```
1   // Computes a map in which the keys are words and values are Lists of words
2   // that differ in only one character from the corresponding key.
3   // Uses a quadratic algorithm (with appropriate Map).
4   public static Map<String,List<String>>
5   computeAdjacentWords( List<String> theWords )
6   {
7       Map<String,List<String>> adjWords = new TreeMap<String,List<String>>( );
8
9       String [ ] words = new String[ theWords.size( ) ];
10
11      theWords.toArray( words );
12      for( int i = 0; i < words.length; i++ )
13          for( int j = i + 1; j < words.length; j++ )
14              if( oneCharOff( words[ i ], words[ j ] ) )
15              {
16                  update( adjWords, words[ i ], words[ j ] );
17                  update( adjWords, words[ j ], words[ i ] );
18              }
19
20      return adjWords;
21  }
22
23  private static <KeyType> void update( Map<KeyType,List<String>> m,
24                                        KeyType key, String value )
25  {
26      List<String> lst = m.get( key );
27      if( lst == null )
28      {
29          lst = new ArrayList<String>( );
30          m.put( key, lst );
31      }
32
33      lst.add( value );
34  }
```

图 4-66　计算一个 Map 对象的函数，该对象以一些单词作为关键字而以只在一个字母处不同的一列单词作为关键字的值。该函数对一个 89 000 单词的词典运行 96 秒

第 3 个算法更复杂，使用一些附加的映射！和前面一样，将单词按照长度分组，然后分别对每组运算。为理解这个算法是如何工作的，假设我们对长度为 4 的单词操作。这时，首先要找出像 wine 和 nine 这样的单词对，它们除第 1 个字母外完全相同。对于长度为 4 的每一个单词，一种做法是删除第 1 个字母，留下一个 3 字母单词代表。这样就形成一个 Map,其中的关键字为这种代表，而其值是所有包含同一代表的单词的一个 List。例如，在考虑 4 字母单词组的第 1 个字母时，代表“ine”对应“dine”、“fine”、“wine”、“nine”、“mine”、“vine”、“pine”、“line”。代表“oot”对应“boot”、“foot”、“hoot”、“loot”、“soot”、“zoot”。每一个作为最后的 Map 的一个值的 List 对象都形成单词的一个集团，其中任何一个单词均可以通过单字母替换变成另一个单词，因此在这个最后的 Map 构成之后，很容易遍历它以及添加一些项到正在计算的原始 Map 中。然后，我们使用一个新的 Map 再处理 4 字母单词组的第 2 个字母。此后是第 3 个字母，最后处理第 4 个字母。

一般概述如下：