第八讲 中断和异常

801-中断和异常的来源

> 大家好，我就是一个CPU,那我的任务呢就是
>
> 进行运算，有人交给我了一个本，上面写了很多的算式，让我进行运算，
>
> 好，那我就开始算了，上面写着3*4，我一算是12，填在后面的空格里，
>
> 还有呢15+15，等于30，卸载后面的空格里。很好。接着运算下一条。
>
> 1234567*7654321，这个一算，
>
> 结果好长啊，后面这个空格只有这么大，填不下，这可怎么办呢？
>
> 早期的电子计算机功能非常明确，那 就是进行数学运算。
>
> 但是如果运算出现了异常情况那应该怎么办呢？ 因为计算机当中都只能提供有限的单元来表示一个数，所以
>
> 有时候就会出现运算结果过大，无法全部保存的情况，这就叫做溢出。
>
> 如果从这个模型机上来看，溢出就会发生在ALU这个运算单元上。
>
> 这个运算单元的元操作数要么来自于寄存器，要么来自于内存。
>
> 我们假设都是32位的，对它们进行运算，比如说进行加法， 那么运算的结果就有可能超过32位。那么假设
>
> 在内存当中的某一条指令是这条运算指令发生了溢出，
>
> 那么CPU能怎么办呢？运算发生了溢出，CPU的硬件电路是很容易检测到的。
>
> 但问题就在于检测到了溢出之后应该怎么办？ 如果完全用硬件电路来处理溢出，那样不够灵活。
>
> 一旦CPU制造出来我们就没办法改变这个电路，从而没有办法修改处理溢出的方法，
>
> 所以最好还是用软件的方法来解决。那么来看看早期的计算机是怎么来解决这个异常的情况。
>
> 这是UNIVAC，也就说ENIAC的主要设计者爱科特和莫克丽
>
> 在离开宾夕法尼亚大学独立创办公司之后设计的那台计算机。
>
> 这是第一个带有异常处理的计算机系统。
>
> 如果它在进行运算时发生了异常情况，比如算术运算溢出，
>
> 它就会转向地址0，执行两条修复指令。
>
> 这个方法的思想是非常重要的。不然的话，一旦出现异常的情况，就只能停机，
>
> 然后等人工来解决问题以后才能继续运行。而如果我们能在地址0这里存放 几条指令，用于修复这个异常的情况，
>
> 从而就有可能在修复完成之后继续执行刚才的运算。
>
> 这样就可以避免停机。
>
> 而且如果之后发现这样的修复方法不太合适，那 只需要通过修改存放在地址0的这个地方的指令
>
> 就可以。用这样的方法来处理异常就非常的灵活和便利。

而这种思想也一直延续到现在。现代计算机 的CPU处理异常从根本上来说也是采用了这个方法，

那么结合我们现在讲的CPU的结构，来看看这个方法的简单原理。

那在ALU进行运算完成之后，

硬件电路就很容易检测出这次运算是否发生了溢出， 所以我们可以在ALU这里增加一根信号线，连到PC的更新部件，

那原先这个PC的更新部件会从几个来源对PC进行选择， 要么增加一个固定的值，比如说PC+4，

要么是根据转移目标地址产生一个新的PC， 那么现在增加一种新的情况，就是当运算溢出的信号

有效时，这个部件就固定产生0，送到PC寄存器，

这样在下一个周期，PC寄存器就会从地址0取出指令进行执行。

那么执行若干条指令之后，就可以修复这个异常带来的影响。

或者至少标记出了这次运算时错误的，以免对后续造成影响。

那如果在通过一些方式，可以记住刚才发生异常时的PC的值，

那就可以再通过PC更新的部件，将这个值装入PC寄存器当中去，从而从刚才发生异常的地- 方开始继续执行。

这就是一个简单的处理运算结果异常的方式。

不过后来的计算机就不仅仅是只进行数学运算，还要与外界

进行很多的交互，我们现在的计算机当中有很多的 输入输出的部件，那当CPU在运行程序的时候，

有可能我们就会在键盘上一个键，或者网络上就会来了一个数据包，

这些输入输出的操作都需要CPU中断现有程序的运行来进行处理， 那么这样的需求其实在很早也就有了。

1954年的DYSEAC是第一个带有外部终端处理的系统。

这台计算机也可以说是最早的移动计算设备。

说道移动计算设备，你可能会想到笔记本电脑。

或者平板电脑。不过这台计算机的移动性和他们有点不太一样，

它是要挂在卡车上移动的。不过它确实是在移动着进行计算。

和UNEVAC处理异常的方式不太一样，它设置了两个程序计数器，

然后根据外部输入输出设备的信号，可以在这两个程序计数器之间进行切换， 这样它就可以交叉执行两端不同的程序，

可以一段程序是运算的功能，而另一端程序是专门用于处理外部的中断。

那后来UNIVAC也增加了对外部中断的处理。

用于对风洞数据的实时收集，

那么简而言之，刚才提到的场景都是CPU在运行的时候遇到的一些事件， 这些事件有可能是线性程序本身出现的，那也有可能是在CPU的外部

甚至是整个系统的外部出现的事件。那么 这时就要求CPU强行中止现在正在运行的程序。

并且从一个新的存储器区域启动相应的程序去处理这些事件。

> 还有一点也很重要，就是在处理完这些事件之后，CPU需要恢复到原来的程序继续运行。
>
> 那这些事件就被称为中断或者异常。
>
> 那究竟中断和异常又有什么区别呢？ 那其实在这件事情上也没有明确的定论，
>
> 不同的体系结构，不同的书籍，不同的研究群体，对它们的定义都有区别。
>
> 其命名方式大体上可以分为这么几类。
>
> 一种是把CPU内部运行程序产生的这些事件
>
> 称为内部中断，而把CPU外部那些输入输出设备产生的事件称为外部中断。
>
> 而这些所有的事件又统称为中断。
>
> 而第二种方式则是将CPU运行程序所引发的特殊的事件称为软件中断，
>
> 而把外部的这些输入输出设备产生的事件称为硬件中断。
>
> 这种方式就是把运行程序看做一种软件的行为， 而把外设的工作则归为一种硬件的行为。
>
> 还有一种常见的方式就是将外部设备发生的这些事件
>
> 称为中断，而将CPU内部运行程序发生的这些事件称为异常。
>
> 这也是一种常见的命名方式。
>
> 但是对这些事件的统称，也是有不同的表示方式。
>
> 有一些把中断和异常统称为中断，也有些地方把中断和异常统称为异常。
>
> 那么这些不同的名词时从不同的角度来理解和表述这件事情，
>
> 但它们所代表的内部事件和外部事件的具体行为都是一样的。
>
> 在我们的课中将采用中断这种统称 而在具体划分上主要采用外部中断和内部中断的表述方式，
>
> 但也会根据不同的场景有时也使用其他的表述方式。这请大家注意。

> 那好了，现在制造这本手册的人告诉我了， 如果你发现运算的结果在这空格里填不下的话，
>
> 那也不要着急，就翻到这个手册的最前面，啊，这个手册最前面的第一行，
>
> 就写了应该怎么处理这种情况，照着这个往下做，就可以了。
>
> 那这样事情就很简单了。
>
> 但是随着我这个运算任务越来越多，我可能会遇到新的问题。
>
> 突然发现不是空格写不下了，这个字母我不认识， 这个运算符我没见过，
>
> 那应该怎么办呢?

802-中断向量表的结构

> 那好，我现在已经知道了，我在运算的时候
>
> 一旦遇到了异常情况，就翻到第一页的第一行开始写的这些操作的指示，开始往下执行。
>
> 开始往下执行，这就能解决问题了。
>
> 但是问题在于这段操作，解决的是我那个运算结果在空格里填不下的问题。
>
> 可是我遇到新的问题应该怎么办呢，我们可能还会遇到很多的其他的问题。
>
> 这个时候呢，这个手册的制造者实际上就需要这些改进了。

它需要把第一页不能只写一种解决情况，

他要画一个表格。在这个表格里分了一二三四五条，填上不同的情况。我们遇到了不认识的操- 作符号。

请看第一页的第二条，如果这个地方粘了一个虫子， 然后你这个数字看不清楚了，请见第一页的第六条。

这样我们就能解决不同的情况了。

那我们先来回顾UNIVAC对异常处理的方式。

当算术运算溢出的时候，UNIVAC就转向地址0去 取出指令。在哪里会执行两条修复指令。这个处理方式

已经是很灵活的了。只不过还有一些值得改进的地方。

例如在存储器当中，如果从地址0开始只预留了两条指令的空间

用于修复指令。那如果后来要改变修复的方式，需要再增加几条指令。

那这里预留的空间是否就有可能不够了。另外当需要处理的异常情况变多的时候，

那就需要根据异常情况的不同区执行不同的异常处理指令。

那如果遇到异常都只能转向地址0，就没有办法处理多种不同的异常了。

因此，在这两个方面都需要进行改进。

那么就来看一看后来，8086是怎么做的?

8086是一个十六位的cpu。它内部有四个16位的通用寄存器。

对外则有16根数据线，但是它的地址显要更多一些，一共有20根。

这样可以寻址的内存空间就是2的20次方，也就是一兆个byte。

那由于它内部的寄存器与运算器都是16位宽的。

要生成20位宽的地址，就得用一定的转换方法。8086采用的是“段加偏移”的方式。

然后对于这一兆的空间不都是可以任意使用的。

有两个区域保留作专门的用途。在这一兆字节的内存空间中， 最低的1K个字节，保留作中断向量表区。

而最高的十六个字节保留为初始化程序区。

我们结合图示来进行说明，这里用填充了颜色的方块用来表示这一兆字节的存贮器。

用8086cpu复位之后，它第一次取指令的操作发出的地址是四个F一个0。

这个地址就是在这一兆内存空间的最高的16个字节的这个地方。

那这个区域实际上是很小的，只能放很少的几条指令。

通常放在这里的是一条无条件的转移指令。

转移到内存空间当中的另一个地方。在那个地方存放着后续的系统程序。

那CPU复位之后，为什么不从全0的地址开始取址呢？那样看起来岂不是更自然一些。

那因为从0开始的地址空间也已经被占用了。

这个1K的字节被用作中断向量表区，它一共存放了

256个中断向量。每个中断向量占四个字节。这样正好就是1K个字节。

那除了这两块专用的区域，其他区域就可以用来存储一般的程序指令和数据。

那在这块区域，还有那些用于进行中断处理的程序。

这些程序就被称为中断服务程序。

而这些程序代码起始地址则被称为中断服务程序的入口地址。

这就是中断程序向量的定义。

现在的CPU一般都能够处理多种不同的中断类型。

那么每个中断类型就对应一个中断向量，一共4个字节。

这四个字节当中，前两个字节用于存放中断服务程序入口的偏移量。

而且是低字节在前，高字节在后。因此对于这个中断向量， 这两个字节就会被存放到

而且前一个字节是寄存器当中的低字节，后一个字节是寄存器当中的高字节。

那么中断向量当中的后两个字节，则对应了终端服务器入口的地址的段基址。

用来存放到代码段计算器，也就是CS计算器。

那么同样，前面那个字节对应了寄存器当中的低字节， 后面的字节对应了寄存器当中的高字节。

那么在8086当中或者是后来X86处理器的实模式下，

就需要用CS和IP这一对寄存器，来指定一个内存的地址。

这个地址的产生方式就是叫做段加偏移。

那钢槽CS寄存器就是一个段寄存器。它是16位的， 而刚才的IP寄存器则对应了偏移量，也是一个16位。

那这两个16位的地址就构成了逻辑地址。

那通常的表示方式就是用一个冒号分隔开这两个16位数。

拿在cpu产生地址是，会将段寄存器当中的数左移4位，

然后加上偏移量，这样加法运算的结就是20位的物理地址。

这就是逻辑地址生成物理地址的方式。实际上是段基值乘以十六加上偏移量。

那对于二进制来说左移四位就相当于乘以16了。

那基于这样的方式，每个中断向量都由两个段基值

和两个向量的偏移地址组成，还因为每一个中断向量占四个字节。

在整个中断向量表中，一共有256个中断向量。

分别命名为0号、1号、一直到255号中断。

那这个中断向量表要在系统里面启动时进行初始化。

那么假设1号向量的初始值是这样的，那当cpu接收到中断时， 如果发现时1号中断。

那因为各个中断向量放置的地址是固定的，那cpu不需要通过执行指令，

直接通过硬件电路的设置，就可以发出内存访问来读取这四个字节的内容。

然后将其中高两个字节送到CS寄存器当中去， 低两个字节送到IP寄存器当中去。

那对于8086来说，这两个寄存器的功能 就相当于我们在之前介绍处理器内部结构时，提到的PC寄存器。

所以这两个寄存器的值一旦发生改变， 下一个周期cpu就会从那个新的地址开始取下一条指令。

那更具段加偏移的计算方法，cpu发出的地址均是43006,。

因此也就是说，在遇到一号中断时， cpu就会转到43006这个地址开始执行程序。

那当然，需要实现把一号中断的服务程序存放在这里。

与此类似，我们还会把0号中断的服务程序放在存储器的另一个地方。

然后将中断程序的起始地址放在段基值和偏移地址，存放在0号中断向量所在的位置。

那当cpu遇到中断时，如果发现是0号中断，则会将0号中断向量对应的内容取出，

分别填到CS和IP寄存器当中去。

这样cpu就会从0号中断符合的起始地址，开始对指令进行执行。

那我们注意到，这些中断服务程序在内存当中的存放顺序并没有要求。

并不需要按照中断类型的顺序。先放0号中断服务程序，再放1号中断服务程序。

而是可以随意放置。只需要把它的起始地址存放在中断向量表的对应位置就可以了。

这样做就比UNIVAC的方式要灵活的多。

一来中断服务程序就可以可长可短， 不用担心在从0开始的地址到底要预留多少的空间才够。

二来中断服务程序的存放位置如果发生改变，也没有关系。

不需要修改cpu的硬件设计，而只需要修改中断向量表中对应的中断向量就可以了。

这样只用初始化好中断向量表， 并在存储器当中准备好对应的中断服务程序。

cpu在遇到中断时就可以自动的对应的跳到中断服务程序进行处理了。

那关于中断向量的相关的计算，我们来看几个简单的练习。

第一，如果中断类型码，就叫做中断类型号为20H， 那中断向量起始的逻辑地址应该是什么呢?

首先我们知道中断向量表示从地址0 开始的，

一共256个中断向量顺序存放，而每个中断向量占四个字节。

所以中断向量存放的位置就是它的中断类型号乘以4， 那这个中断地质就是0000:0080。

那如果这个中断向量中四个字节的内容 分别是10H、20H、30H和40H，

那中断服务程序的入口地址应该是什么呢? 按我们刚才介绍的情况，这四个字节单元低两个字节

对应了IP寄存器，搞两个字节对应了CS寄存器。

而且地址较低的这个字节是放在寄存器当中较低的位置。

所以这中断服务程序的入口地址 应该是4020:2010。

这样的练习就是说明了cpu在遇到了中断之后，

硬件完成的工作。如果cpu现在遇到的中断类型号是20H，

则会通过硬件进行乘4的操作，从而得到这个逻辑地址。然后将这个逻辑地址

发到存储器中，多亏了这四个字节的内容。

然后按照我们刚才规定的原则拼接出了这样两个16位数。

并且把这两个16位数分别存放到CS和IP计算器当中去。

> 这样在下一个实施周期就会将新的地址发送到存储器去 下一条指令了。然后我们再来看另一条练习。
>
> 如果我们现在要为17H号中断新写了一段中断服务程序。
>
> 而且把这段中断服务程序放到了存储器的某一个地方。
>
> 地址是2340H:7890H。那现在我们就需要去更新中断向量表。
>
> 那现在问题就是，我们要更新中断向量表中的哪四个字节？而且更新成什么样的内容？
>
> 那我们可以一起来算一下。因为中断类型号是17H，
>
> 那对应中断向量的地址就应该是它乘以4。由这个地址开始，
>
> 想高地址增长，一共4个字节。所以这四个字节的逻辑地址是这样的。
>
> 那这四个字节单元当中分别应该填写什么样的内容呢？ 那么还是要记住这个原则，如果把这个逻辑地址从右往左看。
>
> 最右边的这个字节放在最低的地址，而最左边的这个地址放在最高的地址。
>
> 这四个字节依次排放。
>
> 所以地址由低到高四个字节内容分别应该是90、78、40和23。
>
> 那这两个练习就分别展示了cpu硬件查找中断向量表的过程。
>
> 和准备好中断服务程序去初始化或者修改中断向量表的过程。
>
> 那对于8086的中断向量表cpu已经固定使用了前五个类型的中断。
>
> 那具体的功能我们后面会再介绍， 那之后的27个中断也是保留给后续的cpu使用的。
>
> 而除了前20个中断，之后的224个的中断则是交给使用cpu的用户自行定义。

> 那好，我们现在知道了原来我们在第一页呢，
>
> 制造者给我们列了一张表格。
>
> 一共有256条。那我们在遇到异常情况的时候呢就可以根据事先的约定对应不同的情况，
>
> 去找对应的那一条表象。表象呢实际上是指一个页码。
>
> 它比如说第四条说反倒第十二页，这第十二而翻过来以后呢。
>
> 写了具体的操作，我们应该做什么情况。这一项还挺复杂，写了好几页，对吧。
>
> 我们都放在第一页是放不下的。所以这就是我们现在要怎么处理异常情况的方法。
>
> 那么随着我们要做的运算任务的增加呢，这个表的内容可能还需要进一步的扩展。
>
> 那么我们后来就来看一看它是怎么扩展的。

803-中断向量表的发展

> 那好，现在这个手册的制造者已经说了，
>
> 他在第一页就留好了一个表格，一共有256行。
>
> 虽然现在没有填写完全，啊，但是后面呢，它随着不断的升级，推出新的手册的时候呢？
>
> 会继续补充后面的一些条款。啊， 那我们现在呢，就来看一看，历代的这个手册它是怎么补充这个条款的。

> 这是8086的总共一兆的地址空间，那在这其中最低的1K 字节固定用来放置中断向量表。

这个中断向量表当中包括256个中断向量，每个中断向量为4个字节。

这4个字节就指向了对应的中断服务程序的入口地址。

那在8086CPU中，已经规定了前5个类型的中断。

类型0为除法错，也就当

除法运算出错时，CPU就会来取得这个中断向量，并调用对应的中断服务程序。

类型1是单步，类型2是非屏蔽中断，类型3是断点中断，

类型4就是我们最开始提到的中断溢出的中断。

从类型5到类型31在8086当中并没有定义，

但它说明了这些中断类型是CPU保留的。随着以后CPU功能的

丰富，将会使用这些中断类型号来提供新的中断服务。

那我们就来简单的看一看后来的CPU又有了哪些新的中断类型。

那在8086之后，新的中断类型在不断地添加进来。比如说类型6，

叫作未定义的操作码。CPU从内存当中取回一条指令编码后，

发现这个编码不属于当前指令系统当中定义的任何一条指令，

那CPU自然就不知道该做对应的什么样的操作。那最开始并没有约定遇到这样的

情况应该怎么办？后来增加了这个类型的中断之后，CPU只要遇到未定义的指令操作码，

就可以产生类型6的中断，并调用对应的中断服务程序。

那这个中断服务程序有可能就会在屏幕上打出一行字， 说遇到了一条未定义的指令，然后就停止当前运行的程序，

后者跳过这条指令继续执行。这就取决于中断服务程序如何去编写了。

那么后面还增加了一些其他类型的中断。

那有一些标了一个星号的是从386开始才有的，标了两个星号的是从486开始才有的。

这里有一些是对原先没有处理的异常情况进行了处理， 更多的则是根据CPU新增的功能添加了新的中断类型。

那随着指令系统体异结构进一步升级和CPU的改进，以后还会定义新的中断类型。

那这是中断向量表内容的变化。

我们再来看一看中断向量表存放的位置。这是现在比较流行的个人计算机的内部

结构。那它如果运行在实模式下，那就可以认为它是一个非常快的8086。

因此当CPU复位之后，也会去1兆地址空间的最高的16个byte的位置

去取第一条指令。那么这个地址会被南北桥芯片组引导到BIOS芯片。

那CPU执行BIOS芯片当中的指令，对主板上的各个设备进行基本的配置。

其中一项工作，就是在组成地址0的地方构建出中断向量表。

那所谓构建中断向量表，也就向主存中最低的那1K的字节填写那些中断向量。

那构建好了中断向量表，又准备好了中断服务程序之后，CPU再遇到中断

就可以访问主存中的中断向量表，并调用对应的中断服务程序了。

但是我们也知道，现在的CPU其实主要不是运行在实模式下，内存也并不是仅仅只有一兆了，

所以CPU对存储器的访问方式也发生了变化。

那我们以指令的寻址为例，在实模式下是用代码断计存器

CS和指令指针计存器IP进行组合。

这两个计存器都是16位的，它们的组合用断加偏移的方式产生一个

20位的地址。但是从386开始，指令指针计存器就从16位扩展到了32位。

也就是EIP计存器。那么它的寻址能力就有了2的32次方也就是4G个字节单元。

而从386开始，32位的CPU对外也就是32位的地址线，

能够寻址的范围也是2的32次方。

那在这时候，指令指针计存器的宽度和实际需要寻址的范围已经是一一对应的了。

所以在保护模式下，虽然逻辑地址还是写成CS计存器和EIP计存器的形式，

但如果你地址的产生方式已经和实模式完全不同了。

在保护模式下，段基址并不是存放 在CS计存器当中，而是存放在内存中的。

那么来看一看这时候CPU是怎么进行存储器寻址的。

那么在内存当中的某个地方，存放着一张表，

称为描述服务表。这张表一共有8192个表项，

每个表项由8个字节构成，就被称为一个描述符。这个描述符

当中，第2，3，4个字节和第7个字节这一共四个字节是基地址，

这个地址就对应了实模式下保存在CS计存器当中的内容。那这个描述符当中，

除了基地址之外，还有一些别的内容。例如断界限，它指明了这个段究竟有多长。

还有权限，这里面有若干个比特，指明了这个段的内容，是否可读，

是否可写等等。那这些描述符是存放在内存中的， CPU又是如何能访问到呢？所以在

CPU当中实际上是修改了CS计存器的使用方式。

我们可以想一想为什么总共有8192个描述符?

这和CS计存器的宽度是有关系的，CS计存器是16位宽的，

所以它一共可以寻址2的16次方个内存单元， 这就是64K。而因为每个描述符是8个字节，

所以8192个描述符刚好是64K。因此用CS计存器 就正好可以查找这么多个描述符。

但是这个描述符表存放的地址并不是从0开始的，

所以想要找到对应的描述符，CPU还得知道这个描述符表存放

的起始地址。所以在CPU当中还需要再设置一个新的计存器，叫"GDTR"，

它用于保存这张描述符表的起始地址。

那么又得问了？这个GDTR的内容是从哪里来的呢？ 实际是因为x86的CPU在启动的时候都会先进入实模式。

在实模式下，就会在内存的某一个地方，先把这张描述符表

都填好，然后将这张表的起始地址填到GDTR计存器当中去。

这也是CPU内部的一个计存器。只不过它不象EX，EDX 这些计存器可以用作数据的运算，

它是一个特殊的计存器，

但也是可以用特定的指令进行访问的。所以在保护模式下，CPU每次要访问存储器，都得先用

CS计存器的内容，加上GDTR计存器的内容得到一个地址，用这个地址去访问存储器，

然后取出这个描述符，再把这个描述符当中4个字节的基地址提取出来，

然后再和指令指针计存器EIP的内容 进行组合，从而得到要访问的存储器地址。

然后再用这个地址去访问存储器，得到想要的指令编码。

那既然在保护模式下，每一次存储器的访问都必须要经历这个 过程，那访问中断向量表也就不例外。而且不但如此，

中断向量表的位置也发生了变化。

在保护模式下，中断向量表也就没有放在地址为0的存储器区域了，

而是可以放在内存的任意地方。

而且它的名字也有了一些变化，叫作中断描述符表。

所以现在，在内存的某一个地方，存放的这张中断描述符表 总共有256个描述符。每个描述符是

8个字节，这8个字节当中，字节0和字节1，

以及字节6，字节7组合起来，一共是32位的地址。而字节2和字节3 则是一个段选择符。

那么当CPU发生中断时，还是会根据中断类型号来查找这个中断描述符表。

那因为现在中断描述符表的起始地址不是0， 所以CPU必须要先知道这个中断描述符表的起始地址。

那个这起始地址也保存在CPU当中的一个计存器， 称为IDTR，就是中断描述符表的地址计存器。

这个计存器的内容也需要由系统初始化软件在建立好中断描述符表后填写进去。

那现在，CPU就需要将中断类型号乘以8再加上IDTR计存器中的内容，

就可以得到对应的中断描述符的地址。然后把这个描述符取回之后，

将段选择符这16位存放到CS计存器当中，然后将地址对应的这 32位存放到EIP计存器当中。

我们注意这个操作的动作实际上和实模式下是类似的， 只不过实模式下每个中断向量是4个字节。

然后把其中两个字节放到CS计存器中，两个字节放到IP计存器中。

现在呢？每个中断描述符是8个字节，

那我们也是将其中的一部分放到CS计存器中，另一部分放到EIP计存器中。

那在完成了这个动作之后，如果在实模式下，下一个周期就直接可以从新的地址开始取址了。

而在保护模式下，则没有那么简单。

我们还需要按照刚才讲的那样，用CS计存器和GDTR计存器配合， 去内存中找出对应的段基值，

然后再和EIP计存器组合， 才能得到对应的内存地址。而那一个地址，才是我们需要调用的中断服务程序的入口

地址。直到那时，CPU才会从中断服务程序的入口 取回指令，真正开始中断的处理。

现在我们已经知道了， 这个表格随着这个手册的更新换代，也在不断的补充进新的内容。

而且在后来呢？还觉得这个表放在第一页不太好，给 它放到了这个手册当中的其他的位置。

那么这个呢？也需要用别的方法，来去标记这个表到底在什么位置。不然我们遇到异常情况不就

找不着了嘛。当然了，这部分内容呢不是我们的重点，只是简单地给大家讲解了一下。

如果有兴趣呢可以去深入地分析相关的资料。

804-中断的处理过程

那我们现在可以放心大胆地进行运算了。

算，算，算。一旦遇到了一个异常情况呢，我们就根据这个异常情况的类型，

去查找这个手册最前面的表格。上面可能是第四种类型，我们找到第四条， 上面写着操作方法在第十二页。

第十二页，好，找到了。那么按照这个操作方法呢一步一步往下执行，

就把这问题给解决了。然后呢，我们就可以继续刚才的运算了，对不对？

可是我刚才是在哪一页进行的运算？ 这在哪儿呢？我怎么回到刚才的运算呢？

我们先来看一看在CPU内部是如何检测中断的。

我们就以x86的实模式为例。在CPU内部， 会有中断的处理电路。如果在运算时发生了异常的情况，

比如出现除法错误，那运算器就会产生相应的中断信号。

那如果遇到其他指令产生的中断，就会产生其他对应的控制信号。

那这些中断都是和正在运行的程序本身有关的， 所以我们叫它软件中断，或者是内部中断。

那还有来自CPU外部的中断。

这些外设有可能是键盘、网络、打印机，等等。

那当键盘按键，或者有收发网络包的情况，

这些外设的芯片或者板卡就会通过主板上的物理连线，发出中断请求信号， 这些信号最终都会连接到CPU内部。

不管来源是哪里，CPU检测到了中断请求信号，就会进行处理。

中断处理的过程是这样的： 首先，CPU会关闭中断响应，也就是不再接受后面其他的外部中断请求了。

注意，只是不接收外部中断请求。

然后将发生中断处的这个指令的地址 保存到栈中，也就是内存当中的一个区域，

这个信息必须要保存好，以便于处理完中断后可以正确地返回当前的程序继续执行。

那究竟是保存发生中断的这条指令的地址， 还是发生中断的这条指令之后的一条指令的地址，

和这个中断具体的类型会有关系。

第三步则由CPU识别中断的来源， 确定中断类型号，从而能够找到相应的中断服务程序的入口地址。

这三步一般都是由硬件自动完成的。

第四步是保存现场，

也就将中断服务程序中可能会改变的寄存器先压栈，也就是放到内存中保存起来。

为什么要这么做呢？我们来看一个例子。在这个主程序片段中， 我们假设执行到这条触发指令的时候发生了中断，

那这时就会调用中断服务程序。

我们假设在这中断服务程序当中，有这么一条指令， 它会改写CX寄存器的值。

那如果我们没有对CX寄存器进行保护的话，在中断返回之后，CX寄存器的内容就已经- 改变了。

然后回到主程序继续往下执行时，这个CX寄存器 的内容，就不再是主程序中之前传到CX寄存器中的内容了。

这样就会导致主程序的错误。所以在中断服务程序中，如果要用到CX寄存器，

就应该先进行压栈，把当前CX寄存器的值保存到存储器中，

然后在中断返回之前，再执行弹栈的操作。这样中断返回之后，

CX寄存器的内容就没有变化，不会影响主程序的运行。

然后第五步，就是执行这个中断服务程序的主体内容。

而且在中断服务程序中，可以在适当的时候重新开放中断， 以便响应其他高优先级的外部中断。

以免中断服务程序本身要执行比较长的时间，造成有高优先级的外部中断长时间无法得到响应，

那如何在中断服务程序当中，重新开放中断呢?

那这就涉及到对标志寄存器的操作。在标志寄存器当中有一些标志位是状态标志。

状态标志一般情况下，是由硬件设置，然后由软件读取。

例如进位标志，就是在运算器产生进位时，由硬件自动设置，

然后由软件的指令读取出来，可能会进行相关的累加操作。

而另一类标志称为控制标志，这些标志

通常是由软件进行设置，然后硬件电路根据这些标志设置的不同，

而执行不同的功能。那么标志寄存器的第九位就是中断标志。

这个标志控制对可屏蔽中断的响应。那外部 中断分为两大类，一类是可屏蔽中断，一类是非屏蔽中断。

IF标志只对外部中断当中的可屏蔽中断起作用。

如果IF等于1，那就允许CPU响应可屏蔽中断请求； 如果IF等于0，那就不允许CPU响应可屏蔽中断请求。

那怎么设置IF标志位的值呢？有两条 指令。STI指令就是把IF位置为1，

而CLI指令就是把IF标志位清零。

这两条指令都是没有操作数的指令。

当然，IF指令对非屏蔽中断和内部中断都是不起作用的。

然后我们来看中断处理过程的最后一步，那就是恢复现场并且返回主程序继续运行。

那返回主程序就需要执行中断返回指令。

中断返回指令它可以不带操作数，

那么这条指令的操作，就是从当前的栈顶弹出三个字， 分别送到IP、CS和FLAGS寄存器当中去。

那么这条指令是放在中断服务程序的末尾，

那么在中断调用时，CPU中的硬件电路会将这三个寄存器的值 压入栈中，所以在执行中断返回指令时，

也会由硬件从栈顶弹出这三个字，再放回到这三个寄存器当中。

那当CS和IP寄存器改变之后， 下一条指令就会回到程序发生中断的地方继续执行了。

当然我们也注意到，这条中断返回指令操作的是CS和IP寄存器，

那在32位和64位的x86中，这个指令 指针寄存器，又被扩展为EIP和RIP寄存器，

它们的宽度都是不一样的，所以后来又有了不同的对应的指令。

我们再用一个图示来看一看刚才介绍的中断处理的过程。

我们就以内部中断为例。当CPU执行到某条指令，

例如就是这条指令，如果此时发生中断，CPU内部就会产生中断信号，

相关的中断处理电路会判断中断的来源，并产生中断类型号。

CPU的硬件电路会将CS和IP寄存器压栈， 这样就保存好了处理完中断后要返回的地址。

同时硬件上还会将FLAGS寄存器压栈，以便保存好当前的各项标志，

以免中断处理程序当中，有些指令会改变程序的标志位。

在硬件上还会清除IF标志位，以起到关中断的作用。

然后根据中断类型号，找到对应的中断向量，也就是新的CS和IP的值，

并以此更新CPU当中的CS和IP寄存器。

当完成这个操作后，CPU就会转到中断服务程序开始执行。

那么在中断服务程序中，也可以执行STI指令，以开放中断。

当完成了中断服务程序之后， 最后一条就是执行中断返回指令。

这条指令会从存储器当中将刚才压栈的三个字弹出来， 并按照对应的顺序，存到CS、IP和FLAGS寄存器当中去，

这样就完成了返回主程序的动作。

这就是中断处理过程的六个主要的步骤。通常情况下

前三步都是由处理中断的硬件电路来完成， 后三部则是由软件，也就是中断服务程序来完成。

但这只是一个大体的分工，在真的要涉及中断服务程序的时候，

必须要针对具体的系统，弄清楚软硬件的分工究竟是怎么样的。

例如在保存现场这件事情上，刚才我们介绍的，标志寄存器是由硬件来负责保存的，

但是在另外的一些系统上，可能硬件就不会自动地保存标志寄存器， 就需要由中断服务程序的软件来进行保存。

那么在设计中断服务程序时，必须要搞清楚这些问题，不然就有可能发生错误。

那现在呢，我们不但有了这张表，而且有了一个规范的操作流程，

我们知道从运算中出现异常情况，怎么查到这张表，怎么找到异常的操作的步骤，

然后呢应该事先做哪些记录，以免丢失信息。

最后我们还知道怎么回到我们刚才运算的那个地方， 继续进行下一步的操作。

这就是一个完整的中断的处理过程。

805-内部中断分类说明

现在，我们已经知道了中断处理的基本过程，

那么就来花一点时间，看一看内部中断到底有哪些不同的类型。

我们还是以x86的实模式为例，这个比较简单，但基本原理都是一样的。

在x86的实模式下，我们要来分析的内部中断就是这四个。

那这四个内部中断所使用的类型号，分别是0、1、3、 4，而加在中间的类型2，是留给外部中断的。

这个非屏蔽中断是外部中断的一种。

那么现在，就分别来看一看这四个内部中断，首先来看类型0的中断。

这个中断是和除法有关的， 我们不妨来回顾一下除法器的结构，在这个除法器支持的运算中，

是有一个64位的被除数，和一个32为的除数，

运算产生的商和余数，分别放到两个32位寄存器当中去， 那这样的设置对于多数除法运算来说，都是没有问题的。

但难免有一些比较特别的情况， 例如除数很小，比如说就是2，而被除数很大，那么一个很大的被除数

除以2之后，得到的商就会超过32位，没有办法放进这个商寄存器当中去。

当运算器遇到这种情况时，就会产生一个除法错中断，这个中断的类型号是0，

所以CPU就会去中断向量表中，取出0号中断向量，然后去执行对应的中断服务程序，

那由于这种情况，就是用0作除数， 这样得到的商应该是无穷大，肯定超过目标寄存器所能表示的范围，

所以这个除法错中断有时也会被称为除0中断。

然后我们再来看4号中断，这个中断叫作溢出中断，

也就是因为算术运算发生了溢出，而引起的中断，

这个中断的产生要借助一条特殊的指令，也就INTO指令，当执行这条指令时，

硬件电路会去检查溢出标志位OF是否位1， 如果为1，则会引起类型为4的内部中断。

那这条指令的格式是这样的，就是INTO，它是一个没有操作数的指令，

比如这条加法指令，在执行时，就有可能发生了溢出，那么运算器 在运行完这个加法后，会去设置标志寄存器当中的标志位，

也就是第11号溢出标志位， 但这个操作本身并不会引发中断，只是将标志位置1，

但如果之后执行了INTO指令，这条指令是会去检查OF标志位，

如果这时OF标志位为1，那就引起了4号中断，

但是如果INTO指令执行时，OF标志位为0，那就什么也不会发生，

这条INTO指令就相当于一条空操作指令， 所以INTO指令通常会安排在算术运算指令之后，

用来检查这个运算是否发生了溢出，并且， 在发生溢出时，就调用中断程序进行处理，

因为这是4号中断，所以我们也可以写成INT 4这样的形式。

注意INT和4之间有一个空格，这个4是INT指令的操作数。

实际上，任何一个类型的中断，都可以采用这样的形式进行调用，

用INT指令带上这个中断的类型号，那我们在后面还会看到这样的例子。

那我们要注意区分的是，这个4号中断， 和刚才介绍的0号中断，在引起中断的时机上是有区别的，

虽然它们都在检查运算时出现的异常情况， 但是0号中断，是在那条除法指令执行后，立刻发生的，

而4号中断则是要在编程时，加入INTO指令进行主动的检查，

因为很多时候，这样的加法运算的溢出，并不需要进行处理， 如果每一次溢出，都要引发中断，反而可能影响程序的性能，

所以在指令系统设计的时候，就把是否要检查这种溢出的情况，交给程序员来进行判断。

那么类型0和类型4这两个中断， 都是和运算结果出现了异常情况有关系的，

而另两个内部中断，则是主要用来进行错误调试的。

其中，类型1中断称为单步中断，要引发这个中断，

需要将标志寄存器当中的TF位置1，这时CPU就处于单步工作方式。

在单步工作方式下，CPU每执行完一条指令，

就会自动的产生一个类型1的中断，然后进入类型1中断服务程序。

这种工作方式主要是用来进行错误调试的，

比如说，你发现CPU执行一段程序有错误，但是又不清楚这个错误具体发生在什么地方，

那就可以将TF标志位置为1，在单步工作方式下进行调试。

通常情况下，我们会在这个类型1的中断服务程序当中， 将CPU当中的各个寄存器的内容，在屏幕上显示出来，

这样CPU每执行一条指令，我们就可以在屏幕上看见CPU当前正在执行的，是哪一条指令，

这条指令的地址是什么，执行这条指令的前后，那些通用寄存器又有什么样的变化，

这样我们就有可能发现，到底在哪一步，发生了不符合我们预期的行为。

这个方式对于调试是很有用的，但是CPU每执行完一条指令，就要产生一个中断， 那程序执行的速度，就是非常慢的，

如果想要调试一个很大的程序，仅用单步中断就会变得比较困难，

所以还有一个用于调试的中断，就是类型3，断点中断。

断点中断通常和单步中断配合使用， 在调试一个很大的程序时，一般我们会先通过断点中断，将错误定位在

这个程序的一小段代码中，然后，再对这一小段代码，用单步的方式，进行跟踪调试，

这样就可以大大提升调试的效率，这个思想也是很简明的，

如果我一个大的程序运算结果出现问题，我们并不会 马上从这个程序的开头，一条一条指令的顺序检查，

而通常会是将这个程序，切成几个大的部分，然后检查每一个部分的结果是否正确。

我们刚才提过，所有的中断，都可以用指令的形式来调用，

那么用INT加中断类型号的这个形式的指令，都是一个两字节的指令，只有断点中断， 是一个例外，INT

3指令是一条单字节长的指令，这就是INT 3指令的编码，

11001100， 那为什么INT n形式的指令都是两个字节的呢？我们只用想一想这个n要表达多大的范围，

我们一共有256个中断类型，所以这个n要表示0到255， 那要表示这些数，需要多少个二进制位呢？

需要8位，对吧，2的8次方，Jiu 是256， 那在前面，还得有一个字节的指令操作码，所以总共是两个字节。

那我们为什么要单独给3号中断，设置一个单字节长的指令编码呢？ 那这和它的使用方式是有关系的，

这个断点中断指令的使用，并不那么简单， 我们要在需要调试的程序当中，选择一个希望中断的位置，

然后用这条断点中断指令，去代替这个位置原有的指令， 当然，我们需要把原有的这条指令保存起来。

这个都是要由调试人员手工来完成的， 替换完以后，我们再次运行这个程序，那用户程序

运行到我们选好的这个中断点的时候，那它就执行了INT 3这条指令，

从而进入了对应的中断服务程序， 那我们就可以在这个中断服务程序当中，去将CPU的各个寄存器的值，都打印在屏幕上，

从而判断执行到这个断点的时候，这个用户程序是否还运行正常。

那如果运行正常，可能我们就需要把这个断点在往后挪一挪，

那如果这个时候，已经由寄存器的值不符合我们的预期了， 那我们就需要将断点，放到更靠前的位置，进行进一步的检查。

但我们还得记得，在这个中断服务程序当中， 需要将这个断点位置，与原有的那条指令的编码，再替换回去，

并且将指令指真寄存器的值，再回退一个字节，

也就是指向这个原有的指令，以保证中断返回之后，CPU能从断点的这个地方继续执行。

我们来看一个例子，假设这是我们要调试的一段程序，这里有5条指令，

左边是它们对应的指令的地址，其中有一些是两个字节的指令，有一些是一个字节的指令。

那如果我们想选择这条INC指令，作为断点， 那么就需要把这条指令的编码，替换成INT

3指令的编码， 这时候就体现出了INT 3这条指令，是一个单字节指令的好处，

因为x86的指令当中，最短的就是一个字节的指令，就像这条INC指令，

那如果断点中断指令，是一个两字节的指令，那么在替换进来之后，就会影响到后续的指令，

而后续的指令，却有可能在这个断点之前执行，比如说就像这段程序代码，

在这个断点之前，就有一个转移指令，直接跳到了断点之后，然后经过条件判断，可能又跳- 转回来，

才继续执行到断点的地方，所以我们将这条INC指令替换成断点中断指令的时候，

一定不能影响后续的指令，这也就是为什么断点中断指令必须要是一个字节的。

那么在这段程序执行的过程中，如果这个条件已经发生了，

那就会运行到断点中断指令，然后CPU内部就会发生中断， 转而去执行3号中断向量所对应的中断服务程序，

那在这个中断服务程序中，我们就可以把AL寄存器的内容打出来，

这样调试人员就可以观察到，这个时候AL寄存器的内容，是否符合我们的预期了，

如果我们发现AL寄存器的内容有错误，那么就可以再次运行这个程序，

并在附近的位置设置CPU进入单步工作模式，进行单步调试， 这样就会比较容易的发现一些隐藏的很深的错误。

那么介绍了这些内部中断，都有共同的特点。首先，它们的中断类型号

是由CPU内部产生的，因为这些异常的情况，就是CPU自己在执行指令的过程中发生的，

所以它是知道到底发生的是什么类型的中断， 而我们后面要介绍的外部中断，则有可能来自不同的外部设备，

所以CPU需要去读取外设，以得知中断类型号，

这是第一个区别。第二个区别，是屏蔽的方式，

那么在内部中断当中，除了单步中断以外，都不可以用软件的方式来进行屏蔽，

也就是，我们不可以通过设置IF这个标志位，来让CPU不响应内部中断。

第三个是优先级，也就是内部中断 和外部中断同时发生时，CPU先处理哪个中断，

那么除了单步中断以外，所有的内部中断优先级，都比外部中断高。

CPU总是优先处理自己内部发生的异常情况。

现在，我们已经了解了内部中断的基本类型，

有两个是用来处理运算的异常情况，还有两个，是CPU用来调试的，

那后来，随着内部中断的类型不断的增加，其中增长的大部分，

都是CPU用来调试和管理用的中断。

806-基于中断的功能调用

那现在，我再执行这些运算之后， 也就是执行这些程序的过程中呢，又遇到了一个奇怪的现象。

这里突然有一个地方写着，请查看紧急操作手册的第二百项，

那我就翻到这个地方去了，翻到前面，然后找第二百项，

然后找到了对应的操作，这个呢，是让我在这个

本子的某一个地方写一个数，这是怎么回事呢？ 一般来说，

刚才我们学到的，都是我在运算当中，遇到了异常的情况，

然后我主动去查找前面的，对应的异常处理的方法，查找中断向量表。

那么现在在这个程序当中，居然主动的写到了， 让我去查找相关的中断项量表，这个是怎么回事情呢？

我们就来看看这个问题。

在x86指令系统当中，其实还提供了一条中断指令， 它的格式是INT，加上一个操作数n，

用这条指令，就可以直接调用对应的中断服务程序， 这个n是0到255当中的一个数，对应的中断类型码，

当CPU执行到指令时，首先会将标志寄存器压栈，

然后清除IF和TF两个标志位，也就是关中断。

再将CS和IP两个寄存器的内容压栈， 然后根据指令中提供的类型码，也就是这个n，

去查找中断向量表，找到对应的中断服务程序的入口地址。

再将入口地址装到CS和IP寄存器当中去，

这样下一步CPU就会到中断服务程序的入口取出下一条指令，继续执行。

那么看到这一个步骤，和我们之前介绍的，CPU处理中断的过程是一样的， 但是区别在于，这是由运行指令主动触发的。

比如，之前CPU当中，如果发生的1号中断， CPU就会来中断向量表当中，取出1号中断向量，

然后转向对应的中断服务程序开始执行， 但如果当前CPU执行程序的过程当中，并没有满足触发1号中断的条件，

而是直接写了一条指令，INT 1

CPU也会到中断向量表当中，取出1号中断向量， 装入CS，IP寄存器，再转到1号中断服务程序，开始执行。

那这样做有什么意义呢？明明没有发生中断，我们为什么要调用这个中断服务程序， 那这种中断指令的使用场景其实有两类，

一类就是CPU的一些专用的中断，就是需要通过调用指令的方式来实现的， 比如说3号中断，那就得写INT

3这样的指令，才可以产生。

这种情况我们在之前已经介绍过了，现在我们来看另外一种情况。

比如说有一类，我们称为BIOS中断，BIOS就是基本输入输出系统， 这是一套不大但是还挺复杂的程序，

存放在ROM当中， 寄存器在刚通电或者复位时，CPU就会从BIOS ROM当中取出第一条指令开始执行，

所以在BIOS当中会提供系统加电自检，和主要的输入输出设备处理程序，等功能模块。

所谓的功能模块，也就是一个一个具有独立功能的程序， 比如从键盘接收一个字符，或是在显示器上显示一个字符，

这些工作不但在系统初始化时需要使用，在后来用户使用计算机的过程中，其实也会用到，

而如果要用户自己去编写，处理这些输入输出设备的程序，那就太麻烦了，

所以BIOS的设计者，就将其中的这些功能各个包装起来，形成了很多个独立的功能模块，

然后将这些独立功能模块的入口地址，放在中断的向量表当中，

那如果我们想使用这些功能，比如说就像在显示器上显示一个字符，

用户就不用去关心，到底用的是什么显示器，要占用多少个像素，等等。

那只需要使用BOIS提供的功能模块，用INT指令调用对应的中断服务程序就可以了。

而且我们可能希望，向这些功能模块传递一些参数，那就可以通过寄存器进行传递，

我们不妨来看一个例子，这个表是BIOS中断的一个片段，

那想使用BOIS中断，首先就得查找BIOS中断的手册，这个手册

一般会提供这样一个表格，列出了BIOS这些功能模块所在用的中断号，

比如说10H，就是用于在显示器上进行显示的一个中断服务程序，

而1AH，则是设置系统时钟的一个中断服务程序。

我们以1AH为例，那如果我们想要改变现在的系统时钟，当然我们可以去分析时钟管理

芯片的功能，通过查找它的手册，来分析如何去改变系统时钟的设置，这可能要花很多个步骤。

那BIOS的设计者，就帮我们封装好了这个功能， 我们只要这样写代码就可以了。

因为我们通过这个表可以看到，如果要设置时钟，我们需要提供一个功能号为1，

因为1AH这个中断里头，其实有多种功能， 我们可能想读出当前时钟的值，也可能要改变当前时钟的值，

这个中断服务程序怎么识别呢，它就要求你在AH寄存器当中放入一个数，

那么在中断服务程序的开始，会先检查AH寄存器，如果里面是0，那它就按照读

时钟的操作，运行后续的代码，如果AH里面的值是1，它就按照设置时钟的操作，执行后续- 的代码，

那现在我们要设置时钟，所以在AH里面先放上1，

然后我们查这张表，知道我们要设置的时间是放在这几个寄存器当中，CH

放在要设置的小时数，CL放要上要设的分钟数，DH是秒，DL是百分之一秒，

而CL和CH组成的寄存器是CX，DL和DH组成的寄存器是DX，

所以我们直接可以通过对CX和DX赋值，来设置这个时间， 那么现在为了简单，我们就设成0点0分0秒，

这些参数准备好以后，我们最后写INT 1AH， 接下来就像是之前介绍过，发生中断的时候一样，

CPU会去中断向量表当中，找到1AH对应的中断向量，然后转移到对应的

中断服务程序开始执行，而这段中断服务程序就是位于BIOS所在的存储区域，

那在这个中断服务程序当中，就会去操作管理 系统时钟的芯片或者部件，完成时钟的更改，

然后再返回到这个主程序当中，继续执行下面的代码。

那这样就把一个和底层硬件细节非常相关的操作给 封装起来，让编程人员可以比较轻松的完成这样的工作，

那BIOS中断其实提供了很多种类型， 可以完成相当丰富的功能，有兴趣的话，可以查找相关BIOS的手册，

但是因为BIOS容量有限， 因此我们还是可以利用这个方法，在上层的软件当中，提供更为丰富的功能，

那我们来介绍一个例子，就是DOS中断， DOS早起的一种操作系统，它占用了一个中断类型号，21H，

和BIOS占据了多个中断类型号不同，DOS中断只有这一个类型号， 但它的功能非常的丰富，常用的文件管理、

存储管理等很复杂的功能，都可以种这个中断服务程序来解决，

那它怎么区分我们到底想使用哪个功能呢？其实刚才BOIS中断给我们提供了这样的思路，

我们可以通过一个寄存器，传入一个参数，来告诉中断服务程序，我们到底想调用那样的功能，

所以所有的DOS中断都只使用这一个中断入口， 而且DOS是一个操作系统，

它所提供的中断功能，比BIOS中断更为齐全、完整， 而且进一步品比了设备的物理特性，让编程的使用，变得更加的方便。

我们也来看一个例子，如果我们想在屏幕上输出一个 $字符，那我们可以查找DOS中断所提供的表格，

DOS中断都是21H，所以这个表里面只需要列出功能号， 那么发现，6号功能是在进行输入输出的操作，

所以我们现在AH寄存器当中，存入6，然后我们进一步发现，

如果我们想输出一个字符，就在DL寄存器当中，放入我们想显示的这个字符，

而如果我们想通过键盘输入这个字符，则只需要在DL寄存器当中，存入FF，

而最后输入的字符，会放在AL寄存器当中，那我们现在还是来看输出，

所以我们在DL寄存器当中，存上$这个字符， 然后调用INT

21H， 这样CPU就会转向21H号中断所对应的中断服务程序，

在这个服务程序当中，首先会检查AH里面的值，确定功能号， 然后就进入到这个功能对应的程序代码段，

再根据DL寄存器的内容，判断出这是一次输出，那这个服务程序接下来就会对显示器进- 行操作，

让对应的字符显示在屏幕的合适的位置， 那这些繁琐的工作，都不需要用户来关心了，

只要简单的调用这个DOS中断就可以了。

当然前提是，这些操作都已经由其它的程序员帮你写好， 并且封装起来，还安装在了你这个电脑上，这样你才可以使用，

归根到底，这些操作都还是要有人来写出程序， 只不过一次写好之后，其他的用户就可以直接调用了，

这样会很方便，所以说无论是BIOS中断，还是DOS中断，或者是其它类似的中断方式，

其本质并不是计算机运行当中发生了异常的情况， 而是利用了现有的中断这种机制，来实现一些系统函数，

代码的调用，以便向高层的软件屏蔽底层硬件的细节， 从而提高编程的便利性，正确性，和可抑制性。

那现在，我们的机制已经讲的很清除了，

在程序执行的过程中，遇到的任何异常情况，我们都可以自如的处理， 而且还可以利用这个机制，完成更加丰富的功能，

所以可以说，我们对任何情况，都已经是能够应付裕如了。