第二章　基础知识

知识点一：内存区域
知识点二：函数调用
知识点三：常见寄存器和栈帧
知识点四：主要寄存器

学习之前，回顾两个问题

编制程序的四个步骤

编辑、编译、链接和运行

编译—检查语法并对单个文件产生可执行程序

链接—多个可执行文件进行关联（函数调用信息），增加启动程序等

如何认识BUG和漏洞

BUG就是缺陷，一种表现就是用户部分操作导致程序崩溃，比如输入超长、除以0等
程序崩溃就意味着用户的输入会影响程序正常执行，意味着可能通过BUG提供的入口输入构造的恶意程序让程序做非法的事情，因此在安全人眼里，BUG是一种软件漏洞

漏洞是网络安全的源起之地，在软件里为什么BUG会造成如此恶劣影响？

知识点一：内存区域

1. 堆栈基础——内存区域

内存区域：一个进程可能被分配到不同的内存区域去执行：

代码区

通常是指用来存放程序执行代码的一块内存区域。这个区域存储着被装入执行的二进制机器代码，处理器会到这个区域取指并执行。

静态数据区

通常是指用来存放程序运行时的全局变量、静态变量等的内存区域。通常，静态数据区包括初始化数据区（Data Segment）和未初始化数据区（BSS Segment）两部分。未初始化数据区BSS区存放的是未初始化的全局变量和静态变量，特点是可读写，在程序执行之前BSS段会自动清0。

堆区

用于动态地分配进程内存。进程可以在堆区动态地请求一定大小的内存，并在用完之后归还给堆区。动态分配和回收是堆区的特点。

栈区

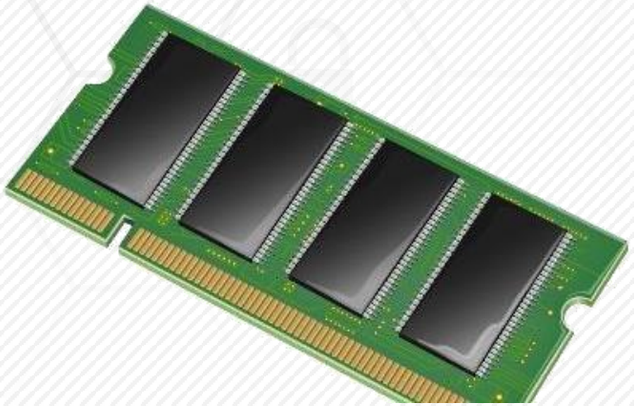
用于支持进程的执行，动态地存储函数之间的调用关系、局部变量等，以保证被调用函数在返回时恢复到母函数中继续执行。

在任何操作系统中，高级语言写出的程序经过编译链接，都会形成一个可执行文件。每个可执行文件包含了二进制级别的机器代码，将被装载到内存的**代码区**；

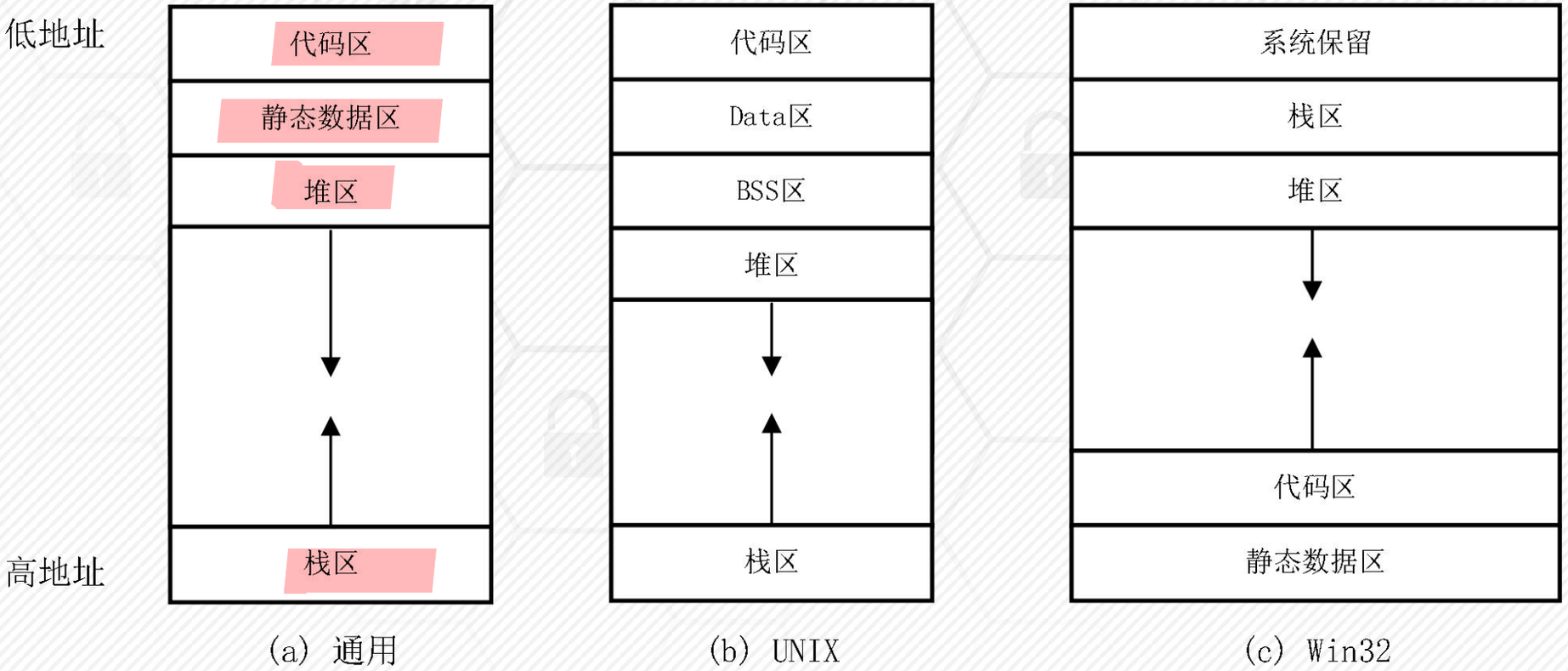
处理器将到内存的**代码区**一条一条地取出指令和操作数，并送入算术逻辑单元进行运算；

如果代码中请求开辟动态内存，则会在内存的**堆区**分配一块大小合适的区域返回给代码区的代码使用；

当函数调用发生时，函数的调用关系等信息会动态地保存在内存的**栈区**，以供处理器在执行完备调用函数的代码时，返回母函数。



进程内存的精确组织形式依赖于操作系统、编译器、链接器以及载入器，不同操作系统有不同的内存组织形式。下图展示了UNIX和Win32可能的进程内存组织形式，虽然次序有差异，但整体上还是按上述四类内存进行组织。



(a) 通用　(b) UNIX　(c) Win32

堆区和栈区

程序在执行的过程需要两种不同类型的内存来协同配合，即栈区和堆区。

栈（stack）区主要存储函数运行时的局部变量、数组等。栈变量在使用时不需要额外的申请操作，系统栈会根据函数中的变量声明自动为其预留内存空间；同样，栈变量的释放也无需程序员参与，由系统栈跟随函数调用的结束自动回收。

栈区是向**低地址扩展**的数据结构，是一种**先入后出**的特殊结构。栈顶的地址和栈的最大容量是系统预先规定好的，在WINDOWS下，栈的默认大小是2M，如果申请的空间超过栈的剩余空间时，将提示溢出。

堆区和栈区

程序在执行的过程需要两种不同类型的内存来协同配合，即栈区和堆区。

堆（heap）区是一种程序运行时动态分配的内存。所谓动态，就是说所需内存的大小在程序设计时不能预先确定或者内存过大无法在栈区分配，需要在程序运行的时候参考用户的反馈。

堆区在使用的时候需要程序员**使用专有的函数进行申请**，如C语言的malloc函数、C++语言的new函数等。它是**向高地址扩展的数据结构，堆的大小受限于计算机的虚拟内存。**

堆区和栈区的区别

申请方式

栈：由系统自动分配。例如，声明一个局部变量int b，系统自动在栈中为b开辟空间。
堆：需要程序员自己申请，并指明大小，在c中malloc函数，如
p1 = (char *)malloc(10)。

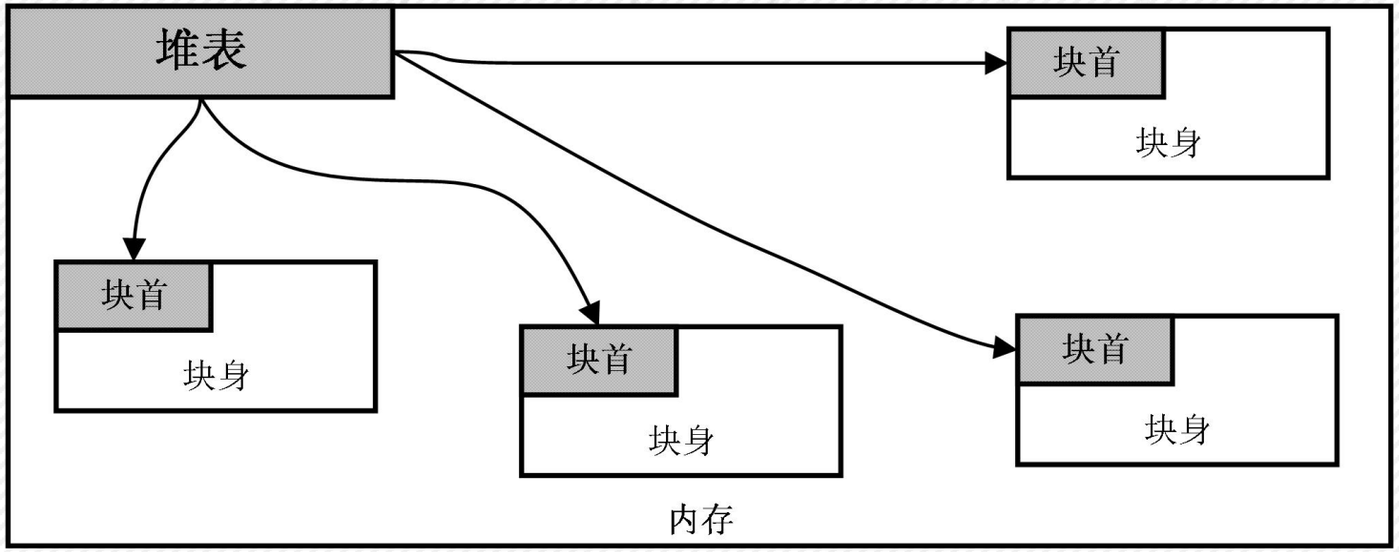
申请效率

栈由系统自动分配，速度较快，但程序员是无法控制的。
堆是由程序员分配的内存，一般速度比较慢，而且容易产生内存碎片，不过用起来方便。

2. 堆结构

堆的内存组织如下图所示，包括堆块和堆表两部分。

- ✓ **堆块是堆的基本组织单位，包括两个部分，即块首和块身。**块首是用来标识这个堆块自身的信息，例如块大小、空闲还是占用等；块身紧随其后，是最终分配给用户使用的数据区。
- ✓ **堆表一般位于整个堆区的开始位置，用于索引堆区中所有堆块的重要信息**，包括堆块的位置、堆块的大小、空闲还是占用等。堆表的数据结构决定了整个堆区的组织方式，是快速检索空闲块、保证块分配效率的关键。堆表在设计的时候，可能会采用平衡二叉树等高效数据结构用于优化查找效率。现代操作系统的堆表往往不止一种数据结构。

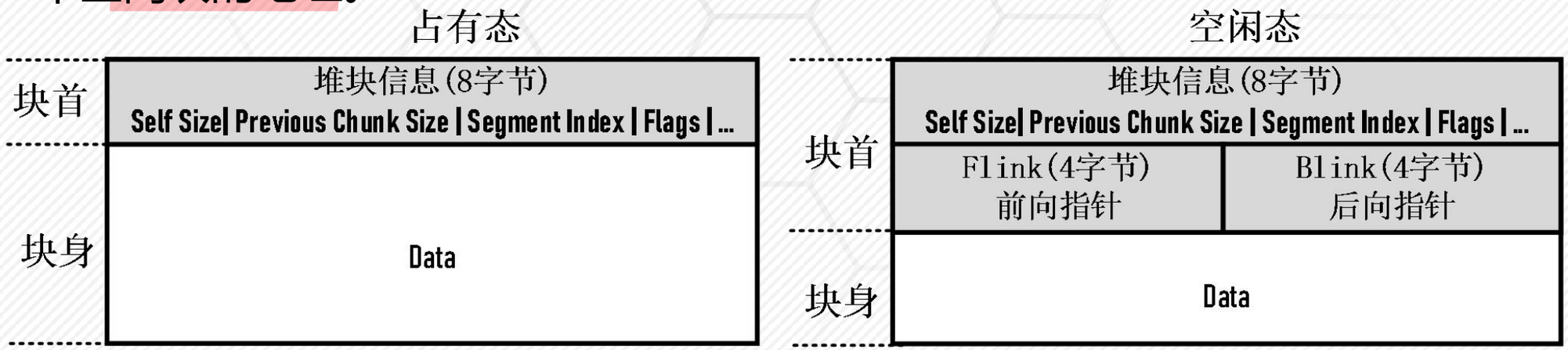


（1）堆块

堆块会有两种状态：**占有态和空闲态**。其中，空闲态的堆块会被链入空链表中，由系统管理。而占有态的堆块会返回一个由程序员定义的句柄，通常是一个堆块指针，来完成对堆块内存的读、写和释放操作，由程序员管理。

占有态堆块和空闲态堆块的示意图如下图所示。块首存放着堆块的信息。

对于空闲态堆块而言，块首额外存储了两个4字节的指针：Flink指针和Blink指针，用于链接系统中的其他空闲堆块。其中，Flink前向指针存储了前一个空闲块的地址，Blink后向指针存储了后一个空闲块的地址。



（1）堆块

注意：指向堆块的指针或者句柄，指向的是块身的首地址。也就是，我们使用函数申请得到的地址指针都会越过8字节(32位系统)的块首，直接指向数据区(块身)。

堆块的大小包括块首在内，如果申请32字节，实际会分配40字节，8字节的块首+32字节的块身。

堆块的单位是8字节，不足8字节按8字节分配。

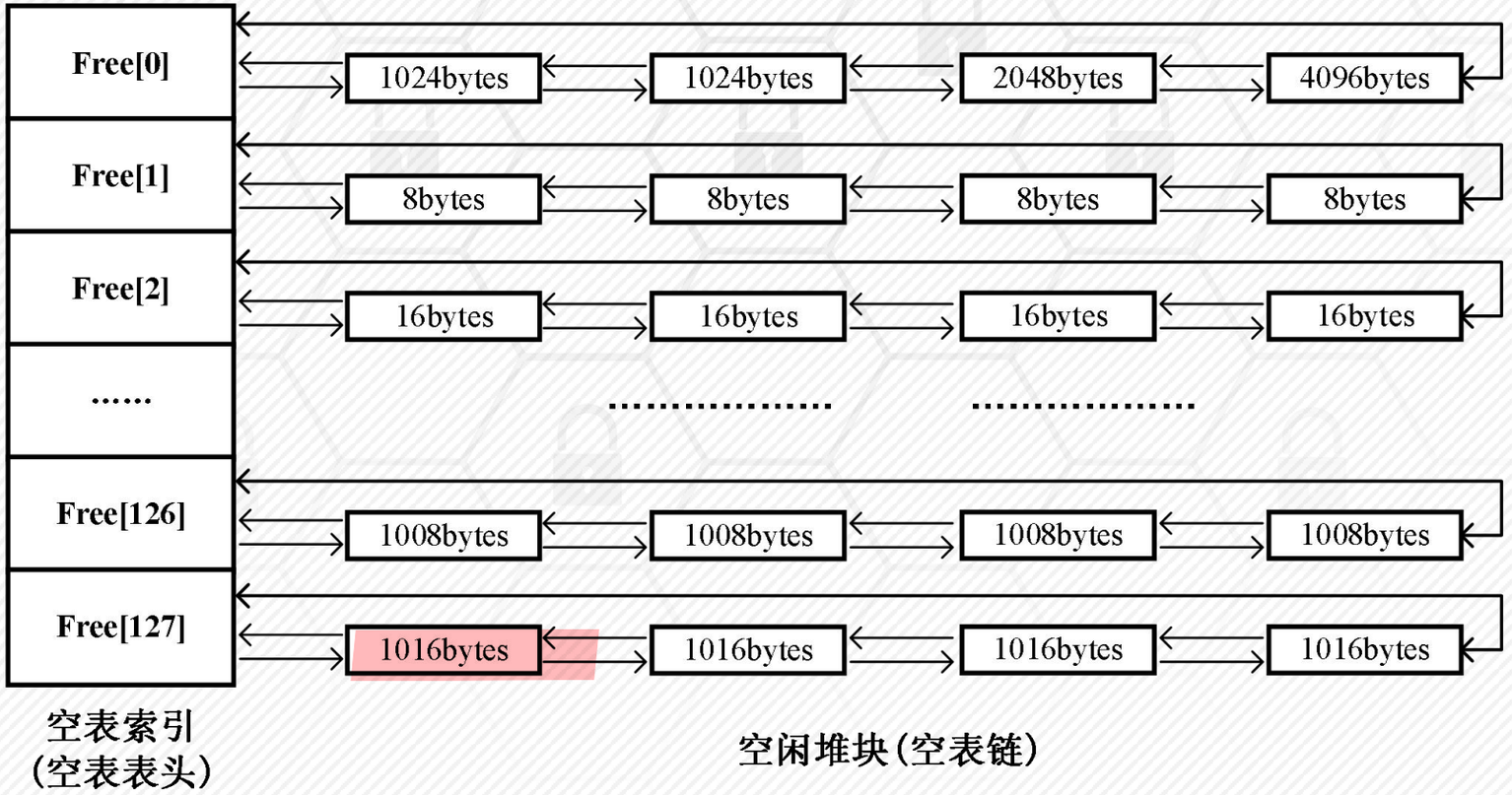
（2）堆表

在Windows系统中，**占有态的堆块被使用它的程序索引，而堆表只索引所有空闲态的堆块**。其中，最重要的堆表有两种：**空闲双向链表freelist（简称空表）和快速单向链表lookaside（简称快表）**。快表是为了加速堆块分配而采用的堆表，从来不发生堆块合并。由于堆溢出一般不利用快表，故不作详述。

空表包含**空表索引**(Freelist array)和**空闲链块**两个部分。空表索引也叫空表表头，是一个大小为128的指针数组，该数组的每一项包括两个指针，用于标识一条空表。

(2) 堆表

空表索引的第二项(free[1])标识了堆中所有大小为8字节的空闲堆块。之后每个索引项指示的空闲堆块递增8字节。把空闲堆块按照大小的不同链入不同的空表，可以方便堆管理系统高效检索指定大小的空闲堆块。**空表索引的第一项free[0]所标识的空表相对比较特殊，这条双向链表链入了所有大于等于1024字节小于512KB的堆块，升序排列。这个空表通常又称为零号空表。**



（3）堆块的分配和释放

以空表为例，来讲解堆块的分配、释放和合并。

堆块分配。依据既定的查找空闲堆块的策略，找到合适的空闲堆块之后，将其状态修改为占用态、把它从堆表中“卸下”、返回一个指向堆块块身的指针给程序使用。

- ✓ **普通空表分配时首先寻找最优的空闲块分配，**若失败，一个稍大些的块会被用于分配。这种**次优分配**发生时，**会先从大块中按请求的大小精确地“割”出一块进行分配，然后给剩下的部分重新标注块首，链入空表。**也就是说，**空表分配存在找零钱的情况。**
- ✓ 零号空表中按照大小升序链着大小不同的空闲块，故在分配时先从free[0]反向查找最后一个块（即最大块），看能否满足要求，如果满足要求，再正向搜索最小能满足要求的空闲堆块进行分配。

（3）堆块的分配和释放

以空表为例，来讲解堆块的分配、释放和合并。

堆块释放。堆块的释放操作包括将堆块状态由占用态改为空闲态、链入相应的堆表。所有释放的堆块都链入相应的表尾。

堆块合并。堆块的分配和释放操作可能引发堆块合并，即当堆管理系统发现两个空闲堆块相邻时，就会进行堆块合并操作。

堆块的合并包括几个动作：将堆块从空表中卸下、合并堆块、修改合并后的块首、链接入新的链表（合并的时候还有一种操作叫内存紧缩）。

知识点二：函数调用

堆栈基础——函数调用

函数调用时候将借助系统栈来完成函数状态的保存和恢复。

```
int func_B(int arg_B1, int arg_B2)
{
    int var_B1, var_B2;
    var_B1 = arg_B1 + arg_B2;
    var_B2 = arg_B1 – arg_B2;
    return var_B1 * var_B2;
}
int func_A(int arg_A1, int arg_A2)
{
    int var_A;
    var_A = func_B(arg_A1, arg_A2) + arg_A1;
    return var_A;
}
```

```
int main(int argc, char **argv, char **envp)
{
int var_main;
var_main = func_A(4, 3);
return var_main;
}
```

在所生成的可执行文件中，代码是以函数为单元进行存储

当CPU在执行调用func_A函数的时候，会从代码区中main函数对应的机器指令的区域跳转到func_A函数对应的机器指令区域，在那里取指并执行.....

那么CPU是怎么知道要去func_A的代码区取指，在执行完func_A后又是怎么知道跳回到main函数（而不是func_B的代码区）的呢？这些跳转地址我们在代码中并没有直接说明，CPU是从哪里获得这些函数的调用及返回的信息的呢？

这些代码区中精确的跳转都是在与系统栈巧妙地配合过程中完成的。

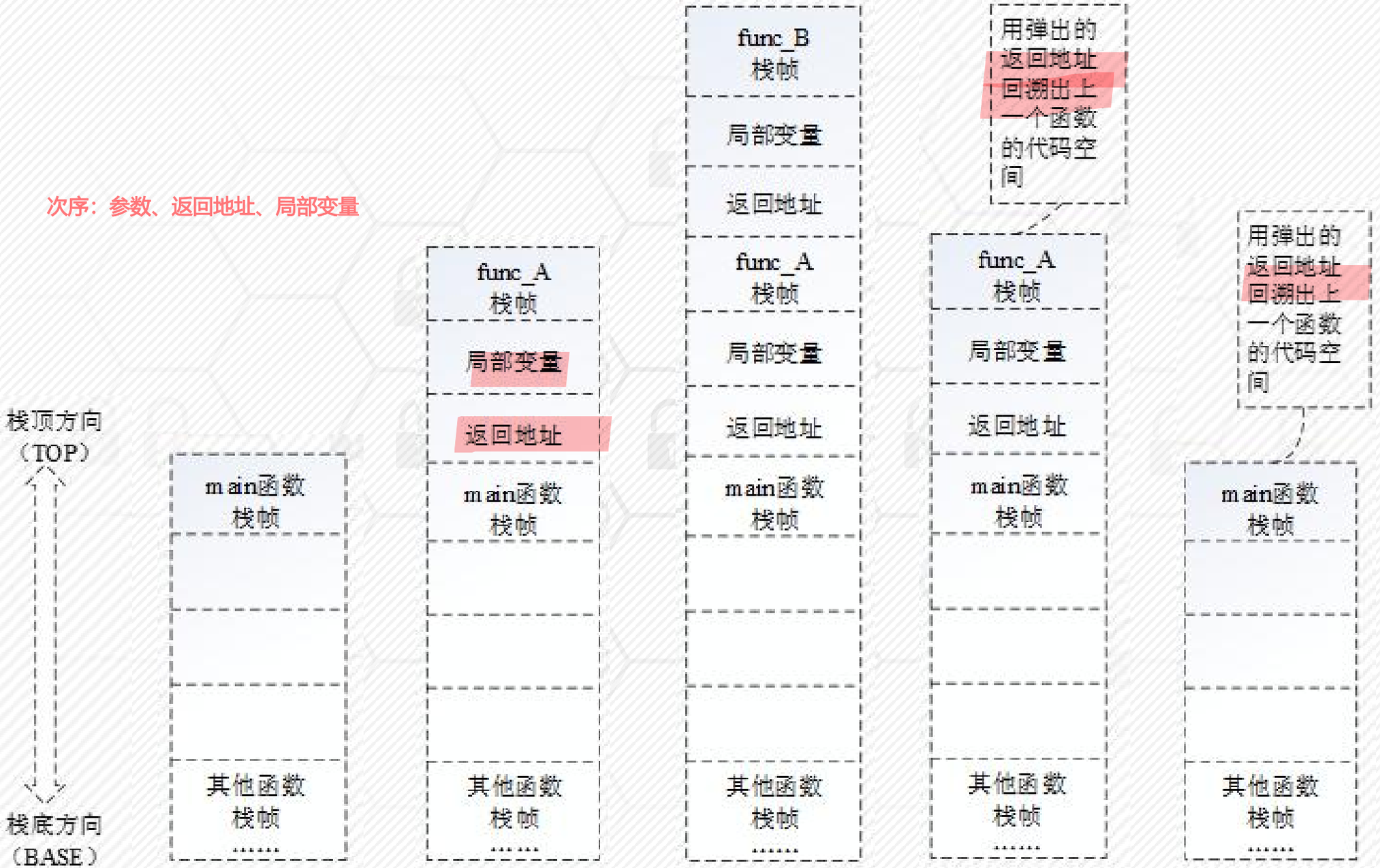
当函数被调用时，系统栈会为这个函数开辟一个新的**栈帧**，并把它压入栈中。

每个栈帧对应着一个未运行完的函数。栈帧中保存了该函数的返回地址和局部变量。从逻辑上讲，栈帧就是一个函数执行的环境：函数参数、函数的局部变量、函数执行完后返回到哪里等等。

栈帧中保存该函数的返回地址、函数参数、局部变量

当函数返回时，系统栈会弹出该函数所对应的栈帧。

次序：参数、返回地址、局部变量



函数调用的步骤：

(1) 参数入栈 ⟶ 将参数从右向左依次压入系统栈中。

(2) 返回地址入栈 ⟶ 将当前代码区调用指令的下一条指令地址压入栈中，供函数返回时继续执行。

返回地址就是：当前代码区调用指令的下一条指令地址

(3) 代码区跳转 ⟶ 处理器从当前代码区跳转到被调用函数的入口处。

(4) 栈帧调整 ⟶ 具体包括：

- 保存当前栈帧状态值，已备后面恢复本栈帧时使用。
- 将当前栈帧切换到新栈帧。

知识点三：常见寄存器

寄存器

寄存器的功能：暂存指令、数据和地址

寄存器（register）是中央处理器CPU的组成部分。寄存器是有限存贮容量的高速存贮部件，它们可用来暂存指令、数据和地址。**我们常常看到32位CPU、64位CPU这样的名称，其实指的就是寄存器的大小。**32位CPU的寄存器大小就是4个字节。

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使用寄存器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。

- 每一个函数独占自己的栈帧空间。当前正在运行的函数的栈帧总是在栈顶。Win32系统提供两个特殊的寄存器用于标识位于**系统栈顶端的栈帧**：

ESP

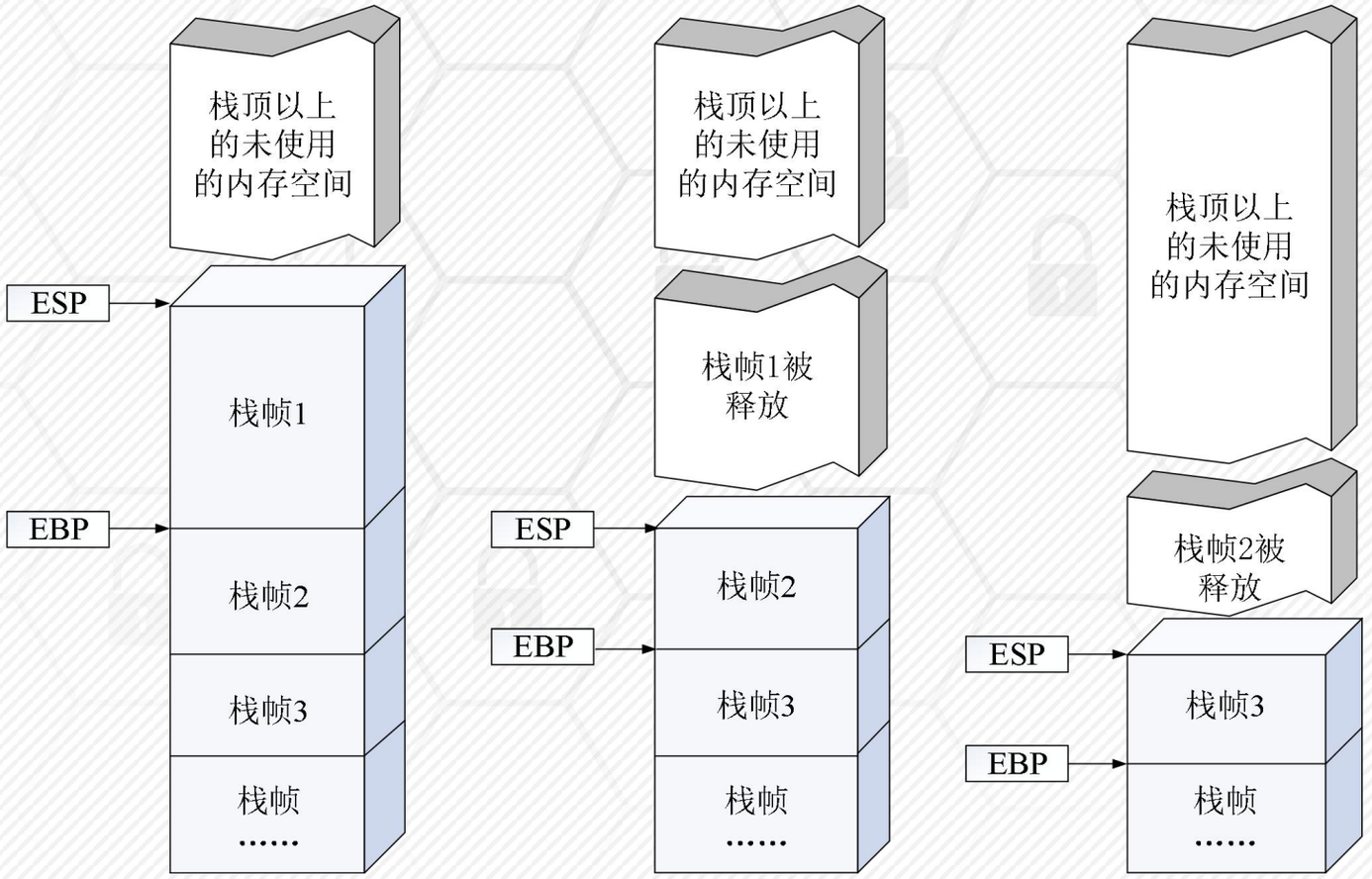
栈指针寄存器（extended stack pointer），其内存放着一个指针，该指针永远指向系统栈最上面一个栈帧的栈顶。

EBP

基址指针寄存器（extended base pointer），其内存放着一个指针，该指针永远指向系统栈最上面一个栈帧的底部。

函数栈帧

ESP和EBP之间的内存空间为**当前栈帧**，EBP标识了**当前栈帧的底部**，ESP标识了**当前栈帧的顶部**。



在函数栈帧中，一般包含以下几类重要信息：

局部变量

为函数局部变量开辟的内存空间。

栈帧状态值

保存前栈帧的顶部和底部（实际上只保存前栈帧的底部，前栈帧的顶部可以通过堆栈平衡计算得到），用于在本帧被弹出后恢复出上一个栈帧。

函数返回地址

保存当前函数调用前的“断点”信息，也就是函数调用前的指令位置，以便在函数返回时能够恢复到函数被调用前的代码区中继续执行指令。

除了与栈相关的寄存器之外，还需要记住另一个至关重要的寄存器。

EIP

指令寄存器（extended instruction pointer），其内存放着一个指针，该指针永远指向下一条等待执行的指令地址。可以说**如果控制了EIP寄存器的内容，就控制了进程**——我们让EIP指向哪里，CPU就会去执行哪里的指令。

在函数调用过程中，结合寄存器看一下如何实现栈帧调整：

- 保存当前栈帧状态值，已备后面恢复本栈帧时使用（EBP入栈）。
- 将当前栈帧切换到新栈帧（将ESP值赋值EBP，更新栈帧底部）

知识点四：主要寄存器

对于寄存器的学习一般
不是很了解作用
2.22

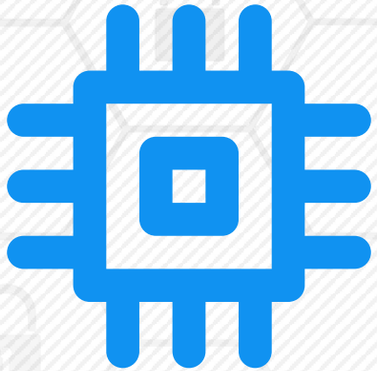
偏移量是什么
有什么用

对汇编语言进行一下回顾，了解重要的寄存器和汇编指令。

2个变址寄存器(ESI和EDI)
2个指针寄存器(ESP和EBP)

6个段寄存器
(ES、CS、SS、DS、FS和GS)

4个数据寄存器
(EAX、EBX、ECX和EDX)



1个指令指针寄存器(EIP)
1个标志寄存器(EFlags)

在汇编语言中，主要有这些寄存器

数据寄存器

数据寄存器主要用来保存操作数和运算结果等信息，从而节省读取操作数所需占用总线和访问存储器的时间。

32位CPU有4个32位的通用寄存器EAX、EBX、ECX和EDX。对低16位数据的存取，不会影响高16位的数据。这些低16位寄存器分别命名为：AX、BX、CX和DX，它和先前的CPU中的寄存器相一致。

4个16位寄存器又可分割成8个独立的8位寄存器(AX：AH-AL、BX：BH-BL、CX：CH-CL、DX：DH-DL)，每个寄存器都有自己的名称，可独立存取。

程序员可利用数据寄存器的这种“可分可合”的特性，灵活地处理字/字节的信息。

EAX通常称为累加器(Accumulator)

可用于乘、 除、输入/输出等操作，它们的使用频率很高。**EAX还通常用于存储函数的返回值。**

EBX称为基地址寄存器(Base Register)

它可作为存储器指针来使用，用来访问存储器。

ECX称为计数寄存器(Count Register)

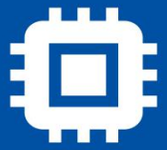
在循环和字符串操作时，要用它来控制循环次数；在位操作中，当移多位时，要用CL来指明移位的位数。

EDX称为数据寄存器(Data Register)

在进行乘、除运算时，可作为默认操作数参与运算，也可用于存放I/O的端口地址。

变址寄存器

变址寄存器主要用来存放操作数的地址，用于堆栈操作和变址运算中计算操作数的有效地址。



32位CPU有2个32位通用寄存器ESI和EDI。其低16位对应先前CPU中的SI和DI，对低16位数据的存取，不影响高16位的数据。

ESI通常在内存操作指令中作为“源地址指针”使用，而EDI通常在内存操作指令中作为“目的地址指针”使用。

指针寄存器

寄存器EBP、ESP称为指针寄存器(Pointer Register)，**主要用于存放堆栈内存储单元的偏移量，用它们可实现多种存储器操作数的寻址方式，**为以不同的地址形式访问存储单元提供方便。

指针寄存器不可分割成8位寄存器。作为通用寄存器，也可存储算术逻辑运算的操作数和运算结果。

EBP为基指针(Base Pointer)寄存器，通过它减去一定的偏移值，来访问栈中的元素；

它们主要用于访问堆栈内的存储单元，并且规定：

ESP为堆栈指针(Stack Pointer)寄存器，它始终指向栈顶。

段寄存器

段寄存器是根据内存分段的管理模式而设置的。内存单元的物理地址由段寄存器的值和一个偏移量组合而成的，标准形式为“段：偏移量”，这样可用两个较少位数的值组合成一个可访问较大物理空间的内存地址。

可以认为，一个段是一本书的某一页，偏移量是一页的某一行

- **CS** 代码段寄存器，其值为代码段的段值
- **SS** 堆栈段寄存器，其值为堆栈段的段值
- **DS** 数据段寄存器，其值为数据段的段值
- **FS** 附加段寄存器，其值为附加数据段的段值
- **ES** 附加段寄存器，其值为附加数据段的段值
- **GS** 附加段寄存器，其值为附加数据段的段值

融合变址寄存器，在很多字符串操作指令中，DS:ESI指向源串，而ES:EDI指向目标串。

指令指针寄存器

指令寄存器（IR，Instruction Register），是临时放置从内存里面取得的程序指令的寄存器，用于存放当前从主存储器读出的正在执行的一条指令。当执行一条指令时，先把它从内存取到数据寄存器（DR，Data Register）中，然后再传送至IR。指令划分为操作码和地址码字段，由二进制数字组成。

指令指针寄存器用英文简称为IP（Instruction Pointer），它虽然也是一种指令寄存器，但是严格意义上和传统的指令寄存器有很大的区别。指令指针寄存器存放下次将要执行的指令在代码段的偏移量。在计算机工作的时候，CPU会从IP中获得关于指令的相关内存地址，然后按照正确的方式取出指令，并将指令放置到原来的指令寄存器中。

32位CPU把指令指针扩展到32位，并记作EIP。

标志寄存器

标志寄存器在32位操作系统中大小是32位的，也就是说，它可以存32个标志。实际上标志寄存器并没有完全被使用，重点认识三个标志寄存器：ZF (零标志)、OF(溢出标志)、CF(进位标志)。

Z-Flag(零标志)：它可以设成0或者1。

O-Flag(溢出标志)：反映有符号数加减运算是否溢出。如果运算结果超过了有符号数的表示范围，则OF置1，否则置0。例如：EAX的值为7FFFFFFF，如果你此时再给EAX加1，OF寄存器就会被设置成1，因为此时EAX寄存器的最高有效位改变了。

C-Flag(进位标志)：用于反映运算是否产生进位或借位。如果运算结果的最高位产生一个进位或借位，则CF置1，否则置0。例，假如某寄存器值为FFFFFFFF，再加上1就会产生进位。