67 | 迭代器模式（下）：如何设计实现一个支持“快照”功能的iterator?

王争 2020-04-06

00:00 09:21

讲述：冯永吉 大小：8.57M

上两节课，我们学习了迭代器模式的原理、实现，并且分析了在遍历集合的同时增删集合元素，产生不可预期结果的原因以及应对策略。

今天，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。

话不多说，让我们正式开始今天的学习吧！

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。关于深拷贝、浅拷贝，我们在第 47 讲中有详细的讲解，你可以回过头去再看一下。

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

我们再来看第二种解决方案。

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。如果你对数组随机访问这块知识点不了解，可以去看我的《数据结构与算法之美》专栏，这里我就不展开讲解了。

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。

重点回顾

好了，今天的内容到此就讲完了。我们一块来总结回顾一下，你需要重点掌握的内容。

今天我们讲了如何实现一个支持“快照”功能的迭代器。其实这个问题本身并不是学习的重点，因为在真实的项目开发中，我们几乎不会遇到这样的需求。所以，基于今天的内容我不想做过多的总结。我想和你说一说，为什么我要来讲今天的内容呢？

实际上，学习本节课的内容，如果你只是从前往后看一遍，看懂就觉得 ok 了，那收获几乎是零。一个好学习方法是，把它当作一个思考题或者面试题，在看我的讲解之前，自己主动思考如何解决，并且把解决方案用代码实现一遍，然后再来看跟我的讲解有哪些区别。这个过程对你分析问题、解决问题的能力的锻炼，代码设计能力、编码能力的锻炼，才是最有价值的，才是我们这篇文章的意义所在。所谓“知识是死的，能力才是活的”就是这个道理。

其实，不仅仅是这一节的内容，整个专栏的学习都是这样的。

在《数据结构与算法之美》专栏中，有同学曾经对我说，他看了很多遍我的专栏，几乎看懂了所有的内容，他觉得都掌握了，但是，在最近第一次面试中，面试官给他出了一个结合实际开发的算法题，他还是没有思路，当时脑子一片放空，问我学完这个专栏之后，要想应付算法面试，还要学哪些东西，有没有推荐的书籍。

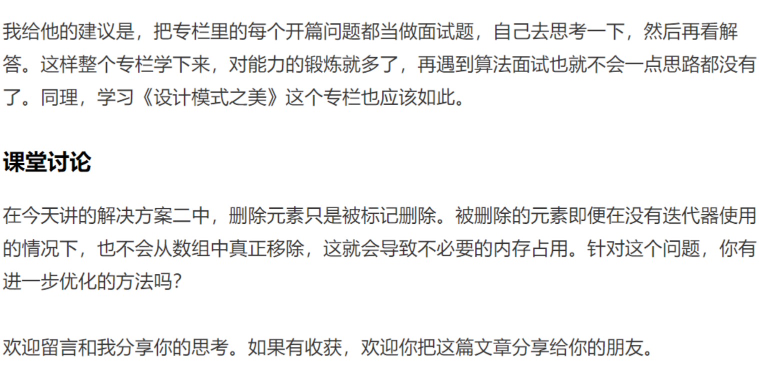
我看了他的面试题之后发现，用我专栏里讲的知识是完全可以解决的，而且，专栏里已经讲过类似的问题，只是换了个业务背景而已。之所以他没法回答上来，还是没有将知识转化成解决问题的能力，因为他只是被动地“看”，从来没有主动地“思考”。**只掌握了知识，没锻炼能力，遇到实际的问题还是没法自己去分析、思考、解决。**

我给他的建议是，把专栏里的每个开篇问题都当做面试题，自己去思考一下，然后再看解答。这样整个专栏学下来，对能力的锻炼就多了，再遇到算法面试也就不会一点思路都没有了。同理，学习《设计模式之美》这个专栏也应该如此。

课堂讨论

在今天讲的解决方案二中，删除元素只是被标记删除。被删除的元素即便在没有迭代器使用的情况下，也不会从数组中真正移除，这就会导致不必要的内存占用。针对这个问题，你有进一步优化的方法吗？

欢迎留言和我分享你的思考。如果有收获，欢迎你把这篇文章分享给你的朋友。



志恒Z

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。

Ctrl + Enter 发表 0/2000字 提交留言

精选留言(32)

LJK

思考题感觉像是数据库的MVCC？
- 容器中维护一个每个迭代器创建时间的列表
- 每次有迭代器创建时就在这个列表中加入自己的创建时间
- 迭代器迭代完成后将列表中对应时间点删除
- 清理容器时，对于容器中每个元素，如果addTime小于这个列表中的最小时间点就可以进行删除

2020-04-06 4 8

辣么大