网络基础之以太网帧,MTU,MSS

mnstory.net

以太网帧

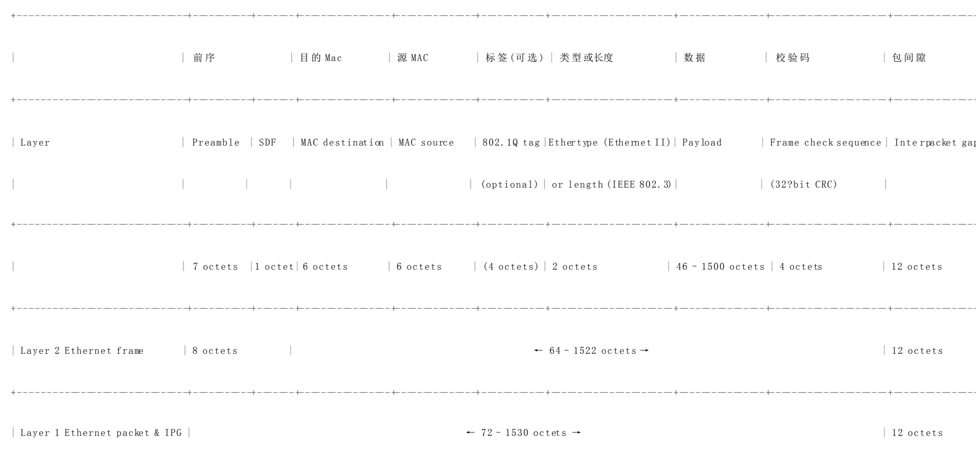
网络数据传输，底层是物理设备的支持，如路由、交换、网卡、网线，通过网线传送电信号或者光信号，自然可以"连续"地传输，但是，考虑到连续传输，如果中间某个信号被干扰，一个 bit 不正确会导致整个串都不正确，损失颇大；其次，考虑到线路的多源共享问题，如果一直传送一个源的数据，其他的数据就得不到发送。于是，聪明的设计者将来自一个源的大段数据，拆分成小段，如果某一小段传输不正确，只影响这一小段，而非整体，根据一定的算法交替发送不同来源的数据，可实现线路共享，而分割的这一小段也就是我们说的帧(Frame)，帧也称为协议数据单元(PDU)。

由于帧是逻辑单元，不同的人根据不同需求来指定帧的格式，肯定有所不同，好在目前帧的格式不是很多，具体可参考 https://wenku.baidu.com/view/25f6f019964bcf84b9d57bb9.html

简单划分，可以划分为两种：

1. IEEE 802.3 以太网标准，已经过多次更新以便将新技术纳入其中。
2. DIX 以太网标准，现在称其为以太网 II(Ethernet II)，是 TCP/IP 网络中使用的以太网帧格式。

两种标准之间的差异很小，可以用一个格式统一表达，参考 https://en.wikipedia.org/wiki/Ethernet_frame ：



	前序		目的 Mac	源 MAC	标签(可选)	类型或长度	数据	校验码	包间隙
Layer	Preamble	SDF	MAC destination	MAC source	802.1Q tag (optional)	Ethertype (Ethernet II) or length (IEEE 802.3)	Payload	Frame check sequence (32?bit CRC)	Interpacket gap
	7 octets	1 octet	6 octets	6 octets	(4 octets)	2 octets	46 - 1500 octets	4 octets	12 octets
Layer 2 Ethernet frame	8 octets		← 64 - 1522 octets →						12 octets
Layer 1 Ethernet packet & IPG	← 72 - 1530 octets →								12 octets

差异表现在:

1. 前序部分,802.3 拆分为 Preamble(7 octets)和 SFD(Start of frame delimiter, 1 octet)；Ethernet II 只有 Preamble(8 octets)。
2. 对 802.3 来说,类型或长度部分,表示的是长度,其值小于等于 1500;对 Ethernet II 来说，表示的是类型，其值大于 1500，通过值大小即可区分两种帧。

通过上图，我们可以看到，以太网帧主要由三部分组成：

1. 数据 Payload 部分(46-1500 octets)。
2. 以太网协议本身占用(18-22 octets)，包括：目的 Mac(6 octets)、源 MAC(6 octets)、标签(可选)(4 octets)、类型或长度(2 octets)、校验码(4 octets)。
3. 以太网协议填充占用(20 octets)，包括：前序(8 octets)、包间隙(12 octets)。

Payload 部分，即为每个帧能负载的字节大小。上图表显示 Payload 为[46, 1500]，但实际测验，低于 46 bytes 的包也可以传送，不知底层是否做了数据填充：

```
# ping -s 0 -M do 11.10.4.2
PING 11.10.4.2 (11.10.4.2) 0(28) bytes of data.
8 bytes from 11.10.4.2: icmp_seq=1 ttl=64
8 bytes from 11.10.4.2: icmp_seq=2 ttl=64
```

```
No.  Time             Source      Destination  Protocol  Length  Info
  1  10:27:00.512455  11.10.4.1   11.10.4.2    ICMP      42      Echo (ping) request  id=0x1526, seq=6/1536, ttl=64 (reply in 2)
  2  10:27:00.512927  11.10.4.2   11.10.4.1    ICMP      42      Echo (ping) reply    id=0x1526, seq=6/1536, ttl=64 (request in 1)

Frame 1: 42 bytes on wire (336 bits), 42 bytes captured (336 bits)
Ethernet II, Src: fe:fc:fe:2a:6f:05 (fe:fc:fe:2a:6f:05), Dst: fe:fc:fe:93:5d:87 (fe:fc:fe:93:5d:87)
Internet Protocol Version 4, Src: 11.10.4.1, Dst: 11.10.4.2
Internet Control Message Protocol

0000  fe fc fe 93 5d 87 fe fc  fe 2a 6f 05 08 00 45 00   ....]... .*o...E.
0010  00 1c 00 00 40 00 40 01  1c cb 0b 0a 04 01 0b 0a   ....@.@. ........
0020  04 02 08 00 e2 d3 15 26  00 06                     .......& ..
```

如上图，整个帧的 Payload 部分负载只有 28 bytes(42-14)，而非 46 bytes。
注： 我们用 ping -s SIZE -M do 来发送指定大小的不分片包，理论上，SIZE 的最大值为 = 路径最小 MTU - IP 头部分(20 bytes) - ICMP 头部分(8 bytes)。

通常，计算以太网帧的大小，需要算上太网协议本身占用部分，于是，以太网帧的大小范围应该是[64=46+18, 1522=1500+22]，其实最开始 EthernetII 和 802.3 规定最大帧为 1518，后来为支持 vlan，1998 年 802.3ac 规定最大帧为 1518+4(vlan)=1522。

先说说，这最小值 64 bytes 是怎么来的。
以太网通过载波侦听多路访问（CSMA）技术控制节点共享访问，CSMA 检查介质是否正在传送信号，如果介质上检测到来自另一节点的载波信号，则表示另一设备正在进行传输，这时候会等待，如果没有检查到，会立即传送，但是，也可能检查误差或故障，两设备同时传输，这时候变产生了数据冲突，数据冲突后需要重传这串数据。一条一定长度的线缆，在一串数据发送过程中，如果冲突，需要反馈 JAM 信号至发送端，这个动作完成前，至少应该这个串的发送过程还没结束，不然到底是哪条数据冲突了都不知道，就没法重传，而个检查反馈的时间，算下来在 10Mbps 的带宽情况下，刚好需要连续发送 64 octets 的数据才不至于错过时间。
不过，现在的 LAN 中设备之间几乎所有有线连接都是全双工连接，这类冲突已经不存在。

MTU

MTU(Maximum Transmission Unit)指的是以太网帧能携带的最大 Payload 大小，上面已经说过了，是[46, 1500] bytes。

而我们常常设置的，有人叫 IP MTU，不过大多数的时候，并没有严格区分。
IP MTU，其确切含义是：一个 IP 报文不分片能通过某介质(如当前网卡)的最大值。IP 报文头部有一个 bit(DF, Don't Fragment flag)表示不分片，不分片的报文因为大小限制，可能过不中间的交换设备，这时候会发送 ICMP 错误回来，设置 DF 的好处是，路径上的转发设备不再需要分包重组，对连续发送来说，可提高转发效率。

理论上 MTU 最小值可以是 Payload 的最小值 46 bytes，而你设置 IP MTU 的时候，并不能设置 46 bytes。

先说 IPv4，实际上 IP MTU 最小值(能设置的值)为 68 bytes：
ifconfig eth0 mtu 68
ifconfig eth0 mtu 67
SIOCSIFMTU: Invalid argument
注：也可以通过 /sys/class/net/eth0/mtu 查看或设置 IP MTU。

之所以不能设置为 46，是因为 IPv4 协议有自己的最小值规定，参考 https://tools.ietf.org/html/rfc791
Every internet module must be able to forward a datagram of 68 octets without further fragmentation. This is because an internet header may be up to 60 octets, and the minimum fragment is 8 octets.
其实就是，一个 IP 报文，不管怎么分片，头部不能分片，既然头部的最大值可能为 60 bytes，再加上最小分片 8 bytes = 68 bytes，所以，IP MTU 最小值为 68 bytes(最大值是 65535 bytes)。

IP MTU 还有一个建议值，不一定是最大值，只是从理论、历史、网络环境因素分析，这个值在普遍网络环境下比较适合，比如，IPv4 的建议值是 1500，前面描述了关于以太网帧结构，应该很清楚了，这 1500 就是以太网帧的 Payload 最大值。
网上也有人给出 1500 的理论分析，因为 1542(1522+12+8)这个值，其搬包效率非常高为 1500/1542=97.28%，再权衡链路共享，以太网帧校验，延迟，重传长度等因素，就选了这个值，也有一个比较有趣的回答是：
I asked Bob Metcalfe where 1518 bytes comes from. Seriously, I met him at a party many years ago and hit him with this question. His answer, "Hmm, well I really don't know!"
参考 http://www.mail-archive.com/cisco@groupstudy.com/msg24534.html

既然特别强调了是 IPv4 协议，那就还有它的升级版，IPv6。
对 IPv6 而言，它的最小值是 1280 bytes(最大值是 65535 bytes，通过 jumbogram 可以支持 4GB 大小)，请参考 https://tools.ietf.org/html/rfc2460
IPv6 requires that every link in the internet have an MTU of 1280 octets or greater. On any link that cannot convey a 1280-octet packet in one piece, link-specific fragmentation and reassembly must be provided at a layer below IPv6.

测试一下 MTU 修改效果：

ifconfig eth0 mtu 1000
ifconfig eth0 | grep mtu
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1000
ping -s 972 -M do 11.10.5.2
PING 11.10.5.2 (11.10.5.2) 972(1000) bytes of data.
980 bytes from 11.10.5.2: icmp_seq=1 ttl=64 time=0.196 ms
980 bytes from 11.10.5.2: icmp_seq=2 ttl=64 time=0.717 ms
ping -s 973 -M do 11.10.5.2
PING 11.10.5.2 (11.10.5.2) 973(1001) bytes of data.
ping: local error: Message too long, mtu=1000
ping: local error: Message too long, mtu=1000

MSS

先说一下 MSS(Maximum Segment Size)最大分段大小，其实是 TCP 协议层的一个概念，和 IP MTU 约束 IP 报文大小类似，MSS 约束的是 TCP 分段大小。在 TCP 握手的时候可以指定 MSS，根据协议可以算出，默认的 MSS 为：MTU(默认 1500 bytes)-IP 头(20 bytes)-TCP 头(20 bytes) = 1460 bytes。

```
No.  Time             Source        Destination   Protocol Length Info
   1 15:20:10.815936  11.10.4.1     10.78.51.48   TCP      74 59938 → 22 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1
   2 15:20:10.815986  10.78.51.48   11.10.4.1     TCP      74 22 → 59938 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 S
   3 15:20:10.816735  11.10.4.1     10.78.51.48   SSHv2    98 Client: Protocol (SSH-2.0-OpenSSH_7.3p1-hpn14v11)

▷ Frame 1: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface 0
▷ Ethernet II, Src: 0a:58:0a:4e:33:01 (0a:58:0a:4e:33:01), Dst: 0a:58:0a:4e:33:30 (0a:58:0a:4e:33:30)
▷ Internet Protocol Version 4, Src: 11.10.4.1, Dst: 10.78.51.48
◢ Transmission Control Protocol, Src Port: 59938, Dst Port: 22, Seq: 0, Len: 0
    Source Port: 59938
    Destination Port: 22
    [Stream index: 0]
    [TCP Segment Len: 0]
    Sequence number: 0    (relative sequence number)
    Acknowledgment number: 0
    Header Length: 40 bytes
  ▷ Flags: 0x002 (SYN)
    Window size value: 29200
    [Calculated window size: 29200]
    Checksum: 0xd744 [unverified]
    [Checksum Status: Unverified]
    Urgent pointer: 0
  ◢ Options: (20 bytes), Maximum segment size, SACK permitted, Timestamps, No-Operation (NOP), Window scale
    ◢ Maximum segment size: 1460 bytes
        Kind: Maximum Segment Size (2)
        Length: 4
        MSS Value: 1460
    ▷ TCP SACK Permitted Option: True
    ▷ Timestamps: TSval 13282310, TSecr 0
    ▷ No-Operation (NOP)
    ▷ Window scale: 7 (multiply by 128)
```

可以通过 iptables 设置 MSS，例如：

iptables -t mangle -I FORWARD -p tcp --tcp-flags SYN,RST,ACK SYN -j TCPMSS --set-mss 1360   # SYNC 包

iptables -t mangle -I FORWARD -p tcp --tcp-flags SYN,RST,ACK SYN,ACK -j TCPMSS --set-mss 1360 # SYNC ACK 包

如果 MSS 设置超过 MTU，路径上可能返回类似错误：

tcpdump -i eth0 icmp -nn
10:20:00.986208 IP 11.10.1.2 > 11.10.1.1: ICMP 172.34.0.2 unreachable - need to frag (mtu 1500), length 556

返回类似错误，不一定代表网络不通，小包是没有问题的，UDP 一般不受影响，只是 TCP 表现得性能下降数倍到数十倍，当然，类似现象也有可能是由于分片不当导致，我在 QEMU 虚拟化环境下(virtio-net)，遇到过此错误(icmp unreachable)，TCP 流里面很多重传包， Dup Ack， Out-Of-Order 包：

Time	Source	Destination	Protocol	Length	Info
2017-06-13 15:20:31.206571	10.78.51.48	11.10.4.1	TCP	66	22 → 59938 [ACK] Seq=46457 Ack=135357465 Win=174336 Len=0 TSval=32089466 TSecr=13302701
2017-06-13 15:20:31.206659	11.10.4.1	10.78.51.48	SSHv2	1514	Client: Encrypted packet (len=1448)
2017-06-13 15:20:31.206676	10.78.51.48	11.10.4.1	TCP	66	22 → 59938 [ACK] Seq=46457 Ack=135358913 Win=182656 Len=0 TSval=32089466 TSecr=13302701
2017-06-13 15:20:31.206760	11.10.4.1	10.78.51.48	SSHv2	1514	Client: [TCP Previous segment not captured] , Encrypted packet (len=1448)
2017-06-13 15:20:31.206769	10.78.51.48	11.10.4.1	TCP	78	[TCP Dup ACK 152041#1] 22 → 59938 [ACK] Seq=46457 Ack=135358913 Win=182656 Len=0 TSval=32089466 TSecr=13302701 SLE=135361809
2017-06-13 15:20:31.206860	11.10.4.1	10.78.51.48	SSHv2	1514	Client: Encrypted packet (len=1448)
2017-06-13 15:20:31.206871	10.78.51.48	11.10.4.1	TCP	78	[TCP Dup ACK 152041#2] 22 → 59938 [ACK] Seq=46457 Ack=135358913 Win=182656 Len=0 TSval=32089467 TSecr=13302701 SLE=135361809
2017-06-13 15:20:31.206977	11.10.4.1	10.78.51.48	SSHv2	1514	Client: [TCP Fast Retransmission] , Encrypted packet (len=1448)
2017-06-13 15:20:31.206990	10.78.51.48	11.10.4.1	TCP	78	22 → 59938 [ACK] Seq=46457 Ack=135360361 Win=181248 Len=0 TSval=32089467 TSecr=13302701 SLE=135361809 SRE=135364705
2017-06-13 15:20:31.207092	11.10.4.1	10.78.51.48	SSHv2	1514	Client: Encrypted packet (len=1448)
2017-06-13 15:20:31.207102	10.78.51.48	11.10.4.1	TCP	78	[TCP Dup ACK 152047#1] 22 → 59938 [ACK] Seq=46457 Ack=135360361 Win=181248 Len=0 TSval=32089467 TSecr=13302701 SLE=135361809
2017-06-13 15:20:31.207235	11.10.4.1	10.78.51.48	TCP	1514	[TCP Out-Of-Order] 59938 → 22 [ACK] Seq=135360361 Ack=46457 Win=41216 Len=1448 TSval=13302701 TSecr=32089467
2017-06-13 15:20:31.207239	11.10.4.1	10.78.51.48	SSHv2	1514	Client: Encrypted packet (len=1448)
2017-06-13 15:20:31.207254	10.78.51.48	11.10.4.1	TCP	66	22 → 59938 [ACK] Seq=46457 Ack=135366153 Win=175744 Len=0 TSval=32089467 TSecr=13302701
2017-06-13 15:20:31.207256	10.78.51.48	11.10.4.1	TCP	66	22 → 59938 [ACK] Seq=46457 Ack=135367601 Win=174336 Len=0 TSval=32089467 TSecr=13302701
2017-06-13 15:20:31.207346	11.10.4.1	10.78.51.48	SSHv2	1514	Client: Encrypted packet (len=1448)
2017-06-13 15:20:31.207364	10.78.51.48	11.10.4.1	TCP	66	22 → 59938 [ACK] Seq=46457 Ack=135369049 Win=182656 Len=0 TSval=32089467 TSecr=13302701
2017-06-13 15:20:31.207461	11.10.4.1	10.78.51.48	SSHv2	1514	Client: [TCP Previous segment not captured] , Encrypted packet (len=1448)
2017-06-13 15:20:31.207471	10.78.51.48	11.10.4.1	TCP	78	[TCP Dup ACK 152055#1] 22 → 59938 [ACK] Seq=46457 Ack=135369049 Win=182656 Len=0 TSval=32089467 TSecr=13302701 SLE=135371945
2017-06-13 15:20:31.207627	11.10.4.1	10.78.51.48	TCP	66	59938 → 22 [RST, ACK] Seq=135373393 Ack=46457 Win=41216 Len=0 TSval=0 TSecr=32089467
2017-06-13 15:20:31.207637	10.78.51.48	11.10.4.1	TCP	78	[TCP Dup ACK 152055#2] 22 → 59938 [ACK] Seq=46457 Ack=135369049 Win=182656 Len=0 TSval=32089467 TSecr=13302701 SLE=135371945
2017-06-13 15:20:31.207720	11.10.4.1	10.78.51.48	TCP	54	59938 → 22 [RST] Seq=135369049 Win=0 Len=0

原因并非是 MSS 和 MTU 不匹配，而是大包发送端启用了 TSO，需要：

```
ethtool -K eth0 tso off
```

然后，我又遇到一次在 QEMU 虚拟化环境下(virtio-net)，这次没有 icmp 错误返回，但是数据报文正常和异常对比是这样的：

No.	Time	Source	Destination	Protocol	Length	Info
90	09:19:14.800425	28.28.0.1	28.255.255.203	TCP	60	56583→31384 [SYN] Seq=0 Win=32768 Len=0 MSS=1460
91	09:19:14.801220	28.255.255.203	28.28.0.1	TCP	58	31384→56583 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460
92	09:19:14.801245	28.28.0.1	28.255.255.203	TCP	60	56583→31384 [ACK] Seq=1 Ack=1 Win=32768 Len=0
93	09:19:14.801293	28.28.0.1	28.255.255.203	HTTP	143	GET /plugin.php?id=info:index HTTP/1.1
94	09:19:14.802237	28.255.255.203	28.28.0.1	TCP	54	31384→56583 [ACK] Seq=1 Ack=90 Win=29200 Len=0
97	09:19:15.111375	28.255.255.203	28.28.0.1	TCP	1514	[TCP segment of a reassembled PDU]
98	09:19:15.111420	28.255.255.203	28.28.0.1	TCP	1514	[TCP segment of a reassembled PDU]
99	09:19:15.111440	28.28.0.1	28.255.255.203	TCP	60	56583→31384 [ACK] Seq=90 Ack=2921 Win=32768 Len=0
100	09:19:15.111478	28.255.255.203	28.28.0.1	TCP	1514	[TCP segment of a reassembled PDU]
101	09:19:15.111482	28.255.255.203	28.28.0.1	TCP	1514	[TCP segment of a reassembled PDU]
102	09:19:15.111498	28.28.0.1	28.255.255.203	TCP	60	56583→31384 [ACK] Seq=90 Ack=5841 Win=32768 Len=0

trace0.pcap [Wireshark 1.12.4 (v1.12.4-0-gb4861da from master-1.12)]

File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help

Filter: tcp.stream eq 0 Expression... Clear Apply Save

No.	Time	Source	Destination	Protocol	Length	Info
89	10:20:46.763971	28.28.0.1	28.255.255.203	TCP	60	33484→31384 [SYN] Seq=0 Win=32768 Len=0 MSS=1460
90	10:20:46.764301	28.255.255.203	28.28.0.1	TCP	58	31384→33484 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460
91	10:20:46.764342	28.28.0.1	28.255.255.203	TCP	60	33484→31384 [ACK] Seq=1 Ack=1 Win=32768 Len=0
92	10:20:46.764438	28.28.0.1	28.255.255.203	HTTP	143	GET /plugin.php?id=info:index HTTP/1.1
93	10:20:46.764711	28.255.255.203	28.28.0.1	TCP	54	31384→33484 [ACK] Seq=1 Ack=90 Win=29200 Len=0
96	10:20:47.036631	28.255.255.203	28.28.0.1	TCP	10176	[TCP segment of a reassembled PDU]
97	10:20:47.244205	28.255.255.203	28.28.0.1	TCP	1514	[TCP Retransmission] 31384→33484 [ACK] Seq=1 Ack=90 Win=29200 Len=1460
101	10:20:47.444258	28.28.0.1	28.255.255.203	TCP	60	33484→31384 [ACK] Seq=90 Ack=1461 Win=32768 Len=0
102	10:20:47.444758	28.255.255.203	28.28.0.1	TCP	2974	[TCP Previous segment not captured] 31384→33484 [ACK] Seq=14601 Ack=90 Win=29200 Len=2920
106	10:20:47.858835	28.255.255.203	28.28.0.1	TCP	1514	[TCP Retransmission] [TCP segment of a reassembled PDU]
109	10:20:48.058895	28.28.0.1	28.255.255.203	TCP	60	33484→31384 [ACK] Seq=90 Ack=2921 Win=32768 Len=0
110	10:20:48.066563	28.255.255.203	28.28.0.1	TCP	2974	31384→33484 [ACK] Seq=17521 Ack=90 Win=29200 Len=2920
115	10:20:48.893703	28.255.255.203	28.28.0.1	TCP	1514	[TCP Retransmission] 31384→33484 [ACK] Seq=2921 Ack=90 Win=29200 Len=1460
118	10:20:49.093751	28.28.0.1	28.255.255.203	TCP	60	33484→31384 [ACK] Seq=90 Ack=4381 Win=32768 Len=0

可以看出，异常是从传送了一个大包开始的，虽然指定了 MTU，虽然指定了 MSS，虽然关闭了 tso，然而并没什么卵用，他还是发出了一个大包，最后发现，问题出在回包端的的另外一个参数 scatter-gather：

```
node-5441045654772 ~ # ethtool -K eth1 sg on
Actual changes:
scatter-gather: on
        tx-scatter-gather: on
tcp-segmentation-offload: on
        tx-tcp-segmentation: on
        tx-tcp6-segmentation: on
generic-segmentation-offload: on
node-5441045654772 ~ # ethtool -K eth1 sg off
Actual changes:
scatter-gather: off
        tx-scatter-gather: off
tcp-segmentation-offload: off
        tx-tcp-segmentation: off [requested on]
        tx-tcp6-segmentation: off [requested on]
generic-segmentation-offload: off [requested on]
```

sg 参数和 tso,gso 有关联，关闭的时候，都关闭，开启的时候，都开启。

2017/7/5