```
1  public class MyArrayList<AnyType> implements Iterable<AnyType>
2  {
3      private int theSize;
4      private AnyType [ ] theItems;
5          ...
6      public java.util.Iterator<AnyType> iterator( )
7        { return new ArrayListIterator<AnyType>( this ); }
8  }
9  class ArrayListIterator<AnyType> implements java.util.Iterator<AnyType>
10 {
11     private int current = 0;
12     private MyArrayList<AnyType> theList;
13          ...
14     public ArrayListIterator( MyArrayList<AnyType> list )
15       { theList = list; }
16
17     public boolean hasNext( )
18       { return current < theList.size( ); }
19     public AnyType next( )
20       { return theList.theItems[ current++ ]; }
21 }
```

图 3-18 迭代器 2 号版本(几乎能够使用)：迭代器是一个顶级类并存储当前位置以及一个连接到 MyArrayList 的链。它不能使用是因为 theItems 在 MyArrayList 类中是私有的

图 3-19 显示另一种解决方案，这种方案能够正确运行：使 ArrayListIterator 为**嵌套类**(nested class)。当我们让 ArrayListIterator 为一个嵌套类时，该类将被放入另一个类(此时就是 MyArrayList)的内部，这个类就叫做**外部类**(outer class)。我们必须用 static 来表示它是嵌套的；若无 static，将得到一个内部类，这有时候好，有时候也不好。嵌套类是许多编程语言的典型的类型。注意，嵌套类可以被设计成 private，这很好，因为此时该嵌套类除能够被外部类 MyArrayList 访问外，其他是不可访问的。更为重要的是，因为嵌套类被认为是外部类的一部分，所以不存在产生不可见问题：theItems 是 MyArrayList 类的可见成员，因为 next 是 MyArrayList 的一部分。

既然我们有了嵌套类，那么就可以讨论内部类。嵌套类的问题在于，在我们的原始设计中，当编写 theItems 而不引用其所在的 MyArrayList 的时候，代码看起来还可以，也似乎有意义，但却是无效的，因为编译器不可能计算出哪个 MyArrayList 在被引用。要是我们自己不必明了这一点那就好多了，而这恰是内部类要求我们所要做的。

当声明一个内部类时，编译器则添加对外部类对象的一个隐式引用，该对象引起内部类对象的构造。如果外部类的名字是 Outer，则隐式引用就是 Outer.this。因此，如果 ArrayListIterator 是作为一个内部类被声明且没有注明 static，那么 MyArrayList.this 和 theList 就都会引用同一个 MyArrayList。这样，theList 就是多余的，并可能被删除。

在每一个内部类的对象都恰好与外部类对象的一个实例相关联的情况下，内部类是有用的。在这种情况下，内部类的对象在没有外部类对象与其关联时是永远不可能存在的。对于 MyArrayList 及其迭代器的情形，图 3-20 指出了 MyArrayList 类和迭代器之间的关系，此时这些内部类都用来实现该迭代器。

theList.theItems 的使用可以由 MyArrayList. this. theItems 代替。这很难说是一种改进，但进一步的简化还是可能的。正如 this.data 可以简写为 data 一样(假设不存在引起冲突的也叫做 data 的另外的变量)，MyArrayList. this. theItems 可以简写为 theItems。图 3-21 指出 Ar-