1.8。不过，很奇怪的是，整体运行时间只下降到 5.1 秒，差距只有 0.2 秒。

进一步观察，我们可以看到，表变大了但是性能提高很小，这是由于哈希函数选择的不好。简单地对字符串的字符编码求和不能产生一个大范围的值。特别是，一个字母最大的编码值是 122，因而 n 个字符产生的和最多是 $122n$。在文档中，最长的 bigram("honorificabilitudinitatibus thou")的和也不过是 3371，所以，我们哈希表中大多数桶都是不会被使用的。此外，可交换的哈希函数，例如加法，不能对一个字符串中不同的可能的字符顺序做出区分。例如，单词"rat"和"tar"会产生同样的和。

我们换成一个使用移位和异或操作的哈希函数。使用这个版本，显示为"Better Hash"，时间下降到了 0.6 秒。一个更加系统化的方法是更加仔细地研究关键字在桶中的分布，如果哈希函数的输出分布是均匀的，那么确保这个分布接近于人们期望的那样。

最后，我们把运行时间降到了大部分时间是花在 strlen 上，而大多数对 strlen 的调用是作为小写字母转换的一部分。我们已经看到了函数 lower1 有二次的性能，特别是对长字符串来说。这篇文档中的单词足够短，能避免二次性能的灾难性的结果；最长的 bigram 只有 32 个字符。不过换成使用 lower2，显示为"Linear Lower"得到很好的性能，整个时间降到了 0.2 秒。

通过这个练习，我们展示了代码剖析能够帮助将一个简单应用程序所需的时间从 3.5 分钟降低到 0.2 秒，得到的性能提升约为 1000 倍。剖析程序帮助我们把注意力集中在程序最耗时的部分上，同时还提供了关于过程调用结构的有用信息。代码中的一些瓶颈，例如二次的排序函数，很容易看出来；而其他的，例如插入到链表的开始还是结尾，只有通过仔细的分析才能看出。

我们可以看到，剖析是工具箱中一个很有用的工具，但是它不应该是唯一一个。计时测量不是很准确，特别是对较短的运行时间(小于 1 秒)来说。更重要的是，结果只适用于被测试的那些特殊的数据。例如，如果在由较少数量的较长字符串组成的数据上运行最初的函数，我们会发现小写字母转换函数才是主要的性能瓶颈。更糟糕的是，如果它只剖析包含短单词的文档，我们可能永远不会发现隐藏着的性能瓶颈，例如 lower1 的二次性能。通常，假设在有代表性的数据上运行程序，剖析能帮助我们对典型的情况进行优化，但是我们还应该确保对所有可能的情况，程序都有相当的性能。这主要包括避免得到糟糕的渐近性能(asymptotic performance)的算法(例如插入算法)和坏的编程实践(例如 lower1)。

1.9.1 中讨论了 Amdahl 定律，它为通过有针对性的优化来获取性能提升提供了一些其他的见解。对于 n-gram 代码来说，当用 quicksort 代替了插入排序后，我们看到总的执行时间从 209.0 秒下降到 5.4 秒。初始版本的 209.0 秒中的 203.7 秒用于执行插入排序，得到 $\alpha=0.974$，被此次优化加速的时间比例。使用 quicksort，花在排序上的时间变得微不足道，得到预计的加速比为 $209/\alpha=39.0$，接近于测量加速比 38.5。我们之所以能获得大的加速比，是因为排序在整个执行时间中占了非常大的比例。然而，当一个瓶颈消除，而新的瓶颈出现时，就需要关注程序的其他部分以获得更多的加速比。

5.15　小结

虽然关于代码优化的大多数论述都描述了编译器是如何能生成高效代码的，但是应用程序员有很多方法来协助编译器完成这项任务。没有任何编译器能用一个好的算法或数据结构代替低效率的算法或数据结构，因此程序设计的这些方面仍然应该是程序员主要关心的。我们还看到妨碍优化的因素，例如内存别名使用和过程调用，严重限制了编译器执行大量优化的能力。同样，程序员必须对消除这些妨碍优化的因素负主要的责任。这些应该被看作好的编程习惯的一部分，因为它们可以用来消除不必要的工作。