LinkedList实现原理分析（Java源码剖析）

六尺帐篷 (/u/f8e9b1c246f1)
2017.08.06 12:28* 字数 1608 阅读 335 评论 0 喜欢 24
(/u/f8e9b1c246f1)

编辑文章 (/writer#/notebooks/15099409/notes/15450868)

> - 本文对LinkedList的实现讨论都基于JDK8版本

Java中的LinkedList类实现了List接口和Deque接口，是一种链表类型的数据结构，支持高效的插入和删除操作，同时也实现了Deque接口，使得LinkedList类也具有队列的特性。LinkedList类的底层实现的数据结构是一个双端的链表。

LinkedList类中有一个内部私有类Node，这个类就代表双端链表的节点Node。这个类有三个属性，分别是前驱节点，本节点的值，后继结点。
源码中的实现是这样的。

```
private static class Node<E> {
        E item;
        Node<E> next;
        Node<E> prev;

        Node(Node<E> prev, E element, Node<E> next) {
            this.item = element;
            this.next = next;
            this.prev = prev;
        }
    }
```

注意这个节点的初始化方法，给定三个参数，分别前驱节点，本节点的值，后继结点。这个方法将在LinkedList的实现中多次调用。

下图是LinkedList内部结构的可视化，能够帮我们更好的理解LinkedList内部的结构。

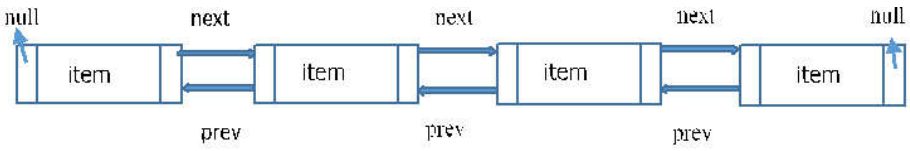

image.png

双端链表由node组成，每个节点有两个reference指向前驱节点和后继结点，第一个节点的前驱节点为null，最后一个节点的后继节点为null。

LinkedList类有很多方法供我们调用。我们不会一一介绍，本文会详细介绍其中几个最核心最基本的方法，LinkedList的创建添加和删除基本都和这几个操作有关。

- linkFirst() method
  首先我们介绍第一个方法，linkFirst（），顾名思义，这个方法是插入第一个节点，我们先直接上代码，看看它的具体实现

```
/**
     * Links e as first element.
     */
    private void linkFirst(E e) {
        final Node<E> f = first;
        final Node<E> newNode = new Node<>(null, e, f);
        first = newNode;
        if (f == null)
            last = newNode;
        else
            f.prev = newNode;
        size++;
        modCount++;
    }
```

我们发现出现了两个变量，first和last这两个变量是LinkedList的成员变量，分别指向头结点和尾节点。他们是如下定义的：

```
/**
     * Pointer to first node.
     * Invariant: (first == null && last == null) ||
     *            (first.prev == null && first.item != null)
     */
    transient Node<E> first;
```

```
/**
     * Pointer to last node.
     * Invariant: (first == null && last == null) ||
     *            (last.next == null && last.item != null)
     */
    transient Node<E> last;
```

我们可以看到注释中的内容。first和last需要维持一个不变量，也就是first和last始终都要维持两种状态：
首先，如果双端链表为空的时候，两个都必须为null
如果链表不为空，那么first的前驱节点一定是null，first的item一定不为null，同理，last的后继节点一定是null，last的item一定不为null。

知道了first和last之后，我们就可以开始分析linkFirst的代码了。
linkFirst的作用就是在first节点的前面插入一个节点，插入完之后，还要更新first节点为新插入的节点，并且同时维持last节点的不变量。

我们开始分析代码，首先用f来临时保存未插入前的first节点，然后调用的node的构造函数新建一个值为e的新节点，这个节点插入之后将作为first节点，所以新节点的前驱节点为null，值为e，后继节点是f,也就是未插入前的first节点。
然后就是维持不变量，首先第一种情况，如果f==null，那就说明插入之前，链表是空的，那么新插入的节点不仅是first节点还是last节点，所以我们要更新last节点的状态，也就是last现在要指向新插入的newNode。
如果f!=null那么就说明last节点不变，但是要更新f的前驱节点为newNode，维持first节点的不变量。
最后size加一就完成了操作。

- linkLast() method
  分析了linkFirst方法，对于 linkLast()的代码就很容易理解了，只不过是变成了插入到last节点的后面。我们直接看代码

```
/**
     * Links e as last element.
     */
    void linkLast(E e) {
        final Node<E> l = last;
        final Node<E> newNode = new Node<>(l, e, null);
        last = newNode;
        if (l == null)
            first = newNode;
        else
            l.next = newNode;
        size++;
        modCount++;
    }
```

到这里我们发现有这个两个方法，我们已经可以实现一个简单队列的插入操作，上面两个方法就可以理解为插入队头元素和队尾元素，这也说明了LinkedList是实现了Deque接口的。
从源码中也可以看出，addfirst和addLast这两个方法内部就是直接调用了linkFirst和LinkLast

```
/**
     * Inserts the specified element at the beginning of this list.
     *
     * @param e the element to add
     */
    public void addFirst(E e) {
        linkFirst(e);
    }

    /**
     * Appends the specified element to the end of this list.
     *
     * <p>This method is equivalent to {@link #add}.
     *
     * @param e the element to add
     */
    public void addLast(E e) {
        linkLast(e);
    }
```

- linkBefore(E e, Node<E> succ)
  下面我们看一个linkBefore方法,从名字可以看出这个方法是在给定的节点前插入一个节点，可以说是linkFirst和linkLast方法的通用版。

```
/**
     * Inserts element e before non-null Node succ.
     */
    void linkBefore(E e, Node<E> succ) {
        // assert succ != null;
        final Node<E> pred = succ.prev;
        final Node<E> newNode = new Node<>(pred, e, succ);
        succ.prev = newNode;
        if (pred == null)
            first = newNode;
        else
            pred.next = newNode;
        size++;
        modCount++;
    }
```

我们可以看到代码的实现原理基本和前面的两个方法一致，这里是假设插入的这个节点的位置是非空的。

- add(int index, E element)
  下面我们看add方法，这个方法就是最常用的，在指定下标插入一个节点。我们先来看下源码的实现，很简单

```
/**
     * Inserts the specified element at the specified position in this list.
     * Shifts the element currently at that position (if any) and any
     * subsequent elements to the right (adds one to their indices).
     *
     * @param index index at which the specified element is to be inserted
     * @param element element to be inserted
     * @throws IndexOutOfBoundsException {@inheritDoc}
     */
    public void add(int index, E element) {
        checkPositionIndex(index);

        if (index == size)
            linkLast(element);
        else
            linkBefore(element, node(index));
    }
```

首先判断给定的index是不是合法的，然后如果index==size，就说明要插入成为最后一个节点，直接调用linklast方法，否则就调用linkBefore方法，我们知道linkBefore需要给定两个参数，一个插入节点的值，一个指定的node，所以我们又调用了Node(index)去找到index的那个node。
我们看一下Node<E> node(int index)方法，这个方法就是找到给定index的node并返回，类似于数组的随机读取，但由于这里是链表，所以要进行查找

```
/**
     * Returns the (non-null) Node at the specified element index.
     */
    Node<E> node(int index) {
        // assert isElementIndex(index);

        if (index < (size >> 1)) {
            Node<E> x = first;
            for (int i = 0; i < index; i++)
                x = x.next;
            return x;
        } else {
            Node<E> x = last;
            for (int i = size - 1; i > index; i--)
                x = x.prev;
            return x;
        }
    }
```

我们看到node的实现并不是像我们想象的那样直接就线性从头查找，而是折半查找，有一个小优化，先判断index在前半段还是后半段，如果在前半段就从头开始找，如果在后半段就从后开始找，这样最坏情况也只要找一半就可以了。

LinkedList的源码实现并不复杂，我们只介绍这几个方法，相信你一定对于它的内部实现原理有了一定的了解，并且也学习到了优秀的代码书写风格和优化。
对于remove操作，有兴趣的读者可以自行研究代码，它类似于add操作，也是基于三个基本方法来实现的。

- unlinkFirst(Node<E> f)
- unlinkLast(Node<E> l)
- unlink(Node<E> x)

Java Collection实现原理分析 (/nb/15099409)

六尺帐篷 (/u/f8e9b1c246f1)
写了 245418 字，被 15240 人关注，获得了 1429 个喜欢
(/u/f8e9b1c246f1)