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。

饭是一口一口的吃，计算机也是一步一步的发展，例如下面这张英特尔公司的 CPU 型号历史：

为了利用性能越来越强悍的计算机，操作系统的也是在逐步变得膨胀和复杂。

为了从最底层来学习操作系统的一些基本原理，我们只有抛开操作系统的外衣，从最原始的硬件和编程方式来入手，才能了解到一些根本的知识。

这篇文章我们就来继续挖掘一下，8086 这个开天辟地的处理器中，是如何利用段机制来对内存进行寻址的。

什么是代码段?

在上一篇文章：Linux 从头学 01：CPU 是如何执行一条指令的？ 中，已经提到过，在处理器的内部，执行每一条指令码时，CPU 是非常机械、非常单纯地从 CS:IP 这 2 个寄存器计算得到转换后的物理地址，从这个物理地址所指向的内存地址处，读取一定长度的指令，然后交给逻辑运算单元(Arithmetic Logic Unit, ALU)去执行。

> 物理地址的计算方式是：CS * 16 + IP。

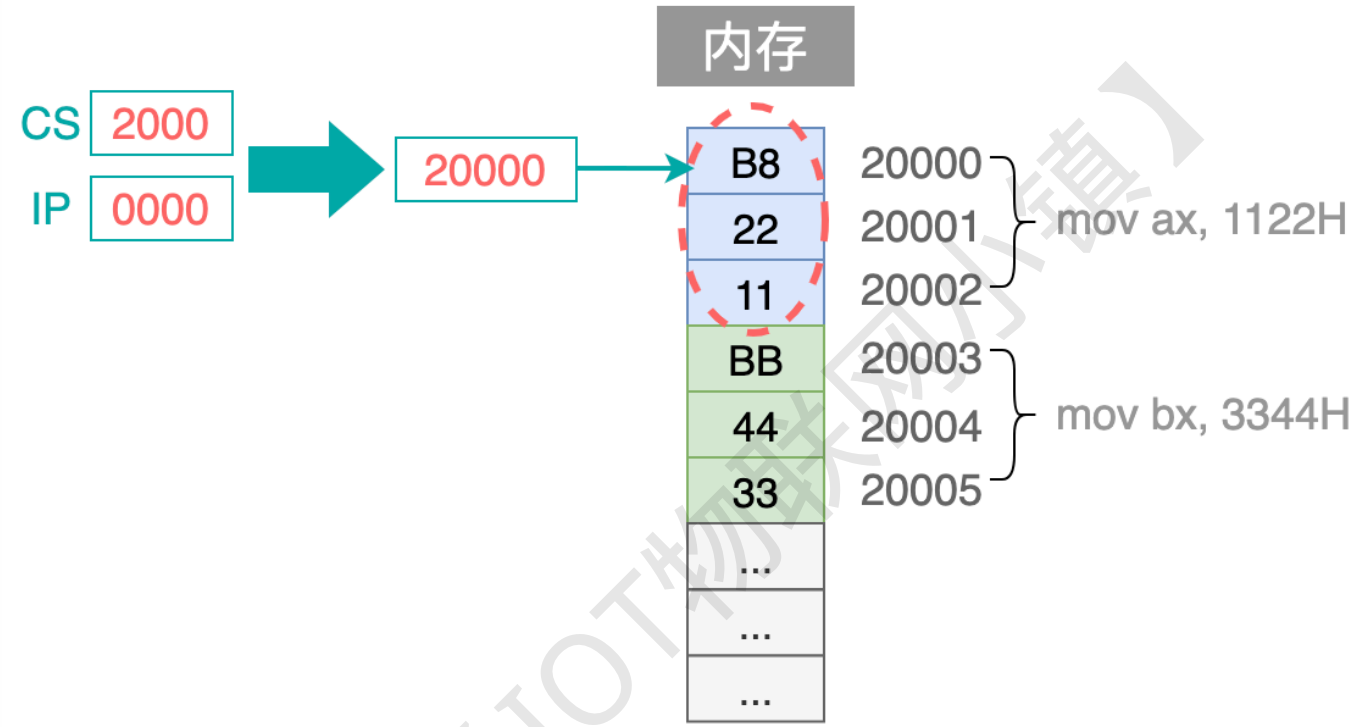
当 CPU 读取一条指令后，根据指令操作码它能够自动知道这条指令一共需要读取多少个字节。

指令被读取之后，IP 寄存器中的内容就会自增，指向内存中下一条指令的地址。

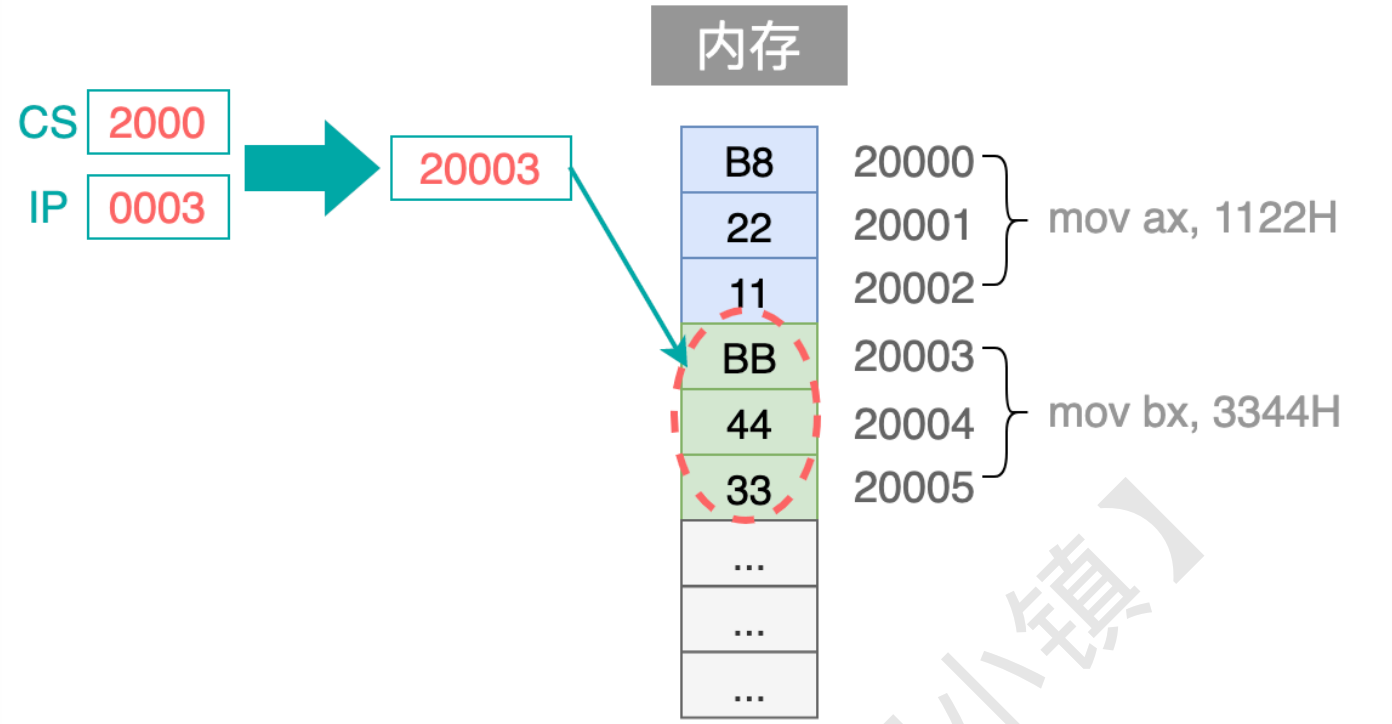
例如，在内存 20000H 开始的地方，存在 2 条指令：

```
mov ax, 1122H
mov bx, 3344H
```

当执行第一条指令时，CS = 2000H，IP = 0000H，经过地址转换之后的物理地址是：2000H * 16 + 0000 = 20000H(乘以 16 也就表示十六进制的数左移 1 位)：



当第一条指令码 B8 22 11 这 3 个字节被读取之后，IP 寄存器中的内容自动增加3`，从而指向下一条指令：



当第二条指令码 BB 44 33 这 3 个字节被读取之后，IP 寄存器中的内容又增加 3，变为 0006H。

正如上篇文章所写，CPU 只是反复的从 CS:IP 指向的内存地址中读取指令码、执行指令，再读取指令码、再执行指令。

可以看出，要完成一个有意义的工作，所有的指令码必须集中在一起，统一放在内存中某个确定的地址空间中，才能被 CPU 依次的读取、执行。

内存中的这块地址空间就叫做一个段，又因为这个段中存储的是代码编译得到的指令，因此又称作代码段。

因此，用来对代码段进行寻址的这两个寄存器 CS 和 IP，它们的含义就非常清楚了：

> CS: 段寄存器，其中的值左移 1 位之后，得到的值就表示代码段在内存中的首地址，或者称作基地址;
>
> IP: 指令指针寄存器，表示一条指令的地址，距离基地址的偏移量，也就是说，IP 寄存器是用来帮助 CPU 记住：哪些指令已经被处理过了，下一个要被处理的指令是哪一个;

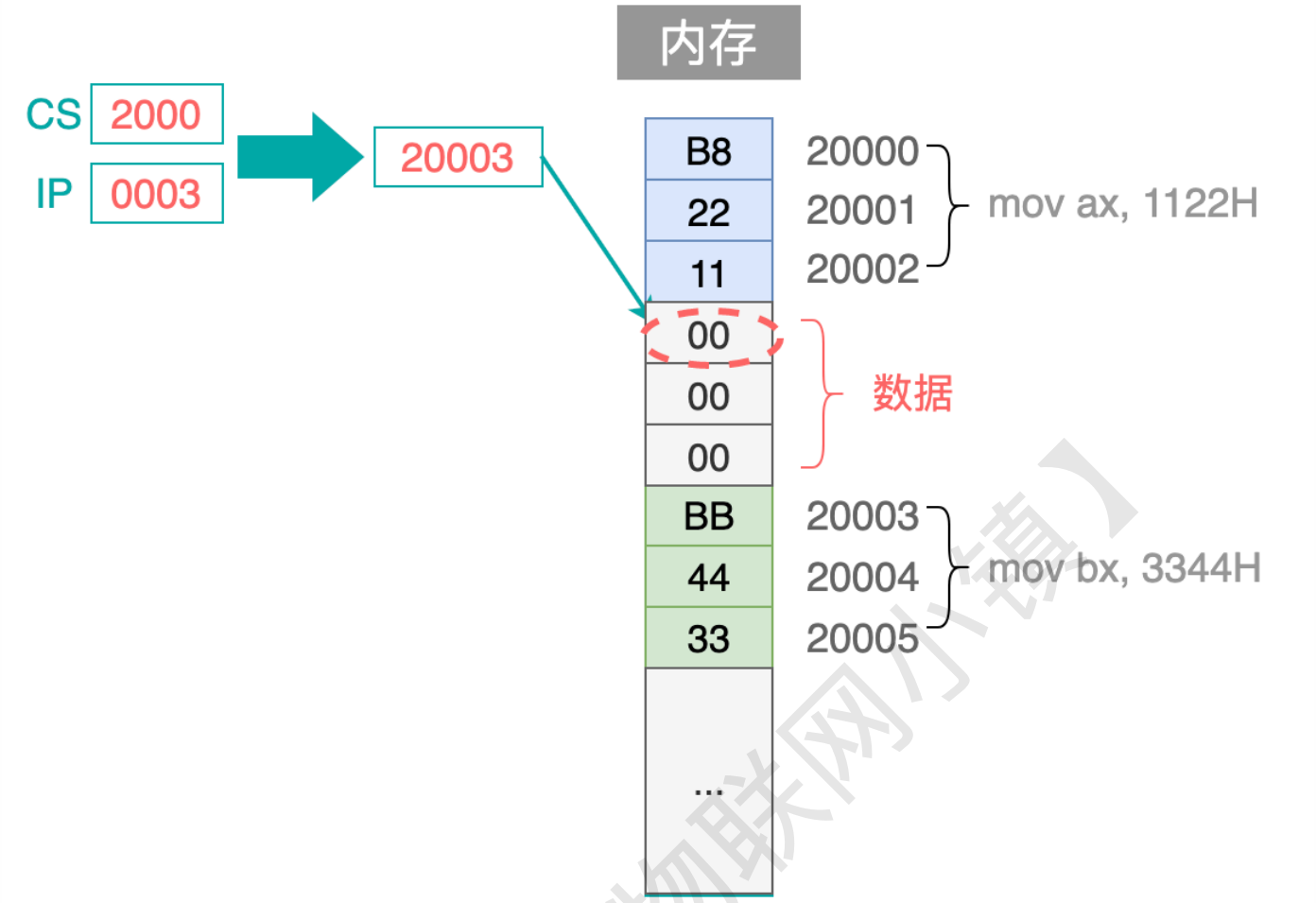
什么是数据段?

作为一个有意义的程序，仅仅只有指令是不够的，还必须操作数据。

这些数据也应该集中放在一起，位于内存中的某个地址空间中，这块地址空间，也是一个段，称作数据段。

也就是说：代码段和数据段，就是内存中的两个地址空间，其中分别存储了指令和数据。

可以想象一下：假如指令和数据不是分开存放的，而是夹杂放在一起，那么 CPU 在读取一条指令时，肯定就会把数据当做指令来读取、执行，就像下面这样，不发生错误才怪呢！



CPU 对内存中数据段的访问方式，与访问代码段是类似的，也是通过一个基地址，再加上一个偏移量来得到数据段中的某个物理地址。

在 8086 处理其中，数据段的段寄存器是 DS，也就是说，当 CPU 执行一条指令，这条指令需要访问数据段时，就会把 DS 这个数据段寄存器中的值左移 1 位之后得到的地址，当做数据段的基地址。

遗憾的是，CPU 中并没有提供一个类似 IP 寄存器的其他寄存器，来表示数据段的偏移地址寄存器。

这其实并不是坏事，因为一个程序在处理数据时，需要对数据进行什么样操作，程序的开发者是最清楚的，因此我们就可以用更灵活的方式来告诉 CPU 应该如何计算数据的偏移地址。

就像猴子掰苞米一样，不需要按照顺序来掰，想掰哪个就掰哪个。同样的，程序在操作数据时，无论操作哪一个数据，直接给出该数据的偏移地址的值就可以了。

数据的类型和长度

但是，在操作数据段中每一个数据，有一个比较重要的概念需要时刻铭记：数据的类型是什么，这个数据在内存中占据的字节数是多少。

我们在高级语言编程中(eg: C 语言)，在定义一个变量的时候，必须明确这个变量的类型是什么。一旦类型确定了，那么它在被加载到内存中之后，所占据的空间大小也就确定了。

比如下面这张图:

假设 30000H 是数据段的基地址(也就意味着 DS 寄存器中的内容是 3000H)，那么 30000H 地址处的数据大小是多少：11H？2211H？还是 44332211H？

这几个都有可能，因为没有确定数据的类型！

我们知道，在 C 语言中，假如有一个指针 ptr最终指向了这里的 30000H 物理地址处(C 代码中的 ptr 是虚拟地址，经过地址转换之后执行这里的 30000H 物理地址)。

如果 ptr 定义成：

```
char *ptr;
```

那么可以说 ptr 指针指向的数值是 11H。

如果 ptr 定义成：

```
int *ptrt;
```

就可以说 ptr 指针指向的数值就是 44332211H(假设是小端格式)。

也就是说，指针 ptr 指向的数据，取决于定义指针变量时的类型。

这是高级语言中的情况，那么在汇编语言中呢？

> PS: 之前我曾说过，文章的主要目的是学习 Linux 操作系统，但是为了学习一些相对底层的内容，在开始阶段必须抛开操作系统的外衣，进入到硬件最近的地方去看。
>
> 但是该怎么看呢？还是要借助一些原始的手段和工具，那么汇编代码无疑就是最好的、也是唯一的手段;
>
> 不过，涉及到的汇编代码都是最简单的，仅仅是为了说明原理;

在汇编语言中，CPU 是通过指令码中的相关寄存器来判断操作数据的长度。

在上一篇文章中说过，相对于寄存器来说，CPU 操作内存的速度是很慢的。

因此，CPU 在对数据段中的数据进行处理的时候，一般都是先把原始数据读取到通用寄存器中(比如：ax, bx, cx dx)，然后进行计算。

得到计算结果之后，再把结果写回到内存的数据段中(如果需要的话)。

那么 CPU 在读写数据时，就根据指令码中使用的寄存器，来决定读写数据的长度。例如:

```
mov ax, [0]
```

其中的 [0] 表示内存的数据段中偏移地址是 0 的位置。

CPU 在执行这条指令的时候，就会到 30000H(假设此时数据段寄存器 DS 的值为 3000H) 这个物理地址处，取出 2 个字节的数据，放到通用寄存器 ax 中，此时 ax 寄存器中的值就是 2211H。

为什么取出 2 个字节？因为 ax 寄存器的长度是 16 位，就是 2 个字节。

那如果只想取 1 个字节，该怎么办?

16 位的通用寄存器 ax 可以拆成 2 个 8 位的寄存器里使用：ah 和 al。

```
mov al, [0]
```

因为指令码中的 al 寄存器是 8 位，因此 CPU 就只读取 30000H 处的一个字节 11，放到 al 寄存器中。(此时 ax 寄存器的高 8 位，也就是 ah 中的值保持不变)

那如果想取 3 个字节或 4 个字节怎么办?

作为相当古老的处理器，8086 CPU 中是 16 位的，只能对 8 位或 16 位的数据进行操作。

寻址范围

从以上内容可以总结得出：

1. 代码段和数据段都是通过 【基地址 + 偏移地址】的方式进行寻址;
2. 基地址都放在各自的段寄存器中，CPU 会自动把段寄存器的值，左移 1 位之后，作为段的基地址;
3. 偏移地址决定了段中的每一个具体的地址，最大偏移地址是 16 个 bit1，也即是 64KB 的空间;

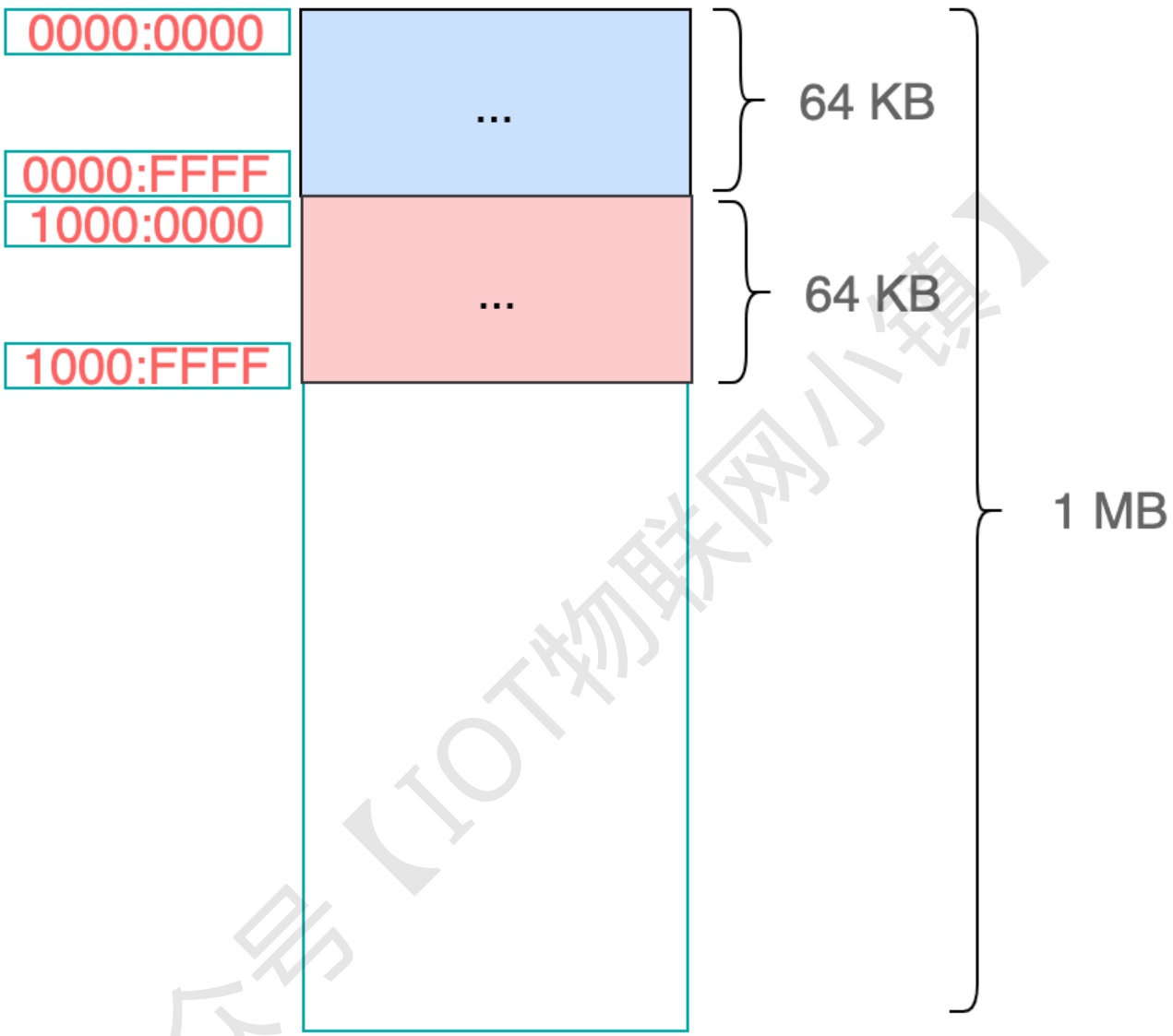
注意：这里的段寄存器左移 1 位，是指十六进制的左移，相当于是乘以 16，因此段的基地址都是 16 的倍数。

我们再来看一下这里的 64 KB 空间，与 20 根地址线有什么瓜葛。

上篇文章说到：8086 处理器有 20 根地址线，一共可以表示 1MB 的内存空间，即使给它更大的空间，它也没有福气去享受，因为寻址不到大于 1 MB 的地址空间啊！

这 1MB 的内存空间，就可以分割为很多个段。

例如：第 1 个段的地址范围是：



我们来计算最后一个段的空间。

段寄存器和偏移地址都取最大值，就是 FFFF:FFFF，先偏移再相加：FFFF0 + FFFF = 10FFEF =1M + 64K - 16Bytes。

超过了 1 MB 的空间大小，但是毕竟只有 20 根地址线，肯定是无法寻址超过 1 MB 地址空间的，因此系统会采取回绕的方式来定位到一个地址空间，类似与数学中的取模操作。

此外还有一点，在表示一个内存地址的时候，一般不会直接给出物理地址的值(比如：3000A)，而是使用 段地址:偏移地址 这样的形式来表示(比如：3000:000A)。

栈

栈也是数据空间的一种，只不过它的操作方式有些特殊而已。

栈的操作方式就是 4 个字：后进先出。

在上面介绍数据段的时候，我们都是在指令码中手动对数据的偏移地址进行设置，指哪打哪，因为这些数据放在什么位置、表示什么意思、怎么来使用，开发者自己心里最门清。

但是栈有些不一样，虽然它的功能也是用来存储数据的，但是操作栈的方式，是由处理器提供的一些专门的指令来操作的：push 和 pop。

> push(入栈): 往栈空间中放入一个数据;
> pop(出栈): 从栈空间中弹出一个数据;

注意：这里的数据是固定 2 个字节，也就是一个字。

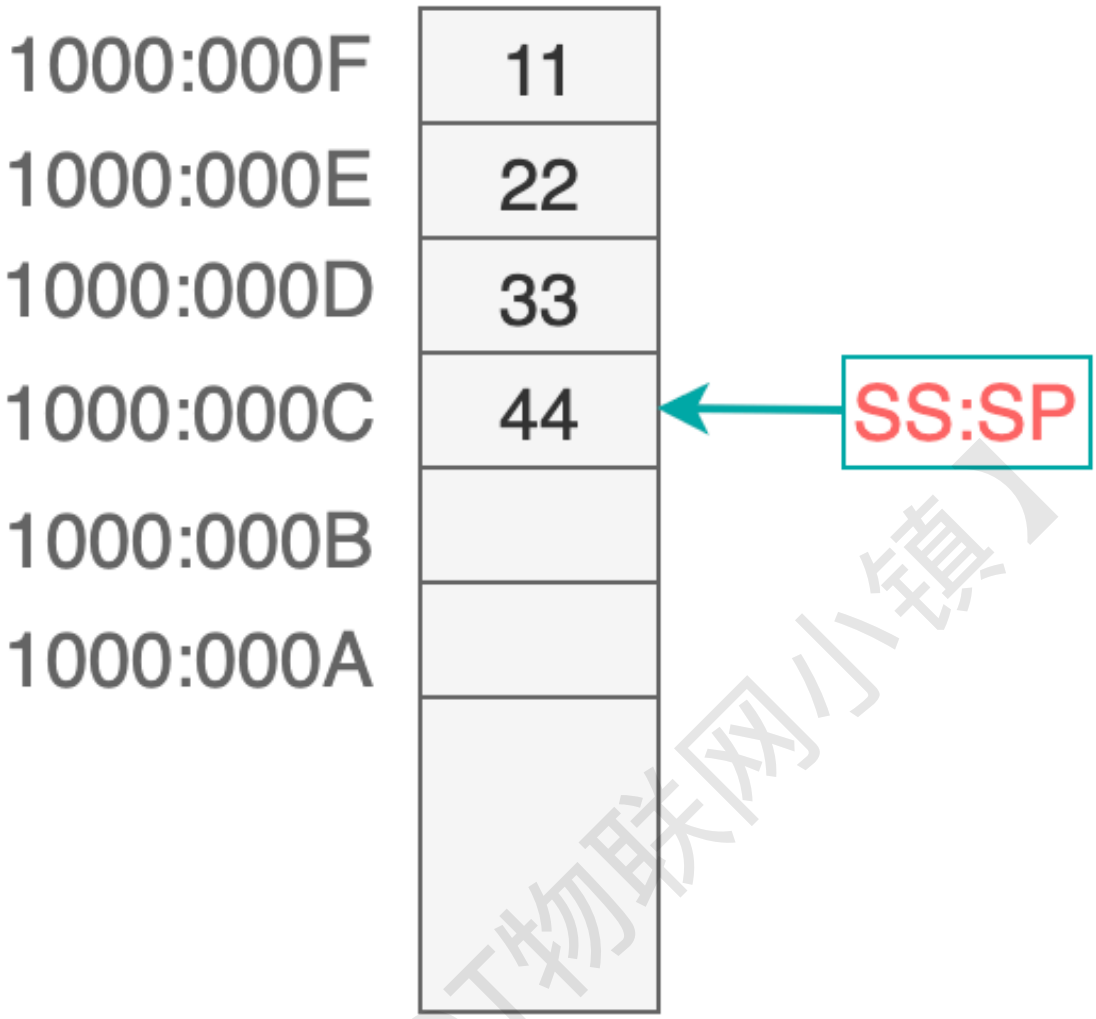
写过 C/C++ 程序的小伙伴都知道：在函数调用的时候，存在入栈操作；在函数返回的时候，存在出栈操作。

既然栈也是指一块内存空间，那么也就是表现为内存中的一个段。

既然是一个段，那肯定就存在一个段寄存器，用来代表它的基地址，这个栈的段寄存器就是 SS。

此外，由于栈在入栈和出栈的时候，是按照连续的地址顺序操作的，因此处理器为栈也提供了一个偏移地址寄存器：SP(称作：栈顶指针)，指向栈空间中最顶上的那个元素的位置。

例如下面这张图：

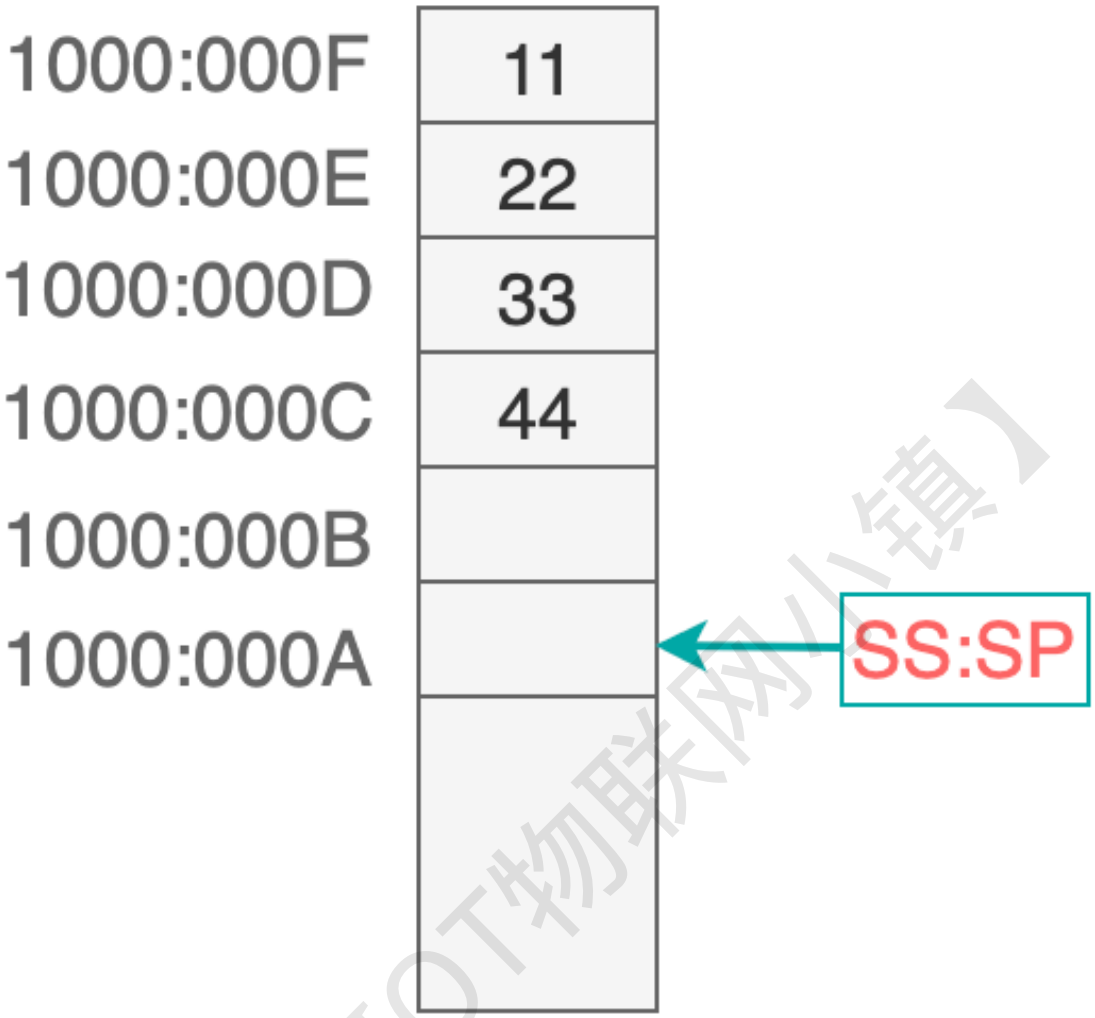


栈空间的基地址是 1000:0000，SS:SP 执行的地址空间是栈顶，此时栈顶中的元素是 44。

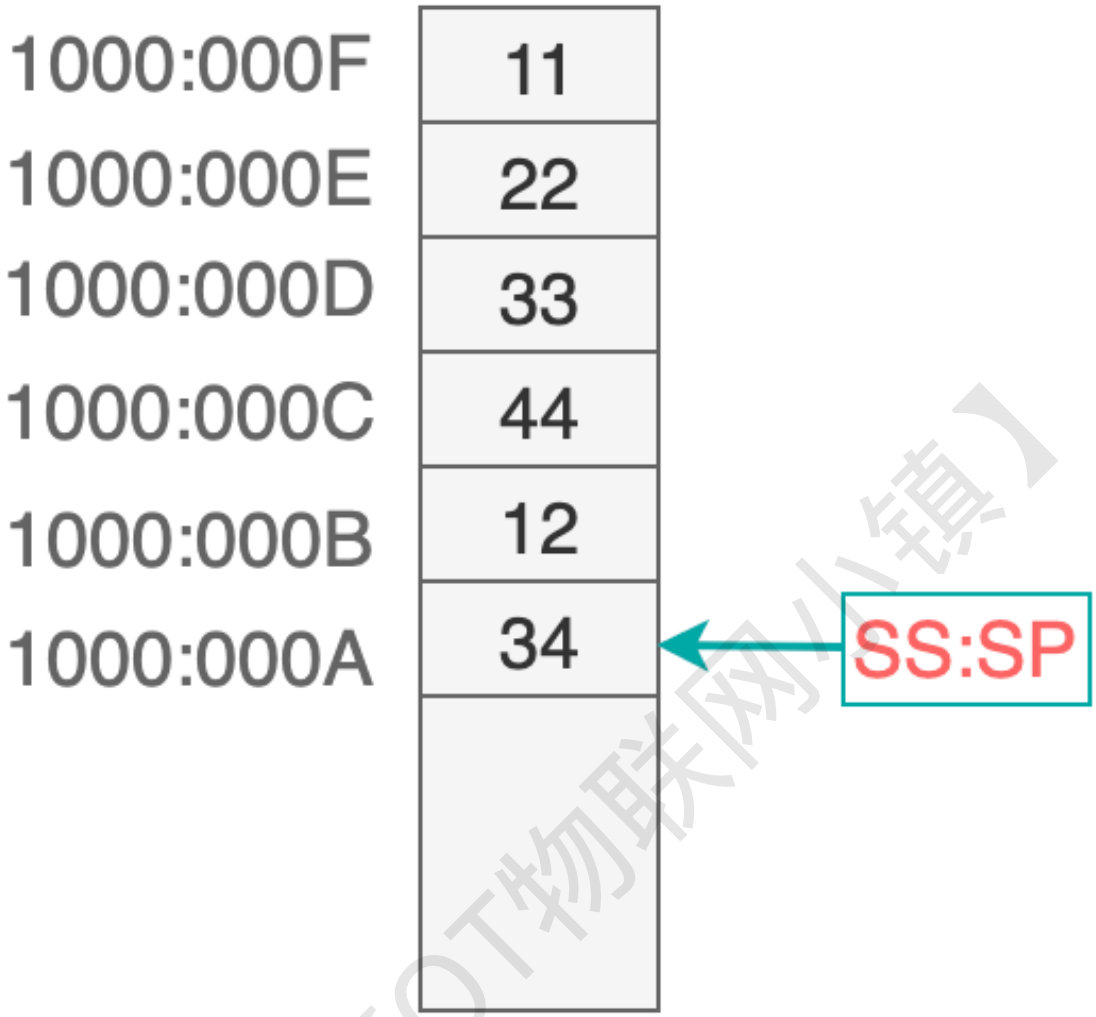
当执行下面这 2 条指令时：

```
mov ax, 1234H
push as
```

栈顶指针寄存器 SP中的值首先减 2，变成 000A：



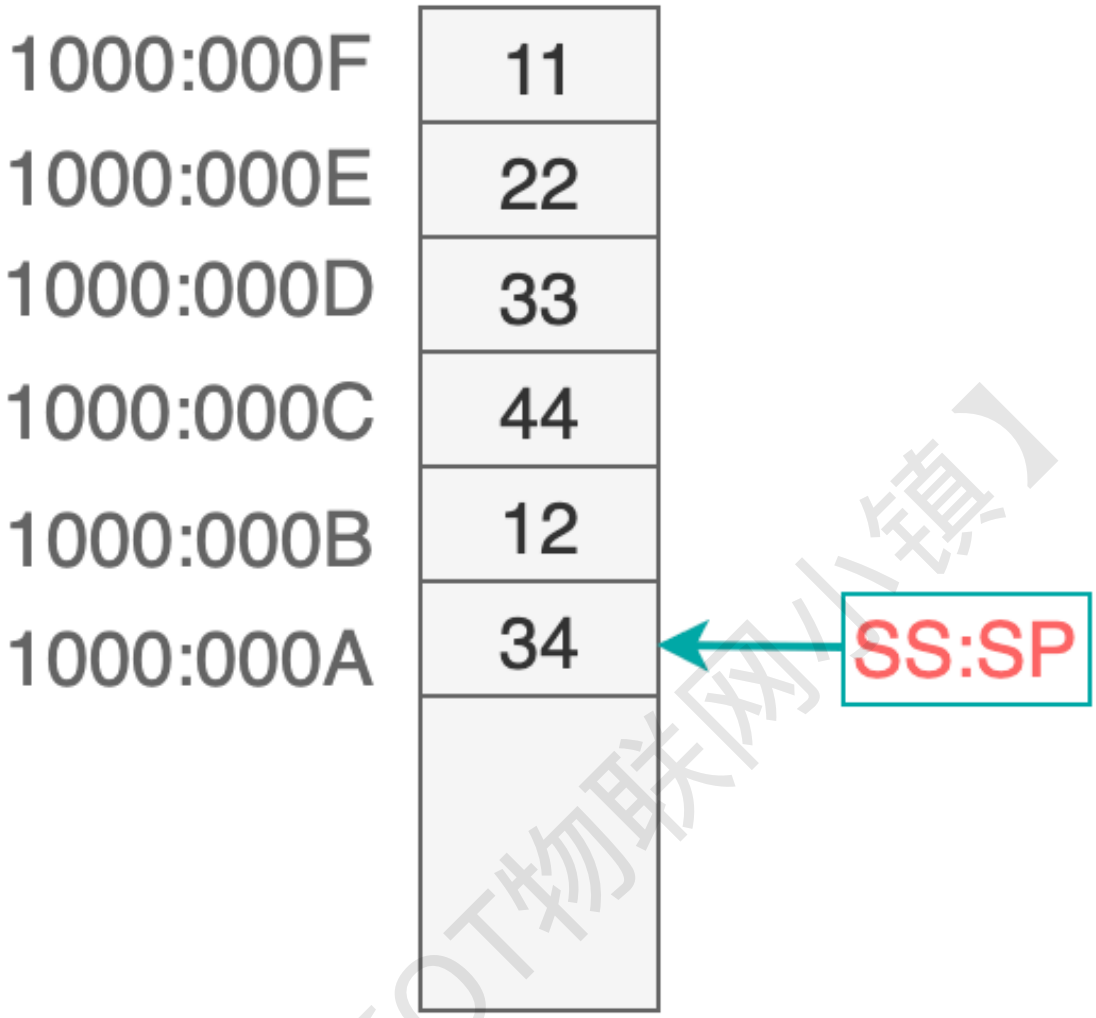
然后，再把寄存器 ax 中的值 1234H 放入 SS:SP 指向的内存单元处：



出栈的操作顺序是相反的：

```
pop bx
```

首先把 SS:SP 指向的内存单元中的数据 1234H 放入寄存器 bx 中，然后把栈顶指针寄存器 SP 中的值加 2，变成 000C：



以上描述的是 8086 处理器中对栈操作的执行过程。

如果你看过其他一些栈相关的描述书籍，可以看出这里使用的是“满递减”的栈操作方式，另外还还有：满递增，空递减，空递增 这几种操作方式。

> 满：是指栈顶指针指向的那个空间中，是一个有效的数据。当一个新数据入栈时，栈顶指针先指向下一个空的位置，然后 把数据放入这个位置;
>
> 空：是指栈顶指针指向的那个空间中，是一个无效的数据。当一个新数据入栈时，先把数据放入这个位置，然后栈顶指针指向下一个空的位置;
>
> 递增：是指在数据入栈时，栈顶指针向高地址方向增长;
>
> 递减：是指在数据入栈时，栈顶指针向低地址方向递减;

实模式和保护模式

从以上对内存的寻址方式中可以看出：只要在可寻址的范围内，我们写的程序是可以对内存中任意一个位置的数据进行操作的。

这样的寻址方式，称之为实模式。实，就是实在、实际的意思，简洁、直接，没有什么弯弯绕。

既然编写代码的是人，就一定会犯一些低级的小错误。或者一些恶意的家伙，故意去操作那些不应该、不可以被操作的内存空间中的代码或数据。

为了对内存进行有效的保护，从 80386 开始，引入了 保护模式 来对内存进行寻址。

> 有些书籍中会提到 IA-32A 这个概念，IA-32 是英特尔 Architecture 32-bit简称，即英特尔32位体系架构，也是在386中首先采用。
>
> 虽然引进了保护模式，但是也存在实模式，即向前兼容。电脑开机后处于实模式，BIOS 加载主引导记录以及进行一些寄存器的设置之后就进入保护模式。

从 386 以后引入的保护模式下，地址线变成了 32 根，最大寻址空间可以达到 4GB。

当然，处理器中的寄存器也变成了 32 位。

我们还是用 段基址 + 偏移量 的方式来计算一个物理地址，假设段寄存器中内容为 0，偏移地址最大长度也是 32 位，那么一个段能表示的最大空间也就是 4GB 。

这也是为什么如今现代处理器中，每个进程的最大可寻址空间是 4GB(一般指的是虚拟地址)。

一句话总结：实模式和保护模式最根本的区别就是 内存是否收到保护。

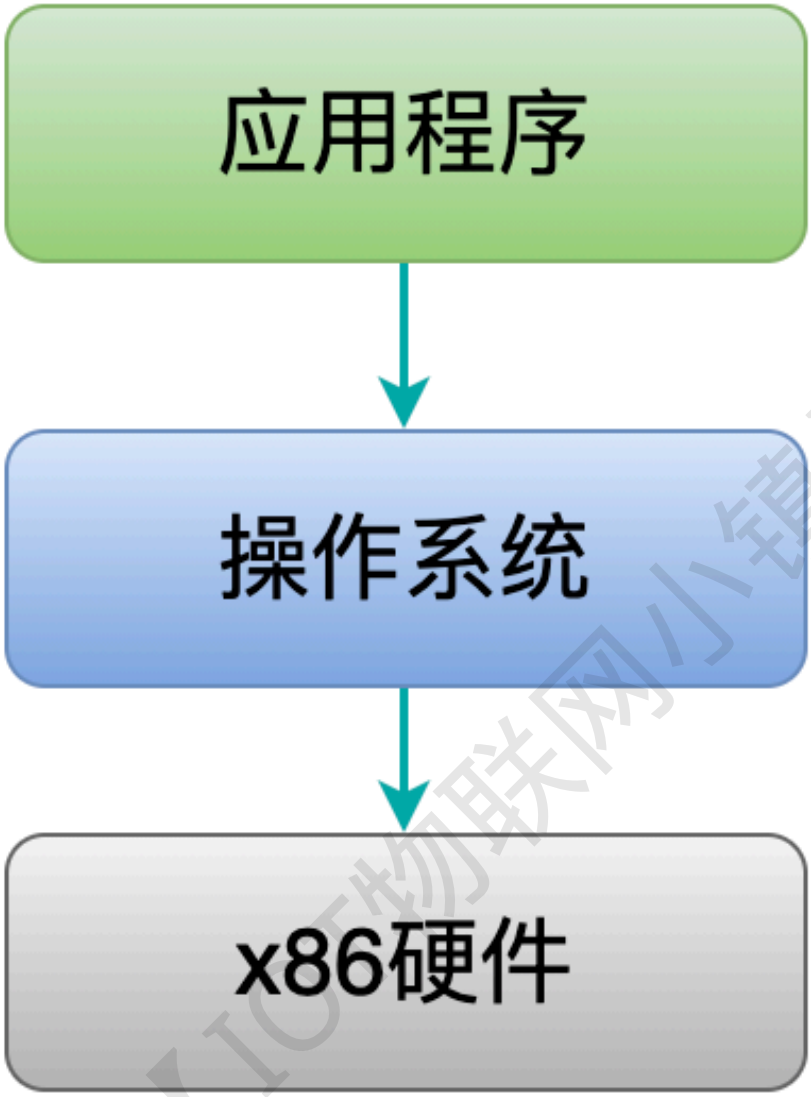
Linux 中的分段策略

上面描述的分段机制是 x86 处理器中所提供的一种内存寻址机制，这仅仅是一种机制而已。

在 x86 处理器之上，运行着 Windows、Linux 获取其它操作系统。

我们开发者是面对操作系统来编程的，写出来的程序是被操作系统接管，并不是直接被 x86 处理器来接管。

相当于操作系统把应用程序和 x86 处理器之间进行了一层隔离:



因此，如何利用 x86 提供的分段机制是操作系统需要操心的问题。

而操作系统提供什么样的策略给应用程序来使用，这就是另外一个问题了。

那么，Linux 操作系统是如何来包装、使用 x86 提供的段寻址方式的呢?

是否还记得上一篇文章中的这张图:

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

这是 Linux2.6 版本中四个主要的段描述符，这里先不用管段描述符是什么，它们最终都是用来描述内存中的一块空间而已。

在现代操作系统中，分段和分页都是对内存的划分和管理方式，在功能上是有点重复的。

Linux 以非常有限的方式使用分段，更喜欢使用分页方式。

上面的这张图，一共定义了 4 个段，每一个段的基地址都是 0x00000000，每一个段的 Limit 都是 0xFFFFF。

从 Limit 的值可以得到：最大值是 2 的 20 次方，只有 1 MB 的空间。

但是其中的 G 字段表示了段的粒度，1 表示粒度是 4 K，因此 1 MB * 4K = 4 GB ，也就是说，段的最大空间是 4 GB。

这 4 个段的基地址和寻址范围都是一样的！主要的区别就是 Type 和 DPL 字段不同。

DPL 表示优先级，2 个用户段(代码段和数据段) 的优先级值是 3，优先级最低（值越大，优先级越低）；2 个内核段(代码段和数据段)的优先级值是 0，优先级最高。

因此，可以得出 Linux 系统中的一个重要结论：逻辑地址与线性地址，在数值上是相等的，因为基地址是 0x00000000。

关于 Linux 中的内存分段和分页寻址方式更详细的内容，我们以后再慢慢聊。

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

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

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。