

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。

转载：欢迎转载文章，转载需注明出处。

[从 16 位进入到 32 位](#)

[8086 的 16 位模式](#)

[80386 的 32 位模式](#)

[从实模式进入到保护模式](#)

[如何进入保护模式](#)

[GDT 全局段描述符表](#)

[GDTR 全局段描述符表寄存器](#)

[段寻址过程描述](#)

在之前的 7 篇文章中，我们一直学习的是最原始的 8086 处理器中的最底层的基本原理，重点是[内存的寻址方式](#)。

也就是：CPU 是如何通过[\[段地址:偏移地址\]](#)，来对内存进行寻址的。

不知道你是否发现了一个问题：

所有的程序都可以对内存中的[任意位置的数据进行读取和修改](#)，即使这个位置并不属于这个应用程序。

这是非常[危险](#)的，想一想那些心怀恶意的黑帽子黑客，如果他们想做一些坏事情，可以说是随心所欲！

面对这样的不安全行为，处理器一点办法都没有。

所以，Intel 从 80286 开始，就对增加了一个叫做[保护模式](#)的机制。

PS: 相应的，之前 8086 中的处理器执行模式就叫做“实模式”。

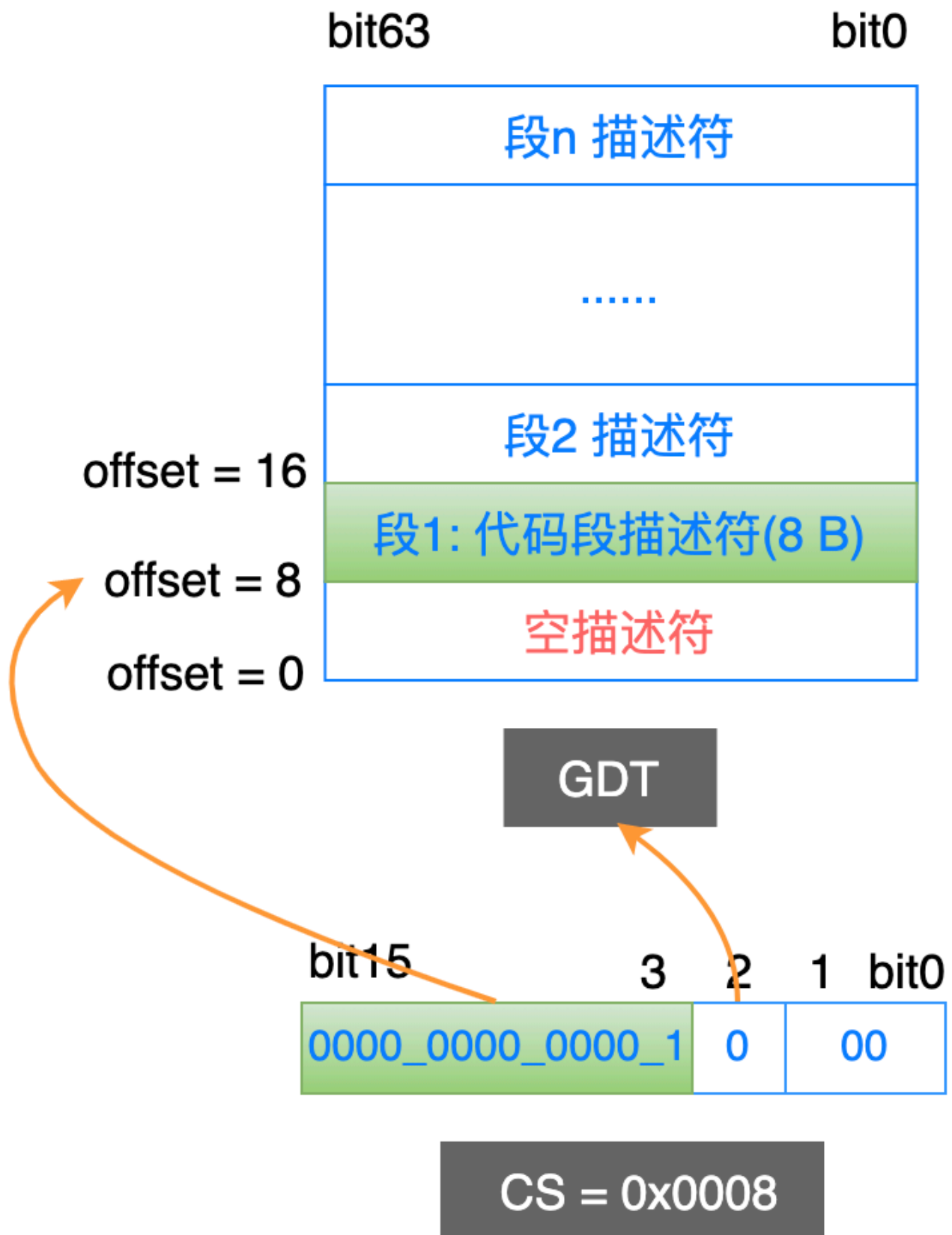
虽然 80286 没有形成一定的气候，但是它对后来的 80386 处理器提供了基础，让 386 获得了极大的成功。

这篇文章，我们就从 80386 处理器开始，聊一聊

[保护模式](#)究竟保护了谁？

底层是通过什么[机制](#)来实现保护模式的？

我们的学习目标，就是弄明白下面这张图：



从 16 位进入到 32 位

8086 的 16 位模式

在 8086 处理器中，所有的寄存器都是 16 位的。

也正因为如此，处理器为了能够得到 20 位的物理地址，需要把段寄存器的内容左移 4 位之后，再加上偏移寄存器的内容，才能得到一个 20 位的物理地址，最终访问最大 1MB 的内存空间。

例如：在访问代码段的时候，把 cs 寄存器左移 4 位，再加上 ip 寄存器，就得到 20 位的物理地址了；

20 位的地址，最大寻址范围就是 2 的 20 次方 = 1 MB 的空间；

还记得我们第 1 篇文章[Linux 从头学 01：CPU 是如何执行一条指令的？](#)中的寄存器示意图吗？

通用寄存器

AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL
	SI	
	DI	
	BP	
	SP	

段寄存器

CS
DS
ES
SS

指令指针寄存器

IP

以上这些寄存器都是 16 位的，在这种模式下，对内存的访问只能分段进行。

而且每一个段的偏移地址，最大只能到 64 KB 的范围(2 的 16 次方)。

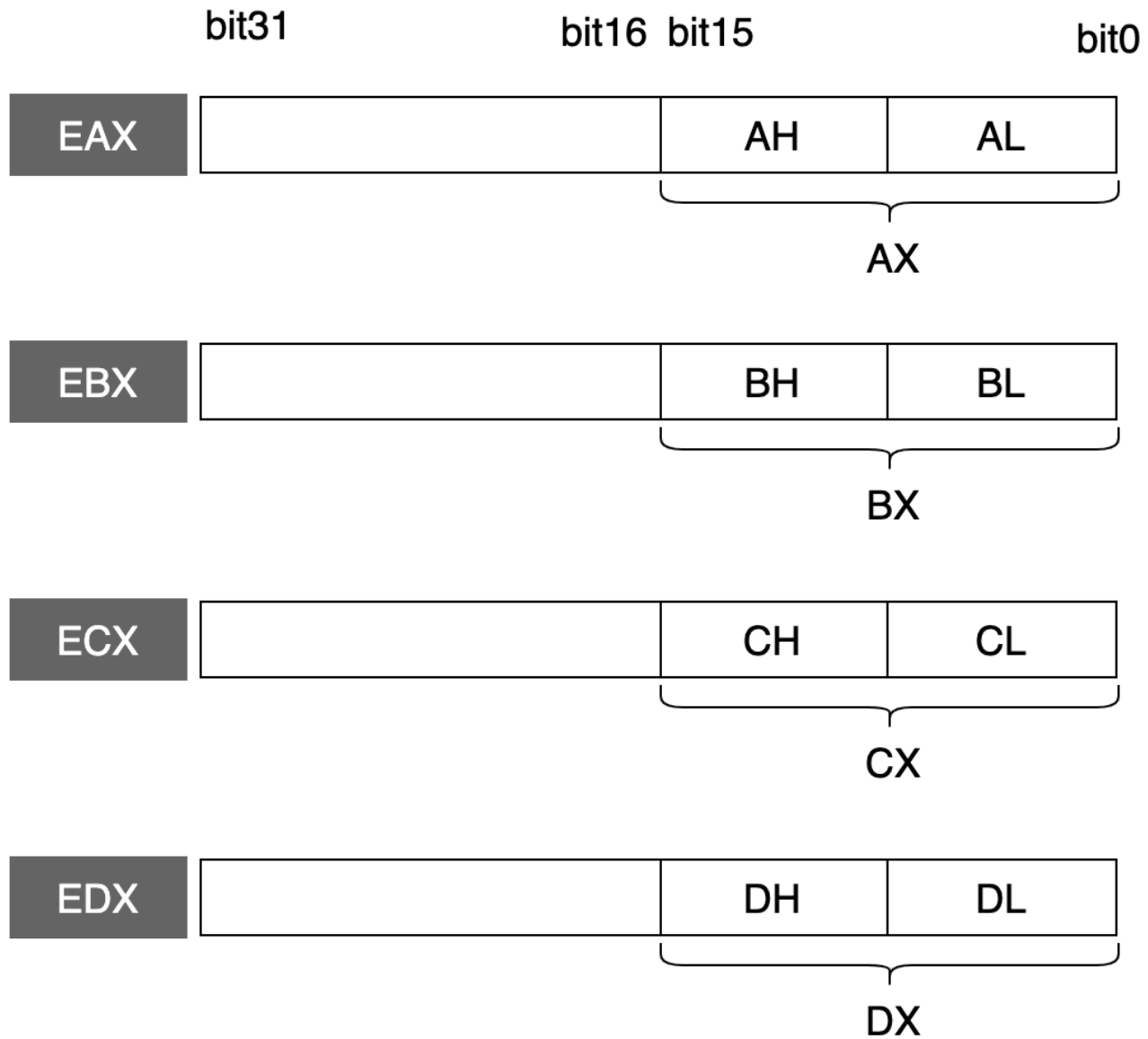
在访问代码段的时候，使用 cs:ip 寄存器；

在访问数据段的时候，使用 ds 寄存器；

在访问栈的时候，使用 ss:sp 寄存器；

80386 的 32 位模式

进入到 32 位的处理器之后，这些寄存器就扩展到 32 位了：

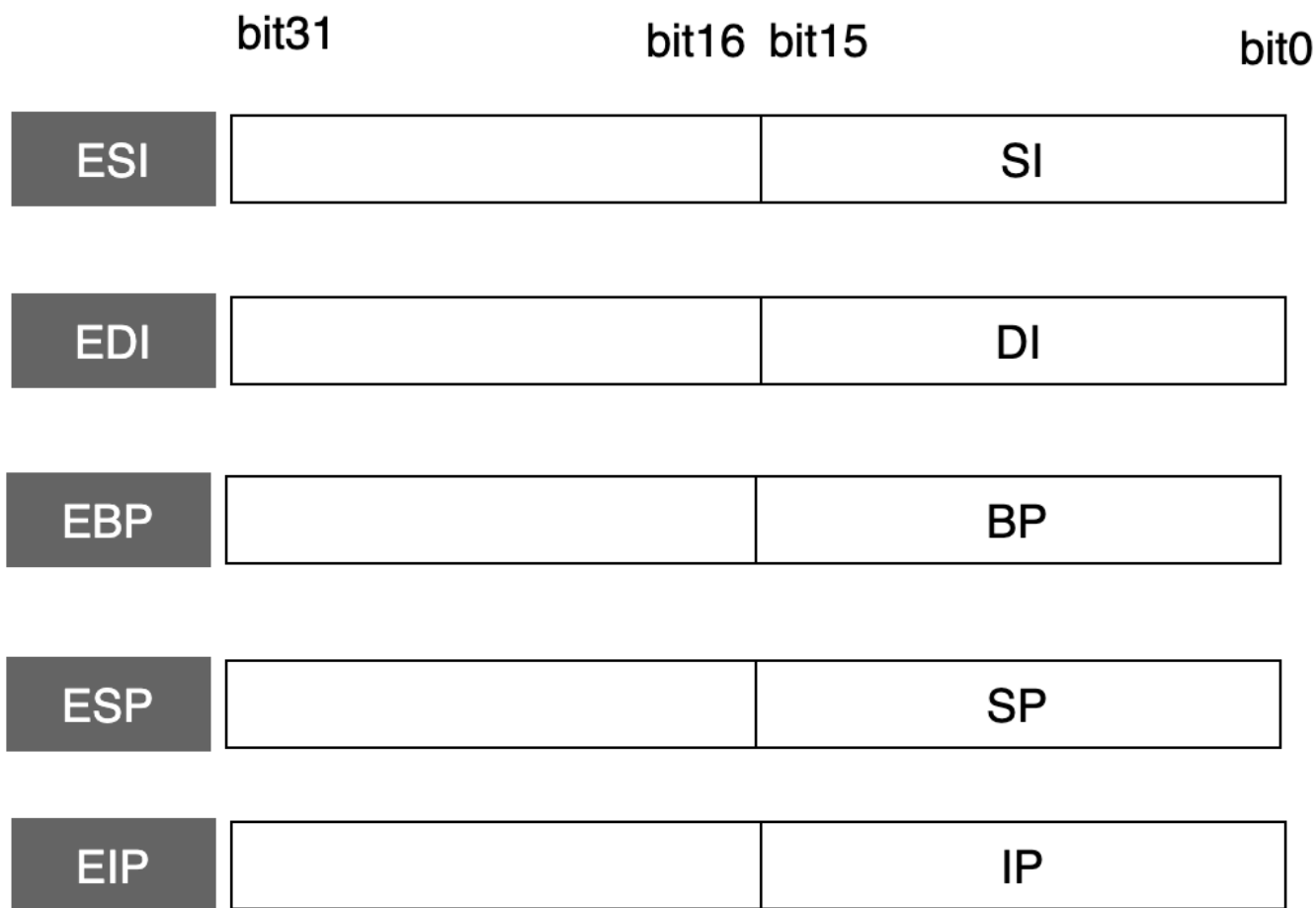


从寄存器的名称上可以出，在最前面增加了字母 E，表示 Extend 的意思。

这些 32 位的寄存器，低 16 位保持与 16 位处理器的兼容性，也就是可以使用 16 位的寄存器(例如：AX)，也可以使用 8 位的寄存器(例如：AH, AL)。

注意：高 16 位不可以独立使用。

下面这张图是 32 位处理器的另外 4 个通用寄存器(注意它们是**不能**按照 8 位寄存器来使用的)：



在 32 位的模式下，处理器中的地址线达到了 32 位，最大的内存空间可寻址能力达到 4 GB(2 的 32 次方)。

在 32 位处理器中，依然可以兼容 16 位的处理模式，此时依然使用 16 位的寄存器；

如果不兼容的话，就会失去很大的市场占有率；

是不是感觉到上面的寄存器示意图中漏掉了什么东西？

是的，图中没有展示出[段寄存器](#)(cs, ds, ss等等)。

这是因为在 32 位模式下，段寄存器依然是 16 位的长度，但是对其中内容的解释，发生了非常非常大的变化。

它们不再表示段的基地址，而是表示一个索引值以及其他信息。

通过这个[索引值\(或者叫索引号\)](#)，到一个表中去查找该段的真正[基地址](#)(有点类似于中断向量表的查找方式)：

	bit15	bit0
CS		
DS		
SS		
ES		
FS		
GS		

有些书上把描述符称之为：段选择子；

也有一些书上把段寄存器中的值称之为索引值，把段描述符在 GDT 中的偏移量称之为选择子；

不必纠结于称呼，明白其中的道理就可以了；

正是因为处理器有 32 根地址线，可寻址的范围已经非常大了(4 GB)，因此理论上它是不需要像 8086 中那样的寻址方式(段地址左移 4 位 + 偏移地址)。

公众号【IOT物联网小镇】

但是由于 x86 处理器的[基因](#)，在 32 位模式下，依然要以段为单位来访问内存。

这里请大家不要绕晕了：刚才描述的段寄存器的内容时，仅仅是说明[如何来找到一个段的基地址](#)，也即是说：

1. 对于 8086 来说，段寄存器中的内容左移 4 位之后，就是段的基地址；
2. 对于 80386 来说，段寄存器中的内容是一个表的索引号，通过这个索引号，去查找表中相应位置中的内容，这个内容中就有段的基地址(如何查找，下文有描述)；

找到了这个段的基地址之后，在访问内存的时候，[仍然是按照段机制+偏移量的方式](#)。

由于在 32 位处理器中，存储偏移地址的寄存器都是 32 位的，最大偏移地址可达 4 GB，所以，我们可以把段的[基地址](#)设置为 0x0000_0000：

最大偏移地址
0xFFFF_FFFF

内存空间

段的基地址
0x0000_0000

这样的分段方式，称作“[平坦模型](#)”，也可以理解为没有分段。

看到这里，是否联想起之前的一篇文章中，我们曾经画过一张 [Linux 操作系统中的分段模型](#)：

段	Base	G	Limit	S	Type	DPL	D/B	P
用户代码段	0x00000000	1	0xffff	1	10	3	1	1
用户数据段	0x00000000	1	0xffff	1	2	3	1	1
内核代码段	0x00000000	1	0xffff	1	10	0	1	1
内核数据段	0x00000000	1	0xffff	1	2	0	1	1

现在是不是大概就明白了：为什么这 4 个段的[基地址和段的长度](#)，都是一样的？

从实模式进入到保护模式

如何进入保护模式

CPU 是如何判断：当前是执行的是[实模式](#)？还是[保护模式](#)？

在处理器内部，有一个寄存器 CR0。这个寄存器的 bit0 位的值，就决定了当前的工作模式：



bit0 = 0: 实模式;
bit1 = 1: 保护模式;

在处理器上电之后，默认状态下是工作在实模式。

当操作系统做好进入保护模式的一切准备工作之后，就把 CR0 寄存器的 bit0 位设置为 1，此后 CPU 就开始工作在[保护模式](#)。



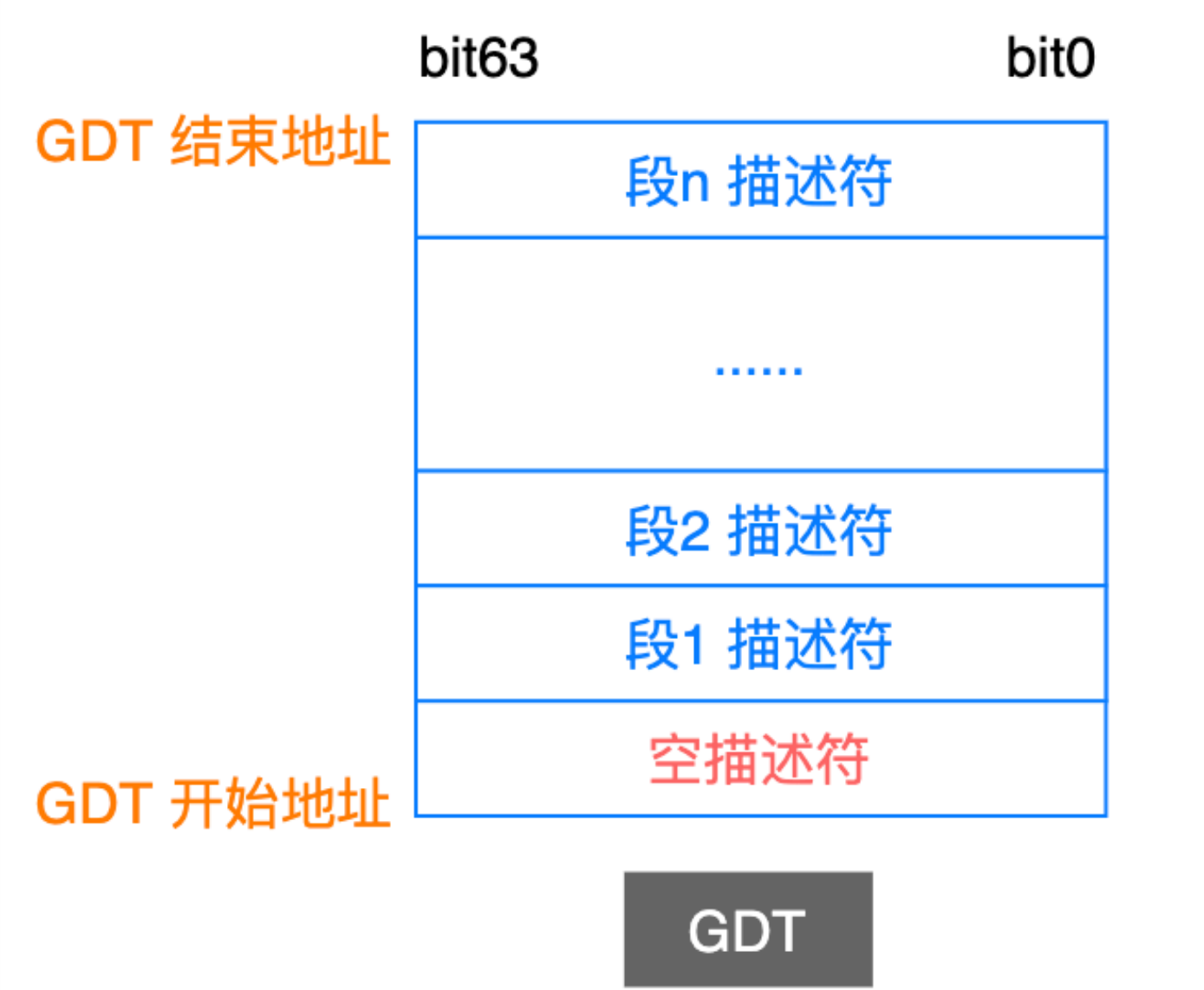
也就是说：在 bit0 设置为 1 之前，CPU 都是按照实模式下的机制来进行寻址(段地址左移 4 位 + 偏移地址)；

当 bit0 设置为 1 之后，CPU 就按照保护模式下的机制来进行寻址(通过段寄存器中的索引号，到一个表中查找段的基地址，然后再加上偏移地址)。

GDT 全局段描述符表

由于这张表中的每一个条目(Entry)，描述的是一个段的基本信息，包括：基地址、段的长度界限、安全级别等等，因此我们称之为全局描述符表(Global Descriper Table, GDT)。

之所以称之为全局的，是因为每一个应用程序还可以把段描述符信息，放在自己的一个私有的局部描述符表中(Local Descriper Table, LDT)，在以后的文章中一定会介绍到。



处理器规定：第一个描述符必须为空，主要是为了规避一些程序错误。

从上图可以看出：GDT 中每一个条目的长度是 8 个字节，其中描述了一个段的具体信息，如下所示：



黄色部分：表示这个段在内存中的地址。

绿色部分：表示这个段的长度是多少。

第一次看到这张图时，是不是心中有 2 个疑问：

1. 为什么段的地址不是用连续的 32 bit 位来表示？
2. 段的界限怎么是 20 位的？20 位只能表示 1 MB 的范围啊？

第一个问题的答案是：历史原因(兼容性)。

第二个问题的答案是：在每一个描述符中的标志位 G，对段的界限进行了进一步的粒度描述：

1. 如果 G = 0: 表示段界限是以字节为单位，此时，段界限的最大表示范围就是 1 MB;
2. 如果 G = 1: 表示段界限是以 4 KB 为单位，此时，段界限的最大表示范围就是 4 GB(1 MB 乘以 4KB);

为了完整性，我把所有标志位的含义都汇总如下，方便参考：

D/B (bit22)：用来决定数据段 or 栈段使用的偏移寄存器是 16 位 还是 32 位。

数据段 D	0	指令指针寄存器使用 16 位的 IP
	1	指令指针寄存器使用 32 位的 EIP
栈段 B	0	栈顶指针寄存器使用 16 位的 SP
	1	栈顶指针寄存器使用 32 位的 ESP

L (bit21)：在 64 位系统中才会使用，暂时先忽略。

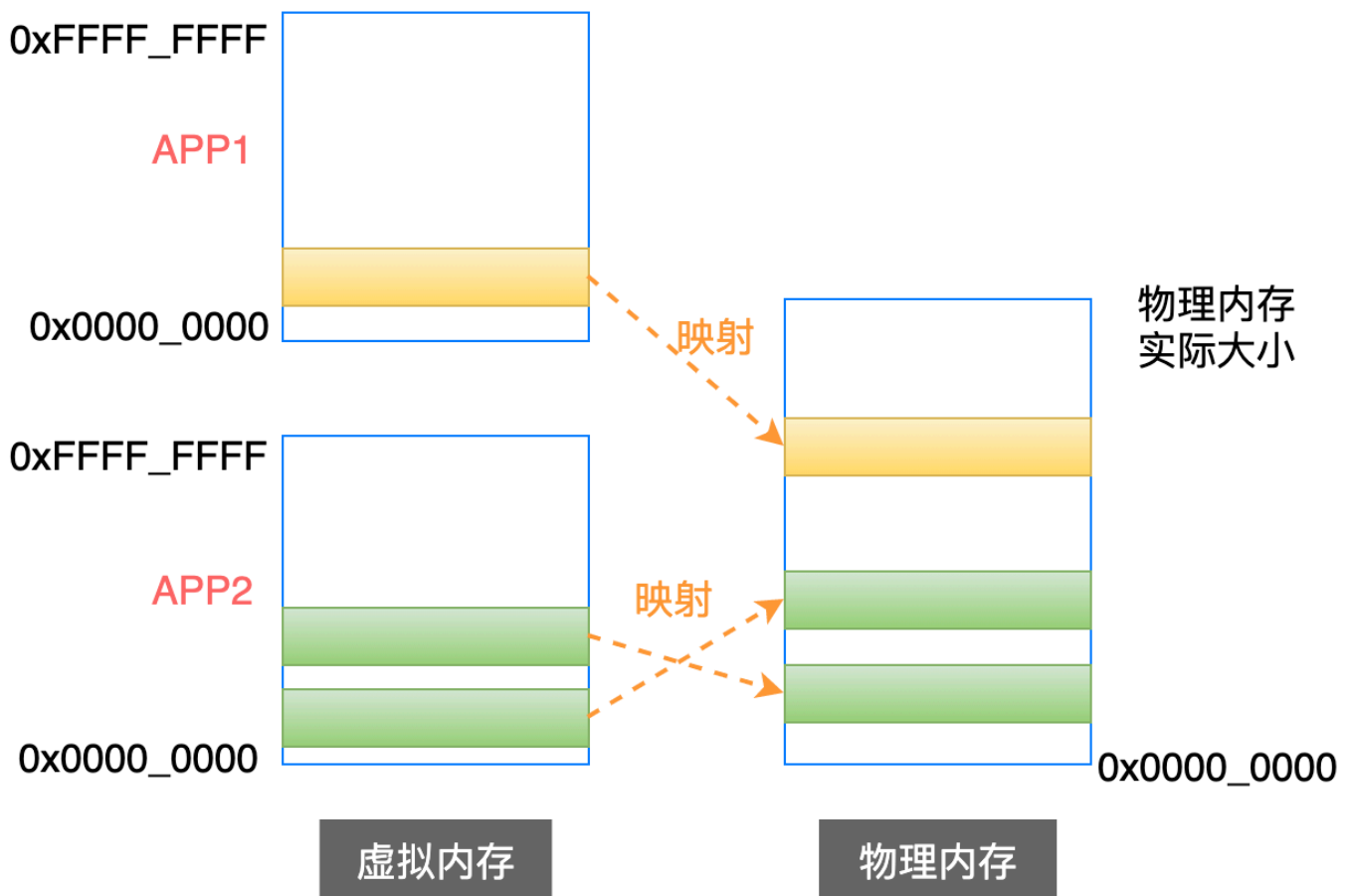
AVL (bit20)：处理器没有使用这一位内容，被操作系统可以利用这一位来做一些事情。

P (bit15)：表示这个段的内容，当前是否已经驻留在物理内存中。

存在位 P	0	这个段不在物理内存中
	1	这个段在物理内存中

在 Linux 系统中，每一个应用程序都拥有 4 GB(32位处理器) 的虚拟内存空间，而且一个系统中可以同时存在多个应用程序。

这些应用程序在虚拟内存中的代码段、数据段等等，最终都是要映射到物理内存中的。



但是物理内存的空间毕竟是有限的，当物理内存紧张的时候，操作系统就会把当前不在执行的那些段的内容，临时保存在硬盘上(此时，这个段描述符的 P 位就设置为 0)，这称之为换出。

当这个被换出的段需要执行时，处理器发现 P 位为 0，就知道段中的内容不在物理内存中，于是就在物理内存中找出一块空闲的空间，然后把硬盘中的内容复制到物理内存中，并且把 P 位设置为 1，这称之为换入。

DPL (bit14 ~ 13): 指定段的特权级别，处理器一共支持 4 个特权级别：0, 1, 2, 3(特权级别最低)。

段的 特权级别 DPL	0	最高特权级
	1	Linux 没有使用
	2	Linux 没有使用
	3	最高特权级

比如：操作系统的代码段的特权级别是 0，而一个应用程序在刚开始启动的时候，操作系统给它分配的特权级别是 3，那么这个应用程序就不能直接去转移到操作系统的代码段去执行。

在 Linux 操作系统中，只利用了 0 和 3 这两个特权级别。

S (bit12): 决定这个段的类型。

类型 S	0	这个段是一个系统段
	1	这个段是一个数据段 or 栈段

TYPE (bit11 ~ 8): 用来描述段的一些属性，例如：可读、可写、扩展方向、代码段的执行特性等等。

段的类型 TYPE	代码段	X	0	XXX
			1	可执行
		C	0	不允许从低特权级直接进入到高特权级
			1	允许从低特权级直接进入到高特权级
		R	0	段内容不可以被读出
			1	段内容可以被读出
	数据段	A	0	段最近没有被访问过
			1	段最近被访问过
		X	0	不可执行
			1	XXX
		E	0	向高地址方向扩展，普通数据段
			1	向低地址方向扩展，栈段
		W	0	段不允许写入
			1	段允许写入
		A	0	段最近没有被访问过
			1	段最近被访问过

这里的依从属性不太好理解，它主要用于决定：从一个低特权级别的代码，是否可以进入另一个高特权级别的代码。

如果可以进入，那么当前任务的请求级别 RPL 是否发生改变(以后会讨论这个问题)。

另外，操作系统可以把 A 标志，加入到物理内存的换出换入计算策略中。

这样的话，就可以避免把最近频繁访问的物理内存换出，达到更好的系统性能。

GDTR 全局段描述符表寄存器

还有一个问题需要处理：GDT 表本身也是数据，也是需要存放在内存中的。

那么：它存放在内存中的什么位置呢？CPU 又怎么能知道这个起始位置呢？

在处理器的内部，有一个寄存器：GDTR (GDT Register)，其中存储了两个信息：

bit47

16 15

bit0

GDT 起始地址

GDT 边界

我们可以从上一篇文章[Linux从头学07：【中断】那么重要，它的本质到底是什么？](#)中，中断向量表的安装过程中进行类比：

1. 程序代码把每一个中断的处理程序地址，放在中断向量表中的对应位置；
2. 中断向量表的起始地址放在内存的 0 地址处；

也就是说：处理器是到固定的地址 0 处，查找中断向量表的，这是一个[固定](#)的地址。

而对于 GDT 表而言，它的起始地址[不是固定](#)的，而是可以放在内存中的任意位置。

只要把这个位置存放到[寄存器 GDTR](#) 中，处理器在需要的时候就可以通过 GDTR 来定位到 GDT 的起始地址。

其实，GDT 在上电刚开始的时候，也不能放在内存中的任意位置。

因为在进入保护模式之前，处理器还是工作在实模式，只能寻址 1 MB 的内存空间，因此，GDT 只能放在 1 MB 内的地址空间中。

在进入保护模式之后，能寻址更大的地址空间了，此时就可以重新把 GDT 放在更大的地址空间中了，然后把这个新的起始地址，存储到 GDTR 寄存器中。

从 GDTR 寄存器中的内容可以看出，它不仅存储了 GDT 的起始地址，而且还限制了 GDT 的长度。

这个长度一共是 16 位，最大值是 64 KB(2 的 16 次方)，而一个段描述符信息是 8 B，那么 64 KB 的空间，最多一共可以存放 8192 个描述符。

这个数字，对于操作系统或者是一般的应用程序来说，是绰绰有余了。

段寻址过程描述

在上面的[段寄存器](#)示意图中，我们只说明了段寄存器依然是 16 位的。

在保护模式下，对其中内容的解释，与实模式下是大不相同的。

我们以代码段寄存器 CS 为例：



代码段寄存器 CS

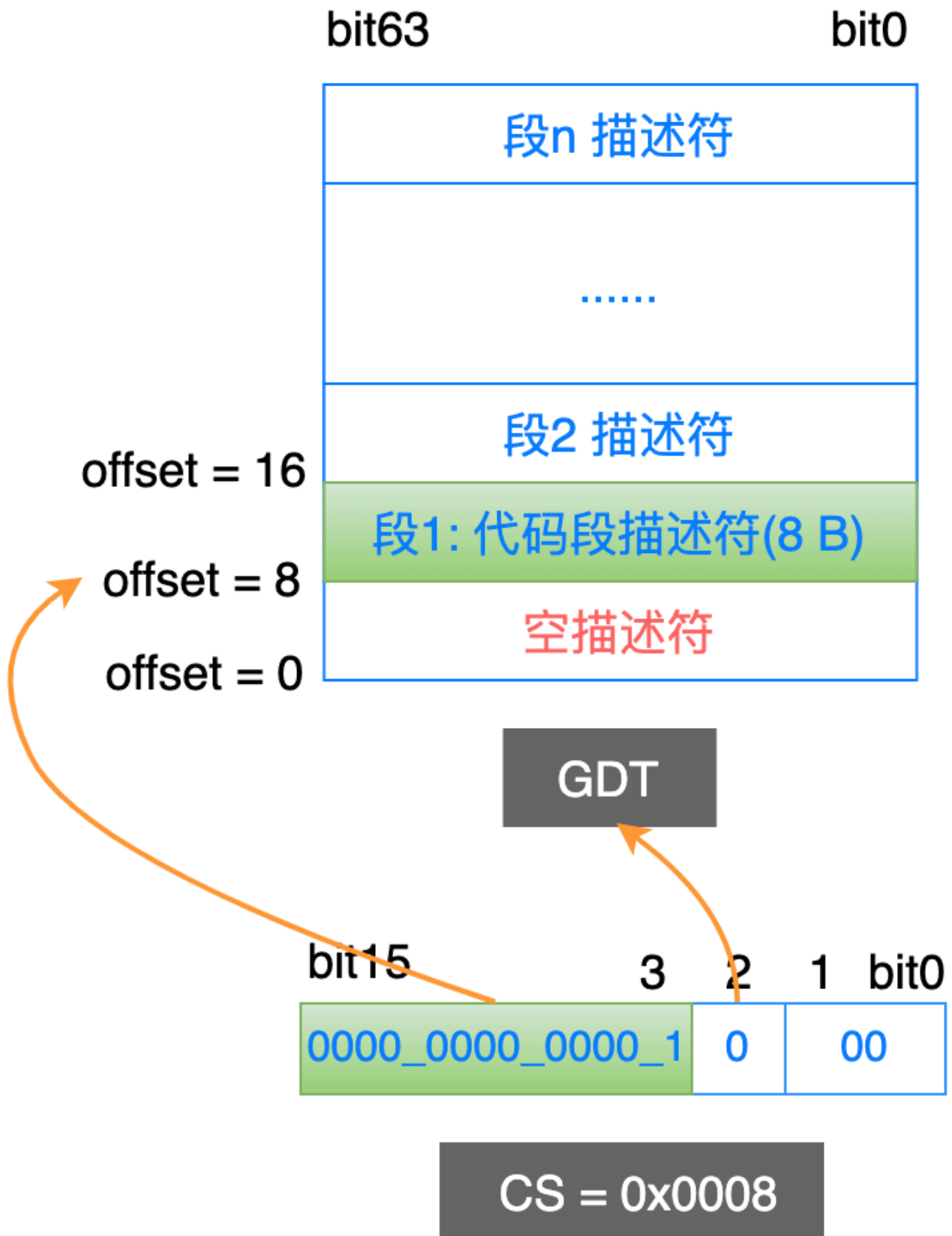
RPL: 表示当前正在执行的这个代码段的请求特权级;

TI: 表示到哪一个表中去找这个段的描述信息: 全局描述符表(GDT) or 局部描述符表(LDT)?

TI = 0 时, 到 GDT 中找段描述符;

TI = 1 时, 到 LDT 中找段描述符;

假设当前代码段寄存器 cs 的值为 0x0008, 处理器按照保护模式的机制来解释其中的内容:



1. TI = 0, 表示到 GDT 中查找段描述符;
2. RPL = 0, 表示请求特权级别是 0;
3. 描述符索引是 1, 表示这个段描述符在 GDT 中的第 1 个条目中。由于每一个描述符占用 8 个字节, 因此这

公众号【IOT物联网小镇】

个描述符的开始地址位于 GDT 中的偏移地址为 8 的位置($1 * 8 = 8$);

找到了这个[段描述符条目](#)之后，就可以从中获取到这个代码段的具体信息了：

1. 代码段的基地址在内存中什么位置;
2. 代码段的最大长度是多少(在获取指令时，如果偏移地址超过这个长度，就引发异常);
3. 代码段的特权级别是多少，当前是否驻留在物理内存中等等;

另外，从上文描述的 GDTR 寄存内容知道，它限制了 GDT 中最多一共可以存放 8192 个描述符。



我们再从[代码段](#)寄存器中，描述符索引字段所占据的 13 个 bit 位可以计算出，最多可以查找 8192 个段描述符。

2 的 13 次方 = 8192。

至此，处理器就在保护模式下，查找到了一个段的所有信息。

下面步骤就是：到这个段所在的内存空间中，执行其中的代码，或者读写其中的数据。

下一篇文章我们继续。。。

----- End -----

这篇文章主要描述了 80386 处理器中的[保护模式](#)下，段寄存器的使用，以及通过段描述符来查找段的具体信息。

从描述的内容来看，已经和我们的最终目标：Linux 操作系统中的执行方式，越来越接近了！

因为这些[底层](#)知识，都是 Linux 操作系统赖以运行的基础。

理解了这些基础内容，后面在学习 Linux 的具体模块时，就可以回过头来查一下它在处理器层面的底层支撑。

最后，如果这篇文章对您有一点帮助，请[转发给身边的技术小伙伴](#)，也是对我继续输出文章的最大鼓励和动力！

让我们一起出发，向着目标继续迈进！

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](#)

公众号【IOT物联网小镇】



微信搜一搜

Q IOT物联网小镇

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。