

面试官：换人！他连 TCP 这几个参数都不懂

[小林coding](#)

每日一句英语学习，每天进步一点点：

前言

TCP 性能的提升不仅考察 TCP 的理论知识，还考察了对于操作系统提供的内核参数的理解与应用。

TCP 协议是由操作系统实现，所以操作系统提供了不少调节 TCP 的参数。

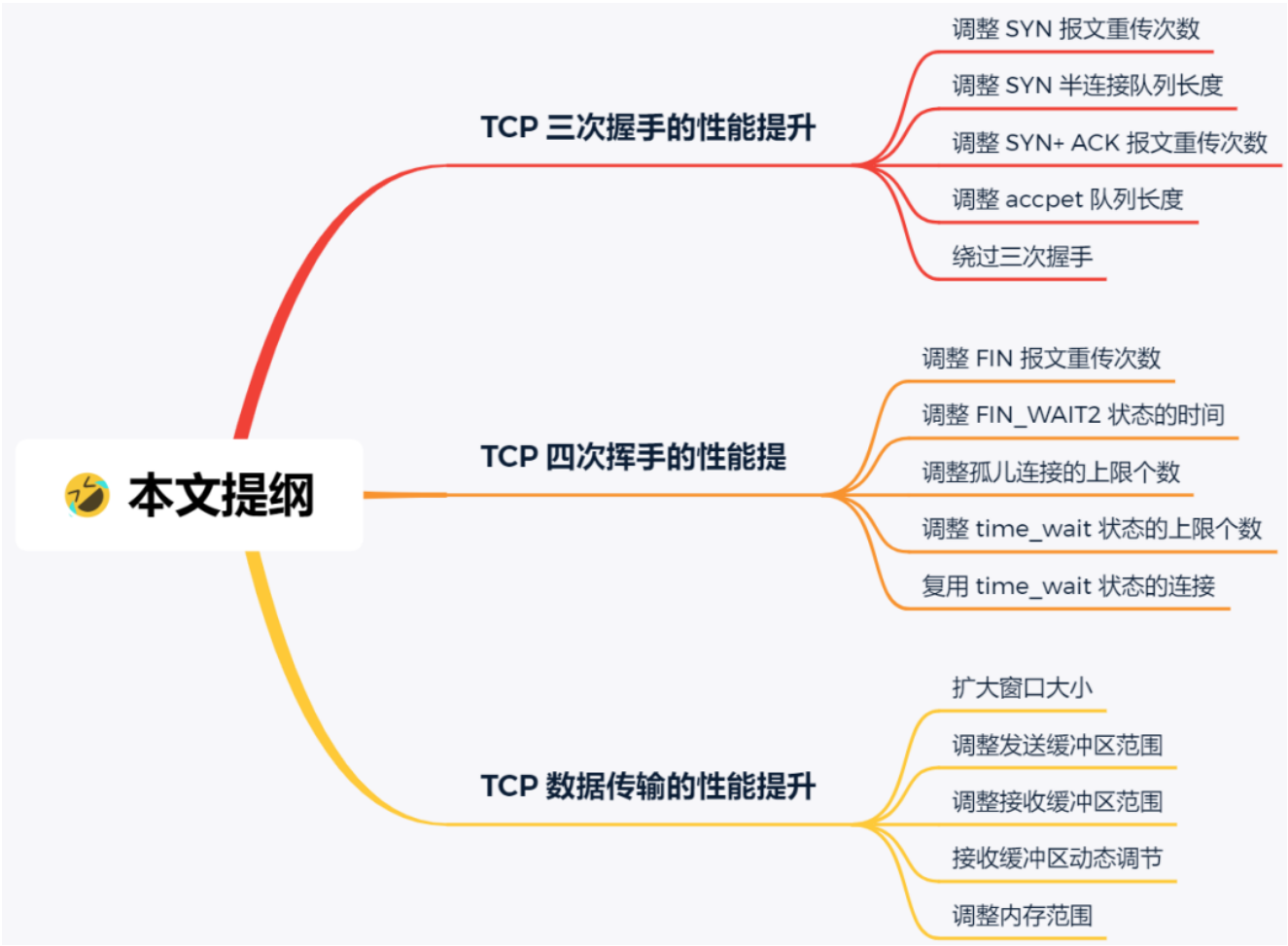
```
[root@lincoding ipv4]# ls -l /proc/sys/net/ipv4/tcp*
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_abort_on_overflow
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_adv_win_scale
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_allowed_congestion_control
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_app_win
-r--r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_available_congestion_control
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_base_mss
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_challenge_ack_limit
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_congestion_control
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_dma_copybreak
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_dsack
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_ecn
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_fack
-rw-r--r-- 1 root root 0 May 27 23:17 /proc/sys/net/ipv4/tcp_fin_timeout
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_frto
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_frto_response
-rw-r--r-- 1 root root 0 May 27 23:17 /proc/sys/net/ipv4/tcp_keepalive_intvl
-rw-r--r-- 1 root root 0 May 27 23:17 /proc/sys/net/ipv4/tcp_keepalive_probes
-rw-r--r-- 1 root root 0 May 27 23:17 /proc/sys/net/ipv4/tcp_keepalive_time
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_limit_output_bytes
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_low_latency
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_max_orphans
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_max_ssthresh
-rw-r--r-- 1 root root 0 May 27 23:17 /proc/sys/net/ipv4/tcp_max_syn_backlog
-rw-r--r-- 1 root root 0 May 27 23:17 /proc/sys/net/ipv4/tcp_max_tw_buckets
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_mem
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_min_tso_segs
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_moderate_rcvbuf
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_mtu_probing
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_no_metrics_save
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_orphan_retries
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_reordering
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_retrans_collapse
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_retries1
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_retries2
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_rfc1337
-rw-r--r-- 1 root root 0 May 27 23:17 /proc/sys/net/ipv4/tcp_rmem
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_sack
```

Linux TCP 参数

如何正确有效的使用这些参数，来提高 TCP 性能是一个不那么简单事情。我们需要针对 TCP 每个阶段的问题来对症下药，而不是病急乱投医。

接下来，将以三个角度来阐述提升 TCP 的策略，分别是：

- TCP 三次握手的性能提升；
- TCP 四次挥手的性能提升；
- TCP 数据传输的性能提升；

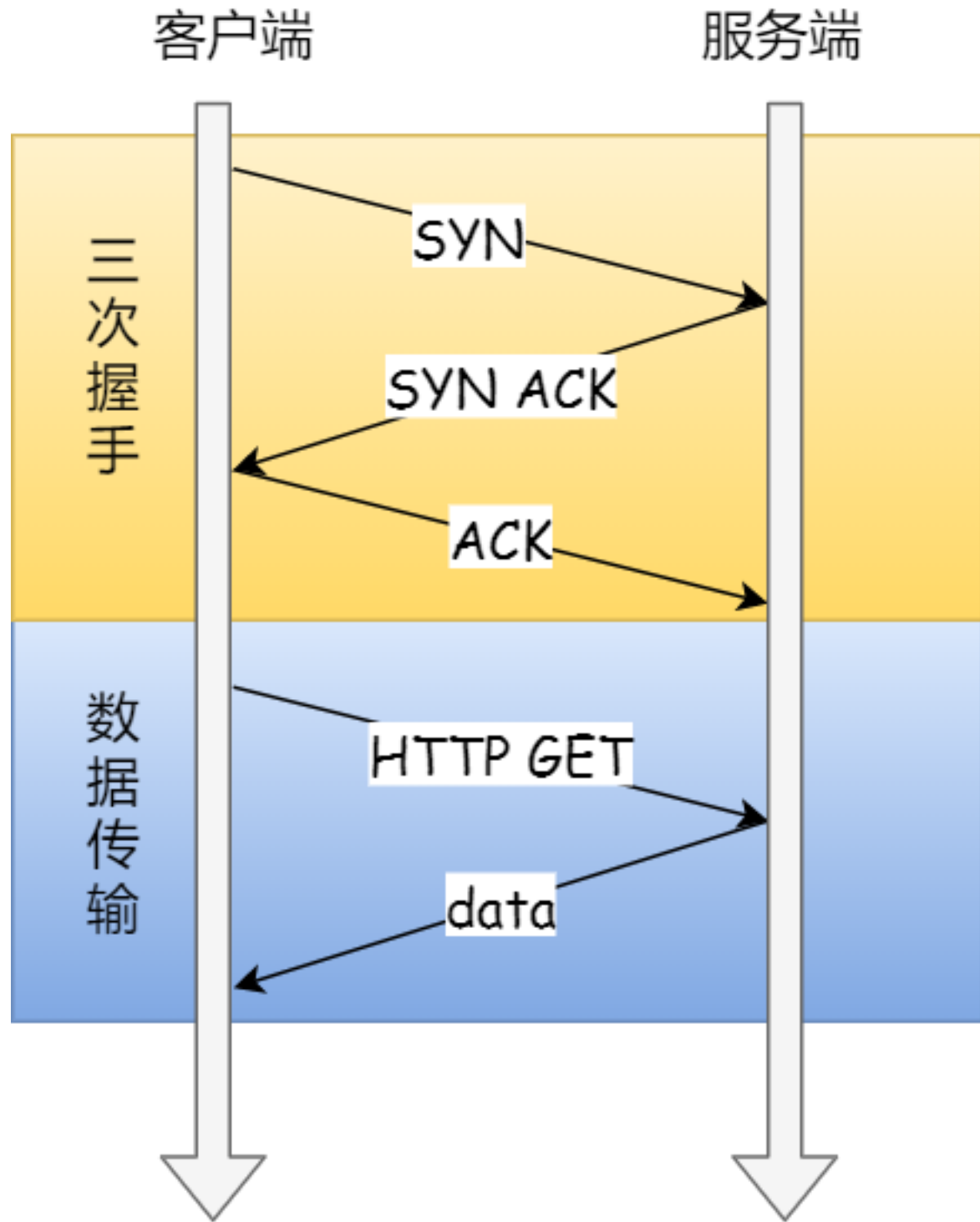


本节提纲

正文

01 TCP 三次握手的性能提升

TCP 是面向连接的、可靠的、双向传输的传输层通信协议，所以在传输数据之前需要经过三次握手才能建立连接。



三次握手与数据传输

那么，三次握手的过程在一个 HTTP 请求的平均时间占比 10% 以上，在网络状态不佳、高并发或者遭遇 SYN 攻击等场景中，如果不能有效正确的调节三次握手中的参数，就会对性能产生很多的影响。

如何正确有效的使用这些参数，来提高 TCP 三次握手的性能，这就需要理解「三次握手的状态变迁」，这样当出现问题时，先用 `netstat` 命令查看是哪个握手阶段出现了

问题，再来对症下药，而不是病急乱投医。

TCP 三次握手的状态变迁

客户端和服务端都可以针对三次握手优化性能。主动发起连接的客户端优化相对简单些，而服务端需要监听端口，属于被动连接方，其间保持许多的中间状态，优化方法相对复杂一些。

所以，客户端（主动发起连接方）和服务端（被动连接方）优化的方式是不同的，接下来分别针对客户端和服务端优化。

客户端优化

三次握手建立连接的首要目的是「同步序列号」。

只有同步了序列号才有可靠传输，TCP 许多特性都依赖于序列号实现，比如流量控制、丢包重传等，这也是三次握手中的报文称为 SYN 的原因，SYN 的全称就叫 *Synchronize Sequence Numbers*（同步序列号）。

TCP 头部

SYN_SENT 状态的优化

客户端作为主动发起连接方，首先它将发送 SYN 包，于是客户端的连接就会处于 SYN_SENT 状态。

客户端在等待服务端回复的 ACK 报文，正常情况下，服务器会在几毫秒内返回 SYN+ACK，但如果客户端长时间没有收到 SYN+ACK 报文，则会重发 SYN 包，重发的次

数由 `tcp_syn_retries` 参数控制，默认是 5 次：

```
# tcp_syn_retries 控制 SYN 包重传的次数，默认值是 5 次
$ echo 5 > /proc/sys/net/ipv4/tcp_syn_retries
```

通常，第一次超时重传是在 1 秒后，第二次超时重传是在 2 秒，第三次超时重传是在 4 秒后，第四次超时重传是在 8 秒后，第五次是在超时重传 16 秒后。没错，每次超时的时间是上一次的 2 倍。

当第五次超时重传后，会继续等待 32 秒，如果仍然服务端没有回应 ACK，客户端就会终止三次握手。

所以，总耗时是 $1+2+4+8+16+32=63$ 秒，大约 1 分钟左右。

SYN 超时重传

你可以根据网络的稳定性和目标服务器的繁忙程度修改 SYN 的重传次数，调整客户端的三次握手时间上限。比如内网中通讯时，就可以适当调低重试次数，尽快把错误暴露给应用程序。

服务端优化

当服务端收到 SYN 包后，服务端会立马回复 SYN+ACK 包，表明确认收到了客户端的序列号，同时也把自己的序列号发给对方。

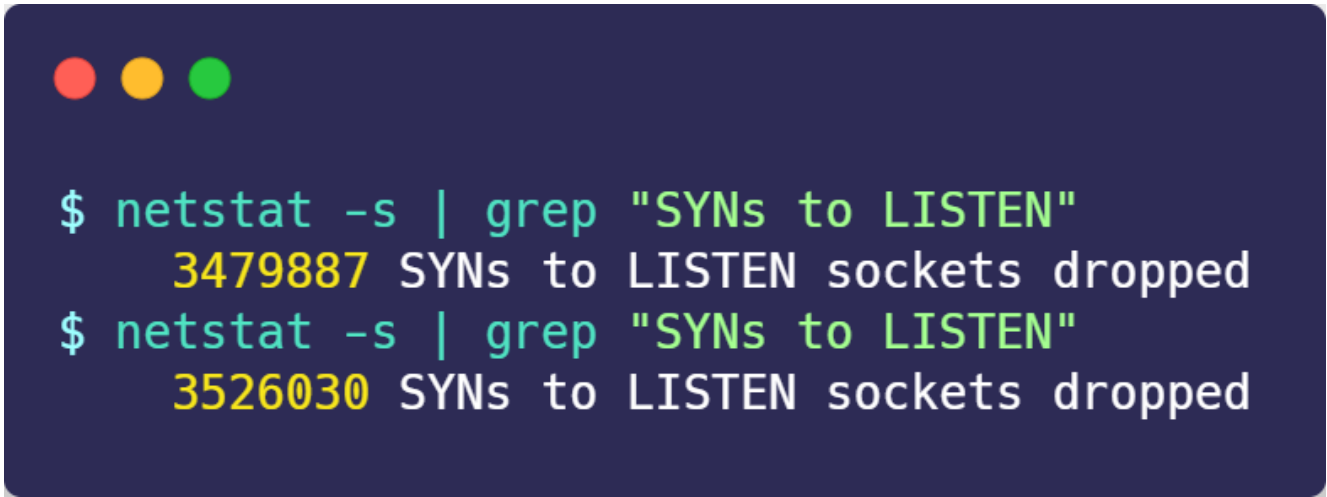
此时，服务端出现了新连接，状态是 `SYN_RCV`。在这个状态下，Linux 内核就会建立一个「半连接队列」来维护「未完成」的握手信息，当半连接队列溢出后，服务端就无法再建立新的连接。

半连接队列与全连接队列

SYN 攻击，攻击的就是就是这个半连接队列。

如何查看由于 SYN 半连接队列已满，而被丢弃连接的情况？

我们可以通过该 `netstat -s` 命令给出的统计结果中，可以得到由于半连接队列已满，引发的失败次数：

A terminal window with a dark blue background and three colored window control buttons (red, yellow, green) in the top left corner. It displays two lines of command output. The first line shows the command `$ netstat -s | grep "SYNs to LISTEN"` followed by the output `3479887 SYNs to LISTEN sockets dropped`. The second line shows the same command followed by the output `3526030 SYNs to LISTEN sockets dropped`.

```
$ netstat -s | grep "SYNs to LISTEN"
3479887 SYNs to LISTEN sockets dropped
$ netstat -s | grep "SYNs to LISTEN"
3526030 SYNs to LISTEN sockets dropped
```

上面输出的数值是累计值，表示共有多少个 TCP 连接因为半连接队列溢出而被丢弃。隔几秒执行几次，如果有上升的趋势，说明当前存在半连接队列溢出的现象。

如何调整 SYN 半连接队列大小？

要想增大半连接队列，不能只单纯增大 `tcp_max_syn_backlog` 的值，还需一同增大 `somaxconn` 和 `backlog`，也就是增大 `accept` 队列。否则，只单纯增

大 `tcp_max_syn_backlog` 是无效的。

增大 `tcp_max_syn_backlog` 和 `somaxconn` 的方法是修改 Linux 内核参数：

```
# 增大 tcp_max_syn_backlog
$ echo 1024 > /proc/sys/net/ipv4/tcp_max_syn_backlog

# 增大 somaxconn
$ echo 1024 > /proc/sys/net/core/somaxconn
```

增大 backlog 的方式，每个 Web 服务都不同，比如 Nginx 增大 backlog 的方法如下：

```
# /usr/local/nginx/conf/nginx.conf
server {
    listen 8088 default backlog=1024;
    server_name localhost;
    ....
}
```

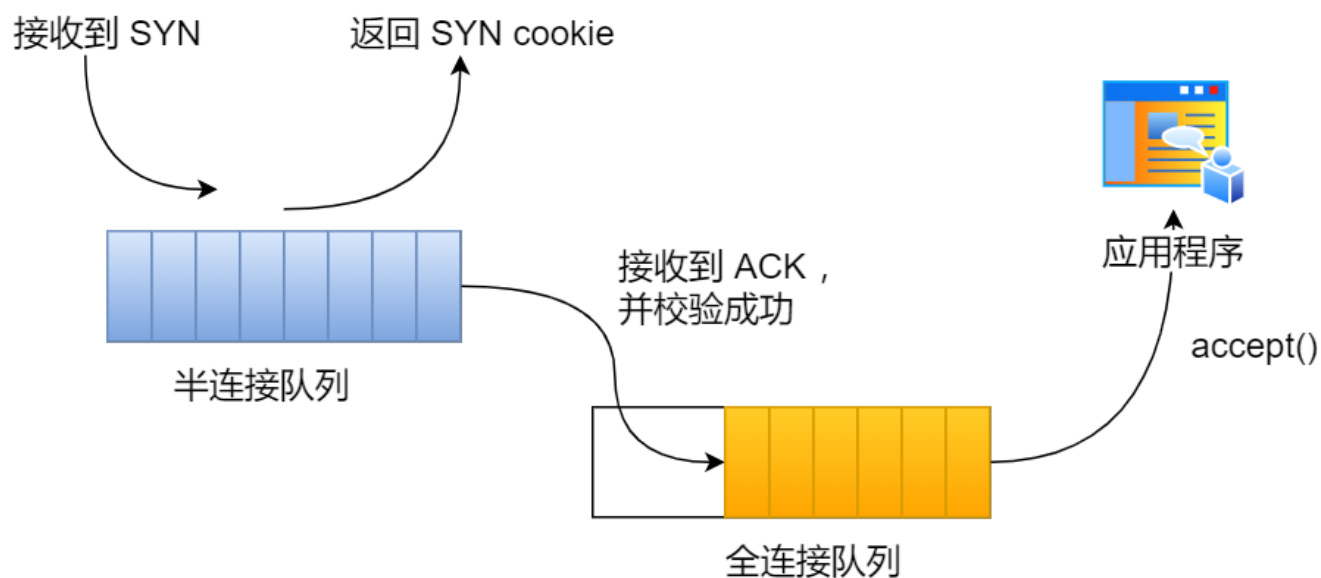
最后，改变了如上这些参数后，要重启 Nginx 服务，因为 SYN 半连接队列和 accept 队列都是在 `listen()` 初始化的。

如果 SYN 半连接队列已满，只能丢弃连接吗？

并不是这样，开启 **syncookies** 功能就可以在不使用 SYN

半连接队列的情况下成功建立连接。

syncookies 的工作原理：服务器根据当前状态计算出一个值，放在己方发出的 SYN+ACK 报文中发出，当客户端返回 ACK 报文时，取出该值验证，如果合法，就认为连接建立成功，如下图所示。



开启 syncookies 功能

syncookies 参数主要有以下三个值：

- 0 值，表示关闭该功能；
- 1 值，表示仅当 SYN 半连接队列放不下时，再启用它；
- 2 值，表示无条件开启功能；

那么在应对 SYN 攻击时，只需要设置为 1 即可：



```
# 开启 tcp_syncookies 功能，默认是开启的
$ echo 1 > /proc/sys/net/ipv4/tcp_syncookies
```


SYN_RCV 状态的优化

当客户端接收到服务器发来的 SYN+ACK 报文后，就会回复 ACK 给服务器，同时客户端连接状态从 SYN_SENT 转换为 ESTABLISHED，表示连接建立成功。

服务器端连接成功建立的时间还要再往后，等到服务端收到客户端的 ACK 后，服务端的连接状态才变为 ESTABLISHED。

如果服务器没有收到 ACK，就会重发 SYN+ACK 报文，同时一直处于 SYN_RCV 状态。

当网络繁忙、不稳定时，报文丢失就会变严重，此时应该调大重发次数。反之则可以调小重发次数。修改重发次数的方法是，调整 **tcp_synack_retries** 参数：

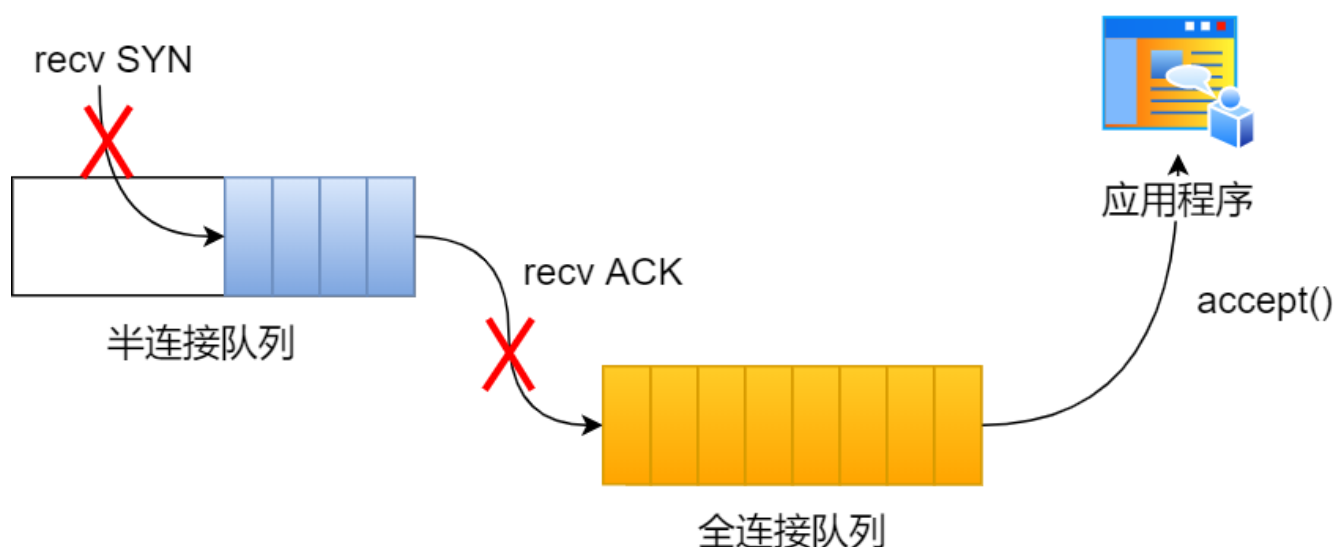


```
# tcp_synack_retries 控制 SYN+ACK 包重传的次数，默认值是 5 次
$ echo 5 > /proc/sys/net/ipv4/tcp_synack_retries
```

tcp_synack_retries 的默认重试次数是 5 次，与客户端重传 SYN 类似，它的重传会经历 1、2、4、8、16 秒，最后一次重传后会继续等待 32 秒，如果服务端仍然没有收到 ACK，才会关闭连接，故共需要等待 63 秒。

服务器收到 ACK 后连接建立成功，此时，内核会把连接从半连接队列移除，然后创建新的完全的连接，并将其添加到 accept 队列，等待进程调用 accept 函数时把连接取出来。

如果进程不能及时地调用 accept 函数，就会造成 accept 队列（也称全连接队列）溢出，最终导致建立好的 TCP 连接被丢弃。



accept 队列溢出

accept 队列已满，只能丢弃连接吗？

丢弃连接只是 Linux 的默认行为，我们还可以选择向客户端发送 RST 复位报文，告诉客户端连接已经建立失败。打开这一功能需要将 `tcp_abort_on_overflow` 参数设置为 1。

```
# 打开后，则当 accpet 队列满了会回 RST，默认值是 0 关闭
$ echo 1 > /proc/sys/net/ipv4/tcp_abort_on_overflow
```

`tcp_abort_on_overflow` 共有两个值分别是 0 和 1，其分别表示：

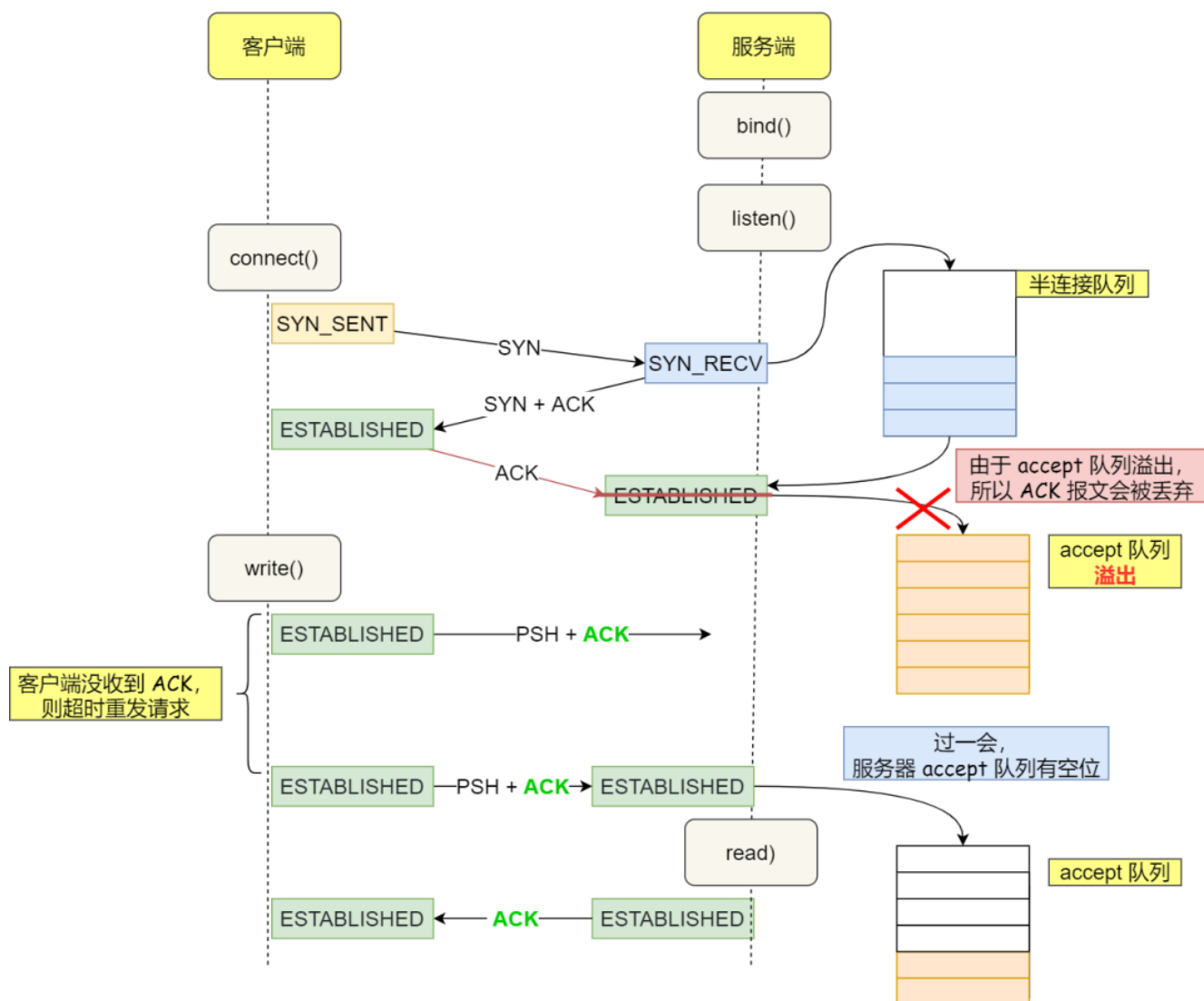
- 0：如果 `accept` 队列满了，那么 server 扔掉 client 发过来的 `ack`；
- 1：如果 `accept` 队列满了，server 发送一个 `RST` 包给 client，表示废掉这个握手过程和这个连接；

如果要知道客户端连接不上服务端，是不是服务端 TCP 全连接队列满的原因，那么可以把 `tcp_abort_on_overflow` 设置为 1，这时如果在客户端异常中可以看到很多 `connection reset by peer` 的错误，那么就可以证明是由于服务端 TCP 全连接队列溢出的问题。

通常情况下，应当把 `tcp_abort_on_overflow` 设置为 0，因为这样更有利于应对突发流量。

举个例子，当 `accept` 队列满导致服务器丢掉了 `ACK`，与此同时，客户端的连接状态却是 `ESTABLISHED`，客户端进程就在建立好的连接上发送请求。只要服务器没有为请求回复 `ACK`，客户端的请求就会被多次「重发」。如果服务器上的进程只是短暂的繁忙造成 **`accept`** 队列满，那么当 **`accept`** 队列有空位时，再次接收到的请求报文由于含有 **`ACK`**，仍然会触发服务器端成功建立连接。

`tcp_abort_on_overflow = 0`



`tcp_abort_on_overflow` 为 0 可以应对突发流量

所以, `tcp_abort_on_overflow` 设为 0 可以提高连接建立的成功率, 只有你非常肯定 TCP 全连接队列会长期溢出时, 才能设置为 1 以尽快通知客户端。

如何调整 `accept` 队列的长度呢?

`accept` 队列的长度取决于 `somaxconn` 和 `backlog` 之间的最小值, 也就是 `min(somaxconn, backlog)`, 其中:

- `somaxconn` 是 Linux 内核的参数, 默认值是 128, 可以通过 `net.core.somaxconn` 来设置其值;
- `backlog` 是 `listen(int sockfd, int backlog)` 函数

中的 backlog 大小；

Tomcat、Nginx、Apache 常见的 Web 服务的 backlog 默认值都是 511。

如何查看服务端进程 accept 队列的长度？

可以通过 `ss -ltn` 命令查看：

- Recv-Q：当前 accept 队列的大小，也就是当前已完成三次握手并等待服务端 `accept()` 的 TCP 连接；
- Send-Q：accept 队列最大长度，上面的输出结果说明监听 8088 端口的 TCP 服务，accept 队列的最大长度为 128；

如何查看由于 accept 连接队列已满，而被丢弃的连接？

当超过了 accept 连接队列，服务端则会丢掉后续进来的 TCP 连接，丢掉的 TCP 连接的个数会被统计起来，我们可以使用 `netstat -s` 命令来查看：

上面看到的 41150 times，表示 accept 队列溢出的次数，注意这个是累计值。可以隔几秒钟执行下，如果这个数字一直在增加的话，说明 accept 连接队列偶尔满了。

如果持续不断地有连接因为 accept 队列溢出被丢弃，就应该调大 backlog 以及 somaxconn 参数。

如何绕过三次握手？

以上我们只是在对三次握手的过程进行优化，接下来我们看看如何绕过三次握手发送数据。

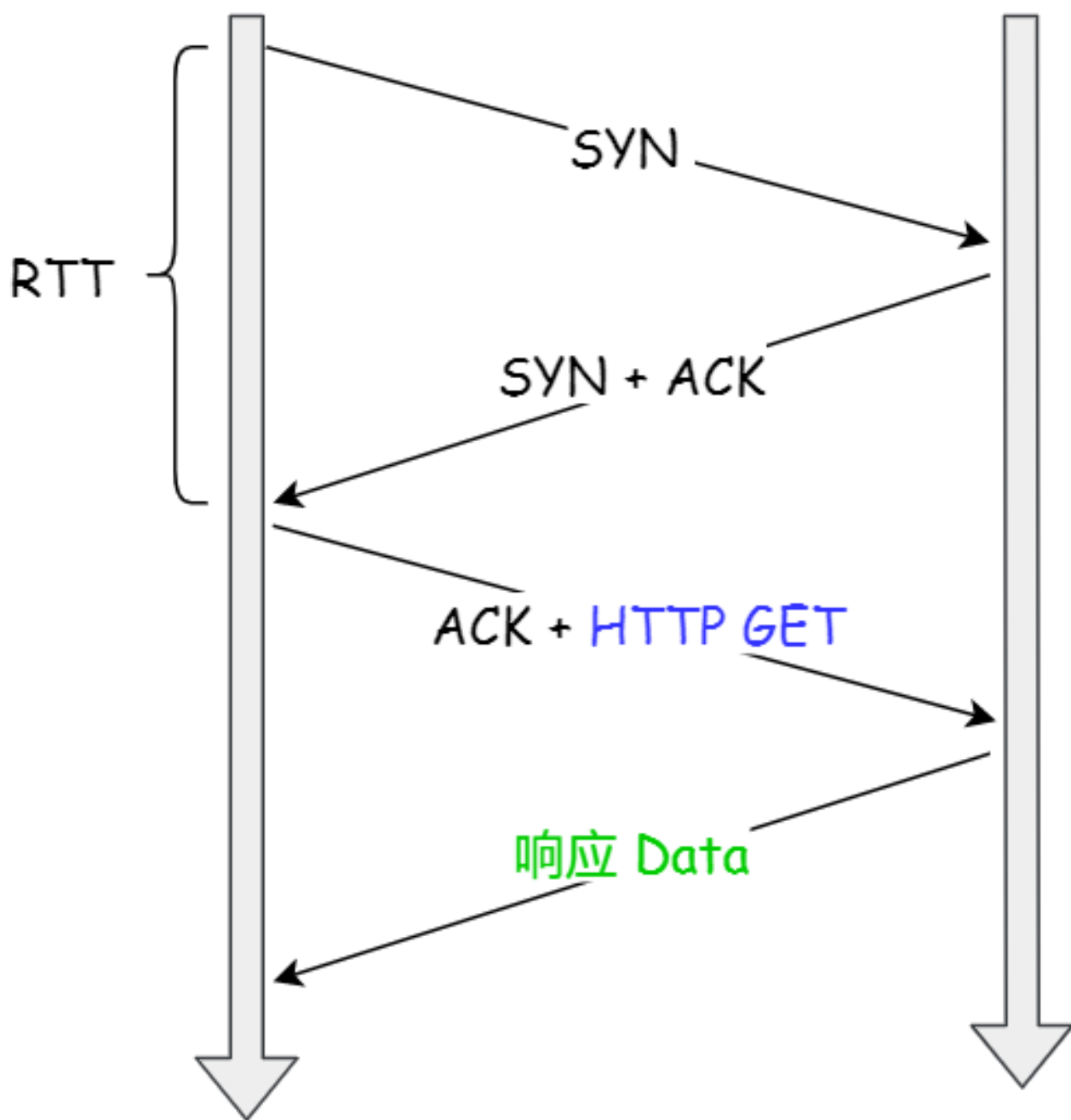
三次握手建立连接造成的后果就是，HTTP 请求必须在一个 RTT（从客户端到服务器一个往返的时间）后才能发送。



客户端



服务端



常规 HTTP 请求

在 Linux 3.7 内核版本之后，提供了 TCP Fast Open 功能，这个功能可以减少 TCP 连接建立的时延。

接下来说说，TCP Fast Open 功能的工作方式。

开启 TCP Fast Open 功能

在客户端首次建立连接时的过程：

1. 客户端发送 SYN 报文，该报文包含 Fast Open 选项，且该选项的 Cookie 为空，这表明客户端请求 Fast Open Cookie；
2. 支持 TCP Fast Open 的服务器生成 Cookie，并将其置于 SYN-ACK 数据包中的 Fast Open 选项以发回客户端；
3. 客户端收到 SYN-ACK 后，本地缓存 Fast Open 选项中的 Cookie。

所以，第一次发起 HTTP GET 请求的时候，还是需要正常的三次握手流程。

之后，如果客户端再次向服务器建立连接时的过程：

1. 客户端发送 SYN 报文，该报文包含「数据」（对于非 TFO 的普通 TCP 握手过程，SYN 报文中不包含「数据」）以及此前记录的 Cookie；
2. 支持 TCP Fast Open 的服务器会对收到 Cookie 进行校验：如果 Cookie 有效，服务器将在 SYN-ACK 报文中对 SYN 和「数据」进行确认，服务器随后将「数据」递送至相应的应用程序；如果 Cookie 无效，服务器将丢弃 SYN 报文中包含的「数据」，且其随后发出的 SYN-ACK 报文将只确认 SYN 的对应序列号；
3. 如果服务器接受了 SYN 报文中的「数据」，服务器可在握手完成之前发送「数据」，这就减少了握手带

来的 1 个 RTT 的时间消耗；

4. 客户端将发送 ACK 确认服务器发回的 SYN 以及「数据」，但如果客户端在初始的 SYN 报文中发送的「数据」没有被确认，则客户端将重新发送「数据」；
5. 此后的 TCP 连接的数据传输过程和非 TFO 的正常情况一致。

所以，之后发起 HTTP GET 请求的时候，可以绕过三次握手，这就减少了握手带来的 1 个 RTT 的时间消耗。

注：客户端在请求并存储了 Fast Open Cookie 之后，可以不断重复 TCP Fast Open 直至服务器认为 Cookie 无效（通常为过期）。

Linux 下怎么打开 TCP Fast Open 功能呢？

在 Linux 系统中，可以通过设置 **tcp_fastopen** 内核参数，来打开 **Fast Open** 功能：

```
# 无论作为客户端还是服务器，都可以使用 Fast Open 功能
$ echo 3 > /proc/sys/net/ipv4/tcp_fastopen
```

tcp_fastopen 各个值的意义：

- 0 关闭
- 1 作为客户端使用 Fast Open 功能

- 2 作为服务端使用 Fast Open 功能
- 3 无论作为客户端还是服务器，都可以使用 Fast Open 功能

TCP Fast Open 功能需要客户端和服务端同时支持，才有效果。

小结

本小结主要介绍了关于优化 TCP 三次握手的几个 TCP 参数。

三次握手优化策略

客户端的优化

当客户端发起 SYN 包时，可以通过 `tcp_syn_retries` 控制其重传的次数。

服务端的优化

当服务端 SYN 半连接队列溢出后，会导致后续连接被丢弃，可以通过 `netstat -s` 观察半连接队列溢出的情况，如果 SYN 半连接队列溢出情况比较严重，可以通过 `tcp_max_syn_backlog`、`somaxconn`、`backlog` 参数来调整 SYN 半连接队列的大小。

服务端回复 SYN+ACK 的重传次数由 `tcp_synack_retries` 参数控制。如果遭受 SYN 攻击，应把 `tcp_syncookies` 参数设置为 1，表示仅在 SYN 队列满后开启 `syncookie` 功能，可以保证正常的连接成功建立。

服务端收到客户端返回的 ACK，会把连接移入 `accpet` 队列，等待进行调用 `accpet()` 函数取出连接。

可以通过 `ss -lnt` 查看服务端进程的 `accept` 队列长度，如果 `accept` 队列溢出，系统默认丢弃 ACK，如果可以把 `tcp_abort_on_overflow` 设置为 1，表示用 RST 通知客户端连接建立失败。

如果 `accpet` 队列溢出严重，可以通过 `listen` 函数的 `backlog` 参数和 `somaxconn` 系统参数提高队列大小，`accept` 队列长度取决于 `min(backlog, somaxconn)`。

绕过三次握手

TCP Fast Open 功能可以绕过三次握手，使得 HTTP 请求减少了 1 个 RTT 的时间，Linux 下可以通过 `tcp_fastopen` 开启该功能，同时必须保证服务端和客户端同时支持。

02 TCP 四次挥手的性能提升

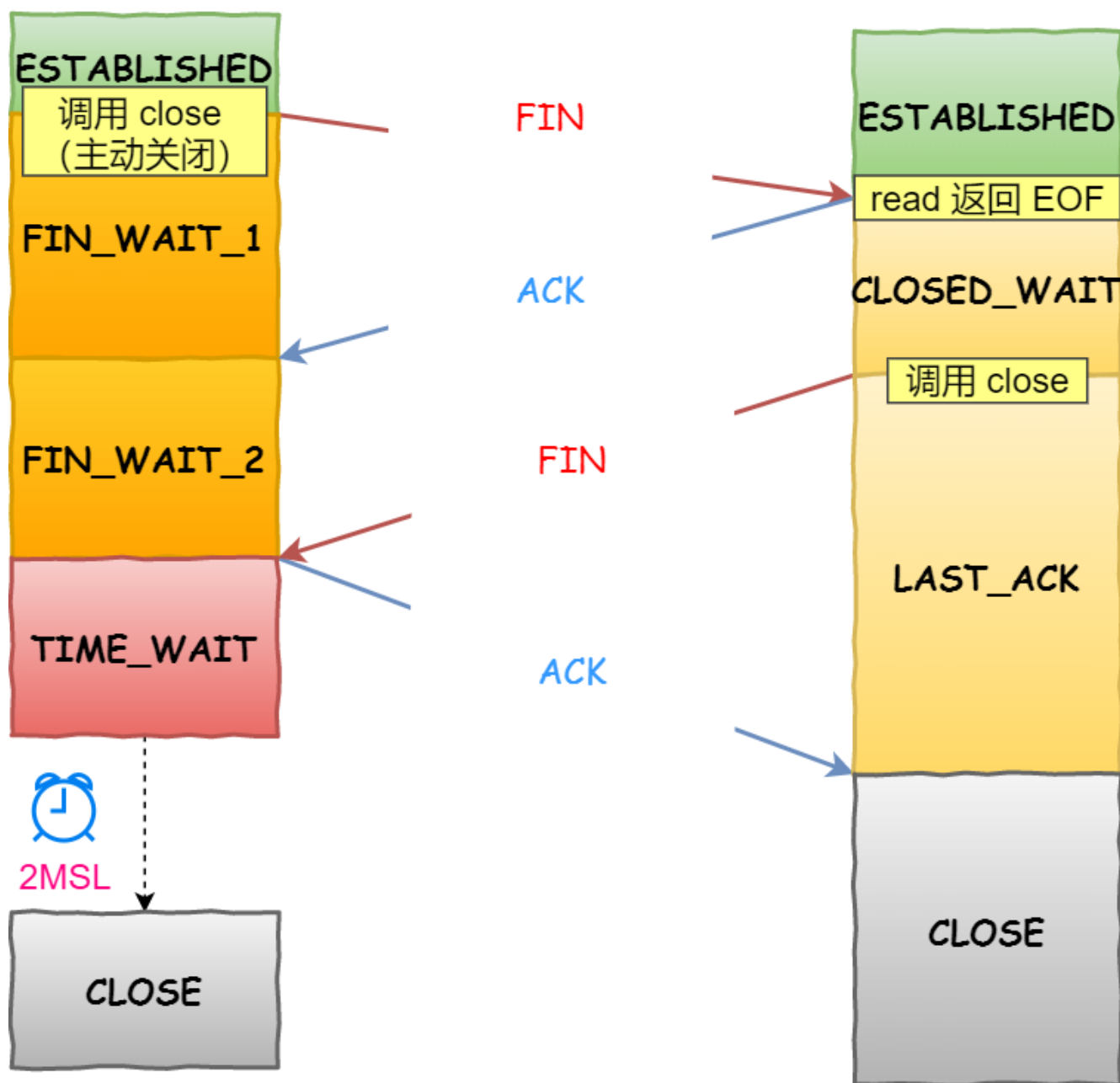
接下来，我们一起来看看针对 TCP 四次挥手关不连接时，如何优化性能。

在开始之前，我们得先了解四次挥手状态变迁的过程。

客户端和服务端双方都可以主动断开连接，通常先关闭连接的一方称为主动方，后关闭连接的一方称为被动方。

客户端

服务端



客户端主动关闭

可以看到，四次挥手过程只涉及了两种报文，分别是 **FIN** 和 **ACK**：

- FIN 就是结束连接的意思，谁发出 FIN 报文，就表示它将不会再发送任何数据，关闭这一方向上的传输通道；
- ACK 就是确认的意思，用来通知对方：你方的发送通道已经关闭；

四次挥手的过程：

- 当主动方关闭连接时，会发送 FIN 报文，此时发送方的 TCP 连接将从 ESTABLISHED 变成 FIN_WAIT1。
- 当被动方收到 FIN 报文后，内核会自动回复 ACK 报文，连接状态将从 ESTABLISHED 变成 CLOSE_WAIT，表示被动方在等待进程调用 close 函数关闭连接。
- 当主动方收到这个 ACK 后，连接状态由 FIN_WAIT1 变为 FIN_WAIT2，也就是表示主动方的发送通道就关闭了。
- 当被动方进入 CLOSE_WAIT 时，被动方还会继续处理数据，等到进程的 read 函数返回 0 后，应用程序就会调用 close 函数，进而触发内核发送 FIN 报文，此时被动方的连接状态变为 LAST_ACK。
- 当主动方收到这个 FIN 报文后，内核会回复 ACK 报文给被动方，同时主动方的连接状态由 FIN_WAIT2 变为 TIME_WAIT，在 Linux 系统下大约等待 1 分钟后，TIME_WAIT 状态的连接才会彻底关闭。
- 当被动方收到最后的 ACK 报文后，被动方的连接就会关闭。

你可以看到，每个方向都需要一个 **FIN** 和一个 **ACK**，因此通常被称为四次挥手。

这里一点需要注意的是：主动关闭连接的，才有 **TIME_WAIT** 状态。

主动关闭方和被动关闭方优化的思路也不同，接下来分别说说如何优化他们。

主动方的优化

关闭的连接的方式通常有两种，分别是 RST 报文关闭和 FIN 报文关闭。


如果进程异常退出了，内核就会发送 RST 报文来关闭，它可以不走四次挥手流程，是一个暴力关闭连接的方式。

安全关闭连接的方式必须通过四次挥手，它由进程调用 `close` 和 `shutdown` 函数发起 FIN 报文（`shutdown` 参数须传入 `SHUT_WR` 或者 `SHUT_RDWR` 才会发送 FIN）。

调用 `close` 函数 和 `shutdown` 函数有什么区别？

调用了 `close` 函数意味着完全断开连接，完全断开不仅指无法传输数据，而且也不能发送数据。此时，调用了 `close` 函数的一方的连接叫做「孤儿连接」，如果你用 `netstat -p` 命令，会发现连接对应的进程名为空。

使用 `close` 函数关闭连接是不优雅的。于是，就出现了一种优雅关闭连接的 `shutdown` 函数，它可以控制只关闭一个方向的连接：



```
int shutdown(int sock, int howto);
```

第二个参数决定断开连接的方式，主要有以下三种方式：

- **SHUT_RD(0)**：关闭连接的「读」这个方向，如果接收缓冲区有已接收的数据，则将会被丢弃，并且后续再收到新的数据，会对数据进行 ACK，然后悄悄地丢弃。也就是说，对端还是会接收到 ACK，在这种情况下根本不知道数据已经被丢弃了。
- **SHUT_WR(1)**：关闭连接的「写」这个方向，这就是常被称为「半关闭」的连接。如果发送缓冲区还有未发送的数据，将被立即发送出去，并发送一个 FIN 报文给对端。
- **SHUT_RDWR(2)**：相当于 SHUT_RD 和 SHUT_WR 操作各一次，关闭套接字的读和写两个方向。

close 和 shutdown 函数都可以关闭连接，但这两种方式关闭的连接，不只功能上有差异，控制它们的 Linux 参数也不相同。

FIN_WAIT1 状态的优化

主动方发送 FIN 报文后，连接就处于 FIN_WAIT1 状态，正常情况下，如果能及时收到被动方的 ACK，则会很快变为 FIN_WAIT2 状态。

但是当迟迟收不到对方返回的 ACK 时，连接就会一直处于 FIN_WAIT1 状态。此时，内核会定时重发 **FIN** 报文，其中重发次数由 **tcp_orphan_retries** 参数控制（注意，orphan 虽然是孤儿的意思，该参数却不只对孤儿连接有效，事实上，它对所有 FIN_WAIT1 状态下的连接都有

效)，默认值是 0。

```
# 调整 FIN 报文重传次数为 5 次，默认值是 0，特指 8 次  
$ echo 5 > /proc/sys/net/ipv4/tcp_orphan_retries
```

你可能会好奇，这 0 表示几次？实际上当为 0 时，特指 8 次，从下面的内核源码可知：

```
/* Calculate maximal number of retries on an orphaned socket. */  
static int tcp_orphan_retries(struct sock *sk, int alive)  
{  
    int retries = sysctl_tcp_orphan_retries; /* May be zero. */  
  
    /* We know from an ICMP that something is wrong. */  
    if (sk->sk_err_soft && !alive)  
        retries = 0;  
  
    /* However, if socket sent something recently, select some safe  
     * number of retries. 8 corresponds to >100 seconds with minimal  
     * RTO of 200msec. */  
    if (retries == 0 && alive)  
        retries = 8;  
    return retries;  
}
```

如果 FIN_WAIT1 状态连接很多，我们就需要考虑降低 tcp_orphan_retries 的值，当重传次数超过 tcp_orphan_retries 时，连接就会直接关闭掉。

对于普遍正常情况时，调低 tcp_orphan_retries 就已经可以了。如果遇到恶意攻击，FIN 报文根本无法发送出去，这由 TCP 两个特性导致的：

- 首先，TCP 必须保证报文是有序发送的，FIN 报文也不例外，当发送缓冲区还有数据没有发送时，FIN 报

文也不能提前发送。

- 其次，TCP 有流量控制功能，当接收方接收窗口为 0 时，发送方就不能再发送数据。所以，当攻击者下载大文件时，就可以通过接收窗口设为 0，这就会使得 FIN 报文都无法发送出去，那么连接会一直处于 FIN_WAIT1 状态。

解决这种问题的方法，是调整 **tcp_max_orphans** 参数，它定义了「孤儿连接」的最大数量：

```
# 调整孤儿连接最大个数
$ echo 16384 > /proc/sys/net/ipv4/tcp_max_orphans
```

当进程调用了 `close` 函数关闭连接，此时连接就会是「孤儿连接」，因为它无法在发送和接收数据。Linux 系统为了防止孤儿连接过多，导致系统资源长时间被占用，就提供了 `tcp_max_orphans` 参数。如果孤儿连接数量大于它，新增的孤儿连接将不再走四次挥手，而是直接发送 RST 复位报文强制关闭。

FIN_WAIT2 状态的优化

当主动方收到 ACK 报文后，会处于 FIN_WAIT2 状态，就表示主动方的发送通道已经关闭，接下来将等待对方发送 FIN 报文，关闭对方的发送通道。

这时，如果连接是用 `shutdown` 函数关闭的，连接可以一直处于 FIN_WAIT2 状态，因为它可能还可以发送或接收

数据。但对于 **close** 函数关闭的孤儿连接，由于无法在发送和接收数据，所以这个状态不可以持续太久，而 **tcp_fin_timeout** 控制了这个状态下连接的持续时长，默认值是 60 秒：



```
# 调整孤儿连接 FIN_WAIT2 状态的持续时间，默认值是 60
$ echo 60 > /proc/sys/net/ipv4/tcp_fin_timeout
```

它意味着对于孤儿连接（调用 **close** 关闭的连接），如果在 60 秒后还没有收到 **FIN** 报文，连接就会直接关闭。

这个 60 秒不是随便决定的，它与 **TIME_WAIT** 状态持续的时间是相同的，后面我们在来说说是为什么是 60 秒。

TIME_WAIT 状态的优化

TIME_WAIT 是主动方四次挥手的最后一个状态，也是最常见的状态。

当收到被动方发来的 **FIN** 报文后，主动方会立刻回复 **ACK**，表示确认对方的发送通道已经关闭，接着就处于 **TIME_WAIT** 状态。在 Linux 系统，**TIME_WAIT** 状态会持续 60 秒后才会进入关闭状态。

TIME_WAIT 状态的连接，在主动方看来确实快已经关闭了。然后，被动方没有收到 **ACK** 报文前，还是处于 **LAST_ACK** 状态。如果这个 **ACK** 报文没有到达被动方，被动方就会重发 **FIN** 报文。重发次数仍然由前面介绍过的 **tcp_orphan_retries** 参数控制。

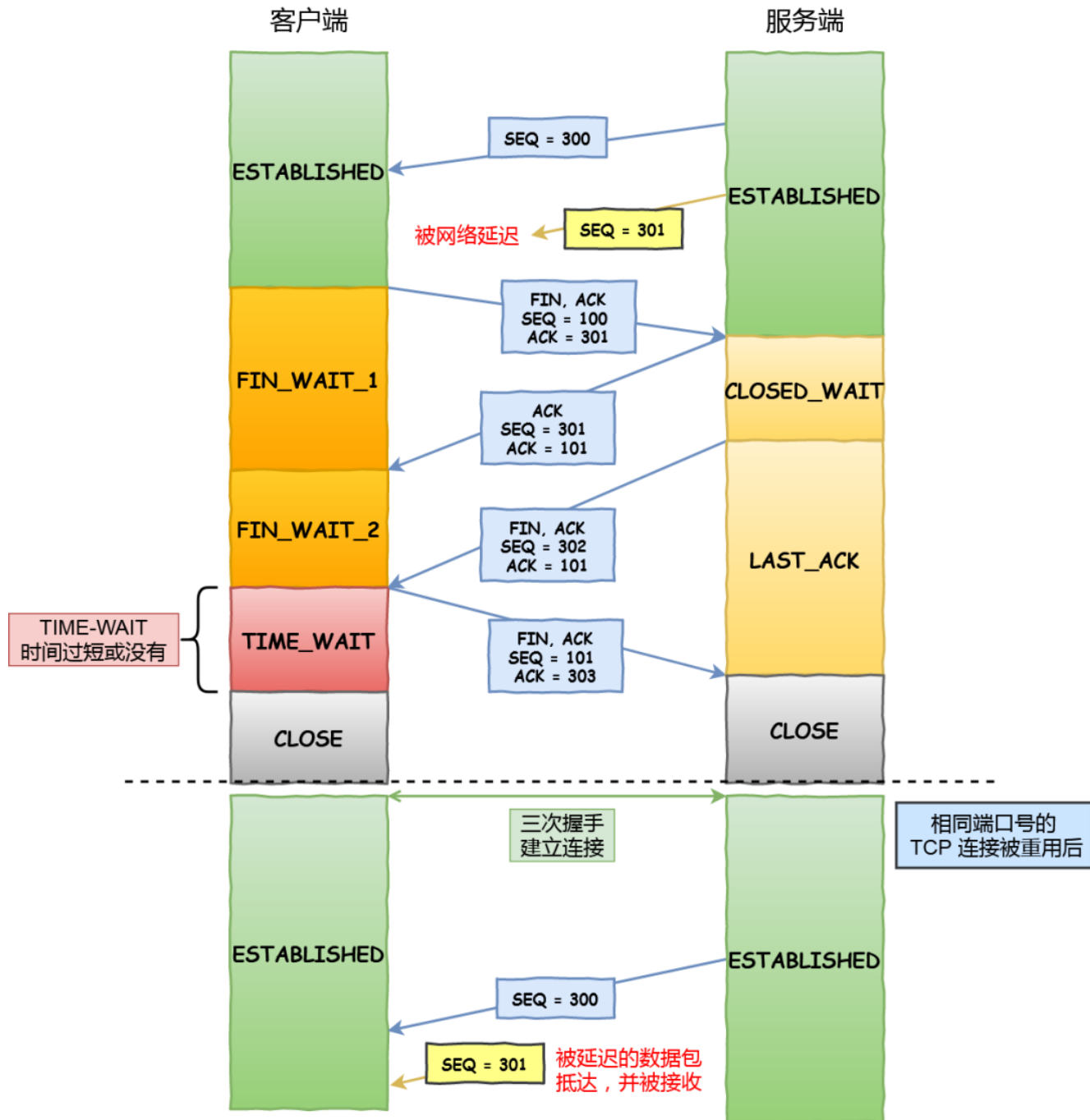
TIME-WAIT 的状态尤其重要，主要是两个原因：

- 防止具有相同「四元组」的「旧」数据包被收到；
- 保证「被动关闭连接」的一方能被正确的关闭，即保证最后的 ACK 能让被动关闭方接收，从而帮助其正常关闭；

原因一：防止旧连接的数据包

TIME-WAIT 的一个作用是防止收到历史数据，从而导致数据错乱的问题。

假设 TIME-WAIT 没有等待时间或时间过短，被延迟的数据包抵达后会发生什么呢？



接收到历史数据的异常

- 如上图黄色框框服务端在关闭连接之前发送的 $SEQ = 301$ 报文，被网络延迟了。
- 这时有相同端口的 TCP 连接被复用后，被延迟的 $SEQ = 301$ 抵达了客户端，那么客户端是有可能正常接收这个过期的报文，这就会产生数据错乱等严重的问题。

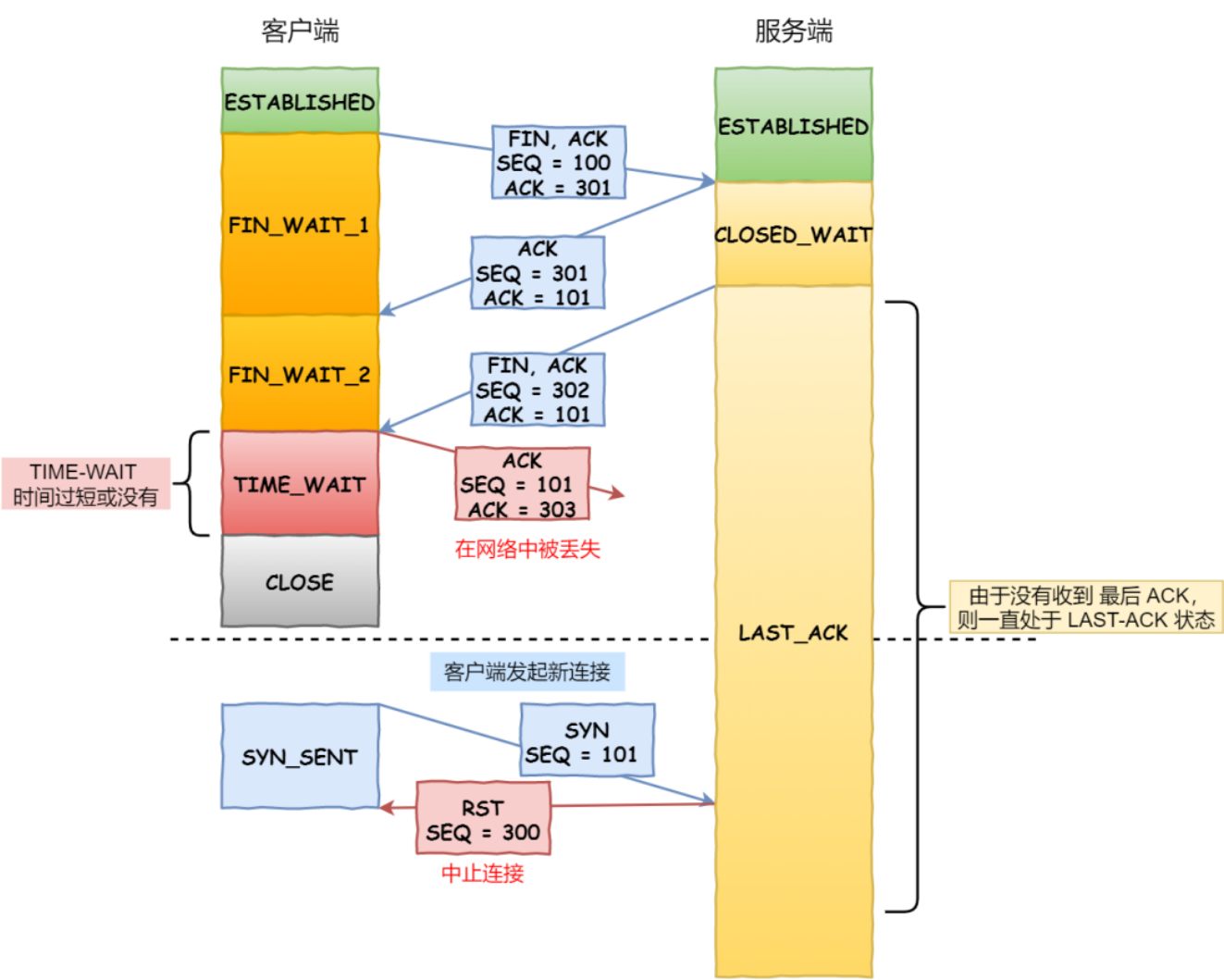
所以，TCP 就设计出了这么一个机制，经过 $2MSL$ 这个时

间，足以让两个方向上的数据包都被丢弃，使得原来连接的数据包在网络中都自然消失，再出现的数据包一定都是新建立连接所产生的。

原因二：保证连接正确关闭

TIME-WAIT 的另外一个作用是等待足够的时间以确保最后的 **ACK** 能让被动关闭方接收，从而帮助其正常关闭。

假设 TIME-WAIT 没有等待时间或时间过短，断开连接会造成什么问题呢？



没有确保正常断开的异常

- 如上图红色框框客户端四次挥手的最后一个 ACK 报文如果在网络中被丢失了，此时如果客户端 TIME-WAIT

过短或没有，则就直接进入了 CLOSE 状态了，那么服务端则会一直处在 LASE-ACK 状态。

- 当客户端发起建立连接的 SYN 请求报文后，服务端会发送 RST 报文给客户端，连接建立的过程就会被终止。

我们再回过头来看看，为什么 TIME_WAIT 状态要保持 60 秒呢？这与孤儿连接 FIN_WAIT2 状态默认保留 60 秒的原理是一样的，因为这两个状态都需要保持 **2MSL** 时长。

MSL 全称是 **Maximum Segment Lifetime**，它定义了一个报文在网络中的最长生存时间（报文每经过一次路由器的转发，IP 头部的 TTL 字段就会减 1，减到 0 时报文就被丢弃，这就限制了报文的最长存活时间）。

为什么是 2 MSL 的时长呢？这其实是相当于至少允许报文丢失一次。比如，若 ACK 在一个 MSL 内丢失，这样被动方重发的 FIN 会在第 2 个 MSL 内到达，TIME_WAIT 状态的连接可以应对。

为什么不是 4 或者 8 MSL 的时长呢？你可以想象一个丢包率达到百分之一的糟糕网络，连续两次丢包的概率只有万分之一，这个概率实在是太小了，忽略它比解决它更具性价比。

因此，**TIME_WAIT** 和 **FIN_WAIT2** 状态的最大时长都是 **2 MSL**，由于在 Linux 系统中，MSL 的值固定为 30 秒，所以它们都是 **60 秒**。

虽然 TIME_WAIT 状态有存在的必要，但它毕竟会消耗系统资源。如果发起连接一方的 **TIME_WAIT** 状态过多，占

满了所有端口资源，则会导致无法创建新连接。

- 客户端受端口资源限制：如果客户端 `TIME_WAIT` 过多，就会导致端口资源被占用，因为端口就65536个，被占满就会导致无法创建新的连接；
- 服务端受系统资源限制：由于一个 四元组表示TCP连接，理论上服务端可以建立很多连接，服务端确实只监听一个端口 但是会把连接扔给处理线程，所以理论上监听的端口可以继续监听。但是线程池处理不了那么多一直不断的连接了。所以当服务端出现大量 `TIME_WAIT` 时，系统资源被占满时，会导致处理不过来新的连接；

另外，Linux 提供了 `tcp_max_tw_buckets` 参数，当 `TIME_WAIT` 的连接数量超过该参数时，新关闭的连接就不再经历 `TIME_WAIT` 而直接关闭：



```
# 调整 timewait 最大个数
$ echo 5000 > /proc/sys/net/ipv4/tcp_max_tw_buckets
```

当服务器的并发连接增多时，相应地，同时处于 `TIME_WAIT` 状态的连接数量也会变多，此时就应当调大 `tcp_max_tw_buckets` 参数，减少不同连接间数据错乱的概率。

`tcp_max_tw_buckets` 也不是越大越好，毕竟内存和端口都是有限的。

有一种方式可以在建立新连接时，复用处于 `TIME_WAIT` 状态的连接，那就是打开 `tcp_tw_reuse` 参数。但是需要注意，该参数是只用于客户端（建立连接的发起方），因为是在调用 `connect()` 时起作用的，而对于服务端（被动连接方）是没有用的。



```
# 打开 tcp_tw_reuse 功能
$ echo 1 > /proc/sys/net/ipv4/tcp_tw_reuse
```

`tcp_tw_reuse` 从协议角度理解是安全可控的，可以复用处于 `TIME_WAIT` 的端口为新的连接所用。

什么是协议角度理解的安全可控呢？主要有两点：

- 只适用于连接发起方，也就是 C/S 模型中的客户端；
- 对应的 `TIME_WAIT` 状态的连接创建时间超过 1 秒才可以被复用。

使用这个选项，还有一个前提，需要打开对 TCP 时间戳的支持（对方也要打开）：



```
# 打开时间戳功能，默认值为 1
$ echo 1 > /proc/sys/net/ipv4/tcp_timestamps
```

由于引入了时间戳，它能带来了些好处：

- 我们在前面提到的 2MSL 问题就不复存在了，因为重复的数据包会因为时间戳过期被自然丢弃；
- 同时，它还可以防止序列号绕回，也是因为重复的数据包会由于时间戳过期被自然丢弃；

老版本的 Linux 还提供了 `tcp_tw_recycle` 参数，但是当开启了它，就有两个坑：

- **Linux** 会加快客户端和服务端 **TIME_WAIT** 状态的时间，也就是它会使得 **TIME_WAIT** 状态会小于 60 秒，很容易导致数据错乱；
- 另外，**Linux** 会丢弃所有来自远端时间戳小于上次记录的时间戳（由同一个远端发送的）的任何数据包。就是要使用该选项，则必须保证数据包的时间戳是单调递增的。那么，问题在于，此处的时间戳并不是我们通常意义上的绝对时间，而是一个相对时间。很多情况下，我们是没法保证时间戳单调递增的，比如使用了 NAT，LVS 等情况；

所以，不建议设置为 1，建议关闭它：



```
# 关闭 tcp_tw_recycle 功能
$ echo 0 > /proc/sys/net/ipv4/tcp_tw_recycle
```

在 Linux 4.12 版本后，Linux 内核直接取消了这一参数。

另外，我们可以在程序中设置 socket 选项，来设置调用

close 关闭连接行为。

```
struct linger so_linger;  
so_linger.l_onoff = 1;  
so_linger.l_linger = 0;  
setsockopt(s, SOL_SOCKET, SO_LINGER, &so_linger, sizeof(so_linger));
```

如果`l_onoff`为非 0，且`l_linger`值为 0，那么调用`close`后，会立刻发送一个 **RST** 标志给对端，该 **TCP** 连接将跳过四次挥手，也就跳过了 **TIME_WAIT** 状态，直接关闭。

但这为跨越 **TIME_WAIT** 状态提供了一个可能，不过是一个非常危险的行为，不值得提倡。

被动方的优化

当被动方收到 **FIN** 报文时，内核会自动回复 **ACK**，同时连接处于 **CLOSE_WAIT** 状态，顾名思义，它表示等待应用进程调用 `close` 函数关闭连接。

内核没有权利替代进程去关闭连接，因为如果主动方是通过 `shutdown` 关闭连接，那么它就是想在半关闭连接上接收数据或发送数据。因此，Linux 并没有限制 **CLOSE_WAIT** 状态的持续时间。

当然，大多数应用程序并不使用 `shutdown` 函数关闭连接。所以，当你用 `netstat` 命令发现大量 **CLOSE_WAIT** 状态。就需要排查你的应用程序，因为可能因为应用程序出现了 **Bug**，`read` 函数返回 0 时，没有调用 `close` 函数。

处于 CLOSE_WAIT 状态时，调用了 close 函数，内核就会发出 FIN 报文关闭发送通道，同时连接进入 LAST_ACK 状态，等待主动方返回 ACK 来确认连接关闭。

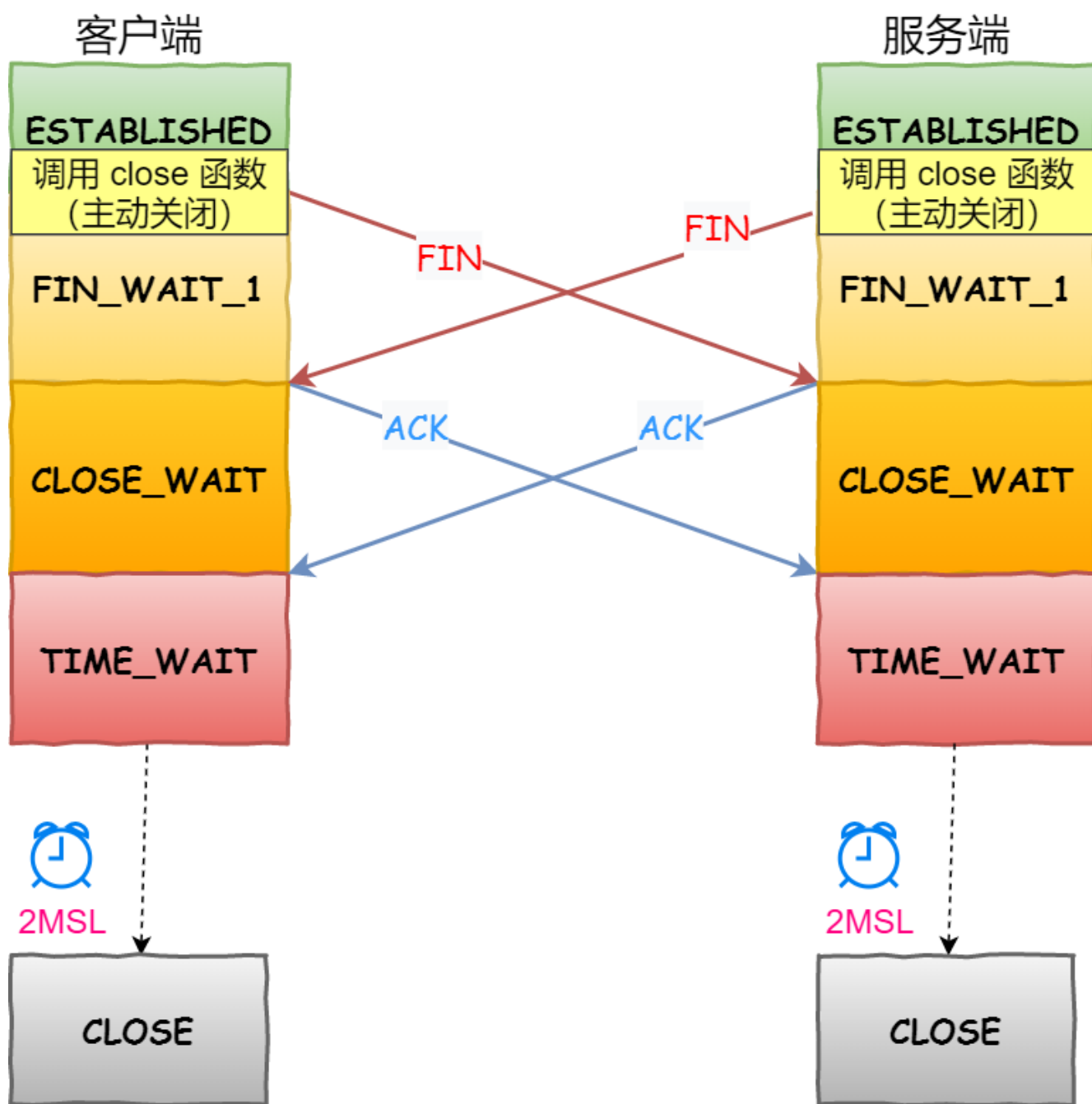
如果迟迟收不到这个 ACK，内核就会重发 FIN 报文，重发次数仍然由 tcp_orphan_retries 参数控制，这与主动方重发 FIN 报文的优化策略一致。

还有一点我们需要注意的，如果被动方迅速调用 **close** 函数，那么被动方的 **ACK** 和 **FIN** 有可能在一个报文中发送，这样看起来，四次挥手会变成三次挥手，这只是一种特殊情况，不用在意。

如果连接双方同时关闭连接，会怎么样？

由于 TCP 是双全工的协议，所以是会出现两方同时关闭连接的现象，也就是同时发送了 FIN 报文。

此时，上面介绍的优化策略仍然适用。两方发送 FIN 报文时，都认为自己是主动方，所以都进入了 FIN_WAIT1 状态，FIN 报文的重发次数仍由 tcp_orphan_retries 参数控制。



同时关闭

接下来，双方在等待 **ACK** 报文的过程中，都等来了 **FIN** 报文。这是一种新情况，所以连接会进入一种叫做 **CLOSING** 的新状态，它替代了 **FIN_WAIT2** 状态。接着，双方内核回复 **ACK** 确认对方发送通道的关闭后，进入 **TIME_WAIT** 状态，等待 **2MSL** 的时间后，连接自动关闭。

小结

针对 TCP 四次挥手的优化，我们需要根据主动方和被动方四次挥手状态变化来调整系统 TCP 内核参数。

优化四次挥手的策略	
策略	TCP 内核参数
调整 FIN 报文重传次数	tcp_orphan_retries
调整 FIN_WAIT2 状态的时间 (只适用 close 函数关闭的连接)	tcp_fin_timeout
调整孤儿连接的上限个数 (只适用 close 函数关闭的连接)	tcp_max_orphans
调整 time_wait 状态的上限个数	tcp_max_tw_buckets
复用 time_wait 状态的连接 (只适用客户端)	tcp_tw_reuse 、 tcp_timestamps

四次挥手的优化策略

主动方的优化

主动发起 FIN 报文断开连接的一方，如果迟迟没收到对方的 ACK 回复，则会重传 FIN 报文，重传的次数由 `tcp_orphan_retries` 参数决定。

当主动方收到 ACK 报文后，连接就进入 FIN_WAIT2 状态，根据关闭的方式不同，优化的方式也不同：

- 如果这是 `close` 函数关闭的连接，那么它就是孤儿连接。如果 `tcp_fin_timeout` 秒内没有收到对方的 FIN 报文，连接就直接关闭。同时，为了应对孤儿连接占用太多的资源，`tcp_max_orphans` 定义了最大孤儿连接的数量，超过时连接就会直接释放。
- 反之是 `shutdown` 函数关闭的连接，则不受此参数限制；

当主动方接收到 FIN 报文，并返回 ACK 后，主动方的连接进入 TIME_WAIT 状态。这一状态会持续 1 分钟，为了防止 TIME_WAIT 状态占用太多的资源，`tcp_max_tw_buckets` 定义了最大数量，超过时连接也会直接释放。

当 TIME_WAIT 状态过多时，还可以通过设置 `tcp_tw_reuse` 和 `tcp_timestamps` 为 1，将 TIME_WAIT 状态的端口复用于作为客户端的新连接，注意该参数只适用于客户端。

被动方的优化

被动关闭的连接方应对非常简单，它在回复 ACK 后就进入了 CLOSE_WAIT 状态，等待进程调用 `close` 函数关闭连接。因此，出现大量 CLOSE_WAIT 状态的连接时，应当从应用程序中找问题。

当被动方发送 FIN 报文后，连接就进入 LAST_ACK 状态，在未等到 ACK 时，会在 `tcp_orphan_retries` 参数的控制下重发 FIN 报文。

03 TCP 传输数据的性能提升

在前面介绍的是三次握手和四次挥手的优化策略，接下来主要介绍的是 TCP 传输数据时的优化策略。

TCP 连接是由内核维护的，内核会为每个连接建立内存缓冲区：

- 如果连接的内存配置过小，就无法充分使用网络带宽，TCP 传输效率就会降低；
- 如果连接的内存配置过大，很容易把服务器资源耗尽，这样就会导致新连接无法建立；

因此，我们必须理解 Linux 下 TCP 内存的用途，才能正确地配置内存大小。

滑动窗口是如何影响传输速度的？

TCP 会保证每一个报文都能够抵达对方，它的机制是这样：报文发出去后，必须接收到对方返回的确认报文 ACK，如果迟迟未收到，就会超时重发该报文，直到收到对方的 ACK 为止。

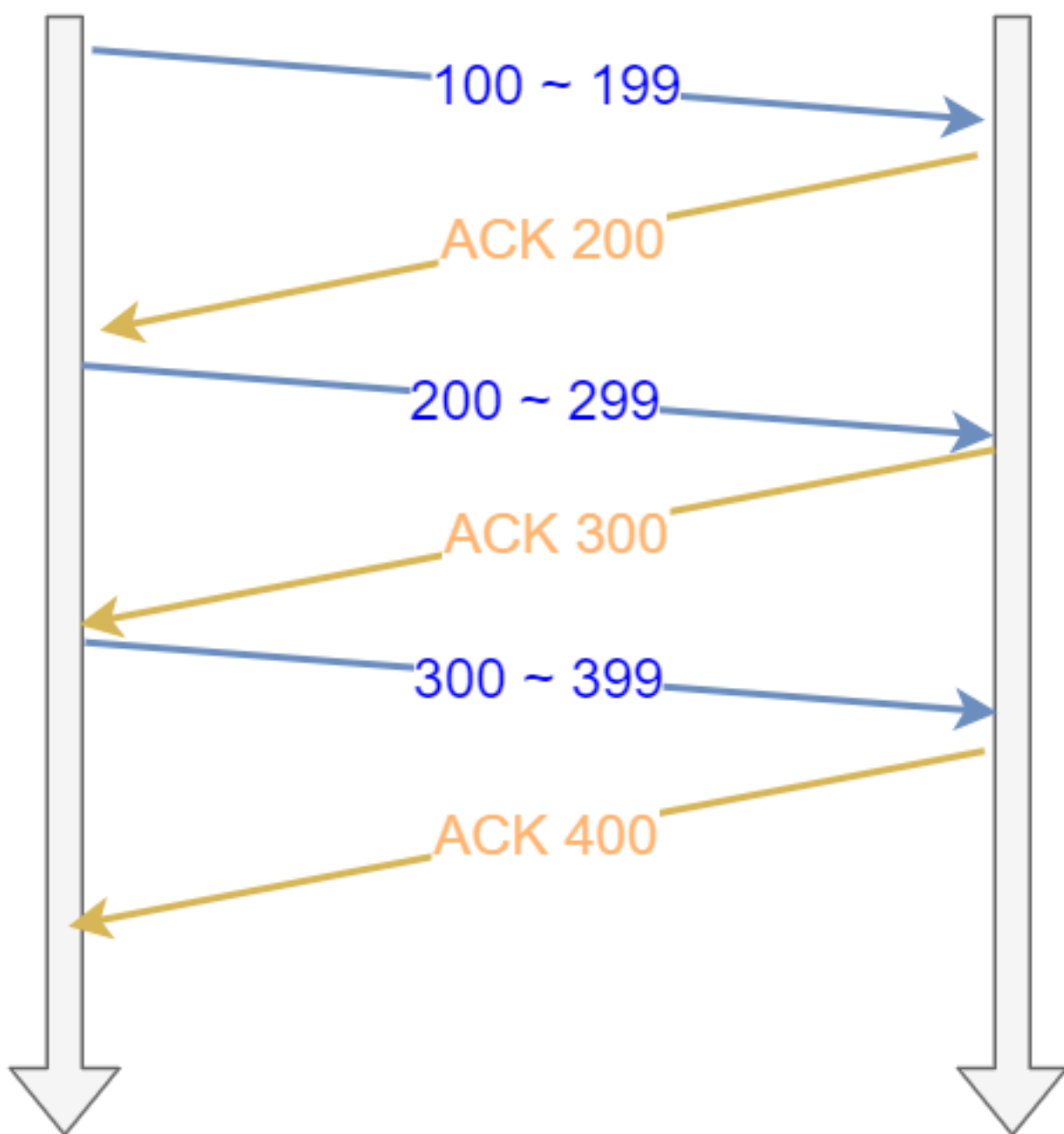
所以，**TCP** 报文发出去后，并不会立马从内存中删除，因为重传时还需要用到它。

由于 TCP 是内核维护的，所以报文存放在内核缓冲区。如果连接非常多，我们可以通过 `free` 命令观察到 `buff/cache` 内存是会增大。

如果 TCP 是每发送一个数据，都要进行一次确认应答。当上一个数据包收到了应答了，再发送下一个。这个模式就有点像我和你面对面聊天，你一句我一句，但这种方式缺点是效率比较低的。

发送方

接收方



为每个数据包确认应答的缺点：
包的往返时间越长，网络的吞吐量会越低

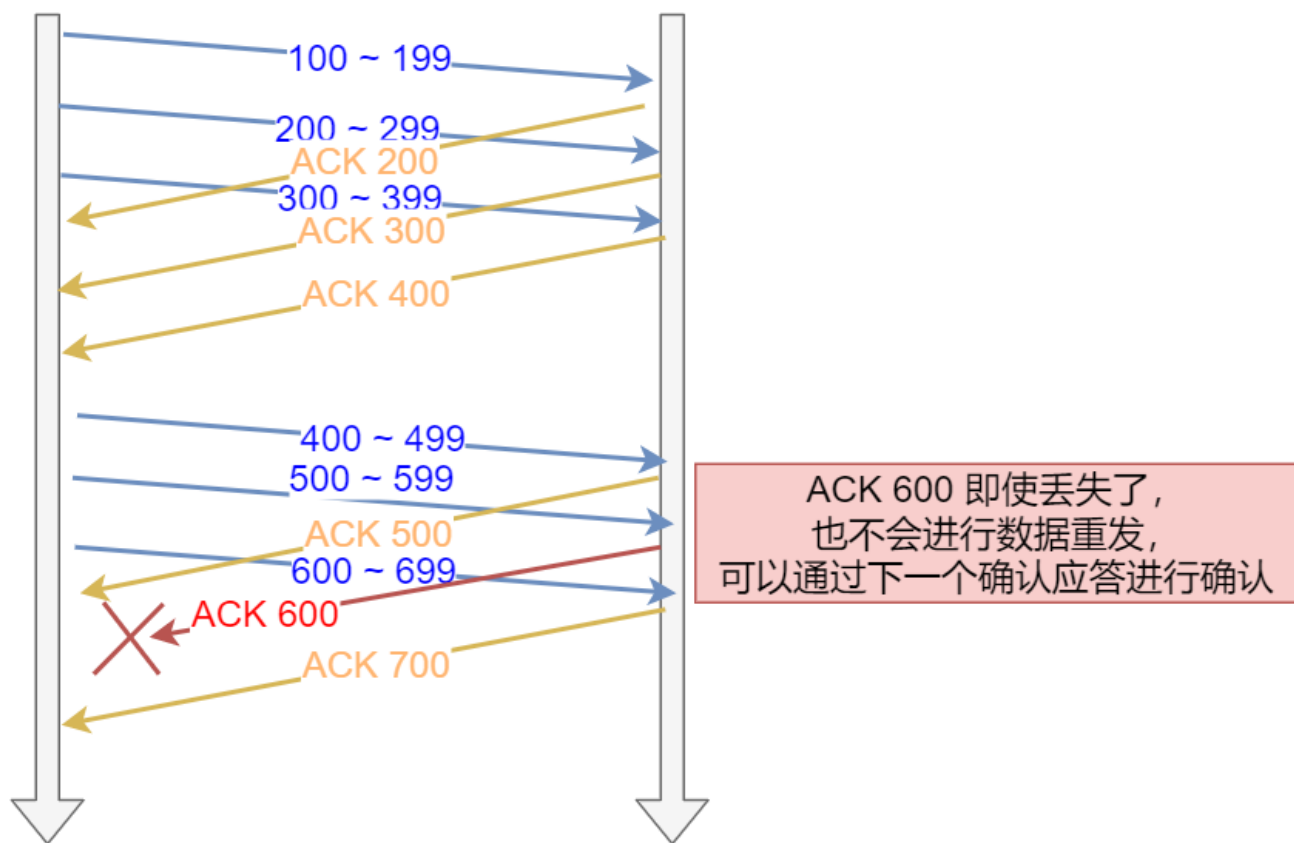
按数据包进行确认应答

所以，这样的传输方式有一个缺点：数据包的往返时间越长，通信的效率就越低。

要解决这一问题不难，并行批量发送报文，再批量确认报文即刻。

发送方

接收方



并行处理

然而，这引出了另一个问题，发送方可以随心所欲的发送报文吗？当然这不现实，我们还得考虑接收方的处理能力。

当接收方硬件不如发送方，或者系统繁忙、资源紧张时，是无法瞬间处理这么多报文的。于是，这些报文只能被丢掉，使得网络效率非常低。

为了解决这种现象发生，**TCP** 提供一种机制可以让「发送方」根据「接收方」的实际接收能力控制发送的数据量，这就是滑动窗口的由来。

接收方根据它的缓冲区，可以计算出后续能够接收多少字节的报文，这个数字叫做接收窗口。当内核接收到报文时，必须用缓冲区存放它们，这样剩余缓冲区空间变小，接收窗口也就变小了；当进程调用 `read` 函数后，数据被

读入了用户空间，内核缓冲区就被清空，这意味着主机可以接收更多的报文，接收窗口就会变大。

因此，接收窗口并不是恒定不变的，接收方会把当前可接收的大小放在 TCP 报文头部中的窗口字段，这样就可以起到窗口大小通知的作用。

发送方的窗口等价于接收方的窗口吗？如果不考虑拥塞控制，发送方的窗口大小「约等于」接收方的窗口大小，因为窗口通知报文在网络传输是存在时延的，所以是约等于的关系。

TCP 头部格式

源端口号 (16位)					目标端口号 (16位)					
序列号 (32位)										
确认应答号 (32位)										
首部长度 (4位)		保留 (6位)		U	A	P	R	S	F	窗口大小 (16位)
				R	C	S	S	Y	I	
				G	K	H	T	N	N	
校验和 (16位)					紧急指针 (16位)					
选项 (长度可变)										

TCP 头部

从上图中可以看到，窗口字段只有 2 个字节，因此它最多能表达 65535 字节大小的窗口，也就是 64KB 大小。

这个窗口大小最大值，在当今高速网络下，很明显是不够用的。所以后续有了扩充窗口的方法：在 TCP 选项字段定义了窗口扩大因子，用于扩大TCP通告窗口，使 TCP

的窗口大小从 2 个字节（16 位） 扩大为 30 位，所以此时窗口的最大值可以达到 1GB (2^{30}) 。

Linux 中打开这一功能， 需要把 `tcp_window_scaling` 配置设为 1（默认打开）：

```
# 启用窗口扩大因子功能，默认即打开
$ echo 1 > /proc/sys/net/ipv4/tcp_window_scaling
```

要使用窗口扩大选项，通讯双方必须在各自的 SYN 报文中发送这个选项：

- 主动建立连接的一方在 SYN 报文中发送这个选项；
- 而被动建立连接的一方只有在收到带窗口扩大选项的 SYN 报文之后才能发送这个选项。

这样看来，只要进程能及时地调用 `read` 函数读取数据，并且接收缓冲区配置得足够大，那么接收窗口就可以无限地放大，发送方也就无限地提升发送速度。

这是不可能的，因为网络的传输能力是有限的，当发送方依据发送窗口，发送超过网络处理能力的报文时，路由器会直接丢弃这些报文。因此，缓冲区的内存并不是越大越好。

如果确定最大传输速度？

在前面我们知道了 TCP 的传输速度，受制于发送窗口与接收窗口，以及网络设备传输能力。其中，窗口大小由内

核缓冲区大小决定。如果缓冲区与网络传输能力匹配，那么缓冲区的利用率就达到了最大化。

问题来了，如何计算网络的传输能力呢？

相信大家都知道网络是有「带宽」限制的，带宽描述的是网络传输能力，它与内核缓冲区的计量单位不同：

- 带宽是单位时间内的流量，表达是「速度」，比如常见的带宽 100 MB/s；
- 缓冲区单位是字节，当网络速度乘以时间才能得到字节数；

这里需要说一个概念，就是带宽时延积，它决定网络中飞行报文的大小，它的计算方式：

$$\text{带宽时延积 BDP} = \text{RTT} * \text{带宽}$$

比如最大带宽是 100 MB/s，网络时延（RTT）是 10ms 时，意味着客户端到服务端的网络一共可以存放 $100\text{MB/s} * 0.01\text{s} = 1\text{MB}$ 的字节。

这个 1MB 是带宽和时延的乘积，所以它就叫「带宽时延积」（缩写为 BDP，Bandwidth Delay Product）。同时，这 1MB 也表示「飞行中」的 TCP 报文大小，它们就在网络线路、路由器等网络设备上。如果飞行报文超过了 1 MB，就会导致网络过载，容易丢包。

由于发送缓冲区大小决定了发送窗口的上限，而发送窗口

又决定了「已发送未确认」的飞行报文的上限。因此，发送缓冲区不能超过「带宽时延积」。

发送缓冲区与带宽时延积的关系：

- 如果发送缓冲区「超过」带宽时延积，超出的部分就没办法有效的网络传输，同时导致网络过载，容易丢包；
- 如果发送缓冲区「小于」带宽时延积，就不能很好的发挥出网络的传输效率。

所以，发送缓冲区的大小最好是往带宽时延积靠近。

怎样调整缓冲区大小？

在 Linux 中发送缓冲区和接收缓冲都是可以用参数调节的。设置完后，Linux 会根据你设置的缓冲区进行动态调节。

调节发送缓冲区范围

先来看看发送缓冲区，它的范围通过 `tcp_wmem` 参数配置：

```
# 调整 TCP 发送缓冲区范围
$ echo "4096 16384 4194304" > /proc/sys/net/ipv4/tcp_wmem
```

上面三个数字单位都是字节，它们分别表示：

- 第一个数值是动态范围的最小值，4096 byte = 4K；

- 第二个数值是初始默认值，87380 byte \approx 86K；
- 第三个数值是动态范围的最大值，4194304 byte = 4096K（4M）；

发送缓冲区是自行调节的，当发送方发送的数据被确认后，并且没有新的数据要发送，就会把发送缓冲区的内存释放掉。

调节接收缓冲区范围

而接收缓冲区的调整就比较复杂一些，先来看看设置接收缓冲区范围的 `tcp_rmem` 参数：

```
# 调整 TCP 接收缓冲区范围
$ echo "4096 87380 6291456" > /proc/sys/net/ipv4/tcp_rmem
```

上面三个数字单位都是字节，它们分别表示：

- 第一个数值是动态范围的最小值，表示即使在内存压力下也可以保证的最小接收缓冲区大小，4096 byte = 4K；
- 第二个数值是初始默认值，87380 byte \approx 86K；
- 第三个数值是动态范围的最大值，6291456 byte = 6144K（6M）；

接收缓冲区可以根据系统空闲内存的大小来调节接收窗口：

- 如果系统的空闲内存很多，就可以自动把缓冲区增大一些，这样传给对方的接收窗口也会变大，因而提升发送方发送的传输数据数量；
- 反正，如果系统的内存很紧张，就会减少缓冲区，这虽然会降低传输效率，可以保证更多的并发连接正常工作；

发送缓冲区的调节功能是自动开启的，而接收缓冲区则需要配置 `tcp_moderate_rcvbuf` 为 1 来开启调节功能：

```
# 启动 TCP 接收缓冲区自动调节功能
$ echo 1 > /proc/sys/net/ipv4/tcp_moderate_rcvbuf
```

调节 TCP 内存范围

接收缓冲区调节时，怎么知道当前内存是否紧张或充分呢？这是通过 `tcp_mem` 配置完成的：

```
# 调整 TCP 内存范围
$ echo "88560 118080 177120" > /proc/sys/net/ipv4/tcp_mem
```

上面三个数字单位不是字节，而是「页面大小」，1 页表示 4KB，它们分别表示：

- 当 TCP 内存小于第 1 个值时，不需要进行自动调节；

- 在第 1 和第 2 个值之间时，内核开始调节接收缓冲区的大小；
- 大于第 3 个值时，内核不再为 TCP 分配新内存，此时新连接是无法建立的；

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。根据当前 `tcp_mem` 最大内存页面数是 177120，当内存为 $(177120 * 4) / 1024K \approx 692M$ 时，系统将无法为新的 TCP 连接分配内存，即 TCP 连接将被拒绝。

根据实际场景调节的策略

在高并发服务器中，为了兼顾网速与大量的并发连接，我们应当保证缓冲区的动态调整的最大值达到带宽时延积，而最小值保持默认的 4K 不变即可。而对于内存紧张的服务而言，调低默认值是提高并发的有效手段。

同时，如果这是网络 IO 型服务器，那么，调大 `tcp_mem` 的上限可以让 TCP 连接使用更多的系统内存，这有利于提升并发能力。需要注意的是，`tcp_wmem` 和 `tcp_rmem` 的单位是字节，而 `tcp_mem` 的单位是页面大小。而且，千万不要在 `socket` 上直接设置 `SO_SNDBUF` 或者 `SO_RCVBUF`，这样会关闭缓冲区的动态调整功能。

小结

本节针对 TCP 优化数据传输的方式，做了一些介绍。

数据传输的优化策略	
策略	TCP 内核参数
扩大窗口大小	tcp_window_scaling
调整发送缓冲区范围	tcp_wmem
调整接收缓冲区范围	tcp_rmem
打开接收缓冲区动态调节	tcp_moderate_rcvbuf
调整内存范围	tcp_mem

数据传输的优化策略

TCP 可靠性是通过 ACK 确认报文实现的，又依赖滑动窗口提升了发送速度也兼顾了接收方的处理能力。

可是，默认的滑动窗口最大值只有 64 KB，不满足当今的高速网络的要求，要想要想提升发送速度必须提升滑动窗口的上限，在 Linux 下是通过设置 tcp_window_scaling 为 1 做到的，此时最大值可高达 1GB。

滑动窗口定义了网络中飞行报文的最大字节数，当它超过带宽时延积时，网络过载，就会发生丢包。而当它小于带宽时延积时，就无法充分利用网络带宽。因此，滑动窗口的设置，必须参考带宽时延积。

内核缓冲区决定了滑动窗口的上限，缓冲区可分为：发送缓冲区 tcp_wmem 和接收缓冲区 tcp_rmem。

Linux 会对缓冲区动态调节，我们应该把缓冲区的上限设置为带宽时延积。发送缓冲区的调节功能是自动打开的，

而接收缓冲区需要把 `tcp_moderate_rcvbuf` 设置为 1 来开启。其中，调节的依据是 TCP 内存范围 `tcp_mem`。

但需要注意的是，如果程序中的 `socket` 设置 `SO_SNDBUF` 和 `SO_RCVBUF`，则会关闭缓冲区的动态整功能，所以不建议在程序设置它俩，而是交给内核自动调整比较好。

有效配置这些参数后，既能够最大程度地保持并发性，也能让资源充裕时连接传输速度达到最大值。

巨人的肩膀

[1] 系统性能调优必知必会.陶辉.极客时间.

[2] 网络编程实战专栏.盛延敏.极客时间.

[3]

<http://www.blogjava.net/yongboy/archive/2013/04/11/397677.html>

[4] <http://blog.itpub.net/31559359/viewspace-2284113/>

[5] <https://blog.51cto.com/professor/1909022>

好文推荐

本篇是 **TCP** 系列的「终章」了，往期的 TCP 图解文章如下：

[TCP 半连接队列和全连接队列满了会发生什么？又该如何应对？](#)

[实战！我用“大白鲨”让你看见 TCP](#)

[你还在为 TCP 重传、滑动窗口、流量控制、拥塞控制发愁吗？看完图解就不愁了](#)

[硬不硬你说了算！近 40 张图解被问千百遍的 TCP 三次握手和四次挥手面试题](#)

唠嗑唠嗑

跟大家说个沉痛的事情。

我想大部分小伙伴都发现了，最近公众号改版，订阅号里的信息流不再是以时间顺序了，而是以推荐算法方式显示顺序。

这对小林这种「周更」的作者，真的一次重重打击，非常的不友好。

因为长时间没发文，公众号可能会把推荐的权重降低，这就会导致很多读者，会收不到我的「最新」的推文，如此下去，那小林文章不就无人问津了？（抱头痛哭 ...）

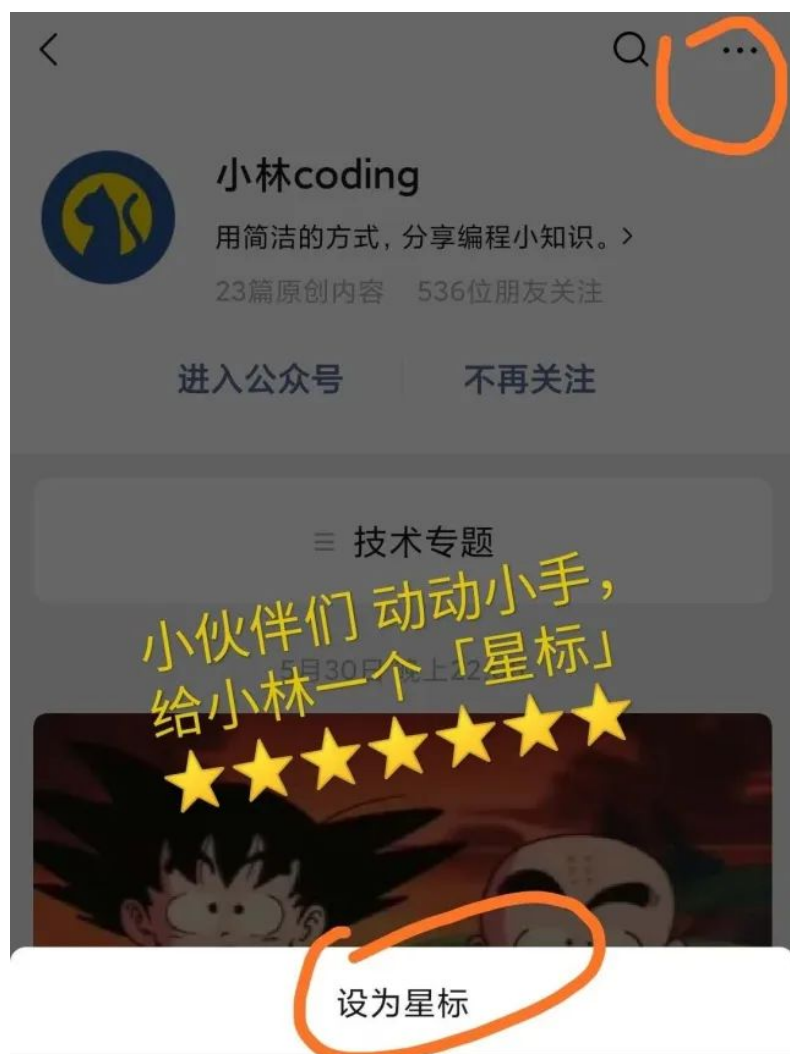
另外，小林更文时间长的原因，不是因为偷懒。

而是为了把知识点「写的更清楚，画的更清晰」，所以这必然会花费更多更长的时间。

如果你认可和喜欢小林的文章，不想错过文章的第一时间推送，可以动动你的小手手，给小林公众号一个「星标」。

平时没事，就让「小林coding」静静地躺在你的订阅号底部，但是你要知道它在这其间并非无所事事，而是在努力地准备着更好的内容，等准备好了，它自然会「蹦出」在你面前。

小林是专为大家图解的工具人，Goodbye，我们下次见！



扫一扫
关注爱图解的
「小林coding」

推荐给朋友