集合类的使用

从早些时候的那幅示意图可以看出，实际上只有三个集合组件：Map，List 和 Set。而且每个接口只有两种或三种实施方案。若需使用由一个特定的接口提供的功能，如何才能决定到底采取哪一种方案呢？

为理解这个问题，必须认识到每种不同的实施方案都有自己的特点、优点和缺点。比如在那张示意图中，可以看到 Hashtable，Vector 和 Stack 的“特点”是它们都属于“传统”类，所以不会干扰原有的代码。但在另一方面，应尽量避免为新的（Java 1.2）代码使用它们。

其他集合间的差异通常都可归纳为它们具体是由什么“后推”的。换言之，取决于物理意义上用于实施目标接口的数据结构是什么。例如，ArrayList，LinkedList 以及 Vector（大致等价于 ArrayList）都实现了 List 接口，所以无论选用哪一个，我们的程序都会得到类似的结果。然而，ArrayList（以及 Vector）是由一个数组后推得到的；而 LinkedList 是根据常规的双重链接列表方式实现的，因为每个单独的对象都包含了数据以及指向列表内前后元素的句柄。正是由于这个原因，假如想在一个列表中部进行大量插入和删除操作，那么 LinkedList 无疑是最恰当的选择（LinkedList 还有一些额外的功能，建立于 AbstractSequentialList 中）。若非如此，就情愿选择 ArrayList，它的速度可能要快一些。

作为另一个例子，Set 既可作为一个 ArraySet 实现，亦可作为 HashSet 实现。ArraySet 是由一个 ArrayList 后推得到的，设计成只支持少量元素，特别适合要求创建和删除大量 Set 对象的场合使用。然而，一旦需要在自己的 Set 中容纳大量元素，ArraySet 的性能就会大打折扣。写一个需要 Set 的程序时，应默认选择 HashSet。而且只有在某些特殊情况下（对性能的提升有迫切的需求），才应切换到 ArraySet。

①决定使用何种 List

在 ArrayList 中进行随机访问（即 get()）以及循环反复是最划得来的；但对于 LinkedList 却是一个不小的开销。但另一方面，在列表中部进行插入和删除操作对于 LinkedList 来说却比 ArrayList 划算得多。我们最好的做法也许是先选择一个 ArrayList 作为自己的默认起点。以后若发现由于大量的插入和删除造成了性能的降低，再考虑换成 LinkedList 不迟。

②决定使用何种 set

进行 add()以及 contains()操作时，HashSet 显然要比 ArraySet 出色得多，而且性能明显与元素的多寡关系不大。一般编写程序的时候，几乎永远用不着使用 ArraySet。

③决定使用何种 Map

选择不同的 Map 实施方案时，注意 Map 的大小对于性能的影响是最大的

即使大小为 10，ArrayMap 的性能也要比 HashMap 差——除反复循环时以外。而在使用 Map 时，反复的作用通常并不重要（get()通常是我们时间花得最多的地方）。TreeMap 提供了出色的 put()以及反复时间，但 get()的性能并不佳。但是，我们为什么仍然需要使用 TreeMap 呢？这样一来，我们可以不把它作为 Map 使用，而作为创建顺序列表的一种途径。树的本质在于它总是顺序排

列的，不必特别进行排序（它的排序方式马上就要讲到）。一旦填充了一个 TreeMap，就可以调用 keySet()来获得键的一个 Set“景象”。然后用 toArray()产生包含了那些键的一个数组。随后，可用 static 方法 Array.binarySearch()快速查找排好序的数组中的内容。当然，也许只有在 HashMap 的行为不可接受的时候，才需要采用这种做法。因为 HashMap 的设计宗旨就是进行快速的检索操作。最后，当我们使用 Map 时，首要的选择应该是 HashMap。只有在极少数情况下才需要考虑其他方法。

在写这个程序期间，TreeMap 的创建速度比其他两种类型明显快得多（但你应亲自尝试一下，因为据说新版本可能会改善 ArrayMap 的性能）。考虑到这方面的原因，同时由于前述 TreeMap 出色的 put()性能，所以如果需要创建大量 Map，而且只有在以后才需要涉及大量检索操作，那么最佳的策略就是：创建和填充 TreeMap；以后检索量增大的时候，再将重要的 TreeMap 转换成 HashMap——使用 HashMap(Map)构建器。同样地，只有在事实证明确实存在性能瓶颈后，才应关心这些方面的问题——先用起来，再根据需要加快速度。