队列I

像堆栈一样，队列是一种线性结构，它遵循着执行操作的特定顺序。比如在开发中常用的线程池就是一个队列，当我们向固定大小的线程池中请求一个线程时，如果线程池中有空闲的资源，那么可以获得到到，但是如果没有空闲的资源，那么线程池的处理一般有两种；一种是非阻塞的处理方式，直接拒绝任务请求；另外一种是阻塞的方式，将请求排队，等到有空闲的线程时，取出排队的请求继续处理。当处理排队请求时，我们又希望公平地处理每一个请求，先进先服务。队列有基于链表与基于数组的实现方式。

1. 基于链表实现的队列可以实现一个无限排队的无界队列，但是可能导致过多请求排队，请求响应的时间过长，对于大部分资源有限又对响应时间敏感的系统，基于链表实现的无限排队的线程池是不适合的。
2. 基于数据实现的队列可以实现一个有界队列，队列的大小有限，所以线程池中的排队请求超过队列的大小时，接下来的请求会被拒绝，这种对于响应时间敏感的系统来说更加的适合；但是队列的大小设置是一个十分重要的因素，队列大导致请求过多，队列小导致无法充分利用系统资源，发挥最大性能。

基于数组方式实现队列（java）

```java
public class ArrayQueue {
    //表示队列的最大容量
    private int maxSize;
    //对列的头
    private int front;
    //队列的尾
    private int rear;
    //数组
    private int[] arr;

    /**
     * 构造器
     * @param arrMaxSize 队列的最大容量
     */
    public ArrayQueue(int arrMaxSize){
        this.maxSize = arrMaxSize;
        this.arr = new int[maxSize];
        //-1表示front的前一个位置
        this.front = -1;
        //-1表示rear的最后一个数据
        this.rear = -1;
    }

    /**
     *判断队列是否已装满
     * @return
     */
    public boolean isFull(){
        return  rear == maxSize-1;
```

```
    }

    /**
     * 判断队列是否为空
     * @return
     */
    public boolean isEmpty(){
        return front == rear;
    }

    /**
     * 向队列中添加数据
     * @param n 数据
     */
    public void addToQueue(int n){
        if(isFull()){
            System.out.println("队列已满，不能加入数据");
            return;
        }
        rear++;
        arr[rear] = n;
    }

    /**
     * 获得队列中的数据
     * @return
     */
    public int getData(){
        if(isEmpty()){
            throw new RuntimeException("队列为空，不能取出数据");
        }
        front++;
        return arr[front];
    }

    /**
     * 输出队列所有元素
     */
    public void Show(){
        if (isEmpty()){
            System.out.println("队列为空");
            return;
        }
        for(int i=0;i<arr.length;i++){
            System.out.printf("arr[%d]=%d\t",i,arr[i]);
            System.out.println();
        }
    }

    /**
     * 获取队列的头部数据
     * @return
     */
    public int headData(){
        if (isEmpty()){
            throw new RuntimeException("队列为空，不能取出数据");
        }
        return arr[front+1];
    }
```

基于数组实现队列(golang)

```golang
package ArrayQueue

import "fmt"

type ArrayQueue struct {
    q        []interface{}
    capacity int
    front    int
    rear     int
}

func NewArrayQueue(n int) *ArrayQueue {
    if n == 0 {
        return nil
    }
    return &ArrayQueue{make([]interface{}, n), n, 0, 0}
}

//判断是队列是否已满
func (this *ArrayQueue) IsFull() bool {
    if this.rear == this.capacity {
        return true
    }
    return false
}

//判断队列是否为空
func (this *ArrayQueue) IsEmpty() bool {
    if this.rear == this.front {
        return true
    }
    return false
}

//出队列操作
func (this *ArrayQueue) DeQueue() interface{} {
    if this.IsEmpty() {
        return nil
    }
    v := this.q[this.front]
    this.front++
    return v
}

//入队操作
func (this *ArrayQueue) EnQueue(v interface{}) bool {
    if this.IsFull() {
        return false
    }
    this.q[this.rear] = v
    this.rear++
    return true
}

//打印所有元素

func (this *ArrayQueue) Show() string {
    if this.IsEmpty() {
```

```
        return "queue is empty(⊙﹏⊙)"
    }
    result := "front"
    for i := this.front; i <= this.rear-1; i++ {
        result += fmt.Sprintf("<-%+v", this.q[i])
    }
    result += "<-rear"
    return result
}

```

循环队列

当使用一个数组实现队列，进行出队与入队操作之后，已使用过的位置不能进行重复使用，导致资源浪费，所以可以使用一个循环队列来保证充分利用资。循环队列是一种基于先进先出（FIFO）操作顺序的线性数据结构，将最后一个位置连接回第一个位置形成一个圆圈，它也被称作“环形缓冲区”。在环形队列中判断队列满了的条件有产生了变化，当(Rear+1)%MaxSize = Front时，此时的队列便不能再进行入队了。实际上Rear执行的地址并没有存储数据，因此循环队列会浪费掉一个位置。

基于数组实现循环队列（java）

```java
public class CircleQueue {
    //表示队列的最大容量
    private int maxSize;
    //对列的头
    private int front;
    //队列的尾
    private int rear;
    //数组
    private int[] arr;

    /**
     * 构造方法
     * @param arrMaxSize 数组的最大容量
     */
    public CircleQueue(int arrMaxSize){
        maxSize = arrMaxSize;
        arr = new int[arrMaxSize];
        rear = 0;
        front = 0;
    }
    /**
     *判断队列是否已装满
     * @return
     */
    public boolean isFull(){
        return  (rear+1)%maxSize == front;
    }

    /**
     * 判断队列是否为空
     * @return
     */
    public boolean isEmpty(){
```

```
        return front == rear;
    }

    /**
     * 想队列中加入数据
     * @param n
     * @return
     */
    public void addToQueue(int n){
        if (isFull()){
            System.out.println("队列已满，不能加入数据");
            return;
        }
        arr[rear] = n;
        //将rear后移，必须进行取模
        rear = (rear+1)%maxSize;
    }

    /**
     * 取出队列中的数据
     * @return
     */
    public int getData(){
        if(isEmpty()){
            throw new RuntimeException("队列为空，不能取出数据");
        }
        //这里需要分析出front是指向队列的第一个元素
        //1使用临时变量存储front的值
        //2先将front后移，再取模
        //3返回临时变量
        int temp = arr[front];
        front = (front+1)%maxSize;
        return  temp;
    }

    public void show(){
        if(isEmpty()){
            System.out.println("队列为空");
            return;
        }
        for(int i = front;i<front+Count();i++){
            System.out.printf("arr[%d] = %d\n",i%maxSize,arr[i%maxSize]);
        }
    }

    /**
     * 队列中的有效元素
     * @return
     */
    public int Count(){
        return (rear+maxSize-front)%maxSize;
    }

    /**
     * 获得头元素
     * @return
     */
    public int headDate(){
        if (isEmpty()){
            throw new RuntimeException("队列为空，不能取出数据");
        }
```

```
        return arr[front];
    }
```

基于数组实现循环队列（golang）

```
package CircleQueue

import "fmt"

type CircleQueue struct {
    q        []interface{}
    capacity int
    rear     int
    front    int
}

//实例化函数
func NewCircleQueue(n int) *CircleQueue {
    if n == 0 {
        return nil
    }
    return &CircleQueue{make([]interface{}, n), n, 0, 0}
}

//队列为空的条件 rear = front
func (this *CircleQueue) IsEmpty() bool {
    if this.rear == this.front {
        return true
    }
    return false
}

//队列已满 (rear+1)%capacity == front
func (this *CircleQueue) IsFull() bool {
    if ((this.rear + 1) % this.capacity) == this.front {
        return true
    }
    return false
}

//数据入队列
func (this *CircleQueue) EnQueue(v interface{}) bool {
    if this.IsFull() {
        return false
    }
    this.q[this.rear] = v
    this.rear = (this.rear + 1) % this.capacity
    return true
}

//数据出队列
func (this *CircleQueue) DeQueue() interface{} {
    if this.IsEmpty() {
        return false
    }
    v := this.q[this.front]
    this.front = (this.front + 1) % this.capacity
```

```
    return v
}

//展示队列
func (this *CircleQueue) Show() string {
    if this.IsEmpty() {
        return "queue is empty(⊙﹏⊙)"
    }
    result := "front"
    var i = this.front
    for true {
        result += fmt.Sprintf("<-%+v", this.q[i])
        i = (i + 1) % this.capacity
        if i == this.rear {
            break
        }
    }
    result += "<-rear"
    return result
}

```

参考资料：

数据结构与算法之美 -王争： https://time.geekbang.org/column/intro/126

geeksforgeeks：https://www.geeksforgeeks.org/queue-data-structure/