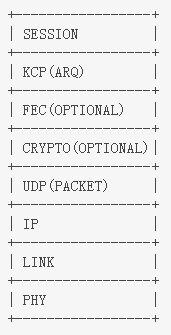
KCP 快速可靠网络传输协议及其与TCP的比较

**1 KCP简介**

KCP协议是一种快速可靠传输ARQ(Automatic Repeat-reQuest)协议，能以比TCP浪费10%-20%的带宽的代价，换取平均延迟降低 30%-40%，且最大延迟降低三倍的传输效果。它跟QUIC协议一样也是基于UDP协议的实现。

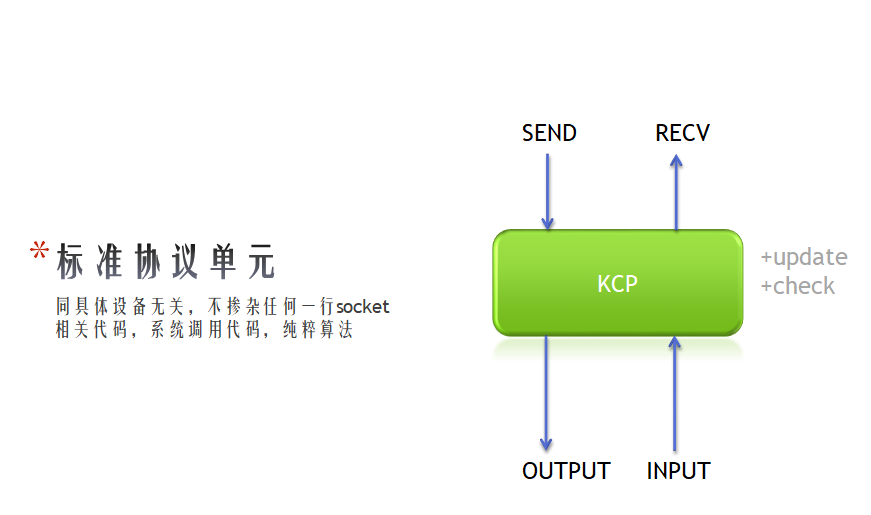
只有在弱网络条件下，KCP相比TCP才有优势。

**1.1 KCP协议在网络分层模型的位置：**

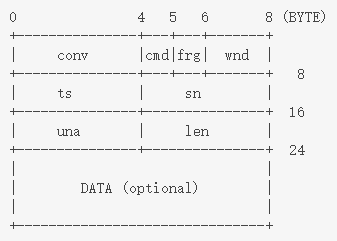


KCP的设计者有意地把KCP依赖的网络通讯进行了解耦。

*“纯算法实现，并不负责底层协议（如UDP）的收发，需要使用者自己定义下层数据包的发送方式，以 callback的方式提供给 KCP。“*



**1.2 KCP协议结构**



**字段含义：**

**conv**: 连接号，UDP是无连接的，conv用于表示来自于哪个客户端。对连接的一种替代, 因为有conv, 所以KCP也是支持多路复用的；

**cmd**: 命令类型，如IKCP\_CMD\_PUSH，IKCP\_CMD\_ACK，IKCP\_CMD\_WASK IKCP\_CMD\_WINS 等；

**frg**: 分片，用户数据可能会被分成多个KCP包并发送出去；

**wnd**: 接收窗口大小，发送方的发送窗口不能超过接收方给出的数值, （其实是接收窗口的剩余大小，这个大小是动态变化的)；

**ts**: 时间序列；

**sn**: 序列号；

**una**: 下一个可接收的序列号，如收到sn=10的包时，una就变为11；

**len**: 数据长度(DATA的长度)；

**data**: 用户数据。

**1.3 使用场景**

适用于需要高速传输环境或对网络实时性和可靠性要求比较高的应用，例如实时pvp游戏（moba等）和视频加速程序等等。

**2 与TCP的对比**

TCP是为流量设计的（每秒内可以传输多少KB的数据），讲究的是充分利用带宽。而 KCP是为流速设计的（单个数据包从一端发送到一端需要多少时间）。TCP信道是一条流速很慢，但每秒流量很大的大运河，而KCP是水流湍急的小激流。KCP有正常模式和快速模式两种，通过以下策略达到提高流速的结果：

**RTO翻倍 vs 不翻倍：**

TCP超时计算是RTOx2，这样连续丢三次包就变成RTOx8了，十分影响传输速度，而KCP启动快速模式后RTO不x2，只x1.5（实验证明1.5这个值相对较好），从而提高了传输速度。

**选择性重传 vs 全部重传：**

TCP丢包时会全部重传丢失的包开始以后的数据，而KCP是选择性重传，只重传真正丢失的数据包。

**快速重传：**

比如发送端发送了1,2,3,4,5几个包，然后收到远端的ACK: 1, 3, 4, 5，当收到ACK3时，KCP知道2被跳过1次，收到ACK4时，知道2被跳过了2次，此时可以认为2号丢失，不用等超时，直接重传2号包，大大改善了丢包时的传输速度。

**延迟ACK vs 非延迟ACK：**

TCP为了充分利用带宽，会延迟发送ACK，这样超时计算会算出较大 RTT时间，延长了丢包时的判断过程。KCP中的ACK是否延迟发送可以调节。

**UNA vs ACK+UNA：**

ARQ模型响应有两种，UNA（此编号前所有包已收到，如TCP）和ACK（该编号包已收到），仅用UNA将导致全部重传，而仅用ACK则会导致丢失成本太高，以往协议都是二选其一，而在KCP协议中，除单独的 ACK包外，所有包都含有UNA信息。

**非退让流控：**

KCP正常模式同TCP一样使用公平退让法则，即发送窗口大小由：发送缓存大小、接收端剩余接收缓存大小、丢包退让及慢启动这四要素决定。但传送及时性要求很高的小数据时，可选择通过配置跳过后两步，仅用前两项来控制发送频率，牺牲部分公平性及带宽利用率，换取了流畅传输的效果。



**3 数据发送/接收流程**

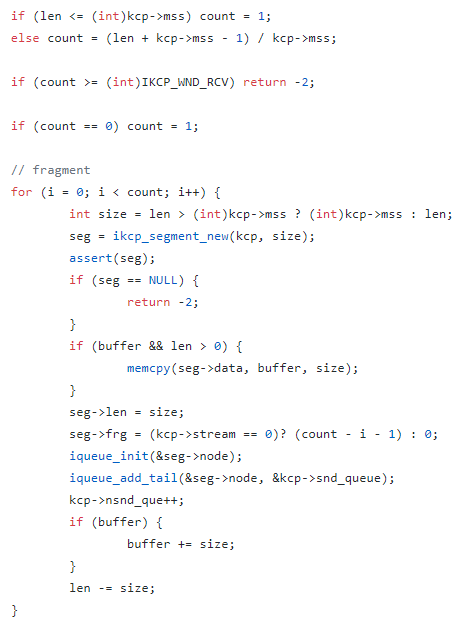
**3.1 发送流程**

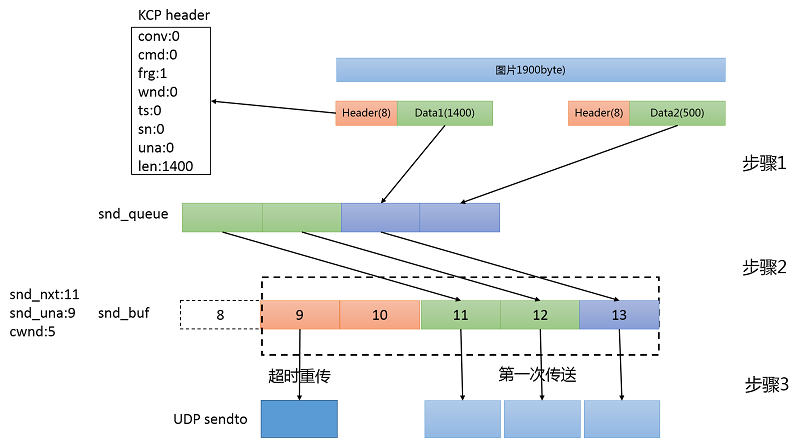
上层逻辑 -> 上层逻辑管理的buffer -> **snd\_queue** -> **snd\_buf** -> output …

**步骤1：**上层逻辑调用ikcp\_send()函数，将需要发送的数据进行分片（即拆分为segment），然后添加入snd\_queue；

添加入snd\_queue

计算分片数量

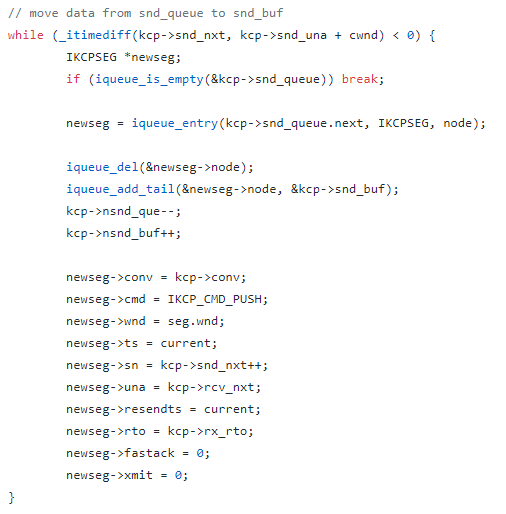


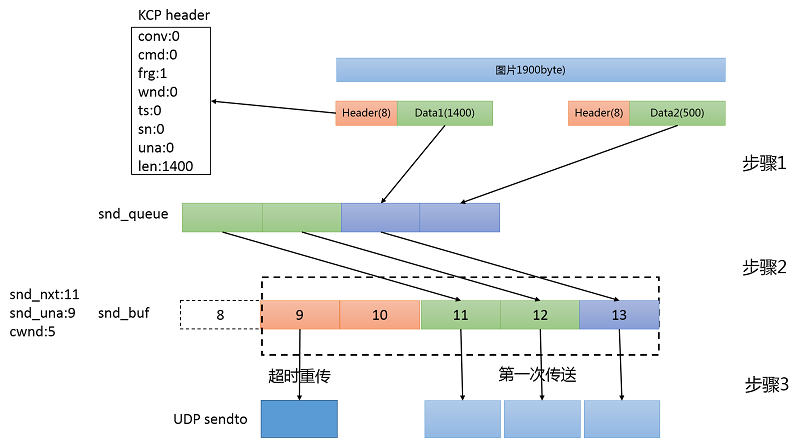


**步骤2：**调用ikcp\_flush()函数，将snd\_queue中的segment转移至snd\_buf中；

从snd\_queue转移至snd\_buf

当窗口还有空闲位置

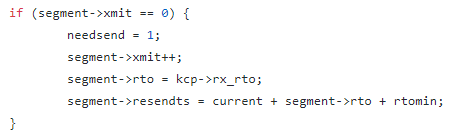




**步骤3：**经过上述步骤后，ikcp\_flush()函数中会对snd\_buf中的segment进行遍历，根据一定的策略决定某一个segment是否需要发送；

1. 对于从未被发送过的segment（xmit == 0），直接判断为需要发送：

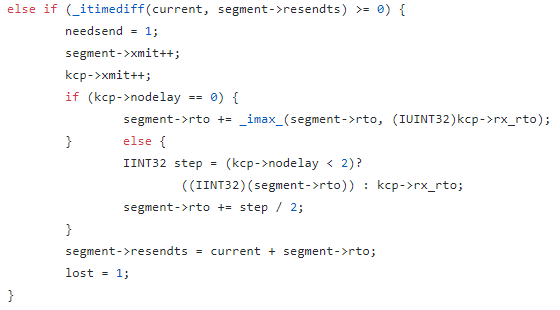
计算超时重传时间



1. 对于已发送过，但等待ACK超时的segment （current >= resendts），认为已丢包，需要重新发送（实现选择性重传），另外还会对RTO进行调整（急速模式下+0.5 RTO）：

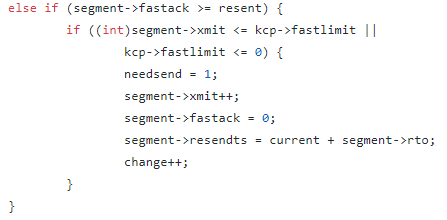
等待超时

RTO×1.5



1. 对于已发送过，且等待ACK还未超时的segment，如果其被跨越ACK的次数fastack（任何sn比它大的segment接收到ACK的计数）超过设定的阈值resent，会触发快速重传机制：

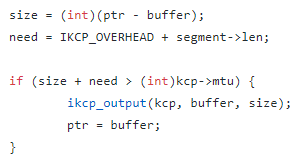
判断是否超出预设的重传次数限制



**步骤4：**数据发送，即调用ikcp\_output()函数发送在上述步骤中判断为需要发送的segment，此过程中会对比较小的segment进行合并，一次性发送提高效率。

segment大小

缓存中已有的数据大小



另外在C/C++版本的源码中，如果某个segment被发送过的次数xmit超过设定的阈值dead\_link，则会判定此时连接状态已断开。

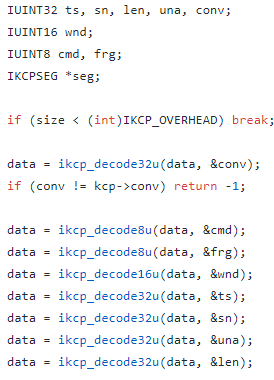
**3.2 接收流程**

KCP的接收过程是将UDP收到的数据进行解包，重新组装顺序的、可靠的数据后交付给用户。

… input -> **rcv\_buf** -> **rcv\_queue** -> 上层逻辑管理的buffer -> 上层逻辑

**步骤1：**调用ikcp\_input()函数，将接收到的数据解包；

解包

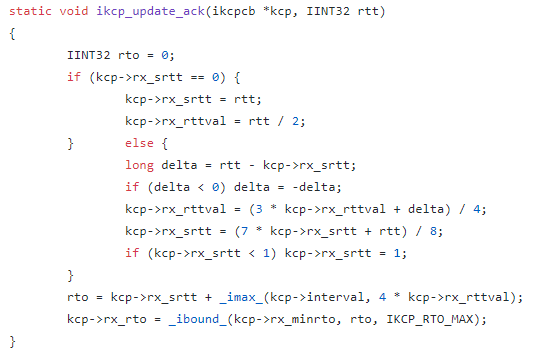


**步骤2：**根据接收到的CMD命令类型完成相应的操作；

**1、IKCP\_CMD\_ACK**

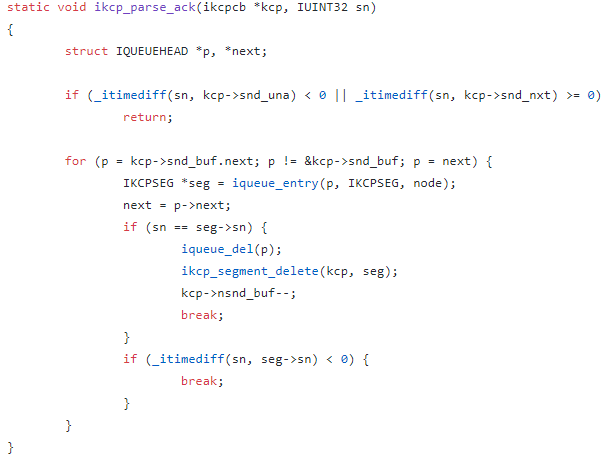
* 接收到数据确认包，调用ikcp\_update\_ack()更新RTO (rx\_rto)：

根据rtt更新rto



* 调用ikcp\_parse\_ack()，将被确认的segment从snd\_buf中移除：

如果找到被确认的segment，从snd\_buf中移除



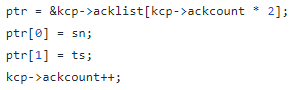
* 调用ikcp\_parse\_fastack()对未被确认segment的fastack进行计数（用于在下次flush时判断是否需要快速重传）：

sn > seg->sn 时



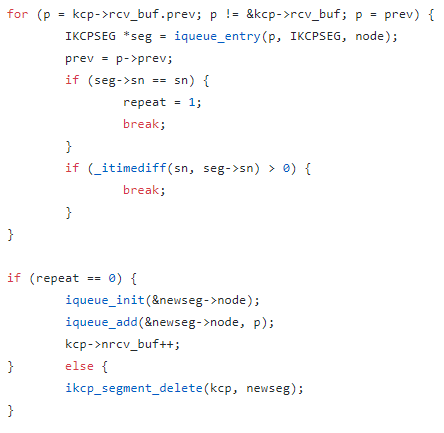
1. **IKCP\_CMD\_PUSH**

* 接收到推送数据的包，调用ikcp\_ack\_push()将数据包的sn和ts信息存入acklist：



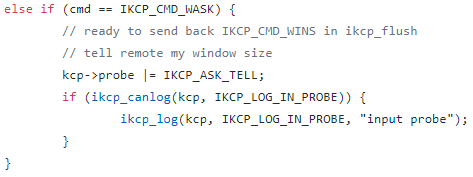
* 如果接收到的数据包在rcv\_queue中尚未被接收过(sn >= rcv\_nxt)，提取包中的信息，并调用ikcp\_parse\_data()检测是否在rcv\_buf中已有重复的segment，如果没有的话，将其加入rcv\_buf，否则丢弃。

检测是否收到重复的segment



1. **IKCP\_CMD\_WASK**

收到请求窗口尺寸的命令，将probe值标记为IKCP\_ASK\_TELL，等待下次flush时发送相应的（命令类型为IKCP\_CMD\_WINS的）数据包：

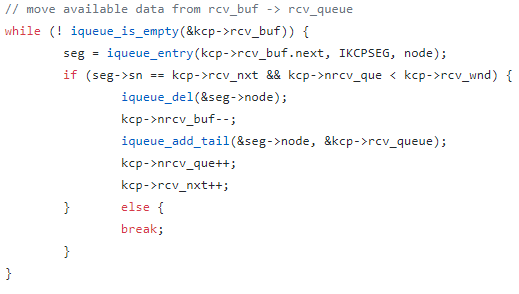


1. **IKCP\_CMD\_WINS**

收到发送窗口尺寸的命令，因为窗口尺寸信息(wnd)已在之前解包时存放入rmt\_wnd，因此此处不做任何处理。

**步骤3：**上层逻辑调用ikcp\_recv()获取接收到的数据。

1. 获取rcv\_queue中的segment，并根据其fragment值对不同分片进行合并，将已提取的segment从rcv\_queue中移除：
2. 如果rcv\_buf不为空，且rcv\_queue中segement的数量还未超过窗口尺寸（nrcv\_que < rcv\_wnd），则将rcv\_buf中的segment向rcv\_queue转移，并在此过程中保证rcv\_queue是有序的（sn == rcv\_nxt）：



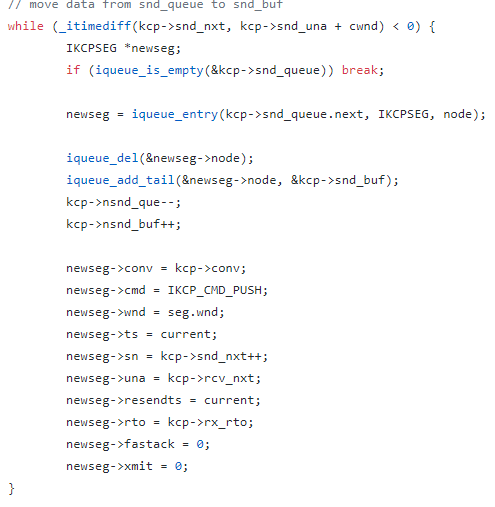
**4 流量控制**

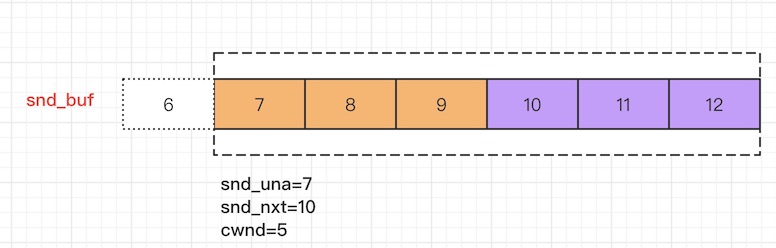
流量控制是点对点的通信量的控制，是一个端到端的问题。总结起来，就是发送方的速度要匹配接收方接收（处理）数据的速度。发送方要抑制自身的发送速率，以便使接收端来得及接收。

KCP的发送机制采用TCP的**滑动窗口**方式，可以非常容易的控制流量。KCP的头中包含wnd（窗口大小），即接收方目前可以接收的大小。接收方每次都会告诉发送方还能接收多少数据包，发送方据此可以计算出合适的cwnd，控制自己发送的数量，确保自己发送的数据不多于接收端可以接收的大小。

ikcp\_flush()在被调用时，会根据上述获得的cwnd将一定数量的segment从snd\_queue转入snd\_buf中，并随后发送snd\_buf中的segment，这些能够发送的segment即为sn在snd\_una与snd\_una+wnd之间的segment（snd\_nxt < snd\_una + cwnd）：

将segment从snd\_queue转入snd\_buf中，直至填满窗口



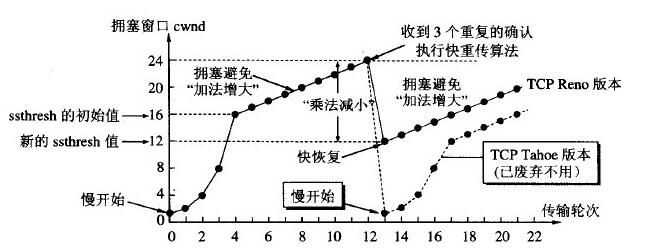


**/6**

* snd\_una 表示未被ack的Segment的最小编号
* snd\_nxt 表示下一个待发送的Segment的编号
* 黄色格子：已被占用
* 蓝色格子：可用

**5 拥塞控制**

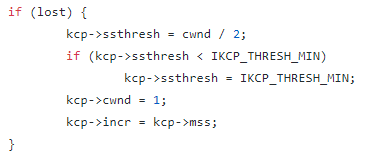
网络中链路的带宽，与整条网络中的交换节点（路由器、交换机、基站等）有关。如果所有使用该链路的流量超出了该链路所能提供的能力，就会发生拥塞，就像车多路窄，就会堵车，车越多堵得越厉害。对于TCP来说，当发送方没有按时接收到确认包，其就认为网络发生了拥堵行为并会降低自身发送数据的速度，主要实现的方式可以归结为慢开始和拥塞避免。



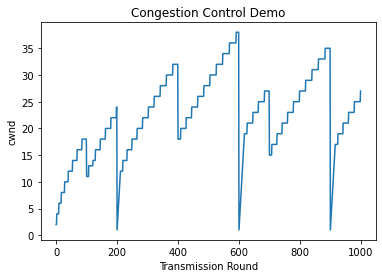
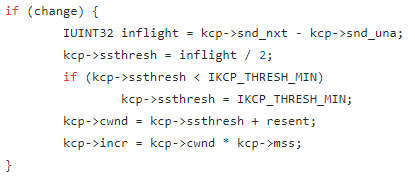
TCP拥塞控制策略

KCP采取的拥塞控制策略为：

* 当发生丢包时，控制窗口(cwnd)大小直接降为1，ssthresh减半：



* 当发生快速重传时，ssthresh大小调整为已经发送但尚未接收到ACK的数据包数目（snd\_nxt – snd\_una）的一半，控制窗口(cwnd)大小则为ssthresh大小 + resent：



KCP拥塞控制策略

虽然流量控制和拥塞控制都涉及到对cwnd大小的确定，但两者的区别是，流量控制是发送和接收两方的事情，而拥塞控制是整个网络的事情。

**6 实例演示 —— 利用netty实现的demo**

**7 附录**

**7.1 Notes**

rcv\_nxt，rcv\_wnd对应于 rcv\_queue；

snd\_nxt，snd\_wnd 对应于 snd\_buf

**7.2 各项参数含义**

conv 会话ID

mtu 最大传输单元

mss 最大分片大小（= mtu – overhead size）

state 连接状态（0xFFFFFFFF表示断开连接，当segment->xmit >= kcp->dead\_link）

snd\_una 第一个未确认的包

snd\_nxt 待发送包的序号

rcv\_nxt 待接收消息序号

ssthresh 拥塞窗口阈值

rx\_rttvar ack接收rtt浮动值

rx\_srtt ack接收rtt静态值

rx\_rto 由ack接收延迟计算出来的复原时间

rx\_minrto 最小复原时间

snd\_wnd 发送窗口大小

rcv\_wnd 接收窗口大小

rmt\_wnd, 远端接收窗口大小

cwnd, 拥塞窗口大小

probe 探查变量，IKCP\_ASK\_TELL表示告知远端窗口大小。IKCP\_ASK\_SEND表示请求远端告知窗口大小

interval 内部flush刷新间隔

ts\_flush 下次flush刷新时间戳

nodelay 是否启动无延迟模式

updated 是否调用过update函数的标识

ts\_probe, 下次探查窗口的时间戳

probe\_wait 探查窗口需要等待的时间

dead\_link 最大重传次数

incr 可发送的最大数据量

fastresend 触发快速重传的重复ack个数

nocwnd 取消拥塞控制

stream 是否采用流传输模式

snd\_queue 发送消息的队列

rcv\_queue 接收消息的队列

snd\_buf 发送消息的缓存

rcv\_buf 接收消息的缓存

acklist 待发送的ack列表

buffer 存储消息字节流的内存

output udp发送消息的回调函数