3.2　程序编码

假设一个 C 程序，有两个文件 p1.c 和 p2.c。我们用 Unix 命令行编译这些代码：

```
linux> gcc -Og -o p p1.c p2.c
```

命令 gcc 指的就是 GCC C 编译器。因为这是 Linux 上默认的编译器，我们也可以简单地用 cc 来启动它。编译选项-Og[⊖]告诉编译器使用会生成符合原始 C 代码整体结构的机器代码的优化等级。使用较高级别优化产生的代码会严重变形，以至于产生的机器代码和初始源代码之间的关系非常难以理解。因此我们会使用-Og 优化作为学习工具，然后当我们增加优化级别时，再看会发生什么。实际中，从得到的程序的性能考虑，较高级别的优化(例如，以选项-O1 或-O2 指定)被认为是较好的选择。

实际上 gcc 命令调用了一整套的程序，将源代码转化成可执行代码。首先，C 预处理器扩展源代码，插入所有用#include 命令指定的文件，并扩展所有用#define 声明指定的宏。其次，编译器产生两个源文件的汇编代码，名字分别为 p1.s 和 p2.s。接下来，汇编器会将汇编代码转化成二进制目标代码文件 p1.o 和 p2.o。目标代码是机器代码的一种形式，它包含所有指令的二进制表示，但是还没有填入全局值的地址。最后，链接器将两个目标代码文件与实现库函数(例如 printf)的代码合并，并产生最终的可执行代码文件 p(由命令行指示符-o p 指定的)。可执行代码是我们要考虑的机器代码的第二种形式，也就是处理器执行的代码格式。我们会在第 7 章更详细地介绍这些不同形式的机器代码之间的关系以及链接的过程。

3.2.1　机器级代码

正如在 1.9.3 节中讲过的那样，计算机系统使用了多种不同形式的抽象，利用更简单的抽象模型来隐藏实现的细节。对于机器级编程来说，其中两种抽象尤为重要。第一种是由指令集体系结构或指令集架构(Instruction Set Architecture，ISA)来定义机器级程序的格式和行为，它定义了处理器状态、指令的格式，以及每条指令对状态的影响。大多数 ISA，包括 x86-64，将程序的行为描述成好像每条指令都是按顺序执行的，一条指令结束后，下一条再开始。处理器的硬件远比描述的精细复杂，它们并发地执行许多指令，但是可以采取措施保证整体行为与 ISA 指定的顺序执行的行为完全一致。第二种抽象是，机器级程序使用的内存地址是虚拟地址，提供的内存模型看上去是一个非常大的字节数组。存储器系统的实际实现是将多个硬件存储器和操作系统软件组合起来，这会在第 9 章中讲到。

在整个编译过程中，编译器会完成大部分的工作，将把用 C 语言提供的相对比较抽象的执行模型表示的程序转化成处理器执行的非常基本的指令。汇编代码表示非常接近于机器代码。与机器代码的二进制格式相比，汇编代码的主要特点是它用可读性更好的文本格式表示。能够理解汇编代码以及它与原始 C 代码的联系，是理解计算机如何执行程序的关键一步。

x86-64 的机器代码和原始的 C 代码差别非常大。一些通常对 C 语言程序员隐藏的处理器状态都是可见的：

- 程序计数器(通常称为“PC”，在 x86-64 中用%rip 表示)给出将要执行的下一条指令在内存中的地址。

⊖ GCC 版本 4.8 引入了这个优化等级。较早的 GCC 版本和其他一些非 GNU 编译器不认识这个选项。对这样一些编译器，使用一级优化(由命令行标志-O1 指定)可能是最好的选择，生成的代码能够符合原始程序的结构。