在第 16 行到第 18 行我们将以向后的方向遍历该链表。否则，我们从终端开始向回走，如图中第 22 行到第 24 行所示。

如图 3-32 所示，LinkedListIterator 具有类似于 ArrayListIterator 的逻辑，但合并了重要的错误检测。该迭代器保留一个当前位置，如第 3 行所示。current 表示包含由调用 next 所返回的项的节点。注意，当 current 被定位于 endMarker 时，对 next 的调用是非法的。

```
1       private class LinkedListIterator implements java.util.Iterator<AnyType>
2       {
3           private Node<AnyType> current = beginMarker.next;
4           private int expectedModCount = modCount;
5           private boolean okToRemove = false;
6
7           public boolean hasNext( )
8             { return current != endMarker; }
9
10          public AnyType next( )
11          {
12              if( modCount != expectedModCount )
13                  throw new java.util.ConcurrentModificationException( );
14              if( !hasNext( ) )
15                  throw new java.util.NoSuchElementException( );
16
17              AnyType nextItem = current.data;
18              current = current.next;
19              okToRemove = true;
20              return nextItem;
21          }
22
23          public void remove( )
24          {
25              if( modCount != expectedModCount )
26                  throw new java.util.ConcurrentModificationException( );
27              if( !okToRemove )
28                  throw new IllegalStateException( );
29
30              MyLinkedList. this .remove( current.prev );
31              okToRemove = false;
32              expectedModCount++;
33          }
34      }
```

图 3-32　MyLinkedList 类的内部 Iterator 类

为了检测在迭代期间集合被修改的情况，迭代器在第 4 行将迭代器被构造时的链表的 modCount 存储在数据域 expectedModCount 中。在第 5 行，如果 next 已经被执行而没有其后的 remove，则布尔数据域 okToRemove 为 true。因此，okToRemove 初始为 false，在 next 方法中置为 true，在 remove 方法中置为 false。

hasNext 是一个简单的例程。和在 java.util.LinkedList 的迭代器中一样，它不检查链表的修改。

next 方法在获得(第 17 行)将要返回(第 20 行)的节点的值后向后推进 current(第 18 行)。okToRemove 在第 19 行被更新。