第二章 堆栈基础

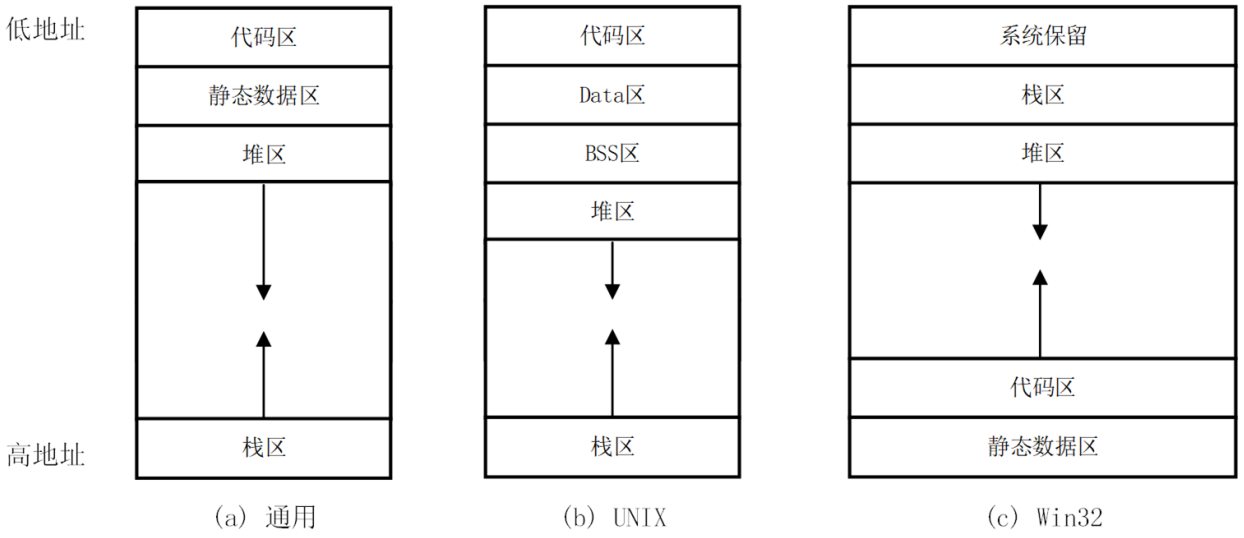
1 内存区域

一个进程可能被分配到不同的内存区域去执行

- 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。

- 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。

不同的操作系统有不同的内存组织形式：



(a) 通用　(b) UNIX　(c) Win32

2 堆区和栈区

2.1 堆区

- 是一种程序运行时动态分配的内存，不能预先确定，需要在使用的时候用专有的函数进行申请，比如说$malloc$函数，new函数等。
  - `p1 = (char *)malloc(10)`
- 堆是一种**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**
- 堆一般由程序员来分配，速度较慢，容易产生内存碎片，使用比较方便

2.2 栈区

- 主要存储函数运行时的局部变量、数组等，不需要额外申请，系统会自动为变量预留内存空间，栈的释放也是函数调用结束后回收
- 栈是一种**向低地址扩展**的数据结构，**先入后出**，默认大小是$2M$，如果申请空间超过栈的剩余空间，就会发生栈溢出
- 栈一般分配的速度较快，程序员无法控制

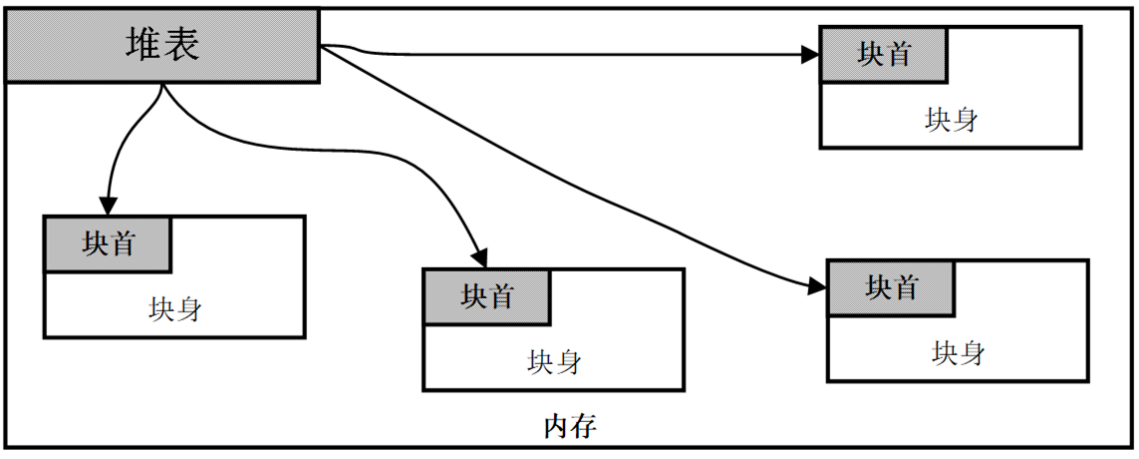
2.3 堆的结构

堆的内存主要分为：**堆块和堆表**

堆块是堆的基本组织单位，包括**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

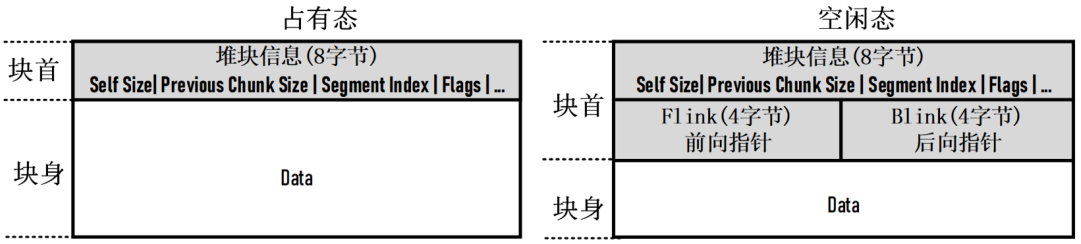
堆表一般位于整个堆区的开始位置，用于索引堆区中所有堆块的重要信息，包括堆块的位置、堆块的大小、空闲还是占用等



2.3.1 堆块

堆块有两种状态：**占有态和空闲态**

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



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**

指向堆块的指针，指向的是块身的首地址，就是说，我们的地址指针不会指向块首的堆块信息，而是直接指向块身的数据区

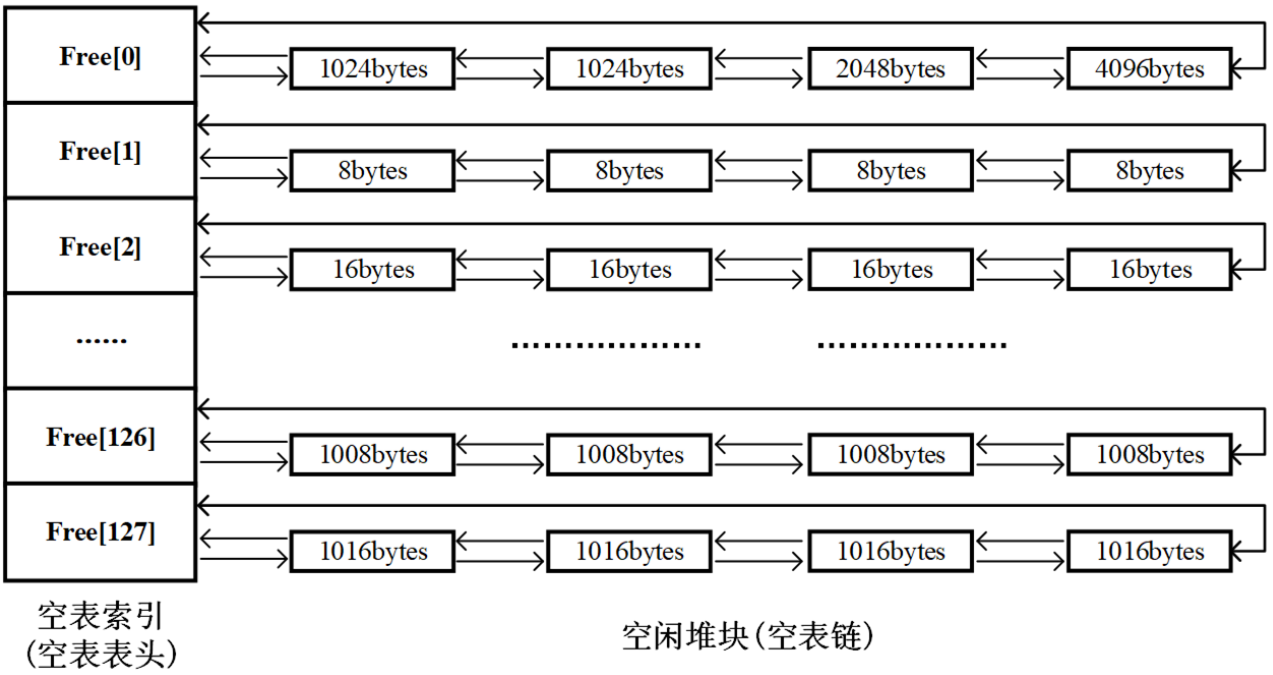
堆块的大小包括块首在内，如果申请32字节，实际会分配**40字节，8字节的块首+32字节的块身**。堆块的**单位是8字节**，不足8字节按8字节分配。

2.3.2 堆表

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）

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

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**。



2.4 堆块的分配与释放

2.4.1 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。

2.4.2 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**

2.4.3 堆块合并

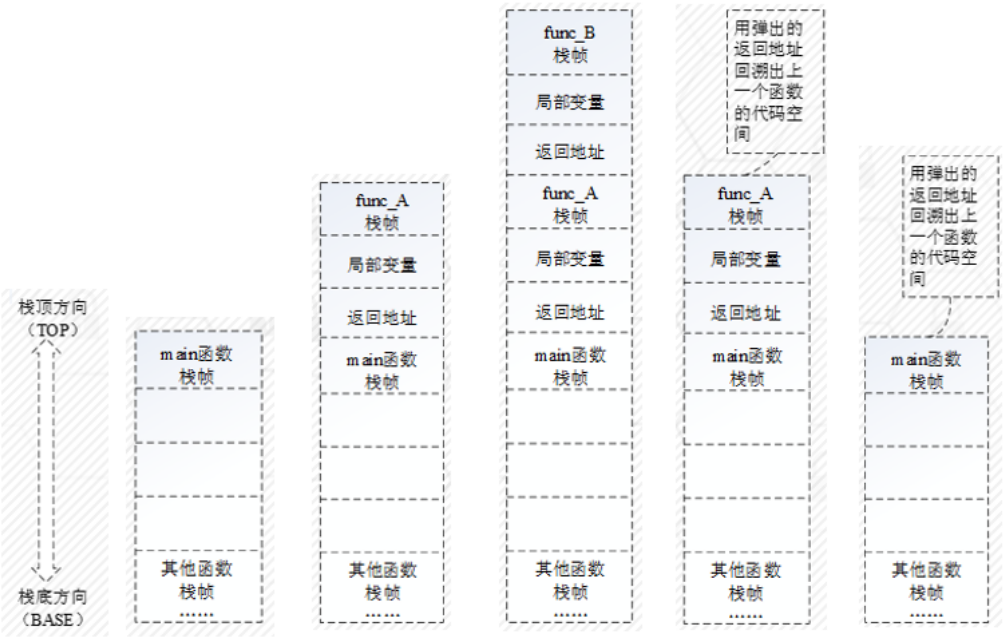
堆块的分配和释放操作可能引发堆块合并，即当堆管理系统发现**两个空闲堆块相邻**时，就会进行**堆块合并**操作。

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）

3 函数调用

借助系统栈来完成函数状态的保存和恢复

调用函数，如何跳转到main函数的位置呢？我们利用系统栈来完成这个调用，当函数被调用时，系统就会给这个函数开辟一个新的栈帧，并将其压入栈中，每一个栈帧都对应了一个没有运行完的函数，在栈中保存了该函数的**返回地址和局部变量**，其实栈帧就是一个函数执行的环境，包括了函数的参数、函数的局部变量、函数执行完之后的返回地址等，当函数返回时，系统栈会弹出该函数所对应的栈帧



函数调用步骤：

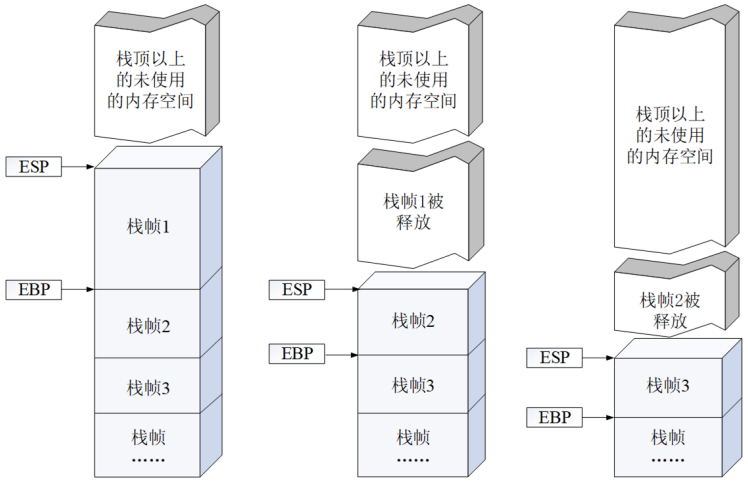
- 参数入栈：将参数从右向左依次压入系统栈中
- 返回地址入栈：将当前代码区调用指令的下一条指令地址压入栈中，供函数返回时继续执行
- 代码区跳转：处理器从当前代码区跳转到被调用函数的入口处
- 栈帧调整：保存当前栈帧状态值，已备后面恢复本栈帧时使用；将当前栈帧切换到新栈帧

4 常见寄存器

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

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

- 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



栈帧中的重要信息：

- 局部变量：为函数局部变量开辟内存空间
- 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

另外一个寄存器：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

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）

5 主要寄存器

- 数据寄存器：**EAX,EBX,ECX,EDX**
  - 上面四个都是32位的，然后还有四个16位的寄存器，AX,BX,CX,DX，都是存储了低16位的数据
  - 这四个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
  - EAX：累加器，用于乘除、输入输出等操作，还可以存储函数的返回值
  - EBX：基地址寄存器，用于访问存储器
  - ECX：计数寄存器，一般在循环中控制循环次数
  - EDX：数据寄存器
- 两个变址寄存器ESI和EDI，两个指针寄存器ESP和EBP
  - 变址寄存器
    - 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
    - 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，不可以分割为8位寄存器

    - 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
    - ESP为堆栈指针(Stack Pointer)寄存器，它始终指向栈顶

- 6个段寄存器：ES、CS、SS、DS、FS和GS
  - CS:代码段寄存器，其值为代码段的段值
  - DS:数据段寄存器，其值为数据段的段值
  - ES:附加段寄存器，其值为附加数据段的段值
  - SS:堆栈段寄存器，其值为堆栈段的段值
  - FS: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
- 指令指针寄存器EIP，标志寄存器EFlags
  - 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
  - 标志寄存器
    - **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