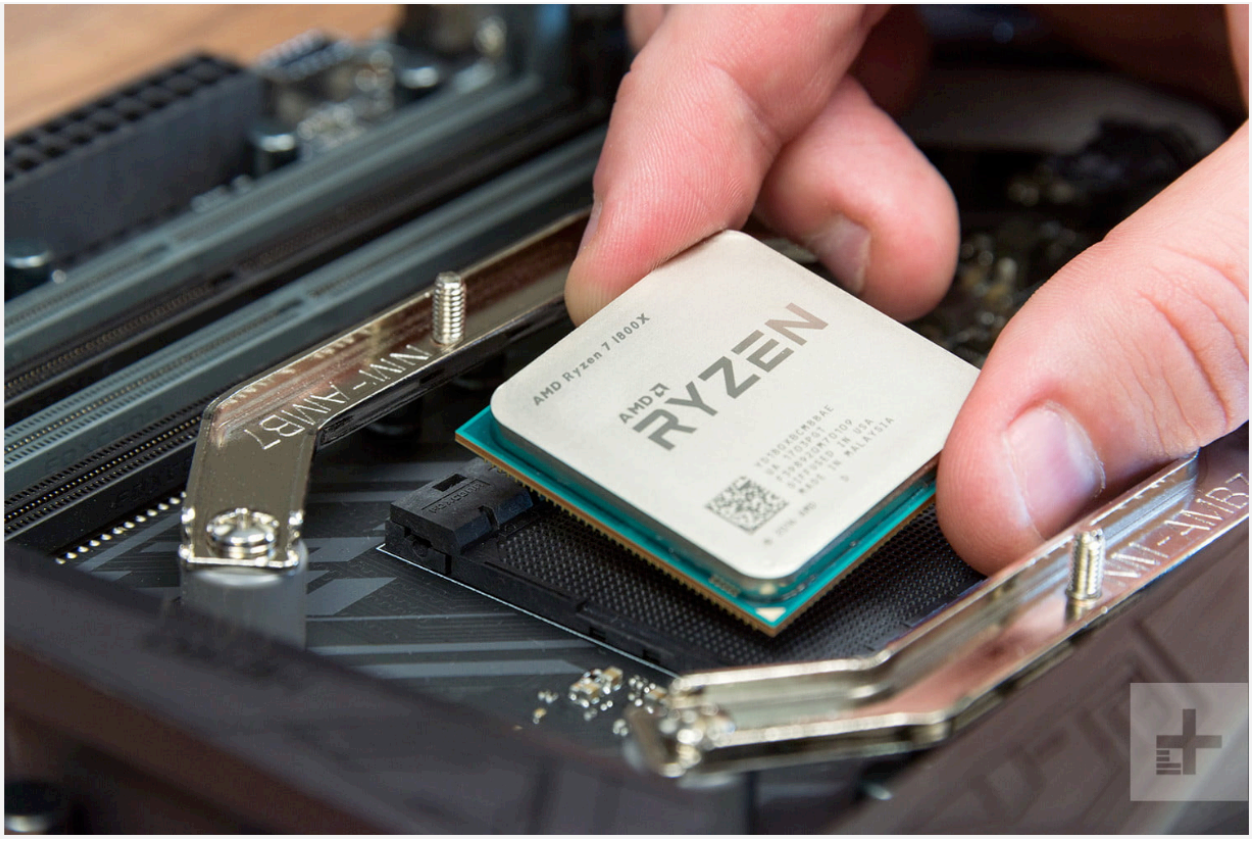
程序员必知的硬核知识大全

- 程序不读入内存就无法运行
- 磁盘构件
  - 磁盘缓存
  - 虚拟内存
    - 虚拟内存与内存的交换方式
- 节约内存
  - 通过 DLL 文件实现函数共有
  - 通过调用 _stdcall 来减少程序文件的大小
  - 磁盘的物理结构
- 操作系统环境
- Windows 操作系统克服了CPU以外的硬件差异
- 不同操作系统的 API 差异性
- FreeBSD Port 帮你轻松使用源代码
- 可以使用虚拟机获取其他环境
  - 提供相同运行环境的 Java 虚拟机
- BIOS 和引导
- 操作系统功能的历史
- 要把操作系统放在第一位
- 系统调用和编程语言的移植性
- 操作系统和高级编程语言使硬件抽象化
- Windows 操作系统的特征
  - 32位操作系统
  - 通过 API 函数集来提供系统调用
  - 提供采用了 GUI 的用户界面
  - 通过 WYSIWYG 实现打印输出
  - 提供多任务功能
  - 提供网络功能和数据库功能
  - 通过即插即用实现设备驱动的自动设定
- 汇编语言和本地代码
- 通过编译器输出汇编语言的源代码
- 不会转换成本地代码的伪指令
- 汇编语言的语法是 操作码 + 操作数
  - 指令解析
  - 函数的调用机制
  - 函数的内部处理
  - 全局变量和局部变量
  - 临时确保局部变量使用的内存空间
  - 循环控制语句的处理
  - 条件分支的处理方法
  - 了解程序运行逻辑的必要性
- 应用和硬件的关系
  - 支持硬件输入输出的 IN 指令和 OUT 指令
  - 测试输入和输出程序
  - 外围设备的中断请求
  - 用中断来实现实时处理
  - 利用 DMA 实现短时间内大量数据传输
  - 文字和图片的显示机制

大家都是程序员，大家都是和计算机打交道的程序员，大家都是和计算机中软件硬件打交道的程序员，大家都是和 `CPU` 打交道的程序员，所以，不管你是玩儿硬件的还是做软件的，你的世界都少不了计算机最核心的 - CPU

CPU是什么

CPU 的全称是 `Central Processing Unit`，它是你的电脑中最 `硬核` 的组件，这种说法一点不为过。CPU 是能够让你的计算机叫 `计算机` 的核心组件，但是它却不能代表你的电脑，CPU 与计算机的关系就相当于大脑和人的关系。它是一种小型的计算机芯片，它嵌入在台式机、笔记本电脑或者平板电脑的主板上。通过在单个计算机芯片上放置数十亿个微型晶体管来构建 CPU。 这些晶体管使它能够执行运行存储在系统内存中的程序所需的计算，也就是说 CPU 决定了你电脑的计算能力。



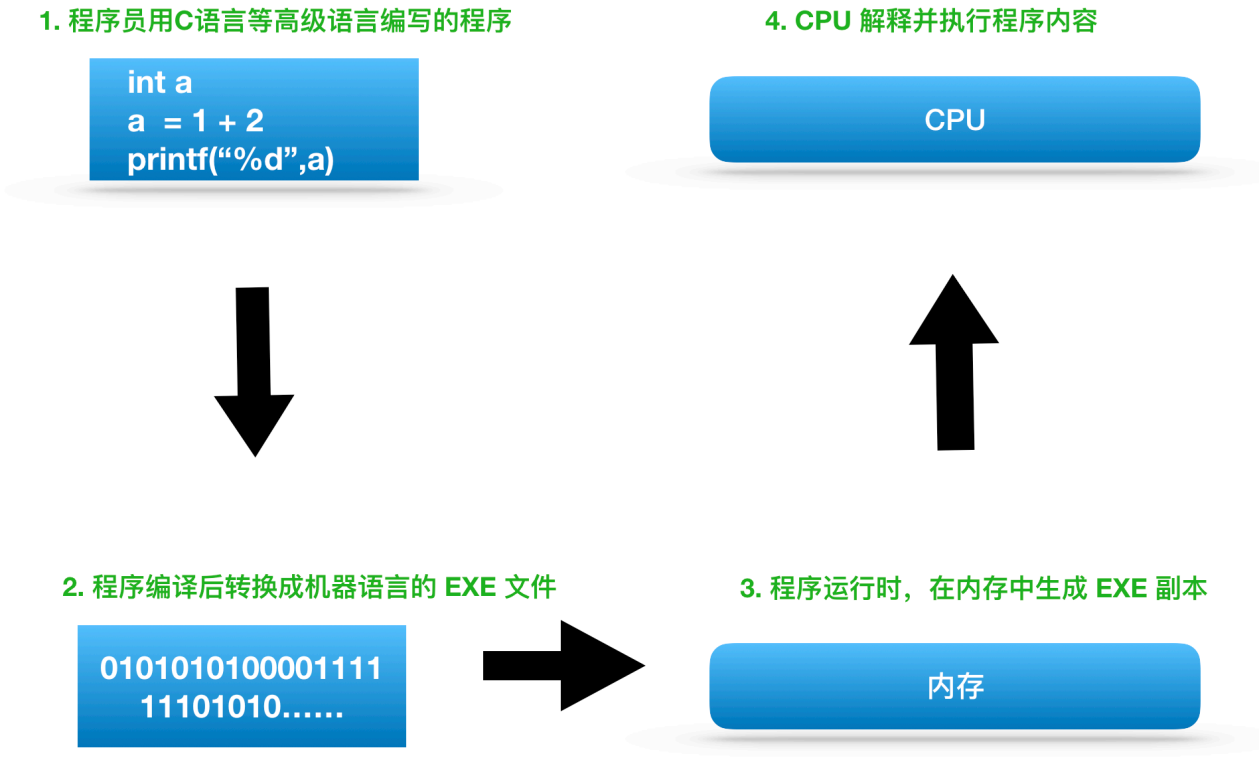
CPU 实际做什么

CPU 的核心是从程序或应用程序获取指令并执行计算。此过程可以分为三个关键阶段：**提取，解码和执行**。CPU从系统的 RAM 中提取指令，然后解码该指令的实际内容，然后再由 CPU 的相关部分执行该指令。

> RAM：随机存取存储器（英语：Random Access Memory，缩写：*RAM*），也叫主存，是与 CPU 直接交换数据的内部存储器。它可以随时读写（刷新时除外），而且速度很快，通常作为操作系统或其他正在运行中的程序的**临时数据存储介质**

CPU 的内部结构

说了这么多 CPU 的重要性，那么 CPU 的内部结构是什么呢？又是由什么组成的呢？下图展示了一般程序的运行流程（以 C 语言为例），可以说了解程序的运行流程是掌握程序运行机制的基础和前提。



程序编译执行的过程

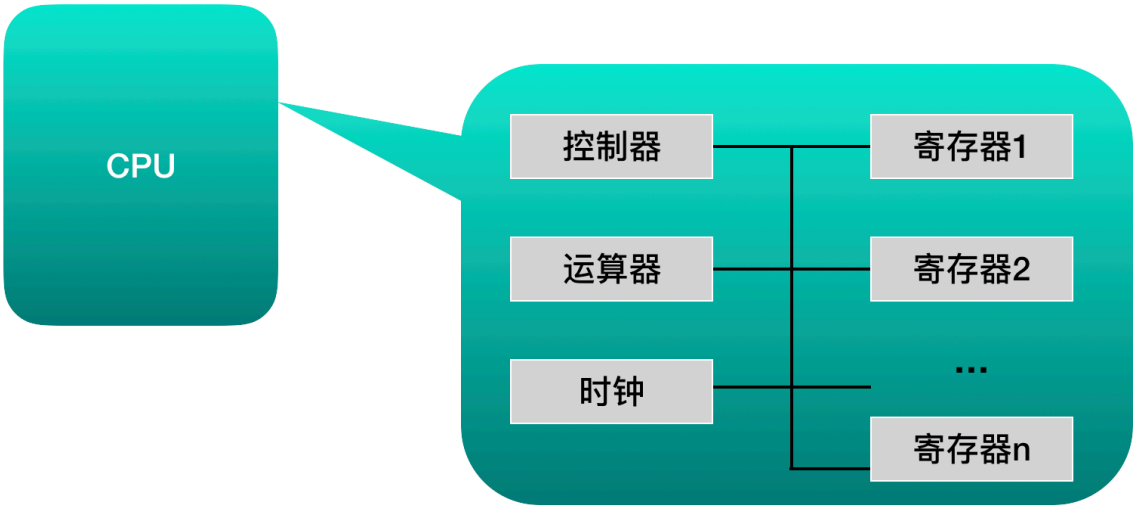
在这个流程中，CPU 负责的就是解释和运行最终转换成机器语言的内容。

CPU 主要由两部分构成：`控制单元` 和 `算术逻辑单元（ALU）`

- 控制单元：从内存中提取指令并解码执行
- 算数逻辑单元（ALU）：处理算数和逻辑运算

CPU 是计算机的心脏和大脑，它和内存都是由许多晶体管组成的电子部件。它接收数据输入，执行指令并处理信息。它与输入/输出（I / O）设备进行通信，这些设备向 CPU 发送数据和从 CPU 接收数据。

从功能来看，CPU 的内部由**寄存器、控制器、运算器和时钟**四部分组成，各部分之间通过电信号连通。



CPU 内部结构图

- 寄存器 是中央处理器内的组成部分。它们可以用来暂存指令、数据和地址。可以将其看作是内存的一种。根据种类的不同，一个 CPU 内部会有 20 - 100个寄存器。
- 控制器 负责把内存上的指令、数据读入寄存器，并根据指令的结果控制计算机
- 运算器 负责运算从内存中读入寄存器的数据
- 时钟 负责发出 CPU 开始计时的时钟信号

接下来简单解释一下内存，为什么说 CPU 需要讲一下内存呢，因为内存是与 CPU 进行沟通的桥梁。计算机所有程序的运行都是在内存中运行的，内存又被称为 主存 ，其作用是存放 CPU 中的运算数据，以及与硬盘等外部存储设备交换的数据。只要计算机在运行中，CPU 就会把需要运算的数据调到主存中进行运算，当运算完成后CPU再将结果传送出来，主存的运行也决定了计算机的稳定运行。

主存通过控制芯片与 CPU 进行相连，由可读写的元素构成，每个字节（1 byte = 8 bits）都带有一个地址编号，**注意是一个字节，而不是一个位**。CPU 通过地址从主存中读取数据和指令，也可以根据地址写入数据。注意一点：当计算机关机时，内存中的指令和数据也会被清除。

CPU 是寄存器的集合体

在 CPU 的四个结构中，我们程序员只需要了解 寄存器 就可以了，其余三个不用过多关注，为什么这么说？因为程序是把寄存器作为对象来描述的。

说到寄存器，就不得不说到汇编语言，我大学是学信息管理与信息系统的，我就没有学过汇编这门课（就算有这门课也不会好好学hhhh），出来混总是要还的，要想作为一个硬核程序员，不能不了解这些概念。说到汇编语言，就不得不说到高级语言，说到高级语言就不得不牵扯出 语言 这个概念。

计算机语言

我们生而为人最明显的一个特征是我们能通过讲话来实现彼此的交流，但是计算机听不懂你说的话，你要想和他交流必须按照计算机指令来交换，这就涉及到语言的问题，计算机是由二进制构成的，它只能听的懂二进制也就是 机器语言 ，但是普通人是无法看懂机器语言的，这个时候就需要一种电脑既能识别，人又能理解的语言，最先出现的就是 汇编语言 。但是汇编语言晦涩难懂，所以又出现了像是 C，C++，Java 的这种高级语言。

所以计算机语言一般分为两种：低级语言（机器语言，汇编语言）和高级语言。使用高级语言编写的程序，经过编译转换成机器语言后才能运行，而汇编语言经过汇编器才能转换为机器语言。

汇编语言

首先来看一段用汇编语言表示的代码清单

```
1   mov eax, dword ptr [ebp-8]   /* 把数值从内存复制到 eax */
2   add eax, dword ptr [ebp-0Ch] /* 把 eax 的数值和内存的数值相加 */
3   mov dword ptr [ebp-4], eax /* 把 eax 的数值（上一步的结果）存储在内存中*/
```

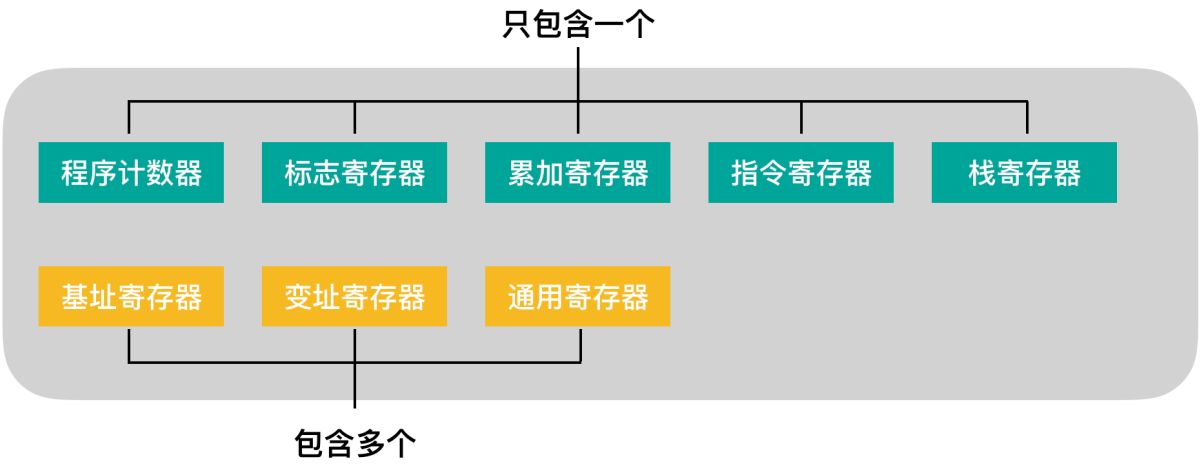
这是采用汇编语言（assembly）编写程序的一部分。汇编语言采用 `助记符(memonic)` 来编写程序，每一个原本是电信号的机器语言指令会有一个与其对应的助记符，例如 `mov,add` 分别是数据的存储（move）和相加（addition）的简写。汇编语言和机器语言是一一对应的。这一点和高级语言有很大的不同，通常我们将汇编语言编写的程序转换为机器语言的过程称为 `汇编` ；反之，机器语言转化为汇编语言的过程称为 `反汇编` 。

汇编语言能够帮助你理解计算机做了什么工作，机器语言级别的程序是通过 `寄存器` 来处理的，上面代码中的 `eax,ebp` 都是表示的寄存器，是 CPU 内部寄存器的名称，所以可以说 **CPU 是一系列寄存器的集合体**。在内存中的存储通过地址编号来表示，而寄存器的种类则通过名字来区分。

不同类型的 CPU ，其内部寄存器的种类，数量以及寄存器存储的数值范围都是不同的。不过，根据功能的不同，可以将寄存器划分为下面这几类

种类	功能
累加寄存器	存储运行的数据和运算后的数据。
标志寄存器	用于反应处理器的状态和运算结果的某些特征以及控制指令的执行。
程序计数器	程序计数器是用于存放下一条指令所在单元的地址的地方。
基址寄存器	存储数据内存的起始位置
变址寄存器	存储基址寄存器的相对地址
通用寄存器	存储任意数据
指令寄存器	储存正在被运行的指令，CPU内部使用，程序员无法对该寄存器进行读写
栈寄存器	存储栈区域的起始位置

其中**程序计数器、累加寄存器、标志寄存器、指令寄存器和栈寄存器**都只有一个，其他寄存器一般有多个。



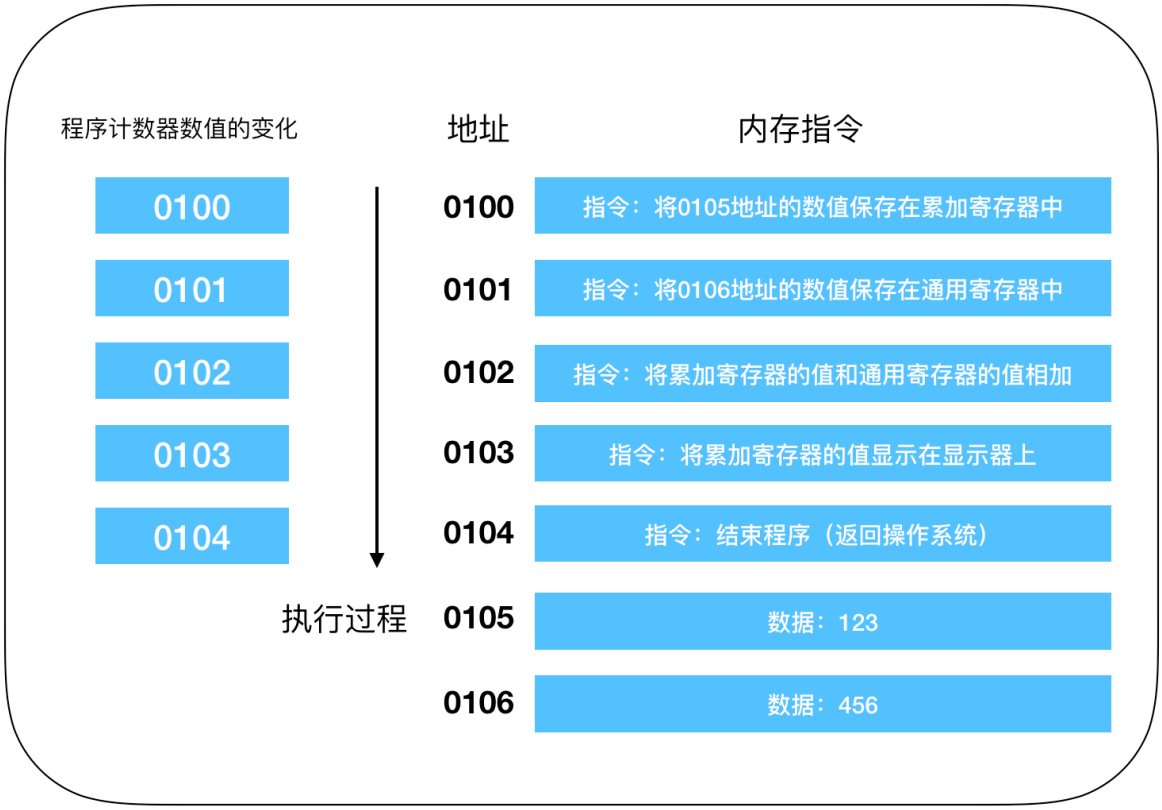
程序员眼中的CPU

程序计数器

`程序计数器(Program Counter)` 是用来存储下一条指令所在单元的地址。

程序执行时，PC的初值为程序第一条指令的地址，在顺序执行程序时， `控制器` 首先按程序计数器所指出的指令地址从内存中取出一条指令，然后分析和执行该指令，同时将PC的值加1指向下一条要执行的指令。

我们还是以一个事例为准来详细的看一下程序计数器的执行过程



内存中配置程序示例

这是一段进行相加的操作，程序启动，在经过编译解析后会由操作系统把硬盘中的程序复制到内存中，示例中的程序是将 123 和 456 执行相加操作，并将结果输出到显示器上。由于使用机器语言难以描述，所以这是经过翻译后的结果，实际上每个指令和数据都可能分布在不同的地址上，但为了方便说明，把组成一条指令的内存和数据放在了一个内存地址上。

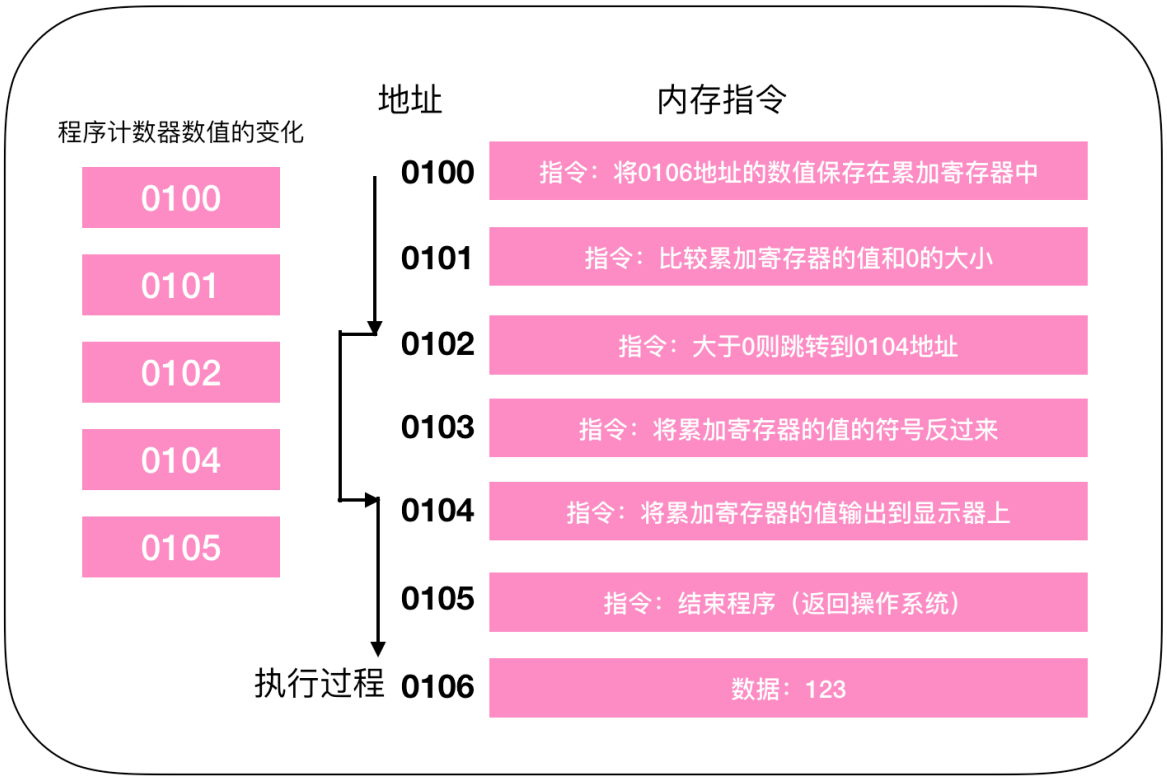
地址 `0100` 是程序运行的起始位置。Windows 等操作系统把程序从硬盘复制到内存后，会将程序计数器作为设定为起始位置 0100，然后执行程序，每执行一条指令后，程序计数器的数值会增加1（或者直接指向下一条指令的地址），然后，CPU 就会根据程序计数器的数值，从内存中读取命令并执行，也就是说，**程序计数器控制着程序的流程**。

条件分支和循环机制

我们都学过高级语言，高级语言中的条件控制流程主要分为三种：`顺序执行、条件分支、循环判断` 三种，顺序执行是按照地址的内容顺序的执行指令。条件分支是根据条件执行任意地址的指令。循环是重复执行同一地址的指令。

- 顺序执行的情况比较简单，每执行一条指令程序计数器的值就是 + 1。
- 条件和循环分支会使程序计数器的值指向任意的地址，这样一来，程序便可以返回到上一个地址来重复执行同一个指令，或者跳转到任意指令。

下面以条件分支为例来说明程序的执行过程（循环也很相似）



条件循环的执行流程

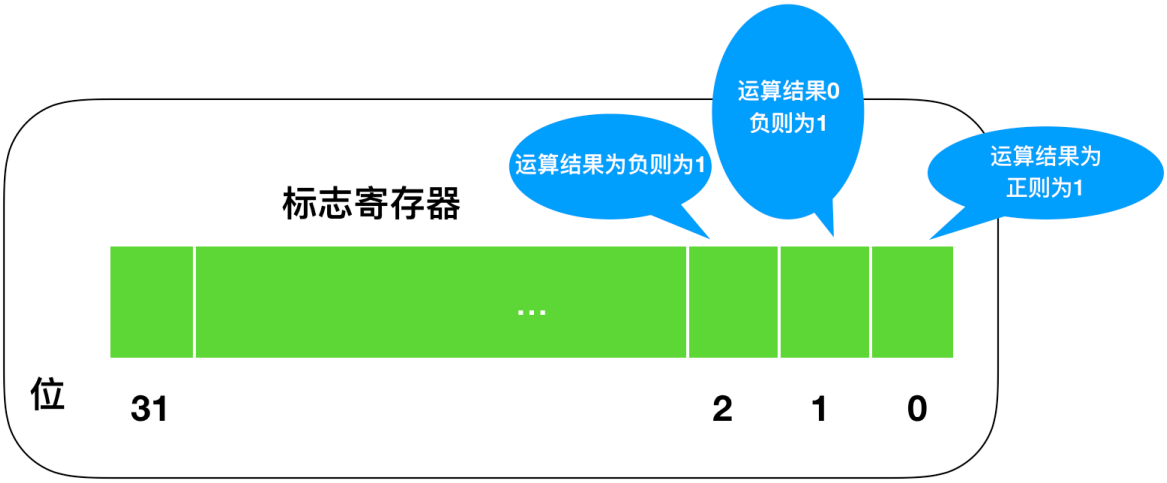
程序的开始过程和顺序流程是一样的，CPU 从0100处开始执行命令，在0100和0101都是顺序执行，PC 的值顺序+1，执行到0102地址的指令时，判断0106寄存器的数值大于0，跳转（jump）到0104地址的指令，将数值输出到显示器中，然后结束程序，0103 的指令被跳过了，这就和我们程序中的 `if()` 判断是一样的，在不满足条件的情况下，指令会直接跳过。所以 PC 的执行过程也就没有直接+1，而是下一条指令的地址。

标志寄存器

条件和循环分支会使用到 `jump（跳转指令）`，会根据当前的指令来判断是否跳转，上面我们提到了 `标志寄存器`，无论当前累加寄存器的运算结果是正数、负数还是零，标志寄存器都会将其保存（也负责溢出和奇偶校验）

> 溢出（overflow）：是指运算的结果超过了寄存器的长度范围
>
> 奇偶校验（parity check）：是指检查运算结果的值是偶数还是奇数

CPU 在进行运算时，标志寄存器的数值会根据当前运算的结果自动设定，运算结果的正、负和零三种状态由标志寄存器的三个位表示。标志寄存器的第一个字节位、第二个字节位、第三个字节位各自的结果都为1时，分别代表着正数、零和负数。

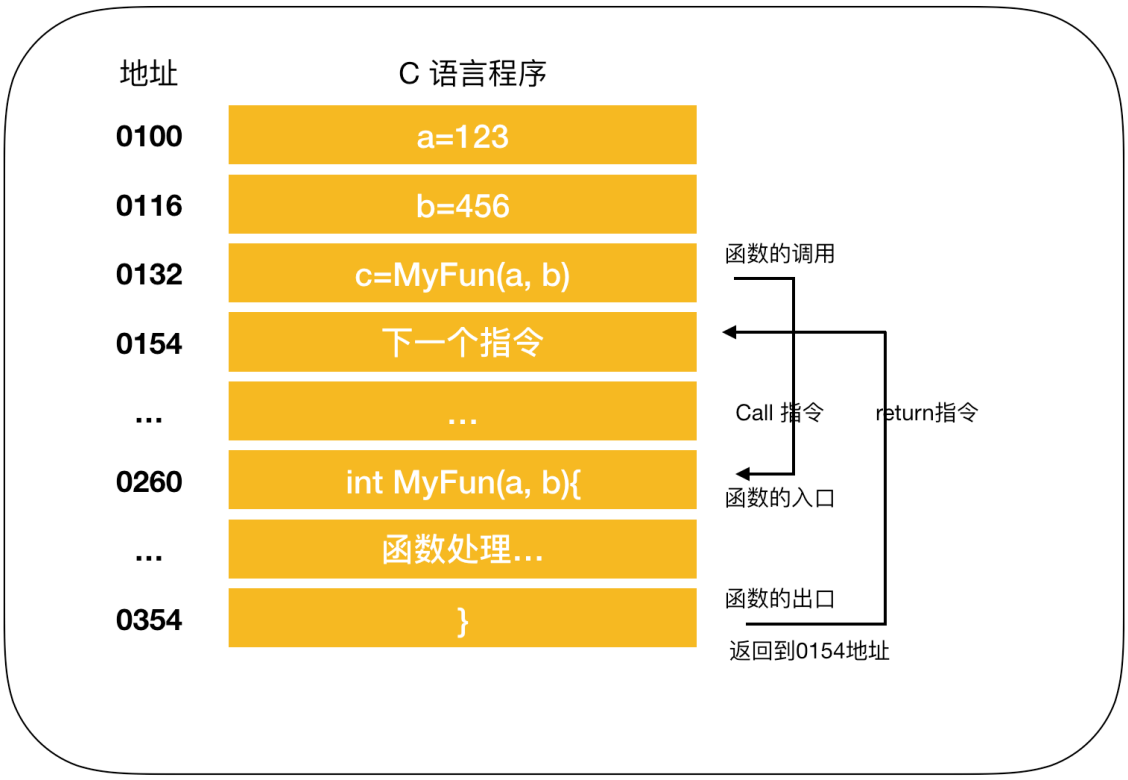


比较运算的标志寄存器示意图

CPU 的执行机制比较有意思，假设累加寄存器中存储的 XXX 和通用寄存器中存储的 YYY 做比较，执行比较的背后，CPU 的运算机制就会做减法运算。而无论减法运算的结果是正数、零还是负数，都会保存到标志寄存器中。结果为正表示 XXX 比 YYY 大，结果为零表示 XXX 和 YYY 相等，结果为负表示 XXX 比 YYY 小。程序比较的指令，实际上是在 CPU 内部做 `减法` 运算。

函数调用机制

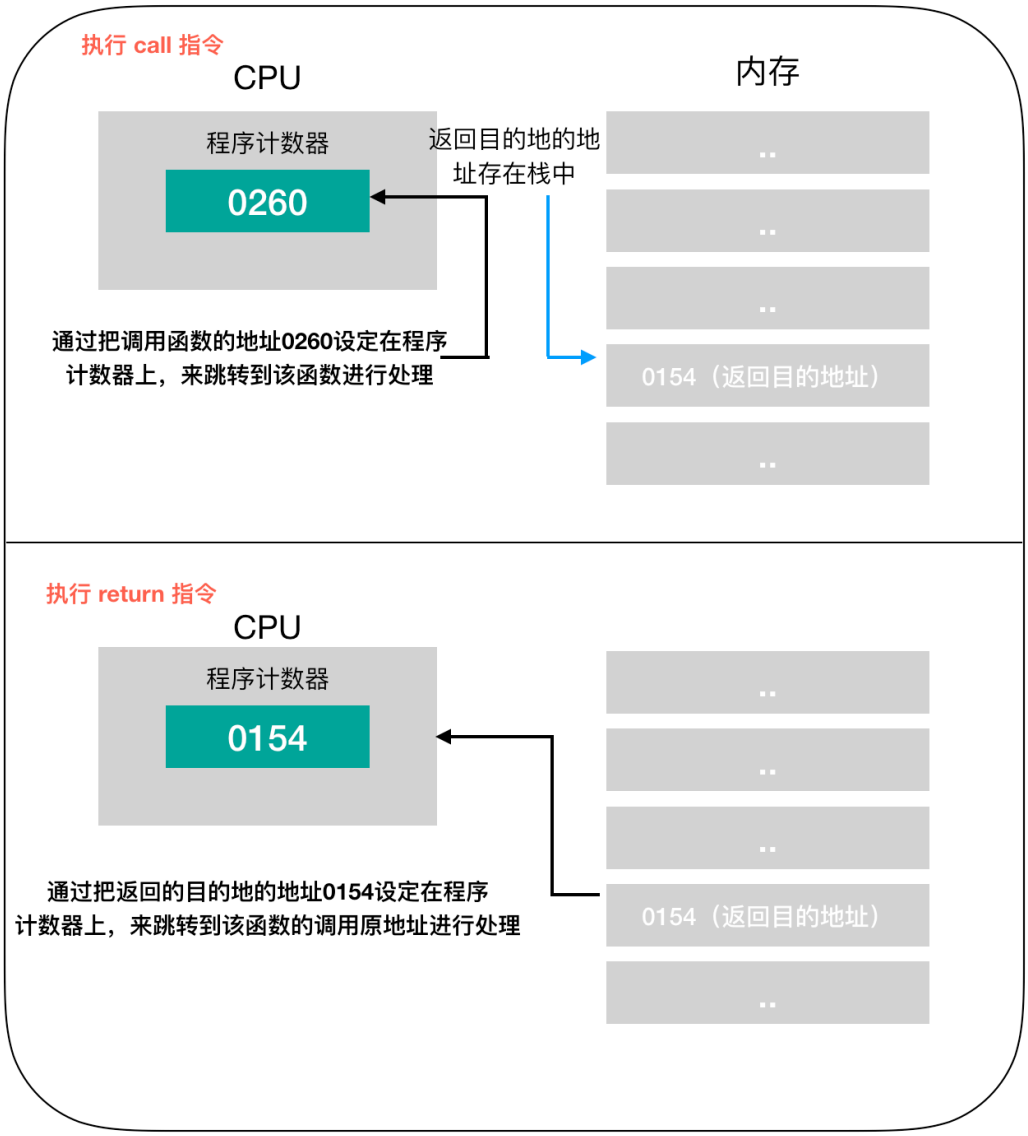
接下来，我们继续介绍函数调用机制，哪怕是高级语言编写的程序，函数调用处理也是通过把程序计数器的值设定成函数的存储地址来实现的。函数执行跳转指令后，必须进行返回处理，单纯的指令跳转没有意义，下面是一个实现函数跳转的例子



程序调用示意图

图中将变量 a 和 b 分别赋值为 123 和 456 ，调用 MyFun(a,b) 方法，进行指令跳转。图中的地址是将 C 语言编译成机器语言后运行时的地址，由于1行 C 程序在编译后通常会变为多行机器语言，所以图中的地址是分散的。在执行完 MyFun(a,b)指令后，程序会返回到 MyFun(a,b) 的下一条指令，CPU 继续执行下面的指令。

函数的调用和返回很重要的两个指令是 `call` 和 `return` 指令，再将函数的入口地址设定到程序计数器之前，call 指令会把调用函数后要执行的指令地址存储在名为栈的主存内。函数处理完毕后，再通过函数的出口来执行 return 指令。return 指令的功能是把保存在栈中的地址设定到程序计数器。MyFun 函数在被调用之前，0154 地址保存在栈中，MyFun 函数处理完成后，会把0154的地址保存在程序计数器中。这个调用过程如下



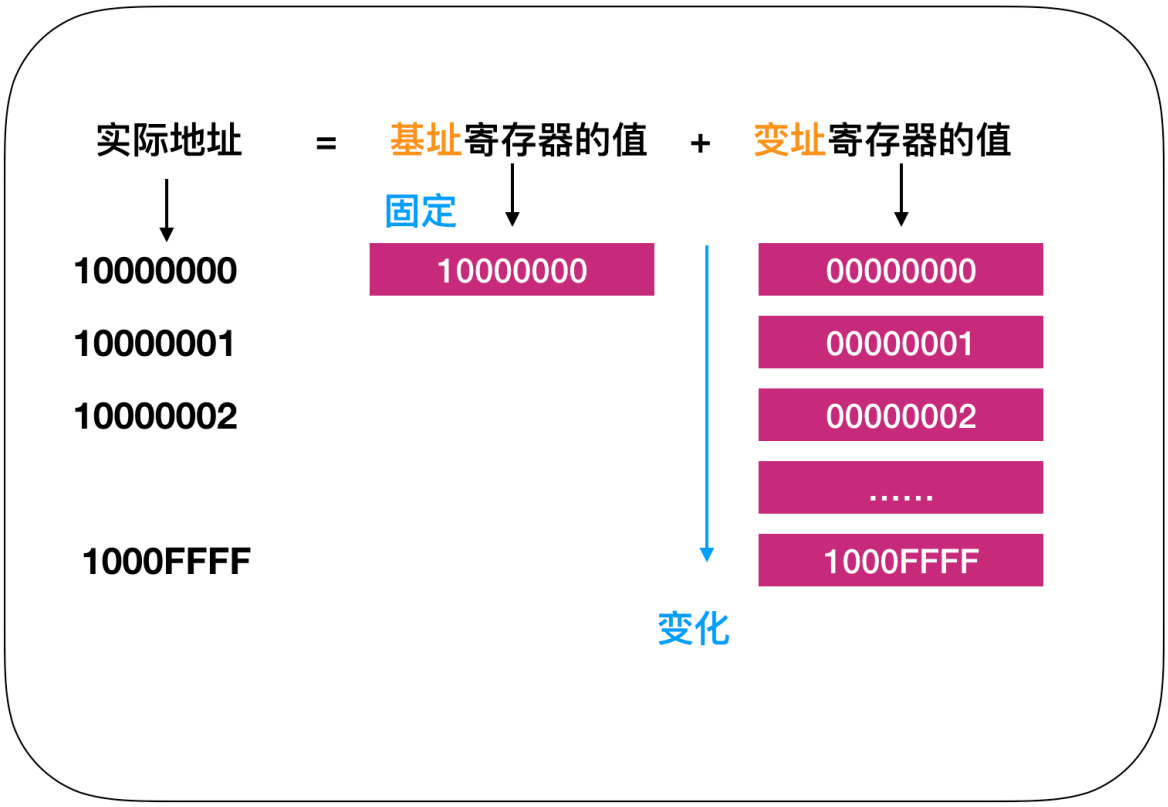
函数调用中程序计数器和栈的职能

在一些高级语言的条件或者循环语句中，函数调用的处理会转换成 call 指令，函数结束后的处理则会转换成 return 指令。

通过地址和索引实现数组

接下来我们看一下基址寄存器和变址寄存器，通过这两个寄存器，我们可以对主存上的特定区域进行划分，来实现类似数组的操作，首先，我们用十六进制数将计算机内存上的 00000000 - FFFFFFFF 的地址划分出来。那么，凡是该范围的内存地址，只要有一个 32 位的寄存器，便可查看全部地址。但如果想要想数组那样分割特定的内存区域以达到连续查看的目的的话，使用两个寄存器会更加方便。

例如，我们用两个寄存器（基址寄存器和变址寄存器）来表示内存的值



基址寄存器和变址寄存器

这种表示方式很类似数组的构造， `数组` 是指同样长度的数据在内存中进行连续排列的数据构造。用数组名表示数组全部的值，通过索引来区分数组的各个数据元素，例如: a[0] - a[4]， `[]` 内的 0 - 4 就是数组的下标。

CPU 指令执行过程

那么 CPU 是如何执行一条条的指令的呢?

几乎所有的冯·诺伊曼型计算机的CPU，其工作都可以分为5个阶段：**取指令、指令译码、执行指令、访存取数、结果写回**。

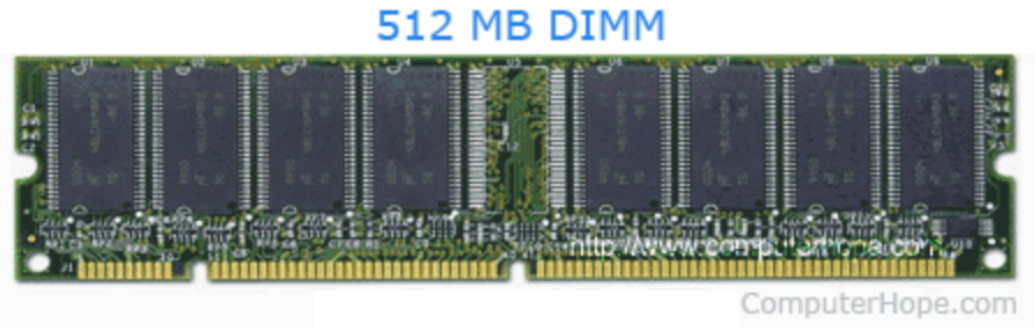
- `取指令` 阶段是将内存中的指令读取到 CPU 中寄存器的过程，程序寄存器用于存储下一条指令所在的地址
- `指令译码` 阶段，在取指令完成后，立马进入指令译码阶段，在指令译码阶段，指令译码器按照预定的指令格式，对取回的指令进行拆分和解释，识别区分出不同的指令类别以及各种获取操作数的方法。
- `执行指令` 阶段，译码完成后，就需要执行这一条指令了，此阶段的任务是完成指令所规定的各种操作，具体实现指令的功能。
- `访问取数` 阶段，根据指令的需要，有可能需要从内存中提取数据，此阶段的任务是：根据指令地址码，得到操作数在主存中的地址，并从主存中读取该操作数用于运算。
- `结果写回` 阶段，作为最后一个阶段，结果写回（Write Back，WB）阶段把执行指令阶段的运行结果数据“写回”到某种存储形式：结果数据经常被写到CPU的内部寄存器中，以便被后续的指令快速地存取；

什么是内存

内存（Memory）是计算机中最重要的部件之一，它是程序与CPU进行沟通的桥梁。计算机中所有程序的运行都是在内存中进行的，因此内存对计算机的影响非常大，内存又被称为 主存 ，其作用是存放 CPU 中的运算数据，以及与硬盘等外部存储设备交换的数据。只要计算机在运行中，CPU 就会把需要运算的数据调到主存中进行运算，当运算完成后CPU再将结果传送出来，主存的运行也决定了计算机的稳定运行。

内存的物理结构

在了解一个事物之前，你首先得先需要 见 过它，你才会有印象，才会有想要了解的兴趣，所以我们首先需要先看一下什么是内存以及它的物理结构是怎样的。

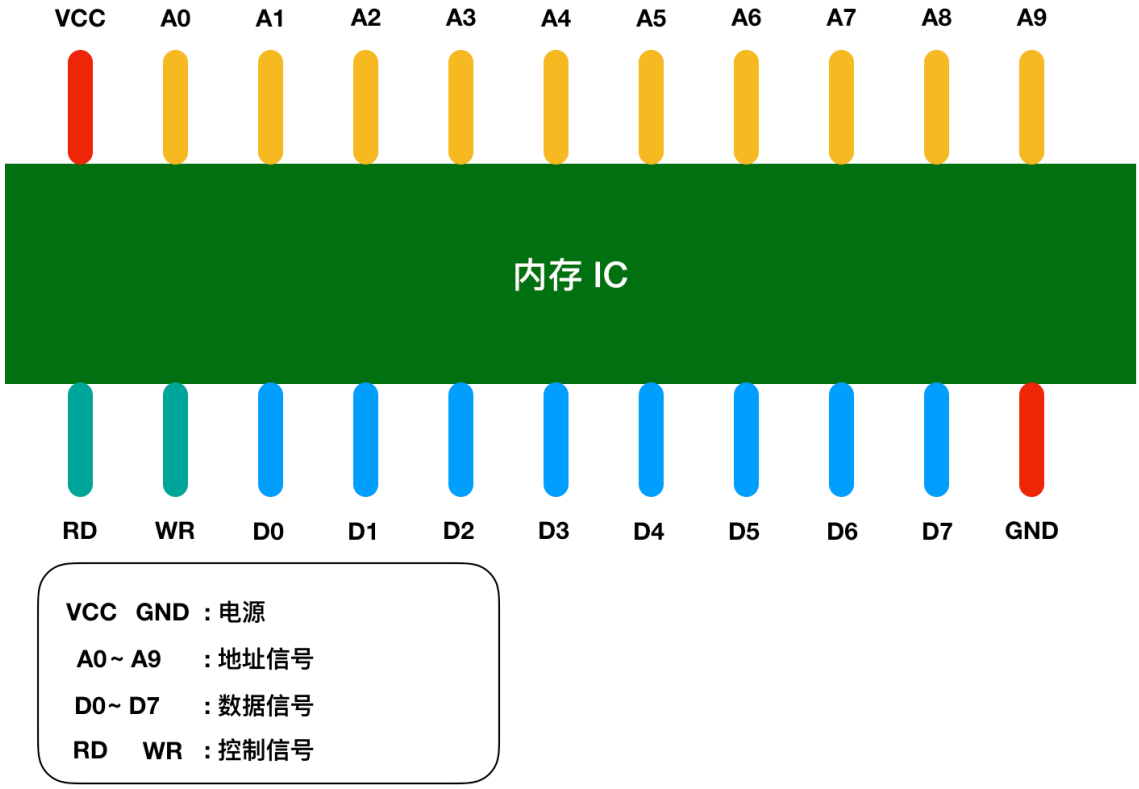


512M内存的物理结构

内存的内部是由各种IC电路组成的，它的种类很庞大，但是其主要分为三种存储器

- 随机存储器（RAM）： 内存中最重要的一种，表示既可以从中读取数据，也可以写入数据。当机器关闭时，内存中的信息会 丢失 。
- 只读存储器（ROM）：ROM 一般只能用于数据的读取，不能写入数据，但是当机器停电时，这些数据不会丢失。
- 高速缓存（Cache）：Cache 也是我们经常见到的，它分为一级缓存（L1 Cache）、二级缓存（L2 Cache）、三级缓存（L3 Cache）这些数据，它位于内存和 CPU 之间，是一个读写速度比内存 更快 的存储器。当 CPU 向内存写入数据时，这些数据也会被写入高速缓存中。当 CPU 需要读取数据时，会直接从高速缓存中直接读取，当然，如需要的数据在Cache中没有，CPU会再去读取内存中的数据。

内存 IC 是一个完整的结构，它内部也有电源、地址信号、数据信号、控制信号和用于寻址的 IC 引脚来进行数据的读写。下面是一个虚拟的 IC 引脚示意图



内存 IC 引脚配置

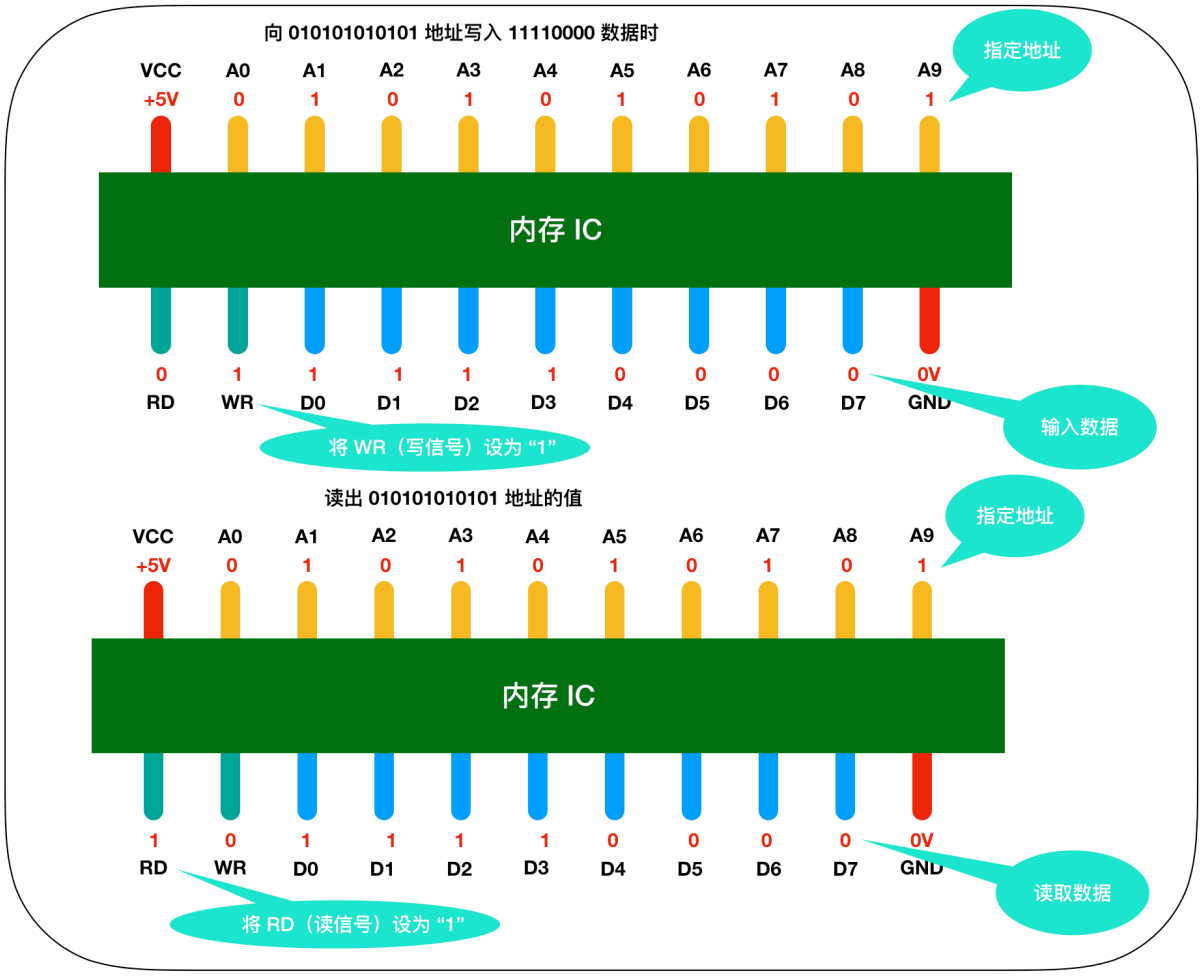
图中 VCC 和 GND 表示电源，A0 - A9 是地址信号的引脚，D0 - D7 表示的是控制信号、RD 和 WR 都是好控制信号，我用不同的颜色进行了区分，将电源连接到 VCC 和 GND 后，就可以对其他引脚传递 0 和 1 的信号，大多数情况下，**+5V 表示1，0V 表示 0**。

我们都知道内存是用来存储数据，那么这个内存 IC 中能存储多少数据呢？D0 - D7 表示的是数据信号，也就是说，一次可以输入输出 8 bit = 1 byte 的数据。A0 - A9 是地址信号共十个，表示可以指定 00000 00000 - 11111 11111 共 2 的 10次方 = `1024个地址` 。每个地址都会存放 1 byte 的数据，因此我们可以得出内存 IC 的容量就是 1 KB。

如果我们使用的是 512 MB 的内存，这就相当于是 512000（512 * 1000） 个内存 IC。当然，一台计算机不太可能有这么多个内存 IC ，然而，通常情况下，一个内存 IC 会有更多的引脚，也就能存储更多数据。

内存的读写过程

让我们把关注点放在内存 IC 对数据的读写过程上来吧！我们来看一个对内存IC 进行数据写入和读取的模型



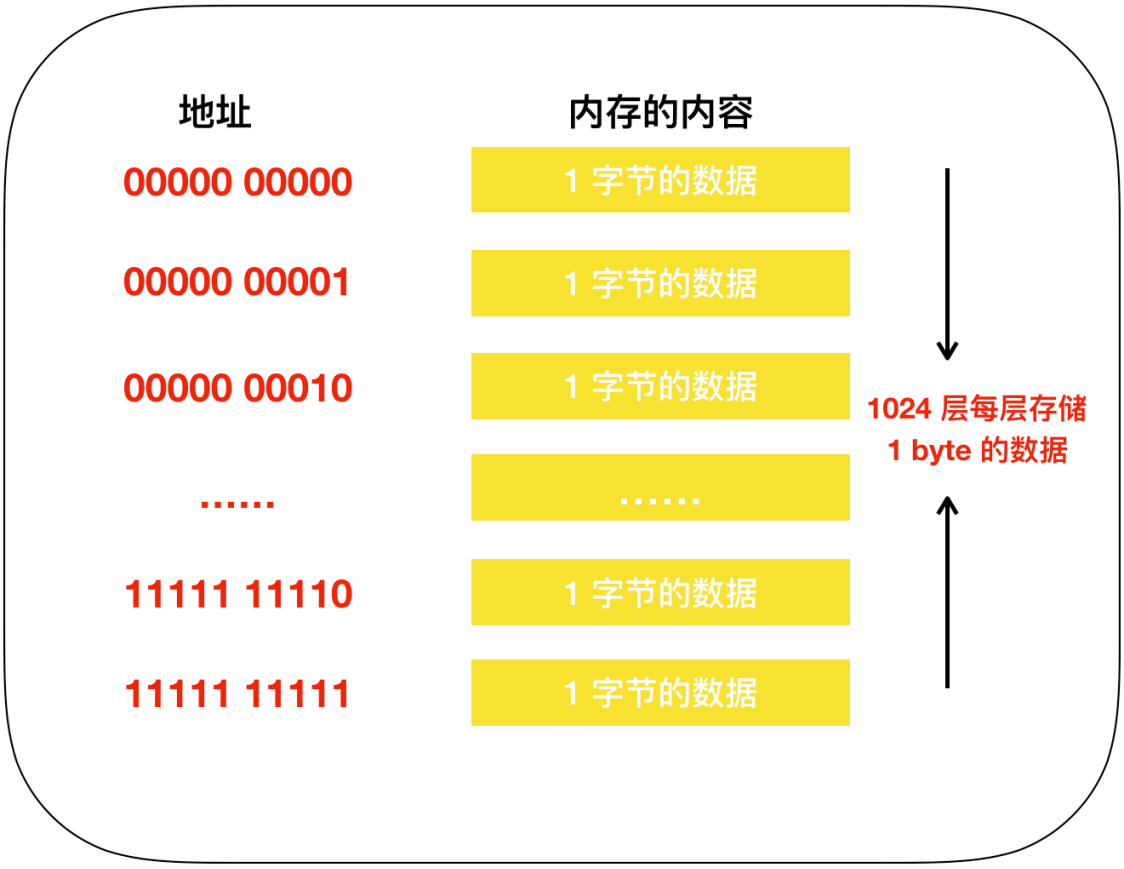
从内存 IC 读取和写入数据

来详细描述一下这个过程，假设我们要向内存 IC 中写入 1byte 的数据的话，它的过程是这样的：

- 首先给 VCC 接通 +5V 的电源，给 GND 接通 0V 的电源，使用 `A0 - A9` 来指定数据的存储场所，然后再把数据的值输入给 `D0 - D7` 的数据信号，并把 `WR (write)` 的值置为 1，执行完这些操作后，即可以向内存 IC 写入数据
- 读出数据时，只需要通过 A0 - A9 的地址信号指定数据的存储场所，然后再将 RD 的值置为 1 即可。
- 图中的 RD 和 WR 又被称为控制信号。其中当WR 和 RD 都为 0 时，无法进行写入和读取操作。

内存的现实模型

为了便于记忆，我们把内存模型映射成为我们现实世界的模型，在现实世界中，内存的模型很想我们生活的楼房。在这个楼房中，1层可以存储一个字节的数据，楼层号就是 `地址` ，下面是内存和楼层整合的模型图



1KB 内存模型图

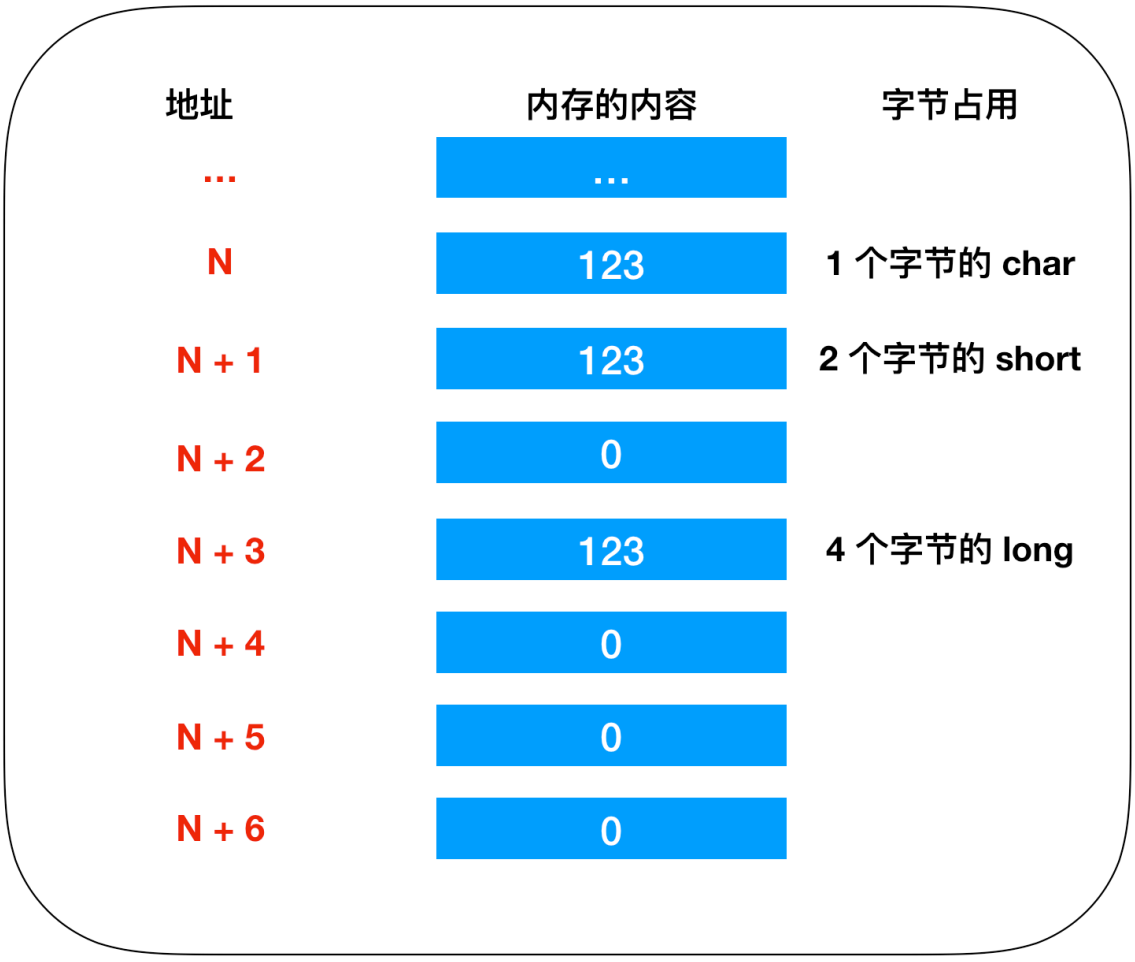
我们知道，程序中的数据不仅只有数值，还有 数据类型 的概念，从内存上来看，就是占用内存大小（占用楼层数）的意思。即使物理上强制以 1 个字节为单位来逐一读写数据的内存，在程序中，通过指定其数据类型，也能实现以特定字节数为单位来进行读写。

下面是一个以特定字节数为例来读写指令字节的程序的示例

```
// 定义变量
char a;
short b;
long c;

// 变量赋值
a = 123;
b = 123;
c = 123;
```

我们分别声明了三个变量 a,b,c ，并给每个变量赋上了相同的 123，这三个变量表示内存的特定区域。通过变量，即使不指定物理地址，也可以直接完成读写操作，操作系统会自动为变量分配内存地址。

这三个变量分别表示 1 个字节长度的 char，2 个字节长度的 short，表示4 个字节的 long。因此，虽然数据都表示的是 123，但是其存储时所占的内存大小是不一样的。如下所示



变量存储示意图

这里的 123 都没有超过每个类型的最大长度，所以 short 和 long 类型为所占用的其他内存空间分配的数值是0，这里我们采用的是 低字节序列 的方式存储

> 低字节序列：将数据低位存储在内存低位地址。
>
> 高字节序列：将数据的高位存储在内存地位的方式称为高字节序列。

内存的使用

指针

指针是 C 语言非常重要的特征，指针也是一种变量，只不过它所表示的不是数据的值，而是内存的地址。通过使用指针，可以对任意内存地址的数据进行读写。

在了解指针读写的过程前，我们先需要了解如何定义一个指针，和普通的变量不同，在定义指针时，我们通常会在变量名前加一个 `*` 号。例如我们可以用指针定义如下的变量

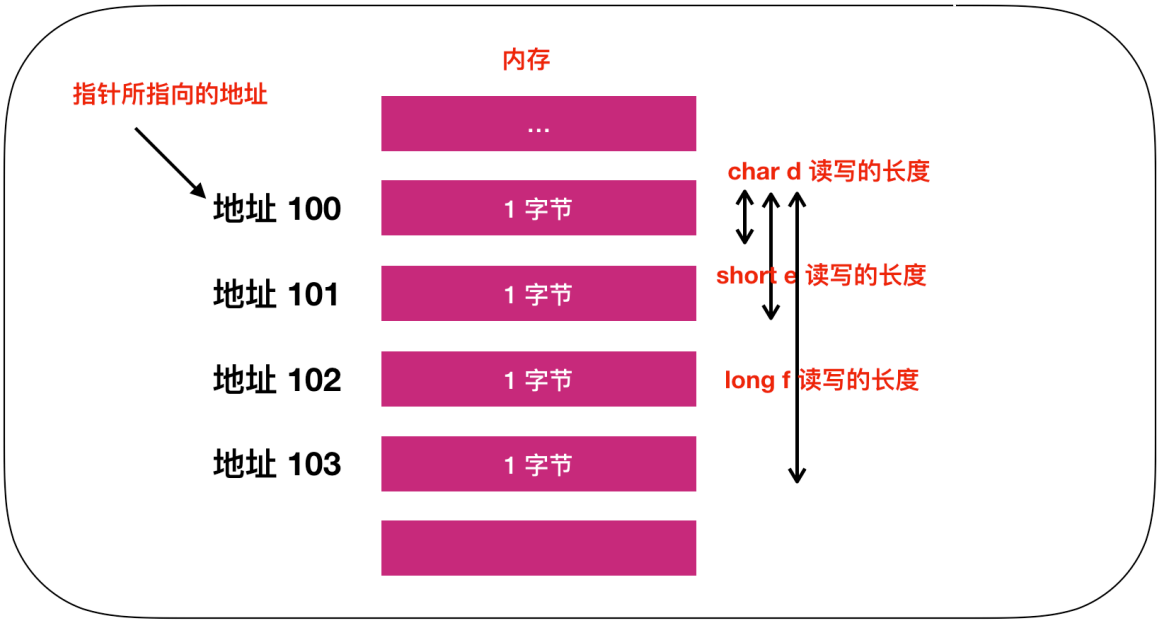
```
1   char *d; // char类型的指针 d 定义
2   short *e; // short类型的指针 e 定义
3   long *f; // long类型的指针 f 定义
```

我们以 32 位计算机为例，32位计算机的内存地址是 4 字节，在这种情况下，指针的长度也是 32 位。然而，变量 d e f 却代表了不同的字节长度，这是为什么呢?

实际上，这些数据表示的是从内存中一次读取的字节数，比如 d e f 的值都为 100，那么使用 char 类型时就能够从内存中读写 1 byte 的数据，使用 short 类型就能够从内存读写 2 字节的数据， 使用 long 就能够读写 4 字节的数据，下面是一个完整的类型字节表

类型	32位	64位
char	1	1
short int	2	2
int	4	4
unsigned int	4	4
float	4	4
double	8	8
long	4	8
long long	8	8
unsigned long	4	8

我们可以用图来描述一下这个读写过程



数据类型读写长度示意图

数组是内存的实现

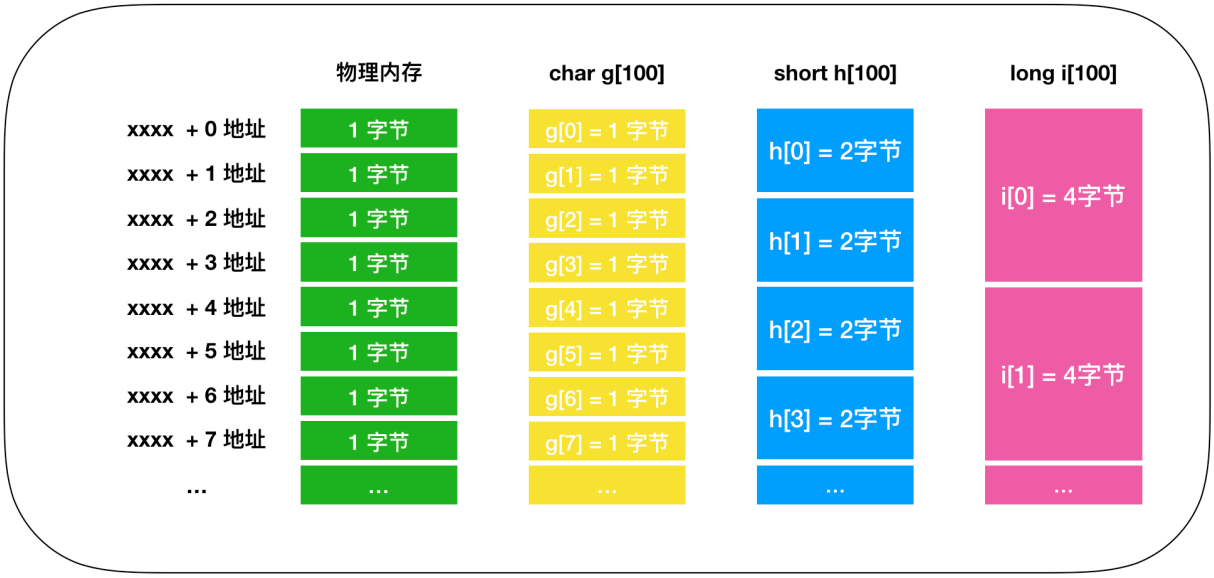
数组是指多个 相同 的数据类型在内存中连续排列的一种形式。作为数组元素的各个数据会通过 下标编号 来区分，这个编号也叫做 索引 ，如此一来，就可以对指定索引的元素进行读写操作。

首先先来认识一下数组，我们还是用 char、short、long 三种元素来定义数组，数组的元素用 `[value]` 扩起来，里面的值代表的是数组的长度，就像下面的定义

```
1   char g[100];
2   short h[100];
3   long i[100];
```

数组定义的数据类型，也表示一次能够读写的内存大小，char 、short 、long 分别以 1 、2 、4 个字节为例进行内存的读写。

数组是内存的实现，数组和内存的物理结构完全一致，尤其是在读写1个字节的时候，当字节数超过 1 时，只能通过逐个字节来读取，下面是内存的读写过程



不同数据类型的数组

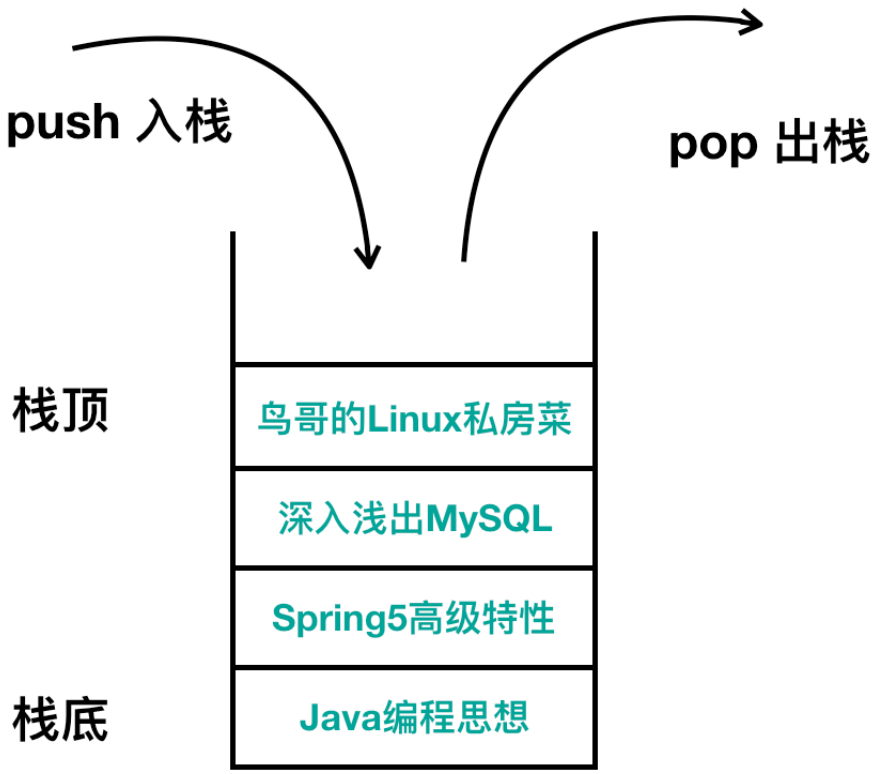
栈和队列

我们上面提到数组是内存的一种实现，使用数组能够使编程更加高效，下面我们就来认识一下其他数据结构，通过这些数据结构也可以操作内存的读写。

栈

栈（stack）是一种很重要的数据结构，栈采用 LIFO（Last In First Out）即 后入先出 的方式对内存进行操作。它就像一个大的收纳箱，你可以往里面放相同类型的东西，比如书，最先放进收纳箱的书在最下面，最后放进收纳箱的书在最上面，如果你想拿书的话， 必须从最上面开始取，否则是无法取出最下面的书籍的。

栈的数据结构就是这样，你把书籍压入收纳箱的操作叫做 压入（push） ，你把书籍从收纳箱取出的操作叫做 弹出（pop） ，它的模型图大概是这样

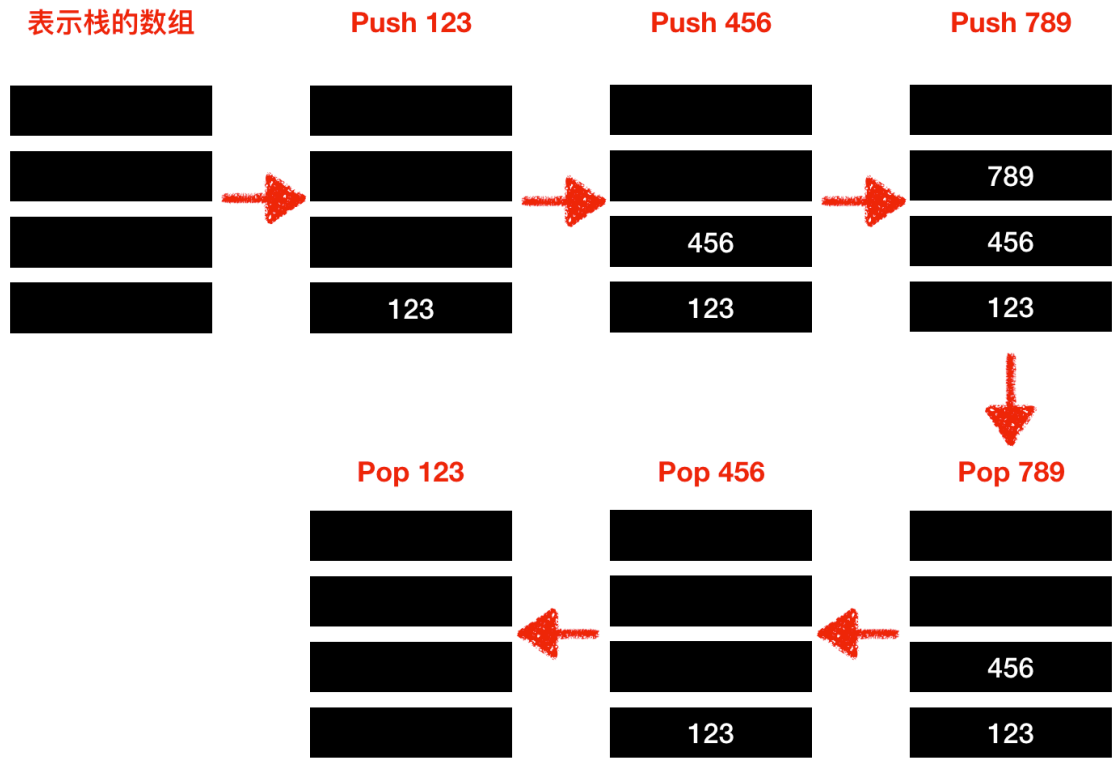


栈的数据结构

入栈相当于是增加操作，出栈相当于是删除操作，只不过叫法不一样。栈和内存不同，它不需要指定元素的地址。它的大概使用如下

```
// 压入数据
Push(123);
Push(456);
Push(789);

// 弹出数据
j = Pop();
k = Pop();
l = Pop();
```

在栈中，LIFO 方式表示栈的数组中所保存的最后面的数据（Last In）会被最先读取出来（First On）。



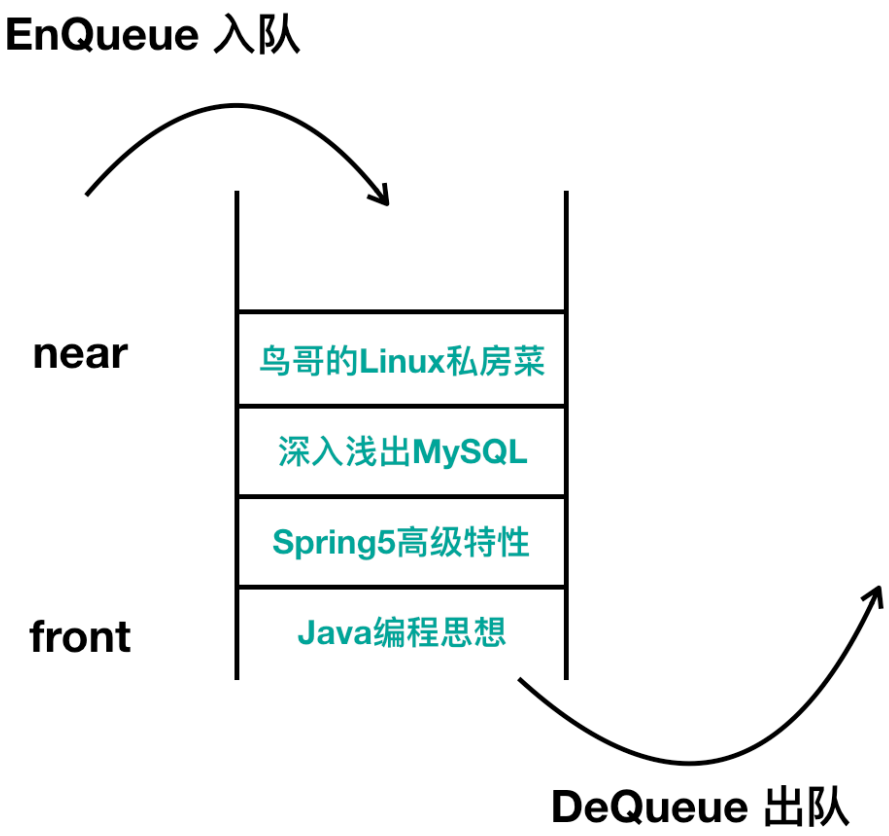
运行时栈的变化

队列

队列 和栈很相似但又不同，相同之处在于队列也不需要指定元素的地址，不同之处在于队列是一种 先入先出(First In First Out) 的数据结构。队列在我们生活中的使用很像是我们去景区排队买票一样，第一个排队的人最先买到票，以此类推，俗话说: 先到先得。它的使用如下

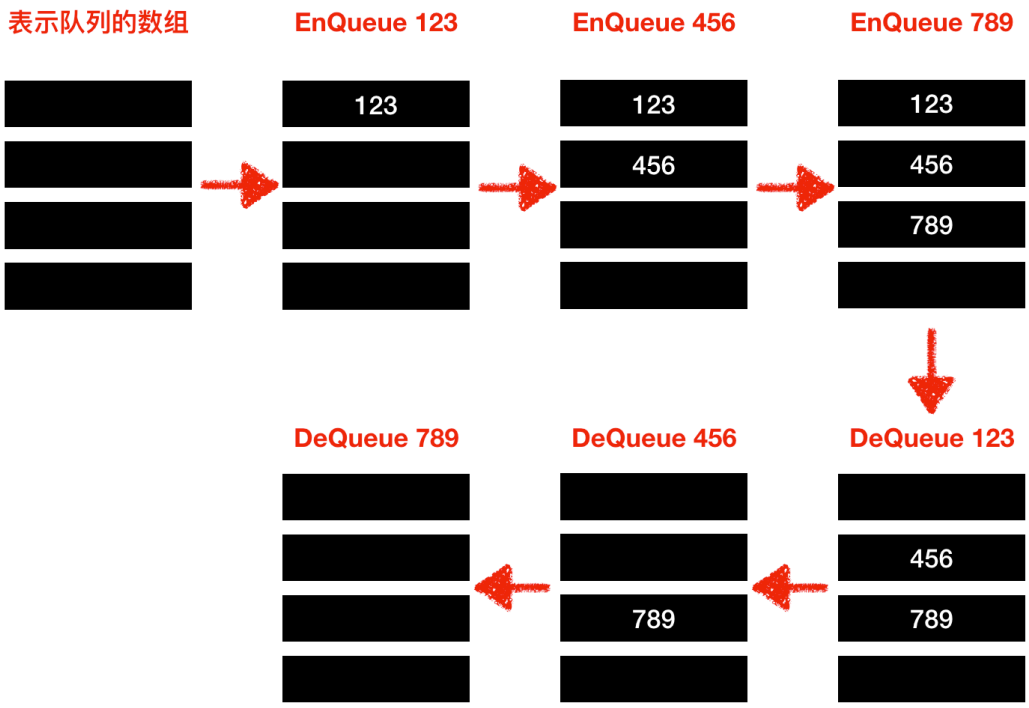
```
// 往队列中写入数据
EnQueue(123);
EnQueue(456);
EnQueue(789);

// 从队列中读出数据
m = DeQueue();
n = DeQueue();
o = DeQueue();
```

向队列中写入数据称为 EnQueue() 入列，从队列中读出数据称为 DeQueue() 。



队列的数据结构

与栈相对，FIFO 的方式表示队列中最先所保存的数据会优先被读取出来。

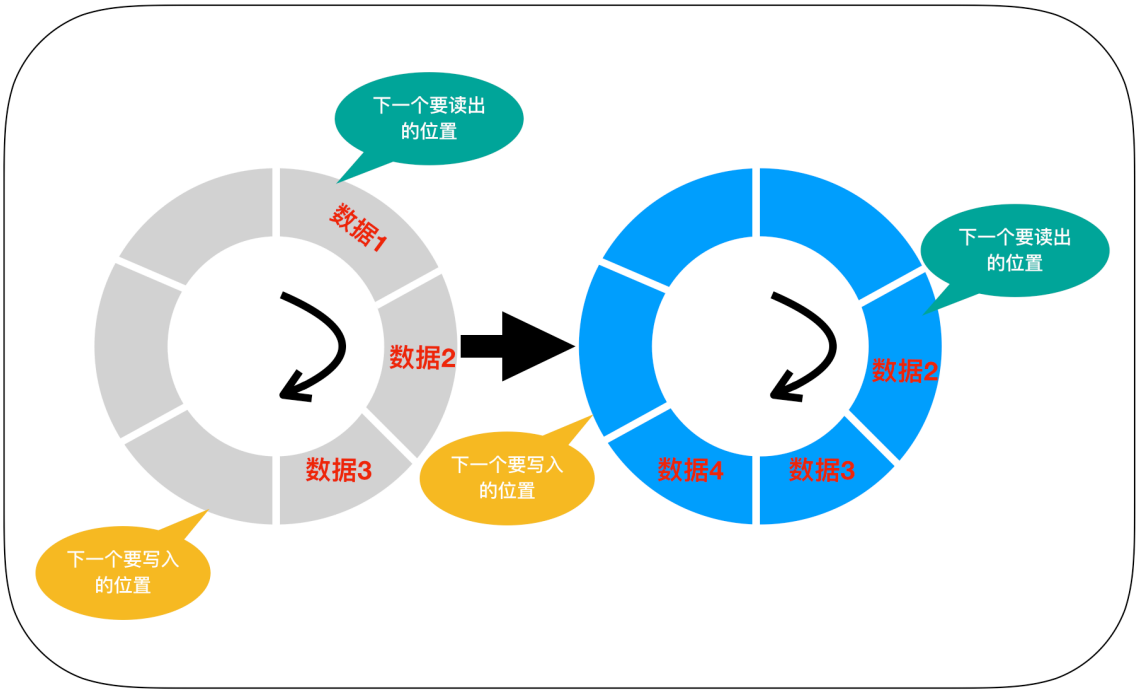


运行时队列的变化

队列的实现一般有两种：`顺序队列` 和 `循环队列`，我们上面的事例使用的是顺序队列，那么下面我们看一下循环队列的实现方式

环形缓冲区

循环队列一般是以 `环状缓冲区(ring buffer)` 的方式实现的，它是一种用于表示一个固定尺寸、头尾相连的缓冲区的数据结构，适合缓存数据流。假如我们要用 6 个元素的数组来实现一个环形缓冲区，这时可以从起始位置开始有序的存储数据，然后再按照存储时的顺序把数据读出。在数组的末尾写入数据后，后一个数据就会从缓冲区的头开始写。这样，数组的末尾和开头就连接了起来。



环形缓冲区模型

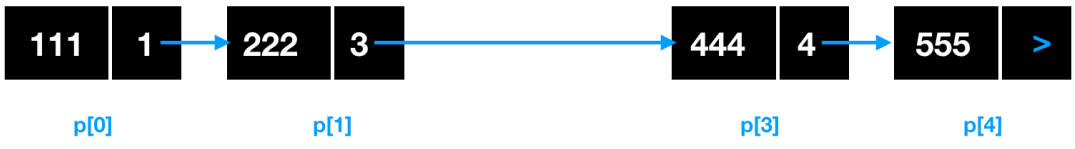
链表

下面我们来介绍一下 `链表` 和 `二叉树` ，它们都是可以不用考虑索引的顺序就可以对元素进行读写的方式。通过使用链表，可以高效的对数据元素进行 `添加` 和 `删除` 操作。而通过使用二叉树，则可以更高效的对数据进行 `检索` 。

在实现数组的基础上，除了数据的值之外，通过为其附带上下一个元素的索引，即可实现 `链表` 。数据的值和下一个元素的地址（索引）就构成了一个链表元素，如下所示

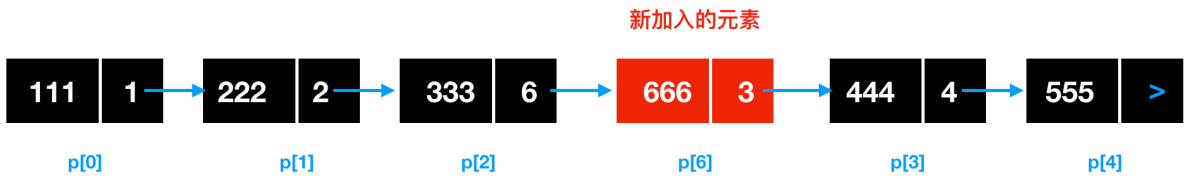
链表的数据结构

对链表的添加和删除都是非常高效的，我们来叙述一下这个添加和删除的过程，假如我们要删除地址为 `p[2]` 的元素，链表该如何变化呢?



链表的删除

我们可以看到，删除地址为 p[2] 的元素后，直接将链表剔除，并把 p[2] 前一个位置的元素 p[1] 的 指针域 指向 p[2] 下一个链表元素的数据区即可。



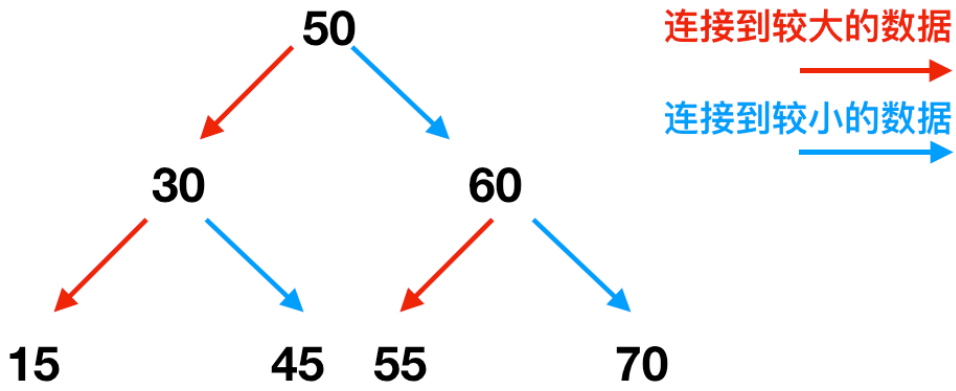
链表的添加

那么对于新添加进来的链表，需要确定 插入位置 ，比如要在 p[2] 和 p[3] 之间插入地址为 p[6] 的元素，需要将 p[6] 的前一个位置 p[2] 的指针域改为 p[6] 的地址，然后将 p[6] 的指针域改为 p[3] 的地址即可。

链表的添加不涉及到 数据的移动 ，所以链表的添加和删除很快，而数组的添加设计到数据的移动，所以比较慢，通常情况下，使用数组来检索数据，使用链表来进行添加和删除操作。

二叉树

二叉树 也是一种检索效率非常高的数据结构，二叉树是指在链表的基础上往数组追加元素时，考虑到数组的大小关系，将其分成左右两个方向的表现形式。假如我们把 50 这个值保存到了数组中，那么，如果接下来要进行值写入的话，就需要和50比较，确定谁大谁小，比50数值大的放右边，小的放左边，下图是二叉树的比较示例



二叉树比较示意图

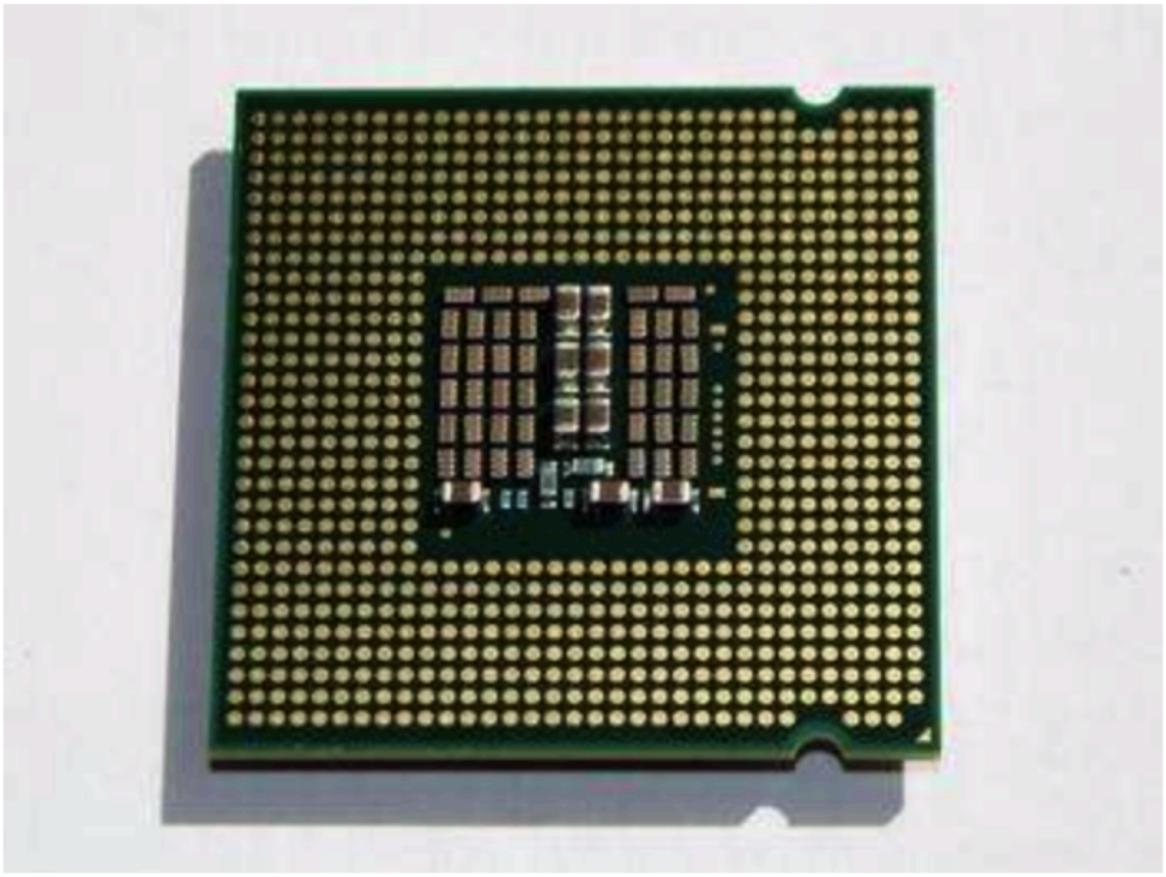
二叉树是由链表发展而来，因此二叉树在追加和删除元素方面也是同样有效的。

这一切的演变都是以内存为基础的。

我们都知道，计算机的底层都是使用二进制数据进行数据流传输的，那么为什么会使用二进制表示计算机呢？或者说，什么是二进制数呢？在拓展一步，如何使用二进制进行加减乘除？二进制数如何表示负数呢？本文将一一为你揭晓。

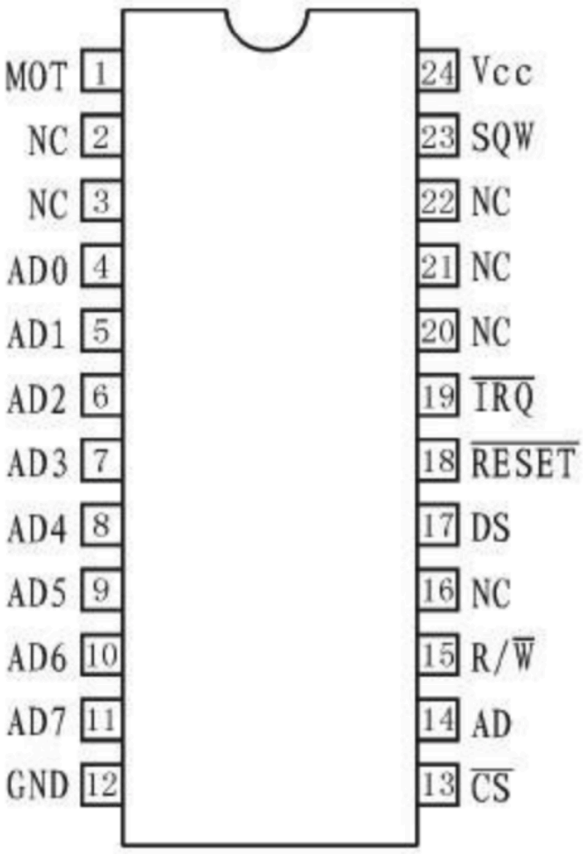
为什么用二进制表示

我们大家知道，计算机内部是由IC电子元件组成的，其中 `CPU` 和 `内存` 也是 IC 电子元件的一种，CPU和内存图如下



CPU

内存 CPU 和 内存使用IC电子元件作为基本单元，IC电子元件有不同种形状，但是其内部的组成单元称为一个个的引脚。有人说CPU 和 内存内部都是超大规模集成电路，其实IC 就是集成电路(Integrated Circuit)。



IC元件切面图

IC元件两侧排列的四方形块就是引脚，IC的所有引脚，只有两种电压: `0V` 和 `5V` ，IC的这种特性，也就决定了计算机的信息处理只能用 0 和 1 表示，也就是二进制来处理。一个引脚可以表示一个 0 或 1 ，所以二进制的表示方式就变成 0、1、10、11、100、101等，虽然二进制数并不是专门为 引脚 来设计的，但是和 IC引脚的特性非常吻合。

计算机的最小集成单位为 `位` ，也就是 `比特(bit)` ，二进制数的位数一般为 8位、16位、32位、64位，也就是 8 的倍数，为什么要跟 8 扯上关系呢？ 因为在计算机中，把 8 位二进制数称为 **一个字节**， 一个字节有 8 位，也就是由 8个bit构成。

> 为什么1个字节等于8位呢？因为 8 位能够涵盖所有的字符编码，这个记住就可以了。

字节是最基本的计量单位，位是最小单位。

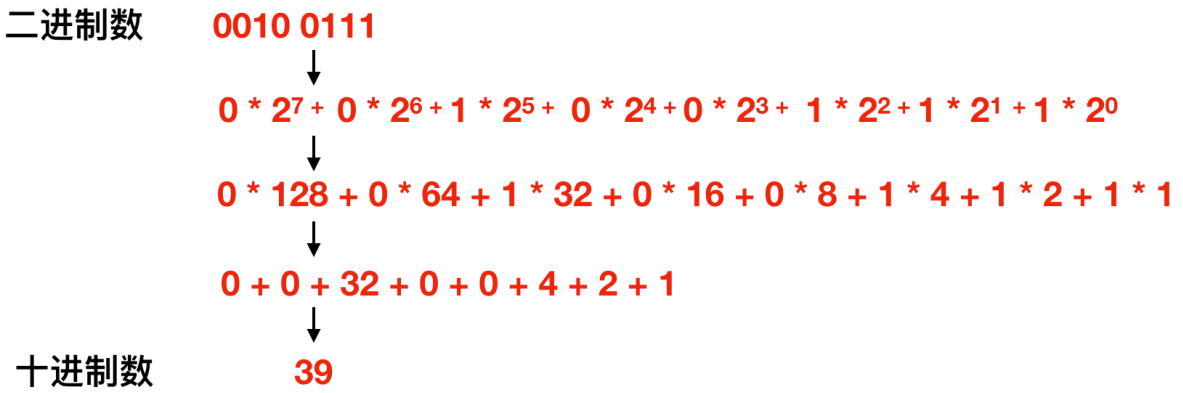
用字节处理数据时，如果数字小于存储数据的字节数 (= 二进制的位数)，那么高位就用 0 填补，高位和数学的数字表示是一样的，**左侧表示高位，右侧表示低位。**比如 这个六位数用二进制数来表示就是 `100111` ，只有6位，高位需要用 0 填充，填充完后是 `00100111` ，占一个字节，如果用 16 位表示就是 `0000 0000 0010 0111` 占用两个字节。

我们一般口述的 32 位和 64位的计算机一般就指的是处理位数，32 位一次可以表示 4个字节，64位一次可以表示8个字节的二进制数。

我们一般在软件开发中用十进制数表示的逻辑运算等，也会被计算机转换为二进制数处理。对于二进制数，计算机不会区分他是 图片、音频文件还是数字，这些都是一些数据的结合体。

什么是二进制数

那么什么是二进制数呢？为了说明这个问题，我们先把 `00100111` 这个数转换为十进制数看一下，二进制数转换为十进制数，直接将各位置上的值 * 位权即可，那么我们将上面的数值进行转换



二进制转十进制表示图

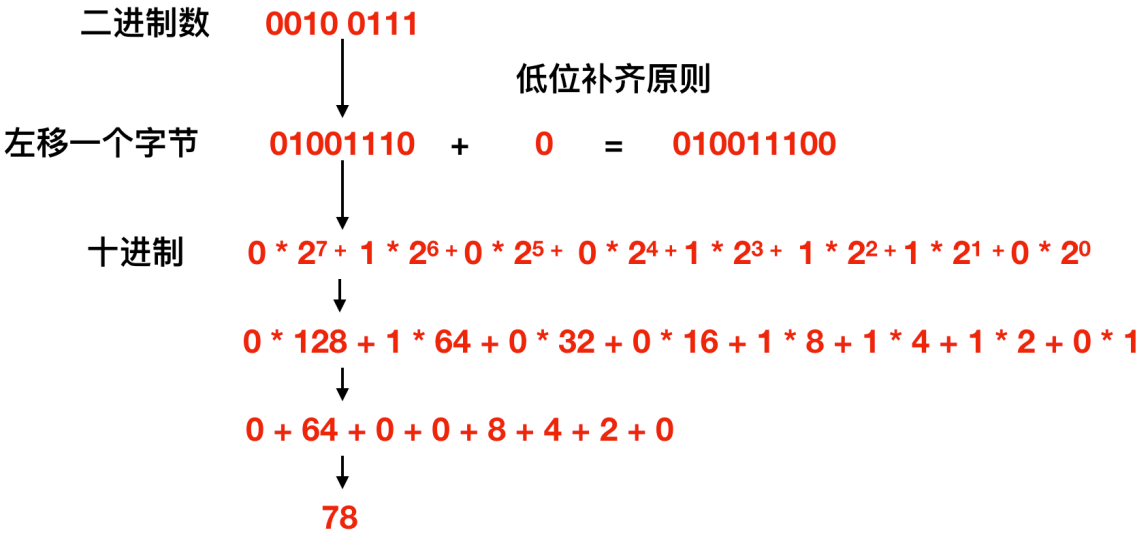
也就是说，二进制数代表的 `00100111` 转换成十进制就是 39，这个 39 并不是 3 和 9 两个数字连着写，而是 3 * 10 + 9 * 1，这里面的 `10 , 1` 就是位权，以此类推，上述例子中的位权从高位到低位依次就是 `7 6 5 4 3 2 1 0` 。这个位权也叫做次幂，那么最高位就是2的7次幂，2的6次幂 等等。二进制数的运算每次都会以2为底，这个2 指得就是基数，那么十进制数的基数也就是 10 。在任何情况下位权的值都是 **数的位数 - 1**，那么第一位的位权就是 1 - 1 = 0， 第二位的位权就睡 2 - 1 = 1，以此类推。

那么我们所说的二进制数其实就是 用0和1两个数字来表示的数，它的基数为2，它的数值就是每个数的位数 * 位权再求和得到的结果，我们一般来说数值指的就是十进制数，那么它的数值就是 3 * 10 + 9 * 1 = 39。

移位运算和乘除的关系

在了解过二进制之后，下面我们来看一下二进制的运算，和十进制数一样，加减乘除也适用于二进制数，只要注意逢 2 进位即可。二进制数的运算，也是计算机程序所特有的运算，因此了解二进制的运算是必须要掌握的。

首先我们来介绍 `移位` 运算，移位运算是指将二进制的数值的各个位置上的元素坐左移和右移操作，见下图



移位过程

上述例子中还是以 39 为例，我们先把十进制的39 转换为二进制的 `0010 0111`，然后 `向左移位 <<` 一个字节，也就变成了 `0100 1110`，那么再把此二进制数转换为十进制数就是上面的78， 十进制的 78 竟然是 十进制39 的2倍关系。我们在让 `0010 0111` 左移两位，也就是 `1001 1100`，得出来的值是 156，相当于扩大了四倍！

因此你可以得出来此结论，左移相当于是数值扩大的操作，那么 `右移 >>` 呢？按理说右移应该是缩小 1/2，1/4 倍，但是39 缩小二倍和四倍不就变成小数了吗？这个怎么表示呢？请看下一节

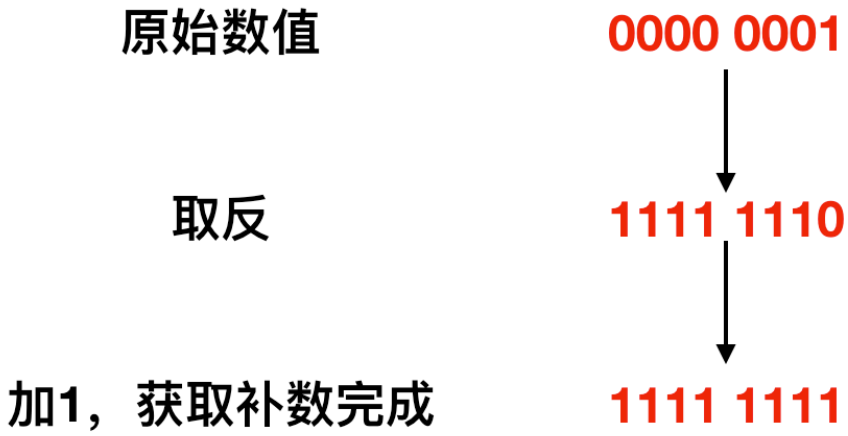
便于计算机处理的补数

刚才我们没有介绍右移的情况，是因为右移之后空出来的高位数值，有 0 和 1 两种形式。要想区分什么时候补0什么时候补1，首先就需要掌握二进制数表示 `负数` 的方法。

二进制数中表示负数值时，一般会把最高位作为符号来使用，因此我们把这个最高位当作符号位。 符号位是 0 时表示 `正数`，是 1 时表示 `负数`。那么 -1 用二进制数该如何表示呢？可能很多人会这么认为： 因为 1 的二进制数是 `0000 0001`，最高位是符号位，所以正确的表示 -1 应该是 `1000 0001`，但是这个答案真的对吗？

计算机世界中是没有减法的，计算机在做减法的时候其实就是在做加法，也就是用加法来实现的减法运算。比如 100 - 50，其实计算机来看的时候应该是 100 + (-50)，为此，在表示负数的时候就要用到 `二进制补数`，补数就是用正数来表示的负数。

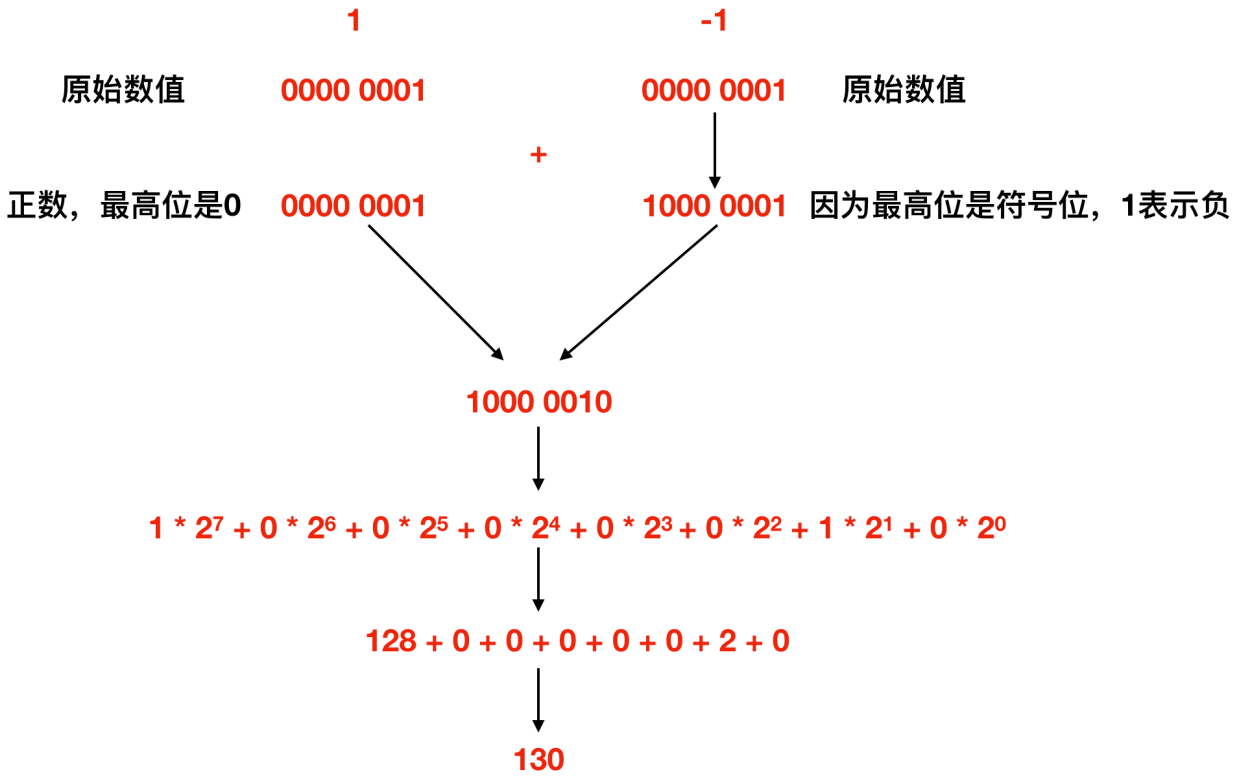
为了获得 `补数`，我们需要将二进制的各数位的数值全部取反，然后再将结果 + 1 即可，先记住这个结论，下面我们来演示一下。



-1 取反过程

具体来说，就是需要先获取某个数值的二进制数，然后对二进制数的每一位做取反操作(0 ---> 1 , 1 ---> 0)，最后再对取反后的数 +1 ，这样就完成了补数的获取。

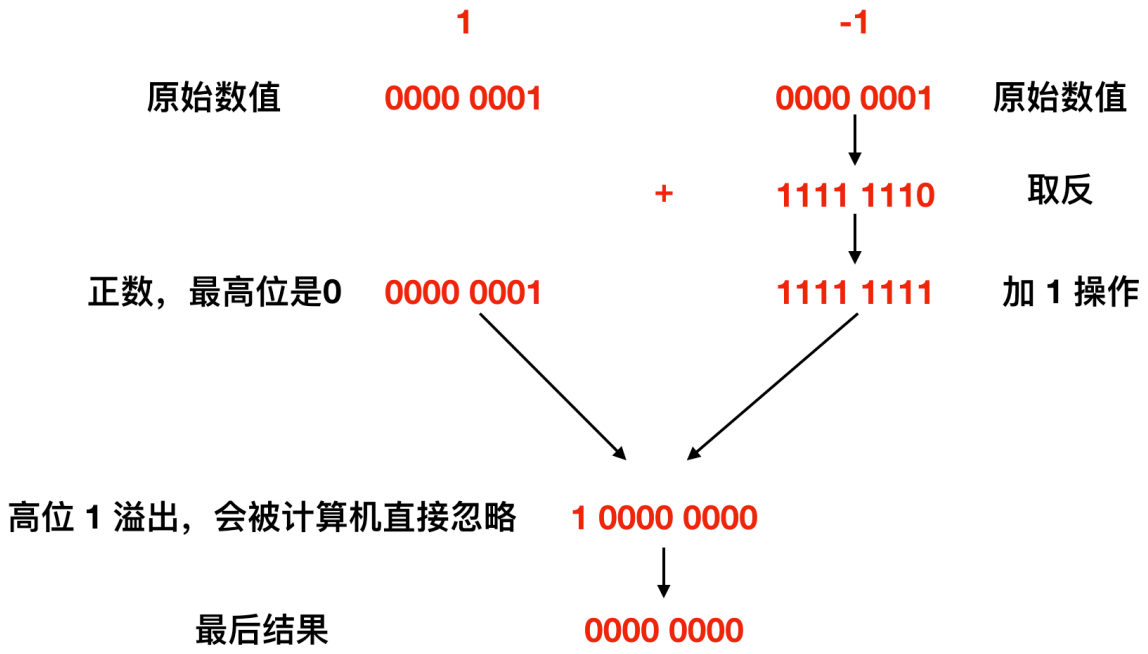
补数的获取，虽然直观上不易理解，但是逻辑上却非常严谨，比如我们来看一下 1 - 1 的这个过程，我们先用上面的这个 `1000 0001` (它是1的补数，不知道的请看上文，正确性先不管，只是用来做一下计算)来表示一下



1 - 1 分析图

奇怪，1 - 1 会变成 130 ，而不是0，所以可以得出结论 `1000 0001` 表示 -1 是完全错误的。

那么正确的该如何表示呢？其实我们上面已经给出结果了，那就是 `1111 1111` ，来论证一下它的正确性

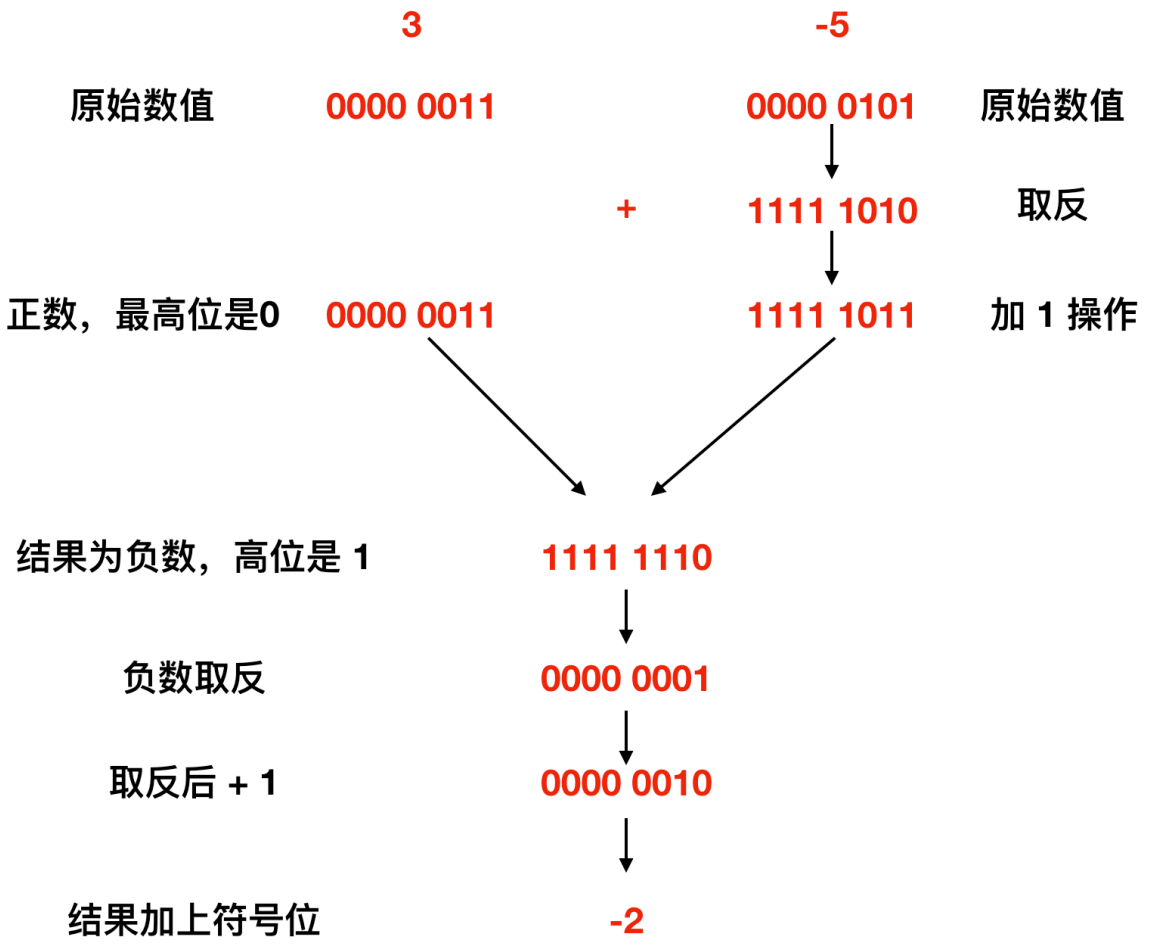


1 - 1 正确的分析图

我们可以看到 1 - 1 其实实际上就是 1 + (-1)，对 -1 进行上面的取反 + 1 后变为 `1111 1111`，然后与 1 进行加法运算，得到的结果是九位的 `1 0000 0000`，结果发生了 `溢出`，计算机会直接忽略掉溢出位，也就是直接抛掉 最高位 1，变为 `0000 0000`。也就是 0，结果正确，所以 `1111 1111` 表示的就是 -1 。

所以负数的二进制表示就是先求其补数，补数的求解过程就是对原始数值的二进制数各位取反，然后将结果 + 1，

当然，结果不为 0 的运算同样也可以通过补数求得正确的结果。不过，有一点需要注意，当运算结果为负的时候，计算结果的值也是以补数的形式出现的，比如 3 - 5 这个运算，来看一下解析过程



3 - 5 的解析过程

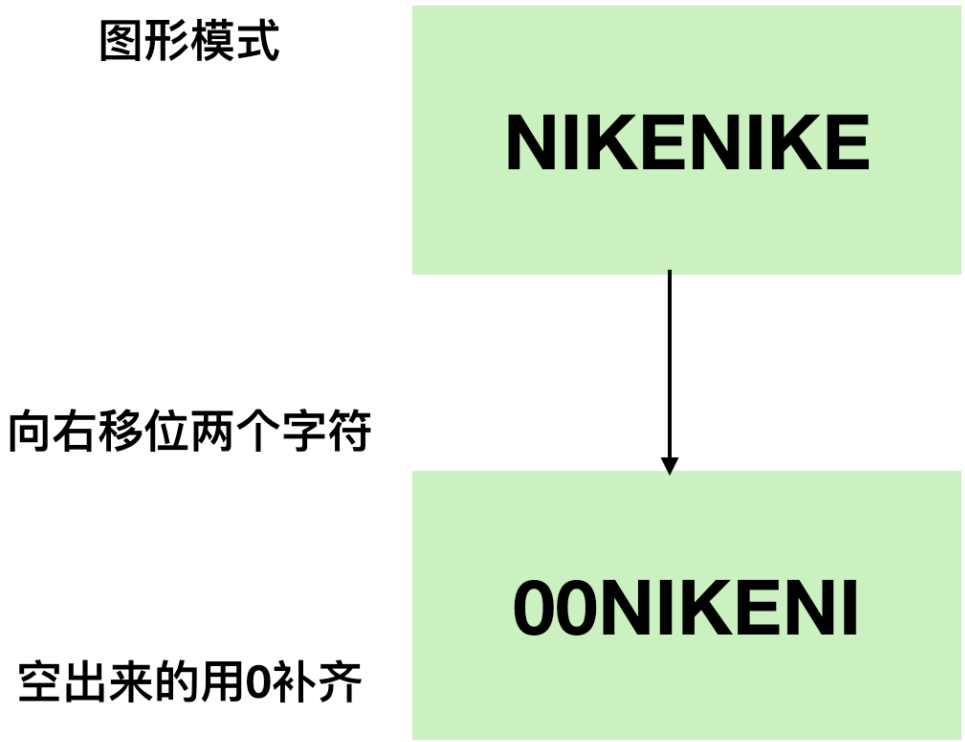
3 - 5 的运算，我们按着上面的思路来过一遍，计算出来的结果是 `1111 1110`，我们知道，这个数值肯定表示负数，但是负数无法直接用十进制表示，需要对其取反+ 1，算出来的结果是 2，因为 `1111 1110` 的高位是 1，所以最终的结果是 -2。

编程语言的数据类型中，有的可以处理负数，有的不可以。比如 C语言中不能处理负数的 `unsigned short` 类型，也有能处理负数的 `short` 类型，都是两个字节的变量，它们都有 2 的十六次幂种值，但是取值范围不一样，short 类型的取值范围是 -32768 - 32767， unsigned short 的取值范围是 0 - 65536。

仔细思考一下补数的机制，就能明白 -32768 比 32767 多一个数的原因了，最高位是 0 的正数有 0 ~ 32767 共 32768 个，其中包括0。最高位是 1 的负数，有 -1 ~ -32768 共 32768 个，其中不包含0。0 虽然既不是正数也不是负数，但是考虑到其符号位，就将其归为了正数。

算数右移和逻辑右移的区别

在了解完补数后，我们重新考虑一下右移这个议题，右移在移位后空出来的最高位有两种情况 `0 和 1` 。当二进制数的值表示图形模式而非数值时，移位后需要在最高位补0，类似于霓虹灯向右平移的效果，这就被称为 `逻辑右移` 。



逻辑右移示意图

将二进制数作为带符号的数值进行右移运算时，移位后需要在最高位填充移位前符号位的值(0 或 1)。这就被称为 `算数右移` 。如果数值使用补数表示的负数值，那么右移后在空出来的最高位补 1，就可以正确的表示 `1/2,1/4,1/8` 等的数值运算。如果是正数，那么直接在空出来的位置补 0 即可。

下面来看一个右移的例子。将 -4 右移两位，来各自看一下移位示意图

逻辑右移和算数右移示意图

如上图所示，在逻辑右移的情况下， -4 右移两位会变成 `63` ， 显然不是它的 1/4，所以不能使用逻辑右移，那么算数右移的情况下，右移两位会变为 `-1` ，显然是它的 1/4，故而采用算数右移。

那么我们可以得出来一个结论：**左移时，无论是图形还是数值，移位后，只需要将低位补 0 即可；右移时，需要根据情况判断是逻辑右移还是算数右移。**

下面介绍一下符号扩展：**将数据进行符号扩展是为了产生一个位数加倍、但数值大小不变的结果，以满足有些指令对操作数位数的要求，例如倍长于除数的被除数，再如将数据位数加长以减少计算过程中的误差。**

以8位二进制为例，符号扩展就是指在保持值不变的前提下将其转换成为16位和32位的二进制数。将 `0111 1111` 这个正的 8位二进制数转换成为 16位二进制数时，很容易就能够得出 `0000 0000 0111 1111` 这个正确的结果，但是像 `1111 1111` 这样的补数来表示的数值，该如何处理？直接将其表示成为 `1111 1111 1111 1111` 就可以了。也就是说，不管正数还是补数表示的负数，只需要将 0 和 1 填充高位即可。

逻辑运算的窍门

掌握逻辑和运算的区别是：将二进制数表示的信息作为四则运算的数值来处理就是 `算数` ，像图形那样，将数值处理为单纯的 `0` 和 `1` 的罗列就是 `逻辑`

计算机能够处理的运算，大体可分为逻辑运算和算数运算， `算数运算` 指的是加减乘除四则运算； `逻辑运算` 指的是对二进制各个数位的 0 和 1分别进行处理的运算，包括**逻辑非(NOT运算)、逻辑与(AND运算)、逻辑或(OR运算)和逻辑异或(XOR运算)**四种。

- `逻辑非` 指的是将 0 变成 1，1 变成 0 的取反操作
- `逻辑与` 指的是"两个都是 1 时，运算结果才是 1，其他情况下是 0"
- `逻辑或` 指的是"至少有一方是 1 时，运算结果为 1，其他情况下运算结果都是 0"
- `逻辑异或` 指的是 "其中一方是 1，另一方是 0时运算结果才是 1，其他情况下是 0"

表 2-1　逻辑非（NOT）的真值表

A 的值	NOT A 的运算结果
0	1
1	0

表 2-2　逻辑与（AND）的真值表

A 的值	B 的值	A AND B 的运算结果
0	0	0
0	1	0
1	0	0
1	1	1

表 2-3　逻辑或（OR）的真值表

A 的值	B 的值	A OR B 的运算结果
0	0	0
0	1	1
1	0	1
1	1	1

表 2-4　逻辑异或（XOR）的真值表

A 的值	B 的值	A XOR B 的运算结果
0	0	0
0	1	1
1	0	1
1	1	0

与或非异或真值表

掌握逻辑运算的窍门，就是要摒弃二进制数表示数值这一个想法。大家不要把二进制数表示的值当作数值，应该把它看成是 开关上的 `ON/OFF` 。

认识压缩算法

我们想必都有过 `压缩` 和 `解压缩` 文件的经历，当文件太大时，我们会使用文件压缩来降低文件的占用空间。比如微信上传文件的限制是100 MB，我这里有个文件夹无法上传，但是我解压完成后的文件一定会小于 100 MB，那么我的文件就可以上传了。

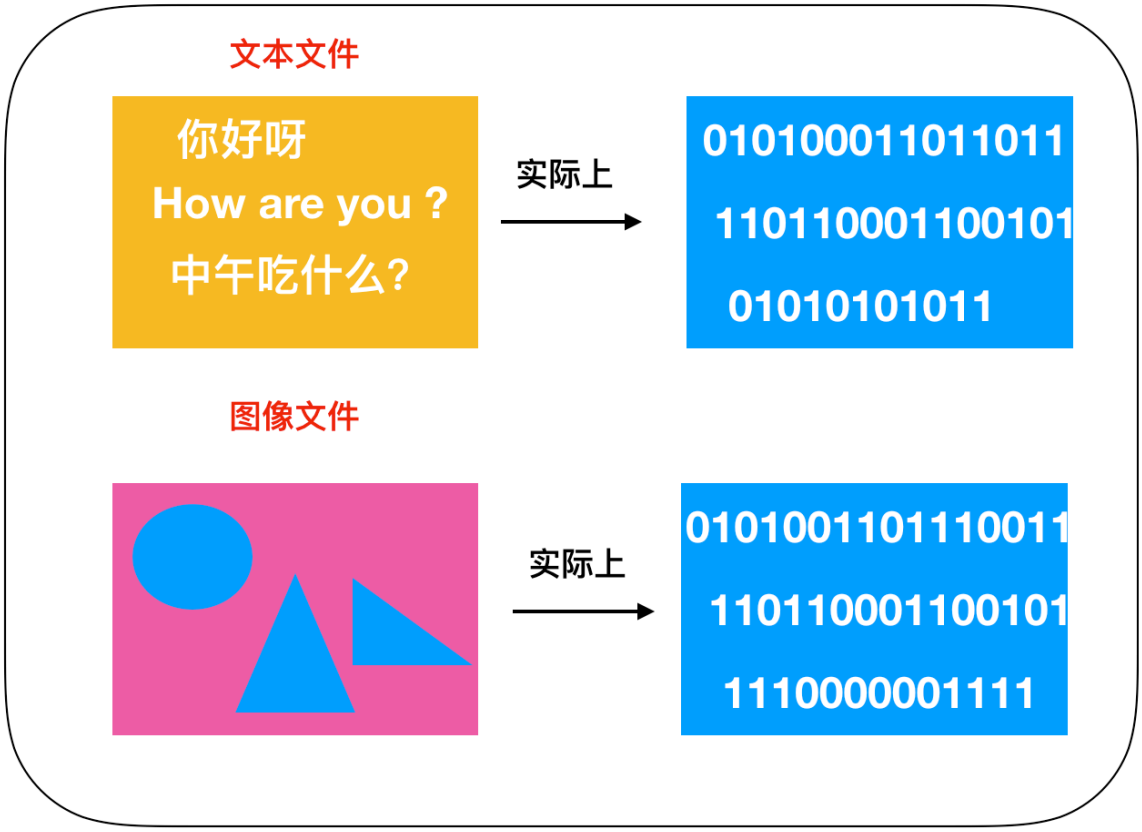
此外，我们把相机拍完的照片保存到计算机上的时候，也会使用压缩算法进行文件压缩，文件压缩的格式一般是 `JPEG` 。

那么什么是压缩算法呢？压缩算法又是怎么定义的呢？在认识算法之前我们需要先了解一下文件是如何存储的

文件存储

文件是将数据存储在磁盘等存储媒介的一种形式。程序文件中最基本的存储数据单位是 `字节` 。文件的大小不管是 xxxKB、xxxMB等来表示，就是因为文件是以字节 `B = Byte` 为单位来存储的。

文件就是字节数据的集合。用 1 字节（8 位）表示的字节数据有 256 种，用二进制表示的话就是 0000 0000 - 1111 1111 。如果文件中存储的数据是文字，那么该文件就是文本文件。如果是图形，那么该文件就是图像文件。在任何情况下，文件中的字节数都是 `连续存储` 的。



文件是字节数据的集合体

压缩算法的定义

上面介绍了文件的集合体其实就是一堆字节数据的集合，那么我们就可以来给压缩算法下一个定义。

`压缩算法（compaction algorithm）` 指的就是数据压缩的算法，主要包括压缩和还原（解压缩）的两个步骤。

其实就是在不改变原有文件属性的前提下，降低文件字节空间和占用空间的一种算法。

根据压缩算法的定义，我们可将其分成不同的类型：

有损和无损

无损压缩：能够 `无失真地` 从压缩后的数据重构，准确地还原原始数据。可用于对数据的准确性要求严格的场合，如可执行文件和普通文件的压缩、磁盘的压缩，也可用于多媒体数据的压缩。该方法的压缩比较小。如差分编码、RLE、Huffman编码、LZW编码、算术编码。

有损压缩：有失真， `不能完全准确地` 恢复原始数据，重构的数据只是原始数据的一个近似。可用于对数据的准确性要求不高的场合，如多媒体数据的压缩。该方法的压缩比较大。例如预测编码、音感编码、分形压缩、小波压缩、JPEG/MPEG。

对称性

如果编解码算法的复杂性和所需时间差不多，则为对称的编码方法，多数压缩算法都是对称的。但也有不对称的，一般是编码难而解码容易，如 Huffman 编码和分形编码。但用于密码学的编码方法则相反，是编码容易，而解码则非常难。

帧间与帧内

在视频编码中会同时用到帧内与帧间的编码方法，帧内编码是指在一帧图像内独立完成的编码方法，同静态图像的编码，如 JPEG；而帧间编码则需要参照前后帧才能进行编解码，并在编码过程中考虑对帧之间的时间冗余的压缩，如 MPEG。

实时性

在有些多媒体的应用场合，需要实时处理或传输数据（如现场的数字录音和录影、播放MP3/RM/VCD/DVD、视频/音频点播、网络现场直播、可视电话、视频会议），编解码一般要求延时≤50 ms。这就需要简单/快速/高效的算法和高速/复杂的CPU/DSP芯片。

分级处理

有些压缩算法可以同时处理不同分辨率、不同传输速率、不同质量水平的多媒体数据，如JPEG2000、MPEG-2/4。

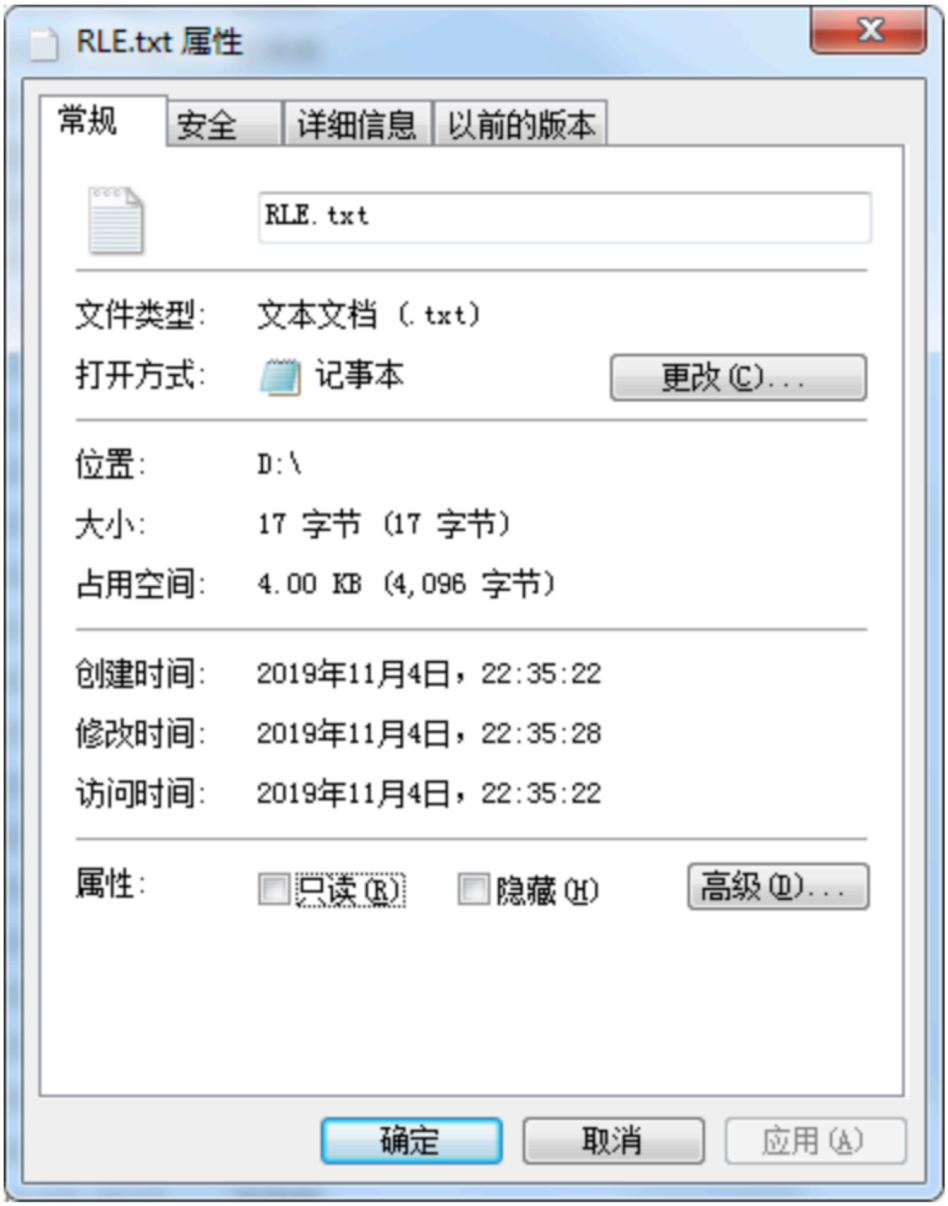
这些概念有些抽象，主要是为了让大家了解一下压缩算法的分类，下面我们就对具体的几种常用的压缩算法来分析一下它的特点和优劣

几种常用压缩算法的理解

RLE 算法的机制

接下来就让我们正式看一下文件的压缩机制。首先让我们来尝试对 `AAAAAABBCDDEEEEEF` 这 17 个半角字符的文件（文本文件）进行压缩。虽然这些文字没有什么实际意义，但是很适合用来描述 `RLE` 的压缩机制。

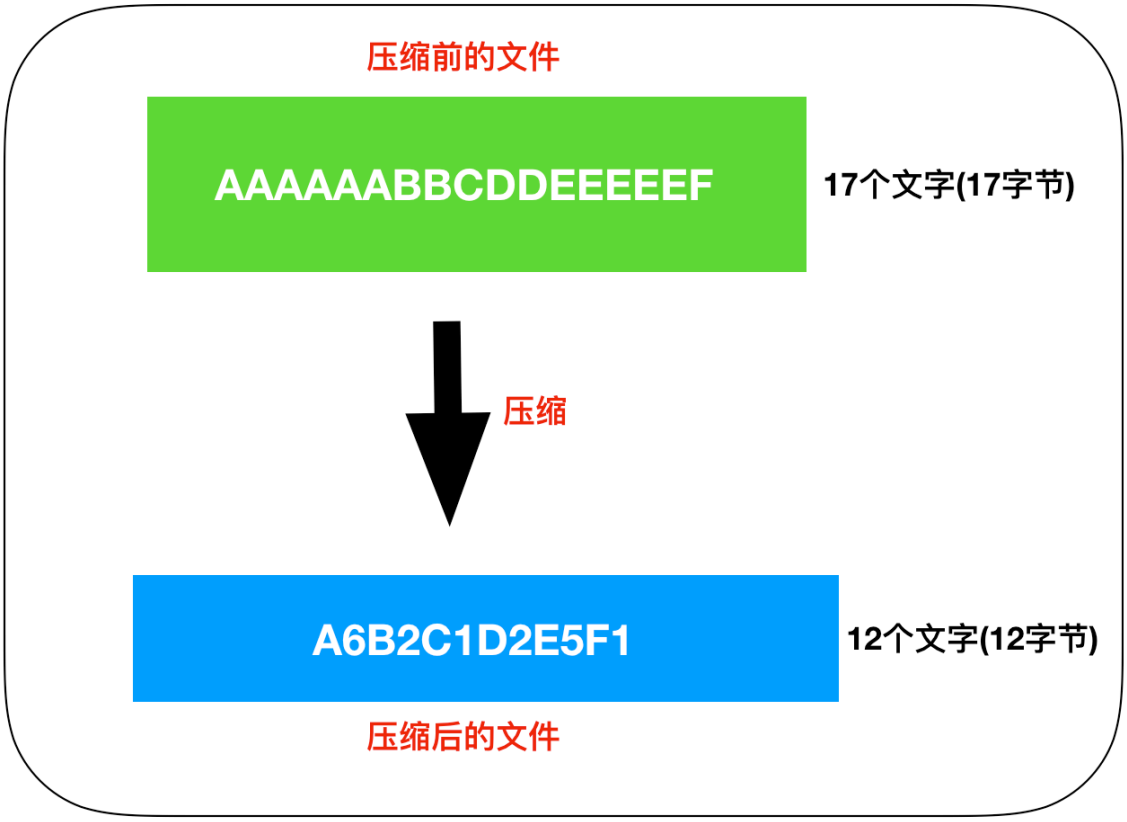
由于半角字符（其实就是英文字符）是作为 1 个字节保存在文件中的，所以上述的文件的大小就是 17 字节。如图



17 英文字符占用空间

那么，如何才能压缩该文件呢？大家不妨也考虑一下，只要是能够使文件小于 17 字节，我们可以使用任何压缩算法。

最显而易见的一种压缩方式我觉得你已经想到了，就是把相同的字符 `去重化` ，也就是 `字符 * 重复次数` 的方式进行压缩。所以上面文件压缩后就会变成下面这样



文件压缩后的文件大小

从图中我们可以看出，**AAAAAABBCDDEEEEEF** 的17个字符成功被压缩成了 **A6B2C1D2E5F1** 的12个字符，也就是 12 / 17 = 70%，压缩比为 70%，压缩成功了。

像这样，把文件内容用 `数据 * 重复次数` 的形式来表示的压缩方法成为 `RLE(Run Length Encoding，行程长度编码)` 算法。RLE 算法是一种很好的压缩方法，经常用于压缩传真的图像等。因为图像文件的本质也是字节数据的集合体，所以可以用 RLE 算法进行压缩

RLE 算法的缺点

RLE 的压缩机制比较简单，所以 RLE 算法的程序也比较容易编写，所以使用 RLE 的这种方式更能让你体会到压缩思想，但是 RLE 只针对特定序列的数据管用，下面是 RLE 算法压缩汇总

文件类型	压缩前文件大小	压缩后文件大小	压缩比率
文本文件	14862字节	29065字节	199%
图像文件	96062字节	38328字节	40%
EXE文件	24576字节	15198字节	62%

通过上表可以看出，使用 RLE 对文本文件进行压缩后的数据不但没有减小反而增大了！几乎是压缩前的两倍！因为文本字符种连续的字符并不多见。

就像上面我们探讨的这样，RLE 算法只针对 `连续` 的字节序列压缩效果比较好，假如有一连串不相同的字符该怎么压缩呢？比如说 `ABCDEFGHIJKLMNOPQRSTUVWXYZ`，26个英文字母所占空间应该是 26 个字节，我们用 RLE 压缩算法压缩后的结果为 `A1B1C1D1E1F1G1H1I1J1K1L1M1N1O1P1Q1R1S1T1U1V1W1X1Y1Z1`，所占用 52 个字节，压缩完成后

的容量没有减少反而增大了！这显然不是我们想要的结果，所以这种情况下就不能再使用 RLE 进行压缩。

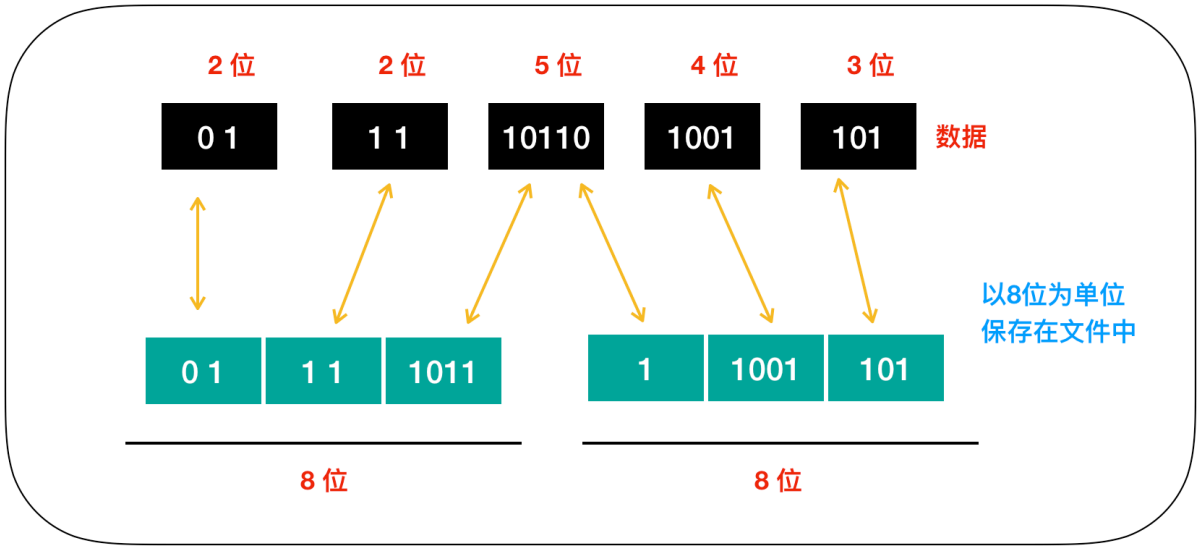
哈夫曼算法和莫尔斯编码

下面我们来介绍另外一种压缩算法，即哈夫曼算法。在了解哈夫曼算法之前，你必须舍弃 半角英文数字的1个字符是1个字节(8位)的数据 。下面我们就来认识一下哈夫曼算法的基本思想。

文本文件是由不同类型的字符组合而成的，而且不同字符出现的次数也是不一样的。例如，在某个文本文件中，A 出现了 100次左右，Q仅仅用到了 3 次，类似这样的情况很常见。哈夫曼算法的关键就在于**多次出现的数据用小于 8 位的字节数表示，不常用的数据则可以使用超过 8 位的字节数表示**。A 和 Q 都用 8 位来表示时，原文件的大小就是 100次 * 8 位 + 3次 * 8 位 = 824位，假设 A 用 2 位，Q 用 10 位来表示就是 2 * 100 + 3 * 10 = 230 位。

> 不过要注意一点，最终磁盘的存储都是以8位为一个字节来保存文件的。

哈夫曼算法比较复杂，在深入了解之前我们先吃点 甜品 ，了解一下 莫尔斯编码 ，你一定看过美剧或者战争片的电影，在战争中的通信经常采用莫尔斯编码来传递信息，例如下面



接下来我们来讲解一下莫尔斯编码，下面是莫尔斯编码的 示例 ，大家把 1 看作是短点(嘀)，把 11 看作是长点(嗒)即可。

字符	对应的位数据	位长
A	1011	4位
B	11010101	8位
C	110101101	9位
D	110101	6位
E	1	1位
F	10101101	8位
时间间隔	00	2位

1：短点　　11：长点　　0：短点和长点的分隔符

莫尔斯编码一般把文本中出现最高频率的字符用 `短编码` 来表示。如表所示，假如表示短点的位是 1，表示长点的位是 11 的话，那么 E（嘀）这一数据的字符就可以用 1 来表示，C（滴答滴答）就可以用 9 位的 `110101101` 来表示。在实际的莫尔斯编码中，如果短点的长度是 1 ，长点的长度就是 3，短点和长点的间隔就是1。这里的长度指的就是声音的长度。比如我们想用上面的 AAAAAABBCDDEEEEEF 例子来用莫尔斯编码重写，在莫尔斯曼编码中，各个字符之间需要加入表示时间间隔的符号。这里我们用 00 加以区分。

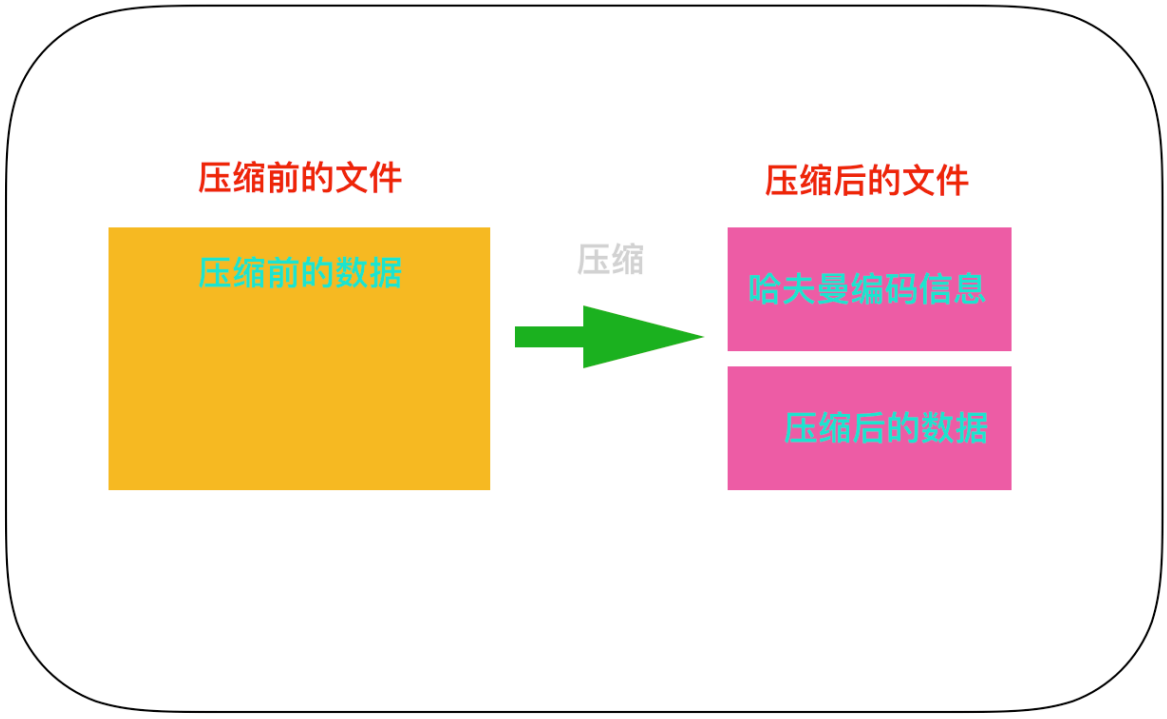
所以，AAAAAABBCDDEEEEEF 这个文本就变为了 A * 6 次 + B * 2次 + C * 1次 + D * 2次 + E * 5次 + F * 1次 + 字符间隔 * 16 = 4 位 * 6次 + 8 位 * 2次 + 9 位 * 1 次 + 6位 * 2次 + 1位 * 5次 + 8 位 * 1次 + 2位 * 16次 = 106位 = 14字节。

所以使用莫尔斯电码的压缩比为 14 / 17 = 82%。效率并不太突出。

用二叉树实现哈夫曼算法

刚才已经提到，莫尔斯编码是根据日常文本中各字符的出现频率来决定表示各字符的编码数据长度的。不过，在该编码体系中，对 AAAAAABBCDDEEEEEF 这种文本来说并不是效率最高的。

下面我们来看一下哈夫曼算法。哈夫曼算法是指，为各压缩对象文件分别构造最佳的编码体系，并以该编码体系为基础来进行压缩。因此，用什么样的编码（哈夫曼编码）对数据进行分割，就要由各个文件而定。用哈夫曼算法压缩过的文件中，存储着哈夫曼编码信息和压缩过的数据。

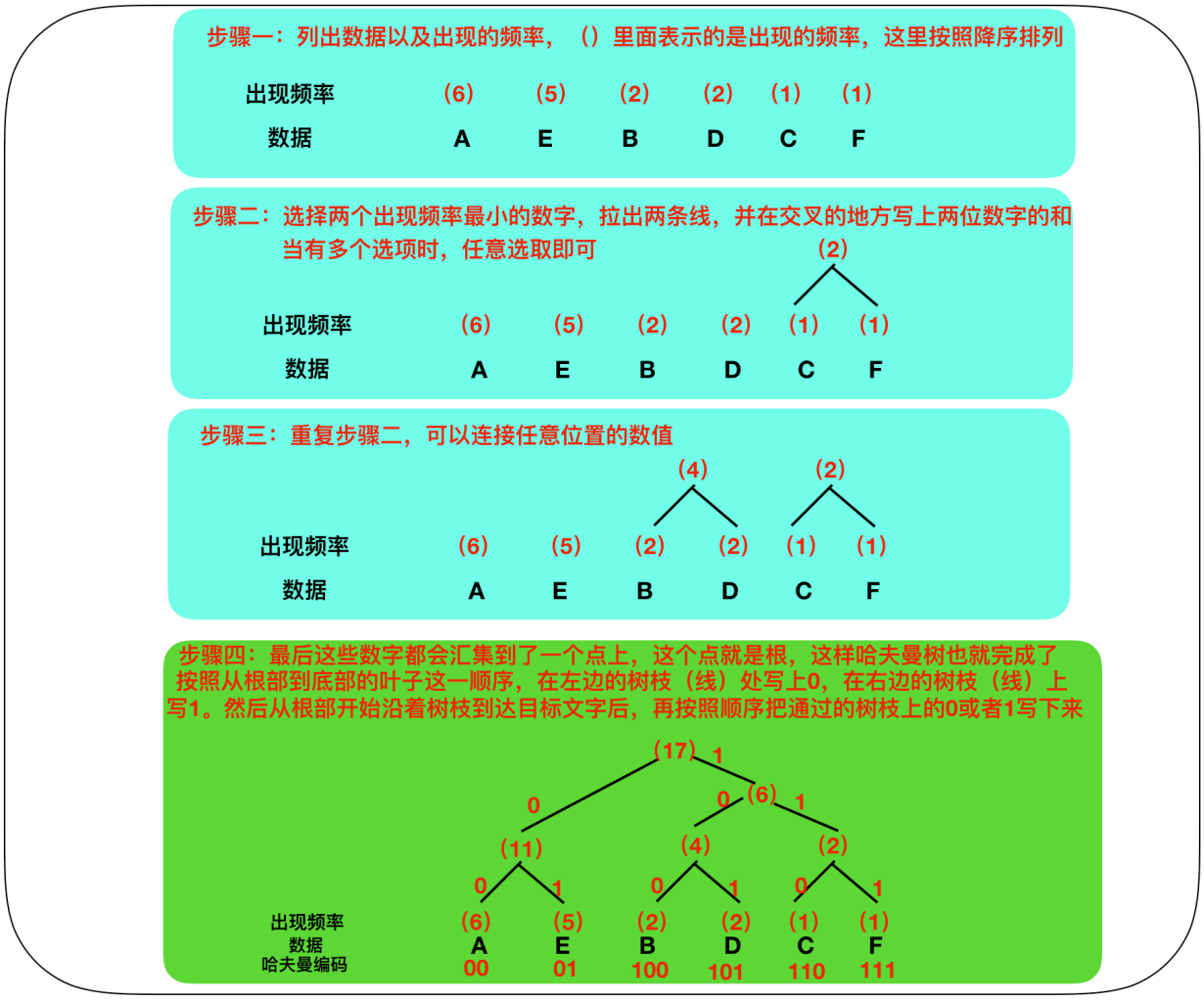


接下来，我们在对 AAAAAABBCDDEEEEEF 中的 A - F 这些字符，按照 出现频率高的字符用尽量少的位数编码来表示 这一原则进行整理。按照出现频率从高到低的顺序整理后，结果如下，同时也列出了编码方案。

字符	出现频率	编码（方案）	位数
A	6	0	1
E	5	1	1
B	2	10	2
D	2	11	2
C	1	100	3
F	1	101	3

在上表的编码方案中，随着出现频率的降低，字符编码信息的数据位数也在逐渐增加，从最开始的 1 位、2位依次增加到3位。不过这个编码体系是存在问题的，你不知道100这个3位的编码，它的意思是用 1、0、0这三个编码来表示 E、A、A 呢？还是用10、0来表示 B、A 呢？还是用100来表示 C 呢。

而在哈夫曼算法中，通过借助哈夫曼树的构造编码体系，即使在不使用字符区分符号的情况下，也可以构建能够明确进行区分的编码体系。不过哈夫曼树的算法要比较复杂，下面是一个哈夫曼树的构造过程。



自然界树的从根开始生叶的，而哈夫曼树则是叶生枝

哈夫曼树能够提升压缩比率

使用哈夫曼树之后，出现频率越高的数据所占用的位数越少，这也是哈夫曼树的核心思想。通过上图的步骤二可以看出，枝条连接数据时，我们是从出现频率较低的数据开始的。这就意味着出现频率低的数据到达根部的枝条也越多。而枝条越多则意味着编码的位数随之增加。

接下来我们来看一下哈夫曼树的压缩比率，用上图得到的数据表示 AAAAAABBCDDEEEEEF 为 000000000000 100100 110 101101 0101010101 111，40位 = 5 字节。压缩前的数据是 17 字节，压缩后的数据竟然达到了惊人的5 字节，也就是压缩比率 = 5 / 17 = 29% 如此高的压缩率，简直是太惊艳了。

大家可以参考一下，无论哪种类型的数据，都可以用哈夫曼树作为压缩算法

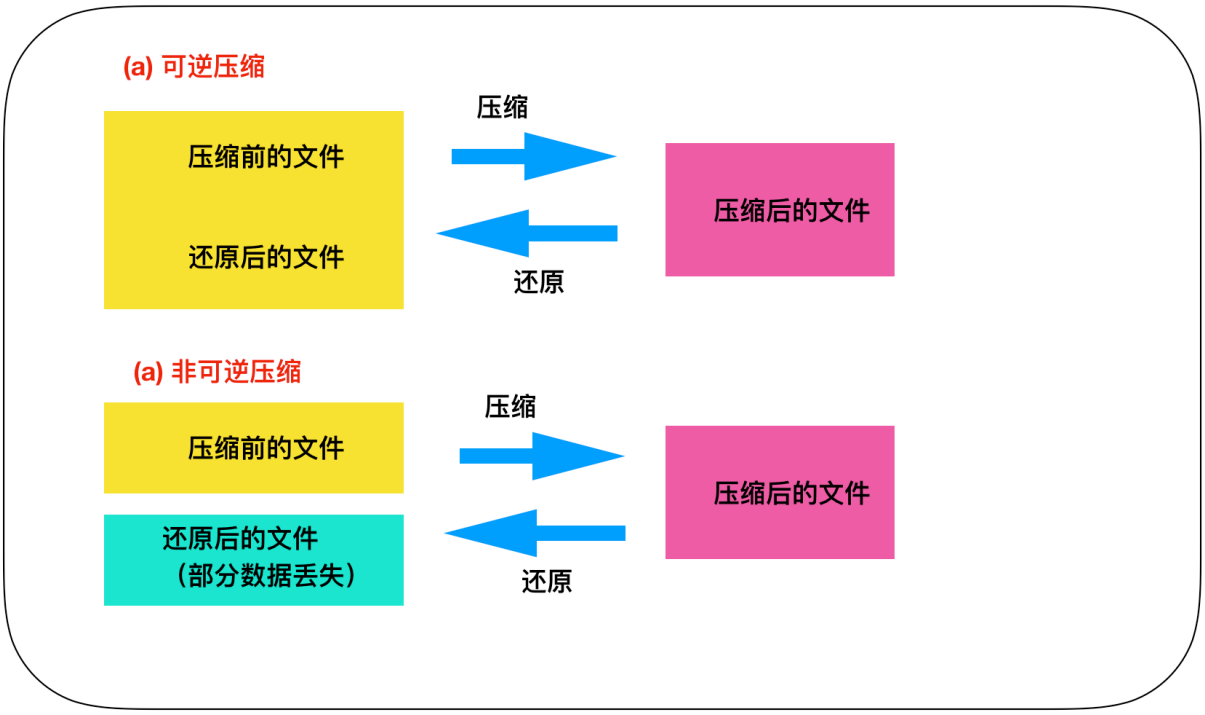
文件类型	压缩前	压缩后	压缩比率
文本文件	14862字节	4119字节	28%
图像文件	96062字节	9456字节	10%
EXE文件	24576字节	4652字节	19%

可逆压缩和非可逆压缩

最后，我们来看一下图像文件的数据形式。图像文件的使用目的通常是把图像数据输出到显示器、打印机等设备上。常用的图像格式有：`BMP`、`JPEG`、`TIFF`、`GIF` 格式等。

- BMP： 是使用 Windows 自带的画笔来做成的一种图像形式
- JPEG：是数码相机等常用的一种图像数据形式
- TIFF: 是一种通过在文件中包含"标签"就能够快速显示出数据性质的图像形式
- GIF： 是由美国开发的一种数据形式，要求色数不超过 256个

图像文件可以使用前面介绍的 RLE 算法和哈夫曼算法，因为图像文件在多数情况下并不要求数据需要还原到和压缩之前一摸一样的状态，允许丢失一部分数据。我们把能还原到压缩前状态的压缩称为 `可逆压缩`，无法还原到压缩前状态的压缩称为 `非可逆压缩` 。



一般来说，JPEG格式的文件是非可逆压缩，因此还原后有部分图像信息比较模糊。GIF 是可逆压缩

我们大家知道，计算机的五大基础部件是 `存储器`、`控制器`、`运算器`、`输入和输出设备`，其中从存储功能的角度来看，可以把存储器分为 `内存` 和 `磁盘`，内存我们上面已经介绍过了，那么下面我们来介绍一下磁盘以及内存和磁盘的关系。

认识磁盘

首先，磁盘和内存都具有存储功能，它们都是存储设备。区别在于，内存是通过 `电流` 来实现存储；磁盘则是通过 `磁记录技术` 来实现存储。内存是一种高速，造假昂贵的存储设备；而磁盘则是速度较慢、造假低廉的存储设备；电脑断电后，内存中的数据会丢失，而磁盘中的数据可以长久保留。内存是属于 `内部存储设备`，硬盘是属于 `外部存储设备`。一般在我们的计算机中，磁盘和内存是相互配合共同作业的。

一般内存指的就是主存（负责存储CPU中运行的程序和数据）；早起的磁盘指的是软磁盘（soft disk，简称软盘），就是下面这个

早期的磁盘 --- 软盘

（2000年的时候我曾经我姑姑家最早的计算机中见到过这个，当时还不知道这是啥，现在知道了。）

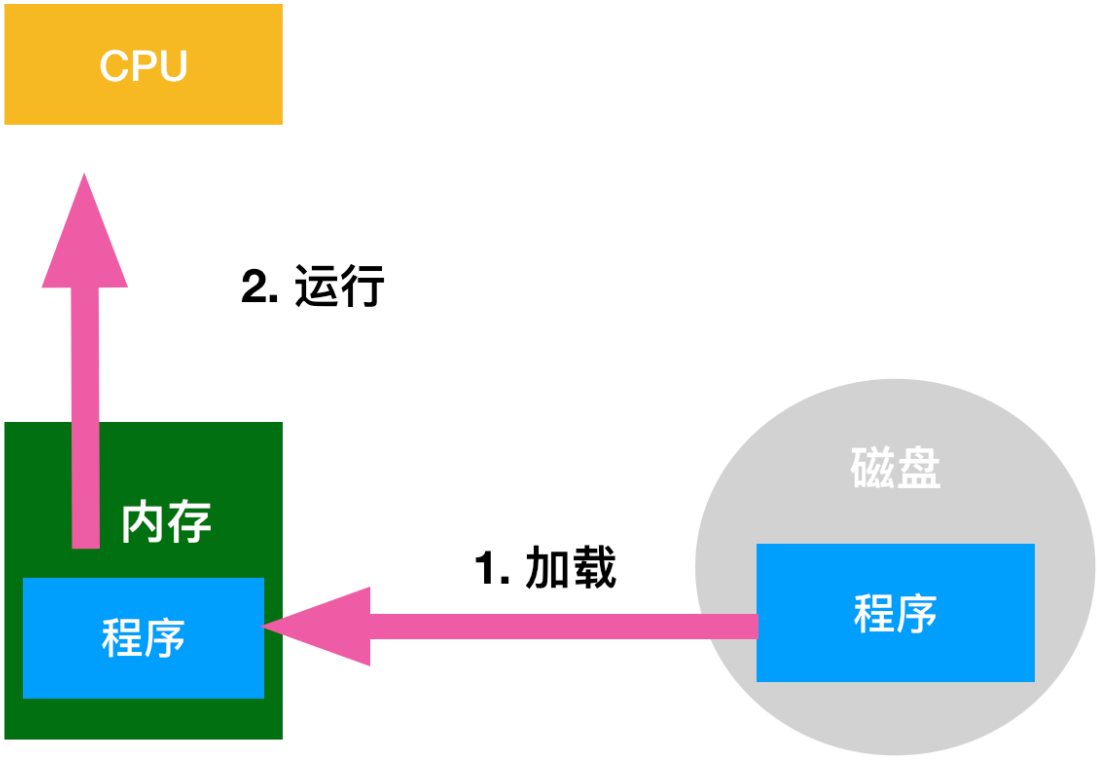
如今常用的磁盘是硬磁盘（hard disk，简称硬盘），就是下面这个

现阶段的磁盘 --- 硬盘

程序不读入内存就无法运行

在了解磁盘前，还需要了解一下内存的运行机制是怎样的，我们的程序被保存在存储设备中，通过使用 CPU 读入来实现程序指令的执行。这种机制称为 存储程序方式 ，现在看来这种方式是理所当然的，但在以前程序的运行都是通过改变计算机的布线来读写指令的。

计算机最主要的存储部件是内存和磁盘。**磁盘中存储的程序必须加载到内存中才能运行**，在磁盘中保存的程序是无法直接运行的，这是因为负责解析和运行程序内容的 CPU 是需要通过程序计数器来指定内存地址从而读出程序指令的。



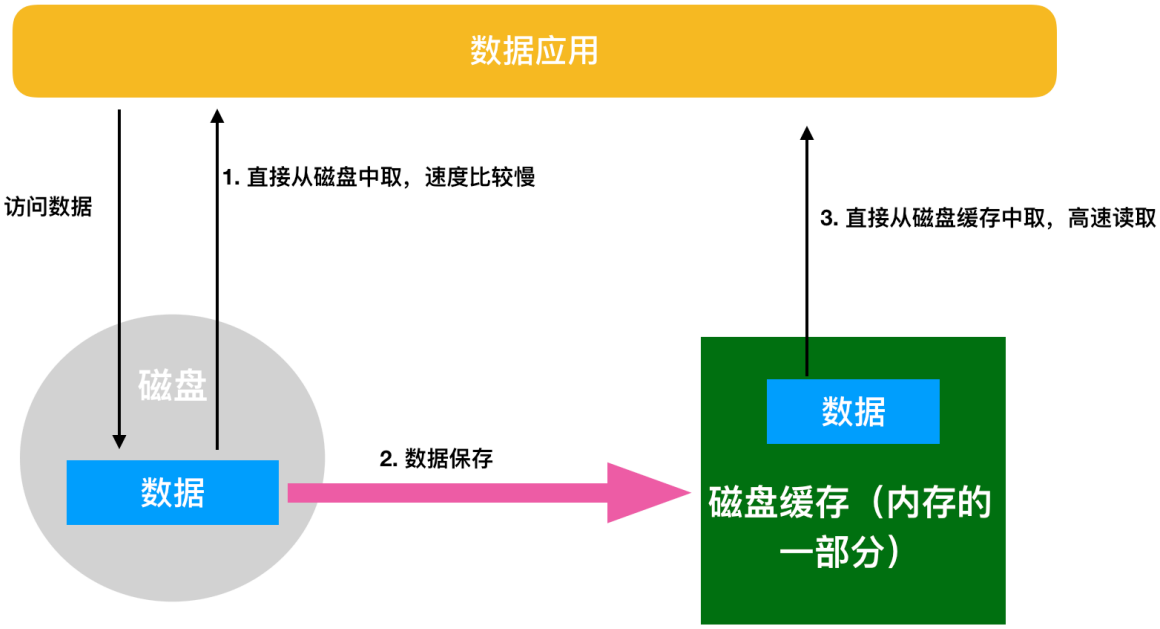
程序运行示意图

磁盘构件

磁盘缓存

我们上面提到，磁盘往往和内存是互利共生的关系，相互协作，彼此持有良好的合作关系。每次内存都需要从磁盘中读取数据，必然会读到相同的内容，所以一定会有一个角色负责存储我们经常需要读到的内容。 我们大家做软件的时候经常会用到 `缓存技术` ，那么硬件层面也不例外，磁盘也有缓存，磁盘的缓存叫做 `磁盘缓存` 。

磁盘缓存指的是把从磁盘中读出的数据存储到内存的方式，这样一来，当接下来需要读取相同的内容时，就不会再通过实际的磁盘，而是通过磁盘缓存来读取。某一种技术或者框架的出现势必要解决某种问题的，那么磁盘缓存就大大**改善了磁盘访问的速度**。



使用磁盘缓存读取数据

Windows 操作系统提供了磁盘缓存技术，不过，对于大部分用户来说是感受不到磁盘缓存的，并且随着计算机的演进，对硬盘的访问速度也在不断演进，实际上磁盘缓存到 Windows 95/98 就已经不怎么使用了。

把低速设备的数据保存在高速设备中，需要时可以直接将其从高速设备中读出，这种缓存方式在web中应用比较广泛，web 浏览器是通过网络来获取远程 web 服务器的数据并将其显示出来。因此，在读取较大的图片的时候，会耗费不少时间，这时 web 浏览器可以把获取的数据保存在磁盘中，然后根据需要显示数据，再次读取的时候就不用重新加载了。

虚拟内存

`虚拟内存` 是内存和磁盘交互的第二个媒介。虚拟内存是指把磁盘的一部分作为 `假想内存` 来使用。这与磁盘缓存是假想的磁盘（实际上是内存）相对，虚拟内存是假想的内存（实际上是磁盘）。

虚拟内存是计算机系统内存管理的一种技术。它使得应用程序认为它拥有 `连续可用` 的内存（一个完整的地址空间），但是实际上，它通常被分割成多个物理碎片，还有部分存储在外部磁盘管理器上，必要时进行数据交换。

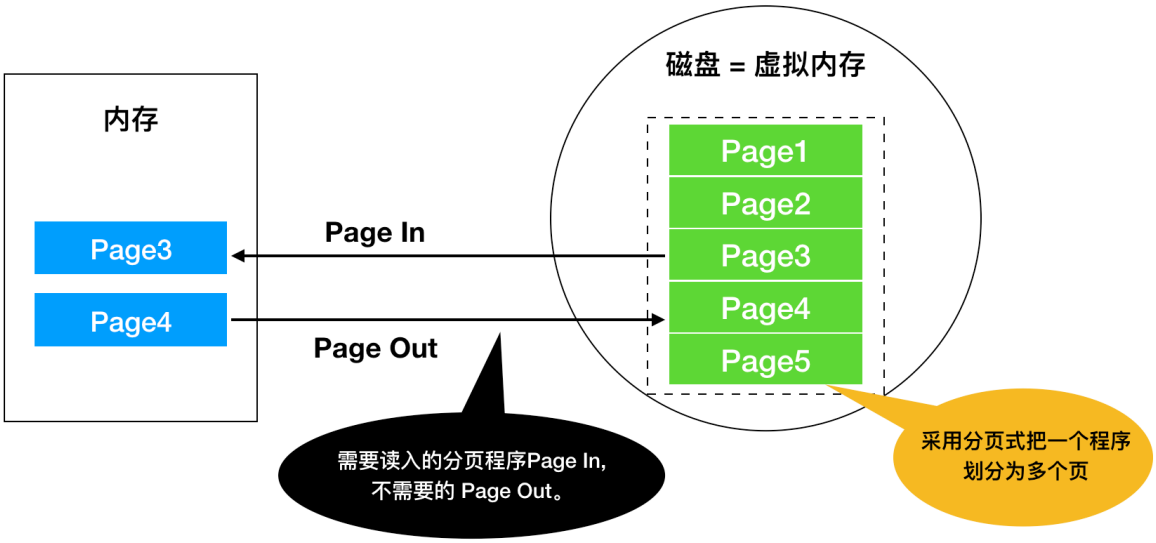
计算机中的程序都要通过内存来运行，如果程序占用内存很大，就会将内存空间消耗殆尽。为了解决这个问题，WINDOWS 操作系统运用了虚拟内存技术，通过拿出一部分硬盘来当作内存使用，来保证程序耗尽内存仍然有可以存储的空间。虚拟内存在硬盘上的存在形式就是 `PAGEFILE.SYS` 这个页面文件。

通过借助虚拟内存，在内存不足时仍然可以运行程序。例如，在只剩 5MB 内存空间的情况下仍然可以运行 10MB 的程序。由于 CPU 只能执行加载到内存中的程序，因此，虚拟内存的空间就需要和内存中的空间进行 `置换（swap）`，然后运行程序。

虚拟内存与内存的交换方式

刚才我们提到虚拟内存需要和内存中的部分内容做置换才可让 CPU 继续执行程序，那么做置换的方式是怎样的呢？又分为哪几种方式呢？

虚拟内存的方法有 `分页式` 和 `分段式` 两种。Windows 采用的是分页式。该方式是指在不考虑程序构造的情况下，把运行的程序按照一定大小的页进行分割，并以 `页` 为单位进行置换。在分页式中，我们把磁盘的内容读到内存中称为 `Page In`，把内存的内容写入磁盘称为 `Page Out`。Windows 计算机的页大小为 4KB，也就是说，需要把应用程序按照 4KB 的页来进行切分，以页（page）为单位放到磁盘中，然后进行置换。



程序的 Page In 和 Page Out

为了实现内存功能，Windows 在磁盘上提供了虚拟内存使用的文件（page file，页文件）。该文件由 Windows 生成和管理，文件的大小和虚拟内存大小相同，通常大小是内存的 1 - 2 倍。

节约内存

Windows 是以图形界面为基础的操作系统。它的前身是 `MS-DOC`，最初的版本可以在 128kb 的内存上运行程序，但是现在想要 Windows 运行流畅的花至少要需要 512MB 的内存，但通常往往是不够的。

也许许多人认为可以使用虚拟内存来解决内存不足的情况，而虚拟内存确实能够在内存不足的时候提供补充，但是使用虚拟内存的 Page In 和 Page Out 通常伴随着低速的磁盘访问，这是一种得不偿失的表现。所以虚拟内存无法从根本上解决内存不足的情况。

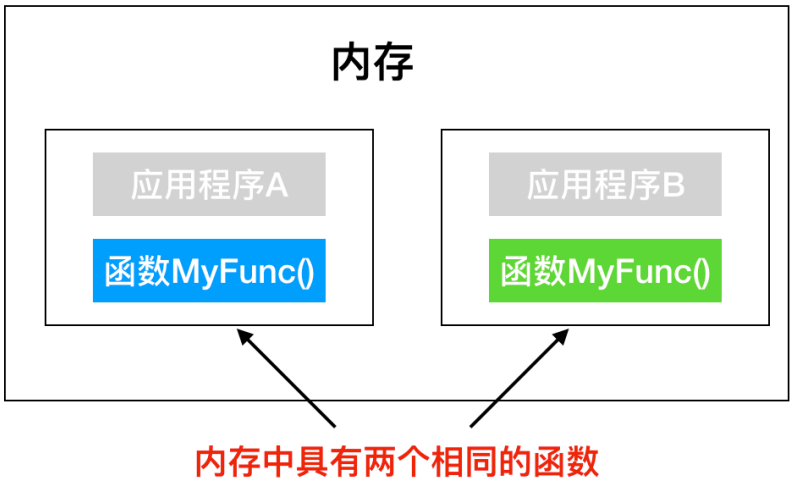
为了从根本上解决内存不足的情况，要么是增加内存的容量，加内存条；要么是优化应用程序，使其尽可能变小。第一种建议往往需要衡量口袋的银子，所以我们只关注第二种情况。

> 注意：以下的篇幅会涉及到 C 语言的介绍，是每个程序员（不限语言）都需要知道和了解的知识。

通过 DLL 文件实现函数共有

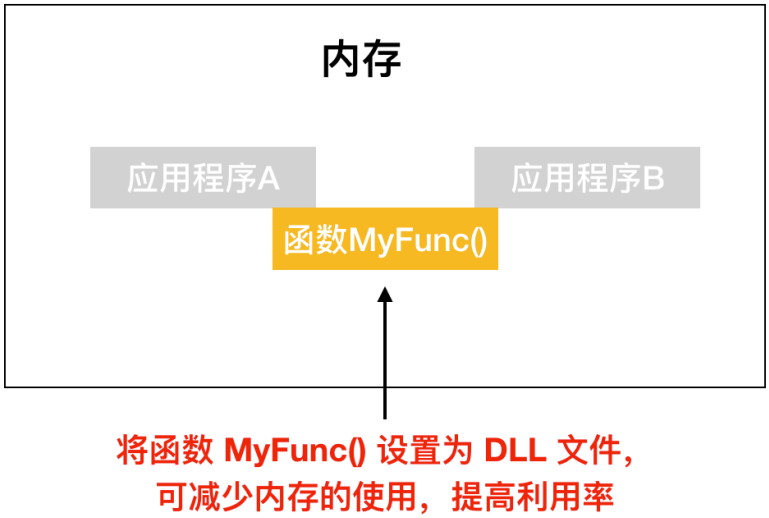
`DLL（Dynamic Link Library）` 文件，是一种 `动态链接库` 文件，顾名思义，是在程序运行时可以动态加载 `Library（函数和数据的集合）` 的文件。此外，多个应用可以共有同一个 DLL 文件。而通过共有一个 DLL 文件则可以达到节约内存的效果。

例如，假设我们编写了一个具有某些处理功能的函数 `MyFunc()` 。应用 A 和 应用 B 都需要用到这个函数，然后在各自的应用程序中内置 MyFunc()（这个称为Static Link，静态链接）后同时运行两个应用，内存中就存在了同一个函数的两个程序，这会造成资源浪费。



静态链接导致内存利用率下降

为了改变这一点，使用 DLL 文件而不是应用程序的执行文件（EXE文件）。因为同一个 DLL 文件内容在运行时可以被多个应用共有，因此内存中存在函数 MyFunc（）的程序就只有一个



使用动态链接DDL改善内存使用

Windows 操作系统其实就是无数个 DLL 文件的集合体。有些应用在安装时，DLL文件也会被追加。应用程序通过这些 DLL 文件来运行，既可以节约内存，也可以在不升级 EXE 文件的情况下，通过升级 DLL 文件就可以完成更新。

通过调用 _stdcall 来减少程序文件的大小

通过调用 `_stdcall` 来减小程序文件的方法，是用 C 语言编写应用时可以利用的高级技巧。我们来认识一下什么是 _stdcall。

_stdcall 是 `standard call(标准调用)` 的缩写。Windows 提供的 DLL 文件内的函数，基本上都是通过 _stdcall 调用方式来完成的，这主要是为了节约内存。另一方面，用 C 语言编写的程序默认都不是 _stdcall 。C 语言特有的调用方式称为 `C 调用` 。C 语言默认不使用 _stdcall 的原因是因为 C 语言所对应的函数传入参数是可变的，只有函数调用方才能知道到底有多少个参数，在这种情况下，栈的清理作业便无法进行。不过，在 C 语言中，如果函数的参数和数量固定的话，指定 _stdcall 是没有任何问题的。

> C 语言和 Java 最主要的区别之一在于 C 语言需要人为控制释放内存空间

C 语言中，在调用函数后，需要人为执行栈清理指令。把不需要的数据从接收和传递函数的参数时使用的内存上的栈区域中清理出去的操作叫做 `栈清理处理` 。

例如如下代码

```
// 函数调用方
void main(){
  int a;
  a = MyFunc(123,456);
}

// 被调用方
int MyFunc(int a,int b){
  ...
}
```

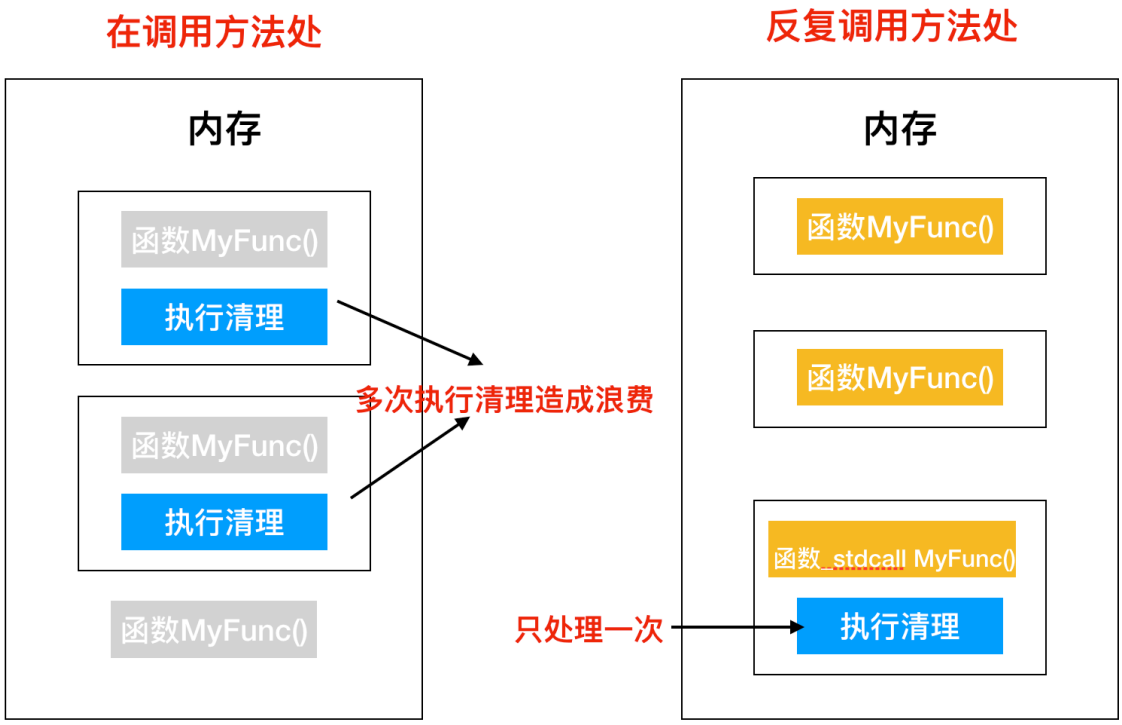
代码中，从 main 主函数调用到 MyFunc() 方法，按照默认的设定，栈的清理处理会附加在 main 主函数这一方。在同一个程序中，有可能会多次调用，导致 MyFunc() 会进行多次清理，这就会造成内存的浪费。

汇编之后的代码如下

```
push 1C8h                          // 将参数 456( = 1C8h) 存入栈中
push 7Bh                           // 将参数 123( = 7Bh) 存入栈中
call @LTD+15 (MyFunc)(00401014)    // 调用 MyFunc 函数
add esp,8                          // 运行栈清理
```

C 语言通过栈来传递函数的参数，使用 `push` 是往栈中存入数据的指令， `pop` 是从栈中取出数据的指令。32 位 CPU 中，1次 push 指令可以存储 4 个字节（32 位）的数据。上述代码由于进行了两次 push 操作，所以存储了 8 字节的数据。通过 `call` 指令来调用函数，调用完成后，栈中存储的数据就不再需要了。于是就通过 add esp,8 这个指令，使存储着栈数据的 esp 寄存器前进 8 位（设定为指向高 8 位字节的地址），来进行数据清理。由于栈是在各种情况下都可以利用的内存领域，因此使用完毕后有必要将其恢复到原始状态。上述操作就是执行栈的清理工作。另外，在 C 语言中，函数的返回值，是通过寄存器而非栈来返回的。

栈执行清理工作，在调用方法处执行清理工作和在反复调用方法处执行清理工作不同，使用 `_stdcall` 标准调用的方式称为反复调用方法，在这种情况下执行栈清理开销比较小。

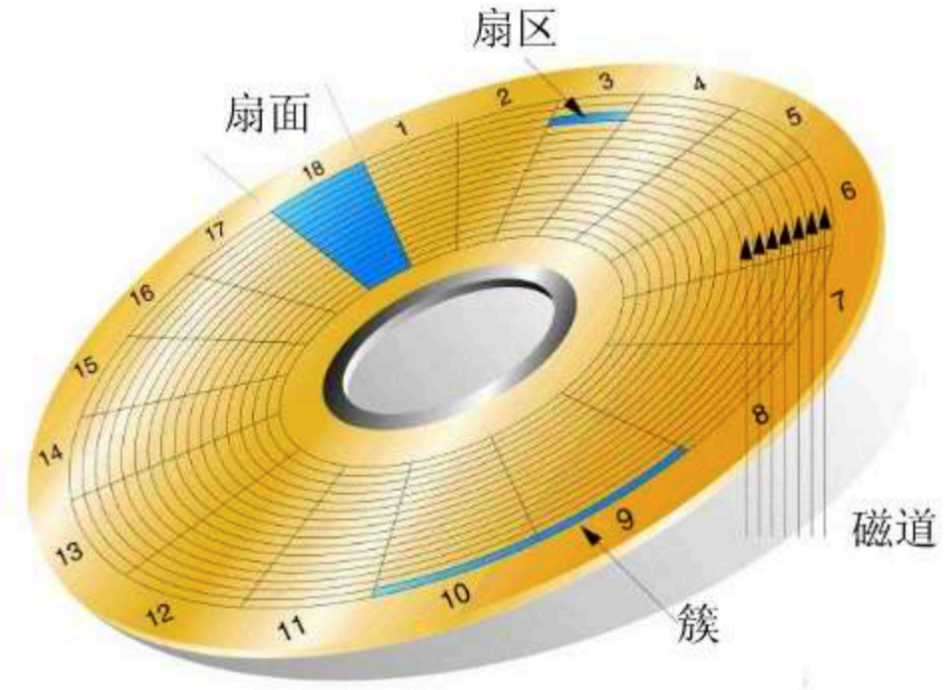


不同方式的调用清理方式也不同

磁盘的物理结构

之前我们介绍了CPU、内存的物理结构，现在我们来介绍一下磁盘的物理结构。**磁盘的物理结构指的是磁盘存储数据的形式**。

磁盘是通过其物理表面划分成多个空间来使用的。划分的方式有两种：`可变长方式` 和 `扇区方式` 。前者是将物理结构划分成长度可变的空间，后者是将磁盘结构划分为固定长度的空间。一般 Windows 所使用的硬盘和软盘都是使用扇区这种方式。扇区中，把磁盘表面分成若干个同心圆的空间就是 `磁道` ，把磁道按照固定大小的存储空间划分而成的就是 `扇区`



磁盘的物理构造

扇区 是对磁盘进行物理读写的最小单位。Windows 中使用的磁盘，一般是一个扇区 512 个字节。不过，Windows 在逻辑方面对磁盘进行读写的单位是扇区整数倍簇。根据磁盘容量不同功能，1簇可以是 512 字节（1 簇 = 1扇区）、1KB（1簇 = 2扇区）、2KB、4KB、8KB、16KB、32KB(1 簇 = 64 扇区)。簇和扇区的大小是相等的。

不管是硬盘还是软盘，不同的文件是不能存储在同一簇中的，否则就会导致只有一方的文件不能删除。所以，不管多小的文件，都会占用 1 簇的空间。这样一来，所有的文件都会占用 1 簇的整数倍的空间。

操作系统环境

程序中包含着 运行环境 这一内容，可以说 **运行环境 = 操作系统 + 硬件** ，操作系统又可以被称为软件，它是由一系列的指令组成的。我们不介绍操作系统，我们主要来介绍一下硬件的识别。

我们肯定都玩儿过游戏，你玩儿游戏前需要干什么？是不是需要先看一下自己的笔记本或者电脑是不是能肝的起游戏？下面是一个游戏的配置（怀念一下 wow）

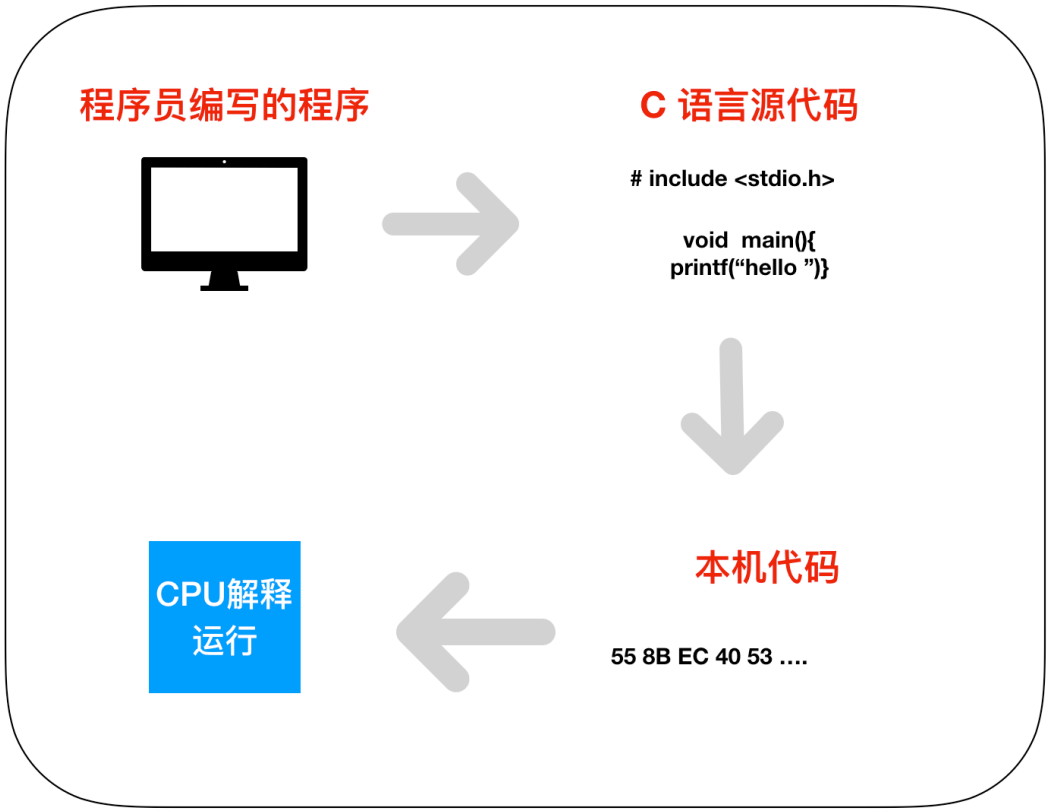
图中的主要配置如下

- 操作系统版本：说的就是应用程序运行在何种系统环境，现在市面上主要有三种操作系统环境，Windows 、Linux 和 Unix ，一般我们玩儿的大型游戏几乎都是在 Windows 上运行，可以说 Windows 是游戏的天堂。Windows 操作系统也会有区分，分为32位操作系统和64位操作系统，互不兼容。
- 处理器：处理器指的就是 CPU，你的电脑的计算能力，通俗来讲就是每秒钟能处理的指令数，如果你的电脑觉得卡带不起来的话，很可能就是 CPU 的计算能力不足导致的。
- 显卡：显卡承担图形的输出任务，因此又被称为图形处理器（Graphic Processing Unit，GPU），显卡也非常重要，比如我之前玩儿的 剑灵 开五档（其实就是图像变得更清晰）会卡，其实就是显卡显示不出来的原因。
- 内存：内存即主存，就是你的应用程序在运行时能够动态分析指令的这部分存储空间，它的大小也能决定你电脑的运行速度。

- 存储空间：存储空间指的就是应用程序安装所占用的磁盘空间，由图中可知，此游戏的最低存储空间必须要大于 5GB，其实我们都会遗留很大一部分用来安装游戏。

从程序的运行环境这一角度来考量的话，CPU 的种类是特别重要的参数，为了使程序能够正常运行，必须满足 CPU 所需的最低配置。

CPU 只能解释其自身固有的语言。不同的 CPU 能解释的机器语言的种类也是不同的。机器语言的程序称为 `本地代码(native code)`，程序员用 C 等高级语言编写的程序，仅仅是文本文件。`文本文件(排除文字编码的问题)` 在任何环境下都能显示和编辑。我们称之为 `源代码` 。通过对源代码进行编译，就可以得到 `本地代码` 。下图反映了这个过程。

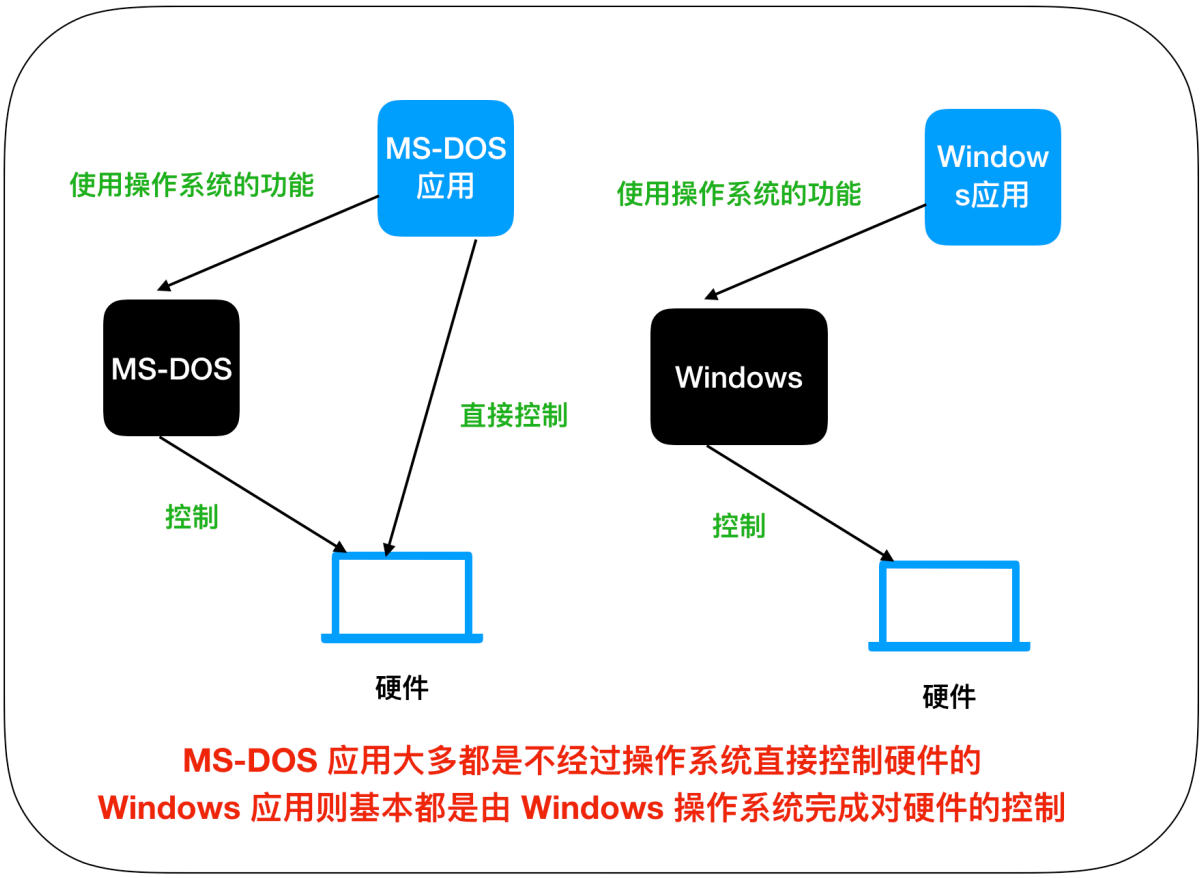


CPU 负责解析并运行从源代码编译而来的本地代码

Windows 操作系统克服了CPU以外的硬件差异

计算机的硬件并不仅仅是由 CPU 组成的，还包括用于存储程序指令的数据和内存，以及通过 I/O 连接的键盘、显示器、硬盘、打印机等外围设备。

在 WIndows 软件中，键盘输入、显示器输出等并不是直接向硬件发送指令。而是通过向 Windows 发送指令实现的。因此，程序员就不用注意内存和 I/O 地址的不同构成了。Windows 操作的是硬件而不是软件，软件通过操作 Windows 系统可以达到控制硬件的目的。



不同操作系统的 API 差异性

接下来我们看一下操作系统的种类。同样机型的计算机，可安装的操作系统类型也会有多种选择。例如：AT 兼容机除了可以安装 Windows 之外，还可以采用 Unix 系列的 Linux 以及 FreeBSD （也是一种Unix操作系统）等多个操作系统。当然，应用软件则必须根据不同的操作系统类型来专门开发。**CPU 的类型不同，所对应机器的语言也不同**,同样的道理，**操作系统的类型不同，应用程序向操作系统传递指令的途径也不同**。

应用程序向系统传递指令的途径称为 `API(Application Programming Interface)` 。Windows 以及 Linux 操作系统的 API，提供了任何应用程序都可以利用的函数组合。因为不同操作系统的 API 是有差异的。所以，如何要将同样的应用程序移植到另外的操作系统，就必须要覆盖应用所用到的 API 部分。

键盘输入、鼠标输入、显示器输出、文件输入和输出等同外围设备进行交互的功能，都是通过 API 提供的。

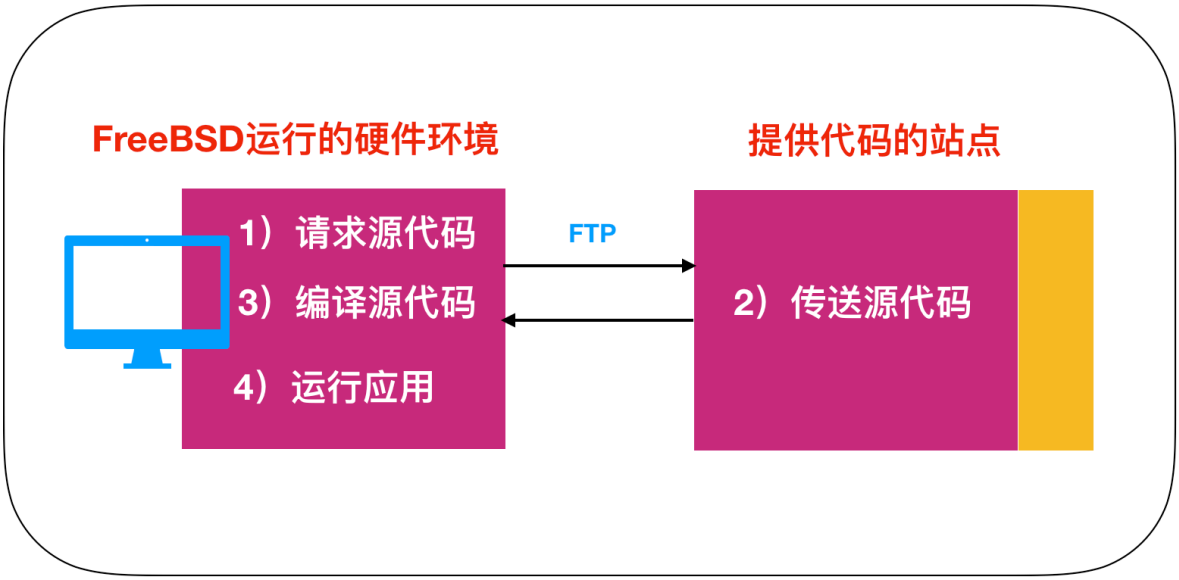
这也就是为什么 Windows 应用程序不能直接移植到 Linux 操作系统上的原因，API 差异太大了。

在同类型的操作系统下，不论硬件如何，API 几乎相同。但是，由于不同种类 CPU 的机器语言不同，因此本地代码也不尽相同。

FreeBSD Port 帮你轻松使用源代码

不知道你有没有这个想法：“既然 CPU 不同会导致本地代码不同，那为何不将源代码直接发送给程序呢？”这确实是一种解决办法，Unix 系列的 `FreeBSD` 操作系统就使用了这种方式。

Unix 系列操作系统 FreeBSD 中，存在一种名为 `Ports` 的机制。该机制能够结合当前运行环境的硬件环境来编译应用的源代码，进而得到可以运行的本地代码。如果目标应用的源代码在硬件上找不到，Ports 就会自动使用 FTP 连接到相应站点下载代码。



FreeBSD 的 Ports 机制

全球有很多站点都提供适用于 FreeBSD 的应用源代码。通过使用 Ports 可以利用的程序源代码，大约有 16000 种。根据不同的领域进行分类，可以随时使用。

FreeBSD 上应用的源代码，大部分是用 C 语言来标注的， `C 编译器` 可以结合 FreeBSD 的运行环境来生成合适的本地代码。

> FTP(File Transfer Protocol) 是连接到互联网上的计算机之间的传送文件的协议。

可以使用虚拟机获取其他环境

即使不通过应用程序的移植，在同一个操作系统上仍然可以使用其他的操作系统，那就是使用 `虚拟机软件` 。虚拟机（Virtual Machine）指通过软件的具有完整硬件系统功能的、运行在一个完全隔离环境中的完整计算机系统。在实体计算机中能够完成的工作在虚拟机中都能够实现。

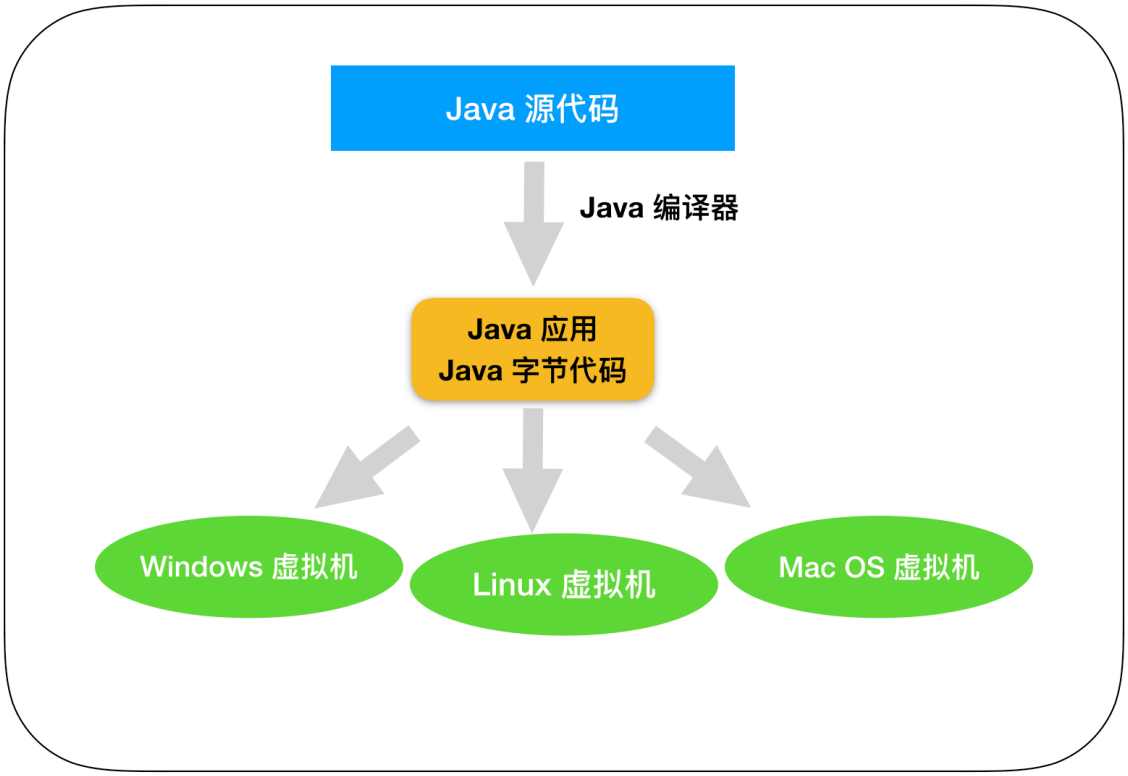
提供相同运行环境的 Java 虚拟机

总算是提到大 Java 了， Java 大法好，除了虚拟机的方法之外，还有一种方法能够提供不依赖于特定硬件和操作系统的程序运行环境，那就是 Java。

大家说的 Java 其实有两层意思，一种是作为编程语言的 Java；一种是作为程序运行环境的 Java。Java 与其他语言相同，都是通过源代码编译后运行的。不过，编译后生成的不是特定 CPU 使用的本地代码，而是名为 `字节代码` 的程序。直接代码的运行环境就称为 `Java 虚拟机(Java Virtual Machine)` 。Java 虚拟机是一边把 Java 字节代码逐一转换为本地代码一边在运行着。

程序运行时，将编译后的字节代码转换为本地代码，这样的操作看上去有些迂回，但由此可以实现相同的字节码可以在不同的操作系统环境下运行。

想象一下，你开发完成的应用部署到 Linux 环境下，是不是什么都不用管?



Java 应用在虚拟机上运行

Windows 有专门的 Windows 虚拟机，Macintosh 有 Macintosh 专门的虚拟机。从操作系统来看，Java虚拟机就是一个应用，从运行环境上来看，Java 虚拟机就是运行环境。

BIOS 和引导

最后对一些比较基础的部分做一些补充说明。程序的运行环境，存在着名为 `BIOS(Basic Input/Output System)` 的系统。BIOS 存储在 ROM 中，是预先内置在计算机主机内部的程序。BIOS 除了键盘、磁盘和显卡等基本控制外，还有 `引导程序` 的功能。引导程序是存储在启动驱动器启示区域的小程序。操作系统的启动驱动器一般硬盘。不过有时也可能是 `CD-ROM` 或软盘。

电脑开机后，BIOS 会确认硬件是否正常运行，没有异常的话会直接启动引导程序。引导程序的功能是把在硬盘等记录的 OS 加载到内存中运行。虽然启动应用是 OS 的功能，但 OS 不能启动自己，是通过引导程序来启动的。

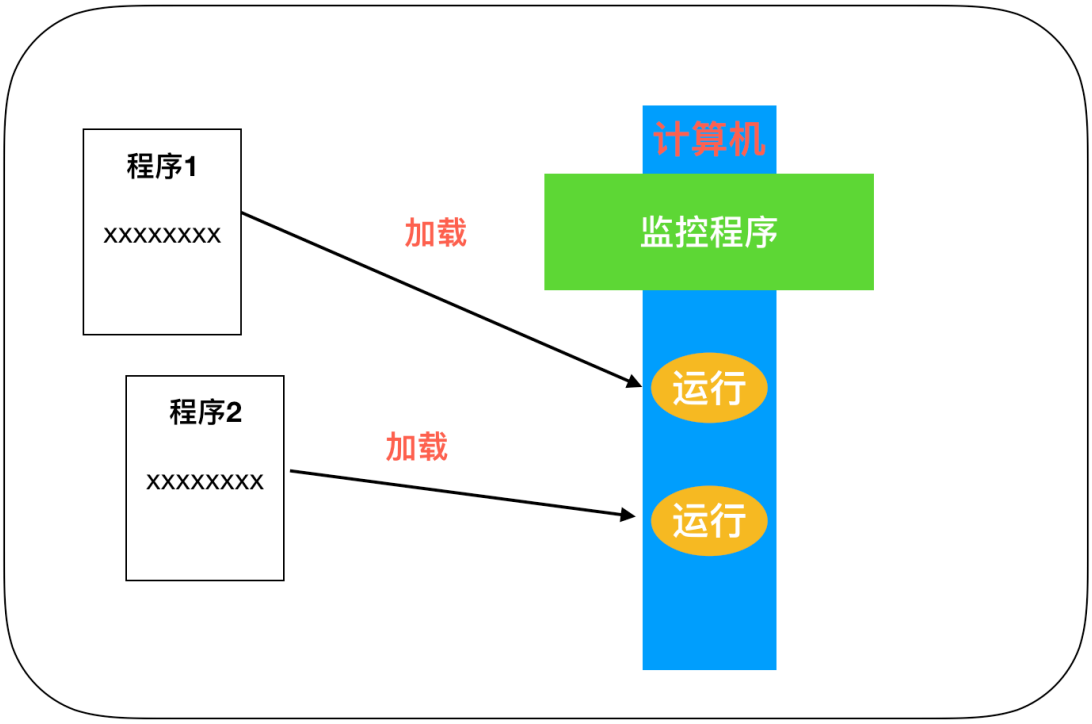
利用计算机运行程序大部分都是为了提高处理效率。例如，Microsoft Word 这样的文字处理软件，是用来提高文本文件处理效率的程序，Microsoft Excel 等表格计算软件，是用来提高账本处理效率的程序。这种为了提高特定处理效率的程序统称为 `应用`

程序员的工作就是编写各种各样的应用来提高工作效率，程序员一般不编写操作系统，但是程序员编写的应用离不开操作系统，下面我们就针对 Windows 操作系统来说明一下操作系统和应用之间的关系。

操作系统功能的历史

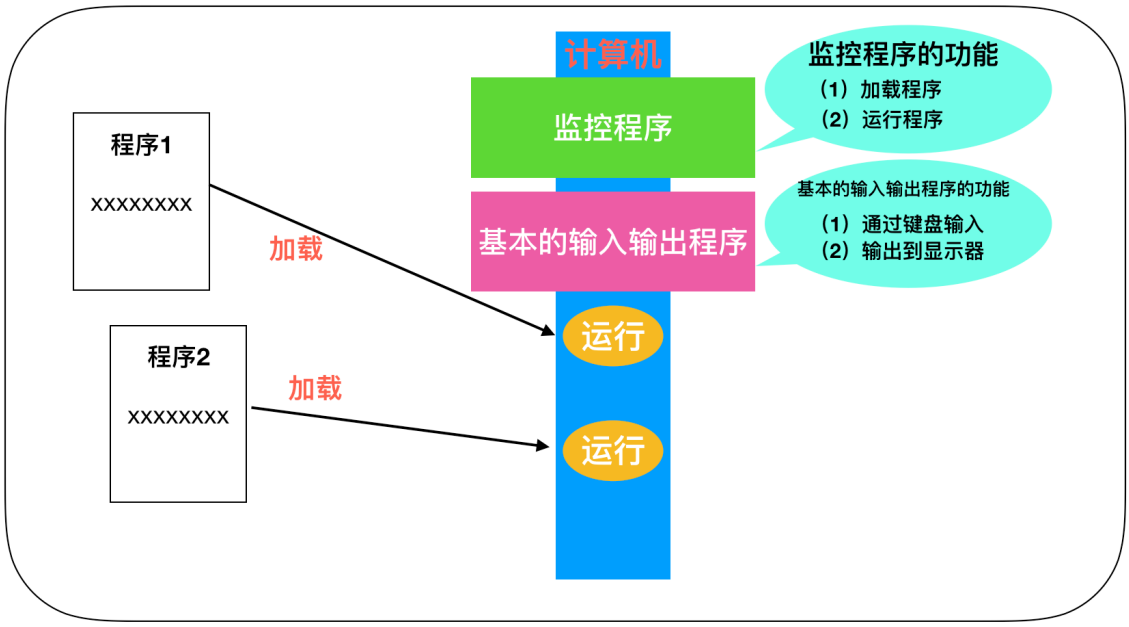
操作系统 其实也是一种软件，任何新事物的出现肯定都有它的历史背景，那么操作系统也不是凭空出现的，肯定有它的历史背景。

在计算机尚不存在操作系统的年代，完全没有任何程序，人们通过各种 按钮 来控制计算机，这一过程非常麻烦。于是，有人开发出了仅具有加载和运行功能的 监控程序 ，这就是操作系统的原型。通过事先启动监控程序，程序员可以根据需要将各种程序加载到内存中运行。虽然仍旧比较麻烦，但比起在没有任何程序的状态下进行开发，工作量得到了很大的缓解。



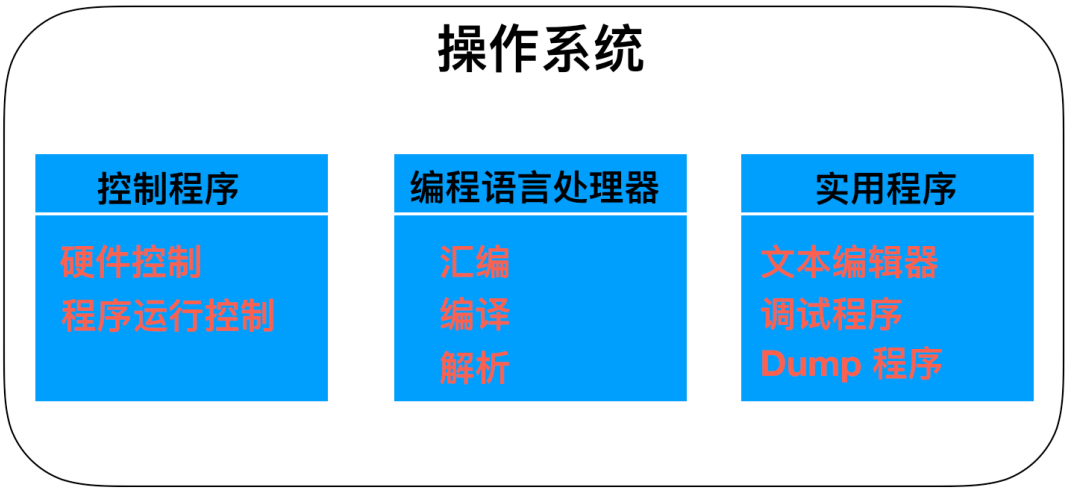
监控程序可以说是操作系统的原型

随着时代的发展，人们在利用监控程序编写程序的过程中发现很多程序都有公共的部分。例如，通过键盘进行文字输入，显示器进行数据展示等，如果每编写一个新的应用程序都需要相同的处理的话，那真是太浪费时间了。因此，基本的输入输出部分的程序就被追加到了监控程序中。初期的操作系统就是这样诞生了。



初期的操作系统 = 监控程序 + 基本的输入输出程序

类似的想法可以共用，人们又发现有更多的应用程序可以追加到监控程序中，比如 `硬件控制程序` ， `编程语言处理器(汇编、编译、解析)` 以及各种应用程序等，结果就形成了和现在差异不大的操作系统，也就是说，其实操作系统是多个程序的集合体。



操作系统多个程序的集合体

> 汇编语言是一种低级语言，也被称为 `符号语言` 。汇编语言是第二代计算机语言，在汇编语言中，用助记符代替机器指令的操作码，用地址符号或标号代替指令或操作数的地址。用一些容易理解和记忆的字母，单词来代替一个特定的指令，比如：用 `ADD` 代表数字逻辑上的加减， `MOV` 代表数据传递等等，通过这种方法，人们很容易去阅读已经完成的程序或者理解程序正在执行的功能，对现有程序的bug修复以及运营维护都变得更加简单方便

可以说共用思想真是人类前进的一大步，对于解放生产力而言简直是太重要了。

要把操作系统放在第一位

对于程序员来说，程序员创造的不是硬件，而是各种应用程序，但是如果程序员只做应用不懂硬件层面的知识的话，是无法成为硬核程序员的。**现在培训机构培养出了一批怎么用的人才，却没有培训出为什么这么做的人才**，毕竟 为什么 不是培训机构教的，而是学校教的，我很相信耗子叔说的话：学习没有速成这回事。言归正题。

在操作系统诞生之后，程序员不需要在硬件层面考虑问题，所以程序员的数量就增加了。哪怕自称 对硬件一窍不通 的人也可能制作出一个有模有样的程序。不过，要想成为一个全面的程序员，有一点需要清楚的就是，掌握硬件的基本知识，并借助操作系统进行抽象化，可以大大提高编程效率。

下面就看一下操作系统是如何给开发人员带来便利的，在 Windows 操作系统下，用 C 语言制作一个具有表示当前时间功能的应用。 `time()` 是用来取得当前日期和时间的函数， `printf()` 是把结果打印到显示器上的函数，如下：

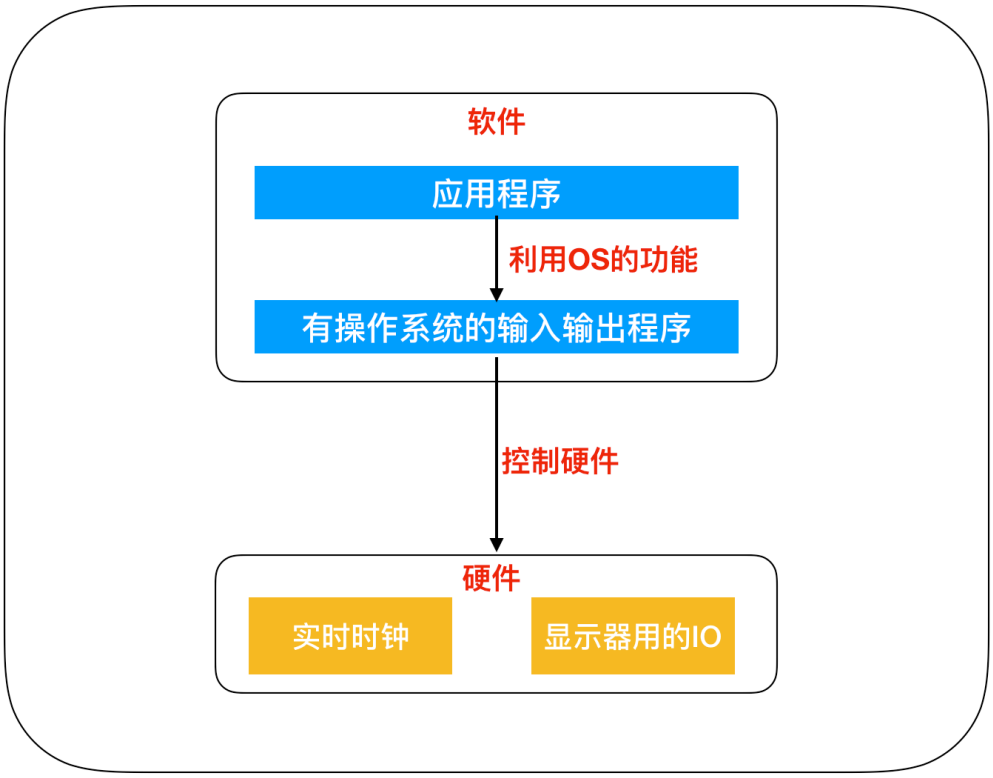
```
#include <stdio.h>
#include <time.h>

void main(){
  // 保存当前日期和时间信息
  time_t tm;

  // 取得当前的日期和时间
  time(&tm);

  // 在显示器上显示日期和时间
  printf("%s\n", ctime(&tm));
}
```

读者可以自行运行程序查看结果，我们主要关注硬件在这段代码中做了什么事情

- 通过 time_t tm，为 time_t 类型的变量申请分配内存空间；
- 通过 time(&tm) ，将当前的日期和时间数据保存到变量的内存空间中
- 通过 printf("%s\n"，ctime(&tm))， 把变量内存空间的内容输出到显示器上。

应用的可执行文件指的是，计算机的 CPU 可以直接解释并运行的本地代码，不过这些代码是无法直接控制硬件的，事实上，这些代码是通过操作系统来间接控制硬件的。变量中涉及到的内存分配情况，以及 time() 和 printf() 这些函数的运行结果，都不是面向硬件而是面向 操作系统 的。操作系统收到应用发出的指令后，首先会对该指令进行解释，然后会对 时钟IC 和显示器用的 I/O 进行控制。

> 计算机中都安装有保存日期和时间的实时时钟(Real-time clock)，上面提到的时钟IC 就是值该实时时钟。

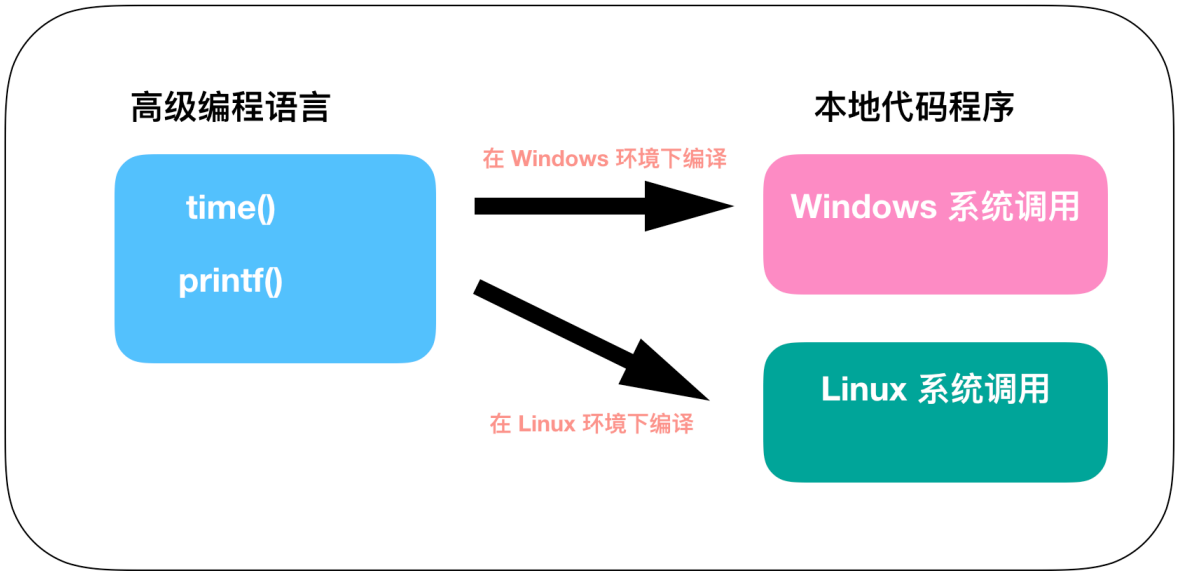


应用程序通过 OS 调度硬件

系统调用和编程语言的移植性

操作系统控制硬件的功能，都是通过一些小的函数集合体的形式来提供的。这些函数以及调用函数的行为称为 `系统调用`，也就是通过应用进而调用操作系统的意思。在前面的程序中用到了 `time()` 以及 `printf()` 函数，这些函数内部也封装了系统调用。

C 语言等高级编程语言并不依存于特定的操作系统，这是因为人们希望不管是 `Windows` 操作系统还是 `Linux` 操作系统都能够使用相同的源代码。因此，高级编程语言的机制就是，使用独自的函数名，然后在编译的时候将其转换为系统调用的方式（也有可能是多个系统调用的组合）。也就是说，**高级语言编写的应用在编译后，就转换成了利用系统调用的本地代码**。



高级语言的函数调用在编译后变为了系统调用

不过，在高级语言中也存在直接调用系统调用的编程语言，不过，利用这种方式做成应用，移植性并不友好。

> 移植性：移植性指的是同样的程序在不同操作系统下运行时所花费的时间，时间越少证明移植性越好。

操作系统和高级编程语言使硬件抽象化

通过使用操作系统提供的系统调用，程序员不必直接编写控制硬件的程序，而且，通过使用高级编程语言，有时也无需考虑系统调用的存在，系统调用往往是自动触发的，操作系统和高级编程语言能够使硬件抽象化，这很了不起。

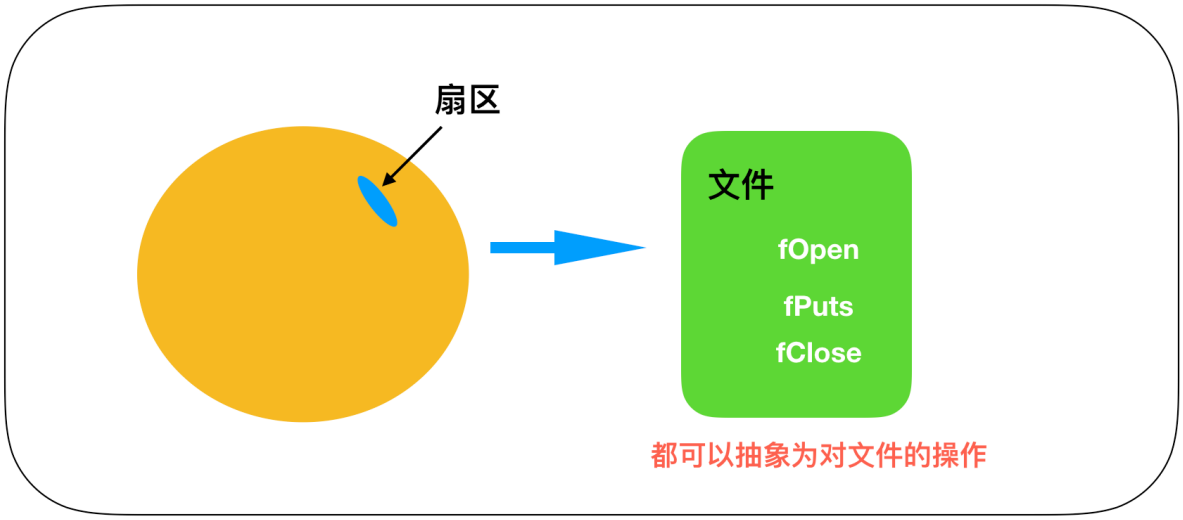
下面让我们看一个硬件抽象化的具体实例

```c
#include <stdio.h>

void main(){

  // 打开文件
  FILE *fp = fopen("MyFile.txt","w");

  // 写入文件
  fputs("你好", fp);

  // 关闭文件
  fclose(fp);
}
```

上述代码使用 C 编写的程序， `fputs()` 是用来往文件中写入字符串的函数， `fclose()` 是用来关闭文件的函数。

磁盘就如同树的年轮，磁盘的读写是以扇区为单位的，通过磁道来寻址，如果直接对硬件读写的话，那么就会变为通过向磁盘用的 I/O 指定扇区位置来对数据进行读写了。

但是，在上面代码中，扇区压根就没有出现过传递给 fopen() 函数的参数，是文件名 `MyFile.txt` 和指定文件写入的 `w` 。传递给 fputs() 的参数，是往文件中写入的字符串"你好" 和 fp，传递给 fclose() 的参数，也仅仅是 fp，也就是说磁盘通过打开文件这个操作，把磁盘抽象化了，打开文件这个操作就可以说是操作硬件的指令。



下面让我们来看一下代码清单中 fp 的功能，变量 fp 中被赋予的是 fopen() 函数的返回值，该值被称为 `文件指针` 。应用打开文件后，操作系统就会自动申请分配用来管理文件读写的内存空间。内存地址可以通过 fopen() 函数的返回值获得。用 fopen() 打开文件后，接下来就是通过制定的文件指针进行操作，正因为如此，fputs() 和 fclose() 以及 fclose() 参数中都制定了文件指针。

由此我们可以得出一个结论，应用程序是通过系统调用，磁盘抽象来实现对硬盘的控制的。

Windows 操作系统的特征

Windows 操作系统是世界上用户数量最庞大的群体，作为 Windows 操作系统的 `资深` 用户，你都知道 Windows 操作系统有哪些特征吗？下面列举了一些 Windows 操作系统的特性

- Windows 操作系统有两个版本：32位和64位
- 通过 `API` 函数集成来提供系统调用
- 提供了采用图形用户界面的用户界面
- 通过 `WYSIWYG` 实现打印输出，WYSIWYG 其实就是 What You See Is What You Get ，值得是显示器上显示的图形和文本都是可以原样输出到打印机打印的。
- 提供多任务功能，即能够同时开启多个任务
- 提供网络功能和数据库功能
- 通过即插即用实现设备驱动的自设定

这些是对程序员来讲比较有意义的一些特征，下面针对这些特征来进行分别的介绍

32位操作系统

这里表示的32位操作系统表示的是**处理效率最高的数据大小**。Windows 处理数据的基本单位是 32 位。这与最一开始在 `MS-DOS` 等16位操作系统不同，因为在16位操作系统中处理32位数据需要两次，而32位操作系统只需要一次就能够处理32位的数据，所以一般在 windows 上的应用，它们的最高能够处理的数据都是 32 位的。

比如，用 C 语言来处理整数数据时，有8位的 `char` 类型，16位的 `short` 类型，以及32位的 `long` 类型三个选项，使用位数较大的 long 类型进行处理的话，增加的只是内存以及磁盘的开销，对性能影响不大。

现在市面上大部分都是64位操作系统了，64位操作系统也是如此。

通过 API 函数集来提供系统调用

Windows 是通过名为 `API` 的函数集来提供系统调用的。API是联系应用程序和操作系统之间的接口，全称叫做 `Application Programming Interface`,应用程序接口。

当前主流的32位版 Windows API 也称为 `Win32 API`，之所以这样命名，是需要和不同的操作系统进行区分，比如最一开始的 16 位版的 `Win16 API`，和后来流行的 `Win64 API` 。

API 通过多个 DLL 文件来提供，各个 API 的实体都是用 C 语言编写的函数。所以，在 C 语言环境下，使用 API 更加容易，比如 API 所用到的 `MessageBox()` 函数，就被保存在了 Windows 提供的 user32.dll 这个 DLL 文件中。

提供采用了 GUI 的用户界面

`GUI(Graphical User Interface)` 指得就是图形用户界面，通过点击显示器中的窗口以及图标等可视化的用户界面，举个例子：Linux 操作系统就有两个版本，一种是简洁版，直接通过命令行控制硬件，还有一种是可视化版，通过光标点击图形界面来控制硬件。

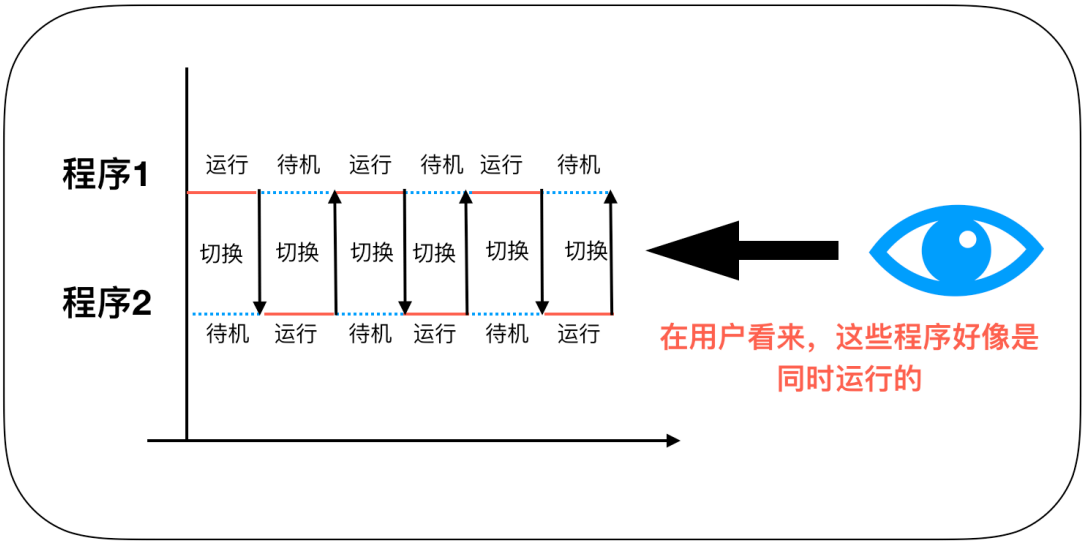
通过 WYSIWYG 实现打印输出

WYSIWYG 指的是显示器上输出的内容可以直接通过打印机打印输出。在 Windows 中，显示器和打印机被认作同等的图形输出设备处理的，该功能也为 WYSIWYG 提供了条件。

借助 WYSIWYG 功能，程序员可以轻松不少。最初，为了是现在显示器中显示和在打印机中打印，就必须分别编写各自的程序，而在 Windows 中，可以借助 WYSIWYG 基本上在一个程序中就可以做到显示和打印这两个功能了。

提供多任务功能

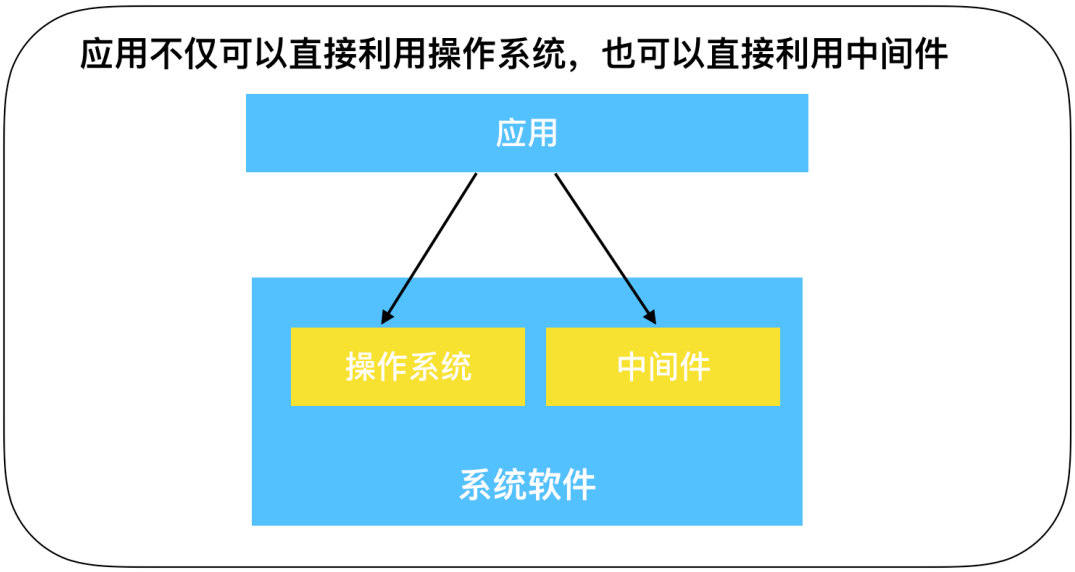
多任务指的就是同时能够运行多个应用程序的功能，Windows 是通过 `时钟分割` 技术来实现多任务功能的。时钟分割指的是短时间间隔内，多个程序切换运行的方式。在用户看来，就好像是多个程序在同时运行，其底层是 `CPU 时间切片`，这也是多线程多任务的核心。



CPU分片，也是时钟分割

提供网络功能和数据库功能

Windows 中，网络功能是作为标准功能提供的。数据库(数据库服务器)功能有时也会在后面追加。网络功能和数据库功能虽然并不是操作系统不可或缺的，但因为它们和操作系统很接近，所以被统称为 `中间件` 而不是应用。意思是处于操作系统和应用的中间层，操作系统和中间件组合在一起，称为 `系统软件` 。应用不仅可以利用操作系统，也可以利用中间件的功能。



应用可以使用操作系统和中间件

相对于操作系统一旦安装就不能轻易更换，中间件可以根据需要进行更换，不过，对于大部分应用来说，更换中间件的话，会造成应用也随之更换，从这个角度来说，更å换中间件也不是那么容易。

通过即插即用实现设备驱动的自动设定

`即插即用(Plug-and-Play)` 指的是新的设备连接(plug) 后就可以直接使用的机制，新设备连接计算机后，计算机就会自动安装和设定用来控制该设备的 `驱动程序`

设备驱动是操作系统的一部分，提供了同硬件进行基本的输入输出的功能。键盘、鼠标、显示器、磁盘装置等，这些计算机中必备的硬件的设备驱动，一般都是随操作系统一起安装的。

有时 DLL 文件也会同设备驱动文件一起安装。这些 DLL 文件中存储着用来利用该新追加的硬件API，通过 API ，可以制作出运行该硬件的心应用。

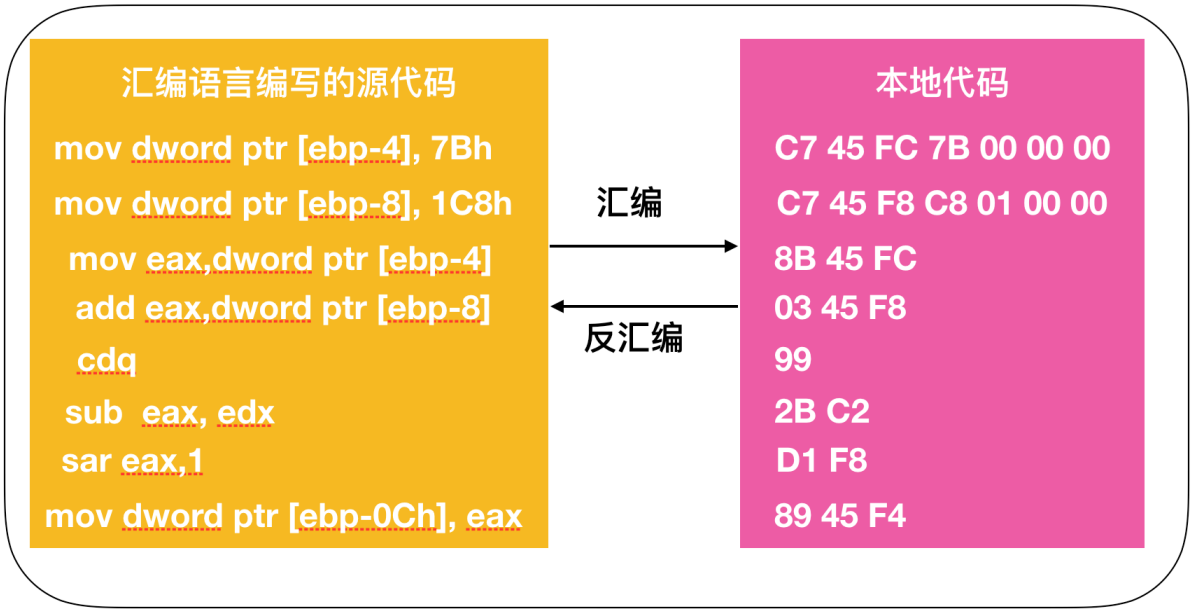
汇编语言和本地代码

计算机 CPU 只能运行本地代码(机器语言)程序，用 C 语言等高级语言编写的代码，需要经过编译器编译后，转换为本地代码才能够被 CPU 解释执行。

但是本地代码的可读性非常差，所以需要使用一种能够直接读懂的语言来替换本地代码，那就是在各本地代码中，附带上表示其功能的英文缩写，比如在加法运算的本地代码加上 `add(addition)` 的缩写、在比较运算符的本地代码中加上 `cmp(compare)` 的缩写等，这些通过缩写来表示具体本地代码指令的标志称为 `助记符` ，使用助记符的语言称为 `汇编语言` 。这样，通过阅读汇编语言，也能够了解本地代码的含义了。

不过，即使是使用汇编语言编写的源代码，最终也必须要转换为本地代码才能够运行，负责做这项工作的程序称为 `编译器` ，转换的这个过程称为 `汇编` 。在将源代码转换为本地代码这个功能方面，汇编器和编译器是同样的。

用汇编语言编写的源代码和本地代码是一一对应的。因而，本地代码也可以反过来转换成汇编语言编写的代码。把本地代码转换为汇编代码的这一过程称为 `反汇编` ，执行反汇编的程序称为 `反汇编程序` 。



本地代码和汇编语言一对一的转换

哪怕是 C 语言编写的源代码，编译后也会转换成特定 CPU 用的本地代码。而将其反汇编的话，就可以得到汇编语言的源代码，并对其内容进行调查。不过，**本地代码变成 C 语言源代码的反编译，要比本地代码转换成汇编代码的反汇编要困难**，这是因为，C 语言代码和本地代码不是一一对应的关系。

通过编译器输出汇编语言的源代码

我们上面提到本地代码可以经过反汇编转换成为汇编代码，但是只有这一种转换方式吗？显然不是，C 语言编写的源代码也能够通过编译器编译称为汇编代码，下面就来尝试一下。

首先需要先做一些准备，需要先下载 `Borland C++ 5.5` 编译器，为了方便，我这边直接下载好了读者直接从我的百度网盘提取即可 （链接: **https://pan.baidu.com/s/19LqVICpn5GcV88thD2AnIA** 密码:hz1u）

下载完毕，需要进行配置，下面是配置说明 （**https://wenku.baidu.com/view/22e2f418650e52ea551898ad.html**），教程很完整跟着配置就可以，下面开始我们的编译过程

首先用 Windows 记事本等文本编辑器编写如下代码

```
// 返回两个参数值之和的函数
int AddNum(int a,int b){
  return a + b;
}

// 调用 AddNum 函数的函数
void MyFunc(){
  int c;
  c = AddNum(123,456);
}
```

编写完成后将其文件名保存为 Sample4.c ，C 语言源文件的扩展名，通常用 `.c` 来表示，上面程序是提供两个输入参数并返回它们之和。

在 Windows 操作系统下打开 `命令提示符` ，切换到保存 Sample4.c 的文件夹下，然后在命令提示符中输入

```
bcc32 -c -S Sample4.c
```

bcc32 是启动 Borland C++ 的命令， `-c` 的选项是指仅进行编译而不进行链接， `-S` 选项被用来指定生成汇编语言的源代码

作为编译的结果，当前目录下会生成一个名为 `Sample4.asm` 的汇编语言源代码。汇编语言源文件的扩展名，通常用 `.asm` 来表示，下面就让我们用编辑器打开看一下 Sample4.asm 中的内容

```
    .386p
    ifdef ??version
    if    ??version GT 500H
    .mmx
    endif
    endif
    model flat
    ifndef  ??version
    ?debug  macro
    endm
    endif
    ?debug  S "Sample4.c"
    ?debug  T "Sample4.c"
```

```
14  _TEXT segment dword public use32 'CODE'
15  _TEXT ends
16  _DATA segment dword public use32 'DATA'
17  _DATA ends
18  _BSS  segment dword public use32 'BSS'
19  _BSS  ends
20  DGROUP  group _BSS,_DATA
21  _TEXT segment dword public use32 'CODE'
22  _AddNum proc  near
23  ?live1@0:
24     ;
25     ;  int AddNum(int a,int b){
26     ;
27    push      ebp
28    mov       ebp,esp
29     ;
30     ;
31     ;      return a + b;
32     ;
33  @1:
34    mov       eax,dword ptr [ebp+8]
35    add       eax,dword ptr [ebp+12]
36     ;
37     ;  }
38     ;
39  @3:
40  @2:
41    pop       ebp
42    ret
43  _AddNum endp
44  _MyFunc proc  near
45  ?live1@48:
46     ;
47     ;  void MyFunc(){
48     ;
49    push      ebp
50    mov       ebp,esp
51     ;
52     ;      int c;
53     ;      c = AddNum(123,456);
54     ;
55  @4:
56    push      456
57    push      123
58    call      _AddNum
59    add       esp,8
60     ;
61     ;  }
62     ;
63  @5:
64    pop       ebp
65    ret
```

```
66   _MyFunc endp
67   _TEXT ends
68     public  _AddNum
69     public  _MyFunc
70     ?debug  D "Sample4.c" 20343 45835
71     end
```

这样，编译器就成功的把 C 语言转换成为了汇编代码了。

不会转换成本地代码的伪指令

第一次看到汇编代码的读者可能感觉起来比较难，不过实际上其实比较简单，而且可能比 C 语言还要简单，为了便于阅读汇编代码的源代码，需要注意几个要点

汇编语言的源代码，是由转换成本地代码的指令（后面讲述的操作码）和针对汇编器的伪指令构成的。伪指令负责把程序的构造以及汇编的方法指示给汇编器（转换程序）。不过伪指令是无法汇编转换成为本地代码的。下面是上面程序截取的伪指令

```
1   _TEXT segment dword public use32 'CODE'
2   _TEXT ends
3   _DATA segment dword public use32 'DATA'
4   _DATA ends
5   _BSS  segment dword public use32 'BSS'
6   _BSS  ends
7   DGROUP  group _BSS,_DATA
8
9   _AddNum proc  near
10  _AddNum endp
11
12  _MyFunc proc  near
13  _MyFunc endp
14
15  _TEXT ends
16    end
```

由伪指令 `segment` 和 `ends` 围起来的部分，是给构成程序的命令和数据的集合体上加一个名字而得到的，称为 `段定义` 。段定义的英文表达具有 `区域` 的意思，在这个程序中，段定义指的是命令和数据等程序的集合体的意思，一个程序由多个段定义构成。

上面代码的开始位置，定义了3个名称分别为 `_TEXT、_DATA、_BSS` 的段定义， `_TEXT` 是指定的段定义， `_DATA` 是被初始化（有初始值）的数据的段定义， `_BSS` 是尚未初始化的数据的段定义。这种定义的名称是由 Borland C++ 定义的，是由 Borland C++ 编译器自动分配的，所以程序段定义的顺序就成为了 `_TEXT、_DATA、_BSS` ，这样也确保了内存的连续性

```
1   _TEXT segment dword public use32 'CODE'
2   _TEXT ends
3   _DATA segment dword public use32 'DATA'
4   _DATA ends
5   _BSS  segment dword public use32 'BSS'
6   _BSS  ends
```

> 段定义(segment) 是用来区分或者划分范围区域的意思。汇编语言的 segment 伪指令表示段定义的起始，ends 伪指令表示段定义的结束。段定义是一段连续的内存空间

而 `group` 这个伪指令表示的是将 `_BSS和_DATA` 这两个段定义汇总名为 DGROUP 的组

```
1   DGROUP  group _BSS,_DATA
```

围起 `_AddNum` 和 `_MyFun` 的 `_TEXT` segment 和 `_TEXT` ends，表示 `_AddNum` 和 `_MyFun` 是属于 `_TEXT` 这一段定义的。

```
1   _TEXT segment dword public use32 'CODE'
2   _TEXT ends
```

因此，即使在源代码中指令和数据是混杂编写的，经过编译和汇编后，也会转换成为规整的本地代码。

`_AddNum proc` 和 `_AddNum endp` 围起来的部分，以及 `_MyFunc proc` 和 `_MyFunc endp` 围起来的部分，分别表示 AddNum 函数和 MyFunc 函数的范围。

```
1   _AddNum proc  near
2   _AddNum endp
3
4   _MyFunc proc  near
5   _MyFunc endp
```

编译后在函数名前附带上下划线 `_` ，是 Borland C++ 的规定。在 C 语言中编写的 AddNum 函数，在内部是以 _AddNum 这个名称处理的。伪指令 proc 和 endp 围起来的部分，表示的是 `过程(procedure)` 的范围。在汇编语言中，这种相当于 C 语言的函数的形式称为过程。

末尾的 `end` 伪指令，表示的是源代码的结束。

汇编语言的语法是 操作码 + 操作数

在汇编语言中，一行表示一对 CPU 的一个指令。汇编语言指令的语法结构是 **操作码 + 操作数**，也存在只有操作码没有操作数的指令。

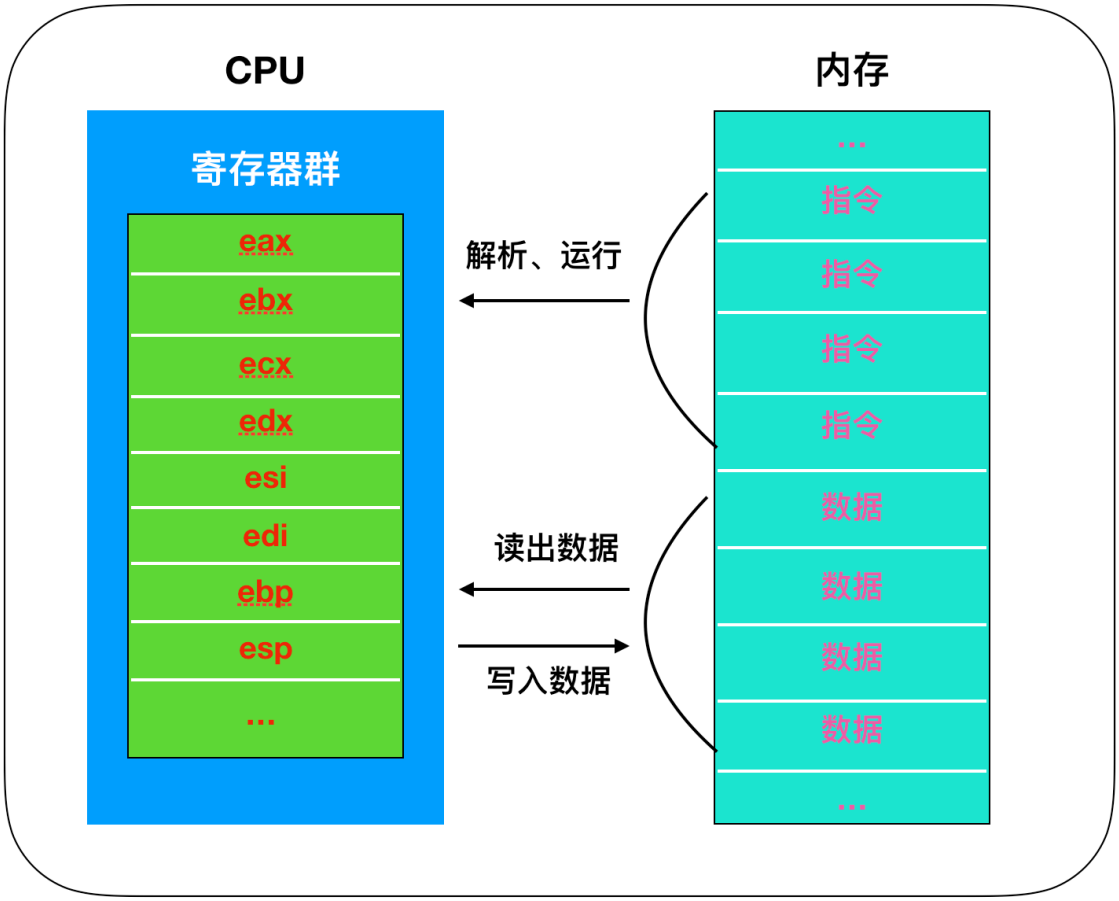
操作码表示的是指令动作，操作数表示的是指令对象。操作码和操作数一起使用就是一个英文指令。比如从英语语法来分析的话，操作码是动词，操作数是宾语。比如这个句子 `Give me money` 这个英文指令的话，Give 就是操作码，me 和 money 就是操作数。汇编语言中存在多个操作数的情况，要用逗号把它们分割，就像是 Give me,money 这样。

能够使用何种形式的操作码，是由 CPU 的种类决定的，下面对操作码的功能进行了整理。

操作码	操作数	功能
mov	A,B	把B的值赋给A
and	A,B	把A和B同时相加，并把结果赋给A
push	A	把A的值存储在栈中
pop	A	从栈中读出值，并将其赋值给A
call	A	调用函数A
ret	无	处理返回给调用源函数

部分操作码及其功能

本地代码需要加载到内存后才能运行，内存中存储着构成本地代码的指令和数据。程序运行时，CPU会从内存中把数据和指令读出来，然后放在 CPU 内部的寄存器中进行处理。



CPU 和 内存的关系

寄存器是 CPU 中的存储区域，寄存器除了具有临时存储和计算的功能之外，还具有运算功能，x86 系列的主要种类和角色如下图所示

寄存器名	名称	主要功能
eax	累加寄存器	运算
ebc	基址寄存器	存储内存地址
ecx	计数寄存器	计算循环次数
edx	数据寄存器	存储数据
esi	源基址寄存器	存储数据发送源的内存地址
edi	目的基址寄存器	存储数据发送目标的内存地址
ebp	扩展基址指针寄存器	存储数据存储领域基点的内存地址
esp	扩展栈指针寄存器	存储栈中最高位数据的内存地址

x86系列 CPU 的主要寄存器

指令解析

下面就对 CPU 中的指令进行分析

最常用的 mov 指令

指令中最常使用的是对寄存器和内存进行数据存储的 `mov` 指令，mov 指令的两个操作数，分别用来指定数据的存储地和读出源。操作数中可以指定寄存器、常数、标签(附加在地址前)，以及用方括号 `([])` 围起来的这些内容。如果指定了没有用 `([])` 方括号围起来的内容，就表示对该值进行处理；如果指定了用方括号围起来的内容，方括号的值则会被解释为内存地址，然后就会对该内存地址对应的值进行读写操作。让我们对上面的代码片段进行说明

```
1    mov     ebp,esp
2    mov     eax,dword ptr [ebp+8]
```

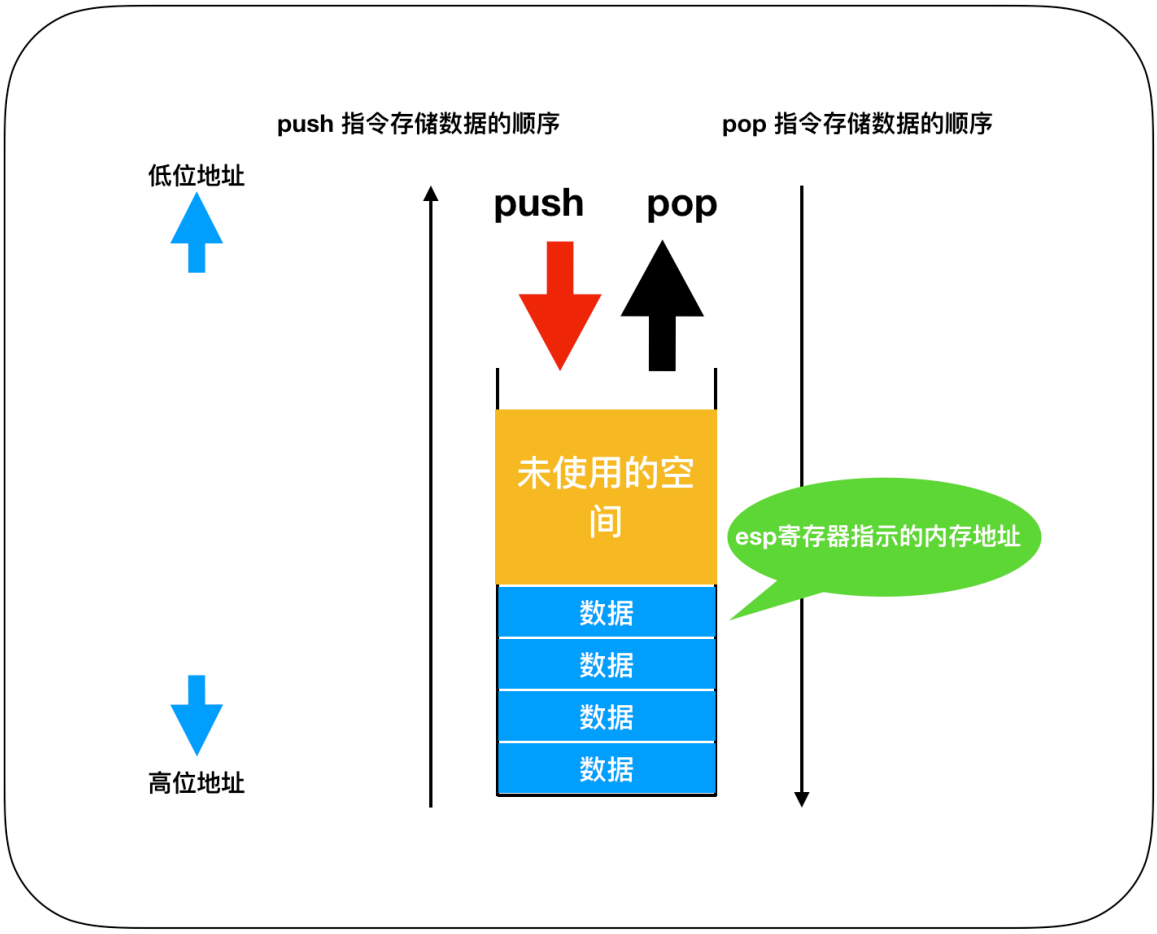
mov ebp,esp 中，esp 寄存器中的值被直接存储在了 ebp 中，也就是说，如果 esp 寄存器的值是100的话那么 ebp 寄存器的值也是 100。

而在 `mov eax,dword ptr [ebp+8]` 这条指令中，ebp 寄存器的值 + 8 后会被解析称为内存地址。如果 ebp

寄存器的值是100的话，那么 eax 寄存器的值就是 100 + 8 的地址的值。 `dword ptr` 也叫做 `double word pointer` 简单解释一下就是从指定的内存地址中读出4字节的数据

对栈进行 push 和 pop

程序运行时，会在内存上申请分配一个称为栈的数据空间。栈（stack）的特性是后入先出，数据在存储时是从内存的下层（大的地址编号）逐渐往上层（小的地址编号）累积，读出时则是按照从上往下进行读取的。



栈的模型

栈是存储临时数据的区域，它的特点是通过 push 指令和 pop 指令进行数据的存储和读出。向栈中存储数据称为 `入栈` ，从栈中读出数据称为 `出栈` ，32位 x86 系列的 CPU 中，进行1次 push 或者 pop，即可处理 32 位（4字节）的数据。

函数的调用机制

下面我们一起来分析一下函数的调用机制，我们以上面的 C 语言编写的代码为例。首先，让我们从 `MyFunc` 函数调用 `AddNum` 函数的汇编语言部分开始，来对函数的调用机制进行说明。栈在函数的调用中发挥了巨大的作用，下面是经过处理后的 MyFunc 函数的汇编处理内容

```
1   _MyFunc    proc    near
2     push      ebp      ; 将 ebp 寄存器的值存入栈中              (1)
3     mov       ebp,esp  ; 将 esp 寄存器的值存入 ebp 寄存器中     (2)
4     push      456      ; 将 456 入栈                            (3)
5     push      123      ; 将 123 入栈                            (4)
6     call      _AddNum  ; 调用 AddNum 函数                       (5)
7     add       esp,8    ; esp 寄存器的值 + 8                     (6)
8     pop       ebp      ; 读出栈中的数值存入 esp 寄存器中        (7)
9     ret                ; 结束 MyFunc 函数，返回到调用源         (8)
10  _MyFunc    endp
```

代码解释中的(1)、(2)、(7)、(8)的处理适用于 C 语言中的所有函数，我们会在后面展示 `AddNum` 函数处理内容时进行说明。这里希望大家先关注(3) - (6) 这一部分，这对了解函数调用机制至关重要。

(3) 和 (4) 表示的是将传递给 AddNum 函数的参数通过 push 入栈。在 C 语言源代码中，虽然记述为函数 AddNum(123,456)，但入栈时则会先按照 456，123 这样的顺序。也就是位于后面的数值先入栈。这是 C 语言的规定。(5) 表示的 call 指令，会把程序流程跳转到 AddNum 函数指令的地址处。在汇编语言中，`函数名` 表示的就是函数所在的内存地址。AddNum 函数处理完毕后，程序流程必须要返回到编号(6) 这一行。call 指令运行后，call 指令的下一行(也就指的是 (6) 这一行)的内存地址(调用函数完毕后要返回的内存地址)会自动的 push 入栈。该值会在 AddNum 函数处理的最后通过 `ret` 指令 pop 出栈，然后程序会返回到 (6) 这一行。

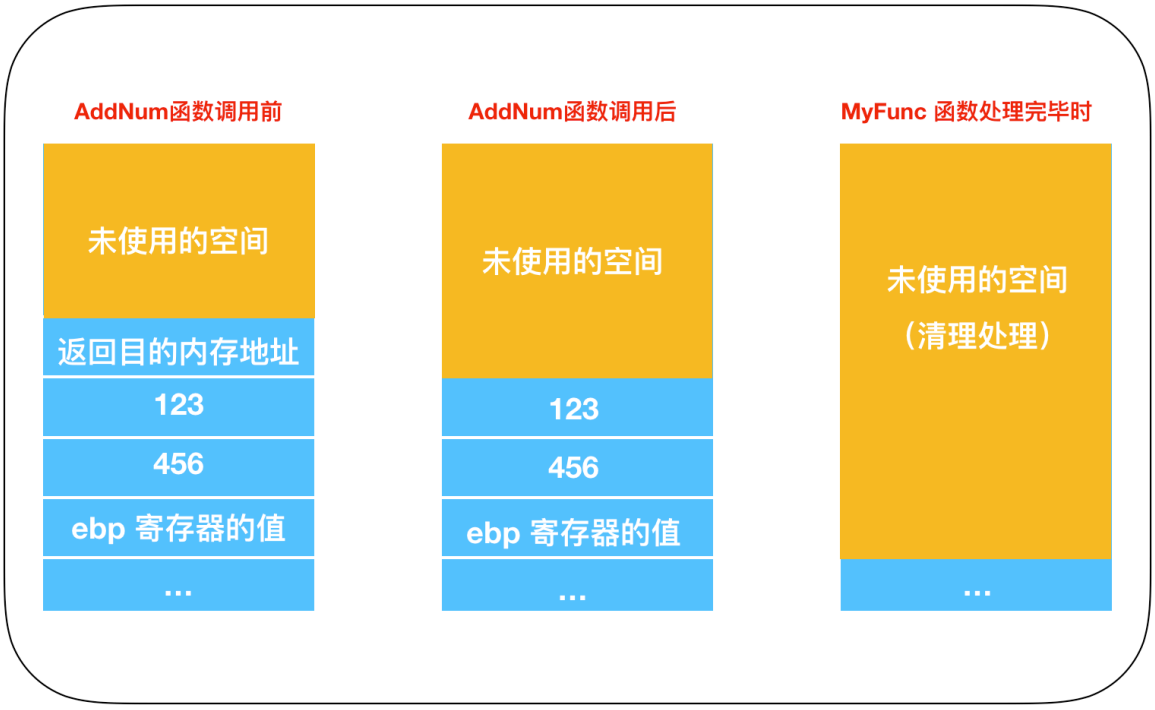
(6) 部分会把栈中存储的两个参数 (456 和 123) 进行销毁处理。虽然通过两次的 pop 指令也可以实现，不过采用 esp 寄存器 + 8 的方式会更有效率(处理 1 次即可)。对栈进行数值的输入和输出时，数值的单位是4字节。因此，通过在负责栈地址管理的 esp 寄存器中加上4的2倍8，就可以达到和运行两次 pop 命令同样的效果。虽然内存中的数据实际上还残留着，但只要把 esp 寄存器的值更新为数据存储地址前面的数据位置，该数据也就相当于销毁了。

我在编译 `Sample4.c` 文件时，出现了下图的这条消息

```
D:\C>bcc32 -c -S Sample4.c
Borland C++ 5.5.1 for Win32 Copyright (c) 1993, 2000 Borland
Sample4.c:
Warning W8004 Sample4.c 10: 'c' is assigned a value that is never used in functi
on MyFunc
```

图中的意思是指 c 的值在 MyFunc 定义了但是一直未被使用，这其实是一项编译器优化的功能，由于存储着 AddNum 函数返回值的变量 c 在后面没有被用到，因此编译器就认为 **该变量没有意义**，进而也就**没有生成与之对应的汇编语言代码**。

下图是调用 AddNum 这一函数前后栈内存的变化



AddNum 函数调用前后栈的变化

函数的内部处理

上面我们用汇编代码分析了一下 Sample4.c 整个过程的代码，现在我们着重分析一下 AddNum 函数的源代码部分，分析一下参数的接收、返回值和返回等机制

```
1   _AddNum     proc    near
2     push      ebp                     ----------(1)
3     mov       ebp,esp                 ----------(2)
4     mov       eax,dword ptr[ebp+8]    ----------(3)
5     add       eax,dword ptr[ebp+12]   ----------(4)
6     pop       ebp                     ----------(5)
7     ret       ----------------------------------(6)
8   _AddNum     endp
```

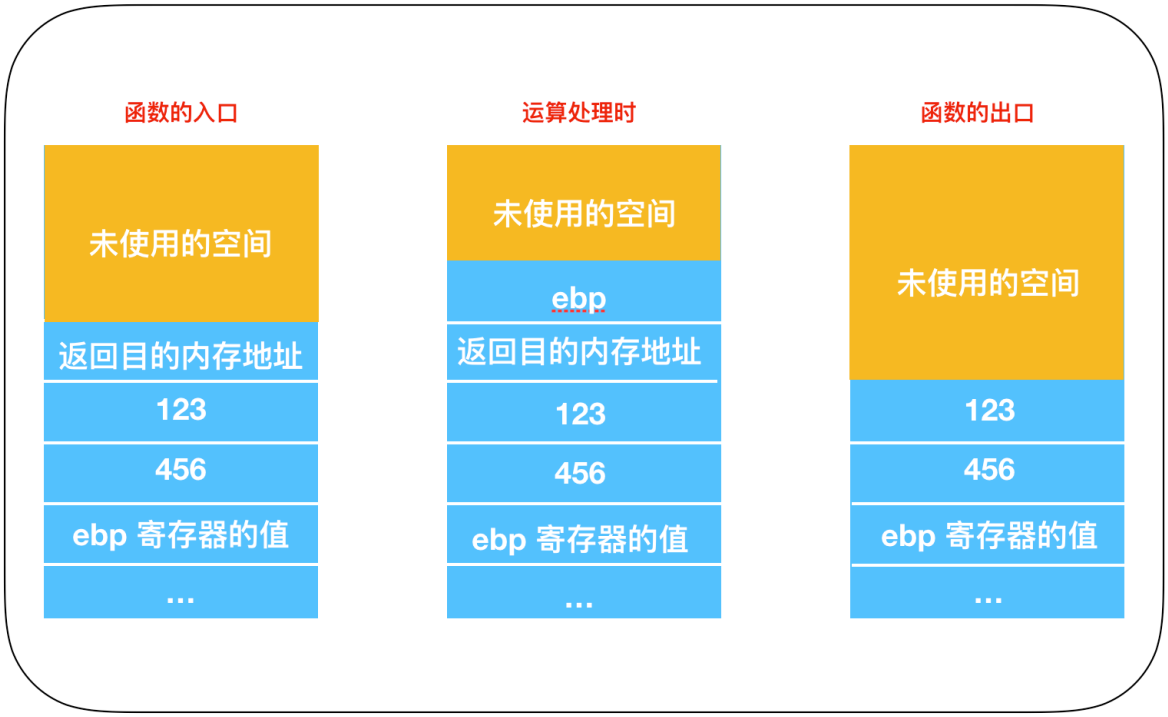
ebp 寄存器的值在(1)中入栈，在(5)中出栈，这主要是为了把函数中用到的 ebp 寄存器的内容，恢复到函数调用前的状态。

(2) 中把负责管理栈地址的 esp 寄存器的值赋值到了 ebp 寄存器中。这是因为，在 mov 指令中方括号内的参数，是不允许指定 esp 寄存器的。因此，这里就采用了不直接通过 esp，而是用 ebp 寄存器来读写栈内容的方法。

(3) 使用[ebp + 8] 指定栈中存储的第1个参数123，并将其读出到 eax 寄存器中。像这样，不使用 pop 指令，也可以参照栈的内容。而之所以从多个寄存器中选择了 eax 寄存器，是因为 eax 是负责运算的累加寄存器。

通过(4) 的 add 指令，把当前 eax 寄存器的值同第2个参数相加后的结果存储在 eax 寄存器中。[ebp + 12] 是用来指定第2个参数456的。在 C 语言中，函数的返回值必须通过 eax 寄存器返回，这也是规定。也就是 **函数的参数是通过栈来传递，返回值是通过寄存器返回的**。

(6) 中 ret 指令运行后，函数返回目的地内存地址会 `自动出栈`，据此，程序流程就会跳转返回到 `(6) (Call _AddNum)` 的下一行。这时，AddNum 函数入口和出口处栈的状态变化，就如下图所示



AddNum 函数内部的栈状态变化

全局变量和局部变量

在熟悉了汇编语言后，接下来我们来了解一下全局变量和局部变量，在函数外部定义的变量称为 `全局变量` ，在函数内部定义的变量称为 `局部变量` ，全局变量可以在任意函数中使用，局部变量只能在函数定义局部变量的内部使用。下面，我们就通过汇编语言来看一下全局变量和局部变量的不同之处。

下面定义的 C 语言代码分别定义了局部变量和全局变量，并且给各变量进行了赋值，我们先看一下源代码部分

```
// 定义被初始化的全局变量
int a1 = 1;
int a2 = 2;
int a3 = 3;
int a4 = 4;
int a5 = 5;

// 定义没有初始化的全局变量
int b1,b2,b3,b4,b5;

// 定义函数
void MyFunc(){
  // 定义局部变量
  int c1,c2,c3,c4,c5,c6,c7,c8,c9,c10;

  // 给局部变量赋值
  c1 = 1;
  c2 = 2;
  c3 = 3;
  c4 = 4;
  c5 = 5;
  c6 = 6;
  c7 = 7;
  c8 = 8;
  c9 = 9;
  c10 = 10;

  // 把局部变量赋值给全局变量
  a1 = c1;
  a2 = c2;
  a3 = c3;
  a4 = c4;
  a5 = c5;
  b1 = c6;
  b2 = c7;
  b3 = c8;
  b4 = c9;
  b5 = c10;
}
```

上面的代码挺暴力的，不过没关系，能够便于我们分析其汇编源码就好，我们用 Borland C++ 编译后的汇编代码如下，编译完成后的源码比较长，这里我们只拿出来一部分作为分析使用（我们改变了一下段定义顺序，删除了部分注释）

```
_DATA segment dword public use32 'DATA'
   align 4
  _a1 label dword
        dd 1
   align 4
  _a2 label dword
        dd 2
   align 4
  _a3 label dword
        dd 3
   align 4
  _a4 label dword
        dd 4
   align 4
  _a5 label dword
        dd 5
_DATA ends

_BSS segment dword public use32 'BSS'
 align 4
  _b1 label dword
        db 4 dup(?)
   align 4
  _b2 label dword
        db 4 dup(?)
   align 4
  _b3 label dword
        db 4 dup(?)
   align 4
  _b4 label dword
        db 4 dup(?)
   align 4
  _b5 label dword
        db 4 dup(?)
_BSS ends

_TEXT segment dword public use32 'CODE'
_MyFunc proc near

 push      ebp
 mov       ebp,esp
 add       esp,-20
 push      ebx
 push      esi
 mov       eax,1
 mov       edx,2
 mov       ecx,3
 mov       ebx,4
 mov       esi,5
 mov       dword ptr [ebp-4],6
 mov       dword ptr [ebp-8],7
 mov       dword ptr [ebp-12],8
```

```
53    mov       dword ptr [ebp-16],9
54    mov       dword ptr [ebp-20],10
55    mov       dword ptr [_a1],eax
56    mov       dword ptr [_a2],edx
57    mov       dword ptr [_a3],ecx
58    mov       dword ptr [_a4],ebx
59    mov       dword ptr [_a5],esi
60    mov       eax,dword ptr [ebp-4]
61    mov       dword ptr [_b1],eax
62    mov       edx,dword ptr [ebp-8]
63    mov       dword ptr [_b2],edx
64    mov       ecx,dword ptr [ebp-12]
65    mov       dword ptr [_b3],ecx
66    mov       eax,dword ptr [ebp-16]
67    mov       dword ptr [_b4],eax
68    mov       edx,dword ptr [ebp-20]
69    mov       dword ptr [_b5],edx
70    pop       esi
71    pop       ebx
72    mov       esp,ebp
73    pop       ebp
74    ret
75
76  _MyFunc   endp
77  _TEXT   ends
```

编译后的程序，会被归类到名为段定义的组。

- **初始化的全局变量，会汇总到名为 _DATA 的段定义中**

```
1   _DATA segment dword public use32 'DATA'
2   ...
3   _DATA ends
```

- **没有初始化的全局变量，会汇总到名为 _BSS 的段定义中**

```
1   _BSS segment dword public use32 'BSS'
2    ...
3   _BSS ends
```

- **被段定义 _TEXT 围起来的汇编代码则是 Borland C++ 的定义**

```
1   _TEXT segment dword public use32 'CODE'
2   _MyFunc proc near
3   ...
4   _MyFunc   endp
5   _TEXT   ends
```

我们在分析上面汇编代码之前，先来认识一下更多的汇编指令，此表是对上面**部分操作码及其功能**的接续

操作码	操作数	功能
add	A,B	把A和B的值相加，并把结果赋值给A
call	A	调用函数A
cmp	A,B	对A和B进行比较，比较结果会自动存入标志寄存器中
inc	A	对A的值 + 1
ige	标签名	和 cmp 命令组合使用。跳转到标签行
jl	标签名	和 cmp 命令组合使用。跳转到标签行
jle	标签名	和 cmp 命令组合使用。跳转到标签行
jmp	标签名	和 cmp 命令组合使用。跳转到标签行
mov	A,B	把 B 的值赋给 A
pop	A	从栈中读取数值并存入A
push	A	把A的值存入栈中
ret	无	将处理返回到调用源
xor	A,B	A和B的位进行亦或比较，并将结果存入A中

我们首先来看一下 `_DATA` 段定义的内容。 `_a1 label dword` 定义了 `_a1` 这个标签。标签表示的是相对于段定义起始位置的位置。由于 `_a1` 在 `_DATA 段` 定义的开头位置，所以相对位置是0。 `_a1` 就相当于是全局变量a1。编译后的函数名和变量名前面会加一个 `(_)` ，这也是 Borland C++ 的规定。 `dd 1` 指的是，申请分配了4字节的内存空间，存储着1这个初始值。 dd指的是 `define double word` 表示有两个长度为2的字节领域(word)，也就是4字节的意思。

Borland C++ 中，由于 `int` 类型的长度是4字节，因此汇编器就把 int a1 = 1 变换成了 `_a1 label dword 和 dd 1` 。同样，这里也定义了相当于全局变量的 a2 - a5 的标签 `_a2 - _a5` ，它们各自的初始值 2 - 5 也被存储在各自的4字节中。

接下来，我们来说一说 `_BSS` 段定义的内容。这里定义了相当于全局变量 b1 - b5 的标签 `_b1 - _b5` 。其中的 `db 4dup(?)` 表示的是申请分配了4字节的领域，但值尚未确定（这里用 ? 来表示）的意思。 `db(define byte)` 表示有1个长度是1字节的内存空间。因而，db 4 dup(?) 的情况下，就是4字节的内存空间。

> 注意：db 4 dup(?) 不要和 dd 4 混淆了，前者表示的是4个长度是1字节的内存空间。而 db 4 表示的则是双字节(= 4 字节) 的内存空间中存储的值是 4

临时确保局部变量使用的内存空间

我们知道，局部变量是临时保存在寄存器和栈中的。函数内部利用栈进行局部变量的存储，函数调用完成后，局部变量值被销毁，但是寄存器可能用于其他目的。所以，**局部变量只是函数在处理期间临时存储在寄存器和栈中的**。

回想一下上述代码是不是定义了10个局部变量？这是为了表示存储局部变量的不仅仅是栈，还有寄存器。为了确保 c1 - c10 所需的域，寄存器空闲的时候就会使用寄存器，寄存器空间不足的时候就会使用栈。

让我们继续来分析上面代码的内容。 `_TEXT` 段定义表示的是 `MyFunc` 函数的范围。在 MyFunc 函数中定义的局部变量所需要的内存领域。会被尽可能的分配在寄存器中。大家可能认为使用高性能的寄存器来替代普通的内存是一种资源浪费，但是编译器不这么认为，只要寄存器有空间，编译器就会使用它。由于寄存器的访问速度远高于内存，所以直接访问寄存器能够高效的处理。局部变量使用寄存器，是 Borland C++ 编译器最优化的运行结果。

代码清单中的如下内容表示的是向寄存器中分配局部变量的部分

```
1   mov       eax,1
2   mov       edx,2
3   mov       ecx,3
4   mov       ebx,4
5   mov       esi,5
```

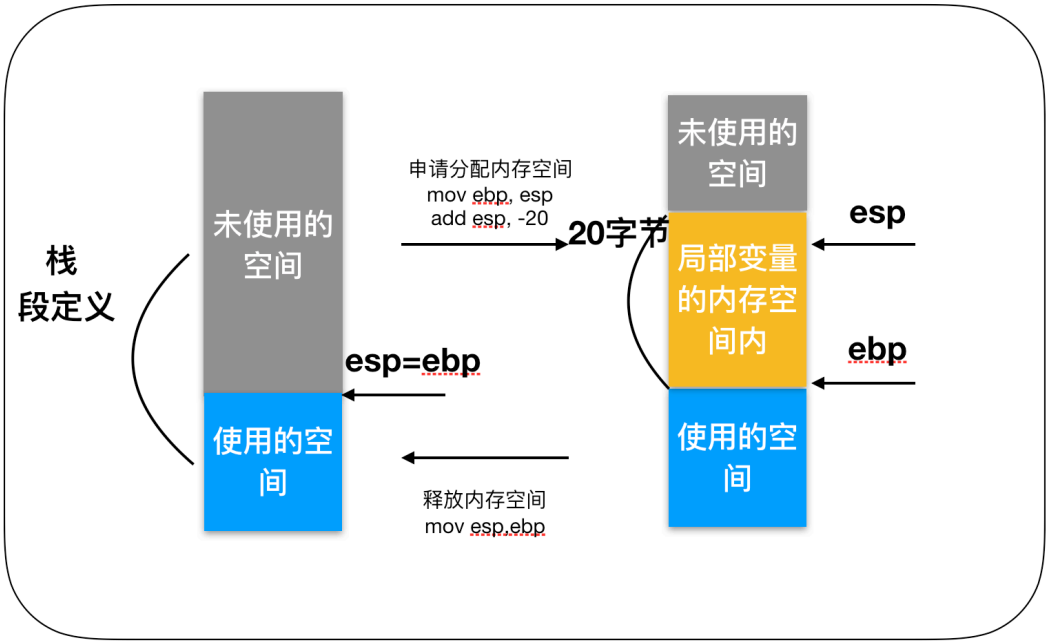
仅仅对局部变量进行定义是不够的，只有在给局部变量赋值时，才会被分配到寄存器的内存区域。上述代码相当于就是给5个局部变量 c1 - c5 分别赋值为 1 - 5。**eax、edx、ecx、ebx、esi** 是 x86 系列32位 CPU 寄存器的名称。至于使用哪个寄存器，是由 `编译器` 来决定的 。

x86 系列 CPU 拥有的寄存器中，程序可以操作的是十几，其中空闲的最多会有几个。因而，局部变量超过寄存器数量的时候，可分配的寄存器就不够用了，这种情况下，编译器就会把栈派上用场，用来存储剩余的局部变量。

在上述代码这一部分，给局部变量c1 - c5 分配完寄存器后，可用的寄存器数量就不足了。于是，剩下的5个局部变量c6 - c10 就被分配给了栈的内存空间。如下面代码所示

```
1   mov       dword ptr [ebp-4],6
2   mov       dword ptr [ebp-8],7
3   mov       dword ptr [ebp-12],8
4   mov       dword ptr [ebp-16],9
5   mov       dword ptr [ebp-20],10
```

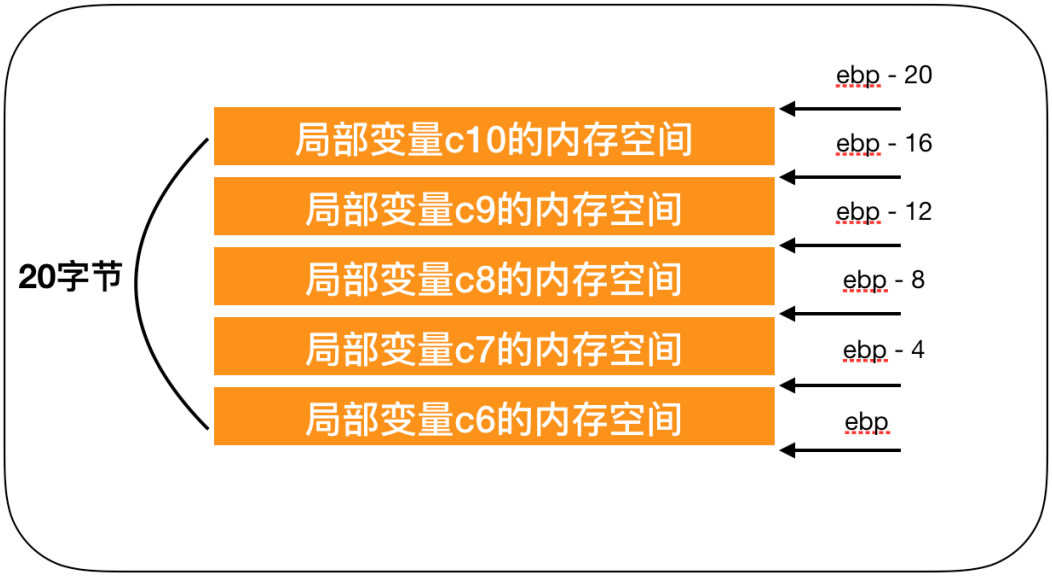
函数入口 `add esp,-20` 指的是，对栈数据存储位置的 esp 寄存器(栈指针)的值做减20的处理。为了确保内存变量 c6 - c10 在栈中，就需要保留5个 int 类型的局部变量（4字节 * 5 = 20 字节）所需的空间。 `mov ebp,esp` 这行指令表示的意思是将 esp 寄存器的值赋值到 ebp 寄存器。之所以需要这么处理，是为了通过在函数出口处 `mov esp ebp` 这一处理，把 esp 寄存器的值还原到原始状态，从而对申请分配的栈空间进行释放，这时栈中用到的局部变量就消失了。这也是栈的清理处理。在使用寄存器的情况下，局部变量则会在寄存器被用于其他用途时自动消失，如下图所示。



用于局部变量的栈空间的申请分配和释放

```
1    mov       dword ptr [ebp-4],6
2    mov       dword ptr [ebp-8],7
3    mov       dword ptr [ebp-12],8
4    mov       dword ptr [ebp-16],9
5    mov       dword ptr [ebp-20],10
```

这五行代码是往栈空间代入数值的部分，由于在向栈申请内存空间前，借助了 `mov ebp, esp` 这个处理，esp 寄存器的值被保存到了 esp 寄存器中，因此，通过使用[ebp - 4]、[ebp - 8]、[ebp - 12]、[ebp - 16]、[ebp - 20] 这样的形式，就可以申请分配20字节的栈内存空间切分成5个长度为4字节的空间来使用。例如，`mov dword ptr [ebp-4],6` 表示的就是，从申请分配的内存空间的下端(ebp寄存器指示的位置)开始向前4字节的地址([ebp - 4]) 中，存储着6这一4字节数据。



将栈内存空间进行分割

循环控制语句的处理

上面说的都是顺序流程，那么现在就让我们分析一下循环流程的处理，看一下 `for 循环` 以及 `if 条件分支` 等 c 语言程序的 `流程控制` 是如何实现的，我们还是以代码以及编译后的结果为例，看一下程序控制流程的处理过程。

```
1   // 定义MySub 函数
2   void MySub(){
3     // 不做任何处理
4
5   }
6
7   // 定义MyFunc 函数
8   void Myfunc(){
9     int i;
10    for(int i = 0;i < 10;i++){
11      // 重复调用MySub十次
12      MySub();
13    }
14  }
```

上述代码将局部变量 i 作为循环条件，循环调用十次 `MySub` 函数，下面是它主要的汇编代码

```
1         xor     ebx, ebx  ; 将寄存器清0
2   @4    call    _MySub    ; 调用MySub函数
3         inc     ebx       ; ebx寄存器的值 + 1
4         cmp     ebx,10    ; 将ebx寄存器的值和10进行比较
5         jl      short @4  ; 如果小于10就跳转到 @4
```

C 语言中的 for 语句是通过在括号中指定循环计数器的初始值(i = 0)、循环的继续条件(i < 10)、循环计数器的更新(i++) 这三种形式来进行循环处理的。与此相对的汇编代码就是通过 `比较指令(cmp)` 和 `跳转指令(jl)` 来实现的。

下面我们来对上述代码进行说明

`MyFunc` 函数中用到的局部变量只有 i ，变量 i 申请分配了 ebx 寄存器的内存空间。for 语句括号中的 i = 0 被转换为 `xor ebx,ebx` 这一处理，xor 指令会对左起第一个操作数和右起第二个操作数进行 XOR 运算，然后把结果存储在第一个操作数中。由于这里把第一个操作数和第二个操作数都指定为了 ebx，因此就变成了对相同数值的 XOR 运算。也就是说不管当前寄存器的值是什么，最终的结果都是 0。类似的，我们使用 `mov ebx,0` 也能得到相同的结果，但是 xor 指令的处理速度更快，而且编译器也会启动最优化功能。

> XOR 指的就是异或操作，它的运算规则是 **如果a、b两个值不相同，则异或结果为1。如果a、b两个值相同，异或结果为0**。
>
> 相同数值进行 XOR 运算，运算结果为0。XOR 的运算规则是，值不同时结果为1，值相同时结果为0。例如 01010101 和 01010101 进行运算，就会分别对各个数字位进行 XOR 运算。因为每个数字位都相同，所以运算结果为0。

ebx 寄存器的值初始化后，会通过 call 指定调用 _MySub 函数，从 _MySub 函数返回后，会执行 `inc ebx` 指令，对 ebx 的值进行 + 1 操作，这个操作就相当于 i++ 的意思，++ 表示的就是当前数值 + 1。

> 这里需要知道 i++ 和 ++i 的区别
>
> i++ 是先赋值，复制完成后再对 i执行 + 1 操作
>
> ++i 是先进行 +1 操作，完成后再进行赋值

`inc` 下一行的 `cmp` 是用来对第一个操作数和第二个操作数的数值进行比较的指令。 `cmp ebx,10` 就相当于 C 语言中的 i < 10 这一处理，意思是把 ebx 寄存器的值与10进行比较。汇编语言中比较指令的结果，会存储在 CPU 的标志寄存器中。不过，标志寄存器的值，程序是无法直接参考的。那如何判断比较结果呢?

汇编语言中有多个 `跳转指令` ，这些跳转指令会根据标志寄存器的值来判断是否进行跳转操作，例如最后一行的 jl，它会根据 cmp ebx,10 指令所存储在标志寄存器中的值来判断是否跳转， `jl` 这条指令表示的就是 `jump on less than(小于的话就跳转)` 。发现如果 i 比 10 小，就会跳转到 @4 所在的指令处继续执行。

那么汇编代码的意思也可以用 C 语言来改写一下，加深理解

```
1       i ^= i;
2   L4: MySub();
3       i++;
4       if(i < 10) goto L4;
```

代码第一行 i ^= i 指的就是 i 和 i 进行异或运算，也就是 XOR 运算，MySub() 函数用 L4 标签来替代，然后进行 i 自增操作，如果i 的值小于 10 的话，就会一直循环 MySub() 函数。

条件分支的处理方法

条件分支的处理方式和循环的处理方式很相似，使用的也是 cmp 指令和跳转指令。下面是用 C 语言编写的条件分支的代码

```
1   // 定义MySub1 函数
2   void MySub1(){
3
4    // 不做任何处理
5   }
6
7   // 定义MySub2 函数
8   void MySub2(){
9
10   // 不做任何处理
11  }
12
13  // 定义MySub3 函数
14  void MySub3(){
15
16   // 不做任何处理
17  }
18
19  // 定义MyFunc 函数
20  void MyFunc(){
21
```

```
    int a = 123;
    // 根据条件调用不同的函数
    if(a > 100){
     MySub1();
    }
    else if(a < 50){
     MySub2();
    }
    else
    {
     MySub3();
    }

  }
```

很简单的一个实现了条件判断的 C 语言代码，那么我们把它用 Borland C++ 编译之后的结果如下

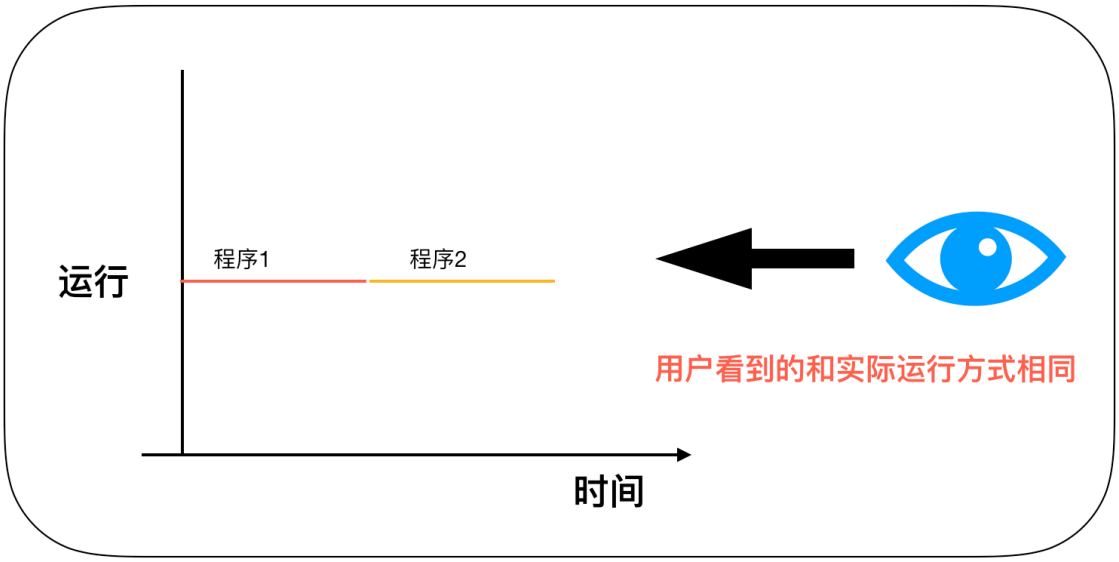
```
  _MyFunc proc near
    push      ebp
    mov       ebp,esp
    mov       eax,123       ; 把123存入 eax 寄存器中
    cmp       eax,100       ; 把 eax 寄存器的值同100进行比较
    jle       short @8      ; 比100小时，跳转到@8标签
    call      _MySub1       ; 调用MySub1函数
    jmp       short @11     ; 跳转到@11标签
  @8:
    cmp       eax,50        ; 把 eax 寄存器的值同50进行比较
    jge       short @10     ; 比50大时，跳转到@10标签
    call      _MySub2       ; 调用MySub2函数
    jmp       short @11     ; 跳转到@11标签
  @10:
    call      _MySub3       ; 调用MySub3函数
  @11:
    pop       ebp
    ret
  _MyFunc endp
```

上面代码用到了三种跳转指令，分别是 `jle(jump on less or equal)` 比较结果小时跳转，`jge(jump on greater or equal)` 比较结果大时跳转，还有不管结果怎样都会进行跳转的 `jmp`，在这些跳转指令之前还有用来比较的指令 `cmp`，构成了上述汇编代码的主要逻辑形式。

了解程序运行逻辑的必要性

通过对上述汇编代码和 C 语言源代码进行比较，想必大家对程序的运行方式有了新的理解，而且，从汇编源代码中获取的知识，也有助于了解 Java 等高级语言的特性，比如 Java 中就有 native 关键字修饰的变量，那么这个变量的底层就是使用 C 语言编写的，还有一些 Java 中的语法糖只有通过汇编代码才能知道其运行逻辑。在某些情况下，对于查找 bug 的原因也是有帮助的。

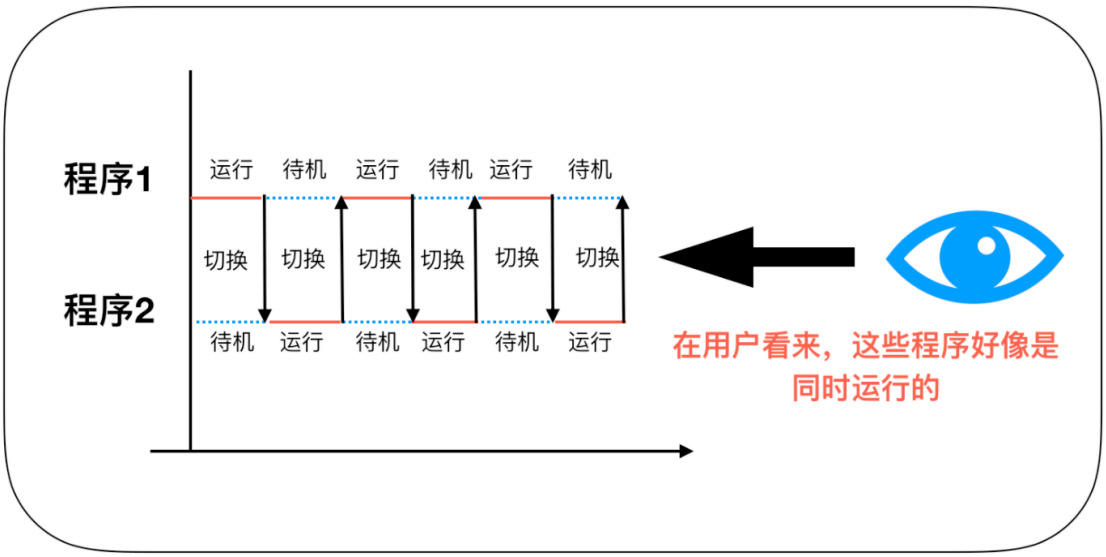
上面我们了解到的编程方式都是串行处理的，那么串行处理有什么特点呢?



串行处理

串行处理最大的一个特点就是 专心只做一件事情 ，一件事情做完之后才会去做另外一件事情。

计算机是支持多线程的，多线程的核心就是 CPU切换，如下图所示



CPU分片，也是时钟分割

我们还是举个实际的例子，让我们来看一段代码

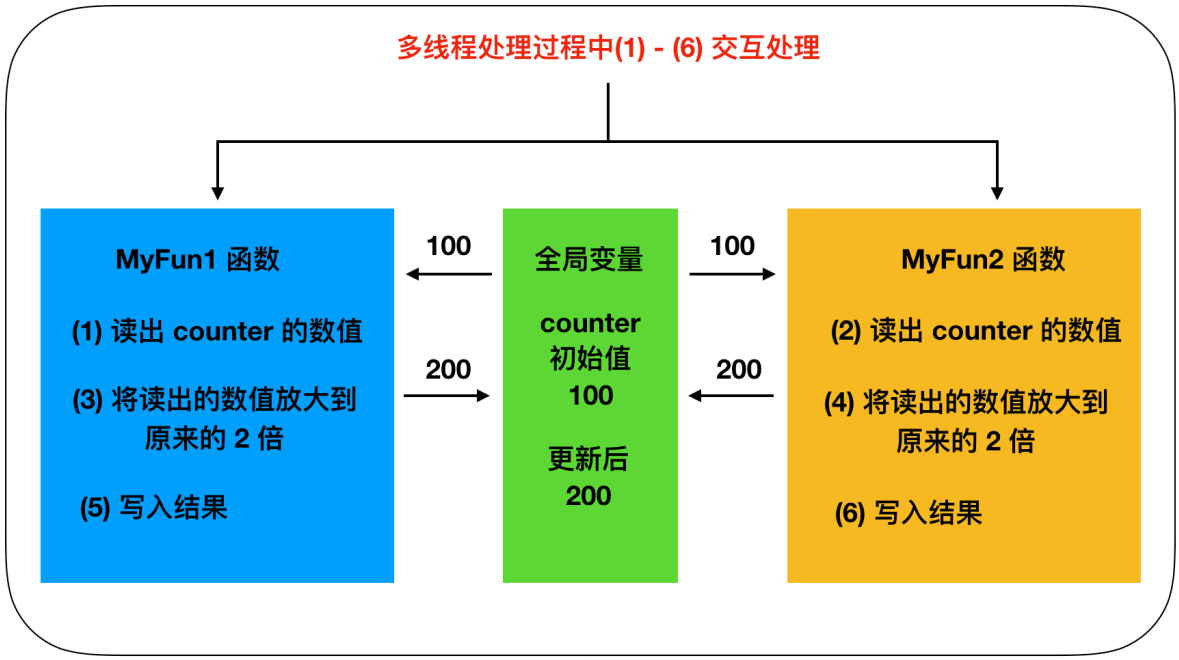
```
// 定义全局变量
int counter = 100;

// 定义MyFunc1()
void MyFunc(){
  counter *= 2;
}

// 定义MyFunc2()
void MyFunc2(){
  counter *= 2;
}
```

上述代码是更新 counter 的值的 C 语言程序，MyFunc1() 和 MyFunc2() 的处理内容都是把 counter 的值扩大至原来的二倍，然后再把 counter 的值赋值给 counter 。这里，我们假设使用 多线程处理 ，同时调用了一次MyFunc1 和 MyFunc2 函数，这时，全局变量 counter 的值，理应编程 100 * 2 * 2 = 400。如果你开启了多个线程的话，你会发现 counter 的数值有时也是 200，对于为什么出现这种情况，如果你不了解程序的运行方式，是很难找到原因的。

我们将上面的代码转换成汇编语言的代码如下

```
mov eax,dword ptr[_counter]   ; 将 counter 的值读入 eax 寄存器
add eax,eax                   ; 将 eax 寄存器的值扩大2倍。
mov dword ptr[_counter],eax   ; 将 eax 寄存器的值存入 counter 中。
```

在多线程程序中，用汇编语言表示的代码每运行一行，处理都有可能切换到其他线程中。因而，假设 MyFun1 函数在读出 counter 数值100后，还未来得及将它的二倍值200写入 counter 时，正巧 MyFun2 函数读出了 counter 的值100，那么结果就将变为 200 。



多线程交互程序处理步骤

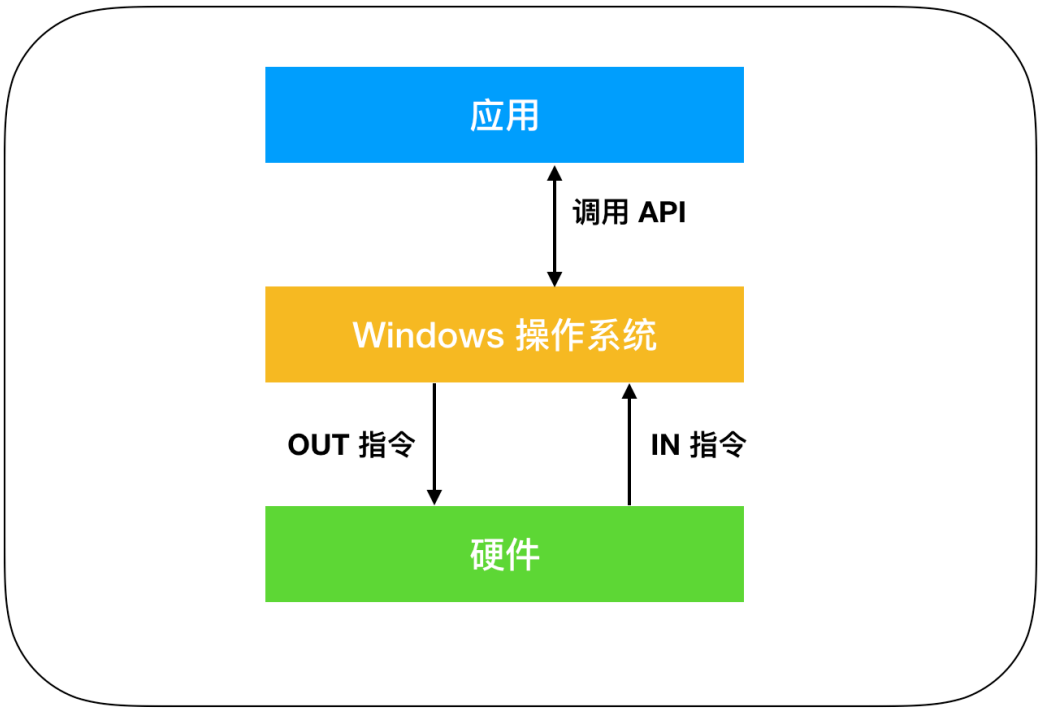
为了避免该bug，我们可以采用以函数或 C 语言代码的行为单位来禁止线程切换的 锁定 方法，或者使用某种线程安全的方式来避免该问题的出现。

现在基本上没有人用汇编语言来编写程序了，因为 C、Java等高级语言的效率要比汇编语言快很多。不过，汇编语言的经验还是很重要的，通过借助汇编语言，我们可以更好的了解计算机运行机制。

应用和硬件的关系

我们作为程序员一般很少直接操控硬件，我们一般通过 C、Java 等高级语言编写的程序起到间接控制硬件的作用。所以大家很少直接接触到硬件的指令，硬件的控制是由 `Windows 操作系统` 全权负责的。

你一定猜到我要说什么了，没错，我会说但是，任何事情没有绝对性，环境的不同会造成结果的偏差。虽然程序员没法直接控制硬件，并且 Windows 屏蔽了控制硬件的细节，但是 Windows 却为你开放了 `系统调用` 功能来实现对硬件的控制。在 Windows 中，系统调用称为 `API`，API 就是应用调用的函数，这些函数的实体被存放在 `DLL` 文件中。



应用通过 API 间接控制硬件

下面我们来看一个通过系统调用来间接控制硬件的实例

假如要在窗口中显示字符串，就可以使用 Windows API 中的 `TextOut` 函数。TextOut 函数的语法（C 语言）如下

```
BOOL TextOut{
  HDC hdc,              // 设备描述表的句柄
  int nXStart,          // 显示字符串的 x 坐标
  int nYStart,          // 显示字符串的 y 坐标
  LPCTSTR lpString,     // 指向字符串的指针
  int cbString          // 字符串的文字数
}
```

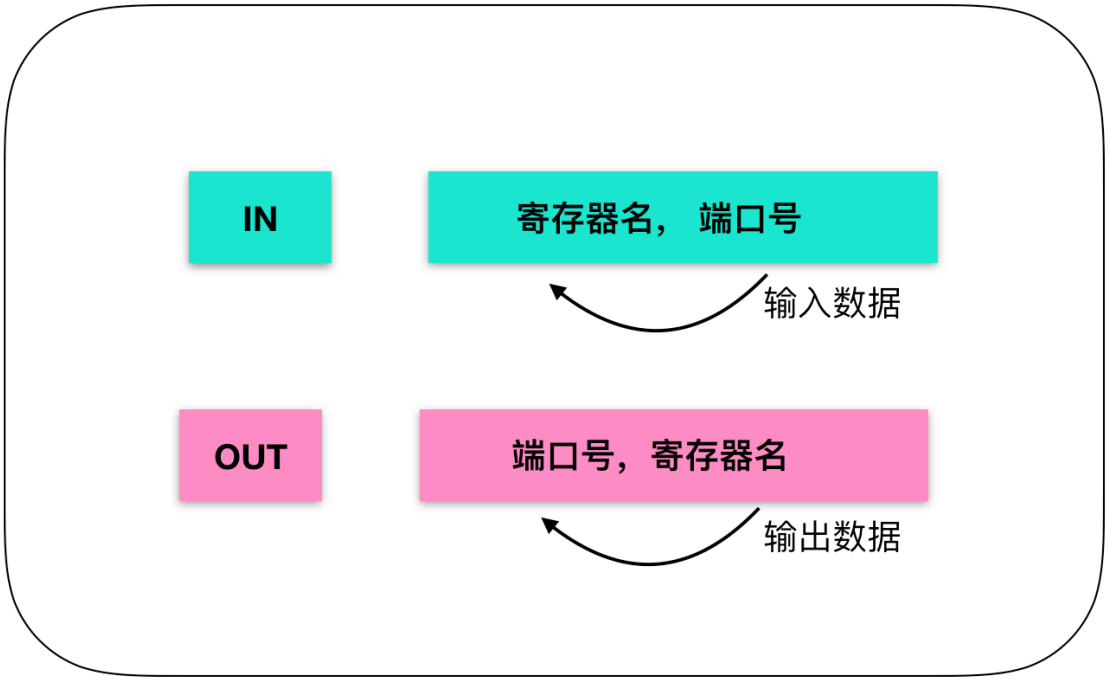
那么，在处理 TextOut 函数的内容时，Windows 做了些什么呢？从结果来看，Windows 直接控制了作为硬件的显示器。但 Windows 本身也是软件，由此可见，Windows 应该向 CPU 传递了某种指令，从而通过软件控制了硬件。

> Windows 提供的 TextOut 函数 API 可以向窗口和打印机输出字符。C 语言提供的 printf 函数，是用来在命令提示符中显示字符串的函数。使用 printf 函数是无法向打印机输出字符的。

支持硬件输入输出的 IN 指令和 OUT 指令

Windows 控制硬件借助的是输入和输出指令。其中具有代表性的两个输入输出指令就是 `IN` 和 `OUT` 指令。这些指令也是汇编语言的助记符。

可以通过 IN 和 OUT 指令来实现对数据的读入和输出，如下图所示



IN 指令和 OUT 指令

也就是说，IN 指令通过指定的端口号输入数据，OUT 指令则是把 CPU 寄存器中存储的数据输出到指定端口号的端口。

那么这个 `端口号` 和 `端口` 是什么呢？你感觉它像不像港口一样？通过标注哪个港口然后进行货物的运送和运出？

下面我们来看一下官方是如何定义端口号和端口的

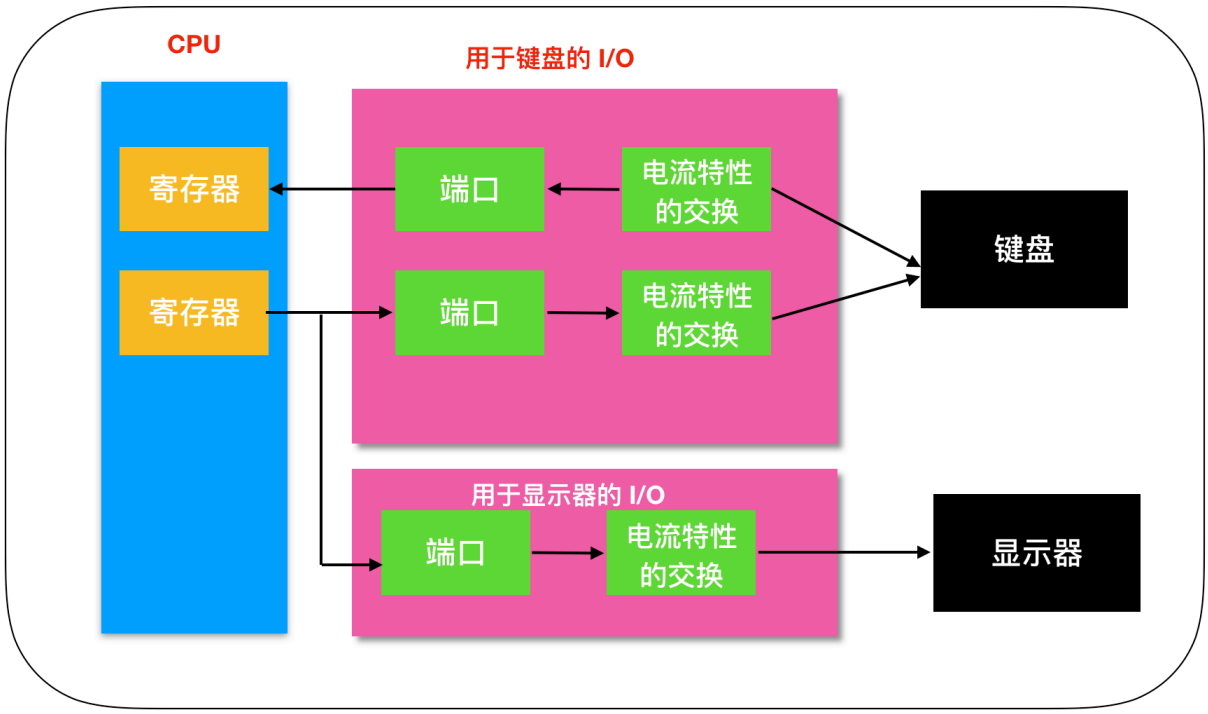
还记得计算机组成原理中计算机的五大组成部分吗，再来回顾一下：**运算器、控制器、存储器、输入设备和输出设备**。我们今天不谈前三个，就说说后面两个输入设备和输出设备，这两个与我们本节主题息息相关。

那么问题来了，IO设备如何实现输入和输出的呢？计算机主机中，附带了用来连接显示器以及键盘等外围设备的 `连接器` 。 而连接器的内部，都连接有用来交换计算机主机同外围设备之间电流特性的 IC。如果 IC 你不明白是什么的话，这些 IC 统称为 `IO 控制器` 。

IO 是 Input/Output 的缩写。显示器、键盘等外围设备都有各自专用的 I/O 控制器。I/O 控制器中有用于临时保存输入输出数据的内存。这个内存就是 `端口(port)` 。端口你就可以把它理解为我们上述说的 港口。IO 控制器内部的内存，也被称为 `寄存器` ，不要慌，这个寄存器和内存中的寄存器不一样。CPU 内存的寄存器是用于进行数据运算处理的，而IO中的寄存器是用于临时存储数据的。

在 I/O 设备内部的 IC 中，有多个端口。由于计算机中连接着很多外围设备，因此也就有很多 I/O 控制器。当然也会有多个端口，一个 I/O 控制器可以控制多个设备，不仅仅只能控制一个。各端口之间通过 `端口号` 进行区分。

端口号也被称为 `I/O地址` 。IN 指令和 OUT 指令在端口号指定的端口和 CPU 之间进行数据的输入和输出。这跟通过内存的地址来对内存进行读写是一样的道理。



以端口为桥梁实现 CPU 和 外围设备的数据传递

测试输入和输出程序

首先让我们利用 IN 指令和 OUT 指令，来进行一个直接控制硬件的实验。假如试验的目的是让一个计算机内置的喇叭（蜂鸣器）发出声音。蜂鸣器封装在计算机内部，但它也是外围设备的一种。

用汇编语言比较繁琐，这次我们用 C 语言来实现。在大部分 C 语言的处理（编译器的种类）中，只要使用 `_asm{ 和 }` 括起来，就可以在其中记述助记符。也就是说，采用这种方式就能够使用 C 语言和汇编语言混合的源代码。

在 AT 兼容机中，蜂鸣器的默认端口号是 61H ，末尾的 H 表示的是十六进制数的意思。用 IN 指令通过该端口号输入数据，并将数据的低2位设定为 ON，然后再通过该端口号用 OUT 指令输出数据，这时蜂鸣器就会发出声音。同样的方法，将数据的低2位设定为 OFF 并输出后，蜂鸣器就停止工作。

位设定为 ON 指的是将该位设定为1，位设定为 OFF 指的是将该位设定为0 。把位设定为 ON，只需要把想要设定为 ON 的位设定为1，其他位设定为0后进行 OR 运算即可。由于这里需要把低2位置为1，因此就是和 03H 进行 OR 运算。03H 用8为二进制来表示的话是 00000011。由于即便高6位存在着具体意义。和0进行OR运算后也不会发生变化，因而就和 03H 进行 OR 运算。把位设定为 OFF，只需要把想要置 OFF 的位设定为0，其他位设定为1后进行 AND 运算即可。由于这里需要把低2位设定为0，

因此就要和 FCH 进行 AND 运算。在源代码中，FCH 是用 0FCH 来记述的。在前面加 0 是汇编语言的规定，表示的是以 A - F 这些字符开头的十六进制数是数值的意思。0FCH 用8位二进制数来表示的话是11111100。由于即便高6位存在着具体意义，和1进行 AND 运算后也不会产生变化，因而就是同 0FCH 进行 OR 运算。

```
void main(){

  // 计数器
  int i;

  // 蜂鸣器发声
  _asm{
    IN  EAX, 61H
    OR  EAX, 03H
    OUT 61H, EAX
  }

  // 等待一段时间
  for(i = 0;i < 1000000;i++);

  // 蜂鸣器停止发生
  _asm{
    IN  EAX, 61H
    AND EAX, 0FCH
    OUT 61H, EAX
  }
}
```

我们对上面的代码进行说明，main 是 C 语言程序起始位置的函数。在该函数中，有两个用 `_asm{}` 围起来的部分，它们中间有一个使用 for 循环的空循环

首先是蜂鸣器发声的部分，通过 IN EAX，61H(助记符不区分大小写)指令，把端口 61H 的数据存储到 CPU 的 EAX 寄存器中。接下来，通过 OR EAX，03H 指令，把 EAX 寄存器的低2位设定成 ON。最后，通过 OUT 61H，EAX 指令，把 EAX 寄存器的内容输出到61端口。使蜂鸣器开始发音。虽然 EAX 寄存器的长度是 32 位，不过由于蜂鸣器端口是8位，所以只需对下8位进行OR运算和AND运算就可以正常工作了。

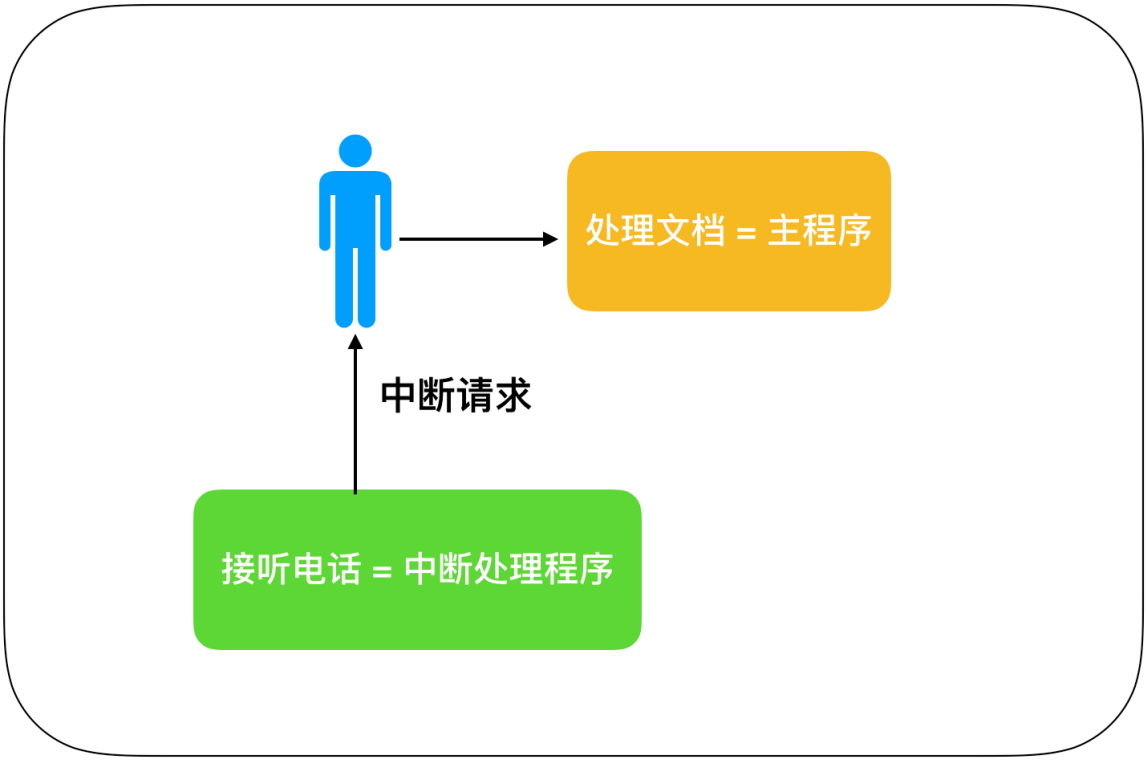
其次是一个重复100次的空循环，主要是为了在蜂鸣器开始发音和停止发音之间稍微加上一些时间间隔。因为现在计算机器的运行速度非常快，哪怕是 100 万次循环，也几乎是瞬时间完成的。

然后是用来控制器蜂鸣器停止发声的部分。首先，通过 IN EAX，61H 指令，把端口 61H 的数据存储到 CPU 的 EAX 寄存器中。接下来，通过 AND EAX，0FCH 指令，把 EAX 寄存器的低2位设定为 OFF。最后，通过 OUT 61H，EAX 指令，把寄存器的 EAX 内容输出到61号端口，使蜂鸣器停止发音。

外围设备的中断请求

`IRQ(Interrupt Request)` 代表的就是中断请求。IRQ 用来暂停当前正在运行的程序，并跳转到其他程序运行的必要机制。该机制被称为 `处理中断` 。中断处理在硬件控制中担当着重要的角色。因为如果没有中断处理，就有可能无法顺畅进行处理的情况。

从中断处理开始到请求中断的程序(中断处理程序)运行结束之前，被中断的程序(主程序)的处理是停止的。这种情况就类似于在处理文档的过程中有电话打进来，电话就相当于是中断处理。假如没有中断处理的发生，就必须等到文档处理完成后才能够接听电话。由此可见，中断处理有着巨大的价值，就像是接听完电话后会返回原来的文档作业一样，中断程序处理完成后，也会返回到主程序中继续。

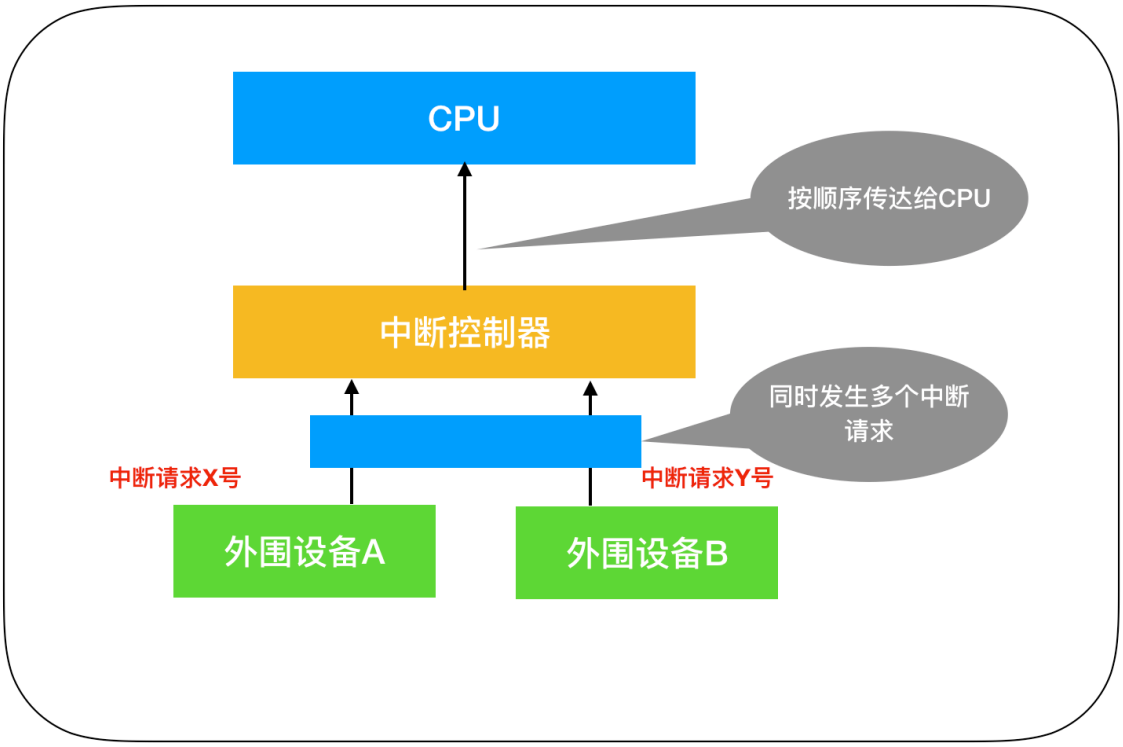


中断请求示意图

实施中断请求的是连接外围设备的 I/O 控制器，负责实施中断处理的是 CPU，外围设备的中断请求会使用不同于 I/O 端口的其他编号，该编号称为 `中断编号` 。在控制面板中查看软盘驱动器的属性时，IRQ 处现实的数值是 06，表示的就是用06号来识别软盘驱动器发出的请求。还有就是操作系统以及 `BIOS` 则会提供响应中断编号的中断处理程序。

> BIOS(Basic Input Output System): 位于计算机主板或者扩张卡上内置的 ROM 中，里面记录了用来控制外围设备的程序和数据。

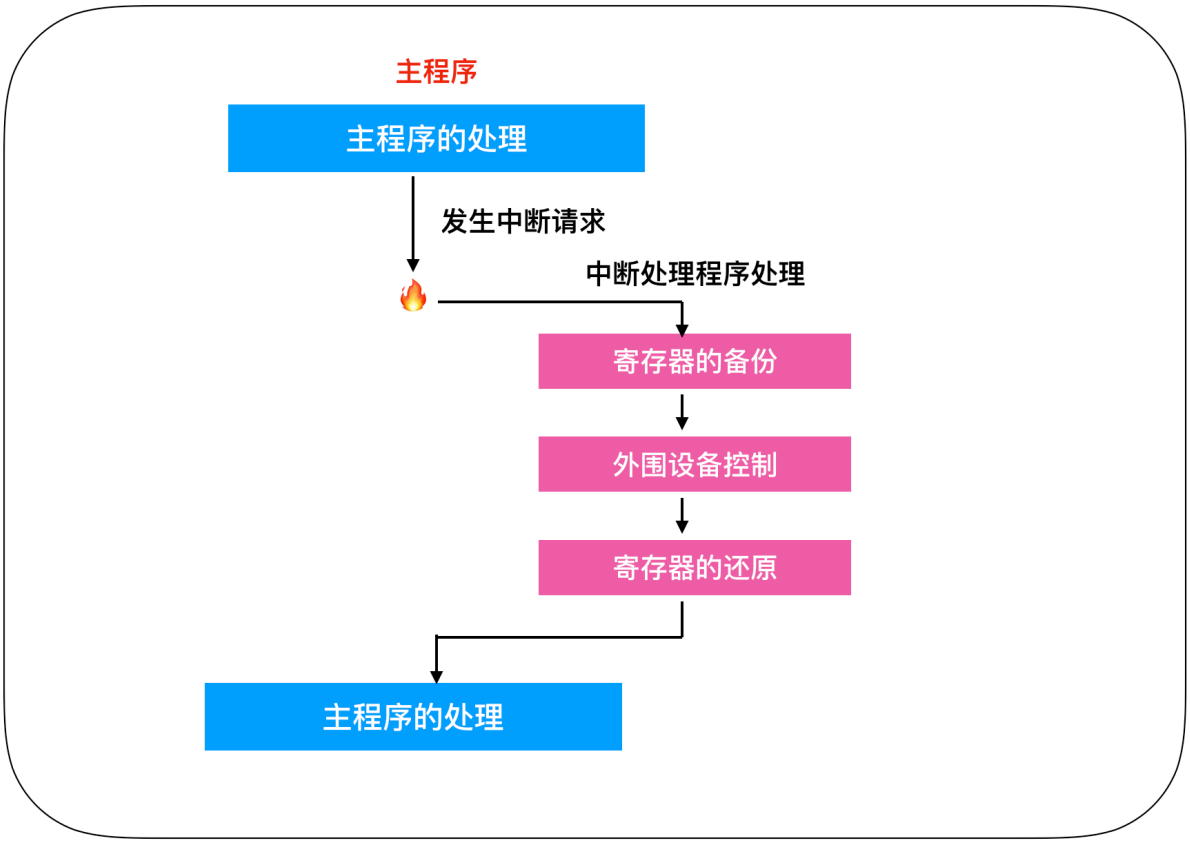
假如有多个外围设备进行中断请求的话， CPU 需要做出选择进行处理，为此，我们可以在 I/O 控制器和 CPU 中间加入名为 `中断控制器` 的 IC 来进行缓冲。中断控制器会把从多个外围设备发出的中断请求有序的传递给 CPU。中断控制器的功能相当于就是缓冲。下面是中断控制器功能的示意图



中断控制器的功能

CPU 在接受到中断请求后，会把当前正在运行的任务中断，并切换到中断处理程序。中断处理程序的第一步处理，就是把 CPU 所有寄存器的数值保存到内存的栈中。在中断处理程序中完成外围设备的输入和输出后，把栈中保存的数值还原到 CPU 寄存器中，然后再继续进行对主程序的处理。

假如 CPU 寄存器数值还没有还原的话，就会影响到主程序的运行，甚至还有可能会使程序意外停止或发生运行时异常。这是因为主程序在运行过程中，会用到 CPU 寄存器进行处理，这时候如果突然插入其他程序的运行结果，此时 CPU 必然会受到影响。所以，在处理完中断请求后，各个寄存器的值必须要还原。只要寄存器的值保持不变，主程序就可以像没有发生过任何事情一样继续处理。



请求中断的处理

用中断来实现实时处理

中断是指计算机运行过程中，出现某些意外情况需主机干预时，机器能自动停止正在运行的程序并转入处理新情况的程序，处理完毕后又返回原被暂停的程序继续运行。

在程序的运行过程中，几乎无时无刻都会发生中断，其原因就是为了实时处理外部输入的数据，虽然程序也可以在不会中断的基础上处理外部数据，但是那种情况下，主程序就会频繁的检查外围设备是否会有数据输入。由于外围设备会有很多个，因此有必要按照顺序来调查。按照顺序检查多个外围设备的状态称为 `轮询` 。对于计算机来说，这种采用轮询的方式不是很合理，如果你正在检查是否有鼠标输入，这时候发生了键盘输入该如何处理呢？结果必定会导致文字的实时处理效率。所以即时的中断能够提高程序的运行效率。

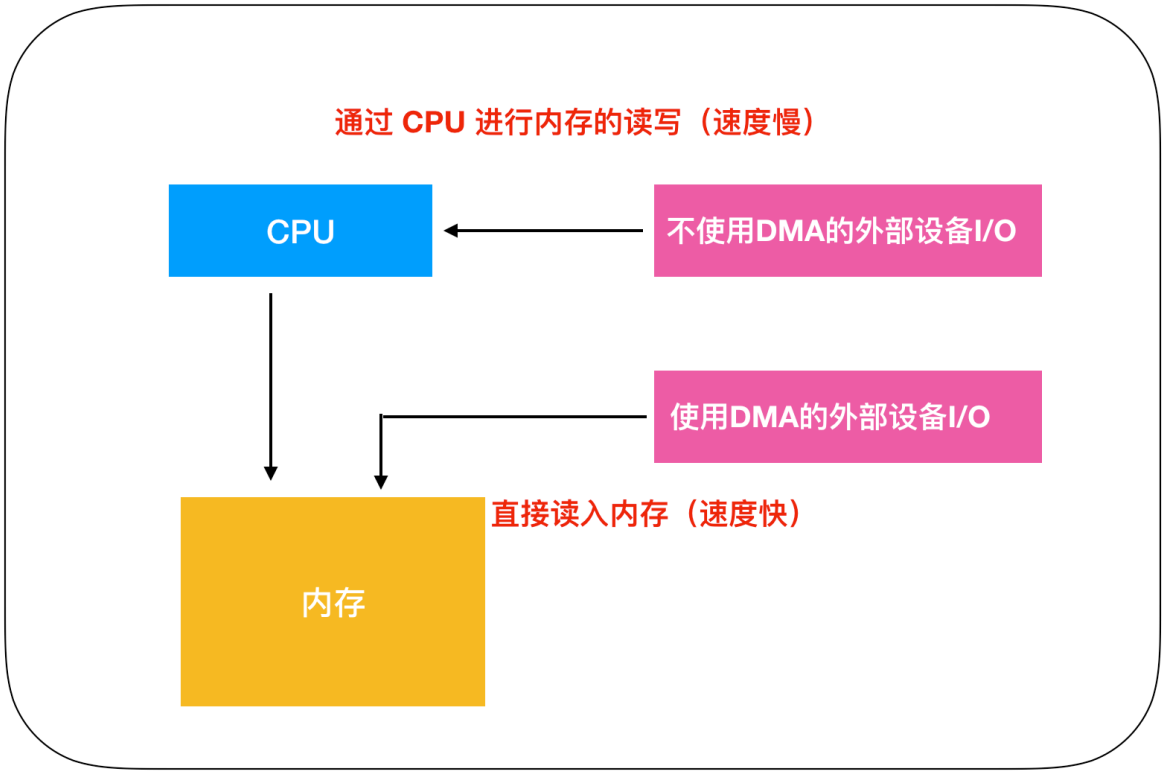
上面只是中断的一种好处，下面汇总一下利用中断能够带来的正面影响

- 提高计算机系统效率。计算机系统中处理机的工作速度远高于外围设备的工作速度。通过中断可以协调它们之间的工作。当外围设备需要与处理机交换信息时，由外围设备向处理机发出中断请求，处理机及时响应并作相应处理。不交换信息时，处理机和外围设备处于各自独立的并行工作状态。
- 维持系统可靠正常工作。现代计算机中，程序员不能直接干预和操纵机器，必须通过中断系统向操作系统发出请求，由操作系统来实现人为干预。主存储器中往往有多道程序和各自的存储空间。在程序运行过程中，如出现越界访问，有可能引起程序混乱或相互破坏信息。为避免这类事件的发生，由存储管理部件进行监测，一旦发生越界访问，向处理机发出中断请求，处理机立即采取保护措施。
- 满足实时处理要求。在实时系统中，各种监测和控制装置随机地向处理机发出中断请求，处理机随时响应并进行处理。
- 提供故障现场处理手段。处理机中设有各种故障检测和错误诊断的部件，一旦发现故障或错误，立

即发出中断请求，进行故障现场记录和隔离，为进一步处理提供必要的依据。

利用 DMA 实现短时间内大量数据传输

上面我们介绍了 I/O 处理和中断的关系，下面我们来介绍一下另外一个机制，这个机制就是 `DMA(Direct Memory Access)` 。DMA 是指在不通过 CPU 的情况下，外围设备直接和主存进行数据传输。磁盘等硬件设备都用到了 DMA 机制，通过 DMA，大量数据可以在短时间内实现传输，之所以这么快，是因为 CPU 作为中介的时间被节省了，下面是 DMA 的传输过程



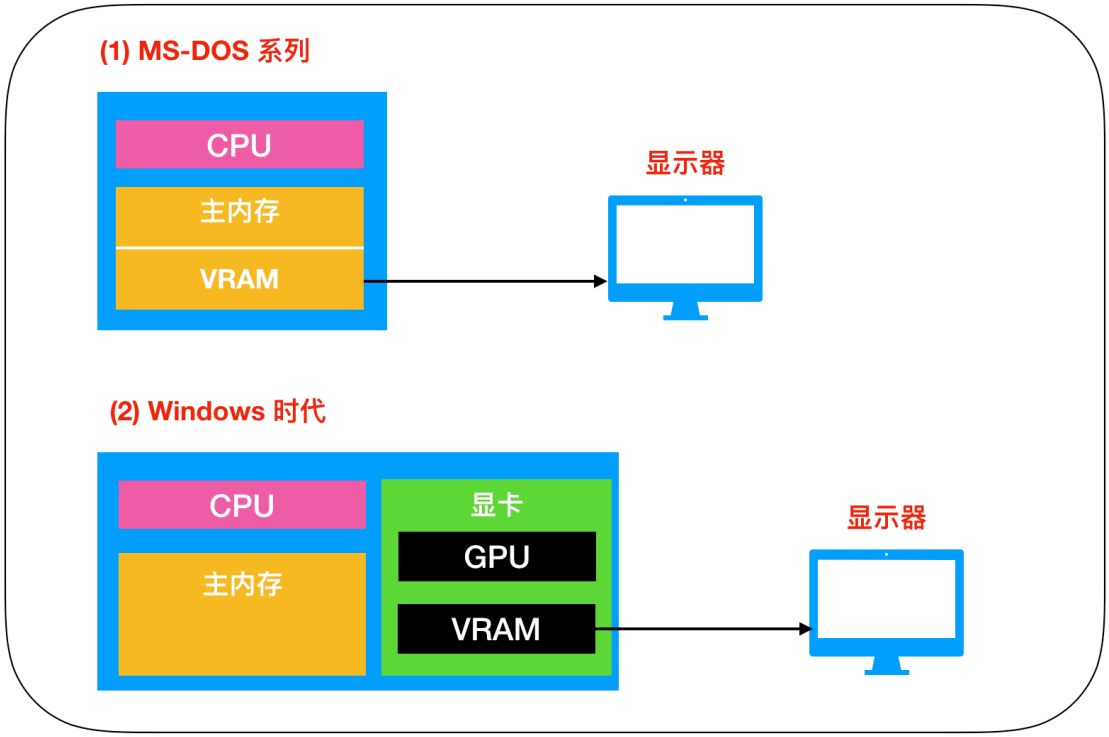
使用 DMA 的外围设备和不使用 DMA 的外部设备

I/O 端口号、IRQ、DMA 通道可以说是识别外围设备的3点组合。不过，IRQ、DMA 通道并不是所有外围设备都具备的。计算机主机通过软件控制硬件时所需要的信息的最低限，是外围设备的 I/O 端口号。IRQ 只对需要中断处理的外围设备来说是必须的，DMA 通道则只对需要 DMA 机制的外围设备来说是必须的。假如多个外围设备都设定成相同的端口号、IRQ 和 DMA 通道的话，计算机就无法正常工作，会提示 `设备冲突` 。

文字和图片的显示机制

你知道文字和图片是如何显示出来的吗？事实上，如果用一句话来简单的概括一下该机制，那就是显示器中显示的信息一直存储在某内存中。该内存称为 `VRAM(Video RAM)` 。在程序中，只要往 VRAM 中写入数据，该数据就会在显示器中显示出来。实现该功能的程序，是由操作系统或者 BIOS 提供，并借助中断来进行处理。

在 `MS-DOS` 时代，对于大部分计算机来说，VRAM 都是主内存的一部分。在现代计算机中， `显卡` 等专用硬件中一般都配置有与主内存相独立的 VRAM 和 GPU（Graphics Processing Unit），也叫做图形处理器或者图形芯片。这是因为，对经常描绘图形的 windows 来说，数百兆的 VRAM 都是必需的。



VRAM 中写入的数据被显示在显示器上

用软件来控制硬件听起来好像很难，但实际上只是利用输入输出指令同外围设备进行输入输出而已。中断处理是根据需要来使用的功能选项。DMA 则直接交给对应的外围设备即可。

虽然计算机领域新技术在不断涌现，但是计算机所能处理的事情，始终只是对输入的数据进行运算，并把结果输出，这一点是永远不会发生变化的。