

Linux 内核 4.4.0 版本源码分析--传输层实现

王鹿鸣, 刘保证

2016 年 6 月 25 日

1	准备部分	1
1.1	用户层 TCP	1
1.2	探寻 tcp_prot, 地图 get~	1
1.3	RFC	2
1.3.1	RFC793——Transmission Control Protocol	2
1.3.2	RFC1323——TCP Extensions for High Performance	5
1.3.3	RFC1337——TIME-WAIT Assassination Hazards in TCP	6
1.3.4	RFC2481——A Proposal to add Explicit Congestion Notification(ECN) to IP	8
1.3.5	RFC2525——Known TCP Implementation Problems	8
1.3.6	RFC3168——The Addition of Explicit Congestion Notification (ECN) to IP	8
1.3.7	RFC6937—Proportional Rate Reduction for TCP	9
1.3.8	RFC7413——TCP Fast Open(Draft)	9
2	网络子系统相关核心数据结构	11
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	18
2.2.5	msghdr	22
2.3	inet 层相关数据结构	22
2.3.1	ip_options	22
2.3.2	inet_request_sock	23
2.3.3	inet_connection_sock_af_ops	23

2.3.4	inet_connect_sock	24
2.3.5	inet_timewait_sock	25
2.3.6	sockaddr & sockaddr_in	26
2.3.7	ip_options	27
2.4	路由相关数据结构	28
2.4.1	dst_entry	28
2.4.2	rtable	29
2.4.3	flowi	30
2.5	TCP 层相关数据结构	30
2.5.1	tcphdr	30
2.5.2	tcp_options_received	31
2.5.3	tcp_sock	32
2.5.4	tcp_request_sock	36
2.5.5	tcp_skb_cb	37
3	TCP 建立连接	39
3.1	TCP 主动打开-客户	39
3.1.1	基本流程	39
3.1.2	第一次握手: 构造并发送 SYN 包	39
3.1.3	第二次握手: 接收 SYN+ACK 包	44
3.1.4	第三次握手——发送 ACK 包	51
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	62
3.2.4	第三次握手: 接收 ACK 段	66
4	TCP 输出	72
4.1	tcp_sendmsg	72
4.2	tcp_transmit_skb	76
4.3	tcp_select_window(struct sk_buff *skb)	77
4.3.1	代码分析	77
5	TCP 输入	81
5.1	Linux 内核网络数据接收流程概览	81
5.2	自底向上调用与自顶向下调用	82
5.2.1	自底向上处理	82
5.2.2	自顶向下处理	88

6	TCP 拥塞控制	99
6.1	拥塞控制实现	99
6.1.1	拥塞控制状态机	99
6.1.2	显式拥塞通知 (ECN)	104
6.1.3	拥塞控制状态的处理及转换	106
6.2	拥塞控制引擎	110
6.2.1	接口	110
6.2.2	CUBIC 拥塞控制算法	113
7	TCP 释放连接	124
7.1	主动关闭	124
7.1.1	第一次握手——发送 FIN	124
7.1.2	第二次握手——接受 ACK	126
7.1.3	第三次握手——接受 FIN	130
7.1.4	第四次握手——发送 ACK	131
7.1.5	同时关闭	133
7.1.6	TIME_WAIT	134
7.2	被动关闭	136
7.2.1	基本流程	136
7.2.2	第一次握手：接收FIN	136
8	非核心函数分析	140
8.1	BSD Socket 层	140
8.1.1	msg_flag	140
8.1.2	数据报类型	141
8.1.3	Sock CheckSum	142
8.1.4	SK Stream	144
8.1.5	sk_stream_wait_connect	144
8.2	Inet	145
8.2.1	inet_hash_connect && __inet_hash_connect	145
8.3	Inet	145
8.3.1	inet_hash_connect && __inet_hash_connect	145
8.3.2	inet_twsk_put	147
8.4	TCP 层	148
8.4.1	TCP 相关参数	148
8.4.2	TCP 相关宏定义	152
8.4.3	TCP CheckSum	153
8.4.4	__tcp_push_pending_frames	154
8.4.5	About ACK	154
8.4.6	About Window Size and Segment Sent	158

8.4.7	<code>tcp_fin_time</code>	159
8.4.8	About Congestion Control	159
8.4.9	About Congestion Control	161
8.4.10	About Retransmit	165
8.4.11	<code>tcp_done</code>	165
8.4.12	<code>tcp_init_nondata_skb</code>	166
8.4.13	<code>before()</code> 和 <code>after()</code>	167
8.4.14	<code>tcp_shutdown</code>	167
8.4.15	<code>tcp_close</code>	168
9	附录: 基础知识	172
9.1	计算机底层知识	172
9.1.1	机器数	172
9.2	GNU/LINUX	173
9.2.1	错误处理	173
9.2.2	调试函数	177
9.3	C 语言	177
9.3.1	结构体初始化	177
9.3.2	位字段	178
9.4	GCC	178
9.4.1	<code>__attribute__</code>	178
9.4.2	分支预测优化	179
9.5	Sparse	180
9.5.1	<code>__bitwise</code>	180
9.6	操作系统	181
9.6.1	RCU	181
9.7	CPU	181
9.8	存储系统	181
9.8.1	字节序	181
9.8.2	缓存 Cache	182

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	2
1.3.1	RFC793—Transmission Control Protocol	2
1.3.1.1	TCP 状态图	2
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)	5
1.3.2.3	PAWS(Protect Against Wrapped Sequence Numbers)	6
1.3.3	RFC1337—TIME-WAIT Assassination Hazards in TCP	6
1.3.3.1	TIME-WAIT Assassination(TWA) 现象	6
1.3.3.2	TWA 的危险性及现有的解决方法	7
1.3.4	RFC2481—A Proposal to add Explicit Congestion Notification(ECN) to IP	8
1.3.5	RFC2525—Known TCP Implementation Problems	8
1.3.6	RFC3168—The Addition of Explicit Congestion Notification (ECN) to IP	8
1.3.6.1	ECN	8
1.3.6.2	TCP 中的 ECN	8
1.3.7	RFC6937-Proportional Rate Reduction for TCP	9
1.3.8	RFC7413—TCP Fast Open(Draft)	9
1.3.8.1	概要	9
1.3.8.2	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    .recvmmsg        = tcp_recvmmsg,
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。

1.3 RFC

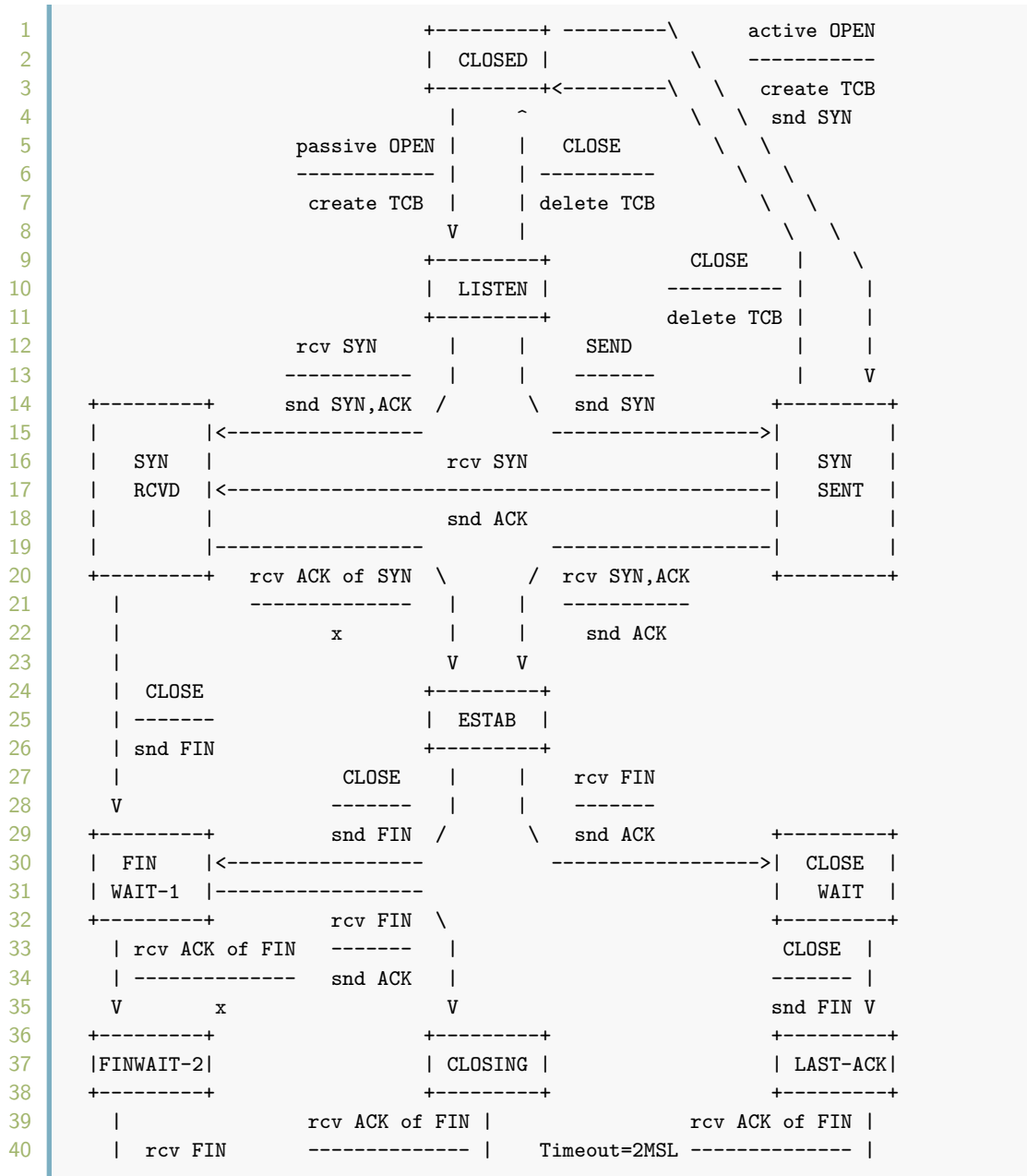
在分析 TCP 的过程中会遇到很多 RFC，在这里，我们将可能会碰到的 RFC 罗列出来，并进行一定的讨论，便于后面的分析。

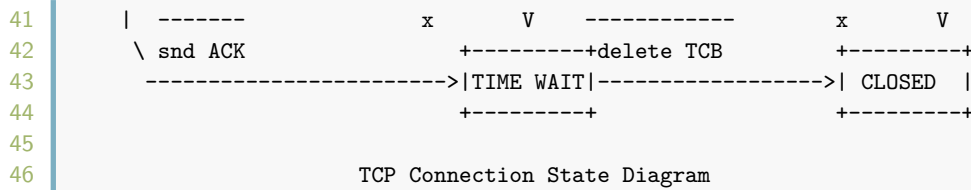
1.3.1 RFC793——Transmission Control Protocol

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：





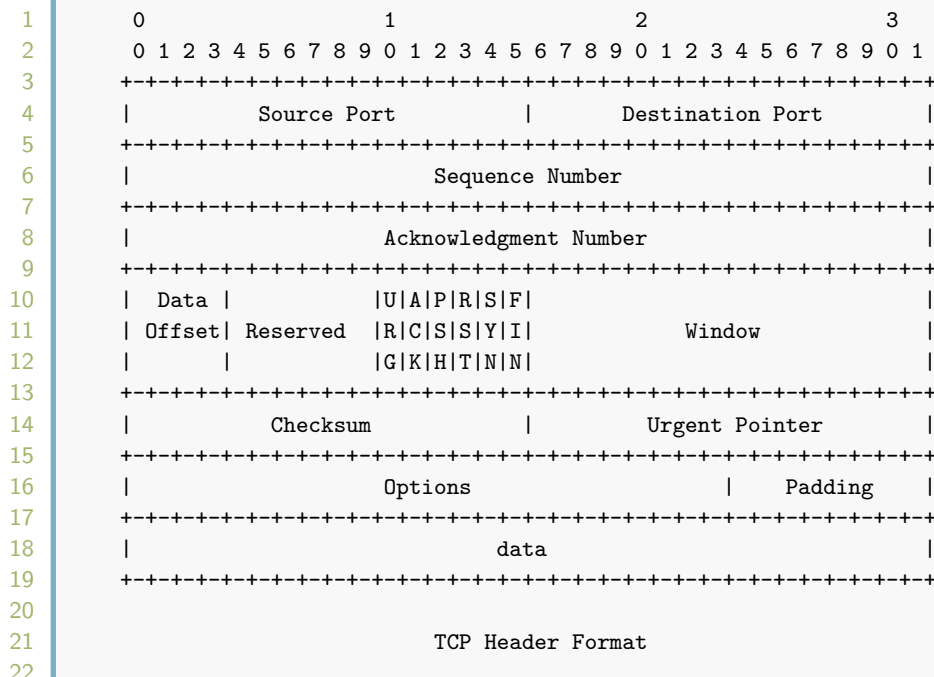
这张图对于后面的分析有很强的指导意义。

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时, 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:



23

Note that one tick mark represents one bit position.

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

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

1. 单独的一个字节，代表选项的类型例如：

```

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 7.1.6 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 RFC2481——A Proposal to add Explicit Congestion Notification(ECN) to IP

1.3.5 RFC2525——Known TCP Implementation Problems

1.3.6 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。

1.3.6.1 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。

1.3.6.2 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：

```

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

```

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。

1.3.7 RFC6937—Proportional Rate Reduction for TCP

1.3.8 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。

1.3.8.1 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。

1.3.8.2 Fast Open 选项格式

该选项的格式如下:

```

1      +-----+-----+-----+-----+-----+-----+-----+-----+
2      | Kind = 34 | Length |
3      +-----+-----+-----+-----+-----+-----+-----+-----+
4      |
5      ~ Cookie ~
6      |
7      +-----+-----+-----+-----+-----+-----+-----+-----+
8      Length      1 byte: range 6 to 18 (bytes); limited by
9                   remaining space in the options field.
10                  The number MUST be even.
11      Cookie      0, or 4 to 16 bytes (Length - 2)

```

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

CHAPTER 2

网络子系统相关核心数据结构

Contents

2.1	网络子系统数据结构架构	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	18
2.2.5	msghdr	22
2.3	inet 层相关数据结构	22
2.3.1	ip_options	22
2.3.2	inet_request_sock	23
2.3.3	inet_connection_sock_af_ops	23
2.3.4	inet_connect_sock	24
2.3.5	inet_timewait_sock	25
2.3.6	sockaddr & sockaddr_in	26
2.3.7	ip_options	27
2.4	路由相关数据结构	28
2.4.1	dst_entry	28
2.4.2	rtable	29
2.4.3	flowi	30
2.5	TCP 层相关数据结构	30
2.5.1	tcphdr	30
2.5.2	tcp_options_received	31
2.5.3	tcp_sock	32
2.5.4	tcp_request_sock	36

2.5.5	tcp_skb_cb	37
2.5.5.1	TCP_SKB_CB	37
2.5.5.2	tcp_skb_cb 结构体	37

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

```

```

45         __be32   skc_rcv_saddr;
46     };
47 };
48 union {
49     unsigned int   skc_hash;
50     __u16          skc_u16hashes[2];
51 };
52 /* skc_dport && skc_num must be grouped as well */
53 union {
54     __portpair     skc_portpair;
55     struct {
56         __be16     skc_dport;
57         __u16      skc_num;
58     };
59 };
60
61 unsigned short     skc_family;
62 volatile unsigned char skc_state;
63 unsigned char      skc_reuse:4;
64 unsigned char      skc_reuseport:1;
65 unsigned char      skc_ipv6only:1;
66 unsigned char      skc_net_refcnt:1;
67 int                skc_bound_dev_if;
68 union {
69     struct hlist_node   skc_bind_node;
70     struct hlist_nulls_node skc_portaddr_node;
71 };
72 struct proto          *skc_prot;
73 possible_net_t        skc_net;
74
75 #if IS_ENABLED(CONFIG_IPV6)
76 struct in6_addr       skc_v6_daddr;
77 struct in6_addr       skc_v6_rcv_saddr;
78 #endif
79
80 atomic64_t            skc_cookie;
81
82 /* following fields are padding to force
83 * offset(struct sock, sk_refcnt) == 128 on 64bit arches
84 * assuming IPV6 is enabled. We use this padding differently
85 * for different kind of 'sockets'
86 */
87 union {
88     unsigned long     skc_flags;
89     struct sock *skc_listener; /* request_sock */
90     struct inet_timewait_death_row *skc_tw_dr; /* inet_timewait_sock */
91 };
92 /*
93 * fields between dontcopy_begin/dontcopy_end
94 * are not copied in sock_copy()
95 */
96 /* private: */
97 int                skc_dontcopy_begin[0];
98 /* public: */
99 union {
100     struct hlist_node   skc_node;
101     struct hlist_nulls_node skc_nulls_node;
102 };

```

```

103     int          skc_tx_queue_mapping;
104     union {
105         int      skc_incoming_cpu;
106         u32      skc_rcv_wnd;
107         u32      skc_tw_rcv_nxt; /* struct tcp_timewait_sock */
108     };
109
110     atomic_t      skc_refcnt;
111     /* private: */
112     int          skc_dontcopy_end[0];
113     union {
114         u32      skc_rxhash;
115         u32      skc_window_clamp;
116         u32      skc_tw_snd_nxt; /* struct tcp_timewait_sock */
117     };
118     /* public: */
119 };

```

2.2.2 sock

```

1  /*
2  Location:
3
4      include/net/sock.h
5
6  Description:
7
8      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。
9      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
10     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
11     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。
12
13  Member:
14     * struct sock - network layer representation of sockets
15     * @_sk_common: shared layout with inet_timewait_sock
16     * @sk_shutdown: SEND_SHUTDOWN 或者 RCV_SHUTDOWN 的掩码
17     * @sk_userlocks: %SO_SNDBUF and %SO_RCVBUF settings
18     * @sk_lock: synchronizer
19     * @sk_rcvbuf: 接受缓冲区的大小（单位为字节）
20     * @sk_wq: sock wait queue and async head
21     * @sk_rx_dst: receive input route used by early demux
22     * @sk_dst_cache: destination cache
23     * @sk_policy: flow policy
24     * @sk_receive_queue: incoming packets
25     * @sk_wmem_alloc: transmit queue bytes committed
26     * @sk_write_queue: Packet sending queue
27     * @sk_omem_alloc: "o" is "option" or "other"
28     * @sk_wmem_queued: persistent queue size
29     * @sk_forward_alloc: space allocated forward
30     * @sk_napi_id: id of the last napi context to receive data for sk
31     * @sk_ll_usec: usecs to busypoll when there is no data
32     * @sk_allocation: allocation mode
33     * @sk_pacing_rate: Pacing rate (if supported by transport/packet scheduler)
34     * @sk_max_pacing_rate: Maximum pacing rate (%SO_MAX_PACING_RATE)
35     * @sk_sndbuf: size of send buffer in bytes
36     * @sk_no_check_tx: %SO_NO_CHECK setting, set checksum in TX packets
37     * @sk_no_check_rx: allow zero checksum in RX packets
38     * @sk_route_caps: route capabilities (e.g. %NETIF_F_TSO)

```

```

39  * @sk_route_nocaps: forbidden route capabilities (e.g. NETIF_F_GSO_MASK)
40  * @sk_gso_type: GSO type (e.g. %SKB_GSO_TCPV4)
41  * @sk_gso_max_size: Maximum GSO segment size to build
42  * @sk_gso_max_segs: Maximum number of GSO segments
43  * @sk_lingertime: %SO_LINGER l_linger setting
44  * @sk_backlog: always used with the per-socket spinlock held
45  * @sk_callback_lock: used with the callbacks in the end of this struct
46  * @sk_error_queue: rarely used
47  * @sk_prot_creator: sk_prot of original sock creator (see ipv6_setsockopt,
48  *                   IPV6_ADDRFORM for instance)
49  * @sk_err: last error
50  * @sk_err_soft: errors that don't cause failure but are the cause of a
51  *               persistent failure not just 'timed out'
52  * @sk_drops: raw/udp drops counter
53  * @sk_ack_backlog: current listen backlog
54  * @sk_max_ack_backlog: listen backlog set in listen()
55  * @sk_priority: %SO_PRIORITY setting
56  * @sk_cgrp_prioidx: socket group's priority map index
57  * @sk_type: socket type (%SOCK_STREAM, etc)
58  * @sk_protocol: which protocol this socket belongs in this network family
59  * @sk_peer_pid: &struct pid for this socket's peer
60  * @sk_peer_cred: %SO_PEERCRED setting
61  * @sk_rcvlowat: %SO_RCVLOWAT setting
62  * @sk_rcvtimeo: %SO_RCVTIMEO setting
63  * @sk_sndtimeo: %SO_SNDTIMEO setting
64  * @sk_thash: computed flow hash for use on transmit
65  * @sk_filter: socket filtering instructions
66  * @sk_timer: sock cleanup timer
67  * @sk_stamp: time stamp of last packet received
68  * @sk_tsflags: SO_TIMESTAMPING socket options
69  * @sk_tskey: counter to disambiguate concurrent tstamp requests
70  * @sk_socket: Identd and reporting IO signals
71  * @sk_user_data: RPC layer private data
72  * @sk_frag: cached page frag
73  * @sk_peek_off: current peek_offset value
74  * @sk_send_head: 发送队列的头指针
75  * @sk_security: used by security modules
76  * @sk_mark: generic packet mark
77  * @sk_classid: this socket's cgroup classid
78  * @sk_cgrp: this socket's cgroup-specific proto data
79  * @sk_write_pending: a write to stream socket waits to start
80  * @sk_state_change: callback to indicate change in the state of the sock
81  * @sk_data_ready: callback to indicate there is data to be processed
82  * @sk_write_space: callback to indicate there is bf sending space available
83  * @sk_error_report: callback to indicate errors (e.g. %MSG_ERRQUEUE)
84  * @sk_backlog_rcv: callback to process the backlog
85  * @sk_destruct: called at sock freeing time, i.e. when all refcnt == 0
86  */
87  struct sock {
88      /*
89       * Now struct inet_timewait_sock also uses sock_common, so please just
90       * don't add nothing before this first member (__sk_common) --acme
91       */
92      struct sock_common __sk_common;
93      #define sk_node __sk_common.skc_node
94      #define sk_nulls_node __sk_common.skc_nulls_node
95      #define sk_refcnt __sk_common.skc_refcnt
96      #define sk_tx_queue_mapping __sk_common.skc_tx_queue_mapping
97

```

```

98  #define sk_dontcopy_begin    __sk_common.skc_dontcopy_begin
99  #define sk_dontcopy_end      __sk_common.skc_dontcopy_end
100 #define sk_hash              __sk_common.skc_hash
101 #define sk_portpair          __sk_common.skc_portpair
102 #define sk_num               __sk_common.skc_num
103 #define sk_dport             __sk_common.skc_dport
104 #define sk_addrpair          __sk_common.skc_addrpair
105 #define sk_daddr             __sk_common.skc_daddr
106 #define sk_rcv_saddr         __sk_common.skc_rcv_saddr
107 #define sk_family            __sk_common.skc_family
108 #define sk_state              __sk_common.skc_state
109 #define sk_reuse              __sk_common.skc_reuse
110 #define sk_reuseport         __sk_common.skc_reuseport
111 #define sk_ipv6only          __sk_common.skc_ipv6only
112 #define sk_net_refcnt         __sk_common.skc_net_refcnt
113 #define sk_bound_dev_if      __sk_common.skc_bound_dev_if
114 #define sk_bind_node         __sk_common.skc_bind_node
115 #define sk_prot               __sk_common.skc_prot
116 #define sk_net                __sk_common.skc_net
117 #define sk_v6_daddr          __sk_common.skc_v6_daddr
118 #define sk_v6_rcv_saddr      __sk_common.skc_v6_rcv_saddr
119 #define sk_cookie             __sk_common.skc_cookie
120 #define sk_incoming_cpu       __sk_common.skc_incoming_cpu
121 #define sk_flags              __sk_common.skc_flags
122 #define sk_rthash             __sk_common.skc_rthash
123
124     socket_lock_t            sk_lock;
125     struct sk_buff_head sk_receive_queue;
126     /*
127      * The backlog queue is special, it is always used with
128      * the per-socket spinlock held and requires low latency
129      * access. Therefore we special case it's implementation.
130      * Note : rmem_alloc is in this structure to fill a hole
131      * on 64bit arches, not because its logically part of
132      * backlog.
133      */
134     struct {
135         atomic_t    rmem_alloc;
136         int         len;
137         struct sk_buff *head;
138         struct sk_buff *tail;
139     } sk_backlog;
140     #define sk_rmem_alloc sk_backlog.rmem_alloc
141     int         sk_forward_alloc;
142
143     __u32       sk_txhash;
144     #ifdef CONFIG_NET_RX_BUSY_POLL
145         unsigned int    sk_napi_id;
146         unsigned int    sk_ll_usec;
147     #endif
148     atomic_t    sk_drops;
149     int         sk_rcvbuf;
150
151     struct sk_filter __rcu *sk_filter;
152     union {
153         struct socket_wq __rcu *sk_wq;
154         struct socket_wq    *sk_wq_raw;
155     };

```

```

156 #ifdef CONFIG_XFRM
157     struct xfrm_policy __rcu *sk_policy[2];
158 #endif
159     struct dst_entry *sk_rx_dst;
160     struct dst_entry __rcu *sk_dst_cache;
161     /* Note: 32bit hole on 64bit arches */
162     atomic_t sk_wmem_alloc;
163     atomic_t sk_omem_alloc;
164     int sk_sndbuf;
165     struct sk_buff_head sk_write_queue;
166     kmemcheck_bitfield_begin(flags);
167     unsigned int sk_shutdown : 2,
168                 sk_no_check_tx : 1,
169                 sk_no_check_rx : 1,
170                 sk_userlocks : 4,
171                 sk_protocol : 8,
172                 sk_type : 16;
173 #define SK_PROTOCOL_MAX US_MAX
174     kmemcheck_bitfield_end(flags);
175     int sk_wmem_queued;
176     gfp_t sk_allocation;
177     u32 sk_pacing_rate; /* bytes per second */
178     u32 sk_max_pacing_rate;
179     netdev_features_t sk_route_caps;
180     netdev_features_t sk_route_nocaps;
181     int sk_gso_type;
182     unsigned int sk_gso_max_size;
183     u16 sk_gso_max_segs;
184     int sk_rcvlowat;
185     unsigned long sk_lingertime;
186     struct sk_buff_head sk_error_queue;
187     struct proto *sk_prot_creator;
188     rwlock_t sk_callback_lock;
189     int sk_err,
190         sk_err_soft;
191     u32 sk_ack_backlog;
192     u32 sk_max_ack_backlog;
193     __u32 sk_priority;
194 #if IS_ENABLED(CONFIG_CGROUP_NET_PRIO)
195     __u32 sk_cgrp_prioidx;
196 #endif
197     struct pid *sk_peer_pid;
198     const struct cred *sk_peer_cred;
199     long sk_rcvtimeo;
200     long sk_sndtimeo;
201     struct timer_list sk_timer;
202     ktime_t sk_stamp;
203     u16 sk_tsflags;
204     u32 sk_tskey;
205     struct socket *sk_socket;
206     void *sk_user_data;
207     struct page_frag sk_frag;
208     struct sk_buff *sk_send_head;
209     __s32 sk_peek_off;
210     int sk_write_pending;
211 #ifdef CONFIG_SECURITY
212     void *sk_security;
213 #endif
214     __u32 sk_mark;

```

```

215 #ifdef CONFIG_CGROUP_NET_CLASSID
216     u32         sk_classid;
217 #endif
218     struct cg_proto    *sk_cgrp;
219     void             (*sk_state_change)(struct sock *sk);
220     void             (*sk_data_ready)(struct sock *sk);
221     void             (*sk_write_space)(struct sock *sk);
222     void             (*sk_error_report)(struct sock *sk);
223     int              (*sk_backlog_rcv)(struct sock *sk,
224                                     struct sk_buff *skb);
225     void             (*sk_destruct)(struct sock *sk);
226 };

```

2.2.3 request_sock

```

1  /*
2  Location:
3
4      /include/net/request_sock.h
5
6  Description:
7
8      struct request_sock - mini sock to represent a connection request
9      该结构用于表示一个简单的 TCP 连接请求。
10
11  */
12  struct request_sock {
13      struct sock_common    __req_common;
14      #define rsk_refcnt      __req_common.skc_refcnt
15      #define rsk_hash        __req_common.skc_hash
16      #define rsk_listener    __req_common.skc_listener
17      #define rsk_window_clamp __req_common.skc_window_clamp
18      #define rsk_rcv_wnd     __req_common.skc_rcv_wnd
19
20      struct request_sock    *dl_next;
21      u16                    mss;
22      u8                    num_retrans; /* number of retransmits */
23      u8                    cookie_ts:1; /* syncookie: encode tcptopts in timestamp */
24      u8                    num_timeout:7; /* number of timeouts */
25      u32                    ts_recent;
26      struct timer_list      rsk_timer;
27      const struct request_sock_ops *rsk_ops;
28      struct sock            *sk;
29      u32                    *saved_syn;
30      u32                    secid;
31      u32                    peer_secid;
32  };

```

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用 `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` 中的数据报总长度 (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     nohdr:1,     /* 只引用了负载 */
8     fclone:2,    /* skbuff 克隆的情况 */
9     peeked:1,    /* peeked 表明该包已经被统计过了，无需再次统计 */
10    head_frag:1,
11    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     /* 表明该 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。

2.2.5 msghdr

```

1  /* Location: include/linux/socket.h */
2  struct msghdr {
3      void            *msg_name;        /* 指向 socket 地址结构体的指针 */
4      int             msg_namelen;     /* socket 地址结构体的大小 */
5      struct iov_iter msg_iter;        /* 数据 */
6      void            *msg_control;    /* 辅助数据 */
7      __kernel_size_t msg_controllen; /* 辅助数据缓冲区大小 */
8      unsigned int    msg_flags;       /* 收到的消息所带的标记 */
9      struct kiocb     *msg_iocb;      /* 指向 iocb 的指针（用于异步请求） */
10 };

```

2.3 inet 层相关数据结构

2.3.1 ip_options

```

1  /*
2  Location:
3
4  Description:
5      @faddr - Saved first hop address
6      @nexthop - Saved nexthop address in LSRR and SSRR
7      @is_strictroute - Strict source route
8      @srr_is_hit - Packet destination addr was our one
9      @is_changed - IP checksum more not valid
10     @rr_needaddr - Need to record addr of outgoing dev
11     @ts_needtime - Need to record timestamp
12     @ts_needaddr - Need to record addr of outgoing dev
13  */
14  struct ip_options {
15      __be32          faddr;
16      __be32          nexthop;
17      unsigned char    optlen;
18      unsigned char    srr;
19      unsigned char    rr;
20      unsigned char    ts;
21      unsigned char    is_strictroute:1,
22                      srr_is_hit:1,
23                      is_changed:1,
24                      rr_needaddr:1,
25                      ts_needtime:1,
26                      ts_needaddr:1;
27      unsigned char    router_alert;
28      unsigned char    cipso;
29      unsigned char    __pad2;
30      unsigned char    __data[0];
31  };

```

2.3.2 inet_request_sock

该结构位于 `/include/net/inet_sock.h` 中。

这个结构的功能呢？

```

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 };

```

2.3.3 inet_connection_sock_af_ops

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

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

```

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_recv_sock)(const struct sock *sk, struct sk_buff *skb,
8                                   struct request_sock *req,
8                                   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  };

```

2.3.4 inet_connect_sock

该结构位于 `/include/net/inet_connect_sock.h` 中，它是所有面向传输控制块的代表。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:         拥塞控制状态
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)]; /* 拥塞控制算法私有空间 */
69 #define ICSK_CA_PRIV_SIZE (8 * sizeof(u64))
70 };

```

2.3.5 inet_timewait_sock

在进入等待关闭的状态时，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_ipv6only    __tw_common.skc_ipv6only
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  /* 超时计时器 */
51  struct timer_list tw_timer;
52  struct inet_bind_bucket *tw_tb;
53  };

```

2.3.6 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。

2.3.7 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, 能够很好地实现 `dst_entry` 与 `rtable` 的转换, 而 `dst_entry` 中又包含邻居项相关的信息, 实现了路由缓存与邻居子系统的关联。

```

1  struct rtable {
2      /* 存储缓存路由项中独立于协议的信息 */
3      struct dst_entry    dst;
4
5      int                rt_genid;
6      /* 表示路由表项的一些特性和标志 */
7      unsigned int        rt_flags;
8      __u16                rt_type;
9      __u8                rt_is_input;
10     __u8                rt_uses_gateway;
11
12     int                rt_iif;
13
14     /* Info on neighbour */
15     __be32                rt_gateway;
16
17     /* Miscellaneous cached information */
18     u32                rt_pmtu;
19
20     u32                rt_table_id;

```

```

21
22     struct list_head    rt_uncached;
23     struct uncached_list *rt_uncached_list;
24 };

```

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_recent 的时间
5                               用于检测 ts_recent 的有效性：如果自从该事件之后已经
6                               经过了超过 24 天的时间，则认为 ts_recent 已无效。
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。当一个含有最后
10                   发送 ACK 中确认序号的段到达时，该段中的时间戳被保存在
11                   ts_recent 中。而当下一个待发送的 TCP 段的时间戳值是由
12                   SKB 中 TCP 控制块的成员 when 填入的，when 字段值是由协议
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 段的时间戳选项中的时间戳值。
17                   */
18      u32 rcv_tsecr; /* Time stamp echo reply
19                   保存最近一次接收到对端的 TCP 段的时间戳选项中的时间戳
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 时间戳选项，1 为有，
24                           0 为无。
25                           */
26      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      dsack : 1, /* D-SACK is scheduled */
30      wscale_ok : 1, /* Wscale seen on SYN packet
31                   标志接收方是否支持窗口扩大因子，只出现在 SYN 段中。
32                   */
33      sack_ok : 4, /* SACK seen on SYN packet
34                   标记是否对方提供 SACK 服务
35                   */
36      snd_wscale : 4, /* Window scaling received from sender */
37      rcv_wscale : 4; /* Window scaling to send to receiver */
38      u8 num_sacks; /* Number of SACK blocks */
39      u16 user_mss; /* mss requested by user in ioctl */
40      u16 mss_clamp; /* Maximal mss, negotiated at connection setup */
41 };

```

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_wl1;        /* 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                          滑动窗口最大值
65                          */
66     u32 rcv_ssthresh;   /* Current window clamp */
67
68     /* Information of the most recently (s)acked skb */
69     struct tcp_rack {
70         struct skb_mstamp mstamp; /* (Re)sent time of the skb */
71         u8 advanced; /* mstamp advanced since last lost marking */
72         u8 reord;    /* reordering detected */
73     } rack;
74     u16 advmss;      /* Advertised MSS
75                      本端能接收的 MSS 上限，在建立时用来通告对方。
76                      */
77     u8 unused;
78     u8 nonagle       : 4, /* Disable Nagle algorithm? */
79     thin_lto         : 1, /* Use linear timeouts for thin streams */
80     thin_dupack      : 1, /* Fast retransmit on first dupack */
81     repair           : 1,
82     frto             : 1; /* F-RTO (RFC5682) activated in CA_Loss */
83     u8 repair_queue;
84     u8 do_early_retrans:1, /* Enable RFC5827 early-retransmit */
85     syn_data:1, /* SYN includes data */
86     syn_fastopen:1, /* SYN includes Fast Open option */
87     syn_fastopen_exp:1, /* SYN includes Fast Open exp. option */
88     syn_data_acked:1, /* data in SYN is acked by SYN-ACK */
89     save_syn:1, /* Save headers of SYN packet */
90     is_cwnd_limited:1, /* forward progress limited by snd_cwnd? */
91     u32 tlp_high_seq; /* snd_nxt at the time of TLP retransmit. */
92
93     /* RTT measurement */
94     u32 srtt_us; /* smoothed round trip time < 3 in usecs */
95     u32 mdev_us; /* medium deviation */
96     u32 mdev_max_us; /* maximal mdev for the last rtt period */
97     u32 rttvar_us; /* smoothed mdev_max */
98     u32 rtt_seq; /* sequence number to update rttvar */
99     struct rtt_meas {
100         u32 rtt, ts; /* RTT in usec and sampling time in jiffies. */
101     } rtt_min[3];
102
103     u32 packets_out; /* Packets which are "in flight"
104                      发送方发送出去但是还未得到确认的 TCP 段的数目
105                      packets_out=SND.NXT-SND.UNA
106                      */
107     u32 retrans_out; /* Retransmitted packets out
108                      重传并且还未得到确认的 TCP 段的数目
109                      */
110     u32 max_packets_out; /* max packets_out in last window */

```

```

111     u32 max_packets_seq; /* right edge of max_packets_out flight */
112
113     u16 urg_data; /* Saved octet of OOB data and control flags */
114     u8 ecn_flags; /* ECN status bits. */
115     u8 keepalive_probes; /* num of allowed keep alive probes */
116     u32 reordering; /* Packet reordering metric. */
117     u32 snd_up; /* Urgent pointer */
118
119 /*
120  * Options received (usually on last packet, some only on SYN packets).
121  */
122     struct tcp_options_received rx_opt;
123
124 /*
125  * Slow start and congestion control (see also Nagle, and Karn & Partridge)
126  */
127     u32 snd_ssthresh; /* Slow start size threshold */
128     u32 snd_cwnd; /* Sending congestion window */
129     u32 snd_cwnd_cnt; /* Linear increase counter
130                      自从上次调整拥塞窗口到目前为止接收到的总 ACK 段数。
131                      如果该字段为零，则说明已经调整了拥塞窗口，且到目前
132                      为止还没有接收到 ACK 段。调整拥塞窗口之后，每接收到
133                      一个 ACK，ACK 段就会使 snd_cwnd_cnt 加 1。
134                      */
135     u32 snd_cwnd_clamp; /* Do not allow snd_cwnd to grow above this */
136     u32 snd_cwnd_used;
137     u32 snd_cwnd_stamp; /* 记录最近一次检验拥塞窗口的时间
138                      在拥塞期间，接收到 ACK 后会进行
139                      拥塞窗口的检验。而在非拥塞期间，为了防止由于应用
140                      程序限制而造成拥塞窗口失效，因此在成功发送段后，
141                      如果有必要也会检验拥塞窗口。
142                      */
143     u32 prior_cwnd; /* Congestion window at start of Recovery. */
144     u32 prr_delivered; /* Number of newly delivered packets to
145                      * receiver in Recovery. */
146     u32 prr_out; /* Total number of pkts sent during Recovery. */
147
148     u32 rcv_wnd; /* Current receiver window */
149     u32 write_seq; /* Tail(+1) of data held in tcp send buffer */
150     u32 notsent_lowat; /* TCP_NOTSENT_LOWAT */
151     u32 pushed_seq; /* Last pushed seq, required to talk to windows */
152     u32 lost_out; /* Lost packets
153                  丢失的数据报
154                  */
155     u32 sacked_out; /* SACK'd packets
156                  启用 ACK 时，通过 SACK 的 TCP 选项标识已接收到的段的数量。
157                  不启用 SACK 时，标识接收到的重复确认的次数，该值在接收到确认新数据段时被清除。
158                  */
159     u32 fackets_out; /* FACK'd packets
160                  记录 SND.UNA 与 (SACK 选项中目前接收方收到的段中最高序号段) 之间的段数。FACK
161                  用 SACK 选项来计算丢失在网络中的段数
162                  lost_out=fackets_out-sacked_out
163                  left_out=fackets_out
164                  找时间画一张图.???? remain ro do.
165                  */
166
167 /* from STCP, retrans queue hinting */
168     struct sk_buff* lost_skb_hint;
169     struct sk_buff* retransmit_skb_hint;

```



```

170
171  /* 000 segments go in this list. Note that socket lock must be held,
172   * as we do not use sk_buff_head lock.
173   */
174  struct sk_buff_head out_of_order_queue;
175
176  /* SACKs data, these 2 need to be together (see tcp_options_write) */
177  struct tcp_sack_block duplicate_sack[1]; /* D-SACK block */
178  struct tcp_sack_block selective_acks[4]; /* The SACKS themselves */
179
180  struct tcp_sack_block rcv_sack_cache[4];
181
182  struct sk_buff *highest_sack; /* skb just after the highest
183   * skb with SACKed bit set
184   * (validity guaranteed only if
185   * sacked_out > 0)
186   */
187
188  int lost_cnt_hint;
189  u32 retransmit_high; /* L-bits may be on up to this seqno */
190
191  u32 prior_ssthresh; /* ssthresh saved at recovery start */
192  u32 high_seq; /* snd_nxt at onset of congestion
193   * 开始拥塞的时候下一个要发送的序号字节
194   */
195
196  u32 retrans_stamp; /* Timestamp of the last retransmit,
197   * also used in SYN-SENT to remember stamp of
198   * the first SYN. */
199  u32 undo_marker; /* snd_una upon a new recovery episode.
200   * 在使用 F-RTO 算法进行发送超时处理, 或进入 Recovery 进行重传,
201   * 或进入 Loss 开始慢启动时, 记录当时 SND.UNA, 标记重传起始点。
202   * 它是检测是否可以进行拥塞控制撤销的条件之一, 一般在完成
203   * 拥塞撤销操作或进入拥塞控制 Loss 状态后会清零。
204   */
205  int undo_retrans; /* number of undoable retransmissions.
206   * 在恢复拥塞控制之前可进行撤销的重传段数。在进入 FTRTO 算法或
207   * 拥塞状态 Loss 时, 清零, 在重传时计数, 是检测是否可以进行拥塞
208   * 撤销的条件之一。
209   */
210  u32 total_retrans; /* Total retransmits for entire connection */
211
212  u32 urg_seq; /* Seq of received urgent pointer */
213  unsigned int keepalive_time; /* time before keep alive takes place */
214  unsigned int keepalive_intvl; /* time interval between keep alive probes */
215
216  int linger2;
217
218  /* Receiver side RTT estimation */
219  struct {
220      u32 rtt;
221      u32 seq;
222      u32 time;
223  } rcv_rtt_est;
224
225  /* Receiver queue space */
226  struct {
227      int space;

```

```

228         u32 seq;
229         u32 time;
230     } rcvq_space;
231
232     /* TCP-specific MTU probe information. */
233     struct {
234         u32 probe_seq_start;
235         u32 probe_seq_end;
236     } mtu_probe;
237     u32 mtu_info; /* We received an ICMP_FRAG_NEEDED / ICMPV6_PKT_TOOBIG
238                  * while socket was owned by user.
239                  */
240
241     #ifdef CONFIG_TCP_MD5SIG
242     /* TCP AF-Specific parts; only used by MD5 Signature support so far */
243     const struct tcp_sock_af_ops *af_specific;
244
245     /* TCP MD5 Signature Option information */
246     struct tcp_md5sig_info __rcu *md5sig_info;
247     #endif
248
249     /* TCP fastopen related information */
250     struct tcp_fastopen_request *fastopen_req;
251     /* fastopen_rsk points to request_sock that resulted in this big
252      * socket. Used to retransmit SYNACKs etc.
253      */
254     struct request_sock *fastopen_rsk;
255     u32 *saved_syn;
256 };

```

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。该结构是 TCP 层在 SKB 区的私有信息控制块。对这个私有信息控制块的赋值一般在本层接收到段或发送段之前进行。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

在 TCP 层, 用该宏访问给定的`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

Contents

3.1	TCP 主动打开-客户	39
3.1.1	基本流程	39
3.1.2	第一次握手：构造并发送 SYN 包	39
3.1.2.1	基本调用关系	39
3.1.2.2	tcp_v4_connect	40
3.1.2.3	tcp_connect	43
3.1.3	第二次握手：接收 SYN+ACK 包	44
3.1.3.1	基本调用关系	44
3.1.3.2	tcp_rcv_state_process	44
3.1.3.3	tcp_rcv_synsent_state_process	45
3.1.4	第三次握手——发送 ACK 包	51
3.1.4.1	基本调用关系	51
3.1.4.2	tcp_send_ack	51
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
3.2.2.1	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	55
3.2.2.4	tcp_rcv_state_process	55
3.2.2.5	tcp_v4_conn_request && tcp_conn_request	57
3.2.2.6	inet_csk_reqsk_queue_add	61
3.2.2.7	inet_csk_reqsk_queue_hash_add	61
3.2.3	第二次握手：发送 SYN+ACK 段	62

3.2.3.1	第二次函数调用关系	62
3.2.3.2	tcp_v4_send_synack	62
3.2.3.3	tcp_make_synack	63
3.2.4	第三次握手: 接收 ACK 段	66
3.2.4.1	第三次握手函数调用关系图	66
3.2.4.2	tcp_v4_do_rcv	66
3.2.4.3	tcp_v4_cookie_check	68
3.2.4.4	tcp_child_process	68
3.2.4.5	tcp_rcv_state_process	69

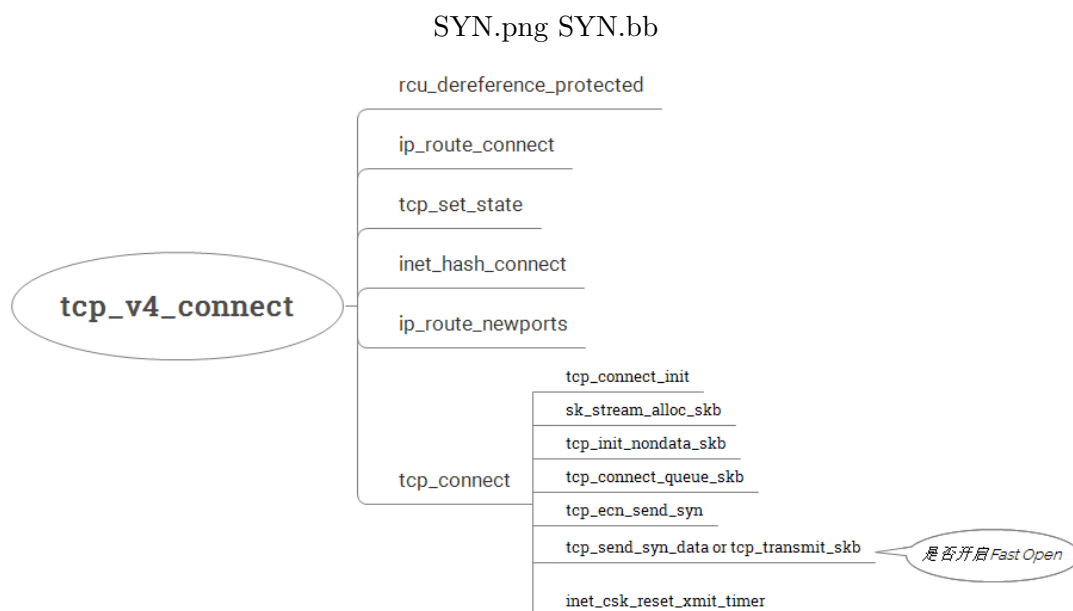
3.1 TCP 主动打开-客户

3.1.1 基本流程

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

3.1.2 第一次握手: 构造并发送 SYN 包

3.1.2.1 基本调用关系



3.1.2.2 tcp_v4_connect

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

```

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     /*
19         sockaddr_in 结构体用于描述一个 Internet (IP) 套接字的地址
20     */
21     struct sockaddr_in *usin = (struct sockaddr_in *)uaddr;
22     struct inet_sock *inet = inet_sk(sk);
23     struct tcp_sock *tp = tcp_sk(sk);
24     __be16 orig_sport, orig_dport;
25     __be32 daddr, nexthop; //网络协议主要使用大端存储, be32 means 32 bits stored w
26     struct flowi4 *fl4;
27     struct rtable *rt;
28     int err;
29     struct ip_options_rcu *inet_opt;
30
31     /* 检验目的地址长度是否有效 */
32     if (addr_len < sizeof(struct sockaddr_in))
33         return -EINVAL; //Invalid argument 错误码为 22
34
35     /* 检验协议族是否正确 */
36     if (usin->sin_family != AF_INET) //IPV4 地址域
37         return -EAFNOSUPPORT; //Address family not supported by protocol
38
39     /* 将下一跳地址和目的地址暂时设置为用户传入的 IP 地址 */
40     nexthop = daddr = usin->sin_addr.s_addr;
41     inet_opt = rcu_dereference_protected(inet->inet_opt,
42                                         sock_owned_by_user(sk));
43     /* 如果选择源地址路由, 则将下一跳地址设置为 IP 选项中的 faddr-first hop address*/
44     if (inet_opt && inet_opt->opt.srr) {
45         if (!daddr)
46             return -EINVAL;
47         nexthop = inet_opt->opt.faddr;
48     }

```

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; //对应于 ipv4 的流
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     /* 判断指针是否有效 */
11     if (IS_ERR(rt)) {
12         err = PTR_ERR(rt);
13         if (err == -ENETUNREACH)    //Network is unreachable
14             IP_INC_STATS(sock_net(sk), IPSTATS_MIB_OUTNOROUTES);
15         return err;
16     }
17
18     /*TCP 不能使用类型为组播或多播的路由缓存项 */
19     if (rt->rt_flags & (RTCF_MULTICAST | RTCF_BROADCAST)) {
20         ip_rt_put(rt);
21         return -ENETUNREACH;    //Network is unreachable
22     }
23
24     /* 如果没有开启源路由功能, 则采用查找到的缓存项 */
25     if (!inet_opt || !inet_opt->opt.srr)
26         daddr = fl4->daddr;
27
28     /* 如果没有设置源地址, 则设置为缓存项中的源地址 */
29     if (!inet->inet_saddr)
30         inet->inet_saddr = fl4->saddr;
31     sk_rcv_saddr_set(sk, inet->inet_saddr);
32
33     /* 如果该传输控制块的时间戳已被使用过, 则重置各状态
34        rx_opt: tcp_options_received
35     */
36     if (tp->rx_opt.ts_recent_stamp && inet->inet_daddr != daddr) {
37         /* Reset inherited state */
38         /* 下一个待发送的 TCP 段中的时间戳回显值 */
39         tp->rx_opt.ts_recent = 0;
40         /* 从接收到的段中取出时间戳 */
41         tp->rx_opt.ts_recent_stamp = 0;
42         /*What does it means repair*/
43         if (likely(!tp->repair))
44             tp->write_seq = 0;
45     }
46
47     /* 在启用了 tw_recycle:time wait recycle 的情况下, 重设时间戳
48        它用来快速回收 TIME_WAIT 连接.
49     */
50     if (tcp_death_row.sysctl_tw_recycle &&
51         !tp->rx_opt.ts_recent_stamp && fl4->daddr == daddr)
52         tcp_fetch_timewait_stamp(sk, &rt->dst);
53
54     /* 设置传输控制块 */
55     inet->inet_dport = usin->sin_port;
56     sk_daddr_set(sk, daddr);
57
58     inet_csk(sk)->icsk_ext_hdr_len = 0;
59     if (inet_opt)
60         inet_csk(sk)->icsk_ext_hdr_len = inet_opt->opt.optlen;
61
62     /* 设置 MSS 大小 */
63     tp->rx_opt.mss_clamp = TCP_MSS_DEFAULT;
64

```



```

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

```

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

```

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;
```

3.1.2.3 tcp_connect

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

```
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;                        //No buffer space available
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 (Explicit Congestion Notification,) 支持
42        显示拥塞控制
43     */
44     tcp_ecn_send_syn(sk, buff);                //what does this function do?
45
46     /* 发送 SYN 包, 这里同时还考虑了 Fast Open 的情况 */
47     err = tp->fastopen_req ? tcp_send_syn_data(sk, buff) :
48         tcp_transmit_skb(sk, buff, 1, sk->sk_allocation);
49     if (err == -ECONNREFUSED)                  /*Connection refused*/
50         return err;
51 }
```

```

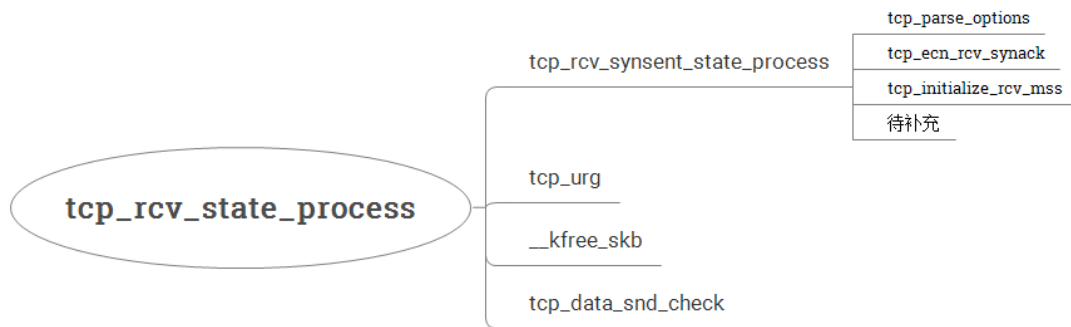
52      /* We change tp->snd_nxt after the tcp_transmit_skb() call
53      * in order to make this packet get counted in tcpOutSegs.
54      */
55      tp->snd_nxt = tp->write_seq;          //send next 下一个待发送的序列
56      /*pushed_seq means Last pushed seq, required to talk to windows??*/
57      tp->pushed_seq = tp->write_seq;
58      TCP_INC_STATS(sock_net(sk), TCP_MIB_ACTIVEOPENS);
59
60      /* 设定超时重传定时器 */
61      inet_csk_reset_xmit_timer(sk, ICSK_TIME_RETRANS,
62                                inet_csk(sk)->icsk_rto, TCP_RTO_MAX);
63      return 0;
64  }

```

3.1.3 第二次握手：接收 SYN+ACK 包

3.1.3.1 基本调用关系

SYN +ACK.png SYN +ACK.bb



3.1.3.2 tcp_rcv_state_process

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

```

1  /*
2  Location:
3
4      net/ipv4/tcp_input.c
5
6  Function:
7
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      /* 标识接收到的 TCP 段不存在 TCP 时间戳选项 */
28      tp->rx_opt.saw_tstamp = 0;
29
30      switch (sk->sk_state) {
31          case TCP_CLOSE:
32              /* CLOSE 状态的处理代码 */
33
34          case TCP_LISTEN:
35              /* LISTEN 状态的处理代码 */
36
37          case TCP_SYN_SENT:
38              /* 处理接收到的数据段 */
39              queued = tcp_rcv_synsent_state_process(sk, skb, th);
40              if (queued >= 0)
41                  return queued;
42
43              /* 处理紧急数据并检测是否有数据需要发送 */
44              tcp_urg(sk, skb, th);
45              __kfree_skb(skb);
46              tcp_data_snd_check(sk);
47              return 0;
48      }
49
50      /* 处理其他情况的代码 */
51  }

```

3.1.3.3 tcp_rcv_synsent_state_process

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

```

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     /*
28      * rcv_tsecr 保存最近一次接收到对端的 TCP 段的时间
29      * 戳选项中的时间戳回显应答。???
30
31      * tsoffset means timestamp offset
32      */
33     if (tp->rx_opt.saw_tstamp && tp->rx_opt.rcv_tsecr)
34         tp->rx_opt.rcv_tsecr -= tp->tsoffset;

```

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

```

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

```

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

```

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)
10         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,我们在 1.3.2 中进行了详细讨论。接下来手动设定了窗口缩放相关的参数,使得缩放不生效。

```

1     /* RFC1323: The window in SYN & SYN/ACK segments is
2      * never scaled.
3      */
4     tp->snd_wnd = ntohs(th->window);    //network to host short int
5
6     /* wscale_ok 标志接收方是否支持窗口扩大因子
7      * window_clamp 滑动窗口最大值
8      */
9     if (!tp->rx_opt.wscale_ok) {
10         tp->rx_opt.snd_wscale = tp->rx_opt.rcv_wscale = 0;

```

```

11         tp->window_clamp = min(tp->window_clamp, 65535U);
12     }

```

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) //sysctl: system control
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 ??? Confused*/
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     #if 0
56         /* Note, we could accept data and URG from this segment.
57          * There are no obstacles to make this (except that we must
58          * either change tcp_recvmss() to prevent it from returning data
59          * before 3WHS completes per RFC793, or employ TCP Fast Open).
60          *
61          * However, if we ignore data in ACKless segments sometimes,
62          * we have no reasons to accept it sometimes.
63          * Also, seems the code doing it in step6 of tcp_rcv_state_process
64          * is not flawless. So, discard packet for sanity.
65          * Uncomment this return to process the data.
66          */
67         return -1;
68     #else
69         goto discard;
70     #endif
71 }
72 /* "fifth, if neither of the SYN or RST bits is set then
73  * drop the segment and return."
74  */
75
76 discard_and_undo:
77     tcp_clear_options(&tp->rx_opt);
78     tp->rx_opt.mss_clamp = saved_clamp;
79     goto discard;
80
81 reset_and_undo:
82     tcp_clear_options(&tp->rx_opt);
83     tp->rx_opt.mss_clamp = saved_clamp;
84     return 1;
85 }

```

3.1.4 第三次握手——发送 ACK 包

3.1.4.1 基本调用关系

3.1.4.2 tcp_send_ack

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

```

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

```

ACK.png ACK.bb



```

8      This routine sends an ack and also updates the window.
9
10     Parameter:
11
12     sk: 传输控制块
13     该函数用于发送 ACK, 并更新窗口的大小 */
14     void tcp_send_ack(struct sock *sk)
15     {
16         struct sk_buff *buff;
17
18         /* 如果当前的套接字已经被关闭了, 那么直接返回。 */
19         if (sk->sk_state == TCP_CLOSE)
20             return;
21         /*what does it using for?*/
22         tcp_ca_event(sk, CA_EVENT_NON_DELAYED_ACK);
23
24         /* We are not putting this on the write queue, so
25          * tcp_transmit_skb() will set the ownership to this
26          * sock.
27          * 为数据包分配空间
28          */
29         buff = alloc_skb(MAX_TCP_HEADER, sk_gfp_atomic(sk, GFP_ATOMIC));
30         if (!buff) {
31             inet_csk_schedule_ack(sk);
32             inet_csk(sk)->icsk_ack.ato = TCP_ATO_MIN;
33             inet_csk_reset_xmit_timer(sk, ICSK_TIME_DACK,
34                                     TCP_DELACK_MAX, TCP_RTO_MAX);
35             return;
36         }
37

```

```

38      /* 初始化 ACK 包 */
39      skb_reserve(buff, MAX_TCP_HEADER);
40      tcp_init_nondata_skb(buff, tcp_acceptable_seq(sk), TCPCHDR_ACK);
41
42      /* We do not want pure acks influencing TCP Small Queues or fq/pacing
43       * too much.
44       * SKB_TRUESIZE(max(1 .. 66, MAX_TCP_HEADER)) is unfortunately ~784
45       * We also avoid tcp_wfree() overhead (cache line miss accessing
46       * tp->tsq_flags) by using regular sock_wfree()
47       */
48      skb_set_tcp_pure_ack(buff);
49
50      /* 添加时间戳并发送 ACK 包 */
51      skb_mstamp_get(&buff->skb_mstamp);
52      tcp_transmit_skb(sk, buff, 0, sk_gfp_atomic(sk, GFP_ATOMIC));
53  }

```

3.2 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。

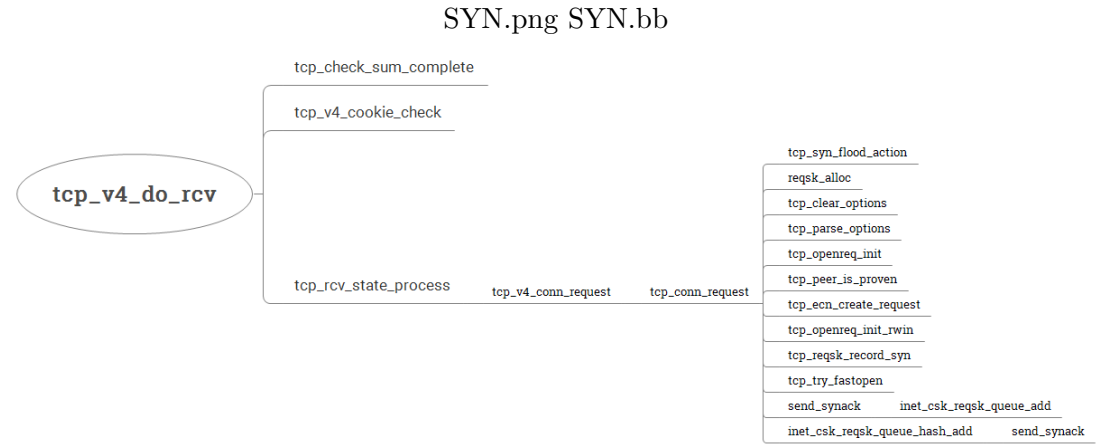
客户端一般就接着回 ack。这时，我们能从 ack 中，取出信息，在一堆 `request_sock` 匹配，看看是否之前有这个 ack 对应的 syn 发过来过。如果之前发过 syn，那么我们现在就能找到 `request_sock`，也就是客户端 syn 时建立的 `request_sock`。此时，我们内核才会为这条流创建 sock 结构体，毕竟，sock 结构体比 `request_sock` 大的多，犯不着三次握手都没建立起来我就建立一个大的结构体。当三次握手建立以后，内核就建立一个相对完整的 sock。所谓相对完整，其实也是不完整。因为如果写过 socket 程序，你就知道，所谓的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

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

3.2.2 第一次握手：接受 SYN 段

3.2.2.1 第一次握手函数调用关系

如图所示。



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。

```
1  /*
2  Location:
3
4      /net/ipv4/tcp_ipv4.c
5
6  Function:
7      The socket must have it's spinlock held when we get
8      here, unless it is a TCP_LISTEN socket.
9
10     We have a potential double-lock case here, so even when
11     doing backlog processing we use the BH locking scheme.
12     This is because we cannot sleep with the original spinlock
13     held.
14
15  Parameter:
16
17     sk: 传输控制块
18     skb: 传输控制块缓冲区
19  */
20  int tcp_v4_do_rcv(struct sock *sk, struct sk_buff *skb)
21  {
22     struct sock *rsk;
23
24     /* 省略无关代码 */
25
26     if (tcp_checksum_complete(skb))
27         goto csum_err;
28
29     if (sk->sk_state == TCP_LISTEN) {
30         struct sock *nsk = tcp_v4_cookie_check(sk, skb);
```

```

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

```

首先，程序先基于伪首部累加和进行全包的校验和，判断包是否传输正确。

其次，程序会进行相应的 cookie 检查。

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

3.2.2.3 tcp_v4_cookie_check

该函数如下：

```

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

```

```

12         sk: 传输控制块
13         skb: 传输控制块缓冲区。
14     */
15     static struct sock *tcp_v4_cookie_check(struct sock *sk, struct sk_buff *skb)
16     {
17         #ifdef CONFIG_SYN_COOKIES
18             const struct tcphdr *th = tcp_hdr(skb);
19
20             if (!th->syn)
21                 sk = cookie_v4_check(sk, skb);
22         #endif
23         return sk;
24     }

```

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     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    Parameter
14
15        sk: 传输控制块
16        skb: 传输控制块缓冲区
17    */
18
19    int tcp_rcv_state_process(struct sock *sk, struct sk_buff *skb)
20    {
21        struct tcp_sock *tp = tcp_sk(sk);
22        struct inet_connection_sock *icsk = inet_csk(sk);
23        const struct tcphdr *th = tcp_hdr(skb);
24        struct request_sock *req;
25        int queued = 0;
26        bool acceptable;
27
28        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 */
29
30        switch (sk->sk_state) {
31            /* 省略无关代码 */
32
33            case TCP_LISTEN:
34                if (th->ack)
35                    return 1;
36
37                if (th->rst)

```

```

38         goto discard;
39
40     if (th->syn) {
41         if (th->fin)
42             goto discard;
43         if (icsk->icsk_af_ops->conn_request(sk, skb) < 0)
44             return 1;
45
46         /* Now we have several options: In theory there is
47          * nothing else in the frame. KA9Q has an option to
48          * send data with the syn, BSD accepts data with the
49          * syn up to the [to be] advertised window and
50          * Solaris 2.1 gives you a protocol error. For now
51          * we just ignore it, that fits the spec precisely
52          * and avoids incompatibilities. It would be nice in
53          * future to drop through and process the data.
54          *
55          * Now that TTCP is starting to be used we ought to
56          * queue this data.
57          * But, this leaves one open to an easy denial of
58          * service attack, and SYN cookies can't defend
59          * against this problem. So, we drop the data
60          * in the interest of security over speed unless
61          * it's still in use.
62          */
63         kfree_skb(skb);
64         return 0;
65     }
66     goto discard;
67
68     /* 省略无关代码 */
69 discard:
70     __kfree_skb(skb);
71 }
72 return 0;
73 }

```

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

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

3.2.2.5 tcp_v4_conn_request && tcp_conn_request

该函数如下：

```

1  /* why make two functions here?
2  Location
3
4  /net/ipv4/tcp_ipv4.c
5

```

```

6  Function:
7
8      服务端用来处理客户端连接请求的函数。
9
10 Parameter:
11
12     sk: 传输控制块
13     skb: 传输控制块缓冲区
14
15 */
16 int tcp_v4_conn_request(struct sock *sk, struct sk_buff *skb)
17 {
18     /* Never answer to SYNs send to broadcast or multicast */
19     if (skb_rtable(skb)->rt_flags & (RTCF_BROADCAST | RTCF_MULTICAST))
20         goto drop;
21
22     return tcp_conn_request(&tcp_request_sock_ops,
23                             &tcp_request_sock_ipv4_ops, sk, skb);
24
25 drop:
26     NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_LISTENDROPS);
27     return 0;
28 }

```

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

```

1  /*
2  Location
3
4      /net/ipv4/tcp_ipv4.c
5
6  Function:
7
8      服务端用来处理客户端连接请求的函数。
9
10 Parameter:
11
12     rsk_ops: 请求控制块的函数接口?? what is the difference?
13     af_ops: TCP 请求块的函数接口
14     sk: 传输控制块
15     skb: 传输控制块缓冲区
16
17 */
18 int tcp_conn_request(struct request_sock_ops *rsk_ops,
19                     const struct tcp_request_sock_ops *af_ops,
20                     struct sock *sk, struct sk_buff *skb)
21 {
22     /* 初始化 len 字段 */
23     struct tcp_fastopen_cookie foc = { .len = -1 };
24     /*tw: time wait    isn: initial sequence num 初始化序列号 */
25     __u32 isn = TCP_SKB_CB(skb)->tcp_tw_isn;
26     struct tcp_options_received tmp_opt;
27     struct tcp_sock *tp = tcp_sk(sk);
28     struct sock *fastopen_sk = NULL;
29     struct dst_entry *dst = NULL;
30     struct request_sock *req;
31     /* 标志是否开启了 SYN_COOKIE 选项 */
32     bool want_cookie = false;
33     /* 路由查找相关的数据结构 */

```



```

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

```

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

```

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

```

warm entries???

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

```

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

```

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

```

1     tcp_rsk(req)->af_specific = af_ops;

```

这一步进行的是为了保护 BGP 会话。???

```

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

```

之后, 清除 TCP 选项, 初始化 `mss_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_tstamp)
2         tcp_clear_options(&tmp_opt);

```

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

```

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_timestamp)) {
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_timestamp)) {
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。

```

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

```

```

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

```

3.2.2.6 inet_csk_reqsk_queue_add

```

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

```

这一个函数所进行的操作就是直接将请求挂在接收队列中。

3.2.2.7 inet_csk_reqsk_queue_hash_add

```

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

```

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

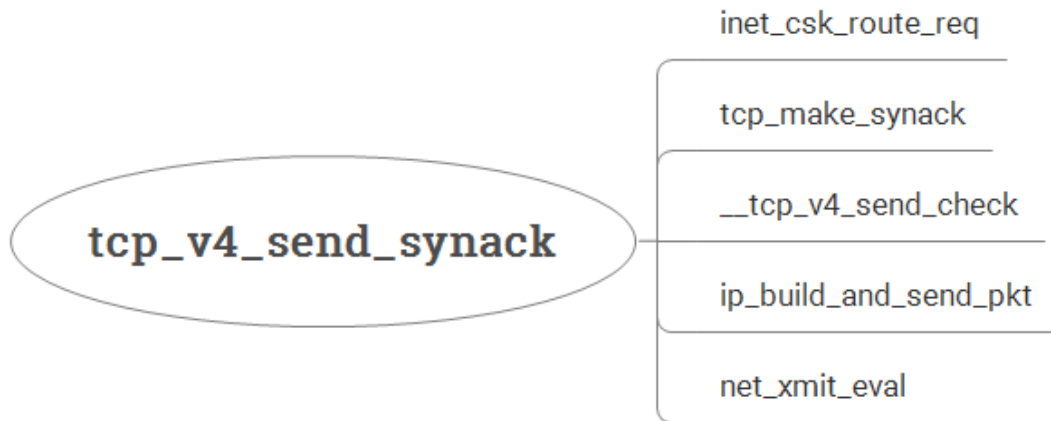
3.2.3 第二次握手：发送 SYN+ACK 段

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

3.2.3.1 第二次函数调用关系

第二次握手的调用函数关系图如下：

SYN+ACK.png SYN+ACK.bb



3.2.3.2 tcp_v4_send_synack

```

1  /*
2  Location:
3
4      net/ipv4/tcp_ipv4.c
5
6  Function:
7
8      Send a SYN-ACK after having received a SYN.
9      This still operates on a request_sock only, not on a big
10     socket.
11
12  Paramaters:
13
14     sk: 传输控制块
15     dst: 存储缓存路由项中独立于协议的信息。
16     fl:
17     req: 请求控制块
18     foc: 快打开缓存
19     attach_req:
20
21  */
22  static int tcp_v4_send_synack(const struct sock *sk, struct dst_entry *dst,
23                               struct flowi *fl,
24                               struct request_sock *req,
25                               struct tcp_fastopen_cookie *foc,
26                               bool attach_req)
27  {
28     const struct inet_request_sock *ireq = inet_rsk(req);
29     struct flowi4 fl4;
30     int err = -1;
31     struct sk_buff *skb;
32
33     /* First, grab a route. */
34     if (!dst && (dst = inet_csk_route_req(sk, &fl4, req)) == NULL)
35         return -1;
36
37     skb = tcp_make_synack(sk, dst, req, foc, attach_req);
38

```

```

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

```

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

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

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

`net_xmit_eval`是什么, 待考虑。

3.2.3.3 tcp_make_synack

函数如下:

```

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

```

```

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

```

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

```

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

```

为 MAC 层, IP 层, TCP 层首部预留必要的空间。

```

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

```

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

```

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

```

根据每一个路由器上的 mss 以及自身的 mss 来得到最大的 mss。

```

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

```

清除选项，并且设置相关时间戳 **to add in future**。

```

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

```

查看是否有 MD5 选项，有的话构造出相应的 md5。

```

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
7  rcu_read_unlock();
8  #endif
9
10 /* Do not fool tcpdump (if any), clean our debris */
11 skb->tstamp.tv64 = 0;
12 return skb;

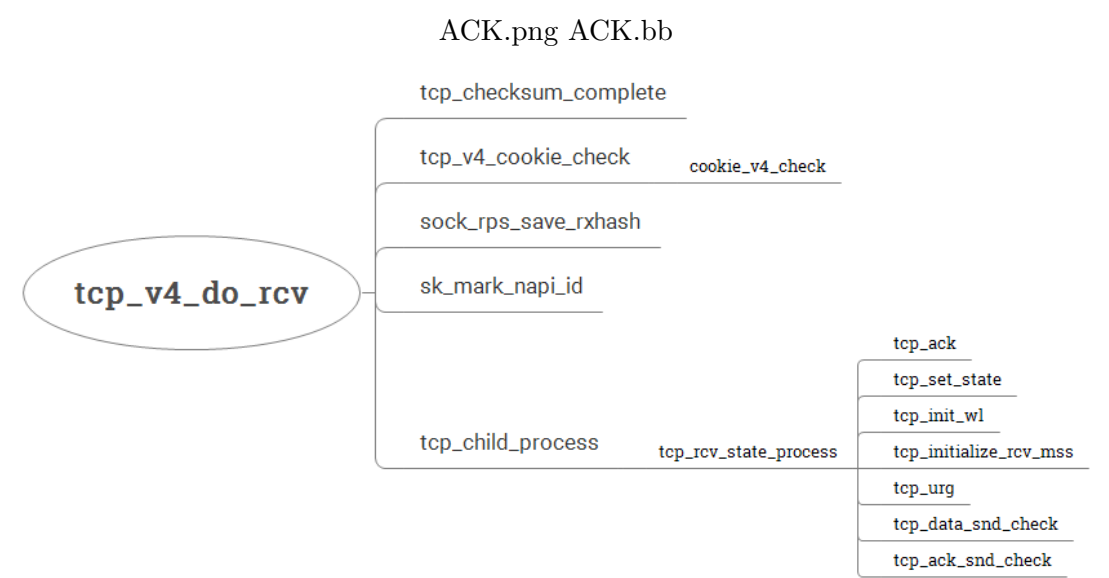
```

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

3.2.4 第三次握手：接收 ACK 段

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

3.2.4.1 第三次握手函数调用关系图



3.2.4.2 tcp_v4_do_rcv

```
1  /*
2  Location:
3
4      /net/ipv4/tcp_ipv4.c
5
6  Function:
7
8      The socket must have it's spinlock held when we get
9      here, unless it is a TCP_LISTEN socket.
10
11      We have a potential double-lock case here, so even when
12      doing backlog processing we use the BH locking scheme.
13      This is because we cannot sleep with the original spinlock
14      held.
15
16  Parameter:
17
18      sk: 传输控制块。
19      skb: 传输控制块缓冲区。
20  */
21  int tcp_v4_do_rcv(struct sock *sk, struct sk_buff *skb)
22  {
23      struct sock *rsk;
24
25      /** 省略无关代码 **/
26  }
```

```

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

```

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

3.2.4.3 tcp_v4_cookie_check

```

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

```

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

端的 ACK 序列号减去 1, 得到 cookie 比较值, 然后将客户端的 IP 地址, 客户端端口, 服务器 IP 地址和服务器端口, 接收到的 TCP 序列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  Location:
3
4      net/ipv4/tcp_minisock.c
5
6  Functions:
7
8      Queue segment on the new socket if the new socket is active,
9      otherwise we just shortcircuit this and continue with
10     the new socket.
11
12     For the vast majority of cases child->sk_state will be TCP_SYN_RECV
13     when entering. But other states are possible due to a race condition
14     where after __inet_lookup_established() fails but before the listener
15     locked is obtained, other packets cause the same connection to
16     be created.
17 Parameters:
18
19     parent: 父传出控制块
20     child: 子传输控制块
21     skb: 传输控制块缓存
22 */
23
24 int tcp_child_process(struct sock *parent, struct sock *child,
25                      struct sk_buff *skb)
26 {
27     int ret = 0;
28     int state = child->sk_state;
29
30     tcp_sk(child)->segs_in += max_t(u16, 1, skb_shinfo(skb)->gso_segs);
31     if (!sock_owned_by_user(child)) {
32         ret = tcp_rcv_state_process(child, skb);
33         /* Wakeup parent, send SIGIO */
34         if (state == TCP_SYN_RECV && child->sk_state != state)
35             parent->sk_data_ready(parent);
36     } else {
37         /* Alas, it is possible again, because we do lookup
38          * in main socket hash table and lock on listening
39          * socket does not protect us more.
40          */
41         __sk_add_backlog(child, skb);
42     }
43
44     bh_unlock_sock(child);
45     sock_put(child);
46     return ret;
47 }
```

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。

CHAPTER 4

TCP 输出

Contents	
4.1	tcp_sendmsg 72
4.2	tcp_transmit_skb 76
4.3	tcp_select_window(struct sk_buff *skb) 77
4.3.1	代码分析 77

4.1 tcp_sendmsg

使用 TCP 发送数据的大部分工作都是在tcp_sendmsg函数中实现的。

```
1  /* Location: net/ipv4/tcp.c
2  *
3  * Parameter:
4  *     sk 传输所使用的套接字
5  *     msg 要传输的用户层的数据包
6  *     size 用户要传输的数据的大小
7  */
8  int tcp_sendmsg(struct sock *sk, struct msghdr *msg, size_t size)
9  {
10     struct tcp_sock *tp = tcp_sk(sk);
11     struct sk_buff *skb;
12     int flags, err, copied = 0;
13     int mss_now = 0, size_goal, copied_syn = 0;
14     bool sg;
15     long timeo;
16
17     /* 对套接字加锁 */
18     lock_sock(sk);
19
20     flags = msg->msg_flags;
21     if (flags & MSG_FASTOPEN) {
22         err = tcp_sendmsg_fastopen(sk, msg, &copied_syn, size);
23         if (err == -EINPROGRESS && copied_syn > 0)
24             goto out;
```

```

25         else if (err)
26             goto out_err;
27     }

```

根据调用者传入的标志位，判断是否启用了快速开启 (Fast Open)。关于 Fast Open 的讨论，详见 1.3.8。如果启用了 Fast Open，则会调用 `tcp_sendmsg_fasopen` 函数进行相关处理。

```

1     timeo = sock_sndtimeo(sk, flags & MSG_DONTWAIT);
2
3     /* 等待连接完成。由于 TCP 是面向连接的，如果还没有建立好连接的话，是无法
4      * 发送任何东西的。不过，这里有一种例外情况，如果是处于 Fast Open 的被动端的话，
5      * 是可以在三次连接的过程中带上数据的。
6      */
7     if (((1 << sk->sk_state) & ~(TCPF_ESTABLISHED | TCPF_CLOSE_WAIT)) &&
8         !tcp_passive_fastopen(sk)) {
9         err = sk_stream_wait_connect(sk, &timeo);
10        if (err != 0)
11            goto do_error;
12    }
13
14    /* TCP repair 是 Linux3.5 引入的新补丁，它能够实现容器在不同的物理主机间迁移。
15     * 它能够在迁移之后，将 TCP 连接重新设置到之前的状态。
16     */
17    if (unlikely(tp->repair)) {
18        if (tp->repair_queue == TCP_RECV_QUEUE) {
19            copied = tcp_send_rcvq(sk, msg, size);
20            goto out_nopush;
21        }
22
23        err = -EINVAL;
24        if (tp->repair_queue == TCP_NO_QUEUE)
25            goto out_err;
26
27        /* 'common' sending to sendq */
28    }
29
30    /* This should be in poll */
31    sk_clear_bit(SOCKWQ_ASYNC_NOSPACE, sk);
32
33    /* 获取 MSS 大小。size_goal 是数据报到达网络设备时所允许的最大长度。
34     * 对于不支持分片的网卡，size_goal 是 MSS 的大小。否则，是 MSS 的整数倍。
35     */
36    mss_now = tcp_send_mss(sk, &size_goal, flags);
37
38    /* Ok commence sending.
39     * copied 是已经从用户数据块复制出来的字节数。
40     */
41    copied = 0;
42
43    err = -EPIPE;
44    if (sk->sk_err || (sk->sk_shutdown & SEND_SHUTDOWN))
45        goto out_err;
46
47    sg = !(sk->sk_route_caps & NETIF_F_SG);
48
49    /* 不断循环，将用户想要发送的东西全部发送出去。 */
50    while (msg_data_left(msg)) {

```



```

51      /* copy 代表本次需要从用户数据块中复制的数据量。 */
52      int copy = 0;
53      int max = size_goal;
54
55      /* 获得队尾的 SKB, 并判断 SKB 剩余的能携带的数据量。 */
56      skb = tcp_write_queue_tail(sk);
57      if (tcp_send_head(sk)) {
58          if (skb->ip_summed == CHECKSUM_NONE)
59              max = mss_now;
60          copy = max - skb->len;
61      }
62
63      /* 如果没法携带足够的數據了, 那么就重新分配一个 SKB。 */
64      if (copy <= 0) {
65          new_segment:
66              /* Allocate new segment. If the interface is SG,
67              * allocate skb fitting to single page.
68              */
69              if (!sk_stream_memory_free(sk))
70                  goto wait_for_sndbuf;
71
72              skb = sk_stream_alloc_skb(sk,
73                                      select_size(sk, sg),
74                                      sk->sk_allocation,
75                                      skb_queue_empty(&sk->sk_write_queue));
76              if (!skb)
77                  goto wait_for_memory;
78
79              /*
80              * Check whether we can use HW checksum.
81              */
82              if (sk->sk_route_caps & NETIF_F_ALL_CSUM)
83                  skb->ip_summed = CHECKSUM_PARTIAL;
84
85              /* 将新的 SKB 放到队尾, 并设定 copy 和 max 的值。 */
86              skb_entail(sk, skb);
87              copy = size_goal;
88              max = size_goal;
89
90              /* All packets are restored as if they have
91              * already been sent. skb_mstamp isn't set to
92              * avoid wrong rtt estimation.
93              */
94              if (tp->repair)
95                  TCP_SKB_CB(skb)->sacked |= TCPCB_REPAIRED;
96      }
97
98      /* Try to append data to the end of skb. */
99      if (copy > msg_data_left(msg))
100          copy = msg_data_left(msg);
101
102      /* 下面的部分在寻找哪里还有空间可以放数据 */
103      if (skb_availroom(skb) > 0) {
104          /* 在 SKB 头部还有一些空间, 重新计算 copy 的值, 以使用该空间。 */
105          copy = min_t(int, copy, skb_availroom(skb));
106          err = skb_add_data_nocache(sk, skb, &msg->msg_iter, copy);
107          if (err)
108              goto do_fault;
109      } else {

```

```

110         bool merge = true;
111         int i = skb_shinfo(skb)->nr_frags;
112         struct page_frag *pfrag = sk_page_frag(sk);
113
114         if (!sk_page_frag_refill(sk, pfrag))
115             goto wait_for_memory;
116
117         /* 判断能否在最后一个分片加数据。 */
118         if (!skb_can_coalesce(skb, i, pfrag->page,
119                                 pfrag->offset)) {
120             if (i == sysctl_max_skb_frags || !sg) {
121                 /* 无法设置分配, 那么就重新分配一个 SKB。 */
122                 tcp_mark_push(tp, skb);
123                 goto new_segment;
124             }
125             merge = false;
126         }
127
128         copy = min_t(int, copy, pfrag->size - pfrag->offset);
129
130         if (!sk_wmem_schedule(sk, copy))
131             goto wait_for_memory;
132
133         /* 将用户数据块复制到 SKB 中。 */
134         err = skb_copy_to_page_nocache(sk, &msg->msg_iter, skb,
135                                         pfrag->page,
136                                         pfrag->offset,
137                                         copy);
138
139         if (err)
140             goto do_error;
141
142         /* 更新 SKB。 */
143         if (merge) {
144             skb_frag_size_add(&skb_shinfo(skb)->frags[i - 1], copy);
145         } else {
146             skb_fill_page_desc(skb, i, pfrag->page,
147                                 pfrag->offset, copy);
148             get_page(pfrag->page);
149         }
150         pfrag->offset += copy;
151     }
152
153     if (!copied)
154         TCP_SKB_CB(skb)->tcp_flags &= ~TCPHDR_PSH;
155
156     /* 更新 TCP 的序号 */
157     tp->write_seq += copy;
158     TCP_SKB_CB(skb)->end_seq += copy;
159     tcp_skb_pcount_set(skb, 0);
160
161     copied += copy;
162     if (!msg_data_left(msg)) {
163         tcp_tx_timestamp(sk, skb);
164         goto out;
165     }
166
167     if (skb->len < max || (flags & MSG_OOB) || unlikely(tp->repair))
168         continue;

```

```

169         /* 检查该数据是否必须立即发送。 */
170         if (forced_push(tp)) {
171             /* 如果需要立即发送, 则调用相关函数将队列里的数据都发送出去 */
172             tcp_mark_push(tp, skb);
173             __tcp_push_pending_frames(sk, mss_now, TCP_NAGLE_PUSH);
174         } else if (skb == tcp_send_head(sk))
175             tcp_push_one(sk, mss_now);
176         continue;
177
178     wait_for_sndbuf:
179         set_bit(SOCK_NOSPACE, &sk->sk_socket->flags);
180         /* 设置当前的状态为无空间状态, 并等待内存空间。 */
181     wait_for_memory:
182         /* 如果已经复制了一定的数据了, 那么将数据先发出去。 */
183         if (copied)
184             tcp_push(sk, flags & ~MSG_MORE, mss_now,
185                     TCP_NAGLE_PUSH, size_goal);
186
187         err = sk_stream_wait_memory(sk, &timeo);
188         if (err != 0)
189             goto do_error;
190
191         mss_now = tcp_send_mss(sk, &size_goal, flags);
192     }

```

之后, 是一系列的错误处理部分。(事实上, 正常退出也要经过这里的out和 out_nopush阶段)

```

1     out:
2         /* 如果发生了超时或者要正常退出, 且已经拷贝了数据, 那么尝试将该数据发出 */
3         if (copied)
4             tcp_push(sk, flags, mss_now, tp->nonagle, size_goal);
5     out_nopush:
6         /* 将 sk 释放, 并返回已经发出的数据量。 */
7         release_sock(sk);
8         return copied + copied_syn;
9
10    do_fault:
11        /* 当拷贝数据发送异常是, 会进入这个分支。如果当前的 SKB 是新分配的,
12         * 那么, 将该 SKB 从发送队列中去除, 并释放该 SKB。
13         */
14        if (!skb->len) {
15            tcp_unlink_write_queue(skb, sk);
16            /* It is the one place in all of TCP, except connection
17             * reset, where we can be unlinking the send_head.
18             */
19            tcp_check_send_head(sk, skb);
20            sk_wmem_free_skb(sk, skb);
21        }
22
23    do_error:
24        /* 如果已经拷贝了数据, 那么, 就将其发出。 */
25        if (copied + copied_syn)
26            goto out;
27    out_err:
28        /* 获取并返回错误码, 释放锁。 */
29        err = sk_stream_error(sk, flags, err);
30        /* make sure we wake any epoll edge trigger waiter */

```

```

31     if (unlikely(skb_queue_len(&sk->sk_write_queue) == 0 && err == -EAGAIN))
32         sk->sk_write_space(sk);
33     release_sock(sk);
34     return err;
35 }

```

4.2 tcp_transmit_skb

```

1

```

4.3 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。

4.3.1 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->advmss);
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 }
```

5.1 Linux 内核网络数据接收流程概览

正如下图所示，Linux 内核的网络数据接收时分为以上 5 个层次的。每个层次都有自己功能，相互依赖，同时又独立成块。下面，我们来依次介绍这些层次。

应用层 对于应用层来说，提供多种接口来接收数据，包括 `read`, `recv`, `recvfrom`。

BSD Socket 层 这一层主要是在应用层进行进一步的处理。

INET Socket 层 当被 IP 层的函数调用的时候，如果发现用户进程因正在操作传输控制块而将其锁定就会将未处理的 TCP 段添加到后备队列中，而一旦用户进程解锁传输控制块，就会理解处理后备队列，将 TCP 段处理之后的直接添加到接收队列中。当被 BSD Socket 的函数调用的时候，此时说明用户此时需要数据，那么我们就利用 `tcp_recvmmsg` 来进行消息接收，当然，起初得判断队列是否为空，不为空才可以直接处理，为空就得进一步进行相关的处理。

IP 层 IP 层则主要是在系统调度的时候，根据数据接收队列中是否由数据进行进一步的检验，判断该包是不是发往该主机。然后进行数据包的重组 (这是因为 TCP 数据有可能因为太大而被分片)，然后调用 INET 层的函数进行进一步的处理。

硬件层 这一部分主要是网卡接收到数据之后，进行发出请求中断，然后调用相关函数进行处理，将接收到的数据放入到数据接收队列中。当然，这时，那些没有经过 FCS 检验过的数据包这时候已经被抛弃了。

而在本文的分析中，我们主要关注的时 TCP 层的实现，故而我们主要分析 INET Socket 层。

TCP 传输控制块主要有三个关于接收的队列，接收队列，`prequeue` 队列和后备队列。

当启用 `tcp_low_latency`(参见非核心部分讲解) 时, TCP 传输控制块在软中断中接收并处理 TCP 段, 然后将其插入到接收队列中, 等待用户进程从接收队列中获取 TCP 段后复制到用户空间中, 最终删除并释放。

而不启用 `tcp_low_latency` 这个选项的时候, 则可以提高 tcp 协议栈的吞吐量及反应速度。TCP 传输控制块在软中断中将 TCP 段添加到 `prequeue` 队列中, 然后立即处理 `prequeue` 队列中的段, 如果用户进程正在读取数据, 则可以直接复制数据到用户空间的缓存区中, 否则添加到接收队列中, 然后从软中断返回。在多数情况下, 有机会处理 `prequeue` 队列中的包, 但只有当用户进程在进行 `recv` 类系统调用之前, 才在软中断中复制数据到用户空间的缓存区。

当然, 在用户进程因操作传输控制块而将其锁定时, 无论是否启用 `tcp_low_latency`, 都会将未处理的 TCP 段添加到后备队列中, 一旦用户进程解锁传输控制块, 就会立即处理后备队列, 将 TCP 段处理之后添加到接收队列中。

5.2 自底向上调用与自顶向下调用

5.2.1 自底向上处理

5.2.1.1 tcp_v4_rcv

正如上面的流程图所展示的样子, 当 IP 层接收到报文, 或将多个分片组装成一个完整的 IP 数据报之后, 会调用传输层的接收函数, 传递给传输层处理。

```

1  /*
2  Location:
3
4      net/ipv4/tcp_ipv4.c
5
6  Function:
7
8      TCP 接收数据的入口。
9
10 Parameter:
11
12     skb: 从 IP 层传递过来的数据报。
13
14 */
15 int tcp_v4_rcv(struct sk_buff *skb)
16 {
17     const struct iphdr *iph;
18     const struct tcphdr *th;
19     struct sock *sk;
20     int ret;
21     struct net *net = dev_net(skb->dev);
22     /* 如果不是发往本机的就直接丢弃 */
23     if (skb->pkt_type != PACKET_HOST)
24         goto discard_it;
25
26     /* Count it even if it's bad */
27     TCP_INC_STATS_BH(net, TCP_MIB_INSEGS);
28     /*
29         如果 TCP 段在传输过程中被分片了, 则到达本地后会在 IP 层重新组装。
    
```

```

30         组装完成后，报文分片都存储在链表中。在此，需把存储在分片中的报文
31         复制到 SKB 的线性存储区域中。如果发生异常，则丢弃该报文。
32     */
33     if (!pskb_may_pull(skb, sizeof(struct tcphdr)))
34         goto discard_it;
35     /* 得到相关报文的头部 */
36     th = tcp_hdr(skb);
37     /*
38         如果 TCP 的首部长度小于不带数据的 TCP 的首部长度，则说明 TCP 数据异常。
39         统计相关信息后，丢弃。
40     */
41     if (th->doff < sizeof(struct tcphdr) / 4)
42         goto bad_packet;
43     /*
44         检测整个 TCP 段长度和 TCP 首部长度是否正常，如有异常，则丢弃。
45     */
46     if (!pskb_may_pull(skb, th->doff * 4))
47         goto discard_it;
48
49     /* An explanation is required here, I think.
50      * Packet length and doff are validated by header prediction,
51      * provided case of th->doff==0 is eliminated.
52      * So, we defer the checks. */
53     /*
54         验证 TCP 首部中的校验和，如校验和有误，则说明报文已损坏，统计相关信息后丢弃。
55     */
56     if (skb_checksum_init(skb, IPPROTO_TCP, inet_compute_pseudo))
57         goto csum_error;

```

接下来，根据 *TCP* 首部中的信息来设置 *TCP* 控制块中的值，因为 *TCP* 首部中的值都是网络字节序的，为了便于后续处理直接访问 *TCP* 首部字段，在此将这些值转换为本机字节序后存储在 *TCP* 的私有控制块中。

```

1         th = tcp_hdr(skb);
2         iph = ip_hdr(skb);
3         /* This is tricky : We move IPCB at its correct location into TCP_SKB_CB()
4          * barrier() makes sure compiler wont play fool ~Waliasing games.
5          */
6         memmove(&TCP_SKB_CB(skb)->header.h4, IPCB(skb),
7                 sizeof(struct inet_skb_parm));
8         barrier();
9
10        TCP_SKB_CB(skb)->seq = ntohl(th->seq);
11        TCP_SKB_CB(skb)->end_seq = (TCP_SKB_CB(skb)->seq + th->syn + th->fin +
12                                    skb->len - th->doff * 4);
13        TCP_SKB_CB(skb)->ack_seq = ntohl(th->ack_seq);
14        TCP_SKB_CB(skb)->tcp_flags = tcp_flag_byte(th);
15        TCP_SKB_CB(skb)->tcp_tw_isn = 0;
16        TCP_SKB_CB(skb)->ip_dsfield = ipv4_get_dsfield(iph);
17        TCP_SKB_CB(skb)->sacked = 0;

```

接下来调用 `__inet_lookup` 在 `ehash` 或 `bhash` 散列表中根据地址和端口来查找传输控制块。如果在 `ehash` 中找到，则表示已经经历了三次握手并且已建立了连接，可以进行正常的通信。如果在 `bhash` 中找到，则表示已经绑定已经绑定了端口，处于侦听状态。如果在两个散列表中都不查找不到，说明此时对应的传输控制块还没有创建，跳转到 `no_tcp_socket` 处处理。

```

1 lookup:
2     sk = __inet_lookup_skb(&tcp_hashinfo, skb, th->source, th->dest);
3     if (!sk)
4         goto no_tcp_socket;

```

接下来继续处理。

```

1 process:
2     //如果是 TIME_WAIT 状态, 就跳转到相应位置处理。
3     if (sk->sk_state == TCP_TIME_WAIT)
4         goto do_time_wait;
5     //NEW_SYN_RECV 状态, ???
6     if (sk->sk_state == TCP_NEW_SYN_RECV) {
7         struct request_sock *req = inet_reqsk(sk);
8         struct sock *nsk;
9
10        sk = req->rsk_listener;
11        if (unlikely(tcp_v4_inbound_md5_hash(sk, skb))) {
12            reqsk_put(req);
13            goto discard_it;
14        }
15        if (unlikely(sk->sk_state != TCP_LISTEN)) {
16            inet_csk_reqsk_queue_drop_and_put(sk, req);
17            goto lookup;
18        }
19        sock_hold(sk);
20        nsk = tcp_check_req(sk, skb, req, false);
21        if (!nsk) {
22            reqsk_put(req);
23            goto discard_and_relse;
24        }
25        if (nsk == sk) {
26            reqsk_put(req);
27        } else if (tcp_child_process(sk, nsk, skb)) {
28            tcp_v4_send_reset(nsk, skb);
29            goto discard_and_relse;
30        } else {
31            sock_put(sk);
32            return 0;
33        }
34    }
35    if (unlikely(iph->ttl < inet_sk(sk)->min_ttl)) {
36        NET_INC_STATS_BH(net, LINUX_MIB_TCPMINTTLDROP);
37        goto discard_and_relse;
38    }

```

继续处理。

```

1     //查找 IPsec 数据库, 如果查找失败, 进行相应处理.
2     if (!xfrm4_policy_check(sk, XFRM_POLICY_IN, skb))
3         goto discard_and_relse;
4     //md5 相关
5     if (tcp_v4_inbound_md5_hash(sk, skb))
6         goto discard_and_relse;
7
8     nf_reset(skb);
9     //过滤器
10    if (sk_filter(sk, skb))
11        goto discard_and_relse;

```

```

12      /*
13          设置 SKB 的 dev 为 NULL, 这是因为现在已经到了传输层, 此时 dev
14          已经不再具有意义。
15      */
16      skb->dev = NULL;
17      /*LISTEN 状态 */
18      if (sk->sk_state == TCP_LISTEN) {
19          ret = tcp_v4_do_rcv(sk, skb);
20          goto put_and_return;
21      }
22
23      sk_incoming_cpu_update(sk);
24      /*
25          在接收 TCP 段之前, 需要对传输控制块加锁, 以同步对传输控制块
26          接收队列的访问。
27      */
28      bh_lock_sock_nested(sk);
29      tcp_sk(sk)->segs_in += max_t(u16, 1, skb_shinfo(skb)->gso_segs);
30      ret = 0;
31      /* 进程此时没有访问传输控制块 */
32      if (!sock_owned_by_user(sk)) {
33          /*prequeue 可读取 */
34          if (!tcp_prequeue(sk, skb))
35              ret = tcp_v4_do_rcv(sk, skb);
36      } else if (unlikely(sk_add_backlog(sk, skb,
37                                      sk->sk_rcvbuf + sk->sk_sndbuf))) {
38          bh_unlock_sock(sk);
39          NET_INC_STATS_BH(net, LINUX_MIB_TCPBACKLOGDROP);
40          goto discard_and_relse;
41      }
42      //解锁。
43      bh_unlock_sock(sk);

```

```

1  put_and_return:
2      sock_put(sk);
3
4      return ret;
5
6      /*
7          处理没有创建传输控制块收到报文, 校验错误, 坏包的情况, 给对端发送 RST 报文。
8      */
9  no_tcp_socket:
10      if (!xfrm4_policy_check(NULL, XFRM_POLICY_IN, skb))
11          goto discard_it;
12
13      if (tcp_checksum_complete(skb)) {
14  csum_error:
15          TCP_INC_STATS_BH(net, TCP_MIB_CSUMERRORS);
16      }
17      bad_packet:
18          TCP_INC_STATS_BH(net, TCP_MIB_INERRS);
19      } else {
20          tcp_v4_send_reset(NULL, skb);
21      }
22
23      /*
24          丢弃数据包。
25      */
26  discard_it:
27      /* Discard frame. */
28      kfree_skb(skb);

```

```

26         return 0;
27
28     discard_and_relse:
29         sock_put(sk);
30         goto discard_it;
31     /* 处理 TIME_WAIT 状态 */
32     do_time_wait:
33         if (!xfrm4_policy_check(NULL, XFRM_POLICY_IN, skb)) {
34             inet_twsk_put(inet_twsk(sk));
35             goto discard_it;
36         }
37
38         if (tcp_checksum_complete(skb)) {
39             inet_twsk_put(inet_twsk(sk));
40             goto csum_error;
41         }
42         switch (tcp_timewait_state_process(inet_twsk(sk), skb, th)) {
43             case TCP_TW_SYN: {
44                 struct sock *sk2 = inet_lookup_listener(dev_net(skb->dev),
45                                                         &tcp_hashinfo,
46                                                         iph->saddr, th->source,
47                                                         iph->daddr, th->dest,
48                                                         inet_iif(skb));
49
49                 if (sk2) {
50                     inet_twsk_deschedule_put(inet_twsk(sk));
51                     sk = sk2;
52                     goto process;
53                 }
54                 /* Fall through to ACK */
55             }
56             case TCP_TW_ACK:
57                 tcp_v4_timewait_ack(sk, skb);
58                 break;
59             case TCP_TW_RST:
60                 goto no_tcp_socket;
61             case TCP_TW_SUCCESS:
62                 ;
63         }
64         goto discard_it;
65     }

```

5.2.1.2 tcp_prequeue

```

1     /*
2     Location:
3
4         net/ipv4/tcp_ipv4.c
5
6     Function:
7
8
9         Packet is added to VJ-style prequeue for processing in process
10        context(进程上下文), if a reader task is waiting. Apparently, this exciting
11        idea (VJ's mail "Re: query about TCP header on tcp-ip" of 07 Sep 93)
12        failed somewhere. Latency? Burstiness? Well, at least now we will
13        see, why it failed. 8)8)
14
15        Parameter:

```

```

16
17     sk: 传输控制块
18     skb: 缓存区
19
20 bool tcp_prequeue(struct sock *sk, struct sk_buff *skb)
21 {
22     struct tcp_sock *tp = tcp_sk(sk);
23     //如果开启了 tcp_low_latency 或者用户没有在读数据, 直接返回
24     if (sysctl_tcp_low_latency || !tp->ucopy.task)
25         return false;
26
27     /*
28      * 数据包长度不大于 tcp 头部的长度, 并且 prequeue 队列为空。
29      */
30     if (skb->len <= tcp_hdrlen(skb) &&
31         skb_queue_len(&tp->ucopy.prequeue) == 0)
32         return false;
33
34     /* Before escaping RCU protected region, we need to take care of skb
35      * dst. Prequeue is only enabled for established sockets.
36      * For such sockets, we might need the skb dst only to set sk->sk_rx_dst
37      * Instead of doing full sk_rx_dst validity here, let's perform
38      * an optimistic check.
39      */
40     if (likely(sk->sk_rx_dst))
41         skb_dst_drop(skb);
42     else
43         skb_dst_force_safe(skb);
44
45     /*
46      * 将接收到的段添加到 prequeue 队列中, 并更新 prequeue 队列消耗的内存。
47      */
48     __skb_queue_tail(&tp->ucopy.prequeue, skb);
49     tp->ucopy.memory += skb->truesize;
50
51     /*
52      * 如果 prequeue 消耗的内存超过接收缓存上限, 则
53      * 立刻处理 prequeue 队列上的段。
54      */
55     if (tp->ucopy.memory > sk->sk_rcvbuf) {
56         struct sk_buff *skb1;
57
58         BUG_ON(sock_owned_by_user(sk));
59
60         while ((skb1 = __skb_dequeue(&tp->ucopy.prequeue)) != NULL) {
61             sk_backlog_rcv(sk, skb1);
62             NET_INC_STATS_BH(sock_net(sk),
63                             LINUX_MIB_TCPPREQUEUEDROPPED);
64         }
65
66         tp->ucopy.memory = 0;
67     } else if (skb_queue_len(&tp->ucopy.prequeue) == 1) { //队列为 1
68         wake_up_interruptible_sync_poll(sk_sleep(sk),
69             POLLIN | POLLRDNORM | POLLRDBAND);
70         //不需要发送 ack, 则复位重新启动延迟确认定时器。
71         if (!inet_csk_ack_scheduled(sk))
72             inet_csk_reset_xmit_timer(sk, ICSK_TIME_DACK,
73                 (3 * tcp_rto_min(sk)) / 4,
74                 TCP_RTO_MAX);
75
76         ///??? problem??page 931
77     }
78
79     return true;
80

```

```
75 }
```

5.2.2 自顶向下处理

5.2.2.1 tcp_recvmsg

我们一步一步对该函数进行分析。

```
1  /*
2  Location:
3
4      net/ipv4/tcp.c
5
6  Function:
7
8      *      This routine copies from a sock struct into the user buffer.
9      *
10     *      Technical note: in 2.3 we work on _locked_ socket, so that
11     *      tricks with *seq access order and skb->users are not required.
12     *      Probably, code can be easily improved even more.
13
14  Parameter:
15
16     sk: 传输控制块。
17     msg:
18     len:
19     nonblock:
20     flags: 读取数据的标志。
21     addrlen:
22  */
23
24  int tcp_recvmsg(struct sock *sk, struct msghdr *msg, size_t len, int nonblock,
25                 int flags, int *addr_len)
26  {
27     struct tcp_sock *tp = tcp_sk(sk);
28     int copied = 0;
29     u32 peek_seq;
30     u32 *seq;
31     unsigned long used;
32     int err;
33     int target;          /* Read at least this many bytes */
34     long timeo;
35     struct task_struct *user_recv = NULL;
36     struct sk_buff *skb, *last;
37     u32 urg_hole = 0;
38     //如果只是为了接收来自套接字错误队列的错误，那就直接执行如下函数。
39     if (unlikely(flags & MSG_ERRQUEUE))
40         return inet_recv_error(sk, msg, len, addr_len);
41     //不是特别理解，， ???
42     if (sk_can_busy_loop(sk) && skb_queue_empty(&sk->sk_receive_queue) &&
43         (sk->sk_state == TCP_ESTABLISHED))
44         sk_busy_loop(sk, nonblock);
```

上述主要处理一些比较异常的情况，接下来就开始进行正常的处理了。

```

1      /*
2          在用户进程进行读取数据之前，必须对传输层进行加锁，这主要是为了在读的过程中，软中断
3          操作传输层，从而造成数据的不同步甚至更为严重的不可预料的结果。
4      */
5      lock_sock(sk);
6      //初始化错误码, Transport endpoint is not connected
7      err = -ENOTCONN;
8      /*
9          如果此时只是处于 LISTEN 状态，表明尚未建立连接，
10         此时不允许用户读取数据，只能返回。
11     */
12     if (sk->sk_state == TCP_LISTEN)
13         goto out;
14     /*
15         获取阻塞读取的超时时间，如果进行非阻塞读取，则超时时间为 0。
16     */
17     timeo = sock_rcvtimeo(sk, nonblock);
18
19     /* Urgent data needs to be handled specially.
20        如果是要读取带外数据，则需要跳转到 recv_urg 进行处理。
21    */
22     if (flags & MSG_OOB)
23         goto recv_urg;

```

继续处理。

```

1      //被修复了??? 啥意思??
2      if (unlikely(tp->repair)) {
3          // Operation not permitted
4          err = -EPERM;
5          //如果只是查看数据的话，就直接跳转到 out 处理
6          if (!(flags & MSG_PEEK))
7              goto out;
8          //????
9          if (tp->repair_queue == TCP_SEND_QUEUE)
10             goto recv_sndq;
11          //Invalid argument
12          err = -EINVAL
13          if (tp->repair_queue == TCP_NO_QUEUE)
14             goto out;
15
16          /* 'common' recv queue MSG_PEEK-ing */
17      }

```

接下来进行数据复制。在把数据从接收缓存复制到用户空间的过程中，会更新当前已复制为止，以及段序号。如果接收数据，则会更新 `copied_seq`，但是如果只是查看数据而并不是从系统缓冲区移走数据，那么不能更新 `copied_seq`。因此，数据复制到用户空间的过程中，区别接收数据还是查看数据是根据是否更新 `copied_seq`，所以这里时根据接收数据还是查看来获取要更新标记的地址，而后面的复制操作就完全不关心时接收还是查看。

最后一行调用相关函数根据是否设置了 `MSG_WAITALL` 来确定本次调用需要接收数据的长度。如果设置了 `MSG_WAITALL` 标志，则读取数据长度为用户调用时输入的参数 `len`。


```

1      seq = &tp->copied_seq;
2      if (flags & MSG_PEEK) {
3          peek_seq = tp->copied_seq;
4          seq = &peek_seq;
5      }
6
7      target = sock_rcvlowat(sk, flags & MSG_WAITALL, len);

```

接下来通过 `urg_data` 和 `urg_seq` 来检测当前是否读取到带外数据。如果在读带外数据，则终止本次正常数据的读取。否则，如果用户进程有信号待处理，则也终止本次的读取。

```

1      do {
2          u32 offset;
3
4          /* Are we at urgent data? Stop if we have read anything or have SIGURG pending. */
5          if (tp->urg_data && tp->urg_seq == *seq) {
6              if (copied)
7                  break;
8              if (signal_pending(current)) {
9                  copied = timeo ? sock_intr_errno(timeo) : -EAGAIN;
10                 break;
11             }
12         }

```

接下来首先获取一个缓存区。

```

1      /* Next get a buffer. */
2      //接下来获取下一个要读取的段。
3      last = skb_peek_tail(&sk->sk_receive_queue);
4      skb_queue_walk(&sk->sk_receive_queue, skb) {
5          last = skb;
6          /* Now that we have two receive queues this
7             * shouldn't happen.
8             如果接收队列中的段序号比较大，则说明也获取不到下一个待获取的段，
9             这样也只能接着处理 prequeue 或后备队列，实际上这种情况不应该发生。
10          */
11          if (WARN(before(*seq, TCP_SKB_CB(skb)->seq),
12                  "recvmmsg bug: copied %X seq %X rcvnxt %X fl %X\n",
13                  *seq, TCP_SKB_CB(skb)->seq, tp->rcv_nxt,
14                  flags))
15              break;
16          /*
17             到此，我们已经获取了下一个要读取的数据段，计算该段开始读取数据的偏移位置，
18             当然，该偏移值必须在该段的数据长度范围内才有效。
19          */
20          offset = *seq - TCP_SKB_CB(skb)->seq;
21          /*
22             由于 SYN 标志占用了一个序号，因此如果存在 SYN 标志，则需要调整
23             偏移。由于偏移 offset 为无符号整型，因此，不会出现负数的情况。
24          */
25          if (TCP_SKB_CB(skb)->tcp_flags & TCPHDR_SYN)
26              offset--;
27          /*
28             只有当偏移在该段的数据长度范围内，才说明待读的段才是有效的，因此，接下来
29             跳转到 found_ok_skb 标签处读取数据。
30          */
31          if (offset < skb->len)

```

```

32         goto found_ok_skb;
33     /*
34         如果接收到的段中有 FIN 标识, 则跳转到 found_fin_ok 处处理。
35     */
36     if (TCP_SKB_CB(skb)->tcp_flags & TCPHDR_FIN)
37         goto found_fin_ok;
38     WARN(!(flags & MSG_PEEK),
39         "recvmmsg bug 2: copied %X seq %X rcvnxt %X fl %X\n",
40         *seq, TCP_SKB_CB(skb)->seq, tp->rcv_nxt, flags);
41 }

```

只有在读取完数据后, 才能在后备队列不为空的情况下处理接收到后备队列中的 TCP 段。否则终止本次读取。

由于是因为用户进程对传输控制块进行的锁定, 将 TCP 段缓存到后备队列, 故而, 一旦用户进程释放传输控制块就应该立即处理后备队列。处理后备队列直接在 `release_sock` 中实现, 以确保在任何时候解锁传输控制块时能立即处理后备队列。

```

1     /* Well, if we have backlog, try to process it now yet. */
2
3     if (copied >= target && !sk->sk_backlog.tail)
4         break;

```

当接收队列中可以读取的段已经读完, 在处理 `prequeue` 或后备队列之前, 我们需要显检查是否会存在一些异常的情况, 如果存在这类情况, 就需要结束这次读取, 返回前当然还顺便检测后备队列是否存在数据, 如果有则还需要处理。

```

1     if (copied) {
2         /*
3             检测条件:
4             1. 有错误发生
5             2. TCP 处于 CLOSE 状态
6             3. shutdown 状态
7             4. 收到信号
8             5. 只是查看数据
9         */
10        if (sk->sk_err ||
11            sk->sk_state == TCP_CLOSE ||
12            (sk->sk_shutdown & RCV_SHUTDOWN) ||
13            !timeo ||
14            signal_pending(current))
15            break;
16    } else {
17        /* 检测 TCP 会话是否即将终结 */
18        if (sock_flag(sk, SOCK_DONE))
19            break;
20        /* 如果有错误, 返回错误码 */
21        if (sk->sk_err) {
22            copied = sock_error(sk);
23            break;
24        }
25        /* 如果是 shutdown, 返回 */
26        if (sk->sk_shutdown & RCV_SHUTDOWN)
27            break;
28        /*
29            如果 TCP 状态为 CLOSE, 而套接口不再终结状态, 则进程可能
30            在读取一个从未建立连接的套接口, 因此, 返回相应的错误码。

```

```

31      */
32      if (sk->sk_state == TCP_CLOSE) {
33          if (!sock_flag(sk, SOCK_DONE)) {
34              /* This occurs when user tries to read
35               * from never connected socket.
36               */
37              copied = -ENOTCONN;
38              break;
39          }
40          break;
41      }
42      /*
43       * 未读到数据，且是非阻塞读取，则返回错误码 Try again.
44       */
45      if (!timeo) {
46          copied = -EAGAIN;
47          break;
48      }
49      /* 检测是否收到数据，并返回相应的错误码 */
50      if (signal_pending(current)) {
51          copied = sock_intr_errno(timeo);
52          break;
53      }
54  }

```

进一步处理。

```

1      //检测是否有确认需要立即发送
2      tcp_cleanup_rbuf(sk, copied);
3      /*
4       * 在未启用慢速路径的情况下，都会到此检测是否
5       * 需要处理 prepare 队列。
6       */
7      if (!sysctl_tcp_low_latency && tp->ucopy.task == user_rcv) {
8          /* Install new reader
9           * 如果此次是第一次检测处理 prepare 队列的话，则需要设置正在读取的
10           * 进程标识符、缓存地址信息，这样当读取进程进入睡眠后， ESTABLISHED
11           * 状态的接收处理就可能把数据复制到用户空间。
12           */
13          if (!user_rcv && !(flags & (MSG_TRUNC | MSG_PEEK))) {
14              user_rcv = current;
15              tp->ucopy.task = user_rcv;
16              tp->ucopy.msg = msg;
17          }
18          //更新当前可以使用的用户缓存的大小。
19          tp->ucopy.len = len;
20          //进行报警信息处理。
21          WARN_ON(tp->copied_seq != tp->rcv_nxt &&
22                  !(flags & (MSG_PEEK | MSG_TRUNC)));
23
24          /* Ugly... If prequeue is not empty, we have to
25           * process it before releasing socket, otherwise
26           * order will be broken at second iteration.
27           * More elegant solution is required!!!
28           *
29           * Look: we have the following (pseudo)queues:
30           *
31           * 1. packets in flight
32           * 2. backlog

```

```

33         * 3. prequeue
34         * 4. receive_queue
35         *
36         * Each queue can be processed only if the next ones
37         * are empty. At this point we have empty receive_queue.
38         * But prequeue_can_ be not empty after 2nd iteration,
39         * when we jumped to start of loop because backlog
40         * processing added something to receive_queue.
41         * We cannot release_sock(), because backlog contains
42         * packets arrived _after_ prequeued ones.
43         *
44         * Shortly, algorithm is clear --- to process all
45         * the queues in order. We could make it more directly,
46         * requeueing packets from backlog to prequeue, if
47         * is not empty. It is more elegant, but eats cycles,
48         * unfortunately.
49         */
50     /*
51         如果 prequeue 队列不为空, 则跳转到 do_orequeue 标签处处理 prequeue 队列
52     */
53     if (!skb_queue_empty(&tp->ucopy.prequeue))
54         goto do_prequeue;
55
56     /* __ Set realtime policy in scheduler __ */
57 }

```

继续处理, 如果读取完数据, 则调用 `release_sock` 来解锁传输控制块, 主要用来处理后备队列, 完成后再调用 `lock_sock` 锁定传输控制块。在调用 `release_sock` 的时候, 进程有可能会出现休眠。

如果数据尚未读取, 且是阻塞读取, 则进入睡眠等待接收数据。这种情况下, `tcp_v4_do_rcv` 处理 TCP 段时可能会把数据直接复制到用户空间。

```

1         if (copied >= target) {
2             /* Do not sleep, just process backlog. */
3             release_sock(sk);
4             lock_sock(sk);
5         } else {
6             sk_wait_data(sk, &timeo, last);
7         }

```

```

1         if (user_rcv) {
2             int chunk;
3
4             /* __ Restore normal policy in scheduler __ */
5             /*
6                 更新剩余用户空间长度以及已经复制到用户空间的数据长度
7             */
8             chunk = len - tp->ucopy.len;
9             if (chunk != 0) {
10                 NET_ADD_STATS_USER(sock_net(sk), LINUX_MIB_TCPDIRECTCOPYFROMBACKLOG, c
11                 len -= chunk;
12                 copied += chunk;
13             }
14             /*
15                 如果接收到接收队列中的数据已经全部复制到用户进程空间,
16                 但 prequeue 队列不为空, 则需要继续处理 prequeue 队列, 并

```

```

17         更新剩余的用户空间的长度和已经复制到用户空间的数据长度。
18         */
19         if (tp->rcv_nxt == tp->copied_seq &&
20             !skb_queue_empty(&tp->ucopy.prequeue)) {
21     do_prequeue:
22             tcp_prequeue_process(sk);
23
24             chunk = len - tp->ucopy.len;
25             if (chunk != 0) {
26                 NET_ADD_STATS_USER(sock_net(sk), LINUX_MIB_TCPDIRECTCOPYFROMPR
27                     len -= chunk;
28                     copied += chunk;
29             }
30         }
31     }

```

处理完 prequeue 队列后,如果有更新 copied_seq,且只是查看数据,则需要更新 peek_seq。然后继续获取下一个段进行处理。

```

1         if ((flags & MSG_PEEK) &&
2             (peek_seq - copied - urg_hole != tp->copied_seq)) {
3             net_dbg_ratelimited("TCP(%s:%d): Application bug, race in MSG_PEEK\n",
4                                 current->comm,
5                                 task_pid_nr(current));
6             peek_seq = tp->copied_seq;
7         }
8         continue;

```

```

1     found_ok_skb:
2         /* Ok so how much can we use? */
3         /*
4             获取该可读取段的数据长度, 在前面的处理中
5             已由 TCP 序号得到本次读取数据在该段中的偏移。
6         */
7         used = skb->len - offset;
8         if (len < used)
9             used = len;
10
11         /* Do we have urgent data here?
12             如果该段中包含带外数据, 则获取带外数据在该段中的偏移。
13             如果偏移在该段可读的范围内, 则表示带外数据有效。
14             进而
15                 如果带外数据偏移为 0, 则说明目前需要的数据正是带外数据,
16                 且带外数据不允许放入到正常的数据流中, 即在普通的数据数据
17                 流中接受带外数据, 则需要调整读取正常数据流的一些参数, 如已
18                 读取数据的序号、正常数据的偏移等。最后, 如果可读数据经过调
19                 整之后为 0, 则说明没有数据可读, 跳过本次读数据过程到 skip_copy 处
20                 处理。
21                 如果带外数据偏移不为 0, 则需要调整本次读取的正常长度直到读到带外
22                 数据为止。
23         */
24         if (tp->urg_data) {
25             u32 urg_offset = tp->urg_seq - *seq;
26             if (urg_offset < used) {
27                 if (!urg_offset) {
28                     if (!sock_flag(sk, SOCK_URGINLINE)) {
29                         ++*seq;
30                         urg_hole++;

```

```

31         offset++;
32         used--;
33         if (!used)
34             goto skip_copy;
35     }
36 } else
37     used = urg_offset;
38 }
39 }

```

接下来处理读取数据的情况。

```

1         if (!(flags & MSG_TRUNC)) {
2             /*
3              * 调用 skb_copy_datagram_msg 来将数据复制到用户空间
4              * 并且根据返回的值来判断是否出现了错误。
5              */
6             err = skb_copy_datagram_msg(skb, offset, msg, used);
7             if (err) {
8                 /* Exception. Bailout! */
9                 if (!copied)
10                     copied = -EFAULT;
11                 break;
12             }
13         }
14         /*
15          * 调整读正常数据流的一些参数，如已读取数据的序号、已读取
16          * 数据的长度，剩余的可以使用的用户空间缓存大小。如果是截短，
17          * 则通过调整这些参数，多余的数据就默默被丢弃了。
18          */
19         *seq += used;
20         copied += used;
21         len -= used;
22         /*tcp_rcv_space_adjust 调整合理的 TCP 接收缓存的大小 */
23         tcp_rcv_space_adjust(sk);

```

```

1 skip_copy:
2     /*
3      * 如果已经完成了对带外数据的处理，则将带外数据标志清零，
4      * 设置首部预测标志，下一个接收到的段，就又可以由首部预测执行快速
5      * 路径还是慢速路径了。
6      */
7     if (tp->urg_data && after(tp->copied_seq, tp->urg_seq)) {
8         tp->urg_data = 0;
9         tcp_fast_path_check(sk);
10    }
11    /*
12     * 如果该段还有数据没有读取（如带外数据），则只能继续处理该段，而不能将
13     * 该段从接收队列中删除。
14     */
15    if (used + offset < skb->len)
16        continue;
17    /* 如果发现段中存在 FIN 标志，则跳转到 found_fin_ok 标签处处理 */
18    if (TCP_SKB_CB(skb)->tcp_flags & TCPHDR_FIN)
19        goto found_fin_ok;
20    /*
21     * 如果已经读完该段的全部数据，且不是查看数据，则可以将该段从接收队列中
22     * 删除，然后继续处理后续的段。

```

```

23      */
24      if (!(flags & MSG_PEEK))
25          sk_eat_skb(sk, skb);
26      continue;
27  /*
28      由于 FIN 标志占一个序号，因此当前读取的序号需要递增。
29      如果已经读完该段的全部数据并且不是查看数据，则可以将该段从
30      接收队列中删除。然后就可以退出了，无需处理后续的段了。
31  */
32  found_fin_ok:
33      /* Process the FIN. */
34      ++*seq;
35      if (!(flags & MSG_PEEK))
36          sk_eat_skb(sk, skb);
37      break;
38  } while (len > 0);

```

```

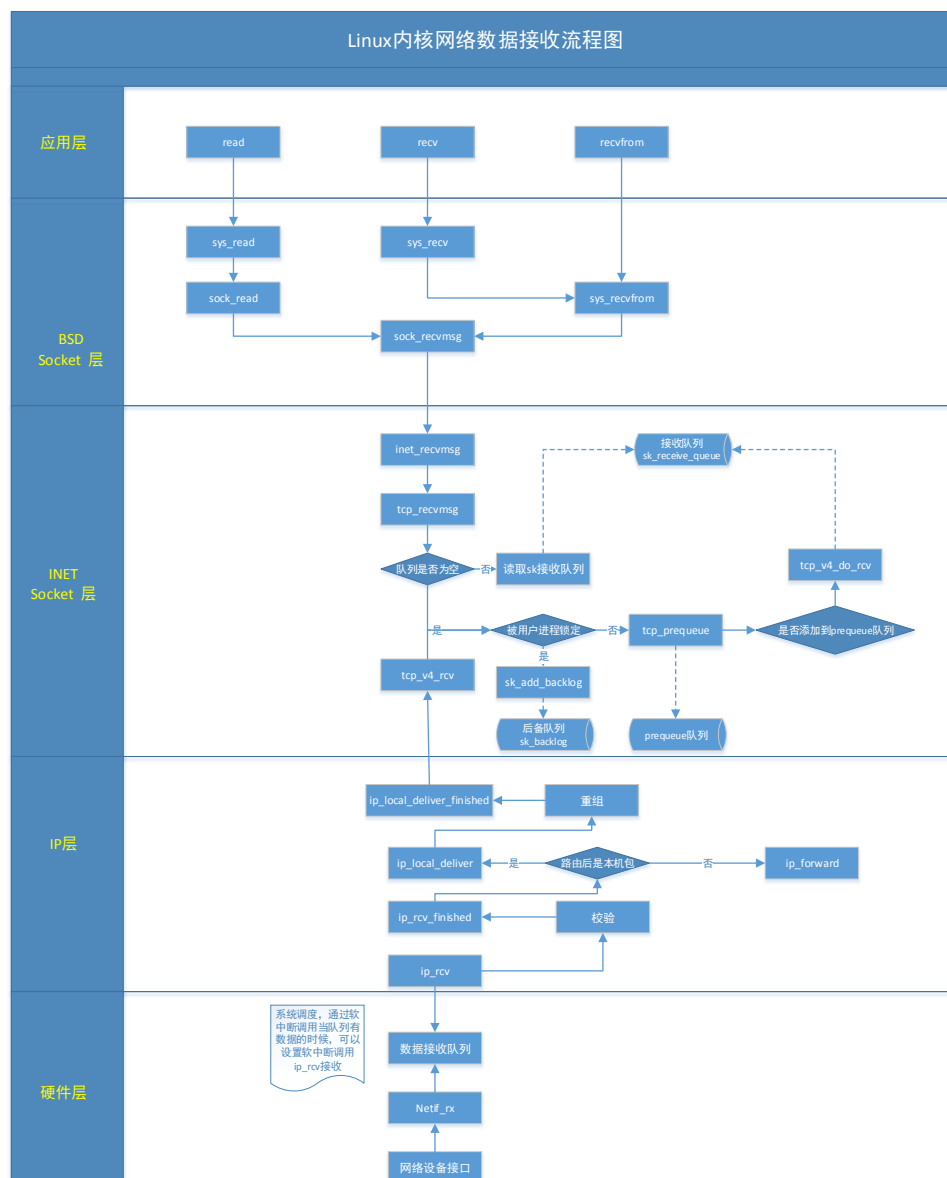
1      if (user_recv) {
2          //prequeue 队列非空
3          if (!skb_queue_empty(&tp->ucopy.prequeue)) {
4              int chunk;
5
6              tp->ucopy.len = copied > 0 ? len : 0;
7              //处理 prequeue 队列。
8              tcp_prequeue_process(sk);
9              /*
10                 如果在处理 prequeue 队列的过程中又有一部分数据复制到用户空间，则需要调整乘
11                 可用用户空间缓存大小和已读取数据的序号。
12             */
13             if (copied > 0 && (chunk = len - tp->ucopy.len) != 0) {
14                 NET_ADD_STATS_USER(sock_net(sk), LINUX_MIB_TCPDIRECTCOPYFROMPREQUEUE,
15                     len -= chunk;
16                     copied += chunk;
17             }
18         }
19         /*
20             最后清零 ucopy.task 和 ucopy.len，表示用户当前没有读取数据。这样当处理 prequeue
21             不会将数据复制到用户空间，因为只有未启用 tcp_low_latency 的情况下，用户进程主动
22             时，才有机会将数据直接复制到用户空间。
23         */
24         tp->ucopy.task = NULL;
25         tp->ucopy.len = 0;
26     }
27 }
28
29 在完成读取数据后，需再次检测是否有必要立即发送ACK段，并根据情况确定是否发送ACK段。在返回之前需解锁
30 返回已经读取的字节数。
31 \begin{minted}[linenos]{C}
32     /* According to UNIX98, msg_name/msg_namelen are ignored
33      * on connected socket. I was just happy when found this 8) --ANK
34      */
35
36     /* Clean up data we have read: This will do ACK frames. */
37     tcp_cleanup_rbuf(sk, copied);
38
39     release_sock(sk);
40     return copied;

```

接下来，进行一些其他情况的处理。

```
1  /*
2      如果在读取的过程中遇到 l 错误，就会跳转到此，解锁传输层然后返回错误码。
3  */
4  out:
5      release_sock(sk);
6      return err;
7  /*
8      如果是接收带外数据，则调用 tcp_recv_urg 接收。
9  */
10 recv_urg:
11     err = tcp_recv_urg(sk, msg, len, flags);
12     goto out;
13 /*
14 ?? 这一步是干什么的呢？
15 */
16 recv_sndq:
17     err = tcp_peek_sndq(sk, msg, len);
18     goto out;
19 }
```


Network Receive.pdf Network Receive.bb



Contents

6.1 拥塞控制实现	99
6.1.1 拥塞控制状态机	99
6.1.1.1 基本转换	99
6.1.1.2 Open 状态	99
6.1.1.3 Disorder 状态	99
6.1.1.4 CWR 状态	100
6.1.1.5 Recovery 状态	101
6.1.1.6 Loss 状态	101
6.1.2 显式拥塞通知 (ECN)	104
6.1.2.1 IP 对 ECN 的支持	104
6.1.2.2 TCP 对 ECN 的支持	105
6.1.3 拥塞控制状态的处理及转换	106
6.1.3.1 拥塞控制状态处理:tcp_fastretrans_alert	106
6.2 拥塞控制引擎	110
6.2.1 接口	110
6.2.1.1 tcp_congestion_ops	110
6.2.1.2 拥塞控制事件	111
6.2.1.3 tcp_register_congestion_control()	111
6.2.1.4 tcp_unregister_congestion_control()	113
6.2.2 CUBIC 拥塞控制算法	113
6.2.2.1 CUBIC 算法介绍	113
6.2.2.2 模块定义	114
6.2.2.3 基本参数及初始化	115
6.2.2.4 ssthresh 的计算	116

6.2.2.5	慢启动和拥塞避免	116
6.2.2.6	拥塞状态机状态变更	120
6.2.2.7	撤销 CWND 减小	120
6.2.2.8	处理 CWND 事件	120
6.2.2.9	收到 ACK	121
6.2.2.10	hystart	121

6.1 拥塞控制实现

6.1.1 拥塞控制状态机

6.1.1.1 基本转换

```

1  enum tcp_ca_state {
2      TCP_CA_Open = 0,
3  #define TCPF_CA_Open      (1<<TCP_CA_Open)
4      TCP_CA_Disorder = 1,
5  #define TCPF_CA_Disorder (1<<TCP_CA_Disorder)
6      TCP_CA_CWR = 2,
7  #define TCPF_CA_CWR      (1<<TCP_CA_CWR)
8      TCP_CA_Recovery = 3,
9  #define TCPF_CA_Recovery (1<<TCP_CA_Recovery)
10     TCP_CA_Loss = 4
11 #define TCPF_CA_Loss      (1<<TCP_CA_Loss)
12 };

```

6.1.1.2 Open 状态

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

6.1.1.3 Disorder 状态

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

6.1.1.4 CWR 状态

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

口减小) 状态。CWR 状态可以被 Recovery 状态或者 Loss 状态中断。进入拥塞窗口减小的函数如下:

```

1  /*
2  Location:
3
4      net/ipv4/tcp_input.c
5
6  Function:
7
8      Enter CWR state. Disable cwnd undo since congestion is proven with ECN
9
10 Parameter:
11
12     sk: 传输控制块
13
14 */
15 void tcp_enter_cwr(struct sock *sk)
16 {
17     struct tcp_sock *tp = tcp_sk(sk);
18     /* 进入 CWR 后就不需要窗口撤消了,
19     因此需要清除拥塞控制的慢启动阈值 */
20     tp->prior_ssthresh = 0;
21     /* 可以看出只有 OPen 状态和 Disorder 状态可以转移到该状态 */
22     if (inet_csk(sk)->icsk_ca_state < TCP_CA_CWR) {
23         /* 湖北人与 CWR 状态后不允许在进行拥塞窗口撤消了 */
24         tp->undo_marker = 0;
25         /* 进行相关的初始化 */
26         tcp_init_cwnd_reduction(sk);
27         /* 设置状态 */
28         tcp_set_ca_state(sk, TCP_CA_CWR);
29     }
30 }

```

对于函数tcp_init_cwnd_reduction的调用如下:

```

1  /*
2  Location:
3
4      net/ipv4/tcp_input.c
5
6  Function:
7
8      The cwnd reduction in CWR and Recovery uses the PRR algorithm in RFC 6937.
9      It computes the number of packets to send (sndcnt) based on packets newly
10 delivered:
11
12     1) If the packets in flight is larger than ssthresh, PRR spreads the
13        cwnd reductions across a full RTT.
14     2) Otherwise PRR uses packet conservation to send as much as delivered.
15        But when the retransmits are acked without further losses, PRR
16        slow starts cwnd up to ssthresh to speed up the recovery.
17
18 parameter:
19
20     sk: 传输控制块
21
22 */
23 static void tcp_init_cwnd_reduction(struct sock *sk)
24 {
25     struct tcp_sock *tp = tcp_sk(sk);
26
27 }

```

```

24      /* 记录发送拥塞时的 snd.nxt */
25      tp->high_seq = tp->snd_nxt;
26      /* snd_nxt at the time of TLP retransmit */
27      tp->tlp_high_seq = 0;
28      /* snd_cwnd_cnt 表示自从上次调整拥塞窗口到
29       目前为止接收到的总 ACK 段数，这里设置为 0 是因为
30       刚刚改变拥塞控制算法为 PRR。
31      */
32      tp->snd_cwnd_cnt = 0;
33      /* prior_cwnd means Congestion window at start of Recovery
34       设置该值为当前拥塞窗口。
35      */
36      tp->prior_cwnd = tp->snd_cwnd;
37      /* Number of newly delivered packets
38       to receiver in Recovery, 设置为 0
39      */
40      tp->prrr_delivered = 0;
41      /* Total number of pkts sent during Recovery */
42      tp->prrr_out = 0;
43      /* 根据给定的拥塞控制算法重新设置拥塞慢启动阈值 */
44      tp->snd_ssthresh = inet_csk(sk)->icsk_ca_ops->ssthresh(sk);
45      /* 设置 TCP_ECN_QUEUE_CWR 标志，标识由于收到显示拥塞通知而进入拥塞状态 */
46      tcp_ecn_queue_cwr(tp);
47  }

```

6.1.1.5 Recovery 状态

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

6.1.1.6 Loss 状态

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

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

调用 `tcp_enter_loss` 进入 Loss 状态，如下：

```

1  /*
2  Location:
3
4      net/ipv4/tcp_input.c
5
6  Function:
7      Enter Loss state. If we detect SACK reneging(违约), forget all SACK information
8      and reset tags completely, otherwise preserve SACKs. If receiver
9      dropped its ofo?? queue, we will know this due to reneging detection.
10
11  Parameter:
12
13      sk: 传输控制块
14
15  */
16  void tcp_enter_loss(struct sock *sk)
17  {
18      const struct inet_connection_sock *icsk = inet_csk(sk);
19      struct tcp_sock *tp = tcp_sk(sk);
20      struct sk_buff *skb;
21      bool new_recovery = icsk->icsk_ca_state < TCP_CA_Recovery;
22      bool is_reneg;          /* is receiver reneging on SACKs? */
23
24      /* Reduce ssthresh if it has not yet been made inside this window. */
25      if (icsk->icsk_ca_state <= TCP_CA_Disorder ||
26          !after(tp->high_seq, tp->snd_una) ||
27          (icsk->icsk_ca_state == TCP_CA_Loss && !icsk->icsk_retransmits)) {
28          /* 保留当前的阈值 */
29          tp->prior_ssthresh = tcp_current_ssthresh(sk);
30          /* 计算新的阈值 */
31          tp->snd_ssthresh = icsk->icsk_ca_ops->ssthresh(sk);
32          /* 发送 CA_EVENT_LOSS 拥塞事件给具体拥塞算法模块 */
33          tcp_ca_event(sk, CA_EVENT_LOSS);
34          /*
35              在下面的函数中会做以下两件事情:
36              tp->undo_marker = tp->snd_una;
37              /* Retransmission still in flight may cause DSACKs later.
38                  undo_retrans 在恢复拥塞控制之前可进行撤销的重传段数 ???
39                  retrans_out 重传并且还未得到确认的 TCP 段的数目
40              */
41              tp->undo_retrans = tp->retrans_out ? : -1;
42          */
43          tcp_init_undo(tp);
44      }
45      /* 拥塞窗口大小设置为 1 */
46      tp->snd_cwnd = 1;
47      /* snd_cwnd_cnt 表示自从上次调整拥塞窗口到
48         目前为止接收到的总 ACK 段数, 自然设置为 0 */
49      tp->snd_cwnd_cnt = 0;
50      /* 记录最近一次检验拥塞窗口的时间 */
51      tp->snd_cwnd_stamp = tcp_time_stamp;
52      /* 重传并且还未得到确认的 TCP 段的数目设置为零 */
53      tp->retrans_out = 0;
54      /* 丢失的包 */
55      tp->lost_out = 0;
56      /* 查看当前的 tp 里是否由 SACK 选项字段,
57         有的话, 返回 1, 没有的话返回 0
58         根据这一点来判断是否需要重置 tp 中选择确认的包 (sacked_out??? confused) 的个数为 0

```

```

59      */
60      if (tcp_is_reno(tp))
61          tcp_reset_reno_sack(tp);
62
63      skb = tcp_write_queue_head(sk);
64      /* 判断接受者是否认为 SACK 违约??? 怎么理解呢? */
65      is_reneg = skb && (TCP_SKB_CB(skb)->sacked & TCPCB_SACKED_ACKED);
66      if (is_reneg) {
67          NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPSACKRENEGING);
68          tp->sacked_out = 0;
69          tp->fackets_out = 0;
70      }
71      /* 清除有关重传的记忆变量 */
72      tcp_clear_all_retrans_hints(tp);
73
74      tcp_for_write_queue(skb, sk) {
75          if (skb == tcp_send_head(sk))
76              break;
77
78          TCP_SKB_CB(skb)->sacked &= (~TCPCB_TAGBITS) | TCPCB_SACKED_ACKED;
79          if (!(TCP_SKB_CB(skb)->sacked & TCPCB_SACKED_ACKED) || is_reneg) {
80              /* 清除 ACK 标志 */
81              TCP_SKB_CB(skb)->sacked &= ~TCPCB_SACKED_ACKED;
82              /* 添加 LOST 标志 */
83              TCP_SKB_CB(skb)->sacked |= TCPCB_LOST;
84              /* 统计丢失段的数量 */
85              tp->lost_out += tcp_skb_pcount(skb);
86              /* 重传的最大序号????? */
87              tp->retransmit_high = TCP_SKB_CB(skb)->end_seq;
88          }
89      }
90      /* 确认没有被确认的 TCP 段的数量 left_out */
91      tcp_verify_left_out(tp);
92
93      /* Timeout in disordered state after receiving substantial DUPACKs
94       * suggests that the degree of reordering is over-estimated.
95       */
96      if (icsk->icsk_ca_state <= TCP_CA_Disorder &&
97          tp->sacked_out >= sysctl_tcp_reordering)
98          /* 重新设置 reordering */
99          tp->reordering = min_t(unsigned int, tp->reordering,
100                                sysctl_tcp_reordering);
101      /* 设置拥塞状态 */
102      tcp_set_ca_state(sk, TCP_CA_Loss);
103      /* 记录发生拥塞时的 snd.nxt */
104      tp->high_seq = tp->snd_nxt;
105      /* 设置 ecn_flags, 表示发送方进入拥塞状态 */
106      tcp_ecn_queue_cwr(tp);
107
108      /* F-RTO RFC5682 sec 3.1 step 1: retransmit SND.UNA if no previous
109       * loss recovery is underway except recurring timeout(s) on
110       * the same SND.UNA (sec 3.2). Disable F-RTO on path MTU probing
111       */
112      tp->frto = sysctl_tcp_frto &&
113                (new_recovery || icsk->icsk_retransmits) &&
114                !inet_csk(sk)->icsk_mtup.probe_size;
115  }

```

6.1.2 显式拥塞通知 (ECN)

在处理网络中的拥塞的时候，有一种方法叫做显式拥塞控制。从名字上看就是，我们会直接收到关于拥塞的通知。至于是如何实现呢？我这里简单说一下原理，然后在详细说明。当 TCP 传递的时候，路由器使用 IP 首部的一对比特位来记录是否出现了拥塞。这样，当 TCP 段到达后，接收方知道报文段是否在某个位置经历过拥塞但是，需要注意的是，发送方才是真正需要了解是否发生了拥塞状况。因此，接收方使用下一个 ACK 通知发送方有拥塞发生。然后，发送方作出响应，缩小自己的拥塞窗口。

我们直到路由器是网络层的设备，所以说，如果想要路由器帮忙记录拥塞控制就必然需要 IP 的支持。当然，除此之外，也需要 TCP 层的支持。

下面是具体的叙述。

6.1.2.1 IP 对 ECN 的支持

IP 首部中的八位的服务类型域 (TOS) 原先在 RFC791 中定义为表明包的发送优先级、时延、吞吐量、可靠性和消耗等特征。在 RFC2474 中被重新定义为包含一个 6 位的区分服务码点 (DSCP) 和两个未用的位。DSCP 值表明一个在路由器上配置的和队列相关联的发送优先级。IP 对 ECN 的支持用到了 TOS 域的剩下的两位。

基本的定义如下：

```
1  enum {
2      INET_ECN_NOT_ECT = 0,           //TOS 后两位为 00：表示不支持 ECN
3      INET_ECN_ECT_1 = 1,           //TOS 后两位为 01：表示支持 ECN
4      INET_ECN_ECT_0 = 2,           //TOS 后两位为 10：表示支持 ECN?? 区别
5      INET_ECN_CE = 3,              //TOS 后两位为 11：表示在某路由器处出现拥塞
6      INET_ECN_MASK = 3,            //ECN 域的掩码
7  };
```

当路由器检测到拥塞的时候，当然在设置之前会检测之前是否出现了拥塞，如果没有就设置 ECN 域为 11。

```
1  /*
2  Location:
3
4      include/net/inet_ecn.h
5
6  Function:
7
8  Parameter:
9  *\
10 static inline int IP_ECN_set_ce(struct iphdr *iph)
11 {
12     u32 check = (__force u32)iph->check;           //force???, 为啥 32 位
13     u32 ecn = (iph->tos + 1) & INET_ECN_MASK;
14
15     /*
16      * After the last operation we have (in binary):
17      * INET_ECN_NOT_ECT => 01
18      * INET_ECN_ECT_1   => 10
19      * INET_ECN_ECT_0   => 11
20      * INET_ECN_CE      => 00
```



```

21      */
22      if (!(ecn & 2))                                //不支持 ECN 的返回 0。已经设置拥塞的不重复设置，返回。
23          return !ecn;
24
25      /*
26      * The following gives us:
27      * INET_ECN_ECT_1 => check += htons(0xFFFD)
28      * INET_ECN_ECT_0 => check += htons(0xFFFE)
29      */
30      check += (__force u16)htons(0xFFFB) + (__force u16)htons(ecn);
31
32      iph->check = (__force __sum16)(check + (check>=0xFFFF)); //重新计算校验码
33      iph->tos |= INET_ECN_CE; //把 ECN 域设置为 11，表示发生了拥塞 */
34      return 1;
35  }

```

这里计算校验码需要我们仔细分析以下，remain to do.????

6.1.2.2 TCP 对 ECN 的支持

路由器使用 IP 包的相关域来设置相关标志位来表示是否发生了拥塞。而主机使用 TCP 的首部来告知发送方，网络正在经历拥塞。

TCP 使用 6 位保留位 (Reserved) 的后两位来支持 ECN。两个新的标志 CWR、ECE 含义如下

CWR CWR 为发送端缩小拥塞窗口标志，用来通知接收端它已经收到了设置 ECE 标志的 ACK。

ECE ECE 有两个作用，在 TCP 三次握手时表明 TCP 端是否支持 ECN；在传输数据时表明接收到的 TCP 段的 IP 首部的 ECN 被设置为 11，即接收端发现了拥塞。

当两个支持 ECN 的 TCP 端进行 TCP 连接时，它们交换 SYN、SYN+ACK、ACK 段。对于支持 ECN 的 TCP 端来说，SYN 段的 ECE 和 CWR 标志都被设置了，SYN 的 ACK 只设置 ECE 标志。

相关宏定义如下：

```

1  /* 本端支持 ECN */
2  #define TCP_ECN_OK 1
3  /* 本端被通知了拥塞，此时作为发送方 */
4  #define TCP_ECN_QUEUE_CWR 2
5  /* 通知对端发生了拥塞，此时作为接收方 */
6  #define TCP_ECN_DEMAND_CWR 4
7  #define TCP_ECN_SEEN 8
8  #define TCPHDR_FIN 0x01
9  #define TCPHDR_SYN 0x02
10 #define TCPHDR_RST 0x04
11 #define TCPHDR_PSH 0x08
12 #define TCPHDR_ACK 0x10
13 #define TCPHDR_URG 0x20
14 #define TCPHDR_ECE 0x40
15 #define TCPHDR_CWR 0x80
16
17 #define TCPHDR_SYN_ECN (TCPHDR_SYN | TCPHDR_ECE | TCPHDR_CWR)

```

6.1.3 拥塞控制状态的处理及转换

`tcp_ack`用于处理 ACK 段, 通过处理 ACK, TCP 可以进行拥塞控制和流控制. 这里我们只是涉及和拥塞控制有关的代码.

```

1  /*
2  Location:
3
4      net/ipv4/tcp_input.c
5
6  Function:
7
8      处理 ack 段
9
10 Parameter
11     待写.
12
13 */
14 static int tcp_ack(struct sock *sk, const struct sk_buff *skb, int flag)
15     /* 省略无关代码 */
16     /*
17         tcp_ack_is_dubious 用于判断
18         1. ACK 是不是重复的
19         2. 是不是带有 SACK 选项并且有 ECE
20         3. 是不是 Open 态
21         只要有一项为真就会返回 1.
22     */
23     if (tcp_ack_is_dubious(sk, flag)) {
24         /*FLAG_NOT_DUP=FLAG_DATA/FLAG_WIN_UPDATE/FLAG_ACKED
25             =FLAG_DATA/FLAG_WIN_UPDATE/FLAG_DATA_ACKED/FLAG_SYN_ACKED
26             FLAG_SND_UNA_ADVANCED: Snd_una was changed (!= FLAG_DATA_ACKED)
27
28         */
29         is_dupack = !(flag & (FLAG_SND_UNA_ADVANCED | FLAG_NOT_DUP));
30         /* 进入拥塞控制处理 */
31         tcp_fastretrans_alert(sk, acked, prior_unsacked, is_dupack, flag);
32     }
33     /* 省略无关代码 */

```

6.1.3.1 拥塞控制状态处理:tcp_fastretrans_alert

```

1  /*
2  Location:
3
4      net/ipv4/tcp_input.c
5
6  Function:
7
8      Process an event, which can update packets-in-flight not trivially.
9      Main goal of this function is to calculate new estimate for left_out,
10     taking into account both packets sitting in receiver's buffer and
11     packets lost by network.
12
13     Besides that it does CWND reduction, when packet loss is detected
14     and changes state of machine.
15
16     It does _not_ decide what to send, it is made in function

```

```

17         tcp_xmit_retransmit_queue().
18
19     Parameter:
20
21         sk: 传输控制块.
22         acked:
23         prior_unsacked: 在拥塞控制状态转换之前, 从发送队列
24                         发出而未得到确认的 TCP 段的数目.
25         is_dupack: 是否有重复 ACK?? 需要好好分析分析.
26         flag: 标志位
27     */
28     static void tcp_fastretrans_alert(struct sock *sk, const int acked,
29                                     const int prior_unsacked,
30                                     bool is_dupack, int flag)
31     {
32         struct inet_connection_sock *icsk = inet_csk(sk);
33         struct tcp_sock *tp = tcp_sk(sk);
34         /**/
35         bool do_lost = is_dupack || ((flag & FLAG_DATA_SACKED) && (tcp_fackets_out(tp) > tp->reorderin
36         /*means what????*/
37         int fast_rexmit = 0;
38         /*WARN 的返回值是什么呢?
39         如果没有待确认的包以及并且选择确认的包的数量不为 0,
40         就置其为零。
41         */
42         if (WARN_ON(!tp->packets_out && tp->sacked_out))
43             tp->sacked_out = 0;
44         /*
45         如果选择确认的包的数量为零, 并且未确认与已选择确认之间存在段, 似乎存在矛盾?
46         to do.
47         */
48         if (WARN_ON(!tp->sacked_out && tp->fackets_out))
49             tp->fackets_out = 0;

```

接下来, 拥塞状态处理机启动。

```

1         /*      Now state machine starts.
2         A. ECE, hence prohibit(阻止) cwnd undoing, the reduction is required.
3         由于接收到显式拥塞通知, 因此禁止窗口撤销, 并且开始减小拥塞窗口。
4         */
5         if (flag & FLAG_ECE)
6             tp->prior_ssthresh = 0;

```

如果接收到的 ACK 指向已记录的 SACK, 这说明我们记录的 SACK 并没有反应接收方的真实状态。也就是说接收方现在已经处于严重的拥塞状态或者在处理上有 bug, 那么我们接下来就要按照重传超时的方式去处理。因为按照正常的逻辑流程, 接受的 ACK 不应该指向已记录的 SACK, 而应指向 SACK 并未包含的, 这说明接收方由于拥塞已经把 SACK 部分接收的段已经丢弃或者处理上有 BUG, 我们必须需要重传了。

```

1         /* B. In all the states check for reneging SACKs. */
2         if (tcp_check_sack_reneging(sk, flag))
3             return;

```

检查当前当前状态的一致性。???what dose if mean???

```

1      /* C. Check consistency of the current state. */
2      tcp_verify_left_out(tp);

```

检测从拥塞状态返回的条件，当 high_seq 被确认时，结束拥塞状态，进入 Open 状态。

```

1      /* D. Check state exit conditions. State can be terminated
2      *   when high_seq is ACKed. */
3      if (icsk->icsk_ca_state == TCP_CA_Open) {
4          //根据条件判断是否输出警告信息
5          WARN_ON(tp->retrans_out != 0);
6          //消除上一次的重传信息
7          tp->retrans_stamp = 0;
8      } else if (!before(tp->snd_una, tp->high_seq)) {
9          /*   进入的条件是 tp->snd_una >= tp->high_seq, 即
10             拥塞时, 记录的 SND.NXT 被确认了。拥塞情况已经好转很多了。
11             此时如果拥塞状态不处于 Open 状态, 可根据当前的状态结束回
12             到 Open 状态。
13             */
14          switch (icsk->icsk_ca_state) {
15              case TCP_CA_CWR:
16                  /* CWR is to be held something *above* high_seq
17                   * is ACKed for CWR bit to reach receiver. */
18                  if (tp->snd_una != tp->high_seq) {
19                      //窗口减小
20                      tcp_end_cwnd_reduction(sk);
21                      //回到 Open 状态??? CA 是什么意思。
22                      tcp_set_ca_state(sk, TCP_CA_Open);
23                  }
24                  break;
25
26              case TCP_CA_Recovery:
27                  //判断对方是否提供了 SACK 服务, 提供, 返回 0, 否则返回 1
28                  if (tcp_is_reno(tp))
29                      //设置 sacked_out 为 0
30                      tcp_reset_reno_sack(tp);
31                  /* 尝试从 Recovery 状态撤销
32                     成功, 就直接返回。
33                     */
34                  if (tcp_try_undo_recovery(sk))
35                      return;
36                  //结束拥塞窗口缩小
37                  tcp_end_cwnd_reduction(sk);
38                  break;
39              }
40      }
41
42      /* Use RACK (Retransmit Ack????) to detect loss
43         有时间好好分析。
44         */
45      if (sysctl_tcp_recovery & TCP_RACK_LOST_RETRANS &&
46          tcp_rack_mark_lost(sk))
47          flag |= FLAG_LOST_RETRANS; //说明重传的丢失
48
49      /* E. Process state. */
50      switch (icsk->icsk_ca_state) {
51          case TCP_CA_Recovery:
52              //判断是否没有段被确认

```

```

53         if (!(flag & FLAG_SND_UNA_ADVANCED)) {
54             //判断是否启用了 SACK 并且收到的是重复的 ACK,
55             if (tcp_is_reno(tp) && is_dupack)
56                 tcp_add_reno_sack(sk); //记录接收到的重复的 ACK 数量
57         } else {
58             if (tcp_try_undo_partial(sk, acked, prior_unsacked, flag))
59                 return;
60             /* Partial ACK arrived. Force fast retransmit. */
61             do_lost = tcp_is_reno(tp) ||
62                 tcp_fackets_out(tp) > tp->reordering;
63         }
64         if (tcp_try_undo_dsack(sk)) {
65             tcp_try_keep_open(sk);
66             return;
67         }
68         break;
69     case TCP_CA_Loss:
70         tcp_process_loss(sk, flag, is_dupack);
71         if (icsk->icsk_ca_state != TCP_CA_Open &&
72             !(flag & FLAG_LOST_RETRANS))
73             return;
74         /* Change state if cwnd is undone or retransmits are lost */
75     default:
76         //判断是否开启了 SACK, 没启用返回 1
77         if (tcp_is_reno(tp)) {
78             if (flag & FLAG_SND_UNA_ADVANCED)
79                 tcp_reset_reno_sack(tp); //重置 sacked
80             if (is_dupack)
81                 tcp_add_reno_sack(sk); //记录接收到的 sack
82         }
83
84         if (icsk->icsk_ca_state <= TCP_CA_Disorder)
85             tcp_try_undo_dsack(sk);
86         //确定能够离开 Disorder 状态, 而进入 Recovery 状态。
87         //如果不进入 Recovery 状态, 判断可否进入 OPEN 状态。
88         if (!tcp_time_to_recover(sk, flag)) {
89             tcp_try_to_open(sk, flag, prior_unsacked);
90             return;
91         }
92
93         /* MTU probe failure: don't reduce cwnd */
94         if (icsk->icsk_ca_state < TCP_CA_CWR &&
95             icsk->icsk_mtup.probe_size &&
96             tp->snd_una == tp->mtu_probe.probe_seq_start) {
97             tcp_mtup_probe_failed(sk);
98             /* Restores the reduction we did in tcp_mtup_probe() */
99             tp->snd_cwnd++;
100             tcp_simple_retransmit(sk);
101             return;
102         }
103
104         /* Otherwise enter Recovery state */
105         tcp_enter_recovery(sk, (flag & FLAG_ECE));
106         fast_rexmit = 1;
107     }
108     //如果接收到重复的 ACK 或者重传队首的段超时,
109     //则要为确定丢失的段更新记分牌
110     if (do_lost)

```

```

111         tcp_update_scoreboard(sk, fast_rexmit);
112         tcp_cwnd_reduction(sk, prior_unsacked, fast_rexmit, flag);
113         //重传重传队列中那些标记为 LOST 的段, 同时重置 RTO 定时器。
114         tcp_xmit_retransmit_queue(sk);
115     }

```

6.2 拥塞控制引擎

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

6.2.1 接口

6.2.1.1 tcp_congestion_ops

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

```

1  struct tcp_congestion_ops {
2      struct list_head    list;
3      u32 key; /* 算法名称的哈希值 */
4      u32 flags;
5
6      /* 初始化私有数据 (可选) */
7      void (*init)(struct sock *sk);
8      /* 释放私有数据 (可选) */
9      void (*release)(struct sock *sk);
10
11     /* 返回 ssthresh (必须实现) */
12     u32 (*sssthresh)(struct sock *sk);
13     /* 计算新的拥塞窗口 (必须实现) */
14     void (*cong_avoid)(struct sock *sk, u32 ack, u32 acked);
15     /* 在改变 ca_state 前会被调用 (可选) */
16     void (*set_state)(struct sock *sk, u8 new_state);
17     /* 处理拥塞窗口相关的事件 (可选) */
18     void (*cwnd_event)(struct sock *sk, enum tcp_ca_event ev);
19     /* 处理 ACK 包到达事件 (可选) */
20     void (*in_ack_event)(struct sock *sk, u32 flags);
21     /* 用于撤销“缩小拥塞窗口” (可选) */
22     u32 (*undo_cwnd)(struct sock *sk);
23     /* 有段被确认时会调用此函数 (可选) */
24     void (*pkts_acked)(struct sock *sk, u32 num_acked, s32 rtt_us);
25     /* 为 inet_diag 准备的获取信息的接口 (可选) */
26     size_t (*get_info)(struct sock *sk, u32 ext, int *attr,
27                        union tcp_cc_info *info);
28
29     /* 拥塞控制算法的名称 */
30     char    name[TCP_CA_NAME_MAX];
31     struct module *owner;
32 };

```

6.2.1.2 拥塞控制事件

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

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

```

1  enum tcp_ca_event {
2      /* 首次传输（无已发出但还未确认的包） */
3      CA_EVENT_TX_START,
4      /* 拥塞窗口重启 */
5      CA_EVENT_CWND_RESTART,
6      /* 拥塞恢复结束 */
7      CA_EVENT_COMPLETE_CWR,
8      /* loss 超时 */
9      CA_EVENT_LOSS,
10     /* ECT 被设置了，但 CE 没有被置位 */
11     CA_EVENT_ECN_NO_CE,
12     /* 收到了设置了 CE 位的 IP 报文 */
13     CA_EVENT_ECN_IS_CE,
14     /* 延迟确认已被发送 */
15     CA_EVENT_DELAYED_ACK,
16     CA_EVENT_NON_DELAYED_ACK,
17 };

```

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

```

1  enum tcp_ca_ack_event_flags {
2      CA_ACK_SLOWPATH      = (1 << 0),    /* 在慢速路径中处理 */
3      CA_ACK_WIN_UPDATE    = (1 << 1),    /* 该 ACK 更新了窗口大小 */
4      CA_ACK_ECE           = (1 << 2),    /* ECE 位被设置了 */
5  };

```

6.2.1.3 tcp_register_congestion_control()

该函数用于注册一个新的拥塞控制算法。

```

1  int tcp_register_congestion_control(struct tcp_congestion_ops *ca)
2  {
3      int ret = 0;
4
5      /* 所有的算法都必须实现 ssthresh 和 cong_avoid ops */
6      if (!ca->ssthresh || !ca->cong_avoid) {
7          pr_err("%s does not implement required ops\n", ca->name);
8          /* 如果没实现，则返回错误 */
9          return -EINVAL;
10     }
11
12     /* 计算算法名称的哈希值，加快比对速度。 */
13     ca->key = jhash(ca->name, sizeof(ca->name), strlen(ca->name));
14
15     spin_lock(&tcp_cong_list_lock);

```

```

16         if (ca->key == TCP_CA_UNSPEC || tcp_ca_find_key(ca->key)) {
17             /* 如果已经注册被注册过了, 或者恰巧 hash 值重了 (极低概率),
18              * 那么返回错误值。
19              */
20             pr_notice("%s already registered or non-unique key\n",
21                      ca->name);
22             ret = -EEXIST;
23         } else {
24             /* 将算法添加到链表中 */
25             list_add_tail_rcu(&ca->list, &tcp_cong_list);
26             pr_debug("%s registered\n", ca->name);
27         }
28         spin_unlock(&tcp_cong_list_lock);
29
30         return ret;
31     }

```

其中, `tcp_ca_find_key` 函数通过哈希值来查找名称。jash 是一种久经考验的性能极佳的哈希算法。据称, 其计算速度和产生的分布都很漂亮。这里计算哈希值正是使用了这种哈希算法。早些版本的内核查找拥塞控制算法, 是通过名字直接查找的。

```

1  /* Simple linear search, don't expect many entries! */
2  static struct tcp_congestion_ops *tcp_ca_find(const char *name)
3  {
4      struct tcp_congestion_ops *e;
5
6      list_for_each_entry_rcu(e, &tcp_cong_list, list) {
7          if (strcmp(e->name, name) == 0)
8              return e;
9      }
10
11     return NULL;
12 }

```

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

```

1  /* Simple linear search, not much in here. */
2  struct tcp_congestion_ops *tcp_ca_find_key(u32 key)
3  {
4      struct tcp_congestion_ops *e;
5
6      list_for_each_entry_rcu(e, &tcp_cong_list, list) {
7          if (e->key == key)
8              return e;
9      }
10
11     return NULL;
12 }

```

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

6.2.1.4 tcp_unregister_congestion_control()

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


```

1 void tcp_unregister_congestion_control(struct tcp_congestion_ops *ca)
2 {
3     spin_lock(&tcp_cong_list_lock);
4     /* 删除该拥塞控制算法 */
5     list_del_rcu(&ca->list);
6     spin_unlock(&tcp_cong_list_lock);
7
8     /* Wait for outstanding readers to complete before the
9      * module gets removed entirely.
10     */
11     /* A try_module_get() should fail by now as our module is
12     * in "going" state since no refs are held anymore and
13     * module_exit() handler being called.
14     */
15     synchronize_rcu();
16 }

```

6.2.2 CUBIC 拥塞控制算法

Cubic 算法是 Linux 中默认的拥塞控制算法。

6.2.2.1 CUBIC 算法介绍

CUBIC 算法的思路是这样的：当发生了一次丢包后，它将此时的窗口大小定义为 W_{max} 。之后，进行乘法减小。这里与标准 TCP 不同，乘法减小时不是直接减小一半，而是乘一个常系数 β 。快重传和快恢复的部分和标准 TCP 一致。当它进入到拥塞避免阶段以后，它根据 cubic 函数凹的部分进行窗口的增长，直至到达 W_{max} 为止。之后，它会根据 cubic 函数凸的部分继续增长。

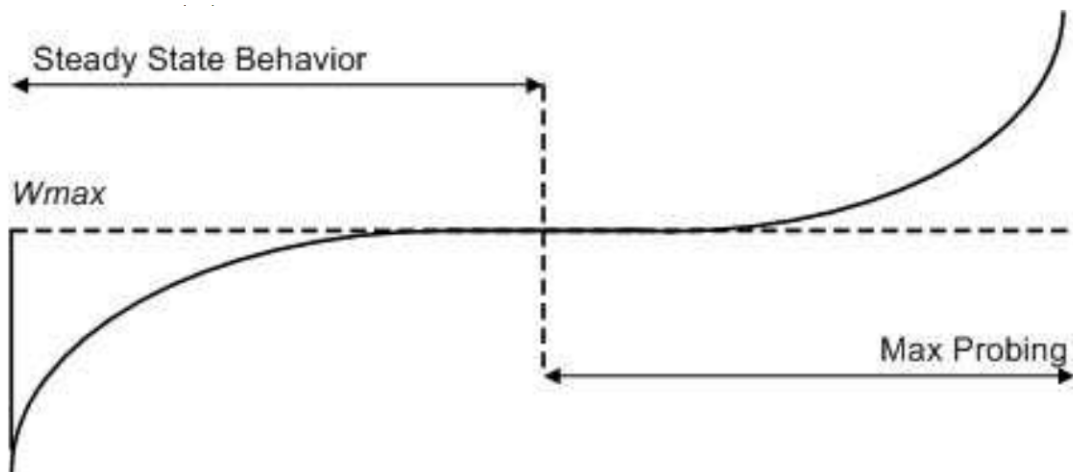


图 6.1: Cubic 函数

6.2.2.2 模块定义

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

```

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

```

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

```

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

```

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

```

1  static struct tcp_congestion_ops cubictcp __read_mostly = {
2      .init          = bictcp_init,
3      .ssthresh      = bictcp_recalc_ssthresh,
4      .cong_avoid     = bictcp_cong_avoid,
5      .set_state      = bictcp_state,
6      .undo_cwnd      = bictcp_undo_cwnd,
7      .cwnd_event     = bictcp_cwnd_event,
8      .pkts_acked     = bictcp_acked,
9      .owner          = THIS_MODULE,
10     .name           = "cubic",
11 };

```

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

```

1  static void __exit cubictcp_unregister(void)
2  {
3      tcp_unregister_congestion_control(&cubictcp);
4  }

```

6.2.2.3 基本参数及初始化

CUBIC 的所需的基本参数定义如下：

```

1  /* BIC TCP Parameters */
2  struct bictcp {
3      u32      cnt;          /* 每次 cwnd 增长 1/cnt 的比例 */
4      u32      last_max_cwnd; /* snd_cwnd 之前的最大值 */
5      u32      loss_cwnd;    /* 最近一次发生丢失的时候的拥塞窗口 */
6      u32      last_cwnd;    /* 最近的 snd_cwnd */
7      u32      last_time;    /* 更新 last_cwnd 的时间 */
8      u32      bic_origin_point; /* bic 函数的初始点 */
9      u32      bic_K;        /* 从当前一轮开始到初始点的时间 */
10     u32      delay_min;    /* 最小延迟 (msec << 3) */
11     u32      epoch_start;  /* 一轮的开始 */
12     u32      ack_cnt;      /* ack 的数量 */
13     u32      tcp_cwnd;     /* estimated tcp cwnd */
14     u16      unused;
15     u8      sample_cnt;    /* 用于决定 curr_rtt 的样本数 */
16     u8      found;         /* 是否找到了退出点? */
17     u32      round_start;  /* beginning of each round */
18     u32      end_seq;      /* end_seq of the round */
19     u32      last_ack;     /* last time when the ACK spacing is close */
20     u32      curr_rtt;     /* the minimum rtt of current round */
21 };

```

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

```

1  static void bictcp_init(struct sock *sk)
2  {
3      struct bictcp *ca = inet_csk_ca(sk);
4
5      bictcp_reset(ca);
6      ca->loss_cwnd = 0;
7
8      if (hystart)
9          bictcp_hystart_reset(sk);
10
11     if (!hystart && initial_ssthresh)
12         tcp_sk(sk)->snd_ssthresh = initial_ssthresh;
13 }

```

其中，`bictcp_reset`函数将必要的参数都初始化为 0 了。

```

1  static inline void bictcp_reset(struct bictcp *ca)
2  {
3      ca->cnt = 0;
4      ca->last_max_cwnd = 0;
5      ca->last_cwnd = 0;

```

```

6      ca->last_time = 0;
7      ca->bic_origin_point = 0;
8      ca->bic_K = 0;
9      ca->delay_min = 0;
10     ca->epoch_start = 0;
11     ca->ack_cnt = 0;
12     ca->tcp_cwnd = 0;
13     ca->found = 0;
14 }

```

6.2.2.4 ssthresh 的计算

门限值的计算过程在**bictcp_recalc_ssthresh**函数中实现。

```

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

```

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

6.2.2.5 慢启动和拥塞避免

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

```

1  static void bictcp_cong_avoid(struct sock *sk, u32 ack, u32 acked)
2  {
3      struct tcp_sock *tp = tcp_sk(sk);
4      struct bictcp *ca = inet_csk_ca(sk);
5
6      if (!tcp_is_cwnd_limited(sk))
7         return;
8
9      /* 当 tp->snd_cwnd < tp->snd_ssthresh 时,
10       * 让拥塞窗口大小正好等于 ssthresh 的大小。并据此计算 acked 的大小。
11       */

```

```

12     if (tcp_in_slow_start(tp)) {
13         if (hystart && after(ack, ca->end_seq))
14             bictcp_hystart_reset(sk);
15         acked = tcp_slow_start(tp, acked);
16         if (!acked)
17             return;
18     }
19     bictcp_update(ca, tp->snd_cwnd, acked);
20     tcp_cong_avoid_ai(tp, ca->cnt, acked);
21 }

```

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

```

1  u32 tcp_slow_start(struct tcp_sock *tp, u32 acked)
2  {
3      /* 新的拥塞窗口的大小等于 ssthresh 和 cwnd 中较小的那一个 */
4      u32 cwnd = min(tp->snd_cwnd + acked, tp->snd_ssthresh);
5
6      /* 如果新的拥塞窗口小于 ssthresh, 则 acked=0.
7       * 否则 acked 为超过 ssthresh 部分的数目。
8       */
9      acked -= cwnd - tp->snd_cwnd;
10     tp->snd_cwnd = min(cwnd, tp->snd_cwnd_clamp);
11
12     return acked;
13 }

```

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

```

1  static inline void bictcp_update(struct bictcp *ca, u32 cwnd, u32 acked)
2  {
3      u32 delta, bic_target, max_cnt;
4      u64 offs, t;
5
6      ca->ack_cnt += acked; /* 统计 ACKed packets 的数目 */
7
8      if (ca->last_cwnd == cwnd &&
9          (s32)(tcp_time_stamp - ca->last_time) <= HZ / 32)
10         return;
11
12     /* CUBIC 函数每个时间单位内最多更新一次 ca->cnt 的值。
13      * 每一次发生 cwnd 减小事件，ca->epoch_start 会被设置为 0.
14      * 这会强制重新计算 ca->cnt.
15      */
16     if (ca->epoch_start && tcp_time_stamp == ca->last_time)
17         goto tcp_friendlyness;
18
19     ca->last_cwnd = cwnd;
20     ca->last_time = tcp_time_stamp;
21
22     if (ca->epoch_start == 0) {
23         ca->epoch_start = tcp_time_stamp; /* 记录起始时间 */
24         ca->ack_cnt = acked; /* 开始计数 */
25         ca->tcp_cwnd = cwnd; /* 同步 cubic 的 cwnd 值 */
26     }

```

```

27         if (ca->last_max_cwnd <= cwnd) {
28             ca->bic_K = 0;
29             ca->bic_origin_point = cwnd;
30         } else {
31             /* 根据公式计算新的 K 值
32              *  $(w_{max} - cwnd) * (srtt \gg 3 / HZ) / c * 2^{(3 * bictcp\_HZ)}$ 
33              */
34             ca->bic_K = cubic_root(cube_factor
35                                   * (ca->last_max_cwnd - cwnd));
36             ca->bic_origin_point = ca->last_max_cwnd;
37         }
38     }
39
40     /* cubic function - calc */
41     /* calculate  $c * time^3 / rtt$ ,
42      * while considering overflow in calculation of  $time^3$ 
43      * (so  $time^3$  is done by using 64 bit)
44      * and without the support of division of 64bit numbers
45      * (so all divisions are done by using 32 bit)
46      * also NOTE the unit of those variables
47      *  $time = (t - K) / 2^{bictcp\_HZ}$ 
48      *  $c = bic\_scale \gg 10$ 
49      *  $rtt = (srtt \gg 3) / HZ$ 
50      * !!! The following code does not have overflow problems,
51      * if the cwnd < 1 million packets !!!
52      */
53
54     t = (s32)(tcp_time_stamp - ca->epoch_start);
55     t += msecs_to_jiffies(ca->delay_min >> 3);
56     /* change the unit from HZ to bictcp_HZ */
57     t <=& BICTCP_HZ;
58     do_div(t, HZ);
59
60     if (t < ca->bic_K) /* t - K */
61         offs = ca->bic_K - t;
62     else
63         offs = t - ca->bic_K;
64
65     /*  $c / rtt * (t - K)^3$  */
66     delta = (cube_rtt_scale * offs * offs * offs) >> (10 + 3 * BICTCP_HZ);
67     if (t < ca->bic_K) /* below origin */
68         bic_target = ca->bic_origin_point - delta;
69     else /* above origin */
70         bic_target = ca->bic_origin_point + delta;
71
72     /* 根据 cubic 函数计算出来的目标拥塞窗口值和当前拥塞窗口值, 计算 cnt 的大小。 */
73     if (bic_target > cwnd) {
74         ca->cnt = cwnd / (bic_target - cwnd);
75     } else {
76         ca->cnt = 100 * cwnd; /* 只增长一小点 */
77     }
78
79     /*
80      * The initial growth of cubic function may be too conservative
81      * when the available bandwidth is still unknown.
82      */
83     if (ca->last_max_cwnd == 0 && ca->cnt > 20)
84         ca->cnt = 20; /* increase cwnd 5% per RTT */
85

```

```

86 tcp_friendlyness:
87     /* TCP 友好性 */
88     if (tcp_friendlyness) {
89         u32 scale = beta_scale;
90
91         /* 推算在传统的 AIMD 算法下, TCP 拥塞窗口的大小 */
92         delta = (cwnd * scale) >> 3;
93         while (ca->ack_cnt > delta) {           /* update tcp cwnd */
94             ca->ack_cnt -= delta;
95             ca->tcp_cwnd++;
96         }
97
98         /* 如果 TCP 的算法快于 CUBIC, 那么就增长到 TCP 算法的水平 */
99         if (ca->tcp_cwnd > cwnd) {
100             delta = ca->tcp_cwnd - cwnd;
101             max_cnt = cwnd / delta;
102             if (ca->cnt > max_cnt)
103                 ca->cnt = max_cnt;
104         }
105     }
106
107     /* 控制增长速率不高于每个 rtt 增长为原来的 1.5 倍 */
108     ca->cnt = max(ca->cnt, 2U);
109 }

```

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

```

1 void tcp_cong_avoid_ai(struct tcp_sock *tp, u32 w, u32 acked)
2 {
3     /* 这里做了一个奇怪的小补丁, 用于解决这样一种情况:
4      * 如果 w 很大, 那么, snd_cwnd_cnt 可能会积累为一个很大的值。
5      * 此后, w 由于种种原因突然被缩小了很多。那么下面计算处理的 delta 就会很大。
6      * 这可能导致流量的爆发。为了避免这种情况, 这里提前增加了一个特判。
7      */
8     if (tp->snd_cwnd_cnt >= w) {
9         tp->snd_cwnd_cnt = 0;
10        tp->snd_cwnd++;
11    }
12
13    /* 累计被确认的包的数目 */
14    tp->snd_cwnd_cnt += acked;
15    if (tp->snd_cwnd_cnt >= w) {
16        /* 窗口增大的大小应当为被确认的包的数目除以当前窗口大小。
17         * 以往都是直接加一, 但直接加一并不是正确的加法增加 (AI) 的实现。
18         * 例如, w 为 10, acked 为 20 时, 应当增加 20/10=2, 而不是 1。
19         */
20        u32 delta = tp->snd_cwnd_cnt / w;
21
22        tp->snd_cwnd_cnt -= delta * w;
23        tp->snd_cwnd += delta;
24    }
25    tp->snd_cwnd = min(tp->snd_cwnd, tp->snd_cwnd_clamp);
26 }

```

6.2.2.6 拥塞状态机状态变更

当拥塞状态机的状态发生改变时，会调用`set_state`函数，对应到 CUBIC 模块中，就是`bictcp_state`函数。

```

1  static void bictcp_state(struct sock *sk, u8 new_state)
2  {
3      if (new_state == TCP_CA_Loss) {
4          bictcp_reset(inet_csk_ca(sk));
5          bictcp_hystart_reset(sk);
6      }
7  }
```

CUBIC 只特殊处理了一种状态：TCP_CA_Loss。可以看到，当进入了 LOSS 以后，就会调用`bictcp_reset`函数，重置拥塞控制参数。这样，拥塞控制算法就会重新从慢启动开始执行。

6.2.2.7 撤销 CWND 减小

CUBIC 通过返回当前拥塞窗口和上一次 LOSS 状态时的拥塞控制窗口中的最大值，来得到窗口缩小前的大小。

```

1  static u32 bictcp_undo_cwnd(struct sock *sk)
2  {
3      struct bictcp *ca = inet_csk_ca(sk);
4
5      return max(tcp_sk(sk)->snd_cwnd, ca->loss_cwnd);
6  }
```

6.2.2.8 处理 CWND 事件

如果目前没有任何数据包在传输了，那么需要重新设定`epoch_start`。这个是为了解决当应用程序在一段时间内不发送任何数据时，`now-epoch_start`会变得很大，由此，根据 Cubic 函数计算出来的目标拥塞窗口值也会变得很大。但显然，这是一个错误。因此，需要在应用程序重新开始发送数据时，重置`epoch_start`的值。在这里CA_EVENT_TX_START事件表明目前所有的包都被确认了（即没有任何正在传输的包），而应用程序又开始发送新的数据包了。所有的包都被确认说明应用程序有一段时间没有发包。因而，在程序又重新开始发包时，需要重新设定 `epoch_start`的值，以便在计算拥塞窗口的大小时，仍能合理地遵循 cubic 函数的曲线。

```

1  static void bictcp_cwnd_event(struct sock *sk, enum tcp_ca_event event)
2  {
3      if (event == CA_EVENT_TX_START) {
4          struct bictcp *ca = inet_csk_ca(sk);
5          u32 now = tcp_time_stamp;
6          s32 delta;
7
8          delta = now - tcp_sk(sk)->lsndtime;
9
10         /* We were application limited (idle) for a while.
11          * Shift epoch_start to keep cwnd growth to cubic curve.
12          */
```



```

13         if (ca->epoch_start && delta > 0) {
14             ca->epoch_start += delta;
15             if (after(ca->epoch_start, now))
16                 ca->epoch_start = now;
17         }
18         return;
19     }
20 }

```

6.2.2.9 收到 ACK

收到 ACK 后, Cubic 模块会重新计算链路的延迟情况。

```

1  /* Location: net/ipv4/tcp_cubic.c
2  *
3  * Track delayed acknowledgment ratio using sliding window
4  * ratio = (15*ratio + sample) / 16
5  */
6  static void bictcp_acked(struct sock *sk, u32 cnt, s32 rtt_us)
7  {
8      const struct tcp_sock *tp = tcp_sk(sk);
9      struct bictcp *ca = inet_csk_ca(sk);
10     u32 delay;
11
12     /* Some calls are for duplicates without timestamps */
13     if (rtt_us < 0)
14         return;
15
16     /* Discard delay samples right after fast recovery */
17     if (ca->epoch_start && (s32)(tcp_time_stamp - ca->epoch_start) < HZ)
18         return;
19
20     delay = (rtt_us << 3) / USEC_PER_MSEC;
21     if (delay == 0)
22         delay = 1;
23
24     /* 当第一次调用或者链路延迟增大时, 重设 delay_min 的值。 */
25     if (ca->delay_min == 0 || ca->delay_min > delay)
26         ca->delay_min = delay;
27
28     /* 当 cwnd 的大小大于阈值后, 会触发 hystart 更新机制 */
29     if (hystart && tcp_in_slow_start(tp) &&
30         tp->snd_cwnd >= hystart_low_window)
31         hystart_update(sk, delay);
32 }

```

6.2.2.10 hystart

在长期的实践中, 人们发现慢启动存在这样一个问题: 如果慢启动时窗口变得很大 (在大带宽网络中), 那么如果慢启动的过程中发生了丢包, 有可能一次丢掉大量的包 (因为每次窗口都会加倍)。HyStart (Hybrid Slow Start) 是一种优化过的慢启动算法, 可以避免传统慢启动过程中的突发性丢包, 从而提升系统的吞吐量并降低系统负载。

在 HyStart 算法中, 慢启动过程仍然是每次将拥塞窗口加倍。但是, 它会使用 ACK 空间和往返延迟作为启发信息, 来寻找一个安全的退出点 (Safe Exit Points)。退出点即

结束慢启动并进入拥塞避免的点。如果在慢启动的过程中发生丢包，那么 HyStart 算法的表现和传统的慢启动协议是一致的。

初始化 HyStart 算法时，会记录当前的时间，上一次的 rtt 值，并重新统计当前的 rtt 值。

```

1  /* Location: net/ipv4/tcp_cubic.c */
2  static inline void bictcp_hystart_reset(struct sock *sk)
3  {
4      struct tcp_sock *tp = tcp_sk(sk);
5      struct bictcp *ca = inet_csk_ca(sk);
6
7      ca->round_start = ca->last_ack = bictcp_clock();
8      ca->end_seq = tp->snd_nxt;
9      ca->curr_rtt = 0;
10     ca->sample_cnt = 0;
11 }

```

每次接到 ACK 以后，如果 tcp 仍处于慢启动状态，且拥塞窗口大小已经大于了一定的值，那么就会通过调用 hystart_update() 进入到 hystart 算法。

```

1  /* Location: net/ipv4/tcp_cubic.c */
2  static void hystart_update(struct sock *sk, u32 delay)
3  {
4      struct tcp_sock *tp = tcp_sk(sk);
5      struct bictcp *ca = inet_csk_ca(sk);
6
7      /* 如果已经找到了退出点，那么直接返回 */
8      if (ca->found & hystart_detect)
9          return;
10
11     if (hystart_detect & HYSTART_ACK_TRAIN) {
12         u32 now = bictcp_clock();
13
14         /* first detection parameter - ack-train detection */
15         if ((s32)(now - ca->last_ack) <= hystart_ack_delta) {
16             ca->last_ack = now;
17             if ((s32)(now - ca->round_start) > ca->delay_min >> 4) {
18                 ca->found |= HYSTART_ACK_TRAIN;
19                 NET_INC_STATS_BH(sock_net(sk),
20                                 LINUX_MIB_TCPHYSTARTTRAINDETECT);
21                 NET_ADD_STATS_BH(sock_net(sk),
22                                 LINUX_MIB_TCPHYSTARTTRAINCWND,
23                                 tp->snd_cwnd);
24                 tp->snd_ssthresh = tp->snd_cwnd;
25             }
26         }
27     }
28 }

```

上面是第一个判别条件，如果收到两次相隔不远的 ACK 之间的时间大于了 rtt 时间的一半，那么，就说明窗口的大小差不多到了网络容量的上限了。这里之所以会右移 4，是因为 delay_min 是最小延迟左移 3 后的结果。rtt 的一半相当于单程的时延。连续收到的两次 ACK 可以用于估计带宽大小。这里需要特别解释一下：由于在慢启动阶段，会一次性发送大量的包，所以，可以假设在网络中，数据是连续发送的。而收到的两个连续的包之间的时间间隔就是窗口大小除以网络带宽的商。作为发送端，我们只能获得两次 ACK 之间

的时间差, 因此用这个时间来大致估计带宽。拥塞窗口的合理大小为 $C = B \times D_{min} + S$, 这里 C 是窗口大小, B 是带宽, D_{min} 是最小的单程时延, S 是缓存大小。由于带宽是基本恒定的, 因此, 只要两次 ACK 之间的时间接近与 D_{min} 就可以认为该窗口的大小是基本合理的, 已经充分的利用了网络。

第二个启发条件是如果当前往返时延的增长已经超过了一定的限度, 那么, 说明网络的带宽已经要被占满了。因此, 也需要退出慢启动状态。

```

1      if (hystart_detect & HYSTART_DELAY) {
2          /* obtain the minimum delay of more than sampling packets */
3          if (ca->sample_cnt < HYSTART_MIN_SAMPLES) {
4              if (ca->curr_rtt == 0 || ca->curr_rtt > delay)
5                  ca->curr_rtt = delay;
6
7              ca->sample_cnt++;
8          } else {
9              if (ca->curr_rtt > ca->delay_min +
10                 HYSTART_DELAY_THRESH(ca->delay_min >> 3)) {
11                  ca->found |= HYSTART_DELAY;
12                  NET_INC_STATS_BH(sock_net(sk),
13                                   LINUX_MIB_TCPHYSTARTDELAYDETECT);
14                  NET_ADD_STATS_BH(sock_net(sk),
15                                   LINUX_MIB_TCPHYSTARTDELAYCWND,
16                                   tp->snd_cwnd);
17                  tp->snd_ssthresh = tp->snd_cwnd;
18              }
19          }
20      }
21  }
```

退出慢启动状态的方法很简单, 都是直接将当前的 `snd_ssthresh` 的大小设定为和当前拥塞窗口一样的大小。这样, TCP 自然就会转入拥塞避免状态。

Contents	
7.1 主动关闭	124
7.1.1 第一次握手——发送 FIN	124
7.1.2 第二次握手——接受 ACK	126
7.1.3 第三次握手——接受 FIN	130
7.1.4 第四次握手——发送 ACK	131
7.1.5 同时关闭	133
7.1.6 TIME_WAIT	134
7.2 被动关闭	136
7.2.1 基本流程	136
7.2.2 第一次握手：接收 FIN	136
7.2.2.1 函数调用关系	137
7.2.2.2 tcp_fin	137

7.1 主动关闭

7.1.1 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  /*
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         /* 如果此时已经发送一个 FIN 了, 就跳过。 */
26         if ((1 << sk->sk_state) &
27             (TCPF_ESTABLISHED | TCPF_SYN_SENT |
28              TCPF_SYN_RECV | TCPF_CLOSE_WAIT)) {
29             /* Clear out any half completed packets. FIN if needed. */
30             if (tcp_close_state(sk))
31                 tcp_send_fin(sk);
32         }
33     }

```

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

```

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

```

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

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

```

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

```

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

7.1.2 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 7.1.5 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_tws((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_TCPTEMEWAITOVERFLOW);
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 }

```

7.1.3 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  }

```

7.1.4 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_process进行处理,根据其返回值,执行相关操作。

```

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

```

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

```

1  static void tcp_v4_timewait_ack(struct sock *sk, struct sk_buff *skb)
2  {
3      struct inet_timewait_sock *tw = inet_twsk(sk);
4      struct tcp_timewait_sock *tcptw = tcp_twsk(sk);
5
6      /* 发送 ACK 包 */
7      tcp_v4_send_ack(sock_net(sk), skb,
8                      tcptw->tw_snd_nxt, tcptw->tw_rcv_nxt,
9                      tcptw->tw_rcv_wnd >> tw->tw_rcv_wscale,
10                     tcp_time_stamp + tcptw->tw_ts_offset,
11                     tcptw->tw_ts_recent,
12                     tw->tw_bound_dev_if,
13                     tcp_twsk_md5_key(tcptw),
14                     tw->tw_transparent ? IP_REPLY_ARG_NOSRCHECK : 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(tsval);
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。

7.1.5 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 7.2.2.2 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。

7.1.6 TIME_WAIT

该状态的处理也在tcp_timewait_state_process函数中。紧接在7.1.3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  }

```


7.2 被动关闭

7.2.1 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。

7.2.2 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。

7.2.2.1 函数调用关系

7.2.2.2 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 8

非核心函数分析

Contents

8.1	BSD Socket 层	140
8.1.1	msg_flag	140
8.1.2	数据报类型	141
8.1.3	Sock CheckSum	142
8.1.3.1	Checksum 相关标志位	142
8.1.3.2	skb_csum_unnecessary	143
8.1.3.3	__skb_checksum_complete	144
8.1.4	SK Stream	144
8.1.5	sk_stream_wait_connect	144
8.2	Inet	145
8.2.1	inet_hash_connect && __inet_hash_connect	145
8.3	Inet	145
8.3.1	inet_hash_connect && __inet_hash_connect	145
8.3.1.1	inet_hash_connect	145
8.3.1.2	__inet_hash_connect	145
8.3.2	inet_twsk_put	147
8.4	TCP 层	148
8.4.1	TCP 相关参数	148
8.4.2	TCP 相关宏定义	152
8.4.2.1	标志宏	152
8.4.2.2	函数宏	153
8.4.3	TCP CheckSum	153
8.4.3.1	tcp_checksum_complete	153
8.4.3.2	tcp_v4_checksum_init	154

8.4.4	<code>__tcp_push_pending_frames</code>	154
8.4.5	About ACK	154
8.4.5.1	<code>tcp_is_sack & tcp_is_reno & tcp_is_fack</code>	154
8.4.5.2	<code>tcp_reset_reno_sack</code>	155
8.4.5.3	<code>tcp_ack_is_dubious</code>	155
8.4.5.4	<code>tcp_check_sack_reneging</code>	156
8.4.5.5	<code>tcp_limit_reno_sacked</code>	157
8.4.5.6	<code>tcp_check_reno_reordering</code>	157
8.4.5.7	<code>tcp_add_reno_sack</code>	158
8.4.6	About Window Size and Segment Sent	158
8.4.6.1	<code>tcp_left_out</code>	158
8.4.6.2	<code>tcp_verify_left_out</code>	159
8.4.7	<code>tcp_fin_time</code>	159
8.4.8	About Congestion Control	159
8.4.8.1	<code>tcp_end_cwnd_reduction</code>	159
8.4.8.2	<code>tcp_try_undo_recovery</code>	160
8.4.9	About Congestion Control	161
8.4.9.1	<code>sysctl_tcp_recovery</code>	161
8.4.9.2	<code>tcp_rack_mark_lost</code>	161
8.4.9.3	163
8.4.9.4	<code>tcp_process_loss</code>	163
8.4.9.5	<code>tcp_try_undo_partial</code>	164
8.4.10	About Retransmit	165
8.4.10.1	Related Macro	165
8.4.11	<code>tcp_done</code>	165
8.4.12	<code>tcp_init_nondata_skb</code>	166
8.4.13	<code>before()</code> 和 <code>after()</code>	167
8.4.14	<code>tcp_shutdown</code>	167
8.4.15	<code>tcp_close</code>	168

8.1 BSD Socket 层

8.1.1 `msg_flag`

`MSG_OOB` 接收或发送带外数据。

`MSG_PEEK` 查看数据，并不从系统缓存区移走数据。

`MSG_DONTROUTE` 无需路由查找，目的地位于本地子网。

MSG_CTRUNC 指明由于缓存区空间不足，一些控制数据已被丢弃。

MSG_PROBE 使用这个标识，实际上并不会进行真正的的数据传递，而是进行路径 MTU 的探测。

MSG_TRUNC 只返回包的真实长度，并截短 (丢弃) 返回长度的数据。

MSG_DONTWAIT 无阻塞接收或发送。如果接收缓存中有数据，则接收数据并立刻返回。没有数据也立刻返回，而不进行任何等待。

MSG_WAITALL 必须一直等待，直到接收到的数据填满用户空间的缓存区。

MSG_CONFIRM 标识网关有效。只用于 SOCK_DGRAM 和 SOCK_RAW 类型的套接口。

MSG_ERRQUEUE 指示除了来自套接字错误队列的错误外，不接受其他数据。

MSG_NOSIGNAL 当另一端终止连接时，请求在基于流的错误套接字上不要发送 SIGPIPE 信号。

MSG_MORE 后续还有数据待发送。

MSG_CMSG_COMPAT 64 位兼容 32 位处理方式。

8.1.2 数据报类型

```

1  /*
2  Location:
3
4      include/uapi/linux/if_packet.h
5
6  Description:
7
8      Packet types
9  */
10 #define PACKET_HOST            0                /* To us */
11 #define PACKET_BROADCAST      1                /* To all */
12 #define PACKET_MULTICAST      2                /* To group */
13 #define PACKET_OTHERHOST      3                /* To someone else */
14 #define PACKET_OUTGOING        4                /* Outgoing of any type, 本机发出的包 */
15 #define PACKET_LOOPBACK        5                /* MC/BRD frame looped back */
16 #define PACKET_USER            6                /* To user space */
17 #define PACKET_KERNEL          7                /* To kernel space */
18 /* Unused, PACKET_FASTROUTE and PACKET_LOOPBACK are invisible to user space */
19 #define PACKET_FASTROUTE      6                /* Fastrouted frame */
20
21 /* Packet socket options */
22
23 #define PACKET_ADD_MEMBERSHIP    1
24 #define PACKET_DROP_MEMBERSHIP  2
25 #define PACKET_RECV_OUTPUT      3
26 /* Value 4 is still used by obsolete turbo-packet. */
27 #define PACKET_RX_RING          5
28 #define PACKET_STATISTICS        6
29 #define PACKET_COPY_THRESH      7
30 #define PACKET_AUXDATA          8
31 #define PACKET_ORIGDEV          9

```

```

32 #define PACKET_VERSION          10
33 #define PACKET_HDRLEN          11
34 #define PACKET_RESERVE         12
35 #define PACKET_TX_RING         13
36 #define PACKET_LOSS            14
37 #define PACKET_VNET_HDR        15
38 #define PACKET_TX_TIMESTAMP    16
39 #define PACKET_TIMESTAMP       17
40 #define PACKET_FANOUT          18
41 #define PACKET_TX_HAS_OFF      19
42 #define PACKET_QDISC_BYPASS    20
43 #define PACKET_ROLLOVER_STATS  21
44 #define PACKET_FANOUT_DATA     22
45
46 #define PACKET_FANOUT_HASH      0
47 #define PACKET_FANOUT_LB       1
48 #define PACKET_FANOUT_CPU      2
49 #define PACKET_FANOUT_ROLLOVER 3
50 #define PACKET_FANOUT_RND      4
51 #define PACKET_FANOUT_QM       5
52 #define PACKET_FANOUT_CBPF     6
53 #define PACKET_FANOUT_EBPF     7
54 #define PACKET_FANOUT_FLAG_ROLLOVER 0x1000
55 #define PACKET_FANOUT_FLAG_DEFRAG 0x8000

```

8.1.3 Sock CheckSum

8.1.3.1 CheckSum 相关标志位

```

1  * A. Checksumming of received packets by device.
2  *
3  * CHECKSUM_NONE:
4  *
5  * Device failed to checksum this packet e.g. due to lack of capabilities.
6  * The packet contains full (though not verified) checksum in packet but
7  * not in skb->csum. Thus, skb->csum is undefined in this case.
8  *
9  * CHECKSUM_UNNECESSARY:
10 *
11 * The hardware you're dealing with doesn't calculate the full checksum
12 * (as in CHECKSUM_COMPLETE), but it does parse headers and verify checksums
13 * for specific protocols. For such packets it will set CHECKSUM_UNNECESSARY
14 * if their checksums are okay. skb->csum is still undefined in this case
15 * though. It is a bad option, but, unfortunately, nowadays most vendors do
16 * this. Apparently with the secret goal to sell you new devices, when you
17 * will add new protocol to your host, f.e. IPv6 8)
18 *
19 * CHECKSUM_UNNECESSARY is applicable to following protocols:
20 *   TCP: IPv6 and IPv4.
21 *   UDP: IPv4 and IPv6. A device may apply CHECKSUM_UNNECESSARY to a
22 *       zero UDP checksum for either IPv4 or IPv6, the networking stack
23 *       may perform further validation in this case.
24 *   GRE: only if the checksum is present in the header.
25 *   SCTP: indicates the CRC in SCTP header has been validated.
26 *
27 * skb->csum_level indicates the number of consecutive checksums found in
28 * the packet minus one that have been verified as CHECKSUM_UNNECESSARY.
29 * For instance if a device receives an IPv6->UDP->GRE->IPv4->TCP packet

```

```

30 * and a device is able to verify the checksums for UDP (possibly zero),
31 * GRE (checksum flag is set), and TCP-- skb->csum_level would be set to
32 * two. If the device were only able to verify the UDP checksum and not
33 * GRE, either because it doesn't support GRE checksum or because GRE
34 * checksum is bad, skb->csum_level would be set to zero (TCP checksum is
35 * not considered in this case).
36 *
37 * CHECKSUM_COMPLETE:
38 *
39 * This is the most generic way. The device supplied checksum of the _whole_
40 * packet as seen by netif_rx() and fills out in skb->csum. Meaning, the
41 * hardware doesn't need to parse L3/L4 headers to implement this.
42 *
43 * Note: Even if device supports only some protocols, but is able to produce
44 * skb->csum, it MUST use CHECKSUM_COMPLETE, not CHECKSUM_UNNECESSARY.
45 *
46 * CHECKSUM_PARTIAL:
47 *
48 * A checksum is set up to be offloaded to a device as described in the
49 * output description for CHECKSUM_PARTIAL. This may occur on a packet
50 * received directly from another Linux OS, e.g., a virtualized Linux kernel
51 * on the same host, or it may be set in the input path in GRO or remote
52 * checksum offload. For the purposes of checksum verification, the checksum
53 * referred to by skb->csum_start + skb->csum_offset and any preceding
54 * checksums in the packet are considered verified. Any checksums in the
55 * packet that are after the checksum being offloaded are not considered to
56 * be verified.
57 *
58 * B. Checksumming on output.
59 *
60 * CHECKSUM_NONE:
61 *
62 * The skb was already checksummed by the protocol, or a checksum is not
63 * required.
64 *
65 * CHECKSUM_PARTIAL:
66 *
67 * The device is required to checksum the packet as seen by hard_start_xmit()
68 * from skb->csum_start up to the end, and to record/write the checksum at
69 * offset skb->csum_start + skb->csum_offset.
70 *
71 * The device must show its capabilities in dev->features, set up at device
72 * setup time, e.g. netdev_features.h:
73 *
74 * NETIF_F_HW_CSUM - It's a clever device, it's able to checksum everything.
75 * NETIF_F_IP_CSUM - Device is dumb, it's able to checksum only TCP/UDP over
76 * IPv4. Sigh. Vendors like this way for an unknown reason.
77 * Though, see comment above about CHECKSUM_UNNECESSARY. 8)
78 * NETIF_F_IPV6_CSUM - About as dumb as the last one but does IPv6 instead.
79 * NETIF_F_... - Well, you get the picture.
80 *
81 * CHECKSUM_UNNECESSARY:
82 *
83 * Normally, the device will do per protocol specific checksumming. Protocol
84 * implementations that do not want the NIC(Network Interface Controller, 网卡)
85 * to perform the checksum calculation should use this flag in their outgoing skbs.
86 *
87 * NETIF_F_FCOE_CRC - This indicates that the device can do FCoE FC CRC

```



```

88      *                                offload. Correspondingly, the FCoE protocol driver
89      *                                stack should use CHECKSUM_UNNECESSARY.
90      *
91      * Any questions? No questions, good.                                --ANK
92      */
93
94      /* Don't change this without changing skb_csum_unnecessary! */
95      #define CHECKSUM_NONE                0
96      #define CHECKSUM_UNNECESSARY        1
97      #define CHECKSUM_COMPLETE          2
98      #define CHECKSUM_PARTIAL            3

```

8.1.3.2 skb_csum_unnecessary

```

1      /*
2      Location:
3
4          include/linux/skbuff.h
5
6      Parameter:
7
8          skb: 传输控制块缓存
9      */
10     static inline int skb_csum_unnecessary(const struct sk_buff *skb)
11     {
12         return ((skb->ip_summed == CHECKSUM_UNNECESSARY) ||
13                 skb->csum_valid ||
14                 (skb->ip_summed == CHECKSUM_PARTIAL &&
15                  skb_checksum_start_offset(skb) >= 0));
16     }

```

如果不要网卡帮忙校验或者 Checksum 合法，或者硬件给出的时部分校验并且校验的偏移必须大于零。则返回 1，否则，返回 0。

8.1.3.3 __skb_checksum_complete

```

1      __sum16 __skb_checksum_complete(struct sk_buff *skb)
2      {
3          __wsum csum;
4          __sum16 sum;
5
6          csum = skb_checksum(skb, 0, skb->len, 0);
7
8          /* skb->csum holds pseudo checksum */
9          sum = csum_fold(csum_add(skb->csum, csum));
10         if (likely(!sum)) {
11             if (unlikely(skb->ip_summed == CHECKSUM_COMPLETE) &&
12                 !skb->csum_complete_sw)
13                 netdev_rx_csum_fault(skb->dev);
14         }
15
16         if (!skb_shared(skb)) {
17             /* Save full packet checksum */
18             skb->csum = csum;
19             skb->ip_summed = CHECKSUM_COMPLETE;
20             skb->csum_complete_sw = 1;
21             skb->csum_valid = !sum;

```

```

22     }
23
24     return sum;
25 }

```

8.1.4 SK Stream

Linux 内核中，提供了一套通用的处理网络数据流的函数。这一套函数是为了共享各种协议中，和处理数据流相关的代码，以实现代码重用。

8.1.5 sk_stream_wait_connect

该函数用于等待套接字完成连接。

```

1  /* Location: net/core/stream.c
2  *
3  * Parameter:
4  *     sk: 套接字
5  *     timeo_p: 等待多长时间
6  *
7  * 该函数必须在 sk 被上锁的情况下被调用。
8  */
9  int sk_stream_wait_connect(struct sock *sk, long *timeo_p)
10 {
11     struct task_struct *tsk = current;
12     DEFINE_WAIT(wait);
13     int done;
14
15     do {
16         int err = sock_error(sk);
17         if (err)
18             return err;
19         if ((1 << sk->sk_state) & ~(TCPF_SYN_SENT | TCPF_SYN_RECV))
20             return -EPIPE;
21         if (!*timeo_p)
22             return -EAGAIN;
23         if (signal_pending(tsk))
24             return sock_intr_errno(*timeo_p);
25
26         prepare_to_wait(sk_sleep(sk), &wait, TASK_INTERRUPTIBLE);
27         sk->sk_write_pending++;
28         /* 等待 TCP 进入连接建立状态 */
29         done = sk_wait_event(sk, timeo_p,
30                             !sk->sk_err &&
31                             !((1 << sk->sk_state) &
32                              ~(TCPF_ESTABLISHED | TCPF_CLOSE_WAIT)));
33         finish_wait(sk_sleep(sk), &wait);
34         sk->sk_write_pending--;
35     } while (!done);
36     return 0;
37 }
38 EXPORT_SYMBOL(sk_stream_wait_connect);

```

8.2 Inet

8.2.1 inet_hash_connect && __inet_hash_connect

8.3 Inet

8.3.1 inet_hash_connect && __inet_hash_connect

8.3.1.1 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 }

```

8.3.1.2 __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 }

```

8.3.2 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 }

```

8.4 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??

8.4.1 TCP 相关参数

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

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

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

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

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

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

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

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

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

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

- `tcp_tw_recycle` BOOLEAN, 默认值是 0 打开快速 TIME-WAIT sockets 回收。除非得到技术专家的建议或要求, 请不要随意修改这个值。(做 NAT 的时候, 建议打开它)
- `tcp_tw_reuse` BOOLEAN, 默认值是 0 该文件表示是否允许重新应用处于 TIME-WAIT 状态的 socket 用于新的 TCP 连接 (这个对快速重启某些服务, 而启动后提示端口已经被使用的情形非常有帮助)
- `tcp_max_orphans` INTEGER, 缺省值是 8192 系统所能处理不属于任何进程的 TCP sockets 最大数量。假如超过这个数量, 那么不属于任何进程的连接会被立即 reset, 并同时显示警告信息。之所以要设定这个限制, 纯粹为了抵御那些简单的 DoS 攻击, 千万不要依赖这个或是人为的降低这个限制 (这个值 Redhat AS 版本中设置为 32768, 但是很多防火墙修改的时候, 建议该值修改为 2000)
- `cp_abort_on_overflow` BOOLEAN, 缺省值是 0 当守护进程太忙而不能接受新的连接, 就象对方发送 reset 消息, 默认值是 false。这意味着当溢出的原因是因为一个偶然的猝发, 那么连接将恢复状态。只有在你确信守护进程真的不能完成连接请求时才打开该选项, 该选项会影响客户的使用。(对待已经满载的 sendmail, apache 这类服务的时候, 这个可以很快让客户端终止连接, 可以给予服务程序处理已有连接的缓冲机会, 所以很多防火墙上推荐打开它)
- `tcp_syncookies` BOOLEAN, 默认值是 0 只有在内核编译时选择了 CONFIG_SYNCOOKIES 时才会发生作用。当出现 syn 等候队列出现溢出时象对方发送 syncookies。目的是为了防止 syn flood 攻击。注意: 该选项千万不能用于那些没有收到攻击的高负载服务器, 如果在日志中出现 synflood 消息, 但是调查发现没有收到 synflood 攻击, 而是合法用户的连接负载过高的原因, 你应该调整其它参数来提高服务器性能。参考: `tcp_max_syn_backlog`, `tcp_synack_retries`, `tcp_abort_on_overflow`。syncookie 严重的违背 TCP 协议, 不允许使用 TCP 扩展, 可能对某些服务导致严重的性能影响 (如 SMTP 转发)。(注意, 该实现与 BSD 上面使用的 tcp proxy 一样, 是违反了 RFC 中关于 tcp 连接的三次握手实现的, 但是对于防御 syn-flood 的确很有用。)
- `tcp_stdurg` BOOLEAN, 默认值为 0 使用 TCP urg pointer 字段中的主机请求解释功能。大部份的主机都使用老旧的 BSD 解释, 因此如果您在 Linux 打开它, 或会导致不能和它们正确沟通。
- `tcp_max_syn_backlog` INTEGER, 对于那些依然还未获得客户端确认的连接请求, 需要保存在队列中最大数目。对于超过 128Mb 内存的系统, 默认值是 1024, 低于 128Mb 的则为 128。如果服务器经常出现过载, 可以尝试增加这个数字。警告! 假如您将此值设为大于 1024, 最好修改 `include/net/tcp.h` 里面的 `TCP_SYNQ_HSIZE`, 以保持 `TCP_SYNQ_HSIZE*16 <= tcp_max_syn_backlog`, 并且编进核心之内。(SYN Flood 攻击利用 TCP 协议散握手的缺陷, 伪造虚假源 IP 地址发送大量 TCP-SYN 半打开连接到目标系统, 最终导致目标系统 Socket 队列资源耗尽而无法接受新的连接。为了应付这种攻击, 现代 Unix 系统中普遍采用多连接队列处理的方式来缓冲 (而不是解决) 这种攻击, 是用一个基本队列处理正常的完全连接应用

(Connect() 和 Accept()), 是用另一个队列单独存放半打开连接。这种双队列处理方式和其他一些系统内核措施 (例如 Syn-Cookies/Caches) 联合应用时, 能够比较有效的缓解小规模 SYN Flood 攻击 (事实证明 <1000p/s) 加大 SYN 队列长度可以容纳更多等待连接的网络连接数, 所以对 Server 来说可以考虑增大该值.)

`tcp_window_scaling` INTEGER, 缺省值为 1 该文件表示设置 tcp/ip 会话的滑动窗口大小是否可变。参数值为布尔值, 为 1 时表示可变, 为 0 时表示不可变。tcp/ip 通常使用的窗口最大可达到 65535 字节, 对于高速网络, 该值可能太小, 这时候如果启用了该功能, 可以使 tcp/ip 滑动窗口大小增大数个数量级, 从而提高数据传输的能力 (RFC 1323)。(对普通地百 M 网络而言, 关闭会降低开销, 所以如果不是高速网络, 可以考虑设置为 0)

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

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

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

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

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

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

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

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

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

max: 用于 TCP socket 发送缓冲的内存最大值。该值不会影响 `net.core.wmem_max`, "静态" 选择参数 `SO_SNDBUF` 则不受该值影响。默认值为 131072(128K)。(对

于服务器而言，增加这个参数的值对于发送数据很有帮助，在我的网络环境中，修改为了 51200 131072 204800)

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

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

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

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

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

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

一般情况下这些值是在系统启动时根据系统内存数量计算得到的。

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

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

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

tcp_low_latency BOOLEAN, 缺省值为 0。该段的英文阐述如下：If set, the TCP stack makes decisions that prefer lower latency as opposed to higher throughput. By default, this option is not set meaning that higher throughput is preferred. An example of an application where this default should be changed would be a Beowulf compute cluster. 可以看出该选项在设置的时候主要是用于高吞吐量的环境，而设置之后，

则是主要用于低延时的环境。这个选项一般情形是禁用，但在构建 Beowulf 集群 (超算) 的时候, 打开它很有帮助。

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

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

8.4.2 TCP 相关宏定义

8.4.2.1 标志宏

```

1  #define FLAG_DATA      0x01      /* Incoming frame contained data.      */
2  #define FLAG_WIN_UPDATE 0x02      /* Incoming ACK was a window update.  */
3  #define FLAG_DATA_ACKED 0x04      /* This ACK acknowledged new data.    */
4  #define FLAG_RETRANS_DATA_ACKED 0x08 /* " " " some of which was retransmitted. */
5  #define FLAG_SYN_ACKED 0x10      /* This ACK acknowledged SYN.        */
6  #define FLAG_DATA_SACKED 0x20     /* New SACK.                          */
7  #define FLAG_ECE       0x40      /* ECE in this ACK                    */
8  #define FLAG_LOST_RETRANS 0x80    /* This ACK marks some retransmission lost */
9  #define FLAG_SLOWPATH   0x100    /* Do not skip RFC checks for window update.*/
10 #define FLAG_ORIG_SACK_ACKED 0x200 /* Never retransmitted data are (s)acked */
11 #define FLAG_SND_UNA_ADVANCED 0x400 /* Snd_una was changed (!= FLAG_DATA_ACKED) */
12 #define FLAG_DSACKING_ACK 0x800   /* SACK blocks contained D-SACK info */
13 #define FLAG_SACK_RENEGING 0x2000 /* snd_una advanced to a sacked seq */
14 #define FLAG_UPDATE_TS_RECENT 0x4000 /* tcp_replace_ts_recent() */
15
16 #define FLAG_ACKED      (FLAG_DATA_ACKED/FLAG_SYN_ACKED)
17 #define FLAG_NOT_DUP    (FLAG_DATA/FLAG_WIN_UPDATE/FLAG_ACKED)
18 #define FLAG_CA_ALERT   (FLAG_DATA_SACKED/FLAG_ECE)
19 #define FLAG_FORWARD_PROGRESS (FLAG_ACKED/FLAG_DATA_SACKED)
20
21 #define TCP_REMNANT (TCP_FLAG_FIN/TCP_FLAG_URG/TCP_FLAG_SYN/TCP_FLAG_PSH)
22 #define TCP_HP_BITS (~ (TCP_RESERVED_BITS/TCP_FLAG_PSH))

```

这里我们需要介绍几个相关的概念, 如下:

SACK Selective Acknowledgment(SACK), 这种方式需要在 TCP 头里加一个 SACK 的东西, ACK 还是 Fast Retransmit 的 ACK, SACK 则是汇报收到的数据碎版。这样, 在发送端就可以根据回传的 SACK 来知道哪些数据到了, 哪些没有到。于是就优化了 Fast Retransmit 的算法。当然, 这个协议需要两边都支持。

Reneging 所谓 Reneging 的意思就是违约, 接收方有权把已经报给发送端 SACK 里的数据给丢了。当然, 我们肯定是不鼓励这样做的, 因为这个事会把问题复杂化。但是, 接收方可能会由于一些极端情况这么做, 比如要把内存给别的更重要的东西。所以, 发送方也不能完全依赖 SACK, 主要还是要依赖 ACK, 并维护 Time-Out。如果后续的 ACK 没有增长, 那么还是要把 SACK 的东西重传, 另外, 接收端这边永远不能把 SACK 的包标记为 Ack。 **注意: SACK 会消费发送方的资源, 试想,**

如果一个攻击者给数据发送方发一堆 SACK 的选项，这会导致发送方开始要重传甚至遍历已经发出的数据，这会消耗很多发送端的资源。

D-SACK Duplicate SACK 又称 D-SACK，其主要使用了 SACK 来告诉发送方有哪些数据被重复接收了。D-SACK 使用了 SACK 的第一个段来做标志，如果 SACK 的第一个段的范围被 ACK 所覆盖，那么就是 D-SACK。如果 SACK 的第一个段的范围被 SACK 的第二个段覆盖，那么就是 D-SACK。引入了 D-SACK，有这么几个好处：

- 1) 可以让发送方知道，是发出去的包丢了，还是回来的 ACK 包丢了。
- 2) 是不是自己的 timeout 太小了，导致重传。
- 3) 网络上出现了先发的包后到的情况（又称 reordering）
- 4) 网络上是不是把我的数据包给复制了。

8.4.2.2 函数宏

8.4.3 TCP CheckSum

8.4.3.1 tcp_checksum_complete

该函数时基于伪首部累加和，完成全包校验和的检测，值得注意的是，该函数用于校验没有负载的 TCP 段。如果需要校验，并且校验成功，则返回 1。

```

1  /*
2  Location:
3      include/net/tcp.h
4  */
5  static inline bool tcp_checksum_complete(struct sk_buff *skb)
6  {
7      return !skb_csum_unnecessary(skb) &&
8              __tcp_checksum_complete(skb);
9  }
```

对于__tcp_checksum_complete函数，如下：

```

1  /*
2  Location:
3      include/net/tcp.h
4  */
5  static inline __sum16 __tcp_checksum_complete(struct sk_buff *skb)
6  {
7      return __skb_checksum_complete(skb);
8  }
```

对于函数__skb_checksum_complete, 参见 BSD Socket 处的叙述。

8.4.3.2 tcp_v4_checksum_init

```

1
```

8.4.4 __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 }

```

8.4.5 About ACK

8.4.5.1 tcp_is_sack & tcp_is_reno & tcp_is_fack

```

1  /*
2   Location:
3
4   include/net/tcp.h
5
6   Description:
7   These functions determine how the current flow behaves in respect of SACK
8   handling. SACK is negotiated with the peer, and therefore it can vary
9   between different flows.
10
11   tcp_is_sack - SACK enabled
12   tcp_is_reno - No SACK
13   tcp_is_fack - FACK enabled, implies SACK enabled
14  */
15  static inline int tcp_is_sack(const struct tcp_sock *tp)
16  {
17      return tp->rx_opt.sack_ok;
18  }
19
20  static inline bool tcp_is_reno(const struct tcp_sock *tp)
21  {
22      return !tcp_is_sack(tp);
23  }
24
25  static inline bool tcp_is_fack(const struct tcp_sock *tp)
26  {
27      return tp->rx_opt.sack_ok & TCP_FACK_ENABLED;
28  }

```

8.4.5.2 tcp_reset_reno_sack

```

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

```

```

7
8     将 sacked_out 字段置为 0
9
10    Parameter:
11
12     tp:???
13
14    */
15    static inline void tcp_reset_reno_sack(struct tcp_sock *tp)
16    {
17        tp->sacked_out = 0;
18    }

```

8.4.5.3 tcp_ack_is_dubious

```

1    /*
2    Location:
3
4     net/ipv4/tcp_input.c
5
6    Function:
7
8     判断一个 ACK 是否可疑。
9
10   Parameter:
11
12     sk: 传输控制块
13     flag:FLAG 标志位
14   */
15   static inline bool tcp_ack_is_dubious(const struct sock *sk, const int flag)
16   {
17       return !(flag & FLAG_NOT_DUP) || (flag & FLAG_CA_ALERT) ||
18           inet_csk(sk)->icsk_ca_state != TCP_CA_Open;
19   }

```

什么样的 ACK 算是可疑的呢? 如下:

非 FLAG_NOT_DUP 通过查看相关的宏定义我们知道 FLAG_NOT_DUP 为 (FLAG_DATA|FLAG_WIN_UPDATE|FLAG_ACKED), 故而 FLAG_ACKED 又为 (FLAG_DATA_ACKED|FLAG_SYN_ACKED), 故而即其所包含的种类有接收的 ACK 段是 (1) 负荷数据携带的;(2) 更新窗口的;(3) 确认新数据的;(4) 确认 SYN 段的。如果上述四种都不属于那是可疑的了。

FLAG_CA_ALERT 通过查看相关的宏定义我们知道 FLAG_CA_ALERT 为 (FLAG_DATA_SACKED|FLAG_ECE), 如果被发现时确认新数据的 ACK 或者在 ACK 中存在 ECE 标志, 即收到显式拥塞通知, 也认为是可疑的。似乎和上面的有矛盾,

非 Open 即当前拥塞状态不为 Open。

8.4.5.4 tcp_check_sack_reneging

如果接收到的确认 ACK 指向之前记录的 SACK, 这说明之前记录的 SACK 并没有反映接收方的真实状态。接收路径上很有可能已经有拥塞发生或者接收主机正在经历严重的拥塞甚至处理出现了 BUG, 因为按照正常的逻辑流程, 接收的 ACK 不应该指向已

记录的 SACK, 而应该指向 SACK 后面未接收的地方。通常情况下, 此时接收方已经删除了保存到失序队列中的段。

为了避免短暂奇怪的看起来像是违约的 SACK 导致更大量的重传, 我们给接收者一些时间, 即 $\max(RTT/2, 10ms)$ 以便于让他可以给我们更多的 ACK, 从而可以使得 SACK 的记分板变得正常一点。如果这个表面上的违约一直持续到重传时间结束, 我们就把 SACK 的记分板清除掉。

```

1  /*
2  Location:
3
4      net/ipv4/tcp_input.c
5
6  Function:
7
8      If ACK arrived pointing to a remembered SACK, it means that our
9      remembered SACKs do not reflect real state of receiver i.e.
10     receiver _host_ is heavily congested (or buggy).
11
12     To avoid big spurious(假的) retransmission bursts(爆发) due to transient SACK
13     scoreboard oddities that look like reneging, we give the receiver a
14     little time ( $\max(RTT/2, 10ms)$ ) to send us some more ACKs that will
15     restore sanity to the SACK scoreboard. If the apparent reneging
16     persists until this RTO then we'll clear the SACK scoreboard.
17
18 Paramater:
19
20     sk: 传输控制块。
21     flag: 相关标志位
22 */
23 static bool tcp_check_sack_reneging(struct sock *sk, int flag)
24 {
25     //如果确认接收方违约了。
26     if (flag & FLAG_SACK_RENEGING) {
27         struct tcp_sock *tp = tcp_sk(sk);
28         //计算超时重传时间
29         unsigned long delay = max(usecs_to_jiffies(tp->srtt_us >> 4),
30                                   msecs_to_jiffies(10));
31         //更新超时重传定时器      ??? 有时间好好分析一下。
32         inet_csk_reset_xmit_timer(sk, ICSK_TIME_RETRANS,
33                                   delay, TCP_RTO_MAX);
34         return true;
35     }
36     return false;
37 }

```

8.4.5.5 tcp_limit_reno_sacked

```

1  /*
2  Location:
3
4      net/ipv4/tcp_input.c
5
6  Function:
7
8      Limits sacked_out so that sum with lost_out isn't ever larger than

```

```

9      packets_out. Returns false if sacked_out adjustment wasn't necessary.
10
11      Parameter:
12
13          tp: ???
14      */
15      static bool tcp_limit_reno_sacked(struct tcp_sock *tp)
16      {
17          u32 holes; //记录丢失的包, ??? 很奇怪, 为什么没能直接利用 lost_out 呢?
18          //既然由重复的 ACK, 那么必然是之前传输的段有丢失的了, 所以至少为 1。
19          holes = max(tp->lost_out, 1U);
20          //但是又不能大于所有的发出去的。
21          holes = min(holes, tp->packets_out);
22
23          if ((tp->sacked_out + holes) > tp->packets_out) {
24              tp->sacked_out = tp->packets_out - holes;
25              return true;
26          }
27          return false;
28      }
29
30      该函数用于检查sacked_out是否过多, 过多则限制, 且返回true, 说明出现reordering了。
31
32      那么我们怎么判断是否有reordering呢? 我们知道dupack可能由lost引起, 也有可能由reorder引起, 那么如果
33      sacked_out + lost_out > packets_out, 则说明sacked_out偏大了, 因为它错误的把由reorder
34      引起的dupack当客户端的sack了。

```

8.4.5.6 tcp_check_reno_reordering

```

1      /*
2      Location:
3
4          net/ipv4/tcp_input.c
5
6      Function:
7
8          If we receive more dupacks than we expected counting segments
9          in assumption of absent reordering, interpret this as reordering.
10         The only another reason could be bug in receiver TCP.
11
12      Parameter:
13
14          sk: 传输控制块
15          addend: ???
16      */
17      static void tcp_check_reno_reordering(struct sock *sk, const int addend)
18      {
19          struct tcp_sock *tp = tcp_sk(sk);
20          //判断是否有 reordering 了
21          if (tcp_limit_reno_sacked(tp))
22              tcp_update_reordering(sk, tp->packets_out + addend, 0); //??? 这个函数到底时用于干什么呢?
23      }

```

8.4.5.7 tcp_add_reno_sack

```

1  /*
2  Location:
3
4      net/ipv4/tcp_input.c
5
6  Location:
7
8      Emulate(仿真) SACKs for SACKless connection: account for a new dupack.
9
10 Parameter:
11
12     sk: 传输控制块
13  */
14
15 static void tcp_add_reno_sack(struct sock *sk)
16 {
17     struct tcp_sock *tp = tcp_sk(sk);
18     tp->sacked_out++;           //增加 sacked 的包数.
19     tcp_check_reno_reordering(sk, 0); // 检查是否有 reordering
20     tcp_verify_left_out(tp);      //判断 left_out 是否大于 packets_out, 出警告信息
21 }

```

8.4.6 About Window Size and Segment Sent

8.4.6.1 tcp_left_out

该函数用于计算已经发出去的 TCP 段 (离开主机) 中一共有多少个 tcp 段还未得到确认。

```

1  static inline unsigned int tcp_left_out(const struct tcp_sock *tp)
2  {
3      return tp->sacked_out + tp->lost_out;
4  }

```

8.4.6.2 tcp_verify_left_out

该函数的功能主要是判断 left_out 是否大于 packets_out, 当然, 这是不可能的, 因为前者是已经发送离开主机的未被确认的段数, 而后者是已经离开发送队列 (不一定离开主机) 但未确认的段数。故而, 这里有一个 WARN_ON, 以便输出相应的警告信息。

```

1  /*
2  Location:
3
4      include/net/tcp.h
5
6  Function:
7
8      Use define here intentionally(有意地) to get WARN_ON location shown at the caller
9
10 Parameter:
11
12  */
13 #define tcp_verify_left_out(tp) WARN_ON(tcp_left_out(tp) > tp->packets_out)

```


8.4.7 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 }
```

8.4.8 About Congestion Control

8.4.8.1 tcp_end_cwnd_reduction

```

1  /*
2  Location:
3
4      net/ipv4/tcp_input.c
5
6  Function:
7
8      结束拥塞窗口减小。
9
10 Parameter:
11
12      sk: 传输控制块。remain to do in the future ???
13 */
14 static inline void tcp_end_cwnd_reduction(struct sock *sk)
15 {
16     struct tcp_sock *tp = tcp_sk(sk);
17
18     /* Reset cwnd to ssthresh in CWR or Recovery (unless it's undone) */
19     if (inet_csk(sk)->icsk_ca_state == TCP_CA_CWR ||
20         (tp->undo_marker && tp->snd_ssthresh < TCP_INFINITE_SSTHRESH)) {
21         tp->snd_cwnd = tp->snd_ssthresh;
22         tp->snd_cwnd_stamp = tcp_time_stamp;
23     }
24     tcp_ca_event(sk, CA_EVENT_COMPLETE_CWR);
25 }
```

8.4.8.2 tcp_try_undo_recovery

```

1  /*
2  Location:
3
4      net/ipv4/tcp_input.c
5
6  Function:
7
8      尝试从恢复状态撤销。
9
10 Parameter:
```

```

11
12     sk: 传输控制块。remain to do in the future ????
13     */
14     static bool tcp_try_undo_recovery(struct sock *sk)
15     {
16         struct tcp_sock *tp = tcp_sk(sk);
17
18         if (tcp_may_undo(tp)) {
19             int mib_idx;
20
21             /* Happy end! We did not retransmit anything
22             * or our original transmission succeeded.
23             */
24             DBGUNDO(sk, inet_csk(sk)->icsk_ca_state == TCP_CA_Loss ? "loss" : "retrans");
25             tcp_undo_cwnd_reduction(sk, false);
26             if (inet_csk(sk)->icsk_ca_state == TCP_CA_Loss)
27                 mib_idx = LINUX_MIB_TCPLOSSUNDO;
28             else
29                 mib_idx = LINUX_MIB_TCPFULLUNDO;
30
31             NET_INC_STATS_BH(sock_net(sk), mib_idx);
32         }
33         if (tp->snd_una == tp->high_seq && tcp_is_reno(tp)) {
34             /* Hold old state until something *above* high_seq
35             * is ACKed. For Reno it is MUST to prevent false
36             * fast retransmits (RFC2582). SACK TCP is safe. */
37             tcp_moderate_cwnd(tp);
38             if (!tcp_any_retrans_done(sk))
39                 tp->retrans_stamp = 0;
40             return true;
41         }
42         tcp_set_ca_state(sk, TCP_CA_Open);
43         return false;
44     }

```

8.4.9 About Congestion Control

8.4.9.1 sysctl_tcp_recovery

```

1     int sysctl_tcp_recovery __read_mostly = TCP_RACK_LOST_RETRANS;

```

8.4.9.2 tcp_rack_mark_lost

```

1     /*
2     Location:
3
4     net/ipv4/tcp_recovery.c
5
6     Function:
7
8     Marks a packet lost, if some packet sent later has been (s)acked.
9     The underlying idea is similar to the traditional dupthresh and FACK
10    but they look at different metrics:
11
12    dupthresh: 3 000 packets delivered (packet count)
13    FAK: sequence delta to highest sacked sequence (sequence space)
14    RACK: sent time delta to the latest delivered packet (time domain)

```

```

15
16     The advantage of RACK is it applies to both original and retransmitted
17     packet and therefore is robust against tail losses. Another advantage
18     is being more resilient to reordering by simply allowing some
19     "settling delay", instead of tweaking the dupthresh.
20
21     The current version is only used after recovery starts but can be
22     easily extended to detect the first loss.
23
24     Parameter:
25
26
27
28     */
29
30     int tcp_rack_mark_lost(struct sock *sk)
31     {
32         struct tcp_sock *tp = tcp_sk(sk);
33         struct sk_buff *skb;
34         u32 reo_wnd, prior_retrans = tp->retrans_out;
35
36         if (inet_csk(sk)->icsk_ca_state < TCP_CA_Recovery || !tp->rack.advanced)
37             return 0;
38
39         /* Reset the advanced flag to avoid unnecessary queue scanning */
40         tp->rack.advanced = 0;
41
42         /* To be more reordering resilient, allow min_rtt/4 settling delay
43          * (lower-bounded to 1000uS). We use min_rtt instead of the smoothed
44          * RTT because reordering is often a path property and less related
45          * to queuing or delayed ACKs.
46          *
47          * TODO: measure and adapt to the observed reordering delay, and
48          * use a timer to retransmit like the delayed early retransmit.
49          */
50         reo_wnd = 1000;
51         if (tp->rack.reord && tcp_min_rtt(tp) != ~0U)
52             reo_wnd = max(tcp_min_rtt(tp) >> 2, reo_wnd);
53
54         tcp_for_write_queue(skb, sk) {
55             struct tcp_skb_cb *scb = TCP_SKB_CB(skb);
56
57             if (skb == tcp_send_head(sk))
58                 break;
59
60             /* Skip ones already (s)acked */
61             if (!after(scb->end_seq, tp->snd_una) ||
62                 scb->sacked & TCPCB_SACKED_ACKED)
63                 continue;
64
65             if (skb_mstamp_after(&tp->rack.mstamp, &skb->skb_mstamp)) {
66
67                 if (skb_mstamp_us_delta(&tp->rack.mstamp,
68                                         &skb->skb_mstamp) <= reo_wnd)
69                     continue;
70
71                 /* skb is lost if packet sent later is sacked */
72                 tcp_skb_mark_lost_uncond_verify(tp, skb);
73                 if (scb->sacked & TCPCB_SACKED_RETRANS) {

```

```

74         scb->sacked &= ~TCP_CB_SACKED_RETRANS;
75         tp->retrans_out -= tcp_skb_pcount(skb);
76         NET_INC_STATS_BH(sock_net(sk),
77             LINUX_MIB_TCPLOSTRETRANSMIT);
78     }
79     } else if (!(scb->sacked & TCP_CB_RETRANS)) {
80         /* Original data are sent sequentially so stop early
81          * b/c the rest are all sent after rack_sent
82          */
83         break;
84     }
85 }
86 return prior_retrans - tp->retrans_out;
87 }
88
89 /* Record the most recently (re)sent time among the (s)acked packets */
90 void tcp_rack_advance(struct tcp_sock *tp,
91     const struct skb_mstamp *xmit_time, u8 sacked)
92 {
93     if (tp->rack.mstamp.v64 &&
94         !skb_mstamp_after(xmit_time, &tp->rack.mstamp))
95         return;
96
97     if (sacked & TCP_CB_RETRANS) {
98         struct skb_mstamp now;
99
100         /* If the sacked packet was retransmitted, it's ambiguous
101          * whether the retransmission or the original (or the prior
102          * retransmission) was sacked.
103          *
104          * If the original is lost, there is no ambiguity. Otherwise
105          * we assume the original can be delayed up to aRTT + min_rtt.
106          * the aRTT term is bounded by the fast recovery or timeout,
107          * so it's at least one RTT (i.e., retransmission is at least
108          * an RTT later).
109          */
110         skb_mstamp_get(&now);
111         if (skb_mstamp_us_delta(&now, xmit_time) < tcp_min_rtt(tp))
112             return;
113     }
114
115     tp->rack.mstamp = *xmit_time;
116     tp->rack.advanced = 1;
117 }

```

8.4.9.3

```

1  /*
2  Location:
3
4      net/ipv4/tcp_input.c
5
6  Function:
7
8      Try to undo cwnd reduction, because D-SACKs acked all retransmitted data
9
10 Parameter:
11

```

```

12     sk: 传输控制块
13     */
14     static bool tcp_try_undo_dsack(struct sock *sk)
15     {
16         struct tcp_sock *tp = tcp_sk(sk);
17
18         if (tp->undo_marker && !tp->undo_retrans) {
19             DBGUNDO(sk, "D-SACK");
20             tcp_undo_cwnd_reduction(sk, false);
21             NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPDSACKUNDO);
22             return true;
23         }
24         return false;
25     }

```

8.4.9.4 tcp_process_loss

```

1     /*
2     Location:
3
4         net/ipv4/tcp_input.c
5
6     Function:
7
8         Process an ACK in CA_Loss state. Move to CA_Open if lost data are
9         recovered or spurious. Otherwise retransmits more on partial ACKs.
10
11     Parameter:
12
13         sk: 传输控制块
14         flag:
15         is_dupack:
16     */
17     static void tcp_process_loss(struct sock *sk, int flag, bool is_dupack)
18     {
19         struct tcp_sock *tp = tcp_sk(sk);
20         bool recovered = !before(tp->snd_una, tp->high_seq);
21
22         if ((flag & FLAG_SND_UNA_ADVANCED) &&
23             tcp_try_undo_loss(sk, false))
24             return;
25
26         if (tp->frto) { /* F-RTT RFC5682 sec 3.1 (sack enhanced version). */
27             /* Step 3.b. A timeout is spurious if not all data are
28              * lost, i.e., never-retransmitted data are (s)acked.
29              */
30             if ((flag & FLAG_ORIG_SACK_ACKED) &&
31                 tcp_try_undo_loss(sk, true))
32                 return;
33
34             if (after(tp->snd_nxt, tp->high_seq)) {
35                 if (flag & FLAG_DATA_SACKED || is_dupack)
36                     tp->frto = 0; /* Step 3.a. loss was real */
37             } else if (flag & FLAG_SND_UNA_ADVANCED && !recovered) {
38                 tp->high_seq = tp->snd_nxt;
39                 __tcp_push_pending_frames(sk, tcp_current_mss(sk),
40                     TCP_NAGLE_OFF);
41                 if (after(tp->snd_nxt, tp->high_seq))

```

```

42         return; /* Step 2.b */
43         tp->frto = 0;
44     }
45 }
46
47 if (recovered) {
48     /* F-RTO RFC5682 sec 3.1 step 2.a and 1st part of step 3.a */
49     tcp_try_undo_recovery(sk);
50     return;
51 }
52 if (tcp_is_reno(tp)) {
53     /* A Reno DUPACK means new data in F-RTO step 2.b above are
54      * delivered. Lower inflight to clock out (re)transmissions.
55      */
56     if (after(tp->snd_nxt, tp->high_seq) && is_dupack)
57         tcp_add_reno_sack(sk);
58     else if (flag & FLAG_SND_UNA_ADVANCED)
59         tcp_reset_reno_sack(tp);
60 }
61 tcp_xmit_retransmit_queue(sk);
62 }

```

8.4.9.5 tcp_try_undo_partial

```

1  /*
2  Location:
3
4      net/ipv4/tcp_input.c
5
6  Function:
7
8      Undo during fast recovery after partial ACK.
9
10 Parameter:
11
12     sk: 传输控制块。
13     acked:
14     prior_unacked:
15     flag:
16 */
17 static bool tcp_try_undo_partial(struct sock *sk, const int acked,
18                                const int prior_unacked, int flag)
19 {
20     struct tcp_sock *tp = tcp_sk(sk);
21
22     if (tp->undo_marker && tcp_packet_delayed(tp)) {
23         /* Plain luck! Hole if filled with delayed
24          * packet, rather than with a retransmit.
25          */
26         tcp_update_reordering(sk, tcp_fackets_out(tp) + acked, 1);
27
28         /* We are getting evidence that the reordering degree is higher
29          * than we realized. If there are no retransmits out then we
30          * can undo. Otherwise we clock out new packets but do not
31          * mark more packets lost or retransmit more.
32          */
33         if (tp->retrans_out) {
34             tcp_cwnd_reduction(sk, prior_unacked, 0, flag);

```

```

35         return true;
36     }
37
38     if (!tcp_any_retrans_done(sk))
39         tp->retrans_stamp = 0;
40
41     DBGUNDO(sk, "partial recovery");
42     tcp_undo_cwnd_reduction(sk, true);
43     NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPPARTIALUNDO);
44     tcp_try_keep_open(sk);
45     return true;
46 }
47 return false;
48 }

```

8.4.10 About Retransmit

8.4.10.1 Related Macro

```

1  /*
2      include/net/tcp.h
3  */
4  /* Use TCP RACK to detect (some) tail and retransmit losses */
5  #define TCP_RACK_LOST_RETRANS 0x1

```

8.4.11 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 }

```

8.4.12 tcp__init__nondata__skb

该函数提供了的功能。函数如下：

```

1  Location:
2
3      net/ipv4/tcp_output.c
4
5  Function:
6
7      初始化不含数据的skb.
8
9  Parameter:
10
11      skb: 待初始化的sk_buff.
12      seq: 序号
13      flags: 标志位
14  static void tcp_init_nondata_skb(struct sk_buff *skb, u32 seq, u8 flags)
15  {
16      /* 设置校验码 */
17      skb->ip_summed = CHECKSUM_PARTIAL;
18      skb->csum = 0;
19
20      /* 设置标志位 */
21      TCP_SKB_CB(skb)->tcp_flags = flags;
22      TCP_SKB_CB(skb)->sacked = 0;
23
24      tcp_skb_pcount_set(skb, 1);
25
26      /* 设置起始序号 */
27      TCP_SKB_CB(skb)->seq = seq;
28      if (flags & (TCPHDR_SYN | TCPHDR_FIN))
29          seq++;
30      TCP_SKB_CB(skb)->end_seq = seq;
31  }

```

8.4.13 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。

8.4.14 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` 时向对方发送 FIN。

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

8.4.15 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.                                --ANK
35      */
36
37      if (sk->sk_state == TCP_FIN_WAIT2) {
38          struct tcp_sock *tp = tcp_sk(sk);
39          if (tp->linger2 < 0) {
40              tcp_set_state(sk, TCP_CLOSE);
41              tcp_send_active_reset(sk, GFP_ATOMIC);
42              NET_INC_STATS_BH(sock_net(sk),
43                              LINUX_MIB_TCPABORTONLINGER);
44          } else {
45              const int tmo = tcp_fin_time(sk);
46
47              if (tmo > TCP_TIMEWAIT_LEN) {
48                  inet_csk_reset_keepalive_timer(sk,
49                                                  tmo - TCP_TIMEWAIT_LEN);
50              } else {
51                  tcp_time_wait(sk, TCP_FIN_WAIT2, tmo);
52                  goto out;
53              }
54          }
55      }
56      if (sk->sk_state != TCP_CLOSE) {
57          sk_mem_reclaim(sk);
58          if (tcp_check_oom(sk, 0)) {
59              tcp_set_state(sk, TCP_CLOSE);
60              tcp_send_active_reset(sk, GFP_ATOMIC);
61              NET_INC_STATS_BH(sock_net(sk),
62                              LINUX_MIB_TCPABORTONMEMORY);
63          }
64      }
65
66      if (sk->sk_state == TCP_CLOSE) {
67          struct request_sock *req = tcp_sk(sk)->fastopen_rsk;
68          /* We could get here with a non-NULL req if the socket is
69          * aborted (e.g., closed with unread data) before 3WHS

```

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

CHAPTER 9

附录: 基础知识

Contents

9.1	计算机底层知识	172
9.1.1	机器数	172
9.1.1.1	正负数与定点数浮点数的表示	172
9.1.1.2	原码	172
9.1.1.3	反码	172
9.1.1.4	补码	172
9.2	GNU/LINUX	173
9.2.1	错误处理	173
9.2.1.1	错误码	173
9.2.1.2	IS_ERR, PTR_ERR, ERR_PTR	175
9.2.2	调试函数	177
9.2.2.1	BUG_ON	177
9.2.2.2	WARN_ON	177
9.3	C 语言	177
9.3.1	结构体初始化	177
9.3.2	位字段	178
9.4	GCC	178
9.4.1	__attribute__	178
9.4.1.1	设置变量属性	179
9.4.1.2	设置类型属性	179
9.4.1.3	设置函数属性	179
9.4.2	分支预测优化	179
9.5	Sparse	180
9.5.1	__bitwise	180

9.6 操作系统	181
9.6.1 RCU	181
9.7 CPU	181
9.8 存储系统	181
9.8.1 字节序	181
9.8.2 缓存 Cache	182
9.8.2.1 __read_mostly	182

9.1 计算机底层知识

9.1.1 机器数

在计算机内的数（称之为“机器数”）值有 3 种表示法：原码、反码和补码。to do
???? 添加关于反码运算的规则，校验码的计算。

9.1.1.1 正负数与定点数浮点数的表示

在计算机内，通常把二进制数的最高位定义为符号位，用 0 表示正数，用 1 表示负数。其余表示数值。

规定小数点位置固定不变的数称为定点数，小数点位置可以变的数称为浮点数。

9.1.1.2 原码

原码 (true form) 是一种计算机中对数字的二进制定点表示方法。原码表示法在数值前面增加了一位符号位（即最高位为符号位）：正数该位为 0，负数该位为 1（0 有两种表示：+0 和 -0），其余位表示数值的大小。

9.1.1.3 反码

正数的反码与其原码相同；负数的反码是对其原码逐位取反，但符号位除外。

反码的运算规则。

9.1.1.4 补码

正数的补码就是其本身，负数的补码是在其原码的基础上，符号位不变，其余各位取反，最后 +1。

由于补码利于运算，故而计算机中数都是利用补码表示的。

9.2 GNU/LINUX

9.2.1 错误处理

9.2.1.1 错误码

```

1  /*
2  Location:
3
4      1 - 34    include/uapi/asm-generic/errno-base.h
5      35- 133   include/uapi/asm-generic/errno.h
6
7  Function:
8
9      定义了 Linux 中的错误码。
10
11  */
12  #define      EPERM          1          /* Operation not permitted */
13  #define      ENOENT         2          /* No such file or directory */
14  #define      ESRCH          3          /* No such process */
15  #define      EINTR          4          /* Interrupted system call */
16  #define      EIO            5          /* I/O error */
17  #define      ENXIO          6          /* No such device or address */
18  #define      E2BIG          7          /* Argument list too long */
19  #define      ENOEXEC        8          /* Exec format error */
20  #define      EBADF          9          /* Bad file number */
21  #define      ECHILD         10         /* No child processes */
22  #define      EAGAIN         11         /* Try again */
23  #define      ENOMEM         12         /* Out of memory */
24  #define      EACCES         13         /* Permission denied */
25  #define      EFAULT         14         /* Bad address */
26  #define      ENOTBLK        15         /* Block device required */
27  #define      EBUSY          16         /* Device or resource busy */
28  #define      EEXIST         17         /* File exists */
29  #define      EXDEV          18         /* Cross-device link */
30  #define      ENODEV         19         /* No such device */
31  #define      ENOTDIR        20         /* Not a directory */
32  #define      EISDIR         21         /* Is a directory */
33  #define      EINVAL         22         /* Invalid argument */
34  #define      ENFILE         23         /* File table overflow */
35  #define      EMFILE         24         /* Too many open files */
36  #define      ENOTTY         25         /* Not a typewriter */
37  #define      ETXTBSY        26         /* Text file busy */
38  #define      EFBIG          27         /* File too large */
39  #define      ENOSPC         28         /* No space left on device */
40  #define      EPIPE         29         /* Illegal seek */
41  #define      EROFS          30         /* Read-only file system */
42  #define      EMLINK         31         /* Too many links */
43  #define      EPIPE         32         /* Broken pipe */
44  #define      EDOM           33         /* Math argument out of domain of func */
45  #define      ERANGE         34         /* Math result not representable */
46
47  #define      EDEADLK        35         /* Resource deadlock would occur */
48  #define      ENAMETOOLONG   36         /* File name too long */
49  #define      ENOLCK         37         /* No record locks available */
50
51  /*
52   * This error code is special: arch syscall entry code will return
53   * -ENOSYS if users try to call a syscall that doesn't exist. To keep
54   * failures of syscalls that really do exist distinguishable from
55   * failures due to attempts to use a nonexistent syscall, syscall
56   * implementations should refrain from returning -ENOSYS.
57   */
58  #define      ENOSYS         38         /* Invalid system call number */

```



```

59 #define ENOTEMPTY 39 /* Directory not empty */
60 #define ELOOP 40 /* Too many symbolic links encountered */
61 #define EWOULDBLOCK EAGAIN /* Operation would block */
62 #define ENMSG 42 /* No message of desired type */
63 #define EIDRM 43 /* Identifier removed */
64 #define ECHRG 44 /* Channel number out of range */
65 #define EL2NSYNC 45 /* Level 2 not synchronized */
66 #define EL3HLT 46 /* Level 3 halted */
67 #define EL3RST 47 /* Level 3 reset */
68 #define ELNRNG 48 /* Link number out of range */
69 #define EUNATCH 49 /* Protocol driver not attached */
70 #define ENOCSI 50 /* No CSI structure available */
71 #define EL2HLT 51 /* Level 2 halted */
72 #define EBADE 52 /* Invalid exchange */
73 #define EBADR 53 /* Invalid request descriptor */
74 #define EXFULL 54 /* Exchange full */
75 #define ENOANO 55 /* No anode */
76 #define EBADRQC 56 /* Invalid request code */
77 #define EBADSLT 57 /* Invalid slot */
78
79 #define EDEADLOCK EDEADLK
80
81 #define EBFONT 59 /* Bad font file format */
82 #define ENOSTR 60 /* Device not a stream */
83 #define ENODATA 61 /* No data available */
84 #define ETIME 62 /* Timer expired */
85 #define ENOSR 63 /* Out of streams resources */
86 #define ENONET 64 /* Machine is not on the network */
87 #define ENOPKG 65 /* Package not installed */
88 #define EREMOTE 66 /* Object is remote */
89 #define ENOLINK 67 /* Link has been severed */
90 #define EADV 68 /* Advertise error */
91 #define ESRMNT 69 /* Srmount error */
92 #define ECOMM 70 /* Communication error on send */
93 #define EPROTO 71 /* Protocol error */
94 #define EMULTIHOP 72 /* Multihop attempted */
95 #define EDOTDOT 73 /* RFS specific error */
96 #define EBADMSG 74 /* Not a data message */
97 #define EOVERFLOW 75 /* Value too large for defined data type */
98 #define ENOTUNIQ 76 /* Name not unique on network */
99 #define EBADFD 77 /* File descriptor in bad state */
100 #define EREMCHG 78 /* Remote address changed */
101 #define ELIBACC 79 /* Can not access a needed shared library */
102 #define ELIBBAD 80 /* Accessing a corrupted shared library */
103 #define ELIBSCN 81 /* .lib section in a.out corrupted */
104 #define ELIBMAX 82 /* Attempting to link in too many shared libraries */
105 #define ELIBEXEC 83 /* Cannot exec a shared library directly */
106 #define EILSEQ 84 /* Illegal byte sequence */
107 #define ERESTART 85 /* Interrupted system call should be restarted */
108 #define ESTRPIPE 86 /* Streams pipe error */
109 #define EUSERS 87 /* Too many users */
110 #define ENOTSOCK 88 /* Socket operation on non-socket */
111 #define EDESTADDRREQ 89 /* Destination address required */
112 #define EMSGSIZE 90 /* Message too long */
113 #define EPROTOTYPE 91 /* Protocol wrong type for socket */
114 #define ENOPROTOPT 92 /* Protocol not available */
115 #define EPROTONOSUPPORT 93 /* Protocol not supported */
116 #define ESOCKTNOSUPPORT 94 /* Socket type not supported */

```

```

117 #define EOPNOTSUPP 95 /* Operation not supported on transport endpoint */
118 #define EPFNOSUPPORT 96 /* Protocol family not supported */
119 #define EAFNOSUPPORT 97 /* Address family not supported by protocol */
120 #define EADDRINUSE 98 /* Address already in use */
121 #define EADDRNOTAVAIL 99 /* Cannot assign requested address */
122 #define ENETDOWN 100 /* Network is down */
123 #define ENETUNREACH 101 /* Network is unreachable */
124 #define ENETRESET 102 /* Network dropped connection because of reset */
125 #define ECONNABORTED 103 /* Software caused connection abort */
126 #define ECONNRESET 104 /* Connection reset by peer */
127 #define ENOBUFS 105 /* No buffer space available */
128 #define EISCONN 106 /* Transport endpoint is already connected */
129 #define ENOTCONN 107 /* Transport endpoint is not connected */
130 #define ESHUTDOWN 108 /* Cannot send after transport endpoint shutdown */
131 #define ETOOMANYREFS 109 /* Too many references: cannot splice */
132 #define ETIMEDOUT 110 /* Connection timed out */
133 #define ECONNREFUSED 111 /* Connection refused */
134 #define EHOSTDOWN 112 /* Host is down */
135 #define EHOSTUNREACH 113 /* No route to host */
136 #define EALREADY 114 /* Operation already in progress */
137 #define EINPROGRESS 115 /* Operation now in progress */
138 #define ESTALE 116 /* Stale file handle */
139 #define EUCLEAN 117 /* Structure needs cleaning */
140 #define ENOTNAM 118 /* Not a XENIX named type file */
141 #define ENAVAIL 119 /* No XENIX semaphores available */
142 #define EISNAM 120 /* Is a named type file */
143 #define EREMOTEIO 121 /* Remote I/O error */
144 #define EDQUOT 122 /* Quota exceeded */
145
146 #define ENOMEDIUM 123 /* No medium found */
147 #define EMEDIUMTYPE 124 /* Wrong medium type */
148 #define ECANCELED 125 /* Operation Canceled */
149 #define ENOKEY 126 /* Required key not available */
150 #define EKEYEXPIRED 127 /* Key has expired */
151 #define EKEYREVOKED 128 /* Key has been revoked */
152 #define EKEYREJECTED 129 /* Key was rejected by service */
153
154 /* for robust mutexes */
155 #define EOWNERDEAD 130 /* Owner died */
156 #define ENOTRECOVERABLE 131 /* State not recoverable */
157
158 #define ERFKILL 132 /* Operation not possible due to RF-kill */
159
160 #define EHWPOISON 133 /* Memory page has hardware error */

```

9.2.1.2 IS_ERR, PTR_ERR, ERR_PTR

相关定义如下:

```

1 /*
2 Location:
3
4     include/linux/err.h
5
6 Description:
7     Kernel pointers have redundant information, so we can use a
8     scheme where we can return either an error code or a normal
9     pointer with the same return value.

```

```

10
11     This should be a per-architecture thing, to allow different
12     error and pointer decisions.
13
14     */
15     #define MAX_ERRNO      4095
16
17     #ifndef __ASSEMBLY__
18
19     #define IS_ERR_VALUE(x) unlikely((x) >= (unsigned long)-MAX_ERRNO)
20
21     static inline void * __must_check ERR_PTR(long error)
22     {
23         return (void *) error;
24     }
25
26     static inline long __must_check PTR_ERR(__force const void *ptr)
27     {
28         return (long) ptr;
29     }
30
31     static inline bool __must_check IS_ERR(__force const void *ptr)
32     {
33         return IS_ERR_VALUE((unsigned long)ptr);
34     }
35
36     static inline bool __must_check IS_ERR_OR_NULL(__force const void *ptr)
37     {
38         return !ptr || IS_ERR_VALUE((unsigned long)ptr);
39     }

```

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

而所谓的错误指针就是指其已经到达了最后一个 page,即内核用最后一页捕捉错误。比如对于 32bit 的系统来说,内核空间最高地址 0xffffffff,那么最后一个 page 就是指的 0xffff000 0xffffffff(假设 4k 一个 page),这段地址是被保留的。内核空间为什么留出最后一个 page? 我们知道一个 page 可能是 4k,也可能是更多,比如 8k,但至少它也是 4k,留出一个 page 出来就可以让我们把内核空间的指针来记录错误了。内核返回的指针一般是指向页面的边界(4k 边界),即 `ptr & 0xfff == 0`。如果你发现你的一个指针指向这个范围中的某个地址,那么你的代码肯定出错了。IS_ERR() 就是判断指针是否有错,如果指针并不是指向最后一个 page,那么没有问题;如果指针指向了最后一个 page,那么说明实际上这不是一个有效的指针,这个指针里保存的实际上是一种错误代码。而通常很常用的方法就是先用 IS_ERR() 来判断是否是错误,然后如果是,那么就调用 PTR_ERR() 来返回这个错误代码。因此,判断一个指针是不是有效的,我们可以调用宏 IS_ERR_VALUE,即判断指针是不是在 (0xffff000, 0xffffffff) 之间,因此,可以用 IS_ERR() 来判断内核函数的返回值是不是一个有效的指针。注意这里用 unlikely() 的用意!

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

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

9.2.2 调试函数

9.2.2.1 BUG_ON

```
1     #define BUG_ON(condition) do { /
2         if (unlikely((condition)!=0)) /
3             BUG(); /
4     } while(0)
```

如果在程序的执行过程中, 觉得该 condition 下是一个 BUG, 可以添加此调试信息, 查看对应堆栈内容。

9.2.2.2 WARN_ON

而 WARN_ON 则是调用 dump_stack, 打印堆栈信息, 不会 OOPS

```
1     #define WARN_ON(condition) do { /
2         if (unlikely((condition)!=0)) { /
3             printk("Badness in %s at %s:%d/n", __FUNCTION__, __FILE__, __LINE__); /
4             dump_stack(); /
5         } /
6     } while (0)
```

9.3 C 语言

9.3.1 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    }
```

```
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。

9.3.2 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。

9.4 GCC

9.4.1 __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)。

9.4.1.1 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)));
```

9.4.1.2 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。

9.4.1.3 设置函数属性

9.4.2 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。

9.5 Sparse

9.5.1 __bitwise

```

1 | #define __bitwise      __attribute__((bitwise))

```

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

```

1 | #ifdef __CHECKER__
2 | #define __bitwise__ __attribute__((bitwise))
3 | #else
4 | #define __bitwise__
5 | #endif
6 | #ifdef __CHECK_ENDIAN__
7 | #define __bitwise __bitwise__
8 | #else
9 | #define __bitwise
10 | #endif
11 |
12 | typedef __u16 __bitwise __le16;
13 | typedef __u16 __bitwise __be16;
14 | typedef __u32 __bitwise __le32;
15 | typedef __u32 __bitwise __be32;
16 | typedef __u64 __bitwise __le64;
17 | typedef __u64 __bitwise __be64;

```

9.6 操作系统

9.6.1 RCU

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

文件系统中, 经常需要查找定位目录, 而对目录的修改相对来说并不多, 这就是 RCU 发挥作用的最佳场景。

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

在 RCU 的实现过程中, 我们主要解决以下问题:

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

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

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

9.7 CPU

9.8 存储系统

9.8.1 字节序

在计算机中内存存储一般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。

9.8.2 缓存 Cache

9.8.2.1 __read_mostly

我们经常需要被读取的数据定义为 `__read_mostly` 类型, 这样 Linux 内核被加载时, 该数据将自动被存放到 Cache 中, 以提高整个系统的执行效率。另一方面, 如果所在的平台没有 Cache, 或者虽然有 Cache, 但并不提供存放数据的接口 (也就是并不允许人工放置数据在 Cache 中), 这样定义为 `__read_mostly` 类型的数据将不能存放在 Linux 内

核中，甚至也不能够被加载到系统内存去执行，将造成 Linux 内核启动失败。??? 需要再仔细去思考。