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

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。

精选留言：

- 辣么大 2020-04-06 16:28:00
  课后思考题：类似数组动态扩容和缩容，删除元素时可以比较被删除元素的总数，在被删除元素总数 < 总数的 1/2 时， 进行resize数组，清空被删除的元素，回收空间。 [2赞]

- Monday 2020-04-06 10:22:28
  在阅读本节代码实现就想到了第二种方案存在类似思考题的问题
  解决方案可以在合适的时候清理带删除标记的元素。本想使用数据库的多版本控制（MVCC)的方案，把所有的迭代器对象存起来，并添加创建时间。但是冒出一个新问题，数据库事务有commit来表示事务已完成，而迭代器的使用完成无法知晓，还在思考方案中…… [2赞]

- LJK 2020-04-06 06:51:41
  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 [2赞]

- 辉哥 2020-04-07 10:05:48
  课堂讨论：可以创建一个object类型的常量，删除数组元素时，可以将被删除数组元素的引用指向该常量。Android中的SparseArray就是采用此方式实现的

- 马以 2020-04-06 22:59:32
  记录一个迭代变量，每迭代一次，计数加一，迭代完一次减一，当为0的时候就可以删除标记为delete的元素了

- 马以 2020-04-06 22:52:19
  是的，这个是和数据库的事务隔离差不多，老师这里用的是时间戳，我们还可以利用一个自增的序列号来控制，都是一样的；

- Ken张云忠 2020-04-06 21:31:30
  在集合增加一个数组field，专门用来记录引用该元素的迭代器个数，当迭代器个数为0且该元素已经标记为删除元素时才真正的删除元素，当需要迭代器使用者在使用完迭代器后需要显示得调用迭代器注销该元素的函数，对于使用者不太友好了。

- webmin 2020-04-06 20:48:53
  可以参考GC的算法，弄一个减化版的优化方法：
  1. 被删除的元素是否还有可能被已经创建的iterator所访问，即被删除的元素还被引用着；（iterator使用完需要有调用关闭动作）
  2. 被删除的元素达到一定量时，按照删除时间清理掉效早删除的元素，清理掉的最晚的被删除元素的删除时间放置在清理标识字段，iterator迭代时检查清理标识字段，如果iterator创建时间早于清理标识字段中的时间丢出异常；

- 筱乐乐哦 2020-04-06 18:52:21
  老师，有个问题：你在文章中说到：让容器既支持快照遍历，又支持随机访问？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
  如果是这样操作，那和浅拷贝的那个比较，没发现有优势呀，老师可以说下这块吗

- Jxin 2020-04-06 18:22:26
  1.给元素打时间截实现快照，空间成本会比较大。这里其实采用复制状态机一样能起到效果，只是空间成本就变成了时间成本。

  2.至于栏主的课后题，这已经是从实现快照，变成快照操作在多线程可见了。那么当前的实现是不严谨的，并发会有数据不符合预期的情况。

  3.不考虑并发问题，仅看如何释放内存这个问题。复制状态机可以将一段时间的log整合，实现快照往前移动（比如redis）。放在这里也一样，定时对元素做整合，将被删除的元素移除即可。（遗憾的时，基于时间截这种，无法在某个快照（状态），结合log，做往后倒退的操作）

- 子豪sirius 2020-04-06 17:24:21
  第二个问题，我想可不可用个数组记录当前有多少个迭代器。每调用一次iterrator方法，迭代器加一；有元素删除时，记录这个时间点的迭代器的数量；当迭代器访问到该元素时，减一，减到0，说明不再有删除该元素时间点之前生成的迭代器来访问了，就可以实际删除该元素。

- 徐凯 2020-04-06 17:01:11
  容器可以放指向元素的指针，当不用时直接将指针释放并置为空

- zj 2020-04-06 16:35:26
  我有个想法，当迭代器创建后，容器元素如果被删除了，则在迭代器创建强引用指向这个容器元素，容器中元素将之前元素的强引用变为弱引用。

当迭代器不再使用后，会被gc掉，从而删除的元素只剩下弱引用了，那下一次gc，这个删除的元素就会被gc掉。

- Jemmy 2020-04-06 16:14:49
  Iterator析构函数中调中ArrayList的清理函数，清理函数并不是每次都清理，设置一个loadFactory，当已删除元素超过这个loadFactory时之行清理

- hanazawakana 2020-04-06 11:25:12
  迭代器结束循环后，删除快照数据。开始创建列表时，再创建一个新的快照数据

- 小晏子 2020-04-06 09:53:58
  对于课后思考题的基本想法就是真正删除那些已经被标记删除的元素，只有这样才能节省内存占用，主要的问题是什么时候删除元素？因为是快照，那么假设用户在某段时间内是需要这个快照做一些事情的，所以就需要快照保留一段时间，不能立即清理元素，所以可以在方案二的实现中增加快照保存时间的时间戳，然后开个线程定时扫描所有元素的删除时间戳，和这个快照保存时间戳进行对比，当当前时间-元素的删除时间>快照保存时间的时候，就清理掉这个元素。

- test 2020-04-06 09:45:19
  课后思考题：定期进行清除

- Lee 2020-04-06 09:16:29
  课后问题可以在构建SnapshotArrayIterator的时候，将ArrayList中删除时间在当前时间之前的已经标注删除的记录清理一次，并不会影响快照实际有效内容。

- Geek_27a248 2020-04-06 08:23:44
  如果有两个数组保存数据，一个有标记时间戳的，一个没有标记时间戳来支持随机访问的，那Arraylist真正的删除数据的方法也是采用的这种吗，这三篇文章看的有点迷糊了，待我从头在慢慢的看来实验一遍。。本篇文章最大的收获是又坚定了我每篇文章都写代码实验一下的决心

- eason2017 2020-04-06 07:12:56
  定时清理里面的数据，做一次同步。期间可能会加锁来保证数据的有效性。