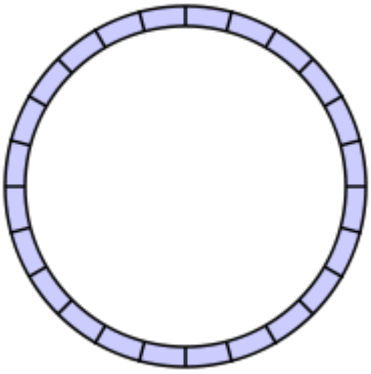
WIKIPEDIA

环形缓冲器

维基百科，自由的百科全书

圆形缓冲区（circular buffer），也称作圆形队列（circular queue），循环缓冲区（cyclic buffer），环形缓冲区（ring buffer），是一种用于表示一个固定尺寸、头尾相连的缓冲区的数据结构，适合缓存数据流。



圆形缓冲区的概念图示。计算机内存是线性地址空间，因此需要采用下述技术来逻辑实现圆形缓冲区

目录

用法

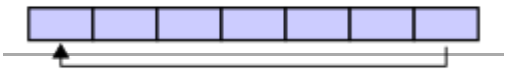
圆形缓冲区的一个有用特性是：当一个数据元素被用掉后，其余数据元素不需要移动其存储位置。相反，一个非圆形缓冲区（例如一个普通的队列）在用掉一个数据元素后，其余数据元素需要向前搬移。换句话说，圆形缓冲区适合实现先进先出缓冲区，而非圆形缓冲区适合后进先出缓冲区。

圆形缓冲区适合于事先明确了缓冲区的最大容量的情形。扩展一个圆形缓冲区的容量，需要搬移其中的数据。因此一个缓冲区如果需要经常调整其容量，用链表实现更为合适。

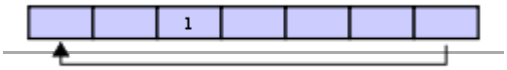
写操作覆盖圆形缓冲区中未被处理的数据在某些情况下是允许的。特别是在多媒体处理时。例如，音频的生产者可以覆盖掉声卡尚未来得及处理的音频数据。

工作过程

一个圆形缓冲区最初为空并有预定的长度。例如，这是一个具有七个元素空间的圆形缓冲区，其中底部的单线与箭头表示“头尾相接”形成一个圆形地址空间：

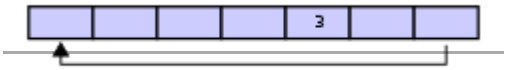


假定1被写入缓冲区中部（对于圆形缓冲区来说，最初的写入位置在哪里是无关紧要的）：



再写入2个元素，分别是2 & 3 — 被追加在1之后：

如果两个元素被处理，那么是缓冲区中最老的两个元素被移除。在本例中，1 & 2被移除，缓冲区中只剩下3:



如果缓冲区中有7个元素，则是满的：

如果缓冲区是满的，又要写入新的数据，一种策略是覆盖掉最老的数据。此例中，2个新数据— A & B — 写入，覆盖了3 & 4:

也可以采取其他策略，禁止覆盖缓冲区的数据，采取返回一个错误码或者抛出异常。

最终，如果从缓冲区中移除2个数据，不是3 & 4 而是 5 & 6 。因为 A & B 已经覆盖了3 & 4：

圆形缓冲区工作机制

由于计算机内存是线性地址空间，因此圆形缓冲区需要特别的设计才可以从逻辑上实现。

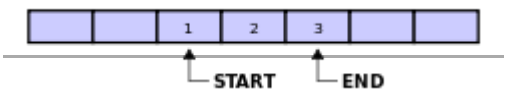
读指针与写指针

一般的，圆形缓冲区需要4个指针：

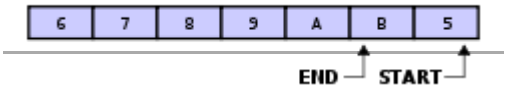
- 在内存中实际开始位置；
- 在内存中实际结束位置，也可以用缓冲区长度代替；
- 存储在缓冲区中的有效数据的开始位置（读指针）；
- 存储在缓冲区中的有效数据的结尾位置（写指针）。

读指针、写指针可以用整型值来表示。

下例为一个未满的缓冲区的读写指针：

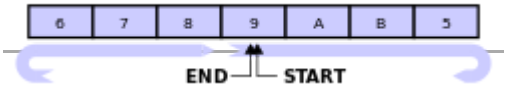


下例为一个满的缓冲区的读写指针：



区分缓冲区满或者空

缓冲区是满、或是空，都有可能出现读指针与写指针指向同一位置：



有多种策略用于检测缓冲区是满、或是空.

总是保持一个存储单元为空

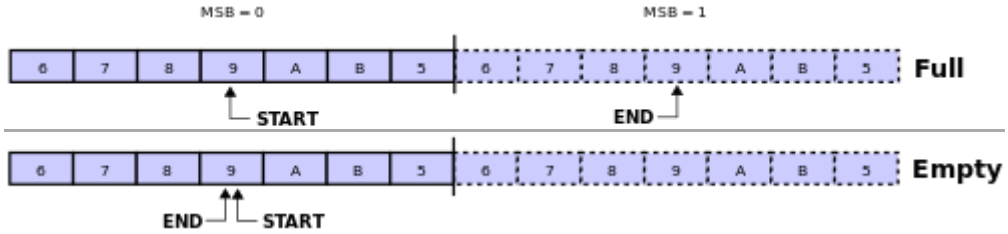
缓冲区中总是有一个存储单元保持未使用状态。缓冲区最多存入$(\mathbf{size}-1)$个数据。如果读写指针指向同一位置，则缓冲区为空。如果写指针位于读指针的相邻后一个位置，则缓冲区为满。这种策略的优点是简单、粗暴；缺点是语义上实际可存数据量与缓冲区容量不一致，测试缓冲区是否满需要做取余数计算。

使用数据计数

这种策略不使用显式的写指针，而是保持着缓冲区内存储的数据的计数。因此测试缓冲区是空是满非常简单；对性能影响可以忽略。缺点是读写操作都需要修改这个存储数据计数，对于多线程访问缓冲区需要并发控制。

镜像指示位

缓冲区的长度如果是n，逻辑地址空间则为0至n-1；那么，规定n至2n-1为镜像逻辑地址空间。本策略规定读写指针的地址空间为0至2n-1，其中低半部分对应于常规的逻辑地址空间，高半部分对应于镜像逻辑地址空间。当指针值大于等于2n时，使其折返（wrapped）到ptr-2n。使用一位表示写指针或读指针是否进入了虚拟的镜像存储区：置位表示进入，不置位表示没进入还在基本存储区。



在读写指针的值相同情况下，如果二者的指示位相同，说明缓冲区为空；如果二者的指示位不同，说明缓冲区为满。这种方法优点是测试缓冲区满/空很简单；不需要做取余数操作；读写线程可以分别设计专用算法策略，能实现精致的并发控制。 缺点是读写指针各需要额外的一位作为指示位。

如果缓冲区长度是2的幂，则本方法可以省略镜像指示位。如果读写指针的值相等，则缓冲区为空；如果读写指针相差n，则缓冲区为满，这可以用条件表达式（写指针 == (读指针 异或 缓冲区长度)）来判断。

```
// This approach adds one bit to end and start pointers
// Circular buffer object
typedef struct {
    int size; // maximum number of elements
    int start; // index of oldest element
    int end; // index at which to write new element
    ElemType *elems; // vector of elements
} CircularBuffer;

void cbInit(CircularBuffer *cb, int size) {
    cb->size = size;
    cb->start = 0;
    cb->end = 0;
    cb->elems = (ElemType *)calloc(cb->size, sizeof(ElemType));
}

void cbPrint(CircularBuffer *cb) {
    printf("size = 0x%x, start = %d, end = %d\n", cb->size, cb->start, cb->end);
}

int cbIsFull(CircularBuffer *cb) {
    return cb->end == (cb->start ^ cb->size); // This inverts the most significant bit of start before comparison
}

int cbIsEmpty(CircularBuffer *cb) {
    return cb->end == cb->start;
}

int cbIncr(CircularBuffer *cb, int p) {
    return (p + 1) & (2 * cb->size - 1); // start and end pointers incrementation is done modulo 2*size
}

void cbWrite(CircularBuffer *cb, ElemType *elem) {
    cb->elems[cb->end & (cb->size - 1)] = *elem;
    if (cbIsFull(cb)) // full, overwrite moves start pointer
        cb->start = cbIncr(cb, cb->start);
    cb->end = cbIncr(cb, cb->end);
}

void cbRead(CircularBuffer *cb, ElemType *elem) {
    *elem = cb->elems[cb->start & (cb->size - 1)];
    cb->start = cbIncr(cb, cb->start);
}
```

读/写 计数

用两个有符号整型变量分别保存写入、读出缓冲区的数据数量。其差值就是缓冲区中尚未被处理的有效数据的数量。这种方法的优点是读线程、写线程互不干扰；缺点是需要额外两个变量。

记录最后的操作

使用一位记录最后一次操作是读还是写。读写指针值相等情况下，如果最后一次操作为写入，那么缓冲区是满的；如果最后一次操作为读出，那么缓冲区是空。 这种策略的缺点是读写操作共享一个标志位，多线程时需要并发控制。

POSIX优化实现

```c
#include <sys/mman.h>
#include <stdlib.h>
#include <unistd.h>

#define report_exceptional_condition() abort ()

struct ring_buffer {
    void *address;
    unsigned long count_bytes;
    unsigned long write_offset_bytes;
    unsigned long read_offset_bytes;
};

// Warning order should be at least 12 for Linux
void ring_buffer_create (struct ring_buffer *buffer, unsigned long order) {
    char path[] = "/dev/shm/ring-buffer-XXXXXX";
    int file_descriptor;
    void *address;
    int status;
    file_descriptor = mkstemp(path);
    if (file_descriptor < 0)
        report_exceptional_condition();
    status = unlink(path);
    if (status)
        report_exceptional_condition();
    buffer->count_bytes = 1UL << order;
    buffer->write_offset_bytes = 0;
    buffer->read_offset_bytes = 0;
    status = ftruncate(file_descriptor, buffer->count_bytes);
    if (status)
        report_exceptional_condition();
    buffer->address = mmap (NULL, buffer->count_bytes << 1, PROT_NONE,
                            MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
    if (buffer->address == MAP_FAILED)
        report_exceptional_condition();
    address =
        mmap(buffer->address, buffer->count_bytes, PROT_READ | PROT_WRITE,
             MAP_FIXED | MAP_SHARED, file_descriptor, 0);
    if (address != buffer->address)
        report_exceptional_condition();
    address = mmap(buffer->address + buffer->count_bytes,
                   buffer->count_bytes, PROT_READ | PROT_WRITE,
                   MAP_FIXED | MAP_SHARED, file_descriptor, 0);
    if (address != buffer->address + buffer->count_bytes)
        report_exceptional_condition();
    status = close(file_descriptor);
    if (status)
        report_exceptional_condition();
}

void ring_buffer_free(struct ring_buffer *buffer) {
    int status;
    status = munmap(buffer->address, buffer->count_bytes << 1);
    if (status)
        report_exceptional_condition ();
}

void *ring_buffer_write_address(struct ring_buffer *buffer) {
    // void pointer arithmetic is a constraint violation.
    return buffer->address + buffer->write_offset_bytes;
}

void ring_buffer_write_advance(struct ring_buffer *buffer, unsigned long count_bytes) {
    buffer->write_offset_bytes += count_bytes;
}

void *ring_buffer_read_address(struct ring_buffer *buffer) {
    return buffer->address + buffer->read_offset_bytes;
}

void ring_buffer_read_advance(struct ring_buffer *buffer, unsigned long count_bytes) {
    buffer->read_offset_bytes += count_bytes;
    if (buffer->read_offset_bytes >= buffer->count_bytes) {
```

```
74          // 如果读指针大于等于缓冲区长度，那些读写指针同时折返回[0, buffer_size]范围内
75          buffer->read_offset_bytes -= buffer->count_bytes;
76          buffer->write_offset_bytes -= buffer->count_bytes;
77      }
78  }
79
80  unsigned long ring_buffer_count_bytes(struct ring_buffer *buffer) {
81      return buffer->write_offset_bytes - buffer->read_offset_bytes;
82  }
83
84  unsigned long ring_buffer_count_free_bytes(struct ring_buffer *buffer) {
85      return buffer->count_bytes - ring_buffer_count_bytes (buffer);
86  }
87
88  void ring_buffer_clear(struct ring_buffer *buffer) {
89      buffer->write_offset_bytes = 0;
90      buffer->read_offset_bytes = 0;
91  }
92
93  /*  Note, that initial anonymous mmap() can be avoided - after initial mmap() for descriptor fd,
94      you can try mmap() with hinted address as (buffer->address + buffer->count_bytes) and if it fails -
95      another one with hinted address as (buffer->address - buffer->count_bytes).
96      Make sure MAP_FIXED is not used in such case, as under certain situations it could end with segfault.
97      The advantage of such approach is, that it avoids requirement to map twice the amount you need initially
98      (especially useful e.g. if you want to use hugetlbfs and the allowed amount is limited)
99      and in context of gcc/glibc - you can avoid certain feature macros
100     (MAP_ANONYMOUS usually requires one of: _BSD_SOURCE, _SVID_SOURCE or _GNU_SOURCE). */
```

Linux内核的kfifo

在Linux内核文件kfifo.h和kfifo.c中，定义了一个先进先出圆形缓冲区实现。如果只有一个读线程、一个写线程，二者没有共享的被修改的控制变量，那么可以证明这种情况下不需要并发控制。kfifo就满足上述条件。kfifo要求缓冲区长度必须为2的幂。读、写指针分别是无符号整型变量。把读写指针变换为缓冲区内的索引值，仅需要“按位与”操作：（指针值 按位与 （缓冲区长度-1））。这避免了计算代价高昂的“求余”操作。且下述关系总是成立：

读指针 + 缓冲区存储的数据长度 == 写指针

即使在写指针达到了无符号整型的上界，上溢出后写指针的值小于读指针的值，上述关系仍然保持成立（这是因为无符号整型加法的性质）。 kfifo的写操作，首先计算缓冲区中当前可写入存储空间的数据长度：

len = min{待写入数据长度, 缓冲区长度 - （写指针 - 读指针）}

然后，分两段写入数据。第一段是从写指针开始向缓冲区末尾方向；第二段是从缓冲区起始处写入余下的可写入数据，这部分可能数据长度为0即并无实际数据写入。

外部链接

- CircularBuffer at the Portland Pattern Repository
- Boost: Templated Circular Buffer Container (http://www.boost.org/doc/libs/1_39_0/libs/circular_buffer/doc/circular_buffer.html)
- http://www.dspguide.com/ch28/2.htm

取自 “https://zh.wikipedia.org/w/index.php?title=環形緩衝區&oldid=53969738”