基本的队列（Queue）数据结构

李睿远

Nov 07, 2025

从“先来后到”的哲学，到计算机科学的核心基石

想象一下，您在咖啡店点单时，前面已经排了几位顾客，您会自然地站到队尾等待。同样，打印机处理多个文档任务时，会按照接收顺序依次打印；客服热线中，来电者也会被放入等待队列，按先后顺序接听。这些场景都遵循一个共同原则 —— 先进先出（First-In-First-Out，FIFO），即先来的对象先被服务。在计算机科学中，这种“排队”思想被抽象为队列数据结构，用于处理需要顺序执行的任务。队列是许多系统的基础组件，从操作系统调度到网络数据包管理，都离不开它的身影。本文将带您从零开始，深入探索队列的概念、实现方式及其广泛应用，帮助您不仅理解“是什么”，更掌握“为什么”和“怎么做”。

1　队列的核心概念与特性

队列是一种操作受限的线性表，它只允许在表的前端进行删除操作，在表的后端进行插入操作。前端通常称为队首（front），后端称为队尾（rear）。队列的核心特性是 FIFO，即最先进入队列的元素将最先被移出。这与栈的后进先出（LIFO）特性形成鲜明对比，例如栈像电梯，最后进入的人最先出去；而队列像隧道，车辆按进入顺序依次通过。

队列的基本操作包括入队、出队、获取队首元素、检查是否为空以及返回大小。入队操作（常用方法名如 enqueue 或 offer）用于向队列尾部添加新元素；出队操作（如 dequeue 或 poll）移除并返回队列头部的元素；获取队首元素操作（如 front 或 peek）返回头部元素但不移除它；检查是否为空操作（isEmpty）判断队列是否没有元素；返回大小操作（size）给出元素个数。对于有界队列，还可以添加检查是否已满的操作（isFull）。这些操作共同定义了队列的行为，确保数据处理的顺序性。

2　队列的实现方式

队列可以通过多种底层数据结构实现，最常见的是基于数组和基于链表的方式。每种方式都有其优缺点，适用于不同场景。下面我们将详细探讨这两种实现，并提供代码示例和解读。

2.1　基于数组的实现（顺序队列）

基于数组的队列使用一个固定大小的数组和两个指针（front 和 rear）来跟踪队列的头部和尾部。初始时，front 和 rear 都指向数组起始位置。入队操作将元素添加到 rear 指向的位置，并后移 rear；出队操作返回 front 指向的元素，并前移 front。然而，这种简单实现会遇到“假溢出”问题：随着元素入队和出队，front 和 rear 指针不断后移，导致数组前半部分空间无法利用，即使数组未满，也无法添加新元素。

解决假溢出的方法是使用循环队列。循环队列将数组视为一个环形结构，当指针移动到数组末尾时，自动绕回开头。关键计算包括：入队时，rear 更新为 $rear = (rear + 1)\%capacity$；出队时，front 更新为 $front = (front + 1)\%capacity$。队列空的条件是 front 等于 rear；队列满的条件是 $(rear + 1)\%capacity == front$，这里牺牲一个存储单元以区分空和满状态。

以下是用 Python 实现循环队列的代码示例。我们定义一个 ArrayQueue 类，包含 items 数组、front 和 rear 指针以及容量 capacity。

```
class ArrayQueue:
    def __init__(self, capacity):
        self.capacity = capacity
        self.items = [None] * capacity
        self.front = 0
        self.rear = 0

    def enqueue(self, item):
        if self.is_full():
            raise Exception("Queue is full")
        self.items[self.rear] = item
        self.rear = (self.rear + 1) % self.capacity

    def dequeue(self):
        if self.is_empty():
            raise Exception("Queue is empty")
        item = self.items[self.front]
        self.front = (self.front + 1) % self.capacity
        return item

    def peek(self):
        if self.is_empty():
            return None
        return self.items[self.front]

    def is_empty(self):
        return self.front == self.rear

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

    def size(self):
        return (self.rear - self.front + self.capacity) % self.capacity
```

在这段代码中，构造函数初始化一个固定容量的数组，并将 front 和 rear 都设为 0。enqueue 方法首先检查队列是否已满，如果未满，则将元素放入 rear 位置，并更新 rear 指针使用模运算实现循环。dequeue 方法检查队列是否为空，如果不空，则返回 front 位置的元素，并更新 front 指针。peek 方法返回队首元素但不移除它。is_empty 和 is_full 方法分别通过比较 front 和 rear 来判断状态。size 方法计算当前元素个数，考虑循环情况。这种实现确保了所有基本操作的时间复杂度为 O(1)，但容量固定，可能不适合动态场景。

2.2 基于链表的实现（链式队列）

基于链表的队列使用节点来存储元素，每个节点包含数据域和指向下一个节点的指针。队列维护两个指针：head 指向链表头节点（队首），tail 指向链表尾节点（队尾）。入队操作在 tail 节点后添加新节点，并更新 tail；出队操作移除 head 节点，并更新 head。链表实现天然支持动态扩容，没有假溢出问题，因为节点可以随时分配。

以下是用 Python 实现链式队列的代码示例。首先定义 Node 类表示链表节点，然后定义 LinkedListQueue 类。

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedListQueue:
    def __init__(self):
        self.head = None
        self.tail = None
        self.count = 0

    def enqueue(self, item):
        new_node = Node(item)
        if self.is_empty():
            self.head = new_node
            self.tail = new_node
        else:
            self.tail.next = new_node
            self.tail = new_node
        self.count += 1

    def dequeue(self):
        if self.is_empty():
            raise Exception("Queue is empty")
        item = self.head.data
        self.head = self.head.next
```

```
        if self.head is None:
            self.tail = None
        self.count -= 1
        return item

    def peek(self):
        if self.is_empty():
            return None
        return self.head.data

    def is_empty(self):
        return self.head is None

    def size(self):
        return self.count
```

在这段代码中，Node 类包含 data 和 next 指针。LinkedListQueue 的构造函数初始化 head 和 tail 为 None，count 记录元素个数。enqueue 方法创建新节点，如果队列为空，则 head 和 tail 都指向新节点；否则将新节点链接到 tail 后，并更新 tail。dequeue 方法检查队列是否为空，如果不空，则返回 head 的数据，更新 head 到下一个节点，如果 head 变为 None，则 tail 也设为 None。peek 方法返回 head 的数据但不移除。is_empty 和 size 方法分别通过 head 和 count 判断状态。链表实现的所有操作时间复杂度也为 O(1)，且容量无限，但每个节点有额外指针开销。

2.3 两种实现方式的对比

基于数组和基于链表的队列实现各有优劣。在时间复杂度上，两种实现的入队、出队操作均为 O(1)，因为都只涉及指针更新。在空间复杂度上，数组实现使用固定容量，内存连续，访问效率高，但可能浪费空间或需要扩容；链表实现容量动态，无浪费，但每个节点有额外指针开销，内存不连续。数组实现适合已知最大大小的场景，例如嵌入式系统；链表实现适合大小变化频繁的应用，如任务调度。选择时需权衡内存使用和性能需求。

3 队列的变体与应用场景

队列不仅限于基本 FIFO 形式，还有多种变体适应不同需求。双端队列（Deque）允许在队列两端进行插入和删除操作，可以同时模拟栈和队列的行为，例如用于实现滑动窗口算法。优先队列（Priority Queue）出队顺序由元素优先级决定，而非入队顺序，通常用堆（Heap）实现，应用在任务调度和 Dijkstra 算法中，其中高优先级任务先处理。

阻塞队列（Blocking Queue）是一种同步工具，当队列为空时，获取操作阻塞线程直到有元素可用；当队列满时，插入操作阻塞直到空间空闲。这在生产者-消费者模型中非常有用，例如多线程环境下，生产者线程生成数据放入队列，消费者线程从队列取出数据，阻塞机制确保线程安全协调。这些变体扩展了队列的应用范围，使其成为操作系统、消息中间件和网络通信的核心组件。

4 实战案例：用队列解决“击鼓传花”游戏

“击鼓传花”游戏是一个经典问题，可以用队列优雅解决。问题描述：一群人围成一圈，传花同时计数，数到特定数字的人被淘汰，最后剩下的人获胜。解决方案利用队列的 FIFO 特性模拟传花过程。

首先，将所有玩家入队。然后循环模拟传花：将队首玩家出队并立刻入队，相当于安全传花一次；当计数达到指定值时，将当前队首玩家出队（淘汰），不再入队。重复直到队列只剩一人。

以下是用之前实现的 LinkedListQueue 解决此问题的代码示例。假设玩家列表为 [Alice, Bob, Charlie, Diana]，计数为 3。

```
def hot_potato(players, num):
    queue = LinkedListQueue()
    for player in players:
        queue.enqueue(player)
    while queue.size() > 1:
        for _ in range(num - 1):
            queue.enqueue(queue.dequeue())
        eliminated = queue.dequeue()
        print(f"淘汰␣:␣{eliminated}")
    return queue.dequeue()

winner = hot_potato(["Alice", "Bob", "Charlie", "Diana"], 3)
print(f"获胜者␣:␣{winner}")
```

在这段代码中，hot_potato 函数初始化队列并填入所有玩家。while 循环继续直到队列大小大于 1。内层 for 循环执行 num-1 次传花：每次将队首玩家出队并立刻入队，相当于传递花束。当内循环结束，当前队首玩家被淘汰（出队并不再入队）。最后返回剩余的获胜者。例如，初始队列为 Alice、Bob、Charlie、Diana，计数 3；第一次循环传递两次后，Charlie 被淘汰；重复过程直到剩一人。这展示了队列如何自然建模顺序处理问题。

队列作为一种基础数据结构，其 FIFO 特性确保了数据处理的公平性和顺序性。我们从核心概念出发，探讨了基于数组和链表的实现方式，数组实现通过循环队列解决假溢出，链表实现支持动态扩容。队列的变体如双端队列、优先队列和阻塞队列，扩展了其应用场景，从算法到系统设计无处不在。通过击鼓传花案例，我们看到了队列在解决实际问题中的实用性。掌握队列不仅有助于理解计算机科学基础，还能为构建高效系统奠定基石。

5 互动与思考题

思考题一：如何用栈来实现一个队列？这需要两个栈协作，一个用于入队，一个用于出队，通过元素转移模拟 FIFO 行为。思考题二：除了文中提到的应用，队列还常见于消息队列系统如 RabbitMQ、事件循环处理等场景。欢迎在评论区分享您的想法和代码实现，共同探讨数据结构的魅力。