我们研究三种最重要的数字表示无符号编码，基于传统的二进制表示法，表示大于或者等于0的数字。补码编码是表示有符号整数的最常见的方式。浮点数编码是表示实数的科学计数法的，以二为基数的版本计算机用这些不同的方表示方法，实现算术运算，例如加法和乘法，类似于对应的整数和实数运算。

浮点运算具有完全不同的数学属性，由于表示的精度有限，浮点运算是不可结合的。整数运算和浮点数运算会有不同的数学属性，是因为他们处理数字表示有限性的方式不同，整数的表示，虽然只能编码一个相对较小的数值范围，但是这种表示是精确的，而浮点数虽然可以编码一个较大的数值范围，但是这种表示只是近似的。

后面我们会讲到大量计算机的安全漏洞，都是由于计算机算术运算的微妙细节引发的。

计算机用几种不同的二进制表示形式来编码数值。

通过直接操作数字的位机，表示我们得到了几种进行算术运算的方式理解，这些技术对于理解编译器产生的机器及代码是非常重要的，编译器会试图优化算术表达式求值的性能。

我们对这部分内容的处理是基于一组核心的数学原理的，从编码的基本定义开始，然后得出一些属性，例如可表示的数字的范围，它们的位级表示以及算术运算的属性，我们相信从这样一个抽象的观点来分析这些内容对你来说是很重要的，因为程序员需要对计算机运算与更为人熟悉的整数和实数运算之间的关系有清晰的理解。

关于C语言历史

C编程语言是贝尔实验室最早开发出来的，目的是和Unix操作系统一起使用，在那个时候大多数系统程序，例如操作系统，为了访问不同数据类型的低级表示，都必须大量的使用汇编代码，比如说像malloc函数提供的内存分配功能，用当时的其他高级语言是无法编写的。

他们的书记录了最初贝尔实验室的C语言版本，随着时间推移经过多个标准化组织的努力，C语言也在不断的演变，1989年美国国家标准协会下的一个组织推出了ASIC C标准，对最初的贝尔实验室的C语言做出了重大修改，它的这个C与原来贝尔实验室的C有很大的不同，尤其是函数声明的方式。

国际标准化组织接替了对C语言进行标准化的任务，在1990年推出了ISO C90，在这一版本中引入了一些新的数据类型，对使用不符合英文语言字符的文本字符串提供了支持更新的版本，2011年得到批准，其中再次添加了更多的数据类型和特性，最近增加的大多数内容都可以向后进入，这意味着根据早期标准至少可以回溯到ISO C90编写的程序，按新的标准编译时会有同样的行为。

根据ISO C11来编译程序，prog.c，我们就使用命令行。

Gcc -std=c11 prog.c

它会根据ANSI或者SOC90标准来编译程序，C90有时候也称为C89执行为它的标准化工作是从1989年开始的。如果使用GL89进行编译的gcc选项可以是无，或者-std=gnu89。

## 2.1 信息存储

大多数计算机使用8位的快或者字节作为最小的可选择的内存单位，而不是访问内存中独立的位机器及程序，将内存输入一个非常大的字节数组称为虚拟内存。内存中的每个字节都有一个唯一的数字来标示成为他的地址，所有可能地址的集合就成为虚拟地址空间。顾名思义，这个虚拟地址空间只是一个展现给机器以及程序的概念性印象，实际的实现是将动态随机访问存储器，闪存磁盘存储器，特殊硬件和操作系统软件结合起来，为程序提供一个看上去统一的字节数组。

接下来我们将讲述编译器和运行时，系统是如何将存储器空间划分为更管可管理的单元来存放不同的程序对象，即程序数据，指令，控制信息。

例如C语言中的一个指针的值都是某个存储块的第1个字节的虚拟地址，C编辑器还把每个指针和类型信息联系起来，这样就可以根据指针值的类型存存成不同的机器及代码来访问存储在指针所指向位置处的值。尽管C编辑器维护着这个类型信息，但是它生成的实际机器级程序并不包含关于数据类型的信息，每个程序对象可以简单地视为一个字节块，而程序本身就是一个字节序列。

2.1.1 16进制表示法

一个字节由8位组成，使用两种符号方法对他描述位模式来说，都不是非常方便，二进制表示法太冗长，而十进制表示法与位模式的互相转化很麻烦，替代方法是以16为基数采用16进制数来表示位模式。

2.1.2 字数据大小

每台计算机都有一个自尝指明指针数据的标称大小，因为虚拟地址是以这样一个字来编码的，所以字常决定的最重要的系统参数就是虚拟地址空间的最大大小，也就是说对于一个资产为W位的机器而言，虚拟地址范围为0至2的W次方减一，程序最多可访问2的W次方个字节。

32位字长限制，虚拟地址空间为4000兆字节，写作4GB。也就是说刚刚超过4×10的9次方字节扩展到64位字节得到的虚拟地址空间为16EB，大约是1.84×10的19次方字节。

2.1.3 寻址和字节顺序

对于跨越多字节的程序对象，我们必须建立两个规则，这个对象的地址是什么？以及在内存中如何排列这些字节，在几乎所有的机器上，都自己的对象都被存储为连续的字节序列，对象的地址为所使用字节中最小的地址。

某些机器选择在内存中按照从最低有效字节的最高有效字节的顺序存储对象，而另一些机器则按照从最高有效自觉到最低有效字节的顺序存储前一种规则，最低有效字节在最前面的方式称为小端法，后一种规则：最高有效字节在最前面的方式称为-大端法。

不过有时候字节顺序会成为问题，首先是在不同类型的机器之间，通过网络传送二进制数据时，一个常见的问题是当小端法机器产生的数据被发送到大端法机器或者反过来时接收程序会发现字里的字节变成了反序的，为了避免这类问题。网络应用程序的代码编写，必须遵循以建立的关于字节顺序的规则，以确保发送方机器，将它的内部表示转换为网络标准，而接收方机器则将网络标准转换为它的内部表示。

第2种情况是当阅读表示整数数据的字节顺序，十字节顺序也很重要，这通常发生在检查机器级程序时。

返回编辑是一种确定可执行程序文件所表示的指令序列的工具。当前程序计数器的值即为下一条将要执行指令的地址。当阅读像此类小端法机器生成的机器级程序表示时，经常会将字节按照相反的顺序显示。

字节顺序变得重要的第3种情况是，当兵器规避正常的类型系统的程序时，在C语言中可以通过使用强制类型转换或联合来允许一种数据类型引用一个对象，而这种数据类型与创建这个对象时，定义的数据类型不同，大多数应用编程都强烈不推荐这种编码技巧，但是他们对于系统级编程来说是非常有用的，甚至是必须的。

参数12345的float数据的字节，和int的数据的字节是完全不同的，也就是说指针值却是完全不同的，这说明不同的机器或操作系统配置使用不同的存储分配规则。

使用printf格式化输出:

百分号C是输出一个字符，指定确定大小数据类型的格式，如int32\_t要更复杂一些。

可以观察到尽管浮点型和整数型数据都是对数值12345的编码，但是他们有截然不同的字节模式：整型为0x00003039，而浮点数为0x46401400。一般而言，这两种格式使用不同的编码方法，如果我们将这些16进制格式拓展为二进制格式，并且将它们移位，就会发现一个有13个相匹配的位的序列。

数据类型void \*是一种特殊类型的指针，没有相关联的类型信息。

这里给出的这些强制类型转换不会改变真实的指针，他们只是告诉编译器，以新的数据类型来看待被指向的数据。

对Linux操作系统可以使用man ascii来获得一张ascii字符码的表。

2.1.4 表示字符串

C语言中，字符串被编码为一个以null字符结尾的字符数组，每个字符都由某个标准编码来表示，最常见的是ascii字符编码，因此如果我们以参数12345和6(包括终止符，而strlen不包括终止符)来运行show\_bytes，我们得到最终结果为31 32 33 34 35 00。请注意是进制数字x的ASCII码正好是0x3x(比如1的ascii码值为0x31)，而最终终止之字节的16进制表示为0x00。在使用ASCII作为字符码的任何系统上，都将得到相同的结果，与字节顺序和字大小规则无关，进而文本数据比二进制数据具有更强的平台独立性。

ASCII字符集适合于编码英语文档，但是在表达一些特殊字符方面并没有太多方法基本编码，称为Unicode的统一字符集，使用32位来表示字符，这好像需要要求文本串中每个字符要占用4个字节，不过可以有一些替代编码，常见的字符只需要一个或两个字迹，而不太常用字符需要多一些的字节数。

2.1.5 表示代码

其实是完全一样的进程，运行在不同操作系统上也会有不同的编码规则，因此二进制代码是不兼容的，二进制代码很少能在不同机器和操作系统组合之间移植。

计算机系统的一个基本概念就是从机器的角度来看程序仅仅只是字节序列，机器没有关于原始源程序的任何信息，除了可能有用来帮助调试的辅助表以外。

2.1.6 布尔代数简介

二进制值使计算机编码存储和操作信息的核心，布尔代数的运算二进制值1和0表示逻辑值true或者false而运算符~,&,｜,^分别表示逻辑运算not,and,all和exclusive-or。

我们可以将上述4个部位运算扩展到位向量的运算位向量就是固定长度为ω，由0和1组成的串。外向量的运算可以定义成参数的每个对应元素之间的运算。我们将。A&b定义为一个长度为ω的位向量，其中第i个元素等于ai&bi。

当考虑长度为ω的位向量上的&,|和~运算时，会得到一种不同的数学形式，我们称之为布尔还不尔还与整数运算有很多相同的属性，例如整数运算的每个值X都有一个加法，逆元而不耳环也有类似的属性，这里的“加法”运算是^，不过这时每个元素的加法逆元是他自己本身。

卫星聊一个很有用的应用，就是表示有限集合，我们可以用位向量[aω-1,…,a1,a0]编码任何子集A⊆｛0,1,…,ω-1｝，其中ai=1当且仅当i∈A。使用这种编码集合的方法布尔运算｜和&分别对应于集合的并和交，而~对应于集合的补。

在大量实际应用中，我们都能看到用位向量来对集合编码。例如第8章，我们会看到很多不同的信号会中断程序执行，我们能够通过指定一个位向量掩码，有选择的。使能或是屏蔽一些信号其中某一位位置上，为1时表明该信号是有效的，只为0表明该信号是被屏蔽的。因而这个掩码表示的就是设置为有效信号的集合。

2.1.7 C语言中的位级运算

C语言的一个很有用的特性，就是它支持按位布尔运算。这些运算能够运到任何整形的数据类型上。正如事例说明的那样确定一个位级表达式结果最好的方法就是将16进制的参数扩展成二进表示并执行二进制运算，然后再转换为16进制。

位移运算的一个常见用法就是实现掩码运算，这里掩码是一个位模式，表示从一个字中选出的位的集合。掩码0xFF(最低的8位为1)表示一个字的低位字节。

2.1.8 C语言中的逻辑运算

C语言还提供了一组逻辑运算符||,&&和!,分别对应命题逻辑中的or,and和not运算。逻辑运算很容易和位移运算相混淆，但是它们的功能是完全不同的。逻辑运算认为所有非0的参数都表示true，而参数零表示false。他们返回1或者0，分别表示结果为true或者false。

也就是说对于非0的表达式，它的实际值就是true，对于林代表的是它的实际值就是false。对于逻辑运算表达式，它和位级运算的重要区别是，如果对第1个参数求值就能。确定表达式的结果，那么逻辑运算符就不会对第2个参数求值。

2.1.9 C语言中的移位运算

C语言中还提供了一组移位运算，向左或者向右移动位模式。 Exo向左移动客为丢弃最高的客位，并在右端补K歌苓一位运算是从左至右可结合的。遵循结合率。

有一个相应的右移运算x>>k，但是他的行为有点微妙，一般而言机器支持两种形式的右移：逻辑右移和算数右移。逻辑右移在左端补K个0得到的结果，算术右移是在左端补K个最高有效位的值，得到结果。这种做法看上去可能有点奇特，但是我们会发现它对有符号整数数据的运算非常有用。

C语言标准并没有明确定义，对于有符号数应该使用哪一种类型的右移，算术右移或者逻辑右移。不幸的，这就意味着任何假设一种或者另一种右移形式的代码，都可能会遇到可移植性问题。哦。实际上几乎所有的编译器机器组合都对有符号数使用算术又一且许多程序员也都假设机器会使用这种友谊，另一方面对于无符号数右移必须是逻辑的。

与C相比，Java对于如何进行右移有明确的定义表达式x>>k会将X算术右移k个位置，而x>>>k会对x作逻辑右移。

补充：

1.移位k位时k很大:

C语言标准很小心的规避了，这种情况下该如何做，在许多机器上当移动一个ω位的值时，移位指令只考虑位移量的低log2ω位，因此实际上位移量就是通过计算k mod ω得到的。另一方面，Java特别要求位移数量应该按照求模的方法来计算。

2.与移位运算有关的操作符优先问题

在C语言中，加法和减法的优先级比一位运算要高，然后按照从左至右结合性规则。在C语言表达式搞错优先级是一种常见的程序错误原因，而且常常很难检查出来，所以当你拿不准的时候，请加上括号。

## 2.2 整数表示

在本节中我们描述用蔚来编码整数的两种不同的方式，一种只能表示非负数，而另一种能够表示负数，0和正数，后面我们将会看到他们在数学属性和机器级实现方面密切相关。我们还会研究扩展或者收缩一个以编码整数，以适应不同长度表示的效果。

2.2.1 整型数据类型

C语言支持多种整形数据类型表示有限范围的整数。其中还给出了典型32位和64位机器的取值范围，每种类型都能用关键字来指定大小，这些关键字包括char, short, long，同时还可以，只是被表示数字是非负数(声明为unsigned)或者可能是负数(默认)。大多数64位机器使用8个字节的表示，比32位机器上使用的4个字节表示的取值范围大很多。要表中可以看出唯一一个与机器相关的取值范围的大小指示符是long。

C和C++都支持有符号(默认)和无符号数。Java只支持有符号数。

2.2.2 无符号数的编码

无符号数的二进制表示有一个很重要的属性，也就是每个介于0~2^ω-1之间的数有唯一一个ω位的值编码。

无符号数编码的唯一性，这是一个双摄。无符号数的一个，每一个整数都可以映射为一个唯一的长度为ω的位模式。

2.2.3 补码编码

对于很多，因为我们还希望有负数值，最常见的有符号数的计算机表示方式就是补码形式。在这个定义中，将字的最高有效位解释为负权。当负二位为一时表示值为负当设置为0，时值为非负。最高有效位xω-1也称为符号位，它的权重为2^ω-1。

拨码编码也具有唯一性，是一个双射。

正如我们会看到的补码的范围是不对称的，这导致了补码运算的某些特殊的属性，并且容易造成程序中细微的错误，之所以会有这样。的不对称性是因为一半的位模式表示负数，而另一半表示非负数，因为0是非负数也就意味着能表示的正数比负数少一个。第二最大的无符号数值刚好比补满了最大值的两倍大一点补码表示中所有表示负数的位模式在无符号表示中变成了正数。注意-1和UMax有相同的位表示,一个全1的串。

C语言标准并没有要求要用补码形式来表示有符号整数，但是几乎所有的机器都是这么做的，长时间如果希望大马就有最大可以这些能够在所有可能机器上运行，那么要做到这种表示范围内的任何可表示的数值范围，也不应该假设有符号会使用任何何种特殊的表示方式，另一方面许多程序的出血都假设使用补码来表示符号数，并且具有典型的取值范围，这些程序也能够在大多数机器和编译器上移植。

补充：关于确定大小的整数类型的更多内容

对于某些程序来说，用某个确定大小的表示来编码数据类型非常重要，比如当编写程序使得能够按照一个标准协议，在intel网上通信时，让数据类型与协议指定的数据类型兼容是非常重要的。

这些数据类型对应着一组宏定义了每个n的值值对应的最小和最大值，这些宏名字形如：int32\_t，他们如下：

确定宽度类型的带格式打印需要使用宏与系统相关的方式扩展为格式串。

当C预处理器遇到仅用空格或其他空白字符分隔的一个字符串常量序列时，就把它们串联起来。

练习题2.18，在第3章，我们将看到有返回编辑生成的列表，返回编辑是一种将可执行程序文件转换为可读性更好的ASCII码形式的程序。这些文件包含许多16进制数字，都是用典型的拨码形式来表示这些值，能够认识这些数字并理解它的意义(例如他们是正数还是负数)，是一种很重要的技巧。

2.2.4 有符号数和无符号数之间的转换

C语言允许在各种不同的数字数据类型之间做强制类型转换，例如假设变量x声明为int，u声明为unsigned。表达式(unsigned)会将x的值转换成一个无符号的数值，而(int)u将u的值。转换成一个有符号的整数。将有符号强制类型转换无符号数或者反过来会得到什么结果呢？可能想象到几种不同的规则，很明显对于在两种形式都能表示的值，我们是想要保持不变的，另一方面将负数转化为无符号数可能会得到0，如果转化为无符号数太大，以至于超过了补码能够表示的范围，可能会得到TMax。不过对于大多数C语言的实现来说，对这个问题的回答都是从位级角度来看的，而不是数的角度。

对-12345进行无符号变换得到的数字是53191，可以看到强制类型转换的结果，保持位置不变，只是改变了解释这些位的方式。-12345的16位补码表示与53191的16位无符号表示是完全一样的。将shot强制类型转化为unsigned short改变数值，但是不改变位表示。

另外对于32位资产来说，无符号形式的4294967295(UMax32)和补码形式的-1的位模式是完全一样的。 将unsigned强制类型转换成int，底层的位表示保存不变。

对于大多数C语言的实现，处理同样资产的有符号数和无符号数之间的相互转换的一般规则是数值可能会改变，但是位模式不变，

在X的补码表示中位xω-1决定了X是否为负。

当将一个有符号数应设为它相应的无符号数时，负数就会变成了大的正数，而非负数会保持不变。

2.2.5 C语言中的有符号数和无符号数

C语言支持所有整形数据类型的有符号和无符号，运算循环C语言标准没有指定，有符号数要采用某种表示，但是几乎所有的机器都使用不满，通常大多数数字都默认是有符号的，要创建一个无符号常量，必须加上后缀字符u或者U。

资源允许，无符号数和有符号数之间的转换，虽然C标准没有精确规定应如何进行这种转换，但大多数系统遵循的原则是底层的位表示保持不变。强制类型转换就会导致转换发生。

另外当一种类型的表达式被复制给另一种类型的变量时，转换是隐式发生的。

当用printf输出数值时，分别用指示符%d,%u,%x以有符号10进制，无符号10进制和16进制格式输出一个数字。注意printf没有使用任何类型信息，所以它可以用指示符%u来输出类型为int的数值，也可以用指示符%d输出类型为unsigned的数值。

游戏员对同时包含有货和无包数表达式的这种处理方式，出现一种奇特的行为，当执行一个运算时，如果它的一个运算数是无符号的，而另一个是无符号的，那么C语言就会隐式地将有符号参数强制类型转换为无符号数，并展示这两个数都是恢复的来执行这个运算。这种方法对于标准的算术运算来说并不大差异，但对于比较运算关系运算符来说，它会导致非直观的结果。比如-1<0u，因为第2个运算符是无符号的第1个运算数就会被隐式的转换为无符号数，因此表达式就等价于4294967295<0u(回想T2Uω(-1)=UMaxω)。这个答案显然是错误的。

网络旁注DATA:TMin C语言中TMin的写法

我们很小心的将TMin32写成-2147483648-1。为什么不简单的写成-2147483648或者0x80000000？看一下C头文件limits.h，注意到他们使用了各门写TMax32和TMin32类似的方法。

/\*Minimum and maximum values a signed int can hold.\*/

#define INT\_MAX 2147483647

#define INT\_MIN (-INT\_MAX-1)

不幸的是，补码表示的不对称性和C语言的转换规则之间奇怪的交互，迫使我们用这种不寻常的方式来写TMin32。虽然理解这个问题，需要我们专业C语言标准一些比较领会的角落，但是他能够帮助我们充分领会整数数据类型和表示的一些细微之处。

2.2.6 扩展一个数字的位表示

这样一个无符号数转为一个更大的数据类型，我们只要简单的在表示的开头添加0，这种运算被称为零扩展。

要将一个波霸数字转化为一个更大的数据类型，可以执行一个符号扩展在表示中添加最高有效位的值。

把short转换为unsigned时，我们先要改变大小之后再完成从有符号到无符号的转换。也就是说。(unsigned)sx等价于(unsigned)(int)sx，求值得到4294954951,而不等价于(unsigned)(unsigned short)sx，后者求是得到53191。事实上这个规则是C语言标准要求的。

2.2.7 截断数字

假设我们不用额外的蔚来扩展一个数值，而是减少表示一个数字的位数。当我们把int强制类型转换为short时。我们就将32位的int截断为了16位的short int。就像前面所看到的这个16位的位模式，就是-12345的补码表示。

截断一个数字可能会改变它的值溢出的一种形式，对于一个无符号数，我们可以很容易得出其数值结果。无佛术阶段只是对基础模操作下结果为0。

补码阶段也具有相似的属性，只不过要将最高位转变为符号位。

2.2.8 关于有符号数和无符号数的建议

就像我们看到的那样，有号数到无符号数的隐式强制类型转换，导致了某些非直观的行为，而这些非直观的特性经常导致程序错误，并且这种包含影视强制类型转换的细微差别的错误很难被发现，婴儿这种强制类型转换是在代码中没有明确指示的情况下发生的，程序员经常忽视了他的影响。

我们看到尤其是有符号数到无符号数的隐式转换，会导致错误或者漏洞的方式，避免这类错误的一种方法，就是绝不使用无符号数，实际上除了C以外，很少有语言支持无符号整数。很明显这种语言的设计者认为他们带来的麻烦比益处多得多。比如Java只支持有符号整数，并且要求以补码运算来实现。正常的右移运算符>>被定义为执行算术右移，特殊的运算符>>>被指定为执行逻辑右移。

地址自然是无符号的，所以系统程序员发现无符号类型是很有帮助的，当实验模运算和多精度运算的数学包时，数字是由字的数组来表示的，无符号值也会非常有用。

## 2.3整数运算

许多刚入门的程序员会非常惊奇的发现，两个正数相加会得到一个负数，而比较表达式x<y和x-y<0会产生不同的结果，这些属性是由于计算机运算的局限性导致的。

2.3.1 无符号加法

更常见的是编程语言支持固定精度的运算，因此像加法和乘法这样的运算不同于它们在整数上的相应运算。对于一个无复合术的加法，简单丢弃任何权重大于2^(ω-1)的位就可以计算出和模2^ω。比如X=9和X=12的未表示分别为1001和1100。它们的和是二十一五位的位表示为10101，但是如果丢弃最高位，我们就得到0101。也就是说十进制的五这就和值，21 mod 16=5一致。

算术运算溢出是指完整的整数结果，不能放到数据类型的字长限制中去。当执行C程序时不会将溢出作为错误而发信号，不过有的时候我们可能希望判断是否发生了溢出。

当计算结果的和小于其中任何一个加数时，就算溢出。

求x的加法逆元就是对无符号数求反，2^ω-1。

2.3.2 补码加法

对于补码加法我们必须确定，当结果(太大)为正或太小(为负)时应该做些什么。

两个数的ω为补码之和与无符号之和有完全相同的位级表示，实际上大多数计算机使用同样的机器指令来执行无符号或者有符号加法。

既然补码加法与无符号加法有相同的问题表示，我们就可以将其参数转化为无符号数直线，无符号数加法，再将结果转换为补码。即可以通过对运算数执行二进制加法，并将结果截断到4位，从而得到结果。

补码加法的正负溢出情况

当两个负数相加得到一个正数，则发生了负溢出，当两个正数相加得到一个负数，则发生了正溢出。

当y等于TMin时，我们也有-y=TMin。在函数任何测试过程中，TMin都应该作为一种测试情况。

2.3.3 补码的非

补码非的位级表示：

计算一个未知表示的值的补码非有几种聪明的方法。

第1种方法是对每一位求补，再对结果加1。 C语言中对任意整数值x，计算表达式-x和~x+1得到的结果完全一样。

计算一个数x的补码非的第二种方法。是建立在将位向量氛围两部分的基础之上的。我们将对K位左边的所有位取反。如何找这个K就是从右边数第1个非0的1就是这一个K。

2.3.4 无符号乘法

将X和Y的乘积截取ω位得到的就是乘法的结果。

2.3.5 补码乘法

C语言中有符号，乘法是通过将2ω位的乘积截断为ω位得到的。将一个补码数截断为ω位相当于先计算该值模2^ω，再把无符号数转换为补码。

2.3.6 乘以常数

以往在大多数机器上整数乘法指令相当慢，需要10个或者更多的时钟周期，而其他整数运算例如加法，减法，位级运算和移位只需要1个时钟周期。因此变压器使用了一项重要的优化，试着用移位和加法运算的组合来代替乘以常数因子的乘法。

左移一个数值，等价于执行一次与二的幂相乘的无符号乘法。

由于固定大小的补码算术运算的位级操作与其符号运算等价，我们就可以对博马运算的二的幂的乘法与左翼之间的关系进行类似描述。

由于整数乘法比移位和加法的代价要大得多许多C语言编译器试图以一位加法和减法的组合来消除很多整数乘以常数的情况。例如一个程序包含表达式x\*14。利用14=2^3+2^2+2^1,编译器会将乘法重新写为X左移三位加X左依两位加X左移一位，将一个乘法运算替换为三个移位和两个加法，无论X是无符号的还是驳马，甚至当乘法会导致溢出时，两个计算都会得到一样的结果。更好的是编译器，还可以利用属性14=2^4-2，将乘法重写为两个移位运算和一个减法。

2.3.7 除以2的幂

在大多数机器上，整数除法要比整数乘法更慢，需要30个或者更多的始终周期除以2的幂也可以用移位运算来实现，只不过我们用的是右移而不是左一。无符号和博码数，分别使用逻辑移位和算术移位来达到目的。

对无符号运算使用移位是非常简单的，部分原因是由于无符号数的右移，一定是逻辑右移。

对于除以2的me的驳马运算来说，情况要稍微复杂一些，首先为了保证负数仍然为负，移位要执行的是算术右移。

对于负数进行算术右移，当需要舍入时移位会导致结果向下舍入。我们可以通过在移位之前偏置这个值，来修正这种不合适的舍入。也就是说通过给X增加一个偏量y-1，然后再将除法向下舍入，当Y整除X时，我们得到Q，否则就得到Q+1。

现在我们看到除以2的幂，可以通过逻辑或者算术右移来实现，这也正是为什么大多数机器上提供这两种类型的，又一不幸的是，这种方法不能推广到任意常数同乘法不同，我们不能用除以2的幂的除法来表示除以任意常数K的除法。

2.3.8 关于整数运算的最后思考

正如我们看到计算机执行整数运算实际上是一种模运算的形式，表示数字的有限资产，限制了可能的值的取值范围，结果运算可能溢出，我们还看到补码表示提供了一种技能，负数也能正数的表示方法，同时使用了与执行无符号算数相同的位级实现,这些运算包括像加法，减法，乘法甚至除法，无论运算数是以无符号形式还是以博马形式表示的，都有完全一样或者非常类似的位级行为。

## 2.4 浮点数

8087是一种为8086处理器提供浮点支持的芯片。

IEEE电气和电子工程师协会。是一个包括所有电子和计算机技术的专业团体，他出版刊物，举办会议，并且建立委员会来定义标准，内容涉及从电力传输到软件工程。另一个IEEE标准的例子是无线网络的802.11标准。

我们将看到IEEE浮点格式中数字是如何表示的，我们还将是如探讨摄入的问题，即当一个数字不能被准确的表示为这种格式时，就必须向上调整或向下调整，然后我们将探讨加法乘法和关系运算符的数学属性，许多程序员认为福田说没意思，我们将看到，因为IEEE格式是定义在一组小而一致的原则上的，所以它实际上是相当优雅和容易理解的。

2.4.1 二进制小数

理解浮点数的第1步是考虑含有小数值的二进制数字。

数字权的定义与十进制小数点符号相关，这意味着小数点左边的数字的权是10的正幂得到整数值，而小数点右边的数字的权是10的负幂，得到小数值。

我们从导师中可以看到，二进制小数点向左移动一位相当于这个数被2÷，二进制小数点向右移动一位，相当于这个数×2。注意0.11111…(2进制)表示的是刚好小于1的数。我们将用简单的表示法1.0-ε表示。

如果我们仅考虑有限长度的编码，那么时间值表示法不能准确的表示，像1/3，5/7这样的数，类似小数的二进制表示法，只能表示那些能够被写成x\*2^y的数。不过我们并不能把它准确地表示为一个二进制小数，我们只能近似地表示他增加二进制表示的长度可以提高表示的准确度。

2.4.2 IEEE浮点表示

定点表示法不能很有效地表示非常大的数字，例如表达式5\*2^100使用101后面跟随100个零的位模式来表示。相反，我们可以通过给定X和Y的值来表示形如x\*2^y的数。

IEEE标准用V=(-1)^s\*M\*2^E的形式来表示一个数。

s表示符号。M表示尾数是一个二进制小数。E使解码对浮点数进行加权，这个权重是2的E次幂。

将浮点数的位表示划分为三个字段，分别对这些值进行编码

第1个单独的符号位S，直接符号编码S。

K位的阶码字段exp=ek-1ek-2…e1e0编码阶码E。

N位小数字段frac=fn-1…f1f0编码尾数M，但是编码出来的值也依赖于阶码字段的值是否等于0。

给定位表示根据exp的值，被编码的值可以分为三种不同的情况，最后一种情况有两个变种。

情况一，格式化的值

这是最普遍的情况，当exp的微模式既不全，为0也不全为一是都属于这类情况，在这种情况中，阶码字段被解释为以偏置量形式表示的有符号整数。也就是说阶码的值是E=e-Bias，其中e是无符号数，而Bias是一个等于2^(k-1)-1的偏置值。由此产生指数的取值范围，对于单进度是-126到+127，而对于双镜度是-1022到+1023。

小数字段frac被解释为描述小数值f，其中0≤f<1，其中二进制表示为0.fn-1…f1f0，也就是二进制小数点在最高有效位的左边。尾数定义为M=1+f。有时这种方式也叫做隐含的以一开头的表示，因为我们可以把M看成一个二进制表达式为1.fn-1fn-2…f0的数字。既然我们总是能够调整阶码E，使得尾数M在范围1≤M<2中(假设没有溢出)，那么这种表示方法是一种轻松获得一个额精度位的技巧。居然第1位总是等于1，那么我们就不需要显示地表示它。

情况二，非格式化的值

当阶码域全为零时，所表示的数是非格式化形式。在这种情况下，阶码值是E=1-Bias，而尾数值是M=f，也就是小数字段的值，不包含隐含的开头的1。

对于非格式化值为什么要这样设置偏置值？使阶码值为1\_Bias而不是-Bias似乎是违反直觉的，但这种方式提供了一种从非格式化值平滑转换到格式化值的方法。

非格式化数有两个用途，首先它提供了一种表示数制理论的方法，因为使用格式化术，我们必须总是使M≥1，因此我们就不能表示0。实际上，+0.0的浮点表示的位模式为全0:符号位是0(表明是一个非格式化的值)，而小数域也全为0，这就得到M=f=0。令人奇怪的是当负号位为一，而其他域全为零时，我们得到值-0.0。根据IEEE的浮点格式，+0.0和-0.0在某些方面被认为是不同的，而在其他方面是相同的。

非格式化术的另外一个功能是表示那些非常接近于0.0的数，他们提供了一种属性，称为逐渐下溢，其中可能的数值分布均匀的接近于0.0。

情况三：特殊值

最后一类特殊值适当指阶码全为一的时候出现的。当小数域全为0时，得到的值表示无穷。当s=0是是+∞，反之为负无穷。当我们把两个非常大的数相乘或者除以0时无穷能够表示溢出的结果，当小数誉为非零时，结果被称为“NaN”，即“不是一个数”(Not a Number)。一些运算的结果不能是实数或无穷就会返回这样的NaN的值，比如当计算根号负1无穷减无穷时，在某些应用中表示未初始化的数据，他们也很有用处。

2.4.3 数字示例

图展示了假定的8位浮点格式的例子，其中有K=4的阶码位和N=3的小数位，偏质量是2^(4-1)-1=7。图中分成了三个区域，还表示三类数字不同的列给出了，接码字段是如何编码阶码E的，小数字段是如何编码尾数M的，以及他们一起是如何形成要表示的值V=2^E\*M的。

可以观察到最大非规格化数7/512和最小规格化数8/512之间的平滑转变。这种平滑转变归功与我们对非规格化数E的定义。如果将E定义为1-Bias，而不是-Bias，我们可以补偿非规格化数的尾数没有隐含的开头的1。

这种表示具有一个有趣的属性，假如我们将图中的值的位表示解释为无符号整数，他们就是按升序排列的，就像他们表示的浮点数一样。这不是偶然的，IEEE格式如此设计就是为了浮点数能够使用整数排序函数来进行排序，当处理复数时有一个小的难点，因为它们有开头的一并且它们是按照降序出现的，但是不需要浮点运算来进行比较也能解决这个问题，参见家庭作业2.84。

2.4.4 舍入

因为表示方法限制了浮点数的范围和精度，所以浮点运算只能近似的表示实数运算。因此对于指X，我们想用一种系统的方法找到最接近的匹配值，它可以用期望的附点形式来表示，这就是收入运算的任务，一个关键问题是在两个可能值的中间确定舍入方向。

IEEE浮点格式定义了4种不同的舍入方式有4种手舍入方式。向欧摄入也被称为最像最接近的值舍入，这是默认的方式。其他三种方式产生实际值的切记，这种方法在数字应用中是很有用的，比如向零舍入方式，把正数向下舍入把负数向上舍入。

像偶数摄入初看是一个相当随意的目标，但计算平均值时，因为像偶涉入在实际情况中避免了这种统计误差，因为他在50%的情况下是向上舍入，在50%的情况下是向下舍入。

2.4.5 浮点运算

IEEE标准指定了一个简单的规则来确定注入加法和乘法，这样的算术运算的结果，把X和Y附近值看成实数运算定义在实数上计算产生结果，这是对实际运算的精确结果，进行舍入后的结果，在实际中浮点运算的设计者，是用一些聪明的小技巧来避免执行这样精确的计算，因为计算只要精确到能够保证得到一个准确的舍入结果就可以了。

IEEE标准中指定浮点运算行为方法的一个优势在于它可以独立于任何具体的硬件或者软件实现，因此我们可以检查它的抽象数学属性，而不必考虑它实际上是如何实现的。

浮点加法不具有结合性，这是缺少了最重要的群属性。浮点乘法也遵循通常乘法所具有的许多属性，这个运算在乘法中是封闭的（虽然可能产生无穷大或NaN），它是可交换的，而且它的乘法单元位为1.0。另一方面由于可能发生溢出或者由于摄入而失去精度，它不具有结合性单精度和双精度结果下不同，另外浮点乘法在加法上不具备分配性。

另一方面，对于任何a，b和c并且a,b,c都不等于NaN，附件乘法满足单调性。

另外我们还可以保证，只要。A≠NaN，就有a\*fa≥0。像我们先前所看到的无符号或补码的乘法，没有这些单调性属性。

对于科学计算程序员和编译器编写者来说，缺乏结合性和分配性是很严重的问题，即使为了在三维空间中确定两条线是否交叉而写代码，这样看上去很简单的任务，也可能成为一个很大的挑战。

2.4.6 C语言中的浮点数

所有的C语言版本提供了两种不同附件数据类型float和double。在支持IEEE浮点格式的机器上这些数据类型只有对应于单精度和双精度浮点，另外这类机器使用像偶数摄入的摄入方式，不幸的是因为C语言标准不要求机器使用IEEE浮点，所以没有标准的方法来改变收入方式，或者得到诸如-0，+∞，-∞或者NaN之类的特殊值。大多数系统提供include头文件和读取这些特征的过程库，但是细节随系统不同而不同，例如当程序文件中出现下列句子时，GNU编译器gcc会定义程序常数INFINITY(表示+∞)和NAN(表示NaN)：

#define \_GNU\_SOURCE 1

#include <math.h>

## 2.5 小结

计算机将信息编码为位(比特)，通常组织成字节序列，有不同的编码方式，用来表示整数实数和字符串不同的计算机模型，在编码数字和多字节数据中的字节顺序是使用不同的约定。

C语言的设计可以包容多种不同字长和数字编码的实现，64位字长的机器逐渐普及，由于64位机器也可以运行32位机器编译的程序，我们的重点就放在了区分32位和64位程序，而不是机器本身，64位程序的优势是可以突破32位程序具有4GB地址限制。

大多数机器对整数使用补码编码，而对浮点数使用IEEE标准754编码。在位级上理解这些编码，并且理解算术运算的数学特性，对于显然是编写的程序，能在全部数值范围上正确运行的程序上来说是很重要的。

在相同长度的无符号和有符号整数之间进行强制类型转化时，大多数C语言实现遵循的原则是底层的位模式不变，在补码机器上对于一个ω位的值 这种情况是由函数T2Uω和U2Tω来描述的， C语言隐式的强制类型转换会出现很多程序员无法预计的结果，常常导致程序错误。

由于编码的程度有限与传统整数和实数运算相比，计算机运算具有非常不同的属性，当超出表示范围时，有限长度能够引起数值溢出，当浮点数非常接近于0.0，从而转换为0时，也会下溢。

和大多数其他程序语言一样，C语言实现的有限整数运算和真实的整数运算相比，有一些特殊的属性，例如，由于溢出，表达式x\*x能够得出负数，但是无符号数和补码的运算都满足整数运算的许多其他属性，包括结合率，分配率交换率，这样就允许编译器做更多的优化。例如用(x<<3)-x取代表达式7\*x时，我们就利用了结合律，交换律和分配律的属性，还利用了移位和乘以二的幂之间的关系。

我们已经看到了集中使用位移运算和算术运算集合的聪明办法。例如使用补码运算，~x+1等价于-x。另外一个例子，一个前面前0后面前一的位模式有助于掩码运算，这种位模式能够通过C表达式(1<<k)-1生成，利用的是这样一个属性及我们想要的位模式的数值为2^k-1。例如表达式（1<<8）-1将产生位模式0xFF。

浮点表示通过将数字编码为x\*2^y的形式来近似的表示实数。最常见的浮点表示方式是由IEEE标准754定义的。它提供了几种不同的精度，最常见的是单精度32位和双精度64位。IEEE浮点也能够表示特殊值+∞，-∞和NaN。

必须小心使用浮点运算，因为浮点运算只有有限的范围和经度，而且并不遵循普通的普遍的属性，比如结合性。