
目录

第零章 开始之前	1
0.1 从语言的发展了解计算机	1
0.2 内存、程序与数据	5
0.3 算法与算法的复杂性	7

第零章 开始之前

0.1 从语言的发展了解计算机

计算机，顾名思义，就是一台可以计算的机器。一般来说，作为一台冷冰冰的机器，它是没有智能的。即“只能根据外部的指令做出预先设定好的几种动作”。这些动作实际上是对计算机的五大基本部件进行操作。

这五大部件有“输入设备”、“输出设备”、“运算器”、“控制器”、“存储器”。输入设备和输出设备顾名思义，就是将数据输入到计算机和从计算机中得到结果的设备。而运算器、控制器和存储器我们稍后会谈到。

利用这五大部件组成的计算机，我们称之为基于“冯诺依曼”结构。这五大部件可不是凭空捏造，而是来之有据的！

早在 1930 年代，人就开始构思一种还不存在的、能执行命令的机器——图灵机。这种机器的发明来源是模仿人类利用纸和笔进行数学运算而得来的。

想要进行数学运算，我们需要一些工具：首先是答题纸，我们需要用答题纸来记述每一步的结果。然后需要草稿纸来暂存计算的中间步骤。然后我们需要笔在纸上写字，需要眼睛来阅读纸上的内容。然后我们需要一系列的基本运算规则例如四则运算、公式定理。只需要这些东西我们就可以构建出无穷的数学世界！

图灵机就是仿照上面描述而虚构出的一种设备。它有一张无限长的纸带（如同纸）：纸带被划分出连续的一串格子，每个格子都用唯一的编号来组织在一起。然后还有一个读写头（笔和眼）：通过读写头我们可以阅读或者修改纸带上的数据。同时还需要一套控制规则（公式定理运算律）：根据当前机器所处状态和当前读写头所指的格子上的符号来确定读写头的下一

步动作，并且改变状态寄存器的值。它还有一个状态寄存器（草稿纸）：用来保存图灵机当前所处的状态，例如：状态 -正常运行；状态 -停机。等等。

图灵机就是这样一台机器，图灵认为它能模拟人类所能进行的任何计算过程。

1945 年，冯·诺依曼发表文章——“关于 EDVAC 的报告草案”。该方案描述了现代电子计算机由“运算器”、“控制器”、“存储器”、“输入设备”和“输出设备”五大部分组成，并且描述了这五大部分之间的关系。并且设计了这台基于二进制的 EDVAC。它的功率为 56KW，占地面积近 50 平方米，重近 8 吨，需要三十个技术人员同时操作。

后来各大公司生产的通用计算机，速度越来越快、功能越来越强大、编程方式越来越简单，但是其根本结构还是未能突破这种几十年前冯诺依曼的设计。

那么，向机器发号施令、协同五大部件工作的人就是程序员。。

而计算机程序设计其实就是程序员将现实的问题转换成计算机可以理解的方式，并且让计算机去执行自己的指令，然后得到结果。

也就是说，计算机程序其实就是一种能够被计算机所识别的指令的序列。计算机只能根据程序中所规定的内容一步一步地进行运转。

那么这种指令是如何被识别被执行的呢？这就要说起计算机的核心部件——CPU 了。CPU 是 Central Processing Unit 的缩写，中文名称叫做“中央处理器”。CPU 内包含了“运算器”和“控制器”两大部分。它是计算机的核心部件。我们可以将它看作是计算机系统的大脑。它被计算机指令所控制（控制器的功能）、根据指令进行运算（运算器的功能）。

而程序员所编制的程序就会被 CPU 去执行。那 CPU 究竟会识别什么样的程序呢？是这样的程序：

代码 1: 机器代码

1	0110	1001	0110	1010	0101	1001	0101	1010
2	1101	1101	1111	0001	0100	1011	0101	0111
3	1001	0110	1101	0100	1001	0011	1111	1101
4	1001	1110	1000	0111	0001	1011	0100	0000
5	0001	0000	0011	0001	0110	1001	1110	0110

上面的代码 1 就是计算机能识别的一种语言。对于人来说它是非常的晦涩难懂，以至于我都不知道上面的代码究竟是什么含义。但是对于计算机来说，处理这种纯数字的代码是计算机非常擅长的事情。我们将其称为“机器语言”

我们再来探索一下上面的代码。每一行是由 32 个 0 或者 1 的字符组成的。CPU 可以轻松识别这些字符的含义。假设识别上面这种语言的计算机每一条指令长度是 32 个 0 或 1 字符组成，我们称这台计算机的字长为 32 位。

我们再假设，这台计算机的每条指令分为两个部分，前半部分的 16 位是“操作码”，CPU 在看到这 16 位的内容时就能识别出来自己所需要进行的工作。

然后指令后半部分的 16 位称为“操作数”。这部分表明该条指令所进行的操作的目标是哪里，或者携带一些关键的信息以协助该操作。

在厂商设计并且生产号一台 CPU 之后，会随着它附带一份上百或者上百页的“技术手册”。这本手册上写明了该型号 CPU 的各种参数、工作方式。最重要的是，它告诉了读者“应该用怎样一种语言来命令 CPU”。

假设你现在是工地的包工头，你在指挥来自世界各地的工人（而且他们都足够笨，笨到你需要非常详细地教他们该如何办事）。你让每个工人都给你拿来一块砖。对中国工人应该说：“从地上捡起一块砖，走到我面前，然后放下”。对美国工人应该说：“Pick up a piece of brick, come up to me, and then put down”。可见对于相同的工作，不通的人需要不通的理解方式。而且当工人太笨的时候，你的“指导”工作大量增加。

机器语言也是类似。不同厂商甚至仅仅是不通型号的 CPU 能接受的指令格式都不太相同。也就是说，上面的机器语言在这台机器上能运行，但是换了另一台机器就不一定能用了。而且如果只是为了让工人去搬一块砖还要费那么多口舌（指令），其实不是非常麻烦！要是能发明一种简单通用的“世界语”，命令只需要用世界语下达一次，无论是亚洲人还是非洲人，都能听懂并且去工作。

所以，你召集了手下一大群工人开会。你规定：“当我举起红色旗子时，你们就立刻从地上拿起一块砖。当我举起黄色旗子时，就都向我走过来。当我举起绿色旗子时，就都把砖放在地上”。

这下方便了，进行同样的事情，你只需要依次举起红色、黄色、绿色旗子就够了，不用多费口舌。而你所付出的开销就是将原来冗长的命令换成一个旗子。作为工人，他们所付出的开销就是当看到某种颜色的旗子，就把它转换成对应的指令。就是这么简单！

计算机厂商发现，如果用上面这种“替换”的方式，很容易做出简单、可读性强、并且有可能在多种不同机器上通用的语言，于是他们就发明了一种语言，叫做“汇编语言”。

汇编语言其实就是对机器语言进行了简单的替换。用几个简短的字母替代了原来 16 位的操作码。用简短的标号替代了原来 16 位的操作数。

代码 2: 汇编语言

```
1 0110 1001 0110 1010 0101 1001 0101 1010 LDI #100
2 1101 1101 1111 0001 0100 1011 0101 0111 ADD R1,R2,R3
```

见上面的代码 2，LDI 和 ADD 是操作码，空格后面的 #100 和 R1,R2,R3 是操作数。我们把 (0100 1001 0110 1010) 这么长而且难以记忆的操作码变成了像 LDI 这么简洁的方式表达。

计算机还是无法识别像汇编语言这样的语言，所以我们还是需要一种叫做“汇编工具”的程序，将汇编语言转换成真正的机器语言才能被计算机所识别。那也就是说，对于同一条汇编指令，我们可以在“转换”的时候，根据程序将会运行的目标平台，转换成不同的机器语言。只要我们做的事情一样，即便机器语言不一样，通过使用不同的转换规则，我们就能够达到同样的效果！汇编语言就这么出现并且持续发展了几十年。

虽然汇编语言看起来那么的美好，但是它毕竟离我们人类的认知方式有些遥远。我们在计算数学题的时候可以说：“将上述过程重复 100 次”（循环结构）。或者有分段函数：“如果 $x < 0$ 则 $y = -1$ ；如果 $x > 0$ 则 $y = 1$ ”（选择结构）。像这样的表述，我们利用汇编语言很难“优雅”地实现。这时候如果能有一种语言能够把上面这几种结构轻松表达出来就好了！

同时，汇编语言通过统一的操作符带来了一定的通用性（可移植性）。但是像操作数是对寄存器进行操作，而不同的计算机系统的寄存器命名方式各有不同，所以它的通用性还是不是很强。这时候如果有一种语言，能够完全忽略寄存器，仅仅像数学运算那样对“变量”进行操作就好了！

于是，高级语言出现了。

高级语言分为很多种，我们在这里只论述一种叫做“命令式语言”的高级语言。这种语言的语义基础是要模拟“数据存储/数据操作”的图灵机模型。十分符合现代计算机的体系结构。

如果说冰冷的机器与人类之间有一道鸿沟，汇编语言、高级语言的出现，都在不断地缩小这一种鸿沟。

高级语言的好处非常多：易学易懂、能够将大量的精力放在算法的实现而非计算机系统底层的研究中。而且由于它的抽象程度高更高，所以可移植性就会更好。

0.2 内存、程序与数据

现在我们来通过程序解释一下冯诺依曼结构计算机的工作原理。上面我们曾经提到过，计算机是被程序控制的。那么程序的执行流程到底是什么样呢？

首先，我们要理解程序的物理结构。一般来说，不论我们用多么高级的语言编写程序，最后都要被转化成当前计算机所能识别的机器语言。这些机器语言按照严格的先后顺序存储在计算机的外存（如硬盘）上。当需要执行的时候，则将其调入主存（内存）。然后由 CPU 控制，按照严格的顺序逐条执行指令。

那么主存（内存）到底是怎样的一种结构？

正如之前我们所讲述的图灵机上的纸带。我们可以将内存想象成类似的形式。在这里我喜欢用“抽屉”作为举例。

假设你有一个非常非常高的柜子，上面从上到下依次排满了抽屉。为了方便定位，你在每个抽屉都编上了号。例如，最上面的抽屉是 0 号，往下一层的抽屉是 1 号，以此类推。既然是抽屉，那在其中肯定能放一定量的东西。由于每个抽屉尺寸规格是完全一致的，所以抽屉中能存放的东西也是一致的。这种柜子其实就是内存最简单的抽象。

总结一下，内存的特性就是这个柜子的特性：有编号、编号是顺序排列的（由 0、1、2 递增）、在每个编号中有一块空间。好了，这就是内存！

当然关于内存还有一些你必须知道的细节：内存的编号从 0 开始到整

个内存的结束称为寻址空间。一般地，每个编号下对应的存储空间为 8 位，我们称其为一字节。也就是说：每一个地址对应一个能存储 1 字节数据的空间

一般来说，在有操作系统的计算机上，程序的执行是受到操作系统控制的，程序执行时的变量等数据也是由操作系统所管理的。对于程序员，我们必须要了解自己程序在操作系统的管理下的运行机制，才能让我们编写出更好的程序。

与“程序”遥相呼应的就是“数据”。数据可以指在程序运算过程中临时存储在主存上的内容。不严格地说，我们可以将这种临时的数据称为变量。如何理解“变量”？变量就如同在数学中的 x, y, z 。在计算机中，变量是一种能够被赋值，能够被取值的元素。它对应着内存上的某个编号，在该编号下以一定的长度存储着数据。

变量是内存上某段数据的逻辑表现，是组成程序的重要元素。例如，假设我们在编号为 A 的杯子中加满牛奶，在编号为 B 的杯子中加满果汁。现在我们想要让两个杯子中的东西交换一下，即在 B 中装满牛奶，在 A 中装满果汁。这时候我们都会想到解决办法：先找到一个杯子 C，将 A 中的牛奶倒入 C 中。再将 B 中的果汁倒入 A 中，再将 C 中的牛奶倒回到 B 中。如此一来，我们就实现了 A 与 B 中内容的交换。在这里 A、B 和 C 都是变量，这个编号就是内存地址，杯子中的空间就是在这个地址下内存中的空间。而牛奶和果汁就是数据。这种利用杯子 C 来让 A 和 B 中的内容互换的流程就称之为“算法”。

对于高级语言来说，可以用如下的伪代码表示：

代码 3: 两个变量交换数据

```
1 a = 1;    // 让 a 等于 1
2 b = 2;    // 让 b 等于 2
3 c = a;    // 让 c 等于 a
4 a = b;    // 让 a 等于 b
5 b = c;    // 让 b 等于 c
```

我们把上面这段代码 3 称为伪代码。是因为这段代码并不是某种计算机能识别的代码，而是人类用来表达想法的工具。我们将会在后来的课程

学习到真正的计算机语言。

0.3 算法与算法的复杂性

高级语言的出现，让硬件设计与软件设计工作逐渐分离。作为程序员，我们所关注的内容主要是控制机器运转的一系列规则。而机器怎么理解这些规则的描述是硬件工程师的工作，与我们没有太大的关系。所以我们工作的重点在于设计一套解决问题的逻辑流程。这套流程的核心就是算法。

算法其实就是完成某种事情的步骤，也就是将某件复杂的事情拆分成很多道简单的工序。例如把 A 杯与 B 杯中的内容互换：1, 从 A 到 C; 2, 从 B 到 A; 3, 从 C 到 B。这样的工序就是一种算法。而简单地算法相互组合又能实现出复杂的算法。例如我们有 5 个高矮各不相同的人排成一列，现在我想让长得最高的人排到第一个位置上，该如何去设计这个算法？

这里为了方便，我们用 `queue` 表示这个队列，用 `queue[i]` 表示第 *i* 个人。

例如 `queue[1]` 就是指排头，`queue[5]` 就是队尾。我们的算法可以这样简单的描述：

代码 4: 按身高排队

```
1 如果 queue[5] 的身高 > queue[4] 的身高则这两人交换位置
2 如果 queue[4] 的身高 > queue[3] 的身高则这两人交换位置
3 如果 queue[3] 的身高 > queue[2] 的身高则这两人交换位置
4 如果 queue[2] 的身高 > queue[1] 的身高则这两人交换位置
```

代码 4 中的每一个小步骤（元素交换），其实都可以看做是程序 3 的一段变体。

我们现在成功地实现了从一条队伍中找到最高的人并且排到最前面。但如果我想让整个队伍从大到小排列该如何做呢？其实如果你明白了上面两个例子的相关性，这道题一定不会难到你！

在执行第一遍代码 4 后，排在第一个的人是队伍里最高的，如果把上面的代码再执行一次，一定会让第二高的人排在第二次！也就是说，如果有 *n* 个人，我就把上面的代码执行 *n* 次，最终得到的结果肯定是已经按顺序排列完毕的队伍！很神奇吧？

这里我常常用中国传统道家文化来类比。道家文化认为万物都是由阴阳两种物质组成，所谓两仪生四象，就是 $2 \times 2 = 4$ ；四象生八卦就是 $4 \times 2 = 8$ 。如此一来，世间万物皆可由太极所产生，无穷无尽。

可见，很对复杂的算法其实都是通过一些简单的算法所组成的。而我们后续课程将会为大家从简单算法讲到复杂的算法！

我们再来讨论讨论算法效率的问题。还是上面道排序问题。

如果对于 5 个人的队伍而言，代码 4 中一共需要进行 4 次操作（我们将比较和交换看作是一个操作）。然后我们需要执行五次代码 4，这样就一共需要 $5 \times 4 = 20$ 次操作。也就是说如果对于 n 个人的队伍而言，共需要进行 $n \times (n - 1)$ 次操作。

这样的是不是有些多呢？有没有更好的方法？我们考虑一下对于上面算法的一个小小的改进。如果一条队伍已经执行了一次代码 4，则排在第一位的人肯定是最高的！则执行第二次代码 4 时，第四步就完全没必要了（因为 `queue[2]` 永远不可能大于 `queue[1]`）。那只执行前三步就足够了，依次类推，我们的计算量是 $4 + 3 + 2 + 1 = 10$ 次操作。对于 n 个人的队伍而言，共需要进行 $(n - 1) \times n / 2$ 次运算。可见这个修改后的算法计算步骤还是会比原来的算法要快一些。

解决同一个问题有多种方法，那么我们该如何衡量某种方法的优劣呢？那我们就要看看程序运行所需要的开销了。显然，程序运行的开销主要是“对存储器使用的开销”，包括内存使用量，外存使用量等等。我们称这种开销为“空间复杂度”。空间复杂度越高代表它所占用的空间越大。

还有一个开销是时间上的开销，计算机在进行一般操作的时候都需要 CPU 花费一些计算时间，而这种计算时间往往反映在操作“次数”上。操作的次数往往跟运算的规模有很大的关系，当数据规模是 n （类比队列中有 n 个人）时，时间复杂度就是 n 的函数（即时间复杂度是 $T(n)$ ）。

例如上面的排序，原始方法 $T(5) = 20$ ，改进后的方法 $T(5) = 10$ ，当然这个数值越小，就说明它的时间复杂度越小，即它所需要的时间也就越少。

但我们是不是就能说改进方法比原始方法快两倍？（因为 $20/10 = 2$ ）。当我们的运算量 n 非常大的时候。会发生什么？我们进行一点儿数学上的推导：

对于原始方法：

$$f_1(n) = \lim_{n \rightarrow \infty} (n \times (n - 1)) = \lim_{n \rightarrow \infty} (n^2 - n) = n^2 \quad (1)$$

设：

$$T_1(n) = O_1(f_1(n)) = O_1(n^2) \quad (2)$$

而对于改进方法：

$$f_2(n) = \lim_{n \rightarrow \infty} \left(\frac{n \times (n - 1)}{2} \right) = \lim_{n \rightarrow \infty} \left(\frac{1}{2}n^2 - \frac{1}{2}n \right) = n^2 \quad (3)$$

设：

$$T_2(n) = O_2(f_2(n)) = O_2(n^2) \quad (4)$$

可见，当 $n \rightarrow \infty$ 时， $T_1(n) = T_2(n)$ 。也就是说即便改进过的算法有一定的优势，但是当数据量足够大的时候，这种优势体现得就不再明显了。一般情况下我们用 $O()$ 的方式来表示算法的时间复杂度。

实际上， $T(n)$ 有很多种常见的情况。

$T(n)$	别称
$T(1) = O(1)$	常数复杂度
$T(n) = O(\log_2 n)$	对数复杂度
$T(n) = O(n)$	线性复杂度
$T(n) = O(n \log_2 n)$	$n \log_2 n$ 复杂度
$T(n) = O(n^2)$	平方阶复杂度
$T(n) = O(n^3)$	立方阶复杂度
$T(n) = O(n!)$	阶乘阶复杂度
$T(n) = O(2^n)$	指数阶复杂度
$T(n) = O(n^n)$	写出这种代码就去死吧复杂度

上面的复杂度在 n 很大的情况下，可以从小到大排列为： $O(1) < O(\log_2 n) < O(n) < O(n \log_2 n) < O(n^2) < O(n^3) < O(n!) < O(n^n)$