3.4 ArrayList 类的实现

在这一节，我们提供便于使用的 ArrayList 泛型类的实现。为避免与类库中的类相混，这里将把我们的类叫做 MyArrayList。我们不提供 MyCollection 或 MyList 接口；MyArrayList 是独立的。在考查 MyArrayList 代码(接近 100 行)之前，先概括主要的细节。

1. MyArrayList 将保持基础数组，数组的容量，以及存储在 MyArrayList 中的当前项数。

2. MyArrayList 将提供一种机制以改变基础数组的容量。通过获得一个新数组，将老数组拷贝到新数组中来改变数组的容量，允许虚拟机回收老数组。

3. MyArrayList 将提供 get 和 set 的实现。

4. MyArrayList 将提供基本的例程，如 size、isEmpty 和 clear，它们是典型的单行程序；还提供 remove，以及两种不同版本的 add。如果数组的大小和容量相同，那么这两个 add 例程将增加容量。

5. MyArrayList 将提供一个实现 Iterator 接口的类。这个类将存储迭代序列中的下一项的下标，并提供 next、hasNext 和 remove 等方法的实现。MyArrayList 的迭代器方法直接返回实现 Iterator 接口的该类的新构造的实例。

3.4.1 基本类

图 3-15 和图 3-16 显示 MyArrayList 类。像它的 Collections API 的对应类一样，存在某种错误检测以保证合理的限界；然而，为了把精力集中在编写迭代器类的基本方面，我们不检测可能使得迭代器无效的结构上的修改，也不检测非法的迭代器 remove 方法。这些检测将在此后 3.5 节 MyLinkedList 的实现中指出，对于这两种表的实现它们是完全相同的。

如 5 到 6 行所示，MyArrayList 把大小及数组作为其数据成员进行存储。

从 11 行到 38 行，是几个短例程，即 clear、size、trimToSize、inEmpty、get 以及 set 的实现。

ensureCapacity 例程如 40 行到 49 行所示。容量的扩充是用与早先描述的相同的方法来完成的：第 45 行存储对原始数组的一个引用，第 46 行是为新数组分配内存，并在第 47 行和 48 行将旧内容拷贝到新数组中。如 42 行和 43 行所示，例程 ensureCapacity 也可以用于收缩基础数组，不过只是当指定的新容量至少和原大小一样时才适用。否则，ensureCapacity 的要求将被忽略。在第 46 行，我们看到一个短语是必须的，因为泛型数组的创建是非法的。我们的做法是创建一个泛型类型限界的数组，然后使用一个数组进行类型转换。这将产生一个编译器警告，但在泛型集合的实现中这是不可避免的。

图中显示了两个版本的 add。第一个 add 是添加到表的末端并通过调用添加到指定位置的较一般的版本而得以简单实现。这种版本从计算上来说是昂贵的，因为它需要移动在指定位置上或指定位置后面的那些元素到一个更高的位置上。add 方法可能要求增加容量。扩充容量的代价是非常昂贵的，因此，如果容量被扩充，那么，它就要变成原来大小的两倍，以避免不得不再次改变容量，除非大小戏剧性地增加(+1 用于大小是 0 的情形)。

remove 方法类似于 add，只是那些位于指定位置上或指定位置后的元素向低位移动一个位置。

剩下的例程处理 iterator 方法和相关迭代器类的实现。在图 3-16 中由第 77 行至第 96 行显示。iterator 方法直接返回 ArrayListIterator 类的一个实例，该类是一个实现 Iterator 接口的类。ArrayListIterator 存储当前位置的概念，并提供 hasNext、next 和 remove 的实现。当前位置表示要被查看的下一元素(的数组下标)，因此初始时当前位置为 0。