## 如何设计实现一个支持“快照”功能的iterator？

我们再来看这样一个问题：如何实现一个支持“快照”功能的迭代器？这个问题算是对上一节课内容的延伸思考，为的是帮你加深对迭代器模式的理解，也是对你分析、解决问题的一种锻炼。你可以把它当作一个面试题或者练习题，在看我的讲解之前，先试一试自己能否顺利回答上来。

### 问题描述

我们先来介绍一下问题的背景：如何实现一个支持“快照”功能的迭代器模式？

理解这个问题最关键的是理解“快照”两个字。所谓“快照”，指我们为容器创建迭代器的时候，相当于给容器拍了一张快照（Snapshot）。之后即便我们增删容器中的元素，快照中的元素并不会做相应的改动。而迭代器遍历的对象是快照而非容器，这样就避免了在使用迭代器遍历的过程中，增删容器中的元素，导致的不可预期的结果或者报错。

接下来，我举一个例子来解释一下上面这段话。具体的代码如下所示。容器 list 中初始存储了 3、8、2 三个元素。尽管在创建迭代器 iter1 之后，容器 list 删除了元素 3，只剩下 8、2 两个元素，但是，通过 iter1 遍历的对象是快照，而非容器 list 本身。所以，遍历的结果仍然是 3、8、2。同理，iter2、iter3 也是在各自的快照上遍历，输出的结果如代码中注释所示。

List<Integer> list = new ArrayList<>();

list.add(3);

list.add(8);

list.add(2);

Iterator<Integer> iter1 = list.iterator();//snapshot: 3, 8, 2

list.remove(new Integer(2));//list：3, 8

Iterator<Integer> iter2 = list.iterator();//snapshot: 3, 8

list.remove(new Integer(3));//list：8

Iterator<Integer> iter3 = list.iterator();//snapshot: 3

// 输出结果：3 8 2

while (iter1.hasNext()) {

System.out.print(iter1.next() + " ");

}

System.out.println();

// 输出结果：3 8

while (iter2.hasNext()) {

System.out.print(iter1.next() + " ");

}

System.out.println();

// 输出结果：8

while (iter3.hasNext()) {

System.out.print(iter1.next() + " ");

}

System.out.println();

如果由你来实现上面的功能，你会如何来做呢？下面是针对这个功能需求的骨架代码，其中包含 ArrayList、SnapshotArrayIterator 两个类。对于这两个类，我只定义了必须的几个关键接口，完整的代码实现我并没有给出。你可以试着去完善一下，然后再看我下面的讲解。

public ArrayList<E> implements List<E> {

// TODO: 成员变量、私有函数等随便你定义

@Override

public void add(E obj) {

//TODO: 由你来完善

}

@Override

public void remove(E obj) {

// TODO: 由你来完善

}

@Override

public Iterator<E> iterator() {

return new SnapshotArrayIterator(this);

}

}

public class SnapshotArrayIterator<E> implements Iterator<E> {

// TODO: 成员变量、私有函数等随便你定义

@Override

public boolean hasNext() {

// TODO: 由你来完善

}

@Override

public E next() {//返回当前元素，并且游标后移一位

// TODO: 由你来完善

}

}

### 解决方案一

我们先来看最简单的一种解决办法。在迭代器类中定义一个成员变量 snapshot 来存储快照。每当创建迭代器的时候，都拷贝一份容器中的元素到快照中，后续的遍历操作都基于这个迭代器自己持有的快照来进行。具体的代码实现如下所示：

public class SnapshotArrayIterator<E> implements Iterator<E> {

private int cursor;

private ArrayList<E> snapshot;

public SnapshotArrayIterator(ArrayList<E> arrayList) {

this.cursor = 0;

this.snapshot = new ArrayList<>();

this.snapshot.addAll(arrayList);

}

@Override

public boolean hasNext() {

return cursor < snapshot.size();

}

@Override

public E next() {

E currentItem = snapshot.get(cursor);

cursor++;

return currentItem;

}

}

这个解决方案虽然简单，但代价也有点高。每次创建迭代器的时候，都要拷贝一份数据到快照中，会增加内存的消耗。如果一个容器同时有多个迭代器在遍历元素，就会导致数据在内存中重复存储多份。不过，庆幸的是，Java 中的拷贝属于浅拷贝，也就是说，容器中的对象并非真的拷贝了多份，而只是拷贝了对象的引用而已。

那有没有什么方法，既可以支持快照，又不需要拷贝容器呢？

### 解决方案二

我们可以在容器中，为每个元素保存两个时间戳，一个是添加时间戳 addTimestamp，一个是删除时间戳 delTimestamp。当元素被加入到集合中的时候，我们将 addTimestamp 设置为当前时间，将 delTimestamp 设置成最大长整型值（Long.MAX\_VALUE）。当元素被删除时，我们将 delTimestamp 更新为当前时间，表示已经被删除。

注意，这里只是标记删除，而非真正将它从容器中删除。

同时，每个迭代器也保存一个迭代器创建时间戳 snapshotTimestamp，也就是迭代器对应的快照的创建时间戳。当使用迭代器来遍历容器的时候，只有满足 addTimestamp<snapshotTimestamp<delTimestamp 的元素，才是属于这个迭代器的快照。

如果元素的 addTimestamp>snapshotTimestamp，说明元素在创建了迭代器之后才加入的，不属于这个迭代器的快照；如果元素的 delTimestamp<snapshotTimestamp，说明元素在创建迭代器之前就被删除掉了，也不属于这个迭代器的快照。

这样就在不拷贝容器的情况下，在容器本身上借助时间戳实现了快照功能。具体的代码实现如下所示。注意，我们没有考虑 ArrayList 的扩容问题，感兴趣的话，你可以自己完善一下。

public class ArrayList<E> implements List<E> {

private static final int DEFAULT\_CAPACITY = 10;

private int actualSize; //不包含标记删除元素

private int totalSize; //包含标记删除元素

private Object[] elements;

private long[] addTimestamps;

private long[] delTimestamps;

public ArrayList() {

this.elements = new Object[DEFAULT\_CAPACITY];

this.addTimestamps = new long[DEFAULT\_CAPACITY];

this.delTimestamps = new long[DEFAULT\_CAPACITY];

this.totalSize = 0;

this.actualSize = 0;

}

@Override

public void add(E obj) {

elements[totalSize] = obj;

addTimestamps[totalSize] = System.currentTimeMillis();

delTimestamps[totalSize] = Long.MAX\_VALUE;

totalSize++;

actualSize++;

}

@Override

public void remove(E obj) {

for (int i = 0; i < totalSize; ++i) {

if (elements[i].equals(obj)) {

delTimestamps[i] = System.currentTimeMillis();

actualSize--;

}

}

}

public int actualSize() {

return this.actualSize;

}

public int totalSize() {

return this.totalSize;

}

public E get(int i) {

if (i >= totalSize) {

throw new IndexOutOfBoundsException();

}

return (E)elements[i];

}

public long getAddTimestamp(int i) {

if (i >= totalSize) {

throw new IndexOutOfBoundsException();

}

return addTimestamps[i];

}

public long getDelTimestamp(int i) {

if (i >= totalSize) {

throw new IndexOutOfBoundsException();

}

return delTimestamps[i];

}

}

public class SnapshotArrayIterator<E> implements Iterator<E> {

private long snapshotTimestamp;

private int cursorInAll; // 在整个容器中的下标，而非快照中的下标

private int leftCount; // 快照中还有几个元素未被遍历

private ArrayList<E> arrayList;

public SnapshotArrayIterator(ArrayList<E> arrayList) {

this.snapshotTimestamp = System.currentTimeMillis();

this.cursorInAll = 0;

this.leftCount = arrayList.actualSize();;

this.arrayList = arrayList;

justNext(); // 先跳到这个迭代器快照的第一个元素

}

@Override

public boolean hasNext() {

return this.leftCount >= 0; // 注意是>=, 而非>

}

@Override

public E next() {

E currentItem = arrayList.get(cursorInAll);

justNext();

return currentItem;

}

private void justNext() {

while (cursorInAll < arrayList.totalSize()) {

long addTimestamp = arrayList.getAddTimestamp(cursorInAll);

long delTimestamp = arrayList.getDelTimestamp(cursorInAll);

if (snapshotTimestamp > addTimestamp && snapshotTimestamp < delTimestamp) {

leftCount--;

break;

}

cursorInAll++;

}

}

}

实际上，上面的解决方案相当于解决了一个问题，又引入了另外一个问题。ArrayList 底层依赖数组这种数据结构，原本可以支持快速的随机访问，在 O(1) 时间复杂度内获取下标为 i 的元素，但现在，删除数据并非真正的删除，只是通过时间戳来标记删除，这就导致无法支持按照下标快速随机访问了。

现在，我们来看怎么解决这个问题：让容器既支持快照遍历，又支持随机访问？

解决的方法也不难，我稍微提示一下。我们可以在 ArrayList 中存储两个数组。一个支持标记删除的，用来实现快照遍历功能；一个不支持标记删除的（也就是将要删除的数据直接从数组中移除），用来支持随机访问。对应的代码我这里就不给出了，感兴趣的话你可以自己实现一下。