练识课堂 -- 01阶段 -- 编程原理

1、编程语言概述

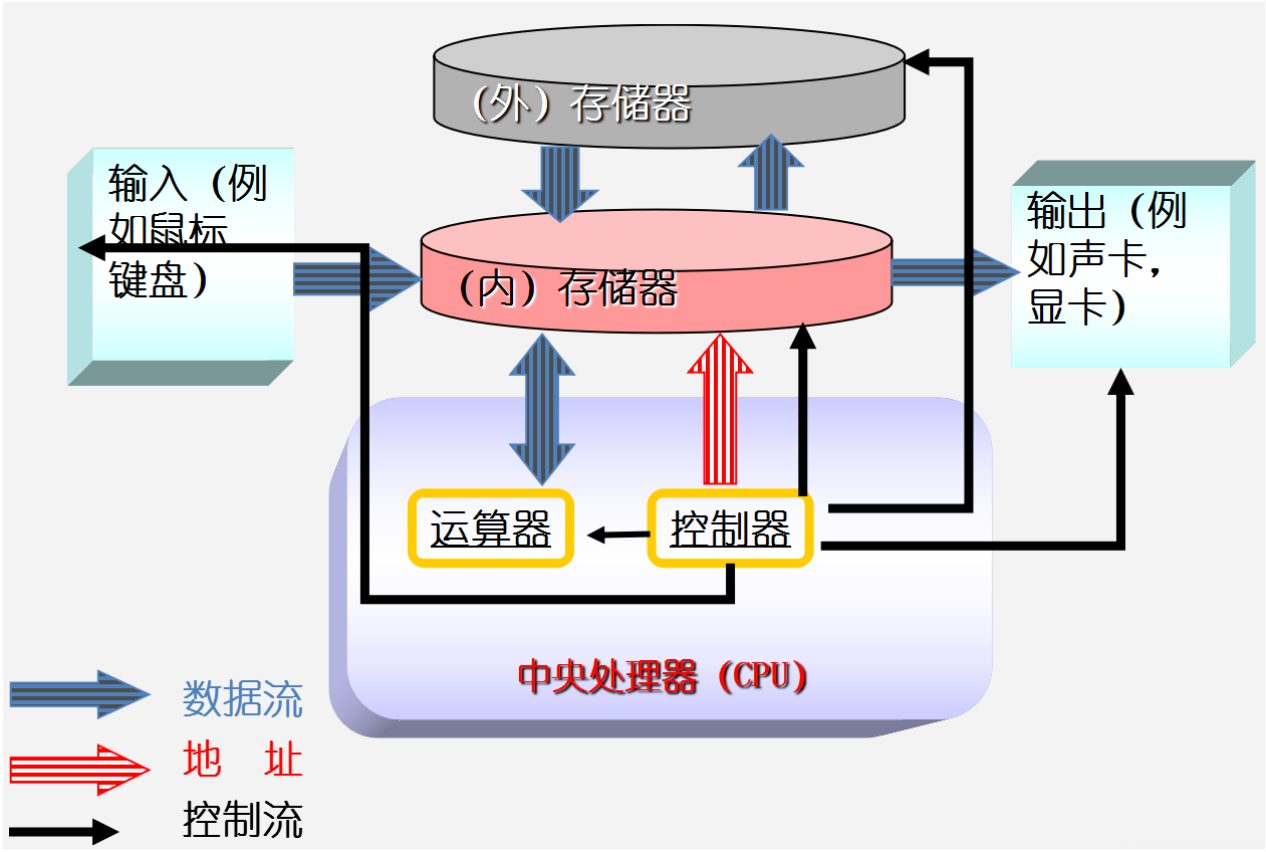
1、1 什么是编程语言

> 一提到语言这个词语，自然会想到的是像英语、汉语等这样的自然语言，因为它是人和人交换信息不可缺少的工具。
>
> 而今天计算机遍布了我们生活的每一个角落，除了人和人的相互交流之外，我们必须和计算机交流。
>
> 用什么的什么样的方式和计算机做最直接的交流呢？人们自然想到的是最古老也最方便的方式——语言，而编程语言就是人和计算机交流的一种语言。
>
> 语言是用来交流沟通的。有一方说，有另一方听，必须有两方参与，这是语言最重要的功能：
>
> - 说的一方传递信息，听的一方接收信息；
> - 说的一方下达指令，听的一方遵循命令做事情。
>
> 语言是人和人交流，编程语言是人和机器交流。只是，人可以不听另外一个人，但是，计算机是无条件服从。
>
> 语言有独特的语法规则和定义，双方必须遵循这些规则和定义才能实现真正的交流。

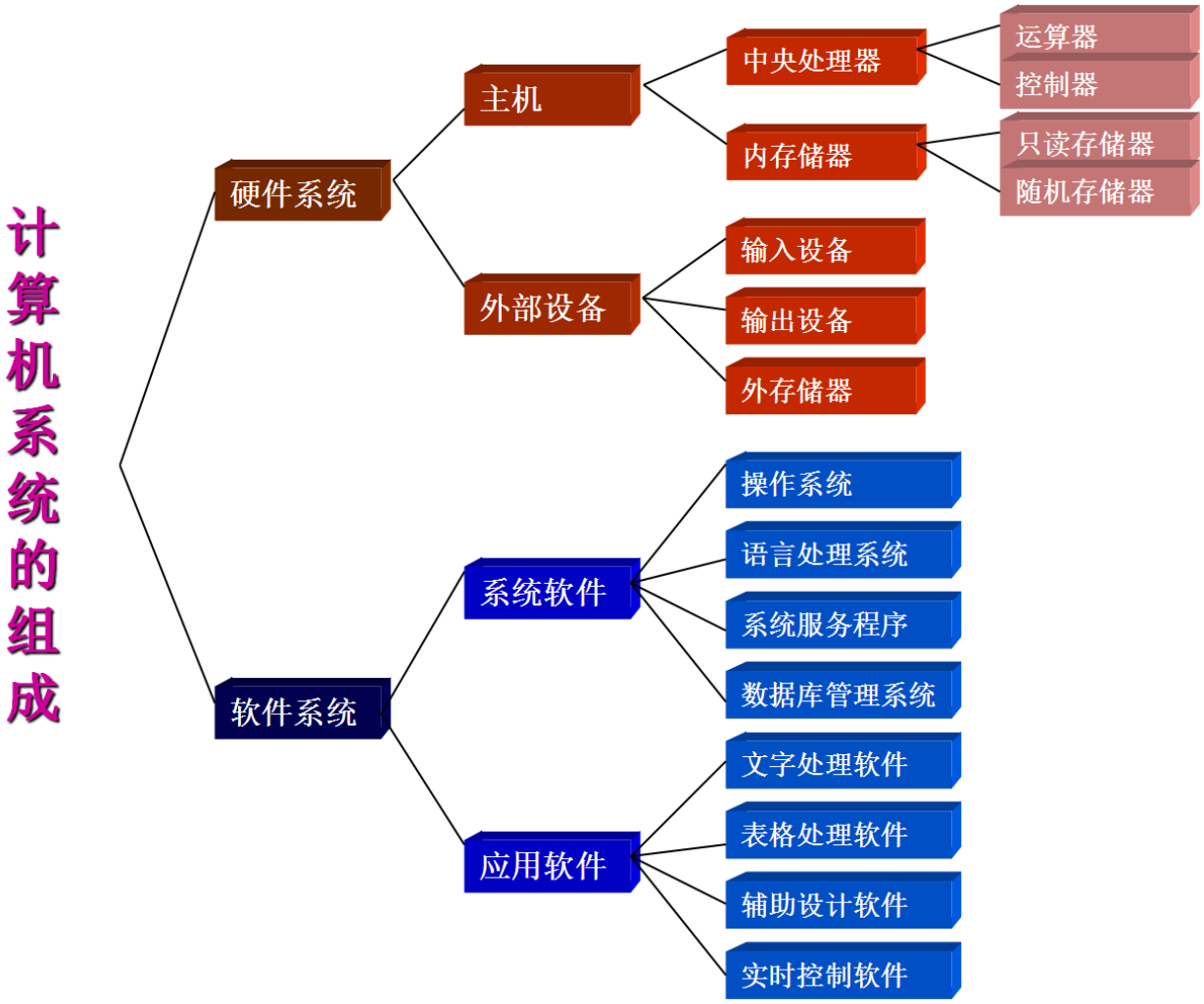
1、2 计算机介绍

1、2、1 计算机结构组成

> 计算机的基本组成:
>
> - 内外存储器:　　实现记忆功能的部件用来存放计算程序及参与运算的各种数据
> - CPU运算器:　负责数据的算术运算和逻辑运算即数据的加工处理
> - CPU控制器:　　负责对程序规定的控制信息进行分析,控制并协调输入,输出操作或内存访问
> - 输入设备:　　实现计算程序和原始数据的输入
> - 输出设备:　　实现计算结果输出


(外）存储器
输入（例如鼠标键盘）
(内）存储器
输出（例如声卡，显卡）
运算器
控制器
中央处理器（CPU）
数据流
地　址
控制流

1、2、2 计算机系统组成


计算机系统的组成
硬件系统
主机
中央处理器
运算器
控制器
内存储器
只读存储器
随机存储器
外部设备
输入设备
输出设备
外存储器
软件系统
系统软件
操作系统
语言处理系统
系统服务程序
数据库管理系统
应用软件
文字处理软件
表格处理软件
辅助设计软件
实时控制软件

1、2、3 程序和指令

> - 指令是对计算机进行程序控制的最小单位。
> - 所有的指令的集合称为计算机的指令系统。
> - 程序是为完成一项特定任务而用某种语言编写的一组指令序列。

1、3 语言发展历程

1、3、1 机器语言

> 计算机的大脑或者说心脏就是CPU，它控制着整个计算机的运作。每种CPU，都有自己的指令系统。这个指令系统，就是该CPU的机器语言。
>
> 机器语言是一组由0和1系列组成的指令码，这些指令码，是CPU制作厂商规定出来的，然后发布出来，请程序员遵守。
>
> 要让计算机干活，就得用机器语言(二级制数)去命令它。这样的命令，不是一条两条，而是上百条。而且不同型号的计算机其机器语言是不相通的，按着一种计算机的机器指令编制的程序，不能在另一种计算机上执行。

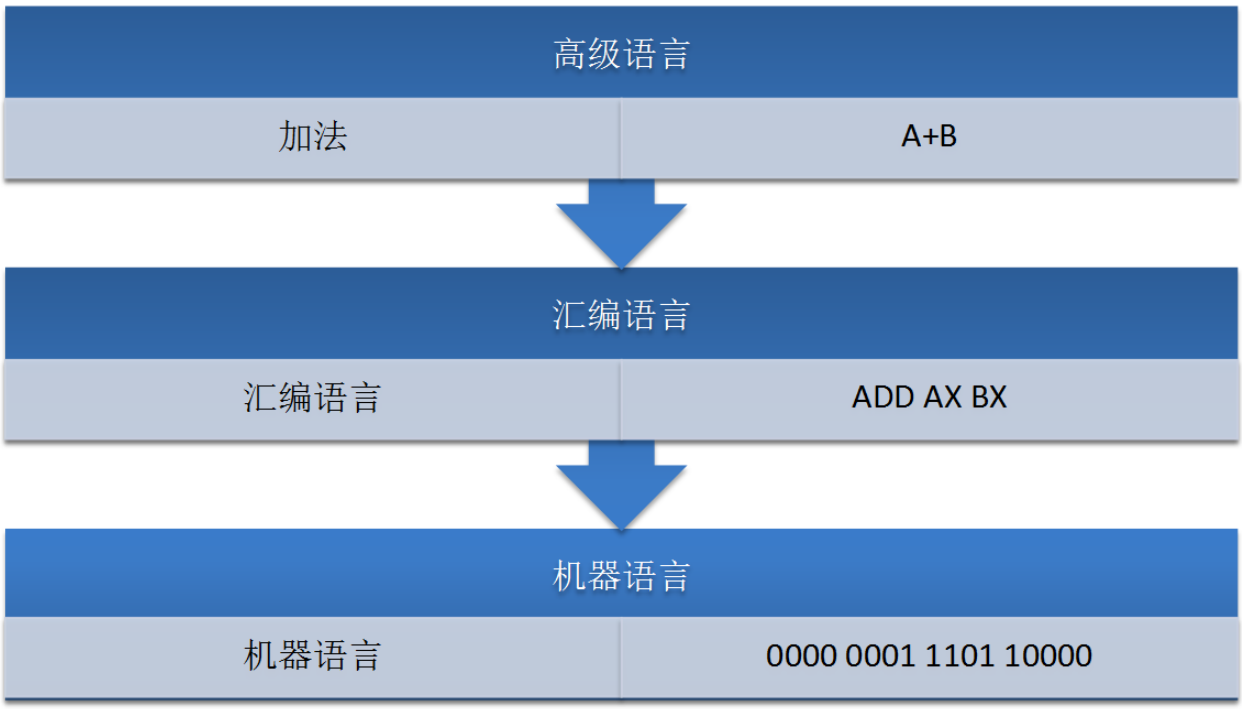
1、3、2 汇编语言和编译器

> 机器语言编程是不是很令人烦恼呢，终于出现了汇编语言，就是一些标识符取代0与1。一门人类可以比较轻松认识的编程语言。
>
> 只是这门语言计算机并不认识，所以人类还不能用这门语言命令计算机做事情。这正如如何才能让中国人说的话美国人明白呢？——翻译！
>
> 所以，有一类专门的程序，既认识机器语言，又认识汇编语言，也就是编译器，将标识符换成0与1，知道怎么把汇编语言翻译成机器语言。

1、3、3 高级语言

> 汇编语言和机器语言都是面向机器的，机器不同，语言也不同。既然有办法让汇编语言翻译成机器语言，难道就不能把其他更人性化的语言翻译成机器语言?
>
> 1954年，Fortran语言出现了，其后相继出现了其他的类似语言。这批语言，使程序员摆脱了计算机硬件的限制，把主要精力放在了程序设计上，不在关注低层的计算机硬件。这类语言，称为高级语言。
>
> 同样的，高级语言要被计算机执行，也需要一个翻译程序将其翻译成机器语言，这就是编译程序，简称编译器。
>
> 这类高级语言解决问题的方法是分析出解决问题所需要的步骤，把程序看作是数据被加工的过程。基于这类方法的程序设计语言成为面向过程的语言。C语言就是这种面向过程的程序设计语言。

1、3、4 语言的层次



2、编译过程

2、1 编译步骤

> **编译步骤**
>
> - 词法分析阶段：读入源程序，对构成源程序的字符流进行扫描和分解，识别出单词。
> - 语法分析阶段：机器通过词法分析，将单词序列分解成不同的语法短语，确定整个输入串能够构成语法上正确的程序。
> - 语义分析阶段：检查源程序上有没有语义错误，在代码生成阶段收集类型信息
> - 中间代码生成阶段：在进行了上述的语法分析和语义分析阶段的工作之后,有的编译程序将源程序变成一种内部表示形式
> - 代码优化：这一阶段的任务是对前一阶段产生的中间代码进行变换或进行改造,目的是使生成的目标代码更为高效,即省时间和省空间
> - 目标代码生成：这一阶段的任务是把中间代码变换成特定机器上的绝对指令代码或可重定位的指令代码或汇编指令代码

2、2 编译器

> 使用编辑器编写程序，由编译器编译后才可以运行，编译器是将易于编写、阅读和维护的高级计算机语言翻译为计算机能解读、运行的低级机器语言的程序。
>
> 编译器就是将 “一种语言（通常为高级语言）” 翻译为 “另一种语言（通常为低级语言）” 的程序。
>
> 一个现代编译器的主要工作流程：
>
> **源代码**(source code) → **预处理器**(preprocessor) → **编译器** (compiler) → **目标代码**(object code) → **链接器**(Linker) → **可执行程序**(executables)

2、2、1 GCC编译器介绍

> GCC（GNU Compiler Collection，GNU 编译器套件），是由 GNU 开发的编程语言编译器。GCC原本作为GNU操作系统的官方编译器，现已被大多数类Unix操作系统（如Linux、BSD、Mac OS X等）采纳为标准的编译器，GCC同样适用于微软的Windows。
>
> GCC最初用于编译C语言，随着项目的发展gcc已经成为了能够编译C、C++、Java、Ada、fortran、Object C、Object C++、Go语言的编译器大家族。

2、2、2 GCC编译流程

> 编译命令格式：
>
> gcc [-option1] ...
>
> g++ [-option1] ...
>
> - 命令、选项和源文件之间使用空格分隔
> - 一行命令中可以有零个、一个或多个选项
> - 文件名可以包含文件的绝对路径，也可以使用相对路径
>
> 如果命令中不包含输出可执行文件的文件名，可执行文件的文件名会自动生成一个默认名，Linux平台为a.out，Windows平台为a.exe
>
> GCC、g++编译常用选项说明：

选项	含义
-o file	指定生成的输出文件名为file
-E	只进行预处理
-S	只进行预处理和编译
-c	只进行预处理、编译和汇编

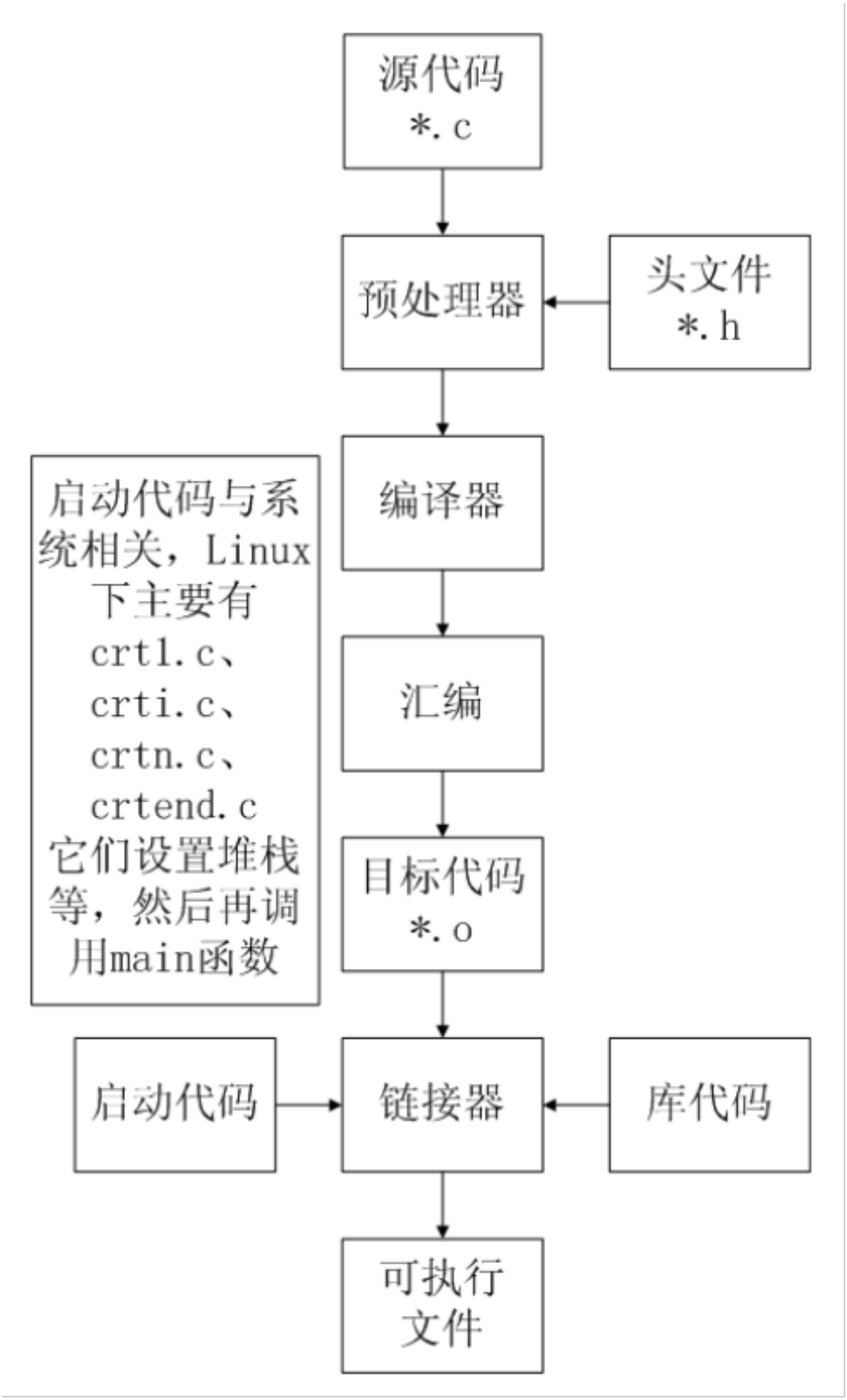
2、2、3 注意事项

> Linux编译后的可执行程序只能在Linux运行，Windows编译后的程序只能在Windows下运行。
>
> 64位的Linux编译后的程序只能在64位Linux下运行，32位Linux编译后的程序只能在32位的Linux运行。
>
> 64位的Windows编译后的程序只能在64位Windows下运行，32位Windows编译后的程序可以在64位的Windows运行。

2、3 编译过程

> 程序编译步骤
>
> 代码编译成可执行程序经过4步：
>
> - 预处理：宏定义展开、头文件展开、条件编译等，同时将代码中的注释删除，这里并不会检查语法
> - 编译：检查语法，将预处理后文件编译生成汇编文件
> - 汇编：将汇编文件生成目标文件(二进制文件)

- 链接：程序是需要依赖各种库的，所以编译之后还需要把库链接到最终的可执行程序中去



2、3、1 分步编译

- 预处理： gcc -E hello.c -o hello.i
- 编 译： gcc -S hello.i -o hello.s
- 汇 编： gcc -c hello.s -o hello.o

- 链 接： gcc hello.o -o hello

选项	含义
-E	只进行预处理
-S(大写)	只进行预处理和编译
-c(小写)	只进行预处理、编译和汇编
-o file	指定生成的输出文件名为 file

文件后缀	含义
.c	C 语言文件
.i	预处理后的 C 语言文件
.s	编译后的汇编文件
.o	编译后的目标文件

2、3、2 一步编译

gcc hello.c -o demo（还是经过：预处理、编译、汇编、链接的过程）

3、 汇编语言

汇编语言的主体是汇编指令，汇编指令和机器指令的差别在于指令的表示方法上，汇编指令是机器指令的助记符，汇编指令是更便于记忆的一种书写格式。它较为有效地解决了机器指令编写程序难度大的问题，汇编语言与人类语言更接近，便于阅读和记忆。

使用编译器，可以把汇编程序转译成机器指令程序。举例如下：

```
机器指令：   1000100111011000
汇编指令：   MOV  AX, BX
```

3、1 常见汇编指令

助记符	说明	助记符	说明
MOV	传送（分配）数值	MUL	两个数值相乘
ADD	两个数值相加	JMP	跳转到一个新位置
SUB	从一个数值中减去另一个数值	CALL	调用一个子程序
PUSH	将 Stack 所指向的地址写入寄存器	POP	用于取出 Stack 最近写入的值
RET	用于终止当前函数的执行并返回		

汇编语言程序：

```
.data                                ;此为数据区
sum DWORD 0                          ;定义名为sum的变量

.code                                ;此为代码区
main PROC
mov eax,3                        ;将数字3送入而eax寄存器
add eax,4                        ;eax寄存器加4
mov sum,eax

INVOKE ExitProcess,0             ;结束程序
main ENDP
```

C语言程序：

```
int add(int a, int b) {
return a + b;
}

int main() {
return add(3, 4);
}
```

GCC将这个程序转成汇编语言：

```
$ gcc -S demo.c
```

汇编语言程序：

```
_add:
push    %ebx
mov     %eax, [%esp+8]
mov     %ebx, [%esp+12]
add     %eax, %ebx
pop     %ebx
ret

_main:
push    3
push    4
call    _add
add     %esp, 8
ret
```

3、2 C语言嵌套汇编代码

```
#include <stdio.h>

int main()
{
  //定义整型变量a, b, c
  int a;
  int b;
  int c;

  __asm
  {
    mov a, 3  //3的值放在a对应内存的位置
    mov b, 4  //4的值放在b对应内存的位置
    mov eax, a  //把a内存的值放在eax寄存器
    add eax, b  //eax和b相加，结果放在eax
    mov c, eax  //eax的值放在c中
  }

  printf("%d\n",  c);//把c的值输出

  return 0;//成功完成
}
```

3、3 反汇编

C语言代码：

```
#include <stdio.h>

int main()
{
  //定义整型变量a, b, c
  int a;
  int b;
  int c;

  a = 3;
  b = 4;
  c = a + b;

  printf("%d\n",  c);//把c的值输出

  return 0;//成功完成
}
```

查看方式：

- 设置断点F9，进入程序调试
- 选择反汇编按钮

- 根据汇编代码分析程序

在Go语言中也可查看汇编语言代码：

```
#将Go语言程序显示为汇编语言
go build -gcflags "-N -l"
go tool objdump -s 'main.Demo' -S ./Go程序.exe
```

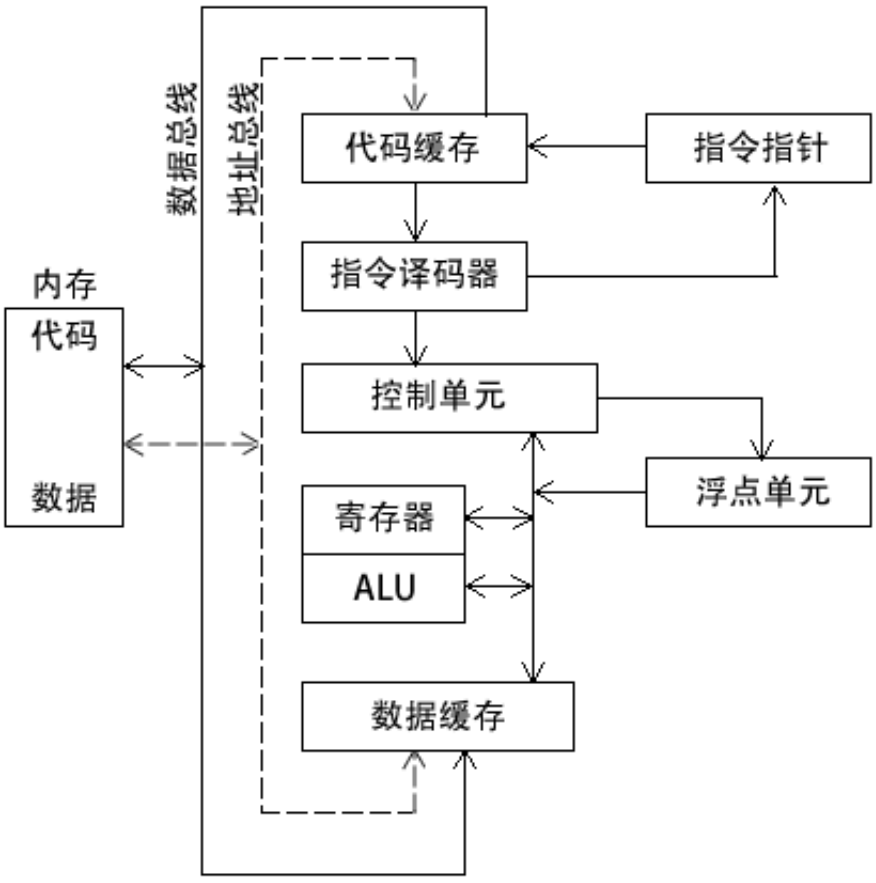
因为Go语言本身会对程序代码进行封装，所以查看的汇编文件会比较大，可以根据“包名.函数名”进行查找后在查看。

Go语言本身支持使用CGO与C语言混合编程，所以可以在Go语言中嵌套汇编语言。

4、CPU结构

中央处理器（CPU，central processing unit）作为计算机系统的运算和控制核心，是信息处理、程序运行的最终执行单元。

下图是一个典型 CPU 中的数据流框图。该图表现了在指令执行周期中相互交互部件之间的关系。在从内存读取程序指令之前，将其地址放到地址总线上。然后，内存控制器将所需代码送到数据总线上，存入代码高速缓存 (code cache)。指令指针的值决定下一条将要执行的指令。指令由指令译码器分析，并产生相应的数值信号送往控制单元，其协调 ALU 和浮点单元。虽然图中没有画出控制总线，但是其上传输的信号用系统时钟协调不同 CPU 部件之间的数据传输。



4、1 CPU构成

CPU主要由**运算器**、**控制器**、**寄存器组**和**内部总线**构成。

- 运算器：

  由算术逻辑单元ALU、通用寄存器、数据暂存器等组成。程序状态字寄存器接受从控制器送来的命令并执行相应的动作，主要负责对数据的加工和处理。

- 控制器：

  由程序计数器PC、指令寄存器IR、地址寄存器AR、数据寄存器DR、指令译码器等。负责对程序规定的控制信息进行分析,控制并协调输入,输出操作或内存访问

- 寄存器：

  寄存器的功能是存储二进制代码，它是由具有存储功能的触发器组合起来构成的。

- 总线：

  总线是一种内部结构，它是cpu、内存、输入、输出设备传递信息的公用通道，主机的各个部件通过总线相连接，外部设备通过相应的接口电路再与总线相连接。

4、2 寄存器

CPU 本身只负责运算，不负责储存数据。数据一般都储存在内存之中，CPU 要用的时候就去内存读写数据。但是，CPU 的运算速度远高于内存的读写速度，为了避免被拖慢，CPU 都自带一级缓存和二级缓存。基本上，CPU 缓存可以看作是读写速度较快的内存。

但是，CPU 缓存还是不够快，另外数据在缓存里面的地址是不固定的，CPU 每次读写都要寻址也会拖慢速度。因此，除了缓存之外，CPU 还自带了寄存器（register），用来储存最常用的数据。也就是说，那些最频繁读写的数据（比如循环变量），都会放在寄存器里面，CPU 优先读写寄存器，再由寄存器跟内存交换数据。

寄存器名字

8位	16位	32位	64位
A	AX	EAX	RAX
B	BX	EBX	RBX
C	CX	ECX	RCX
D	DX	EDX	RDX

4、3 寄存器、缓存、内存三者关系

按与CPU远近来分，离得最近的是寄存器，然后缓存(CPU缓存)，最后内存。

CPU计算时，先预先把要用的数据从硬盘读到内存，然后再把即将要用的数据读到寄存器。于是CPU〈---〉寄存器〈---〉内存，这就是它们之间的信息交换。

那为什么有缓存呢？因为如果经常操作内存中的同一址地的数据，就会影响速度。于是就在寄存器与内存之间设置一个缓存。

因为从缓存提取的速度远高于内存。当然缓存的价格肯定远远高于内存，不然的话，机器里就没有内存的存在。

> 由此可以看出，从远近来看：CPU〈---〉寄存器〈---〉缓存 〈---〉内存。

5、 进制

> 进制也就是进位制，是人们规定的一种进位方法。 对于任何一种进制—X进制，就表示某一位置上的数运算时是逢X进一位。 十进制是逢十进一，十六进制是逢十六进一，二进制就是逢二进一，以此类推，x进制就是逢x进位。

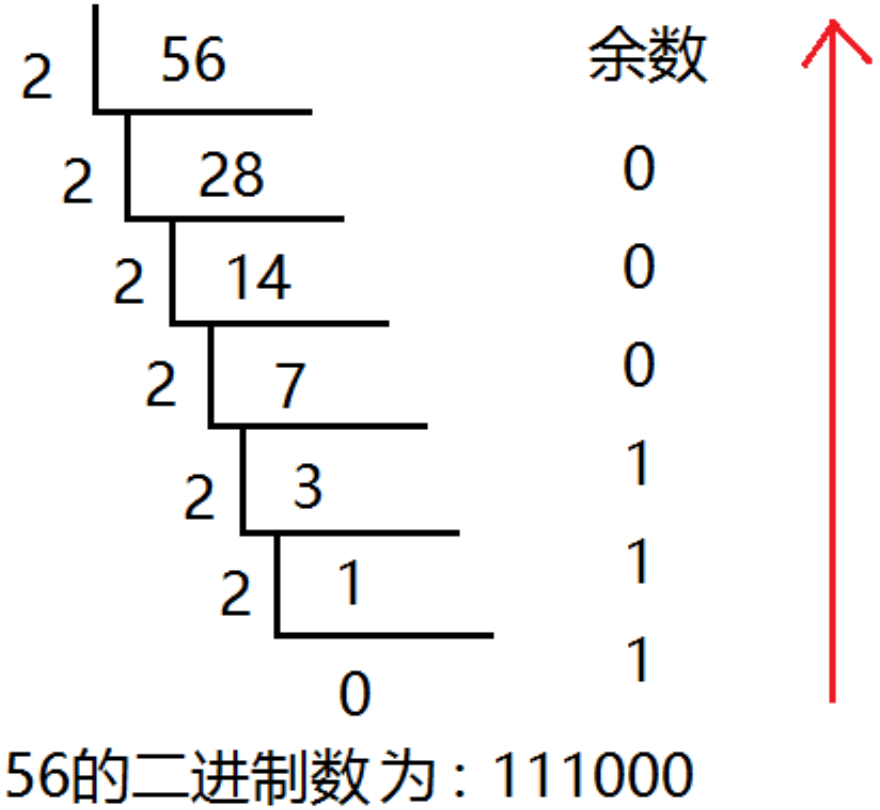
十进制	二进制	八进制	十六进制
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

5、1 二进制

> 二进制是计算技术中广泛采用的一种数制。二进制数据是用0和1两个数码来表示的数。它的基数为2，进位规则是“逢二进一”，借位规则是“借一当二”。
>
> 当前的计算机系统使用的基本上是二进制系统，数据在计算机中主要是以补码的形式存储的。

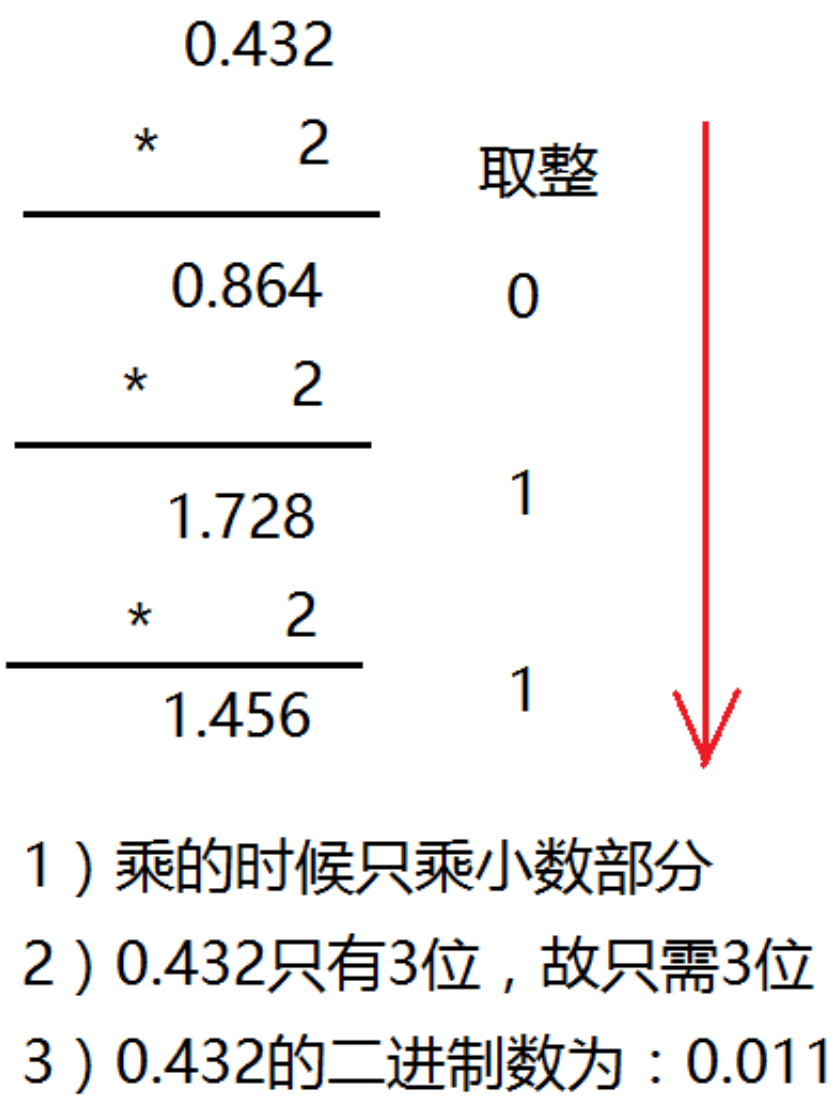
术语	含义
bit(比特)	一个二进制代表一位，一个位只能表示0或1两种状态。
Byte(字节)	一个字节为8个二进制，称为8位，计算机中存储的最小单位是字节。
WORD(双字节)	2个字节，16位
DWORD	两个WORD，4个字节，32位
1b	1bit，1位
1B	1Byte,1字节，8位
1k，1K	1024B
1M(1兆)	1024k, 1024*1024
1G	1024M
1T	1024G
1Kb(千位)	1024bit,1024位
1KB(千字节)	1024Byte，1024字节
1Mb(兆位)	1024Kb = 1024 * 1024bit
1MB(兆字节)	1024KB = 1024 * 1024Byte

十进制转化二进制的方法：用十进制数除以2，分别取余数和商数，商数为0的时候，将余数倒着数就是转化后的结果。



56的二进制数为：111000

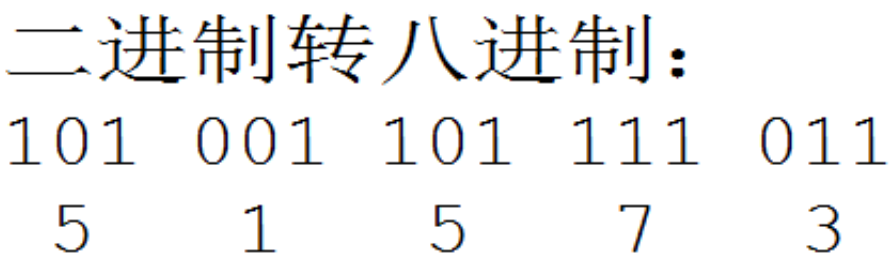
十进制的小数转换成二进制：小数部分和2相乘，取整数，不足1取0，每次相乘都是小数部分，顺序看取整后的数就是转化后的结果。

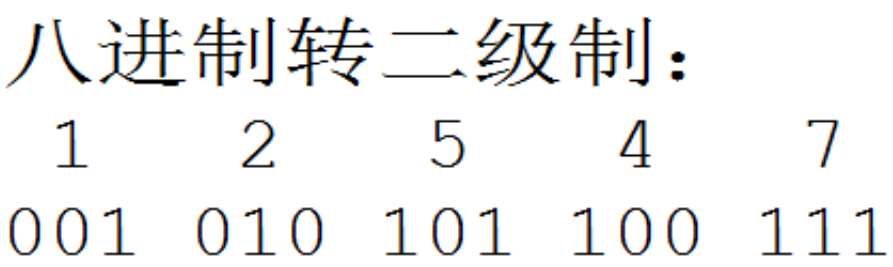


5、2 八进制

八进制，Octal，缩写OCT或O，一种以8为基数的计数法，采用0，1，2，3，4，5，6，7八个数字，逢八进1。一些编程语言中常常以数字0开始表明该数字是八进制。

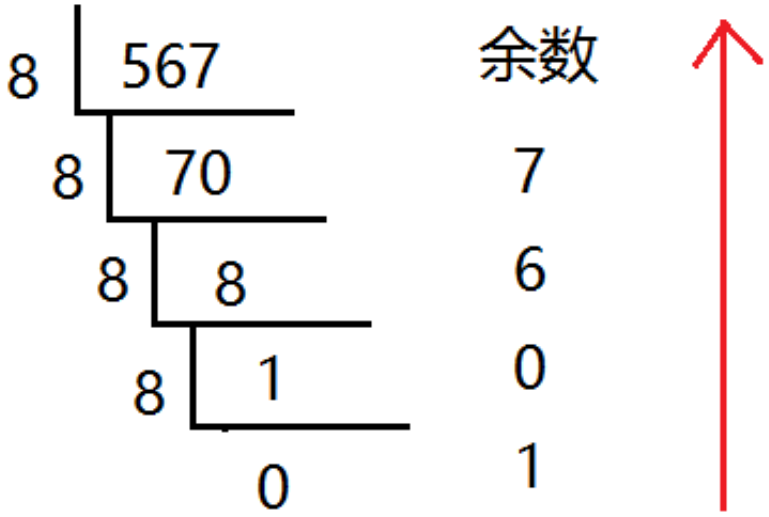
八进制的数和二进制数可以按位对应（八进制一位对应二进制三位），因此常应用在计算机语言中。





十进制转化八进制的方法：

用十进制数除以8，分别取余数和商数，商数为0的时候，将余数倒着数就是转化后的结果。



567的八进制数位：1067

5、3 十六进制

十六进制（英文名称：Hexadecimal），同我们日常生活中的表示法不一样，它由0-9，A-F组成，字母不区分大小写。与10进制的对应关系是：0-9对应0-9，A-F对应10-15。

十六进制的数和二进制数可以按位对应（十六进制一位对应二进制四位），因此常应用在计算机语言中。

二进制转十六进制：

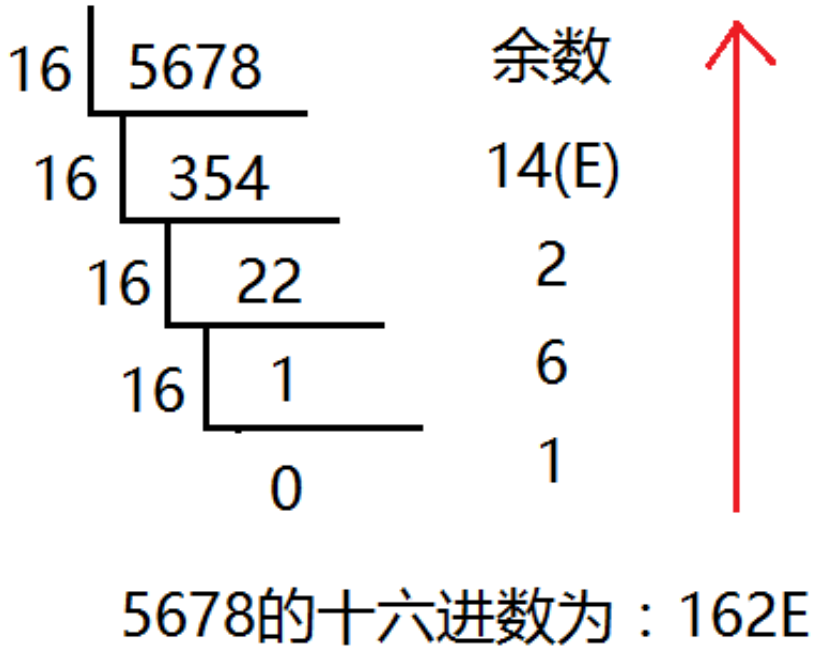
```
0101 0111 1010 1011 0101 0111 1101 0111 0000
 5    7    A    B    5    7    D    7    0
```

十六进制转二级制：

```
 A    B    4    2    D
1010 1011 0100 0010 1101
```

十进制转化十六进制的方法：

用十进制数除以16，分别取余数和商数，商数为0的时候，将余数倒着数就是转化后的结果。



5678的十六进数为：162E

5、4 进制在代码中表现形式

十进制	以正常数字1-9开头，如123
八进制	以数字0开头，如0123
十六进制	以0x开头，如0x123
二进制	C语言和Go语言都不能直接书写二进制数

C 语言代码

```c
#include <stdio.h>

int main()
{
  int a = 123;    //十进制方式赋值
  int b = 0123;   //八进制方式赋值，  以数字0开头
  int c = 0xABC;  //十六进制方式赋值
```

```
  //如果在printf中输出一个十进制数那么用%d，八进制用%o，十六进制是%x
  printf("十进制：%d\n",a );
  printf("八进制：%o\n", b);  //%o,为字母o,不是数字
  printf("十六进制：%x\n", c);

  return 0;
}
```

Go语言代码

```
func main() {
  a := 10   //十进制
  b := 010  //八进制
  c := 0x10 //十六进制
  //%d 占位符 表示输出一个十进制整型数据
  fmt.Printf("%d\n",a)
  //%o 占位符 表示输出一个十进制整型数
  fmt.Printf("%o\n",b)
  //%x %X 占位符 表示输出一个十进制整型数据
  fmt.Printf("%x\n",c)
}
```

6、 数值存储方式

6、1 原码

一个数的原码(原始的二进制码)有如下特点：

- 最高位做为符号位，0表示正,为1表示负
- 其它数值部分就是数值本身绝对值的二进制数
- 负数的原码是在其绝对值的基础上，最高位变为1

十进制数	原码
+15	0000 1111
-15	1000 1111
+0	0000 0000
-0	1000 0000

原码表示法简单易懂，与带符号数本身转换方便，只要符号还原即可，但当两个正数相减或不同符号数相加时，必须比较两个数哪个绝对值大，才能决定谁减谁，才能确定结果是正还是负，所以原码不便于加减运算。

6、2 反码

- 对于正数，反码与原码相同
- 对于负数，符号位不变，其它部分取反(1变0,0变1)

十进制数	反码
+15	0000 1111
-15	1111 0000
+0	0000 0000
-0	1111 1111

反码运算也不方便，通常用来作为求补码的中间过渡。

6、3 补码

在计算机系统中，数值一律用补码来存储。

补码特点：

- 对于正数，原码、反码、补码相同
- 对于负数，其补码为它的反码加1
- 补码符号位不动，其他位求反，最后整个数加1，得到原码

十进制数	补码
+15	0000 1111
-15	1111 0001
+0	0000 0000
-0	0000 0000

6、4 补码的意义

用8位二进制数分别表示+0和-0

十进制数	原码
+0	0000 0000
-0	1000 0000

十进制数	反码
+0	0000 0000
-0	1111 1111

不管以原码方式存储，还是以反码方式存储，0也有两种表示形式。为什么同样一个0有两种不同的表示方法呢?

但是如果以补码方式存储，补码统一了零的编码：

十进制数	补码
+0	0000 0000
-0	10000 0000由于只用8位描述，最高位1丢弃，变为0000 0000

计算9-6的结果

以原码方式相加：

十进制数	原码
9	0000 1001
-6	1000 0110

最高位的1溢出,剩余8位二进制表示的是3，正确。

$$\begin{array}{r} 0000\ 1001 \\ +1111\ 1010 \\ \hline 10000\ 0011 \end{array}$$

在计算机系统中，数值一律用补码来存储，主要原因是：

- 统一了零的编码
- 将符号位和其它位统一处理
- 将减法运算转变为加法运算
- 两个用补码表示的数相加时，如果最高位(符号位)有进位，则进位被舍弃

7 指针和内存

7、1 内存

内存含义：

- 存储器：计算机的组成中，用来存储程序和数据，辅助CPU进行运算处理的重要部分。
- 内存：内部存贮器，暂存程序/数据——掉电丢失 SRAM、DRAM、DDR、DDR2、DDR3。
- 外存：外部存储器，长时间保存程序/数据—掉电不丢ROM、ERRROM、FLASH（NAND、NOR）、硬盘、光盘。

内存是沟通CPU与硬盘的桥梁：

- 暂存放CPU中的运算数据
- 暂存与硬盘等外部存储器交换的数据

7、2 物理存储器和存储地址空间

有关内存的两个概念：物理存储器和存储地址空间。

物理存储器：实际存在的具体存储器芯片。

- 主板上装插的内存条
- 显示卡上的显示RAM芯片
- 各种适配卡上的RAM芯片和ROM芯片

存储地址空间：对存储器编码的范围。我们在软件上常说的内存是指这一层含义。

- 编码：对每个物理存储单元（一个字节）分配一个号码
- 寻址：可以根据分配的号码找到相应的存储单元，完成数据的读写

7、3 内存地址

将内存抽象成一个很大的一维字符数组。

编码就是对内存的每一个字节分配一个32位或64位的编号（与32位或者64位处理器相关）。

这个内存编号我们称之为内存地址。

内存中的每一个数据都会分配相应的地址：

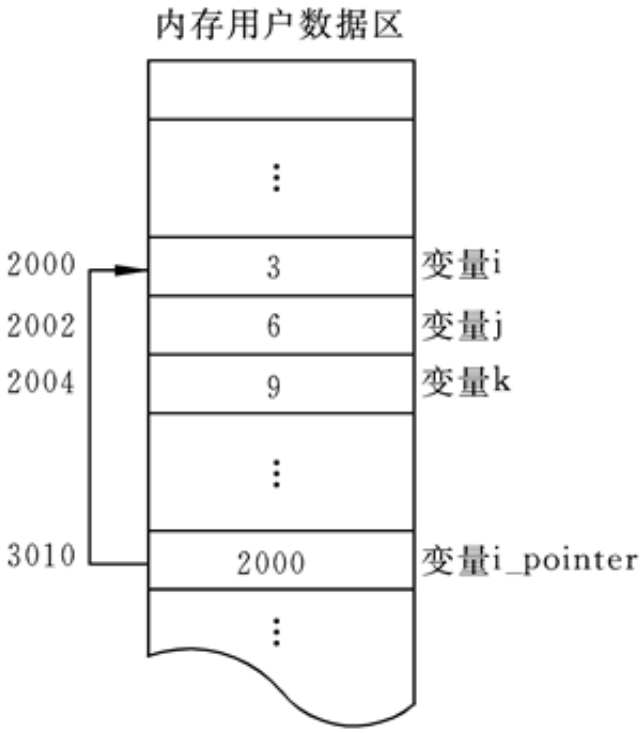
- char:占一个字节分配一个地址
- int: 占四个字节分配四个地址
- float、struct、函数、数组等

7、4 指针

只要将数据存储在内存中都会为其分配内存地址。内存地址使用十六进数据表示。

内存为每一个字节分配一个32位或64位的编号（与32位或者64位处理器相关）。

- 内存区的每一个字节都有一个编号，这就是“地址”。
- 如果在程序中定义了一个变量，在对程序进行编译或运行时，系统就会给这个变量分配内存单元，并确定它的内存地址(编号)
- 指针的实质就是内存“地址”。指针就是地址，地址就是指针。
- 指针是内存单元的编号，指针变量是存放地址的变量。
- 通常叙述时会把指针变量简称为指针，实际他们含义并不一样。



7、4、1 指针变量的定义和使用

- 指针也是一种数据类型，指针变量也是一种变量
- 指针变量指向谁，就把谁的地址赋值给指针变量
- “*”操作符操作的是指针变量指向的内存空间

C语言代码

```c
#include <stdio.h>

int main()
{
  int a = 0;
  char b = 100;
  printf("%p, %p\n", &a, &b); //打印a, b的地址

  //int *代表是一种数据类型，int*指针类型，p才是变量名
  //定义了一个指针类型的变量，可以指向一个int类型变量的地址
  int *p;
  p = &a;//将a的地址赋值给变量p，p也是一个变量，值是一个内存地址编号
  printf("%d\n", *p);//p指向了a的地址，*p就是a的值

  char *p1 = &b;
  printf("%c\n", *p1);//*p1指向了b的地址，*p1就是b的值

  return 0;
}
```

Go语言代码

> 可以通过指针变量来存储，所谓的指针变量：就是用来存储任何一个值的内存地址。

```
//定义指针变量
var 指针变量名  //默认初始值为nil  指向内存地址编号为0的空间
var 指针变量名 *数据类型 = &变量
```

```
func main() {
 var i int = 10
 //指针类型变量
 //指针变量也是变量  指针变量指向了变量的内存地址
 //对变量取地址 将结果赋值给指针变量
 var p *int = &i
 //打印指针变量p的值 同时也是i的地址
 fmt.Println(p)
}
```

7、4、2 通过指针间接修改变量的值

> C语言代码

```
int a = 0;
int b = 11;
int *p = &a;

*p = 100;
printf("a = %d, *p = %d\n", a, *p);

p = &b;
*p = 22;
printf("b = %d, *p = %d\n", b, *p);
```

> Go语言代码

```
var a int = 10

//var p *int = &a
//通过自动推导类型创建指针变量
p := &a

//通过指针间接修改变量的值
*p = 123
fmt.Println(a)
```

8 作用域

> 作用域（scope），通常来说，一段程序代码中所用到的名字并不总是有效/可用的，而限定这个名字的可用性的代码范围就是这个名字的作用域。

> 作用域的使用提高了程序逻辑的局部性，增强程序的可靠性，减少名字冲突。

8、1 局部变量

> 局部变量，一般情况下代码块{}内部定义的变量都是自动变量，它有如下特点：
>
> - 在一个函数内定义，只在函数范围内有效
> - 在复合语句中定义，只在复合语句中有效
> - 随着函数调用的结束或复合语句的结束局部变量的声明生命周期也结束
> - 如果没有赋初值，内容为默认初始值

8、2 全局变量

> 在函数外定义，可被本文件及其它文件中的函数所共用，若其它文件中的函数调用此变量，须用包名和全局变量名
>
> - 全局变量的生命周期和程序运行周期一样
> - 同包文件的全局变量不可重名

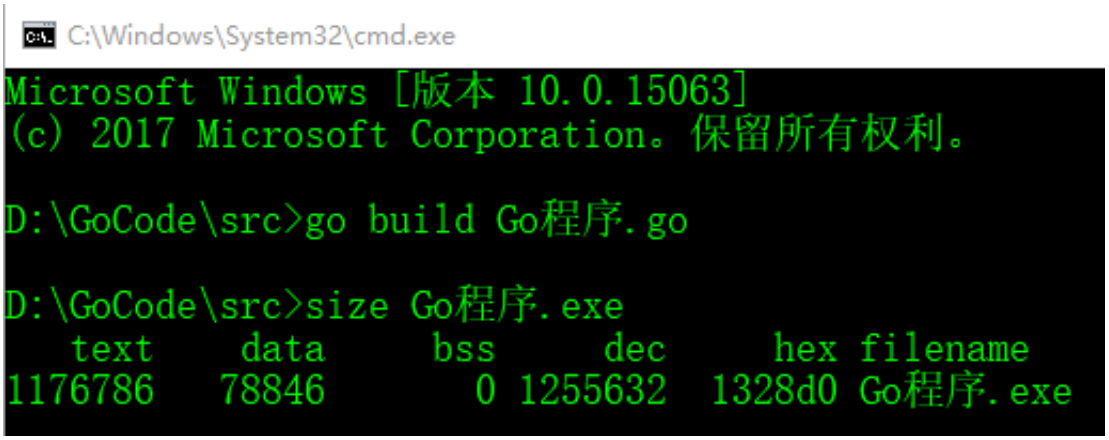
8、3 函数作用域

> 根据函数名首字母是否为大写，函数的作用域也会不同
>
> - 首字母小写，可以在包内多个文件中使用
> - 首字母大写，可以在保外使用，格式为：包名.函数名

9 内存管理

9、1 内存分区

> 代码经过预处理、编译、汇编、链接4步后生成一个可执行程序。
>
> 在 Windows 下，程序是一个普通的可执行文件，以下列出一个二进制可执行文件的基本情况：



> 通过上图可以得知，在没有运行程序前，也就是说程序没有加载到内存前，可执行程序内部已经分好三段信息，分别为**代码区（text）**、**数据区（data）**和**未初始化数据区（bss）**3 个部分。
>
> 有些人直接把data和bss合起来叫做**静态区**或**全局区**。

9、2 代码区（text）

> 存放 CPU 执行的机器指令。通常代码区是可共享的（即另外的执行程序可以调用它），使其可**共享**的目的是对于频繁被执行的程序，只需要在内存中有一份代码即可。代码区通常是**只读**的，使其只读的原因是防止程序意外地修改了它的指令。另外，代码区还规划了局部变量的相关信息。

9、3 全局初始化数据区/静态数据区（data）

> 该区包含了在程序中明确被初始化的全局变量、已经初始化的静态变量（包括全局静态变量和局部静态变量）和常量数据（如字符串常量）。

9、4 未初始化数据区（bss）

> 存入的是全局未初始化变量和未初始化静态变量。未初始化数据区的数据在程序开始执行之前被内核初始化为 0 或者空（nil）。

> 程序在加载到内存前，代码区和全局区(data和bss)的大小就是固定的，程序运行期间不能改变。
>
> 然后，运行可执行程序，系统把程序加载到内存，除了根据可执行程序的信息分出代码区（text）、数据区（data）和未初始化数据区（bss）之外，还额外增加了**栈区**、**堆区**。

9、5 栈区（stack）

> 栈是一种先进后出的内存结构，由编译器自动分配释放，存放函数的参数值、返回值、局部变量等。
>
> 在程序运行过程中实时加载和释放，因此，局部变量的生存周期为申请到释放该段栈空间。

9、6 堆区（heap）

> 堆是一个大容器，它的容量要远远大于栈，但没有栈那样先进后出的顺序。用于动态内存分配。堆在内存中位于BSS区和栈区之间。
>
> 根据语言的不同，如C语言、C++语言，一般由程序员分配和释放，若程序员不释放，程序结束时由操作系统回收。
>
> Go语言、Java、python等都有垃圾回收机制（GC），用来自动释放内存。