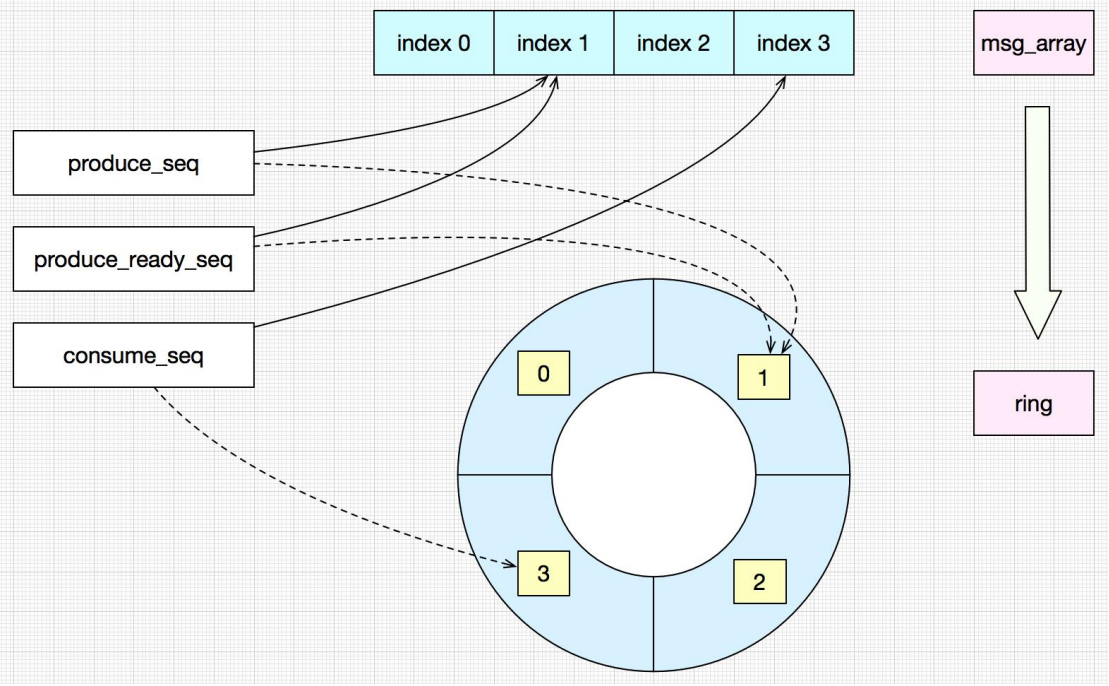
作者：代兴　邮箱：503268771@qq.com　Vx 公众号：东方架构师
https://github.com/drink-cat/Book_Program-Principles

ringbuffer 的含义和示意图

ringbuffer 表示环形队列，把数组包装为逻辑圆环，用 CAS 指令操作序号实现并发安全。
ringbuffer 适用于生产者消费者模式，生产者往队列写消息，消费者从队列读消息。
序号 seq 是核心，记录消息编号，多线程 CAS 操作序号，判断边界条件(队列满、队列空)。

ringbuffer 的示意图：（可以有多种变体）



使用 C 和 CAS 指令实现自定义的 ringbuffer

编写代码： ringbuffer.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <syscall.h>
#include <stdbool.h>

// 消息数组的大小。
#define msg_array_size 4

// 环形队列
struct ring_buffer
```

```
{
    volatile uint64_t produce_seq;         // 生产的序号
    volatile uint64_t produce_ready_seq;   // 生产就绪的序号
    volatile uint64_t consume_seq;         // 消费的序号
    struct msg *msg_array[msg_array_size]; // 数组。元素是指针。
};

// 一个消息
struct msg
{
    char name[30];
    int index;
};

struct ring_buffer buffer = {
    .produce_seq = 0, .produce_ready_seq = 0, .consume_seq = 0};

// 生产者。创建 1 个消息
void func_produce_msg()
{
    // 线程 ID
    pid_t taskid = syscall(SYS_gettid);
    while (true)
    {
        // 如果生产速度超过消费速度，需要等待消息被消费
        if (buffer.produce_seq >= (buffer.consume_seq + msg_array_size))
        {
            continue;
        }

        // 获得下一个生产的序号
        buffer.produce_seq++;
        uint64_t next_seq = buffer.produce_seq;

        // 创建一个消息。放入数组。
        struct msg *msg = malloc(sizeof(struct msg));
        int index = next_seq % msg_array_size;
        sprintf(msg->name, "Msg-%d-%llu", taskid, next_seq);
        msg->index = index;
        buffer.msg_array[index] = msg;

        // 标识产生了新的消息
        printf(" thread = %d Produce seq = %llu msg = %s \n",
               taskid, next_seq, msg->name);
        buffer.produce_ready_seq = next_seq;

        // 控制速率。方便测试多种场景。
        // sleep(2);
    }
```

```
}

// 消费者。消费 1 个消息
void func_consume_msg()
{
    // 线程 ID
    pid_t taskid = syscall(SYS_gettid);

    while (true)
    {
        // 检查是否有新的消息
        uint64_t ready_seq = buffer.produce_ready_seq;
        uint64_t consume_seq = buffer.consume_seq;
        // 如果没有新的消息，就继续检查
        if (consume_seq >= ready_seq)
        {
            continue;
        }
        // 有新消息，尝试获得这个消息
        bool pick_ok = __sync_bool_compare_and_swap(
            &buffer.consume_seq, consume_seq, consume_seq + 1);
        // 如果竞争失败，就继续检查
        if (!pick_ok)
        {
            continue;
        }
        // 当前线程，竞争到这个消息。消费这个消息
        uint64_t now_seq = consume_seq + 1;
        int index = now_seq % msg_array_size;
        struct msg *msg = buffer.msg_array[index];
        buffer.msg_array[index] = NULL;
        printf(" thread = %d Consume seq = %llu msg = %s \n",
               taskid, now_seq, msg->name);
        free(msg);

        // 控制速率。方便测试多种场景。
        sleep(2);
    }
}

int main()
{
    // 进程 ID
    pid_t pid = syscall(SYS_getpid);
    printf(" process = %d \n", pid);

    // 单个生产者
    pthread_t thread_producer1;
    pthread_create(&thread_producer1, NULL, func_produce_msg, NULL);
```

```
    // 多个消费者
    pthread_t thread_consumer1;
    pthread_create(&thread_consumer1, NULL, func_consume_msg, NULL);
    pthread_t thread_consumer2;
    pthread_create(&thread_consumer2, NULL, func_consume_msg, NULL);

    sleep(80000);
    return 0;
}
```

编译代码：

```
gcc ringbuffer.c -lpthread -o ringbuffer
```

运行代码：

```
[root@local ringbuffer]# ./ringbuffer
 process = 16912
 thread = 16913 Produce seq = 1 msg = Msg-16913-1
 thread = 16915 Consume seq = 1 msg = Msg-16913-1
 thread = 16913 Produce seq = 2 msg = Msg-16913-2
 thread = 16913 Produce seq = 3 msg = Msg-16913-3
 thread = 16913 Produce seq = 4 msg = Msg-16913-4
 thread = 16913 Produce seq = 5 msg = Msg-16913-5
 thread = 16913 Produce seq = 6 msg = Msg-16913-6
 thread = 16914 Consume seq = 2 msg = Msg-16913-2
 thread = 16915 Consume seq = 3 msg = Msg-16913-3
 thread = 16913 Produce seq = 7 msg = Msg-16913-7
 thread = 16914 Consume seq = 4 msg = Msg-16913-4
 thread = 16913 Produce seq = 8 msg = Msg-16913-8
 thread = 16915 Consume seq = 5 msg = Msg-16913-5
 thread = 16913 Produce seq = 9 msg = Msg-16913-9
```

查看线程状态：

```
[root@local ringbuffer]# top -H -p 16912
top - 11:50:50 up 19 days, 17:56,  1 user,  load average: 0.64, 0.36, 0.19
Threads:   4 total,   1 running,   3 sleeping,   0 stopped,   0 zombie
%Cpu(s): 34.3 us,  1.0 sy,  0.0 ni, 64.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  2851708 total,   636536 free,   453936 used,  1761236 buff/cache
KiB Swap:  2097148 total,  2097148 free,        0 used.  2072052 avail Mem

   PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
 16913 root      20   0   96632    396    312 R 92.3  0.0   0:29.63 ringbuffer
 16912 root      20   0   96632    396    312 S  0.0  0.0   0:00.00 ringbuffer
 16914 root      20   0   96632    396    312 S  0.0  0.0   0:00.00 ringbuffer
 16915 root      20   0   96632    396    312 S  0.0  0.0   0:00.00 ringbuffer
```

分析结果：

代码为单生产者多消费者模式，没有设置退出条件，生产者线程、消费者线程一直运行，这里截取起始的输出。

生产者线程，使用函数 func_produce_msg 生产消息。因为只有 1 个线程，所以不需要 CAS 指令。

消费者线程，使用函数 func_consume_msg 消费消息。因为有多个线程，所以需要 CAS 指令，这里使用 __sync_bool_compare_and_swap。
生产者线程的 ID 为 16913，在循环中判断序号状态，使得 CPU 使用率达到 92.3。
消费者线程的 ID 为 16914、16915，使用 sleep(2)延长消费的耗时，表示生产快、消费慢的场景。

环形队列 ring_buffer，包含属性 produce_seq、produce_ready_seq、consume_seq、msg_array。
属性 produce_seq，表示生产的序号。创建消息，序号依次加 1，把消息写到消息数组的对应位置。
属性 produce_ready_seq，表示生产就绪的序号。创建消息完成后，再更新此属性，影响后续的消费操作。
属性 consume_seq，表示消费的序号。消费消息，序号依次加 1，从消息数组的对应位置读出消息。
属性 msg_array，表示消息数组。容量有限，为了防止覆盖元素，必须判断 seq，数组有空余位置才能创建消息。
属性 produce_seq、produce_ready_seq、consume_seq 使用 volatile 修饰，加强可见性。

使用 sleep，控制生产或消费的速率，方便测试多种场景。
如果生产速度大于消费速度，则队列被消息占满，队列没有位置放置新的消息。此时，生产者线程自旋等待。
如果消费速度大于生产速度，则队列有空闲位置，队列可以放置新的消息。如果消息被消费完，消费者线程自旋等待。