头节点，则第 1 个节点前面没有节点)。图 3-22 显示一个带有头节点和尾节点的双链表。图 3-23 显示一个空链表。图 3-24 则显示 MyLinkedList 类的概要和部分的实现。

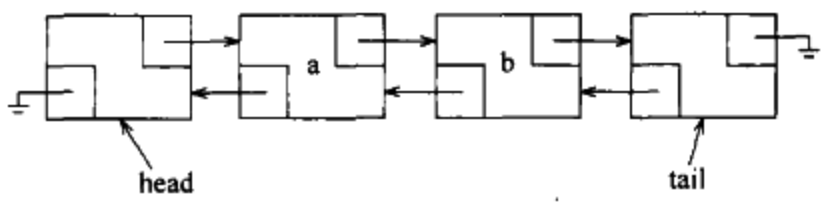

图 3-22　具有头节点和尾节点的双链表

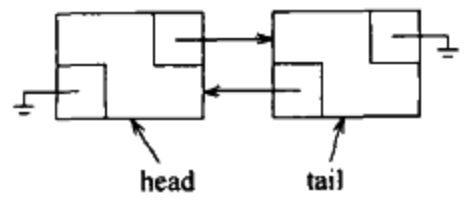

图 3-23　具有头节点和尾节点的空链表

我们在第 3 行可以看到私有嵌套 Node 类声明的开头部分。图 3-25 显示这个由所存储的一项组成的 Node 类——它的连接到前一个 Node 的链和下一个 Node 的链，还有一个构造方法。所有的数据成员都是公用的。我们知道，在一个类中，数据成员通常是私有的。然而，在一个嵌套类中的成员甚至在外部类中也是可见的。由于 Node 类是私有的，因此在 Node 类中的那些数据成员的可见性是无关紧要的；那些 MyLinkedList 的方法能够见到所有的 Node 数据成员，而 MyLinkedList 外面的那些类则根本见不到 Node 类。

现在回到图 3-24，第 44 行到第 47 行包含 MyLinkedList 的数据成员，即到头节点和到尾节点的引用。我们也掌握一个数据成员的大小，从而 size 方法可以以常数时间实现。在第 45 行有一个附加的数据域，用来帮助迭代器检测集合中的变化。modCount 代表自从构造以来对链表所做改变的次数。每次对 add 或 remove 的调用都将更新 modCount。其想法在于，当一个迭代器被建立时，他将存储集合的 modCount。每次对一个迭代器方法(next 或 remove)的调用都将用该链表内的当前 modCount 检测在迭代器内存储的 modCount，并且当这两个计数不匹配时抛出一个 ConcurrentModificationException 异常。

MyLinkedList 类的其余部分由构造方法、迭代器的实现以及一些方法组成。许多方法都只是一行代码。

图 3-26 中的 clear 方法由构造方法调用。它创建并连接头节点和尾节点，然后设置大小为 0。

在图 3-24 的第 41 行可以看到私有内部 LinkedListIterator 类的声明的开头部分。当我们在后面看到其具体实现时将讨论这些细节。

图 3-27 解释一个包含 x 的新节点是如何被拼接在由 p 引用的一个节点和 p.prev 之间的。这些节点链的赋值可以描述如下：

```
Node newNode = new Node( x, p.prev, p );  // 第1步和第2步
p.prev.next = newNode;                    // 第3步
p.prev = newNode;                         // 第4步
```

第 3 步和第 4 步可以合并，结果只有两行：

```
Node newNode = new Node( x, p.prev, p );  // 第1步和第2步
p.prev = p.prev.next = newNode;           // 第3步和第4步
```

可是这两行还可以合并，得到：

```
p.prev = p.prev.next = new Node( x, p.prev, p );
```

这就缩短了图 3-28 中的例程 addBefore。

图 3-29　指出删除一个节点的逻辑过程。如果 p 引用正在被删除的节点，那么在该节点被断开连接和可以被虚拟机回收之前只有两个链改动：

```
p.prev.next = p.next;
p.next.prev = p.prev;
```

图 3-30 显示基本的私有 remove 例程，该例程包含上述两行代码。