

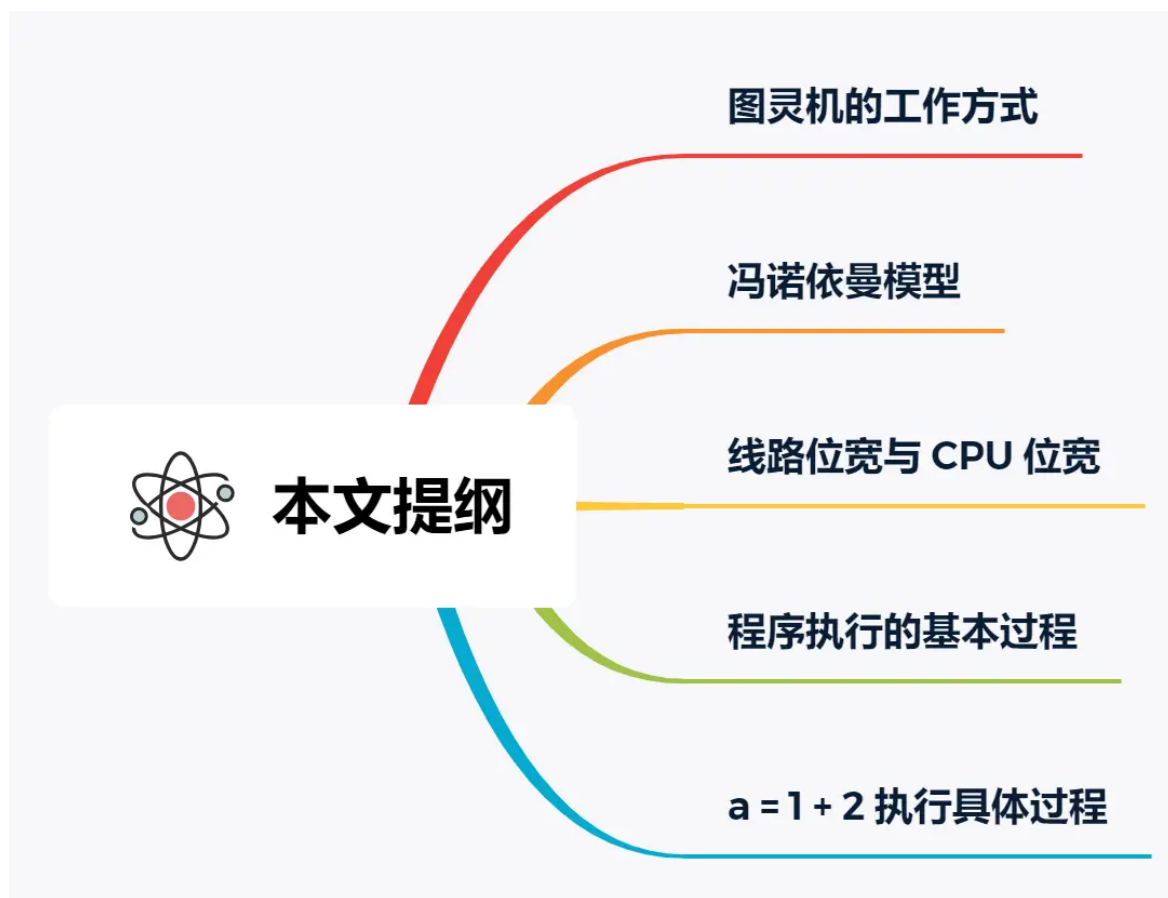
2.1 CPU 是如何执行程序的呢？

代码写了那么多，你知道 $a = 1 + 2$ 这条代码是怎么被 CPU 执行的吗？

软件用了那么多，你知道软件的 32 位和 64 位之间的区别吗？再来 32 位的操作系统可以运行在 64 位的电脑上吗？64 位的操作系统可以运行在 32 位的电脑上吗？如果不行，原因是什么？

CPU 看了那么多，我们都知道 CPU 通常分为 32 位和 64 位，你知道 64 位相比 32 位 CPU 的优势在哪吗？64 位 CPU 的计算性能一定比 32 位 CPU 高很多吗？

不知道也不用慌张，接下来就循序渐进的、一层一层的攻破这些问题。

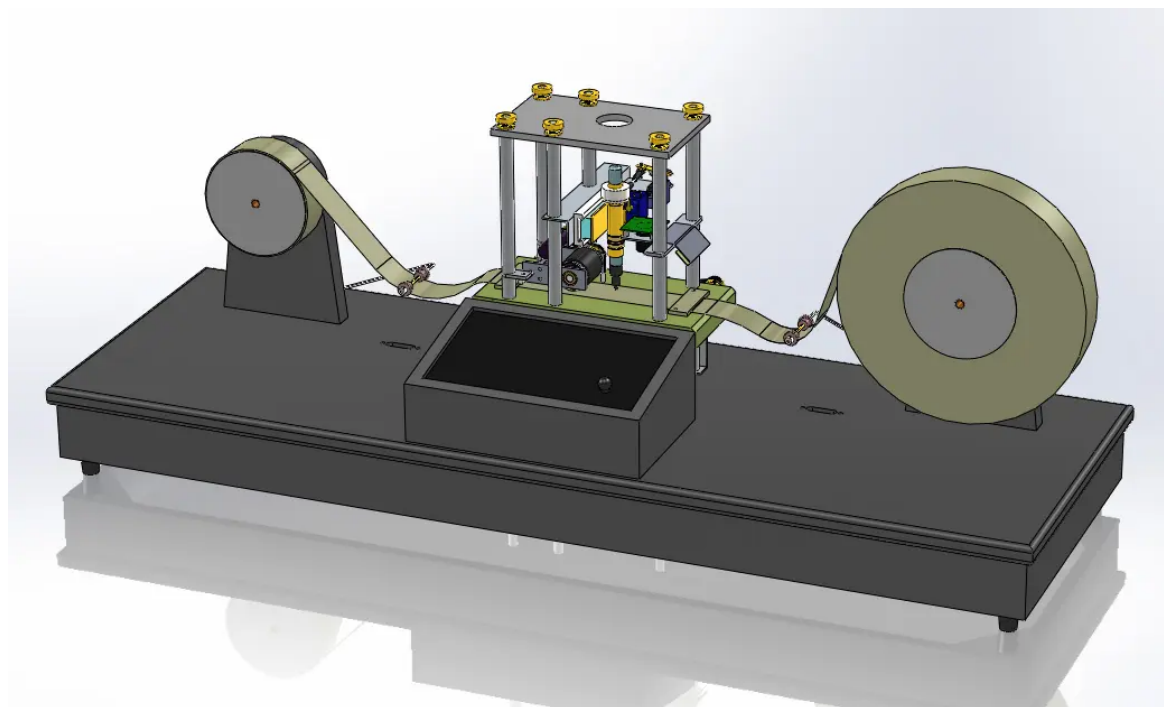


图灵机的工作方式

要想知道程序执行的原理，我们可以先从「图灵机」说起，图灵的基本思想是用机器来模拟

[目录](#)[侧边栏](#)[夜间](#)[技术群](#)[资料](#)[支持我](#)[上一篇](#)[下一篇](#)

图灵机长什么样子呢？你从下图可以看到图灵机的实际样子：



图灵机的基本组成如下：

- 有一条「纸带」，纸带由一个个连续的格子组成，每个格子可以写入字符，纸带就好比内存，而纸带上的格子的字符就好比内存中的数据或程序；
- 有一个「读写头」，读写头可以读取纸带上任意格子的字符，也可以把字符写入到纸带的格子；
- 读写头上有一些部件，比如存储单元、控制单元以及运算单元：1、存储单元用于存放数据；2、控制单元用于识别字符是数据还是指令，以及控制程序的流程等；3、运算单元用于执行运算指令；

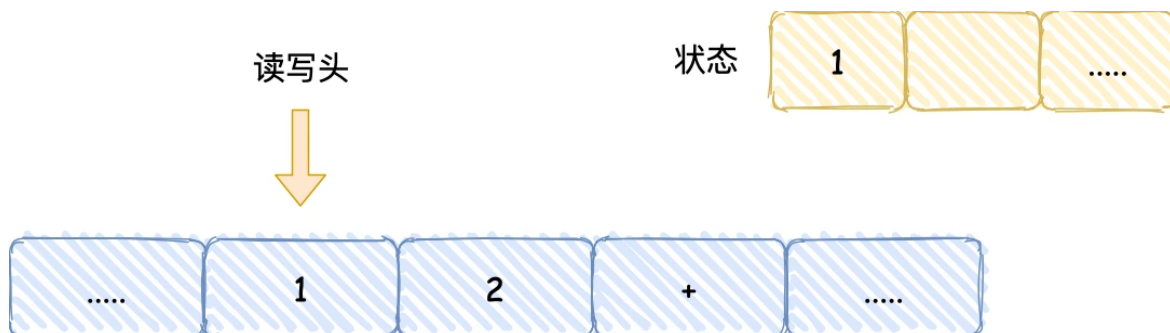
知道了图灵机的组成后，我们以简单数学运算的 $1 + 2$ 作为例子，来看看它是怎么执行这行代码的。

- 首先，用读写头把「1、2、+」这3个字符分别写入到纸带上的3个格子，然后读写头先停在1字符对应的格子上；

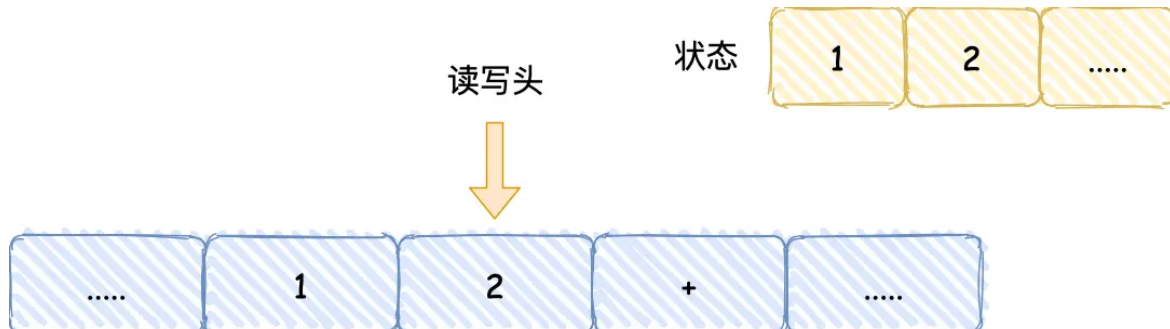
[目录](#)[侧边栏](#)[夜间](#)[技术群](#)[资料](#)[支持我](#)[上一篇](#)[下一篇](#)

[首页](#) [图解网络](#) [图解系统](#) [图解 MySQL](#) [图解 Redis](#) [学习路线](#) [网站动态](#) [Github](#)

- 接着，读写头读入 1 到存储设备中，这个存储设备称为图灵机的状态；



- 然后读写头向右移动一个格，用同样的方式把 2 读入到图灵机的状态，于是现在图灵机的状态中存储着两个连续的数字，1 和 2；



- 读写头再往右移动一个格，就会碰到 + 号，读写头读到 + 号后，将 + 号传输给「控制单元」，控制单元发现是一个 + 号而不是数字，所以没有存入到状态中，因为 + 号是运算符指令，作用是加和目前的状态，于是通知「运算单元」工作。运算单元收到要加和状态中的值的通知后，就会把状态中的 1 和 2 读入并计算，再将计算的结果 3 存放到状态中；



目录



侧边栏



夜间



技术群



资料



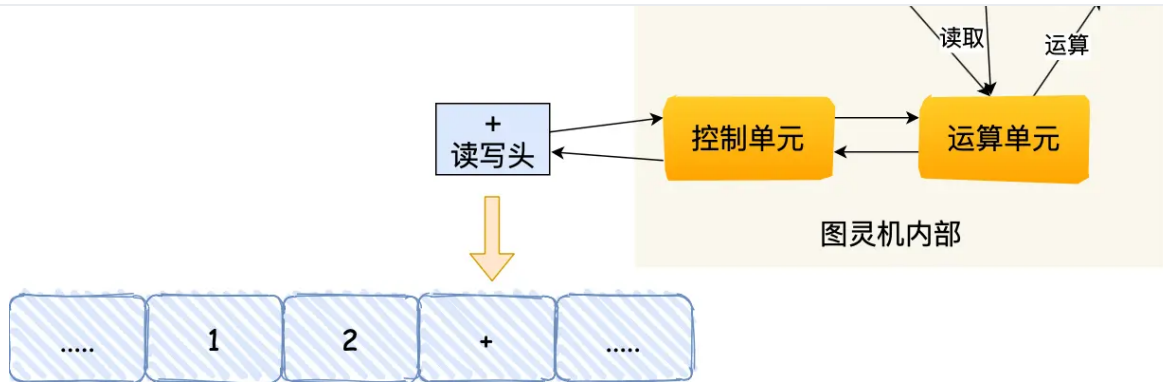
支持我



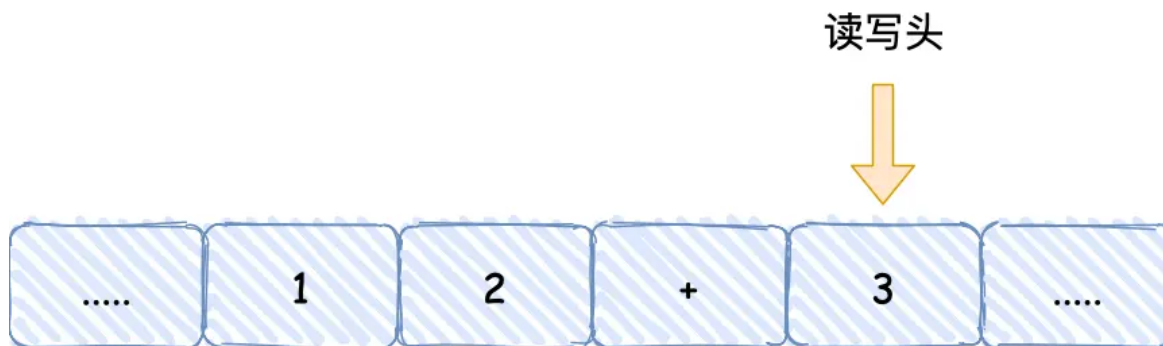
上一篇



下一篇



- 最后，运算单元将结果返回给控制单元，控制单元将结果传输给读写头，读写头向右移动，把结果 3 写入到纸带的格子中；



通过上面的图灵机计算 $1 + 2$ 的过程，可以发现图灵机主要功能就是读取纸带格子中的内容，然后交给控制单元识别字符是数字还是运算符指令，如果是数字则存入到图灵机状态中，如果是运算符，则通知运算单元读取状态中的数值进行计算，计算结果最终返回给读写头，读写头把结果写入到纸带的格子中。

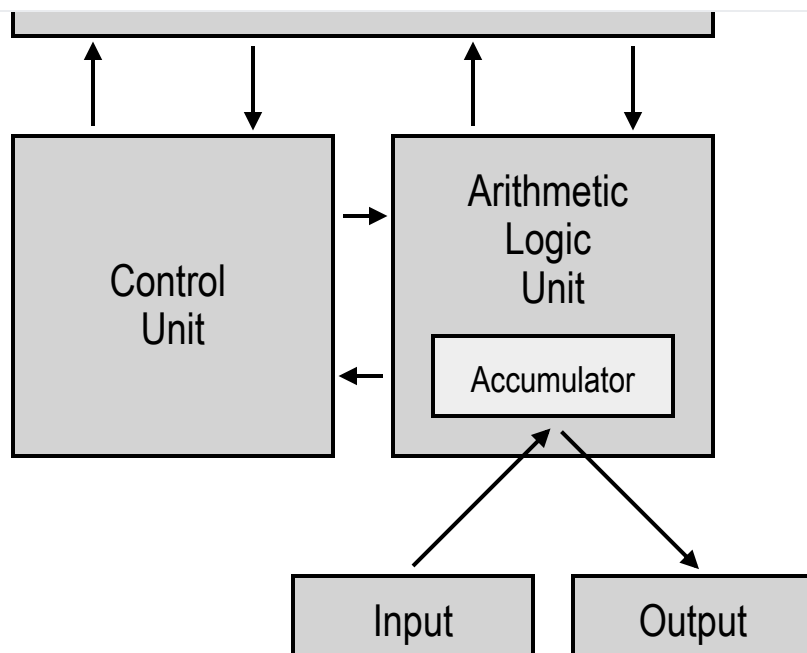
事实上，图灵机这个看起来很简单的工作方式，和我们今天的计算机是基本一样的。接下来，我们一同再看看当今计算机的组成以及工作方式。

冯诺依曼模型

在 1945 年冯诺依曼和其他计算机科学家们提出了计算机具体实现的报告，其遵循了图灵机的设计，而且还提出用电子元件构造计算机，并约定了用二进制进行计算和存储。

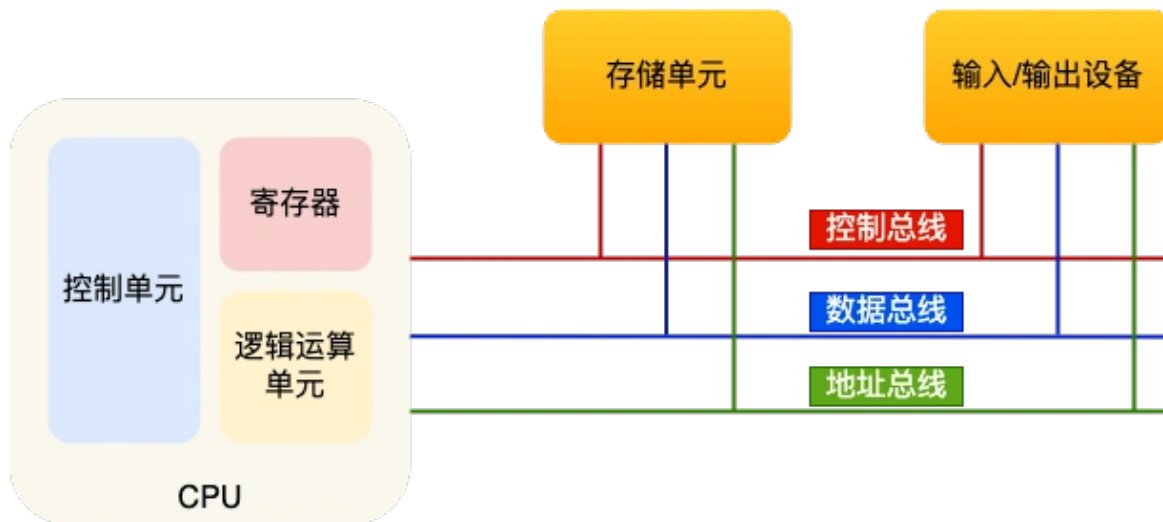
最重要的是定义计算机基本结构为 5 个部分，分别是**运算器**、**控制器**、**存储器**、**输入设备**、**输出设备**，这 5 个部分也被称为**冯诺依曼模型**。

[目录](#)[侧边栏](#)[夜间](#)[技术群](#)[资料](#)[支持我](#)[上一篇](#)[下一篇](#)

[首页](#) [图解网络](#) [图解系统](#) [图解 MySQL](#) [图解 Redis](#) [学习路线](#) [网站动态](#) [Github](#)

运算器、控制器是在中央处理器里的，存储器就是我们常见的内存，输入输出设备则是计算机外接的设备，比如键盘就是输入设备，显示器就是输出设备。

存储单元和输入输出设备要与中央处理器打交道的话，离不开总线。所以，它们之间的关系如下图所示：



接下来，分别介绍内存、中央处理器、总线、输入输出设备。

内存

我们的程序和数据都是存储在内存，存储的区域是线性的。

[目录](#)[侧边栏](#)[夜间](#)[技术群](#)[资料](#)[支持我](#)[上一篇](#)[下一篇](#)

内存的地址是从 0 开始编号的，然后自增排列，最后一个地址为内存总字节数 - 1，这种结构好似我们程序里的数组，所以内存的读写任何一个数据的速度都是一样的。

中央处理器

中央处理器也就是我们常说的 CPU，32 位和 64 位 CPU 最主要区别在于一次能计算多少字节数据：

- 32 位 CPU 一次可以计算 4 个字节；
- 64 位 CPU 一次可以计算 8 个字节；

这里的 32 位和 64 位，通常称为 CPU 的位宽。

之所以 CPU 要这样设计，是为了能计算更大的数值，如果是 8 位的 CPU，那么一次只能计算 1 个字节 0~255 范围内的数值，这样就无法一次完成计算 $10000 * 500$ ，于是为了能一次计算大数的运算，CPU 需要支持多个 byte 一起计算，所以 CPU 位宽越大，可以计算的数值就越大，比如说 32 位 CPU 能计算的最大整数是 4294967295。

CPU 内部还有一些组件，常见的有**寄存器**、**控制单元**和**逻辑运算单元**等。其中，控制单元负责控制 CPU 工作，逻辑运算单元负责计算，而寄存器可以分为多种类，每种寄存器的功能又不尽相同。

CPU 中的寄存器主要作用是存储计算时的数据，你可能好奇为什么有了内存还需要寄存器？原因很简单，因为内存离 CPU 太远了，而寄存器就在 CPU 里，还紧挨着控制单元和逻辑运算单元，自然计算时速度会很快。

常见的寄存器种类：

- **通用寄存器**，用来存放需要进行运算的数据，比如需要进行加和运算的两个数据。
- **程序计数器**，用来存储 CPU 要执行下一条指令「所在的内存地址」，注意不是存储了下一条要执行的指令，此时指令还在内存中，程序计数器只是存储了下一条指令的地址。
- **指令寄存器**，用来存放程序计数器指向的指令，也就是指令本身，指令被执行完成之前，指令都存储在这里。

总线

总线是用于 CPU 和内存以及其他设备之间的通信，总线可分为 3 种：

- **地址总线**，用于指定 CPU 将要操作的内存地址；

[目录](#)[侧边栏](#)[夜间](#)[技术群](#)[资料](#)[支持我](#)[上一篇](#)[下一篇](#)

响应，这时也需要控制总线；

当 CPU 要读写内存数据的时候，一般需要通过下面这三个总线：

- 首先要通过「地址总线」来指定内存的地址；
- 然后通过「控制总线」控制是读或写命令；
- 最后通过「数据总线」来传输数据；

输入、输出设备

输入设备向计算机输入数据，计算机经过计算后，把数据输出给输出设备。期间，如果输入设备是键盘，按下按键时是需要和 CPU 进行交互的，这时就需要用到控制总线了。

线路位宽与 CPU 位宽

数据是如何通过线路传输的呢？其实是通过操作电压，低电压表示 0，高压电压则表示 1。

如果构造了高低高这样的信号，其实就是 101 二进制数据，十进制则表示 5，如果只有一条线路，就意味着每次只能传递 1 bit 的数据，即 0 或 1，那么传输 101 这个数据，就需要 3 次才能传输完成，这样的效率非常低。

这样一位一位传输的方式，称为串行，下一个 bit 必须等待上一个 bit 传输完成才能进行传输。当然，想一次多传一些数据，增加线路即可，这时数据就可以并行传输。

为了避免低效率的串行传输的方式，线路的位宽最好一次就能访问到所有的内存地址。

CPU 要想操作的内存地址就需要地址总线：

- 如果地址总线只有 1 条，那每次只能表示「0 或 1」这两种地址，所以 CPU 能操作的内存地址最大数量为 2 (2^1) 个（注意，不要理解成同时能操作 2 个内存地址）；
- 如果地址总线有 2 条，那么能表示 00、01、10、11 这四种地址，所以 CPU 能操作的内存地址最大数量为 4 (2^2) 个。

那么，想要 CPU 操作 4G 大的内存，那么就需要 32 条地址总线，因为 $2^{32} = 4G$ 。

知道了线路位宽的意义后，我们再来看看 CPU 位宽。

CPU 的位宽最好不要小于线路位宽，比如 32 位 CPU 控制 40 位宽的地址总线和数据总线的话，工作起来就会非常复杂且麻烦，所以 32 位的 CPU 最好和 32 位宽的线路搭配，因为 32

[目录](#)[侧边栏](#)[夜间](#)[技术群](#)[资料](#)[支持我](#)[上一篇](#)[下一篇](#)

[首页](#) [图解网络](#) [图解系统](#) [图解 MySQL](#) [图解 Redis](#) [学习路线](#) [网站动态](#) [Github](#)

32 位数字和 2 个高位 32 位数字来计算，先加个两个低位的 32 位数字，算出进位，然后加和两个高位的 32 位数字，最后再加上进位，就能算出结果了，可以发现 32 位 CPU 并不能一次性计算出加和两个 64 位数字的结果。

对于 64 位 CPU 就可以一次性算出加和两个 64 位数字的结果，因为 64 位 CPU 可以一次读入 64 位的数字，并且 64 位 CPU 内部的逻辑运算单元也支持 64 位数字的计算。

但是并不代表 64 位 CPU 性能比 32 位 CPU 高很多，很少应用需要算超过 32 位的数字，所以**如果计算的数额不超过 32 位数字的情况下，32 位和 64 位 CPU 之间没什么区别的，只有当计算超过 32 位数字的情况下，64 位的优势才能体现出来。**

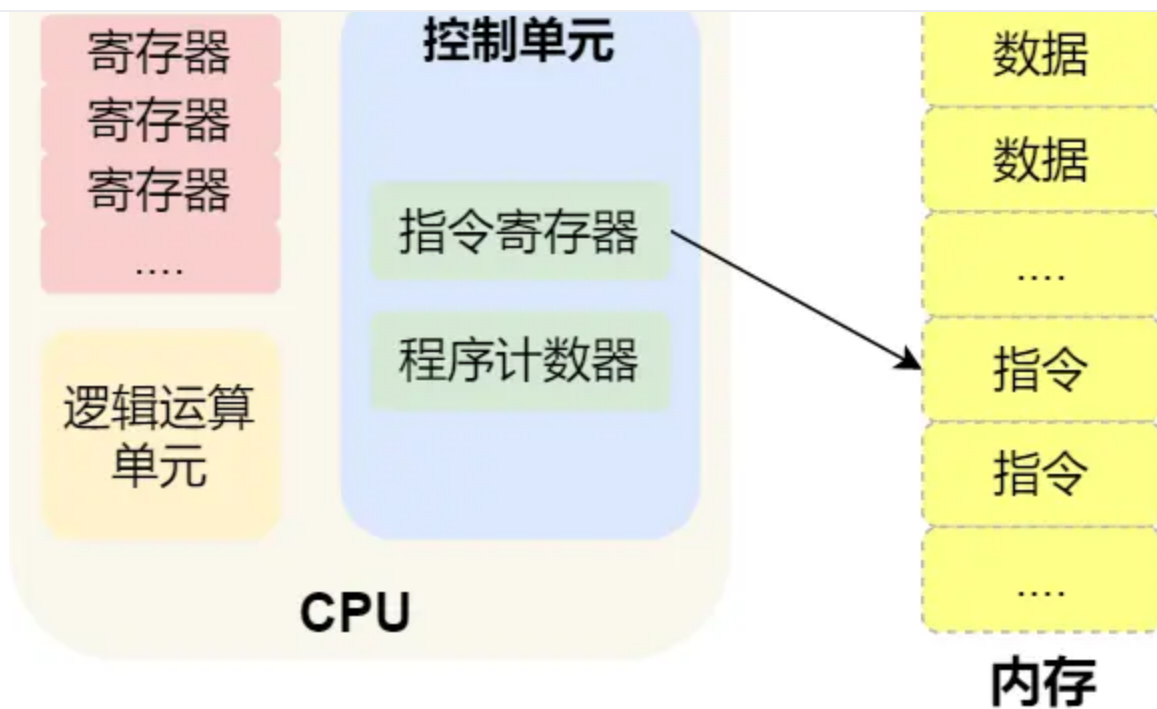
另外，32 位 CPU 最大只能操作 4GB 内存，就算你装了 8 GB 内存条，也没用。而 64 位 CPU 寻址范围则很大，理论最大的寻址空间为 2^{64} 。

程序执行的基本过程

在前面，我们知道了程序在图灵机的执行过程，接下来我们来看看程序在冯诺依曼模型上是怎么执行的。

程序实际上是一条一条指令，所以程序的运行过程就是把每一条指令一步一步的执行起来，负责执行指令的就是 CPU 了。

[目录](#)[侧边栏](#)[夜间](#)[技术群](#)[资料](#)[支持我](#)[上一篇](#)[下一篇](#)

[目录](#)[侧边栏](#)[夜间](#)[技术群](#)[资料](#)[支持我](#)[上一篇](#)[下一篇](#)

那 CPU 执行程序的过程如下：

- 第一步，CPU 读取「程序计数器」的值，这个值是指令的内存地址，然后 CPU 的「控制单元」操作「地址总线」指定需要访问的内存地址，接着通知内存设备准备数据，数据准备好后通过「数据总线」将指令数据传给 CPU，CPU 收到内存传来的数据后，将这个指令数据存入到「指令寄存器」。
- 第二步，CPU 分析「指令寄存器」中的指令，确定指令的类型和参数，如果是计算类型的指令，就把指令交给「逻辑运算单元」运算；如果是存储类型的指令，则交由「控制单元」执行；
- 第三步，CPU 执行完指令后，「程序计数器」的值自增，表示指向下一条指令。这个自增的大小，由 CPU 的位宽决定，比如 32 位的 CPU，指令是 4 个字节，需要 4 个内存地址存放，因此「程序计数器」的值会自增 4；

简单总结一下就是，一个程序执行的时候，CPU 会根据程序计数器里的内存地址，从内存里面把需要执行的指令读取到指令寄存器里面执行，然后根据指令长度自增，开始顺序读取下一条指令。

CPU 从程序计数器读取指令、到执行、再到下一条指令，这个过程会不断循环，直到程序执行结束，这个不断循环的过程被称为 **CPU 的指令周期**。

a = 1 + 2 执行具体过程

CPU 是不认识 `a = 1 + 2` 这个字符串，这些字符串只是方便我们程序员认识，要想这段程序能跑起来，还需要把整个程序翻译成**汇编语言**的程序，这个过程称为编译成汇编代码。

针对汇编代码，我们还需要用汇编器翻译成机器码，这些机器码由 0 和 1 组成的机器语言，这一条条机器码，就是一条条的**计算机指令**，这个才是 CPU 能够真正认识的东西。

下面来看看 `a = 1 + 2` 在 32 位 CPU 的执行过程。

程序编译过程中，编译器通过分析代码，发现 1 和 2 是数据，于是程序运行时，内存会有个专门的区域来存放这些数据，这个区域就是「数据段」。如下图，数据 1 和 2 的区域位置：

- 数据 1 被存放到 0x100 位置；
- 数据 2 被存放到 0x104 位置；

注意，数据和指令是分开区域存放的，存放指令区域的地方称为「正文段」。

	地址	内容
正文段 指令存放区域	0x20c	set R2 -> 0X108
	0x208	add R0 R1 R2
	0x204	load 0x104 -> R1
	0x200	load 0x100 -> R0

数据段 数据存放区域
	0x104	数据 2
	0x100	数据 1

编译器会把 `a = 1 + 2` 翻译成 4 条指令，存放到正文段中。如图，这 4 条指令被存放到了 0x200 ~ 0x20c 的区域中：

- 0x200 的内容是 `load` 指令将 0x100 地址中的数据 1 装入到寄存器 `R0` ；
- 0x204 的内容是 `load` 指令将 0x104 地址中的数据 2 装入到寄存器 `R1` ；
- 0x208 的内容是 `add` 指令将寄存器 `R0` 和 `R1` 的数据相加，并把结果存放到寄存器 `R2` ；
- 0x20c 的内容是 `store` 指令将寄存器 `R2` 中的数据存回数据段中的 0x108 地址中，这



目录



侧边栏



夜间



技术群



资料



支持我



上一篇



下一篇

条指令。

上面的例子中，由于是在 32 位 CPU 执行的，因此一条指令是占 32 位大小，所以你会发现每条指令间隔 4 个字节。

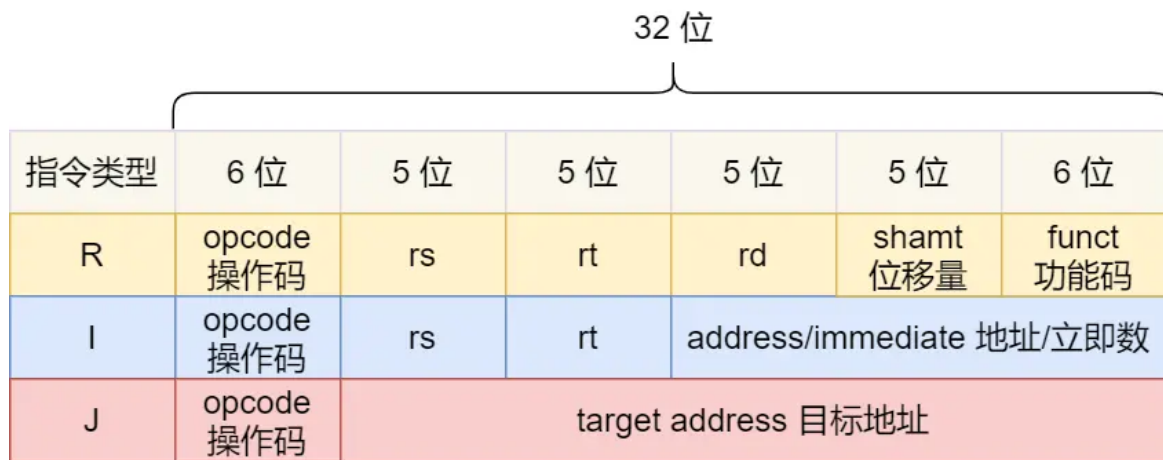
而数据的大小是根据你在程序中指定的变量类型，比如 `int` 类型的数据则占 4 个字节，`char` 类型的数据则占 1 个字节。

指令

上面的例子中，图中指令的内容我写的是简易的汇编代码，目的是为了理解指令的具体内容，事实上指令的内容是一串二进制数字的机器码，每条指令都有对应的机器码，CPU 通过解析机器码来知道指令的内容。

不同的 CPU 有不同的指令集，也就是对应着不同的汇编语言和不同的机器码，接下来选用最简单的 MIPS 指令集，来看看机器码是如何生成的，这样也能明白二进制的机器码的具体含义。

MIPS 的指令是一个 32 位的整数，高 6 位代表着操作码，表示这条指令是一条什么样的指令，剩下的 26 位不同指令类型所表示的内容也就不相同，主要有三种类型 R、I 和 J。



一起具体看看这三种类型的含义：

- **R 指令**，用在算术和逻辑操作，里面有读取和写入数据的寄存器地址。如果是逻辑位移操作，后面还有位移操作的「位移量」，而最后的「功能码」则是再前面的操作码不够的时候，扩展操作码来表示对应的具体指令的；
- **I 指令**，用在数据传输、条件分支等。这个类型的指令，就没有了位移量和功能码，也没有了第三个寄存器，而是把这三部分直接合并成了一个地址值或一个常数；

[目录](#)[侧边栏](#)[夜间](#)[技术群](#)[资料](#)[支持我](#)[上一篇](#)[下一篇](#)

把结果放入到 R2 中，翻译成机器码。

32 位							
指令	指令类型	操作码 6 位	rs 5 位	rt 5 位	rd 5 位	位移量 5 位	功能码 6 位
add	R	000000	00000	00001	00010	00000	100000
十六进制表示		0x00011020					

加和运算 add 指令是属于 R 指令类型：

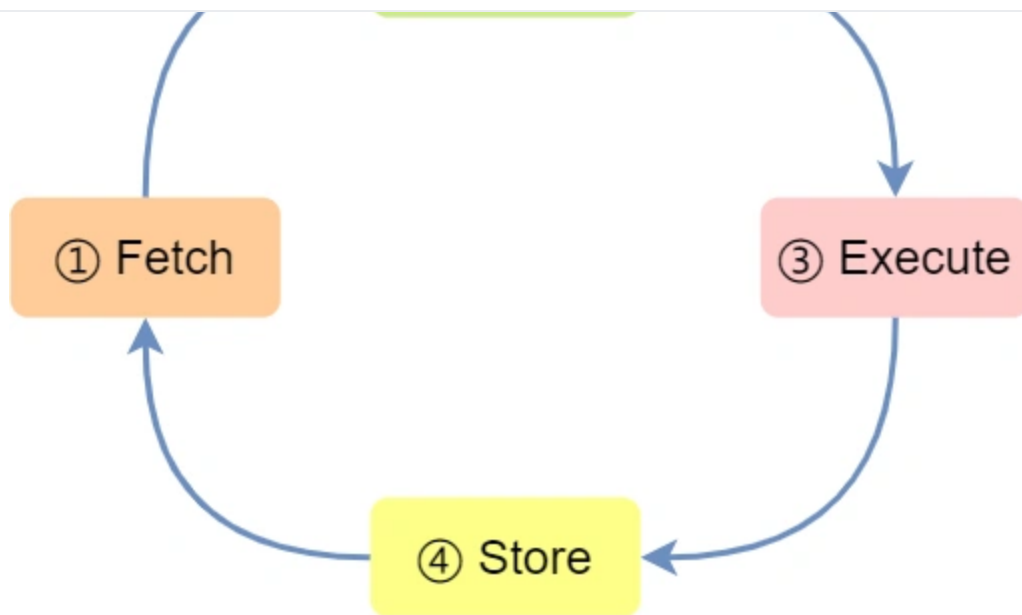
- add 对应的 MIPS 指令里操作码是 000000，以及最末尾的功能码是 100000，这些数值都是固定的，查一下 MIPS 指令集的手册就能知道的；
- rs 代表第一个寄存器 R0 的编号，即 00000；
- rt 代表第二个寄存器 R1 的编号，即 00001；
- rd 代表目标的临时寄存器 R2 的编号，即 00010；
- 因为不是位移操作，所以位移量是 00000

把上面这些数字拼在一起就是一条 32 位的 MIPS 加法指令了，那么用 16 进制表示的机器码则是 0x00011020。

编译器在编译程序的时候，会构造指令，这个过程叫做指令的编码。CPU 执行程序的时候，就会解析指令，这个过程叫作指令的解码。

现代大多数 CPU 都使用流水线的方式来执行指令，所谓的流水线就是把一个任务拆分成多个小任务，于是一条指令通常分为 4 个阶段，称为 4 级流水线，如下图：

[目录](#)[侧边栏](#)[夜间](#)[技术群](#)[资料](#)[支持我](#)[上一篇](#)[下一篇](#)



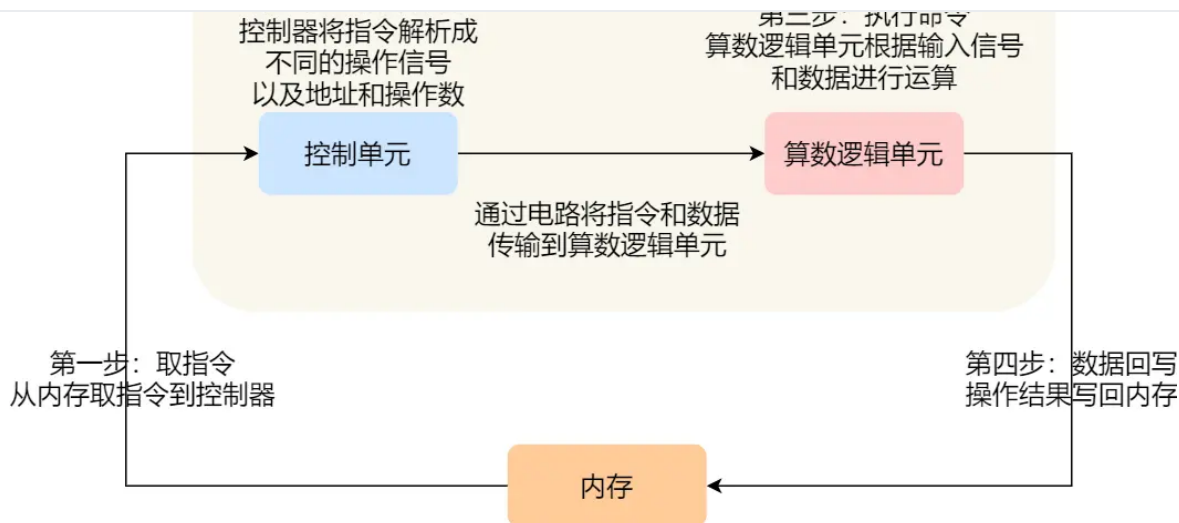
四个阶段的具体含义：

1. CPU 通过程序计数器读取对应内存地址的指令，这个部分称为 **Fetch (取得指令)**；
2. CPU 对指令进行解码，这个部分称为 **Decode (指令译码)**；
3. CPU 执行指令，这个部分称为 **Execution (执行指令)**；
4. CPU 将计算结果存回寄存器或者将寄存器的值存入内存，这个部分称为 **Store (数据回写)**；

上面这 4 个阶段，我们称为**指令周期 (Instruction Cycle)**，CPU 的工作就是一个周期接着一个周期，周而复始。

事实上，不同的阶段其实是由计算机中的不同组件完成的：

[目录](#)[侧边栏](#)[夜间](#)[技术群](#)[资料](#)[支持我](#)[上一篇](#)[下一篇](#)



- 取指令的阶段，我们的指令是存放在**存储器**里的，实际上，通过程序计数器和指令寄存器取出指令的过程，是由**控制器**操作的；
- 指令的译码过程，也是由**控制器**进行的；
- 指令执行的过程，无论是进行算术操作、逻辑操作，还是进行数据传输、条件分支操作，都是由**算术逻辑单元**操作的，也就是由**运算器**处理的。但是如果是一个简单的无条件地址跳转，则是直接在**控制器**里面完成的，不需要用到运算器。

指令的类型

指令从功能角度划分，可以分为 5 大类：

- 数据传输类型的指令**，比如 `store/load` 是寄存器与内存间数据传输的指令，`mov` 是将一个内存地址的数据移动到另一个内存地址的指令；
- 运算类型的指令**，比如加减乘除、位运算、比较大小等等，它们最多只能处理两个寄存器中的数据；
- 跳转类型的指令**，通过修改程序计数器的值来达到跳转执行指令的过程，比如编程中常见的 `if-else`、`switch-case`、函数调用等。
- 信号类型的指令**，比如发生中断的指令 `trap`；
- 闲置类型的指令**，比如指令 `nop`，执行后 CPU 会空转一个周期；

指令的执行速度

CPU 的硬件参数都会有 `GHz` 这个参数，比如一个 `1 GHz` 的 CPU，指的是时钟频率是 `1 G`，代表着 1 秒会产生 `1G` 次数的脉冲信号，每一次脉冲信号高低电平转换就是一个周期，称为时钟周期。

[目录](#)[侧边栏](#)[夜间](#)[技术群](#)[资料](#)[支持我](#)[上一篇](#)[下一篇](#)

一个时钟周期一定能执行完一条指令吗？答案是不一定的，大多数指令不能在一个时钟周期完成，通常需要若干个时钟周期。不同的指令需要的时钟周期是不同的，加法和乘法都对应着一条 CPU 指令，但是乘法需要的时钟周期就要比加法多。

如何让程序跑的更快？

程序执行的时候，耗费的 CPU 时间少就说明程序是快的，对于程序的 CPU 执行时间，我们可以拆解成 **CPU 时钟周期数 (CPU Cycles)** 和 **时钟周期时间 (Clock Cycle Time)** 的乘积。

程序的 CPU 执行时间 = CPU 时钟周期数 x 时钟周期时间

时钟周期时间就是我们前面提及的 CPU 主频，主频越高说明 CPU 的工作速度就越快，比如我手头上的电脑的 CPU 是 2.4 GHz 四核 Intel Core i5，这里的 2.4 GHz 就是电脑的主频，时钟周期时间就是 1/2.4G。

要想 CPU 跑的更快，自然缩短时钟周期时间，也就是提升 CPU 主频，但是今非彼日，摩尔定律早已失效，当今的 CPU 主频已经很难再做到翻倍的效果了。

另外，换一个更好的 CPU，这个也是我们软件工程师控制不了的事情，我们应该把目光放到另外一个乘法因子 —— CPU 时钟周期数，如果能减少程序所需的 CPU 时钟周期数量，一样也是能提升程序的性能的。

对于 CPU 时钟周期数我们可以进一步拆解成：「**指令数 x 每条指令的平均时钟周期数 (Cycles Per Instruction, 简称 CPI)**」，于是程序的 CPU 执行时间的公式可变成如下：

程序的 CPU 执行时间 = 指令数 x CPI x 时钟周期时间

因此，要想程序跑的更快，优化这三者即可：

- **指令数**，表示执行程序所需要多少条指令，以及哪些指令。这个层面是基本靠编译器来优化，毕竟同样的代码，在不同的编译器，编译出来的计算机指令会有各种不同的表示方式。
- **每条指令的平均时钟周期数 CPI**，表示一条指令需要多少个时钟周期数，现代大多数 CPU 通过流水线技术 (Pipeline)，让一条指令需要的 CPU 时钟周期数尽可能的少；

[目录](#)[侧边栏](#)[夜间](#)[技术群](#)[资料](#)[支持我](#)[上一篇](#)[下一篇](#)

[首页](#) [图解网络](#) [图解系统](#) [图解 MySQL](#) [图解 Redis](#) [学习路线](#) [网站动态](#) [Github](#)

的，CPU 跑的越快，散热的压力就会越大，CPU 会很容易奔溃。

很多厂商为了跑分而跑分，基本都是在这三个方面入手的哦，特别是超频这一块。

总结

最后我们再来回答开头的问题。

64 位相比 32 位 CPU 的优势在哪吗？64 位 CPU 的计算性能一定比 32 位 CPU 高很多吗？

64 位相比 32 位 CPU 的优势主要体现在两个方面：

- 64 位 CPU 可以一次计算超过 32 位的数字，而 32 位 CPU 如果要计算超过 32 位的数字，要多步骤进行计算，效率就没那么高，但是大部分应用程序很少会计算那么大的数字，所以**只有运算大数字的时候，64 位 CPU 的优势才能体现出来，否则和 32 位 CPU 的计算性能相差不大。**
- 64 位 CPU 可以**寻址更大的内存空间**，32 位 CPU 最大的寻址地址是 4G，即使你加了 8G 大小的内存，也还是只能寻址到 4G，而 64 位 CPU 最大寻址地址是 2^{64} ，远超于 32 位 CPU 最大寻址地址的 2^{32} 。

你知道软件的 32 位和 64 位之间的区别吗？再来 32 位的操作系统可以运行在 64 位的电脑上吗？64 位的操作系统可以运行在 32 位的电脑上吗？如果不行，原因是什么？

64 位和 32 位软件，实际上代表指令是 64 位还是 32 位的：

- 如果 32 位指令在 64 位机器上执行，需要一套兼容机制，就可以做到兼容运行了。但是**如果 64 位指令在 32 位机器上执行，就比较困难了，因为 32 位的寄存器存不下 64 位的指令；**
- 操作系统其实也是一种程序，我们也会看到操作系统会分成 32 位操作系统、64 位操作系统，其代表意义就是操作系统中程序的指令是多少位，比如 64 位操作系统，指令也就是 64 位，因此不能装在 32 位机器上。

总之，硬件的 64 位和 32 位指的是 CPU 的位宽，软件的 64 位和 32 位指的是指令的位宽。



目录



侧边栏



夜间



技术群



资料



支持我



上一篇



下一篇

关注作者



扫一扫，关注「小林coding」公众号

图解计算机基础
认准**小林coding**

每一张图都包含小林的认真
只为帮助大家能更好的理解

① 关注公众号回复「**图解**」
获取图解系列 PDF

② 关注公众号回复「**加群**」
拉你进百人技术交流群

哈喽，我是小林，就爱图解计算机基础，如果觉得文章对你有帮助，欢迎微信搜索「小林coding」，关注后，回复「网络」再送你图解网络 PDF

上次更新: 9/14/2022, 1:15:16 PM

[← 图解系统介绍](#)

[2.2 磁盘比内存慢几万倍？ →](#)

评论

Powered by [GitHub](#) & [Vssue](#)



目录



侧边栏



夜间



技术群



资料



支持我



上一篇



下一篇

[首页](#) [图解网络](#) [图解系统](#) [图解 MySQL](#) [图解 Redis](#) [学习路线](#) [网站动态](#) [Github](#)



目录



侧边栏



夜间



技术群



资料



支持我



上一篇



下一篇

使用 GitHub 帐号登录后发表评论

使用 GitHub 登录

登录后查看评论