TCP内核实现

#### 准备工作

本文依据KERNEL 6.5展开讲解TCP是如何在内核中实现数据的接收和发送。平台设定在Arm64上。

本文分成三个大部分，第一部分是初始化，第二部分讲解TCP（客/服）是如何执行的，不考虑任何拥塞的情况，设定网络质量良好。第三部分讲解在有拥塞的时候的处理。

一些TCP的基本知识会在讲解代码的时候引入。

### 第一部分：

网络初始化主要是有inet\_init/sock\_init完成

Sock\_init优先级最高，函数调用链：

1. sock\_init
2. |- net\_sysctl\_init
3. |- skb\_init
4. |- init\_inodecache
5. |- register\_filesystem
6. |- kern\_mount

net\_sysctl\_init主要注册/proc/sys/net目录（proc如何注册会在文件系统实现中说明）。注册命名空间的sysctl操作函数结构体到first\_device链表尾部。这个阶段还命名空间还没有初始化。

分配跟skb相关的缓存描述指针（该部分会在内存管理中说明），skbuff\_cache， skbuff\_fclone\_cache，skb\_small\_head\_cache

为扩展skb分配skbuff\_ext\_cache

为sock\_inode即socket文件注册缓存描述指针，sock\_inode\_cachep。所有的socket文件描述inode都将从这个缓存描述指针进行分配。

注册socket文件系统，见下图



其中file\_systems是内核注册的所有文件系统类型的链表指针，顺着这张链表就可以快速了解当前系统所注册使用的所有文件系统类型。

Mount sock文件系统

文件系统能够让内核使用，必须先挂载，挂载文件系统是由kern\_mount函数负责。

Kern\_mount函数调用vfs\_kern\_mount函数完成对sock文件系统的挂载，这部分会在文件系统一文中详细解释，这里不做分析。我们看下经过kern\_mount处理后的一些结构体的变化：

挂载后一些结构体的变化如下：



这里面的fc是一个临时结构体，等内核挂载好后，就会删除该结构体。该文件系统的inode是socket\_alloc.vfs\_inode成员。

Inet\_init函数调用链：

1. inet\_init
2. |- proto\_register
3. |- sock\_register
4. |- ip\_static\_sysctl\_init
5. |- inet\_add\_protocol
6. |- inet\_register\_protosw
7. |- tcp\_init

Proto\_register主要为tcp分配几个重要的slab， prot->slab主要为了给tcp\_sock分配内存空间， tcp\_sock是TCP链接保活期间的一个重要的控制结构体。 其次request\_sock->slab，为请求的connect分配空间（后文会提到为什么需要一个单独的请求缓存分配区域）。 Twsk->prot->twsk\_slab，处于TCP\_TIME\_WAIT队列的套接字处理所需要的缓存



Sock\_register注册PF\_INET协议簇的创建函数，注册目的地为net\_families数组：



向inet\_protos注册协议栈操作函数



Inet\_register\_protosw， 向网络层（IP）注册操作函数



Tcp\_init：

创建孤儿tcp（不属于任何进程的TCP连接）定时器， 设置bind\_bucket的cache指针。已建立连接的Hash表 （这里的已建立连接是指 TCP\_ESTABLISHED ≤ state < TCP\_CLOSE）， BIND Hash表。Tcp\_init\_mem设置TCP使用内存的阈值，这里有三个值分别是sysctl\_tcp\_mem 0，1，2， 即min，pressure， max。简单来说如果处于≤min的时候TCP对内存不会产生影响，如果处于≥pressure的时候，TCP需要进入内存压力模式，直到内存≤min为止。超过max，所有的socket分配都将失败。具体的计算方法可以根据

|  |
| --- |
| grep -e zone -e managed -e high /proc/zoneinfo | grep -A 2 zone |

根据managed的和减去high的和除以16得到。这里的数值是page数，不是字节大小

设置tcp read/write mem，这里的数值是字节数，表示TCP能接收的读/写缓存区大小，这里也有三个值，分别是最小值，默认值及最大值。 最小值是在内存压力下也需要保证的缓冲，一般都是4K，即处在tcp\_mem[1] ~ tcp\_mem[2]之间的时候。最大值一般读在6M，写在4M左右。

Tcp\_v4\_init主要作用是为每个cpu创建一个SOCK\_RAW/IPPROTO\_TCP的原始套接字，这是为了什么? 这里暂不做分析，等后续讲实现的时候再做说明。创建的socket保存在ipv4\_tcp\_sk里面，这个变量是每个cpu都创建一个，保存在.data..percpu段里面。

注册tcp\_sk\_ops操作函数，这个操作函数指针主要是为了初始化TCP连接的默认参数，比如重传次数，延时ACK是否开启等。这里也涉及到网络命名空间的知识。Init\_net是一个全局参数，所创建的命名空间全局参数会连接到net\_namespace\_list。

接下来注册默认的拥塞控制newreno

注册TSQ tasklet任务，这是一个tasklet任务。

Mptcp\_init，MultiPath TCP，允许TCP连接使用多个路径来最大化使用信道资源。

### 第二部分：

#### TCP 客户端实现

首先来看下最简单的TCP 客户端程序

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <string.h>
4. #include <unistd.h>
5. #include <arpa/inet.h>
7. **int** main (void) {
8. **int** sockfd;
9. **struct** sockaddr\_in server\_addr;
11. sockfd = socket (AF\_INET, SOCK\_STREAM, 0);
12. **if** (sockfd == -1) {
13. perror("Socket creation failed");
14. **return** 1;
15. }
17. server\_addr.sin\_family = AF\_INET;
18. server\_addr.sin\_port = htons(8080);
19. server\_addr.sin\_addr.s\_addr = inet\_addr("127.0.0.1");
21. **if** (connect (sockfd, (**struct** sockaddr\*) &server\_addr, **sizeof**(server\_addr)) == -1) {
22. perror("Connection failed");
23. **return** 1;
24. }
26. **char** buffer [1024];
27. Memset (buffer, 0, **sizeof**(buffer));
28. Strcpy (buffer, "Hello, server!");
29. Send (sockfd, buffer, strlen(buffer), 0);
31. Recv (sockfd, buffer, **sizeof**(buffer), 0);
32. Printf ("Received from server: %s\n", buffer);
34. Close (sockfd);
36. **return** 0;
37. }

这是一段最简单的TCP客户端代码，代码主要是建立TCP socket，连接到地址为127.0.0.1，端口为8080的服务器上。接下来发送Hello Server至服务器，接收等待服务器的响应，并把响应打印至屏幕，关闭TCP连接。

1. 创建socket

客户端使用socket函数来创建一个套接字，并返回socket类型的文件描述符。

Socket函数属于GLIBC， 通过GLIBC处理后最终变成以下代码：

1. **register** **long** \_x8 asm ("x8") = (name);
2. asm **volatile** ("svc    0      //syscall" # name
3. : "=r" (\_x0) : "r" (\_x8) ASM\_ARGS\_##nr : "memory");
4. \_sys\_result = \_x0;

在Arm64里面系统调用参数都是放在X0 – X7里面，所以系统调用最大支持8个参数。调用名会放入x8寄存器，在这里就是socket。

Arm64 使用svc 产生异常进入kernel。X0寄存器保存第一个参数，同时也会保存系统调用的返回值。

进入内核调用\_\_sys\_socket函数

函数调用链：

1. \_\_sys\_socket
2. |- \_\_sys\_socket\_create
3. |- sock\_map\_fd

\_\_sys\_socket\_create函数调用链：

1. \_\_sys\_socket\_create
2. |-  \_\_sock\_create
3. |- sock\_alloc
4. |- inet\_create

这里我忽略security的一些调用，这些调用比如security\_socket\_create主要是为了让selinux等一些内核模块来检查的，对整体的TCP实现并无任何实质上的影响。

Sock\_alloc函数创建struct socket结构体， UNIX家族系统把一切视为文件，当然socket也不例外，函数里面使用的是已经初始化好的socket文件系统里面分配相应的inode结构体（这部分在Linux系统启动里面会说明， 关于文件系统会在Linux文件系统实现里面说明）。这里就直接说明分配inode函数是使用sock\_alloc\_inode函数，函数分配struct socket\_alloc结构体，让我们来看一下这个结构体：

1. **struct** socket\_alloc {
2. **struct** socket socket;
3. **struct** inode vfs\_inode;
4. };

我们可以很明显看到socket的文件描述结构和socket结构体。Socket结构体主要包含了本套接字整个生命周期的状态，用户等待队列，网络协议无关套接字结构体（struct sock），套接字文件系统指针等一系列的信息。

设置好该文件的访问模式（拥有者，组，其他，可读写），所有者等，并返回socket结构体指针。

回到\_\_sock\_create函数，从已经注册的网络协议模块数组里面查找当前的协议簇是否存在，如果不存在，则通过request\_module进行对处理这个协议模块的查找和插入。这边忽略这部分内容，因为TCP隶属于AF\_INET协议簇，内核初始化就已经注册好了。

继续运行inet\_create函数，我们注意到socket函数的参数是AF\_INET, SOCK\_STREAM, 0。其中第一个表示协议簇，第二个表示当前协议簇里面的类型，第三个是表示使用协议，那为什么第三个参数是0呢？填0 是为了方便记忆，实际上我们应该填6（IPPROTO\_TCP），但是每个协议类型是不一样的，比如UDP就是17。为了方便记忆第三个参数可以填相应的值，或者0。然后在inet\_create里面把真实的protocol值替换之前传入的0。如果找不到相应的协议簇，会先设置错误类型为协议不支持（EPROTONOSUPPORT），然后尝试加载该协议簇处理模块，还是找不到的话，错误返回。

设置tcp\_prot和inet\_stream\_ops到。 Inet\_stream\_ops支持从客户端下来的调用，tcp\_prot是该协议类型处理相应调用的函数。

分配协议无关套接字结构体（struct sock），初始化sock结构体：

1. **if** (INET\_PROTOSW\_REUSE & answer\_flags)
2. sk->sk\_reuse = SK\_CAN\_REUSE;

TCP有2MSL等待时间，为了防止迷失数据包再次传入。所以默认是不能开启port重用。

1. inet->is\_icsk = (INET\_PROTOSW\_ICSK & answer\_flags) != 0;

表示该协议是一个面向连接的协议。

1. **if** (READ\_ONCE(net->ipv4.sysctl\_ip\_no\_pmtu\_disc))
2. inet->pmtudisc = IP\_PMTUDISC\_DONT;
3. **else**
4. inet->pmtudisc = IP\_PMTUDISC\_WANT;

检查是否支持动态路径MTU检测，这个值由net.ipv4.ip\_no\_pmtu\_disc控制，默认是需要。

1. sk->sk\_destruct     = inet\_sock\_destruct;
2. sk->sk\_protocol     = protocol;
3. sk->sk\_backlog\_rcv = sk->sk\_prot->backlog\_rcv;

注册sock结构体销毁函数及backlog receive处理函数，即tcp\_v4\_do\_rcv

协议初始化tcp\_v4\_init\_sock

1. tcp\_v4\_init\_sock
2. |- tcp\_init\_sock

Tcp\_init\_sock主要初始化一些TCP定义的默认参数，比如超时时间，延时ACK时间，默认窗口大小， 拥塞控制如下：

1. **if** (!net\_eq(net, &init\_net) &&
2. bpf\_try\_module\_get(init\_net.ipv4.tcp\_congestion\_control,
3. init\_net.ipv4.tcp\_congestion\_control->owner))
4. net->ipv4.tcp\_congestion\_control = init\_net.ipv4.tcp\_congestion\_control;
5. **else**
6. net->ipv4.tcp\_congestion\_control = &tcp\_reno;

如果有指定拥塞控制的，则调用指定的拥塞控制模块。默认是RENO算法。在Debian系统上默认是CUBIC拥塞控制算法。注册结构体为cubictcp。

注册ipv4\_speific操作函数结构体到icsk\_af\_ops

处理完毕后返回，回到\_\_sys\_socket。执行sock\_mp\_fd。

1. sock\_map\_fd
2. |- get\_unused\_fd\_flags
3. |- sock\_alloc\_file
4. |- fd\_install

首先获取未使用的文件描述符，使用get\_unused\_fd\_flags。分配新的file结构，这个结构体表示一个打开的文件，有操作该文件的函数。因为当前是socket文件，具体的socket内容会注册到file->private字段。使用fd\_install把当前文件加载到当前进程的文件表， 最后返回文件描述符，这个文件描述符会保存到X0寄存器。系统返回后

\_sys\_result = \_x0;

把返回值赋值给\_sys\_result， GLIBC返回该值到sockfd。



1. 连接

Socket连接需要知道对端的地址和端口，这些信息由sockaddr\_in保存，并由connect传入，同理GLIBC使用svc进入内核。

1. \_\_sys\_connect
2. |- fdget
3. |- move\_addr\_to\_kernel
4. |- \_\_sys\_connect\_file
   1. **int** \_\_sys\_connect (**int** fd, **struct** sockaddr \_\_user \*uservaddr, **int** addrlen)

这里的系统调用有个不一样的地方uservaddr前面有一个\_\_user限定符，这个符号是第一次出现，意味着这个参数里面的内容是在用户空间，使用时应该特别小心，因为是不可信的地址。

Fdget使用文件描述符从当前进程的文件表里面找到相应的文件结构，通过前一章的图可以清楚地了解使用fd从已打开文件数组里面找到相应的file结构体。

使用copy\_from\_user把uservaddr的内容复制到struct sockaddr\_storage结构体

调用\_\_sys\_connect\_file

1. \_\_sys\_connect\_file
2. |- sock\_from\_file
3. |- inet\_stream\_connect

Sock\_from\_file先比较找到的文件结构的操作函数结构体是不是socket\_file\_ops， 这个结构体是read，write等函数的最终执行函数。如果是，表明是socket文件，返回private指针值，这个值前面讲过就是struct socket的内容

Inet\_stream\_connect这个函数比较重要，着重讲解一下：

1. **if** (uaddr) {
2. **if** (addr\_len < **sizeof**(uaddr->sa\_family))
3. **return** -EINVAL;
5. **if** (uaddr->sa\_family == AF\_UNSPEC) {
6. err = sk->sk\_prot->disconnect(sk, flags);
7. sock->state = err ? SS\_DISCONNECTING : SS\_UNCONNECTED;
8. **goto** out;
9. }
10. }

进入函数首先检查uaddr的长度是否合理，否返回无效的参数值错误

检查协议簇是否已经定义，如果没有定义，调用disconnect函数，根据disconnect返回值来设置socket的状态是SS\_DISCONNECTING或SS\_UNCONNECTED，并返回错误值。

1. **switch** (sock->state) {
2. **default**:
3. err = -EINVAL;
4. **goto** out;
5. **case** SS\_CONNECTED:
6. err = -EISCONN;
7. **goto** out;
8. **case** SS\_CONNECTING:
9. **if** (inet\_sk(sk)->defer\_connect)
10. err = is\_sendmsg ? -EINPROGRESS : -EISCONN;
11. **else**
12. err = -EALREADY;
13. **break**;
14. **case** SS\_UNCONNECTED:
15. err = -EISCONN;
16. **if** (sk->sk\_state != TCP\_CLOSE)
17. **goto** out;
19. **if** (BPF\_CGROUP\_PRE\_CONNECT\_ENABLED(sk)) {
20. err = sk->sk\_prot->pre\_connect(sk, uaddr, addr\_len);
21. **if** (err)
22. **goto** out;
23. }
25. err = sk->sk\_prot->connect(sk, uaddr, addr\_len);
26. **if** (err < 0)
27. **goto** out;
29. sock->state = SS\_CONNECTING;
31. **if** (!err && inet\_sk(sk)->defer\_connect)
32. **goto** out;
34. err = -EINPROGRESS;
35. **break**;
36. }

程序是第一次连接，初始状态是SS\_UNCONNECTED，在switch里面直接跳到第四个分支。

TCP是一个以数据报文驱动的状态机，之前初始化sock结构体时，sk->sk\_state的初始状态为TCP\_CLOSE。

检查是否打开了cgroup 数据报文过滤，这里不讨论cgroup，默认不打开。

执行sk->sk\_prot->connect即tcp\_v4\_connect，是connect的最主要处理函数：

1. tcp\_v4\_connect:
2. **if** (addr\_len < **sizeof**(**struct** sockaddr\_in))
3. **return** -EINVAL;
5. **if** (usin->sin\_family != AF\_INET)
6. **return** -EAFNOSUPPORT;

进入函数首先判断参数的合法性，如果用户层传入的sockaddr内容长度太小，返回无效参数。因为TCP属于AF\_INET协议簇，所以如果协议簇不匹配，则返回在当前指定的协议簇下不支持该协议。

1. tcp\_v4\_connect:
2. nexthop = daddr = usin->sin\_addr.s\_addr;
3. inet\_opt = rcu\_dereference\_protected(inet->inet\_opt,
4. lockdep\_sock\_is\_held(sk));
5. **if** (inet\_opt && inet\_opt->opt.srr) {
6. **if** (!daddr)
7. **return** -EINVAL;
8. nexthop = inet\_opt->opt.faddr;
9. }

赋值下一跳以及对端地址值。检查是否设置源记录路由（inet\_opt->opt.srr），SRR（source record route）有两种：1. 宽松源记录路由；2. 严格源记录路由。如果设置了那就把下一跳地址改成记录在faddr的地址。

1. tcp\_v4\_connect:
2. orig\_sport = inet->inet\_sport;
3. orig\_dport = usin->sin\_port;
4. fl4 = &inet->cork.fl.u.ip4;
5. rt = ip\_route\_connect(fl4, nexthop, inet->inet\_saddr,
6. sk->sk\_bound\_dev\_if, IPPROTO\_TCP, orig\_sport,
7. orig\_dport, sk);
8. **if** (IS\_ERR(rt)) {
9. err = PTR\_ERR(rt);
10. **if** (err == -ENETUNREACH)
11. IP\_INC\_STATS(net, IPSTATS\_MIB\_OUTNOROUTES);
12. **return** err;
13. }

赋值源，目的端口到orig\_sport和orig\_dport，取出struct flowi4结构体，查找路由表及相应设备（详细内容会在路由章节介绍）

1. tcp\_v4\_connect:
2. **if** (rt->rt\_flags & (RTCF\_MULTICAST | RTCF\_BROADCAST)) {
3. ip\_rt\_put(rt);
4. **return** -ENETUNREACH;
5. }
7. **if** (!inet\_opt || !inet\_opt->opt.srr)
8. daddr = fl4->daddr;

我们假设前面的路由查找肯定是成功的，接下来就是检查rt是否为多播或广播，是的话返回错误，我们不能向一个多播或广播地址建立连接。

如果没有设置源记录路由选项，那么把目的地址改成前面查到到的目的地址。注意这时查到到的目的地址不一定是我们要求发送的目的地址，有可能会是网关地址。

1. tcp\_v4\_connect:
2. tcp\_death\_row = &sock\_net(sk)->ipv4.tcp\_death\_row;
4. **if** (!inet->inet\_saddr) {
5. err = inet\_bhash2\_update\_saddr(sk,  &fl4->saddr, AF\_INET);
6. **if** (err) {
7. ip\_rt\_put(rt);
8. **return** err;
9. }
10. } **else** {
11. sk\_rcv\_saddr\_set(sk, inet->inet\_saddr);
12. }

TCP death row是处理处于关闭的连接，然后在适当时机回收资源。

接下来如果没有设置过源地址那就更新下源地址和sk\_rcv\_saddr。如果设置了，就更新sk\_rcv\_saddr。正常情况我们是没有设置源地址

1. tcp\_v4\_connect:
2. **if** (tp->rx\_opt.ts\_recent\_stamp && inet->inet\_daddr != daddr) {
3. /\* Reset inherited state \*/
4. tp->rx\_opt.ts\_recent    = 0;
5. tp->rx\_opt.ts\_recent\_stamp = 0;
6. **if** (likely(!tp->repair))
7. WRITE\_ONCE(tp->write\_seq, 0);
8. }
10. inet->inet\_dport = usin->sin\_port;
11. sk\_daddr\_set(sk, daddr);

重置TCP的timestamp。设置目的地址和端口。这里的目的地址有可能跟真正的目的地址不同，见上。

1. tcp\_v4\_connect:
2. inet\_csk(sk)->icsk\_ext\_hdr\_len = 0;
3. **if** (inet\_opt)
4. inet\_csk(sk)->icsk\_ext\_hdr\_len = inet\_opt->opt.optlen;
6. tp->rx\_opt.mss\_clamp = TCP\_MSS\_DEFAULT;

如果设置了选项，那么需要把选项长度设置好。并设置默认的MSS

1. tcp\_v4\_connect:
2. tcp\_set\_state(sk, TCP\_SYN\_SENT);

设置目前的状态为TCP\_SYN\_SENT，记住这个状态。

1. tcp\_v4\_connect:
2. err = inet\_hash\_connect(tcp\_death\_row, sk);
3. **if** (err)
4. **goto** failure;

到这里为止我们已经知道了源，目的地址和目的端口，那么需要给这个链接做散列表，以便进入报文能快速找到相应的处理实例。但，现在我们还不知道源端口，

1. **if** (!inet\_sk(sk)->inet\_num)
2. port\_offset = inet\_sk\_port\_offset(sk);
3. **return** \_\_inet\_hash\_connect(death\_row, sk, port\_offset,
4. \_\_inet\_check\_established);

进入inet\_hash\_connect，先检查是否已经设置了源端口。正常来讲只有SOCK\_RAW类型才会设置源端口。如果没有先计算port\_offset, 在\_\_inet\_hash\_connect找到未经使用的port，并把4元组hash。

1. rt = ip\_route\_newports(fl4, rt, orig\_sport, orig\_dport,
2. inet->inet\_sport, inet->inet\_dport, sk);
3. **if** (IS\_ERR(rt)) {
4. err = PTR\_ERR(rt);
5. rt = NULL;
6. **goto** failure;
7. }

之前找寻路由的时候并没有源端口，所以获得源端口后需要重新更新RT item

1. sk->sk\_gso\_type = SKB\_GSO\_TCPV4;
2. sk\_setup\_caps(sk, &rt->dst);
3. rt = NULL;

设置类型为SKB\_GSO\_TCPV4， GSO（Generic Segmentation Offload）指尽可能推迟数据分片直到发送到网卡之前，此时会检查网卡是否支持分片（TSO，UFO），如果支持就直接把数据发送到网卡，让网卡进行分片。如果不支持就分片，然后发送给网卡。

其中TSO支持TCP数据包，UFO支持UDP数据包

在sock结构体中存储路由缓存。

1. **if** (likely(!tp->repair)) {
2. **if** (!tp->write\_seq)
3. WRITE\_ONCE(tp->write\_seq,
4. secure\_tcp\_seq(inet->inet\_saddr,
5. inet->inet\_daddr,
6. inet->inet\_sport,
7. usin->sin\_port));
8. WRITE\_ONCE(tp->tsoffset,
9. secure\_tcp\_ts\_off(net, inet->inet\_saddr,
10. inet->inet\_daddr));
11. }
13. atomic\_set(&inet->inet\_id, get\_random\_u16());

初始化SEQ，即SYN携带的SEQ

初始化timestamp offset

设置inet\_id

1. **if** (tcp\_fastopen\_defer\_connect(sk, &err))
2. **return** err;
3. **if** (err)
4. **goto** failure;

检查是否设置TCP fastopen。Fastopen是一种简化握手手续的拓展，通过SYN中的TFO cookie（TCP OPTION）来验证一个之前连接过的客户端，如果验证成功，就可以在三次握手中最终的ACK包收到之前就开始发送数据。当然fastopen要两边都支持才能生效。我们例子不涉及fastopen，暂时跳过

1. err = tcp\_connect(sk);
3. **if** (err)
4. **goto** failure;

接下来就是真正的发送SYN包进行连接。

让我们把目光移到tcp\_connect函数：

1. **int** tcp\_connect(**struct** sock \*sk)
2. {
3. **struct** tcp\_sock \*tp = tcp\_sk(sk);
4. **struct** sk\_buff \*buff;
5. **int** err;
7. tcp\_call\_bpf(sk, BPF\_SOCK\_OPS\_TCP\_CONNECT\_CB, 0, NULL);
9. **if** (inet\_csk(sk)->icsk\_af\_ops->rebuild\_header(sk))
10. **return** -EHOSTUNREACH; /\* Routing failure or similar. \*/
12. tcp\_connect\_init(sk);
14. **if** (unlikely(tp->repair)) {
15. tcp\_finish\_connect(sk, NULL);
16. **return** 0;
17. }
19. buff = tcp\_stream\_alloc\_skb(sk, sk->sk\_allocation, **true**);
20. **if** (unlikely(!buff))
21. **return** -ENOBUFS;
23. tcp\_init\_nondata\_skb(buff, tp->write\_seq++, TCPHDR\_SYN);
24. tcp\_mstamp\_refresh(tp);
25. tp->retrans\_stamp = tcp\_time\_stamp(tp);
26. tcp\_connect\_queue\_skb(sk, buff);
27. tcp\_ecn\_send\_syn(sk, buff);
28. tcp\_rbtree\_insert(&sk->tcp\_rtx\_queue, buff);
30. err = tp->fastopen\_req ? tcp\_send\_syn\_data(sk, buff) :
31. tcp\_transmit\_skb(sk, buff, 1, sk->sk\_allocation);
32. **if** (err == -ECONNREFUSED)
33. **return** err;
34. WRITE\_ONCE(tp->snd\_nxt, tp->write\_seq);
35. tp->pushed\_seq = tp->write\_seq;
36. buff = tcp\_send\_head(sk);
37. **if** (unlikely(buff)) {
38. WRITE\_ONCE(tp->snd\_nxt, TCP\_SKB\_CB(buff)->seq);
39. tp->pushed\_seq   = TCP\_SKB\_CB(buff)->seq;
40. }
41. TCP\_INC\_STATS(sock\_net(sk), TCP\_MIB\_ACTIVEOPENS);
43. inet\_csk\_reset\_xmit\_timer(sk, ICSK\_TIME\_RETRANS,
44. inet\_csk(sk)->icsk\_rto, TCP\_RTO\_MAX);
45. **return** 0;
46. }

从前图我们知道icsk\_af\_ops是一个指向ipv4\_specific的指针，所以调用ipv4\_specific->rebuild\_header即inet\_sk\_rebuild\_header函数，这个函数主要是检查路由是否存在，如果存在则什么也不干，直接返回。如果路由不存在，则重新寻找，按寻找结果返回相应的出错值

1. tcp\_connect\_init:
2. tp->tcp\_header\_len = **sizeof**(**struct** tcphdr);
3. **if** (READ\_ONCE(sock\_net(sk)->ipv4.sysctl\_tcp\_timestamps))
4. tp->tcp\_header\_len += TCPOLEN\_TSTAMP\_ALIGNED;

得到TCP header的长度（20字节），如果需要TIMESTAMP选项，继续加上TSTAMP长度（12）。

1. tcp\_connect\_init:
2. #ifdef CONFIG\_TCP\_MD5SIG
3. **if** (tp->af\_specific->md5\_lookup(sk, sk))
4. tp->tcp\_header\_len += TCPOLEN\_MD5SIG\_ALIGNED;
5. #endif

如果支持MD5SIG，加上该选项长度

1. tcp\_connect\_init:
2. **if** (tp->rx\_opt.user\_mss)
3. tp->rx\_opt.mss\_clamp = tp->rx\_opt.user\_mss;
4. tp->max\_window = 0;
5. tcp\_mtup\_init(sk);
6. tcp\_sync\_mss(sk, dst\_mtu(dst));

如果用户有自己的MAX SEGMENT长度，则记录在mss\_clamp字段

设置max\_window为0，这是对端窗口大小，现在不知道对端所以设置为0。

为该socket设置MTU（maximum transmission unit）探测功能，MTU探测由sysctl\_tcp\_mtu\_probing指定。

1. unsigned **int** tcp\_sync\_mss(**struct** sock \*sk, u32 pmtu)
2. {
3. **struct** tcp\_sock \*tp = tcp\_sk(sk);
4. **struct** inet\_connection\_sock \*icsk = inet\_csk(sk);
5. **int** mss\_now;
7. **if** (icsk->icsk\_mtup.search\_high > pmtu)
8. icsk->icsk\_mtup.search\_high = pmtu;
10. mss\_now = tcp\_mtu\_to\_mss(sk, pmtu);
11. mss\_now = tcp\_bound\_to\_half\_wnd(tp, mss\_now);
13. /\* And store cached results \*/
14. icsk->icsk\_pmtu\_cookie = pmtu;
15. **if** (icsk->icsk\_mtup.enabled)
16. mss\_now = min(mss\_now, tcp\_mtu\_to\_mss(sk, icsk->icsk\_mtup.search\_low));
17. tp->mss\_cache = mss\_now;
19. **return** mss\_now;
20. }

同步MSS，如果前面的MTU比当前的PMTU（path MTU）还要大，就设置为当前的PMTU。

计算MSS，需要去掉相应的头部（如果PTMU是1500，那么MSS就是1460）。

最后把MSS保存到tp->mss\_cache

1. tcp\_connect\_init
2. tcp\_ca\_dst\_init(sk, dst);

初始化相应的拥塞控制。

1. tcp\_connect\_init
2. **if** (!tp->window\_clamp)
3. tp->window\_clamp = dst\_metric(dst, RTAX\_WINDOW);
4. tp->advmss = tcp\_mss\_clamp(tp, dst\_metric\_advmss(dst));
6. tcp\_initialize\_rcv\_mss(sk);

如果没有设置window的最大值，就使用下一跳的最大窗口值

设置advmss即出口MSS值，用于跟对端交换的MSS。

rcv\_mss是接收MSS的大小，现在是猜测对端的MSS。Rcv\_mss下限是88字节。

1. tcp\_connect\_init
2. **if** (sk->sk\_userlocks & SOCK\_RCVBUF\_LOCK &&
3. (tp->window\_clamp > tcp\_full\_space(sk) || tp->window\_clamp == 0))
4. tp->window\_clamp = tcp\_full\_space(sk);

如果窗口通告大小比用户设置接收缓存即tcp\_rmem[1]还要大或没有设置窗口大小，设置窗口通告大小为接收缓存的大小。这里注意如果设置了TCP ADV WIN SCALE，返回的窗口大小是或来计算的。假如tcp\_adv\_win\_scale为1，rcvbuf为131072，则窗口最大值为65536，因为TCP的缓存是用户和内核共享的，为了隔离网络延时和本机调度延时，所以会留出一部分给应用作为缓存。

1. tcp\_connect\_init
2. rcv\_wnd = tcp\_rwnd\_init\_bpf(sk);
3. **if** (rcv\_wnd == 0)
4. rcv\_wnd = dst\_metric(dst, RTAX\_INITRWND);

计算初始接收窗口，或者从下一跳窗口获得，注意这个rcv\_wnd是多少个MSS的意思，真正窗口计算的时候需要乘MSS的大小。

1. tcp\_connect\_init
2. tcp\_select\_initial\_window(sk, tcp\_full\_space(sk),
3. tp->advmss - (tp->rx\_opt.ts\_recent\_stamp ? tp->tcp\_header\_len - **sizeof**(**struct** tcphdr) : 0),
4. &tp->rcv\_wnd,
5. &tp->window\_clamp,
6. READ\_ONCE(sock\_net(sk)->ipv4.sysctl\_tcp\_window\_scaling),
7. &rcv\_wscale,
8. rcv\_wnd);
10. tp->rx\_opt.rcv\_wscale = rcv\_wscale;
11. tp->rcv\_ssthresh = tp->rcv\_wnd;

计算接收窗口，保存在tp->rcv\_wnd上。如果接收窗口大于UINT16\_MAX，则需要把窗口阶系数保存到rx\_opt.rcv\_wscale上，以便于发送SYN的时候通告对端。

1. tcp\_connect\_init
2. WRITE\_ONCE(sk->sk\_err, 0);
3. sock\_reset\_flag(sk, SOCK\_DONE);
4. tp->snd\_wnd = 0;
5. tcp\_init\_wl(tp, 0);
6. tcp\_write\_queue\_purge(sk);
7. tp->snd\_una = tp->write\_seq;
8. tp->snd\_sml = tp->write\_seq;
9. tp->snd\_up = tp->write\_seq;
10. WRITE\_ONCE(tp->snd\_nxt, tp->write\_seq);

清空error值，重置flag，初始期望接收窗口为0

初始窗口更新序列为0

未确认序列是write\_seq，最近传输小包的最后一个字节为write\_seq，紧急指针write\_seq。下次发送的序列号为write\_seq

1. tcp\_connect\_init
2. **if** (likely(!tp->repair))
3. tp->rcv\_nxt = 0;
4. **else**
5. tp->rcv\_tstamp = tcp\_jiffies32;
6. tp->rcv\_wup = tp->rcv\_nxt;
7. WRITE\_ONCE(tp->copied\_seq, tp->rcv\_nxt);
9. inet\_csk(sk)->icsk\_rto = tcp\_timeout\_init(sk);
10. inet\_csk(sk)->icsk\_retransmits = 0;
11. tcp\_clear\_retrans(tp);

初始化下一个接收序列及最后一次发送窗口更新时的rcv\_nxt（rcv\_wup）

未读的数据开始（copied\_seq）， copied\_seq之前的数据表示已经读取。

初始化重传计时器数值，以及重传的一系列标值。

1. tcp\_connect
2. **if** (unlikely(tp->repair)) {
3. tcp\_finish\_connect(sk, NULL);
4. **return** 0;
5. }
7. buff = tcp\_stream\_alloc\_skb(sk, sk->sk\_allocation, **true**);
8. **if** (unlikely(!buff))
9. **return** -ENOBUFS;

回到tcp\_connect函数，检查是否为TCP repair （这里有一种情况就是把正在运行的容器从一个主机迁移到另一个主机，就需要考虑TCP repair，这里不讨论这种情况）

分配skb buffer，如果分配不出来，就会进入TCP内存压力状态，直到内存压力解除为止

1. tcp\_connect
2. tcp\_init\_nondata\_skb(buff, tp->write\_seq++, TCPHDR\_SYN);

初始化TCP header，这里并没有真正写入data，在skbuff里面有一个cb[48]的数组，临时编成TCP control block，记录发送的sequence，标记以及其他

1. tcp\_connect
2. tcp\_mstamp\_refresh(tp);
3. tp->retrans\_stamp = tcp\_time\_stamp(tp);

记录最近一次时间（纳秒级），最近发送/接收数据包时间，第一次SYN发送时间（也包括最近一次重传时间）

1. tcp\_connect
2. tcp\_connect\_queue\_skb(sk, buff);
3. tcp\_ecn\_send\_syn(sk, buff);
4. tcp\_rbtree\_insert(&sk->tcp\_rtx\_queue, buff);

tcp\_connect\_queu\_skb主要记录发送数据的大小，记录在sk\_wmem\_queued里面。还需要把sk\_forward\_alloc的值减去需要发送的长度。

Tcp\_ecn\_send\_syn，显式拥塞窗口通告，这里的显示指的是路由器在出现拥塞的时候会标记IP头部的2位来记录，当TCP段到达后，接收方知道这个报文已经经历过拥塞。但是问题是发送方并不知道，所以当接收方看到ECE|CWR后，就会在下一个ACK里面通知发送方，路径有拥塞。

把数据报文压入红黑树里面缓存。

1. tcp\_connect
2. err = tp->fastopen\_req ? tcp\_send\_syn\_data(sk, buff) :
3. tcp\_transmit\_skb(sk, buff, 1, sk->sk\_allocation);
4. **if** (err == -ECONNREFUSED)
5. **return** err;

发送报文，这里需要判断是否打开fastopen

1. tcp\_connect
2. WRITE\_ONCE(tp->snd\_nxt, tp->write\_seq);
3. tp->pushed\_seq = tp->write\_seq;
4. buff = tcp\_send\_head(sk);
5. **if** (unlikely(buff)) {
6. WRITE\_ONCE(tp->snd\_nxt, TCP\_SKB\_CB(buff)->seq);
7. tp->pushed\_seq   = TCP\_SKB\_CB(buff)->seq;
8. }
9. TCP\_INC\_STATS(sock\_net(sk), TCP\_MIB\_ACTIVEOPENS);
11. inet\_csk\_reset\_xmit\_timer(sk, ICSK\_TIME\_RETRANS,
12. inet\_csk(sk)->icsk\_rto, TCP\_RTO\_MAX);
13. **return** 0;

接下来最主要的就是重置超时定时器，如果超时需要重新发送SYN

然后重新回到tcp\_v4\_connect函数，没有错误，回到\_\_inet\_stream\_connect函数中。

1. \_\_inet\_stream\_connect
2. sock->state = SS\_CONNECTING;
4. **if** (!err && inet\_sk(sk)->defer\_connect)
5. **goto** out;
7. /\* Just entered SS\_CONNECTING state; the only
8. \* difference is that return value in non-blocking
9. \* case is EINPROGRESS, rather than EALREADY.
10. \*/
11. err = -EINPROGRESS;
12. **break**;

设置sock状态，设置err为-EINPROGRESS，这个error值主要是为了非阻塞返回的时候判断。

1. \_\_inet\_stream\_connect
2. timeo = sock\_sndtimeo(sk, flags & O\_NONBLOCK);
4. **if** ((1 << sk->sk\_state) & (TCPF\_SYN\_SENT | TCPF\_SYN\_RECV)) {
5. **int** writebias = (sk->sk\_protocol == IPPROTO\_TCP) &&
6. tcp\_sk(sk)->fastopen\_req &&
7. tcp\_sk(sk)->fastopen\_req->data ? 1 : 0;
9. **if** (!timeo || !inet\_wait\_for\_connect(sk, timeo, writebias))
10. **goto** out;
12. err = sock\_intr\_errno(timeo);
13. **if** (signal\_pending(current))
14. **goto** out;
15. }
17. **if** (sk->sk\_state == TCP\_CLOSE)
18. **goto** sock\_error;
20. sock->state = SS\_CONNECTED;
21. err = 0;
22. out:
23. **return** err;

获取timeout值，如果是非阻塞的话为0

现在的TCP状态为TCP\_SYN\_SENT，我们不考虑fastopen，writebias为0

如果是非阻塞，就返回EINPROGRESS，表示还没有完成connect，应用需要进一步检查，如果是阻塞，那就进入inet\_wait\_for\_connect进行等待直到超时或者ACK回来。最后设置状态为SS\_CONNECTED，后返回

至此connect就运行结束。接下来就是等待服务器的SYN+ACK响应，当服务器的SYN+ACK报文返回后，由ip分发后进入tcp\_v4\_do\_rcv，我们先略过一些处理方案，直接进入tcp\_rcv\_state\_process

1. tcp\_rcv\_state\_process
2. **case** TCP\_SYN\_SENT:
3. tp->rx\_opt.saw\_tstamp = 0;
4. tcp\_mstamp\_refresh(tp);
5. queued = tcp\_rcv\_synsent\_state\_process (sk, skb, th);
6. **if** (queued >= 0)
7. **return** queued;
9. tcp\_urg(sk, skb, th);
10. \_\_kfree\_skb(skb);
11. tcp\_data\_snd\_check(sk);
12. **return** 0;

前面说过当前的TCP状态时TCP\_SYN\_SENT，进入函数后执行case

第3，4行计算timestamp，这里saw\_tstamp是对端的timestamp是否存在，然后进入tcp\_rcv\_synsent\_state\_process函数

1. tcp\_rcv\_synsent\_state\_process
2. tcp\_parse\_options(sock\_net(sk), skb, &tp->rx\_opt, 0, &foc);
3. **if** (tp->rx\_opt.saw\_tstamp && tp->rx\_opt.rcv\_tsecr)
4. tp->rx\_opt.rcv\_tsecr -= tp->tsoffset;

解析TCP头部选项部分，后详解

接下来处理是否是ACK报文，服务器端发送的是SYN+ACK，所以进入ack处理

1. tcp\_rcv\_synsent\_state\_process
2. **if** (th->ack) {
3. **if** (!after(TCP\_SKB\_CB(skb)->ack\_seq, tp->snd\_una) ||
4. after(TCP\_SKB\_CB(skb)->ack\_seq, tp->snd\_nxt)) {
5. **if** (icsk->icsk\_retransmits == 0)
6. inet\_csk\_reset\_xmit\_timer(sk,
7. ICSK\_TIME\_RETRANS,
8. TCP\_TIMEOUT\_MIN, TCP\_RTO\_MAX);
9. **goto** reset\_and\_undo;
10. }

检查序列号是否符合ack > ISS 或者 ack = snd\_nxt，否则如果还没有进行重传，重置超时定时器，进入reset\_and\_undo分支，丢弃已经保存的TCP选项，重置MSS

1. tcp\_rcv\_synsent\_state\_process
2. **if** (th->ack) {
3. **if** (tp->rx\_opt.saw\_tstamp && tp->rx\_opt.rcv\_tsecr &&
4. !between(tp->rx\_opt.rcv\_tsecr, tp->retrans\_stamp,
5. tcp\_time\_stamp(tp))) {
6. NET\_INC\_STATS(sock\_net(sk),
7. LINUX\_MIB\_PAWSACTIVEREJECTED);
8. **goto** reset\_and\_undo;
9. }

检查时间戳是否合适，否则进入reset\_and\_undo分支

1. tcp\_rcv\_synsent\_state\_process
2. **if** (th->ack) {
3. **if** (th->rst) {
4. tcp\_reset(sk, skb);
5. consume:
6. \_\_kfree\_skb(skb);
7. **return** 0;
8. }

运行到此意味着ACK已经是合法的，检查是否由rst标记，如果有，进行reset处理。向应用报告connect reset，关闭socket（设置SOCK\_CLOSED），丢弃报文等。

1. tcp\_rcv\_synsent\_state\_process
2. **if** (!th->syn) {
3. SKB\_DR\_SET(reason, TCP\_FLAGS);
4. **goto** discard\_and\_undo;
5. }

数据报文是SYN+ACK，跳过该处