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。

在软件开发中，中断是一个绕不开的重要话题，但是，不知道您是否遇到过这样的困惑：

很多书籍、文章在介绍中断相关的知识点时，说的都挺有道理。

这篇文章对中断的讲解很正确，那篇文章在描述中断的时候也挺对的，但是，这两篇文章中，怎么有些内容是矛盾的啊？！

单独看任何一篇文章感觉都有道理，看的越多，反而越迷糊？

好比在森林里迷路了，如果只有一个指南针，肯定能走出来。

但是，如果你有 2 个指南针，所指的方向却是相反的，这个时候应该相信谁呢？！

我们仔细梳理了一下就会发现：每一篇文章都是在一定的语境、一定的上下文环境中来讲解的，不同文章的矛盾之处，恰恰是它们所描述的那个上下文大环境不同。

> 上下文环境，就是描述当前正在执行的程序相关的静态信息，比如：有哪些代码段，栈空间在哪里，进程描述信息在什么位置，当前执行到哪一条指令等等。

如果我们没有一个全局的视角，在同一个上下文环境中来对比不同的文章，就会让自己的理解和认识越来越蒙圈。

因此，对于这种概念比较庞杂，无法用某种确定的逻辑来贯穿的知识点，在脑袋中一定要有一幅全局的地图。

只有对这个全局的地图掌握了，在具体学习每一个局部的知识点时，才能知道自己所处的位置在哪里，才不至于走偏。

这篇文章，我们继续去繁从简，从 8086 这个最简单的处理器入手，来聊一下关于中断的一些知识。

有了这个储备，理清了基本的脉络之后，以后再去学习 Linux 系统中的中断相关内容时，才会有原来如此的感觉！

中断向量与中断描述符

中断向量这个词很时髦，也很神秘！

按道理，不应该在第一部分就端上中断向量这盘硬菜，应该从中断源开始聊起。

但是，毕竟我们已经学习过那么多关于中断的知识了，脑袋中肯定是对中断已经有了一些的基本认知。

所以，在这里我们还是首先来明确一下中断向量和中断描述符这个问题。

在前面的文章中已经聊过关于实模式和保护模式的问题，在 【Linux 从头学】这个系列中，我们一直以来描述的都是实模式下的事情。

> 本文是实模式下的最后一篇文章，下一篇文章将会进入保护模式。

那么，中断向量就是工作在实模式下的，处理器通过中断号和中断向量，来定位到相应的中断处理程序。

而中断描述符呢，就是工作在保护模式下，处理器通过中断号和中断描述符，来定位到相应的中断处理程序。

也就是说：中断向量和中断描述符，它俩的根本作用是一样的。

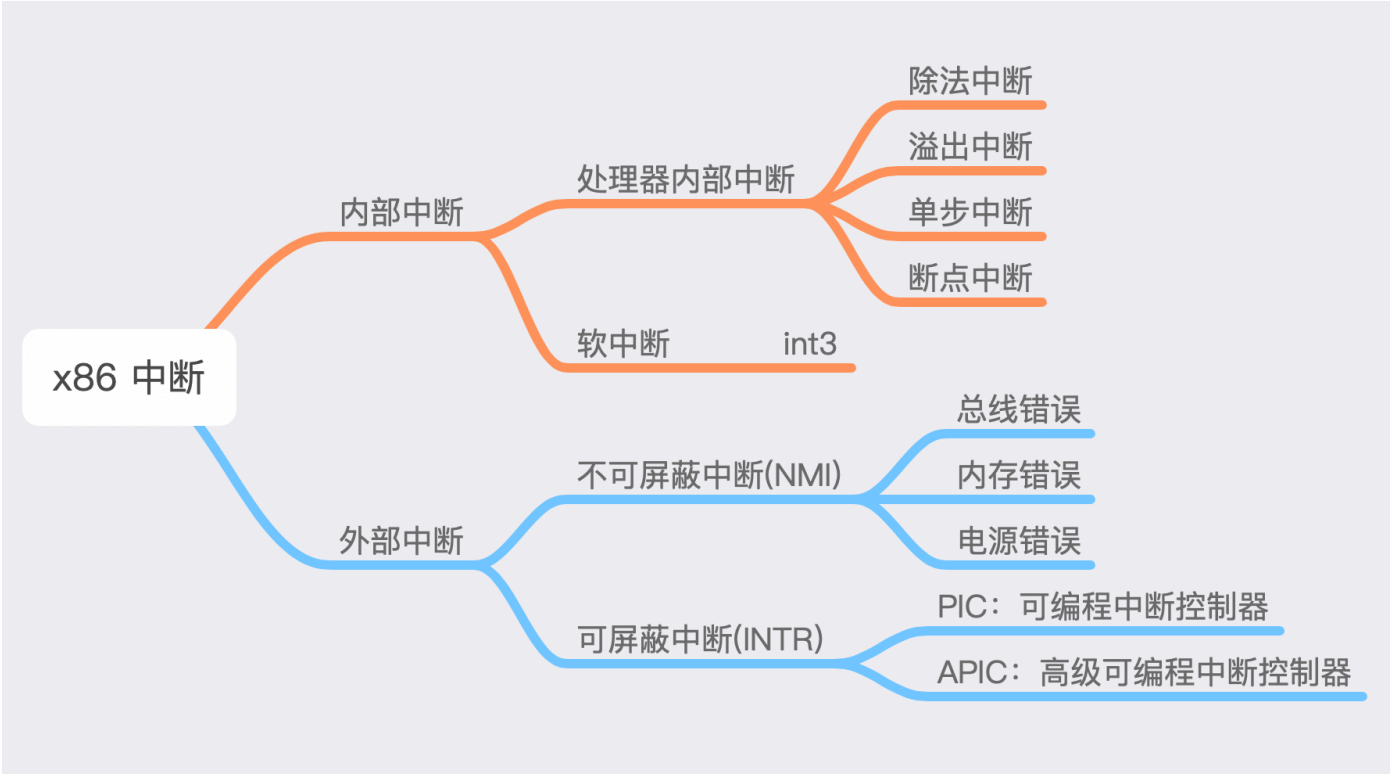
只是它们存在于不同的大环境中，而且从描述上也能感觉到，保护模式下的中断描述符会更复杂一些，功能也更强大一些。

它俩就像一对兄弟一样，从外表上看是差不多，功能也是类似。但是透入到内部去看，就会发现有很多的不同之处。

因此，这篇文章我们讲解的就是在实模式下的中断，这一点请大家先明白。

中断的分类

在 x86 系统中，中断的分类如下：



内部中断

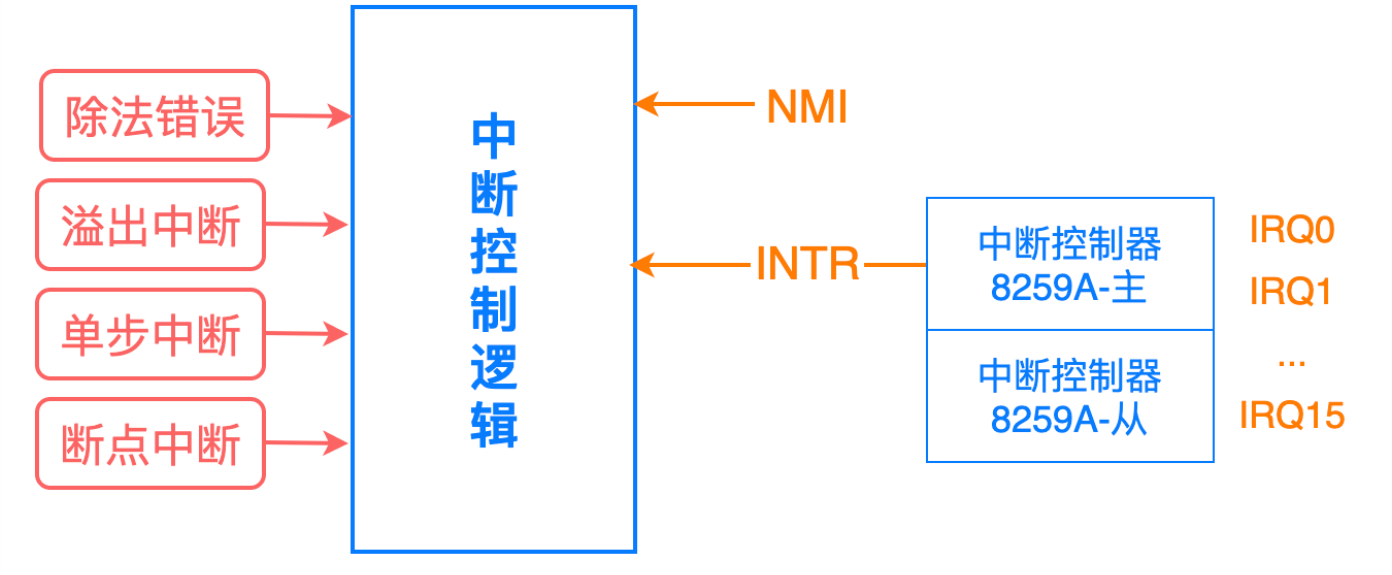
所谓的内部中断，是在 CPU 内部产生并进行处理的。比如：

1. CPU 遇到一条除以 0 的指令时，将产生 0 号中断，并调用相应的中断处理程序;
2. CPU 遇到一条不存在的非法指令时，将产生 6 号中断，并调用相应的中断处理程序;

对于内部中断，有时候也称之为异常。

软中断也属于内部中断，是非常有用的，它是由 `int` 指令触发的。比如 `int3` 这条指令，`gdb` 就是利用它来实现对应用程序的调试。

很久之前写过这样的一篇文章原来gdb的底层调试原理这么简单，其中就描述了 `gdb` 是如何通过插入一个 `int` 指令，来替换被调试程序的指令码，从而实现断点调试功能的。



外部中断

x86 CPU 上有 2 个中断引脚：INT 和 INTR，分别对应：不可屏蔽中断和可屏蔽中断。

所谓不可屏蔽，就是说：中断不可以被忽视，CPU 必须处理这个中断。

如果不处理，程序就没法继续执行。

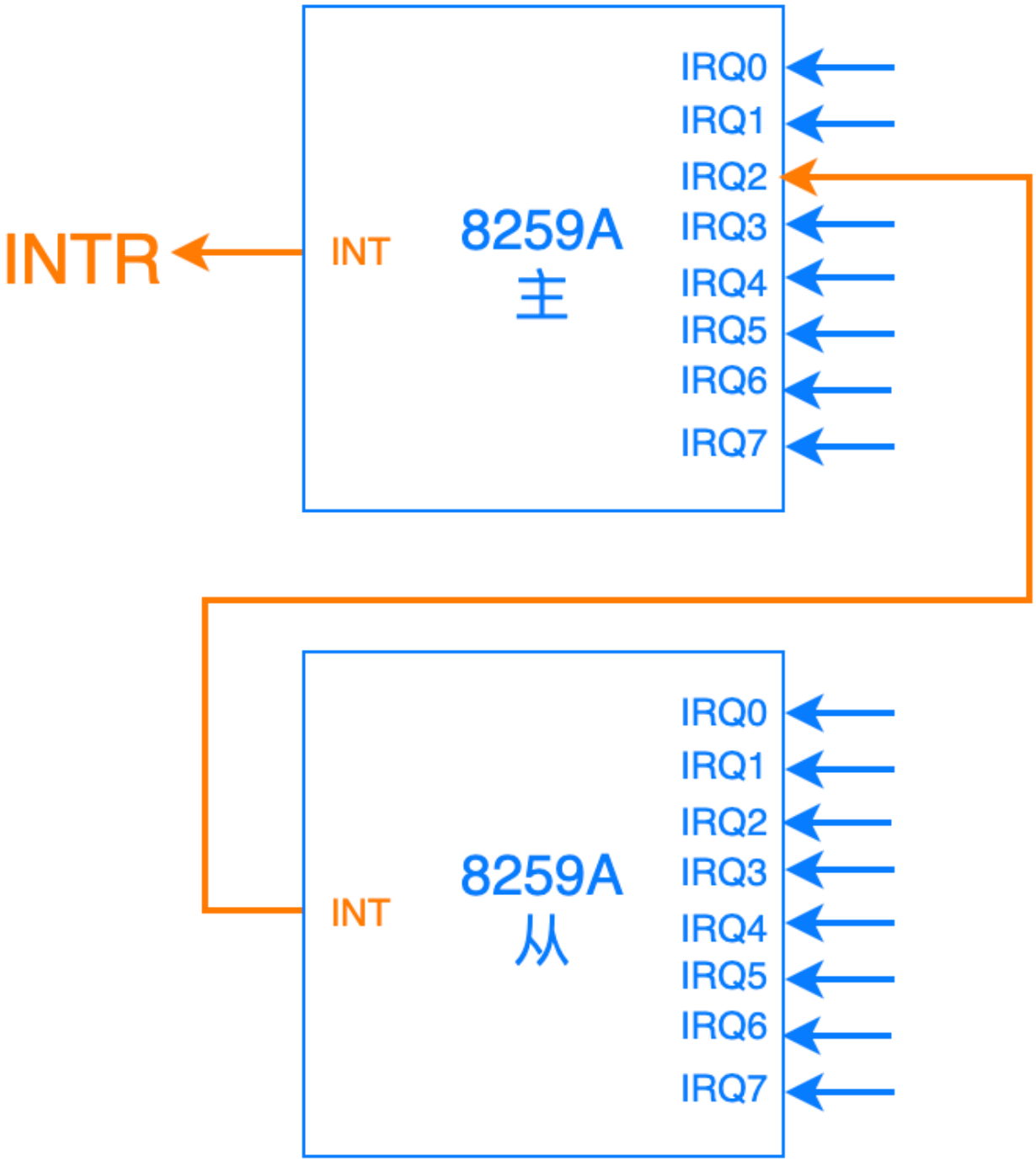
而对于可屏蔽中断，CPU 可以忽略它不执行，因为这类中断不会对系统的执行造成致命的影响。

对于外部的可屏蔽中断，CPU 上只有一根 INTR 引脚，但是需要产生中断信号的设备那么多，如何对众多的中断信号进行区分呢?

一般都是通过可编程中断控制器(Programmable Interrupt Controller, PIC)，在计算机中使用最多的就是 8259a 芯片。

虽然现代计算机都已经是 APIC(高级可编程中断控制器) 了，但是由于 8259a 芯片是那么的经典，大部分描述外部中断的文章都会用它来举例。

每一片 8259a 可以提供 8 个中断输入引脚，两片芯片级联在一起，就可以提供 15 个中断信号：



> 1. 主片的输出引脚 INT 连接到 CPU 的 INTR 引脚上;
> 2. 从片的输出引脚 INT 连接到主片的引脚 2 上;

这样的话，两片 8259a 芯片就可以向 CPU 提供 15 个中断信号了，比如：鼠标、键盘、串口、硬盘等等外设。

> 8259a 之所以称作可编程，是因为它的内部有相关的寄存器。
>
> 可以通过指定的端口号，对这些寄存器进行设置，让 8 根 IRQ 中断线上的信号，在送到 CPU 时，对应不同的中断号。

另外，对于外部可屏蔽中断，有 2 层的屏蔽机制：

> 1. 在 8259 芯片中，有中断屏蔽寄存器，可以对 IRQ0 ~ IRQ7 输入引脚进行屏蔽;

> 2. 在 CPU 内部，也有一个标志寄存器，可以对某一类中断信号进行屏蔽;

中断号

在 x86 处理器中，一共支持 256 个中断，每一个中断都分配了一个中断号，从 0 到 255。

其中，0 ~ 31 号中断向量被保留，用来处理异常和非屏蔽中断(其中只有 2 号向量用于非屏蔽中断，其余全部是异常)。

当 BIOS 或者操作系统提供了异常处理程序之后，当一个异常产生时，就会通过中断向量表找到响应的异常处理程序，查找的过程马上就会介绍到。

从中断号 32 开始，全部分配给外部中断。

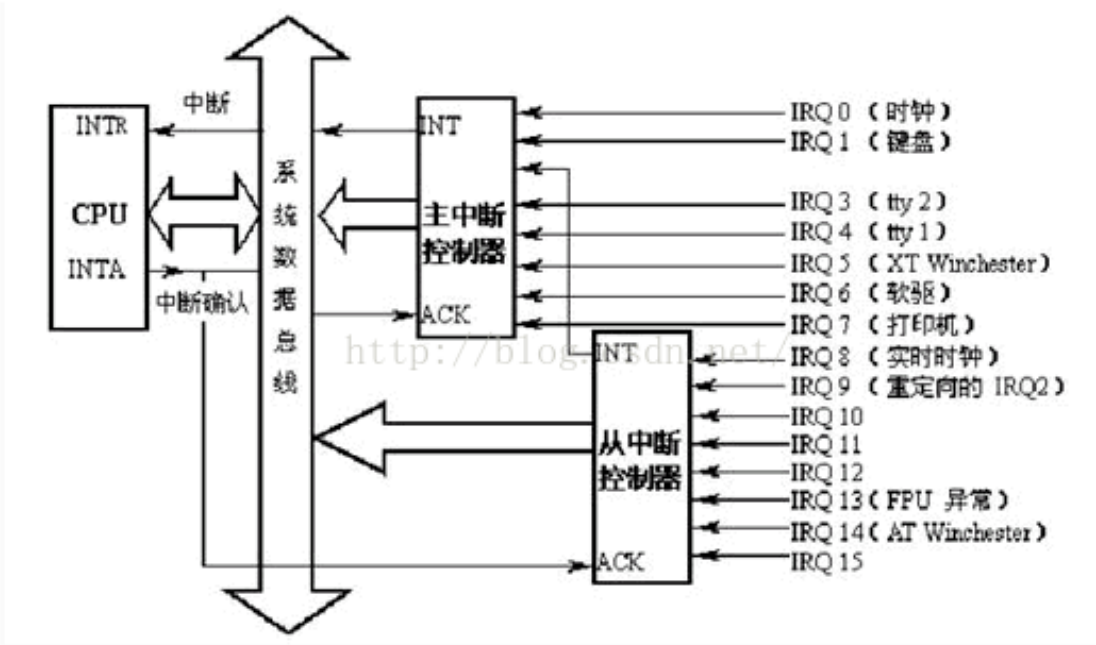
比如：

> 系统定时器中断 IRQ0，分配的就是 32 号中断;
>
> Linux 的系统调用，分配的就是 128 号中断;

我们来分别看一下内部中断和外部中断相关的中断号：

中断号	描述	功能 or 来源
0	除法错误	除法指令
1	单步中断	调试程序
2	非屏蔽中断	NMI
3	断点中断	调试程序

对于通过 8259a 可编程中断控制器接入的中断信号分配如下图所示：



刚才已经说过，8259a 是可编程的，假如我们通过配置寄存器，把 IRQ0 的中断号设置为 32, 那么主片上 IRQ1 ~ IRQ7 所对应的中断号依次加 1，从片上 IRQ8~IRQ15 对应的中断号也是依次递增。

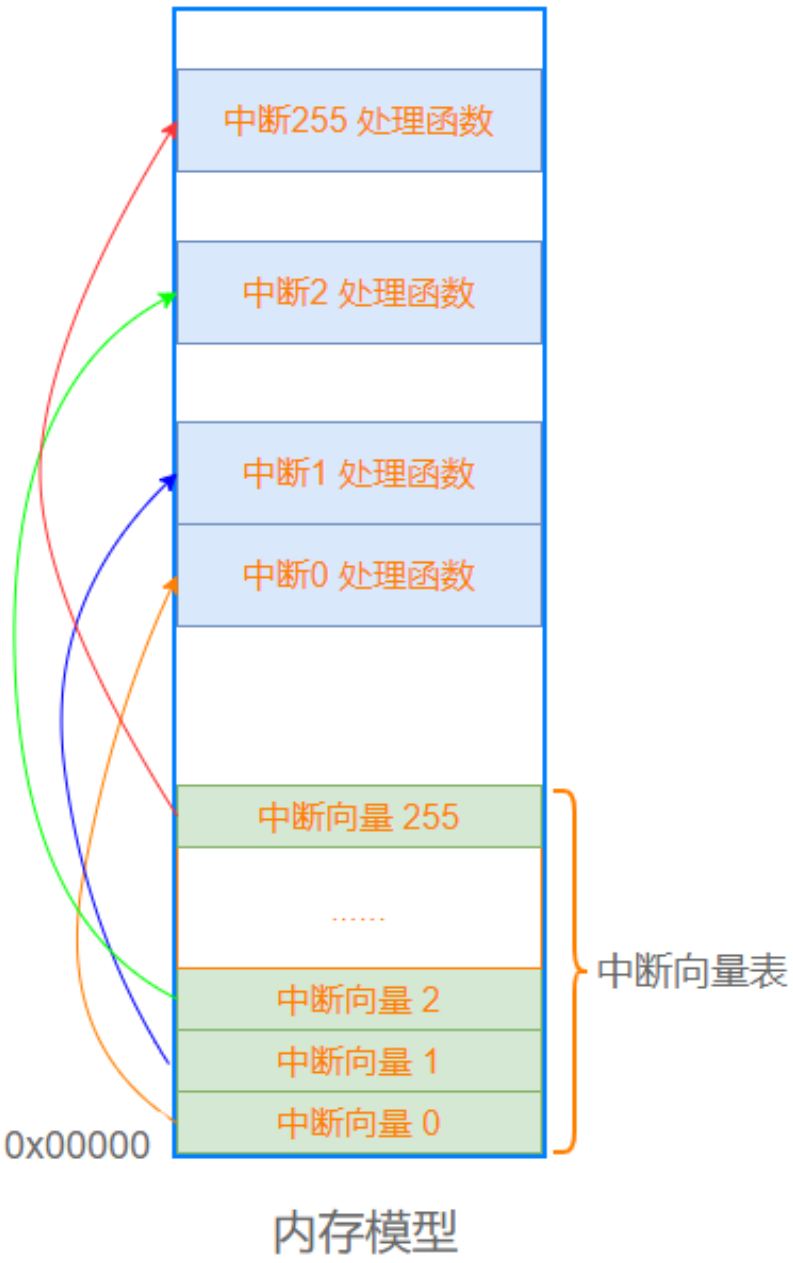
所以，有时候我们可以在代码中断看到下面的宏定义：

```
#define  IRQ0     32    // 电脑系统计时器
#define  IRQ1     33    // 键盘
#define  IRQ2     34    // 与 IRQ9 相接，MPU-401 MD 使用
#define  IRQ3     35    // 串口设备
#define  IRQ4     36    // 串口设备
#define  IRQ5     37    // 建议声卡使用
#define  IRQ6     38    // 软驱传输控制使用
#define  IRQ7     39    // 打印机传输控制使用
#define  IRQ8     40    // 即时时钟
#define  IRQ9     41    // 与 IRQ2 相接，可设定给其他硬件
#define  IRQ10    42    // 建议网卡使用
#define  IRQ11    43    // 建议 AGP 显卡使用
#define  IRQ12    44    // 接 PS/2 鼠标，也可设定给其他硬件
#define  IRQ13    45    // 协处理器使用
#define  IRQ14    46    // IDE0 传输控制使用
#define  IRQ15    47    // IDE1 传输控制使用
```

中断向量和中断处理程序

当一个中断发生的时候，CPU 获取到该中断对应的中断号，下一步就是要确定调用哪一个函数来处理这个中断，这个函数就称作中断服务程序(Interrupt Service Routine，ISR)，有时候也称作中断处理程序、中断处理函数，本质都一样。

中断向，就是通过中断号去查找处理程序的重要的桥梁！



内存模型

中断向量的本质

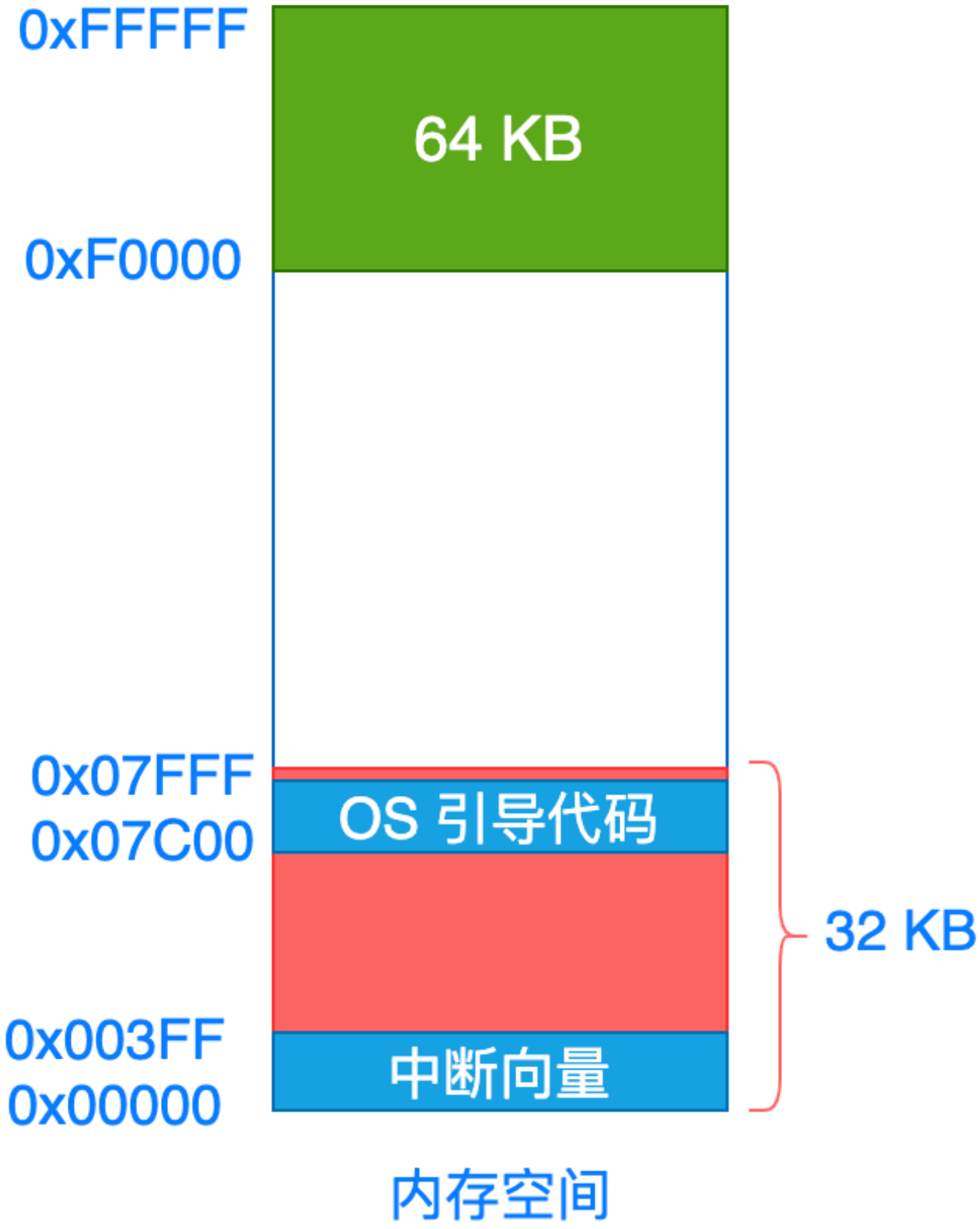
在 8086 中，一个中断向量，就是一个 段地址:中断处理函数偏移量 这样的一对数据，通过这个数据，就可以定位到内存中指定位置的那个中断处理函数。

非常类似于高级编程语言中的函数指针，就是用来指向一个函数的开始地址。

8086 规定：256 个中断向量，必须从内存的 0 地址处开始存放。

每一个中断向量占用 4 个字节(2 个字节的段地址，2 个字节的偏移地址)，256 个中断一共占用了 1024 个字节的空间。

之前的文章中，已经介绍过相关的内存模型，如下图所示：



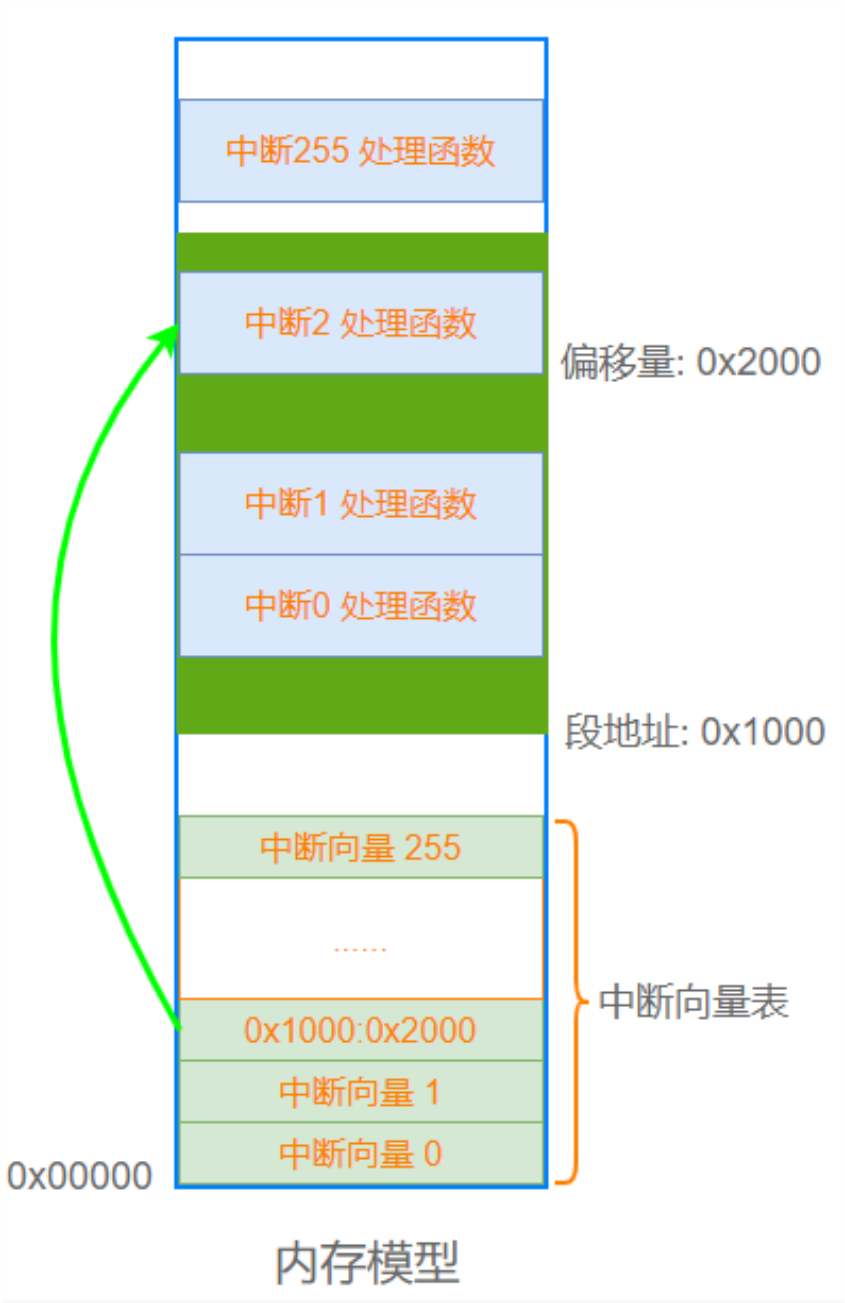
如果把一个中断向量看作函数指针，那么这个中断向量表就相当于是函数指针数组。

举例：

假设 2 号中断被触发了，CPU 就会到中断向量表中查找 2 号中断的中断向量。

因为每一个中断向量占据 4 个字节，那么 2 号中断向量的开始地址就是 2 * 4 = 8，第 8 个字节。

然后在第 8 个字节开始，取 4 个字节的内容：0x1000:0x2000。



意思是：2 号中断的处理函数，在段地址为 0x1000，偏移量为 0x2000 的位置处。

那么 CPU 就按照 8086 的物理地址计算方式，得到中断处理函数的物理地址为 0x12000 (段地址左移 4 位 + 偏移地址)，于是就跳转到该函数地址处去执行。

> 由于 Linux 系统是运行在保护模式，在这个模式下，当发生中断时，是通过中断描述符来查找中断处理函数的。
>
> 每一个中断描述符，描述了一个中断处理函数所在段的选择子和偏移量，本质上也是用来查找一个中断处理函数。

中断处理程序的安装

既然通过中断向量，找到了中断处理程序，那么这些中断处理程序都是谁放在内存中的呢?

如果您看过一些比较底层的计算机书籍，就能看到一般都会举例：如何手动的把一个普通函数设置为一个中断处理程序。

操作步骤是:

1. 在代码中，写一个普通函数;
2. 把这个函数的指令码，搬运到内存中的某一个位置;
3. 把这个位置(段地址:偏移量)，作为一个中断向量，设置到中断向量表中;

此时，如果发生了该中断，你所提供的函数就作为中断处理函数被执行了。

当然了，在一个计算机系统中，BIOS、操作系统和各种外设，会自动为我们提供很多基本的中断处理函数的。

比如：BIOS 中就提供了软中断、内部中断、硬件中断等处理函数，这些函数是固化在 BIOS 的代码中的(映射到 BIOS 所在的 ROM 芯片上)，BIOS 只需要把这些处理函数的地址，写入到中断向量表中的相应位置即可。

在之前的文章中提到过，内存中的某些位置是映射到外设的 ROM，在这些外设的 ROM 中也存在一些外设自带的程序。

BIOS 在启动时，会扫描这些映射到外设的内存空间，通过某些关键字信息，如果发现外设有自带的程序，就会去执行。

这些外设程序一般是进行一些自身的初始化，并填写相关的中断向量表，使它们指向外设自带的中断处理程序。

对于操作系统来说就更不用说了，它会重新安排自己需要的中断处理函数，这部分内容我们以后再一起学习、讨论！

中断现场的保护和恢复

当一个中断发生的时候，肯定有一个正在执行的程序被打断。

当中断处理函数执行结束之后，这个被打断的程序需要从刚才被打断的地方继续执行(暂时先不要考虑从中断返回点，进行多任务切换的事情)。

而一个程序执行的上下文环境，就是处理器中的各种寄存器内容：代码段寄存器 cs，指令指针寄存器 sp，标志寄存器 FLAGS。

但是，在中断处理程序中，也需要使用这些寄存器。

处理器中的这些寄存器，就是每一个程序执行时上下文信息的存储容器，当然也包括终端处理程序！

因此，在进入中断处理程序之前，CPU 会自动的把这些寄存器 push 到栈中保存起来，然后再跳转到中断处理程序中去执行。

当中断处理程序执行结束后，CPU 会从栈中弹出这些内容，恢复到相应的寄存器中，于是被打断的程序就可以继续执行了。

总结：中断的本质

从功能的角度看，中断有 2 个作用：

> 1. 提供执行异步序列的机制；
> 2. 给应用程序提供进入系统层的入口；

关于第 2 点，以后在介绍到 Linux 中的 `int 0x80` 中断就非常清楚了，也就是通过中断，让应用层的程序有机会进入到系统代码中去执行。

因为应用层与操作系统层的代码，是工作在不同的安全级别。

为了系统的安全，Linux 操作系统提供了这样的一个机制，让低安全级别的应用程序，进入到高安全级别的操作系统代码中去执行，毕竟所有的硬件等系统资源都是由操作系统来统一管理的。

我们再从中断处理程序的安装角度来看，中断本质上就是增加了一层间接性：通过固定位置的中断向量表，让中断处理函数的实际地址可以被动态的放在任意位置。

为什么这么说?

假如操作系统想为某一个中断提供处理函数，那么这个处理函数的地址放在内存中的什么位置比较合适?

需要考虑 CPU, 内存大小和布局等多种因素，非常复杂！

而通过使用中断向量表，就在一个固定位置处存放了很多个“指针”。

当中断处理函数放在内存中某个任意位置之后，让“指针”指向这个函数的地址就可以了，从而达到解耦的目的。

这样的话，无论是发生硬件中断，还是应用层代码通过中断门来调用操作系统提供的函数，只要触发相应的中断就可以了，简化了 CPU 的设计。

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

关于中断的相关内容，还有很多需要学习，任重而道远！

特别是在 Linux 系统中，中断处理又分为上半部分、下半部分，而下半部分又可以根据不同的功能需求采取不同的机制来处理。

我仍然是持有之前的观点：磨刀不误砍柴工。

把学习周期拉长，一点一滴的积累，Haste Makes Waste！

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。