KCP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。

整个协议只有 ikcp.h, ikcp.c两个源文件，可以方便的集成到用户自己的协议栈中。也许你实现了一个P2P，或者某个基于 UDP的协议，而缺乏一套完善的ARQ可靠协议实现，那么简单的拷贝这两个文件到现有项目中，稍微编写两行代码，即可使用。
kcp源码的网站kcp源码

源码分析

我们先看一下kcp的协议和一些KCP通信中的概念

1KCP协议头

```
0               4   5   6       8 (BYTE)
+---------------+---+---+-------+
|     conv      |cmd|frg|  wnd  |
+---------------+---+---+-------+   8
|     ts        |     sn        |
+---------------+---------------+  16
|     una       |     len       |
+---------------+---------------+  24
|                               |
|        DATA (optional)        |
|                               |
+-------------------------------+
```

conv: 用于区分kcp会话ID，通信双方需要相同id
cmd: 操作指令有PUSH(数据包)，ACK(确认包)，WASK(询问窗口大小的包)，WINS（告知窗口大小的包）
frg: 分片段号
wnd: 接收方窗口大小
ts: 包发送的时间戳
sn: 包的序列号
una: 未确认的包的序列号
len: DATA的长度
DATA: 数据

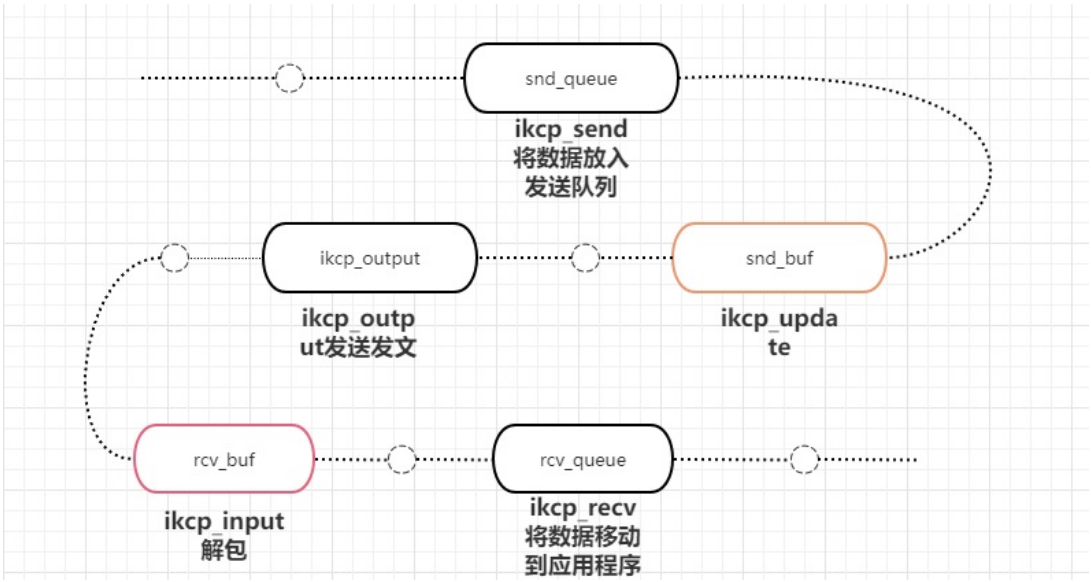
KCP通信中的概念

看一下kcp的控制块

```
//---------------------------------------------------------------------
// IKCPCB
//---------------------------------------------------------------------
struct IKCPCB
{
        IUINT32 conv, mtu, mss, state;        //mtu 最大传输单元   mss最大分片大小
        IUINT32 snd_una, snd_nxt, rcv_nxt;   //snd_una第一个未确认的包   snd_nxt下一个待分配的包  rcv_nxt待接收的下一个包
        IUINT32 ts_recent, ts_lastack, ssthresh;    //ssthresh拥塞窗口的阈值
        IINT32 rx_rttval, rx_srtt, rx_rto, rx_minrto;   //rx_rttval是rtt浮动值  rx_srtt是rtt静态值   rx_rto是ack接收计算出的重传超时时间  rx_minrto最小重传超时时间
        IUINT32 snd_wnd, rcv_wnd, rmt_wnd, cwnd, probe;  //rmt_wnd 是远端接收窗口大小 cwnd是拥塞窗口大小 probe是探查变量
        IUINT32 current, interval, ts_flush, xmit;
        IUINT32 nrcv_buf, nsnd_buf;
        IUINT32 nrcv_que, nsnd_que;
        IUINT32 nodelay, updated;
        IUINT32 ts_probe, probe_wait;      //ts_probe是下次探查窗口的时间戳  probe_wait是探查窗口的等待时间
        IUINT32 dead_link, incr;
        struct IQUEUEHEAD snd_queue;   // 待发送的数据队列，调用ikcp_send()后数据会放入snd_buf中
        struct IQUEUEHEAD rcv_queue;   //  接收到消息队列 数据是连续的
        struct IQUEUEHEAD snd_buf;      //已经发送但是未收到ack的数据队列
        struct IQUEUEHEAD rcv_buf;      //接收到的数据缓存 可能是不连续的包
        IUINT32 *acklist;
        IUINT32 ackcount;
        IUINT32 ackblock;
        void *user;
        char *buffer;
        int fastresend;
        int fastlimit;
        int nocwnd, stream;   //nocwnd是否取消拥塞控制,  stream&626159;否采用流式传输
        int logmask;
        int (*output)(const char *buf, int len, struct IKCPCB *kcp, void *user);
        void (*writelog)(const char *log, struct IKCPCB *kcp, void *user);
};
```

2KCP的数据流向

简单的说就是 snd_queue------>>>snd_buf---------->>>rcv_buf---------->>>rcv_queue



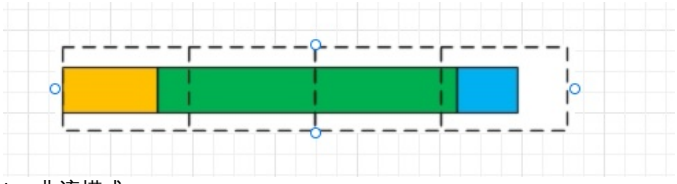
主要的四个函数为

```
int ikcp_send(ikcpcb *kcp, const char *buffer, int len)
int ikcp_input(ikcpcb *kcp, const char *data, long size)
void ikcp_flush(ikcpcb *kcp)
int ikcp_recv(ikcpcb *kcp, char *buffer, int len)
```

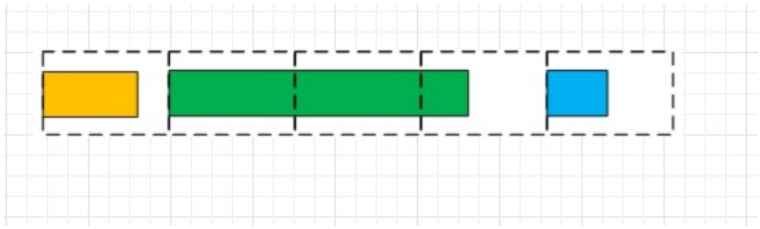
ikcp_send

kcp发送数据报文有两种模式，流模式和非流模式，通过控制块中的stream字段标记。简而言之流模式是在报文发送中如果发送队列的前一个报文段数据没有填满，会尽可能的将后一个报文的数据填入其中。非流模式就是一次输入数据对应一个报文，如果数据大小大于mss就将其拆分成多个。流模式的粘包和分包问题需要应用层来解决

KCP流模式



kcp非流模式



```
//---------------------------------------------------------------------
// user/upper level send, returns below zero for error
//---------------------------------------------------------------------
int ikcp_send(ikcpcb *kcp, const char *buffer, int len)
{
	IKCPSEG *seg;
	int count, i;

	assert(kcp->mss > 0);
	if (len < 0) return -1;

	// append to previous segment in streaming mode (if possible)
	//如果是流式模式，前一个报文的数据段没满需要将其填满
	if (kcp->stream != 0) {
		if (!iqueue_is_empty(&kcp->snd_queue)) {
			IKCPSEG *old = iqueue_entry(kcp->snd_queue.prev, IKCPSEG, node);
			if (old->len < kcp->mss) {	//mss最大分片大小
				int capacity = kcp->mss - old->len;
				int extend = (len < capacity)? len : capacity;
				seg = ikcp_segment_new(kcp, old->len + extend); //新建一个分片
				assert(seg);
				if (seg == NULL) {
					return -2;
				}
				iqueue_add_tail(&seg->node, &kcp->snd_queue);	//将这个分片加入snd_queue
				memcpy(seg->data, old->data, old->len);
				if (buffer) {
					memcpy(seg->data + old->len, buffer, extend);
					buffer += extend;
```

```
                                }
                                seg->len = old->len + extend;
                                seg->frg = 0;
                                len -= extend;
                                iqueue_del_init(&old->node);
                                ikcp_segment_delete(kcp, old);
                        }
                }
                if (len <= 0) {
                        return 0;
                }
        }

        if (len <= (int)kcp->mss) count = 1;
        else count = (len + kcp->mss - 1) / kcp->mss;

        if (count >= (int)IKCP_WND_RCV) return -2;

        if (count == 0) count = 1;    // count是报文数量

        // fragment
        for (i = 0; i < count; i++) {
                int size = len > (int)kcp->mss ? (int)kcp->mss : len;
                seg = ikcp_segment_new(kcp, size);
                assert(seg);
                if (seg == NULL) {
                        return -2;
                }
                if (buffer && len > 0) {
                        memcpy(seg->data, buffer, size);
                }
                seg->len = size;
                //非流模式分配段号从大到小
                seg->frg = (kcp->stream == 0)? (count - i - 1) : 0;
                iqueue_init(&seg->node);
                iqueue_add_tail(&seg->node, &kcp->snd_queue);
                kcp->nsnd_que++;
                if (buffer) {
                        buffer += size;
                }
                len -= size;
        }

        return 0;
}
```

ikcp_flush

ikcp_flush由上层应用调用ikcp_update来驱动KCP发送数据一般每隔一段时间（10-100ms）调用一次
ikcp_flush会发送ack报文，检查是否对远端窗口进行探测&565292;发送报文，调整拥塞窗口

```
//---------------------------------------------------------------------
// ikcp_flush
//---------------------------------------------------------------------
void ikcp_flush(ikcpcb *kcp)
{
        IUINT32 current = kcp->current;
        char *buffer = kcp->buffer;
        char *ptr = buffer;
        int count, size, i;
        IUINT32 resent, cwnd;
        IUINT32 rtomin;
        struct IQUEUEHEAD *p;
        int change = 0;   //快速重传的发生
        int lost = 0;           //报文丢失
        IKCPSEG seg;

        // 'ikcp_update' haven't been called.
        //上层调用ikcp_update来让kcp发送数据
        if (kcp->updated == 0) return;

        seg.conv = kcp->conv;
        seg.cmd = IKCP_CMD_ACK;
        seg.frg = 0;
        seg.wnd = ikcp_wnd_unused(kcp);
        seg.una = kcp->rcv_nxt;
        seg.len = 0;
        seg.sn = 0;
        seg.ts = 0;

        // flush acknowledges
        //将acklist中的ack报文发送出去，在ikcp_input中会通过ikcp_ack_push将收到的报文的sn和ts字段写入acklist中
        count = kcp->ackcount;
        for (i = 0; i < count; i++) {
                size = (int)(ptr - buffer);
                if (size + (int)IKCP_OVERHEAD > (int)kcp->mtu) {
                        ikcp_output(kcp, buffer, size);
                        ptr = buffer;
```

```
        }
        ikcp_ack_get(kcp, i, &seg.sn, &seg.ts);
        ptr = ikcp_encode_seg(ptr, &seg);
}

kcp->ackcount = 0;

// probe window size (if remote window size equals zero)
// 检查当前是否需要对远端窗口进行探测。
if (kcp->rmt_wnd == 0) {
        if (kcp->probe_wait == 0) {
                kcp->probe_wait = IKCP_PROBE_INIT;
                kcp->ts_probe = kcp->current + kcp->probe_wait; // 初始化探测间隔和下一次探测时间
        }
        else {
                if (_itimediff(kcp->current, kcp->ts_probe) >= 0) { //当前时间 > 下一次探查窗口的时间
                        if (kcp->probe_wait < IKCP_PROBE_INIT)
                                kcp->probe_wait = IKCP_PROBE_INIT;
                        kcp->probe_wait += kcp->probe_wait / 2; //等待时间变为之前的1.5倍
                        if (kcp->probe_wait > IKCP_PROBE_LIMIT)
                                kcp->probe_wait = IKCP_PROBE_LIMIT;  //若超过上限，设置为上限值
                        kcp->ts_probe = kcp->current + kcp->probe_wait;  //计算下次探查窗口的时间戳
                        kcp->probe |= IKCP_ASK_SEND;  //设置探查变量。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
                }
        }
}       else {
        kcp->ts_probe = 0;
        kcp->probe_wait = 0;
}

// flush window probing commands
// 将窗口探测报文发送出去
if (kcp->probe & IKCP_ASK_SEND) {
        seg.cmd = IKCP_CMD_WASK;
        size = (int)(ptr - buffer);
        if (size + (int)IKCP_OVERHEAD > (int)kcp->mtu) {
                ikcp_output(kcp, buffer, size);
                ptr = buffer;
        }
        ptr = ikcp_encode_seg(ptr, &seg);
}

// flush window probing commands
// 将窗口回复报文发送出去
if (kcp->probe & IKCP_ASK_TELL) {
        seg.cmd = IKCP_CMD_WINS;
        size = (int)(ptr - buffer);
        if (size + (int)IKCP_OVERHEAD > (int)kcp->mtu) {
                ikcp_output(kcp, buffer, size);
                ptr = buffer;
        }
        ptr = ikcp_encode_seg(ptr, &seg);
}

kcp->probe = 0;

// calculate window size
//计算本次可发送数据的窗口大小
cwnd = _imin_(kcp->snd_wnd, kcp->rmt_wnd);
if (kcp->nocwnd == 0) cwnd = _imin_(kcp->cwnd, cwnd);

// move data from snd_queue to snd_buf
//将snd_que的数据移动到snd_buf中并且要确保发送的数据不会超过接收方的接收队列
while (_itimediff(kcp->snd_nxt, kcp->snd_una + cwnd) < 0) {
        IKCPSEG *newseg;
        if (iqueue_is_empty(&kcp->snd_queue)) break;

        newseg = iqueue_entry(kcp->snd_queue.next, IKCPSEG, node); //snd_queue：发送消息的队列

        iqueue_del(&newseg->node);
        iqueue_add_tail(&newseg->node, &kcp->snd_buf);  //然后把删除的节点，加入到kcp的发送缓存队列中
        kcp->nsnd_que--;
        kcp->nsnd_buf++;

        newseg->conv = kcp->conv;
        newseg->cmd = IKCP_CMD_PUSH;
        newseg->wnd = seg.wnd;
        newseg->ts = current;
        newseg->sn = kcp->snd_nxt++;  //下一个待发报的序号
        newseg->una = kcp->rcv_nxt;
        newseg->resendts = current;   //下次超时重传的时间戳
        newseg->rto = kcp->rx_rto;    //由ack接收延迟计算出来的重传超时时间
        newseg->fastack = 0;          //收到ack时计算的该分片被跳过的累计次数
        newseg->xmit = 0;                       //发送该分片的次数
}

// calculate resent
//设置快重传次数和重传间隔 重传次数由kcp->fastresend设置 nodelay被激活的时候 报文的超时重传时间变为1.5倍
```

```
resent = (kcp->fastresend > 0)? (IUINT32)kcp->fastresend : 0xffffffff;
rtomin = (kcp->nodelay == 0)? (kcp->rx_rto >> 3) : 0;

// flush data segments
for (p = kcp->snd_buf.next; p != &kcp->snd_buf; p = p->next) {
        IKCPSEG *segment = iqueue_entry(p, IKCPSEG, node);
        int needsend = 0;

        //xmit为0说明是第一次发送
        if (segment->xmit == 0) {
                needsend = 1;
                segment->xmit++;
                segment->rto = kcp->rx_rto;
                segment->resendts = current + segment->rto + rtomin;  //设置下次超时重传的时间戳
        }
        //现在的时间超过重发时间但还在send_buf中需要重发 超时重发
        else if (_itimediff(current, segment->resendts) >= 0) {
                needsend = 1;
                segment->xmit++;
                kcp->xmit++;
                //更新超时重传的时间
                if (kcp->nodelay == 0) {
                        segment->rto += _imax_(segment->rto, (IUINT32)kcp->rx_rto);
                }       else {
                        IINT32 step = (kcp->nodelay < 2)?
                                ((IINT32)(segment->rto)) : kcp->rx_rto;
                        segment->rto += step / 2;
                }
                segment->resendts = current + segment->rto;
                lost = 1;   //记录一下说明出现了报文丢失
        }
        //快速重传
        else if (segment->fastack >= resent) {
                if ((int)segment->xmit <= kcp->fastlimit ||
                        kcp->fastlimit <= 0) {
                        needsend = 1;
                        segment->xmit++;
                        segment->fastack = 0;
                        segment->resendts = current + segment->rto;
                        change++;
                }
        }
        //有needsend标记的报文才发送
        if (needsend) {
                int need;
                segment->ts = current;
                segment->wnd = seg.wnd;
                segment->una = kcp->rcv_nxt;

                size = (int)(ptr - buffer);
                need = IKCP_OVERHEAD + segment->len;

                //报文长度大于mtu就要发送
                if (size + need > (int)kcp->mtu) {
                        ikcp_output(kcp, buffer, size);
                        ptr = buffer;
                }

                ptr = ikcp_encode_seg(ptr, segment);

                if (segment->len > 0) {
                        memcpy(ptr, segment->data, segment->len);
                        ptr += segment->len;
                }

                if (segment->xmit >= kcp->dead_link) {
                        kcp->state = (IUINT32)-1;
                }
        }
}

// flash remain segments
//发送上面剩余的报文
size = (int)(ptr - buffer);
if (size > 0) {
        ikcp_output(kcp, buffer, size);
}

// update ssthresh
//inflight是当前发送窗口大小，如果发送快重传将拥塞窗口阈值变为发送窗口的一半，拥塞窗口变为阈值加resent 标志着进入拥塞控制
if (change) {
        IUINT32 inflight = kcp->snd_nxt - kcp->snd_una;
        kcp->ssthresh = inflight / 2;
        if (kcp->ssthresh < IKCP_THRESH_MIN)
                kcp->ssthresh = IKCP_THRESH_MIN;
        kcp->cwnd = kcp->ssthresh + resent;
        kcp->incr = kcp->cwnd * kcp->mss;
}
```

```
        //出现超时重传说明包很可能丢了网络不畅通直接拥塞窗口变成1
        if (lost) {
                kcp->ssthresh = cwnd / 2;
                if (kcp->ssthresh < IKCP_THRESH_MIN)
                        kcp->ssthresh = IKCP_THRESH_MIN;
                kcp->cwnd = 1;
                kcp->incr = kcp->mss;
        }

        if (kcp->cwnd < 1) {
                kcp->cwnd = 1;
                kcp->incr = kcp->mss;
        }
}
```

ikcp_input

```
//---------------------------------------------------------------------
// input data
//---------------------------------------------------------------------
int ikcp_input(ikcpcb *kcp, const char *data, long size)
{
IUINT32 prev_una = kcp->snd_una; // 缓存一下当前的 snd_una
IUINT32 maxack = 0, latest_ts = 0;
int flag = 0;

if (ikcp_canlog(kcp, IKCP_LOG_INPUT)) {
        ikcp_log(kcp, IKCP_LOG_INPUT, "[RI] %d bytes", (int)size);
}

if (data == NULL || (int)size < (int)IKCP_OVERHEAD) return -1;

//循环解析数据包
while (1) {
        IUINT32 ts, sn, len, una, conv;
        IUINT16 wnd;
        IUINT8 cmd, frg;
        IKCPSEG *seg;

        if (size < (int)IKCP_OVERHEAD) break;
        //解析报文的kcp头部字段
        data = ikcp_decode32u(data, &conv);
        if (conv != kcp->conv) return -1;

        data = ikcp_decode8u(data, &cmd);
        data = ikcp_decode8u(data, &frg);
        data = ikcp_decode16u(data, &wnd);
        data = ikcp_decode32u(data, &ts);
        data = ikcp_decode32u(data, &sn);
        data = ikcp_decode32u(data, &una);
        data = ikcp_decode32u(data, &len);

        size -= IKCP_OVERHEAD;

        if ((long)size < (long)len || (int)len < 0) return -2;

        if (cmd != IKCP_CMD_PUSH && cmd != IKCP_CMD_ACK &&
                cmd != IKCP_CMD_WASK && cmd != IKCP_CMD_WINS)
                return -3;


        //获得远端的窗口大小
        kcp->rmt_wnd = wnd;

        //通过una删除send_buf中小于una的分片
        ikcp_parse_una(kcp, una);

        //更新本地的una数据
        ikcp_shrink_buf(kcp);

        if (cmd == IKCP_CMD_ACK) {
                if (_itimediff(kcp->current, ts) >= 0) {
                        //更新ack其中涉及经典的Rtt算法，第二个参数为rtt数据往返时间
                        ikcp_update_ack(kcp, _itimediff(kcp->current, ts));
                }
                //通过sn说明包收到将对应的包在snd_buf中移除
                ikcp_parse_ack(kcp, sn);
                //更新本地的una数据，因为snd_buf可能变了
                ikcp_shrink_buf(kcp);

                // 记录最大的ack包的sn值
                if (flag == 0) {
                        flag = 1;
                        maxack = sn;
```

```
                        latest_ts = ts;
                }       else {
                        if (_itimediff(sn, maxack) > 0) {
                        # ifndef IKCP_FASTACK_CONSERVE
                                maxack = sn;
                                latest_ts = ts;
                        # else
                                if (_itimediff(ts, latest_ts) > 0) {
                                        maxack = sn;
                                        latest_ts = ts;
                                }
                        # endif
                        }
                }
                if (ikcp_canlog(kcp, IKCP_LOG_IN_ACK)) {
                        ikcp_log(kcp, IKCP_LOG_IN_ACK,
                                "input ack: sn=%lu rtt=%ld rto=%ld", (unsigned long)sn,
                                (long)_itimediff(kcp->current, ts),
                                (long)kcp->rx_rto);
                }
        }
        //收到的是数据包
        else if (cmd == IKCP_CMD_PUSH) {
                if (ikcp_canlog(kcp, IKCP_LOG_IN_DATA)) {
                        ikcp_log(kcp, IKCP_LOG_IN_DATA,
                                "input psh: sn=%lu ts=%lu", (unsigned long)sn, (unsigned long)ts);
                }
                if (_itimediff(sn, kcp->rcv_nxt + kcp->rcv_wnd) < 0) {
                        //记录当前包的ack到acklist中后面会在ikcp_flush中回复ack报文
                        ikcp_ack_push(kcp, sn, ts);
                        if (_itimediff(sn, kcp->rcv_nxt) >= 0) {
                                seg = ikcp_segment_new(kcp, len);
                                seg->conv = conv;
                                seg->cmd = cmd;
                                seg->frg = frg;
                                seg->wnd = wnd;
                                seg->ts = ts;
                                seg->sn = sn;
                                seg->una = una;
                                seg->len = len;

                                if (len > 0) {
                                        memcpy(seg->data, data, len);
                                }
                                //分析data没收到过就将其加入到snd_buf中
                                ikcp_parse_data(kcp, seg);
                        }
                }
        }
        //收到的是询问窗口大小的包就设置probe 在ikcp_flush告知窗口大小
        else if (cmd == IKCP_CMD_WASK) {
                // ready to send back IKCP_CMD_WINS in ikcp_flush
                // tell remote my window size
                kcp->probe |= IKCP_ASK_TELL;
                if (ikcp_canlog(kcp, IKCP_LOG_IN_PROBE)) {
                        ikcp_log(kcp, IKCP_LOG_IN_PROBE, "input probe");
                }
        }
        else if (cmd == IKCP_CMD_WINS) {
                // do nothing
                if (ikcp_canlog(kcp, IKCP_LOG_IN_WINS)) {
                        ikcp_log(kcp, IKCP_LOG_IN_WINS,
                                "input wins: %lu", (unsigned long)(wnd));
                }
        }
        else {
                return -3;
        }
        //让data指向下一个包的首字节
        data += len;
        size -= len;
}

if (flag != 0) {
        //根据记录的maxack对snd_buf中的小于maxack的报文的fastack++用于快速重传
        ikcp_parse_fastack(kcp, maxack, latest_ts);
}

//una更新了进行流量控制或者拥塞控制
if (_itimediff(kcp->snd_una, prev_una) > 0) {
        if (kcp->cwnd < kcp->rmt_wnd) {
                IUINT32 mss = kcp->mss;
                //慢启动
                if (kcp->cwnd < kcp->ssthresh) {
                        kcp->cwnd++;
                        kcp->incr += mss;
                }
                // 拥塞控制
```

```
                else {
                        if (kcp->incr < mss) kcp->incr = mss;
                        kcp->incr += (mss * mss) / kcp->incr + (mss / 16);
                        if ((kcp->cwnd + 1) * mss <= kcp->incr) {
                        # if 1
                                kcp->cwnd = (kcp->incr + mss - 1) / ((mss > 0)? mss : 1);
                        # else
                                kcp->cwnd++;
                        # endif
                        }
                }
                if (kcp->cwnd > kcp->rmt_wnd) {
                        kcp->cwnd = kcp->rmt_wnd;
                        kcp->incr = kcp->rmt_wnd * mss;
                }
        }
}

return 0;

}
```

ikcp_recv

```
//---------------------------------------------------------------------
// user/upper level recv: returns size, returns below zero for EAGAIN
//---------------------------------------------------------------------
int ikcp_recv(ikcpcb *kcp, char *buffer, int len)
{
        struct IQUEUEHEAD *p;
        int ispeek = (len < 0)? 1 : 0;
        int peeksize;
        int recover = 0;
        IKCPSEG *seg;
        assert(kcp);

        if (iqueue_is_empty(&kcp->rcv_queue))
                return -1;

        if (len < 0) len = -len;

        //探查报文长度
        peeksize = ikcp_peeksize(kcp);

        if (peeksize < 0)
                return -2;

        if (peeksize > len)
                return -3;

        //检查是否需要进行窗口恢复
        if (kcp->nrcv_que >= kcp->rcv_wnd)
                recover = 1;

        // merge fragment
        //拷贝rcv_queue到buffer,buff就是上层取到的数据
        for (len = 0, p = kcp->rcv_queue.next; p != &kcp->rcv_queue; ) {
                int fragment;
                seg = iqueue_entry(p, IKCPSEG, node);
                p = p->next;

                if (buffer) {
                        memcpy(buffer, seg->data, seg->len);
                        buffer += seg->len;
                }

                len += seg->len;
                fragment = seg->frg;

                if (ikcp_canlog(kcp, IKCP_LOG_RECV)) {
                        ikcp_log(kcp, IKCP_LOG_RECV, "recv sn=%lu", (unsigned long)seg->sn);
                }

                if (ispeek == 0) {
                        iqueue_del(&seg->node);
                        ikcp_segment_delete(kcp, seg);
                        kcp->nrcv_que--;
                }

                // frg=0就说明包取完了
                if (fragment == 0)
                        break;
        }

        assert(len == peeksize);

        // move available data from rcv_buf -> rcv_queue
        //根据sn按序将报文将rcv_buf中的数据移到rcv_queue中, 需要先清queue后面才能将数据从buf移入queue中
```

```
        while (! iqueue_is_empty(&kcp->rcv_buf)) {
                seg = iqueue_entry(kcp->rcv_buf.next, IKCPSEG, node);
                if (seg->sn == kcp->rcv_nxt && kcp->nrcv_que < kcp->rcv_wnd) {
                        iqueue_del(&seg->node);
                        kcp->nrcv_buf--;
                        iqueue_add_tail(&seg->node, &kcp->rcv_queue);
                        kcp->nrcv_que++;
                        kcp->rcv_nxt++;
                }       else {
                        break;
                }
        }

        // fast recover
        if (kcp->nrcv_que < kcp->rcv_wnd && recover) {
                // ready to send back IKCP_CMD_WINS in ikcp_flush
                // tell remote my window size
                kcp->probe |= IKCP_ASK_TELL;
        }

        return len;
}
```

3 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。

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，可选择通过配置跳过后两步，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。