



图 6.2 循环向量示例

考虑以模板类buffer实现. 假设buffer<T>类中包含一个长为 $n$ 的vector<T>型向量Q, 需要实现如下成员函数:

```

0 void buffer<T>::push(const T& item);    // 将新元素item放到队尾.
1 void buffer<T>::pop();                  // 删除队首元素.
2 bool buffer<T>::full() const;           // 判断队列是否已满.
3 bool buffer<T>::empty() const;          // 判断队列是否为空.

```

此外, 我们还得保证 $n > 0$ . 为简便起见, 下面略去代码实现中关于模板参数的部分, 并且以图6.2为例讲解.

而在讨论设计之前, 必须对其做简单的模型假设. 首先push操作必须和full操作搭配使用, 而pop操作也必须和empty操作搭配使用. 其次push操作和pop操作的总次数基本相等, 因为每个元素的出入基本守恒. 基于上述考虑, 我们必须设计出综合性能较好的方案, 即不能为某个成员函数特殊优化. 一般而言, 实现一种抽象数据类型之前, 对其作出简单且有针对性的考察会大有裨益.

### 6.3.1 使用两个位置指示

不妨在buffer模板类中设置front和rear两个size\_t型变量表示队列首尾元素的位置信息, 下面基于这两个变量实现buffer模板类的成员函数.

#### 利用真实首尾位置

容易想到的方案是真实位置: 令front为队首元素位置(对应图6.2中 $q_0$ ), 而rear为队尾元素位置(对应图6.2中 $q_4$ ). 于是入队和出队操作分别实现为:

```

0 void buffer<T>::push(const T& item)
1 {
2     rear = (rear + 1) % n;    // 先改变队尾位置.
3     Q[rear] = item;          // 再插入新元素.
4 }
5
6 void buffer<T>::pop()
7 {
8     front = (front + 1) % n;
9 }

```

据此可设计判断队空和队满的条件. 若队列中仅有一个元素, 则此时`front`和`rear`应相等, 逆向思考此前的入队操作, 于是`empty`函数为:

```
0  bool buffer<T>::empty() const
1  {
2      return (front == (rear + 1) % n);
3  }
```

但是随之而来的问题是队列中不能存储 $n$ 个元素, 因为这样会使队空和队满的判断条件完全一样而招致错误(例如队列全满时无法执行出队操作). 为此需限制队列中最少得留一个空位, 因此`full`函数为:

```
0  bool buffer<T>::full() const
1  {
2      return (front == (rear + 2) % n);
3  }
```

`front`和`rear`的初值需满足队空条件, 且最好取值为 $Q$ 中真实位置, 例如`front`可取0, 而`rear`取 $n - 1$ .

#### 队尾位置挪后

可以看出, 直接使用真实队尾位置时, 判断队空和队满的条件都比较复杂. 关键在于对`rear`的操作, 不妨将其向后挪一格, 让`rear`指向队尾后一元素的位置(即图6.2中 $e_0$ ). 于是可实现`buffer`模板类的成员函数如下:

```
0  void buffer<T>::push(const T& item)
1  {
2      Q[rear] = item;           // 注意此时先插入新元素.
3      rear = (rear + 1) % n;    // 再改变队尾位置.
4  }
5
6  void buffer<T>::pop()
7  {
8      front = (front + 1) % n;
9  }
10
11 bool buffer<T>::empty() const
12 {
13     return (front == rear);
14 }
15
```

```
16 bool buffer<T>::full() const
17 {
18     return (front == (rear + 1) % n);
19 }
```

可以看出这里实际上相当于用 $(\text{rear} + 1) \% n$ 替换了 $\text{rear}$ , 从而减少了判断时的算术运算. 有兴趣的读者可思考为何不用 $(\text{rear} + 2) \% n$ 替换 $\text{rear}$ 而去简化判断队满操作呢?

$\text{front}$ 和 $\text{rear}$ 的初值选取规则可相应给出. 注意为保证首次入队时数据能存于 $Q[0]$ , 应令 $\text{front}$ 和 $\text{rear}$ 初始均为0.

事实上, 此方案的物理意义也很明确, 即 $\text{front}$ 指向队首元素, 而 $\text{rear}$ 指向当前需要插入的位置. 处理队首元素找 $\text{front}$ 即可, 而 $\text{rear}$ 位置当前无元素, 入队可直接放入.

### 队首位置提前

虽然从逻辑上看,  $\text{front}$ 向前挪一格可以取得与 $\text{rear}$ 向后挪一格相同的效果, 但是这样会让队首元素变为 $Q[(\text{front} + 1) \% n]$ , 而实际中我们常常要操作队首元素而很少关心队尾元素, 因此该方案不甚合适.

### 6.3.2 使用计数信息

仅使用队列首尾位置不但不能利用向量的所有空间, 还会让判断队空和队满的条件变得复杂. 然而, 究其本质是信息不足. 实际上, 无论在何种方案中, 若固定队列的 $\text{front}$ 变量, 则队列中元素个数存在 $n + 1$ 种情况(注意存在队空状态), 而 $\text{rear}$ 变量的取值只有 $n$ 种可能(从0到 $n - 1$ ). 矛盾正在这里. 此外, 这也是前文中我们先定好队空条件再考虑队满条件的原因, 若设定队满条件为向量所有元素均被利用, 那么队空条件就无法考虑了.

由于队列中有一个元素无法利用, 我们索性再加入变量 $\text{count}$ 来记录队列中当前元素个数, 从而简化了判断队空和队满操作. 而从抽象数据类型角度看, 队列通常得具备 $\text{size}$ 成员函数, 因此 $\text{count}$ 变量一身二任, 非常划算.

### 取模运算优化

注意到这里的取模运算其实意义不大而且速度较慢, 仅在变量取值为 $n$ 时才变成0, 所以我们可以改用判断语句来提升效率.

### 6.3.3 缓冲区

我们采用 $\text{count}$ 作为状态信息, 注意到缓冲区实际上是一个循环意义下的区间(队尾位置挪后), 所以改用 $\text{left}$ 和 $\text{right}$ 的表述形式, 这样就得到了一个较好的缓冲区实现方案.

在线代码 27

缓冲区模板类    ↗ <https://github.com/xiexiexx/DSAD/blob/main/second-edition/src/buffer.h>



```
0  #include <vector>
1
2  #ifndef BUFFER_CLASS
3  #define BUFFER_CLASS
4
5  template <typename T>
6  class buffer {
7  public:
8      buffer(size_t n);
9      T& front();
10     const T& front() const;
11     void push(const T& item);
12     void pop();
13     size_t size() const;
14     bool empty() const;
15     bool full() const;
16     size_t capacity() const;
17 private:
18     std::vector<T> Q;
19     // 缓冲区容量实为Q.size(), 单独设置length是为了避免频繁调用.
20     size_t length;
21     // 缓冲区元素个数.
22     size_t count;
23     // 缓冲区循环意义下的区间[left, right).
24     size_t left;
25     size_t right;
26 };
27
28 // 构造函数的实现.
29 template <typename T>
30 buffer<T>::buffer(size_t n)
31     : Q(n), length(n), count(0), left(0), right(0)
32 {
33 }
34
35 // 返回队首.
36 template <typename T>
37 T& buffer<T>::front()
```

```
38 {
39     return Q[left];
40 }
41
42 // 返回队首的常量版本.
43 template <typename T>
44 const T& buffer<T>::front() const
45 {
46     return Q[left];
47 }
48
49 // 入队操作.
50 template <typename T>
51 void buffer<T>::push(const T& item)
52 {
53     // 由用户判断是否缓冲区已满, 此处只实现入队.
54     Q[right++] = item;
55     if (right == length)
56         right = 0;
57     ++count;
58 }
59
60 // 出队操作.
61 template <typename T>
62 void buffer<T>::pop()
63 {
64     if (++left == length)
65         left = 0;
66     --count;
67 }
68
69 // 返回缓冲区实有元素个数.
70 template <typename T>
71 size_t buffer<T>::size() const
72 {
73     return count;
74 }
75
```

```
76 // 判断缓冲区是否为空.
77 template <typename T>
78 bool buffer<T>::empty() const
79 {
80     return (count == 0);
81 }
82
83 // 判断缓冲区是否已满.
84 template <typename T>
85 bool buffer<T>::full() const
86 {
87     return (count == length);
88 }
89
90 // 返回缓冲区最大容量.
91 template <typename T>
92 size_t buffer<T>::capacity() const
93 {
94     return size;
95 }
96
97 #endif // BUFFER_CLASS
```

## 6.4 queue的实现原理

在线代码 28

Clang队列实现 ↗ <https://github.com/llvm/llvm-project/blob/main/libcxx/include/queue>



STL中的queue仅仅是一个适配器, 其底层由deque或list实现, 读者可仿照stack实现之.<sup>1</sup> 不过, 队列更注重空间的高效利用, 我们已经在缓冲区的实现中有所体会. 实际上, 缓冲区可以拓展到双端队列,<sup>2</sup> 进而特化为队列.

在线代码 29

Clang双端队列实现 ↗ <https://github.com/llvm/llvm-project/blob/main/libcxx/include/deque>



<sup>1</sup> C++标准将deque作为stack和queue的默认底层容器.

<sup>2</sup> 前文已经用循环向量设计并实现了模板类buffer, 不妨在其基础上添加操作以实现双端可操作的缓冲区, 另外再考虑扩容的实现.