一觉醒来写程序

DPDK 无锁队列Ring Library原理（学习笔记）

参考自DPDK官方文档原文：http://doc.dpdk.org/guides-20.02/prog_guide/ring_lib.html

针对自己的理解做了一些辅助解释。

1 前置知识

1.1 CAS

学习无锁队列前先看一个基本概念，CAS原子指令操作。

CAS（Compare and Swap，比较并替换）原子指令，用来保障数据的一致性。

指令有三个参数，当前内存值V、旧的预期值A、更新的值B，当且仅当预期值A和内存值V相同时，将内存值修改为B并返回true，否则什么都不做，并返回false。

在DPDK中封装后的函数如下：

```
rte_atomic32_cmpset(&r->prod.head, *old_head, *new_head)
```

&r->prod.head指向当前内存值，*old_head为执行该操作前将r->prod.head存储到临时变量的值，*new_head为即将更新的值。

只有r->prod.head == *old_head才会将r->prod.head更新为*new_head

1.2 其他ring实现的参考（了解）

1）FreeBSD中Ring实现的参考

在FreeBSD 8.0中添加了以下代码，并在某些网络设备驱动程序中使用（至少在Intel驱动程序中）：

- bufring.h in FreeBSD
- bufring.c in FreeBSD

2）Linux中无锁Ring的实现

http://lwn.net/Articles/340400/

2 Ring Library

2.1 介绍

ring是一个有限大小的链表，它具有以下属性：

- FIFO(First Input First Output)简单说就是指先进先出
- 大小固定，指针存储在表中
- 无锁实现

- 多消费者或单消费者出队
- 多生产者或单生产者入队
- 批量（Bulk）出队：如果成功，将指定数量的对象出队; 否则失败
- 批量入库：如果成功，将指定数量的对象入队; 否则失败
- 爆发（Burst）出队：如果指定数量的对象无法满足，则将最大可用数量的对象出队
- 爆发入队：如果指定数量的对象无法满足，则将最大可用数量的对象入队

这种数据结构相比于链表队列优势：

- 更快：比较void *大小的数据，只需要执行单次CAS指令，而不需要执行2次CAS指令
- 比完全无锁队列简单
- 适用于批量入队/出队操作。因为指针存储在表中，多个对象出队并不会像链表队列那样产生大量的缓存未命中，此外，多个对象批量出队不会比单个对象出队开销大

缺点如下：

- 大小固定
- 它在许多情况下，内存方面的成本比链表列表的成本更高。空环至少包含N个指针。

Ring库的用例包括：

- DPDK应用之间的信息交互
- 内存池中的使用

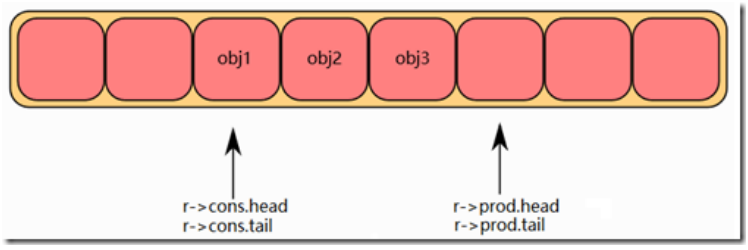
注：

一个Ring被唯一的名字识别，当尝试创建两个名字相同的Ring时，rte_ring_create()函数会在第二次执行时返回NULL。

2.2 Ring实现原理

本节介绍环形缓存的运作方式。ring结构由两对head，tail组成，一对被生产者使用（cons），一对被消费者使用（prod），

在后续介绍中，r->cons.head和r->cons.tail 分别指向消费者的头和尾，r->prod.head和r->prod.tail指向生产者的头和尾。



下文中每种图形代表了环形缓存ring的一个简单状态。局部变量在队列图形的上方表示（如cons_head，prod_head等都是局部变量），ring结构相关变量在队列图形的下方表示（以r->开头）。

2.2.1 单生产者入队

本节介绍当单生产者添加一个对象到ring时发生了什么。在这个例子中，仅只有一个生产者，仅只有生产者的head和tail（r->cons.head、r->cons.tail）索引被修改了，在初始状态， 它们指向相同的位置。

2.2.2.1 入队第一步

使用局部变量保存r->prod.head 和 r->cons.tail，同时prod_next局部变量指向prod_head的下一个元素，若是批量入队就指prod_head的下N个元素。假如ring里没有足够的空间（通过检查cons_tail），入队函

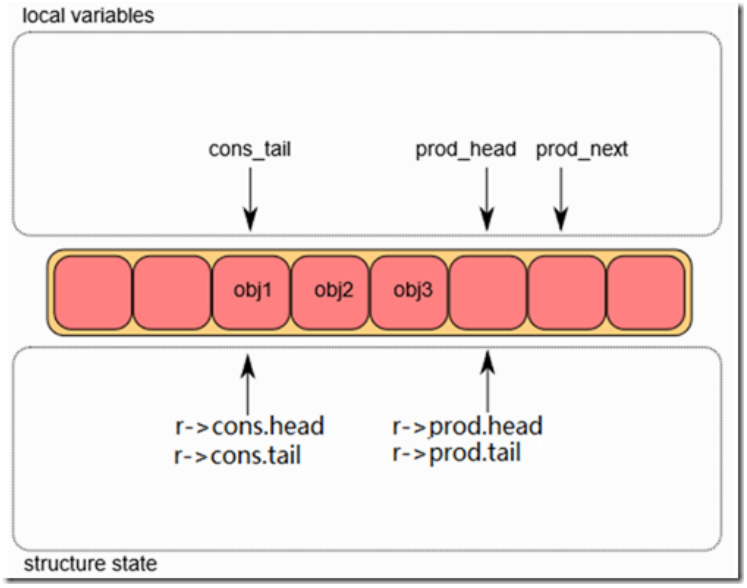
随笔档案

阅读排行榜

评论排行榜

推荐排行榜

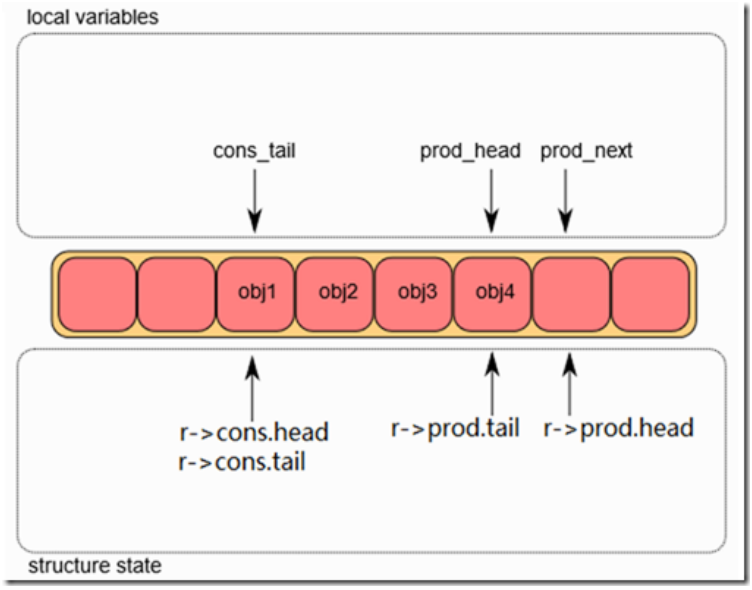
数将返回错误。



2.2.2.2 入队第二步

修改ring结构体里的r->prod.head 索引，将它指向局部变量prod_next指向的位置。

将“新增对象的指针”(下图中的obj4)复制到ring里。



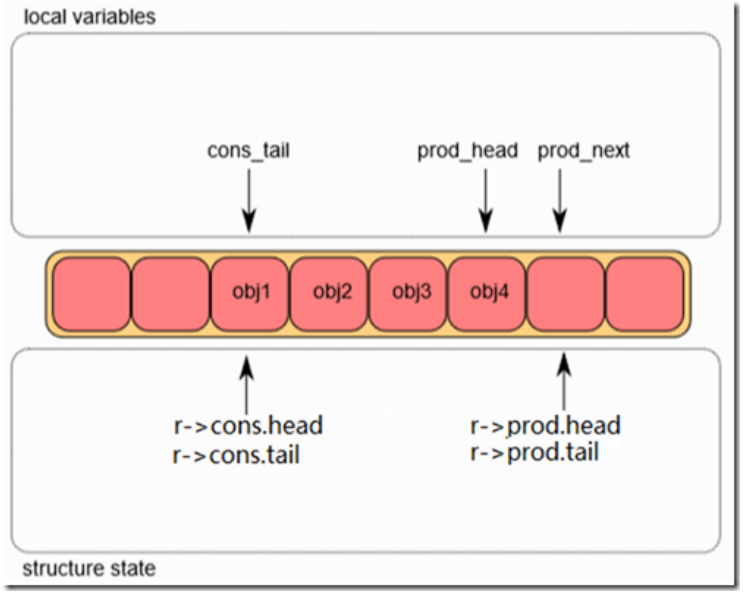
2.2.2.3 入队最后一步

一旦添加对象被复制到ring后，ring结构体里的 r->prod.tail索引将指向 r->prod.head的位置，入队操作完成。

2. Mac中wireshark如何抓取HTTPS流量(1)
3. WAF功能介绍（入门扫盲篇）(1)
4. 如何在没有core文件的情况下用dmesg+addr2line定位段错误(1)

最新评论

1. Re:TCP三次握四次挥手里seq和ack号的【正确】理解
ack ACK既然含义不同，文中应该正常区分大小写，不然是真的混乱。
--少林功夫好
2. Re:WAF功能介绍（入门扫盲篇）
@叶常落 我也是转载，稍作修改。文本开头的链接是原文。...
--一觉醒来写程序
3. Re:DPDK开发环境搭建（学会了步骤适合各版本）
@flamboyante 检查每一步环节是否正确，比如网卡是不是绑定成功了、大页内存是否分配出来。...
--一觉醒来写程序
4. Re:DPDK开发环境搭建（学会了步骤适合各版本）
请问我测试testpmd的时候，一直发送和接受都是0怎么办啊
--flamboyante
5. Re:如何在没有core文件的情况下用dmesg+addr2line定位段错误
@ims- 我用虚拟机+centos7.3下重新试了一下是可以的。 不过我在其他地方用debian9.11试了不行。看来地址还是要根据实际情况来分析，感谢提出问题，已更新博客。...
--一觉醒来写程序



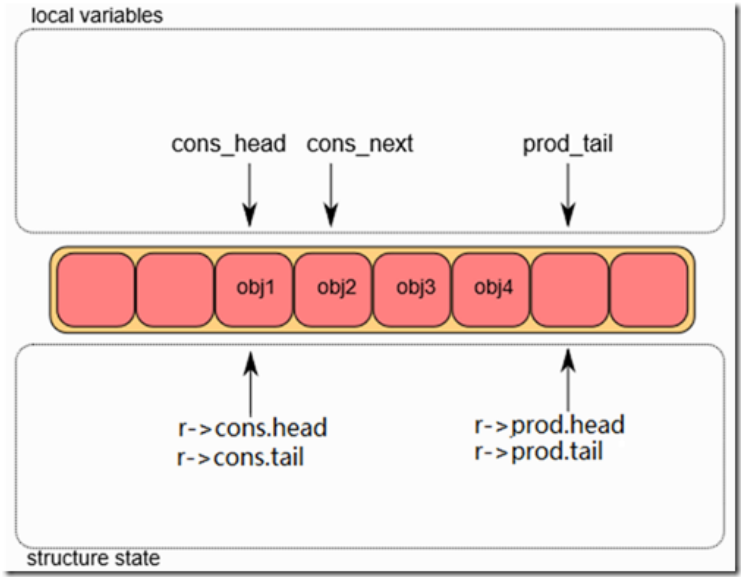
2.2.3 单消费者出队

在这个例子中，仅有一个消费者，仅有消费者的head和tail（r->cons.head 和 r->cons.tail）索引被修改了。

初始状态， r->cons.head 和 r->cons.tai指向相同的位置。

2.2.2.1 出队第一步

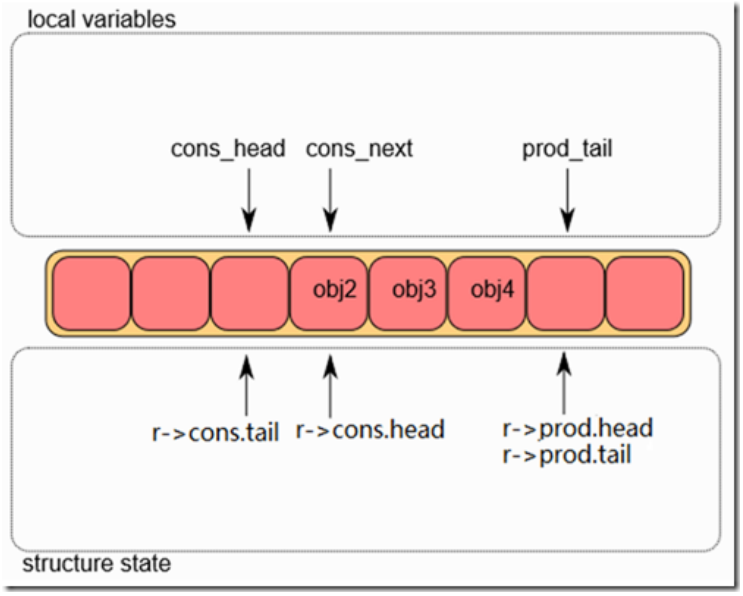
使用局部变量保存r->cons.head 和 r->prod.tail。 cons_next局部变量指向cons_head的下一个元素，若是批量出队就指向cons_head的下N个元素。假如ring里没有足够的空间（通过检查prod_tail），出队函数将返回错误。



2.2.2.2 出队第二步

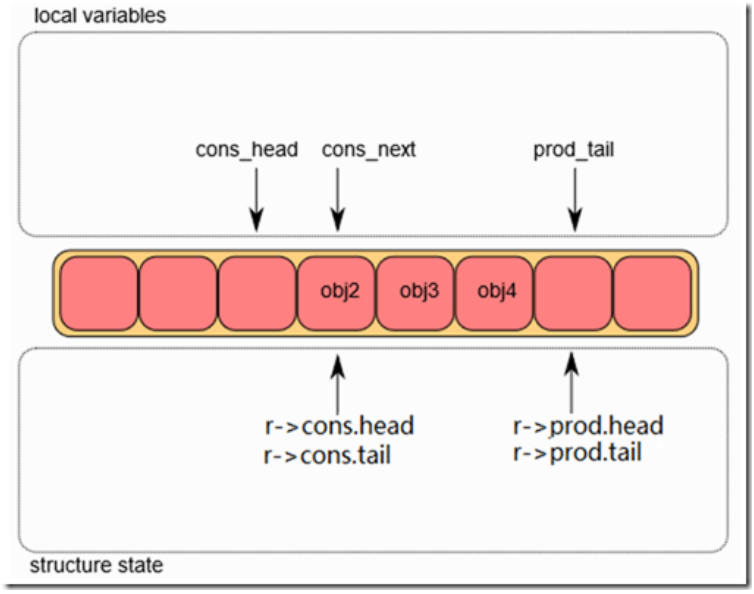
修改ring结构体里的r->cons.head 索引，将它指向局部变量cons_next指向的位置。

将对象的指针(上图ring中的obj1)复制到用户传进来的指针中。



2.2.2.3 出队最后一步

ring结构体中的 ring->cons.tail索引指向和 ring->cons.head，局部变量cons_next相同的位置（obj2的位置）。出队操作完成。



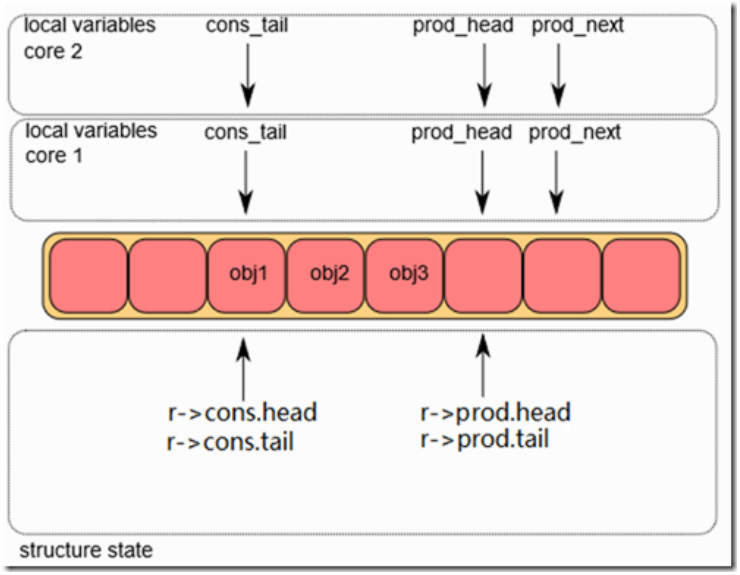
2.2.4 多生产者入队

在这个例子中，仅有生产者的head和tail（r->prod.head 和 r->prod.tail）被修改了。初始状态， 它们指向相同的位置。

2.2.4.1 多生产者入队第一步

在两个生产者core中（这个core可以理解成同时运行的线程或进程），各自的局部变量都保存r->prod.head 和 r->cons.tail。 各自的局部变量prod_next索引指向r->prod.head的下一个元素，如果是批量入队，指向下N个元素。

假如ring里没有足够的空间（检查cons_tail获知），入队函数将返回错误。



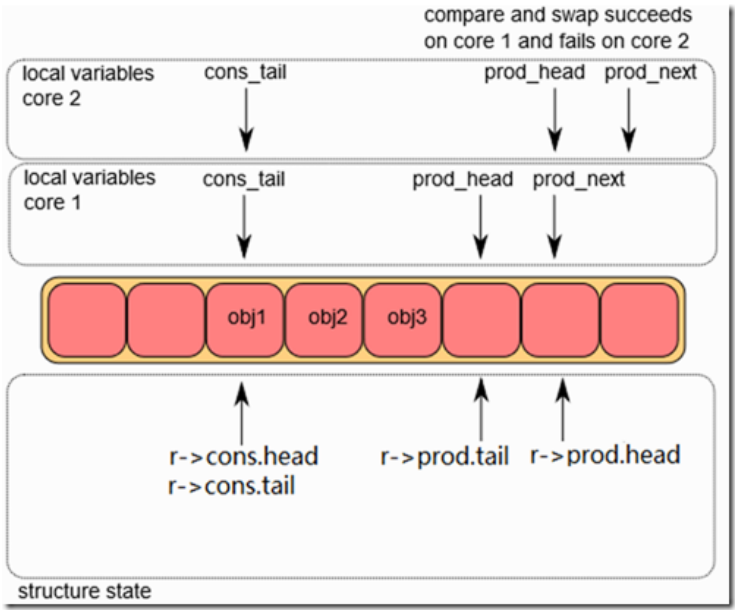
2.2.4.2 多生产者入队第二步

修改ring结构体里的r->prod.head 索引，将它指向局部变量prod_next指向的位置。这个操作是通过使用Compare And Swap (CAS)执行完成的， Compare And Swap (CAS)包含以下原子操作：

- 如果r->prod.head索引和局部变量prod_head索引不相等，CAS操作失败，代码将重新从第一步开始执行。
- 否则，将r->prod.head索引指向局部变量prod_next的位置，CAS操作成功，继续下一步处理。

> 注：涉及到了两个core同时对r->prod.head读取，使用了volatile修饰。同样的prod和cons的2对head和tail都是用了volatile修饰。

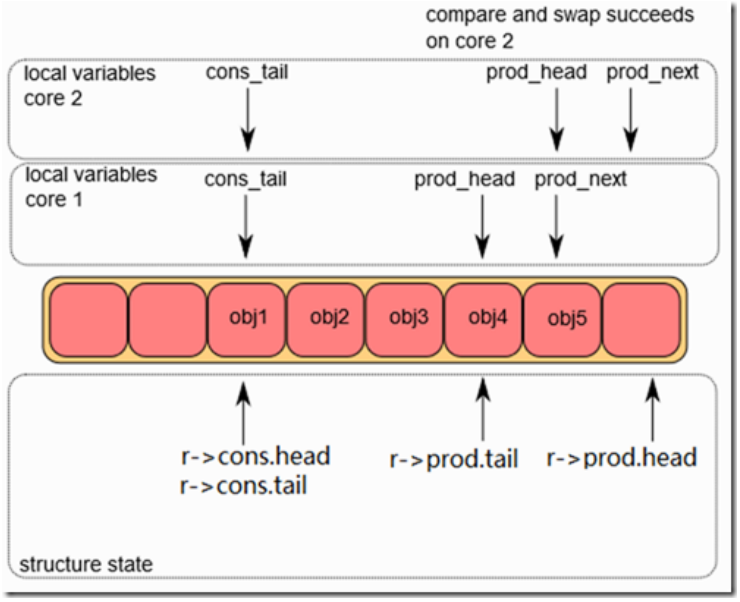
在下图中，生产者core1执行成功，生产者core2重新从第一步开始执行。



2.2.4.3 多生产者入队第三步

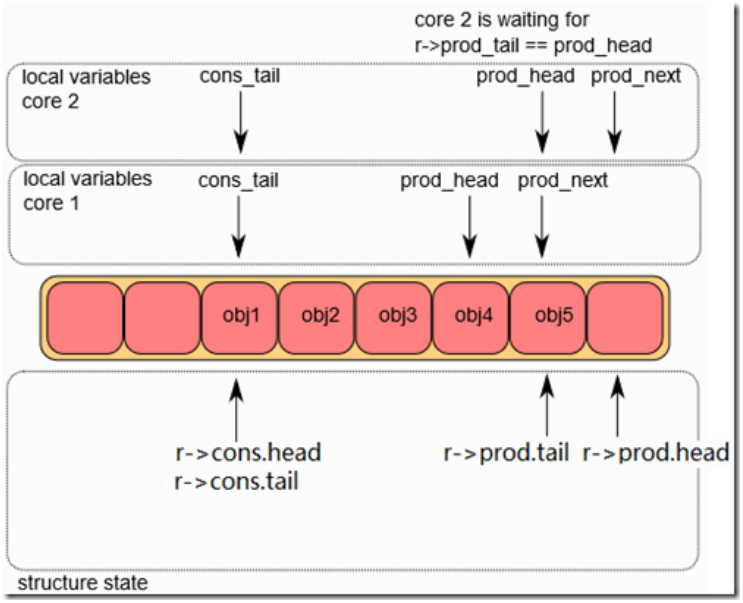
生产者core2中CAS指令重试成功，r->prod.head位置被更新。

生产者core1更新对象obj4到ring中，生产者core2更新对象obj5到ring中。



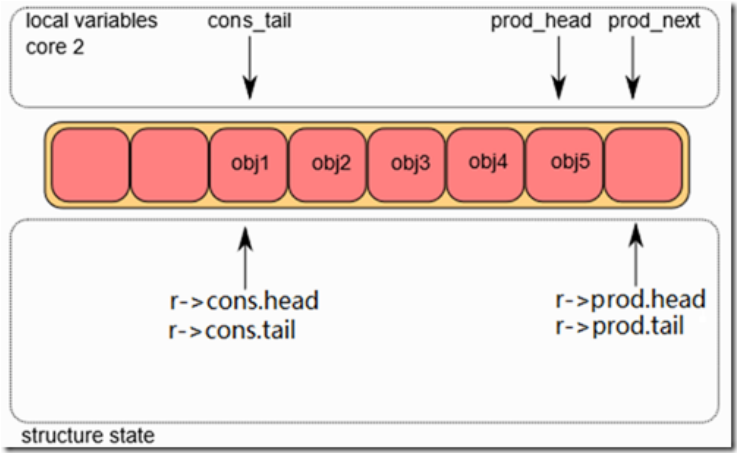
2.2.4.4 多生产者入队第四步

现在每个生产者core都想通过CAS更新 r->prod.tail索引。生产者core代码中，只有r->prod.tail等于自己局部变量prod_head才能被更新，显然从上图中可知，只有生产者core1才能满足，生产者core1完成了入队操作。



2.2.4.5 多生产者入队最后一步

一旦生产者core1更新了r->prod.tail后，生产者core2也可以更新r->prod.tail了。至此，生产者core2也完成了入队操作。



注：

1）从修改r->prod.head和r->prod.tail的步骤来看，存在“重试”的动作（代码里看是通过while循环不断尝试），因此虽然说是无锁，但是在多生产者情况下还是会有竞争。在创建队列时需要传入是否多生产者的标记，这个标记一定要正确，否则影响性能或准确性。

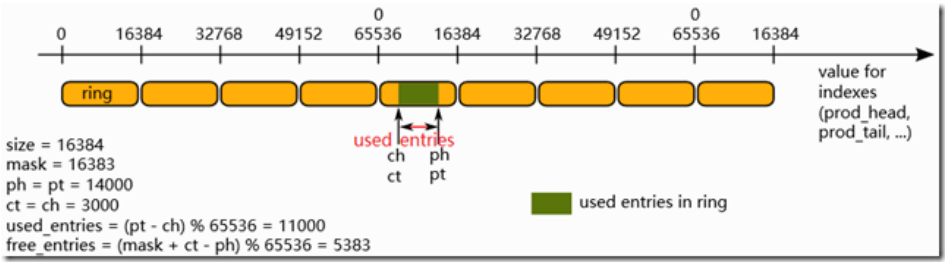
2）多消费者情况类似，参考上文可以推导出，这里就不再重复。

2.2.5 关于32位取模索引

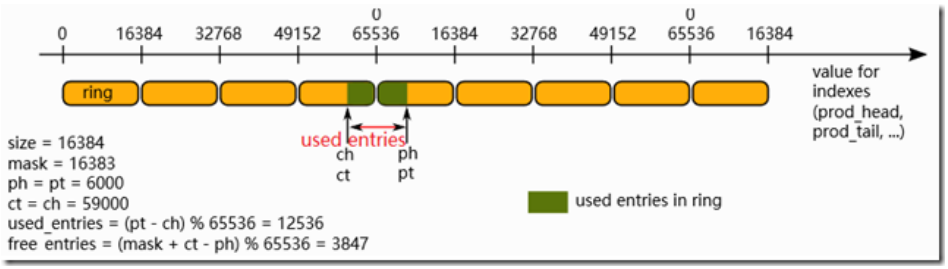
在前面的图例中，prod_head, prod_tail, cons_head 和 cons_tail 都是用箭头表示的。但在实际的代码实现中，他们的值并不是0和ring大小减一之间的数值。索引的大小范围是0—2^32-1，当访问ring中的数据时，真正的索引等于ring中索引值和掩码与之后的值。32 bit取模的意思是如果索引操作（加减）的结果的值超出了32 bit数据的范围，溢出的值忽略，只看省下的位组成的数。

下面的两个例子帮助解释索引在ring中如何使用的，为了简便，例子操作的是16位而不是32位。另外，关键的四个索引也被定义成16位的整数，现实代码实现是用得32位的整数。

例1：ring包含了11000条目



例2：ring包含了12536个条目



为了便于理解，上面的例子中使用模65536的操作。在真实代码实现中，这是冗长低效的，但是当结果溢出时是自动完成的。

代码实现总是将producer 和 consumer保持0—ring大小减1的距离。 这个特性的好处是我们能在两个32位索引值之间做减法，且差值永远在0—ring大小减1范围内：这也是为什么结果溢出不是什么大问题。

在任何时候，已经使用的条目和空闲的条目永远在0—ring大小减1之间。