对janus的线程模型优化思路整理

janus现有的线程模型

如下图，红框表示线程，如果有多个红框，表示线程池；在janus中大量随意地使用了线程，比如对音频房间的插件里，每个room都有一个线程进行混音，每个room的participant都有一个编码线程，设想一下，如果有10个房间，每个房间有1000人，那么就会有10000个线程，这对于任何一个服务器来讲都带来了负载压力，不可取（participant线程池是我前段时间重构的，重构后100个participant大约节约了200M~300M内存）；多线程共享资源的话，就带来了“锁”的开销（锁是系统调用，同步时会有阻塞，会有上下文切换），而janus大量地采用了“锁”的方案。

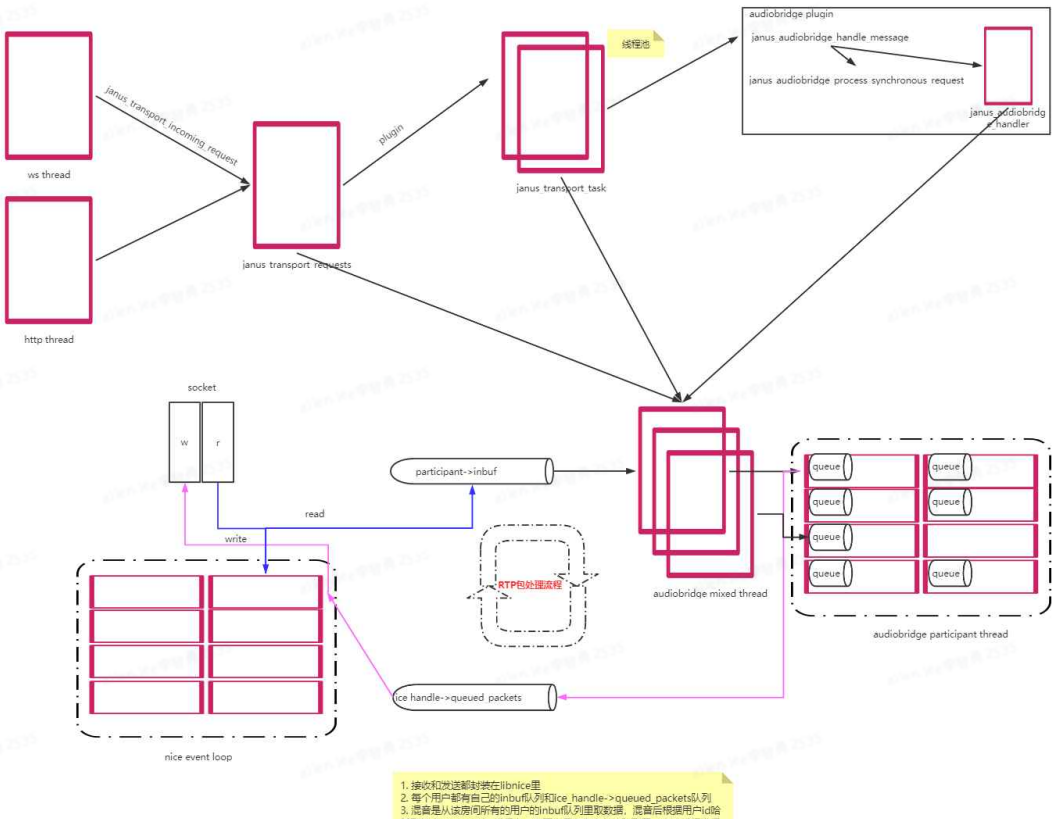
```
/* Structs */
typedef struct janus_audiobridge_room {
    guint64 room_id;    /* Unique room ID */
    gchar *room_name;   /* Room description */
    gchar *room_secret;   /* Secret needed to manipulate (e.g., destroy) this room */
    gchar *room_pin;    /* Password needed to join this room, if any */
    gboolean is_private;  /* Whether this room is 'private' (as in hidden) or not */
    uint32_t sampling_rate;  /* Sampling rate of the mix (e.g., 16000 for wideband; can be 8, 12, 16, 24 or 48kHz) */
    gboolean audiolevel_ext; /* Whether the ssrc-audio-level extension must be negotiated or not for new joins */
    gboolean audiolevel_event; /* Whether to emit event to other users about audiolevel */
    int audio_active_packets; /* amount of packets with audio level for checkup */
    int audio_level_average; /* average audio level */
    gboolean record;   /* Whether this room has to be recorded or not */
    gchar *record_file;   /* Path of the recording file */
    FILE *recording;    /* File to record the room into */
    gint64 record_lastupdate; /* Time when we last updated the wav header */
    gboolean destroy;   /* Value to flag the room for destruction */
    GHashTable *participants; /* Map of participants */
    gboolean check_tokens;  /* Whether to check tokens when participants join (see below) */
    GHashTable *allowed;  /* Map of participants (as tokens) allowed to join */
    GThread *thread;   /* Mixer thread for this room */
    volatile gint destroyed; /* Whether this room has been destroyed */
    janus_mutex mutex;   /* Mutex to lock this room instance */
    /* RTP forwarders for this room's mix */
    GHashTable *rtp_forwarders; /* RTP forwarders list (as a hashmap) */
    OpusEncoder *rtp_encoder; /* Opus encoder instance to use for all RTP forwarders */
    janus_mutex rtp_mutex;  /* Mutex to lock the RTP forwarders list */
    int rtp_udp_sock;   /* UDP socket to use to forward RTP packets */
    janus_refcount ref;   /* Reference counter for this room */
} janus_audiobridge_room;
```

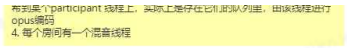
```
typedef struct janus_audiobridge_session {
    janus_plugin_session *handle;
    gint64 sdp_sessid;
    gint64 sdp_version;
    gpointer participant;
    volatile gint started;
    volatile gint hangingup;
    volatile gint destroyed;
    janus_refcount ref;
} janus_audiobridge_session;
```

```
typedef struct janus_audiobridge_participant {
    janus_audiobridge_session *session;
    janus_audiobridge_room *room; /* Room */
    guint64 user_id;  /* Unique ID in the room */
    gchar *display;   /* Display name (opaque value, only meaningful to application) */
    gboolean prebuffering; /* Whether this participant needs pre-buffering of a few packets (just joined) */
    volatile gint active; /* Whether this participant can receive media at all */
    volatile gint encoding; /* Whether this participant is currently encoding */
    volatile gint decoding; /* Whether this participant is currently decoding */
    gboolean muted;   /* Whether this participant is muted */
    int volume_gain;  /* Gain to apply to the input audio (in percentage) */
    int opus_complexity; /* Complexity to use in the encoder (by default, DEFAULT_COMPLEXITY) */
    /* RTP stuff */
    GList *inbuf;   /* Incoming audio from this participant, as an ordered list of packets */
    gint64 last_drop;  /* When we last dropped a packet because the imcoming queue was full */
    janus_mutex qmutex;  /* Incoming queue mutex */
    int opus_pt;   /* Opus payload type */
    int extmap_id;   /* Audio level RTP extension id, if any */
    int dBov_level;   /* Value in dBov of the audio level (last value from extension) */
    int audio_active_packets; /* Participant's number of audio packets to accumulate */
    int audio_dBov_sum;    /* Participant's accumulated dBov value for audio level */
    gboolean talking;  /* Whether this participant is currently talking (uses audio levels extension) */
    janus_rtp_switching_context context; /* Needed in case the participant changes room */
    /* Opus stuff */
    OpusEncoder *encoder;  /* Opus encoder instance */
    OpusDecoder *decoder;  /* Opus decoder instance */
    gboolean fec;    /* Opus FEC status */
    uint16_t expected_seq;  /* Expected sequence number */
    uint16_t probation;   /* Used to determine new ssrc validity */
    uint32_t last_timestamp; /* Last in seq timestamp */
    gboolean reset;   /* Whether or not the Opus context must be reset, without re-joining the room */

    /* added by allen.lee */
    gint thread_index;

    // move to thread pool, delete this point, add by allen.lee
    //GThread *thread;   /* Encoding thread for this participant */
    janus_recorder *arc;  /* The Janus recorder instance for this user's audio, if enabled */
    janus_mutex rec_mutex;  /* Mutex to protect the recorder from race conditions */
    volatile gint destroyed; /* Whether this room has been destroyed */
    janus_refcount ref;   /* Reference counter for this participant */
} janus_audiobridge_participant;
```





图一

并发模型正确的姿势

引用一段话：

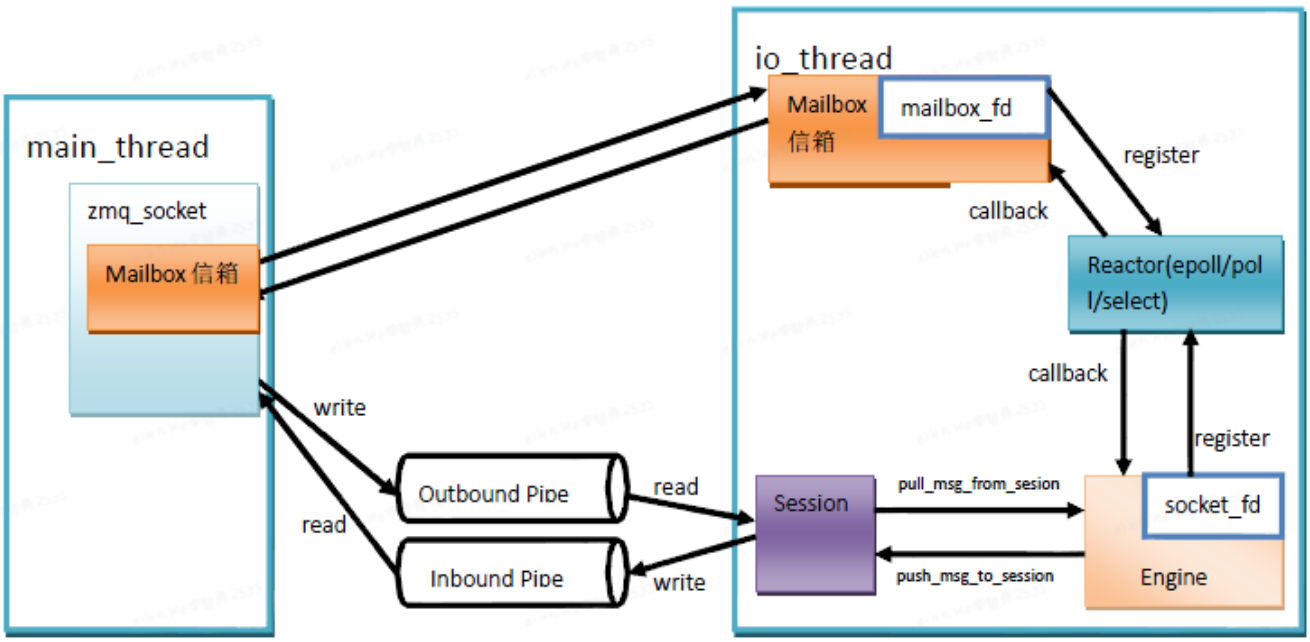
Our previous experience with messaging systems showed that using multiple threads in a classic way (critical sections, semaphores, etc.) doesn't yield much performance improvement. In fact, a multi-threaded version of a messaging system can be slower than a single-threaded one, even if measured on a multi-core box. Individual threads are simply spending too much

time waiting for each other while, at the same time, eliciting a lot of context switching that slows the system down.

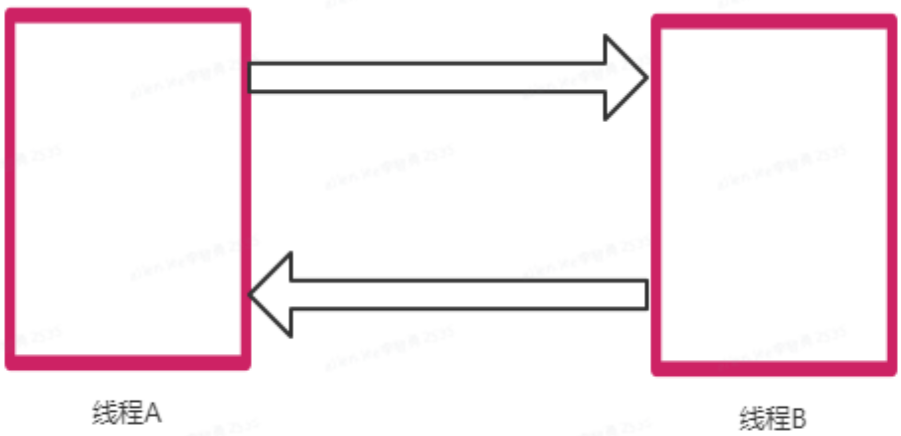
Given these problems, we've decided to go for a different model. The goal was to avoid locking entirely and let each thread run at full speed. The communication between threads was to be provided via asynchronous messages (events) passed between the threads. This, as insiders know, is the classic *actor model*.

也就是说多进程/线程并发模型中，最好是actor模型，且在线程间采用“通信”而非“锁”去做同步，比如在线程A中要设置flag为TRUE，那么较好的方式是把对该flag的操作封装成“消息”发送给线程B，然后线程B设置flag为TRUE。

在ZMQ中，线程通信的模型为：

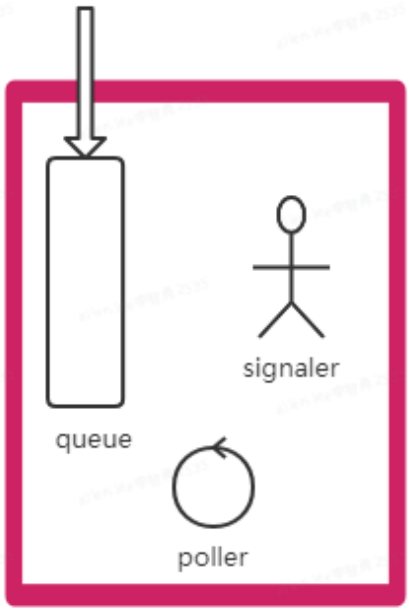


而我们现在改造的模型为：



具体点就是，线程又三个重要部分组成：poller，signaler和queue，poller就是actor模型的事件反应堆，由signaler去触发，queue就是用来存储消息的，那在这个模型中queue会是被两个线程所有，即读和写的线程，其实细微点讲，queue它由读指针和写指针组成，读线程操作读指针，写线程

操作写指针，但是如果有多个写线程怎么办？**并发编程的真谛就是不共享资源**，多个写线程解构成不同的写线程操作不同的queue就OK了，每个queue只有一个读和一个写。



线程A

当然，此次封装在ZMQ库的部分源码上进行了较少的修改，仅仅移除了读等待的操作，为了描述方便，此次封装的线程通信库命名为tsn(thread share nothing):

1. 线程间使用mailbox通信，mailbox包含了queue和signaler；
2. mailbox仅适用于单读单写的场景；
3. 支持批量写入和批量读取，即多次write后才调用signaler去唤醒读线程去读，读和写一次性移动flush（刷新）指针，细节不在展开；
4. 基于原子操作的无锁队列；
5. 跨平台。

对janus的优化

有了tsn库之后，继续进行了封装，提出了信号槽（slot）的和线程的ctx的概念，这些是借取了QT的一些概念。也即比如线程A和线程B之间要通信，那么线程A有一个slot供线程B去写入，线程B有一个slot供线程A去写入。如果有8个生产线程对8个消费线程，那么实际上就是有8*8*2共128个slot。

```
struct janus_tsn_thread_slot_s{
    tsn_mailbox_t         mailbox;
    poller_event_t        *event;
    tsn_poller_handler_t  handler;
    tsn_poller_t          *poller;
    unsigned int          slot_id;
};

typedef struct janus_tsn_thread_slot_s janus_tsn_thread_slot_t;
```

```
10
11 /* 线程上下文 */
12 struct janus_tsn_thread_ctx_s{
13    tsn_poller_t            poller;
14    janus_tsn_thread_slot_t **slots;
15    unsigned int            slots_size;
16 };
17
18 typedef struct janus_tsn_thread_ctx_s janus_tsn_thread_ctx_t;
```

有了这个封装后，还有一个就是消息的封装，这个是一个command_t的结构：

```
1  enum command_type_t
2  {
3      stop_cmd,    // mailbox destroyed, and try to exit loop
4      plug_cmd,    // insert an object to object list or hash table
5      unplug_cmd,
6      rtp_cmd,     // forward rtp package internal
7      rtp_active_cmd,
8      rtp_inactive_cmd,
9      attach_req_cmd,  // attach object to other object, when received this cmd, we
   increase ref of object
10     attach_cmd,
11     attach_ack_cmd,
12     dettach_req_cmd,
13     dettach_cmd, // dettach object to other object, when received this cmd, we
   decrease ref of object
14     dettach_ack_cmd,
15     activate_read_cmd,
16     activate_write_cmd,
17     term_req_cmd, // destroy object request
18     term_cmd,     // destroy object notify
19     term_ack_cmd, // destroy object ack
20     mod_cmd,      // modify parameter of object
21     hangup_cmd,   // hangeup session
22     done_cmd
23 };
24 struct command_s
25 {
26     //  Object to process the command.
27     void *destination;
28     int dtype;   // destination object type
29     unsigned long long obj_id;  // object id
30     tsn_command_type_t ct;   // command type
31
32     void *argv1;  // parameters
33     void *argv2;  // parameters
```

```
};
```

通过代码可知我们可以向某个线程的某个slot写入一个命令，该命令作用在某个对象上，执行什么操作，操作的一些参数是什么等等。

如图一，我们就可以把线程间共享的inbuf和queued_packets还有participant中共享的这些队列给重构为无锁异步通信的模式，并且把涉及的janus_ice_handle，janus_audiobridge_room，janus_audiobridge_participant的诸多关于创建，销毁，hangup，active等等操作需要的“锁“给去掉。

```
ps = participants_list;
while(ps) {
    janus_audiobridge_participant *p = (janus_audiobridge_participant *)ps->data;
    janus_mutex_lock(&p->qmutex);
    if(!p->session || !g_atomic_int_get(&p->session->started) ||
!g_atomic_int_get(&p->active) || p->muted || p->prebuffering || !p->inbuf) {
        janus_mutex_unlock(&p->qmutex);
        ps = ps->next;
        continue;
    }
    GList *peek = g_list_first(p->inbuf);
    janus_audiobridge_rtp_relay_packet *pkt = (janus_audiobridge_rtp_relay_packet
*)(peek ? peek->data : NULL);
 ...
 ...
   janus_mutex_unlock(&p->qmutex);
   ps = ps->next;
}
```

成了这个：

```
static void janus_audiobridge_mixer_in_event(void *argv){
    janus_tsn_thread_slot_t *slot = (janus_tsn_thread_slot_t*)argv;

    while(true){
        command_t cmd = {0};
        if (0 == janus_tsn_slot_recv(slot, &cmd)){
            // TODO
            if (cmd.ct == plug_cmd){
                janus_audiobridge_room *audiobridge = (janus_audiobridge_room
*)cmd.destination;
                janus_audiobridge_participant *participant =
(janus_audiobridge_participant *)cmd.argv1;
                janus_refcount_increase(&participant->ref);
                g_hash_table_insert(audiobridge->mixer_participants,
janus_uint64_dup(participant->user_id), participant);
            } else if (cmd.ct == unplug_cmd){
```

```
                janus_audiobridge_room *audiobridge = (janus_audiobridge_room
*)cmd.destination;
                janus_audiobridge_participant *participant =
(janus_audiobridge_participant *)cmd.argv1;
                janus_refcount_decrease(&participant->ref);
                g_hash_table_remove(audiobridge->mixer_participants,
&participant->user_id);
            } else if (cmd.ct == rtp_active_cmd){
                janus_audiobridge_participant *participant =
(janus_audiobridge_participant *)cmd.destination;
                g_atomic_int_set(&participant->active_ext, 1);
            } else if (cmd.ct == rtp_inactive_cmd){
                janus_audiobridge_participant *participant =
(janus_audiobridge_participant *)cmd.destination;
                g_atomic_int_set(&participant->active_ext, 0);
            } else if (cmd.ct == rtp_cmd){
                /* Enqueue the decoded frame */
                janus_audiobridge_participant *participant =
(janus_audiobridge_participant *)cmd.destination;
                janus_audiobridge_rtp_relay_packet *pkt =
(janus_audiobridge_rtp_relay_packet *)cmd.argv1;
                        /* Insert packets sorting by sequence number */
                        participant->inbuf = g_list_insert_sorted(participant-
>inbuf, pkt, &janus_audiobridge_rtp_sort);
                        if(participant->prebuffering) {
...
...
```

当然在这个过程中还需要做到一个尽善尽美的就是如何去批量操作RTP包，这样一次唤醒线程就可以处理完所有的RTP包，那么会减少系统调用和epoll_wait引起的线程休眠次数，降低CPU消耗，这在对libnice的改造中得到了验证。

目前在janus中的线程主要有：

一 IO类线程，包括：

a. http/https的REST API的线程池，收到数据后调用janus core的janus_transport_incoming_request处理，处理完后调用janus_http_send_message把回复的消息再挂载到REST API的线程池，该模块调用了libmicrohttpd库；

b. ws/wss的单线程，收到数据后调用janus_transport_incoming_request处理，处理完之后调用janus_websockets_send_message把回复的消息存到websocket client的messages队列，由ws/wss（websocket的API）的线程调用lws接口发送给客户端

c. rtp媒体流和ice handler对象的IO线程（hloop），收到rtp包后调用每个插件的incoming_rtp接口处理，处理后调用gateway的relay_rtp接口传回给ice handler的IO线程（hloop）

二 信令类处理线程，包括：

a. session超时检测的watch dog线程；
b. janus_transport_request线程；
c. janus_transport_task的异步请求处理线程池
d. 每个插件的handler线程，比如audiobridge的janus_audiobridge_handler线程，videoroom的janus_videoroom_handler线程等等，他们的请求来自a，b和c的信令处理线程；

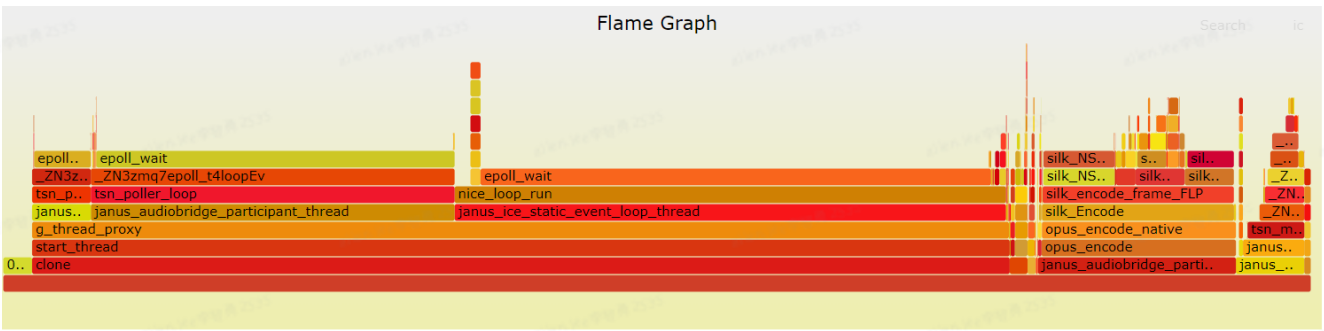
三 插件的媒体处理线程，包括：

a. audiobridge的mix线程，每个房间对应一个；
b. audiobridge的participant线程，主要是opus编码；
c. videoroom的relay_rtp线程；

第一类线程是数据的来源和出口，IO密集型的；第二类和第三类是媒体和信令处理的线程，CPU密集型的；第一类的IO对象挂断时会调用janus_transport_gone销毁所有的session或者是ice模块调用插件的hangup_media和destroy_session做相应处理，也就是说第一类控制着第二类及第三类线程中的对象；反过来所有的信令线程和媒体处理线程也控制着第一类线程，比如destory和changeroom，kick等等信令；我们需要做的就是线程间共享的对象变成不共享，那我们除了线程间通信，更应该把第二类和第三类线程依据会话（janus_session）属性，把某个用户相关的所有对象包括RTP媒体处理信息等固定在某个线程，这样媒体属性的变化就不再需要进行线程间通信处理，也就是媒体处理和信令处理放在同一个线程处理；第一类和第二，三类之间需要传递hange_media/rtp/destroy_session/request/message等命令，也就是说只有rtp和对象创建销毁等级别的命令，而没有属性变化（共享的就是属性，带来了原子锁和mutex操作）的命令。

遇到的坑

在系列优化前，先做了个对比mutex的压测，压测采用的方法是8个生产线程对应8个消费线程，为了测试的公平性，随机操作某个对象的mutex，对比的结果是tsn比mutex要高30~50倍；至此，略有欣慰，但是引入到janus后，测试100路音频时，却发现CPU几乎干到了100%，这是为什么呢？于是抓了火焰图如下：



发现epoll_wait调用的几乎占有了60~70%的CPU，这是因为tsn的poller是基于epoll进行封装的，但是我们每插入一条command，都会调用signaler去唤醒一个poller，进而去读取queue的

command并进行相应处理，epoll_wait会引起线程休眠，会导致线程上下文切换，而线程上下文切换是十分耗时的。针对此问题就是减少对poller的唤醒，途径如下：

1. 尽量批量write命令到tsn的slot（queue）里；
2. 检测write命令的流速，如果流速较快，调用poll/select进行2~3毫秒的休眠；

其中第二点在zmq中已经实现，但是此次优化未检测流速，仅仅是如果发现有多次要空转进入epoll_wait前，就主动调用poll/select进行休眠，这样会带来一个好处是尽量让写线程多write一些command到slot里，然后批量处理这些command。

后记：janus在模块设计中确实很优秀，比如插件化封装，提高了扩展性，但是插件是基于接口调用封装的，比如我们要调用hangup_session，那就是一种同步方式，这种同步的调用如果发生在不同的线程，那么会是一场灾难，比如现在可以见到的各种锁，还有基于次业务流程很僵硬，带来的后果是性能低，二次开发的成本非常高，而在业界有一个EasyDarwin的流媒体开源项目，却完全是基于消息驱动的插件似开发，除了可扩展性之外，做到了按照io的fd进行负载均衡，还有一个用户的所有的消息只会映射到某个固定的线程上处理，那么这就带来了单线程的顺序化处理，即很多时候根本不用加锁，同时插件可以叠加对RTP包处理，即过滤器模式的设计，这在以后的业务开发中会带来很高的开发效率。

tsn ：http://git.ids111.com/idreamsky/social/janus/tree/dev/tsn