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。

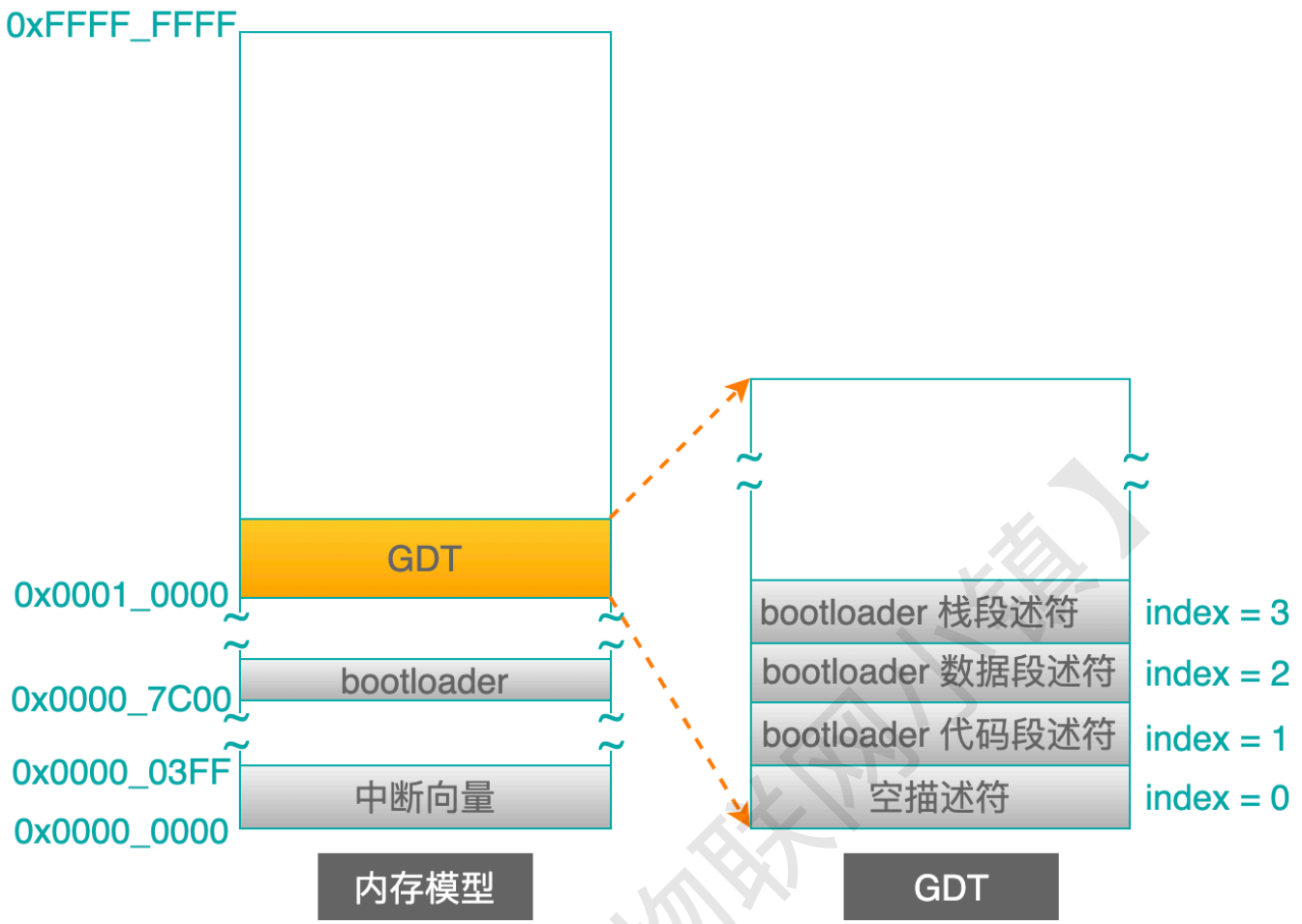
不论是在 x86 平台上，还是在嵌入式平台上，系统的启动一般都经历了 bootloader 到 操作系统，再到应用程序，这样的三级跳过程。

每一个相互交接的过程，都是我们学习的重点。

这篇文章，我们仍然以 x86 平台为例，一起来看一下：从上电之后，系统是如何一步一步的进入应用程序的入口地址。

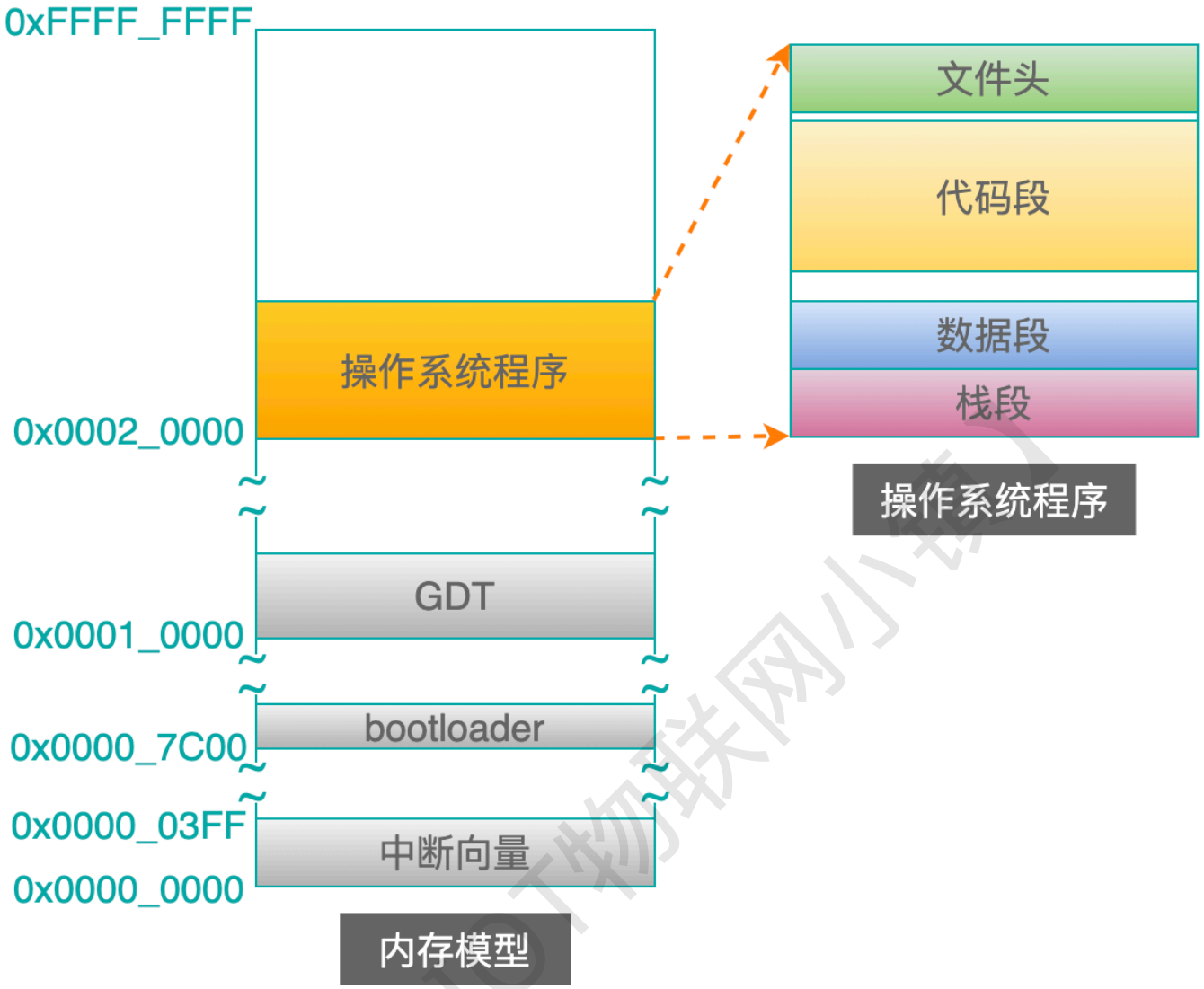
bootloader 跳转到操作系统

在上一篇文章中，讨论了 bootloader 在进入保护模式之后，在地址 0x0001_0000 处创建了全局描述符表(GDT)，表中创建了 3 个段描述符：



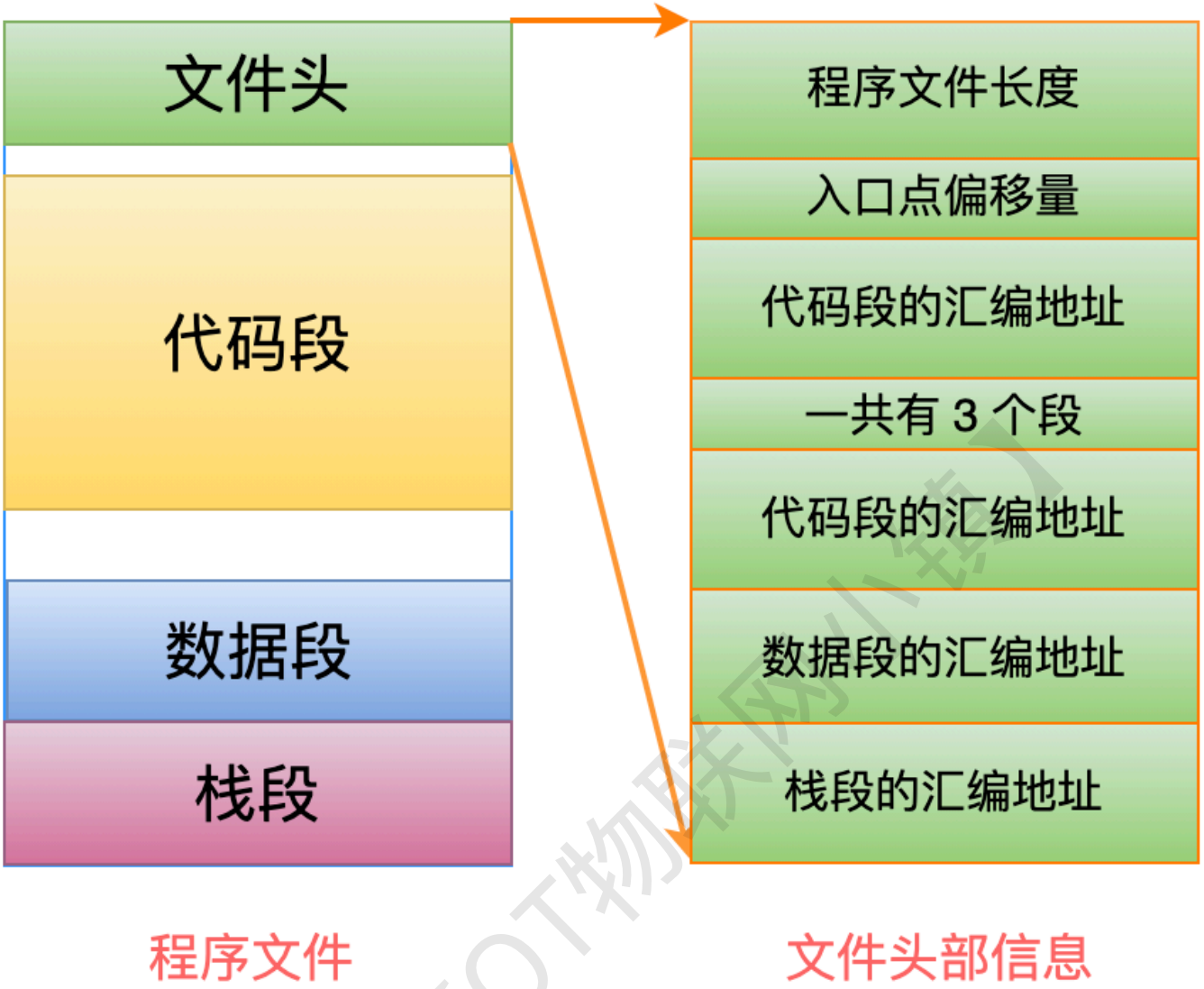
只要在 GDT 中创建了这 3 个描述符，然后把 GDT 的地址(eg: 0x0001_0000)设置到 GDTR 寄存器中，此时就可以进入保护模式工作了(设置 CR0 寄存器的 bit0 为 1)。

之前的第 6 篇文章中Linux从头学06：16张结构图，彻底理解【代码重定位】的底层原理，我们是假设 bootloader 把操作系统程序读取到内存 0x0002_0000 的位置，这里继续使用这个示例：



关于文件头 header 的内容，与实模式下是不同的。

在实模式下，header 的布局如下图：



bootloader 在把操作系统，从硬盘加载到内存中之后，从 header 中取得 3 个段的汇编地址(即：段的开始地址相对于文件开始的偏移量)，然后计算得到段的基地址，最后把段基地址写回到 header 的这 3 个段地址空间中。

这样的话，操作系统开始执行时，就可以从 header 中准确的获取到每一个段的基地址了，然后就可以设置相应的段寄存器，进入正确的执行上下文了。

那么在保护模式下呢，操作系统需要的就不是段的基地址了，而是要获取到每一个段的描述符才行。

很显然，需要借助 bootloader 才可以完成这个目标，也就是：

1. 在 GDT 中为操作系统程序中的三个段，建立相应的描述符;
2. 把每一个段的描述符索引号，写回到操作系统程序的 header 中;

注意：

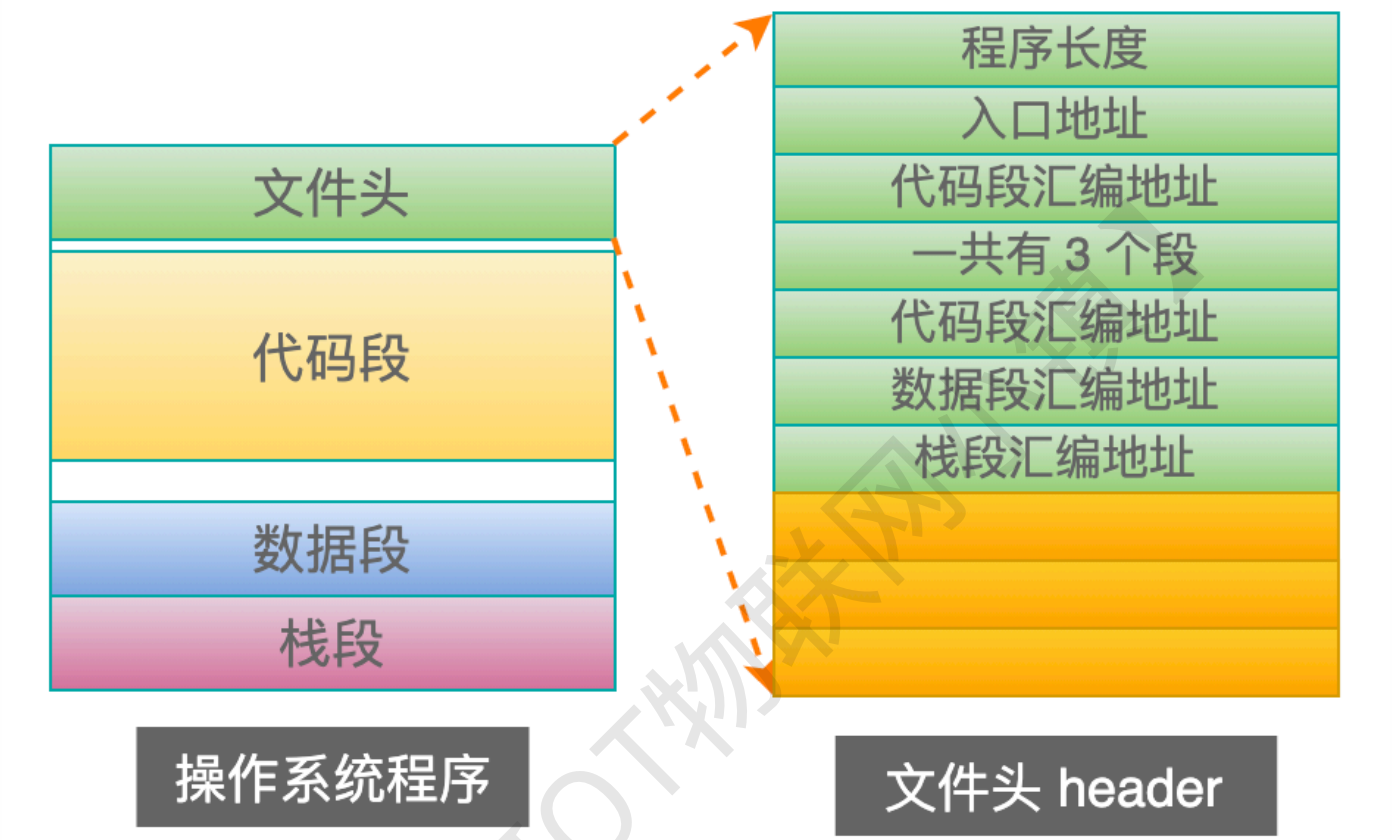
这里描述的仅仅是一个可能的过程，主要用来理解原理。

有些系统可以用不同的实现方式，例如：在进入操作系统之后，在另外一个位置存放 GDT，并重新创建其中的段描述符。

操作系统的 header 布局

既然 header 需要作为媒介，来接收 bootloader 往其中写入段索引号，所以 bootloader 与 OS 就要协商好，写在什么位置?

可以按照之前的方式，直接覆写在每个段的汇编地址位置，也可以写在其他的位置，例如：



其中，最后的 3 个位置可以用来接收操作系统的三个段索引号。

建立操作系统的三个段描述符

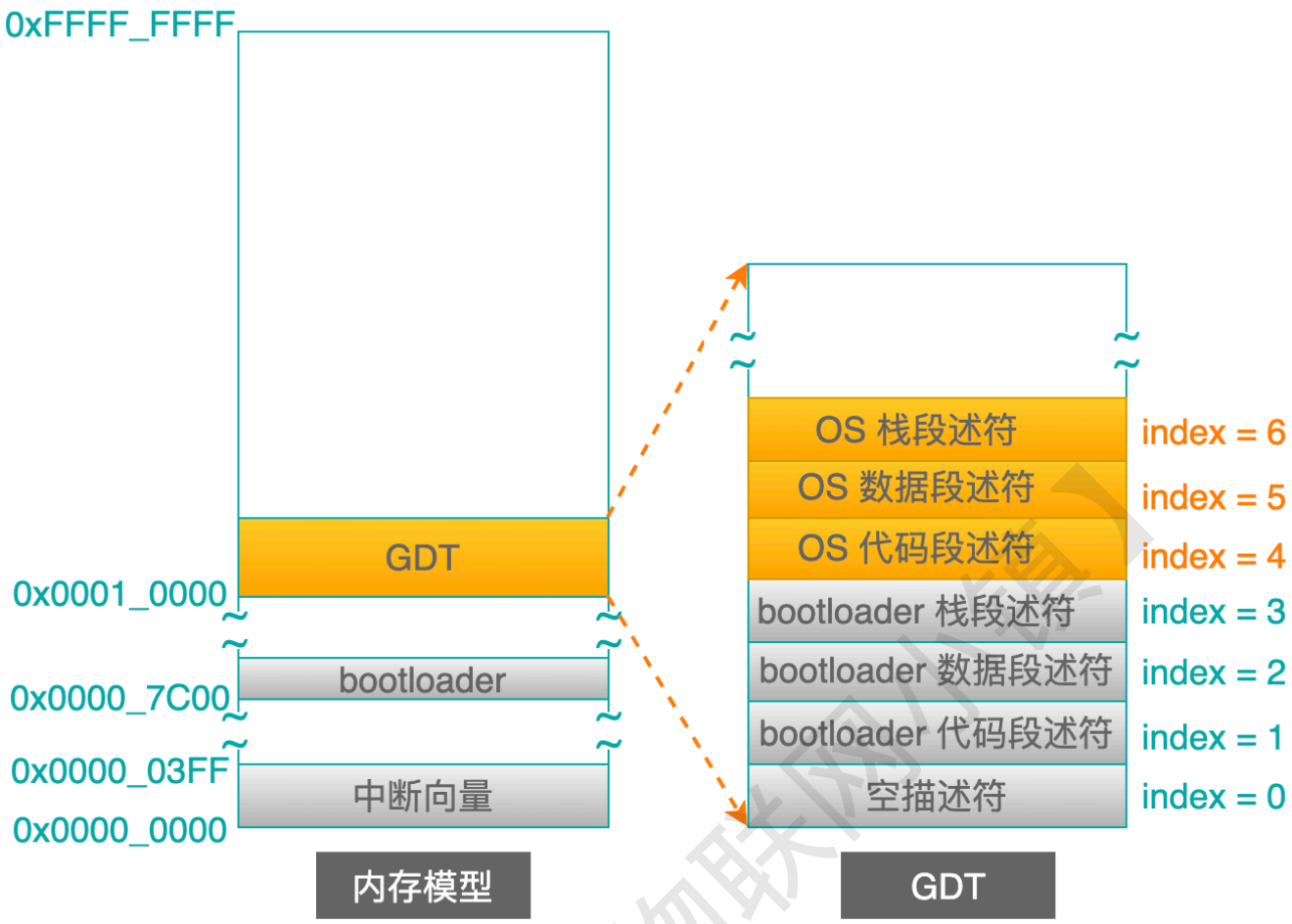
bootloader 把 OS 加载到内存中之后，会解析 OS 的 header 中数据，得到每个段的基地址以及界限。

虽然 header 中没有明确的记录每个段的界限，可以根据下一个段的开始地址，来计算得到上一个段的长度。

我们可以联想一下：

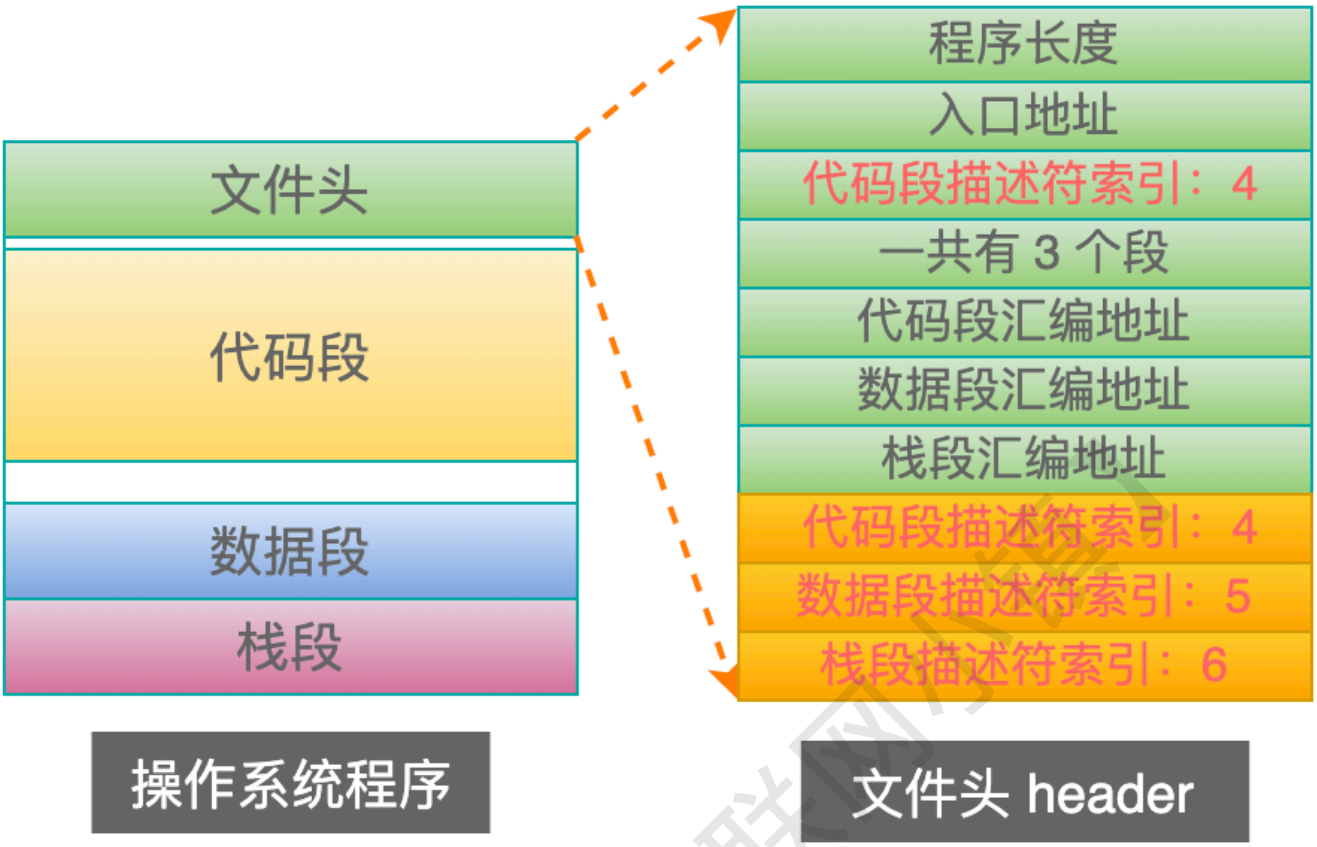
现代 Linux 系统中 ELF 文件的格式，在文件头部中记录了每一个段的长度，具体解析请参考这篇文章：Linux系统中编译、链接的基石-ELF文件：扒开它的层层外衣，从字节码的粒度来探索。

此时，bootloader 就可以利用这几个信息：段基地址、界限、类型以及其他属性，来构造出相应的段描述符了(下图橙色部分)：



> PS：这里的示例只为操作系统创建了 3 个段描述符，实际情况也许有更多的段。

OS 段描述符建立之后，bootloader 再把这 3 个段描述符在 GDT 中的索引号，填写到 OS 的 header 中相应的位置：



上图中，“入口地址”下面的那个 4，本质上是不需要的，加上更有好处，好处如下：

当从 bootloader 跳入到操作系统的入口地址时，需要告诉处理器两件事情：

1. 代码段的索引号;
2. 代码的入口地址;

因此，把入口地址和索引号放在一起，有助于 bootloader 直接使用跳转语句，进入到 OS 的 start 标记处开始执行。

操作系统跳转到应用程序

从现代操作系统来看，这个标题是有错误的：

操作系统是应用程序的下层支撑，相当于是应用程序的 runtime，怎么能叫做跳转到应用程序呢?

其实我想表达的意思是：操作系统是如何加载、执行一个应用程序的。

既然是保护模式，那么操作系统就承担起重要的职责：保护系统不会受到每一个应用程序的恶意破坏！

因此，操作系统：把应用程序从硬盘上复制到内存中之后，跳入应用程序的第一条指令之前，需要为应用程序分配好内存资源：

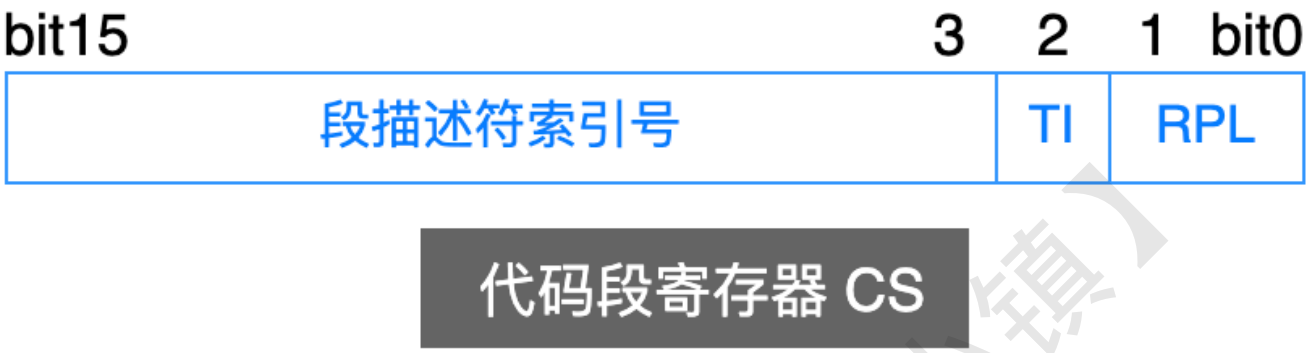
1. 代码段的基地址、界限、类型和权限等信息;
2. 数据段的基地址、界限、类型和权限等信息;

> 3. 栈段的基地址、界限、类型和权限等信息;

以上这些信息，都以段描述符的形式，创建在 GDT 中。

PS: 在现代操作系统中，应用程序都会有一个自己私有的局部描述符表 LDT，专门存储应用程序自己的段描述符。

还记得之前讨论过的下面这张图吗?

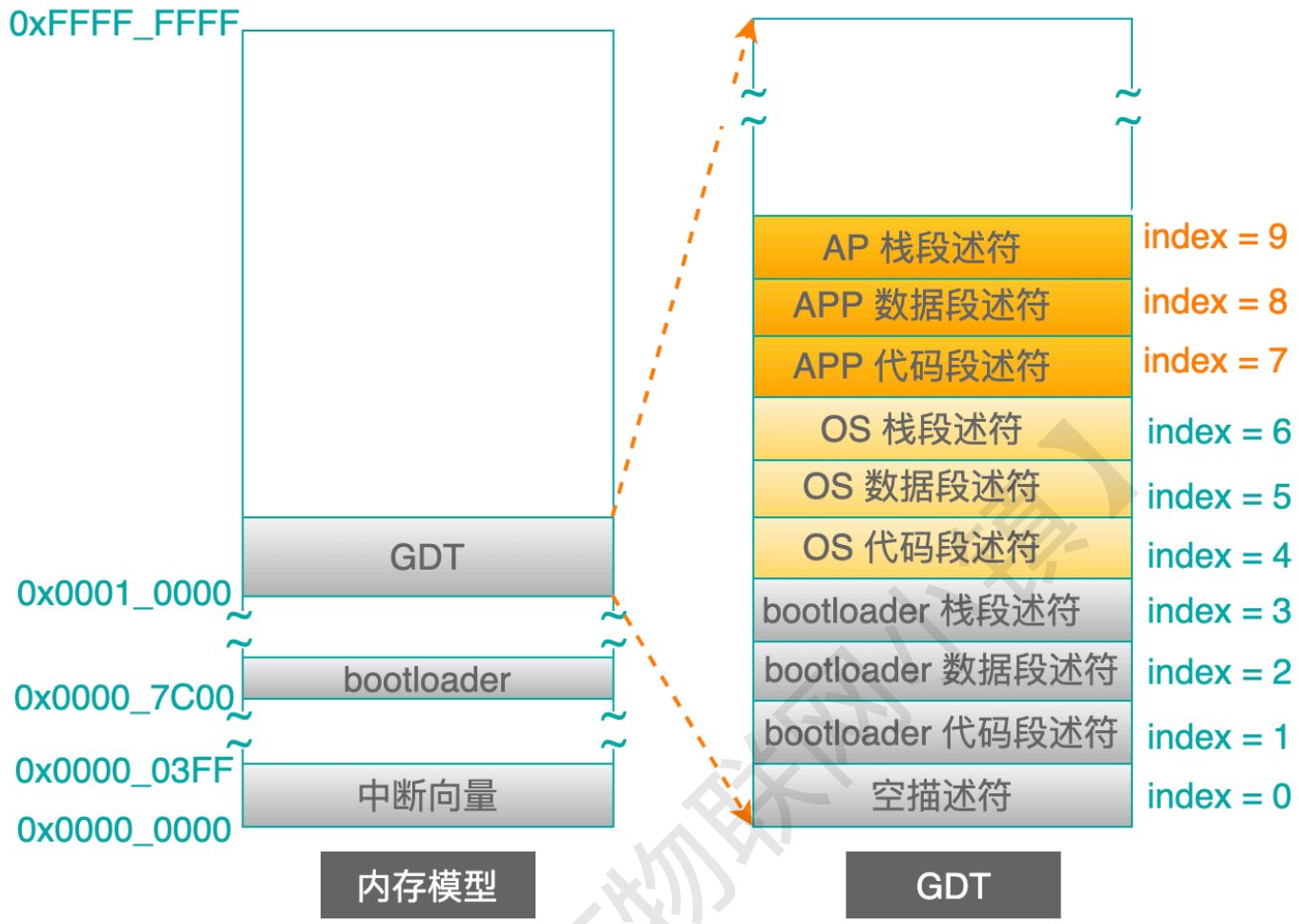


段寄存器的 bit2 位 TI 标志，就说明了需要到 GDT 中查找段描述符？还是到 LDT 中去查找?

为了方便起见，我们就把所有的段描述符都放在 GDT 中。

就犹如 bootloader 为 OS 创建段描述符一样，OS 也以同样的步骤为应用程序来创建每一个段描述符。

此时的 GDT 就是下面这样:



从这张图中已经可以看出一个问题了：

如果所有应用程序的段描述符都放在全局的 GDT 中，当应用程序结束之后，还得去更新 GDT，势必给操作系统的代码带来很多麻烦。

因此，更合理的方式应该是放在应用程序私有的 LDT 中，这个问题，以后还会进一步讨论到。

不管怎样，OS 启动应用程序的整体流程如下：

1. 操作系统把应用程序读取到内存中的某个空闲位置；
2. 操作系统分析应用程序 header 部分的信息；
3. 操作系统为应用程序创建每一个段描述符，并且把索引号写回到 header 中；
4. 跳转到应用程序的入口地址，应用程序从 header 中获取到每个段索引号，设置好自己的执行上下文(即：设置好各种寄存器)；

应用程序调用操作系统中的函数

这里的函数可以理解成系统调用，也就是操作系统为所有的应用程序提供的公共函数。

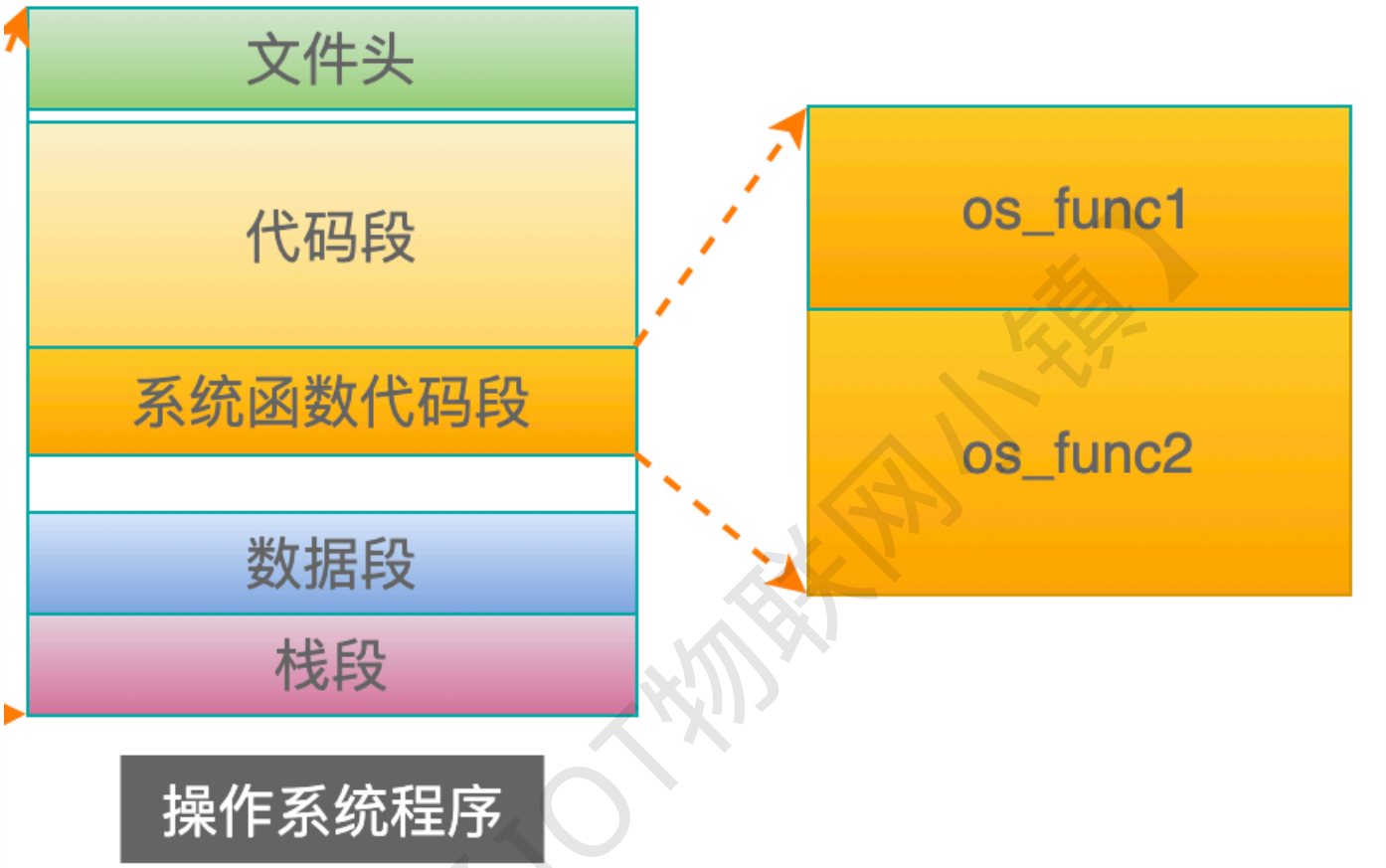
在 Linux 系统中，系统调用是通过中断来实现的，在中断处理器程序中，再通过一个寄存器来标识：当前应用程序想调用哪一个系统函数，也就是说：每一个系统函数都有一个固定的数字编号。

再回到我们当前讨论的 x86 处理器中，操作系统提供系统函数的最简单的方法就是：

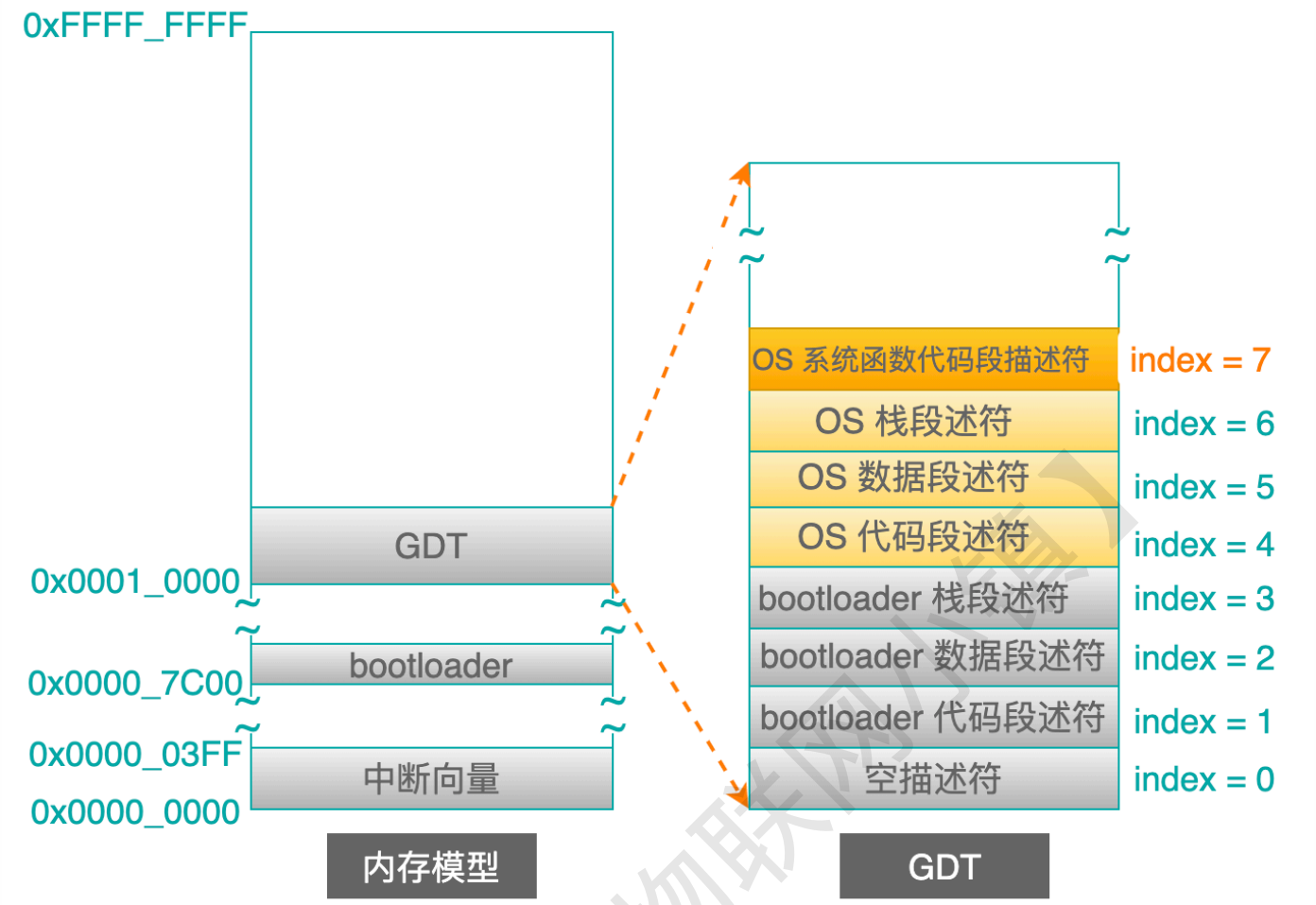
把所有的系统函数都放在一个单独的代码段中，把这个段的索引号以及每一个系统函数的偏移地址告诉应用程序。

这样的话，应用程序就可以通过这 2 个信息调用到系统函数了。

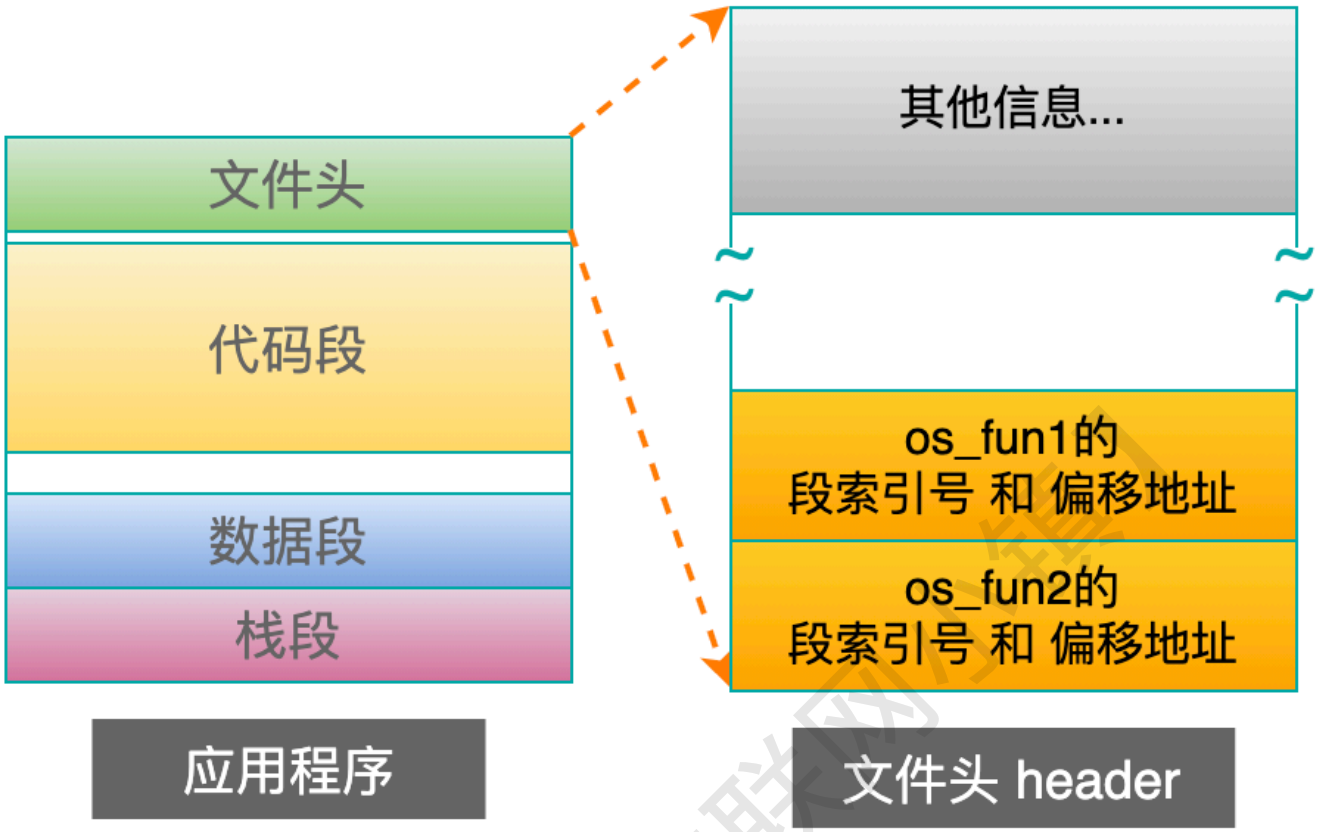
假如：有 2 个系统函数 os_func1 和 os_func2，放在一个独立的段中：



既然 OS 中多了一个代码段，那么 bootloader 就需要帮助它在 GDT 中多创建一个段描述符：



在应用程序的 header 中，预留一个足够大的空间来存放每一个系统函数的跳转信息(系统函数的段索引号和函数的偏移地址):



应用程序有了这个信息之后，当需要调用 os_func1 时，就直接跳转到相应的 段索引号:函数偏移地址，就可以调用到这个系统函数了。

这里同样的会引出 2 个问题：

> 1. 如果操作系统提供的系统函数很多，应用程序也很多，那么操作系统在加载每一个应用程序时，岂不是要忙死了？而且应用程序也不知道应该保留多大的空间来存放这些系统函数的跳转信息；
> 2. 在执行系统函数时，此时代码段、数据段都是属于操作系统的势力范围，但是栈基址和栈顶指针使用的仍然是应用程序拥有的栈，这样合理吗？

对于第一个问题，所以 Linux 中通过中断，提供一个统一的调用入口地址，然后通过一个寄存器来区分是哪一个函数。

对于第二个问题，Linux 在加载每一个应用程序时，会在内核中建立与该应用程序相关的数据结构，并且在内核中创建一块内存空间，专门用作：从这个应用程序跳转到内核中执行代码时，所使用的栈空间。

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

从 bootloader 到操作系统，再到应用程序，这个三级跳的最简流程就讨论结束了。

希望对你有小小的帮助，谢谢！

方便的话，也请你转发给身边的技术小伙伴，让我们一块进步！

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。