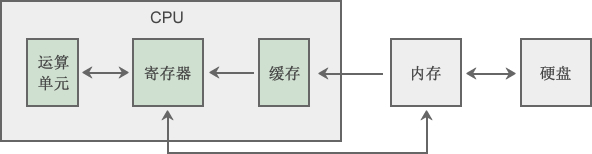
C语言和内存（非常重要）

# 一个程序在计算机中是如何运行的？

程序是保存在硬盘中的，要载入内存才能运行，CPU也被设计为只能从内存中读取数据和指令。

对于CPU来说，内存就是一个**存放数据和指令**的地方。



三者的读取速度是一次减小的，而且减小的比例很大。

在这里面运算单元相当于cpu的大脑，负责加减乘除，比较等运算工作，每种运算都有对应的电路支持，速度很快。

**寄存器（Register）**是CPU内部非常小、非常快速的**存储部件**，它的容量很有限，对于32位的CPU，每个寄存器一般能存储32位（4个字节）的数据，对于64位的CPU，每个寄存器一般能存储64位（8个字节）的数据。为了完成各种复杂的功能，现代CPU都内置了几十个甚至上百个的寄存器，嵌入式系统功能单一，寄存器数量较少。

我们经常听说多少位的CPU，指的就是寄存器的的位数。现在个人电脑使用的CPU已经进入了64位时代，例如 Intel 的 Core i3、i5、i7 等。

寄存器在程序的执行过程中至关重要，不可或缺，它们可以用来完成数学运算、控制循环次数、控制程序的执行流程、标记CPU运行状态等。例如，EIP（Extern Instruction Pointer ）寄存器的值是下一条指令的地址，CPU执行完当前指令后，会根据 EIP 的值找到下一条指令，改变 EIP 的值，就会改变程序的执行流程；CR3 寄存器保存着当前进程页目录的物理地址，切换进程就会改变 CR3 的值，这将在《MMU部件以及对内存权限的控制》中讲解；EBP、ESP 寄存器用来指向栈的底部和顶部，函数调用会改变 EBP 和 ESP 的值，这将在《栈的概念以及栈溢出》中讲解。

## CPU内部为什么要设置缓存

虽然内存的读取速度已经很快了，但是和CPU比起来，还是有很大差距的，不是一个数量级的，如果每次都从内存中读取数据，会严重拖慢CPU的运行速度，CPU经常处于等待状态，无事可做。在CPU内部设置一个缓存，可以将使用频繁的数据暂时读取到缓存，需要同一地址上的数据时，就不用大老远地再去访问内存，直接从缓存中读取即可。

大家在购买CPU时，也会经常关心缓存容量，例如 Intel Core i7 3770K 的三级缓存为 8MB，二级缓存为 256KB，一级缓存为 32KB。容量越大，CPU越强悍。

缓存的容量是有限的，CPU只能从缓存中读取到部分数据，对于使用不是很频繁的数据，会绕过缓存，直接到内存中读取。所以不是每次都能从缓存中得到数据，这就是缓存的命中率，能够从缓存中读取就命中，否则就没命中。关于缓存的命中率又是一门学问，哪些数据保留在缓存，哪些数据不保留，都有复杂的算法。

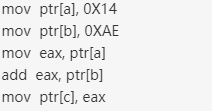
## CPU指令

要想让CPU工作，必须借助特定的指令，例如 add 用于加法运算，sub 用于除法运算，cmp 用于比较两个数的大小，这称为CPU的指令集（Instruction Set）。我们的C语言代码最终也会编译成一条一条的CPU指令。不同型号的CPU支持的指令集会有所差异，但绝大部分是相同的。

C语言示例：



生成的CPU指令（其实是汇编指令，还需要进行简单的转换才会变成CPU指令）



mov 和 add 都是CPU指令：

1) mov 用来将一个数值移动到一个存储位置。这个数值可以是一个常数，也可以在内存或者寄存器上；这个存储位置可以是寄存器或者内存。

第一条指令中，ptr[a]表示变量 a 的地址，0X14是一个数值，mov ptr[a], 0X14表示**把数值 0X14 移动到 ptr[a] 指向的内存**，也就是给变量 a 赋值。第二条指令与此类似。

第三条指令中，eax是寄存器的名字，该寄存器常用在加法运算中，用来保存某个加数或运算结果，mov eax, ptr[a]表示**把变量 a 的值移动到寄存器 eax 中。**

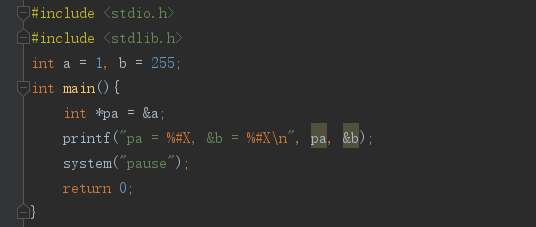
第五条指令表示把**寄存器 eax 的值移动到变量 c 中**，此时 exa 中的值为 a、b 相加的和。

2) add 用来将两个数值相加，这两个数值可以在寄存器或者内存中，add 会将相加的结果放在第一个数所在的位置。第四条指令add eax, ptr[b]表示**把 eax 和 ptr[b] 中的数值相加，并把结果放在 eax 中。**

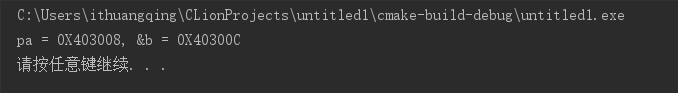
总起来讲：第一二条指令给变量 a、b 赋值，第三四条指令完成加法运算，第五条指令将运算结果赋值给变量 c。

# 虚拟内存

有如下的代码：

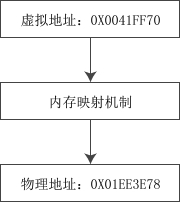


结果为：



这里取到的都是变量的内存地址，也就是存放这些数值的内存空间的地址。

这些地址都是假的，不是真实的物理地址，而是虚拟地址，虚拟地址通过CPU的转换才能对应到物理地址，而且每次程序运行时，操作系统都会重新安排虚拟地址和物理地址的对应关系，哪一段物理内存空闲就使用哪一段。如下图所示：



## 虚拟地址

虚拟地址的整个想法是这样的：把程序给出的地址看做是一种虚拟地址（Virtual Address），然后通过**某些映射的方法**，将这个虚拟地址转换成实际的物理地址。这样，只要我们能够妥善地控制这个虚拟地址到物理地址的映射过程，就可以保证程序每次运行时都可以使用相同的地址。

例如，上面代码中变量 a 的地址是 0X402000，第一次运行时它对应的物理内存地址可能是 0X12ED90AA，第二次运行时可能又对应 0XED90，而我们的程序不需要关心这些，这些繁杂的内存管理工作交给操作系统处理即可。

让我们回到程序的运行本质上来。用户程序在运行时不希望介入到这些复杂的内存管理过程中，作为普通的程序，它需要的是一个简单的执行环境，有自己的内存，有自己的CPU，好像整个程序占有整个计算机而不用关心其他的程序。

除了在编程时可以使用固定的内存地址，给程序员带来方便外，使用虚拟地址还能够使不同程序的地址空间相互隔离，提高内存使用效率。

## 使不同程序的地址空间相互隔离

因为在编程过程中使用到的内存地址都是虚拟地址，需要经过CPU的转换才能连接真实的物理地址，而这些操作都是由操作系统完成，所以，即使不同的程序操作同一个内存地址，但是连接到的确实不同的物理地址，这样不同程序的真实物理地址就相互隔离了，这样就更加安全，互不影响。

## 提高内存使用效率

使用虚拟地址后，操作系统会更多地介入到内存管理工作中，这使得控制内存权限成为可能。例如，我们希望保存数据的内存没有执行权限，保存代码的内存没有修改权限，操作系统占用的内存普通程序没有读取权限等。

另外，当物理内存不足时，操作系统能够更加灵活地控制换入换出的粒度，磁盘 I/O 是非常耗时的工作，这能够从很大程度上提高程序性能。

## 中间层思想

在计算机中，为了让操作更加直观、易于理解、增强用户体验，开发者经常会使用一件法宝——增加中间层，即使用一种间接的方式来屏蔽复杂的底层细节，只给用户提供简单的接口。虚拟地址是使用中间层的一个典型例子。  
  
实际上，计算机的整个发展过程就是不断引入新的中间层：

* 计算机的早期，程序都是直接运行在硬件之上，自己负责硬件的管理工作；程序员也使用二进制进行编程，需要处理各种边界条件和安全问题。
* 后来人们不能忍受了，于是开发出了操作系统，让它来管理各种硬件，同时发明了汇编语言，减轻程序员的负担。
* 随着软件规模的不断增大，使用汇编语言编程开始变得捉襟见肘，不仅学习成本高，开发效率也很低，于是C语言诞生了。C语言编译器先将C代码翻译为汇编代码，再由汇编器将汇编代码翻译成机器指令。
* 随着计算机的发展，硬件越来越强大，软件越来越复杂，人们又不满足于使用C语言了，于是 C++、Java、C#、PHP 等现代化的编程语言诞生了。

# 虚拟地址空间以及编译模式

所谓虚拟地址空间，就是程序可以使用的虚拟地址的有效范围。虚拟地址和物理地址的映射关系由操作系统决定，相应地，虚拟地址空间的大小也由操作系统决定，但还会受到编译模式的影响。

## CPU的数据处理能力

CPU是计算机的核心，决定了计算机的数据处理能力和寻址能力，也即决定了计算机的性能。CPU一次（一个时钟内）能处理的数据的大小由寄存器的位数和数据总线的宽度（也即有多少根数据总线）决定，我们通常所说的多少位的CPU，除了可以理解为寄存器的位数，也可以理解数据总线的宽度，通常情况下它们是相等的。



数据总线和主频都是CPU的重要指标：数据总线决定了CPU单次的数据处理能力，主频决定了CPU单位时间内的数据处理次数，它们的乘积就是CPU单位时间内的数据处理量。

**数据总线宽度和CPU的寄存器位数是一致的**

我们常常听说，CPU主频在计算机的发展过程中飞速提升，从最初的几十 KHz，到后来的几百 MHz，再到现在的 4GHz，终于因为硅晶体的物理特性很难再提升，只能向多核方向发展。在这个过程中，CPU的数据总线宽度也在成倍增长，从早期的8位、16位，到后来的32位，现在我们计算机大部分都在使用64位CPU。

## 数据总线和地址总线

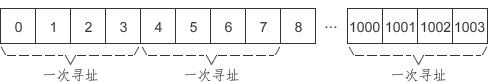
需要注意的是，数据总线和地址总线不是一回事，**数据总线用于在CPU和内存之间传输数据，地址总线用于在内存上定位数据**，它们之间没有必然的联系，宽度并不一定相等。实际情况是，地址总线的宽度往往随着数据总线的宽度而增长，以访问更大的内存。

# C语言内存对齐，提高寻址效率

计算机内存是以字节（Byte）为单位划分的，理论上CPU可以访问任意编号的字节，但实际情况并非如此。

CPU 通过地址总线来访问内存，一次能处理几个字节的数据，就命令地址总线读取几个字节的数据。32 位的 CPU 一次可以处理4个字节的数据，那么每次就从内存读取4个字节的数据；少了浪费主频，多了没有用。64位的处理器也是这个道理，每次读取8个字节。

以32位的CPU为例，实际寻址的步长为4个字节，也就是只对编号为 4 的倍数的内存寻址，例如 0、4、8、12、1000 等，而不会对编号为 1、3、11、1001 的内存寻址。如下图所示：



这样做可以以最快的速度寻址：不遗漏一个字节，也不重复对一个字节寻址。

对于程序来说，一个变量最好位于一个寻址步长的范围内，这样一次就可以读取到变量的值；如果跨步长存储，就需要读取两次，然后再拼接数据，效率显然降低了。

例如一个 int 类型的数据，如果地址为 8，那么很好办，对编号为 8 的内存寻址一次就可以。如果编号为 10，就比较麻烦，CPU需要先对编号为 8 的内存寻址，读取4个字节，得到该数据的前半部分，然后再对编号为 12 的内存寻址，读取4个字节，得到该数据的后半部分，再将这两部分拼接起来，才能取得数据的值。

**将一个数据尽量放在一个步长之内，避免跨步长存储，这称为内存对齐。**在32位编译模式下，默认以4字节对齐；在64位编译模式下，默认以8字节对齐。

# 内存分页机制，完成虚拟地址的映射

这主要是讲解虚拟地址和物理地址之间的映射方法的，一开始采用的是以程序为单位，把一段与程序运行所需要的同等大小的虚拟空间映射到某物理空间。

这种以整个程序为单位的方法很好地解决了不同程序地址不隔离的问题，同时也能够在程序中使用固定的地址。

不过，以程序为单位对虚拟内存进行映射时，如果物理内存不足，被换入换出到磁盘的是整个程序，这样势必会导致大量的磁盘读写操作，严重影响运行速度，所以这种方法还是显得粗糙，粒度比较大。

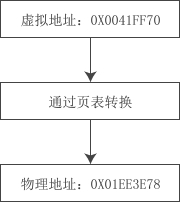
以整个程序为单位进行映射，不仅会将暂时用不到的数据从磁盘中读取到内存，也会将过多的数据一次性写入磁盘，这会严重降低程序的运行效率。

**现代计算机都使用分页（Paging）的方式对虚拟地址空间和物理地址空间进行分割和映射，以减小换入换出的粒度，提高程序运行效率。**

# 内存分页机制的实现

这个内容不太好理解，内存分页机制主要就是解决虚拟地址和物理地址的映射的。

现代操作系统都使用分页机制来管理内存，这使得每个程序都拥有自己的地址空间。每当程序使用虚拟地址进行读写时，都必须转换为实际的物理地址，才能真正在内存条上定位数据。如下图所示：



# 内存管理单元—MMU

在CPU内部，有一个部件叫做MMU（Memory Management Unit，内存管理单元），由它来负责**将虚拟地址映射为物理地址**，如下图所示：

http://c.biancheng.net/cpp/uploads/allimg/160324/1-1603241P132O8.jpg

在页映射模式下，CPU 发出的是虚拟地址，也就是我们在程序中看到的地址，这个地址会先交给 MMU，经过 MMU 转换以后才能变成了物理地址。

即便是这样，MMU也要访问好几次内存，性能依然堪忧，所以在MMU内部又增加了一个缓存，专门用来存储页目录和页表。MMU内部的缓存有限，当页表过大时，也只能将部分常用页表加载到缓存，但这已经足够了，因为经过算法的巧妙设计，可以将缓存的命中率提高到 90%，剩下的10%的情况无法命中，再去物理内存中加载页表。

**CR3 是CPU内部的一个寄存器，专门用来保存页目录的物理地址。**

每个程序在运行时都有自己的一套页表，切换程序时，只要改变 CR3 寄存器的值就能够切换到对应的页表。

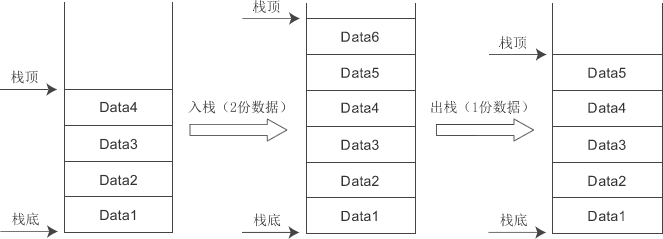
# Linux下C语言程序的内存布局（内存模型）

一个C语言程序的内存在整个地址空间中是如何分布的呢？数据在哪里？代码在哪里？为什么要这样分布？

程序内存在地址空间中的分布情况称为**内存模型**（Memory Model）。内存模型由操作系统构建，在Linux和Windows下有所差异，并且会受到编译模式的影响

# 栈的概念

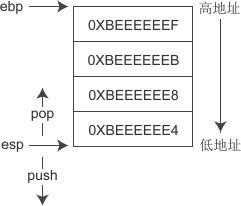
在计算机中，栈可以理解为一个特殊的容器，用户可以将数据依次放入栈中，然后再将数据按照相反的顺序从栈中取出。也就是说，先放入的数据最后才能取出，而最后放入的数据必须先取出。这称为先进后出（First In Last Out）原则。  
  
放入数据常称为入栈或压栈（Push），取出数据常称为出栈或弹出（Pop）。如下图所示：



可以发现，栈底始终不动，出栈入栈只是在移动栈顶，当栈中没有数据时，栈顶和栈底重合。  
  
从本质上来讲，栈是一段连续的内存，需要同时记录栈底和栈顶，才能对当前的栈进行定位。在现代计算机中，通常使用ebp寄存器指向栈底，而使用esp寄存器指向栈顶。随着数据的进栈出栈，esp 的值会不断变化，进栈时 esp 的值减小，出栈时 esp 的值增大。

**ebp 和 esp 都是CPU中的寄存器**：ebp 是 Extend Base Pointer 的缩写，通常用来指向栈底；esp 是 Extend Stack Pointer 的缩写，通常用来指向栈顶。

如下图所示是一个栈的实例：



## 栈的大小以及栈溢出

栈的内存是有限的，而且它的大小是跟编译器有关，编译器会为栈内存指定一个最大值，不同的编译器下栈的内存是不同的，在Linux GCC下默认的是8M。

栈也称为堆栈，堆依然是堆，所以堆栈并不包含堆。

# 一个在栈上的函数（类比Java虚拟机栈）

函数的调用和栈是分不开的（在Java中，一个线程的开始就会产生一个Java虚拟机栈，函数的调用到结束对应着一个栈帧的入栈和出栈。）

当发生函数调用时，会将函数运行需要的信息全部压入栈中，这个被称为**栈帧**或者**活动记录**

**栈帧一般包括以下几个东西：**

1. **函数的返回地址**

什么是返回地址，就是函数执行完成后从哪里开始继续执行后面的代码，比如以下代码



站在C语言的角度，fun函数执行完成后会继续执行c=a+b语句，那么这个返回地址就是指的这个c=a+b语句的内存地址。

注意：C语言代码最终会被编译为机器指令，确切地说，返回地址应该是下一条指令的地址，这里之所以说是下一条C语言语句的地址，仅仅是为了更加直观地说明问题。

**2、参数和局部变量**

有些编译器，或者编译器在开启优化选项的情况下，会通过寄存器来传递参数，而不是将参数压入栈中，我们暂时不考虑这种情况。

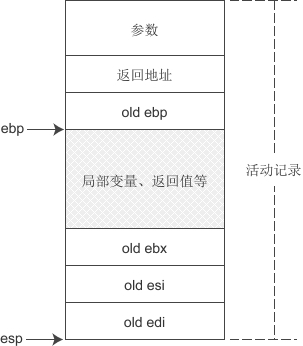
**3、编译器自动生成的临时数据**

当函数返回值的长度较大（比如占用40个字节）时，会先将返回值压入栈中，然后再交给函数调用者。

当返回值的长度较小（char、int、long 等）时，不会被压入栈中，而是先将返回值放入寄存器，再传递给函数调用者。

**4、一些需要保存的寄存器**

例如 ebp、ebx、esi、edi 等。之所以要保存寄存器的值，是为了在函数退出时能够恢复到函数调用之前的场景，继续执行上层函数。



## 如何定位数据

由于 esp 的值会随着数据的入栈而不断变化，要想根据 esp 找到参数、局部变量等数据是比较困难的，所以在实现上是根据 ebp 来定位栈内数据的。ebp 的值是固定的，数据相对 ebp 的偏移也是固定的，ebp 的值加上偏移量就是数据的地址。

# C语言动态内存分配

在进程的地址空间中，代码区、常量区、全局数据区的内存在程序启动时就已经分配好了，它们大小固定，不能由程序员分配和释放，只能等到程序运行结束由操作系统回收。这称为静态内存分配。  
  
栈区和堆区的内存在程序运行期间可以根据实际需求来分配和释放，不用在程序刚启动时就备足所有内存。这称为动态内存分配。  
  
使用静态内存的优点是速度快，省去了向操作系统申请内存的时间，缺点就是不灵活，缺乏表现力，例如不能控制数据的作用范围，不能使用较大的内存。而使用动态内存可以让程序对内存的管理更加灵活和高效，需要内存就立即分配，而且需要多少就分配多少，从几个字节到几个GB不等；不需要时就立即回收，再分配给其他程序使用。

## 堆和栈的区别

栈区和堆区的管理模式有所不同：栈区内存由系统分配和释放，不受程序员控制；堆区内存完全由程序员掌控，想分配多少就分配多少，想什么时候释放就什么时候释放，非常灵活。  
  
程序启动时会为栈区分配一块大小适当的内存，对于一般的函数调用这已经足够了，函数进栈出栈只是 ebp、esp 寄存器指向的变换，或者是向已有的内存中写入数据，不涉及内存的分配和释放。当函数中有较大的局部数组时，比如 1024\*10 个元素，编译器就会在函数代码中插入针对栈的动态内存分配函数，这样函数被调用时才分配内存，不调用就不分配。  
  
我们经常听说“栈内存的分配效率要高于堆”就是这个道理，因为大部分情况下并没有真的分配栈内存，仅仅是对已有内存的操作。

## 动态内存分配函数

堆（Heap）是唯一由程序员控制的内存区域，我们常说的动态内存分配也是在这个区域。在堆上分配和释放内存需要用到C语言标准库中的几个函数：[malloc()](http://c.biancheng.net/cpp/html/137.html)、[calloc()](http://c.biancheng.net/cpp/html/134.html)、[realloc()](http://c.biancheng.net/cpp/html/2859.html) 和 [free()](http://c.biancheng.net/cpp/html/135.html)。

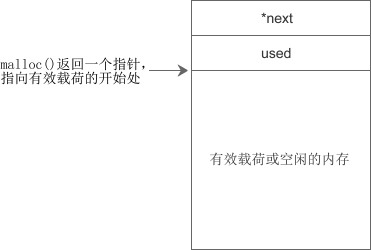
# malloc()背后的实现原理——内存池

相对于栈而言，堆这片内存面临着一个稍微复杂的行为模式：在任意时刻，程序可能发出请求，要么申请一段内存，要么释放一段已经申请过的内存，而且申请的大小从几个字节到几个GB都有可能，我们不能假设程序一次申请多少堆空间，因此，堆的管理显得较为复杂。  
  
那么，使用 malloc() 在堆上分配内存到底是如何实现的呢？  
  
一种做法是把 malloc() 的内存管理交给系统内核去做，既然内核管理着进程的地址空间，那么如果它提供一个系统调用，可以让 malloc() 使用这个系统调用去申请内存，不就可以了吗？当然这是一种理论上的做法，但实际上这样做的性能比较差，因为每次程序申请或者释放堆空间都要进行系统调用。我们知道系统调用的性能开销是比较大的，当程序对堆的操作比较频繁时，这样做的结果会严重影响程序的性能。  
  
比较好的做法就是 malloc() 向操作系统申请一块适当大小的堆空间，然后由 malloc() 自己管理这块空间。  
  
malloc() 相当于向操作系统“批发”了一块较大的内存空间，然后“零售”给程序用。当全部“售完”或程序有大量的内存需求时，再根据实际需求向操作系统“进货”。当然 malloc() 在向程序零售堆空间时，必须管理它批发来的堆空间，不能把同一块地址出售两次，导致地址的冲突。于是 malloc() 需要一个算法来管理堆空间，这个算法就是堆的**分配算法**。

在程序运行过程中，堆内存从低地址向高地址连续分配，随着内存的释放，会出现不连续的空闲区域，如下图所示：

http://c.biancheng.net/cpp/uploads/allimg/160421/1-1604211416312C.jpg  
图1：已分配内存和空闲内存相间出现

带阴影的方框是已被分配的内存，白色方框是空闲内存或已被释放的内存。程序需要内存时，malloc() 首先遍历空闲区域，看是否有大小合适的内存块，如果有，就分配，如果没有，就向操作系统申请（发生系统调用）。为了保证分配给程序的内存的连续性，malloc() 只会在一个空闲区域中分配，而不能将多个空闲区域联合起来。  
  
内存块（包括已分配和空闲的）的结构类似于链表，它们之间通过指针连接在一起。在实际应用中，一个内存块的结构如下图所示：



链表是一种经典的堆内存管理方式，经常被用在教学中，很多C语言教程都会提到“栈内存的分配类似于数据结构中的栈，而堆内存的分配却类似于数据结构中的链表”就是源于此。  
  
链表式内存管理虽然思路简单，容易理解，但存在很多问题，例如：

* 一旦链表中的 pre 或 next 指针被破坏，整个堆就无法工作，而这些数据恰恰很容易被越界读写所接触到。
* 小的空闲区域往往不容易再次分配，形成很多内存碎片。
* 经常分配和释放内存会造成链表过长，增加遍历的时间。

针对链表的缺点，后来人们提出了位图和对象池的管理方式，而现在的 malloc() 往往采用多种方式复合而成，不同大小的内存块往往采用不同的措施，以保证内存分配的安全和效率。

## 内存池

不管具体的分配算法是怎样的，为了减少系统调用，减少物理内存碎片，malloc() 的整体思想是先向操作系统申请一块大小适当的内存，然后自己管理，这就是内存池（Memory Pool）。  
  
内存池的研究重点不是向操作系统申请内存，而是对已申请到的内存的管理，这涉及到非常复杂的算法，是一个永远也研究不完的课题，除了C标准库自带的 malloc()，还有一些第三方的实现，比如 Goolge 的 tcmalloc 和 jemalloc。  
  
我们知道，C/C++是编译型语言，没有内存回收机制，程序员需要自己释放不需要的内存，这在给程序带来了很大灵活性的同时，也带来了不少风险，例如C/C++程序经常会发生内存泄露，程序刚开始运行时占用内存很少，随着时间的推移，内存使用不断增加，导致整个计算机运行缓慢。  
  
内存泄露的问题往往难于调试和发现，或者只有在特定条件下才会复现，这给代码修改带来了不少障碍。为了提高程序的稳定性和健壮性，后来的 Java、Python、C#、JavaScript、PHP 等使用了虚拟机机制的非编译型语言都加入了垃圾内存自动回收机制，这样程序员就不需要管理内存了，系统会自动识别不再使用的内存并把它们释放掉，避免内存泄露。可以说，这些高级语言在底层都实现了自己的内存池，也即有自己的内存管理机制。

## 池化技术

在计算机中，有很多使用“池”这种技术的地方，除了内存池，还有连接池、线程池、对象池等。以服务器上的线程池为例，它的主要思想是：先启动若干数量的线程，让它们处于睡眠状态，当接收到客户端的请求时，唤醒池中某个睡眠的线程，让它来处理客户端的请求，当处理完这个请求，线程又进入睡眠状态。  
  
所谓“池化技术”，就是程序先向系统申请过量的资源，然后自己管理，以备不时之需。之所以要申请过量的资源，是因为每次申请该资源都有较大的开销，不如提前申请好了，这样使用时就会变得非常快捷，大大提高程序运行效率。

# 重点解释一下

**栈内存的分配类似于数据结构中的栈，而堆内存的分配却类似于数据结构中的链表**

**为啥？**

# 野指针

如果一个指针指向的内存没有访问权限，或者指向一块已经释放掉的内存，那么就无法对该指针进行操作，这样的指针称为野指针（Wild Pointer）。

## 规避野指针

要想规避野指针，就要养成良好的编程习惯：  
1) 指针变量如果暂时不需要赋值，一定要初始化为NULL，因为任何指针变量刚被创建时不会自动成为NULL指针，它的缺省值是随机的。  
  
2) 当指针指向的内存被释放掉时，要将指针的值设置为 NULL，因为 free() 只是释放掉了内存，并为改变指针的值。

# C语言内存泄漏（内存丢失）

使用 malloc()、calloc()、realloc() 动态分配的内存，如果没有指针指向它，就无法进行任何操作，这段内存会一直被程序占用，直到程序运行结束由操作系统回收。

第一次分配 100 字节的内存，并将 p 指向它；第二次分配 50 字节的内存，依然使用 p 指向它。  
  
这就导致了一个问题，第一次分配的 100 字节的内存没有指针指向它了，而且我们也不知道这块内存的地址，所以就再也无法找回了，也没法释放了，这块内存就成了垃圾内存，虽然毫无用处，但依然占用资源，唯一的办法就是等程序运行结束后由操作系统回收。  
  
这就是内存泄露（Memory Leak），可以理解为程序和内存失去了联系，再也无法对它进行任何操作。  
  
内存泄漏形象的比喻是“操作系统可提供给所有程序使用的内存空间正在被某个程序榨干”，最终结果是程序运行时间越长，占用内存空间越来越多，最终用尽全部内存空间，整个系统崩溃。

free() 函数的用处在于实时地回收内存，如果程序很简单，程序结束之前也不会使用过多的内存，不会降低系统的性能，那么也可以不用写 free() 函数。当程序结束后，操作系统会释放内存。  
  
但是如果在开发大型程序时不写 free() 函数，后果是很严重的。这是因为很可能在程序中要重复一万次分配10MB的内存，如果每次进行分配内存后都使用 free() 函数去释放用完的内存空间， 那么这个程序只需要使用10MB内存就可以运行。但是如果不使用 free() 函数，那么程序就要使用100GB 的内存！这其中包括绝大部分的虚拟内存，而由于虚拟内存的操作需要读写磁盘，因此，这样会极大地影响到系统的性能，系统因此可能崩溃。  
  
因此，在程序中使用 malloc() 分配内存时都对应地写出一个 free() 函数是一个良好的编程习惯。这不但体现在处理大型程序时的必要性，并能在一定程度上体现程序优美的风格和健壮性。