作 者：道哥，10+年的嵌入式开发老兵。

公众号：【IOT物联网小镇】，专注于：C/C++、Linux操作系统、应用程序设计、物联网、单片机和嵌入式开发等领域。 公众号回复【书籍】，获取 Linux、嵌入式领域经典书籍。

前两篇文章，我们一起学习了 8086 处理器中关于 CPU、内存的基本使用方式，重点对段寄存器和内存的寻址方式进行了介绍。

可能有些小伙伴会对此不屑：现在都是多核的现代处理器，操作系统已经变得非常的强大，为何还去学习这些古董知识？

前几天看到下面这段话，可以来回答这个问题：

“我们都希望学习最新的、使用的东西，但学习的过程是客观的。”

“任何合理的学习过程（尽可能排除走弯路、盲目探索、不成系统）都是一个循序渐进的过程。”

“我们必须先通过一个易于全面把握的事物，来学习和探索一般的规律和方法。”

就拿学习 Linux 操作系统来说，作为一个长期的学习计划，不太可能一上来就阅读最新的 Linux 5.13 版本的代码。

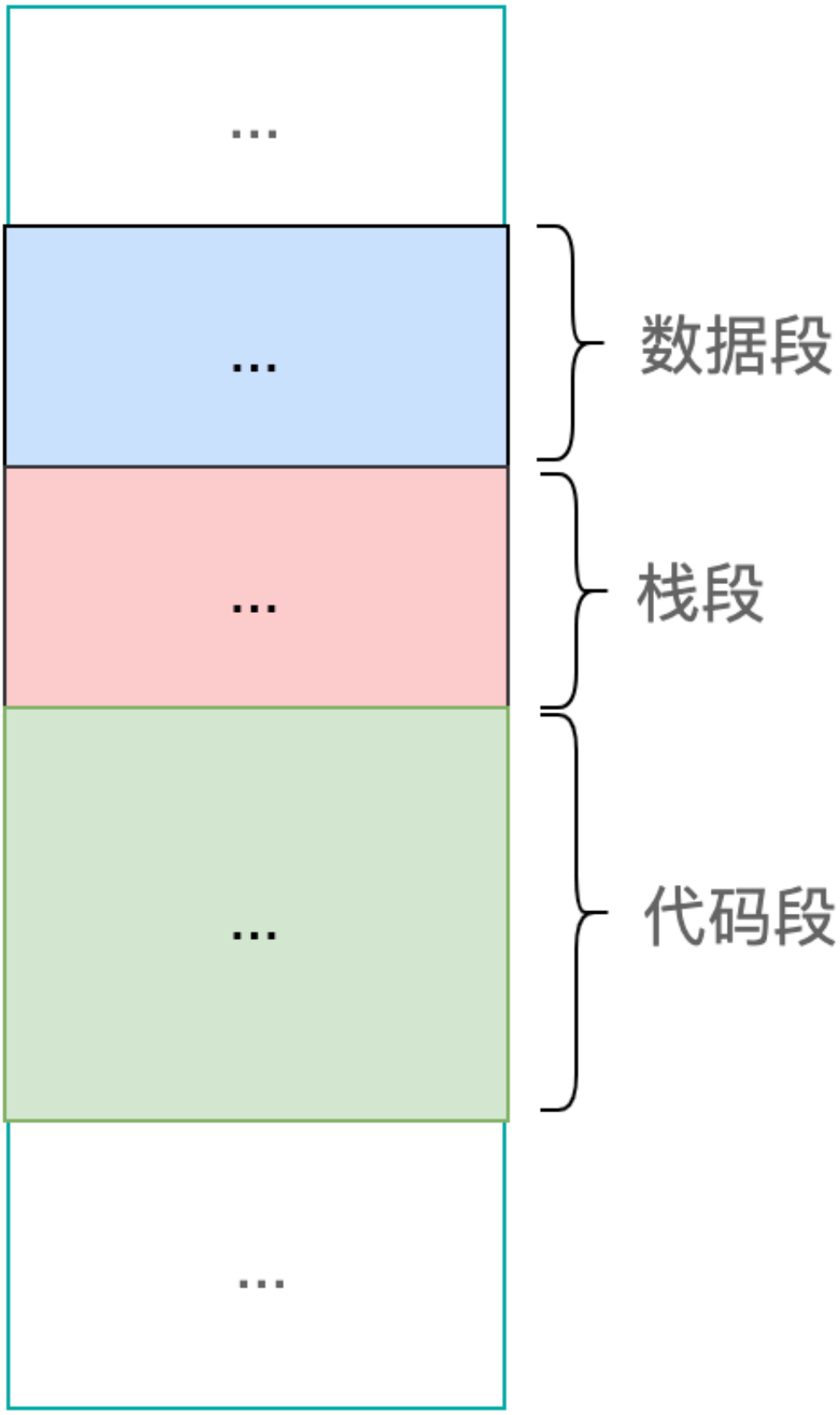
更有可能是先学习 0.11 版本，理解了其中的一些原理、思想之后，再循序渐进的向高版本进行学习、探索。

那么对于 《Linux 从头学》这个系列的文章来说，我是希望自己能够把学习路线再拉长一些，从更底层的硬件机制、驱动原理开始，由简入繁，一步一步最终把 Linux 操作系统这个块硬骨头给啃下来。

那么今天我们就继续 8086 下的学习，来看看一个相对“完整”程序的基本结构。

几个重要的段寄存器

在 x86 系统中，段寻址机制以及相关的寄存器是如此的重要，以至于我忍不住在这里，把几个段寄存器再小结一下。



> 代码段：用来存放代码，段的基地址放在寄存器 CS 中，指令指针寄存器 IP 用来表示下一条指令在段中的偏移地址;

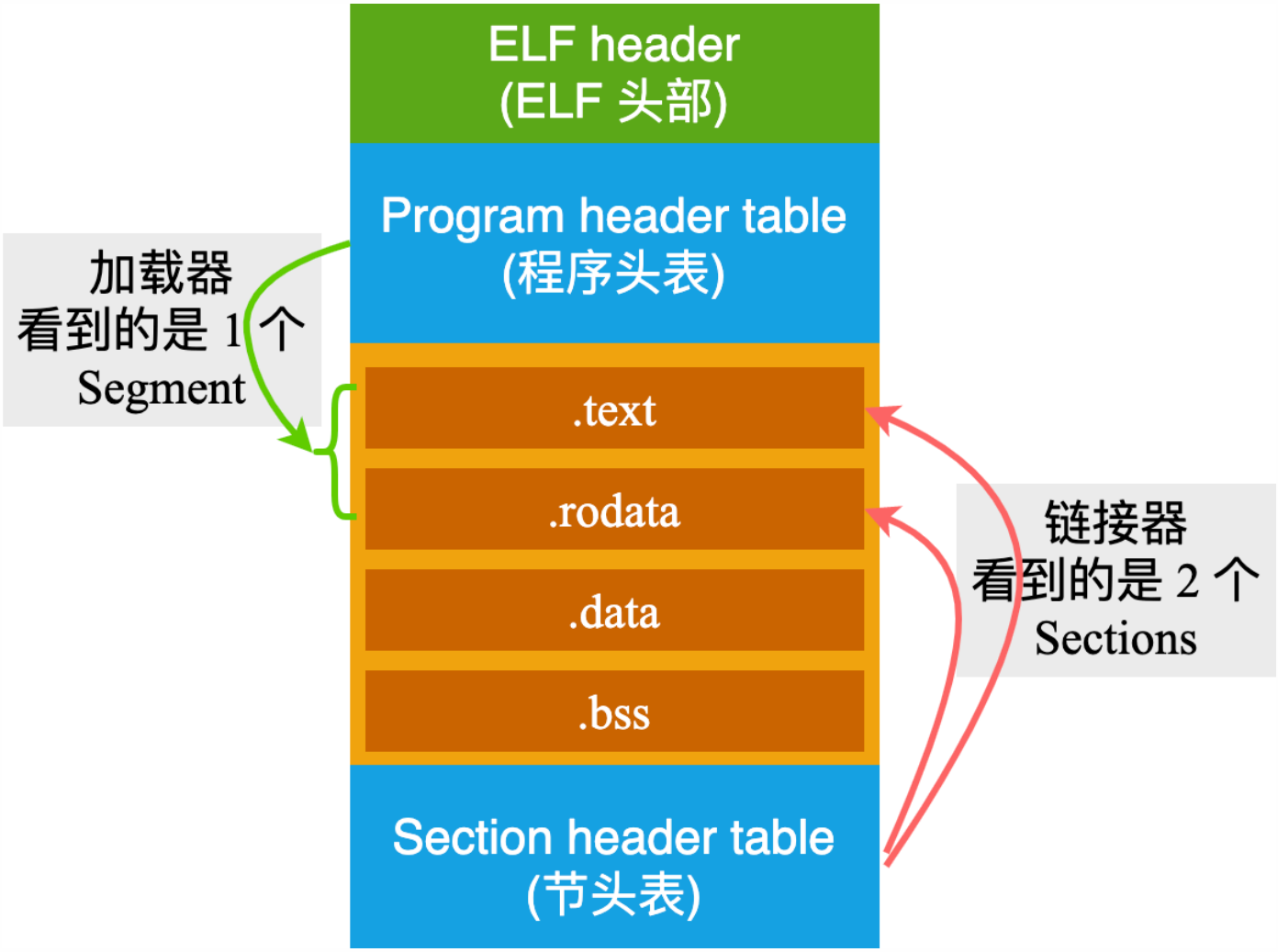
> 数据段：用来存放程序处理的数据，段的基地址存放在寄存器 DS 中。对数据段中的某个数据进行操作时，直接在汇编代码中通过立即数或寄存器来指定偏移地址；
>
> 栈段：本质上也是用来存放数据，只不过它的操作方式比较特殊而已：通过 PUSH 和 POP 指令来进行操作。段的基地址存放在寄存器 SS 中，栈顶单元的偏移地址存放在寄存器 IP 中。

这里的段，本质上是我们把内存上的某一块连续的存储空间，专门存储某一类的数据。

我们之所以能够这么做，是因为 CPU 通过以上几个寄存器，让我们这样的"安排"称为一种可能。

一句话总结：CPU 将内存中的某个段的内容当做代码，是因为 CS:IP 指向了那里；CPU 将某个段当做栈，是因为 CS:SP 指向了那里。

在之前的一篇文章中，演示了一个 ELF 格式的可执行文件中，具体包含了哪些段《Linux系统中编译、链接的基石-ELF文件：扒开它的层层外衣，从字节码的粒度来探索》：

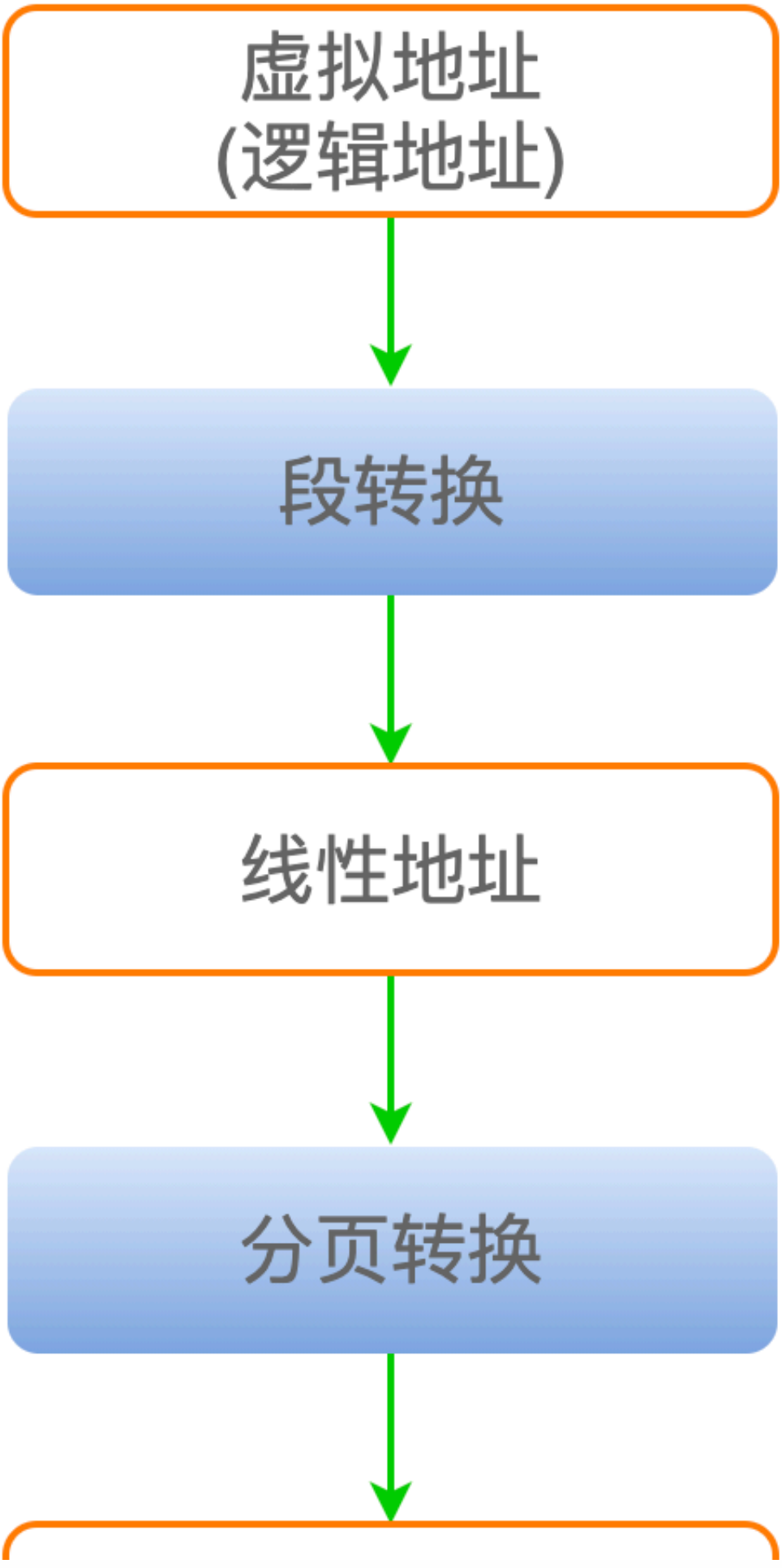


虽然这张图中描述的段结构更复杂，但是从本质上来说，它与 8086 中描述的段结构是一样的！

Linux 2.6 中的线性地址区间

在一个现代操作系统中，一个进程中使用的的地址空间，一般称作虚拟地址(也称作逻辑地址)。

虚拟地址首先经过段转换，得到线性地址；然后线性地址再经过分页转换，得到最终的物理地址。


虚拟地址
(逻辑地址)
段转换
线性地址
分页转换

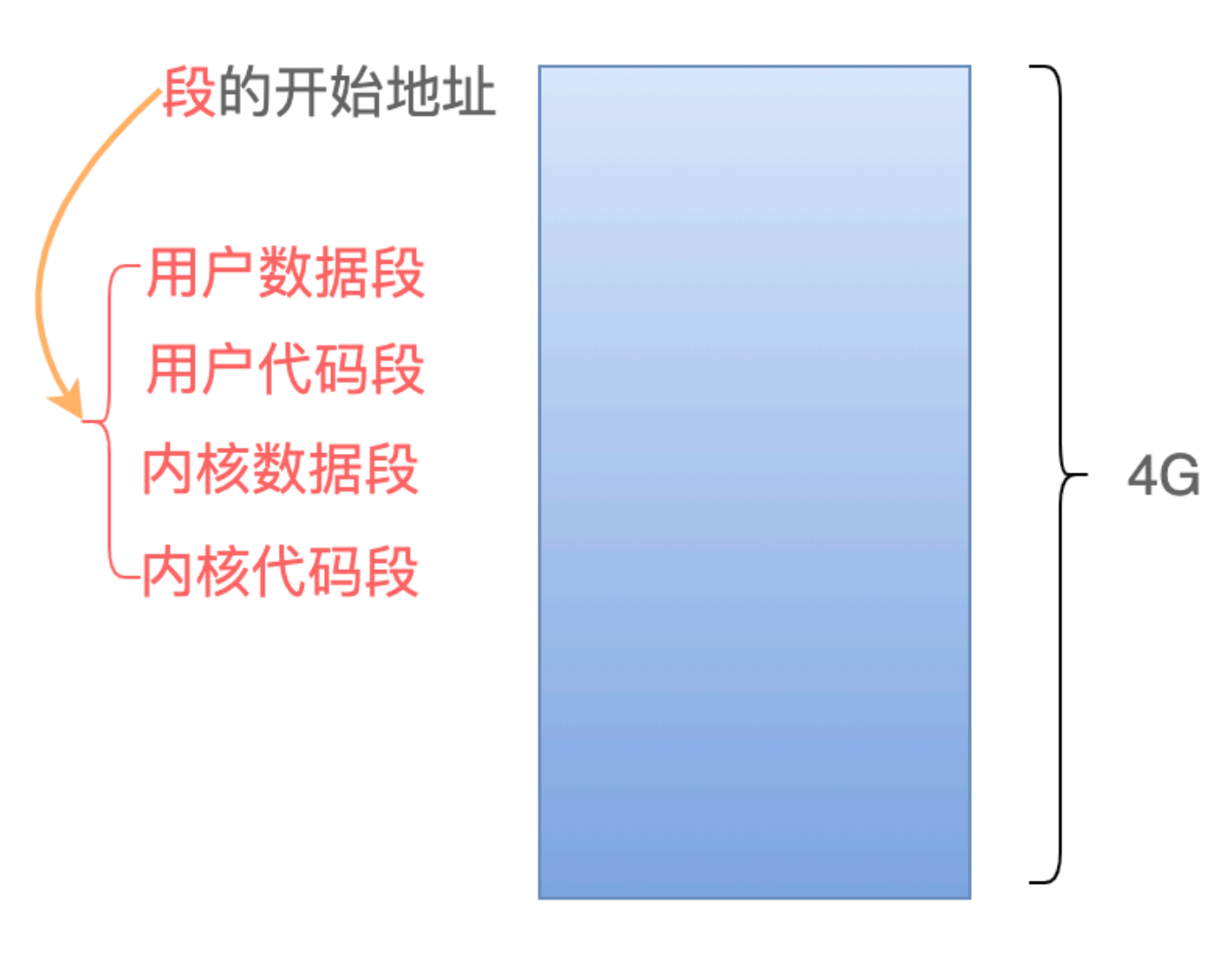
> 这里再啰嗦一下，很多书籍中队内存地址的称呼比较多，都是根据作者的习惯来称呼。
>
> 我是按照上图的方式来理解的：编译器产生的地址叫做虚拟地址，也叫做逻辑地址，然后经过两级转换，得到最终的物理地址。

在 Linux 2.6 代码中，由于 Linux 把整个 4 GB 的地址空间当做一个“扁平”的结果来处理(段的基地址是 0x0000_0000，偏移地址的最大值是 4GB)，因此虚拟地址(逻辑地址)在数值上等于线性地址。

我们再结合上次给出的这张图来理解：

段	Base	G	Limit	S	Type	DPL	D/B	P
用户代码段	0x00000000	1	0xfffff	1	10	3	1	1
用户数据段	0x00000000	1	0xfffff	1	2	3	1	1
内核代码段	0x00000000	1	0xfffff	1	10	0	1	1
内核数据段	0x00000000	1	0xfffff	1	2	0	1	1

这张图的意思是：在 Linux 2.6 中，用户代码段的开始地址是 0，最大范围是 4 GB；用户数据段的开始地址是 0，最大范围也是 4 GB；内核的数据段和代码段也是如此。



> 为什么：虚拟地址(逻辑地址)在数值上等于线性地址?
>
> 线性地址 = 段基址 + 虚拟地址(偏移量)，因为段基址为 0 ，所以线性地址在数值上等于虚拟地址。

Linux 之所以要这样安排，是因为它不想过多的利用 x86 提供的段机制来进行内存地址的管理，而是想充分利用分页机制来进行更加灵活的地址管理。

还有一点需要提醒一下：

在上述描述的文字中，我都会标明一个机制或者策略，它是由 x86 平台提供的，还是由 Linux 操作系统提供的。

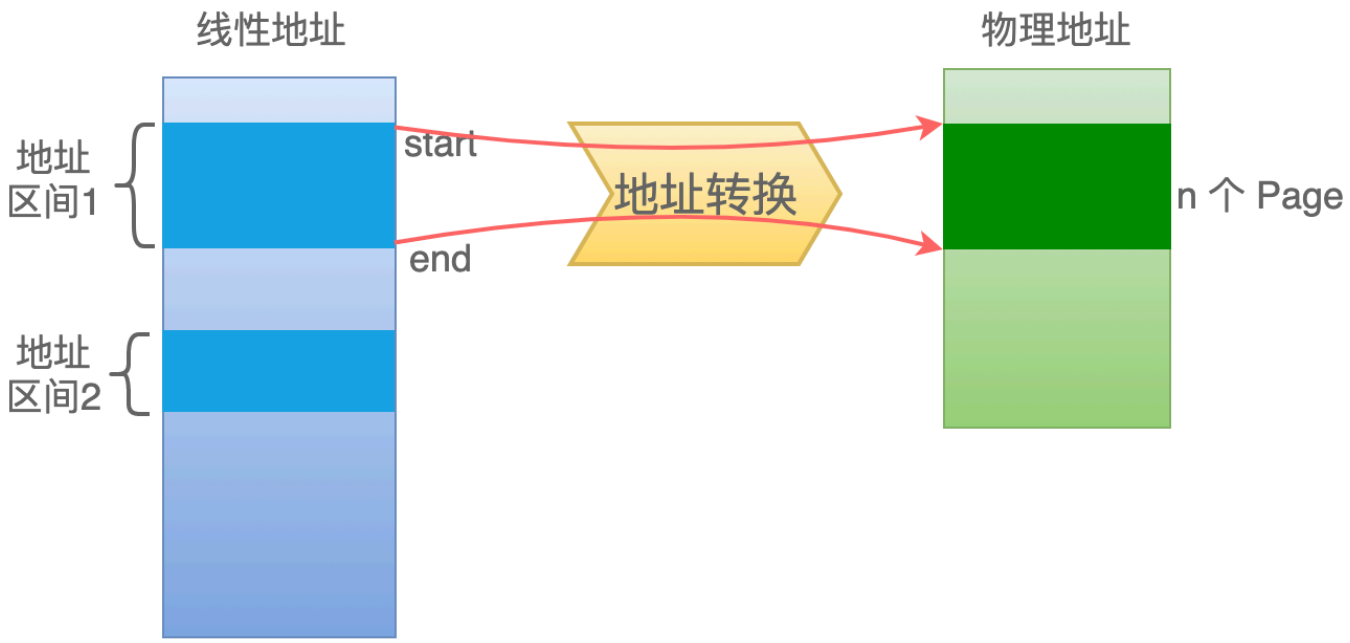
对于分页机制也是如此，x86 硬件提供了分页机制，但是 Linux 在 x86 提供的这个分页机制的基础上，进行了扩展，以达到更加灵活的内存地址管理目的。

因此，各位小伙伴在看一些书籍的时候，心中要有一个谱：当前描述内容的上下文环境是什么。

当我们创建一个进程的时候，在内核中就会记录这个进程所拥有的所有线性地址区间。

进程所拥有的所有线性地址区间是一个动态的过程，根据程序的需求随时进行扩展或缩小。例如：把一个文件映射到内存，动态加载/卸载一个动态库等等。

我们知道，内核在操作物理内存的时候，是通过“页框”这个单位来管理的。



一个页框可以包含 1-n 个页，每一页的大小一般是 4 KB，这是对物理内存的管理。

一个线性地址区间可以包含多个物理页。每一个线性地址最终通过多级的页表转换，来最终得到一个物理地址。

注意：上图中，线性地址区间1，映射到物理地址空间中的 N 个 Page，这些 Page 有可能是连续的，也有可能不是连续的。

虽然在物理内存中是不连续的，但是由于被分页转换机制进行了屏蔽，我们在应用程序中都是按照连续的空间来使用的。

一个“完整”的 8086 汇编程序

我们再继续回到 8086 系统中来。

这里描述的地址，经过段地址转换之后，就是一个物理地址，没有经过复杂的页表转换。

这也是我们以 8086 系统作为学习平台的目的：抛开复杂的操作系统，直接探索底层的东西。

在这个最简单的汇编程序中，会使用到 3 个段：代码段，数据段和栈段。

前面已经说到：所谓的段，就是一个地址空间。既然是一个地址空间，必然包含 2 个元素：从什么地方开始，长度是多少。

还是直接上代码：

```
assume ds:addr1, ss:addr2, cs:addr3

addr1 segment           ; 把数据段安排在这个位置
        db 32 dup (0)   ; 这 32 个字节，是数据段的大小
addr1 end

addr2 segment           ; 把栈段安排在这个位置
```

```
        db 32 dup(0)    ; 这 32 个字节，是栈段的大小
addr2 end

addr3 segment           ; 把代码段安排在这个位置
start
        mov ax, addr1
        mov ds, ax      ; 设置数据段寄存器

        mov ax, addr2
        mov ss, ax      ; 设置栈段寄存器
        mov sp, 20h     ; 设置栈顶指针寄存器

        ...             ; 其他代码
addr3 ends

end start
```

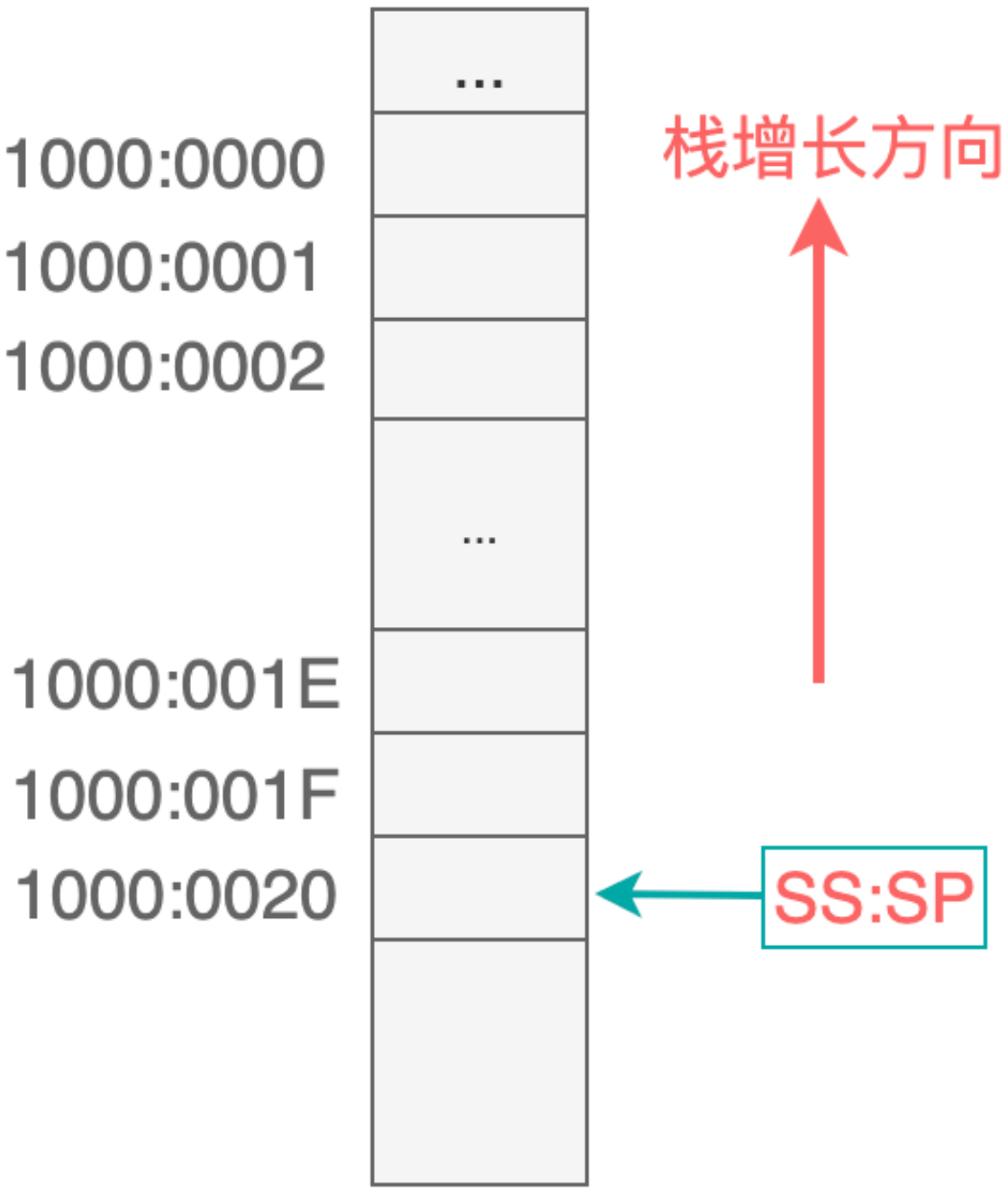
以上就是一个汇编代码的基本程序结构，我们给它安排了 3 个段。

3 个标号：addr1、addr2 和 addr3，代表了每一个段的开始地址。在代码段的开始部分，把数据段标号 addr1 代表的地址，赋值给 DS 寄存器；把栈段标号 addr2 代表的地址，赋值给 SS 寄存器。

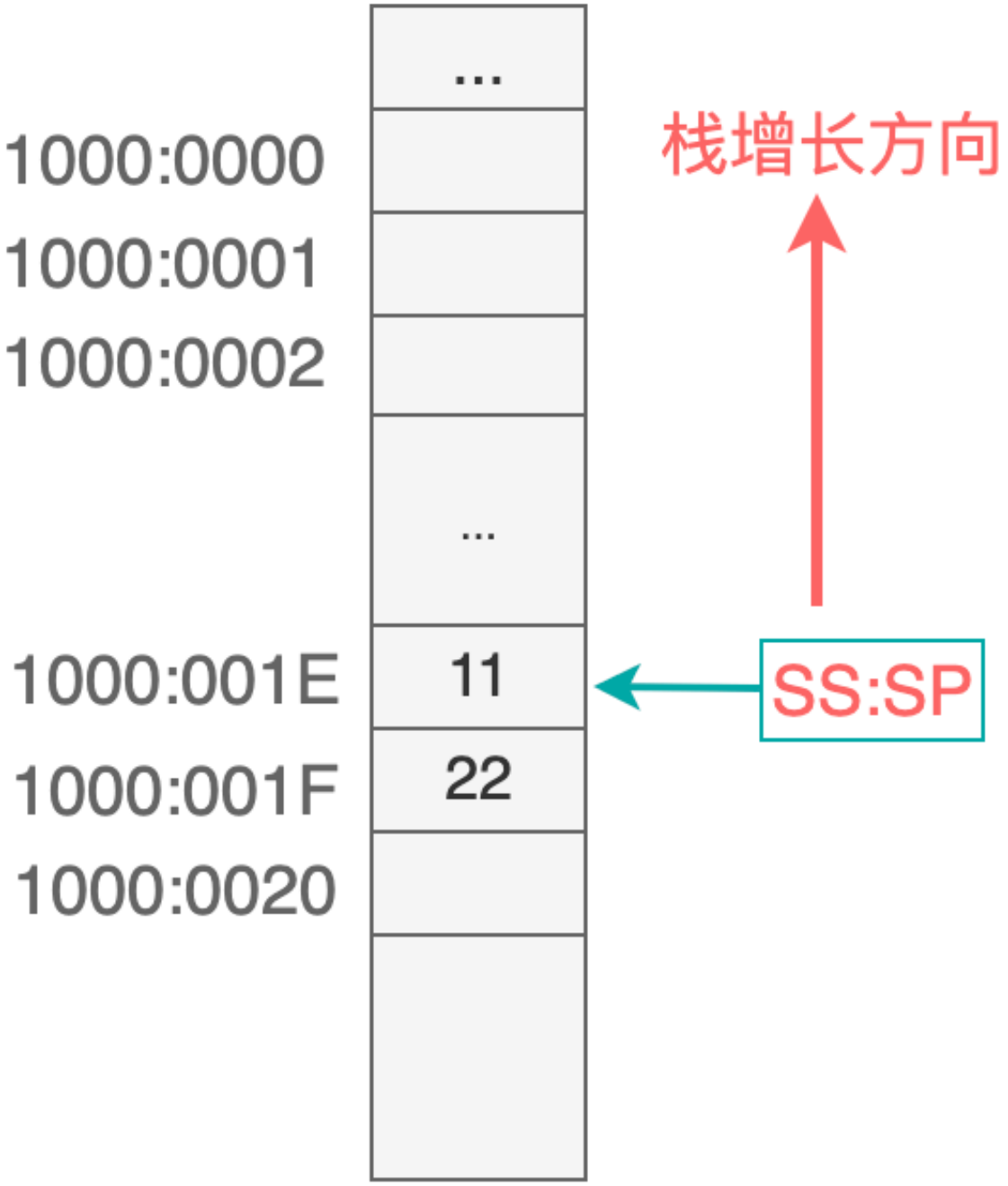
> 这里的标号，是不是与 C 语言中的 goto 标号很类似？ 都是表示一个地址。

注意这里赋值给栈顶指针 SP 寄存器的值是 20H。

因为栈段的使用是从高地址向低地址方向进行的，所以需要把栈顶指针设置为最大地址单元的下一个地址空间。



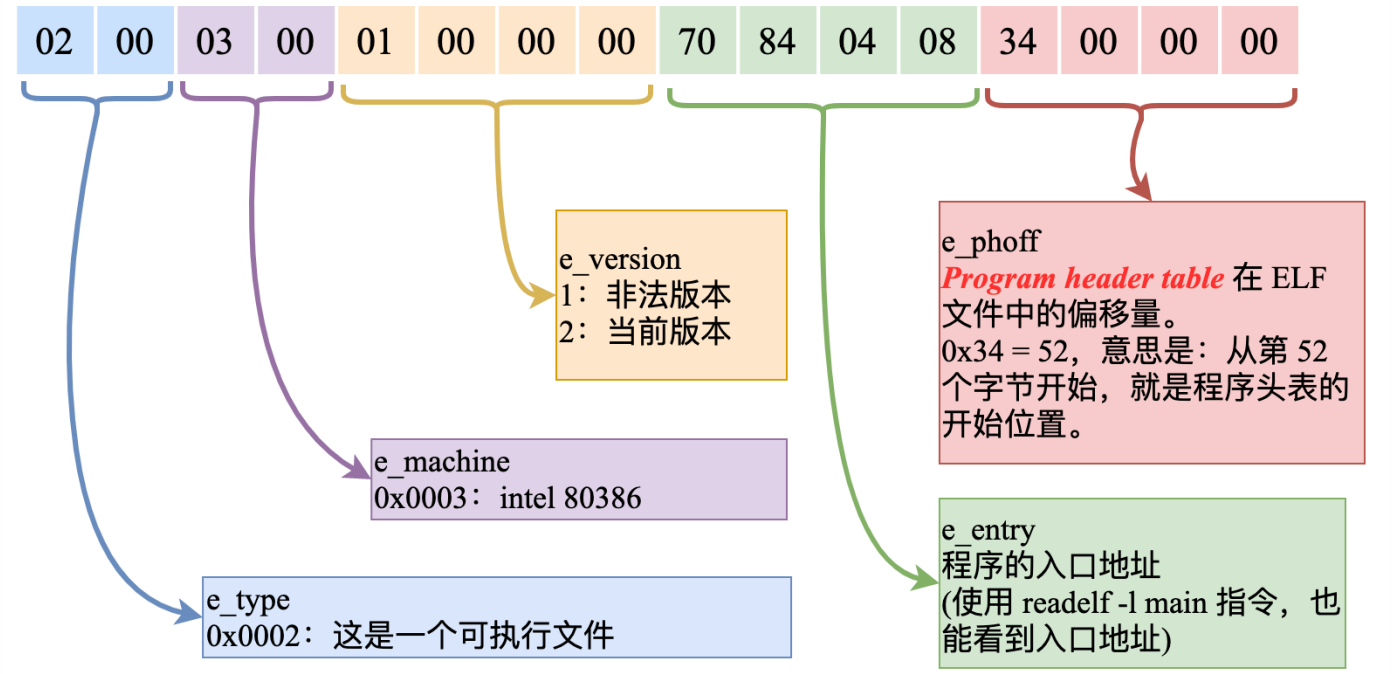
假设把第一个数据入栈时(eg: 先执行 `mov ax, 1234h`，再执行 `push ax`)，CPU 要做的事情是: 先执行 `SP = SP - 2`，此时 `SS:SP` 指向 `1000:001E`，然后再把 `1234h` 存储到这个地址空间:



另外，代码中最后一句 `end start`，用来告诉编译器：代码段中 `start` 标号代表的地址，就是这个程序的入口地址，编译之后这个入口地址信息也会被写入可执行程序中。

当可执行文件被加载到内存中之后，加载程序会找到这个入口地址，然后把 CS:IP 设置为指向这个入口地址，从而开始执行第一条指令。

我们再来对比一下《Linux系统中编译、链接的基石-ELF文件：扒开它的层层外衣，从字节码的粒度来探索》中列出的 `ELF` 可执行文件中的入口地址，它与上面 `8086` 下的 `start` 标号代表的入口地址，在本质上都是一样的道理：



------ End ------

推荐阅读

【1】C语言指针-从底层原理到花式技巧，用图文和代码帮你讲解透彻
【2】一步步分析-如何用C实现面向对象编程
【3】原来gdb的底层调试原理这么简单
【4】内联汇编很可怕吗？看完这篇文章，终结它！

其他系列专辑：精选文章、C语言、Linux操作系统、应用程序设计、物联网

公众号【IOT物联网小镇】

C/C++、物联网、嵌入式、Lua语言

Linux 操作系统、应用程序开发设计

扫码关注公众号

道哥 个人微信

喜欢请分享，满意点个赞，最后点在看。