3）低级优化。结构化代码以利用硬件功能。

- 展开循环，降低开销，并且使得进一步的优化成为可能。
- 通过使用例如多个累积变量和重新结合等技术，找到方法提高指令级并行。
- 用功能性的风格重写条件操作，使得编译采用条件数据传送。

最后要给读者一个忠告，要警惕，在为了提高效率重写程序时避免引入错误。在引入新变量、改变循环边界和使得代码整体上更复杂时，很容易犯错误。一项有用的技术是在优化函数时，用检查代码来测试函数的每个版本，以确保在这个过程没有引入错误。检查代码对函数的新版本实施一系列的测试，确保它们产生与原来一样的结果。对于高度优化的代码，这组测试情况必须变得更加广泛，因为要考虑的情况也更多。例如，使用循环展开的检查代码需要测试许多不同的循环界限，保证它能够处理最终单步迭代所需要的所有不同的可能的数字。

5.14　确认和消除性能瓶颈

至此，我们只考虑了优化小的程序，在这样的小程序中有一些很明显限制性能的地方，因此应该是集中注意力对它们进行优化。在处理大程序时，连知道应该优化什么地方都是很难的。本节会描述如何使用*代码剖析程序*(code profiler)，这是在程序执行时收集性能数据的分析工具。我们还展示了一个系统优化的通用原则，称为 Amdahl 定律(Amdahl's law)，参见 1.9.1 节。

5.14.1　程序剖析

程序剖析(profiling)运行程序的一个版本，其中插入了工具代码，以确定程序的各个部分需要多少时间。这对于确认程序中我们需要集中注意力优化的部分是很有用的。剖析的一个有力之处在于可以在现实的基准数据(benchmark data)上运行实际程序的同时，进行剖析。

Unix 系统提供了一个剖析程序 GPROF。这个程序产生两种形式的信息。首先，它确定程序中每个函数花费了多少 CPU 时间。其次，它计算每个函数被调用的次数，以执行调用的函数来分类。这两种形式的信息都非常有用。这些计时给出了不同函数在确定整体运行时间中的相对重要性。调用信息使得我们能理解程序的动态行为。

用 GPROF 进行剖析需要 3 个步骤，就像 C 程序 `prog.c` 所示，它运行时命令行参数为 `file.txt`：

1）程序必须为剖析而编译和链接。使用 GCC(以及其他 C 编译器)，就是在命令行上简单地包括运行时标志“`-pg`”。确保编译器不通过内联替换来尝试执行任何优化是很重要的，否则就可能无法正确刻画函数调用。我们使用优化标志`-Og`，以保证能正确跟踪函数调用。

```
linux> gcc -Og -pg prog.c -o prog
```

2）然后程序像往常一样执行：

```
linux> ./prog file.txt
```

它运行得会比正常时稍微慢一点(大约慢 2 倍)，不过除此之外唯一的区别就是它产生了一个文件 `gmon.out`。

3）调用 GPROF 来分析 `gmon.out` 中的数据。

```
linux> gprof prog
```