第 3 章 chapter 3

逻 辑 思 维

计算机科学是逻辑的继续，只是换了载体。

Computer Science is the continuation of Logic by other means.

——乔治·戈特洛布(Georg Gottlob)，2011

第 1 章讲述了计算机科学的发展概貌。我们看到，自 ENIAC 数字电子计算机发明以来，计算机的能力(计算速度)进入了随时间指数增长的轨道。历史发展还验证了巴贝扬断言：计算速度是像黄金一样的硬通货，可以兑换成其他产品和服务。计算能力的指数增长使得计算机、计算机科学、计算思维日益渗透到人类社会生产生活的各个方面，产生了许许多多的使用模式和计算机应用。它们都是通过运行在计算机上的计算过程体现的。

但是，我们尚未讨论计算机科学的一个根本问题：计算过程可以用来解决什么问题？

这个问题还可以有多个变种与细化。

(1) 计算过程可以用来解决哪些问题？这些问题称为可计算问题。

(2) 存不存在计算过程不可解的问题？

(3) 存不存在一种通用的计算机？什么是通用计算机？通用是何含义？

我们可以将计算过程理解为在计算机上通过操作数字符号变换信息的过程。操作数字符号的最基本的理论模型是布尔逻辑，需要考虑系统状态时的基本理论模型是图灵机。本章将讨论布尔逻辑和图灵机，并对上述问题给出明确的回答。

(1) 计算过程可以用来解决图灵可计算问题。

(2) 存在任何计算过程都不可解的问题。

(3) 存在通用计算机，即任何合理定义的计算机都可被该通用计算机模拟。

逻辑思维是一种普遍的能力和思维方式，在其他学科也有体现，尤其在数学中体现最为明显。计算机科学的逻辑思维有别于其他科学的逻辑思维，强调比特精准以及能够机械地自动执行的特点。

3.1 布 尔 逻 辑

3.1.1 命题逻辑

我们先用一个实例直观地了解命题逻辑，随后再讨论它的基本概念，包括命题、连接词、真值表、基本性质、范式，以及布尔函数和布尔表达式。

【实例 3.1】 探险者难题。

一位探险者在奥斯仙境旅行，他想要去翡翠城，但路上必须经过说谎国。说谎国的人永远说谎话，而诚实国的人永远讲真话。一天探险者走到了一个岔路口，两条路分别通向诚实国和说谎国，他不知道哪一条路是去往说谎国的路。正在他犹豫不决的时候，路上来了两个人，已知其中一人来自诚实国，另一人来自说谎国。探险者需要问这两个人哪一条是去往说谎国的路，请问他应该如何进行询问？

为了体现逻辑的作用，这里我们提出如下要求：探险者只能问一次问题，而且问题的答案只能是“是”或者“否”。

解答：为了叙述方便，我们将这两个人分别称为 A 和 B，两条路分别记为 s 和 t。探险者的一种提问方法如下：提问 A，“你对于‘你来自诚实国’和‘路 s 通往说谎国’这两个问题的回答是相同的吗？”若 A 回答“是”，则应该走 s 这条路；若 A 回答“否”，则应该走 t 这条路。

上面的提问方式的表述有一些复杂，下面我们给上述解答一个简单的证明，希望能从中体现出逻辑的作用。其中会用到一些逻辑的符号和推理规则，其含义在后面的章节中会做具体介绍，暂时不能完全看懂也没有关系。

我们用 $A=1$ 表示 A 来自诚实国，$A=0$ 表示 A 来自说谎国。同理，用 $B=1$ 表示 B 来自诚实国，$B=0$ 表示 B 来自说谎国。用 $s=1$ 表示路 s 通往说谎国，路 $s=0$ 通往诚实国（即另一条路 t 通往说谎国）。

注意，无论探险者问 A 或者 B：“你来自诚实国吗？”，回答总是“是”，无论他们真正来自哪个国家。但是，如果问 A：$A\oplus s=?$ 其结果是，无论 A 的值是 1 或 0，其回答总是 $\neg s$（s 的非）。这里的 $A\oplus s$ 表示 A 与 s 的异或，即 A 与 s 的值是否相等：若相等，$A\oplus s$ 的值为 0，否则值为 1。这里我们使用了如下性质：$1\oplus s=\neg s$，$0\oplus s=s$，另外要注意到在 $A=0$ 时他将说谎。

因此，如果 $A\oplus s=0$，即 A 回答“是”，或者说“你来自诚实国”和“路 s 通往说谎国”这两个问题的回答相同，那么说明 $s=1$，因此应该走 s 这条路。相反，如果 A 回答“否”，则应该走 t 这条路。

1. 命题

命题分为简单命题和复合命题。“今天下雨”“$x^2\geqslant 0$”“π 是超越数”都是简单命题。“$y<3$，并且 $y>0$”“a 是素数，或者 $a\equiv 1(\mathrm{mod}\ 4)$”都是复合命题。

计算机科学使用布尔逻辑（Boolean logic）表征命题逻辑（proposition logic），在布尔逻辑中只有两个值：“真（True，T）”和“假（False，F）”。在布尔逻辑中我们假定所有的命题，或者为真命题，或者为假命题，用 1 表示“真”，0 表示“假”。

2. 连接词

简单命题可以通过连接词组合成复合命题，常见的连接词有与、或、非、蕴含、异或等。

（1）**合取**符号 $\wedge$，即“**与**”（conjunction，and）：$x\wedge y=1$ 当且仅当 $x=y=1$，也就是说只要 x 和 y 中有一个为假，那么 $x\wedge y$ 为假。例如：$x^2<1$ 的解，$x>-1$ 并且 $x<1$（即 $-1<x<1$）可以记作 $(x>-1)\wedge(x<1)$。

（2）**析取**符号 $\vee$，即“**或**”（disjunction，or）：$x\vee y=0$ 当且仅当 $x=y=0$，也就是说只要

x 和 y 中有一个为真，那么 $x \vee y$ 为真。例如：$x^2 \geqslant 1$ 的解，$x \leqslant -1$ 或者 $x \geqslant 1$ 可以记作 $(x \leqslant -1) \vee (x \geqslant 1)$。

(3) **非**符号 ¬(negation，not)：$\neg x = 0$ 当且仅当 $x = 1$，即非 x 为假当且仅当 x 为真。通常我们也用 $\bar{x}$ 表示命题 x 的非，例如：$\overline{x \vee y} = \neg(x \vee y)$。由于布尔逻辑中只有 0 和 1 两个值，因此 $\neg(\neg x) = x$，即否定之否定即为肯定。

(4) **蕴含**符号→(implication)：$x \rightarrow y = 1$ 当且仅当 $x = 0$ 或者 $y = 1$，即前提为假或者结论为真。这里要特别注意，当前提为假的时候，不管结论是否正确，此命题都是真命题。一个例子：山无陵，江水为竭，冬雷震震，夏雨雪，天地合，乃敢与君绝。

(5) **异或**符号⊕(exclusive or)：$x \oplus y = 1$ 当且仅当 $x \neq y$，即异或能用来判断 x 与 y 是否相等。如果把 x 和 y 都当作数值来看，那么 $x \oplus y = x + y \pmod 2$。需要特别注意 $1 \oplus 1 = 0$，因此异或常被用来做取补的运算。

3. 真值表

对于每一个布尔函数，可以将所有变量的全部可能取值以及所对应的最终函数值都列在一个表里，每一行对应所有变量的一种取值，此即一个布尔函数的**真值表**。表 3.1 列出了合取、析取、蕴含以及异或操作的真值表。

表 3.1　合取、析取、蕴含、异或操作的真值表

x	y	$x \wedge y$	$x \vee y$	$x \rightarrow y$	$x \oplus y$
0	0	0	0	1	0
0	1	0	1	1	1
1	0	0	1	0	1
1	1	1	1	1	0

4. 布尔逻辑的若干基本性质

下面列出了关于合取、析取、非、异或操作的一些基本性质，我们在此略去这些性质的证明过程，请读者利用真值表自行验证。

(1) 交换律：$x \wedge y = y \wedge x$，$x \vee y = y \vee x$，$x \oplus y = y \oplus x$。

(2) 结合律：$x \wedge (y \wedge z) = (x \wedge y) \wedge z$，$x \vee (y \vee z) = (x \vee y) \vee z$，$x \oplus (y \oplus z) = (x \oplus y) \oplus z$。

(3) 分配律：$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$，$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$。

(4) 幺元律：$x \vee 0 = x$，$x \wedge 1 = x$，$x \oplus 0 = x$。0 是 $\vee$ 和 $\oplus$ 的幺元，1 是 $\wedge$ 的幺元。针对异或操作，有 $x \oplus 1 = \bar{x}$，1 不是 $\oplus$ 的幺元。

(5) 极元律：$x \wedge 0 = 0$，$x \vee 1 = 1$。0 是 $\wedge$ 的极元，1 是 $\vee$ 的极元。

(6) 幂等律：$x \vee x = x$，$x \wedge x = x$。

(7) 吸收律：$x \vee (x \wedge y) = x$，$x \wedge (x \vee y) = x$。

(8) 互补律：$x \wedge \bar{x} = 0$，$x \vee \bar{x} = 1$。

(9) 双重否定律：$\bar{\bar{x}} = x$，即 $\neg(\neg x) = x$。

(10) De Morgan 律(德摩根定律)：$\overline{x \wedge y}=\bar{x} \vee \bar{y}$，$\overline{x \vee y}=\bar{x} \wedge \bar{y}$。

借助德摩根定律，我们可以推出 $x \wedge y=\overline{\bar{x} \vee \bar{y}}$，以及 $x \vee y=\overline{\bar{x} \wedge \bar{y}}$，因此我们可以将一个命题中的所有“与”都去掉，替换成为“或”和“非”。同样，我们可以把一个命题中所有的“或”去掉，而替换成“与”和“非”。例如：$x \wedge (y \vee z)=\overline{\bar{x} \vee \overline{(y \vee z)}}$。

对于“蕴含”和“异或”，我们也可以将它们表示成关于“与”“或”“非”的命题，进而可以只用“与”和“非”(或者“或”和“非”)来表示，得到下列性质。

(11) $x \rightarrow y=\bar{x} \vee y$。

(12) $x \oplus y=(\bar{x} \wedge y) \vee (x \wedge \bar{y})$，$x \oplus y=(x \vee y) \wedge (\bar{x} \vee \bar{y})$。

5. 析取范式和合取范式

对于一些更加复杂的命题，例如$(x \oplus y) \rightarrow z$，我们是否能够也将其写成只包含“与”“或”“非”的命题呢？给定任意的复合命题，能否写成只包含“与”“或”“非”的命题呢？答案是肯定的！一种方式是反复借助上述基本性质进行展开。另外一种重要的方法是借助真值表来进行展开。让我们先用一个例子说明。表 3.2 是$(x \oplus y) \rightarrow z$ 的真值表。

表 3.2 $(x \oplus y) \rightarrow z$ 的真值表

x	y	z	$(x \oplus y) \rightarrow z$
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

从表 3.2 中可以看出，当(x,y,z)的取值分别是$(0,0,0)$，$(0,0,1)$，$(0,1,1)$，$(1,0,1)$，$(1,1,0)$，$(1,1,1)$时，命题$(x \oplus y) \rightarrow z$ 的取值为真。对于$(0,0,0)$，我们可以用一个由“与”和“非”连接的命题 $\bar{x} \wedge \bar{y} \wedge \bar{z}$ 与之对应，命题 $\bar{x} \wedge \bar{y} \wedge \bar{z}$ 为真当且仅当(x,y,z)的取值为$(0,0,0)$。对于其他 5 项，我们同样可以写出相应的由“与”和“非”连接的命题：$\bar{x} \wedge \bar{y} \wedge z$，$\bar{x} \wedge y \wedge z$，$x \wedge \bar{y} \wedge z$，$x \wedge y \wedge \bar{z}$ 和 $x \wedge y \wedge z$。最后我们将这 6 个命题用“或”连接如下：

$$(\bar{x} \wedge \bar{y} \wedge \bar{z}) \vee (\bar{x} \wedge \bar{y} \wedge z) \vee (\bar{x} \wedge y \wedge z) \vee (x \wedge \bar{y} \wedge z) \vee (x \wedge y \wedge \bar{z}) \vee (x \wedge y \wedge z)$$

这就给出了$(x \oplus y) \rightarrow z$ 的表示，或者说，上述由“或”连接起来的“与”“非”式的真值表就是表 3.2。这里的道理是因为：上述式子是若干个命题的析取连接(“或”)在一起。根据“或”的性质，上式值为“真”当且仅当其中至少某一个子命题为“真”。而每一个子命题都是若干命题变元(或者它们的“非”)的合取(“与”)，其值为真当且仅当(x,y,z)取我们要的某一个值。所以

$$(x \oplus y) \rightarrow z=(\bar{x} \wedge \bar{y} \wedge \bar{z}) \vee (\bar{x} \wedge \bar{y} \wedge z) \vee (\bar{x} \wedge y \wedge z) \vee (x \wedge \bar{y} \wedge z) \vee (x \wedge y \wedge \bar{z}) \vee (x \wedge y \wedge z)$$

上述这种通过将真值表中的每一个取值为 1 的行对应到一个“与”(合取)连接式,最后用“或”(析取)将所有各行对应的命题连接起来的表示,称为这一命题的“**析取范式**”。

【定理 3.1】 析取范式。

任何一个有 n 个变元的命题 $F(x_1,x_2,\cdots,x_n)$,一定可以表示成如下析取范式:

$$F(x_1,x_2,\cdots,x_n)=Q_1 \vee Q_2 \vee \cdots \vee Q_m$$

其中,每一个 Q_i 都是这 n 个变元或其“非”的合取(“与”)连接式,即 $Q_i=l_1 \wedge l_2 \wedge \cdots \wedge l_n$,其中 $l_j=x_j$ 或者 $\bar{x}_j$。 ■

整个析取范式的长度 m 是命题 F 的真值表中函数值为 1 的行数,每一个 Q_i 恰好对应其中一行,在 Q_i 中每一个变量都会出现,如果这一行中对应的变量 x_j 取值是 1,则 $l_j=x_j$,如果对应的 x_j 取值是 0,则 $l_j=\bar{x}_j$。

读到这里善于思考的读者自然会问一个问题:如果我们从真值表里的 0 出发,我们是否也能够写出 $(x \oplus y)\rightarrow z$ 的表示?这个问题的答案也是肯定的。我们可以为值为 0 的两行(0,1,0)和(1,0,0)写出其用“或”连接的表示:$x \vee \bar{y} \vee z$ 和 $\bar{x} \vee y \vee z$,注意这里写的方式和之前是有区别的。最后使用“与”将两项连接起来,即

$$(x \oplus y)\rightarrow z=(x \vee \bar{y} \vee z) \wedge (\bar{x} \vee y \vee z)$$

对于这样一个表示,我们称之为 $(x \oplus y)\rightarrow z$ 的**合取范式**。

【定理 3.2】 合取范式。

任何一个有 n 个变元的命题 $F(x_1,x_2,\cdots,x_n)$,一定可以表示成如下合取范式:

$$F(x_1,x_2,\cdots,x_n)=Q_1 \wedge Q_2 \wedge \cdots \wedge Q_m$$

其中,每一个 Q_i 都是这 n 个变元或其“非”的析取(“或”)连接式,即 $Q_i=l_1 \vee l_2 \vee \cdots \vee l_n$,$l_j=x_j$ 或者 $\bar{x}_j$。 ■

整个析取范式的长度 m 是 F 的真值表中值为 0 的行数,每一个 Q_i 对应其中一行,在 Q_i 中每一个变量都会出现,如果这一行中对应的变量 x_j 取值是 0,则 $l_j=x_j$,如果对应的 x_j 取值是 1,则 $l_j=\bar{x}_j$。

【实例 3.2】 试分别写出 $x\rightarrow(y \oplus z)$ 的合取范式和析取范式。

解:首先写出 $x\rightarrow(y \oplus z)$ 的真值表如表 3.3 所示。

表 3.3 $x\rightarrow(y \oplus z)$ 的真值表

x	y	z	$x\rightarrow(y \oplus z)$
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

分别根据真值表中 0 的行和 1 的行，可以写出相应的合取范式和析取范式如下：

$$x\rightarrow(y\oplus z)=(\bar{x}\vee y\vee z)\wedge(\bar{x}\vee\bar{y}\vee\bar{z})$$

$$x\rightarrow(y\oplus z)=(\bar{x}\wedge\bar{y}\wedge\bar{z})\vee(\bar{x}\wedge\bar{y}\wedge z)\vee(\bar{x}\wedge y\wedge\bar{z})\vee(\bar{x}\wedge y\wedge z)\vee(x\wedge\bar{y}\wedge z)\vee(x\wedge y\wedge\bar{z})$$

6. 布尔函数与布尔表达式

任意命题事实上定义了一个布尔函数，从给定的 n 个变量值，映射到特定函数值。

【定义 3.1】 布尔函数。

一个 n 变量的布尔函数是一个数学函数 $f:\{0,1\}^n\rightarrow\{0,1\}$，函数映射 f 由其真值表确定。 ■

命题 $x\rightarrow(y\oplus z)$ 定义了一个 3 变量布尔函数，它的函数映射 f 由上述真值表确定，或等价地表达为下述函数定义

$$f(x,y,z)=\begin{cases}0 & x=1,y=0,z=0\\0 & x=1,y=1,z=1\\1 & \text{其他组合}\end{cases}$$

【实例 3.3】 n 个变量的布尔函数一共有多少个？

让我们从简单的情形开始。

零个变元的布尔函数(常函数)有 1 和 0，一共 2 个，即函数值恒等于 1 或恒等于 0 两个。

一个变元的布尔函数有：1、0、x 和 $\bar{x}$，一共 4 个。

二个变元的布尔函数有：1、0、x、$\bar{x}$、y、$\bar{y}$、$x\wedge y$、$x\vee y$、$x\oplus y$，…

这样一个一个地去数很容易遗漏，下面从另一个角度来看这个问题。考虑函数的真值表，二元函数真值表一共有 $2^2=4$ 行，每一行的值可以是 0，也可以是 1，一共两种选择。因此根据乘法原理，所有不同的可能性有 $2^4=16$ 种，即不同的布尔函数有 16 个。如果我们要一一列出所有的这些函数，可以通过真值表的方式，也可以通过写出它们的合取(析取)范式的形式。

仿照上面的推理，我们可以从 2 元布尔函数推广到一般的 n 元布尔函数：n 元布尔函数的真值表一共有 2^n 行，每一行对应的函数取值可以有 0 和 1 两种，因此不同的布尔函数一共有 2^{2^n} 个。

上述刻画布尔函数的方式是一种枚举方式，将每一种输入组合对应的输出组合一一枚举列出。这种方式的优点是完备，不会漏掉任一个映射。它的明显缺点是烦琐，当输入变量 n 较大时很花时间。另一个不太明显的缺点是，枚举定义没有揭示函数的特点。采用布尔表达式刻画布尔函数，常常可以避免这两个缺点。事实上我们在描述命题时，已经用了这种方式。例如，奇偶函数 $y=x_1\oplus x_2\oplus\cdots\oplus x_n$ 很容易用布尔表达式表示。

【定义 3.2】 布尔表达式。

0、1 称为布尔常量。只能取值 0 或 1 的变量 $x_1,x_2,\cdots,x_n$ 称为布尔变量。用逻辑连接词连接布尔常量或逻辑变量形成的表达式称为 n 变量布尔表达式，简称布尔表达式。 ■

逻辑连接词也称为逻辑运算或逻辑算子。常见的逻辑连接词是逻辑**与**(即合取 $\wedge$，也可写作逻辑乘 $\cdot$)、逻辑**或**(即析取 $\vee$，也可写作逻辑加 $+$)、逻辑**非**($\neg$，也可将 $\neg x$ 记为 $\bar{x}$)、**异或**($\oplus$)、**蕴含**($\rightarrow$)。

例如，$(\bar{x} \vee y \vee z) \wedge (\bar{x} \vee \bar{y} \vee \bar{z})$就是一个 3 变量布尔表达式，采用了与、或、非 3 种运算。它也可被记为$(\bar{x}+y+z) \cdot (\bar{x}+\bar{y}+\bar{z})$。

要注意的是，布尔函数有唯一性，即每一个布尔函数都有唯一的真值表；但是，同一个布尔函数可用多个不同的布尔表达式表示。例如，$x \rightarrow (y \oplus z)$、$(\bar{x} \vee y \vee z) \wedge (\bar{x} \vee \bar{y} \vee \bar{z})$是两个不同的布尔表达式，但它们都表示同样的布尔函数，对应同样的真值表。我们说两个布尔表达式是**等价的**，如果它们对应同样真值表。反之，$x \rightarrow (y \oplus z)$、$(y \oplus z) \rightarrow x$ 这两个不同的布尔表达式则不是等价的，因为它们不对应于相同的真值表。

【实例 3.4】 n 个变量的互不等价的布尔表达式一共有多少个？

有 2^{2^n} 个。

【实例 3.5】 假如不考虑是否等价，n 个变量的布尔表达式一共有多少个？

有无穷多个。

【实例 3.6】 列出 2 个变量的互不等价的布尔表达式。

我们提出一条思路，完整答案留作练习。

2 个变量的互不等价的布尔表达式一共有 $2^{2^2}=16$ 个。写出每一个的真值表，再根据真值表写出析取范式。

通过 3.1.1 节的布尔逻辑的 12 条基本性质，我们可以从一个布尔表达式得到另一个等价的布尔表达式。这往往相当于中小学数学中的化简操作。

【实例 3.7】 列出 2 个变量的互不等价的布尔表达式，采用最简表达式形式。

2 个变量的互不等价的布尔表达式一共有 $2^{2^2}=16$ 个。按照逻辑连接词的个数从小到大逐级覆盖等价布尔表达式。每一步可先写出一个布尔表达式，再根据布尔逻辑的 10 条基本性质试图化简，从而写出最简表达式。记 2 变量布尔函数为 $y=f(x_1,x_2)$。

第一步：逻辑连接词的个数为 0 的表达式。列出所有布尔常量和布尔变量。得到 4 个布尔表达式：0、1、x_1、x_2。它们对应的真值表列举如图 3.1 所示。

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	0

(a) y=0

x_1	x_2	y
0	0	1
0	1	1
1	0	1
1	1	1

(b) y=1

x_1	x_2	y
0	0	0
0	1	0
1	0	1
1	1	1

(c) $y=x_1$

x_1	x_2	y
0	0	0
0	1	1
1	0	0
1	1	1

(d) $y=x_2$

图 3.1　逻辑连接词个数为 0 表达式的真值表

第二步：逻辑连接词的个数为 1 的表达式。将一个连接词作用于一个或多个布尔常量和布尔变量，得到下述新的 7 个布尔表达式，即 y 等于 $\bar{x}_1$，$\bar{x}_2$，x_1+x_2，$x_1 \cdot x_2$，$x_1 \oplus x_2$，$x_1 \rightarrow x_2$，$x_2 \rightarrow x_1$。它们对应的真值表列举如图 3.2 所示。

第三步：逻辑连接词的个数为 2 的表达式。我们已经得到 4+7=11 个最简表达式了。还剩下 5 个最简表达式，每个包含 2 个连接词。作为练习，请同学们写出剩余的 5 个最简布尔表达式。可能在第三步就完成了，也可能还需要更多步。

命题往往对应单输出布尔函数，简称布尔函数。简单的扩展可得到多输出布尔函数。

x_1	x_2	y
0	0	1
0	1	1
1	0	0
1	1	0

(a) $y=\bar{x}_1$

x_1	x_2	y
0	0	1
0	1	0
1	0	1
1	1	0

(b) $y=\bar{x}_2$

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

(c) $y=x_1+x_2$

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

(d) $y=x_1 \cdot x_2$

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

(e) $y=x_1 \oplus x_2$

x_1	x_2	y
0	0	1
0	1	1
1	0	0
1	1	1

(f) $y=x_1 \rightarrow x_2$

x_1	x_2	y
0	0	1
0	1	0
1	0	1
1	1	1

(g) $y=x_2 \rightarrow x_1$

图 3.2 逻辑连接词个数为 1 表达式的真值表

【定义 3.3】 n 个输入 m 个输出的布尔函数。

一个 n 输入 m 输出的布尔函数是一个数学函数 $f:\{0,1\}^n \rightarrow \{0,1\}^m$，函数映射 f 由其真值表确定。 ■

【实例 3.8】 全加器的真值表与布尔函数。

全加器是二进制加法的基本单元，其真值表如表 3.4 所示。给定 3 个输入变量的值，全加器产生对应的 2 个输出值。3 个输入变量 X、Y、C_{in} 分别代表 2 个本位输入和进位输入，2 个输出变量 Z、C_{out} 分别代表本位输出和进位输出。

表 3.4 全加器的真值表

X	Y	C_{in}	Z	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

全加器是一个 3 输入 2 输出的布尔函数 $f:\{0,1\}^3 \rightarrow \{0,1\}^2$，函数映射 f 是

$$f(X,Y,C_{in})=\begin{cases}(0,0) & X=0,Y=0,C_{in}=0\\(0,1) & X=0,Y=1,C_{in}=1\\(0,1) & X=1,Y=0,C_{in}=1\\(0,1) & X=1,Y=1,C_{in}=0\\(1,1) & X=1,Y=1,C_{in}=1\\(1,0) & \text{其他组合}\end{cases}$$

【实例 3.9】 n 个输入 m 个输出的布尔函数一共有多少个？留作练习。

【实例 3.10】 全加器的布尔表达式。

全加器有 2 个输出，可用 2 个布尔表达式刻画。假设 3 个输入变量 X、Y、C_{in} 分别代表 2 个本位输入和进位输入，2 个输出变量 Z、C_{out} 分别代表 2 个本位输出和进位输出。那么，全加器的 2 个布尔表达式是 $Z=X\oplus Y\oplus C_{in}$ 以及 $C_{out}=(X\cdot Y)+(X\oplus Y)\cdot C_{in}$。

可以看出，这种布尔表达式刻画比真值表枚举更加简洁，且体现了全加器的特点。

7. 布尔逻辑的应用

布尔逻辑主要关注针对布尔常数和布尔变量的逻辑运算。在应用中，往往体现为针对单个比特的运算。我们做几个练习，通过动手动脑进一步理解布尔逻辑。

【实例 3.11】 比特、字节与整数的异同。

理解比特、字节类型与整数类型的最简途径是编写运行一个短小程序。我们特别注意 Go 语言的按位与运算 &、按位或 |、按位异或运算 ^。

给定下列 Numbers.go 程序，先心算或手算一下输出结果是什么，然后再运行程序验证。

```
package main                                    //Numbers.go
import "fmt"
func main() {
    X := byte(63)                               //可将此行换成 X := 63,理解整数类型
    fmt.Printf("2+X=%d; 2-X=%d; X/2=%d; X%%2=%d\n",2+X,2-X,X/2,X%2)
    fmt.Printf("X>>1=%d; X<<1=%d; ",X>>1,X<<1)
    fmt.Printf("X&1=%d; X|1=%d; X^1=%d\n",X&1,X|1,X^1)
    fmt.Printf("X>>1=%08b; X<<1=%08b; ",X>>1,X<<1)
    fmt.Printf("X&1=%08b; X|1=%08b; X^1=%08b\n",X&1,X|1,X^1)
    X++
    fmt.Printf("X++ =%d\n",X)
    X--
    fmt.Printf("X-- =%d\n",X)
}
```

上述程序呈现了字节变量的算术逻辑运算。有两处需要注意的细节。第一，求余运算的字符'%'已经被用于标记%d 等占位符了，如何打印出表示求余运算的'%'字符？答案是使用%%。这是第一个打印语句出现了 X%%2 的原因。第二，如果一个字节类型的表达式值超过了[0，255]，即产生了溢出，系统会报错吗？不会报错！避免错误是程序员的责任。

程序 Numbers.go 的输出结果如下：

```
> go run Numbers.go
2+X=65; 2-X=195; X/2=31; X%2=1
X>>1=31; X<<1=126; X&1=1; X|1=63; X^1=62
X>>1=00011111; X<<1=01111110; X&1=00000001; X|1=00111111; X^1=00111110
X++ =64
X-- =63
>
```

注意几个结果：①错误结果 2－X＝195，产生溢出但没有报错；②正确结果 X/2＝31，而不是 31.5；③X 右移 1 位相当于 X 除以 2(X＞＞1＝31)；④X 左移 1 位相当于 X 乘以 2(X＜＜1＝126)；⑤按位与 X & 1 = 00111111 & 00000001 = 00000001 = 1；⑥按位或 X | 1 = 00111111 | 00000001 = 00111111；⑦按位异或 X ^ 1 = 00111111 ^ 00000001 = 00111110。

【实例 3.12】 奇偶值。

一个数的奇偶值是看它有奇数个还是偶数个 1 比特。如有偶数个，奇偶值为 0；如有奇数个，则奇偶值为 1。下列 Go 程序 parity.go 计算出 parityValue＝$X_0 \oplus X_1 \oplus X_2 \oplus \cdots \oplus X_{63}$，给定 64 比特整数 $X=(X_{63}X_{62}\cdots X_0)_2$。注意，按位异或运算在 Go 语言中的符号是^。

```
package main                          //parity.go
import "fmt"
func parity(X int) int {
   parityValue := 0
   for i := 0; i<64; i++ {            //X is a 64-bit integer
      parityValue ^= X & 1            //parityValue = parityValue ^ (X & 1)
      X = X >> 1                      //shift X right one bit
   }
   return parityValue
}
func main() {
   a := 63                            //63 = 00111111,有 6 个 1,即偶数个 1,奇偶值为 0
   fmt.Println(parity(a))
   a = 127                            //127 = 01111111,有 7 个 1,即奇数个 1,奇偶值为 1
   fmt.Println(parity(a))
}
```

程序的执行结果如下。

```
> go run parity.go
0
1
>
```

上述程序中，parityValue ^= X & 1 语句是 parityValue = parityValue ^ (X & 1)语句

的简写。其中，& 是按位与运算，^是按位异或运算。如 $X=(X_{63}X_{62}\cdots X_0)_2$ 且 $Y=(Y_{63}X_{62}\cdots Y_0)_2$，有 $X\&Y=(X_{63}\wedge Y_{63},X_{62}\wedge Y_{62},\cdots,X_0\wedge Y_0)_2$ 且 $X\hat{}Y=(X_{63}\oplus Y_{63},X_{62}\oplus Y_{62},\cdots,X_0\oplus Y_0)_2$。表达式 X & 1 保持最右 1 比特不变，将其他比特清零，即 $X\&1=(00\cdots00X_0)_2$。表达式 $X>>1$ 将 X 右移 1 比特，即 $X>>1=(0X_{63}X_{62}\cdots X_2X_1)$。

【实例 3.13】 奇偶校验。

奇偶值可用作奇偶校验，即根据奇偶值报错。例如，ASCII 码只需要 7 比特 $D_6D_5D_4D_3D_2D_1D_0$，多出来的最高比特 D_7 可用作奇偶比特(parity bit)。大写字母 A 的 7 比特码是 $D_6D_5D_4D_3D_2D_1D_0=1000001$。假设规定奇偶值为 0，我们有

$$D_7\oplus D_6\oplus D_5\oplus D_4\oplus D_3\oplus D_2\oplus D_1\oplus D_0=0$$

$$D_7\oplus 1\oplus 0\oplus 0\oplus 0\oplus 0\oplus 0\oplus 1=0$$

得到 $D_7=0$。A 加了奇偶比特之后的 8 比特码是 $D_7D_6D_5D_4D_3D_2D_1D_0=01000001$。假设硬件出错，使得某个比特翻转了，即 0 变成 1 或 1 变成 0，奇偶校验可报错。例如，假如 D_6 变成了 0，则有 $0\oplus0\oplus0\oplus0\oplus0\oplus0\oplus0\oplus1=1$，奇偶值是 1，奇偶校验报错。

【实例 3.14】 三模块冗余容错。

上述奇偶校验只能针对单比特故障报错。它不能指出是哪个比特错了，更不能纠错。

能够自动纠错的系统称为容错系统(fault tolerant system)。一种常用的容错技术是三模块冗余(triple modular redundancy，TMR)技术，如图 3.3 所示。

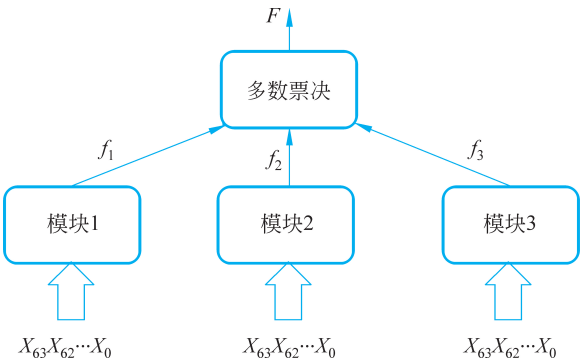

图 3.3 三模块冗余容错技术示意

假设我们已经设计了一个系统 $Y=f(X_{63},X_{62},\cdots,X_0)$，想得到一个容错系统 F。一个简单的方法是将 f 复制成 3 份(3 个模块，其输出记为 f_1、f_2、f_3)，并将 f_1、f_2、f_3 连到一个多数票决模块(majority voting module)，即可得到容错系统 F。多数票决模块的功能是：输出 $F=f_1$、f_2、f_3 的多数，即如果 f_1、f_2、f_3 的 2 个或 3 个为 0，则 $F=0$；反之 $F=1$。这个系统可以容忍模块 1、模块 2、模块 3 任一出错。

【实例 3.15】 汉明码。

三模块冗余技术需要将一个系统视为模块完整复制成 3 份，大大增加了成本。有些技术不需要完整复制整个系统。汉明码(Hamming code)就是比三模块冗余成本低得多的容错技术，由图灵奖得主理查德·汉明(Richard Hamming)教授发明。

考虑单比特纠错的场景，汉明码的主要思想如下。假设一个系统产生 m 个数据比特(data bits)，其中某个比特可能出错。我们添加 k 个奇偶校验比特(parity bits)，形成总共

$m+k$ 个比特的码字(codeword)。校验比特的巧妙设计,使得码字的任何比特出错,总会让 k 个校验比特形成的值指向码字中出错比特的位置。

由于码字一共有 $m+k$ 个比特,加上未出错的 1 种情况,总共需要区分 $m+k+1$ 种情况。因此,校验比特的个数 k 必须满足 $2^k \geqslant m+k+1$。当 $m=4$ 时,$k=3$,冗余度$(m+k)/m=7/4=1.75$,即需要 75%的额外比特用作校验。当 $m=64$ 时,$k=7$,冗余度降低到$(m+k)/m=71/64=1.1094$,即需要 10.94%的额外比特。当 $m=512$ 时,$k=10$,冗余度仅为 71/64=1.0195,即仅需要 1.95%的额外比特。

让我们仔细看看当 $m=4$ 与 $k=3$ 的情况,如表 3.5 所示。4 个数据比特记为 $D_3D_2D_1D_0$,3 个奇偶校验比特记为 $P_2P_1P_0$。注意码字中 7 个比特 $D_3D_2D_1\ P_2D_0P_1P_0$ 的特定位置。

表 3.5 汉明码校验比特的编码原理示意

码字比特	D_3	D_2	D_1	P_2	D_0	P_1	P_0
位置	7	6	5	4	3	2	1
二进制	111	110	101	100	011	010	001
P_0	√		√		√		√
P_1	√	√			√	√	
P_2	√	√	√	√			

表 3.5 中,3 个奇偶校验比特 $P_2P_1P_0$ 按照下列**汉明码规则**求出。

(1) 校验比特 P_0 对应其位置的二进制表示最低位(最右位)为 1 的比特。有 4 个位置的二进制表示末位为 1,即 001、011、101、111,对应的比特是 P_0、D_0、D_1、D_3。这 4 个比特的奇偶校验值应为 0,即 $P_0 \oplus D_0 \oplus D_1 \oplus D_3$。换言之,$P_0=D_0 \oplus D_1 \oplus D_3$。

(2) 校验比特 P_1 对应其位置的二进制表示中间一位为 1 的比特。这 4 个位置是 010、011、110、111,对应的比特是 P_1、D_0、D_2、D_3。因此,$P_1=D_0 \oplus D_2 \oplus D_3$。

(3) 校验比特 P_2 对应其位置的二进制表示最高位为 1 的比特。这 4 个位置是 100、101、110、111,对应的比特是 P_2、D_1、D_2、D_3。因此,$P_2=D_1 \oplus D_2 \oplus D_3$。

可以验证,当码字 $D_3D_2D_1\ P_2D_0P_1P_0$ 的任何单个比特出错时,3 个校验比特的值将指向出错比特的位置。

例如,给定数据比特 $D_3D_2D_1D_0=1101$,我们有

$$P_0=D_0 \oplus D_1 \oplus D_3=1 \oplus 0 \oplus 1=0$$
$$P_1=D_0 \oplus D_2 \oplus D_3=1 \oplus 1 \oplus 1=1$$
$$P_2=D_1 \oplus D_2 \oplus D_3=0 \oplus 1 \oplus 1=0$$

即,正确的原始码字是 $D_3D_2D_1\ P_2D_0P_1P_0=1100110$。

当 D_1(码字位置 5)出错,有错的数据比特 $D'_3D'_2D'_1D'_0=1111$,我们从 $D'_3D'_2D'_1D'_0$ 计算新的 3 个校验比特:

$$P''_0=D'_0 \oplus D'_1 \oplus D'_3=1 \oplus 1 \oplus 1=1$$
$$P''_1=D'_0 \oplus D'_2 \oplus D'_3=1 \oplus 1 \oplus 1=1$$
$$P''_2=D'_1 \oplus D'_2 \oplus D'_3=1 \oplus 1 \oplus 1=1$$

由于只考虑单比特纠错情况，当 D_1 出错时，码字的其他比特是正确的。因此，可能有错的码字 $D'_3D'_2\ D'_1P'_2D'_0P'_1P'_0$ 中的校验比特 $P'_2P'_1P'_0$ 应与原来的校验比特一样，即 $P'_2P'_1P'_0 = P_2P_1P_0 = 010$。对新校验比特 $P''_2P''_1P''_0$ 与可能出错的校验比特 $P'_2P'_1P'_0$ 做按位异或操作，得出 3 个症状比特(syndrome bits)$S_2S_1S_0$：

$$S_0 = P''_0 \oplus P'_0 = 1 \oplus 0 = 1$$
$$S_1 = P''_1 \oplus P'_1 = 1 \oplus 1 = 0$$
$$S_2 = P''_2 \oplus P'_2 = 1 \oplus 0 = 1$$

因此，$S_2S_1S_0 = 101 = 5$，即第 5 号位置出错。系统只需翻转第 5 号比特即可纠错。

小结一下，汉明码的单比特纠错过程有下述主要事件。

(1) 给定数据比特 $D_3D_2D_1D_0$，按照汉明码规则计算出 3 个校验比特 $P_2P_1P_0$，形成**原始码字** $D_3D_2D_1\ P_2D_0P_1P_0$。

(2) 原始码字的任何单个比特可能出错，即任何数据比特或任何校验比特可能出错，形成**可能有错码字** $D'_3D'_2D'_1P'_2D'_0P'_1P'_0$，其中包括**可能有错数据比特** $D'_3D'_2D'_1D'_0$，以及**可能有错校验比特** $P'_2P'_1P'_0$。

(3) 按照汉明码规则从可能有错数据比特 $D'_3D'_2D'_1D'_0$ 计算出 3 个**新校验比特** $P''_2P''_1P''_0$。

(4) 从新校验比特 $P''_2P''_1P''_0$ 与可能有错的校验比特 $P'_2P'_1P'_0$ 得出 3 个**症状比特** $S_2S_1S_0$。当 $S_2S_1S_0 = 000$ 时，系统没有出错；当 $S_2S_1S_0 \neq 000$ 时，$S_2S_1S_0$ 指向出错比特的位置。例如，当 $S_2S_1S_0 = 101 = 5$ 时，第 5 号位置(即数据比特 D_1)出错；当 $S_2S_1S_0 = 100 = 4$ 时，第 4 号位置(即校验比特 P_2)出错。

上述(7,4)汉明码在计算机内存中广泛使用，也称为 ECC 校验码，其中 ECC 是 Error Correction Code 的缩写。

添加了 ECC 模块的内存纠错原理如图 3.4 所示，可分 3 种情况。

情况 1：内存没有错误，则图 3.4 5 个步骤行为如下。

① 将 4 个数据比特 $D_3D_2D_1D_0$ 写入内存，同时算出校验比特 $P_2P_1P_0$ 并写入内存。例如，假设数据比特 $D_3D_2D_1D_0 = 1101$，则校验比特 $P_2P_1P_0 = 010$。因此，原始码字为 $D_3D_2D_1\ P_2D_0P_1P_0 = 1100110$。

② 若干时间后，从内存读出可能有错码字 $D'_3D'_2D'_1P'_2D'_0P'_1P'_0$。内存没有错误，因此可能有错码字等于原始码字，即 $D'_3D'_2D'_1P'_2D'_0P'_1P'_0 = D_3D_2D_1\ P_2D_0P_1P_0 = 1100110$。

③ 算出新校验比特 $P''_2P''_1P''_0 = 010$。

④ 算出症状比特 $S_2S_1S_0 = 000$。

⑤ 因为 $S_2S_1S_0 = 000$，无须翻转任何码字比特。内存读出数据比特 1101。

情况 2：内存的某个数据比特出错，则图 3.4 5 个步骤行为如下。

① 同情况 1。

② 从内存读出可能有错码字 $D'_3D'_2D'_1P'_2D'_0P'_1P'_0$。假设第 5 号位置($D_1$)出错，则有错码字 $D'_3D'_2D'_1P'_2D'_0P'_1P'_0 = D_3D_2D'_1\ P_2D_0P_1P_0 = 1110110$。

③ 算出新校验比特 $P''_2P''_1P''_0 = 111$。

④ 算出症状比特 $S_2S_1S_0 = 101$。

⑤ 由于 $S_2S_1S_0 \neq 000$，翻转 $S_2S_1S_0 = 101$ 指向位置(5)的码字比特(D'_1)。翻转 $D'_1 = 1$ 后得到 $D_1 = 0$。因此，内存读出正确的数据比特 1101。

情况 3：内存的某个校验比特出错，则图 3.4 5 个步骤行为如下。

① 同情况 1。

② 从内存读出可能有错码字 $D'_3D'_2D'_1P'_2D'_0P'_1P'_0$。假设第 4 号位置($P_2$)出错，则有错码字 $D'_3D'_2D'_1P'_2D'_0P'_1P'_0 = D_3D_2D_1P'_2D_0P_1P_0 = 1101110$。

③ 算出新校验比特 $P''_2P''_1P''_0 = P_2P_1P_0 = 010$。

④ 算出症状比特 $S_2S_1S_0 = 100$。

⑤ 由于 $S_2S_1S_0 \neq 000$，翻转 $S_2S_1S_0 = 100$ 指向位置(4)的码字比特(P'_2)。因此，我们依然看到内存读出正确的数据比特 1101。

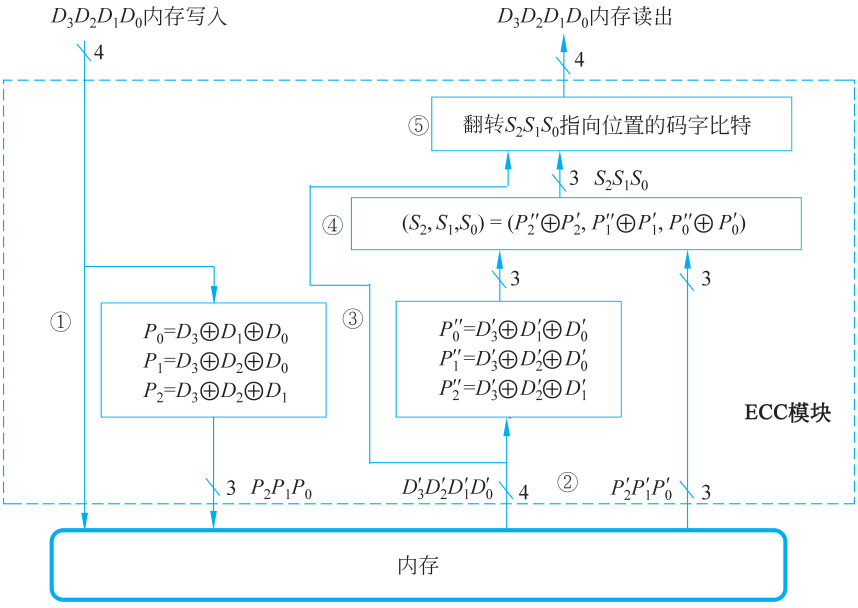

图 3.4 (7,4)汉明码例子：ECC 内存示意

【实例 3.16】 在程序中使用掩码操作一字节的特定比特。

如何操作一字节中的特定比特？使用掩码(mask)是一种常见方式。

让我们看一个具体例子：用字符'K'的每两比特替换掉数组 A 的相应比特。

假设我们想在字节数组 A = [0xD1,0xC9,0xDA,0xDA] 中隐藏 ASCII 字符'K'=75=01001011。一个做法是用'K'的每两比特替换掉目标字节的最低两比特。Go 代码如下。

```
package main                //replace.go
import "fmt"
func main() {
   A := [4]byte{0xD1,0xC9,0xDA,0xDA}
   fmt.Printf("Before: \tA = [%b %b %b %b]\n",A[0],A[1],A[2],A[3])
   data := byte('K')
   for i := 0; i < len(A); i++ {
```

```
        v := data & 0x3          //retain last 2 bits of 'K'   使用掩码 0x3
        A[i] = A[i] & 0xFC       //clear last 2 bits of A[i]   使用掩码 0xFC
        A[i] = A[i] | v          //set last 2 bits of A[i] with those of 'K'
        data = data >> 2         //repeat with the next 2 bits of 'K'
    }
    fmt.Printf("After: \t\tA = [%b %b %b %b]\n",A[0],A[1],A[2],A[3])
}
```

程序执行结果如下。注意，字符'K' = 75 = 01001011。其中，'K'的 11 两比特藏在 A[0]，10 两比特藏在 A[1]，00 两比特藏在 A[2]，01 两比特藏在 A[3]。

```
> go run replace.go
Before:      A = [11010001 11001001 11011010 11011010]
After:       A = [11010011 11001010 11011000 11011001]
>
```

该程序采用了制表符\t，使得两行输出对齐，便于比较替换前后数组 A 的二进制值。可以看出，每个数组元素只改变了最右 2 比特。我们聚焦 for 循环，只考虑 i=0，其他 i 值是类似的。当 i=0 时，data='K'=01001011 且 A[i]=A[0]= 11010001。循环体的执行细节如下。

循环体的第一条语句采用了掩码 3，即 0x3= 00000011。按位与的结果是，变量 data 的最右 2 比特保留下来，其他比特清零。

```
v := data & 0x3              //v= 01001011 & 00000011 = 00000011
                             //保留'K'的最右 2 比特
```

循环体的第二条语句采用了掩码 0xFC，即 0xFC = 11111100。按位与的结果是，变量 A[0] = 11010001 的左边 6 个比特保留下来，最右 2 比特清零，A[0]= 11010000。

```
A[i] = A[i] & 0xFC           //A[i]=11010001 & 11111100 = 11010000
                             //将 A[i] 的最右 2 比特清零，左边 6 比特不变
```

第三条语句采用按位或，将两个掩码操作结果拼起来，即将 A[0] 保留下来的左边 6 比特，与'K'保留下来的最右 2 比特，拼在一起。这就实现了“将'K'最右 2 比特替换 A[0]的相应比特”的操作。

```
A[i] = A[i] | v              //A[i]=11010000  |  00000011 = 11010011
                             //用'K'最右 2 比特替换 A[0]的相应比特
```

3.1.2 谓词逻辑

谓词逻辑(predicative logic)也称为**一阶逻辑**(first-order logic)。它拓展了命题逻辑。与简单命题逻辑相比，谓词逻辑还额外包含了断言和量化。

所谓断言是一个会传回“真”或“假”的函数。考虑下列句子：“苏格拉底是哲学家”“柏拉图是哲学家”。在命题逻辑里，上述两句被视为两个不相关的命题，分别记为 p 及 q。在一阶逻辑里，上述两句可以使用断言以更加相似的方法来表示：如果用 Phil(x)表示 x 是哲学家，那么，若 a 代表苏格拉底，则 Phil(a)为第一个命题 p；若 b 代表柏拉图，则 Phil(b)为第二个命题 q。

所谓量化通过量词来体现。在自然语言中我们常常会使用“所有”“某些”等量词，在谓词逻辑中有两个量词，分别是**全称量词** $\forall$ 和**存在量词** $\exists$。前者等同于“每一个”“所有”“一切”等，后者等同于“存在着”“至少有一个”。

【实例 3.17】 任何一个自然数，要么它本身为偶数，要么加 1 后为偶数。

$\forall n[\mathrm{Even}(n) \vee \mathrm{Even}(n+1)]$，其中断言 Even($n$)表示 n 是偶数。

【实例 3.18】 存在末 4 位是 9999 的素数。

$\exists n[\mathrm{Prime}(n) \wedge (n \equiv 9999 (\mathrm{mod} 10^4))]$，其中断言 Prime($n$)表示 n 是素数。

一个命题中可以存在多个量词。有多个量词的命题中，量词的顺序将可能表达完全不同的含义，因此不能随意修改量词的顺序。请看下面的例子，其中第一个是真命题，第二个是假命题。

【实例 3.19】 对于任意 x，都存在 y，使得 $y=x+1$。

$\forall x, \exists y(y=x+1)$。

【实例 3.20】 存在一个 y，对于任意的 x，都有 $y=x+1$。

$\exists y, \forall x(y=x+1)$。

量词的范围。我们可以为每个量词后面的变量指定一个特定的取值范围，这个范围被称为这个变量的论域或量化范围。

【实例 3.21】 任意有理数都可以写成两个整数的商。

$\forall x \in \mathbb{Q}, \exists p, q \in \mathbb{Z}\left[x=\frac{p}{q}\right]$（$\mathbb{Q}$、$\mathbb{Z}$分别代表有理数集合和整数集合，下同）。

其中，变量 x 的论域为有理数，p 和 q 的论域为整数。

【实例 3.22】 存在无穷多个素数。

$\forall n \in \mathbb{N}, \exists m \in \mathbb{N}, \forall p, q \in \mathbb{N}, p, q>1[(m>n) \wedge (m \neq pq)]$（$\mathbb{N}$代表自然数集合，下同）。

解释：这里用 m 表示所论述的素数。一个自然数是素数意味它不能写成两个大于 1 的自然数的乘积。因此，在表达式中，将其写为任意两个大于 1 的自然数 p 和 q 的乘积都不等于 m。为了表达出这样的 m 有无穷多个，我们引入一个中间变量 n，不论 n 有多大，总是有比它更大的素数 m 存在。

【实例 3.23】 对于 $n>2$，丢潘图方程 $x^n+y^n=z^n$ 不存在非平凡整数解。

$\forall a, b, c, n \in \mathbb{Z}[(abc \neq 0) \wedge (n>2) \rightarrow (a^n+b^n \neq c^n)]$。

解释：丢潘图方程的平凡解是指 x、y、z 中有一个变量为 0 的整数解，对于这里讨论的费马方程 $x^n+y^n=z^n$ 这类解总是存在的，例如 $x=0, y=z$，因此我们更关心一个丢潘图方程是否存在非平凡的整数解。“不存在”等价于“任意……都不”，上述式子用“任意 a、b、c 都不”来替代“不存在 a、b、c”，上式中我们用 $abc \neq 0$ 来表示 a、b、c 都不等于 0。

上述例子中我们使用了如下性质。

性质：$\neg(\exists xF(x))=\forall x\neg F(x)$，$\neg(\forall xF(x))=\exists x\neg F(x)$。

【实例 3.24】 存在无穷多对孪生素数。

$\forall n\in\mathbb{N},\exists m\in\mathbb{N},\forall p,q\in\mathbb{N}[(m>n)\wedge((p,q>1)\rightarrow(pq\neq m)\wedge(pq\neq m+2))]$。

解释：孪生素数是指差为 2 的两个素数。在上式中，对于任意两个大于 1 的正整数 p 和 q，m 和 $m+2$ 都不能表示成它们的乘积，也就是说 m 和 $m+2$ 是一对孪生素数。而对于任意大的 n，总是存在比它大的 m，保证 m 和 $m+2$ 是一对孪生素数，这样就刻画了孪生素数有无穷多对。注：此猜想被称为“孪生素数猜想”。

【实例 3.25】 对于任何一个正整数 n，如果是奇数则乘 3 并加 1，如果是偶数则除于 2，重复此过程最终总可以得到 1。

$$\forall n,\exists m,[f^{(m)}(n)=1],\text{其中 } f(n)=\begin{cases}3n+1, & n\equiv 1(\bmod 2)\\ \dfrac{n}{2}, & n\equiv 0(\bmod 2)\end{cases}$$

解释：这里 $f^{(m)}(n)$表示将 f 复合作用在 n 上 m 次，即 $f(f(\cdots f(n)\cdots))$。这个猜想被称之为“角谷猜想”，或者“奇偶归一猜想”“$3n+1$ 猜想”等，目前这个猜想还未被解决。

3.2 图灵机模型

3.2.1 定理机器证明与吴方法

机器证明，又称数学机械化，要求在证明过程中，每前进一步之后，都有一个确定的规则来选择下一步，沿着这一路径前进，最终到达需要的结论。通过这样方式，人们希望回避开那些技巧性极强的数学证明，用现代计算机强大的计算能力来取而代之。

机器证明的思想可以回溯到 17 世纪法国数学家笛卡儿(Rene Descartes，1596—1650)。笛卡儿曾经有过一个伟大的设想：“一切问题都可以化为数学问题，一切数学问题都可以化为代数问题，一切代数问题都可以化为代数方程的求解问题。”笛卡儿并没有只停留在空想，他所创立的解析几何，建立了空间形式和数量关系之间的桥梁，实现了初等几何问题的代数化求解。

20 世纪初希尔伯特(David Hilbert，1862—1943)更明确地提出了公理系统的机械化判定问题：给定一个公理系统，是否存在一种机械的方法(即现在所谓的算法)，可以验证每一个命题是否为真？在下面章节我们将会看到，希尔伯特的要求太高了。哥德尔不完备性定理指出：能对所有的命题进行机械化判定的方法是不存在的。

虽然希望使用一个算法来对所有的命题进行判定是做不到了，但是针对一些特定领域的具体的问题，使用机械化的方法仍然是可行的。例如，吴文俊先生提出的以多项式组零点集为基本点的消元方法(吴方法)可以应用于大量几何定理的机器证明。

图论中著名的四色定理断言：任何平面图都可以 4 染色，使得任何相邻的顶点都不同色。四色定理最早由格斯里(Francis Guthrie，1831—1899)在 1852 年提出，这个问题困扰了数学家一百多年，最终在 1976 由阿佩尔(Kenneth Appel)和哈肯(Wolfgang Haken)利用计算机所证明。他们证明的思路是这样的：如果在一张平面图中出现了某种特定的结构，那么他们就可以把这一个局部用一个更小的结构替代(即对原图进行约简)，同时保持 4 染

色性质的不变。也就是说，如果新的图能够被 4 染色，那么原来大的图也可以被 4 染色。例如，一个度不大于 3 的顶点就要可以被去掉，因为它不会影响整张图是否可以被 4 染色。阿佩尔和哈肯证明，一共有 1936 张互相不可约简的平面图，对于其他任何一张平面图，总可以通过不停地约简图中特定的结构，最终到达这 1936 张图中的某一张。最后他们借助计算机的帮助，经过超过 1000 小时的计算，终于验证了所有这 1936 张图都可以被 4 染色，进而证明了四色定理。四色定理是第一个用计算机辅助证明的数学定理。

3.2.2 有穷自动机

在介绍图灵机之前，让我们从一个简单的自动售卖机开始：一台自动售卖机可以售卖多种商品，可以接受多种币值的纸币或者硬币，同时还可以找零。为了简化问题，我们假设自动售卖机只卖两种商品——可乐和饼干，可乐的价格是 5 元 1 听，饼干的价格是 10 元一包，假设售卖机只能接受 5 元和 10 元的纸币，同时任何时刻售卖机内部剩余的金额最多不超过 10 元。

让我们用一种数学化的语言来描述自动售卖机的工作原理，我们用一些状态来表示自动售卖机的状态。首先售卖机有一个初始状态，我们将其记作 q_0。如果此时用户塞入一张 5 元的纸币，售卖机将进入一个新的状态，为了区分该状态与 q_0 的区别，我们将这一个状态记作 q_1。如果在状态 q_1 下用户选择购买一听可乐，那么售卖机将吐出一听可乐，并回到状态 q_0。另一方面，如果在 q_1 状态下用户再塞入 5 元，那么售卖机内一共有 10 元，从而进入另外一个状态 q_2，现在用户可能会选择购买可乐（或者饼干），那么售卖机将吐出可乐（或者饼干），并根据剩余的金额进入相应的状态 q_1（或者 q_0）。在任何状态下，如果用户选择“终止交易”，则售卖机会退回剩余金额，并回到 q_0 状态。

相比于上面的状态转移规则，状态转移图（图 3.5）更加清晰并简洁描述了售卖机的功能。

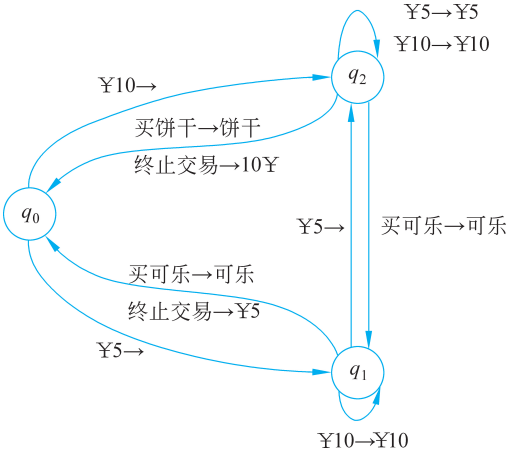

图 3.5 售卖机的状态转移图

其中，箭头（→）前面表示售卖机的输入，箭头后面表示售卖机的动作。例如，在 q_2 到 q_1 的路径上的“买可乐→可乐”表示如果售卖机在 q_2 状态，用户选择购买可乐，则售卖机跳

到 q_1 状态并吐出一听可乐。

自动售卖机所对应的计算模型被称为“有穷自动机”，也称为有限状态自动机(finite state automata)。这一计算模型基本上可以看作和我们计算机的 CPU 对应，但它的一个重要不足是该模型没有计算机的“内存”，也就是说没有存储器，只能通过自动机的状态来做存储。

3.2.3 图灵机

在前面的章节中我们通过介绍布尔逻辑和一阶逻辑，使得大家对逻辑有了初步的了解，我们看到所有的数学问题、所有的计算和证明推导过程都可以使用逻辑的语言来精确地描述，我们也看到了使用“机器”可以自动地来证明一些数学定理，我们也提到了希尔伯特的“公理系统机械化判定的思想”，到底希尔伯特心目中的通用的“机器”是什么样子？1936 年图灵给出了他的回答——图灵机。

图灵机既有处理器(CPU)也有存储器(内存)。下面我们严格定义图灵机。

【定义 3.4】 图灵机。

图灵机是一个七元组：$\{Q,\Sigma,\Gamma,\delta,q_0,q_{accept},q_{reject}\}$，其中，$Q$、$\Sigma$、$\Gamma$ 都是有限集合，B 是空白字符(Blank)的记号。

(1) **状态集合**：Q。

(2) **输入字母表**：Σ。

(3) **带字母表**：Γ，其中 $B\in\Gamma$。

(4) **转移函数**：$\delta:(Q-\{q_{accept},q_{reject}\})\times\Sigma\to Q\times\Gamma\times\{\rightarrow,\leftarrow\}$。

(5) **起始状态**：$q_0\in Q$。

(6) **接受状态**：q_{accept}。

(7) **拒绝状态**：q_{reject}。

图灵机(单带)是这样完成计算任务的：给定一个写有输入的右端无限长纸带，图灵机从纸带上输入的最左端出发，从初始状态开始，按照转移函数进行状态转移以及左右移动和写操作，最终进入接受状态或者拒绝状态。 ■

也可以用状态转移图来描述图灵机(下面的讨论中，我们都用状态转移图来描述图灵机)。

【实例 3.26】 判定一个 0-1 字符串中包含 1 的个数是奇数还是偶数。

分析：可以用两个状态来区分当前已经读过的串中 1 的个数是奇数还是偶数，如图 3.6 所示。

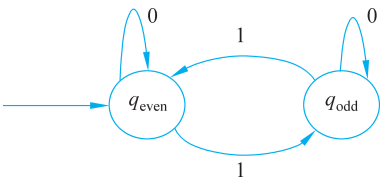

图 3.6 状态转移图：已经读过的串中 1 的个数是奇数还是偶数

其中，q_{even} 表示目前读到了偶数个 1；q_{odd} 表示目前读到了奇数个 1。自动机初始状态为 q_{even}。任意状态下，读到 1 就会跳到另一个状态。

【实例 3.27】 给定字符串 1^m0^n，判断 m 是否等于 n，如果 $m=n$ 输出 Yes，否则输出 No。

分析：最容易想到的方法是先统计 1 和 0 的个数，然后再比较。但是图灵机只有有限的存储(控制器中的状态)，不可能只使用内部状态来记录下字符串中所有的 1 或者 0(事实上可以证明，如果只用内部状态做记录，即不能对纸带

进行写操作的话，图灵机是不可能完成这个任务的）。所以如果想统计字符串中 1 和 0 的个数，可以开辟纸带上某个空白的地方做计数器。但是这样的图灵机的状态转移图画出来太复杂。

方案 1：一种不需要统计 1 和 0 的个数，直接来判断 1 的个数和 0 的个数是否相同的方法。具体的做法是：将 0 和 1 进行配对（即建立一个 1-1 对应），如果能够完全配对，则 0 和 1 一样多，否则不一样多。图灵机的设计如图 3.7 所示。

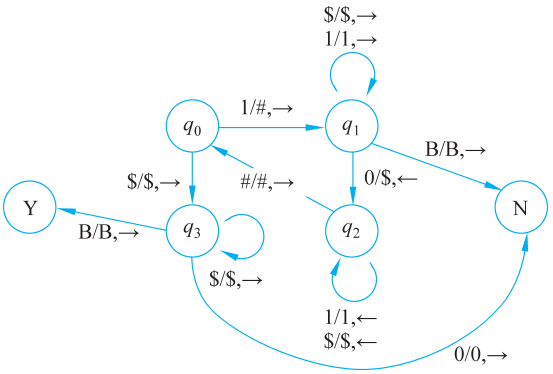

图 3.7 方案 1 图灵机状态转移图

图灵机读写头初始在纸带的最左端，然后往右移动，如果读到 1，置为 ♯，然后继续往右移动直到读到 0，替换为 $，或者没有读到 0，这时就可以断定 1 的个数和 0 的个数不相等，输出 No(N)。然后读写头再回到纸带的左端重复上述过程，直到最后如果纸带上只剩下 1 或者 0，则输出 No(N)，如果纸带上既没有 0 也没有 1，那么输出 Yes(Y)。

不难看出，如果输入的字符串为 1^n0^n，图灵机的运行步数为 $O(n^2)$。

方案 2：采用二分法的思想，将判断“$m=n$?”转化为比较 m 和 n 在二进制下的每一比特是否都相等。这里有一个问题，如何才能够得到 m 和 n 在二进制表示下的某一位是什么？图 3.8 给出了一个巧妙的方法，它不需要把 m 和 n 先算出来，就可以给出每一位。具

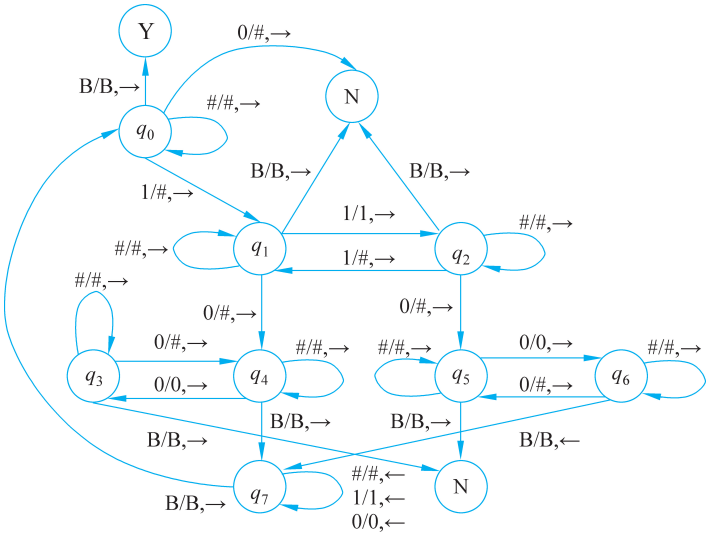

图 3.8 方案 2 图灵机状态转移图