基于数组或链表实现Map

> 程 序 员 常 用 的 IDEA 插 件 ：
> https://github.com/silently9527/ToolsetIdeaPlugin
> 微信公众号：贝塔学Java

前言

JAVA中的Map主要就是将一个键和一个值联系起来。虽然JAVA中已经提供了很多Map的实现，为了学习并掌握常用的数据结构，从本篇开始我将自己实现Map的功能，本篇主要是通过数组和链表两种方式实现，之后提供二叉树，红黑树，散列表的版本实现。通过自己手写各个版本的Map实现，掌握每种数据结构的优缺点，可以在实际的工作中根据需要选择适合的Map。

Map API的定义

在开始之前，我们需要先定义出Map的接口定义，后续的版本都会基于此接口实现

```
public interface Map<K, V> {
    void put(K key, V value);

    V get(K key);

    void delete(K key);

    int size();

    Iterable<K> keys();

    default boolean contains(K key) {
        return get(key) != null;
    }

    default boolean isEmpty() {
        return size() == 0;
    }
}
```

这个接口是最简单的一个Map定义，相信这些方法对于java程序员来说不会陌生；

基于链表实现Map

1. 基于链表实现首先我们需要定义一个Node节点，表示我们需要存储的key、vlaue

```
class Node {
    K key;
    V value;
    Node next;
```

```
    public Node(K key, V value, Node next) {
        this.key = key;
        this.value = value;
        this.next = next;
    }

}
```

1. get方法的实现思路是遍历链表，然后比较每个Node中的key是否相等，如果相等就返回value，否则返回null

```
@Override
public V get(K key) {
    return searchNode(key).map(node -> node.value).orElse(null);
}

public Optional<Node> searchNode(K key) {
    for (Node node = root; node != null; node = node.next) {
        if (node.key.equals(key)) {
            return Optional.of(node);
        }
    }
    return Optional.empty();
}
```

1. put方法的实现思路也是遍历链表，然后比较每个Node的key值是否相等，如果相等那么覆盖掉value，如果未查找到有key相等的node，那么就新建一个Node放到链表的开头

```
@Override
public void put(K key, V value) {
    Optional<Node> optionalNode = searchNode(key);

    if (optionalNode.isPresent()) {
        optionalNode.get().value = value;
        return;
    }
    this.root = new Node(key, value, root);
}
```

1. delete方法实现同样也需要遍历链表，因为我们的是单向链表，删除某个节点有两种思路，第一种，在遍历链表的时候记录下当前节点的上一个节点，把上一个节点的next指向当前节点next;第二种，当遍历到需要删除的节点时，把需要删除节点的next的key、value完全复制到需要删除的节点，把next指针指向next.next，比如：first - > A -> B -> C -> D -> E -> F -> G -> NULL,要删除 C 节点，就把D节点完全复制到c中，然后C -> E，变相删除了C

```
@Override
public void delete(K key) {
// 第一种实现：
//        for (Node node = first, preNode = null; node != null; preNode = node, node = n
```

```
//            if (node.key.equals(key)) {
//                if (Objects.isNull(preNode)) {
//                    first = first.next;
//                } else {
//                    preNode.next = node.next;
//                }
//            }
//        }

// 第二中实现：
    for (Node node = first; node != null; node = node.next) {
        if (node.key.equals(key)) {
            Node next = node.next;
            node.key = next.key;
            node.value =next.value;
            node.next = next.next;
        }
    }
}
```

分析上面基于链表实现的map，每次的put、get、delete都需要遍历整个链表，非常的低效，无法处理大量的数据，时间复杂度为O(N)

基于数组实现Map

基于链表的实现非常低效，因为每次操作都需要遍历链表，假如我们的数据是有序的，那么查找的时候我们可以使用二分查找法，那么get方法会加快很多

为了体现出我们的Map是有序的，我们需要重新定义一个有序的Map

```
public interface SortedMap<K extends Comparable<K>, V> extends Map<K, V> {
    int rank(K key);
}
```

该定义要求key必须实现接口Comparable，rank方法如果key值存在就返回对应在数组中的下标，如果不存在就返回小于key键的数量

- 在基于数组的实现中，我们会定义两个数组变量分部存放keys、values；
- rank方法的实现：由于我们整个数组都是有序的，我们可以二分查找法（可以查看《老哥是时候来复习下数据结构与算法了》），如果存在就返回所在数组的下表，如果不存在就返回0

```
@Override
public int rank(K key) {
    int lo = 0, hi = size - 1;
    while (lo <= hi) {
        int mid = (hi - lo) / 2 + lo;
        int compare = key.compareTo(keys[mid]);
        if (compare > 0) {
            lo = mid + 1;
        } else if (compare < 0) {
```

```
            hi = mid - 1;
        } else {
            return mid;
        }
    }
    return lo;
}
```

- get方法实现：基于rank方法，判断返回的keys[index]与key进行比较，如果相等返回values[index]，不相等就返回null

```
@Override
public V get(K key) {
    int index = this.rank(key);
    if (index < size && key.compareTo(keys[index]) == 0) {
        return values[index];
    }
    return null;
}
```

- put方法实现：基于rank方法，判断返回的keys[index]与key进行比较，如果相等直接修改values[index]的值，如果不相等表示不存在该key，需要插入并且移动数组

```
@Override
public void put(K key, V value) {
    int index = this.rank(key);
    if (index < size && key.compareTo(keys[index]) == 0) {
        values[index] = value;
        return;
    }

    for (int j = size; j > index; j--) {
        this.keys[j] = this.keys[j--];
        this.values[j] = this.values[j--];
    }
    keys[index] = key;
    values[index] = value;
    size++;
}
```

- delete方法实现：通过rank方法判断该key是否存在，如果不存在就直接返回，如果存在需要移动数组

```
@Override
public void delete(K key) {
    int index = this.rank(key);
    if (Objects.isNull(keys[index]) || key.compareTo(keys[index]) != 0) {
        return;
    }
    for (int j = index; j < size - 1; j++) {
        keys[j] = keys[j + 1];
        values[j] = values[j + 1];
    }
```

```
        keys[size - 1] = null;
        values[size - 1] = null;
        size--;
    }
```

> 基于数组实现的Map，虽然get方法采用的二分查找法，很快O(logN)，但是在处理大量数据的情况下效率依然很低，因为put方法还是太慢；下篇我们将基于二叉树来实现Map，继续改进提升效率

文中所有源码已放入到了github仓库

https://github.com/silently9527/JavaCore

最后（点关注，不迷路）

文中或许会存在或多或少的不足、错误之处，有建议或者意见也非常欢迎大家在评论交流。

最后，**写作不易，请不要白嫖我哟**，希望朋友们可以**点赞评论关注**三连，因为这些就是我分享的全部动力来源🙏