Kcp协议文档

目录

[一．协议 2](#_Toc517688160)

[1.1．简介 2](#_Toc517688161)

[1.2．Kcp为什么存在 2](#_Toc517688162)

[二．技术特性 2](#_Toc517688163)

[三．基本使用 3](#_Toc517688164)

[3.1.创建/销毁kcp对象 3](#_Toc517688165)

[3.2．设置回调函数 4](#_Toc517688166)

[3.3．循环调用update 4](#_Toc517688167)

[3.4．输入下层数据包 4](#_Toc517688168)

[3.5．数据的收发 4](#_Toc517688169)

[3.6．Kcp源码流程图 5](#_Toc517688170)

[四．实现细节 5](#_Toc517688171)

[4.1.IKCPSEG结构体 5](#_Toc517688172)

[4.2.IKCPCB结构体 6](#_Toc517688173)

[4.3.kcp设置中的常量值 8](#_Toc517688174)

[4.4.核心函数 9](#_Toc517688175)

[五．整体框架 10](#_Toc517688176)

[5.1．数据整体收发过程 10](#_Toc517688177)

[5.2．数据发送过程 12](#_Toc517688178)

[5.3.数据接收过程 13](#_Toc517688179)

[5.4．流量控制与拥塞控制 15](#_Toc517688180)

[六．协议配置 15](#_Toc517688181)

[6.1．工作模式： 15](#_Toc517688182)

[6.2．ikcp\_update调用时间设置 16](#_Toc517688183)

[6.3．设置发送/接受窗口大小 16](#_Toc517688184)

[6.4．设置最大传输单元 16](#_Toc517688185)

[6.5．最小RTO： 17](#_Toc517688186)

[6.6．设置回调函数（下层协议发送函数） 17](#_Toc517688187)

[6.7．获取发送端等待的数据包数量 17](#_Toc517688188)

[6.8．获取回话id 17](#_Toc517688189)

# 一．协议

## 1.1．简介

KCP是一个快速可靠协议，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。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。 连时钟都需要外部传递进来，内部不会有任何一次系统调用。

## 1.2．Kcp为什么存在

首先要看TCP与UDP的区别，TCP与UDP都是传输层的协议，比较两者的区别主要应该是说TCP比UDP多了什么？

**面向连接：**TCP接收方与发送方维持了一个状态（建立连接，断开连接），双方知道对方还在。

**可靠的：**发送出去的数据对方一定能够接收到，而且是按照发送的顺序收到的。

**流量控制与拥塞控制：**TCP靠谱通过滑动窗口确保，发送的数据接收方来得及收。TCP无私，发生数据包丢失的时候认为整个网络比较堵，自己放慢数据发送速度。

TCP协议的可靠与无私让使用TCP开发更为简单，同时它的这种设计也导致了慢的特点。UDP协议简单，所以它更快。但是，UDP毕竟是不可靠的，应用层收到的数据可能是缺失、乱序的。KCP协议就是在保留UDP快的基础上，提供可靠的传输，应用层使用更加简单。

其他差别，TCP以字节流的形式，UDP以数据包的形式。很多人以为，UDP是不可靠的，所以sendto(1000),接收端recvfrom(1000)可能会收到900。这个是错误的。所谓数据包，就是说UDP是有界的，sendto(300),sendto(500)；接收到，recvfrom(1000),recvfrom(1000)那么可能会收到300，500或者其中一个或者都没收到。UDP应用层发送的数据，在接收缓存足够的情况下，要么收到全的，要么收不到。

**总结：TCP可靠简单，但是复杂无私，所以速度慢。KCP尽可能保留UDP快的特点下，保证可靠。**

# 二．技术特性

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），以10%-20%带宽浪费的代价换取了比 TCP快30%-40%的传输速度。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，十分恐怖，而KCP启动快速模式后不x2，只是x1.5（实验证明1.5这个值相对比较好），提高了传输速度；在非急速模式下，每次+RTO。

**选择性重传 vs 全部重传：**

TCP丢包时会全部重传从丢的那个包开始以后的数据，KCP是选择性重传，只重传真正丢失的数据包。

**快速重传：**

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，延迟发送ACK（NODELAY都没用），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），光用UNA将导致全部重传，光用ACK则丢失成本太高，以往协议都是二选其一，而 KCP协议中，除去单独的 ACK包外，所有包都有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，仅用前两项来控制发送频率。以牺牲部分公平性及带宽利用率之代价，换取了开着BT都能流畅传输的效果。

# 三．基本使用

## 3.1.创建/销毁kcp对象

ikcpcb\* ikcp\_create(IUINT32 conv, void \*user);

创建一个kcp对象，同一个会话中的两个conv必须是相同的，user作为用户标识（发端与收端），kcp->output回调函数需要user参数。

void ikcp\_release(ikcpcb \*kcp);

销毁一个kcp对象，清除空间。

## 3.2．设置回调函数

int udp\_output(const char \*buf, int len, ikcpcb \*kcp, void \*user);//回调函数定义

// 设置回调函数

void ikcp\_setoutput(ikcpcb \*kcp, int (\*output)(const char \*buf, int len,

    ikcpcb \*kcp, void \*user))

{

kcp->output = udp\_output;

}

其中udp\_output为下层传输函数指针，buf为数据缓冲(sendto发送该数据)，len为长度，uesr用于区别多个kcp对象

## 3.3．循环调用update

void ikcp\_update(ikcpcb \*kcp, IUINT32 current);

根据当前时间判断是否达到下一次刷新时间戳（current-ts\_flush>0则调用ikcp\_flush），或者用ikcp\_check来确定下次调用update的时间。

Update会定时执行，不断更新snd\_buf中的包（从sndque中取出存入或者收到ack删除相应的包）

## 3.4．输入下层数据包

int ikcp\_input(ikcpcb \*kcp, const char \*data, long size);

对收到的下层数据包利用input将包分片存储在rcv\_buf中。这里的data是通过下层接受函数收到的数据（udp中的recvfrom获取data数据）

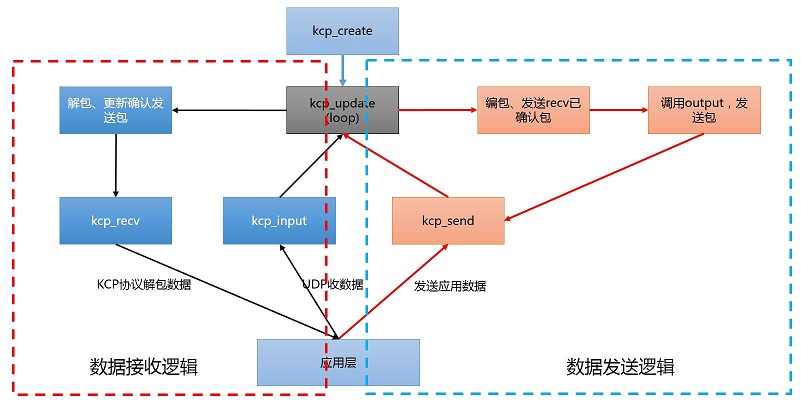
## 3.5．数据的收发

int ikcp\_send(ikcpcb \*kcp, const char \*buffer, int len);

int ikcp\_recv(ikcpcb \*kcp, char \*buffer, int len);

处理下层协议的输入输出后，就可以调用ikcp\_send发送数据，ikcp\_recv接受数据

## 3.6．Kcp源码流程图



UDP收到的包，不断通过kcp\_input喂给KCP，KCP会对这部分数据（KCP协议数据）进行解包，重新封装成应用层用户数据，应用层通过kcp\_recv获取。应用层通过kcp\_send发送数据，KCP会把用户数据拆分kcp数据包，通过kcp\_output，以UDP（send）的方式发送。

# 四．实现细节

## 4.1.IKCPSEG结构体

数据包：

0 4 5 6 8 (BYTE)

+---------------+---+---+------ -+

| conv |cmd | frg | wnd |

+---------------+---+---+--------+ 8

| ts | sn |

+---------------+----------------+ 16

| una | len |

+---------------+-----------------+ 24

| |

| DATA (optional) |

+---------------------------------+

struct IKCPSEG

{

    struct IQUEUEHEAD node;//节点

    IUINT32 conv; //会话id

    IUINT32 cmd; //分片的作用，IKCP\_CMD\_PUSH:数据分片 IKCP\_CMD\_ACK:ack分片 IKCP\_CMD\_WASK接受窗口大小的询问命令 IKCP\_CMD\_WINS:接受窗口大小的告知命令

    IUINT32 frg; //分片id

    IUINT32 wnd; //剩余窗口大小

    IUINT32 ts; //时间戳

    IUINT32 sn; //序号

    IUINT32 una; //下一个待确认序号(滑动窗口左端)

    IUINT32 len; //分片长度

    char data[1]; //数据

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

//不随数据包发送出去，只作为发端的标记，添加到snd\_buf时设置初始值

    IUINT32 resendts; //下次超时重传的时间戳

    IUINT32 rto; //分片的重传超时时间

IUINT32 fastack; //收到ack时，分片被跳过的次数

    IUINT32 xmit; //发送分片的次数

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

};

## 4.2.IKCPCB结构体

|  |  |
| --- | --- |
| conv, mtu, mss, state | 会话id||mtu||mss||state表示连接状态 |
| snd\_una,  snd\_nxt,  rcv\_nxt | 下一个未确认的包||  下一个待发送包的序号||  下一个期望收到的序号 |
| ts\_recent,  ts\_lastack, ssthresh | ts\_lastack和ts\_recent暂时未使用设置为0  ssthresh拥塞控制门限 |
| rx\_rttval,  rx\_srtt,  rx\_rto,  rx\_minrto | ack接收rtt平均偏差，衡量rtt的抖动  ack接收rtt加权值，平滑值  由ack接收延迟计算的重传超时时间  最小rto |
| snd\_wnd,  rcv\_wnd,  rmt\_wnd,  cwnd,  probe | 发送窗口大小  接受窗口大小  远端可用窗口大小  拥塞窗口大小  probe探查变量 |
| current,  interval,  ts\_flush,  xmit; | 当前时间  刷新时间间隔，  下一次刷新时间戳，  最大重发次数 |
| nrcv\_buf, nsnd\_buf;  nrcv\_que, nsnd\_que | buf大小  que大小 |
| nodelay,  updated | 是否启动无延迟模式，1启动  是否调用update函数 |
| ts\_probe,  probe\_wait | 下次探查窗口的时间戳，  探查窗口需要等待的时间 |
| dead\_link,  incr | 最大重传次数，  可发送的最大数量 |
| snd\_queue;  rcv\_queue;  snd\_buf;  rcv\_buf; | 队列缓冲 |
| acklist;  ackcount;  ackblock; | //待发送的ack列表，1个ack块={sn,ts}  //待发送的ack数量  //块数，表示最大容量 |
| user | 用户标识，标记发端与收端 |
| buffer | 消息字节流缓冲区，缓存封装好的||待发送的字节流数据，（后续作为udp中的sendto函数的buf参数） |
| fastresend | 快重传门限，分片的fastack> fastresend时，触发快重传 |
| nocwnd,  stream | nocwnd表示是否关闭流控（为0，表示考虑拥塞控制），stream是否采用流传输模式（为0表示采用消息模式） |
| output | 下层传输函数 |
| Logmask  writelog | 日志掩码  写log函数 |

struct IKCPCB

{

    IUINT32 conv, mtu, mss, state; //会话id||mtu||mss||state表示连接状态

    IUINT32 snd\_una, snd\_nxt, rcv\_nxt; //第一个未确认的包||下一个待分配的包的序号||待接收消息的序号

    IUINT32 ts\_recent, ts\_lastack, ssthresh; //ssthresh拥塞控制门限，ts\_lastack和ts\_recent暂时未使用设置为0

    IINT32 rx\_rttval, rx\_srtt, rx\_rto, rx\_minrto; //ack接收rtt浮动值||ack接收rtt静态值||由ack接收延迟计算出来的重传超时时间||最小rto

    IUINT32 snd\_wnd, rcv\_wnd, rmt\_wnd, cwnd, probe; //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,最小重传超时时间

    IUINT32 current, interval, ts\_flush, xmit; //当前时间，刷新时间间隔，下一次刷新时间戳，最大重发次数

    IUINT32 nrcv\_buf, nsnd\_buf; // buf大小

    IUINT32 nrcv\_que, nsnd\_que; //que大小

    IUINT32 nodelay, updated; //是否启动无延迟模式，是否调用update函数

    IUINT32 ts\_probe, probe\_wait; //下次探查窗口的时间戳，探查窗口需要等待的时间

    IUINT32 dead\_link, incr; //最大重传次数，可发送的最大数量

    /\* 队列缓冲\*/

    struct IQUEUEHEAD snd\_queue;

    struct IQUEUEHEAD rcv\_queue;

    struct IQUEUEHEAD snd\_buf;

    struct IQUEUEHEAD rcv\_buf;

    IUINT32 \*acklist; //待发送的ack列表

    IUINT32 ackcount;//待发送的ack数

    IUINT32 ackblock;//一个block（ack）包含{sn,ts}这两个数据

    void \*user; //标识发端与收端

    char \*buffer; //消息字节流缓冲区域

    int fastresend; //快重传门限

    int nocwnd, stream; //nocwnd表示是否关闭流控（为0，表示考虑拥塞），stream是否采用流传输模式

    int logmask;

    int (\*output)(const char \*buf, int len, struct IKCPCB \*kcp, void \*user); //下层传输函数

    void (\*writelog)(const char \*log, struct IKCPCB \*kcp, void \*user);

};

## 4.3.kcp设置中的常量值

const IUINT32 IKCP\_RTO\_NDL = 30;        // no delay min rto

const IUINT32 IKCP\_RTO\_MIN = 100;       // normal min rto

const IUINT32 IKCP\_RTO\_DEF = 200; //defult rto

const IUINT32 IKCP\_RTO\_MAX = 60000; ///max rto

const IUINT32 IKCP\_CMD\_PUSH = 81;       // cmd: push data数据分片

const IUINT32 IKCP\_CMD\_ACK = 82;       // cmd: ack分片

const IUINT32 IKCP\_CMD\_WASK = 83;       // cmd: window probe (ask)

const IUINT32 IKCP\_CMD\_WINS = 84;       // cmd: window size (tell)

const IUINT32 IKCP\_ASK\_SEND = 1;        // need to sendIKCP\_CMD\_WASK

const IUINT32 IKCP\_ASK\_TELL = 2;        // need to send IKCP\_CMD\_WINS

const IUINT32 IKCP\_WND\_SND = 32; //send wnd

const IUINT32 IKCP\_WND\_RCV = 128; // must >= max fragment size,接收窗口大小

const IUINT32 IKCP\_MTU\_DEF = 1400; //defult mtu size

const IUINT32 IKCP\_ACK\_FAST = 3; //快重传门限

const IUINT32 IKCP\_INTERVAL = 100; //update时间间隔

const IUINT32 IKCP\_OVERHEAD = 24; //kcp首部大小（字节）

const IUINT32 IKCP\_DEADLINK = 20; //最大重传次数默认值

const IUINT32 IKCP\_THRESH\_INIT = 2; //拥塞门限默认值

const IUINT32 IKCP\_THRESH\_MIN = 2; //拥塞门限最小值

const IUINT32 IKCP\_PROBE\_INIT = 7000;       // 7 secs to probe window size

const IUINT32 IKCP\_PROBE\_LIMIT = 120000;    // up to 120 secs to probe window

## 4.4.核心函数

1. Ikcp\_send(kcp,buffer,len)：

作用：（根据kcp->mss）将buffer分片成seg加入到kcp->snd\_queue，队列大小没有明确限制

（只更新了snd\_que）

返回值：正常结束返回0，len<0返回-1，分片空间分配出错返回-2；

1. Ikcp\_update(kcp, current)：

作用：根据当前时间和刷新间隔设置数据下次刷新时间，调用ikcp\_flush，将snd\_queue里面的数据加入到snd\_buf队列中，snd\_buf里面的节点个数要受到min(snd\_wnd,rmt\_wnd)的限制（拥塞控制的情况下受到min(snd\_wnd,rmt\_wnd,cwnd)的限制），flush再调用ikcp\_output将队列中的数据通过函数回调（kcp->output），发送出去；数据包发送时，要标记该包是否发送快重传或者丢包，对于快重传要更新拥塞窗口||快重传门限||最大可发送数据量，对于丢包，要设置慢启动（cwnd=1）和最大可发送数量=mss；

如有待确认的数据包的（ackcount!=0）,会发送确认包

（更新snd\_que，snd\_buf）

返回值：无返回值

1. Ikcp\_input(kcp,data,size)：解析收到的udp数据包；

作用：如果是数据分片，对收到的数据包设置ack，data就是udp收到的数据，数据的前24字节为kcpseg首部，逐一提取结构体每个字段，解析成seg（在[rcv\_nxt，rcv\_nxt+rcv\_wnd]范围内才加入）加入到rcv\_buf（buf中的包有序，但不一定连续）中，再将rcv\_buf中的数据同步到rcv\_queue中（sn==rcv\_nxt才加入，保证que中的数据包有序且连续），受rcv\_wnd门限限制，

如果是ack，则更新snd\_buf中的数据包（还有rto）

如果是窗口请求，设置probe=ask\_tell

如果是窗口大小应答，则将wnd写入log

最后进行拥塞控制（设置cwnd||ssthread||incr，进行慢启动和拥塞避免）

（更新rcv\_buf,rcv\_que,snd\_buf，acklist）

返回值：正常结束返回0，错误返回负数

1. Ikcp\_recv(kcp,buffer,len)：

作用：将kcp->rcv\_queue里的分片解析为数据存入buffer返还给用户，并且同步kcp->rcv\_buf中的分片到kcp->rcv\_queue中，个数受rcv\_wnd门限限制（kcp->nrcv\_que<kcp->rcv\_wnd）可以理解为接受窗口

（更新rcv\_buf,rcv\_que）

返回值：正常结束返回读取的数据长度，错误返回负数

# 五．整体框架

## 5.1．数据整体收发过程

第一次update:

1. P1端将待发送的数据buf分片成kcp的数据包格式，插入待发送队列（snd\_que）中；将snd\_que中的数据包取出按序存放到snd\_buf中，数据包通过下层协议（udp）发送
2. P2端将udp收到的数据包转换成seg数据包格式，对收到seg包，进行确认（添加ack块进acklist）；再将kcp数据包按序加入到rcv\_que中，然后提取rcv\_que中的包还原成数据buf；

第二次update：

1. P2端将acklist中的ack块封装成分片发送出去，并继续执行发送自身缓冲中的数据（如果有）
2. P1端对收到的数据包进行判定（ack分片，数据包分片），若是ack分片则删除snd\_buf中的对应的缓存包；若是数据分片，则加入到rcv\_buf中，并更新rcv\_buf->rcv\_que；

…………循环

**数据流的轨迹为**：

Kcp2

Kcp1

S\_que

R\_que

R\_buf

S\_buf

data

S\_buf

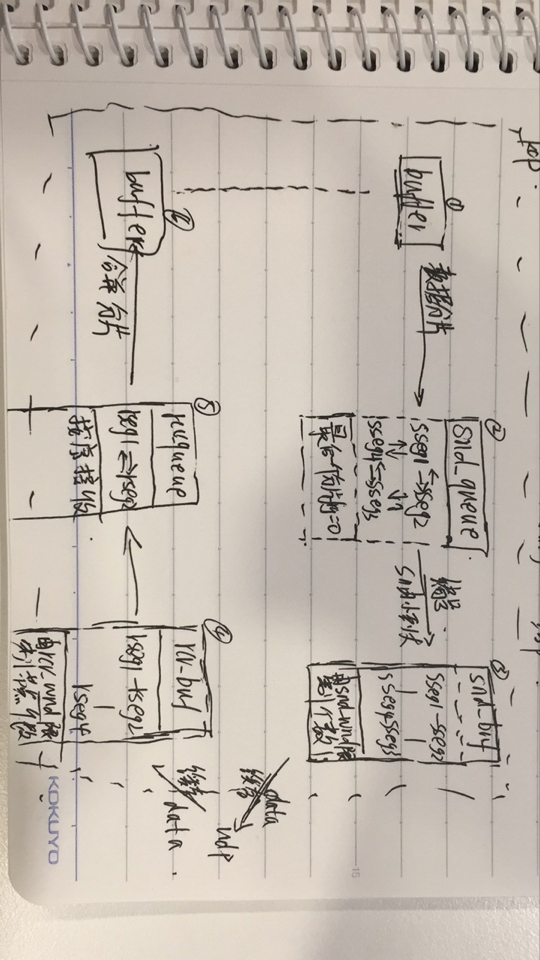
S\_que

R\_que

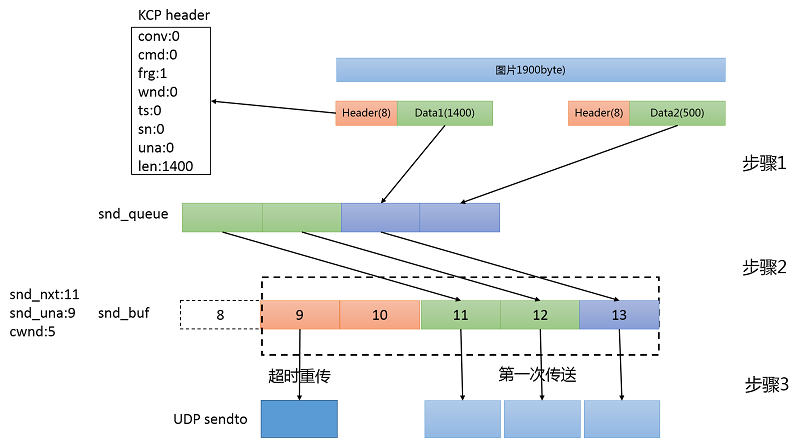
R\_buf

data

**单个kcp数据流轨迹：**



## 5.2．数据发送过程

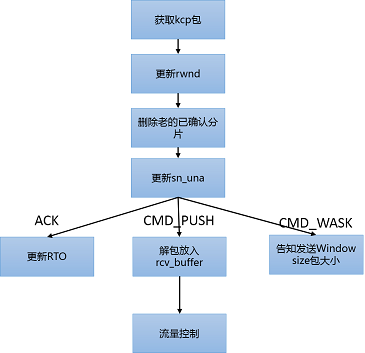


**步骤1，2（snd\_queue🡪snd\_buf）：**如上图中，snd\_queue待发送队列中有4个KCP包等待发送，这个时候snd\_nxt下一个发送的kcp包序列号为11，snd\_una下一个确认的KCP包为9（8已经确认，9，10已经发送但是还没得到接收方的确认）。因为cwnd=5，发送队列中还有2个发送了但是还未得到确认，所以可以从待发送队列中取前面的3个KCP包放入到发送队列中，序列号分别设置为11,12,13。

**步骤3（发送snd\_buf中的数据）：**发送队列中包含两种类型的数据，已发送但是尚未被接收方确认的数据，没被发送过的数据。没发送过的数据比较好处理，直接发送即可。重点在于已经发送了但是还没被接收方确认的数据，**该部分的策略直接决定着协议快速、高效与否。**KCP主要使用两种策略来决定是否需要重传KCP数据包，**超时重传、快速重传、选择重传。**

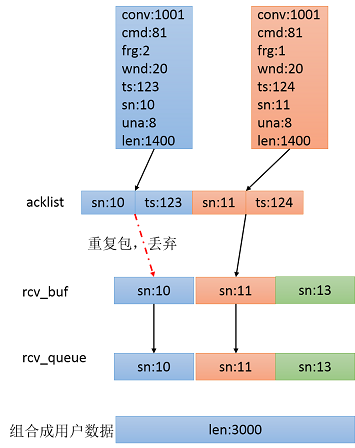
## 5.3.数据接收过程

接收处理流程图：



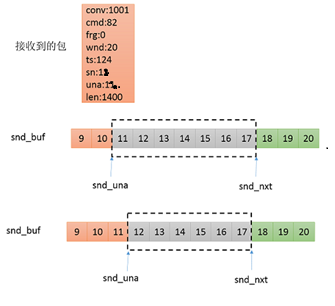
**1、IKCP\_CMD\_PUSH数据发送命令**  
a、KCP会把收到的数据包的sn及ts放置在acklist中，两个相邻的节点为一组，分别存储sn和ts。update时会读取acklist，**并以IKCP\_CMD\_ACK的命令返回确认包**。如下图中，收到了两个kpc包，acklist中会分别存放10,123,11,124。  
b、kcp数据包放置rcv\_buf队列。丢弃接收窗口之外的和重复的包。然后将rcv\_buf中的包，移至rcv\_queue。（假定）原来的rcv\_buf中已经有sn=10和sn=13的包了，sn=10的kcp包已经在rcv\_buf中了，因此新收到的包会直接丢弃掉，sn=11的包放置至rcv\_buf中。  
c、把rcv\_buf中前面连续的数据sn=11，12，13全部移动至rcv\_queue，rcv\_nxt也变成14。

**rcv\_queue的数据是连续的，rcv\_buf可能是间隔的**d、kcp\_recv函数，用户获取接收到数据（去除kcp头的用户数据）。该函数根据frg，把kcp包数据进行组合返回给用户。



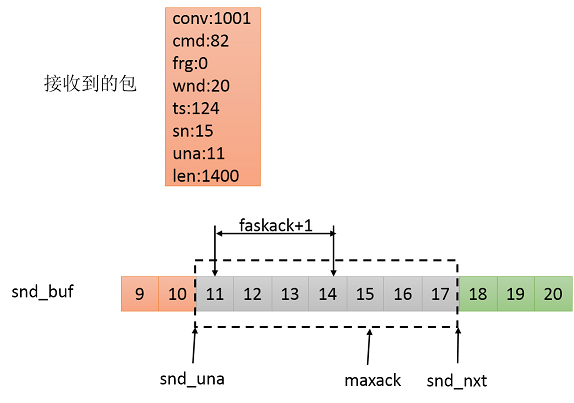
**2、IKCP\_CMD\_ACK数据确认包**  
两个作用：1、RTO更新，2、确认发送包接收方已接收到。

**正常情况：**收到的sn为11,una为12。表示sn为11的已经确认，下一个等待接收的为12。发送队列中，待确认的一个包为11，这个时候snd\_una向后移动一位，序列号为11的包从发送队列中删除。



[ 数据确认包处理流程 ]

**异常情况**：如下图所示，sn!=11的情况均为异常情况。sn<11表示，收到重复确认的包，如本来以为丢失的包重新又收到了，所以产生重复确认的包；sn>17，收到没发送过的序列号，概率极低，可能是conv没变重启程序导致的；其他条件，统计fastack是否大于fastresend，是则启动快速重传。



[ KCP快速确认 ]

## 5.4．流量控制与拥塞控制

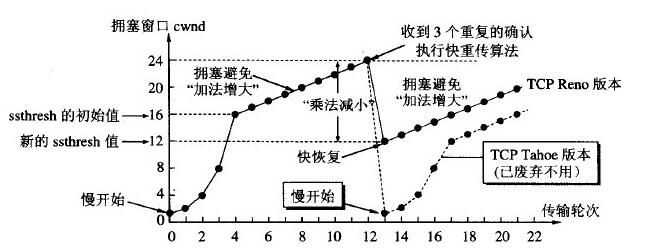
**流量控制：**

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，即接收方目前可以接收的大小。能够发送的数据即为snd\_una与snd\_una+wnd之间的数据。接收方每次都会告诉发送方我还能接收多少，发送方就控制下，确保自己发送的数据不多于接收端可以接收的大小。

KCP默认为32，即可以接收最大为32\*MTU=43.75kB。KCP采用update的方式，更新间隔为10ms，那么KCP限定了你最大传输速率为4375kB/s，在高网速传输大内容的情况下需要调用ikcp\_wndsize调整接收与发送窗口。

KCP的主要特色在于实时性高，对于实时性高的应用，如果发生数据堆积会造成延迟的持续增大。建议从应用侧更好的控制发送流量与网络速度持平，避免缓存堆积延迟。（详见<https://github.com/skywind3000/kcp/wiki/Flow-Control-for-Users>）

**拥塞控制：**



**KCP发生丢包的情况下的拥塞控制策略与TCP Tahoe版本的策略一致**。TCP Reno版本已经使用快恢复策略。因此，丢包的情况下，其实KCP拥塞控制策略比TCP更为苛刻。

**KCP在发生快速重传时，采用的是TCP快恢复的策略**。**控制窗口调整为已经发送没有接收到ack的数据包数目的一半+resent。**

# 六．协议配置

## 6.1．工作模式：

int ikcp\_nodelay(ikcpcb \*kcp, int nodelay, int interval, int resend, int nc);

* nodelay ：是否启用无延迟模式，0不启用；1启用。
* interval ：刷新时间间隔，单位毫秒，比如 10ms或者 20ms
* resend ：快速重传模式，默认0关闭，可以设置2（2次ACK跨越将会直接重传）
* nc ：是否关闭流控，默认是0代表不关闭（考虑拥塞），1代表关闭（不考虑拥塞）。
* 普通模式： ikcp\_nodelay(kcp, 0, 40, 0, 0);
* 极速模式： ikcp\_nodelay(kcp, 1, 10, 2, 1);

## 6.2．ikcp\_update调用时间设置

IUINT32 ikcp\_check(const ikcpcb \*kcp, IUINT32 current);

决定何时调用ikcp\_update。如果没有ikcp\_input/\_send的调用，则返回调用ikcp\_update的时间戳。

**意义**：如果需要同时管理大规模的 KCP连接（比如大于3000个），比如你正在实现一套类 epoll的机制，那么为了避免每秒钟对每个连接调用大量的调用 ikcp\_update，我们可以使用 ikcp\_check 来大大减少 ikcp\_update调用的次数。 ikcp\_check返回值会告诉你需要在什么时间点再次调用 ikcp\_update（如果中途没有 ikcp\_send, ikcp\_input的话，否则中途调用了 ikcp\_send, ikcp\_input的话，需要在下一次interval时调用 update）

使用该方法，原来在处理2000个 kcp连接且每 个连接每10ms调用一次update，改为 check机制后，cpu从 60%降低到 15%。

**用法**：每次调用了 ikcp\_update后，使用 ikcp\_check决定下次什么时间点再次调用 ikcp\_update，而如果中途发生了 ikcp\_send, ikcp\_input 的话，在下一轮 interval 立马调用 ikcp\_update和 ikcp\_check。

## 6.3．设置发送/接受窗口大小

int ikcp\_wndsize(ikcpcb \*kcp, int sndwnd, int rcvwnd)

设置窗口大小，单位是数据包，发送窗口默认是IKCP\_WND\_SND(32)，接受窗口默认为IKCP\_WND\_RCV(128)，

## 6.4．设置最大传输单元

int ikcp\_setmtu(ikcpcb \*kcp, int mtu)

设置mtu的大小，默认为IKCP\_MTU\_DEF（1400），单位是包；

对于给定的mtu，都有相应的mss（=mtu-IKCP\_OVERHEAD）kcp->buffer（=ikcp+malloc((mtu+ IKCP\_OVERHEAD)\*3)）相对应，IKCP\_OVERHEAD设置为24（表示首部的24字节）

## 6.5．最小RTO：

不管是 TCP还是 KCP计算 RTO时都有最小 RTO的限制，即便计算出来RTO为40ms，由于默认的 RTO是100ms，协议只有在100ms后才能检测到丢包，快速模式下为30ms，可以手动更改该值：

如kcp->rx\_minrto = 10;

## 6.6．设置回调函数（下层协议发送函数）

void ikcp\_setoutput(ikcpcb \*kcp, int (\*output)(const char \*buf, int len,

    ikcpcb \*kcp, void \*user))

将kcp->output设置为 udp\_output（调用udp的sendto）

## 6.7．获取发送端等待的数据包数量

int ikcp\_waitsnd(const ikcpcb \*kcp);

返回nsnd\_que+nsnd\_buf；

## 6.8．获取回话id

IUINT32 ikcp\_getconv(const void \*ptr)

通过数据指针ptr，解码获取数据中的conv，返回conv