C 语言：标准与实现

The Standards and Implementations of the C Programming Language

Volume 1

x86/GNU/Linux Edition

Rev 0.9c
Released on August 28

版本号	发布日期	主要的变化
0.9	07-01-04	初始版本
0.9a	07-06-04	1）整本书改用 16 开本进行排版，页数因此有所改变 2）所有命令的名字用不同的字体明确标识出来 3）代码、表格与文字之间适当加上空行以方便阅读 4）增加了几处参考文献条目
0.9b	07-11-04	1）所有用户键入的内容用粗体表示 2）把所有“ld-2.3.2.so”替换成“ld-linux.so.2” 3）补充关于动态库代码在运行期被载入程序映射进入进程地址空间的内容 4）关于“.size”语句的更正描述
0.9c	07-20-04	1）增加进程的虚拟地址空间描述，图 03-01、图 05-01 2）增加 Linux 中用户进程和内核的虚拟地址空间描述，图 01-05 3）增加关于中断、异常的基础知识 4）给本书的 PDF 文件增加书签

前言

转瞬间一年过去了，我终于可以向读者呈上本书的第一卷。原先我曾承诺会在这个时候完成整部书，非常遗憾的是，我没有实现这个目标。不过，我心里竟有某种欢喜的情绪，虽然读者暂时能够看到的仅仅是第一卷，但这也意味着我将更加详细地剖析 C 语言的方方面面，最终受益的仍然是渴望开卷的有心人。

目前，32 位微处理器仍然在桌面计算机领域占据主流地位，因此这里还是使用基于 IA-32 系列处理器的 PC 作为硬件平台。由于某些型号的 i386/i486 处理器可能没有配备浮点运算处理器，而 Pentium 处理器也缺乏某些高级特性，所以建议读者最好能够拥有一台基于 P6 处理器的 PC，P6 处理器家族主要有以下常见的处理器：

Intel Pentium II/III/4（包括对应型号的 Celeron/Xeon）

AMD K7/Athlon XP（包括对应型号的 Duron）

另外，根据我的风格本书一如既往地采用 GNU/Linux 系统作为示例平台进行讲解。GNU/Linux 系统虽然有很多不同的发行版本，但基本上不会对读者构成不利的影响，本书的结论不会因 GNU/Linux 发行版本的不同而产生明显的差异。当然，如果读者手头上的 GNU/Linux 系统能够和当前主流版本保持一定的同步那就最好。在众多可供使用的 GNU/Linux 发行版本中，本书选择使用 Slackware GNU/Linux 9.1，读者可以上网进行免费下载。尽管几乎每个 GNU/Linux 发行版本都提供了 GCC，但我们可能仍然需要自己动手安装最新版本的 GCC，因为版本越新意味着对标准的支持越完善，安装的详细步骤见附录。

最后，我要对下列人士表示最衷心的感谢：

我的父母，姚善刚先生和曾秋燕女士，他们一直给我提供足够的物质支持，让我可以全力投入到写作中去。我的朋友，肖原、曹文花、王文娟、肖涛、陈伟斌等，他们慷慨地给我提供了必要的帮助。

欢迎读者对本书提出批评和建议，通过 fredyao@163.com 可以联系作者。

姚新颜

2004 年 8 月 27 日　广州

C 语言和操作系统

在这里再次结合

目录

参考资料

[ANSI 1986]
ANSI X3.4-1986:
Coded Character Sets - 7-Bit American National Standard Code for Information Interchange

[Bovet 2002]
Daniel P. Bovet, Marco Cesati:
Understanding the Linux Kernel (2nd Edition)
O'Reilly & Associates, 2002 ISBN: 0596002130

[David 2001]
David A. Wheeler:
Program Library HOWTO

[Dean 1994]
Dean Elsner, Jay Fenlason & friends:
Using as

[Goldberg 1991]
David Goldberg:
What Every Computer Scientist Should Know About Floating Point Arithmetic
ACM Computing Surveys, Vol 23, No 1, March 1991

[Harbison & Steele 2002]
Samuel Harbison and Guy Steele:
C: A Reference Manual (5th Edition)
Prentice Hall, 2002 ISBN: 0-13-089592-X

[IEC 1989]
IEC 60559:1989
Binary floating-point arithmetic for microprocessor systems

[IEEE 1985]
IEEE Std 754-1985:
IEEE Standard for Binary Floating-Point Arithmetic

[IEEE 1987]
IEEE Std 854-1987:
IEEE Standard for Radix-Independent Floating-Point Arithmetic

[IEEE 1993]
IEEE Std 1596.5-1993
IEEE Standard for Shared-Data Formats Optimized for Scalable Coherent Interface(SCI) Processors

[Intel 2004]
Intel Corp.:
IA-32 Intel Architecture Software Developer's Manual
Volume 1, 2, 3

[ISO 1998]
ISO/IEC 14882:1998
Programming languages — C++

[ISO 1999]
ISO/IEC 9899:1999/Cor1:2001
Programming languages — C

[Kahan 1996]
Prof.W.Kahan:
IEEE Standard 754 for Binary Floating-Point Arithmetic

[K&R 1988]
Brian W. Kernighan and Dennis M. Ritchie:
The C Programming Language(2nd Edition)
Prentice-Hall, 1988 ISBN: 0-13-110362-8

[Konstantin 2001]
Konstantin Boldyshev:
Linux Assembly HOWTO

[Linden 1994]
Peter van der Linden:
Expert C Programming
Prentice Hall, 1994 ISBN: 0-13-177429-8

[Meyers 2000]
Randy Meyers:
The New C: Introducing C99
C/C++ Users Journal, Vol 18, No 10, October 2000

[Meyers 2000b]
Randy Meyers:
The New C: Integers in C99
C/C++ Users Journal, Vol 18, No 12, December 2000
Vol 19, No 1, January 2001
Vol 19, No 2, February 2001

[Neumann 2000]
John von Neumann:
The Computer and the Brain(2nd edition)
Yale University Press, 2000 ISBN: 0-300-08473-0

[Plauger 1992]
P.J. Plauger:
The Standard C Library
Prentice Hall, 1992 ISBN: 0-13-131509-9

[Ritchie 1978]
Dennis M. Ritchie:
C Reference Manual
Bell Telephone Laboratories, 1978

[SPARC 1993]
SPARC International, Inc.:
The SPARC Architecture Manual(Version 9)
Prentice Hall, 1993 ISBN: 0-130-99227-5

[Stallman 2002]
Richard M. Stallman:
GNU Compiler Collection Internals

[Stallman 2004]
Richard M. Stallman, Roland Pesch, Stan Shebs:
Debugging with GDB

[Stallman 2004a]
Richard M. Stallman:
Using and Porting the GNU Compiler Collection

[Steve 1994]
Steve Chamberlain:
Using ld

#1

基础知识

01 基本概念

1642年，法国数学家 Blaise Pascal（1623-1662，布莱斯·帕斯卡）发明了第一台计算器 Pascaline，它有8个刻度盘，最多可以进行8位数的加法运算。从此，人类正式开始了制造自动计算机的旅程[1]。到20世纪40年代初为止，人们已经制造了大量相当复杂的自动计算机。这些计算机有一个共同的特点，那就是依靠模拟量进行运算。譬如，机械式计算机可能利用圆盘的旋转角度、而电气式计算机则可能通过电流的大小或电压的高低表示运算量。这类模拟计算机借助精巧的机械、电气结构进行算术四则运算，但它们的发展前景受到很大限制，这主要表现在几个方面：

- 精度

由于模拟量（旋转角度、电流大小等）的特点，人们无法任意提高运算的精度。当时最优秀的模拟计算机也只能达到10^{-5}数量级的精度，而且，想要再提高精度非常困难，模拟机已经差不多达到了它的极限。

- 存储

模拟计算机没有存储的概念，模拟量从初始状态经过运算后直接得出最后结果，在这个过程中并没有把中间结果或部分结果保存起来并在需要的时候再取出使用的机制，这就限制了运算的规模。

- 控制

模拟计算机在运算的过程中仍然需要人们作出某种程度的安排、调整。例如，对于使用转盘的机械计算机来说，表示输入的圆盘和表示输出的圆盘通过某种方式相互关联，而这种关联方式是可以根据实际计算的需要而重新组装。也就是说，求解不同问题的时候，人们可能要改变模拟计算机的某些部件以适合具体的计算。

面对日益增长的计算需求，人们开始设计数字计算机。相应地，数字计算机具有以下特点[2]：

- 数值用离散信号序列表示

和模拟计算机不同，数字计算机使用离散的信号表示运算量，例如在一根导线上，有脉冲和无脉冲就是一对离散的信号，它可以表示两个值。通过把多个信号并行或串行起来，人们就得到一系列数值的表示方法，而且精度可以根据需要大幅度地提高。

- 基本运算具有严格的逻辑性

数字计算机的基本运算表现为某些严格适用的逻辑规则，以二进制的加法运算为例，当两个二进制数值相加时，每个位的数字什么时候为“0”、什么时候为“1”，在什么情况下应该进位等这些特性具有严格一致的逻辑。

- 存储部件的必要性

由于数字计算机的运算部件具有严格的逻辑性，所以每一个运算部件只能进行一种基本运算，这就需要存储部件[3]保存运算的中间结果，以便计算机在后面的运算中能够重新使用它。

1945年，美国数学家 John von Neumann（1903-1957，约翰·冯·诺依曼）进一步提出“程序存储式计算机”的概念：数字计算机通过一系列的指令进行控制，这些指令其实也是一些符号序列，它们分别完成不同的任务。存储部件的每一个区域单元都被编号[4]，指令保存在存储部件中。这样，指令只要包括运算的类型以及参与运算的存储单元

的编号即可作出控制指示。数字计算机在指令的控制下，从存储单元取出数值进行运算，然后把结果放回存储单元，如此不断重复，直到计算完成。而整个计算过程需要用到的指令与数据则保存在预先编好的程序里面，并在计算一开始的时候从外部设备加载到存储部件。

这个划时代的设计思想使“Von Neumann”成为程序存储式计算机的代名词[5]。沿着“程序存储控制”这个思路发展，得益于半导体制造技术的不断改进，计算机经历了电子管、晶体管、集成电路、超大规模集成电路等阶段。1972年，Intel 推出了代号为“4004”的微处理芯片，4004是第一块把计算机核心处理部分（包括运算单元、控制单元）所有元件封装在同一块芯片的产品。此后，微处理器的运算能力成为整个计算机工业发展水平的重要标志。70年代末，IBM 推出个人用的计算机产品，这种被称为 PC 的微型计算机销量逐年上升，并且随着微处理器的升级换代而拥有越来越强大的计算能力。

今天，参与市场竞争的计算机系统为数不少，虽然它们各有特色，但对于绝大多数程序员来说，它们都只不过是一台最简单的冯•诺依曼机，再复杂的结构也可以用下面这些基本部件进行概括：处理器、内存、外部存储设备、输入设备、输出设备以及控制单元（负责以上部件之间的数据交换）[6]，如图01-01：

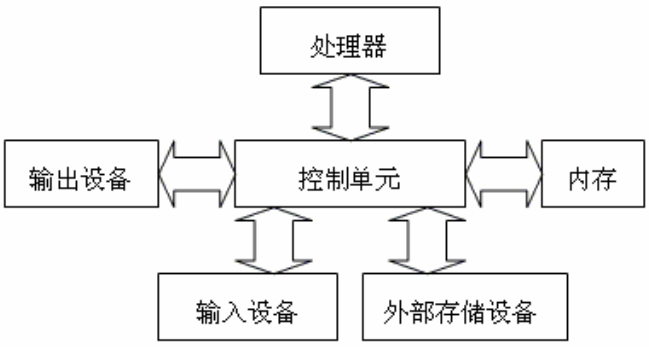


图 01-01

处理器从外部存储设备把程序（指令和数据的集合）载入内存，在一系列初始化动作之后进程开始执行。处理器通过输入、输出设备与用户进行交互，得到必要的输入数据从而完成运算，最后输出结果。在操作系统的协调下，计算机不断重复上述过程。

※ 程序（program）在内存中运行的实例称为“进程（process）”。程序与进程的区别很简单，比方说，你在按照一份图纸制作玩具模型，那么你就是处理器，这份图纸就是“程序”，它指示你应该怎样做；而你整个制作的过程就相当于“进程”，它是你依据图纸所进行的一次实际任务。同一个程序可以对应多个进程又或者不同的程序可以对应不同的进程，正如你可以按照同一份图纸制作多个模型或者按照不同的图纸制作不同的模型。 ※

现代计算机微处理器的设计异常复杂，不同架构的处理器往往有很大的差别，但在某种抽象层次上，它们也有异曲同工之处，譬如下面这些特性：

- **至少具有两个不同的特权级别**

不同特权级别的代码拥有不同的权限，例如，操作系统通常不允许用户进程直接访问

计算机硬件端口、不允许用户进程访问某些关键的系统寄存器等等，但操作系统自己本身却要访问这些资源。于是，处理器必须提供某种机制来判断代码所进行的操作是否合法，这就引入了特权级别的概念。一般地，处理器都至少支持两个特权级别：内核级和用户级。

内核级是操作系统核心代码使用的级别，处理器完全信任内核级的代码，这些代码可以不受限制地访问各种系统资源（包括某些重要的寄存器、特定的内存区域、硬件端口等等）；用户级是用户进程使用的级别，处理器在执行用户级代码时会自动设置很多严格的保护限制，如果用户进程的指令试图突破这些限制，处理器就会进入相应的异常处理。

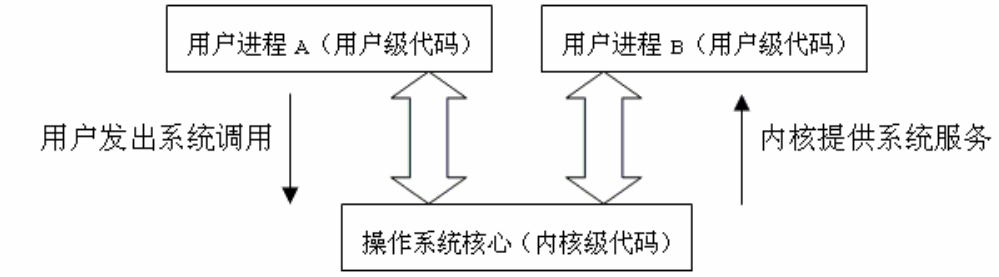


图 01-02

不同架构的处理器通过各种各样的方式允许用户进程调用内核代码为自己提供系统服务，从而间接地实现对某些关键资源的访问。这些系统服务完成某些重要的基本操作（例如打开一个文件、新建一个进程等等）。显然，系统服务代码工作在内核级，因此，用户进程必须以某种操作系统允许的方式才能调用它们。[7]

- **支持虚拟内存**

简单地概括，这种虚拟机制的关键在于区分虚拟地址与物理地址。譬如说，32位的处理器通常拥有4GB 的虚拟地址空间，但这并不意味着每一台使用32位处理器的计算机都必须配备4GB 物理内存。实际上，超过99%的32位计算机都是工作在内存远远小于4GB的状态中。这里面并没有什么惊人的奥妙，启用虚拟内存机制的处理器（在操作系统的协助下）会把进程需要访问到的虚拟地址通过内存管理单元转换成物理地址，再通过该物理地址真正对内存进行读写操作，如图01-03。

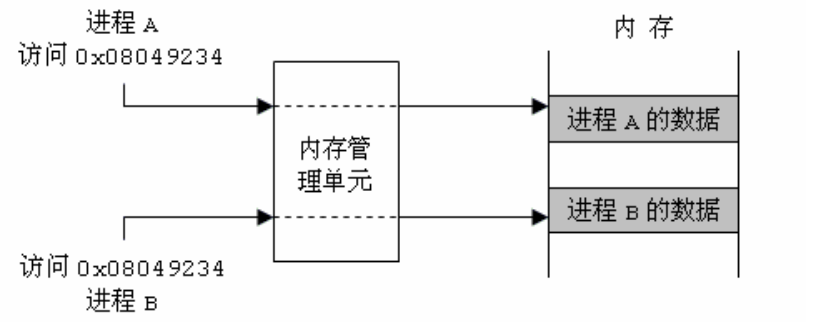


图 01-03

使用虚拟内存之后，每个进程都可以使用同一个相对较大的地址空间而无须担心地址

发生冲突，因为不同进程的虚拟地址空间会被处理器映射到不同的物理地址空间，从而使不同的进程相互分隔开来，每一个进程都以为自己在独自使用整个虚拟地址空间，若非得到操作系统的允许，任何进程都不可能访问到其它进程的指令和数据。

不同的处理器实现虚拟内存的方式虽然不一样，但基本上都是以“分页”机制为基础。例如，图01-04是 P6处理器转换32位虚拟地址的主要方式[8]。

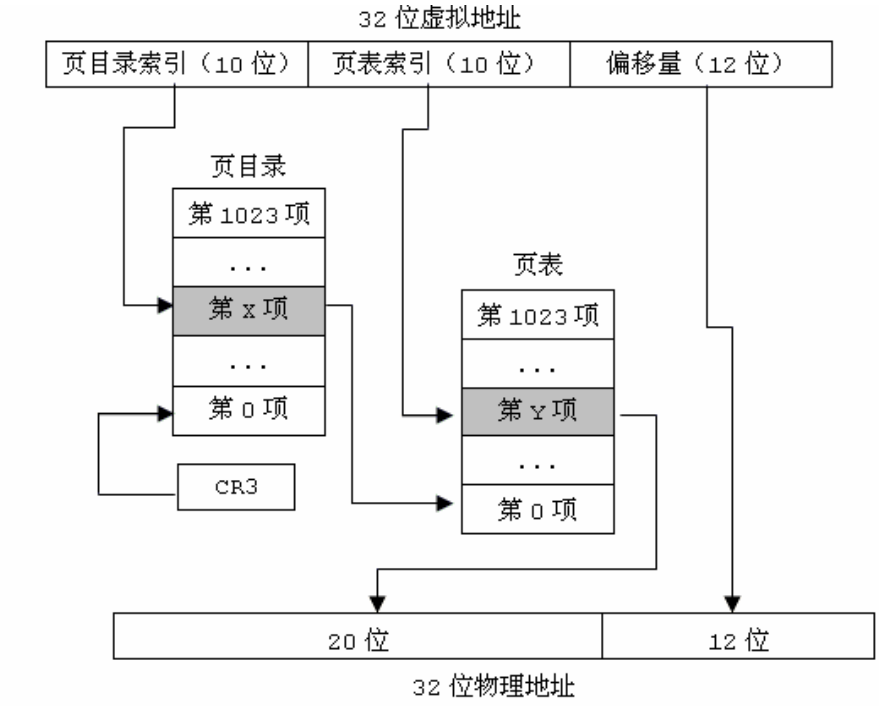


图 01-04

虚拟地址的高20位跟物理地址的高20位并没有任何固定的联系，虚拟地址的高20位数据仅仅是两个索引号。系统控制寄存器 CR3指出“页目录（Page Directory）”的起始物理地址，处理器根据虚拟地址给出的数据（位22~位31）作为索引号在一个拥有2^{10}项的页目录中找到对应的项目(假设是第 X 项),里面含有多个信息,包括对应“页表(Page Table)”的起始物理地址。然后，处理器同样以虚拟地址的相关数据（位12~位21）作为索引号在一个拥有1024项的页表中找到对应的项目（假设是第 Y 项),其中同样含有多项信息，包括最终要生成的物理地址的高20位，这20位与虚拟地址的低12位（位0~位11）一起构成了一个完整的32位物理地址。[9]

可以看到，当 CR3的值不改变时，所有高20位相同的虚拟地址处于同一个小组，小组内的所有地址都拥有相同的特性，因为它们共享同一个页表项。每一个小组都对应一块内存区域，大小为2^{12}字节。我们把这块区域称为“页（page)”。

对于不同的进程,操作系统只需用很小的内存空间维护一份该进程独自使用的页目录与页表，在进程切换时，处理器会自动保存 CR3的值，以便切换回来时仍然能找到属于该进程的页目录和页表。由于不同的进程对应不同的页目录与页表，所以即使两个进程使用相同的虚拟地址也不会发生任何冲突。当然，这一切需要操作系统的密切配合。

在页目录和页表的项目中都有描述对应结构是否存在的标志位（这里不妨称它为“P位”，“P”代表“Present”），如果页目录中某一个项目的P位为“1”，则表示此项目对应的页表不存在；如果页表中某一个项目的P位为“0”，则表示此项目对应的页不存在。当要访问的地址所处的页表或页不存在时，处理器会进入一个异常处理过程，根据具体情况决定下一步的行动。例如其中一种情况是：页表或页虽然不在物理内存中，但之前已经被保存到外部存储设备（硬盘），这时，处理器只需把页表或页从外部存储设备复制到内存，然后重新执行引起异常的指令即可[10]。计算机之所以能够在物理内存远小于虚拟内存的情况下正常运行的奥妙正在于此，在操作系统的指挥下，处理器把以页为单位的数据在内存与磁盘之间搬来搬去，内存空间不够就用磁盘空间来搭救。

另外，页目录和页表的项目中还有控制对应结构是否可写的标志位（这里称之为“R/W位”，“R”代表“Read”，“W”代表“Write”）。当要对一个地址进行写操作时，如果该地址所处的页表或页的属性是“可读不可写”，处理器会进入一个异常处理过程。如果证实进程对只读内存区域进行写操作的确是违规行为，则进程通常会接收到一个信号，要是进程没有捕捉这个信号进行处理，那系统的默认动作是终止该进程[11]。通过设置页目录或页表的R/W位，操作系统不仅可以控制进程对某些内存区域的访问方式，还能够实现一些更高级的内存管理策略。

不要以为虚拟内存机制只能够实现进程地址空间的隔离，其实，基于分页的虚拟内存机制同样可以实现内存区域在不同的进程之间进行共享。因为页目录和页表是由操作系统进行管理的，如果操作系统允许某些进程共享一些内存页面，则只须把对应的那部分页目录和页表项目设置成指向相同的物理地址即可。这样，虽然每个进程使用各自的页目录和页表，但最终访问的物理内存地址却是相同的，从而实现内存区域的共享。

事实上，操作系统都会利用内存页面共享机制来减少物理内存的消耗。这方面的典型例子很多，譬如用户多次运行同一个程序的时候，虽然是启动了对应的多个进程，但由于这些进程的代码都是相同的，所以实际上操作系统只给它们的代码部分分配一份物理内存页面。至于这些进程的数据页面，则除非发生了写操作（这意味着某个进程的数据页面产生了改变），否则连数据部分的物理内存页面也是共享的。通过内存页面共享，操作系统最大限度地节约物理内存的使用。

另外，操作系统把虚拟地址空间同样看作是一种系统资源进行管理。虽然32位处理器拥有4GB的虚拟地址空间，但这并不意味着每个进程都可以自由使用任何一个虚拟地址。一般地，操作系统会保留某些区域不让用户进程使用，这些区域永远不可能分配给用户进程使用，虚拟地址“0”就是一个最常见的例子。[12]

Linux把4GB的虚拟地址空间划分为两部分：用户地址空间和内核地址空间，通常情况下，分界线是0xC0000000，即地址0x00000000~0xBFFFFFFF属于用户空间、地址0xC0000000~0xFFFFFFFF属于内核空间。用户进程只能访问用户空间的地址，而内核进程可以访问全部地址。一部分的内核空间被用来对物理内存进行映射，假设系统装有256MB的物理内存，则内核进程访问0xC0000000~0xCFFFFFFF的虚拟内存其实就是访问0x00000000~0x0FFFFFFF的物理内存，如图01-05。

除了属于内核空间的那部分地址之外，即使是剩下的虚拟地址，操作系统同样不允许进程随意使用，进程如果要获得更大的虚拟地址空间必须向操作系统提交申请，成功通过批准后才能使用额外的地址空间。

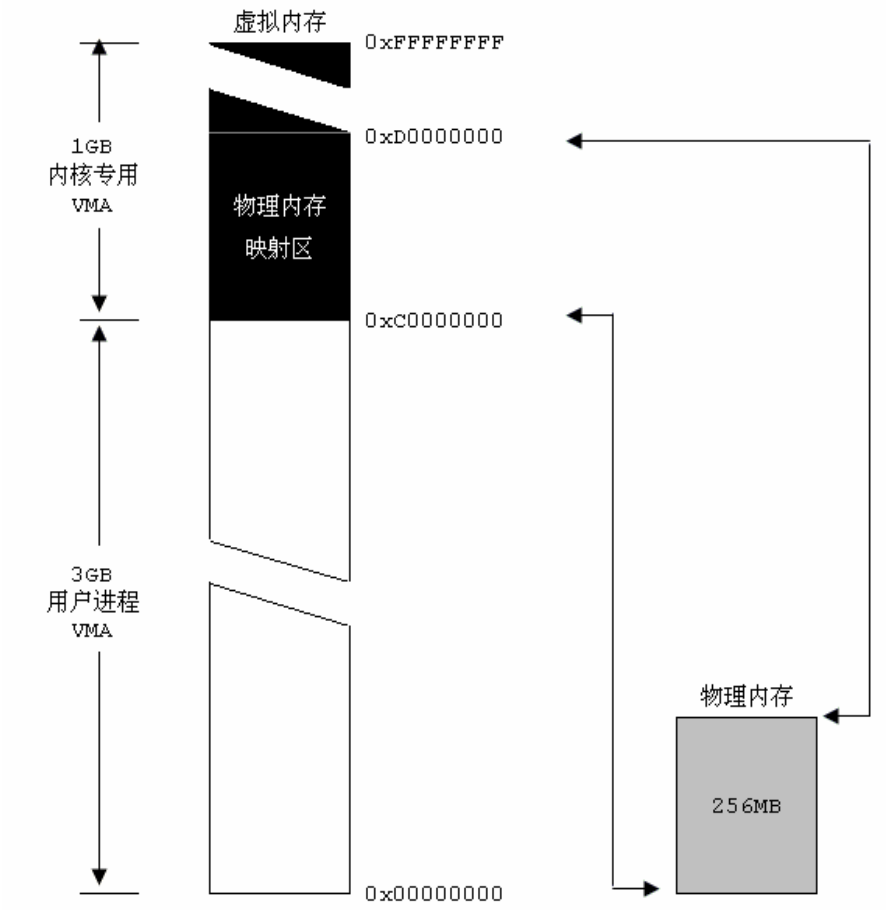


图 01-05

※ Linux 内核固定地会被加载到物理地址0x00100000，因此内核本身会占用从0xC0100000开始的一段逻辑地址空间，其大小依内核的具体配置而定。在笔者的系统上，Linux 内核使用了超过4MB 的空间，以下是内核符号表的几个关键符号的地址：

```
c0100000  A  _text
c030fe43  A  _etext
c03e0100  A  _edata
c03e2000  A  __init_begin
c0404000  A  __bss_start
c04240dc  A  _end
c0425000  A  pg0
```

由于 x86平台的 Linux 内核将最高的128MB 虚拟地址空间保留以作其它用途，所以能够直接映射到内核地址空间的物理内存最多只有896MB，对于那些装有超过900MB 内存的 x86系统，Linux 内核使用更加复杂的方法把物理内存映射到内核地址空间，这项特性需要在编译内核时进行选定。不过，照目前的状况看来，大多数 x86系统所配备的内存数量都不会超过512MB，因此本书不打算详细讨论 Linux 的“大内存模式”，有兴趣的读者可以参阅相关技术资料。 ※

粗略地说，当用户进程向系统申请使用更多的内存，譬如最常见的是调用标准 C 库函数 malloc()要求在堆里面分配内存，Linux 的做法是把响应的过程分为两个阶段：分配虚拟地址空间和分配物理内存页面。前面已经提到，如果没有系统的批准，那即使是属于用户地址空间的虚拟地址，用户进程同样不能访问。所以，当系统接纳进程要求使用更多内存的申请后第一步要做的就是给进程分配一块虚拟地址区域，这个区域通常比进程实际申请的内存空间要稍微大一点，这样，进程就拥有了合法访问对应虚拟地址区域的权限。但是，系统并不会立即为进程真正分配物理内存页面，因为系统的策略是尽量推迟物理内存页面的分配。于是，内核会通过设置有关的页目录、页表等数据结构把刚刚分配的虚拟内存页面全部指向同一个物理内存页面（零页面），这个物理内存页面在系统初始化时就已经被零填充，并且属性是可读不可写。一旦进程真正开始使用申请得到的内存空间，也就是说，往对应区域进行写操作[13]，系统就会进入异常处理，直到这时系统才真正为进程分配对应的物理内存页面，然后重新设定页目录和页表，让虚拟内存页面对应新的物理内存页面。从异常返回后，处理器会重新执行刚才那条引发异常的指令，就好象什么也没发生过一样。

- **中断和异常**

无论是中断还是异常，它们都会改变处理器当前的指令执行序列，因为它们的出现本身就意味着某个特殊的事件发生了从而需要处理器立刻跟进。由于在众多的技术资料以及标准之间存在不太一致的用法，所以这里不打算严格定义中断和异常的概念，不过，给出大致适当的描述还是完全可以做到的。

我们大体上按照 Intel 在 x86处理器技术文档中的观点来使用中断和异常这两个术语。简单地说，中断分两类，一类是外部中断，它是外部设备发送给处理器的请求信号，外部中断是随机的、异步的；而另一类是软件中断，进程通过它直接进入某个向量（vector）所对应的处理逻辑。P6处理器一共有256个向量，用0至255来进行编号。在前面的32个向量中，仅有2号向量是留给中断使用的，它对应的是不可屏蔽外部中断（NMI），因此，其它所有的中断，不管是外部中断还是软件中断，都是使用32~255号向量。

异常，顾名思义就是指处理器察觉到某种不正常的情形产生了，对于 P6处理器，异常使用前32个向量（除了2号向量）。如果再仔细分类，异常可以分为3类：程序错误异常、编程异常以及机器检测异常。一般地，我们最常见的是程序错误异常，例如前面在讨论虚拟内存机制时涉及到的“缺页异常”。

根据异常产生后处理器执行后续指令的不同，Intel 把程序错误异常分为“故障（fault）”、“陷阱（trap）”和“终止（abort）”。如果处理器在异常处理完毕之后返回并重新执行原先引发异常的那条指令，那么该异常就属于“故障”；如果处理器在异常返回后执行的是引发异常的指令的下一条指令，那么该异常就称为“陷阱”；如果处理器在异常发生后不会返回原先的指令执行路线那该异常就是“终止”。

不过，在其它地方，“陷阱”这个术语多数用来描述处理器会重新执行引发异常的指令的那类异常，这和 Intel 的做法显然不一样，因为 Intel 已经使用“故障”来描述它了。本书选择使用“陷阱”，毕竟这是大家的共识，而对于无必要区分“故障”、“陷阱”、“终止”等术语的场合，本书一律使用“异常”。

P6处理器向量表

向量	代表符号	描述	类型
0	#DE	Divide Error	Fault
1	#DB	RESERVED	
2		NMI	Interrupt
3	#BP	Breakpoint	Trap
4	#OF	Overflow	Trap
5	#BR	BOUND Range Exceeded	Fault
6	#UD	Invalid Opcode	Fault
7	#NM	No Math Coprecessor	Fault
8	#DF	Double Fault	Abort
9			
10	#TS	Invalid TSS	Fault
11	#NP	Segment Not Present	Fault
12	#SS	Stack-Segment Fault	Fault
13	#GP	General Protection	Fault
14	#PF	Page Fault	Fault
15		RESERVERD	
16	#MF	Math Fault	Fault
17	#AC	Alignment Check	Fault
18	#MC	Machine Check	Abort
19	#XF	SIMD Floating-point Exception	Fault
20-31		RESERVED	
32-255		User Defined Interrupt	Interrupt

14

在 Linux 系统中，用户进程只能够使用下面4条指令：

```
int 3
into
bound
int 0x80
```

分别引发3号、4号、5号编程异常以及0x80号软件中断，其余所有向量对应的中断、异常均被内核接管，用户进程不能直接通过“int *N*”这样的指令引发编程异常或软件中断。如果处理器引发了异常，Linux 使用标准的信号机制通知用户进程，我们将在后面看到如何利用 C 标准库的 signal()函数注册自己的函数从而使用户进程在接收到系统信号之后可以对异常进行处理。

*　　*　　*　　*　　*　　*

[0101]

中国古代的算盘不是计算机，因为它没有任何（自动）运算的能力。使用算盘的人必须熟记各种指

法规则（例如“三下五去二”等）然后亲自运用这些法则参与运算，也就是说，真正进行运算的其实是人而不是算盘。归根到底，算盘只是一套设计巧妙的记录系统，它的作用仅仅与人们进行手工计算时的纸、笔相当。

[0102]

请参阅[Neumann 2000]。

[0103]

从那时起，存储部件一直被称为“memory（内存）”。

[0104]

同样，存储单元的编号一直被称为“address（地址）”。

[0105]

人们习惯把“程序存储式计算机”称为“冯·诺依曼计算机”，其实英国计算机科学家 Alan Turing（1912-1954，阿兰·图灵）几乎在同一时间也提出了类似的构想。第一台程序存储式计算机在1952年制造成功。到了今天，无论是微型电脑还是超级计算机都基于程序存储式计算机的思想进行设计。

[0106]

这里为了简化问题进行适当的抽象，笔者当然不会认为控制单元可以直接输出显示信号给显示器（集成了显卡的芯片组除外）。

[0107]

P6处理器上的 Linux 提供0x80号软中断作为用户调用系统服务的统一入口，附录 E 有 Linux 系统服务的简单列表。

[0108]

工作在保护模式下的 P6处理器有三种地址概念：逻辑地址、线性地址和物理地址。但大多数操作系统，包括 Linux，都使用“带保护的平坦模式（Protected Flat Model）”管理内存。在这种模式下，逻辑地址和线性地址其实是一样的，相当于我们这里说的虚拟地址。另外，P6处理器支持4MB 大小的页，而且可以通过 PAE 模式支持64GB 物理内存，当启用上述任何一种特性时，32位虚拟地址的转换机制与正文中描述的不完全相同。

[0109]

值得注意的是，处理器负责把虚拟地址转换为物理地址，但转换过程中需要用到的页目录和页表等数据结构却是由操作系统负责构建和维护的。虚拟内存机制是处理器的特性，但离不开操作系统的参与。

[0110]

理论上，页表、页都可以被交换出内存从而保存在磁盘当中，但实际上并不是所有的页表、页都有机会被换出内存，操作系统基于效率方面的考虑通常会采取对应的特定交换策略，一个最明显的例子是，Linux 内核自身的代码、数据页面以及所有的页表（无论是内核所使用的还是用户进程所使用的）都永远不会被换出内存。

[0111]

在你的32/64位平台上用 C 或 C++编译器编译以下这个程序，然后运行试试：

```
const int i = 0;
const int *p = &i;
int main(){
    *(int*)p = 0;           /* 对只读内存区域进行写操作 */
}
```

[0112]

在你的32/64位平台上用 C 或 C++编译器编译以下这个程序，然后运行试试：

```
int main(){
    *(int*)0 = 0;           /* 访问虚拟地址“0”*/
}
```

[0113]

如果是读操作，那么进程读取到的将是零页面里面的零，系统不会为读操作分配物理页面，因为对刚刚申请得到的内存区域进行读操作要么是程序的逻辑有问题要么进程本身就是期望得到零。

[0114]

i386及其后的 x86处理器都不会再引发9号异常。

02　P6处理器的栈

P6处理器只配有很少量的通用寄存器，因此P6处理器在“过程（procedure）”的调用、返回中更多地依赖于“栈”这种数据结构的使用来实现参数的传递，8个通用寄存器中有2个专门负责栈的寻址操作：ESP和EBP，前者是“栈指针”，而后者则被称为“栈框指针”。

ESP始终指向栈的顶部[1]，每次压栈/出栈操作都会导致ESP数值的减少/增加。在32位系统上，每次压栈操作“PUSH　SRC”相当于（连续）完成两个步骤：

```
ESP ← ESP - 4                    /* 处理器把ESP的数值减去4 */
[ESP] ← SRC                      /* 把压栈对象复制到ESP所指向的内存区域 */
```

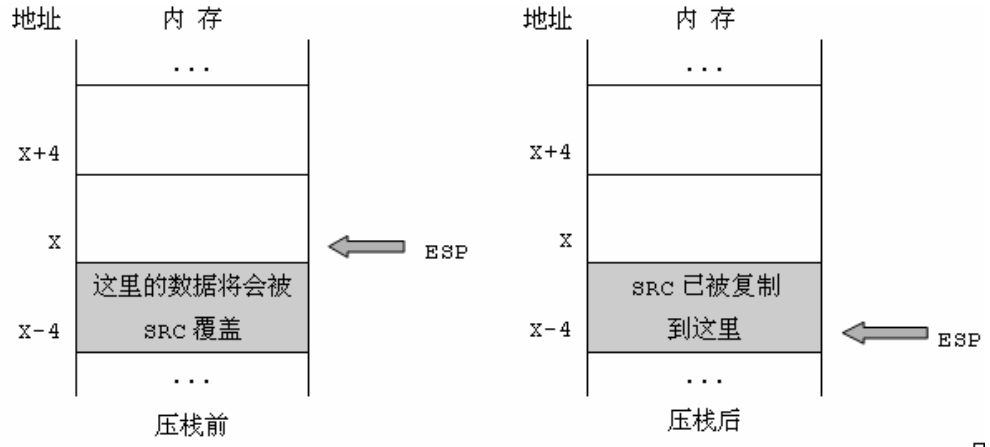


图 02-01

而每次出栈操作“POP　DEST”则相当于（连续）完成：

```
DEST ← [ESP]                     /* 出栈对象被复制到DEST */
ESP ← ESP + 4                    /* 处理器把ESP的数值增加4 */
```

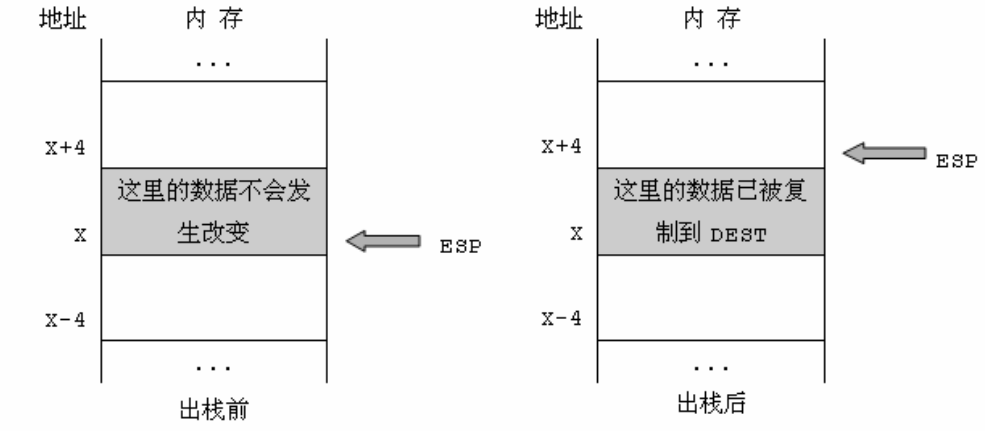


图 02-02

明白这些细节后我们再来看看过程的调用。现在假设有3个过程：A、B和C，它们的关系是A调用了B，B又调用了C，C没有调用任何其它过程，于是处理器在执行完过程

C 之后返回 B，执行完过程 B 后又返回 A，然后继续执行 A 剩下的指令，如图02-03：

```
procedure A{          procedure B{          procedure C{
    ...                   ...                   ...
    call B                call C                ...
    ...                   ...                   ...
}                     }                     }
```

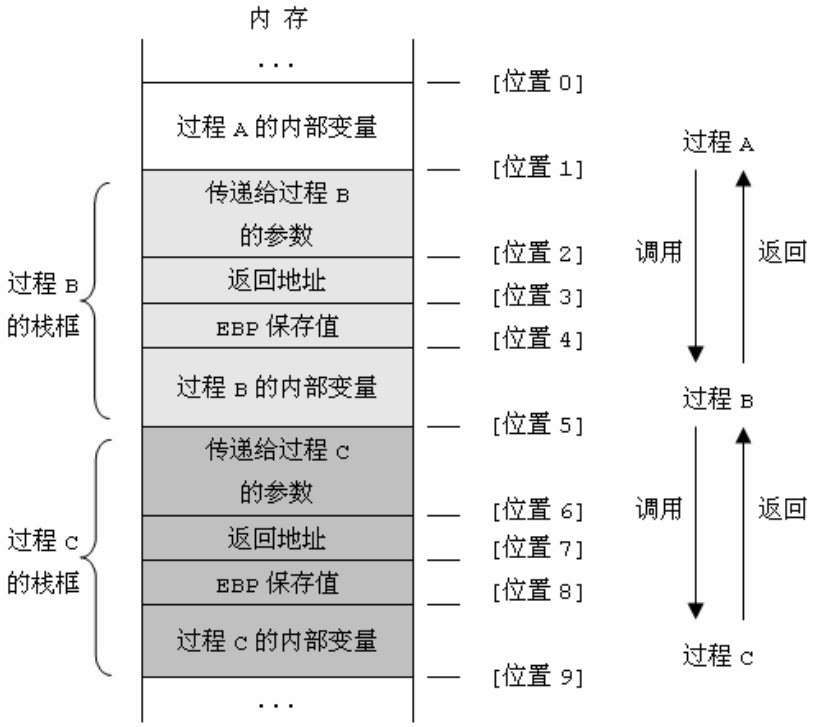


图 02-03

处理器刚转入过程 A 后，会用一条“PUSH EBP”指令保存 EBP 的值，然后把 ESP 的值复制给 EBP，这时 ESP 和 EBP 都指向[位置0]。

由于要给过程 A 的内部变量分配空间，ESP 会减去一个数值，具体减去多少由 A 的内部变量所占用的总空间决定，这时 ESP 指向[位置1]，虽然 EBP 和 ESP 都可以用来寻址栈里面的数据，但通常处理器会用 EBP 寻址 A 的内部变量。

当要调用过程 B 时，处理器会用 PUSH 指令把参数压栈，于是 ESP 指向[位置2]。然后处理器执行 CALL 指令转入过程 B，CALL 指令会自动把返回地址（即过程 A 中紧接着 CALL 指令的下一条指令的地址）压栈，因此刚进入 B 时 ESP 指向[位置3]。接下来发生的事就跟过程 A 里面的一样：处理器把 EBP 压栈、把 ESP 的值复制给 EBP。此刻，ESP 和 EBP 都指向[位置4]。

为过程 B 的内部变量预留空间后，ESP 指向[位置5]，在 B 的整个执行过程中，处理器用 EBP 寻址 A 传给 B 的参数和 B 自己的内部变量。

调用过程 C 发生的事跟前面说的完全相同：参数压栈、ESP 指向[位置6]，执行 CALL

指令（返回地址自动压栈）、ESP 指向[位置7]。进入过程 C 后，先压栈保存 EBP 的值、再把 ESP 的值赋给 EBP、这时 EBP 和 ESP 都指向[位置8]，然后为 C 的内部变量分配空间，ESP 指向[位置9]，在 C 的整个执行过程里，处理器也是用 EBP 寻址 B 传给 C 的参数以及 C 的内部变量。

过程 C 没有调用其它过程，所以当 C 执行完毕之后就要返回 B。要返回 B，关键是要恢复 EBP 和 ESP 的值。处理器用两条指令完成这项工作：

```
MOV  ESP, EBP
POP  EBP
```

第一条指令使 ESP 重新指向[位置8]，第二条指令使 EBP 重新指向[位置4]、使 ESP 指向[位置7]。然后处理器通过 RET 指令返回过程 B。RET 指令会自动从当前 ESP 指向的地方取出返回地址，处理器根据这个返回地址把执行点转移到过程 B 中紧接着调用 C 的 CALL 指令的下一条指令，刚刚返回到 B 时，ESP 指向[位置6]。处理器一般会马上把 ESP 调整到[位置5]，否则那些传给 C 的参数会继续占用栈空间而导致浪费。同时，由于 EBP 已经指向[位置4]，所以处理器在执行 B 剩下的指令时完全不用改变对参数和内部变量的寻址方式，就好像从来没有调用过 C 一样。

当 B 执行完毕需要返回 A 时，处理器同样通过上面两条指令恢复 ESP 和 EBP，这时 EBP 指向[位置0]、ESP 指向[位置3]。接着执行 RET 指令返回到 A，ESP 指向[位置2]，处理器在调整 ESP 使其指向[位置1]后继续执行 A 余下的指令……

以上的讨论是针对普遍情况的，如果在过程的调用中没有参数的传递，处理器就省略参数压栈的步骤而直接用 CALL 指令跳转；如果某些过程没有自己的内部变量，处理器就省略为内部变量分配空间而调整 ESP 的指令。除此之外的其它工作（包括进入过程后的 EBP 压栈、复制 ESP 的值给 EBP、在返回之前用两条指令恢复 EBP 和 ESP 以及调整 ESP 来收回参数占用的栈空间等）处理器无论如何都会照做，因为这些步骤关系到栈的正常使用。

在整个活动中，EBP 的角色很重要。它的贡献主要有：

- 负责参数及内部变量的寻址

上面提到，ESP 也可以完成同样的功能。但由于 ESP 经常随着栈的变动而改变，所以参数和内部变量相对于 ESP 的偏移量也经常改变。如果用 ESP 寻址，编译器就必须准确地调整各种偏移量，这样计算量会大大增加，有些情况下甚至很复杂[2]。而 EBP 则不同，每当进入一个新的过程后它就保持不变，相当于一个固定的指针，通过它来寻址参数和内部变量是很轻松的。

- 为栈的跟踪调试提供支持

很多时候，我们需要调试程序告诉我们在整个过程调用链中栈的变化情况。譬如在上面的例子里，当进入到过程 C 的时候，如果我们需要调试程序把过程 B 和过程 A 的栈使用情况全部描绘出来，那么，使用 ESP 完成这个任务在某些情况下同样极其困难，而用 EBP 就很好办：在过程 C 里面，EBP 指向[位置8]。调试程序很清楚地知道从[位置8]开始的连续4个字节里保存着EBP在进入过程C之前的值，它只需取出这个值就可以知道[位置4]是调用 C 的上一级过程 B 的内部变量存储区的起始边界，从而准确知道过程 B 的栈布局[3]；同理，它也知道从[位置4]开始的4个字节保存着调用 B 的上一级过程 A 的内部变量存储区的起始边界……就这样，依靠 EBP 的帮助，调试程序即使在最底层的过程 C 里

面也能够把所有上级过程各自的参数和内部变量悉数准确地描绘出来。

图02-03以及相应的描述主要是为了让大家对过程调用中栈的变化有个基本印象，在实际的代码中，编译器基于各种考虑会作出某些改动，例如：

- 过程的内部变量空间分配与子过程参数空间分配同时完成，以图02-03为例，进入过程 A 之后，ESP 的值从[位置0]直接变化到[位置2]；进入过程 B 之后，ESP 的值从从[位置4]直接变化到[位置6]。
- 传递参数时使用 MOV 指令而不使用 PUSH 指令。很自然地，如果 ESP 指针直接调整到为子过程的参数预留出空间的位置，我们显然就不能再用 PUSH 指令传递参数。
- 用“LEAVE”指令代替“MOV ESP, EBP”和“POP EBP”两条指令
- 如果我们明确表示不需要 EBP 提供栈框信息[4]，编译器可以省去进入过程后保存 EBP 以及退出过程前恢复 EBP 的动作并且直接使用 ESP 来寻址过程的参数和内部变量。

在本小节的最后，我们讨论一下字节顺序（byte order）的概念：对于某种硬件平台，多字节数据在内存中是以怎样的顺序存放呢？

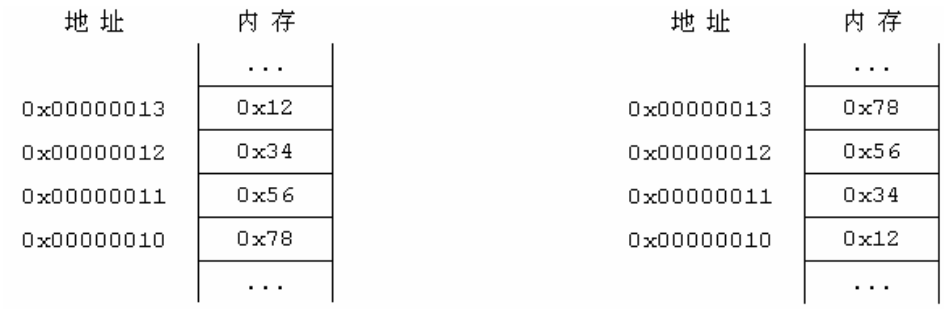


图 02-04

譬如有一个32位的整数0x12345678（占4个字节），假设这个数存放在起始地址为0x00000010的内存中，那么，从0x00000010到0x00000013这4个字节内存的情况有两种可能：数据的低字节部分存放在低地址内存或者数据的高字节部分存放在低地址内存。如图02-04。

各种处理器架构在这个问题上可以分为对应的两大类：Little-Endian 和 Big-Endian。IA-32属于 LE 类;大多数 RISC 架构的 CPU(例如 UltraSPARC、PowerPC 等）则属于 BE 类。本书的所有例子都是在 P6平台上编译执行，请大家在阅读的时候对这个问题给予适当的留意。[5]

*　　*　　*　　*　　*　　*

[0201]

注意，“栈的顶部（top of stack)”是指栈的会动态变化的那一端。

[0202]

例如过程内部使用了变长数组。

[0203]

知道了分界线，再通过源程序文件知道内部变量和参数的类型从而知道它们各自占用多少空间，这两者配合起来也就知道了当前过程的整个栈的布局。

[0204]

对于 ***gcc*** 和 ***g++***，可以使用“-fomit-frame-poiter”编译选项。

[0205]

如果有可能，试试分别在 LE 平台和 BE 平台编译、执行下面的 C 程序：

```
#include <stdio.h>
int main()
{
    int i = 0x00000041;
    printf("%s\n", (char*)&i);
}
```

03 从汇编语言开始

简单地说，一个程序最重要的两个部分分别是数据和对数据进行操作的指令。因此，我们必须要清楚地了解一个程序的数据与指令是如何被组织起来的。虽然当今的程序文件结构已经变得比较复杂，绝大部分人不可能对每一处细节都了如指掌，但如果只是要求有一个正确的总体认识，还是不难做到的。下面就以一个最简单的汇编程序开始我们的探索旅程：

```
A01     # Code 03-01, file name: 0301.s
A02         .data
A03         .align 4
A04     msg:
A05         .string "Hello, Linux!\n"
A06         .align 4
A07     len:
A08         .long 14
A09         .text
A10     .globl _start
A11         .align 4
A12     _start:
A13         movl  $4, %eax
A14         movl  $1, %ebx
A15         movl  $msg, %ecx
A16         movl  len, %edx
A17         int   $0x80
A18         movl  $1, %eax
A19         xorl  %ebx, %ebx
A20         int   $0x80
```

在03-01里面，数据部分（用“.data”标识）包括：

一个有15个字符的字符串（A05），符号“msg”代表它的起始地址（A04）；

一个32位整形变量（A08），符号“len”代表它的地址（A07）；

而代码部分（用“.text”标识）则包括了8条指令（A13~A20），符号“_start”（A12）代表第一条指令的地址。

大家可以先编译、链接、运行03-01，看看程序的输出：

```
$as  -o  0301.o  0301.s
$ld  0301.o
$./a.out
Hello, Linux!
$
```

※ 03-01对 Linux 内核进行了两次系统调用，指令“int $0x80”表示请求内核服务。A13~A17是4号系统调用，在指定的文件（文件描述符编号是“1”，代表标准输出文件）中写入特定数量（14个）的字符；A18~A20是1号系统调用，通知操作系统结束本进程返回到 SHELL 环境。 ※

正如大家看到的，我们要先用汇编语言编译程序把0301.s 编译成目标代码（object code）文件0301.o 才能继续下一步操作。现在我们就观察一下目标代码文件0301.o 究竟有些什么东西：

```
$as  -o  0301.o  0301.s
$objdump  -xd  0301.o

0301.o:     file format elf32-i386
0301.o
architecture: i386, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x00000000

Sections:
Idx Name   Size      VMA         LMA         File off  Algn
0   .text  00000020  00000000    00000000  00000034  2**2
           CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
1   .data  00000014  00000000    00000000  00000054  2**2
           CONTENTS, ALLOC, LOAD, DATA
2   .bss   00000000  00000000    00000000  00000068  2**2
           ALLOC

SYMBOL TABLE:
00000000   l   d    .text   00000000
00000000   l   d    .data   00000000
00000000   l   d    .bss    00000000
00000000   l        .data   00000000  msg
00000010   l        .data   00000000  len
00000000   g        .text   00000000  _start

Disassembly of section .text:
00000000 <_start>:
 0: b8 04 00 00 00          mov    $0x4,%eax
 5: bb 01 00 00 00          mov    $0x1,%ebx
 a: b9 00 00 00 00          mov    $0x0,%ecx
                       b: R_386_32     .data
 f: 8b 15 10 00 00 00       mov    0x10,%edx
```

```
                    11: R_386_32    .data
15: cd 80                   int    $0x80
17: b8 01 00 00 00          mov    $0x1,%eax
1c: 31 db                   xor    %ebx,%ebx
1e: cd 80                   int    $0x80
```

带有“-xd”选项的 ***objdump*** 命令的输出有4部分：

第1部分是关于文件的总体信息，例如文件格式（32位 ELF）、对应的平台（i386）、是否需要重定位（需要）、是否有符号表（有）以及开始执行点的地址（0x00000000）。

对于上面的信息，最关键的是 HAS_RELOC 标志。由于03-01有两条语句（A15、A16）访问数据区里的数据，但数据的地址（即 msg 和 len 的值）在未经过链接程序进行链接之前是不能确定的，所以0301.o 必须给出某种标志，指示链接程序对相应的代码进行重定位。同理大家可以理解为什么开始执行点的地址是“0x00000000”（这个0x00000000无实质意义），因为“_start”的值此刻仍然不能确定。

第2部分显示了文件各个区域（section）的详细信息。“.text”区储存指令代码，“.data”区和“.bss”区都储存数据，区别是“.data”区保存有初始值的数据，而“.bss”区则保存没有初始值的数据的信息。

很直观地，我们知道“.text”区的大小是32（0x20）字节，其中第1条指令相对于文件开头的偏移量是52（0x34）个字节；而“.data”区则占用了20（0x14）字节，第一个数据相对于文件开头的偏移量是84（0x54）字节。并且，大家可以发现各个区的起始地址（LMA 和 VMA[1]）都是0x00000000（因为所有地址都未确定）。

第3部分是符号表。这里记录着各个符号所代表的数据或代码在相应区域里的偏移量以及一些属性信息。例如符号“msg”所代表的数据的地址相对于“.data”区起始地址[2]的偏移量是0x00000000；“len”所代表的数据的偏移量是0x00000010；“_start”所代表的代码的地址相对于“.text”区起始地址的偏移量是0x00000000。

第4部分是代码的反汇编。这里清楚地指示出有两个地方必须由链接程序进行重定位：相对于“_start”偏移量为“0x0b”和“0x11”的连续4个字节。对照03-01，我们发现这正好就是 A15和 A16对应的指令。在目标代码文件0301.o 里面，由于“msg”和“len”的值（即它们所代表的数据的地址）未最后确定，所以在代码中填入的是两个偏移量：“0x00000000”和“0x00000010”。等到链接程序确定了“.data”区起始地址之后，“msg”和“len”的值也就随之确定了，这时链接程序只要在需要重定位的这两个地方分别加上“.data”区的起始地址值即可。

现在我们用链接程序完成0301.o 的链接并生成最终的可执行文件 a.out，再用 ***objdump*** 观察它（下面的输出内容省略了部分无关信息）：

```
$ld  0301.o
$objdump  -xd  a.out

a.out:     file format elf32-i386
a.out
architecture: i386, flags 0x00000112:
```

```
EXEC_P, HAS_SYMS, D_PAGED
start address 0x08048074

Program Header:
 LOAD off 0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
  filesz  0x00000094 memsz 0x00000094 flags r-x
 LOAD off 0x00000094 vaddr 0x08049094 paddr 0x08049094 align 2**12
  filesz  0x00000014 memsz 0x00000014 flags rw-

Sections:
Idx Name     Size        VMA         LMA         File off    Algn
0   .text    00000020    08048074    08048074    00000074    2**2
             CONTENTS, ALLOC, LOAD, READONLY, CODE
1   .data    00000014    08049094    08049094    00000094    2**2
             CONTENTS, ALLOC, LOAD, DATA
2   .bss     00000000    080490a8    080490a8    000000a8    2**2
             ALLOC

SYMBOL TABLE:
08048074      l    d    .text    00000000
08049094      l    d    .data    00000000
080490a8      l    d    .bss     00000000
08049094      l         .data    00000000    msg
080490a4      l         .data    00000000    len
08048074      g         .text    00000000    _start

Disassembly of section .text:
08048074 <_start>:
8048074:      b8 04 00 00 00       mov    $0x4,%eax
8048079:      bb 01 00 00 00        mov    $0x1,%ebx
804807e:      b9 94 90 04 08        mov    $0x8049094,%ecx
8048083:      8b 15 a4 90 04 08     mov    0x80490a4,%edx
8048089:      cd 80                 int    $0x80
804808b:      b8 01 00 00 00        mov    $0x1,%eax
8048090:      31 db                 xor    %ebx,%ebx
8048092:      cd 80                 int    $0x80
```

与0301.o对比，我们可以发现：

第1部分，“HAS_RELOC”没有了（因为程序代码已经通过链接程序完成了重定位），增加了“EXEC_P”和“D_PAGED”，前者表示本文件是可执行文件，后者表示启用了分

页功能。另外，程序的起始执行点确定为0x08048074。

※ 全局符号“_start”代表整个程序第一条指令的地址，每个用户进程都是从“_start”开始执行指令。因此，一个程序必须有而且只能有一个全局的“_start”符号。 ※

第2部分是a.out的“程序头”，它告诉操作系统在程序载入内存运行的时候如何给进程分配页面。由于03-01极其简单，所以这里只有两段内容，第1段是关于存放代码指令的页面，该页面的属性为“r-x”（可读可执行不可写），程序的指令机器码在文件里和在内存里的大小都是148（0x94）字节[3]；第2段是关于存放数据的页面，该页面的属性为“rw-”（可读可写不可执行），程序的数据在文件里和在内存里的大小都是20（0x14）字节[4]。

第3部分仍然是各区域的信息：

- VMA和LMA不再是0x00000000
- “.text”区没有了“RELOC”标志
- 各区域在文件内的位置稍有变动（“File off”的值变大了，因为系统需要更多的空间存放相关信息）

第4部分，符号表最明显的变化就是：

- “.text”区、“.data”区和“.bss”区都有了确定的起始地址，分别为：0x08048074、0x08049094和0x080490a8
- “msg”、“len”和“_start”同样都有了固定的值，分别为：0x08048094、0x080490a8和0x08049074

第5部分清楚地显示了重定位之后的指令机器码。所有指令都有固定的地址值，而不像先前那样只有偏移值。注意0x0804807f开始的4个字节及0x08048085开始的4个字节都已经改变，前者的新值是0x08049094（在0301.o里对应于该处的值是0x00000000），由于“.data”区的起始地址是0x08049094，而：

0x08049094 + 0x00000000 = 0x08049094

检查符号表可以发现“msg”的值正好就是0x08049094。同理，后者的新值是0x080490a4（在0301.o里对应的值是0x00000010），“.data”区的起始地址是0x08049094：

0x08049094 + 0x00000010 = 0x080490a4

检查符号表，“len”的值就是0x080490a4。当程序在内存中运行时，上面的两条指令完全可以正确地访问到需要访问的数据。

唯一令大家感到不解的可能是：为什么不索性把整个程序的执行点（即第一条指令的地址）整整齐齐地安排在代码页面的起始地址0x08048000呢？同样，第一个数据的地址为什么不是0x08049000而是0x08049094？

答案很简单，因为程序的“.text”区保存在可执行文件a.out的偏移0x74处（一共32字节），而“data”区则保存在偏移0x94处（一共20字节）。 如果把执行点和第一个数据分别安排在0x08048000和0x08049000，则在操作系统将程序从磁盘载入内存的复制过程中还需要小心翼翼地处理这些偏移量：0x74、0x94……计算比较烦琐复杂。一旦采用上面的地址分配策略（即指令和数据的地址的低12位与它们在文件中的偏移量一

致），系统就可以很简单地把程序载入到内存中。

假设第一条指令保存在文件偏移 A、“.text”区占 N 字节、“.data”区占 M 字节（一般地，第一个数据相应地就会保存在文件偏移 A+N），代码页的起始分配地址是 B（B 一定是4096的倍数），则按下面的方法计算：

```
X = (A+N) mod 4096
Y = A+N-X
IF (X == 0)
    Z = Y
ELSE
    Z = Y+4096
```

然后把文件前面的（A+N）个字节复制到从地址 B 开始的内存区域，接着把文件从偏移 Y 开始的（X+M）个字节复制到从地址（B+Z）开始的内存区域即可完成代码和数据的正确装载。03-01虽然只有“.text”和“.data”区，但这里描述的原理同样适用于以后更为复杂、有更多个区的程序。

不过，03-01不能代表所有重定位的情况。我们再来看看另一种情形：

```
A01     # Code 03-01A, file name: 0301a.s
A02         .data
A03         .align 4
A04     .globl msg
A05     msg:
A06         .string “Hello, Linux!\n”
A07         .align 4
A08     .globl len
A09     len:
A10         .long 14
A11         .text
A12     .globl _start
A13         .align 4
A14     _start:
A15         movl  $4, %eax
A16         movl  $1, %ebx
A17         movl  $msg, %ecx
A18         movl  len, %edx
A19         int   $0x80
A20         movl  $1, %eax
A21         xorl  %ebx, %ebx
A22         int   $0x80
```

03-01A 和03-01极其相似，甚至运行结果也是完全一模一样。但由于多了两行：

```
.globl msg
.globl len
```

编译器对它的具体处理在细节上便有所不同。这两行新增语句的作用是告诉编译器，符号“msg”和“len”是全局可见的，就是说，其它目标代码文件也可以引用这两个符号所代表的数据。

我们看看“把 msg、len 设置成全局符号”而引起的变化：

```
$as  -o  0301a.o  0301a.s
$objdump  -xd  0301a.o
0301a.o:    file format elf32-i386
0301a.o
architecture: i386, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x00000000

Sections:
Idx Name   Size        VMA         LMA         File off    Algn
0   .text  00000020    00000000    00000000    00000034    2**2
           CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
1   .data  00000014    00000000    00000000    00000054    2**2
           CONTENTS, ALLOC, LOAD, DATA
2   .bss   00000000    00000000    00000000    00000068    2**2
           ALLOC

SYMBOL TABLE:
00000000     l     d    .text    00000000
00000000     l     d    .data    00000000
00000000     l     d    .bss     00000000
00000000     g           .data    00000000  msg
00000010     g           .data    00000000  len
00000000     g           .text    00000000  _start

Disassembly of section .text:
00000000 <_start>:
 0: b8 04 00 00 00               mov    $0x4,%eax
 5: bb 01 00 00 00               mov    $0x1,%ebx
 a: b9 00 00 00 00               mov    $0x0,%ecx
                      b: R_386_32    msg
 f: 8b 15 00 00 00 00           mov    0x10,%edx
                     11: R_386_32   len
15: cd 80                        int    $0x80
```

```
17: b8 01 00 00 00              mov    $0x1,%eax
1c: 31 db                       xor    %ebx,%ebx
1e: cd 80                       int    $0x80
```

我们很容易就从 ***objdump*** 命令的输出信息里找到两个主要的差异：

- 在符号表部分，“msg”和“len”的链接性质是“g”，即“global”，这表示“msg”和“len”是全局符号，在全局范围内都可以被引用；而在0301.o 的对应位置上，“msg”和“len”的链接性质是“l”，即“local”。
- 在反汇编部分，需要链接程序重定位的两处地方的值都是0x00000000，并且编译器明确指出将来重定位时链接程序应该用“msg”的值直接填入从“_start“开始偏移0x0000000b 的4个字节、用“len”的地址直接填入偏移0x00000011的4个字节；而在0301.o 中，需要重定位的两个地方的值分别是0x00000000和0x00000010，并且编译器指示将来重定位时链接程序必须用该文件那个“.data”区的最终地址分别加上这两个地方原先的值从而得到正确地址值。

到目前为止，我们已经掌握到重要线索。编译器对全局符号和局部符号有不同的处理，对于全局符号，链接程序直接用该符号的值（即链接程序为该符号所代表的数据分配的起始地址）填入需要重定位的字节；对于局部符号，链接程序需要先将某个区（可能是“.data”、“.bss”或者“.text”，视该符号代表有初始值数据、无初始值数据又或者代码而定）的起始地址和该符号对应的偏移量相加，然后再把这个相加的结果填入需要重定位的字节。

上面举的例子是基于数据的重定位，其实代码的重定位也是如此，譬如：

```
A01     # Code 03-01B, file name: 0301b.s
A02         .text
A03         .align 4
A04     f:
A05         ret
A06     .globl _start
A07         .align 4
A08     _start:
A09         call  f
A10         movl  $1, %eax
A11         xorl  %ebx, %ebx
A12         int   $0x80
```

03-01B 更简单，A09调用 f 所代表的代码（A05），我们看看重定位的具体安排：

```
$as  -o  0301b.o  0301b.s
$objdump  -xd  0301b.o

...
Disassembly of section .text:
```

```
00000000 <f>:
0:  c3                ret

00000001 <_start>:
 1: e8 fa ff ff ff    call   0 <f>
 6: b8 01 00 00 00    mov    $0x1,%eax
 b: 31 db             xor    %ebx,%ebx
 d: cd 80             int    $0x80
```

上面省略了其它部分的输出，仅仅保留重定位信息。call 指令后面的4个字节值是0xfffffffa，即(-6)，这是一个偏移量，起算点是 call 指令后面那条指令的地址，从表中可以知道，无论最后链接程序给 call 指令后面那条 mov 指令分配的地址是多少，总之该地址加上(-6)这个偏移量就一定是 f 的值，call 指令肯定可以正确地跳转到 ret 指令那里。

对比一下“f”是全局符号的情形：

```
A01     # Code 03-01C, file name: 0301c.s
A02         .text
A03         .align 4
A04     .globl f
A05     f:
A06         ret
A07     .globl _start
A08         .align 4
A09     _start:
A10         call  f
A11         movl  $1, %eax
A12         xorl  %ebx, %ebx
A13         int   $0x80
```

仍然通过 ***objdump*** 命令查看重定位信息：

```
$as  -o  0301c.o  0301c.s
$objdump  -xd  0301c.o

...
Disassembly of section .text:
00000000 <f>:
0:  c3                ret

00000001 <_start>:
 1: e8 fc ff ff ff    call   2 <_start+0x1>
```

```
                    2: R_386_PC32   f
 6: b8 01 00 00 00   mov    $0x1,%eax
 b: 31 db            xor    %ebx,%ebx
 d: cd 80            int    $0x80
```

这次编译器明确指出重定位时链接程序应该使用 f 的地址值，不过，要注意两点：

▪ 从“_start”起算偏移1个字节的4个字节值是0xfffffffc，这个值仅仅是为了指出这4个字节的起始地址距离 call 指令后面那条指令地址的偏移量是(-4)，除此之外没有任何意义，因此 ***objdump*** 把 call 指令反汇编为：

```
call   2 <_start+0x1>
```

我们无须理会，这只是“f”的值未进行最后确定的暂时现象。

▪ 由于 P6处理器 call 指令的要求，链接程序最后填入那4个字节的不是“f”的值，而是“f”相对于 mov 指令的偏移量。但是，这和前面“f”是局部符号、call 指令直接通过固定的偏移量调用“f”是两回事。因为当“f”是全局符号时，偏移量必须要等到链接程序给“f”分配了值（即“f”所代表的代码的地址）之后才能计算确定；而当“f”是局部符号时，对应的偏移量一直是固定的，无须等到最后链接程序来计算确定。

至此，我们已经考察了数据与代码在不同情况下的重定位，是时候解决疑问了。为什么编译器会根据符号的链接性质（全局的或局部的）分别作出不同的重定位要求呢？

首先，我们必须明白，无论是全局符号还是局部符号，链接程序为它们确定地址的方式是相同的。以数据的重定位为例，假设我们需要链接两个目标代码文件 A.o 和 B.o 来生成最后的可执行文件，很自然，A.o 有自己的“.data”区，B.o 也有自己的“.data”区。当进行链接时，链接程序首先通过计算得出可执行文件的“.data”区的大小，这个总的“.data”区必须可以容纳 A.o 的所有“.data”区数据以及 B.o 的所有“.data”区数据。然后链接程序为这个总的“.data”区确定合适的起始地址，并同时得到 A.o、B.o 各自的“.data”区相对于总的“.data”区的正确偏移量。由于 A.o、B.o 各自的数据相对于各自“.data”区的偏移量早在编译阶段就已经被编译器计算好，因此到这个时候链接程序完全可以计算出所有数据的最终地址。譬如，A.o 里面有一个名字为“len”的符号，它代表一个4字节数据，而 A.o 的“.data”区相对于总的“.data”区的偏移量是 *X*，“len”在 A.o 的“.data”区中的偏移量是 *Y*，总的“.data”区的起始地址是 *D*，那么符号“len”的值（即它代表的数据的地址）就是(*D*+*X*+*Y*)。

编译器对全局符号和局部符号分别作出不同的重定位指示是为了实现一个原则：如果出现相同名字的局部符号和全局符号，则链接程序应选择局部符号进行链接。假设有两个文件 A.o 和 B.o 参与链接，A.o 的“.data”区里有一个数据，该数据用局部符号“len”代表，“len”在 A.o 的“.data”区中的偏移量是0x00000010；而 B.o 的“.data”区也有一个数据，该数据用全局符号“len”代表，现在，A.o 的代码访问“len”：

```
mov  len, %edx
```

显然，这里访问的“len”应该被理解为 A.o 自己的“len”而不是 B.o 的“len”，只要编译器给出的重定位指示是下面这个样子：

```
... : 8b 15 10 00 00 00          mov    0x10, %edx
                                 ... : R_386_32    .data
```

链接程序就可以保证 A.o 的代码所访问的确实是 A.o 自己的“len”。

我们都知道：

```
R_386_32    .data
```

的意思是通知链接程序用该模块（即 A.o）的“.data”区起始地址加上偏移量0x00000010从而得到正确的最终地址完成重定位。很清楚，A.o 的“.data”区起始地址（即总的“.data”区起始地址加上 A.o“.data”区相对于总“.data”区起始地址的偏移量）加上偏移量0x00000010所得到的地址必然属于 A.o 自己的“len”，而绝对不会是 B.o 的“len”！因此，代码最后访问的只能是 A.o 自己的“len”。

03-01是一个很小的程序，代码和数据只占用很少的空间，仅仅由 ***objdump*** 命令的输出我们可以知道，进程自身的代码和数据只占用2个内存页面，如果程序代码访问了其它没有分配给进程的内存页面又会发生什么事呢？

※ ***objdump*** 命令不可能显示出栈页面的信息，事实上，系统把程序载入内存时还要为进程的栈分配页面。对于03-01，系统给进程初始分配的虚拟地址空间一共由3个页面组成，它们分别是：

0x08048000~0x08048FFF（存放代码）

0x08049000~0x0804A000（存放数据）

0xBFFFF000~0xBFFFFFFF（进程的栈）

图03-01描绘了进程的虚拟地址空间。 ※

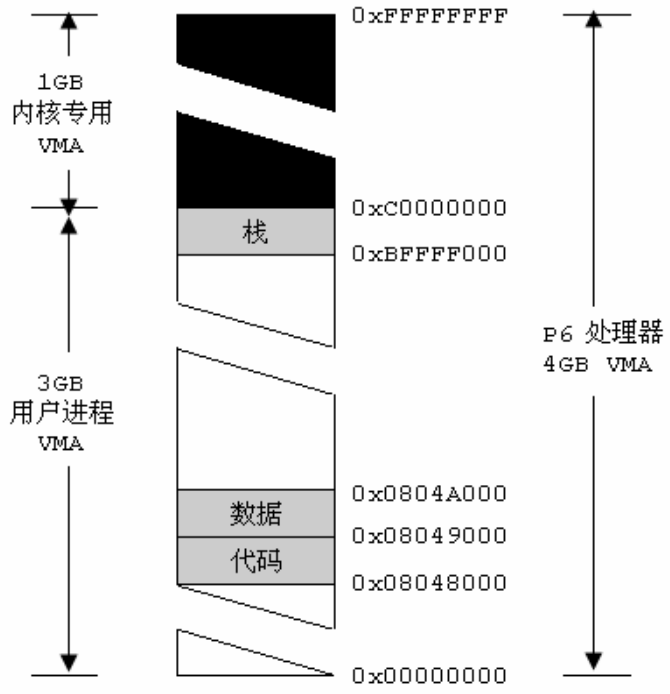


图 03-01

我们把03-01稍稍改动一下：

```
A01     # Code 03-02, file name: 0302.s
A02         .data
A03         .align 4
A04     msg:
A05         .string "Hello, Linux!\n"
A06         .align 4
A07     len:
A08         .long 14
A09         .text
A10     .globl _start
A11         .align 4
A12     _start:
A13         movl  $4, %eax
A14         movl  $1, %ebx
A15         movl  $msg, %ecx
A16         movl  len, %edx
A17         int   $0x80
A18         movl  $msg, %eax
A19         andl  $0xfffff000, %eax
A20         addl  $4096, %eax
A21         movl  (%eax), %ebx
A22         movl  $1, %eax
A23         xorl  %ebx, %ebx
A24         int   $0x80
```

同03-01一样，03-02首先会打印一个字符串（A13~A17），然后它试图访问 msg 所在页面的相邻下一个页面首地址的数据（A18~A21）。假设 msg 的值是0x08049094，那执行完 A18后，eax 的值就是0x08049094，A19会把 eax 的低12位清零，这时，eax 的值变为0x08049000，然后 A20会使 eax 的值变成0x0804A000，A21指示处理器把地址为0x0804A000的4个字节读入 ebx。编译、链接、执行03-02：

```
$as  -o  0302.o  0302.s
$ld  0302.o
$./a.out
Hello, Linux!
Segmentation fault
```

操作系统首先察觉到进程试图读取的数据所在的（物理内存）页面不存在，于是进入异常处理，然后异常处理程序会进一步发现对应的虚拟内存页面根本就没有分配给进程，

这时进程会接收到系统发出的信号，由于我们在03-02中没有对该信号进行任何处理，所以系统采取了默认的行动——强制终止进程的运行。前面已经说过，虽然32位处理器通常拥有4GB 的虚拟内存空间,但这并不是说每个进程都可以随便访问4GB 地址空间里面的任意一个地址。操作系统会根据进程的实际需要分配相应数量的虚拟地址空间以及真正的物理内存空间，03-02很好地印证了这一点。现在再来看看下面这个更简单的程序：

```
A01     # Code 03-03, file name: 0303.s
A02         .section  .rodata
A03         .align 4
A04     var:
A05         .long 0
A06         .text
A07     .globl _start
A08         .align 4
A09     _start:
A10         movl  $1, %eax
A11         xorl  %ebx, %ebx
A12         int   $0x80
```

03-03只有一个数据：var，而且没有任何访问这个数据的代码。但这里仍然有特别之处：var 存放的区域是“.rodata”（Read-Only DATA）区，顾名思义，存放在这个区的数据都是只读不可写的。我们先用 ***objdump*** 看个究竟：

```
$as  -o  0303.o  0303.s
$ld  0303.o
$objdump  -xd  a.out

a.out:     file format elf32-i386
a.out
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x08048074

Program Header:
 LOAD off 0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
  filesz  0x00000084 memsz 0x00000084 flags r-x
 LOAD off 0x00000084 vaddr 0x08049084 paddr 0x08049084 align 2**12
  filesz  0x00000000 memsz 0x00000000 flags rw-

Sections:
Idx Name     Size        VMA         LMA         File off    Algn
0   .text    00000009    08048074    08048074    00000074    2**2
```

```
                CONTENTS, ALLOC, LOAD, READONLY, CODE
1   .rodata     00000004    08048080    08048080    00000080    2**2
                CONTENTS, ALLOC, LOAD, READONLY, DATA
2   .data       00000000    08049084    08049084    00000084    2**2
                CONTENTS, ALLOC, LOAD, DATA
3   .bss        00000000    08049084    08049084    00000084    2**2
                ALLOC

SYMBOL TABLE:
08048074      l    d   .text     00000000
08048080      l    d   .rodata   00000000
08049084      l    d   .data     00000000
08049084      l    d   .bss      00000000
08048080      l        .rodata   00000000  var
08048074      g        .text     00000000  _start

Disassembly of section .text:
08048074 <_start>:
8048074:      b8 01 00 00 00       mov    $0x1,%eax
8048079:      31 db                xor    %ebx,%ebx
804807b:      cd 80                int    $0x80
```

仔细观察上面的输出信息，我们可以知道只读数据被安排存放在代码页（没错，是代码页!），代码页本身的属性是“r-x”，由于正常情况下程序代码根本不会去“执行”存放在代码页的只读数据，所以把只读数据放在代码页即可节省一部分空间（不用为之专门分配数据页）又能达到保证数据不被修改的目的（代码页是不可写的）。如果进程企图修改只读数据区的数据会怎样？做个小试验：

```
A01     # Code 03-04, file name: 0304.s
A02         .section  .rodata
A03         .align 4
A04     var:
A05         .long 0
A06         .text
A07     .globl _start
A08         .align 4
A09     _start:
A10         movl  $1, var
A11         movl  $1, %eax
A12         xorl  %ebx, %ebx
A13         int   $0x80
```

A10企图修改 var 的值，我们编译、链接，执行03-03：

```
$as  -o  0304.o  0304.s
$ld  0304.o
$./a.out
Segmentation fault
```

系统发现进程修改只读页面的数据，马上进入相应的异常处理，并进一步知道进程的确是违规访问，于是给进程发送信号，03-04没有捕获这个信号进行处理，所以系统按照默认方式采取行动——立即终止进程。

最后，我们再看看“.bss”区和“.data”区有什么不同：

```
A01     # Code 03-05, file name: 0305.s
A02         .bss
A03         .align 4
A04     array:
A05         .zero 4096
A06         .text
A07     .globl _start
A08         .align 4
A09     _start:
A10         movl  $1, array
A11         movl  $1, %eax
A12         xorl  %ebx, %ebx
A13         int  $0x80
```

与前面的例子不同，03-05没有把任何数据放在“.data”区，但在“.bss”区却占有一个4096字节的数据块（A03~A6），并且这个数据块的所有字节在进程刚开始运行时的初值都是0。

编译、链接03-05：

```
$as  -o  0305.o  0305.s
$ld  0305.o
$ls  -l  a.out
-rwxr-xr-x   1   yxy    users       690    Jun 23 19:55   a.out*
$objdump  -xd  a.out

a.out:    file format elf32-i386
a.out
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x08048074
```

```
Program Header:
 LOAD off 0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
   filesz  0x00000087 memsz 0x00000087 flags r-x
 LOAD off 0x00000088 vaddr 0x08049088 paddr 0x08049088 align 2**12
   filesz  0x00000000 memsz 0x00001000 flags rw-

Sections:
Idx Name   Size       VMA        LMA        File off    Algn
0   .text  00000013   08048074   08048074   00000074    2**2
           CONTENTS, ALLOC, LOAD, READONLY, CODE
1   .data  00000000   08049088   08049088   00000088    2**2
           CONTENTS, ALLOC, LOAD, DATA
2   .bss   00001000   08049088   08049088   00000088    2**2
           ALLOC

SYMBOL TABLE:
08048074      l    d    .text     00000000
08049088      l    d    .data     00000000
08049088      l    d    .bss      00000000
08049088      l         .bss      00000000    array
08048074      g         .text     00000000    _start


Disassembly of section .text:
08048074 <_start>:
 8048074:     c7 05 88 90 04 08 01      movl   $0x1,0x8049088
 804807b:     00 00 00
 804807e:     b8 01 00 00 00            mov    $0x1,%eax
 8048083:     31 db                     xor    %ebx,%ebx
 8048085      cd 80                     int    $0x80
```

上面的输出结果很有趣，文件“a.out”的大小才690字节，但其中的“.bss”区却有一个4096字节的数据块。这其实就是问题的答案：“.bss”区的数据并不保存在文件中，系统是在载入程序的时候根据有关信息给进程的“.bss”区数据分配页面的。什么样的数据可以放在“.bss”区？——没有初始值的那些数据。既然没有初始值，那我们就不用浪费文件空间去保存它们（“.data”区的数据都有初始值，所以要在文件中开辟相应的空间保存那些初始值）。留意一下“Program Header”的第2段：“.bss”区的“filesz（在文件中的大小）”是0x00000000而“memsz（在内存中的大小）”是0x00001000。

※ 在很多平台(包括 Linux)上,系统给“.bss”区分配内存页面后会马上往这些页面“写零”,“.bss”区的数据实际上全部被初始化为“0”。 ※

由于“.bss”区的数据大小不能根据文件的实际大小来判断,所以我们需要用 ***size*** 命令去观察可执行文件的“.data”、“.bss”、“.text”区在内存一共占用多少空间:

```
$size  a.out
   text     data      bss      dec        hex    filename
     19        0     4096     4115       1013    a.out
$
```

* * * * * *

[0301]

LMA(Logical Memory Address)指逻辑地址;VMA(Virtual Memory Address)指虚拟地址。在后面我们可以发现, 对于 P6/Linux 平台它们的值是相同的。

[0302]

请联系上下文来理解“.data”、“.text”和“.bss”这些词语,有时它们代表一个区域,有时它们也(很自然地)意味着该区域的起始地址。

[0303]

虽然真正的指令是从0x08048074开始,但在这里 ***objdump*** 是从页的起始地址0x08048000开始算起,所以指令部分的大小就变成了0x94个字节。

[0304]

可能你会奇怪:难道数据在文件里的大小会和在内存里的大小不一样吗?是的,对于那些存放在“.bss”区的数据来说的确如此。

04 编译、链接和库

虽然到目前为止我们的示例程序都是基于单个文件的，但其实在上一节我们已经涉及到关于多个文件分别进行编译然后链接产生可执行文件的部分讨论。的确，在真正的软件开发中，程序的规模都很大，一个人不可能独自完成所有的代码，因此，由不同的人分别编写不同的模块然后各自进行编译，最后再根据需要进行必要的链接是唯一的选择。这里我们仍然用例子来示范这个过程：

```
A01      # Code 04-01A, file name: 0401a.s
A02          .text
A03      .globl _start
A04          .align 4
A05      _start:
A06          call  main
A07          movl  $1, %eax
A08          xorl  %ebx, %ebx
A09          int  $0x80
```

```
A01      # Code 04-01B, file name: 0401b.s
A02          .text
A03      .globl main
A04          .align 4
A05      main:
A06          movl  $4, %eax
A07          movl  $1, %ebx
A08          movl  $msg, %ecx
A09          movl  len, %edx
A10          int  $0x80
A11          ret
```

```
A01      # Code 04-01C, file name: 0401c.s
A02          .data
A03      .globl msg
A04          .align 4
A05      msg:
A06          .string “Hello, Linux!\n”
A07      .globl len
A08          .align 4
A09      len:
A10          .long 14
```

上面三个汇编代码文件完成的功能和03-01完全相同，只不过把代码部分和数据定义部分分开，代码部分包括两个文件：04-01A 的 A06通过 call 指令调用 main（main 代表某条指令的地址）；而04-01B 则利用04-01C 的数据进行系统调用（A06~A10）以完成字符串的显示，然后通过 A11的 ret 指令返回04-01A，这时04-01A 再通过一次系统调用（A07~A09）结束进程。

请注意，我们这样的表达完全是基于源代码的角度。一旦04-01A、04-01B、04-01C 各自编译完毕、链接程序完成链接过程生成最后的可执行文件之后，由于所有指令和数据都在同一个文件（a.out）中，那时就不适宜再用“谁的指令”、“谁的数据”之类的词语。

下面我们详细查看04-01系列的目标代码文件：

```
$as  -o  0401a.o  0401a.s
$objdump  -xd  0401a.o

0401a.o:     file format elf32-i386
0401a.o
architecture: i386, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x00000000

Sections:
Idx Name   Size         VMA        LMA        File off    Algn
0   .text   0000000e     00000000   00000000   00000034    2**2
            CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
1   .data   00000000     00000000   00000000   00000044    2**2
            CONTENTS, ALLOC, LOAD, DATA
2   .bss    00000000     00000000   00000000   00000044    2**2
            ALLOC

SYMBOL TABLE:
00000000   l   d   .text    00000000
00000000   l   d   .data    00000000
00000000   l   d   .bss     00000000
00000000   g       .text    00000000   _start
00000000           *UND*    00000000   main

Disassembly of section .text:
00000000 <_start>:
 0: e8 fc ff ff ff         call   1 <_start+0x1>
                     1: R_386_PC32   main
 5: b8 01 00 00 00         mov    $0x1,%eax
 a: 31 db                  xor    %ebx,%ebx
```

```
 c: cd 80                  int    $0x80

RELOCATION RECORDS FOR [.text]:
OFFSET     TYPE               VALUE
00000001   R_386_PC32         main
```

对以上输出信息我们已经相当熟悉：

▪ 符号表有一项是关于“main”的。由于04-01A 用 call 指令调用 main，但在04-01A 中又没有任何关于 main 的定义，所以符号表中 main 的属性为“UND”（undefined）。

▪ 有一个地方需要重定位，相对于“_start”偏移为1字节的连续4个字节稍后应由链接程序用正确的偏移值填入。

再来看看04-01B 的目标代码：

```
$as  -o  0401b.o  0401b.s
$objdump  -xd  0401b.o

0401b.o:     file format elf32-i386
0401b.o
architecture: i386, flags 0x00000011:
 HAS_RELOC, HAS_SYMS
start address 0x00000000

Sections:
Idx Name   Size      VMA         LMA         File off  Algn
0   .text  00000018  00000000    00000000  00000034  2**2
           CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
1   .data  00000000  00000000    00000000  0000004c  2**2
           CONTENTS, ALLOC, LOAD, DATA
2   .bss   00000000  00000000    00000000  0000004c  2**2
           ALLOC

SYMBOL TABLE:
00000000   l   d    .text   00000000
00000000   l   d    .data   00000000
00000000   l   d    .bss    00000000
00000000   g        .text   00000000  main
00000000            *UND*   00000000  msg
00000000            *UND*   00000000  len

Disassembly of section .text:
```

```
00000000 <main>:
 0: b8 04 00 00 00          mov    $0x4,%eax
 5: bb 01 00 00 00          mov    $0x1,%ebx
 a: b9 00 00 00 00          mov    $0x0,%ecx
                      b: R_386_32    msg
 f: 8b 15 10 00 00 00       mov    0x10,%edx
                     11: R_386_32    len
15: cd 80                   int    $0x80
17: c3                      ret
```

由于“msg”和“len”都没有在04-01B 里面进行定义，所以在符号表这两个符号的属性是“*UND*”，并且对于全局符号（“msg”和“len”只能是全局符号），编译器指示链接程序在重定位时必须用它们的地址分别填入从“main”开始偏移0xb、0x11的两处地方。

至于04-01C 的目标代码就相对简单些，该文件仅仅包含两个全局符号的定义：

```
$as  -o  0401c.o  0401c.s
$objdump  -xd  0401c.o

0401c.o:    file format elf32-i386
0401c.o
architecture: i386, flags 0x00000010:
HAS_SYMS
start address 0x00000000

Sections:
Idx Name  Size       VMA         LMA         File off    Algn
0  .text  00000000   00000000    00000000    00000034    2**2
          CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
1  .data  00000014   00000000    00000000    00000034    2**2
          CONTENTS, ALLOC, LOAD, DATA
2  .bss   00000000   00000000    00000000    00000048    2**2
          ALLOC

SYMBOL TABLE:
00000000  l   d   .text  00000000
00000000  l   d   .data  00000000
00000000  l   d   .bss   00000000
00000000  g       .data  00000000    msg
00000010  g       .data  00000000    len
```

我们最后完成链接，查看可执行文件：

```
$ld  0401a.o  0401b.o  0401c.o
$objdump  -xd  a.out

a.out:     file format elf32-i386
a.out
architecture: i386, flags 0x00000112:
 EXEC_P, HAS_SYMS, D_PAGED
start address 0x08048074

Program Header:
  LOAD off 0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
    filesz  0x0000009c memsz 0x0000009c flags r-x
  LOAD off 0x0000009c vaddr 0x0804909c paddr 0x0804909c align 2**12
    filesz  0x00000014 memsz 0x00000014 flags rw-

Sections:
Idx Name  Size       VMA         LMA       File off   Algn
0   .text 00000028   08048074    08048074  00000074   2**2
          CONTENTS, ALLOC, LOAD, READONLY, CODE
1   .data 00000014   0804909c    0804909c  0000009c   2**2
          CONTENTS, ALLOC, LOAD, DATA
2   .bss  00000000   080490b0    080490b0  000000b0   2**2
          ALLOC

SYMBOL TABLE:
08048074  l   d   .text    00000000
0804909c  l   d   .data    00000000
080490b0  l   d   .bss     00000000
0804909c  g       .data    00000000  msg
08048074  g       .text    00000000  _start
08048084  g       .text    00000000  main
080490ac  g       .data    00000000  len

Disassembly of section .text:
08048074 <_start>:
 8048074:  e8 0b 00 00 00        call   8048084 <main>
 8048079:  b8 01 00 00 00        mov    $0x1,%eax
 804807e:  31 db                 xor    %ebx,%ebx
 8048080:  cd 80                 int    $0x80
```

```
08048084 <main>:
 8048084:  b8 04 00 00 00        mov    $0x4,%eax
 8048089:  bb 01 00 00 00        mov    $0x1,%ebx
 804808e:  b9 9c 90 04 08        mov    $0x804909c,%ecx
 8048093:  8b 15 ac 90 04 08     mov    0x80490ac,%edx
 8048099:  cd 80                 int    $0x80
 804809b:  c3                    ret
```

各个符号都已经有了确定的值，无论是“_start”还是“main”里面的代码都已经完成重定位。0x08048074的 call 指令由于 main 的值为0x08048084，所以原本为0xfffffffc 的地方现在填入了正确的偏移0x0000000b。[1]

对于代码和数据分布在多个文件的程序来说，链接程序要在这多个文件中寻找相应的符号，这些符号要么是：

- 全局可见的，例如04-01B 的“main”以及04-01C 的“msg”和“len”。当链接程序在某个模块发现了对全局符号的引用之后，它就会在所有参与链接的目标代码文件的符号表里寻找该符号。
- 局部可见的，具有局部链接属性的符号只能被本文件的代码引用，而且对于同名的符号，局部符号优先于全局符号。

在汇编代码里面，要把符号设置为全局可见，只需明确加上“.globl xxx”（假设xxx 为符号名），例如：

```
    .data
.globl varg
varg:
    .long 2
```

上面的“varg”就是全局符号。在 ***objdump*** 命令输出的符号表中，全局符号都用“g”进行标识：

```
SYMBOL TABLE:
...        g      .data    ...         varg
```

而局部可见的符号则用“.local xxx”语句表示（如果什么语句都没有则系统默认符号是局部可见的），例如：

```
    .data                          .data
    .local varl                 varl:
varl:                  或者
    .long 3                        .long 3
```

“varl”是局部符号，在符号表用“l”表示：

```
SYMBOL TABLE:
...        l      .data    ...         varl
```

除了 ***objdump*** 之外，我们还可以使用 ***nm*** 命令来查看目标代码文件、可执行文件、库文件的符号表，例如对于04-01最后生成的可执行文件来说，有：

```
$nm  a.out
...
08048074  T  _start
080490ac  D  len
08048084  T  main
0804909c  D  msg
```

对于每一行输出，第一项是符号的值（即符号所代表的数据或指令的地址）；第二项是符号的属性（全局的还是局部的、代表代码还是指令）；第三项是符号的名字。

暂时来说，我们只需了解6种属性：T、t、D、d、B、b。T 和 t 表示该符号代表指令，这些指令无疑肯定属于“.text”区；D 和 d 表示该符号代表有初始值数据，这些数据属于“.data”区；B 和 b 表示该符号代表无初始值数据，这些数据属于“.bss”区。同时，大写字母表示全局符号，小写字母表示局部符号的。对于有定义但仍未能确定准确值的符号（譬如目标代码的那些符号），第一项的值就用它们在各自区域的偏移量代替：

```
$nm  0401c.o
00000010  D  len
00000000  D  msg
```

至于被引用到、但没有进行定义的符号则用“U”表示，并且地址一项为空白：

```
$nm  0401a.o
00000000  T  _start
          U  main
```

链接程序需要依靠目标代码文件的符号表给出相应的信息来完成链接过程，因此，对于目标代码文件（包括后面讨论的库文件）来说，符号表是绝对不可缺少的。对于可执行文件，由于已经完成重定位，所以我们可把符号表部分去掉以减少程序占用的磁盘空间：

```
$ls  -l  a.out
-rwxr-xr-x    1   yxy     users        769  Jun  26  10:34  a.out*
$strip  a.out
$ls  -l  a.out
-rwxr-xr-x    1   yxy     users        404  Jun  26  17:49  a.out*
```

可以看到，使用 ***strip*** 命令去掉程序的符号表后，程序占用的磁盘空间大为减少[2]。不过，千万不要对目标代码文件和库文件进行 ***strip***，没有了符号表它们只是废物而已。

从上面我们了解到，如果软件的规模比较大，那我们可以把程序分为多个模块，这些模块可以是数据、代码或者两样都有。总之，当所有模块都编译成目标代码文件：

```
1.o  2.o  3.o  ...  n.o
```

我们就可以用链接程序生成最后的可执行文件：

```
ld  1.o  2.o  3.o  ...  n.o
```

这便引出另一个问题：如果有某些代码通用性比较强，几乎每次都要用到，譬如说1.o里的代码 T1和2.o 里的代码 T2很常用，那我们有什么办法既可以重复使用 T1、T2又不会使代码的管理复杂化。

显然，每次链接时都把1.o 和2.o 加进去是行不通的。因为1.o 和2.o 可能会引用到其它符号，链接程序必须逐个地为那些被引用到的符号进行链接，而这些符号的定义很可能是位于3.o、4.o……于是我们不断地把其它模块加入链接命令来尝试，直到链接程序为所有被引用到的符号找到定义之处为止。这样的工作方式效率很低，即使1.o 和2.o 没有其它符号需要链接，但我们仍然需要记住：T1在1.o、T2在2.o，当用到 T1就把1.o 加入链接命令、用到 T2就把2.o……万一以后可重复使用的代码多起来——譬如多达上百个时，我们又怎可能全部记住它们位于哪个文件呢？为了解决这些问题，人们引入了库文件协助程序员高效地使用可重复代码。

库（library）有两种：静态库和动态库。本节我们主要讨论静态库。

静态库其实就是一个包含了多个目标代码文件的大文件，例如我们可以把1.o、2.o、3.o……全部放入一个静态库文件，当我们需要用到 T1时，通过链接命令告诉链接程序到该库文件寻找 T1就可以了。链接程序会：

- 自动搜索库文件以找出被引用到的符号的定义之处并完成相应的链接
- 忽略某些没有被引用到的符号

下面首先给大家示范一下静态库的创建和使用，例如下面有4个模块：

04-02调用04-02B 定义的 main；

04-02B 读取04-02A 定义的 var；

04-02C 定义了 fun，但谁也没有调用它；

```
A01         # Code 04-02, file name: 0402.s
A02             .text
A03         .globl _start
A04             .align 4
A05         _start:
A06             call  main
A07             movl  $1, %eax
A08             int  $0x80

A01         # Code 04-02A, file name: 0402a.s
A02             .data
A03             .align 4
A04         .globl var
A05         var:
A06             .long 2

A01         # Code 04-02B, file name: 0402b.s
A02             .text
A03             .align 4
```

```
A04         .globl main
A05         main:
A06             movl  var, %ebx
A07             ret

A01         #Code 04-02C, file name: 0402c.s
A02             .text
A03             .align 4
A04         .globl fun
A05         fun:
A06             xorl  %eax, %eax
A07             ret
```

我们编译04-02系列所有文件：

```
$as  -o  0402.o  0402.s
$as  -o  0402a.o  0402a.s
$as  -o  0402b.o  0402b.s
$as  -o  0402c.o  0402c.s
```

我们把0402a.o、0402b.o、0402c.o 放入一个静态库文件 lib0402.a 中：

```
$ar  rc  lib0402.a  0402a.o  0402b.o  0402c.o
```

上面的 ar 命令创建一个名为“lib0402.a”的静态库文件，这个文件里面包含有0402a.o、0402b.o、0402c.o 等3个目标代码文件。

先用 ***nm*** 命令看看库文件的符号表：

```
$nm  lib0402.a
0402a.o:
00000000  D  var
0402b.o:
00000000  T  main
          U  var
0402c.o:
00000000  T  fun
```

库文件内每个模块的符号表被依次输出，从这样的信息中你是否猜到库文件的实质呢？用 ***objdump*** 命令看一下：

```
$objdump  -x  lib0402.a
In archive lib0402.a

0402a.o:     file format elf32-i386
rw-r--r--   1000/100     465   Jun 27 16:55 2003   0402a.o
architecture: i386, flags 0x00000010:
```

```
 HAS_SYMS
start address 0x00000000

Sections:
Idx Name  Size      VMA       LMA       File off  Algn
0   .text 00000000  00000000  00000000  00000034  2**2
          CONTENTS, ALLOC, LOAD, READONLY, CODE
1   .data 00000004  00000000  00000000  00000034  2**2
          CONTENTS, ALLOC, LOAD, DATA
2   .bss  00000000  00000000  00000000  00000038  2**2
          ALLOC

SYMBOL TABLE:
00000000   l   d   .text  00000000
00000000   l   d   .data  00000000
00000000   l   d   .bss   00000000
00000000   g       .data  00000000    var

0402b.o:    file format elf32-i386
rw-r--r--   1000/100    544   Jun 27 16:56 2003   0402b.o
architecture: i386, flags 0x00000011:
 HAS_RELOC, HAS_SYMS
start address 0x00000000

Sections:
Idx Name   Size      VMA       LMA       File off  Algn
0   .text  00000007  00000000  00000000  00000034  2**2
           CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
1   .data  00000000  00000000  00000000  0000003c  2**2
           CONTENTS, ALLOC, LOAD, DATA
2   .bss   00000000  00000000  00000000  00000038  2**2
           ALLOC

SYMBOL TABLE:
00000000   l  d  .text    00000000
00000000   l  d  .data    00000000
00000000   l  d  .bss     00000000
00000000   g     .text    00000000  main
00000000         *UND*    00000000  var
```

```
0402c.o:     file format elf32-i386
rw-r--r--   1000/100    465   Jun 27 16:58 2003   0402c.o
architecture: i386, flags 0x00000010:
HAS_SYMS
start address 0x00000000

Sections:
Idx Name   Size       VMA       LMA       File off  Algn
0   .text  00000003   00000000  00000000  00000034  2**2
           CONTENTS, ALLOC, LOAD, READONLY, CODE
1   .data  00000000   00000000  00000000  00000038  2**2
           CONTENTS, ALLOC, LOAD, DATA
2   .bss   00000000   00000000  00000000  00000038  2**2
           ALLOC

SYMBOL TABLE:
00000000  l   d   .text  00000000
00000000  l   d   .data  00000000
00000000  l   d   .bss   00000000
00000000  g       .text  00000000   fun
```

正如大家看到的那样，静态库文件就是这么简单：它相当于把多个目标代码文件集中存放在一个更“大”的文件里面。另外，库文件会为每个模块维护一个时间戳，以便日后可以对照文件记录的时间来确定文件的新旧从而决定是否更新模块。[3]

有了库文件 lib0402.a，我们就可以这样进行链接：

```
$ld  0402.o  lib0402.a
```

或者：

```
$ld  0402.o  -L.  -l0402
```

上面的命令假定0402.o 和 lib0402.a 在同一目录下。前者直接给出库文件名进行链接，后者指示 ***ld*** 在当前目录（“.”代表当前目录）下搜索库文件（该文件的名字是“lib0402.a”或者“lib0402.so”）。

链接成功后我们再用 ***nm*** 命令观察可执行文件 a.out 的符号表：

```
$nm  a.out
...
08048074  T  _start
08048080  T  main
08049088  D  var
```

0402.o 的“_start”引用04-02b.o 的“main”，而“main”又引用04-02a.o 的“var”，所以在最后的 a.out 里面，“main”、“var”都被链接进来。而0403c.o 的“fun”没有被任何模块引用，因此它没有出现在 a.out 里面。用 ***objdump*** 命令了解更

详细的信息：

```
$objdump  -xd  a.out

a.out:    file format elf32-i386
a.out
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x08048074

Program Header:
  LOAD off 0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
    filesz  0x00000088 memsz 0x00000088 flags r-x
  LOAD off 0x00000088 vaddr 0x08049088 paddr 0x08049088 align 2**12
    filesz  0x00000004 memsz 0x00000004 flags rw-

Sections:
Idx Name   Size       VMA        LMA       File off  Algn
0   .text   00000014   08048074   08048074  00000074  2**2
           CONTENTS, ALLOC, LOAD, READONLY, CODE
1   .data   00000004   08049088   08049088  00000088  2**2
           CONTENTS, ALLOC, LOAD, DATA
2   .bss    00000000   0804908c   0804908c  0000008c  2**2
           ALLOC

SYMBOL TABLE:
08048074    l    d  .text  00000000
08049088    l    d  .data  00000000
0804908c    l    d  .bss   00000000
08048074    g       .text  00000000  _start
08048080    g       .text  00000000  main
08049088    g       .data  00000000  var

Disassembly of section .text:
08048074 <_start>:
 8048074:   e8 07 00 00 00          call   8048080 <main>
 8048079:   b8 01 00 00 00          mov    $0x1,%eax
 804807e:   cd 80                   int    $0x80

08048080 <main>:
 8048080:   8b 1d 88 90 04 08       mov    0x8049088,%ebx
```

```
8048086:   c3                        ret
```

“.text”区只占20（0x14）字节，“fun”的确没有进入 a.out，这和前面说过的“对库文件进行链接可以（在某种程度上）避免把无关的代码或数据复制到最终的可执行文件中”完全一致。

然而，事情还没有结束——静态库真的就能绝对避免把无关的数据或代码复制到可执行文件吗？

看看这个例子：

1.s

```
    .text
.globl _start
_start:
    call  f
    movl  $1, %eax
    int $0x80
```

2.s

```
    .text
.globl f
f:
    ret
.globl g
g:
    ret
```

我们在上面的2.s 里定义了“f”、“g”，而1.s 只引用“f “，让我们把2.s 制作成静态库然后进行链接：

```
$as  -o  1.o  1.s
$as  -o  2.o  2.s
$ar rc  libx.a  2.o
$ld  1.o  libx.a
$nm  a.out
...
08048074  T  _start
0804807c  T  f
0804807d  T  g
```

即使没有被引用，“g”还是被复制到可执行文件了。

链接程序对静态库的链接操作其实是非常简单的，前面已经说过，一个静态库文件就是多个目标代码文件组成的大文件。在链接的时候，链接程序从库里面的第一个模块开始寻找对应的符号定义，找到的话就把该符号所在的目标代码文件的 “.text”区、“.data”区、“.bss”区（当然也包括符号表的所有对应内容）全部复制到可执行文件并计算出正确的地址。上面的符号“g”以及它所代表的指令即使没有被任何模块引用也同样进入到最后的可执行文件完全是因为处于同一模块的“f”被引用过。

使用静态库虽然可以避免把位于不同目标代码文件的无关内容复制到可执行文件，但不能避免把位于同一目标代码文件的无关内容链接进去，要完全避免把无关代码、数据链接入程序文件的话我们必须使用动态库。

另外，我们使用静态库时还要注意两点：

▪ 注意链接命令的参数顺序。

链接程序对静态库的链接有一个非常笨拙的特点：如果链接程序第一次遇到某个静态库文件而又没有在该库中搜索到任何需要的符号则它会在随后的搜索中忽略该静态库。譬如在04-02的示例中，如果我们这样链接：

```
$ld  lib0402.a  0402.o
```

链接程序就会报告说找不到符号“main”。因为在上面的链接命令里，我们把静态库文件 lib0402.a 放在0402.o 的前面，当 *ld* 第一次碰到 lib0402.a 时，系统连哪些符号需要链接都未知道，自然不会在 lib0402.a 里面找到任何需要的符号，于是 *ld* 在以后的链接活动中忽略 lib0402.a，即使稍后的0402.o 有一个对“main”的引用而“main”其实就在 lib0402.a 里，*ld* 还是报告说找不到“main”。

再举一例，假设有3个文件的符号表如下所示：

x.o

```
00000000  T  _start
          U  f
```

liby.a

```
00000000  T  g
```

libz.a

```
00000000  T  f
          U  g
```

x.o 引用“f”，“f”的定义在静态库 libz.a 里，而假设“f”又引用了“g”，“g”的定义在静态库 liby.a 中。如果这样链接：

```
$ld  x.o  liby.a  libz.a
```

则链接程序会报告找不到“g”。因为 *ld* 第一次遇到 liby.a 时会搜索 x.o 里面所有 undefined 的符号（在这里只有一个“f”），显然它什么都找不到，于是 *ld* 决定在随后的搜索中忽略 liby.a，当 *ld* 为 libz.a 的“f”所引用的“g”寻找定义时已经没有可供搜索的库文件了，*ld* 只能报错退出。

其实只要换一下顺序就可以链接成功：

```
$ld  x.o  libz.a  liby.a
```

▪ 同一个库文件可以存在相同名字的模块或符号。

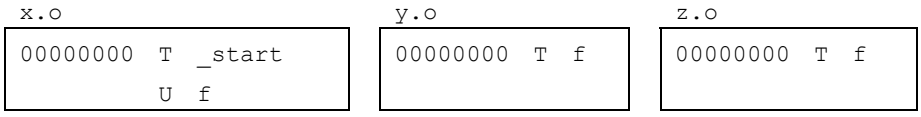
由于静态库仅仅是把目标代码集中存放在一起而其它的事一律不管，所以我们可以把相同名字的目标代码文件加进同一个库文件，也可以让同一个库文件的不同模块里出现相同名字的符号。仍然以04-02为例，譬如：

```
$rm  lib0402.a
$ar  rc  lib0402.a  0402a.o
$mv  0402b.o  0402a.o
$ar  q  lib0402.a  0402a.o
$mv  0402c.o  0402a.o
$ar  q  lib0402.a  0402a.o
$nm  lib0402.a
0402a.o:
00000000  D  var
0402a.o:
00000000  T  main
          U  var
```

```
0402a.o:
00000000  T  fun
$ld  0402.o  lib0402.a
```

上面先删除 lib0402.a，然后重新创建静态库文件 lib0402.a，首先把0402a.o加入库，接着把0402b.o 和0402c.o 都改名为0402a.o，再陆续加入库文件[4]，结果lib0402.a 有3个模块，3个模块的名字都是0402a.o。而且，我们最后用 *ld* 命令链接0402示例程序，结果竟然是成功的。

又或者，假设有3个目标代码文件：



x.o 引用“f”，而 y.o 和 z.o 都定义了“f”，现在把2.o 和3.o 同时加入同一个静态库文件 libf.a：

```
$ar  rc  libf.a  y.o  z.o
$ld  x.o  libf.a
```

链接依然成功。显然，无论是存在相同名字的模块，还是在不同模块之间存在相同名字的符号，*ld* 仍然按照通常的规则进行符号的搜索和链接。

上面的 libf.a 里面有两个模块 y.o、z.o，各自都有一个“f”可以被链接程序选中，但 y.o 在库文件中的位置比 z.o 要靠前，所以 *ld* 会先发现 y.o 的“f”，结果就是 y.o 的“f”被链接入可执行文件。如果创建库文件时所用的命令是这样：

```
$ar  rc  libf.a  z.o  y.o
```

则链接时被选中的就是 z.o 定义的“f”，因为这次 z.o 在 y.o 的前面，链接程序会先找到 z.o 定义的“f”。

静态库的这两个特点容易导致库的使用和管理上出现混乱，所以大家在使用静态库时要特别小心。

链接静态库的程序直接包含库代码的副本，文件体积较大，并且所有链接同一段库代码的程序所对应的进程都在内存中拥有一份该代码的副本，无论是磁盘空间还是内存空间都存在浪费。例如，程序 A、B、C 都链接静态库 L 的代码 F，现在用户启动 A、B、C，那么就有3个含有 F 的进程在运行，在这3个进程中的任何一个结束之前， 3份代码 F 同时占用着内存，如图04-01。[5]

为了节省磁盘和内存空间，人们推出了动态库作为解决方案。动态库其实也是由代码和数据组成，只不过链接程序不会把动态库的代码复制到程序文件内部，而是在运行时由操作系统负责把库代码载入内存并映射到进程的地址空间。还是用上面的例子，程序 A、B、C 都使用动态库 L 的代码 F，用户启动 A、B、C 之后，一共有3个进程使用 F，如图04-02。

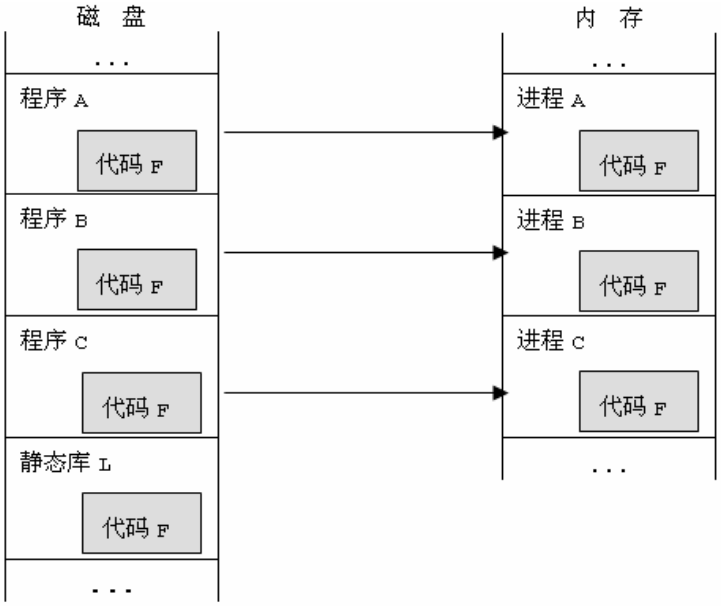


图 04-01

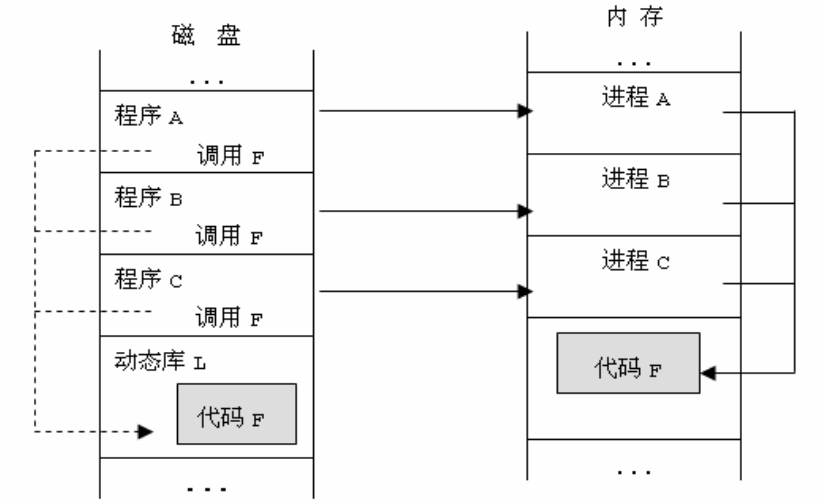


图 04-02

无论是在磁盘还是在内存，都只有一份代码 F。 在程序进行链接的时候，链接程序首先在动态库里寻找对应的符号，如果没有找到则链接失败，如果找到，则把符号及动态库文件名记录下来，链接程序绝对不会把库代码复制一份到可执行文件中去。

在用户运行程序的时候，操作系统根据可执行文件的相关信息知道进程需要调用哪些动态库的代码，它会先查找系统记录，如果内存中已经有相同的代码被载入，则根据这些代码的物理地址计算出合适的虚拟地址填入进程数据区的某张表，同时完成必要的地址映射（还记得 P6的页目录、页表吗？），进程根据那张表就能够正确调用代码；如果内存中并没有相同的代码，则操作系统先把库代码载入内存，接着再完成上述的步骤。

由于动态库技术的实现跟具体的编译器密切相关，每个编译平台都有自己的一套规则和做法，并且还涉及到大量系统定义的数据结构，所以我们一般不可能直接用汇编语言写动态库代码[6]，而必须用高级语言例如 C/C++编写，然后指示 C/C++编译器把 C/C++代码编译成可以用作动态库的机器代码。

和静态库相比，动态库有很明显的优势：

- 节省空间

这个特点在图04-02中已经表达得很清楚。

- 升级容易

由于静态库的代码被复制到可执行文件，所以当静态库文件更新时我们必须重新链接程序，以便把新的库代码复制到可执行文件。而动态库的特点则决定了即使库文件有改动我们也无须重新链接，因为动态库的代码根本没有被复制到可执行文件，当运行程序时，系统会直接从新的库文件载入新的库代码。

此外，动态库还有一个特点：同一个库文件里面不能有名字相同的符号。前面提到，静态库允许多个同名模块存在，并且在不同模块之间可以有同名符号。现在，动态库对这方面的限制更加严格，在一个动态库文件中，每个符号的名字都必须是独一无二的。[7]

更方便的是，我们在链接命令中可以任意安排动态库文件出现的顺序：

```
$ld liby.so x.o
```

如果是静态库，类似上面的链接命令多数会失败，因为库文件出现在目标代码的前面。但如果使用的是动态库，则顺序可以是任意的，凡是在命令行出现过的动态库，链接程序都会对它进行搜索。

* * * * * *

[0401]

call 指令的操作数可以是一个32位的偏移量但不能是一个直接的32位地址，所以这里要先算出 main（0x08048084）相对于 call 指令的下一条指令（0x08048079）的偏移（0x0000000b），然后用这个偏移作为 call 指令的操作数。

[0402]

由于磁盘的空间分配是以簇为单位的，虽然 a.out 大小由769字节减少到404字节，但仍然占用一个磁盘簇。这里所说的占用磁盘空间减少必须在一些体积相对大一点的程序才能获得实际效果。

[0403]

当我们用 ar 命令企图用某些目标代码文件更新库的某些模块时，系统就会对比两者的日期，如果库的模块的时间戳比目标代码文件要早，则进行替换，否则不进行更新。

[0404]

这里的 ar 命令用了“q”参数选项，目的是指示 ar 在模块加入时不要替换掉已经存在的同名模块。如果用“r”参数则 ar 会执行替换。

[0405]

如果很不幸代码 F 所在的目标代码文件里面不止 F 一段代码而是有其它好多段代码，则磁盘和内存的浪费就更加严重了。因为前面已经说过，链接程序会把静态库里面对应目标代码文件的“.text”、“.data”、“.bss”区全部内容都复制到可执行文件。

[0406]

如果不涉及对其它代码或数据的访问，那我们还是可以用汇编语言编写动态库代码。但对于复杂一点的情况，由于组成动态库的代码通常必须是位置无关代码（Position Independent Code，PIC），因此我们就不得不借助高级语言编译器了，手工编写 PIC 极其复杂、容易出错，。

[0407]

注意，这里说同一个动态库文件不允许出现相同的符号，但如果是在不同的动态库文件中，相同的符号是可以和平共处的。假设在一个链接命令中，有一个以上的动态库文件包含了符合要求的符号定义，则链接程序还是会根据“先找到先链接，然后忽略其余库文件”的原则进行处理，结果就是对该符号的引用被链接到命令行中最先出现的符合要求的动态库的对应符号定义处。

05 动态库简介

首先，我们用一个最简单的例子演示一下动态库的使用：

0501.s

```
    .text
.globl _start
_start:
    call  f
    movl  $1, %eax
    int   $0x80
```

0502.s

```
    .text
.globl f
    .type f, @function
f:
    ret
    .size f, .-f
```

0503.s

```
    .text
.globl g
    .type g, @function
g:
    ret
    .size g, .-g
```

0502.s 和0503.s 的代码非常简单，都是只有一条“ret”返回指令，唯一让我们感到陌生的是以下两条语句（“x”代表某个符号）：

```
.type x, @function
.size x, .-x
```

目前我们暂时不理会这些语句的用意何在，我们首先看看制作动态库文件的过程：

```
$as  -o  0502.o  0502.s
$as  -o  0503.o  0503.s
$ld  -shared  –o  libfg.so  0502.o  0503.o
```

ld 命令可以生成动态库文件，只要加上“-shared”参数选项，我们可以用 *objdump* 命令观察一下 libfg.so 是否还像静态库文件那样仅仅由若干个目标代码拼凑而成：

```
$objdump  -xd  libfg.so

libfg.so:    file format elf32-i386
libfg.so
architecture: i386, flags 0x00000150:
HAS_SYMS, DYNAMIC, D_PAGED
start address 0x00000224

Program Header:
    LOAD off 0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**12
     filesz  0x00000229 memsz 0x00000229 flags r-x
    LOAD off 0x0000022c vaddr 0x0000122c paddr 0x0000122c align 2**12
     filesz  0x00000064 memsz 0x00000064 flags rw-
 DYNAMIC off 0x0000022c vaddr 0x0000122c paddr 0x0000122c align 2**12
     filesz  0x00000058 memsz 0x00000058 flags rw-
Dynamic Section:
 HASH     0x94
 STRTAB    0x1e8
```

```
 SYMTAB    0xe8
 STRSZ     0x3c
 SYMENT    0x10

Sections:
Idx Name          Size        VMA         LMA         File off    Algn
0   .hash         00000054    00000094    00000094    00000094    2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
1   .dynsym       00000100    000000e8    000000e8    000000e8    2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
2   .dynstr       0000003c    000001e8    000001e8    000001e8    2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
3   .text         00000005    00000224    0000224     0000224     2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
4   .data         00000000    0000122c    0000122c    0000022c    2**2
                  CONTENTS, ALLOC, LOAD, DATA
5   .dynamic      00000058    0000122c    0000122c    0000022c    2**2
                  CONTENTS, ALLOC, LOAD, DATA
6   .got          0000000c    00001284    00001284    00000284    2**2
                  CONTENTS, ALLOC, LOAD, DATA
2   .bss          00000000    00001290    00001290    00000290    2**2
                  ALLOC

SYMBOL TABLE:
00000094    l    d    .hash       00000000
000000e8    l    d    .dynsym     00000000
000001e8    l    d    .dynstr     00000000
00000224    l    d    .text       00000000
0000122c    l    d    .data       00000000
0000122c    l    d    .dynamic    00000000
00001284    l    d    .got        00000000
00001290    l    d    .bss        00000000
0000122c    g     O   *ABS*       00000000    _DYNAMIC
00000224    g     F   .text       00000001    f
00000228    g     F   .text       00000001    g
00001284    g     O   *ABS*       00000000    _GLOBAL_OFFSET_TABLE_

Disassembly of section .text:
00000224 <f>:
   224:     c3                  ret
```

```
   225:      90                  nop
   226:      90                  nop
   227:      90                  nop

00000228 <g>:
   228:      c3                  ret
```

至少从表面上可以看出，f 和 g 不再分别属于不同的模块，它们显然是位于同一个组织单元内。上一节曾经讲过，在静态库文件中，系统容许不同的模块里面存在相同名字的符号，但在动态库文件中，各个符号的名字必须是唯一的。大家可以自行测试一下，把0503.s 里面的“g”替换成“f”，然后编译0503.s，再重新制作动态库文件 libfg.so，立刻就会得到出错信息：

```
0503.o(.text+0x0): In function ‘f’:
: multiple definition of ‘f’
0502.o(.text+0x0): first defined here
```

ld 发现0502.o 和0503.o 都有一个符号名为“f”，从而无法正确生成动态库文件。这从一个侧面反映了“f”和“g”都是从属于整个动态库文件的。

现在我们编译0501.s，然后把它和 libfg.so 链接：

```
$as  -o  0501.o  0501.s
$ld  0501.o  -L. -lfg  --dynamic-linker /lib/ld-linux.so.2
```

在正确生成可执行文件 a.out 之后，我们首先用 *nm* 查看一下符号表：

```
$nm  a.out
...
080481f8  T  _start
         U  f
```

由输出结果我们可以很清楚地知道，符号“g”所代表的代码确实没有被复制到可执行文件，因为符号表中只有“f”而没有“g”。而且，符号“f”的属性是“U”(undefined)，虽然我们以前也碰到过“U”属性，但那是在目标代码文件里面，现在的情形是可执行文件内出现“U”属性的符号，这证明“f”所代表的代码要到程序运行的时候才会被真正复制到进程的地址空间。同时，*objdump* 命令的输出显示了：

```
$objdump  -h  a.out

a.out:     file format elf32-i386
Sections:
Idx   Name          Size      VMA        LMA        File off    Algn
    6   .text       0000000c   080481f8  080481f8   000001f8    2**2
                    CONTENTS, ALLOC, LOAD, READONLY, CODE
```

显然，a.out 的“.text”区大小为12字节，而目标代码文件0501.o 的“.text”

区大小也是12字节，这说明可执行文件中并不存在“f”所代表的代码，否则 a.out 的“.text”区就应该大于12字节。

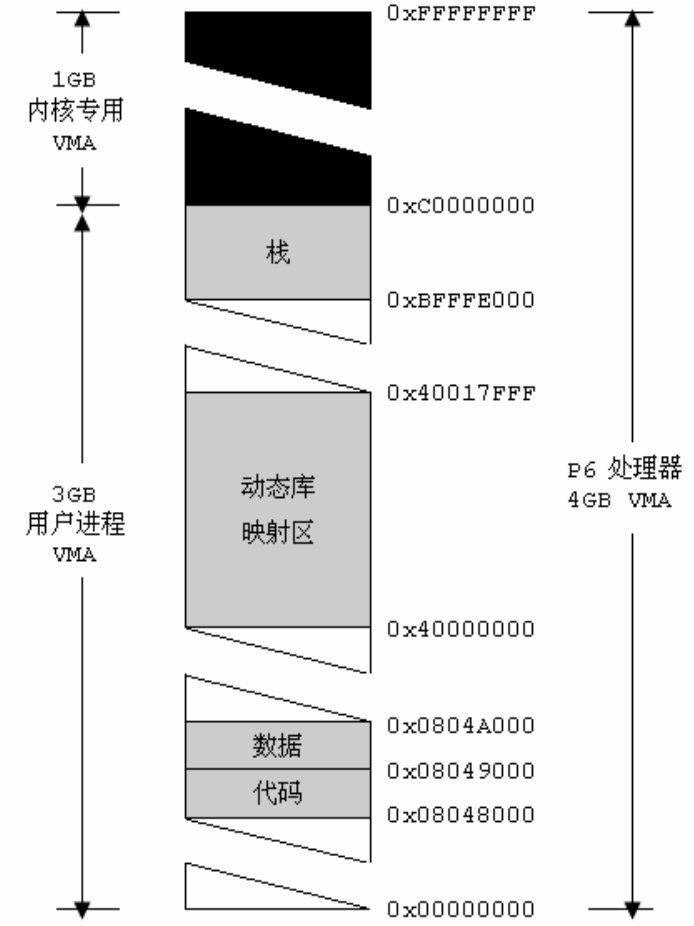


图 05-01

上面只是观察可执行文件本身得到的信息，要真正体会动态库代码在运行时被载入程序映射到进程的地址空间，最简单的方法是使用调试程序（debugger），下面我们利用 ***gdb*** 演示一下整个过程。

首先，我们要在生成的目标代码文件中加入调试信息，因此要使用“--gdwarf2”编译选项：

```
$as  --gdwarf2  -o  0501.o  0501.s
$as  --gdwarf2  -o  0502.o  0502.s
$as  --gdwarf2  -o  0503.o  0503.s
$ld  -shared  -o  libfg.so  0502.o  0503.o
$ld  0501.o  -L. -lfg  --dynamic-linker  /lib/ld-linux.so.2
```

然后，我们先设置环境变量 LD_LIBRARY-PATH，好让载入程序正确找到 libfg.so，接着就可以用 ***gdb*** 调试 a.out：

```
$export  LD_LIBRARY_PATH=/home/yxy/05
$gdb  a.out
   ...
(gdb) l
1          .text
2       .globl _start
3       _start:
4          call f
5          mov $1, %eax
6          int $0x80
7
(gdb) b  4
Breakpoint 1 at 0x80481f4: file 0501.s, line 4.
(gdb) r
Starting program: /home/yxy/05/a.out
Breakpoint 1, _start () at 0501.s:4
4      call f
Current language:  auto; currently asm
(gdb) i  proc
process 1273
cmdline = '/home/yxy/05/a.out'
cwd = '/home/yxy/05'
exe = '/home/yxy/05/a.out'
(gdb) i  share
From        To          Syms  Read   Shared Object Library
0x400162ac  0x400162b1  Yes          ./libfg.so
(gdb) disas  0x400162ac
Dump of assembler code for function f:
0x400162ac <f+0>:      ret
0x400162ad <f+1>:      nop
0x400162ae <f+2>:      nop
0x400162af <f+3>:      nop
End of assembler dump.
(gdb) s
f () at 0502.s:5
5      ret
(gdb) i  reg
   ...
eip          0x400162ac      0x400162ac
   ...
(gdb) c
Continuing.
Program exited with code 0200.
(gdb) q
$
```

当启动 ***gdb*** 开始调试 a.out 之后，我们先用“l(list)”命令显示出 a.out 对应的源代码文件0501.s，然后用“b(break)”命令在第4行设置一个断点。这样，当运行程序时，执行点一到达第4行就会自动停下来。这一切准备妥当后，我们用“r(run)”

命令通知 ***gdb*** 运行 a.out。接着，gdb 果然在第4行的前面停下来。这个时候，我们用“i(info)”命令查看进程引用了哪些动态库，***gdb*** 返回的结果自然就是当前目录下的“libfg.so”，并且动态库的代码地址被安排在0x400162ac 到0x400162b1。可见，代码已经出现在进程的地址空间内，进程完全可以访问这些动态库代码。

要是大家还不敢确信的话，我们再用“disas(disassemble)”命令来反汇编从地址0x400162ac 开始的内容，结果表明地址0x400162ac 处是一条 ret 指令，这正是0502.s 中代码“f”对应的唯一一条指令。

用“s(step)”命令进入“f”，这时，ret 指令正是处理器马上要执行的指令，我们用“i(info)”命令查看主要的寄存器的内容，发现当前 eip 的值就是0x400162ac，这确凿地表明0x400162ac 真的是“f”对应的代码的起始地址。

最后，我们用“c(continue)”命令让程序一直运行下去直到结束，再用“q(quit)”命令退出 ***gdb***。整个进程的虚拟地址空间分配状况如图05-01。

其实，GNU/Linux 系统本身已经为我们提供了进程的部分关键信息，上面我们在使用 ***gdb*** 调式 a.out 的时候已经通过“i proc”知道进程的 PID 是1273，如果在这个时候我们在另外一个终端窗口键入：

```
$cat /proc/1273/maps
08048000-08049000 r-xp 00000000 03:07 394302    /home/yxy/05/a.out
08049000-0804a000 rw-p 00000000 03:07 394302    /home/yxy/05/a.out
40000000-40014000 r-xp 00000000 03:07 586990    /lib/ld-2.3.2.so
40014000-40015000 rw-p 00013000 03:07 586990    /lib/ld-2.3.2.so
40015000-40016000 rw-p 40015000 00:00 0
40016000-40017000 r-xp 00000000 03:07 394298    /home/yxy/05/libfg.so
40017000-40018000 rw-p 00000000 03:07 394298    /home/yxy/05/libfg.so
bfffe000-c0000000 rwxp bfffe000 00:00 0
ffffe000-fffff000 ---p 00000000 00:00 0
```

命令的结果清楚地描述了系统给进程（PID 为1273）分配的虚拟地址空间，其中，属性里有“x”的页面存放代码和只读数据、有“w”的页面存放普通数据、既有“x”又有“w”的页面是进程的栈。最后一行对应的页面并不属于进程的地址空间，我们不用理会。

明白了动态链接和静态链接的主要区别之后，我们讨论第一个问题——装载程序如何确定可执行文件所需要的动态库的位置。既然要到程序真正被载入内存开始运行的时候装载程序才把动态库的相关内容加载到进程的地址空间，那么前提当然就是动态库文件能够被正确定位。一般地，装载程序至少有四种寻找程序所需要的动态库的常规途径，下面分别详细介绍。

- **在链接命令中直接指定库文件的绝对路径**

这是最硬性的定位方式，绝对路径被链接程序记录在可执行文件中，到了程序将要运行的时候，只会出现两种情况：装载程序要么根据绝对路径正确找到所需要的动态库，要么找不到绝对路径所指定的库文件而报错退出从而程序无法运行。注意，对于使用绝对路径定位动态库的可执行文件，装载程序不会试图再使用其它方式寻找库文件，只要找不到

绝对路径所对应的库文件，装载程序就会立即停止载入。还是使用上面的例子：

```
$ld 1.o /home/yxy/05/libfg.so --dynamic-linker /lib/ld-linux.so.2
```

显然，“*/home/yxy/05/libfg.so*”是一个绝对路径，我们用 ***objdump*** 观察一下：

```
$objdump -x a.out
  ...
Dynamic Section:
  NEEDED      /home/yxy/05/libfg.so
  ...
```

在“Dynamic Section”中，我们可以看到 a.out 需要链接一个动态库，它的绝对路径就是“*/home/yxy/05/libfg.so*”，装载程序正是根据这个信息去定位 libfg.so。如果装载程序顺利找到该库文件，一切正常；如果找不到，程序就无法运行：

```
$cd /home/yxy/05/
$./a.out
$
$mv libfg.so libfg.so.bak
$./a.out
./a.out: error while loading shared libraries: /home/yxy/05/libfg.so:
cannot open shared object file: No such file or directory
$
```

上面先运行 a.out，由于装载程序顺利找到 libfg.so，所以程序正常运行。然后我们把 libfg.so 改名，于是在第二次运行 a.out 时装载程序就无法找到 libfg.so，结果只能是报错退出。

使用绝对路径指定动态库的优点是安全性高，因为别人无法替换程序所需要的库文件。由于绝对路径信息本身内置于程序之中，而且装载程序拒绝装载任何其它路径的库文件，这就保证了程序一旦运行则其使用的动态库必然位于原先指定的绝对路径上，除非其它人拥有该绝对路径的写权限从而有机会直接替换掉库文件，否则任何“鱼目混珠”的事情都不会发生。

同时，使用绝对路径的做法也意味了程序在安装之前其使用的库文件就已经被限制在某个固定的目录之中，这显然不太适合除系统管理员和系统软件发行商之外的应用程序发布者。系统管理员（譬如 root 用户），或者系统软件发行商（例如某个 Linux 版本的发布者），他们出于安全考虑的确可以把某些程序的动态库安排存放在某些特定的目录然后对敏感程序使用指定绝对路径的方式进行链接。由于系统管理员、系统软件发布者都拥有预先规划整个系统的能力和权限，所以他们可以这样做。但对于普通的应用程序发布者，他显然不能使用绝对路径去定位自己的库，因为他不太可能强制所有安装他的软件的用户都按照他预先设想的绝对路径去建立目录。

- **在链接命令中指定优先搜索路径 RPATH**
- **设置环境变量 LD_LIBRARY_PATH 以提供搜索路径**
- **使用系统动态库路径信息缓存**

上面三种方式虽然不完全相同，但它们有一个共同点：非强制性，就是说，即使装载程序在指定的目录找不到需要的库文件也不会立即报错，而是继续按照对应的优先顺序继

续搜索。譬如，某个程序内置了 RPATH，但装载程序按照 RPATH 找不到动态库，那么装载程序会根据当前环境变量 LD_LIBRARY_PATH 的值继续搜索，如果系统没有设置这两个环境变量又或者根据环境变量进行寻找仍然失败，那么装载程序就会依照系统默认的动态库搜索路径作最后的尝试。当所有的这些努力都失败，装载程序才宣布放弃，报错退出。

在链接命令中指定 RPATH 很简单，仍然以上面的代码为例：

```
$ld  1.o  -L /home/yxy/05  -lfg  --dynamic-linker  /lib/ld-linux.so.2   \
-rpath  /home/yxy/05
```

由于这次我们不再使用绝对路径进行链接，所以必须用“-L 绝对路径/相对路径”以及“-lxxx”选项，这样链接程序才能找到对应的动态库文件。当然，直接写出库文件的名字也可以：

```
$ld  1.o  -L /home/yxy/05  libfg.so  -rpth  /home/yxy/05    \
--dynamic-linker  /lib/ld-linux.so.2
```

不妨用 *objdump* 查看、对比一下：

```
$objdump  -x  a.out
  ...
Dynamic Section:
  NEEDED      libfg.so
  RPATH       /home/yxy/05/
  ...
```

结果很清楚，在“NEEDED”一栏中只有库文件名字而不是整个绝对路径，还多了一个“RPATH”项，它指出了优先搜索路径“/home/yxy/05”。

另外，我们还可以在同一文件中指定多个 RPATH：

```
$ld  1.o  -L /home/yxy/05  -lfg   \
--dynamic-linker  /lib/ld-linux.so.2  -rpath  /home/yxy/05:/tmp:/xyz
$objdump  -x  a.out
  ...
Dynamic Section:
  NEEDED      libfg.so
  RPATH       /home/yxy/05:/tmp:/xyz
  ...
```

上面我们一下子就给出了三个优先搜索目录，它们分别是“/home/yxy/05”、“/tmp”和“/xyz”，其中，“/xyz”是虚构出来的、系统中根本就不存在这个目录，但这完全不影响我们把它作为优先搜索路径之一。

使用 RPATH 的特点是既指出了优先搜索路径，又不受限于当前的目录状况。假设有个软件发布者，他开发了一个程序，该程序要使用某个动态库文件 libABC.so。出于安全性的考虑，他要求使用该程序的人必须把 libABC.so 安装在受保护的系统目录“/lib”中，并且装载程序应该优先载入“*/lib/libABC.so*”，这个本来不难办到，使用绝对路径方式就可以满足要求。但这位开发者是在一台公共主机上进行开发工作的，他自己不是该主机的系统管理员，因此无法把 libABC.so 复制到主机的“/lib”目录里进行测试。

这个时候，RPATH 方式就可以解决问题。他只要使用 RPATH 指定“/lib”为优先搜索路径即可。上面我们说过，和绝对路径方式一样，RPATH 方式同样是把某些路径信息内置在程序里，但 RPATH 方式给自己留了后路，如果按照 RPATH 找不到库文件还可以有其它途径继续找。还是以上面的开发者为例，在他的开发平台上，由于没有相应的权限，他无法真的把 libABC.so 复制到“/lib”目录，但这无所谓，反正装载程序依照 RPATH 找不到 libABC.so 还有其它办法，所以程序仍然能够正常运行，开发和测试不成问题。而到了使用该软件的机器上，管理员先把 libABC.so 安装在“/lib”目录，然后在程序运行的时候，装载程序按照 RPATH 的设定自然会优先搜索“*/lib/libABC.so*”，这正是开发者的本意所在。

如果用户既不使用绝对路径也不使用 RPATH，那么可以通过设置环境变量来提供动态库的搜索路径。譬如：

```
$ld 1.o -L /home/yxy/05 -lfg --dynamic-linker /lib/ld-linux.so.2
$objdump -x a.out
  ...
Dynamic Section:
  NEEDED     libfg.so
  ...
$./a.out
./a.out: error while loading shared libraries: libfg.so: cannot open shared
object file: No such file or directory
$export LD_LIBRARY_PATH=/home/yxy/05
$./a.out
$
```

注意一下链接命令，没有绝对路径形式的库文件名，没有“-rpath”选项，再留意 ***objdump*** 命令的输出，“NEEDED”一项中只有库文件名，可见，这次生成的可执行文件 a.out 并没有内置任何的路径信息。在第一次运行时，装载程序由于找不到 libfg.so 而报错，然后我们设置了环境变量 LD_LIBRARY_PATH，于是第二次运行 a.out 时一切正常，因为这次装载程序顺利搜索到“*/home/yxy/05/libfg.so*”。

一般地，用户都拥有在自己登录 SHELL 里面设置环境变量的权限，而系统管理员也可以在相关文件预先为所有登录用户设置统一的环境变量，所以使用环境变量的方式最灵活，不过，用户可以自行改变环境变量从而欺骗装载程序从另外一个地方载入库文件，这个冒充的库文件含有同样名字的符号，于是危险代码就会被链接入进程的地址空间。众所周知，某些系统程序一旦运行，其对应进程的权限很大，因此，为了防止动态库欺骗，各种系统都有相应的措施。一般的做法是由装载程序进行检查，只要程序的拥有者和启动者不是同一个用户，装载程序就会清除动态库路径环境变量，从而在稍后的搜索中排除了程序启动者自己设置的路径。

如果程序既没有内置动态库的绝对路径、优先搜索路径，又没有设置环境变量 LD_LIBRARY_PATH，那么装载程序就会搜索系统的动态库路径信息缓存。在 GNU/Linux 中，系统程序 ldconfig 根据配置文件“*/etc/ld.so.conf*”维护着一份系统动态库路径信息缓存，这些缓存信息记录在“*/etc/ld.so.cache*”中。通常在系统启动的时

候，脚本程序会运行 ldconfig 建立一份缓存，在下一次运行 ldconfig 之前这份缓存不会被改变。当前面三种方式都找不到动态库文件时，系统会自动搜索缓存信息试图定位库文件，结果可能成功也可能失败（譬如在上次运行 ldconfig 之后库文件被删除）。如果我们要改变这份缓存，只要先修改 ld.so.conf 文件，在里面增加或减少路径信息，保存后再删除 ld.so.cache 文件，然后运行 ldconfig 程序就可以建立全新的一份路径缓存，当然这些操作都要求系统管理员权限。

下面，我们简要说明一下0502.s、0503.s 新出现的两个语句（“X”代表某个符号）：

```
type X, @function
size X, .-X
```

第一句是关于符号“X”的类型信息（X 代表指令还是代表数据），如果是“@function”则意味着“X”代表指令，如果是“@object” 则“X”代表数据。虽然编译器根据符号属于“.text”区还是“.data”（“.bss”）区可以判断符号代表的是代码还是数据，因此即使没有该语句也不会出现什么大问题，但对于代表指令的符号，要是它位于某个动态库里面，那么是否使用“.type”语句将会影响指令最终被安排在“.text”区还是“.bss”区。假设没有使用“.type”语句明确指定动态链接的代码必须被安排在“.text”区的地址空间内，则链接程序可能会把该动态库代码安排在 “.bss”区的地址空间内。虽然不会影响程序的运行，但不要忘记，“.bss”区是“可写”的，这就是说，如果动态库代码被安排在“.bss”区，那程序的其它代码完全有可能在程序运行期间改变这些代码。想象一下，万一程序的设计有问题，例如对“.bss”区数据进行写操作时存在某种逻辑错误而导致动态库代码被覆盖，则程序可能会出现莫名其妙的崩溃。

第二句是关于符号所代表的代码或数据的长度信息，如果要生成 COFF 格式的文件那么它将是必不可少的。由于 GNU/Linux 的目标代码文件、库以及可执行文件都是基于 ELF 格式，所以该语句在我们的平台上其实是可以省略的。但加上它之后我们可以在 ***objdump*** 命令的输出中了解某段代码或某个数据的长度，而且，从后面的小节中大家可以观察到，在 GNU C/C++编译器所生成的汇编代码里面所有的符号都有自己的“.type”和“.size”语句，因此我们为了统一起见今后一律加上该语句。

动态库文件不是若干模块的简单组合，它是一个不可分割的整体，因此，我们无法像更新静态库某个模块那样去更新动态库。其实，从前面 ***objdump*** 命令的输出我们也已经可以猜到这一点：

```
$objdump  -xd  libfg.so
    ...
00000224    g    F  .text   00000001    f
00000228    g    F  .text   00000001    g
    ...
```

“f”和“g”所代表的代码已经固定在文件偏移224、228字节的位置上，如果我们要更新“f”，无疑要保证新的代码不能超过4字节，这是不可能的；即使采取把新的代码安排在旧代码的后面的策略，库文件内部也会存在大量的空隙造成磁盘空间的浪费。因此，当我们需要更新某个动态库时，哪怕是仅仅更新其中的一小段代码也得重新生成整个库文

件。

另外，前面我们提到，绝对不能用 ***strip*** 命令施加于静态库文件，因为 ***strip*** 命令会把静态库的符号表删除，这样的话静态库文件也就无法使用了。相比之下，动态库文件除了通常的符号表之外还额外拥有一份动态符号表，***strip*** 命令只会删除普通的符号表而不会删除动态符号表，所以，即使施加了 ***strip*** 命令，我们仍然可以正常使用动态库文件。同样道理，利用 ***strip*** 命令删除使用动态库的可执行文件的符号表绝对不会影响程序的执行。如果我们用 ***nm*** 命令无法看到这些文件的符号时，只须加上“-D”参数选项即可，例如：

```
$strip  libfg.so
$nm  libfg.so
nm: libfg.so: no symbols
$nm  -D  libfg.so
    ...
00000224    f
00000228    g
    ...
```

使用动态库不仅可以节省空间（包括磁盘空间和内存空间），更新、升级工作还省时省力。我们已经知道，由于链接程序会把静态库的代码复制到可执行文件，因此理论上如果静态库有任何改动（哪怕是很小的改动）人们都要重新链接一次，这样才能以新的库代码取代旧的库代码。于是，静态库的升级工作显得非常烦琐，牵一发而动全身，整个部署过程的灵活性和适应性太差。相比之下，动态库的更新工作可谓易如反掌，下面我们就来着重讨论一下动态库的升级。

首先，我们必须明白，库的更新只有两种：可兼容更新与不兼容更新，动态库也不例外。可兼容更新意味着动态库的接口或者动态库代码的行为没有任何变化，因此应用程序可以直接使用新版本的动态库，譬如我们针对库代码作出某种优化、采用新的算法、改善了性能表现等等，这些更新都属于可兼容更新。相反，如果库的接口或者库代码的行为发生改变，则该更新便属于不兼容更新，譬如我们增加或减少库函数、对库函数的调用参数作出更改、给某个库函数定义了与以往不同的行为等等。

我们要追求的目标是：如果没有进行不兼容更新，已有的程序和新的程序都应该能够平滑过渡到最新版本的库；如果进行了不兼容更新，那么已有的程序继续使用旧版本的库从而正常工作，新的程序则开始使用新版本的库。

对于静态库，无论是可兼容更新或者不兼容更新，我们都要重新链接程序才有意义。所以上述目标根本无法达到；对于动态库，只要稍微作一些部署，更新工作就相当简单轻松。

为了让不同的程序使用不同版本的动态库，我们必须以不同的名字区分这些库文件。最常见的做法是在文件的后面加上详细的版本信息，譬如：

```
libABC.so.2.1
libxyz.so.4.2.5
```

上面分别是两个动态库的文件名，第一个库文件使用2级版本信息，第二个库文件则使用3级版本信息。无论使用多少级的版本信息，我们的安排始终都是：

libABC.so.2.1 比 libABC.so.2.0 要新

libABC.so.3.0 比 libABC.so.2.9 要新

即主版本号较高者其版本较新，若主版本号相同则次版本号较高者其版本较新……如此类推，一直比较下去直到分出新旧为止。

虽然我们现在可以从文件名字知道动态库的版本新旧，但很显然，我们不可能把这么详细的版本信息内置在可执行文件里，譬如说：

```
$ld x.o libABC.so.2.1 --dynamic-linker /lib/ld-linux.so.2
```

这样的做法肯定不行。因为要是这样去链接程序，那么以后运行该程序就一定要在系统搜索路径内存在“libABC.so.2.1”这样一个库文件，否则载入程序会报错退出。假设我们要升级该动态库，新版本的库是“libABC.so.2.2”，由于只是次版本号改变了，意味着这次更新是可兼容更新，就是说，凡是以前链接“libABC.so.2.1”的程序都应该无缝过渡到“libABC.so.2.2”。但载入程序根据可执行文件的内置信息偏偏硬是要求动态库的名字必须是“libABC.so.2.1”，于是，无奈之下我们惟有把“libABC.so.2.1”改名、备份，然后把“libABC.so.2.2”改名为“libABC.so.2.1”又或者新建立一个名字为“libABC.so.2.1”的符号连接，这个符号连接其实指向“libABC.so.2.2”。

大家应该看出有什么不妥了，明明我们最新版本的库文件是“libABC.so.2.2”，却要改名为“libABC.so.2.1”（或者用“libABC.so.2.1”的符号连接指向它），混乱无可避免地产生了。不知内情的人还真的猜不到其中的原因，况且，当库文件不止一个，而是几十个的时候，恐怕我们得专门用个小本详细记录下来，某个库的真实版本是多少尽管目前表面上在系统中它的版本是什么，想想看，这样的升级要是再多进行几次，保证最后谁都不清楚哪个是哪个了。况且，原先我们把详细的版本信息加入到库文件名字就是想给所有的用户清楚地知道该库文件目前的状态，现在又改来改去、弄得面目全非，岂不违背当初的意愿吗？

所以，一个最重要的结论是：带有详细版本信息的真实库文件名字不能出现在可执行文件的内置信息中。但是，如果不使用真实文件名进行链接，那么我们就只能使用下面的方式：

```
$ld x.o -L. -lABC --dynamic-linker /lib/ld-linux.so.2
```

利用“-L”和“-l”配合指出库所在的路径（该路径信息不会内置入可执行文件）以及（不带版本信息的）库文件名字，这样就避免了直接使用库文件的真实名字进行链接。不过，问题又出现了，根据“-l”参数，链接程序只知道要寻找的库是“libABC.so”，但我们只有“libABC.so.2.1”、“libABC.so.2.2”，显然，链接程序会因为找不到“libABC.so”而报错退出。很自然地，我们只有建立一个名字为 “libABC.so”的符号连接，它指向目前最新版本的库文件“libABC.so.2.2”，这样我们才能利用最新版本的库进行开发工作。

可惜，事情还没有完，如果使用不带版本信息的“libABC.so”进行链接，那么可执行文件的内置信息就变为“libABC.so”。假设我们过一段时间后把库文件升级为新版本“libABC.so.3.0”，糟糕了。这次更新连主版本号也改了，那意味着本次升级是不兼容更新。为了使从今以后的开发工作都能够使用“libABC.so.3.0”，我们必须把符号连接改为指向“libABC.so.3.0”，但这么一改，以前使用“libABC.so”的程序全

都有问题了，因为它们其实是使用“libABC.so.2.2”的，现在“libABC.so”突然指向新的、不兼容的“libABC.so.3.0”，说不定某个时候就全乱套了。

这说明一个事实：完全不把版本信息内置到可执行文件也是不行的。

归纳一下就可以得到这些线索：我们要把详细的版本信息加到库文件的名字当中以清楚区分版本的新旧，但又不能使用带有详细版本信息的真实文件名字进行连接，同时我们又要想办法让可执行文件自己在某种程度上懂得区分库的版本，并且自动选择使用（兼容的）最新版本的库。怎么办呢？

为了完满解决这个难题，动态库引入了“soname”的机制，我们可以在制作动态库时指定该库文件的 soname：

```
$ld -shared -o libABC.so.2.1 -h libABC.so.2 a.o b.o c.o
```

上面动态库的文件名是“libABC.so.2.1”，它的 soname 是“libABC.so.2”，如果用 *objdump* 命令查看：

```
$objdump -x libABC.so.2.1
  ...
Dynamic Section:
  SONAME      libABC.so.2
  ...
```

链接程序对内置了 soname 信息的动态库进行链接时，会自动把 soname 代替动态库的真实文件名，于是将来载入程序搜索的只会是 soname 所指向的库文件。譬如，假设动态库“libABC.so.2.1”的 soname 是“libABC.so.2”，那么，即使我们这样链接程序：

```
$ld x.o libABC.so.2.1 --dynamic-linker /lib/ld-linux.so.2
$objdump -x a.out
  ...
Dynamic Section:
  NEEDED      libABC.so.2
  ...
```

在可执行文件的“NEEDED”一项中出现的仍然是“libABC.so.2”，就是说，当运行该程序时，载入程序搜索的动态库文件是“libABC.so.2”而不是“libABC.so.2.1”，尽管我们在链接命令中给出的其实是后者。

有了 soname 机制的配合，我们就可以利用符号连接解决原先颇令人头痛的动态库更新问题。假设我们现在开始使用某个动态库，它的真实名字是“libABC.so.2.1”，它的 soname 是“libABC.so.2”，我们只须在同一目录下建立两个符号连接：

```
libABC.so.2         （指向 libABC.so.2.1）
libABC.so           （指向 libABC.so.2.1）
```

接下来就可以应付所有的升级情况，只要遵循以下原则：

存在符号连接“libABC.so.x”，它指向所有“libABC.so.x.y”中“y”最大的那个文件；存在符号连接“libABC.so”，它指向所有“libABC.so.x.y”中“x.y”最大的那个文件。

让我们把所有情况列举出来。在第一次升级之前，我们使用 libABC.so.2.1进行开

发工作，链接命令的形式类似为：

```
$ld  x.o  -L. -lABC  --dynamic-linker  /lib/ld-linux.so.2
```

链接程序根据命令行的信息，知道应该搜索“libABC.so”这个文件进行链接。而此时，符号连接“libABC.so”正指向“libABC.so.2.1”，于是，链接程序实际上使用文件“libABC.so.2.1”进行链接，但由于动态库文件“libABC.so.2.1”的 soname 是“libABC.so.2”，所以可执行文件的“NEEDED”一项中记录的是“libABC.so.2”。当我们运行该程序时，载入程序搜索“libABC.so.2”，刚好有符号连接“libABC.so.2”指向“libABC.so.2.1”，因此载入程序顺利找到正确版本的动态库。

过了一段时间，我们要更新动态库，新的库文件是“libABC.so.2.2”，它的 soname 自然也是“libABC.so.2”。我们要做的只是把新文件复制到同一目录下，然后把符号连接“libABC.so.2”、“libABC.so”都改为指向“libABC.so.2.2”。这对于那些使用“libABC.so.2.1”的程序没有任何不利影响，因为载入程序根据“NEEDED”一项的信息仍然会搜索“libABC.so.2”，而此刻“libABC.so.2”已经指向“libABC.so.2.2”，于是旧程序顺利过渡至新的（兼容）版本的动态库。至于程序开发，仍然使用如下形式的链接命令：

```
$ld  y.o  -L. -lABC  --dynamic-linker  /lib/ld-linux.so.2
```

因为符号连接“libABC.so”已经指向“libABC.so.2.2”，所以链接程序会使用“libABC.so.2.2”进行链接，但由于它的 soname 是“libABC.so.2”，所以最后生成的可执行文件中“NEEDED”一项对应的信息仍然是“libABC.so.2”，可以想象，运行这个可执行文件完全没问题。

又过了一段时间，我们要进行新的升级，新版本的库文件是“libABC.so.3.0”，它的 soname 是“libABC.so.3”，由于主版本号不同了，所以这次是不兼容更新。我们把新的库文件复制到同一目录下，然后建立一个新的符号连接“libABC.so.3”，它指向“libABC.so.3.0”，再把符号连接“libABC.so”改为指向“libABC.so.3.0”。

那些使用旧版本的程序一点都没有受到影响，因为它们的“NEEDED”一项中指明需要的库文件是“libABC.so.2”，而符号连接“libABC.so.2”仍然指向“libABC.so.2.2”，可以想象，旧的程序依然能够正常运行。开发工作同样也没问题，类似下面的链接命令仍旧工作良好：

```
$ld  z.o  -L. -lABC  --dynamic-linker  /lib/ld-linux.so.2
```

由于这时符号连接“libABC.so”已经指向“libABC.so.3.0”，所以链接命令使用的就是“libABC.so.3.0”，但它的 soname 是“libABC.so.3”，因此最后的可执行文件中“NEEDED”一项记录的是“libABC.so.3”。运行该可执行文件，载入程序搜索“libABC.so.3”，恰好符号连接“libABC.so.3”指向“libABC.so.3.0”，于是载入程序顺利使用新版本的库……

无论是可兼容更新还是不兼容更新，我们始终能够让旧程序正确使用到旧版本动态库中相对最新的那个文件，又可以让开发工作顺利使用新版本动态库中最新的那个文件，目标终于达到了。

*　　*　　*　　*　　*　　*

[0501]

06 ISO C99标准

制定 ISO C 标准的主要目的在于尽可能地提高 C 程序在多种系统平台上的可移植性，根据标准，C 程序员和 C 实现者可以在最大程度上进行沟通：前者可以编写符合标准的程序，在各种环境中被一致接受，而后者则可以构造符合标准的平台，接受（包括编译、执行等）所有符合规范的 C 程序。

ISO C99对 C 语言的以下内容作出规定：

- 格式（representation）
- 语法（syntax）和限制（constraint）
- 语义规则（semantic rule）
- 输入数据的格式
- 输出数据的格式
- 实现（implementaion）上的各种限制（restriction and limit）

如果一个 C 程序仅仅使用：

- ISO C99里那些有明确定义的特性
- ISO C99标准库

并且满足：

- 不产生任何依赖于

  unspecified behavior（未指定行为）

  undefined behavior（未定义行为）

  implementation-defined behavior（由实现定义的行为）

的输出

- 不超出任何实现上的最小限制

则该程序就是一个“（和 ISO C99）严格一致的程序”（strictly conforming program）。

※ ISO C 标准提供两种或以上的选择、实现可以自由决定使用其中的一种，这种选择所导致的行为就是 unspecified behavior。例如函数参数的求值顺序：

```
f(g(), h());
```

根据 ISO C99的规定，实现可以选择先执行 g()、再执行 h()、然后执行 f(g(), h())，也可以选择先执行 h()、再执行 g()、最后执行 f(h(), g())。

当出现不可移植的或者错误的程序结构又或者错误的数据而 ISO C 标准又没有进一步规定各种实现的具体反应时，实现可以自行决定接下来的行动是什么，这种选择所导致的行为就是 undefined behavior。实现可以选择：完全忽略错误、执行某种特定的操作、终止程序的编译或执行等等。例如整型数值的溢出：

```
int j = 0x7FFFFFFF;
int i = (++j) ;
```

显然， (++j)已经超过 int 类型的最大值，在这种情形下，某些实现可能会忽略溢出，另外一些实现则可能会终止进程并打印出错信息。

对于某些 unspecified behavior，如果标准规定实现必须明确指出如何选择，则这些行为就称

为 implementation-defined behavior。例如带符号整型负值的“按位右移运算”：

```
int i = -1;
i = i >> 5;
```

ISO C 标准规定，每个实现都必须对这种情况明确指出自己的选择，究竟 i 按位右移5位后是：

```
11111111 11111111 11111111 11111111
```

还是：

```
00000111 11111111 11111111 11111111
```

※

“（和 ISO C99）一致的实现”（conforming implementation）则有两种情况：

- conforming hosted implementation
- conforming freestanding implementation

前者必须能够接受所有“严格一致的程序”，后者必须能够接受所有满足下面条件的“严格一致的程序”：

- 不使用复数类型
- 只使用这些头文件对应的库：

<float.h> <iso646.h> <limits.h> <stdarg.h> <stdbool.h> <stddef.h> <stdint.h>

任何“一致的实现”都可以有自己的扩展（例如增加库的功能等），但有一个条件：不能因为扩展而导致严格一致程序的表现行为有所改变。

能够被“一致的实现”所接受的程序称为“一致的程序”。

从上面的定义可以知道，“严格一致的程序”可以被所有（对应的）“一致的实现”接受，但“一致的程序”只被某些“一致的实现”接受而被另外一些“一致的实现”拒绝。例如：

```
static int sum(int a, int b){   return (a + b);   }
int main(){
    return  sum(4, -4);
}
```

是“严格一致的程序”，它可以被所有“一致的实现”接受。

又譬如：

```
#include <stdio.h>
int main(){
    printf("Hello!\n");
}
```

是“严格一致的程序”，它可以被所有 conforming hosted implementation 接受，但会被 conforming freestanding implementation 拒绝，因为它使用了<stdio.h>对应的库。而：

```
#include <sys/stat.h>
int main(){
    chmod("/tmp/abc.txt", S_IRUSR|S_IWUSR);
}
```

是“一致的程序”，它可以被某种提供 POSIX.1接口的“一致的实现”（例如 GNU/Linux 系统）所接受，但另外某些“一致的实现”可能拒绝它，因为 chmod 属于某种实现的扩展而不是 C 标准库函数。

每个 C 语言实现环境要做的工作是：

- 把 C 源文件转换（translate）为 C 程序 [1]
- 执行 C 程序

对应前者的系统环境称为“转换环境”，对应后者的系统环境称为“执行环境”，转换环境和执行环境不一定完全一致。

一个 C 程序可以由多个分散的源文件组成，这些源文件经过以下步骤最终转换为一个 C 程序：

1）映射源文件中的多字节字符到源字符集；用对应的单字符替换所有三联符序列（trigraph sequence）；

2）把所有“\NL”字符组合（“\”是反斜杠，“NL”代表新行符）删除掉；

3）源文件被分解为预处理记号（prepocessing token）以及空白字符序列（sequence of white-space），然后把所有的注释用一个空格字符替换掉，所有的 NL 字符被保留下来，至于是否把除 NL 外的所有其它空白字符序列用一个空格字符替换掉则是 implementation-defined 的；

4）执行预处理指令（例如宏的展开等），通过#include 指令包含进来的源文件会递归地完成步骤1）、2）、3）、4）的处理，所有预处理指令在执行完毕后被删除；

5）将所有字符常量以及字符串常量中的源字符集字符、转换字符序列替换为相应的执行字符集字符；

6）合并相邻的字符串常量；

7）到目前为止，源文件由预处理记号以及分隔它们的空白字符组成，在本步骤中，所有的预处理记号被转换为记号，由这些记号组成的源文件经过语法、语义分析后被转换为一个 translation unit；

8）所有对外部实体（object）及函数（function）的引用被正确链接，所有必需的 translation unit 被合并并最终转换成为一个 C 程序。

以上处理顺序在每一个符合标准的 C 实现环境都是一致的，我们尤其需要注意其中的某些细节，例如删除“\NL”字符组合发生在删除注释以及执行宏替换的前面。

看看下面的代码片段：

```
#include<stdio.h>
extern int errCode;
extern const char * const errMsg[];
#define prtErrMsg()   \
                      // If any error occurs we write these.    \
                      fprintf(stderr, “Error: %d\t”, errCode),   \
                      fprintf(stderr, “%s\n”, errMsg[errCode])
   ...
   prtErrMsg();
   ...
```

以上代码定义了一个打印出错信息的宏 prtErrMsg()，它首先在 stderr 上打印错误代码号 errCode，然后再打印对应的错误消息 errMsg[errCode]。该宏的定义比较长，因此被分开数行进行书写。

但是，由于注释的存在，宏 prtErrMsg()实际上并没有按照原来的意图进行定义。因为编译器在完成步骤2）之后，这段代码就变成：

```
#include<stdio.h>
extern int errCode;
extern const char * const errMsg[];
#define prtErrMsg()   // If...fprintf(…)...fprintf(…)
    ...
    prtErrMsg();
    ...
```

很明显，接下来的步骤3）会把注释用一个空格字符替换掉，从而使到 prtErrMsg 根本没有具体定义，那个调用宏的语句经过预处理之后仅仅是一个空语句“;”。

前面曾经提到，一个“严格一致的程序”不能超出任何实现上的最小限制，这些限制包括转换环境的限制和执行环境的限制，其中转换环境的限制如下：

- 不超过127层嵌套的 block

例如下面的函数 f 里面有2层嵌套的 block：

```
void f()
{
    if (...)
    {                           /* 第1层开始 */
        while (...)
        {                       /* 第2层开始 */
            ...
        }                       /* 第2层结束 */
    }                           /* 第1层结束 */
}
```

- 不超过63层嵌套的条件包含

例如下面的预处理指令使用2层嵌套的条件包含：

```
#if (...)
    #if (...)
        #include <stdio.h>
    #endif
#endif
```

- 在一个算术类型、或者结构体、联合、不完整等类型的声明中有不超过12个指针、数组、函数的 declarator（包括它们的任意组合）参与说明

例如：

```
int (*arr[8])(void);            /* 3个declarator */
```

- 在一个“full declarator”里面有不超过63层嵌套的加括号的 declarator。

例如：

```
void **p;            /* 2层嵌套， 相当于void (*(*p)); */
```

- 在一个 full expression 里面有不超过63层嵌套的加括号的 expression

例如：

```
a = b * (c + d * (e - f));          /* 2层嵌套 */
```

- 内部标识符（internal identifier）或者宏（macro）的名字不超过63个字符（每个 UCS 字符或扩展源字符集字符均按一个字符计算）
- 外部标识符（external identifier）的名字不超过31个字符（编码值不大于0x0000FFFF 的 UCS 字符按6个字符计算，编码值大于或等于0x00010000的 UCS 字符按10个字符计算，扩展源字符集字符按和它相同的 UCS 字符所对应的字符数计算）
- 在一个 translation unit 里面有不超过4095个外部标识符
- 在一个 block 里面有不超过511个具有 block scope 的内部标识符
- 在一个预处理转换单元（prepocessing translation unit）里面同时定义不超过4095个宏
- 用#include 指令包含文件的嵌套层数不超过15

例如下面的 example.c 里面就有2层的嵌套包含，example.c 包含 aaa.h，aaa.h 包含 bbb.h（第1层嵌套），bbb.h 又包含 ccc.h（第2层嵌套）：

aaa.h

```
#ifndef AAA_H
#    define AAA_H
#    include “bbb.h”
#endif
```

bbb.h

```
#ifndef BBB_H
#    define BBB_H
#    include “ccc.h”
#endif
```

ccc.h

```
#ifndef CCC_H
#    define CCC_H
#    define BUFSIZE 64;
#endif
```

example.c

```
#include <aaa.h>
char arr[BUFSIZE] = {0};
int main()
{}
```

- 函数的参数不超过127个
- 调用函数时使用的参数不超过127个
- 宏的参数不超过127个
- 调用宏时使用的参数不超过127个
- 逻辑源代码行（logical source line）的字符数不超过4095个
- 字符串常量或宽字符串常量的字符数不超过4095个
- 对象（object）的大小不超过65535字节（本限制仅适用于 hosted

implemention）

- switch 语句的 case 标号不超过1023个
- 结构体或联合体的成员不超过1023个
- 枚举所包含的枚举常数不超过1023个
- 结构体或联合体的嵌套定义层数不超过63

例如下面的结构体定义用了2层嵌套：

```
struct A
{
    char *p;
    struct B
    {                           /* 第1层嵌套 */
        int i;
        double d;
        struct C
        {                       /* 第2层嵌套 */
            char arr[16];
            float f;
        }sc;
    }sb;
};
```

执行环境的限制主要表现在数值处理方面，这一部分内容将在后面的章节中详细讨论。对应于两种“一致的实现”，C 的执行环境同样有两种情况：

- hosted environment
- freestanding environment

在 hosted 环境中，当一个 C 程序启动（startup）时系统会调用 main 函数，并且所有静态存储对象的初始化会在 main 函数被调用之前完成， main 函数没有函数原型，所有的 hosted 环境至少应该接受下面的两种 main 定义形式：

```
int main(void){ ... }
int main(int argc, char *argv[]){ ... }
```

argc 和 argv 是执行环境传递给 main 函数的系统参数。例如用户用以下命令执行一个 C 程序：

```
$./a.out  abc  xyz  12345
```

则该进程的 main 函数接收到的 argc 和 argv 如图06-01。

C99规定：

- argc 不能为负数。
- argv[argc]是一个 null 指针。
- argv[0]、argv[1]…argv[argc-1]都是 char 指针，其中 argv[0]指向的是程序名字，argv[1]…argv[argc-1]指向的是命令行参数。

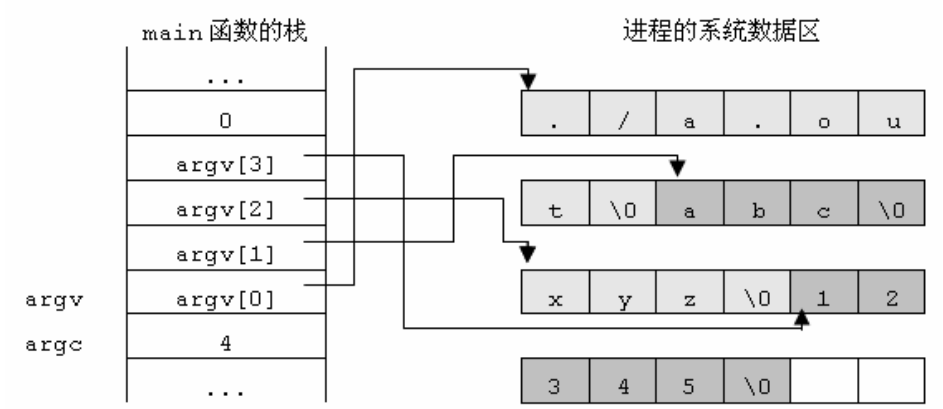


图 06-01

通常情况下，C 程序可以通过3种方式结束执行：

▪ 用户以“return retValue;”语句从 main 函数返回（retValue 是 int 值）。例如：

```
int main(){
    ...
    return intValue_1;                /* 返回*/
    ...
    return intValue_2;                /* 返回*/
}
```

▪ 用户直接用一个 int 值作为参数调用 abort、exit、_Exit 等函数。例如：

```
int main(){
    ...
    exit(intValue_1);            /* 调用 exit 退出 */
    ...
    abort(intValue_2);          /* 调用 abort 退出 */
    ...
}
```

▪ 执行点到达 main 函数的“}”，系统自动用“0”作为 main 函数的返回值。例如：

```
int main()                          int main(){
{                                       ...
    ...                 相当于          return 0;
}                                   }
```

我们可以粗略地概括一个 C 程序的内部逻辑结构如图06-02：[2]

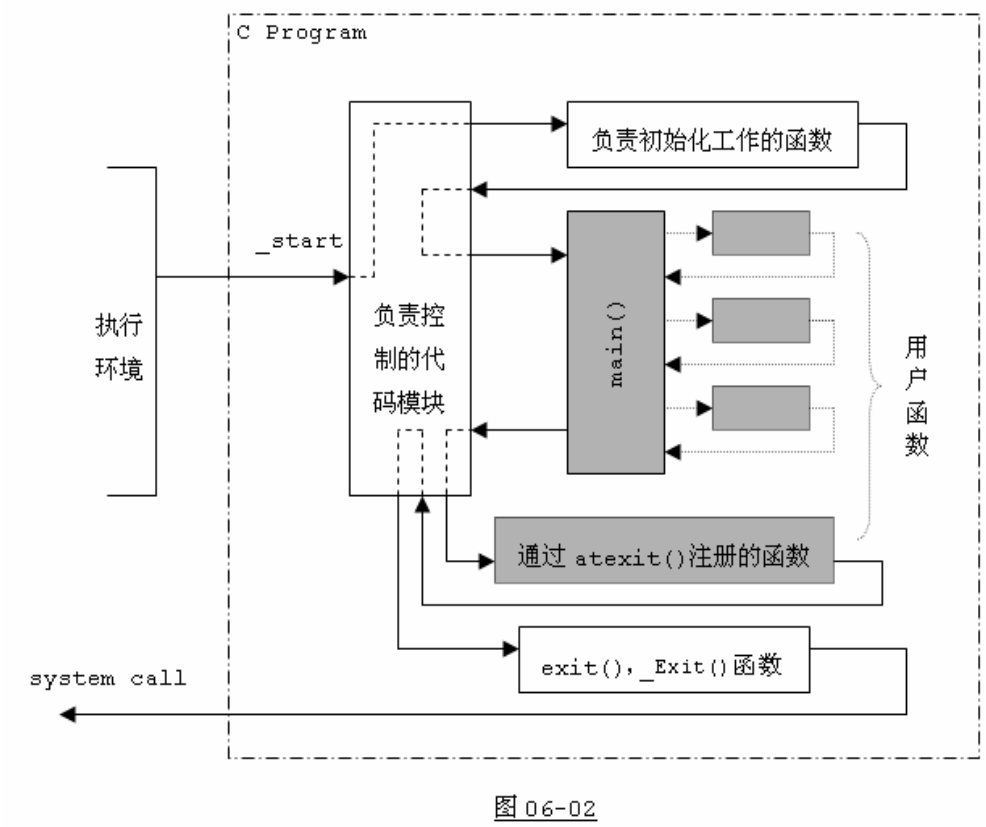


图 06-02

在图06-02中，灰色部分的函数由用户自己编写，其余的则由编译器厂商或独立的标准库供应商提供。而负责初始化工作的函数其中一个主要的任务就是打开三个标准输入、输出文件，这三个文件对应的文件指针是：

stdin （标准输入）

stdout（标准输出）

stderr（标准错误输出）

这就是我们并没有在自己的 C 程序里创建三个标准输入、输出文件却可以使用它们的原因，相关函数早在 main 函数执行之前就已经把一切准备妥当了。

用户可以正常地从 main 函数返回退出程序，也可以在 main 函数或其它用户自己定义的函数里直接调用 abort()、exit()、_Exit()等函数退出程序，负责清理工作的函数或者 abort()、exit()、_Exit()等函数在返回执行环境前都会自动关闭所有打开的文件（包括三个标准输入、输出文件）。

*　　*　　*　　*　　*　　*

[0601]

注意，这里的术语“C 程序（C program)”应理解为我们通常所说的可执行文件，因为标准已经

用术语“C 源文件（C source file)”来表示我们通常所说的“C 程序文件”这个概念。

[0602]

图06-02的描述仅适用于 C 程序结束执行的其中两种情形，至于用户直接调用 abort()、exit()和_Exit()函数退出的图示这里限于篇幅没有给出。

07　C 源文件的编译和链接

从本节起我们开始使用 GNU C 编译器 ***gcc*** 编译 C 源文件，这里的“C 编译器”其实是一个接口程序，它会根据需要分别调用 C 语言编译程序、汇编语言编译程序、链接程序来完成一个 C 源文件的编译过程。先看以下简单的例子：

```
C01       /* Code 07-01, file name: 0701.c */
C02       #include <stdio.h>
C03       int main()
C04       {
C05           printf("Hello, Linux!\n");
C06           return 0;
C07       }
```

我们用“-v”选项编译07-01（下面的输出信息有省略）：

```
$gcc -v 0701.c
cc1 0701.c -o /tmp/ccWri9A6.s
#include <...> search starts here:
/opt/gcc/lib/gcc-lib/i686-pc-linux-gnu/4.0/include
as -o /tmp/ccqUiPy4.o /tmp/ccWri9A6.s
collect2 ld-linux.so.2 crt1.o crti.o crtbegin.o /tmp/ccqUiPy4.o
crtend.o crtn.o -lc -lgcc -lgcc_eh
```

可以看到，***gcc*** 首先会调用 C 语言编译程序 ***cc1***编译07-01，而在编译过程中，***cc1*** 首先对0701.c 进行预处理[1]，由于07-01使用了#include 指令包含<stdio.h>，所以***cc1***需要在设定的目录中展开搜索，当预处理完成后，***cc1***就会进行编译工作，并在临时目录“/tmp”下生成汇编语言程序文件 ccWri9A6.s。然后 ***gcc*** 会调用 ***as*** 把 ccWri9A6.s 编译成目标代码文件 ccqUiPy4.o，最后 ***gcc*** 调用 ***collect2***进行链接（***collect2***再调用 ***ld*** 来完成主要的链接工作），生成最终的可执行文件 a.out，如图07-01。

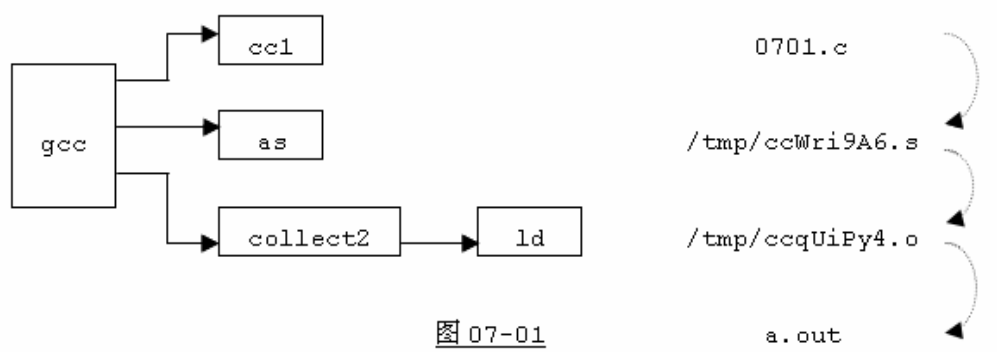


图 07-01

我们还可以指示 ***gcc*** 把编译工作停止在某些阶段。

例如：

```
$gcc  -E  0701.c > 0701.i
$gcc  -S  0701.c
$gcc  -c  0701.c
```

“-E”选项的意思是一旦 ***gcc*** 完成 C 源文件的预处理就停止编译，本来默认情况下 ***gcc*** 会把预处理后的内容输出到标准输出（一般是终端屏幕）上，这里用重定向符“>”把输出重定向到文件0701.i。

“-S”选项则通知 ***gcc*** 生成对应的汇编语言程序后就停下来，默认情况下生成的汇编文件的名字是0701.s。

“-c”选项指示 ***gcc*** 生成目标代码文件后就停止动作，默认情况下生成的目标代码文件的名字是0701.o。

不仅如此，***gcc*** 还可以接受各个阶段生成的文件作为输入，例如：

```
$gcc  0701.i
$gcc  0701.s
$gcc  0701.o
```

上面的命令最终都会生成可执行文件 a.out，与 C 源文件作为输入相比，区别只不过是省略了相应的某些编译步骤而已。

两者结合的情况同样被 ***gcc*** 接受，例如：

```
$gcc  -S  0701.i
$gcc  -c  0701.i
$gcc  -c  0701.s
```

以上命令不以 C 源文件作为输入，而且把编译过程停止在对应的某个阶段。

甚至我们亲自调用汇编程序 ***as*** 也是可以的：

```
$gcc  -S  0701.c
$as  -o  0701.o  0701.s
$gcc  0701.o
```

不过，由于 ***gcc*** 能够自动根据文件类型[2]调用相应的处理程序，所以今后我们一律使用 gcc 作为统一的命令接口。不论是预处理、生成汇编文件、生成目标代码文件还是进行链接，我们都使用 ***gcc***。

值得一提的是链接过程，不像第一章那些极其简单的汇编程序，C 源文件经过编译生成目标代码文件之后还要与多个静态库、动态库进行链接才能正确产生可执行文件，这个步骤所使用的参数比较多，所以我们一般很难直接使用 ld 命令去链接 C 程序对应的目标代码，必须通过 ***gcc*** 这个命令接口去完成相应的工作。即使我们需要对链接过程进行某些细微的控制（譬如加入某些链接选项等），我们也不会直接使用 ld 命令，而是通过 ***gcc*** 传递有关信息：

```
$gcc  -Wl,-M  0701.c
```

上面的命令向 ***ld*** 传递了一个参数选项“-M”，通知 ***ld*** 链接成功后在标准输出打印可执行文件的内存映射列表（link map）。

从前面的章节我们已经知道一个 C 程序可以有很多个独立的 C 源文件，这些文件可以分别进行编译，然后制作成库文件，又或者链接到一起，生成可执行文件。假设当前目录下有一个项目，它由三个 C 源文件（x.c，y.c，z.c）和一个静态库文件 libf.a 组

成，那我们可以这样生成可执行文件：

```
$gcc x.c y.c z.c -L. -lf
```

或者：

```
$gcc x.c y.c z.c libf.a
```

又或者：

```
$gcc *.c
$gcc x.o y.o z.o libf.a
```

甚至：

```
$gcc z.c
$gcc x.c y.c z.o libf.a
```

对于C源文件来说，如果最后要生成可执行文件，则经过编译后生成的目标代码文件通常必须和C运行期(C Run Time)目标代码文件以及C标准库(C standard library)进行链接。运行期目标代码文件的任务主要就是完成相关的系统设置以及进行C程序的初始化、清理工作等等（参看图06-02）。至于C标准库，那是因为通常我们都会在C程序里调用C标准函数（例如07-01的printf），所以必须和C标准库进行链接。

Linux的C标准库文件通常是：[3]

```
libc.a          （静态库）
libc.so         （动态库）
```

一般地，*gcc*会自动选择动态库版本进行链接，除非用特殊的链接选项加以指定。

为什么C运行期代码要以目标代码文件的形式存在而不是放入库文件呢？这里简单地解释一下其中的一个原因。

还记得每个可执行文件中代表第一条指令地址的符号“_start”吗？*ld*在链接时会自动寻找这个“_start”，如果没有找到则默认以命令行中第一个出现的目标代码文件的“.text”区地址作为起始执行点。在C程序中，“_start”必须由C运行期代码提供，如果把运行期代码放入库文件，我们就无法保证*ld*一定会把这些代码复制到可执行文件，譬如07-01就是一个很好的例子，07-01没有引用任何其它符号，在这种情形下，“_start”所代表的代码根本就不可能被*ld*复制到可执行文件，于是，到最后*ld*就会找不到“_start”这个起始执行点而产生错误的输出。

因此，C运行期代码必须以目标代码文件的形式参与链接。我们都知道，*ld*在链接目标代码文件时会把每个目标代码文件的“.data”区、“.text”区、“.bss”区等全部复制到可执行文件，这样就保证了*ld*最后肯定可以找到“_start”。

在本节的最后，我们来看看GNU/Linux系统中，用户的main()函数是怎样被调用的。在GNU C库里，crt1.o[4]包含了进程的入口“_start”，它所做的事情大体如下：

```
_start:
    ...
    pushl  %esp                         /* 用户栈空间的最高地址 */
    pushl  %edx
    pushl  $__libc_csu_fini
    pushl  $__libc_csu_init
```

```
    pushl  %ecx                          /* argv */
    pushl  %esi                          /* argc */
    pushl  $main                         /* main()函数的地址 */
    call   __libc_start_main
    hlt                                  /* 特权指令 */
```

在调整好栈空间之后，__libc_start_main()的7个参数分别入栈，然后代码的执行点就转移到__libc_start_main()函数，由它来接管下面的工作：

```
int __libc_start_main( int (*main)(int, char **, char **),
                       int argc, char ** ubp_av,
                       void (*init)(void), void (*fini)(void),
                       void (*rtld_fini)(void), void *stack_end )
{
    int result;
    ...
    result = main(argc, argv, __environ);
    ...
    exit(result);
}
```

__libc_start_main()的流程并不复杂，它检查三个函数指针参数 init、fini、rtld_fini 是否为 NULL，如果不是，就通过__cxa_atexit()函数为后两者注册以便将来在 exit()函数中进行调用，并执行 init 所指向的函数。

然后，main()函数被调用，其返回值赋给 int 变量 result。传给 main()的3个参数中，前两个是通过栈传递下来的参数，第三个是全局系统变量。

最后，__libc_start_main()以 result 为参数调用 exit()函数。由于 exit()函数不会返回到__libc_start_main()，所以正常情况下__libc_start_main()也不会返回到“_start”所代表的那段入口代码。为了在意外返回的情形下也能终止进程，“_start”代码的最后安排了一条 HLT 指令。HLT 是特权指令，只能在内核态执行，如果在用户态执行该指令会引发系统13号异常，这时 Linux 会发送 SIGSEGV 信号给进程，如果进程不捕获该信号则系统立刻会强制终止进程。

exit()函数的工作是调用以前通过 atexit()函数注册的函数以及负责清理工作的系统函数，完成这些任务后，它调用_exit()函数[5]。_exit()函数直接通过 Linux 的1号系统调用结束进程：

```
void exit(int status)
{
    while (__exit_funcs != NULL)
    {
        ...                              /* 逐个调用 atexit()注册的函数 */
    }
```

```
    ...                                 /* 调用负责清理工作的函数 */
    _exit(status);
}

void _exit(int status)
{
    INLINE_SYSCALL (exit_group, 1, status);   /* 这是一个进行系统调用的宏 */
}
```

透过上面的代码，大家已经很清楚 main()函数在什么时候被谁调用以及返回调用这之后系统又继续做了什么工作，这里需要解释一下 main()的参数。main()函数没有函数原型，而 C 标准也没有直接规定每个实现平台的 main()应该如何定义，它仅仅指出系统至少应该接受两种形式的函数定义：

```
int main(){ ... }                                   /* 形式（1） */
int main(int argc, char **argv){ ... }              /* 形式（2） */
```

由函数__libc_start_main()调用 main()的语句知道，在 GNU/Linux 实现中，main()实际上有3个参数：

```
int main(int argc, char **argv,  char ** __environ){ ... }
```

GNU/Linux 的实现是符合 C 标准的，因为当我们使用形式(1)或者形式(2)的 main()定义时，系统并不会产生任何错误，唯一的后果仅仅是我们无法直接用参数名访问栈空间对应位置的数据罢了（但可以通过其它途径例如系统提供的有关库函数进行访问）。出于可移植性的考虑，我们应该仅仅使用形式（1）或形式（2）的 main()定义，因为只有它们才是每个 C 实现都会接受的定义形式。

*　　*　　*　　*　　*　　*

[0701]

虽然 Unix 的开发环境通常都提供一个独立的 C/C++预处理程序（名字一般是“***cpp***”）给用户使用，但现在 ***gcc*** 的 ***cc1***已经包含了 ***cpp*** 的功能，所以 ***cc1***无须再额外调用 ***cpp*** 来进行预处理。

[0702]

gcc 根据文件的后缀名进行文件类型的判断，最常见的文件类型如下：

*.h	C/C++头文件
*.c	C 源文件
*.cpp *.cc *.C *.cxx *.c++	C++源文件
*.i	经过预处理的 C 源文件
*.ii	经过预处理的 C++源文件
*.s	汇编语言程序文件
*.o	目标代码文件

[0703]

在 Slackware GNU/Linux 9.1系统中，libc.a 的绝对路径是：

/usr/lib/libc.a

libc.so 的绝对路径是：

/usr/lib/libc.so

但 libc.so 只是一个 ***ld*** 脚本，目的是告诉 ***ld*** 用以下两个库文件进行链接：

/lib/libc.so.6

/usr/lib/libc_nonshared.a（某些函数没有动态库版本因此还有一个小的静态库作补充）

而 libc.so.6其实是一个符号连接（symbol link），它指向真正的动态库文件：

/lib/libc-2.3.1.so

[0704]

crt1.o 是由汇编语言代码文件 start.S 编译而成的，所以里面全部是汇编代码。

[0705]

```
void _exit(int status);
```

不是 C 语言的标准库函数，它由 POSIX.1标准进行定义，GNU/Linux 符合 POSIX 规范，因此 GNU C 库提供该函数的实现。和它等价的 C 语言标准库函数是：

```
void _Exit(int status);
```

用户既可以使用_exit()也可以使用_Exit()，事实上，在 GNU C 库中两者对应的是同一段代码。

08 C 语言的变量

在 C 程序里，变量（variable）就是数据，函数（function）就是指令。

对于变量，如果按照作用域（scope）来划分，可以分为：

外部变量（external variable）

内部变量（internal variable）

顾名思义，外部变量是在函数的外部进行定义的，它的作用范围是整个全局空间或其中的子集；而内部变量是在函数的内部进行定义的，它的作用范围局限于自身所在的函数内部或其中的子集。

如果按照存储性质（storage）来划分，则变量可以分为：

自动变量（auto variable）

静态变量（static variable）

自动变量的生存期依赖于某一段代码，当开始执行这段代码时，自动变量会被创建（分配空间、赋予初始值），当这段代码结束执行时，自动变量的空间会被收回，从而自动变量被销毁；而静态变量的生存期则从进程的开始一直持续到进程的结束，并不依赖某些代码的执行与否。

由于外部变量肯定是静态变量、自动变量肯定是内部变量，所以如果同时考虑作用域和存储性质，则只有以下三种情形：外部变量、静态内部变量、自动变量。例如：

```
int i;
static double d;
int main()
{
    float f;
    static int n;
    {
        long l;
    }
}
```

上面的代码中，i 是外部变量，作用域是全局空间；d 也是外部变量，作用域是自身所在的源文件；f 是自动变量，作用域是整个 main 函数内部；n 是静态内部变量，作用域是定义点到 main 函数末尾；l 是自动变量，作用域是从定义点到最近的右大括号之间的区域。

你是否能够猜出外部变量、自动变量和静态内部变量的存储方式呢？我们现在就通过实际的例子来寻找这个问题的答案。

先看看外部变量和静态内部变量：

```
C01         /* Code 08-02, file name: 0802.c */
C02         int i = 1;
C03         static int n = 2;
C04         int main()
```

```
C05         {
C06             static int st = 3;
C07             return 0;
C08         }
```

把08-02编译成汇编语言程序文件，然后用“less”命令查看：

```
$gcc  -S  0802.c
$less  0802.s
```

以下是0802.s 的部分内容：

```
A01         .globl i
A02             .data
A03             .align 4
A04             .type   i,@object
A05             .size   i,4
A06         i:
A07             .long   1
A08             .align 4
A09             .type   n,@object
A10             .size   n,4
A11         n:
A12             .long   2
A13             .align 4
A14             .type   st.0,@object
A15             .size   st.0,4
A16         st.0:
A17             .long   3
```

根据第一章的知识，很显然，08-02的外部变量 i、n 以及静态内部变量 st 都会被安排到“.data”区存放。于是，我们也就很容易理解为什么外部变量和静态内部变量的生存期是整个进程运行的时间。

※ 用户完全可能会定义一个名字也是 st 的外部变量，又或者用户在多个函数里面定义了各自的静态内部变量 st，这些行为都是合法的，所以编译器为了避免出现名字冲突必须修改所有静态内部变量的名字。***gcc*** 的做法是：假设静态内部变量的名字是 xxx，按照出现的顺序，第一个出现的静态内部变量 xxx 的名字改为 xxx.0，第二个出现的静态内部变量 xxx 改为 xxx.1，第三个改为 xxx.2，如此类推。 ※

如果还不敢确定，我们可以用 ***nm*** 命令验证一下：

```
$gcc  0802.c
$nm  a.out
...
08049390 D i
```

```
08049394 d n
08049398 d st.0
```

那么，是否可以下结论说，外部变量和静态内部变量一定会被编译器安排到“.data”区保存呢？应该说，编译器把外部变量和静态内部变量放到“.data”区一定是没有错的，但某些系统（包括 Linux）会根据实际情况作出小小变动以取得某些优化效果：

```
C01         /* Code 08-03, file name: 0803.c */
C02         int i = 0;
C03         static int n = 0;
C04         int main()
C05         {
C06             static int st = 0;
C07             return 0;
C08         }
```

08-03跟08-02唯一不同的是 i、n、st 的初始值都是零，用 ***nm*** 看看：

```
$gcc  0803.c
$nm  a.out
...
08049488 B i
0804948c b n
08049490 b st.0
```

这一次，外部变量和静态内部变量被编译器安排在“.bss”区，对此你是否感到奇怪呢？其实，在03小节的最后已经提到，某些系统在给“.bss”区分配完内存页面后会马上往这些页面写零，这样做虽然付出极小的时间代价，但有很大的机会可以减少程序的体积。

08-03就是一个现成的例子，那些原本应该存放在“.data”区的数据（i、n、st.0）由于它们的初始值刚好也是零，所以 ***gcc*** 采取“变通”的手法，把这些数据转移到“.bss”区，这对于进程没有任何不良影响，反正系统会保证它们的初始值都是零！最重要的是，“.bss”区的数据不占用磁盘空间，如果程序拥有大量初始值为零的外部变量（例如一个大数组），那么采用这种策略的确会带来相当显著的节省效果。

现在，我们再来看看自动变量被编译器存放到哪里：

```
C01         /* Code 08-04, file name: 0804.c */
C02         void f()
C03         {
C04             int x = 1;
C05             {
C06                 ++x;
C07                 int y = 2;
```

```
C08                 --y;
C09             }
C10         }
```

※ 注意，08-04和08-05需要加上 C99编译选项“-std=c99”，因为这两个代码例子都使用了 C99的新特性：自动变量不在函数体或复合语句的最前面进行定义。使用这个特性是由于作者想更清楚地演示自动变量即使在函数 “很后面”或者“很里面”的地方进行定义编译器仍然会在一进入函数就立刻分配好它们的空间，使用 C99编译选项的意思绝对不是说只有 C99编译器才会在一开始就为全部自动变量分配空间。 ※

```
$gcc  -std=c99 -S  0804.c
$less  0804.s
```

下面是0804.s 的主要内容：

```
A01             .text
A02         .globl f
A03             .type   f, @function
A04         f:
A05             pushl   %ebp
A06             movl    %esp, %ebp
A07             subl    $8, %esp
A08             movl    $1, -4(%ebp)
A09             leal    -4(%ebp), %eax
A10             incl    (%eax)
A11             movl    $2, -8(%ebp)
A12             leal    -8(%ebp), %eax
A13             decl    (%eax)
A14             leave
A15             ret
A16             .size f, .-f
```

不用多说，大家很清楚地看到自动变量的存放地点是进程的栈。我们很容易理解为什么自动变量的生存期总是依赖于某一段代码，因为只有执行这段代码的时候有关指令才为自动变量在栈里分配相应的空间，而当代码执行完毕时，有关指令就会调整栈指针收回自动变量占用的空间，这时自动变量也就不复存在了。

以08-04的函数 f() 为例，在它内部定义了两个自动变量（C04、C06），其中，x 的作用域是整个 f 函数的内部，因为它们是在函数体的最前面定义的；而 C06定义的自动变量 y 的作用域仅仅是 C07到 C09之间的区域。通过查看汇编代码可以知道，一进入 f 函数，系统就保存起 ebp(A05)，然后把 esp 的值赋给 ebp(A06)，再把 esp 减去8(A07)，至此，两个自动变量的空间已经分配完毕（一个 int 变量占4字节，两个共占8字节），如图08-01。

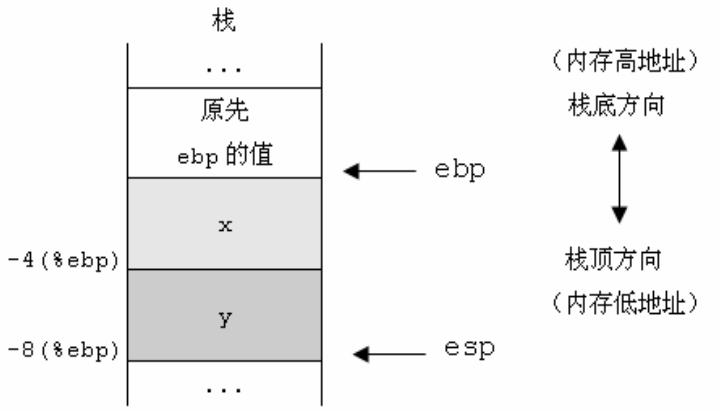


图 08-01

而且，编译器为了提高效率，通常在刚进入一个函数时就为所有自动变量分配空间，而不是真正要到访问自动变量时才为它们分配。08-04的函数 f()里面有两个自动变量：x 和 y，大家可能以为编译器应该执行完 C06、遇到 C07时才开始为 y 分配空间，对吧？——难道不是吗，自动变量要到定义它的时候才拥有自己的空间呀。是的，当编译器碰到C07时才用指令调整栈指针 esp，使之减去4从而为 y 开辟空间这种做法没有错，但效率很低。试想一下，一碰到自动变量的定义就用指令调整 esp，而当执行点退出自动变量作用域时为了销毁自动变量又必须再用指令调整 esp，这样反反复复、来回折腾，实在没有必要。因此，编译器都是一开始就为所有可能出现的自动变量分配空间，至于对这些自动变量的访问，编译器仍然严格遵从 C 语言关于作用域的规定，就是说，空间是早就分配好的，但是否可以访问则仍然按照规章办事。08-04中，自动变量 y 的空间其实在一进入函数 f 时就已经分配好了，但直到 C07之后的语句才可以访问 y。

再举一个例子：

```
C01         /* Code 08-05, file name: 0805.c */
C02         int g(void);
C03         void f()
C04         {
C05             int i;
C06             if (g() != 0)
C07             {
C08                 i = 0;
C09                 int array[20];
C10                 for ( ; i < 20; ++i)
C11                     array[i] = i * i;
C12             }
C13         }
```

08-05更具说服力，C06的 if 语句要根据函数 g()的返回值判断是否进入 C08，如果 g()的返回值不为零，则 C08~C11会被执行。由于 C09定义了自动变量 array，编译

器根本是不可能预先知道执行点是否会进入C08，但编译器给出的汇编代码仍然是在一进入函数f()的时候立即调整esp为“未必会派上用场”的array开辟空间：[1]

```
$gcc  -std=c99 -S  -O2  0805.c
$less  0805.s
```

0805.s的部分内容如下：

```
A01          f:
A02              pushl   %ebp
A03              movl   %esp, %ebp
A04              subl   $88, %esp
A05              call   g
A06              ...
```

根据A04，为了开辟空间给自动变量，esp一下子就减少了88。变量i占4字节，数组array有20个元素，每个占4字节，即array占80字节，4+80=84，还有4个字节是编译器为了边界“对齐（alignment）”而额外减掉的[2]，所以esp一共要减去88。

到现在为止，我们已经知道三种类型的变量对应的存储地点：

- 外部变量存放在“.data”或“.bss”区
- 静态内部变量存放在“.data”或“.bss”区
- 自动变量存放在栈里面

接下来再讨论一下外部变量可见范围的扩展以及不同类型变量间的屏蔽。对于外部变量，虽然它们的作用域是整个全局空间又或者其中一个子集，但有时我们仍然需要使用变量声明扩展它们的可见范围：

```
C01          /* Code 08-06, file name: 0806.c */
C02          extern int x;
C03          int main()
C04          {
C05              x = 3;
C06              return 0;
C07          }
C08          int x = 1;
```

上面的C08定义了外部变量x，很显然，x的作用域是整个全局空间，也就是说，任何函数的代码都有机会访问它。但在08-06中，访问x的语句C05出现在定义x的语句C08的前面，这时我们就需要C02去告诉编译器究竟C05访问的x是个什么东西。如果没有C02，编译器就会拒绝工作、报错退出，因为它不清楚x是什么，是char、short还是long呢？要知道，不同类型的变量对应的内存空间是不同的，编译器必须准确知道变量的类型才能决定使用什么指令去访问多少长度的内存（一个字节、两个字节还是四个字节等等）。C02的作用就是告诉编译器：某个地方已经定义了一个外部int变量x，但这里的代码可能需要访问它。于是编译器就知道如何生成正确的指令。

下面的变量声明也是基于同样的道理：

main.c

```
extern int x;
int main()
{      x = 3;   return 0;    }
```

var.c

```
int x = 1;
```

外部变量 x 在 var.c 文件中定义，而另一个文件 main.c 的代码需要访问它，所以必须使用声明告诉编译器 x 的类型。

如果外部变量和自动变量的名字相同的话，外部变量就会被“屏蔽”，代码访问的是自动变量，例如：

```
C01       /* Code 08-07, file name: 0807.c */
C02       #include <stdio.h>
C03       int x = 1;
C04       void f(){    printf("in f(): x = %d\n", x);    }
C05       int main()
C06       {
C07           int x = 7;
C08           printf("in main(): x = %d\n", ++x);
C09           f();
C10           return 0;
C11       }
```

08-07里面的 C03定义了全局外部变量 x，而 main 函数里面刚好又有一个自动变量 x，C08对 x 执行自增操作后打印 x 的新值，然后 C09调用函数 f()打印 x 的值，从结果就可以知道究竟 main 函数访问的 x 和函数 f()访问的 x 究竟是不是同一个 x：

```
$gcc  0807.c
$./a.out
in main(): x = 8
in f(): x = 1
```

显然，main 函数修改的是它自己内部的自动变量 x，打印出来的是自动变量自增后的值（7+1=8），而函数 f()访问的是 C03定义的外部变量 x，并且从数值上看出这个外部变量并没有受到任何修改，仍然保持初始值“1”。

不仅自动变量会屏蔽外部变量，自动变量之间也会发生屏蔽：

```
C01         /* Code 08-08, file name: 0808.c */
C02         int main()
C03         {
C04             int x = 1;
C05             if ( x !=0 )
C06             {
C07                 int x = 7;
C08                 printf("inner: x = %d\n", ++x);
```

```
C09             }
C10             printf("outer: x = %d\n", x);
C11             return 0;
C12         }
```

如果没有C07，那么C08修改、打印的x无疑是C04定义的自动变量x，但现在if复合语句里面又定义了自动变量x，结果会怎样：

```
$gcc 0808.c
$./a.out
inner: x = 8
outer: x = 1
```

结果表明，C08修改、打印的是C07定义的x，C10打印的x是C04定义的x（因为这时已经不在C07定义的x的作用域内），并且从数值上显示出C04定义的x根本没有被修改过。

*　　*　　*　　*　　*　　*

[0801]

注意，编译命令使用了“-O2”优化选项，如果没有“-O2”，*gcc* 为内部变量开辟的空间有时会超出实际的需要，造成浪费，不过，这种现象对我们本节的讨论没有任何影响。

[0802]

内存边界对齐的概念在后面的章节有详细的论述

09 外部变量的声明、定义和链接性质

我们知道，外部变量的声明（declaration）只是告诉编译器变量的类型，编译器并不会为声明中的变量分配存储空间，而定义（definition）则不同，不仅告诉编译器变量的类型，还会促使编译器为变量分配空间。因此，外部变量的声明可以重复多次，只要这些声明保持一致即可，但外部变量的定义则只能有一次，编译器不可能为同一个变量多次分配空间。声明和定义既有联系又有区别，当然，要正确区分外部变量的声明和定义很容易，关键是看两个方面：

- 有没有初始化部分
- 有没有使用 extern 关键字

如此一来就有4种情形：

带有初始化部分、没有 extern 关键字：

```
int i = 1;                              /* 定义 */
static double d = 1.0;                  /* 定义 */
```

没有初始化部分、使用 extern 关键字：

```
extern int x;                           /* 声明 */
extern double y;                        /* 声明 */
```

带有初始化部分、使用 extern 关键字：

```
extern int i = 1;                       /* 定义 */
extern double d = 1.0;                  /* 定义 */
```

没有初始化部分、没有 extern 关键字：

```
int i;                                  /* 暂时定义 */
static double d;                        /* 暂时定义 */
```

※ 在K&R C时代，C语言的很多设计仍然不够成熟，D.M.Ritchie在PDP-11上实现C语言时就允许外部变量在不同文件或同一文件中定义多次，只要满足一个限制：在这些定义中最多只能有一个带有初始化部分。后来，C 语言在逐渐完善的过程中已经抛弃了对同一个外部变量进行多次定义的做法，但由于有大量的旧式代码遗留下来，为了兼容这些代码，C89引入了暂时定义（tentative definition）的概念。 ※

暂时定义可以在同一文件或不同文件中重复多次：

```
C01         /* Code 09-01, file name: 0901.c
C02         int i;
C03         int i;
C04         int main()
C05         {
C06             i = 7;
C07             return 0;
C08         }
C09         int i;
```

09-01中的C02、C03、C09都是对同一个外部变量 i 的暂时定义。又例如：

main.c

```
int i;
int main(){
    i = 7;
    return 0;
}
```

var.c

```
int i;
```

上面的两个文件 main.c 和 var.c 里都有外部变量 i 的暂时定义。

对于暂时定义，如果相应的空间（全局或局部）存在相同名字的变量定义，则所有对该变量的暂时定义都会被链接程序忽略，一切就跟没有暂时定义一样；如果空间里没有相同名字的变量定义，则系统会为该变量分配一次存储空间，然后所有对该变量的暂时定义均被忽略。

下面我们通过具体的例子进行解释，首先查看一下09-01的汇编代码：

```
$gcc  -S  0901.c
$less  0901.s
```

相关内容如下：

```
A01             ...
A02         main:
A03             ...
A04             movl  $7, i
A05             ...
A06
A07             .comm  i,4,4
```

很显然，A07就是外部变量 i 的暂时定义所对应的汇编语句，“.comm”表示由于是暂时定义，编译器暂时还不会为该变量分配任何空间，后面的两个“4”表示如果将来为 i 分配空间，则 i 的大小是4个字节并且 i 的地址应该是4字节对齐。

汇编代码给我们的信息就这么多，要真的深入了解 i 的性质还得查看目标代码文件：

```
$gcc  -c  0901.c
$objdump  -xd  0901.o

0901.o:     file format elf32-i386
0901.o
architecture: i386, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x00000000

Sections:
Idx Name      Size      VMA       LMA       File off  Algn
```

```
0   .text      00000021  00000000  00000000  00000034  2**2
               CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
1   .data      00000000  00000000  00000000  00000058  2**2
               CONTENTS, ALLOC, LOAD, DATA
2   .bss       00000000  00000000  00000000  00000058  2**2
               ALLOC
3   .comment   00000012  00000000  00000000  00000058  2**0
               CONTENTS, READONLY

SYMBOL TABLE:
00000000     l    d    .text     00000000
00000000     l    d    .data     00000000
00000000     l    d    .bss      00000000
00000000     l    d    .comment  00000000
00000000     g     F   .text     00000021  main
00000004           O   *COM*     00000004  i

Disassembly of section .text:
00000000 <main>:
 0: 55                          push   %ebp
 1: 89 e5                       mov    %esp,%ebp
 3: 83 ec 08                    sub    $0x8,%esp
 6: 83 e4 f0                    add    $0xfffffff0,%esp
 9: b8 00 00 00 00              mov    $0x0,%eax
 e: 29 c4                       sub    %eax,%esp
10: c7 05 00 00 00 00 07        movl   $0x7,0x0
17: 00 00 00
                    12: R_386_32    i
1a: b8 00 00 00 00              mov    $0x0,%eax
1f: c9                         leave
20: c3                         ret
```

以上输出和以前的例子基本一致，在<main>的偏移0x12处需要链接程序用外部变量 i 的地址进行重定位，符号表里虽然有一个 i，但这个 i 既不在“.data”区也不在“.bss”区，***objdump*** 用一个特殊的标识“*COM*”表示它的属性。

而 ***nm*** 命令则用“C”表示“*COM*”数据：

```
$nm  0901.o
00000004 C i
00000000  T  main
```

“*COM*”数据不占用相应大小的磁盘空间，这一点跟“.bss”区的数据类似。因

为既然是暂时定义，那么变量肯定没有初始值，既然没有初始值，编译器也就无须浪费磁盘空间存储该暂时定义的变量。

最重要的是，链接程序不一定会为“*COM*”数据分配内存地址，这一点和“.bss”区的数据又有本质上的区别。“.bss”区的数据虽然不占用磁盘空间，但链接程序仍然会为它们分配地址。一旦程序被装载到内存开始运行，“.bss”区的数据在内存中一定会占据相应大小的空间。而“*COM*”数据是否由链接程序分配地址还得看是否存在相同名字的符号，如果有相同链接性质的符号（该符号代表“.data”或“.bss”区的数据），那么对应的“*COM*”数据将会被链接程序忽略；如果不存在相同链接性质的符号，“*COM*”数据就会拥有链接程序分配的内存地址，并且在最终的可执行文件中“升级”为“.bss”数据。

※ ***objdump*** 对0901.o的输出里面，字符串“*COM*”后的两个32位值所表示的意思和其它类型的数据不同，前面的值表示数据需要占用内存空间的大小，后面的值表示该数据的内存地址对齐要求，例如，i的大小是4字节，地址必须是4字节对齐。之所以如此，完全是因为“*COM*”数据不一定会被分配地址，因此在目标代码文件中它们是没有偏移量的，于是编译器必须利用两个32位值的其中一个存放“*COM*”数据的内存地址对齐要求，而另一个则用来存放数据的长度。为什么“.data”、“.bss”区数据不需要额外保存内存地址对齐要求呢？因为“.data”、“.bss”数据已经有偏移量，这个编译器所分配的偏移量本身已经满足数据的地址对齐要求。 ※

由于09-01没有全局外部变量i的定义，所以在链接完成后，原本作为“*COM*”数据的i会升级为“.bss”数据：

```
$gcc  0901.o
$nm  a.out
...
08049498 B i
```

为了验证如果存在相同名字的外部变量定义时暂时定义会被忽略，我们先用 ***size*** 命令查看0901.c生成的可执行文件：[1]

```
$size  a.out
   text    data     bss     dec     hex     filename
    658     252       8     918     396     a.out
```

然后编辑09-01a：

```
       C01         /* Code 09-01a, file name: 0901a.c
       C02         int i = 0;
```

编译09-01、09-01a后再次运行 ***size*** 命令：

```
$gcc  0901.c  0901a.c
$size  a.out
   text    data     bss     dec     hex     filename
    658     252       8     918     396     a.out
```

很明显，在新的可执行文件中，“.bss”数据仍然是占用8字节，这个数字和没有定义只有暂时定义的09-01生成的可执行文件是完全相同的。这充分证明了链接程序看到0901a.o中已经存在“.bss”数据i之后的确忽略了0901.o里面的“*COM*”数据i。

决定是否把暂时定义的变量升级为“.bss”数据的只能是链接程序，因为对于多个独立分布的C源文件来说,编译程序在编译阶段不可能知道在其它文件里是否存在相同名字的外部变量定义，从而没有办法决定是否升级暂时定义的变量为“.bss”数据。只有到了链接阶段，链接程序才有机会在所有的目标代码文件中查找是否存在同名的“.bss”数据来决定是否升级暂时定义变量。

但决定是否忽略暂时定义的可以是编译程序或者链接程序。当在同一个C源文件里已经存在相同名字的外部变量的定义时，编译程序马上就可以决定忽略该文件中的所有对该变量的暂时定义而不用等到链接阶段由链接程序来决定。把09-01的C2、C3、C9中的任意一个（只能是一个）改写成定义：

```
int i = 0;
```

然后编译成目标代码文件用 ***objdump*** 查看，是否还有“*COM*”数据的出现？我们会发现“*COM*”已经消失，取而代之的是“.bss”，因为这时的09-01内部已经存在同名外部变量的定义，编译程序完全可以决定忽略09-01里所有对变量i的暂时定义。

上面讨论了“*COM*”数据与同名“.bss”数据相遇链接程序忽略“*COM*”数据的情况，如果是“*COM*”碰到“*COM*”呢？这时候链接程序会提升其中的一个“*COM*”数据为“.bss”数据，然后忽略所有其余的“*COM*”数据。

例如：

x.c

```
int i;
int main(){
    i = 4;
}
```

y.c

```
int i;
```

z.c

```
int i;
```

分别编译，用 ***nm*** 命令查看：

```
$gcc -c x.c y.c z.c
$nm x.o y.o z.o
x.o:
00000004 C i
00000000 T main
y.o:
00000004 C i
z.o:
00000004 C i
```

可见3个目标代码都有一个“*COM*”数据i，现在我们链接x.o、y.o、z.o，生成可执行文件，再用 ***size*** 命令查看“.bss”区的大小、用 ***nm*** 命令查看符号表：

```
$gcc  x.o  y.o  z.o
$size  a.out
   text    data     bss     dec     hex    filename
    642     252       8     902     386    a.out
$nm  a.out
...
08049488 B i
```

尽管x.o、y.o、z.o里面都有“*COM*”数据i，但链接程序生成的可执行文件的“.bss”区大小和只有一个“*COM*”数据的09-01相同，均为8字节[2]，而且符号表显示只有唯一一个全局符号i，这就很有力地验证了前面的论述。

暂时定义是C标准为了兼容旧式C代码而作出的妥协性安排，一个合格的C程序员在开发中不应该再使用暂时定义。对于外部变量，要么声明（可以多次声明），要么定义（只能定义一次）。而为了让C编译器知道我们要“定义”变量，我们必须初始化外部变量。也就是说，不要写这样的代码：

```
int i;
```

因为这样的写法会被C编译器看作是暂时定义，应该这样写：

```
int i = 0;
```

※ C语言的暂时定义带来很多麻烦，所以C++摈弃了这种语法。在C++里面，只有声明和定义，带有初始化部分的是定义，没有初始化部分又没有使用extern关键字的也是定义（相当于默认初始值为零），其余的就是声明，例如：

```
extern int i = 2;                    // 定义
int i;                               // 定义
static int i;                        // 定义
extern int i;                        // 声明                    ※
```

为了保证外部变量类型在所有的文件中具有相同的类型，我们有必要重新认识全局外部变量的声明的意义。以前我们仅仅为了让编译器知道在另一个文件里进行定义的外部变量的类型而使用声明，现在，我们还要扩大声明的使用范围，即使在定义外部变量的文件中也要使用声明。这时候，声明相当于一份契约，正是这份契约保证了外部变量的使用和定义完全一致。下面有一个关于暂时定义的错误：

main.c

```
int a[1024];
int main()
{    a[1023] = 1;    }
```

var.c

```
int a = 0;
```

main.c的暂时定义中a是数组，而var.c的定义里a是int变量，既然存在相同名字的外部变量定义，于是链接程序会忽略数组的暂时定义。但由于两个目标代码文件中符号a所代表的数据长度不一致（一个是4096字节，另一个是4字节），链接程序会发出

警告：

```
$gcc  main.c  var.c
/usr/bin/ld: Warning: size of symbol 'a' changed from 4096 to 4
in /tmp/ccnYNCHc.o
```

然而，在这种明显错误的情况下，可执行文件还是被 *ld* 生成。在这里，链接程序的做法是把 a 作为 int 变量处理，分配给它4个字节的内存空间。假设 a 的地址是 V，则 main 的代码会访问地址（V+4092）开始的连续4个字节！

如果程序员没有注意到链接程序给出的上述警告，以为一切顺利，但一运行程序就出现莫名其妙的问题：

```
$./a.out
Segmentation fault
```

当然，在这里我们很清楚地知道这种错误发生的根源：进程试图访问的页面并不存在，系统进入异常处理，异常处理程序发现这个页面根本就没有分配给进程，这时进程会接收到系统发出的信号，在没有对该信号进行任何处理的情况下，系统强行终止进程并打印出错信息。

别以为一定要使用暂时定义才会导致失误，以下的例子没有使用暂时定义，但错误更隐蔽，至少链接程序没有发出任何警告：

文件 A

```
extern double x;
...
    x = 3.0;
...
```

文件 B

```
int x = 1;
int y = 2;
...
```

在 A 里面，编译器把 x 作为 double 变量（占内存8字节）进行处理；而实际上在文件 B 中 x 是 int 变量（占内存4字节），更要命的是紧接着 x 的是 int 变量 y 的定义，于是，错误不可避免地发生了。[3]

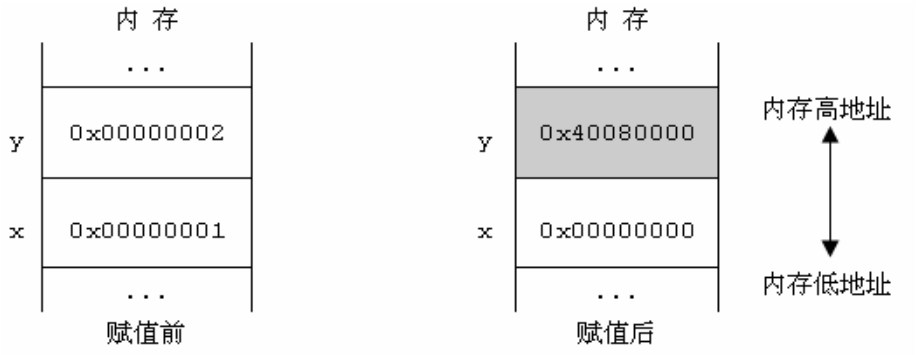


图 09-01

赋值语句（x = 3.0;）导致内存中8个字节的内容被修改，这很不幸地波及到变量 y。如图09-01，变量 y 的值由原来的0x00000002无声无息地变为0x40080000！如果变量 x 和变量 y 分别是由不同的代码模块负责访问而这不同的代码模块又刚好是由不同

的人员进行编写的话，要查出这种错误是非常耗费时间和精力的。如果文件 A、B 都同时使用一致的声明，编译器就能在编译阶段及时发现错误给出警告信息：

文件 A

```
extern double x;
...
    x = 3.0;
...
```

文件 B

```
extern double x;
int x = 1;
int y = 2;
...
```

```
error: conflicting types for 'x'
error: previous declaration of 'x'
```

同样，对 int 变量 y 也应作类似的处理才能保证在所有文件里 y 的类型均相同。

因此，我们通常把所有全局外部变量的声明放进一个（或多个，视具体情况例如工程的规模大小而定）头文件，然后通过#include 预处理指令包含这些头文件：

head.h

```
#ifndef HEAD_H
    #define HEAD_H
    extern double x;
    extern int y;
#endif
```

文件 A

```
#include "head.h"
...
    x = 3.0;
    y = 5;
...
```

文件 B

```
#include "head.h"
double x = 1.0;
int y = 2;
...
```

由此可知，头文件的作用之一就是为其它文件提供统一的全局外部变量声明。

在本节的最后，我们讨论一下外部变量的链接性质（linkage）。外部变量的链接性质有两种：外部的（external）和内部的（internal）。拥有外部链接性质的外部变量可以在全局空间被所有源文件的代码访问，而内部链接性质的外部变量则只能被与变量位于同一源文件的代码访问。

决定外部变量的链接性质的是两个关键字：extern 和 static，分别对应外部的和内部的链接性质。使用 extern 修饰的外部变量拥有外部链接性质，使用 static 修饰的外部变量拥有内部链接性质，既没有使用 extern 又没有使用 static 则外部变量的链接性质默认是外部的。例如：

```
extern int i = 1;          // 定义，变量的链接性质为外部的
static int n = 2;          // 定义，变量的链接性质为内部的
int x = 3;                 //  定义，变量的链接性质为外部的
int y;                     // 暂时定义，变量的链接性质为外部的
static int z;              //  暂时定义，变量的链接性质为内部的
```

在对应的汇编代码中，外部链接性质的变量就是全局符号，用“.globl”语句表示；内部链接性质的变量就是局部符号，用“.local”语句表示（或者既没有“.globl”又没有“.local”），例如：

```
int i = 2;              /* 定义，i 的链接性质是外部的 */
```

对应的汇编代码是：

```
    .data
    .align 4
.globl i
i:
    .long 2
```

这样，“i”就是全局符号，如果用 ***nm*** 命令查看符号表应该可以看到类似下面的信息：

```
xxxxxxxx D i
```

这表示 i 代表的是“.data”区的数据，并且 i 是全局可见的，所有对全局外部变量 i 的引用都可以在这里找到 i 的定义从而进行链接。

如果上面 i 的初始值是零，则 i 会被安排在“.bss”区：

```
    .bss
    .align 4
.globl i
i:
    .long 0
```

nm 命令的输出如下：

```
xxxxxxxx B i
```

现在，我们用 static 修饰外部变量：

```
static int i = 2;          /* 定义，i 的链接性质是内部的 */
```

则对应的汇编代码就是：

```
        .data                          .data
        .align 4                       .align 4
    .local i              或       i:
    i:                                 .long 2
        .long 2
```

用 ***nm*** 命令可以看到类似下面的信息：

```
xxxxxxxx d i
```

这表示 i 代表的是“.data”区的数据，但 i 是局部可见的，只有同一源文件的代码才可以访问这个 i。

同理，如果 i 的初始值是零，则 i 会被安排到“.bss”区：

```
xxxxxxxx b i
```

以上说的是定义，如果是暂时定义，情形也类似，例如：

```
int i;                  /* 暂时定义，i 的链接性质是外部的 */
```

对应的汇编代码是：

```
.comm  i,4,4
```

如果同一文件内不存在同名变量的定义时，则用 ***nm*** 命令观察编译后的目标代码文件

符号表如下：

```
xxxxxxxx C i
```

将来在最后链接的时候，如果存在同名的全局符号（属性为“D”或“B”）则该文件的符号 i（属性为“C”）被忽略；如果不存在同名全局符号，则链接程序把其中一个属性为“C”的符号 i 升级为属性“B”，对它进行链接，然后忽略所有的属性为“C”的符号 i。

而对应：

```
static int i;            /* 暂时定义，i 的链接性质是内部的 */
```

的汇编代码是：

```
.local i
.comm  i,4,4
```

稍有不同的是，无论是否存在同名的变量定义，用 ***nm*** 查看编译后的目标代码文件都会发现：

```
xxxxxxxx b i          或        xxxxxxxx d i
```

至于是“b”（“.bss”区数据）还是“d”（“.data”区数据）要看 i 的初始值，如果存在同名变量的定义并且初始值不为零，那符号 i 的属性就是“d”；如果存在同名变量的定义但初始值为零又或者不存在同名变量的定义因而编译器决定把暂时定义升级为定义则最后符号 i 的属性就是“b”。总之，由于是静态外部变量，所以编译器根本不用查看其它文件就可以决定是否把暂时定义升级为定义，这就可以解释为什么不存在属性为“C”的符号。

在 C 语言里，对同一个外部变量可以有多个暂时定义，但只能有一个定义。于是，当暂时定义与定义之间在链接性质的问题存在不一致时，情况便变得复杂起来。例如：

```
int i = 2;               /* 定义，i 的链接性质是外部的 */
static int i;            /* 暂时定义，i 的链接性质是内部的 */
```

像上面这种情况编译器一定会报错，因为对同一个外部变量 i，它的链接性质不可能既是外部的又是内部的，类似这样的冲突不可能解决。又譬如：

```
static int i;            /* 暂时定义，i 的链接性质是内部的 */
int i;                   /* 暂时定义，i 的链接性质是外部的 */
```

也会导致编译器报错，道理同上。

至于声明与暂时定义以及声明与定义之间存在的不一致，编译器一般会发出警告提醒程序员，例如：

```
extern int i;            /* 声明一个外部链接性质的变量 */
...
    ++i;
...
static int i = 2;        /* 定义，i 的链接性质是内部的*/
```

虽然按照声明，i 的链接性质应该理解为外部的[4]，但事实上，在链接的时候由于存在同名的局部符号，所以最后链接的是位于本文件的局部符号 i。编译器一般不会为这种

不算严重的问题报错，通常只是给出一条警告信息：

```
warning: static declaration for `i' follows non-static
```

*　　*　　*　　*　　*　　*

[0901]

变量 i 只占4字节，***size*** 命令输出8字节是因为还有其它存放在“.bss”区的数据。其实，我们不必在意具体数字，只需要注意该数字是否发生改变即可。

[0902]

参见[0901]。

[0903]

double 型数值3.0以16进制表示为：0x4008000000000000。

[0904]

声明一个外部变量意味着告诉编译器在其它某个源文件中存在该变量的定义，尽管对该变量的定义完全可以在同一文件里，因此，声明一个外部变量 x 就其意义本身来说相当于默认这样一个事实：

某个源文件里定义了一个链接性质为外部的变量 x，以下代码要访问的 x 就是它了……

因此，不存在“对静态外部变量进行声明”这么一回事。静态外部变量通常都是在源文件一开始的地方立即进行定义，实践中并没有声明静态外部变量这一做法。说到这里大家应该可以明白为什么对外部变量进行声明会使用 extern 关键字，并且我们从来没有看到诸如：

```
extern static int i;
```

之类的声明。记住：对于静态外部变量，只有暂时定义或定义，没有声明。

10 函数的原型声明和链接性质

函数实质上就是一段代码，调用一个函数意味着执行某段代码。在前面汇编语言的章节中我们已经了解到，函数的名字其实也是一个符号，编译程序用指令：

```
call  funcName
```

进行函数调用，稍后由链接程序找到符号 funcName 的定义之处然后通过计算确定 funcName 的合适地址并填入 call 指令的相应位置，这个过程叫做重定位。编译程序只要判断出某一个 C 语句是函数调用就可以很轻易地通过 call 指令实现，因此，正确调用函数的关键在于准确地传递参数，在调用函数的外部环境与函数内部之间就参数的个数及类型取得一致的理解。在 K&R C 时代，我们可以直接调用函数而无须给出任何声明：

```
C01       /* Code 10-01, file name: 1001.c
C02       int main()
C03       {
C04           f();
C05           return 0;
C06       }
```

尽管直到 C04仍然没有任何关于函数 f()的声明,但这完全不影响编译器判断出 C04 是一个函数调用语句。我们用“-S”选项编译10-01后查看生成的汇编代码如下：

```
A01            .text
A02         .globl main
A03            .type  main,@function
A04         main:
A05            ...
A06            call  f
A07            ...
A08            .size  main, .-main
```

可以看出，编译程序判断出 C04是函数调用语句后生成对应的 call 指令，在输出目标代码文件之后，剩下的工作则由链接程序来完成。因此，如果函数的调用与定义存在不一致就会造成严重错误：

x.c

```
int main(){
    int s = sum();
}
```

y.c

```
int sum(int a, int b){
    return (a + b);
}
```

我们很容易就能看出 x.c 和 y.c 在函数 sum()的参数上的不一致，但上面的程序却可以顺利通过编译：

```
$gcc  x.c  y.c
```

之所以会这样，完全是因为 K&R C 不对函数的参数进行类型检查[1]，程序员给出什么参数，编译器不管三七二十一，进行必要的调整后就压栈、然后调用函数。在一开始的时

候 C 语言的基本数据类型只有4种：char、int、float 和 double，前两个属于整数类，后两个属于浮点实数类。当时的 C 语言是这样规定的：

对于参数（可以是变量或常数），如果是 char 则提升为 int，如果是 float 则提升为 double；对于返回值和参数，如果没有指明类型的就默认是 int。例如：

```
char c = ...;
f(c);
```

由于参数 c 是 char 值，基于上述原则，编译器首先把 c 提升为 int 类型然后用该 int 值作为参数调用函数 f()。而在函数 f()内部，编译器通过函数定义：

```
f(x)
char x;                  /* K&R C 风格的函数定义*/
{...}
```

同样知道传过来的参数 x 本来是 char 值，现在已经被提升为 int 值。又譬如：

```
float f = ...;
g(f);
```

由于参数 f 是 float 值，基于上述原则，编译器首先会把 f 提升为 double 值，然后再以该 double 值作为参数调用函数 g()。而在函数 g()内部，编译器通过函数定义：

```
g(x)
float x;                  /* K&R C 风格的函数定义*/
{...}
```

同样知道参数 x 本来是 float 值，现在已经被提升为 double 值。

这个规则一直沿用到 K&R C。为什么 C 语言会对函数的参数作出这些默认的提升规则呢？很简单，因为这样会简化编译器的设计工作。既然 char 会被提升为 int，float 会被提升为 double，而指针类型通常和 int 的长度相同，也就是说，在函数里面编译器实际上只需要处理两种数据类型（int 和 double）即可。

前面关于 K&R C 风格的函数定义都没有包括返回值的类型，因为编译器对没有明确指出类型的返回值或参数都默认为 int，例如：

```
f(x, y, z)
char x; float z;          /* K&R C 风格的函数定义*/
{...}
```

函数 f()的返回值是 int，参数有3个：x（类型是 char）、y（代码没有指出类型，于是编译器默认是 int）和 z（类型是 float）。

我们在 K&R C 里面几乎不需要声明函数，除非函数的返回值是浮点实数类型（float 或 double）。例如下面这个 K&R C 风格的程序：

```
C01         /* Code 10-02, file name: 1002.c */
C02         main()
C03         {
C04             double sum();
C05             double d = sum(1.2, 3.4);
C06             return 0;
```

```
C07         }
C08         double sum(a, b)
C09         double a, b;
C10         {    return (a + b);    }
```

注意，10-02的C04不能省略，如果没有C04，错误就会出现——因为如果不预先声明函数sum()返回浮点实数类型值，则编译器一看到C05就知道sum是一个函数，并且默认它的返回值是int类型。由于系统返回整数和返回浮点实数所用的寄存器根本不一样，于是在main函数里编译器认为sum用返回整数的寄存器返回int；而在sum内部，编译器用返回浮点实数的寄存器返回double值，一切都乱套了。

不过，C04的函数声明没有包含任何参数的类型信息，只有光秃秃一对括号，这就是K&R C函数声明的特点：括号内没有任何字符。因此，K&R C的函数声明对参数仍然无法进行类型检查。为了改善这种情况，C90给C语言增加了很重要的一项特性[2]：原型声明（prototype declaration），例如：

```
int f(int, double);          /* 原型声明 */
f(123, 4.56);                /* 函数调用 */
```

在原型声明的那对括号里面有了很重要的信息：

函数f()要有2个参数才能调用，且参数的类型分别是int和double。如果我们传递的参数个数不对，编译器将会报错。例如：

```
int f(int, double);         /* 原型声明 */
f(123);                     /* 错误，这行代码不能通过编译 */
```

值得注意的是，编译器调用有原型声明的函数时不再根据K&R C的规则对参数进行提升，例如：

```
float f = 1.0F;
g(f);                       /* 参数f会被提升为double值 */
```

上面的函数g()没有原型声明，所以编译器会依照K&R C的惯例先把float值参数f提升为double值再以该double值作参数调用g()。如果这样写：

```
int g(float);               /* 原型声明 */
float f = 1.0F;
g(f);                       /* 参数f不会被提升为double值 */
```

编译器看到原型声明，就不再按照K&R C的老规则行事。

我们不能把C90引进的原型声明和老式的K&R C函数定义搭配在一起使用，例如：

x.c

```
int g(float);
int main()
{
    float f = 1.0F;
    g(f);
}
```

y.c

```
g(a)
float a;
{
    ...
}
```

上面 y.c 的函数定义是 K&R C 风格的，编译器看到这种风格的函数体时，自然会假定调用这个函数的时候参数会按照 K&R C 惯例进行调整。于是编译器会认为传递进来的参数 a 是一个已经被提升为 double 型值（8字节）；而实际上，在 x.c 中，由于有原型声明，编译器的的确确是以 float 值（4字节）作为参数调用函数 g()的。这样，错误便产生了。同样的道理，如果没有原型声明就调用函数，而函数又是用 C90风格定义的，混乱一样会出现，例如：

x.c

```
int main()
{
    float f = 1.0F;
    g(f);
}
```

y.c

```
int g(float a)
{
    ...
}
```

这一次调用函数 g()的时候由于没有看到原型声明，编译器就认为这是 K&R C 代码，参数 f 会被先提升为 double 值（8字节）再传递进函数；而 y.c 的函数定义是 C90风格，编译器自然认为参数 a 是原汁原味、没有经过任何提升的 float 值（4字节），误会再一次发生了。

不像 C++，C90并没有要求程序员一定要使用原型声明，因为 C90要保持对 K&R C 的兼容。早期的 C 代码都是按照 K&R C 编写的，累积到 C90出台的时候，这些代码的数量已经相当可观，为了让这些代码继续工作一段时间，C90还是保留了两道缺口：

1）程序员可以不预先声明就调用函数。这时，相关的规则仍然和 K&R C 一致。

2）K&R C 风格的函数声明仍然起作用。

尤其要注意的是，括号里面没有任何东西的函数声明不是原型声明，编译器不会为使用 K&R C 声明的函数检查参数：

```
int f();
f(123);                    /* 可以通过编译 */
f(4.5, 8.7);               /* 可以通过编译 */
```

如果要声明一个不带任何参数的函数原型，必须加上 C90引入的新关键字“void”：

```
int f(void);
f();                       /* 可以通过编译 */
f(123);                    /* 错误，不能通过编译 */
```

另外，原型声明不仅对函数调用有约束作用，其实它对函数的定义一样有效。例如：

```
int f(int);
int f(double){...}         /* 错误，不能通过编译 */
```

上面的代码不能通过编译，因为原型声明和函数定义不一致，即使函数 f()可能根本就没有被调用过。

※ 如果没有函数原型，C 编译器就会自动提升函数参数的类型，这个规则从 K&R C 一直延续到今天的 C99，它的影响力无疑是相当大的。

C 的初学者常常会对 printf()家族函数感到不解，就以最简单的 printf()函数为例，为什么它

可以打印所有长度的整数类型以及 double、long double 浮点实数，偏偏就不能打印 float 值呢？

由于 printf()函数的参数个数是可变的，所以，默认的类型提升规则总是会起作用，即使有函数原型也如此：

```
int printf(const char * format, ...);
```

根据这份原型声明，编译器除了知道最左边的参数肯定是一个指针（const char*）之外就什么都不知道了，它能够做的事，就是按顺序地把参数进行类型提升然后复制到栈里面。哪些参数需要类型提升呢？这就是老规则的管辖范围了，凡是长度比 int 小的就提升为 int，float 提升为 double，然后调用函数，这样编译器就完成任务了。

而在 printf()函数内部，C 的 I/O 库要求实现者必须把已经提升过的参数用强制转换还原回来，这是为了避免打印出来的数值超出范围。譬如，某个程序员这样调用 printf()函数：

```
int i = 0x12345678;
printf("%Here is a value with short type: hd\n", i);
```

本来，printf()函数期待的是一个被提升为 int 类型的 short 值，如果 printf()函数完全信任调用函数的人，它可能就会把 int 值0x12345678直接送到进行下一步处理的函数去，因为它不知道这个0x12345678其实不是经由一个 short 值提升得到的、0x12345678根本就是一个错误的参数。于是，在下一步进行处理的函数就会把0x12345678打印在屏幕上：

```
Here is a value with short type: 305419896
```

很明显，一个 short 值不可能是305419896。I/O 库的设计者认为我们必须避开如此尴尬的场面，所以标准库要求 printf()函数在进行进一步处理参数值之前首先要检查该参数是否经过类型提升，如果是的话就要对参数实行强制转换，以上面的代码为例，printf()函数由字符串“%hd”就知道应该先对0x12345678进行强制转换：

```
int temp = (short)v;                    /* 假设 v 代表传过来的参数 */
```

这样的话，temp 的值就是(short)0x12345678，即0x5678，接下来 printf()函数就可以把 temp 传给进行下一步处理的函数，可以预料的是，经过这样的处理，到最后一定不会出现上面的奇怪现象。

对于整数类型，要把长度大的值强制转换为长度小的值是非常容易的。譬如说，原来的参数是16位的，那么只需要取 int 值的低16位就行。不过，对于浮点实数类型事情就不是这么简单，要把 double 值转换回 float 值虽然不算是一件十分复杂的工作，但这种操作涉及到浮点运算异常[3]，为了避免出现复杂的局面，C 的 I/O 库从设计的一开始就绕开了这个问题，printf()函数直接把 double 值传给进行下一步处理的函数，这样做的结果就是我们看不到 float 类型出现在 printf()函数的打印名单上。这种状况一直存在到今时今日，即使是最新的 C99标准库，printf()函数也一样没有打印 float 值的选项。 ※

那么，如果函数的声明、原型声明与定义之间存在不一致会怎样呢？下面给出若干原则：

- （同一地点能够看到的多个）原型声明不能有任何的不一致。
- （同一地点能够看到的多个）K&R C 风格声明的返回值类型必须一致。
- K&R C 风格的函数声明和原型声明可以“和平共处”，前提是返回值类型必须相同。
- 是否对函数参数进行检查以当时是否已经看到原型声明为准。
- （同一地点能够看到的）声明（包括 K&R C 风格声明和原型声明）与函数定义的

返回值类型必须一致。

- （同一地点能够看到的）函数的原型声明和定义必须完全一致。
- 函数定义本身也意味着一种声明，编译器按照函数定义的风格去确定如何调整参数，但无论如何，调用时的参数个数必须和定义时的一致。[3]

在本节的最后，我们讨论一下函数的链接性质。

C 语言允许将代码分布在不同的源文件里面，各自独立编译，然后把所有（需要用到的）目标代码文件、库链接起来就能够产生可执行文件。所以，和外部变量一样，函数也有链接性质，这关系到某个文件的代码能否调用其他文件定义的函数。

不过，对于函数来说，链接特性并不复杂，只有两种情况：“外部的”和“内部的”，如果没有使用 static 关键字，函数就是“全局可见的”，具有外部的链接性质，即任何一个文件定义的函数都能在其他任何一个文件中调用。

全局函数在汇编代码中通常有如下形式：

```
    .text
.globl  f
    .type  f, @function
f:
    ...
    .size  f, .-f
```

正是“.globl f”语句把 f 定义为全局符号，因此所有目标代码文件都能对 f 进行引用，如果用 ***nm*** 查看符号表，则有类似下面的输出：

```
$nm  example.o
...
xxxxxxxx  T  f
```

“T” 表示 f 是一个全局符号、代表一段代码。

经过 static 修饰的函数具有内部的链接性质，只能被同一文件的代码进行调用，其他文件根本无法看到 static 函数的存在从而不能进行引用。static 函数的汇编形式如下：

```
    .text
    .type  f, @function
f:
    ...
    .size  f, .-f
```

容易看出，仅仅是少了一句“.globl f”而已。如果用 ***nm*** 命令查看，则有：

```
$nm  example.o
...
xxxxxxxx  t  f
```

“t”表示 f 是一个局部符号、代表一段代码。

和外部变量一样，函数有可能在原型声明与定义里对链接性质的描述存在不一致，如果真的发生这种情况会怎样？

与外部变量不同，函数没有暂时定义和定义之分，所以编译器的处理原则很简单。究竟函数的链接性质被处理成外部的还是内部的，关键在于函数定义本身以及出现在定义之前的原型声明中是否用了 static 关键字修饰，如果有，函数的链接性质就是内部的，反之则是外部的。

例如：

```
extern int f(int);
...
static int f(int x){...}            /* 内部的链接性质 */
```

上面的函数定义里明确使用了 static，所以 f()的链接性质是内部的。

```
static int f(int);
...
int f(int x){...}                   /* 内部的链接性质 */
```

上面函数定义的前面已经出现了使用 static 的原型声明，因此 f()的链接性质是内部的。

```
int f(int x){...}                    /* 外部的链接性质 */
static int f(int);
```

上面的函数定义出现在原型之前，又没有使用 static 修饰，所以 f()的链接性质是外部的。

当然，对于所有链接性质不一致的情形，编译器都会发出相应的警告。

最后，提醒一下，如果我们想声明一个 static 函数，不要这样写：

```
int main()
{
    static int f(void);
    ...
}
```

原因有二：

1）编译器会误以为你想声明一个存储性质是 static（就像静态内部变量那样）的函数，于是它会给你警告信息（放心，仅仅是警告），因为函数是没有存储性质的（变量才有）。

2）如果你真的想声明一个 static 函数，那么应该在函数的外面声明，这样才能保证在声明之后定义的那个函数的确被处理成 static 函数。

例如：

```
static int f(void);
int main(){...}
```

```
int f(){...}                /* f 被处理成 static 函数 */
```

如果这样写：

```
int main()
{
    static int f(void);
    ...
}
int f(){...}                /* f 仍然是全局函数 */
```

因为声明同样有可见范围，上面在 main 函数里面声明 f，于是这个声明只在 main 内部有效。当到了 main 的外面，编译器就看不到 f 的前面有 static 修饰，于是 f 的链接性质还是被处理成外部的。

*　　*　　*　　*　　*　　*

[1001]

应该说，K&R C 根本就没办法检查。函数的定义是允许放在其他文件里面的，在引入函数原型之前，这个问题无法解决。而 C90虽然有了函数原型从而可以对函数进行类型检查，但 C90为了兼容 K&R C 的代码，在没有原型声明的情况下照样不会对函数进行类型检查。在没有原型声明的情况下，除非函数的定义位于同一文件中并且先于函数调用语句出现，否则编译器无法对函数进行参数检查。

[1002]

这个特性是从 C++“引进”的。

[1003]

为什么这里只是说参数个数必须一样呢？因为 C 语言存在类型转换：

```
void f(int);
f(12.34);                   /* 这一句没有错误 */
```

尽管在定义的时候 f()的参数是 int 类型，而调用时 f()的参数是 double 型常数12.34，但这不会引起任何编译错误和警告，因为 C 编译器认为你要它先把12.34强制转换成整数（即12）然后再用这个整数作参数调用 f()，而这是 C 语言允许的行为。

[1004]

浮点运算异常

11 内存地址对齐

在前面的小节里，我们经常碰到下面这条汇编语句：

```
.align  X                          # X通常是4或者8
```

大家可能已经对此有一些初步的了解，该语句的作用是指出变量的地址对齐要求，编译器会把这信息记录下来，将来链接程序为变量安排内存地址时就会作出适当的调整。例如，有一个32位的外部变量 i 的定义如下：

```
    .data                  # 数据会被安排在“.data”区
    .globl  i              # 链接属性是外部的
    .align  4              # 给该变量分配的地址必须是4字节对齐
i:
    .long  1234            # i 的初始值是1234
```

当链接程序给 i 分配地址时，它会看到目标代码文件的相关记录，从而知道 i 的地址必须是4字节对齐，经过一系列的计算、调整，到最后 i 的地址的二进制形式肯定是(“*”号代表任意的二进制数字)：

```
******** ******** ******** ******00
```

换算成十六进制，如果用“*”号代表任意的十六进制数字，则下面的4种地址：

```
*******0
*******4
*******8
*******C
```

都符合对齐要求，因而均有机会分配给 i；而其它形式的地址，例如：

```
*******2
*******7
```

等等就不符合要求，从而不可能分配给 i。

以 P6处理器为例，使数据的地址按一定的边界对齐主要有三方面的考虑：

- 提高效率

P6处理器读/写内存中的数据需要一个或数个内存总线周期，对于8位数据，处理器一个周期就可以访问完毕；对于16位数据，如果数据的地址是2字节对齐（被2整除），P6处理器耗费一个总线周期就可以访问完毕；如果不是双字节对齐，但只要它的地址没有横跨在相邻两个对齐的32位数据地址上，则处理器仍然只需一个内存总线周期就可完成操作，否则处理器需要两个周期才能完成操作；对于32位数据，如果数据的地址是4字节对齐（被4整除）的话，P6处理器耗费一个总线周期就可以访问完毕，否则需要两个周期进行操作；对于64位数据，如果数据的地址是8字节对齐（被8整除）的话，P6处理器耗费一个总线周期就可以访问完毕，否则需要两个周期进行操作。可见，对于最常用的32/64位数据，访问地址对齐的数据比访问地址没有对齐的数据在理论上节省50%的时间，因此，我们可以看到 ***gcc*** 在默认情况下生成的汇编代码总是把数据按照对应的对齐规则指示链接程序安排适当的地址。

由于要满足对齐规定，因此某些情况下部分数据要占据更多的空间，例如：

```
struct s
{
    char c;              /* sizeof(struct s)等于8 */
    int i;
};
```

上面的struct s会占用多大的空间呢？虽然char和int分别占用1个和4个字节，但 struct s 在默认条件下占用的空间不是5个字节而是8个字节。基于对齐的考虑，编译器会在成员 c 和成员 i 之间填充3个字节从而把 struct s 凑够8个字节。

原因很简单，成员 i 是32位数据，为了尽量提高效率，它的地址必须满足4字节对齐，但成员 c 只占一个字节，如果不作出调整那么只要碰到稍微复杂一点的情形就无法保证成员 i 的地址是4字节对齐，譬如数组：

```
struct s arr[10];           /* arr 的地址该如何设置？ */
```

唯一的办法就是加入一些填充字节，让成员 i 从成员 c 后面跳过3个字节再开始并且

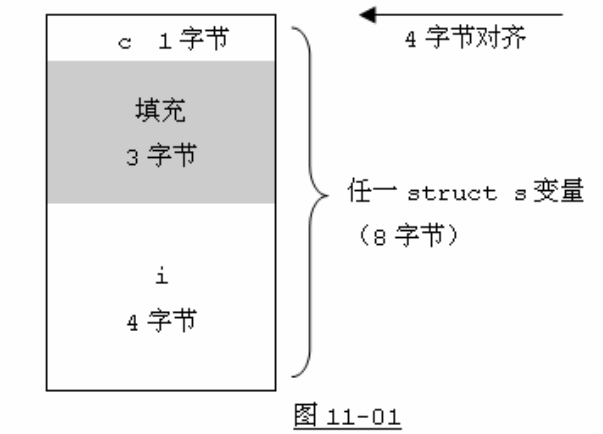


图 11-01

把所有 struct s 变量的地址定为4字节对齐即可：

由于整个 struct s 变量的地址是4字节对齐，而成员 c 和填充字节一共又占用4字节，因此到最后成员 i 的地址肯定同样是4字节对齐。

编译器这种“用空间换时间”的策略固然可行，但我们必须了解清楚该策略可能造成的副作用，从而尽量避免空间的浪费。例如：

```
struct s
{
    char x;          /* z 本身占用1字节，编译器需要填充3字节 */
    int y;           /* y 占用4字节 */
    short z;         /* z 本身占用2字节，编译器需要填充2字节 */
};
```

上面是一个失败的 struct 布局安排，编译器为了满足成员 y 的4字节对齐要求，它必须要填充5个字节，导致 struct s 最后总共要占用12字节，但事实上，只要稍微改变一下数据成员的声明次序我们马上就可以节省33%的空间[1]：

```
struct s
{
    char x;             /* x 本身占用1字节，编译器再填充1字节 */
    short z;            /* z 占用2字节 */
    int y;              /* y 占用4字节 */
};
```

如果程序使用到大量的 struct s 变量时，节省下来的数据空间便相当可观。

通过下面简单的代码可以总结 *gcc* 对于外部变量的地址安排：

```
C01         /* Code 11-01, file name: 1101.c */
C02         short s = 2;
C03         int i = 3;
C04         long l = 4;
C05         long long ll = 5;
C06         float f = 1.0;
C07         double d = 2.0;
C08         int main(){}
```

把11-01编译成汇编代码文件，其中关于对齐的主要信息如下：

```
.globl s                .globl ll
   .align 2                .align 8
.globl i                .globl f
   .align 4                .align 4
.globl l                .globl d
   .align 4                .align 8
```

自然地，short 变量 s 占2字节，地址被安排为2字节对齐；int 变量 i、long 变量 l、float 变量 f 各自占4字节，地址都被安排为4字节对齐，long long 变量 ll、double 变量 d 各自占8字节，地址均被安排为8字节对齐。对于更复杂的情况，例如数组或者结构体，*gcc* 根据大致相同的原则进行调整，无论如何，最后的布局肯定会满足每一个成员自身对地址的对齐要求。

以上是外部变量的地址对齐情况，我们现在分析一下 *gcc* 如何保证自动变量的地址对齐。众所周知，自动变量被安排存放在栈里面，当函数被实际执行时，该函数的栈才真正得到需要的空间。前面我们已经知道 P6处理器是如何通过 ESP 和 EBP 寄存器访问栈里面的自动变量，归根到底，ESP 和 EBP 存放的仍然是内存的逻辑地址，处理器通过这些逻辑地址访问自动变量，因此，为了提高效率，和外部变量一样，自动变量的地址同样被编译器按照一定的规则进行安排以满足地址对齐的要求。而且，由于在函数内部对自动变量的访问次数相当频繁，所以自动变量的地址是否对齐对程序的性能有非常大的影响。

我们先来看看 main()函数：

```
C01         /* Code 11-02, file name: 1102.c */
C02         int main(){}
```

上面是一个最简单的 main()函数，编译11-02：

```
$gcc  -std=c99  -S  1102.c
$less  1102.s
```

对应的主要汇编代码如下：

```
A01         main:
A02                 pushl   %ebp
A03                 movl    %esp, %ebp
A04                 subl    $8, %esp
A05                 andl    $-16, %esp
A06                 movl    $0, %eax
A07                 subl    %eax, %esp
A08                 movl    $0, %eax
A09                 leave
A10                 ret
```

无论 ESP 的具体数值是多少，经过 A05之后，ESP 的值一定能被16整除，就是说，main()函数在通过 call 指令调用其它函数之前 ESP 的值肯定是16字节对齐的，有了这个约定，编译器在其它函数内部便知道应该如何分配自动变量的地址，因为其它函数的一开始总是先在栈里保存 EBP 的值：

```
A01         f:
A02                 pushl   %ebp
A03                 movl    %esp, %ebp
A04                 ...
A05                 leave
A06                 ret
```

于是，编译器完全可以预料到：当进入其它函数保存 EBP 之后，ESP 的值一定是8字节对齐的。如图11-02，在 call 指令执行前的瞬间，ESP 的值能被16整除，由于 call 会把返回地址压栈，所以在进入其它函数后 ESP 的值减少4，然后 EBP 的值入栈，ESP 的值再减少4，这样，ESP 的值一共减少8。因此，这时 ESP 的值一定可以被8整除。

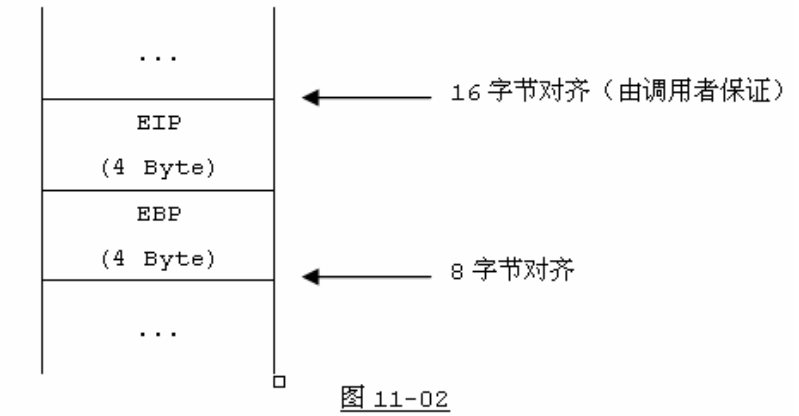


图 11-02

编译器按照自动变量的类型分配适当的地址以满足对应的对齐要求，如果自动变量的声明次序不当，浪费空间的现象同样存在。例如：

```
void g();
void f()
{
    int i;
    double x;
    int n;
    double y;
    g();
}
```

函数 f()内部有4个自动变量，分别是 int 变量 i、n 和 double 变量 x、y，表面

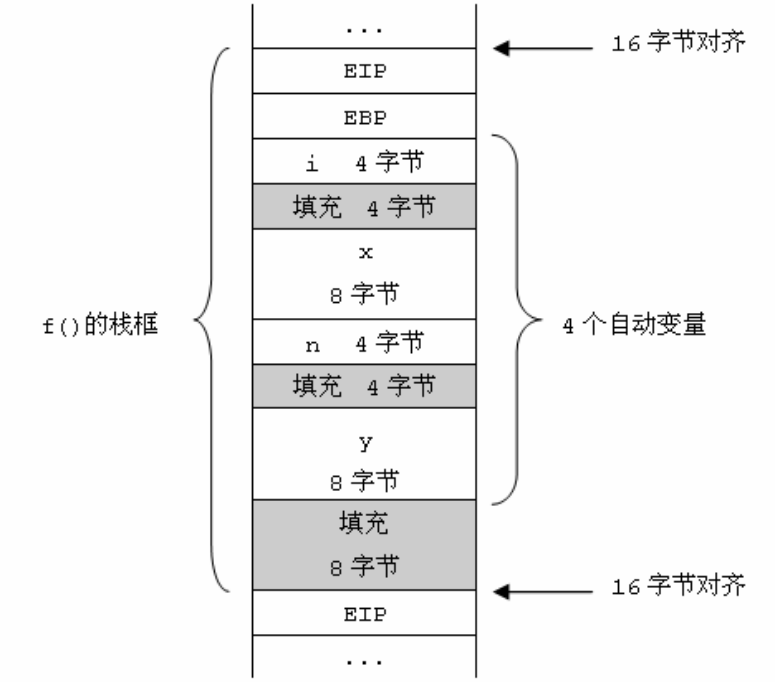


图 11-03

上看来，4个变量只需24字节的栈空间就足够，实际上，进入函数 f()后，为了给自动变量分配适当的地址，ESP 必须至少减去40。如图11-03，按照 int 变量地址4字节对齐、double 变量地址8字节对齐的规则，以及保证在调用函数 g()之前 ESP 的值必须被16整除，编译器不得不分配40字节的空间，其中的16字节纯粹是为了填充，空间的浪费率达40%。

只要稍微修改一下，调整自动变量的声明次序，编译器就不用浪费这么多的空间去填充间隙：

```
void g();
void f()
{
```

```
    int i;
    int n;
    double x;
    double y;
    g();
}
```

这次我们把两个 int 变量的声明紧挨着放在一起，于是原先用来填充的4个字节刚好可以成为变量 n 的空间，这样一来，其它填充字节也不需要了，编译器仅仅把 ESP 的值减少24就可以满足所有的对齐要求。

显然，修改后的函数 f()不存在任何的空间浪费，如图11-04。

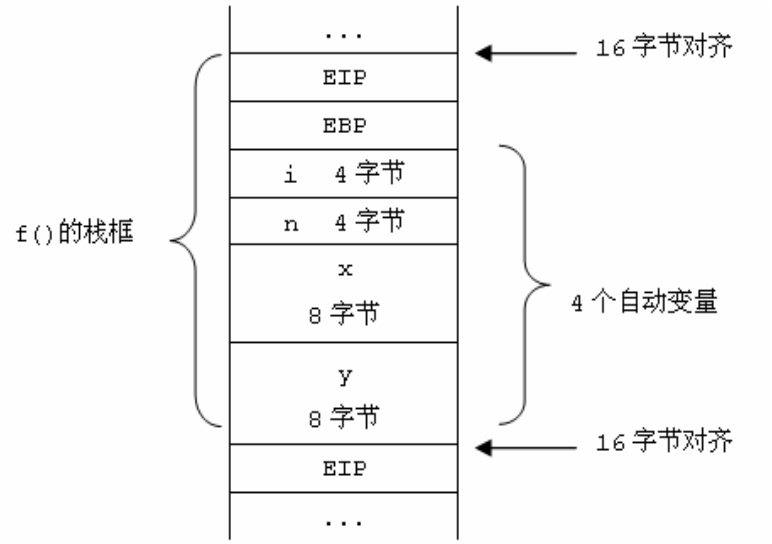


图 11-04

以上讨论的仅仅是单个变量的简单组合，如果自动变量涉及到结构体或数组，情形将会更为复杂。例如 ***gcc*** 默认情况下对元素个数大于1的 double 数组在栈里面分配空间时会把首个元素的地址安排为16字节对齐，可以想象，不良的声明次序会导致更多的空间浪费。

▪ 某些 SSE、SSE2指令要求内存操作数的地址必须是16字节对齐的，否则就引发异常。Pentium III 处理器开始支持 SSE 指令，Pentium 4处理器开始支持 SSE2指令，而这些指令的其中一部分需要地址是16字节对齐的内存操作数，例如：

movdqa（SSE 指令）

movapd（SSE2指令）

当使用这些指令并且其中一个操作数是内存操作数时，内存操作数的地址必须满足16字节对齐，否则处理器引发13号异常：

```
C01         /* Code 11-03, file name: 1103.c */
C02         double array[4];
```

```
C03         int main()
C04         {
C05             __asm__("movdqa  array, %xmm0   \n\t"
C06                     "movdqa  array+4, %xmm0");
C07         }
```

编译后（在 Pentium III/4处理器系统上）执行11-03：

```
$gcc  1103.c
$./a.out
Segmentation fault
```

Linux 对13号异常的处理同样是给进程发送 SIGSEGV 信号，如果进程没有捕获该信号进行处理则系统立即终止进程并打印出错信息。

- 保证对内存的访问是原子操作（atomic operation）

在多处理器环境下，为了保证各个处理器对内存操作的正确性，系统必须确保某些读/写操作在一个内存总线周期内完成。从前面我们已经知道，对于地址满足适当对齐条件的16/32/64位数据，P6处理器均可以在一个内存总线周期内完成访问，这很自然地满足原子操作的要求。如果数据地址不满足对应的对齐条件，那么处理器必须用2个内存总线周期才能完成读/写操作，万一在这2个周期之间其它处理器刚好获得总线的使用权并且在随后的连续2个周期内访问到该数据，则该访问的结果将是不正确的。虽然 P6处理器可以使用 LOCK 命令前缀在一条指令执行完毕之前锁住总线的使用权，但 LOCK 前缀的使用范围有限，并不是所有指令都可以加上 LOCK 前缀，例如最常见的 MOV 指令就不能使用 LOCK 前缀。因此，要保证数据的完整性，最好让数据的地址按照适当的规则对齐从而保证处理器访问内存是原子操作。[2]

*　　*　　*　　*　　*　　*

[1101]

大家可能会问，编译器自己偷偷把 x、y、z 的次序重新安排以便满足 y、z 的对齐要求就行了，何必要我们亲自动手呢？这是因为 C 标准规定，对于 struct 成员，编译器最后的布局安排必须符合“先声明的成员的地址偏移比后声明的成员的地址偏移小”这么一条规则，因此编译器可以私自填充字节，但不能擅自修改成员之间的顺序。

[1102]

仅仅依靠数据地址对齐无法保证所有竞争条件下的数据完整性，多处理器环境下需要正确的算法和相关指令（例如 LOCK 前缀和 BTC 指令等）配合，这里讨论的只是处理器访问内存的最低层操作的原子性，设计具体的算法时可以作为参考。

#2

数值运算

12 整数类型

粗略地概括，整数类型分为两大类：无符号整数（unsigned integer）和带符号整数（signed integer）。

对于无符号整数，一般来说，所有的数位都被用来表示数值[1]。就是说，一个 N 位的无符号整数，每个数位对应的权值是 2^k（k = 0,1,2...N-1）：

10010011 00101010 10110111 11001001

如果上面是一个32位无符号整数，则它的数值是：

$2^0+2^3+2^6+2^7+2^8+2^9+2^{10}+2^{12}+2^{13}+2^{15}+2^{17}+2^{19}+2^{21}+2^{24}+2^{25}+2^{28}+2^{31}$

即2469050313。

容易得出结论，一个 N 位的无符号整数能够表示的数值范围是[0, 2^N-1]，因此，无符号整数在任何时候的数值都是非负的。

对于带符号整数，数位被划分为符号位（只能占一位）与数值位（剩余的其它位）。一个 N 位的带符号整数，一般来说，最高位是符号位，其余的都是数值位，每个数值位对应的权值同样是 2^k（k = 0,1,2...N-2）；而符号位的权值在不同的计算机体系中可能有不同的解释：

1）权值是零（符号位是“1”则表示数值为负）

2）权值是 $-(2^{N-1})$

3）权值是 $-(2^{N-1}-1)$

例如：

10010011 00101010 10110111 11001001

如果上面是一个32位带符号整数，则它对应于上述三种情形的数值分别是：

$-(2^0+2^3+2^6+2^7+2^8+2^9+2^{10}+2^{12}+2^{13}+2^{15}+2^{17}+2^{19}+2^{21}+2^{24}+2^{25}+2^{28})$

$2^0+2^3+2^6+2^7+2^8+2^9+2^{10}+2^{12}+2^{13}+2^{15}+2^{17}+2^{19}+2^{21}+2^{24}+2^{25}+2^{28}+(-2^{31})$

$2^0+2^3+2^6+2^7+2^8+2^9+2^{10}+2^{12}+2^{13}+2^{15}+2^{17}+2^{19}+2^{21}+2^{24}+2^{25}+2^{28}+(-2^{31}+1)$

即(-321566665)、(-1825916983)和(-182516983)。

情形1）称为“符号-绝对值”模式，情形2）就是我们所熟悉的“2的补码”方式，情形3）是“1的补码”方式。至于具体根据哪一种情况对带符号整数作出解释则由每一种实现自行定义。不过，几乎所有的现代计算机架构均采用“2的补码”解释带符号整数，因此，如果一个实现没有特别说明，我们就可以认为它就是基于“2的补码”方式。

不难得出结论，一个 N 位的带符号整数能够表示的数值范围在三种情形下分别是：

1）$[-(2^{N-1}-1),(2^{N-1}-1)]$

2）$[-2^{N-1},(2^{N-1}-1)]$

3）$[-(2^{N-1}-1),(2^{N-1}-1)]$

值得注意的是，情形1）和情形3）都会产生一个重复的零值：

对于情形1），符号位是“1”其它位全是零的值按照上面的解释也是零；对于情形3），所有位都是“1”的值按照上面的解释也是零。C 标准规定，具体实现可以把这个重复的零值解释为一个特殊的触发值或者“负零（negative zero）”。解释为触发值意味着计

算机保留该值用作表示某种情况的出现(例如数值的溢出)从而引致系统进入陷阱(trap)处理，如果解释为负零，则意味着该系统支持负零的使用，这时，C 标准进一步规定，负零只能由以下途径产生：

- 由&，|，^，~，<<，>>等运算产生
- 由+，-，*，/，%等运算产生（而且其中一个操作数必须是负零、最后产生的结果也是零）
- 由对应的复合赋值运算（即 &=，|=，^=，~=，<<=，>>=，+=，-=，*=，/=，%=）产生

至于产生负零后系统是否自动把负零转换成正常零（normal zero），以及在实际的变量储存中是否使用负零，则由支持负零的平台实现自行决定，而对于不支持负零的平台，在&，|，^，~，<<，>>等运算中产生该数值是 undefined behavior。

很多时候人们容易忘记无符号整数的一个最基本的特性：永不为负，于是在某些场合一不小心就出现愚蠢的错误，其中比较常见的是计数器变量（下面的代码试图从大到小打印[0，99]区间内的所有整数）：

```
unsigned i;
for (i = 99; i >= 0; --i)           /* 看出问题吗？ */
   printf("%u\n", i);
```

上面代码的意图很明显，计数器 i 从99开始，每次减1，递减到0。在这个过程中以 i 为参数调用函数 printf()，总共打印100次，然后退出 for 语句。

但是，计数器 i 是无符号整数，编译器会用相应的指令去判断 i 是否大于或等于零，结果很清楚，作为无符号整数的 i 一定是永远大于或等于零的，就是说，上面的 for 语句是一个死循环。对于无符号整数，尽量避免使用<，>，<=，>=等运算符而优先选用!= ：

```
unsigned i = 100;
while (i != 0)                  /* OK */
{
   --i;
   printf("%u\n", i);
}
```

C89引入了两种新的整数类型：short 和 long，很自然 short 和 long 也是新增加的关键字。现在，C99再增加一种整数类型：long long，至此，C 语言一共有5种标准带符号整数类型以及5种标准无符号整数类型，标准带符号整数类型和标准无符号整数类型合称为标准整数类型[2]：

标准整数类型	
标准带符号整数类型	标准无符号整数类型
signed char	unsigned char
signed short	unsigned short
signed int	unsigned int

sigend long	unsigned long
signed long long	unsigend long long

C99标准并没有硬性规定具体到某种平台上的某种整数类型究竟占用多少字节、能够表示多大范围的数值等，只是给出一条原则和一个参考数值集合，只要同时满足这两方面条件就算是符合 C 标准。原则是：long long 能够表示的数值范围必须大于或等于 long 能够表示的数值范围，long 能够表示的数值范围必须大于或等于 int 能够表示的数值范围，int 能够表示的数值范围必须大于或等于 short 能够表示的数值范围，short 能够表示的数值范围必须大于或等于 signed char 能够表示的数值范围。

每一种标准整数类型的最大、最小值在该平台对应的 C 标准头文件<limits.h>中给出，具体实现的数值应该在绝对值上大于或等于 C99列出的参考值而且符号相同：

宏	参考值
CHAR_BIT	8
SCHAR_MIN	-127 即 $-(2^7-1)$
SCHAR_MAX	+127 即 2^7-1
UCHAR_MAX	+255 即 2^8-1
CHAR_MIN CHAR_MAX	依具体情况而定
SHRT_MIN	-32767 即 $-(2^{15}-1)$
SHRT_MAX	+32767 即 $2^{15}-1$
USHRT_MAX	+65535 即 $2^{16}-1$
INT_MIN	-32767
INT_MAX	+32767
UINT_MAX	+65535
LONG_MIN	-2147483647 即 $-(2^{31}-1)$
LONG_MAX	+2147483647 即 $2^{31}-1$
ULONG_MAX	+4294967295 即 $2^{32}-1$
LLONG_MIN	-9223372036854775807 即 $-(2^{63}-1)$
LLONG_MAX	+9223372036854775807 即 $2^{63}-1$
ULLONG_MAX	+18446744073709551615 即 $2^{64}-1$

※ 上面的 SCHAR_MIN、SHRT_MIN、INT_MIN、LONG_MIN 和 LLONG_MIN 分别表示 signed char、short、int、long 和 long long 的最小值；SCHAR_MAX、SHRT_MAX、INT_MAX、LONG_MAX 和 LLONG_MAX 分别表示 signed char、short、int、long 和 long long 的最大值；UCHAR_MAX、USHRT_MAX、UINT_MAX、ULONG_MAX 和 ULLONG_MAX 分别表示 unsigned char、unsigned short、unsigned int、unsigned long 和 unsigned long long 的最大值，而 CHAR_MIN 和 CHAR_MAX 则表示 char 的最小值和最大值。一部分 C 实现把 char 视为 unsigned char，在这些平台上，以下写法是等价的：

```
char c;
```

unsigned char c;

如果要使用带符号的 char 类型必须明确指示：signed char c;

而另一部分的 C 实现则相反，把 char 看作 signed char，在这些平台上，以下写法是相同的：

char c;

signed char c;

如果要使用无符号 char 类型必须明确指示：unsigned char c;

因此，如果是前者，上面列表中的 CHAR_MIN 等于0、CHAR_MAX 等于 UCHAR_MAX，如果是后者，则 CHAR_MIN 等于 SCHAR_MIN、CHAR_MAX 等于 SCHAR_MAX。 ※

32位计算机架构上的 C 语言实现通常选用 ILP32模式[3]：

整数类型	长度
char	1 byte (8 bit)
short	2 byte (16 bit)
int	4 byte (32 bit)
long	4 byte (32 bit)
long long	8 byte (64 bit)

例如，P6/GNU/Linux 的<limits.h>的对应内容如下[4]：

```
#define  CHAR_BIT     8
#define  SCHAR_MIN    (-128)
#define  SCHAR_MAX    127
#define  UCHAR_MAX    255
#ifdef  __CHAR_UNSIGNED__
   #define  CHAR_MIN    0
   #define  CHAR_MAX    UCHAR_MAX
#else
   #define  CHAR_MIN    SCHAR_MIN
   #define  CHAR_MAX    SCHAR_MAX
#endif
#define  SHRT_MIN     (-32768)
#define  SHRT_MAX     32767
#define  USHRT_MAX    65535
#define  INT_MIN      (-2147483648)
#define  INT_MAX      2147483647
#define  UINT_MAX     4294967295U
#define  LONG_MIN     (-2147483648L)
#define  LONG_MAX     2147483647L
#define  ULONG_MAX    4294967295UL
#define  LLONG_MIN    (-9223372036854775808LL)
```

```
#define  LLONG_MAX    9223372036854775807LL
#define  ULLONG_MAX   18446744073709551615ULL
```

而64位计算机架构上的C语言实现通常选用LP64模式[5]：

整数类型	长度
char	1 byte (8 bit)
short	2 byte (16 bit)
int	4 byte (32 bit)
long	8 byte (64 bit)
long long	8 byte (64 bit)

这里有另外一份<limits.h>，选自一台Sun U30工作站的Solaris 9操作系统，相关内容如下[6]：

```
#define  CHAR_BIT     8
#define  SCHAR_MIN    (-128)
#define  SCHAR_MAX    127
#define  UCHAR_MAX    255
#ifdef  __CHAR_UNSIGNED__
   #define  CHAR_MIN    0
   #define  CHAR_MAX    UCHAR_MAX
#else
   #define  CHAR_MIN    SCHAR_MIN
   #define  CHAR_MAX    SCHAR_MAX
#endif
#define  SHRT_MIN     (-32768)
#define  SHRT_MAX     32767
#define  USHRT_MAX    65535
#define  INT_MIN      (-2147483648)
#define  INT_MAX      2147483647
#define  UINT_MAX     4294967295U
#define  LONG_MIN     (-9223372036854775808L)
#define  LONG_MAX     9223372036854775807L
#define  ULONG_MAX    18446744073709551615UL
#define  LLONG_MIN    (-9223372036854775808LL)
#define  LLONG_MAX    9223372036854775807LL
#define  ULLONG_MAX   18446744073709551615ULL
```

大家可以自行验证，以上两份<limits.h>的内容都是符合C99标准的。

※ limits.h 刚好也是 GNU/Linux（以及 Solaris 等）系统存放 POSIX 标准要求定义的宏的头文件，所以里面除了 C 标准要求定义的内容之外它还包含了很多系统相关的项目，准确地说，在这些系统上，limits.h 不仅仅是 C 标准库头文件而且还是系统头文件。通常，limits.h 会随着系统一起发行并保持相当长时间的稳定，即使我们的 Unix 系统没有安装编译器，“/usr/include”目录下一般仍然会有一份 limit.h。 ※

即使是同一种整数类型，在不同的平台上也可能有不同的长度（例如 long 在 ILP32 和 LP64中分别是32位和64位），为了方便大家跨平台使用整数类型能够“心中有数”，C99增加了标准头文件<stdint.h>，里面定义了一系列的类型，这些类型在所有符合C99的实现上都具有一致的语义，下面介绍其中最重要的几类。

- 具有准确长度的整数类型：

```
int8_t     int16_t       int32_t       int64_t
uint8_t    uint16_t      uint32_t      uint64_t
```

上面的 int*N*_t 表示长度为 *N* 位（不含填充位）、使用“2的补码”的带符号整数类型，而 uint*N*_t 则表示长度为 *N* 位（不含填充位）的无符号整数类型，这些整数类型对应的最大、最小值分别为：

宏	参考值（精确）	
INT*N*_MIN	$-(2^{N-1})$	
INT*N*_MAX	$2^{N-1}-1$	（*N* = 8，16，32，64）
UINT*N*_MAX	$2^{N}-1$	

上述准确长度的整数类型定义是可选的（optional），如果某种平台不支持标准对这些整数类型的规定就不能定义这些类型。比如，标准要求 int*N*_t 类型必须是使用“2的补码”的整数类型，现在某种平台上使用的整数类型是基于符号-绝对值模式，那么该平台上的 C 实现就不能定义 int*N*_t 类型以及宏 INT*N*_MIN、INT*N*_MAX；又譬如，某种平台的无符号整数类型带有填充位，于是不能满足最大值为$2^{N}-1$的要求，那该平台的 C 实现就不能定义 uint*N*_t 类型以及宏 UINT*N*_MAX。总之，对于准确长度整数类型和表示它们最大、最小值的宏，具体的实现要是进行定义的话必须保证符合 C99标准的规定从而使到每个宏对应的值与 C 标准给出的参考值完全一致。[7]

基于 P6的 GNU/Linux 系统完全满足 C99对准确长度整数类型定义的要求（不带填充位、使用“2的补码”），因此在 ***gcc*** 的<stdint.h>里面有如下定义：

```
typedef  signed char             int8_t
typedef  short                   int16_t
typedef  int                     int32_t
typedef  long long               int64_t
typedef  unsigned char           uint8_t
typedef  unsigned short          uint16_t
typedef  unsigned int            uint32_t
typedef  unsigned long long      uint64_t
```

```
#define  INT8_MIN                (-128)
#define  INT16_MIN               (-32768)
#define  INT32_MIN               (-2147483648)
#define  INT64_MIN               (-9223372036854775808LL)
#define  INT8_MAX                127
#define  INT16_MAX               32767
#define  INT32_MAX               2147483647
#define  INT64_MAX               9223372036854775807LL
#define  UINT8_MAX               255
#define  UINT16_MAX              65535
#define  UINT32_MAX              4294967295U
#define  UINT64_MAX              18446744073709551615ULL
```

- 满足指定长度的最小整数类型：

```
int_least8_t      int_least16_t
uint_least8_t     uint_least16_t
int_least32_t     int_least64_t
uint_least32_t    uint_least64_t
```

int_least*N*_t表示长度大于或等于*N*位的最小带符号整数类型，uint_least*N*_t表示长度大于或等于*N*位的最小无符号整数类型。注意，这里所说的长度应该排除填充位，例如，某种平台上的32位带符号整数有8个填充位，则该整数的长度实际上是24位。上面列出的8个整数类型是C99要求每个实现都必须进行定义的，这8个类型对应的最大、最小值以宏的形式进行定义，不过，这次考虑到不是所有平台都使用“2的补码”，因此C99给出的参考值略有不同，并且要求每个C实现给出的实际值必须在绝对值上大于或等于参考值同时符号相同即可：

宏	参考值	
INT_LEAST*N*_MIN	$-(2^{N-1}-1)$	
INT_LEAST*N*_MAX	$2^{N-1}-1$	（*N* = 8，16，32，64）
UINT_LEAST*N*_MAX	$2^{N}-1$	

以下是P6/GNU/Linux的GCC实现：

```
typedef  signed char              int_least8_t
typedef  short                    int_least16_t
typedef  int                      int_least32_t
typedef  long long                int_least64_t
typedef  unsigned char            uint_least8_t
typedef  unsigned short           uint_least16_t
typedef  unsigned int             uint_least32_t
typedef  unsigned long long       uint_least64_t
```

```
#define  INT_LEAST8_MIN            (-128)
#define  INT_LEAST16_MIN           (-32768)
#define  INT_LEAST32_MIN           (-2147483648)
#define  INT_LEAST64_MIN           (-9223372036854775808LL)
#define  INT_LEAST8_MAX            127
#define  INT_LEAST16_MAX           32767
#define  INT_LEAST32_MAX           2147483647
#define  INT_LEAST64_MAX           9223372036854775807LL
#define  UINT_LEAST8_MAX           255
#define  UINT_LEAST16_MAX          65535
#define  UINT_LEAST32_MAX          4294967295U
#define  UINT_LEAST64_MAX          18446744073709551615ULL
```

- 运算速度最快、满足指定长度的最小整数类型：

```
int_fast8_t       int_fast16_t
uint_fast8_t      uint_fast16_t
int_fast32_t      int_fast64_t
uint_fast32_t     uint_fast64_t
```

int_fast*N*_t表示长度大于或等于*N*位并且在该平台上运算速度最快的带符号整数类型，uint_fast*N*_t 表示长度大于或等于 *N* 位并且在该平台上运算速度最快的无符号整数类型。同样地，填充位不算入长度。

上面列出的8个类型是 C99要求每个实现都必须定义的，C 标准同时给出了对应的最大、最小参考值，实现所定义的宏必须在绝对值上大于或等于参考值并拥有相同的符号：

宏	参考值	
INT_FAST*N*_MIN	$-(2^{N-1}-1)$	
INT_FAST*N*_MAX	$2^{N-1}-1$	（*N* = 8，16，32，64）
UINT_FAST*N*_MAX	$2^{N}-1$	

下面是 P6/GNU/Linux 的 GCC 实现：

```
typedef  signed char              int_fast8_t
typedef  int                      int_fast16_t
typedef  int                      int_fast32_t
typedef  long long                int_fast64_t
typedef  unsigned char            uint_fast8_t
typedef  unsigned int             uint_fast16_t
typedef  unsigned int             uint_fast32_t
typedef  unsigned long long       uint_fast64_t
#define  INT_FAST8_MIN            (-128)
#define  INT_FAST16_MIN           (-2147483648)
#define  INT_FAST32_MIN           (-2147483648)
```

```
#define  INT_FAST64_MIN          (-9223372036854775808LL)
#define  INT_FAST8_MAX           127
#define  INT_FAST16_MAX          2147483647
#define  INT_FAST32_MAX          2147483647
#define  INT_FAST64_MAX          9223372036854775807LL
#define  UINT_FAST8_MAX          255
#define  UINT_FAST16_MAX         4294967295U
#define  UINT_FAST32_MAX         4294967295U
#define  UINT_FAST64_MAX         18446744073709551615ULL
```

- 适合存放 void 指针的整数类型：

```
intptr_t     uintptr_t
```

这两种类型都是可选的，它们分别表示足够用来存放 void 指针的最小带符号、无符号整数类型，C99给出参考值，规定对应的宏定义必须在绝对值上大于或等于参考值而且符号相同：

宏	参考值
INTPTR_MIN	$-(2^{15}-1)$
INTPTR_MAX	$2^{15}-1$
UINTPTR_MAX	$2^{16}-1$

下面是 P6/GNU/Linux 的 GCC 实现：

```
typedef  int               intptr_t
typedef  unsigned int      uintptr_t
#define  INTPTR_MIN        (-2147483648)
#define  INTPTR_MAX        2147483647
#define  UINTPTR_MAX       4294967295U
```

- 系统最大的整数类型：

```
intmax_t     uintmax_t
```

顾名思义，它们分别代表该平台上长度最大的带符号、无符号整数类型，C99要求每个实现都必须定义它们，对应的宏在绝对值上要大于或等于参考值并且符号相同：

宏	参考值
INTMAX_MIN	$-(2^{63}-1)$
INTMAX_MAX	$2^{63}-1$
UINTMAX_MAX	$2^{64}-1$

以下是 P6/GNU/Linux 的 GCC 实现：

```
typedef  long long               intmax_t
typedef  unsigned long long      uintmax_t
```

```
#define  INTMAX_MIN              (-9223372036854775808LL)
#define  INTMAX_MAX              9223372036854775807LL
#define  UINTMAX_MAX             18446744073709551615ULL
```

了解 C 语言的所有这些最重要的整数类型之后，下一个问题就是搞清楚编译器在怎样的情况下会把一个变量或者常数看作什么样的整数类型。在 C 语言刚刚被设计出来的时候，要回答这个问题实在太容易了，因为那时一共只有两种整数类型：char 和 int。[8] 在实际的运算当中，char 总是先被提升为 int，例如：

```
char x, y, z;
z = x + y;
```

上面的运算过程其实是：

```
char x, y, z;
z = (int)x + (int)y;
```

编译器会在寄存器里把原本是 char 类型的 x 和 y 分别提升为 int 类型，然后把两个 int 数值相加，接着再把得出来的 int 数值结果重新转换为 char 类型赋给 z。至于把 x 和 y 提升为 int，在不同平台的做法稍有差别：一部分平台使用带符号扩展（以使用“2的补码”为例）：

```
                 10011011
            11111111 10011011      带符号扩展
```

而另一部分平台则使用无符号扩展：

```
                 10011011
            00000000 10011011      无符号扩展
```

前面已经提到，在函数的参数传递过程中，char 类型的参数同样是先被提升为 int 类型然后再传给函数的。

※ 也许大家心里有点纳闷：为什么 char 要扩展成 int 才能参与运算？为什么不能直接把两个 char 相加？要回答这个问题，得先了解 RISC 处理器的某些特点。对于 RISC 处理器，一般地，所有参与运算的对象都只能是寄存器操作数，例如加法运算就只能是两个寄存器的值相加（即使有第3个操作数也只能是一个不太大的常数），既没有内存操作数与寄存器操作数相加也没有仅仅用两个寄存器的最低8位相加的指令。由于变量存放在内存，因此在任何运算之前 RISC 处理器必须先把变量从内存复制到寄存器，等到运算完毕再把寄存器的值复制回内存。

现在，假设32位的 RISC 处理器有32个通用寄存器，分别编号为 r0，r1，r2…r31，而 x、y、z 是 char 变量，则“z=x+y;”这行代码对应的 RISC 汇编指令一般具有下面的形式：

```
ldsb  x, %r1              // 把 x 复制到 r1并进行带符号扩展
ldsb  y, %r2              // 把 y 复制到 r2并进行带符号扩展
add   %r2, %r1            // r1和 r2相加，结果存放在 r1
stb   %r1, z              // 把 r1的最低8位复制给 z
```

很明显，在 RISC 处理器上要对 char 进行运算，无论如何首先得把 char 变量复制到通用寄存器并进行相应的扩展，并且对每一个 char 变量均是如此，这对于使用 RISC 处理器的程序员只是再普通不过的常识。而 PC 程序员受 IA-32的 CISC 设计思想的影响，一开始很难接受 C 语言的这种安排，认

为它简直就是多此一举。其实，回顾一下历史可以知道，当初 D.M.Ritchie 是在16位的 DEC PDP-11 上首次实现 C 语言的（PDP 系列属于 RISC），C 语言与硬件操作紧密相关的特性对他的设计思路有着重大的影响。很自然地，当时 C 语言的 int 类型占16位，刚好是 PDP-11通用寄存器的长度。因此，char 总是提升为 int 再参与运算与其说是 C 语言的规则不如说是实现平台的特点在语言上的反映。

后来，由于 PC 的普及，到处都是 IA-32机器。对于两个 char 变量相加，基于 CISC 的 IA-32有完全不同的解决方案（下面的 x、y、z 是 char 变量）：

```
movb  x, %al              // 把 x 复制到 AL 寄存器
addb  y, %al              // 把 y 与 AL 相加，结果存放在 AL
movb  %al, z              // 把 AL 的值复制给 z
```

和 RISC 不同，IA-32有长度为8位的寄存器 AL，可以直接把 char 变量复制到 AL 从而省去把 x 扩展成 int 这一步，其次，IA-32允许内存操作数与寄存器操作数相加，仍然放在内存的变量 y 没有复制到寄存器就可以直接和 AL 相加，因此把 y 提升为 int 的步骤也没有必要了。如果强行要求 IA-32 仍然按照老规矩做事，则反而导致代码效率的下降：

```
movsbl  x, %eax           // 把 x 复制到 EAX 并进行带符号扩展
movsbl  y, %edx           // 把 y 复制到 EDX 并进行带符号扩展
addl   %edx, %eax         // EAX 和 EDX 相加，结果存放在 EAX
movb   %al, z             // 把 AL 的值复制给 z
```

可见，IA-32按照老规则生成的代码有4条指令，除了 EAX 外还动用了 EDX 寄存器，显然比前面的版本效率低、占用资源多（前面的代码只有3条指令，只用 AL 寄存器）。 ※

从 C89开始，长度较小的整数类型在运算前必须被提升的必然性有所改变，C89允许具体的 C 语言实现在满足某些条件的前提下不对相关类型进行提升而直接运算。例如：

```
char x, y, z;
z = x + y;
```

“正统的做法”是先将 x 和 y 提升为 int 类型，然后相加，最后把得到的结果截短（truncate）再赋给 z。这种处理既符合 K&R C 的老规则，同时也是 C89认可的正确做法（尤其在 RISC 平台上只能这样处理）。但 C89基于某些平台代码优化的考虑很灵活地留了后着，如果实现平台能够确认(x+y)的情况属于下面两者之一：

1）(x+y)的结果不会导致一个 char 类型的溢出（overflow）

2）(x+y)的结果虽然导致一个 char 类型的溢出，但这个溢出是“无声无息的”并且溢出后的残留数值部分恰好与正统做法得到的结果相同

则该实现可以选择不对 x 和 y 进行提升而直接把 char 类型值相加然后赋给 z。

这里有必要对 C89给出的前提条件进行简单的解释。

1）因为其它 C 实现有可能选择按照正统做法行事，在这种情形下，(x+y)是不可能溢出的：x 和 y 都被提升为 int 类型，两个原本属于 char 范围的数值相加根本不可能导致一个 int 类型溢出。因此，所有选择正统做法先提升再相加的实现都不会遇到溢出。现在，ANSI C 给某些 C 实现一个偷工减料的机会，但这些实现不能惹来麻烦。否则，对同一个表达式，某些平台按照一种做法溢出了，某些平台按照另一种做法却没有溢出，这是 C 语言标准委员会所不能接受的。

2）整数类型的溢出被 C 标准定义为 undefined behavior，前面说过，这意味着

一个不该发生的错误发生了，具体的实现可以忽略它对它视而不见，也可以认为该做点什么譬如关掉计算机电源等等。总之，做什么都可以，C 标准委员会不对这种事情负任何责任。但绝大多数的 C 实现都选择沉默的态度，以 P6/GNU/Linux 的 GCC 实现为例，编译器生成的指令根本没有对整数的溢出存在任何的防范、检测与报警[9]，而且对于使用“2 的补码”的处理器，“溢出后的残留数值恰好与正统做法的结果相同”这个条件很自然地得到满足。

C 标准的这项特别条款可以针对各种比 int 短的整数类型而不仅仅局限于 char，事实上，short 类型占16位，同样地，P6可以直接使用 AX 寄存器对 short 变量进行操作而无须扩展到 EAX、EDX 再运算。

※ 不过，特别条款的使用受到相当严格的条件限制，成功的优化并不是普遍能够实施的，例如：

```
short x, y, z;
int i;
z = (x + y);                                    /* 可以使用特别条款 */
i = (x + y);                                    /* 不能使用特别条款 */
```

上面的代码中，同样是表达式(x+y)，当赋值给 z 时可以不把 x、y 提升为 int，但当赋值给 i 时就必须先提升为 int 再运算。原因很清楚，关键在于(x+y)的赋值对象，如果是 z，即使(x+y)超出 short 的数值范围也不用担心，因为溢出后的残留数值与先提升后运算最后截短所形成的结果相同；如果是 i，那一旦(x+y)超出 short 的数值范围（譬如 x=0x7FFFF、y=0x7FFFF 的情况），则提升和不提升分别带来不同的运算结果，按照 C 标准，这种情况下就不能使用特别条款了。类似的情形还有：

```
i = (x - y) + i;                                /* 不能使用特别条款 */
```

在上面的语句中，由于(x-y)要和 int 类型的 i 相加，因此(x-y)先被提升为 int 再参与运算是肯定要的，问题是计算(x-y)的时候是否一定要把 x、y 先提升为 int 呢？能否先使用特别条款计算(x-y)然后再把计算结果提升为 int 去和 i 相加？答案是否定的。原因和上面讲的一样，**在这种情况下特别条款的使用与否将会导致不同的结果**，所以实现不能使用特别条款去优化代码。 ※

另外，在 C 语言中，单字节字符常数的类型从一开始到现在都是 int（单字节字符常数指'A'、'\n'、'\045'、'\x33'等常量）。本来大家很自然地以为'A'应该是 char 类型的，不是吗？例如：

```
char c = 'A';
```

上面这个表达式的确引起了我们的错觉。'A'明显代表一个单字节字符值，把这个代表字符的值赋给 char 变量，可谓“门当户对”的事啊……然而，'A'虽然代表一个单字节字符值，但同时它的值必须要用与 int 等长的空间来存储，因此它的身份是不折不扣的 int。这也是 char 变量必须提升为 int 再参与运算的规则的另一个反映——反正到头来还是得提升为 int，索性一开始就给它 int 的身份！至于上面那个表达式，它的作用相当于把一个曾经提升为 int 类型的 char 数值还原回 char 而已，完全没有问题。虽然 C89开始确立了一系列的新规则，但“单字节字符常量的类型是 int”这个惯例仍然被保留下来，即使到了 C99也同样如此：

```
C01         /* Code 12-01, file name: 1201.c
C02         #include <stdio.h>
```

```
C03         int main()
C04         {
C05             printf("%d\n",sizeof('A'));
C06             return 0;
C07         }
```

由于 sizeof()是编译期就确定的数值，我们完全可以从汇编代码中看出它的具体值，不过这里还是编译成可执行文件再运行：

```
$gcc  -std=c99  1201.c
$./a.out
4
```

显然，'A'要占用4个字节，在32位平台上这恰好就是一个 int 的长度。

综合上面所说，在 C 语言的早期，要确定一个表达式里的整型变量或者常数的具体类型并不复杂，原本就只有两种整数类型，在运算和传递参数时 char 又总是先被提升为 int，加上整型常数属于 int 类型、单字节字符常量也属于 int 类型——一切都非常清楚。

后来，随着 C 语言的进一步发展，K&R C 引入了无符号整数的概念以及 unsigned 关键字，并增加了 short、long 两种整数类型。这时，C 语言已经拥有以下整数类型：

```
char      unsigned char
short     unsigned short
int       unsigned int
long      unsigned long
```

情况开始变得复杂。在进一步分析 unsigned 引入的问题之前，我们先清算一下关于 char 的旧帐。在 D.M.Ritchie 设计 C 语言的早期，char 变量只是用来存放 ASCII 字符，由于 ASCII 是7位编码体系[10]，用最小的存储单位——字节来存放 char 变量完全足够。虽然 ASCII 字符值只占用一个字节的低7位从而使到一个 char 变量不应该出现负值（PDP-11使用2的补码而 char 变量的最高位总是零），D.M.Ritchie 本人还是把 char 实现为带符号整数类型，允许 char 变量出现负值，也就是说，char 只不过是比 int 短一些的整数类型罢了，尽管当初设计它的目的是存放永不为负的 ASCII 字符值。大家可以猜到，在 char 为带符号整数的实现里，系统通常采用带符号扩展把 char 提升为 int，因为这样才能保证扩展后的数值和原先一样。

不过，其它一些 C 实现平台的做法有所不同。例如，某些平台使用8位的扩展编码体系，为了能够表达字符集里所有的字符，这些平台上的 char 变量连最高位也派上用场了。在这种情形下，很难说服人们把最高位是“1”的 char 值看作负数，于是，在这部分的 C 实现中，char 的存在方式恰好和当初的设计目的不谋而合——char 值永不为负。0xFF 不是“-1”而是255。与此对应的是，这些平台通常采用无符号扩展把 char 提升为 int。

到了 K&R C 引入 unsigned 关键字以后，基于 char 的分歧已经根深蒂固，unsigned char、unsigned short、unsigned int、unsigned long 都属于无符号整数类型，

而 short、int、long 属于带符号整数类型，这些都毫无疑问，偏偏 char 例外。char 属于什么类型？在那些一直把 char 实现为带符号整数的平台上，char 属于带符号类型；而在另外那些使用8位字符编码的实现里，char 名副其实地拒绝负数的概念，很明显这些平台上的 char 相当于 unsigned char。

大家可能会问，在那些坚持 char 永不为负的实现里岂不是少了一种8位带符号整数类型？因为在 char 被实现为带符号整数的平台上可以通过加上 unsigned 关键字来构造非负的 char 变量，但却没有办法让另外那些平台使用永远非负的 char 来存放负数啊。

对，正因为如此，C89才会引入 signed 关键字。如果不了解这段历史，大家肯定会认为 signed 关键字简直就是废物——已经专门有一个 unsigned 被发明出来表示无符号整数的概念，那只要不使用 unsigned 来修饰不就意味着带符号整数吗，何必再来一个 signed 呢？对于 short、int、long 这些类型来说，signed 真的是多余，但对于 char，关键字 signed 是必要的。否则大家就不知道自己的 char 变量在不同的平台会有什么表现，在一个平台上是负数，在另一个平台上又被解释为永远非负，这与 C 的标准化完全相抵。有了 signed，我们就能避开这个陷阱，明确表达自己究竟想怎样：

```
signed char x;          /* 无论在哪里，x 都是带符号整型变量 */
unsigned char y;        /* 无论在哪里，y 都是无符号整型变量 */
```

所以，前面在讲述标准整数类型时，笔者已经在注释里着重强调：属于标准带符号整数类型的是 signed char（而不是 char）！char 既不属于标准带符号整数类型也不属于标准无符号整数类型，它属于历史遗物。[11]

交代完 char 的历史之后，我们再来看看 K&R C 的整数类型转换规则：

1）char、short 仍然先被提升为 int 再参与运算

2）其中一个运算量是 long，则另外一个运算量也被提升为 long

3）其中一个计算量是无符号的类型，则另外一个运算量也被转换为无符号类型

例如：

```
char x;    short y;   long z;    int i;
unsigned char u;
x + y → (int)x + (int)y                                  /* 规则1 */
z - y → z - (long)y                                      /* 规则2 */
i * z → (long)i * z                                      /* 规则2 */
u + z → (unsigned long)u + (unsigned long)z              /* 规则2、3 */
u - y → (unsigned int)u - (unsigned int)((int)y)     /* 规则1、2、3 */
```

总的来说，K&R C 的转换原则是 char、short 提升为 int 或 long，带符号类型转换为无符号类型。这在某些情况下会使负数丢失符号而出现奇怪的结果：

```
unsigned short x = 1;
int y = -1;
(x < y)? f() : g();
```

上面的 x 虽然本来在数值上应该是大于 y 的，但由于 K&R C 的转换规则：

```
(x < y) → ((unsigned int)x < (unsigned int)y)
```

y 转换为 unsigned int 后忽然变成一个很大的值：在“2的补码”里面，“-1”的值（按32位计算）是0xFFFFFFFF，对于无符号整数来说，这是（2^{32}-1），当然比“1”要大！所以上面的算术比较结果为“真”，于是 f()函数会被调用。听起来好象有点不可思议，“-1”比“1”大，但 K&R C 规则的确会导致如此现象。

另外一个特别需要注意的问题是，当系统要把长度较小的带符号整数类型转换为长度较大的无符号整数类型时，正确的做法是把带符号整数类型通过带符号扩展转换成与最终的无符号整数类型长度相同的带符号整数类型，然后再进一步转换为无符号类型。例如，如果要把 short 类型数值转换为 unsigned long 类型数值，则编译器会先用带符号扩展把 short 转换为 long，然后才把 long 再转换为 unsigned long。千万不要以为编译器会直接用无符号扩展把 short 类型数值立刻转换到 unsigned long 类型数值。如果是长度较小的无符号整数类型要转换成长度较大的带符号（或者无符号）整数类型，则一律直接使用无符号扩展即可。这条规则从 K&R C 一直沿用到现在的 C99。

至于整型常数的类型确定，K&R C 的相关规则同样很简单，如果没有加“L”或“l”后缀，则整型常数的类型是 int，如果 int 类型不足以存放常数则常数的类型变为 long，如果有后缀“L”或“l”常数的类型就是 long。例如：

```
1234                        /* 类型为 int */
-3674L                      /* 类型为 long */
0xABCDEF                    /* 类型为 long，假设 int 为16位、long 为32位 */
```

C89对整数类型的提升和常数类型的确定规则进行了相当大的修改，使之更加详细合理。整个类型提升及转换过程被 C 标准称为“寻常算术转换（Usual Arithmetic Convertion）”（以后本书将用简写组合“UAC”指代“寻常算术转换”）：

1）其中一个运算量是 unsigned long，则另外一个运算量也被提升为 unsigned long

2）一个运算量是 long，另外一个运算量是 unsigned int，则要看该实现里 long 是否可以表示所有的 unsigned int 数值，如果可以，类型为 unsigned int 的变量被转换成 long；否则两个变量都被转换为 unsigned long

3）其中一个运算量是 long，则另外一个运算量也被提升为 long

4)其中一个运算量是 unsigned int，则另外一个运算量也被提升为 unsigned int

5）两个变量都提升为 int（某些平台如果满足一定的条件可以不提升而直接运算）

编译器按1）到5）的顺序判断，当遇到适用的规则就直接使用该条规则而不再理会其它规则。例如：

```
short s;    int i;    long x;
unsigned short u;    unsigned int y;     unsigned long z;
z + i → z + (unsigned long)((long)i)      /* 规则1 */
x - y → x - (long)y                    /* 规则2，假设 long 的长度比 int 大 */
x - y → (unsigned long)x - (unsigned long)y)
                                       /* 规则2，假设 long 和 int 的长度相同 */
z % s → z % (unsigned long)((long)s)      /* 规则3 */
y * i → y * (unsigned int)i               /* 规则4 */
```

```
s / u → (int)s / (int)u                    /* 规则5 */
```

C89并没有像 K&R C 那样无条件保留 unsigned，这反映在所有5条 UAC 规则都没有提到如果其中一个变量是无符号则另外一个也必然被转换为无符号类型。而且，我们不难看出 C89的规则是尽量保留变量原来的值而不是保留无符号类型这种性质，以规则2）为例，当 long 和 unsigned int 碰在一起时，编译器会首先判断 long 是否足够存放 unsigned int 的所有数值，如果可以就最好了，把 unsigned int 转为 long，结果自然是 long，这样可以最大限度地保留原来的值，因为 long 变量可能会是负值，如果不分青红皂白一律像 K&R C 那样转为无符号类型则负值会丢失。只有当 long 不能表示所有的 unsigned int 数值时编译器才无可奈何地把两者都转换为 unsigned long。这种情形通常发生在 long 和 int 长度相同的平台上，例如 P6/GNU/Linux。

很明显，C89的规则并不保证一定不会发生数值丢失的情况，从上面的5条规则我们知道，如果无符号整型的长度既大于或等于带符号整型、又大于或等于 int 的长度，则由于这时带符号整型会被转换为无符号整型从而有可能丢失符号导致数值的改变。例如在32位平台上仍然有“-1”比“1”大的怪事：

```
int i = -1;
unsigned u = 1;
(i < u)? f() : g();                /* 表达式返回值是零，函数 g()被调用 */
```

总之，在一个表达式里同时使用带符号和无符号整数是一个不良习惯，我们完全没有必须那样做的理由。

同样地，“短”的带符号类型转换到“长”的无符号类型要经历两个步骤，这个上面已经说过。

C89确定整型常数类型的规则并不复杂：

- 对于十进制表示的整数，如果没有后缀，则常数的类型依次为 int、long、unsigned long；如果有后缀“U”(或“u”)，则常数的类型依次为 unsigned int、unsigned long；如果有后缀“L”(或“l”)，则常数的类型依次为 long 和 unsigned long；
- 对于八进制和十六进制表示的整数，如果没有后缀，则常数的类型依次为 int、unsigned int、long、unsigned long；如果有后缀则情形与十进制常数相同。

上面依次列举的类型以数值范围足够用来表示常数、并且最靠前的那个为准。例如在 P6/GNU/Linux 上的 GCC 实现有：

```
12345678                           /* 类型为 int */
0x00AABB99U                        /* 类型为 unsigned int */
-678L                              /* 类型为 long */
05643UL                            /* 类型 unsigned long */
```

C99为标准整数类型的提升和转换设定了更加详细的规则，虽然使用了一些新术语，但基本思想仍然和 C89是一致的。在 C99中，每一种标准整数类型都有自己的“整数转换级别（integer conversion rank)”，提升和转换规则都要依据这个级别来确定该如何进行。关于“级别”有下面几条规定：

- 任意两种带符号整数类型的 rank 不能相同，即使它们的长度实际上是相等的。例如，在 ILP32模式中，int 和 long 都是32位的带符号整数类型，它们的数值范围其实

是一样的，但尽管如此，int 和 long 的 rank 还是不相同的。

- 类型的长度越大，级别越高。
- long long 的级别比 long 要高，long 又比 int 高，int 比 short 高，short 比 signed char 高。
- 对于同一种标准整数类型，带符号类型和无符号类型的级别相同。例如 unsigned long 和 long 的级别相同。
- char 的级别和 signed char 一致。

有了 rank 的概念，要表述 C99的 UAC 规则就非常容易。首先，对于所有级别低于 int 的整数类型，如果 int 能够表示提升前的数值，则该数值被提升为 int 类型，否则提升为 unsigned int 类型；而所有级别大于或等于 int 的整数类型则无须提升。然后，使用下列规则对运算量进行转换：

1）如果两边的操作数类型相同则不需要任何转换。

2）如果两边都是带符号类型或都是无符号类型，则把低级别类型的操作数转换为高级别的类型。

3）如果一个是带符号的（用 S 表示），另一个是无符号的（用 U 表示），并且 U 的级别高于或等于 S 的级别，则把 S 转换为 U 的类型。

4）如果一个是带符号的（用 S 表示），另一个是无符号的（用 U 表示），并且 S 的类型能够表示 U 的类型的所有数值，则把 U 转换为 S 的类型。

5）如果一个是带符号的（用 S 表示），另一个是无符号的（用 U 表示），并且以上规则都不适用，则把 S 和 U 转换为与 S 的类型级别相同的无符号类型。

虽然 C99的表述复杂一些，但其核心原则和 C89依然相同，只不过 C99多了 long long 和 unsigned long long，如果再用 C89那套表述方法会很罗嗦。现在引入 rank 的概念之后，一切都用 rank 的高低进行判断，所有规则都显得更加清晰明了。

C89的特别条款 C99中仍然有效，只要满足前面提到的前提条件，具体实现可以省去把低级别整型变量提升为 int（或 unsigned int）这一步而直接使用未经提升的数值进行运算。

最后，C99用来确定整型常数的规则并没有很大变化，除了因为增加新类型和新后缀带来的必要补充之外，基本上和 C89一致（但也不是完全相同）：

- 对于十进制表示的整型常数，如果没有后缀，则常数的类型依次为 int、long、long long；带有后缀“U”（或“u”）的类型依次为 unsigned int、unsigned long、unsigned long long；带有后缀“L”（或“l”）的类型依次为 long、long long；带有后缀“UL”（或“ul”、“Ul”、“uL”）的类型依次为 unsigned long、unsigned long long；带有后缀“LL”（或“ll”）的类型为 long long；带有后缀“ULL”（或“uLL”、“Ull”、“ull”）的类型为 unsigned long long。
- 对于八进制和十六进制表示的整型常数，如果没有后缀，则常数的类型依次为 int、unsigned int、long、unsigned long、long long、unsigned long long；带有后缀“U”（或“u”）的类型依次为 unsigned int、unsigned long、unsigned long long；带有后缀“L”（或“l”）的类型依次为 long、unsigned long、long long、unsigned long long；带有后缀“UL”（或“ul”、“Ul”、“uL”）的类型依次为 unsigned long、unsigned long long；带有后缀“LL”（或“ll”）的类型依次为 long long、

unsigned long long；带有后缀“ULL”（或“uLL”、“Ull”、“ull”）的类型为 unsigned long long。

同样地，上面依次列举的类型以数值范围足够用来表示常数、并且最靠前的那个为准。例如在 P6/GNU/Linux 上的 GCC 实现有：

```
12345                          /* 类型为 int */
1234567890987654               /* 类型为 long long */
0x3E876FBAC34UL                /* 类型为 unsigned long long */
```

*　　*　　*　　*　　*　　*

[1201]

C 标准允许实现平台的无符号整数类型带有填充位（padding bit），例如，可以有这么一个32位的无符号整数，它的高8位是填充位，另有用途，真正用来存储数值的只是低24位。带符号整数同样也存在类似的情况。但我们通常都不会碰到这种情况，所以笔者在这里不考虑填充位的问题。

[1202]

注意，属于标准带符号整数类型的是 signed char，不是 char！

[1203]

ILP32指 int（I）、long（L）和指针（P）类型都是32位。

[1204]

笔者进行了某些简化，力图使各种数值一目了然，虽然和真正文件的实际表达有小小不同，但实质上仍然是一样的。有兴趣的朋友可以亲自查看/usr/include/limits.h。

[1205]

LP64指 long（L）和指针（P）类型都是64位。

[1206]

笔者同样进行了简化。

[1207]

如果平台的确支持符合 C99要求的这些整数类型，那么标准要求该平台的 C 实现必须定义这些标准长度整数类型。所以，这样概括可能更省事：符合要求的一定要定义，不符合要求的不能去定义。

[1208]

请参阅[Ritchie 1974].

[1209]

C 实现一般不会在整数溢出这种问题上浪费时间。

[1210]

请参阅[ANSI 1986]

[1211]

本书的很多地方仍然使用 char 而没有使用标准的 signed char，原因有二：

1）P6/GNU/Linux 上的 ***gcc*** 默认情况下认为 char 就是 signed char，所以在不会造成理解困难的地方笔者会偷懒少写一个 signed；

2）正文中某些地方讲的情况恰恰就是 signed 关键字出现之前的历史发展状况，那肯定更不能使用 signed 了。

13 整数的运算

C 语言涉及的整数运算不多，归纳起来只有下面几类：

- 常规算术运算，例如：加（+）、减（-）、乘（*）、除（/）、取模（%）、求相反数（-）、前/后缀自增（++）、前/后缀自减（--）；
- 关系运算，例如：大于（>）、小于（<）、大于或等于（>=）、小于或等于（<=）、等于（==）、不等于（!=）；
- 位运算，例如：按位与（&）、按位或（|）、按位异或（^）、按位非（~）、左移位（<<）、右移位（>>）；
- 赋值运算，例如简单赋值（=）、复合赋值（+=，-=，*=，/=，%=，<<=，>>=，&=，|=，^=）；
- 逻辑运算，例如：取反（!）、逻辑与（&&）、逻辑或（||）；

首先我们看一下最简单的赋值运算：

```
E1 = E2            （E1必须是左值）
```

这里我们只讨论 E1、E2都是整型变量、整型常数所组成的表达式的情况。假设 E1 的类型是 T1，则简单赋值运算进行的实际操作是：

1）计算 E2的值 V2

2）把 V2转换为类型 T1，转换后的值用(T1)V2表示

3）计算 E1

4）把 E1所代表的对象的值更新为(T1)V2

要注意的是，第3）步的操作完全可以放在最前面，就是说，编译器可以先计算 E1 再计算 E2也可以先计算 E2再计算 E1。例如：

```
int f(void);
int *g(void);
*g() = f();
```

上面的例子中，E1就是(*g())，E2就是 f()。无论是先调用 g()以计算 E1还是先调用 f()以计算 E2都是符合 C 标准的。有的编译器选择先调用 g()，而另一些编译器则会先调用 f()。

不过，对于整数来说，无论 E1、E2看起来多么复杂，实际上根据 V2和 T1的长度分类最后只有三种情况：

1）T1的长度小于 V2的长度

2）T1的长度大于 V2的长度

3）T1的长度等于 V2的长度

对于1)，很明显，要把一个“较宽的”整数值赋给一个“较窄的”的整型变量，完全有可能发生数值损失。例如：[1]

```
short s;                              /* 16位的 s */
int i;                                /* 32位的 i */
s = i  →  s = (short)i                /* 截短 */
```

以 ILP32模式为例，short 是16位（2字节）的，而 int 是32位（4字节）的，上面赋值运算的结果就是把 i 的低16位数值复制给 s，因此，如果 i 的高16位含有足以影

响整个数值的部分则 s 实际上和 i 根本不相等。非常容易举出这样的例子，譬如 i 是0x00010000，那经过截短赋值之后 s 就是0x0000。无论 s、i 是带符号或者无符号整数类型，0x0000与0x00010000根本就不是一回事。

而2）的情形则刚好与1）相反，把“较窄的”整数值赋给“较宽的”整型变量，不会发生数值损失，但数值的解释可能发生变化。扩展操作会因为被扩展数值的类型不同而存在差别，带符号整数类型适用带符号扩展，无符号整数类型适用无符号扩展。例如：

```
short s;  int i;                          /* 带符号整数 */
unsigned short u;  unsigned n;            /* 无符号整数 */
i = s  →  i = (int)s                      /* 带符号扩展 */
i = u  →  i = (unsigned int)u;            /* 无符号扩展 */
n = s  →  n = (int)s                      /* 带符号扩展 */
n = u  →  n = (unsigned int)u             /* 无符号扩展 */
```

上面，无论是赋值给 i，还是赋值给 n，当被扩展对象是 s 的值（带符号类型）时，扩展是带符号的扩展；当被扩展对象是 u 的值（无符号类型）时，扩展就变成无符号的扩展。这里只须注意一种情形：当带符号的 s 赋值给无符号的 n 时，数值的解释产生了改变，例如 s 的值是0xFFFF（即-1），则赋值后 n 的值是0xFFFFFFFF（即2^{32}-1）。

在情况3）中，由于“=”两边的整数类型长度相等，自然不会发生任何截短、扩展操作，无论是带符号整数类型还是无符号整数类型，一律“原版复制”，当然，数值的解释也可能会发生改变，例如：

```
int i;                          /* 带符号整数 */
unsigned u;                     /* 无符号整数 */
i = u;                          /* u 的值原封不动地复制给 i */
u = i;                          /* i 的值原封不动地复制给 u */
```

上面，假设 u 的值是0xFFFFFFFF（2^{32}-1），则经过赋值操作之后 i 的值就是“-1”，虽然从二进制的值来看 u 和 i 完全相同，但 i 是带符号整数类型所以 i 的值被解释为“-1”。尽管数值的解释产生变化，该赋值表达式仍然被编译器接受：

```
unsigned u;
u = -1;                                 /* 有点滑稽但没有错 */
```

上面把“-1”这个 int 值赋给 unsigned int 变量 u，但编译器不会表示异议，“-1”即是0xFFFFFFFF，于是 u 在赋值语句执行后就是0xFFFFFFFF（2^{32}-1）。

简单赋值表达式和其它表达式一样，具有对应的值以及类型。表达式(E1 = E2)本身的类型和值分别就是 T1和(T1)V2。例如：

```
int i = 0;   short s;
s = i + 4;
```

上面的赋值表达式左操作数是 s，s 的类型是 short，值是4，所以整个赋值表达式的类型就是 short，整个赋值表达式的值是4。

由于赋值表达式的值就是左操作数的值，我们可以在同一语句中多次使用“=”：

```
int x, y, z;
z = y = x;                          /* 相当于 z = (y = x); */
```

上面，编译器首先把 x 的值复制给 y，而赋值表达式本身也返回 y 的值，所以同样的

值又被复制到 z。不过，赋值表达式的值不是“左值”，就是说，不能直接对赋值表达式本身进行赋值，例如：

```
int x, y, z;
(z = y) = x;                    /* 错，(z = y)返回的不是左值 */
```

当然，如果仅仅是“使用”赋值表达式的值而不是对赋值表达式进行赋值，那即使赋值表达式出现在“=”的左边也是合法的，例如：

```
int x, *y, *z;
*(z = y) = x;                   /* 没问题 */
```

上面虽然(z = y)出现在“=”的左边，但赋值对象不是(z = y)的返回值本身，而是(*z)，所以完全没有问题。C 语言的设计者已经考虑到赋值表达式的这种特性，因此所有的赋值运算都是“右结合”的，我们在一连串的连续赋值中无须自己加上多余的括号：

```
a = b = c = d = e;              /* 没问题 */
```

搞清楚简单赋值运算的概念之后再来看复合赋值就很容易理解，假设用“op=”代表任意一个合法的复合赋值操作符（譬如“+=”、“-=”等），则复合赋值运算：

```
E1 op= E2
```

相当于：

```
E1 = E1 op (E2)
```

除了一点，那就是在复合赋值中 E1只被计算一次。

如果不使用复合赋值，那么，在下面的赋值过程中：

```
E1 = E1 op (E2)
```

E1应该被计算2次，第一次是在求“=”右边表达式的值的时候，第二次是在求“=”左边表达式值的时候。我们看一个例子：

```
int *f(void);
*f() = *f() + 4;                /* 函数 f()被执行了两次 */
```

这里，求右边的值时调用了函数 f()，求左边的值时又一次地调用了函数 f()，所以函数 f()总共被执行了两次。

整个赋值过程发生的事情如下：

1）调用 f()，返回值存放在寄存器 A

2）把寄存器 A 指向的内存区域的值复制到寄存器 B

3）寄存器 B 的值加上4，结果存放在寄存器 B

4）再次调用 f()，返回值存放在寄存器 A

5）把寄存器 B 的值复制到寄存器 A 指向的内存区域

如果我们使用复合赋值：

```
int *f(void);
*f() += 4;                      /* 函数 f()只被执行一次 */
```

则按照 C 标准的规定，函数 f()仅仅被调用一次。

整个复合赋值过程发生的事情如下：

1）调用 f()，返回值存放在寄存器 A

2）把寄存器 A 指向的内存区域的值复制到寄存器 B

3）寄存器 B 的值加上4，结果存放在寄存器 B

4）把寄存器 B 的值复制到寄存器 A 指向的内存区域

可以看出，惟一不同的只是第3步之后没有再次调用函数 f()。

当然，如果 E1相对很简单（譬如仅仅由单个变量组成），则上述分别就不存在了：

```
int x;
x += 4;                    /* 与 x = x + 4; 完全等价 */
```

最后，提醒一下，E2两旁那一对括号不是多余的，例如：

```
int x, y;
x *= y - 2;
```

上面的复合赋值中，E2是(y-2)，所以实际上进行的运算是：

```
x = x * (y - 2);           /* 是(y-2)与 x 相乘 */
```

而不是：

```
x = x * y - 2;             /* 这样就不符合“*=”的语义了 */
```

自然地，和简单赋值一样，复合赋值表达式本身也有值和类型，也就是“op=”左操作数的值和类型，我们同样可以连写：

```
int x, y, z;
z += y -= x *= 4;          /* 没问题 */
```

接下来我们逐个讨论整数最基本的算术运算，首先是加（+）和减（-）。

对于采用“2的补码”的系统，加/减运算无须区分操作数是带符号的或是无符号的，一律使用简单的二进制加/减法即可得到对应的正确结果，当然，前提是没有溢出：

```
int x, y, z;
int main(){
    x = -10;
    y = 20;
    z = x + y;
}
```

对应的汇编代码

```
movl  $-10, x
movl  $20, y
movl  y, %eax
addl  x, %eax
movl  %eax, z
```

上面带符号整型变量 x 的值是0xFFFFFFF6，y 的值是0x00000014，z 等于 x 与 y 的和：

```
          11111111 11111111 11111111 11110110      x
+         00000000 00000000 00000000 00010100      y
      1   00000000 00000000 00000000 00001010      z
```

处理器对 x 和 y 执行二进制加法（通过 ADD 指令），我们可以看到，（舍弃进位后）得出的32位值是10，这结果无疑是正确的。前面的章节已经提到，*N* 位带符号整型能够表示$[-2^{N-1}, 2^{N-1}-1]$之间的所有整数，容易证明，只要(x+y)落在该区间范围内，二进制加法完全可以保证结果的准确性。

但如果(x+y)的值超出上述对应范围，也就是说产生了溢出，则结果总是不正确的，例如：

```
int x, y, z;
int main(){
    x = 0x7FFFFFFF;
    y = 1;
    z = x + y;
}
```

对应的汇编代码

```
movl  $2147483647, x
movl  $1, y
movl  y, %eax
addl  x, %eax
movl  %eax, z
```

上面 x 的值是32位带符号整数的最大值，虽然 y 很小，但足以令(x+y)溢出，从而处理器返回不正确的结果：

```
            01111111 11111111 11111111 11111111      x
+           00000000 00000000 00000000 00000001      y
------------------------------------------------
            10000000 00000000 00000000 00000000      z
```

对于带符号整型变量 z 来说，0x80000000是(-2147483648)，两个正整数相加得出一个负整数结果，这明显是错误的。原因很简单，(2147483647+1）已经超出32位带符号整数的数值表达范围，二进制加法算出的结果不再是正确值。

要注意区分“进位（Carry)”和“溢出（Overflow)”。正如前面例子所表明的那样，运算后丢弃“进位”值并不意味着结果不正确。事实是，在使用“2的补码”的系统中，对于任意两个 *N* 位带符号整数，只要它们的和同样落在 *N* 位带符号整数所能够表达的数值范围内，舍弃二进制加法产生的“进位”值（如果有的话）所得到的结果恰恰总是正确的。

※ 各种 CPU 对整数加/减运算的“溢出”有不同的处理。P6处理器的 EFLAGS 寄存器中有对应的标志位（用 OF 表示），每次执行完整数运算指令后，处理器都会根据实际情况对 OF 置“1”或清零。所有的整数运算指令都不会直接引发“整数溢出异常”（4号异常），程序员既可以忽略所有的溢出，也可以对溢出采取行动，譬如在紧接上一条整数运算指令的后面马上发出“INTO”指令，这样处理器就会立刻根据 OF 是否置“1”来判断是否引发4号异常。

MIPS32/64处理器没有“溢出”标志，但它们对应同一种整数运算的指令都有两种版本，譬如加法就有 ADD 和 ADDU 指令。如果使用 ADD 指令，则 CPU 会根据计算结果是否溢出来判断是否直接引发“整数溢出异常”；如果使用 ADDU 指令，则 CPU 会忽略任何的溢出从而不会引发异常。

UltraSPARC 处理器的情况和 P6类似，UltraSPARC 有两种版本的整数运算指令，仍然以加法为例，ADD 和 ADDcc 指令都可以完成加法运算，区别是 ADD 指令不影响 CCR 寄存器的标志位，而 ADDcc 指令则会根据实际的计算情况对 CCR 的各个标志位（包括溢出标志位 CCR.xcc.v）置“1”或清零。ADD 和 ADDcc 都不会使处理器引发异常，程序员可以使用 TVS 指令让处理器根据 CCR.xcc.v 是否置“1”来决定是否引发相应的异常。

C 标准把整数的溢出列为 undefined behavior，这意味着 C 实现可以对溢出采取任何处理。不过，事实上各种 C 实现都果断地忽略所有可能的整数溢出。由前面的汇编代码大家已经知道，C 编译器从来没有在整数运算指令后面再发出 INTO 指令，而在 MIPS32/64平台上，C 编译器从来都是使用不会引发溢出异常的“U”版本指令（譬如 ADDU、SUBU 等）进行整数加/减运算。当然，如果程序员执意要引发溢出异常，那他还是完全可以做到的，使用适当的汇编代码发出对应的指令即可。在 P6/GNU/Linux

上，INTO 引发的4号异常的处理过程非常简单，它仅仅是向引发异常的进程发送 SIGSEGV 信号，进程只须捕获该信号就可以对溢出作进一步的处理，如果进程没有捕获 SIGSEGV 信号则系统马上终止该进程并在系统标准输出终端打印出错信息“Segmentation fault”。 ※

带符号整数运算后存在溢出的可能，那么，无符号整数又怎样呢？无符号整数会“溢出”吗？答案很清楚，不会。我们都知道，一个 *N* 位无符号整数类型的取值范围是[0, 2^N-1]，C 标准进一步规定：如果两个无符号整数值进行运算后得到的结果值（R）超过该结果所属无符号类型（T）能够表达的最大值（M）则应该把(R mod (M+1))作为最终的结果值（类型仍然是 T）。

例如：

```
unsigned x, y, z;
int main(){
    x = 0xFFFFFFFF;
    y = 0xF000000F;
    z = x + y;
}
```

对应的汇编代码

```
movl  $-1, x
movl  $-268435441, y
movl  y, %eax
addl  x, %eax
movl  %eax, z
```

上面的 x、y 都是 unsigned int 类型，(x+y)的类型也是 unsigned int，并且其数值(0x1F000000E)本来已经超过 unsigned int 的最大值0xFFFFFFFF，但根据 C 标准，(x+y)最终的正确结果值应该是：

(0xFFFFFFFF + 0xF000000F) mod 2^{32}

即：0xF000000E(4026531854)

就是说，按照 C 标准的规定，无符号整数永远不会溢出。这是很自然的事，C 语言引入无符号整数的目的本来就是为了表达非负整数对2^N取模的值，准确地说，无符号整数表示的是取模运算的结果而不是通常意义下的准确值。

在使用“2的补码”的系统上，实现 C 语言对无符号整数运算的规定实在太简单了，仍以加法运算为例，仅仅使用简单的二进制加法即可：

```
            11111111 11111111 11111111 11111111     x
+           11110000 00000000 00000000 00001111     y
-------------------------------------------------------
        1   11110000 00000000 00000000 00001110     z
```

在这里，丢弃进位值之后得到的32位值就是 C 标准要求的正确值。正是由于二进制加法能够同时适用于带符号整数和无符号整数的加法运算，因此上面编译器生成的汇编代码都是相同的，均使用 ADD 指令。事实上，P6处理器的 ADD 指令执行的就是简单的二进制加法操作，完全没有区分操作数究竟是带符号还是无符号的。处理器在 ADD 指令执行完毕后循例会设置 OF 标志位，它会假设相加的两个操作数都是带符号整数，然后根据运算结果决定是否对 OF 置“1”。所以，如果我们实际上是使用无符号整数进行运算的话，OF 是否置“1”根本就没有任何意义。

在加法运算中，处理器不区分带符号整数和无符号整数，归根到底都是执行 ADD 指令而已，但这并不意味着我们可以毫无顾忌地在一个表达式里面同时使用带符号和无符号整数。

假设 x 是 int，y 是 unsigned int，看看下面两种情形：

1）x 的值是0x80000000，y 的值是0x00000001，(x+y)就等于0x80000001

```
              10000000 00000000 00000000 00000000        x
 +            00000000 00000000 00000000 00000001        y
-------------------------------------------------------------
              10000000 00000000 00000000 00000001    (x+y)
```

x 是带符号类型，0x80000000意味着(-2^{31})，而 y 是无符号类型，0x00000001很显然就是1，按道理，应该有：

$-2^{31} + 1 = -2147483647$

如果我们把结果赋给 int 变量，那么该结果值将会得到正确的解释，因为对于带符号整数来说，0x80000001的确就是(-2147483647)；如果我们把结果赋给 unsigned int 变量，则随后对该值的解释是不正确的，因为对于无符号整数来说，0x80000001意味着2147483649。可见，在本例中，结果值应该被解释为带符号整数才符合实际情况。

2）x 的值是0x7FFFFFFF，y 的值是0x00000001，(x+y)就等于0x80000000

```
              01111111 11111111 11111111 11111111        x
 +            00000000 00000000 00000000 00000001        y
-------------------------------------------------------------
              10000000 00000000 00000000 00000000    (x+y)
```

x 是带符号类型，0x7FFFFFFF 被解释为$(2^{31}-1)$，而 y 是无符号类型，0x00000001就是1，我们期望的结果是：

$(2^{31} - 1) + 1 = 2^{31} = 2147483648$

如果我们把结果赋给 int 变量，那么随后对该值的解释就是不正确的，因为对于带符号整数，0x80000000意味着(-2147483648)；如果把结果赋给 unsigned int 变量则没有问题，因为对于无符号整数来说，0x80000000刚好就是2147483648。因此，在本例中，必须把结果值解释为无符号整数才符合逻辑。

上面的两个例子说明了混合使用带符号整数和无符号整数的恶劣后果，我们一会儿要把结果解释为带符号整数，一会儿又要把结果解释为无符号整数，情况相当混乱，有些时候我们很难准确预测究竟在本次运算当中该把结果解释为带符号的还是无符号的，即使可以预测到，代码中交替出现的带符号和无符号变量也会令人莫名其妙。所以，除了极少数场合（譬如需要用到对2^N取模的值），我们应该坚持仅仅使用带符号整数，即使真的要用到无符号整数，也千万不要让它和带符号整数同时出现在同一个表达式里面。

※ 上面仅仅讨论了带符号整数的溢出，而没有讨论到二进制加法的“进位”问题。在所有的例子里面，进位值都被丢弃，这可能令大家误会进位值是多余的。其实，进位值还是很有用的。譬如，P6处理器的

EFLAGS 寄存器就有 CF 标志位，每次算术运算后处理器都会根据有否进位而置“1”或清零。这样的话，通过和其它指令（ADC、SBB）配合就能实现更“长”的整数运算。仍然以加法为例，如果我们要把两个 long long 数值相加得出一个 long long 的结果，则可以先把两个64位整数的低32位值相加，结果放在 eax，ADD 指令会正确设置 CF 位，然后使用 ADC 指令把两个64位整数的高32位值以及 CF 的值相加，结果放在 edx。这样，edx:eax 就是最终的64位结果值。 ※

带符号和无符号整数的减法运算同样很简单，原理和加法运算一样，处理器执行简单的二进制减法操作，只不过再没有“进位”的概念，取而代之的是“借位”，在带符号整数的减法中，丢弃借位值恰恰返回正确的结果，当然，前提仍然是没有溢出。P6处理器照样会在减法运算之后设置 EFLAGS 的 CF 和 OF 位，我们同样不应该混合使用带符号和无符号整数。总之，一切都和前面讨论过的加法运算异曲同工，减法和加法本来就是同一回事。

熟悉加法和减法之后，我们就很容易理解求相反数（-）、自增（++）和自减（--）运算。对于 *N* 位整数 x，求相反数运算：

```
(-x)
```

均与下面的减法运算等价：

```
(0 - x)
```

当 x 是8/16/32位整数类型时，利用 P6处理器的 NEG 指令可以直接得到 x 的相反数，事实上 NEG 指令仅仅是执行简单的二进制减法，因此，NEG 指令的 CF 和 OF 设置情况与计算(0-x)时完全相同。根据“2的补码”中求相反数的方法，利用 NEG 和 ADC 指令的组合我们可以在 P6处理器中求出64位整数的相反数。

※ 由于求相反数运算完全可以用减法运算代替，很多处理器例如 MIPS32/64和 UltraSPARC 都没有类似 P6的 NEG 那样直接求整数相反数的指令，在这些平台上编译器实际上通过(0-x)来实现(-x)。 ※

这里必须要注意一些问题：

▪ 对 *N* 位带符号整数类型求 (-2^{N-1}) 的相反数。

由于(0-(-2^{N-1}))是2^{N-1}，所以运算的结果就是2^{N-1}，但2^{N-1}已经超出 *N* 位带符号整数的数值范围：

```
int x = 0x80000000, y;
y = -x;                                 /* y 得到的值是错误的 */
```

不过，对于长度小于 int 的带符号整数类型，如果用长度大于或等于(2*N*)的带符号整数来保存(-2^{N-1})的相反数则可以避免溢出：

```
short x = 0x8000;
int y;
y = -x;                                 /* y 得到的值是正确的 */
```

上面由于 x 先要被提升为 int，所以(-x)运算得出的结果是0x00008000，这个值是正确的。

▪ 对无符号整数求相反数

通常来说，对无符号整数求相反数这种运算是没有意义的。譬如，对于32位无符号

整数 x，它的值是0xFFFFFFFF，现在求相反数(-x)。按照数学逻辑，(-x)应该等于$(1-2^N)$，但实际上，处理器对(0-x)的运算结果是0x00000001，无论把它赋给无符号整数类型还是带符号整数类型都得不到合理的解释。

现在，我们继续讨论整数的乘（*）、除（/）和取模（%）运算。

焦点仍然集中在带符号整数与无符号整数是否存在差异这个问题上。以乘法为例，处理器对两种类型的整数的乘法运算操作是否相同呢？在上面的加/减运算中，处理器一律进行简单的二进制加/减运算就能得到正确的结果，乘法是否同样如此？

*N*位带符号整数类型的数值范围是$[-2^{N-1}, 2^{N-1}-1]$，如果 x、y 都是 *N* 位带符号整数，则(x*y)的数值范围就是$[(2^{N-1}-2^{2N-2}), 2^{2N-2}]$，这已经远远超出 *N* 位带符号整数类型的表达范围，但仍然落在(2*N*)位带符号整数的数值范围（$[-2^{2N-1}, 2^{2N-1}-1]$）内，因此，要准确地表示两个 *N* 位带符号整数的乘积需要使用(2*N*)位带符号整数类型。

类似地，*N*位无符号整数类型的数值范围是$[0, 2^N-1]$，如果 x、y 都是 *N* 位无符号整数，则(x*y)的数值范围就是$[0, (2^{2N}-2^{N+1}+1)]$，这同样已经远远超出 *N* 位无符号整数类型的表达范围，但仍然落在(2*N*)位无符号整数的数值范围（$[0, 2^{2N}-1]$）内，所以，要准确地表示两个 *N* 位无符号整数的乘积需要使用(2*N*)位无符号整数类型。

由于乘法的运算结果比较大，因此这里不用32位操作数作为例子，不失一般性，我们以两个4位整数相乘进行讲解。当然，C 语言里面是没有4位整数的，但这里由4位整数总结得出的规律同样适用于所有其它长度的整数类型。不难推断出，4位带符号整数类型的数值范围是[-8, 7]，4位无符号整数类型的数值范围是[0, 15]。

现在，我们首先看看无符号整数的乘法运算，因为它比较简单。假设 x、y 都是4位无符号整数，x 的值是$(0110)_2$，y 的值是$(1001)_2$，那么理论上(x*y)应该是54。整个运算过程如下：

```
                0110        x
*               1001        y
-----------------------------
            00000110
            00110000
-----------------------------
            00110110    (x*y)
```

为了观察方便，这里在每一步的运算中都直接把该次乘法的值扩展到8位二进制数值（反正最后的结果还是要用到8位无符号整数才能正确表示）。(x*y)的值是$(00110110)_2$，即54，确实和理论值相符。容易证明，*N* 位无符号二进制乘法就是简单的二进制乘法，所有二进制位的运算都遵循“1和0相乘得0、1和1相乘得1”的法则，最后把逐次相乘的值相加（采用简单二进制加法）就能够得出正确值。当然，运算过程的临时值以及最终的结果值的长度都是2*N*。

再接着看带符号整数的乘法，假设 x、y 都是4位带符号整数，x 的值是$(0110)_2$，y 的值是$(1001)_2$，那么理论上(x*y)应该是(-42)：

```
                0110        x
*               1001        y
-----------------------------
            00000110
            11010000
-----------------------------
            11010110    (x*y)
```

上面是处理器进行带符号二进制乘法的运算过程，(x*y)是$(11010110)_2$，即(-42)。很明显，带符号二进制乘法不是简单的二进值乘法，这里 y 的最高位是“1”，这个“1”与 x 的各个位相乘并不遵循“1和0相乘得0、1和1相乘得1”法则，原因很简单，在 *N* 位带符号整数里面，最高位的权值是(-2^{N-1})而不是2^{N-1}，所以，在带符号二进制乘法里，最高位与其它位相乘时，处理器可以先用简单二进制乘法得到的临时值，再取这个临时值的相反数。例如在刚才的例子里，y 的最高位是“1”，这个“1”与 x 即$(0110)_2$相乘时，可以先按照简单二进制乘法进行计算，得到的临时值是$(00110000)_2$，然后取相反数得到$(11010000)_2$，这就是 y 的最高位与 x 相乘的正确值，最后把逐次乘得的值相加（同样采用简单二进制加法）即可算出正确结果。容易看出，两个 *N* 位带符号整数相乘，所有中间值及结果都必须采用2*N* 位的带符号整数类型进行存储才能保证最后结果的准确。

P6处理器提供了带符号整数乘法指令（IMUL）和无符号整数乘法指令（MUL），我们只须向处理器指出参与乘法运算的两个操作数，然后就可以通过乘法指令直接得到最后结果。自然地，带符号乘法指令会把两个操作数都看作是带符号整数，无符号乘法指令会把两个操作数都看作是无符号整数。前面已经提到，两个32位整数相乘得到的值必须要用64位整数类型来存放才能保证结果的正确，因此，严格来说，我们必须动用 P6处理器的两个寄存器（edx：eax）来保存运算结果，假设两个32位整数（地址分别用 A、B 代表）相乘，则对应64位保存方式的乘法指令的用法如下：

```
movl  (A), %eax          或          movl  (A), %eax
imull (B)                            mull  (B)
```

当 IMUL 或 MUL 指令只有一个32位的操作数时，处理器就假定另一个32位的操作数存放在 eax 寄存器，于是它把两者相乘得出结果，然后把结果存放在 edx 和 eax 里，edx 保存64位值的高32位，eax 保存低32位。

※ 某些32位处理器也通过类似的方法保存64位乘积，例如 MIPS32处理器的 MULT（带符号乘法）和 MULTU（无符号乘法）指令就是使用两个专门的寄存器（HI 和 LO）来存放两个32位整数相乘的结果值，HI 寄存器保存64位值的高32位，LO 寄存器保存低32位。 ※

但是，麻烦还是出现了。还记得 C 语言整数运算结果的类型是如何确定吗？两个 int 的算术运算结果也是 int！换句话说，对于 C 语言来说，两个32位整数的乘积仍然是32位整数。正如前面一直强调的那样，如果使用32位整数类型存放两个32位整数的乘积完全有可能出现错误的结果，不过，遗憾的是，我们无法在 C 语言的领域内解决这个问题，因为这是 C 语言的固有规则所设下的限制。不要以为像这样就绕过问题了：

```
int x, y;
long long z = x * y;                    /* 这样不能解决问题 */
```

上面的 z 虽然是 long long 类型，长度的确为64位，足够存放(x*y)的值。但可惜的是，编译器一早就断定(x*y)的类型是 int，所以，赋值表达式的作用仅仅是把(x*y)这个32位整数（既乘积的低32位值）带符号扩展为64位整数。下面会说到，编译器会使用对应的乘法指令，这些指令只返回64位乘积的低32位值。

※ 如果我们想完整地保存64位乘积，必须借助汇编语言，通过在C代码中嵌入汇编指令突破C语言的限制：

```
int x, y;  long long z;
__asm__ ( " imull  %1 " : "=A"(z) : "m"(x), "a"(y) );                    ※
```

对应C语言的规则，编译器固然可以使用只有一个32位操作数的乘法指令，然后仅仅把eax寄存器的值看成是32位整数相乘的结果，就是说，对于下面的C代码：

```
extern int x, y, z;
z = x * y;
```

编译器可以这样实现：

```
movl  x, %eax
imull y
movl  %eax, z                    /* 丢弃用edx存放的高32位值 */
```

但这样处理器还是动用了edx寄存器，其实edx的值根本没有派上用场。所以，在C语言的实现里，编译器通常使用的是带有两个操作数版本的乘法指令：

```
movl  x, %eax
imull y, %eax
movl  %eax, z
```

上面IMUL指令进行的操作是把y与eax的值相乘然后把乘积的低32位存放在eax里。于是，节省了资源（没有用到edx）却得到同样的效果。

虽然在C语言里面两个32位整数的乘积仍然是32位整数，但对于其它长度比int（或unsigned int）小的整数类型来说，数值损失还是可以避免的。例如：

```
short x, y;
int z;
z = x * y;                       /* 这次z得到的乘积是准确的 */
```

根据C语言的类型确定规则，编译器会先把x、y都提升为int（32位值），然后计算(x*y)，最后把得出的32位值赋给z。x、y原先是short，长度是16位，前面已经证明，两个16位整数的乘积可以用一个32位整数类型来准确表示，尽管在32位整数乘法运算过程中处理器基于两个操作数版本的乘法指令丢弃了64位结果值的高32位，但这对真实的结果值没有任何影响，64位结果的低32位值已经能够正确地表示两个short值的乘积，被处理器丢掉的高32位全部与低32位部分的最高位相同，要么全是零，要么全是“1”。所以这一次z确实得到了(x*y)的正确值。

同理，两个signed char、两个unsigned char相乘、两个unsigned short相乘、signed char与short相乘以及unsigned char与unsigned short相乘都可以通过把乘积赋给int、long甚至long long来避免数值错误，当然，带符号整数相乘的结果应该赋给带符号整数类型，无符号整数相乘的结果应该赋给无符号整数类型。至于两个int相乘、两个unsigned int相乘、两个long相乘、两个unsigned long相乘、两个long long相乘以及两个unsigned long long相乘都有可能出现错误的结果。对于32位处理器，当任何一个乘数的长度大于或等于32时，错误就有可能发生；对于64位处理器，当任何一个乘数的长度大于或等于64时，错误就有可能出现。

※ MIPS64处理器通过两个64位寄存器（HI 和 LO）来存放128位的乘积从而给64位整数相乘提供了最大限度的支持。基于 x86-64架构的 Athlon 64系列处理器同样支持64位整数乘法，128位的乘积保存在 rdx（高64位）和 rax（低64位）中。 ※

从上面的讨论我们已经很清楚这么一个事实，带符号整数乘法和无符号整数乘法不是同一回事。在加法和减法运算中，处理器对带符号整数与无符号整数统一进行简单的二进制加法和减法，只要带符号整数的运算结果没有溢出则结果总是正确的，而在乘法中，处理器却要求明确给出操作数的类型，带符号整数使用带符号整数乘法，无符号整数使用无符号整数乘法，相同的二进制值用两种乘法运算得出的结果截然不同。所以，在 C 语言里，我们必须清楚自己代码的乘法运算究竟是带符号的还是无符号的，而这个归根到底还是由前面论述的整数类型确定规则所决定。编译器会先把参与乘法运算的两个整数值进行对应的提升或者转换，提升或者转换的结果是两个操作数拥有相同的类型，如果这个类型是带符号的编译器就使用带符号乘法指令，如果这个类型是无符号的编译器就使用无符号乘法指令。

例如：

```
short x;  int y;  unsigned long z;
x * y  →  (int)x * y                                  /* 带符号乘法 */
x * z  →  (unsigned long)((long)x) * z                /* 无符号乘法 */
```

现在，我们来分析一个有趣的现象：两个长度为 *N* 的整数相乘，如果以 S 来表示用带符号乘法得出的运算结果、以 U 来表示用无符号乘法得出的运算结果，则 S 和 U 的低 *N* 位是相同的。要给出证明很容易，假设：x 和 y 是 *N* 位整数，x 最高位的值是 a（a 要么是零、要么是“1”），把 x 最高位清零后得到的 *N* 位整数是 A；y 最高位的值是 b（b 要么是零、要么是“1”），把 y 最高位清零后得到的 *N* 位整数是 B。于是，如果把 x、y 看成是 *N* 位带符号整数的话，它们所表达的真实值就应该是：

$$x = a*(-2^{N-1}) + A$$

$$y = b*(-2^{N-1}) + B$$ /* *N* 位带符号整数的最高位的权值是(-2^{N-1}) */

如果把 x、y 看成是 *N* 位无符号整数的话，它们所表达的真实值就是：

$$x' = a*2^{N-1} + A$$

$$y' = b*2^{N-1} + B$$ /* *N* 位无符号整数的最高位的权值是2^{N-1} */

那么：

$$\begin{aligned} & x*y - x'*y' \\ &= [a*(-2^{N-1}) + A]*[b*(-2^{N-1}) + B] - [a*2^{N-1} + A]*[b*2^{N-1} + B] \\ &= (-2^{N})*(a*B + b*A) \end{aligned}$$

可见，无论 a、b、A、B 的取值如何，(x*y - x′*y′)的低 *N* 位全都是零，也就是说，(x*y)与(x′*y′)的低 *N* 位是相同的，(x*y)与(x′*y′)的差异仅仅反映在高 *N* 位上。

本来，两个 *N* 位整数相乘的积必须用(2*N*)位的整数类型来存放，但由于前面分析过的原因，在 C 语言里面事实往往就是我们只能得到乘积的低 *N* 位。而现在又证明了对于

低 *N* 位数值来说，带符号整数乘法与无符号整数乘法没有任何区别，于是，编译器可以偷懒，在绝大多数情况下只使用一种乘法指令，某些处理器甚至只支持一种版本的乘法指令。

※ 对于 UltraSPARC 处理器，用来计算64位整数（包括带符号和无符号）乘法的指令只有一条，那就是 SPARC-V9规范中的 MULX 指令。MULX 计算两个64位整数的乘积并仅仅保留乘积的低64位。

MIPS32/64 处理器除了支持保存 64/128 位乘积的 MULT/DMULT（带符号乘法）指令以及 MULTU/DMULTU（“U”版本，无符号乘法）指令外，还支持只保存乘积的低32/64位的 MUL/DMUL 指令，但 MUL/DMUL 指令没有对应的“U”版本，因为 MUL/DMUL 指令只保留乘积的“低半截”，显然，在这种情形下“U”版本的 MUL/DMUL 指令完全是多余的。 ※

和加法/减法一样，C 实现都不会去判断乘法结果是否溢出。不过，P6处理器仍然会根据每次乘法的运算结果设置 CF 和 OF。两个 *N* 位整数相乘，无论使用什么版本的乘法指令，P6处理器都会先按照指令是带符号乘法还是无符号乘法计算出对应的 (2*N*) 位乘积，然后只须检查这个乘积是否和低 *N* 位的数值相等就可以作出相应的判断：对于带符号乘法运算，处理器会先将乘积的低 *N* 位数值作带符号扩展然后比较两者，如果两者相等，则表示低 *N* 位的数值就是本次乘法的准确结果，处理器会把 CF 和 OF 标志清零；如果两者不相等，则表示仅仅取低 *N* 位的数值并不能得到本次乘法的实际结果，这时处理器会把 CF 和 OF 置“1”。对于无符号乘法运算。如果高 *N* 位全是零，则表示低 *N* 位的数值就是本次乘法的准确结果，处理器会把 CF 和 OF 标志清零；如果高 *N* 位不全是零，那么意味着仅仅取低 *N* 位的数值是不能准确得到本次乘法的实际结果，这时处理器会把 CF 和 OF 置“1”。例如：

```
int x = 2;
int y = 0x80000000;
int z = x * y;                          /* 溢出了 */
```

上面这段代码的运行结果是 z 的值变为零，尽管编译器编译出来的指令没有对溢出作任何检测，但由于 P6处理器会根据是否溢出设置 OF，所以如果我们万一真的要对溢出采取措施的话，仍然可以通过汇编指令去判断，譬如：

```
int x = 2;
int y = 0x80000000;
int z = x * y;
__asm__ ("into");               /* 如果溢出就引发4号异常 */
```

虽然 P6处理器只支持32位整数乘法指令，但 GNU/Linux/GCC 同样实现了 C99的 (unsigned)long long 类型的乘法运算，不过，由于 C 语言的类型规则，两个 (unsigned)long long 整数相乘结果仍然是(unsigned)long long 类型，所以在使用64位整数乘法时同样要注意乘积有可能溢出。

整数的除法（/）和取模运算（%）同样不那么简单。C 标准规定：

1）在除法、取模运算中，如果第2个操作数为零会导致 undefined behavior。

2）整数除法运算的结果是两个整数相除所得出的商，商只取整数部分；取模运算的结果则是两个整数相除所得出的余数，余数的符号与被除数相同。

3）如果(a/b)没有溢出，则((a/b)*b+a%b)应该等于 a。

在 C 语言里，两个 N 位带符号整数相除只有两个情况可能会导致溢出，一是除数为零，另一个就是被除数是 (-2^{N-1}) 而除数是“-1”。前者很容易理解，而后者也非常明显，因为 (-2^{N-1}) 除以“-1”得出的商是 2^{N-1}，这已经超过 N 位带符号整数的最大值 $(2^{N-1}-1)$。

各种处理器对整数除法导致的溢出有不同的处理策略，P6/GNU/Linux 会给进程发送 SIGFPE 信号，指出存在算术运算错误，如果进程不捕获信号进行处理则系统会终止该进程，并在屏幕上打印“Floating point exception”。

先看一个简单的例子：

```
C01          /* Code 13-01, file name: 1301.c
C02          int x = 0x80000000;
C03          int y = -1;
C04          int z;
C05          int main()
C06          {
C07              z = x / y;          /* 除法溢出 */
C08          }
```

编译、运行13-01，马上可以看到：

```
$gcc  1301.c
$./a.out
Floating point exception
```

如果观察13-01中对应除法运算的汇编代码就更清楚了：

```
movl  x, %eax
cltd
idivl  y
movl  %eax, z
```

首先，把被除数 x 放到 eax 寄存器，然后用 cdq 指令[2]作带符号扩展，扩展部分放在 edx，这样的话，edx：eax 就构成一个64位的被除数，然后通过 IDIV 指令进行除法运算，如果没有发生除法溢出，则 IDIV 指令会把商放在 eax，余数放在 edx。

对于两个 N 位带符号整数相除，如果除数为零或者被除数是 (-2^{N-1}) 而除数是“-1”，则 IDIV 除法指令会引发 P6处理器的0号异常。GNU/Linux 对0号异常的处理是给进程发送 SIGFPE 信号，进程可以捕获该信号进行相关处理也可以忽略信号。如果进程忽略 SIGFPE 信号，则系统立刻终止进程；如果进程捕获 SIGFPE 信号，则完成相关处理后 P6处理器会重新执行 IDIV 指令，这是因为0号异常属于“（执行）失败（fault）”类异常，从异常处理返回后处理器仍然要重新执行那条曾经引发异常的指令。

※ P6处理器对除法溢出的处理过于谨慎，连被除数是 (-2^{N-1}) 而除数是“-1”的情况也会导致0号异常。对于 UltraSPARC 处理器，只有除数为零的情况才会引发异常，当被除数是 (-2^{N-1}) 而除数是“-1”时除法指令 SDIVX 会返回 (-2^{N-1}) 作为运算结果。MIPS32/64处理器根本就没有对应除法溢出的异常，即使除数为零处理器也不会引发任何异常（当然，这时除法指令返回的运算结果也是错误的）。由于 C

语言实现会在除数为零的情况下由操作系统发送 SIGFPE 信号，所以 MIPS32/64平台上的 C 实现通常会在除法指令之前加插其它指令来判断除数是否为零，如果除数为零则用 BREAK 指令引发处理器异常从而通知操作系统发送 SIGFPE 信号给进程。 ※

由于被除数是(-2^{N-1})而除数是“-1”的情况会导致 P6处理器进入0号异常，所以带符号整数除法不可能援引 C 标准的特别条款进行代码优化。例如：

```
short x, y, z;
z = x / y;
```

上面，x、y 必须被先提升为 int 然后才能参与除法运算。这是因为在除法运算中，提升后再运算不会引发异常，而不提升就直接运算则会引发异常。假设除法运算前 x 的值是(-32768)、y 的值是(-1)，那么，按照标准做法，x、y 被提升为 int，即0xFFFF8000和0xFFFFFFFF。(x/y)的类型为 int，(x/y)的值等于32768（即0x00008000），显然，这种情形下 IDIV 指令不会引发0号异常。当把(x/y)赋值给 z 时，32位值的低16位被复制到 z，所以 z 的值是0x8000，即(-32768)。

如果编译器错误地援引特别条款进行优化，则 x、y 未经提升就直接用 IDIV 指令相除，跟32位整数除法类似，处理器的 dx：ax 存放被除数 x（x 是16位操作数，带符号扩展到 dx 寄存器），除数是 y（16位操作数），商放在 ax，余数放在 dx。显然，这时 IDIV 指令会引发0号异常。

无符号整数除法必须使用与带符号整数除法不同的指令进行运算，而且对于 *N* 位无符号整数而言，除法溢出只会在除数为零的情况下发生，这是因为商的取值范围同样是[0，2^N-1]，即使除数取最小值“1”，除法结果也不会溢出。和乘法一样，编译器根据除号两边的整数类型来确定使用带符号除法还是无符号除法。

例如：

```
short x;   int y;   unsigned long z;
x / y  →  (int)x / y                              /* 带符号除法 */
x / z  →  (unsigned long)((long)x) / z            /*无符号除法 */
```

无符号除法可以适用 C 标准的特别条款，编译器能够为 P6处理器进行适当的优化，譬如两个 unsigned short 整数相除可以不提升就直接进行无符号除法运算：

```
unsigned short x;
unsigned short y;
unsigned short z;
z = x / y;
```

对应的汇编代码

```
movw   x, %ax
movl   $0, %edx
divw   y
movw   %ax, z
```

和乘法类似，GNU/Linux/GCC 同样支持64位整数除法，这为我们使用 C99的(unsigned)long long 类型提供完善的实现环境。GCC 是通过编译器内置的函数__divdi3()和__udivdi3()来实现64位带符号和无符号整数的除法。例如：

```
long long x, y, z;
int main(){
    z = x / y;
}
```

对应的汇编代码片段为：

```
pushl   y+4
pushl   y
pushl   x+4
pushl   x
call    __divdi3            /* 通过函数计算64位整数除法 */
movl    %eax, z
movl    %edx, z+4           /* 结果值存放在edx：eax */
```

通过函数__divdi3()、__udivdi3()进行64位整数除法计算时，如果除数为零，则__divdi3()和__udivdi3()同样会引发 P6处理器的0号异常；如果被除数为0x8000000000000000而除数为(-1)则函数__divdi3()不会引发异常而是返回0x8000000000000000，这是64位带符号整数除法的特别之处。

取模运算与除法运算关系极其密切，我们求整数的取模结果实际上就是通过除法运算间接进行的，上面已经解释过，P6处理器的8/16/32位除法指令 IDIV（带符号）和 DIV（无符号）会把除法运算的商放在 al/ax/eax 寄存器、对应的余数则放在 dl/dx/edx 寄存器，很明显，dl/dx/edx 就是我们所需要的取模运算结果。

由于带符号整数的取模运算是通过带符号整数除法运算间接获得结果、无符号整数的取模运算则是通过无符号整数除法运算间接获得结果。因此，带符号整数的取模运算不能引用特别条款，而无符号整数的取模运算则可以引用特别条款。这是因为在除法运算中会导致溢出异常的情况同样会在取模运算中引发异常。

对于 P6处理器，下面的代码都会引发0号异常（尽管它们表面上不是在进行除法运算）：

```
int x, y = 0, z;
z = x % y;                          /* 除数为零 */
```

或者：

```
int x = 0x80000000, y = -1, z;
z = x % y;                          /* 除法溢出 */
```

P6/GNU/Linux 的64位带符号、无符号整数取模运算是 ***gcc*** 通过编译器内置函数__moddi3()和__umoddi3()来完成的（前者是带符号取模、后者是无符号取模）。

由于__moddi3()和__umoddi3()同样需要进行64位整数除法运算，因此当取模运算符右边的64位操作数是零时，__moddi3()和__umoddi3()同样会引发0号异常；但当取模运算符左边操作数等于0x8000000000000000、右边操作数等于(-1)时，__moddi3()不会引发异常，这也是64位带符号整数取模运算与其它长度小于64位的带符号整数取模运算的主要区别。

下面继续讨论整数的移位运算。

表达式：E1 << E2 （或 E1 >> E2）表示对 E1所表示的对象进行移位运算，其中E1、E2都必须是整数类型[3]。以左移位为例，移位运算进行的实际操作是：

1）计算 E2的值 V2，对 V2进行类型提升，提升后的值是 P2

2）计算 E1的值 V1，对 V1进行类型提升，得到的值用 P1表示，P1的类型是 T

3）对 P1执行二进制左移位运算，移位次数是 P2，得到的结果值是 R

最后，整个表达式(E1 << E2)的值和类型分别就是 R 和 T。

和其它表达式的计算类似，1）和2）的顺序不固定，编译器可以先计算 E2得到 V2也可以先计算 E1得到 V1。另外，类型提升可以是空操作，什么都不做。例如：

```
int x, y;
x << y;
```

上面由于 x 和 y 已经是 int 类型，所以类型提升实际上没有做任何事。

必须注意的是，整个移位表达式的值是 R，R 是由 P1经过二进制移位得到的，所以 R的类型和 P1一样（都是 T），T 则是 V1进行类型提升后的类型。特别是，与加减乘除等算术运算不同，P1和 P2的类型并不一定相同，因为这里仅仅要求对 P1、P2各自进行类型提升，而没有要求对 P1、P2执行 UAC 操作。

例如：

```
short x;  long long y;
x << y;  →  (int)x << y;
```

上面，表达式(x<<y)的类型是 int 而不是 long long。

由于二进制移位次数是 P2，因此如果 P2是负数或者 P2大于或等于 T 的长度，那么这样的移位操作显然没有意义。C 标准规定，进行这种无意义的移位操作将会导致 undefined behavior。

前面已经说过，P1是经过类型提升得到的值，因此类型 T 的长度肯定大于或等于 int，当 T 是32位的类型（譬如 int、unsigned int）时，P2如果大于或等于32则移位操作是没有意义的；当 T 是64位的类型（譬如 long long、unsigned long long）时，P2如果大于或等于64则移位操作是没有意义的。

P6处理器的 SAL/SAR/SHR 指令分别可以进行算术左移位/算术右移位/逻辑右移位操作，移位次数由 CL 寄存器的值指定。SAL/SAR/SHR 指令会自动把移位次数限制在31以内，当 CL 的值小于32时，SAL/SAR/SHR 根据 CL 的值进行移位；当 CL 的值大于或等于32时，SAL/SAR/SHR 的操作是移位31次。

对于 P6/GNU/Linux 平台，GCC 一律把移位次数放入 CL 寄存器。如果我们在表达式中直接用整型常数指定移位次数，而这个常数又大于或等于32、甚至更离谱是个负数，则

gcc 会发出警告。例如：

```
int x;
x << 32;
...
warning: left shift count >= width of type
```

但 *gcc* 仍然会把移位数值的低8位放入 CL 寄存器然后执行 SAL/SAR/SHR 指令；如

果我们是使用变量等其它方式指定移位次数，那么 ***gcc*** 就无法在编译期作出检查，它只是直接把移位次数的低8位放入 CL 然后执行移位指令。P6处理器在执行 SAL/SAR/SHR 指令时，CL 的数值是作为无符号整数看待的，而另一方面处理器会自动限制移位次数。

※ 其它的32位处理器也有类似设计来防止移位次数大于或等于32。譬如 MIPS32处理器关于移位运算有两种版本的指令：SLL/SRL/SRA 和 SLLV/SRLV/SRAV。前者适用于直接以整型常数指定移位次数的场合，移位次数被记录在指令的二进制编码中（占5个位），后者把某个寄存器的低5位值作为移位次数，因此，MIPS32处理器无论如何都不会进行移位次数大于或等于32的移位操作。 ※

对于左移位操作，各种 C 实现都一样，就是把原来数值的最高位去掉然后在最低位后面补零从而构成一个新的值，这个值就是左移位一次所得到的结果。

例如：

```
                10110101 10101110 00010111 10101101        原来的数值
去掉最高位       0110101 10101110 00010111 10101101   0      补零
                01101011 01011100 00101111 01011010        左移位结果
```

左移位操作在某些情况下可以用作快速的乘法运算，由于移位指令的执行时间比乘法指令少得多，因此，（当其中一个乘数是2^K时）编译器出于优化代码的考虑可能会用移位运算取代乘法运算。显然，每左移位一次相当于把原值乘以2，左移位 K 次相当于原值乘以2^K。对于 N 位无符号整数类型我们可以放心地进行这种优化，因为无符号整数不存在溢出的问题；对于 N 位带符号整数类型，虽然乘法运算结果的低 N 位数值与移位运算结果一致，但两种运算中 P6处理器对 OF 标志位的设置情况有所不同，如果我们要根据 OF 确定是否发生溢出便只能使用乘法指令。当然，前面提到，各种 C 实现都会忽略乘法溢出，因此这也不是什么大问题，优化还是可以进行的。

事实上，***gcc*** 就是这样做的：

```
extern int x;
x *= 4;                         /* 乘法运算 */
```

对应的汇编代码是：

```
movl  x, %eax
sall  $2, %eax                  /* 左移位2次，因为4是2^2 */
movl  %eax, x
```

对于右移位，情况就有点微妙了。C 标准规定，对于无符号整数类型以及带符号整数类型但数值是非负的整数，右移位操作就是把原来数值的最低位去掉然后在最高位前面补零从而构成一个新的值，这个值就是右移位一次所得到的结果。

例如：

```
                00110101 10101110 00010111 10101101      原来的数值
   补零      0  00110101 10101110 00010111 1010110       去掉最低位
                00011010 11010111 00001011 11010110      右移位结果
```

不难看出，这种情形下的右移位操作实际上相当于除数为2的除法运算，移一次就是除以2，移 K 次就意味着除以 2^K。因此，编译器完全可以用右移位操作取代除数为 2^K 的除法运算。

对于带符号整数类型、其数值又是负数的整数，C 标准把右移位表达式的值定义为 implementation-defined，就是说，由各个 C 实现根据自己平台的情况进行选择，当然，选择只有两种：在最高位前面补零和补“1”。

如果是补零，可以想象，负值马上就变为正值，这种结果不太合理；如果是补“1”，那么原先的负值仍然为负值，右移位运算依然和除数为 2^K 的除法运算有联系。C 实现的选择受处理器的限制，如果处理器提供了相关指令，那么该平台上的 C 实现一般都会选择补“1”，毕竟这才是合理的结果。如果处理器没有右移位补“1”的指令，那 C 实现也就只能选择补零了。幸运的是，大多数处理器均提供相应的指令实现负数右移位时补“1”的操作。P6处理器对基本的右移位操作提供两条指令：SAR 和 SHR，SAR 指令会在移位时补上原最高位的值，原先最高位是零就补零、是“1”就补“1”；而 SHR 则一律补零。这样，编译器只要对无符号整数的右移位一律使用 SHR 指令，对带符号整数的右移位则全部使用 SAR 指令就能保证结果是合理的。

这里必须要强调一点，对于带符号整数，如果数值为负，那么右移位运算的结果与除数为 2^K 的除法运算结果并不总是相同的，当被移走的那些位里面含有“1”时，移位运算就会得出与除法不同的数值。

一个 N 位（N 等于32或64）的带符号整数 x，其值为负，现在我们比较一下除以 2^K 和右移 K 位各自得出的结果，其中 K 在(0, N-1]内。

不失一般性，x 总是能够表示成：

$x = -2^{N-1} + a*2^K + A$　　/* A 的数值范围是[0, 2^K-1] */

/* a 的数值范围是[0, 2^{N-1-K}-1] */

对于除数为 2^K 的除法运算：

$x / 2^K = -2^{N-1-K} + a + A*2^{-K}$

根据 a 的数值范围可知，$(-2^{N-1-K} + a)$ 小于或等于 (-1) 并且一定是整数，而 $(A*2^{-K})$ 则大于或等于零但一定是小于“1”的。如果要把除法结果表示为小数形式，那么有：

$(x / 2^K)$ 的整数部分 $= -2^{N-1-K} + a - 1$

$(x / 2^K)$ 的小数部分 $= 1 - (A*2^{-K})$　　/* 当 A 不等于零时 */

以及：

$(x / 2^K)$ 的整数部分 $= -2^{N-1-K} + a$

$(x / 2^K)$ 的小数部分 $= 0$　　/* 当 A 等于零时 */

对于右移 K 位运算，很容易看出，(x >> K) 的值就是 $(-2^{N-1-K} + a)$。

所以，除非 A 等于零，否则除法运算结果，即 $(x/2^K)$ 的整数部分，和右移位运算结果是不相等的，而 A 等于零就意味着被移走的 K 个位全部是零。

看一个实际的例子就更清楚了。

假设 x 等于 (-5)，现在比较一下 (x/4) 和 (x>>2)。

(-5)的二进制补码是：

11111111111111111111111111111011　　即0xFFFFFFFB

(-5)/4的值是(-1)：

11111111111111111111111111111111　　即0xFFFFFFFF

(-5)>>2的值是(-2)：

11111111111111111111111111111110　　即0xFFFFFFFE

对于负数，我们不难发现除法运算与右移位运算的区别所在，(-5)除以4的完整结果是(-1.25)，C语言的整数除法运算是取(-1.25)往零的方向上最近的整数(-1)作为结果，而右移位运算是取往负无穷大方向上最近的整数作为结果，因此，除非被除数能够被整除，否则除法结果总是比右移位结果大“1”。

编译器对带符号整数除以2^K的除法运算不可以仅仅用一条 SAR 移位指令进行优化，因为编译器不可能在编译期知道作为被除数的带符号整数是否为负、是否能被整除。不过，对于支持算术右移（即根据符号位补位的右移）指令的平台来说，优化还是可以进行的，只是稍微复杂一点。由上面的分析可知，对于负数 x，除法运算 $x/2^K$ 与右移位运算 $(x+2^K-1)>>K$ 的结果相同，编译器正是利用这一点进行有关的代码优化。

例如，对于带符号整数 x 除以2^K，***gcc*** 优化代码的逻辑如下：

```
%r ← x;                          /* x 的值复制到寄存器 */
IF (%r < 0)                      /* 判断是否为负数 */
   %r += 2^K-1;                  /* 如果是负数则加上2^K-1 */
FI
%r >> K;                         /* 算术右移 K 位 */
```

对整数的左、右移位运算了解透彻之后再来看其它位运算就相对简单，这些位运算包括：按位非、按位与、按位或以及按位异或。即：

op E1

E1 op E2　　(op 指 ~，&，|，^ 等操作符)

上面表达式对应的操作是（注意，按位非运算中没有 E2）：

1）分别计算 E1、E2的值 V1、V2（求值次序不定），对 V1、V2执行 UAC，假设执行 UAC 之后 V1、V2的类型为 T，对应的值分别是 P1、P2

2）计算(P1 op P2)或者(op P1)的值 R ，op 指~、&、|、^等运算

3）位运算表达式的结果值就是 R，类型为 T

我们看到，由于各种位运算要求有相同类型的操作数，所以在进行运算之前编译器必须执行 UAC 操作。

上面的4种位运算直接操作数值的每一个二进制位，所以和平常的算术运算关系不大，P6处理器有对应的指令（NOT、AND、OR 和 XOR）进行这4种位运算。

现在我们讨论一下整数的关系运算，它们是：大于（>）、小于（<）、大于或等于（>=）、小于或等于（<=）、等于（==）和不等于（!=）。对于表达式：

E1 op E2　　(op 指 >，<，>=，<=，==，!= 等操作符)

实际上进行的操作是：

1）计算 E1、E2的值 V1、V2（求值顺序不定），对 V1、V2执行 UAC，假设执行完 UAC 之后 V1、V2对应的值分别是 P1、P2

2）判断(P1 op P2)是否为真

3）如果(P1 op P2)为真，则表达式的值是“1”，否则表达式的值是零，无论表达式的值是“1”还是零，类型都是 int

再一次地，我们看到 UAC 操作必须在关系运算前进行，因为接下来进行比较的两个操作数需要拥有相同的类型。

需要注意的是，“==”和“!=”的优先级比其它关系运算低。

例如：

```
a > b == c < d;
```

对应的操作是：

```
(a > b) == (c < d);
```

由于比较大小的判断涉及到同一数值的不同解释，所以带符号整数和无符号整数之间的区别仍然需要我们打醒十二分精神给予留意。看这个经典的例子：

```
unsigned x = 1;                          /* 无符号整数 */
if ( x > -1)
    f();
else
    g();
```

到最后，被执行的究竟是函数 f()还是 g()？毫无疑问，答案是 g()。因为在比较大小之前，编译器对两个操作数分别进行了 UAC 操作，根据整数类型确定规则，最后参与比较大小的是两个 unsigned int 数值：0x00000001和0xFFFFFFFF，对于无符号整数，0xFFFFFFFF 当然要比0x00000001大，于是 g()被调用。

P6处理器使用 CMP 指令比较两个整数值的大小，CMP 指令实际上进行减法操作并按照具体情况设置各个标志位（CF/OF/SF/ZF），然后就可以通过 SETcc 系列指令得到比较结果或者通过 Jcc 系列指令实现相关跳转。例如：

```
extern int x, y;
if (x > y)
    f();
...
```

对应的汇编代码是：

```
     movl    x, %eax
     cmpl    y, %eax
     jle     .L2
     call    f
.L2:
     ...
```

代码的逻辑非常简单，首先把 x 复制到 eax 寄存器，然后用 CMP 指令比较 y 和 eax 的值，紧着接 JLE 指令根据 CMP 指令设置的相关标志位判断是否进行跳转，如果标志位

的数值表明比较结果是小于或等于（less or equal），则直接跳转到“.L2”处执行其它指令，否则调用函数 f()、从 f()返回后继续执行余下的指令。

上面已经提到，CMP 指令实际上进行的是如假包换的减法运算，它对标志位的设置和减法指令 SUB 完全一致，CMP 和 SUB 的区别仅仅是 CMP 不会把减法运算的结果写进目的操作数。由于使用“2的补码”，P6处理器在减法运算中一律实行简单二进制减法，无论对带符号整数还是无符号整数，对于相同的二进制值，减法得到的结果都是一样的。因此，在比较大小的时候，关键是看对应的符号位数值组合。例如，假设 x 是0x0FFFFFFF、y 是0x80000000，则(x-y)就是0x8FFFFFFF，同时 P6处理器的标志位设置情况是：

CF=1，简单二进制减法过程中出现了借位，因此 CF 被置“1”

OF=1，如果把 x、y、(x-y)看作是带符号整数，则正数减负数却得到负数，因此 OF 被置“1”

SF=1，如果把(x-y)看作带符号整数，则(x-y)的符号位是“1”，因此 SF 被置“1”

ZF=0，(x-y)不为零，所以 ZF 被置零

有了这4个标志位，处理器就可以判断 x 和 y 的大小关系。譬如，如果 x、y 是带符号整数，则 x 大于 y，因为 ((SF xor OF) or ZF)等于零；如果 x、y 是无符号整数，则 x 小于 y，因为 CF 等于“1”。

可见，对于不同类型的整数，处理器能够根据不同的标志位数值组合会得出不同的大小判断，至于处理器选择哪些标志位进行判断，这是由 SETcc 和 Jcc 系列指令决定的。以 Jcc 系列指令为例：

JG	参与比较的数值是带符号整数，如果大小关系是大于则跳转
JL	参与比较的数值是带符号整数，如果大小关系是小于则跳转
JE	如果大小关系是等于则跳转
JA	参与比较的数值是无符号整数，如果大小关系是大于则跳转
JB	参与比较的数值是无符号整数，如果大小关系是小于则跳转
JNE	如果大小关系是不等于则跳转
JGE	参与比较的数值是带符号整数，如果大小关系是大于或等于则跳转
JLE	参与比较的数值是带符号整数，如果大小关系是小于或等于则跳转
JAE	参与比较的数值是无符号整数，如果大小关系是大于或等于则跳转
JBE	参与比较的数值是无符号整数，如果大小关系是小于或等于则跳转
...	...

显然，使用哪条指令是由编译器决定的，而编译器就是根据 UAC 操作之后参与比较的两个操作数的类型来确定应该使用什么指令。因此，从一开始我们就要清楚参与比较大小的两个整数值的类型，否则比较出来的结果会让我们大吃一惊，就好象前面那个“-1”比“1”大的例子那样。

对于64位整数的大小比较，P6处理器依然可以胜任。尽管处理器每一次只能比较两个32位操作数，但可以把连续两次32位值比较组合起来从而完成64位值的比较。例如，要比较两个64位整数 x、y 的大小，可以先用 CMP 指令比较 x、y 的高32位值，如果本次比较未能得出比较的结果（这通常因为 x、y 的高32位值相等）则再次用 CMP 指令比较 x、y 的低32位值，无论如何，经过两次比较之后一定可以判断出 x、y 的大小关系。

最后，我们再来看看整数类型的逻辑运算：

```
! E1
E1 && E2
E1 || E2
```

逻辑运算很简单，编译器不需要对操作数进行 UAC 操作，C 标准仅仅规定，如果 E1 的值不为零，则(!E1)的值为零，否则(!E1)的值是“1”；如果 E1、E2其中任何一个的值为零则(E1 && E2)的值就是零，否则(E1 && E2)的值是“1”；如果 E1、E2其中任何一个的值是“1”，那么(E1 || E2)的值就是“1”，否则(E1 || E2)的值为零，以上三种逻辑运算结果的类型都是 int。

与前面其它运算对 E1、E2求值的过程不同，在逻辑运算中，C 标准规定编译器总是先对 E1求值，如果 E1的值是零则表达式(E1 && E2)中的 E2不被求值；如果 E1的值是“1”则表达式(E1 || E2)中的 E2不被求值。

例如：

```
int x = 0;
if (x && (++x))                 /* (++x)永远不会被执行 */
    ++x;
```

上面，由于在计算(x && (++x))的过程中(++x)没有被求值，因此 x 的值到最后仍然是零。

P6处理器仍然是利用 CMP 指令进行整数的逻辑运算，只不过这一回更省事了，两个操作数 E1、E2全部与零进行比较即可，比较结果要么是相等要么是不相等，然后用 SETE/SETNE 或者 JE/JNE 指令进行下一步操作。

*　　*　　*　　*　　*　　*

[1301]

如非特别指出，本节后面所有的讨论均以使用“2的补码”的系统为例作示范讲解。

[1302]

在基于 AT&T 风格的汇编代码中，原来的 Intel 指令 cbw、cwde、cwd、cdq 被分别改为以 cbtw、cwtl、cwtd、cltd 表示。

[1303]

注意，枚举类型也属于整数类型。

14 浮点实数类型

有两个简单的事实：第一，整数类型不能用来表示那些绝对值“很大的”数值，即使是64位的 long long 类型，其最大（小）值大约只不过是9×10^{18}（-9×10^{18}），这个数值范围远远不能满足通常的科学、工程计算需要。就算把长度再加倍，使用128位整数也不能彻底解决问题，况且我们不可能耗费这么多的存储空间去记录这些其实还不算很大的数值。第二，除了整数以外，我们在很多场合还需要使用小数，由于小数的某些特点我们不能使用表示整数的方式去表示小数。因此，引入一种不同于整数类型的记数方式成为处理器设计者的必然选择。其实，在平时人们已经一直在使用类似的记数方法，例如通常的科学记数法：

1234.5 被记为 $+1.2345\times10^{3}$

计算机处理器广泛采用的浮点记数方式也是基于同样的结构原理，就是说，一个浮点实数由以下部分组成：符号、有效数字、底、指数。不失一般性，我们可以把一个浮点实数表示为：

$(-1)^{s}b^{E}(d_0.d_1d_2\ldots d_{p-1})$

s 只能取“0”或者“1”这两个值，s 是“0”时，浮点实数为非负数；s 是“1”时，浮点实数为非正数，因此我们只需用一个二进制位就能表示浮点实数的符号。b 是浮点实数的底，绝大多数处理器的底都是2，但也有极少数例外，例如 IBM 的 S/370系统的底就是16。当然，在同一平台上对于每一个浮点实数这个底都是相同的数值。E 就是指数，b^{E} 构成一个幂。$d_0.d_1d_2\ldots d_{p-1}$就是浮点实数的有效数字部分（一共有 p 位有效数字），显然，p 表征浮点实数的精度，p 越大，有效数字越多，数值精度越高。

上面只是浮点实数的一般表达式，但具体到一个浮点实数的长度是多少字节、精度是多少位有效数字、指数部分如何表示等等仍然未能确定。事实上，当初的处理器的确在这些问题上存在不少差异，为了统一计算机处理器上的浮点实数格式，让浮点实数据能跨平台使用创造条件，IEEE 在1985年制订了二进制浮点运算标准，标准编号为754，ANSI 于同年批准 IEEE 754成为美国国家标准[1]。IEEE 754描述的是那些选择2作为浮点实数的底的处理器所应该遵循的规范，但对于像 IBM S/370这些底不是2的系统便有点不适用，于是 IEEE 在1987年又推出与底数无关的浮点运算标准，标准编号是854。同年，ANSI 批准 IEEE 854成为美国国家标准[2]。随后，国际组织 IEC 在1989年批准 IEEE 754/854成为国际标准，文档编号是 IEC 559:1989，后来标准文档进行了更新，编号更改为 IEC 60559。至此，浮点运算标准正式在世界范围内确立并逐步得到广泛支持，所有支持浮点运算的处理器都完全支持（或部分支持）IEEE 754/854标准。C99标准的浮点运算包含了对 IEC 60559的支持，就是说，凡是符合 IEC 60559标准的实现同时也符合 C99标准，因此，从 C99开始我们可以在 IEC 60559标准的规范下使用 C 语言进行浮点运算而再也不必担心可移植性问题。

虽然不是所有的计算机处理器都选择2作为浮点实数的底，但支持 IEEE 754的处理器毕竟占绝大多数，因此我们将会详细分析 IEEE 754标准，并且在今后的讲解中均默认以2为底，除非有特别说明。在 IEEE 754中，指定长度的浮点实数格式只有两种：single（单精度浮点实数）和 double（双精度浮点实数），single 的长度固定为4字节，double 的长度固定为8字节。其中，在 single 的32个二进制位里面，1个位存放符号，8个位存

放指数，23个位存放有效数字；而在 double 的64个位中，1个位存放符号，11个位存放指数，52个位存放有效数字：

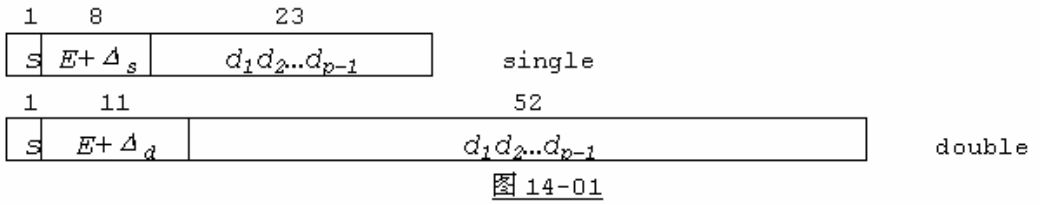


图 14-01

要注意以下几点：

由于第一位有效数字 d_0的值固定是“1”，所以 d_0并不存放在有效数字域，因此，虽然 single 格式中只有23个位存放有效数字，但 single 其实有24位有效数字，即 p 等于24，同理 double 有53位有效数字，即 p 等于53。

指数域存放的不是浮点实数的真正指数 E，而是$(E+\Delta)$，对于 single，Δ_s是127；对于 double，Δ_d等于1023。并且指数域里的不同数值有不同的意义，对于 single 格式，指数域有8位，这意味着$(E+\Delta_s)$的数值范围是[0, 255]：

1）$(E+\Delta_s)$等于255而 $d_1d_2\ldots d_{p-1}$中至少有一位不为零，则该浮点实数属于 NaN；[3]

2）$(E+\Delta_s)$等于255而 $d_1d_2\ldots d_{p-1}$全部为零，则该浮点实数表示“+∞”（如果 s 等于“0”零）或“-∞”（如果 s 等于“1”）；

3）$(E+\Delta_s)$落在(0, 255)中，则该浮点实数的值是：$(-1)^s2^E(1.d_1d_2\ldots d_{p-1})$；

4）$(E+\Delta_s)$等于0而 $d_1d_2\ldots d_{p-1}$中至少有一位不为零，则该浮点实数的值是：$(-1)^s2^{-126}(0.d_1d_2\ldots d_{p-1})$；

5）$(E+\Delta_s)$等于0而 $d_1d_2\ldots d_{p-1}$全部为零，则该浮点实数表示“+0”（如果 s 等于“0”零）或“-0”（如果 s 等于“1”）；

而对于 double 格式，指数域有11位，这意味着$(E+\Delta_s)$的数值范围是[0, 2047]：

1）$(E+\Delta_d)$等于2047而 $d_1d_2\ldots d_{p-1}$中至少有一位不为零，则该浮点实数属于 NaN；

2）$(E+\Delta_d)$等于2047而 $d_1d_2\ldots d_{p-1}$全部为零，则该浮点实数表示“+∞”（如果 s 等于“0”零）或“-∞”（如果 s 等于“1”）；

3）$(E+\Delta_d)$落在(0, 2047)中，则该浮点实数的值是：$(-1)^s2^E(1.d_1d_2\ldots d_{p-1})$；

4）$(E+\Delta_d)$等于0而 $d_1d_2\ldots d_{p-1}$中至少有一位不为零，则该浮点实数的值是：$(-1)^s2^{-1022}(0.d_1d_2\ldots d_{p-1})$；

5）$(E+\Delta_d)$等于0而 $d_1d_2\ldots d_{p-1}$全部为零，则该浮点实数表示“+0”（如果 s 等于“0”零）或“-0”（如果 s 等于“1”）。

根据这些规定，我们不难算出 IEEE 754中 single 和 double 类型浮点实数的最大（有限）值，对于 single 格式的浮点实数，最大（有限）值的二进制形式为：

01111111011111111111111111111111

即 $S_{MAX} = (1.111\ldots1)_2 \times 2^{127} = (2^{24}-1)\times2^{104}$ [4]

这里我们先求出 S_{MAX}的十进制数量级：

$\lg(S_{MAX}) = \log_2(S_{MAX}) \times \lg2$

$$= [\log_2(2^{24}-1)+104] \times \lg 2$$
$$\approx 38.5$$

因此，S_{MAX} 的数量级是10^{38}。至于 single 的最小（有限）值 S_{MIN}，仅仅是与最大（有限）值符号相反，数量级同样是10^{38}。

对于 double 格式的浮点实数，最大（有限）值的二进制形式为：

0111111111101111111111111111111111111111111111111111111111111111

即 $D_{MAX} = (1.111\ldots1)_2 \times 2^{1023} = (2^{53}-1)\times 2^{971}$

D_{MAX} 的十进制数量级是：

$$\lg(D_{MAX}) = \log_2(D_{MAX}) \times \lg 2$$
$$= [\log_2(2^{53}-1)+971] \times \lg 2$$
$$\approx 308.3$$

因此，D_{MAX} 的数量级是10^{308}。至于 double 的最小（有限）值 D_{MIN}，仅仅是与最大（有限）值符号相反，数量级同样是10^{308}。

在前面对 single 和 double 的值进行的分类中，除去“+0”和“-0”外，有限值分两种：规格化（normalized）浮点值和非规格化（denormalized）浮点值。对于 singal 格式，规格化值的指数部分（8个位）不全为零，而非规格化值的指数部分则全是零；对于 double 格式，规格化值的指数部分（11个位）不全为零，而非规格化值的指数部分则全是零。对于规格化值，小数点左边的值被默认是“1”；对于非规格化值，小数点左边的值被默认是“0”，当然，single 和 double 格式的浮点实数都不会在数据中保存这个值。

我们很容易算出最小规格化正值以及最小非规格化正值的十进制数量级，对于 single 格式，最小规格化正值的二进制形式为：

00000000100000000000000000000000

即 $S_{NMINP} = (1.000\ldots0)_2 \times 2^{-126} = 2^{-126}$

$$\lg(S_{NMINP}) = \log_2(S_{NMINP}) \times \lg 2$$
$$= -126 \times \lg 2$$
$$\approx -37.9$$

因此 S_{NMINP} 的数量级是10^{-38}。

而最小非规格化正值的二进制形式为：

00000000000000000000000000000001

即 $S_{DMINP} = (0.000\ldots01)_2 \times 2^{-126} = 2^{-149}$

$$\lg(S_{DMINP}) = \log_2(S_{DMINP}) \times \lg 2$$
$$= -149 \times \lg 2$$
$$\approx -44.9$$

因此 S_{DMINP} 的数量级是10^{-45}。

对于 double 格式，最小规格化正值的二进制形式为：

0000000000010000000000000000000000000000000000000000000000000000

即 $D_{NMINP} = (1.000\ldots0)_2 \times 2^{-1022} = 2^{-1022}$

$\lg(D_{NMINP}) = \log_2(D_{NMINP}) \times \lg 2$

$= -1022 \times \lg 2$

≈ -307.7

因此 D_{NMINP} 的数量级是10^{-308}。

而最小非规格化正值的二进制形式为：

0000000000000000000000000000000000000000000000000000000000000001

即 $D_{DMINP} = (0.000\ldots01)_2 \times 2^{-1022} = 2^{-1074}$

$\lg(D_{DMINP}) = \log_2(D_{DMINP}) \times \lg 2$

$= -1074 \times \lg 2$

≈ -323.3

因此 D_{DMINP} 的数量级是10^{-324}。

IEEE 754除了定义 single 和 double 格式的浮点实数外还定义了 single extended（单精度扩展）和 double extended（双精度扩展）浮点实数：

格式参数	格式类型			
	single	single ext	double	double ext
p	24	≥ 32	53	≥64
E_{MAX}	+127	≥ +1023	+1023	≥ +16383
E_{MIN}	-126	≤ -1022	-1022	≤ -16382
Δ	+127	不指定	+1023	不指定
指数位数	8	≥ 11	11	≥ 15
数据长度	32	≥ 43	64	≥ 79

扩展精度格式有绝对值更大的指数和更多的有效数字，但 IEEE 754指出，扩展精度格式是可选的，就是说，各个实现平台可以不支持扩展精度格式。事实上，大多数浮点运算处理器的确只支持 single 和 double 两种格式的浮点实数，只有少数的处理器，例如 P6以及 Motorola 的680x0系列处理器支持双精度扩展格式。另外，还有个别处理器，譬如 UltraSPARC，支持四倍精度格式“quad”。quad 格式由另一个 IEEE 标准（IEEE 1596.5-1993）[6]进行定义，虽然没有被列入 IEEE 754标准因此暂时未能得到广泛的支持，但由于 quad格式已经由其它 IEEE 标准定义，因此 quad 很有可能在将来成为工业界广泛支持的第三种标准浮点实数格式。

这里要特别说一下 double ext 格式，P6处理器所支持的 double ext 的格式参数值是上面列出的最小值：p 等于64、E_{MAX} 为(+16383)、E_{MIN} 为(-16382)、Δ等于(+16383)、指数位数是15、数据长度是10字节（80位）。上面79这个数字是 IEEE 754按照省略掉小数点左边的那个位计算出来的，但实际上处理器不可能支持长度为79位的数据格式（因为数据位数必须是8的倍数），因此，那些支持 double ext 的处理器就把

原来并不存储在数据中的那个“默认位”改为显式存储在数据中，这样就凑够80位，P6处理器同样也不例外。

确定了浮点实数的格式之后，我们也许最关心的就是浮点实数的舍入（rounding）方式以及由此带来的误差。在浮点运算或者格式转换的过程中，系统经常要对浮点值进行舍入以适应目标格式。IEEE 754规定各实现平台必须提供4种舍入方式：

- **round to nearest**

这是IEEE 754指定的默认舍入方式。顾名思义，系统将把浮点值转换为最接近原值的适当形式。例如，假设某一次浮点运算的结果是：

0 00000001 00000000000000000000000 111

这个数值是浮点处理器内部进行运算产生的，现在要把结果保存到32位的 single 浮点寄存器，于是系统必须要对数值进行舍入处理，如果舍入方式是“to nearest”，那么最后寄存器的值应该是：

0 00000001 00000000000000000000000**1**

即 $S = (1.00000000000000000000001)_2 \times 2^{-126}$

明显地，S 是最接近原值的 single 浮点实数。

如果有两个值都同样接近原值，那么，根据 IEEE 754标准，系统应该选择最后一个有效数字为“0”的那个值作为舍入结果。例如，系统要把浮点值：

0 00000001 00000000000000000000000 1

舍入为 single 浮点实数，依照“nearest”方式，有两个值都满足要求：

0 00000001 00000000000000000000000**1**

0 00000001 00000000000000000000000**0**

这时，系统应该选择后者作为结果值，因为它的最后一位有效数字是“0”。

不过要注意一点，当浮点值的绝对值达到或超过$2^{E_{MAX}}(2-2^{-p})$时，按照 IEEE 754的规定，系统在“round to nearest”模式下必须用相同符号的“∞”表示舍入结果。以 single 浮点实数为例，如果原值达到或超过：

0 11111110 11111111111111111111111 1

则“to nearest”方式的舍入结果应该是“+∞”：

0 11111111 00000000000000000000000

- **round toward +∞**

在这种舍入方式下，系统必须选择最接近原值而又大于或等于原值的浮点实数作为舍入结果。例如，系统要将：

0 00000001 00000000000000000000000 1

舍入为 single 浮点实数，方式是“toward +∞”。虽然有两个浮点实数都最接近原值，但如果加上“大于或等于”原值这个条件，那么符合要求的只有一个：

0 00000001 00000000000000000000000**1**

因此，“toward +∞”方式的舍入结果是：

$(1.00000000000000000000001)_2 \times 2^{-126}$

显然，当原值大于$2^{E_{MAX}}(2-2^{1-p})$时，舍入结果就是“+∞”，例如：

0 11111110 11111111111111111111111 01

最接近而又大于或等于它的 single 浮点实数只能是“+∞”：

```
0 11111111 00000000000000000000000
```

- **round toward -∞**

这种舍入方式与2）相似，只不过换了方向。系统必须选择最接近原值而又小于或等于原值的浮点实数作为舍入结果。例如，系统要把：

```
0 00000001 00000000000000000000000 1
```

舍入为 single 值，虽然有两个数都最接近原值，但只有一个是小于或等于它：

```
0 00000001 00000000000000000000000
```

因此“toward -∞”方式的舍入结果就是：

$(1.00000000000000000000000)_2 \times 2^{-126}$

- **round toward 0**

不难想象，这种舍入方式要求系统选择最接近原值而绝对值又小于或等于原值的浮点实数作为舍入结果。例如，在“toward 0”方式下，浮点值：

```
0 11111110 11111111111111111111111 1
```

舍入为 single 浮点实数，则舍入结果是：

```
0 11111110 11111111111111111111111
```

即 $(1.11111111111111111111111)_2 \times 2^{127}$

上面多个例子说明一个简单的事实：同一个数值在不同的舍入方式下具有不同的舍入结果，尽管这些结果在对应的舍入方式中都是最接近原值的数值，因此，我们必须留意浮点运算系统当前工作在什么舍入方式下这个细节。

现在我们讨论一下二进制浮点实数与十进制浮点实数之间的转换问题。很显然，由于十进制的所有数字都能被数字“2”在有限次运算之内除尽，因此所有二进制浮点实数都能转换为只有有限个有效数字的十进制浮点实数。但反过来就不一样了，并不是所有的十进制浮点实数都能转换为只有有限个有效数字的二进制浮点实数，譬如“0.1”就不行：

$10^{-1} = (1.10011001100110011001100110011001100110011001100110011001……)_2 \times 2^{-4}$

不过，即使某些十进制浮点实数可以转换成只有有限个有效数字的二进制浮点实数，但受二进制浮点实数格式的限制（single 只有24个有效数字、double 则有53个），可能会在最后由于必须进行舍入而产生误差。这个问题其实同样存在于二进制浮点实数转换为十进制浮点实数的过程中，尽管我们心里很清楚转换后的十进制浮点实数只有有限位有效数字，但受到各种条件的限制，我们通常不能把每个这些有效数字全部都储存起来或打印出来，于是。进行必要的舍入是在所难免的。因此，无论是二进制浮点实数转换成十进制浮点实数，还是十进制浮点实数转换成二进制浮点实数，一般都会遇到舍入所带来的问题。我们最关心的恰恰是，转换过程中的这些舍入误差究竟在多大程度上影响数值的准确性。

首先分析一下二进制浮点实数转换为十进制浮点实数。这个问题其实是：对于某个二进制浮点实数，我们要用多少个十进制有效数字才能表示它才足够。“足够”的意思是：当我们把这个十进制浮点实数再转换回二进制浮点实数时所得到的结果和原来的二进制

浮点实数完全相等。由于 single 和 double 分别有24、53位二进制有效数字，这对于我们的讨论来说实在太大，所以我们先用一个简化的模型来说明基本的原理。假设有一种二进制浮点实数格式，名称是“simple”。simple 有5位二进制有效数字，现在问，需要多少位十进制有效数字来表示 simple 浮点实数才能保证将来可以正确还原？反应快的朋友可能会马上说，5个二进制有效数字的组合一共只有32（即2^5）种，因此只需要2位十进制有效数字就可以保证转换的正确性，因为2位十进制数字可以表示100（即10^2）种数值，而100明显大于32——足够了。然而，事实并非如此。譬如在数值区间[10,16)里，simple 浮点实数一共有12种数值，而2位十进制有效数字的浮点实数就只有6种数值，显然地，用2位十进制有效数字来表示 simple 浮点实数并不够准确：

simple 浮点值	十进制值（3位有效数字）	十进制值（2位有效数字）
$(1010.0)_2$	10.0	10
$(1010.1)_2$	10.5	11
$(1011.0)_2$	11.0	11
$(1011.1)_2$	11.5	12
$(1100.0)_2$	12.0	12
$(1100.1)_2$	12.5	13
$(1101.0)_2$	13.0	13
$(1101.1)_2$	13.5	14
$(1110.0)_2$	14.0	14
$(1110.1)_2$	14.5	15
$(1111.0)_2$	15.0	15
$(1111.1)_2$	15.5	16

显然，只用2位十进制有效数字得到的数值无法进行有效区分，系统不可能根据同一个十进制值还原出两个不同的 simple 浮点实数。后面我们将会证明，使用3位有效数字的十进制浮点实数就可以保证还原的准确性，这里只是以数值区间[1, 2)为例，看看3位十进制有效数字的转换情况是怎样的：

simple 浮点值	十进制值（5位有效数字）	十进制值（3位有效数字）
$(1.0000)_2$	1.0000	1.00
$(1.0001)_2$	1.0625	1.06
$(1.0010)_2$	1.1250	1.13
$(1.0011)_2$	1.1875	1.19
$(1.0100)_2$	1.2500	1.25
$(1.0101)_2$	1.3125	1.31
$(1.0110)_2$	1.3750	1.38
$(1.0111)_2$	1.4375	1.44
$(1.1000)_2$	1.5000	1.50
$(1.1001)_2$	1.5625	1.56

$(1.1010)_2$	1.6250	1.63
$(1.1011)_2$	1.6875	1.69
$(1.1100)_2$	1.7500	1.75
$(1.1101)_2$	1.8125	1.81
$(1.1110)_2$	1.8750	1.88
$(1.1111)_2$	1.9375	1.94

虽然完整地表示 simple 浮点实数最多甚至需要5位十进制有效数字，但如果仅仅只是保证还原过程不产生偏差，则3位有效数字就足够。譬如要把1.94转换回 simple 浮点实数，那么我们可以通过计算得到1.94对应的二进制浮点值是：

$1.94 = (1.1111000\ldots)_2$

无疑，最接近$(1.1111000\ldots)_2$的 simple 浮点值就是$(1.1111)_2$，整个转换过程：

$(1.1111)_2 \rightarrow 1.94 \rightarrow (1.1111)_2$

没有任何损失，simple 浮点实数$(1.1111)_2$转换成十进制浮点实数1.94之后再转换回来仍然是$(1.1111)_2$，同理可以分析其它数值的情形。

现在，我们引入一个基本原理。

在某个数值区间内给出：

1）一系列 *M* 进制浮点实数 X_i，所有的 X_i 具有相同的 *M* 进制有效数字位数及指数

2）一系列 *N* 进制浮点实数 Y_j，所有的 Y_j 具有相同的 *N* 进制有效数字位数及指数

如果把 X_i 转换成对应的 *N* 进制浮点实数 Y_{ij}，集合$\{Y_{ij}\}$是集合$\{Y_j\}$的子集

那么，要保证转换 $X_i \rightarrow Y_{ij} \rightarrow X_i$ 是准确的，只要 Y_j 的 *ulp* 总是小于 X_i 的 *ulp* 即可。

缩写“*ulp*”（Unit in the Last Place）表示一个数的最后一个数位的单位数值，例如：

3.14	*ulp* 是10^{-2}
2.987×10^4	*ulp* 是10
$(1.0011)_2\times2^{-7}$	*ulp* 是2^{-11}

显然，有效数字位数以及指数都能影响 *ulp* 的大小。

在上述原理中，由于 X_i 的 *ulp* 大于 Y_i 的 *ulp*，即是说，相邻两个 X_i 之间的“空隙”总是大于相邻两个 Y_i 之间的“空隙”，于是在区间$[Y_i-ulp/2, Y_i+ulp/2]$[7]中要么不包含 Y_{ij}、要么只包含一个 Y_{ij}。这样的话，由 X_i 到 Y_{ij} 的转换就是唯一的，从而由 Y_{ij} 到 X_i 的转换也是唯一的，这样就保证了 Y_{ij} 能够还原成 X_i。

以前面的 simple 浮点实数为例，假设二进制指数是3，那么 simple 浮点实数集合：

$\{X_i\} = \{ (1000.0)_2, (1000.1)_2, (1001.0)_2, (1001.1)_2,$
$(1010.0)_2, (1010.1)_2, (1011.0)_2, (1011.1)_2,$
$(1100.0)_2, (1100.1)_2, (1101.0)_2, (1101.1)_2,$
$(1110.0)_2, (1110.1)_2, (1111.0)_2, (1111.1)_2 \}$

X_i 的 *ulp* 等于(2^{-1})，即0.5。X_i 分布在数值区间[8, 16)中，由于区间[8, 16)刚好横跨在10^0和10^1的交界，我们可以把[8, 16)分成两部分：[8, 10)与[10, 16)。

我们来看看为什么只有2位十进制有效数字的浮点实数不能保证转换的唯一性。

显然地，在区间[10, 16)里，2位十进制有效数字组成的 Y_j 的指数是1，于是 Y_j 的 ulp 等于1。Y_j 的 ulp 大于 X_i 的 ulp，区间[Y_i-0.5, Y_i+0.5]里面包含有不止一个 Y_{ij}。譬如，在区间[12-0.5, 12+0.5]中就有2个 Y_{ij}：12、12（当 X_i 等于$(1011.1)_2$、$(1100.0)_2$时对应的 Y_{ij} 都是12），所以，从 X_i 到 Y_{ij} 的转换不是唯一的，系统无法根据两个重叠的 Y_{ij} 还原出各自对应的 X_i。如果有效数字增加到3个，那么在区间[10, 16)里 Y_j 的指数是1，于是 Y_j 的 ulp 等于0.1，比 X_i 的 ulp 小，因此在区间[Y_i-0.1, Y_i+0.1]里要么没有 Y_{ij} 要么只有一个 Y_{ij}，这个结论大家可以自行验证。

根据这个原理，我们找到了计算“转换-还原”过程所需有效数字位数的方法。

给出任意一个二进制浮点实数$2^E(d_0.d_1d_2\ldots d_{p-1})$，[8]必定存在一个只有有限个有效数字的十进制浮点实数与之相等，我们把这个十进制浮点实数表示为$10^m(D_0.D_1D_2\ldots D_{k-1})$，于是有：

$$2^E(d_0.d_1d_2\ldots d_{p-1}) = 10^m(D_0.D_1D_2\ldots D_{k-1}) \quad (1)$$

在这里，二进制浮点实数有 p 位有效数字，十进制浮点实数有 k 位有效数字，我们需要解决的问题是：在十进制浮点实数的 k 个有效数字当中选取多少位就可以保证准确的“转换-还原”。假设选取 n 位，那么依照前面的原理，Y_i 的 ulp 必须小于 X_i 的 ulp，即：

$$10^{m-(n-1)} < 2^{E-(p-1)}$$

两边取对数、整理后得到：

$$n > (1+p\lg 2)+\lg(10^m/2^{E+1})$$

由等式(1)可知：

$$10^m/2^{E+1} < 1$$

即：$\lg(10^m/2^{E+1}) < 0$

但同时也存在某些数值使得 $\lg(10^m/2^{E+1})$ 相当接近0

因此，实际上 n 必须满足：

$$n > (1+p\lg 2)$$

才可以保证准确地“转换-还原”。考虑到 n 是正整数，我们把上式改写为：

$$n = \mathbf{ceil}(1+p\lg 2)$$

很意外地，我们发现 n 其实和二进制浮点实数的指数无关，n 的大小仅仅取决于二进制浮点实数的有效数字个数 p。

不妨计算一下前面 simple 浮点实数对应的 n：

$$n = \mathbf{ceil}(1+5\times\lg 2) = 3$$

这个数值我们已经用一些实际数据检验过。

IEEE 754的 single、double 对应的 n 值分别是：

$$n_S = \mathbf{ceil}(1+24\times\lg 2) = \mathbf{ceil}(8.2) = 9$$

$$n_D = \mathbf{ceil}(1+53\times\lg 2) = \mathbf{ceil}(16.95) = 17$$

当我们把 single 浮点实数转换成十进制浮点实数，为保证该十进制浮点实数将来可以正确地还原回原先的 single 浮点实数，则十进制浮点实数的有效数字最少不能少于9位；如果把 double 浮点实数转换成十进制浮点实数，为保证该十进制浮点实数将来能够正确地还原回原先的 double 浮点实数，则十进制浮点实数的有效数字最少不能少于17

位。

要是换一种更直接、更容易理解的表达方式，我们可以这样说，假设 S_1、S_2是任意两个不相等的 single 浮点实数，把它们转换为对应的十进制浮点实数 X_1、X_2后，不可能存在这样一种情况：X_1和 X_2的数量级相等以及前9位有效数字全部相同；假设 Y_1、Y_2是任意两个不相等的 double 浮点实数，把它们转换为对应的十进制浮点实数 Y_1、Y_2后，不可能存在这样一种情况：Y_1和 Y_2的数量级相等以及前17位有效数字全部相同。于是，依靠数量级以及前9/17位十进制有效数字我们就可以在还原过程中准确分辨出任意两个 single/double 浮点实数，我们不妨把这条规律称为“B-D-B 定则”。

上面讨论的是二进制浮点实数转换为十进制浮点实数的问题，问题的核心是需要多少个十进制有效数字才能保证能够准确地还原回原先的二进制浮点实数。下面讨论的问题刚好相反：当十进制浮点实数转换为二进制浮点实数时，至多允许十进制浮点实数有多少位有效数字才能保证将来能够准确地还原回原先的十进制浮点实数。

这次的“转换-还原”过程是：十进制浮点实数 → 二进制浮点实数 → 十进制浮点实数。

例如 single 浮点实数有24位二进制有效数字，那么当我们把一个具有 q 位十进制有效数字的浮点实数转换为 single 格式时，在保证将来可以正确地还原回该十进制数值的前提下，这个 q 最大能取多少？

显然，q 不可能等于9。因为根据前面的计算，当 q 等于9时，X_i 的 ulp 大于 Y_i 的 ulp，[9]而原理告诉我们，这时从 X_i 到 Y_{ij} 的转换不是唯一的。不过，依照同样的方法我们还是可以计算出 q 的上限。

给出任意一个十进制浮点实数 $10^m(D_0.D_1D_2\ldots D_{q-1})$，我们要把它转换为只有有限个有效数字的二进制浮点实数 $2^E(d_0.d_1d_2\ldots d_{p-1})$，为了保证转换是无损失的，根据原理，$Y_i$ 的 ulp 必须小于 X_i 的 ulp，即：

$$2^{E-(p-1)} < 10^{m-(q-1)}$$

两边取对数、整理后得到：

$$q < (p-1)\lg 2+\lg(10^{m+1}/2^E)$$

由于：$2^E(d_0.d_1d_2\ldots d_{p-1}) \le 10^m(D_0.D_1D_2\ldots D_{q-1})$ [10]

因此：$10^{m+1}/2^E > 1$

即：$\lg(10^{m+1}/2^E) > 0$

但同时也存在某些数值使得 $\lg(10^{m+1}/2^E)$ 相当接近0

所以，实际上 q 必须满足：

$$q < (p-1)\lg 2$$

才可以保证“转换-还原”的正确性。考虑到 q 是正整数，我们把上式改写为：

$$q = \mathbf{floor}((p-1)\lg 2)$$

可以看到，q 只和 p 有关。

IEEE 754的 single、double 对应的 q 值分别是：

$$q_S = \mathbf{floor}((24-1)\times\lg 2) = \mathbf{floor}(6.9) = 6$$

$$q_D = \mathbf{floor}((53-1)\times\lg 2) = \mathbf{floor}(15.7) = 15$$

当我们把一个十进制浮点实数转换成 single 浮点实数，为保证 single 浮点实数

将来可以正确地还原回该十进制浮点实数，则十进制浮点实数的有效数字最多不能超过6位；如果把一个十进制浮点实数转换成 double 浮点实数，为保证 double 浮点实数将来能够正确地还原回该十进制浮点实数，则十进制浮点实数的有效数字最多不能超过15位。

同样，要是换一种表达方式，我们可以这样说，假设 X_1、X_2是任意两个不相等的十进制浮点实数，只要 X_1和 X_2的前6位有效数字不完全相同，那么把它们转换为对应的 single 浮点实数 S_1、S_2后，S_1和 S_2就肯定不相等；假设 Y_1、Y_2是任意两个不相等的十进制浮点实数，只要 Y_1和 Y_2的前15位有效数字不完全相同，那么把它们转换为对应的 double 浮点实数 D_1、D_2后，D_1和 D_2就肯定不相等。类似地，我们把该规律称为“D-B-D定则”。

C 语言标准规定了三种浮点实数类型（real floating type）：float、double 和 long double，浮点实数类型和整数类型都属于实数类型（real type）。但 C99标准没有直接指定每一种浮点实数类型的指数、有效数字位数等参数具体是多少，C 标准仅仅在头文件<float.h>中给出了部分参考值，各实现平台对应的具体值必须在绝对值上大于或等于参考值而符号和参考值保持一致。

<float.h>定义的宏如下（“*”表示该宏是 C99新增加的）：

宏	参考值	宏	参考值
FLT_ROUNDS		FLT_EVAL_METHOD *	
FLT_RADIX	2	DECIMAL_DIG *	10
FLT_MANT_DIG		FLT_DIG	6
DBL_MANT_DIG		DBL_DIG	10
LDBL_MANT_DIG		LDBL_DIG	10
FLT_MIN_EXP		FLT_MIN_10_EXP	-37
DBL_MIN_EXP		DBL_MIN_10_EXP	-37
LDBL_MIN_EXP		LDBL_MIN_10_EXP	-37
FLT_MAX_EXP		FLT_MAX_10_EXP	37
DBL_MAX_EXP		DBL_MAX_10_EXP	37
LDBL_MAX_EXP		LDBL_MAX_10_EXP	37
FLT_MIN	1E-37	FLT_MAX	1E+37
DBL_MIN	1E-37	DBL_MAX	1E+37
LDBL_MIN	1E-37	LDBL_MAX	1E+37
FLT_EPSILON	1E-5		
DBL_EPSILON	1E-9		
LDBL_EPSILON	1E-9		

FLT_ROUNDS 代表浮点运算系统的默认舍入方式，C99给出5个有固定含义的值：

```
-1      无法确定
 0      round toward 0
 1      round to nearest
```

2　　round toward +∞

3　　round toward -∞

各实现平台如果还有其它舍入方式则可以自行定义其它（整数）值。

FLT_EVAL_METHOD（C99新增加）表示浮点运算系统进行浮点运算时使用的临时格式，例如，在某些系统上，浮点处理器内部会先把 float、double 浮点实数全部转换为更大的浮点实数格式再进行运算，而某些系统则不会进行这样的转换。前者的典型代表是P6处理器，我们已经知道，P6处理器除了支持 IEEE 754的 single、double 浮点实数之外还支持80位的 double ext 浮点实数，因此处理器内部8个浮点运算寄存器的长度都是80位的。P6处理器的默认工作方式是：当处理器从内存装入32/64位浮点实数时，浮点单元会自动把32/64位浮点实数转换为80位浮点实数，此后的所有浮点运算均使用80位浮点实数，等到运算完毕处理器从寄存器输出结果时浮点单元又会自动把80位浮点实数舍入为32/64位的结果再复制到内存。而属于后者的系统很多，几乎带有浮点处理功能的所有 RISC 处理器都是，例如 MIPS32/64系统。带有浮点协处理器的 MIPS32/64系统只支持 single 和 double 浮点实数，并且在指令的层面上对两种格式进行划分，就是说，两个 single 浮点实数进行运算使用一条指令，而两个 double 浮点实数进行同样的运算又是使用另一条指令。

C99给出4个有固定含义的值：

-1　　无法确定

0　　使用相同长度的格式进行运算

1　　float、double 一律转换为 double 格式再运算，long double 则还是以 long double 格式进行运算

2　　所有运算都以 long double 格式进行

各实现平台如果还有其它运算方式则可以自行定义其它（整数）值，不过，自行定义的值必须是负数。

FLT_RADIX 是浮点处理系统的底，除了最常见的2之外，还有10（HP 的某些工作站系列）和16（IBM 的 S/370系统）等。

FLT_MANT_DIG、DBL_MANT_DIG、LDBL_MANT_DIG 分别表示 float、double、long double 类型的二进制有效数字个数。

DECIMAL_DIG（C99新增加）表示长度最大的浮点实数类型（即 long double）对应的 n 值，即准确的“B→D→B 转换-还原”中最小的十进制有效数字位数值。

FLT_DIG、DBL_DIG、LDBL_DIG 分别表示 float、double、long double 类型对应的 q 值，即准确的“D→B→D 转换-还原”中最大的十进制有效数字位数值。

FLT_MIN_EXP、DBL_MIN_EXP、LDBL_MIN_EXP 分别表示 float、double、long double 类型中满足“(2^{E-1})是规格化浮点实数”的最小 E 值。

FLT_MIN_10_EXP、DBL_MIN_10_EXP、LDBL_MIN_10_EXP 分别表示 float、double、long double 类型中满足“(10^{E})是规格化浮点实数”的最小 E 值。

FLT_MAX_EXP、DBL_MAX_EXP、LDBL_MAX_EXP 分别表示 float、double、long double 类型中满足“(2^{E-1})是规格化浮点实数”的最大 E 值。

FLT_MAX_10_EXP、DBL_MAX_10_EXP、LDBL_MAX_10_EXP 分别表示 float、double、long double 类型中满足“(10^{E})是规格化浮点实数”的最大 E 值。

FLT_MIN、DBL_MIN、LDBL_MIN 分别表示 float、double、long double 类型中对应的最小规格化正值 S_{NMINP}、D_{NMINP}、X_{NMINP}。

FLT_MAX、DBL_MAX、LDBL_MAX 分别表示 float、double、long double 类型中对应的最大规格化值 S_{MAX}、D_{MAX}、X_{MAX}。

FLT_EPSILON、DBL_EPSILON、LDBL_EPSILON 分别表示 float、double、long double 类型中对应的与“1”最接近的浮点值减去“1”的差。

我们现在分析一下 P6系统典型的<float.h>实现。P6处理器支持 IEEE 754的 single、double、double ext 格式浮点实数，分别对应 C 语言的 float、double、long double 类型。其中，double ext 格式的 p、E_{MIN}、E_{MAX}、Δ 值分别是：64、16383、(-16382)、16383，因此，double ext 格式的各项基本参数为：

$lg(X_{MAX}) = log_2(X_{MAX})\times lg2 = [log_2(2^{64}-1)+16320]\times lg2 \approx 4932.1$

最大规格化值 X_{MAX} 的数量级是10^{4932}。

$n_X = \mathbf{ceil}(1+64\times lg2) = \mathbf{ceil}(20.3) = 21$

如果把 double ext 浮点实数转换成十进制浮点实数，为保证该十进制浮点实数将来能够正确地还原回原先的 double ext 浮点实数，则十进制浮点实数的有效数字最少不能少于21位。

$q_X = \mathbf{floor}((64-1)\times lg2) = \mathbf{floor}(18.96) = 18$

如果把一个十进制浮点实数转换成 double ext 浮点实数，为保证 double ext 浮点实数将来能够正确地还原回该十进制浮点实数，则十进制浮点实数的有效数字最多不能超过18位。

另外，single、double、double ext 类型中与“1”最接近的浮点值减去“1”的差分别为：

$\varepsilon_S = 2^{-23} = 1.19209290\times10^{-7}$

$\varepsilon_D = 2^{-52} = 2.2204460492503131\times10^{-16}$

$\varepsilon_X = 2^{-63} = 1.0842021724855044 3401\times10^{-19}$

single、double、double ext 类型中对应的最小规格化正值分别为：

$S_{NMINP} = 2^{-126} = 1.17549435\times10^{-38}$

$D_{NMINP} = 2^{-1022} = 2.2250738585072014\times10^{-308}$

$X_{NMINP} = 2^{-16382} = 3.3621031431120935 0626\times10^{-4932}$

single、double、double ext 类型中对应的最大规格化值分别为：

$S_{MAX} = (2-2^{-23})\times2^{127} = 3.40282347\times10^{38}$

$D_{MAX} = (2-2^{-52})\times2^{1023} = 1.7976931348623157\times10^{308}$

$X_{MAX} = (2-2^{-63})\times2^{16383} = 1.1897314953572317 6502\times10^{4932}$

single、double、double ext 类型中满足“(10^E)是规格化浮点实数”的最小 E 值分别为：

$E_S = \mathbf{ceil}(lg(S_{NMINP})) = \mathbf{ceil}(-37.9) = -37$

$E_D = \mathbf{ceil}(lg(D_{NMINP})) = \mathbf{ceil}(-307.7) = -307$

$E_X = \mathbf{ceil}(lg(X_{NMINP})) = \mathbf{ceil}(-4931.5) = -4931$

这里唯一要注意的是，single、double、double ext 类型中满足“(10^E)是规格化浮点实数”的最大 E 值分别就是 single、double、double ext 类型最大规格化

值的数量级指数值；而 single、double、double ext 类型中满足“(10^E)是规格化浮点实数”的最小 E 值却不是 single、double、double ext 类型最小规格化正值的数量级指数值。正如计算 E_S、E_D、E_X的过程所揭示的那样，3对数值各自相差“1”。[11]

于是，对应 P6系统的<float.h>典型实现如下：

```
#define    FLT_ROUNDS           1
#define    FLT_EVAL_METHOD      2
#define    FLT_RADIX            2
#define    DECIMAL_DIG          21
#define    FLT_MANT_DIG         24
#define    FLT_EPSILON          1.19209290E-07F
#define    FLT_DIG              6
#define    FLT_MIN_EXP          (-125)
#define    FLT_MIN_10_EXP       (-37)
#define    FLT_MIN              1.17549435E-38F
#define    FLT_MAX_EXP          128
#define    FLT_MAX_10_EXP       38
#define    FLT_MAX              3.40282347E+38F
#define    DBL_MANT_DIG         53
#define    DBL_EPSILON          2.2204460492503131E-16
#define    DBL_DIG              15
#define    DBL_MIN_EXP          (-1021)
#define    DBL_MIN_10_EXP       (-307)
#define    DBL_MIN              2.2250738585072014E-308
#define    DBL_MAX_EXP          1024
#define    DBL_MAX_10_EXP       308
#define    DBL_MAX              3.362103143112093506 26E+308
#define    LDBL_MANT_DIG        64
#define    LDBL_EPSILON         1.08420217248550443401E-19L
#define    LDBL_DIG             18
#define    LDBL_MIN_EXP         (-16381)
#define    LDBL_MIN_10_EXP      (-4931)
#define    LDBL_MIN             3.362103143112093506 26E-4932L
#define    LDBL_MAX_EXP         16384
#define    LDBL_MAX_10_EXP      4932
#define    LDBL_MAX             1.189731495357231765 02E+4932L
```

从上面的讨论可以知道，虽然 C 标准并没有明确规定浮点实数类型的太多细节，但由于 IEEE 754已经成为工业标准，绝对多数的浮点处理器都是遵循 IEEE 754进行设计的，因此，在大部分场合我们的确可以认为：C 语言的 float 和 double 就是对应于 IEEE 754

的 single 和 double。至于 C 里面的 long double 类型，除了极少数支持超过64位的浮点实数格式的处理器之外，大部分系统上 long double 其实就是 double，两者均表示64位浮点实数类型,这种情形和32位平台上 int 和 long 的长度都是32位有点相似。事实上，我们使用得最广泛的仍然是 float、double 格式的浮点实数，因为它们在跨平台的应用中具有最大限度的兼容性。虽然 P6处理器和 UltraSPARC 处理器分别支持80位和128位的 long double 类型，但这些系统毕竟为数不多，今后我们的讨论基本上集中在32位的 float 类型和64位的 double 类型。

在展开进一步的讨论前，我们先利用上面的计算结果分析一下浮点实数变量的初始化。当我们需要确保某个浮点实数变量能够取得某个二进制数值，需要使用多少位有效数字才足够呢？例如，我们想把 float 变量 f 初始化为最小规格化正值，而最小规格化正值的二进制形式为：

```
0 00000001 00000000000000000000000
```

它对应的浮点实数是2^{-126}，即：

$1.1754943508222875080\ldots\times10^{-38}$

究竟需要使用多少位十进制有效数字才能保证变量 f 最后的确取到上面的二进制值？前面的转换-还原理论告诉我们，对于 float 变量，9位十进制有效数字就足够了。即：

```
float f = 1.17549435E-38F                /* 9位十进制有效数字 */
```

如果有效数字少于9位，则不能保证 float 变量取到的值和预期的二进制值一致，如果有效数字多于9位，则完全没有必要。因为根据“B-D-B 定则”，当十进制有效数字达到9位时，f 不可能取到其它的二进制值。

而对于 double 变量，17位十进制有效数字就足够了。

另一方面，是不是所有前9位有效数字不完全相同的十进制浮点实数都会转换成不同的 float 二进制值呢？答案是否定的，例如：

```
float f1 = 16777219.0F;
float f2 = 16777220.0F;
```

虽然 f1、f2的前9位有效数字不完全相同，但最后它们的取值都是：

```
0 10010111 00000000000000000000010
```

毫无疑问，如果使用 printf()之类的函数把 f1、f2打印成十进制数值的话，两者对应的打印结果肯定是完全相同。其实根据“D-B-D 定则”，只有前6位有效数字不完全相同的十进制浮点实数才必然地转换出不同的二进制浮点实数值，因此我们不能期望从第7个有效数字开始不相同的两个十进制 float 浮点实数一定能转换成两个互不相同的二进制浮点值。对于 double 浮点实数，只有前15位有效数字不完全相同的十进制浮点实数才必然地转换成不同的二进制浮点值。

C 语言只有3种浮点实数类型，因此编译器对浮点型实常数和实变量的类型确定也就简单了许多。不过，要注意的是，前面提到的 K&R C 规则对浮点实数同样起作用：

- 表达式（包括函数的参数）中的所有 float 数值都被提升为 double 数值
- 浮点常数的类型是 double

C89增加了 long double 类型，而且兼容 K&R C 的规则，在没有函数原型声明的情况下，函数参数中的所有 float 数值仍然会被提升为 double 数值，浮点常数的类型

仍然是 double，除非加上显式的后缀“f/F”或“l/L”。特别条款也适用于浮点实数类型，如果编译器能够保证不进行类型提升就直接进行运算的结果与先进行类型提升再进行运算的结果相同，则编译器可以自行优化代码而省略类型提升这一步。

例如：

```
float x, y, z;
z = x + y;
```

按照 K&R C 规则，计算 z 的过程如下：

```
z = (float)((double)x + (double)y)
```

x、y 必须被提升为 double 类型才能参与加法运算，并且得到的结果值同样是 double 类型，最后才把 double 值转换为 float 值再赋给 z。

C89允许编译器在满足“结果相同”的前提条件下这样优化代码：

```
z = x + y;                    /* 直接运算、赋值 */
```

譬如，在 UltraSPARC 系统上，传统的“先提升后运算”规则对应下面的伪汇编代码：

```
%f4  ←  x
fstod   %f4, %f4              /* float 提升为 double */
%f6  ←  y
fstod   %f6, %f6              /* float 提升为 double */
faddd   %f6, %f4, %f4         /* 计算两个 double 值的和 */
fdtos   %f4, %f4              /* 把 double 值转换为 float 值 */
z   ←  %f4
```

而引用 C89特别条款后的伪汇编代码则相当简略：

```
%f4  ←  x
%f5  ←  y
fadds   %f5, %f4, %f4          /* 直接计算两个 float 值的和 */
z  ←  %f4
```

由于优化效果显著，现代编译器都会尽可能地引用特别条款。不过，也不是任何场合都可以进行优化的，例如：

```
float x, y;
double z;
z = x + y;
```

由于(x+y)可能超出 float 类型的最大有限值，所以编译器不能引用特别条款，只能老老实实地先提升再运算：

```
z = (double)x + (double)y
```

函数的 float 参数不再一律提升为 double，而是根据函数原型来确定。如果函数原型中有对应的参数类型说明，则 float 参数不会被提升：

```
void f(float);
float x;
f(x);                    /* 传给函数的是4个字节的 float 值 */
```

上面 x 是 float 变量，由于函数原型已经指出函数 f()的参数是 float 类型，因

此 x 的值不会被提升为 double 就直接传递给函数。

如果没有函数原型又或者即使有函数原型但没有对应参数的类型说明，则 K&R C 的提升规则就要起作用：

```
float x;
int f();              /* 这是函数声明但不是函数原型 */
void g(int, ...);  /* 这是函数原型但没有浮点参数的相关信息 */
f(x);                 /* x 的值先提升为 double 值再传给函数 */
g(4, x);              /* x 的值先提升为 double 值再传给函数 */
```

C99对浮点实数类型的提升、转换规则基本上和 C89相同，K&R C 规则仍然保留下来，特别条款也依然存在。

更值得我们注意的是浮点实数类型与整数类型之间的转换，先看一下浮点实数转换为整数的问题。

当浮点实数类型（用“T1”表示）被转换为整数类型（用“T2”表示）时，它的小数部分会被去掉从而只剩下整数部分，这个剩下来的整数部分就是转换后的值。如果 T2 无法存放转换后的值，那就会导致 undefined behavoir。

以 P6处理器为例，编译器用来实现浮点实数转换为整数的通常是 FISTP 指令，该指令可以把浮点寄存器 st(0)的值转换为16/32/64位三种长度的带符号整数值(浮点寄存器的值不改变)，然后把整数值复制到内存操作数指定的地址。例如：

```
extern int i;
extern float f;
i = f;                          /* float 值转换为 int 值 */
```

对应的汇编指令主要是：

```
flds     f           # 把 f 的值复制到浮点寄存器 st(0)
...                  # 修改浮点处理器的舍入方式为“round toward 0”
fistpl   i           # 转换为32位带符号整数值然后复制给 i
...                  # 恢复浮点处理器的舍入方式为“round to nearest”
```

其中，FISTP 指令在进行数值转换时就会判断目标操作数是否足够用来存放转换后的整数值，三种目标操作数对应的带符号整数值范围分别是$[-2^{15}, 2^{15}-1]$、$[-2^{31}, 2^{31}-1]$和$[-2^{63}, 2^{63}-1]$，如果转换后的值超出对应的范围则 P6处理器会根据 FPU 控制寄存器的 IM 位是否清零决定是否引发16号异常。而且，处理器的判断是严格限定对应的数值范围，例如，要是内存操作数是32位的，那么如果转换后的整数值是2^{31}，则处理器仍然判定该数值超出转换范围，尽管2^{31}是0x80000000，而0x80000000完全可以存放在32位内存操作数中（当然，对数值的解释完全不同）。这和我们前面讨论的整数之间的转换有点不一样，在整数转换过程中，类似：

```
unsigned u = 0x80000000U;          /* 无符号整数值231 */
int i = u;                         /* 转换 */
```

的整数转换虽然实际上没有意义，但各种编译器以及运行平台事实上都会默许这样的转换，尽管2^{31}已经超出 int 的数值范围，但赋值后 i 的值仍然是0x80000000（解释为“-1”）。现在，FISTP 指令的做法却是严格判断是否超出数值范围，这一点我们必须搞

清楚。

在Linux平台上，FPU控制寄存器的6个异常屏蔽位：PM/UM/OM/ZM/DM/IM位默认全部置“1”，因此，P6处理器不会引发16号异常。不过，用户进程可以自行修改FPU控制寄存器的值，从而使得处理器会根据具体情况引发16号异常。Linux对16号异常的处理是给用户进程发送SIGFPE信号，如果用户进程不捕获该信号处理则系统立即终止进程并在终端打印出“Floating point exception”。

影响引发异常的不仅仅是异常屏蔽位，还有编译器的相关策略。前面说过，FISTP指令可以有三种长度的内存操作数，编译器可以自由决定使用何种长度的内存操作数。假设编译器的策略是使用和T2长度相等的操作数，那仍然还有signed char和unsigned char没有对应的操作数，如果要把浮点实数转换为signed char，例如：

```
extern float f;
extern signed char c;
c = f;
```

那编译器必然要先把f转换为16/32/64位带符号整数然后再截短，例如：

```
flds    f
...
fistpl  temp            # temp是32位内存操作数
movb    temp, c         # 截短，把temp的最低字节复制给c
...
```

于是便肯定存在一部分数值本来超出signed char的数值范围但没有机会去引发异常，譬如128。128没有超出32位带符号整数的范围，因此无论FPU控制寄存器的IM位是否清零P6处理器都不会引发异常。同样道理，如果编译器一律使用64位内存操作数的FISTP指令然后再根据具体情况进行截短，那么将会有更多的数值根本不可能引发异常。总之，在浮点实数转换为整数的过程中，一旦转换后的类型不能存放转换值，我们很难说后果将会是什么。当然，C标准完全可以保证处于特定数值范围的浮点实数一定会得到正确的转换，假设T2的长度是N位，那么$(-2^N-1, 2^N)$范围内的浮点实数都可以正确转换为整数，这是确凿无疑的。

对于浮点实数转换为整数，只要没有超出对应数值范围，我们几乎可以马上说出转换后的数值，很简单，就是原来浮点实数的整数部分。从上面的例子看出，编译器的实现是先将P6处理器的浮点舍入方式改变为“round toward 0”，然后执行FISTP指令。显然，“去掉小数部分”与“在round toward 0方式下转换为整数”是相同的概念。

现在，我们讨论整数转换为浮点实数的问题。在整数转换为浮点实数的过程中，我们不用担心数值超界，因为即使是最短的float类型，它的数量级也达到10^{38}，目前最大的整数类型long long的最大值也不超过10^{19}，所以浮点实数类型肯定足以存放任何整数值。我们需要认真考虑的是精确度，因为浮点实数的有效数字有限，根据前面的计算，float只能保证最多6位十进制有效数字的整数进行准确的“转换-还原”，double则只允许15位。看这个例子：

```
C01         /* Code 14-01, file name: 1401.c
C02         #include <stdio.h>
C03         int main()
```

```
C04         {
C05             int i = 16777219;
C06             float f = i;
C07             int n = f;
C08             printf("i=%d\nf=%f\nn=%d\n", i, f, n);
C09         }
```

编译、执行14-01：

```
$gcc  1401.c
$./a.out
i=16777217
f=16777216.000000
n=16777216
```

很清楚，原先 i 的初始值是16777217，转换为 float 类型之后，无论是 f 还是再转换回来的 n 都不是16777217，而是16777219。如果我们观察一下 f 的二进制值便很清楚地知道为什么。

首先，16777216 = 2^{24}，把16777216表示为 float 格式：

```
0 10010111 00000000000000000000000
```

即：$1.00000000000000000000000\times2^{24}$，由于指数是24，所以 *ulp* 是2，但对于整数类型来说，*ulp* 是1。根据“转换-还原”原理，float 不能保证准确还原。16777217比12777216大1，如果用更多的有效数字去表达12777217，那么对应12777217的二进制值应该是：

```
0 10010111 00000000000000000000001
```

可惜，float 只有24位有效数字，因此最接近12777217的 float 值是：

```
0 10010111 00000000000000000000000
```

或：

```
0 10010111 00000000000000000000001
```

C 标准规定，在这种情况下，C 实现可以自行选择任何一个数值作为16777217的转换值，因为两个值都同样接近原值。但根据 IEEE 754的原则，在 round to nearest 方式下优先选择最低有效数字位是零的值。这就是我们上面看到的结果，f 的值最后就是16777216。

同理，我们马上可以知道，在16777217后面的16777219又会遇到相同的情况，最接近16777219的 float 值只有16777218和16777220，因此，如果把16777219转换为 float 值，我们得到的结果将是16777220。大家可以自行验证这个结论。

对于 P6处理器，编译器使用 FILD 指令实现整数转换为浮点实数。同样地，FILD 可以有16/32/64位三种长度的内存操作数，处理器统一把三种长度的带符号整数转换为80位浮点实数存放在浮点寄存器 st(0)。不同于 FISTP 指令，FILD 不存在数值超界的问题，也没有任何舍入误差。这完全是因为转换的结果是80位浮点实数格式，我们知道，double ext 格式有64位二进制有效数字，即使是最长的64位带符号整数，它的数值范围也只不过$[-2^{63}, 2^{63}-1]$，就是说，即使二进制指数达到62，double ext 浮点实数

多达64位的有效数字仍然使到 *ulp* 的值是0.5（仍然小于1），至于二进制指数达到63的唯一一个值(-2^{63})，80位浮点实数完全能够准确表示它。所以，FILD 指令的确不存在任何转换上的舍入误差。[12]但由于我们通常都是使用 float、double 浮点实数，因此在处理器把内部的80位浮点实数舍入为32/64位的 float/double 浮点实数后，有效数字位数不足带来的误差还是无法避免。一部分整数值的有效数字位数超过 q(q_S=6，q_D=15)就会引起转换误差，而另一部分整数值的有效数字位数即使超过 q 却没有误差，这种情况我们在上面的例子中已经见过。总之，如果我们一定要保证不存在任何转换误差的话，那就必须遵守“6/15”规则。

最后，必须要说明一点，P6处理器的 FILD 指令把所有整数都看作是带符号整数，编译器如果看到要转换的整数属于 unsigned 类型，它就会把对应的内存操作数的长度提高一个级别，譬如，当要把一个 unsigned short 值转换为浮点实数，编译器就使用带32位内存操作数的 FILD 指令，当要把一个 unsigned int 值转换为浮点实数，则编译器使用带64位内存操作数的 FILD 指令，由于 FILD 的内存操作数最多只能是64位，因此我们无法将[2^{63},2^{64}-1]内的整数转换为浮点实数（FILD 指令把那些值看作是带符号整数值）。

知道了整数类型和浮点实数类型各自的提升、转换以及它们之间相互的转换规则之后，这里给出更详细的 C99寻常算术转换（UAC）步骤：

1）如果两边任何一个操作数是 long double，则另一个应被转换为 long double；

2）如果两边任何一个操作数是 double，则另一个应被转换为 double；

3）如果两边任何一个操作数是 float，则另一个应被转换为 float；

然后，对于所有级别（rank）低于 int 的整数类型，如果 int 能够表示提升前的数值，则该数值被提升为 int 类型，否则提升为 unsigned int 类型；而所有级别大于或等于 int 的整数类型则无须提升。接着再使用下列规则对运算量进行转换：

1）如果两边的操作数类型相同则不需要任何转换。

2）如果两边都是带符号类型或都是无符号类型，则把低级别类型的操作数转换为高级别的类型。

3）如果一个是带符号的（用 S 表示），另一个是无符号的（用 U 表示），并且 U 的级别高于或等于 S 的级别，则把 S 转换为 U 的类型。

4）如果一个是带符号的（用 S 表示），另一个是无符号的（用 U 表示），并且 S 的类型能够表示 U 的类型的所有数值，则把 U 转换为 S 的类型。

5）如果一个是带符号的（用 S 表示），另一个是无符号的（用 U 表示），并且以上规则都不适用，则把 S 和 U 转换为与 S 的类型级别相同的无符号类型。

例如：

```
int i;   short s;    float x;    double y;
(i + x) * (y - l)   →  (double)((float)i + x) * (y - (doube)s)
(x + i * s) / y   →  (double)(x + (float)(i * (int)s)) / y
```

*　　*　　*　　*　　*　　*

[1401]

请参阅[IEEE 1985]。

[1402]

请参阅[IEEE 1987]。

[1403]

NaN（Not a Number）表示该二进制值并不表示任何有意义的浮点实数，NaN 有两种：SNaN（Signaling NaN）和 QNaN（Quiet NaN）。后面会讲到，如果参与浮点运算的操作数中有 SNaN 则处理器通常会引发某种异常，如果是 QNaN 则不会引发异常。

[1404]

single 有24个二进制有效数字，但小数点左边那个有效数字没有被存放在数据中。

[1405]

double 有53个二进制有效数字，但小数点左边那个有效数字没有被存放在数据中。

[1406]

请参阅[IEEE 1993]。

[1407]

这里的 *ulp* 是指 Y_i 的 *ulp*。

[1408]

这里讨论的都是正数，负数的情形是一样的，只不过符号相反而已。

[1409]

注意，这个时候 X_i 代表十进制浮点实数，Y_i 代表二进制浮点实数。

[1410]

前面已经提到，不是所有的十进制浮点实数都能准确地用有限个有效数字的二进制浮点实数表示，因此对于只有 *p* 位有效数字的二进制浮点实数来说，该不等式总是成立的。

[1411]

以 single 为例，single 类型的最小规格化正值是1.17549435×10^{-38}，该值的数量级是10^{-38}，但1.0×10^{-38}已经比1.17549435×10^{-38}还要小，因此，满足“(10^E)是规格化浮点实数”的最小 *E* 值不是(-38)而是(-37)。

[1412]

有人可能会感到不解：明明$(2^{63}-1)$有19位有效数字，而前面已经证明了，80位浮点实数格式能够保证最多18位有效数字的十进制浮点实数的准确“转换-还原”，为什么却说 FILD 指令不存在任何误差呢？

道理很简单，前面证明的结论适用于所有的十进制浮点实数，只要有效数字不超过18位，80位浮点实数都保证能够正确还原，无论是：

123456789098765432

还是

$1.23456789098765432\times10^{-100}$

而 FILD 指令转换的对象是整数，整数的 *ulp* 固定是1，这就削弱了前提条件，问题变成了80位浮点实数能够保证正确地“转换-还原”多少位有效数字的整数。一如正文分析的那样，答案明显大于18。

15 浮点实数运算及异常

IEEE 754实现环境必须对浮点实数提供以下基本运算、操作的支持：

1）算术运算，包括加法、减法、乘法（*）、除法、求余[1]和求平方根；

2）不同格式浮点实数之间的转换；

3）浮点实数与整数之间的转换；

4）把浮点实数舍入为最接近的整数（结果仍然是浮点实数格式）；

5）二进制浮点实数与十进制浮点实数之间的转换；

6）浮点实数之间的大小比较。

IEEE 754规定，对系统所有格式的浮点实数都要提供上述操作的支持，简单地说，对于一个支持 single 和 double 浮点实数格式的典型环境，无论是对 single 还是 double 浮点实数都可以进行以上6类操作。

对应于这6类操作，P6处理器分别提供了以下指令：

1）FADD、FSUB、FMUL、FDIV、FPREM1、FSQRT；

2）FLD、FST/FSTP；

3）FILD、FIST/FISTP；

4）FRNDINT

5）（没有现成的指令，必须通过具体的算法进行一系列计算）

6）FCOM/FUCOM

而在 C 语言中，除了位运算（即&，|，^，~，<<，>>）、取模运算（%）以及相应的复合赋值运算（即%=，<=，>>=，&=，|=，^=）之外，其它运算操作符均可以对浮点实数进行操作，包括：

- +，-，*，/，求相反数，++，--　　（算术运算）
- >，<，>=，<=，==，!=　　（关系运算）
- =，+=，-=，*=，/=　　（赋值运算）
- &&，　!，||　　（逻辑运算）

和整数运算一样，如果有必要的话，在参与运算之前浮点实数会被施加适当的 UAC 操作，以提升、转换为合适的类型，例如：

```
float x;   double y, z;
z = x + y;  →  z = (double)x + y;     /* x 先转换为 double 类型再参与运算 */
```

关于这方面的内容今后不再反复提及，大家参考 UAC 规则即可。而浮点实数的一些运算则具有特殊的性质，稍后我们将会详细讨论该问题。

在各种浮点实数的运算、操作过程中，当出现某些情况时，系统必须通过特定的方式通知用户，IEEE 754用“异常（exception）”这个术语来表示这些情况。至于系统用什么方式去通知用户则没有具体规定，处理器可以设置某个标志位或者自动进入异常处理过程。IEEE 754定义了5种异常，分别是：Invalid Operation（无效操作）、Division by Zero（除数为零）、Overflow（上溢）、Underflow（下溢）、Inexact（不准确）。这些异常可以分为两大类，一类是由参与运算的操作数所导致的，另一类则是由运算的结果所引起的。“无效操作”和“除数为零”这两种异常显然属于前者，因为浮点处理器还没有真正的计算而仅仅依据操作数就知道应该引发异常；“上溢”、“下溢”和“不准确”

异常则属于后者，处理器要等到具体计算结束的时候才知道是否应该引发异常。

对于 P6处理器，它的 FPU 控制寄存器分别有6个异常屏蔽位，如果对应屏蔽位置“1”，则处理器遇到浮点异常时仅仅设置 FPU 状态寄存器相应的标志位而不会引发16号系统异常；如果对应屏蔽位置“0”，则处理器遇到浮点异常时不仅设置 FPU 状态寄存器相应的标志位还会引发16号系统异常。下面我们详细讨论一下 IEEE 754定义的5种浮点实数运算异常。

- **Invalid Operation** （以下用“754-IO”表示）

当满足下列情况时，系统应该引发754-IO 异常，然后，系统要么进入陷阱（trap）处理过程要么用 QNaN 作为对应的浮点实数运算结果：

1）对 SNaN 的所有操作

2）(+∞)+(-∞)、(-∞)+(+∞)、(+∞)-(+∞)、(-∞)-(-∞)

3）0*(+∞)、0*(-∞)

4）0/0、(+∞)/(-∞)、(+∞)/(+∞)、(-∞)/(-∞)、(-∞)/(+∞)

5）求余运算(x *REM* y)中，y 等于0或者 x 是(+∞)或(-∞)

6）求平方根运算√x 中，x 小于0

7）在二进制浮点实数转换为整数或十进制浮点实数时，如果操作数是“∞”、NaN^2 又或者转换后的结果会溢出，而系统又没有其它的方式表示该情形的产生

8）对两个浮点实数进行第2类方式的大小比较运算（稍后对此有详细说明），两个操作数中至少有一个是 NaN

我们这里先用一个小程序检验一下 P6处理器是否符合 IEEE 754的规定，让大家对754-IO 异常有个初步的了解：

```
C01       /* Code 15-01, file name: 1501.c
C02       #include<stdio.h>
C03       #include<fenv.h>
C04       #include<math.h>
C05       #pragma STDC FENV_ACCESS ON
C06       int snan = 0x7F800001;             /* SNaN */
C07       int qnan = 0x7FC00001;             /* QNaN */
C08       int pinf = 0x7F800000;             /* +∞ */
C09       int ninf = 0xFF800000;             /* -∞ */
C10       int pzero = 0x00000000;              /* +0.0 */
C11       int nzero = 0x80000000;              /* -0.0 */
C12       float f;
C13       long l;
C14
C15       void test()
C16       {
C17          int fp_except = fetestexcept(FE_INVALID);
C18          if (fp_except & FE_INVALID)
C19          {
```

```
C20             printf("FPU raised 754-IO exception\n");
C21             feclearexcept(FE_INVALID);
C22         }
C23         else
C24             printf("FPU did not raise 754-IO exception\n");
C25     }
C26
C27     int main()
C28     {
C29         feclearexcept(FE_ALL_EXCEPT);
C30         printf("SNaN + (+0.0)\t");
C31         f = *(float*)(&snan) + 0.0;
C32         test();
C33         printf("(+INF) + (-INF)\t");
C34         f = *(float*)(&pinf) + *(float*)(&ninf);
C35         test();
C36         printf("(+INF) * (+0.0)\t");
C37         f = *(float*)(&pinf) * 0.0;
C38         test();
C39         printf("(+0.0) / (-0.0)\t");
C40         f = *(float*)(&pzero) / (*(float*)(&nzero));
C41         test();
C42         printf("(+INF) / (+INF)\t");
C43         f = *(float*)(&pinf) / *(float*)(&pinf);
C44         test();
C45         printf("1.0 REM (+0.0)\t");
C46         remainder(1.0, 0.0);
C47         test();
C48         printf("SQRT(-1.0)\t");
C49         sqrt(-1.0);
C50         test();
C51         printf("(int)(+INF)\t");
C52         l = lrint(*(float*)(&pinf));
C53         test();
C54         printf("SNaN > 1.0\t");
C55         if (*(float*)(&snan) > 1.0)
C56             ;
C57         else
C58             test();
C59     }
```

15-01使用了C99新增的标准库函数和宏，这些函数和宏被声明和定义在<fenv.h>头文件，我们后面再详细讲述它们的用法。这里暂时只需知道函数 test()可以检查 P6处理器是否标示出754-IO异常的产生。我们编译、运行15-01：

```
$gcc -std=c99 1501.c -lm [3]
$./a.out
SNaN + (+0.0)        FPU raised 754-IO exception
(+INF) + (-INF)      FPU raised 754-IO exception
(+INF) * (+0.0)      FPU raised 754-IO exception
(+0.0) / (-0.0)      FPU raised 754-IO exception
(+INF) / (+INF)      FPU raised 754-IO exception
1.0 REM (+0.0)       FPU raised 754-IO exception
SQRT(-1.0)           FPU raised 754-IO exception
(int)(+INF)          FPU raised 754-IO exception
SNaN > 1.0           FPU raised 754-IO exception
```

运行结果显示 P6处理器的确按照 IEEE 754标准规定的那样对上述8种情形标示出754-IO异常的产生。这里有必要对15-01作出一些解释。

▪ IEEE 754规定指数部分的位全部是“1”而有效数字部分的位不是全部为“0”的二进制浮点实数是 NaN，并且每一个 IEEE 754实现系统都必须至少定义一个 SNaN（signaling NaN）、至少定义一个QNaN（quiet NaN）。换言之，究竟SNaN和QNaN的二进制值具体是什么，IEEE 754没有指明，我们只知道SNaN和QNaN所在的范围是（以single格式为例，“x”表示该位数值可以是“0”又可以是“1”）：

```
x11111111000000000000000000000001 ~~ x11111111111111111111111111111111
```

对于P6处理器，落在以下范围的值都被看作SNaN：

```
x11111111000000000000000000000001 ~~ x11111111011111111111111111111111
```

而落在以下范围的值则被看作QNaN：

```
x11111111100000000000000000000000 ~~  x11111111111111111111111111111111
```

因此，在15-01里面我们可以把全局变量snan、qnan分别初始化为0x7F800001、0x7FC00000来代表SNaN、QNaN。但在随后的加法运算过程中，我们不能直接这样写：

```
f = snan + 0.0;              /* snan的类型是int */
```

根据UAC规则，上面的加法、赋值操作实际上是：

```
f = (float)((double)snan + 0.0);            /* 对snan进行强制转换 */
```

然而，我们并不希望snan被转换为double类型，因为一旦经过转换，snan的值就不是SNaN了。所有我们用了一点小戏法保证snan的值能够原封不动地参与加法运算：

```
f = *(float*)(&snan) + 0.0;              /* 参与运算的是snan原来的值 */
```

我们通过强制转换(&snan)的类型为(float*)来“欺骗”编译器把snan的值看作是float值从而能够直接参与加法运算。[4]

※ 不是所有的系统都好象P6处理器那样定义SNaN、QNaN，例如MIPS32/64就是刚好相反的：上面对应SNaN的值被定义成QNaN、上面对应QNaN的值被定义成SNaN。而SPARC-V9的定义方法则和P6

处理器一致。 ※

▪ GNU/Linux 默认情况下把 P6处理器 FPU 控制寄存器的6个浮点异常屏蔽位全部置“1”，因此，当出现 IEEE 754异常时 P6处理器仅仅是把 FPU 状态寄存器的相关异常标志位置“1”而不会引发系统16号异常。于是我们只需使用 C99在<fenv.h>新增加的库函数 fetestexcept()就可以知道是否出现了754-IO 异常。test()主要就是完成这项工作，如果处理器的确引发了754-IO 异常那么 test()函数还会进一步调用 C99新库函数 feclearexcept()清除 P6处理器 FPU 状态寄存器对应754-IO 异常的标志位，为下一次检测做好准备。

▪ 在测试情形4）时，由于 ***gcc*** 一看到除数为零就会优化代码，这便导致处理器没有机会发出754-IO 异常，因此我们不得不再次使用“诡计”，把“f = (+0.0)/(-0.0);”写成：

```
f = *(float*)(&pzero) / (*(float*)(&nzero));
```

5

▪ 函数 remainder()、sqrt()、lrint()都是<math.h>里面的数学库函数，分别用来进行求余、平方根以及浮点实数转换为整数等运算。

▪ 在最后一个测试中，我们进行大小关系运算：

```
if (*(float*)(&snan) > 1.0)
```

用 SNaN 与1.0比较大小，比较的结果当然是“假”，关系表达式返回的 int 值是“0”，因此我们在 else 子句中调用 test()检测异常。检测结果和 IEEE 754的规定一致，的确出现了754-IO 异常。在这里一定要使用 SNaN 才能引发754-IO 异常，原因后面再讲述。

这里再补充一下关于 NaN 的用法。顾名思义，SNaN 是专门用来发出某种信号的，因此对它进行的所有6类操作都会引发754-IO 异常，如果引发异常后系统没有进入陷阱（trap）处理过程并且本次操作必须返回一个浮点实数结果，则按规定系统应该返回一个 QNaN。与 SNaN 相比，QNaN 的确“安静”很多，除了上面的情形7）和情形8）有可能引发754-IO 异常外，其它所有对 QNaN 的操作系统一律不引发任何异常。两个操作数中的一个或两个都是 QNaN 时，系统要么用 QNaN 作为浮点实数结果返回，要么就是符合情形7)、情形8）而引发754-IO 异常，例如：

```
QNaN + QNaN          = QNaN
QNaN → 整数          引发754-IO 异常
```

▪ **Division by Zero** （以下用“754-DZ”表示）

当除数为零（包括“+0”和“-0”）而被除数是有限值时，系统应该引发754-DZ 异常，随后，系统要么进入陷阱处理过程，要么返回相应符号的“∞”作为结果。例如：

```
3.0 / +0.0 = +∞
3.0 / -0.0 = -∞
-3.0 / -0.0 = +∞
```

注意754-DZ 异常的引发条件有严格的限制——“被除数是有限值”，在其他情况下：

除数为零，被除数也是零，则系统应该引发754-IO 异常；

除数为零，被除数是“∞”，则系统不引发异常，直接返回相应符号的“∞”作为结

果；

除数为零，被除数是 SNaN，则系统引发754-IO 异常

除数为零，被除数是 QNaN，则系统不引发异常，直接返回 QNaN 作为结果。

我们通过15-02观察 P6处理器对754-DZ 异常的实现：

```
C01       /* Code 15-02, file name: 1502.c */
C02       #include<stdio.h>
C03       #include<fenv.h>
C04       #pragma STDC FENV_ACCESS ON
C05       int qnan = 0x7FC00000;
C06       int pinf = 0x7F800000;
C07       int pzero = 0x00000000;
C08       int nzero = 0x80000000;
C09       float f;
C10
C11       void test()
C12       {
C13          int fp_except = fetestexcept(FE_DIVBYZERO);
C14          if (fp_except & FE_DIVBYZERO)
C15          {
C16             printf("FPU raised 754-DZ exception\n");
C17             feclearexcept(FE_DIVBYZERO);
C18          }
C19          else
C20             printf("FPU did not raise 754-DZ exception\n");
C21       }
C22
C23       int main()
C24       {
C25          feclearexcept(FE_ALL_EXCEPT);
C26          printf("(+INF) / (-0.0)\t");
C27          f = *(float*)(&pinf) / (*(float*)(&nzero));
C28          printf("Result is %F\t", f);
C29          test();
C30          printf("(-3.0) / (-0.0)\t");
C31          f = (-3.0) / (*(float*)(&nzero));
C32          printf("Result is %F\t", f);
C33          test();
C34          printf("QNaN / (+0.0)\t");
C35          f = *(float*)(&qnan) / (*(float*)(&pzero));
C36          printf("Result is %F\t", f);
```

```
C37           test();
C38       }
```

编译、执行15-02：

```
$gcc  -std=c99  1502.c  -lm
$./a.out
(+INF) / (-0.0)  Result is -INF     FPU did not raise 754-DZ exception
(-3.0) / (-0.0)  Result is INF      FPU raised 754-DZ exception
QNaN / (+0.0)    Result is NAN      FPU did not raise 754-DZ exception
```

可以看到，除了“(-3.0)/(-0.0)”之外，处理器并没有引发754-DZ 异常，这和我们前面描述的非常一致。

- **Overflow** （以下用“754-OF”表示）

当操作结果作为浮点实数在绝对值上已经超过目标格式所对应的最大有限值时，处理器应该引发754-OF 异常。这里“超过最大有限值”有两种情形：

1）实际结果的指数直接大于对应格式的最大指数值。例如，signle 浮点实数的最大指数是127，如果浮点运算操作结果是1×2^{128}，那就很明显，的确是上溢。

2）实际结果的指数等于对应格式的最大指数值，但有效数字的舍入使得最后结果的指数仍然大于最大指数值。例如，如果作为运算结果的 single 值是：

```
0 11111110 11111111111111111111111 111
```

即：$(1.11111111111111111111111111)_2\times2^{127}$，由于 single 只有24位有效数字，上面的值（有27位有效数字）必须进行舍入，在“round to nearest”模式下，舍入结果毫无疑问应该是1×2^{128}，同样发生上溢。

舍入模式	运算结果的符号	上溢后返回的值
round to nearest	+	+∞
	-	-∞
round to +∞	+	+∞
	-	绝对值最大的负值
round to -∞	+	最大的正值
	-	-∞
round toward 0	+	最大的正值
	-	绝对值最大的负值

在上面第2）种情况中，舍入模式直接影响是否上溢，不同的舍入模式能够导致不同的结果。例如上面那个有27位有效数字的 single 值如果是在“round toward 0”或者“round to -∞”模式下则不会发生上溢，从而不会引发754-OF 异常。一旦产生754-IO 异常，处理器要么进入陷阱处理过程，要么返回某个值作为结果。这个值具体是什么，同样倚赖于舍入模式，上表给出了各种情形对应的返回值。

类似于前面，我们用15-03测试754-OF 异常以及发生异常后的返回值：

```
C01       /* Code 15-03, file name: 1503.c
C02       #include<stdio.h>
C03       #include<fenv.h>
C04       #pragma STDC FENV_ACCESS ON
C05       long long pvalOF = 0x47EFFFFFFC000000;
C06       long long nvalOF = 0xC7EFFFFFFC000000;
C07       double pd, nd;
C08       float pf, nf;
C09
C10       void test()
C11       {
C12          int fp_except = fetestexcept(FE_OVERFLOW);
C13          if (fp_except & FE_OVERFLOW)
C14          {
C15             printf("FPU raised 754-OF exception\n");
C16             feclearexcept(FE_OVERFLOW);
C17          }
C18          else
C19             printf("FPU did not raise 754-OF exception\n");
C20       }
C21
C22       void round()
C23       {
C24          pf = pd;
C25          printf("pf = %2.1E\t", pf);
C26          test();
C27          nf = nd;
C28          printf("nf = %2.1E\t", nf);
C29          test();
C30       }
C31
C32       int main()
C33       {
C34          feclearexcept(FE_ALL_EXCEPT);
C35          pd = *(double*)(&pvalOF);
C36          nd = *(double*)(&nvalOF);
C37          printf("Round to nearest:\n");
C38          round();
C39          printf("Round to +INF:\n");
```

```
C40           fesetround(FE_UPWARD);
C41           round();
C42           printf("Round to -INF:\n");
C43           fesetround(FE_DOWNWARD);
C44           round();
C45           printf("Round toward 0:\n");
C46           fesetround(FE_TOWARDZERO);
C47           round();
C48       }
```

编译、运行15-03：

```
$gcc  -std=c99  1503.c  -lm
$./a.out
Round to nearest:
pf = INF          FPU raised 754-OF exception
nf = -INF         FPU raised 754-OF exception
Round to +INF:
pf = INF          FPU raised 754-OF exception
nf = -3.4E+38     FPU did not raise 754-OF exception
Round to -INF:
pf = 3.4E+38      FPU did not raise 754-OF exception
nf = -INF         FPU raised 754-OF exception
Round toward 0:
pf = 3.4E+38      FPU did not raise 754-OF exception
nf = -3.4E+38     FPU did not raise 754-OF exception
```

这里对源程序及运行结果进行一些解释。首先我们构造了两个 double 值：

$\pm(1.11111111111111111111111111)\times2^{127}$，显而易见，它们都有27位有效数字并且指数刚好等于 float 浮点实数的最大指数值，然后我们分别把它们赋给 float 变量 pf、nf，这样在赋值的过程中 P6处理器必然要进行舍入。15-03利用 C99新库函数 fesetround()分别把处理器的舍入模式逐次修改[6]为“round to +∞”、“round to -∞”以及“round toward 0”，接着再分别用 round()函数进行赋值，round()会调用 test()函数进行754-OF 异常的检测。

运行结果验证了前面的描述，在指数没有直接超出最大指数值的情形下；是否引发754-OF 异常的确受舍入模式的影响，在“round to nearest”下，无论是正值还是负值，舍入结果均引发754-OF 异常，并分别返回“±∞”；而在“round toward 0”下，两个值的舍入结果都没有引发754-OF 异常，返回的分别是正/负绝对值最大的值。

当然，15-03只是反映运算结果的指数值没有直接超过最大指数值的情形，要是运算结果的指数值已经直接超过对应格式的最大指数值，则无论是什么舍入模式处理器都会引发754-OF 异常，然后返回对应的结果（详见上面的表格）。

以上是出现754-OF异常后处理器不进入陷阱处理过程而直接返回结果的情况，如果处理器进入陷阱处理器过程又会发生什么事呢？IEEE 754要求处理器在进入陷阱之前调整导致上溢的结果，统一把指数域的值减去 α，α 的计算公式如下：

$$\alpha = 3 \times 2^{n-2}$$

其中，n 是指数域的长度，对于 single、double、double ext（80位）来说，调整值 α 分别为：192、1536、24576。例如，当 single 运算结果是：

0 11111110 11111111111111111111 111

在“round to nearest”模式下，上述结果导致754-OF 异常。如果处理器进入陷阱处理过程而不是直接返回结果，则处理器应该在陷阱处理过程之前把上面的值舍入、调整为：

0 00111111 00000000000000000000000

因为原来导致上溢的结果经过舍入处理是2^{128}，把指数值减去192之后变成2^{-64}，对应的 single 浮点实数就是上面给出的值。

我们稍后再给出 IEEE 754这样规定的原因。P6处理器由于把所有浮点实数运算都以 double ext 格式进行，虽然我们可以通过修改 FPU 控制寄存器让处理器以 single、double 格式进行运算，但这种修改只作用于有效数字，对于指数部分的值无效，处理器仍然使用 double ext 格式长达15位长度的指数域参与运算。只有当使用15位指数域也无法存放结果的指数值从而导致上溢、并且 FPU 控制寄存器的 OM 位是“0” 时（在引发754-OF 异常后处理器立即进入16号系统异常处理过程），P6处理器才会调整导致上溢的结果的指数域（减去24576）。而且该调整操作仅限于寄存器操作数，如果目标操作数是内存操作数，由于此时 P6处理器并不会把任何数值写入内存操作数，因此实际上也就没有类似的调整操作。

- **Underflow** （以下用“754-UF”表示）

当运算结果的绝对值太小并且造成一定的精度损失时，系统应该引发754-UF 异常。绝对值怎样才算“太小”呢？IEEE 754给出的界限是$2^{E_{MIN}}$，即指数域的值等于“1”而有效数字全部为“0”的值，以 single 格式为例，这个值就是：

0 00000001 00000000000000000000000

我们知道，这正是 single 格式中的最小规格化正值。绝对值小于这个最小规格化正值就算是“太小”。系统可以选择两种方式去判定是否满足“绝对值太小”这个条件：

1）以舍入之后的数值为准（after rounding）

2）以舍入之前的数值为准（before rounding）

仍然以 single 格式为例，如果某一次的运算结果是：

$(1.11111111111111111111111)_2 \times 2^{-127}$

而 single 的最小规格化正值是：

$(1.00000000000000000000000)_2 \times 2^{-126}$

按照方式1），该结果不满足“绝对值太小”的条件，因为进行舍入处理（假定是“round to nearest”模式）之后，该数值刚好等于最小规格化正值；而按照方式2），则该结果满足“绝对值太小”的条件，因为在舍入处理之前，该值的的确确小于最小规格化正值。P6处理器采用方式1）进行判定。

当处理器判断结果值“绝对值太小”后，它会自动对数值进行非规格化，使数值以非规格化的形式保存，例如某个 single 运算结果经过舍入后是：

$(1.11100000000000000000000)_2\times2^{-128}$

显然，该值小于最小规格化正值2^{-126}，于是处理器会把它非规格化为：

$(0.01111000000000000000000)_2\times2^{-126}$

这个非规格化 single 值的二进制形式就是：

```
0 00000000 01111000000000000000000
```

非规格化值的指数域全部为“0”。

非规格化处理虽然通过移动小数点达到了表示“绝对值太小”的值的目的，但由于受到有效数字位数的限制可能会导致精度损失。例如：

```
int i = 0x00FFFFFF;
float f = *(float*)(&i);
f = f / 4.0;
```

我们看看计算(f/4.0)的过程。首先，f 的二进制值为：

$(1.11111111111111111111111)_2\times2^{-126}$

于是(f/4.0)就等于：

$(1.11111111111111111111111)_2\times2^{-128}$

显然，该结果小于(2^{-126})，处理器必须进行非规格化处理，处理之后的值就是（假定处于“round to nearest”模式下）：

$(0.10000000000000000000000)_2\times2^{-126}$

但精确的结果本来应该是：

$(0.01111111111111111111111)_2\times2^{-126}$

可见，由于被逼再次进行舍入操作，非规格化处理的确引起了精度损失。

根据 IEEE 754标准，系统同样可以选择两种方式去判定是否“存在精度损失”：

1）仅仅由于非规格化而造成的精度损失，这意味着最后结果与假定指数值没有限制而得出的结果不一致

2）广义上的结果不精确，这意味着最后结果与假定指数值和有效数字都没有限制而得出的结果不一致

这两种方式的区别是前者仅仅把非规格化造成的误差看作是精度损失，后者则把舍入造成的误差也看作是精度损失。例如，某次运算结果是：

$(1.00000000000000000000001)_2\times2^{-127}$

现在要把该值赋给 float 变量，由于 single 浮点实数只有24位有效数字，因此，非规格化的最后结果为（假设处于“round to nearest”舍入模式）：

$(0.10000000000000000000000)_2\times2^{-126}$

假定 single 格式根本没有任何对指数值的限制，那么我们完全可以把运算结果写成：

$(1.00000000000000000000000)_2\times2^{-127}$

显然：

$(0.10000000000000000000000)_2\times2^{-126} = (1.00000000000000000000000)_2\times2^{-127}$

因此如果以方式1）作判定，则本次操作不存在精度损失。

假定 single 格式没有任何对指数值以及有效数字的限制，那么运算结果应该就是$(1.00000000000000000000001)_2\times2^{-127}$，很清楚地：

$(1.00000000000000000000001)_2\times2^{-127} \neq (0.10000000000000000000000)_2\times2^{-126}$

所以如果以方式2）作判定，本次操作的确存在精度损失。

P6处理器采用方式2）作为判定标准。

IEEE 754明确指出，系统判断出结果“绝对值太小”之后，有两种选择：

1）如果系统实现了754-UF 对应的陷阱处理并且启用了该陷阱处理，则系统应该马上引发754-UF 异常并进入对应的陷阱处理过程，并且在陷阱处理接管之前调整导致下溢的结果，统一往该结果的指数域加上 α， α 的计算公式同上面754-OF 异常的一样。

2）如果系统没有实现754-UF 对应的陷阱处理又或者该陷阱处理被禁止、被屏蔽，则系统必须在非规格化处理之后进一步判断本次操作是否存在精度损失，如果系统的判定是没有精度损失，则系统不引发754-UF 异常，返回非规格化值后正常结束本次操作；如果系统的判定是存在精度损失，则系统应该引发754-UF 异常。[7]

同样地，P6处理器由于把所有浮点实数运算都以 double ext 格式进行，处理器使用15位长度的指数域参与运算。因此，只有当15位的指数域也无法存放结果的指数值从而导致下溢、并且 FPU 控制寄存器的 UM 位是“0” 时（在引发754-OF 异常后处理器立即进入16号系统异常处理过程），P6处理器才会调整导致下溢的结果的指数域（加上24576）。而且该调整操作仅限于寄存器操作数，如果目标操作数是内存操作数，由于此时 P6处理器并不会把任何数值写入内存操作数，因此实际上也就没有类似的调整操作。

我们通过15-04观察一下 P6处理器判定“绝对值太小”以及“存在精度损失”的标准：

```
C01       /* Code 15-04, file name: 1504.c
C02       #include<stdio.h>
C03       #include<fenv.h>
C04       #pragma STDC FENV_ACCESS ON
C05       long long valNotUF = 0x380FFFFFFC000000;
C06       long long valUF = 0x3800000010000000;
C07       float f;
C08
C09       void test()
C10       {
C11          int fp_except = fetestexcept(FE_UNDERFLOW);
C12          if (fp_except & FE_UNDERFLOW)
C13          {
C14             printf("FPU raised 754-UF exception\n");
C15             feclearexcept(FE_UNDERFLOW);
C16          }
C17          else
C18             printf("FPU did not raise 754-UF exception\n");
C19       }
```

```
C20
C21       int main()
C22       {
C23           feclearexcept(FE_ALL_EXCEPT);
C24           f = *(double*)(&valNotUF);
C25           printf("f = %2.1E\t", f);
C26           test();
C27           printf("f/4.0 = %2.1E\t", f/4.0);
C28           test();
C29           f = *(double*)(&valUF);
C30           printf("f = %2.1E\t", f);
C31           test();
C32       }
```

编译、运行15-04：

```
$gcc  -std=c99  1504.c  -lm
$./a.out
f = 1.2E-38      FPU did not raise 754-UF exception
f/4.0 = 2.9E-39  FPU did not raise 754-UF exception
f = 5.9E-39      FPU raised 754-UF exception
```

对15-04及其运行结果的解释如下：

首先，我们把 double 类型的 valNotUF 值：

$(1.11111111111111111111111)_2 \times 2^{-127}$

赋给 float 变量 f，以此观察 P6处理器判定是否“绝对值太小”的时刻是数值舍入前还是数值舍入后。如果是舍入之前进行判定，那上面的值毫无疑问属于“绝对值太小”的情况，随后由于 P6处理器必须对它进行舍入处理（因为它有25位有效数字），这样的话，精度损失便在所难免，用 fetestexcept()函数应该可以检测出处理器引发了754-UF 异常。但运行结果的第一行显示出 P6处理器根本没有引发754-UF 异常，所以从反面证实了处理器是根据舍入之后的数值进行判定是否“绝对值太小”。

然后，我们分别用两个“绝对值太小”的值测试处理器什么时候才会引发754-UF 异常。第1个值是(f/4.0)，第2个值是：

$(1.00000000000000000000001)_2 \times 2^{-127}$

结果，(f/4.0)没有导致754-UF 异常，因为处理器进行非规格化后得到的值和“绝对值太小”的原值完全相等；而第2个值则导致处理器引发754-UF 异常，因为非规格化之后的值与“绝对值太小”的原值不相等而被 P6处理器判定为“存在精度损失”。这恰恰证明了 P6处理器是以“广义上的结果不精确”作为判定是否“存在精度损失”的依据，因为如果处理器是按照非规格化是否引起精度损失作为判定依据的话，则上面的值是不应该导致处理器引发754-UF异常的。这里需要说明一下 IEEE 754规定在754-OF/754-UF 的陷阱处理过程中处理器要在其它代码接管之前通过减去/加上 α 来调整结果指数域的

目的。

事实上，在计算类似下面的数值时：

$$Q = \frac{(a_1 + b_1)\cdot(a_2 + b_2)\cdot(a_3 + b_3)\cdot(...)\cdot(a_N + b_N)}{(c_1 + d_1)\cdot(c_2 + d_2)\cdot(c_3 + d_3)\cdot(...)\cdot(c_M + d_M)}$$

虽然分子、分母会在计算过程中可能会频繁出现上溢或下溢，但最后结果往往都是正常的数值。这时，处理器在754-OF/754-UF 异常陷阱处理程序接管之前利用 α 来调整指数值就很有必要了：一方面计算过程可以继续进行下去而不必因为异常而中断，另一方面只要陷阱处理程序正确地统计出发生上溢/下溢的次数就可以在最后正确还原出真正的结果。譬如，假设计算期间曾经发生 X 次上溢、Y 次下溢，则只要往最后结果的指数域加上修正值“α*(X-Y)”就可以得到真正的计算结果。

- **Inexact** （以下用“754-IE”表示）

如果满足：

条件1）进行舍入处理之后得到的结果和原来的值不相等

或者

条件2）发生754-OF 异常之后处理器没有进入对应的陷阱处理过程

则处理器应该引发754-IE 异常，然后要么进入对应754-IE 异常的陷阱处理过程、要么返回舍入处理后的结果（由条件1导致的异常）或者适当的值（由条件2导致的异常，具体值详见讲述“上溢”时给出的表格）。

754-IE 异常实在是“家常便饭”，人们通常会忽略它，因为在浮点实数运算中，“不精确的结果”是意料中事，例如，把 double 值赋给 float 变量，必须进行舍入，如果24位有效数字无法准确表示原来的值，则误差是不可避免的；进行算术运算，例如(1.0/10.0)，其结果0.1无法用有限位二进制有效数字进行准确表示，这个时候舍入误差又出来了；出现754-OF 异常之后处理器如果不进入陷阱处理过程，就要返回（前面给出的表格里）对应的值，存在的误差往往更大；同样，出现了“绝对值太小”的值之后处理器没有进入754-UF 异常对应的陷阱处理过程，然后由于非规格化造成的精度损失而引发754-UF 异常这种情形也符合754-IE 异常的引发条件。总之，754-IE 异常是我们经常碰到的、最普遍的异常。

对于 P6处理器，如果异常屏蔽位 OM/UM 置“1”，即754-OF/754-UF 异常发生后处理器不会进入陷阱处理（16号系统异常）过程，那么754-IE 异常总是伴随754-OF/754-UF 异常同步产生的，因为这两者均意味着“结果不准确”。

下面，我们通过15-05观察一下754-IE 异常：

```
C01         /* Code 15-05, file name: 1505.c
C02         #include<stdio.h>
C03         #include<fenv.h>
C04         #pragma STDC FENV_ACCESS ON
C05         long long valUF = 0x3800000010000000;
C06         float f1, f2;
```

```
C07
C08         void test()
C09         {
C10             int fp_except = fetestexcept(FE_INEXACT);
C11             if (fp_except & FE_INEXACT)
C12             {
C13                 printf("FPU raised 754-IE exception\n");
C14                 feclearexcept(FE_INEXACT);
C15             }
C16             else
C17                 printf("FPU did not raise 754-IE exception\n");
C18         }
C19
C20         int main()
C21         {
C22             feclearexcept(FE_ALL_EXCEPT);
C23             printf("1.0 / 10.0\t");
C24             f1 = 1.0;
C25             f2 = 10.0;
C26             f1 = f1 / f2;                    8
C27             test();
C28             printf("1E38 / 1E-38\t");
C29             f1 = 1E38;
C30             f2 = 1E-38;
C31             f1 = f1 / f2;
C32             test();
C33             printf("Denormalization\t");
C34             f1 = *(double*)(&valUF);
C35             test();
C36         }
```

编译、运行15-05：

```
$gcc  -std=c99  1505.c  -lm
$./a.out
1.0 / 10.0          FPU raised 754-IE exception
1E38 / 1E-38        FPU raised 754-IE exception
Denormalization     FPU raised 754-IE exception
```

上面，第1次754-IE 异常是由于(1.0/10.0)的运算结果无法用目标格式准确表示、从而必须进行舍入处理而导致误差所引发的；第2次754-IE 异常是由于754-OF 异常没

有进入陷阱处理过程、直接返回“+∞”而引发的；第3次754-IE异常是由于非规格化处理存在精度损失而引发754-UF异常的“并发症”。

熟悉了IEEE 754规定的上述5种异常之后，我们再分析一下P6处理器的若干特点。首先就是前面已经反复提到的，P6处理器默认以80位double ext格式进行所有的浮点实数运算操作，所以引发浮点异常的时机通常会“滞后”少许，例如某次运算结果的绝对值可能已经远远超过double格式最大的正值，但由于P6处理器用80位的浮点寄存器进行运算、保存该结果，因此这个时候处理器并不会引发754-OF异常。直到要把结果进行舍入处理然后复制给内存中的某个double变量时，P6处理器才会基于“数值太大、目标操作数无法准确存放”而引发754-OF异常。

第二点就是P6处理器在754-OF/754-UF异常引发后如果是直接进入陷阱处理（即16号系统异常）过程的，那么在陷阱处理过程接手之前，处理器对两种不同的目标操作数类型分别有不同的处理。如果是浮点寄存器作为目标操作数，则处理器会依照IEEE 754的指示自动对指数域通过“减去/加上α值”进行调整，并且把调整后的值放入目标操作数；如果是内存操作数作为目标操作数，那么处理器既不会自动用α值调整导致754-OF/754-UF的结果的指数域，也不会把任何数值写入目标操作数。由于在内部都是用80位浮点实数格式进行运算操作，尽管可以设置FPU控制寄存器的PC域使处理器以兼容32/64位精度模式工作，但该设置仅仅影响有效数字的位数，处理器在任何时候依然都是使用15位的指数域，因此P6处理器的α值只有一个（24576）而不是三个。

第三点就是P6处理器进入16号系统异常处理过程的时机。当引发浮点异常后，并且对应的异常屏蔽位是“0”时，处理器无疑是会进入陷阱处理过程。但具体什么时候进入却受下一条指令的影响。当处理器随后执行WAIT/FWAIT指令、或者任何一条具有“waiting”性质的浮点指令时系统才会进入16号系统异常处理过程。因此，如果我们在整数指令、浮点指令混合的指令序列中要求处理器必须在引发浮点异常的指令的下一条指令执行之前进入16号系统异常，则通常要在可能引发异常的指令后面加插一条WAIT/FWAIT指令。当然，无论是否会进入陷阱处理过程，处理器在引发异常的指令执行完毕之前已经对FPU状态寄存器对应的异常标志位设置完毕。所以通过检查FPU状态寄存器的相关异常标志位可以立刻知道是否引发了浮点异常。

另外，IEEE 754只是规定实现系统必须提供少数几种最基本的浮点实数运算操作：加、减、乘、除、求余、算术平方根、大小比较以及各种转换，而在实际的开发中，我们往往需要使用更多的数学运算，例如三角函数、对数等。在如何提供对这些运算的支持这个问题上，各种处理器的做法有不少差异。例如基于RISC的MIPS32/64处理器基本上就没有提供支持其它各种数学运算的指令，对于该系统来说，如果要计算复杂的数学函数，必须通过一定的算法把函数计算过程分解为最基本的浮点实数运算操作，通常这些函数都以各种数学运算库的形式由软件厂商提供。

相比之下，P6系列处理器则额外提供了不少指令直接支持某些数学函数：

F2XM1	计算 2^X-1
FCOS	计算 cos(X)
FSIN	计算 sin(X)
FPATAN	计算 arctg(X/Y)
FYL2X	计算 $Y*\log_2 X$

FSCALE　　计算 $Y*2^X$

...　　...

因此，某些基于 P6处理器开发的计算函数可能只需要相对较少的指令就能完成同一个计算任务。不过，由于复杂浮点运算指令的执行周期相当长，而且执行效率还受到处理器其它方面设计的影响，所以，完全没有理由仅仅因为 P6处理器拥有额外的数学运算指令就认为它在数学计算方面占有优势。

作为普通 C 程序员，我们一般不需要自己去开发全新的计算函数，因为 C99的数学函数库已经为我们提供了相当丰富的选择，C 程序员只要有可能都应该使用这些标准库函数而不是自己去编写相同功能的函数。

IEEE 754标准在附录中建议各个实现平台向用户额外提供10个函数，因为使用这些函数有助于增强计算程序在不同系统之间的可移植性。系统可以选择单纯由硬件实现这些函数，也可以选择通过软件配合硬件来实现。现实中，一方面由于指令数目的限制，大多数处理器不可能全部通过指令的形式实现这些函数，另一方面，使用高级语言接口将会有更好的可移植性。

这10个函数如下：

1）copysign(x, y)

仅仅把 x 的符号换成 y 的符号而保持 x 的其它位不变所得到的值就是该函数的返回结果。

2）(-x)

仅仅把 x 的符号位取反而保持 x 的其它位不变所得到的值就是该函数的返回结果。

3）scalb(y, *N*)

返回$(y\times 2^N)$。

4）$\log_b(x)$

返回 x 的指数，注意，不是指数域的值而是真正的指数。

5）nextafter(x, y)

返回从 x 到 y 方向上最接近 x 的下一个值。

6）finite(x)

如果满足：$-\infty < x < +\infty$函数返回“真”否则返回“假”。

7）isnan(x)

如果 x 是 NaN 函数返回“真”否则返回“假”。

8）x<>y

如果 x 和 y 的关系是：x>y 或者 x<y，那么上面的关系式返回“真”否则返回“假”。

9）unordered(x, y)

如果 x 和 y 的关系是不能确定函数返回“真”否则返回“假”

10）class(x)

该函数返回的结果应该指出 x 属于以下类型中的哪一种：SNaN，QNaN，$-\infty$，$+\infty$，-0，+0，规格化负值，规格化正值，非规格化负值，非规格化正值。而且，class(x)永远不引发异常，即使 x 是 SNaN。

最后，IEEE 754还补充说明：对于类似“把 x 的值复制给 y 而 x 是 SNaN”以及在上面第1、2、6、7个函数中 x 是 SNaN 的情形，各系统可以自行决定是否引发754-IO

异常。

在结束本节之前，我们详细地探讨一下浮点实数的大小比较问题。敏锐的读者可能已经发现 IEEE 754浮点实数的格式安排有某种巧妙之处，浮点系统可以部分地借助整数大小比较的逻辑来完成浮点实数的比较操作。对于两个相同格式的浮点实数 x、y，如果除去属于 NaN 的二进制值，那么比较大小的逻辑将是非常简单的：

- x 和 y 的符号位不相同（一个是“0”另一个是“1”）

不失一般性，假设 x 的符号位是“0”，则除非 x 和 y 的其它所有位都是“0”，否则 x 肯定大于 y，因为 x 是非负数、y 是非正数。当 x 和 y 的其它所有位都是“0”时，x 就是“+0”、y 就是“-0”，IEEE 754规定，在大小比较运算中，(+0)和(-0)相等。上面 x 和 y 的地位是对称的，如果 x 的符号位是“1”而 y 的符号位是“0”则结论刚好相反，道理是一样的。

- x 和 y 的符号位相同（同时是“0”或同时是“1”）

这时只须比较 x 和 y 的其它所有位的大小即可，借助比较无符号整数大小的逻辑，系统得出(x 的余下部分)和(y 的余下部分)的关系，如果 x、y 的符号是“0”，则 x 和 y 的大小关系如下：

(x 的余下部分) > (y 的余下部分)，则 x > y ；

(x 的余下部分) = (y 的余下部分)，则 x = y ；

(x 的余下部分) < (y 的余下部分)，则 x < y ；

如果 x、y 的符号是“1”，则 x 和 y 的大小关系如下：

(x 的余下部分) > (y 的余下部分)，则 x < y ；

(x 的余下部分) = (y 的余下部分)，则 x = y ；

(x 的余下部分) < (y 的余下部分)，则 x > y ；

由上可知，在不涉及 NaN 的情况下，浮点实数大小比较运算的速度几乎和整数大小比较运算相同，处理器根据浮点实数的原始二进制值几乎马上就可以得到比较结果。很多不熟悉 IEEE 754标准的人总是想当然地认为既然浮点实数是“不精确”的因此其大小比较运算也是不可靠的、存在一定误差的。最流行的一个讹传是，某些人认为一个浮点实数变量 x 和“0.0”进行比较以确定是否相等需要借助一个绝对值很小的数，例如1E-10，然后这样写：

```
if abs(x - 0.0) < 1E-10
   f();         /* 这里是 x 等于0.0的代码部分 */
else
   g();         /* 这里是 x 不等于0.0的代码部分 */
```

其实，稍微熟悉 IEEE 754浮点实数格式的读者朋友仅仅依靠想象力就完全可以质疑这种做法的依据。因为0.0的二进制值只有两个（以 single 格式为例）：

```
0 00000000 00000000000000000000000          /* +0 */
1 00000000 00000000000000000000000          /* -0 */
```

试想，和这两个二进制值进行比较，要得出是否相等的结论是多么的简单、多么的精确，怎会存在“比较误差”呢？通过前面列出的比较运算逻辑，处理器轻而易举就能确定 x 和0的关系究竟是大于、小于或相等。

注意的是，上面的讨论同样适用于“±∞”。总之，对于除 NaN 之外的所有浮点实数

值，以下关系永远成立：

-∞ < 负数 < ±0 < 正数 < +∞

事实上，由于“∞”的指数域的位全部是“1”所以根据上面的算法逻辑完全保证了“+∞”大于所有其它值而“-∞”则小于所有其它值。

问题的关键其实集中在NaN上面。IEEE 754规定，每一个浮点实数值与另一个浮点实数值有且只能有以下4种关系的其中一种：大于、小于、等于、不能确定（unordered）。大于、小于、等于这3种关系无疑是用在包括“±∞”在内的非NaN浮点实数值身上，而一旦两个值之中有任何一个是NaN，不管是SNaN还是QNaN，它们之间的关系就是“不能确定”。

我们只须注意两点：

- 当关系是不能确定时，各种关系运算表达式返回的结果

这个很容易搞清楚，C语言一共才有：

>（大于）、<（小于）、>=（大于或相等）、<=（小于或相等）、==（相等）、!=（不相等）6种关系运算符，当两个浮点实数值的关系是“不能确定”时，除“!=”关系表达式返回“1”之外其它所有关系表达式都是返回“0”。

这个规定应该不难理解，因为NaN根本不是有意义的浮点实数值，所以NaN和其它有意义的浮点实数值之间以及两个NaN之间不存在谁大谁小的问题，更不可能相等，唯一合理的关系就是不相等。　　于是，一些看上去有点古怪的事出现了，例如下面的C代码：

```
float x;
...
if (x != x)              /* x难道还能不等于它自己吗？！*/
    ...                  /* 这部分代码有机会被执行吗？！*/
else
    ...
```

对于整数类型、指针等基于整数运算的变量而言，“(x != x)”这样的写法确实是多余的，但对于拥有NaN的浮点实数类型，(x != x)却是测试x是否NaN的最简单有效的方法，因为对于浮点实数来说，当且仅当x是NaN时关系表达式(x != x)才会返回“1”。同样道理，用表达式 (x == x)也可以测试出x是否NaN，当且仅当x是NaN时表达式(x == x)才会返回“0”。

- 当参与比较运算的任何一方是NaN时，系统是否会引发异常

这个问题的答案取决于系统如何进行比较运算。IEEE 754指出，系统可以通过两类方式进行比较运算：

1）输入的是两个参与比较的操作数，运算的结果是返回4种关系的其中之一。对于这种方式进行的比较运算，系统应该按“常规”办事，那就是，当任何一方是SNaN时，系统应该引发754-IO异常（之后要么进入陷阱处理过程，要么直接返回“不能确定”的比较结果）；当任何一方是NaN（但没有SNaN）时系统直接返回“不能确定”的比较结果。P6处理器的FUCOM指令就是按照这种方式工作的。

2）输入的是两个参与比较的操作数以及4种关系中的其中之一，系统运算之后返回的结果是“真”或“假”。这种方式其实就是让系统对预先给出的关系进行判断，所以结

果只有两个，原先给出的关系要么成立要么不成立。对此种工作方式，IEEE 754规定，只有当预先给出的关系是：“>”，“<”，“>=”，“<=”而任何一方是NaN时[9]，系统才引发754-IO异常（之后要么进入陷阱处理过程要么直接返回“假”）。这也是对QNaN进行操作却引发754-IO异常的两种情况之一。

P6处理器以第1类方式进行大小比较运算，而且FPU控制寄存器的IM位默认值是“1”，所以我们在C程序里对浮点实数进行比较运算虽然会引发754-IO异常但不会导致处理器进入16号系统异常处理过程。

回到前面那个浮点实数变量与0.0比较是否相等的例子，我们完全可以写：

```
if (x == 0.0)
    f();          /* 这里是x等于0.0的代码部分 */
else
    g();          /* 这里是x不等于0.0的代码部分 */
```

因为如果x不是NaN，那么x和0.0的关系完全可以（精确地）描述，要么相等要么不相等；如果x是SNaN，那么按照第1类方式工作的系统就会引发754-IO异常，之后要么进入陷阱处理过程要么直接返回“不能确定”的结果，这意味着表达式(x == 0.0)返回“0”，按照第2类方式工作的系统不会引发754-IO异常而直接返回相应的判断结果（根据具体指令而定），同理，表达式(x == 0.0)肯定返回“0”；如果x是QNaN，则两种系统都不会引发754-IO异常，第1类系统直接返回“不能确定”的结果，即表达式(x == 0.0)返回“0”，第2类系统也是直接返回相应的判断结果（根据具体指令而定），表达式(x == 0.0)仍然是返回“0”。

我们把上面的代码编译成汇编指令，由此观察一下作为第一类系统的P6处理器进行浮点实数大小比较运算的具体细节：

```
A01              flds    x
A02              fldz
A03              fxch    %st(1)
A04              fucompp
A05              fnstsw  %ax
A06              sahf
A07              jp      .L5
A08              je      .L3
A09         .L5:
A10              jmp     .L2
A11         .L3:
A12              call    f
A13              jmp     .L4
A14         .L2:
A15              call    g
A16         .L4:
A17              ...
```

A01把 x 值复制（在复制的同时完成相应格式的扩展）到80位浮点寄存器 st(0)，然后 A02的 FLDZ 指令把0.0装入浮点寄存器，这时我们不妨假设 x 和0.0的位置如图15-01a。接着 A03通过 FXCH 指令把两个值的位置对调，如图15-01b。关键是 A04的 FUCOMMP 指令，它对寄存器 st(0)和 st(1)的值进行比较，然后把 st(0)和 st(1)的值弹出浮点寄存器栈弃置掉，如图15-01c。

比较的结果存放在 FPU 状态寄存器的 C0、C2、C3位(即第8、10、14位)，如图15-01d，之后 A05把 FPU 状态寄存器的值保存到 ax 中，A06再通过 SAHF 指令把 ah 寄存器的第0、2、4、6、7位复制到 eflags 标志寄存器的第0、2、4、6、7位，于是 eflags 寄存器的 CF、PF、ZF 位分别就是表示比较结果的 C0、C2、C3的值，如图15-01e。

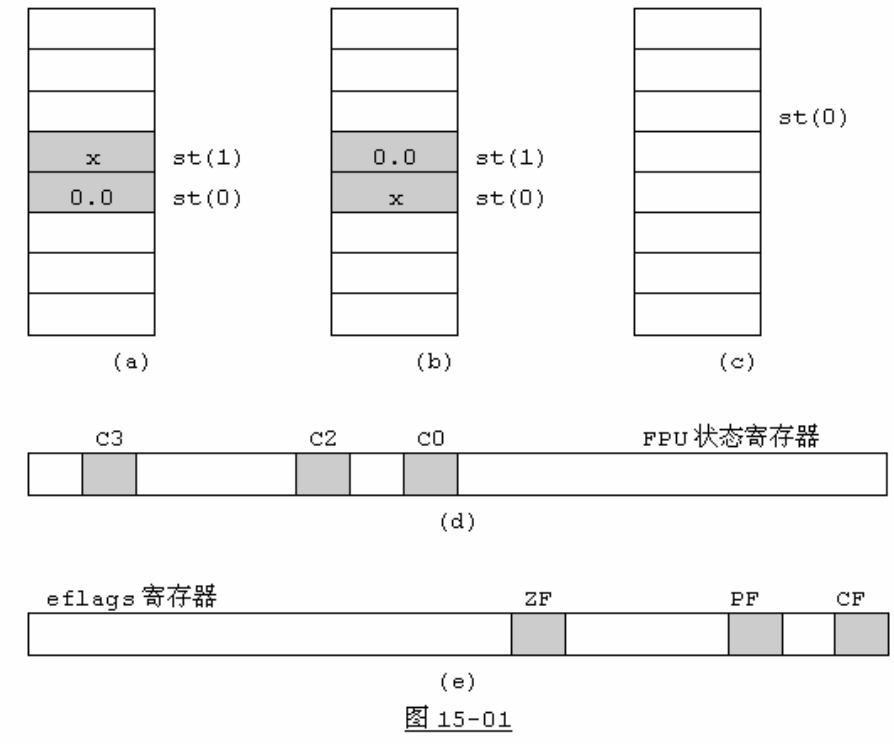


图 15-01

FUCOMPP 指令对 C0、C2、C3的设置情况如下：

	C3	C2	C0
st(0) > st(1)	0	0	0
st(0) < st(1)	0	0	1
st(0) = st(1)	1	0	0
unoredered	1	1	1

我们的 C 代码是要比较 x 是否和0.0相等，从汇编代码已经知道，当 FUCOMPP 指令开始执行时 x 存放在 st(0)，0.0存放在 st(1)，所以，如果 C3-C2-C0的组合值是“1-0-0”则意味着 x 等于0.0，否则 x 就不等于0.0。汇编代码 A07首先把“1-1-1”这种情况筛选出来，因为4种关系中只有“unordered”的 C2是“1”。执行完 SAHF 指

令之后 eflags 寄存器的 PF 位就是 C2的值，因此 A07用“JP”指令跳到“.L5”然后再跳到“.L2”执行函数 g()。A08通过“JE”指令检查 ZF 位是否置“1”而决定是否跳转，ZF 就是 C3，在剩下的3种关系中只有相等关系的 C3是“1”，于是 JE 指令可以把相等关系筛选出来，如果 ZF 是“1”就跳到“.L3”执行函数 f()然后跳到“.L4”，否则按顺序执行 A10而跳到“.L2”从而调用函数 g()。

和0.0比较绝对是精确的，但和其它浮点数值进行比较又会如何呢？这时候，我们必须注意舍入误差对比较结果的影响。即使是对同一个浮点实数，一旦经过舍入处理，比较结果可能就会不一样。先看这个：

```
if (0.1 == 0.1)
    f();
else
    g();
```

毫无疑问，函数 f()会被执行。因为虽然0.1不能用有限位二进制有效数字去表示，但由于上面“==”操作符两边的0.1在装入浮点寄存器时都进行了相同的舍入处理，所以尽管浮点寄存器里的值都不是精确地等于0.1，但两者比较的结果依然是相等，关系表达式返回“1”。再看这个：

```
float f = 0.1;
if (f == 0.1)
    f();
else
    g();
```

上面代码的执行结果是函数 g()被调用，也就是说，表达式(f == 0.1)返回“0”。因为常数0.1的类型是 double，把 double 值赋给 float 变量时如果进行了舍入处理那么 float 变量所得到的值就不再和原先的 double 值相等。由于把0.1转换为 double 值之后它的二进制有效数字超过24位，因此在赋给变量 f 时必须进行舍入处理。这样的话，f 的值不可能再等于 double 值0.1，关系表达式返回“0”是理所当然的。

由于和浮点实常数进行是否相等的比较运算因舍入误差的问题而引来各种古怪的现象，因此应该尽量避免使用“!=”和“==”操作符，以下是一个典型例子：

```
for (double d = 0.0; d != 0.3; d += 0.1)
    printf("%f\n", d);
```

写这段代码的人显然期望 for 语句可以输出3个值：

```
0.000000
0.100000
0.200000
```

然而，实际上代码的运行结果却是：

```
0.000000
0.100000
0.200000
```

```
0.300000
0.400000
...
```

for 语句陷入了死循环出不来，printf 语句不断打印新的 d 值，而且，令人迷惑不解的是，从打印结果看，变量 d 明明曾经取值0.3，但就是没有结束循环，反而一直打印下去。

让我们把代码稍作修改——还记得 double 值需要17位十进制有效数字才能准确区分吗？我们把 double 常数0.3以及变量 d 的值用17位有效数字打印出来看看：

```
printf("%17.17f\n******************\n", 0.3);
for (double d = 0.0; d != 0.3; d += 0.1)
    printf("%17.17f\n", d);
```

这一次的运行结果如下：

```
0.29999999999999999
******************
0.00000000000000000
0.10000000000000001
0.20000000000000001
0.30000000000000004
0.40000000000000002
...
```

采用17位十进制有效数字的打印结果清楚地表明一个事实，变量 d 的值从来就没有和 double 常数0.3相等过。于是 for 语句陷入死循环也就不奇怪了。我们深入地观察一下整个运算过程，首先，double 常数0.1的二进制形式为：

$$(1.10011001100110011001100110011001100110011001100110011010)_2\times 2^{-4}$$

第1次迭代时，系统打印出 d 的值之后把0.1和 d 相加；第2次迭代时，又要把0.1加给 d，这次加法的结果是：

$$(1.10011001100110011001100110011001100110011001100110011010)_2\times 2^{-3}$$

第3次迭代时，d 再次加上0.1，结果是：

$$(1.00110011001100110011001100110011001100110011001100110100)_2\times 2^{-2}$$

而 double 常数0.3的二进制形式为：

$$(1.00110011001100110011001100110011001100110011001100110011)_2\times 2^{-2}$$

可以看到，两个过程都导致了舍入误差：

- 浮点实常数转换为只有有限位有效数字的二进制形式
- 加法运算

在这种情形下，使用“!=”和“==”操作符是行不通的。对于上面的代码，如果要确保在适当的时候能够终止循环，我们可以改用“<=”：

```
for (double d = 0.0; d <= 0.3; d += 0.1)
    printf("%f\n", d);
```

使用“<=”、“<”、“>”、“>=”等操作符之后，我们实际上是对某个范围内的数值进

行操作，所以代码不再因为和某一个数值点比较是否相等时存在舍入误差而陷入死循环。

可能有人仍然不满意，由于使用“<=”、“<”、“>”、“>=”操作符有可能存在迭代次数比精确预期的多一次或者少一次的问题（这发生在迭代子刚好以很微小的幅度小于或大于终止值时）。解决方案很简单：在需要能够精确符合预期迭代次数的场合不要使用浮点数。的确，浮点数从被设计出来的时候起就不是作为迭代子使用的，在需要精确符合预期迭代次数的时候，我们没有理由不使用整数类型。至少，到目前为止还没有出现一个项目是因为无法使用浮点数作为精确迭代子而失败的。

*　　*　　*　　*　　*　　*

[1501]

IEEE 754定义求余运算为：

p *REM* q = p − q × n　　/* 其中 q ≠ 0， n 是最接近(p/q)的整数 */

[1502]

注意，“∞”包括”+∞”和”-∞”，NaN 包括 SNaN 和 QNaN，以下同。

[1503]

注意，“-lm”选项不能缺少，因为按照 Unix 系统的惯例，<math.h>对应的标准 C 库函数不在 libc.*里面，而是被单独存放在 libm.*文件中，而默认情况下 ***gcc*** 不会与 libm.*进行链接，所以我们必须使用“-lm”来指示 ***gcc*** 到 libm.*寻找对应的数学库函数。

[1504]

当然，由于浮点常数0.0的类型是 double，因此 snan 的值即使作为 float 值参与加法运算按照 UAC 规则也要先提升为 double 类型，不过这已经无关紧要，因为 float 类型的 SNaN 提升为 double 类型之后仍然是 SNaN，我们的目的已经达到。

[1505]

除数的最外层那一对括号是不必要的，这里加上它们是为了防止粗心的读者漏掉除号“/”与“*”之间的空格从而导致编译器以为这里是一个注释（“/*”）的开始部分。

[1506]

只需修改3次，因为在一开始 P6处理器默认的舍入模式就是“round to nearest”。

[1507]

注意，这时引发754-UF 异常之后不会进入754-UF 对应的陷阱处理过程，因为正是由于没有 754-UF 异常对应的陷阱处理系统才走到了这一步，此刻系统能够采取的行动通常是设置对应的异常标志位。

[1508]

之所以这样写是因为要避免编译器优化代码，如果写成：

f = 1.0/10.0;

则编译器在编译阶段就会计算出结果然后直接赋给变量，下面的 C31也是同样道理。

[1509]

这里针对 C 语言作了某些简化：我假设该系统只有“=”、“!=”、“>”、“<”、“>=”、“<=”等6种预先给出的关系。那么，根据正文的论述，其中4种会引发异常而另外两种则不会。

16 复数类型

C99正式提供对纯虚数类型（imaginary type）以及复数类型（complex type）的支持：

```
float _Imaginary
double _Imaginary
long double _Imaginary
float _Complex
double _Complex
long double _Complex
```

“_Imaginary”、“_Complex”是新的关键字，纯虚数类型、复数类型和浮点实数类型合称浮点类型（floating type），每个浮点类型都有一个与其对应的实数类型，对于浮点实数类型，与其对应的实数类型就是它本身，例如与 float 对应的实数类型就是 float；对于纯虚数以及复数类型，与其对应的实数类型就是把“_Imaginary”或“_Complex”关键字去掉后得到的类型，例如与 double _Complex 对应的实数类型就是 double。

由于 C99规定纯虚数类型的实现是可选的（optional），因此大部分的 C 编译器（包括 *gcc*）并不支持纯虚数类型。其实这并不影响我们使用纯虚数，因为“实部为零”的复数就是纯虚数，只要编译器支持复数类型，我们也就完全可以使用纯虚数。鉴于此，下面的内容不再对纯虚数类型进行讨论。

为了统一复数类型的存储方式，C99规定：

“T _Complex”类型（T 只能是 float、double、long double 其中之一）的值在存储方式以及内存地址对齐要求上均和数组“T array[2]”相同，而且，array[0]存放复数的实部，array[1]存放复数的虚部。譬如：

```
float _Complex fc = 1.0F + I * 2.0F;
```

假设 fc 的地址是0x080490A0，那么在内存映像中 fc 的存放方式如图16-01：

图 16-01

从地址0x080490A0开始的4个字节存放 fc 的实部（float 值1.0），从地址0x080490A4开始的4个字节存放 fc 的虚部（float 值2.0）。

前面提到，纯虚数类型其实可以通过复数类型间接提供支持，如果我们想使用纯虚数，完全可以这样写：

```
double _Complex dc = I * 5.7;
```

编译器会判断出这是一个实部为“0”的复数，并且自动地帮我们添上一个适当类型的浮点值0.0。假设 dc 的地址是0x080490A0，那么从地址0x080490A0开始的8个字节

存放 double 值0.0，从地址0x080490A8开始的8个字节存放 double 值5.7。

另外，复数类型与复数类型、实数类型之间同样可以互相进行转换：

- 复数类型与复数类型之间的转换

这种转换实质上就是两对浮点实数之间的转换，例如：

```
double _Complex dc = 1.0 + I * 2.0;
float _Complex fc = dc;
```

意味着：

```
fc = (float)1.0 + I * (float)2.0
```

即是说，(T1 _Complex)值转换为 (T2 _Complex)值的操作其实是分别把前者的实部、虚部从类型 T1转换到类型 T2，从而构成一个(T2 _Complex)值。

- 复数类型与实数类型之间的转换

这种转换有连续两个步骤，先进行复数类型与和它对应的实数类型之间的转换（通过增加或者去掉虚部），然后再进行复数类型之间或者实数类型之间的转换。例如：

```
float f = 1.0;
double _Complex dc = f;
```

上面的转换分两步走，首先把实数类型转换为复数类型：

```
float _Complex temp = f;
/* temp = (float)1.0 + I * (float)0.0; */
```

然后把 temp 从 float _Complex 转换为 double _Complex:

```
double _Complex dc = temp;          /* dc = 1.0 + I * 0.0; */
```

又譬如：

```
double _Complex dc = 1.0 +  I * 2.0;
int i = dc;
```

这次的转换过程则是先把复数类型转换为与其对应的实数类型：

```
double temp = dc;                    /* temp = 1.0; */
```

再把 temp 从 double 转换为 int:

```
int i = temp;                        /* i = (int)1.0; */
```

注意，为了使大家容易理解，上面的例子把转换过程明显地分成两步，中间还引入了临时变量 temp，实际上，编译器生成的最终代码并不一定就是这样截然地分成两个阶段。作为 C 程序员，我们只需要清楚地知道转换后的结果是什么就可以了。

在使用复数类型的例子里，我们还使用了一个“I”，这个“I”从数学的角度讲无疑就是指虚数单位。但问题是怎么让编译器认识这个“I”呢？如果没有任何其它措施，编译器就会顺理成章地认为“I”是一个标识符，编译错误在所难免——代码没有声明变量“I”就使用了它！要了解这个“I”的由来，必须回顾一下历史。其实早在 C99出台之前很多编译器已经私底下提供对复数类型的支持，大家的思路基本都相同，那就是各自通过厂商自行定义的头文件<complex.h>定义虚数单位“I”，并且引入新的关键字“complex”，然后扩展编译器的功能，使其能够对复数类型提供各种支持（包括初始化、简单赋值、类型转换、四则运算等等），这样的话，包含了头文件<complex.h>之后，用户就可以非常自然地使用复数类型。

现在，C99正式支持复数类型，但如何使旧的代码仍然能够被新的 C99编译器正常编

译呢？一方面，C99引入了非常特别的新关键字“_Complex”，以区别所有私底下支持复数类型的编译器自己使用的关键字[1]。这样，凡是使用“_Complex”的都是符合 C99标准的新代码；另一方面，C99把一个全新的<complex.h>加入标准库，旧代码的兼容性以及虚数单位的定义均由该头文件提供支持。在标准头文件<complex.h>中，C99规定各实现平台必须定义3个宏[2]：

```
complex     _Complex_I     I
```

_Complex_I 和 I 是虚数单位，complex 是前标准时代各编译器使用得最广泛的扩展关键字，用来替换“_Complex_I”和“I”的常数的类型必须是 const float _Complex。我们可以看看 ***gcc*** 的<complex.h>的相关内容：

```
#ifndef  _COMPLEX_H
    ...
    #define  _COMPLEX_H
    #define  complex   _Complex
    #define  _Complex_I   (__extension__ 1.0iF)
    #define  I   _Complex_I
    ...
#endif
```

在<complex.h>的帮助下，旧的代码现在可以继续工作。例如：

```
#include<complex.h>
int main(){
    double complex x = 1.0 + I * 2.0;
    double complex y = 4.0 + I * 3.0;
    double complex z = x * y;
}
```

进行预处理之后，上面的旧代码就变成：

```
int main()
{
    double _Complex x = 1.0 + (__extension__ 1.0iF) * 2.0;
    double _Complex y = 4.0 + (__extension__ 1.0iF) * 3.0;
    double _Complex z = x * y;
}
```

基于 C99标准的 ***gcc*** 完全可以正常编译。虽然每家厂商对“_Complex_I”的具体定义形式可能不相同，上面的(__extension__ 1.0iF)仅仅是 ***gcc*** 自己的实现，但这完全不影响我们跨平台使用标准的复数类型[3]。只要包含标准头文件<complex.h>，我们就可以在所有符合 C99标准的平台以统一的方式使用复数类型。

大家都知道，C 语言的<math.h>标准库为我们提供了浮点实数类型的数学运算支持，在这个库里面有三角函数、指数函数、对数函数、统计函数等等常用的数学函数，用户可以直接调用，非常方便。C99既然增加了复数类型，自然也会提供复数类型的数学运算函

数库。C99在<complex.h>里面提供了一批用于复数运算的数学函数，当然，由于复数的数学性质，某些实数运算对复数来说没有意义，因此<complex.h>里面的数学函数比<math.h>的相对要少一些。也许，C 标准把提供更完备的数学函数库这个任务留给了各个函数库厂商，目前来说，更令人兴奋的是 C99的另一项新特性。

首先，和 C89不同的是，C99为每一个（实数的或复数的）数学函数都配备了三个版本。例如实数的正弦函数，在 C89的<math.h>中，只有一个原型：

```
double sin(double x);
```

而 C99的<math.h>里面则有三个：

```
float sinf(float x);
double sin(double x);
long double sinl(long double x);
```

明显地，C99增加了参数和返回值是 float、long double 类型的 sin()函数，并且以最后一个字母进行区分。同样，在<complex.h>中，复数正弦函数也有三个版本：

```
float _Complex csinf(float _Complex z);
double _Complex csin(double _Complex z);
long double _Complex csinl(long double _Complex z);
```

这三个函数分别对应参数和返回值是 float _Complex、double _Complex、long double _Complex 的情形。

作为程序员，我们固然可以明确指示具体调用哪一个版本的函数：

```
float x; double y; long double _Complex z;
sinf(x);            /* 调用 float 版本 sin()函数 */
sin(y);             /* 调用 double 版本 sin()函数 */
csinl(z);           /* 调用 long double _Complex 版本 sin()函数 */
```

C99为了让用户可以更加方便地调用正确版本的数学函数而引进了“type-generic math”特性，我们只要统一使用一个函数名，编译器就可以正确判断出应该调用的是哪个函数。例如：

```
float x; double y; long double _Complex z;
sin(x);             /* 调用 float 版本的 sin()函数 */
sin(y);             /* 调用 double 版本的 sin()函数 */
sin(z);             /* 调用 long double _Complex 版本的 sin()函数 */
```

上面的代码中，我们不管参数究竟是浮点实数还是复数，也不管精度是 float、double 还是 long double，一律使用 sin()进行调用。C99编译器会自动判断出我们调用的分别是 sinf()、sin()和 csinl()。

要使用 C99这项新特性，我们只需做一件事：包含标准头文件<tgmath.h>即可。由于<tgmath.h>本身就包含了<math.h>和<complex.h>，因此，源代码文件一旦包含了<tgmath.h>，我们在同一文件中就不需要使用<math.h>和<complex.h>了。

这里对“type-generic math”特性的实现技术作一些简单的分析。C99编译器要实现这项特性其实很容易，归根到底是依靠宏的替换。<tgmath.h>定义了一系列的宏，以正弦函数为例，对应的宏可能是这样的：

```
#define   sin(x)    ( (__typeof__(x) == 1) ?
                    ((sizeof(x) == 4) ? sinf(x) : sin(x))
                    : (__typeof(x)__ == 2) ?
                    ((sizeof(x) == 8) ? csinf(x) : csin(x))
                    : sin((double)(x)) )
```

[4]

为了简化，上面假设实现平台的浮点实数只有两种精度：float 和 double，在这种平台上 long double 其实也就是 double，于是复数类型实际上也只有两种：float _Complex 和 double _Complex。编译器只要进行某些扩展，譬如__typeof__()运算符，就可以实现“type-generic math”。现在假定__typeof__()的返回结果是：

x 的类型	__typeof__(x)
整数类型	0
浮点实数类型	1
复数类型	2
其它	...

有了__typeof__()运算符，再配合标准的 sizeof()，我们就可以用宏确定出应该调用哪个版本的正弦函数。我们先判断参数 x 是否浮点实数类型，如果是，再进一步判断 x 占用空间的大小，要是 sizeof(x)是4，则 x 就是 float，应该调用 sinf()，否则 x 就是 double，应该调用 sin()；如果__typeof__(x)不等于1，即 x 不是浮点实数类型，我们就再检测一次看看它是不是复数类型，如果是，就进一步判断 x 占用空间的大小，如果是8，则 x 就是 float _Complex，应该调用 csinf()，否则 x 就是 double _Complex，应该调用 csin()；最后，如果 x 既不是浮点实数也不是复数，则我们把 x 强制转换为 double 类型然后调用 sin()。这样，用户通过统一名称调用的其实是宏，背地里编译器根据宏定义判断出应该调用哪一个版本的真实函数，一切都在正常运作。

不过，有一个细节需要注意。<tgmath.h>本身包含了<math.h>和<complex.h>，这两个头文件又有大量的数学函数原型，仍然以正弦函数为例，<math.h>中肯定有它的原型声明：

```
extern double sin(double x);
```

很显然，函数原型绝对不应该参与宏替换，就是说，<tgmath.h>定义的宏 sin(x)不能影响到<math.h>中对 sin()函数的原型声明。所以，<tgmath.h>定义的这些宏必须出现在<math.h>、<complex.h>的函数原型声明的后面，这样才能保证预处理程序不会把函数原型也当作是宏的调用。

上面只是粗略的分析，而且仅仅是其中一种比较简单的实现方案，各个编译器厂商以及标准库的实现者可以有很多技术方式支持 C99的“type-generic math”特性。

*　　*　　*　　*　　*　　*

[1601]

以下划线开头、随后紧跟大写字母的标识符是保留给标准制定者（即 C 标准委员会）使用的，所以

绝对不会有任何一家厂商的编译器使用“_Complex”作为自己的扩展关键字。

[1602]

一共有5个，其中有2个是可选的：imaginary、_Imaginary_I，只有那些支持纯虚数的编译器才需要定义这2个宏。

[1603]

因为我们在C代码中使用的虚数单位是宏“I”或者“_Complex_I”，它们是C99标准库规定必须有的，每个实现平台都会把它们替换成该平台所接受的具体形式，所以使用它们丝毫不会影响程序的可移植性。

[1604]

放心，这里不会存在“递归替换”而使预处理器陷入死循环。C 标准规定，当进行一次宏替换后，即使在替换后的记号序列中发现与该宏相匹配的记号也不会再次进行替换。

附录

A C语言的发展历史

20世纪60年代后期，MIT、GE 和 AT&T Bell 实验室的合作研究项目 Multics 宣告失败，以 Ken Tompson 为首的小组意识到 Multics 根本不具有使用价值之后开始了全新的探索旅程。后来的事实证明，探索的成果足以载入史册，因为它催生了两个在计算机业界举足轻重的产品：Unix 操作系统和 C 语言。下面我们就来简单回顾一下这段不寻常的历史。[1]

Multics 失败后，Ken Tompson 立即着手研究如何建立一个完善的计算环境，他的目标在今天看来具有划时代的意义，不少革命性的概念被提出和实践，譬如进程控制机制、树型结构的文件系统、运行在用户级的命令解释器以及对设备的统一访问接口等。

Ken Tompson 和小组成员喜欢尽量使用高级语言编写程序，由于他们不喜欢 PL/I，所以他们比较多地使用 BCPL。BCPL 是 Martin Richards 在60年代开发的，BCPL 编译器被移植到 Multics 和 GECOS 系统上。当然，Ken Tompson 他们之所以使用高级语言进行编程，并不是他们早在那时就意识到可移植性的问题，而仅仅是出于个人的喜好。

当研究取得实质性进展后，Ken Tompson 使用 DEC PDP-7的汇编语言写了第一个 Unix 内核。这个过程并不是在 PDP-7本身上进行的，因为那台 PDP-7机器只是一台“裸机”，根本没有任何软件可供使用。Ken Tompson 先在 GE635机器上写代码，然后通过 PDP-7汇编器编译并输出结果——纸带。当 Unix 的基本内核、文本编辑器、简单的 shell 以及 rm、cat、cp 等命令都完成后，PDP-7就拥有了一个可独立工作的操作环境。直到这时，开发工作才正式转到 PDP-7机器上进行。有趣的是，当时还没有链接程序、库这些东西，程序代码必须全部位于同一个文件里，然后汇编器直接输出可执行文件 a.out。显然，a.out 的意思就是“assembler's output”，这个名字一直沿用至今，即使我们现在已经拥有链接程序。

1969年，Doug McIlroy 设计了一种新的系统高级语言 TMG，它可以用来编写编译器程序。受此激发，Ken Tompson 决定为 Unix 开发一种系统编程语言——B 语言。他成功地在 PDP-7上实现了 B 语言的编译器，不过，这个编译器并不直接输出机器指令，而是通过 threaded code 来解释执行。由于在 PDP-7上面 B 语言的运行速度太慢，因此不可能用 B 语言重新编写 Unix 内核以及大部分的系统程序，但人们也开始使用 B 语言编写一些小程序。

1970年， DEC 公司的新型号产品 PDP-11机器开始在 Bell 实验室投入使用。硬件速度的加快以及内存、磁盘的容量增加使越来越多的程序可以用 B 语言编写。这个时候，一些小型的库出现了，借助于库的可重用性，更大规模的程序陆续出现，其中最著名的一个是 Steve Johnson 的 Yacc。此外，Ken Tompson 还使用 PDP-11汇编重写了 Unix 内核以及一些基本命令。

在使用 PDP-11进行开发的过程中，B 语言的局限性逐渐显露出来。最突出的一点就是 B 语言是无类型（typeless）的，就是说，语言根本没有数据类型的概念，内存被看作仅仅是线性连续的单元。在最初 B 语言被开发的时候所使用的机器是基于 word（16bit）寻址的，而 PDP-11是基于 byte（8bit）寻址的，这使得 B 语言在 PDP-11上对字符数据的处理显得相当笨拙。每次都是由库代码把紧凑排列的字符数据复制到内存，在内存中这些字符数据各自相隔1个字节，当程序对这些松散的字符数据处理完毕后，

再由库代码重新“压缩”好复制到磁盘。在字节寻址的 PDP-11机器上重复这种过程是非常幼稚的做法。

而且，随着浮点实数格式的出现，B 语言的危机也越来越大，因为一个16bit 的 word 很明显放不下一个浮点实数，这对于 B 语言来说无疑是一个重大的打击。此外，基于对 word 寻址的默认，B 语言的指针被看作是 word 数组的索引，这对于字节寻址的 PDP-11 同样是别扭的规定。

种种迹象表明，在编程语言中引入类型机制是必然的选择。

1971年，Dennis Ritchie 首先作出尝试，他部分地扩充了 B 语言，包括增加 char 类型以及修改编译器令其直接输出 PDP-11汇编指令，扩展后的B语言被称为NB(New B)。这样，NB 就拥有 char、int、char 数组、int 数组、char 指针和 int 指针等类型。在 NB 中，数组的语意仍然和 BCPL、B 的数组一样，但数组、指针的相关运算已经和具体的类型相联系。不过，当 Dennis Ritchie 试图为 NB 添加 struct 类型时，NB 那些从 B 继承下来的特性便显得非常不合时宜。由于 B 的数组必须保存自身的起始地址，所以如果 struct 里面含有数组则初始化的工作相当麻烦。这些困难推动了编程语言的一次飞跃，从无类型到有类型，从 B 语言到 C 语言。

BCPL、B 和 C 都是面向计算机的系统编程语言，它们体积小，依靠库提供输入/输出以及交互功能，由于它们在教高的层次进行抽象，因此具有相当不错的可移植性。这三种语言有很多相似之处，程序均由一系列的全局变量声明以及函数声明组成，BCPL 允许过程嵌套定义，B 和 C 则不允许。三种语言都接受分离的独立单元编译以及提供某种机制在一个文件中包含另外某些文件的内容。BCPL 依赖于程序员手工设置全局矢量表中的偏移量以保证全局变量和函数的正确链接，早期的 B 没有链接概念，而后期的 B 以及 C 则使用独立的链接程序进行链接。从 BCPL 到 B 的发展过程中，某些习惯被更改，譬如赋值操作符改用“=”表示，而“/* …… */”则表示注释。BCPL、B 和 C 对字符数据都没有太多的额外支持，字符串仅仅被看作一串字符序列。BCPL 在字符串前部记录串的长度，而 B 和 C 则采取在串后部添加终止符的办法避开显式记录串长度带来的限制。B 语言在后期引入“=+”等操作符，C 也同样采用，直到1976年改为“+=”等。同样，“++”等操作符早在 B 语言时代也已经出现，后来更进一步分为前缀/后缀“++”等。

在 C 语言中，指针和数组的区别更加清晰，数组不再内值自身的起始地址，当数组名字出现在表达式里面时，我们实际上是在使用一个指向数组首个元素的指针。这项特性使到很多 B 语言代码仍然可以继续有效，当然，那些直接对数组名字进行赋值操作的代码在 C 语言中则不再有意义。总之，C 语言赋予数组全新的解释，这有利于引入类型机制。

此外，C 语言和它的前辈的重大的区别是 C 拥有一个完整的类型系统以及相应的声明语法。NB 只有 char、int、char 数组、int 数组、char 指针和 int 指针，而 C 语言基于通用类型的设计特性使它可以表达任何类型的实体，包括数组元素实体、函数返回实体以及指针指向的实体。C 的声明语法保证了这一概念的实现，例如：

```
int i, *pi, **ppi;
```

上面的声明意味着在表达式里 i、(*pi)、(*(*ppi))都是一个 int。C 的这些设计思路部分来自 Algol 68的启发。

随后，Dennis Ritchie 接纳了 Alan Snyder 的建议，采用“&&”、“||”表示逻辑短路运算操作符。在“&&”、“||”被引入之前，BCPL、B 的条件语句必须根据上下

文来判断自身的语义，这引起了不少混乱。在 B 里，“&”和“==”的优先极相同，为了使转换过程尽量平滑，Dennis Ritchie 在 C 语言里依然保留这一规定，但在今天看来，这不是一个好主意。因为在类似下面的代码中：

```
if ((x && mask) == y)
    ...
```

里面的那一对括号很容易被人们遗忘而使到预期的运算顺序被改变。

1972到1973年间，C 语言陆续有几项改进，其中最突出的是预处理机制的引入。最初的预处理只支持#include 和不带参数的#define，后来逐渐增强至可以定义带参数的宏以及支持条件编译。

1973年后期，C 语言已经基本成型。C 编译器以及语言本身的成熟使人们可以考虑用 C 重写 Unix 的内核。另外，C 编译器还被移植到 Honeywell 635和 IBM 360/370机器上。由于库对 C 语言至关重要，因此人们开始重视 C 库的开发。Lesk 编写的 portable I/O package 后来经重写而成为 C 的标准 I/O 库。

在1973年至1980年期间，C 再次引入新特性。unsigned、long、union 和 enum 类型相继加入 C 语言。对 Unix 内核的重写成功增强了人们使用 C 语言的信心，大量工具软件用 C 语言重写并移植到其它平台。同时，Steve Johnson、Ken Tompson 和 Dennis Ritchie 开始把 Unix 移植到 Interdata 8/32机器上。

到了1977年的时候，可移植性和类型安全越来越被人们重视。C 语言仍然未彻底洗掉“无类型”的烙印，譬如说，指针类型和整数类型几乎没有清晰的区别，它们甚至可以相互赋值，于是强制转换机制被引入以表示明显的类型转换。

Interdata 8/32的 Unix 移植工作成功后，Tom London 和 John Reiser 把 Unix 移植到 VAX 11/780上。由于 VAX 系列机器在当时被广泛使用从而使 Unix 在 AT&T 内外迅速传播。虽然从70年代中期 Unix 已经在 AT&T 内部的多个计划中被使用，但它在外界的流行却是可移植性的功劳。

1978年，Dennis Ritchie 和 Brian Kernighan 出版了 *The C Programming Language* 一书，尽管它缺乏一些 C 语言的最新特性的描述，但该书仍然成为了当时的事实标准。到了80年代，几乎所有系统都提供 C 编译器。从大型商业软件到个人编程，C 语言都被广泛地使用。1982年，人们开始意识到 C 语言需要被标准化。当时，*K&R* 已经不能满足精确描述语言的需要，它没有提及 void 和 enum 类型，虽然众多的独立编译器产品都支持 struct 以及在函数中传递和返回 struct 变量，但 K&R 对此也缺乏完整描述。同时，由于 C 语言在商业合同中被广泛使用，因此提出一个官方标准是非常重要的。于是，1983年 X3J11委员会成立，它制定的 C 语言标准草案在1989年被 ANSI 批准成为正式的美国标准（ANSI X3.159:1989），随后，ISO 在1990年接纳该标准为国际标准（ISO/IEC 9899:1990）。在这个过程中，X3J11委员会一直非常认真地考虑各种对 C 语言的扩展，他们的意图很清晰，那就是在通用的、已经存在的 C 语言定义中，以提高可移植性为目标，整理出一个无歧义的、一致的 C 语言标准。

X3J11仅仅引入了一个极其重要的改动，那是从 C++学习得来的函数原型。旧的函数声明：

```
double sin();
```

改为：

```
double sin(double);
```

尽管新的原型声明与旧式风格的定义存在配合上的问题，但 X3J11仍然坚持认为新风格优越，作为妥协的产物，标准允许两种形式并存。

此外，X3J11还作出了一些细微的调整，例如 const 和 volatile 修饰字以及类型提升规则。无论如何，C 标准总算成功地解释所有变动以及为实现者给出足够精确的描述。而且，X3J11还注意到一个完整的标准 C 库同样重要，因此 X3J11花费了大量的时间来设计和描述 C 的标准库。

在 C 语言标准颁布后，X3J11委员会的主要任务是解释标准，但它属下的 NCEG（Numerical C Extensions Group）仍然进行扩充 C 语言的工作，这些扩充更多的是在数值运算领域。

C 语言有两点特性使它区别于其它同类。一个是数组与指针的关系，另一个是声明语法，同时，这也是 C 语言备受批评的地方。初学者经常对 C 语言的这两个特性感到困惑，而一些历史偶然事件则更加剧了混乱。最突出的不当之处是编译器对类型错误的容忍，由于 C 脱胎于无类型语言，它不可能一下子就以一个全新的面孔出现在早期的使用者面前，因此人们必须一边发展它一边为了兼容已有的代码而容忍它。

1977年甚至更后期的编译器都允许整数和指针的相互赋值，虽然 *K&R* 已经足够清楚（但不完全）地描述到类型规则，不过 *K&R* 没有强迫编译器必须据此作出修改。而某些规则引起了不良的后果，譬如方括号在函数定义中的使用：

```
int f(a)
int a[];
{ ... }
```

显然，在 C 语言中，a 只能是一个指针，但部分使用者产生了误会，以为他们可以把一个由数组生成的指针作为参数传递给函数。

在 K&R C 中，保证所有参数类型正确是程序员自己的责任，编译器不会作出任何检查。这个不太好的设计由 X3J11进行了改善。至于 C 语言的“*”操作符，它在普通表达式里工作良好，但在更复杂的表达式中，人们需要添加必要的括号：

```
int *fp();
int (*fp)();
```

上面两个声明是不同的，而：

```
int *(*pfp)();
```

则显得更复杂。

C 语言有丰富的类型可供声明，另一方面 C 的声明必须由里向外地看，这是比较难掌握的。尽管 Sethi 研究得出结论，只要把间接操作符由前缀操作符变为后缀操作符就会使情况得到改善，但那个时候已经太迟了。除此之外，C 的声明仍然是一个相当合理的设计。

C 语言对数组的处理方法主要来源于实践的沉淀，当然，这些有历史渊源的设计还是具有不少优点。在 C 里，指针和数组的关系非常密切，尽管有些难度，但依然可以被人们熟练掌握以运用。更重要的是，C 语言的描述能力相当强大，譬如在运行时长度才确定的变长数组只需用一些基本规则就可以表达。特别地，字符串使用与数组相同的处理机制，对比一下 Algol 68和 Pascal 就知道 C 的优越。Algol 68支持固定长度数组以及变长

数组，但实现变长数组需要一大堆语言机制，这对编译器的要求非常高，某些编译器根本没有提供变长数组的支持；原来的 Pascal 只有固定长度的数组和字符串，这对实际应用来说存在一定的限制，虽然在后来人们作了某些修改，但改进后的 Pascal 并没有流行起来。

C 语言把字符串作为字符数组外加一个终止符来处理，配合以特别的初始化规则，这使得 C 语言能够很方便地处理字符串，因为字符串和其它任何数组都是在统一的语义规则中进行描述，比起把字符串单独列为一个类型要好。当然，代价就是某些对字符串的操作比较复杂，因为代码必须要找到字符串的尾部。

不过，C 语言对数组的处理总体来说存在对以后扩展及优化处理不利的因素。C 程序中到处可见的指针，包括显式声明的或者由数组转化而来的，都要求优化者必须小心谨慎地进行处理。优秀的编译器能够知道大多数的指针可能会改变哪些实体，但对于一些重要的场合编译器依然无能为力。例如在支持矢量运算的计算机上很难对函数参数是数组的情形进行优化，因为编译器无法判断对应的两个内存区域是否存在重叠。

从本质上讲，C 语言对数组的描述是如此的特别因此对数组的改动及扩展都会很难适应目前的语言，甚至对多维变长数组的支持都不是十分容易的事。总之，C 语言以一个简单的机制覆盖了字符串和数组的大部分应用，但给以后的扩展留下了一定的困难。

C 语言保持了体积小的特点，C 编译器也同样如此。C 语言支持的类型和操作都是实践证明所必要的、由真实的机器提供实际支持的。同时，C 语言又足够抽象地使程序拥有相当好的可移植性。而且，C 和它的核心库总是维持一种适当的关系，它们覆盖了大部分的需要但又不会庞大到超出这个范围而显得臃肿。

最后，C 语言在发展过程中的改动相对较小，可以说非常稳定。它增加的特性非常少，和 Fortran、Pascal 等语言对比来看更是如此。尽管在一开始设计 C 语言时并没有过多关注它的可移植性，但结果 C 语言被成功地运用在各种平台上，从 PC 到大型机，从小型软件到操作系统。

*　　*　　*　　*　　*　　*

[A01]

本文根据 Dennis Ritchie 的论文 *The Development of the C Language* 整理而成，特此声明。

B P6/GCC 汇编语言简介

本书以 GNU/Linux 为实验平台，很自然地采用 ***gcc*** 作为 C 编译器，由于大家可能对 GCC 汇编代码感觉比较陌生，因此在这里有必要向大家简单介绍一下。注意，这里所说的汇编代码包括：

- 直接被汇编语言编译程序 as 接受的独立汇编代码
- 被 GCC 接受的嵌入式汇编代码

虽然两者有相似之处，但性质完全不同。前者存在于汇编代码文件（“*.s”）中，as 可以直接识别这些代码；后者在 C 源文件（“*.c”）中使用，这些嵌入的汇编代码必须首先由 ***gcc*** 进行识别处理，然后 ***gcc*** 再把它们转化为独立汇编代码，随后才能交由 as 进行编译。在本书中我们大量使用的是独立汇编代码，所以这里首先对它进行介绍。

对应 P6处理器的 AT&T 风格独立汇编代码和 Intel 风格汇编代码尽管有不同之处，但其实差别很小，大家只需注意以下几点即可：

- 指令带有后缀，指示操作数的长度（8位、16位还是32位）[1]：

```
movb  $90, -24(%ebp)                    # 操作数是8位的
movw  %ax, %bx                          # 操作数是16位的
movl  %ebx, %eax                        # 操作数是32位的
```

- 源操作数（或常数）和目标操作数的位置与 Intel 风格刚好相反。在 Intel 风格的代码中，目标操作数位于“前面”，源操作数（或常数）位于“后面”；而在 AT&T 风格的代码中，源操作数（或常数）位于“前面”，目标操作数位于“后面”：

```
mov   eax, ebx                          # Intel 风格（把 ebx 的内容复制到 eax）
movl  %ebx, %eax                        # AT&T 风格
add   eax, 8                            # Intel 风格（使 eax 的值增加8）
addl  $8, %eax                          # AT&T 风格
```

- 寄存器名前面加上“%”， 常数（包括符号的值）以“$”开头：

```
movl  $1234, %eax                       # 把十进制常数1234放入 eax
movl  $0x64, %eax                       # 把十六进制常数0x64放入 eax
movl  $var, %ebx                        # 把符号 var 的值[2]放到 ebx
```

- 间接寻址用圆括号表示：

```
movl  0x8048090, %eax                   # 寻址方式：[偏移量]
movl  (%ebp), %eax                      # 寻址方式：[基址]
movl  0x123(%ebx), %ecx                 # 寻址方式：[基址+偏移量]
movl  0x123(, %ebx, 4)                  # 寻址方式：[索引*系数+偏移量]
movl  0x123(%edx, %ebx)                 # 寻址方式：[基址+索引+偏移量]
movl  0x123(%esi, %edi, 8), %eax        # 寻址方式：[基址+索引*系数+偏移量]
```

总的说来，AT&T 风格的独立汇编代码并不复杂，对于本身已经熟悉 P6处理器汇编指

令的读者，理解起来应该毫无困难。下面我们重点讨论在 C 文件中使用的嵌入式汇编。

首先，独立汇编代码不需改动就可以作为嵌入汇编代码使用，例如：

```
C01         /* Code B-01, file name: b01.c */
C02         int main()
C03         {
C04             __asm__(" movl  %eax, %ebx ");
C05         }
```

B-01是一个极其简单的示例，在 main()函数里，我们使用 GCC 的“__asm__”扩展关键字通知编译器随后的括号内有若干条嵌入式汇编代码。注意，这里的括号、双引号、分号一个也不能少。

如果有不止一行的嵌入式汇编代码，那么有几个办法：

- 反复使用“__asm__”关键字

```
C01     /* Code B-02, file name: b02.c */
C02     int var = 1;
C03     int main()
C04     {
C05         __asm__(" movl  var, %%eax ");
C06         __asm__(" incl  %%eax ");
C07         __asm__(" movl  %%eax, var ");
C08     }
```

- 使用分号

```
C01     /* Code B-03, file name: b03.c */
C02     int var = 1;
C03     int main()
C04     {
C05         __asm__
C06         (
C07             " movl  var, %eax; "
C08             " incl %eax;       "
C09             " movl %eax, var   "
C10         );
C11     }
```

- 使用转义字符

```
C01     /* Code B-04, file name: b04.c */
C02     int var = 1;
C03     int main()
C04     {
```

```
C05         __asm__
C06         (
C07             " movl  var, %%eax      \n\t "
C08             " incl  %%eax           \n\t "
C09             " movl  %%eax, var           "
C10          );
C11     }
```

B-02、B-03、B-04所做的事情完全是一样的，都是把全局 int 变量 var 的值由1增加到2。在 B-04的嵌入汇编代码中，“\n\t”是转义字符，目的是让 ***gcc*** 在转换时添加这两个字符，由于在独立汇编文件中汇编语言编译程序依靠回车符（“\n”）和分号辨别汇编指令的结尾，因此 B-03/04能够保证多行的嵌入汇编代码可以被转换为对应的多行独立汇编代码。

B-02/03/04对应的汇编代码是一样的，我们以 B-04为例，把它编译成汇编代码文件并查看其中的内容：

```
$gcc  -S  b04.c
$less  b04.s

   ...
main:
   ...
#APP
   movl  var, %eax
   incl  %eax
   movl  %eax, var
#NO_APP
   ...
```

很清楚，“#APP”和“#NO_APP”之间的语句就是我们的嵌入式汇编代码，可见，当我们使用独立汇编代码作为嵌入式汇编时，***gcc*** 基本上只是原封不动地把这些代码复制到汇编代码文件（当然，对“__asm__”、括号、双引号、转义字符等的分析处理仍然需要 ***gcc*** 来完成）。

既然独立汇编代码已经可以直接使用在 C 文件中，为何还需要额外的嵌入式汇编代码格式呢？从上面的例子大家已经明白，独立汇编代码是由汇编语言编译程序进行编译的，所以我们必须详细地指明使用什么寄存器、内存操作数的地址等等，但在一个 C 源文件里，对于存放在栈里面的自动变量，要准确知道其地址不是一件轻而易举的事，尤其是当内部变量比较多的时候。其次，有时我们可能不需要具体指定使用某一个寄存器，只要能够完成任务即可，至于选用哪个寄存器是编译器自己的选择。最后，可能我们真的很懒，不愿意为具体细节操心，我们只关心指令而不关心输入和输出过程的细节。总之，GCC 提供的嵌入式汇编扩展能够满足这一切需求。

嵌入式汇编代码的格式如下：

```
__asm__
(
    " instrutions "
    : output operands (optional)
    : input operands (optional)
    : clobbered registers (optional)
);
```

我们以一个简单的例子开始：

```
int main()
{
    int x, y, z;
    ...                          /* 把 x、y 的和赋给 z */
}
```

现在要使用嵌入式汇编完成上面的代码，分析后可知，要计算 x、y 的和，一般可以通过加法运算来完成，即使用 add 指令，显然，输入操作数是 x、y，输出操作数是 z，如果我们并不关心编译器选用哪个寄存器完成此次加法运算，则代码如下：

```
int main()
{
    int x, y, z;
    __asm__
    (
        " addl  %1, %0 "
        : "=r"(z)
        : "m"(x), "0"(y)
    );
}
```

在 add 指令中，我们不直接指定某个寄存器，而是使用“%0”、“%1”来表示相加的两个操作数，至于%0、%1究竟是寄存器操作数还是内存操作数，均由下面的约束描述进行说明。

根据格式，第一个冒号后面的第一个描述适用于指令中的操作数%0，第二个描述适用于指令中的操作数%1……如此类推。在例子中，输出操作数一栏只有一个约束描述，显然它就是对应操作数%0的。约束描述清楚地指出%0的性质。譬如在例子中， %0的约束描述是“r”，这表示%0是一个寄存器操作数（并且是通用寄存器）。字母“r”前面的“=”表示指令执行完毕后编译器还要把%0的值输出到某个变量，输出操作数一栏的所有约束描述前面都要加上“=”以区别输入操作数的约束描述。在约束描述后面的“(z)”是告诉编译器当 add 指令执行完毕后要把%0的值复制给变量 z。

明白了%0的约束描述，再来看第二个冒号后面的约束描述就很容易理解。由于输出操作数一栏只有一个约束描述（它对应%0），所以输入操作数一栏的第一个约束描述适用于操作数%1。字母“m”表示%1是一个内存操作数，“(x)”表示%1应该从变量 x 获得初始值，但%1本身就是内存操作数，所以这意味着%1就是变量 x。

输入操作数一栏里有两个约束描述，但 add 指令只有两个操作数（%0和%1），因此，第二个输入操作数约束描述不是关于%0和%1的，那么它有什么意义呢？不要忘记，我们是要计算 x、y 的和，现在 x 就是内存操作数%1，而寄存器操作数%0还没有用 y 的值进行初始化，所以，最后一个约束描述的目的并不是给出某个操作数的约束而是用某个值去初始化对应的操作数。初始化哪个操作数？字符“0”已经很清楚地告诉编译器：在 add 指令执行前先用变量 y 的值初始化操作数%0。

我们把上面的 C 代码编译成汇编代码，其中嵌入式汇编对应的内容如下：

```
    ...
    movl  -8(%ebp), %eax
#APP
    addl  -4(%ebp), %eax
#NO_APP
    movl  %eax, -12(%ebp)
    ...
```

在 main()函数里，变量 x、y、z 的地址分别是-4(%ebp)、-8(%ebp)、-12(%ebp)。上面三条指令中，第一条是把变量 y 的值复制到 eax 寄存器，第二条是把变量 x 和 eax 的值相加（结果存放在 eax），第三条是把 eax 的值复制给变量 z。其中，add 指令是我们亲自指定使用的，其余指令是 ***gcc*** 根据我们的指示自动添加的，同时，我们也没有具体指定使用哪个寄存器，***gcc*** 选择使用 eax。

上面的示例中我们省略了第3个冒号以及 clobbered registers 项，这是因为我们并不需要限制编译器对寄存器的选择。我们对代码稍作修改，添加多一行：

```
int main()
{
    int x, y, z;
    __asm__
    (
        " addl  %1, %0 "
        : "=r"(z)
        : "m"(x), "0"(y)
        : "%eax"                        /* 告诉编译器不要使用 eax 寄存器 */
    );
}
```

本来，对于“r”约束描述的操作数，编译器可以选择使用任何一个通用寄存器，现

在，我们明确指出 eax 寄存器不能被使用，编译器就会作出相应调整：

```
    ...
    movl  -8(%ebp), %edx
#APP
    addl  -4(%ebp), %edx
#NO_APP
    movl  %edx, -12(%ebp)
    ...
```

很清楚，编译器知道 eax 不能被使用之后选择了 edx 寄存器。

我们既可以使用%0、%1……等代表指令的操作数，也可以直接指定编译器使用哪些寄存器，这通常需要两方面的配合：

- 直接在指令中给出完整的寄存器名字
- 使用对应该寄存器的约束描述

例如：

```
int main()
{
    int x, y, z;
    __asm__
    (
        " addl  %%edx, %%eax "
        : "=a"(z)
        : "a"(x), "d"(y)
    );
}
```

这是计算 x、y 的和并赋给 z 的嵌入式汇编代码的第二个版本。我们没有使用抽象的%0、%1等操作数然后由编译器自行选择寄存器，而是直接在 add 指令中指出把 edx 的值和 eax 的值相加（结果存放在 eax）。这意味着在 add 指令执行前我们必须保证已经把 eax、edx 用 x、y 的值进行初始化。由于没有%0、%1等抽象操作数，所以自然也就不存在哪个约束描述对应指令中的哪个操作数的问题。于是，例子里的约束描述仅仅是告诉编译器如何处理操作数的初始化以及结果的输出。

第一个冒号后面的约束描述中，“=”以及字母“a”表示编译器要在指令执行完毕后要把 a 寄存器的值复制给某个变量，本来，a 寄存器可以是 al、ax 或者 eax，至于究竟是 al、ax 还是 eax，则由编译器根据相应的 C 变量的长度来决定。由于 add 指令执行完毕后结果要被复制到变量 z，而变量 z 的长度是32位，因此这里的 a 寄存器就是 eax。

第二个冒号后面的约束描述相信大家已经很清楚，它们分别告诉编译器在指令执行之前用变量 x、y 的值初始化 eax、edx 寄存器。

上面的嵌入式汇编代码对应的独立汇编代码是：

```
    movl  -4(%ebp), %eax
    movl  -8(%ebp), %edx
#APP
    addl  %edx, %eax
#NO_APP
    movl  %eax, -12(%ebp)
```

由此可见，当我们直接在嵌入式汇编代码中指定具体的寄存器名字时，通常要使用相应的约束描述来进行初始化又或者输出结果。

虽然上面没有给出直接的例子，但我们应该知道，一段嵌入式汇编代码可以没有输入操作数的约束描述，例如：

```
int main()
{
    int x;
    __asm__
    (
        " incl  %0 "                    /* 变量 x 自增 */
        : "=m"(x)
    );
}
```

也可以没有输出操作数的约束描述，例如：

```
int main()
{
    int x;
    __asm__
    (
        " incl  %0 "                    /* 变量 x 自增 */
        :                               /* 这里的冒号要照写 */
        : "m"(x)
    );
}
```

最后，要注意的是，编译器在优化代码时可能会擅自删除我们的嵌入式汇编代码，如果我们想防止编译器对嵌入汇编作出任何的改动，可以加上“__volatile__”扩展关键字，例如：

```
int main(){
    int x, y;
    long long z;
    __asm__  __volatile__
```

```
        (
            " imull  %1 "                    /* 计算(x*y)的值 */
            : "=A"(z)
            : "m"(x), "a"(y)
        );
    }
```

关于约束描述的具体情况，读者可进一步参阅[Stallman 2004]。

*　　*　　*　　*　　*　　*

[B01]

在不会产生歧义的场合，后缀其实能够被省略掉，例如：

```
mov  %eax, %ebx
```

是完全正确的。

为了风格统一，本书所有汇编指令都带有适当的后缀。

[B02]

作为对比，请读者注意：

```
movl  $var, %ebx              # 把符号 var 的值放入 ebx
movl  var, %ebx               # 把符号 var 所代表的数据的值放入 ebx
```

C GCC 的安装与使用

GCC 是由 GNU 组织负责进行开发、维护的自由软件项目，根据名字我们可以知道这个项目包含不止一个编译器产品。目前，GCC 支持 C、C++、Objective-C、Java、Fortran 和 Ada 等6种编程语言，所以，GCC 总共有6个编译器。本书主要讲述 C 语言，某些时候可能会涉及到 C++，因此我们只要拥有 GCC 的其中两个编译器（***gcc*** 和 ***g++***）即可上机测试代码。几乎所有的 GNU/Linux 版本都会附带一整套编程软件供用户选择安装，尽管这些软件的版本未必是最新的，但我们可以利用它们来编译源代码，从而得到最新版本的软件。下面以 Slackware GNU/Linux 9.1为例，简单介绍一下如何编译 GCC 3.4.0。

首先，我们要到 GNU 的网站下载源代码，gcc 3.4.0和 g++ 3.4.0对应的源代码文件分别是 gcc-core-3.4.0.tar.gz 和 gcc-g++-3.4.0.tar.gz，把这两个文件复制到某个临时目录，譬如“*/tmp*”，然后以 root 身分执行下面的命令：

```
#cd /tmp
#tar zxvf gcc-core-3.4.0.tar.gz
#tar zxvf gcc-g++-3.4.0.tar.gz
```

操作成功后，*/tmp* 下面就会新增一个目录 *gcc-3.4.0*，我们进入这个目录就可以开始编译前的配置工作：

```
#cd gcc-3.4.0
#mkdir build
#cd build
#../configure --prefix=/opt/GCC340 --enable-threads=posix
```

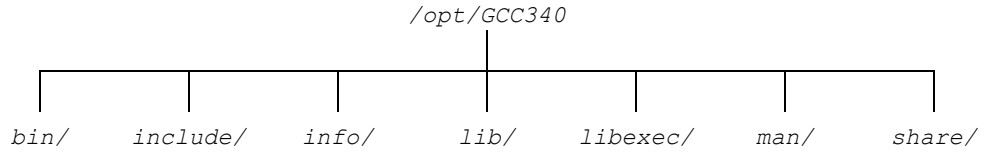
虽然有多个配置选项可供使用，但通常我们只须指定安装目录，在上面的命令中，我们指示 GCC 的安装路径是*/opt/GCC340*，如果不具体指出安装目录，则系统的默认路径是*/usr/local*。当配置完毕后我们就可以正式开始编译源代码：

```
#make bootstrap
```

整个编译过程大约持续几十分钟，具体时间视机器的速度而定。如果没有出现任何编译错误的信息，则表明编译成功，我们接下来可以执行安装：

```
#make install
```

键入这个命令后，安装脚本会把编译好的可执行文件、库以及其它所有必需的文件复制到我们一开始指定的安装目录*/opt/GCC340*。至此，*/opt/GCC340*的目录结构如下：



由于 GCC 的默认编译选项带有“-g”，所以生成的可执行文件以及库都附带了不必要的调试信息，我们不会去调试编译器和库本身，因此这些调试信息对我们来说没有任何用处，为了节省磁盘空间，我们可以自行把这些调试信息剔除，而且，对于可执行文件来说，

符号表也是不必要的，同样可以删除：

```
#cd /opt/GCC340
#strip  bin/*
#strip  libexec/gcc/i686-pc-linux-gnu/3.4.0/*
#strip  -g  lib/*
#strip  -g  lib/gcc/i686-pc-linux-gnu/3.4.0/*
```

这样一来，程序和库所占用的磁盘空间大约可以减少40%。

此外，我们还要确保今后在命令行中调用的是新版本的 GCC，为此，我们先找出目前旧版本 GCC 的位置所在：

```
#which  gcc  g++
/usr/bin/gcc
/usr/bin/g++
#mv  /usr/bin/gcc  /usr/bin/gcc.old
#mv  /usr/bin/g++  /usr/bin/g++.old
#ln  -s  /opt/GCC340/bin/gcc  /usr/bin/gcc
#ln  -s  /opt/GCC340/bin/g++  /usr/bin/g++
```

which 命令的输出告诉我们旧版本的 gcc/g++位于*/usr/bin* 目录，于是我们通过 ***mv*** 命令把旧版本的两个编译器程序改名（增加后缀“.old”以作区别），然后建立新的符号连接，这些新的符号连接指向新版本的编译器程序。

最后，由于我们在编译 g++时还编译了新的 C++动态库 libstdc++.so，为了能够正常使用这个新的 C++库，我们需要给系统指出库的详细路径。正如前面在动态库一节中介绍的那样，我们可以通过修改*/etc/ld.so.conf* 实现。使用任何文本编辑器（譬如 ***vi***）在*/etc/ld.so.conf* 的最前面加上一行：

```
/opt/GCC340/lib
```

即可。保存后删掉已有的*/etc/ld.so.cache* 并马上运行 ldconfig 命令：

```
#rm  /etc/ld.so.cache
#ldconfig
```

这样就建立了新的系统搜索缓存文件，而且把新的库路径添加到系统的搜索列表中。最后，我们把源代码及编译过程的中间文件全部删除：

```
#cd  /tmp
#rm  -rf  gcc-*
```

整个编译、安装过程宣告结束。从现在开始，我们的系统上正式拥有最新版本的 GCC 编译器。我们可以试验一下：

```
#gcc  -v
Reading specs from /opt/GCC340/lib/gcc/i686-pc-linux-gnu/3.4.0/specs
Configured with:./configure --prefix=/opt/GCC340 --enable-threads=posix
Thread model: posix
gcc version 3.4.0
#
```

这段信息的出现意味着我们已经正确安装了 GCC 3.4.0。

gcc 3.4.0完全支持 C89标准，对 C99标准也提供了大部分的支持。同时，***gcc*** 也有自己的扩展功能集，包括在 C89标准基础上进行扩充的 GNU89以及在 C99标准基础上进行扩充的 GNU99。在 ***gcc*** 还没有对 C99标准提供完整支持之前，GNU89是 ***gcc*** 默认的编译标准，将来等到 ***gcc*** 能够完整支持 C99标准时，GNU99就会成为 ***gcc*** 的默认编译标准。如果我们在程序中没有使用 GNU89提供的扩展功能，并且希望自己的代码严格符合 C89/C99标准，可以通过对应的编译选项作出指示：

```
$gcc  -std=c89  -pedantic  sample.c
$gcc  -std=c99  -pedantic  sample.c
```

注意，“-pedantic”选项一定要同时加上去，否则编译器即使发现不符合标准的地方也不会发出警告。

当使用“-std=c89”编译选项时，GNU89的扩展关键字“asm”、“typeof”和“inline”就会失效；当使用 “-std=c99”编译选项时，GNU99的扩展关键字“asm”、“typeof”就会失效。失效的意思是编译器会把它们当作普通的标识符进行处理，譬如：

```
int main()
{
   asm(“movl  %eax, %ebx”);
}
```

如果在默认的 GNU89编译模式下，***gcc*** 会知道以扩展关键字“asm”开头的语句是嵌入式汇编代码，从而作出正确的处理。一旦使用“-std=c89”或“-std=c99”编译选项，***gcc*** 就不再认为“asm”是关键字，上面的嵌入式汇编语句会被处理成函数调用，***gcc*** 根据语法分析，认为 asm()是一个函数，它的参数是字符串“movl %eax, %ebx”的地址，显然，这是错误的。因此，如果要使用“-std=c89”编译选项，我们应该用“__asm__”代替“asm”、用“__typeof__”代替“typeof”、用“__inline__”代替“inline”；如果要使用“-std=c99”编译选项，只要用“__asm__”代替“asm”、用“__typeof__”代替“typeof”即可，因为“inline”已经正式成为 C99标准的关键字。

关于 GCC 的详细使用说明大家可以进一步参阅[Stallman 2004]。

*　　*　　*　　*　　*　　*

[C01]

D GPL 原文

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed

on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and

distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are

prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR

CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

one line to give the program's name and an idea of what it does.
Copyright (C) *yyyy name of author*

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) *year name of author*
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details

type 'show w'. This is free software, and you are welcome
to redistribute it under certain conditions; type 'show c'
for details.

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than 'show w' and 'show c'; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright
interest in the program 'Gnomovision'
(which makes passes at compilers) written
by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License.

E Linux-2.6.6系统调用简表

编号	系统服务
0	restart_syscall
1	exit
2	fork
3	read
4	write
5	open
6	close
7	waitpid
8	creat
9	link
10	unlink
11	execve
12	chdir
13	time
14	mknod
15	chmod
16	lchown16
17	ni_syscall
18	stat
19	lseek
20	getpid
21	mount
22	oldumount
23	setuid16
24	getuid16
25	stime
26	ptrace
27	alarm
28	fstat
29	pause
30	utime
62	ustat
63	dup2
64	getppid
65	getpgrp

编号	系统服务
31	ni_syscall [1]
32	ni_syscall
33	access
34	nice
35	ni_syscall
36	sync
37	kill
38	rename
39	mkdir
40	rmdir
41	dup
42	pipe
43	times
44	ni_syscall
45	brk
46	setgid16
47	getgid16
48	signal
49	geteuid16
50	getegid16
51	acct
52	umoun
53	ni_syscall
54	ioctl
55	fcntl
56	ni_syscall
57	setpgid
58	ni_syscall
59	olduname
60	umask
61	chroot
96	getpriority
97	setpriority
98	ni_syscall
99	statfs

66	setsid
67	sigaction
68	sgetmask
69	ssetmask
70	setreuid16
71	setregid16
72	sigsuspend
73	sigpending
74	sethostname
75	setrlimit
76	old_getrlimit
77	getrusage
78	gettimeofday
79	settimeofday
80	getgroups16
81	setgroups16
82	old_select
83	symlink
84	lstat
85	readlink
86	uselib
87	swapon
88	reboot
89	old_readdir
90	old_mmap
91	munmap
92	truncate
93	ftruncate
94	fchmod
95	fchown16
100	fstatfs
101	ioperm
102	socketcall
103	syslog
104	setitimer
105	getitimer
106	newstat
107	newlstat
108	newfstat
109	uname
110	iopl
111	vhangup
112	ni_syscall
113	vm86old
114	wait4
115	swapoff
116	sysinfo
117	ipc
118	fsync
119	sigreturn
120	clone
121	setdomainname
122	newuname
123	modify_ldt
124	adjtimex
125	mprotect
126	sigprocmask
127	ni_syscall
128	init_module
129	delete_module
130	ni_syscall
131	quotactl
132	getpgid
133	fchdir
134	bdflush
135	sysfs
136	personality
137	ni_syscall
164	setresuid16
165	getresuid16
166	vm86
167	ni_syscall
168	poll
169	nfsservctl
170	setresgid16
171	getresgid16

```
138     setfsuid16
139     setfsgid16
140     llseek
141     getdents
142     select
143     flock
144     msync
145     readv
146     writev
147     getsid
148     fdatasync
149     sysctl
150     mlock
151     munlock
152     mlockall
153     munlockall
154     sched_setparam
155     sched_getparam
156     sched_setscheduler
157     sched_getscheduler
158     sched_yield
159     sched_get_priority_max
160     sched_get_priority_min
161     sched_rr_get_interval
162     nanosleep
163     mremap
198     lchown
199     getuid
200     getgid
201     geteuid
202     getegid
203     setreuid
204     setregid
205     getgroups
206     setgroups
207     fchown
208     setresuid
209     getresuid
```

```
172     prctl
173     rt_sigreturn
174     rt_sigaction
175     rt_sigprocmask
176     rt_sigpending
177     rt_sigtimedwait
178     rt_sigqueueinfo
179     rt_sigsuspend
180     pread64
181     pwrite64
182     chown16
183     getcwd
184     capget
185     capset
186     sigaltstack
187     sendfile
188     ni_syscall
189     ni_syscall
190     vfork
191     getrlimit
192     mmap2
193     truncate64
194     ftruncate64
195     stat64
196     lstat64
197     fstat64
232     listxattr
233     llistxattr
234     flistxattr
235     removexattr
236     lremovexattr
237     fremovexattr
238     tkill
239     sendfile64
240     futex
241     sched_setaffinity
242     sched_getaffinity
243     set_thread_area
```

210	setresgid	244	get_thread_area
211	getresgid	245	io_setup
212	chown	246	io_destroy
213	setuid	247	io_getevents
214	setgid	248	io_submit
215	setfsuid	249	io_cancel
216	setfsgid	250	fadvise64
217	pivot_root	251	ni_syscall
218	mincore	252	exit_group
219	madvise	253	lookup_dcookie
220	getdents64	254	epoll_create
221	fcntl64	255	epoll_ctl
222	ni_syscall	256	epoll_wait
223	ni_syscall	257	remap_file_pages
224	gettid	258	set_tid_address
225	readahead	259	timer_create
226	setxattr	260	timer_settime
227	lsetxattr	261	timer_gettime
228	fsetxattr	262	timer_getoverrun
229	getxattr	263	timer_delete
230	lgetxattr	264	clock_settime
231	fgetxattr	265	clock_gettime
266	clock_getres	275	ni_syscall
267	clock_nanosleep	276	ni_syscall
268	statfs64	277	mq_open
269	fstatfs64	278	mq_unlink
270	tgkill	279	mq_timedsend
271	utimes	280	mq_timedreceive
272	fadvise64_64	281	mq_notify
273	ni_syscall	282	mq_getsetattr
274	ni_syscall		

*　　*　　*　　*　　*　　*

[E01]

“ni_syscall”表示：

1） 在新版本的内核中已经被取消　　又或者

2） 预留给新的系统服务，但该服务暂时还没有被实现。

F 相关网络资源

http://cm.bell-labs.com/cm/cs/who/dmr/
Dennis M. Ritchie 的个人主页

http://www.ansi.org
美国国家标准协会

http://www.cuj.com
技术杂志 C/C++ Users Journal

http://www.gnu.org
最大的自由软件组织 GNU 的大本营

http://www.gnu.org/software/gcc
GCC 的官方网址

http://www.incits.org
美国国家信息技术委员会

http://www.linux.org
这个站点可以帮助你了解自由操作系统 Linux

http://www.linuxiso.org
各种自由操作系统的光盘镜像下载站点

http://www.yaoxy.com
本书的维护网站