



# 聊聊 TIME\_WAIT

# 什么是 TIME\_WAIT

TIME\_WAIT 是 TCP 所有状态中最不好理解的一种状态。首先，我们需要明确，**只有主动断开的那一方才会进入 TIME\_WAIT 状态**，且会在那个状态持续 2 个 MSL (Max Segment Lifetime) 。

# MSL: Max Segment Lifetime

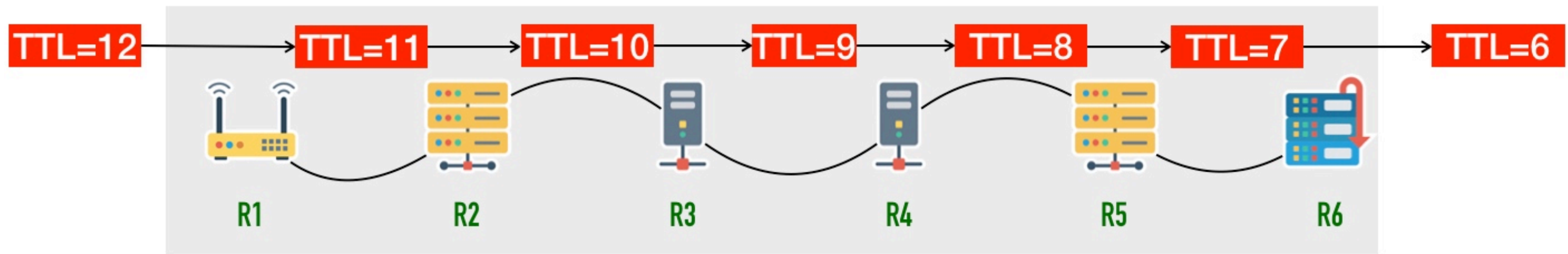
MSL（报文最大生存时间）是 TCP 报文在网络中的最大生存时间。这个值与 IP 报文头的 TTL 字段有密切的关系。

IP 报文头中有一个 8 位的存活时间字段（Time to live, TTL），这个存活时间存储的不是具体的时间，而是一个 IP 报文最大可经过的路由数，每经过一个路由器，TTL 减 1，当 TTL 减到 0 时这个 IP 报文会被丢弃。如果一个报文从源主机到目的主机之间

# Time to live, TTL

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
版本 (Version)				IHL				DS 字段						ECN		总长度 (Total Length)																
标识 (Identification)															标记 (Flags)		分片偏移 (Fragment Offset)															
生存时间 (TTL)								协议 (Protocol)							头部校验和 (Header Checksum)																	
源 IP 地址 (Source IP Address)																																
目标 IP 地址 (Destination IP Address)																																
选项 (Options)																																

TTL 经过路由器不断减小，假设初始的 TTL 为 12，经过下一个路由器 R1 以后 TTL 变为 10，后面每经过一个路由器以后 TTL 减 1



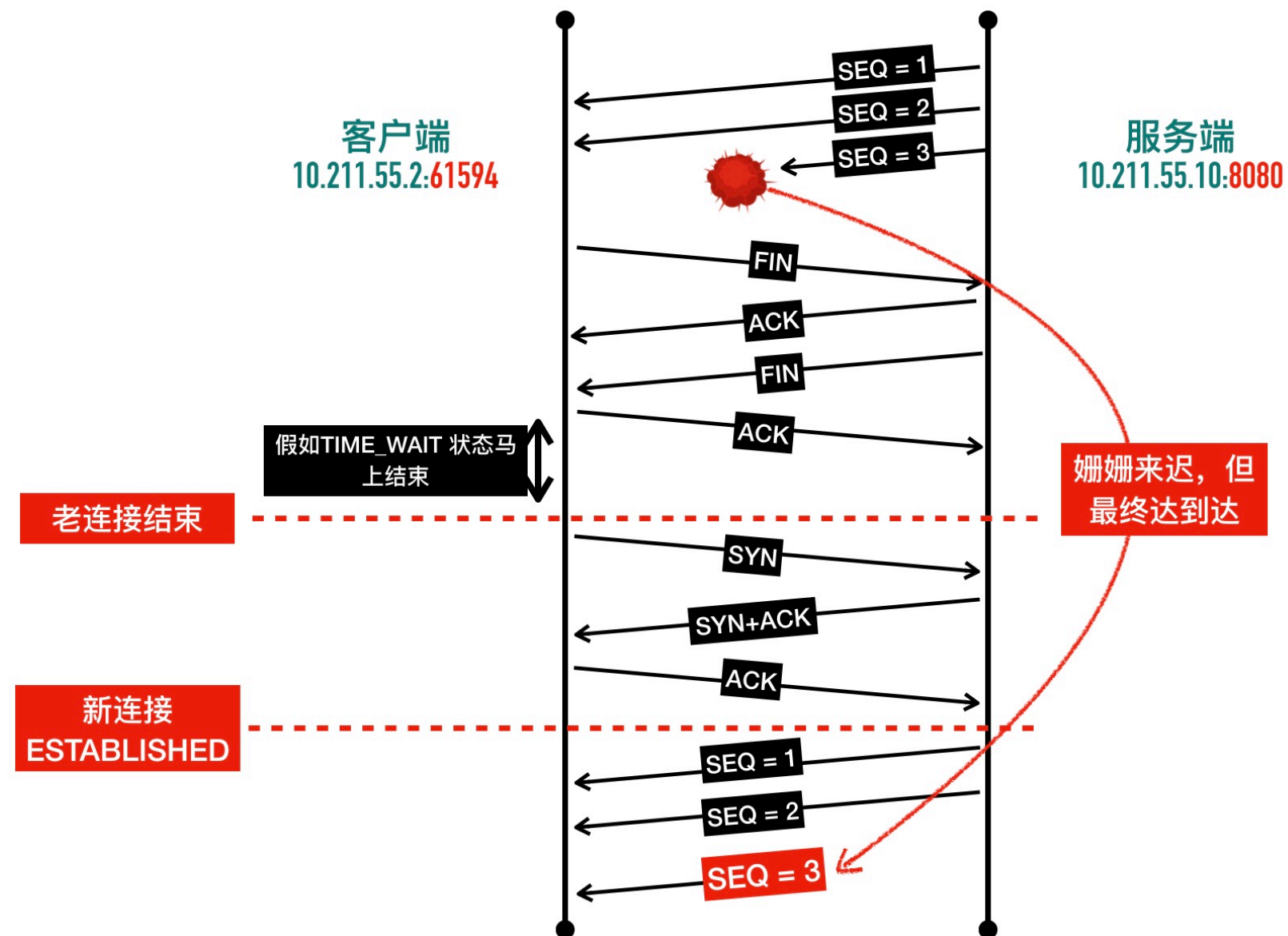
# 构造一个 TIME\_WAIT

要构造一个 TIME\_WAIT 非常简单，只需要建立一个 TCP 连接，然后断开某一方连接，主动断开的那一方就会进入 TIME\_WAIT 状态



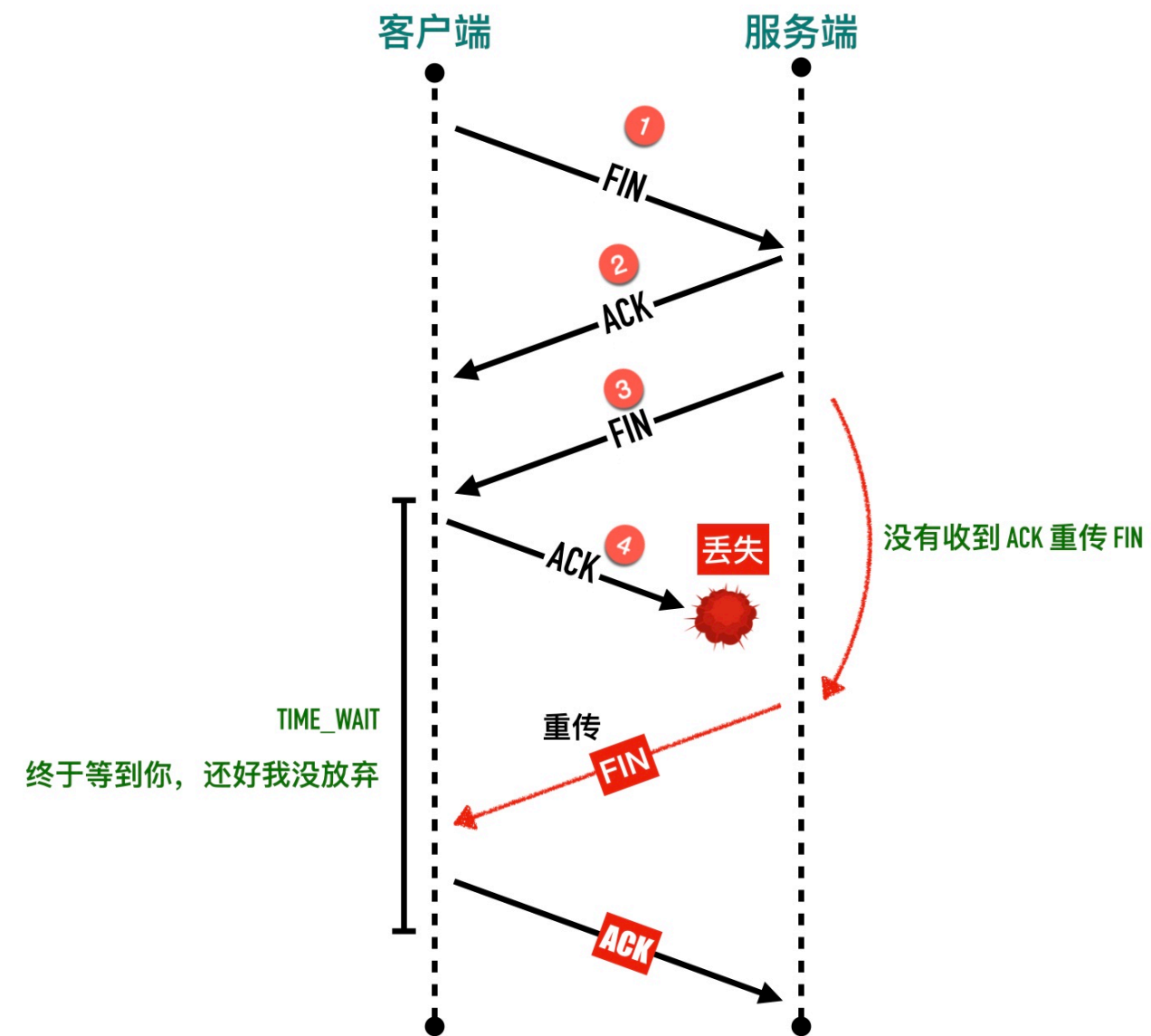
TIME\_WAIT 存在的原因是什么

第一个原因是：数据报文可能在发送途中延迟但最终会到达，因此要等老的“迷路”的重复报文段在网络中过期失效，这样可以避免用**相同**源端口和目标端口创建新连接时收到旧连接姗姗来迟的数据包，造成数据错乱。

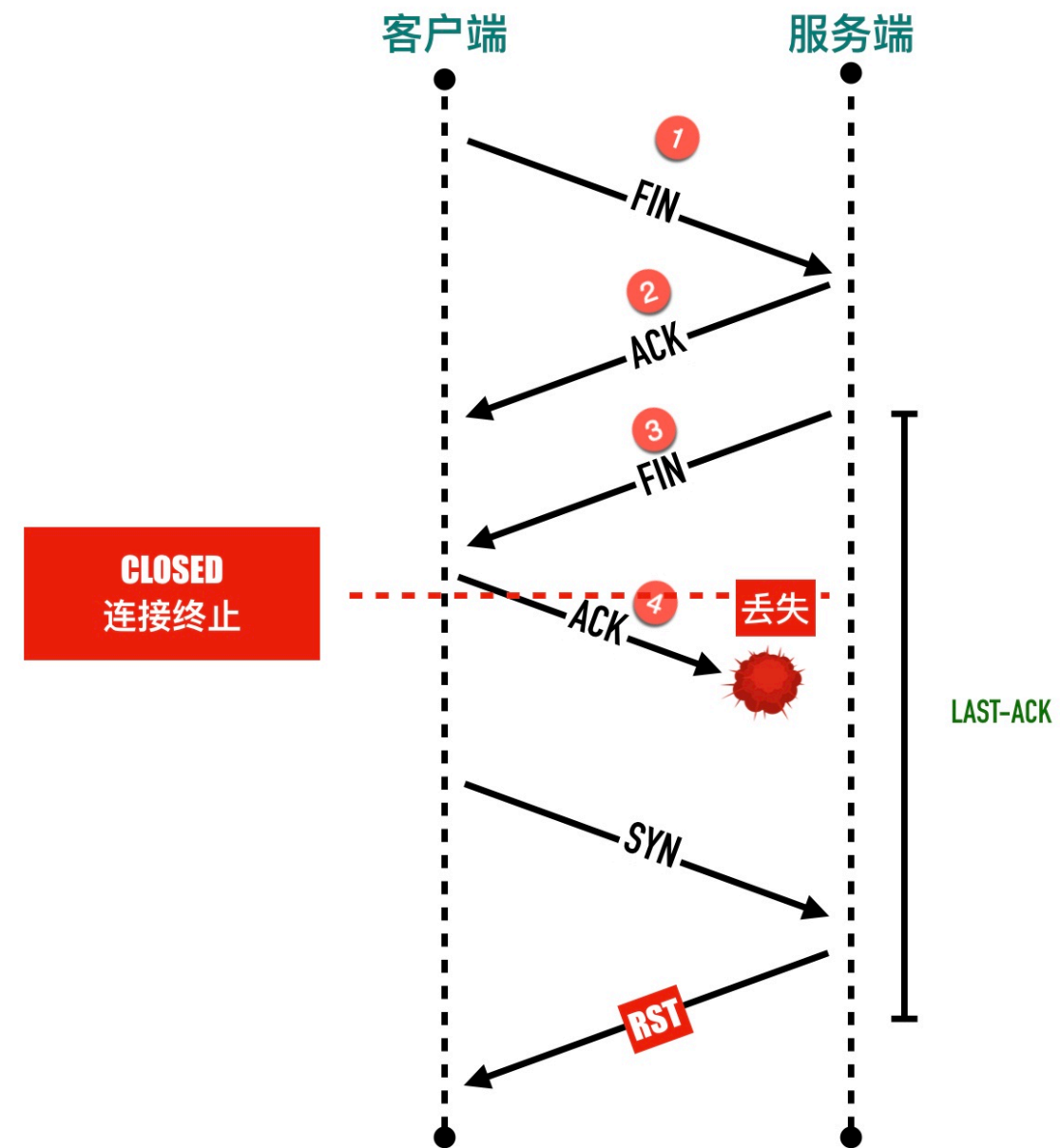




第二个原因是确保可靠实现 TCP 全双工终止连接。关闭连接的四次挥手中，最终的 ACK 由主动关闭方发出，如果这个 ACK 丢失，对端（被动关闭方）将重发 FIN，如果主动关闭方不维持 TIME\_WAIT 直接进入 CLOSED 状态，则无法重传 ACK，被动关闭方因此不能及时可靠释放。



如果四次挥手的第 4 步中客户端发送了给服务端的确认 ACK 报文以后不进入 TIME\_WAIT 状态，直接进入 CLOSED 状态，然后重用端口建立新连接会发生什么呢？



# 为什么时间是两个 MSL

- 1 个 MSL 确保四次挥手中主动关闭方最后的 ACK 报文最终能达到对端
- 1 个 MSL 确保对端没有收到 ACK 重传的 FIN 报文可以到达

$2MS = \text{去向 ACK 消息最大存活时间 (MSL)} + \text{来向 FIN 消息的最大存活时间 (MSL)}$

# TIME\_WAIT 的问题

在一个非常繁忙的服务器上，如果有大量 TIME\_WAIT 状态的连接会怎么样呢？

- 连接表无法复用
- socket 结构体内存占用

## 连接表无法复用

因为处于 `TIME_WAIT` 的连接会存活 `2MSL` (60s)，意味着相同的 TCP 连接四元组（源端口、源 ip、目标端口、目标 ip）在一分钟之内都没有办法复用，通俗一点来讲就是“占着茅坑不拉屎”。

假设主动断开的一方是客户端，对于 web 服务器而言，目标地址、目标端口都是固定值（比如本机 ip + 80 端口），客户端的 IP 也是固定的，那么能变化的就只有端口了，在一台 Linux 机器上，端口最多是 65535 个（2 个字节）。如果客户端与服务器通信全部使用短连接，不停的创建连接，接着关闭连接，客户端机器会造成大量的 TCP 连接进入 `TIME_WAIT` 状态。

可以来写一个简单的 shell 脚本来测试一下，使用 nc 命令连接 redis 发送 ping 命令以后断开连接。

```
for i in {1..10000}; do
    echo ping | nc localhost 6379
done
```

查看一下处于 TIME\_WAIT 状态的连接的个数，短短的几秒钟内，TIME\_WAIT 状态的连接已经有了 8000 多个。

```
netstat -tnpa | grep -i 6379 | grep TIME_WAIT | wc -l
8192
```

如果在 60s 内有超过 65535 次 redis 短连接操作，就会出现端口不够用的情况，这也是使用连接池的一个重要原因。

# 应对 TIME\_WAIT 的各种操作

针对 TIME\_WAIT 持续时间过长的问题，Linux 新增了几个相关的选项，`net.ipv4.tcp_tw_reuse` 和 `net.ipv4.tcp_tw_recycle`。这两个参数都依赖于 TCP 头部的扩展选项：timestamp

# TCP 头部时间戳选项（TCP Timestamps Option, TSopt）

除了我们之前介绍的 MSS、Window Scale 还有一个非常重要的选项：时间戳（TCP Timestamps Option, TSopt）

0			1								2				3						
源端口（Source port）											目标端口（Destination port）										
序列号（Sequence number）																					
确认号（Acknowledgment number）																					
头部长度的		保留		N	C	E	U	A	P	R	S	F	窗口大小（Window Size）								
			W	C	R	A	P	R	S	S	Y	I									
			R	E	G	C	H	T	N	N											
校验和（Checksum）											紧急指针（Urgent pointer）										
选项（Options）、填充（Padding）																					




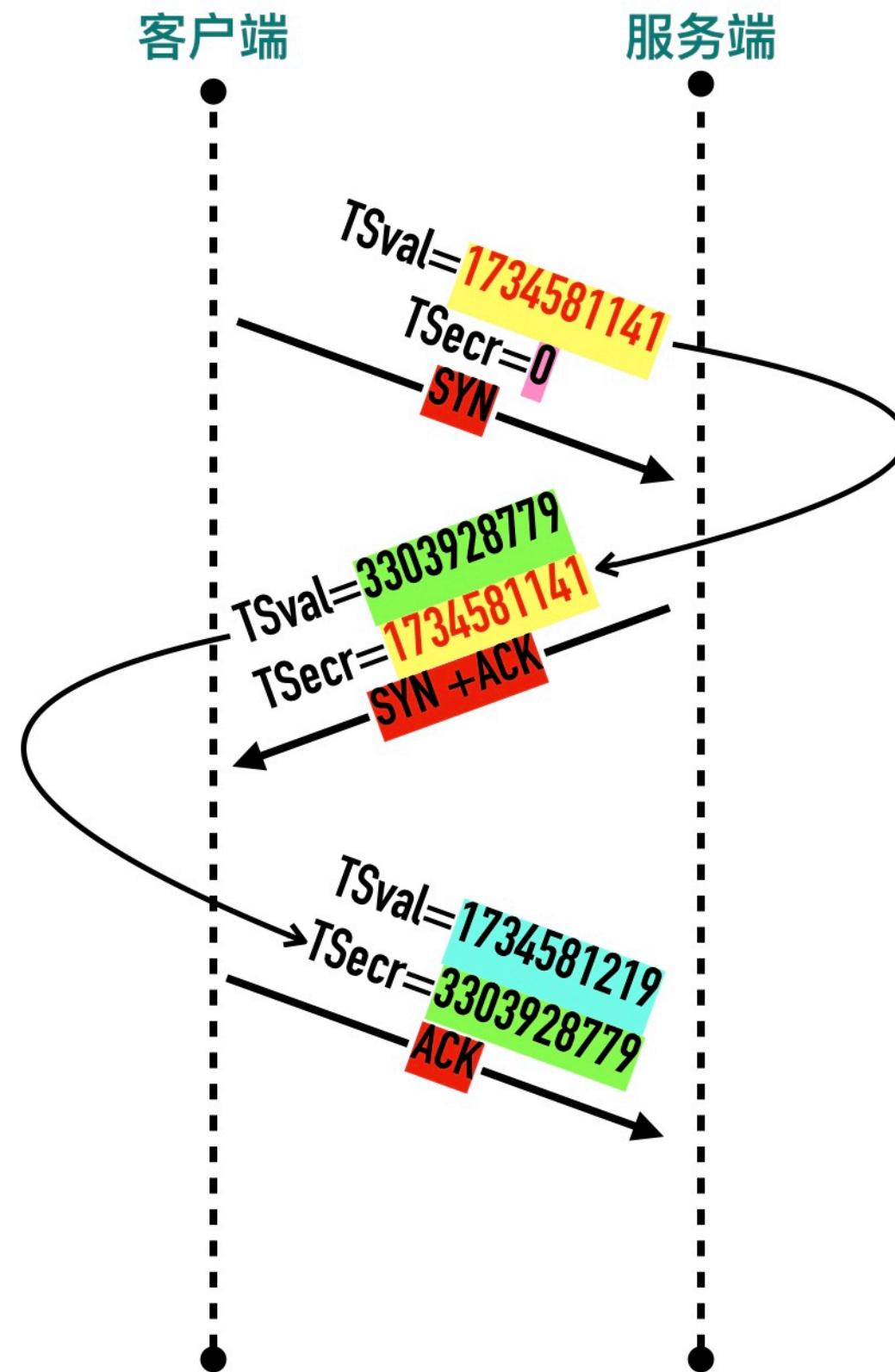
它由四部分构成：类别（kind）、长度（Length）、发送方时间戳（TS value）、回显时间戳（TS Echo Reply）。时间戳选项类别（kind）的值等于 8，用来与其它类型的选项区分。长度（length）等于 10。两个时间戳相关的选项都是 4 字节。

total = 10 bytes

1	1	4	4
类别 Kind=8	长度 Length=10	TS value (TSVal)	TS Echo Reply (TSecr)

是否使用时间戳选项是在三次握手里面的 SYN 报文里面确定的。下面的包是curl github.com抓包得到的结果。

Apply a display filter ... 					Expressi
No.	Time	Source	Destination	Info	
1	0.000000	10.0.0.7	13.229.188.59	50662 → 80 [SYN] Seq=2231862070 Win=65535 Len=0 MSS=1460 WS=64 TSval=1734581141 TSecr=0 SACK_PERM=1	
2	0.078164	13.229.188.59	10.0.0.7	80 → 50662 [SYN, ACK] Seq=418656693 Ack=2231862071 Win=28480 Len=0 MSS=1436 SACK_PERM=1 TSval=3303928779 TSecr=1734581141 WS	
3	0.078228	10.0.0.7	13.229.188.59	50662 → 80 [ACK] Seq=2231862071 Ack=418656694 Win=132416 Len=0 TSval=1734581219 TSecr=3303928779	
4	0.078284	10.0.0.7	13.229.188.59	GET / HTTP/1.1	
5	0.169766	13.229.188.59	10.0.0.7	HTTP/1.1 301 Moved Permanently	
6	0.169826	10.0.0.7	13.229.188.59	50662 → 80 [ACK] Seq=2231862145 Ack=418656778 Win=132288 Len=0 TSval=1734581310 TSecr=3303928801	
7	0.169913	10.0.0.7	13.229.188.59	50662 → 80 [FIN, ACK] Seq=2231862145 Ack=418656778 Win=132288 Len=0 TSval=1734581310 TSecr=3303928801	
8	0.260800	13.229.188.59	10.0.0.7	80 → 50662 [FIN, ACK] Seq=418656778 Ack=2231862146 Win=28672 Len=0 TSval=3303928823 TSecr=1734581310	
9	0.260868	10.0.0.7	13.229.188.59	50662 → 80 [ACK] Seq=2231862146 Ack=418656779 Win=132288 Len=0 TSval=1734581401 TSecr=3303928823	
10	30.309833	10.0.0.7	13.250.177.2...	50581 → 443 [ACK] Seq=189931929 Ack=1507509824 Win=2048 Len=0	
11	30.392125	13.250.177.2...	10.0.0.7	[TCP ACKed unseen segment] 443 → 50581 [ACK] Seq=1507509824 Ack=189931930 Win=34 Len=0 TSval=3303942436 TSecr=1734566307	
12	33.018686	10.0.0.7	13.250.177.2...	50582 → 443 [ACK] Seq=4013548056 Ack=2545276076 Win=2048 Len=0	
13	33.111367	13.250.177.2...	10.0.0.7	[TCP ACKed unseen segment] 443 → 50582 [ACK] Seq=2545276076 Ack=4013548057 Win=32 Len=0 TSval=3303937349 TSecr=1734568968	
14	45.232962	13.250.177.2...	10.0.0.7	[TCP ACKed unseen segment] [TCP Previous segment not captured] 443 → 50581 [FIN, ACK] Seq=1507509848 Ack=189931930 Win=34 Le	
15	45.232970	13.250.177.2...	10.0.0.7	[TCP ACKed unseen segment] [TCP Out-of-Order] 443 → 50581 [PSH, ACK] Seq=1507509824 Ack=189931930 Win=34 Len=24 TSval=330394	
16	45.233049	10.0.0.7	13.250.177.2...	[TCP Previous segment not captured] 50581 → 443 [ACK] Seq=189931930 Ack=1507509824 Win=2048 Len=0 TSval=1734625947 TSecr=330	
[Calculated window size: 65535]					
Checksum: 0x9167 [unverified]					
[Checksum Status: Unverified]					
Urgent pointer: 0					
▼ Options: (24 bytes), Maximum segment size, No-Operation (NOP), Window scale, No-Operation (NOP), No-Operation (NOP), Timestamps, SACK permitted, End of Option List (E					
▶ TCP Option - Maximum segment size: 1460 bytes					
▶ TCP Option - No-Operation (NOP)					
▶ TCP Option - Window scale: 6 (multiply by 64)					
▶ TCP Option - No-Operation (NOP)					
▶ TCP Option - No-Operation (NOP)					
▼ TCP Option - Timestamps: TSval 1734581141, TSecr 0					
Kind: Time Stamp Option (8)					
Length: 10					
Timestamp value: 1734581141					
Timestamp echo reply: 0					
▶ TCP Option - SACK permitted					
▶ TCP Option - End of Option List (EOL)					



有几个需要说明的点

- 时间戳是一个单调递增的值，与我们所知的 epoch 时间戳不是一回事。这个选项不要求两台主机进行时钟同步
- timestamps 是一个双向的选项，如果只要有一方不开启，双方都将停用 timestamps。比如下面是 curl www.baidu.com得到的包

Apply a display filter ... <⌘/>				
No.	Time	Source	Destination	Info
1	0.000000	10.0.0.7	183.232.231...	58960 → 80 [SYN] Seq=3881555073 Win=65535 Len=0 MSS=1460 WS=64 TSval=1737593452 TSecr=0 SACK_PERM=1
2	0.012829	183.232.231...	10.0.0.7	80 → 58960 [SYN, ACK] Seq=1294766963 Ack=3881555074 Win=8192 Len=0 MSS=1452 WS=32 SACK_PERM=1
3	0.012894	10.0.0.7	183.232.231...	58960 → 80 [ACK] Seq=3881555074 Ack=1294766964 Win=262144 Len=0
4	0.013027	10.0.0.7	183.232.231...	GET / HTTP/1.1
5	0.022267	183.232.231...	10.0.0.7	80 → 58960 [ACK] Seq=1294766964 Ack=3881555151 Win=24832 Len=0
6	0.022720	183.232.231...	10.0.0.7	80 → 58960 [ACK] Seq=1294766964 Ack=3881555151 Win=24832 Len=0 [TCP segment of 6 - ...]

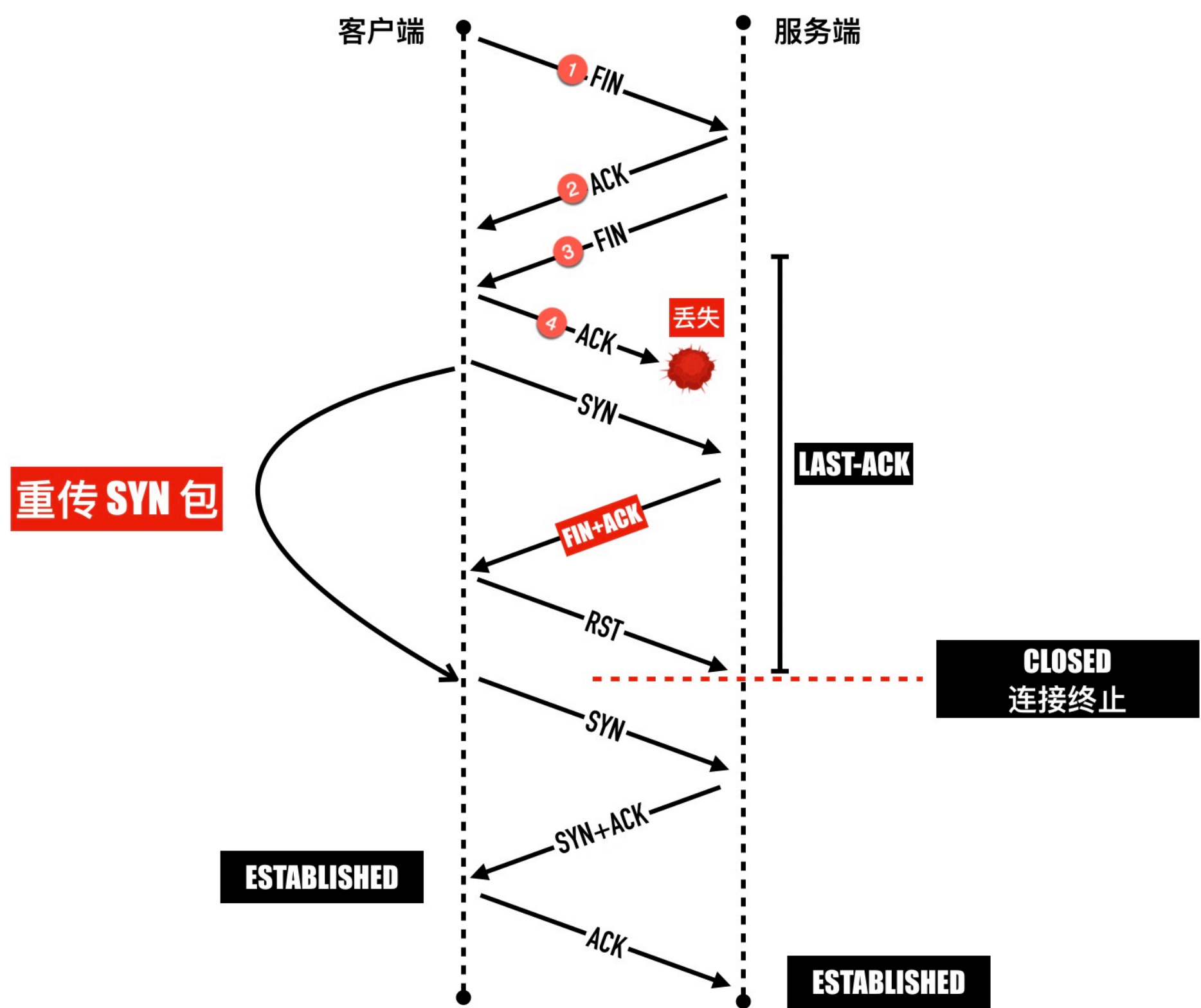
## tcp\_tw\_reuse 选项

缓解紧张的端口资源，一个可行的方法是重用“浪费”的处于 TIME\_WAIT 状态的连接，当开启 `net.ipv4.tcp_tw_reuse` 选项时，处于 TIME\_WAIT 状态的连接可以被重用。



下面把主动关闭方记为 A， 被动关闭方记为 B， 它的原理是：

- 如果主动关闭方 A 收到的包时间戳比当前存储的时间戳小，说明是一个迷路的旧连接的包，直接丢弃掉
- 如果因为 FIN 包丢失导致被动关闭方还处于LAST-ACK状态，这时 A 发送SYN 包想三次握手建立连接，这个时候处于 LAST-ACK 阶段的被动关闭方 B 会回复 FIN，因为这时 A 处于SYN-SENT阶段会回以一个 RST 包给 B，B 这端的连接会进入 CLOSED 状态，A 因为没有收到 SYN 包的 ACK，会重传 SYN，后面就一切顺利了。



## tcp\_tw\_recycle 选项

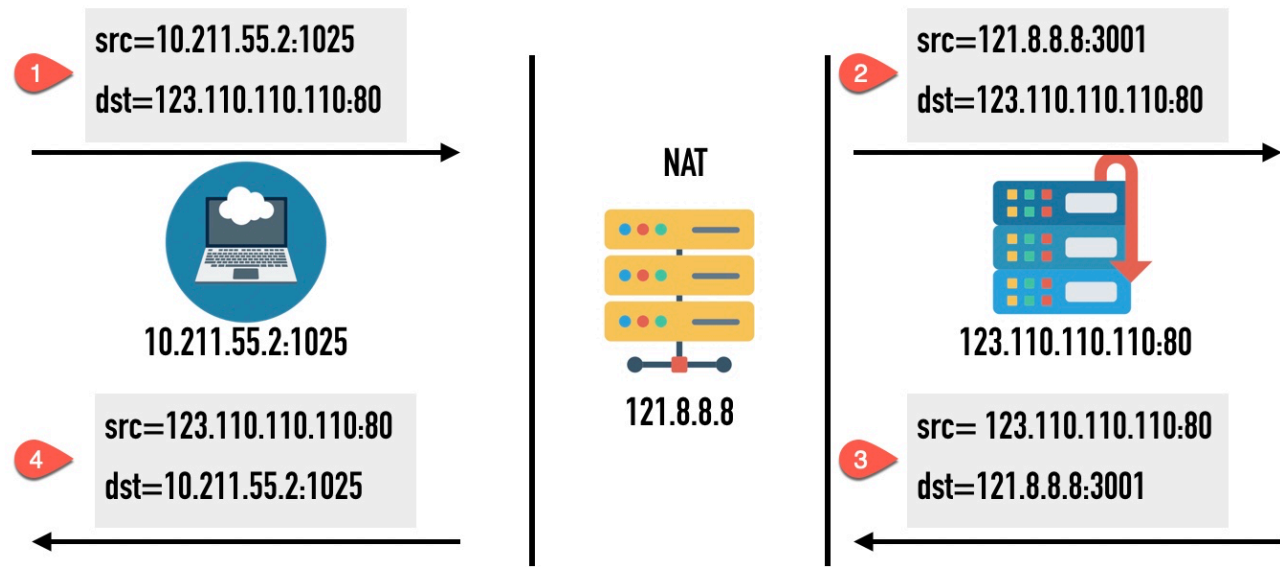
tcp\_tw\_recycle 是一个比 tcp\_tw\_reuse 更激进的方案，系统会缓存每台主机（即 IP）连接过来的最新的时间戳。对于新来的连接，如果发现 SYN 包中带的时间戳与之前记录的来自同一主机的同一连接的分组所携带的时间戳相比更旧，则直接丢弃。如果更新则接受复用 TIME-WAIT 连接。

这种机制在客户端与服务端一对一的情况下没有问题，如果经过了 NAT 或者负载均衡，问题就很严重了。



# 什么是 NAT呢？

NAT（Network Address Translator）的出现是为了缓解 IP 地址耗尽的临时方案，IPv4 的地址是 32 位，全部利用最多只能提 42.9 亿个地址，去掉保留地址、组播地址等剩下的只有 30 多亿



内网		外网	
IP	端口	IP	端口
10.211.55.2	1025	121.8.8.8	3001
10.211.55.5	1116	121.8.8.8	34565
...	...	...	...

# NAT 优缺点

它有两个明显的优点：

- 出口 IP 共享：通过一个公网地址可以让许多机器连上网络，解决 IP 地址不够用的问题
- 安全隐私防护：实际的机器可以隐藏自己真实的 IP 地址

当然也有明显的弊端：NAT 会对包进行修改，有些协议无法通过 NAT。

## 当 tcp\_tw\_recycle 遇上 NAT

当 tcp\_tw\_recycle 遇上 NAT 时，因为客户端出口 IP 都一样，会导致服务端看起来都在跟同一个 host 打交道。不同客户端携带的 timestamp 只跟自己相关，如果一个时间戳较大的客户端 A 通过 NAT 与服务器建连，时间戳较小的客户端 B 通过 NAT 发送的包服务器认为是过期重复的数据，直接丢弃，导致 B 无法正常建连和发数据。

# 小结

TIME\_WAIT 状态是最容易造成混淆的一个概念，这个状态存在的意义是

- 可靠的实现 TCP 全双工的连接终止（处理最后 ACK 丢失的情况）
- 避免当前关闭连接与后续连接混淆（让旧连接的包在网络中消逝）

## 思考题

假设 MSL 是 60s, 请问系统能够初始化一个新连接然后主动关闭的最大速率是多少? (忽略1~1024区间的端口)