TCP 高级实验

王鹿鸣, 刘保证

2016 年 6 月 14 日

目录

CHAPTER 1

准备部分

Contents

1.1 用户层 TCP

用户层的 TCP 编程模型大致如下，对于服务端，调用 listen 监听端口，之后接受客户端的请求，然后就可以收发数据了。结束时，关闭 socket。

```
// Server
socket(...,SOCK_STREAM,0);
bind(...,&server_address, ...);
listen(...);
accept(..., &client_address, ...);
recv(..., &clientaddr, ...);
close(...);
```

对于客户端，则调用 connect 连接服务端，之后便可以收发数据。最后关闭 socket。

```
socket(...,SOCK_STREAM,0);
connect();
send(...,&server_address,...);
```

那么根据我们的需求，我们着重照顾连接的建立、关闭和封包的收发过程。

1.2 探寻 tcp_prot，地图 get~

一般游戏的主角手中，都会有一张万能的地图。为了搞定 TCP，我们自然也是需要一张地图的，要不连该去找那个函数看都不知道。很有幸，在`tcp_ipv4.c`中，`tcp_prot`定义了`tcp`的各个接口。

`tcp_prot`的类型为`struct proto`，是这个结构体是为了抽象各种不同的协议的差异性而存在的。类似面向对象中所说的接口 (Interface) 的概念。这里，我们仅保留我们关系的部分。

```
struct proto tcp_prot = {
        .name                   = "TCP",
        .owner                  = THIS_MODULE,
        .close                  = tcp_close,
        .connect                = tcp_v4_connect,
        .disconnect             = tcp_disconnect,
        .accept                 = inet_csk_accept,
        .destroy                = tcp_v4_destroy_sock,
        .shutdown               = tcp_shutdown,
        .setsockopt             = tcp_setsockopt,
        .getsockopt             = tcp_getsockopt,
        .recvmsg                = tcp_recvmsg,
        .sendmsg                = tcp_sendmsg,
        .sendpage               = tcp_sendpage,
        .backlog_rcv            = tcp_v4_do_rcv,
        .get_port               = inet_csk_get_port,
        .twsk_prot              = &tcp_timewait_sock_ops,
        .rsk_prot               = &tcp_request_sock_ops,
};
```

通过名字，我大致筛选出来了这些函数，初步判断这些函数与实验所关心的功能相关。对着这张“地图”，就可以顺藤摸瓜，找出些路径了。

先根据参考书《Linux 内核源码剖析——TCP/IP 实现》中给出的流程图，找出所有和需求相关的部分。

首先找三次握手相关的部分: 从客户端的角度, 发起连接需要调用`tcp_v4_connect`, 该函数会进一步调用`tcp_connect`,在这个函数中,会调用`tcp_send_syn_data` 发送 SYN 报文, 并设定超时计时器。第二次握手相关的接收代码在`tcp_rcv_state_process`中, 该函数实现了除`ESTABLISHED`和`TIME_WAIT`之外所有状态下的接收处理。`tcp_send_ack`函数实现了发送 ACK 报文。从服务端的角度, 则还需实现`listen`调用和 `accept`调用。二者都是服务端建立连接所需要的部分。

封包的封装发送部分，所对应的函数是`tcp_sendmsg`，实现对数据的复制、切割和发送。TCP 的重传接口为`tcp_retransmit_skb`，这里尚有疑问，因为这个函数是负责处理重传的, 而不是判断是否应当重传的。所以并不明确到底是否该重新实现这一部分。

TCP 封包的接收在`tcp_rcv_established`函数中, 根据目前有限的资料看, TCP 的滑动窗口机制应该在这一部分，更细节的内容待确认。

- tcp_transmit_skb
- tcp_rcv_state_process
- tcp_connect
- tcp_rcv_synsent_state_process
- tcp_rcv_established
- tcp_send_ack
- tcp_sendmsg
- tcp_retransmit_skb
- tcp_rcv_established

1.3 RFC

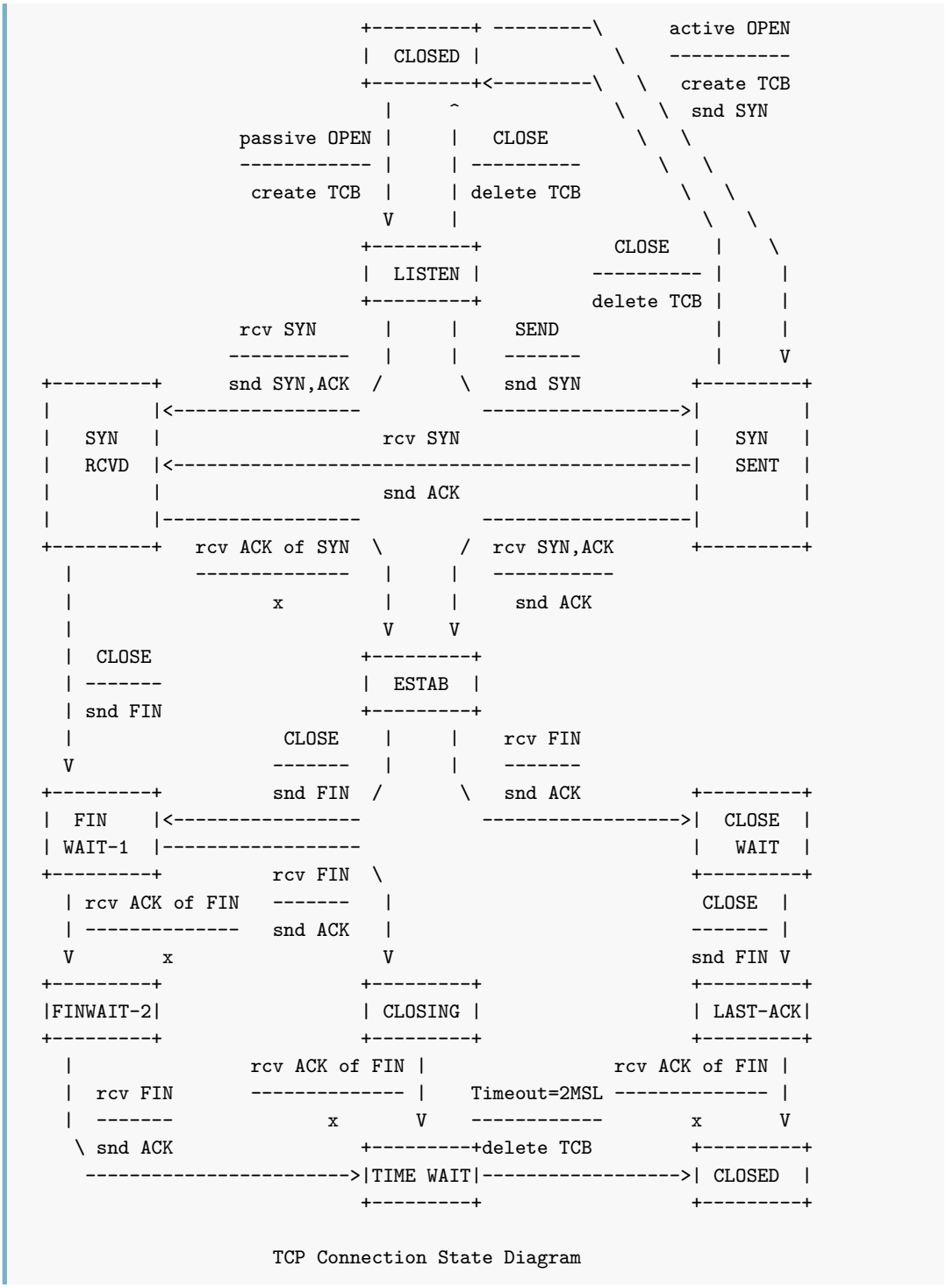
在分析 TCP 的过程中会遇到很多 RFC，在这里，我们将可能会碰到的 RFC 罗列出来，并进行一定的讨论，便于后面的分析。

1.3.1 RFC793——Transmission Control Protocol

该 RFC 正是定义了 TCP 协议的那份 RFC。在该 RFC 中，可以查到 TCP 的很多细节，帮助后续的代码分析。

1.3.1.1 TCP 状态图

在 RFC793 中，给出了 TCP 协议的状态图，图中的 TCB 代表 TCP 控制块。原图如下所示:



TCP Connection State Diagram

这张图对于后面的分析有很强的指导意义。

连接部分分为了主动连接和被动连接。主动连接是指客户端从 CLOSED 状态主动发出连接请求，进入 SYN-SENT 状态，之后收到服务端的 SYN+ACK 包，进入 ESTAB 状态(即连接建立状态)，然后回复 ACK 包，完成三次握手。这一部分的代码我们将在3.1中进行详细分析。被动连接是从 listen 状态开始，监听端口。随后收到 SYN 包，进入 SYN-RCVD 状态，同时发送 SYN+ACK 包，最后，收到 ACK 后，进入 ESTAB 状态，完成被动连接的三次握手过程。这一部分的详细讨论在3.2中完成。

连接终止的部分也被分为了两部分进行实现，主动终止和被动终止。主动终止是上

图中从 ESTAB 状态主动终止连接，发送 FIN 包的过程。可以看到，主动终止又分为两种情况，一种是 FIN 发出后，收到了发来的 FIN（即通信双方同时主动关闭连接），此时转入 CLOSING 状态并发送 ACK 包。收到 ACK 后，进入 TIME WAIT 状态。另一种是收到了 ACK 包，转入 FINWAIT-2 状态，最后收到 FIN 后发送 ACK，完成四次握手，进入 TIME WAIT 状态。最后等数据发送完或者超时后，删除 TCB，进入 CLOSED 状态。被动终止则是接收到 FIN 包后，发送了 ACK 包，进入 CLOSE WAIT 状态。之后，当这一端的数据也发送完成后，发送 FIN 包，进入 LAST-ACK 状态，接收到 ACK 后，进入 CLOSED 状态。

1.3.1.2 TCP 头部格式

RFC793 中，对于 TCP 头部格式的描述摘录如下：

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |           |U|A|P|R|S|F|                               |
| Offset| Reserved  |R|C|S|S|Y|I|            Window             |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

                        TCP Header Format

      Note that one tick mark represents one bit position.
```

这张图可以很方便地读出各个位占多长，它上面的标识是十进制的，很容易读。这里我们挑出我们比较关心的 Options 字段来解读。因为很多 TCP 的扩展都是通过新增选项来实现的。

选项总是在 TCP 头部的最后，且长度是 8 位的整数倍。全部选项都被包括在校验和中。选项可从任何字节边界开始。选项的格式有 2 种情况**有待补充**：

1. 单独的一个字节，代表选项的类型例如：

   ```
   End of Option List
   +--------+
   |00000000|
   +--------+
   Kind=0
   ```

2. 第一个字节代表选项的类型，紧跟着的一个字节代表选项的长度，后面跟着选项的数据。例如：

```
Maximum Segment Size
+--------+--------+---------+--------+
|00000010|00000100| max seg size     |
+--------+--------+---------+--------+
  Kind=2  Length=4
```

1.3.2 RFC1323——TCP Extensions for High Performance

1.3.2.1 简介

这个 RFC 主要是考虑高带宽高延迟网络下如何提升 TCP 的性能。该 RFC 定义了新的 TCP 选项，以实现窗口缩放 (window scaled) 和时间戳 (timestamp)。这里的时间戳可以用于实现两个机制：RTTM(Round Trip Time Measurement) 和 PAWS(Protect Against Wrapped Sequences)。

在 RFC1323 中提出，在这类高带宽高延迟网络下，有三个主要的影响 TCP 性能的因素：

窗口尺寸限制 在 TCP 头部中，只有 16 位的一个域用于说明窗口大小。也就是说，窗口大小最大只能达到 $2^{16} = 64K$ 字节。解决这一问题的方案是增加一个窗口缩放选项，我们会在1.3.2.2中进一步讨论。

丢包后的恢复 丢包会导致 TCP 重新进入慢启动状态，导致数据的流水线断流。在引入了快重传和快恢复后，可以解决丢包率为一个窗口中丢一个包的情况下的问题。但是在引入了窗口缩放以后，由于窗口的扩大，丢包的概率也随之增加。很容易使 TCP 进入到慢启动状态，影响网络性能。为了解决这一问题，需要引入 SACK 机制，但在这个 RFC 中，不讨论 SACK 相关的问题。

往返时间度量 RTO(Retransmission timeout) 是 TCP 性能的一个很基础的参数。在 RFC1323 中介绍了一种名为 RTTM 的机制，利用一个新的名为 Timestamps 的选项来对时间进行进一步的统计。

1.3.2.2 窗口缩放 (Window Scale)

这一扩展将原有的 TCP 窗口扩展到 32 位。而根据 RFC793 中的定义，TCP 头部描述窗口大小的域仅有 16 位。为了将其扩展为 32 位，该扩展定义了一个新的选项用于表示缩放因子。这一选项仅会出现在 SYN 段，此后，所有通过该连接的通信，其窗口大小都会受到这一选项的影响。

该选项的格式为：

```
+---------+---------+---------+
| Kind=3  |Length=3 |shift.cnt|
+---------+---------+---------+
```

`kind`域为 3，`length`域为 3，后面跟着 3 个字节的 `shift.cnt`，代表缩放因子。TCP 的 Options 域的选项的格式在 1.3.1.2中已有说明。这里采用的是第二种格式。

在启用了窗口缩放以后，TCP 头部中的接收窗口大小，就变为了真实的接收窗口大小右移 `shift.cnt`的值。RFC 中对于该选项的实现的建议是在传输控制块中，按照 32 位整型来存储所有的窗口值，包括发送窗口、接受窗口和拥塞窗口。

作为接收方，每当收到一个段时（除 SYN 段外），通过将发来的窗口值左移来得到正确的值。

```
SND.WND = SEG.WND << Snd.Wind.Scale
```

作为发送方则每次在发包前，将发送窗口的值右移，然后再封装在封包中。

```
SEG.WND = RCV.WND >> Rcv.Wind.Scale
```

1.3.2.3 PAWS(Protect Against Wrapped Sequence Numbers)

PAWS 是一个用于防止旧的重复封包带来的问题的机制。它采用了 TCP 中的 Timestamps 选项。该选项的格式为：

```
+-------+-------+---------------------+---------------------+
|Kind=8 |  10   |   TS Value (TSval)  |TS Echo Reply (TSecr)|
+-------+-------+---------------------+---------------------+
    1       1              4                     4
```

TSval(Timestamp Value) 包含了 TCP 发送方的时钟的当前值。如果 ACK 位被设置了的话，TSecr(Timestamp Echo Reply) 会包含一个由对方发送过来的最近的时钟值。

PAWS 的算法流程为：

1. 如果当前到达的段 SEG 含有 Timestamps 选项，且 SEG.TSval < TS.Recent 且该 TS.Recent 是有效的，那么则认为当前到达的分段是不可被接受的，需要丢弃掉。
2. 如果段超过了窗口的范围，则丢弃它（与正常的 TCP 处理相同）
3. 如果满足 SEG.SEQ≤Last.ACK.sent（最后回复的 ACK 包中的时间戳），那么，将 SEG 的时间戳记录为 TS.Recent。
4. 如果 SEG 是正常按顺序到达的，那么正常地接收它。
5. 其他情况下，将该段视作正常的在窗口中，但顺序不正确的 TCP 段对待。

1.3.3 RFC1337——TIME-WAIT Assassination Hazards in TCP

在 TCP 连接中，存在`TIME_WAIT`这样一个阶段。该阶段会等待 2MSL 的时间，以使得属于当前连接的所有的包都消失掉。这样可以保证再次用相同端口建立连接时，不会有属于上一个连接的滞留在网络中的包对连接产生干扰。

1.3.3.1 TIME-WAIT Assassination(TWA) 现象

TCP 的连接是由四元组（源 IP 地址，源端口，目的 IP 地址，目的端口）唯一决定的。但是，存在这样一种情况，当一个 TCP 连接关闭，随后，客户端又使用相同的 IP 和端口号向服务端发起连接，即产生了和之前的连接一模一样的四元组。此时，如果网络中还存在上一个连接遗留下来的包，就会出现各类的问题。对于这一问题，RFC793 中定义了相关的机制进行应对。

1. 三次握手时会拒绝旧的 SYN 段，以避免重复建立连接。
2. 通过判断序列号可以有效地拒绝旧的或者重复的段被错误地接受。
3. 通过选择合适的 ISN(Initial Sequence Number) 可以避免旧的连接和新的连接的段的序列号空间发生重叠。
4. TIME-WAIT 状态会等待足够长的时间，让旧的滞留在网络上的段因超过其生命周期而消失。
5. 在系统崩溃后，在系统启动时的静默时间可以使旧的段在连接开始前消失。

然而，其中的`TIME_WAIT`状态的相关机制却是不可靠的。网络中滞留的段有可能会使得`TIME_WAIT`状态被意外结束。这一现象即是 TIME-WAIT Assassination 现象。RFC1337 中给出了一个实例：

```
      TCP A                                                TCP B

1.  ESTABLISHED                                          ESTABLISHED

      (Close)
2.  FIN-WAIT-1  --> <SEQ=100><ACK=300><CTL=FIN,ACK>  --> CLOSE-WAIT

3.  FIN-WAIT-2  <-- <SEQ=300><ACK=101><CTL=ACK>      <-- CLOSE-WAIT

                                                         (Close)
4.  TIME-WAIT   <-- <SEQ=300><ACK=101><CTL=FIN,ACK>  <-- LAST-ACK

5.  TIME-WAIT   --> <SEQ=101><ACK=301><CTL=ACK>      --> CLOSED

- - - - - - - - - - - - - - - - - - - - - - - - - -

5.1. TIME-WAIT   <--  <SEQ=255><ACK=33> ... old duplicate

5.2  TIME-WAIT   --> <SEQ=101><ACK=301><CTL=ACK>    -->  ????

5.3  CLOSED      <-- <SEQ=301><CTL=RST>             <--  ????
   (prematurely)
```

可以看到，TCP A 收到了一个遗留的 ACK 包，之后响应了这个 ACK。TCP B 收到这个莫名其妙的响应后，会发出 RST 报，因为它认为发生了错误。收到 RST 包后，TCP A 的 TIME-WAIT 状态被终止了。然而，此时还没有到 2MSL 的时间。

1.3.3.2 TWA 的危险性及现有的解决方法

RFC1337 中列举了三种 TWA 现象带来的危险。

1. 滞留在网络上的旧的数据段可能被错误地接受。
2. 新的连接可能陷入到不同步的状态中。如接收到一个旧的 ACK 包等情况。
3. 新的连接可能被滞留在网络中的旧的 FIN 包关闭。或者是 SYN-SENT 状态下出现了意料之外的 ACK 包等。都可能导致新的连接被终止。

而解决 TWA 问题的方法较为简单，直接在 TIME-WAIT 阶段忽略掉所有的 RST 段即可。在4.1.6中的代码中可以看到，Linux 正是采用了这种方法来解决该问题。

1.3.4 RFC2525——Known TCP Implementation Problems

1.3.5 RFC3168——The Addition of Explicit Congestion Notification (ECN) to IP

该 RFC 为 TCP/IP 网络引入了一种新的用于处理网络拥塞问题的方法。在以往的 TCP 中，人们假定网络是一个不透明的黑盒。而发送方通过丢包的情况来推测路由器处发生了拥塞。这种拥塞控制方法对于交互式的应用效果并不理想，且可能对吞吐量造成负面影响。因为此时网络可能已经计入到了拥塞状态（路由器的缓存满了，因此发生了丢包)。为了解决这一问题，人们已经引入了主动队列管理算法（AQM)。允许路由器在拥塞发生的早期，即队列快要满了的时候，就通过主动丢包的方式，告知发送端要减慢发送速率。然而，主动丢包会引起包的重传，这会降低网络的性能。因此，RFC3168 引入了 ECN（显式拥塞通知）机制。

1.3.5.1 ECN

在引入 ECN 机制后，AQM 通知发送端的方式不再局限于主动丢包，而是可以通过 IP 头部的 Congestion Experienced(CE) 来通知支持 ECN 机制的发送端发生了拥塞。这样，接收端可以不必通过丢包来实现 AQM，减少了发送端因丢包而产生的性能下降。

ECN 在 IP 包中的域如下所示：

```
+-----+-----+
| ECN FIELD |
+-----+-----+
ECT  CE  [Obsolete] RFC 2481 names for the ECN bits.
 0   0   Not-ECT
 0   1   ECT(1)
 1   0   ECT(0)
 1   1   CE
```

Not-ECT 代表该包没有采用 ECN 机制。ECT(1) 和 ECT(0) 均代表该设备支持 ECN 机制。CE 用于路由器向发送端表明网络是否正在经历拥塞状态。

1.3.5.2 TCP 中的 ECN

在 TCP 协议中，利用 TCP 头部的保留位为 ECN 提供了支持。这里规定了两个新的位：ECE 和 CWR。ECE 用于在三次握手中协商手否启用 ECN 功能。CWR 用于让接收端决定合适可以停止设置 ECE 标志。增加 ECN 支持后的 TCP 头部示意图如下：

```
0     1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|               |               | C | E | U | A | P | R | S | F |
| Header Length |   Reserved    | W | C | R | C | S | S | Y | I |
|               |               | R | E | G | K | H | T | N | N |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

对于一个 TCP 连接，一个典型的基于 ECN 的序列如下：

1. 发送方在发送的包中设置 ECT 位，表明发送端支持 ECN。
2. 在该包经历一个支持 ECN 的路由器时，该路由器发现了拥塞，准备主动丢包。此时，它发现正要丢弃的包支持 ECN。它会放弃丢包，而是设置 IP 头部的 CE 位，表明经历了拥塞。
3. 接收方收到设置了 CE 位的包后，在回复的 ACK 中，设置 ECE 标记。
4. 发送端收到了一个设定了 ECE 标记的 ACK 包，进入和发生丢包一样的拥塞控制状态。
5. 发送端在下一个要发送的包中设定 CWR 位，以通知接收方它已经对 ECE 位进行了相应处理。

通过这种方式，就可以在经历网络拥塞时减少丢包的情况，并通知客户端网络的拥塞状态。

1.3.6 RFC6937–Proportional Rate Reduction for TCP

1.3.7 RFC7413——TCP Fast Open(Draft)

RFC7413 目前还处于草案状态，它引入了一种试验性的特性：允许在三次握手阶段的 SYN 和 SYN-ACK 包中携带数据。相较于标准的三次握手，引入 TFO(TCP Fast Open) 机制可以节省一个 RTT 的时间。该 RFC 由 Google 提交，并在 Linux 和 Chrome 中实现了对该功能的支持。此后，越来越多的软件也支持了该功能。

1.3.7.1 概要

TFO 中，最核心的一个部分是 Fast Open Cookie。这个 Cookie 是由服务器生成的，在初次进行常规的 TCP 连接时，由客户端向服务器请求，之后，则可利用这个 Cookie 在后续的 TCP 连接中在三次握手阶段交换数据。

请求 Fast Open Cookie 的过程为：

1. 客户端发送一个带有 Fast Open 选项且 cookie 域为空的 SYN 包
2. 服务器生成一个 cookie，通过 SYN-ACK 包回复给客户端
3. 客户端将该 cookie 缓存，并用于后续的 TCP Fast Open 连接。

建立 Fast Open 连接的过程如下：

1. 客户端发送一个带有数据的 SYN 包，同时在 Fast Open 选项里带上 cookie。
2. 服务端验证 cookie 的有效性，如果有效，则回复 SYN-ACK，并将数据发给应用程序。
3. 客户端发送 ACK 请求，确认服务器发来的 SYN-ACK 包及其中的数据。
4. 至此，三次握手已完成，后续过程与普通 TCP 连接一致。

1.3.7.2 Fast Open 选项格式

该选项的格式如下：

```
                                +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
                                |  Kind = 34    |    Length     |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                                                             |
   ~                           Cookie                            ~
   |                                                             |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   Length          1 byte: range 6 to 18 (bytes); limited by
                           remaining space in the options field.
                           The number MUST be even.
   Cookie          0, or 4 to 16 bytes (Length - 2)
```

这里注意，虽然图示中 Cookie 按照 32 位对齐了，但是这并不是必须的。

CHAPTER 2

网络子系统相关核心数据结构

Contents

2.1 网络子系统数据结构架构

2.2 sock 底层数据结构

2.2.1 sock_common

```
/**
 *  struct sock_common - minimal network layer representation of sockets
 *  @skc_daddr: Foreign IPv4 addr
 *  @skc_rcv_saddr: Bound local IPv4 addr
 *  @skc_hash: hash value used with various protocol lookup tables
 *  @skc_u16hashes: two u16 hash values used by UDP lookup tables
 *  @skc_dport: placeholder for inet_dport/tw_dport
 *  @skc_num: placeholder for inet_num/tw_num
 *  @skc_family: network address family
 *  @skc_state: Connection state
 *  @skc_reuse: %SO_REUSEADDR setting
 *  @skc_reuseport: %SO_REUSEPORT setting
 *  @skc_bound_dev_if: bound device index if != 0
 *  @skc_bind_node: bind hash linkage for various protocol lookup tables
 *  @skc_portaddr_node: second hash linkage for UDP/UDP-Lite protocol
 *  @skc_prot: protocol handlers inside a network family
 *  @skc_net: reference to the network namespace of this socket
 *  @skc_node: main hash linkage for various protocol lookup tables
 *  @skc_nulls_node: main hash linkage for TCP/UDP/UDP-Lite protocol
 *  @skc_tx_queue_mapping: tx queue number for this connection
 *  @skc_flags: place holder for sk_flags
 *      %SO_LINGER (l_onoff), %SO_BROADCAST, %SO_KEEPALIVE,
 *      %SO_OOBINLINE settings, %SO_TIMESTAMPING settings
 *  @skc_incoming_cpu: record/match cpu processing incoming packets
 *  @skc_refcnt: reference count
 *
 *  This is the minimal network layer representation of sockets, the header
 *  for struct sock and struct inet_timewait_sock.
 */
struct sock_common {
    /* skc_daddr and skc_rcv_saddr must be grouped on a 8 bytes aligned
     * address on 64bit arches : cf INET_MATCH()
     */
    union {
        __addrpair  skc_addrpair;
        struct {
            __be32  skc_daddr;
            __be32  skc_rcv_saddr;
        };
    };
    union  {
        unsigned int    skc_hash;
        __u16       skc_u16hashes[2];
    };
    /* skc_dport && skc_num must be grouped as well */
    union {
```

```
47          __portpair  skc_portpair;
48          struct {
49              __be16  skc_dport;
50              __u16   skc_num;
51          };
52      };
53
54      unsigned short      skc_family;
55      volatile unsigned char  skc_state;
56      unsigned char       skc_reuse:4;
57      unsigned char       skc_reuseport:1;
58      unsigned char       skc_ipv6only:1;
59      unsigned char       skc_net_refcnt:1;
60      int         skc_bound_dev_if;
61      union {
62          struct hlist_node   skc_bind_node;
63          struct hlist_nulls_node skc_portaddr_node;
64      };
65      struct proto        *skc_prot;
66      possible_net_t      skc_net;
67
68  #if IS_ENABLED(CONFIG_IPV6)
69      struct in6_addr     skc_v6_daddr;
70      struct in6_addr     skc_v6_rcv_saddr;
71  #endif
72
73      atomic64_t      skc_cookie;
74
75      /* following fields are padding to force
76       * offset(struct sock, sk_refcnt) == 128 on 64bit arches
77       * assuming IPV6 is enabled. We use this padding differently
78       * for different kind of 'sockets'
79       */
80      union {
81          unsigned long   skc_flags;
82          struct sock *skc_listener; /* request_sock */
83          struct inet_timewait_death_row *skc_tw_dr; /* inet_timewait_sock */
84      };
85      /*
86       * fields between dontcopy_begin/dontcopy_end
87       * are not copied in sock_copy()
88       */
89      /* private: */
90      int         skc_dontcopy_begin[0];
91      /* public: */
92      union {
93          struct hlist_node   skc_node;
94          struct hlist_nulls_node skc_nulls_node;
95      };
96      int         skc_tx_queue_mapping;
97      union {
98          int     skc_incoming_cpu;
99          u32     skc_rcv_wnd;
100         u32     skc_tw_rcv_nxt; /* struct tcp_timewait_sock  */
101     };
102
103     atomic_t        skc_refcnt;
104     /* private: */
```

```
105     int                     skc_dontcopy_end[0];
106     union {
107         u32     skc_rxhash;
108         u32     skc_window_clamp;
109         u32     skc_tw_snd_nxt; /* struct tcp_timewait_sock */
110     };
111     /* public: */
112 };
```

2.2.2 sock

sock 结构是比较通用的网络层描述块，构成传输控制块的基础，与具体的协议族无关。它描述了各协议族的公共信息，因此不能直接作为传输层控制块来使用。不同协议族的传输层在使用该结构的时候都会对其进行拓展，来适合各自的传输特性。例如，`inet_sock`结构由 sock 结构及其它特性组成，构成了 IPV4 协议族传输控制块的基础。结构如下：

```
1  /**
2    * struct sock - network layer representation of sockets
3    * @__sk_common: shared layout with inet_timewait_sock
4    * @sk_shutdown: SEND_SHUTDOWN 或者 RCV_SHUTDOWN 的掩码
5    * @sk_userlocks: %SO_SNDBUF and %SO_RCVBUF settings
6    * @sk_lock:   synchronizer
7    * @sk_rcvbuf: 接受缓冲区的大小（单位为字节）
8    * @sk_wq: sock wait queue and async head
9    * @sk_rx_dst: receive input route used by early demux
10   * @sk_dst_cache: destination cache
11   * @sk_policy: flow policy
12   * @sk_receive_queue: incoming packets
13   * @sk_wmem_alloc: transmit queue bytes committed
14   * @sk_write_queue: Packet sending queue
15   * @sk_omem_alloc: "o" is "option" or "other"
16   * @sk_wmem_queued: persistent queue size
17   * @sk_forward_alloc: space allocated forward
18   * @sk_napi_id: id of the last napi context to receive data for sk
19   * @sk_ll_usec: usecs to busypoll when there is no data
20   * @sk_allocation: allocation mode
21   * @sk_pacing_rate: Pacing rate (if supported by transport/packet scheduler)
22   * @sk_max_pacing_rate: Maximum pacing rate (%SO_MAX_PACING_RATE)
23   * @sk_sndbuf: size of send buffer in bytes
24   * @sk_no_check_tx: %SO_NO_CHECK setting, set checksum in TX packets
25   * @sk_no_check_rx: allow zero checksum in RX packets
26   * @sk_route_caps: route capabilities (e.g. %NETIF_F_TSO)
27   * @sk_route_nocaps: forbidden route capabilities (e.g NETIF_F_GSO_MASK)
28   * @sk_gso_type: GSO type (e.g. %SKB_GSO_TCPV4)
29   * @sk_gso_max_size: Maximum GSO segment size to build
30   * @sk_gso_max_segs: Maximum number of GSO segments
31   * @sk_lingertime: %SO_LINGER l_linger setting
32   * @sk_backlog: always used with the per-socket spinlock held
33   * @sk_callback_lock: used with the callbacks in the end of this struct
34   * @sk_error_queue: rarely used
35   * @sk_prot_creator: sk_prot of original sock creator (see ipv6_setsockopt,
36   *           IPV6_ADDRFORM for instance)
37   * @sk_err: last error
38   * @sk_err_soft: errors that don't cause failure but are the cause of a
39   *           persistent failure not just 'timed out'
```

```
40    * @sk_drops: raw/udp drops counter
41    * @sk_ack_backlog: current listen backlog
42    * @sk_max_ack_backlog: listen backlog set in listen()
43    * @sk_priority: %SO_PRIORITY setting
44    * @sk_cgrp_prioidx: socket group's priority map index
45    * @sk_type: socket type (%SOCK_STREAM, etc)
46    * @sk_protocol: which protocol this socket belongs in this network family
47    * @sk_peer_pid: &struct pid for this socket's peer
48    * @sk_peer_cred: %SO_PEERCRED setting
49    * @sk_rcvlowat: %SO_RCVLOWAT setting
50    * @sk_rcvtimeo: %SO_RCVTIMEO setting
51    * @sk_sndtimeo: %SO_SNDTIMEO setting
52    * @sk_txhash: computed flow hash for use on transmit
53    * @sk_filter: socket filtering instructions
54    * @sk_timer: sock cleanup timer
55    * @sk_stamp: time stamp of last packet received
56    * @sk_tsflags: SO_TIMESTAMPING socket options
57    * @sk_tskey: counter to disambiguate concurrent tstamp requests
58    * @sk_socket: Identd and reporting IO signals
59    * @sk_user_data: RPC layer private data
60    * @sk_frag: cached page frag
61    * @sk_peek_off: current peek_offset value
62    * @sk_send_head: 发送队列的头指针
63    * @sk_security: used by security modules
64    * @sk_mark: generic packet mark
65    * @sk_classid: this socket's cgroup classid
66    * @sk_cgrp: this socket's cgroup-specific proto data
67    * @sk_write_pending: a write to stream socket waits to start
68    * @sk_state_change: callback to indicate change in the state of the sock
69    * @sk_data_ready: callback to indicate there is data to be processed
70    * @sk_write_space: callback to indicate there is bf sending space available
71    * @sk_error_report: callback to indicate errors (e.g. %MSG_ERRQUEUE)
72    * @sk_backlog_rcv: callback to process the backlog
73    * @sk_destruct: called at sock freeing time, i.e. when all refcnt == 0
74   */
75  struct sock {
76      /*
77       * Now struct inet_timewait_sock also uses sock_common, so please just
78       * don't add nothing before this first member (__sk_common) --acme
79       */
80      struct sock_common  __sk_common;
81  #define sk_node         __sk_common.skc_node
82  #define sk_nulls_node       __sk_common.skc_nulls_node
83  #define sk_refcnt       __sk_common.skc_refcnt
84  #define sk_tx_queue_mapping __sk_common.skc_tx_queue_mapping
85
86  #define sk_dontcopy_begin   __sk_common.skc_dontcopy_begin
87  #define sk_dontcopy_end     __sk_common.skc_dontcopy_end
88  #define sk_hash         __sk_common.skc_hash
89  #define sk_portpair     __sk_common.skc_portpair
90  #define sk_num          __sk_common.skc_num
91  #define sk_dport        __sk_common.skc_dport
92  #define sk_addrpair     __sk_common.skc_addrpair
93  #define sk_daddr        __sk_common.skc_daddr
94  #define sk_rcv_saddr        __sk_common.skc_rcv_saddr
95  #define sk_family       __sk_common.skc_family
96  #define sk_state        __sk_common.skc_state
97  #define sk_reuse        __sk_common.skc_reuse
```

```
98  #define sk_reuseport        __sk_common.skc_reuseport
99  #define sk_ipv6only     __sk_common.skc_ipv6only
100 #define sk_net_refcnt       __sk_common.skc_net_refcnt
101 #define sk_bound_dev_if     __sk_common.skc_bound_dev_if
102 #define sk_bind_node        __sk_common.skc_bind_node
103 #define sk_prot         __sk_common.skc_prot
104 #define sk_net          __sk_common.skc_net
105 #define sk_v6_daddr     __sk_common.skc_v6_daddr
106 #define sk_v6_rcv_saddr __sk_common.skc_v6_rcv_saddr
107 #define sk_cookie       __sk_common.skc_cookie
108 #define sk_incoming_cpu     __sk_common.skc_incoming_cpu
109 #define sk_flags        __sk_common.skc_flags
110 #define sk_rxhash       __sk_common.skc_rxhash
111
112     socket_lock_t       sk_lock;
113     struct sk_buff_head sk_receive_queue;
114     /*
115      * The backlog queue is special, it is always used with
116      * the per-socket spinlock held and requires low latency
117      * access. Therefore we special case it's implementation.
118      * Note : rmem_alloc is in this structure to fill a hole
119      * on 64bit arches, not because its logically part of
120      * backlog.
121      */
122     struct {
123         atomic_t    rmem_alloc;
124         int     len;
125         struct sk_buff  *head;
126         struct sk_buff  *tail;
127     } sk_backlog;
128 #define sk_rmem_alloc sk_backlog.rmem_alloc
129     int         sk_forward_alloc;
130
131     __u32           sk_txhash;
132 #ifdef CONFIG_NET_RX_BUSY_POLL
133     unsigned int        sk_napi_id;
134     unsigned int        sk_ll_usec;
135 #endif
136     atomic_t        sk_drops;
137     int         sk_rcvbuf;
138
139     struct sk_filter __rcu  *sk_filter;
140     union {
141         struct socket_wq __rcu  *sk_wq;
142         struct socket_wq    *sk_wq_raw;
143     };
144 #ifdef CONFIG_XFRM
145     struct xfrm_policy __rcu *sk_policy[2];
146 #endif
147     struct dst_entry    *sk_rx_dst;
148     struct dst_entry __rcu  *sk_dst_cache;
149     /* Note: 32bit hole on 64bit arches */
150     atomic_t        sk_wmem_alloc;
151     atomic_t        sk_omem_alloc;
152     int         sk_sndbuf;
153     struct sk_buff_head sk_write_queue;
154     kmemcheck_bitfield_begin(flags);
155     unsigned int        sk_shutdown  : 2,
156                 sk_no_check_tx : 1,
```

```
                sk_no_check_rx : 1,
                sk_userlocks : 4,
                sk_protocol  : 8,
                sk_type      : 16;
#define SK_PROTOCOL_MAX U8_MAX
    kmemcheck_bitfield_end(flags);
    int         sk_wmem_queued;
    gfp_t           sk_allocation;
    u32         sk_pacing_rate; /* bytes per second */
    u32         sk_max_pacing_rate;
    netdev_features_t   sk_route_caps;
    netdev_features_t   sk_route_nocaps;
    int         sk_gso_type;
    unsigned int        sk_gso_max_size;
    u16         sk_gso_max_segs;
    int         sk_rcvlowat;
    unsigned long           sk_lingertime;
    struct sk_buff_head sk_error_queue;
    struct proto        *sk_prot_creator;
    rwlock_t        sk_callback_lock;
    int         sk_err,
                sk_err_soft;
    u32         sk_ack_backlog;
    u32         sk_max_ack_backlog;
    __u32           sk_priority;
#if IS_ENABLED(CONFIG_CGROUP_NET_PRIO)
    __u32           sk_cgrp_prioidx;
#endif
    struct pid      *sk_peer_pid;
    const struct cred   *sk_peer_cred;
    long            sk_rcvtimeo;
    long            sk_sndtimeo;
    struct timer_list   sk_timer;
    ktime_t         sk_stamp;
    u16         sk_tsflags;
    u32         sk_tskey;
    struct socket       *sk_socket;
    void            *sk_user_data;
    struct page_frag    sk_frag;
    struct sk_buff      *sk_send_head;
    __s32           sk_peek_off;
    int         sk_write_pending;
#ifdef CONFIG_SECURITY
    void            *sk_security;
#endif
    __u32           sk_mark;
#ifdef CONFIG_CGROUP_NET_CLASSID
    u32         sk_classid;
#endif
    struct cg_proto     *sk_cgrp;
    void            (*sk_state_change)(struct sock *sk);
    void            (*sk_data_ready)(struct sock *sk);
    void            (*sk_write_space)(struct sock *sk);
    void            (*sk_error_report)(struct sock *sk);
    int         (*sk_backlog_rcv)(struct sock *sk,
                          struct sk_buff *skb);
    void                (*sk_destruct)(struct sock *sk);
};
```

2.2.3 request_sock

该结构位于/include/net/request_sock.h中。该结构用于表示一个简单的 TCP 连接请求。

```
/* struct request_sock - mini sock to represent a connection request
 */
struct request_sock {
    struct sock_common      __req_common;
#define rsk_refcnt          __req_common.skc_refcnt
#define rsk_hash            __req_common.skc_hash
#define rsk_listener            __req_common.skc_listener
#define rsk_window_clamp        __req_common.skc_window_clamp
#define rsk_rcv_wnd         __req_common.skc_rcv_wnd

    struct request_sock     *dl_next;
    u16             mss;
    u8              num_retrans; /* number of retransmits */
    u8              cookie_ts:1; /* syncookie: encode tcpopts in timestamp */
    u8              num_timeout:7; /* number of timeouts */
    u32             ts_recent;
    struct timer_list       rsk_timer;
    const struct request_sock_ops   *rsk_ops;
    struct sock         *sk;
    u32             *saved_syn;
    u32             secid;
    u32             peer_secid;
};
```

2.2.4 sk_buff

struct sk_buff这一结构体在各层协议中都会被用到。该结构体存储了网络数据报的所有信息。包括各层的头部以及 payload，以及必要的各层实现相关的信息。

该结构体的定义较长，需要一点一点分析。结构体的开头为

```
  union {
    struct {
      /* These two members must be first. */
      struct sk_buff          *next;
      struct sk_buff          *prev;

      union {
        ktime_t         tstamp;
        struct skb_mstamp skb_mstamp;
      };
    };
    struct rb_node  rbnode; /* used in netem and tcp stack */
  };
```

可以看到，sk_buff可以被组织成两种数据结构：双向链表和红黑树。且一个sk_buff不是在双向链表中，就是在红黑树中，因此，采用了 union 来节约空间。next 和 prev 两个域是用于双向链表的结构体，而 rbnode 是红黑树相关的结构。

包的到达/发送时间存放在union {ktime_t tstamp;struct skb_mstamp skb_mstamp;};中，之所以这里有两种不同的时间戳类型，是因为有时候调用ktime_get()的成本太高。因

此,内核开发者希望能够在 TCP 协议栈中实现一个轻量级的微秒级的时间戳。struct skb_mstamp正是结合了local_clock()和 jiffies二者，而实现的一个轻量级的工具。当然，根据内核邮件列表中的说法，并不是任何时候都可以用该工具替换调ktime_get()的。因此，在struct sk_buff结构体中，采用union的方式同时保留了这二者。

在定义完数据结构相关的一些部分后，又定义了如下的结构体

```
  /* 拥有该 sk_buff 的套接字的指针 */
  struct sock             *sk;
  /* 与该包关联的网络设备 */
  struct net_device       *dev;
  /* 控制用的缓冲区，用于存放各层的私有数据 */
  char                    cb[48] __aligned(8);
  /* 存放了目的地项的引用计数 */
  unsigned long           _skb_refdst;
  /* 析构函数 */
  void                    (*destructor)(struct sk_buff *skb);
#ifdef CONFIG_XFRM
  /* xfrm 加密通道 */
  struct  sec_path        *sp;
#endif
#if IS_ENABLED(CONFIG_BRIDGE_NETFILTER)
  /* 保存和 bridge 相关的信息 */
  struct nf_bridge_info   *nf_bridge;
#endif
```

其中的char cb[48]比较有意思，各层都使用这个 buffer 来存放自己私有的变量。这里值得注意的是，如果想要跨层传递数据，则需要使用 skb_clone()。XFRM 则是 Linux 在 2.6 版本中引入的一个安全方面的扩展。

之后，又定义了一些长度相关的字段。len代表 buffer 中的数据长度（含各协议的头部），以及分片长度。而len代表分片中的数据的长度。mac_len是 MAC 层头部的长度。hdr_len是一个克隆出来的可写的头部的长度。

```
unsigned int            len,
                        data_len;
__u16                   mac_len,
                        hdr_len;
```

kmemcheck 是内核中的一套内存检测工具。kmemcheck_bitfield_begin 和 kmemcheck_bitfield_end 可以用于说明一段内容的起始和终止位置。其代码定义如下：

```
#define kmemcheck_bitfield_begin(name)  \
        int name##_begin[0];

#define kmemcheck_bitfield_end(name)    \
        int name##_end[0];
```

通过定义，我们不难看出，这两个宏是用于在代码中产生两个对应于位域的起始地址和终止地址的符号的。当然，这两个宏是为 kmemcheck 的功能服务的。如果没有开启该功能的话，这两个宏的定义为空，也即不会产生任何作用。

```
/* Following fields are _not_ copied in __copy_skb_header()
* Note that queue_mapping is here mostly to fill a hole.
*/
kmemcheck_bitfield_begin(flags1);
__u16       queue_mapping;      /* 对于多队列设备的队列关系映射                    */
__u8        cloned:1,           /* 是否被克隆                                  */
            nohdr:1,            /* 只引用了负载                                  */
            fclone:2,           /* skbuff 克隆的情况                             */
            peeked:1,           /* peeked 表明该包已经被统计过了，无需再次统计 */
            head_frag:1,
            xmit_more:1;        /* 在队列中有更多的 SKB 在等待                    */
/* one bit hole */
kmemcheck_bitfield_end(flags1);
```

在这段定义中，内核将一系列的标志位命名为了 flags1，利用那两个函数可以在生成的代码中插入 `flags1_begin`和`flags1_end`两个符号。这样，当有需要的时候，可以通过这两个符号找到这一段的起始地址和结束地址。

紧接着是一个包的头部，这一部分再次使用了类似上面的方法，用了两个零长度的数组 `headers_start`和`headers_end`来标明头部的起始和终止地址。

```
        /* 在 __copy_skb_header() 中，只需使用一个 memcpy() 即可将 headers_start/end
         * 之间的部分克隆一份。
         */
        /* private: */
        __u32                   headers_start[0];
        /* public: */

/* if you move pkt_type around you also must adapt those constants */
#ifdef __BIG_ENDIAN_BITFIELD
#define PKT_TYPE_MAX    (7 << 5)
#else
#define PKT_TYPE_MAX    7
#endif
#define PKT_TYPE_OFFSET()       offsetof(struct sk_buff, __pkt_type_offset)

        __u8                    __pkt_type_offset[0];
        /* 该包的类型 */
        __u8                    pkt_type:3;
        __u8                    pfmemalloc:1;
        /* 是否允许本地分片 (local fragmentation) */
        __u8                    ignore_df:1;
        /* 表明该 skb 和连接的关系 */
        __u8                    nfctinfo:3;
        /* netfilter 包追踪标记 */
        __u8                    nf_trace:1;
        /* 驱动（硬件）给出来的 checksum */
        __u8                    ip_summed:2;
        /* 允许该 socket 到队列的对应关系发生变更 */
        __u8                    ooo_okay:1;
        /* 表明哈希值字段 hash 是一个典型的 4 元组的通过传输端口的哈希 */
        __u8                    l4_hash:1;
        /* 表明哈希值字段 hash 是通过软件栈计算出来的 */
        __u8                    sw_hash:1;
        /* 表明 wifi_acked 是否被设置了 */
        __u8                    wifi_acked_valid:1;
        /* 表明帧是否在 wifi 上被确认了 */
```

```
37          __u8                    wifi_acked:1;
38
39          /* 请求 NIC 将最后的 4 个字节作为以太网 FCS 来对待 */
40          __u8                    no_fcs:1;
41          /* Indicates the inner headers are valid in the skbuff. */
42          __u8                    encapsulation:1;
43          __u8                    encap_hdr_csum:1;
44          __u8                    csum_valid:1;
45          __u8                    csum_complete_sw:1;
46          __u8                    csum_level:2;
47          __u8                    csum_bad:1;
48
49  #ifdef CONFIG_IPV6_NDISC_NODETYPE
50          __u8                    ndisc_nodetype:2; /* 路由类型（来自链路层） */
51  #endif
52          /* 标明该 skbuff 是否被 ipvs 拥有 */
53          __u8                    ipvs_property:1;
54          __u8                    inner_protocol_type:1;
55          __u8                    remcsum_offload:1;
56          /* 3 or 5 bit hole */
57
58  #ifdef CONFIG_NET_SCHED
59          __u16                   tc_index;       /* traffic control index */
60  #ifdef CONFIG_NET_CLS_ACT
61          __u16                   tc_verd;        /* traffic control verdict */
62  #endif
63  #endif
64
65          union {
66                  __wsum          csum; /* 校验码 */
67                  struct {
68                          /* 从 skb->head 开始到应当计算校验码的起始位置的偏移 */
69                          __u16   csum_start;
70                          /* 从 csum_start 开始到存储校验码的位置的偏移 */
71                          __u16   csum_offset;
72                  };
73          };
74          __u32                   priority; /* 包队列的优先级 */
75          int                     skb_iif; /* 到达的设备的序号 */
76          __u32                   hash; /* 包的哈希值 */
77          __be16                  vlan_proto; /* vlan 包装协议 */
78          __u16                   vlan_tci; /* vlan tag 控制信息 */
79  #if defined(CONFIG_NET_RX_BUSY_POLL) || defined(CONFIG_XPS)
80          union {
81                  unsigned int    napi_id; /* 表明该 skb 来源的 NAPI 结构体的 id */
82                  unsigned int    sender_cpu;
83          };
84  #endif
85          union {
86  #ifdef CONFIG_NETWORK_SECMARK
87                  __u32           secmark; /* 安全标记 */
88  #endif
89  #ifdef CONFIG_NET_SWITCHDEV
90                  __u32           offload_fwd_mark; /* fwding offload mark */
91  #endif
92          };
93
94          union {
95                  __u32           mark; /* 通用的包的标记位 */
```

```
                __u32           reserved_tailroom;
        };

        union {
                __be16          inner_protocol; /* 协议（封装好的） */
                __u8            inner_ipproto;
        };

        /* 已封装的内部传输层头部 */
        __u16                   inner_transport_header;
        /* 已封装的内部网络层头部 */
        __u16                   inner_network_header;
        /* 已封装的内部链路层头部 */
        __u16                   inner_mac_header;

        /* 驱动（硬件）给出的包的协议类型 */
        __be16                  protocol;
        /* 传输层头部 */
        __u16                   transport_header;
        /* 网络层头部 */
        __u16                   network_header;
        /* 数据链路层头部 */
        __u16                   mac_header;

        /* private: */
        __u32                   headers_end[0];
```

最后是一组是管理相关的字段。其中，head和end 代表被分配的内存的起始位置和终止位置。而data和tail 则是实际数据的起始和终止位置。

```
/* These elements must be at the end, see alloc_skb() for details.  */
        sk_buff_data_t          tail;
        sk_buff_data_t          end;
        unsigned char           *head,
                                *data;
        unsigned int            truesize;
        atomic_t                users;
```

users是引用计数，所以是个原子的。truesize是数据报的真实大小。

2.3 inet 层相关数据结构

2.3.1 ip_options

```
/*
Location:

Description:
        @faddr - Saved first hop address
        @nexthop - Saved nexthop address in LSRR and SSRR
        @is_strictroute - Strict source route
        @srr_is_hit - Packet destination addr was our one
        @is_changed - IP checksum more not valid
        @rr_needaddr - Need to record addr of outgoing dev
        @ts_needtime - Need to record timestamp
        @ts_needaddr - Need to record addr of outgoing dev
*/
```

```
struct ip_options {
        __be32                faddr;
        __be32                nexthop;
        unsigned char        optlen;
        unsigned char        srr;
        unsigned char        rr;
        unsigned char        ts;
        unsigned char        is_strictroute:1,
                        srr_is_hit:1,
                        is_changed:1,
                        rr_needaddr:1,
                        ts_needtime:1,
                        ts_needaddr:1;
        unsigned char        router_alert;
        unsigned char        cipso;
        unsigned char        __pad2;
        unsigned char        __data[0];
};
```

2.3.2 inet_request_sock

该结构位于/include/net/inet_sock.h中。

这个结构的功能呢？

```
struct inet_request_sock {
    struct request_sock req;
#define ir_loc_addr     req.__req_common.skc_rcv_saddr
#define ir_rmt_addr     req.__req_common.skc_daddr
#define ir_num          req.__req_common.skc_num
#define ir_rmt_port     req.__req_common.skc_dport
#define ir_v6_rmt_addr      req.__req_common.skc_v6_daddr
#define ir_v6_loc_addr      req.__req_common.skc_v6_rcv_saddr
#define ir_iif          req.__req_common.skc_bound_dev_if
#define ir_cookie       req.__req_common.skc_cookie
#define ireq_net        req.__req_common.skc_net
#define ireq_state      req.__req_common.skc_state
#define ireq_family     req.__req_common.skc_family

    kmemcheck_bitfield_begin(flags);
    u16         snd_wscale : 4,
                rcv_wscale : 4,
                tstamp_ok  : 1,
                sack_ok    : 1,
                wscale_ok  : 1,
                ecn_ok     : 1,
                acked      : 1,
                no_srccheck: 1;
    kmemcheck_bitfield_end(flags);
    u32                     ir_mark;
    union {
        struct ip_options_rcu   *opt;
        struct sk_buff      *pktopts;
    };
};
```

2.3.3 inet_connection_sock_af_ops

该结构位于/include/net/inet_connect_sock.h中, 其后面的 af 表示 address of function 即函数地址, ops 表示 operations，即操作。

该结构封装了一组与传输层有关的操作集,包括向网络层发送的接口、传输层的setsockopt接口等。

```
struct inet_connection_sock_af_ops {
    int     (*queue_xmit)(struct sock *sk, struct sk_buff *skb, struct flowi *fl);
    void        (*send_check)(struct sock *sk, struct sk_buff *skb);
    int     (*rebuild_header)(struct sock *sk);
    void        (*sk_rx_dst_set)(struct sock *sk, const struct sk_buff *skb);
    int     (*conn_request)(struct sock *sk, struct sk_buff *skb);
    struct sock *(*syn_recv_sock)(const struct sock *sk, struct sk_buff *skb,
                      struct request_sock *req,
                      struct dst_entry *dst,
                      struct request_sock *req_unhash,
                      bool *own_req);
    u16     net_header_len;
    u16     net_frag_header_len;
    u16     sockaddr_len;
    int     (*setsockopt)(struct sock *sk, int level, int optname,
                  char __user *optval, unsigned int optlen);
    int     (*getsockopt)(struct sock *sk, int level, int optname,
                  char __user *optval, int __user *optlen);
#ifdef CONFIG_COMPAT
    int     (*compat_setsockopt)(struct sock *sk,
                int level, int optname,
                char __user *optval, unsigned int optlen);
    int     (*compat_getsockopt)(struct sock *sk,
                int level, int optname,
                char __user *optval, int __user *optlen);
#endif
    void        (*addr2sockaddr)(struct sock *sk, struct sockaddr *);
    int     (*bind_conflict)(const struct sock *sk,
                     const struct inet_bind_bucket *tb, bool relax);
    void        (*mtu_reduced)(struct sock *sk);
};
```

2.3.4 inet_connect_sock

该结构位于/include/net/inet_connect_sock.h中，它是所有面向传输控制块的表示。其在inet_sock的基础上，增加了有关连接，确认，重传等成员。

```
/** inet_connection_sock - INET connection oriented sock
 *
 * @icsk_accept_queue:     FIFO of established children
 * @icsk_bind_hash:    Bind node
 * @icsk_timeout:      Timeout
 * @icsk_retransmit_timer: Resend (no ack)
 * @icsk_rto:          Retransmit timeout
 * @icsk_pmtu_cookie       Last pmtu seen by socket
 * @icsk_ca_ops        Pluggable congestion control hook
 * @icsk_af_ops        Operations which are AF_INET{4,6} specific
 * @icsk_ca_state:     拥塞控制状态
 * @icsk_retransmits:      Number of unrecovered [RTO] timeouts
```

```
 * @icsk_pending:      Scheduled timer event
 * @icsk_backoff:      Backoff
 * @icsk_syn_retries:      Number of allowed SYN (or equivalent) retries
 * @icsk_probes_out:       unanswered 0 window probes
 * @icsk_ext_hdr_len:      Network protocol overhead (IP/IPv6 options)
 * @icsk_ack:          Delayed ACK control data
 * @icsk_mtup;         MTU probing control data
 */
struct inet_connection_sock {
    /* inet_sock has to be the first member! */
    struct inet_sock      icsk_inet;
    struct request_sock_queue icsk_accept_queue;
    struct inet_bind_bucket   *icsk_bind_hash;
    unsigned long         icsk_timeout;
    struct timer_list     icsk_retransmit_timer;
    struct timer_list     icsk_delack_timer;
    __u32             icsk_rto;
    __u32             icsk_pmtu_cookie;
    const struct tcp_congestion_ops *icsk_ca_ops;
    const struct inet_connection_sock_af_ops *icsk_af_ops;
    unsigned int          (*icsk_sync_mss)(struct sock *sk, u32 pmtu);
    __u8              icsk_ca_state:6,
                  icsk_ca_setsockopt:1,
                  icsk_ca_dst_locked:1;
    __u8              icsk_retransmits;
    __u8              icsk_pending;
    __u8              icsk_backoff;
    __u8              icsk_syn_retries;
    __u8              icsk_probes_out;
    __u16             icsk_ext_hdr_len;
    struct {
        __u8          pending;   /* ACK is pending             */
        __u8          quick;     /* Scheduled number of quick acks     */
        __u8          pingpong;  /* The session is interactive         */
        __u8          blocked;   /* Delayed ACK was blocked by socket lock */
        __u32         ato;       /* Predicted tick of soft clock       */
        unsigned long     timeout;   /* Currently scheduled timeout        */
        __u32         lrcvtime;  /* timestamp of last received data packet */
        __u16         last_seg_size; /* Size of last incoming segment      */
        __u16         rcv_mss;   /* MSS used for delayed ACK decisions     */
    } icsk_ack;
    struct {
        int       enabled;

        /* Range of MTUs to search */
        int       search_high;
        int       search_low;

        /* Information on the current probe. */
        int       probe_size;

        u32       probe_timestamp;
    } icsk_mtup;
    u32           icsk_user_timeout;

    u64           icsk_ca_priv[64 / sizeof(u64)]; /* 拥塞控制算法私有空间 */
#define ICSK_CA_PRIV_SIZE      (8 * sizeof(u64))
};
```

2.3.5 inet_timewait_sock

在进入到等待关闭的状态时，已经不需要完整的传输控制块了。Linux 为了减轻在重负载情况下的内存消耗，定义了这个简化的结构体。

```
/* include/net/inet_timewait_sock.h
 *
 * 这个结构体的存在主要是为了解决重负载情况下的内存负担问题。
 */
struct inet_timewait_sock {
        /* 此处也使用了 sock_common 结构体。
         * Now struct sock also uses sock_common, so please just
         * don't add nothing before this first member (__tw_common) --acme
         */

        struct sock_common      __tw_common;
        /* 通过宏定义为 sock_common 中的结构体起一个 tw 开头的别名。 */
#define tw_family               __tw_common.skc_family
#define tw_state                __tw_common.skc_state
#define tw_reuse                __tw_common.skc_reuse
#define tw_ipv6only             __tw_common.skc_ipv6only
#define tw_bound_dev_if         __tw_common.skc_bound_dev_if
#define tw_node                 __tw_common.skc_nulls_node
#define tw_bind_node            __tw_common.skc_bind_node
#define tw_refcnt               __tw_common.skc_refcnt
#define tw_hash                 __tw_common.skc_hash
#define tw_prot                 __tw_common.skc_prot
#define tw_net                  __tw_common.skc_net
#define tw_daddr                __tw_common.skc_daddr
#define tw_v6_daddr             __tw_common.skc_v6_daddr
#define tw_rcv_saddr            __tw_common.skc_rcv_saddr
#define tw_v6_rcv_saddr         __tw_common.skc_v6_rcv_saddr
#define tw_dport                __tw_common.skc_dport
#define tw_num                  __tw_common.skc_num
#define tw_cookie               __tw_common.skc_cookie
#define tw_dr                   __tw_common.skc_tw_dr

        /* 超时时间 */
        int                     tw_timeout;
        /* 子状态，用于区分 FIN_WAIT2 和 TIMEWAIT */
        volatile unsigned char  tw_substate;
        /* 窗口缩放 */
        unsigned char           tw_rcv_wscale;

        /* 下面的部分都和 inet_sock 中的成员相对应的。 */
        __be16                  tw_sport;
        kmemcheck_bitfield_begin(flags);
        /* And these are ours. */
        unsigned int            tw_kill         : 1,
                                tw_transparent  : 1,
                                tw_flowlabel    : 20,
                                tw_pad          : 2,    /* 2 bits hole */
                                tw_tos          : 8;
        kmemcheck_bitfield_end(flags);
        /* 超时计时器 */
        struct timer_list       tw_timer;
        struct inet_bind_bucket *tw_tb;
};
```

2.3.6 sockaddr & sockaddr_in

sockaddr用于描述一个地址。

```
/* include/linux/socket.h */
struct sockaddr {
        sa_family_t     sa_family;      /* 地址所属的协议族，AF_xxx       */
        char            sa_data[14];    /* 在协议下的地址 */
};
```

可以看出sockaddr是一个较为通用的描述方法。可以支持任意的网络层协议。那么具体到我们的情况，就是 IP 网络。下面是 IP 网络下，该结构体的定义。

```
/* include/uapi/linux/in.h
 * 该结构体用于描述一个 Internet (IP) 套接字的地址
 */
struct sockaddr_in {
  __kernel_sa_family_t  sin_family;     /* 这里和 sockaddr 是对应的，填写 IP 网络 */
  __be16                sin_port;       /* 端口号                                 */
  struct in_addr        sin_addr;       /* Internet 地址                          */

  /* 填充位，为了将 sockaddr_in 填充到和 sockaddr 一样长 */
  unsigned char         __pad[__SOCK_SIZE__ - sizeof(short int) -
                        sizeof(unsigned short int) - sizeof(struct in_addr)];
};
```

sockaddr的使用方法是在需要的地方直接强制转型成相应网络的结构体。因此，需要让二者一样大。这就是为何sockaddr_in要加填充位的原因。

2.3.7 ip_options

```
/** struct ip_options - IP Options
 *
 * @faddr - 保存的第一跳地址
 * @nexthop - 保存在 LSRR 和 SSRR 的下一跳地址
 * @is_strictroute - 严格的源路由
 * @srr_is_hit - 包目标地址命中
 * @is_changed - IP 校验和不合法
 * @rr_needaddr - 需要记录出口设备的地址
 * @ts_needtime - 需要记录时间戳
 * @ts_needaddr - 需要记录出口设备的地址
 */
struct ip_options {
        __be32          faddr;
        __be32          nexthop;
        unsigned char   optlen;
        unsigned char   srr;
        unsigned char   rr;
        unsigned char   ts;
        unsigned char   is_strictroute:1,
                        srr_is_hit:1,
                        is_changed:1,
                        rr_needaddr:1,
                        ts_needtime:1,
                        ts_needaddr:1;
        unsigned char   router_alert;
        unsigned char   cipso;
```

```
        unsigned char   __pad2;
        unsigned char   __data[0];
};

struct ip_options_rcu {
        struct rcu_head rcu;
        struct ip_options opt;
};
```

2.4 路由相关数据结构

2.4.1 dst_entry

该结构位于/include/net/dst.h中。

最终生成的 IP 数据报的路由称为目的入口 (dst_entry)，目的入口反映了相邻的外部主机在本地主机内部的一种“映象”。它是与协议无关的目的路由缓存相关的数据结构，保护了路由缓存链接在一起的数据结构成员变量、垃圾回收相关的成员变量、邻居项相关的成员、二层缓存头相关的成员、输入/输出函数指针以用于命中路由缓存的数据包进行后续的数据处理等。

```
/* Each dst_entry has reference count and sits in some parent list(s).
 * When it is removed from parent list, it is "freed" (dst_free).
 * After this it enters dead state (dst->obsolete > 0) and if its refcnt
 * is zero, it can be destroyed immediately, otherwise it is added
 * to gc list and garbage collector periodically checks the refcnt.
 */
struct dst_entry {
    struct rcu_head     rcu_head;
    struct dst_entry    *child;
    struct net_device       *dev;
    struct  dst_ops         *ops;
    unsigned long       _metrics;
    unsigned long           expires;
    struct dst_entry    *path;
    struct dst_entry    *from;
#ifdef CONFIG_XFRM
    struct xfrm_state   *xfrm;
#else
    void            *__pad1;
#endif
    int         (*input)(struct sk_buff *);
    int         (*output)(struct net *net, struct sock *sk, struct sk_buff *skb);

    unsigned short      flags;
#define DST_HOST        0x0001
#define DST_NOXFRM      0x0002
#define DST_NOPOLICY        0x0004
#define DST_NOHASH      0x0008
#define DST_NOCACHE     0x0010
#define DST_NOCOUNT     0x0020
#define DST_FAKE_RTABLE     0x0040
#define DST_XFRM_TUNNEL     0x0080
#define DST_XFRM_QUEUE      0x0100
#define DST_METADATA        0x0200

```

```
36      unsigned short      pending_confirm;
37
38      short           error;
39
40      /* A non-zero value of dst->obsolete forces by-hand validation
41       * of the route entry.  Positive values are set by the generic
42       * dst layer to indicate that the entry has been forcefully
43       * destroyed.
44       *
45       * Negative values are used by the implementation layer code to
46       * force invocation of the dst_ops->check() method.
47       */
48      short           obsolete;
49  #define DST_OBSOLETE_NONE   0
50  #define DST_OBSOLETE_DEAD   2
51  #define DST_OBSOLETE_FORCE_CHK  -1
52  #define DST_OBSOLETE_KILL   -2
53      unsigned short      header_len; /* more space at head required */
54      unsigned short      trailer_len;    /* space to reserve at tail */
55  #ifdef CONFIG_IP_ROUTE_CLASSID
56      __u32           tclassid;
57  #else
58      __u32           __pad2;
59  #endif
60
61  #ifdef CONFIG_64BIT
62      struct lwtunnel_state   *lwtstate;
63      /*
64       * Align __refcnt to a 64 bytes alignment
65       * (L1_CACHE_SIZE would be too much)
66       */
67      long            __pad_to_align_refcnt[1];
68  #endif
69      /*
70       * __refcnt wants to be on a different cache line from
71       * input/output/ops or performance tanks badly
72       */
73      atomic_t        __refcnt;   /* client references    */
74      int         __use;
75      unsigned long       lastuse;
76  #ifndef CONFIG_64BIT
77      struct lwtunnel_state   *lwtstate;
78  #endif
79      union {
80          struct dst_entry    *next;
81          struct rtable __rcu *rt_next;
82          struct rt6_info     *rt6_next;
83          struct dn_route __rcu   *dn_next;
84      };
85  };
```

2.4.2 rtable

该结构位于/include/net/route.h中。

这是ipv4路由缓存相关结构体,保护了该路由缓存查找的匹配条件,即**struct flowi**类型的变量、目的 ip、源 ip、下一跳网关地址、路由类型等。当然了，还有最重要的，保护了一个协议无关的**dst_entry**变量，能够很好地实现**dst_entry**与 rtable 的转换，

而dst_entry中又包含邻居项相关的信息，实现了路由缓存与邻居子系统的关联。

```
struct rtable {
        /* 存储缓存路由项中独立于协议的信息 */
    struct dst_entry    dst;

    int         rt_genid;
        /* 表示路由表项的一些特性和标志 */
    unsigned int        rt_flags;
    __u16           rt_type;
    __u8            rt_is_input;
    __u8            rt_uses_gateway;

    int         rt_iif;

    /* Info on neighbour */
    __be32          rt_gateway;

    /* Miscellaneous cached information */
    u32         rt_pmtu;

    u32         rt_table_id;

    struct list_head    rt_uncached;
    struct uncached_list    *rt_uncached_list;
};
```

2.4.3 flowi

该数据结构位于/include/net/flow.h中，它是与路由查找相关的数据结构。

```
struct flowi {
    union {
        struct flowi_common __fl_common;
        struct flowi4       ip4;
        struct flowi6       ip6;
        struct flowidn      dn;
    } u;
#define flowi_oif   u.__fl_common.flowic_oif
#define flowi_iif   u.__fl_common.flowic_iif
#define flowi_mark  u.__fl_common.flowic_mark
#define flowi_tos   u.__fl_common.flowic_tos
#define flowi_scope u.__fl_common.flowic_scope
#define flowi_proto u.__fl_common.flowic_proto
#define flowi_flags u.__fl_common.flowic_flags
#define flowi_secid u.__fl_common.flowic_secid
#define flowi_tun_key   u.__fl_common.flowic_tun_key
} __attribute__((__aligned__(BITS_PER_LONG/8)));
```

2.5 TCP 层相关数据结构

2.5.1 tcphdr

该数据结构位于/include/uapi/linux/tcp.h 中。

```
    struct tcphdr {
        __be16  source;
        __be16  dest;
        __be32  seq;
        __be32  ack_seq;
    #if defined(__LITTLE_ENDIAN_BITFIELD)
        __u16   res1:4,
            doff:4,
            fin:1,
            syn:1,
            rst:1,
            psh:1,
            ack:1,
            urg:1,
            ece:1,
            cwr:1;
    #elif defined(__BIG_ENDIAN_BITFIELD)
        __u16   doff:4,
            res1:4,
            cwr:1,
            ece:1,
            urg:1,
            ack:1,
            psh:1,
            rst:1,
            syn:1,
            fin:1;
    #else
    #error  "Adjust your <asm/byteorder.h> defines"
    #endif
        __be16  window;
        __sum16 check;
        __be16  urg_ptr;
    };
```

2.5.2 tcp_options_received

该结构位于/include/linux/tcp.h中，其主要表述 TCP 头部的选项字段。

```
struct tcp_options_received {
/*  PAWS/RTTM data  */
    long    ts_recent_stamp;  /* Time we stored ts_recent (for aging)
                                 记录从接收到的段中取出时间戳设置到 ts_recent 的时间
                                 用于检测 ts_recent 的有效性：如果自从该事件之后已经
                                 经过了超过 24 天的时间，则认为 ts_recent 已无效。
                               */
    u32 ts_recent;            /* Time stamp to echo next
                                 下一个待发送的 TCP 段中的时间戳回显值。当一个含有最后
                                 发送 ACK 中确认序号的段到达时，该段中的时间戳被保存在
                                 ts_recent 中。而当下一个待发送的 TCP 段的时间戳值是由
                                 SKB 中 TCP 控制块的成员 when 填入的，when 字段值是由协议
                                 栈取系统时间变量 jiffies 的低 32 位。
                               */
    u32 rcv_tsval;            /* Time stamp value
                                 保存最近一次接收到对段的 TCP 段的时间戳选项中的时间戳值。
                               */
    u32 rcv_tsecr;            /* Time stamp echo reply
```

```
                                保存最近一次接收到对端的 TCP 段的时间戳选项中的时间戳
                                回显应答。
                               */
    u16     saw_tstamp : 1,    /* Saw TIMESTAMP on last packet
                                标识最近一次接收到的 TCP 段是否存在 TCP 时间戳选项，1 为有，
                                0 为无。
                               */
        tstamp_ok : 1,         /* TIMESTAMP seen on SYN packet
                                标识 TCP 连接是否启动时间戳选项。
                               */
        dsack : 1,             /* D-SACK is scheduled                    */
        wscale_ok : 1,        /* Wscale seen on SYN packet
                                                    标志接收方是否支持窗口扩大因子，只出现在
                                                   */
        sack_ok : 4,           /* SACK seen on SYN packet                */
        snd_wscale : 4,        /* Window scaling received from sender    */
        rcv_wscale : 4;        /* Window scaling to send to receiver     */
    u8  num_sacks;             /* Number of SACK blocks                  */
    u16 user_mss;              /* mss requested by user in ioctl         */
    u16 mss_clamp;             /* Maximal mss, negotiated at connection setup */
};
```

2.5.3 tcp_sock

该数据结构位于/include/linux/tcp.h中。

该数据结构时 TCP 协议的控制块，它在inet_connection_sock结构的基础上扩展了**滑动窗口协议、拥塞控制算法**等一些 TCP 的专有属性。

```
struct tcp_sock {
    /* inet_connection_sock has to be the first member of tcp_sock */
    struct inet_connection_sock inet_conn;
    u16 tcp_header_len; /* 需要发送的 TCP 头部的字节数       */
    u16 gso_segs;   /* Max number of segs per GSO packet    */

/*
 *  Header prediction flags
 *  0x5?10 << 16 + snd_wnd in net byte order
 */
    __be32  pred_flags;

/*
 *  RFC793 variables by their proper names. This means you can
 *  read the code and the spec side by side (and laugh ...)
 *  See RFC793 and RFC1122. The RFC writes these in capitals.
 */
    u64 bytes_received; /* RFC4898 tcpEStatsAppHCThruOctetsReceived
                * sum(delta(rcv_nxt)), or how many bytes
                * were acked.
                */
    u32 segs_in;    /* RFC4898 tcpEStatsPerfSegsIn
                * total number of segments in.
                */
    u32 rcv_nxt;    /* 下一个待接收的字节号      */
    u32 copied_seq; /* Head of yet unread data      */
    u32 rcv_wup;    /* rcv_nxt on last window update sent   */
    u32 snd_nxt;    /* 下一个待发送的序列         */
    u32 segs_out;   /* RFC4898 tcpEStatsPerfSegsOut
```

```
                * The total number of segments sent.
                */
    u64 bytes_acked;    /* RFC4898 tcpEStatsAppHCThruOctetsAcked
                * sum(delta(snd_una)), or how many bytes
                * were acked.
                */
    struct u64_stats_sync syncp; /* protects 64bit vars (cf tcp_get_info()) */

    u32 snd_una;    /* 待 ACK 的第一个字节的序号     */
    u32 snd_sml;    /* Last byte of the most recently transmitted small packet */
    u32 rcv_tstamp; /* timestamp of last received ACK (for keepalives) */
    u32 lsndtime;   /* timestamp of last sent data packet (for restart window) */
    u32 last_oow_ack_time;  /* timestamp of last out-of-window ACK */

    u32 tsoffset;   /* timestamp offset */

    struct list_head tsq_node; /* anchor in tsq_tasklet.head list */
    unsigned long   tsq_flags;

    /* Data for direct copy to user */
    struct {
        struct sk_buff_head prequeue;
        struct task_struct  *task;
        struct msghdr       *msg;
        int         memory;
        int         len;
    } ucopy;

    u32 snd_wl1;    /* Sequence for window update       */
    u32 snd_wnd;    /* 发送窗口  */
    u32 max_window; /* Maximal window ever seen from peer   */
    u32 mss_cache;  /* Cached effective mss, not including SACKS */

    u32 window_clamp;   /* Maximal window to advertise
                                                    滑动窗口最大值
                                                */
    u32 rcv_ssthresh;   /* Current window clamp         */

    /* Information of the most recently (s)acked skb */
    struct tcp_rack {
        struct skb_mstamp mstamp; /* (Re)sent time of the skb */
        u8 advanced; /* mstamp advanced since last lost marking */
        u8 reord;    /* reordering detected */
    } rack;
    u16 advmss;     /* Advertised MSS
                                                本端能接收的 MSS 上限，在建立时用来通告对方。
                                            */
    u8  unused;
    u8  nonagle     : 4,/* Disable Nagle algorithm?             */
        thin_lto    : 1,/* Use linear timeouts for thin streams */
        thin_dupack : 1,/* Fast retransmit on first dupack      */
        repair      : 1,
        frto        : 1;/* F-RTO (RFC5682) activated in CA_Loss */
    u8  repair_queue;
    u8  do_early_retrans:1,/* Enable RFC5827 early-retransmit  */
        syn_data:1, /* SYN includes data */
        syn_fastopen:1, /* SYN includes Fast Open option */
        syn_fastopen_exp:1,/* SYN includes Fast Open exp. option */
```

```
88          syn_data_acked:1,/* data in SYN is acked by SYN-ACK */
89          save_syn:1, /* Save headers of SYN packet */
90          is_cwnd_limited:1;/* forward progress limited by snd_cwnd? */
91      u32 tlp_high_seq;   /* snd_nxt at the time of TLP retransmit. */
92
93  /* RTT measurement */
94      u32 srtt_us;    /* smoothed round trip time << 3 in usecs */
95      u32 mdev_us;    /* medium deviation         */
96      u32 mdev_max_us;    /* maximal mdev for the last rtt period */
97      u32 rttvar_us;  /* smoothed mdev_max            */
98      u32 rtt_seq;    /* sequence number to update rttvar */
99      struct rtt_meas {
100         u32 rtt, ts;    /* RTT in usec and sampling time in jiffies. */
101     } rtt_min[3];
102
103     u32 packets_out;    /* Packets which are "in flight"    */
104     u32 retrans_out;    /* Retransmitted packets out
105                                                     重传并且还未得到确认的 TCP 段的数目
106                                                 */
107     u32 max_packets_out;  /* max packets_out in last window */
108     u32 max_packets_seq;  /* right edge of max_packets_out flight */
109
110     u16 urg_data;   /* Saved octet of OOB data and control flags */
111     u8  ecn_flags;  /* ECN status bits.         */
112     u8  keepalive_probes; /* num of allowed keep alive probes   */
113     u32 reordering; /* Packet reordering metric.        */
114     u32 snd_up;     /* Urgent pointer       */
115
116 /*
117  *      Options received (usually on last packet, some only on SYN packets).
118  */
119     struct tcp_options_received rx_opt;
120
121 /*
122  *  Slow start and congestion control (see also Nagle, and Karn & Partridge)
123  */
124     u32 snd_ssthresh;   /* Slow start size threshold        */
125     u32 snd_cwnd;   /* Sending congestion window        */
126     u32 snd_cwnd_cnt;   /* Linear increase counter
127                                                     自从上次调整拥塞窗口到目前为止接收到的总 ACK 段数
128                                                     如果该字段为零，则说明已经调整了拥塞窗口，且到目
129                                                     为止还没有接收到 ACK 段。调整拥塞窗口之后，每接收
130                                                     一个 ACK，ACK 段就会使 snd_cwnd_cnt 加 1。
131                                                 */
132     u32 snd_cwnd_clamp; /* Do not allow snd_cwnd to grow above this */
133     u32 snd_cwnd_used;
134     u32 snd_cwnd_stamp;         /* 记录最近一次检验拥塞窗口的时间
135                                                     在拥塞期间，接收到 ACK 后会进行
136                                                     拥塞窗口的检验。而在非拥塞期间，为了防止由于应用
137                                                     程序限制而造成拥塞窗口失效，因此在成功发送段后，
138                                                     如果有必要也会检验拥塞窗口。
139                                                 */
140     u32 prior_cwnd; /* Congestion window at start of Recovery. */
141     u32 prr_delivered;  /* Number of newly delivered packets to
142                  * receiver in Recovery. */
143     u32 prr_out;    /* Total number of pkts sent during Recovery. */
144
145     u32 rcv_wnd;    /* Current receiver window      */
```

```
146     u32 write_seq;  /* Tail(+1) of data held in tcp send buffer */
147     u32 notsent_lowat;  /* TCP_NOTSENT_LOWAT */
148     u32 pushed_seq; /* Last pushed seq, required to talk to windows */
149     u32 lost_out;   /* Lost packets         */
150     u32 sacked_out; /* SACK'd packets
151                                             启用 ACK 时，通过 SACK 的 TCP 选项标识已接收到的段的数量
152                                             不启用 SACK 时，标识接收到的重复确认的次数，该值在接收到
153                                         */
154     u32 fackets_out;    /* FACK'd packets           */
155
156     /* from STCP, retrans queue hinting */
157     struct sk_buff* lost_skb_hint;
158     struct sk_buff *retransmit_skb_hint;
159
160     /* OOO segments go in this list. Note that socket lock must be held,
161      * as we do not use sk_buff_head lock.
162      */
163     struct sk_buff_head out_of_order_queue;
164
165     /* SACKs data, these 2 need to be together (see tcp_options_write) */
166     struct tcp_sack_block duplicate_sack[1]; /* D-SACK block */
167     struct tcp_sack_block selective_acks[4]; /* The SACKS themselves*/
168
169     struct tcp_sack_block recv_sack_cache[4];
170
171     struct sk_buff *highest_sack;   /* skb just after the highest
172                      * skb with SACKed bit set
173                      * (validity guaranteed only if
174                      * sacked_out > 0)
175                      */
176
177     int     lost_cnt_hint;
178     u32     retransmit_high;    /* L-bits may be on up to this seqno */
179
180     u32 prior_ssthresh; /* ssthresh saved at recovery start */
181     u32 high_seq;   /* snd_nxt at onset of congestion   */
182
183     u32 retrans_stamp;  /* Timestamp of the last retransmit,
184                  * also used in SYN-SENT to remember stamp of
185                  * the first SYN. */
186     u32 undo_marker;    /* snd_una upon a new recovery episode.
187                                                 在使用 F-RTO 算法进行发送超时处理，或进入 Recove
188                                                 或进入 Loss 开始慢启动时，记录当时 SND.UNA，标记
189                                                 它是检测是否可以进行拥塞控制撤销的条件之一，一般
190                                                 拥塞撤销操作或进入拥塞控制 Loss 状态后会清零。
191                                             */
192     int undo_retrans;   /* number of undoable retransmissions.
193                                                 在恢复拥塞控制之前可进行撤销的重传段数。在进入 F
194                                                 拥塞状态 Loss 时，清零，在重传时计数，是检测是否
195                                                 撤销的条件之一。
196                                             */
197     u32 total_retrans;  /* Total retransmits for entire connection */
198
199     u32 urg_seq;    /* Seq of received urgent pointer */
200     unsigned int        keepalive_time;   /* time before keep alive takes place */
201     unsigned int        keepalive_intvl;  /* time interval between keep alive probes */
202
203     int         linger2;
204
```

```
205 /* Receiver side RTT estimation */
206     struct {
207         u32 rtt;
208         u32 seq;
209         u32 time;
210     } rcv_rtt_est;
211 
212 /* Receiver queue space */
213     struct {
214         int space;
215         u32 seq;
216         u32 time;
217     } rcvq_space;
218 
219 /* TCP-specific MTU probe information. */
220     struct {
221         u32       probe_seq_start;
222         u32       probe_seq_end;
223     } mtu_probe;
224     u32 mtu_info; /* We received an ICMP_FRAG_NEEDED / ICMPV6_PKT_TOOBIG
225                    * while socket was owned by user.
226                    */
227 
228 #ifdef CONFIG_TCP_MD5SIG
229 /* TCP AF-Specific parts; only used by MD5 Signature support so far */
230     const struct tcp_sock_af_ops    *af_specific;
231 
232 /* TCP MD5 Signature Option information */
233     struct tcp_md5sig_info  __rcu *md5sig_info;
234 #endif
235 
236 /* TCP fastopen related information */
237     struct tcp_fastopen_request *fastopen_req;
238     /* fastopen_rsk points to request_sock that resulted in this big
239      * socket. Used to retransmit SYNACKs etc.
240      */
241     struct request_sock *fastopen_rsk;
242     u32 *saved_syn;
243 };
```

2.5.4 tcp_request_sock

```
1  struct tcp_request_sock {
2      struct inet_request_sock    req;
3      const struct tcp_request_sock_ops *af_specific;
4      struct skb_mstamp       snt_synack;             /* first SYNACK sent time */
5      bool                tfo_listener;
6      u32             txhash;
7      u32             rcv_isn;
8      u32             snt_isn;
9      u32             last_oow_ack_time;              /* last SYNACK */
10     u32             rcv_nxt;                        /* the ack # by SYNACK. For
11                                                         * FastOpen it's the seq#
12                                                         * after data-in-SYN.
13                                                     */
14 };
```

2.5.5 tcp_skb_cb

在2.2.4中，我们分析过cb。在这一节中，我们将看到 TCP 层具体是如何使用这个控制缓冲区 (Control Buffer) 的。

2.5.5.1 TCP_SKB_CB

该宏用于访问给定的sk_buff的控制缓冲区的变量。在后续的章节中，可以在很多函数中看到它的身影。该宏的定义如下：

```
#define TCP_SKB_CB(__skb)       ((struct tcp_skb_cb *)&((__skb)->cb[0]))
```

可以看到，该宏实际上是将cb的指针强制转型成tcp_skb_cb 结构体的指针。也就是说，TCP 对于控制缓冲区的使用，可以从tcp_skb_cb 的定义分析出来。

2.5.5.2 tcp_skb_cb 结构体

tcp_skb_cb结构体用于将每个 TCP 包中的控制信息传递给发送封包的代码。该结构体的定义如下：

```
struct tcp_skb_cb {
        __u32           seq;            /* 起始序号      */
        __u32           end_seq;        /* SEQ + FIN + SYN + datalen    */
        union {
                /* Note : tcp_tw_isn is used in input path only
                 *        (isn chosen by tcp_timewait_state_process())
                 *
                 *        tcp_gso_segs/size are used in write queue only,
                 *        cf tcp_skb_pcount()/tcp_skb_mss()
                 */
                __u32           tcp_tw_isn;
                struct {
                        u16     tcp_gso_segs;
                        u16     tcp_gso_size;
                };
        };
        __u8            tcp_flags;      /* TCP 头部的标志位 */

        __u8            sacked;         /* SACK/FACK 标志位          .    */
```

紧接着，定义了一些宏作为标志

```
#define TCPCB_SACKED_ACKED      0x01    /* SKB 被确认了                 */
#define TCPCB_SACKED_RETRANS    0x02    /* SKB 被重传了                 */
#define TCPCB_LOST              0x04    /* SKB 已丢失                   */
#define TCPCB_TAGBITS           0x07    /* 标志位掩码                   */
#define TCPCB_REPAIRED          0x10    /* SKB 被修复了 (no skb_mstamp)  */
#define TCPCB_EVER_RETRANS      0x80    /* SKB 曾经被重传过               */
#define TCPCB_RETRANS           (TCPCB_SACKED_RETRANS|TCPCB_EVER_RETRANS| \
                                TCPCB_REPAIRED)
```

接下来又继续定义 TCP 相关的位。

```
        __u8            ip_dsfield;     /* IPv4 tos or IPv6 dsfield     */
        /* 1 byte hole */
        __u32           ack_seq;        /* ACK 的序号                    */
        union {
                struct inet_skb_parm    h4;
#if IS_ENABLED(CONFIG_IPV6)
                struct inet6_skb_parm   h6;
#endif
        } header;       /* For incoming frames          */
};
```

CHAPTER 3

TCP 建立连接过程

Contents

3.1 TCP 主动打开-客户

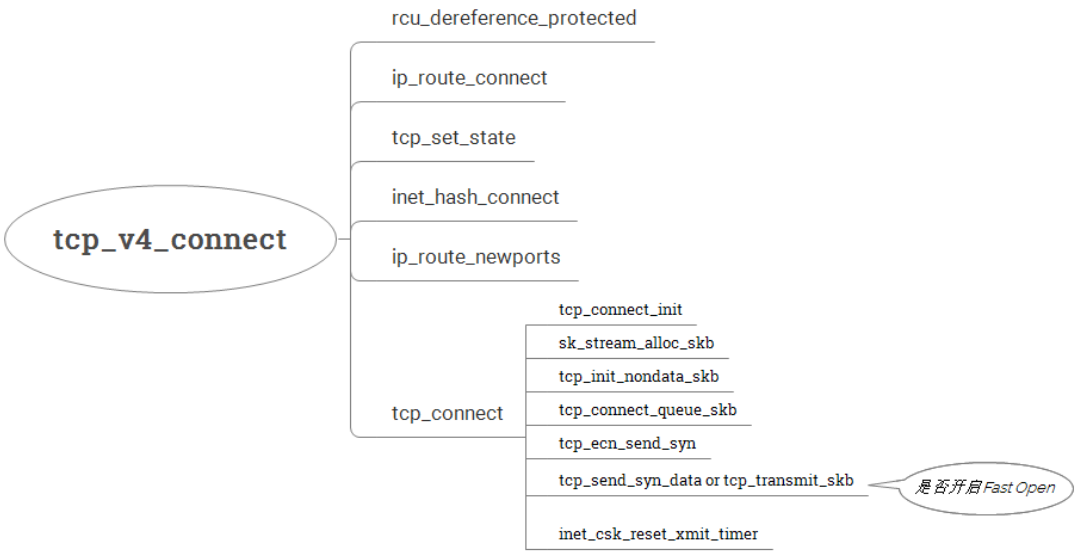
3.1.1 基本流程

客户端的主动打开是通过 connect 系统调用等一系列操作来完成的，这一系统调用最终会调用传输层的`tcp_v4_connect`函数。基本就是客户端发送 SYN 段，接收 SYN+ACK 段，最后再次发送 ACK 段给服务器。

3.1.2 第一次握手：构造并发送 SYN 包

3.1.2.1 基本调用关系

SYN.png SYN.bb



3.1.2.2 tcp_v4_connect

tcp_v4_connect的主要作用是进行一系列的判断，初始化传输控制块并调用相关函数发送 SYN 包。

```
/*
Location:

        net/ipv4/tcp_ipv4.c

Function:

        这个函数会初始化一个发送用的连接。

Parameters:

        sk: 传输控制块
        uaddr: 通用地址结构，包含所属协议字段和相应的地址字段。
        addrlen        :
*/
int tcp_v4_connect(struct sock *sk, struct sockaddr *uaddr, int addr_len)
{
                /*
                        sockaddr_in 结构体用于描述一个 Internet (IP) 套接字的地址
                */
        struct sockaddr_in *usin = (struct sockaddr_in *)uaddr;
        struct inet_sock *inet = inet_sk(sk);
        struct tcp_sock *tp = tcp_sk(sk);
        __be16 orig_sport, orig_dport;
        __be32 daddr, nexthop;                                          //网络协议主要使用大端存储，be32
        struct flowi4 *fl4;
        struct rtable *rt;
        int err;
        struct ip_options_rcu *inet_opt;

        /* 检验目的地址长度是否有效 */
        if (addr_len < sizeof(struct sockaddr_in))
                return -EINVAL;                                          //Invalid argument 错误码为 22

        /* 检验协议族是否正确 */
        if (usin->sin_family != AF_INET)                //IPV4 地址域
                return -EAFNOSUPPORT;                       //Address family not supported by protoco

        /* 将下一跳地址和目的地址暂时设置为用户传入的 IP 地址 */
        nexthop = daddr = usin->sin_addr.s_addr;
        inet_opt = rcu_dereference_protected(inet->inet_opt,
                                             sock_owned_by_user(sk));
        /* 如果选择源地址路由，则将下一跳地址设置为 IP 选项中的 faddr-first hop address*/
        if (inet_opt && inet_opt->opt.srr) {
                if (!daddr)
                        return -EINVAL;
                nexthop = inet_opt->opt.faddr;
        }
```

源地址路由是一种特殊的路由策略。一般路由都是通过目的地址来进行的。而有时也需要通过源地址来进行路由，例如在有多个网卡等情况下，可以根据源地址来决定走哪个网卡等等。

```
        orig_sport = inet->inet_sport;
        orig_dport = usin->sin_port;
        fl4 = &inet->cork.fl.u.ip4;                    //对应于 ipv4 的流
        /* 获取目标的路由缓存项，如果路由查找命中，则生成一个相应的路由缓存项，这个缓存项不但可以用于
                        当前待发送的 SYN 段，而且对后续的所有数据包都可以起到一个加速路由查找的作用。 */
        rt = ip_route_connect(fl4, nexthop, inet->inet_saddr,
                              RT_CONN_FLAGS(sk), sk->sk_bound_dev_if,
                              IPPROTO_TCP,
                              orig_sport, orig_dport, sk);
                /* 判断指针是否有效 */
                if (IS_ERR(rt)) {
                err = PTR_ERR(rt);
                if (err == -ENETUNREACH)        //Network is unreachable
                        IP_INC_STATS(sock_net(sk), IPSTATS_MIB_OUTNOROUTES);
                return err;
        }

                /*TCP 不能使用类型为组播或多播的路由缓存项 */
        if (rt->rt_flags & (RTCF_MULTICAST | RTCF_BROADCAST)) {
                ip_rt_put(rt);
                return -ENETUNREACH;                //Network is unreachable
        }

        /* 如果没有开启源路由功能，则采用查找到的缓存项 */
        if (!inet_opt || !inet_opt->opt.srr)
                daddr = fl4->daddr;

        /* 如果没有设置源地址，则设置为缓存项中的源地址 */
        if (!inet->inet_saddr)
                inet->inet_saddr = fl4->saddr;
        sk_rcv_saddr_set(sk, inet->inet_saddr);

        /* 如果该传输控制块的时间戳已被使用过，则重置各状态
                        rx_opt:  tcp_options_received
                */
        if (tp->rx_opt.ts_recent_stamp && inet->inet_daddr != daddr) {
                /* Reset inherited state */
                                /* 下一个待发送的 TCP 段中的时间戳回显值 */
                tp->rx_opt.ts_recent       = 0;
                                /* 从接收到的段中取出时间戳 */
                tp->rx_opt.ts_recent_stamp = 0;
                                /*What does it means repair*/
                if (likely(!tp->repair))
                        tp->write_seq      = 0;
        }

        /*          在启用了 tw_recycle:time wait recycle 的情况下，重设时间戳
                        它用来快速回收 TIME_WAIT 连接.
                */
        if (tcp_death_row.sysctl_tw_recycle &&
            !tp->rx_opt.ts_recent_stamp && fl4->daddr == daddr)
                tcp_fetch_timewait_stamp(sk, &rt->dst);

        /* 设置传输控制块 */
        inet->inet_dport = usin->sin_port;
        sk_daddr_set(sk, daddr);

        inet_csk(sk)->icsk_ext_hdr_len = 0;
```

```
59          if (inet_opt)
60                  inet_csk(sk)->icsk_ext_hdr_len = inet_opt->opt.optlen;
61
62          /* 设置 MSS 大小 */
63          tp->rx_opt.mss_clamp = TCP_MSS_DEFAULT;
64
65          /* Socket identity is still unknown (sport may be zero).
66           * However we set state to SYN-SENT and not releasing socket
67           * lock select source port, enter ourselves into the hash tables and
68           * complete initialization after this.
69           */
70          /* 将 TCP 的状态设置为 SYN_SENT */
71          tcp_set_state(sk, TCP_SYN_SENT);
72          err = inet_hash_connect(&tcp_death_row, sk);
73          if (err)
74                  goto failure;
75
76          sk_set_txhash(sk);
77
78                  /* 如果源端口或者目的端口发生改变，则需要重新查找路由，并用新的路由缓存项更新 sk 中保存的路
79          rt = ip_route_newports(fl4, rt, orig_sport, orig_dport,
80                                 inet->inet_sport, inet->inet_dport, sk);
81          if (IS_ERR(rt)) {
82                  err = PTR_ERR(rt);
83                  rt = NULL;
84                  goto failure;
85          }
86          /* 将目的地址提交到套接字  */
87          sk->sk_gso_type = SKB_GSO_TCPV4;
88          sk_setup_caps(sk, &rt->dst);
89
90          /*          如果没有设置序号，则计算初始序号
91                           序号与双方的地址与端口号有关系
92                  */
93          if (!tp->write_seq && likely(!tp->repair))
94                  tp->write_seq = secure_tcp_sequence_number(inet->inet_saddr,
95                                                             inet->inet_daddr,
96                                                             inet->inet_sport,
97                                                             usin->sin_port);
98
99          /*          计算 IP 首部的 id 域的值
100                          全局变量 jiffies 用来记录自系统启动以来产生的节拍的总数
101                 */
102         inet->inet_id = tp->write_seq ^ jiffies;
103
104         /* 调用 tcp_connect 构造并发送 SYN 包 */
105         err = tcp_connect(sk);
106
107         rt = NULL;
108         if (err)
109                 goto failure;
110
111         return 0;
```

总结起来，tcp_v4_connect是在根据用户提供的目的地址，设置好了传输控制块，为传输做好准备。如果在这一过程中出现错误，则会跳到错误处理代码。

```
failure:
        /* 将状态设定为 TCP_CLOSE, 释放端口，并返回错误值。
         */
        tcp_set_state(sk, TCP_CLOSE);
        ip_rt_put(rt);
        sk->sk_route_caps = 0;
        inet->inet_dport = 0;
        return err;
```

3.1.2.3 tcp_connect

上面的tcp_v4_connect会进行一系列的判断，之后真正构造 SYN 包的部分被放置在了tcp_connect中。接下来，我们来分析这个函数。

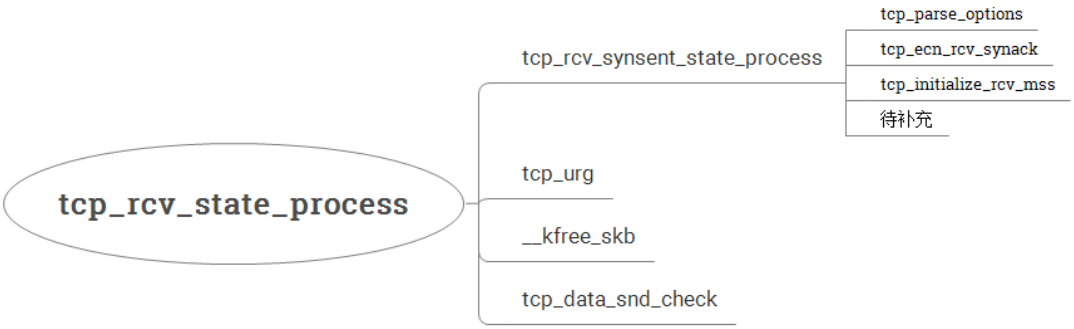
```
/*
Location:

        net/ipv4/tcp_output.c

Function:

        该函数用于构造并发送 SYN 包。

Parameter:

        sk: 传输控制块。

 */
int tcp_connect(struct sock *sk)
{
        struct tcp_sock *tp = tcp_sk(sk);
        struct sk_buff *buff;
        int err;

        /* 初始化 tcp 连接 */
        tcp_connect_init(sk);

        if (unlikely(tp->repair)) {                              //what does it mean repair?
                /* 如果 repair 位被置 1, 那么结束 TCP 连接 */
                tcp_finish_connect(sk, NULL);
                return 0;
        }

        /* 分配一个 sk_buff */
        buff = sk_stream_alloc_skb(sk, 0, sk->sk_allocation, true);
        if (unlikely(!buff))
                return -ENOBUFS;                                 //No buffer space available

        /* 初始化 skb, 并自增 write_seq 的值 */
        tcp_init_nondata_skb(buff, tp->write_seq++, TCPHDR_SYN);
        /* 设置时间戳 */
        tp->retrans_stamp = tcp_time_stamp;
        /* 将当前的 sk_buff 添加到发送队列中 */
        tcp_connect_queue_skb(sk, buff);
        /* ECN (Explicit Congestion Notification,) 支持
                                显示拥塞控制
                */
```

```
44          tcp_ecn_send_syn(sk, buff);                         //what does this function do?
45
46          /* 发送 SYN 包，这里同时还考虑了 Fast Open 的情况 */
47          err = tp->fastopen_req ? tcp_send_syn_data(sk, buff) :
48                tcp_transmit_skb(sk, buff, 1, sk->sk_allocation);
49          if (err == -ECONNREFUSED)                           /*Connection refused*/
50                  return err;
51
52          /* We change tp->snd_nxt after the tcp_transmit_skb() call
53           * in order to make this packet get counted in tcpOutSegs.
54           */
55          tp->snd_nxt = tp->write_seq;                //send next 下一个待发送的序列
56                  /*pushed_seq means Last pushed seq, required to talk to windows???*/
57                  tp->pushed_seq = tp->write_seq;
58          TCP_INC_STATS(sock_net(sk), TCP_MIB_ACTIVEOPENS);
59
60          /* 设定超时重传定时器 */
61          inet_csk_reset_xmit_timer(sk, ICSK_TIME_RETRANS,
62                                    inet_csk(sk)->icsk_rto, TCP_RTO_MAX);
63          return 0;
64  }
```

3.1.3 第二次握手：接收 SYN+ACK 包

3.1.3.1 基本调用关系

SYN +ACK.png SYN +ACK.bb



3.1.3.2 tcp_rcv_state_process

tcp_rcv_state_process实现了 TCP 状态机相对较为核心的一个部分。该函数可以处理除 ESTABLISHED 和 TIME_WAIT 状态以外的情况下的接收过程。这里，我们仅关心主动连接情况下的处理。在3.1.2.2中，我们分析源码时得出，客户端发送 SYN 包后，会将状态机设置为TCP_SYN_SENT状态。因此，我们仅在这里分析该状态下的代码。

```
1   /*
2   Location:
3
4           net/ipv4/tcp_input.c
5
6   Function:
7
```

```
        This function implements the receiving procedure of RFC 793 for
        all states except ESTABLISHED and TIME_WAIT.
        It's called from both tcp_v4_rcv and tcp_v6_rcv and should be
        address independent.

Paramater:

        sk: 传输控制块。
        skb: 缓存块。
*/
int tcp_rcv_state_process(struct sock *sk, struct sk_buff *skb)
{
        struct tcp_sock *tp = tcp_sk(sk);
        struct inet_connection_sock *icsk = inet_csk(sk);
        const struct tcphdr *th = tcp_hdr(skb);
        struct request_sock *req;
        int queued = 0;
        bool acceptable;

                /* 标识接收到的 TCP 段不存在 TCP 时间戳选项 */
        tp->rx_opt.saw_tstamp = 0;

        switch (sk->sk_state) {
                case TCP_CLOSE:
                        /* CLOSE 状态的处理代码 */

                case TCP_LISTEN:
                        /* LISTEN 状态的处理代码 */

                case TCP_SYN_SENT:
                        /* 处理接收到的数据段 */
                        queued = tcp_rcv_synsent_state_process(sk, skb, th);
                        if (queued >= 0)
                                return queued;

                        /* 处理紧急数据并检测是否有数据需要发送 */
                        tcp_urg(sk, skb, th);
                        __kfree_skb(skb);
                        tcp_data_snd_check(sk);
                        return 0;
                }

        /* 处理其他情况的代码 */
}
```

3.1.3.3 tcp_rcv_synsent_state_process

具体的处理代码被放在了tcp_rcv_synsent_state_process中，通过命名就可以看出，该函数是专门用于处理 SYN_SENT 状态下收到的数据的。

```
/*
Location:

        net/ipv4/tcp_input.c

Function:

        处理 syn 已经发送的状态。
```

```
9
10  Parameter:
11
12          sk: 传输控制块
13          skb: 缓存
14          th:tcp 报文的头部
15
16  */
17  static int tcp_rcv_synsent_state_process(struct sock *sk, struct sk_buff *skb,
18                                           const struct tcphdr *th)
19  {
20          struct inet_connection_sock *icsk = inet_csk(sk);
21          struct tcp_sock *tp = tcp_sk(sk);
22          struct tcp_fastopen_cookie foc = { .len = -1 };
23          int saved_clamp = tp->rx_opt.mss_clamp;
24
25          /* 解析 TCP 选项，并保存在传输控制块中 */
26          tcp_parse_options(skb, &tp->rx_opt, 0, &foc);
27                  /*
28                          rcv_tsecr 保存最近一次接收到对端的 TCP 段的时间
29                          戳选项中的时间戳回显应答。???
30
31                          tsoffset means timestamp offset
32                  */
33          if (tp->rx_opt.saw_tstamp && tp->rx_opt.rcv_tsecr)
34                  tp->rx_opt.rcv_tsecr -= tp->tsoffset;
```

接下来的部分，就是按照 TCP 协议的标准来实现相应的行为。注释中出现的 RFC793 即是描述 TCP 协议的 RFC 原文中的文本。

```
1           if (th->ack) {
2                   /* rfc793:
3                    * "If the state is SYN-SENT then
4                    *    first check the ACK bit
5                    *      If the ACK bit is set
6                    *        If SEG.ACK =< ISS, or SEG.ACK > SND.NXT, send
7                    *        a reset (unless the RST bit is set, if so drop
8                    *        the segment and return)"
9                    * ISS 代表初始发送序号 (Initial Send Sequence number)
10                                   snd_una 在输出的段中，最早一个未被确认段的序号
11                   */
12                  if (!after(TCP_SKB_CB(skb)->ack_seq, tp->snd_una) ||
13                      after(TCP_SKB_CB(skb)->ack_seq, tp->snd_nxt))
14                          goto reset_and_undo;
15
16                  if (tp->rx_opt.saw_tstamp && tp->rx_opt.rcv_tsecr &&
17                      !between(tp->rx_opt.rcv_tsecr, tp->retrans_stamp,
18                               tcp_time_stamp)) {
19                          NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_PAWSACTIVEREJECTED);
20                          goto reset_and_undo;
21                  }
```

上面的一段根据 RFC 在判断 ACK 的值是否在初始发送序号和下一个序号之间，如果不在，则发送一个重置。

```
                /* 此时, ACK 已经被接受了
                 *
                 * "If the RST bit is set
                 *    If the ACK was acceptable then signal the user "error:
                 *    connection reset", drop the segment, enter CLOSED state,
                 *    delete TCB, and return."
                 */

                if (th->rst) {
                        tcp_reset(sk);
                        goto discard;
                }
```

接下来, 判断了收到的包的 RST 位, 如果设置了 RST, 则丢弃该分组, 并进入 CLOSED 状态。

```
                /* rfc793:
                 *   "fifth, if neither of the SYN or RST bits is set then
                 *    drop the segment and return."
                 *
                 *    See note below!
                 *                                        --ANK(990513)
                 */
                if (!th->syn)
                        goto discard_and_undo;
```

之后，根据 RFC 的说法，如果既没有设置 SYN 位，也没有设置 RST 位，那么就将分组丢弃掉。前面已经判断了 RST 位了，因此，这里判断一下 SYN 位。

接下来就准备进入到 ESTABLISHED 状态了。

```
                /* rfc793:
                 *   "If the SYN bit is on ...
                 *    are acceptable then ...
                 *    (our SYN has been ACKed), change the connection
                 *    state to ESTABLISHED..."
                 */

                tcp_ecn_rcv_synack(tp, th);

                /* 初始化与窗口有关的参数 */
                tcp_init_wl(tp, TCP_SKB_CB(skb)->seq);
                tcp_ack(sk, skb, FLAG_SLOWPATH);

                /* Ok.. it's good. Set up sequence numbers and
                 * move to established.
                 */
                tp->rcv_nxt = TCP_SKB_CB(skb)->seq + 1;
                tp->rcv_wup = TCP_SKB_CB(skb)->seq + 1;
```

对于 SYN 和 SYN/ACK 段是不进行窗口放大的。关于 RFC1323 窗口放大相关的内容, 我们在 1.3.2中进行了详细讨论。接下来手动设定了窗口缩放相关的参数, 使得缩放不生效。

```
                /* RFC1323: The window in SYN & SYN/ACK segments is
                 * never scaled.
                 */
                tp->snd_wnd = ntohs(th->window);        //network to host short int

                                /* wscale_ok 标志接收方是否支持窗口扩大因子
                                   window_clamp 滑动窗口最大值
                                */
                                if (!tp->rx_opt.wscale_ok) {
                        tp->rx_opt.snd_wscale = tp->rx_opt.rcv_wscale = 0;
                        tp->window_clamp = min(tp->window_clamp, 65535U);
                }
```

根据时间戳选项，设定相关字段及 TCP 头部长度。

```
                if (tp->rx_opt.saw_tstamp) {
                        tp->rx_opt.tstamp_ok= 1;
                        tp->tcp_header_len =
                                sizeof(struct tcphdr) + TCPOLEN_TSTAMP_ALIGNED;
                        tp->advmss -= TCPOLEN_TSTAMP_ALIGNED;
                        tcp_store_ts_recent(tp);
                } else {
                        tp->tcp_header_len = sizeof(struct tcphdr);
                }
```

之后会根据设定开启 FACK 机制。FACK 是在 SACK 机制上发展来的。SACK 用于准确地获知哪些包丢失了需要重传。开启 SACK 后，可以让发送端只重传丢失的包。而当重传的包比较多时，会进一步导致网络繁忙，FASK 用来做重传过程中的拥塞控制。

```
                if (tcp_is_sack(tp) && sysctl_tcp_fack)        //sysctl: system control
                        tcp_enable_fack(tp);
```

最后初始化 MTU、MSS 等参数并完成 TCP 连接过程。

```
                tcp_mtup_init(sk);
                tcp_sync_mss(sk, icsk->icsk_pmtu_cookie);
                tcp_initialize_rcv_mss(sk);

                /* Remember, tcp_poll() does not lock socket!
                 * Change state from SYN-SENT only after copied_seq
                 * is initialized. */
                tp->copied_seq = tp->rcv_nxt;

                smp_mb();
                tcp_finish_connect(sk, skb);
```

此后，开始处理一些特殊情况。Fast Open 启用的情况下，SYN 包也会带有数据。这里调用 `tcp_rcv_fastopen_synack`函数处理 SYN 包附带的数据。

```
                if ((tp->syn_fastopen || tp->syn_data) &&
                    tcp_rcv_fastopen_synack(sk, skb, &foc))
                        return -1;

                /* 根据情况进入延迟确认模式 ??? Confused*/
                if (sk->sk_write_pending ||
                    icsk->icsk_accept_queue.rskq_defer_accept ||
                    icsk->icsk_ack.pingpong) {
                        /* Save one ACK. Data will be ready after
```

```
                         * several ticks, if write_pending is set.
                         *
                         * It may be deleted, but with this feature tcpdumps
                         * look so _wonderfully_ clever, that I was not able
                         * to stand against the temptation 8)     --ANK
                         */
                        inet_csk_schedule_ack(sk);
                        icsk->icsk_ack.lrcvtime = tcp_time_stamp;
                        tcp_enter_quickack_mode(sk);
                        inet_csk_reset_xmit_timer(sk, ICSK_TIME_DACK,
                                                  TCP_DELACK_MAX, TCP_RTO_MAX);

discard:
                        __kfree_skb(skb);
                        return 0;
                } else {
                        /* 回复 ACK 包 */
                        tcp_send_ack(sk);
                }
                return -1;
        }
```

最后是一些异常情况的处理。

```
        /* 进入该分支意味着包中不包含 ACK */

        if (th->rst) {
                /* rfc793:
                 * "If the RST bit is set
                 *
                 *      Otherwise (no ACK) drop the segment and return."
                 * 如果收到了 RST 包，则直接丢弃并返回。
                 */

                goto discard_and_undo;
        }

        /* PAWS 检查 */
        if (tp->rx_opt.ts_recent_stamp && tp->rx_opt.saw_tstamp &&
            tcp_paws_reject(&tp->rx_opt, 0))
                goto discard_and_undo;

        /* 仅有 SYN 而无 ACK 的处理 */
        if (th->syn) {
                /* We see SYN without ACK. It is attempt of
                 * simultaneous connect with crossed SYNs.
                 * Particularly, it can be connect to self.
                 */
                tcp_set_state(sk, TCP_SYN_RECV);

                /* 下面的处理和前面几乎一样 */
                if (tp->rx_opt.saw_tstamp) {
                        tp->rx_opt.tstamp_ok = 1;
                        tcp_store_ts_recent(tp);
                        tp->tcp_header_len =
                                sizeof(struct tcphdr) + TCPOLEN_TSTAMP_ALIGNED;
                } else {
                        tp->tcp_header_len = sizeof(struct tcphdr);
                }
```

```
36
37                  tp->rcv_nxt = TCP_SKB_CB(skb)->seq + 1;
38                  tp->copied_seq = tp->rcv_nxt;
39                  tp->rcv_wup = TCP_SKB_CB(skb)->seq + 1;
40
41                  /* RFC1323: The window in SYN & SYN/ACK segments is
42                   * never scaled.
43                   */
44                  tp->snd_wnd    = ntohs(th->window);
45                  tp->snd_wl1    = TCP_SKB_CB(skb)->seq;
46                  tp->max_window = tp->snd_wnd;
47
48                  tcp_ecn_rcv_syn(tp, th);
49
50                  tcp_mtup_init(sk);
51                  tcp_sync_mss(sk, icsk->icsk_pmtu_cookie);
52                  tcp_initialize_rcv_mss(sk);
53
54                  tcp_send_synack(sk);
55  #if 0
56                  /* Note, we could accept data and URG from this segment.
57                   * There are no obstacles to make this (except that we must
58                   * either change tcp_recvmsg() to prevent it from returning data
59                   * before 3WHS completes per RFC793, or employ TCP Fast Open).
60                   *
61                   * However, if we ignore data in ACKless segments sometimes,
62                   * we have no reasons to accept it sometimes.
63                   * Also, seems the code doing it in step6 of tcp_rcv_state_process
64                   * is not flawless. So, discard packet for sanity.
65                   * Uncomment this return to process the data.
66                   */
67                  return -1;
68  #else
69                  goto discard;
70  #endif
71          }
72          /* "fifth, if neither of the SYN or RST bits is set then
73           * drop the segment and return."
74           */
75
76  discard_and_undo:
77          tcp_clear_options(&tp->rx_opt);
78          tp->rx_opt.mss_clamp = saved_clamp;
79          goto discard;
80
81  reset_and_undo:
82          tcp_clear_options(&tp->rx_opt);
83          tp->rx_opt.mss_clamp = saved_clamp;
84          return 1;
85  }
```

ACK.png ACK.bb

3.1.4 第三次握手——发送 ACK 包

3.1.4.1 基本调用关系

3.1.4.2 tcp_send_ack

在3.1.3分析的代码的最后，我们看到它调用了`tcp_send_ack()`来发送 ACK 包，从而实现第三次握手。

```
/*
Location

        net/ipv4/tcp_output.c

Function:

        This routine sends an ack and also updates the window.

Parameter:

        sk: 传输控制块
该函数用于发送 ACK，并更新窗口的大小 */
void tcp_send_ack(struct sock *sk)
{
        struct sk_buff *buff;

        /* 如果当前的套接字已经被关闭了，那么直接返回。 */
        if (sk->sk_state == TCP_CLOSE)
                return;
                /*what does it using for?*/
```

```
22          tcp_ca_event(sk, CA_EVENT_NON_DELAYED_ACK);
23
24          /* We are not putting this on the write queue, so
25           * tcp_transmit_skb() will set the ownership to this
26           * sock.
27           * 为数据包分配空间
28           */
29          buff = alloc_skb(MAX_TCP_HEADER, sk_gfp_atomic(sk, GFP_ATOMIC));
30          if (!buff) {
31                  inet_csk_schedule_ack(sk);
32                  inet_csk(sk)->icsk_ack.ato = TCP_ATO_MIN;
33                  inet_csk_reset_xmit_timer(sk, ICSK_TIME_DACK,
34                                            TCP_DELACK_MAX, TCP_RTO_MAX);
35                  return;
36          }
37
38          /* 初始化 ACK 包 */
39          skb_reserve(buff, MAX_TCP_HEADER);
40          tcp_init_nondata_skb(buff, tcp_acceptable_seq(sk), TCPHDR_ACK);
41
42          /* We do not want pure acks influencing TCP Small Queues or fq/pacing
43           * too much.
44           * SKB_TRUESIZE(max(1 .. 66, MAX_TCP_HEADER)) is unfortunately ~784
45           * We also avoid tcp_wfree() overhead (cache line miss accessing
46           * tp->tsq_flags) by using regular sock_wfree()
47           */
48          skb_set_tcp_pure_ack(buff);
49
50          /* 添加时间戳并发送 ACK 包 */
51          skb_mstamp_get(&buff->skb_mstamp);
52          tcp_transmit_skb(sk, buff, 0, sk_gfp_atomic(sk, GFP_ATOMIC));
53  }
```

3.1.5 tcp_transmit_skb

```
1   /* 为 SYN 包计算 TCP 选项，这个函数中计算出来的还不是最终的格式。
2    */
3   static unsigned int tcp_syn_options(struct sock *sk, struct sk_buff *skb,
4                                       struct tcp_out_options *opts,
5                                       struct tcp_md5sig_key **md5);
6   /* 为已经建立连接的套接字计算 TCP 选项，这个函数中计算出来的还不是最终的格式。
7    */
8   static unsigned int tcp_established_options(struct sock *sk, struct sk_buff *skb,
9                                               struct tcp_out_options *opts,
10                                              struct tcp_md5sig_key **md5);
11  /* 在 skb 中为头部流出空间。
12   */
13  skb_push(skb, tcp_header_size);
14  /* 判断 skb 是否为一个纯 ACK。这里把实现也放出来了。可以看到，纯 ACK 的包最显著
15   * 的特点是其长度。Linux 里通过判断长度直接快速判断出 skb 是否为一个纯 ACK。
16   */
17  static inline bool skb_is_tcp_pure_ack(const struct sk_buff *skb)
18  {
19          return skb->truesize == 2;
20  }
21  /* 重置传输层的 header 的指针？
22   */
23  static inline void skb_reset_transport_header(struct sk_buff *skb)
```

```
{
        skb->transport_header = skb->data - skb->head;
}
/* 选择发送窗口的大小
 */
tcp_select_window(sk);
tcp_urg_mode(tp);
before(tcb->seq, tp->snd_up);
tcp_options_write((__be32 *)(th + 1), tp, &opts);
tcp_event_ack_sent(sk, tcp_skb_pcount(skb));
tcp_event_data_sent(tp, sk);
queue_xmit();
tcp_enter_cwr(sk);
net_xmit_eval(err);
```

3.1.6 tcp_select_window(struct sk_buff *skb)

这个函数的作用是选择一个新的窗口大小以用于更新tcp_sock。返回的结果根据RFC1323（详见1.3.2）进行了缩放。

3.1.6.1 代码分析

```
static u16 tcp_select_window(struct sock *sk)
{
        struct tcp_sock *tp = tcp_sk(sk);
        u32 old_win = tp->rcv_wnd;
        u32 cur_win = tcp_receive_window(tp);
        u32 new_win = __tcp_select_window(sk);
        /* old_win 是接收方窗口的大小。
         * cur_win 当前的接收窗口大小。
         * new_win 是新选择出来的窗口大小。
         */

        /* 当新窗口的大小小于当前窗口的大小时，不能缩减窗口大小。
         * 这是 IEEE 强烈不建议的一种行为。
         */
        if (new_win < cur_win) {
                /* Danger Will Robinson!
                 * Don't update rcv_wup/rcv_wnd here or else
                 * we will not be able to advertise a zero
                 * window in time.  --DaveM
                 *
                 * Relax Will Robinson.
                 */
                if (new_win == 0)
                        NET_INC_STATS(sock_net(sk),
                                      LINUX_MIB_TCPWANTZEROWINDOWADV);
                /* 当计算出来的新窗口小于当前窗口时，将新窗口设置为大于 cur_win
                 * 的 1<<tp->rx_opt.rcv_wscale 的整数倍。
                 */
                new_win = ALIGN(cur_win, 1 << tp->rx_opt.rcv_wscale);
        }
        /* 将当前的接收窗口设置为新的窗口大小。*/
        tp->rcv_wnd = new_win;
        tp->rcv_wup = tp->rcv_nxt;

        /* 判断当前窗口未越界。*/
```

```
36          if (!tp->rx_opt.rcv_wscale && sysctl_tcp_workaround_signed_windows)
37                  new_win = min(new_win, MAX_TCP_WINDOW);
38          else
39                  new_win = min(new_win, (65535U << tp->rx_opt.rcv_wscale));
40
41          /* RFC1323 缩放窗口大小。这里之所以是右移，是因为此时的 new_win 是
42           * 窗口的真正大小。所以返回时需要返回正常的可以放在 16 位整型中的窗口大小。
43           * 所以需要右移。
44           */
45          new_win >>= tp->rx_opt.rcv_wscale;
46
47          /* If we advertise zero window, disable fast path. */
48          if (new_win == 0) {
49                  tp->pred_flags = 0;
50                  if (old_win)
51                          NET_INC_STATS(sock_net(sk),
52                                        LINUX_MIB_TCPTOZEROWINDOWADV);
53          } else if (old_win == 0) {
54                  NET_INC_STATS(sock_net(sk), LINUX_MIB_TCPFROMZEROWINDOWADV);
55          }
56
57          return new_win;
58  }
```

在这个过程中，还调用了`__tcp_select_window(sk)`来计算新的窗口大小。该函数会尝试增加窗口的大小，但是有两个限制条件：

1. 窗口不能收缩 (RFC793)
2. 每个 socket 所能使用的内存是有限制的。

RFC 1122 中说：

> "the suggested [SWS] avoidance algorithm for the receiver is to keep RECV.NEXT + RCV.WIN fixed until: RCV.BUFF - RCV.USER - RCV.WINDOW >= min(1/2 RCV.BUFF, MSS)"
>
> 推荐的用于接收方的糊涂窗口综合症的避免算法是保持 recv.next+rcv.win 不变，直到：RCV.BUFF - RCV.USER - RCV.WINDOW >= min(1/2 RCV.BUFF, MSS)

换句话说，就是除非缓存的大小多出来至少一个 MSS 那么多字节，否则不要增长窗口右边界的大小。

然而，根据 Linux 注释中的说法，被推荐的这个算法会破坏头预测 (header prediction)，因为头预测会假定`th->window`不变。严格地说，保持`th->window`固定不变会违背接收方的用于防止糊涂窗口综合症的准则。在这种规则下，一个单字节的包的流会引发窗口的右边界总是提前一个字节。当然，如果发送方实现了预防糊涂窗口综合症的方法，那么就不会出现问题。

Linux 的 TCP 部分的作者们参考了 BSD 的实现方法。BSD 在这方面的做法是是，如果空闲空间小于最大可用空间的 $\frac{1}{4}$，且空闲空间小于 mss 的 $\frac{1}{2}$，那么就把窗口设置为 0。否则，只是单纯地阻止窗口缩小，或者阻止窗口大于最大可表示的范围 (the largest

representable value)。BSD 的方法似乎“意外地”使得窗口基本上都是 MSS 的整倍数。且很多情况下窗口大小都是固定不变的。因此，Linux 采用强制窗口为 MSS 的整倍数，以获得相似的行为。

```
u32 __tcp_select_window(struct sock *sk)
{
        struct inet_connection_sock *icsk = inet_csk(sk);
        struct tcp_sock *tp = tcp_sk(sk);
        int mss = icsk->icsk_ack.rcv_mss;
        int free_space = tcp_space(sk);
        int allowed_space = tcp_full_space(sk);
        int full_space = min_t(int, tp->window_clamp, allowed_space);
        int window;

        /* 如果 mss 超过了总共的空间大小，那么把 mss 限制在允许的空间范围内。 */
        if (mss > full_space)
                mss = full_space;

        if (free_space < (full_space >> 1)) {
                /* 当空闲空间小于允许空间的一半时。 */
                icsk->icsk_ack.quick = 0;

                if (tcp_under_memory_pressure(sk))
                        tp->rcv_ssthresh = min(tp->rcv_ssthresh,
                                               4U * tp->advmss);

                /* free_space 有可能成为新的窗口的大小，因此，需要考虑
                 * 窗口扩展的影响。
                 */
                free_space = round_down(free_space, 1 << tp->rx_opt.rcv_wscale);

                /* 如果空闲空间小于 mss 的大小，或者低于最大允许空间的的 1/16，那么，
                 * 返回 0 窗口。否则，tcp_clamp_window() 会增长接收缓存到 tcp_rmem[2]。
                 * 新进入的数据会由于内酣限制而被丢弃。对于较大的窗口，单纯地探测 mss 的
                 * 大小以宣告 0 窗口有些太晚了（可能会超过限制）。
                 */
                if (free_space < (allowed_space >> 4) || free_space < mss)
                        return 0;
        }

        if (free_space > tp->rcv_ssthresh)
                free_space = tp->rcv_ssthresh;

        /* 这里处理一个例外情况，就是如果开启了窗口缩放，那么就没法对齐 mss 了。
         * 所以就保持窗口是对齐 2 的幂的。
         */
        window = tp->rcv_wnd;
        if (tp->rx_opt.rcv_wscale) {
                window = free_space;

                /* Advertise enough space so that it won't get scaled away.
                 * Import case: prevent zero window announcement if
                 * 1<<rcv_wscale > mss.
                 */
                if (((window >> tp->rx_opt.rcv_wscale) << tp->rx_opt.rcv_wscale) != window)
                        window = (((window >> tp->rx_opt.rcv_wscale) + 1)
                                  << tp->rx_opt.rcv_wscale);
        } else {
```

```
55                  /* 如果内存条件允许，那么就把窗口设置为 mss 的整倍数。
56                   * 或者如果 free_space > 当前窗口大小加上全部允许的空间的一半，
57                   * 那么，就将窗口大小设置为 free_space
58                   */
59                  if (window <= free_space - mss || window > free_space)
60                          window = (free_space / mss) * mss;
61                  else if (mss == full_space &&
62                           free_space > window + (full_space >> 1))
63                          window = free_space;
64          }
65
66          return window;
67  }
```

3.2 TCP 被动打开-服务器

3.2.1 基本流程

tcp 想要被动打开，就必须得先进行 listen 调用。而对于一台主机，它如果想要作为服务器，它会在什么时候进行 listen 调用呢？不难想到，它在启动某个需要 TCP 连接的高级应用程序的时候，就会执行 listen 调用。经过 listen 调用之后，系统内部其实创建了一个监听套接字，专门负责监听是否有数据发来，而不会负责传输数据。

当客户端的第一个 syn 包到达服务器时，其实 linux 内核并不会创建 sock 结构体，而是创建一个轻量级的`request_sock`结构体，里面能唯一确定某个客户端发来的 syn 的信息，接着就发送 syn、ack 给客户端。

客户端一般就接着回 ack。这时，我们能从 ack 中，取出信息，在一堆`request_sock`匹配，看看是否之前有这个 ack 对应的 syn 发过来过。如果之前发过 syn，那么现在我们就能找到`request_sock`，也就是客户端 syn 时建立的`request_sock`。此时，我们内核才会为这条流创建 sock 结构体，毕竟，sock 结构体比`request_sock`大的多，犯不着三次握手都没建立起来我就建立一个大的结构体。当三次握手建立以后，内核就建立一个相对完整的 sock。所谓相对完整，其实也是不完整。因为如果写过 socket 程序，你就知道，所谓的真正完整，是建立 socket，而不是 sock(socket 结构体中有一个指针 sock *sk，显然 sock 只是 socket 的一个子集)。那么我们什么时候才会创建完整的 socket，或者换句话说，什么时候使得 sock 结构体和文件系统关联从而绑定一个 fd，用这个 fd 就可以用来传输数据呢？所谓 fd(file descriptor)，一般是 BSD Socket 的用法，用在 Unix/Linux 系统上。在 Unix/Linux 系统下，一个 socket 句柄，可以看做是一个文件，在 socket 上收发数据，相当于对一个文件进行读写，所以一个 socket 句柄，通常也用表示文件句柄的 fd 来表示。

如果你有 socket 编程经验，那么你一定能想到，那就是在 accept 系统调用时，返回了一个 fd，所以说，是你在 accept 时，你三次握手完成后建立的 sock 才绑定了一个 fd。

3.2.2 第一次握手：接受 SYN 段

3.2.2.1 第一次握手函数调用关系

如图所示。

SYN.png SYN.bb

3.2.2.2 tcp_v4_do_rcv

在进行第一次握手的时候，TCP 必然处于 LISTEN 状态。传输控制块接收处理的段都由tcp_v4_do_rcv来处理。

```
/*
Location:

        /net/ipv4/tcp_ipv4.c

Function:
        The socket must have it's spinlock held when we get
        here, unless it is a TCP_LISTEN socket.

        We have a potential double-lock case here, so even when
        doing backlog processing we use the BH locking scheme.
        This is because we cannot sleep with the original spinlock
        held.

Parameter:

        sk: 传输控制块
        skb: 传输控制块缓冲区
*/
int tcp_v4_do_rcv(struct sock *sk, struct sk_buff *skb)
{
    struct sock *rsk;

    /* 省略无关代码 */

    if (tcp_checksum_complete(skb))
        goto csum_err;

    if (sk->sk_state == TCP_LISTEN) {
        struct sock *nsk = tcp_v4_cookie_check(sk, skb);
```

```
31
32          if (!nsk)
33              goto discard;
34          if (nsk != sk) {
35              sock_rps_save_rxhash(nsk, skb);
36              sk_mark_napi_id(nsk, skb);
37              if (tcp_child_process(sk, nsk, skb)) {
38                  rsk = nsk;
39                  goto reset;
40              }
41              return 0;
42          }
43      } else
44          sock_rps_save_rxhash(sk, skb);
45
46      if (tcp_rcv_state_process(sk, skb)) {
47          rsk = sk;
48          goto reset;
49      }
50      return 0;
51
52  reset:
53      tcp_v4_send_reset(rsk, skb);
54  discard:
55      kfree_skb(skb);
56      /* Be careful here. If this function gets more complicated and
57       * gcc suffers from register pressure on the x86, sk (in \%ebx)
58       * might be destroyed here. This current version compiles correctly,
59       * but you have been warned.
60       */
61      return 0;
62
63  csum_err:
64      TCP_INC_STATS_BH(sock_net(sk), TCP_MIB_CSUMERRORS);
65      TCP_INC_STATS_BH(sock_net(sk), TCP_MIB_INERRS);
66      goto discard;
67  }
```

首先，程序先基于伪首部累加和进行全包的校验和，判断包是否传输正确。

其次，程序会进行相应的 cookie 检查。

最后，程序会继续调用`tcp_rcv_state_process`函数处理接收到的 SYN 段。

3.2.2.3 `tcp_v4_cookie_check`

该函数如下：

```
1   /*
2   Location
3
4           net/ipv4/tcp_ipv4.c
5
6   Function
7
8
9
10  Parameter
11
```

```
        sk: 传输控制块
        skb: 传输控制块缓冲区。
*/
static struct sock *tcp_v4_cookie_check(struct sock *sk, struct sk_buff *skb)
{
#ifdef CONFIG_SYN_COOKIES
    const struct tcphdr *th = tcp_hdr(skb);

    if (!th->syn)
        sk = cookie_v4_check(sk, skb);
#endif
    return sk;
}
```

一般情况下，当前 linux 内核都会定义CONFIG_SYN_COOKIES宏的，显然对于第一次握手的时候，接收到的确实是 syn 包，故而直接返回了 sk。

3.2.2.4 tcp_rcv_state_process

该函数位于/net/ipv4/tcp_input.c中。与第一次握手相关的代码如下：

```
/*
Location:

        /net/ipv4/tcp_input.c

Function:

 *  This function implements the receiving procedure of RFC 793 for
 *  all states except ESTABLISHED and TIME_WAIT.
 *  It's called from both tcp_v4_rcv and tcp_v6_rcv and should be
 *  address independent.

Parameter

        sk: 传输控制块
        skb: 传输控制块缓冲区
*/

int tcp_rcv_state_process(struct sock *sk, struct sk_buff *skb)
{
    struct tcp_sock *tp = tcp_sk(sk);
    struct inet_connection_sock *icsk = inet_csk(sk);
    const struct tcphdr *th = tcp_hdr(skb);
    struct request_sock *req;
    int queued = 0;
    bool acceptable;

    tp->rx_opt.saw_tstamp = 0;  /*saw_tstamp 表示在最新的包上是否看到的时间戳选项 */

    switch (sk->sk_state) {
    /* 省略无关代码 */

        case TCP_LISTEN:
            if (th->ack)
                return 1;

            if (th->rst)
```

```
                goto discard;

            if (th->syn) {
                if (th->fin)
                    goto discard;
                if (icsk->icsk_af_ops->conn_request(sk, skb) < 0)
                    return 1;

                /* Now we have several options: In theory there is
                 * nothing else in the frame. KA9Q has an option to
                 * send data with the syn, BSD accepts data with the
                 * syn up to the [to be] advertised window and
                 * Solaris 2.1 gives you a protocol error. For now
                 * we just ignore it, that fits the spec precisely
                 * and avoids incompatibilities. It would be nice in
                 * future to drop through and process the data.
                 *
                 * Now that TTCP is starting to be used we ought to
                 * queue this data.
                 * But, this leaves one open to an easy denial of
                 * service attack, and SYN cookies can't defend
                 * against this problem. So, we drop the data
                 * in the interest of security over speed unless
                 * it's still in use.
                 */
                kfree_skb(skb);
                return 0;
            }
            goto discard;

            /* 省略无关代码 */
discard:
        __kfree_skb(skb);
    }
    return 0;
}
```

显然，所接收到的包的 ack、rst、fin 字段都不为 1，故而这时开始进行连接检查，判断是否可以允许连接。经过不断查找，我们发现`icsk->icsk_af_ops->conn_request`最终会掉用`tcp_v4_conn_request`进行处理。如果 syn 段合法，内核就会为该连接请求创建连接请求块，并且保存相应的信息。否则，就会返回 1, 原函数会发送 reset 给客户端表明连接请求失败。

当然，如果收到的包的 ack 字段为 1, 那么由于此时链接还未建立，故该包无效，返回 1, 并且调用该函数的函数会发送 reset 包给对方。如果收到的是 rst 字段或者既有 fin 又有 syn 的字段，那就直接销毁，并且释放内存。

3.2.2.5 `tcp_v4_conn_request && tcp_conn_request`

该函数如下：

```
/* why make two functions here?
Location

        /net/ipv4/tcp_ipv4.c

```

```
Function:

        服务端用来处理客户端连接请求的函数。

Parameter:

        sk: 传输控制块
        skb: 传输控制块缓冲区
*/
int tcp_v4_conn_request(struct sock *sk, struct sk_buff *skb)
{
    /* Never answer to SYNs send to broadcast or multicast */
    if (skb_rtable(skb)->rt_flags & (RTCF_BROADCAST | RTCF_MULTICAST))
        goto drop;

    return tcp_conn_request(&tcp_request_sock_ops,
                &tcp_request_sock_ipv4_ops, sk, skb);

drop:
    NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_LISTENDROPS);
    return 0;
}
```

如果一个 SYN 段是要被发送到广播地址和组播地址，则直接 drop 掉，然后返回 0。否则的话，就继续调用tcp_conn_request进行连接处理。如下：

```
/*
Location

        /net/ipv4/tcp_ipv4.c

Function:

        服务端用来处理客户端连接请求的函数。

Parameter:

        rsk_ops: 请求控制块的函数接口?? what is the difference?
        af_ops:TCP 请求块的函数接口
        sk: 传输控制块
        skb: 传输控制块缓冲区
*/
int tcp_conn_request(struct request_sock_ops *rsk_ops,
             const struct tcp_request_sock_ops *af_ops,
             struct sock *sk, struct sk_buff *skb)
{
        /* 初始化 len 字段 */
    struct tcp_fastopen_cookie foc = { .len = -1 };
    /*tw: time wait    isn: initial sequence num 初始化序列号 */
        __u32 isn = TCP_SKB_CB(skb)->tcp_tw_isn;
    struct tcp_options_received tmp_opt;
    struct tcp_sock *tp = tcp_sk(sk);
    struct sock *fastopen_sk = NULL;
    struct dst_entry *dst = NULL;
    struct request_sock *req;
        /* 标志是否开启了 SYN_COOKIE 选项 */
    bool want_cookie = false;
        /* 路由查找相关的数据结构 */
```

```
33      struct flowi fl;
34
35      /* TW(time wait) buckets are converted to open requests without
36       * limitations, they conserve resources and peer is
37       * evidently real one.
38       */
39      if ((sysctl_tcp_syncookies == 2 ||
40           inet_csk_reqsk_queue_is_full(sk)) && !isn) {
41          want_cookie = tcp_syn_flood_action(sk, skb, rsk_ops->slab_name);
42          if (!want_cookie)
43              goto drop;
44      }
```

首先，如果打开了 SYNCOOKIE 选项，并且 SYN 请求队列已满并且 isn 为 0，然后通过函数 **tcp_syn_flood_action** 判断是否需要发送 syncookie。如果没有启用 syncookie 的话，就会返回 false，此时不能接收新的 SYN 请求，会将所收到的包丢掉。

```
1       /* Accept backlog is full. If we have already queued enough
2        * of warm entries in syn queue, drop request. It is better than
3        * clogging syn queue with openreqs with exponentially increasing
4        * timeout.
5        */
6       if (sk_acceptq_is_full(sk) && inet_csk_reqsk_queue_young(sk) > 1) {
7           NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_LISTENOVERFLOWS);
8           goto drop;
9       }
```

warm entries???

如果连接队列长度已经达到上限且 SYN 请求队列中至少有一个握手过程中没有重传过段，则丢弃当前请求。

```
1       req = inet_reqsk_alloc(rsk_ops, sk, !want_cookie);
2       if (!req)
3           goto drop;
```

这时调用`inet_reqsk_alloc`分配一个连接请求块，用于保存连接请求信息，同时初始化在连接过程中用来发送 ACK/RST 段的操作集合，以便在建立连接过程中能方便地调用这些接口。如果申请不了资源的话，就会放弃此次连接请求。

```
1       tcp_rsk(req)->af_specific = af_ops;
```

这一步进行的是为了保护 BGP 会话。???

```
1       tcp_clear_options(&tmp_opt);
2       tmp_opt.mss_clamp = af_ops->mss_clamp;
3       tmp_opt.user_mss  = tp->rx_opt.user_mss;
4       tcp_parse_options(skb, &tmp_opt, 0, want_cookie ? NULL : &foc);
```

之后，清除 TCP 选项，初始化`mss_vlamp`和`user_mss`. 然后调用`tcp_parse_options`解析 SYN 段中的 TCP 选项，查看是否有相关的选项。

```
1       if (want_cookie && !tmp_opt.saw_tstamp)
2           tcp_clear_options(&tmp_opt);
```

如果启动了 syncookies，并且 TCP 选项中没有存在时间戳，则清除已经解析的 TCP 选项。这是因为计算`syn_cookie`必须用到时间戳。

```
    tmp_opt.tstamp_ok = tmp_opt.saw_tstamp;          //tstamp_ok 表示在收到的 SYN 包上看到的 TIMESTAMP
    tcp_openreq_init(req, &tmp_opt, skb, sk);
```

这时，根据收到的 SYN 段中的选项和序号来初始化连接请求块信息。

```
    /* Note: tcp_v6_init_req() might override ir_iif for link locals */
    inet_rsk(req)->ir_iif = sk->sk_bound_dev_if;                    //bound device index if != 0 ?

    af_ops->init_req(req, sk, skb);

    if (security_inet_conn_request(sk, skb, req))
        goto drop_and_free;
```

这一部分于 IPV6 以及安全检测有关，这里不进行详细讲解。安全检测失败的话，就会丢弃 SYN 段。

```
    if (!want_cookie && !isn) {
        /* VJ's idea. We save last timestamp seen
         * from the destination in peer table, when entering
         * state TIME-WAIT, and check against it before
         * accepting new connection request.
         *
         * If "isn" is not zero, this request hit alive
         * timewait bucket, so that all the necessary checks
         * are made in the function processing timewait state.
         */
        if (tcp_death_row.sysctl_tw_recycle) {
            bool strict;

            dst = af_ops->route_req(sk, &fl, req, &strict);

            if (dst && strict &&
                !tcp_peer_is_proven(req, dst, true,
                        tmp_opt.saw_tstamp)) {
                NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_PAWSPASSIVEREJECTED);
                goto drop_and_release;
            }
        }
        /* Kill the following clause, if you dislike this way. */
        else if (!sysctl_tcp_syncookies &&
             (sysctl_max_syn_backlog - inet_csk_reqsk_queue_len(sk) <
              (sysctl_max_syn_backlog >> 2)) &&
             !tcp_peer_is_proven(req, dst, false,
                         tmp_opt.saw_tstamp)) {
            /* Without syncookies last quarter of
             * backlog is filled with destinations,
             * proven to be alive.
             * It means that we continue to communicate
             * to destinations, already remembered
             * to the moment of synflood.
             */
            pr_drop_req(req, ntohs(tcp_hdr(skb)->source),
                    rsk_ops->family);
            goto drop_and_release;
        }
```

```
40
41          isn = af_ops->init_seq(skb);
42      }
```

如果没有开启 syncookie 并且 isn 为 0 的话，其中的第一个 if 从对段信息块中获取时间戳，在新的连接请求之前检测 **PAWS**。后边的表明在没有启动 syncookies 的情况下受到 synflood 攻击，丢弃收到的段。之后由源地址，源端口，目的地址以及目的端口计算出服务端初始序列号。

```
1       if (!dst) {
2           dst = af_ops->route_req(sk, &fl, req, NULL);
3           if (!dst)
4               goto drop_and_free;
5       }
6           /* 显示拥塞控制 */
7       tcp_ecn_create_request(req, skb, sk, dst);
8
9           /* 如果开启了 cookie 的话，就需要进行相应的初始化 */
10      if (want_cookie) {
11          isn = cookie_init_sequence(af_ops, sk, skb, &req->mss);
12          req->cookie_ts = tmp_opt.tstamp_ok;
13          if (!tmp_opt.tstamp_ok)
14              inet_rsk(req)->ecn_ok = 0;
15      }
16
17      tcp_rsk(req)->snt_isn = isn;
18      tcp_rsk(req)->txhash = net_tx_rndhash();
19      tcp_openreq_init_rwin(req, sk, dst);
20      if (!want_cookie) {
21          tcp_reqsk_record_syn(sk, req, skb);
22          fastopen_sk = tcp_try_fastopen(sk, skb, req, &foc, dst);
23      }
24
25          /* 如果开启了 fastopen 的话，顺便发送数据
26                  否则就是简单地发送。
27          */
28      if (fastopen_sk) {
29          af_ops->send_synack(fastopen_sk, dst, &fl, req,
30                      &foc, false);
31          /* Add the child socket directly into the accept queue */
32          inet_csk_reqsk_queue_add(sk, req, fastopen_sk);
33          sk->sk_data_ready(sk);
34          bh_unlock_sock(fastopen_sk);
35          sock_put(fastopen_sk);
36      } else {
37          tcp_rsk(req)->tfo_listener = false;
38          if (!want_cookie)
39              inet_csk_reqsk_queue_hash_add(sk, req, TCP_TIMEOUT_INIT);
40          af_ops->send_synack(sk, dst, &fl, req,
41                      &foc, !want_cookie);
42          if (want_cookie)
43              goto drop_and_free;
44      }
45      reqsk_put(req);
46      return 0;
47
48  drop_and_release:
```

```
49      dst_release(dst);
50  drop_and_free:
51      reqsk_free(req);
52  drop:
53      NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_LISTENDROPS);
54      return 0;
```

3.2.2.6 inet_csk_reqsk_queue_add

```
1   struct sock *inet_csk_reqsk_queue_add(struct sock *sk,
2                         struct request_sock *req,
3                         struct sock *child)
4   {
5       struct request_sock_queue *queue = &inet_csk(sk)->icsk_accept_queue;
6
7       spin_lock(&queue->rskq_lock);
8       if (unlikely(sk->sk_state != TCP_LISTEN)) {
9           inet_child_forget(sk, req, child);
10          child = NULL;
11      } else {
12          req->sk = child;
13          req->dl_next = NULL;
14          if (queue->rskq_accept_head == NULL)
15              queue->rskq_accept_head = req;
16          else
17              queue->rskq_accept_tail->dl_next = req;
18          queue->rskq_accept_tail = req;
19          sk_acceptq_added(sk);
20      }
21      spin_unlock(&queue->rskq_lock);
22      return child;
23  }
```

这一个函数所进行的操作就是直接将请求挂在接收队列中。

3.2.2.7 inet_csk_reqsk_queue_hash_add

```
1   void inet_csk_reqsk_queue_hash_add(struct sock *sk, struct request_sock *req,
2                       unsigned long timeout)
3   {
4       reqsk_queue_hash_req(req, timeout);
5       inet_csk_reqsk_queue_added(sk);
6   }
```

首先将连接请求块保存到父传输请求块的散列表中，并设置定时器超时时间。之后更新已存在的连接请求块数，并启动连接建立定时器。

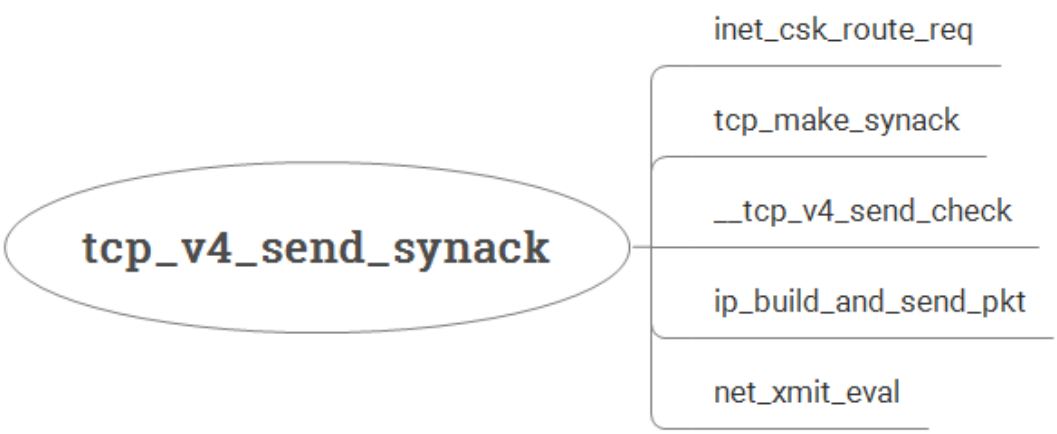
3.2.3 第二次握手：发送 SYN+ACK 段

在第一次握手的最后调用了af_ops->send_synack函数,而该函数最终会调用tcp_v4_send_synack函数进行发送, 故而这里我们这里就从这个函数进行分析。

3.2.3.1 第二次函数调用关系

第二次握手的调用函数关系图如下：

SYN+ACK.png SYN+ACK.bb



3.2.3.2 tcp__v4__send__synack

```
/*
Location:

        net/ipv4/tcp_ipv4.c

Function:

    Send a SYN-ACK after having received a SYN.
    This still operates on a request_sock only, not on a big
    socket.

Paramaters:

    sk: 传输控制块
    dst: 存储缓存路由项中独立于协议的信息。
    fl:
    req: 请求控制块
    foc: 快打开缓存
    attach_req:

*/
static int tcp_v4_send_synack(const struct sock *sk, struct dst_entry *dst,
                  struct flowi *fl,
                  struct request_sock *req,
                  struct tcp_fastopen_cookie *foc,
                  bool attach_req)
{
    const struct inet_request_sock *ireq = inet_rsk(req);
    struct flowi4 fl4;
    int err = -1;
    struct sk_buff *skb;

    /* First, grab a route. */
    if (!dst && (dst = inet_csk_route_req(sk, &fl4, req)) == NULL)
        return -1;

    skb = tcp_make_synack(sk, dst, req, foc, attach_req);

```

```
39      if (skb) {
40          __tcp_v4_send_check(skb, ireq->ir_loc_addr, ireq->ir_rmt_addr);
41
42          err = ip_build_and_send_pkt(skb, sk, ireq->ir_loc_addr,
43                                      ireq->ir_rmt_addr,
44                                      ireq->opt);
45          err = net_xmit_eval(err);
46      }
47
48      return err;
49  }
```

首先，如果传进来的 dst 为空或者根据连接请求块中的信息查询路由表，如果没有查到，那么就直接退出。

否则就，跟据当前的传输控制块，路由信息，请求等信息构建 syn+ack 段。

如果构建成功的话，就生成 TCP 校验码，然后调用`ip_build_and_send_pkt`生成 IP 数据报并且发送出去。

`net_xmit_eval`是什么, 待考虑。

3.2.3.3 `tcp_make_synack`

函数如下:

```
1   /*
2   Location:
3
4           net/ipv4/ycp_output.c
5
6   Function:
7           tcp_make_synack - Prepare a SYN-ACK.
8           Allocate one skb and build a SYNACK packet.
9           @dst is consumed : Caller should not use it again.
10
11                  该函数用来构造一个 SYN+ACK 段,
12                  并初始化 TCP 首部及 SKB 中的各字段项,
13                  填入相应的选项, 如 MSS, SACK, 窗口扩大因子, 时间戳等。
14
15  Parameter:
16          sk          : listener socket
17          dst         : dst entry attached to the SYNACK
18          req         : request_sock pointer
19          foc         :
20          attach_req  :
21   */
22  struct sk_buff *tcp_make_synack(const struct sock *sk, struct dst_entry *dst,
23                  struct request_sock *req,
24                  struct tcp_fastopen_cookie *foc,
25                  bool attach_req)
26  {
27      struct inet_request_sock *ireq = inet_rsk(req);
28      const struct tcp_sock *tp = tcp_sk(sk);
29      struct tcp_md5sig_key *md5 = NULL;
30      struct tcp_out_options opts;
31      struct sk_buff *skb;
32      int tcp_header_size;
33      struct tcphdr *th;
```

```
34      u16 user_mss;
35      int mss;
36
37      skb = alloc_skb(MAX_TCP_HEADER, GFP_ATOMIC);
38      if (unlikely(!skb)) {
39          dst_release(dst);
40          return NULL;
41      }
```

首先为将要发送的数据申请发送缓存，如果没有申请到，那就会返回 NULL。这里有一个unlikely优化分析，参见 appendix(to do by ref).

```
1       /* Reserve space for headers. */
2       skb_reserve(skb, MAX_TCP_HEADER);
```

为 MAC 层，IP 层，TCP 层首部预留必要的空间。

```
1       if (attach_req) {
2           skb_set_owner_w(skb, req_to_sk(req));
3       } else {
4           /* sk is a const pointer, because we want to express multiple
5            * cpu might call us concurrently.
6            * sk->sk_wmem_alloc in an atomic, we can promote to rw.
7            */
8           skb_set_owner_w(skb, (struct sock *)sk);
9       }
10      skb_dst_set(skb, dst);
```

根据attach_req来判断该执行如何执行相关操作 **to do in the future**。然后设置发送缓存的目的路由 **a little confused, why need this**。

```
1       mss = dst_metric_advmss(dst);
2       user_mss = READ_ONCE(tp->rx_opt.user_mss);
3       if (user_mss && user_mss < mss)
4           mss = user_mss;
```

根据每一个路由器上的 mss 以及自身的 mss 来得到最大的 mss。

```
1       memset(&opts, 0, sizeof(opts));
2   #ifdef CONFIG_SYN_COOKIES
3       if (unlikely(req->cookie_ts))
4           skb->skb_mstamp.stamp_jiffies = cookie_init_timestamp(req);
5       else
6   #endif
7       skb_mstamp_get(&skb->skb_mstamp);
```

清除选项，并且设置相关时间戳 **to add in future**。

```
1   #ifdef CONFIG_TCP_MD5SIG
2       rcu_read_lock();
3       md5 = tcp_rsk(req)->af_specific->req_md5_lookup(sk, req_to_sk(req));
4   #endif
```

查看是否有 MD5 选项，有的话构造出相应的 md5.

```
    skb_set_hash(skb, tcp_rsk(req)->txhash, PKT_HASH_TYPE_L4);
    tcp_header_size = tcp_synack_options(req, mss, skb, &opts, md5, foc) +
              sizeof(*th);

    skb_push(skb, tcp_header_size);
    skb_reset_transport_header(skb);
```

得到 tcp 的头部大小，然后进行大小设置，并且重置传输层的头部。

```
    th = tcp_hdr(skb);
    memset(th, 0, sizeof(struct tcphdr));
    th->syn = 1;
    th->ack = 1;
    tcp_ecn_make_synack(req, th);
    th->source = htons(ireq->ir_num);
    th->dest = ireq->ir_rmt_port;
```

清空 tcp 头部，并设置 tcp 头部的各个字段。

```
    /* Setting of flags are superfluous here for callers (and ECE is
     * not even correctly set)
     */
    tcp_init_nondata_skb(skb, tcp_rsk(req)->snt_isn,
                 TCPHDR_SYN | TCPHDR_ACK);

    th->seq = htonl(TCP_SKB_CB(skb)->seq);
    /* XXX data is queued and acked as is. No buffer/window check */
    th->ack_seq = htonl(tcp_rsk(req)->rcv_nxt);

    /* RFC1323: The window in SYN & SYN/ACK segments is never scaled. */
    th->window = htons(min(req->rsk_rcv_wnd, 65535U));
    tcp_options_write((__be32 *)(th + 1), NULL, &opts);
    th->doff = (tcp_header_size >> 2);
    TCP_INC_STATS_BH(sock_net(sk), TCP_MIB_OUTSEGS);
```

首先初始化不含数据的 tcp 报文，然后设置相关的序列号，确认序列号，窗口大小，选项字段，以及 TCP 数据偏移，之所以除以 4，是 doff 的单位是 32 位字，即以四个字节长的字为计算单位。

```
#ifdef CONFIG_TCP_MD5SIG
    /* Okay, we have all we need - do the md5 hash if needed */
    if (md5)
        tcp_rsk(req)->af_specific->calc_md5_hash(opts.hash_location,
                             md5, req_to_sk(req), skb);
    rcu_read_unlock();
#endif

    /* Do not fool tcpdump (if any), clean our debris */
    skb->tstamp.tv64 = 0;
    return skb;
}
```

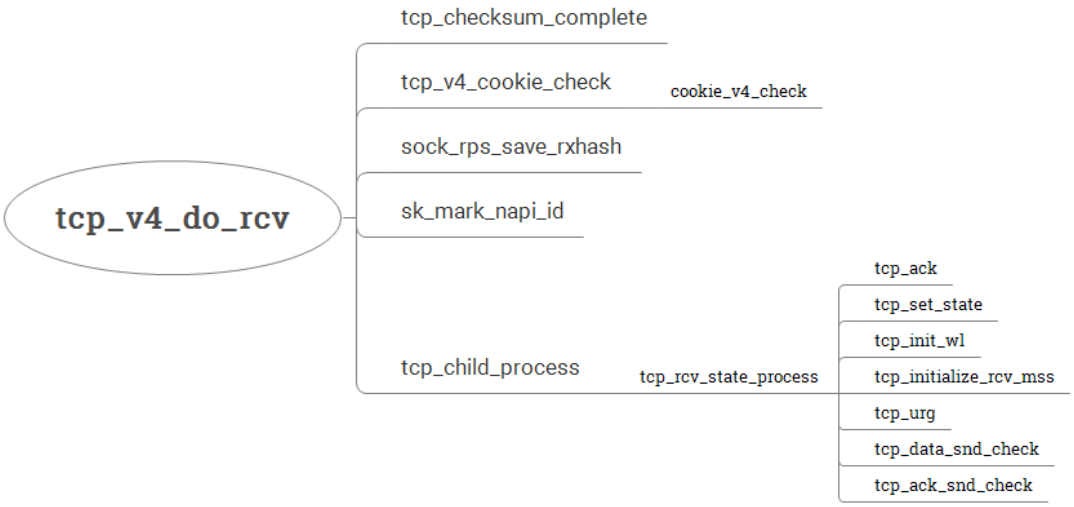
最后判断是否需要 md5 哈希值，如果需要的话，就进行添加。**what does it mean in the middle?**，然后返回生成包含 SYN+ACK 段的 skb。

3.2.4 第三次握手：接收 ACK 段

在服务器第二次我受的最后启动了建立连接定时器，等待客户端最后一次握手的 ACK 段。

3.2.4.1 第三次握手函数调用关系图

ACK.png ACK.bb



3.2.4.2 tcp_v4_do_rcv

```
/*
Location:

        /net/ipv4/tcp_ipv4.c

Function:

        The socket must have it's spinlock held when we get
        here, unless it is a TCP_LISTEN socket.

        We have a potential double-lock case here, so even when
        doing backlog processing we use the BH locking scheme.
        This is because we cannot sleep with the original spinlock
        held.

Parameter:

        sk: 传输控制块。
        skb: 传输控制块缓冲区。
*/
int tcp_v4_do_rcv(struct sock *sk, struct sk_buff *skb)
{
    struct sock *rsk;

    /** 省略无关代码 **/

```

```
27      if (tcp_checksum_complete(skb))
28          goto csum_err;
29
30      if (sk->sk_state == TCP_LISTEN) {
31          struct sock *nsk = tcp_v4_cookie_check(sk, skb);
32
33          if (!nsk)
34              goto discard;
35          if (nsk != sk) {
36              sock_rps_save_rxhash(nsk, skb);
37              sk_mark_napi_id(nsk, skb);
38              if (tcp_child_process(sk, nsk, skb)) {
39                  rsk = nsk;
40                  goto reset;
41              }
42              return 0;
43          }
44      } else
45          sock_rps_save_rxhash(sk, skb);
46      /** 省略无关代码 **/
47  reset:
48      tcp_v4_send_reset(rsk, skb);
49  discard:
50      kfree_skb(skb);
51      /* Be careful here. If this function gets more complicated and
52       * gcc suffers from register pressure on the x86, sk (in %ebx)
53       * might be destroyed here. This current version compiles correctly,
54       * but you have been warned.
55       */
56      return 0;
57
58  csum_err:
59      TCP_INC_STATS_BH(sock_net(sk), TCP_MIB_CSUMERRORS);
60      TCP_INC_STATS_BH(sock_net(sk), TCP_MIB_INERRS);
61      goto discard;
62  }
```

在服务器最后一次握手的时候，其实传输控制块仍然处于 LISTEN 状态，但是这时候 cookie 检查得到的传输控制块已经不是侦听传输控制块了，故而会执行`tcp_child_process`来初始化子传输控制块。如果初始化失败的话（返回值非零），就会给客户端发送 RST 段进行复位。

3.2.4.3 tcp_v4_cookie_check

```
1   static struct sock *tcp_v4_cookie_check(struct sock *sk, struct sk_buff *skb)
2   {
3   #ifdef CONFIG_SYN_COOKIES
4       const struct tcphdr *th = tcp_hdr(skb);
5
6       if (!th->syn)
7           sk = cookie_v4_check(sk, skb);
8   #endif
9       return sk;
10  }
```

在现在的 Linux 内核中一般都会定义`CONFIG_SYN_COOKIES`宏，此时在第三次握手阶段，并不是 syn 包，内核就会执行`cookie_v4_check`。在这个函数中，服务器会将客户

端的 ACK 序列号减去 1, 得到 cookie 比较值，然后将客户端的 IP 地址，客户端端口，服务器 IP 地址和服务器端口，接收到的 TCP 序列好以及其它一些安全数值等要素进行 hash 运算后，与该 cookie 比较值比较，如果相等，则直接完成三次握手，此时不必查看该连接是否属于请求连接队列。

3.2.4.4 tcp_child_process

该函数位于/net/ipv4/minisocks.c中，子传输控制块开始处理 TCP 段。

```
/*
Location:

        net/ipv4/tcp_minisock.c

Functions:

    Queue segment on the new socket if the new socket is active,
    otherwise we just shortcircuit this and continue with
    the new socket.

    For the vast majority of cases child->sk_state will be TCP_SYN_RECV
    when entering. But other states are possible due to a race condition
    where after __inet_lookup_established() fails but before the listener
    locked is obtained, other packets cause the same connection to
    be created.
Parameters:

        parent: 父传出控制块
        child: 子传输控制块
        skb: 传输控制块缓存
*/

int tcp_child_process(struct sock *parent, struct sock *child,
              struct sk_buff *skb)
{
    int ret = 0;
    int state = child->sk_state;

    tcp_sk(child)->segs_in += max_t(u16, 1, skb_shinfo(skb)->gso_segs);
    if (!sock_owned_by_user(child)) {
        ret = tcp_rcv_state_process(child, skb);
        /* Wakeup parent, send SIGIO */
        if (state == TCP_SYN_RECV && child->sk_state != state)
            parent->sk_data_ready(parent);
    } else {
        /* Alas, it is possible again, because we do lookup
         * in main socket hash table and lock on listening
         * socket does not protect us more.
         */
        __sk_add_backlog(child, skb);
    }

    bh_unlock_sock(child);
    sock_put(child);
    return ret;
}
```

首先，如果此时刚刚创建的新的子传输控制块没有被用户进程占用，则根据作为第三次握手的 ACK 段，调用tcp_rcv_state_process继续对子传输控制块做初始化。否则的话，只能将其加入后备队列中，等空闲时再进行处理。虽然这种情况出现的概率小，但是也是有可能发生的。

3.2.4.5 tcp_rcv_state_process

该函数位于/net/ipv4/tcp_input.c中。

该函数用来处理 ESTABLISHED 和TIME_WAIT状态以外的 TCP 段,这里处理SYN_RECV状态。

```
    acceptable = tcp_ack(sk, skb, FLAG_SLOWPATH |
                         FLAG_UPDATE_TS_RECENT) > 0;
```

首先对收到的 ACK 段进行处理判断是否正确接收，如果正确接收就会发送返回非零值。

```
    switch (sk->sk_state) {
        case TCP_SYN_RECV:
            if (!acceptable)
                return 1;

            if (!tp->srtt_us)
                tcp_synack_rtt_meas(sk, req);

            /* Once we leave TCP_SYN_RECV, we no longer need req
             * so release it.
             */
            if (req) {
                tp->total_retrans = req->num_retrans;
                reqsk_fastopen_remove(sk, req, false);
            } else {
                /* Make sure socket is routed, for correct metrics. */
                icsk->icsk_af_ops->rebuild_header(sk);
                tcp_init_congestion_control(sk);

                tcp_mtup_init(sk);
                tp->copied_seq = tp->rcv_nxt;
                tcp_init_buffer_space(sk);
            }
            smp_mb();
            tcp_set_state(sk, TCP_ESTABLISHED);
            sk->sk_state_change(sk);
```

进行一系列的初始化，开启相应拥塞控制等，并且将 TCP 的状态置为 ESTABLISHED。

```
            /* Note, that this wakeup is only for marginal crossed SYN case.
             * Passively open sockets are not waked up, because
             * sk->sk_sleep == NULL and sk->sk_socket == NULL.
             */
            if (sk->sk_socket)
                sk_wake_async(sk, SOCK_WAKE_IO, POLL_OUT);
```

发信号给那些将通过该套接口发送数据的进程，通知它们套接口目前已经可以发送数据了。

```
            tp->snd_una = TCP_SKB_CB(skb)->ack_seq;
            tp->snd_wnd = ntohs(th->window) << tp->rx_opt.snd_wscale;
            tcp_init_wl(tp, TCP_SKB_CB(skb)->seq);

            if (tp->rx_opt.tstamp_ok)
                tp->advmss -= TCPOLEN_TSTAMP_ALIGNED;
```

初始化传输控制块的各个字段，对时间戳进行处理。

```
            if (req) {
                /* Re-arm the timer because data may have been sent out.
                 * This is similar to the regular data transmission case
                 * when new data has just been ack'ed.
                 *
                 * (TFO) - we could try to be more aggressive and
                 * retransmitting any data sooner based on when they
                 * are sent out.
                 */
                tcp_rearm_rto(sk);
            } else
                tcp_init_metrics(sk);
```

为该套接口初始化路由。

```
            tcp_update_pacing_rate(sk);

            /* Prevent spurious tcp_cwnd_restart() on first data packet */
            tp->lsndtime = tcp_time_stamp;

            tcp_initialize_rcv_mss(sk);
            tcp_fast_path_on(tp);
            break;
```

更新最近一次的发送数据报的时间，初始化与路径 MTU 有关的成员，并计算有关 TCP 首部预测的标志。

```
    /* step 6: check the URG bit */
    tcp_urg(sk, skb, th);
```

检测带外数据标志位。

```
    /* step 7: process the segment text */
    switch (sk->sk_state) {
        /** 省略无关代码 **/
        case TCP_ESTABLISHED:
            tcp_data_queue(sk, skb);
            queued = 1;
            break;
    }
```

对已接收到的 TCP 段排队，在建立连接阶段一般不会收到 TCP 段。

```
    /* tcp_data could move socket to TIME-WAIT */
    if (sk->sk_state != TCP_CLOSE) {
        tcp_data_snd_check(sk);
        tcp_ack_snd_check(sk);
    }

    if (!queued) {
```

```
8   discard:
9           __kfree_skb(skb);
10      }
11      return 0;
```

显然此时状态不为 CLOSE，故而就回去检测是否数据和 ACK 要发送。其次，根据 queue 标志来确定是否释放接收到的 TCP 段，如果接收到的 TCP 段已添加到接收队列中，则不释放。

CHAPTER 4

TCP 连接释放

Contents

4.1 主动关闭

4.1.1 第一次握手——发送 FIN

通过 shutdown 系统调用，主动关闭 TCP 连接。该系统调用最终由`tcp_shutdown`实现。代码如下：

```
/*
Location:

        net/ipv4/tcp.c

Function:

        Shutdown the sending side of a connection. Much like close except
```

```
9           that we don't receive shut down or sock_set_flag(sk, SOCK_DEAD).
10
11  Parameter:
12
13          sk: 传输控制块
14          how:???
15  */
16  void tcp_shutdown(struct sock *sk, int how)
17  {
18          /*      We need to grab some memory, and put together a FIN,
19           *      and then put it into the queue to be sent.
20           *              Tim MacKenzie(tym@dibbler.cs.monash.edu.au) 4 Dec '92.
21           */
22          if (!(how & SEND_SHUTDOWN))
23                  return;
24
25          /* 如果此时已经发送一个 FIN 了，就跳过。 */
26          if ((1 << sk->sk_state) &
27              (TCPF_ESTABLISHED | TCPF_SYN_SENT |
28               TCPF_SYN_RECV | TCPF_CLOSE_WAIT)) {
29                  /* Clear out any half completed packets.  FIN if needed. */
30                  if (tcp_close_state(sk))
31                          tcp_send_fin(sk);
32          }
33  }
```

该函数会在需要发送 FIN 时，调用tcp_close_state()来设置 TCP 的状态。该函数会根据当前的状态，按照1.3.1.1中给出的状态图。

```
1   static const unsigned char new_state[16] = {
2     /* 当前状态:              新的状态:        动作:      */
3     [0 /* (Invalid) */]   = TCP_CLOSE,
4     [TCP_ESTABLISHED]     = TCP_FIN_WAIT1 | TCP_ACTION_FIN,
5     [TCP_SYN_SENT]        = TCP_CLOSE,
6     [TCP_SYN_RECV]        = TCP_FIN_WAIT1 | TCP_ACTION_FIN,
7     [TCP_FIN_WAIT1]       = TCP_FIN_WAIT1,
8     [TCP_FIN_WAIT2]       = TCP_FIN_WAIT2,
9     [TCP_TIME_WAIT]       = TCP_CLOSE,
10    [TCP_CLOSE]           = TCP_CLOSE,
11    [TCP_CLOSE_WAIT]      = TCP_LAST_ACK  | TCP_ACTION_FIN,
12    [TCP_LAST_ACK]        = TCP_LAST_ACK,
13    [TCP_LISTEN]          = TCP_CLOSE,
14    [TCP_CLOSING]         = TCP_CLOSING,
15    [TCP_NEW_SYN_RECV]    = TCP_CLOSE,    /* should not happen ! */
16  };
17
18  static int tcp_close_state(struct sock *sk)
19  {
20          int next = (int)new_state[sk->sk_state];
21          int ns = next & TCP_STATE_MASK;
22
23          /* 根据状态图进行状态转移 */
24          tcp_set_state(sk, ns);
25
26          /* 如果需要执行发送 FIN 的动作，则返回真 */
27          return next & TCP_ACTION_FIN;
28  }
```

可以看出,只有当当前状态为 TCP_ESTABLISHED、TCP_SYN_RECV、TCP_CLOSE_WAIT

时，需要发送 FIN 包。这个也和 TCP 状态图一致。如果需要发送 FIN 包，则会调用 `tcp_send_fin`。

```
1  void tcp_send_fin(struct sock *sk)
2  {
3          struct sk_buff *skb, *tskb = tcp_write_queue_tail(sk);
4          struct tcp_sock *tp = tcp_sk(sk);
5
6          /* 这里做了一些优化，如果发送队列的末尾还有段没有发出去，则利用该段发送 FIN。 */
7          if (tskb && (tcp_send_head(sk) || tcp_under_memory_pressure(sk))) {
8            /* 如果当前正在发送的队列不为空，或者当前 TCP 处于内存压力下，则进行该优化 */
9  coalesce:
10                 TCP_SKB_CB(tskb)->tcp_flags |= TCPHDR_FIN;
11                 TCP_SKB_CB(tskb)->end_seq++;
12                 tp->write_seq++;
13                 if (!tcp_send_head(sk)) {
14                         /* This means tskb was already sent.
15                          * Pretend we included the FIN on previous transmit.
16                          * We need to set tp->snd_nxt to the value it would have
17                          * if FIN had been sent. This is because retransmit path
18                          * does not change tp->snd_nxt.
19                          */
20                         tp->snd_nxt++;
21                         return;
22                 }
23         } else {
24                 /* 为封包分配空间 */
25                 skb = alloc_skb_fclone(MAX_TCP_HEADER, sk->sk_allocation);
26                 if (unlikely(!skb)) {
27                         /* 如果分配不到空间，且队尾还有未发送的包，利用该包发出 FIN。 */
28                         if (tskb)
29                                 goto coalesce;
30                         return;
31                 }
32                 skb_reserve(skb, MAX_TCP_HEADER);
33                 sk_forced_mem_schedule(sk, skb->truesize);
34                 /* FIN eats a sequence byte, write_seq advanced by tcp_queue_skb(). */
35                 /* 构造一个 FIN 包，并加入发送队列。 */
36                 tcp_init_nondata_skb(skb, tp->write_seq,
37                                      TCPHDR_ACK | TCPHDR_FIN);
38                 tcp_queue_skb(sk, skb);
39         }
40         __tcp_push_pending_frames(sk, tcp_current_mss(sk), TCP_NAGLE_OFF);
41 }
```

在函数的最后，将所有的剩余数据一口气发出去，完成发送 FIN 包的过程。至此，主动关闭过程的第一次握手完成。

4.1.2 第二次握手——接受 ACK

在发出 FIN 后，接收端会回复 ACK 确认收到了请求。从这里开始有两种情况，这里先考虑教科书式的四次握手的情况。双方同时发出 FIN 的情况会在4.1.5中描述。根据状态图，主动发出 FIN 包后，会进入`FIN_WAIT1`状态。根据这一信息，可以从`tcp_rcv_state_process`中，找到相应的代码。

```
int tcp_rcv_state_process(struct sock *sk, struct sk_buff *skb)
{
        struct tcp_sock *tp = tcp_sk(sk);
        struct inet_connection_sock *icsk = inet_csk(sk);
        const struct tcphdr *th = tcp_hdr(skb);
        struct request_sock *req;
        int queued = 0;
        bool acceptable;

        tp->rx_opt.saw_tstamp = 0;
        switch (sk->sk_state) {
        case TCP_CLOSE:
                goto discard;

        case TCP_LISTEN:
                /* LISTEN 状态处理代码，略去 */

        case TCP_SYN_SENT:
                /* SYN-SENT 状态处理代码，略去 */
        }

        /* fastopen 相关代码及各类合法性判断，略去 */

        switch (sk->sk_state) {
        case TCP_SYN_RECV:
                /* SYN-RECV 状态处理代码，略去 */

        case TCP_FIN_WAIT1: {
                struct dst_entry *dst;
                int tmo;

                /* 如果当前的套接字为开启了 Fast Open 的套接字，且该 ACK 为
                 * 接收到的第一个 ACK，那么这个 ACK 应该是在确认 SYNACK 包，
                 * 因此，停止 SYNACK 计时器。
                 */
                if (req) {
                        /* Return RST if ack_seq is invalid.
                         * Note that RFC793 only says to generate a
                         * DUPACK for it but for TCP Fast Open it seems
                         * better to treat this case like TCP_SYN_RECV
                         * above.
                         */
                        if (!acceptable)
                                return 1;
                        /* 移除 fastopen 请求 */
                        reqsk_fastopen_remove(sk, req, false);
                        tcp_rearm_rto(sk);
                }
                if (tp->snd_una != tp->write_seq)
                        break;

                /* 收到 ACK 后，转移到 TCP_FIN_WAIT2 状态，将发送端关闭。 */
                tcp_set_state(sk, TCP_FIN_WAIT2);
                sk->sk_shutdown |= SEND_SHUTDOWN;

                /* 确认路由缓存有效 */
                dst = __sk_dst_get(sk);
                if (dst)
```

```
59                      dst_confirm(dst);
60
61              /* 唤醒等待该套接字的进程 */
62              if (!sock_flag(sk, SOCK_DEAD)) {
63                      /* Wake up lingering close() */
64                      sk->sk_state_change(sk);
65                      break;
66              }
67
68              /* 如果所有发送的字节都被确认了，那么进入关闭状态。 */
69              if (tp->linger2 < 0 ||
70                  (TCP_SKB_CB(skb)->end_seq != TCP_SKB_CB(skb)->seq &&
71                   after(TCP_SKB_CB(skb)->end_seq - th->fin, tp->rcv_nxt))) {
72                      tcp_done(sk);
73                      NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPABORTONDATA);
74                      return 1;
75              }
```

转换到TCP_FIN_WAIT2以后，计算接受 fin 包的超时时间。如果还能留出 TIMEWAIT 阶段的时间（TIMEWAIT 阶段有最长时间限制），那么在此之前，就激活保活计时器保持连接。如果时间已经不足了，就主动调用tcp_time_wait 进入 TIMEWAIT 状态。

```
1               tmo = tcp_fin_time(sk);
2               if (tmo > TCP_TIMEWAIT_LEN) {
3                       inet_csk_reset_keepalive_timer(sk, tmo - TCP_TIMEWAIT_LEN);
4               } else if (th->fin || sock_owned_by_user(sk)) {
5                       /* Bad case. We could lose such FIN otherwise.
6                        * It is not a big problem, but it looks confusing
7                        * and not so rare event. We still can lose it now,
8                        * if it spins in bh_lock_sock(), but it is really
9                        * marginal case.
10                       */
11                      inet_csk_reset_keepalive_timer(sk, tmo);
12              } else {
13                      /* 进入 TCP_FIN_WAIT2 状态等待。 */
14                      tcp_time_wait(sk, TCP_FIN_WAIT2, tmo);
15                      goto discard;
16              }
17              break;
18
19              /* 其余状态处理代码，略去 */
20      }
21
22      /* step 6: check the URG bit */
23      tcp_urg(sk, skb, th);
24
25      /* step 7: process the segment text */
26      switch (sk->sk_state) {
27        /* 其他状态处理代码，略去 */
28
29      case TCP_FIN_WAIT1:
30      case TCP_FIN_WAIT2:
31              /* RFC 793 says to queue data in these states,
32               * RFC 1122 says we MUST send a reset.
33               * BSD 4.4 also does reset.
34               */
35              if (sk->sk_shutdown & RCV_SHUTDOWN) {
36                      if (TCP_SKB_CB(skb)->end_seq != TCP_SKB_CB(skb)->seq &&
```

```
37                              after(TCP_SKB_CB(skb)->end_seq - th->fin, tp->rcv_nxt)) {
38                                  NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPABORTONDATA);
39                                  /* 如果接收端已经关闭了，那么发送 RESET。 */
40                                  tcp_reset(sk);
41                                  return 1;
42                          }
43                  }
44                  /* Fall through */
45
46             /* 其他状态处理代码，略去 */
47          }
48
49          /* tcp_data could move socket to TIME-WAIT */
50          if (sk->sk_state != TCP_CLOSE) {
51                  tcp_data_snd_check(sk);
52                  tcp_ack_snd_check(sk);
53          }
54
55          if (!queued) {
56  discard:
57                  __kfree_skb(skb);
58          }
59          return 0;
60  }
```

执行完该段代码后，则进入了FIN_WAIT2状态。

由于进入到FIN_WAIT2状态后，不会再处理 TCP 段的数据。因此，出于资源和方面的考虑，采用了一个较小的结构体tcp_timewait_sock来取代正常的 TCP 传输控制块。TIME_WAIT也是可作同样处理。该替换过程通过函数tcp_time_wait完成。

```
1   void tcp_time_wait(struct sock *sk, int state, int timeo)
2   {
3           const struct inet_connection_sock *icsk = inet_csk(sk);
4           const struct tcp_sock *tp = tcp_sk(sk);
5           struct inet_timewait_sock *tw;
6           bool recycle_ok = false;
7
8           if (tcp_death_row.sysctl_tw_recycle && tp->rx_opt.ts_recent_stamp)
9                   recycle_ok = tcp_remember_stamp(sk);
10
11          /* 分配空间 */
12          tw = inet_twsk_alloc(sk, &tcp_death_row, state);
13
14          if (tw) {
15                  struct tcp_timewait_sock *tcptw = tcp_twsk((struct sock *)tw);
16                  const int rto = (icsk->icsk_rto << 2) - (icsk->icsk_rto >> 1);
17                  struct inet_sock *inet = inet_sk(sk);
18
19                  /* 将值复制给对应的域 */
20                  tw->tw_transparent      = inet->transparent;
21                  tw->tw_rcv_wscale       = tp->rx_opt.rcv_wscale;
22                  tcptw->tw_rcv_nxt       = tp->rcv_nxt;
23                  tcptw->tw_snd_nxt       = tp->snd_nxt;
24                  tcptw->tw_rcv_wnd       = tcp_receive_window(tp);
25                  tcptw->tw_ts_recent     = tp->rx_opt.ts_recent;
26                  tcptw->tw_ts_recent_stamp = tp->rx_opt.ts_recent_stamp;
27                  tcptw->tw_ts_offset     = tp->tsoffset;
```

```
28                  tcptw->tw_last_oow_ack_time = 0;
29
30                  /* 部分对于 ipv6 和 md5 的处理，略过 */
31
32                  /* Get the TIME_WAIT timeout firing. */
33                  if (timeo < rto)
34                          timeo = rto;
35
36                  if (recycle_ok) {
37                          tw->tw_timeout = rto;
38                  } else {
39                          tw->tw_timeout = TCP_TIMEWAIT_LEN;
40                          if (state == TCP_TIME_WAIT)
41                                  timeo = TCP_TIMEWAIT_LEN;
42                  }
43
44                  /* 启动定时器 */
45                  inet_twsk_schedule(tw, timeo);
46                  /* 将 timewait 控制块插入到哈希表中，替代原有的传输控制块 */
47                  __inet_twsk_hashdance(tw, sk, &tcp_hashinfo);
48                  inet_twsk_put(tw);
49          } else {
50                  /* 当内存不够时，直接关闭连接 */
51                  NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPTIMEWAITOVERFLOW);
52          }
53
54          /* 更新一些测量值并关闭原来的传输控制块 */
55          tcp_update_metrics(sk);
56          tcp_done(sk);
57  }
```

4.1.3 第三次握手——接受 FIN

此时，由于已经使用了 timewait 控制块取代了 TCP 控制块。因此，对应的处理代码不再位于`tcp_rcv_state_process`中，而是换到了`tcp_timewait_state_process`函数中。该函数的代码如下，可以看到参数中已经变成了`inet_timewait_sock`。

```
1   enum tcp_tw_status
2   tcp_timewait_state_process(struct inet_timewait_sock *tw, struct sk_buff *skb,
3                              const struct tcphdr *th)
4   {
5           struct tcp_options_received tmp_opt;
6           struct tcp_timewait_sock *tcptw = tcp_twsk((struct sock *)tw);
7           bool paws_reject = false;
8
9           tmp_opt.saw_tstamp = 0;
10          if (th->doff > (sizeof(*th) >> 2) && tcptw->tw_ts_recent_stamp) {
11                  tcp_parse_options(skb, &tmp_opt, 0, NULL);
12
13                  if (tmp_opt.saw_tstamp) {
14                          tmp_opt.rcv_tsecr       -= tcptw->tw_ts_offset;
15                          tmp_opt.ts_recent       = tcptw->tw_ts_recent;
16                          tmp_opt.ts_recent_stamp = tcptw->tw_ts_recent_stamp;
17                          paws_reject = tcp_paws_reject(&tmp_opt, th->rst);
18                  }
19          }
```

检测收到的包是否含有时间戳选项，如果有，则进行 PAWS 相关的检测。之后，开始进

行 TCP_FIN_WAIT2相关的处理。

```
        if (tw->tw_substate == TCP_FIN_WAIT2) {
                /* 重复 tcp_rcv_state_process() 所进行的所有检测 */

                /* 序号不在窗口内，发送 ACK */
                if (paws_reject ||
                    !tcp_in_window(TCP_SKB_CB(skb)->seq, TCP_SKB_CB(skb)->end_seq,
                                   tcptw->tw_rcv_nxt,
                                   tcptw->tw_rcv_nxt + tcptw->tw_rcv_wnd))
                        return tcp_timewait_check_oow_rate_limit(
                                tw, skb, LINUX_MIB_TCPACKSKIPPEDFINWAIT2);

                /* 如果收到 RST 包，则销毁 timewait 控制块并返回 TCP_TW_SUCCESS */
                if (th->rst)
                        goto kill;

                /* 如果收到 SYN 包，则销毁并发送 RST */
                if (th->syn && !before(TCP_SKB_CB(skb)->seq, tcptw->tw_rcv_nxt))
                        goto kill_with_rst;

                /* 如果收到 DACK，则释放该控制块 */
                if (!th->ack ||
                    !after(TCP_SKB_CB(skb)->end_seq, tcptw->tw_rcv_nxt) ||
                    TCP_SKB_CB(skb)->end_seq == TCP_SKB_CB(skb)->seq) {
                        inet_twsk_put(tw);
                        return TCP_TW_SUCCESS;
                }

                /* 之后只有两种情况，有新数据或收到 FIN 包 */
                if (!th->fin ||
                    TCP_SKB_CB(skb)->end_seq != tcptw->tw_rcv_nxt + 1) {
                        /* 如果收到了新的数据或者序号有问题，
                         * 则销毁控制块并发送 RST。
                         */
kill_with_rst:
                        inet_twsk_deschedule_put(tw);
                        return TCP_TW_RST;
                }

                /* 收到了 FIN 包，进入 TIME_WAIT 状态 */
                tw->tw_substate   = TCP_TIME_WAIT;
                tcptw->tw_rcv_nxt = TCP_SKB_CB(skb)->end_seq;
                /* 如果启用了时间戳选项，则设置相关属性 */
                if (tmp_opt.saw_tstamp) {
                        tcptw->tw_ts_recent_stamp = get_seconds();
                        tcptw->tw_ts_recent       = tmp_opt.rcv_tsval;
                }

                /* 启动 TIME_WAIT 定时器 */
                if (tcp_death_row.sysctl_tw_recycle &&
                    tcptw->tw_ts_recent_stamp &&
                    tcp_tw_remember_stamp(tw))
                        inet_twsk_reschedule(tw, tw->tw_timeout);
                else
                        inet_twsk_reschedule(tw, TCP_TIMEWAIT_LEN);
                return TCP_TW_ACK;
        }

```

```
58          /* TIME_WAIT 阶段处理代码 */
59
60  }
```

4.1.4 第四次握手——发送 ACK

在tcp_v4_rcv中,如果发现目前的连接处于FIN_WAIT2 或TIME_WAIT状态,则调用tcp_timewait_state进行处理，根据其返回值，执行相关操作。

```
1   switch (tcp_timewait_state_process(inet_twsk(sk), skb, th)) {
2           case TCP_TW_SYN: {
3                   struct sock *sk2 = inet_lookup_listener(dev_net(skb->dev),
4                                                           &tcp_hashinfo,
5                                                           iph->saddr, th->source,
6                                                           iph->daddr, th->dest,
7                                                           inet_iif(skb));
8                   if (sk2) {
9                           inet_twsk_deschedule_put(inet_twsk(sk));
10                          sk = sk2;
11                          goto process;
12                  }
13                  /* Fall through to ACK */
14          }
15          case TCP_TW_ACK:
16                  /* 回复 ACK 包 */
17                  tcp_v4_timewait_ack(sk, skb);
18                  break;
19          case TCP_TW_RST:
20                  goto no_tcp_socket;
21          case TCP_TW_SUCCESS:;
22  }
```

根据上面的分析,在正常情况下,tcp_timewait_state_process会返回 TCP_TW_ACK,因此，会调用tcp_v4_timewait_ack。该函数如下:

```
1   static void tcp_v4_timewait_ack(struct sock *sk, struct sk_buff *skb)
2   {
3           struct inet_timewait_sock *tw = inet_twsk(sk);
4           struct tcp_timewait_sock *tcptw = tcp_twsk(sk);
5
6           /* 发送 ACK 包 */
7           tcp_v4_send_ack(sock_net(sk), skb,
8                           tcptw->tw_snd_nxt, tcptw->tw_rcv_nxt,
9                           tcptw->tw_rcv_wnd >> tw->tw_rcv_wscale,
10                          tcp_time_stamp + tcptw->tw_ts_offset,
11                          tcptw->tw_ts_recent,
12                          tw->tw_bound_dev_if,
13                          tcp_twsk_md5_key(tcptw),
14                          tw->tw_transparent ? IP_REPLY_ARG_NOSRCCHECK : 0,
15                          tw->tw_tos
16                          );
17
18          /* 释放 timewait 控制块 */
19          inet_twsk_put(tw);
20  }
```

紧接着又将发送 ACK 包的任务交给tcp_v4_send_ack来执行。

```
/* 下面的代码负责在 SYN_RECV 和 TIME_WAIT 状态下发送 ACK 包。
 *
 * The code following below sending ACKs in SYN-RECV and TIME-WAIT states
 * outside socket context is ugly, certainly. What can I do?
 */
static void tcp_v4_send_ack(struct net *net,
                            struct sk_buff *skb, u32 seq, u32 ack,
                            u32 win, u32 tsval, u32 tsecr, int oif,
                            struct tcp_md5sig_key *key,
                            int reply_flags, u8 tos)
{
        const struct tcphdr *th = tcp_hdr(skb);
        struct {
                struct tcphdr th;
                __be32 opt[(TCPOLEN_TSTAMP_ALIGNED >> 2)
#ifdef CONFIG_TCP_MD5SIG
                           + (TCPOLEN_MD5SIG_ALIGNED >> 2)
#endif
                          ];
        } rep;
        struct ip_reply_arg arg;

        memset(&rep.th, 0, sizeof(struct tcphdr));
        memset(&arg, 0, sizeof(arg));

        /* 构造参数和 TCP 头部 */
        arg.iov[0].iov_base = (unsigned char *)&rep;
        arg.iov[0].iov_len  = sizeof(rep.th);
        if (tsecr) {
                rep.opt[0] = htonl((TCPOPT_NOP << 24) | (TCPOPT_NOP << 16) |
                                   (TCPOPT_TIMESTAMP << 8) |
                                   TCPOLEN_TIMESTAMP);
                rep.opt[1] = htonl(tsval);
                rep.opt[2] = htonl(tsecr);
                arg.iov[0].iov_len += TCPOLEN_TSTAMP_ALIGNED;
        }

        /* 交换发送端和接收端 */
        rep.th.dest    = th->source;
        rep.th.source  = th->dest;
        rep.th.doff    = arg.iov[0].iov_len / 4;
        rep.th.seq     = htonl(seq);
        rep.th.ack_seq = htonl(ack);
        rep.th.ack     = 1;
        rep.th.window  = htons(win);

        /* 略去和 MD5 相关的部分代码 */

        /* 设定标志位和校验码 */
        arg.flags = reply_flags;
        arg.csum = csum_tcpudp_nofold(ip_hdr(skb)->daddr,
                                      ip_hdr(skb)->saddr, /* XXX */
                                      arg.iov[0].iov_len, IPPROTO_TCP, 0);
        arg.csumoffset = offsetof(struct tcphdr, check) / 2;
        if (oif)
                arg.bound_dev_if = oif;
        arg.tos = tos;
        /* 调用 IP 层接口将包发出 */
```

```
59          ip_send_unicast_reply(*this_cpu_ptr(net->ipv4.tcp_sk),
60                                skb, &TCP_SKB_CB(skb)->header.h4.opt,
61                                ip_hdr(skb)->saddr, ip_hdr(skb)->daddr,
62                                &arg, arg.iov[0].iov_len);
63
64          TCP_INC_STATS_BH(net, TCP_MIB_OUTSEGS);
65  }
```

至此，四次握手就完成了。

4.1.5 同时关闭

还有一种情况是双方同时发出了 FIN 报文，准备关闭连接。表现在 TCP 的状态图上，就是在发出 FIN 包以后又收到了 FIN 包，因此进入了 CLOSING 状态。该段代码在4.2.2.2 中进行了解析。之后，CLOSING 状态等待接受 ACK，就会进入到下一个状态在`tcp_rcv_state_process`中，处理`TCP_CLOSING`的代码如下：

```
1           case TCP_CLOSING:
2                   if (tp->snd_una == tp->write_seq) {
3                           tcp_time_wait(sk, TCP_TIME_WAIT, 0);
4                           goto discard;
5                   }
6                   break;
```

如果收到的 ACK 没问题则转入`TIME_WAIT`状态，利用 timewait 控制块完成后续的工作。

```
1   switch (sk->sk_state) {
2           case TCP_CLOSE_WAIT:
3           case TCP_CLOSING:
4           case TCP_LAST_ACK:
5                   if (!before(TCP_SKB_CB(skb)->seq, tp->rcv_nxt))
6                           break;
7           case TCP_FIN_WAIT1:
8           case TCP_FIN_WAIT2:
9                   /* RFC 793 says to queue data in these states,
10                   * RFC 1122 says we MUST send a reset.
11                   * BSD 4.4 also does reset.
12                   */
13                  if (sk->sk_shutdown & RCV_SHUTDOWN) {
14                          if (TCP_SKB_CB(skb)->end_seq != TCP_SKB_CB(skb)->seq &&
15                              after(TCP_SKB_CB(skb)->end_seq - th->fin, tp->rcv_nxt)) {
16                                  NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPABORTONDATA);
17                                  tcp_reset(sk);
18                                  return 1;
19                          }
20                  }
21                  /* Fall through */
22          case TCP_ESTABLISHED:
23                  tcp_data_queue(sk, skb);
24                  queued = 1;
25                  break;
26  }
```

如果段中的数据正常，且接口没有关闭，那么就收下数据。否则，就直接忽略掉数据段的数据。

4.1.6 TIME_WAIT

该状态的处理也在tcp_timewait_state_process函数中。紧接在 4.1.3中处理FIN_WAIT2状态的代码之后。此时，说明当前的状态为TIME_WAIT。

```
        /*
         *      Now real TIME-WAIT state.
         *
         *      RFC 1122:
         *      "When a connection is [...] on TIME-WAIT state [...]
         *      [a TCP] MAY accept a new SYN from the remote TCP to
         *      reopen the connection directly, if it:
         *
         *      (1)  assigns its initial sequence number for the new
         *      connection to be larger than the largest sequence
         *      number it used on the previous connection incarnation,
         *      and
         *
         *      (2)  returns to TIME-WAIT state if the SYN turns out
         *      to be an old duplicate".
         */

        if (!paws_reject &&
            (TCP_SKB_CB(skb)->seq == tcptw->tw_rcv_nxt &&
             (TCP_SKB_CB(skb)->seq == TCP_SKB_CB(skb)->end_seq || th->rst))) {
                /* 序号没有回卷，仍在窗口中。 */

                if (th->rst) {
                        /* This is TIME_WAIT assassination, in two flavors.
                         * Oh well... nobody has a sufficient solution to this
                         * protocol bug yet.
                         */
                        if (sysctl_tcp_rfc1337 == 0) {
kill:
                                inet_twsk_deschedule_put(tw);
                                return TCP_TW_SUCCESS;
                        }
                }
                /* 重新激活定时器 */
                inet_twsk_reschedule(tw, TCP_TIMEWAIT_LEN);

                if (tmp_opt.saw_tstamp) {
                        tcptw->tw_ts_recent       = tmp_opt.rcv_tsval;
                        tcptw->tw_ts_recent_stamp = get_seconds();
                }

                inet_twsk_put(tw);
                return TCP_TW_SUCCESS;
        }
```

如果处于TIME_WAIT状态时，受到了 Reset 包，那么，按照 TCP 协议的要求，应当重置连接。但这里就产生了一个问题。本来TIME_WAIT之所以要等待 2MSL 的时间，就是为了避免在网络上滞留的包对新的连接造成影响。但是，此处却可以通过发送 rst 报文强行重置连接。重置意味着该连接会被强行关闭，跳过了 2MSL 阶段。这样就和设立 2MSL 的初衷不符了。具体的讨论见1.3.3。如果启用了 RFC1337，那么就会忽略掉这个 RST 报文。

```
        /* 之后是超出窗口范围的情况。

           All the segments are ACKed immediately.

           The only exception is new SYN. We accept it, if it is
           not old duplicate and we are not in danger to be killed
           by delayed old duplicates. RFC check is that it has
           newer sequence number works at rates <40Mbit/sec.
           However, if paws works, it is reliable AND even more,
           newer sequence number works at rates <40Mbit/sec.
           However, if paws works, it is reliable AND even more,
           we even may relax silly seq space cutoff.

           RED-PEN: we violate main RFC requirement, if this SYN will appear
           old duplicate (i.e. we receive RST in reply to SYN-ACK),
           we must return socket to time-wait state. It is not good,
           but not fatal yet.
         */

        if (th->syn && !th->rst && !th->ack && !paws_reject &&
            (after(TCP_SKB_CB(skb)->seq, tcptw->tw_rcv_nxt) ||
             (tmp_opt.saw_tstamp &&
              (s32)(tcptw->tw_ts_recent - tmp_opt.rcv_tsval) < 0))) {
                /* 如果可以接受该 SYN 请求，那么重新计算 isn 号，并发出 syn。 */
                u32 isn = tcptw->tw_snd_nxt + 65535 + 2;
                if (isn == 0)
                        isn++;
                TCP_SKB_CB(skb)->tcp_tw_isn = isn;
                return TCP_TW_SYN;
        }

        if (paws_reject)
                NET_INC_STATS_BH(twsk_net(tw), LINUX_MIB_PAWSESTABREJECTED);
```

此后，如果收到了序号绕回的包，那么就重置TIME_WAIT定时器，并返回 TCP_TW_ACK。

```
        if (!th->rst) {
                /* In this case we must reset the TIMEWAIT timer.
                 *
                 * If it is ACKless SYN it may be both old duplicate
                 * and new good SYN with random sequence number <rcv_nxt.
                 * Do not reschedule in the last case.
                 */
                if (paws_reject || th->ack)
                        inet_twsk_reschedule(tw, TCP_TIMEWAIT_LEN);

                return tcp_timewait_check_oow_rate_limit(
                        tw, skb, LINUX_MIB_TCPACKSKIPPEDTIMEWAIT);
        }
        inet_twsk_put(tw);
        return TCP_TW_SUCCESS;
}
```

4.2 被动关闭

4.2.1 基本流程

在正常的被动关闭开始时，TCP 控制块目前处于 ESTABLISHED 状态，此时接收到的 TCP 段都由tcp_rcv_established函数来处理，因此 FIN 段必定要做首部预测，当然预测一定不会通过，所以 FIN 段是走**慢速路径**处理的。

在慢速路径中，首先，首先进行 TCP 选项的处理 (假设存在)，然后根据段的序号检测该 FIN 段是不是期望接收的段。如果是，则调用tcp_fin函数进行处理。如果不是，则说明在 TCP 段传输过程中出现了失序，因此将该 FIN 段缓存到乱序队列中，等待它之前的所有 TCP 段都到其后才能作处理。

4.2.2 第一次握手：接收FIN

在这个过程中,服务器段主要接收到了客户端的 FIN 段,并且跳转到了CLOSE_WAIT状态。

4.2.2.1 函数调用关系

4.2.2.2 tcp_fin

此时,首先调用的传输层的函数为tcp_rcv_established,然后走慢速路径由tcp_data_queue函数处理。在处理的过程中，如果 FIN 段时预期接收的段，则调用tcp_fin函数处理，否则，将该段暂存到乱序队列中，等待它之前的 TCP 段到齐之后再做处理，这里我们不说函数tcp_rcv_established与tcp_data_queue了，这些主要是在数据传送阶段介绍的函数。我们直接介绍tcp_fin函数。

```
/*
Location:

    net/ipv4/tcp_input.c

Function:

    Process the FIN bit. This now behaves as it is supposed to work
    and the FIN takes effect when it is validly part of sequence
    space. Not before when we get holes.

    If we are ESTABLISHED, a received fin moves us to CLOSE-WAIT
    (and thence onto LAST-ACK and finally, CLOSE, we never enter
    TIME-WAIT)

    If we are in FINWAIT-1, a received FIN indicates simultaneous
    close and we go into CLOSING (and later onto TIME-WAIT)

    If we are in FINWAIT-2, a received FIN moves us to TIME-WAIT.

Parameters:

    sk: 传输控制块
*/
```

```
25 static void tcp_fin(struct sock *sk)
26 {
27     struct tcp_sock *tp = tcp_sk(sk);
28
29     inet_csk_schedule_ack(sk);
30
31     sk->sk_shutdown |= RCV_SHUTDOWN;
32     sock_set_flag(sk, SOCK_DONE);
```

首先，接收到 FIN 段后需要调度、发送 ACK。

其次，设置了传输控制块的套接口状态为RCV_SHUTDOWN，表示服务器端不允许在接收数据。

最后，设置传输控制块的SOCK_DONE标志，表示 TCP 会话结束。

当然，接收到 FIN 字段的时候，TCP 有可能并不是处于TCP_ESTABLISHED的，这里我们一并介绍了。

```
1     switch (sk->sk_state) {
2         case TCP_SYN_RECV:
3         case TCP_ESTABLISHED:
4             /* Move to CLOSE_WAIT */
5             tcp_set_state(sk, TCP_CLOSE_WAIT);
6             inet_csk(sk)->icsk_ack.pingpong = 1;          //what does it means pingpong?  to do.
7             break;
```

如果 TCP 块处于TCP_SYN_RECV或者TCP_ESTABLISHED状态，接收到 FIN 之后，将状态设置为CLOSE_WAIT, 并确定延时发送 ack。

```
1         case TCP_CLOSE_WAIT:
2         case TCP_CLOSING:
3             /* Received a retransmission of the FIN, do
4              * nothing.
5              */
6             break;
7         case TCP_LAST_ACK:
8             /* RFC793: Remain in the LAST-ACK state. */
9             break;
```

在TCP_CLOSE_WAIT或者TCP_CLOSING状态下收到的 FIN 为重复收到的 FIN，忽略。

在TCP_LAST_ACK状态下正在等待最后的 ACK 字段，故而忽略。

```
1         case TCP_FIN_WAIT1:
2             /* This case occurs when a simultaneous close
3              * happens, we must ack the received FIN and
4              * enter the CLOSING state.
5              */
6             tcp_send_ack(sk);
7             tcp_set_state(sk, TCP_CLOSING);
8             break;
```

显然，只有客户端和服务器端同时关闭的情况下，才会出现这种状态，而且此时双方必须都发送 ACK 字段，并且将状态置为TCP_CLOSING。

```
        case TCP_FIN_WAIT2:
            /* Received a FIN -- send ACK and enter TIME_WAIT. */
            tcp_send_ack(sk);
            tcp_time_wait(sk, TCP_TIME_WAIT, 0);
            break;
        default:
            /* Only TCP_LISTEN and TCP_CLOSE are left, in these
             * cases we should never reach this piece of code.
             */
            pr_err("%s: Impossible, sk->sk_state=%d\n",
                   __func__, sk->sk_state);
            break;
    }
```

在TCP_FIN_WAIT2状态下接收到 FIN 段，根据 TCP 状态转移图应该发送 ACK 响应服务器端。

在其他状态，显然是不能收到 FIN 段的。

```
    /* It _is_ possible, that we have something out-of-order _after_ FIN.
     * Probably, we should reset in this case. For now drop them.
     */
    __skb_queue_purge(&tp->out_of_order_queue);
    if (tcp_is_sack(tp))
        tcp_sack_reset(&tp->rx_opt);
    sk_mem_reclaim(sk);

    if (!sock_flag(sk, SOCK_DEAD)) {
        sk->sk_state_change(sk);

        /* Do not send POLL_HUP for half duplex close. */
        if (sk->sk_shutdown == SHUTDOWN_MASK ||
            sk->sk_state == TCP_CLOSE)
            sk_wake_async(sk, SOCK_WAKE_WAITD, POLL_HUP);
        else
            sk_wake_async(sk, SOCK_WAKE_WAITD, POLL_IN);
    }
}
```

清空接到的乱序队列上的段，再清除有关 SACK 的信息和标志，最后释放已接收队列中的段。

如果此时套接口未处于 DEAD 状态，则唤醒等待该套接口的进程。如果在发送接收方向上都进行了关闭，或者此时传输控制块处于 CLOSE 状态，则唤醒异步等待该套接口的进程，通知它们该连接已经停止，否则通知它们连接可以进行写操作。

CHAPTER 5

TCP 拥塞控制

Contents

5.1 拥塞控制实现

5.1.1 拥塞控制状态机

5.1.1.1 基本转换

```
enum tcp_ca_state {
        TCP_CA_Open = 0,
#define TCPF_CA_Open        (1<<TCP_CA_Open)
        TCP_CA_Disorder = 1,
#define TCPF_CA_Disorder (1<<TCP_CA_Disorder)
        TCP_CA_CWR = 2,
#define TCPF_CA_CWR         (1<<TCP_CA_CWR)
        TCP_CA_Recovery = 3,
#define TCPF_CA_Recovery (1<<TCP_CA_Recovery)
        TCP_CA_Loss = 4
#define TCPF_CA_Loss        (1<<TCP_CA_Loss)
};
```

5.1.1.2 Open 状态

Open 状态是常态，在这种状态下 TCP 发送方通过优化后的快速路径来处理接收 ACK。当一个确认到达时，发送方根据拥塞窗口时小于还是大于慢启动阈值，按慢启动或者拥塞避免来增大拥塞窗口。

5.1.1.3 Disorder 状态

当发送方检测到 DACK(重复确认) 或者 SACK(选择性确认) 时，将转变为 Disorder(无序) 状态。在该状态下，拥塞窗口不做调整，而是每个新到的段出发一个新的数据段的发送。因此，TCP 发送方遵循包守恒原则，该原则规定一个新包只有在一个老的包离开网络后才发送。在实践中该规定的表现类似于 IETF 的传输提议，允许当拥塞窗口较小或是上个传输窗口中有大量数据段丢失时，使用快速重传以更有效地恢复。

5.1.1.4 CWR 状态

TCP 发送方可能从显示拥塞通知、ICMP 源端抑制 (ICMP source quench) 或是本地设备接收到拥塞通知。当收到一个拥塞通知时，发送方并不立刻减小拥塞窗口，而是每隔一个新到的 ACK 减小一个段直到窗口的大小减半为止。发送方在减小拥塞窗口大小的过程中不会有明显的重传，这就处于 CWR(Congestion Window Reduced, 拥塞窗口减小) 状态。CWR 状态可以被 Revcovery 状态或者 Loss 状态中断。进入拥塞窗口减小的函数如下：

```
/*
Location:

        net/ipv4/tcp_input.c

Function:

        Enter CWR state. Disable cwnd undo since congestion is proven with ECN
```

```
9
10  Parameter:
11
12          sk: 传输控制块
13  */
14  void tcp_enter_cwr(struct sock *sk)
15  {
16          struct tcp_sock *tp = tcp_sk(sk);
17          /* 进入 CWR 后就不需要窗口撤消了,
18        因此需要清除拥塞控制的慢启动阈值
19          */
20          tp->prior_ssthresh = 0;
21          /* 可以看出只有 OPen 状态和 Disorder 状态可以转移到该状态 */
22          if (inet_csk(sk)->icsk_ca_state < TCP_CA_CWR) {
23                  /* 湖北人与 CWR 状态后不允许在进行拥塞窗口撤消了 */
24                  tp->undo_marker = 0;
25                  /* 进行相关的初始化 */
26                  tcp_init_cwnd_reduction(sk);
27                  /* 设置状态 */
28                  tcp_set_ca_state(sk, TCP_CA_CWR);
29          }
30  }
```

对于函数tcp_init_cwnd_reduction的调用如下:

```
1   /*
2   Location:
3
4           net/ipv4/tcp_input.c
5
6   Function:
7           The cwnd reduction in CWR and Recovery uses the PRR algorithm in RFC 6937.
8           It computes the number of packets to send (sndcnt) based on packets newly
9           delivered:
10                  1) If the packets in flight is larger than ssthresh, PRR spreads the
11                          cwnd reductions across a full RTT.
12                  2) Otherwise PRR uses packet conservation to send as much as delivered.
13                          But when the retransmits are acked without further losses, PRR
14                          slow starts cwnd up to ssthresh to speed up the recovery.
15
16  parameter:
17
18          sk: 传输控制块
19  */
20  static void tcp_init_cwnd_reduction(struct sock *sk)
21  {
22          struct tcp_sock *tp = tcp_sk(sk);
23
24          /* 记录发送拥塞时的 snd.nxt*/
25          tp->high_seq = tp->snd_nxt;
26          /*snd_nxt at the time of TLP retransmit*/
27          tp->tlp_high_seq = 0;
28          /*snd_cwnd_cnt 表示自从上次调整拥塞窗口到
29           目前为止接收到的总 ACK 段数, 这里设置为 0 是因为
30           刚刚改变拥塞控制算法为 PRR。
31          */
32          tp->snd_cwnd_cnt = 0;
33          /*prior_cwnd means Congestion window at start of Recovery
34                  设置该值为当前拥塞窗口。
```

```
35          */
36          tp->prior_cwnd = tp->snd_cwnd;
37          /*Number of newly delivered packets
38            to receiver in Recovery, 设置为 0
39          */
40          tp->prr_delivered = 0;
41          /*Total number of pkts sent during Recovery*/
42          tp->prr_out = 0;
43          /* 根据给定的拥塞控制算法重新设置拥塞慢启动阈值 */
44          tp->snd_ssthresh = inet_csk(sk)->icsk_ca_ops->ssthresh(sk);
45          /* 设置 TCP_ECN_QUEUE_CWR 标志，标识由于收到显示拥塞通知而进入拥塞状态 */
46          tcp_ecn_queue_cwr(tp);
47  }
```

5.1.1.5 Recovery 状态

当足够多的连续重复 ACK 到达后，发送方重传第一个没有被确认的段，进入 Recovery(恢复) 状态。默认情况下，进入 Recovery 状态的条件是三个连续的重复 ACK，TCP 拥塞控制规范也是这么推荐的。在 Recovery 状态期间，拥塞窗口的大小每隔一个新到的确认而减少一个段，和 CWR 状态类似。这种窗口减小过程终止与拥塞窗口大小等于 ssthresh，即进入 Recovery 状态时，窗口大小的一半。拥塞窗口在恢复期间不增大，发送方重传那些被标记为丢失的段，或者根据包守恒原则在新数据上标记前向传输。发送方保持 Recovery 状态直到所有进入 Recovery 状态时正在发送的数据段都成功地被确认，之后该发送方恢复 OPEN 状态，重传超时有可能中断 Recovery 状态。

5.1.1.6 Loss 状态

当一个 RTO 到期后，发送方进入 Loss 状态。所有正在发送的数据段标记为丢失，拥塞窗口设置为一个段，发送方因此以慢启动算法增大拥塞窗口。Loss 和 Recovery 状态的区别是，Loss 状态下，拥塞窗口在发送方设置为一个段后增大，而 Recovery 状态下，拥塞窗口只能被减小。Loss 状态不能被其他的状态中断，因此，发送方只有在所有 Loss 开始时正在传输的数据都得到成功确认后，才能退到 Open 状态。例如，快速重传不能在 Loss 状态期间被触发，这和 NewReno 规范是一致的。

当接收到的 ACK 的确认已经被之前的 SACK 确认过，这意味着我们记录的 SACK 信息不能呢个反映接收方的实际状态，此时，也会进入 Loss 状态。

调用`tcp_enter_loss`进入 Loss 状态，如下：

```
1   /*
2   Location:
3
4           net/ipv4/tcp_input.c
5
6   Function:
7           Enter Loss state. If we detect SACK reneging(违约), forget all SACK information
8           and reset tags completely, otherwise preserve SACKs. If receiver
9           dropped its ofo?? queue, we will know this due to reneging detection.
10
11  Parameter:
12
```

```
13          sk: 传输控制块
14
15  */
16  void tcp_enter_loss(struct sock *sk)
17  {
18          const struct inet_connection_sock *icsk = inet_csk(sk);
19          struct tcp_sock *tp = tcp_sk(sk);
20          struct sk_buff *skb;
21          bool new_recovery = icsk->icsk_ca_state < TCP_CA_Recovery;
22          bool is_reneg;                          /* is receiver reneging on SACKs? */
23
24          /* Reduce ssthresh if it has not yet been made inside this window. */
25          if (icsk->icsk_ca_state <= TCP_CA_Disorder ||
26              !after(tp->high_seq, tp->snd_una) ||
27              (icsk->icsk_ca_state == TCP_CA_Loss && !icsk->icsk_retransmits)) {
28                  /* 保留当前的阈值 */
29                  tp->prior_ssthresh = tcp_current_ssthresh(sk);
30                  /* 计算新的阈值 */
31                  tp->snd_ssthresh = icsk->icsk_ca_ops->ssthresh(sk);
32                  /* 发送 CA_EVENT_LOSS 拥塞事件给具体拥塞算法模块 */
33                  tcp_ca_event(sk, CA_EVENT_LOSS);
34                  /*
35                          在下面的函数中会做以下两个事情:
36                          tp->undo_marker = tp->snd_una;
37                          /* Retransmission still in flight may cause DSACKs later.
38                                  undo_retrans 在恢复拥塞控制之前可进行撤销的重传段数???
39                                  retrans_out 重传并且还未得到确认的 TCP 段的数目
40                          */
41                          tp->undo_retrans = tp->retrans_out ? : -1;
42                  */
43                  tcp_init_undo(tp);
44          }
45          /* 拥塞窗口大小设置为 1*/
46          tp->snd_cwnd            = 1;
47          /*snd_cwnd_cnt 表示自从上次调整拥塞窗口到
48           目前为止接收到的总 ACK 段数, 自然设置为 0*/
49          tp->snd_cwnd_cnt   = 0;
50          /* 记录最近一次检验拥塞窗口的时间 */
51          tp->snd_cwnd_stamp = tcp_time_stamp;
52          /* 重传并且还未得到确认的 TCP 段的数目设置为零 */
53          tp->retrans_out = 0;
54          /* 丢失的包 */
55          tp->lost_out = 0;
56          /* 查看当前的 tp 里是否由 SACK 选项字段,
57          有的话, 返回 1, 没有的话返回 0
58          根据这一点来判断是否需要重置 tp 中选择确认的包 (sacked_out??? confused) 的个数为 0
59          */
60          if (tcp_is_reno(tp))
61                  tcp_reset_reno_sack(tp);
62
63          skb = tcp_write_queue_head(sk);
64          /* 判断接受者是否认为 SACK 违约??? 怎么理解呢？*/
65          is_reneg = skb && (TCP_SKB_CB(skb)->sacked & TCPCB_SACKED_ACKED);
66          if (is_reneg) {
67                  NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPSACKRENEGING);
68                  tp->sacked_out = 0;
69                  tp->fackets_out = 0;
70          }
71          /* 清除有关重传的记忆变量 */
```

```
72          tcp_clear_all_retrans_hints(tp);
73
74          tcp_for_write_queue(skb, sk) {
75                  if (skb == tcp_send_head(sk))
76                          break;
77
78                  TCP_SKB_CB(skb)->sacked &= (~TCPCB_TAGBITS)|TCPCB_SACKED_ACKED;
79                  if (!(TCP_SKB_CB(skb)->sacked&TCPCB_SACKED_ACKED) || is_reneg) {
80                          /* 清除 ACK 标志 */
81                          TCP_SKB_CB(skb)->sacked &= ~TCPCB_SACKED_ACKED;
82                          /* 添加 LOST 标志 */
83                          TCP_SKB_CB(skb)->sacked |= TCPCB_LOST;
84                          /* 统计丢失段的数量 */
85                          tp->lost_out += tcp_skb_pcount(skb);
86                          /* 重传的最大序号??????*/
87                          tp->retransmit_high = TCP_SKB_CB(skb)->end_seq;
88                  }
89          }
90          /* 确认没有被确认的 TCP 段的数量 left_out*/
91          tcp_verify_left_out(tp);
92
93          /* Timeout in disordered state after receiving substantial DUPACKs
94           * suggests that the degree of reordering is over-estimated.
95           */
96          if (icsk->icsk_ca_state <= TCP_CA_Disorder &&
97              tp->sacked_out >= sysctl_tcp_reordering)
98                  /* 重新设置 reordering*/
99                  tp->reordering = min_t(unsigned int, tp->reordering,
100                                         sysctl_tcp_reordering);
101         /* 设置拥塞状态 */
102         tcp_set_ca_state(sk, TCP_CA_Loss);
103         /* 记录发生拥塞时的 snd.nxt*/
104         tp->high_seq = tp->snd_nxt;
105         /* 设置 ecn_flags, 表示发送方进入拥塞状态 */
106         tcp_ecn_queue_cwr(tp);
107
108         /* F-RTO RFC5682 sec 3.1 step 1: retransmit SND.UNA if no previous
109          * loss recovery is underway except recurring timeout(s) on
110          * the same SND.UNA (sec 3.2). Disable F-RTO on path MTU probing
111          */
112         tp->frto = sysctl_tcp_frto &&
113                    (new_recovery || icsk->icsk_retransmits) &&
114                    !inet_csk(sk)->icsk_mtup.probe_size;
115 }
```

5.1.2 拥塞控制状态的处理及转换

5.1.3 显式拥塞通知 (ECN)

5.2 拥塞控制引擎

在 Linux 中, 实现了多种不同的拥塞控制算法。为了简化拥塞控制算法的编写, Linux 的开发者们实现了一套拥塞控制引擎或者说是框架, 专门用于实现拥塞控制算法。只要实现必要的接口, 即可完成一套拥塞控制算法, 极大地简化了拥塞控制算法的开发工作。

5.2.1 接口

5.2.1.1 tcp_congestion_ops

`struct tcp_congestion_ops`结构体描述了一套拥塞控制算法所需要支持的操作。其原型如下：

```
struct tcp_congestion_ops {
        struct list_head        list;
        u32 key; /* 算法名称的哈希值 */
        u32 flags;

        /* 初始化私有数据 (可选) */
        void (*init)(struct sock *sk);
        /* 释放私有数据  (可选) */
        void (*release)(struct sock *sk);

        /* 返回 ssthresh (必须实现) */
        u32 (*ssthresh)(struct sock *sk);
        /* 计算新的拥塞窗口 (必须实现) */
        void (*cong_avoid)(struct sock *sk, u32 ack, u32 acked);
        /* 在改变 ca_state 前会被调用 (可选) */
        void (*set_state)(struct sock *sk, u8 new_state);
        /* 处理拥塞窗口相关的事件 (可选) */
        void (*cwnd_event)(struct sock *sk, enum tcp_ca_event ev);
        /* 处理 ACK 包到达事件 (可选) */
        void (*in_ack_event)(struct sock *sk, u32 flags);
        /* 用于撤销“缩小拥塞窗口” (可选) */
        u32  (*undo_cwnd)(struct sock *sk);
        /* 有段被确认时会调用此函数 (可选) */
        void (*pkts_acked)(struct sock *sk, u32 num_acked, s32 rtt_us);
        /* 为 inet_diag 准备的获取信息的接口 (可选) */
        size_t (*get_info)(struct sock *sk, u32 ext, int *attr,
                           union tcp_cc_info *info);

        /* 拥塞控制算法的名称 */
        char            name[TCP_CA_NAME_MAX];
        struct module   *owner;
};
```

5.2.1.2 拥塞控制事件

为了将拥塞控制算法相关的部分抽取处理，Linux 的开发者们采用了事件机制，即在发生和拥塞控制相关的事件后，调用拥塞控制算法中的事件处理函数，以通知拥塞控制模块，具体发生了什么。而作为实现拥塞控制算法的开发者，则无需关心事件是如何发生的，以及相关的实现，只要专注于事件所对应的需要执行的算法即可。

当发生相关事件是会调用`cwnd_event`函数。该函数会传入一个枚举值作为参数，代表具体发生的事件。该枚举值的定义如下：

```
enum tcp_ca_event {
        /* 首次传输（无已发出但还未确认的包） */
        CA_EVENT_TX_START,
        /* 拥塞窗口重启 */
        CA_EVENT_CWND_RESTART,
        /* 拥塞恢复结束 */
```

```
        CA_EVENT_COMPLETE_CWR,
        /* loss 超时 */
        CA_EVENT_LOSS,
        /* ECT 被设置了，但 CE 没有被置位 */
        CA_EVENT_ECN_NO_CE,
        /* 收到了设置了 CE 位的 IP 报文 */
        CA_EVENT_ECN_IS_CE,
        /* 延迟确认已被发送 */
        CA_EVENT_DELAYED_ACK,
        CA_EVENT_NON_DELAYED_ACK,
};
```

当收到了 ACK 包时，会调用`in_ack_event()`。此时也会传递一些信息给拥塞控制算法。相关的定义如下：

```
enum tcp_ca_ack_event_flags {
        CA_ACK_SLOWPATH         = (1 << 0),     /* 在慢速路径中处理 */
        CA_ACK_WIN_UPDATE       = (1 << 1),     /* 该 ACK 更新了窗口大小 */
        CA_ACK_ECE              = (1 << 2),     /* ECE 位被设置了 */
};
```

5.2.1.3 `tcp_register_congestion_control()`

该函数用于注册一个新的拥塞控制算法。

```
int tcp_register_congestion_control(struct tcp_congestion_ops *ca)
{
        int ret = 0;

        /* 所有的算法都必须实现 ssthresh 和 cong_avoid ops */
        if (!ca->ssthresh || !ca->cong_avoid) {
                pr_err("%s does not implement required ops\n", ca->name);
                /* 如果没实现，则返回错误 */
                return -EINVAL;
        }

        /* 计算算法名称的哈希值，加快比对速度。 */
        ca->key = jhash(ca->name, sizeof(ca->name), strlen(ca->name));

        spin_lock(&tcp_cong_list_lock);
        if (ca->key == TCP_CA_UNSPEC || tcp_ca_find_key(ca->key)) {
                /* 如果已经注册被注册过了，或者恰巧 hash 值重了 (极低概率)，
                 * 那么返回错误值。
                 */
                pr_notice("%s already registered or non-unique key\n",
                          ca->name);
                ret = -EEXIST;
        } else {
                /* 将算法添加到链表中 */
                list_add_tail_rcu(&ca->list, &tcp_cong_list);
                pr_debug("%s registered\n", ca->name);
        }
        spin_unlock(&tcp_cong_list_lock);

        return ret;
}
```

其中，`tcp_ca_find_key`函数通过哈希值来查找名称。jash 是一种久经考验的性能极佳

的哈希算法。据称，其计算速度和产生的分布都很漂亮。这里计算哈希值正是使用了这种哈希算法。早些版本的内核查找拥塞控制算法，是通过名字直接查找的。

```
/* Simple linear search, don't expect many entries! */
static struct tcp_congestion_ops *tcp_ca_find(const char *name)
{
        struct tcp_congestion_ops *e;

        list_for_each_entry_rcu(e, &tcp_cong_list, list) {
                if (strcmp(e->name, name) == 0)
                        return e;
        }

        return NULL;
}
```

可以看到，每次查找都要对比字符串，效率较低。这里为了加快查找速度，对名字进行了哈希，并通过哈希值的比对来进行查找。

```
/* Simple linear search, not much in here. */
struct tcp_congestion_ops *tcp_ca_find_key(u32 key)
{
        struct tcp_congestion_ops *e;

        list_for_each_entry_rcu(e, &tcp_cong_list, list) {
                if (e->key == key)
                        return e;
        }

        return NULL;
}
```

一般情况下，额外的拥塞控制算法都作为单独的模块实现。在模块初始化时，调用 tcp_register_congestio 数来进行注册，之后，即可使用新的拥塞控制算法。

5.2.1.4 tcp_unregister_congestion_control()

与注册相对应的，自然有注销一个拥塞控制算法的方法。tcp_unregister_congestion_control用于撤销一个拥塞控制算法。其实现如下：

```
void tcp_unregister_congestion_control(struct tcp_congestion_ops *ca)
{
        spin_lock(&tcp_cong_list_lock);
        /* 删除该拥塞控制算法 */
        list_del_rcu(&ca->list);
        spin_unlock(&tcp_cong_list_lock);

        /* Wait for outstanding readers to complete before the
         * module gets removed entirely.
         *
         * A try_module_get() should fail by now as our module is
         * in "going" state since no refs are held anymore and
         * module_exit() handler being called.
         */
```

```
15          synchronize_rcu();
16  }
```

5.2.2 CUBIC 拥塞控制算法

Cubic 算法是 Linux 中默认的拥塞控制算法。

5.2.2.1 模块定义

在 Linux 中，拥塞控制算法是作为模块单独实现的。Cubic 也不例外。首先，定义了模块的基本信息

```
1  module_init(cubictcp_register);
2  module_exit(cubictcp_unregister);
3
4  MODULE_AUTHOR("Sangtae Ha, Stephen Hemminger");
5  MODULE_LICENSE("GPL");
6  MODULE_DESCRIPTION("CUBIC TCP");
7  MODULE_VERSION("2.3");
```

这部分定义包括了模块作者、所使用的许可证、版本等等。其中，最重要的是定义了模块的初始化函数，和模块被卸载时所要调用的函数。初始化函数如下：

```
1  static int __init cubictcp_register(void)
2  {
3          BUILD_BUG_ON(sizeof(struct bictcp) > ICSK_CA_PRIV_SIZE);
4
5          /* 预先计算缩放因子（此时假定 SRTT 为 100ms） */
6
7          beta_scale = 8*(BICTCP_BETA_SCALE+beta) / 3
8                  / (BICTCP_BETA_SCALE - beta);
9
10         cube_rtt_scale = (bic_scale * 10);      /* 1024*c/rtt */
11
12         /* 计算公式 (wmax-cwnd) = c/rtt * K^3 中的 K
13          * 可得 K = cubic_root( (wmax-cwnd)*rtt/c )
14          * 注意，这里 K 的单位是 bictcp_HZ=2^10，  而不是 HZ
15          */
16
17         /* 1/c * 2^2*bictcp_HZ * srtt */
18         cube_factor = 1ull << (10+3*BICTCP_HZ); /* 2^40 */
19
20         /* divide by bic_scale and by constant Srtt (100ms) */
21         do_div(cube_factor, bic_scale * 10);
22
23         return tcp_register_congestion_control(&cubictcp);
24 }
```

在初始化时，根据预先设定好的参数，按照 Cubic 算法的公式计算好`cube_factor`。之后，调用`tcp_register_congestion_control`函数注册 Cubic 算法。Cubic 算法所实现的操作如下：

```
static struct tcp_congestion_ops cubictcp __read_mostly = {
        .init           = bictcp_init,
        .ssthresh       = bictcp_recalc_ssthresh,
        .cong_avoid     = bictcp_cong_avoid,
        .set_state      = bictcp_state,
        .undo_cwnd      = bictcp_undo_cwnd,
        .cwnd_event     = bictcp_cwnd_event,
        .pkts_acked     = bictcp_acked,
        .owner          = THIS_MODULE,
        .name           = "cubic",
};
```

根据这里定义好的操作,我们就可以去理解对应的实现。在模块被卸载时,将调用cubictcp_unregister函数。该函数只有一个职责——将 Cubic 算法从拥塞控制引擎中注销掉。

```
static void __exit cubictcp_unregister(void)
{
        tcp_unregister_congestion_control(&cubictcp);
}
```

5.2.2.2 基本参数及初始化

CUBIC 的所需的基本参数定义如下:

```
/* BIC TCP Parameters */
struct bictcp {
        u32     cnt;            /* 每次 cwnd 增长 1/cnt 的比例 */
        u32     last_max_cwnd;  /* snd_cwnd 之前的最大值 */
        u32     loss_cwnd;      /* 最近一次发生丢失的时候的拥塞窗口 */
        u32     last_cwnd;      /* 最近的 snd_cwnd */
        u32     last_time;      /* 更新 last_cwnd 的时间 */
        u32     bic_origin_point;/* bic 函数的初始点 */
        u32     bic_K;          /* 从当前一轮开始到初始点的时间 */
        u32     delay_min;      /* 最小延迟 (msec << 3) */
        u32     epoch_start;    /* 一轮的开始 */
        u32     ack_cnt;        /* ack 的数量 */
        u32     tcp_cwnd;       /* estimated tcp cwnd */
        u16     unused;
        u8      sample_cnt;     /* 用于决定 curr_rtt 的样本数 */
        u8      found;          /* 是否找到了退出点? */
        u32     round_start;    /* beginning of each round */
        u32     end_seq;        /* end_seq of the round */
        u32     last_ack;       /* last time when the ACK spacing is close */
        u32     curr_rtt;       /* the minimum rtt of current round */
};
```

初始化时，首先重置了 CUBIC 所需的参数，之后，将loss_cwnd设置为 0。因为此时尚未发送任何丢失，所以初始化为 0。最后根据是否启用 hystart 机制来决定是否进行相应的初始化。最后，如果设置了initial_ssthresh，那么就用该值作为初始的snd_ssthresh。

```
static void bictcp_init(struct sock *sk)
{
        struct bictcp *ca = inet_csk_ca(sk);

        bictcp_reset(ca);
```

```
6           ca->loss_cwnd = 0;
7
8           if (hystart)
9                   bictcp_hystart_reset(sk);
10
11          if (!hystart && initial_ssthresh)
12                  tcp_sk(sk)->snd_ssthresh = initial_ssthresh;
13  }
```

其中，bictcp_reset函数将必要的参数都初始化为 0 了。

```
1   static inline void bictcp_reset(struct bictcp *ca)
2   {
3           ca->cnt = 0;
4           ca->last_max_cwnd = 0;
5           ca->last_cwnd = 0;
6           ca->last_time = 0;
7           ca->bic_origin_point = 0;
8           ca->bic_K = 0;
9           ca->delay_min = 0;
10          ca->epoch_start = 0;
11          ca->ack_cnt = 0;
12          ca->tcp_cwnd = 0;
13          ca->found = 0;
14  }
```

5.2.2.3 ssthresh 的计算

门限值的计算过程在bictcp_recalc_ssthresh函数中实现。

```
1   static u32 bictcp_recalc_ssthresh(struct sock *sk)
2   {
3           const struct tcp_sock *tp = tcp_sk(sk);
4           struct bictcp *ca = inet_csk_ca(sk);
5
6           ca->epoch_start = 0;    /* end of epoch */
7
8           /* Wmax and fast convergence */
9           if (tp->snd_cwnd < ca->last_max_cwnd && fast_convergence)
10                  ca->last_max_cwnd = (tp->snd_cwnd * (BICTCP_BETA_SCALE + beta))
11                          / (2 * BICTCP_BETA_SCALE);
12          else
13                  ca->last_max_cwnd = tp->snd_cwnd;
14
15          ca->loss_cwnd = tp->snd_cwnd;
16
17          return max((tp->snd_cwnd * beta) / BICTCP_BETA_SCALE, 2U);
18  }
```

这里涉及到了 Fast Convergence 机制。该机制的存在是为了加快 CUBIC 算法的收敛速度。在网络中，一个新的流的加入，会使得旧的流让出一定的带宽，以便给新的流让出一定的增长空间。为了增加旧的流释放的带宽量，CUBIC 的作者引入了 Fast Convergence 机制。每次发生丢包后，会对比此次丢包时拥塞窗口的大小和之前的拥塞窗口大小。如果小于了之前拥塞窗口的最大值，那么就说明可能是有新的流加入了。此时，就多留出一些带宽给新的流使用，以使得网络尽快收敛到稳定状态。

5.2.2.4 慢启动和拥塞避免

处于拥塞避免状态时，计算拥塞窗口的函数为bictcp_cong_avoid。

```
static void bictcp_cong_avoid(struct sock *sk, u32 ack, u32 acked)
{
        struct tcp_sock *tp = tcp_sk(sk);
        struct bictcp *ca = inet_csk_ca(sk);

        if (!tcp_is_cwnd_limited(sk))
                return;

        /* 当 tp->snd_cwnd < tp->snd_ssthresh 时,
         * 让拥塞窗口大小正好等于 ssthresh 的大小。并据此计算 acked 的大小。
         */
        if (tcp_in_slow_start(tp)) {
                if (hystart && after(ack, ca->end_seq))
                        bictcp_hystart_reset(sk);
                acked = tcp_slow_start(tp, acked);
                if (!acked)
                        return;
        }
        bictcp_update(ca, tp->snd_cwnd, acked);
        tcp_cong_avoid_ai(tp, ca->cnt, acked);
}
```

这里不妨举个例子。如果 ssthresh 的值为 6，初始 cwnd 为 1。那么按照 TCP 的标准，拥塞窗口大小的变化应当为 1,2,4,6 而不是 1,2,4,8。当处于慢启动的状态时，acked 的数目完全由慢启动决定。慢启动部分的代码如下：

```
u32 tcp_slow_start(struct tcp_sock *tp, u32 acked)
{
        /* 新的拥塞窗口的大小等于 ssthresh 和 cwnd 中较小的那一个 */
        u32 cwnd = min(tp->snd_cwnd + acked, tp->snd_ssthresh);

        /* 如果新的拥塞窗口小于 ssthresh, 则 acked=0。
         * 否则 acked 为超过 ssthresh 部分的数目。
         */
        acked -= cwnd - tp->snd_cwnd;
        tp->snd_cwnd = min(cwnd, tp->snd_cwnd_clamp);

        return acked;
}
```

也就是说，如果满足 $cwnd < ssthresh$，那么，bictcp_cong_avoid就表现为慢启动。否则，就表现为拥塞避免。拥塞避免状态下，调用bictcp_update来更新拥塞窗口的值。

```
static inline void bictcp_update(struct bictcp *ca, u32 cwnd, u32 acked)
{
        u32 delta, bic_target, max_cnt;
        u64 offs, t;

        ca->ack_cnt += acked;   /* 统计 ACKed packets 的数目 */

        if (ca->last_cwnd == cwnd &&
            (s32)(tcp_time_stamp - ca->last_time) <= HZ / 32)
                return;

```

```
12          /* CUBIC 函数每个时间单位内最多更新一次 ca->cnt 的值。
13           * 每一次发生 cwnd 减小事件，ca->epoch_start 会被设置为 0.
14           * 这会强制重新计算 ca->cnt。
15           */
16          if (ca->epoch_start && tcp_time_stamp == ca->last_time)
17                  goto tcp_friendliness;
18
19          ca->last_cwnd = cwnd;
20          ca->last_time = tcp_time_stamp;
21
22          if (ca->epoch_start == 0) {
23                  ca->epoch_start = tcp_time_stamp;       /* 记录起始时间 */
24                  ca->ack_cnt = acked;                    /* 开始计数 */
25                  ca->tcp_cwnd = cwnd;                    /* 同步 cubic 的 cwnd 值 */
26
27                  if (ca->last_max_cwnd <= cwnd) {
28                          ca->bic_K = 0;
29                          ca->bic_origin_point = cwnd;
30                  } else {
31                          /* 根据公式计算新的 K 值
32                           * (wmax-cwnd) * (srtt>>3 / HZ) / c * 2^(3*bictcp_HZ)
33                           */
34                          ca->bic_K = cubic_root(cube_factor
35                                                 * (ca->last_max_cwnd - cwnd));
36                          ca->bic_origin_point = ca->last_max_cwnd;
37                  }
38          }
39
40          /* cubic function - calc*/
41          /* calculate c * time^3 / rtt,
42           *  while considering overflow in calculation of time^3
43           * (so time^3 is done by using 64 bit)
44           * and without the support of division of 64bit numbers
45           * (so all divisions are done by using 32 bit)
46           *  also NOTE the unit of those veriables
47           *        time  = (t - K) / 2^bictcp_HZ
48           *        c = bic_scale >> 10
49           * rtt  = (srtt >> 3) / HZ
50           * !!! The following code does not have overflow problems,
51           * if the cwnd < 1 million packets !!!
52           */
53
54          t = (s32)(tcp_time_stamp - ca->epoch_start);
55          t += msecs_to_jiffies(ca->delay_min >> 3);
56          /* change the unit from HZ to bictcp_HZ */
57          t <<= BICTCP_HZ;
58          do_div(t, HZ);
59
60          if (t < ca->bic_K)              /* t - K */
61                  offs = ca->bic_K - t;
62          else
63                  offs = t - ca->bic_K;
64
65          /* c/rtt * (t-K)^3 */
66          delta = (cube_rtt_scale * offs * offs * offs) >> (10+3*BICTCP_HZ);
67          if (t < ca->bic_K)                              /* below origin*/
68                  bic_target = ca->bic_origin_point - delta;
69          else                                            /* above origin*/
```

```
70                  bic_target = ca->bic_origin_point + delta;
71
72          /* 根据 cubic 函数计算出来的目标拥塞窗口值和当前拥塞窗口值，计算 cnt 的大小。*/
73          if (bic_target > cwnd) {
74                  ca->cnt = cwnd / (bic_target - cwnd);
75          } else {
76                  ca->cnt = 100 * cwnd;              /* 只增长一小点 */
77          }
78
79          /*
80           * The initial growth of cubic function may be too conservative
81           * when the available bandwidth is still unknown.
82           */
83          if (ca->last_max_cwnd == 0 && ca->cnt > 20)
84                  ca->cnt = 20;   /* increase cwnd 5% per RTT */
85
86  tcp_friendliness:
87          /* TCP 友好性 */
88          if (tcp_friendliness) {
89          u32 scale = beta_scale;
90
91                  /* 推算在传统的 AIMD 算法下，TCP 拥塞窗口的大小 */
92                  delta = (cwnd * scale) >> 3;
93                  while (ca->ack_cnt > delta) {           /* update tcp cwnd */
94                          ca->ack_cnt -= delta;
95                          ca->tcp_cwnd++;
96                  }
97
98                  /* 如果 TCP 的算法快于 CUBIC，那么就增长到 TCP 算法的水平 */
99                  if (ca->tcp_cwnd > cwnd) {
100                         delta = ca->tcp_cwnd - cwnd;
101                         max_cnt = cwnd / delta;
102                         if (ca->cnt > max_cnt)
103                                 ca->cnt = max_cnt;
104                 }
105         }
106
107         /* 控制增长速率不高于每个 rtt 增长为原来的 1.5 倍 */
108         ca->cnt = max(ca->cnt, 2U);
109 }
```

在更新完窗口大小以后，CUBIC 模块没有直接改变窗口值，而是通过调用 `tcp_cong_avoid_ai` 来改变窗口大小的。这个函数原本只是单纯地每次将窗口大小增加一定的值。但是在经历了一系列的修正后，变得较为难懂了。

```
1   void tcp_cong_avoid_ai(struct tcp_sock *tp, u32 w, u32 acked)
2   {
3           /* 这里做了一个奇怪的小补丁，用于解决这样一种情况：
4            * 如果 w 很大，那么，snd_cwnd_cnt 可能会积累为一个很大的值。
5            * 此后，w 由于种种原因突然被缩小了很多。那么下面计算处理的 delta 就会很大。
6            * 这可能导致流量的爆发。为了避免这种情况，这里提前增加了一个特判。
7            */
8           if (tp->snd_cwnd_cnt >= w) {
9                   tp->snd_cwnd_cnt = 0;
10                  tp->snd_cwnd++;
11          }
12
13          /* 累计被确认的包的数目 */
```

```
14          tp->snd_cwnd_cnt += acked;
15          if (tp->snd_cwnd_cnt >= w) {
16                  /* 窗口增大的大小应当为被确认的包的数目除以当前窗口大小。
17                   * 以往都是直接加一，但直接加一并不是正确的加法增加（AI）的实现。
18                   * 例如，w 为 10，acked 为 20 时，应当增加 20/10=2，而不是 1。
19                   */
20                  u32 delta = tp->snd_cwnd_cnt / w;
21
22                  tp->snd_cwnd_cnt -= delta * w;
23                  tp->snd_cwnd += delta;
24          }
25          tp->snd_cwnd = min(tp->snd_cwnd, tp->snd_cwnd_clamp);
26  }
```

5.2.2.5 hystart

CHAPTER 6

非核心函数分析

Contents

6.1 SKB

6.2 Inet

6.2.1 inet_hash_connect && __inet_hash_connect

6.2.1.1 inet_hash_connect

```c
/*
 * Bind a port for a connect operation and hash it.
 */
int inet_hash_connect(struct inet_timewait_death_row *death_row,
              struct sock *sk)
{
    u32 port_offset = 0;

    if (!inet_sk(sk)->inet_num)
        port_offset = inet_sk_port_offset(sk);
    return __inet_hash_connect(death_row, sk, port_offset,
                   __inet_check_established);
}
```

6.2.1.2 __inet_hash_connect

```c
int __inet_hash_connect(struct inet_timewait_death_row *death_row,
        struct sock *sk, u32 port_offset,
        int (*check_established)(struct inet_timewait_death_row *,
            struct sock *, __u16, struct inet_timewait_sock **))
{
    struct inet_hashinfo *hinfo = death_row->hashinfo;
    const unsigned short snum = inet_sk(sk)->inet_num;
    struct inet_bind_hashbucket *head;
    struct inet_bind_bucket *tb;
    int ret;
    struct net *net = sock_net(sk);

    if (!snum) {
        int i, remaining, low, high, port;
        static u32 hint;
        u32 offset = hint + port_offset;
        struct inet_timewait_sock *tw = NULL;

        inet_get_local_port_range(net, &low, &high);
        remaining = (high - low) + 1;

        /* By starting with offset being an even number,
         * we tend to leave about 50% of ports for other uses,
         * like bind(0).
         */
        offset &= ~1;

        local_bh_disable();
        for (i = 0; i < remaining; i++) {
            port = low + (i + offset) % remaining;
            if (inet_is_local_reserved_port(net, port))
                continue;
            head = &hinfo->bhash[inet_bhashfn(net, port,
                    hinfo->bhash_size)];
            spin_lock(&head->lock);

            /* Does not bother with rcv_saddr checks,
             * because the established check is already
             * unique enough.
             */
            inet_bind_bucket_for_each(tb, &head->chain) {
```

```
                if (net_eq(ib_net(tb), net) &&
                    tb->port == port) {
                    if (tb->fastreuse >= 0 ||
                        tb->fastreuseport >= 0)
                        goto next_port;
                    WARN_ON(hlist_empty(&tb->owners));
                    if (!check_established(death_row, sk,
                                port, &tw))
                        goto ok;
                    goto next_port;
                }
            }

            tb = inet_bind_bucket_create(hinfo->bind_bucket_cachep,
                    net, head, port);
            if (!tb) {
                spin_unlock(&head->lock);
                break;
            }
            tb->fastreuse = -1;
            tb->fastreuseport = -1;
            goto ok;

        next_port:
            spin_unlock(&head->lock);
        }
        local_bh_enable();

        return -EADDRNOTAVAIL;

ok:
        hint += (i + 2) & ~1;

        /* Head lock still held and bh's disabled */
        inet_bind_hash(sk, tb, port);
        if (sk_unhashed(sk)) {
            inet_sk(sk)->inet_sport = htons(port);
            inet_ehash_nolisten(sk, (struct sock *)tw);
        }
        if (tw)
            inet_twsk_bind_unhash(tw, hinfo);
        spin_unlock(&head->lock);

        if (tw)
            inet_twsk_deschedule_put(tw);

        ret = 0;
        goto out;
    }

    head = &hinfo->bhash[inet_bhashfn(net, snum, hinfo->bhash_size)];
    tb  = inet_csk(sk)->icsk_bind_hash;
    spin_lock_bh(&head->lock);
    if (sk_head(&tb->owners) == sk && !sk->sk_bind_node.next) {
        inet_ehash_nolisten(sk, NULL);
        spin_unlock_bh(&head->lock);
        return 0;
    } else {
```

```
100             spin_unlock(&head->lock);
101             /* No definite answer... Walk to established hash table */
102             ret = check_established(death_row, sk, snum, NULL);
103     out:
104             local_bh_enable();
105             return ret;
106         }
107     }
```

6.2.2 inet_twsk_put

该函数用于释放inet_timewait_sock结构体。

```
1   void inet_twsk_put(struct inet_timewait_sock *tw)
2   {
3           /* 减小引用计数，如果计数为 0，则释放它 */
4           if (atomic_dec_and_test(&tw->tw_refcnt))
5                   inet_twsk_free(tw);
6   }
```

6.3 TCP 层

TCP_INC_STATS_BH

rcu_read_unlock 出现在tcp_make_synack中于 MD5 相关的部分。

net_xmit_eval 定时器??

6.3.1 TCP 相关参数

tcp_syn_retries INTEGER, 默认值是 5 对于一个新建连接，内核要发送多少个 SYN 连接请求才决定放弃。不应该大于 255，默认值是 5，对应于 180 秒左右时间。(对于大负载而物理通信良好的网络而言, 这个值偏高, 可修改为 2. 这个值仅仅是针对对外的连接, 对进来的连接, 是由 tcp_retries1 决定的)

tcp_synack_retries INTEGER, 默认值是 5 对于远端的连接请求 SYN，内核会发送 SYN + ACK 数据报，以确认收到上一个 SYN 连接请求包。这是所谓的三次握手 (threeway handshake) 机制的第二个步骤。这里决定内核在放弃连接之前所送出的 SYN+ACK 数目。不应该大于 255，默认值是 5，对应于 180 秒左右时间。(可以根据上面的 tcp_syn_retries 来决定这个值)

tcp_keepalive_time INTEGER, 默认值是 7200(2 小时) 当 keepalive 打开的情况下，TCP 发送 keepalive 消息的频率。(由于目前网络攻击等因素, 造成了利用这个进行的攻击很频繁, 曾经也有 cu 的朋友提到过, 说如果 2 边建立了连接, 然后不发送任何数据或者 rst/fin 消息, 那么持续的时间是不是就是 2 小时, 空连接攻击?tcp_keepalive_time 就是预防此情形的. 我个人在做 nat 服务的时候的修改值为 1800 秒)

tcp_keepalive_probes INTEGER, 默认值是 9 TCP 发送 keepalive 探测以确定该连接已经断开的次数。(注意: 保持连接仅在 SO_KEEPALIVE 套接字选项被打开是才发送. 次数默认不需要修改, 当然根据情形也可以适当地缩短此值. 设置为 5 比较合适)

tcp_keepalive_intvl INTEGER, 默认值为 75 探测消息发送的频率，乘以 tcp_keepalive_probes 就得到对于从开始探测以来没有响应的连接杀除的时间。默认值为 75 秒，也就是没有活动的连接将在大约 11 分钟以后将被丢弃。(对于普通应用来说, 这个值有一些偏大, 可以根据需要改小. 特别是 web 类服务器需要改小该值,15 是个比较合适的值)

tcp_retries1 INTEGER, 默认值是 3 放弃回应一个 TCP 连接请求前，需要进行多少次重试。RFC 规定最低的数值是 3，这也是默认值，根据 RTO 的值大约在 3 秒 - 8 分钟之间。(注意: 这个值同时还决定进入的 syn 连接)

tcp_retries2 INTEGER, 默认值为 15 在丢弃激活 (已建立通讯状况) 的 TCP 连接之前，需要进行多少次重试。默认值为 15, 根据 RTO 的值来决定, 相当于 13-30 分钟 (RFC1122 规定，必须大于 100 秒).(这个值根据目前的网络设置, 可以适当地改小, 我的网络内修改为了 5)

tcp_orphan_retries INTEGER, 默认值是 7 在近端丢弃 TCP 连接之前，要进行多少次重试。默认值是 7 个，相当于 50 秒 - 16 分钟，视 RTO 而定。如果您的系统是负载很大的 web 服务器，那么也许需要降低该值，这类 sockets 可能会耗费大量的资源。另外参的考 tcp_max_orphans 。(事实上做 NAT 的时候, 降低该值也是好处显著的, 我本人的网络环境中降低该值为 3)

tcp_fin_timeout INTEGER, 默认值是 60 对于本端断开的 socket 连接，TCP 保持在 FIN-WAIT-2 状态的时间。对方可能会断开连接或一直不结束连接或不可预料的进程死亡。默认值为 60 秒。过去在 2.2 版本的内核中是 180 秒。您可以设置该值，但需要注意，如果您的机器为负载很重的 web 服务器，您可能要冒内存被大量无效数据报填满的风险，FIN-WAIT-2 sockets 的危险性低于 FIN-WAIT-1 ，因为它们最多只吃 1.5K 的内存，但是它们存在时间更长。另外参考 tcp_max_orphans。(事实上做 NAT 的时候, 降低该值也是好处显著的, 我本人的网络环境中降低该值为 30)

tcp_max_tw_buckets INTEGER, 默认值是 180000 系统在同时所处理的最大 timewait sockets 数目。如果超过此数的话，time-wait socket 会被立即砍除并且显示警告信息。之所以要设定这个限制，纯粹为了抵御那些简单的 DoS 攻击，千万不要人为的降低这个限制，不过，如果网络条件需要比默认值更多，则可以提高它 (或许还要增加内存)。(事实上做 NAT 的时候最好可以适当地增加该值)

tcp_tw_recycle BOOLEAN, 默认值是 0 打开快速 TIME-WAIT sockets 回收。除非得到技术专家的建议或要求，请不要随意修改这个值。(做 NAT 的时候，建议打开它)

tcp_tw_reuse BOOLEAN, 默认值是 0 该文件表示是否允许重新应用处于 TIME-WAIT 状态的 socket 用于新的 TCP 连接 (这个对快速重启动某些服务, 而启动后提示端口已经被使用的情形非常有帮助)

tcp_max_orphans INTEGER, 缺省值是 8192 系统所能处理不属于任何进程的 TCP sockets 最大数量。假如超过这个数量，那么不属于任何进程的连接会被立即 reset，并同时显示警告信息。之所以要设定这个限制，纯粹为了抵御那些简单的 DoS 攻击，千万不

要依赖这个或是人为的降低这个限制 (这个值 Redhat AS 版本中设置为 32768, 但是很多防火墙修改的时候, 建议该值修改为 2000)

cp_abort_on_overflow BOOLEAN, 缺省值是 0 当守护进程太忙而不能接受新的连接, 就象对方发送 reset 消息, 默认值是 false。这意味着当溢出的原因是因为一个偶然的猝发, 那么连接将恢复状态。只有在你确信守护进程真的不能完成连接请求时才打开该选项, 该选项会影响客户的使用。(对待已经满载的 sendmail,apache 这类服务的时候, 这个可以很快让客户端终止连接, 可以给予服务程序处理已有连接的缓冲机会, 所以很多防火墙上推荐打开它)

tcp_syncookies BOOLEAN, 默认值是 0 只有在内核编译时选择了 CONFIG_SYNCOOKIES 时才会发生作用。当出现 syn 等候队列出现溢出时象对方发送 syncookies。目的是为了防止 syn flood 攻击。注意: 该选项千万不能用于那些没有收到攻击的高负载服务器, 如果在日志中出现 synflood 消息, 但是调查发现没有收到 synflood 攻击, 而是合法用户的连接负载过高的原因, 你应该调整其它参数来提高服务器性能。参考: tcp_max_syn_backlog、tcp_synack_retries、tcp_abort_on_overflow。syncookie 严重的违背 TCP 协议, 不允许使用 TCP 扩展, 可能对某些服务导致严重的性能影响 (如 SMTP 转发)。(注意, 该实现与 BSD 上面使用的 tcp proxy 一样, 是违反了 RFC 中关于 tcp 连接的三次握手实现的, 但是对于防御 syn-flood 的确很有用.)

tcp_stdurg BOOLEAN, 默认值为 0 使用 TCP urg pointer 字段中的主机请求解释功能。大部份的主机都使用老旧的 BSD 解释, 因此如果您在 Linux 打开它，或会导致不能和它们正确沟通。

tcp_max_syn_backlog INTEGER, 对于那些依然还未获得客户端确认的连接请求，需要保存在队列中最大数目。对于超过 128Mb 内存的系统，默认值是 1024 ，低于 128Mb 的则为 128。如果服务器经常出现过载，可以尝试增加这个数字。警告！假如您将此值设为大于 1024，最好修改 include/net/tcp.h 里面的 TCP_SYNQ_HSIZE ，以保持 TCP_SYNQ_HSIZE*16<=tcp_max_syn_backlog ，并且编进核心之内。(SYN Flood 攻击利用 TCP 协议散布握手的缺陷, 伪造虚假源 IP 地址发送大量 TCP-SYN 半打开连接到目标系统, 最终导致目标系统 Socket 队列资源耗尽而无法接受新的连接。为了应付这种攻击, 现代 Unix 系统中普遍采用多连接队列处理的方式来缓冲 (而不是解决) 这种攻击, 是用一个基本队列处理正常的完全连接应用 (Connect() 和 Accept()), 是用另一个队列单独存放半打开连接。这种双队列处理方式和其他一些系统内核措施 (例如 Syn-Cookies/Caches) 联合应用时, 能够比较有效的缓解小规模的 SYN Flood 攻击 (事实证明 <1000p/s) 加大 SYN 队列长度可以容纳更多等待连接的网络连接数, 所以对 Server 来说可以考虑增大该值.)

tcp_window_scaling INTEGER, 缺省值为 1 该文件表示设置 tcp/ip 会话的滑动窗口大小是否可变。参数值为布尔值, 为 1 时表示可变, 为 0 时表示不可变。tcp/ip 通常使用的窗口最大可达到 65535 字节, 对于高速网络, 该值可能太小, 这时候如果启用了该功能, 可以使 tcp/ip 滑动窗口大小增大数个数量级, 从而提高数据传输的能力 (RFC

1323)。(对普通地百 M 网络而言，关闭会降低开销，所以如果不是高速网络，可以考虑设置为 0)

tcp_timestamps BOOLEAN, 缺省值为 1 Timestamps 用在其它一些东西中，可以防范那些伪造的 sequence 号码。一条 1G 的宽带线路或许会重遇到带 out-of-line 数值的旧 sequence 号码 (假如它是由于上次产生的)。Timestamp 会让它知道这是个' 旧封包'。(该文件表示是否启用以一种比超时重发更精确的方法（RFC 1323）来启用对 RTT 的计算；为了实现更好的性能应该启用这个选项。)

tcp_sack BOOLEAN, 缺省值为 1 使用 Selective ACK，它可以用来查找特定的遗失的数据报— 因此有助于快速恢复状态。该文件表示是否启用有选择的应答（Selective Acknowledgment)，这可以通过有选择地应答乱序接收到的报文来提高性能 (这样可以让发送者只发送丢失的报文段)。(对于广域网通信来说这个选项应该启用，但是这会增加对 CPU 的占用。)

tcp_fack BOOLEAN, 缺省值为 1 打开 FACK 拥塞避免和快速重传功能。(注意, 当 tcp_sack 设置为 0 的时候，这个值即使设置为 1 也无效)

tcp_dsack BOOLEAN, 缺省值为 1 允许 TCP 发送" 两个完全相同" 的 SACK。

tcp_ecn BOOLEAN, 缺省值为 0 打开 TCP 的直接拥塞通告功能。

tcp_reordering INTEGER, 默认值是 3 TCP 流中重排序的数据报最大数量。(一般有看到推荐把这个数值略微调整大一些, 比如 5)

tcp_retrans_collapse BOOLEAN，缺省值为 1 对于某些有 bug 的打印机提供针对其 bug 的兼容性。(一般不需要这个支持, 可以关闭它)

n(3 个 INTEGER 变量) min, default, max。min：为 TCP socket 预留用于发送缓冲的内存最小值。每个 tcp socket 都可以在建议以后都可以使用它。默认值为 4096(4K)。

default：为 TCP socket 预留用于发送缓冲的内存数量，默认情况下该值会影响其它协议使用的 net.core.wmem_default 值，一般要低于 net.core.wmem_default 的值。默认值为 16384(16K)。

max: 用于 TCP socket 发送缓冲的内存最大值。该值不会影响 net.core.wmem_max，" 静态" 选择参数 SO_SNDBUF 则不受该值影响。默认值为 131072(128K)。(对于服务器而言，增加这个参数的值对于发送数据很有帮助, 在我的网络环境中, 修改为了 51200 131072 204800)

(3 个 INTEGER 变量) min, default, max。min：为 TCP socket 预留用于接收缓冲的内存数量，即使在内存出现紧张情况下 tcp socket 都至少会有这么多数量的内存用于接收缓冲，默认值为 8K。

default：为 TCP socket 预留用于接收缓冲的内存数量，默认情况下该值影响其它协议使用的 net.core.wmem_default 值。该值决定了在 tcp_adv_win_scale、tcp_app_win 和 tcp_app_win=0 默认值情况下，TCP 窗口大小为 65535。默认值为 87380

max:用于 TCP socket 接收缓冲的内存最大值。该值不会影响 net.core.wmem_max，" 静态" 选择参数 SO_SNDBUF 则不受该值影响。默认值为 128K。默认值为 87380*2 bytes。(可以看出，.max 的设置最好是 default 的两倍, 对于 NAT 来说主要该增加它, 我的网络里为 51200 131072 204800)

n(3 个 INTEGER 变量) low, pressure, high。low：当 TCP 使用了低于该值的内存页面数时，TCP 不会考虑释放内存。(理想情况下，这个值应与指定给 tcp_wmem 的第 2 个值相匹配 - 这第 2 个值表明，最大页面大小乘以最大并发请求数除以页大小 (131072 * 300 / 4096)。)

pressure：当 TCP 使用了超过该值的内存页面数量时，TCP 试图稳定其内存使用，进入 pressure 模式，当内存消耗低于 low 值时则退出 pressure 状态。(理想情况下这个值应该是 TCP 可以使用的总缓冲区大小的最大值 (204800 * 300 / 4096)。)

high：允许所有 tcp sockets 用于排队缓冲数据报的页面量。(如果超过这个值，TCP 连接将被拒绝，这就是为什么不要令其过于保守 (512000 * 300 / 4096) 的原因了。在这种情况下，提供的价值很大，它能处理很多连接，是所预期的 2.5 倍；或者使现有连接能够传输 2.5 倍的数据。我的网络里为 192000 300000 732000)

一般情况下这些值是在系统启动时根据系统内存数量计算得到的。

tcp_app_win INTEGER, 默认值是 31 保留 max($window/2^{tcp\_app\_win}$, mss) 数量的窗口由于应用缓冲。当为 0 时表示不需要缓冲。

tcp_adv_win_scale INTEGER, 默认值为 2 计算缓冲开销 $bytes/2^{tcp\_adv\_win\_scale}$(如果 tcp_adv_win_scale > 0) 或者 $bytes - bytes/2^{-tcp\_adv\_win\_scale}$(如果 tcp_adv_win_scale $<= 0$)。

tcp_rfc1337 BOOLEAN, 缺省值为 0 这个开关可以启动对于在 RFC1337 中描述的"tcp 的 time-wait 暗杀危机" 问题的修复。启用后，内核将丢弃那些发往 time-wait 状态 TCP 套接字的 RST 包.

tcp_low_latency BOOLEAN, 缺省值为 0 允许 TCP/IP 栈适应在高吞吐量情况下低延时的情况；这个选项一般情形是的禁用。(但在构建 Beowulf 集群的时候, 打开它很有帮助)

tcp_westwood BOOLEAN, 缺省值为 0 启用发送者端的拥塞控制算法，它可以维护对吞吐量的评估，并试图对带宽的整体利用情况进行优化；对于 WAN 通信来说应该启用这个选项。

tcp_bic BOOLEAN, 缺省值为 0 为快速长距离网络启用 Binary Increase Congestion；这样可以更好地利用以 GB 速度进行操作的链接；对于 WAN 通信应该启用这个选项。

6.3.2 __tcp_push_pending_frames

```c
/* 将等待在队列中的包全部发出。 */
void __tcp_push_pending_frames(struct sock *sk, unsigned int cur_mss,
                               int nonagle)
{
        /* 如果此时连接已经关闭了，那么直接返回。*/
        if (unlikely(sk->sk_state == TCP_CLOSE))
                return;

        /* 关闭 nagle 算法，将剩余的部分发送出去。 */
        if (tcp_write_xmit(sk, cur_mss, nonagle, 0,
                           sk_gfp_atomic(sk, GFP_ATOMIC)))
                tcp_check_probe_timer(sk);
}
```

6.3.3 tcp_fin_time

计算等待接收 FIN 的超时时间。超时时间至少为 $\frac{7}{2}$ 倍的 rto。

```c
static inline int tcp_fin_time(const struct sock *sk)
{
        int fin_timeout = tcp_sk(sk)->linger2 ? : sysctl_tcp_fin_timeout;
        const int rto = inet_csk(sk)->icsk_rto;

        if (fin_timeout < (rto << 2) - (rto >> 1))
                fin_timeout = (rto << 2) - (rto >> 1);

        return fin_timeout;
}
```

6.3.4 tcp_done

```c
/* 该函数用于完成关闭 TCP 连接，回收并清理相关资源。 */
void tcp_done(struct sock *sk)
{
        struct request_sock *req = tcp_sk(sk)->fastopen_rsk;

        /* 当套接字状态为 SYN_SENT 或 SYN_RECV 时，更新统计数据。 */
        if (sk->sk_state == TCP_SYN_SENT || sk->sk_state == TCP_SYN_RECV)
                TCP_INC_STATS_BH(sock_net(sk), TCP_MIB_ATTEMPTFAILS);

        /* 将连接状态设置为关闭，并清除定时器。 */
        tcp_set_state(sk, TCP_CLOSE);
        tcp_clear_xmit_timers(sk);
        /* 当启用了 Fast Open 时，移除 fastopen 请求 */
        if (req)
                reqsk_fastopen_remove(sk, req, false);

        sk->sk_shutdown = SHUTDOWN_MASK;

        /* 如果状态不为 SOCK_DEAD，则唤醒等待着的进程。 */
        if (!sock_flag(sk, SOCK_DEAD))
                sk->sk_state_change(sk);
        else
                inet_csk_destroy_sock(sk);
}
```

6.3.5 tcp_init_nondata_skb

该函数提供了的功能。函数如下：

```
Location:

        net/ipv4/tcp_output.c

Function:

        初始化不含数据的skb.

Parameter:

        skb:待初始化的sk_buff。
          seq:序号
        flags:标志位
static void tcp_init_nondata_skb(struct sk_buff *skb, u32 seq, u8 flags)
{
        /* 设置校验码 */
        skb->ip_summed = CHECKSUM_PARTIAL;
        skb->csum = 0;

        /* 设置标志位 */
        TCP_SKB_CB(skb)->tcp_flags = flags;
        TCP_SKB_CB(skb)->sacked = 0;

        tcp_skb_pcount_set(skb, 1);

        /* 设置起始序号 */
        TCP_SKB_CB(skb)->seq = seq;
        if (flags & (TCPHDR_SYN | TCPHDR_FIN))
                seq++;
        TCP_SKB_CB(skb)->end_seq = seq;
}
```

6.3.6 before() 和 after()

在一些需要判断序号前后的地方出现了`before()`和`after()` 这两个函数。这两个函数的定义如下

```
/* include/net/tcp.h
 * 比较两个无符号 32 位整数
 */
static inline bool before(__u32 seq1, __u32 seq2)
{
        return (__s32)(seq1-seq2) < 0;
}
#define after(seq2, seq1)       before(seq1, seq2)
```

可以看到，这两个函数实际上就是将两个数直接相减。之所以要单独弄个函数应该是为了避免强制转型造成影响。序号都是 32 位无符号整型。

6.3.7 tcp_shutdown

`tcp_shutdown`是 TCP 的 shutdown 系统调用的传输层接口实现，由套接口层的实现`inet_shutdown`调用。

```
/*
Location:

        net/ipv4/tcp.c

Function:

        Shutdown the sending side of a connection. Much like close except
        that we don't receive shut down or sock_set_flag(sk, SOCK_DEAD).

Parameter:

        sk: 传输控制块。
        how:
*/
void tcp_shutdown(struct sock *sk, int how)
{
        /*        We need to grab some memory, and put together a FIN,
         *        and then put it into the queue to be sent.
         *                Tim MacKenzie(tym@dibbler.cs.monash.edu.au) 4 Dec '92.
         */
        if (!(how & SEND_SHUTDOWN))
                return;

        /* If we've already sent a FIN, or it's a closed state, skip this. */
        if ((1 << sk->sk_state) &
            (TCPF_ESTABLISHED | TCPF_SYN_SENT |
             TCPF_SYN_RECV | TCPF_CLOSE_WAIT)) {
                /* Clear out any half completed packets.  FIN if needed. */
                if (tcp_close_state(sk))
                        tcp_send_fin(sk);
        }
}
```

如果是发送方向的关闭，并且 TCP 状态为 ESTABLISHED、SYN_SENT、SYN_RECV 或 CLOSE_WAIT 时，根据 TC 状态迁移图和当前的状态设置新的状态，并在需要发送 FIN 时，调用 FIN 时，调用`tcp_send_fin`时向对方发送 FIN。

而对于接收方向的关闭，则无需向对方发送 FIN，因为可能还需要向对方发送数据。至于接收方向的关闭的实现，在 recvmsg 系统调用中发现设置了 RCV_SHUTDOWN 标志会立即返回。

6.3.8 tcp_close

```
/*
Location:

        net/ipv4/tcp.c

Function:



Parameter:

        sk: 传输控制块。
```

```
13          timeout: 在真正关闭控制块之前，可以发送剩余数据的时间。
14
15  */
16  void tcp_close(struct sock *sk, long timeout)
17  {
18          struct sk_buff *skb;
19          int data_was_unread = 0;
20          int state;
21
22          lock_sock(sk);
23          sk->sk_shutdown = SHUTDOWN_MASK;
24
25          if (sk->sk_state == TCP_LISTEN) {
26                  tcp_set_state(sk, TCP_CLOSE);
27
28                  /* Special case. */
29                  inet_csk_listen_stop(sk);
30
31                  goto adjudge_to_death;
32          }
```

首先，对传输控制块加锁。然后设置关闭标志为 SHUTDOWN_MASK，表示进行双向的关闭。

如果套接口处于侦听状态，这种情况处理相对比较简单，因为没有建立起连接，因此无需发送 FIN 等操作。设置 TCP 的状态为 CLOSE，然后终止侦听。最后跳转到 adjudge_to_death 处进行相关处理。

```
1           /*  We need to flush the recv. buffs.  We do this only on the
2            *  descriptor close(什么意思), not protocol-sourced(什么意思) closes, because the
3            *  reader process may not have drained(消耗) the data yet!
4            */
5           while ((skb = __skb_dequeue(&sk->sk_receive_queue)) != NULL) {
6                   u32 len = TCP_SKB_CB(skb)->end_seq - TCP_SKB_CB(skb)->seq;
7
8                   if (TCP_SKB_CB(skb)->tcp_flags & TCPHDR_FIN)
9                           len--;
10                  data_was_unread += len;
11                  __kfree_skb(skb);
12          }
13
14          sk_mem_reclaim(sk);
```

因为是关闭连接，因此需要释放已接收队列上的段，同时统计释放了多少数据，然后回收缓存。

```
1           /* If socket has been already reset (e.g. in tcp_reset()) - kill it. */
2           if (sk->sk_state == TCP_CLOSE)
3                   goto adjudge_to_death;
```

如果 socket 本身就是 close 状态的话，直接跳到 adjudge_to_death 就好。

```
1           /* As outlined in RFC 2525, section 2.17, we send a RST here because
2            * data was lost. To witness the awful effects of the old behavior of
3            * always doing a FIN, run an older 2.1.x kernel or 2.0.x, start a bulk
4            * GET in an FTP client, suspend the process, wait for the client to
5            * advertise a zero window, then kill -9 the FTP client, wheee...
6            * Note: timeout is always zero in such a case.
```

```
         */
        if (unlikely(tcp_sk(sk)->repair)) {
                sk->sk_prot->disconnect(sk, 0);
        } else if (data_was_unread) {
                /* Unread data was tossed, zap the connection. */
                NET_INC_STATS_USER(sock_net(sk), LINUX_MIB_TCPABORTONCLOSE);
                tcp_set_state(sk, TCP_CLOSE);
                tcp_send_active_reset(sk, sk->sk_allocation);
        } else if (sock_flag(sk, SOCK_LINGER) && !sk->sk_lingertime) {
                /* Check zero linger _after_ checking for unread data. */
                sk->sk_prot->disconnect(sk, 0);
                NET_INC_STATS_USER(sock_net(sk), LINUX_MIB_TCPABORTONDATA);
        } else if (tcp_close_state(sk)) {
                /* We FIN if the application ate all the data before
                 * zapping the connection.
                 */

                /* RED-PEN. Formally speaking, we have broken TCP state
                 * machine. State transitions:
                 *
                 * TCP_ESTABLISHED -> TCP_FIN_WAIT1
                 * TCP_SYN_RECV        -> TCP_FIN_WAIT1 (forget it, it's impossible)
                 * TCP_CLOSE_WAIT -> TCP_LAST_ACK
                 *
                 * are legal only when FIN has been sent (i.e. in window),
                 * rather than queued out of window. Purists blame.
                 *
                 * F.e. "RFC state" is ESTABLISHED,
                 * if Linux state is FIN-WAIT-1, but FIN is still not sent.
                 *
                 * The visible declinations are that sometimes
                 * we enter time-wait state, when it is not required really
                 * (harmless), do not send active resets, when they are
                 * required by specs (TCP_ESTABLISHED, TCP_CLOSE_WAIT, when
                 * they look as CLOSING or LAST_ACK for Linux)
                 * Probably, I missed some more holelets.
                 *                                              --ANK
                 * XXX (TFO) - To start off we don't support SYN+ACK+FIN
                 * in a single packet! (May consider it later but will
                 * probably need API support or TCP_CORK SYN-ACK until
                 * data is written and socket is closed.)
                 */
                tcp_send_fin(sk);
        }

        sk_stream_wait_close(sk, timeout);
```

```
adjudge_to_death:
        state = sk->sk_state;
        sock_hold(sk);
        sock_orphan(sk);

        /* It is the last release_sock in its life. It will remove backlog. */
        release_sock(sk);


        /* Now socket is owned by kernel and we acquire BH lock
           to finish close. No need to check for user refs.
```

```
12          */
13         local_bh_disable();
14         .(sk);
15         WARN_ON(sock_owned_by_user(sk));
16
17         percpu_counter_inc(sk->sk_prot->orphan_count);
18
19         /* Have we already been destroyed by a softirq or backlog? */
20         if (state != TCP_CLOSE && sk->sk_state == TCP_CLOSE)
21                 goto out;
22
23         /*        This is a (useful) BSD violating of the RFC. There is a
24          *        problem with TCP as specified in that the other end could
25          *        keep a socket open forever with no application left this end.
26          *        We use a 1 minute timeout (about the same as BSD) then kill
27          *        our end. If they send after that then tough - BUT: long enough
28          *        that we won't make the old 4*rto = almost no time - whoops
29          *        reset mistake.
30          *
31          *        Nope, it was not mistake. It is really desired behaviour
32          *        f.e. on http servers, when such sockets are useless, but
33          *        consume significant resources. Let's do it with special
34          *        linger2        option.                                    --ANK
35          */
36
37         if (sk->sk_state == TCP_FIN_WAIT2) {
38                 struct tcp_sock *tp = tcp_sk(sk);
39                 if (tp->linger2 < 0) {
40                         tcp_set_state(sk, TCP_CLOSE);
41                         tcp_send_active_reset(sk, GFP_ATOMIC);
42                         NET_INC_STATS_BH(sock_net(sk),
43                                         LINUX_MIB_TCPABORTONLINGER);
44                 } else {
45                         const int tmo = tcp_fin_time(sk);
46
47                         if (tmo > TCP_TIMEWAIT_LEN) {
48                                 inet_csk_reset_keepalive_timer(sk,
49                                                 tmo - TCP_TIMEWAIT_LEN);
50                         } else {
51                                 tcp_time_wait(sk, TCP_FIN_WAIT2, tmo);
52                                 goto out;
53                         }
54                 }
55         }
56         if (sk->sk_state != TCP_CLOSE) {
57                 sk_mem_reclaim(sk);
58                 if (tcp_check_oom(sk, 0)) {
59                         tcp_set_state(sk, TCP_CLOSE);
60                         tcp_send_active_reset(sk, GFP_ATOMIC);
61                         NET_INC_STATS_BH(sock_net(sk),
62                                         LINUX_MIB_TCPABORTONMEMORY);
63                 }
64         }
65
66         if (sk->sk_state == TCP_CLOSE) {
67                 struct request_sock *req = tcp_sk(sk)->fastopen_rsk;
68                 /* We could get here with a non-NULL req if the socket is
69                  * aborted (e.g., closed with unread data) before 3WHS
```

```
70                  * finishes.
71                  */
72                 if (req)
73                         reqsk_fastopen_remove(sk, req, false);
74                 inet_csk_destroy_sock(sk);
75         }
76         /* Otherwise, socket is reprieved until protocol close. */
77
78 out:
79         bh_unlock_sock(sk);
80         local_bh_enable();
81         sock_put(sk);
82 }
```

CHAPTER 7

附录: 基础知识

Contents

7.1 GNU/LINUX

7.1.1 字节转换

7.1.1.1 ntohs-Network to Host Short Int

ntohs() 用于将一个无符号短整形数从网络字节顺序转换为主机字节顺序。这是因为网络中的字节一般都是大端存储。

```
1
```

7.1.2 错误处理

7.1.2.1 错误码

```
/*
Location:

        1 -  34   include/uapi/asm-generic/errno-base.h
        35- 133          include/uapi/asm-generic/errno.h

Function:

        定义了 Linux 中的错误码。
*/
#define        EPERM            1        /* Operation not permitted */
#define        ENOENT            2        /* No such file or directory */
#define        ESRCH            3        /* No such process */
#define        EINTR            4        /* Interrupted system call */
#define        EIO                    5        /* I/O error */
#define        ENXIO            6        /* No such device or address */
#define        E2BIG            7        /* Argument list too long */
#define        ENOEXEC              8        /* Exec format error */
#define        EBADF            9        /* Bad file number */
#define        ECHILD           10        /* No child processes */
#define        EAGAIN           11        /* Try again */
#define        ENOMEM           12        /* Out of memory */
#define        EACCES           13        /* Permission denied */
#define        EFAULT           14        /* Bad address */
#define        ENOTBLK           15        /* Block device required */
#define        EBUSY            16        /* Device or resource busy */
#define        EEXIST           17        /* File exists */
#define        EXDEV            18        /* Cross-device link */
#define        ENODEV           19        /* No such device */
#define        ENOTDIR           20        /* Not a directory */
#define        EISDIR           21        /* Is a directory */
#define        EINVAL           22        /* Invalid argument */
#define        ENFILE           23        /* File table overflow */
#define        EMFILE           24        /* Too many open files */
#define        ENOTTY           25        /* Not a typewriter */
#define        ETXTBSY           26        /* Text file busy */
#define        EFBIG            27        /* File too large */
#define        ENOSPC           28        /* No space left on device */
#define        ESPIPE           29        /* Illegal seek */
#define        EROFS            30        /* Read-only file system */
#define        EMLINK           31        /* Too many links */
#define        EPIPE            32        /* Broken pipe */
```

```
43  #define        EDOM                33        /* Math argument out of domain of func */
44  #define        ERANGE               34        /* Math result not representable */
45
46  #define        EDEADLK               35        /* Resource deadlock would occur */
47  #define        ENAMETOOLONG        36        /* File name too long */
48  #define        ENOLCK               37        /* No record locks available */
49
50  /*
51   * This error code is special: arch syscall entry code will return
52   * -ENOSYS if users try to call a syscall that doesn't exist.  To keep
53   * failures of syscalls that really do exist distinguishable from
54   * failures due to attempts to use a nonexistent syscall, syscall
55   * implementations should refrain from returning -ENOSYS.
56   */
57  #define        ENOSYS               38        /* Invalid system call number */
58
59  #define        ENOTEMPTY        39        /* Directory not empty */
60  #define        ELOOP               40        /* Too many symbolic links encountered */
61  #define        EWOULDBLOCK        EAGAIN        /* Operation would block */
62  #define        ENOMSG               42        /* No message of desired type */
63  #define        EIDRM               43        /* Identifier removed */
64  #define        ECHRNG               44        /* Channel number out of range */
65  #define        EL2NSYNC        45        /* Level 2 not synchronized */
66  #define        EL3HLT               46        /* Level 3 halted */
67  #define        EL3RST               47        /* Level 3 reset */
68  #define        ELNRNG               48        /* Link number out of range */
69  #define        EUNATCH               49        /* Protocol driver not attached */
70  #define        ENOCSI               50        /* No CSI structure available */
71  #define        EL2HLT               51        /* Level 2 halted */
72  #define        EBADE               52        /* Invalid exchange */
73  #define        EBADR               53        /* Invalid request descriptor */
74  #define        EXFULL               54        /* Exchange full */
75  #define        ENOANO               55        /* No anode */
76  #define        EBADRQC               56        /* Invalid request code */
77  #define        EBADSLT               57        /* Invalid slot */
78
79  #define        EDEADLOCK        EDEADLK
80
81  #define        EBFONT               59        /* Bad font file format */
82  #define        ENOSTR               60        /* Device not a stream */
83  #define        ENODATA               61        /* No data available */
84  #define        ETIME               62        /* Timer expired */
85  #define        ENOSR               63        /* Out of streams resources */
86  #define        ENONET               64        /* Machine is not on the network */
87  #define        ENOPKG               65        /* Package not installed */
88  #define        EREMOTE               66        /* Object is remote */
89  #define        ENOLINK               67        /* Link has been severed */
90  #define        EADV               68        /* Advertise error */
91  #define        ESRMNT               69        /* Srmount error */
92  #define        ECOMM               70        /* Communication error on send */
93  #define        EPROTO               71        /* Protocol error */
94  #define        EMULTIHOP        72        /* Multihop attempted */
95  #define        EDOTDOT               73        /* RFS specific error */
96  #define        EBADMSG               74        /* Not a data message */
97  #define        EOVERFLOW        75        /* Value too large for defined data type */
98  #define        ENOTUNIQ        76        /* Name not unique on network */
99  #define        EBADFD               77        /* File descriptor in bad state */
100 #define        EREMCHG               78        /* Remote address changed */
```

```
101 #define        ELIBACC                 79        /* Can not access a needed shared library */
102 #define        ELIBBAD                 80        /* Accessing a corrupted shared library */
103 #define        ELIBSCN                 81        /* .lib section in a.out corrupted */
104 #define        ELIBMAX                 82        /* Attempting to link in too many shared libraries */
105 #define        ELIBEXEC        83        /* Cannot exec a shared library directly */
106 #define        EILSEQ                84        /* Illegal byte sequence */
107 #define        ERESTART        85        /* Interrupted system call should be restarted */
108 #define        ESTRPIPE        86        /* Streams pipe error */
109 #define        EUSERS                87        /* Too many users */
110 #define        ENOTSOCK        88        /* Socket operation on non-socket */
111 #define        EDESTADDRREQ        89        /* Destination address required */
112 #define        EMSGSIZE        90        /* Message too long */
113 #define        EPROTOTYPE        91        /* Protocol wrong type for socket */
114 #define        ENOPROTOOPT        92        /* Protocol not available */
115 #define        EPROTONOSUPPORT        93        /* Protocol not supported */
116 #define        ESOCKTNOSUPPORT        94        /* Socket type not supported */
117 #define        EOPNOTSUPP        95        /* Operation not supported on transport endpoint */
118 #define        EPFNOSUPPORT        96        /* Protocol family not supported */
119 #define        EAFNOSUPPORT        97        /* Address family not supported by protocol */
120 #define        EADDRINUSE        98        /* Address already in use */
121 #define        EADDRNOTAVAIL        99        /* Cannot assign requested address */
122 #define        ENETDOWN        100        /* Network is down */
123 #define        ENETUNREACH        101        /* Network is unreachable */
124 #define        ENETRESET        102        /* Network dropped connection because of reset */
125 #define        ECONNABORTED        103        /* Software caused connection abort */
126 #define        ECONNRESET        104        /* Connection reset by peer */
127 #define        ENOBUFS                105        /* No buffer space available */
128 #define        EISCONN                106        /* Transport endpoint is already connected */
129 #define        ENOTCONN        107        /* Transport endpoint is not connected */
130 #define        ESHUTDOWN        108        /* Cannot send after transport endpoint shutdown */
131 #define        ETOOMANYREFS        109        /* Too many references: cannot splice */
132 #define        ETIMEDOUT        110        /* Connection timed out */
133 #define        ECONNREFUSED        111        /* Connection refused */
134 #define        EHOSTDOWN        112        /* Host is down */
135 #define        EHOSTUNREACH        113        /* No route to host */
136 #define        EALREADY        114        /* Operation already in progress */
137 #define        EINPROGRESS        115        /* Operation now in progress */
138 #define        ESTALE                116        /* Stale file handle */
139 #define        EUCLEAN                117        /* Structure needs cleaning */
140 #define        ENOTNAM                118        /* Not a XENIX named type file */
141 #define        ENAVAIL                119        /* No XENIX semaphores available */
142 #define        EISNAM                120        /* Is a named type file */
143 #define        EREMOTEIO        121        /* Remote I/O error */
144 #define        EDQUOT                122        /* Quota exceeded */
145
146 #define        ENOMEDIUM        123        /* No medium found */
147 #define        EMEDIUMTYPE        124        /* Wrong medium type */
148 #define        ECANCELED        125        /* Operation Canceled */
149 #define        ENOKEY                126        /* Required key not available */
150 #define        EKEYEXPIRED        127        /* Key has expired */
151 #define        EKEYREVOKED        128        /* Key has been revoked */
152 #define        EKEYREJECTED        129        /* Key was rejected by service */
153
154 /* for robust mutexes */
155 #define        EOWNERDEAD        130        /* Owner died */
156 #define        ENOTRECOVERABLE        131        /* State not recoverable */
157
158 #define ERFKILL                132        /* Operation not possible due to RF-kill */
159
```

```
160 #define EHWPOISON       133     /* Memory page has hardware error */
```

7.1.2.2 IS_ERR,PTR_ERR,ERR_PTR

相关定义如下:

```
1  /*
2  Location:
3
4          include/linux/err.h
5
6  Description:
7           Kernel pointers have redundant information, so we can use a
8          scheme where we can return either an error code or a normal
9          pointer with the same return value.
10
11         This should be a per-architecture thing, to allow different
12         error and pointer decisions.
13
14 */
15 #define MAX_ERRNO       4095
16
17 #ifndef __ASSEMBLY__
18
19 #define IS_ERR_VALUE(x) unlikely((x) >= (unsigned long)-MAX_ERRNO)
20
21 static inline void * __must_check ERR_PTR(long error)
22 {
23         return (void *) error;
24 }
25
26 static inline long __must_check PTR_ERR(__force const void *ptr)
27 {
28         return (long) ptr;
29 }
30
31 static inline bool __must_check IS_ERR(__force const void *ptr)
32 {
33         return IS_ERR_VALUE((unsigned long)ptr);
34 }
35
36 static inline bool __must_check IS_ERR_OR_NULL(__force const void *ptr)
37 {
38         return !ptr || IS_ERR_VALUE((unsigned long)ptr);
39 }
```

要想明白上述的代码，首先要理解内核空间。所有的驱动程序都是运行在内核空间，内核空间虽然很大，但总是有限的，而在这有限的空间中，其最后一个 page 是专门保留的，也就是说一般人不可能用到内核空间最后一个 page 的指针。换句话说，你在写设备驱动程序的过程中，涉及到的任何一个指针，必然有三种情况：有效指针；NULL，空指针；错误指针，或者说无效指针。

而所谓的错误指针就是指其已经到达了最后一个 page，即内核用最后一页捕捉错误。比如对于 32bit 的系统来说，内核空间最高地址 0xffffffff，那么最后一个 page 就是指的 0xfffff000 0xffffffff(假设 4k 一个 page)，这段地址是被保留的。内核空间为什么留出最后一个 page? 我们知道一个 page 可能是 4k，也可能是更多，比如 8k，但至少它也

是 4k，留出一个 page 出来就可以让我们把内核空间的指针来记录错误了。内核返回的指针一般是指向页面的边界 (4k 边界)，即`ptr & 0xfff == 0`。如果你发现你的一个指针指向这个范围中的某个地址，那么你的代码肯定出错了。IS_ERR() 就是判断指针是否有错，如果指针并不是指向最后一个 page，那么没有问题；如果指针指向了最后一个 page，那么说明实际上这不是一个有效的指针，这个指针里保存的实际上是一种错误代码。而通常很常用的方法就是先用 IS_ERR() 来判断是否是错误，然后如果是，那么就调用 PTR_ERR() 来返回这个错误代码。因此，判断一个指针是不是有效的，我们可以调用宏 IS_ERR_VALUE，即判断指针是不是在 (0xfffff000, 0xffffffff) 之间，因此，可以用 IS_ERR() 来判断内核函数的返回值是不是一个有效的指针。注意这里用 unlikely() 的用意!

至于 PTR_ERR(), ERR_PTR()，只是强制转换以下而已。而 PTR_ERR() 只是返回错误代码，也就是提供一个信息给调用者，如果你只需要知道是否出错，而不在乎因为什么而出错，那你当然不用调用 PTR_ERR() 了。

如果指针指向了最后一个 page，那么说明实际上这不是一个有效的指针。这个指针里保存的实际上是一种错误代码。而通常很常用的方法就是先用 IS_ERR() 来判断是否是错误，然后如果是，那么就调用 PTR_ERR() 来返回这个错误代码。

7.2 C 语言

7.2.1 结构体初始化

```
typedef struct{
        int  a;
        char ch;
}flag;
/*
        目的是将 a 初始化为 1,ch 初始化为'u'.
*/
/* 法一：分别初始化 */
flag tmp;
tmp.a=1;
tmp.ch='u';

/* 法二：点式初始化 */
flag tmp={.a=1,.ch='u'};                //注意两个变量之间使用 , 而不是;
/* 法三：*/
flag tmp={
                        a:1,
                        ch:'u'
                };
```

当然，我们也可以使用上述任何一种方法只对结构体中的某几个变量进行初始化。

7.2.2 位字段

在存储空间极为宝贵的情况下，有可能需要将多个对象保存在一个机器字中。而在 linux 开发的早期，那时确实空间极其宝贵。于是乎，那一帮黑客们就发明了各种各样的办法。一种常用的办法是使用类似于编译器符号表的单个二进制位标志集合，即定义一

系列的 2 的指数次方的数，此方法确实有效。但是，仍然十分浪费空间。而且有可能很多位都利用不到。于是乎，他们提出了另一种新的思路即位字段。我们可以利用如下方式定义一个包含 3 位的变量。

```
struct {
        unsigned int a:1;
        unsigned int b:1;
        unsigned int c:1;
}flags;
```

字段可以不命名，无名字段，即只有一个冒号和宽度，起到填充作用。特殊宽度 0 可以用来强制在下一个字边界上对齐，一般位于结构体的尾部。

冒号后面表示相应字段的宽度 (二进制宽度)，即不一定非得是 1 位。字段被声明为`unsigned int`类型，以确保它们是无符号量。

当然我们需要注意，机器是分大端和小端存储的。因此，我们在选择外部定义数据的情况下爱，必须仔细考虑那一端优先的问题。同时，字段不是数组，并且没有地址，因此不能对它们使用&运算符。

7.3 GCC

7.3.1 __attribute__

GNU C 的一大特色就是`__attribute__`机制。`__attribute__`可以设置函数属性 (Function Attribute)、变量属性 (Variable Attribute) 和类型属性 (Type Attribute)。

`__attribute__`的书写特征是: `__attribute__`前后都有两个下划线, 并切后面会紧跟一对原括弧，括弧里面是相应的`__attribute__`参数。

`__attribute__`的语法格式为: `__attribute__`((attribute-list))

`__attribute__`的位置约束为:放于声明的尾部";"之前。参考博客:http://www.cnblogs.com/astwish/p/

关键字`__attribute__`也可以对结构体 (struct) 或共用体 (union) 进行属性设置。大致有六个参数值可以被设定，即: aligned, packed, `transparent_union`, unused, deprecated 和 `may_alias` 。

在使用`__attribute__`参数时, 你也可以在参数的前后都加上两个下划线, 例如, 使用`__aligned__`而不是 aligned , 这样, 你就可以在相应的头文件里使用它而不用关心头文件里是否有重名的宏定义。aligned (alignment) 该属性设定一个指定大小的对齐格式 (以字节为单位)。

7.3.1.1 设置变量属性

下面的声明将强制编译器确保 (尽它所能) 变量类型为`int32_t`的变量在分配空间时采用 8 字节对齐方式。

```
typedef int int32_t __attribute__ ((aligned(8)));
```

7.3.1.2 设置类型属性

下面的声明将强制编译器确保（尽它所能）变量类型为 struct S 的变量在分配空间时采用 8 字节对齐方式。

```
struct S {
        short b[3];
} __attribute__ ((aligned (8)));
```

如上所述，你可以手动指定对齐的格式，同样，你也可以使用默认的对齐方式。如果 aligned 后面不紧跟一个指定的数字值，那么编译器将依据你的目标机器情况使用最大最有益的对齐方式。例如：

```
struct S {
        short b[3];
} __attribute__ ((aligned));
```

这里，如果 sizeof（short）的大小为 2（byte），那么，S 的大小就为 6 。取一个 2 的次方值，使得该值大于等于 6 ，则该值为 8，所以编译器将设置 S 类型的对齐方式为 8 字节。aligned 属性使被设置的对象占用更多的空间，相反的，使用 packed 可以减小对象占用的空间。需要注意的是，attribute 属性的效力与你的连接器也有关，如果你的连接器最大只支持 16 字节对齐，那么你此时定义 32 字节对齐也是无济于事的。

7.3.1.3 设置函数属性

7.3.2 分支预测优化

现代处理器均为流水线结构。而分支语句可能导致流水线断流。因此，很多处理器均有分支预测的功能。然而，分支预测失败所导致的惩罚也是相对高昂的。为了提升性能，Linux 的很多分支判断中都使用了 `likely()`和`unlikely()`这组宏定义来人工指示编译器，哪些分支出现的概率极高，以便编译器进行优化。

这里有两种定义，一种是开启了分支语句分析相关的选项时，内核会采用下面的一种定义

```
/* include/linux/compiler.h
 * 采用 __builtin_constant_p(x) 来忽略常量表达式。
 */
# ifndef likely
#  define likely(x)     (__builtin_constant_p(x) ? !!(x) : __branch_check__(x, 1))
# endif
# ifndef unlikely
#  define unlikely(x)   (__builtin_constant_p(x) ? !!(x) : __branch_check__(x, 0))
# endif
```

在该定义下，`__branch_check__`用于跟踪分支结果并更新统计数据。

如果不开启该选项，则定义得较为简单：

```
# define likely(x)      __builtin_expect(!!(x), 1)
# define unlikely(x)    __builtin_expect(!!(x), 0)
```

其中`__builtin_expect`是 GCC 的内置函数，用于指示编译器，该条件语句最可能的结果是什么。

7.4 Sparse

7.4.1 __bitwise

```
#define __bitwise    __attribute__((bitwise))
```

该宏主要是为了确保变量是相同的位方式 (比如 bit-endian, little-endiandeng)，对于使用了__bitwise宏的变量，Sparse 会检查这个变量是否一直在同一种位方式 (big-endian, little-endian 或其他) 下被使用。如果此变量再多个位方式下被使用了，Sparse 会给出警告。例子:

```
#ifdef __CHECKER__
#define __bitwise__ __attribute__((bitwise))
#else
#define __bitwise__
#endif
#ifdef __CHECK_ENDIAN__
#define __bitwise __bitwise__
#else
#define __bitwise
#endif

typedef __u16 __bitwise __le16;
typedef __u16 __bitwise __be16;
typedef __u32 __bitwise __le32;
typedef __u32 __bitwise __be32;
typedef __u64 __bitwise __le64;
typedef __u64 __bitwise __be64;
```

7.5 操作系统

7.5.1 RCU

RCU(Read-Copy Update), 是数据同步的一种方式，在当前的 Linux 内核中发挥着重要的作用。RCU 主要针对的数据对象是链表，目的是提高遍历读取数据的效率，为了达到目的使用 RCU 机制读取数据的时候不对链表进行耗时的加锁操作。这样在同一时间可以有多个线程同时读取该链表，并且允许一个线程对链表进行修改 (修改的时候，需要加锁)。RCU 适用于需要频繁的读取数据，而相应修改数据并不多的情景，例如在文件系统中，经常需要查找定位目录，而对目录的修改相对来说并不多，这就是 RCU 发挥作用的最佳场景。

Linux 内核源码当中, 关于 RCU 的文档比较齐全, 你可以在 /Documentation/RCU/ 目录下找到这些文件。

在 RCU 的实现过程中，我们主要解决以下问题:

1、在读取过程中，另外一个线程删除了一个节点。删除线程可以把这个节点从链表中移除，但它不能直接销毁这个节点，必须等到所有的读取线程读取完成以后，才进行销毁操作。RCU 中把这个过程称为宽限期 (Grace period)。

2、在读取过程中，另外一个线程插入了一个新节点，而读线程读到了这个节点，那么需要保证读到的这个节点是完整的。这里涉及到了发布-订阅机制 (Publish-Subscribe

Mechanism)。

3、保证读取链表的完整性。新增或者删除一个节点，不至于导致遍历一个链表从中间断开。但是 RCU 并不保证一定能读到新增的节点或者不读到要被删除的节点。

7.6 CPU

7.6.1 字节序

CPU 分为大端和小端两种。而在网络传输的过程中，大小端的不一致会带来问题。因此，网络协议中对于字节序都有明确规定。一般采用大端序。

Linux 中，对于这一部分的支持放在了`include/linux/byteorder/generic.h` 中。而实现，则交由体系结构相关的代码来完成。

```
/* 下面的函数用于进行对 16 位整型或者 32 位整型在网络传输格式和本地格式之间的转换。
 */
ntohl(__u32 x)
ntohs(__u16 x)
htonl(__u32 x)
htons(__u16 x)
```

上面函数的命名规则是末尾的 l 代表 32 位，s 代表 16 位。n 代表 network，h 代表 host。根据命名规则，不难知道函数的用途。比如 htons 就是从本地的格式转换的网络传输用的格式，转换的是 16 位整数。