

# TCP 高级实验

王鹿鸣, 刘保证

2016 年 6 月 8 日

<b>1</b>	<b>准备部分</b>	<b>1</b>
1.1	用户层 TCP . . . . .	1
1.2	探寻 tcp_prot, 地图 get~ . . . . .	1
1.3	RFC . . . . .	3
1.3.1	RFC793——Transmission Control Protocol . . . . .	3
1.3.2	RFC1323——TCP Extensions for High Performance . . . . .	5
1.3.3	RFC1337——TIME-WAIT Assassination Hazards in TCP . . . . .	7
1.3.4	RFC2525——Known TCP Implementation Problems . . . . .	8
1.3.5	RFC3168——The Addition of Explicit Congestion Notification (ECN) to IP . . . . .	8
1.3.6	RFC7413——TCP Fast Open(Draft) . . . . .	9
<b>2</b>	<b>网络子系统相关核心数据结构</b>	<b>11</b>
2.1	网络子系统数据结构架构 . . . . .	11
2.2	sock 底层数据结构 . . . . .	11
2.2.1	sock_common . . . . .	11
2.2.2	sock . . . . .	13
2.2.3	request_sock . . . . .	17
2.2.4	sk_buff . . . . .	17
2.3	inet 层相关数据结构 . . . . .	22
2.3.1	inet_request_sock . . . . .	22
2.3.2	inet_connection_sock_af_ops . . . . .	22
2.3.3	inet_connect_sock . . . . .	23
2.3.4	inet_timewait_sock . . . . .	24
2.3.5	sockaddr & sockaddr_in . . . . .	25
2.3.6	ip_options . . . . .	26
2.4	路由相关数据结构 . . . . .	26

2.4.1	dst_entry	26
2.4.2	rtable	28
2.4.3	flowi	29
2.5	TCP 层相关数据结构	29
2.5.1	tcphdr	29
2.5.2	tcp_options_received	30
2.5.3	tcp_sock	31
2.5.4	tcp_request_sock	34
2.5.5	tcp_skb_cb	35
<b>3</b>	<b>TCP 建立连接过程</b>	<b>37</b>
3.1	TCP 主动打开-客户	37
3.1.1	基本流程	37
3.1.2	第一次握手：构造并发送 SYN 包	37
3.1.3	第二次握手：接收 SYN+ACK 包	41
3.1.4	第三次握手——发送 ACK 包	47
3.1.5	tcp_transmit_skb	48
3.1.6	tcp_select_window(struct sk_buff *skb)	49
3.2	TCP 被动打开-服务器	52
3.2.1	基本流程	52
3.2.2	第一次握手：接受 SYN 段	53
3.2.3	第二次握手：发送 SYN+ACK 段	60
3.2.4	第三次握手：接收 ACK 段	64
<b>4</b>	<b>TCP 连接释放</b>	<b>69</b>
4.1	主动关闭	69
4.1.1	第一次握手——发送 FIN	69
4.1.2	第二次握手——接受 ACK	71
4.1.3	第三次握手——接受 FIN	74
4.1.4	第四次握手——发送 ACK	76
4.1.5	同时关闭	78
4.1.6	TIME_WAIT	79
4.2	被动关闭	81
4.2.1	基本流程	81
4.2.2	第一次握手：接收FIN	81
<b>5</b>	<b>非核心函数分析</b>	<b>85</b>
5.1	SKB	85
5.2	Inet	85
5.2.1	inet_hash_connect && __inet_hash_connect	85
5.2.2	inet_twsk_put	87

5.3	TCP 层 . . . . .	87
5.3.1	__tcp_push_pending_frames . . . . .	87
5.3.2	tcp_fin_time . . . . .	88
5.3.3	tcp_done . . . . .	88
5.3.4	tcp_init_nodata_skb . . . . .	89
5.3.5	before() 和 after() . . . . .	89
5.3.6	tcp_shutdown . . . . .	89
5.3.7	tcp_close . . . . .	90
<b>6</b>	<b>附录: 基础知识</b>	<b>95</b>
6.1	C 语言 . . . . .	95
6.1.1	结构体初始化 . . . . .	95
6.1.2	位字段 . . . . .	95
6.2	GCC . . . . .	96
6.2.1	__attribute__ . . . . .	96
6.2.2	分支预测优化 . . . . .	97
6.3	操作系统 . . . . .	98
6.4	CPU . . . . .	98
6.4.1	字节序 . . . . .	98

# CHAPTER 1

准备部分

## Contents

1.1	用户层 TCP	1
1.2	探寻 tcp_prot, 地图 get~	1
1.3	RFC	3
1.3.1	RFC793—Transmission Control Protocol	3
1.3.1.1	TCP 状态图	3
1.3.1.2	TCP 头部格式	4
1.3.2	RFC1323—TCP Extensions for High Performance	5
1.3.2.1	简介	5
1.3.2.2	窗口缩放 (Window Scale)	6
1.3.2.3	PAWS(Protect Against Wrapped Sequence Numbers)	6
1.3.3	RFC1337—TIME-WAIT Assassination Hazards in TCP	7
1.3.3.1	TIME-WAIT Assassination(TWA) 现象	7
1.3.3.2	TWA 的危险性及现有的解决方法	8
1.3.4	RFC2525—Known TCP Implementation Problems	8
1.3.5	RFC3168—The Addition of Explicit Congestion Notification (ECN) to IP	8
1.3.5.1	ECN	8
1.3.5.2	TCP 中的 ECN	9
1.3.6	RFC7413—TCP Fast Open(Draft)	9
1.3.6.1	概要	10
1.3.6.2	Fast Open 选项格式	10

## 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`。

```

1 // Server
2 socket(...,SOCK_STREAM,0);
3 bind(...,&server_address, ...);
4 listen(...);
5 accept(..., &client_address, ...);
6 recv(..., &clientaddr, ...);
7 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`。

```

1 socket(...,SOCK_STREAM,0);
2 connect();
3 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。

```

1 struct proto tcp_prot = {
2     .name           = "TCP",
3     .owner          = THIS_MODULE,
4     .close          = tcp_close,
5     .connect        = tcp_v4_connect,
6     .disconnect     = tcp_disconnect,
7     .accept         = inet_csk_accept,
8     .destroy        = tcp_v4_destroy_sock,
9     .shutdown       = tcp_shutdown,
10    .setsockopt      = tcp_setsockopt,
11    .getsockopt      = tcp_getsockopt,
12    .recvmsg         = tcp_recvmsg,
13    .sendmsg         = tcp_sendmsg,
14    .sendpage        = tcp_sendpage,
15    .backlog_rcv     = tcp_v4_do_rcv,
16    .get_port        = inet_csk_get_port,
17    .twsk_prot       = &tcp_timewait_sock_ops,
18    .rsk_prot        = &tcp_request_sock_ops,
19 };

```

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

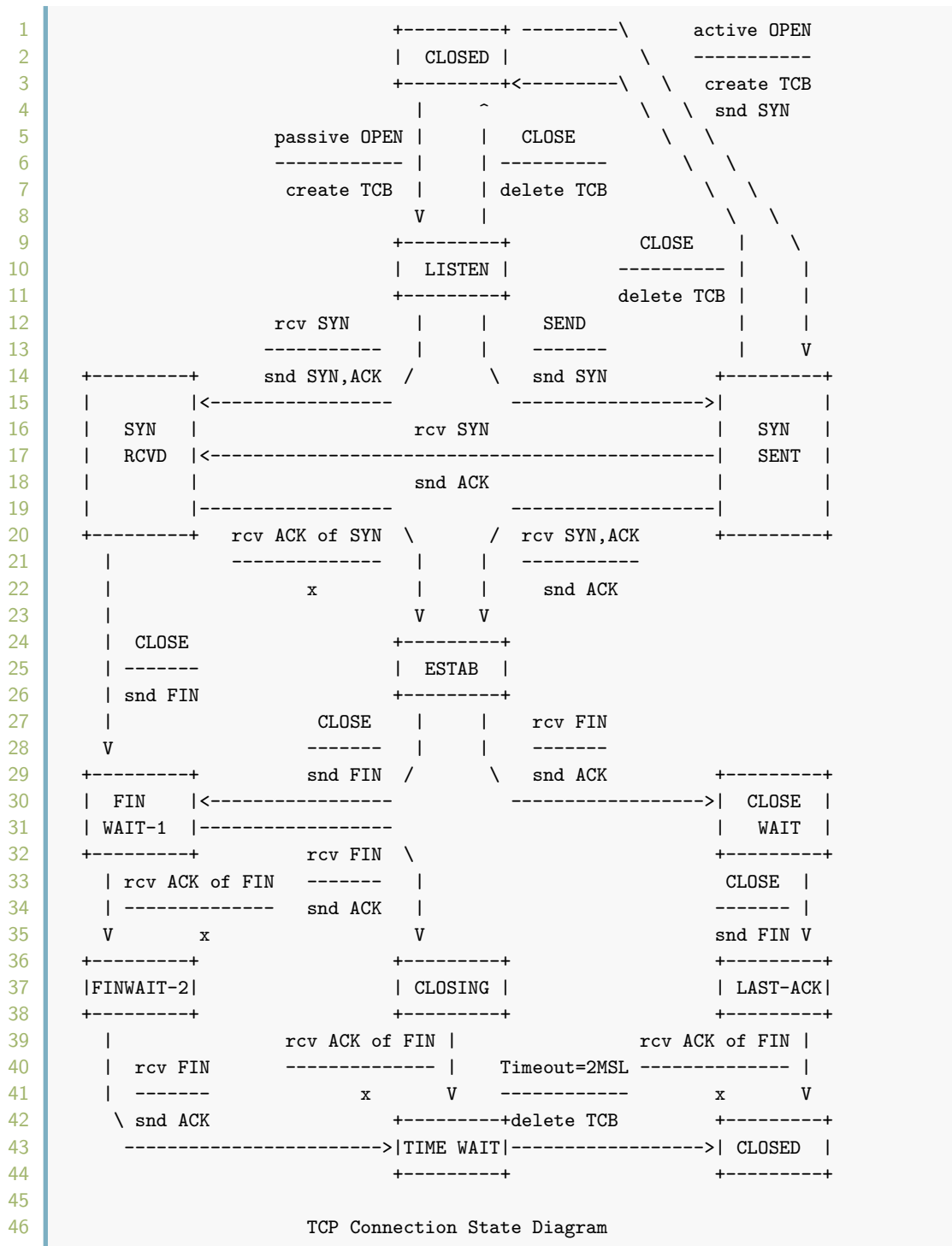
在分析 TCP 的过程中会遇到很多 RFC，在这里，我们将可能会碰到的 RFC 罗列出来，并进行一定的讨论，便于后面的分析。

### 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：



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。

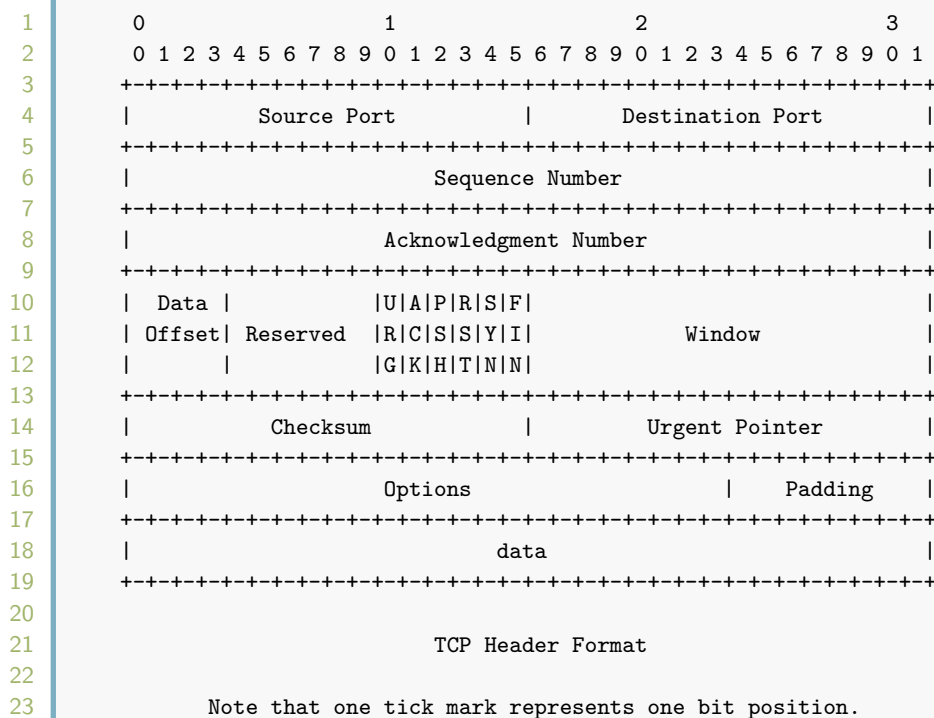
连接终止的部分也被分为了两部分进行实现，主动终止和被动终止。主动终止是上



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：



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：

```

1 End of Option List
2 +-----+
3 |00000000|
4 +-----+
5 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：

```

1      Maximum Segment Size
2      +-----+-----+-----+
3      |00000010|00000100| max seg size  |
4      +-----+-----+-----+
5      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：

```

1      +-----+-----+-----+
2      | Kind=3  |Length=3 |shift.cnt|
3      +-----+-----+-----+

```

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。

```
1 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。

```
1 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：

```
1 +-----+-----+-----+-----+
2 |Kind=8 | 10   | TS Value (TSval) |TS Echo Reply (TSecr)|
3 +-----+-----+-----+-----+
4      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  $SEG.SEQ \leq Last.ACK.sent$ （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			
2			
3	1. ESTABLISHED		ESTABLISHED
4			
5	(Close)		
6	2. FIN-WAIT-1	--> <SEQ=100><ACK=300><CTL=FIN,ACK>	--> CLOSE-WAIT
7			
8	3. FIN-WAIT-2	<-- <SEQ=300><ACK=101><CTL=ACK>	<-- CLOSE-WAIT
9			
10			(Close)
11	4. TIME-WAIT	<-- <SEQ=300><ACK=101><CTL=FIN,ACK>	<-- LAST-ACK
12			
13	5. TIME-WAIT	--> <SEQ=101><ACK=301><CTL=ACK>	--> CLOSED
14			
15	- - - - -		
16			
17	5.1. TIME-WAIT	<-- <SEQ=255><ACK=33> ... old duplicate	
18			
19	5.2 TIME-WAIT	--> <SEQ=101><ACK=301><CTL=ACK>	--> ????
20			
21	5.3 CLOSED	<-- <SEQ=301><CTL=RST>	<-- ????
22	(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：

```

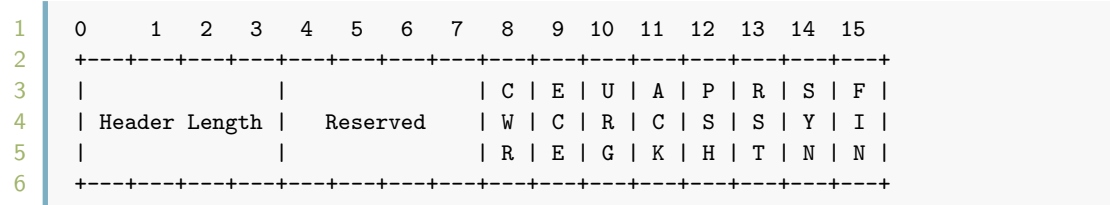
1  +-----+-----+
2  | ECN FIELD |
3  +-----+-----+
4  ECT  CE  [Obsolete] RFC 2481 names for the ECN bits.
5      0   0   Not-ECT
6      0   1   ECT(1)
7      1   0   ECT(0)
8      1   1   CE

```

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是否启用 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:



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 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。

#### 1.3.6.1 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

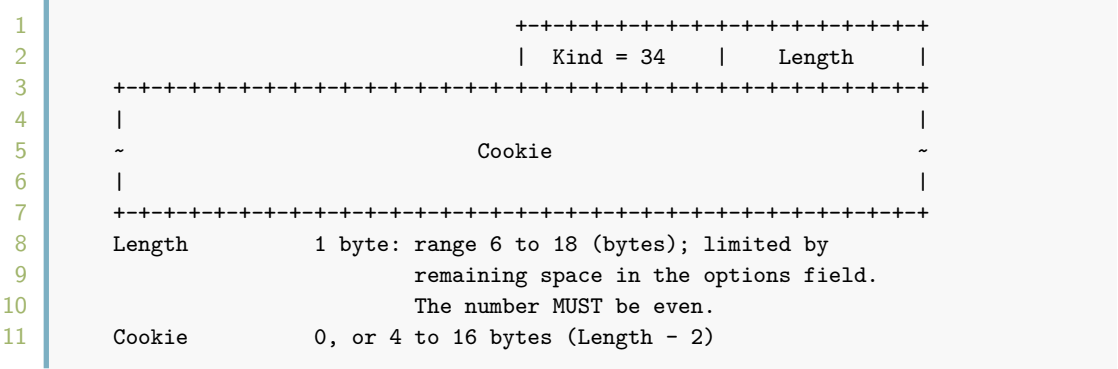
- 2. 服务器生成一个 cookie，通过 SYN-ACK 包回复给客户端
- 3. 客户端将该 cookie 缓存，并用于后续的 TCP Fast Open 连接。

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。

1.3.6.2 Fast Open 选项格式

该选项的格式如下：



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

<b>2.1</b>	<b>网络子系统数据结构架构</b>	<b>11</b>
<b>2.2</b>	<b>sock 底层数据结构</b>	<b>11</b>
2.2.1	sock_common	11
2.2.2	sock	13
2.2.3	request_sock	17
2.2.4	sk_buff	17
<b>2.3</b>	<b>inet 层相关数据结构</b>	<b>22</b>
2.3.1	inet_request_sock	22
2.3.2	inet_connection_sock_af_ops	22
2.3.3	inet_connect_sock	23
2.3.4	inet_timewait_sock	24
2.3.5	sockaddr & sockaddr_in	25
2.3.6	ip_options	26
<b>2.4</b>	<b>路由相关数据结构</b>	<b>26</b>
2.4.1	dst_entry	26
2.4.2	rtable	28
2.4.3	flowi	29
<b>2.5</b>	<b>TCP 层相关数据结构</b>	<b>29</b>
2.5.1	tcphdr	29
2.5.2	tcp_options_received	30
2.5.3	tcp_sock	31
2.5.4	tcp_request_sock	34
2.5.5	tcp_skb_cb	35
2.5.5.1	TCP_SKB_CB	35



## 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

```

1  /**
2  * struct sock_common - minimal network layer representation of sockets
3  * @skc_daddr: Foreign IPv4 addr
4  * @skc_rcv_saddr: Bound local IPv4 addr
5  * @skc_hash: hash value used with various protocol lookup tables
6  * @skc_u16hashes: two u16 hash values used by UDP lookup tables
7  * @skc_dport: placeholder for inet_dport/tw_dport
8  * @skc_num: placeholder for inet_num/tw_num
9  * @skc_family: network address family
10 * @skc_state: Connection state
11 * @skc_reuse: %SO_REUSEADDR setting
12 * @skc_reuseport: %SO_REUSEPORT setting
13 * @skc_bound_dev_if: bound device index if != 0
14 * @skc_bind_node: bind hash linkage for various protocol lookup tables
15 * @skc_portaddr_node: second hash linkage for UDP/UDP-Lite protocol
16 * @skc_prot: protocol handlers inside a network family
17 * @skc_net: reference to the network namespace of this socket
18 * @skc_node: main hash linkage for various protocol lookup tables
19 * @skc_nulls_node: main hash linkage for TCP/UDP/UDP-Lite protocol
20 * @skc_tx_queue_mapping: tx queue number for this connection
21 * @skc_flags: place holder for sk_flags
22 *      %SO_LINGER (l_onoff), %SO_BROADCAST, %SO_KEEPAIVE,
23 *      %SO_OOBINLINE settings, %SO_TIMESTAMPING settings
24 * @skc_incoming_cpu: record/match cpu processing incoming packets
25 * @skc_refcnt: reference count
26 *
27 * This is the minimal network layer representation of sockets, the header
28 * for struct sock and struct inet_timewait_sock.
29 */
30 struct sock_common {
31     /* skc_daddr and skc_rcv_saddr must be grouped on a 8 bytes aligned
32      * address on 64bit arches : cf INET_MATCH()
33      */
34     union {
35         __addrpair   skc_addrpair;
36         struct {
37             __be32   skc_daddr;
38             __be32   skc_rcv_saddr;
39         };
40     };
41     union {
42         unsigned int   skc_hash;
43         __u16          skc_u16hashes[2];
44     };
45     /* skc_dport & skc_num must be grouped as well */
46     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_thash: 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_ipu6only __sk_common.skc_ipu6only

```

```

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_rthash         __sk_common.skc_rthash
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,
157                     sk_no_check_rx : 1,
158                     sk_userlocks : 4,

```

```

159         sk_protocol : 8,
160         sk_type      : 16;
161 #define SK_PROTOCOL_MAX US_MAX
162     kmemcheck_bitfield_end(flags);
163     int      sk_wmem_queued;
164     gfp_t    sk_allocation;
165     u32      sk_pacing_rate; /* bytes per second */
166     u32      sk_max_pacing_rate;
167     netdev_features_t sk_route_caps;
168     netdev_features_t sk_route_nocaps;
169     int      sk_gso_type;
170     unsigned int sk_gso_max_size;
171     u16      sk_gso_max_segs;
172     int      sk_rcvlowat;
173     unsigned long sk_lingertime;
174     struct sk_buff_head sk_error_queue;
175     struct proto *sk_prot_creator;
176     rwlock_t sk_callback_lock;
177     int      sk_err,
178         sk_err_soft;
179     u32      sk_ack_backlog;
180     u32      sk_max_ack_backlog;
181     __u32    sk_priority;
182 #if IS_ENABLED(CONFIG_CGROUP_NET_PRIO)
183     __u32    sk_cgrp_prioidx;
184 #endif
185     struct pid *sk_peer_pid;
186     const struct cred *sk_peer_cred;
187     long      sk_rcvtimeo;
188     long      sk_sndtimeo;
189     struct timer_list sk_timer;
190     ktime_t    sk_stamp;
191     u16      sk_tsflags;
192     u32      sk_tskey;
193     struct socket *sk_socket;
194     void      *sk_user_data;
195     struct page_frag sk_frag;
196     struct sk_buff *sk_send_head;
197     __s32     sk_peek_off;
198     int      sk_write_pending;
199 #ifdef CONFIG_SECURITY
200     void      *sk_security;
201 #endif
202     __u32     sk_mark;
203 #ifdef CONFIG_CGROUP_NET_CLASSID
204     u32      sk_classid;
205 #endif
206     struct cg_proto *sk_cgrp;
207     void (*sk_state_change)(struct sock *sk);
208     void (*sk_data_ready)(struct sock *sk);
209     void (*sk_write_space)(struct sock *sk);
210     void (*sk_error_report)(struct sock *sk);
211     int (*sk_backlog_rcv)(struct sock *sk,
212         struct sk_buff *skb);
213     void (*sk_destruct)(struct sock *sk);
214 };

```

### 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。

```

1  /* struct request_sock - mini sock to represent a connection request
2  */
3  struct request_sock {
4      struct sock_common    __req_common;
5      #define rsk_refcnt      __req_common.skc_refcnt
6      #define rsk_hash        __req_common.skc_hash
7      #define rsk_listener    __req_common.skc_listener
8      #define rsk_window_clamp __req_common.skc_window_clamp
9      #define rsk_rcv_wnd     __req_common.skc_rcv_wnd
10
11     struct request_sock    *dl_next;
12     u16                    mss;
13     u8                    num_retrans; /* number of retransmits */
14     u8                    cookie_ts:1; /* syncookie: encode tcptopts in timestamp */
15     u8                    num_timeout:7; /* number of timeouts */
16     u32                    ts_recent;
17     struct timer_list      rsk_timer;
18     const struct request_sock_ops *rsk_ops;
19     struct sock            *sk;
20     u32                    *saved_syn;
21     u32                    secid;
22     u32                    peer_secid;
23 };

```

### 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

```

1  union {
2      struct {
3          /* These two members must be first. */
4          struct sk_buff    *next;
5          struct sk_buff    *prev;
6
7          union {
8              ktime_t        tstamp;
9              struct skb_mstamp skb_mstamp;
10         };
11     };
12     struct rb_node    rbnode; /* used in netem and tcp stack */
13 };

```

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

```

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

```

1  unsigned int         len,
2                       data_len;
3  __u16               mac_len,
4                       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:

```

1  #define kmemcheck_bitfield_begin(name) \
2      int name##_begin[0];
3
4  #define kmemcheck_bitfield_end(name) \
5      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。



```

1  /* Following fields are _not_ copied in __copy_skb_header()
2  * Note that queue_mapping is here mostly to fill a hole.
3  */
4  kmemcheck_bitfield_begin(flags1);
5  __u16      queue_mapping;      /* 对于多队列设备的队列关系映射          */
6  __u8       cloned:1,           /* 是否被克隆                      */
7  __u8       nohdr:1,            /* 只引用了负载                      */
8  __u8       fclone:2,           /* skbuff 克隆的情况                */
9  __u8       peeked:1,           /* peeked 表明该包已经被统计过了, 无需再次统计 */
10  __u8       head_frag:1,
11  __u8       xmit_more:1;        /* 在队列中有更多的 SKB 在等待      */
12  /* one bit hole */
13  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。

```

1  /* 在 __copy_skb_header() 中, 只需使用一个 memcpy() 即可将 headers_start/end
2  * 之间的部分克隆一份。
3  */
4  /* private: */
5  __u32      headers_start[0];
6  /* public: */
7
8  /* if you move pkt_type around you also must adapt those constants */
9  #ifdef __BIG_ENDIAN_BITFIELD
10 #define PKT_TYPE_MAX    (7 << 5)
11 #else
12 #define PKT_TYPE_MAX    7
13 #endif
14 #define PKT_TYPE_OFFSET()    offsetof(struct sk_buff, __pkt_type_offset)
15
16 __u8      __pkt_type_offset[0];
17 /* 该包的类型 */
18 __u8      pkt_type:3;
19 __u8      pfmemalloc:1;
20 /* 是否允许本地分片 (local fragmentation) */
21 __u8      ignore_df:1;
22 /* 表明该 skb 和连接的关系 */
23 __u8      nfctinfo:3;
24 /* netfilter 包追踪标记 */
25 __u8      nf_trace:1;
26 /* 驱动 (硬件) 给出来的 checksum */
27 __u8      ip_summed:2;
28 /* 允许该 socket 到队列的对应关系发生变更 */
29 __u8      ooo_okay:1;
30 /* 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 */
31 __u8      l4_hash:1;
32 /* 表明哈希值字段 hash 是通过软件栈计算出来的 */
33 __u8      sw_hash:1;
34 /* 表明 wifi_acked 是否被设置了 */
35 __u8      wifi_acked_valid:1;
36 /* 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 */

```

```

96         __u32         reserved_tailroom;
97     };
98
99     union {
100         __be16         inner_protocol; /* 协议 (封装好的) */
101         __u8           inner_ipproto;
102     };
103
104     /* 已封装的内部传输层头部 */
105     __u16             inner_transport_header;
106     /* 已封装的内部网络层头部 */
107     __u16             inner_network_header;
108     /* 已封装的内部链路层头部 */
109     __u16             inner_mac_header;
110
111     /* 驱动 (硬件) 给出的包的协议类型 */
112     __be16            protocol;
113     /* 传输层头部 */
114     __u16             transport_header;
115     /* 网络层头部 */
116     __u16             network_header;
117     /* 数据链路层头部 */
118     __u16             mac_header;
119
120     /* private: */
121     __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。

```

1     /* These elements must be at the end, see alloc_skb() for details. */
2     sk_buff_data_t    tail;
3     sk_buff_data_t    end;
4     unsigned char     *head,
5                       *data;
6     unsigned int       truesize;
7     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 inet\_request\_sock

该结构位于`/include/net/inet_sock.h`中。

这个结构的功能呢？

```

1     struct inet_request_sock {
2         struct request_sock req;
3         #define ir_loc_addr      req.__req_common.skc_rcv_saddr
4         #define ir_rmt_addr     req.__req_common.skc_daddr
5         #define ir_num          req.__req_common.skc_num
6         #define ir_rmt_port     req.__req_common.skc_dport
7         #define ir_v6_rmt_addr  req.__req_common.skc_v6_daddr
8         #define ir_v6_loc_addr  req.__req_common.skc_v6_rcv_saddr
9         #define ir_iif          req.__req_common.skc_bound_dev_if

```

```

10  #define ir_cookie      req->__req_common.skc_cookie
11  #define ireq_net       req->__req_common.skc_net
12  #define ireq_state     req->__req_common.skc_state
13  #define ireq_family    req->__req_common.skc_family
14
15  kmemcheck_bitfield_begin(flags);
16  u16      snd_wscale : 4,
17          rcv_wscale : 4,
18          tstamp_ok  : 1,
19          sack_ok     : 1,
20          wscale_ok   : 1,
21          ecn_ok      : 1,
22          acked       : 1,
23          no_srccheck: 1;
24  kmemcheck_bitfield_end(flags);
25  u32      ir_mark;
26  union {
27      struct ip_options_rcu *opt;
28      struct sk_buff *pktopts;
29  };
30  };

```

### 2.3.2 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。

```

1  struct inet_connection_sock_af_ops {
2      int      (*queue_xmit)(struct sock *sk, struct sk_buff *skb, struct flowi *fl);
3      void     (*send_check)(struct sock *sk, struct sk_buff *skb);
4      int      (*rebuild_header)(struct sock *sk);
5      void     (*sk_rx_dst_set)(struct sock *sk, const struct sk_buff *skb);
6      int      (*conn_request)(struct sock *sk, struct sk_buff *skb);
7      struct sock *(*syn_rcv_sock)(const struct sock *sk, struct sk_buff *skb,
8                                  struct request_sock *req,
9                                  struct dst_entry *dst,
10     struct request_sock *req_unhash,
11     bool *own_req);
12     u16      net_header_len;
13     u16      net_frag_header_len;
14     u16      sockaddr_len;
15     int      (*setsockopt)(struct sock *sk, int level, int optname,
16                           char __user *optval, unsigned int optlen);
17     int      (*getsockopt)(struct sock *sk, int level, int optname,
18                           char __user *optval, int __user *optlen);
19 #ifdef CONFIG_COMPAT
20     int      (*compat_setsockopt)(struct sock *sk,
21                                  int level, int optname,
22                                  char __user *optval, unsigned int optlen);
23     int      (*compat_getsockopt)(struct sock *sk,
24                                  int level, int optname,
25                                  char __user *optval, int __user *optlen);
26 #endif
27     void     (*addr2sockaddr)(struct sock *sk, struct sockaddr *);
28     int      (*bind_conflict)(const struct sock *sk,

```

```

29         const struct inet_bind_bucket *tb, bool relax);
30     void      (*mtu_reduced)(struct sock *sk);
31 };

```

### 2.3.3 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。

```

1  /* inet_connection_sock - INET connection oriented sock
2  *
3  * @icsk_accept_queue:      FIFO of established children
4  * @icsk_bind_hash:        Bind node
5  * @icsk_timeout:          Timeout
6  * @icsk_retransmit_timer: Resend (no ack)
7  * @icsk_rto:              Retransmit timeout
8  * @icsk_pmtu_cookie        Last pmtu seen by socket
9  * @icsk_ca_ops             Pluggable congestion control hook
10 * @icsk_af_ops             Operations which are AF_INET{4,6} specific
11 * @icsk_ca_state:          Congestion control state
12 * @icsk_retransmits:       Number of unrecovered [RTO] timeouts
13 * @icsk_pending:           Scheduled timer event
14 * @icsk_backoff:           Backoff
15 * @icsk_syn_retries:       Number of allowed SYN (or equivalent) retries
16 * @icsk_probes_out:        unanswered 0 window probes
17 * @icsk_ext_hdr_len:       Network protocol overhead (IP/IPv6 options)
18 * @icsk_ack:               Delayed ACK control data
19 * @icsk_mtup;              MTU probing control data
20 */
21 struct inet_connection_sock {
22     /* inet_sock has to be the first member! */
23     struct inet_sock      icsk_inet;
24     struct request_sock_queue icsk_accept_queue;
25     struct inet_bind_bucket *icsk_bind_hash;
26     unsigned long          icsk_timeout;
27     struct timer_list      icsk_retransmit_timer;
28     struct timer_list      icsk_delack_timer;
29     __u32                  icsk_rto;
30     __u32                  icsk_pmtu_cookie;
31     const struct tcp_congestion_ops *icsk_ca_ops;
32     const struct inet_connection_sock_af_ops *icsk_af_ops;
33     unsigned int            (*icsk_sync_mss)(struct sock *sk, u32 pmtu);
34     __u8                    icsk_ca_state:6,
35                             icsk_ca_setsockopt:1,
36                             icsk_ca_dst_locked:1;
37     __u8                    icsk_retransmits;
38     __u8                    icsk_pending;
39     __u8                    icsk_backoff;
40     __u8                    icsk_syn_retries;
41     __u8                    icsk_probes_out;
42     __u16                   icsk_ext_hdr_len;
43     struct {
44         __u8                pending; /* ACK is pending */
45         __u8                quick; /* Scheduled number of quick acks */
46         __u8                pingpong; /* The session is interactive */
47         __u8                blocked; /* Delayed ACK was blocked by socket lock */
48         __u32                ato; /* Predicted tick of soft clock */
49         unsigned long         timeout; /* Currently scheduled timeout */

```

```

50     __u32      lrcvtime; /* timestamp of last received data packet */
51     __u16      last_seg_size; /* Size of last incoming segment */
52     __u16      rcv_mss; /* MSS used for delayed ACK decisions */
53 } icsk_ack;
54 struct {
55     int        enabled;
56
57     /* Range of MTUs to search */
58     int        search_high;
59     int        search_low;
60
61     /* Information on the current probe. */
62     int        probe_size;
63
64     u32        probe_timestamp;
65 } icsk_mtup;
66 u32          icsk_user_timeout;
67
68     u64        icsk_ca_priv[64 / sizeof(u64)];
69 #define ICSK_CA_PRIV_SIZE      (8 * sizeof(u64))
70 };

```

### 2.3.4 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。

```

1  /* include/net/inet_timewait_sock.h
2  *
3  * 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。
4  */
5  struct inet_timewait_sock {
6      /* 此处也使用了 sock_common 结构体。
7       * Now struct sock also uses sock_common, so please just
8       * don't add nothing before this first member (__tw_common) --acme
9       */
10
11     struct sock_common    __tw_common;
12     /* 通过宏定义为 sock_common 中的结构体起一个 tw 开头的别名。 */
13     #define tw_family      __tw_common.skc_family
14     #define tw_state       __tw_common.skc_state
15     #define tw_reuse       __tw_common.skc_reuse
16     #define tw_ip6only     __tw_common.skc_ip6only
17     #define tw_bound_dev_if __tw_common.skc_bound_dev_if
18     #define tw_node        __tw_common.skc_nulls_node
19     #define tw_bind_node   __tw_common.skc_bind_node
20     #define tw_refcnt      __tw_common.skc_refcnt
21     #define tw_hash        __tw_common.skc_hash
22     #define tw_prot        __tw_common.skc_prot
23     #define tw_net         __tw_common.skc_net
24     #define tw_daddr       __tw_common.skc_daddr
25     #define tw_v6_daddr    __tw_common.skc_v6_daddr
26     #define tw_rcv_saddr    __tw_common.skc_rcv_saddr
27     #define tw_v6_rcv_saddr __tw_common.skc_v6_rcv_saddr
28     #define tw_dport       __tw_common.skc_dport
29     #define tw_num         __tw_common.skc_num
30     #define tw_cookie      __tw_common.skc_cookie
31     #define tw_dr          __tw_common.skc_tw_dr

```

```

32
33     /* 超时时间 */
34     int                tw_timeout;
35     /* 子状态, 用于区分 FIN_WAIT2 和 TIMEWAIT */
36     volatile unsigned char tw_substate;
37     /* 窗口缩放 */
38     unsigned char      tw_rcv_wscale;
39
40     /* 下面的部分都和 inet_sock 中的成员相对应的。 */
41     __be16             tw_sport;
42     kmemcheck_bitfield_begin(flags);
43     /* And these are ours. */
44     unsigned int       tw_kill        : 1,
45                          tw_transparent : 1,
46                          tw_flowlabel  : 20,
47                          tw_pad       : 2,    /* 2 bits hole */
48                          tw_tos       : 8;
49     kmemcheck_bitfield_end(flags);
50     /* 超时时器 */
51     struct timer_list   tw_timer;
52     struct inet_bind_bucket *tw_tb;
53 };

```

### 2.3.5 sockaddr & sockaddr\_in

sockaddr用于描述一个地址。

```

1  /* include/linux/socket.h */
2  struct sockaddr {
3      sa_family_t    sa_family;    /* 地址所属的协议族, AF_xxx */
4      char           sa_data[14]; /* 在协议下的地址 */
5  };

```

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。

```

1  /* include/uapi/linux/in.h
2  * 该结构体用于描述一个 Internet (IP) 套接字的地址
3  */
4  struct sockaddr_in {
5      __kernel_sa_family_t sin_family; /* 这里和 sockaddr 是对应的, 填写 IP 网络 */
6      __be16               sin_port;    /* 端口号 */
7      struct in_addr       sin_addr;    /* Internet 地址 */
8
9      /* 填充位, 为了将 sockaddr_in 填充到和 sockaddr 一样长 */
10     unsigned char         __pad[__SOCK_SIZE__ - sizeof(short int) -
11                                sizeof(unsigned short int) - sizeof(struct in_addr)];
12 };

```

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。

### 2.3.6 ip\_options

```

1  /** struct ip_options - IP Options
2  *
3  * @faddr - 保存的第一跳地址
4  * @nexthop - 保存在 LSRR 和 SSRR 的下一跳地址
5  * @is_strictroute - 严格的源路由
6  * @srr_is_hit - 包目标地址命中
7  * @is_changed - IP 校验和不合法
8  * @rr_needaddr - 需要记录出口设备的地址
9  * @ts_needtime - 需要记录时间戳
10 * @ts_needaddr - 需要记录出口设备的地址
11 */
12 struct ip_options {
13     __be32      faddr;
14     __be32      nexthop;
15     unsigned char optlen;
16     unsigned char srr;
17     unsigned char rr;
18     unsigned char ts;
19     unsigned char is_strictroute:1,
20                 srr_is_hit:1,
21                 is_changed:1,
22                 rr_needaddr:1,
23                 ts_needtime:1,
24                 ts_needaddr:1;
25     unsigned char router_alert;
26     unsigned char cipso;
27     unsigned char __pad2;
28     unsigned char __data[0];
29 };
30
31 struct ip_options_rcu {
32     struct rcu_head rcu;
33     struct ip_options opt;
34 };

```

## 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。

```

1  /* Each dst_entry has reference count and sits in some parent list(s).
2  * When it is removed from parent list, it is "freed" (dst_free).
3  * After this it enters dead state (dst->obsolete > 0) and if its refcnt
4  * is zero, it can be destroyed immediately, otherwise it is added
5  * to gc list and garbage collector periodically checks the refcnt.
6  */
7  struct dst_entry {
8     struct rcu_head rcu_head;

```



```

9      struct dst_entry    *child;
10     struct net_device    *dev;
11     struct dst_ops        *ops;
12     unsigned long        _metrics;
13     unsigned long        expires;
14     struct dst_entry    *path;
15     struct dst_entry    *from;
16     #ifdef CONFIG_XFRM
17         struct xfrm_state *xfrm;
18     #else
19         void                *__pad1;
20     #endif
21     int                    (*input)(struct sk_buff *);
22     int                    (*output)(struct net *net, struct sock *sk, struct sk_buff *skb);
23
24     unsigned short        flags;
25     #define DST_HOST        0x0001
26     #define DST_NOXFRM      0x0002
27     #define DST_NOPOLICY    0x0004
28     #define DST_NOHASH      0x0008
29     #define DST_NOCACHE     0x0010
30     #define DST_NOCOUNT     0x0020
31     #define DST_FAKE_RTABLE 0x0040
32     #define DST_XFRM_TUNNEL 0x0080
33     #define DST_XFRM_QUEUE  0x0100
34     #define DST_METADATA    0x0200
35
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, 通过该 union 能够很好的实现 `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。

```

1  struct rtable {
2      struct dst_entry  dst;
3
4      int                rt_genid;
5      unsigned int       rt_flags;
6      __u16              rt_type;
7      __u8               rt_is_input;
8      __u8               rt_uses_gateway;
9
10     int                rt_iif;
11
12     /* Info on neighbour */
13     __be32             rt_gateway;
14
15     /* Miscellaneous cached information */
16     u32                rt_pmtu;
17
18     u32                rt_table_id;
19
20     struct list_head    rt_uncached;
21     struct uncached_list *rt_uncached_list;
22 };

```

### 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。

```

1  struct flowi {
2      union {
3          struct flowi_common __fl_common;
4          struct flowi4      ip4;
5          struct flowi6      ip6;
6          struct flowidn     dn;
7      } u;
8      #define flowi_oif    u.__fl_common.flowic_oif
9      #define flowi_iif    u.__fl_common.flowic_iif
10     #define flowi_mark    u.__fl_common.flowic_mark
11     #define flowi_tos     u.__fl_common.flowic_tos
12     #define flowi_scope   u.__fl_common.flowic_scope
13     #define flowi_proto   u.__fl_common.flowic_proto
14     #define flowi_flags   u.__fl_common.flowic_flags
15     #define flowi_secid   u.__fl_common.flowic_secid
16     #define flowi_tun_key u.__fl_common.flowic_tun_key
17 } __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。

```

1  struct tcphdr {
2      __be16 source;
3      __be16 dest;
4      __be32 seq;
5      __be32 ack_seq;
6      #if defined(__LITTLE_ENDIAN_BITFIELD)
7          __u16  res1:4,
8              doff:4,
9              fin:1,
10             syn:1,
11             rst:1,
12             psh:1,
13             ack:1,
14             urg:1,
15             ece:1,
16             cwr:1;
17     #elif defined(__BIG_ENDIAN_BITFIELD)
18         __u16  doff:4,
19             res1:4,
20             cwr:1,
21             ece:1,
22             urg:1,
23             ack:1,
24             psh:1,
25             rst:1,
26             syn:1,
27             fin:1;
28     #else
29     #error "Adjust your <asm/byteorder.h> defines"
30     #endif
31     __be16 window;
32     __sum16 check;

```

```

33     __be16 urg_ptr;
34 };

```

### 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。

```

1  struct tcp_options_received {
2  /* PAWS/RTTM data */
3      long    ts_recent_stamp; /* Time we stored ts_recent (for aging)
4                               记录从接收到的段中取出时间戳设置到 ts_re
5                               用于检测 ts_recent 的有效性：如果自从该
6                               经过了超过 24 天的时间，则认为 ts_rece
7                               */
8      u32 ts_recent;           /* Time stamp to echo next
9                               下一个待发送的 TCP 段中的时间戳回显值。
10                              发送 ACK 中确认序号的段到达时，该段中的
11                              ts_recent 中。而当下一个待发送的 TCP 段
12                              SKB 中 TCP 控制块的成员 when 填入的，wh
13                              栈取系统时间变量 jiffies 的低 32 位。
14                              */
15      u32 rcv_tsval;           /* Time stamp value
16                              保存最近一次接收到对段的 TCP 段的时间戳
17                              */
18      u32 rcv_tsecr;           /* Time stamp echo reply
19                              保存最近一次接收到对段的 TCP 段的时间戳
20                              回显应答。
21                              */
22      u16     saw_tstamp : 1, /* Saw TIMESTAMP on last packet
23                              标识最近一次接收到的 TCP 段是否存在 TCP
24                              0 为无。
25                              */
26      u16     tstamp_ok : 1,  /* TIMESTAMP seen on SYN packet
27                              标识 TCP 连接是否启动时间戳选项。
28                              */
29      u16     dsack : 1,       /* D-SACK is scheduled */
30      u16     wscale_ok : 1,   /* Wscale seen on SYN packet */
31      u16     sack_ok : 4,     /* SACK seen on SYN packet */
32      u16     snd_wscale : 4,  /* Window scaling received from sender */
33      u16     rcv_wscale : 4;  /* Window scaling to send to receiver */
34      u8 num_sacks;            /* Number of SACK blocks */
35      u16 user_mss;            /* mss requested by user in ioctl */
36      u16 mss_clamp;           /* Maximal mss, negotiated at connection setup */
37 };

```

### 2.5.3 tcp\_sock

该数据结构位于 `/include/linux/tcp.h` 中。

该数据结构是 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。

```

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

```

```

6
7  /*
8  * Header prediction flags
9  * 0x5?10 << 16 + snd_wnd in net byte order
10 */
11  __be32  pred_flags;
12
13  /*
14  * RFC793 variables by their proper names. This means you can
15  * read the code and the spec side by side (and laugh ...)
16  * See RFC793 and RFC1122. The RFC writes these in capitals.
17  */
18  u64 bytes_received; /* RFC4898 tcpEStatsAppHCThruOctetsReceived
19                      * sum(delta(rcv_nxt)), or how many bytes
20                      * were acked.
21                      */
22  u32 segs_in; /* RFC4898 tcpEStatsPerfSegsIn
23              * total number of segments in.
24              */
25  u32 rcv_nxt; /* 下一个待接收的字节号 */
26  u32 copied_seq; /* Head of yet unread data */
27  u32 rcv_wup; /* rcv_nxt on last window update sent */
28  u32 snd_nxt; /* 下一个待发送的序列 */
29  u32 segs_out; /* RFC4898 tcpEStatsPerfSegsOut
30              * The total number of segments sent.
31              */
32  u64 bytes_acked; /* RFC4898 tcpEStatsAppHCThruOctetsAcked
33                  * sum(delta(snd_una)), or how many bytes
34                  * were acked.
35                  */
36  struct u64_stats_sync syncp; /* protects 64bit vars (cf tcp_get_info()) */
37
38  u32 snd_una; /* 待 ACK 的第一个字节的序号 */
39  u32 snd_sml; /* Last byte of the most recently transmitted small packet */
40  u32 rcv_tstamp; /* timestamp of last received ACK (for keepalives) */
41  u32 lsndtime; /* timestamp of last sent data packet (for restart window) */
42  u32 last_oow_ack_time; /* timestamp of last out-of-window ACK */
43
44  u32 tsoffset; /* timestamp offset */
45
46  struct list_head tsq_node; /* anchor in tsq_tasklet.head list */
47  unsigned long tsq_flags;
48
49  /* Data for direct copy to user */
50  struct {
51      struct sk_buff_head prequeue;
52      struct task_struct *task;
53      struct msghdr *msg;
54      int memory;
55      int len;
56  } ucopy;
57
58  u32 snd_wll; /* Sequence for window update */
59  u32 snd_wnd; /* 发送窗口 */
60  u32 max_window; /* Maximal window ever seen from peer */
61  u32 mss_cache; /* Cached effective mss, not including SACKS */
62
63  u32 window_clamp; /* Maximal window to advertise */

```

```

64     u32 rcv_ssthresh;    /* Current window clamp          */
65
66     /* Information of the most recently (s)acked skb */
67     struct tcp_rack {
68         struct skb_mstamp mstamp; /* (Re)sent time of the skb */
69         u8 advanced; /* mstamp advanced since last lost marking */
70         u8 reord;    /* reordering detected */
71     } rack;
72     u16 advmss;      /* Advertised MSS          */
73     u8  unused;
74     u8  nonagle      : 4, /* Disable Nagle algorithm?          */
75         thin_lto     : 1, /* Use linear timeouts for thin streams */
76         thin_dupack   : 1, /* Fast retransmit on first dupack      */
77         repair        : 1,
78         frto          : 1; /* F-RTO (RFC5682) activated in CA_Loss */
79     u8 repair_queue;
80     u8 do_early_retrans:1, /* Enable RFC5827 early-retransmit */
81         syn_data:1, /* SYN includes data */
82         syn_fastopen:1, /* SYN includes Fast Open option */
83         syn_fastopen_exp:1, /* SYN includes Fast Open exp. option */
84         syn_data_acked:1, /* data in SYN is acked by SYN-ACK */
85         save_syn:1, /* Save headers of SYN packet */
86         is_cwnd_limited:1; /* forward progress limited by snd_cwnd? */
87     u32 tlp_high_seq; /* snd_nxt at the time of TLP retransmit. */
88
89     /* RTT measurement */
90     u32 rtt_us; /* smoothed round trip time < 3 in usecs */
91     u32 mdev_us; /* medium deviation */
92     u32 mdev_max_us; /* maximal mdev for the last rtt period */
93     u32 rttvar_us; /* smoothed mdev_max */
94     u32 rtt_seq; /* sequence number to update rttvar */
95     struct rtt_meas {
96         u32 rtt, ts; /* RTT in usec and sampling time in jiffies. */
97     } rtt_min[3];
98
99     u32 packets_out; /* Packets which are "in flight" */
100    u32 retrans_out; /* Retransmitted packets out */
101    u32 max_packets_out; /* max packets_out in last window */
102    u32 max_packets_seq; /* right edge of max_packets_out flight */
103
104    u16 urg_data; /* Saved octet of OOB data and control flags */
105    u8  ecn_flags; /* ECN status bits. */
106    u8  keepalive_probes; /* num of allowed keep alive probes */
107    u32 reordering; /* Packet reordering metric. */
108    u32 snd_up; /* Urgent pointer */
109
110    /*
111     * Options received (usually on last packet, some only on SYN packets).
112     */
113    struct tcp_options_received rx_opt;
114
115    /*
116     * Slow start and congestion control (see also Nagle, and Karn & Partridge)
117     */
118    u32 snd_ssthresh; /* Slow start size threshold */
119    u32 snd_cwnd; /* Sending congestion window */
120    u32 snd_cwnd_cnt; /* Linear increase counter */
121    u32 snd_cwnd_clamp; /* Do not allow snd_cwnd to grow above this */
122    u32 snd_cwnd_used;

```

```

123     u32 snd_cwnd_stamp;
124     u32 prior_cwnd; /* Congestion window at start of Recovery. */
125     u32 prr_delivered; /* Number of newly delivered packets to
126                        * receiver in Recovery. */
127     u32 prr_out; /* Total number of pkts sent during Recovery. */
128
129     u32 rcv_wnd; /* Current receiver window */
130     u32 write_seq; /* Tail(+1) of data held in tcp send buffer */
131     u32 notsent_lowat; /* TCP_NOTSENT_LOWAT */
132     u32 pushed_seq; /* Last pushed seq, required to talk to windows */
133     u32 lost_out; /* Lost packets */
134     u32 sacked_out; /* SACK'd packets */
135     u32 fackets_out; /* FACK'd packets */
136
137     /* from STCP, retrans queue hinting */
138     struct sk_buff* lost_skb_hint;
139     struct sk_buff *retransmit_skb_hint;
140
141     /* 000 segments go in this list. Note that socket lock must be held,
142      * as we do not use sk_buff_head lock.
143      */
144     struct sk_buff_head out_of_order_queue;
145
146     /* SACKs data, these 2 need to be together (see tcp_options_write) */
147     struct tcp_sack_block duplicate_sack[1]; /* D-SACK block */
148     struct tcp_sack_block selective_acks[4]; /* The SACKS themselves */
149
150     struct tcp_sack_block rcv_sack_cache[4];
151
152     struct sk_buff *highest_sack; /* skb just after the highest
153                                  * skb with SACKed bit set
154                                  * (validity guaranteed only if
155                                  * sacked_out > 0)
156                                  */
157
158     int lost_cnt_hint;
159     u32 retransmit_high; /* L-bits may be on up to this seqno */
160
161     u32 prior_ssthresh; /* ssthresh saved at recovery start */
162     u32 high_seq; /* snd_nxt at onset of congestion */
163
164     u32 retrans_stamp; /* Timestamp of the last retransmit,
165                       * also used in SYN-SENT to remember stamp of
166                       * the first SYN. */
167     u32 undo_marker; /* snd_una upon a new recovery episode. */
168     int undo_retrans; /* number of undoable retransmissions. */
169     u32 total_retrans; /* Total retransmits for entire connection */
170
171     u32 urg_seq; /* Seq of received urgent pointer */
172     unsigned int keepalive_time; /* time before keep alive takes place */
173     unsigned int keepalive_intvl; /* time interval between keep alive probes */
174
175     int linger2;
176
177     /* Receiver side RTT estimation */
178     struct {
179         u32 rtt;
180         u32 seq;
181         u32 time;

```

```

182     } rcv_rtt_est;
183
184     /* Receiver queue space */
185     struct {
186         int space;
187         u32 seq;
188         u32 time;
189     } rcvq_space;
190
191     /* TCP-specific MTU probe information. */
192     struct {
193         u32 probe_seq_start;
194         u32 probe_seq_end;
195     } mtu_probe;
196     u32 mtu_info; /* We received an ICMP_FRAG_NEEDED / ICMPV6_PKT_TOOBIG
197                  * while socket was owned by user.
198                  */
199
200     #ifdef CONFIG_TCP_MD5SIG
201     /* TCP AF-Specific parts; only used by MD5 Signature support so far */
202     const struct tcp_sock_af_ops *af_specific;
203
204     /* TCP MD5 Signature Option information */
205     struct tcp_md5sig_info __rcu *md5sig_info;
206     #endif
207
208     /* TCP fastopen related information */
209     struct tcp_fastopen_request *fastopen_req;
210     /* fastopen_rsk points to request_sock that resulted in this big
211      * socket. Used to retransmit SYNACKs etc.
212      */
213     struct request_sock *fastopen_rsk;
214     u32 *saved_syn;
215 };

```

#### 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：

```
1 #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：

```
1 struct tcp_skb_cb {
2     __u32      seq;          /* 起始序号 */
3     __u32      end_seq;      /* SEQ + FIN + SYN + datalen */
4     union {
5         /* Note : tcp_tw_isn is used in input path only
6          * (isn chosen by tcp_timewait_state_process())
7          *
8          * tcp_gso_segs/size are used in write queue only,
9          * cf tcp_skb_pcount()/tcp_skb_mss()
10        */
11        __u32      tcp_tw_isn;
12        struct {
13            u16      tcp_gso_segs;
14            u16      tcp_gso_size;
15        };
16    };
17     __u8      tcp_flags;     /* TCP 头部的标志位 */
18
19     __u8      sacked;        /* SACK/ACK 标志位 . */
20 }
```

紧接着，定义了一些宏作为标志

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

接下来又继续定义 TCP 相关的位。

```
1     __u8      ip_dsfield;    /* IPv4 tos or IPv6 dsfield */
2     /* 1 byte hole */
3     __u32      ack_seq;      /* ACK 的序号 */
4     union {
5         struct inet_skb_parm h4;
6         #if IS_ENABLED(CONFIG_IPV6)
7             struct inet6_skb_parm h6;
8         #endif
9     } header;                /* For incoming frames */
```

```
10  };
```

## CHAPTER 3

## TCP 建立连接过程

### Contents

<b>3.1 TCP 主动打开-客户</b>	<b>37</b>
3.1.1 基本流程	37
3.1.2 第一次握手: 构造并发送 SYN 包	37
3.1.2.1 tcp_v4_connect	37
3.1.2.2 tcp_connect	40
3.1.3 第二次握手: 接收 SYN+ACK 包	41
3.1.4 第三次握手——发送 ACK 包	47
3.1.4.1 tcp_send_ack	47
3.1.5 tcp_transmit_skb	48
3.1.6 tcp_select_window(struct sk_buff *skb)	49
3.1.6.1 代码分析	49
<b>3.2 TCP 被动打开-服务器</b>	<b>52</b>
3.2.1 基本流程	52
3.2.2 第一次握手: 接受 SYN 段	53
3.2.2.1 第一次握手函数调用关系	53
3.2.2.2 tcp_v4_do_rcv	53
3.2.2.3 tcp_v4_cookie_check	54
3.2.2.4 tcp_rcv_state_process	54
3.2.2.5 tcp_v4_conn_request && tcp_conn_request	56
3.2.2.6 inet_csk_reqsk_queue_add	59
3.2.2.7 inet_csk_reqsk_queue_hash_add	60
3.2.3 第二次握手: 发送 SYN+ACK 段	60
3.2.3.1 第二次函数调用关系	60
3.2.3.2 tcp_v4_send_synack	60

3.2.3.3	tcp_make_synack . . . . .	62
3.2.4	第三次握手: 接收 ACK 段 . . . . .	64
3.2.4.1	第三次握手函数调用关系图 . . . . .	64
3.2.4.2	tcp_v4_do_rcv . . . . .	64
3.2.4.3	tcp_v4_cookie_check . . . . .	65
3.2.4.4	tcp_child_process . . . . .	66
3.2.4.5	tcp_rcv_state_process . . . . .	66

## 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 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。

```

1  /*
2  Location:
3
4      net/ipv4/tcp_ipv4.c
5
6  Function:
7
8      这个函数会初始化一个发送用的连接。
9
10 Parameters:
11
12     sk          : 传输控制块
13     uaddr       : 通用地址结构, 包含所属协议字段和相应的地址字段。
14     addrlen     :
15 */
16 int tcp_v4_connect(struct sock *sk, struct sockaddr *uaddr, int addr_len)
17 {
18     struct sockaddr_in *usin = (struct sockaddr_in *)uaddr;
19     struct inet_sock *inet = inet_sk(sk);
20     struct tcp_sock *tp = tcp_sk(sk);
21     __be16 orig_sport, orig_dport;
22     __be32 daddr, nexthop;
23     struct flowi4 *fl4;
24     struct rtable *rt;
25     int err;
26     struct ip_options_rcu *inet_opt;
27
```

```

28      /* 检验目的地址长度是否有效 */
29      if (addr_len < sizeof(struct sockaddr_in))
30          return -EINVAL;
31
32      /* 检验协议族是否正确 */
33      if (usin->sin_family != AF_INET)
34          return -EAFNOSUPPORT;
35
36      /* 将下一跳地址和目的地址暂时设置为用户传入的 IP 地址 */
37      nexthop = daddr = usin->sin_addr.s_addr;
38      inet_opt = rcu_dereference_protected(inet->inet_opt,
39                                          sock_owned_by_user(sk));
40      /* 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 */
41      if (inet_opt && inet_opt->opt.srr) {
42          if (!daddr)
43              return -EINVAL;
44          nexthop = inet_opt->opt.faddr;
45      }

```

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。

```

1      orig_sport = inet->inet_sport;
2      orig_dport = usin->sin_port;
3      fl4 = &inet->cork.fl.u.ip4;
4      /* 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
5         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。*/
6      rt = ip_route_connect(fl4, nexthop, inet->inet_saddr,
7                           RT_CONN_FLAGS(sk), sk->sk_bound_dev_if,
8                           IPPROTO_TCP,
9                           orig_sport, orig_dport, sk);
10     if (IS_ERR(rt)) {
11         err = PTR_ERR(rt);
12         if (err == -ENETUNREACH)
13             IP_INC_STATS(sock_net(sk), IPSTATS_MIB_OUTNOROUTES);
14         return err;
15     }
16
17     /*TCP 不能使用类型为组播或多播的路由缓存项 */
18     if (rt->rt_flags & (RTCF_MULTICAST | RTCF_BROADCAST)) {
19         ip_rt_put(rt);
20         return -ENETUNREACH;
21     }
22
23     /* 如果没有开启源路由功能, 则采用查找到的缓存项 */
24     if (!inet_opt || !inet_opt->opt.srr)
25         daddr = fl4->daddr;
26
27     /* 如果没有设置源地址, 则设置为缓存项中的源地址 */
28     if (!inet->inet_saddr)
29         inet->inet_saddr = fl4->saddr;
30     sk_rcv_saddr_set(sk, inet->inet_saddr);
31
32     /* 如果该传输控制块的时间戳已被使用过, 则重置各状态 */
33     if (tp->rx_opt.ts_recent_stamp && inet->inet_daddr != daddr) {
34         /* Reset inherited state */
35         tp->rx_opt.ts_recent = 0;

```

```

36         tp->rx_opt.ts_recent_stamp = 0;
37         if (likely(!tp->repair))
38             tp->write_seq = 0;
39     }
40
41     /* 在启用了 tw_recycle 的情况下, 重设时间戳 */
42     if (tcp_death_row.sysctl_tw_recycle &&
43         !tp->rx_opt.ts_recent_stamp && fl4->daddr == daddr)
44         tcp_fetch_timewait_stamp(sk, &rt->dst);
45
46     /* 设置传输控制块 */
47     inet->inet_dport = usin->sin_port;
48     sk_daddr_set(sk, daddr);
49
50     inet_csk(sk)->icsk_ext_hdr_len = 0;
51     if (inet_opt)
52         inet_csk(sk)->icsk_ext_hdr_len = inet_opt->opt.optlen;
53
54     /* 设置 MSS 大小 */
55     tp->rx_opt.mss_clamp = TCP_MSS_DEFAULT;
56
57     /* Socket identity is still unknown (sport may be zero).
58      * However we set state to SYN-SENT and not releasing socket
59      * lock select source port, enter ourselves into the hash tables and
60      * complete initialization after this.
61      */
62     /* 将 TCP 的状态设置为 SYN_SENT */
63     tcp_set_state(sk, TCP_SYN_SENT);
64     err = inet_hash_connect(&tcp_death_row, sk);
65     if (err)
66         goto failure;
67
68     sk_set_txhash(sk);
69
70     /* 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 中保存的路由 */
71     rt = ip_route_newports(fl4, rt, orig_sport, orig_dport,
72                           inet->inet_sport, inet->inet_dport, sk);
73     if (IS_ERR(rt)) {
74         err = PTR_ERR(rt);
75         rt = NULL;
76         goto failure;
77     }
78     /* 将目的地址提交到套接字 */
79     sk->sk_gso_type = SKB_GSO_TCPV4;
80     sk_setup_caps(sk, &rt->dst);
81
82     /* 如果没有设置序号, 则计算初始序号 */
83     if (!tp->write_seq && likely(!tp->repair))
84         tp->write_seq = secure_tcp_sequence_number(inet->inet_saddr,
85                                                    inet->inet_daddr,
86                                                    inet->inet_sport,
87                                                    usin->sin_port);
88
89     /* 计算 IP 首部的 id 域的值 */
90     inet->inet_id = tp->write_seq ^ jiffies;
91
92     /* 调用 tcp_connect 构造并发送 SYN 包 */
93     err = tcp_connect(sk);
94

```

```

95     rt = NULL;
96     if (err)
97         goto failure;
98
99     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。

```

1 failure:
2     /* 将状态设定为 TCP_CLOSE, 释放端口, 并返回错误值。
3     */
4     tcp_set_state(sk, TCP_CLOSE);
5     ip_rt_put(rt);
6     sk->sk_route_caps = 0;
7     inet->inet_dport = 0;
8     return err;

```

### 3.1.2.2 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。

```

1  /*
2  Location:
3
4      net/ipv4/tcp_output.c
5
6  Function:
7
8      该函数用于构造并发送 SYN 包。
9
10 Parameter:
11
12     sk: 传输控制块。
13
14     */
15 int tcp_connect(struct sock *sk)
16 {
17     struct tcp_sock *tp = tcp_sk(sk);
18     struct sk_buff *buff;
19     int err;
20
21     /* 初始化 tcp 连接 */
22     tcp_connect_init(sk);
23
24     if (unlikely(tp->repair)) { //what does it mean repair?
25         /* 如果 repair 位被置 1, 那么结束 TCP 连接 */
26         tcp_finish_connect(sk, NULL);
27         return 0;
28     }
29
30     /* 分配一个 sk_buff */
31     buff = sk_stream_alloc_skb(sk, 0, sk->sk_allocation, true);
32     if (unlikely(!buff))
33         return -ENOMEM;
34

```

```

35      /* 初始化 skb, 并自增 write_seq 的值 */
36      tcp_init_nondata_skb(buff, tp->write_seq++, TCPHDR_SYN);
37      /* 设置时间戳 */
38      tp->retrans_stamp = tcp_time_stamp;
39      /* 将当前的 sk_buff 添加到发送队列中 */
40      tcp_connect_queue_skb(sk, buff);
41      /* ECN 支持 */
42      tcp_ecn_send_syn(sk, buff);
43
44      /* 发送 SYN 包, 这里同时还考虑了 Fast Open 的情况 */
45      err = tp->fastopen_req ? tcp_send_syn_data(sk, buff) :
46          tcp_transmit_skb(sk, buff, 1, sk->sk_allocation);
47      if (err == -ECONNREFUSED)
48          return err;
49
50      /* We change tp->snd_nxt after the tcp_transmit_skb() call
51       * in order to make this packet get counted in tcpOutSegs.
52       */
53      tp->snd_nxt = tp->write_seq;
54      tp->pushed_seq = tp->write_seq;
55      TCP_INC_STATS(sock_net(sk), TCP_MIB_ACTIVEOPENS);
56
57      /* 设定超时重传定时器 */
58      inet_csk_reset_xmit_timer(sk, ICSK_TIME_RETRANS,
59                              inet_csk(sk)->icsk_rto, TCP_RTO_MAX);
60      return 0;
61  }

```

### 3.1.3 第二次握手：接收 SYN+ACK 包

`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 3.1.2.1 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
8      This function implements the receiving procedure of RFC 793 for
9      all states except ESTABLISHED and TIME_WAIT.
10     It's called from both tcp_v4_rcv and tcp_v6_rcv and should be
11     address independent.
12
13  Paramater:
14
15     sk: 传输控制块。
16     skb: 缓存块。
17  */
18  int tcp_rcv_state_process(struct sock *sk, struct sk_buff *skb)
19  {
20      struct tcp_sock *tp = tcp_sk(sk);
21      struct inet_connection_sock *icsk = inet_csk(sk);

```



```

22     const struct tcphdr *th = tcp_hdr(skb);
23     struct request_sock *req;
24     int queued = 0;
25     bool acceptable;
26
27     tp->rx_opt.saw_tstamp = 0;
28
29     switch (sk->sk_state) {
30     case TCP_CLOSE:
31         /* CLOSE 状态的处理代码 */
32
33     case TCP_LISTEN:
34         /* LISTEN 状态的处理代码 */
35
36     case TCP_SYN_SENT:
37         /* 处理接收到的数据段 */
38         queued = tcp_rcv_synsent_state_process(sk, skb, th);
39         if (queued >= 0)
40             return queued;
41
42         /* 处理紧急数据并检测是否有数据需要发送 */
43         tcp_urg(sk, skb, th);
44         __kfree_skb(skb);
45         tcp_data_snd_check(sk);
46         return 0;
47     }
48
49     /* 处理其他情况的代码 */
50 }

```

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。

```

1  /*
2  Location:
3
4      net/ipv4/tcp_input.c
5
6  Function:
7
8      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 if (tp->rx_opt.saw_tstamp && tp->rx_opt.rcv_tsecr)
28     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        */
11        if (!after(TCP_SKB_CB(skb)->ack_seq, tp->snd_una) ||
12            after(TCP_SKB_CB(skb)->ack_seq, tp->snd_nxt))
13            goto reset_and_undo;
14
15        if (tp->rx_opt.saw_tstamp && tp->rx_opt.rcv_tsecr &&
16            !between(tp->rx_opt.rcv_tsecr, tp->retrans_stamp,
17                    tcp_time_stamp)) {
18            NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_PAWSACTIVEREJECTED);
19            goto reset_and_undo;
20        }

```

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。

```

1         /* 此时, ACK 已经被接受了
2          *
3          * "If the RST bit is set
4          *   If the ACK was acceptable then signal the user "error:
5          *   connection reset", drop the segment, enter CLOSED state,
6          *   delete TCB, and return."
7          */
8
9         if (th->rst) {
10             tcp_reset(sk);
11             goto discard;
12         }

```

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。

```

1         /* rfc793:
2          * "fifth, if neither of the SYN or RST bits is set then
3          *   drop the segment and return."
4          *
5          * See note below!
6          *
7          * --ANK(990513)
8          */
9         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。

```

1      /* rfc793:
2      *   "If the SYN bit is on ...
3      *   are acceptable then ...
4      *   (our SYN has been ACKed), change the connection
5      *   state to ESTABLISHED..."
6      */
7
8      tcp_ecn_rcv_synack(tp, th);
9
10     /* 初始化与窗口有关的参数 */
11     tcp_init_wl(tp, TCP_SKB_CB(skb)->seq);
12     tcp_ack(sk, skb, FLAG_SLOWPATH);
13
14     /* Ok.. it's good. Set up sequence numbers and
15     * move to established.
16     */
17     tp->rcv_nxt = TCP_SKB_CB(skb)->seq + 1;
18     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，我们会在 1.3.2 中详细讨论。接下来手动设定了窗口缩放相关的参数，使得缩放不生效。

```

1      /* RFC1323: The window in SYN & SYN/ACK segments is
2      * never scaled.
3      */
4      tp->snd_wnd = ntohs(th->window);
5
6      if (!tp->rx_opt.wscale_ok) {
7          tp->rx_opt.snd_wscale = tp->rx_opt.rcv_wscale = 0;
8          tp->window_clamp = min(tp->window_clamp, 65535U);
9      }

```

根据时间戳选项，设定相关字段及 TCP 头部长度。

```

1      if (tp->rx_opt.saw_tstamp) {
2          tp->rx_opt.tstamp_ok = 1;
3          tp->tcp_header_len =
4              sizeof(struct tcphdr) + TCPOLEN_TSTAMP_ALIGNED;
5          tp->advms = TCPOLEN_TSTAMP_ALIGNED;
6          tcp_store_ts_recent(tp);
7      } else {
8          tp->tcp_header_len = sizeof(struct tcphdr);
9      }

```

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，FACK 用来做重传过程中的拥塞控制。

```

1         if (tcp_is_sack(tp) && sysctl_tcp_fack)
2             tcp_enable_fack(tp);

```

最后初始化 MTU、MSS 等参数并完成 TCP 连接过程。

```

1         tcp_mtup_init(sk);
2         tcp_sync_mss(sk, icsk->icsk_pmtu_cookie);
3         tcp_initialize_rcv_mss(sk);
4
5         /* Remember, tcp_poll() does not lock socket!
6          * Change state from SYN-SENT only after copied_seq
7          * is initialized. */
8         tp->copied_seq = tp->rcv_nxt;
9
10        smp_mb();
11        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。

```

1         if ((tp->syn_fastopen || tp->syn_data) &&
2             tcp_rcv_fastopen_synack(sk, skb, &foc))
3             return -1;
4
5         /* 根据情况进入延迟确认模式 */
6         if (sk->sk_write_pending ||
7             icsk->icsk_accept_queue.rskq_defer_accept ||
8             icsk->icsk_ack.pingpong) {
9             /* Save one ACK. Data will be ready after
10              * several ticks, if write_pending is set.
11              *
12              * It may be deleted, but with this feature tcpdumps
13              * look so _wonderfully_ clever, that I was not able
14              * to stand against the temptation 8)      --ANK
15              */
16             inet_csk_schedule_ack(sk);
17             icsk->icsk_ack.lrcvtime = tcp_time_stamp;
18             tcp_enter_quickack_mode(sk);
19             inet_csk_reset_xmit_timer(sk, ICSK_TIME_DACK,
20                                     TCP_DELACK_MAX, TCP_RTO_MAX);
21
22         discard:
23             __kfree_skb(skb);
24             return 0;
25         } else {
26             /* 回复 ACK 包 */
27             tcp_send_ack(sk);
28         }
29         return -1;
30     }

```

最后是一些异常情况的处理。

```

1         /* 进入该分支意味着包中不包含 ACK */
2
3         if (th->rst) {
4             /* rfc793:
5              * "If the RST bit is set

```

```

6          *
7          *      Otherwise (no ACK) drop the segment and return."
8          * 如果收到了 RST 包, 则直接丢弃并返回。
9          */
10
11         goto discard_and_undo;
12     }
13
14     /* PAWS 检查 */
15     if (tp->rx_opt.ts_recent_stamp && tp->rx_opt.saw_tstamp &&
16         tcp_paws_reject(&tp->rx_opt, 0))
17         goto discard_and_undo;
18
19     /* 仅有 SYN 而无 ACK 的处理 */
20     if (th->syn) {
21         /* We see SYN without ACK. It is attempt of
22          * simultaneous connect with crossed SYNs.
23          * Particularly, it can be connect to self.
24          */
25         tcp_set_state(sk, TCP_SYN_RECV);
26
27         /* 下面的处理和前面几乎一样 */
28         if (tp->rx_opt.saw_tstamp) {
29             tp->rx_opt.tstamp_ok = 1;
30             tcp_store_ts_recent(tp);
31             tp->tcp_header_len =
32                 sizeof(struct tcphdr) + TCPOLEN_TSTAMP_ALIGNED;
33         } else {
34             tp->tcp_header_len = sizeof(struct tcphdr);
35         }
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_wll = 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
56         #if 0
57         /* Note, we could accept data and URG from this segment.
58          * There are no obstacles to make this (except that we must
59          * either change tcp_recvmg() to prevent it from returning data
60          * before 3WHS completes per RFC793, or employ TCP Fast Open).
61          *
62          * However, if we ignore data in ACKless segments sometimes,
63          * we have no reasons to accept it sometimes.
64          * 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 }

```

### 3.1.4 第三次握手——发送 ACK 包

#### 3.1.4.1 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。

```

1  /* 该函数用于发送 ACK, 并更新窗口的大小 */
2  void tcp_send_ack(struct sock *sk)
3  {
4      struct sk_buff *buff;
5
6      /* 如果当前的套接字已经被关闭了, 那么直接返回。 */
7      if (sk->sk_state == TCP_CLOSE)
8          return;
9
10     tcp_ca_event(sk, CA_EVENT_NON_DELAYED_ACK);
11
12     /* We are not putting this on the write queue, so
13     * tcp_transmit_skb() will set the ownership to this
14     * sock.
15     * 为数据包分配空间
16     */
17     buff = alloc_skb(MAX_TCP_HEADER, sk_gfp_atomic(sk, GFP_ATOMIC));
18     if (!buff) {
19         inet_csk_schedule_ack(sk);
20         inet_csk(sk)->icsk_ack.ato = TCP_ATO_MIN;
21         inet_csk_reset_xmit_timer(sk, ICSK_TIME_DACK,
22                                 TCP_DELACK_MAX, TCP_RTO_MAX);
23         return;
24     }
25
26     /* 初始化 ACK 包 */
27     skb_reserve(buff, MAX_TCP_HEADER);

```

```

28     tcp_init_nondata_skb(buff, tcp_acceptable_seq(sk), TCPHDR_ACK);
29
30     /* We do not want pure acks influencing TCP Small Queues or fq/pacing
31      * too much.
32      * SKB_TRUESIZE(max(1 .. 66, MAX_TCP_HEADER)) is unfortunately ~784
33      * We also avoid tcp_wfree() overhead (cache line miss accessing
34      * tp->tsq_flags) by using regular sock_wfree()
35      */
36     skb_set_tcp_pure_ack(buff);
37
38     /* 添加时间戳并发送 ACK 包 */
39     skb_mstamp_get(&buff->skb_mstamp);
40     tcp_transmit_skb(sk, buff, 0, sk_gfp_atomic(sk, GFP_ATOMIC));
41 }

```

### 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 中为头部留出空间。
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)
24 {
25     skb->transport_header = skb->data - skb->head;
26 }
27 /* 选择发送窗口的大小
28 */
29 tcp_select_window(sk);
30 tcp_urg_mode(tp);
31 before(tcb->seq, tp->snd_up);
32 tcp_options_write((__be32 *) (th + 1), tp, &opts);
33 tcp_event_ack_sent(sk, tcp_skb_pcount(skb));
34 tcp_event_data_sent(tp, sk);
35 queue_xmit();
36 tcp_enter_cwr(sk);
37 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

```

1  static u16 tcp_select_window(struct sock *sk)
2  {
3      struct tcp_sock *tp = tcp_sk(sk);
4      u32 old_win = tp->rcv_wnd;
5      u32 cur_win = tcp_receive_window(tp);
6      u32 new_win = __tcp_select_window(sk);
7      /* old_win 是接收方窗口的大小。
8       * cur_win 当前的接收窗口大小。
9       * new_win 是新选择出来的窗口大小。
10     */
11
12     /* 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。
13      * 这是 IEEE 强烈不建议的一种行为。
14     */
15     if (new_win < cur_win) {
16         /* Danger Will Robinson!
17          * Don't update rcv_wup/rcv_wnd here or else
18          * we will not be able to advertise a zero
19          * window in time. --DaveM
20          *
21          * Relax Will Robinson.
22          */
23         if (new_win == 0)
24             NET_INC_STATS(sock_net(sk),
25                             LINUX_MIB_TCPWANTZEROWINDOWADV);
26         /* 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
27          * 的 1<<tp->rx_opt.rcv_wscale 的整数倍。
28          */
29         new_win = ALIGN(cur_win, 1 << tp->rx_opt.rcv_wscale);
30     }
31     /* 将当前的接收窗口设置为新的窗口大小。 */
32     tp->rcv_wnd = new_win;
33     tp->rcv_wup = tp->rcv_nxt;
34
35     /* 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 `RCV.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，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。

```

1  u32 __tcp_select_window(struct sock *sk)
2  {
3      struct inet_connection_sock *icsk = inet_csk(sk);
4      struct tcp_sock *tp = tcp_sk(sk);
5      int mss = icsk->icsk_ack.rcv_mss;
6      int free_space = tcp_space(sk);
7      int allowed_space = tcp_full_space(sk);
8      int full_space = min_t(int, tp->window_clamp, allowed_space);
9      int window;
10

```

```

11      /* 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。 */
12      if (mss > full_space)
13          mss = full_space;
14
15      if (free_space < (full_space >> 1)) {
16          /* 当空闲空间小于允许空间的一半时。 */
17          icsk->icsk_ack.quick = 0;
18
19          if (tcp_under_memory_pressure(sk))
20              tp->rcv_ssthresh = min(tp->rcv_ssthresh,
21                                     4U * tp->advms);
22
23          /* free_space 有可能成为新的窗口的大小, 因此, 需要考虑
24             * 窗口扩展的影响。
25             */
26          free_space = round_down(free_space, 1 << tp->rx_opt.rcv_wscale);
27
28          /* 如果空闲空间小于 mss 的大小, 或者低于最大允许空间的 1/16, 那么,
29             * 返回 0 窗口。否则, tcp_clamp_window() 会增长接收缓存到 tcp_rmem[2]。
30             * 新进入的数据会由于内酯限制而被丢弃。对于较大的窗口, 单纯地探测 mss 的
31             * 大小以宣告 0 窗口有些太晚了 (可能会超过限制)。
32             */
33          if (free_space < (allowed_space >> 4) || free_space < mss)
34              return 0;
35      }
36
37      if (free_space > tp->rcv_ssthresh)
38          free_space = tp->rcv_ssthresh;
39
40      /* 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。
41         * 所以就保持窗口是对齐 2 的幂的。
42         */
43      window = tp->rcv_wnd;
44      if (tp->rx_opt.rcv_wscale) {
45          window = free_space;
46
47          /* Advertise enough space so that it won't get scaled away.
48             * Import case: prevent zero window announcement if
49             * 1<rcv_wscale > mss.
50             */
51          if (((window >> tp->rx_opt.rcv_wscale) << tp->rx_opt.rcv_wscale) != window)
52              window = (((window >> tp->rx_opt.rcv_wscale) + 1)
53                         << tp->rx_opt.rcv_wscale);
54      } 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(sock 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

#### 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。该函数位于/net/ipv4/tcp\_ipv4.c 中。该函数会根据不同的 TCP 状态进行不同的处理,这里我们只是讨论服务器第一次握手的函数处理过程。

```
1  /* The socket must have it's spinlock held when we get
2   * here, unless it is a TCP_LISTEN socket.
3   *
4   * We have a potential double-lock case here, so even when
5   * doing backlog processing we use the BH locking scheme.
6   * This is because we cannot sleep with the original spinlock
7   * held.
```

## First Shake Hand of Server.png First Shake Hand of Server.bb



```

8  /*
9  int tcp_v4_do_rcv(struct sock *sk, struct sk_buff *skb)
10 {
11     struct sock *rsk;
12
13     /* 省略无关代码 */
14
15     if (tcp_checksum_complete(skb))
16         goto csum_err;
17
18     if (sk->sk_state == TCP_LISTEN) {
19         struct sock *nsk = tcp_v4_cookie_check(sk, skb);
20
21         if (!nsk)
22             goto discard;
23         if (nsk != sk) {
24             sock_rps_save_rxhash(nsk, skb);
25             sk_mark_napi_id(nsk, skb);
26             if (tcp_child_process(sk, nsk, skb)) {
27                 rsk = nsk;
28                 goto reset;
29             }
30             return 0;
31         }
32     } else
33         sock_rps_save_rxhash(sk, skb);
34
35     if (tcp_rcv_state_process(sk, skb)) {
36         rsk = sk;
37         goto reset;
38     }
39     return 0;
40
41 reset:
42     tcp_v4_send_reset(rsk, skb);
43 discard:
44     kfree_skb(skb);
45     /* Be careful here. If this function gets more complicated and
46      * gcc suffers from register pressure on the x86, sk (in %ebx)
47      * might be destroyed here. This current version compiles correctly,
48      * but you have been warned.
49     */

```

```

50     return 0;
51
52     csum_err:
53     TCP_INC_STATS_BH(sock_net(sk), TCP_MIB_CSUMERRORS);
54     TCP_INC_STATS_BH(sock_net(sk), TCP_MIB_INERRS);
55     goto discard;
56 }

```

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  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 }

```

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：

```

1  /*
2   * This function implements the receiving procedure of RFC 793 for
3   * all states except ESTABLISHED and TIME_WAIT.
4   * It's called from both tcp_v4_rcv and tcp_v6_rcv and should be
5   * address independent.
6   */
7
8  int tcp_rcv_state_process(struct sock *sk, struct sk_buff *skb)
9  {
10     struct tcp_sock *tp = tcp_sk(sk);
11     struct inet_connection_sock *icsk = inet_csk(sk);
12     const struct tcphdr *th = tcp_hdr(skb);
13     struct request_sock *req;
14     int queued = 0;
15     bool acceptable;
16
17     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 */
18
19     switch (sk->sk_state) {
20         /* 省略无关代码 */
21
22         case TCP_LISTEN:

```

```

23         if (th->ack)
24             return 1;
25
26         if (th->rst)
27             goto discard;
28
29         if (th->syn) {
30             if (th->fin)
31                 goto discard;
32             if (icsk->icsk_af_ops->conn_request(sk, skb) < 0)
33                 return 1;
34
35             /* Now we have several options: In theory there is
36              * nothing else in the frame. KA9Q has an option to
37              * send data with the syn, BSD accepts data with the
38              * syn up to the [to be] advertised window and
39              * Solaris 2.1 gives you a protocol error. For now
40              * we just ignore it, that fits the spec precisely
41              * and avoids incompatibilities. It would be nice in
42              * future to drop through and process the data.
43              *
44              * Now that TTCP is starting to be used we ought to
45              * queue this data.
46              * But, this leaves one open to an easy denial of
47              * service attack, and SYN cookies can't defend
48              * against this problem. So, we drop the data
49              * in the interest of security over speed unless
50              * it's still in use.
51              */
52             kfree_skb(skb);
53             return 0;
54         }
55         goto discard;
56
57         /* 省略无关代码 */
58     discard:
59         __kfree_skb(skb);
60     }
61     return 0;
62 }

```

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` 最终会调用 `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

该函数位于 `/net/ipv4/tcp_ipv4/tcp_ipv4.c` 中，函数如下：

```

1  int tcp_v4_conn_request(struct sock *sk, struct sk_buff *skb)
2  {
3      /* Never answer to SYNs send to broadcast or multicast */
4      if (skb_rtable(skb)->rt_flags & (RTCF_BROADCAST | RTCF_MULTICAST))
5          goto drop;
6
7      return tcp_conn_request(&tcp_request_sock_ops,
8                             &tcp_request_sock_ipv4_ops, sk, skb);
9
10 drop:
11     NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_LISTENDROPS);
12     return 0;
13 }

```

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。

`tcp_conn_request` 位于 `/include/net/tcp_input.c` 中。

```

1  int tcp_conn_request(struct request_sock_ops *rsk_ops,
2                      const struct tcp_request_sock_ops *af_ops,
3                      struct sock *sk, struct sk_buff *skb)
4  {
5      struct tcp_fastopen_cookie foc = { .len = -1 }; //初始化 len 字段
6      __u32 isn = TCP_SKB_CB(skb)->tcp_tw_isn; //tw: time wait   isn: initial sequence num
7      struct tcp_options_received tmp_opt;
8      struct tcp_sock *tp = tcp_sk(sk);
9      struct sock *fastopen_sk = NULL;
10     struct dst_entry *dst = NULL;
11     struct request_sock *req;
12     bool want_cookie = false; //标志是否开启了 SYN_COOKIE 选项
13     struct flowi fl; //路由查找相关的数据结构
14
15     /* Tw(time wait) buckets are converted to open requests without
16      * limitations, they conserve resources and peer is
17      * evidently real one.
18      */
19     if ((sysctl_tcp_syncookies == 2 ||
20         inet_csk_reqsk_queue_is_full(sk)) && !isn) {
21         want_cookie = tcp_syn_flood_action(sk, skb, rsk_ops->slab_name);
22         if (!want_cookie)
23             goto drop;
24     }

```

首先, 前面??? 如果 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`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_vlump`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_timestamp)
2     tcp_clear_options(&tmp_opt);

```

如果启动了 syncookies，并且 TCP 选型中没有存在时间戳，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。

```

1 tmp_opt.timestamp_ok = tmp_opt.saw_timestamp; //timestamp_ok 表示在收到的 SYN 包上看到的 TIMESTAMP
2 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。

```

1 /* Note: tcp_v6_init_req() might override ir_iif for link locals */
2 inet_rsk(req)->ir_iif = sk->sk_bound_dev_if; //bound device index if != 0 ?
3
4 af_ops->init_req(req, sk, skb);
5
6 if (security_inet_conn_request(sk, skb, req))
7     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。

```

1 if (!want_cookie && !isn) {
2     /* VJ's idea. We save last timestamp seen
3      * from the destination in peer table, when entering
4      * state TIME-WAIT, and check against it before
5      * accepting new connection request.
6      *
7      * If "isn" is not zero, this request hit alive
8      * timewait bucket, so that all the necessary checks
9      * are made in the function processing timewait state.
10    */

```



```

11     if (tcp_death_row.sysctl_tw_recycle) {
12         bool strict;
13
14         dst = af_ops->route_req(sk, &fl, req, &strict);
15
16         if (dst && strict &&
17             !tcp_peer_is_proven(req, dst, true,
18                 tmp_opt.saw_tstamp)) {
19             NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_PAWSPASSIVEREJECTED);
20             goto drop_and_release;
21         }
22     }
23     /* Kill the following clause, if you dislike this way. */
24     else if (!sysctl_tcp_syncookies &&
25             (sysctl_max_syn_backlog - inet_csk_reqsk_queue_len(sk) <
26              (sysctl_max_syn_backlog >> 2)) &&
27             !tcp_peer_is_proven(req, dst, false,
28                 tmp_opt.saw_tstamp)) {
29         /* Without syncookies last quarter of
30          * backlog is filled with destinations,
31          * proven to be alive.
32          * It means that we continue to communicate
33          * to destinations, already remembered
34          * to the moment of synflood.
35          */
36         pr_drop_req(req, ntohs(tcp_hdr(skb)->source),
37             rsk_ops->family);
38         goto drop_and_release;
39     }
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。显然, 对于目前的 linux 内核, 都不会在意这个。

```

1     if (!dst) {
2         dst = af_ops->route_req(sk, &fl, req, NULL);
3         if (!dst)
4             goto drop_and_free;
5     }
6
7     tcp_ecn_create_request(req, skb, sk, dst);
8
9     if (want_cookie) {
10         isn = cookie_init_sequence(af_ops, sk, skb, &req->mss);
11         req->cookie_ts = tmp_opt.tstamp_ok;
12         if (!tmp_opt.tstamp_ok)
13             inet_rsk(req)->ecn_ok = 0;
14     }
15
16     tcp_rsk(req)->snt_isn = isn;
17     tcp_rsk(req)->txhash = net_tx_rndhash();
18     tcp_openreq_init_rwin(req, sk, dst);
19     if (!want_cookie) {

```

```

20     tcp_reqsk_record_syn(sk, req, skb);
21     fastopen_sk = tcp_try_fastopen(sk, skb, req, &foc, dst);
22 }
23 if (fastopen_sk) {
24     af_ops->send_synack(fastopen_sk, dst, &fl, req,
25                         &foc, false);
26     /* Add the child socket directly into the accept queue */
27     inet_csk_reqsk_queue_add(sk, req, fastopen_sk);
28     sk->sk_data_ready(sk);
29     bh_unlock_sock(fastopen_sk);
30     sock_put(fastopen_sk);
31 } else {
32     tcp_rsk(req)->tfo_listener = false;
33     if (!want_cookie)
34         inet_csk_reqsk_queue_hash_add(sk, req, TCP_TIMEOUT_INIT);
35     af_ops->send_synack(sk, dst, &fl, req,
36                       &foc, !want_cookie);
37     if (want_cookie)
38         goto drop_and_free;
39 }
40 reqsk_put(req);
41 return 0;
42
43 drop_and_release:
44     dst_release(dst);
45 drop_and_free:
46     reqsk_free(req);
47 drop:
48     NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_LISTENDROPS);
49     return 0;

```

暂时不懂,,, 等等在分析。。。。。。

### 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。

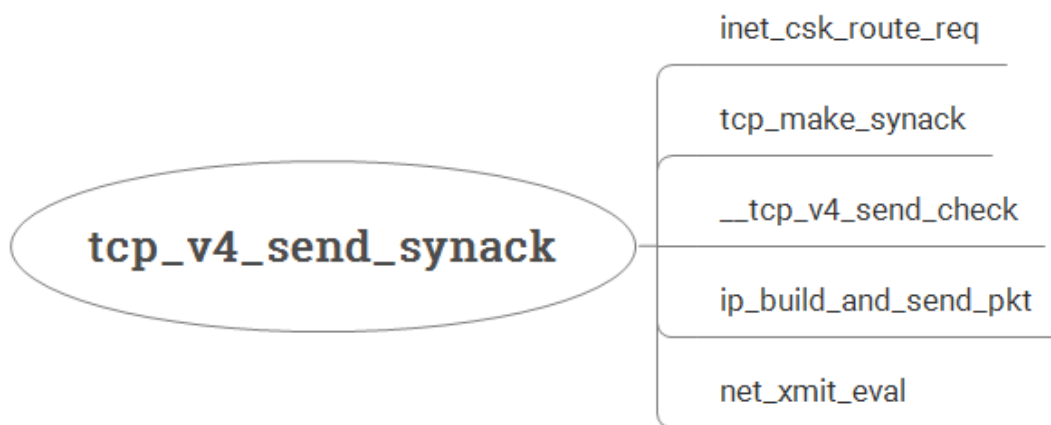
## 3.2.3 第二次握手：发送 SYN+ACK 段

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：

Second Shake Hand of Server.png Second Shake Hand of Server.bb



### 3.2.3.2 tcp\_v4\_send\_synack

```

1  /*
2  Function:
3      Send a SYN-ACK after having received a SYN.
4      This still operates on a request_sock only, not on a big
5      socket.
6  Paramaters:
7      struct sock *sk           :
8      struct dst_entry *dst     :
9      struct flowi *fl          :
10     struct request_sock *req   :
11     struct tcp_fastopen_cookie *foc :

```

```

12     bool attach_req          :
13     */
14     static int tcp_v4_send_synack(const struct sock *sk, struct dst_entry *dst,
15                                   struct flowi *fl,
16                                   struct request_sock *req,
17                                   struct tcp_fastopen_cookie *foc,
18                                   bool attach_req)
19     {
20         const struct inet_request_sock *ireq = inet_rsk(req);
21         struct flowi4 fl4;
22         int err = -1;
23         struct sk_buff *skb;
24
25         /* First, grab a route. */
26         if (!dst && (dst = inet_csk_route_req(sk, &fl4, req)) == NULL)
27             return -1;
28
29         skb = tcp_make_synack(sk, dst, req, foc, attach_req);
30
31         if (skb) {
32             __tcp_v4_send_check(skb, ireq->ir_loc_addr, ireq->ir_rmt_addr);
33
34             err = ip_build_and_send_pkt(skb, sk, ireq->ir_loc_addr,
35                                         ireq->ir_rmt_addr,
36                                         ireq->opt);
37             err = net_xmit_eval(err);
38         }
39
40         return err;
41     }

```

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

该函数用来构造一个 `SYN+ACK` 段, 并初始化 TCP 首部及 SKB 中的各字段项, 填入相应的选项, 如 `MSS`, `SACK`, 窗口扩大因子, 时间戳等。函数如下:

```

1  /*
2  Function:
3      tcp_make_synack - Prepare a SYN-ACK.
4      Allocate one skb and build a SYNACK packet.
5      @dst is consumed : Caller should not use it again.
6  Parameter:
7      sk          : listener socket
8      dst         : dst entry attached to the SYNACK
9      req         : request_sock pointer
10     foc         :
11     attach_req  :
12  */

```

```

13 struct sk_buff *tcp_make_synack(const struct sock *sk, struct dst_entry *dst,
14                                struct request_sock *req,
15                                struct tcp_fastopen_cookie *foc,
16                                bool attach_req)
17 {
18     struct inet_request_sock *ireq = inet_rsk(req);
19     const struct tcp_sock *tp = tcp_sk(sk);
20     struct tcp_md5sig_key *md5 = NULL;
21     struct tcp_out_options opts;
22     struct sk_buff *skb;
23     int tcp_header_size;
24     struct tcphdr *th;
25     u16 user_mss;
26     int mss;
27
28     skb = alloc_skb(MAX_TCP_HEADER, GFP_ATOMIC);
29     if (unlikely(!skb)) {
30         dst_release(dst);
31         return NULL;
32     }

```

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，参见后面。todo.

```

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。

```
1         skb_set_hash(skb, tcp_rsk(req)->txhash, PKT_HASH_TYPE_L4);
2         tcp_header_size = tcp_synack_options(req, mss, skb, &opts, md5, foc) +
3             sizeof(*th);
4
5         skb_push(skb, tcp_header_size);
6         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。

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

清空 tcp 头部，并设置 tcp 头部的各个字段。

```
1         /* Setting of flags are superfluous here for callers (and ECE is
2          * not even correctly set)
3          */
4         tcp_init_nondata_skb(skb, tcp_rsk(req)->snt_isn,
5             TCPHDR_SYN | TCPHDR_ACK);
6
7         th->seq = htonl(TCP_SKB_CB(skb)->seq);
8         /* XXX data is queued and acked as is. No buffer/window check */
9         th->ack_seq = htonl(tcp_rsk(req)->rcv_nxt);
10
11         /* RFC1323: The window in SYN & SYN/ACK segments is never scaled. */
12         th->window = htons(min(req->rsk_rcv_wnd, 65535U));
13         tcp_options_write((__be32 *) (th + 1), NULL, &opts);
14         th->doff = (tcp_header_size >> 2);
15         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。

```
1     #ifdef CONFIG_TCP_MD5SIG
2         /* Okay, we have all we need - do the md5 hash if needed */
3         if (md5)
4             tcp_rsk(req)->af_specific->calc_md5_hash(opts.hash_location,
5                 md5, req_to_sk(req), skb);
6         rcu_read_unlock();
7     #endif
8
9     /* Do not fool tcpdump (if any), clean our debris */
```

```

10     skb->tstamp.tv64 = 0;
11     return skb;
12 }

```

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

#### 3.2.4.2 tcp\_v4\_do\_rcv

```

1  /* The socket must have it's spinlock held when we get
2   * here, unless it is a TCP_LISTEN socket.
3   *
4   * We have a potential double-lock case here, so even when
5   * doing backlog processing we use the BH locking scheme.
6   * This is because we cannot sleep with the original spinlock
7   * held.
8   */
9  int tcp_v4_do_rcv(struct sock *sk, struct sk_buff *skb)
10 {
11     struct sock *rsk;
12
13     /** 省略无关代码 **/
14
15     if (tcp_checksum_complete(skb))
16         goto csum_err;
17
18     if (sk->sk_state == TCP_LISTEN) {
19         struct sock *nsk = tcp_v4_cookie_check(sk, skb);
20
21         if (!nsk)
22             goto discard;
23         if (nsk != sk) {
24             sock_rps_save_rxhash(nsk, skb);
25             sk_mark_napi_id(nsk, skb);
26             if (tcp_child_process(sk, nsk, skb)) {
27                 rsk = nsk;
28                 goto reset;
29             }
30             return 0;
31         }
32     } else
33         sock_rps_save_rxhash(sk, skb);
34     /** 省略无关代码 **/
35 reset:
36     tcp_v4_send_reset(rsk, skb);
37 discard:
38     kfree_skb(skb);
39     /* Be careful here. If this function gets more complicated and
40      * gcc suffers from register pressure on the x86, sk (in %ebx)

```

```

41     * might be destroyed here. This current version compiles correctly,
42     * but you have been warned.
43     */
44     return 0;
45
46 csum_err:
47     TCP_INC_STATS_BH(sock_net(sk), TCP_MIB_CSUMERRORS);
48     TCP_INC_STATS_BH(sock_net(sk), TCP_MIB_INERRS);
49     goto discard;
50 }

```

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 序列号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。

```

1  /*
2  Functions:
3      Queue segment on the new socket if the new socket is active,
4      otherwise we just shortcircuit this and continue with
5      the new socket.
6
7      For the vast majority of cases child->sk_state will be TCP_SYN_RECV
8      when entering. But other states are possible due to a race condition
9      where after __inet_lookup_established() fails but before the listener
10     locked is obtained, other packets cause the same connection to
11     be created.
12 Parameters:
13

```



```

14  */
15
16  int tcp_child_process(struct sock *parent, struct sock *child,
17                      struct sk_buff *skb)
18  {
19      int ret = 0;
20      int state = child->sk_state;
21
22      tcp_sk(child)->segs_in += max_t(u16, 1, skb_shinfo(skb)->gso_segs);
23      if (!sock_owned_by_user(child)) {
24          ret = tcp_rcv_state_process(child, skb);
25          /* Wakeup parent, send SIGIO */
26          if (state == TCP_SYN_RECV && child->sk_state != state)
27              parent->sk_data_ready(parent);
28      } else {
29          /* Alas, it is possible again, because we do lookup
30           * in main socket hash table and lock on listening
31           * socket does not protect us more.
32           */
33          __sk_add_backlog(child, skb);
34      }
35
36      bh_unlock_sock(child);
37      sock_put(child);
38      return ret;
39  }

```

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。

```

1      acceptable = tcp_ack(sk, skb, FLAG_SLOWPATH |
2                          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。

```

1      switch (sk->sk_state) {
2          case TCP_SYN_RECV:
3              if (!acceptable)
4                  return 1;
5
6              if (!tp->srtt_us)
7                  tcp_synack_rtt_meas(sk, req);
8
9              /* Once we leave TCP_SYN_RECV, we no longer need req
10               * so release it.
11               */

```

```

12         if (req) {
13             tp->total_retrans = req->num_retrans;
14             reqsk_fastopen_remove(sk, req, false);
15         } else {
16             /* Make sure socket is routed, for correct metrics. */
17             icsk->icsk_af_ops->rebuild_header(sk);
18             tcp_init_congestion_control(sk);
19
20             tcp_mtup_init(sk);
21             tp->copied_seq = tp->rcv_nxt;
22             tcp_init_buffer_space(sk);
23         }
24         smp_mb();
25         tcp_set_state(sk, TCP_ESTABLISHED);
26         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。

```

1         /* Note, that this wakeup is only for marginal crossed SYN case.
2          * Passively open sockets are not waked up, because
3          * sk->sk_sleep == NULL and sk->sk_socket == NULL.
4          */
5         if (sk->sk_socket)
6             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。

```

1         tp->snd_una = TCP_SKB_CB(skb)->ack_seq;
2         tp->snd_wnd = ntohs(th->window) << tp->rx_opt.snd_wscale;
3         tcp_init_wl(tp, TCP_SKB_CB(skb)->seq);
4
5         if (tp->rx_opt.timestamp_ok)
6             tp->advms = TCPOLEN_TSTAMP_ALIGNED;

```

初始化传输控制块的各个字段，对时间戳进行处理。

```

1         if (req) {
2             /* Re-arm the timer because data may have been sent out.
3              * This is similar to the regular data transmission case
4              * when new data has just been ack'ed.
5              *
6              * (TFO) - we could try to be more aggressive and
7              * retransmitting any data sooner based on when they
8              * are sent out.
9              */
10            tcp_rearm_rto(sk);
11        } else
12            tcp_init_metrics(sk);

```

为该套接口初始化路由。

```

1         tcp_update_pacing_rate(sk);
2
3         /* Prevent spurious tcp_cwnd_restart() on first data packet */
4         tp->lsndtime = tcp_time_stamp;
5
6         tcp_initialize_rcv_mss(sk);

```

```

7         tcp_fast_path_on(tp);
8         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。

```

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

```

检测带外数据标志位。

```

1     /* step 7: process the segment text */
2     switch (sk->sk_state) {
3         /** 省略无关代码 **/
4         case TCP_ESTABLISHED:
5             tcp_data_queue(sk, skb);
6             queued = 1;
7             break;
8     }

```

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。

```

1     /* tcp_data could move socket to TIME-WAIT */
2     if (sk->sk_state != TCP_CLOSE) {
3         tcp_data_snd_check(sk);
4         tcp_ack_snd_check(sk);
5     }
6
7     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。

Contents	
4.1 主动关闭 . . . . .	69
4.1.1 第一次握手——发送 FIN . . . . .	69
4.1.2 第二次握手——接受 ACK . . . . .	71
4.1.3 第三次握手——接受 FIN . . . . .	74
4.1.4 第四次握手——发送 ACK . . . . .	76
4.1.5 同时关闭 . . . . .	78
4.1.6 TIME_WAIT . . . . .	79
4.2 被动关闭 . . . . .	81
4.2.1 基本流程 . . . . .	81
4.2.2 第一次握手：接收FIN . . . . .	81
4.2.2.1 函数调用关系 . . . . .	81
4.2.2.2 tcp_fin . . . . .	81

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:

```
1 void tcp_shutdown(struct sock *sk, int how)
2 {
3     /*      We need to grab some memory, and put together a FIN,
4      *      and then put it into the queue to be sent.
5      *      Tim MacKenzie(tym@dibbler.cs.monash.edu.au) 4 Dec '92.
6      */
7     if (!(how & SEND_SHUTDOWN))
8         return;
```

```

9
10     /* 如果此时已经发送一个 FIN 了, 就跳过。 */
11     if ((1 << sk->sk_state) &
12         (TCPF_ESTABLISHED | TCPF_SYN_SENT |
13          TCPF_SYN_RECV | TCPF_CLOSE_WAIT)) {
14         /* Clear out any half completed packets. FIN if needed. */
15         if (tcp_close_state(sk))
16             tcp_send_fin(sk);
17     }
18 }

```

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。

```

1  int tcp_rcv_state_process(struct sock *sk, struct sk_buff *skb)
2  {
3      struct tcp_sock *tp = tcp_sk(sk);
4      struct inet_connection_sock *icsk = inet_csk(sk);
5      const struct tcphdr *th = tcp_hdr(skb);
6      struct request_sock *req;
7      int queued = 0;
8      bool acceptable;
9
10     tp->rx_opt.saw_tstamp = 0;
11     switch (sk->sk_state) {
12     case TCP_CLOSE:
13         goto discard;
14
15     case TCP_LISTEN:

```

```

16      /* LISTEN 状态处理代码, 略去 */
17
18      case TCP_SYN_SENT:
19          /* SYN-SENT 状态处理代码, 略去 */
20      }
21
22      /* fastopen 相关代码及各类合法性判断, 略去 */
23
24      switch (sk->sk_state) {
25      case TCP_SYN_RECV:
26          /* SYN-RECV 状态处理代码, 略去 */
27
28      case TCP_FIN_WAIT1: {
29          struct dst_entry *dst;
30          int tmo;
31
32          /* 如果当前的套接字为开启了 Fast Open 的套接字, 且该 ACK 为
33           * 接收到的第一个 ACK, 那么这个 ACK 应该是在确认 SYNACK 包,
34           * 因此, 停止 SYNACK 计时器。
35           */
36          if (req) {
37              /* Return RST if ack_seq is invalid.
38               * Note that RFC793 only says to generate a
39               * DUPACK for it but for TCP Fast Open it seems
40               * better to treat this case like TCP_SYN_RECV
41               * above.
42               */
43              if (!acceptable)
44                  return 1;
45              /* 移除 fastopen 请求 */
46              reqsk_fastopen_remove(sk, req, false);
47              tcp_rearm_rto(sk);
48          }
49          if (tp->snd_una != tp->write_seq)
50              break;
51
52          /* 收到 ACK 后, 转移到 TCP_FIN_WAIT2 状态, 将发送端关闭。 */
53          tcp_set_state(sk, TCP_FIN_WAIT2);
54          sk->sk_shutdown |= SEND_SHUTDOWN;
55
56          /* 确认路由缓存有效 */
57          dst = __sk_dst_get(sk);
58          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。

```

1  if (tw->tw_substate == TCP_FIN_WAIT2) {
2      /* 重复 tcp_rcv_state_process() 所进行的所有检测 */
3
4      /* 序号不在窗口内, 发送 ACK */
5      if (paws_reject ||
6          !tcp_in_window(TCP_SKB_CB(skb)->seq, TCP_SKB_CB(skb)->end_seq,
7                        tcptw->tw_rcv_nxt,
8                        tcptw->tw_rcv_nxt + tcptw->tw_rcv_wnd))
9          return tcp_timewait_check_oow_rate_limit(
10             tw, skb, LINUX_MIB_TCPACKSKIPPEDFINWAIT2);
11
12     /* 如果收到 RST 包, 则销毁 timewait 控制块并返回 TCP_TW_SUCCESS */
13     if (th->rst)

```

```

14         goto kill;
15
16         /* 如果收到 SYN 包, 则销毁并发送 RST */
17         if (th->syn && !before(TCP_SKB_CB(skb)->seq, tcptw->tw_rcv_nxt))
18             goto kill_with_rst;
19
20         /* 如果收到 DACK, 则释放该控制块 */
21         if (!th->ack ||
22             !after(TCP_SKB_CB(skb)->end_seq, tcptw->tw_rcv_nxt) ||
23             TCP_SKB_CB(skb)->end_seq == TCP_SKB_CB(skb)->seq) {
24             inet_twsk_put(tw);
25             return TCP_TW_SUCCESS;
26         }
27
28         /* 之后只有两种情况, 有新数据或收到 FIN 包 */
29         if (!th->fin ||
30             TCP_SKB_CB(skb)->end_seq != tcptw->tw_rcv_nxt + 1) {
31             /* 如果收到了新的数据或者序号有问题,
32              * 则销毁控制块并发送 RST。
33              */
34             kill_with_rst:
35                 inet_twsk_deschedule_put(tw);
36                 return TCP_TW_RST;
37         }
38
39         /* 收到了 FIN 包, 进入 TIME_WAIT 状态 */
40         tw->tw_substate = TCP_TIME_WAIT;
41         tcptw->tw_rcv_nxt = TCP_SKB_CB(skb)->end_seq;
42         /* 如果启用了时间戳选项, 则设置相关属性 */
43         if (tmp_opt.saw_tstamp) {
44             tcptw->tw_ts_recent_stamp = get_seconds();
45             tcptw->tw_ts_recent = tmp_opt.rcv_tsval;
46         }
47
48         /* 启动 TIME_WAIT 定时器 */
49         if (tcp_death_row.sysctl_tw_recycle &&
50             tcptw->tw_ts_recent_stamp &&
51             tcp_tw_remember_stamp(tw))
52             inet_twsk_reschedule(tw, tw->tw_timeout);
53         else
54             inet_twsk_reschedule(tw, TCP_TIMEWAIT_LEN);
55         return TCP_TW_ACK;
56     }
57
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_next, tcptw->tw_rcv_next,
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。

```

1 /* 下面的代码负责在 SYN_RECV 和 TIME_WAIT 状态下发送 ACK 包。
2  *
3  * The code following below sending ACKs in SYN-RECV and TIME-WAIT states
4  * outside socket context is ugly, certainly. What can I do?
5  */
6 static void tcp_v4_send_ack(struct net *net,
7                             struct sk_buff *skb, u32 seq, u32 ack,
8                             u32 win, u32 tsval, u32 tsecr, int oif,
9                             struct tcp_md5sig_key *key,
10                             int reply_flags, u8 tos)
11 {
12     const struct tcphdr *th = tcp_hdr(skb);
13     struct {

```

```

14         struct tcphdr th;
15         __be32 opt[(TCPOLEN_TSTAMP_ALIGNED >> 2)
16 #ifdef CONFIG_TCP_MD5SIG
17             + (TCPOLEN_MD5SIG_ALIGNED >> 2)
18 #endif
19             ];
20     } rep;
21     struct ip_reply_arg arg;
22
23     memset(&rep.th, 0, sizeof(struct tcphdr));
24     memset(&arg, 0, sizeof(arg));
25
26     /* 构造参数和 TCP 头部 */
27     arg.iov[0].iov_base = (unsigned char *)&rep;
28     arg.iov[0].iov_len = sizeof(rep.th);
29     if (tsecr) {
30         rep.opt[0] = htonl((TCPOPT_NOP << 24) | (TCPOPT_NOP << 16) |
31             (TCPOPT_TIMESTAMP << 8) |
32             TCPOLEN_TIMESTAMP);
33         rep.opt[1] = htonl(tsva);
34         rep.opt[2] = htonl(tsecr);
35         arg.iov[0].iov_len += TCPOLEN_TSTAMP_ALIGNED;
36     }
37
38     /* 交换发送端和接收端 */
39     rep.th.dest = th->source;
40     rep.th.source = th->dest;
41     rep.th.doff = arg.iov[0].iov_len / 4;
42     rep.th.seq = htonl(seq);
43     rep.th.ack_seq = htonl(ack);
44     rep.th.ack = 1;
45     rep.th.window = htons(win);
46
47     /* 略去和 MD5 相关的部分代码 */
48
49     /* 设定标志位和校验码 */
50     arg.flags = reply_flags;
51     arg.csum = csum_tcpudp_nofold(ip_hdr(skb)->daddr,
52         ip_hdr(skb)->saddr, /* XXX */
53         arg.iov[0].iov_len, IPPROTO_TCP, 0);
54     arg.csumoffset = offsetof(struct tcphdr, check) / 2;
55     if (oif)
56         arg.bound_dev_if = oif;
57     arg.tos = tos;
58     /* 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。

```

1      /*
2      *      Now real TIME-WAIT state.
3      *
4      *      RFC 1122:
5      *      "When a connection is [...] on TIME-WAIT state [...]
6      *      [a TCP] MAY accept a new SYN from the remote TCP to
7      *      reopen the connection directly, if it:
8      *
9      *      (1) assigns its initial sequence number for the new
10     *      connection to be larger than the largest sequence
11     *      number it used on the previous connection incarnation,
12     *      and
13     *
14     *      (2) returns to TIME-WAIT state if the SYN turns out
15     *      to be an old duplicate".
16     */
17
18     if (!paws_reject &&
19         (TCP_SKB_CB(skb)->seq == tcptw->tw_rcv_nxt &&
20          (TCP_SKB_CB(skb)->seq == TCP_SKB_CB(skb)->end_seq || th->rst))) {
21         /* 序号没有回卷, 仍在窗口中。 */
22
23         if (th->rst) {
24             /* This is TIME_WAIT assassination, in two flavors.
25              * Oh well... nobody has a sufficient solution to this
26              * protocol bug yet.
27              */
28             if (sysctl_tcp_rfc1337 == 0) {
29 kill:
30                 inet_twsk_deschedule_put(tw);
31                 return TCP_TW_SUCCESS;
32             }
33         }
34         /* 重新激活定时器 */
35         inet_twsk_reschedule(tw, TCP_TIMEWAIT_LEN);
36
37         if (tmp_opt.saw_tstamp) {
38             tcptw->tw_ts_recent = tmp_opt.rcv_tsval;
39             tcptw->tw_ts_recent_stamp = get_seconds();
40         }
41
42         inet_twsk_put(tw);
43         return TCP_TW_SUCCESS;
44     }

```

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。

```

1      /* 之后是超出窗口范围的情况。
2
3      All the segments are ACKed immediately.
4

```

```

5      The only exception is new SYN. We accept it, if it is
6      not old duplicate and we are not in danger to be killed
7      by delayed old duplicates. RFC check is that it has
8      newer sequence number works at rates <40Mbit/sec.
9      However, if paws works, it is reliable AND even more,
10     newer sequence number works at rates <40Mbit/sec.
11     However, if paws works, it is reliable AND even more,
12     we even may relax silly seq space cutoff.
13
14     RED-PEN: we violate main RFC requirement, if this SYN will appear
15     old duplicate (i.e. we receive RST in reply to SYN-ACK),
16     we must return socket to time-wait state. It is not good,
17     but not fatal yet.
18     */
19
20     if (th->syn && !th->rst && !th->ack && !paws_reject &&
21         (after(TCP_SKB_CB(skb)->seq, tcptw->tw_rcv_nxt) ||
22          (tmp_opt.saw_tstamp &&
23           (s32)(tcptw->tw_ts_recent - tmp_opt.rcv_tsval) < 0))) {
24         /* 如果可以接受该 SYN 请求, 那么重新计算 isn 号, 并发出 syn. */
25         u32 isn = tcptw->tw_snd_nxt + 65535 + 2;
26         if (isn == 0)
27             isn++;
28         TCP_SKB_CB(skb)->tcp_tw_isn = isn;
29         return TCP_TW_SYN;
30     }
31
32     if (paws_reject)
33         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。

```

1      if (!th->rst) {
2          /* In this case we must reset the TIMEWAIT timer.
3          */
4          * If it is ACKless SYN it may be both old duplicate
5          * and new good SYN with random sequence number <rcv_nxt.
6          * Do not reschedule in the last case.
7          */
8          if (paws_reject || th->ack)
9              inet_twsk_reschedule(tw, TCP_TIMEWAIT_LEN);
10
11          return tcp_timewait_check_oow_rate_limit(
12              tw, skb, LINUX_MIB_TCPACKSKIPPEDTIMEWAIT);
13      }
14      inet_twsk_put(tw);
15      return TCP_TW_SUCCESS;
16  }

```

## 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 段到齐之后再作处理，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。

```

1  /*
2  Location:
3
4      net/ipv4/tcp_input.c
5
6  Function:
7
8      Process the FIN bit. This now behaves as it is supposed to work
9      and the FIN takes effect when it is validly part of sequence
10     space. Not before when we get holes.
11
12     If we are ESTABLISHED, a received fin moves us to CLOSE-WAIT
13     (and thence onto LAST-ACK and finally, CLOSE, we never enter
14     TIME-WAIT)
15
16     If we are in FINWAIT-1, a received FIN indicates simultaneous
17     close and we go into CLOSING (and later onto TIME-WAIT)
18
19     If we are in FINWAIT-2, a received FIN moves us to TIME-WAIT.
20
21 Parameters:
22
23     sk: 传输控制块
24 */
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。

```

1      case TCP_FIN_WAIT2:
2          /* Received a FIN -- send ACK and enter TIME_WAIT. */
3          tcp_send_ack(sk);
4          tcp_time_wait(sk, TCP_TIME_WAIT, 0);
5          break;
6      default:
7          /* Only TCP_LISTEN and TCP_CLOSE are left, in these
8             * cases we should never reach this piece of code.
9             */
10         pr_err("%s: Impossible, sk->sk_state=%d\n",

```

```

11         __func__, sk->sk_state);
12         break;
13     }

```

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。

```

1     /* It is possible, that we have something out-of-order after FIN.
2      * Probably, we should reset in this case. For now drop them.
3      */
4     __skb_queue_purge(&tp->out_of_order_queue);
5     if (tcp_is_sack(tp))
6         tcp_sack_reset(&tp->rx_opt);
7     sk_mem_reclaim(sk);
8
9     if (!sock_flag(sk, SOCK_DEAD)) {
10         sk->sk_state_change(sk);
11
12         /* Do not send POLL_HUP for half duplex close. */
13         if (sk->sk_shutdown == SHUTDOWN_MASK ||
14             sk->sk_state == TCP_CLOSE)
15             sk_wake_async(sk, SOCK_WAKE_WAITD, POLL_HUP);
16         else
17             sk_wake_async(sk, SOCK_WAKE_WAITD, POLL_IN);
18     }
19 }

```

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

非核心函数分析

Contents	
5.1	SKB . . . . . 85
5.2	Inet . . . . . 85
5.2.1	inet_hash_connect && __inet_hash_connect . . . . . 85
5.2.1.1	inet_hash_connect . . . . . 85
5.2.1.2	__inet_hash_connect . . . . . 85
5.2.2	inet_twsk_put . . . . . 87
5.3	TCP 层 . . . . . 87
5.3.1	__tcp_push_pending_frames . . . . . 87
5.3.2	tcp_fin_time . . . . . 88
5.3.3	tcp_done . . . . . 88
5.3.4	tcp_init_nondata_skb . . . . . 89
5.3.5	before() 和 after() . . . . . 89
5.3.6	tcp_shutdown . . . . . 89
5.3.7	tcp_close . . . . . 90

5.1 SKB

5.2 Inet

5.2.1 inet\_hash\_connect && \_\_inet\_hash\_connect

5.2.1.1 inet\_hash\_connect

```
1  /*
2  * Bind a port for a connect operation and hash it.
3  */
```

```

4  int inet_hash_connect(struct inet_timewait_death_row *death_row,
5                        struct sock *sk)
6  {
7      u32 port_offset = 0;
8
9      if (!inet_sk(sk)->inet_num)
10         port_offset = inet_sk_port_offset(sk);
11     return __inet_hash_connect(death_row, sk, port_offset,
12                               __inet_check_established);
13 }

```

### 5.2.1.2 \_\_inet\_hash\_connect

```

1  int __inet_hash_connect(struct inet_timewait_death_row *death_row,
2                        struct sock *sk, u32 port_offset,
3                        int (*check_established)(struct inet_timewait_death_row *,
4                                                struct sock *, __u16, struct inet_timewait_sock **))
5  {
6      struct inet_hashinfo *hinfo = death_row->hashinfo;
7      const unsigned short snum = inet_sk(sk)->inet_num;
8      struct inet_bind_hashbucket *head;
9      struct inet_bind_bucket *tb;
10     int ret;
11     struct net *net = sock_net(sk);
12
13     if (!snum) {
14         int i, remaining, low, high, port;
15         static u32 hint;
16         u32 offset = hint + port_offset;
17         struct inet_timewait_sock *tw = NULL;
18
19         inet_get_local_port_range(net, &low, &high);
20         remaining = (high - low) + 1;
21
22         /* By starting with offset being an even number,
23         * we tend to leave about 50% of ports for other uses,
24         * like bind(0).
25         */
26         offset &= ~1;
27
28         local_bh_disable();
29         for (i = 0; i < remaining; i++) {
30             port = low + (i + offset) % remaining;
31             if (inet_is_local_reserved_port(net, port))
32                 continue;
33             head = &hinfo->bhash[inet_bhashfn(net, port,
34                                                hinfo->bhash_size)];
35             spin_lock(&head->lock);
36
37             /* Does not bother with rcv_saddr checks,
38             * because the established check is already
39             * unique enough.
40             */
41             inet_bind_bucket_for_each(tb, &head->chain) {
42                 if (net_eq(ib_net(tb), net) &&
43                     tb->port == port) {
44                     if (tb->fastreuse >= 0 ||
45                         tb->fastreuseport >= 0)

```

```

46         goto next_port;
47         WARN_ON(hlist_empty(&tb->owners));
48         if (!check_established(death_row, sk,
49                               port, &tw))
50             goto ok;
51         goto next_port;
52     }
53 }
54
55     tb = inet_bind_bucket_create(hinfo->bind_bucket_cachep,
56                                net, head, port);
57     if (!tb) {
58         spin_unlock(&head->lock);
59         break;
60     }
61     tb->fastreuse = -1;
62     tb->fastreuseport = -1;
63     goto ok;
64
65 next_port:
66     spin_unlock(&head->lock);
67 }
68 local_bh_enable();
69
70 return -EADDRNOTAVAIL;
71
72 ok:
73     hint += (i + 2) & ~1;
74
75     /* Head lock still held and bh's disabled */
76     inet_bind_hash(sk, tb, port);
77     if (sk_unhashed(sk)) {
78         inet_sk(sk)->inet_sport = htons(port);
79         inet_ehash_nolisten(sk, (struct sock *)tw);
80     }
81     if (tw)
82         inet_twsk_bind_unhash(tw, hinfo);
83     spin_unlock(&head->lock);
84
85     if (tw)
86         inet_twsk_deschedule_put(tw);
87
88     ret = 0;
89     goto out;
90 }
91
92 head = &hinfo->bhash[inet_bhashfn(net, snum, hinfo->bhash_size)];
93 tb = inet_csk(sk)->icsk_bind_hash;
94 spin_lock_bh(&head->lock);
95 if (sk_head(&tb->owners) == sk && !sk->sk_bind_node.next) {
96     inet_ehash_nolisten(sk, NULL);
97     spin_unlock_bh(&head->lock);
98     return 0;
99 } 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 }

```

### 5.2.2 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 }

```

## 5.3 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??

### 5.3.1 \_\_tcp\_push\_pending\_frames

```

1  /* 将等待在队列中的包全部发出。 */
2  void __tcp_push_pending_frames(struct sock *sk, unsigned int cur_mss,
3                               int nonagle)
4  {
5      /* 如果此时连接已经关闭了, 那么直接返回。 */
6      if (unlikely(sk->sk_state == TCP_CLOSE))
7          return;
8
9      /* 关闭 nagle 算法, 将剩余的部分发送出去。 */
10     if (tcp_write_xmit(sk, cur_mss, nonagle, 0,
11                       sk_gfp_atomic(sk, GFP_ATOMIC)))
12         tcp_check_probe_timer(sk);
13 }

```

### 5.3.2 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。

```

1  static inline int tcp_fin_time(const struct sock *sk)
2  {
3      int fin_timeout = tcp_sk(sk)->linger2 ? : sysctl_tcp_fin_timeout;
4      const int rto = inet_csk(sk)->icsk_rto;
5
6      if (fin_timeout < (rto << 2) - (rto >> 1))
7          fin_timeout = (rto << 2) - (rto >> 1);
8
9      return fin_timeout;
10 }

```

## 5.3.3 tcp\_done

```

1  /* 该函数用于完成关闭 TCP 连接，回收并清理相关资源。 */
2  void tcp_done(struct sock *sk)
3  {
4      struct request_sock *req = tcp_sk(sk)->fastopen_rsk;
5
6      /* 当套接字状态为 SYN_SENT 或 SYN_RECV 时，更新统计数据。 */
7      if (sk->sk_state == TCP_SYN_SENT || sk->sk_state == TCP_SYN_RECV)
8          TCP_INC_STATS_BH(sock_net(sk), TCP_MIB_ATTEMPTFAILS);
9
10     /* 将连接状态设置为关闭，并清除定时器。 */
11     tcp_set_state(sk, TCP_CLOSE);
12     tcp_clear_xmit_timers(sk);
13     /* 当启用了 Fast Open 时，移除 fastopen 请求 */
14     if (req)
15         reqsk_fastopen_remove(sk, req, false);
16
17     sk->sk_shutdown = SHUTDOWN_MASK;
18
19     /* 如果状态不为 SOCK_DEAD，则唤醒等待着的进程。 */
20     if (!sock_flag(sk, SOCK_DEAD))
21         sk->sk_state_change(sk);
22     else
23         inet_csk_destroy_sock(sk);
24 }

```

## 5.3.4 tcp\_init\_nondata\_skb

该函数提供了初始化不含数据的 skb 的功能。函数原型如下：

```

1  tcp_init_nondata_skb(struct sk_buff *skb, u32 seq, u8 flags);

```

**skb** 待初始化的 sk\_buff。

**seq** 序号

**flags** 标志位

```

1  static void tcp_init_nondata_skb(struct sk_buff *skb, u32 seq, u8 flags)
2  {
3      /* 设置校验码 */
4      skb->ip_summed = CHECKSUM_PARTIAL;
5      skb->csum = 0;
6
7      /* 设置标志位 */
8      TCP_SKB_CB(skb)->tcp_flags = flags;
9      TCP_SKB_CB(skb)->sacked = 0;
10
11     tcp_skb_pcount_set(skb, 1);
12
13     /* 设置起始序号 */
14     TCP_SKB_CB(skb)->seq = seq;
15     if (flags & (TCPHDR_SYN | TCPHDR_FIN))

```



```

16         seq++;
17         TCP_SKB_CB(skb)->end_seq = seq;
18     }

```

### 5.3.5 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

```

1  /* include/net/tcp.h
2  * 比较两个无符号 32 位整数
3  */
4  static inline bool before(__u32 seq1, __u32 seq2)
5  {
6      return (__s32)(seq1-seq2) < 0;
7  }
8  #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。

### 5.3.6 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。

```

1  /*
2  Location:
3
4      net/ipv4/tcp.c
5
6  Function:
7
8      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      /* If we've already sent a FIN, or it's a closed state, skip this. */
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 }

```

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` 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。

### 5.3.7 tcp\_close

```

1  /*
2  Location:
3
4      net/ipv4/tcp.c
5
6  Function:
7
8
9
10 Parameter:
11
12     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.
7      */
8      if (unlikely(tcp_sk(sk)->repair)) {
9          sk->sk_prot->disconnect(sk, 0);
10     } else if (data_was_unread) {
11         /* Unread data was tossed, zap the connection. */
12         NET_INC_STATS_USER(sock_net(sk), LINUX_MIB_TCPABORTONCLOSE);
13         tcp_set_state(sk, TCP_CLOSE);
14         tcp_send_active_reset(sk, sk->sk_allocation);
15     } else if (sock_flag(sk, SOCK_LINGER) && !sk->sk_lingertime) {
16         /* Check zero linger_after_ checking for unread data. */
17         sk->sk_prot->disconnect(sk, 0);
18         NET_INC_STATS_USER(sock_net(sk), LINUX_MIB_TCPABORTONDATA);
19     } else if (tcp_close_state(sk)) {
20         /* We FIN if the application ate all the data before
21         * zapping the connection.
22         */
23
24         /* RED-PEN. Formally speaking, we have broken TCP state
25         * machine. State transitions:
26         *
27         * TCP_ESTABLISHED -> TCP_FIN_WAIT1
28         * TCP_SYN_RECV      -> TCP_FIN_WAIT1 (forget it, it's impossible)
29         * TCP_CLOSE_WAIT -> TCP_LAST_ACK
30         *
31         * are legal only when FIN has been sent (i.e. in window),
32         * rather than queued out of window. Purists blame.
33         *
34         * F.e. "RFC state" is ESTABLISHED,

```

```

35      * if Linux state is FIN-WAIT-1, but FIN is still not sent.
36      *
37      * The visible declinations are that sometimes
38      * we enter time-wait state, when it is not required really
39      * (harmless), do not send active resets, when they are
40      * required by specs (TCP_ESTABLISHED, TCP_CLOSE_WAIT, when
41      * they look as CLOSING or LAST_ACK for Linux)
42      * Probably, I missed some more holelets.
43      *
44      * XXX (TFO) - To start off we don't support SYN+ACK+FIN
45      * in a single packet! (May consider it later but will
46      * probably need API support or TCP_CORK SYN-ACK until
47      * data is written and socket is closed.)
48      */
49      tcp_send_fin(sk);
50  }
51
52  sk_stream_wait_close(sk, timeout);

1  adjudge_to_death:
2      state = sk->sk_state;
3      sock_hold(sk);
4      sock_orphan(sk);
5
6      /* It is the last release_sock in its life. It will remove backlog. */
7      release_sock(sk);
8
9
10     /* Now socket is owned by kernel and we acquire BH lock
11        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.
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 retrieved until protocol close. */
77
78 out:
79     bh_unlock_sock(sk);
80     local_bh_enable();
81     sock_put(sk);
82 }

```

Contents	
6.1	C 语言 . . . . . 95
6.1.1	结构体初始化 . . . . . 95
6.1.2	位字段 . . . . . 95
6.2	GCC . . . . . 96
6.2.1	__attribute__ . . . . . 96
6.2.1.1	设置变量属性 . . . . . 96
6.2.1.2	设置类型属性 . . . . . 96
6.2.1.3	设置函数属性 . . . . . 97
6.2.2	分支预测优化 . . . . . 97
6.3	操作系统 . . . . . 98
6.4	CPU . . . . . 98
6.4.1	字节序 . . . . . 98

6.1 C 语言

6.1.1 结构体初始化

```
1 typedef struct{
2     int a;
3     char ch;
4 }flag;
5 /*
6     目的是将 a 初始化为 1, ch 初始化为 'u'.
7 */
8 /* 法一：分别初始化 */
9 flag tmp;
10 tmp.a=1;
11 tmp.ch='u';
12
```

```

13  /* 法二：点式初始化 */
14  flag tmp={.a=1,.ch='u'};           //注意两个变量之间使用 , 而不是;
15  /* 法三：*/
16  flag tmp={
17              a:1,
18              ch:'u'
19  };

```

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。

### 6.1.2 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。

```

1  struct {
2      unsigned int a:1;
3      unsigned int b:1;
4      unsigned int c:1;
5  }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段边界上对齐，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。

## 6.2 GCC

### 6.2.1 \_\_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)。

#### 6.2.1.1 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。

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

#### 6.2.1.2 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。

```
1 struct S {  
2     short b[3];  
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:

```
1 struct S {  
2     short b[3];  
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。

#### 6.2.1.3 设置函数属性

### 6.2.2 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



```

1  /* include/linux/compiler.h
2   * 采用 __builtin_constant_p(x) 来忽略常量表达式。
3   */
4  # ifndef likely
5  #  define likely(x)      (__builtin_constant_p(x) ? !!(x) : __branch_check__(x, 1))
6  # endif
7  # ifndef unlikely
8  #  define unlikely(x)    (__builtin_constant_p(x) ? !(x) : __branch_check__(x, 0))
9  # 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:

```

1  # define likely(x)      __builtin_expect(!(x), 1)
2  # 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。

## 6.3 操作系统

## 6.4 CPU

### 6.4.1 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。

```

1  /* 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。
2   */
3  ntohs(__u32 x)
4  ntohs(__u16 x)
5  htonl(__u32 x)
6  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。