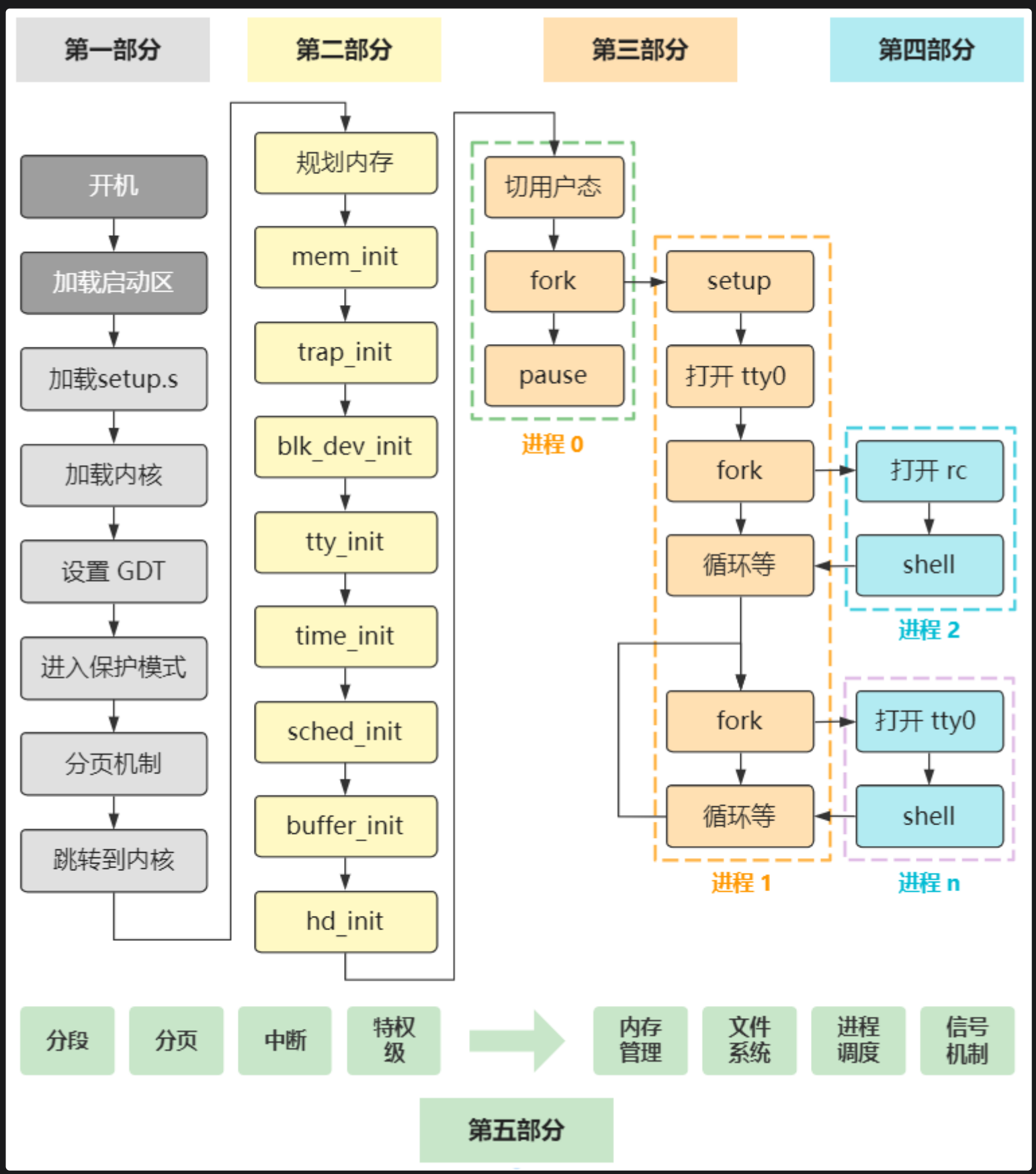
第四回

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

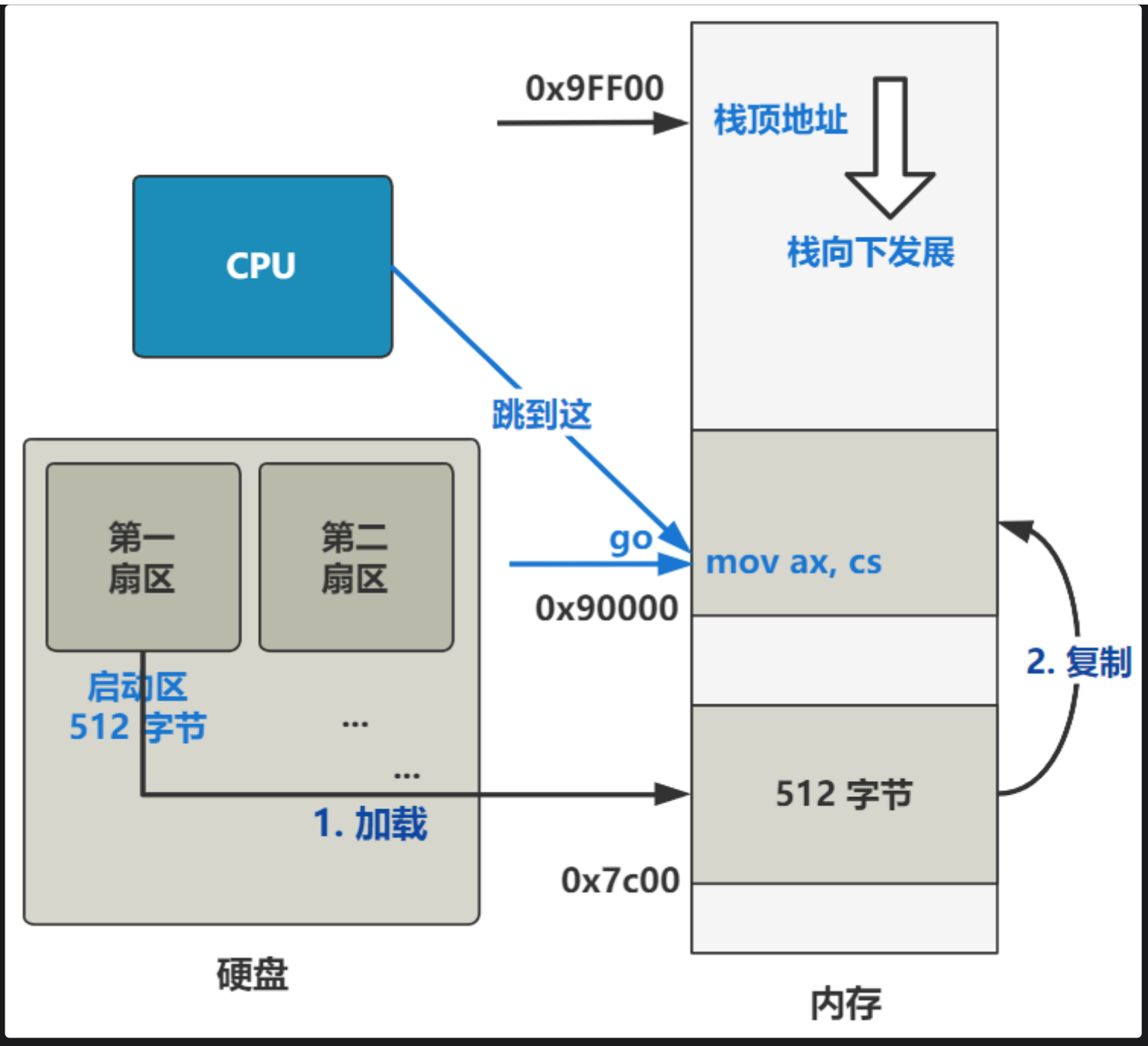
第三回 做好最最基础的准备工作

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

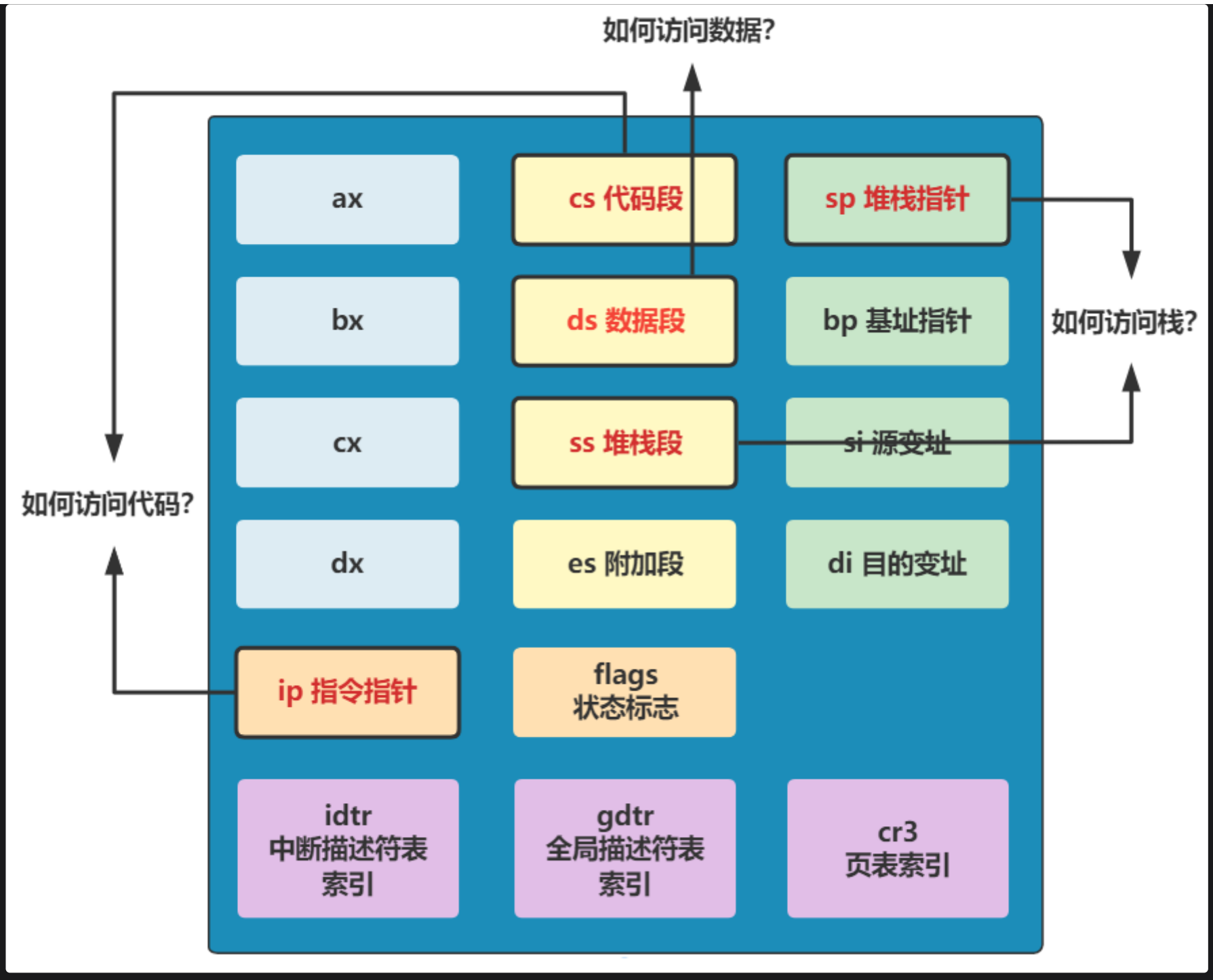
------- **正文开始** -------

书接上回，上回书咱们说到，操作系统的一些最最最最基础的准备工作，已经准备好了。



如这张图所示，此时操作系统短短几行代码，将**数据段寄存器 ds** 和**代码段寄存器 cs** 设置为了 **0x9000**，方便代码的跳转与数据的访问。并且，将**栈顶地址 ss:sp** 设置在了离代码的位置 0x90000 足够遥远的 **0x9FF00**，保证栈向下发展不会轻易撞见代码的位置。

简单说，就是设置了如何访问数据的**数据**段，如何访问代码的**代码**段，以及如何访问栈的**栈顶**指针，也即初步做了一次**内存规划**，从 CPU 的角度看，访问内存，就这么三块地方而已。



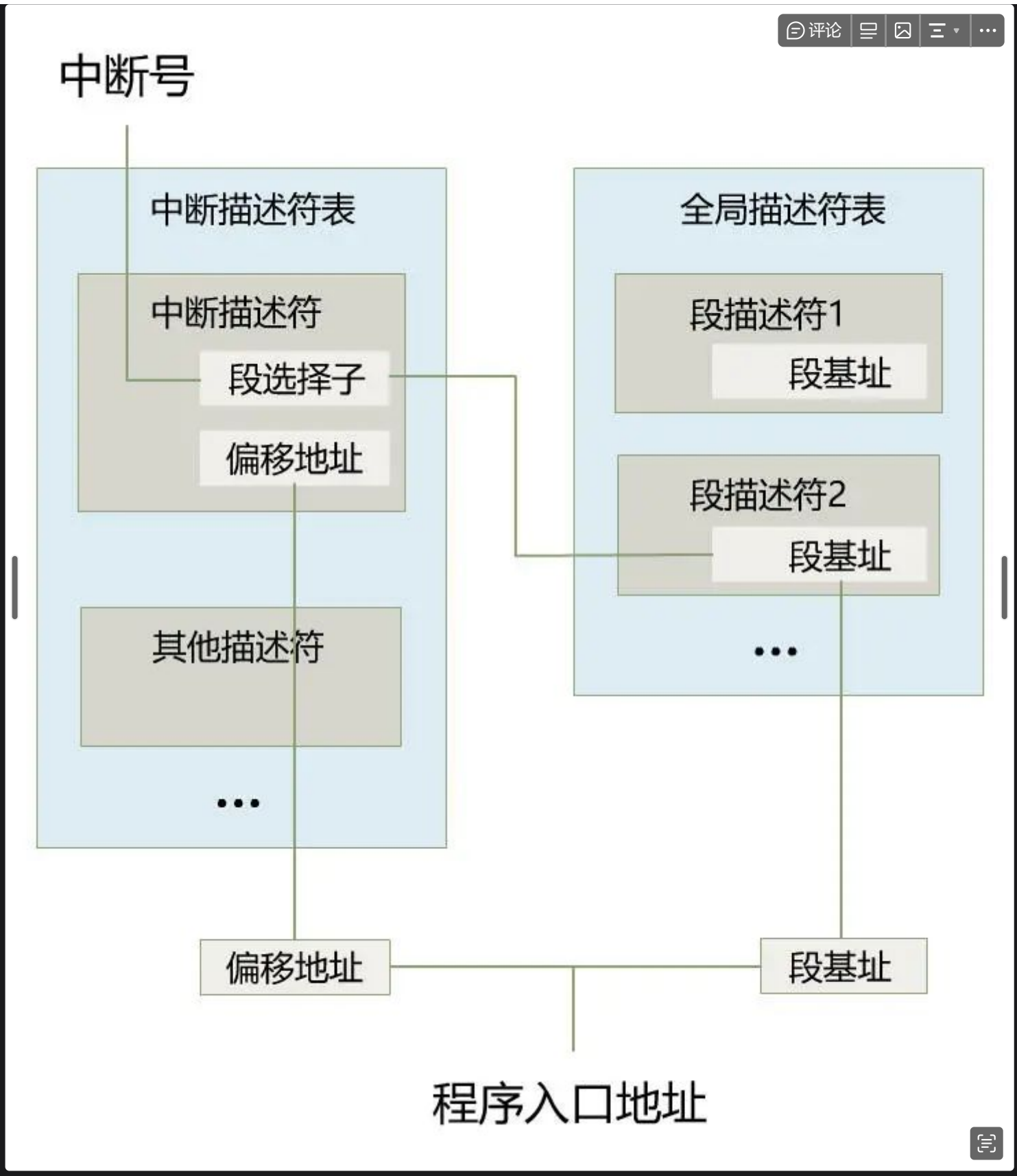
做好这些基础工作后，接下来就又该新的一翻折腾了，我们接着往下看。

```
load_setup:    mov dx,#0x0000      ; drive 0, head 0    mov cx,#0x0002      ; sector 2, track 0    mov bx,#0x0200      ; address = 512, in
0x9000    mov ax,#0x0200+4    ; service 2, nr of sectors    int 0x13            ; read it    jnc ok_load_setup       ; ok - continue    mov
dx,#0x0000    mov ax,#0x0000      ; reset the diskette    int 0x13    jmp load_setupok_load_setup:    ...
```

这里有两个 **int 指令**我们还没见过。

注意这个 int 是汇编指令，可不是高级语言的整型变量哟。int 0x13 表示**发起 0x13 号中断**，这条指令上面给 dx、cx、bx、ax 赋值都是作为这个中断程序的参数。

中断是啥如果你不理解，先不要管，如果你就是放不下，那可以看一眼我之前的文章：认认真真的聊聊中断，里面讲得非常细致。

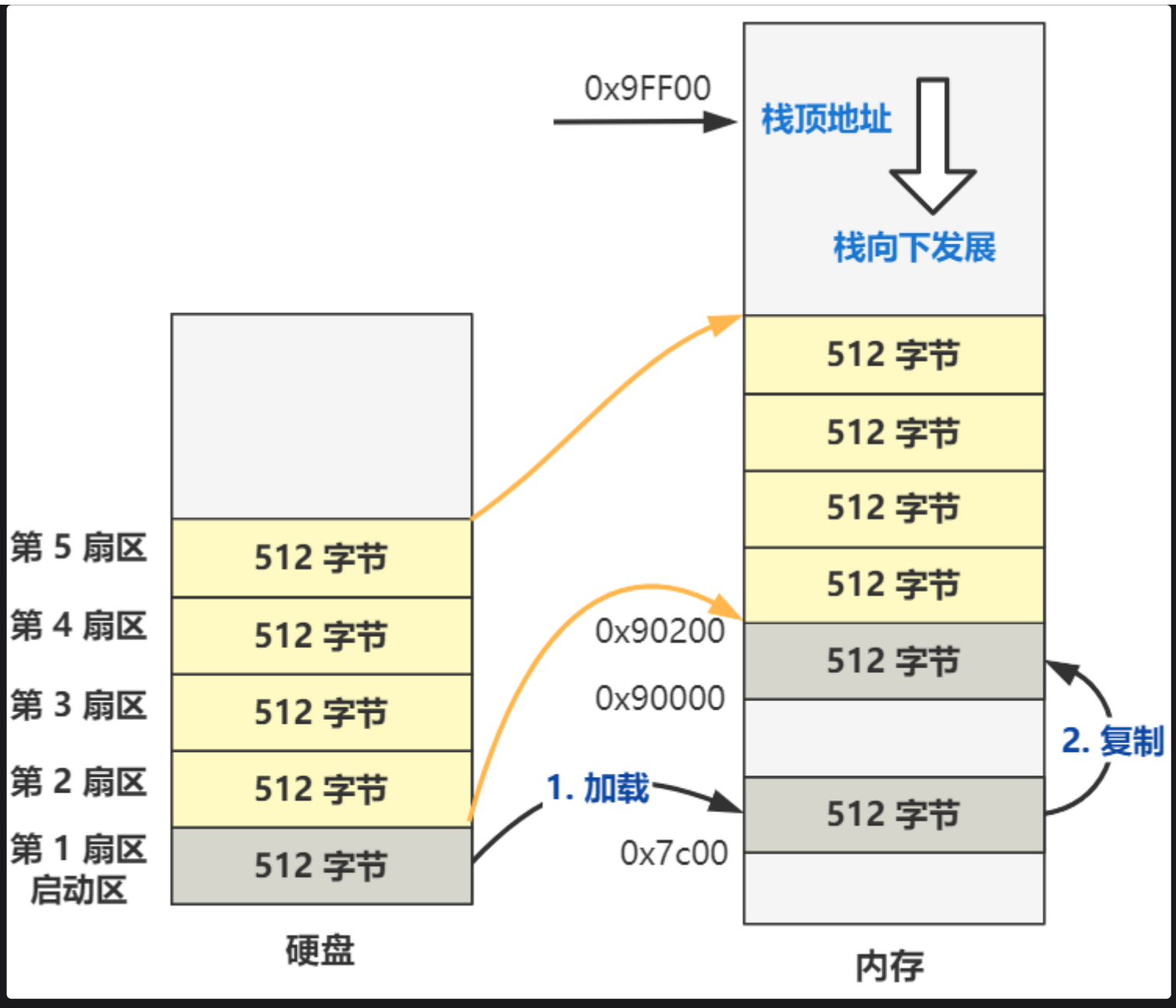


总之这个中断发起后，CPU 会通过这个**中断号**，去寻找对应的**中断处理程序的入口地址**，并**跳转**过去执行，逻辑上就相当于**执行了一个函数**。而 0x13 号中断的处理程序是 BIOS 提前给我们写好的，是**读取磁盘**的相关功能的函数。

之后真正进入操作系统内核后，中断处理程序是需要我们自己去重新写的，这个在后面的章节中，你会不断看到各个模块注册自己相关的中断处理程序，所以不要急。此时为了方便就先用 BIOS 提前给我们写好的程序了。

可见即便是操作系统的源码，有时也需要去调用现成的函数方便自己，并不是造轮子的人就非得完全从头造。

本段代码的注释已经写的很明确了，直接说最终的作用吧，**就是将硬盘的第 2 个扇区开始，把数据加载到内存 0x90200 处，共加载 4 个扇区**，图示其实就是这样。



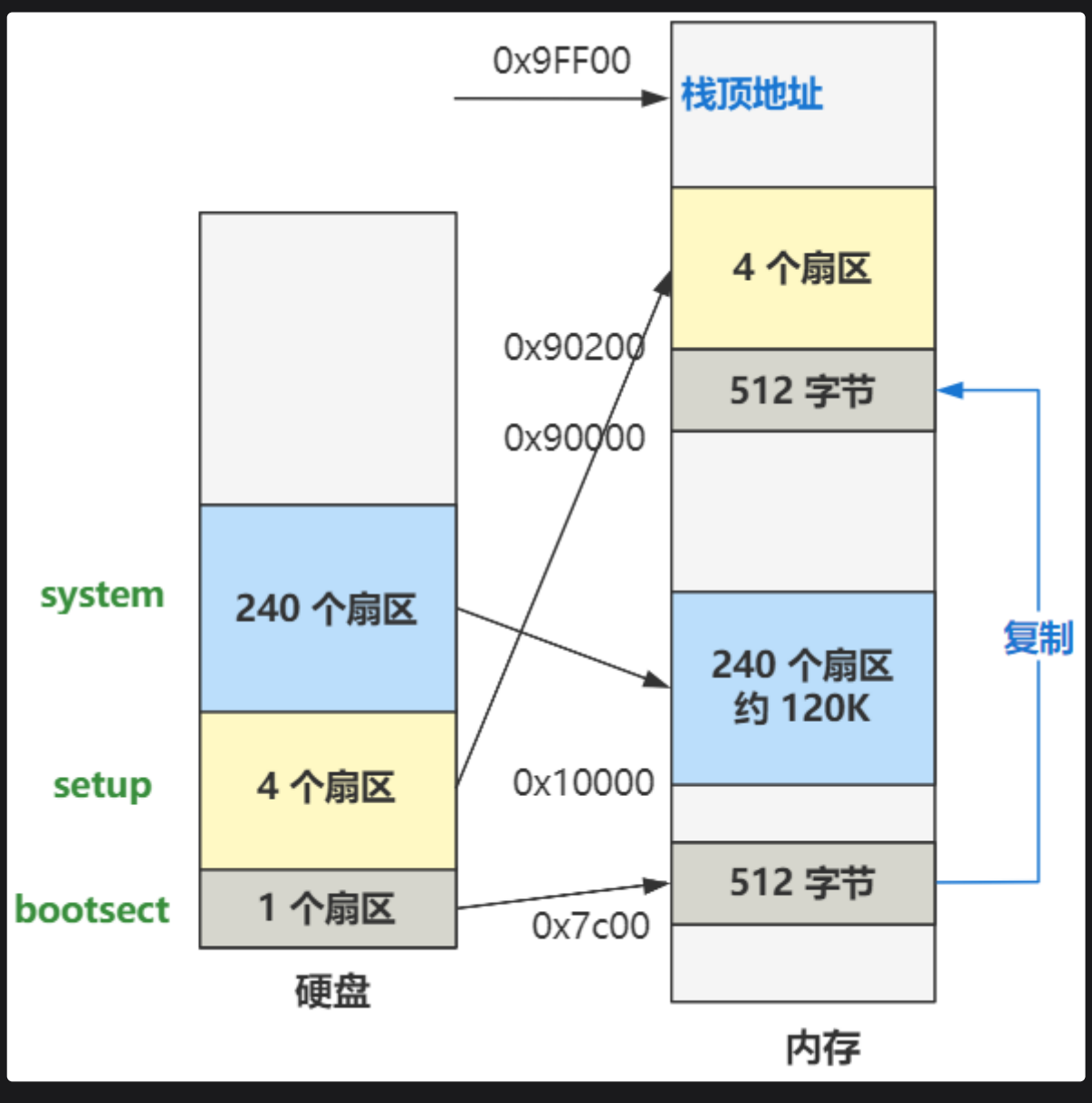
为了图片清晰表达意思，可能比例就不那么严谨了，大家不必纠结。

可以看到，如果复制成功，就跳转到 **ok_load_setup** 这个标签，如果失败，则会不断重复执行这段代码，也就是重试。那我们就别管重试逻辑了，直接看成功后跳转的 ok_load_setup 这个标签后的代码。

```
ok_load_setup:    ...    mov ax,#0x1000    mov es,ax       ; segment of 0x10000    call read_it    ...    jmpi 0,0x9020
```

这段代码省略了很多非主逻辑的代码，比如在屏幕上输出 Loading system ... 这个字符串以防止用户等烦了。

剩下的主要代码就都写在这里了，就这么几行，其作用是**把从硬盘第 6 个扇区开始往后的 240 个扇区，加载到内存 0x10000 处**，和之前的从硬盘捣腾到内存是一个道理。



至此，整个操作系统的**全部代码**，就已经全部从硬盘中，被搬迁到内存来了。

然后又通过一个熟悉的段间跳转指令 jmpi 0,0x9020，跳转到 **0x90200** 处，就是硬盘第二个扇区开始处的内容。

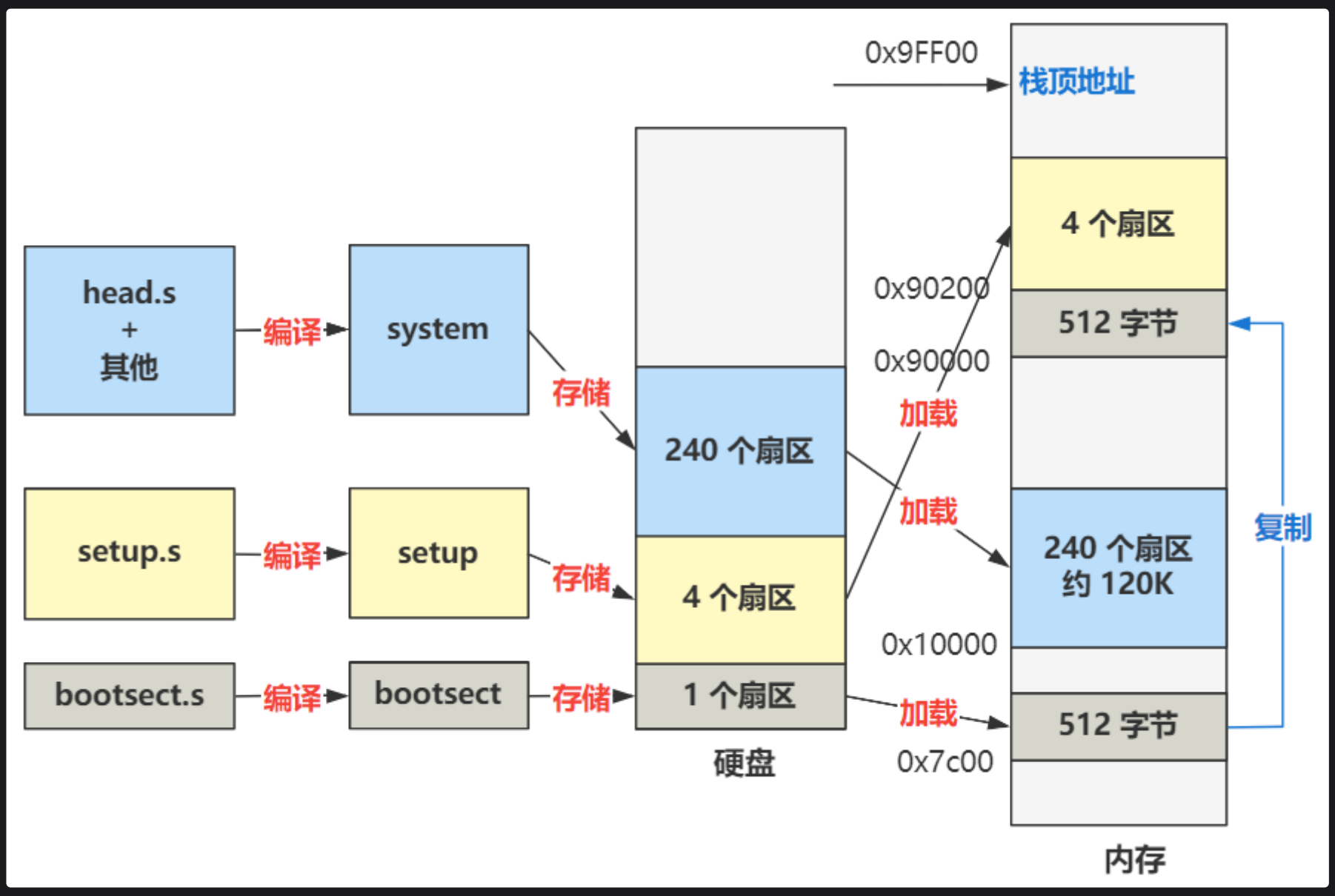
那这里的内容是什么呢？先不急，我们借这个机会把整个操作系统的编译过程说下。整个编译过程，就是通过 **Makefile** 和 **build.c** 配合完成的，最终会：

1. 把 bootsect.s 编译成 bootsect 放在硬盘的 1 扇区。

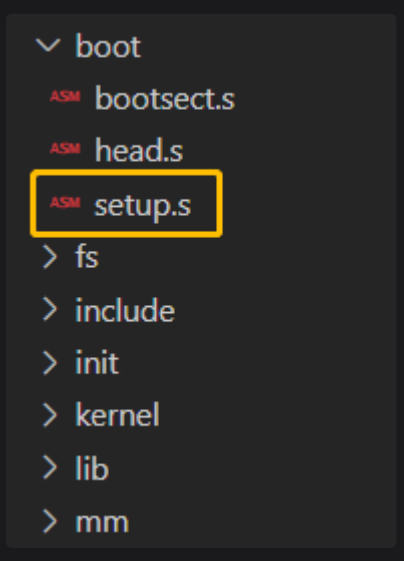
2. 把 setup.s 编译成 setup 放在硬盘的 2~5 扇区。

3. 把剩下的全部代码（head.s 作为开头）编译成 system 放在硬盘的随后 240 个扇区。

所以整个路径就是这样的。



所以，我们即将跳转到的内存中的 **0x90200** 处的代码，就是从硬盘第二个扇区开始处加载到内存的。第二个扇区的最开始处，那也就是 setup.s 文件的第一行代码咯。



那这个代码是什么呢？我们后面再说，不过先打开 setup.s 这个文件看看吧。

```
start:  mov ax,#0x9000  ; this is done in bootsect already, but...  mov ds,ax  mov ah,#0x03  ; read cursor pos  xor bh,bh  int 0x10  ; save it in known place, con_init fetches  mov [0],dx  ; it from 0x90000.  ...
```

好了，到目前为止，你是不是觉得，我去，这前面编译放在硬盘的位置，和后面代码写死的跳转地址，竟然如此地强耦合？那万一整错了咋办。

是啊，就是这样，你以为呢？**在操作系统刚刚开始建立的时候，那是完全自己安排前前后后的关系**，一个字节都不能偏，就是这么强耦合，需要小心翼翼，需要大脑时刻保持清醒，规划好自己写的代码被编译并存储在硬盘的哪个位置，而随后又会被加载到内存的哪个位置，不能错乱。

但这也是很有好处的，那就是在这个阶段，你完完全全知道每一步跳转，每一步数据访问都是怎么设计和规划的，不存在黑盒。

不像我们在写高级语言的时候，完全不知道是怎么底层帮我们做了多少工作。虽然这解脱了程序员关心底层细节的烦恼，但在遇到问题或者想知道原理的时候，就显得很讨厌了。所以珍惜这个阶段吧！

而且，你在上层之所以能那么随心所欲，很多底层细节完全不用考虑，很省心，正是因为像今天这样以及之后每一章的各种底层代码小心翼翼的做了很多铺垫。

好了，本文的内容就结束了。这也标志着我们走完了**第一个操作系统源码**文件 **bootsect.s**，开始向下一个文件 **setup.s** 进发了！

后面的世界越来越精彩，欲知后事如何，且听下回分解。

------- **多说两句** -------

先给大家留个课后作业，文中不是提到了 BIOS 提供了很多中断函数方便操作系统刚启动的时候调用么？这些中断都是什么？大家负责去找一份**一手资料**（注意是一手资料，不要网上整理的二手博客），并上传到我的 **GitHub** 上（文末阅读原文就是）。

不知不觉已经第四回了，刚刚才把 bootsect.s 这个汇编文件讲完，它所做的事情无非就是把硬盘中的数据复制到内存，然后挪来挪去的，并且根据放置在内存中的位置，设置了各种段基址寄存器的值。

在解答读者疑问时，我发现有**两种特别极端的人**。

一种是把这几讲的内容想得极其复杂，什么代码段和数据段一样会不会相互影响，这些地址是虚拟地址还是物理地址，为什么要不断从内存一个地方挪动到另一个地方，BIOS 是怎么映射到内存的，硬盘中 512 字节是数据还是代码，等等。

这样的人，我建议把你所知道的一切先忘掉，就把这几讲所说的东西理解清楚，因为它就是很简单，不涉及那么多乱七八糟的知识。你所产生的那些疑问，在这个阶段根本就不存在这些问题，现在内存就在这，你想怎么玩就怎么玩，无非就是访问数据，执行代码，自己能安排明白即可。

还有一种人，就是把这几讲的内容想得特别简单，这几讲确实就是复制硬盘数据和内存数据为主，但这就是操作系统呀！它就是不断通过这样简简单单的动作，把自己一点一点搞复杂的呢。

首先它靠 BIOS 做了第一步加载动作，然后就可以用加载的这 512 字节，去加载更多在硬盘中的代码和数据，那整个过程就自己把自己加载完整了，你不觉得这个过程也很伟大和奇妙么？会不会有解答了你之前一直困惑的什么东西呢？

多想想，多看看每回的扩展部分和延伸部分。