无锁的线程池 和内存池 还有无锁的队列 的设计思路是什么呢?

查看全部 24 个回答

什么是无锁

本文承接着上一篇文章：玉米：多线程（一）：C++11 atomic和内存序 ，继续对原子操作进行研究。

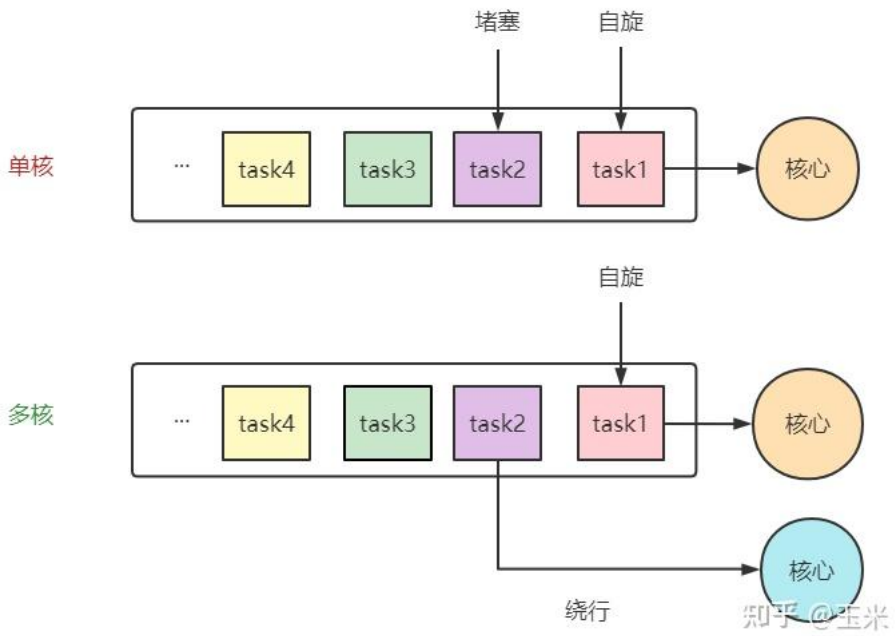
上一篇文章的最后，我说无锁编程o其实终究还是有锁，只不过是把互斥锁换成了自旋锁，将lock-free编程翻译成无锁是个历史遗留问题。

自旋锁基于CAS实现，相较于互斥锁mutexo的优点就在于不会频繁地切换上下文，导致缓存失效，以及省去线程休眠唤醒等等调度的开销。

但是自旋锁也并非完美无缺，自旋锁的意义在于把当前的线程变成了一个不可抢占式的任务，那么当自旋锁锁住的业务工作时间太久，其它线程和操作系统也拿它没办法，就会造成其它线程的无限等待。

所以我们大致可以得出这样一个结论：

- 如果处理器性能不是很好，且只有单核，那么使用互斥锁其实是要优于自旋锁的，因为哪怕当前的线程执行太久也可以被休眠切换，使得多任务并发执行。
- 如果处理器性能比较好，且核心数很多，那么使用自旋锁应该是优于互斥锁的，虽然当前线程使用自旋锁可能会持久执行一段时间，但是它也只能霸占一个核心，其它线程可以到其它核心上执行完，也就是多任务并行执行，不会陷入无限等待。



事实上，使用无锁编程在如今不应该是值得太忧虑的事情，毕竟哪家大厂会拿一台破处理器当服务器呢？肯定是有钱就上更好的。

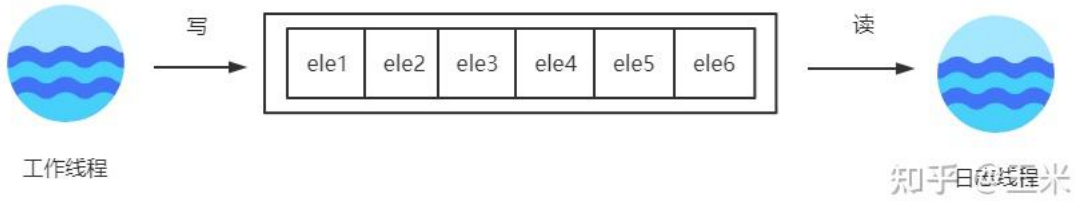
所以学习一下无锁编程，将其应用在项目中我觉得还是挺有意义的。

单写单读无锁队列

将无锁应用在队列中是一个很恰如其分的尝试，因为队列的操作比较简单，最核心的只有push和pop操作，很容易改成无锁编程。

在实际的应用中，队列又经常用来给两个线程之间传递消息，一个线程往里读，一个线程往外写。

举个最简单的例子，在服务器中，我们有个工作线程，用来执行客户端传来的业务，还有一个日志线程q负责给数据库写日志，当工作线程执行完业务，就可以往这个队列里写这条业务的日志，而日志线程源源不断地从这个队列里面读日志出来，并写入到数据库。



那么上面例子中，临界资源的竞态会发生在什么地方？

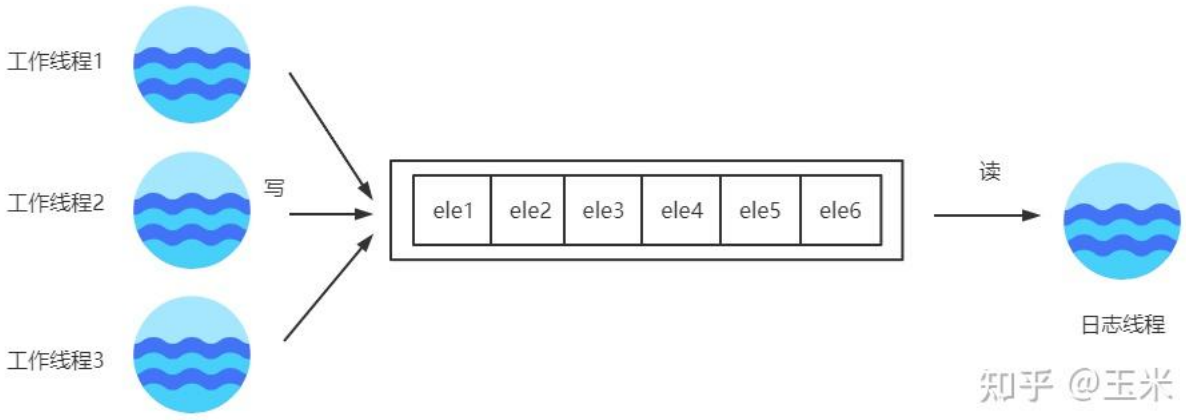
答案是读写的同时发生，当工作线程push到一半，日志线程就开始pop，这样肯定会出问题。

如果是常规做法，我们就会让工作线程push之前lock mutex，然后push后再unlock mutex，日志线程则同理。

如果是无锁编程，我们让两条线程各自push和pop即可，因为两个操作被自旋锁保护，不会被另一个线程干扰。

多写多读无锁队列

上一个例子有个比较大的问题，虽然我们解决了读写的冲突，但是实际服务器的工作线程不可能只有一条。我们有多个工作线程，每个线程同时往队列里写数据，也就是多写冲突，该怎么办？



常规做法当然还是很好想，我们让每个工作线程push之前lock mutexq，然后push后再unlock mutex。

如果是无锁编程，其实也并不难，我们让多个线程同时利用循环的CAS指令（其实就是自旋锁的源码）尝试获得队列的back（尾巴），也就是插入位置，只有拿到真正的尾巴那个线程才能破开循环，往里面push，其它没拿到尾巴的继续陷入循环再次尝试。

可以简单写个伪码

```
def write:
    q = new_node()
```

```
do{
    p = back()
}while(not CAS(p→next,NULL,q))
CAS(back(),p,q)
```

我的选择和手撕代码

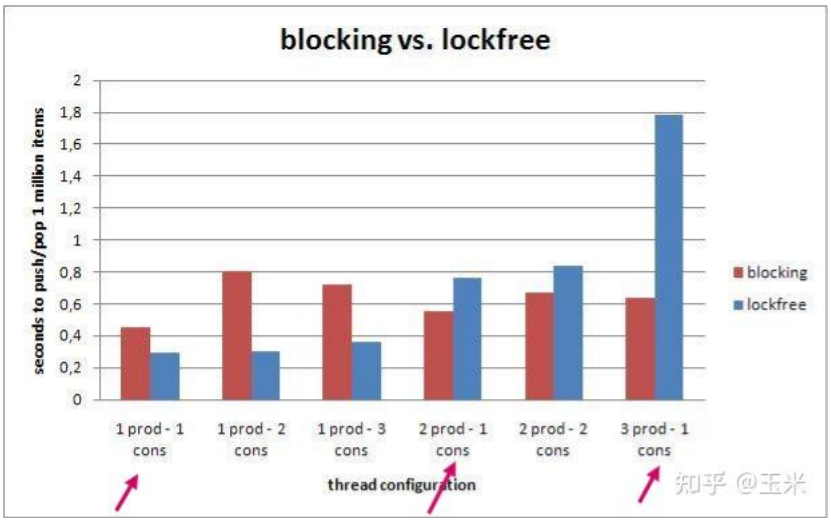
本次手撕一个单写单读的队列，基于ZeroMQ的无锁队列实现一个改版，简单易读。

选择这个版本出于几个方面的考虑。

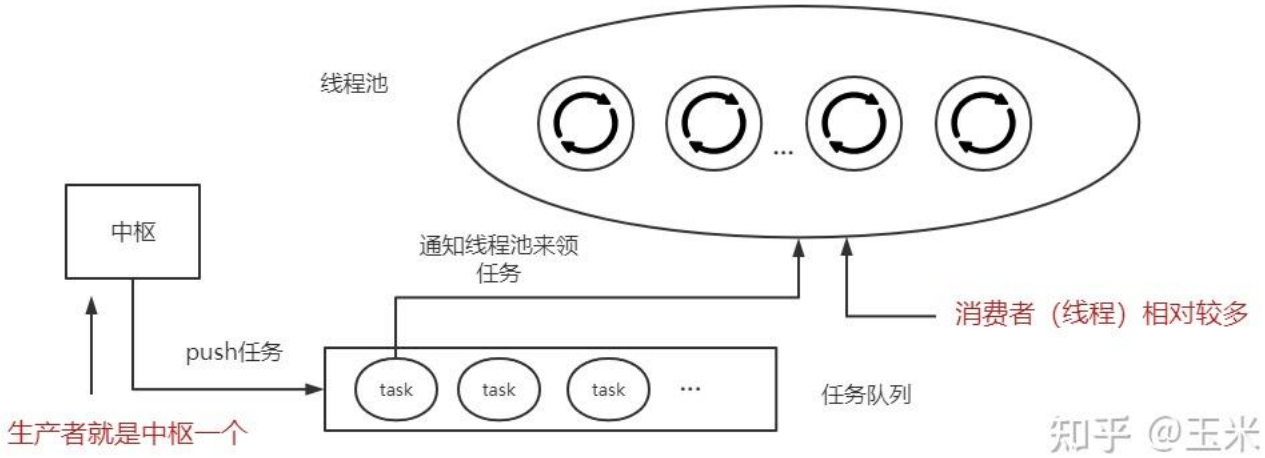
- 为什么不多写多读呢，多写多读的无锁队列不应该更好更强大吗？

其实并非如此，我个人觉得，世界上没有最好的编程语言，只有最适合某个业务场景的语言，做数据库的不用c语言写，非要用python；写前端的不用javascript，非要用java，这不是一种敢于挑战现实的勇敢或者无畏，而是没有深刻理解每种语言的特点和优劣造成的笑话。

多写多读的无锁队列也是如此，在这篇文章中 sniperHW：基于数组的无锁队列(译) 提到了这个问题，当生产者（工作线程）的数量多于两个以上，对lock-free的队列读写（100W次）所消耗的时间（注意纵坐标的指标是时间），反而远远高于一个正常使用mutex的队列。



所以无锁队列并非完美，我们需要找到最适合其使用的地方，那便是生产者少，但是消费者多的应用场景，比如我之前实现的线程池：玉米：c++手撕代码（七）池式编程：线程池 其中的任务队列，就很适合用lock-free队列来存。



- 那么是不是和日志线程的通信就不该用lock-free，老老实实用mutex?

如果你的服务器是将线程作为业务执行单元，那么确实如此。

但是我写这个队列的目的，是为了给我设计的服务器项目中使用的，我设计的服务器将协程作为业务执行单元，协程的特点是在一个线程上串行运行，所以在单核上的任何时刻，都只有一个协程在运行，并不会造成多写冲突。那么就可以用这个单写单读的lock-free队列和日志线程进行通信了。

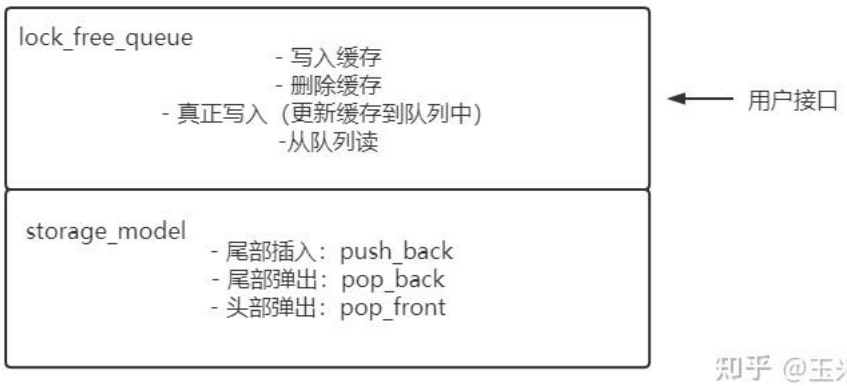
不了解协程也没关系，有关于协程的问题，后面我也会写一些文章详细解说，比如怎么用c/c++手撕一个协程库（画饼= v =）

- 为什么选择ZeroMQ的这个版本

无锁队列的概念很早就提出了，所以实现版本也很多，一开始准备改写的是一个多写多读的Ringbuffer的版本，后来觉得ZeroMQ的缓冲写机制可能更适合我的项目，所以还是选择了这个版本。

然而ZeroMQ这个版本的源码依旧是变量名起的很抽象，看得我脑壳痛，所以还是自己写一版，略微改了底层的结构，更容易理解和阅读。

整体结构概念图



我们可以把整体结构看做两层。

lock_free_queue作为最上层的适配器，提供给用户api主要就是读和写，值得一提的是这个缓冲写机制。主要思路就是在队列的尾部有一个缓存结构，每次写入的时候，都可以选择是写入到缓存里还是真正写入，因为可以写

到缓存里，所以也有反悔机制，可以删掉缓存里的值。

storage_model作为下层的一个存储结构，可以看成是一个类似deque的STL容器，因为上层队列要实现入队，出队，还有反悔机制，所以下层至少要提供push_back, pop_back, pop_front三个接口。

这两层就类似c++中queue和deque之间的关系。queue在deque的基础上，这种方法在实际应用中也很常见，属于设计模式中也比较常见的适配器模式。

注：上面概念图中的api不是实际的api，只是提供一个大概结构，真正要实现的话，api会更多一些。

提供原子操作的指针

首先我们需要定义一个atomic的指针类型，该指针类型可以提供原子cas和原子xchg的操作。不论是lock_free_queue类还是storage_model类都会用到这个类型，具体用法后面会说。

原子操作atomic_cas的原语是如果ptr==cmp，就给其赋新值new_ptr，并返回旧值ptr，如果ptr≠cmp，则什么也不做，返回值依然是原来的ptr。

原子操作atomic_set的原语则更简单，就是将ptr设置成新的值new_ptr， 并返回旧值ptr即可。

因为我的处理器是x86的，所以就用汇编实现了这两个原子操作，汇编的含义在上一篇文章 玉米：多线程（一）：C++11 atomic和内存序 已经讲过，就不再赘述， 其实如果不想用这个重复造的轮子，也可以直接使用c++11的std::atomic类，效果相同。

```
//-实现一个提供原子操作的指针类型
template<typename T>
class atomic_ptr{
private:
    volatile T* ptr = nullptr;
public:
    //-cas函数,比较ptr和cmp,如果相同,就赋予ptr新值new_ptr，并返回旧值。
    T* atomic_cas(T* cmp, T* new_ptr){
        int *old_ptr;
        __asm__ volatile ("lock; cmpxchg %2, %3"
        : "=a" (old_ptr), "=m" (ptr)
        : "r" (new_ptr), "m" (ptr), "0" (cmp)
        : "cc", "memory");
        return old_ptr;
    }
    //-atomic_set,设置ptr为为新值。返回ptr旧值
    T* atomic_set(T* new_ptr){
        T* old;
        __asm__ volatile ("lock; xchg %0, %2"
                          : "=r" (old), "=m" (ptr)
                          : "m" (ptr), "0"(new_ptr));
        return old;
    }
    //-非原子set
    void set(T* new_ptr){
        ptr = new_ptr;
    }
    //-禁止移动和拷贝
    atomic_ptr() = default;
    atomic_ptr(const atomic_ptr&) = delete;
    const atomic_ptr& operator=(const atomic_ptr&) = delete;
    atomic_ptr(atomic_ptr &&) = delete;
```

```
    const atomic_ptr & operator = (atomic_ptr && ) = delete;
};
```

底层数据结构实现

整体的代码结构：

```
template<typename T, int buffer_length>
class storage_model{
private:
    //-私有内部类
    struct storage_node{
        T buffer[buffer_length];
        storage_node * prev;
        storage_node * next;
    };
    //-第一个元素所在的节点（链表头结点，即便整个队列没有元素该节点也会存在）
    storage_node *begin_node;
    //-当前队列中第一个元素所在的begin_node中的buffer的下标
    int begin_index;

    //-最后一个元素所在的节点（并不一定是当前链表的尾结点）
    storage_node *last_node;
    //-当前队列中最后一个元素所在的last_node中的buffer的下标
    int last_index;

    //-链表尾结点，可能只是一个malloc后无元素的空节点
    storage_node *end_node;
    //-最后一个malloc的元素在end_node中的buffer下标
    int end_index;

    //-假设队列缩减容量，需要释放空的node，会以这个原子指针来存一个最新free掉的node，
    //-预留给下次malloc的时候备用
    atomic_ptr<storage_node> new_free_node;

public:
    //-构造函数
    storage_model();

    //-尾部插入元素
    void push_back(const T& ele);

    //-给链表last_index后追加(预留)一个T元素的空间
    void malloc_back();

    //-尾部弹出元素（实际上改last_node和last_index的指向就行）
    void pop_back();

    //-回收预留的T元素大小的空间
    void free_back();

    //-头部弹出元素（由于不用实现头部插入，所以永远不需要给头部预留T空间，pop头部原理和free_back类似）
    //-(实际上改begin_node和begin_index的指向就行)
    void pop_front();

    //-获取第一个元素引用
    T& front();

    //-获取最后一个元素的引用
    T& back();

    //-返回预留的插入位置
    T& next();

    //-析构函数
```

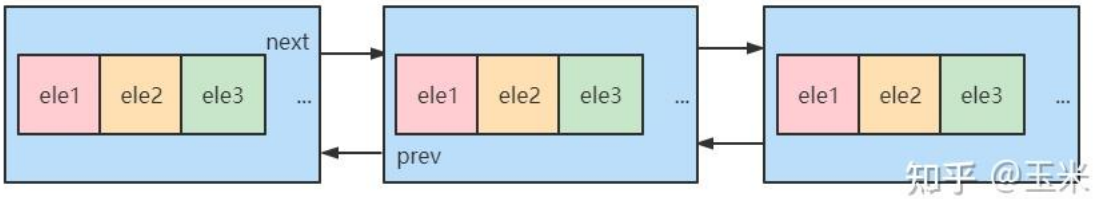
```
    ~storage_model();

    //-禁止拷贝和移动
    storage_model(const storage_model&) = delete;
    const storage_model & operator = (const storage_model &) = delete;
    storage_model(storage_model &&) = delete;
    const storage_model & operator = (storage_model && ) = delete;
};
```

我们先抛开代码，看下整个storage_model的数据结构图。

底层的数据结构也分为两层。

从第一层看，是一个双向链表 。

从第二层看，每个链表节点中都有一份数组。



所以对于链表的节点，我们如此定义。

```
    //-私有内部类
    struct storage_node{
        T buffer[buffer_length];
        storage_node * prev;
        storage_node * next;
    };
```

对于整个链表容器，有如下数据成员

- begin_node：第一个元素所在的节点（链表头结点，即便整个队列没有元素该节点也会存在
- begin_index：当前队列中第一个元素所在的begin_node中的buffer的下标
- last_node：最后一个元素所在的节点（并不一定是当前链表的尾结点）
- last_index：当前队列中最后一个元素所在的last_node中的buffer的下标
- end_node：链表尾结点，可能只是一个malloc后无元素的空节点
- end_index：最后一个malloc的元素在end_node中的buffer下标
- new_free_node：该变量的类型为刚刚定义的原子指针，假设队列缩减容量，需要释放空的node，会以这个原子指针来存一个最新free掉的node，预留给下次malloc的时候备用

begin大家应该都好理解，但是last和end这两个的区别直接看确实不容易理解。

千万注意这里的 last 和 end 并不是提供缓存和反悔机制用的，那是lock_free_queue干的事，这里的 last 和 end提供的是一个预存机制。

什么是预存？为什么要预存？

我们可以想象一个场景，上大学暑假放假回家，你是会提前买好票，还是等放假当天才开始买票？

相信如果是住的远的同学大多都会选择前者，当我们提前买好了票，意味着放假当天可以直接回家。而如果放假当天才开始买票，可能会面临着看票，选票，支付等一系列操作，甚至可能无票，总之就是可能需要等待一定时间才能回家。

而对于想要往storage_model里存数据的场景，也是一样的，storage_model表层基于链表这样一个结构，意味着它并不是一个blocking-queue（有界队列），可以无限拓展，而无限拓展意味着必须需要动态分配新的空间，

如果我们每次向队列中插入元素，都需要malloc一个node，那么这个队列的效率必然会很低。

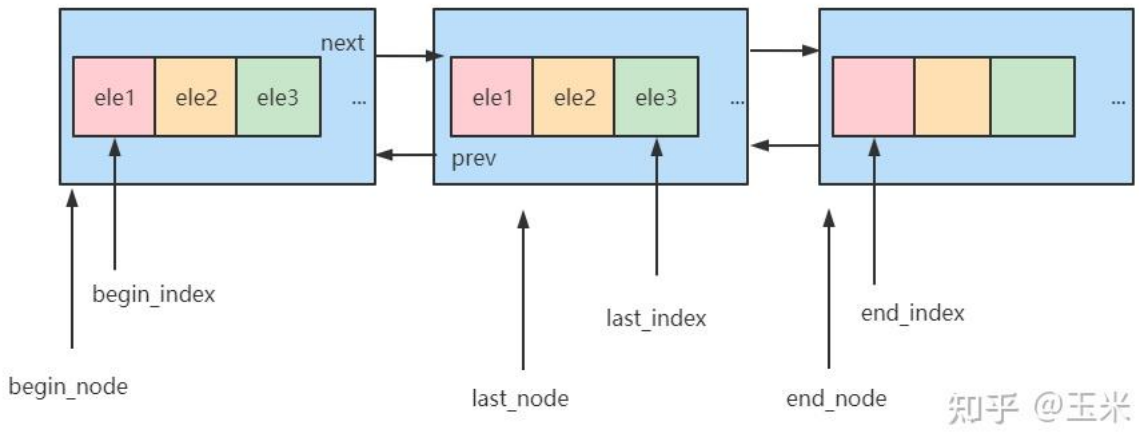
因此storage_model设置了两个机制来解决这个问题。

- 第一便是，每个node节点都有一个数组，作为malloc操作的缓冲，只有当该节点数组填完后，才会malloc下一个节点。
- 第二便是预存了，当我们每次向队列插入一个元素后，就会往后预存下一个元素插入的位置，如果当前插入的元素正好填满了node中的数组，那么预存检查到这种情况，就会直接malloc下一个node。

所以 last_node 和 last_index 两者可以唯一标识当前插入的最后一个元素位置，称为back。

end_node 和 end_index 两者可以唯一标识当前预存元素所在的位置，称为end。

在空间上两个位置是不一定连续的，但是在整个链表的逻辑顺序上，预存元素end的位置永远在最后一个元素位置back后面。



new_free_node的意义则在于：假设当前情况是上图的样子，这个时候队列调用pop_back操作，此时back前移，而end和back在逻辑顺序上是紧挨着的，意味着end也会前移，并且从一个node的开头，前移到上一个node的结尾，这个时候，就会产生一个尾部的空node。

从道理上，我们应该free掉这个空的node，但是考虑到malloc一次node的代价很大，我们为什么不能模仿c语言中malloc/free的机制，（每次free并不会真的调用brk调整内存边界，而是将free掉的内存用链表组织起来，留给下次malloc的时候备用），将该节点先暂存起来，等队列又调用push_back的时候用。

因此，new_free_node的指针指向的就是一个最新被free掉的空node，至于为什么，我们只存一个最新的，而不是存多个，再用链表组织起来，这还是权衡了空间和时间的优解，因为没有必要浪费太多这个额外空间，本身链表节点已经有数组在其中作为缓冲，不会存在大量malloc和free的情况，只需要存一个备用即可。

至此，该storage_model的结构已经很清晰了，实现剩下几个api也就不难了。

注意一下，为了保证入队出队的效率，所有的api我都写在了类中，因此隐式内联。

构造函数

```
    storage_model():
    begin_node((storage_node*)calloc(1,sizeof(storage_node))),//-初始就有一个node
    begin_index(0),
    last_node(nullptr),//-lastnode是最后一个元素所在的node，没有元素，所以为空
    last_index(0),
    end_node(begin_node),//-链表尾即链表头
    end_index(0)
    {};
```

不论队列中有没有元素，总是会有begin_node作为一个临时虚空节点（dummy）的存在。

尾部插入元素：push_back，尾部预存元素：malloc_back

```
    //-尾部插入元素
    void push_back(const T& ele){
        //-将新元素插入到尾部预留的T元素大小的空间
        last_node = end_node;
        last_index = end_index;
        last_node→buffer[last_index] = ele;
        //-给下一个插入的元素预留空间
        malloc_back();
    }

    //-给链表last_index后追加(预留)一个T元素的空间
    void malloc_back(){
        if(++end_index < buffer_length){
            return;
        }
        //-需要新的node
        //-返回free node的元素，看看有没有空闲的node
        storage_node * temp = new_free_node.atomic_set(nullptr);
        if(!temp){//-如果没有free node，只能重新分配
            temp = (storage_node*)calloc(1,sizeof(storage_node));
            if(!temp){
                perror("calloc");
                exit(EXIT_FAILURE);
            }
        }
        end_index = 0;
        end_node→next = temp;
        temp →prev = end_node;
        temp →next = nullptr;
        end_node = temp;
    }
```

对于插入元素，因为end已经预存好了插入位置，直接更新last为end即可，然后插入完后都显示地预存下一个元素。

对于预存元素，每次看是否需要malloc新节点，如果要，还要判断有没有之前备用的free节点。

尾部弹出元素：pop_back，回收预存元素：free_back

```
    //-尾部弹出元素（实际上改last_node和last_index的指向就行）
    void pop_back(){
        if(--last_index == -1){
            //-如果弹出该元素后，node的buffer清空
            last_index = buffer_length - 1;
            last_node = last_node→prev;
        }
        //-回收预留的T元素大小的空间
        free_back();
    }

    //-回收预留的T元素大小的空间
    void free_back(){
        if(--end_index == -1){//-如果删除了预留的空间，end_node的buffer清空
            end_index = buffer_length -1;
            end_node = end_node→prev;
            storage_node * old_free = new_free_node.atomic_set(end_node→next);//-将需要清空的节点更新到new_
            if(old_free){//-真正free掉旧的
                free(old_free);
            }
            end_node →next = nullptr;
        }
    }
```

和之前的 push_back 和 malloc_back是对称的操作。

头部弹出元素：pop_front

```
//-头部弹出元素（由于不用实现头部插入，所以永远不需要给头部预留T空间，pop头部原理和free_back类似）
    //-(实际上改begin_node和begin_index的指向就行)
    void pop_front(){
        if(++begin_index == buffer_length){
            begin_index = 0;
            begin_node = begin_node→next;
            storage_node * old_free = new_free_node.atomic_set(begin_node→prev);
            if(old_free){
                free(old_free);
            }
            begin_node → prev = nullptr;
        }
    }
```

因为头部没有预存空间，所以只需要判断是否要free掉空节点即可。

front，back，end分别代表了三个元素（首元素，最后一个元素，预存元素）的位置

```
//-获取第一个元素引用
    T& front(){
        return begin_node→buffer[begin_index];
    }

    //-获取最后一个元素的引用
    T& back(){
        return last_node→buffer[last_index];
    }

    //-返回预留的插入位置
    T& end() {
        return end_node → buffer[end_index];
    }
```

析构函数

```
//-析构函数
    ~storage_model(){
        //-顺着node链表free
        while(begin_node≠end_node){
            last_node = begin_node;//-析构的时候last_node就没用了，拿来当个临时指针用而已
            begin_node = begin_node→next;
            free(last_node);
        }
        free(end_node);
        //-如果有free_node也删除
        last_node = new_free_node.atomic_set(nullptr);
        if(last_node){
            free(last_node);
        }
    };
```

上层适配器实现

按照一开始的概念图，我们现在要利用 lock_free_queue 复合storage_model结构，重新封装接口。

整体代码框架：

```
//-最上层适配器，无锁队列
template<typename T, int buffer_length>
class lock_free_queue{
private:
    //-复合之前定义的存储结构,表层是个双向链表，所以叫它list
    storage_model<T,buffer_length> sm_list;
    T* producer_end = nullptr;//-对于生产者可见的队列中最后一个元素
    T* consumer_end = nullptr;//-对于消费者可见的队列中的最后一个元素
```

```
    T* accumulate_head = nullptr;//-指向累计缓存（未更新）的第一个元素，如果该指针更新，意味着缓存的元素全部更新到队列
    atomic_ptr<T> sign;//-一个标志位，主要用来标志消费者是否读完队列所有元素，

public:
    //-构造
    lock_free_queue();

    //-向队列写数据(支持缓冲追加),如果complete==false则只会累计在缓存里，不会真的更新队列
    //-若complete == true则立刻把所有累计的一起更新
    void write(const T& ele,bool complete);

    //-删除累计缓存的最后一个数据，循环调用此方法可以清空所有数据
    bool clear_last_push_buffer();

    //-write完并complete为true，可以更新read_end,而返回值代表消费者是否休眠。
    bool updateReadEnd();


    //-判断当前是否可读
    bool checkRead();

    //-每次读数据，都必须利用自旋锁预读，成功的线程才会真正读取
    bool read();
};
```

很容易发现，在数据成员上，lock_free_queue仅仅是在storage_model的基础上添加了四个指针，

- producer_end：对于生产者可见的队列中最后一个元素
- consumer_end：对于消费者可见的队列中的最后一个元素
- accumulate_head：指向累计缓存的第一个元素，如果该指针更新，意味着缓存的元素全部更新到队列里
- sign： 一个标志位，主要用来标志消费者是否读完队列所有元素。

同时，别忘了还有storage_model中的几个（引用）指针。

- front()：指向首元素
- end()：指向预存元素

所以我们需要联合这六个指针的意义才能弄明白整个结构。

但是直接空口说会很难以理解，所以我准备先说lock_free_queue中几个提供给用户的api的实现，并在过程中阐述这些指针的意义，也许会更合适。

接口1：构造函数

```
    //-构造
    lock_free_queue(){
        //-先预留一个插入位置
        sm_list.malloc_back();
        accumulate_head = &sm_list.end();
        sign.set(nullptr);
        producer_end = accumulate_head;
        consumer_end = accumulate_head;
    }
```

严格意义上，这并不是接口，但是需要提醒一下，虽然我们在底层的storage_model中，每次插入元素后都会预存下一位，但是当一开始就没有元素的时候，我们就需要自己显示地malloc_back一个位置，等待后续插入。

至于后面的一堆指针初始化，先不用管，后面会讲；

接口2：写数据（写入缓存、写入并刷新缓存）

```
    //-向队列写数据(支持缓冲追加),如果complete==false则只会累计在缓存里，不会真的更新队列
    //-若complete == true则立刻把所有累计的一起更新
```

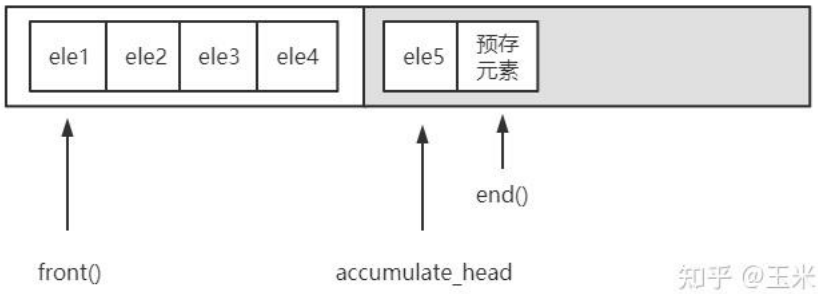
```
    void write(const T& ele,bool complete){
        sm_list.push_back(ele);
        if(complete){
            accumulate_head = &sm_list.end();
        }
    }
```

我们利用complete这个参数，将写入缓存、写入并刷新缓存两个操作融合在一个api中，当我们设置complete为false，意味着写入的数据会累积在缓存中，当我们设置complete为true的时候，再写入，此时会将新元素连同之前累积的元素一起真正更新到队列中。

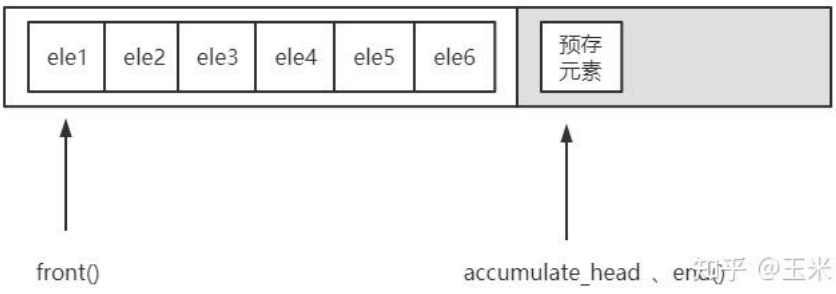
巧妙的是，lock_free_queue只添加了accumulate_head这一个指针就实现了缓存的功能。

accumulate_head 永远指向累计缓存的第一个元素，如果该指针更新，意味着缓存的元素全部更新到队列里。

举个例子，当我们设置complete为false的时候插入元素5



当我们再设置complete为true的时候插入元素6



所以我们只需要靠 accumulate_head 和 &end() 这两个指针，就可以计算出整个缓存中是否有元素？有多少元素？而在接口1中，因为一开始没有元素，当我们预存一个元素以后，就需要更新accumulate_head到end()的位置。

弄懂了这个，下一个接口也很容易实现。

接口3：删除缓存的最后一个元素

```
//-删除累计缓存的最后一个数据，循环调用此方法可以清空所有数据
    bool clear_last_push_buffer(){
        //-如果缓存没数据，就直接返回
        if(accumulate_head == &sm_list.end()){
            return false;
        }
        else{
            sm_list.pop_back();
            return true;
        }
    }
```

由上图可知，当accumulate_head ==&sm_list.end()的时候，意味着缓存为空，没有元素可以删除。

否则我们调用pop_back接口，删除最后一个元素即可，而如果我们需要删除缓存中所有元素，循环调用此方法直到返回false为止。

接口4：更新生产者队列可读终点

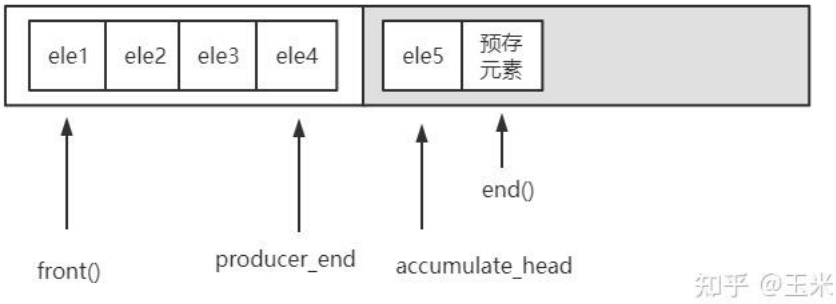
```
//-更新producer_end（生产者可见队列终点）,而返回值代表消费者是否之前读完了队列中所有元素。
bool updateProducerEnd(){
    //-如果标志位的旧值没有变，意味着消费者线程之前没读完，依然在运行
    if(sign.atomic_cas(producer_end,accumulate_head) == producer_end){
        //-更新可读元素结尾
        producer_end = accumulate_head;
        return false;
    }else{
        //-如果旧值≠read_end,只可能是被消费者读完所有元素后改成nullptr了，
        sign.set(accumulate_head);
        producer_end = accumulate_head;
        return true;//-反映出之前消费者是否读完了数据，若为true，则后续可以通知生产者给其发信号告知有新元素可读了
    }
}
```

可以发现，接口2，3，4都是提供给生产者线程的。

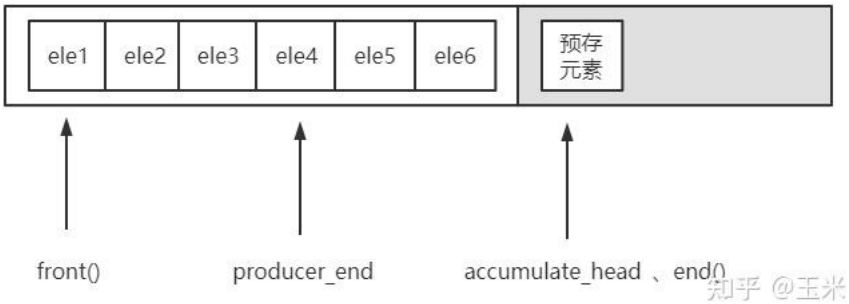
lock_free_queue对写数据设置了两道门槛，即便当生产者设置complete为true，将缓存中的数据真正写入到队列，这些数据目前也只有生产者可见，消费者并不可见，所以producer_end就代表了生产者可见的数据队列终点。

一般来说，当消费者调用write后，都会紧接着调用updateProducerEnd，更新生产者可见队列终点，所以producer_end 会一直指向accumulate_head。

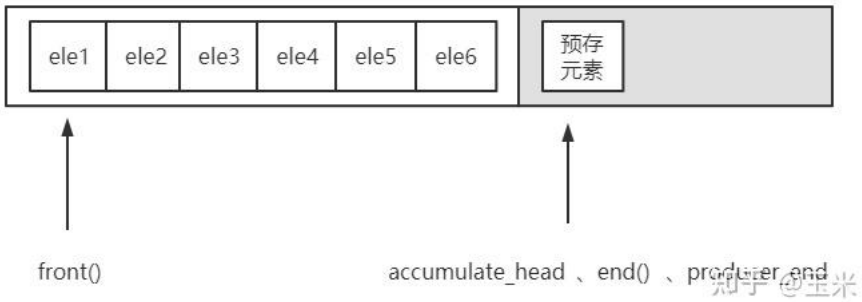
当我们complete false write，producer_end依然指向ele4



当我们complete true write之后



当我们调用updateProducerEnd()后，producer_end才真正更新到accumulate_head的位置。



那么就有个疑惑，为什么要设置两道门槛？既然producer_end始终指向accumulate_head，那直接用accumulate_head指示生产者可见队列终点位置不是更方便吗？

这里就必须强调一点，producer_end的作用除了代表生产者可见队列终点，最根本的作用其实是配合原子指针sign用作比较的锚点，这也是本文最难理解的地方。

这个atomic_ptr<T> sign 到底是什么意思呢。

这还要从updateProducerEnd()这个接口的意义出发，除了更新可见队列终点，其还有一层隐藏的含义，通过返回值来体现，即表示当前队列的空闲状态，updateProducerEnd()的返回值代表了消费者是否读完了所有元素，既然读完了，意味着队列处于空闲状态，生产生就可以通过返回值调整自己生产的速率。

而接口中，消费者是否读完了所有元素，就是通过sign来判断的，所以它是一个标志位。

一般情况下，当队列中有元素的时候，sign会一直等于producer_end，当accumulate_head更新从而≠producer_end，也就是队列有新元素插入的时候，producer_end和sign会同步更新到accumulate_head。

当队列中无元素，也就是消费者读完了所有数据，而生产者还没有生产新数据，此时消费者就会设置该sign为nullptr，并返回读取失败的状态。

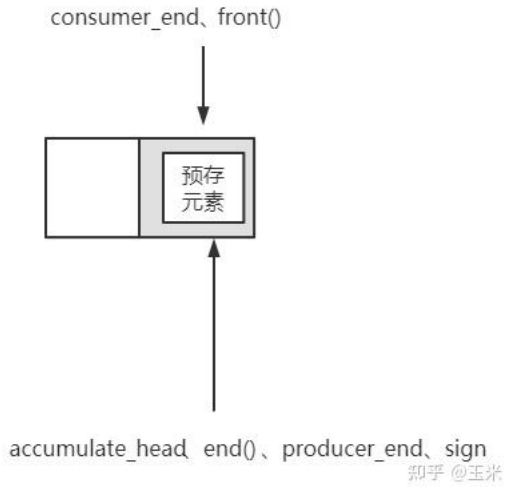
所以sign标志只有两个状态：producer_end和nullptr，只要其为nullptr，就反映出消费者读完状态和队列空闲情况。

接口 5 和 6 就是提供给消费者的api，让我们配合这两个接口一起理解。

```
    //-判断当前是否可读
    bool checkRead(){
        if(&sm_list.front() ≠ consumer_end && consumer_end){
            return true;//-队列中还有元素没读完，返回true
        }
        //-front走到了accumulate_head的位置，意味着读完了，设置sign为nullptr
        consumer_end = sign.atomic_cas(&sm_list.front(),nullptr);
        if(consumer_end == &sm_list.front() || !consumer_end){
            return false;
        }
        //-如果生产者又生产了新的元素
        return true;
    }

    bool read(){
        if(!checkRead()){
            return false;
        }
        sm_list.pop_front();
        return true;
    }
```

consumer_end代表着消费者可见的队列终点，其同样也是sign的一个锚点，该值可以由accumulate_head经由sign这么一个中转最后赋值给了consumer_end，而producer_end同样也是accumulate_head经由sign中转最后赋值的。

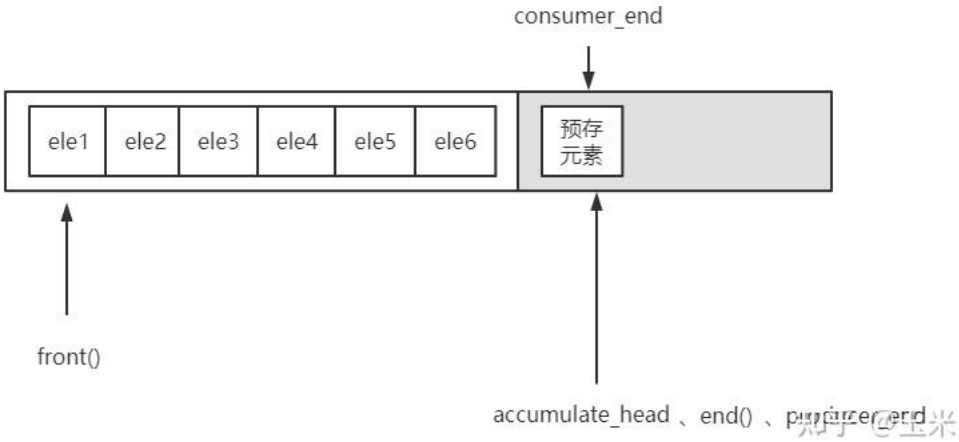
当队列处于空元素状态，此时指针指向如下：



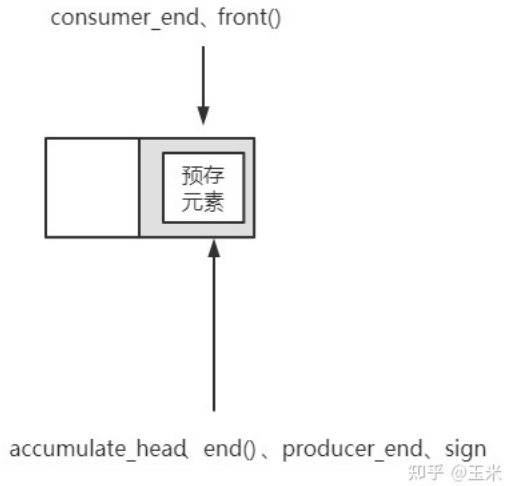
这个状态，其实就是接口1中所有指针的初始化状态。

当生产者线程生产了元素，且期间消费者线程没有运行，accumulate_head会更新，然后生产者调用updateProducerEnd()后，sign更新，sign更新以后带动producer_end一同指向accumulate_head。

这个时候，一旦消费者开始重新跑，comsumer_end就会被sign带动指向accumulate_head。



而在消费者不断读的过程中，假设生产者停止生产，front()指针会一直向后推进，一直到消费者发现front()无法再推进，也就是front()==end()的时候，此时comsumer_end和front()指向相同的位置。意味着所有数据都读完了。



此时，接口5中，sign和front也指向相同的位置，sign.atomic_cas(&sm_list.front(),nullptr)这条cas指令就会成功，消费者会将sign标志位设为nullptr，也就印证了刚刚的接口4。

测试代码

我们可以写一个demo来测试一下：

还是以工作线程和日志线程来距离，因为日志线程要连接数据库写磁盘，读取速度肯定没有工作线程写入队列的速度快。

```
//-生产者快，消费者慢
void* work_thread_func(void* arg){
    lock_free_queue<int,1> *lfqptr = (lock_free_queue<int,1>*) arg;
    for (int i = 0; i < 100;++i) {
        lfqptr→write(i,true);
        cout<<"write "<<i<<" to queue"<<endl;
        if(lfqptr → updateProducerEnd()){
            cout<<"log threads read all"<<endl;
        }
        usleep(200);
    }
    return nullptr;
}

void* log_thread_func(void* arg){
    lock_free_queue<int,1> *lfqptr = (lock_free_queue<int,1>*) arg;
    while(true){
        int temp;
        if(lfqptr →read(temp)){
            cout<<"read "<< temp<<" from queue"<<endl;
        }
        usleep(500);
    }
    return nullptr;
}

int main() {
    //-定义一个无锁队列
    lock_free_queue<int,1> lfq;
    pthread_t tid1;
    pthread_t tid2;
    pthread_create(&tid1, NULL,work_thread_func,&lfq);
    pthread_create(&tid2, NULL,log_thread_func,&lfq);
    pthread_join(tid1,NULL);
    pthread_join(tid2,NULL);
```

```
    return 0;
}
```

完整代码

```
// *基于zeroMQ的改写的单写单读无锁队列，简化易读
#include <stdlib.h>
#include <errno.h>
#include <stdio.h>
#include <iostream>
#include <pthread.h>
#include <unistd.h>

using namespace std;
//-实现一个提供原子操作的指针类型
template<typename T>
class atomic_ptr{
private:
    volatile T* ptr = nullptr;
public:
    //-cas函数,比较ptr和cmp,如果相同,就赋予ptr新值new_ptr，并返回旧值。
    T* atomic_cas(T* cmp, T* new_ptr){
        int *old_ptr;
        __asm__ volatile ("lock; cmpxchg %2, %3"
        : "=a" (old_ptr), "=m" (ptr)
        : "r" (new_ptr), "m" (ptr), "0" (cmp)
        : "cc", "memory");
        return old_ptr;
    }
    //-atomic_set,设置ptr为为新值。返回ptr旧值
    T* atomic_set(T* new_ptr){
        T* old;
        __asm__ volatile ("lock; xchg %0, %2"
                          : "=r" (old), "=m" (ptr)
                          : "m" (ptr), "0"(new_ptr));
        return old;
    }
    //-非原子set
    void set(T* new_ptr){
        ptr = new_ptr;
    }
    //-禁止移动和拷贝
    atomic_ptr() = default;
    atomic_ptr(const atomic_ptr&) = delete;
    const atomic_ptr& operator=(const atomic_ptr&) = delete;
    atomic_ptr(atomic_ptr &&) = delete;
    const atomic_ptr & operator = (atomic_ptr && ) = delete;
};


//-实现无锁队列的存储数据结构
//-第一层看，是一个双向链表
//-第二层看，每个链表节点中有数组 buffer_length-每个节点内数组的长度
template<typename T, int buffer_length>
class storage_model{
private:
    //-私有内部类
    struct storage_node{
        T buffer[buffer_length];
        storage_node * prev;
        storage_node * next;
    };
    //-第一个元素所在的节点（链表头结点，即便整个队列没有元素该节点也会存在）
    storage_node *begin_node;
    //-当前队列中第一个元素所在的begin_node中的buffer的下标
```

```
    int begin_index;

    //-最后一个元素所在的节点（并不一定是当前链表的尾结点）
    storage_node *last_node;
    //-当前队列中最后一个元素所在的last_node中的buffer的下标
    int last_index;

    //-链表尾结点，可能只是一个malloc后无元素的空节点
    storage_node *end_node;
    //-最后一个malloc的元素在end_node中的buffer下标
    int end_index;

    //-假设队列缩减容量，需要释放空的node，会以这个原子指针来存一个最新free掉的node，
    //-预留给下次malloc的时候备用
    atomic_ptr<storage_node> new_free_node;

public:
    //-构造函数
    storage_model():
    begin_node((storage_node*)calloc(1,sizeof(storage_node))),//-初始就有一个node
    begin_index(0),
    last_node(nullptr),//-lastnode是最后一个元素所在的node，没有元素，所以为空
    last_index(0),
    end_node(begin_node),//-链表尾即链表头
    end_index(0)
    {};

    //-尾部插入元素
    void push_back(const T& ele){
        //-将新元素插入到尾部预留的T元素大小的空间
        last_node = end_node;
        last_index = end_index;
        last_node→buffer[last_index] = ele;
        //-给下一个插入的元素预留空间
        malloc_back();
    }

    //-给链表last_index后追加(预留)一个T元素的空间
    void malloc_back(){
        if(++end_index < buffer_length){
            return;
        }
        //-需要新的node
        //-返回free node的元素，看看有没有空闲的node
        storage_node * temp = new_free_node.atomic_set(nullptr);
        if(!temp){//-如果没有free node，只能重新分配
            temp = (storage_node*)calloc(1,sizeof(storage_node));
            if(!temp){
                perror("calloc");
                exit(EXIT_FAILURE);
            }
        }
        end_index = 0;
        end_node→next = temp;
        temp →prev = end_node;
        temp →next = nullptr;
        end_node = temp;
    }

    //-尾部弹出元素（实际上改last_node和last_index的指向就行）
    void pop_back(){
        if(--last_index == -1){
            //-如果弹出该元素后，node的buffer清空
            last_index = buffer_length - 1;
            last_node = last_node→prev;
```

```
    }
    //-回收预留的T元素大小的空间
    free_back();
}

//-回收预留的T元素大小的空间
void free_back(){
    if(--end_index == -1){//-如果删除了预留的空间，end_node的buffer清空
        end_index = buffer_length -1;
        end_node = end_node→prev;
        storage_node * old_free = new_free_node.atomic_set(end_node→next);//-将需要清空的节点更新到new_
        if(old_free){//-真正free掉旧的
            free(old_free);
        }
        end_node →next = nullptr;
    }
}

//-头部弹出元素（由于不用实现头部插入，所以永远不需要给头部预留T空间，pop头部原理和free_back类似）
//-(实际上改begin_node和begin_index的指向就行)
void pop_front(){
    if(++begin_index == buffer_length){
        begin_index = 0;
        begin_node = begin_node→next;
        storage_node * old_free = new_free_node.atomic_set(begin_node→prev);
        if(old_free){
            free(old_free);
        }
        begin_node → prev = nullptr;
    }
}

//-获取第一个元素引用
T& front(){
    return begin_node→buffer[begin_index];
}

//-获取最后一个元素的引用
T& back(){
    return last_node→buffer[last_index];
}

//-返回预留的插入位置
T& end() {
    return end_node → buffer[end_index];
}

//-析构函数
~storage_model(){
    //-顺着node链表free
    while(begin_node≠end_node){
        last_node = begin_node;//-析构的时候last_node就没用了，拿来当个临时指针用而已
        begin_node = begin_node→next;
        free(last_node);
    }
    free(end_node);
    //-如果有free_node也删除
    last_node = new_free_node.atomic_set(nullptr);
    if(last_node){
        free(last_node);
    }
};

//-禁止拷贝和移动
storage_model(const storage_model&) = delete;
const storage_model & operator = (const storage_model &) = delete;
```

```
    storage_model(storage_model &&) = delete;
    const storage_model & operator = (storage_model && ) = delete;
};

//-最上层适配器，无锁队列
template<typename T, int buffer_length>
class lock_free_queue{
private:
    //-复合之前定义的存储结构,表层是个双向链表，所以叫它list
    storage_model<T,buffer_length> sm_list;
    T* producer_end = nullptr;//-对于生产者可见的队列中最后一个元素
    T* consumer_end = nullptr;//-对于消费者可见的队列中的最后一个元素
    T* accumulate_head = nullptr;//-指向累计缓存（未更新）的第一个元素，如果该指针更新，意味着缓存的元素全部更新到队列
    atomic_ptr<T> sign;//-一个标志位，主要用来标志消费者是否读完队列所有元素，

public:
    //-构造
    lock_free_queue(){
        //-先预留一个插入位置
        sm_list.malloc_back();
        accumulate_head = &sm_list.end();
        sign.set(nullptr);
        producer_end = accumulate_head;
        consumer_end = accumulate_head;

    }

    //-向队列写数据(支持缓冲追加),如果complete==false则只会累计在缓存里，不会真的更新队列
    //-若complete == true则立刻把所有累计的一起更新
    void write(const T& ele,bool complete){
        sm_list.push_back(ele);
        if(complete){
            accumulate_head = &sm_list.end();
        }
    }

    //-删除累计缓存的最后一个数据，循环调用此方法可以清空所有数据
    bool clear_last_push_buffer(){
        //-如果缓存没数据，就直接返回
        if(accumulate_head == &sm_list.end()){
            return false;
        }
        else{
            sm_list.pop_back();
            return true;
        }
    }

    //-更新producer_end（生产者可见队列终点）,而返回值代表消费者是否之前读完了队列中所有元素。
    bool updateProducerEnd(){
        //-如果标志位的旧值没有变，意味着消费者线程之前没读完，依然在运行
        if(sign.atomic_cas(producer_end,accumulate_head) == producer_end){
            //-更新可读元素结尾
            producer_end = accumulate_head;
            return false;
        }else{
            //-如果旧值≠read_end,只可能是被消费者读完所有元素后改成nullptr了，
            sign.set(accumulate_head);
            producer_end = accumulate_head;
            return true;//-反映出之前消费者是否读完了数据，若为true，则后续可以通知生产者给其发信号告知有新元素可读了
        }
    }


    //-判断当前是否可读
    bool checkRead(){
```

```
        if(&sm_list.front() != consumer_end && consumer_end){
            return true;//-队列中还有元素没读完，返回true
        }
        //-front走到了accumulate_head的位置，意味着读完了，设置sign为nullptr
        consumer_end = sign.atomic_cas(&sm_list.front(),nullptr);
        if(consumer_end == &sm_list.front() || !consumer_end){
            return false;
        }
        //-如果生产者又生产了新的元素
        return true;
    }

    bool read(T& result){
        if(!checkRead()){
            return false;
        }
        result = sm_list.front();
        sm_list.pop_front();
        return true;
    }
};

//-生产者快，消费者慢
void* work_thread_func(void* arg){
    lock_free_queue<int,1> *lfqptr = (lock_free_queue<int,1>*) arg;
    for (int i = 0; i < 100;++i) {
        lfqptr->write(i,true);
        cout<<"write "<<i<<" to queue"<<endl;
        if(lfqptr -> updateProducerEnd()){
            cout<<"log threads read all"<<endl;
        }
        usleep(200);
    }
    return nullptr;
}

void* log_thread_func(void* arg){
    lock_free_queue<int,1> *lfqptr = (lock_free_queue<int,1>*) arg;
    while(true){
        int temp;
        if(lfqptr ->read(temp)){
            cout<<"read "<< temp<<" from queue"<<endl;
        }
        usleep(500);
    }
    return nullptr;
}

int main() {
    //-定义一个无锁队列
    lock_free_queue<int,1> lfq;
    pthread_t tid1;
    pthread_t tid2;
    pthread_create(&tid1, NULL,work_thread_func,&lfq);
    pthread_create(&tid2, NULL,log_thread_func,&lfq);
    pthread_join(tid1,NULL);
    pthread_join(tid2,NULL);
    return 0;
}
```

编辑于 2022-07-28 10:52 · IP 属地陕西

更多回答