

# 基本的循环队列 (Circular Queue) 数据结构

马浩琨

Sep 13, 2025

在日常生活中，我们经常遇到排队的场景，例如在超市结账时，顾客按照先来后到的顺序依次处理，这就是队列的直观体现。队列是一种先进先出 (FIFO) 的线性数据结构，它只允许在队尾添加元素，在队头删除元素。这种特性使得队列在计算机科学中广泛应用于任务调度、缓冲处理等场景。

然而，当使用普通数组实现队列时，我们会遇到一个棘手的问题：假溢出。假溢出指的是数组前端有可用的空闲空间，但由于队尾指针已经到达数组末尾，无法再插入新元素，导致空间浪费。这种现象不仅降低了存储效率，还可能引发程序错误。为了解决这个问题，循环队列应运而生。循环队列通过将数组首尾相连，逻辑上形成一个环，从而高效利用空间，避免了假溢出。

## 1 循环队列的核心思想与工作原理

循环队列是一种基于固定大小数组的数据结构，它通过两个指针 (`front` 和 `rear`) 的循环移动来实现队列操作。其核心在于“循环”二字：当指针到达数组末尾时，通过取模运算，下一个位置会回到数组开头，从而形成逻辑上的环。

初始化时，`front` 和 `rear` 指针都设置为 0，表示队列为空。入队操作时，首先检查队列是否已满，然后将元素放入 `rear` 指向的位置，并将 `rear` 指针前进一位，使用公式 `rear = (rear + 1) % capacity` 来实现循环。出队操作时，检查队列是否为空，然后取出 `front` 指向的元素，并将 `front` 指针前进一位，同样使用公式 `front = (front + 1) % capacity`。

取模运算在这里扮演了关键角色。例如，假设数组容量为 5，当 `rear` 为 4 时， $(4 + 1) \% 5 = 0$ ，这意味着 `rear` 指针会从未尾回到起点，实现了循环移动。这种机制确保了队列空间的高效利用，避免了假溢出。

## 2 循环队列的边界条件处理

在循环队列中，一个重要的挑战是如何区分队列“空”和“满”的状态。因为当队列空时，`front` 等于 `rear`；当队列满时，`front` 也可能等于 `rear`，这会导致混淆。为了解决这个问题，常用的方法有两种：浪费一个空间法和维护计数器法。

浪费一个空间法是最常用且直观的方法。它规定数组中始终浪费一个单元，作为“满”的标识。判空条件是 `front == rear`，判满条件是 `(rear + 1) % capacity == front`。这种方法的优点是逻辑清晰，代码简单，缺点是牺牲了一个存储单元。例如，如果数组容量为 5，那么实际只能存储 4 个元素。

维护计数器法则是通过一个额外变量 `count` 来记录队列中的元素个数。判空时检查 `count == 0`，判满时检查 `count == capacity`。这种方法的优点是不浪费空间，但缺点是需要维护计数器，增加了操作的开销。

在本文的后续代码实现中，我们将采用浪费一个空间法，因为它更常见且易于理解。

### 3 代码实现

以下是一个基于 Python 的循环队列实现。我们定义一个类 CircularQueue，包含成员变量 queue（列表）、front、rear 和 capacity。

```
1 class CircularQueue:
2     def __init__(self, k):
3         self.capacity = k
4         self.queue = [None] * k # 初始化固定大小的数组
5         self.front = 0
6         self.rear = 0
7
8     def enqueue(self, value):
9         if self.is_full():
10             raise Exception("Queue is full")
11         self.queue[self.rear] = value
12         self.rear = (self.rear + 1) % self.capacity # 循环移动 rear 指针
13
14     def dequeue(self):
15         if self.is_empty():
16             raise Exception("Queue is empty")
17         item = self.queue[self.front]
18         self.front = (self.front + 1) % self.capacity # 循环移动 front 指针
19         return item
20
21     def get_front(self):
22         if self.is_empty():
23             return -1
24         return self.queue[self.front]
25
26     def get_rear(self):
27         if self.is_empty():
28             return -1
29         return self.queue[(self.rear - 1) % self.capacity] # 注意 rear 指向的是下一个空
30         → 位，所以取前一个位置
31
32     def is_empty(self):
33         return self.front == self.rear
```

```
35     def is_full(self):  
         return (self.rear + 1) % self.capacity == self.front
```

在初始化方法 `__init__` 中，我们创建了一个大小为  $k$  的数组，并初始化指针。入队方法 `enqueue` 首先检查队列是否已满，然后插入元素并更新 `rear` 指针。出队方法 `dequeue` 检查是否为空，然后取出元素并更新 `front` 指针。获取队头和队尾元素的方法中，需要注意 `rear` 指针指向的是下一个空位，因此获取队尾时需要使用  $(\text{rear} - 1) \% \text{capacity}$  来定位最后一个元素。判空和判满方法基于浪费一个空间法的条件实现。

## 4 循环队列的应用场景

循环队列在计算机系统中有着广泛的应用。在操作系统中，它用于进程调度和消息传递，确保任务按照顺序处理。在网络领域，循环队列作为数据包缓冲区，帮助实现流量控制，避免数据丢失。在多媒体应用中，如音乐播放器或视频流，循环队列用于管理播放列表或缓冲数据，提供流畅的用户体验。此外，在任何需要高效、有界的生产者-消费者模型场景中，循环队列都能发挥其优势，例如实时数据处理和事件队列管理。

循环队列的主要优点是高效利用固定大小的内存空间，解决了假溢出问题，并且所有操作的时间复杂度都是  $O(1)$ 。与普通数组队列相比，循环队列在空间利用上更高效，而普通队列可能因为假溢出浪费空间。在操作时间复杂度上，两者都是  $O(1)$ ，但循环队列的实现稍复杂，需要处理循环和边界条件。普通队列适用于无界或数据量未知的场景，而循环队列更适合有界、固定大小的缓冲区场景。

通过本文的讲解，希望读者能深入理解循环队列的核心思想，并动手实现它。这种数据结构不仅提升了效率，也体现了计算机科学中的精巧设计。鼓励读者在实际项目中应用循环队列，并探索其变种，如动态扩容版本。