

# 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，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)

```
1 public class ArrayQueue {
2     //表示队列的最大容量
3     private int maxSize;
4     //对列的头
5     private int front;
6     //队列的尾
7     private int rear;
8     //数组
9     private int[] arr;
10
11     /**
12      * 构造器
13      * @param arrMaxSize 队列的最大容量
14      */
15     public ArrayQueue(int arrMaxSize){
16         this.maxSize = arrMaxSize;
17         this.arr = new int[maxSize];
18         //-1表示front的前一个位置
19         this.front = -1;
20         //-1表示rear的最后一个数据
21         this.rear = -1;
22     }
23
24     /**
25      *判断队列是否已装满
26      * @return
27      */
28     public boolean isFull(){
29         return rear == maxSize-1;
```

```
30     }
31
32     /**
33      * 判断队列是否为空
34      * @return
35      */
36     public boolean isEmpty(){
37         return front == rear;
38     }
39
40     /**
41      * 向队列中添加数据
42      * @param n 数据
43      */
44     public void addToQueue(int n){
45         if(isFull()){
46             System.out.println("队列已满，不能加入数据");
47             return;
48         }
49         rear++;
50         arr[rear] = n;
51     }
52
53     /**
54      * 获得队列中的数据
55      * @return
56      */
57     public int getData(){
58         if(isEmpty()){
59             throw new RuntimeException("队列为空，不能取出数据");
60         }
61         front++;
62         return arr[front];
63     }
64
65     /**
66      * 输出队列所有元素
67      */
68     public void Show(){
69         if (isEmpty()){
70             System.out.println("队列为空");
71             return;
72         }
73         for(int i=0;i<arr.length;i++){
74             System.out.printf("arr[%d]=%d\t",i,arr[i]);
75             System.out.println();
76         }
77     }
78
79     /**
80      * 获取队列的头部数据
81      * @return
82      */
83     public int headData(){
84         if (isEmpty()){
85             throw new RuntimeException("队列为空，不能取出数据");
86         }
87         return arr[front+1];
88     }
```

## 基于数组实现队列(golang)

```
1 package ArrayQueue
2
3 import "fmt"
4
5 type ArrayQueue struct {
6     q []interface{}
7     capacity int
8     front int
9     rear int
10 }
11
12 func NewArrayQueue(n int) *ArrayQueue {
13     if n == 0 {
14         return nil
15     }
16     return &ArrayQueue{make([]interface{}, n), n, 0, 0}
17 }
18
19 //判断是队列是否已满
20 func (this *ArrayQueue) IsFull() bool {
21     if this.rear == this.capacity {
22         return true
23     }
24     return false
25 }
26
27 //判断队列是否为空
28 func (this *ArrayQueue) IsEmpty() bool {
29     if this.rear == this.front {
30         return true
31     }
32     return false
33 }
34
35 //出队列操作
36 func (this *ArrayQueue) DeQueue() interface{} {
37     if this.IsEmpty() {
38         return nil
39     }
40     v := this.q[this.front]
41     this.front++
42     return v
43 }
44
45 //入队操作
46 func (this *ArrayQueue) EnQueue(v interface{}) bool {
47     if this.IsFull() {
48         return false
49     }
50     this.q[this.rear] = v
51     this.rear++
52     return true
53 }
54
55 //打印所有元素
56
57 func (this *ArrayQueue) Show() string {
58     if this.IsEmpty() {
```

```

59         return "queue is empty(⊙_⊙)"
60     }
61     result := "front"
62     for i := this.front; i <= this.rear-1; i++ {
63         result += fmt.Sprintf("<-%+v", this.q[i])
64     }
65     result += "<-rear"
66     return result
67 }
68

```

## 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源。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）

```

1  public class CircleQueue {
2      //表示队列的最大容量
3      private int maxSize;
4      //对列的头
5      private int front;
6      //队列的尾
7      private int rear;
8      //数组
9      private int[] arr;
10
11     /**
12      * 构造方法
13      * @param arrMaxSize 数组的最大容量
14      */
15     public CircleQueue(int arrMaxSize){
16         maxSize = arrMaxSize;
17         arr = new int[arrMaxSize];
18         rear = 0;
19         front = 0;
20     }
21     /**
22      *判断队列是否已装满
23      * @return
24      */
25     public boolean isFull(){
26         return (rear+1)%maxSize == front;
27     }
28
29     /**
30      * 判断队列是否为空
31      * @return
32      */
33     public boolean isEmpty(){

```

```

34         return front == rear;
35     }
36
37     /**
38      * 想队列中加入数据
39      * @param n
40      * @return
41      */
42     public void addToQueue(int n){
43         if (isFull()){
44             System.out.println("队列已满，不能加入数据");
45             return;
46         }
47         arr[rear] = n;
48         //将rear后移，必须进行取模
49         rear = (rear+1)%maxSize;
50     }
51
52     /**
53      * 取出队列中的数据
54      * @return
55      */
56     public int getData(){
57         if(isEmpty()){
58             throw new RuntimeException("队列为空，不能取出数据");
59         }
60         //这里需要分析出front是指向队列的第一个元素
61         //1使用临时变量存储front的值
62         //2先将front后移，再取模
63         //3返回临时变量
64         int temp = arr[front];
65         front = (front+1)%maxSize;
66         return temp;
67     }
68
69     public void show(){
70         if(isEmpty()){
71             System.out.println("队列为空");
72             return;
73         }
74         for(int i = front;i<front+Count();i++){
75             System.out.printf("arr[%d] = %d\n",i%maxSize,arr[i%maxSize]);
76         }
77     }
78
79     /**
80      * 队列中的有效元素
81      * @return
82      */
83     public int Count(){
84         return (rear+maxSize-front)%maxSize;
85     }
86
87     /**
88      * 获得头元素
89      * @return
90      */
91     public int headDate(){
92         if (isEmpty()){
93             throw new RuntimeException("队列为空，不能取出数据");
94         }

```

```
95     return arr[front];
96 }
```

## 基于数组实现循环队列 (golang)

```
1  package CircleQueue
2
3  import "fmt"
4
5  type CircleQueue struct {
6      q      []interface{}
7      capacity int
8      rear   int
9      front  int
10 }
11
12 //实例化函数
13 func NewCircleQueue(n int) *CircleQueue {
14     if n == 0 {
15         return nil
16     }
17     return &CircleQueue{make([]interface{}, n), n, 0, 0}
18 }
19
20 //队列为空的条件 rear = front
21 func (this *CircleQueue) IsEmpty() bool {
22     if this.rear == this.front {
23         return true
24     }
25     return false
26 }
27
28 //队列已满 (rear+1)%capacity == front
29 func (this *CircleQueue) IsFull() bool {
30     if ((this.rear + 1) % this.capacity) == this.front {
31         return true
32     }
33     return false
34 }
35
36 //数据入队列
37 func (this *CircleQueue) EnQueue(v interface{}) bool {
38     if this.IsFull() {
39         return false
40     }
41     this.q[this.rear] = v
42     this.rear = (this.rear + 1) % this.capacity
43     return true
44 }
45
46 //数据出队列
47 func (this *CircleQueue) DeQueue() interface{} {
48     if this.IsEmpty() {
49         return false
50     }
51     v := this.q[this.front]
52     this.front = (this.front + 1) % this.capacity
```

```

53     return v
54 }
55
56 //展示队列
57 func (this *CircleQueue) Show() string {
58     if this.IsEmpty() {
59         return "queue is empty(⊙__⊙)"
60     }
61     result := "front"
62     var i = this.front
63     for true {
64         result += fmt.Sprintf("<-%+v", this.q[i])
65         i = (i + 1) % this.capacity
66         if i == this.rear {
67             break
68         }
69     }
70     result += "<-rear"
71     return result
72 }
73

```

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/>