作 者：道哥，10+年的嵌入式开发老兵。

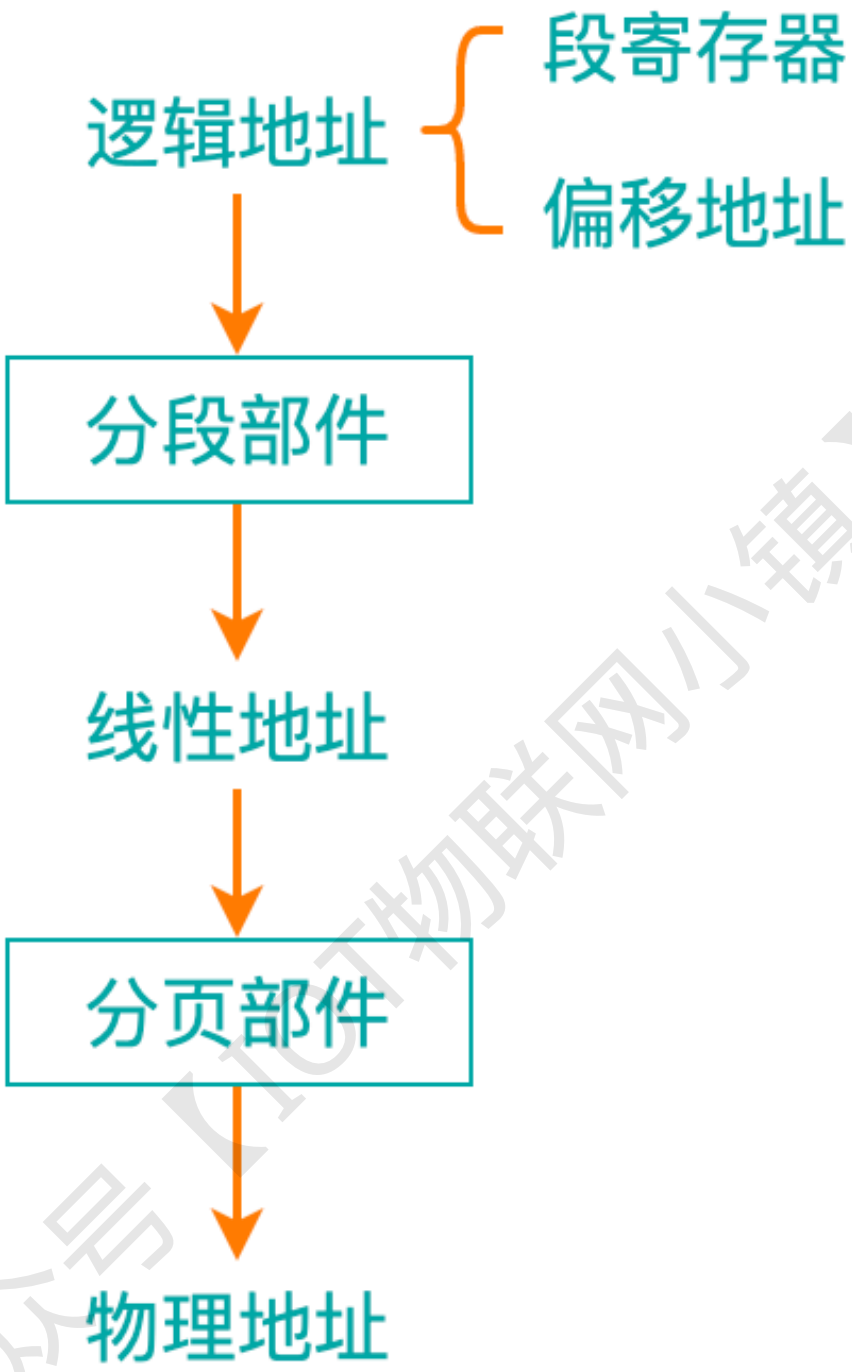
公众号：【IOT物联网小镇】，专注于：C/C++、Linux操作系统、应用程序设计、物联网、单片机和嵌入式开发等领域。 公众号回复【书籍】，获取 Linux、嵌入式领域经典书籍。

在上一篇文章中，我们已经顺利的从实模式，过渡到了保护模式。

保护模式与实模式最本质的区别就是：保护模式使用了全局描述符表，用来保存每一个程序(bootloader，操作系统，应用程序)使用到的每个段信息：开始地址，长度，以及其他一些保护参数。

这篇文章，我们来看一下 `bootloader` 是如何来进行自我进化到保护模式的，然后深入看一下保护模式是如何对内存进行安全保护的。

作为背景知识，我们先来看一下 `x86` 中的地址变换过程：



x86 处理器中的分页机制是可以被关闭的，此时线性地址就等于物理地址，这也是我们一直讨论的情况。

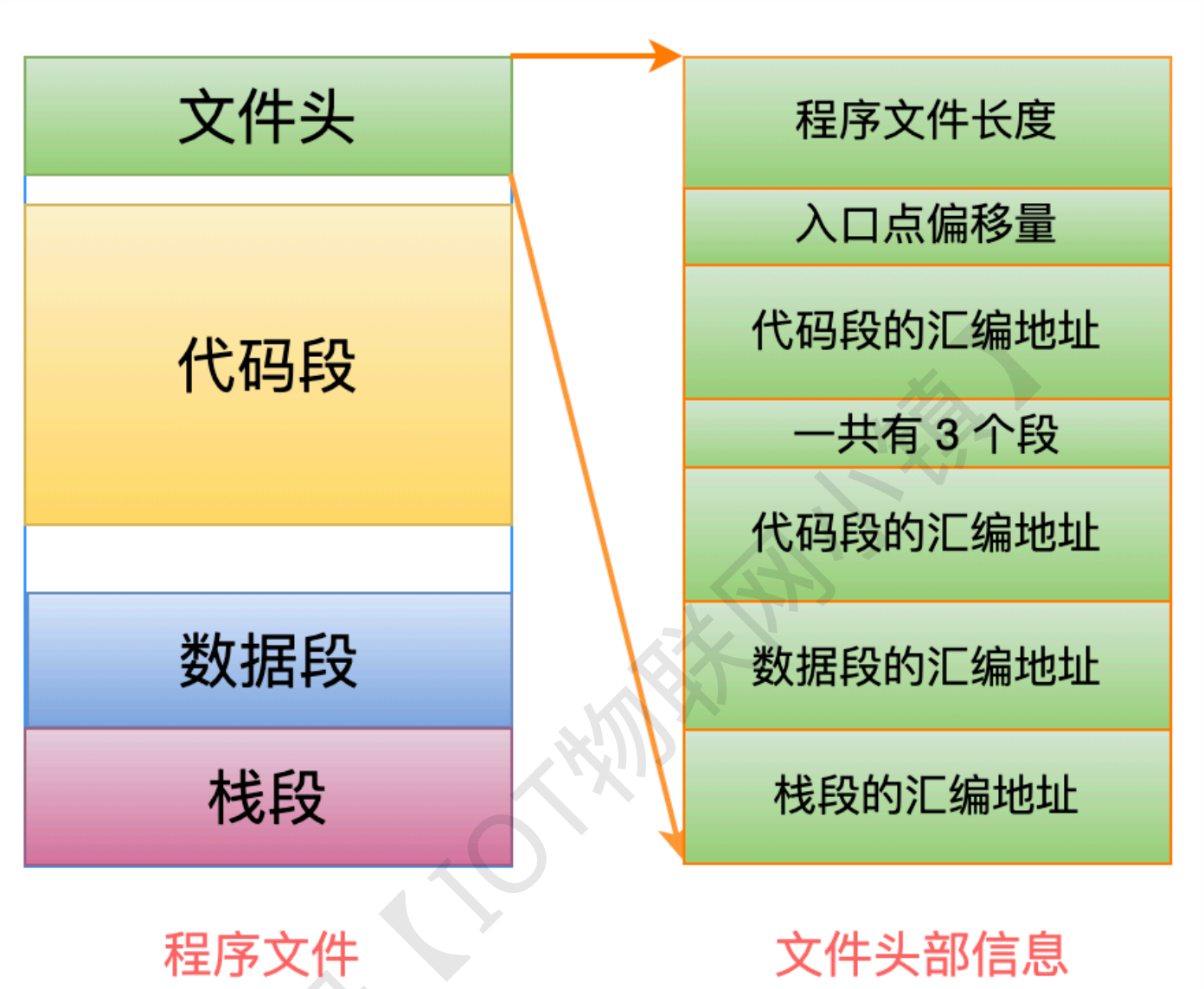
> 下一篇文章，我们就把 x86 中的分页机制打开，并与 Linux 中的分段和分页机制进行对比。

实模式：bootloader 为程序计算段的基地址

在之前的文章：Linux从头学06：16张结构图，彻底理解【代码重定位】的底层原理中，我们讨论了 `bootloader` 是如何把应用程序读取到内存中，最后跳入到程序的入口地址的。

> 这里所说的程序，可以是操作系统，也可以是应用程序。

下面这张图，是程序被加载到内存中之后，header 中的信息：

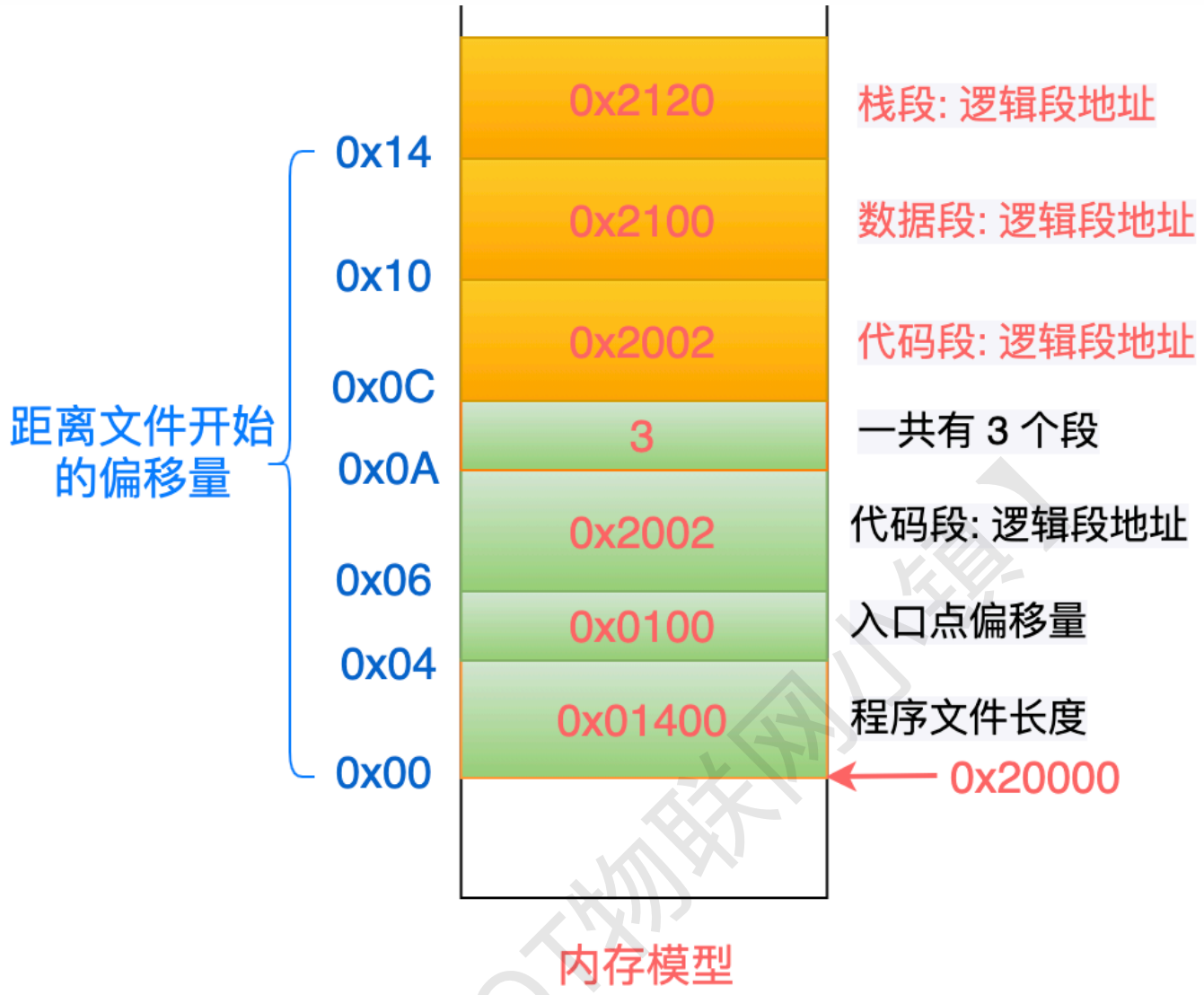


因为程序是被 bootloader 动态读取到内存中的，它是不知道自己被放在内存中的什么位置，因此它也不知道自己代码段、数据段、栈的开始地址。

但是，程序要想能够正常执行，就必须要知道这些信息，那怎么办？

只有 bootloader 才能解决问题，因为是它来把程序从硬盘加载到内存中的。

因此，bootloader 在跳入程序的入口地址之前，必须把其中的代码段、数据段、栈段的基地址计算出来，然后写入到程序的 header 中，如下图所示：



这样的话，程序开始执行时，就可以从自己的 header 中获取到这 3 个段基地址，并且赋值给相应的寄存器，从而顺利的执行程序。

也就是说：程序的 header 空间，充当了 bootloader 与它进行信息交互的媒介，用来传递 3 个段寄存器的基地址。

以上的这个过程，一直工作在实模式，因此就没有段描述符什么事情。

在以后文章中，我们还会看到在保护模式下，bootloader 仍然会利用 OS 的 header 空间，来传递段的索引号。然后 OS 利用这个段索引号，去查找 GDT 表，从而找到每一个段的基地址以及其他一些保护信息。

保护模式：bootloader 为自己创建段描述符

bootloader 从 BIOS 接管系统之后，刚开始是运行在实模式下的。

当它完成一些准备工作之后，就可以进入保护模式了，也就是把 CR0 寄存器的 bit0 设置为 1。

这个准备工作中，最重要的就是：建立 GDT 这个表，并且把 GDT 的开始地址，存储到寄存器 GDTR 中。

下面这张图，是 bootloader 被加载到内存中的布局图：

bootloader 被加载到 0x0000_7C00 地址处。

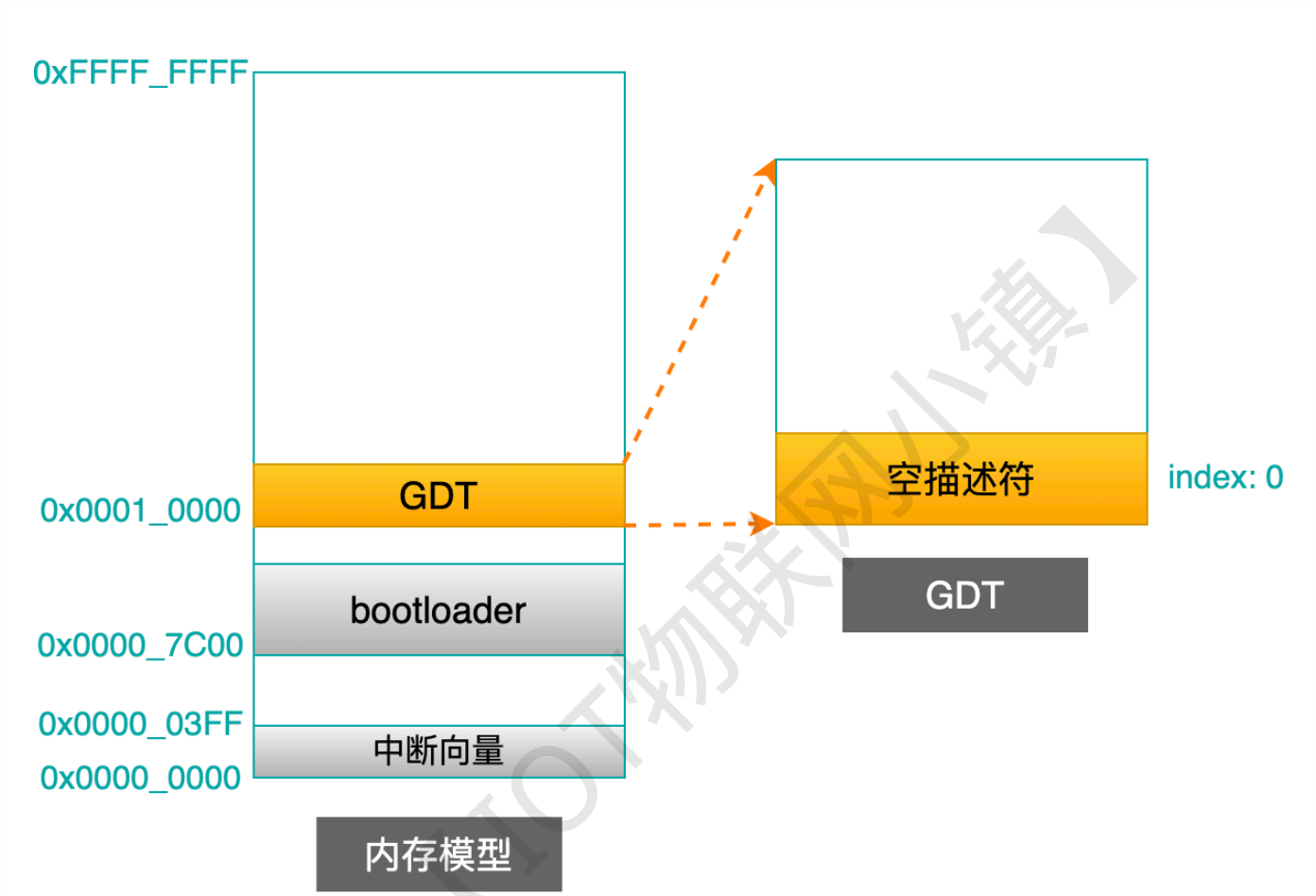
它最少需要创建 3 个段描述符：代码段、数据段和栈段。

确定 GDT 的地址

在创建段描述符之前，需要先确定: 把 GDT 表放在内存中的什么位置?

暂且就把它放在 `0x0001_0000` 这个地址吧，距离零地址 64K 的位置。

按照处理器的要求，在第 1 个表项(称之为 item 或者 entry，每本书上都不一样)必须为空描述符（index = 0）。



创建代码段描述符

`bootloader` 的代码放在 `0x0000_7C00` 开始的地址，长度是 `512B`。

根据这些信息，就可以构造出代码段的描述符了:

创建数据段描述符

bootloader 待会需要把操作系统或其他应用程序，从硬盘读取到内存中，例如：读取到 0x0002_0000 的位置。

那么 bootloader 就必须能够访问到这个位置，并且是以数据段的读写方式。

为了利用全部的 4G 内存空间，bootloader 可以把这 4G 空间，作为一个数据段来定义它的描述符，如下：

创建栈段描述符

理论上，bootloader 可以使用内存中的任意一块空闲空间，来作为自己的栈。

因为栈在 push 操作的时候，是向低地址方向增长的。

因此很多书籍都会把栈顶基地址设置为 bootloader 的开始地址，也就是 0x0000_7C00 地址处，并且把栈的空间大小限制在 4K 的范围。

根据以上这些信息，就可以创建出栈的段描述符，如下：

当以上这几个段的描述符都创建好之后，就可以把 GDT 的地址(0x0001_0000)，设置到 GDTR 寄存器中了。

最后，再把 CR0 寄存器的 bit0 设置为 1，就正式的进入保护模式来执行 bootloader 中后面的代码了。

段描述符是如何确保段的安全访问的?

段寄存器高速缓存

进入保护模式之后，虽然对段寄存器中内容的解释改变了，但是执行每一条指令，还是需要使用到这些段寄存器的: cs, ds, ss等等。

想象一下：每执行一条指令，都会从逻辑地址中，获取到段索引号，然后去查找 GDT 表，从而定位到段的基地址。

大家都知道程序有个"局部性"原理，也就是连续执行的代码，都是集中在一段连续的程序空间中的。

这个连续的程序空间，它们都是在同一个代码段中，因此段的基地址都是相同的，那么它们都属于 GDT 中同一个代码段描述符所代表的段空间。

如果每一条指令都去查表，就会影响到程序的执行效率。

所以，处理器内部就为每一个段寄存器，安排了一个高速缓存。

拿代码段寄存器 cs 来说：当执行一条指令的时候，如果它与上一条指令中的段索引号不同，才会根据新的段索引号到 GDT 中查找相应的段描述符表项。

查找到之后，就把这个表项的内容复制到 cs 寄存器的高速缓存中。

当继续执行后面的指令时，如果逻辑地址中的段索引号没有变化，处理器就直接从高速缓存中读取段描述，从而避免了查表操作，提升了系统效率。

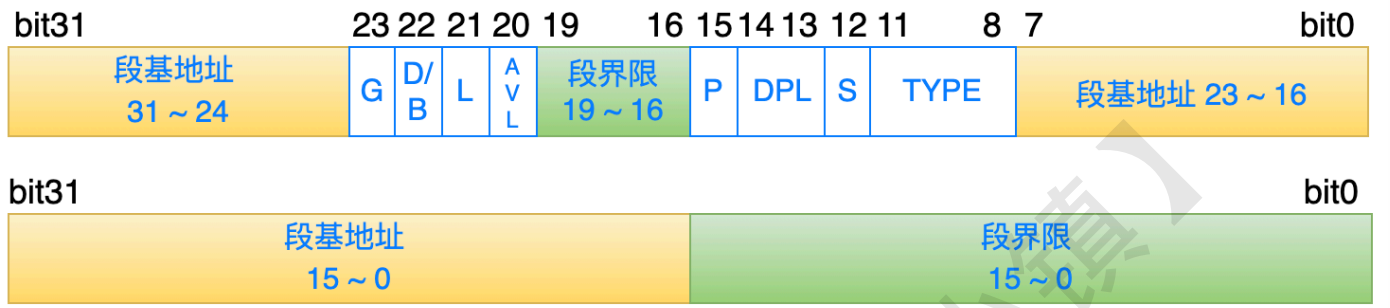
对段寄存器本身的保护

当逻辑地址中段寄存器的索引号改变时，就会根据新的索引号，到 GDT 中去查表。

> 当然了，这个索引号不能超过 GDT 的界限。

当定位到某一个描述符表项之后，就开始进行一系列检查。

再来看一下每一个段描述符 8 个字节的内容：



bit8 ~ bit11 定义了当前这个段的类型。

假如：我们在切换代码段空间的时候，不小心犯错，定位到了 GDT 中的一个数据段描述符表项，那么处理器就能够及时发现：

“当前这个段描述符的类型是数据段，你却把它当做代码段来使用，禁止，杀无赦！”

因此，处理器就会拒绝把这个段描述符复制到代码段的高速缓存中，从而对代码段寄存器进行了保护。

对段界限的检查

在通过了第一层的段类型保护之后，还会继续对段的界限进行检查，这就要使用到逻辑地址中的偏移地址（EIP）了。

如果偏移地址超过了描述符中规定的界限，那么就说明发生错误了。

例如：在 bootloader 的代码段描述符中，最大的界限是 512B，如果把 EIP 设置为 0x0000_1000，那就肯定错误了。

因为这个地址压根就不属于代码段的空间范围。

对于数据段来说比较有意思，因为我们把数据段描述符的基地址设置为 0x0000_0000，段的界限是整个 4G 的空间，所以它可以对整个内存进行操作。

多想一步：

代码段也是属于这 4G 空间，因此可以通过数据段，来改写代码段空间中的指令内容。

也就是说：如果你想修改代码段的指令，直接通过代码段来操作是不可以的。

因为代码段描述符中规定了：代码段的内容只能被读取、执行，但是不能被写入。

此时，就可以另辟蹊径：代码段也放在 4G 的空间，那么就可以通过数据段的可写特性，来改写代码段中的指令。

想一想 gdb 的调试过程，是不是就利用了这个道理？

> 在文末的推荐阅读中，就有一篇文章来介绍 gdb 的调试原理，有兴趣的小伙伴可以看一下。

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

至此，我们对保护模式下，段描述符的相关内容，就全部讨论结束了，不知道对你是否有帮助。

在准备这篇文章的时候，我特意看了一下 《深入理解 Linux 内核》这部书的第二章："内存寻址"部分的内容。

书中直接把 x86 处理器中实模式和保护模式的寻址方式作为结论告诉我们了，但是并没有具体的讲解其中的原理。

如果把之前的这几篇文章都理解了，再去看 Linux 内核的相关书籍，就不会那么吃力了。

Linux 虽然很复杂，但是它也是建立在处理器所提供的基本功能上的。

就像顶尖的乒乓球运动员许昕，打出那么多匪夷所思的神仙球，并不总是妙手偶得，而是建立在他们平时严格、机械、枯燥的日常训练，所练就的扎实的基本功。

如果没有这些坚实的基本功作为支撑，任何高级的花式技巧都只能是昙花一现。

学习也是一样！

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！

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。