二

11 二进制编码：“手持两把锟斤拷，口中疾呼烫烫烫”？

上算法和数据结构课的时候，老师们都会和你说，程序 = 算法 + 数据结构。如果对应到组成原理或者说硬件层面，算法就是我们前面讲的各种计算机指令，数据结构就对应我们接下来要讲的二进制数据。

众所周知，现代计算机都是用 0 和 1 组成的二进制，来表示所有的信息。前面几讲的程序指令用到的机器码，也是使用二进制表示的；我们存储在内存里面的字符串、整数、浮点数也都是用二进制表示的。万事万物在计算机里都是 0 和 1，所以呢，搞清楚各种数据在二进制层面是怎么表示的，是我们必备的一课。

大部分教科书都会详细地从整数的二进制表示讲起，相信你在各种地方都能看到对应的材料，所以我就不再啰啰嗦嗦地讲这个了，只会快速地浏览一遍整数的二进制表示。

然后呢，我们重点来看一看，大家在实际应用中最常遇到的问题，也就是文本字符串是怎么表示成二进制的，特别是我们会遇到的乱码究竟是怎么回事儿。我们平时在开发的时候，所说的 Unicode 和 UTF-8 之间有什么关系。理解了这些，相信以后遇到任何乱码问题，你都能手到擒来了。

理解二进制的“逢二进一”

二进制和我们平时用的十进制，其实并没有什么本质区别，只是平时我们是“逢十进一”，这里变成了“逢二进一”而已。每一位，相比于十进制下的 0～9 这十个数字，我们只能用 0 和 1 这两个数字。

任何一个十进制的整数，都能通过二进制表示出来。把一个二进制数，对应到十进制，非常简单，就是把从右到左的第 N 位，乘上一个 2 的 N 次方，然后加起来，就变成了一个十进制数。当然，既然二进制是一个面向程序员的“语言”，这个从右到左的位置，自然是从 0 开始的。

比如 0011 这个二进制数，对应的十进制表示，就是
0×23+0×22+1×21+1×200×23+0×22+1×21+1×20 =3=3，代表十进制的 3。

对应地，如果我们想要把一个十进制的数，转化成二进制，使用**短除法**就可以了。也就是，把十进制数除以 2 的余数，作为最右边的一位。然后用商继续除以 2，把对应的余数紧靠着刚才余数的右侧，这样递归迭代，直到商为 0 就可以了。

比如，我们想把 13 这个十进制数，用短除法转化成二进制，需要经历以下几个步骤：

	商	余数	二进制位
13 / 2	6	1	1
6 / 2	3	0	0
3 / 2	1	1	1
1 / 2	0	1	1

因此，对应的二进制数，就是 1101。

刚才我们举的例子都是正数，对于负数来说，情况也是一样的吗？我们可以把一个数最左侧的一位，当成是对应的正负号，比如 0 为正数，1 为负数，这样来进行标记。

这样，一个 4 位的二进制数， 0011 就表示为 +3。而 1011 最左侧的第一位是 1，所以它就表示 -3。这个其实就是整数的**原码表示法**。原码表示法有一个很直观的缺点就是，0 可以用两个不同的编码来表示，1000 代表 0， 0000 也代表 0。习惯万事一一对应的程序员看到这种情况，必然会被“逼死”。

于是，我们就有了另一种表示方法。我们仍然通过最左侧第一位的 0 和 1，来判断这个数的正负。但是，我们不再把这一位当成单独的符号位，在剩下几位计算出的十进制前加上正负号，而是在计算整个二进制值的时候，在左侧最高位前面加个负号。

比如，一个 4 位的二进制补码数值 1011，转换成十进制，就是 $-1\times23+0\times22+1\times21+1\times20-1\times23+0\times22+1\times21+1\times20 =-5=-5$。如果最高位是 1，这个数必然是负数；最高位是 0，必然是正数。并且，只有 0000 表示 0，1000 在这样的情况下表示 -8。一个 4 位的二进制数，可以表示从 -8 到 7 这 16 个整数，不会白白浪费一位。

当然更重要的一点是，用补码来表示负数，使得我们的整数相加变得很容易，不需要做任何特殊处理，只是把它当成普通的二进制相加，就能得到正确的结果。

我们简单一点，拿一个 4 位的整数来算一下，比如 -5 + 1 = -4，-5 + 6 = 1。我们各自把它们转换成二进制来看一看。如果它们和无符号的二进制整数的加法用的是同样的计算方式，这也就意味着它们是同样的电路。

	十进制数		二进制表示			
	-5		1	0	1	1
+	1		0	0	0	1
进位			-	1	1	-
加法结果	-4		1	1	1	0

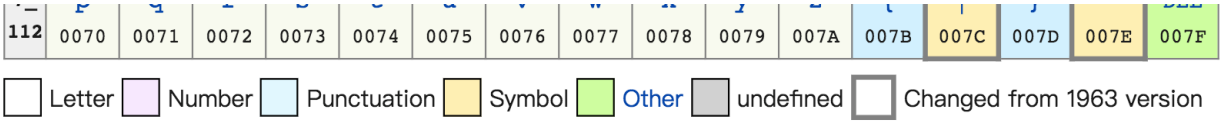
	十进制数		二进制表示			
	-5		1	0	1	1
+	6		0	1	1	0
进位		1	1	1	-	-
加法结果	1	溢出丢弃	0	0	0	1

字符串的表示，从编码到数字

不仅数值可以用二进制表示，字符乃至更多的信息都能用二进制表示。最典型的例子就是**字符串**（Character String）。最早计算机只需要使用英文字符，加上数字和一些特殊符号，然后用 8 位的二进制，就能表示我们日常需要的所有字符了，这个就是我们常常说的**ASCII 码**（American Standard Code for Information Interchange，美国信息交换标准代码）。

ASCII (1977/1986)

	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
0_ 0	NUL 0000	SOH 0001	STX 0002	ETX 0003	EOT 0004	ENQ 0005	ACK 0006	BEL 0007	BS 0008	HT 0009	LF 000A	VT 000B	FF 000C	CR 000D	SO 000E	SI 000F
1_ 16	DLE 0010	DC1 0011	DC2 0012	DC3 0013	DC4 0014	NAK 0015	SYN 0016	ETB 0017	CAN 0018	EM 0019	SUB 001A	ESC 001B	FS 001C	GS 001D	RS 001E	US 001F
2_ 32	SP 0020	! 0021	" 0022	# 0023	$ 0024	% 0025	& 0026	' 0027	(0028	) 0029	* 002A	+ 002B	, 002C	- 002D	. 002E	/ 002F
3_ 48	0 0030	1 0031	2 0032	3 0033	4 0034	5 0035	6 0036	7 0037	8 0038	9 0039	: 003A	; 003B	< 003C	= 003D	> 003E	? 003F
4_ 64	@ 0040	A 0041	B 0042	C 0043	D 0044	E 0045	F 0046	G 0047	H 0048	I 0049	J 004A	K 004B	L 004C	M 004D	N 004E	O 004F
5_ 80	P 0050	Q 0051	R 0052	S 0053	T 0054	U 0055	V 0056	W 0057	X 0058	Y 0059	Z 005A	[005B	\ 005C	] 005D	^ 005E	_ 005F
6_ 96	` 0060	a 0061	b 0062	c 0063	d 0064	e 0065	f 0066	g 0067	h 0068	i 0069	j 006A	k 006B	l 006C	m 006D	n 006E	o 006F
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL



图片来源

ASCII 码就好比一个字典，用 8 位二进制中的 128 个不同的数，映射到 128 个不同的字符里。比如，小写字母 a 在 ASCII 里面，就是第 97 个，也就是二进制的 0110 0001，对应的十六进制表示就是 61。而大写字母 A，就是第 65 个，也就是二进制的 0100 0001，对应的十六进制表示就是 41。

在 ASCII 码里面，数字 9 不再像整数表示法里一样，用 0000 1001 来表示，而是用 0011 1001 来表示。字符串 15 也不是用 0000 1111 这 8 位来表示，而是变成两个字符 1 和 5 连续放在一起，也就是 0011 0001 和 0011 0101，需要用两个 8 位来表示。

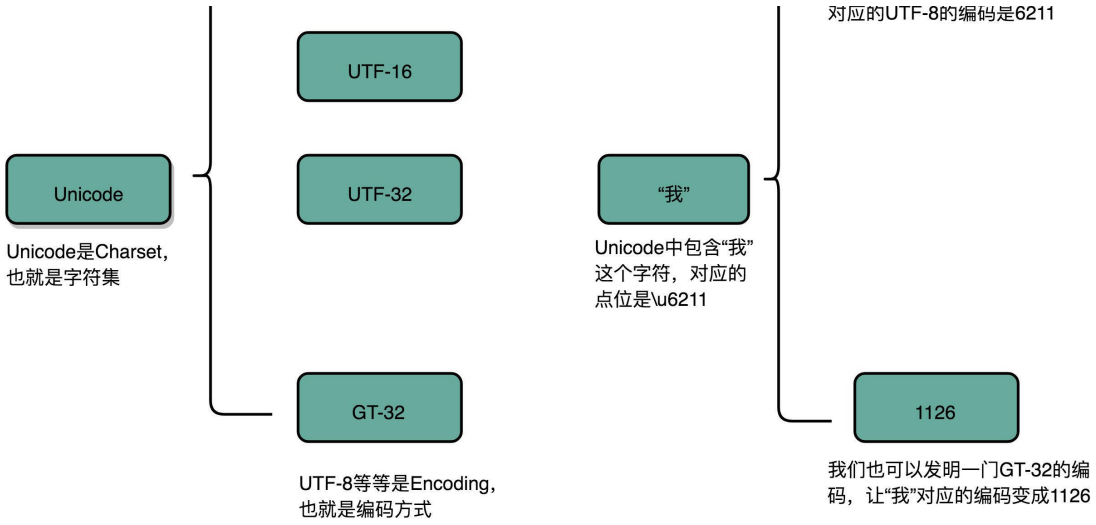
我们可以看到，最大的 32 位整数，就是 2147483647。如果用整数表示法，只需要 32 位就能表示了。但是如果用字符串来表示，一共有 10 个字符，每个字符用 8 位的话，需要整整 80 位。比起整数表示法，要多占很多空间。

这也是为什么，很多时候我们在存储数据的时候，要采用二进制序列化这样的方式，而不是简单地把数据通过 CSV 或者 JSON，这样的文本格式存储来进行序列化。**不管是整数也好，浮点数也好，采用二进制序列化会比存储文本省下不少空间。**

ASCII 码只表示了 128 个字符，一开始倒也堪用，毕竟计算机是在美国发明的。然而随着越来越多的不同国家的人都用上了计算机，想要表示譬如中文这样的文字，128 个字符显然是不太够用的。于是，计算机工程师们开始各显神通，给自己国家的语言创建了对应的**字符集**（Charset）和**字符编码**（Character Encoding）。

字符集，表示的可以是字符的一个集合。比如“中文”就是一个字符集，不过这样描述一个字符集并不准确。想要更精确一点，我们可以说，“第一版《新华字典》里面出现的所有汉字”，这是一个字符集。这样，我们才能明确知道，一个字符在不在这个集合里面。比如，我们日常说的 Unicode，其实就是一个字符集，包含了 150 种语言的 14 万个不同的字符。

而字符编码则是对于字符集里的这些字符，怎么一一用二进制表示出来的一个字典。我们上面说的 Unicode，就可以用 UTF-8、UTF-16，乃至 UTF-32 来进行编码，存储成二进制。所以，有了 Unicode，其实我们可以用不止 UTF-8 一种编码形式，我们也可以自己发明一套 GT-32 编码，比如就叫作 Geek Time 32 好了。只要别人知道这套编码规则，就可以正常传输、显示这段代码。



同样的文本，采用不同的编码存储下来。如果另外一个程序，用一种不同的编码方式来进行解码和展示，就会出现乱码。这就好像两个军队用密语通信，如果用错了密码本，那看到的消息就会不知所云。在中文世界里，最典型的就是“手持两把锟斤拷，口中疾呼烫烫烫”的典故。

我曾经听说过这么一个笑话，没有经验的同学，在看到程序输出“烫烫烫”的时候，以为是程序让 CPU 过热发出报警，于是尝试给 CPU 降频来解决问题。

既然今天要彻底搞清楚编码知识，我们就来弄清楚“锟斤拷”和“烫烫烫”的来龙去脉。

锟斤拷锟斤拷PP么
On 2013锟斤拷12锟斤拷13锟斤拷 10:28, akirya(锟斤拷[锟斤拷实锟揭诧拷锟斤拷什么锟斤拷谓锟侥伙拷锟斤拷]) wrote:

> 锟斤拷锟斤拷锟斤拷锟斤拷锟街○拷锟杰久讹拷没锟斤拷么锟斤拷锟街癸拷锟斤拷
>
> 锟斤拷 2013锟斤拷6锟斤拷2锟斤拷锟斤拷锟斤拷锟斤拷锟斤拷UTC+8锟斤拷锟斤拷10时31锟斤拷34锟诫，carrie写锟斤拷锟斤拷锟斤拷
>> 锟揭爸帮拷说锟斤拷锟斤拷要女 锟斤拷锟斤拷
>> 锟斤拷女锟斤拷说锟斤拷锟斤拷要锟斤拷锟绞憋拷锟斤拷锟斤拷烁锟斤拷锟斤拷汁
>> 平时锟斤拷识锟斤拷锟斤剿诧拷锟劫○拷锟叫碉拷也锟杰多，锟斤拷只锟斤拷锟斤拷识锟斤拷锟斤拷知锟斤拷锟斤拷么锟酵筹拷锟斤拷剩女锟剿○拷锟斤拷
>> 锟斤拷锟斤拷锟斤拷锟教筹拷蝼糠锟斤拷羌锟斤拷锟秸□拷锟斤拷锟斤拷锟斤拷锟斤拷锟皆憋拷锟斤拷潜冉峡锟斤拷椎摹锟wbr>锟斤拷锟斤拷锟斤拷锟斤拷锟坛锟斤拷锟斤拷锟斤拷锟斤拷锟斤拷锟斤拷锟揭搅拷注锟斤拷锟剿○拷锟斤拷锟斤拷锟斤拷锟还锟角比斤拷锟阶的★拷锟斤拷锟斤拷 锟斤拷锟斤拷锟斤拷锟斤拷锟揭的达拷锟斤拷
>> •••

搜索了一下我自己的个人邮件历史记录，不出意外，里面出现了各种“锟斤拷”

首先，“锟斤拷”的来源是这样的。如果我们想要用 Unicode 编码记录一些文本，特别是一些遗留的老字符集内的文本，但是这些字符在 Unicode 中可能并不存在。于是，Unicode 会统一把这些字符记录为 U+FFFD 这个编码。如果用 UTF-8 的格式存储下来，就是\xef\xbf\xbd。如果连续两个这样的字符放在一起，\xef\xbf\xbd\xef\xbf\xbd，这个时候，如果程序

把这个字符，用 GB2312 的方式进行 decode，就会变成“锟斤拷”。这就好比我们用 GB2312 这本密码本，去解密别人用 UTF-8 加密的信息，自然没办法读出有用的信息。

而“烫烫烫”，则是因为如果你用了 Visual Studio 的调试器，默认使用 MBCS 字符集。“烫”在里面是由 0xCCCC 来表示的，而 0xCC 又恰好是未初始化的内存的赋值。于是，在读到没有赋值的内存地址或者变量的时候，电脑就开始大叫“烫烫烫”了。

了解了这些原理，相信你未来在遇到中文的编码问题的时候，可以做到“手中有粮，心中不慌”了。

总结延伸

到这里，相信你发现，我们可以用二进制编码的方式，表示任意的信息。只要建立起字符集和字符编码，并且得到大家的认同，我们就可以在计算机里面表示这样的信息了。所以说，如果你有心，要发明一门自己的克林贡语并不是什么难事。

不过，光是明白怎么把数值和字符在逻辑层面用二进制表示是不够的。我们在计算机组成里面，关心的不只是数值和字符的逻辑表示，更要弄明白，在硬件层面，这些数值和我们一直提的晶体管和电路有什么关系。下一讲，我就会为你揭开神秘的面纱。我会从时钟和 D 触发器讲起，最终让你明白，计算机里的加法，是如何通过电路来实现的。

推荐阅读

关于二进制和编码，我推荐你读一读《编码：隐匿在计算机软硬件背后的语言》。从电报机到计算机，这本书讲述了很多计算设备的历史故事，当然，也包含了二进制及其背后对应的电路原理。

上一页 下一页