云风的 BLOG

思绪来得快去得也快，偶尔会在这里停留

« skynet 的新组播方案 | 返回首页 | skynet v0.2.0 发布 »

skynet 消息队列调度算法的一点说明

最近接连有几位同学询问 skynet 的消息队列算法中为什么引入了一个独立的 flags bool 数组的问题。时间久远，我自己差点都忘记设计初衷了。今天在代码里加了点注释，防止以后忘记。

其实当时我就写过一篇 blog 记录过，这篇 blog 下面的评论中也有许多讨论。今天把里面一些细节再展开说一次：

我用了一个循环队列来保存 skynet 的二级消息队列，代码是这样的：

```
#define GP(p) ((p) % MAX_GLOBAL_MQ)

static void
skynet_globalmq_push(struct message_queue * queue) {
    struct global_queue *q= Q;

    uint32_t tail = GP(__sync_fetch_and_add(&q->tail,1));
    q->queue[tail] = queue;
    __sync_synchronize();
    q->flag[tail] = true;
}

struct message_queue *
skynet_globalmq_pop() {
    struct global_queue *q = Q;
    uint32_t head =  q->head;
    uint32_t head_ptr = GP(head);
    if (head_ptr == GP(q->tail)) {
        return NULL;
    }

    if(!q->flag[head_ptr]) {
        return NULL;
    }

    __sync_synchronize();

    struct message_queue * mq = q->queue[head_ptr];
    if (!__sync_bool_compare_and_swap(&q->head, head, head+1)) {
        return NULL;
    }
    q->flag[head_ptr] = false;

    return mq;
}
```

有同学问我，为什么要用一个单独的 flag 数组。用指针数组里的指针是否为空来判断不是更简单吗？

见 skynet 的第 68 个 PR 。

代码可以写成这样：

```
#define GP(p) ((p) % MAX_GLOBAL_MQ)
```

```
static void
skynet_globalmq_push(struct message_queue * queue) {
    struct global_queue *q= Q;

    uint32_t tail = GP(__sync_fetch_and_add(&q->tail,1));
    // 如果线程在这里挂起，q->queue[tail] 将不为空，
    // 却没有更新到新的值
    q->queue[tail] = queue;
}

struct message_queue *
skynet_globalmq_pop() {
    struct global_queue *q = Q;
    uint32_t head =  q->head;
    uint32_t head_ptr = GP(head);
    if (head_ptr == GP(q->tail)) {
        return NULL;
    }

    if (!q->queue[head_ptr]) {
        return NULL;
    }
    struct message_queue * mq = q->queue[head_ptr];
    // 这里无法确保 mq 读到的是 push 进去的值。
    // 它有可能是队列用完一圈后，上一个版本的值。
    if (!__sync_bool_compare_and_swap(&q->head, head, head+1)) {
        return NULL;
    }

    q->queue[head_ptr] = NULL;

    return mq;
}
```

这样做其实是有陷阱的，我标记在里代码中。

这种情况只有在 64K 的队列全部转过一圈，某个 push 线程一直挂起在递增 tail 指针，还来不及写入新的值的位置。

看起来这种情况很难发生，但在我们早期的测试中的确出现了。

所以说，并发程序写起来一定要特别谨慎（不要随便改动之前推敲过的代码）。一不小心就掉坑里了。

另外，以前的代码有一个限制：当活跃的（有消息的）服务总数超过 64K 的时候，这段代码就不能正常工作了。虽然一个 skynet 节点中的服务数量很难超过这个限制（因为无消息的服务不会在全局队列中），但理论上一个 skynet 节点支持的服务数量上限是远大于 64K 的。

我这次在增加注释的同时加了几行代码，用了一个额外的链表来保存那些因为队列满无法立刻排进去的服务。在出队列的时候，再尝试把它们从链表中取回来。

```
static void
skynet_globalmq_push(struct message_queue * queue) {
    struct global_queue *q= Q;

    if (q->flag[GP(q->tail)]) {
        // The queue may full seldom, save queue in list
        assert(queue->next == NULL);
        struct message_queue * last;
        do {
            last = q->list;
```

```
            queue->next = last;
        } while(!__sync_bool_compare_and_swap(&q->list, last, queue));

        return;
    }

    uint32_t tail = GP(__sync_fetch_and_add(&q->tail,1));
    // The thread would suspend here, and the q->queue[tail] is last version ,
    // but the queue tail is increased.
    // So we set q->flag[tail] after changing q->queue[tail].
    q->queue[tail] = queue;
    __sync_synchronize();
    q->flag[tail] = true;
}

struct message_queue *
skynet_globalmq_pop() {
    struct global_queue *q = Q;
    uint32_t head =  q->head;

    if (head == q->tail) {
        // The queue is empty.
        return NULL;
    }

    uint32_t head_ptr = GP(head);

    struct message_queue * list = q->list;
    if (list) {
        // If q->list is not empty, try to load it back to the queue
        struct message_queue *newhead = list->next;
        if (__sync_bool_compare_and_swap(&q->list, list, newhead)) {
            // try load list only once, if success , push it back to the queue.
            list->next = NULL;
            skynet_globalmq_push(list);
        }
    }

    // Check the flag first, if the flag is false, the pushing may not complete.
    if(!q->flag[head_ptr]) {
        return NULL;
    }

    __sync_synchronize();

    struct message_queue * mq = q->queue[head_ptr];
    if (!__sync_bool_compare_and_swap(&q->head, head, head+1)) {
        return NULL;
    }
    q->flag[head_ptr] = false;

    return mq;
}
```

云风 提交于 May 7, 2014 01:53 PM | 固定链接

COMMENTS

可以研究下 Disruptor 的锁实现。。。很有价值的。。

Posted by: yynote | (7) August 8, 2014 11:24 AM

// but the queue tail is increased.

Posted by: 庞统 | (6) June 5, 2014 05:21 AM

rf////

Posted by: 庞统 | (5) June 5, 2014 05:16 AM

大师，看你的消息队列设计，逻辑层从工作线程里取消息出来然后处理消息，逻辑线程对共享数据，还是要不断的加锁吧。是不是逻辑层编写起来非常复杂。

Posted by: beginer | (4) May 11, 2014 11:45 AM

先填入queue[tail]，然后再tail + 1，这样做是否可以避免读取错误的问题？

Posted by: psybeing | (3) May 8, 2014 04:58 PM

这个也是并发的常见坑...
串行处理被并发操作时，闭锁条件必需是一个独立且最后操作的开关

Posted by: Julius | (2) May 8, 2014 10:43 AM

循环队列的想法很不错，就是看了这个想法，才想到放到我自己的网络消息里的。

Posted by: lite3 | (1) May 7, 2014 06:17 PM

POST A COMMENT

非这个主题相关的留言请到：留言本

名字：

Email 地址：

为了验证您是人类，请将六加一的结果（阿拉伯数字七）填写在下面：

URL:

记住我的信息?

留言：
（不欢迎在留言中粘贴程序代码）

提交