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。

在上一篇文章中Linux从头学05-系统启动过程中的几个神秘地址，你知道是什么意思吗？，我们以几个重要的内存地址为线索，介绍了 x86 系统在上电开机之后：

1. CPU 如何执行第一条指令;
2. BIOS 中的程序如何被执行;
3. 操作系统的引导代码(bootloader) 被读取到物理内存中被执行;

下一个环节，就应该是引导程序(`bootloader`)把操作系统程序，读取到内存中，然后跳入到操作系统的第一条指令处开始执行。

这篇文章，我们继续以 `8086` 这个简单的处理器为原型，把程序的加载过程描述一下。其中的重点部分就是代码重定位，我们用画图的方式，尽我所能，把程序加载、地址重定位的计算过程描述清楚。

PS: 文中所说的程序、操作系统文件，都是指同一个东西。

程序的结构

为了便于下面的理解，我们有必要把待加载的操作系统程序的文件结构先介绍一下。

当然了，这里介绍的文件结构，是一个非常简化版本的操作系统程序，本质上与我们平常所写的应用程序没有什么差别，因此我们也可以把它看做一个普通的程序文件。

操作系统程序静静的躺在硬盘中，等待 bootloader 来读取，此时 bootloader 可以看做一个加载器。

它俩毕竟是属于两个不同的东西，为了让 bootloader 知道程序的长度，需要某种“协议”来进行沟通，这个“协议”就是程序文件的头信息(Header)。

也就是说，在程序的开头部分，会详细的介绍自己，包括：程序的总长度是多少字节，一共有多少个段，入口地址在什么位置等等。

还记得之前介绍过的 Linux 系统中使用的 ELF 文件格式吗？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

那篇文章把一个典型的 Linux ELF 格式的可执行文件彻底拆解了一遍，可以看到，在 ELF 文件的头部信息中，详细描述了文件中每一部分内容。

其实 Windows 中的程序格式(PE 格式)也是类似的，它与 ELF 格式来源于同一个祖宗。

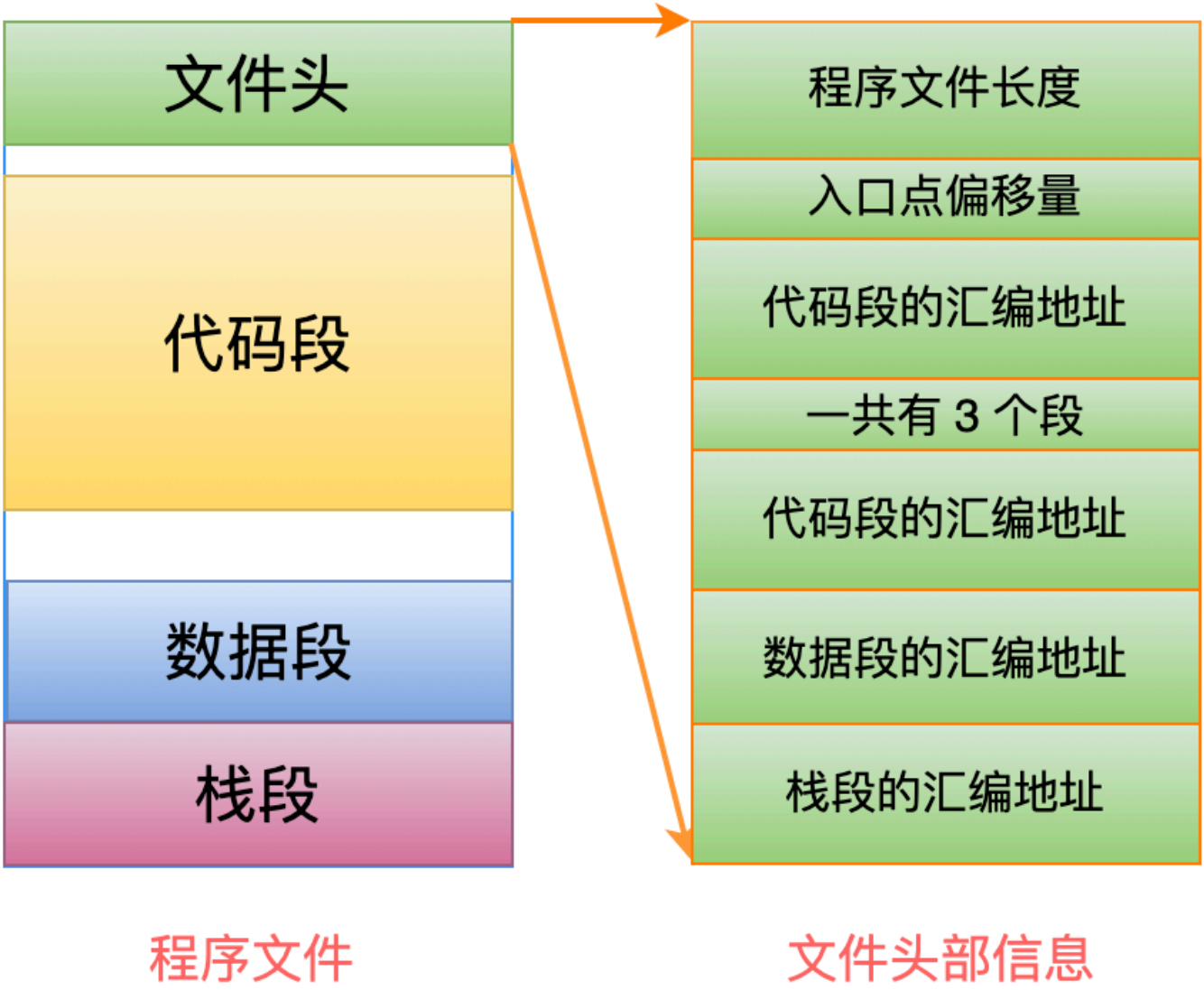
1. 程序头(Header)的描述信息

为了便于描述，我们假设程序中包括 3 个段：代码段，数据段和栈段，再加上程序头部信息，一共是 4 个组成部分。如下所示：

为什么中间留有白色的空白?

因为每一个段并不是紧挨着排列的，为了段地址能够内存对齐(16个字节对齐)，段与段之间可能会空余一段空间，这些空间里的数据都是无效的。

刚才说了，为了能够让加载器(bootloader)尽可能的了解自己，程序文件会在自己的 Header 部分，详细的描述自己的信息:



有了这样的描述信息，bootloader 就能够知道一共要读取多少个字节的程序文件，跳转到哪个位置才能让操作系统的指令开始执行。

2. 关于汇编地址

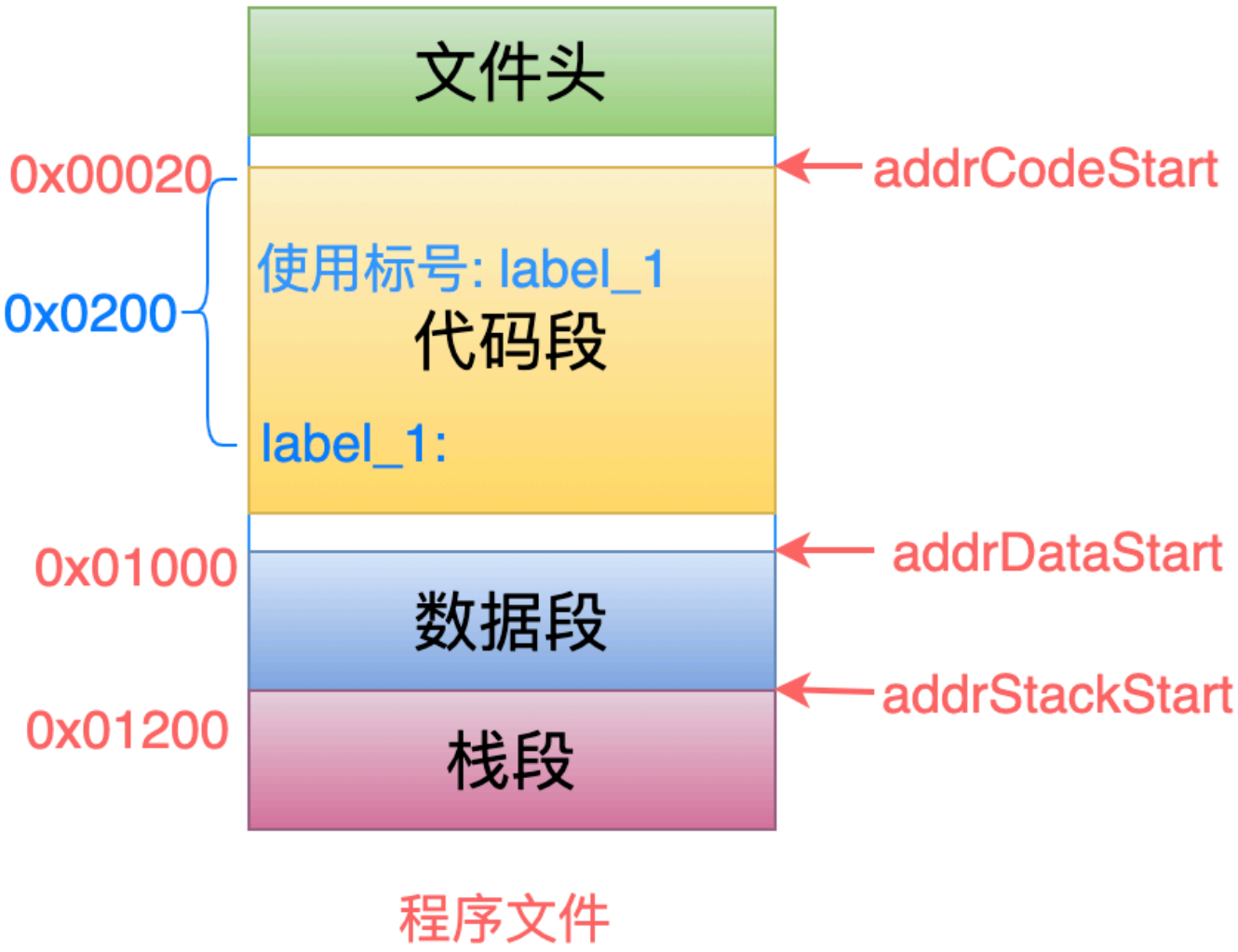
在程序的头信息中，可以看到汇编地址和偏移量这样的信息。

编译器在编译源代码的时候，它是不知道 bootloader 会把程序加载到内存中的什么位置的。

bootloader 会查看哪个位置有足够的空间，找到一个可用的位置之后，就把操作系统程序读取到这个位置，可以看做是一个动态的过程。

因此，编译器在编译阶段用来定位变量、标签等使用的地址，都是相对于当前段的开始地址来计算的。

还是拿刚才的图片来举例：



我们假设 Header 部分是 32 个字节，三个段的开始地址分别是：

> 代码段 addrCodeStart： 0x00020（距离文件的第一个字节是 32 Bytes）；
>
> 数据段 addrDataStart： 0x01000（距离文件的第一个字节是 4K Bytes）；
>
> 栈段 addrStackStart： 0x01200（距离文件的第一个字节是 4K+512 Bytes）；

在代码段中，定义了一个标签 label_1，它距离代码段的开始位置(0x00020)是 512 个字节(0x0200)。

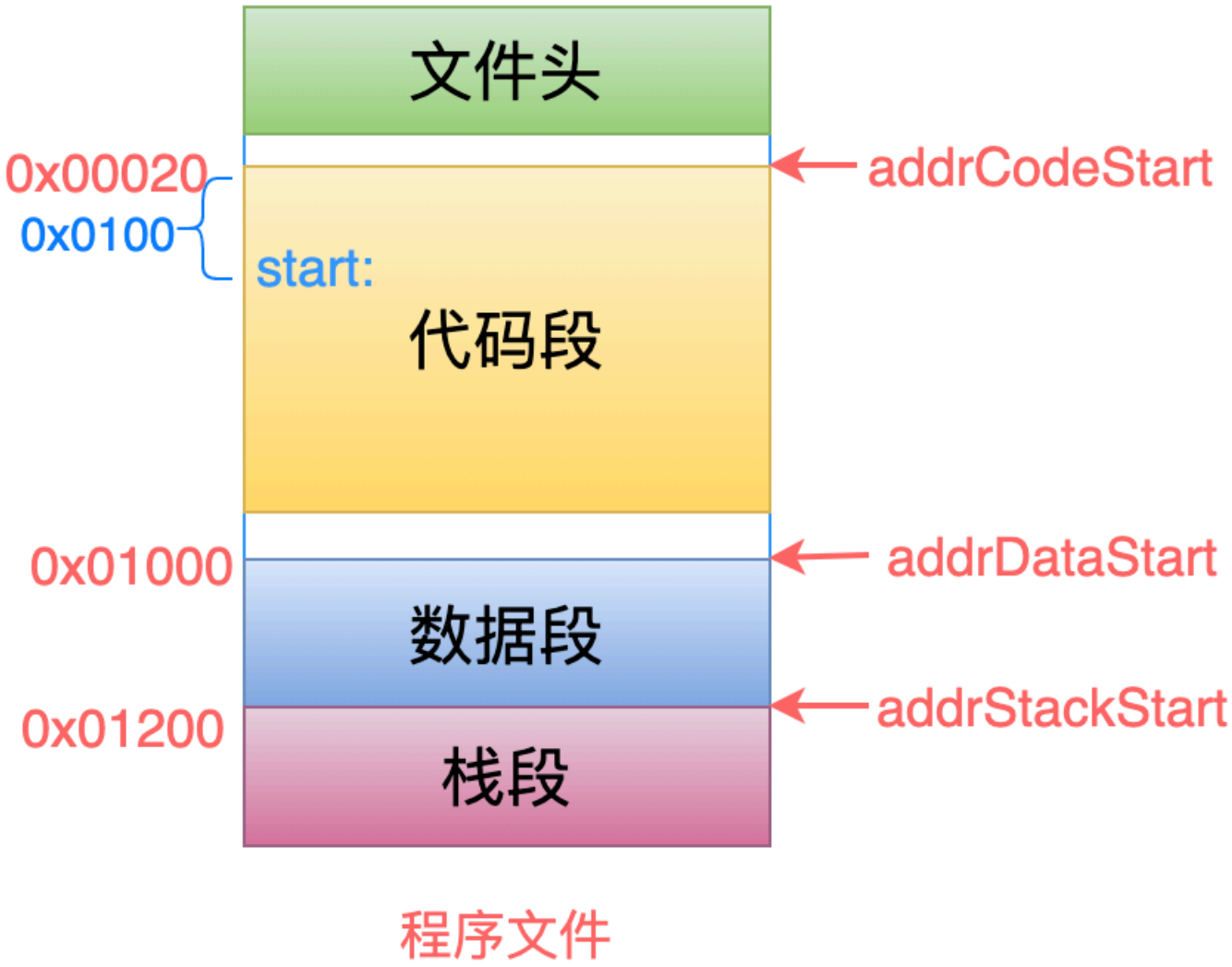
同时，可以算出它距离文件开头的第一个字节就是 512 + 32 = 544 字节，因为代码段的开始地址距离文件头部是 32 个字节。

在 label_1 之前的代码中，会引用到这个标签。

那么在使用的地方，将会填上 0x0200，表示：引用的这个位置是距离代码段开始地址的 512 字节处。

以上的这些地址，指的就是汇编地址。

我们再来拿程序的入口地址偏移量来举例，入口地址是通过 start 标签来定义的：



假设：在代码段中，入口地址标签 `start` 位于代码段开始位置的 `0x0100` 偏移处，也就是距离代码段开始位置的 256 个字节。

那么，在程序的 `Header` 信息中，入口点偏移量的位置就要填写 `0x0100`，这样的话，`bootloader` 把程序读取到内存中之后，就能从这里获取到程序入口点的偏移地址，然后经过一系列的重定位，就可以准确跳转到程序的第一条指令的地方去执行了。

按照刚才假设的地址信息，程序头 `Header` 中的信息就是下面这个样子：

最右侧的蓝色字体，表示每一个项目占用的字节数，一共是 24 个字节。

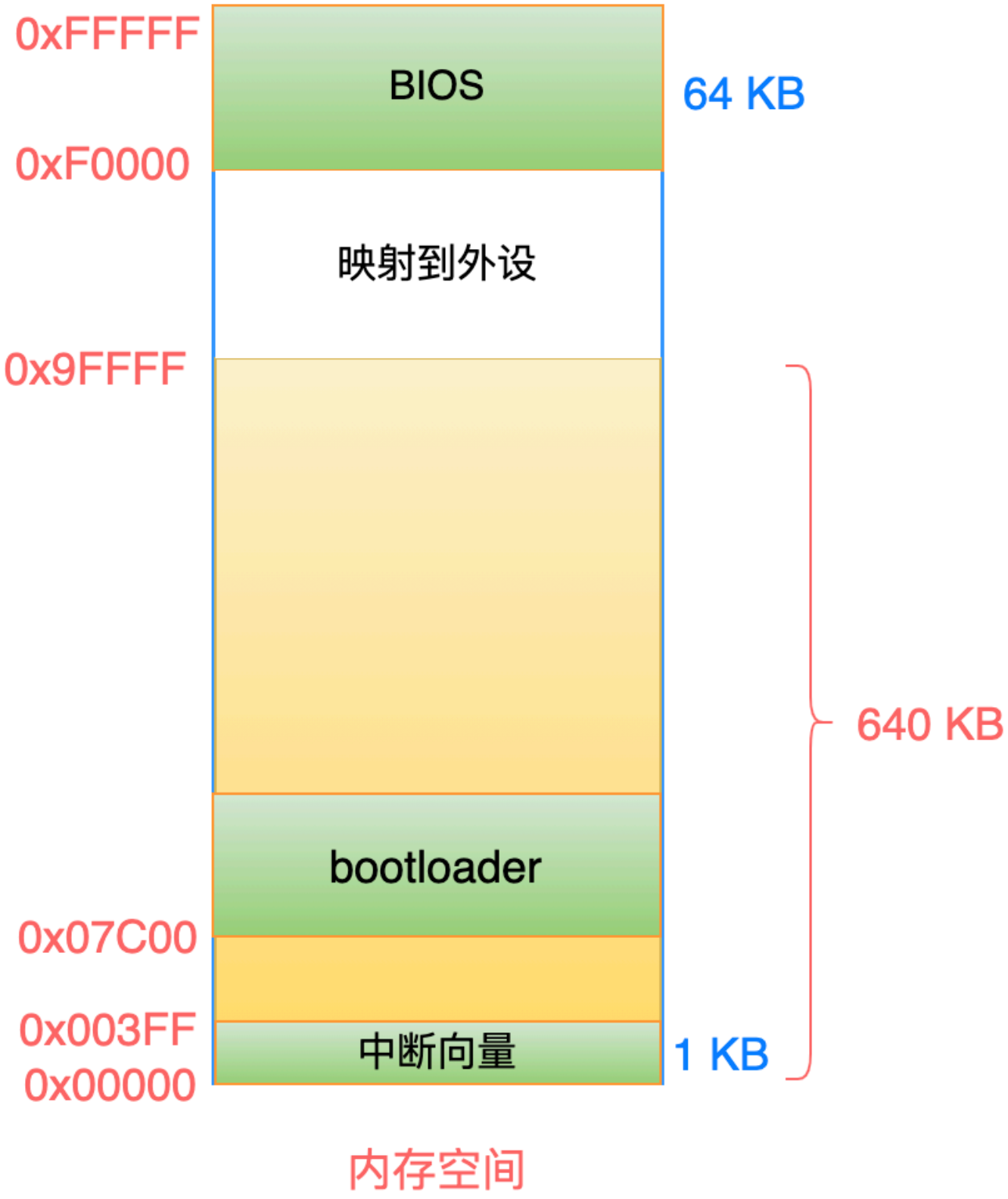
刚才说到，每一个段的开始地址都是按照 16 字节对齐的，因此在 Header 之后，要空余 8 个字节的空间，之后，才是代码段的开始地址(0x00020 = 32 Bytes)。

bootloader 把程序从硬盘读取到内存

1. 读取到内存中的什么位置?

bootloader 在把操作系统文件，从硬盘上读取到内存之前，必须决定一件事情：把文件内容存放到内存中的什么位置?

从上一篇文章我们了解到，在读取操作系统之前，内存布局模型是下面这样的：



注意：这是 8086 系统中，20 根地址线能够寻址的 1 MB 的地址空间。

其中顶部的 64 KB，映射到 ROM 中的 BIOS 程序。

底部从 0 开始的 1 KB 地址空间，是存储 256 个中断向量(下一篇文章准备聊聊中断的事情)。

中间的从 0x07C00 地址开始的地方，是 BIOS 从硬盘的引导区读取的 bootloader 程序所存放的地方。

黄色部分的空间一共是 640 KB 的空间，都是映射到 RAM 中的，因此，有足够大的空闲地址空间来存储操作系统程序文件。

假设：bootloader 就决定从地址 0x20000 开始(128 KB)，存放从硬盘中读取的操作系统程序文件。

2. bootloader 设置数据段基地址

从硬盘上读取文件，是按照扇区为读取单位的，也就是每次读取一个扇区(512 字节)。

至于如何通过指定扇区号、发送端口命令，来从硬盘上读取数据，这是另一个话题，暂且不表，我们把目光集中在 bootloader 上。

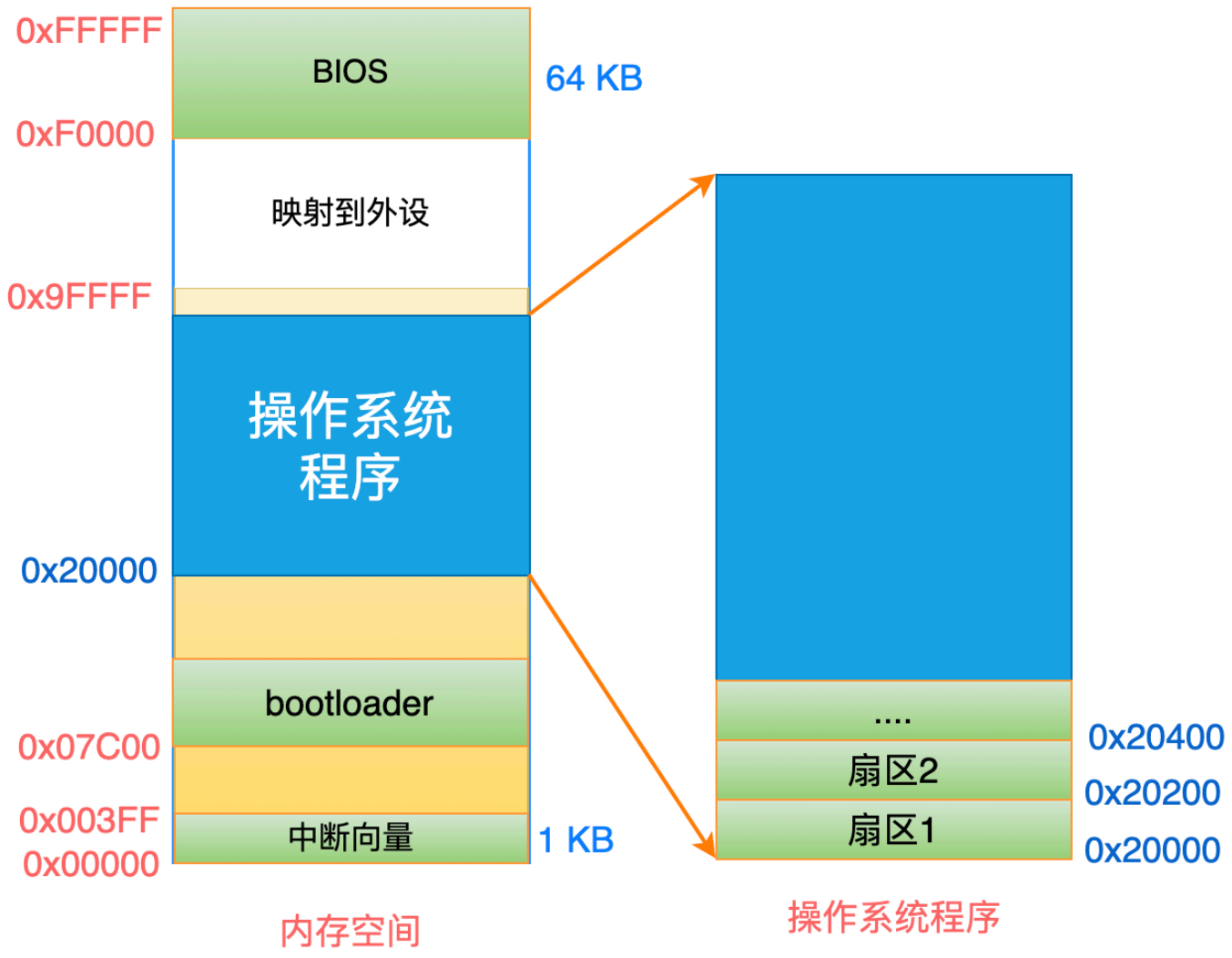
对于 bootloader 来说，读取操作系统文件就相当于读取普通的数据。

既然已经决定把读取的数据从地址 0x20000 开始存放，那么 bootloader 就要把数据段寄存器 ds 设置为 0x2000，这样的话，经过逻辑地址的计算公式：

> 物理地址 = 逻辑段地址 * 16 + 偏移地址

才能得到正确的物理地址，例如：

> 读取的第 1 个扇区的数据放在：0x2000:0x0000 地址处;
>
> 读取的第 2 个扇区的数据放在：0x2000:0x0200 地址处;
>
> 读取的第 3 个扇区的数据放在：0x2000:0x0400 地址处;
>
> ...
>
> 读取的第 10 个扇区的数据放在：0x2000:0x1200 地址处;

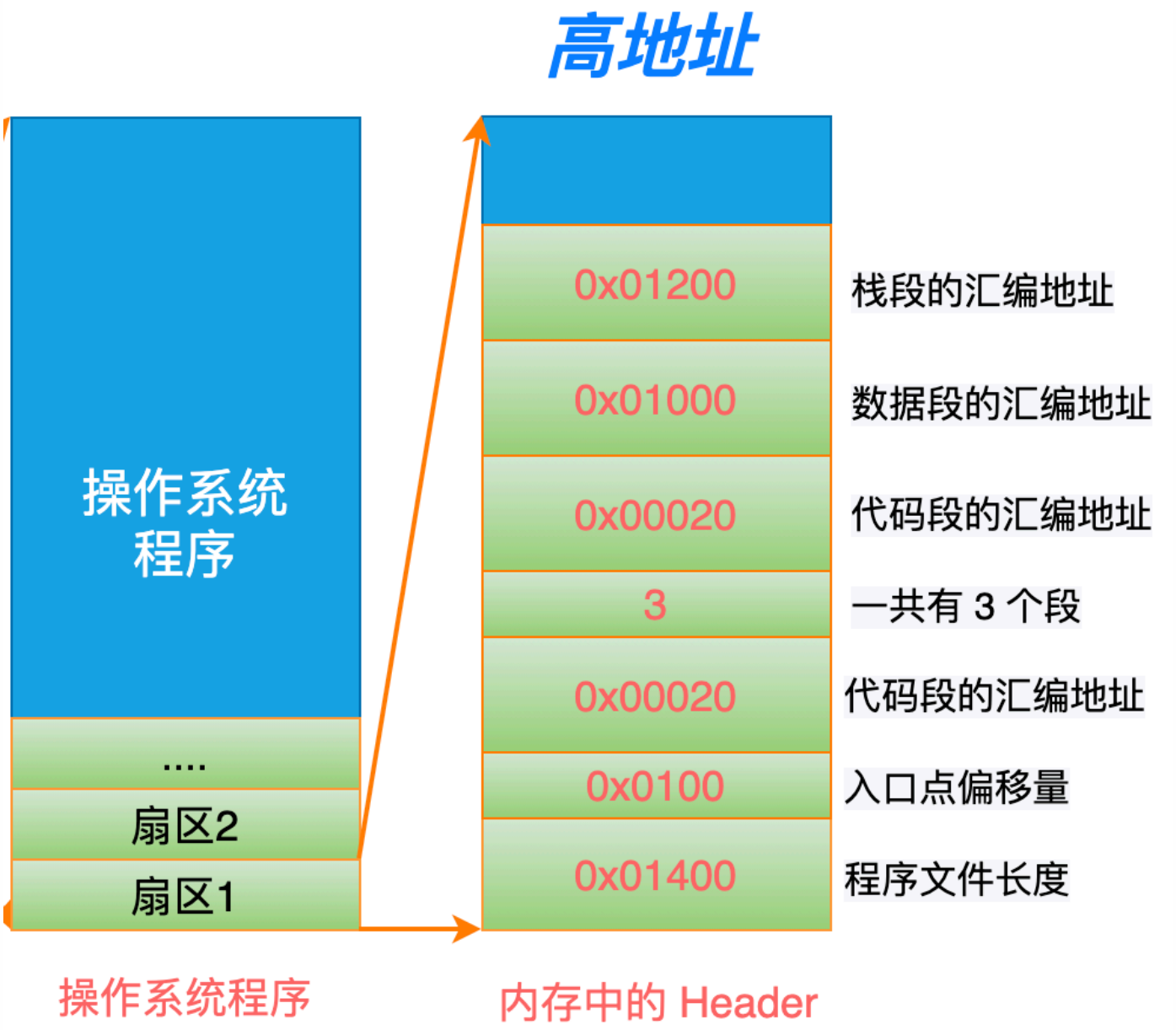


3. bootloader 读取所有扇区

bootloader 需要把操作系统程序的所有内容读取到内存中，需要读取的长度是多少呢?

程序文件的 Header 中有这个信息，因此，bootloader 需要先读取程序文件的第一个扇区，也就是 512 字节，放在 0x20000 开始的位置。

我们继续假设一下：程序的总长度是 5K 字节(0x01400)，那么程序文件的前 512 个字节(第一个扇区)读取到内存中，就是下面这个样子：



注意：这是文件内容被读取到内存中的布局，最下面是低地址，最上面是高地址，这与前面描述静态文件中内容的顺序是相反的。

读取了第一个扇区之后，就可以取出 `0x20000` 开始的 4 个字节的数据：`0x01400`，得到程序文件的总长度: 5 K 字节。

每个扇区是 512 字节，5 K 字节就是 10 个扇区。

第一个扇区已经读取了，那么还需要继续读取剩下的 9 个扇区。

于是，`bootloader` 把所有扇区的数据，依次读取到：0x2000:0x0000, 0x2000:0x0200, 0x2000:0x0400, ... 0x2000:0x1200 地址处。

4. 如果程序文件超过 64 KB 怎么办?

这里有一个延伸的问题可以思考一下:

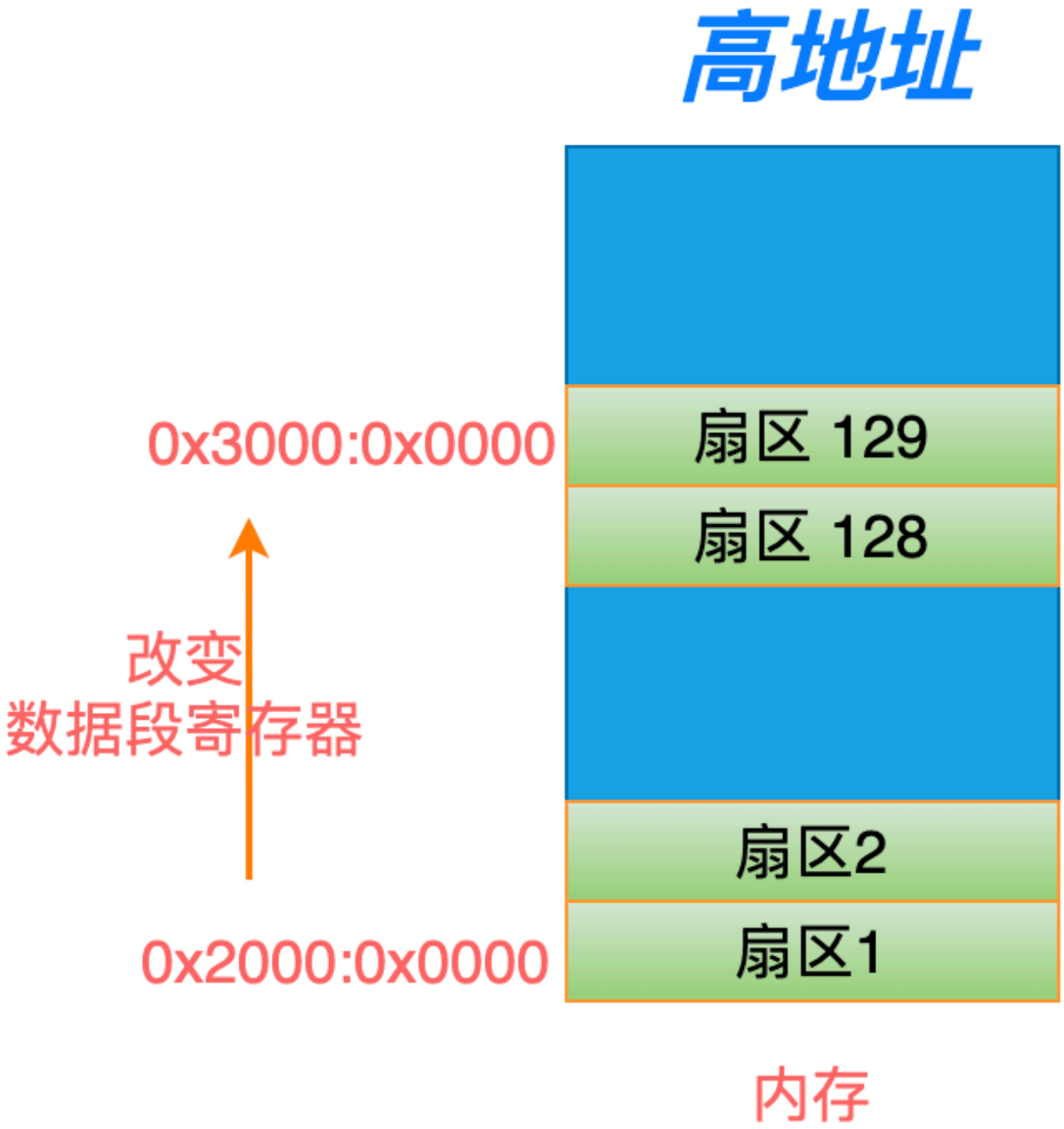
8086 的段寻址方式，由于偏移量寄存器的长度是 16 位，最大只能表示 64 KB 的空间。

我们所假设的例子中，程序文件只有 5 KB，在一个数据段内完全可以包括，因此 bootloader 可以一直用 0x2000:偏移量 的方式来读取文件内容。

那如果程序的长度是 100 KB，超过了偏移量的 64 KB 最大寻址空间，那么 bootloader 应该怎么样做才能正确把 100 KB 的程序读取到内存中?

解答:

可以在读取文件的过程中，动态的增加数据段逻辑地址。



比如，在读取前面的 64 KB 数据(扇区 1 ~ 扇区 128)时，段寄存器 ds 设置为 0x2000。

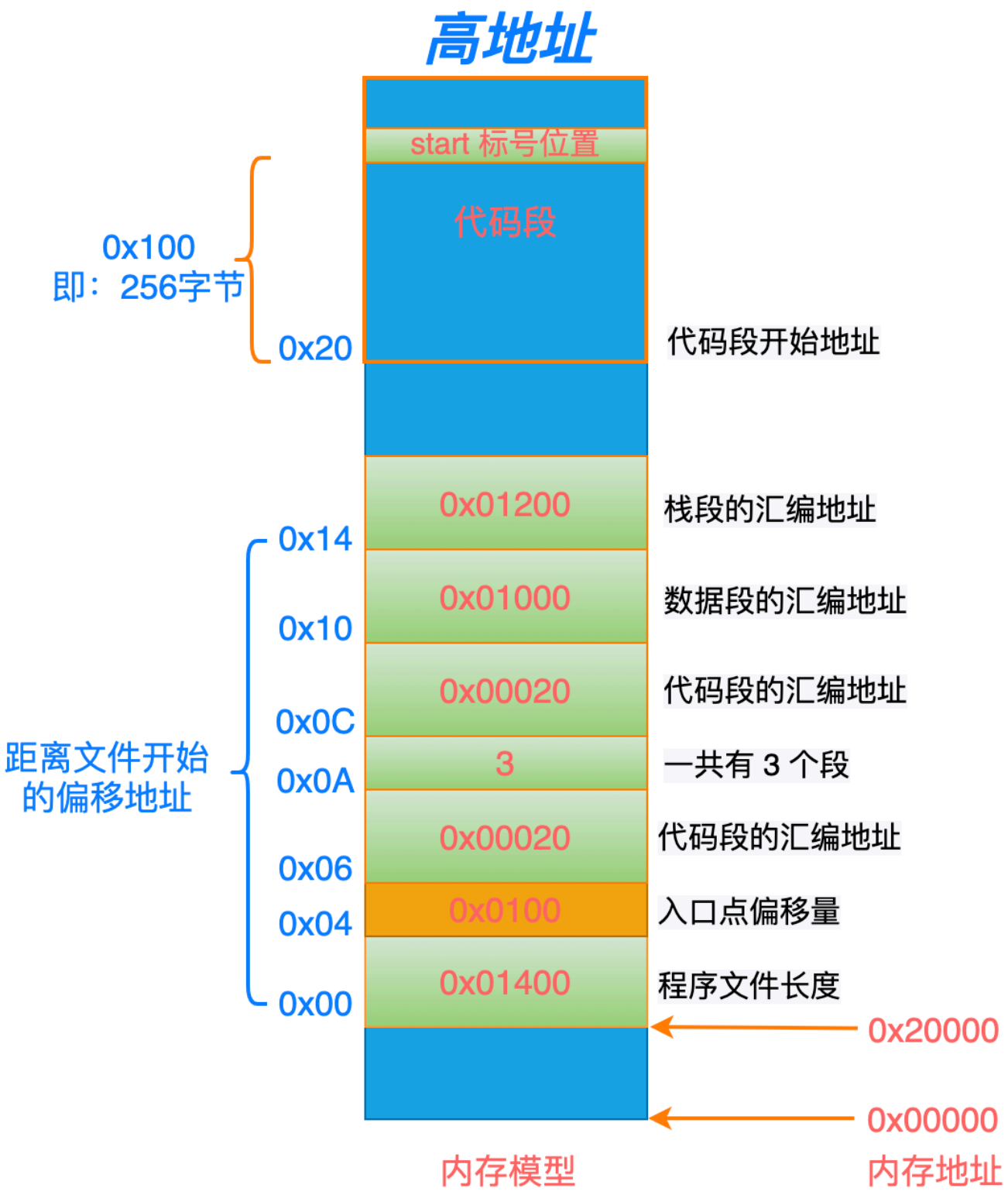
在读取第 65 KB 数据(扇区 129)之前，把段寄存器 ds 设置为 0x3000，这样读取的数据就从 0x3000:0x0000 处开始存放了。

代码重定位

现在，操作系统程序已经被读取到内存中了，下一个步骤就是：跳转到操作系统的程序入口点去执行！

程序入口点重定位

程序入口点的偏移量，已经被记录在 Header 中了(0x04 ~ 0x05 字节，橙色部分)：

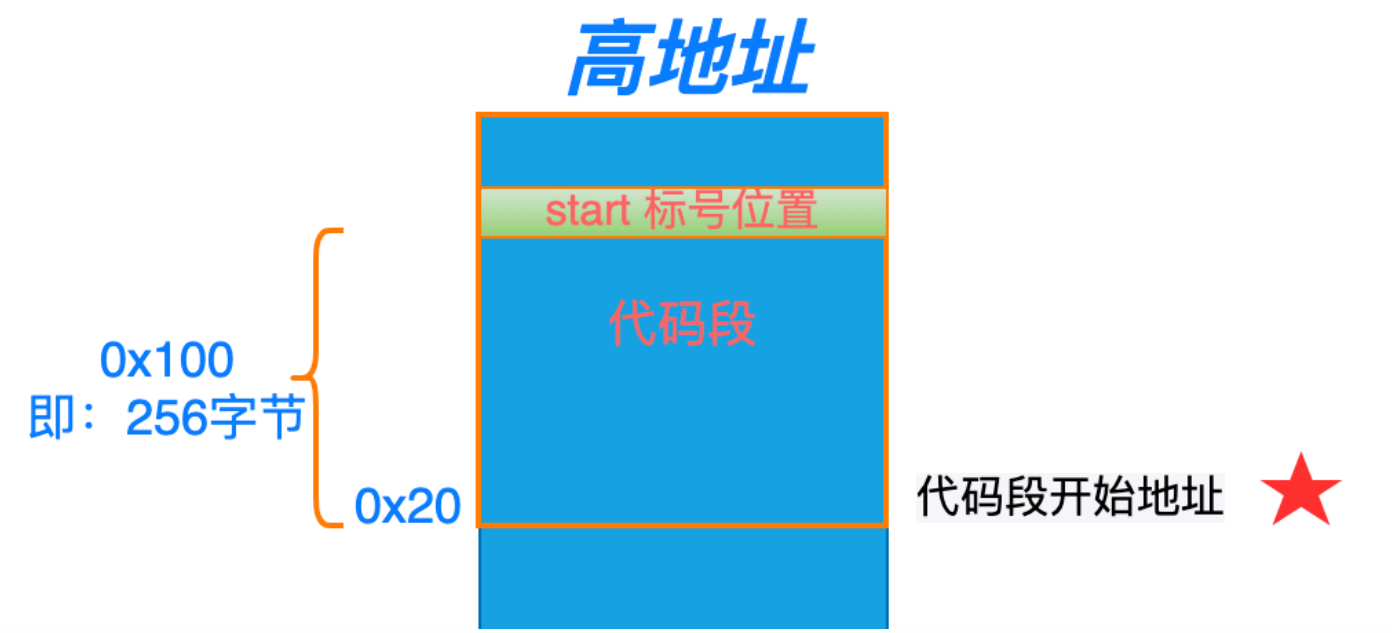


Header 中记录的代码段中入口点 start 标签的偏移量是 0x100，即：入口点距离代码段的开始地址是 256 个字节。

> 同样的道理，代码段中所有相关的地址，都是相对于代码段的开始地址来计算偏移量的。

因此，如果（这里是如果啊） bootloader 把代码段的开始地址(不是整个文件的开始)，直接放到内存的 0x00000 地址处，那么代码段里所有地址就都不用再修改了，可以直接设置：cs = 0x0000, ip=0x0100，这样就直接跳转到 start 标签的地方开始执行了。

可惜，bootloader 是把操作系统程序读取到地址 0x20000 开始的地方，因此，需要把代码段寄存器 cs 设置为当前代码段在内存中的实际开始位置，也即是下面这个五角星的位置：



> 以上两段文字，可以再多读几遍！

在 Header 中，0x06，0x07, 0x08, 0x09 这 4 个字节的数据 0x00020，就是代码段的开始位置距离程序文件开头的字节数。

只要把这个数值(0x00020)，与文件存储的开始地址(0x20000)相加，就可以得到代码段的开始地址在物理内存中的绝对地址：

> 0x00020 + 0x20000 = 0x20020

即：代码段的开始地址，位于物理内存中 0x20020 的位置。

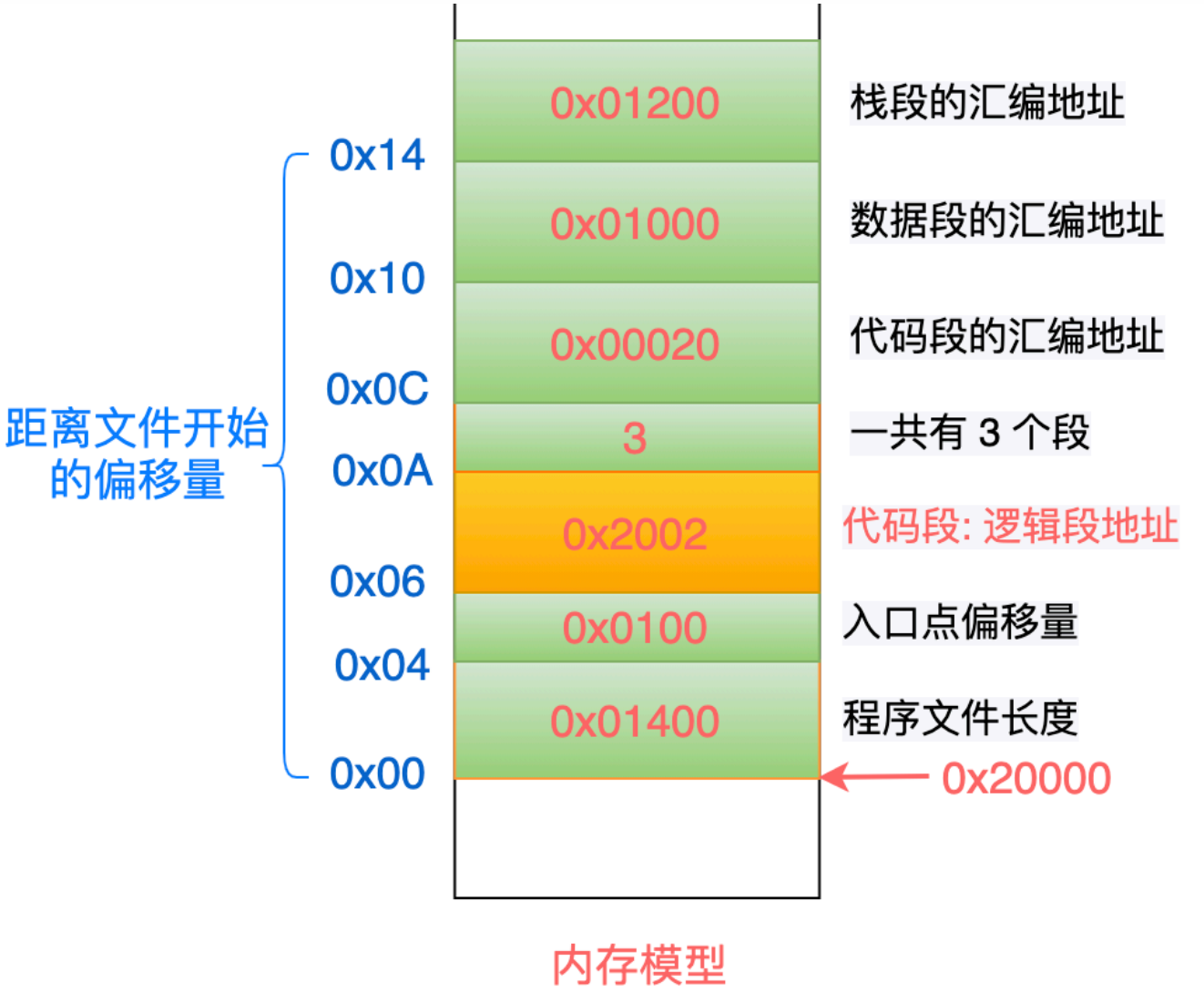
对于一个物理地址，我们可以用多种不同的逻辑地址来表示，例如：

> 0x20020 = 0x2002:0x0000
> 0x20020 = 0x2000:0x0020
> 0x20020 = 0x1FF0:0x0120

面对这 3 个选择，我们当然是选择第 1 个，而且只能选择第 1 个，因为代码段内部所有的地址偏移，在编译的时候都是基于 0 地址的(也即是上面所说的汇编地址)，或者称作相对地址。

明白了这个道理之后，就可以把 cs:ip 设置为 0x2002:0x0100，这样 CPU 就会到 start 标签处执行了。

但是，在进行这个操作之前还有其他几件事情需要处理，因此，要把代码段的逻辑段地址 0x2002, 写回到 Header 中的 0x06 ~ 0x09 这 4 个字节中保存起来(橙色部分)：



段表重定位

此时，系统还是在 bootloader 的控制之下，数据段寄存器 ds 仍然为 0x2000，想一想为什么？

> 因为 bootloader 读取操作系统程序的第一扇区之前，希望把数据读取到物理地址 0x20000 的地方，右移一位就得到了逻辑段地址 0x2000，把它写入到数据段寄存器 ds 中。
>
> 我们一直忽略了 bootloader 使用的栈空间，因为这部分与文件主题无关。

操作系统程序如果想要执行，必须使用自己程序文件中的数据段和栈段。

但是，Header 中记录的这 2 个段的开始地址，都是相对于程序文件开头而言的。

而且操作系统文件并不知道：自己被 bootloader 读取到内存中的什么位置？

因此，bootloader 也需要把这 2 个段，在内存中的开始地址进行重新计算，然后更新到 Header 中。

这样的话，当操作系统程序开始执行的时候，才能从 Header 中得到数据段和栈段的逻辑段地址。

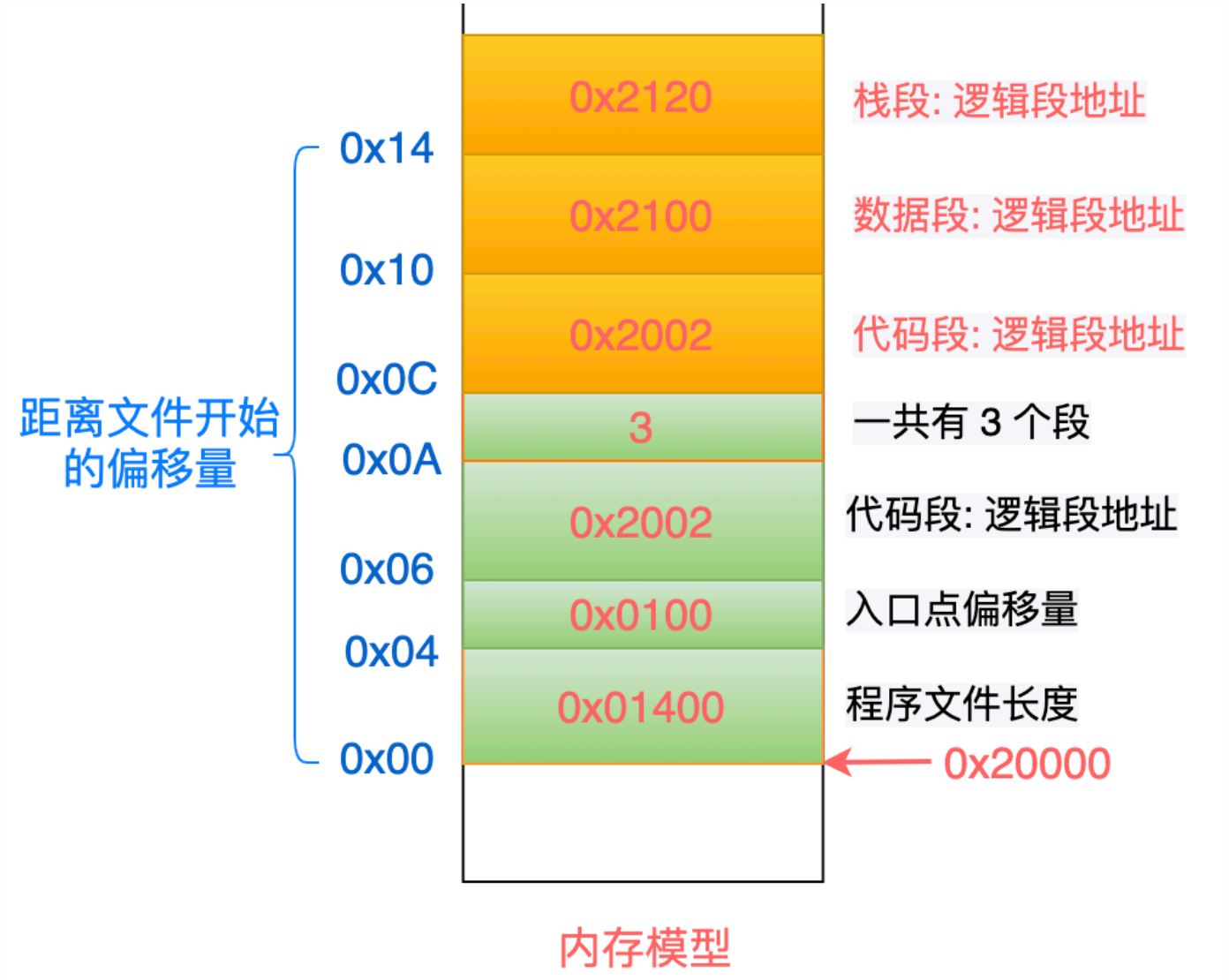
当然了，这里所举的示例中只有 3 个段，一个实际的程序可能会包括很多个段，每一个段的地址都需要进行重定位。

bootloader 从 Header 的 0x0A ~ 0x0B 这 2 个字节，可以得到一共有多少个段地址需要重定位。

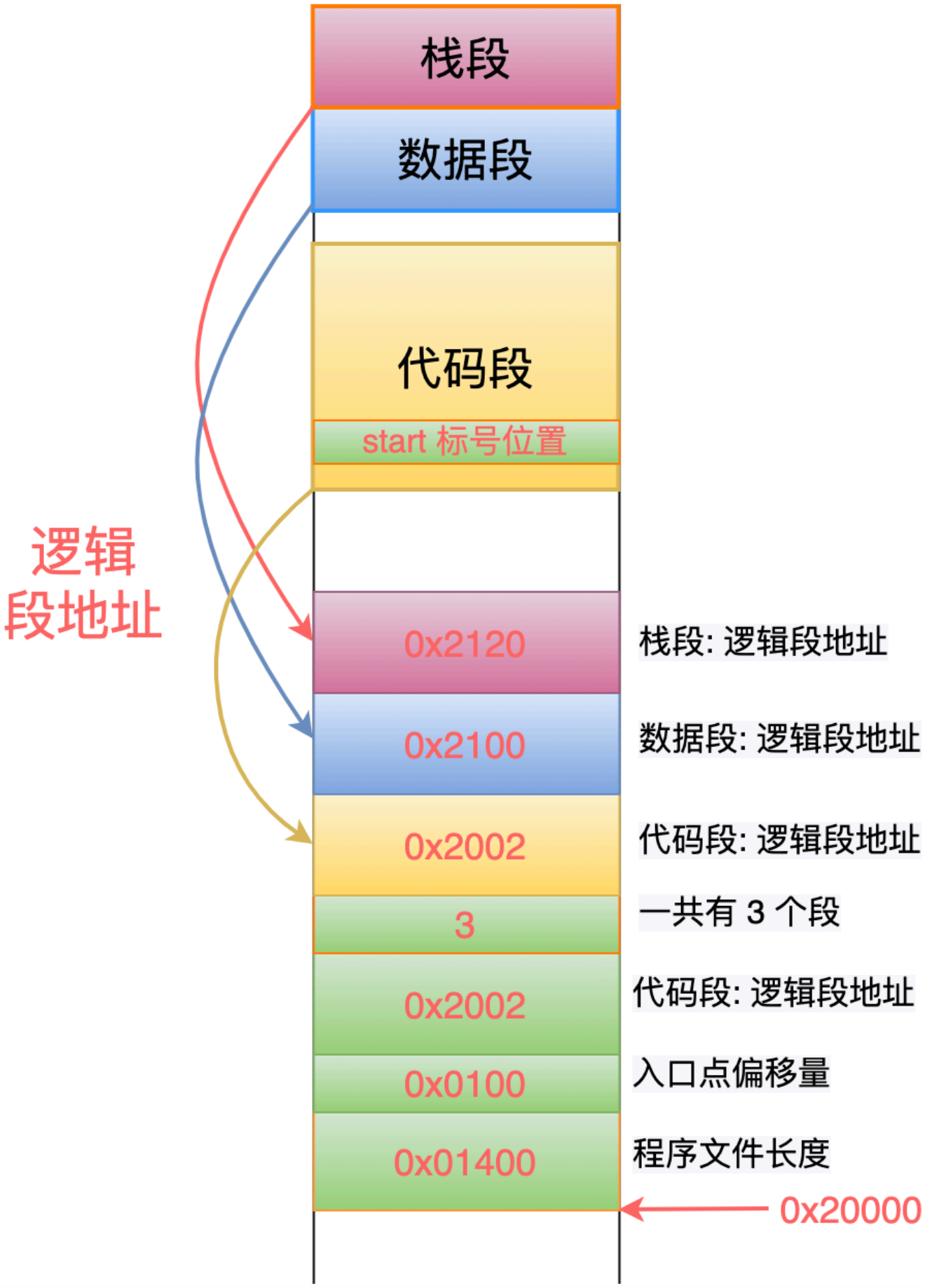
然后按照顺序，依次读取每一个段的偏移地址，加上程序文件的加载地址(0x20000)，计算出实际的物理地址，然后再得到逻辑段地址，具体如下：

> 代码段偏移量 0x00020：0x20000 + 0x00020 = 0x20020(物理地址)，右移一位得到逻辑段地址：0x2002;
>
> 数据段偏移量 0x0x01000: 0x20000 + 0x01000 = 0x21000(物理地址)，右移一位得到逻辑段地址：0x2100;
>
> 栈段 段偏移量 0x0x01200: 0x20000 + 0x01200 = 0x21200(物理地址)，右移一位得到逻辑段地址：0x2120;

下图橙色部分：



我们把代码段、数据段、栈段在内存中的布局模型全部画出来：


栈段
数据段
代码段
start 标号位置
逻辑
段地址
0x2120
栈段: 逻辑段地址
0x2100
数据段: 逻辑段地址
0x2002
代码段: 逻辑段地址
3
一共有 3 个段
0x2002
代码段: 逻辑段地址
0x0100
入口点偏移量
0x01400
程序文件长度
0x20000

跳转到程序的入口地址

万事俱备，只欠东风！

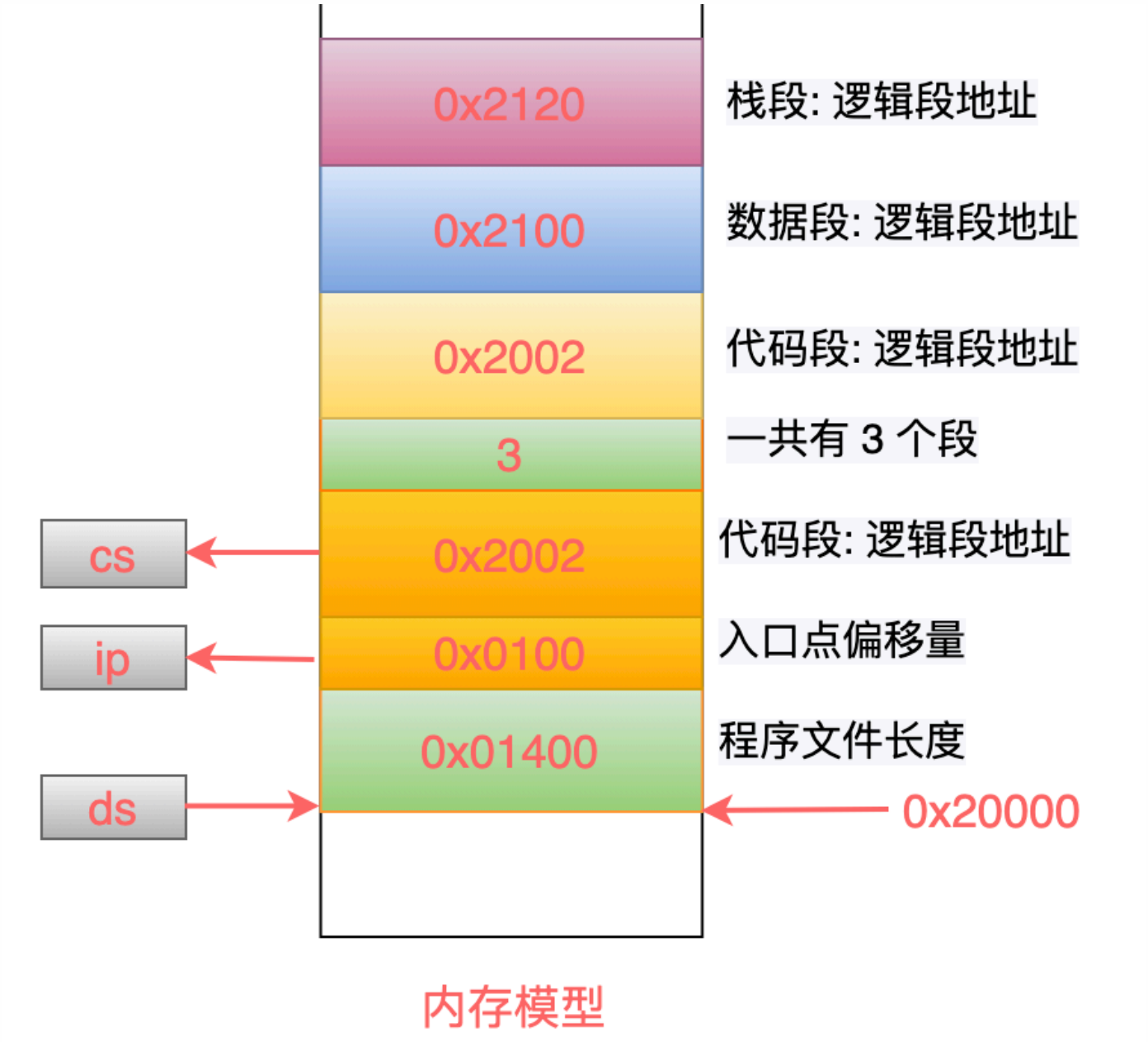
一切工作已经准备就绪，最后一步就是进入操作系统程序中代码段的 `start` 入口点了。

在上面的准备工作中，`bootloader` 已经把程序代码段的逻辑段地址 `0x2002`，保存在 `Header` 中的 0x06 ~ 0x09 这 4 个字节中了，只要把它赋值给代码段寄存器 `cs` 即可。

程序入口点位于 `start` 标签处，它距离代码段的开始位置偏移 `0x100`，保存在 `Header` 中的 0x04 ~ 0x05 这 2 个字节，只要把它赋值给指令指针寄存器 `ip` 即可。

我们可以手动读取，然后赋值。

也可以直接利用 8086 CPU 中的这条指令： jmp [0x04] 来实现 `cs:ip` 的赋值。



因为此刻还是在 `bootloader` 的控制下，数据段寄存器 `ds` 的值仍然为 `0x2000`，因此跳转到 `0x2000:0x04` 内置中所表示的地址，就可以把正确的逻辑段地址和指令地址赋值给 `cs:ip`，从而开始执行操作系统程序的第一条指令。

操作系统程序的执行

操作系统的第一条指令在执行时，数据段寄存器 ds 和 栈段寄存器 cs 中的值，仍然为 bootloader 中所设置的值。

因此，操作系统首先要把这 2 个段寄存器设置为自己程序文件的值，然后才能开始后续指令的执行。

上文已经说过，每一个段在内存中的逻辑段地址，已经被 bootloader 重新计算，并且更新到了 Header 中。

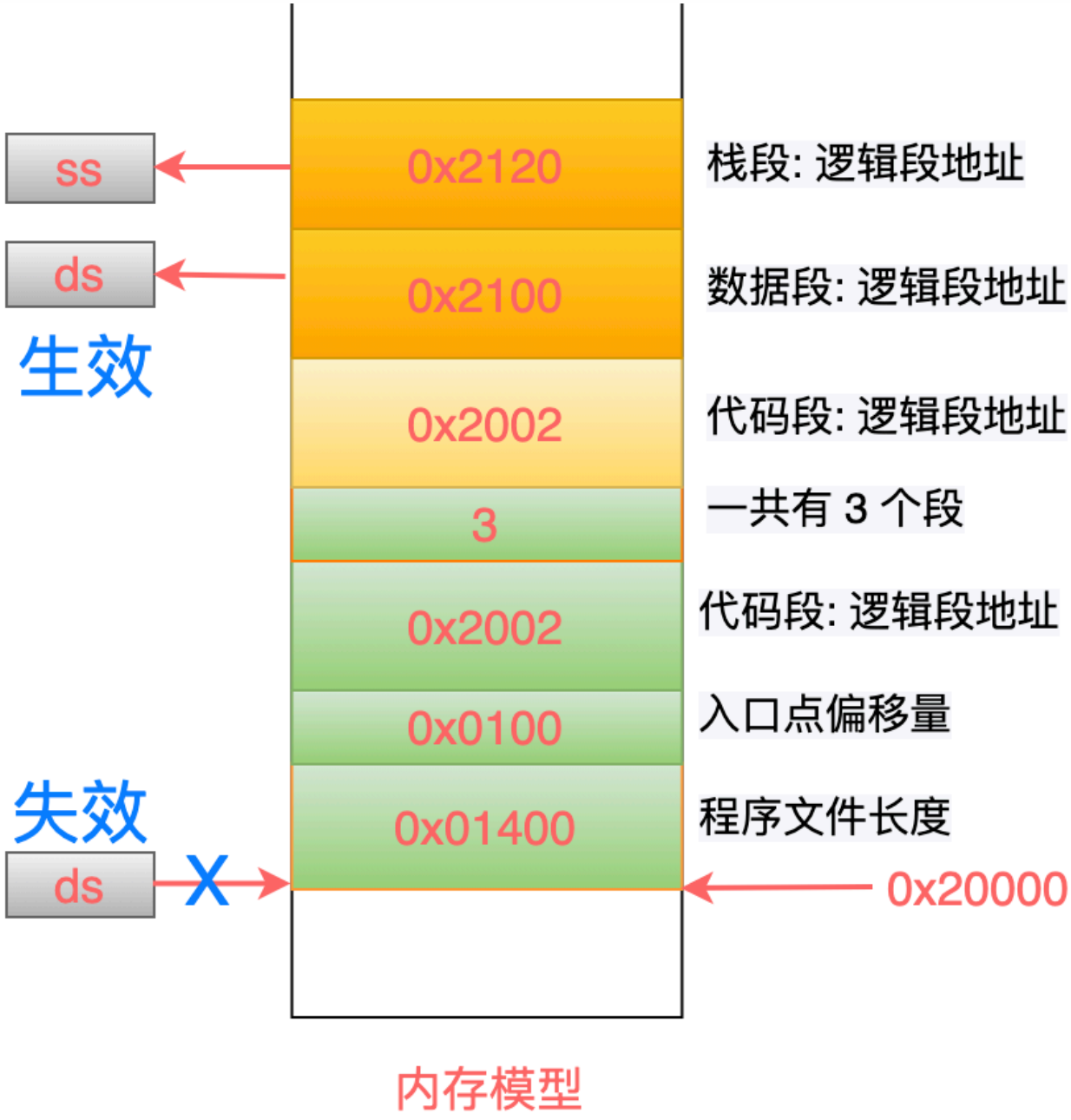
所以，操作系统就可以从 ds:0x14 的位置，读取新的栈段逻辑地址 0x2120，并把它赋值给栈段寄存器 cs。

从这个时候开始，所有的栈操作就是操作系统程序自己的了。

> 注意：此时数据段寄存器 ds 仍然没有改变，仍然是 bootloader 中使用的 0x2000。

然后再从 ds:0x10 的位置读取新的数据段逻辑地址 0x2100，并把它赋值给数据段寄存器 ds。

从这个时候开始，所有的数据操作就是操作系统程序自己的了。



注意：给 cs、ds 的赋值顺序不能颠倒。

如果先给 ds 赋值，那么再去 Header 中读取 cs 逻辑段地址的时候，就没法定位了。

因为此时 ds 寄存器已经指向了新的地址(ds = 0x2100)，没法再去 0x2000:0x14 地址处获取数据了。

最后还有一点，对于栈操作，除了设置栈的段寄存器 ss 外，还需要设置栈顶指针寄存器 sp。

我们假设程序中设置的栈空间是 512 字节，栈顶指针是向低地址方向增长的，因此，需要把 sp 初始化为 512。

至此，操作系统程序终于可以愉快的开始执行了！

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

这篇文章，我们描述了关于代码重定位的最底层原理。

在以后学习到 Linux 中的重定位相关知识时，会接触到更多的概念和技巧，但是最底层的基本原理都是相通的。

希望这篇文章，能够成为你前进路上的垫脚石！

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

星标公众号，能更快找到我！

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。