



下载APP



03 | 双端队列：并行计算中的工作窃取算法如何实现？

2021-12-16 黄清昊

《算法实战高手课》

课程介绍 >



讲述：黄清昊

时长 13:46 大小 12.62M



你好，我是微扰君。

目前我们已经学习了 vector 动态数组和 list 双向链表两种 STL 中的序列式容器了，今天我们继续学习另一种常见的序列式数据结构，双端队列。

在并行计算中，我们常常会用多进程处理一些复杂的计算任务。为了能够通过多进程加速计算，我们除了需要对任务进行合理的切分，也需要将任务合理公平地分配到每一个进程。简单来说就是，我们希望每个进程都不至于闲着。那怎么样能做到这件事呢？



其实有一种非常常用的算法，工作窃取算法，就可以用来达成这个目标，它就需要用到我们今天的主角——双端队列。

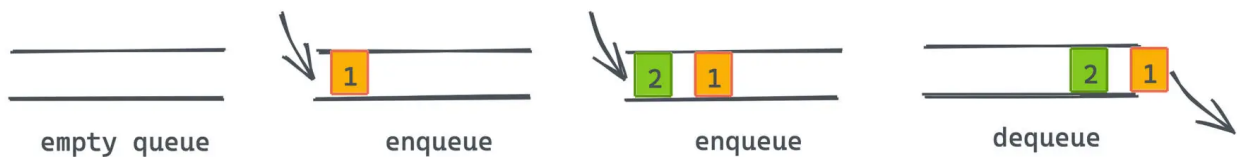
队列

要介绍双端队列，我们先来聊一聊队列，queue。什么是队列呢？

从概念上来说其实非常好理解，因为它的特性和“队列”这个词在现实生活中的意思是一致的，那就是 FIFO 先进先出。简单来说就是排队。

比如说现在到很多餐厅就餐，服务员都会给你发一个号码让你排队，等有空位的时候，服务员叫号是按照取号的顺序来的，肯定是先来取号的人结束排队去入座；这样的约束就是先进先出。

显然这种**先进先出的队列也是一种典型的序列式数据结构**；和数组最大的区别就在于，它是一个有约束的序列式数据结构，因为先进先出的特性要求我们，所有的插入操作必须在队列的尾部进行，而所有的删除操作则必须在队列的头部进行。



queue 的特性FIFO 先进先出



上图就是一个对队列入队、出队操作的示例。我们注意到先入队的元素一定会比后入队的元素更早出队。这一特性和思想在许多业务系统或者基础软件、操作系统、计算机网络中都有应用，比如在操作系统中的 CPU 调度中，进程资源使用 CPU 的顺序就用队列来排序。

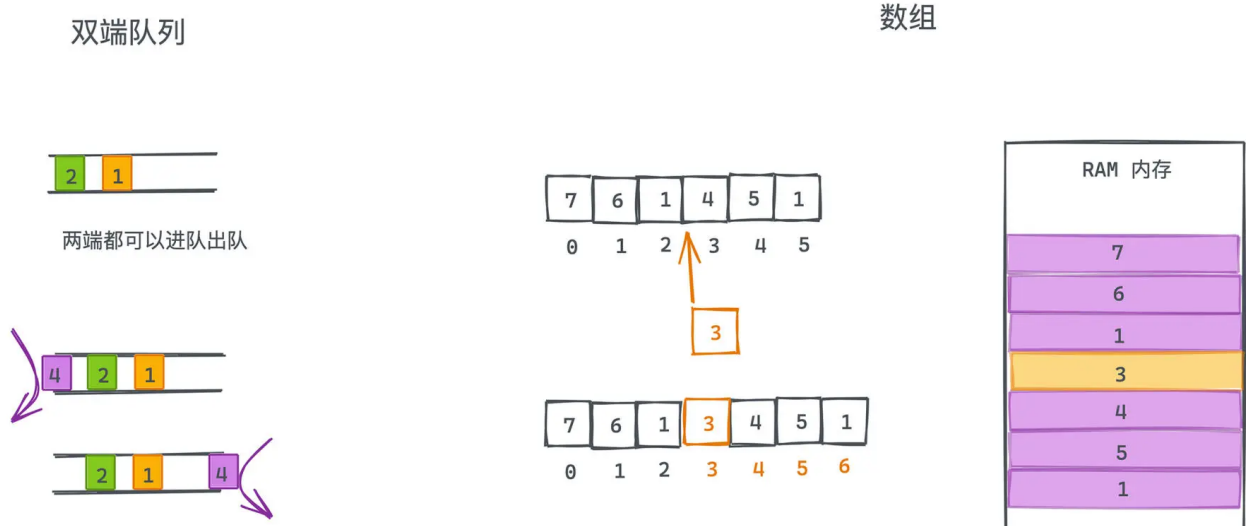
双端队列

队列和链表一样也会展现出更多种类的队列，比如带权重的优先队列、或者只能一边进一边出的单端队列。

我们今天要实现的双端队列 `double ended queue`，双端队列是其中一种，相比于普通队列而言，双端队列是两端开口的，在队列的头尾两端都可以进行进队和出队操作，让我们在使用队列时有了更大的灵活性。

你肯定想问，数组也可以在两边插入数据呀，那双端队列和数组有什么区别呢？

首先，数组头部的插入操作复杂度很高，如果我们并不需要快速随机访问，这种操作的复杂度是完全可以避免的，这是双端队列和数组的一个很大区别。更本质的地方在于，**双端队列仅仅是一个两端都支持 FIFO 插入删除操作的队列，语义上来说并不支持数组基于下标在指定位置的修改、插入和删除的操作。**



当然，我们是可以用数组或者链表来模拟实现双端循环队列的，只要暴露出经过剪裁的且满足 FIFO 的语义方法就可以了。

比如可以开一个大小为 N 的数组 `array`，用两个数字 `rear` 和 `front` 代表队列的前端和尾端。在前端插入 `target`，只需要 `array[(--front+N)%N] = target`，这样既扩展了前端的边界，也达到了插入 `target` 的效果。`%N` 也就是要对 N 取模，主要也就是为了处理越界的问题，这样当数组的前端 `read` 到达小于 0 的位置时，就会马上变成 $N-1$ ，也就实现了一个循环队列。

Deque 实现

虽然说，可以用数组或者链表来实现队列，但 **C++ 并没有选择依赖已有的序列式容器 `vector` 或者 `list` 来实现**，原因是什么呢？你可以先想一想。

带着这个问题，我们一起来学习后面的内容，看看 STL 中的 `deque` 是如何实现一个高效好用的双端队列的。

我个人认为，在 STL 序列化容器的空间分配中，`deque` 可能是最复杂的，这也可能会对你阅读源码造成一定的障碍，但是不要害怕，如果只是为了搞清 `deque` 设计的大致思想，我们完全可以将内存分配的部分当成黑盒来看，这对搞清楚 `deque` 的原理并没有什么影响。

Deque 的内存布局

`deque` 的内存布局，可以说同时具备了 `list` 和 `vector` 的特点。

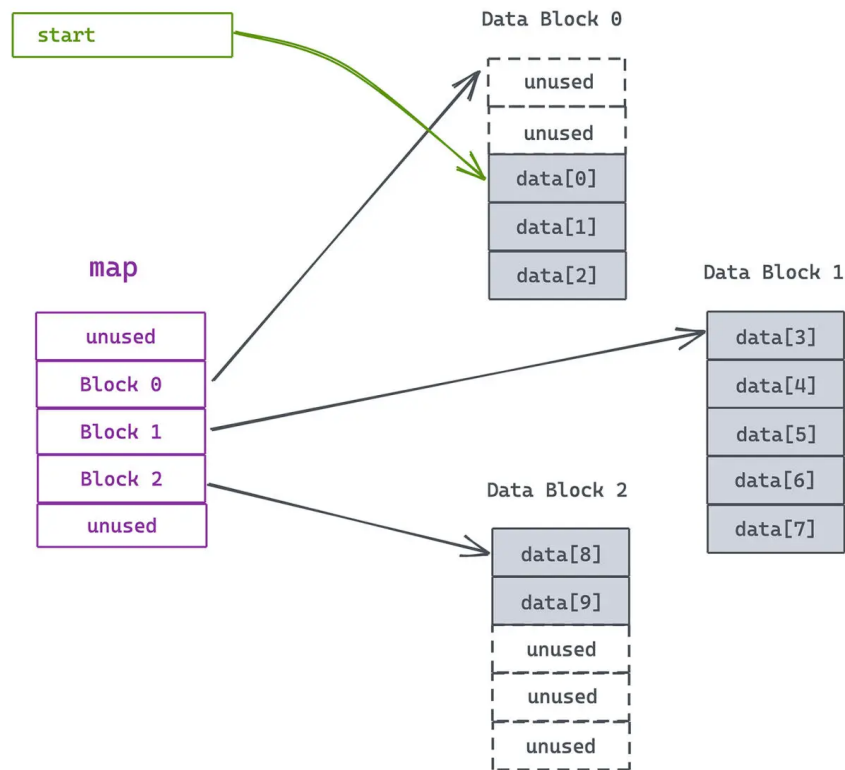
`deque` 的内存布局是由一段段连续的空间、用另一个类似数组的东西将这些空间的地址信息拼接在一起组成的，真实存放数据的就是那一段段连续的空间。在首尾两端插入和删除的时间复杂度是 $O(1)$ 。以插入为例，每次一段连续的空间元素被用完的时候，会直接申请一段新的空间并链接到 `deque` 的分段空间末尾。

所以 `deque` 既不像 `vector` 那样每次扩容都需要付出复制和拷贝的高昂代价，也不会像链表那样每次插入一个新的节点都需要申请一次内存。

当然这也导致了非常复杂的控制流程，`deque` 的代码量也远远多于 `vector` 和 `list`。

为了维护一段段连续的内存空间，`deque` 需要维护一个被称为 `map` 的成员变量；这个 `map` 数据结构起到了管理真正用于存储队列元素的一段段连续线性空间的作用。那一段段连续的线性空间，我们称为缓冲区。

`map` 的示意图如下：



可以认为 `map` 是一个数组，每个元素指向了一段缓冲区的地址。而缓冲区对应了一段指定大小的连续内存空间，默认大小为 512 bytes。

复制代码

```

1  template <class _Tp, class _Alloc>
2  class _Deque_base {
3      ...
4  protected:
5      _Tp** _M_map;
6      size_t _M_map_size;
7      iterator _M_start;
8      iterator _M_finish;
9      ...
10 }
```

因此 `_M_map` 在数据结构中的表现就是一个二级指针。`_M_map_size` 指的就是 deque 中 `map` 的空间大小，即在 `map` 中最多能存储多少个指针。如果 `map` 的空间已经被用满了，我们也会对 `map` 进行一次重新分配迁移的操作，核心思想和 `vector` 的重分配其实是一样的，我们马上具体讲。

Deque 的迭代器

介绍完内存布局和基本数据结构，下一个重点就是 STL 的通用访问模式，迭代器的实现了。

正是因为 **deque 底层实质是分段连续空间**，operator++ 和 operator-- 的实现也变得更困难一些，迭代器既要能找到与当前缓冲区相邻的缓冲区在哪；也需要知道目前访问的地方是否已经到当前缓冲区的边缘，只有这样到边缘时，才能正确跳转。

为了方便达到这一目标，我们需要在迭代器的数据结构中记录一下迭代器在当前缓冲区的位置，同时记录当前缓冲区的开始位置和结束位置，以及缓冲区的 map 指针：

[复制代码](#)

```

1  template <class _Tp, class _Ref, class _Ptr>
2  struct _Deque_iterator {
3      typedef _Deque_iterator<_Tp, _Tp&, _Tp*>          iterator;
4      typedef _Deque_iterator<_Tp, const _Tp&, const _Tp*> const_iterator;
5      static size_t _S_buffer_size() { return __deque_buf_size(sizeof(_Tp)); }
6      ...
7      typedef _Tp** _Map_pointer; // 缓冲区指针
8      ...
9      _Tp* _M_cur; // 当前缓冲区的位置
10     _Tp* _M_first; // 缓冲区的左边界线
11     _Tp* _M_last; // 缓冲区的右边界
12     _Map_pointer _M_node;
13     _Deque_iterator(_Tp* __x, _Map_pointer __y)
14         : _M_cur(__x), _M_first(*__y),
15           _M_last(*__y + _S_buffer_size()), _M_node(__y) {}
16 }

```

有了位置的记录，operator++ 可以这样实现：

[复制代码](#)

```

1  _Self& operator++() {
2      ++_M_cur;
3      if (_M_cur == _M_last) {
4          _M_set_node(_M_node + 1);
5          _M_cur = _M_first;
6      }
7      return *this;
8  }
9  void _M_set_node(_Map_pointer __new_node) {
10     _M_node = __new_node;
11     _M_first = *__new_node;
12     _M_last = _M_first + difference_type(_S_buffer_size());

```

```
13     }
```

核心的就是 `_M_set_node` 方法，如果我们发现 `M_cur` 已经达到了当前缓冲区的尾部，就将它移动到下一段缓冲区的头部，更新迭代器中当前 `map` 的位置。另外，也需要将 `_M_first` 和 `_M_last` 更新为新的缓冲区的左确界和右虚界。

-- 的操作类似：

[复制代码](#)

```
1 _Self& operator--() {
2     if (_M_cur == _M_first) {
3         _M_set_node(_M_node - 1);
4         _M_cur = _M_last;
5     }
6     --_M_cur;
7     return *this;
8 }
```

我们发现 `M_cur` 达到缓冲区头部的时候，就要将它移动到当前缓冲区的前一段缓冲区了，调用 `set_node` 方法即可。

到这里就完成了迭代器的主要接口，这让我们将内存实质不连续的真相隐藏了起来，取而代之地提供了一个非常简洁好用的遍历 `deque` 的接口。

好啦，学完 `deque` 的内存布局 and 迭代器如何实现，你知道它的基础操作该怎么写了吗？

Deque 的基础操作

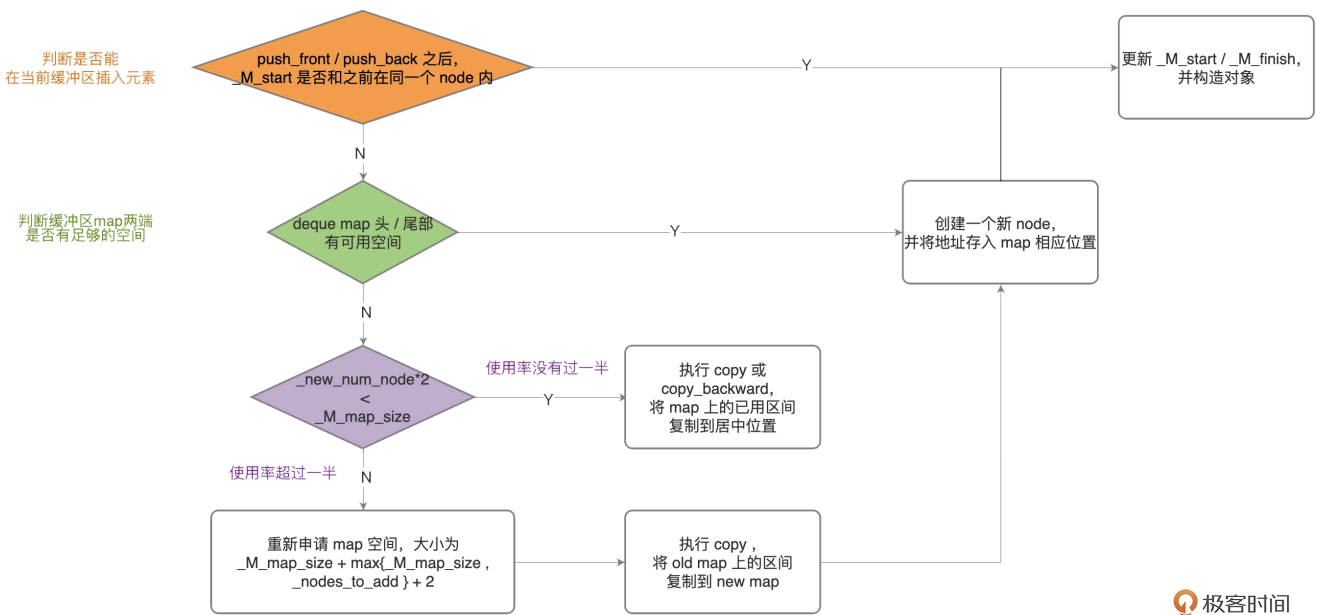
相比于 `vector` 和 `list` 来说，`deque` 支持的操作要少得多，只有基本的 `push` 和 `pop` 实现，因为队列语义保证了我们不会在队列中间进行插入删除操作，也就不需要支持 `insert` 和 `erase` 这样的操作了。

不过正因为内存布局复杂，`deque` 的内存管理扩缩容的逻辑也比较复杂，我们了解大概思想就可以了。如果你感兴趣可以自行查阅 `deque` 源码。

push 操作

Deque 的第一个操作当然是 `push_front` 和 `push_back`，因为我们实现的是双端队列，所以头部尾部都有可能插入数据。

遇到内存不足的时候，deque 会按照下图的逻辑进行扩容，有几个检查点，首先判断是不是能在当前缓冲区插入元素，如果可以，直接插入就行；如果不能，就要检查缓冲区 `map` 两端是否有足够的空间；如果有的话，也很简单，直接创建一个新的缓冲区并存入 `map`。



极客时间

关键是在 `map` 空间不足的时候，也就是插入的数据已经达到 `map` 头部或者尾部缓冲区的边界时，我们可以分两种情况讨论：

1. **如果 `map` 使用率已经超过一半**，我们就可以重新申请更大的空间，把老的 `map` 上的数据拷贝到新的区域。这里注意，`map` 中指向的那些缓冲区里的数据并不用变化，只需要一个更大的 `map` 去放那些缓冲区的指针，和动态数组扩容的方式如出一辙。
2. **`map` 使用率没有超过一半**，这时候我们认为申请新的空间可能是浪费的，所以只是将数据重新调整到 `map` 中间的位置，当然也要进行一次拷贝。这可能会帮我们节约大量的空间。

翻译成代码如下：

复制代码

```

1 void push_back(const value_type& __t) {
2     if (_M_finish._M_cur != _M_finish._M_last - 1) {
3         construct(_M_finish._M_cur, __t);

```




```

4      ++_M_finish._M_cur;
5  }
6  else
7      _M_push_back_aux(__t);
8  }
9  template <class _Tp, class _Alloc>
10 void deque<_Tp, _Alloc>::_M_push_back_aux()
11 {
12     _M_reserve_map_at_back();
13     *(_M_finish._M_node + 1) = _M_allocate_node();
14     __STL_TRY {
15         construct(_M_finish._M_cur);
16         _M_finish._M_set_node(_M_finish._M_node + 1);
17         _M_finish._M_cur = _M_finish._M_first;
18     }
19     __STL_UNWIND(_M_deallocate_node(*(_M_finish._M_node + 1)));
20 }

```

pop 操作

pop 操作不再需要处理插入导致的扩容拷贝问题, 相对来说就显得简单很多。以 pop_back 为例, 我们只需要关注是否已经 pop 到某一段缓冲区的边界。

 复制代码

```

1 void pop_back() {
2     if (_M_finish._M_cur != _M_finish._M_first) {
3         --_M_finish._M_cur;
4         destroy(_M_finish._M_cur);
5     }
6     else
7         _M_pop_back_aux();
8 }
9 // Called only if _M_finish._M_cur == _M_finish._M_first.
10 template <class _Tp, class _Alloc>
11 void deque<_Tp, _Alloc>::_M_pop_back_aux()
12 {
13     _M_deallocate_node(_M_finish._M_first);
14     _M_finish._M_set_node(_M_finish._M_node - 1);
15     _M_finish._M_cur = _M_finish._M_last - 1;
16     destroy(_M_finish._M_cur);
17 }

```

如果发现当前迭代器已经和缓冲区的首位置相同, 除了释放掉当前的内存, 还需要释放掉整段缓冲区的内存, 并且将迭代器的缓冲区指针, 指向当前缓冲区前一段的位置, 这可以

通过 `_M_set_node` 方法达成。当然，由于我们还需要 `pop` 一个节点，所以会将 `_M_cur` 指向 `_M_finish._M_last-1` 的位置。

C++ 的选择

现在掌握了 `deque` 的实现和基本操作，我们来回答一下为什么 C++ 不选择依赖已有的序列式容器来实现 `deque`？

其实我们已有的容器就两个，一个是 `vector`，另外一种就是 `list`。

显然，基于 `vector` 实现，不能真的在头部插入元素，会产生 $O(N)$ 的时间开销，我们只能用一个固定大小的 `vector` 来模拟循环队列，具体实现方式前面说过。但这样就导致我们**必须事先确定数组的最大容量，让它的大小是实现分配好的，这就和数组一样，也会产生内存浪费和无法动态扩容的问题。**

不过在最大容量能确定的场景下，用 `vector` 也是一种非常常见的循环队列实现方式。

而基于 `list`，看起来首尾都可以 $O(1)$ 的时间插入，但对数据的随机读取性能会很差；且每次插入元素都需要申请内存，相比于 `deque` 一次申请一段内存的方式也会带来额外的性能开销。而 **`list` 的最大优势，任意位置的快速插入 / 删除能力，我们却用不上。**

所以基于 `deque` 的使用场景，C++ 设计了基于 `map` 分段存储的双端队列的数据结构，能同时具备 `list` 和 `vector` 的特点。

总结

队列的基本特性是 FIFO，也就是先进先出，它能衍生出几种不同的形式，包括循环队列、双端队列，既可以通过数组实现，也可以通过链表实现。

STL 的 `deque` 是一种双端队列的实现，内存布局是由一段段连续内存串联起来的，在队列两端都可以 `pop` 和 `push` 数据。因为复杂的内存分配，代码实现的难度要高很多。但更多的复杂性还是体现在内存管理中，只要我们通过迭代器等模式，将底层的逻辑封装起来，相信你也看到了，`pop` 和 `push` 操作的思路其实是非常清晰好懂的。

现在你知道为什么说工作窃取算法需要用到双端队列了吗？

我们一起来看看。为了更公平也更高效地分配每个进程负责的任务，我们可能会多开很多个队列去存储任务，每个进程就去消费一个队列中的任务，这样就可以有效避免进程间的竞争。因为任务先进先出，用一个普通的单向队列就可以完成了。

但是你可能很难保证任务划分得非常均匀，使得每个进程完成所有任务的时间都差不多。这不是一个很好解决的问题。但是如果我们**换一个思路，不再费心让任务分配得均匀，只是简单地允许先完成任务的进程，去其他进程的队列盗取任务，是不是就不会有进程闲置了呢？**

不过怎么盗取，可以让我们仍然尽量规避进程间的竞争问题呢？相信你已经想到答案了，没错，就是双端队列。我们让盗取任务的进程，从队列的另一端盗取就行了，这样只有队列长度为 1 的时候才会出现竞争。当然还有很多实现细节，你感兴趣的话可以去看一下 Java 中 ForkJoinPool 的实现。

课后作业

最后，同样给你留一个课后作业。我们讲解了如何用数组实现队列，也提到队列同样可以通过链表来实现？你可以试着实现一下吗？

欢迎你留言与我讨论交流~

分享给需要的人，Ta订阅后你可得 **20 元现金奖励**

 生成海报并分享

 赞 1  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 [02 | 双向链表：list如何实现高效地插入与删除？](#)

精选留言 (5)

 写留言



灵奈山艾付

2021-12-16

有一种双端队列的实现方法是用两个 queue，头对头，这样也可以做到类似 deque 的效果。但是为什么 STL 不采用这种实现方式呢，疑惑。

展开



那一刻

2021-12-16

请问老师，如果 map 使用率已经超过一半，我们就可以重新申请更大的空间，把老的 map 上的数据拷贝到新的区域。请问什么这么做呢？如果不重新申请大的内存，而是增加一个block，如此岂不是节省了拷贝的开销？

展开



魔

2021-12-16

还有一点 这个玩意我感觉和go的gmp 一样一样的

展开



魔

2021-12-16

简单的总结:

看了下老师的文字描述，感觉这个deque在我的理解上就是 map是一个不定长数组了 然后里面的每个元素就是一段连续的空间(也是节点数组) 这样就可以拼接起来实现双端队列。然后对于扩容操作 如果发现某一个端点在map层用完了 那么判断是否超过总容量的百分之五十 如果没有超过证明某一段比较数据集中，另外一点数据较少可以移动到中间来 无需...

展开



Paul Shan

2021-12-16

Deque的实现分了两层，第一层是不定长的循环对列管理数据块指针，第二层是定长的数据块，管理实际元素的存取。和链表相比，这种实现因为有了定长的数据块，可以减少添加和删除内存的数目，也省去了每个节点的指针。有了第一层不定长的循环队列，对于插入和删除元素都能做到 $O(1)$ 的均摊复杂度，这里使用的指针数目是使用双向链表 $1/2m$ ，（m是块的大小）

展开



