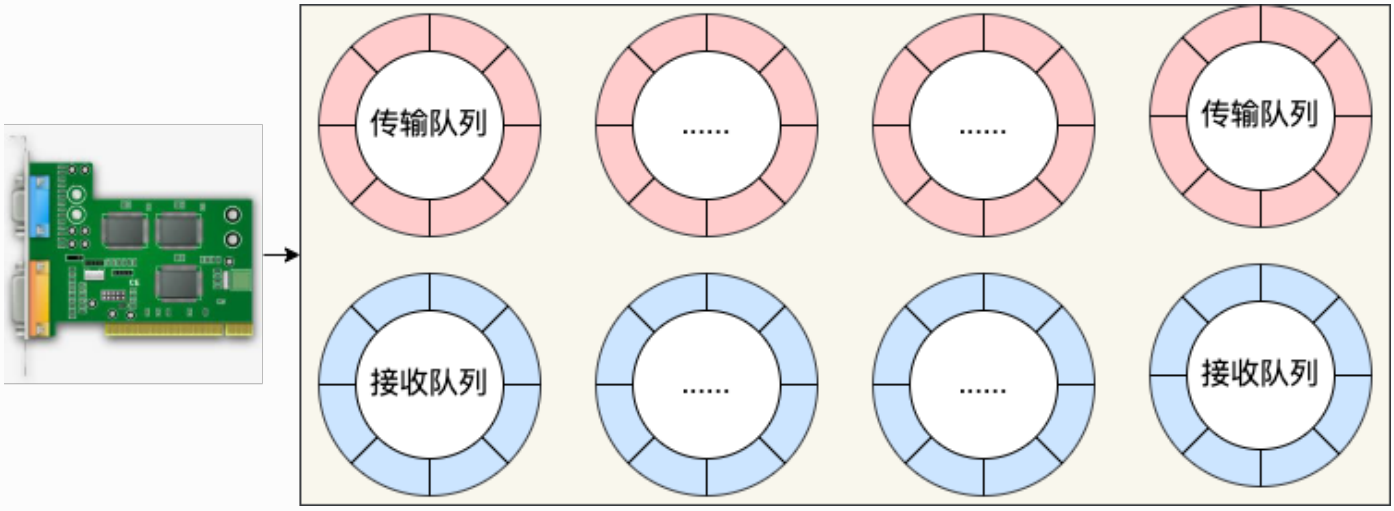
> 问1：在服务器上查看 /proc/softirqs，为什么 NET_RX 要比 NET_TX 大的多的多?

传输完成最终会触发 NET_RX，而不是 NET_TX。 所以自然你观测 /proc/softirqs 也就能看到 NET_RX 更多了。

好，现在你已经对内核是怎么发送网络包的有一个全局上的把握了。不要得意，我们需要了解的细节才是更有价值的地方，让我们继续！！

3.2 网卡启动准备

现在的服务器上的网卡一般都是支持多队列的。每一个队列上都是由一个 RingBuffer 表示的，开启了多队列以后的的网卡就会对应有多个 RingBuffer。



网卡在启动时最重要的任务之一就是分配和初始化 RingBuffer，理解了 RingBuffer 将会非常有助于后面我们掌握发送。因为今天的主题是发送，所以就以传输队列为例，我们来看下网卡启动时分配 RingBuffer 的实际过程。

在网卡启动的时候，会调用到 __igb_open 函数，RingBuffer 就是在这里分配的。

```
//file: drivers/net/ethernet/intel/igb/igb_main.c
static int __igb_open(struct net_device *netdev, bool resuming)
{
  struct igb_adapter *adapter = netdev_priv(netdev);

  //分配传输描述符数组
  err = igb_setup_all_tx_resources(adapter);

  //分配接收描述符数组
  err = igb_setup_all_rx_resources(adapter);

  //开启全部队列
  netif_tx_start_all_queues(netdev);
}
```

在上面 __igb_open 函数调用 igb_setup_all_tx_resources 分配所有的传输 RingBuffer，调用 igb_setup_all_rx_resources 创建所有的接收 RingBuffer。

```
//file: drivers/net/ethernet/intel/igb/igb_main.c
static int igb_setup_all_tx_resources(struct igb_adapter
*adapter)
{
  //有几个队列就构造几个 RingBuffer
  for (i = 0; i < adapter->num_tx_queues; i++) {
    igb_setup_tx_resources(adapter->tx_ring[i]);
  }
}
```

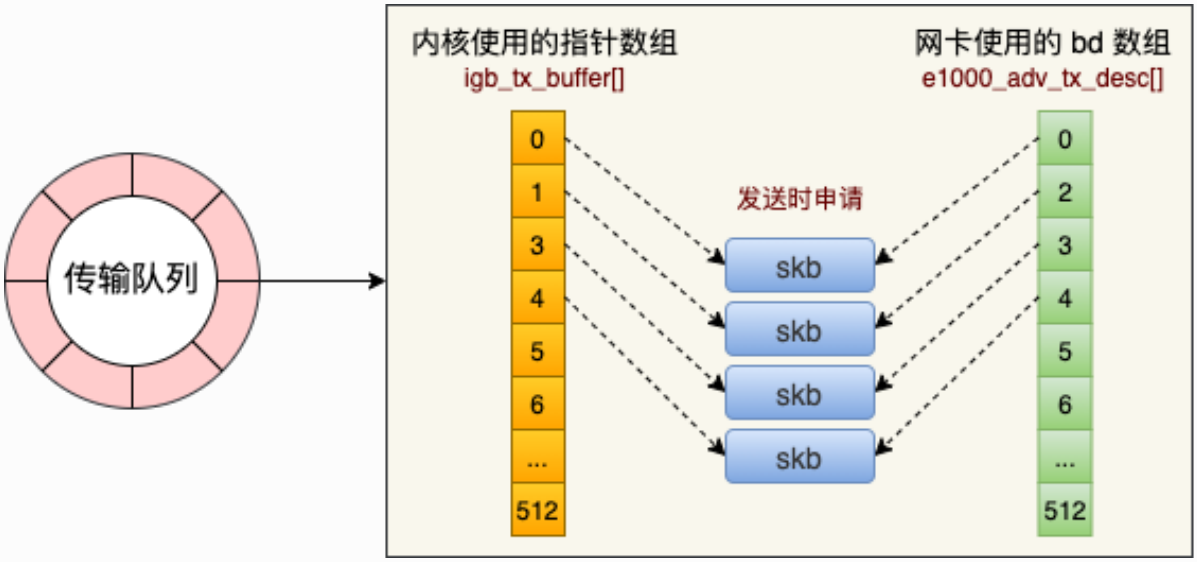
真正的 RingBuffer 构造过程是在 igb_setup_tx_resources 中完成的。

```
//file: drivers/net/ethernet/intel/igb/igb_main.c
int igb_setup_tx_resources(struct igb_ring *tx_ring)
{
  //1.申请 igb_tx_buffer 数组内存
  size = sizeof(struct igb_tx_buffer) * tx_ring->count;
  tx_ring->tx_buffer_info = vzalloc(size);

  //2.申请 e1000_adv_tx_desc DMA 数组内存
  tx_ring->size = tx_ring->count * sizeof(union
e1000_adv_tx_desc);
  tx_ring->size = ALIGN(tx_ring->size, 4096);
  tx_ring->desc = dma_alloc_coherent(dev, tx_ring->size,
            &tx_ring->dma, GFP_KERNEL);

  //3.初始化队列成员
  tx_ring->next_to_use = 0;
  tx_ring->next_to_clean = 0;
}
```

从上述源码可以看到，实际上一个 RingBuffer 的内部不仅仅是一个环形队列数组，而是有两个。

1）igb_tx_buffer 数组：这个数组是内核使用的，通过 vzalloc 申请的。
2）e1000_adv_tx_desc 数组：这个数组是网卡硬件使用的，硬件是可以通过 DMA 直接访问这块内存，通过 dma_alloc_coherent 分配。

这个时候它们之间还没有啥联系。将来在发送的时候，这两个环形数组中相同位置的指针将都将指向同一个 skb。这样，内核和硬件就能共同访问同样的数据了，内核往 skb 里写数据，网卡硬件负责发送。



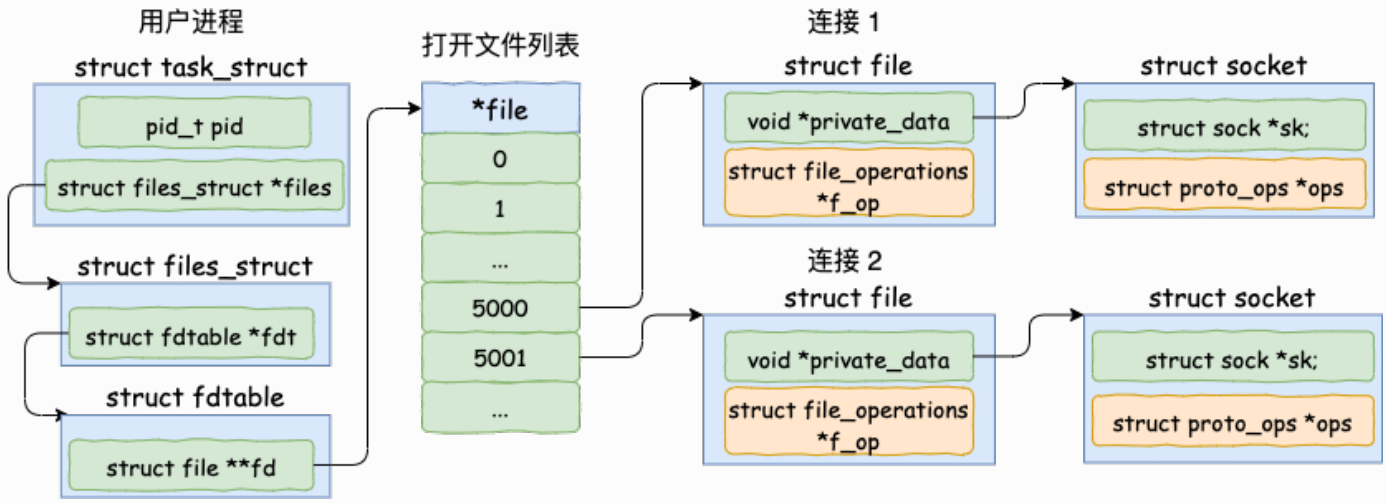
最后调用 netif_tx_start_all_queues 开启队列。另外，对于硬中断的处理函数 igb_msix_ring 其实也是在 __igb_open 中注册的。

3.3 ACCEPT 创建新 SOCKET

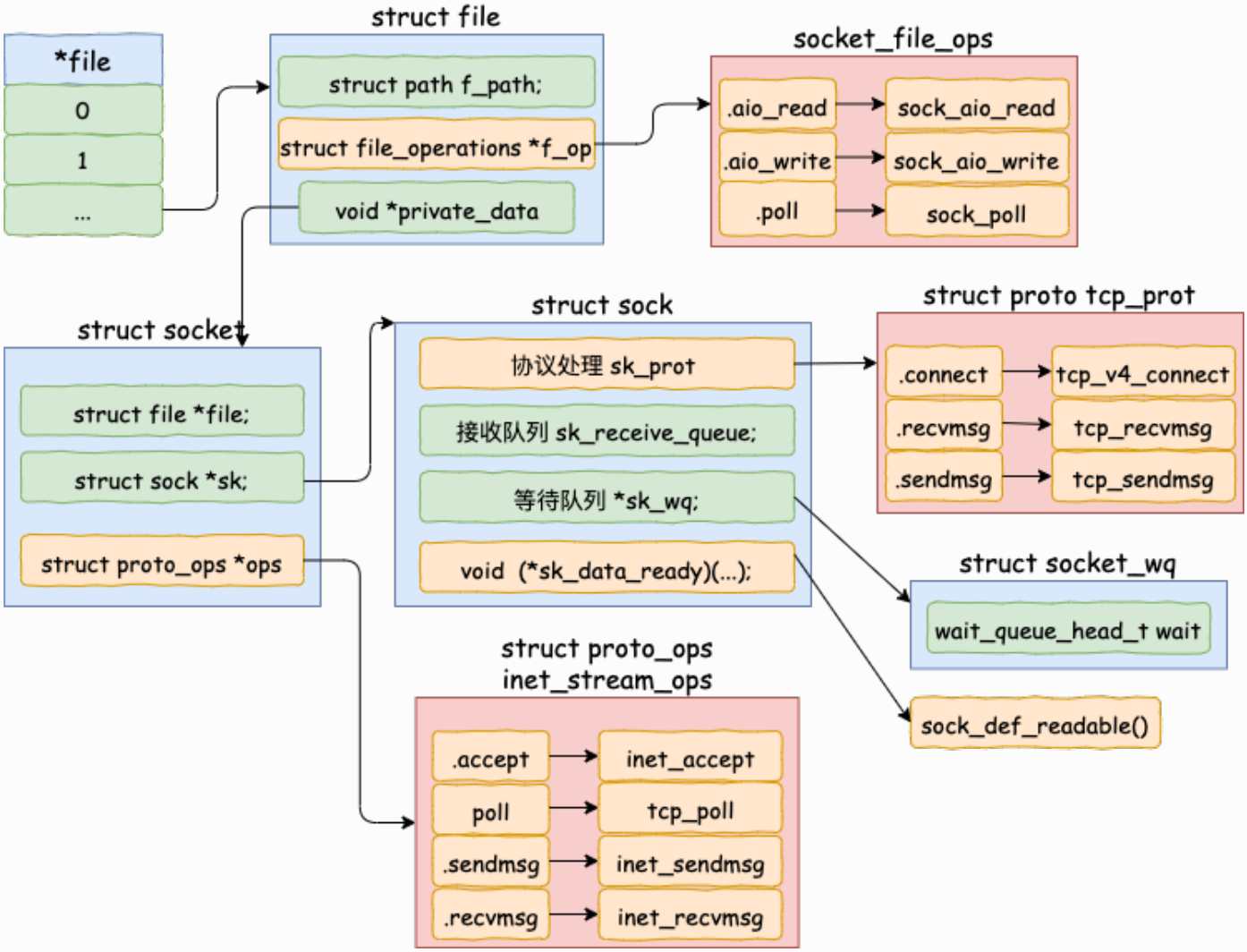
在发送数据之前，我们往往还需要一个已经建立好连接的 socket。

我们就以开篇服务器缩微源代码中提到的 accept 为例，当 accept 之后，进程会创建一个新的 socket 出来，然后把它放到当前进程的打开文件列表中，专门用于和对应的客户端通信。

假设服务器进程通过 accept 和客户端建立了两条连接，我们来简单看一下这两条连接和进程的关联关系。



其中代表一条连接的 socket 内核对象更为具体一点的结构图如下。



为了避免喧宾夺主，accept 详细的源码过程这里就不介绍了，感兴趣请参考 《图解 | 深入揭秘 epoll 是如何实现 IO 多路复用的！》。一文中的第一部分。

今天我们还是把重点放到数据发送过程上。

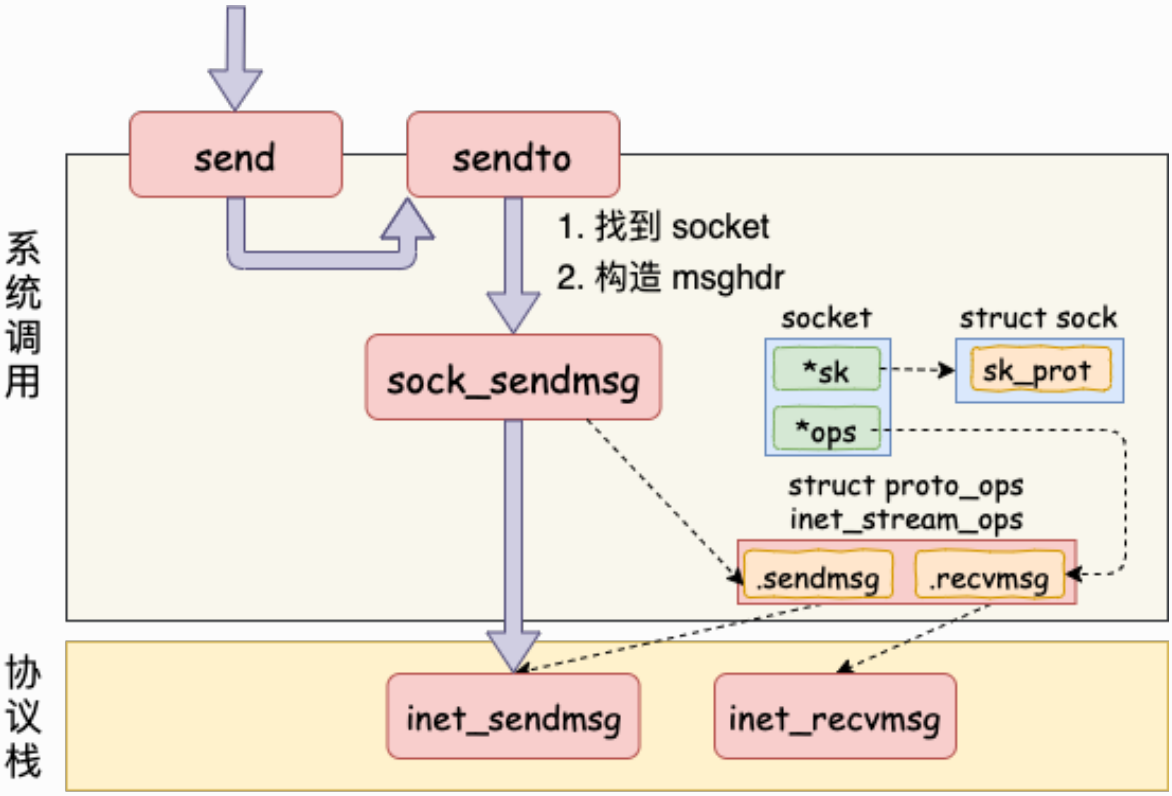
3.4 发送数据真正开始

3.4.1 send 系统调用实现

send 系统调用的源码位于文件 net/socket.c 中。在这个系统调用里，内部其实真正使用的是 sendto 系统调用。整个调用链条虽然不短，但其实主要只干了两件简单的事情，

- 第一是在内核中把真正的 socket 找出来，在这个对象里记录着各种协议栈的函数地址。
- 第二是构造一个 struct msghdr 对象，把用户传入的数据，比如 buffer地址、数据长度啥的，统统都装进去.

剩下的事情就交给下一层，协议栈里的函数 inet_sendmsg 了，其中 inet_sendmsg 函数的地址是通过 socket 内核对象里的 ops 成员找到的。大致流程如图。



有了上面的了解，我们再看起源码就要容易许多了。源码如下：

```
//file: net/socket.c
SYSCALL_DEFINE4(send, int, fd, void __user *, buff, size_t,
len,
    unsigned int, flags)
```

```
{
  return sys_sendto(fd, buff, len, flags, NULL, 0);
}

SYSCALL_DEFINE6(......)
{
  //1.根据 fd 查找到 socket
  sock = sockfd_lookup_light(fd, &err, &fput_needed);

  //2.构造 msghdr
  struct msghdr msg;
  struct iovec iov;

  iov.iov_base = buff;
  iov.iov_len = len;
  msg.msg_iovlen = 1;

  msg.msg_iov = &iov;
  msg.msg_flags = flags;
  ......

  //3.发送数据
  sock_sendmsg(sock, &msg, len);
}
```

从源码可以看到，我们在用户态使用的 send 函数和 sendto 函数其实都是 sendto 系统调用实现的。send 只是为了方便，封装出来的一个更易于调用的方式而已。

在 sendto 系统调用里，首先根据用户传进来的 socket 句柄号来查找真正的 socket 内核对象。接着把用户请求的 buff、len、flag 等参数都统统打包到一个 struct msghdr 对象中。

接着调用了 sock_sendmsg => __sock_sendmsg ==> __sock_sendmsg_nosec。
在__sock_sendmsg_nosec 中，调用将会由系统调用进入到协议栈，我们来看它的源码。

```
//file: net/socket.c
static inline int __sock_sendmsg_nosec(...)
{
  ......
  return sock->ops->sendmsg(iocb, sock, msg, size);
}
```

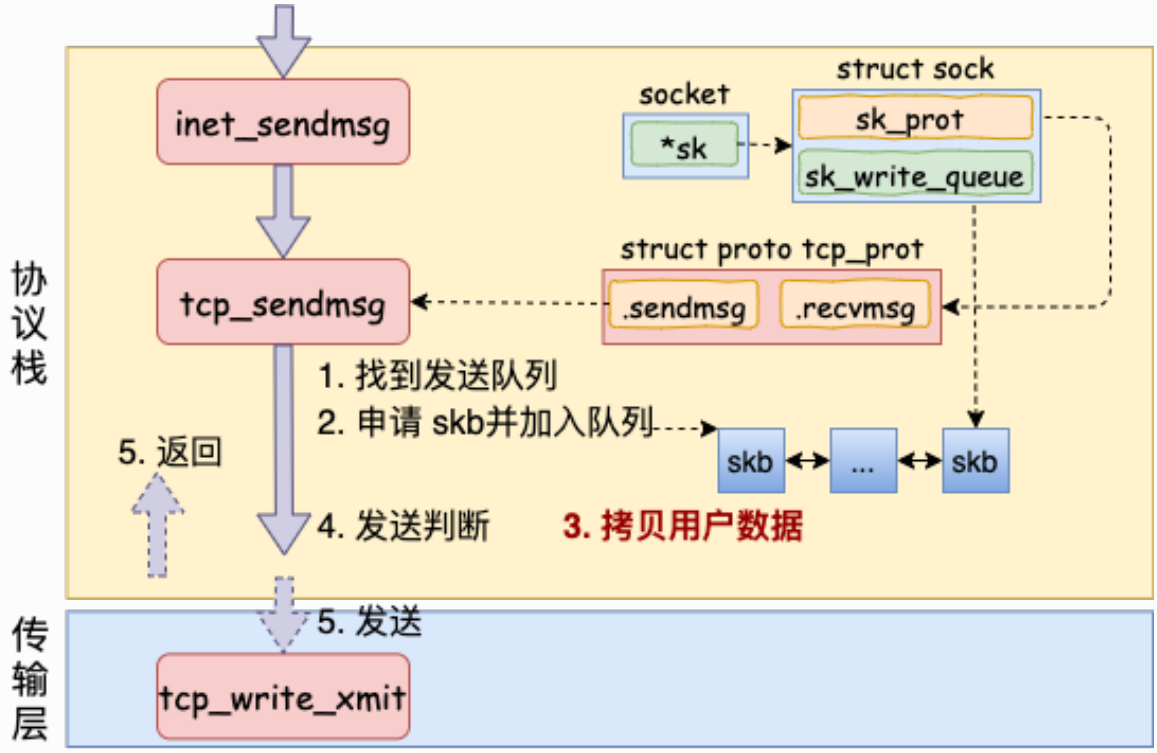
通过第三节里的 socket 内核对象结构图，我们可以看到，这里调用的是 sock->ops->sendmsg 实际执行的是 inet_sendmsg。这个函数是 AF_INET 协议族提供的通用发送函数。

3.4.2 传输层处理

1）传输层拷贝

在进入到协议栈 inet_sendmsg 以后，内核接着会找到 socket 上的具体协议发送函数。对于 TCP 协议来说，那就是 tcp_sendmsg（同样也是通过 socket 内核对象找到的）。

在这个函数中，内核会申请一个内核态的 skb 内存，将用户待发送的数据拷贝进去。注意这个时候不一定会真正开始发送，如果没有达到发送条件的话很可能这次调用直接就返回了。大概过程如图：



我们来看 inet_sendmsg 函数的源码。

```
//file: net/ipv4/af_inet.c
int inet_sendmsg(......)
{
  ......
  return sk->sk_prot->sendmsg(iocb, sk, msg, size);
}
```

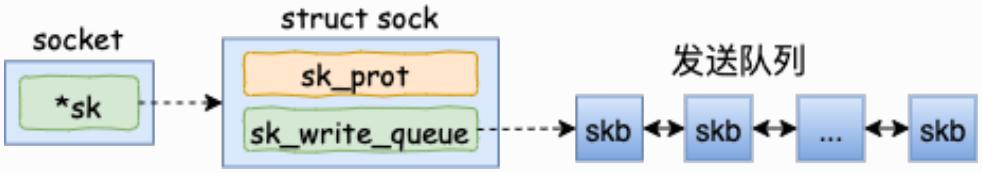
在这个函数中会调用到具体协议的发送函数。同样参考第三节里的 socket 内核对象结构图，我们看到对于 TCP 协议下的 socket 来说，来说 sk->sk_prot->sendmsg 指向的是 tcp_sendmsg（对于 UPD 来说是 udp_sendmsg）。

tcp_sendmsg 这个函数比较长，我们分多次来看它。 先看这一段

```
//file: net/ipv4/tcp.c
int tcp_sendmsg(...)
{
  while(...){
    while(...){
      //获取发送队列
      skb = tcp_write_queue_tail(sk);

      //申请skb 并拷贝
      ......
    }
  }
}
```

```
//file: include/net/tcp.hstatic inline struct sk_buff
*tcp_write_queue_tail(const struct sock *sk){ return
skb_peek_tail(&sk->sk_write_queue);}
```

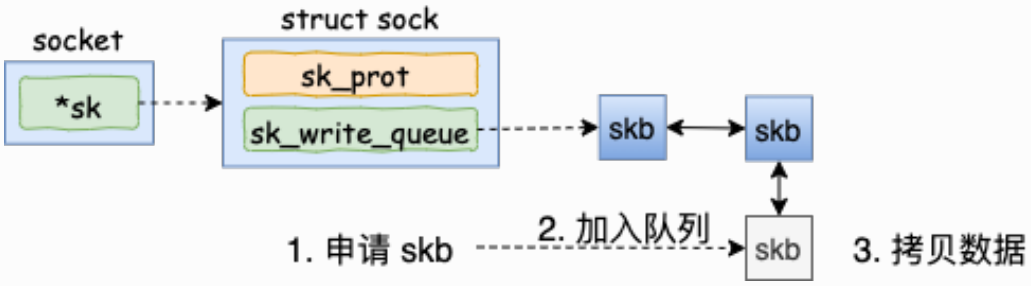
理解对 socket 调用 tcp_write_queue_tail 是理解发送的前提。如上所示，这个函数是在获取 socket 发送队列中的最后一个 skb。 skb 是 struct sk_buff 对象的简称，用户的发送队列就是该对象组成的一个链表。



我们再接着看 tcp_sendmsg 的其它部分。

```
//file: net/ipv4/tcp.cint tcp_sendmsg(struct kiocb *iocb,
struct sock *sk, struct msghdr *msg,    size_t size){ //获取用户
传递过来的数据和标志  iov = msg->msg_iov; //用户数据地址  iovlen =
msg->msg_iovlen; //数据块数为1  flags = msg->msg_flags; //各种标志
//遍历用户层的数据块 while (--iovlen >= 0) {   //待发送数据块的地址
unsigned char __user *from = iov->iov_base;   while (seglen >
0) {      //需要申请新的 skb      if (copy <= 0) {        //申请
skb，并添加到发送队列的尾部        skb = sk_stream_alloc_skb(sk,
select_size(sk, sg),                 sk->sk_allocation);
//把 skb 挂到socket的发送队列上        skb_entail(sk, skb);      }
// skb 中有足够的空间      if (skb_availroom(skb) > 0) {       //
拷贝用户空间的数据到内核空间，同时计算校验和        //from是用户空间的数
据地址        skb_add_data_nocache(sk, skb, from, copy);      }
......
```

这个函数比较长，不过其实逻辑并不复杂。其中 msg->msg_iov 存储的是用户态内存的要发送的数据的 buffer。接下来在内核态申请内核内存，比如 skb，并把用户内存里的数据拷贝到内核态内存中。**这就会涉及到一次或者几次内存拷贝的开销**。



至于内核什么时候真正把 skb 发送出去。在 tcp_sendmsg 中会进行一些判断。

```
//file: net/ipv4/tcp.cint tcp_sendmsg(...){ while(...){
while(...){     //申请内核内存并进行拷贝      //发送判断       if
(forced_push(tp)) {        tcp_mark_push(tp, skb);
__tcp_push_pending_frames(sk, mss_now, TCP_NAGLE_PUSH);     }
else if (skb == tcp_send_head(sk))        tcp_push_one(sk,
mss_now);          }     continue;    } }}
```

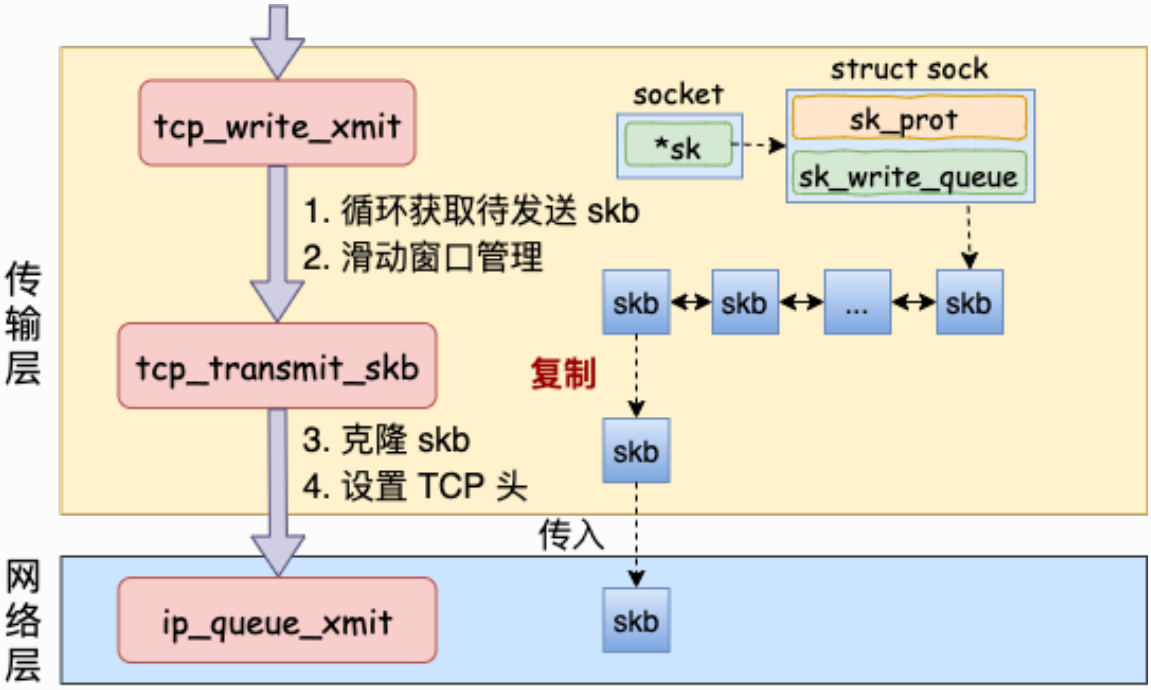
只有满足 forced_push(tp) 或者 skb == tcp_send_head(sk) 成立的时候，内核才会真正启动发送数据包。其中 forced_push(tp) 判断的是未发送的数据数据是否已经超过最大窗口的一半了。

条件都不满足的话，**这次的用户要发送的数据只是拷贝到内核就算完事了！**

2）传输层发送

假设现在内核发送条件已经满足了，我们再来跟踪一下实际的发送过程。 对于上小节函数中，当满足真正发送条件的时候，无论调用的是 __tcp_push_pending_frames 还是 tcp_push_one 最终都实际会执行到 tcp_write_xmit。

所以我们直接从 tcp_write_xmit 看起，这个函数处理了传输层的拥塞控制、滑动窗口相关的工作。满足窗口要求的时候，设置一下 TCP 头然后将 skb 传到更低的网络层进行处理。



我们来看下 tcp_write_xmit 的源码。

```
//file: net/ipv4/tcp_output.cstatic bool tcp_write_xmit(struct
sock *sk, unsigned int mss_now, int nonagle,        int
push_one, gfp_t gfp){  //循环获取待发送 skb while ((skb =
tcp_send_head(sk)))   {   //滑动窗口相关    cwnd_quota =
tcp_cwnd_test(tp, skb);    tcp_snd_wnd_test(tp, skb, mss_now);
tcp_mss_split_point(...);   tso_fragment(sk, skb, ...);
......    //真正开启发送    tcp_transmit_skb(sk, skb, 1, gfp);
}}
```

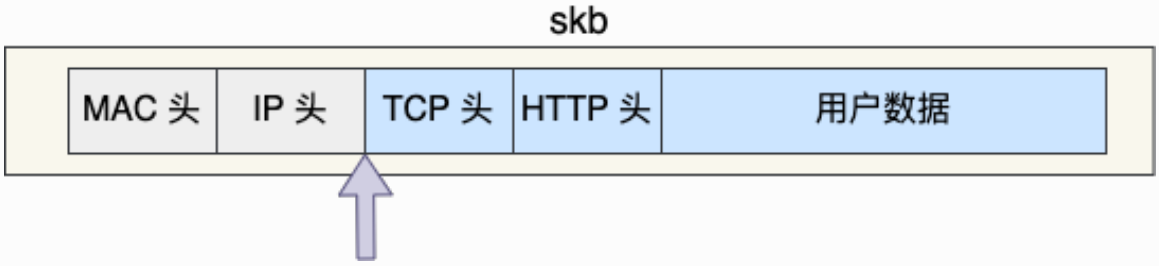
可以看到我们之前在网络协议里学的滑动窗口、拥塞控制就是在这个函数中完成的，这部分就不过多展开了，感兴趣同学自己找这段源码来读。我们今天只看发送主过程，那就走到了tcp_transmit_skb。

```
//file: net/ipv4/tcp_output.cstatic int tcp_transmit_skb(struct sock *sk, struct sk_buff *skb, int clone_it,        gfp_t gfp_mask){  //1.克隆新 skb 出来  if (likely(clone_it)) {   skb = skb_clone(skb, gfp_mask);   ......  } //2.封装 TCP 头  th = tcp_hdr(skb);  th->source    = inet->inet_sport; th->dest    = inet->inet_dport; th->window    = ...;  th->urg     = ...; ......  //3.调用网络层发送接口 err = icsk->icsk_af_ops->queue_xmit(skb, &inet->cork.fl);}
```

第一件事是先克隆一个新的 skb，这里重点说下为什么要复制一个 skb 出来呢?

是因为 skb 后续在调用网络层，最后到达网卡发送完成的时候，这个 skb 会被释放掉。而我们知道 TCP 协议是支持丢失重传的，在收到对方的 ACK 之前，这个 skb 不能被删除。所以内核的做法就是每次调用网卡发送的时候，实际上传递出去的是 skb 的一个拷贝。等收到 ACK 再真正删除。

第二件事是修改 skb 中的 TCP header，根据实际情况把 TCP 头设置好。这里要介绍一个小技巧，skb 内部其实包含了网络协议中所有的 header。在设置 TCP 头的时候，只是把指针指向 skb 的合适位置。后面再设置 IP 头的时候，在把指针挪一挪就行，避免频繁的内存申请和拷贝，效率很高。



tcp_transmit_skb 是发送数据位于传输层的最后一步，接下来就可以进入到网络层进行下一层的操作了。调用了网络层提供的发送接口icsk->icsk_af_ops->queue_xmit()。

在下面的这个源码中，我们的知道了 queue_xmit 其实指向的是 ip_queue_xmit 函数。

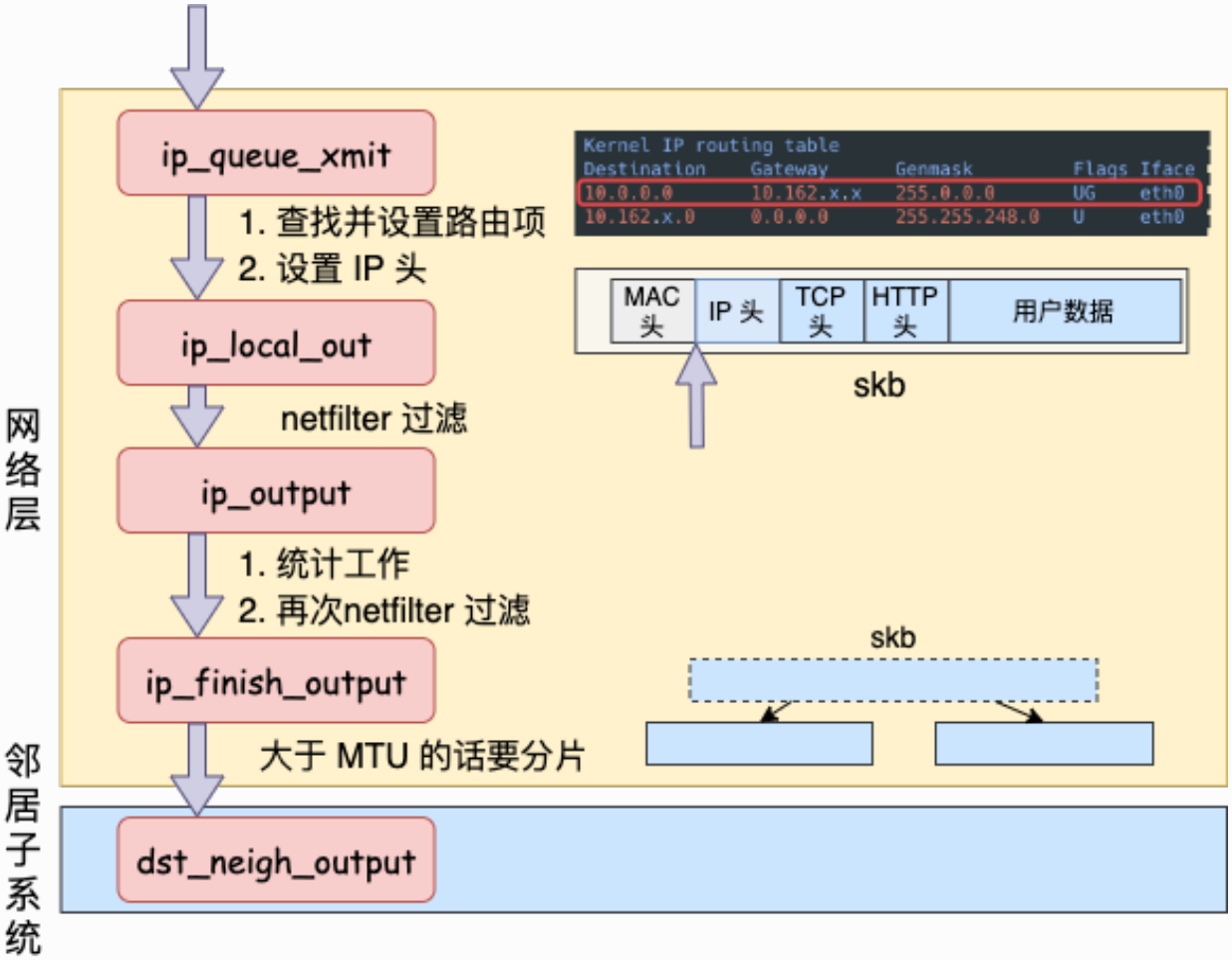
```
//file: net/ipv4/tcp_ipv4.c
const struct inet_connection_sock_af_ops ipv4_specific = {
   .queue_xmit     = ip_queue_xmit,
   .send_check     = tcp_v4_send_check,
   ...
}
```

自此，传输层的工作也就都完成了。 数据离开了传输层，接下来将会进入到内核在网络层的实现里。

3.4.3 网络层发送处理

Linux 内核网络层的发送的实现位于 net/ipv4/ip_output.c 这个文件。传输层调用到的 ip_queue_xmit 也在这里。（从文件名上也能看出来进入到 IP 层了，源文件名已经从 tcp_xxx 变成了 ip_xxx。）

在网络层里主要处理路由项查找、IP 头设置、netfilter 过滤、skb 切分（大于 MTU 的话）等几项工作，处理完这些工作后会交给更下层的邻居子系统来处理。



我们来看网络层入口函数 ip_queue_xmit 的源码：

```
//file: net/ipv4/ip_output.c
int ip_queue_xmit(struct sk_buff *skb, struct flowi *fl)
{
  //检查 socket 中是否有缓存的路由表
  rt = (struct rtable *)__sk_dst_check(sk, 0);
  if (rt == NULL) {
    //没有缓存则展开查找
    //则查找路由项， 并缓存到 socket 中
    rt = ip_route_output_ports(...);
    sk_setup_caps(sk, &rt->dst);
  }

  //为 skb 设置路由表
  skb_dst_set_noref(skb, &rt->dst);

  //设置 IP header
  iph = ip_hdr(skb);
  iph->protocol = sk->sk_protocol;
  iph->ttl      = ip_select_ttl(inet, &rt->dst);
  iph->frag_off = ...;

  //发送
  ip_local_out(skb);
}
```

ip_queue_xmit 已经到了网络层，在这个函数里我们看到了网络层相关的功能路由项查找，如果找到了则设置到 skb 上（没有路由的话就直接报错返回了）。

在 Linux 上通过 route 命令可以看到你本机的路由配置。

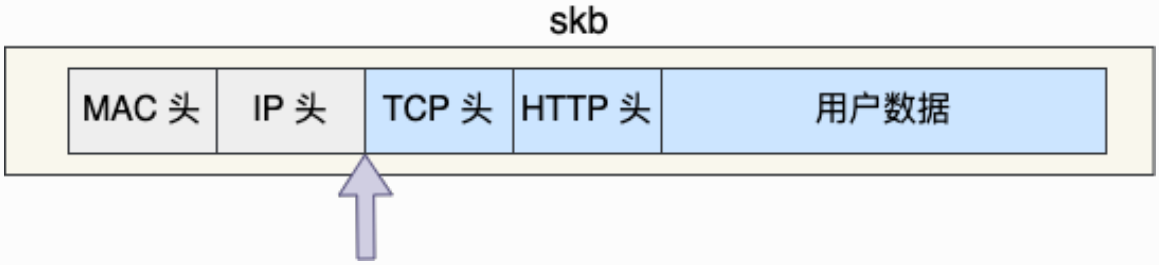
```
[(redacted) ~]# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
10.0.0.0        10.(redacted)   255.0.0.0       UG    0      0        0 eth0
10.(redacted)   0.0.0.0         255.255.248.0   U     0      0        0 eth0
169(redacted)   0.0.0.0         255.255.0.0     U     1002   0        0 eth0
```

在路由表中，可以查到某个目的网络应该通过哪个 Iface（网卡），哪个 Gateway（网卡）发送出去。查找出来以后缓存到 socket 上，下次再发送数据就不用查了。

接着把路由表地址也放到 skb 里去。

```
//file: include/linux/skbuff.h
struct sk_buff {
  //保存了一些路由相关信息
  unsigned long    _skb_refdst;
}
```

接下来就是定位到 skb 里的 IP 头的位置上，然后开始按照协议规范设置 IP header。



再通过 ip_local_out 进入到下一步的处理。

```
//file: net/ipv4/ip_output.c
int ip_local_out(struct sk_buff *skb)
{
  //执行 netfilter 过滤
  err = __ip_local_out(skb);

  //开始发送数据
  if (likely(err == 1))
    err = dst_output(skb);
  ......
```

在 ip_local_out => __ip_local_out => nf_hook 会执行 netfilter 过滤。如果你使用 iptables 配置了一些规则，那么这里将检测是否命中规则。
如果你设置了非常复杂的 netfilter 规则，在这个函数这里将会导致你的进程 CPU 开销会极大增加。

还是不多展开说，继续只聊和发送有关的过程 dst_output。

```
//file: include/net/dst.h
static inline int dst_output(struct sk_buff *skb)
{
  return skb_dst(skb)->output(skb);
}
```

此函数找到到这个 skb 的路由表（dst 条目），然后调用路由表的 output 方法。这又是一个函数指针，指向的是 ip_output 方法。

```
//file: net/ipv4/ip_output.c
int ip_output(struct sk_buff *skb)
{
  //统计
  .....

  //再次交给 netfilter，完毕后回调 ip_finish_output
  return NF_HOOK_COND(NFPROTO_IPV4, NF_INET_POST_ROUTING, skb,
NULL, dev,
        ip_finish_output,
        !(IPCB(skb)->flags & IPSKB_REROUTED));
}
```

在 ip_output 中进行一些简单的，统计工作，再次执行 netfilter 过滤。过滤通过之后回调 ip_finish_output。

```
//file: net/ipv4/ip_output.c
static int ip_finish_output(struct sk_buff *skb)
{
  //大于 mtu 的话就要进行分片了
  if (skb->len > ip_skb_dst_mtu(skb) && !skb_is_gso(skb))
    return ip_fragment(skb, ip_finish_output2);
  else
    return ip_finish_output2(skb);
}
```

在 ip_finish_output 中我们看到，**如果数据大于 MTU 的话，是会执行分片的。**

> 实际 MTU 大小确定依赖 MTU 发现，以太网帧为 1500 字节。之前 QQ 团队在早期的时候，会尽量控制自己数据包尺寸小于 MTU，通过这种方式来优化网络性能。因为分片会带来两个问题：1、需要进行额外的切分处理，有额外性能开销。2、只要一个分片丢失，整个包都得重传。所以避免分片既杜绝了分片开销，也大大降低了重传率。

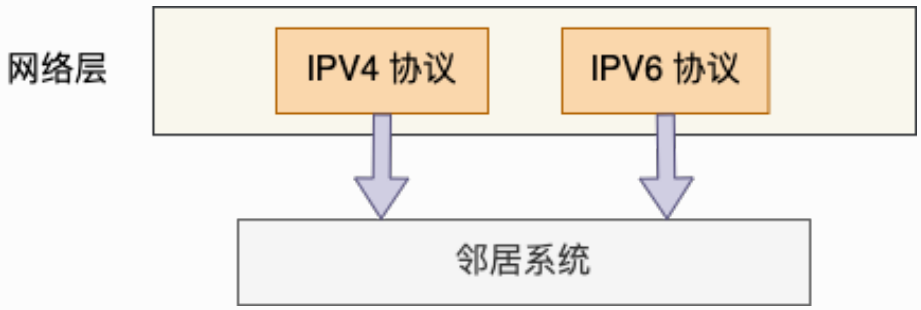
在 ip_finish_output2 中，终于发送过程会进入到下一层，邻居子系统中。

```
//file: net/ipv4/ip_output.c
static inline int ip_finish_output2(struct sk_buff *skb)
{
  //根据下一跳 IP 地址查找邻居项，找不到就创建一个
  nexthop = (__force u32) rt_nexthop(rt, ip_hdr(skb)->daddr);
  neigh = __ipv4_neigh_lookup_noref(dev, nexthop);
  if (unlikely(!neigh))
    neigh = __neigh_create(&arp_tbl, &nexthop, dev, false);

  //继续向下层传递
  int res = dst_neigh_output(dst, neigh, skb);
}
```

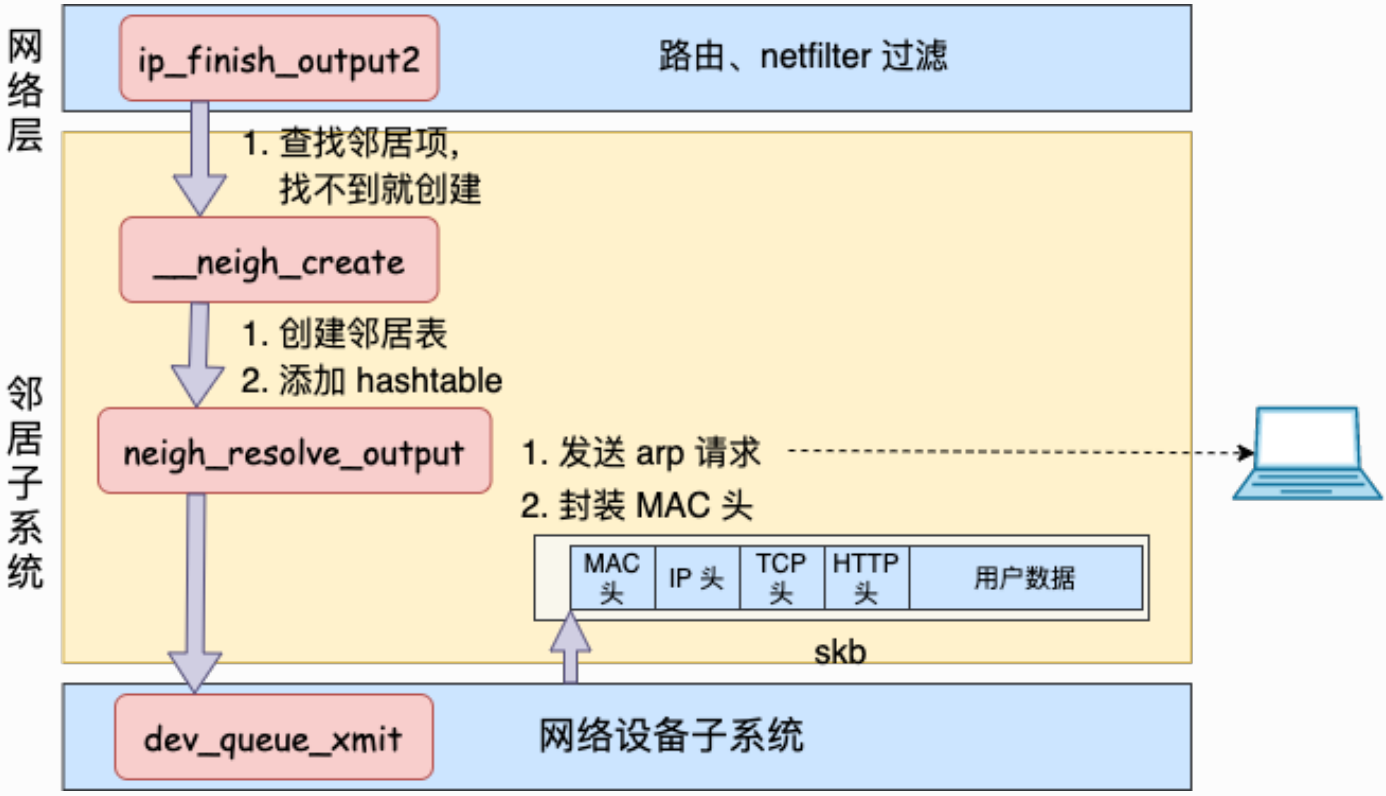
3.4.4 邻居子系统

邻居子系统是位于网络层和数据链路层中间的一个系统，其作用是对网络层提供一个封装，让网络层不必关心下层的地址信息，让下层来决定发送到哪个 MAC 地址。

而且这个邻居子系统并不位于协议栈 net/ipv4/ 目录内，而是位于 net/core/neighbour.c。因为无论是对于 IPv4 还是 IPv6 ，都需要使用该模块。



在邻居子系统里主要是查找或者创建邻居项，在创造邻居项的时候，有可能会发出实际的 arp 请求。然后封装一下 MAC 头，将发送过程再传递到更下层的网络设备子系统。大致流程如图。



理解了大致流程，我们再回头看源码。在上面小节 ip_finish_output2 源码中调用了 __ipv4_neigh_lookup_noref。它是在 arp 缓存中进行查找，其第二个参数传入的是路由下一跳 IP 信息。

```
//file: include/net/arp.h
extern struct neigh_table arp_tbl;
static inline struct neighbour *__ipv4_neigh_lookup_noref(
  struct net_device *dev, u32 key)
{
  struct neigh_hash_table *nht =
rcu_dereference_bh(arp_tbl.nht);

  //计算 hash 值，加速查找
  hash_val = arp_hashfn(......);
  for (n = rcu_dereference_bh(nht->hash_buckets[hash_val]);
     n != NULL;
     n = rcu_dereference_bh(n->next)) {
    if (n->dev == dev && *(u32 *)n->primary_key == key)
      return n;
  }
}
```

如果查找不到，则调用 __neigh_create 创建一个邻居。

```
//file: net/core/neighbour.c
struct neighbour *__neigh_create(......)
{
  //申请邻居表项
  struct neighbour *n1, *rc, *n = neigh_alloc(tbl, dev);

  //构造赋值
  memcpy(n->primary_key, pkey, key_len);
  n->dev = dev;
  n->parms->neigh_setup(n);

  //最后添加到邻居 hashtable 中
  rcu_assign_pointer(nht->hash_buckets[hash_val], n);
  ......
```

有了邻居项以后，此时仍然还不具备发送 IP 报文的能力，因为目的 MAC 地址还未获取。调用 dst_neigh_output 继续传递 skb。

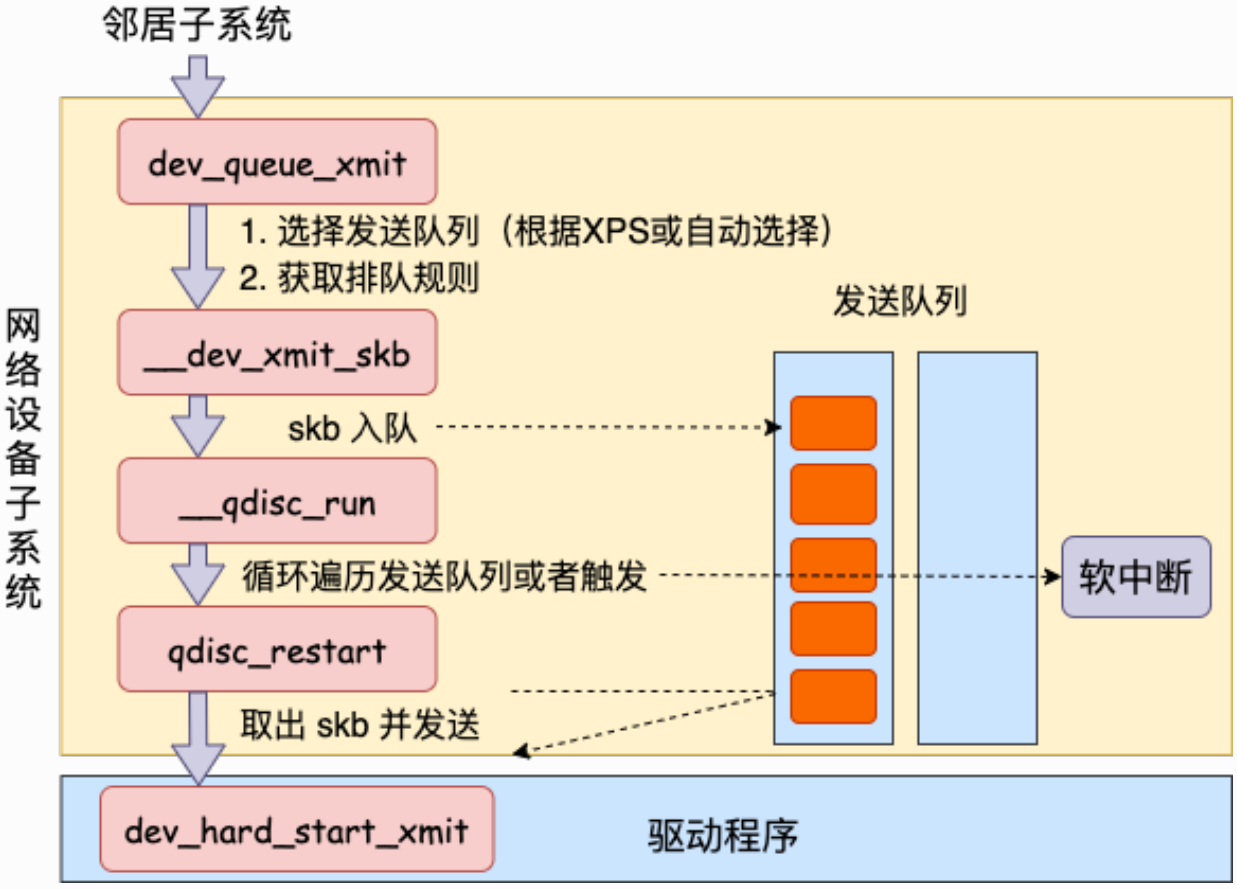
```
//file: include/net/dst.h
static inline int dst_neigh_output(struct dst_entry *dst,
       struct neighbour *n, struct sk_buff *skb)
{
  ......
  return n->output(n, skb);
}
```

调用 output，实际指向的是 neigh_resolve_output。在这个函数内部有可能会发出 arp 网络请求。

```
//file: net/core/neighbour.c
int neigh_resolve_output(){

  //注意：这里可能会触发 arp 请求
  if (!neigh_event_send(neigh, skb)) {

    //neigh->ha 是 MAC 地址
    dev_hard_header(skb, dev, ntohs(skb->protocol),
                neigh->ha, NULL, skb->len);
    //发送
    dev_queue_xmit(skb);
  }
}
```

当获取到硬件 MAC 地址以后，就可以封装 skb 的 MAC 头了。最后调用 dev_queue_xmit 将 skb 传递给 Linux 网络设备子系统。

3.4.5 网络设备子系统



邻居子系统通过 dev_queue_xmit 进入到网络设备子系统中来。

```
//file: net/core/dev.c
int dev_queue_xmit(struct sk_buff *skb)
{
  //选择发送队列
  txq = netdev_pick_tx(dev, skb);

  //获取与此队列关联的排队规则
  q = rcu_dereference_bh(txq->qdisc);

  //如果有队列，则调用__dev_xmit_skb 继续处理数据
  if (q->enqueue) {
    rc = __dev_xmit_skb(skb, q, dev, txq);
    goto out;
  }

  //没有队列的是回环设备和隧道设备
  ......
}
```

开篇第二节网卡启动准备里我们说过，网卡是有多个发送队列的（尤其是现在的网卡）。上面对 netdev_pick_tx 函数的调用就是选择一个队列进行发送。

netdev_pick_tx 发送队列的选择受 XPS 等配置的影响，而且还有缓存，也是一套小复杂的逻辑。这里我们只关注两个逻辑，首先会获取用户的 XPS 配置，否则就自动计算了。代码见 netdev_pick_tx => __netdev_pick_tx。

```
//file: net/core/flow_dissector.c
u16 __netdev_pick_tx(struct net_device *dev, struct sk_buff
*skb)
{
  //获取 XPS 配置
  int new_index = get_xps_queue(dev, skb);

  //自动计算队列
  if (new_index < 0)
    new_index = skb_tx_hash(dev, skb);}
```

然后获取与此队列关联的 qdisc。在 linux 上通过 tc 命令可以看到 qdisc 类型，例如对于我的某台多队列网卡机器上是 mq disc。

```
#tc qdisc
qdisc mq 0: dev eth0 root
```

大部分的设备都有队列（回环设备和隧道设备除外），所以现在我们进入到 __dev_xmit_skb。

```
//file: net/core/dev.c
static inline int __dev_xmit_skb(struct sk_buff *skb, struct
Qdisc *q,
         struct net_device *dev,
         struct netdev_queue *txq)
{
  //1.如果可以绕开排队系统
  if ((q->flags & TCQ_F_CAN_BYPASS) && !qdisc_qlen(q) &&
       qdisc_run_begin(q)) {
    ......
  }

  //2.正常排队
  else {

    //入队
    q->enqueue(skb, q)

    //开始发送
    __qdisc_run(q);
  }
}
```

上述代码中分两种情况，1 是可以 bypass（绕过）排队系统的，另外一种是正常排队。我们只看第二种情况。

先调用 q->enqueue 把 skb 添加到队列里。然后调用 __qdisc_run 开始发送。

```
//file: net/sched/sch_generic.c
void __qdisc_run(struct Qdisc *q)
{
  int quota = weight_p;

  //循环从队列取出一个 skb 并发送
```

```
    while (qdisc_restart(q)) {

      // 如果发生下面情况之一，则延后处理：
      // 1. quota 用尽
      // 2. 其他进程需要 CPU
      if (--quota <= 0 || need_resched()) {
        //将触发一次 NET_TX_SOFTIRQ 类型 softirq
        __netif_schedule(q);
        break;
      }
    }
  }
```

在上述代码中，我们看到 while 循环不断地从队列中取出 skb 并进行发送。注意，这个时候其实都占用的是用户进程的系统态时间(sy)。 只有当 quota 用尽或者其它进程需要 CPU 的时候才触发软中断进行发送。

所以这就是为什么一般服务器上查看 /proc/softirqs，一般 NET_RX 都要比 NET_TX 大的多的第二个原因。对于读来说，都是要经过 NET_RX 软中断，而对于发送来说，只有系统态配额用尽才让软中断上。

我们来把精力在放到 qdisc_restart 上，继续看发送过程。

```
static inline int qdisc_restart(struct Qdisc *q)
{
  //从 qdisc 中取出要发送的 skb
  skb = dequeue_skb(q);
  ...

  return sch_direct_xmit(skb, q, dev, txq, root_lock);
}
```

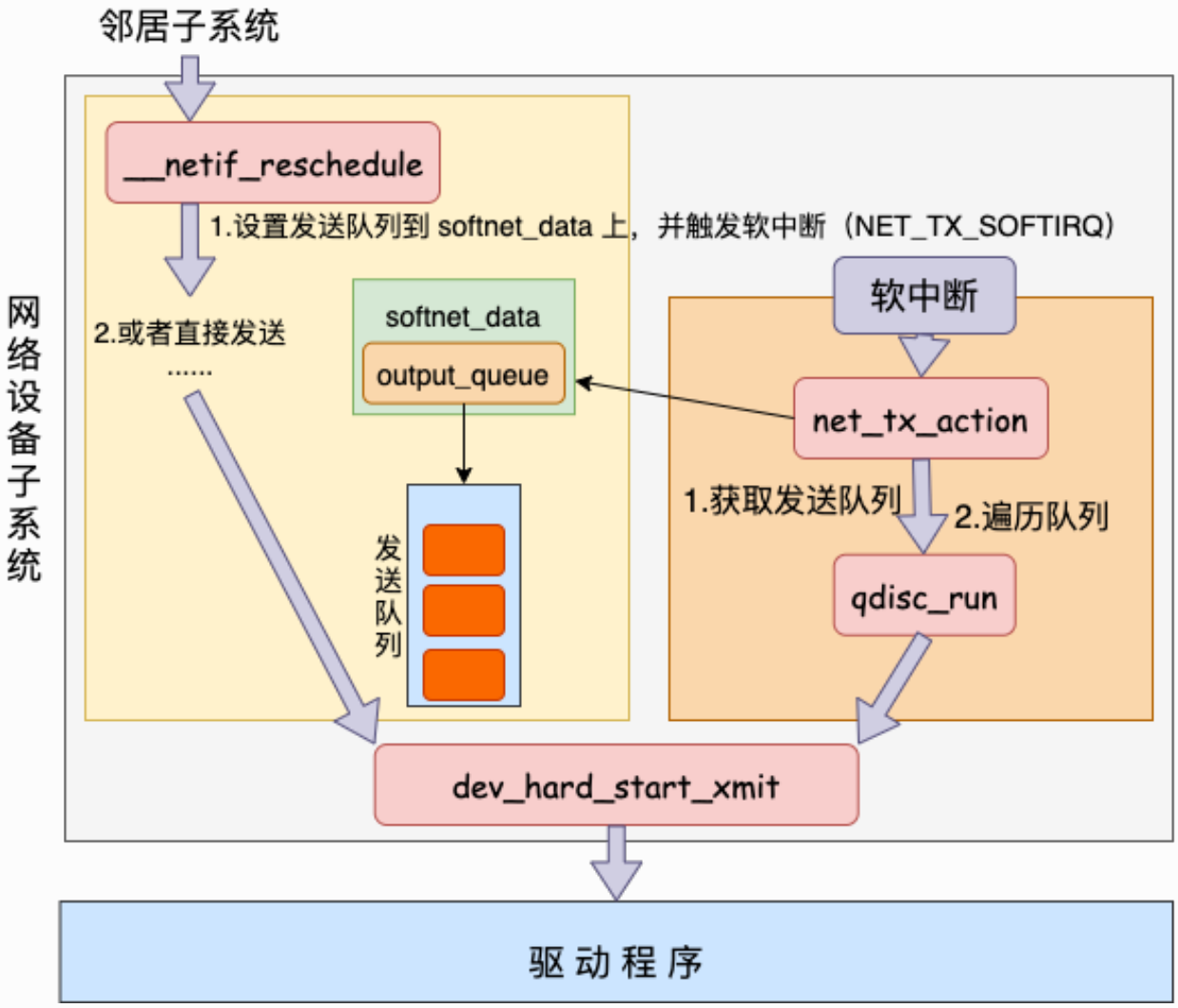
qdisc_restart 从队列中取出一个 skb，并调用 sch_direct_xmit 继续发送。

```
//file: net/sched/sch_generic.c
int sch_direct_xmit(struct sk_buff *skb, struct Qdisc *q,
       struct net_device *dev, struct netdev_queue *txq,
       spinlock_t *root_lock)
{
  //调用驱动程序来发送数据
  ret = dev_hard_start_xmit(skb, dev, txq);
}
```

3.4.6 软中断调度

在 4.5 咱们看到了如果系统态 CPU 发送网络包不够用的时候，会调用 __netif_schedule 触发一个软中断。该函数会进入到 __netif_reschedule，由它来实际发出 NET_TX_SOFTIRQ 类型软中断。

软中断是由内核线程来运行的，该线程会进入到 net_tx_action 函数，在该函数中能获取到发送队列，并也最终调用到驱动程序里的入口函数 dev_hard_start_xmit。



```
//file: net/core/dev.c
static inline void __netif_reschedule(struct Qdisc *q)
{
  sd = &__get_cpu_var(softnet_data);
  q->next_sched = NULL;
  *sd->output_queue_tailp = q;
  sd->output_queue_tailp = &q->next_sched;

  ......
  raise_softirq_irqoff(NET_TX_SOFTIRQ);
}
```

在该函数里在软中断能访问到的 softnet_data 里设置了要发送的数据队列，添加到了 output_queue 里了。紧接着触发了 NET_TX_SOFTIRQ 类型的软中断。（T 代表 transmit 传输）

软中断的入口代码我这里也不详细扒了，感兴趣的同学参考《图解Linux网络包接收过程》一文中的 3.2 小节 - ksoftirqd内核线程处理软中断。

我们直接从 NET_TX_SOFTIRQ softirq 注册的回调函数 net_tx_action讲起。用户态进程触发完软中断之后，会有一个软中断内核线程会执行到 net_tx_action。

牢记，这以后发送数据消耗的 CPU 就都显示在 si 这里了，不会消耗用户进程的系统时间了。

```
//file: net/core/dev.c
static void net_tx_action(struct softirq_action *h)
{
  //通过 softnet_data 获取发送队列
  struct softnet_data *sd = &__get_cpu_var(softnet_data);

  // 如果 output queue 上有 qdisc
  if (sd->output_queue) {

    // 将 head 指向第一个 qdisc
    head = sd->output_queue;

    //遍历 qdsics 列表
    while (head) {
      struct Qdisc *q = head;
      head = head->next_sched;
```

```
        //发送数据
        qdisc_run(q);
      }
    }
  }
```

软中断这里会获取 softnet_data。前面我们看到进程内核态在调用 __netif_reschedule 的时候把发送队列写到 softnet_data 的 output_queue 里了。 软中断循环遍历 sd->output_queue 发送数据帧。

来看 qdisc_run，它和进程用户态一样，也会调用到 __qdisc_run。

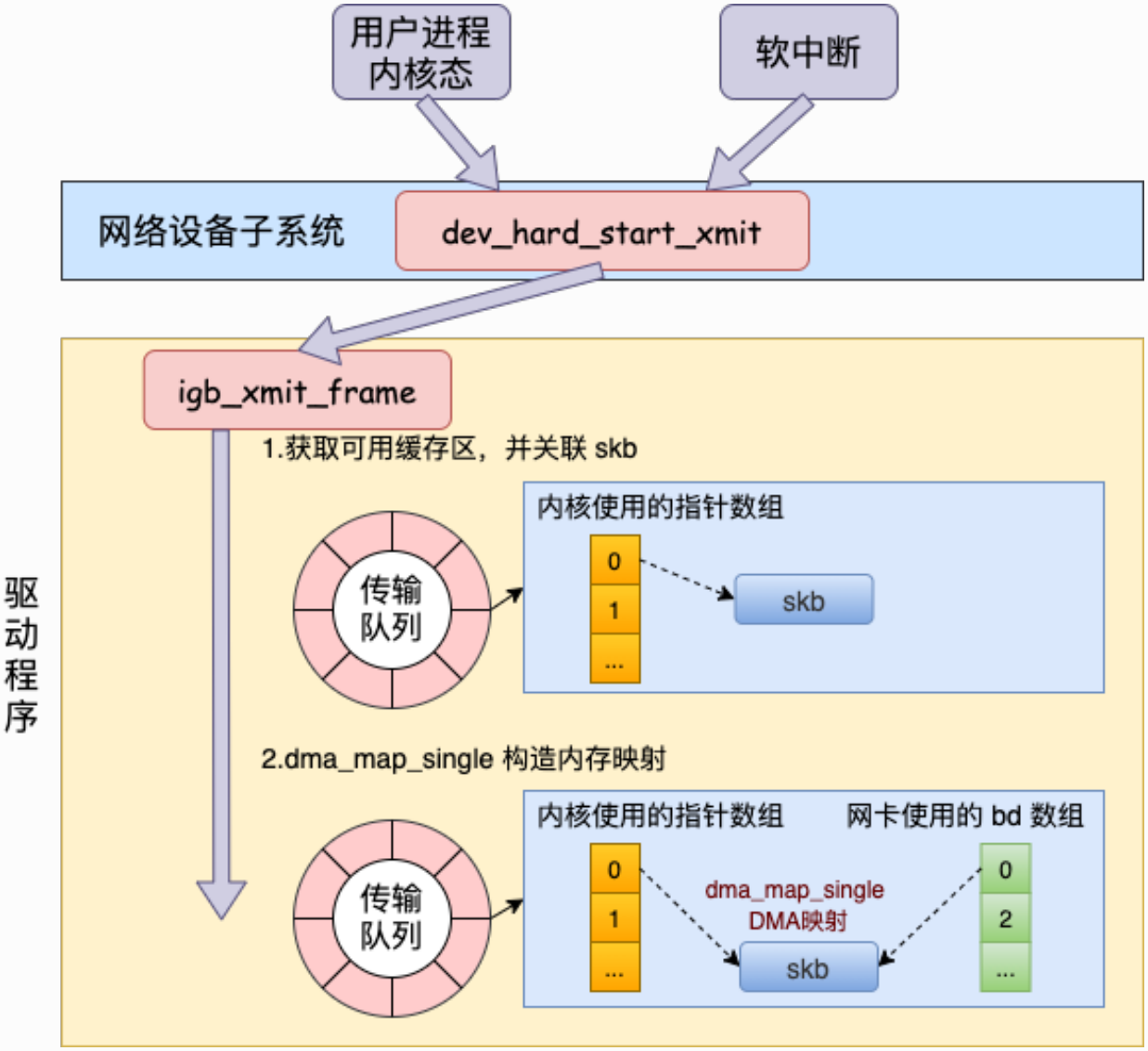
```
//file: include/net/pkt_sched.h
static inline void qdisc_run(struct Qdisc *q)
{
  if (qdisc_run_begin(q))
    __qdisc_run(q);
}
```

然后一样就是进入 qdisc_restart => sch_direct_xmit，直到驱动程序函数 dev_hard_start_xmit。

3.4.7 igb 网卡驱动发送

我们前面看到，无论是对于用户进程的内核态，还是对于软中断上下文，都会调用到网络设备子系统中的 dev_hard_start_xmit 函数。在这个函数中，会调用到驱动里的发送函数 igb_xmit_frame。

在驱动函数里，将 skb 会挂到 RingBuffer上，驱动调用完毕后，数据包将真正从网卡发送出去。



我们来看看实际的源码：

```
//file: net/core/dev.c
int dev_hard_start_xmit(struct sk_buff *skb, struct net_device
*dev,
      struct netdev_queue *txq)
{
  //获取设备的回调函数集合 ops
  const struct net_device_ops *ops = dev->netdev_ops;

  //获取设备支持的功能列表
  features = netif_skb_features(skb);

  //调用驱动的 ops 里面的发送回调函数 ndo_start_xmit 将数据包传给网卡设
备
  skb_len = skb->len;
  rc = ops->ndo_start_xmit(skb, dev);
```

```
}
```

其中 ndo_start_xmit 是网卡驱动要实现的一个函数，是在 net_device_ops 中定义的。

```
//file: include/linux/netdevice.h
struct net_device_ops {
  netdev_tx_t   (*ndo_start_xmit) (struct sk_buff *skb,
                struct net_device *dev);

}
```

在 igb 网卡驱动源码中，我们找到了。

```
//file: drivers/net/ethernet/intel/igb/igb_main.c
static const struct net_device_ops igb_netdev_ops = {
  .ndo_open    = igb_open,
  .ndo_stop    = igb_close,
  .ndo_start_xmit    = igb_xmit_frame,
  ...
};
```

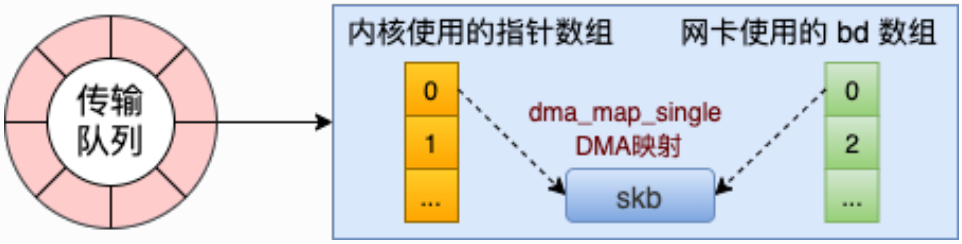
也就是说，对于网络设备层定义的 ndo_start_xmit， igb 的实现函数是 igb_xmit_frame。这个函数是在网卡驱动初始化的时候被赋值的。具体初始化过程参见《图解Linux网络包接收过程》一文中的 2.4 节，网卡驱动初始化。

所以在上面网络设备层调用 ops->ndo_start_xmit 的时候，会实际上进入 igb_xmit_frame 这个函数中。我们进入这个函数来看看驱动程序是如何工作的。

```
//file: drivers/net/ethernet/intel/igb/igb_main.c
static netdev_tx_t igb_xmit_frame(struct sk_buff *skb,
          struct net_device *netdev)
{
  ......
  return igb_xmit_frame_ring(skb, igb_tx_queue_mapping(adapter,
skb));
}

netdev_tx_t igb_xmit_frame_ring(struct sk_buff *skb,
        struct igb_ring *tx_ring)
```

```
{
  //获取TX Queue 中下一个可用缓冲区信息
  first = &tx_ring->tx_buffer_info[tx_ring->next_to_use];
  first->skb = skb;
  first->bytecount = skb->len;
  first->gso_segs = 1;

  //igb_tx_map 函数准备给设备发送的数据。
  igb_tx_map(tx_ring, first, hdr_len);
}
```

在这里从网卡的发送队列的 RingBuffer 中取下来一个元素，并将 skb 挂到元素上。



igb_tx_map 函数处理将 skb 数据映射到网卡可访问的内存 DMA 区域。

```
//file: drivers/net/ethernet/intel/igb/igb_main.c
static void igb_tx_map(struct igb_ring *tx_ring,
          struct igb_tx_buffer *first,
          const u8 hdr_len)
{
  //获取下一个可用描述符指针
  tx_desc = IGB_TX_DESC(tx_ring, i);

  //为 skb->data 构造内存映射，以允许设备通过 DMA 从 RAM 中读取数据
  dma = dma_map_single(tx_ring->dev, skb->data, size,
DMA_TO_DEVICE);

  //遍历该数据包的所有分片,为 skb 的每个分片生成有效映射
  for (frag = &skb_shinfo(skb)->frags[0];; frag++) {

    tx_desc->read.buffer_addr = cpu_to_le64(dma);
    tx_desc->read.cmd_type_len = ...;
    tx_desc->read.olinfo_status = 0;
  }

  //设置最后一个descriptor
```

```
    cmd_type |= size | IGB_TXD_DCMD;
    tx_desc->read.cmd_type_len = cpu_to_le32(cmd_type);

    /* Force memory writes to complete before letting h/w know
there
     * are new descriptors to fetch
     */
    wmb();
}
```

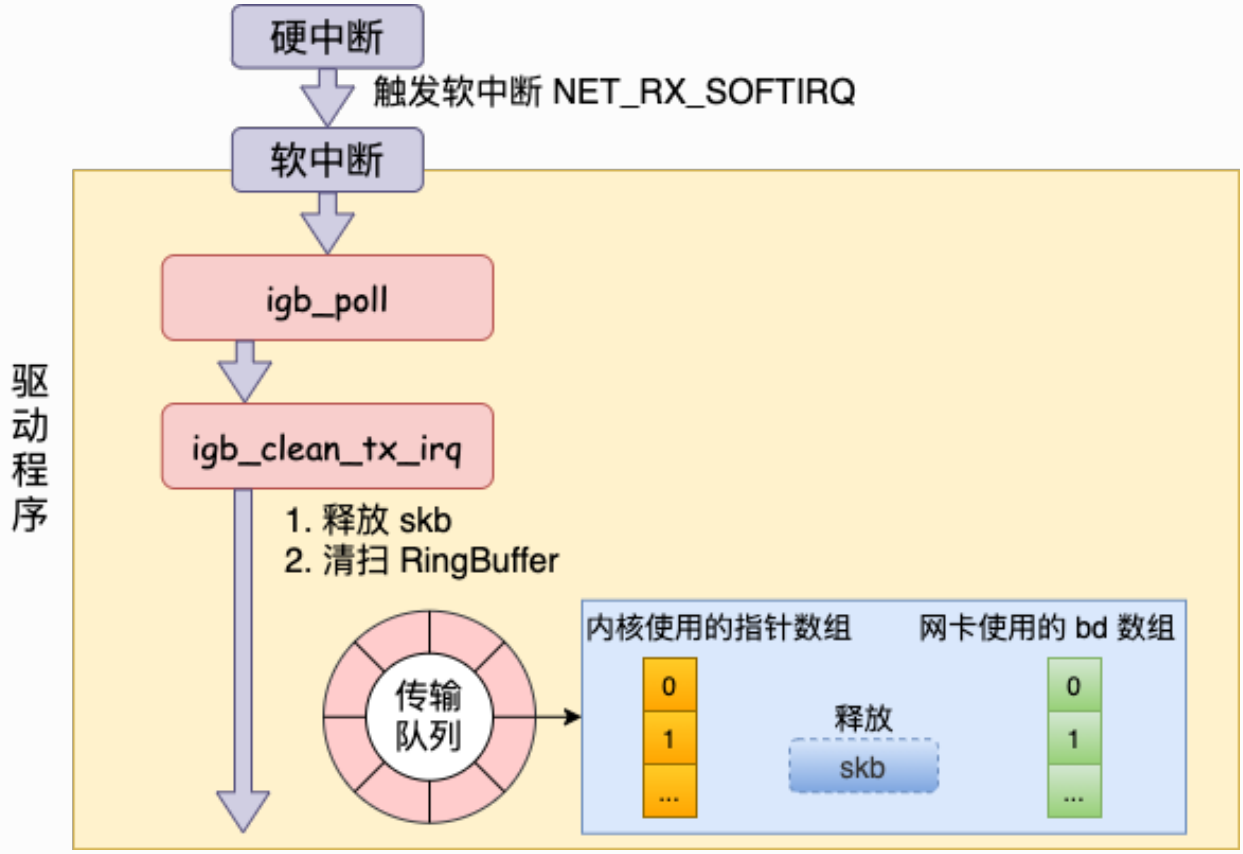
当所有需要的描述符都已建好，且 skb 的所有数据都映射到 DMA 地址后，驱动就会进入到它的最后一步，触发真实的发送。

3.4.8 发送完成硬中断

当数据发送完成以后，其实工作并没有结束。因为内存还没有清理。当发送完成的时候，网卡设备会触发一个硬中断来释放内存。

在《图解Linux网络包接收过程》一文中的 3.1 和 3.2 小节，我们详细讲述过硬中断和软中断的处理过程。

在发送硬中断里，会执行 RingBuffer 内存的清理工作，如图。



再回头看一下硬中断触发软中断的源码。

```
//file: drivers/net/ethernet/intel/igb/igb_main.c
static inline void ____napi_schedule(...){
  list_add_tail(&napi->poll_list, &sd->poll_list);
  __raise_softirq_irqoff(NET_RX_SOFTIRQ);
}
```

这里有个很有意思的细节，无论硬中断是因为是有数据要接收，还是说发送完成通知，**从硬中断触发的软中断都是 NET_RX_SOFTIRQ**。 这个我们在第一节说过了，这是软中断统计中 RX 要高于 TX 的一个原因。

好我们接着进入软中断的回调函数 igb_poll。在这个函数里，我们注意到有一行 igb_clean_tx_irq，参见源码：

```
//file: drivers/net/ethernet/intel/igb/igb_main.c
static int igb_poll(struct napi_struct *napi, int budget)
{
  //performs the transmit completion operations
  if (q_vector->tx.ring)
    clean_complete = igb_clean_tx_irq(q_vector);
  ...
}
```

我们来看看当传输完成的时候，igb_clean_tx_irq 都干啥了。

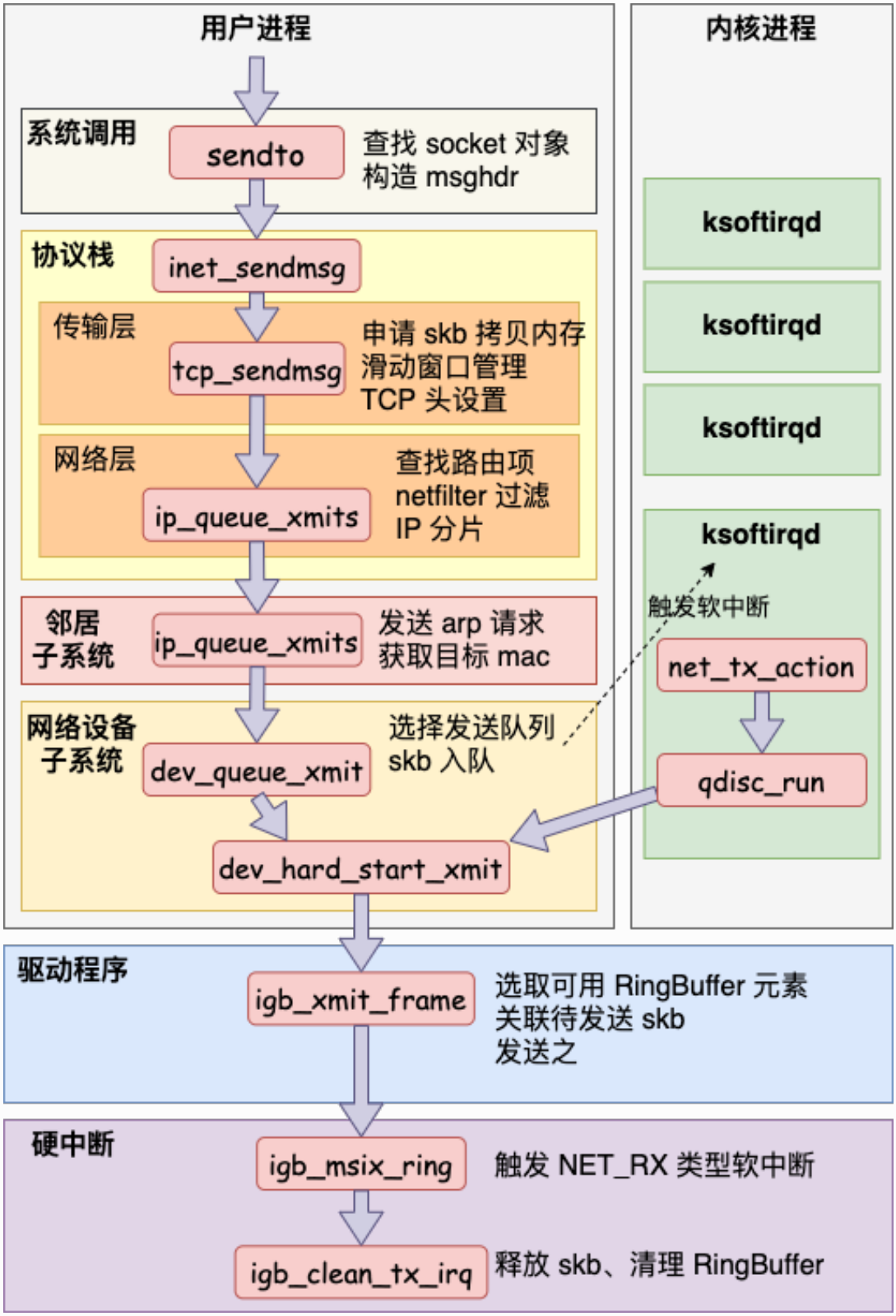
```
//file: drivers/net/ethernet/intel/igb/igb_main.c
static bool igb_clean_tx_irq(struct igb_q_vector *q_vector)
{
  //free the skb
  dev_kfree_skb_any(tx_buffer->skb);

  //clear tx_buffer data
  tx_buffer->skb = NULL;
  dma_unmap_len_set(tx_buffer, len, 0);

  // clear last DMA location and unmap remaining buffers */
  while (tx_desc != eop_desc) {
  }
}
```

无非就是清理了 skb，解除了 DMA 映射等等。 到了这一步，传输才算是基本完成了。

为啥我说是基本完成，而不是全部完成了呢？因为传输层需要保证可靠性，所以 skb 其实还没有删除。它得等收到对方的 ACK 之后才会真正删除，那个时候才算是彻底的发送完毕。

最后

用一张图总结一下整个发送过程



了解了整个发送过程以后，我们回头再来回顾开篇提到的几个问题。

1.我们在监控内核发送数据消耗的 CPU 时，是应该看 sy 还是 si？

在网络包的发送过程中，用户进程（在内核态）完成了绝大部分的工作，甚至连调用驱动的事情都干了。 只有当内核态进程被切走前才会发起软中断。 发送过程中，绝大部分（90%）以上的开销都是在用户进程内核态消耗掉的。

只有一少部分情况下才会触发软中断（NET_TX 类型），由软中断 ksoftirqd 内核进程来发送。

所以，在监控网络 IO 对服务器造成的 CPU 开销的时候，不能仅仅只看 si，而是应该把 si、sy 都考虑进来。

2. 在服务器上查看 /proc/softirqs，为什么 NET_RX 要比 NET_TX 大的多的多?

之前我认为 NET_RX 是读取，NET_TX 是传输。对于一个既收取用户请求，又给用户返回的 Server 来说。 这两块的数字应该差不多才对，至少不会有数量级的差异。但事实上，飞哥手头的一台服务器是这样的：

```
[                ]$ cat /proc/softirqs
                    CPU0       CPU1       CPU2       CPU3
          HI:           0          0          0          0
       TIMER: 4189404746 3011986206 2435264887 2544464569
      NET_TX:      343981     260256     224167     234717
      NET_RX: 1379163125 1065550662  901100884  926004272
       BLOCK:        1940          0          0          0
BLOCK_IOPOLL:           0          0          0
     TASKLET:           1          0          0          0
       SCHED: 3894836698 3286402891 2877234633 2777895189
     HRTIMER:    15575069   21099408   21018737   19124602
         RCU:   856741846 3915616388 3285649482 3389076096
```

经过今天的源码分析，发现这个问题的原因有两个。

第一个原因是当数据发送完成以后，通过硬中断的方式来通知驱动发送完毕。但是硬中断无论是有数据接收，还是对于发送完毕，触发的软中断都是 NET_RX_SOFTIRQ，而并不是 NET_TX_SOFTIRQ。

第二个原因是对于读来说，都是要经过 NET_RX 软中断的，都走 ksoftirqd 内核进程。而对于发送来说，绝大部分工作都是在用户进程内核态处理了，只有系统态配额用尽才会发出 NET_TX，让软中断上。

综上两个原因，那么在机器上查看 NET_RX 比 NET_TX 大的多就不难理解了。

3.发送网络数据的时候都涉及到哪些内存拷贝操作?

这里的内存拷贝，我们只特指待发送数据的内存拷贝。

第一次拷贝操作是内核申请完 skb 之后，这时候会将用户传递进来的 buffer 里的数据内容都拷贝到 skb 中。如果要发送的数据量比较大的话，这个拷贝操作开销还是不小的。

第二次拷贝操作是从传输层进入网络层的时候，每一个 skb 都会被克隆一个新的副本出来。网络层以及下面的驱动、软中断等组件在发送完成的时候会将这个副本删除。传输层保存着原始的 skb，在当网络对方没有 ack 的时候，还可以重新发送，以实现 TCP 中要求的可靠传输。

第三次拷贝不是必须的，只有当 IP 层发现 skb 大于 MTU 时才需要进行。会再申请额外的 skb，并将原来的 skb 拷贝为多个小的 skb。

> 这里插入个题外话，大家在网络性能优化中经常听到的零拷贝，我觉得这有点点夸张的成分。TCP 为了保证可靠性，第二次的拷贝根本就没法省。如果包再大于 MTU 的话，分片时的拷贝同样也避免不了。

看到这里，相信内核发送数据包对于你来说，已经不再是一个完全不懂的黑盒了。本文哪怕你只看懂十分之一，你也已经掌握了这个黑盒的打开方式。这在你将来优化网络性能时你就会知道从哪儿下手了。

最后，还愣着干啥，赶紧帮飞哥**赞、再看、转发**三连走起!

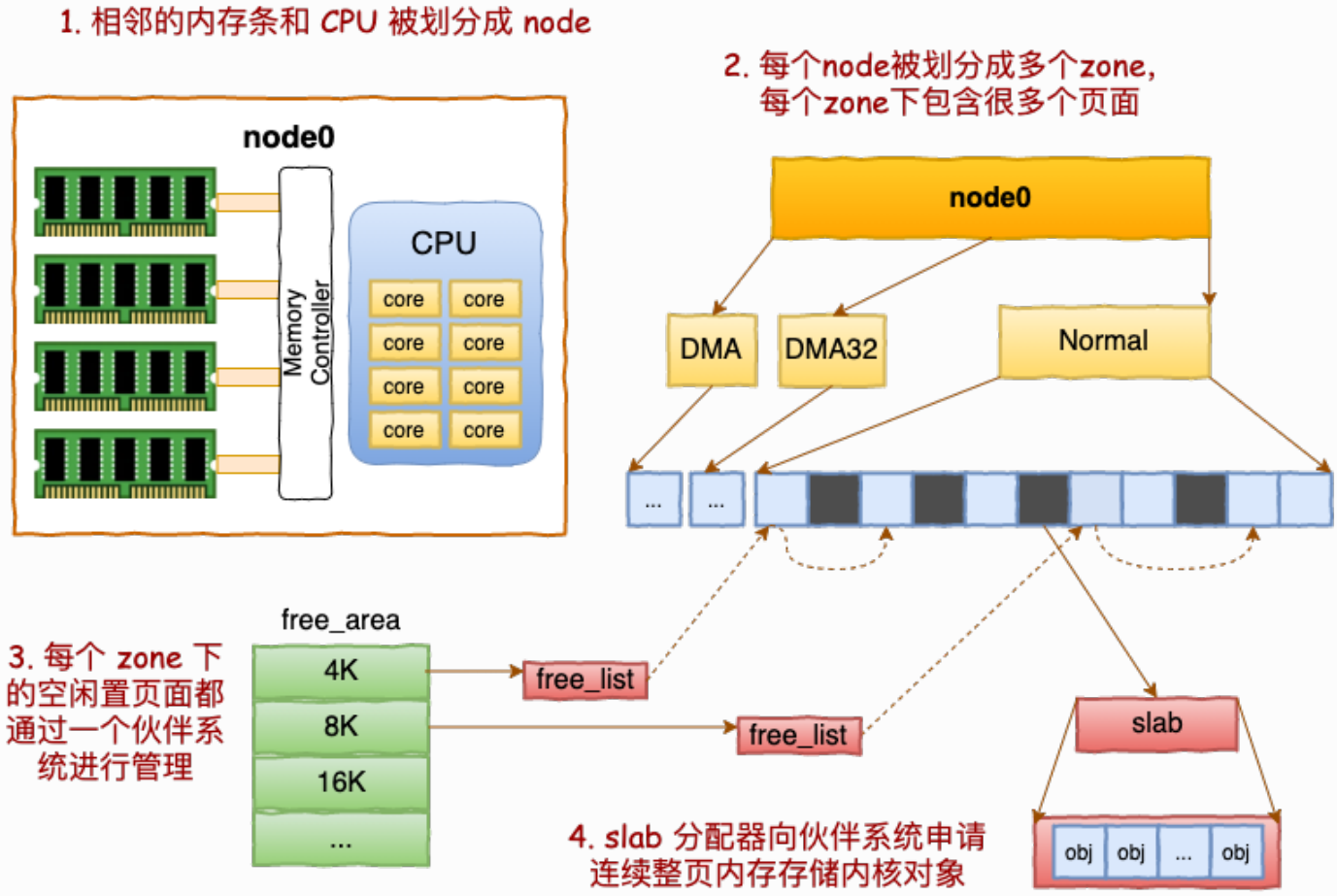
公众号

作者微信

四、TCP 连接的内存开销

4.1 内核对象是如何使用内存的

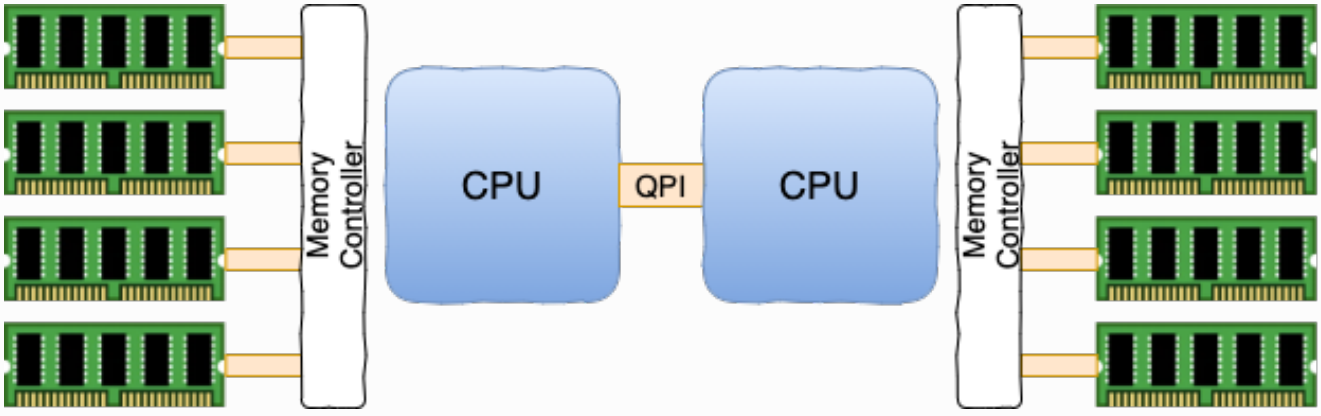

老哥，为什么我们看 tcp 内存开销的时候需要用 slabtop 等命令呢?
不同于给应用程序提供的虚拟内存机制，内核使用 slab 的分配器来申请内存。
啊，，，这么说 Linux 内核鸟悄儿的给自己开了个小灶啊
可以这么理解
所以查看内核内存开销，就要相应地使用 slab 相关的命令和工具了，比如你刚说的slabtop
那老兄可以给我多讲讲内核是怎么使用内存的吗?
木的问题。总体上来说，内核想把内存条用起来分成如下四步



现在你可能还觉得Node、zone、伙伴系统、slab这些东东还有那么一点点陌生。别怕，接下来我们结合动手观察，把它们逐个来展开细说。（下面的讨论都基于Linux 3.10.0版本）

4.1.1 NODE 划分

在现代的服务器上，内存和CPU都是所谓的NUMA架构



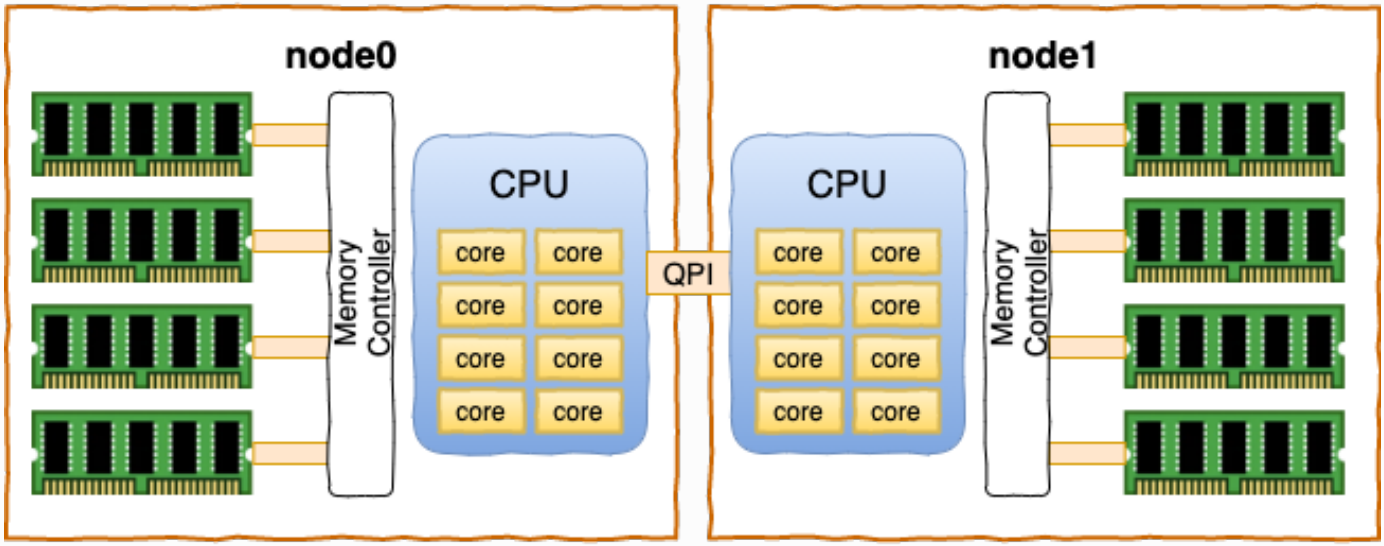
CPU往往不止是一颗。通过dmidecode命令看到你主板上插着的CPU的详细信息

```
Processor Information  //第一颗CPU
    SocketDesignation: CPU1
    Version: Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz
    Core Count: 8
    Thread Count: 16
Processor Information  //第二颗CPU
    Socket Designation: CPU2
    Version: Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz
    Core Count: 8
```

内存也不只是一条。dmidecode同样可以查看到服务器上插着的所有内存条，也可以看到它是和哪个CPU直接连接的。

```
//CPU1 上总共插着四条内存
Memory Device
    Size: 16384 MB
    Locator: CPU1 DIMM A1
Memory Device
    Size: 16384 MB
    Locator: CPU1 DIMM A2
......
//CPU2 上也插着四条
Memory Device
    Size: 16384 MB
    Locator: CPU2 DIMM E1
Memory Device
    Size: 16384 MB
    Locator: CPU2 DIMM F1
......
```

每一个CPU以及和他直连的内存条组成了一个 **node（节点）**。

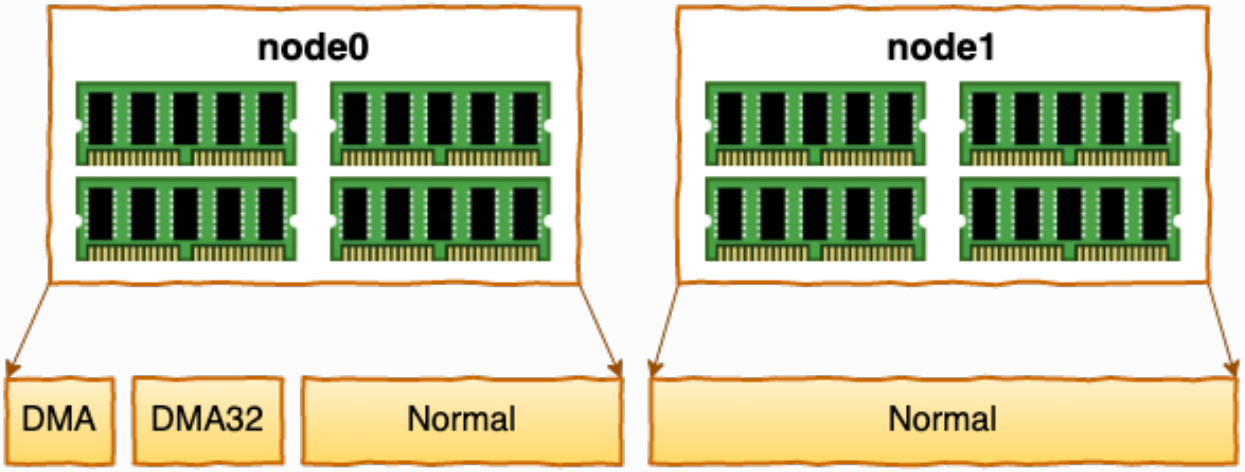


在你的机器上，你可以使用numactl你可以看到每个node的情况

```
numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 16 17 18 19 20 21 22 23
node 0 size: 65419 MB
node 1 cpus: 8 9 10 11 12 13 14 15 24 25 26 27 28 29 30 31
node 1 size: 65536 MB
```

4.1.2 ZONE 划分

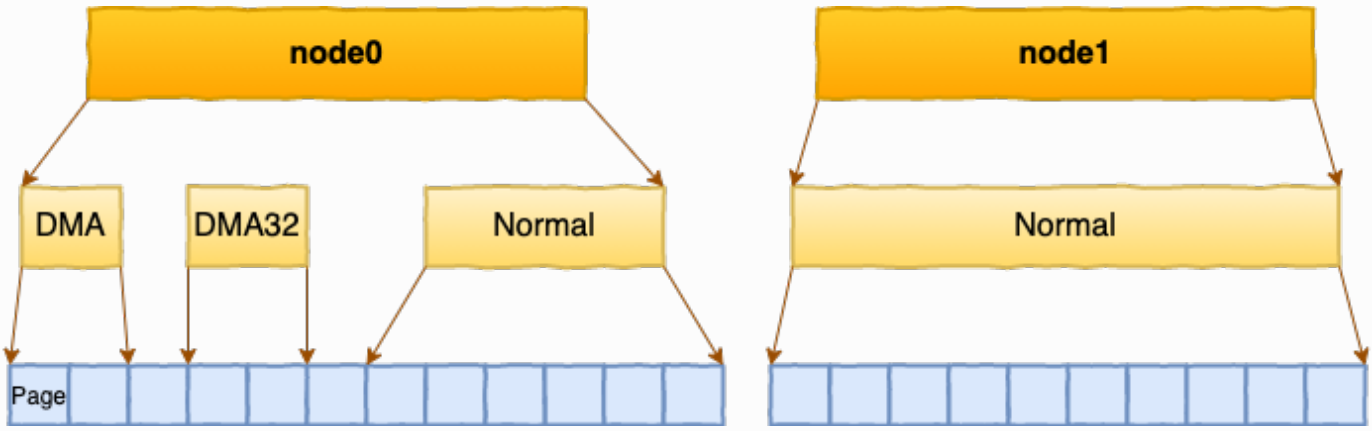
每个 node 又会划分成若干的 **zone（区域）** 。zone 表示内存中的一块范围



- ZONE_DMA：地址段最低的一块内存区域，ISA(Industry Standard Architecture)设备DMA访问
- ZONE_DMA32：该Zone用于支持32-bits地址总线的DMA设备，只在64-bits系统里才有效
- ZONE_NORMAL：在X86-64架构下，DMA和DMA32之外的内存全部在NORMAL的Zone里管理

> 为什么没有提 ZONE_HIGHMEM 这个zone？ 因为这是 32 位机时代的产物。现在应该没谁在用这种古董了吧。

在每个zone下，都包含了许许多多个 Page（页面）， 在linux下一个Page的大小一般是 4 KB。



在你的机器上，你可以使用通过 zoneinfo 查看到你机器上 zone 的划分，也可以看到每个 zone 下所管理的页面有多少个。

```
# cat /proc/zoneinfo
Node 0, zone      DMA
    pages free     3973
        managed  3973
Node 0, zone    DMA32
    pages free     390390
        managed  427659
Node 0, zone   Normal
    pages free     15021616
        managed  15990165
Node 1, zone   Normal
    pages free     16012823
        managed  16514393
```

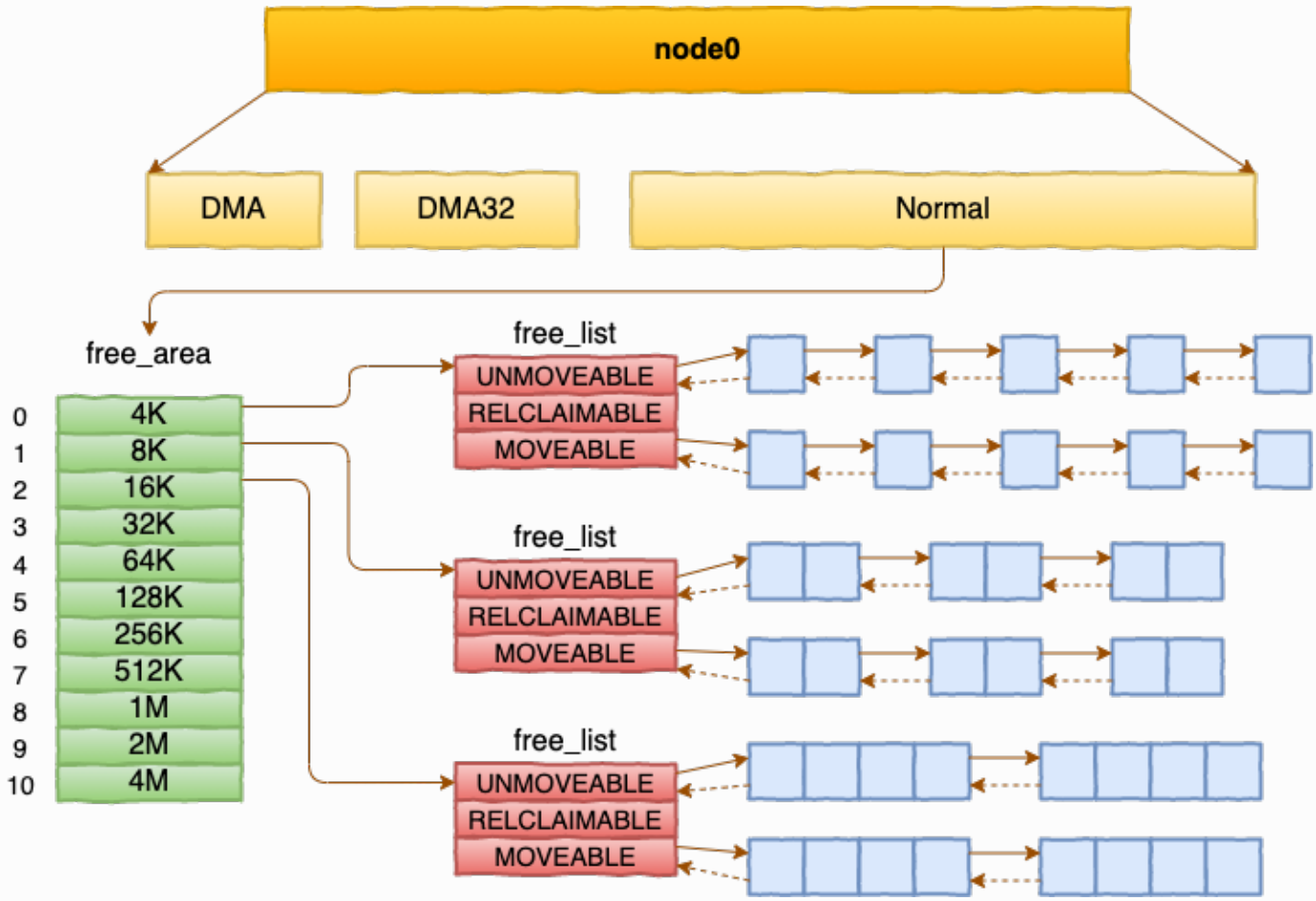
每个页面大小是4K，很容易可以计算出每个 zone 的大小。比如对于上面 Node1 的 Normal， 16514393 * 4K = 66 GB。

4.1.3 基于伙伴系统管理空闲页面

每个 zone 下面都有如此之多的页面，Linux使用**伙伴系统**对这些页面进行高效的管理。 在内核中，表示 zone 的数据结构是 `struct zone` 。 其下面的一个数组 free_area 管理了绝大部分可用的空闲页面。这个数组就是**伙伴系统**实现的重要数据结构。

```
//file: include/linux/mmzone.h
#define MAX_ORDER 11
struct zone {
    free_area   free_area[MAX_ORDER];
    ......
}
```

free_area是一个11个元素的数组，在每一个数组分别代表的是空闲可分配连续4K、8K、16K、.....、4M内存链表。



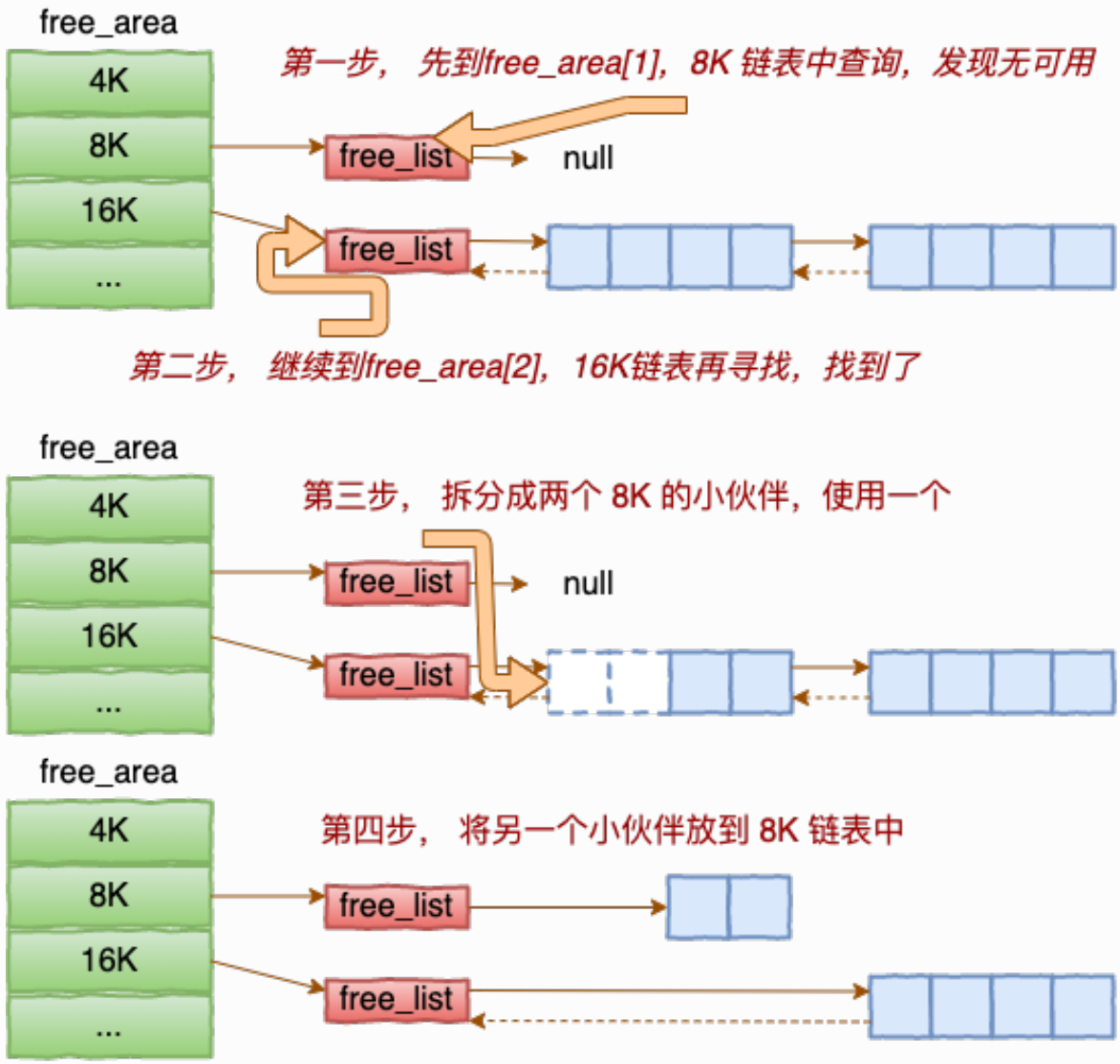
通过 `cat /proc/pagetypeinfo`，你可以看到当前系统里伙伴系统里各个尺寸的可用连续内存块数量。

```
Free pages count per migrate type at order       0      1      2      3      4      5      6      7      8      9     10
Node    0, zone      DMA, type    Unmovable      1      0      1      0      2      1      1      0      1      0      0
Node    0, zone      DMA, type  Reclaimable      0      0      0      0      0      0      0      0      0      0      0
Node    0, zone      DMA, type      Movable      0      0      0      0      0      0      0      0      0      0      3
Node    0, zone      DMA, type      Reserve      0      0      0      0      0      0      0      0      0      1      0
Node    0, zone      DMA, type          CMA      0      0      0      0      0      0      0      0      0      0      0
Node    0, zone      DMA, type      Isolate      0      0      0      0      0      0      0      0      0      0      0
Node    0, zone    DMA32, type    Unmovable    153    126     44     11     15     18     15     18     13     10     64
Node    0, zone    DMA32, type  Reclaimable      1      0     48     61     67     43     30     10      8      4      1
Node    0, zone    DMA32, type      Movable    250   1478    617    174    100     48     14      3      1      1    280
Node    0, zone    DMA32, type      Reserve      0      0      0      0      0      0      0      0      0      0      1
Node    0, zone    DMA32, type          CMA      0      0      0      0      0      0      0      0      0      0      0
Node    0, zone    DMA32, type      Isolate      0      0      0      0      0      0      0      0      0      0      0
Node    0, zone   Normal, type    Unmovable    688    193    563    772    673    549    474    434    377    305   1459
Node    0, zone   Normal, type  Reclaimable    592    869   1499    926    981    674    455    281    165     80     20
Node    0, zone   Normal, type      Movable      0  24481  12880   3555   1378    950    385    131     76     51  12352
Node    0, zone   Normal, type      Reserve      0      0      0      0      0      0      0      0      0      0      1
Node    0, zone   Normal, type          CMA      0      0      0      0      0      0      0      0      0      0      0
Node    0, zone   Normal, type      Isolate      0      0      0      0      0      0      0      0      0      0      0
```

内核提供分配器函数 `alloc_pages` 到上面的多个链表中寻找可用连续页面。

```
struct page * alloc_pages(gfp_t gfp_mask, unsigned int order)
```

`alloc_pages` 是怎么工作的呢？我们举个简单的小例子。假如要申请8K-连续两个页框的内存。为了描述方便，我们先暂时忽略UNMOVEABLE、RELCLAIMABLE等不同类型



> 伙伴系统中的伙伴指的是两个内存块，大小相同，地址连续，同属于一个大块区域。

基于伙伴系统的内存分配中，有可能需要将大块内存拆分成两个小伙伴。在释放中，可能会将两个小伙伴合并再次组成更大块的连续内存。

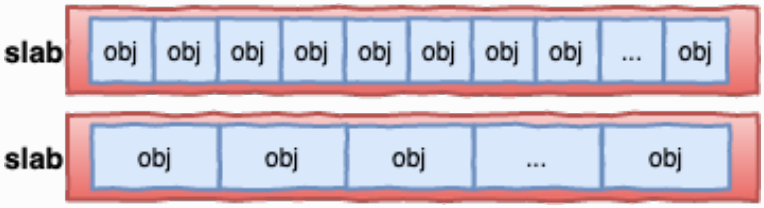
2.1.4 SLAB 管理器

说到现在，不知道你注意到没有。目前我们介绍的内存分配都是**以页面（4KB）为单位的**。

对于各个内核运行中实际使用的对象来说，多大的对象都有。有的对象有1K多，但有的对象只有几百、甚至几十个字节。如果都直接分配一个 4K的页面 来存储的话也太败家了，所以伙伴系统并不能直接使用。

在伙伴系统之上，**内核又给自己搞了一个专用的内存分配器， 叫slab或slub**。 这两个词老混用，为了省事，接下来我们就统一叫 slab 吧。

这个分配器最大的特点就是，一个slab内只分配特定大小、甚至是特定的对象。这样当一个对象释放内存后，另一个同类对象可以直接使用这块内存。通过这种办法极大地降低了碎片发生的几率。

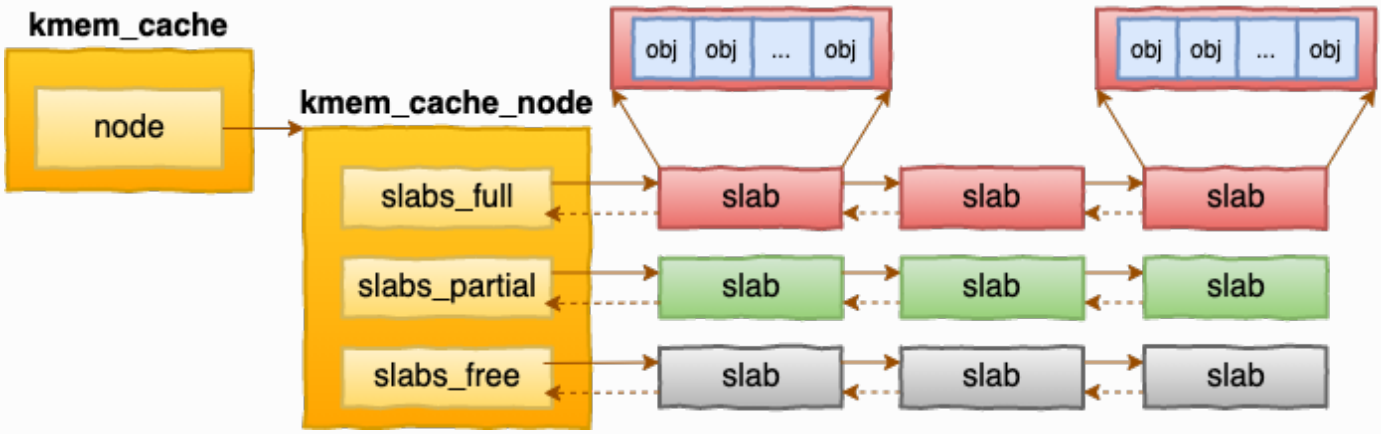


slab相关的内核对象定义如下：

```
//file: include/linux/slab_def.h
struct kmem_cache {
    struct kmem_cache_node **node
    ......
}

//file: mm/slab.h
struct kmem_cache_node {
    struct list_head slabs_partial;
    struct list_head slabs_full;
    struct list_head slabs_free;
    ......
}
```

每个cache都有满、半满、空三个链表。 每个链表节点都对应一个 slab，一个 slab 由 1 个或者多个内存页组成。

在每一个 slab 内都保存的是同等大小的对象。 一个cache的组成示意图如下：

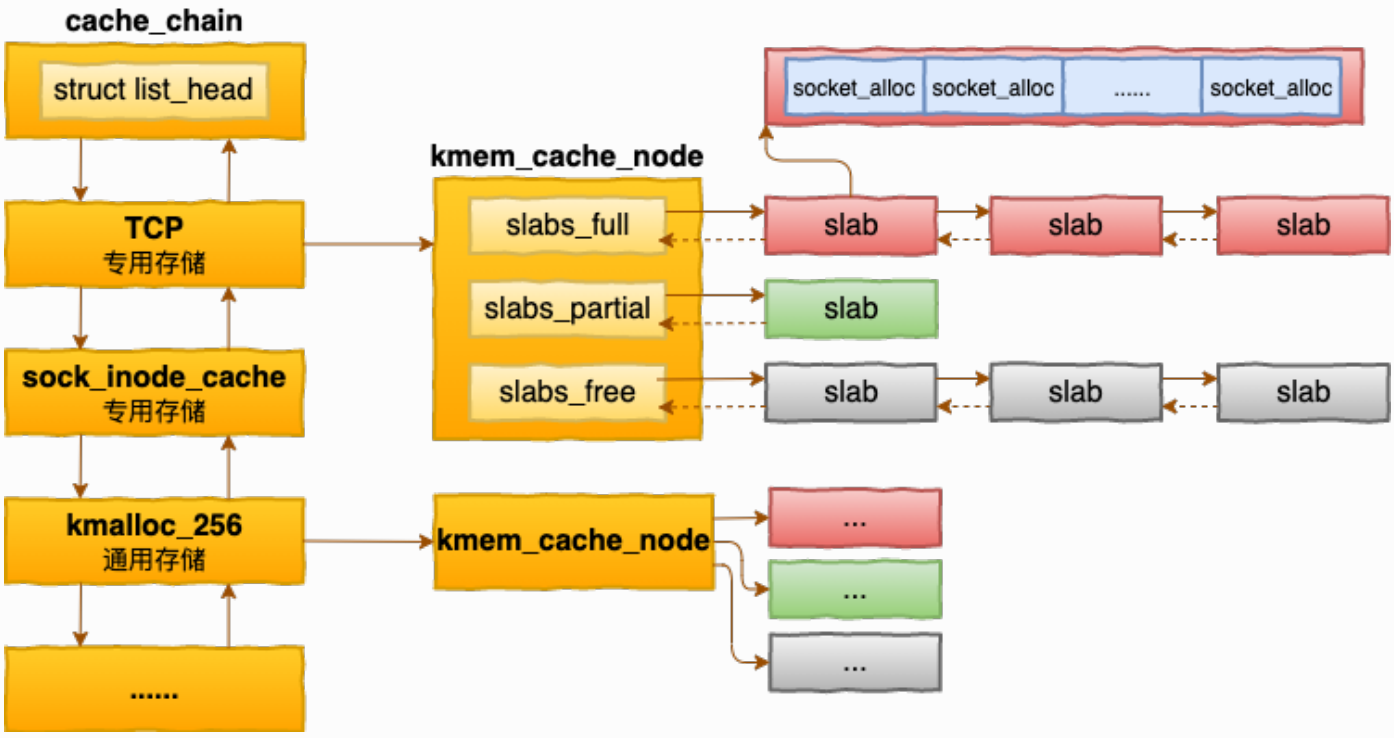


当 cache 中内存不够的时候，会调用基于伙伴系统的分配器（__alloc_pages函数）请求整页连续内存的分配。

```
//file: mm/slab.c
static void *kmem_getpages(struct kmem_cache *cachep,
          gfp_t flags, int nodeid)
{
    ......
    flags |= cachep->allocflags;
    if (cachep->flags & SLAB_RECLAIM_ACCOUNT)
        flags |= __GFP_RECLAIMABLE;

    page = alloc_pages_exact_node(nodeid, ...);
    ......
}
```

```
//file: include/linux/gfp.h
static inline struct page *alloc_pages_exact_node(int nid,
        gfp_t gfp_mask,unsigned int order)
{
    return __alloc_pages(gfp_mask, order, node_zonelist(nid,
gfp_mask));
}
```

内核中会有很多个 `kmem_cache` 存在。它们是在linux初始化，或者是运行的过程中分配出来的。它们有的是专用的，有的是通用的。

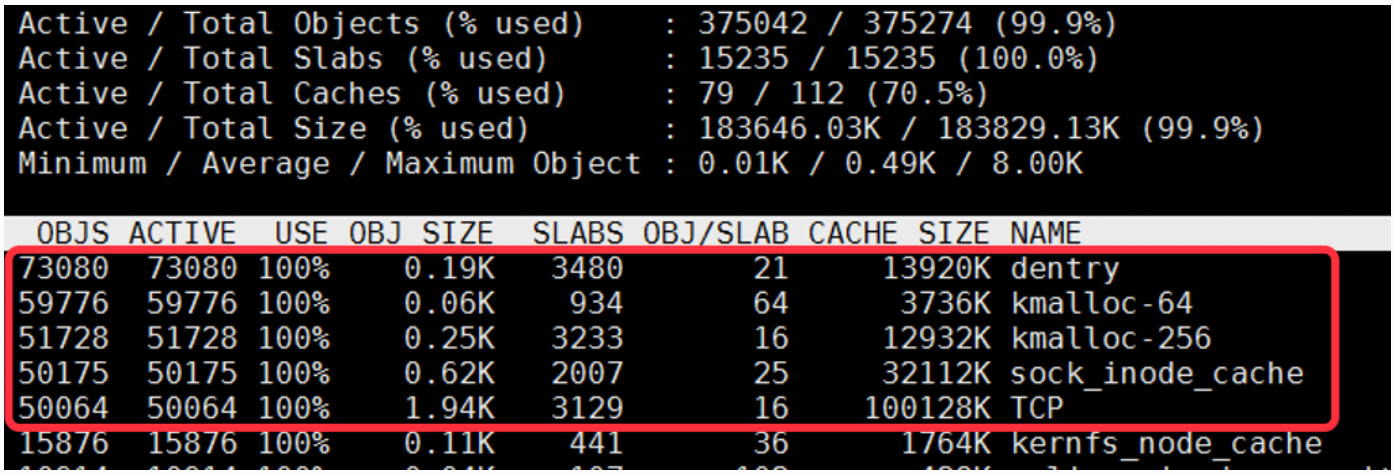


上图中，我们看到 socket_alloc 内核对象都存在 TCP的专用 kmem_cache 中。

通过查看 `/proc/slabinfo` 我们可以查看到所有的 kmem cache。

```
slabinfo - version: 2.1
# name            <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab>
sharedfactor> : slabdata <active_slabs> <num_slabs> <sharedavail>
kvm_vcpu               0      0  16256    2    8 : tunables    0    0    0 : slab
kvm_mmu_page_header       0       0    168   48    2 : tunables    0    0    0 : sl
xfs_dqtrx            992    992    528   31    4 : tunables    0    0    0 : slab
xfs_dquot             68     68    472   34    4 : tunables    0    0    0 : slab
xfs_icr              560    560    144   28    1 : tunables    0    0    0 : slab
xfs_ili             8480   8692    152   53    2 : tunables    0    0    0 : slab
xfs_inode           6837   7050   1088   30    8 : tunables    0    0    0 : slab
xfs_efd_item        2480   2800    400   40    4 : tunables    0    0    0 : slab
xfs_da_state         102    102    480   34    4 : tunables    0    0    0 : slab
xfs_btree_cur       1248   1248    208   39    2 : tunables    0    0    0 : slab
xfs_log_ticket      6996   6996    184   44    2 : tunables    0    0    0 : slab
nfsd4_openowners       0      0    440   37    4 : tunables    0    0    0 : slab
```

另外 linux 还提供了一个特别方便的命令 slabtop 来按照占用内存从大往小进行排列。这个命令用来分析 slab 内存开销非常的方便。



无论是 `/proc/slabinfo`，还是 slabtop 命令的输出。里面都包含了每个 cache 中 slab的如下两个关键信息。

- objsize：每个对象的大小
- objperslab： 一个 slab 里存放的对象的数量

在 `/proc/slabinfo` 还多输出了一个pagesperslab。 展示了一个slab 占用的页面的数量，每个页面4K，这样也就能算出每个 slab 占用的内存大小。

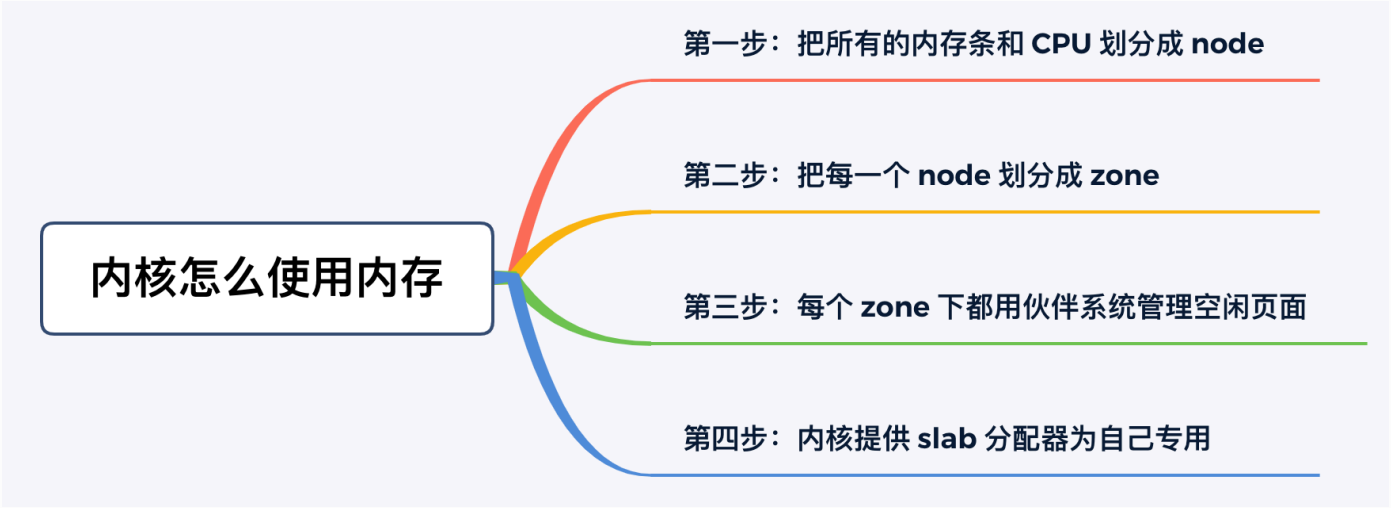
最后，slab 管理器组件提供了若干接口函数，方便自己使用。举三个例子：

- **kmem_cache_create**: 方便地创建一个基于 slab 的内核对象管理器。
- **kmem_cache_alloc**: 快速为某个对象申请内存
- **kmem_cache_free**: 归还对象占用的内存给 slab 管理器

在内核的源码中，可以大量见到 kmem_cache 开头函数的使用。

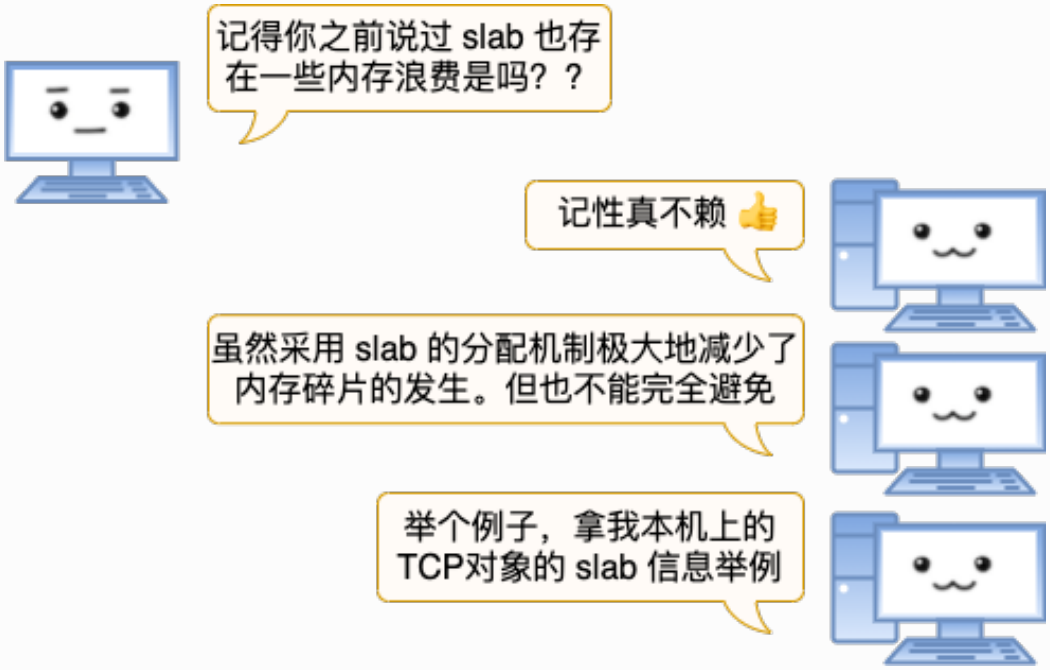
4.1.5 总结

通过上面描述的几个步骤，内核高效地把内存用了起来。



- 第一步：把所有的内存条和 CPU 划分成 node
- 第二步：把每一个 node 划分成 zone
- 第三步：每个 zone 下都用伙伴系统管理空闲页面
- 第四步：内核提供 slab 分配器为自己专用

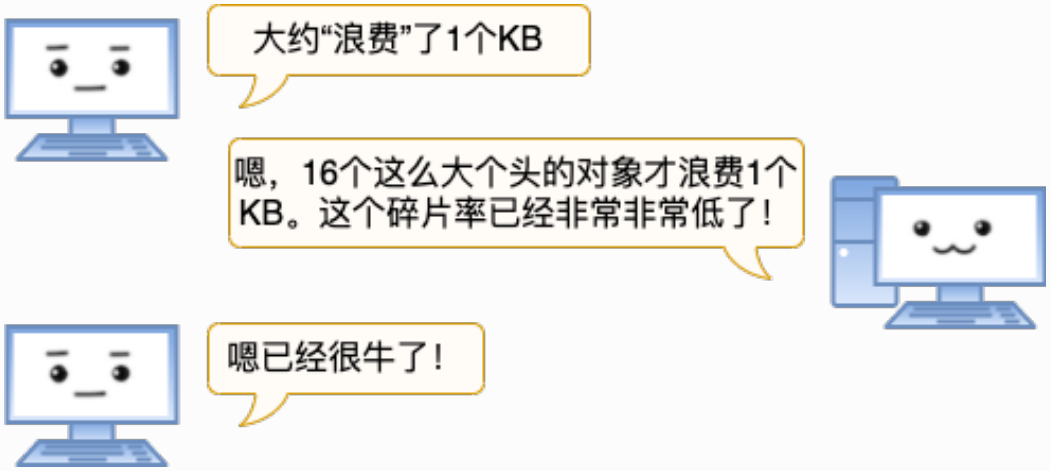
前三步是基础模块，为应用程序分配内存时的请求调页组件也能够用到。但第四步，就算是内核的小灶了。内核根据自己的使用场景，量身打造的一套自用的高效内存分配管理机制。




```
# cat /proc/slabinfo | grep TCP
TCP                    288    384    1984    16    8
```

“可以看到 TCP cache下每个 slab 占用 8 个 Page，也就是 8* 4096 = 32768KB。
该对象的单个大小是 1984 字节 字节，每个slab内放了 16 个对象。1984*16=31744”

“这个时候再多放一个 TCP 对象又放不下，剩下的 1K 内存就只好“浪费”掉了。 但是鉴于 slab 机制整体提供的高性能、以及低碎片的效果，这一点点的额外开销还是很值得的。”

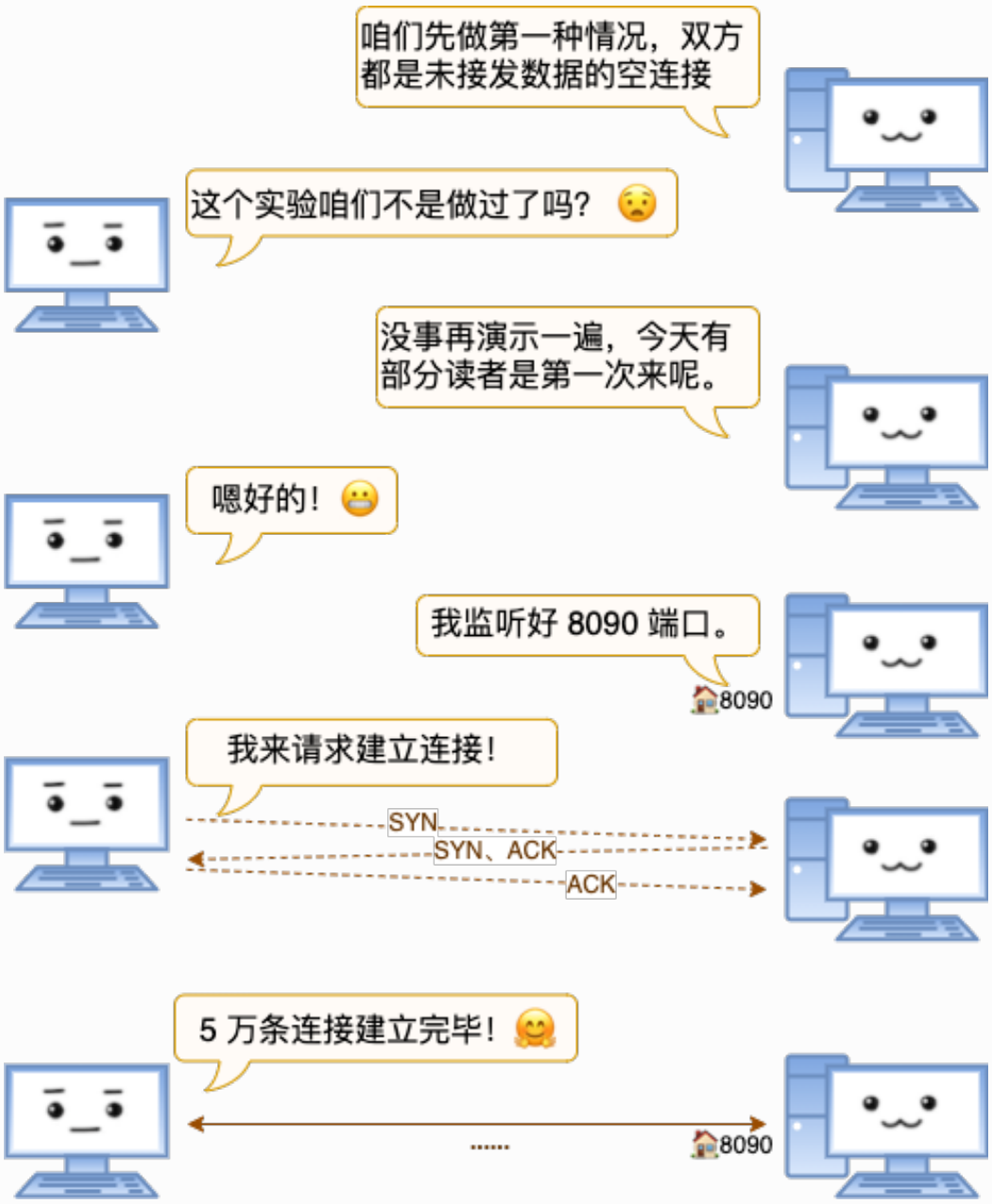


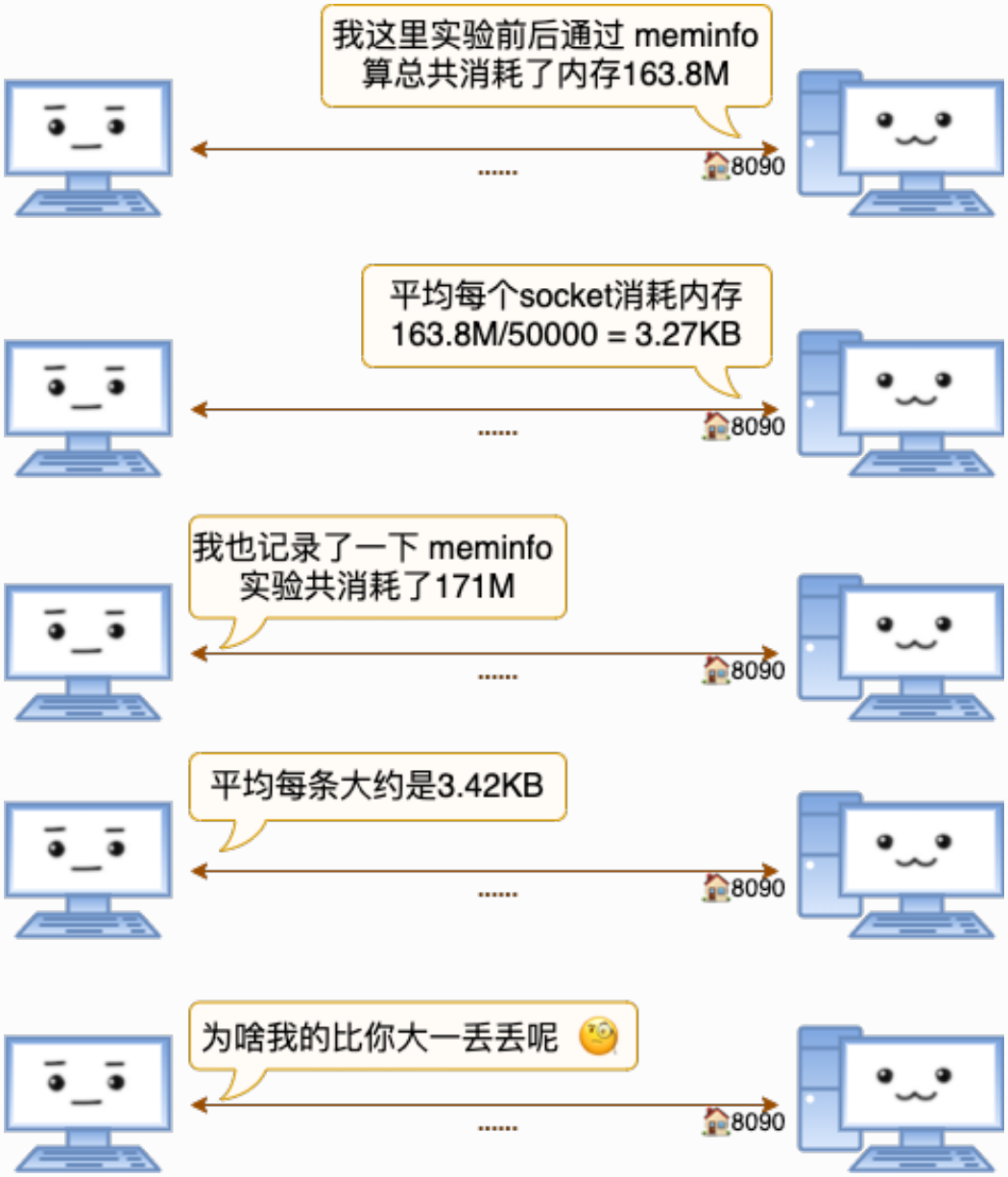
4.2 实验测试 TCP 连接内存开销

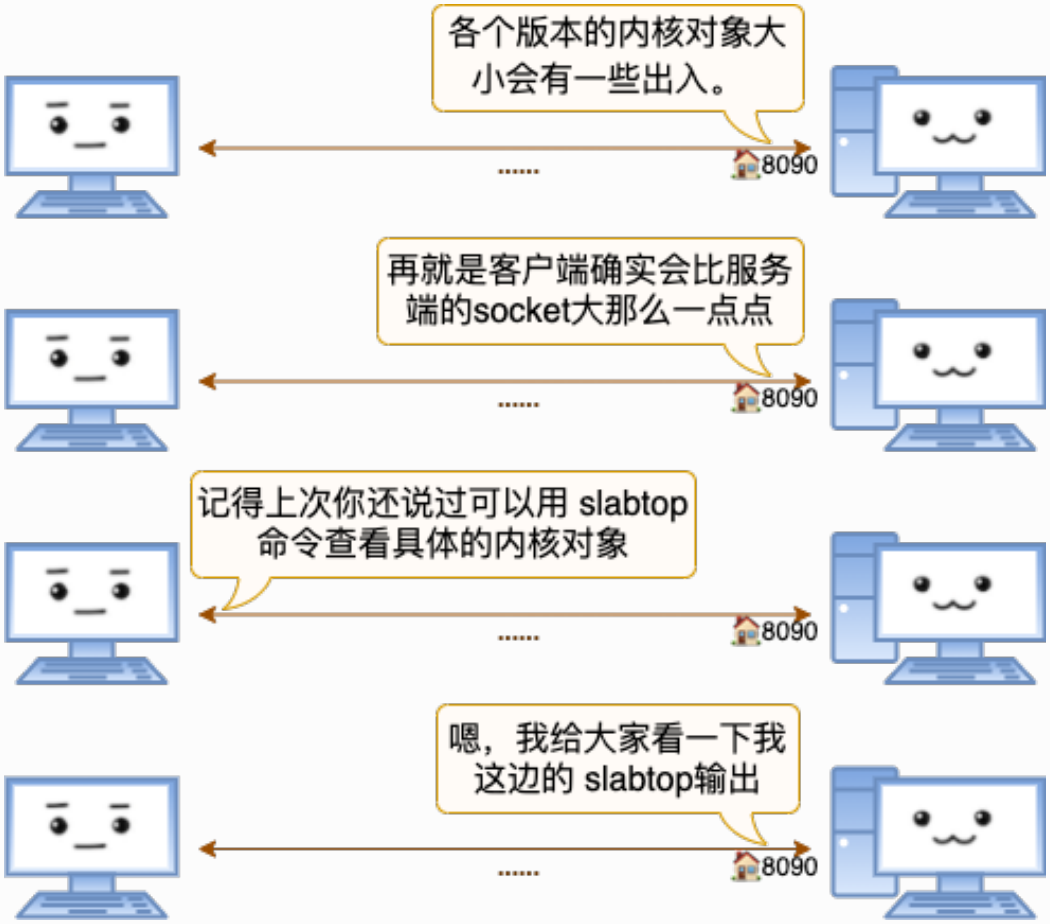
> 实际中 TCP 连接上肯定是要进行数据的收发的，而且还会有 TIME_WAIT 等其它状态。在这些复杂情况下，一条连接占用多大内存呢？飞哥用做了七天的实验结果告诉你！


老兄，我有个疑惑 😢
啥疑惑，说来听听?
咱们之前看一条空的 ESTAB 状态 TCP 连接消耗3.3KB左右内存。
没错，之前的实验你也看到了 🤓
但是我觉得咱们的实验做的还不够全面 😳
实际工作的时候我们不收发数据是不可能的 😢
嗯，这个咱们之前只是简单提过发送和接收需要额外的缓存区。
如果有数据收发，实际内存占用是啥样的? 😂

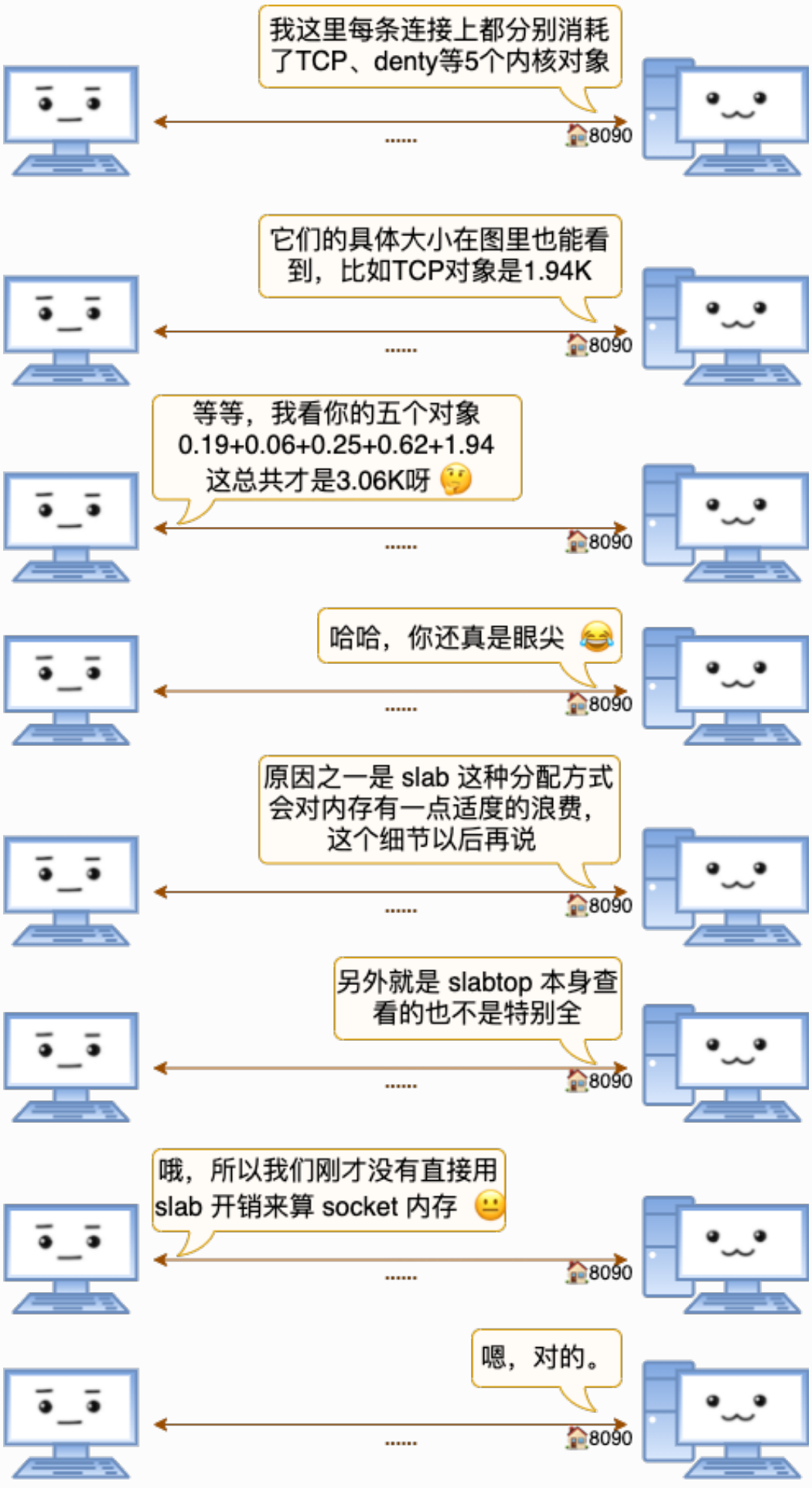
4.2.1 ESTABLISH空连接


咱们先做第一种情况，双方都是未接发数据的空连接
这个实验咱们不是做过了吗？
没事再演示一遍，今天有部分读者是第一次来呢。
嗯好的！
我监听好 8090 端口。
8090
我来请求建立连接！
SYN
SYN、ACK
ACK
5 万条连接建立完毕！
.....
8090

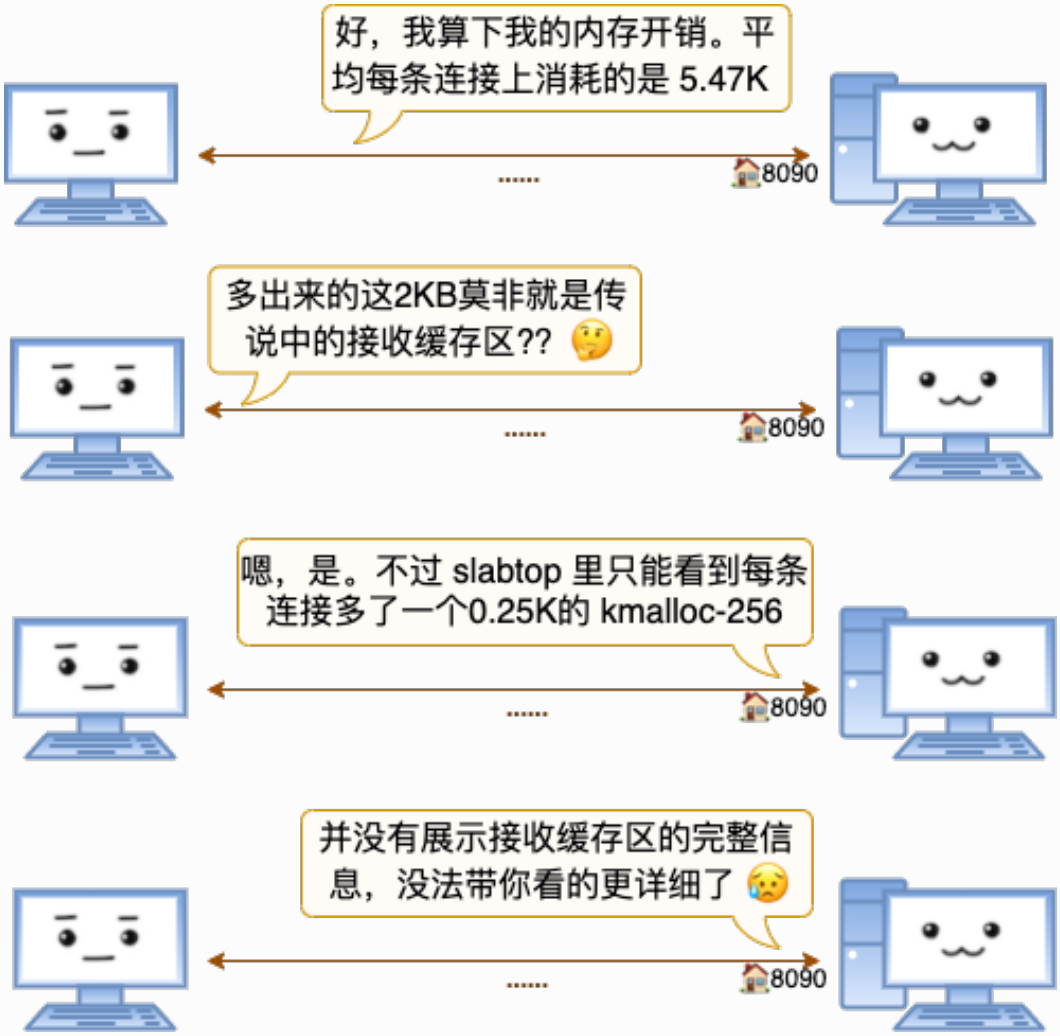

我这里实验前后通过 meminfo
算总共消耗了内存163.8M
.....
8090
平均每个socket消耗内存
163.8M/50000 = 3.27KB
.....
8090
我也记录了一下 meminfo
实验共消耗了171M
.....
8090
平均每条大约是3.42KB
.....
8090
为啥我的比你大一丢丢呢
.....
8090



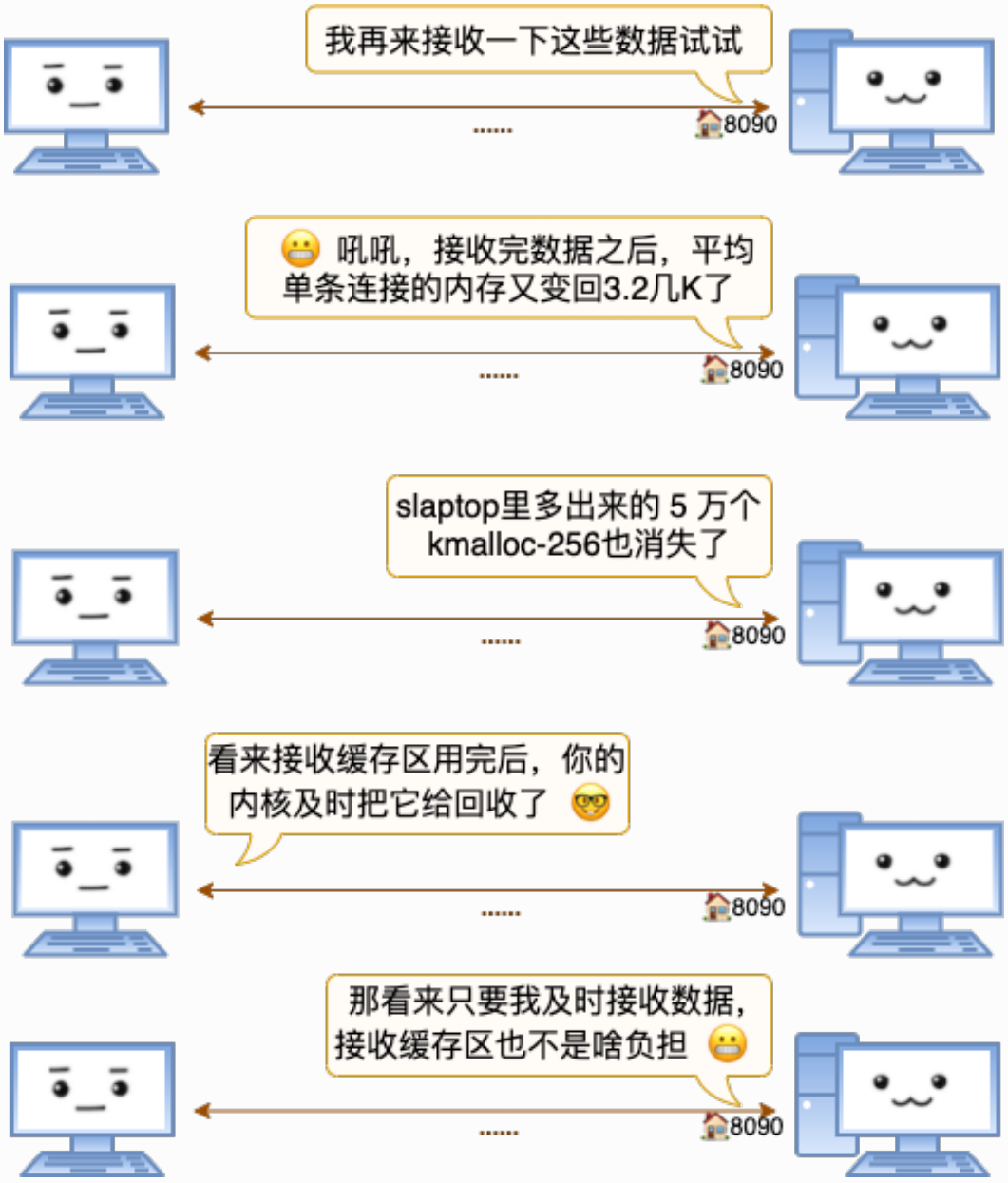

```
Active / Total Objects (% used)    : 375042 / 375274 (99.9%)
Active / Total Slabs (% used)      : 15235 / 15235 (100.0%)
Active / Total Caches (% used)     : 79 / 112 (70.5%)
Active / Total Size (% used)       : 183646.03K / 183829.13K (99.9%)
Minimum / Average / Maximum Object : 0.01K / 0.49K / 8.00K

  OBJS ACTIVE  USE OBJ SIZE  SLABS OBJ/SLAB CACHE SIZE NAME
73080  73080 100%    0.19K   3480       21     13920K dentry
59776  59776 100%    0.06K    934       64      3736K kmalloc-64
51728  51728 100%    0.25K   3233       16     12932K kmalloc-256
50175  50175 100%    0.62K   2007       25     32112K sock_inode_cache
50064  50064 100%    1.94K   3129       16    100128K TCP
15876  15876 100%    0.11K    441       36      1764K kernfs_node_cache
```

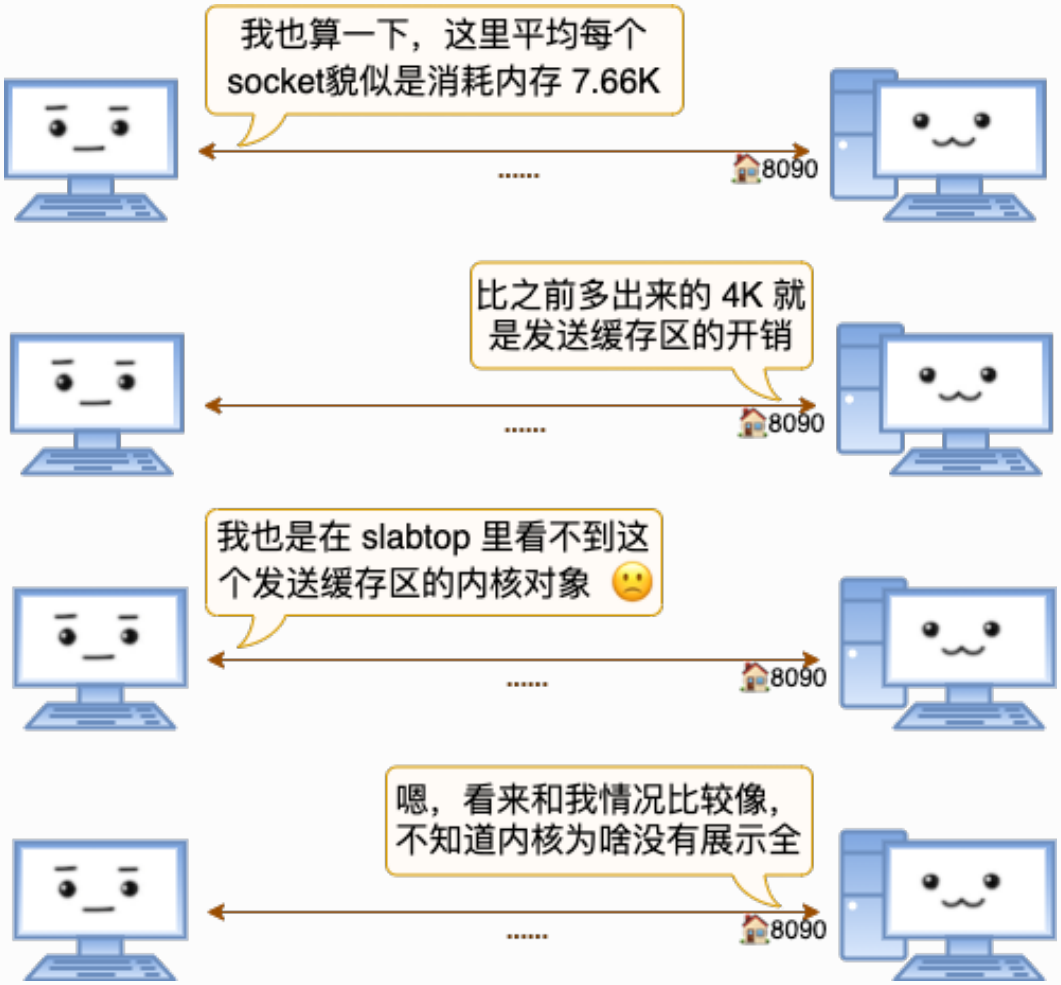

我这里每条连接上都分别消耗了TCP、denty等5个内核对象
8090
它们的具体大小在图里也能看到，比如TCP对象是1.94K
8090
等等，我看你的五个对象
0.19+0.06+0.25+0.62+1.94
这总共才是3.06K呀
8090
哈哈，你还真是眼尖
8090
原因之一是 slab 这种分配方式会对内存有一点适度的浪费，这个细节以后再说
8090
另外就是 slabtop 本身查看的也不是特别全
8090
哦，所以我们刚才没有直接用 slab 开销来算 socket 内存
8090
嗯，对的。
8090

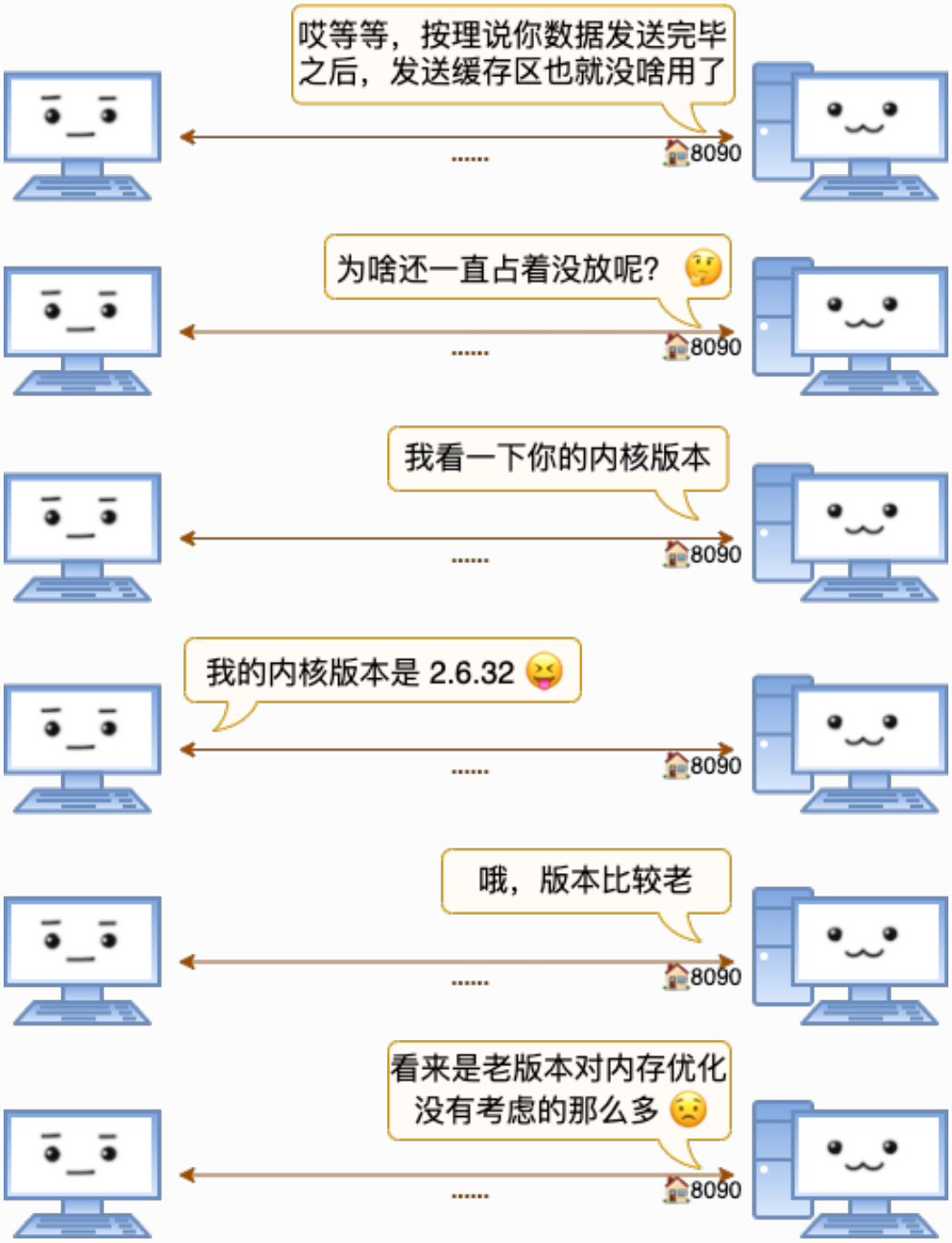
4.2.2 实验2：客户端 => 服务器发送数据测试




```
Active / Total Objects (% used)    : 430822 / 431661 (99.8%)
Active / Total Slabs (% used)      : 18599 / 18599 (100.0%)
Active / Total Caches (% used)     : 78 / 112 (69.6%)
Active / Total Size (% used)       : 198005.84K / 198326.77K (99.8%)
Minimum / Average / Maximum Object : 0.01K / 0.46K / 8.00K

  OBJS ACTIVE  USE OBJ SIZE  SLABS OBJ/SLAB CACHE SIZE NAME
102016 101945  99%    0.25K   6376       16     25504K kmalloc-256
 74025  74025 100%    0.19K   3525       21     14100K dentry
 59648  59648 100%    0.06K    932       64      3728K kmalloc-64
 50208  50208 100%    1.94K   3138       16    100416K TCP
 50175  50175 100%    0.62K   2007       25     32112K sock_inode_cache
 15912  15912 100%    0.11K    442       36      1768K kernfs_node_cache
 10989  10989 100%    0.58K    407       27      6512K inode_cache
 10812  10812 100%    0.04K    106      102       424K selinux_inode_security
  6156   6083  98%    0.21K    342       18      1368K vm_area_struct
  5499   5499 100%    0.10K    141       39       564K buffer_head
  4096   4096 100%    0.01K      8      512        32K kmalloc-8
```

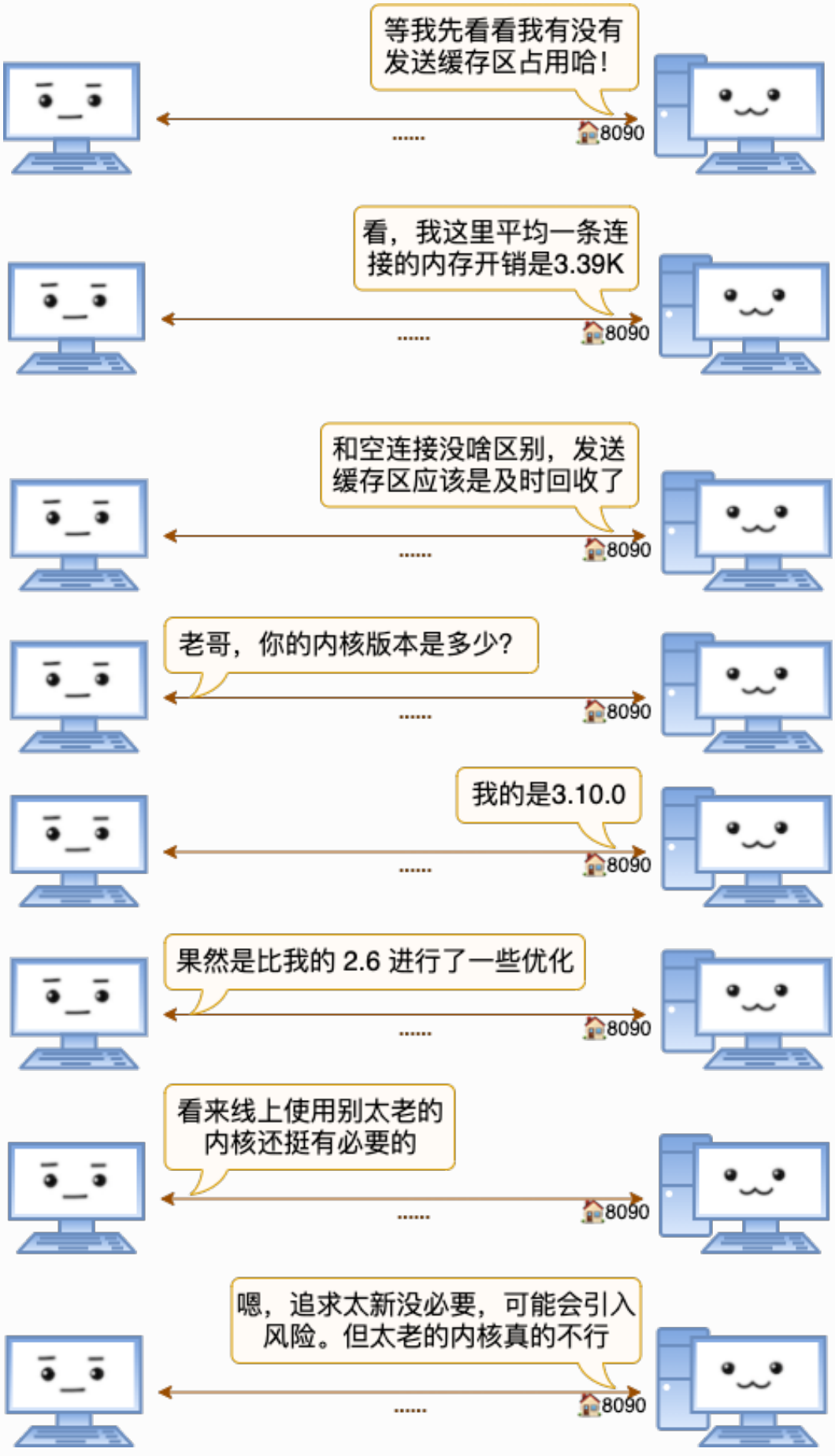

我再来接收一下这些数据试试
.....
8090
吼吼，接收完数据之后，平均单条连接的内存又变回3.2几K了
.....
8090
slaptop里多出来的 5 万个kmalloc-256也消失了
.....
8090
看来接收缓存区用完后，你的内核及时把它给回收了
.....
8090
那看来只要我及时接收数据，接收缓存区也不是啥负担
.....
8090

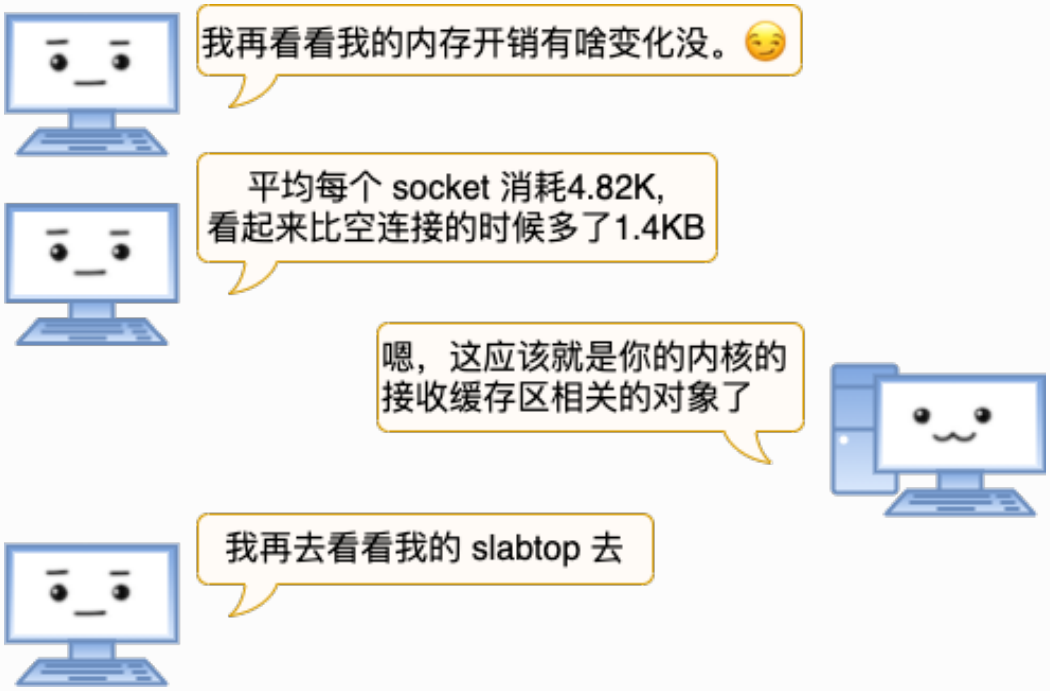

我也算一下，这里平均每个socket貌似是消耗内存 7.66K
.....
8090
比之前多出来的 4K 就是发送缓存区的开销
.....
8090
我也是在 slabtop 里看不到这个发送缓存区的内核对象 😕
.....
8090
嗯，看来和我情况比较像，不知道内核为啥没有展示全
.....
8090



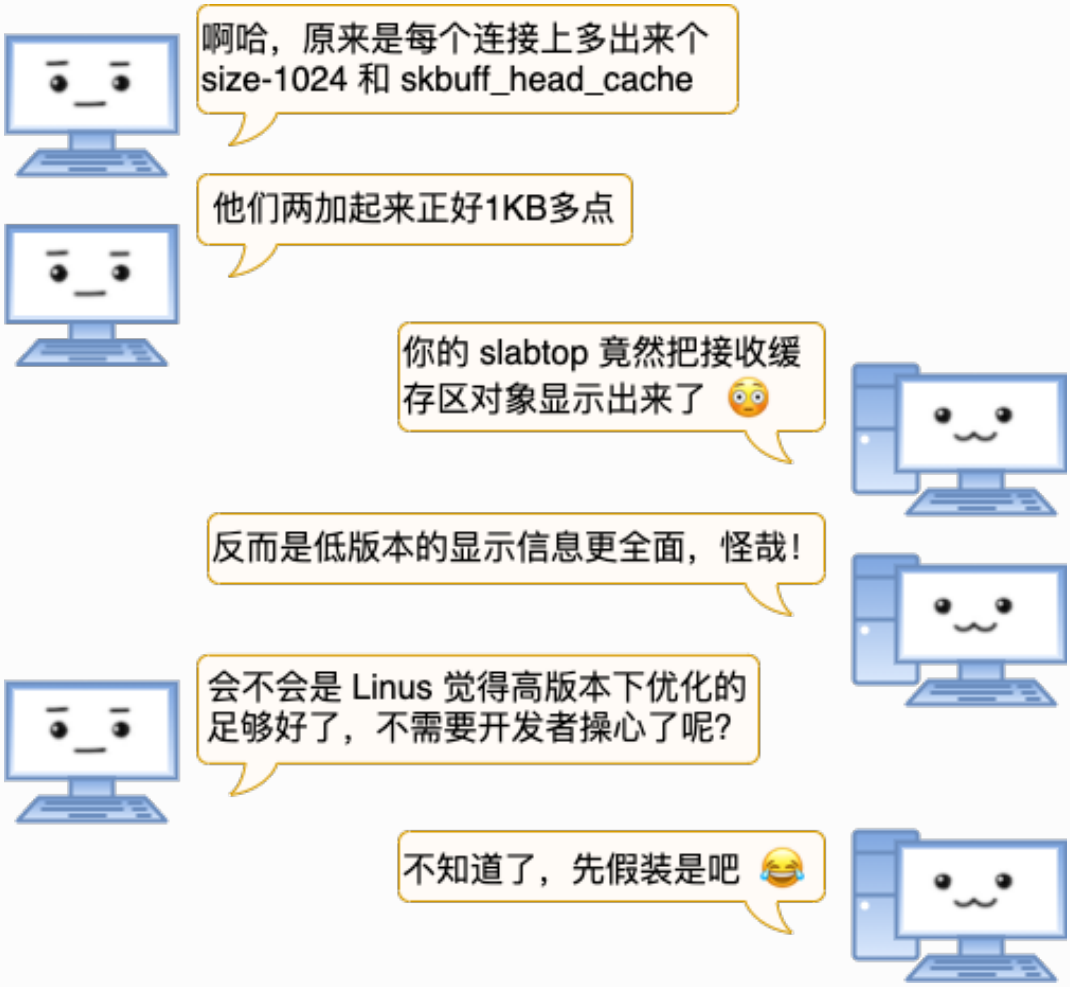
4.2.3 实验3: 服务器 => 客户端发送数据测试


老哥，这次咱两把数据方向反一下
哈哈好，这次我给你发数据
我来重新建立连接
我每条连接上都发个
"I am server！" 过去
5万条连接建立完毕
数据我先不接收的呢


等我先看看我有没有发送缓存区占用哈！
8090
看，我这里平均一条连接的内存开销是3.39K
8090
和空连接没啥区别，发送缓存区应该是及时回收了
8090
老哥，你的内核版本是多少?
8090
我的是3.10.0
8090
果然是比我的 2.6 进行了一些优化
8090
看来线上使用别太老的内核还挺有必要的
8090
嗯，追求太新没必要，可能会引入风险。但太老的内核真的不行
8090

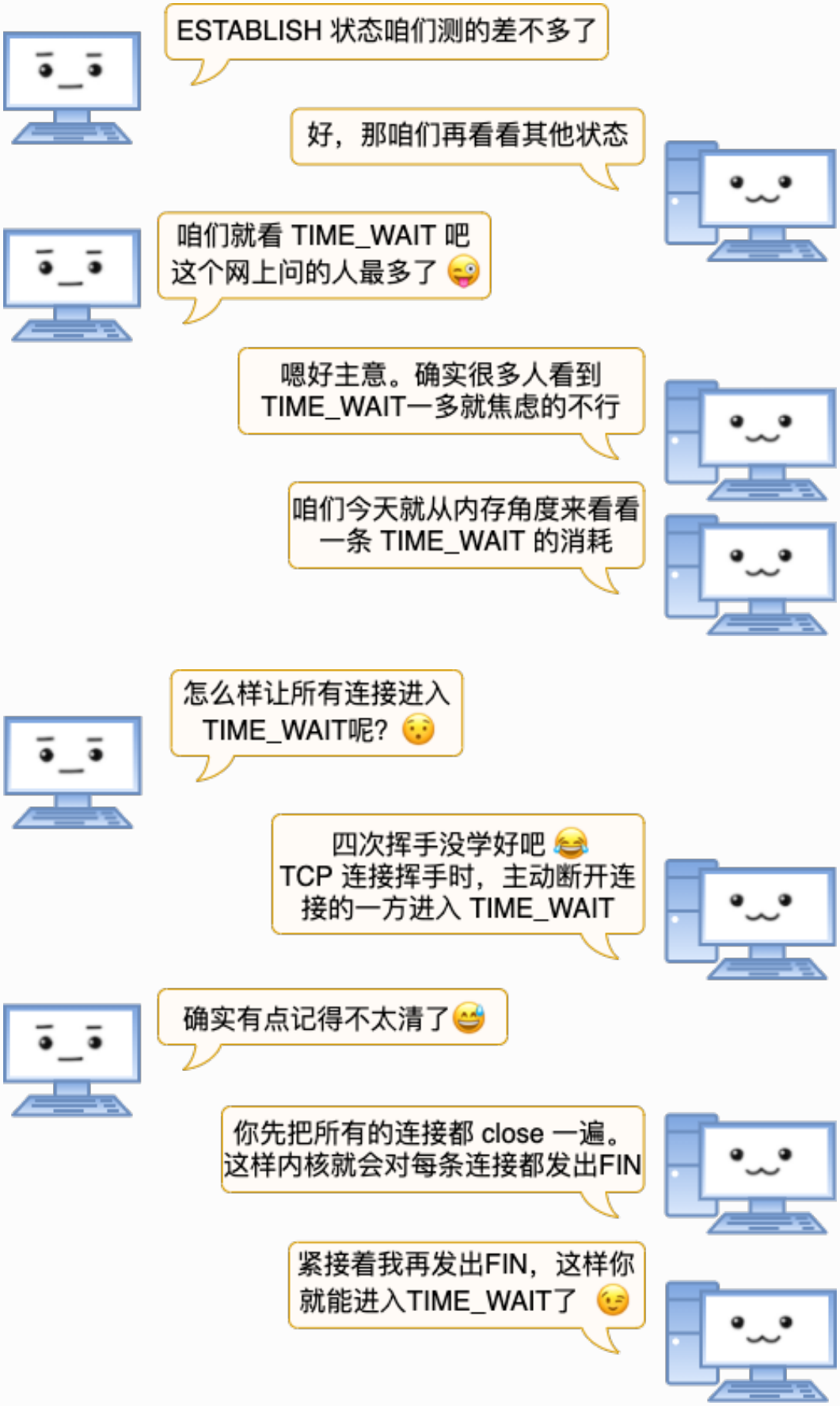


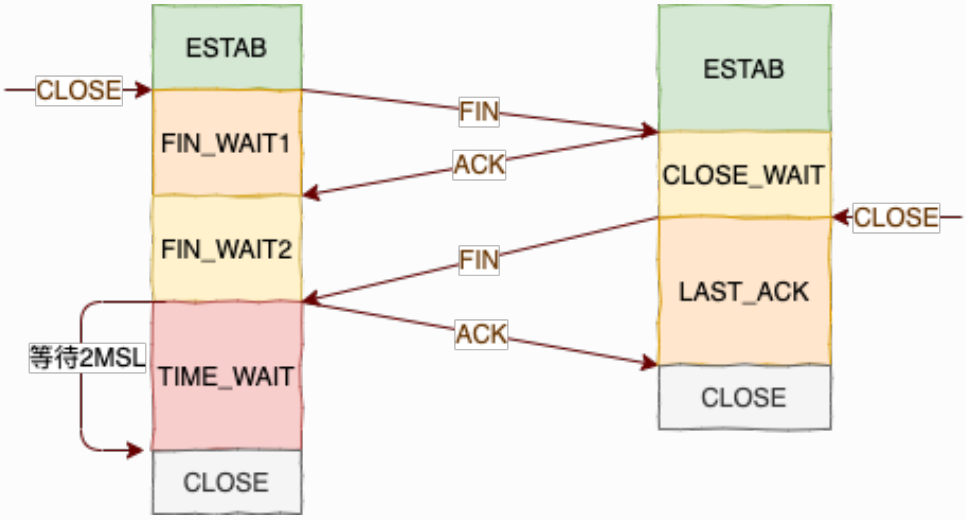

```
 OBJS ACTIVE  USE OBJ SIZE  SLABS OBJ/SLAB CACHE SIZE NAME
63120  63096  99%    0.19K   3156       20     12624K dentry
51300  51180  99%    0.19K   2565       20     10260K filp
50540  50540 100%    1.00K  12635        4     50540K size-1024
50160  50160 100%    0.69K  10032        5     40128K sock_inode_cache
50032  50014  99%    0.06K    848       59      3392K tcp_bind_bucket
50020  50017  99%    1.62K  12505        4    100040K TCP
50010  50010 100%    0.25K   3334       15     13336K skbuff_head_cache
25134  25074  99%    0.06K    426       59      1704K size-64
24864  24558  98%    0.03K    222      112       888K size-32
21571  21547  99%    0.10K    583       37      2332K buffer_head
 7344   7331  99%    0.58K   1224        6      4896K inode_cache
 7102   7077  99%    0.07K    134       53       536K selinux_inode_security
```


啊哈，原来是每个连接上多出来个size-1024 和 skbuff_head_cache
他们两加起来正好1KB多点
你的 slabtop 竟然把接收缓存区对象显示出来了
反而是低版本的显示信息更全面，怪哉！
会不会是 Linus 觉得高版本下优化的足够好了，不需要开发者操心了呢?
不知道了，先假装是吧


另外我的内核参数 tcp_rmem 的最小值是 4096， 是4K
内核实际消耗的只比空连接多了1.4K
这个 tcp_rmem 的最小值貌似没起作用嘿
我这里 tcp_rmem 最小也是 4K, 但上个实验里看接收缓存区平均也只是2K大
这是为啥?
内核接收缓存区实际分配，不仅仅看tcp_rmem，还会受SO_RCVBUF、实际发送数据大小的影响。
总之是一套比较复杂的分配策略。所以很多时候得动手试，不能老看网上的解释。
我这次接收一下数据再看看。
接收完之后单 TCP 连接内存又回到3K多了
看来你的接收缓存区倒是及时回收了

4.2.4 实验4： 非 ESTABLISH 状态





哈哈，都想起来了。

不过记得关闭 tw_reuse 和tw_recycle，否则也无法正常进入TIME_WAIT

好的，我先来 close

嗯。我等收到你的 FIN 以后再 close

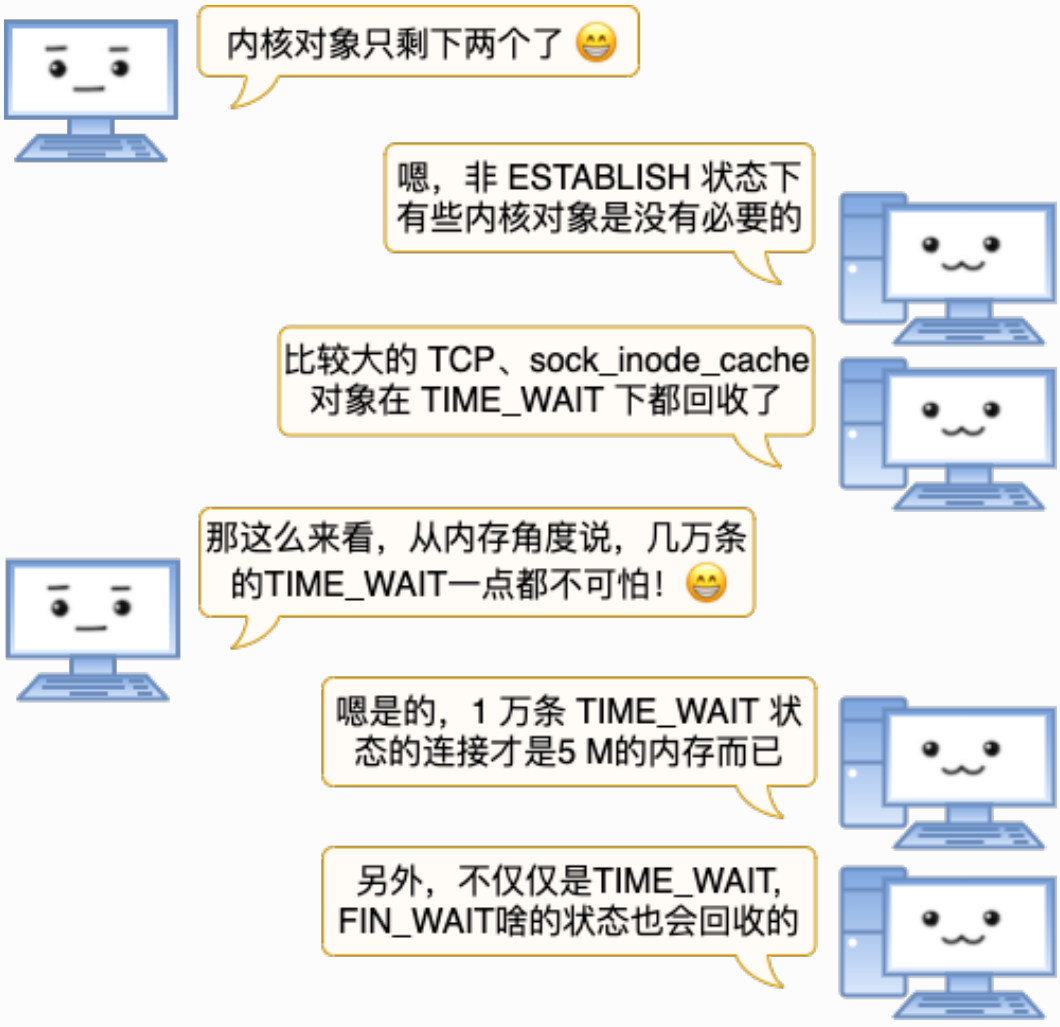
我这里成功看到 5W 个 TIME_WAIT啦！ 🤗

平均每条连接内存消耗算下来也就是 0.5K

再给大伙看看你的 slabtop 输出

好嘞，请看！

```
Active / Total Objects (% used)    : 231418 / 236792 (97.7%)
Active / Total Slabs (% used)      : 11126 / 11142 (99.9%)
Active / Total Caches (% used)     : 114 / 206 (55.3%)
Active / Total Size (% used)       : 47398.82K / 48266.21K (98.2%)
Minimum / Average / Maximum Object : 0.02K / 0.20K / 4096.00K

 OBJS ACTIVE  USE OBJ SIZE  SLABS OBJ/SLAB CACHE SIZE NAME
50032  50015  99%    0.06K    848       59      3392K tcp_bind_bucket
50010  50007  99%    0.25K   3334       15     13336K tw_sock_TCP
24864  24632  99%    0.03K    222      112       888K size-32
24457  24433  99%    0.10K    661       37      2644K buffer_head
14540  14060  96%    0.19K    727       20      2908K dentry
13275  12675  95%    0.06K    225       59       900K size-64
 7350   7350 100%    0.58K   1225        6      4900K inode_cache
```



4.2.5 总结

我们把实验中的数据来总结一下

序号	TCP状态	收发数据	客户端单socket	服务端单socket	备注
实验1	ESTAB	无	3.42K	3.27K	socket_alloc等核心内核对象
实验2	ESTAB	客户端发送 服务器不收	7.66K	5.47K	客户端的发送缓存区没回收 服务器也多了接收缓存区
	ESTAB	客户端发送 服务器接收		3.24K	服务器接收缓存区用完回收了
实验3	ESTAB	服务器发送 客户端不收	4.82K	3.39K	服务端发送缓存区及时回收了 客户端多了size-1024等内核对象
	ESTAB	服务器发送 客户端接收	3.56K		客户端接收缓存区用完回收了
实验4	TIME_WAIT	无	0.5K	0K	TIME_WAIT 下会回收无用对象 服务端就直接关闭了

可见，内核在 socket 内存开销优化上采取了不少方法

- 1. 内核会尽量及时回收发送缓存区、接收缓存区，但高版本做的更好
- 2. 发送接收缓存区最小并不一定是 rmem 内核参数里的最小值，实际可能会更小
- 3. 其它状态下，例如对于TIME_WAIT还会回收非必要的 socket_alloc 等对象

另外注意，我们测试时发送的数据量很小。如果发送数据量大了的话，发送缓存区和接收缓存区会消耗的更大。其对应的内核参数分别是 tcp_rmem 和 tcp_wmem。但是一旦发送或接收完毕，内核还是会尽量回收这块内存。

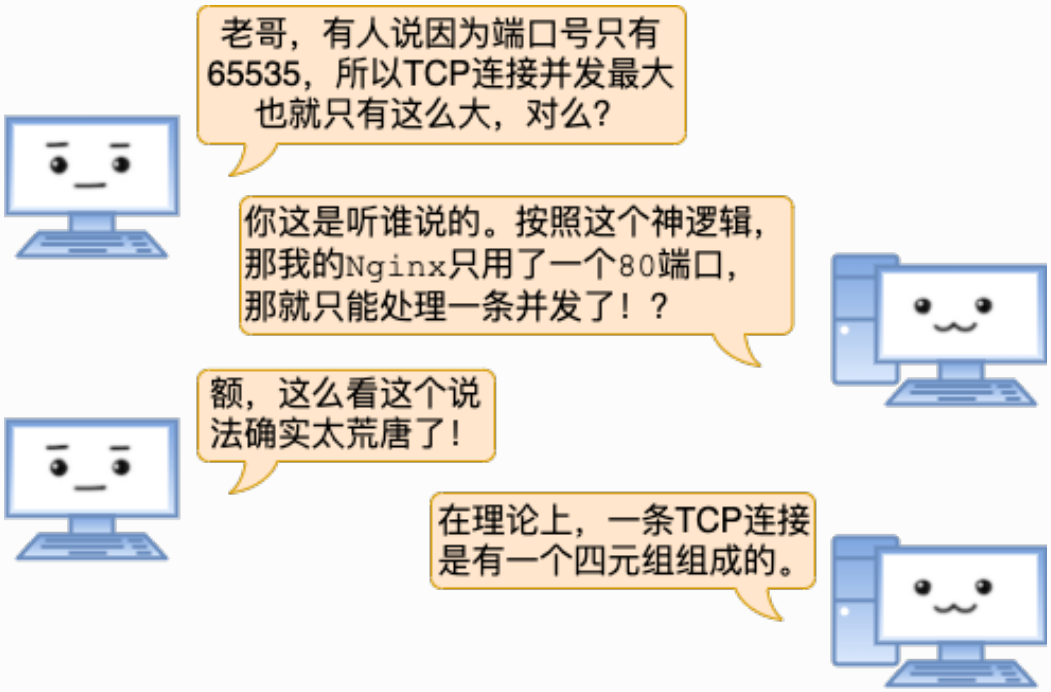
4.3 一台服务器最大可以支撑多少条TCP连接

在网络开发中，我发现有很多同学对一个基础问题始终是没有彻底搞明白。那就是一台服务器最大究竟能支持多少个网络连接？我想我有必要单独发一篇文章来好好说一下这个问题。

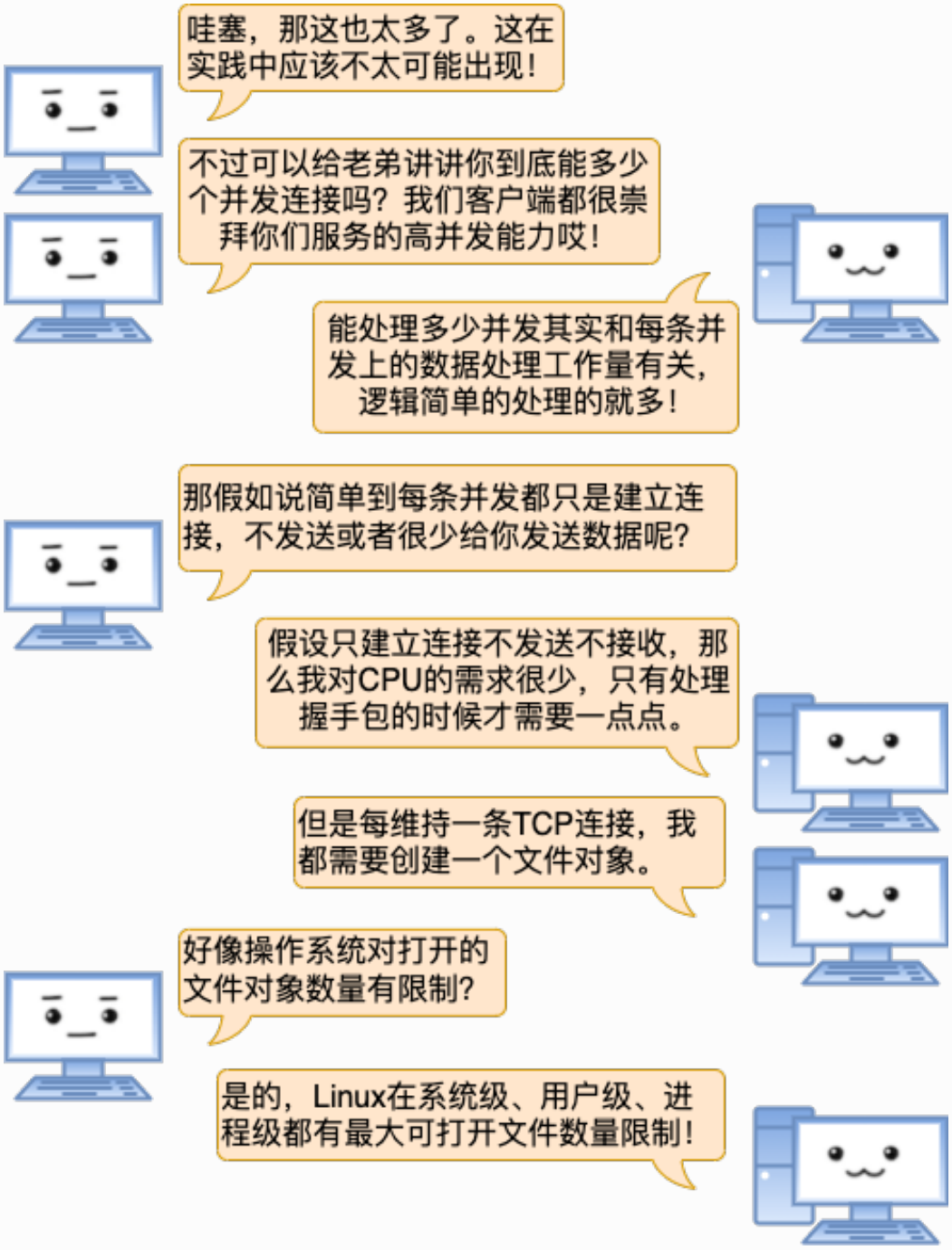
很多同学看到这个问题的第一反应是65535。原因是：“听说端口号最多有65535个，那长连接就最多保持65535个了”。是这样的吗？还有的人说：“应该受TCP连接里四元组的空间大小限制，算起来是200多万亿个！”

如果你对这个问题也是理解的不够彻底，那么今天讲个故事讲给你听！

4.3.1 一次关于服务器端并发的聊天

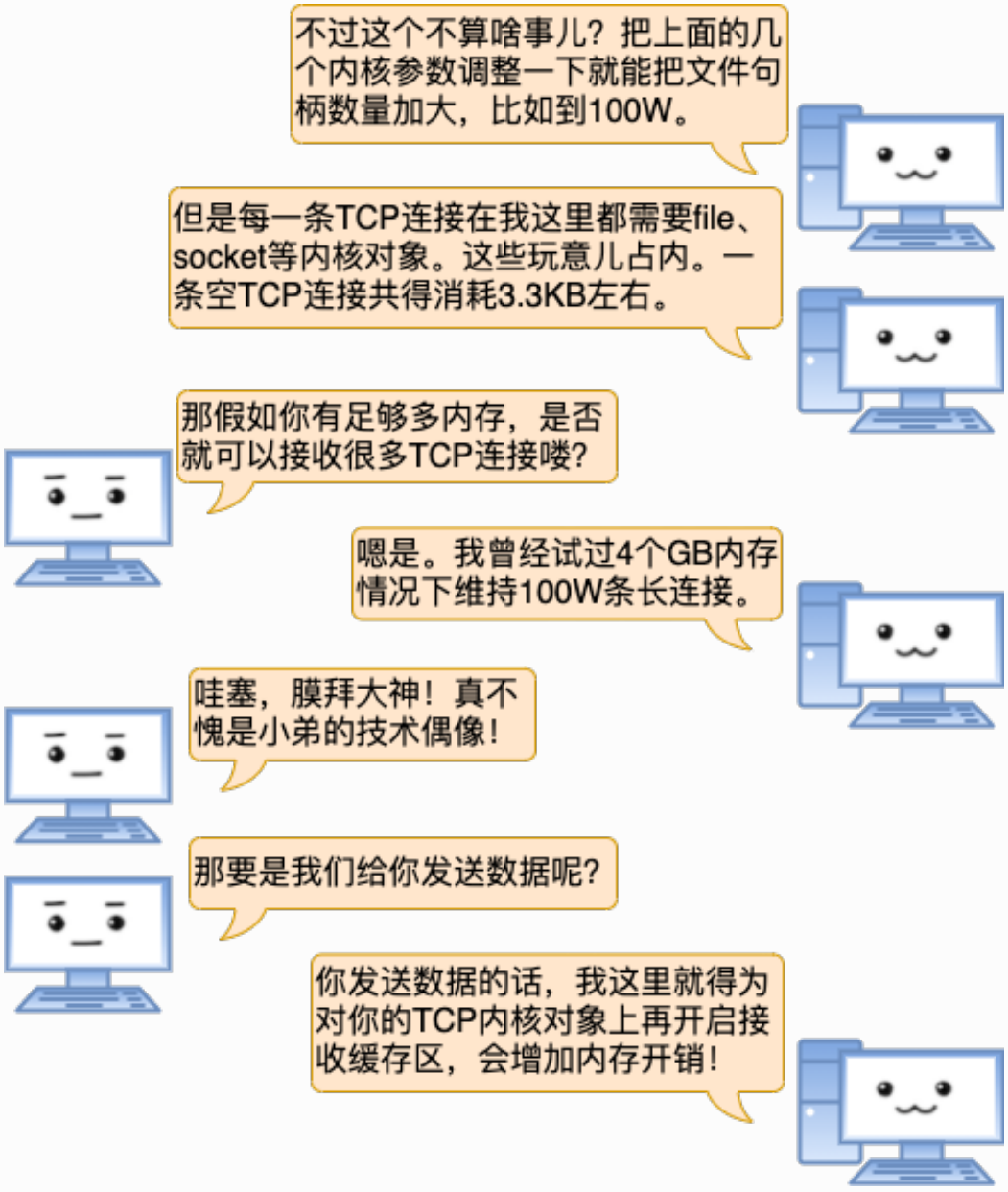


> "TCP连接四元组是源IP地址、源端口、目的IP地址和目的端口。任意一个元素发生了改变，那么就代表的是一条完全不同的连接了。拿我的Nginx举例，它的端口是固定使用80。另外我的IP也是固定的，这样目的IP地址、目的端口都是固定的。剩下源IP地址、源端口是可变的。所以理论上我的Nginx上最多可以建立2的32次方（ip数）×2的16次方（port数）个连接。这是两百多万亿的一个大数字！！"



> "进程每打开一个文件（linux下一切皆文件，包括socket），都会消耗一定的内存资源。如果有不怀好心的人启动一个进程来无限的创建和打开新的文件，会让服务器崩溃。所以linux系统出于安全角度的考虑，在多个位置都限制了可打开的文件描述符的数量，包括系统级、用户级、进程级。这三个限制的含义和修改方式如下："

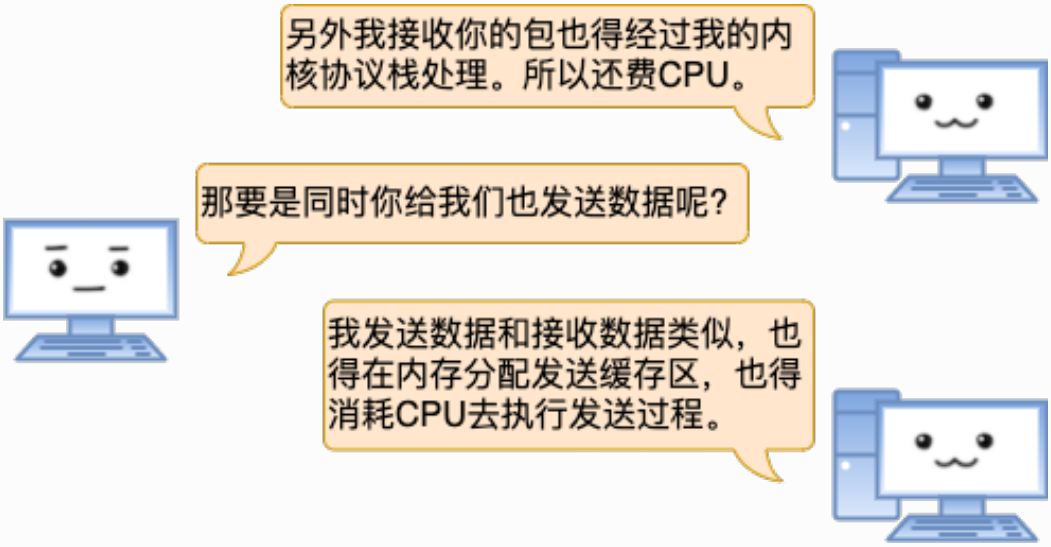
- 系统级：当前系统可打开的最大数量，通过fs.file-max参数可修改
- 用户级：指定用户可打开的最大数量，修改/etc/security/limits.conf
- 进程级：单个进程可打开的最大数量，通过fs.nr_open参数可修改



> "我的接收缓存区大小是可以配置的，通过sysctl命令就可以查看。"

```
$ sysctl -a | grep rmem
net.ipv4.tcp_rmem = 4096  87380 8388608
net.core.rmem_default = 212992
net.core.rmem_max = 8388608
```

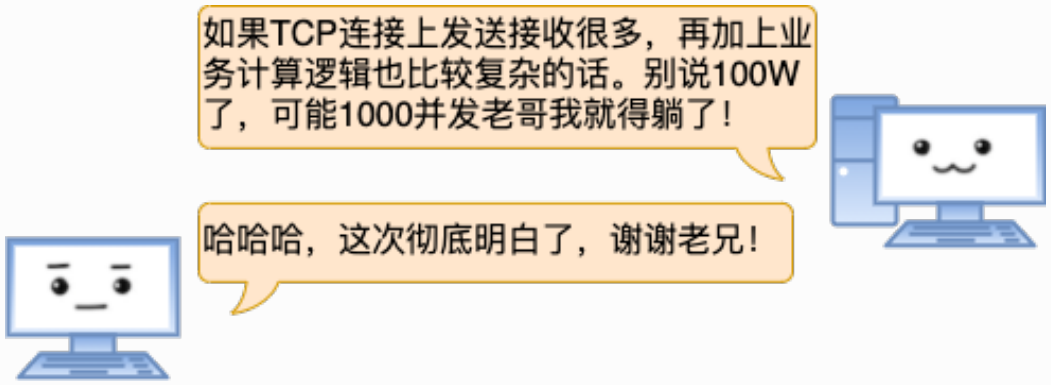
> "其中在tcp_rmem"中的第一个值是为你们的TCP连接所需分配的最少字节数。该值默认是4K，最大的话8MB之多。也就是说你们有数据发送的时候我需要至少为对应的socket再分配4K内存，甚至可能更大。"



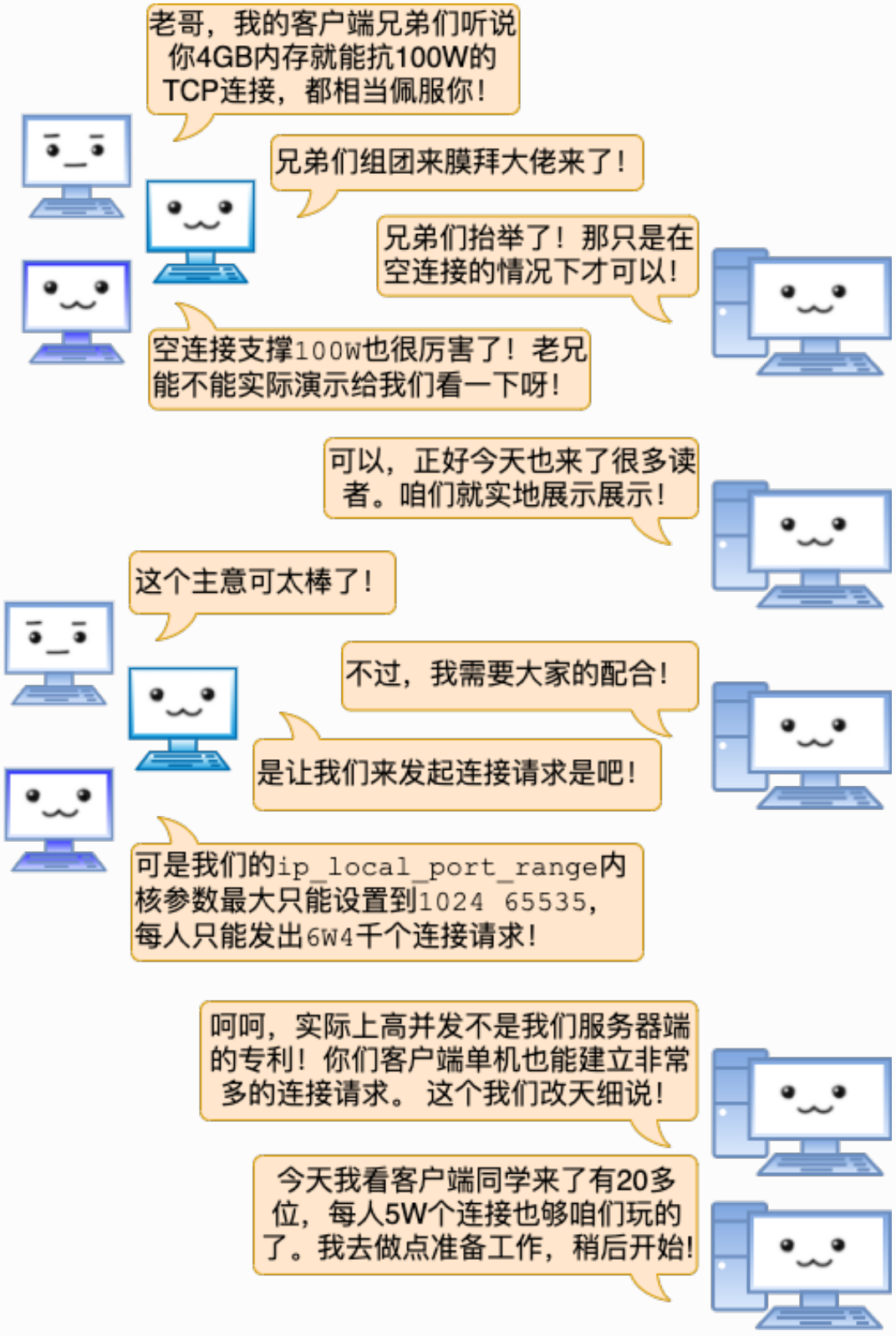
> "TCP分配发送缓存区的大小受参数net.ipv4.tcp_wmem配置影响。"

```
$ sysctl -a | grep wmem
net.ipv4.tcp_wmem = 4096  65536 8388608
net.core.wmem_default = 212992
net.core.wmem_max = 8388608
```

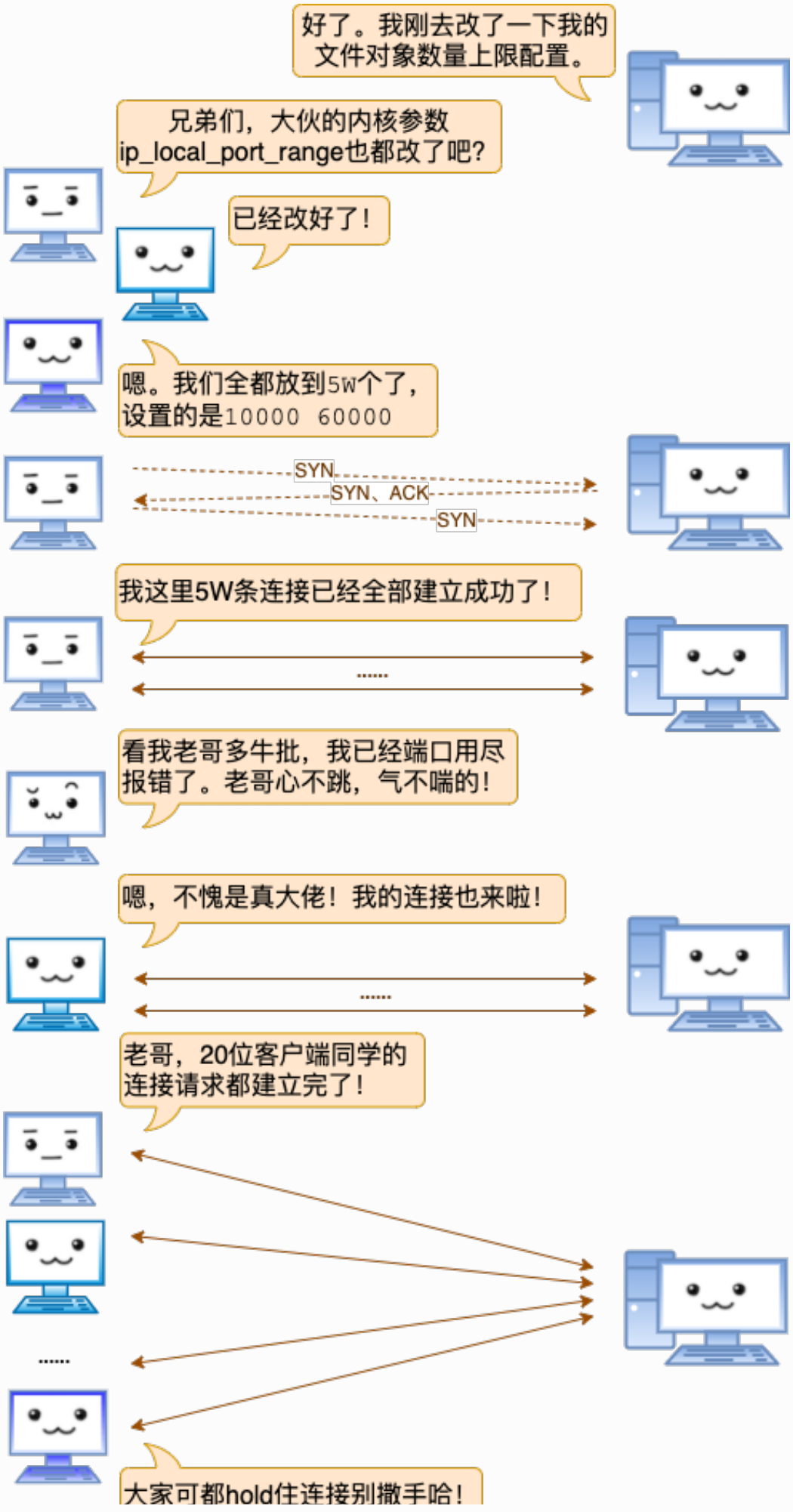
> "在net.ipv4.tcp_wmem"中的第一个值是发送缓存区的最小值，默认也是4K。当然了如果数据很大的话，该缓存区实际分配的也会比默认值大。"

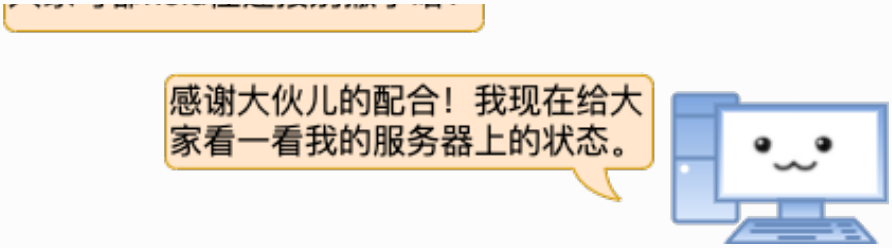


4.3.2 服务器端百万连接达成记



> “准备啥呢，还记得前面说过Linux对最大文件对象数量有限制，所以要想完成这个实验，得在用户级、系统级、进程级等位置把这个上限加大。我们实验目的是100W，这里都设置成110W，这个很重要！因为得保证做实验的时候其它基础命令例如ps，vi等是可用的。“


好了。我刚去改了一下我的文件对象数量上限配置。
兄弟们，大伙的内核参数ip_local_port_range也都改了吧?
已经改好了！
嗯。我们全都放到5W个了，设置的是10000 60000
SYN
SYN、ACK
SYN
我这里5W条连接已经全部建立成功了！
.....
看我老哥多牛批，我已经端口用尽报错了。老哥心不跳，气不喘的！
嗯，不愧是真大佬！我的连接也来啦！
.....
老哥，20位客户端同学的连接请求都建立完了！
.....
大家可都hold住连接别撒手哈！



活动连接数量确实达到了100W：

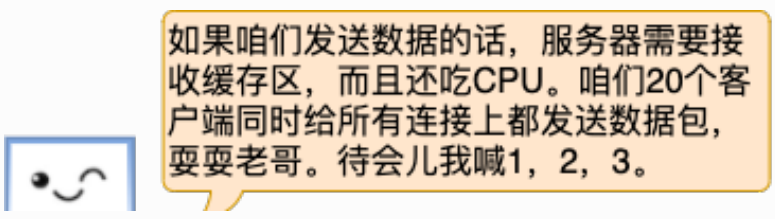
```
$ ss -n | grep ESTAB | wc -l
1000024
```

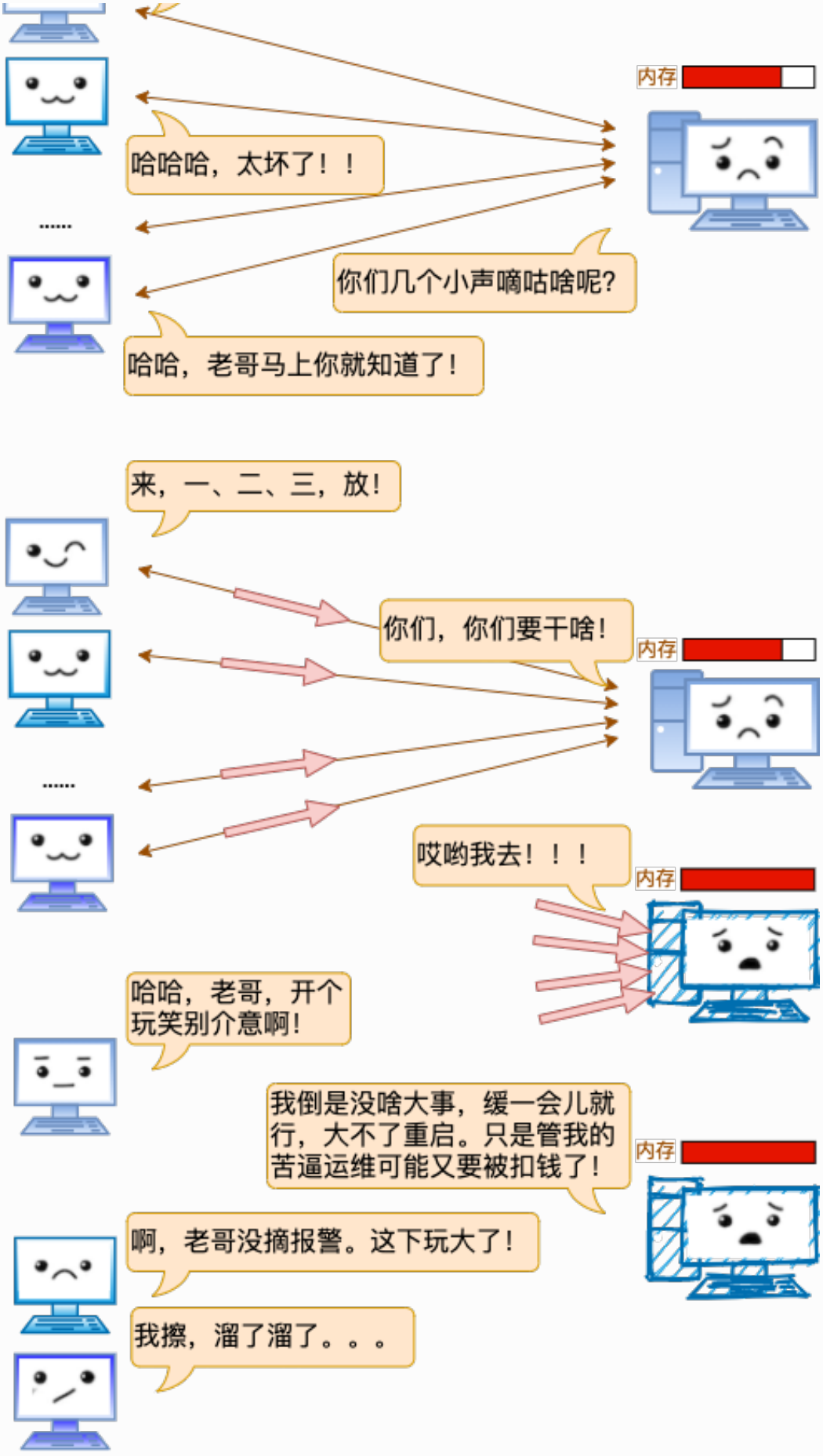
当前机器内存总共是3.9GB，其中内核Slab占用了3.2GB之多。 MemFree和Buffers加起来也只剩下100多MB了：

```
$ cat /proc/meminfo
MemTotal:         3922956 kB
MemFree:            96652 kB
MemAvailable:        6448 kB
Buffers:            44396 kB
......
Slab:            3241244KB kB
```

通过slabtop命令可以查看到densty、flip、sock_inode_cache、TCP四个内核对象都分别有100W个：

```
Active / Total Objects (% used)    : 4106309 / 4159066 (98.7%)
Active / Total Slabs (% used)      : 557252 / 557274 (100.0%)
Active / Total Caches (% used)     : 122 / 206 (59.2%)
Active / Total Size (% used)       : 2715631.63K / 2721707.92K (99.8%)
Minimum / Average / Maximum Object : 0.02K / 0.65K / 4096.00K

  OBJS ACTIVE  USE OBJ SIZE  SLABS OBJ/SLAB CACHE SIZE NAME
1009720 1008844  99%    0.19K  50486       20    201944K dentry
1002400 1001800  99%    0.19K  50120       20    200480K filp
1000195 1000195 100%    0.69K 200039        5    800156K sock_inode_cache
1000016 1000014  99%    1.62K 250004        4   2000032K TCP
 41477   13557  32%    0.06K    703       59      2812K size-64
 27888   25318  90%    0.03K    249      112       996K size-32
  8094    6433  79%    0.20K    426       19      1704K vm_area_struct
  7587    7581  99%    0.14K    281       27      1124K sysfs_dir_cache
  7392    5052  68%    0.05K     96       77       384K anon_vma_chain
```




内存
哈哈哈，太坏了！！
.....
你们几个小声嘀咕啥呢?
哈哈，老哥马上你就知道了！
来，一、二、三，放！
你们，你们要干啥！
内存
.....
哎哟我去！！！
内存
哈哈，老哥，开个玩笑别介意啊！
我倒是没啥大事，缓一会儿就行，大不了重启。只是管我的苦逼运维可能又要被扣钱了！
内存
啊，老哥没摘报警。这下玩大了！
我擦，溜了溜了。。。

4.3.3 总结

互联网后端的业务特点之一就是高并发. 但是一台服务器最大究竟能支持多少个TCP连接，这个问题似乎却又在困惑着很多同学。 希望今天过后，你能够将这个问题踩在脚下摩擦！

学习是一件痛苦的事情，尤其咱们号里很多读者朋友都是工作满一天了再来看我的技术号的文章的。我一直都在琢磨到底怎么样组织技术内容形式，能让大家理解起来更能省一点脑细胞呢。这篇服务器的最大并发数的文章是早就想发的，但是写了两三个版本都不满意。今天终于想出了一种让大家更容易理解的方式，算过了自己这关了。

如果您喜欢我的文章、并觉得它有用，期望您能不吝把它转发到你的朋友圈，技术群。或者哪怕是点个赞，点个再看都可以。触达更多的技术同学并收获大家的反馈将极大地提升彦飞的创作动力！

公众号：「开发内功修炼」

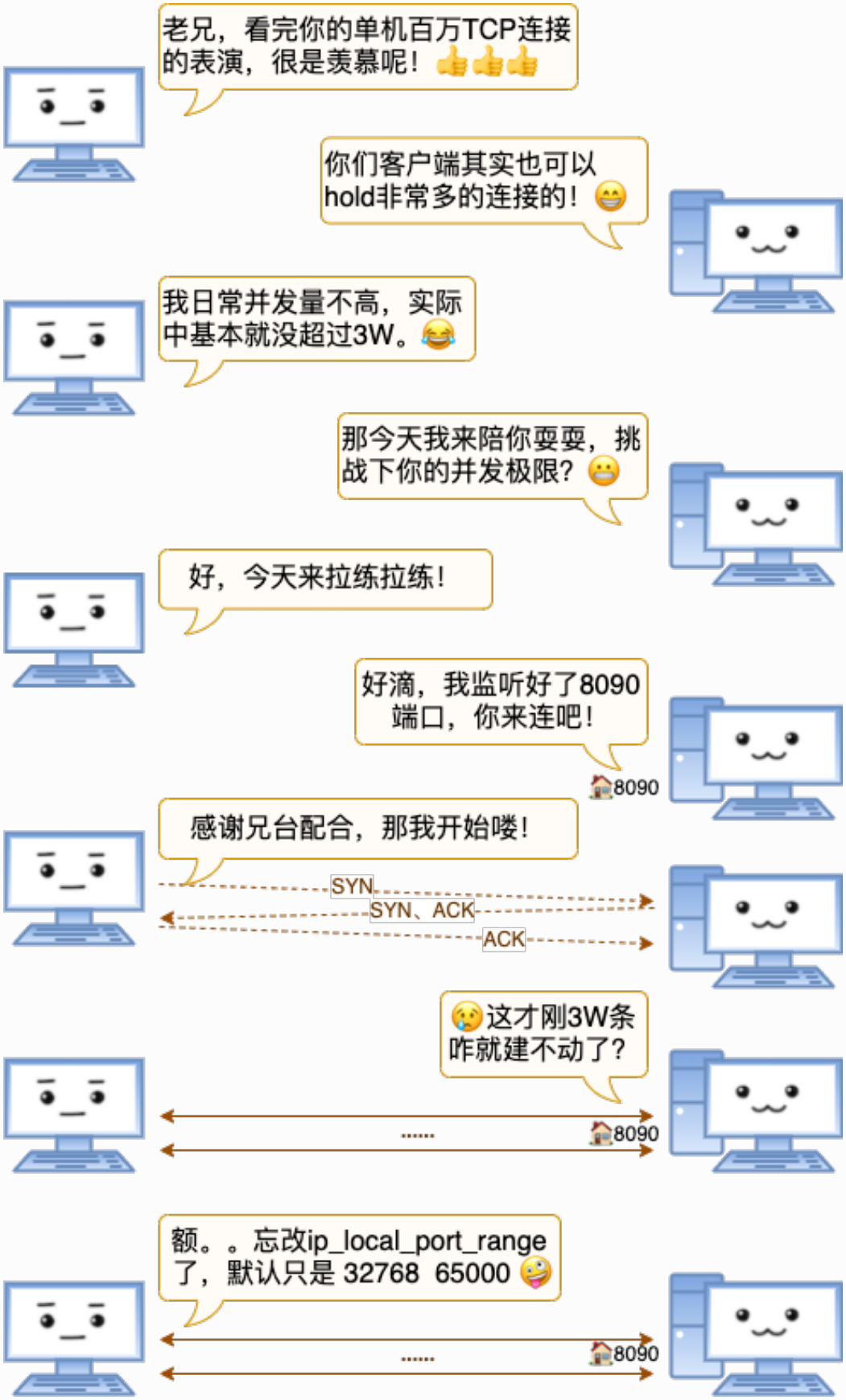
了解你的每一比特、 用好你的每一纳秒！

公众号

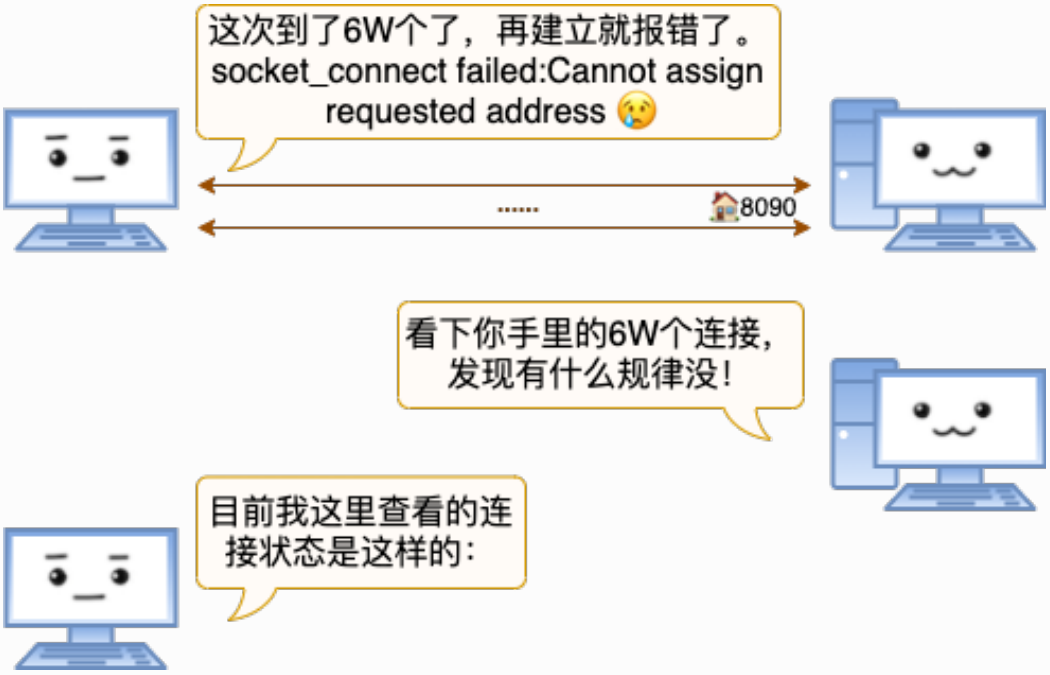
作者微信

4.4 一台客户端可以支撑多少条 TCP 连接

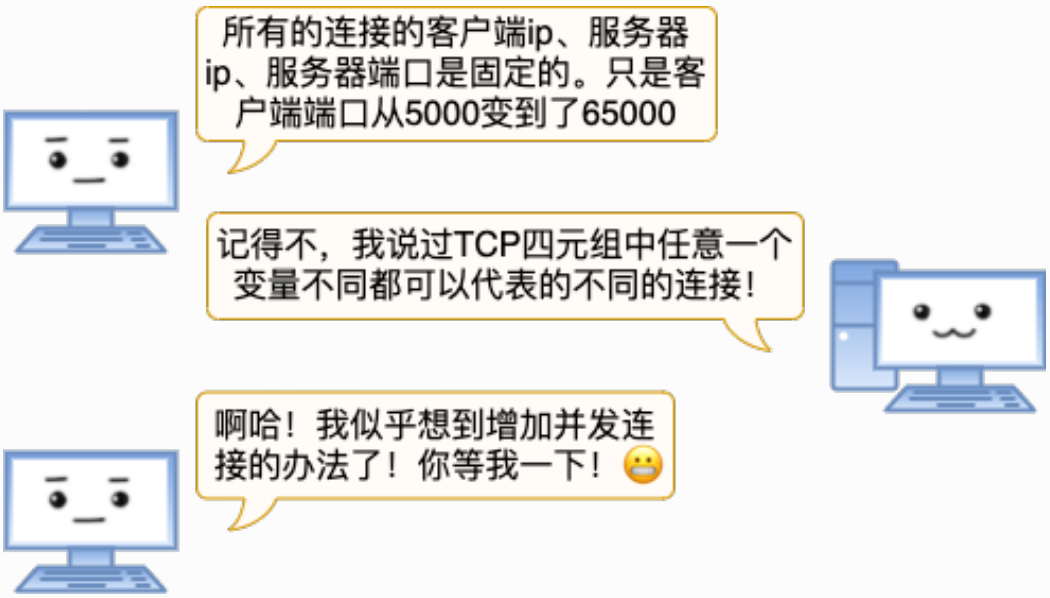
4.4.1 客户端机的疑问




```
echo "5000 65000" > /proc/sys/net/ipv4/ip_local_port_range
```

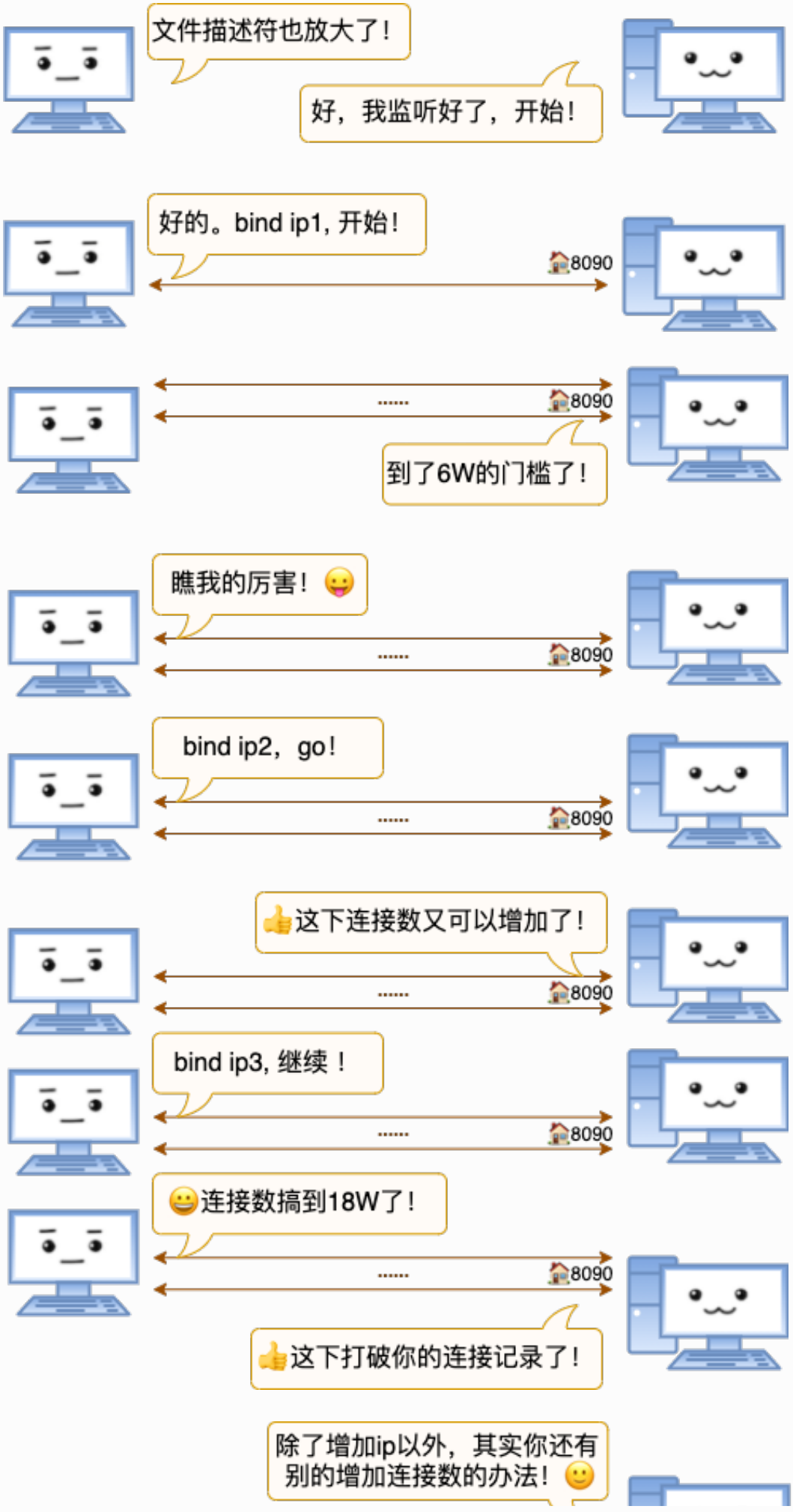


- 连接1： 192.168.1.101 5000 192.168.1.100 8090
- 连接2： 192.168.1.101 5001 192.168.1.100 8090
- 连接N： 192.168.1.101 ... 192.168.1.100 8090
- 连接6W： 192.168.1.101 65000 192.168.1.100 8090




```
//修改整个系统能打开的文件描述符为20W
echo 200000 > /proc/sys/fs/file-max


//修改所有用户每个进程可打开文件描述符为20W
#vi /etc/sysctl.conf
fs.nr_open=210000
#sysctl -p
#vi /etc/security/limits.conf
*  soft  nofile  200000
*  hard  nofile  200000
```

> 注意: limits中的hard limit不能超过nr_open, 所以要先改nr_open。而且最好是在sysctl.conf中改。避免重启的时候 hard limit生效了，nr_open不生效导致启动问题。

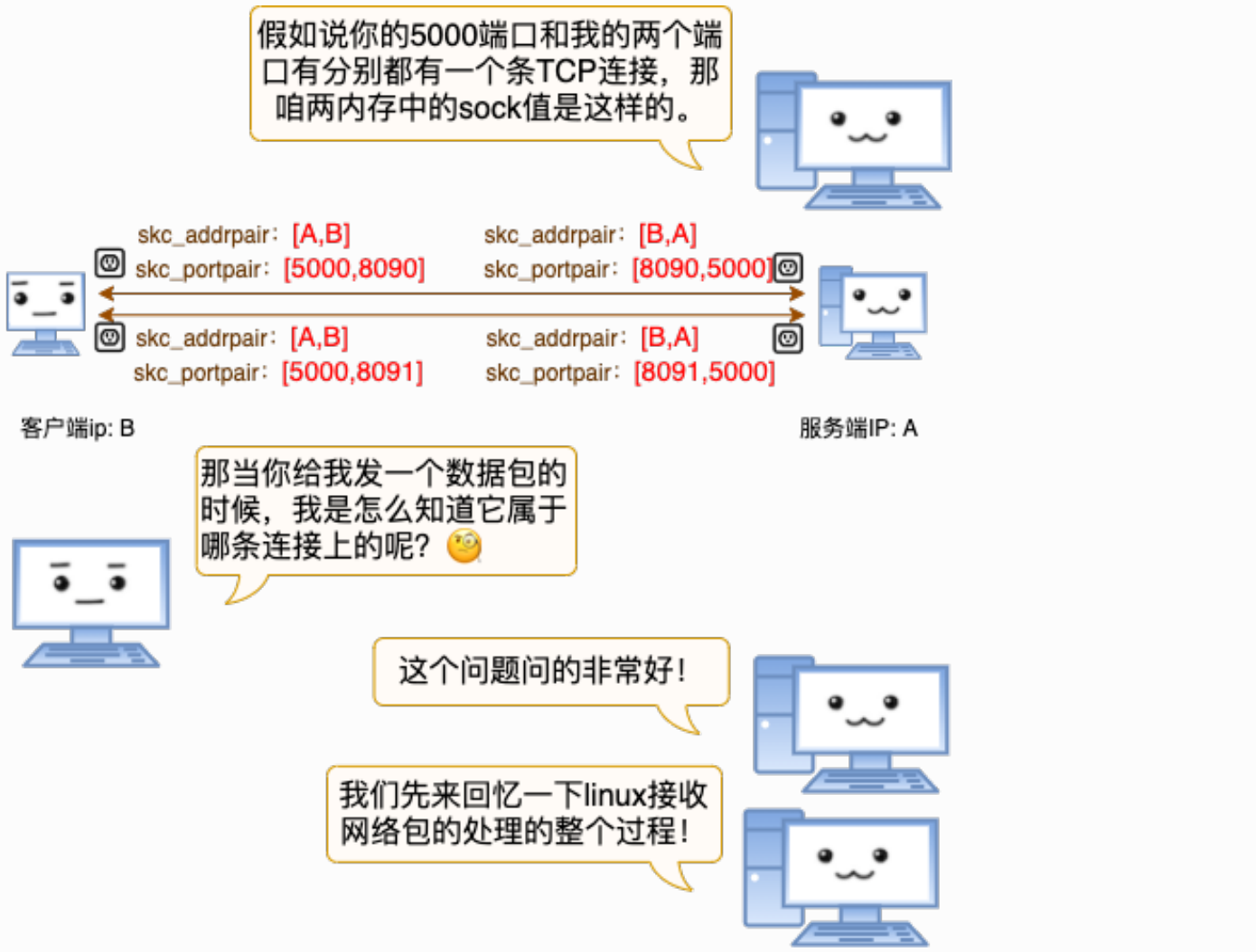

文件描述符也放大了！
好，我监听好了，开始！
好的。bind ip1, 开始！
8090
.....
8090
到了6W的门槛了！
瞧我的厉害！
.....
8090
bind ip2，go！
.....
8090
这下连接数又可以增加了！
.....
8090
bind ip3, 继续！
.....
8090
连接数搞到18W了！
.....
8090
这下打破你的连接记录了！
除了增加ip以外，其实你还有别的增加连接数的办法！

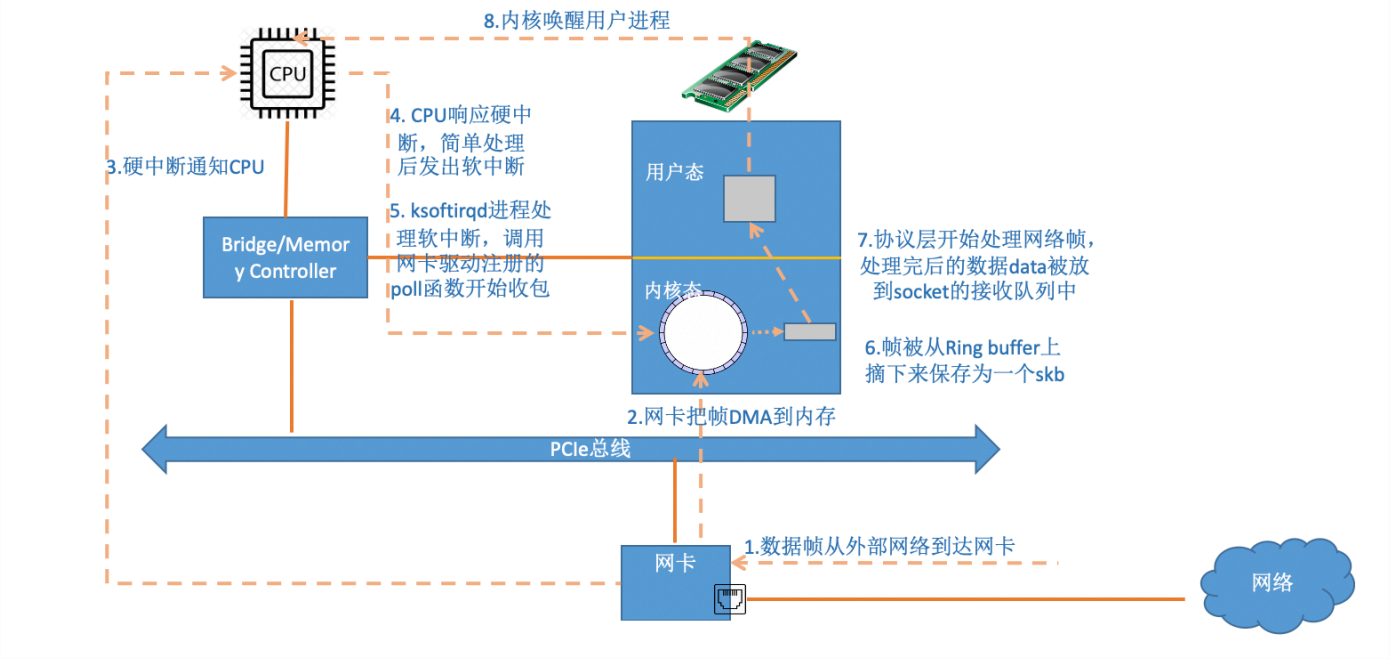

还有别的方法？
日常工作中你会只连接我一个server吗？
不会，我一般需要访问redis、mysql，kv等多种server
连接不同的server的话，你的端口是可以复用的！
老哥，我有点蒙。我一直觉得我们客户端一个端口就是只能是用于一个TCP连接呢。
你可能是把TCP连接和端口给混淆了！
TCP连接本质上是你和我分别在内存维护的一对socket内核对象，它们只要能找到对方就算一条连接！
那端口和连接的关系是啥？
端口就是socket对象找到它的另一半的信物之一。来我来带你看看socket对象的内核代码！
啊，内核代码，哥我头疼...
放心，我保证让你只使用到大一的c语言水平！！

“socket中有一个主要的数据结构sock_common，在它里面有两个联合体。”

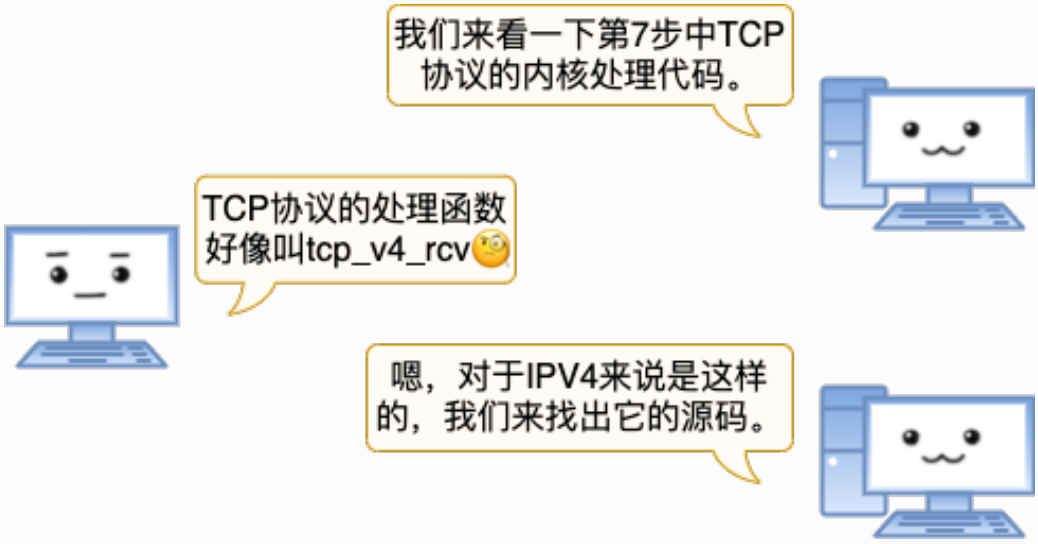
```
// file: include/net/sock.h
struct sock_common {
  union {
    __addrpair  skc_addrpair; //TCP连接IP对儿
    struct {
      __be32  skc_daddr;
      __be32  skc_rcv_saddr;
    };
  };
  union {
    __portpair  skc_portpair; //TCP连接端口对儿
    struct {
      __be16  skc_dport;
      __u16 skc_num;
    };
  };
  ......
}
```

“其中skc_addrpair记录的是TCP连接里的IP对儿，skc_portpair记录的是端口对儿。”



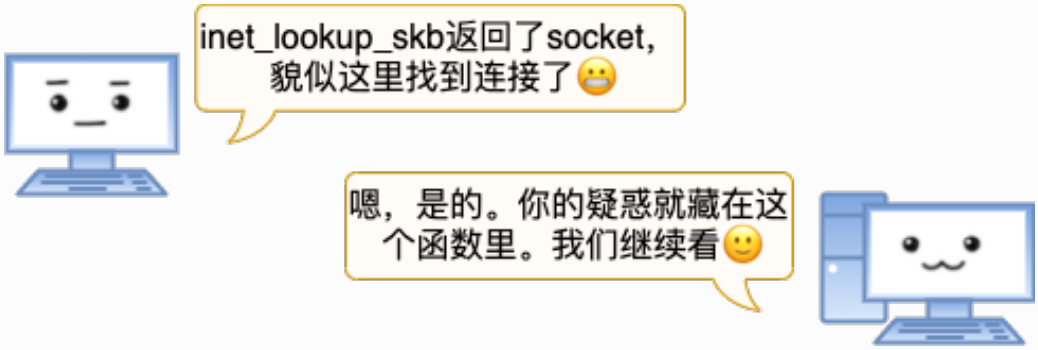


“在网络包到达网卡之后，依次经历DMA、硬中断、软中断等处理，最后被送到socket的接收队列中了。”



“对于TCP协议来说，协议处理的入口函数是 tcp_v4_rcv。我们看一下它的代码”

```
// file: net/ipv4/tcp_ipv4.c
int tcp_v4_rcv(struct sk_buff *skb)
{
  ......
  th = tcp_hdr(skb); //获取tcp header
  iph = ip_hdr(skb); //获取ip header

  sk = __inet_lookup_skb(&tcp_hashinfo, skb, th->source, th-
>dest);
  ......
}
```



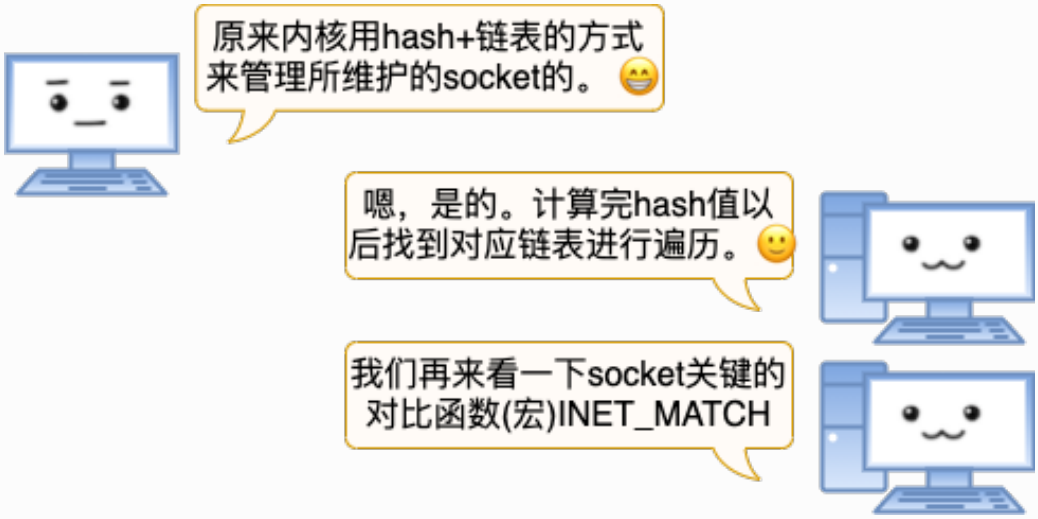

```
// file: include/net/inet_hashtables.h
static inline struct sock *__inet_lookup(struct net *net,
            struct inet_hashinfo *hashinfo,
            const __be32 saddr, const __be16 sport,
            const __be32 daddr, const __be16 dport,
```

```
            const int dif)
{
  u16 hnum = ntohs(dport);
  struct sock *sk = __inet_lookup_established(net, hashinfo,
        saddr, sport, daddr, hnum, dif);

  return sk ? : __inet_lookup_listener(net, hashinfo, saddr,
sport,
                daddr, hnum, dif);
}
```

“先判断有没有连接状态的socket，这会走到__inet_lookup_established函数中”

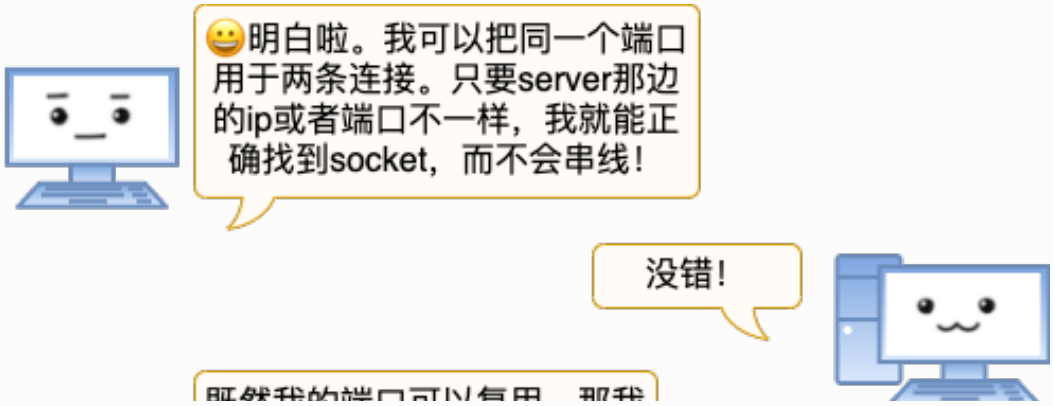
```
struct sock *__inet_lookup_established(struct net *net,
          struct inet_hashinfo *hashinfo,
          const __be32 saddr, const __be16 sport,
          const __be32 daddr, const u16 hnum,
          const int dif)
{
  //将源端口、目的端口拼成一个32位int整数
  const __portpair ports = INET_COMBINED_PORTS(sport, hnum);
  ......

  //内核用hash的方法加速socket的查找
  unsigned int hash = inet_ehashfn(net, daddr, hnum, saddr,
sport);
  unsigned int slot = hash & hashinfo->ehash_mask;
  struct inet_ehash_bucket *head = &hashinfo->ehash[slot];

begin:
  //遍历链表，逐个对比直到找到
  sk_nulls_for_each_rcu(sk, node, &head->chain) {
    if (sk->sk_hash != hash)
      continue;
    if (likely(INET_MATCH(sk, net, acookie,
              saddr, daddr, ports, dif))) {
      if (unlikely(!atomic_inc_not_zero(&sk->sk_refcnt)))
        goto begintw;
      if (unlikely(!INET_MATCH(sk, net, acookie,
            saddr, daddr, ports, dif))) {
        sock_put(sk);
        goto begin;
```

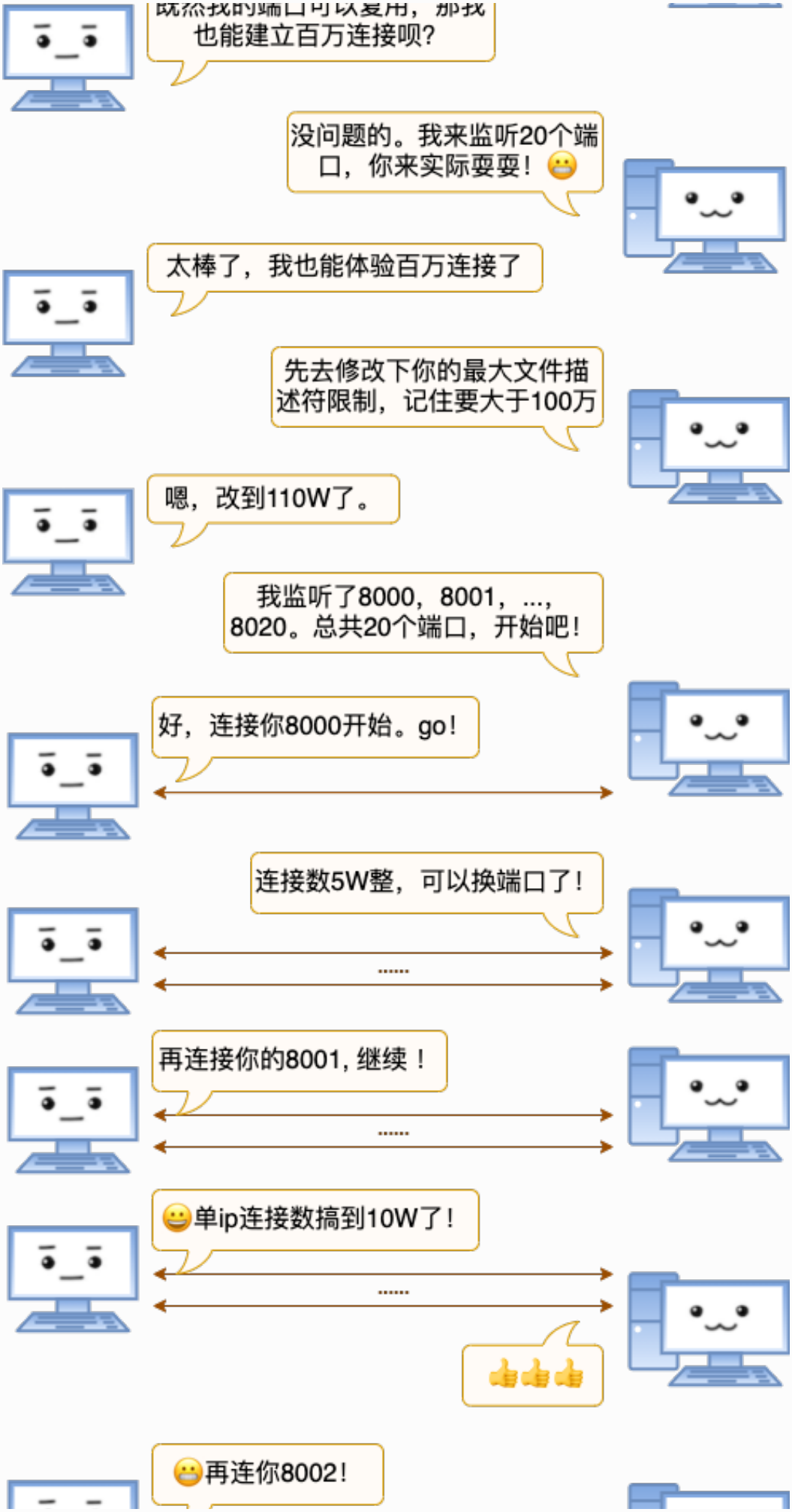
```
        }
        goto out;
      }
    }
  }
```

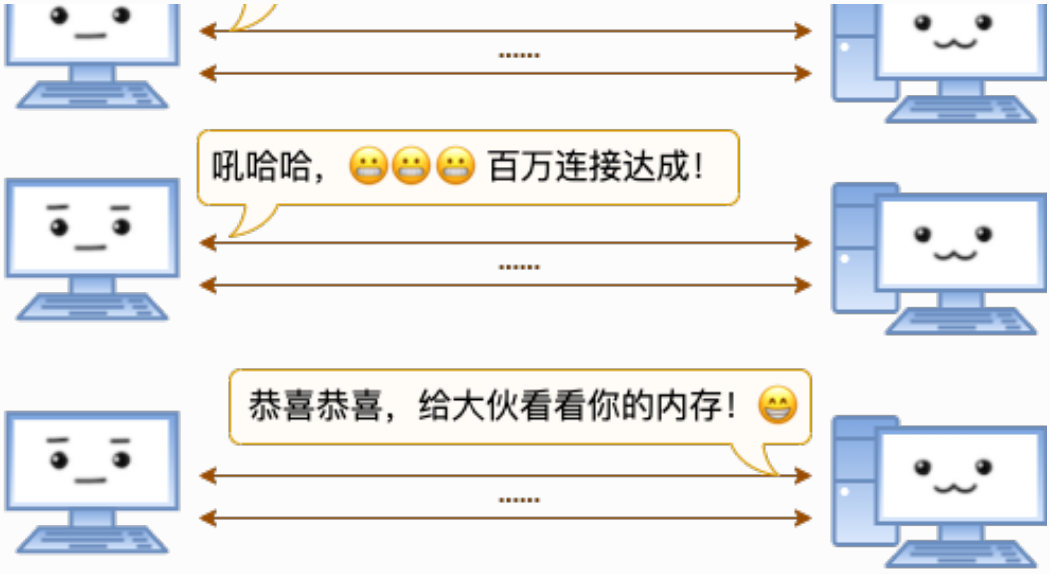



```
// include/net/inet_hashtables.h
#define INET_MATCH(__sk, __net, __cookie, __saddr, __daddr,
__ports, __dif) \
  ((inet_sk(__sk)->inet_portpair == (__ports))  &&     \
   (inet_sk(__sk)->inet_daddr == (__saddr)) &&     \
   (inet_sk(__sk)->inet_rcv_saddr == (__daddr)) &&     \
   (!(__sk)->sk_bound_dev_if  ||         \
     ((__sk)->sk_bound_dev_if == (__dif)))  &&     \
   net_eq(sock_net(__sk), (__net)))
```

“在INET_MATCH中将网络包tcp header中的__saddr、__daddr、__ports和Linux中的socket中inet_portpair、inet_daddr、inet_rcv_saddr进行对比。如果匹配socket就找到了。当然除了ip和端口，INET_MATCH还比较了其它一些东东，所以TCP还有五元组、七元组之类的说法。”

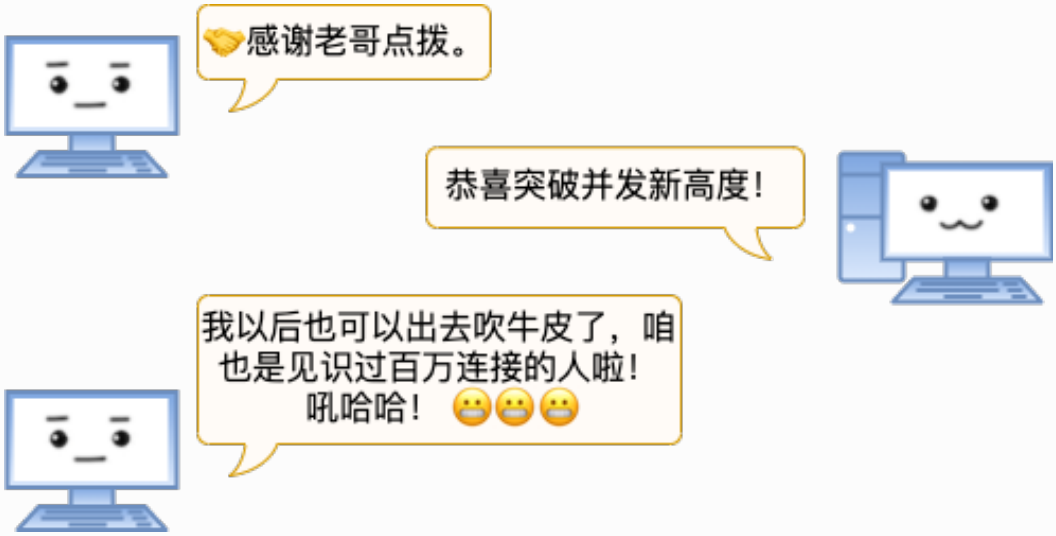



既然我的端口可以复用，那我也能建立百万连接呗?
没问题的。我来监听20个端口，你来实际耍耍！
太棒了，我也能体验百万连接了
先去修改下你的最大文件描述符限制，记住要大于100万
嗯，改到110W了。
我监听了8000，8001，...，8020。总共20个端口，开始吧！
好，连接你8000开始。go！
连接数5W整，可以换端口了！
.....
再连接你的8001，继续！
.....
单ip连接数搞到10W了！
.....
再连你8002！




```
# cat /etc/redhat-release
Red Hat Enterprise Linux Server release 6.2 (Santiago)

# ss -ant | grep ESTAB |wc -l
1000013

# cat /proc/meminfo
MemTotal:        3925408 kB
MemFree:           97748 kB
Buffers:           35412 kB
Cached:           119600 kB
......
Slab:            3241528 kB
```

```
Active / Total Objects (% used)    : 4182728 / 4197619 (99.6%)
Active / Total Slabs (% used)      : 557103 / 557107 (100.0%)
Active / Total Caches (% used)     : 111 / 206 (53.9%)
Active / Total Size (% used)       : 2718325.41K / 2720569.27K (99.9%)
Minimum / Average / Maximum Object : 0.02K / 0.65K / 4096.00K

  OBJS ACTIVE  USE OBJ SIZE  SLABS OBJ/SLAB CACHE SIZE NAME
1007560 1007560 100%    0.19K  50378       20    201512K dentry
1004420 1004340  99%    0.19K  50221       20    200884K filp
1000175 1000175 100%    0.69K 200035        5    800140K sock_inode_cache
1000024 1000024 100%    1.62K 250006        4   2000048K TCP
 60062   60008  99%    0.06K   1018       59      4072K tcp_bind_bucket
 24528   24471  99%    0.03K    219      112       876K size-32
 17575    9376  53%    0.10K    475       37      1900K buffer_head
 16549   16105  97%    0.20K    871       19      3484K vm_area_struct
 13806   12117  87%    0.06K    234       59       936K size-64
  9086    8594  94%    0.05K    118       77       472K anon_vma_chain
```



4.4.2 总结

客户端每建立一个连接就要消耗一个端口，所以很多同学当看到客户端机器上连接数一旦超过3W、5W就紧张的不行，总觉得机器要出问题了。

这篇文章第一版很早就写出来了，不过飞哥又打磨了有一周之多。在文中我们展示了一下TCP socket的少量内核代码。通过源码来看：

TCP连接就是在客户机、服务器上的一对儿的socket。它们都在各自内核对象上记录了双方的ip对儿、端口对儿（也就是我们常说的四元组），通过这个在通信时找到对方。

TCP连接发送方在发送网络包的时候，会把这份信息复制到IP Header上。网络包带着这份信物穿过互联网，到达目的服务器。目的服务器内核会按照 IP 包 header 中携带的信物（四元组）去匹配找到正确的socket(连接)。

在这个过程里我们可以看到，客户端的端口只是这个四元组里的一元而已。哪怕两条连接用的是同一个端口号，只要客户端ip不一样，或者是服务器不一样都不影响内核正确寻找到对应的连接，而不会串线！

所以在客户端增加TCP最大并发能力有两个方法。第一个办法，为客户端配置多个ip。第二个办法，连接多个不同的server。

> 不过这两个办法最好不要混用。因为使用多 IP 时，客户端需要bind。一旦bind之后，内核建立连接的时候就不会选择用过的端口了。bind函数会改变内核选择端口的策略~~

最后我们亲手实验证明了客户端也可以突破百万的并发量级。相信读过此文的你，以后再也不用再惧怕65535这个数字了。

4.5 实验单机100 万 TCP 连接的源码

很多读者在看完百万 TCP 连接的系列文章之后，反馈问我有没有测试源码。也想亲自动手做出来体验体验。这里为大家的实践精神点赞。

测试百万连接我用到的方案有两种，今天用一篇文章都给大家分享出来。

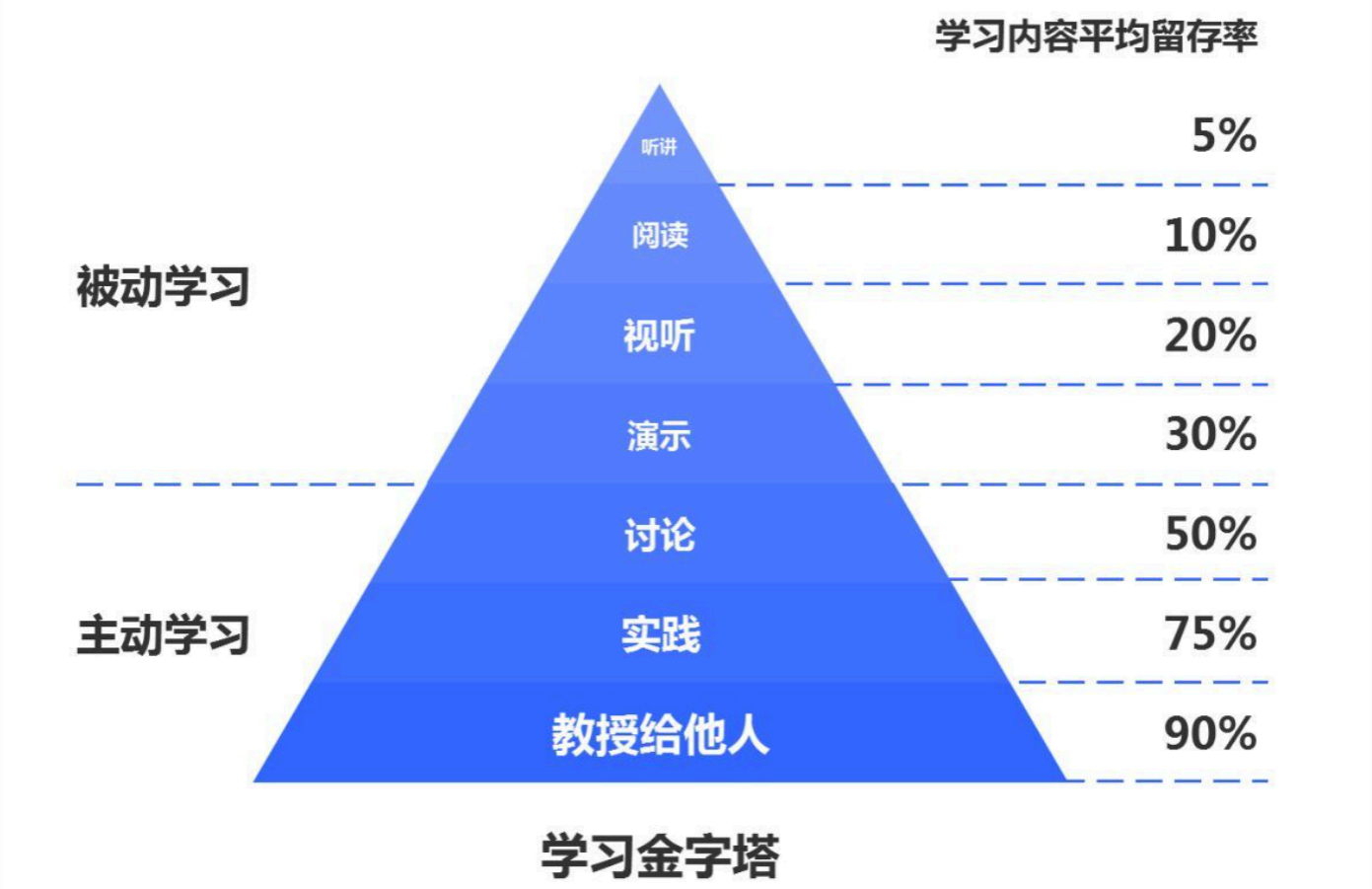
- 第一种是服务器端只开启一个进程，然后使用很多个客户端 ip 来连接
- 第二种是服务器开启多个进程，这样客户端就可以只使用一个 ip 即可

为了能让大部分同学都能用最低的时间成本达成百万连接结果，飞哥写了 c、java、php 三种版本的源码。两个方案对应的代码地址如下：

方案一： https://github.com/yanfeizhang/coder-kung-fu/tree/main/tests/network/test02
方案二： https://github.com/yanfeizhang/coder-kung-fu/tree/main/tests/network/test03

鉴于整个实验做起来还是有点小复杂，本文会配合从头到尾讲述每一个试验步骤，让大家动手起来更轻松。本文描述的步骤适用于任意一种语言。建议大家有空都动手耍耍。

为什么非得要进行实验呢，因为我觉得只有动手实践过，很多东西才能真正掌握。这里引用下埃德加 · 戴尔提出的学习金字塔理论图。根据他的研究结果可以看出，实践要比单纯的阅读效率要高好几倍。



所以我的文章中很多都是在介绍理论的同时夹杂着实际动手的实验结果，这种方式写文章投入的时间成本要高很多。但是，我觉得值！

4.5.1 TCP 并发理论基础

在做这个实验之前，需要你具备一些理论基础。这些在之前的文章都单独详细讲过，这里我把它们都简单再概括一下。

服务器理论最大并发数

TCP连接四元组是由源IP地址、源端口、目的IP地址和目的端口构成。

当四元组中任意一个元素发生了改变，那么就代表的是一条完全不同的新连接。

我们算下服务器上理论上能达成的最高并发数量。拿我们常用的 Nginx 举例，假设它的 IP 是 A，端口80。这样就只剩下源IP地址、源端口是可变的。

IP 地址是一个 32 位的整数，所以源 IP 最大有 2 的 32 次方这么多个。 端口是一个 16 位的整数，所以端口的数量就是 2 的 16 次方。

2 的 32 次方（ip数）× 2的 16 次方（port数）大约等于两百多万亿。

所以理论上，我们每个 server 可以接收的连接上限就是两百多万亿。（不过每条 TCP 连接都会消耗服务器内存，实践中绝不可能达到这个理论数字，稍后我们就能看到。）

客户端理论最大并发数

> 注意：这里的客户端是一个角色，并不具体指的是哪台机器。当你的 java/c/go 程序响应用户请求的时候，它是服务端。当它访问 redis/mysql 的时候，你这台机器就变成客户端角色了。这里假设我们一台机器只用来当客户端角色。

我们再算一下客户端的最大并发数的上限。

很多同学认为一台 Linux 客户端最多只能发起 64 k 条 TCP 连接。因为TCP 协议规定的端口数量有 65535 个，但是一般的系统里 1024 以下的端口都是保留的，所以没法用。可用的大约就是 64 k 个。

但实际上客户端可以发出的连接远远不止这个数。咱们看看以下两种情况

情况1： 这个 64 k 的端口号实际上说的是一个 ip 下的可用端口号数量。而一台 Linux 机器上是可以配置多个 IP 的。假如配置了 20 个 IP，那这样一台客户端机就可以发起 120 万多个 TCP 连接了。

情况2： 再退一步讲，假定一台 Linux 上确实只有一个 IP，那它就只能发起 64 k 条连接了吗？ 其实也不是的。

根据四元组的理论，只要服务器的 IP 或者端口不一样，即使客户端的 IP 和端口是一样的。这个四元组也是属于一条完全不同的新连接。

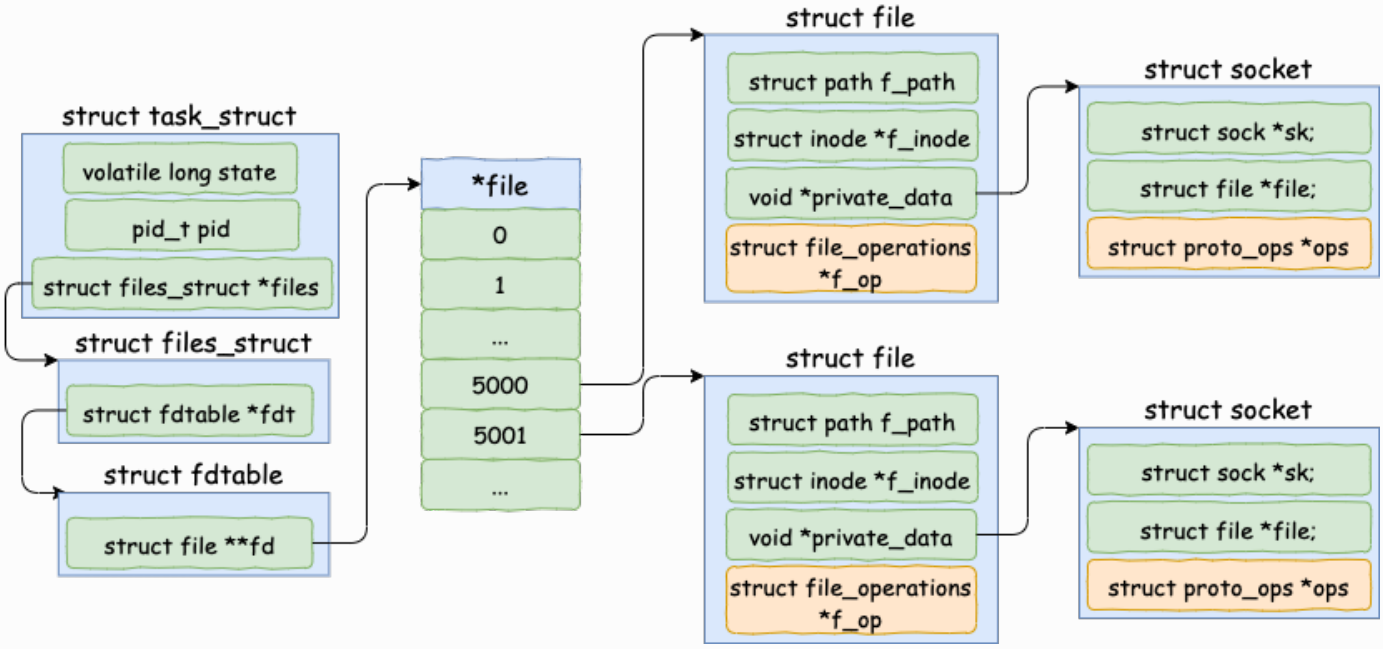
比如下面的两条连接里，虽然客户端的 IP 和端口完全一样，但由于服务器侧的端口不同，所以仍然是两条不同的连接。

- 连接1：客户端IP 10000 服务器IP 10000
- 连接2：客户端IP 10000 服务器IP 20000

所以一台客户端机器理论并发最大数是一个比服务器的两百万亿更大的一个天文数字（因为四元组里每一个元素都能变）。这里就不展开计算了，因为已经没有意义了。

Linux 最大文件描述符限制

linux 下一切皆文件，包括 socket。所以每当进程打开一个 socket 时候，内核实际上都会创建包括 file 在内的几个内核对象。该进程如果打开了两个 socket，那么它的内核对象结构如下图。



进程打开文件时消耗内核对象，换一句直白的话就是**打开文件对象吃内存**。所以linux系统出于安全角度的考虑，在多个位置都限制了可打开的文件描述符的数量，包括系统级、进程级、用户进程级。

- fs.file-max： 当前系统可打开的最大数量
- fs.nr_open： 当前系统单个进程可打开的最大数量
- nofile： 每个用户的进程可打开的最大数量

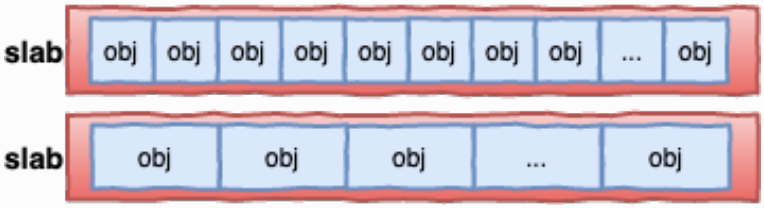
本文的实验要涉及对以上参数的修改。

TCP 连接的内存开销

介绍内存开销之前，需要先理解内核的内存使用方式。只有理解了这个，才能深刻理解 TCP 连接的内存开销。

Linux 内核和应用程序使用的是完全不同的两套机制。 Linux 给它的内核对象分配使用 SLAB 的方式。

一个 slab 一般由一个或者多个 Page 组成（每个 Page 一般为 4 KB）。在一个 slab 内只分配特定大小、甚至是特定的对象。这样当一个对象释放内存后，另一个同类对象可以直接使用这块内存。通过这种办法极大地降低了碎片发生的几率。



Linux 提供了 slabtop 命令来按照占用内存从大往小进行排列，这对我们查看内核对象的内存开销非常方便。

在 Linux 3.10.0 版本中，创建一个socket 需要消耗 densty、flip、sock_inode_cache、TCP 四个内核对象。这些对象加起来总共需要消耗大约 3 KB 多一点的内存。

> 如果连接上有数据收发的话，还需要消耗发送、接收缓存区。这两个缓存区占用内存影响因素比较多，既受收发数据的大小，也受 tcp_rmem、tcp_wmem 等内核参数，还取决于服务器进程能否及时接收（及时接收的话缓存区就能回收）。总之影响因素比较多，不同业务之间实际情况差别太大，比较复杂。所以不在本文讨论范围之内。

4.5.2 方案一：多 IP 客户端发起百万连接

了解了理论基础后，可以开始准备实验了。

本文实验需要准备两台机器。一台作为客户端，另一台作为服务器。如果你选用的是 c 或者 php 源码，这两台机器内存只要大于 4GB 就可以。 如果使用的是 Java 源码，内存要大于 6 GB。对 cpu 配置无要求，哪怕只有 1 个核都够用。

本方案中采用的方法是在一台客户端机器上配置多个 ip 的方式来发起所有的 tcp 连接请求。所以需要为你的客户端准备 20 个 IP，而且要确保这些 IP 在内网环境中没有被其它机器使用。如果实在选不出这些 IP，那么可以直接跳到方案 2。

> 除了用 20 个 IP 以外，也可以使用 20 台客户端。每个客户端发起 5 万个连接同时来连接这一个server。但是这个方法实际操作起来太困难了。

客户端机和服务器分别下载源码： https://github.com/yanfeizhang/coder-kung-fu/tree/main/tests/network/test02

下面我们来详细看每一个实验步骤。

调整客户端可用端口范围

默认情况下，Linux 只开启了 3 万多个可用端口。但我们今天的实验里，客户端一个进程要达到 5 万的并发。所以，端口范围的内核参数需要修改。

```
#vi /etc/sysctl.conf
net.ipv4.ip_local_port_range = 5000 65000
```

执行 sysctl -p 使之生效。

调整客户端最大可打开文件数

我们要测试百万并发，所以客户端的系统级参数 fs.file-max 需要加大到 100 万。另外 Linux 上还会存在一些其它的进程要使用文件，所以我们需要多打一些余量出来，直接设置到 110 万。

对于进程级参数 fs.nr_open 来说，因为我们开启 20 个进程来测，所以它设置到 60000 就够了。这些都在 /etc/sysctl.conf 中修改。

```
#vi /etc/sysctl.conf
fs.file-max=1100000
fs.nr_open=60000
```

sysctl -p 使得设置生效。并使用 sysctl -a 查看是否真正 work。

```
#sysctl -p
#sysctl -a
fs.file-max = 1100000
fs.nr_open = 60000
```

接着再加大用户进程的最大可打开文件数量限制（nofile）。这两个是用户进程级的，可以按不同的用户来区分配置。 这里为了简单，就直接配置成所有用户 * 了。每个进程最大开到 5 万个文件数就够了。同样预留一点余地，所以设置成 55000。 这些是在 /etc/security/limits.conf 文件中修改。

> 注意 hard nofile 一定要比 fs.nr_open 要小，否则可能导致用户无法登陆。

```
# vi /etc/security/limits.conf
*  soft  nofile  55000
*  hard  nofile  55000
```

配置完后，开个新控制台即可生效。 使用 ulimit 命令校验是否生效成功。

```
#ulimit -n
55000
```

服务器最大可打开文件句柄调整

服务器系统级参数 fs.file-max 也直接设置成 110 万。 另外由于这个方案中服务器是用单进程来接收客户端所有的连接的，所以进程级参数 fs.nr_open， 也一起改成 110 万。

```
#vi /etc/sysctl.conf
fs.file-max=1100000
fs.nr_open=1100000
```

sysctl -p 使得设置生效。并使用 sysctl -a 验证是否真正生效。

接着再加大用户进程的最大可打开文件数量限制（nofile），也需要设置到 100 万以上。

```
# vi /etc/security/limits.conf
*  soft  nofile  1010000
*  hard  nofile  1010000
```

配置完后，开个新控制台即可生效。 使用 ulimit 命令校验是否成功生效。

为客户端配置额外 20 个 IP

假设可用的 ip 分别是 CIP1，CIP2，.....，CIP20，你也知道你的子网掩码。

> 注意：这 20 个 ip 必须不能和局域网的其它机器冲突，否则会影响这些机器的正常网络包的收发。

在客户端机器上下载的源码目录 test02 中，找到你喜欢用的语言，进入到目录中找到 tool.sh。修改该 shell 文件，把 IPS 和 NETMASK 都改成你真正要用的。

为了确保局域网内没有这些 ip，最好先执行代码中提供的一个小工具来验证一下

```
make ping
```

当所有的 ip 的 ping 结果均为 false 时，进行下一步真正配置 ip 并启动网卡。

```
make ifup
```

使用 ifconfig 命令查看 ip 是否配置成功。

```
#ifconfig
eth0
eth0:0
eth0:1
...
eth:19
```

开始实验

ip 配置完成后，可以开始实验了。

在服务端中的 tool.sh 中可以设置服务器监听的端口，默认是 8090。启动 server

```
make run-srv
```

使用 netstat 命令确保 server 监听成功。

```
netstat -nlt | grep 8090
tcp  0   0.0.0.0:8090  0.0.0.0:*  LISTEN
```

在客户端的 tool.sh 中设置好服务器的 ip 和端口。然后开始连接

```
make run-cli
```

同时，另启一个控制台。使用 watch 命令来实时观测 ESTABLISH 状态连接的数量。

实验过程中不会一帆风顺，可能会有各种意外情况发生。 这个实验我前前后后至少花了有一周时间，所以你也不要第一次不成功就气馁。 遇到问题根据错误提示看下是哪里不对。然后调整一下，重新做就是了。 重做的时候需要重启客户端和服务器。

对于客户端，杀掉所有的客户端进程的方式是

```
make stop-cli
```

对于服务器来说由于是单进程的，所以直接 ctrl + c 就可以终止服务器进程了。 如果重启发现端口被占用，那是因为操作系统还没有回收，等一会儿再启动 server 就行。

当你发现连接数量超过 100 万的时候，你的实验就成功了。

```
watch "ss -ant | grep ESTABLISH"
1000013
```

这个时候别忘了查看一下你的服务端、客户端的内存开销。

先用 cat proc/meminfo 查看，重点看 slab 内存开销。

```
$ cat /proc/meminfo
MemTotal:        3922956 kB
MemFree:           96652 kB
MemAvailable:       6448 kB
Buffers:           44396 kB
......
Slab:           3241244KB kB
```

再用 slabtop 查看一下内核都是分配了哪些内核对象，它们每个的大小各自是多少。

```
Active / Total Objects (% used)    : 7092948 / 7100127 (99.9%)
Active / Total Slabs (% used)      : 318449 / 318449 (100.0%)
Active / Total Caches (% used)     : 84 / 115 (73.0%)
Active / Total Size (% used)       : 3684296.78K / 3688080.59K (99.9%)
Minimum / Average / Maximum Object : 0.01K / 0.52K / 8.00K

  OBJS ACTIVE  USE OBJ SIZE  SLABS OBJ/SLAB CACHE SIZE NAME
2204160 2204160 100%    0.19K 104960       21    419840K dentry
1067968 1067968 100%    0.06K  16687       64     66748K kmalloc-64
1013552 1013526  99%    0.25K  63347       16    253388K kmalloc-256
1000225 1000225  99%    0.62K  40009       25    640144K sock_inode_cache
1000048 1000048 100%    1.94K  62503       16   2000096K TCP
```

如果发现你的内核对象和上图不同，也不用惊慌。因为不同版本的 Linux 内核使用的内核对象名称和数量可能会有些许差异。

另外注意有的同学可能是碰到实验的时候连接非常慢的问题，那是因为你的客户端和服务器离的太近了。RTT 极有可能是小于 0.5 ms。这时可能瞬时会把连接队列打满，所以需要改改代码，连接的时候多 sleep 几次就行了。

结束实验

实验结束的时候，服务器进程直接 ctrl + c 取消运行就可以。客户端由于是多进程的，可能需要手工关闭一下。

```
make stop-cli
```

最后记得取消为实验临时配置的新 ip

```
make ifdown
```

4.5.3 方案二：单 IP 客户端发起百万连接

如果不纠结于非得让一个 Server 进程达成百万连接，只要是 Linux 服务器上总共能达到就行，那么就还有另外一个方法。

那就是在服务器端的 Linux 上开启多个 server，每个 server 都监听不同的端口。然后在客户端也启动多个进程来连接。每一个客户端进程都连接不同的 server 端口。客户端上发起连接时只要不调用 bind，那么一个特定的端口是可以在不同的 server 之间复用的。

同样，实验源码也有 c、java、php 三个语言的版本。
准备好两台机器。一台作为客户端，另一台作为服务器。分别下载如下源码：

https://github.com/yanfeizhang/coder-kung-fu/tree/main/tests/network/test03

调整可用端口范围

同方案一，客户端机端口范围的内核参数也是需要修改的。

```
#vi /etc/sysctl.conf
net.ipv4.ip_local_port_range = 5000 65000
```

执行 sysctl -p 使之生效。

客户端加大最大可打开文件数

同方案一，客户端的 fs.file-max 也需要加大到 110 万。进程级的参数 fs.nr_open 设置到 60000。

```
#vi /etc/sysctl.conf
fs.file-max=1100000
fs.nr_open=60000
```

sysctl -p 使得设置生效。并使用 sysctl -a 查看是否真正生效

客户端的 nofile 设置成 55000

```
# vi /etc/security/limits.conf
*  soft  nofile  55000
*  hard  nofile  55000
```

配置完后，开个新控制台即可生效。

服务器最大可打开文件句柄调整

同方案一，调整服务器最大可打开文件数。不过方案二服务端是分了 20 个进程，所以 fs.nr_open 改成 60000 就足够了。

```
#vi /etc/sysctl.conf
fs.file-max=1100000
fs.nr_open=60000
```

sysctl -p 使得设置生效。并使用 sysctl -a 验证。

接着再加大用户进程的最大可打开文件数量限制（nofile），这个也是 55000。

```
# vi /etc/security/limits.conf
*  soft  nofile  55000
*  hard  nofile  55000
```

> 再次提醒： hard nofile 一定要比 fs.nr_open 要小，否则可能导致用户无法登陆。

配置完后，开个新控制台即可生效。

开始实验

启动 server

```
make run-srv
```

使用 netstat 命令确保 server 监听成功。

```
netstat -nlt | grep 8090
tcp  0  0  0.0.0.0:8100  0.0.0.0:*  LISTEN
tcp  0  0  0.0.0.0:8101  0.0.0.0:*  LISTEN
......
tcp  0  0  0.0.0.0:8119  0.0.0.0:*  LISTEN
```

回到客户端机器，修改 tool.sh 中的服务器 ip。端口会自动从 tool.sh 中加载。然后开始连接

```
make run-cli
```

同时，另启一个控制台。使用 watch 命令来实时观测 ESTABLISH 状态连接的数量。

期间如果做失败了，需要重新开始的话，那需要先杀掉所有的进程。在客户端执行 `make stop-cli`，在服务器端是执行 `make stop-srv`。重新执行上述步骤。

当你发现连接数量超过 100 万的时候，你的实验就成功了。

```
watch "ss -ant | grep ESTABLISH"
1000013
```

记得查看同样一下你的服务端、客户端的内存开销。

```
# cat /etc/redhat-release
Red Hat Enterprise Linux Server release 6.2 (Santiago)

# ss -ant | grep ESTAB |wc -l
1000013

# cat /proc/meminfo
MemTotal:        3925408 kB
MemFree:           97748 kB
Buffers:           35412 kB
Cached:           119600 kB
......
Slab:            3241528 kB
```

再用 slabtop 查看一下 top 内核对象。

```
Active / Total Objects (% used)    : 4182728 / 4197619 (99.6%)
Active / Total Slabs (% used)      : 557103 / 557107 (100.0%)
Active / Total Caches (% used)     : 111 / 206 (53.9%)
Active / Total Size (% used)       : 2718325.41K / 2720569.27K (99.9%)
Minimum / Average / Maximum Object : 0.02K / 0.65K / 4096.00K

  OBJS ACTIVE  USE OBJ SIZE  SLABS OBJ/SLAB CACHE SIZE NAME
1007560 1007560 100%    0.19K  50378       20    201512K dentry
1004420 1004340  99%    0.19K  50221       20    200884K filp
1000175 1000175 100%    0.69K 200035        5    800140K sock_inode_cache
1000024 1000024 100%    1.62K 250006        4   2000048K TCP
 60062   60008  99%    0.06K   1018       59      4072K tcp_bind_bucket
 24528   24471  99%    0.03K    219      112       876K size-32
 17575    9376  53%    0.10K    475       37      1900K buffer_head
 16549   16105  97%    0.20K    871       19      3484K vm_area_struct
 13806   12117  87%    0.06K    234       59       936K size-64
  9086    8594  94%    0.05K    118       77       472K anon_vma_chain
```

实验结束的时候，记得`make stop-cli`结束所有客户端进程，`make stop-srv`结束所有服务器进程。

4.5.4 最后多扯一点

经过网络篇这几篇文章的学习，相信大家已经不会再觉得百万并发有多么的高深了。

并发只是描述服务器端程序的指标之一，并不是全部

一条不活跃的 TCP 连接开销仅仅只是 3 KB 多点的内存而已。现代的一台服务器都上百 GB 的内存，如果只是说并发，单机千万（C10000K）都可以。

在互联网服务器端应用场景里，除了一些基于长连接的 push 场景以外。其它的大部分业务里讨论并发都要和业务结合起来。抛开业务逻辑单纯地说并发多高其实并没有太大的意义。

因为在这些场景中，**服务器开销大头往往都不是连接本身，而是在每条连接上的数据收发、以及请求业务逻辑处理**。

这就好比你作为一个开发同学，在公司内建立了和十个产品经理的业务联系。这并不代表你的并发能力真的能达到十，很有可能是一位产品的需求就能把你的时间打满。

另外就是不同的业务之间，单纯比较并发也不一定有意义。

假设同样的服务器配置，A 业务的单机能支撑 1 万并发，B 业务只能撑 1 千。这也并不一定就说明 A 业务的性能比 B 业务好。因为 B 业务的请求处理逻辑可能是相当的复杂，比如要进行复杂的压缩、加解密。而 A 业务的处理很简单内存读取个变量就返回了。

扩展说一下，本文配套代码中仅仅只是作为测试使用，所以写的比较简单。是直接阻塞式地 accept，将接收过来的新连接也雪藏了起来，并没有读写发生。

如果在你的项目实践中真的确实需要百万条 TCP 连接，那么一般来说还需要高效的 IO 事件管理。在 c 语言中，就是直接用 epoll 系列的函数来管理。对于 Java 语言来说，就是 NIO。（Golang 中不用操心，net 包中把 IO 事件管理都已经封装好了）

另外飞哥建立了一个技术群，欢迎大家到群里来进一步交流。先加我微信 (zhangyanfei748527)，我来拉大家。

五、TCP 连接的时间开销

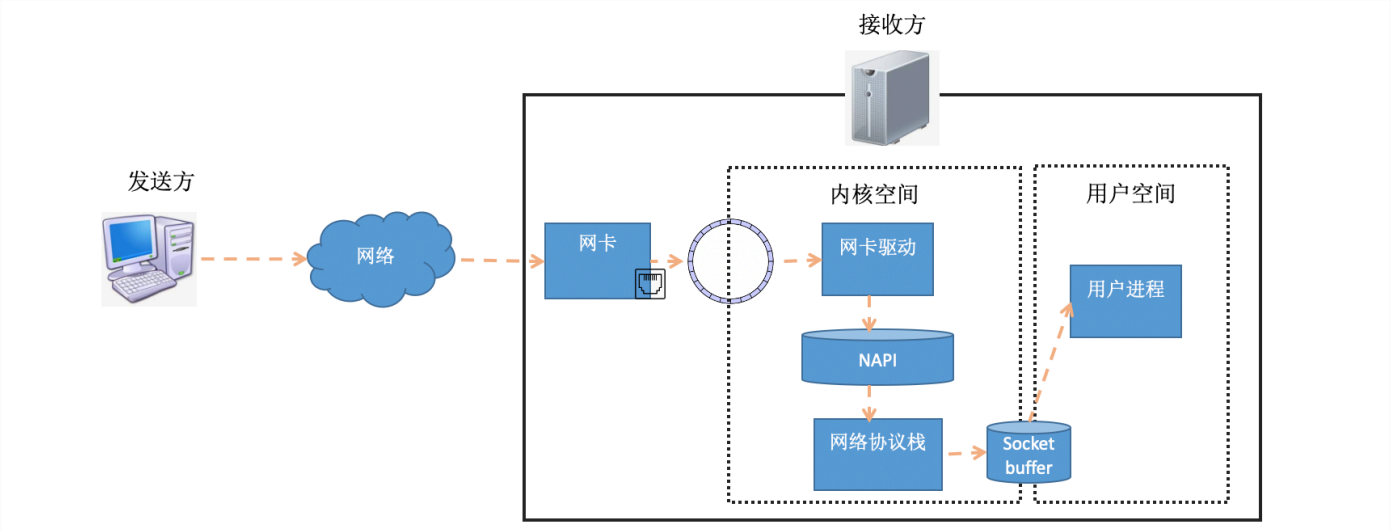
在互联网后端日常开发接口的时候中，不管你使用的是C、Java、PHP还是Golang，都避免不了需要调用mysql、redis等组件来获取数据，可能还需要执行一些rpc远程调用，或者再调用一些其它restful api。 在这些调用的底层，基本都是在使用TCP协议进行传输。这是因为在传输层协议中，TCP协议具备可靠的连接，错误重传，拥塞控制等优点，所以目前应用比UDP更广泛一些。

相信你也一定听闻过TCP也存在一些缺点，那就是老生常谈的开销要略大。但是各路技术博客里都在单单说开销大、或者开销小，而少见不给出具体的量化分析。不客气一点，这都是营养不大的废话。经过日常工作的思考之后，我更想弄明白的是，开销到底多大。一条TCP连接的建立需要耗时延迟多少，是多少毫秒，还是多少微秒？能不能有一个哪怕是粗略的量化估

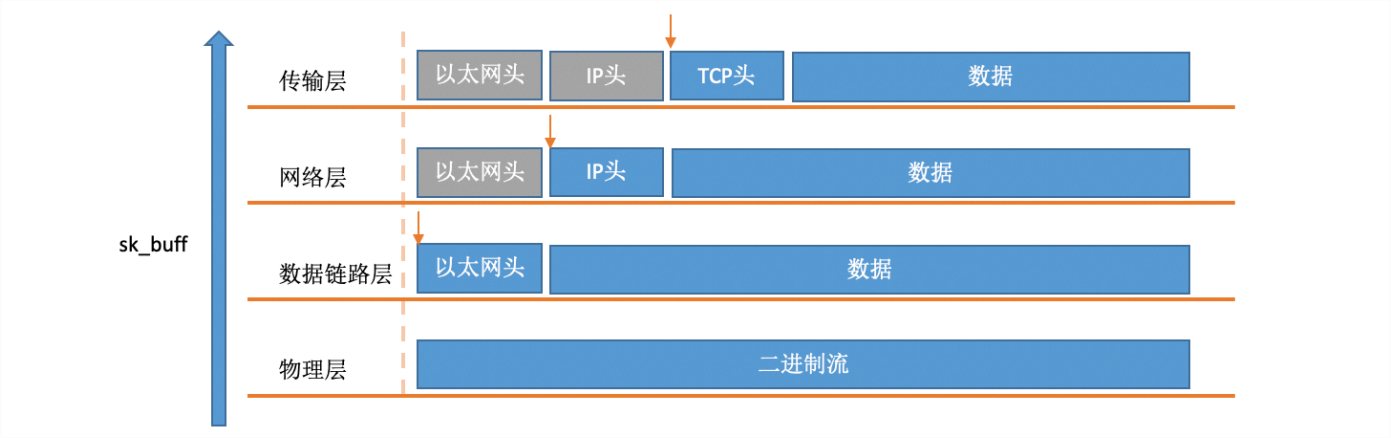
计？当然影响TCP耗时的因素有很多，比如网络丢包等等。我今天只分享我在工作实践中遇到的比较高发的各种情况。

5.1 正常TCP连接建立过程

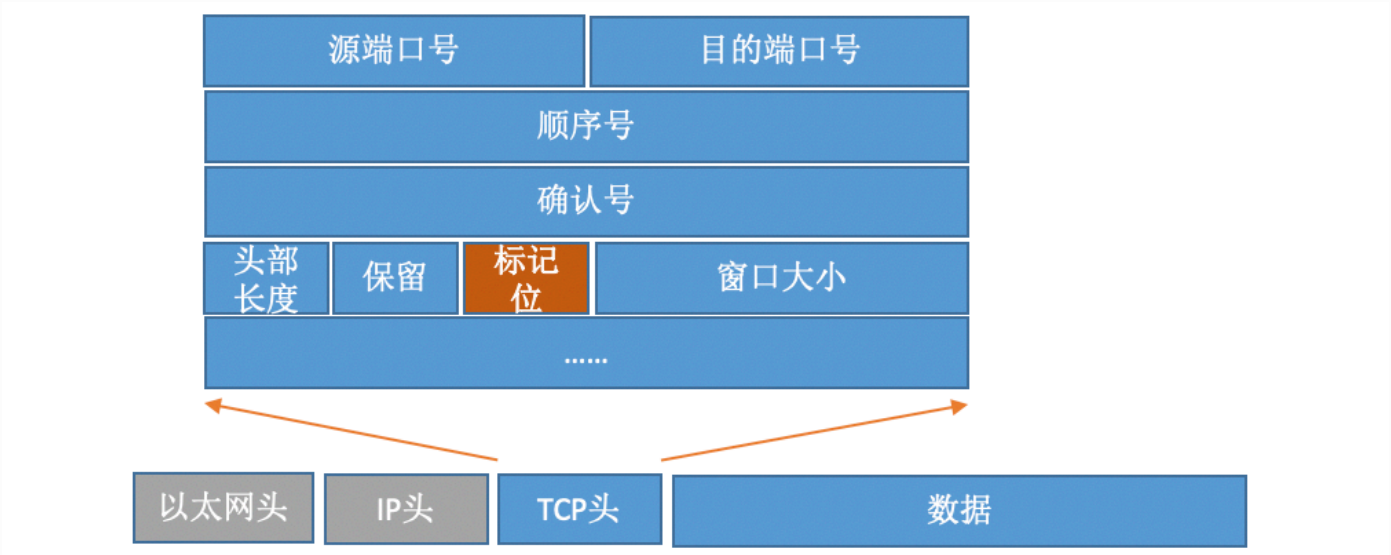
要想搞清楚TCP连接的建立耗时，我们需要详细了解连接的建立过程。在前文《图解Linux网络包接收过程》中我们介绍了数据包在接收端是怎么被接收的。数据包从发送方出来，经过网络到达接收方的网卡。在接收方网卡将数据包DMA到RingBuffer后，内核经过硬中断、软中断等机制来处理（如果发送的是用户数据的话，最后会发送到socket的接收队列中，并唤醒用户进程）。



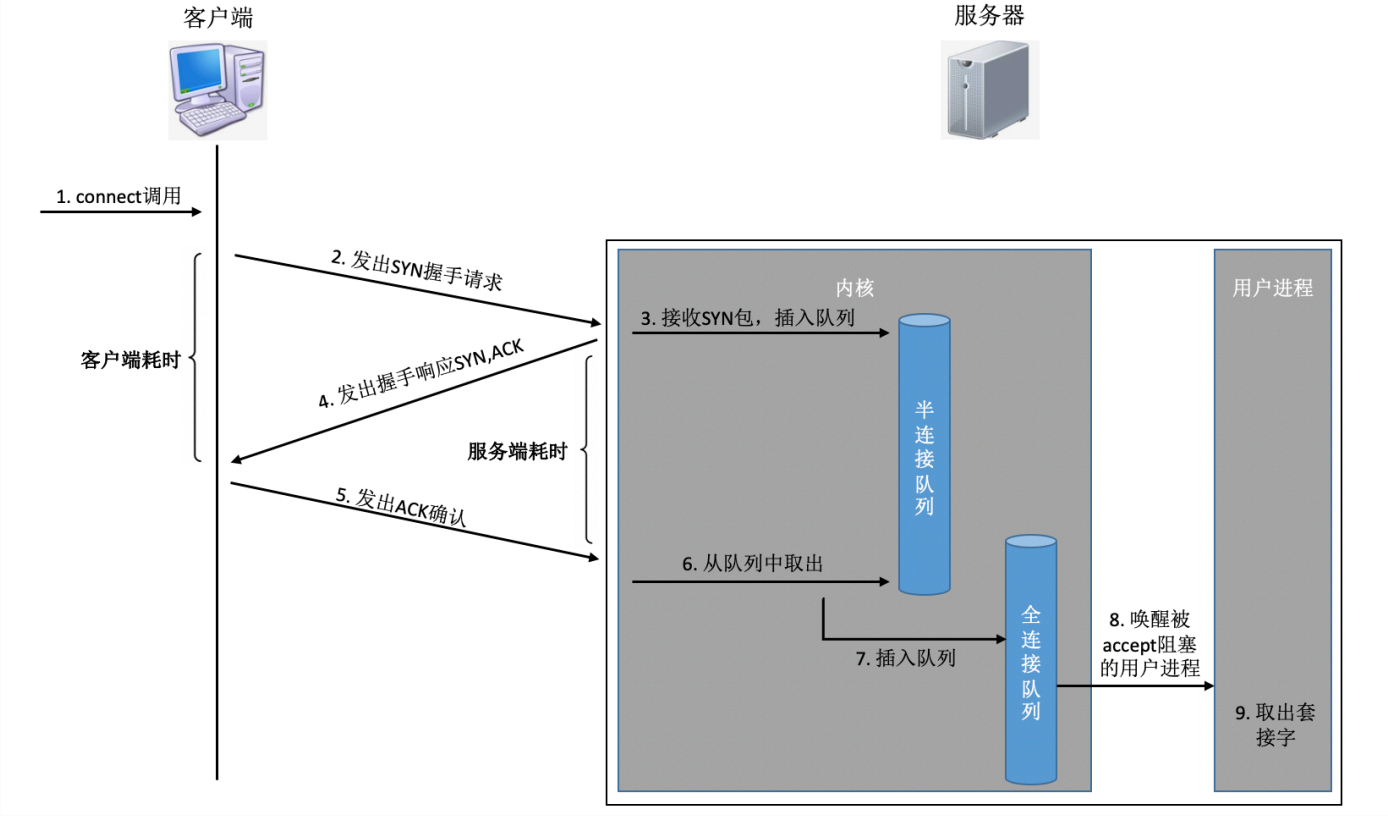
在软中断中，当一个包被内核从RingBuffer中摘下来的时候，在内核中是用`struct sk_buff`结构体来表示的(参见内核代码`include/linux/skbuff.h`)。其中的data成员是接收到的数据，在协议栈逐层被处理的时候，通过修改指针指向data的不同位置，来找到每一层协议关心的数据。



对于TCP协议包来说，它的Header中有一个重要的字段-flags。如下图：



通过设置不同的标记为，将TCP包分成SYNC、FIN、ACK、RST等类型。客户端通过connect系统调用命令内核发出SYNC、ACK等包来实现和服务器TCP连接的建立。在服务器端，可能会接收许许多多的连接请求，内核还需要借助一些辅助数据结构-半连接队列和全连接队列。我们来看一下整个连接过程：



在这个连接过程中，我们来简单分析一下每一步的耗时

- 客户端发出SYNC包：客户端一般是通过connect系统调用来发出SYN的，这里牵涉到本机的系统调用和软中断的CPU耗时开销
- **SYN传到服务器**：SYN从客户端网卡被发出，开始"跨过山和大海，也穿过人山人海~~~"，这是一次长途远距离的网络传输
- 服务器处理SYN包：内核通过软中断来收包，然后放到半连接队列中，然后再发出SYN/ACK响应。又是CPU耗时开销
- **SYC/ACK传到客户端**：SYC/ACK从服务器端被发出后，同样跨过很多山、可能很多大海来到客户端。又一次长途网络跋涉
- 客户端处理SYN/ACK：客户端内核收包并处理SYN后，经过几us的CPU处理，接着发出ACK。同样是软中断处理开销
- **ACK传到服务器**：和SYN包，一样，再经过几乎同样远的路，传输一遍。 又一次长途网络跋涉
- 服务端收到ACK：服务器端内核收到并处理ACK，然后把对应的连接从半连接队列中取出来，然后放到全连接队列中。一次软中断CPU开销
- 服务器端用户进程唤醒：正在被accpet系统调用阻塞的用户进程被唤醒，然后从全连接队列中取出来已经建立好的连接。一次上下文切换的CPU开销

以上几步操作，可以简单划分为两类：

- 第一类是内核消耗CPU进行接收、发送或者是处理，包括系统调用、软中断和上下文切换。它们的耗时基本都是几个us左右。具体的分析过程可以参见《一次系统调用开销到底有多大？》、《软中断会吃掉你多少CPU？》、《进程/线程切换会用掉你多少

CPU？》这三篇文章。

- 第二类是网络传输，当包被从一台机器上发出以后，中间要经过各式各样的网线、各种交换机路由器。所以网络传输的耗时相比本机的CPU处理，就要高的多了。根据网络远近一般在几ms~到几百ms不等。。

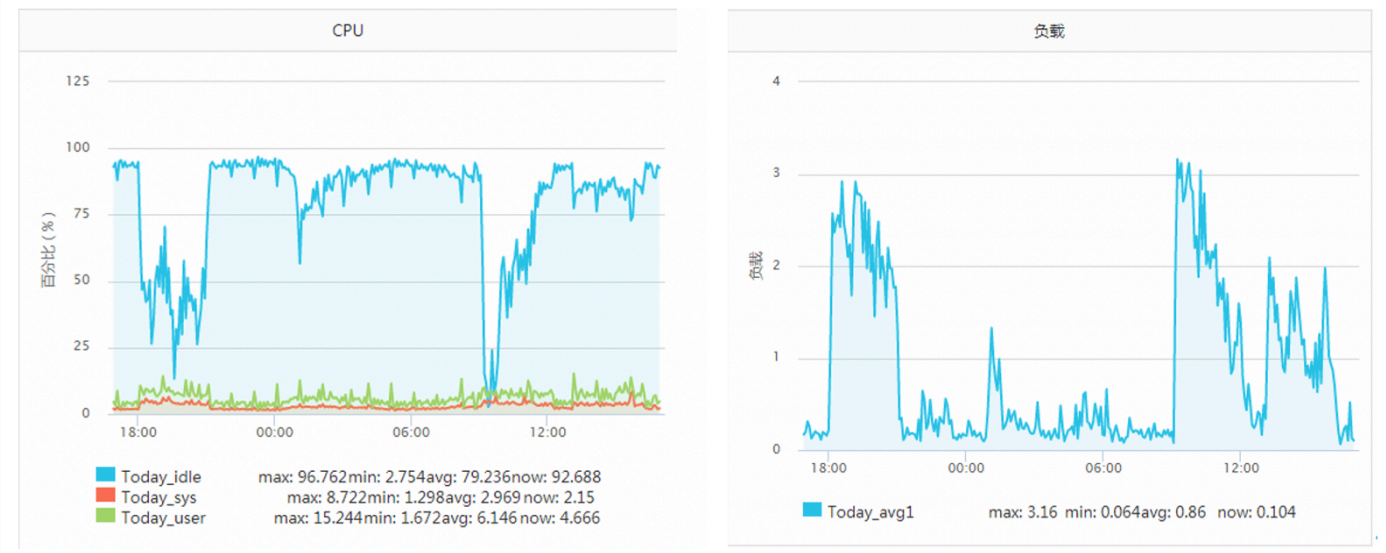
1ms就等于1000us，因此网络传输耗时比双端的CPU开销要高1000倍左右，甚至更高可能还到100000倍。所以，在正常的TCP连接的建立过程中，一般可以考虑网络延时即可。一个RTT指的是包从一台服务器到另外一台服务器的一个来回的延迟时间。所以从全局来看，TCP连接建立的网络耗时大约需要三次传输，再加上少许的双方CPU开销，总共大约比1.5倍RTT大一点点。不过从客户端视角来看，只要ACK包发出了，内核就认为连接是建立成功了。所以如果在客户端打点统计TCP连接建立耗时的话，只需要两次传输耗时-既1个RTT多一点的时间。（对于服务器端视角来看同理，从SYN包收到开始算，到收到ACK，中间也是一次RTT耗时）

5.2 TCP 连接建立时的异常情况

上一节可以看到在客户端视角，在正常情况下一次TCP连接总的耗时也就就大约是一次网络RTT的耗时。如果所有的事情都这么简单，我想我的这次分享也就没有必要了。事情不一定总是这么美好，总会有意外发生。在某些情况下，可能会导致连接时的网络传输耗时上涨、CPU处理开销增加、甚至是连接失败。现在我们说一下我在线上遇到过的各种沟沟坎坎。

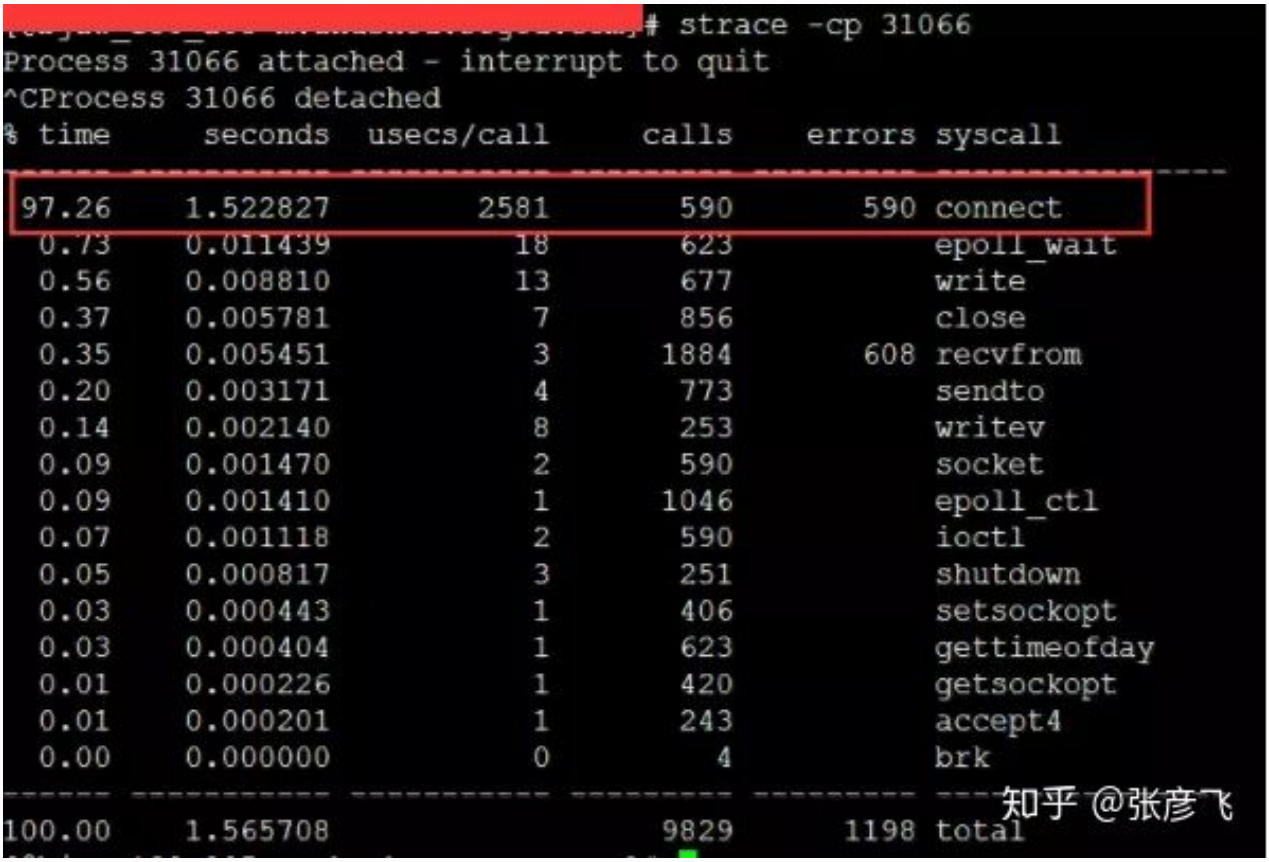
客户端connect系统调用耗时失控

正常一个系统调用的耗时也就是几个us（微秒）左右。但是在《追踪将服务器CPU耗光的凶手!》一文中笔者的一台服务器当时遇到一个状况，某次运维同学转达过来说该服务CPU不够用了，需要扩容。当时的服务器监控如下图：



该服务之前一直每秒抗2000左右的qps，CPU的idel一直有70%+。怎么突然就CPU一下就不够用了呢。而且更奇怪的是CPU被打到谷底的那一段时间，负载却并不高（服务器为4核机器，负载3-4是比较正常的）。 后来经过排查以后发现当TCP客户端TIME_WAIT有30000左右，导致可用端口不是特别充足的时候，connect系统调用的CPU开销直接上涨了100多倍，每次耗时达到了2500us（微秒），达到了毫秒级别。

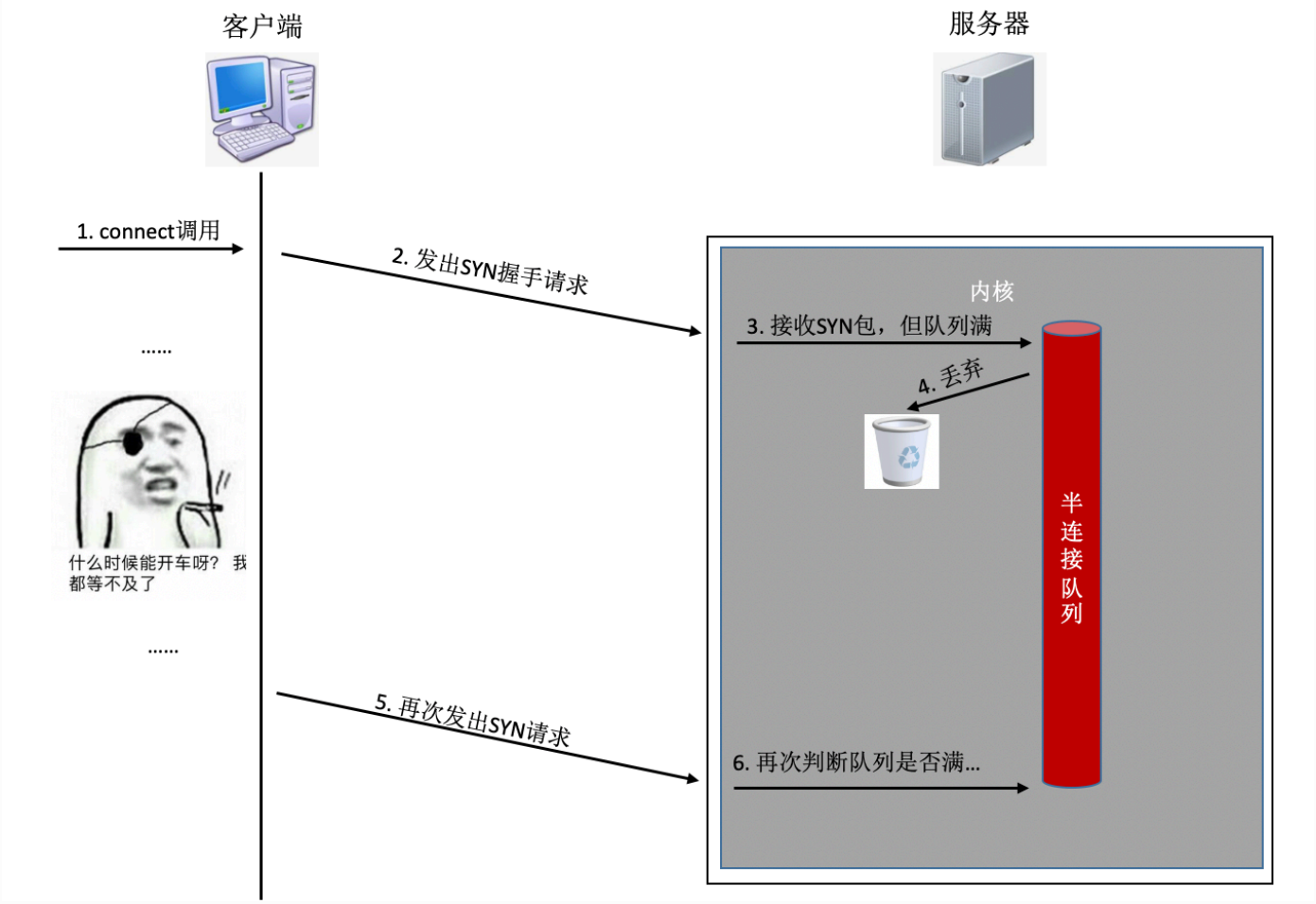
```
[redacted]# strace -cp 31066
Process 31066 attached - interrupt to quit
^CProcess 31066 detached
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 22.89    0.008559          37       234           sendto
 21.73    0.008123          33       249           epoll_wait
 11.21    0.004191          22       188       188 connect
 10.42    0.003895          15       262           close
  7.14    0.002668           5       535       153 recvfrom
  6.74    0.002519          13       188           socket
  5.88    0.002198           6       344           epoll_ctl
  4.04    0.001510          10       148           write
  3.44    0.001286          10       130           setsockopt
  3.34    0.001248           5       250           gettimeofday
  0.99    0.000371           5        74           writev
  0.71    0.000264           1       188           ioctl
  0.63    0.000235           3        74           accept4
  0.52    0.000195           3        74           shutdown
  0.34    0.000128           1       188           getsockopt
------ ----------- ----------- --------- --------- ----------------
```



当遇到这种问题的时候，虽然TCP连接建立耗时只增加了2ms左右，整体TCP连接耗时看起来还可接受。但是这里的问题在于这2ms多都是在消耗CPU的周期，所以问题不小。
解决起来也非常简单，办法很多：修改内核参数net.ipv4.ip_local_port_range多预留一些端口号、改用长连接都可以。

半/全连接队列满

如果连接建立的过程中，任意一个队列满了，那么客户端发送过来的syn或者ack就会被丢弃。客户端等待很长一段时间无果后，然后会发出TCP Retransmission重传。拿半连接队列举例：



要知道的是上面TCP握手超时重传的时间是秒级别的。也就是说一旦server端的连接队列导致连接建立不成功，那么光建立连接就至少需要秒级以上。而正常的在同机房的情况下只是不到1毫秒的事情，整整高了1000倍左右。尤其是对于给用户提供实时服务的程序来说，用户体验将会受到较大影响。如果连重传也没有握手成功的话，很可能等不及二次重试，这个用户访问直接就超时了。

还有另外一个更坏的情况是，它还有可能会影响其它的用户。假如你使用的是进程/线程池这种模型提供服务，比如php-fpm。我们知道fpm进程是阻塞的，当它响应一个用户请求的时候，该进程是没有办法再响应其它请求的。假如你开了100个进程/线程，而某一段时间内有50个进程/线程卡在和redis或者mysql服务器的握手连接上了（**注意：这个时候你的服务器是TCP连接的客户端一方**）。这一段时间内相当于你可以用的正常工作的进程/线程只有50个了。而这个50个worker可能根本处理不过来，这时候你的服务可能就会产生拥堵。再持续稍微时间长一点的话，可能就产生雪崩了，整个服务都有可能会受影响。

既然后果有可能这么严重，那么我们如何查看我们手头的服务是否有因为半/全连接队列满的情况发生呢？在客户端，可以抓包查看是否有SYN的TCP Retransmission。如果有偶发的TCP Retransmission，那就说明对应的服务端连接队列可能有问题了。

在服务端的话，查看起来就更方便一些了。`netstat -s` 可查看到当前系统半连接队列满导致的丢包统计，但该数字记录的是总丢包数。你需要再借助 `watch` 命令动态监控。如果下面的数字在你监控的过程中变了，那说明当前服务器有因为半连接队列满而产生的丢包。你可能需要加大你的半连接队列的长度了。

```
$ watch 'netstat -s | grep LISTEN'
    8 SYNs to LISTEN sockets ignored
```

对于全连接队列来说呢，查看方法也类似。

```
$ watch 'netstat -s  | grep overflowed'
    160 times the listen queue of a socket overflowed
```

如果你的服务因为队列满产生丢包，其中一个做法就是加大半/全连接队列的长度。 半连接队列长度Linux内核中，主要受tcp_max_syn_backlog影响 加大它到一个合适的值就可以。

```
# cat /proc/sys/net/ipv4/tcp_max_syn_backlog
1024
# echo "2048" > /proc/sys/net/ipv4/tcp_max_syn_backlog
```

全连接队列长度是应用程序调用listen时传入的backlog以及内核参数net.core.somaxconn二者之中较小的那个。你可能需要同时调整你的应用程序和该内核参数。

```
# cat /proc/sys/net/core/somaxconn
128
# echo "256" > /proc/sys/net/core/somaxconn
```

改完之后我们可以通过ss命令输出的 `Send-Q` 确认最终生效长度：

```
$ ss -nlt
Recv-Q Send-Q Local Address:Port Address:Port
0      128    *:80              *:*
```

> `Recv-Q` 告诉了我们当前该进程的全连接队列使用长度情况。如果 `Recv-Q` 已经逼近了 `Send-Q` ,那么可能不需要等到丢包也应该准备加大你的全连接队列了。

如果加大队列后仍然有非常偶发的队列溢出的话，我们可以暂且容忍。如果仍然有较长时间处理不过来怎么办？另外一个做法就是直接报错，不要让客户端超时等待。例如将Redis、Mysql等后端接口的内核参数tcp_abort_on_overflow为1。如果队列满了，直接发reset给client。告诉后端进程/线程不要痴情地傻等。这时候client会收到错误"connection reset by peer"。**牺牲一个用户的访问请求，要比把整个站都搞崩了还是要强的。**

5.3 TCP 连接耗时实测

我写了一段非常简单的代码，用来在客户端统计每创建一个TCP连接需要消耗多长时间。

```
<?php
$ip = {服务器ip};
$port = {服务器端口};
$count = 50000;
function buildConnect($ip,$port,$num){
  for($i=0;$i<$num;$i++){
    $socket = socket_create(AF_INET,SOCK_STREAM,SOL_TCP);
    if($socket ==false) {
        echo "$ip $port socket_create() 失败的原因
是:".socket_strerror(socket_last_error($socket))."\n";
        sleep(5);
      continue;
    }

    if(false == socket_connect($socket, $ip, $port)){
        echo "$ip $port socket_connect() 失败的原因
是:".socket_strerror(socket_last_error($socket))."\n";
        sleep(5);
      continue;
    }
    socket_close($socket);
  }
}

$t1 = microtime(true);
buildConnect($ip, $port, $count);
```

```
echo (($t2-$t1)*1000).'ms';
```

在测试之前，我们需要本机linux可用的端口数充足，如果不够50000个，最好调整充足。

```
# echo "5000   65000" /proc/sys/net/ipv4/ip_local_port_range
```

正常情况

注意：无论是客户端还是服务器端都不要选择有线上服务在跑的机器，否则你的测试可能会影响正常用户访问

首先我的客户端位于河北怀来的IDC机房内，服务器选择的是公司广东机房的某台机器。执行ping命令得到的延迟大约是37ms，使用上述脚本建立50000次连接后，得到的连接平均耗时也是37ms。这是因为前面我们说过的，对于客户端来看，第三次的握手只要包发送出去，就认为是握手成功了，所以只需要一次RTT、两次传输耗时。虽然这中间还会有客户端和服务端的系统调用开销、软中断开销，但由于它们的开销正常情况下只有几个us(微秒)，所以对总的连接建立延时影响不大。

接下来我换了一台目标服务器，该服务器所在机房位于北京。离怀来有一些距离，但是和广东比起来可要近多了。这一次ping出来的RTT是1.6~1.7ms左右，在客户端统计建立50000次连接后算出每条连接耗时是1.64ms。

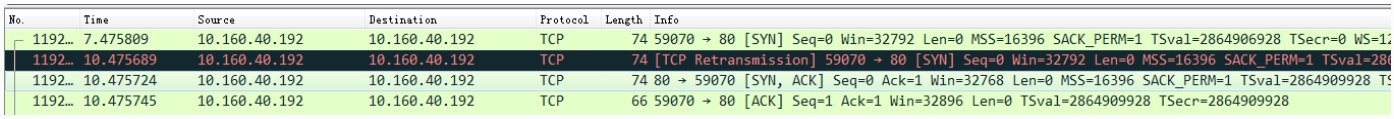
再做一次实验，这次选中实验的服务器和客户端直接位于同一个机房内，ping延迟在0.2ms~0.3ms左右。跑了以上脚本以后，实验结果是50000 TCP连接总共消耗了11605ms，平均每次需要0.23ms。

> **线上架构提示**：这里看到同机房延迟只有零点几ms，但是跨个距离不远的机房，光TCP握手耗时就涨了4倍。如果再要是跨地区到广东，那就是百倍的耗时差距了。线上部署时，理想的方案是将自己服务依赖的各种mysql、redis等服务和自己部署在同一个地区、同一个机房（再变态一点，甚至可以是甚至是同一个机架）。因为这样包括TCP链接建立啥的各种网络包传输都要快很多。要尽可能避免长途跨地区机房的调用情况出现。

连接队列溢出

测试完了跨地区、跨机房和跨机器。这次为了快，直接和本机建立连接结果会咋样呢？
Ping本机ip或127.0.0.1的延迟大概是0.02ms，本机ip比其它机器RTT肯定要短。我觉得肯定连接会非常快，嗯实验一下。连续建立5W TCP连接，总时间消耗27154ms，平均每次需要0.54ms左右。嗯！？怎么比跨机器还长很多？
有了前面的理论基础，我们应该想到了，由于本机RTT太短，所以瞬间连接建立请求量很大，就会导致全连接队列或者半连接队列被打满的情况。一旦发生队列满，当时撞上的那个连接请求就得需要3秒+的连接建立延时。所以上面的实验结果中，平均耗时看起来比RTT高很多。

在实验的过程中，我使用tcpdump抓包看到了下面的一幕。原来有少部分握手耗时3s+，原因是半连接队列满了导致客户端等待超时后进行了SYN的重传。



No.	Time	Source	Destination	Protocol	Length	Info
1192…	7.475809	10.160.40.192	10.160.40.192	TCP	74	59070 → 80 [SYN] Seq=0 Win=32792 Len=0 MSS=16396 SACK_PERM=1 TSval=2864906928 TSecr=0 WS=12
1192…	10.475689	10.160.40.192	10.160.40.192	TCP	74	[TCP Retransmission] 59070 → 80 [SYN] Seq=0 Win=32792 Len=0 MSS=16396 SACK_PERM=1 TSval=286
1192…	10.475724	10.160.40.192	10.160.40.192	TCP	74	80 → 59070 [SYN, ACK] Seq=0 Ack=1 Win=32768 Len=0 MSS=16396 SACK_PERM=1 TSval=2864909928 TS
1192…	10.475745	10.160.40.192	10.160.40.192	TCP	66	59070 → 80 [ACK] Seq=1 Ack=1 Win=32896 Len=0 TSval=2864909928 TSecr=2864909928

我们又重新改成每500个连接，sleep 1秒。嗯好，终于没有卡的了（或者也可以加大连接队列长度）。结论是本机50000次TCP连接在客户端统计总耗时102399 ms，减去sleep的100秒后，平均每个TCP连接消耗0.048ms。比ping延迟略高一些。这是因为当RTT变的足够小的时候，内核CPU耗时开销就会显现出来了，另外TCP连接要比ping的icmp协议更复杂一些，所以比ping延迟略高0.02ms左右比较正常。

5.4 结论

TCP连接建立异常情况下，可能需要好几秒，一个坏处就是会影响用户体验，甚至导致当前用户访问超时都有可能。另外一个坏处是可能会诱发雪崩。所以当你的服务器使用短连接的方式访问数据的时候，一定要学会要监控你的服务器的连接建立是否有异常状态发生。如果有，学会优化掉它。当然你也可以采用本机内存缓存，或者使用连接池来保持长连接，通过这两种方式直接避免掉TCP握手挥手的各种开销也可以。

再说正常情况下，TCP建立的延时大约就是两台机器之间的一个RTT耗时，这是避免不了的。但是你可以控制两台机器之间的物理距离来降低这个RTT，比如把你要访问的redis尽可能地部署的离后端接口机器近一点，这样RTT也能从几十ms削减到最低可能零点几ms。

最后我们再思考一下，如果我们把服务器部署在北京，给纽约的用户访问可行吗?
前面的我们同机房也好，跨机房也好，电信号传输的耗时基本可以忽略（因为物理距离很近），网络延迟基本上是转发设备占用的耗时。但是如果是跨越了半个地球的话，电信号的传输耗时我们可得算一算了。
北京到纽约的球面距离大概是15000公里，那么抛开设备转发延迟，仅仅光速传播一个来回（RTT是Rround trip time，要跑两次），需要时间 = 15,000,000 *2 / 光速 = 100ms。实际的延迟可能比这个还要大一些，一般都得200ms以上。建立在这个延迟上，要想提供用户能访问的秒级服务就很困难了。所以对于海外用户，最好都要在当地建机房或者购买海外的服务器。

六、LINUX 网络性能优化建议

本章的很多优化建议都会涉及到内核参数的修改。

对于修改内核参数飞哥建议使用编辑 /etc/sysctl.conf 文件的方式来改，编辑完了执行一下 sysctl -p 就能生效。不推荐采用 echo xxx > 的这种方式，重启的配置丢失会给你造成很多不必要的困惑。

6.1 耗时优化建议

我们前文分析了 TCP 连接的耗时问题，如果在我们的自己的项目中该怎么进行优化呢？飞哥大致整理了如下的一些建议：

建议1：尽量减少不必要的网络 IO

我要给出的第一个建议就是不必要用网络 IO 的尽量不用。

是的，网络在现代的互联网世界里承载了很重要的角色。用户通过网络请求线上服务、服务器通过网络读取数据库中数据，通过网络构建能力无比强大分布式系统。网络很好，能降低模块的开发难度，也能用它搭建出更强大的系统。但是这不是你滥用它的理由！

我曾经见过有的同学要请求几个第三方的服务，这些服务提供了一个 C 或者 Java 语言的 SDK，说是 SDK 其实就是简单的一次 udp 或者 TCP 请求的封装而已。这个同学呢，不熟悉 C 和 Java 语言的代码，为了省事就直接在自己的机器上把这些 SDK 部署上来，然后自己就通过本机网络 IO 调用这个 SDK。我们接手这个项目以后，分析了一下这几个 SDK 的实现，直接照着在自己的服务里写了一遍就完事了。效果是该项目 CPU 整体核数削减了 20%+。（而且项目的部署难度，可维护性也都得到了极大的提升）

看过咱们前文的内核实现你就知道了，网络的开销是很大的。先说发送一个网络包，首先得从用户态切换到内核态，花费一次系统调用的开销。进入到内核以后，又得经过冗长的协议栈，这会花费不少的 CPU 周期。最后再由内核放到网卡的发送队列里，网卡把数据发送出去。接收端呢，软中断花费不少的 CPU 周期收到接收队列里，唤醒或者通知用户进程来处理。当服务端处理完以后，还得把结果再发过来。又得来这么一遍，最后你的进程才能收到结果。

建议2：内网调用不要用外网域名

我觉得这是非常基础的一个要求，但是很多初级的程序员非常容易犯错。

假如说你所在负责的服务需要调用兄弟部门的一个搜索接口，假设兄弟部门的接口是：

"http://www.sogou.com/wq?key=开发内功修炼"

那既然是兄弟部门，那很可能这个接口和你的服务是部署在一个机房的。即使没有部署在一个机房，一般也是有专线可达的。**所以你不要直接请求 www.sogou.com， 而是应该使用该服务在公司对应的内网域名**。在我们搜狗内部，每一个外网服务都会配置一个对应的内网域名，我相信你们公司也有。

为什么要这么做，原因有以下几点

1）调用外网服务接口会更慢。本来内网可能过个交换机就能达到兄弟部门的机器，你非得上外网都一圈再回来，时间上肯定会慢的多。

2）带宽成本高。在互联网服务里，除了机器以外，另外一块很大的成本就是 IDC 机房的带宽成本。 两台机器在内网不管如何通信都不涉及到带宽的计算。但是一旦你去外网兜了一圈回来，行了，一进一出全部要缴带宽费，你说亏不亏！！

3）NAT 单点瓶颈。一般的服务器都没有外网 IP，所以要想请求外网的资源，必须要经过 NAT 服务器。但是一个公司的机房里几千台服务器中，承担 NAT 角色的可能就那么几台。一来它很容易成为瓶颈。我们的业务就遇到过好几次 NAT 故障导致外网请求事变的情形。 二来是个单点，一旦 NAT 机器挂了，你的服务就挂了，故障率大大增加。

建议3：调用者与被调用机器尽可能部署的近一些

我们看到在全连接队列、半连接队列不出问题的情况下， TCP 握手的时间基本取决与两台机器的 RTT 耗时。虽然我们没办法去掉这个耗时，但是只要两台机器距离足够近，我们就可以把 RTT 降到最低。

比如你的服务是部署在北京机房的，你们你调用的 mysql、redis最好都位于北京机房内部。尽量不要跨过千里万里跑到广东的 IDC 机房去请求数据，即使你有专线，耗时也会大大增加！

建议4： 如果请求频繁，请弃用短连接改用长连接

如果你的服务器频繁请求某个 server，比如 redis 缓存。那么一个比较好的方法是使用长连接。这样的好处有

1）节约了握手的时间开销。短连接中每次请求都需要服务和缓存之间进行握手，这样每次都得让用户多等一个握手的时间开销。

2）规避了队列满的问题。前面我们看到当全连接或者半连接队列溢出的时候，服务器直接丢包。而客户端呢并不知情，所以傻傻地等 3 秒才会重试。要知道 tcp 本身并不是专门为互联网服务设计的。这个 3 秒的超时对于互联网用户的体验影响是致命的。

3）端口数不容易出问题。端连接中，在释放连接的时候，客户端使用的端口需要进入 TIME_WAIT 状态，等待 2 MSL的时间才能释放。所以如果连接频繁，端口数量很容易不够用。而长连接就固定使用那么几十上百个端口就够用了。

建议5：如果可能尽量合并网络请求

这个建议我们举个实践中的例子可能更好理解。

假如有一个 redis，里面存了每一个 App 的信息（应用名、包名、版本、截图等等）。你现在需要根据用户安装应用列表来查询数据库中有哪些应用比用户的版本更新，如果有则提醒用户更新。

那么最好不要写出如下的代码

```
for(安装列表 as 包名){
   redis->get(包名)
   ...
}
```

这样的话，假如用户安装了 100 个 App。那么你的服务器就需要和 redis 进行 100 次网络请求。 总耗时最少是 100 个 RTT 起。

更好的方法是应该使用 redis 中提供的批量获取命令，如 hmget、pipeline等，经过一次网络 IO 就获取到所有想要的数据。

建议6：注意你的半连接、全连接队列的长度

如果你使用了短连接，那么一定在服务流量大的时候要关注服务器上的这两个队列是否存在溢出的情况。如果有，请加大它！因为一旦出现因为连接队列导致的握手问题，那么耗时都是秒级以上了。

我们如何查看我们手头的服务是否有因为半/全连接队列满的情况发生 呢?

在服务器上，netstat -s 可查看到当前系统半连接队列满导致的丢包统计。但该数字记录的是总丢包数，你需要再借助 watch 命令动态监控。如果下面的数字在你监控的过 程中变了，那说明当前服务器有因为半/全连接队列满而产生的丢包。你可能需要加大你的半/全连接队列的长度了。

```
$ watch 'netstat -s'
8 SYNs to LISTEN sockets ignored  //半连接队列满导致的丢包
160 times the listen queue of a socket overflowed //全连接队列满导致的丢包
```

如果你手头并没有服务器的权限，只是发现自己的客户端机连接某个 server 出现耗时长，想定位一下是否是因为握手队列的问题。那也有间接的办法，可以 tcpdump 抓包查看是否有 SYN 的TCP Retransmission。如果有偶发的TCP Retransmission， 那就说明对应的服务端连接队列可能有问题了。

Linux 内核中半连接队列长度主要受内核参数 tcp_max_syn_backlog 影响。如果需要，可以加大它。全连接队列是应用程序调用 listen时传入的 backlog 以及内核参数 net.core.somaxconn 二者之中较小的那个。如果需要加大，可能两个参数都需要改。

6.2 内存优化建议

Tcp 连接的内存开销并不大， 单纯 socket 也就是 3.3 KB 左右。发送缓存区、接收缓存区可能随着数据的收发会占用一些，但是内核也会尽量会去回收它。所以 TCP 连接本身使用的内存上，能做的优化不多。只能在收发缓存区，还有进程内存上理理思路了。

建议1：设置合适的收发缓存区大小

内核为每条 TCP 连接上的收发缓存区都设置了内核参数来控制。每条连接上收发缓存区的最大值由内核参数 net.core.rmem_max 和 net.core.wmem_max 来控制。该值设置的越高，能支持的网速就越高。该值越低就越节约内存！

例如，我手头某台机器上这两个参数的默认值是 8MB 左右。据说如果要让万兆网卡能全速率工作的，该值需要设置到 16MB 以上。

```
#sysctl -a
net.core.rmem_max = 8388608
net.core.wmem_max = 8388608
...
```

开启收发缓存区的自动调优，相关参数有

```
#sysctl -a
net.ipv4.tcp_moderate_rcvbuf = 1
net.ipv4.tcp_rmem = 4096  87380 8388608
net.ipv4.tcp_wmem = 4096  65536 8388608
```

这三个值分别是最小值、默认值和最大字节数（但是我的实验里却发现实际开销中最小值能比 rmem还小，这块的细节还没有去深入扒源码分析）。修改方式和 *mem_max类似，想要网速就加大，尤其是最大数。想要省内存就减小。

建议2：使用 mmap 减少拷贝

假如你要发送一个文件给另外一台机器上，那么比较基础的做法是先调用 read 把文件读出来，再调用 send 把数据把数据发出去。这样数据需要频繁地在内核态内存和用户态内存之间拷贝。

而使用 mmap 系统调用的话，映射进来的这段地址空间的内存在用户态和内核态都是可以使用的。如果你发送数据是发的是 mmap 映射进来的数据，则内核直接就可以从地址空间中读取，然后拷贝到连接的发送缓存区中就行了。

建议3： sendfile

在 mmap 发送文件的方式里，还需要涉及内核态和用户态的上下文切换。 如果你只是想把一个文件发送出去，而不关心它的内容，则可以调用另外一个做的更极致的系统调用 - sendfile。在这个系统调用里，彻底把读文件和发送文件给合并起来了，连系统调用的开销就又省了一次。

6.3 CPU 优化建议

网络在 cpu 的，可做的优化措施比较的多。我们挨个细说。

建议1：监控 ringbuffer与调优

前面我们看到，当网线中的数据帧到达网卡后，第一站就是RingBuffer（网卡通过DMA机制将数据帧送到RingBuffer中）。因此我们第一个要监控和调优的就是网卡的RingBuffer，我们使用 `ethtool` 来来查看一下Ringbuffer。

```
# ethtool -g eth0
Ring parameters for eth0:
Pre-set maximums:
RX:     4096
RX Mini:    0
RX Jumbo:   0
TX:     4096
Current hardware settings:
RX:     512
RX Mini:    0
RX Jumbo:   0
TX:     512
```

这里看到我手头的网卡设置RingBuffer最大允许设置到4096，目前的实际设置是512。

> 这里有一个小细节，ethtool查看到的是实际是Rx bd的大小。Rx bd位于网卡中，相当于一个指针。RingBuffer在内存中，Rx bd指向RingBuffer。Rx bd和RingBuffer中的元素是一一对应的关系。在网卡启动的时候，内核会为网卡的Rx bd在内存中分配RingBuffer，并设置好对应关系。

在Linux的整个网络栈中，RingBuffer起到一个任务的收发中转站的角色。对于接收过程来讲，网卡负责往RingBuffer中写入收到的数据帧，ksoftirqd内核线程负责从中取走处理。只要ksoftirqd线程工作的足够快，RingBuffer这个中转站就不会出现问题。但是我们设想一下，假如某一时刻，瞬间来了特别多的包，而ksoftirqd处理不过来了，会发生什么？这时RingBuffer可能瞬间就被填满了，后面再来的包网卡直接就会丢弃，不做任何处理！

那我们怎么样能看一下，我们的服务器上是否有因为这个原因导致的丢包呢？ 前面我们介绍的四个工具都可以查看这个丢包统计，拿 `ethtool` 来举例：

```
# ethtool -S eth0
......
rx_fifo_errors: 0
tx_fifo_errors: 0
```

rx_fifo_errors如果不为0的话（在 ifconfig 中体现为 overruns 指标增长），就表示有包因为RingBuffer装不下而被丢弃了。那么怎么解决这个问题呢？很自然首先我们想到的是，加大RingBuffer这个"中转仓库"的大小。通过ethtool就可以修改。

```
# ethtool -G eth1 rx 4096 tx 4096
```

这样网卡会被分配更大一点的"中转站"，可以解决偶发的瞬时的丢包。不过这种方法有个小副作用，那就是排队的包过多会增加处理网络包的延时。所以另外一种解决思考更好，。那就是让内核处理网络包的速度更快一些，而不是让网络包傻傻地在RingBuffer中排队。怎么加快内核消费RingBuffer中任务的速度呢，别着急，我们继续往下看...

建议2：硬中断监控与调优

硬中断的情况可以通过内核提供的伪文件 `/proc/interrupts` 来进行查看。

```
$ cat  /proc/interrupts
           CPU0       CPU1       CPU2       CPU3
  0:          34          0          0          0   IO-APIC-edge
     timer
 ......
 27:         351          0          0 1109986815   PCI-MSI-edge
     virtio1-input.0
 28:        2571          0          0          0   PCI-MSI-edge
     virtio1-output.0
 29:           0          0          0          0   PCI-MSI-edge
     virtio2-config
 30:     4233459 1986139461     244872     474097   PCI-MSI-edge
     virtio2-input.0
 31:           3          0          2          0   PCI-MSI-edge
     virtio2-output.0
```

上述结果是我手头的一台虚机的输出结果。上面包含了非常丰富的信息，让我们一一道来：

- 网卡的输入队列 `virtio1-input.0` 的中断号是27
- 27号中断都是由CPU3来处理的
- 总的中断次数是1109986815。

这里有两个细节我们需要关注一下。

1）为什么输入队列的中断都在CPU3上呢?

这是因为内核的一个配置，在伪文件系统中可以查看到。

```
#cat /proc/irq/27/smp_affinity
8
```

`smp_affinity` 里是CPU的亲和性的绑定，8是二进制的1000,第4位为1，代表的就是第4个CPU核心-CPU3。

2）对于收包来过程来讲，硬中断的总次数表示的是Linux收包总数吗?
不是，硬件中断次数不代表总的网络包数。第一网卡可以设置中断合并，多个网络帧可以只发起一次中断。第二NAPI 运行的时候会关闭硬中断，通过poll来收包。

现在的主流网卡基本上都是支持多队列的，我们可以通过将不同的队列分给不同的CPU核心来处理，从而加快Linux内核处理网络包的速度。这是最为有用的一个优化手段。

每一个队列都有一个中断号，可以独立向某个CPU核心发起硬中断请求，让CPU来 `poll` 包。通过将接收进来的包被放到不同的内存队列里，多个CPU就可以同时分别向不同的队列发起消费了。这个特性叫做RSS（Receive Side Scaling，接收端扩展）。通过 `ethtool` 工具可以查看网卡的队列情况。

```
# ethtool -l eth0
Channel parameters for eth0:
Pre-set maximums:
RX:   0
TX:   0
Other:    1
Combined: 63
Current hardware settings:
RX:   0
TX:   0
Other:    1
Combined: 8
```

上述结果表示当前网卡支持的最大队列数是63，当前开启的队列数是8。对于这个配置来讲，最多同时可以有8个核心来参与网络收包。如果你想提高内核收包的能力，直接简单加大队列数就可以了，这比加大RingBuffer更为有用。因为加大RingBuffer只是给个更大的空间让网络帧能继续排队，而加大队列数则能让包更早地被内核处理。 `ethtool` 修改队列数量方法如下：

```
#ethtool -L eth0 combined 32
```

我们前文说过，硬中断发生在哪一个核上，它发出的软中断就由哪个核来处理。所有通过加大网卡队列数，这样硬中断工作、软中断工作都会有更多的核心参与进来。

> 每一个队列都有一个中断号，每一个中断号都是绑定在一个特定的CPU上的。如果你不满意某一个中断的CPU绑定，可以通过修改/proc/irq/{中断号}/smp_affinity来实现。

一般处理到这里，网络包的接收就没有大问题了。但如果你有更高的追求，或者是说你并没有更多的CPU核心可以参与进来了，那怎么办？放心，我们也还有方法提高单核的处理网络包的接收速度。

建议3：硬中断合并

先来讲一个实际中的例子，假如你是一位开发同学，和你对口的产品经理一天有10个小需求需要让你帮忙来处理。她对你有两种中断方式：

- 第一种：产品经理想到一个需求，就过来找你，和你描述需求细节，然后让你帮你来改
- 第二种：产品经理想到需求后，不来打扰你，等攒够5个来找你一次，你集中处理

我们现在不考虑及时性，只考虑你的工作整体效率，你觉得那种方案下你的工作效率会高呢？或者换句话说，你更喜欢哪一种工作状态呢？很明显，只要你是一个正常的开发，都会觉得第二种方案更好。对人脑来讲，频繁的中断会打乱你的计划，你脑子里刚才刚想到一半技术方案可能也就废了。当产品经理走了以后，你再想捡起来刚被中断之的工作的时候，很可能得花点时间回忆一会儿才能继续工作。

对于CPU来讲也是一样，CPU要做一件新的事情之前，要加载该进程的地址空间，load进程代码，读取进程数据，各级别cache要慢慢热身。因此如果能适当降低中断的频率，多攒几个包一起发出中断，对提升CPU的工作效率是有帮助的。所以，网卡允许我们对硬中断进行合并。

现在我们来看一下网卡的硬中断合并配置。

```
# ethtool -c eth0
Coalesce parameters for eth0:
Adaptive RX: off  TX: off
......

rx-usecs: 1
rx-frames: 0
rx-usecs-irq: 0
rx-frames-irq: 0
......
```

我们来说一下上述结果的大致含义

- Adaptive RX: 自适应中断合并，网卡驱动自己判断啥时候该合并啥时候不合并
- rx-usecs： 当过这么长时间过后，一个RX interrupt就会被产生
- rx-frames：当累计接收到这么多个帧后，一个RX interrupt就会被产生

如果你想好了修改其中的某一个参数了的话，直接使用 `ethtool -C` 就可以，例如：

```
ethtool -C eth0 adaptive-rx on
```

不过需要注意的是，减少中断数量虽然能使得Linux整体吞吐更高，不过一些包的延迟也会增大，所以用的时候得适当注意。

建议4：软中断 budget 调整

不知道你有没有听说过番茄工作法，它的大致意思就是你要有一整段的不被打扰的时间，集中精力处理某一项作业。这一整段时间时长被建议是25分钟。
对于我们的Linux的处理软中断的ksoftirqd来说，它也和番茄工作法思路类似。一旦它被硬中断触发开始了工作，它会集中精力处理一波儿网络包（绝不只是1个），然后再去做别的事情。

我们说的处理一波儿是多少呢，策略略复杂。我们只说其中一个比较容易理解的，那就是 `net.core.netdev_budget` 内核参数。

```
#sysctl -a | grep
net.core.netdev_budget = 300
```

这个的意思说的是，ksoftirqd一次最多处理300个包，处理够了就会把CPU主动让出来，以便Linux上其它的任务可以得到处理。 那么假如说，我们现在就是想提高内核处理网络包的效率。那就可以让ksoftirqd进程多干一会儿网络包的接收，再让出CPU。至于怎么提高，直接修改不这个参数的值就好了。

```
#sysctl -w net.core.netdev_budget=600
```

如果要保证重启仍然生效，需要将这个配置写到/etc/sysctl.conf

建议5：软中断 Gro 合并

GRO和硬中断合并的思想很类似，不过阶段不同。硬中断合并是在中断发起之前，而GRO已经到了软中断上下文中了。

如果应用中是大文件的传输，大部分包都是一段数据，不用GRO的话，会每次都将一个小包传送到协议栈（IP接收函数、TCP接收）函数中进行处理。开启GRO的话，Linux就会智能进行包的合并，之后将一个大包传给协议处理函数。这样CPU的效率也是就提高了。

```
ethtool -k eth0 | grep generic-receive-offload
generic-receive-offload: on
```

如果你的网卡驱动没有打开GRO的话，可以通过如下方式打开。

```
# ethtool -K eth0 gro on
```

> GRO说的仅仅只是包的接收阶段的优化方式，对于发送来说是GSO。

建议6：减少拷贝

在日常的网络使用场景中，如果是发送文件的场景。则要避免使用 read + send 的系统调用组合。通过 mmap 或者 sendfile 等系统调用不但能减少内存的开销，也还能减少拷贝时的CPU花费。

建议7：减少进程上下文切换

尽量避免使用同步阻塞式的网络 IO，比如 recvfrom。你可能说高性能网络 IO 都优化到今天这个程度了，这种问题应该很少了吧？但其实阻塞 IO 仍然大量地存在。

比如说当你的 C++ Server 需要 访问 mysql 的时候，在workflow出现之前，一般就是使用传统的 mysql 客户端。传统客户端的特点就是一个用户线程以同步阻塞的方式等待连接上的数据。而我们的服务器是要处理很多用户的请求的，这就不得已需要创建很多的线程才行。除了引入额外的创建成本不说，频繁的上下文切换将带来大量无谓的 CPU 开销。

类似的还有 http、redis 和 kafka 客户端。只要用到它们，就又会将自己精心优化过的服务拖回到阻塞的低性能泥潭中。

不过现在有一些封装的很好的网络框架例如 Sogou Workflow，Golang 的 net 包等在网络客户端角色上也早已彻底摒弃了这种低效的模式！

建议8：配置充足的端口范围

客户端在调用 connect 系统调用发起连接的时候，需要先选择一个可用的端口。内核在选用端口的时候，是采用随机撞的方式。如果端口不充足的话，内核可能需要循环撞很多次才能撞上一个可用的，这也会导致花费更多的 CPU 周期。因此不要等到端口用尽报错了才开始加大端口范围。

```
#vi /etc/sysctl.conf
net.ipv4.ip_local_port_range = 5000 65000
```

如果端口加大了仍然不够用，那么可以考虑开启端口 reuse。这样客户端的端口在连接断开的时候就不需要等待 2MSL 的时间了，1 s 就可以回收。

开启这个参数之前需要保证 tcp_timestamps 是开启的。

```
net.ipv4.tcp_timestamps = 1
net.ipv4.tcp_tw_reuse = 1
```

七、项目实际问题分析

7.1 我的机器上出现了 3 万多个TIME_WAIT，开销有哪些?

其实这种情况只能算是 warning，而不是 error!

从内存的角度来考虑，一条 TIME_WAIT 状态的连接仅仅是 0.5 KB 的内存而已。

从端口的角度来考虑，占用的端口只是针对特定的 server 来说是占用了。只要下次连接的 server 不一样（ip 或者 端口不一样都算），那么这个端口仍然可以用来发起 TCP 连接。

无论是从内存的角度，还是端口的角度一条 TIME_WAIT 的开销都并不那么可怕。如果你确实连这一点点开销都想省，那你可以考虑打开 tcp_tw_recycle、 tcp_tw_reuse。如果再彻底一些，也可以干脆直接用长连接代替频繁的短连接。

7.2 我的 REDIS SERVER 上 6000 多个长连接，会不会把我搞垮?

之前大家在fpm里有的时候会用长连接来提高效率。但是对于fpm总量比较多，我是没敢用长连接的。比如单php服务器1000fpm，有20台php服务器。

那么，具备今天的知识以后，我们来看一下Redis server维持1000*20的TCP连接的开销都有哪些？如果真的用长连接的话问题大不大。

先看端口

每个客户ip只是用了1000个端口，离linux默认的3W差着远呢。所以客户端的端口使用率是没有问题的。（有一种情况要注意，就是php连了很多个redis，甚至包括mysql等，这时候就要注意端口总量了）

再看服务器端，服务器的端口不影响TCP连接数量，所以服务器端也不存在端口的问题。

再看内存

客户机来说，每个客户机仅仅维护1000个连接，占用的内存可以忽略了。不到 10 M 而已。对于 redis 来说，每个长连接大约需要 7 k 的内存（假设我们发送的数据都没有超过 4 k ）。那么 2 万个连接需要的内存为 140 M 内存。

再看 CPU和耗时

先说数据传输，短连接和长连接这块都是要占用的，所以没有讨论的必要。

再说建立连接，这个长连接是有优势的，省去了频繁的 TCP 握手包对服务机的软硬中断开销。而且还减少了客户机的单个服务的响应时间（一般情况下通往内网机器的 RTT 1ms 以内，so 你的服务也会节约 1ms），这样服务能力也有所提升。

结论

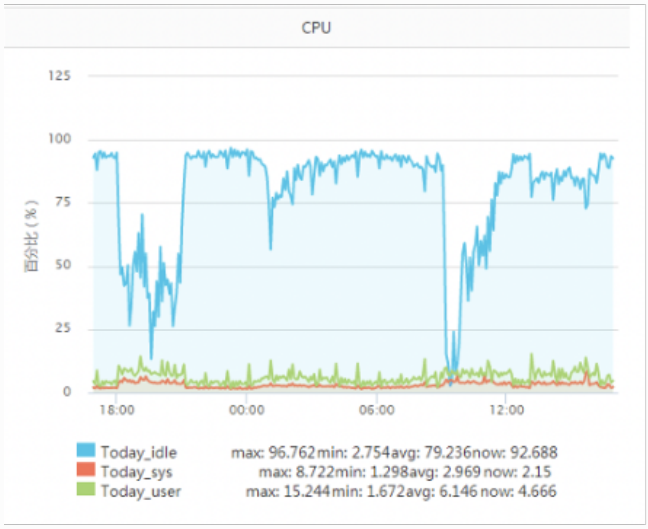
如果你的server压力不是特别大，短连接挺好。多一点cpu的开销，但是没有维护连接的成本，也不会触发server文件句柄超默认值的问题。

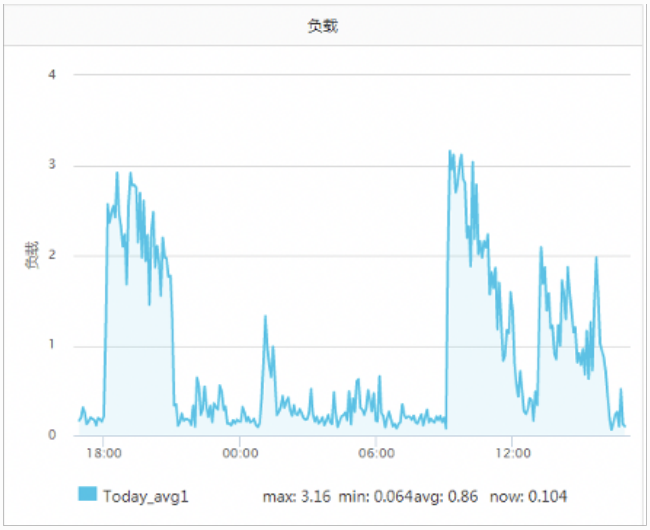
但是如果你的server请求量较大，看起来Redis维护成千上万的长连接很可怕，但其实用量化的指标对其计算一遍以后发现，长连接的开销其实只是多一些内存占用而已，另外就是可能需要适当调节一下服务器的最大文件句柄限制。Don't afraid，如果到了该用长连接的时候就用。

7.3 服务器负载很正常，但是 CPU 被打到底了怎么回事?

之前，我一直觉得 CPU 和负载是正相关关系。CPU 高的话，负载就会高。但其实负载是统计就绪状态进程的排队情况。

但如果进程还没到 Ready 状态的时候， cpu 就被占了许多的话。就会出现CPU占用较高，但负载却不高。（因为进程还没有收到数据，都是内核在玩命干活）。见下图：





7.4 我要做一个长连接推送模块，1亿用户需要多少台机器?

前文中我们看到， 100 万条空的 TCP 连接仅仅只消耗掉了 3 GB 多一点的内存而已。

对于长连接的推送模块这种服务来说， 给客户端发送数据只是偶尔的，一般一天也就顶多发送一次两次的。绝大部分情况下 TCP 连接都会空闲的。

假设你的服务器内存是 128 GB 的。 那么一台服务器可以考虑支持 500 万条的并发。这样会消耗大约不到 20 G 的内存用来保存这 500 万条连接对应的 socket。 还剩下 100 G 还多的内存，用来应对发送缓存区的开销足够了。

所以，1亿用户，仅仅只需要 20 台 机器，够了!

八、前沿技术展望

前面我们看到了内核在接收网络包的时候要经过很长的接收路径。如果你的服务对网络要求确实特别特别的高，而且各种优化措施也都用过了，那么现在还有终极优化大招 -- Kernel-ByPass 技术。

说白了就是各家想法设法地绕开内核，直接在用户态进行数据的收发，进而避免内核开销。

SOLARFLARE 的软硬件方案

Solarflare 除了硬件网卡之外，还提供了配套的 kernel bypass 软件解决方案，还不止一个：Onload, ef_vi 和 Tcpdirect

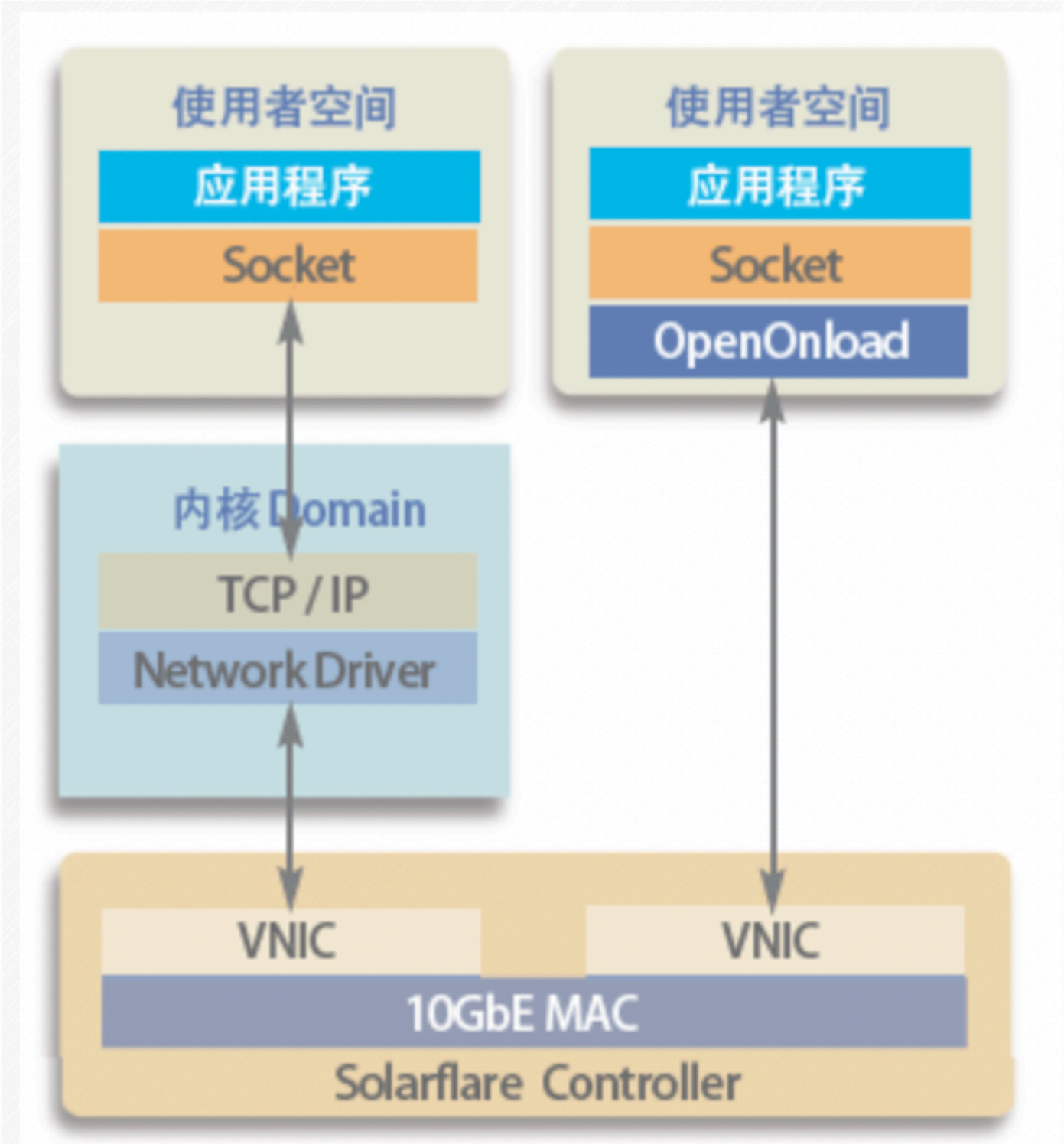

使用者空间
应用程序
Socket
使用者空间
应用程序
Socket
OpenOnload
内核 Domain
TCP / IP
Network Driver
VNIC
VNIC
10GbE MAC
Solarflare Controller

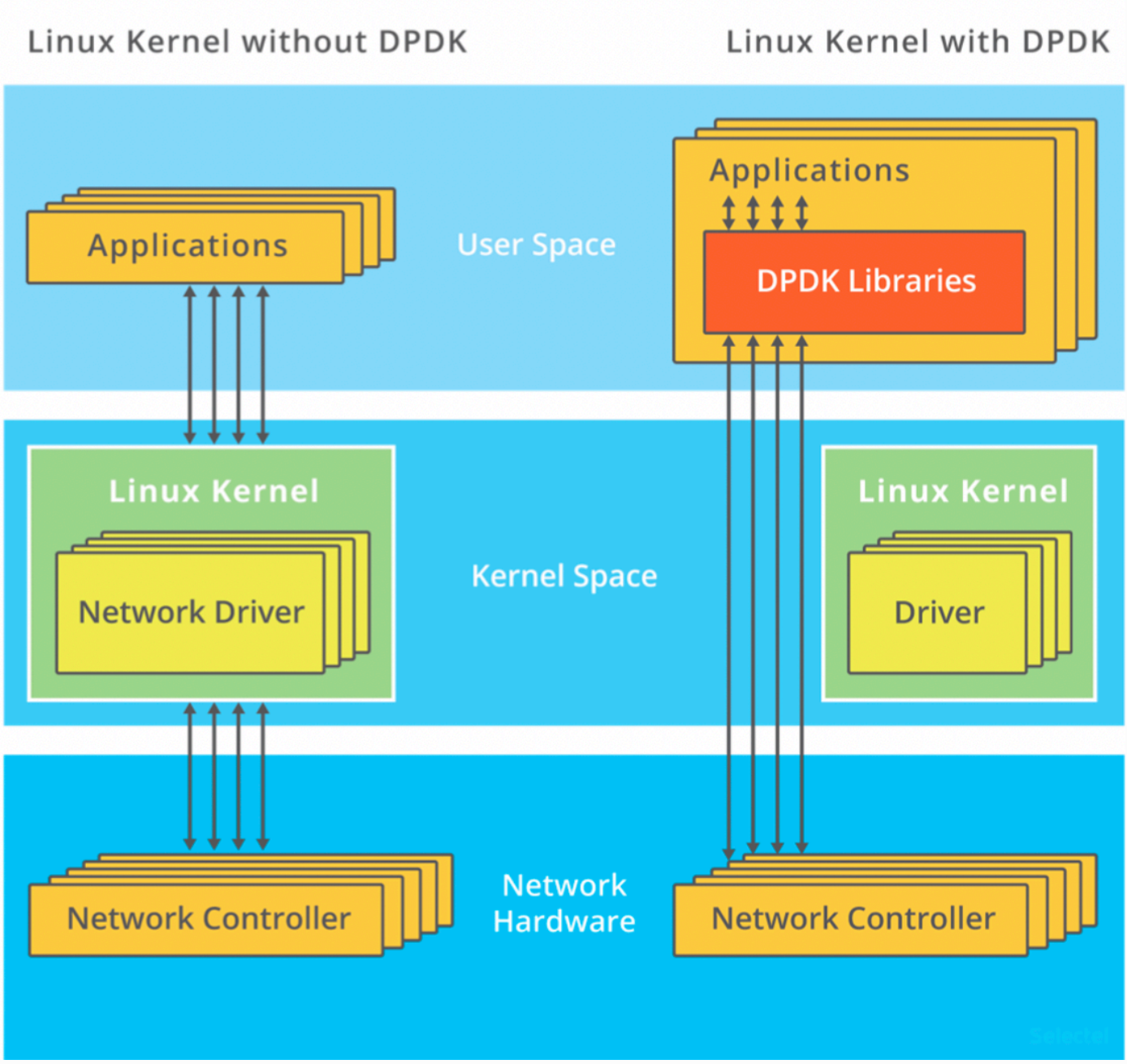
Table 1: ½ RTT latency for a 32 byte message

Acceleration	Protocol	25Gb	10Gb	Notes	Page
Onload	UDP	1022ns	1095ns	sfnt-pingpong	8
		1034ns	1107ns	netperf	8
	TCP	1025ns	1110ns	sfnt-pingpong	8
		1032ns	1119ns	netperf	8
TCPDirect	UDP	783ns	864ns	zfudppingpong	9
		968ns	1022ns	No CTPIO	10
	TCP	795ns	870ns	zftcppingpong	9
ef_vi	UDP	750ns	819ns	eflatency	9
Kernel	UDP	2658ns	2750ns	sfnt-pingpong	10
	TCP	3124ns	3257ns	sfnt-pingpong	10

参见： http://www.ht1.co.kr/cn/sub02_02.php

DPDK

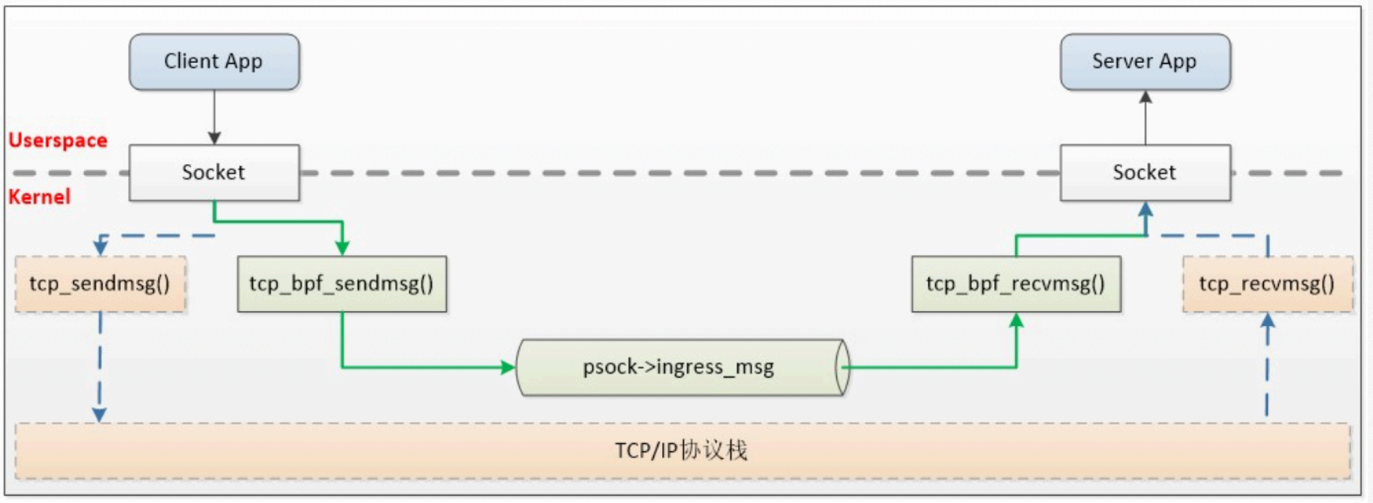
同样节缩短了繁杂的硬中断、软中断、内核协议处理，用户态内存拷贝等过程。



直接由用户进程绕开内核态进行数据的收发！

基于 EBPF 的 SOCKOPS

绕开 Linux 协议栈进行本机网络通信，适用于 nginx =》本机 fpm，或者是 sidecar 之类的应用场景。



公 众 号: 「开 发 内 功 修 炼」

了 解 你 的 每 一 比 特 、 用 好 你 的 每 一 纳 秒 !

公 众 号

作 者 微 信

九、推荐书单

经常收到后台读者发过来同样的问题，看完「开发内功修炼」以后觉得写的不错，问可否帮推荐几本书更系统地学习一下。今天干脆就写一篇文章统一回复一下。

经典计算机图书的价值非常巨大，比普通的书的质量要好百倍。所以要看就要看经典，否则会浪费很多时间。其中很多好书不单单是要看一遍，而且在工作中遇到问题的时候随时要拿出来查。虽然我给大家搜集了电子版，但我仍然建议你最好是拥有纸质版。

- 第一、当你有问题想去查的时候，纸质的要比电子的更容易定位
- 第二、也支持一下出版社和作者，回馈下他们的劳动

计算机系统

- 1.《编码：隐匿在计算机软硬件背后的语言》
- 2.《操作系统之哲学原理 第2版》
- 3.《计算机是怎样跑起来的》
- 4.《计算机体系结构：量化研究方法》
- 5.《计算机组成原理》
- 6.**《深入理解计算机系统》**
- 7.《现代操作系统》

计算机网络

- 1.《 TCP-IP详解卷1、卷2、卷3》
- 2.《 UNIX网络编程卷1、卷2》
- 3.《 Wireshark 网络分析就这么简单》
- 4.《图解 TCP + IP 第5版》
- 5.《图解 HTTP 》
- 6.《网络是怎么样连接的》

Linux 内核实现

- 1.**《深入理解 Linux 内核（第三版）》**
- 2.《Linux 内核设计与实现》
- 3.**《深入理解 LINUX 网络技术内幕》**
- 4.现代体系结构上的 unix 系统

- 5.**《追踪Linux TCP/IP代码运行》**
- 6.**Linux源代码：https://mirrors.edge.kernel.org/pub/linux/kernel/**

Linux 环境编程

- 1.**《Go 语言设计与实现》**
- 2.《Go 专家编程》
- 3.**《Linux 环境编程：从应用到内核》**
- 4.《深入理解Linux》

性能分析与调优

- 1、**《性能之巅 洞悉系统、企业与云计算》**
- 2、perf
- 3、BPF

在我公众号「**开发内功修炼**」 后台回复「**电子书**」，即可获取百度网盘下载地址。

如果上述地址失效，请及时联系我更新。以上电子资料仅供学习分享之用，侵删！

公众号：「开发内功修炼」

了解你的每一比特、 用好你的每一纳秒！

公众号

作者微信