skynet 学习笔记 - 消息队列

介绍

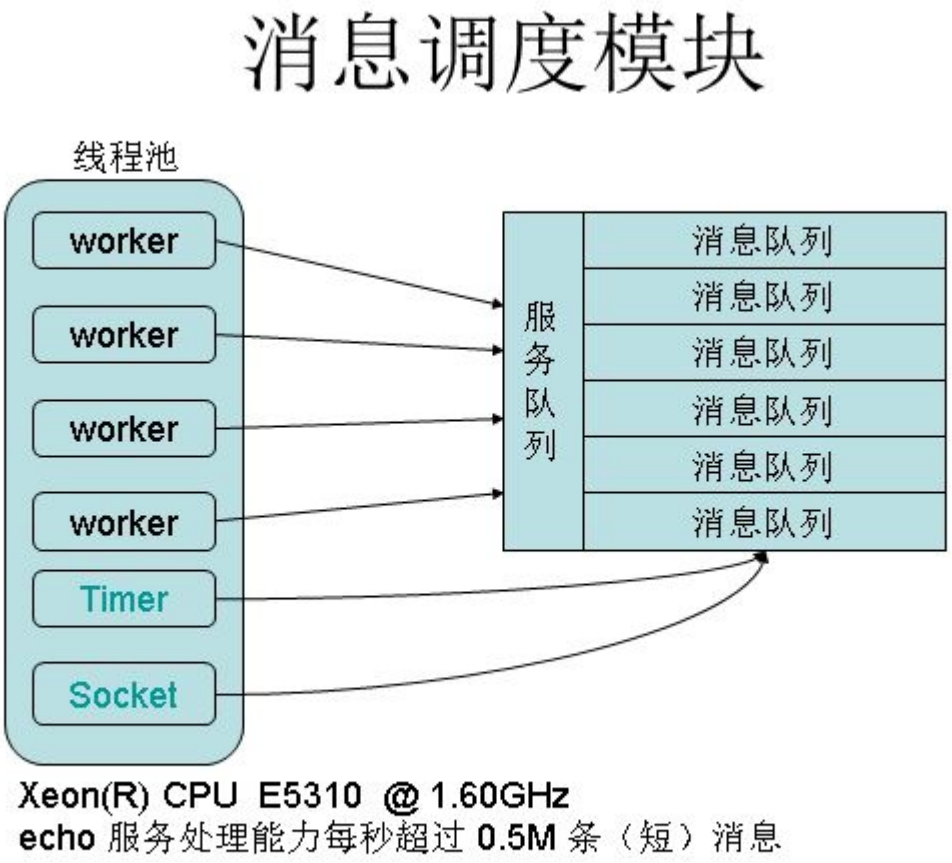
Skynet 是一个为网络游戏服务器设计的轻量框架。

这个游戏框架的特点是：

- 实现一个类似 Erlang 的 Actor 模型的服务端编程环境
- 运行效率高，追求单机性能
- 不关注分布式，追求高实时的相应速度
- 业务层采用 Lua 沙盒，开发调试方便。
- 适合对 C/Lua 熟悉的团队

消息调度机制

Skyent 核心部分是一个消息调度机制。示意图如下



说明:

- Skynet 是一个独立的进程，其中运行着若干个 worker 线程。
- worker 线程会从消息队列中取出消息，找到对应的处理函数，进行分发。
- timer 线程实现了定时机制。
- socket 线程主要工作是监听 epoll 事件，管理网络操作。

我会用单独的篇幅来分析各个重点

- 消息队列
- 服务
- 沙盒服务
- snlua 调度
- 定时器

- Socket
- 玩家代理服务

首先是消息队列

消息队列

Skynet 维护了两级消息队列。

- 每个服务都有一个私有的消息队列。队列中是一个个发送给它的消息。
- 一个全局消息队列。里面放的是若干个不为空的服务队列。

下面，主要内容是

- 服务队列的结构
- 服务队列操作
- 全局消息队列的结构
- 全局消息队列操作
- 工作线程分发消息
- 参考资料

服务队列的结构

```
struct message_queue {

    uint32_t handle;   // 服务地址

    int cap;           // 数组

    int head;          // queue 实现了循环队列。 head 和 tail 分别是头和尾

    int tail;

    struct skynet_message *queue;    // 保存消息的数组.

    struct message_queue *next;

    ...
```

```
};
```

由于服务队列是属于服务的，所以服务队列的生命周期和服务一致：载入服务的时候生成，卸载服务的时候删除。

服务是通过 *skynet_context_new* 载入的，在此函数中，可以找到对应的服务队列的生成语句：

```
struct message_queue * queue = ctx->queue = skynet_mq_create(ctx->handle);



struct message_queue *
skynet_mq_create(uint32_t handle) {
    struct message_queue *q = skynet_malloc(sizeof(*q));
    q->handle = handle;
    q->cap = DEFAULT_QUEUE_SIZE;          // 队列大小
    q->head = 0;
    q->tail = 0;
    // ...
    q->queue = skynet_malloc(sizeof(struct skynet_message) * q->cap);
    q->next = NULL;


    return q;
}
```

handle 就是队列所属服务的地址。 通过 handle, 就可以找到服务对应的结构体。

可以看到，queue 是个数组，可以存放 DEFAULT_QUEUE_SIZE 个消息， 默认大小是 1024 个。

实际上 queue 是用数组实现了一个循环队列。

服务队列操作

服务队列主要支持 2 个操作

- 向服务队列中添加消息
- 从服务队列中取出消息

添加消息到队列中，如果队列满了，会触发自动扩容的操作 *expand_queue*. 新扩容的数组大小是原先的 2 倍。

```
void

skynet_mq_push(struct message_queue *q, struct skynet_message *message) {

    q->queue[q->tail] = *message;

    if (++ q->tail >= q->cap) {

        q->tail = 0;

    }

    if (q->head == q->tail) {

        expand_queue(q);

    }
```

```
    ...

}
```

1
2
3
4
5
6
7
8
9
10
11
12
13

那么，取出消息后，数组占有空间少的时候，会收缩吗？答案是不。

```
int

skynet_mq_pop(struct message_queue *q, struct skynet_message *message) {

    // ...

    if (q->head != q->tail) {

        // 这个就是取出的消息

        *message = q->queue[q->head++];

        int head = q->head;

        int tail = q->tail;

        int cap = q->cap;

        // head 重新指向数组头

        if (head >= cap) {
```

```
            q->head = head = 0;
        }
        int length = tail - head;
        if (length < 0) {
            length += cap;
        }
    }

    // ...
}
```

全局消息队列的结构

Skynet 进程只有 1 个全局消息队列。 在 Skynet 启动的时候会进行初始化。

```
    skynet_mq_init();


    struct global_queue {
        struct message_queue *head;        // 指向第一个服务队列
        struct message_queue *tail;        // 指向最后一个服务队列
        struct spinlock lock;              // 并发同步
    };
```

你可能很好奇， 这个链表，如何指向下一个成员。 其实之前已经提到了

```
struct message_queue {
    ...
    struct message_queue *next;
}
```

为了效率，并不是简单的把所有的服务队列都塞到全局队列中，而是只塞入非空的服务队列，

这样 worker 线程就不会得到空的服务队列而浪费 CPU。

全局消息队列操作

全局消息队列主要支持 2 个操作

- 向全局队列中添加服务队列
- 从全局队列中取出服务队列

添加服务队列

```
void
skynet_globalmq_push(struct message_queue * queue) {
    struct global_queue *q= Q;
    SPIN_LOCK(q)
    assert(queue->next == NULL);
    if(q->tail) {
        q->tail->next = queue;
        q->tail = queue;
```

```
    } else {
        q->head = q->tail = queue;
    }
    SPIN_UNLOCK(q)
}
```

取出服务队列

```
struct message_queue *
skynet_globalmq_pop() {
    struct global_queue *q = Q;

    SPIN_LOCK(q)
    struct message_queue *mq = q->head;
    if(mq) {
        q->head = mq->next;
        if(q->head == NULL) {
            assert(mq == q->tail);
            q->tail = NULL;
        }
        mq->next = NULL;
    }
```

```
    SPIN_UNLOCK(q)



    return mq;

}
```

这些操作都使用了锁进行保护。早期版本，采用无锁机制，结果引入了并发的 BUG 。

工作线程分发消息

Skynet 启动多个 worker 线程进行消息分发，线程个数是可以设置的，官方建议配置为 cpu 核数。

每个 worker 线程执行的入口函数是 *thread_worker* 。

```
    // skynet_start.c#start
    for (i=0;i<thread;i++) {
        // ...
        create_thread(&pid[i+3], thread_worker, &wp[i]);
    }
```

忽略枝叶， *thread_worker* 会不停地调用 *skynet_context_message_dispatch*

```
struct message_queue *

skynet_context_message_dispatch(struct skynet_monitor *sm, struct message_queue *q, int weight) {
    // 从全局队列中取出一个服务队列
```

```
if (q == NULL) {
    q = skynet_globalmq_pop();
    if (q==NULL)
        return NULL;
}

// 找到服务队列所属的服务上下文
uint32_t handle = skynet_mq_handle(q);
struct skynet_context * ctx = skynet_handle_grab(handle);
// ...

// 为了调度公平，每次只弹出一个消息
int i,n=1;
struct skynet_message msg;

for (i=0;i<n;i++) {
    if (skynet_mq_pop(q,&msg)) {
        skynet_context_release(ctx);
        return skynet_globalmq_pop();
    } else if (i==0 && weight >= 0) {
```

```
            n = skynet_mq_length(q);
            n >>= weight;
        }
        // 检查服务是否过载
        int overload = skynet_mq_overload(q);
        if (overload) {
            skynet_error(ctx, "May overload, message queue length = %d", overload);
        }

        skynet_monitor_trigger(sm, msg.source , handle);

        if (ctx->cb == NULL) {
            skynet_free(msg.data);
        } else {
            // 进行消息分发
            dispatch_message(ctx, &msg);
        }

        skynet_monitor_trigger(sm, 0,0);
    }
```

```
    // ...

    return q;
}
```

函数 *_dispatch_message* 会调用这个服务的 callback 。

```
static void
_dispatch_message(struct skynet_context *ctx, struct skynet_message *msg) {
    int type = msg->sz >> HANDLE_REMOTE_SHIFT;
    size_t sz = msg->sz & HANDLE_MASK;
    // ...
    if (!ctx->cb(ctx, ctx->cb_ud, type, msg->session, msg->source, msg->data, sz))
        skynet_free(msg->data);
}
```