Skynet基本数据结构----消息队列

我们的创建的服务，需要通过消息来驱动，而一个服务要获取消息，是从消息队列里取的。skynet包含两级消息队列，一个global_mq，他包含一个head和tail指针，分别指向次级消息队列的头部和尾部，另外还有一个次级消息队列，这个一个单向链表。消息的派发机制是，工作线程，会从global_mq里pop一个次级消息队列来，然后从次级消息队列中，pop出一个消息，并传给context的callback函数，在完成驱动以后，再将次级消息队列push回global_mq中，数据结构如下所示：

```
// skynet_mq.h
struct skynet_message {
    uint32_t source;            // 消息发送方的服务地址

    // 如果这是一个回应消息，那么要通过session找回对应的一次请求，在lua层，我们每次调用call的时候，都会往对
    // 方的消息队列中，push一个消息，并且生成一个session，然后将本地的协程挂起，挂起时，会以session为key，协程句
    // 柄为值，放入一个table中，当回应消息送达时，通过session找到对应的协程，并将其唤醒。后面章节会详细讨论
    int session;

    void * data;        // 消息地址
    size_t sz;          // 消息大小
};

// skynet_mq.c
#define DEFAULT_QUEUE_SIZE 64
#define MAX_GLOBAL_MQ 0x10000

// 0 means mq is not in global mq.
// 1 means mq is in global mq , or the message is dispatching.

#define MQ_IN_GLOBAL 1
#define MQ_OVERLOAD 1024

struct message_queue {
    // 自旋锁，可能存在多个线程，向同一个队列写入的情况，加上自旋锁避免并发带来的发现，
    //后面会讨论互斥锁，自旋锁，读写锁和条件变量的区别
    struct spinlock lock;

    uint32_t handle;                // 拥有此消息队列的服务的id
    int cap;                        // 消息大小
    int head;                       // 头部index
    int tail;                       // 尾部index
    int release;                    // 是否能释放消息
    int in_global;                  // 是否在全局消息队列中，0表示不是，1表示是
    int overload;                   // 是否过载
    int overload_threshold;
    struct skynet_message *queue;   // 消息队列
    struct message_queue *next;     // 下一个次级消息队列的指针
};

struct global_queue {
    struct message_queue *head;
    struct message_queue *tail;
    struct spinlock lock;
};

static struct global_queue *Q = NULL;
```

上面我们已经讨论了，一个服务如何被消息驱动，现在我们来讨论，消息是如何写入到消息队列中去的。我们要向一个服务发消息，最终是通过调用skynet.send接口，将消息插入到该服务专属的次级消息队列的，次级消息队列的内容，并不是context结构的一部分（context只是引用了他的指针），因此，在一个服务执行callback的同时，其他服务（可能是多个线程内执行callback的其他服务）可以向它的消息队列里push消息，而mq的push操作，是加了一个自旋锁，以避免多个线程，同时操作一个消息队列。**lua层的skynet.send接口，最终会调到c层的skynet_context_push**。这个接口实质上，是通过**handle**将**context**指针取出来，然后再往消息队列里**push**消息：

```
// skynet_server.c
int
skynet_context_push(uint32_t handle, struct skynet_message *message) {
    struct skynet_context * ctx = skynet_handle_grab(handle);
    if (ctx == NULL) {
        return -1;
    }
    skynet_mq_push(ctx->queue, message);
    skynet_context_release(ctx);

    return 0;
}

// skynet_handle.c
struct skynet_context *
skynet_handle_grab(uint32_t handle) {
    struct handle_storage *s = H;
    struct skynet_context * result = NULL;

    rwlock_rlock(&s->lock);

    uint32_t hash = handle & (s->slot_size-1);
    struct skynet_context * ctx = s->slot[hash];
    if (ctx && skynet_context_handle(ctx) == handle) {
        result = ctx;
        skynet_context_grab(result);
    }

    rwlock_runlock(&s->lock);

    return result;
}
```

因为我们访问一个服务的机会，远大于创建一个服务并写入列表的机会，因此这里用了读写锁，在通过handle获取context指针时，加了一个读取锁，这样当在读取的过程中，同时有新的服务创建，并且存在要扩充skynet_context list容量的风险，因此不论如何，他都应当被阻塞住，直到所有的读取锁都释放掉。 次级消息队列，实际上是一个数组，并且用两个**int**型数据，分别指向他的头部和尾部（**head**和**tail**），不论是**head**还是**tail**，当他们的值**>=**数组尺寸时，都会进行回绕（即从下标为**0**开始，比如值为数组的**size**时，会被重新赋值为**0**），在**push**操作后，**head**等于**tail**意味着队列已满（此时，队列会扩充两倍，并从头到尾重新赋值，此时**head**指向**0**，而**tail**为扩充前，数组的大小），在**pop**操作后，**head**等于**tail**意味着队列已经空了（后面他会从**skynet**全局消息队列中，被剔除掉）。

head < tail的情况

head > tail的情况

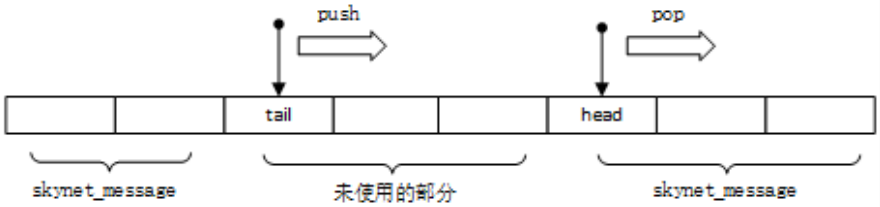

图3