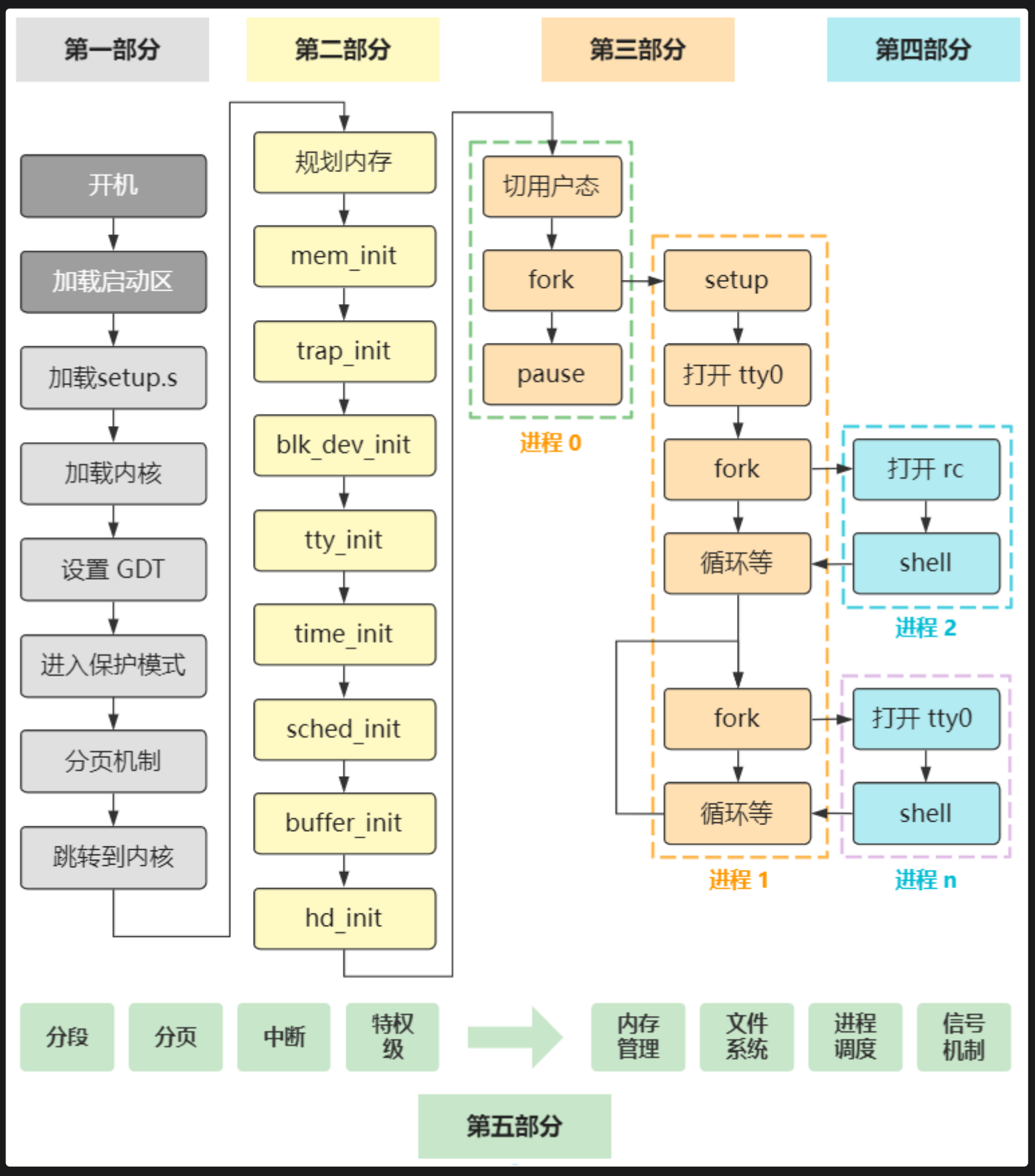
第六回

新读者看这里，老读者直接跳过。

本系列会以一个读小说的心态，从开机启动后的代码执行顺序，带着大家阅读和赏析 Linux 0.11 全部核心代码，了解操作系统的技术细节和设计思想。



你会跟着我一起，看着一个操作系统从啥都没有开始，一步一步最终实现它复杂又精巧的设计，读完这个系列后希望你能发出感叹，原来操作系统源码就是这破玩意。

以下是**已发布文章**的列表，详细了解本系列可以先从开篇词看起。

开篇词

第一回 | 最开始的两行代码

第二回 | 自己给自己挪个地儿

第三回 | 做好最最基础的准备工作

第四回 | 把自己在硬盘里的其他部分也放到内存来

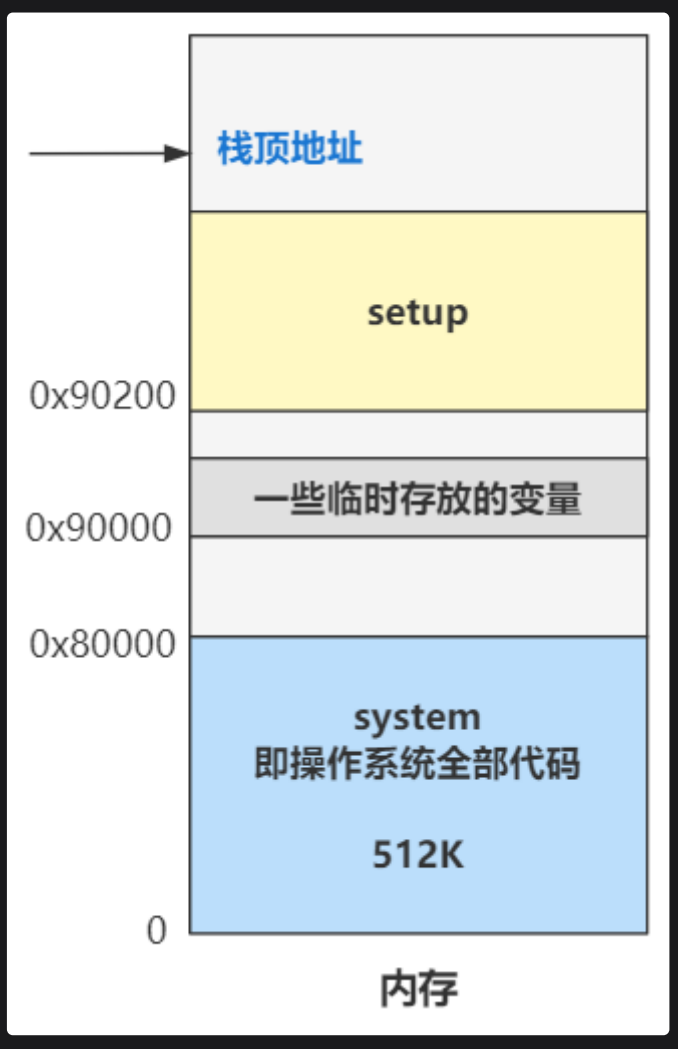
第五回 | 进入保护模式前的最后一次折腾内存

本系列的 GitHub 地址如下（文末阅读原文可直接跳转）

https://github.com/sunym1993/flash-linux0.11-talk

------- 正文开始 -------

书接上回，上回书咱们说到，操作系统又折腾了一下内存，之后的很长一段时间内存布局就不会变了，终于稳定下来了，目前它长这个样子。



0 地址开始处存放着操作系统的全部代码吗，也就是 system 模块，0x90000 位置处往后的几十个字节存放着一些设备的信息，方便以后使用。

内存地址	长度(字节)	名称
0x90000	2	光标位置
0x90002	2	扩展内存数
0x90004	2	显示页面
0x90006	1	显示模式
0x90007	1	字符列数
0x90008	2	未知
0x9000A	1	显示内存
0x9000B	1	显示状态
0x9000C	2	显卡特性参数
0x9000E	1	屏幕行数
0x9000F	1	屏幕列数
0x90080	16	硬盘 1 参数表
0x90090	16	硬盘 2 参数表
0x901FC	2	根设备号

是不是十分清晰？不过别高兴得太早，清爽的内存布局，是方便后续操作系统的大显身手！

接下来就要进行真正的第一项大工程了，那就是**模式的转换**，需要从现在的 16 位的**实模式**转变为之后 32 位的**保护模式**。

当然，虽说是一项非常难啃的大工程，但从代码量看，却是少得可怜，所以不必太过担心。

每次讲这里都十分的麻烦，因为这是 **x86 的历史包袱**问题，现在的 CPU 几乎都是支持 32 位模式甚至 64 位模式了，很少有还仅仅停留在 16 位的实模式下的 CPU。所以我们要为了这个历史包袱，**写一段模式转换的代码**，如果 Intel CPU 被重新设计而不用考虑兼容性，那么今天的代码将会减少很多甚至不复存在。

所以不用担心，听懂就听懂，听不懂就拉倒，放宽心。

我不打算直接说实模式和保护模式的区别，我们还是跟着代码慢慢品味，来。

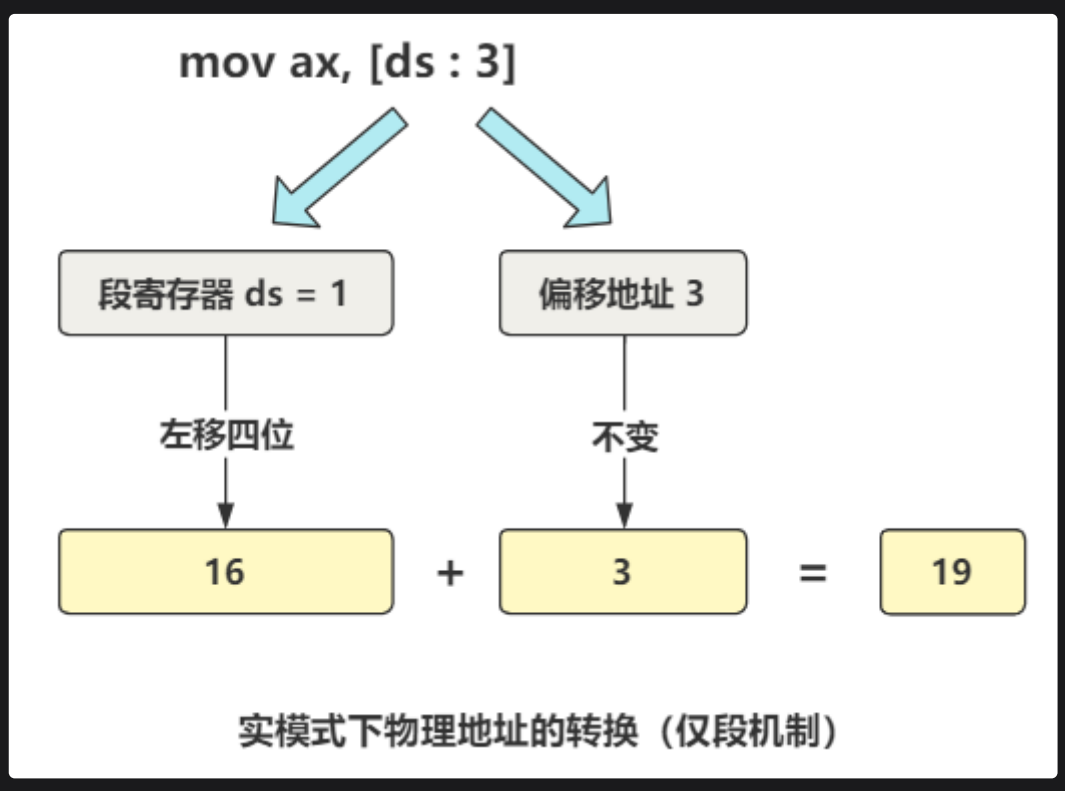
这里仍然是 setup.s 文件中的代码咯。

```
lidt  idt_48     ; load idt with 0,0lgdt  gdt_48     ; load gdt with whatever appropriateidt_48:    .word   0    ; idt limit=0    .word   0,0   ; idt base=0L
```

上来就是两行看不懂的指令，别急。

要理解这两条指令，就涉及到实模式和保护模式的第一个区别了。我们现在还处于实模式下，这个模式的 CPU 计算物理地址的方式还记得么？不记得的话看一下 第一回 最开始的两行代码

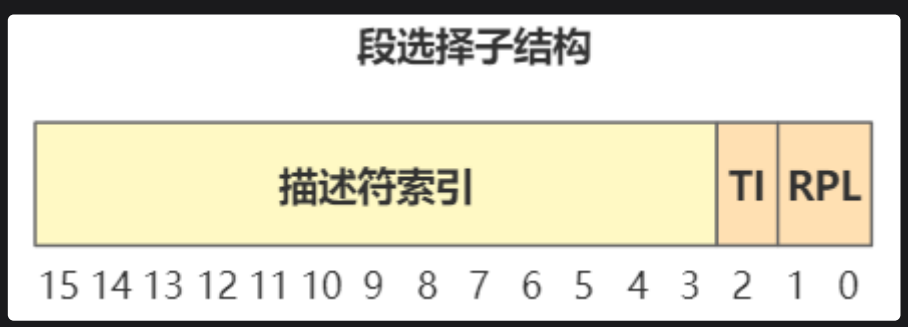
就是段基址左移四位，再加上偏移地址。比如：



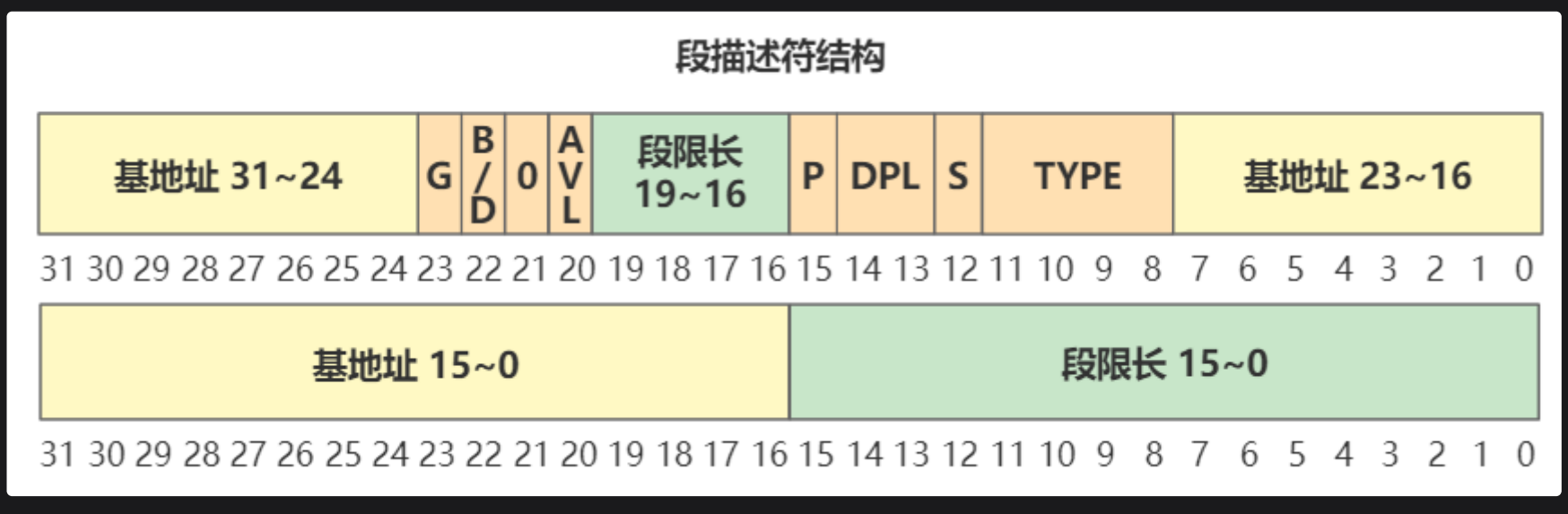
实模式下物理地址的转换（仅段机制）

是不是觉得很别扭，那更别扭的地方就要来了。当 CPU 切换到**保护模式**后，同样的代码，内存地址的计算方式还不一样，你说气不气人？

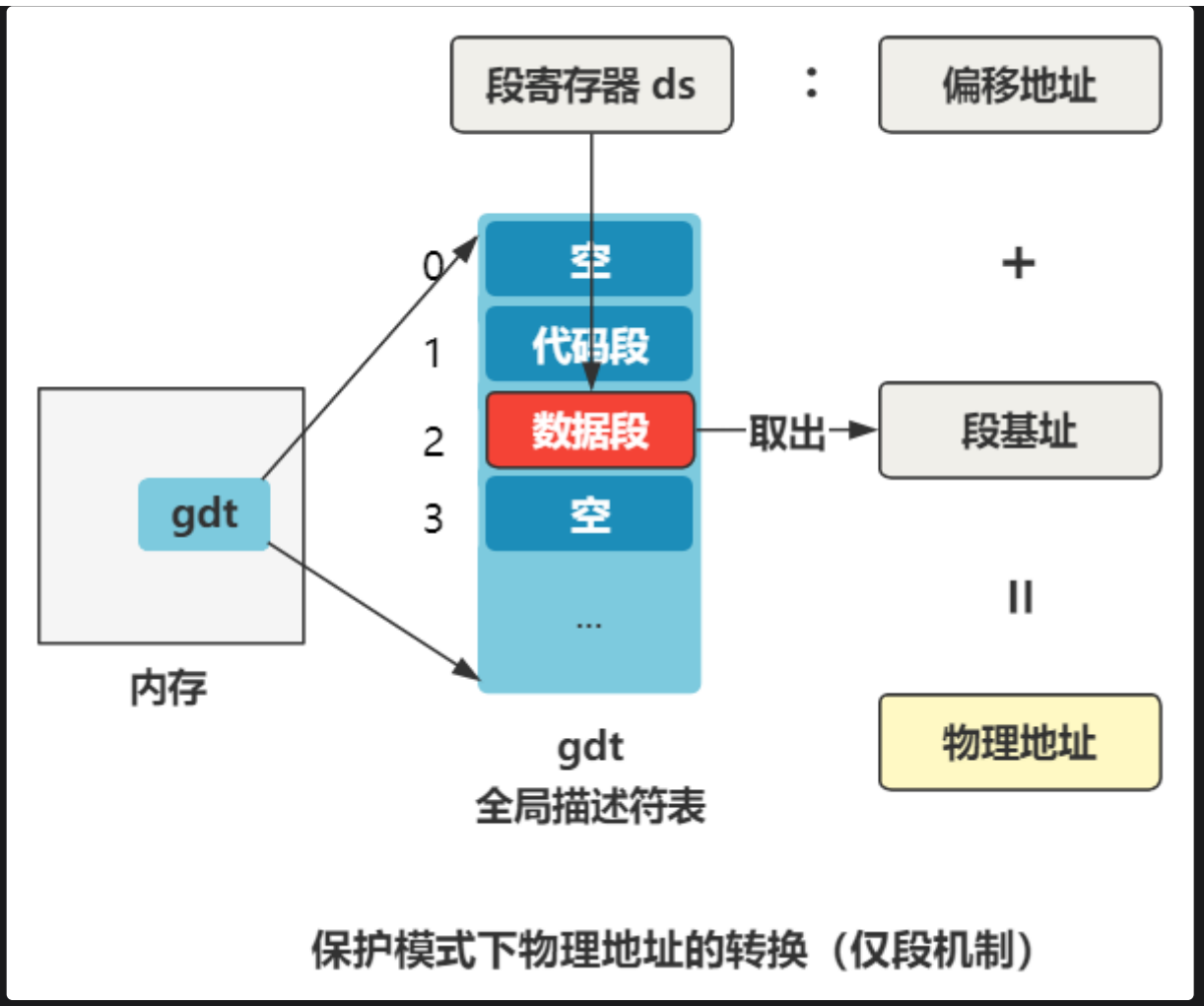
变成啥样了呢？刚刚那个 ds 寄存器里存储的值，在实模式下叫做**段基址**，在保护模式下叫**段选择子**。段选择子里存储着**段描述符**的索引。



通过段描述符索引，可以从**全局描述符表 gdt** 中找到一个段描述符，段描述符里存储着段基址。



段基址取出来，再和偏移地址相加，就得到了物理地址，整个过程如下。



保护模式下物理地址的转换（仅段机制）

你就说烦不烦吧？同样一段代码，实模式下和保护模式下的结果还不同，但没办法，x86 的历史包袱我们不得不考虑，谁让我们没其他 CPU 可选呢。

总结一下就是，**段寄存器（比如 ds、ss、cs）里存储的是段选择子，段选择子去全局描述符表中寻找段描述符，从中取出段基址。**

好了，那问题自然就出来了，**全局描述符表（gdt）**长什么样？它在哪？怎么让 CPU 知道它在哪？

长什么样先别管，一定又是一个令人头疼的数据结构，先说说它在哪？在内存中呗，那么怎么告诉 CPU 全局描述符表（gdt）在内存中的什么位置呢？答案是由操作系统把这个位置信息存储在一个叫 **gdtr** 的寄存器中。

怎么存呢？就是刚刚那条指令。

```
lgdt    gdt_48
```

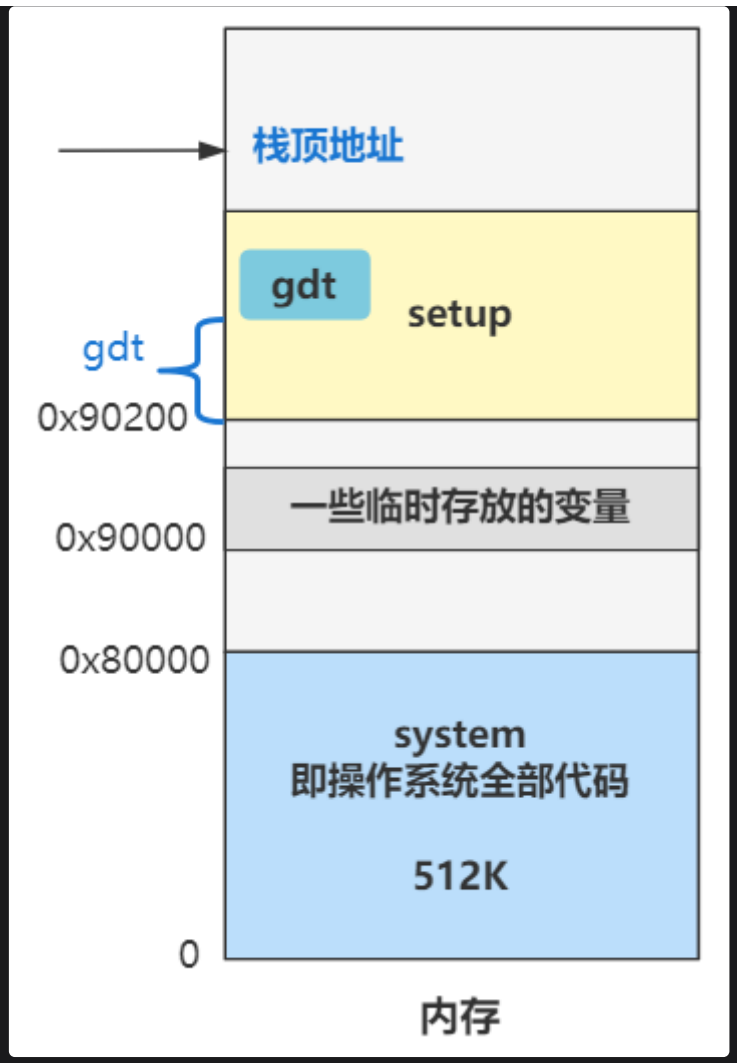
其中 **lgdt** 就表示把**后面的值（gdt_48）**放在 **gdtr** 寄存器中，gdt_48 标签，我们看看它长什么样。

```
gdt_48:    .word    0x800       ; gdt limit=2048, 256 GDT entries    .word    512+gdt,0x9 ; gdt base = 0X9xxxx
```

可以看到这个标签位置处表示一个 48 位的数据，其中高 32 位存储着的正是全局描述符表 gdt 的内存地址

0x90200 + gdt

gdt 是个标签，表示在本文件内的偏移量，而本文件是 **setup.s**，编译后是放在 **0x90200** 这个内存地址的，还记得吧？所以要加上 0x90200 这个值。

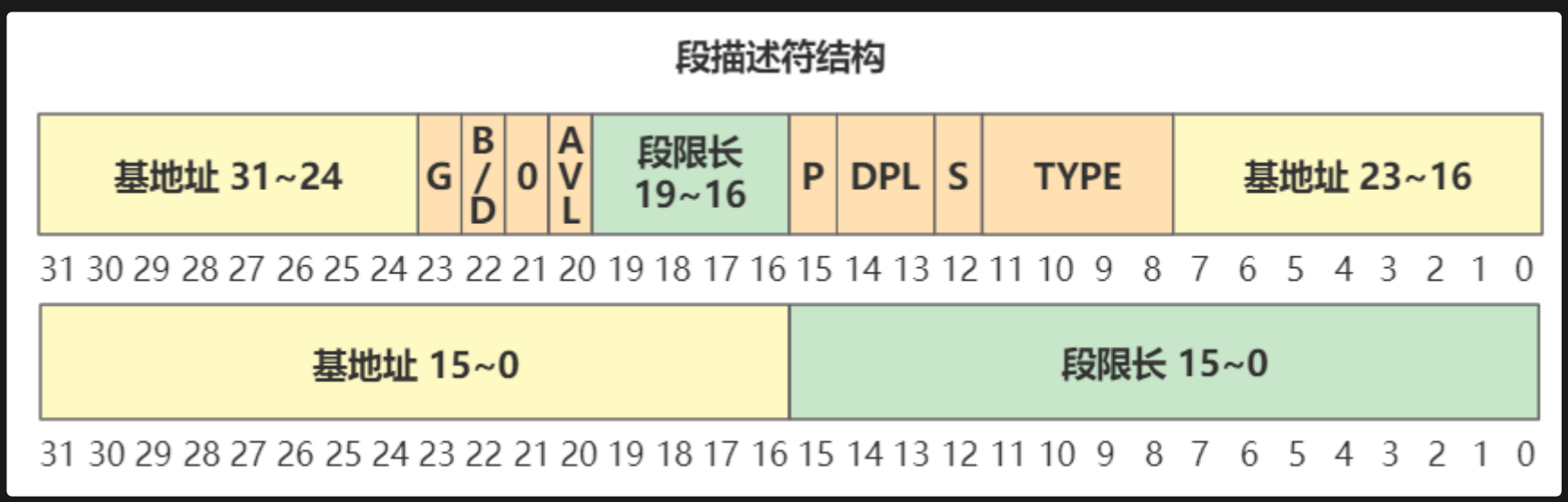


那 gdt 这个标签处，就是全局描述符表在内存中的真正数据了。

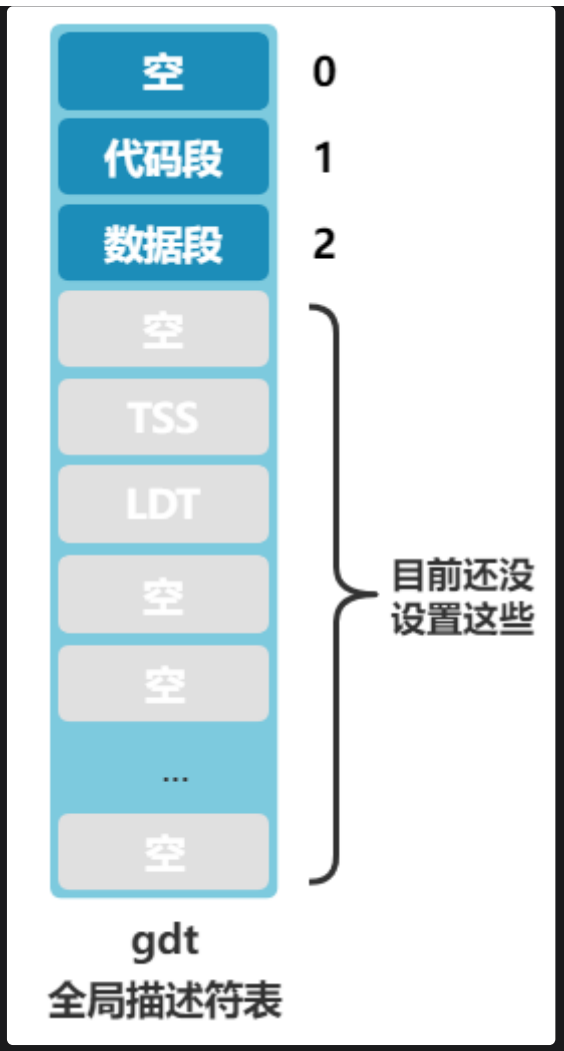
```
gdt:    .word    0,0,0,0    ; dummy    .word    0x07FF    ; 8Mb - limit=2047 (20484096=8Mb)    .word    0x0000    ; base address=0    .word    0x9A00    ; code read/exec    .word    0x00C0    ; granularity=4096, 386    .word    0x07FF    ; 8Mb - limit=2047 (20484096=8Mb)    .word    0x0000    ; base address=0    .word    0x9200    ; data read/write    .word    0x00C0    ; granularity=4096, 386
```

具体细节不用关心，跟我看重点。

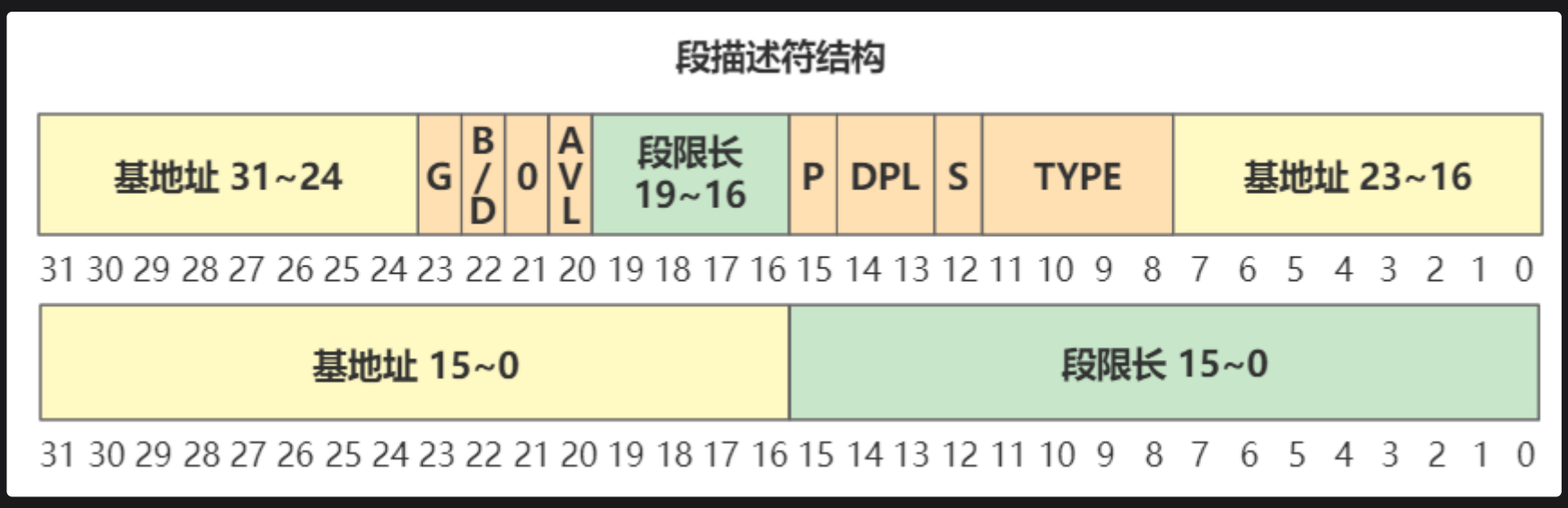
根据刚刚的段描述符格式。



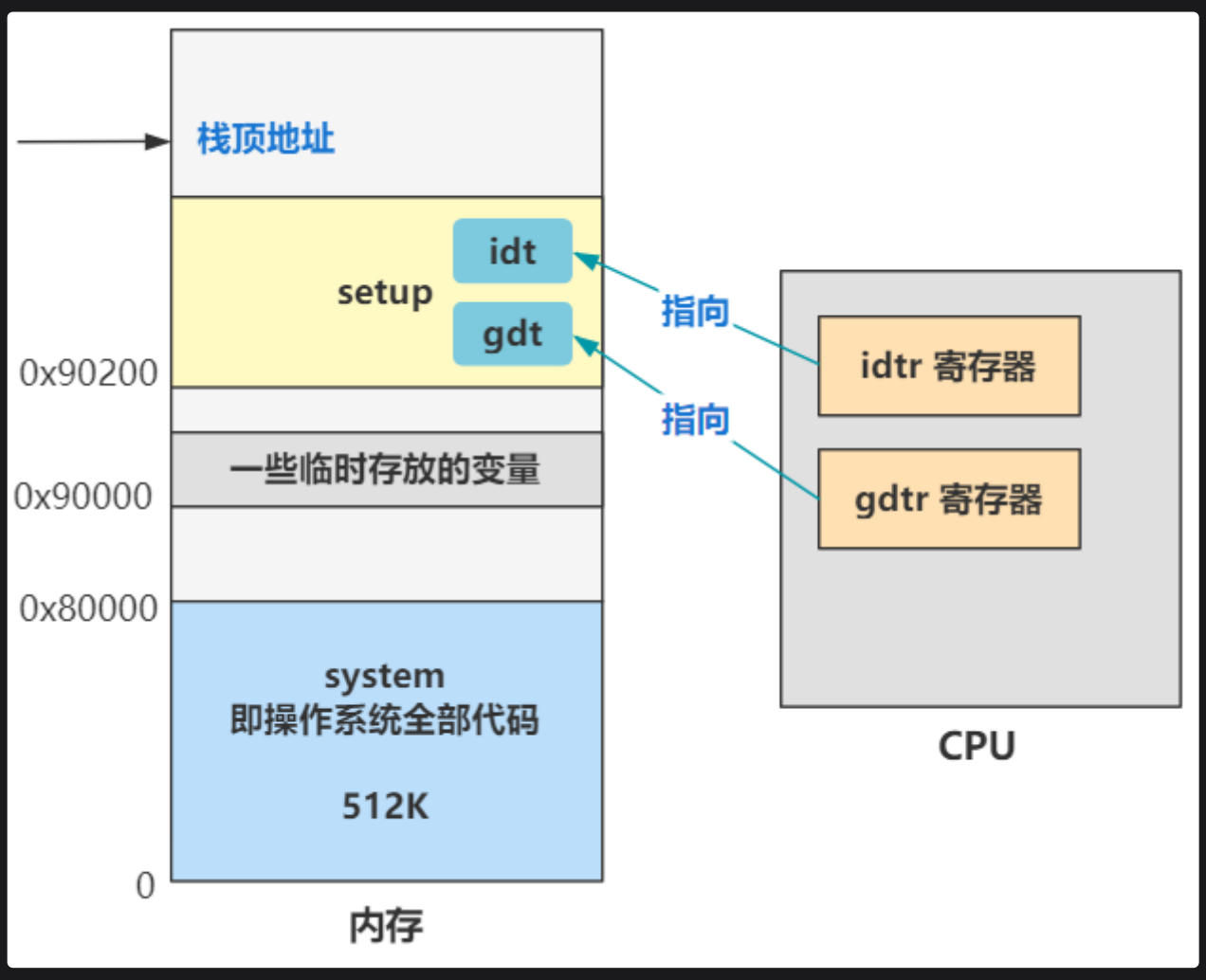
可以看出目前全局描述符表有三个段描述符，第一个为**空**，第二个是**代码段描述符（type=code）**，第三个是**数据段描述符（type=data）**，第二个和第三个段描述符的段基址都是 0，也就是之后在逻辑地址转换物理地址的时候，通过段选择子查找到无论是代码段还是数据段，取出的段基址都是 0，那么物理地址将直接等于程序员给出的逻辑地址（准确说是逻辑地址中的偏移地址）。先记住这点就好。



具体段描述符的细节还有很多，就不展开了，比如这里的高 22 位就表示它是代码段还是数据段。



接下来我们看看目前的内存布局，还是别管比例。



这里我把 **idtr** 寄存器也画出来了，这个是**中断描述符表**，其原理和全局描述符表一样。全局描述符表是让段选择子去里面寻找段描述符用的，而中断描述符表是用来在发生中断时，CPU 拿着中断号去中断描述符表中寻找中断处理程序的地址，找到后就跳到相应的中断程序中去执行，具体我们后面遇到了再说。

好了，今天我们就讲，操作系统设置了个**全局描述符表 gdt**，为后面切换到**保护模式**后，能去那里寻找到段描述符，然后拼凑成最终的物理地址，就这个作用。当然，还有很多段描述符，作用不仅仅是转换成最终的物理地址，不过这是后话了。

这仅仅是进入保护模式前准备工作的其中一个，后面的路还长着呢。欲知后事如何，且听下回分解。

------- 本回扩展资料 -------

保护模式下逻辑地址到线性地址（不开启分页时就是物理地址）的转化，看 Intel 手册：

Volume 3 Chapter 3.4 Logical And Linear Addresses

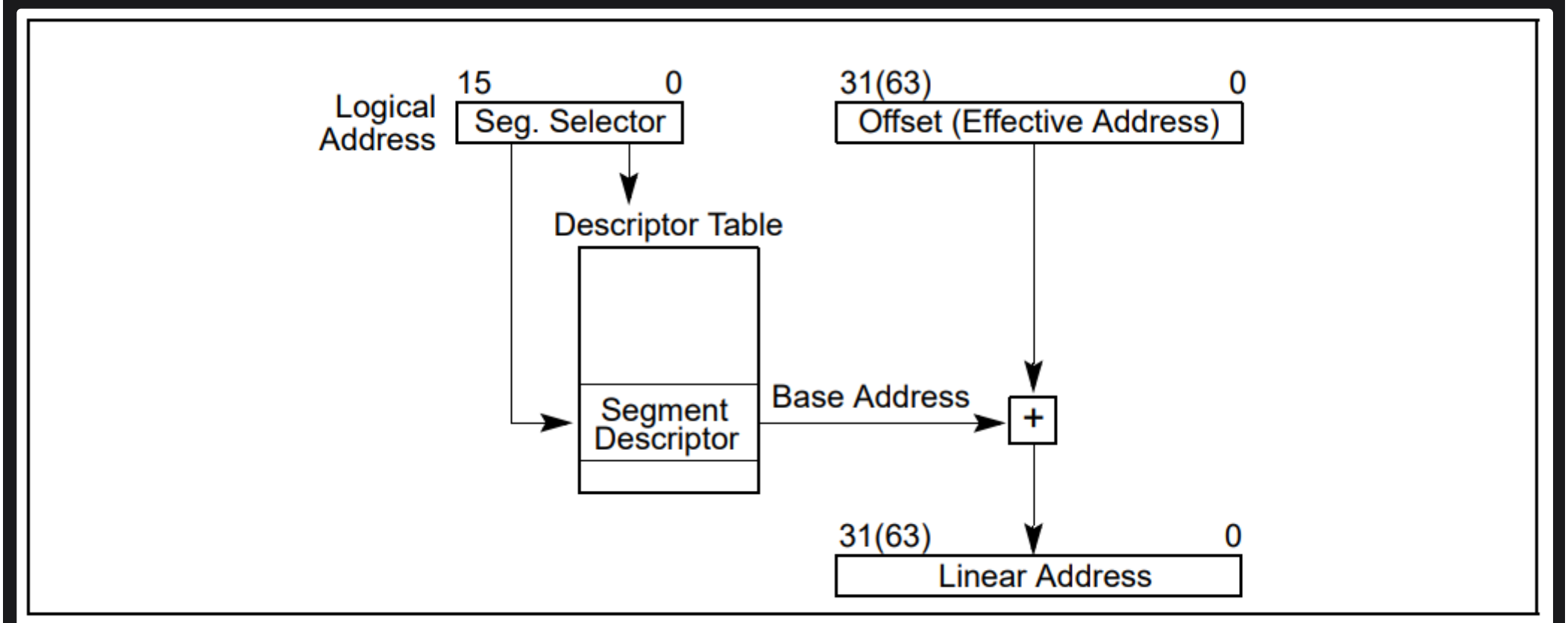


Figure 3-5. Logical Address to Linear Address Translation

段描述符结构和详细说明，看 Intel 手册：

Volume 3 Chapter 3.4.5 Segment Descriptors

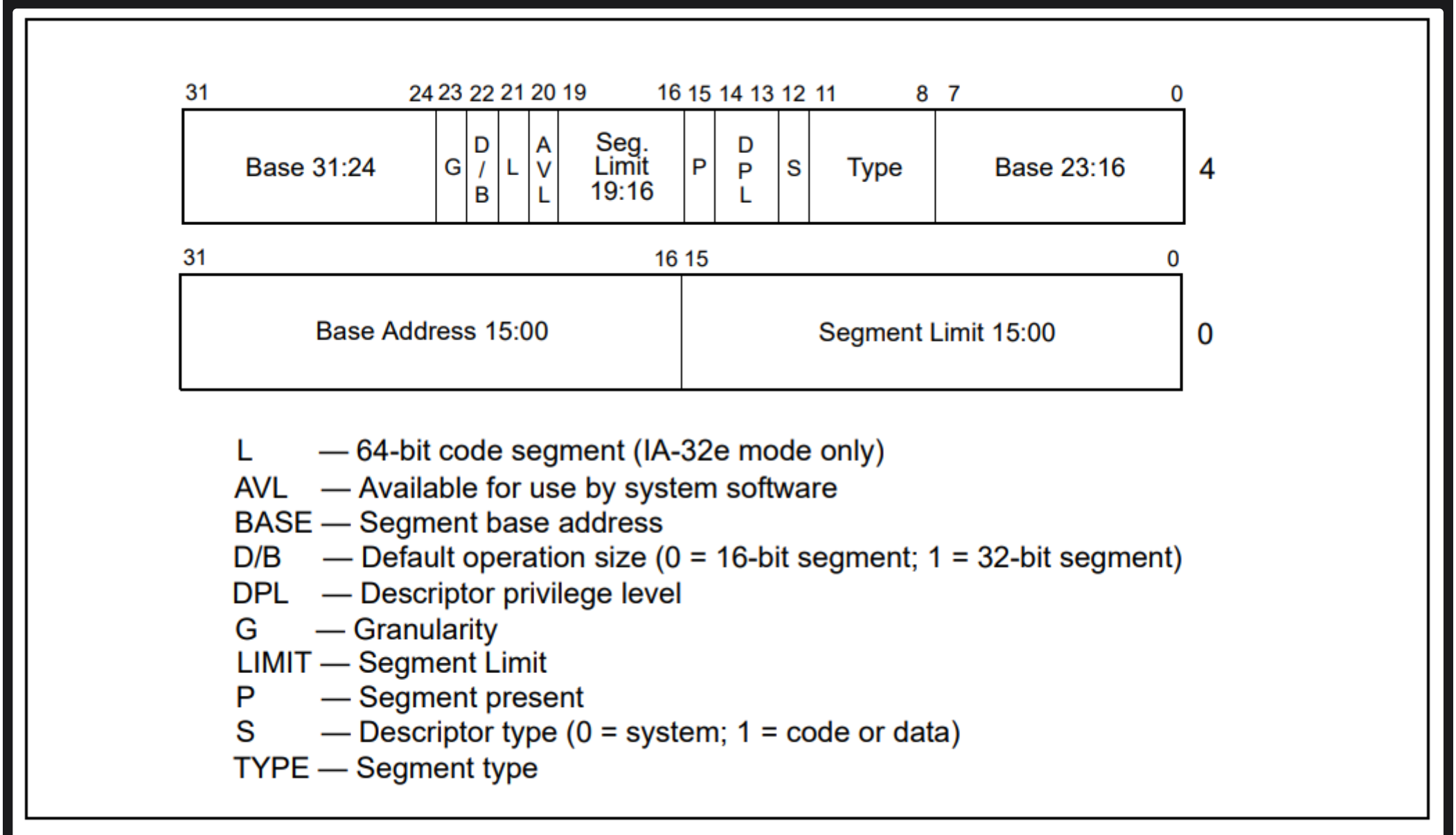


Figure 3-8. Segment Descriptor

比如文中说的数据段与代码段的划分，其实还有更细分的权限控制。

Table 3-1. Code- and Data-Segment Types

Type Field					Descriptor Type	Description
Decimal	11	10 E	9 W	8 A		
0	0	0	0	0	Data	Read-Only
1	0	0	0	1	Data	Read-Only, accessed
2	0	0	1	0	Data	Read/Write
3	0	0	1	1	Data	Read/Write, accessed
4	0	1	0	0	Data	Read-Only, expand-down
5	0	1	0	1	Data	Read-Only, expand-down, accessed
6	0	1	1	0	Data	Read/Write, expand-down
7	0	1	1	1	Data	Read/Write, expand-down, accessed
		C	R	A		
8	1	0	0	0	Code	Execute-Only
9	1	0	0	1	Code	Execute-Only, accessed
10	1	0	1	0	Code	Execute/Read
11	1	0	1	1	Code	Execute/Read, accessed
12	1	1	0	0	Code	Execute-Only, conforming
13	1	1	0	1	Code	Execute-Only, conforming, accessed
14	1	1	1	0	Code	Execute/Read-Only, conforming
15	1	1	1	1	Code	Execute/Read-Only, conforming, accessed