双端队列（Deque）数据结构

叶家炜

Jul 01, 2025

双端队列（Deque，全称 Double-Ended Queue）是一种支持在两端高效进行插入和删除操作的线性数据结构。与传统队列严格的 FIFO（先进先出）规则和栈的 LIFO（后进先出）规则不同，Deque 融合了两者的特性，允许开发者根据需求自由选择操作端。这种灵活性使其成为解决特定问题的利器。
为什么需要 Deque？在实际开发中，诸多场景需要两端操作能力。例如实现撤销操作历史记录时，新操作从前端加入而旧操作从后端移除；滑动窗口算法中需要同时维护窗口两端的数据；工作窃取算法和多线程任务调度也依赖双端操作特性。Deque 的核心操作包括 addFront/addRear 插入、removeFront/removeRear 删除以及 peekFront/peekRear 查看操作，这些构成了其基本能力集。

1　双端队列的抽象行为与操作

理解 Deque 需要明确其操作定义与边界条件。前端插入 addFront(item) 和后端插入 addRear(item) 在队列满时需扩容；删除操作 removeFront() 和 removeRear() 在空队列时报错；辅助方法 isEmpty() 判断队列空状态，size() 返回元素数量。这些操作共同定义了 Deque 的抽象行为。
可视化理解操作流程：假设初始为空队列，执行 addFront(A) 后队列为「A」；接着 addRear(B) 形成「A ← → B」结构；执行 removeFront() 移除 A 剩下「B」；最后 removeRear() 移除 B 回归空队列。这种动态过程清晰展示了 Deque 的双端操作特性。

2　实现方案：双向链表与循环数组

2.1　双向链表实现方案

双向链表方案通过节点间的双向指针实现高效端操作。节点类设计包含数据域和前后指针：

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

队列主体维护头尾指针和大小计数器：

```
class LinkedListDeque:
   def __init__(self):
```

```
        self.front = None # 头指针指向首节点
        self.rear = None # 尾指针指向末节点
        self._size = 0
```

addFront 操作创建新节点并更新头指针：新节点 next 指向原头节点，原头节点 prev 指向新节点。时间复杂度稳定为 $O(1)$，无扩容开销。优势在于动态扩容灵活，代价是每个节点需额外存储两个指针，空间开销为 $O(n) + 2 \times n \times ptr_{size}$。

2.2 循环数组实现方案

循环数组方案使用固定容量数组，通过模运算实现逻辑循环：

```
class ArrayDeque:
    def __init__(self, capacity=10):
        self.capacity = max(1, capacity)
        self.items = [None] * self.capacity
        self.front = 0 # 指向队首元素索引
        self.rear = 0 # 指向队尾后第一个空位索引
        self.size = 0
```

核心在于下标的循环计算：index = (current + offset) % capacity。队列满判断依据为 (rear + 1) % capacity == front。均摊时间复杂度为 $O(1)$，但扩容时需 $O(n)$ 数据迁移。优势是内存连续访问高效，缺陷是扩容需数据搬移。

3 代码实现：循环数组详解

以下为循环数组实现的完整代码，含详细注释：

```
class ArrayDeque:
    def __init__(self, capacity=10):
        self.capacity = max(1, capacity) # 确保最小容量为 1
        self.items = [None] * self.capacity
        self.front = 0 # 指向第一个有效元素
        self.rear = 0 # 指向下一个插入位置
        self.size = 0 # 当前元素数量

    def _resize(self, new_cap):
        """扩容迁移数据，保持元素物理顺序"""
        new_items = [None] * new_cap
        # 按逻辑顺序复制元素：从 front 开始连续取 size 个
        for i in range(self.size):
            new_items[i] = self.items[(self.front + i) % self.capacity]
        self.items = new_items
```

```
        self.front = 0 # 重置 front 到新数组首
        self.rear = self.size # rear 指向最后一个元素后
        self.capacity = new_cap

    def addFront(self, item):
        """前端插入：front 逆时针移动"""
        if self.size == self.capacity:
            self._resize(2 * self.capacity) # 容量翻倍
        # 计算新 front 位置（循环左移）
        self.front = (self.front - 1) % self.capacity
        self.items[self.front] = item
        self.size += 1

    def addRear(self, item):
        """后端插入：直接写入 rear 位置"""
        if self.size == self.capacity:
            self._resize(2 * self.capacity)
        self.items[self.rear] = item
        self.rear = (self.rear + 1) % self.capacity
        self.size += 1

    def removeFront(self):
        if self.isEmpty():
            raise Exception("Deque is empty")
        item = self.items[self.front]
        self.front = (self.front + 1) % self.capacity # 顺时针移动
        self.size -= 1
        return item

    def removeRear(self):
        if self.isEmpty():
            raise Exception("Deque is empty")
        # rear 指向空位，需先回退到末元素
        self.rear = (self.rear - 1) % self.capacity
        item = self.items[self.rear]
        self.size -= 1
        return item
```

扩容函数 _resize 通过遍历原数组，按逻辑顺序（从 front 开始）复制元素到新数组，确保数据连续性。前端插入时 front 逆时针移动（索引减一），利用模运算处理越界；后端插入直接写入 rear 位置并顺时针移动。删

除操作需特别注意 removeRear 时 rear 指向空位，需先回退获取末元素。

4 复杂度与性能对比

两种实现方案的时间复杂度对比显著：

操作	双向链表	循环数组（均摊）
addFront	$O(1)$	$O(1)$
addRear	$O(1)$	$O(1)$
removeFront	$O(1)$	$O(1)$
removeRear	$O(1)$	$O(1)$

空间开销方面：双向链表需 $O(n)$ 基础空间加上 $2 \times n \times ptr_{size}$ 指针开销；循环数组仅需 $O(n)$ 连续空间但可能包含空闲位。选择依据明确：频繁动态伸缩场景用双向链表，已知最大容量时循环数组更优。

5 应用场景实战

5.1 滑动窗口最大值（LeetCode 239）

Deque 在此算法中维护单调递减序列：

```
deque = ArrayDeque()
result = []
for i, num in enumerate(nums):
    # 清除小于当前值的尾部元素
    while not deque.isEmpty() and num > nums[deque.peekRear()]:
        deque.removeRear()
    deque.addRear(i) # 存入当前索引
    # 移除移出窗口的头部元素
    if deque.peekFront() == i - k:
        deque.removeFront()
    # 记录窗口最大值
    if i >= k - 1:
        result.append(nums[deque.peekFront()])
```

Deque 头部始终存储当前窗口最大值索引。当新元素 $nums_i$ 加入时，循环移除尾部小于 $nums_i$ 的元素，确保队列单调递减。同时检测并移除超出窗口的头部元素。该实现时间复杂度优化至 $O(n)$。

5.2 多层级撤销操作

在支持多级撤销的编辑器中，Deque 可高效管理操作历史：

```
class UndoManager:
```

```
    def __init__(self, max_history=100):
        self.history = ArrayDeque(max_history)
        self.redo_stack = []

    def execute(self, command):
        command.execute()
        self.history.addFront(command) # 新操作前端插入
        self.redo_stack.clear()

    def undo(self):
        if not self.history.isEmpty():
            cmd = self.history.removeFront() # 移除最近操作
            cmd.undo()
            self.redo_stack.append(cmd) # 存入重做栈
```

新操作从 Deque 前端插入，撤销时移除前端操作。当历史记录达到容量上限时，最旧操作自动从后端移除。这种设计完美平衡了空间效率和操作时效性。

双端队列的核心价值在于双端操作的高效性与栈/队列特性的统一抽象。实现选择需权衡场景：小规模动态数据适用双向链表；大规模预知容量数据优选循环数组。延伸思考包括线程安全实现方案（如加锁或原子操作）和循环数组内存碎片优化策略（如间隙压缩算法）。

测试用例验证实现正确性：

```
def test_ArrayDeque():
    dq = ArrayDeque(3)
    dq.addRear(2) # 状态 : [2]
    dq.addFront(1) # 状态 : [1, 2]
    dq.addRear(3) # 状态 : [1, 2, 3] → 触发扩容
    assert dq.size == 3
    assert dq.removeFront() == 1 # 状态 : [2, 3]
    assert dq.removeRear() == 3 # 状态 : [2]
    assert not dq.isEmpty()
```

该用例覆盖基础操作、边界扩容和状态转换，确保实现符合预期。掌握 Deque 将显著提升开发者解决复杂问题的能力。