CST 生存指南

15 CST Geno

October 16, 2025

版本 0.0（构建次数 683，22218 字）

前言

计算机科学与技术是一个涵盖了丰富内容的学科。

在常人认知中，“会计算机”经常与“会写程序”和“会修电脑”关联在一起，但是真正深入计算机科学与技术并不意味着止步于写一两个程序或者修一两台电脑。计算机科学与技术是一个包含了数学、逻辑、物理等学科内容的综合学科。

在现代社会，大多数人的日常生活与学习已经离不开计算机。有的人只是需要使用计算机来进行简单的文字处理，有的人使用它进行重复动作的自动化，有的人使用它支撑工作的流程。有的人非常熟悉计算机的各个方面，有的人则对计算机的了解只限于日常操作中涉及的内容。

本书将试图描述计算机方面的尽可能广的知识。虽然，这客观上会受到作者主观意识的限制，包括且不限于作者的知识范围、个人见解等。作者会试图使用更多资料来令内容更加有可信度，但是必然会不可避免地存在缺失、错漏的情况，请各位读者不吝斧正。

写作规范

本书使用 XeLaTeX 写成，格式、内容及表述遵循着一个 Geno 模仿其他排版规范而生造出来的不严谨的规范，在此列举如下：

1. 【文法】本书使用简体中文编写，文风将尽量与教科书类似，行文应符合现代汉语语法和语言习惯，避免欧化汉语表达。
2. 【结构】每一“章”（\chapter）围绕一个核心主题，在一“章”中展开若干“节”（\section），在“节”中可以展开“小节”（\subsection, \subsubsection, …）。
3. 【文件命名】每一节对应一个 `.tex` 文件，文件名为大驼峰（CamelCase）[1]形式的该节标题的英文翻译，并放置在以该节所在的章的大驼峰形式英文翻译的文件夹内。
4. 【章首】每一章均有一个 `Intro.tex` 文件，描述了该章的标题及内容摘要。
5. 【留白】中文与英文或数字之间需要 1 个半节空格[2]。
6. 【概念】首次出现的概念、术语，应使用简体中文表达，并给出一个英文翻译及直观的解释。
7. 【人名】对于国外人名，在正文中使用英文名称，在引用中使用人名原文。
8. 【书名】对于国外书名，在正文中使用简体中文翻译，在引用中使用书名原文。
9. 【脚注】对于附加的注释，使用 \footnote 进行脚注，和 / 或使用 \cite 进行引文。[3]

[1] 即每个单词均大写首字母，且单词与单词之间没有任何间隔符或连字符。
[2] 中文文案排版指北[1]
[3] 本条需细化。

10. 【引用】对于参考文献，使用形如 author-year-title 的格式，其中 author 为作者的简称，year 为发表年份，title 为标题首字母。且，当某作者只被引用一个文献时，year 与 title 均省略；当某作者在某一年只被引用一个文献时，title 省略。

目录

Chapter 0

课前学习

在学习计算机之前，我们有必要学习一些前置知识。这些前置知识并不会直接提升你的计算机技能，但是它们会在未来的各个方面细致入微地贯穿我们的学习生涯。

0.1 科学解决问题

在学习与生活的过程中，必然会遇到大量的问题。以计算机为例，从软件安装到环境配置，从代码运行到漏洞利用，从买卡装机到硬件开发，无时无刻不充满着大量的技巧、规范与约定。一旦对某个知识点不甚了解，就可能会出现大量的问题。

当遇到一个问题之后，我们将会选择放弃，或者努力地解决它。放弃会非常轻松，但是不断地往正确方向努力可以让你走向成功。本节将试图描述一些解决问题过程中的科学方法。

本文作者认为，“科学解决问题”包含了两个步骤，一个是问题的提出，一个是问题的解答。

0.1.1 提问的智慧

本节部分或全部借鉴自 Eric S. Raymond 的《提问的智慧》[2]，剩余的来自 Geno 的亲身实践。

总而言之，一个合格的问题，应至少包含以下几个部分：

1. 目标：“我准备做什么”；

2. 预期：“我希望看到什么”；
3. 现象：“我实际看到了什么”；
4. 尝试：“我试着做过什么（但并没有效果（不然也不需要提问题了））”。

0.1.2 回答的技巧

0.2 正确的坐姿

0.3 英文能力

由于一些历史渊源，英语确立了国际通用交流语言的地位，有大量的技术文档与标准、科研论文、报错日志、提示信息等均使用英文撰写。因此，英文能力是学习计算机，特别是深入学习计算机所必备的技能。

现代意义上的电子计算机一开始是美国在二战时期为了计算导弹弹道而制作的巨型计算器。美国和前苏联在二战和冷战期间均努力发展计算机，但美国发展速度更快，且抢先一步发展出了互联网，而我国大规模引入计算机及互联网时间均较晚。另外，作为二战与冷战时期的产物，早期的计算机硬件、操作系统及软件等在人机交互方面几乎仅考虑了制造国的通用语言，与语言相关的规定及限制甚至成为了标准的一部分。而为了让较老旧的设备也可以继续使用，一些新的标准会选择兼容旧标准，这导致了早期的带有语言限制的标准可能流传到现在。例如 ASCII 作为一个经典的通用编码只考虑了英文字母，甚至不包含一些拉丁字母的西欧变体，这导致大量的计算机技术文档、论文与标准、程序命令与操作系统提示等均使用了英文。

种种历史结合的结果造成了现在的局面：英语成为事实上的国际通用交流语言及互联网上的通用语。因此，如果要深入研究计算机的历史或者前沿方向，流畅的英语阅读能力是必不可少的技能。

Chapter 1

逻辑与推理

Geno 一开始是将 2 记数基础作为第一章的，但是后来认识到集合论、逻辑等是更加本质的东西，于是调换了顺序。

逻辑指的是客观事物的规律，推理则是利用总结出来的规律预言新的规律的过程。

逻辑推理是生物的本能，甚至是不具备高等智力的生物也存在以条件反射为基础的类似于简单逻辑推理的能力，因此逻辑推理是生物的本能，也是萌芽时间最早的学科之一。

现代科学中的理论大部分是建构在逻辑与推理上的，因此本书将首先讲解逻辑与推理。

1.1 三段论逻辑

古希腊著名逻辑学家 Aristotle 几乎是人类有流传记录历史上第一位系统分析并完整描述逻辑学观点的人，被认为是古希腊形式逻辑之父。他的逻辑学的核心是三段论逻辑（Syllogism Logic）。

他的六篇逻辑学著作被他的传人 Andronicus of Rhodes 汇总为《工具论》[4]。

4 [3]

Chapter 2

记数基础

计算机的基础是数学，数学的一个古老的问题就是数的记录与表示。因此我们要先认识一些日常生活中与计算机中会见到的记数系统，是为记数基础。

2.1 进位制

我们回想一下我们婴幼儿时候学习计数的样子:

伸出一只手的五根手指，弯曲一根手指，再弯曲第二根手指，接着是第三根、第四根、第五根手指，直到整只手变成一个拳头；为了继续计数下去，我们伸出了另一只手，并继续弯曲第六、七、八、九、十根手指。

2.1.1 十进制记数法

古人在一开始，也是使用这样的计数方式，这也是沿用至今的最方便的“随身携带”的简单计算工具，成语“屈指可数”也体现了这一点：较为简单的数就是使用“屈指”来计数的。但是随着生产力的发展，“十”已经不能满足要求了，于是祖先们在遇到“十”之后，便在地上放一块小石头，或者在绳子上打一个结，再重新使用双手计数。

由于进化的偶然，包括人类在内的灵长类动物均具有两只手，每只手上有五根手指。因此，历史上的不少文明不约而同地发展出了“逢十进一”的计数法，这被普遍认为是十进制的生物学起源。

使用手指来记数，有一些明显的缺点，比如说它无法记录。因此，我们创造出了十个汉字，即一、二、三、四、五、六、七、八、九、十，以记录

数字。当要记录的数比“十”大的时候，我们会继续叠加使用这些汉字，例如“十一”、“十二”，乃至“二十”，一直到“九十九”。在“九十九”之后，我们又需要新的汉字，即“百”，来表示十个“十”的数值。对于更大的数，我们还需要用于表达十个“百”的“千”、用于表达十个“千”的“万”，等等。

> 汉语中对于更大的数字的权重，传统上有亿、兆、京、垓、秭、穰、沟、涧、正、载十个名称。但对于这些名称具体表达的权重大小存在不同的定义，这也造成了现代汉语表达上的混淆。
>
> - 下数：每十倍使用下一个权重名称[5]，即 10 万为 1 亿 (10^6)，10 亿为 1 兆 (10^7)，10 兆为 1 京 (10^8)，依此类推。
> - 万进：每万倍使用下一个权重名称[6]，即 1 万万为 1 亿 (10^8)，1 万亿为 1 兆 (10^{12})，1 万兆为 1 京 (10^{16})，依此类推。
> - 中数：每万万倍使用下一个权重名称[7]，即 1 万万为 1 亿 (10^8)，1 万万亿为 1 兆 (10^{16})，1 万万兆为 1 京 (10^{24})，依此类推。
> - 上数：递归地使用更小的权重名称，直到用完才使用更大的权重名称[8]，即 1 万万为 1 亿 (10^8)，1 亿亿为 1 兆 (10^{16})，1 兆兆为 1 京 (10^{32})，依此类推。
>
> 另外，《中华人民共和国法定计量单位》又定义了“兆”为 10^6。[9]
>
> 因此，“兆”字在不同的上下文中会有不同的所指，需要根据语境及地区以确定其所表示的值。
>
> 另外，佛教中存在着表达更大的数的名称，递进规律使用了“上数”的定义[10]，但这些名称并没有现实用途。

而古印度人，创造出了九个符号，演化至今成为了最流行的“阿拉伯数字”，即 1、2、3、4、5、6、7、8、9。当要记录的数字比 9 大的时候，我

---

5 “十十变之”[4]卷上 12，“以十进”[5]卷一 15

6 “以万进”[5]卷一 15

7 “万万变之”[4]卷上 12，“万万曰亿，万万亿曰兆，……”[6]卷上 3

8 “数穷则变”[4]卷上 12，“以自乘之数进”[5]卷上 15

9 所表示的因数：10^6，词头名称：兆，词头符号：M[7]附件表 5

10 “百千百千名一拘梨。拘梨拘梨名一不变。……不可说转不可说转名一不可说转转。”[8]卷第二十九 1

们往“前”“进一”并使用新符号 0 进行占位。这个“进”导致相同的数码在不同的位置上会表示不同的意义，例如

100

的 1 表示一个“百”，而

10000

的 1 表示一个“万”。至于为何“进”的方向是向左而非向右，这只是约定俗成的原因。

“十进制记数法”（Decimal Notation），就是使用十个符号记录数字，并在记录超出“十”的范围的数字时，在不同的位置上使用这十个符号表征不同“权重”的值的记数系统。

2.1.2 通用的进位制记数法

“进位制记数法”（Positional Notation），或者称为“位值制记数法”（Place-Value Notation），就是将十进制的定义推广到十以外的任意的数上，使用若干个不同的符号记录数字，并且使用不同的位置表达同一个符号的不同“权重”的记数系统。

在进位制中，不同的位置上的数码，要乘以它所在的位的“权重”。对于十进制而言，这个“权重”就是我们通常所说的“个”、“十”、“百”、“千”、“万”等：当记录的数字达到 9 之后，下一个数“逢十进一”，将第二位增添 1，而当前位被清零，整个数变成了 10；同样，在第三位上，要等待第二位被第一位累积到 9 之后，再被第一位的 9 由于逢十进一而导致连进两位从而才增添 1。于是，在最右一位上的数字，代表的就是有多少个单位“1”；而它左边一位的数字，代表的是有多少个“十”，再往左一位的数字，代表有多少个“百”，或者说有多少个“十个十”。由此可以发现，某一位上的“权重”其实就是进位所需要的数的大小，自乘若干次的结果，而这个次数就是它与最右边一位的距离。因此，这个“进位所需要的数”，或者说“逢几进一”的“几”，是一个进位制记数法中的关键值。

“底数”，或者称为“基数”（Base），定义为一个进位制记数法中的可用数码的数量，同时也是一个“逢几进一”的“几进制记数法”中的“几”。

在有了通用的进位制的概念之后，我们可以非常容易地定义其它的进位制。

2.1.3 二进制

二进制（Binary），即以 2 为基数的进位制。现代意义上的以 2 为基数的进位制，是由 Gottfried Wilhelm Leibniz 受伏羲八卦图的影响在 1703 年

完成的论文[9]中提出的。由于二进制的每一位只有两种状态，因此使用若干个可以表达两种状态的物体即可表达一个二进制数，而电子电路恰好满足这个条件，例如电路的“通”与“断”，电压的“高”与“低”，等等，因此现代的计算机都以二进制作为电路层面最底层的数学基础。

根据前文所表达的“进位制”的概念，我们可以很容易地推出二进制中的“逢二进一”是如何应用到整数的表示中的：

0. 首先从 0 开始；
1. 下一个数为 1；
2. 再下一个数即满足了逢二进一的条件，因而为 10；
3. 下一个数为 11；
4. 再下一个数时在第二位和第一位上均满足了“逢二进一”，因而连续进两位而成为 100；
5. ……

可以看出，二进制虽然状态数简单，但是表达出来的长度随着所要表达的数的增大而快速增长，因而二进制的表达通常过于冗长。就此我们需要一些二进制的衍生进制。

2.1.4 二进制的衍生进制

最容易想到也几乎是唯一的二进制衍生方式，就是把一个二进制数进行分组，把连续的几位“合并”到一起，并使用新的数码来表示它们。

常见的二进制的衍生进制是八进制 (Octal) 和十六进制 (Hexadecimal)。八进制通过把二进制每三位压缩为一位的方式实现合并，而十六进制则是每四位合并为一位。

顾名思义，八进制逢八进一，需要八个数码。为了复用已有的符号，一般八进制使用十进制所使用的前八个符号，即 0 到 7 八个数码。

而十六进制，需要十六个数码，通常的十进制并不能满足需要，因此需要额外使用六个符号。在计算机的历史上，有过多种不同的符号选择方式：

设备型号	10	11	12	13	14	15
IBM System/360[11]	A	B	C	D	E	F
Monrobot XI[12]	S	T	U	V	W	X
DATAmatic 1000[13]	B	C	D	E	F	G
LGP-30[14]	f	g	j	k	q	l
PDS 1020[15]	L	C	A	S	M	D

Table 2.1: 历史上出现过的不同的十六进制数码[16]

后来，随着 IBM 在商业上的成功与其他厂商的式微，IBM 所使用的以 ABCDEF 作为十六进制的额外的六个符号的方式成为了事实上的十六进制表示的标准。

2.2 进位制的表示

显然，同一个数在不同的进位制下会有不同的表示，相同表示的数在不同的进位制下也会表示不同的值。因此，我们在讨论不同进位制的时候，需要对进位制做出表示，于是出现了进位制的记号或者说表示方式的问题。

2.2.1 记号表示

数学上的各种记号是一个自洽且略显混乱的系统，记号通常由发现或者试图简化某些运算的数学家的想象力、学习的语言数量及该记号所准备表达的含义所决定；而在计算机方面，记号通常会受到输入输出设备及系统的限制。因此，在数学和计算机（编程）方面的记号，会有所不同。

- 数学上较为常见的记号是，将数直接写出，然后将底数作为下标附在

11 [10]APPENDIX E: HEXADECIMAL-DECIMAL NUMBER CONVERSION TABLE

12 [11]Sexadecimal System，有文章指出这样的奇怪排布是出于穿孔卡片方便[12]，但是 Geno 并没有找到这个型号的穿孔卡片的样子。

13 “Similarly, ”hexadecimal” is used to denote any of the sixteen characters (ten decimal digits and six special symbols called hex B, C, D, E, F, and G)...”[13]Programmer's Language: Constants，不使用 A 的原因可能是为了避免与下文说明的使用 A 作为 Alphanumeric constant 的声明相混淆。

14 [14]Hexadecimal Digits.，很可能是把英文字母表去掉用于命令的字母[14]Summary of Orders[15]The Command Table，再去掉诸如数字 0 与大写英文字母 O、数字 1 与小写英文字母 l 等容易混淆的字母之后剩下的列表。

15 [16]Table of Printout Abbreviations，似乎是使用 LDS、CPY、ADD、SUB、MPY、DIV 六个可以在键盘[16]Data Flow In The PDS 1020 Computer 上直接输入的指令的首字母。

16 [17]The Early Days of Hexadecimal

其后。例如，二进制的 1010 记为

$$1010_2$$

。有时则会将数或基数使用小括号包裹，例如

$$1010_{(2)}$$

或

$$(1010)_2$$

。

- 程序语言方面，在不同的程序语言与场合中有不同的约定，并且大多数时候仅指定了十进制、十六进制和八进制的记号，少数时候指定了二进制的记号，少见可以指定任意进制的记号。最为常见的记号来自 C 语言，指定了十六进制前缀 `0x` 和八进制前缀 `0`，另外还有一些例如 `#`、`x` 等十六进制前缀。有时需要根据上下文才能确定具体进制的基底，但是在程序语言方面大多数时候看到的是十进制或十六进制。

2.3 二进制的计算机表示

我们需要找到一种可以只用二进制表示各种数，并且直觉上符合基本的四则运算的二进制数的记录方式，才能满足现代的二进制计算机的要求。另外，由于存储空间是有限的，因此我们要考虑的记录方式的长度也是有限的：在计算机的现实中，我们并不存在数学那样的理论上可以无限添加数字长度的简易记录方式（因此，这些记录方式会有一个“能表达的数的范围”的概念，以后我们将称之为“数据范围”。相关的内容将在后续章节中再详细描述，这里暂时忽略）。

2.3.1 二补数

在本节中，为了讨论的方便，我们暂不讨论小数（同样，后续的章节会介绍小数的记录方式），只考虑一个 8 位的存储整数的设备（即，必须而且只能使用 8 位二进制来表示，若超过 8 位则将其截断，只保留最右边的 8 位）。

我们非常容易想到的记录方式是，直接将要记录的数字转成二进制，比如说将 1 表达为 0000 0001，将 2 表达为 0000 0010 等，但是这种表达方式的弊端也非常明显：它无法记录负数。为了记录负数，我们首先要经过一段探索的历程，以取得一种符合期望的记录方式。

总的来说，我们希望它：

1. 与现有的二进制记录方式尽可能兼容；
2. 互为相反数的两个数相加结果为 0；
3. 减去一个数，等于加上这个数的相反数；
4. 表达结果直接相加减，等于相加减结果的表达；
5. 不断“加 1”之后能走遍一个范围内的全部数，并且在溢出截断之后再次加“回”到某个数的时候，其所表达的数值不变。

我们将我们所期望的记录方式分为两部分，一部分是表达正数的规范，另一部分是表达负数的规范，并分别定义：

- 首先，借鉴我们现有的把正负号放在数的最开头的写法，我们任意地定义最左边的一位为符号位，并且任意地定义 0 表示正数，1 表示负数，剩下的位用来表达实际的数。
- 然后，我们定义，如果表达的是正数，那么“实际的数”这一节直接采用所欲表达的数的二进制形式。
- 剩下的就是如何定义负数的表达了。

利用最后一条规则，我们可以观察从 1 连续减去两个 1 以到达 -1 的过程，以进行如下的思考：

二进制 0000 0001 减去 0000 0001 之后得到了 0000 0000，再减去 0000 0001 时就会发生“不够减”的情况，这便让我们联想到小学时学减法所学到的方式：可以往“前”去“借 1”。但是对于这个 0000 0000，“借 1”会往前一直借到开头，我们便虚空补上一个 1 用来借，变成 1 0000 0000，减掉 0000 0001 之后，我们就得到了 1111 1111，看起来是 -1 的某种表示。

> 由于我们一开始假定了一个 8 位的设备，所以在设备之外的第 9 位上发生的事情其实与设备无关，这个虚空补上的 1 并不会实际影响计算，所以我们可以放心地补上一个虚空的 1。

我们可以继续这个过程，得到 -2 的表示 1111 1110，-3 的表示 1111 1101……，一直到 -128 的表示 1000 0000。

> 再减下去就变成 0111 1111 了，它以 0 开头，根据前面的期望定义，它是正数 127，已经被定义了，我们不能搞出冲突的定义，所以负数的部分到 1000 0000 为止。

现在，我们有了一个负数的二进制表示的一一对应的方式，然后试图通过观察总结一下它的规律：抛开“负数的记录方式”这个框架的话，对于一个正数 x 而言，它的负数 $-x$ 的二进制表示与 x 的二进制表示相加的结果恰好都是 1 0000 0000，亦即一个负数 $-x$ 的二进制表示是 1 0000 0000 减去其绝对值的二进制表示的结果。这里的 1 0000 0000 在数值上恰好是 2^8，而 8 是前文我们指定的设备位数。就此，我们便摸索出了一套能够表达正数与负数的二进制表示方式。

接下来，我们看看这一套二进制表示方式能否符合前文描述的要求：

1. 与现有的二进制记录方式尽可能兼容：能，正数部分直接使用，负数部分使用 2 的设备位数次方减去去绝对值；
2. 互为相反数的两个数相加结果为 0：能，在 8 位的设备上，相加结果是 1 0000 0000，截断之后是 0；
3. 减去一个数，等于加上这个数的相反数：能，在 8 位设备上，加上一个数的相反数，等于加上 1 0000 0000 减去这个数的差，由于“加上 1 0000 0000”在 8 位设备上等同于加 0，对结果无影响，因此加上一个数的相反数等于减去这个数；
4. 表达结果直接相加减，等于相加减结果的表达：能，上一条可以将这一条合并成二进制的加法，而二进制的加法符合这一条要求；
5. 不断“加 1”之后能走遍一个范围内的全部数，并且在溢出截断之后再次加“回”到某个数的时候，其所表达的数值不变：需要额外定义补充，在 0111 1111 加 1 时，需要补充一条规则“满 128 减 256”，在 1111 1111 加 1 时的行为符合要求。

由此，我们得到了一个行为符合预期的可以记录 -2^k 到 $2^k - 1$ 范围内的整数的 k 位二进制记录方式[18]：

1. 左起第一位为数符，并定义 0 表示正，1 表示负；
2. 剩下的 $k - 1$ 位为尾数，并定义，当记录正数时，记录数的二进制表示；当记录负数时，记录数加上 2^k 的结果。

前面，我们在表达负数 $-a$ 的时候，提及了一个与它对应的数 $2^k - a$。事实上，由于我们已经定义好一个整数二进制记录方式，并且我们已经证明它符合一条定理“减去一个数，等于加上这个数的相反数”，因此这个计算方式其实对正数也适用，因此我们可以将这个数的概念泛化。

类比于在十进制下我们把 9 结尾的数与 1 补足到整十，由于 $2^k - a$ 恰好能够与 a 补整到 2^k，因此我们把 $2^k - a$ 称为 a 的“以 2 为基底的补足数”，简称为二补数（2's Complement）。

2.3.2 一补数

前面指定了一种整数的二进制的记录方式，但是整数的二进制的记录方式并非只有一种：从形式的简洁上，我们可以很容易想到另外一种表达：

1. 全部位都用来记录欲表达的数的绝对值；
2. 如果欲表达的数是负数，那么每一位都取反（即，1 变成 0，0 变成 1）。

我们容易验证，这种表达方式也是符合大部分期望的：

1. 与现有的二进制记录方式尽可能兼容：能，正数部分直接使用，负数部分按位取反；
2. 互为相反数的两个数相加结果为 0：能，在 8 位设备上，互为相反数的两个数相加的结果是 1111 1111，按照刚刚的定义它是 -0。
3. 减去一个数，等于加上这个数的相反数：能，在 8 位设备上，加上一个数的相反数，等于加上 1 0000 0000 减去这个数的差，由于“加上 1 0000 0000”在 8 位设备上等同于加 0，对结果无影响，因此加上一个数的相反数等于减去这个数；
4. 表达结果直接相加减，等于相加减结果的表达：能，上一条可以将这一条合并成二进制的加法，而二进制的加法符合这一条要求；
5. 不断“加 1”之后能走遍一个范围内的全部数，并且在溢出截断之后再次加“回”到某个数的时候，其所表达的数值不变：不符合，1111 1111 加 1 之后的结果是 0000 0000，但其对应的十进制表达 $(-0)+1=0$ 并不成立。

最后一条期望，这种表达方式并不满足。但是，如果补充一个定义，“如果发生了溢出，则把溢出值移动到最低位再相加”，那么它便能符合期望：1111 1111 加 1 之后的结果是 1 0000 0000，按照补充定义需要把 1 移动到最右边与 0000 0000 相加得到 0000 0001 才得出最终结果，即 $(-0)+1=1$ 的成立。

事实上，这种表达方式在负数方面只与二补数相差了 1：二补数使用了 2^k 来减去对应的数，而这种表达方式使用了 2^k-1，因此这种表达方式表达的负数的二进制，等于这个数的二补数的二进制减去 1，于是这种表达方式被称为“一补数”（1's Complement）。

从另一个方面也可以看出它命名为“一补数”的合理性：按照上面的定义，一个数与它的一补数相加，能得到一串 1，恰好能满足与二补数相

仿的“补足”的内涵（某一位上相加的结果恰好为进位制基底），而这里的“补足”的基底看起来是“1”。

2.3.3 原码、反码与补码[17]

我们任意地指定三个概念，原码、反码与补码。

原码指定，

- 最高位为符号位，并任意地指定 0 表示正数，1 表示负数；
- 剩下的位为数值位，记录准备表达的数的绝对值。

反码指定，当表达正数时，使用数的二进制表达；当表达负数时，使用一补数，即

- 最高位为符号位，0 表示正数，1 表示负数；
- 剩下的位为数值位，当表示正数时直接记录准备表达的数的二进制，当表示负数时记录准备表达的数的二进制的各位取反。

补码指定，当表达正数时，使用数的二进制表达；当表达负数时，使用二补数，即

- 最高位为符号位，0 表示正数，1 表示负数；
- 剩下的位为数值位，当表示正数时直接记录准备表达的数的二进制，当表示负数时记录准备表达的数的二进制的各位取反加 1 的结果。

由于使用补码表达的二进制数，可以在包括正数、负数和 0 范围内的整数中直接进行正确的加减法，因此，虽然一补码在历史上曾经有过较为广泛的运用，但现代主流计算机中均使用以二补码为基础的补码作为整数的二进制记录方式。

[17]这似乎是国内特有的概念。Geno 在进行了一定的探索后得出了这个结论，但暂时无法确定该结论是否正确。一补数、二补数的概念似乎只是反码、补码概念的子集：简体中文的文献经常提及“正数的反码、补码……，负数的反码、补码……”，但英文文献似乎仅提及“正数的相反数可以用二补数表示为……”，即简体中文文献将反码与补码视为一种独立的包含了正数、负数与 0 的记数方式，而英文文献仅把补数视为依赖于正数与 0 的一种非正数的记数方式。

2.4 不同进位制的转换

根据进位制的原理，我们可以计算出十进制数所对应的其他进位制的表示，也可以将一个其他进位制的表示转换为十进制。这样，借助我们熟悉的十进制，我们就能做到任意进位制的相互转换。

若源进位制与目标进位制的基数都是某一个数的整数次幂的话，我们也可以不借助十进制直接进行不同进位制的转换。另外，如果非常熟悉目标进位制的运算，那么直接转换也是可行的。

2.4.1 十进制转换为其他进位制

将一个十进制数转换为其他进位制有几种方法。整数部分可以使用短除法或者退位法，小数部分可以使用连乘法或者退位法，然后将两部分合起来即可。

整数的短除法

首先，将欲转换的整数除以目标进制的基底，记录余数在目标进制下的表示，并将商继续进行这种除法，直到商为 0 为止。最后将记录下的余数从最末到最初的顺序书写，即为该整数在目标进制下的表示。

例如，欲将十进制整数 13579 转换为十六进制，如 2.1 所示：

1. 将该十进制整数除以目标进制的基底（即 16），得到商 848，余数 11 转换为目标进制的结果为 B；
2. 商 848 继续除以 16 得到商 53，余数 0（整除）；
3. 53 除以 16 得到商 3，余数 5；
4. 3 除以 16 得到商 0，余数 3，短除法结束；
5. 将计算得到的余数从下往上读取一遍，即得到结果 $(350B)_{16}$。

整数的退位法

将目标进制的基底的乘方序列以十进制表示出来，然后从 1 开始往左写，直到比欲转换的整数大为止。然后将这一行数从左向右依次与欲转换的数比较并按需做减法，直到其被减到 0 为止：如果欲转换的数比较大的

$$
\begin{array}{rrcl}
16 \mid \underline{13579} & & & \\
16 \mid \underline{848} & 11 & \to & \mathrm{B} \\
16 \mid \underline{53} & 0 & \to & 0 \\
16 \mid \underline{3} & 5 & \to & 5 \\
0 & 3 & \to & 3
\end{array}
\quad \uparrow
$$

Figure 2.1: 短除法转换十进制整数为其他进制的示例

乘方序列	4096	512	64	8	1
结果	0	6	0	3	0
要减去多少	0	3072	0	24	
剩余	3096	24	24	0	

Figure 2.2: 退位法转换十进制整数为其他进制的示例

话，就将欲转换的数减掉被比较的数若干次，直到比被比较的数小，然后在下方写上减去的次数在目标进制下的表示；如果是被比较的数比较大的话，就跳过它，并在下方写上 0。如果欲转换的数已经被减到 0 了但是第一行数仍未结束，那么在剩下的数的下方均写上 0。最终得到的第二行数即为该数在目标进制下的表示。

例如，欲将十进制整数 3096 转换为八进制，如 2.2 所示：

1. 将目标进制的基底的乘方序列以十进制表示从 1 开始往左写，直到比这个十进制数大为止，得到数列 4096、512、64、8、1；

2. 然后将这一行数从左向右依次与这个数比较，第一个是 4096，比 3072 大，故在 4096 下方写 0；

3. 下一个数是 512，比 3096 小，尝试发现减去 6 次之后的结果 24 开始比 512 小，故在 512 下方写 6；

4. 下一个数 64 比 24 大，故跳过，在 64 下方写 0；

5. 下一个数 8 可以将 24 减去 3 次得到 0，故在 8 下方写 3，并结束循环；

6. 最后剩余一个数 1，在其下方写上 0；

7. 将第二行数从左向右读取，并舍弃开头的 0，即得到结果 $(6030)_8$。

		.3125
	×	2
0		.625
	×	2
1		.25
	×	2
0		.5
	×	2
↓ 1		.

Figure 2.3: 连乘法转换十进制小数为其他进制的示例

小数的连乘法

将小数乘以目标进制的基底，记录乘积的整数部分在目标进制下的表示，并将乘积的小数部分继续进行这个运算，直到小数部分为 0 为止，然后将记录到的整数部分从最初到最末的顺序书写，即为该小数在目标进制下的表示。

注意，将小数进行进制转换时，是可能出现“死循环”，即永远无法达到“小数部分为 0”的情况的。这是非常常见的情况，其实就像 $\frac{1}{3}$ 在十进制下表示为无限循环小数 $0.\overline{3}$ 而在三进制下是简单的 0.1 一样，此时转换结果会是一个无限循环小数或无限不循环小数。

> 10 分解质因数的结果是 2×5，只有 2 和 5 两个素因子，因此目标进制如果含有 2 或 5 之外的素因子，十进制下的有限小数就会在转换时变成一个无限小数。

例如，欲将十进制小数 0.3125 转换为二进制，如 2.3 所示：

1. 将小数乘以目标进制的基底（即 2），得到 0.625，整数部分为 0，小数部分为 .625；
2. 小数部分 .625 继续乘以 2 得到 1.25，整数部分为 1，小数部分为 .25；
3. .25 乘以 2 得到 0.5，整数部分为 0，小数部分为 .5；
4. .5 乘以 2 得到 1，整数部分为 1，小数部分为 .，连乘法结束；
5. 将计算得到的整数部分从上往下读取一遍，即得到结果 $(0.0101)_2$。

乘方序列	1	0.0625	0.00390625
结果	0	9	3
要减去多少	0	0.5625	0.01171875
剩余	0.57421875	0.01171875	0

Figure 2.4: 退位法转换十进制小数为其他进制的示例

小数的退位法

与整数的退位法类似，但是所需要的乘方序列是目标进制的基底的负整数次方的数列。

例如，欲将十进制小数 0.57421875 转换为十六进制，如 2.4 所示：

1. 将目标进制的基底的负数次方序列以十进制表示从 1 开始往右写，由于无法事先知道有多少位因此先写一位，得到数列 1、0.0625；
2. 然后从 1 的下一个数开始，试着减去足够多次的 0.0625 但不变成负数，将能够减去的最大次数 9 写于其下方，并记录减去之后的剩余 0.01171875；
3. 再计算一位乘方序列也就是 16^{-2}，得到 0.00390625；
4. 将刚刚的剩余 0.01171875 做这种试减，将能够减去的最大次数 3 写于其下方，减去之后的剩余为 0，故结束循环；
5. 将第二行数从左向右读取，并补上开头的 0.，即得到结果 $(0.93)_{16}$。

2.4.2 其他进位制转换为十进制

将一个非十进制的数转换为十进制，只需要将各位上的数与对应位的权重的积求和即可。

例如，欲将十六进制数 $(BAD.BEEF)_{16}$ 转换为十进制，如 2.5 所示：

1. 将欲转换的数的每一位都转为十进制；
2. 从小数点开始，往上是源基底的 0 次方并将指数递增，往下是源基底的 -1 次方并将指数递减，写出每一位的权重；
3. 每一位均乘以对应的权重，得到积；
4. 将积求和，即得到结果 2989.7458343505859375。

原数	十进制		权重		积
B	→11	×	16^2	=	2816
A	→10	×	16^1	=	160
D	→13	×	16^0	=	13
.	→.	×	.	=	0
B	→11	×	16^{-1}	=	0.6875
E	→14	×	16^{-2}	=	0.054 687 5
E	→14	×	16^{-3}	=	0.003 417 968 75
F	→15	×	16^{-4}	=	0.000 228 881 835 937 5
				+	
					2989.745 834 350 585 937 5

Figure 2.5: 转换十六进制数为十进制数的示例

源数		G	E	N	O	.	H	I	
中间表示		121	112	212	220	.	122	200	
重新分组	12	11	12	21	22	20	.	12	22	00
结果	5	4	5	7	8	6	.	4	8	0

Figure 2.6: 相同整数的不同幂次进位制之间简便转换的示例

2.4.3 存在乘方关系的不同进位制的简便转换

如果源进制的基底 k_s 和目标进制的基底 k_d 都是同一个整数 k 的整数次幂，那么可以简单地把欲转换的数以小数点为起点向左右逐位转换成 $log_k(k_s)$ 位 k 进制数，然后以小数点为起点向左右每 $log_k(k_d)$ 位分为一组，并将每组转换为一位目标进制的数，即可不经过十进制进行直接转换。

例如，欲将二十七进制数 $(GENO.HI)_{27}$ 转化为九进制，如 2.6 所示：

1. 检查源进位制与目标进位制的基底，发现源进位制的基底 $27 = 3^3$，目标进位制的基底 $9 = 3^2$，均为 3 的整数次幂，符合这个简便转换的条件；

2. 将欲转换的数逐位转化为中间表示，即 3 进制的数 $(121\ 112\ 212\ 220.122\ 200)_3$；

3. 将中间表示每 2 位 $(= log_3(9))$ 作为一组，从小数点开始向两边重新分组，得到 $(12\ 11\ 12\ 21\ 22\ 20.12\ 22\ 00)_3$；

4. 将每一组分别转为目标进位制的数，即得结果 $(545786.48)_9$，整数部分最高位和小数部分最低位的 0 可以省略。

2.5 定点数与浮点数

解决了整数的二进制记录问题之后，我们需要考虑小数的记录的问题。

我们可以参考在整数时使用的方式，直接记录小数的二进制，但是这会造成一个问题：由于二进制只有两个可用的记号 0 和 1，而且它们都已经被用来表达数字，因此已经没有记号可以用来在数码之间标记出小数点的位置且不造成歧义。显然，“小数点在哪里”这个问题有两种解决方案，一种是事先约定好小数点的位置，一种是划出一段空间用来记录小数点的位置。

2.5.1 定点数

对于一个有 k 位的设备，如果我们事先约定小数点的位置在 n 位与 $n+1$ 位之间的话，那么这个小数的表达方式就能总结为如下几点：

1. 左起第一位为符号位；
2. 左起第二位到第 n 位是整数部分的二进制记录；
3. 左起第 $n+1$ 位到第 k 位是小数部分的二进制记录。

此时它可以表达的数的范围是 $(-\underbrace{11\cdots11}_{n-1}.\underbrace{11\cdots11}_{k-n})_2 - (-0.\underbrace{00\cdots00}_{k-n}1)_2$ 和 $(0.\underbrace{00\cdots00}_{k-n}1)_2 - (\underbrace{11\cdots11}_{n-1}.\underbrace{11\cdots11}_{k-n})_2$，或者说 $-\frac{2^{(k-1)}-1}{2^{(k-n)}} - -\frac{1}{2^{(k-n)}}$ 和 $\frac{1}{2^{(k-n)}} - \frac{2^{(k-1)}-1}{2^{(k-n)}}$，分度值为 $(0.\underbrace{00\cdots00}_{k-n}1)_2$，或者说 $\frac{1}{2^{(k-n)}}$。

一个具体的例子如 2.2 所示。

对于以上的定义，

这种记录小数的方式，在一开始小数点的位置就被固定了，因此称为“定点数”（Fixed Point）：比如说，对于二进制小数 $(100.11)_2$ 来说，不管如何修改 k 和 n，这个数的小数点的位置总是在第二个 0 和第二个 1 之间，这个小数点的位置是固定不变的，这就是“定点数”一名的来源。

然而，这种记录方式有一个问题：精度与范围只能二选一。如果使用更小的 n，精度就能提高，但是所能表达的数的范围便急剧减小；如果提升表达的数的范围，就需要使用更大的 n，这导致了精度的降低。如果我们将所使用的权重画出来，就会是如图 2.7 所示的情况：

n	最小值	最大值	分度值
1	$-0.992\,187\,5$	$0.992\,187\,5$	$0.007\,812\,5$
2	$-1.984\,375$	$1.984\,375$	$0.015\,625$
3	$-3.968\,75$	$3.968\,75$	$0.031\,25$
4	-7.9375	7.9375	0.0625
5	-15.875	15.875	0.125
6	-31.75	31.75	0.25
7	-63.5	63.5	0.5
8	-127	127	1

Table 2.2: $k = 8$ 时 n 的值对应的数据范围

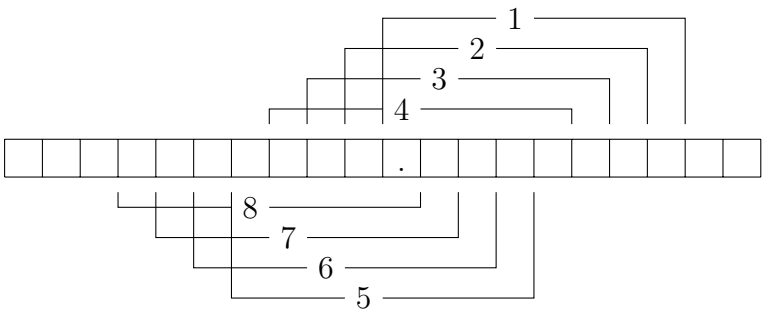

Figure 2.7: 存储位的位置

如果固定小数点的位置，那么我们便无法灵活地控制表达范围与精度。那么我们能否找到一种方式，可以让小数点的位置根据我们的数值的大小动起来，就此根据需要灵活地控制精度与表达范围呢?

2.5.2 浮点数

在记录小数时，为了解决定点数的缺点，我们希望

1. 能表达的数的范围要大一点;
2. 相对精度的数量级保持平衡。一个浅显的例子是，对于几千万上亿的数字，我们所期望的绝对精度必然不会跟零点零零几的一致，前者相差几十上百没有太大影响，而后者哪怕只是相差零点一都会造成核爆般的震动，因此我们并不需要保持绝对精度，我们只需要保持与这个数的数量级相符的“相对”精度。

基于这两个要求，我们可以在经过思考后发现，科学记数法（Scientific Notation）的思想似乎符合我们的愿望。

在平时，在遇到很大或者很小的数时候，我们通常使用科学记数法来描述这个数，以避免数 0 的烦恼。科学记数法使用形如 $a \times 10^b$ 的形式来记录一个数，并限定 $1 \leqslant a < 10$，b 为整数。例如，数 16000 可以用科学记数法记为

$$1.6 \times 10^4$$

，其中 $\times$ 号前面的尾数（Mantissa）a 记录了原本的数字，后面是一个乘方的形式，其中乘方的底数事实上指示了原数的进制，称为基（Base），乘方的指数（Exponent）指示了小数点的偏移。

这种记数法符合我们的愿望的理由是:

1. 能表达的数的范围要大一点：是的。科学记数法本就是为表达较大的数与较小的数而诞生的，要表达很大或者很小的数只需要修改指数即可，而不需要浪费非常大的空间来写 0，将 0 写在指数部分带来的表达范围的扩增比写在尾数部分大得多了。
2. 相对精度的数量级保持平衡：是的。乘方的形式无形中将相对精度的位置进行了适当的调整。例如，如果限制尾数保留小数点后三位，那么对于十亿一百万 1.001×10^9 来说，能够波动的最小误差是 0.001×10^9 即一百万，而对于零点零零一零零一 1.001×10^{-3} 来说，能够波动的最小误差是 0.001×10^{-3} 即百万分之一，不论原数的大小其最小误差均为原数数量级的千分之一，相对来说保持了平衡。

由此，我们确认借鉴科学记数法可以解决小数的二进制记录的问题。

在不加限制地使用这种记数法时，小数点在整个数中的相对位置是可以变化的，例如 16000 和 1.6×10^4 与 16×10^3 甚至 0.0016×10^7 都表示同一个数，因此使用了类似于科学记数法记录的二进制小数得名“浮点数”（Floating Point Number）。为了保证相同数字的表示形式的唯一性，我们借鉴科学记数法的定义，将尾数 a 满足 $1 \leqslant a < 10$ 的形式的表示形式指定为规格化（Normalized）[18]数，其他的称为非规格化（Denormalized）数。

由于现代通用的计算机均使用二进制，因此我们借鉴来的记录方式应该使用（十进制表示的）$a \times 2^b$，而且因为基固定为 2 因此可以省略，于是我们只需要记录 a 和 b 便可以记录这个数。并且，如果确保记录的浮点数必定为规格化数的话，二进制表示的 b 必定是 $\overline{1.b_1b_2\cdots b_n}$ 的以 1 开头的形式，因此 b 开头的 1 也可以省略。

当然，在我们使用相同的长度来表达一个小数的时候，定点数可以完全使用这一段长度来记录它，但浮点数需要在这段有限的长度中抠出一部分用于保存指数信息，这会损失一定的精度。但是，我们一方面以此换来了更大的数值范围，而且只要权衡好抠出的量的话损失少量的精度在大部分情况下是可以接受的。

2.6 浮点数的记录方式

在漫漫的计算机历史长河中，曾经出现过大量的浮点数记录方式。在被标准化之前，这些浮点数记录方式可以大致分成三类[19]：

1. 1 位符号位，若干位（加上定值以便于记录负数的）指数位，若干位尾数位；

2. 若干位（二补数记录的）指数位，若干位（二补数记录的）尾数位；

3. 1 位尾数符号位，若干位尾数位，1 位指数符号位，若干位指数位。

出现浮点数记录方式百花齐放的原因是，一方面当时也是一个各种计算机百花齐放的年代，另一方面当时的硬件并不非常发达，芯片制造商倾向于复用整数运算模块，而一些特殊的浮点数记录方式方便了在自家芯片上的复用。

[18]另有“规范化”、“规约化”、“规则化”、“正规化”等翻译。

2.6.1 IEEE 754

1985 年 3 月 21 日，电气电子工程师学会（IEEE）通过了《二进制浮点数算术的 IEEE 标准》[19]（IEEE 754）。该标准成为了浮点数表示的业界规范。

前置定义

IEEE 754 标准首先定义了一个浮点数规范所需的几个参数：

- p 是有效位数的数量，即精度；
- $E_{\max}$ 是所能记录的最大的指数值；
- $E_{\min}$ 是所能记录的最小的指数值；

接着定义了一个完整的浮点数规范所需包含的定义：

- 形如 $(-1)^s 2^E (b_0 \cdot b_1 b_2 \cdots b_{p-1})$[20]的数。其中：
  - s 是 0 或者 1；
  - E 是 $E_{\max}$（含）和 $E_{\min}$（含）之间的正整数；
  - b_i 是 0 或者 1。
- 两个无穷大，$+\infty$ 和 $-\infty$；
- 至少一个会触发异常信号（Signaling）的 NaN[21]；
- 至少一个不会触发异常信号（Quiet）的 NaN。

由此，一个浮点数格式包含以下三部分：

- 1 位符号位 s；
- 偏移指数（Biased exponent）e，等于 E 加上一个指定的偏移；
- 分数（Fraction，其实也就是上文所说的“尾数”）f，等于 $\cdot b_1 b_2 \cdots b_{p-1}$。

[19] IEEE 754–1985 IEEE Standard for Binary Floating-Point Arithmetic[20]

[20] “$b_0 \cdot b_1$” 中的 “$\cdot$” 是小数点，而非乘号。这里括号里表示的是数码直接写在一起的连接，类似于数学记号上的 $\overline{b_1 b_2 \cdots b_n}$，而非连乘法。下同。

[21] Not a Number 的简写。

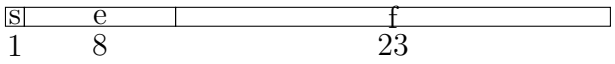

Figure 2.8: IEEE 754 单精度浮点数示意图

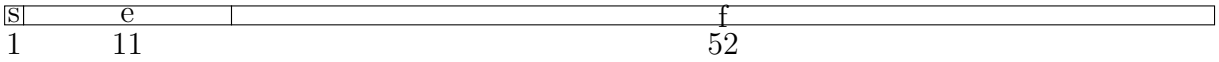

Figure 2.9: IEEE 754 双精度浮点数示意图

基础格式

IEEE 754 在如上的基础上，定义了两种浮点数：单精度（Single）浮点数和双精度（Double）浮点数。

单精度浮点数由 32 位二进制数组成，从左到右分别是 1 位符号位 s，8 位偏移指数 e 和 23 位分数 f，如图 2.8 所示，

并定义：

1. 如果 e 是 255 且 f 不为 0（形如 s 11111111 $bbbb\cdots bbbb$），那么其表示的值为 NaN；
2. 如果 e 是 255 且 f 为 0（形如 s 11111111 $0000\cdots 0000$），那么其表示的值为 $(-1)^s\infty$（即正负无穷大）；
3. 如果 e 在 0（不含）和 255（不含）之间（通常情况），那么其表示的值为 $(-1)^s 2^{e-127}(1\cdot f)$；
4. 如果 e 为 0 但 f 不为 0（形如 s 00000000 $bbbb\cdots bbbb$），那么其表示的值为非规格化浮点数 $(-1)^s 2^{-126}(0\cdot f)$；
5. 如果 e 和 f 均为 0（形如 s 000000000 $0000\cdots 0000$），那么其表示的值为 $(-1)^s 0$（即正负 0）。

双精度浮点数由 64 位二进制数组成，从左到右分别是 1 位符号位 s，11 位偏移指数 e 和 52 位分数 f，如图 2.9 所示，

并定义：

1. 如果 e 是 2047 且 f 不为 0（形如 s 11111111111 $bbbb\cdots bbbb$），那么其表示的值为 NaN；

参数	单精度	单精度扩展	双精度	双精度扩展
p	24	$\geqslant 32$	53	$\geqslant 64$
$E_{\max}$	$+127$	$\geqslant +1023$	$+1023$	$\geqslant +16383$
$E_{\min}$	-126	$\leqslant -1022$	-1022	$\leqslant -16382$
指数偏移量	$+127$	未指定	$+1023$	未指定
指数位数	8	$\geqslant 11$	11	$\geqslant 15$
总位数	32	$\geqslant 43$	64	$\geqslant 79$

Table 2.3: IEEE 754 描述的四种浮点数格式的要求

2. 如果 e 是 2047 且 f 为 0（形如 s 11111111111 0000$\cdots$0000），那么其表示的值为 $(-1)^s\infty$（即正负无穷大）；
3. 如果 e 在 0（不含）和 2047（不含）之间（通常情况），那么其表示的值为 $(-1)^s 2^{e-1023}(1 \cdot f)$；
4. 如果 e 为 0 但 f 不为 0（形如 s 00000000000 $bbbb\cdots bbbb$），那么其表示的值为非规格化浮点数 $(-1)^s 2^{-1022}(0 \cdot f)$；
5. 如果 e 和 f 均为 0（形如 s 00000000000 0000$\cdots$0000），那么其表示的值为 $(-1)^s 0$（即正负 0）。

扩展格式

IEEE 754 还定义了单精度扩展格式和双精度扩展格式所需要遵守的规范，如表格 2.3 所示，但是对具体实现方式不做出任何要求。

另外，IEEE 754 还定义了浮点数相关的运算等，由于本章仅关注记数基础，因此运算相关的会放到后面进行讲解。

2.7 非进位制的记数系统

我们前面讨论的都是进位制的记数系统，但除此之外还有一些非进位制的记数系统（Non-positional Number System）。

非进位制的记数系统，是数码所表示的数值与位置无关的记数系统的统称。所谓“数码所表示的数值与位置无关”的含义，可以先回想一下前面在描述进位制时的内容：

> “进位制记数法”……就是……使用若干个不同的符号记录数字，并且使用不同的位置表达同一个符号的不同“权重”的记数系

字母	I	V	X	L	C	D	M
数值	1	5	10	50	100	500	1000

Table 2.4: 罗马数字常用的拉丁字母及其值

统。

> 在进位制中，不同的位置上的数码，要乘以它所在的位的“权重”。对于十进制而言，这个“权重”就是我们通常所说的“个”、“十”、“百”、“千”、“万”等：……于是，在最右一位上的数字，代表的就是有多少个单位“1”；而它左边一位的数字，代表的是有多少个“十”，再往左一位的数字，代表有多少个“百”，或者说有多少个“十个十”。

这就是“数码所表示的数值与位置有关”的一个例子：例如对于十进制数 11000 而言，第一个 1 表示的是“一个万”，第二个 1 表示的是“一个千”，相同的数码在不同的位置上表示不同的含义，且位置信息在数值表记中作为权重体现，这就是“数码所表示的数值与位置有关”的情况。

“数码所表示的数值与位置无关”的情况则与此相反：一个数码在某个位置上，并不表示它必然具有某个指定的权重，它所代表的数值可能要根据其上下文确定。

2.7.1 罗马数字

罗马数字（Roman Numeral）是古罗马使用的记数系统，使用拉丁字母的一套结合规则表达数字，现今依然见于例如年份记载、钟表、编号等场合。

具体的值见于表 2.4。

确定的结合规则包括：

1. 连续重复几次就是几倍；
2. 数值大的字母的右边写上数值小的字母，是两个数值相加；若是左边则是相减。

另外有一些常见的规则，例如连续重复的次数不超过 3 次等，但是并不是严格的；还有一些不常用的表示其他数值的字母，此处不再赘述。

2.7.2 格雷码

格雷码（Gray Code）是 1953 年由美国工程师 Frank Gray 提出的一种令相邻的数字只相差一位的二进制记数方式，常用于数据传输等场合。

自然数转换为格雷码之后的序列可以以如下方式构造：

- 令 a_0 为 $0, 1$；
- 令 a_n 为 a_{n-1} 与逆序且每一个数的最左边补上一个 1 的 a_{n-1}；
- 则 a_n 为 0 到 2^{n-1} 之间的自然数的格雷码表示。

二进制数转换为格雷码，只需要将二进制数的相邻两位做不进位加法然后开头补上一个 1 即可；格雷码转换为二进制数则需要先写上开头的 1，然后从左起第二位格雷码开始，将每一位格雷码与上一位二进制位做不进位加法得到下一位二进制位。

2.7.3 二进制编码的十进制

二进制编码的十进制（Binary-Coded Decimal，简称 BCD）是 1928 年由 IBM 发明的一种将十进制数逐位转化为二进制来拼接的记数方式，在一些不需要运算的场合（例如屏幕数字显示等）比较常用。

最常用的 BCD 是 8421 BCD，即每一位十进制数都转化为 4 位二进制数，而这 4 位的权重从左到右分别是 8、4、2、1。此外其它场合常用的还有 5421 BCD、4221 BCD 等。

另有一种称为“余 3 码”（Excess 3 Code）的 BCD，其构造方式是在转化为 8421 BCD 的时候，在每一位都加上 3 之后再转化为二进制。这种方式的好处是在进行加减法运算时较为方便，并且每一位十进制转化后不存在 0000 和 1111，在数据传输时可以简单地排除一些错误。

除此之外还有使用格雷码来作为二进制编码方式的 BCD。

Chapter 3

彩虹与色域

Chapter 4

未分类

某一天（`git log` 说是 2022 年 9 月 2 日）Geno 突然想把这本书变为一个类似于博客之类的东西，把偶然学到的、想到的以及同学问过的一些东西整理一下放在这里，未来本书写到相关章节时可以直接移动到对应位置，但目前仍然未有相关的恰当章节，因而归入“未分类”一章。

当然，这一章的东西中，与计算机技术没有关系的那些文章中，绝大部分内容只是个人观点，有一些观点并非正确，或者可能没有对错之分。

参考文献

[1] Sparanoid. 中文文案排版指北[EB/OL]. (2014-05-17). https://github.com/sparanoid/chinese-copywriting-guidelines.

[2] RAYMOND E S. How To Ask Questions The Smart Way[EB/OL]. (2014-03-21). http://www.catb.org/~esr/faqs/smart-questions.html.

[3] ΘΔΙΟΣ . [M].

[4] 甄鸾. 五经算术[M]. 北周. eprint: https://ctext.org/wiki.pl?if=gb&res=835362&remap=gb.

[5] 允祉. 御制数理精蕴[M]. 1722. eprint: https://ctext.org/wiki.pl?if=gb&res=933279&remap=gb.

[6] 孙子. 孙子算经[M]. 四世纪. eprint: https://ctext.org/sunzi-suan-jing/zhs.

[7] 国务院. 国务院关于在我国统一实行法定计量单位的命令[M]. 1984. eprint: https://gkml.samr.gov.cn/nsjg/jls/201902/t20190225_291134.html.

[8] 佛陀跋陀罗. 大方广佛华严经[M]. 东晋. eprint: https://ctext.org/wiki.pl?if=gb&res=632172&remap=gb.

[9] LEIBNIZ G W. Explication de l'arithmétique binaire, Qui se sert des seuls caractères 0 et 1; avec des Remarques sur son utilité, et sur ce qu'elle donne le sens des anciennes figures Chinoises de Fohy[J]. Histoire de l'Academie Royale des Sciences, 1703: 85-89.

[10] IBM Operating System/360 Assembler Language[A]. 1965.

[11] Monrobot XI Program Manual[A]. 1964. eprint: https://archive.org/details/MonrobotXIProgramManual.

[12] MANN J. MONROBOT XI[EB/OL]. (2007-03-15). https://users.monash.edu.au/~johnm/weblog/2007/03/15/.

[13] DATAmatic 1000 Automatic Programming Manual Volume 1 The Assembly Program[A]. 1957.

[14] LGP-30 Programming Manual[A]. 1957. eprint: http://www.bitsavers.org/pdf/royalPrecision/LGP-30/LGP-30_Programming_Manual_Apr57.pdf.

[15] LGP-30 - The Royal Precision Electronic Computer[A]. 1956. eprint: http://s3data.computerhistory.org/brochures/rpc.lgp-30.1956.102646223.pdf.

[16] PACIFIC DATA SYSTEMS I. An Engineer's Guide To The PDS 1020 [M]. 1964. eprint: http://bitsavers.org/pdf/pacificDataSystems/EG2-3M_Engineers_Guide_To_The_PDS1020_Apr64.pdf.

[17] SAVARD J J G. Computer Arithmetic[EB/OL]. (2013-03-22). http://quadibloc.com/comp/cp02.htm.

[18] Von NEUMANN J. First Draft of a Report on the EDVAC[J]. IEEE Annals of the History of Computing, 1993, 15(4): 27-75. DOI: 10.1109/85.238389.

[19] SAVARD J J G. Floating-Point Formats[EB/OL]. (2005-12-31). http://www.quadibloc.com/comp/cp0201.htm.

[20] IEEE. IEEE Standard for Binary Floating-Point Arithmetic[EB/OL]. (1985-03-21). https://standards.ieee.org/ieee/754/993/.