第 2 章 VMX 架构基础

为了支持虚拟机环境，在计算机结构体系中的 CPU 域和 PCI 总线域上，Intel 提供了三个层面的虚拟化技术（Intel Virtualization Technology）。

（1）基于处理器的虚拟化技术（Intel VT-x）：全称为 Virtualization Technology for x86，在处理器上实现了虚拟化技术，它的实现架构是 Virtual-Machine Extensions（VMX）。在 VMX 下引入了两种处理器模式，即 VMX root 与 VMX non-root，分别支持 host 与 guest 软件的运行，使 guest 软件能直接在处理器硬件上运行。

（2）基于 PCI 总线域设备实现的 I/O 虚拟化技术（Intel VT-d）：全称为 Virtualization Technology for Directed I/O，这个虚拟化技术实现在芯片组（Chipset）层面上，提供了诸如 DMA remapping 与 Interrupt rempping 等技术来支持外部设备 I/O 访问的直接虚拟化。

（3）基于网络的虚拟化技术（Intel VT-c）：全称为 Virtualization Technology for Connectivity，部署在 Intel 的网络设备上，这也是基于 PCI 总线域设备的虚拟化技术。

根据 Intel 的介绍，在所有 Intel 的 10 Gigabit Server Adapter 和部分 Gigabit Server Adapter 上，VT-c 提供了两个关键的技术：Virtual Machine Device Queues（VMDq）和 Virtual Machine Direct Connect（VMDc）。Intel server adapter 上的 VMDq 可以做到为各个 guest OS 处理分类好的 packet，能明显地提高 I/O 吞吐量。而 VMDc 使用 PCI-SIG Single Root I/O（SR-IOV）技术，允许虚拟机直接访问网络 I/O 硬件改善虚拟化性能。

在处理器虚拟化方面，Intel 也为 Itanium 平台提供了虚拟化，即 Intel VT-i。本书中所讨论的虚拟化技术主要是基于处理器虚拟化（Intel VT-x）中的 Virtual-Machine Extensions（VMX）架构。关于 VT-d 技术读者可以下载《Intel Virtualization Technology for Directed I/O Architecture Specification》文档，它并不在本书讨论的范围。VT-d 技术在桌面平台上并不是每款处理器都提供，VT-c 技术提供在服务器平台上。

2.1　虚拟化概述

虚拟机的运用需要以处理器平台提供 Virtualization Technology（VT，虚拟化技术）作为前提，是对资源的虚拟化管理的结果。在虚拟化技术出现之前，软件运行在物理机器上，对物理资源进行直接控制，譬如设备的中断请求，guest 的内存访问，设备的 I/O 访问等。软件更改这些资源的状态将直接反映在物理资源上，或者设备的请求得到直接响应。

在 CPU 端的虚拟化里，实现了 VMX（Virtual-Machine Extensions，虚拟机扩展）架构。在这个虚拟机架构里，存在两种角色环境：VMM（Virtual Machine Monitor，虚拟机监管者）和 VM（Virtual Machine，虚拟机）。host 端软件运行在 VMM 环境里，可以仅仅作为 hypervisor 角色存在（作为全职的虚拟机管理者），或者包括 VMM（虚拟机监管者）职能的 host OS。

guest 端软件运行在 VM 环境里。一个 VM 代表着一个虚拟机实例，一个处理器平台可以有多个虚拟机实例存在，由于 VM 里的资源被虚拟化，每个 VM 是彼此独立的。

guest 软件访问的资源受到 VMM 的监管，guest 希望修改某些物理资源时，VMM 返回一个虚拟化后的结果给 guest。例如，guest 软件对 8259 中断控制器的状态进行修改，VMM 拦截这个修改，进行虚拟化操作，实施修改或者不修改 8259 中断控制器的物理状态，反馈一个不真实的结果给 guest 软件。

图 2-1 展示了前面提及的虚拟机环境里的三种虚拟化管理：

（1）设备中断请求

（2）guest 内存访问

（3）设备的 I/O 访问

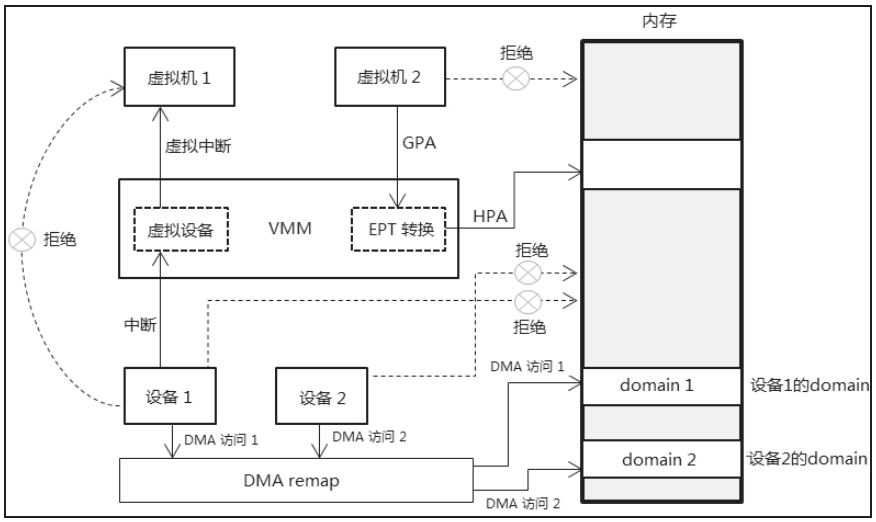

图 2-1

设备的中断请求经由 VMM 监管，模拟出虚拟设备反馈一个虚拟中断给 guest 执行，在这个模型里，设备中断请求不直接发给 guest 执行。而 guest 访问的物理地址也不是最终的物理地址，而是经过 EPT 进行转换才得到最终物理地址。设备 1 和设备 2 使用 DMA 访问时，它们最终的目标物理地址经过 VT-d 技术的 DMA 重新映射功能映射到属于自己的 domain（区域）。

2.1.1 虚拟设备

在设备发生中断请求时，这就产生了设备的 ISR（中断服务例程）是由 VMM（或 host OS）处理，还是直接由 VM（guest OS）处理（或由 VMM 转发给 VM）的问题，即 ISR 是运行在 host 还是 guest 端的问题。

依赖于 VMM 对外部设备所有权的设计产生了两种模型：

- 设备是属于 host 所有，物理设备的 ISR 由 VMM 来执行。
- 设备分配给 VM 使用，那么 ISR 将在 guest 环境里运行。

host 和 guest 都有自己的 IDT（中断描述符表），host vector 对应着物理设备 ISR 在 host IDT 的位置，而 guest vector 对应着 VMM 反馈给 guest 的中断请求在 guest IDT 上的位置。中断请求通过 VMM 给 VM 使用 injection event（注入事件）的手段来实现。

host vector 和 guest vector 并不相等，但在设备分配给 VM 使用的模型里，如果外部中断的控制权在 guest 手中，那么 guest vector 可以对应物理设备 ISR 在 guest IDT 的位置，也就是物理设备 ISR 完全由 guest 执行，而 VMM 并不监管或转发。

如果外部中断的控制权需要掌握在 VMM 手中，通过开启“external-interrupt exiting”功能，并且结合“acknowledge interrupt on exit”位的设置来实现。当发生外部中断时，VM 停止工作，处理器控制权切换回 VMM。

在设备所有权归 VMM 的模型里，虚拟设备应运而生，它是 VMM 在物理设备之上抽象出的虚拟设备概念。中断发生时，VMM 的 vector 是物理设备对应在 host IDT 里，VMM 执行物理设备的 ISR。然后模拟 guest 的 ISR 处理流程，如发送 EOI 命令给中断控制器，更新中断控制器状态。根据对应的 guest vector 注入一个外部中断给 guest 在自己的 IDT 找到 ISR 执行，这个 guest ISR 可能并不能做实际的收尾工作，如发送 EOI 命令（被 VMM 接管）或更新中断控制器。

host vector 代表着来自平台的物理设备，而 guest vector 代表着来自虚拟设备，这两个 vector 值不一致，这个虚拟设备由 VMM 维护而代表着物理设备在 VM 环境中的名字。虚拟设备可能并不存在实体功能，只是逻辑表述上的抽象概念。

在设备分配给 VM 使用的模型里，guest vector 与 host vector 可能一致也可能不一致，但物理设备的 ISR 由 guest 去运行，代表着 ISR 的收尾工作由 guest 去执行。VMM 截取中断请求后根据 guest vector，同样使用事件注入手段转发外部中断给 guest 执行，而 VMM 可能并不做其他工作。这是在 host 夺取外部中断控制权的前提下，前面所述在

guest 掌控外部中断权时，设备 ISR 直接在 guest 里执行。

当然，VMM 也可以不拦截外部中断，这样外部中断就直接通过 guest-IDT 进行 deliver 执行，而不需要经过 VMM 转发。

2.1.2　地址转换

host 软件与 guest 软件都运行在物理平台上，需要 guest 不能干扰 VMM 的执行。譬如，guest 软件访问 100000h 物理地址，但这个物理地址可能属于 host 的私有空间，或者 host 也需要访问 100000h 物理地址。VMM 的设计需要 guest 不能访问到这个真实的物理地址，VMM 通过 EPT（Extend Page Table，扩展页表）来实现“**guest 端物理地址到 host 端物理地址**”的转换，使得 guest 访问到其他的物理区域。

也因为，物理平台上可能有多个 VM 在运行，例如虚拟机 1 与虚拟机 2 也可能会访问到同一个物理区域。为保证每个 VM 的物理地址空间隔离，保持独立性，也需要将 guest 的内存访问进行转换。内存虚拟化是一项很重要的工作，引进的 EPT（扩展页表）机制是实现内存虚拟化的重要手段。EPT 的实现原理和分页机制里的转换页表一样，经过多级转换产生最终的物理地址。

在开启 EPT 机制的情况下，产生了两个地址概念：**GPA**（Guest Physical Address）和 **HPA**（Host Physical Address），HPA 是真正的物理地址。guest 软件访问的物理地址都属于 GPA，而 host 软件访问的物理地址则属于 HPA。而在没启用 EPT 机制的情况下，guest 软件访问的物理地址就是最终的物理地址。

这里还有另一个重要的概念：**guest paging-structure table**（guest 的页结构表），也就是 guest 内保护模式分页机制下的线性地址到物理地址转换使用的页表。这个页表项内使用的物理地址是 GPA（例如 CR3 的页目录指针基址），而 **EPT paging-structure table**（EPT 页表结构）页表项使用的是 HPA。

2.1.3　设备的 I/O 访问

在由 CPU 发起访问外部设备时，需要通过 PCI/PCIe 总线的内存读写事务，I/O 读写事务或配置读写事务进行。当 guest 软件发起这样的访问时，VMM 可以通过内存虚拟化和 I/O 地址虚拟化达到虚拟化设备的目的。

当由设备主动发起访问内存时（即 DMA 读写事务），在 DMA 读写下 CPU 不参于其中的执行过程，也就不能为这个读写内存提供地址转换，VMM 单纯依赖 CPU 来监控设备的访问是比较困难的。

于是，基于 PCI 总线域的虚拟化需要提供，VT-d 技术正是为了解决这些而提出的。VT-d 的其中一个重要功能是进行 DMA remapping（DMA 重新映射），DMA remapping 机制在芯片组或 PCI 设备上实现地址转换功能，其原理和分页机制下的虚拟地址转换到物理地址是相似的。

DMA remapping 需要识别设备的 source-id，这个 source-id 代表着发起 DMA 访问设备（即 requester，请求者）的身份，实际上它就是 PCI 总线域的 ID，由 bus、device 及 function 组成。根据 source-id 找到设备对应的页表结构，然后通过页表进行转换。

如图 2-2 所示，首先，一个被称为 Root-entry table 的结构需要在内存中构造，由 Root-entry 指出 Context-entry table，再由 context-entry 得到页表结构，最后经页表得到最终的物理地址。

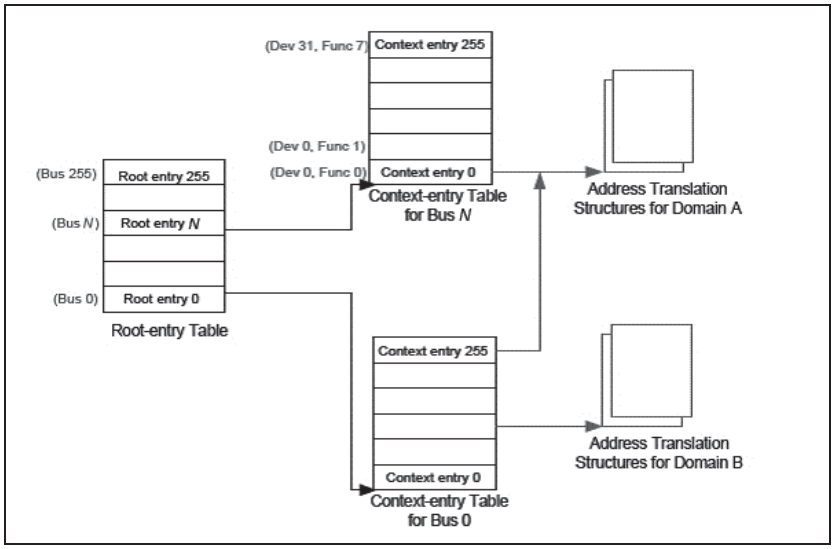

图 2-2

Root-entry table 的基址被提供在 MCH（Memory Control Hub）部件，一般位于 Bus0、Device0、Function0 设备（内存控制器）扩展空间里的 MEREMAPBAR 寄存器，在这个寄存器提供一个 remap MMIO 空间，其中的 RTADDR_REG 寄存器提供 Root-entry table 指针值，但是对于不同的处理器，这个 remap MMIO 空间的基址也存放在不同的位置。

DMA remapping 提出了一个 domain（域）的概念，实际上就是为设备在内存中分配一个对应的区域。例如，为设备 1 分配 domain 1，为设备 2 分配 domain 2，它们访问各自独立的区域。这个 domain 值可能被用作标记，处理器使用它来标记内部的 cache。当然，不同的设备也可以访问同一个 domain，但必须使用同一个 domain 值。

2.2 VMX 架构

Intel 实现了 VMX 架构来支持 CPU 端的虚拟化技术（Intel VT-x 技术），引入了一种新的处理器操作模式，被称为 **VMX operation**。也加入了一系列的 VMX 指令支持 VMX operation 模式。

2.2.1　VMM 与 VM

在 VMX 架构下定义了两类软件的角色和环境：

- VMM（Virtual Machine Monitor，虚拟机监管者）
- VM（Virtual Machine，虚拟机）

VMM 代表一类在 VMX 架构下的管理者角色，它可以是以 hypervisor 软件形式独立存在，也可以是在 host OS 中集成了 VMM 组件，也就是在 host OS 中提供了虚拟机管理者的职能。软件运行在 VMM 环境下拥有物理平台的控制权，并且监控每个 VM 的运行。

VM 代表着虚拟机实例，VMM 在需要执行虚拟机实例代码时，需要进入 VM 环境。在一个 VMX 架构里可以有多个虚拟机实例。每个 VM 像是在一个独立的物理机器环境里，有自己的内存、中断甚至设备等。但 VM 本身并不会意识到自己运行在虚拟的环境里，VM 里的资源实际上掌控在 VMM 手中，由 VMM 仿真出一些资源反馈给 VM。

host 端软件可以是 VMM 或者 host OS，而 guest 端软件就是运行在 VM 实例环境里。guest OS 就像运行在普通的物理机器上，执行自己的应用程序、kernel 组件及 driver 组件等。

在虚拟化平台设计里，VM 不能影响到 VMM 软件的执行，并且每个 VM 之间也要确保独立互不干扰，这些工作需要 VMM 这个管理者对 VM 进行一些监控以及配置。VMM 软件监控着每个 VM 对资源的访问，并限制某些资源访问。典型的，VMM 可以允许或拒绝某个 VM 环境响应外部中断请求。又如，当 VM 里的 guest 软件发生#PF（Page Fault）异常时，VMM 接管并分析#PF 异常产生的原因，进行或者不进行某些处理，然后反射回 guest 执行属于自己的#PF 异常处理。

2.2.2　VMXON 与 VMCS 区域

在 VMX 架构下，至少需要实现一个被称为“**VMXON region**”，以及一个被称为“**VMCS region**”的物理区域，VMXON 区域对应于 VMM，VMM 使用 VMXON 区域对一些数据进行记录和维护。而每个 VM 也需要有自己对应的 **VMCS**（Virtual Machine Control Structure，虚拟机控制结构）区域，VMM 使用 VMCS 区域来配置 VM 的运行环境，以及控制 VM 的运行。

在进入 VMX operation 模式前，必须先为 VMM 准备一份 VMXON 区域，同样在进入 VM 前也必须准备相应的 VMCS 区域，并配置 VM 的运行环境。一个 VM 对应一份 VMCS 区域，因此，VMX 平台上有多少份 VM 实例就需要维护多少份 VMCS 区域。

处理器进入 VMX operation 模式需要执行 VMXON 指令，一个指向 VMXON 区域的物理指针被作为 VMXON 指令的操作数提供。而进入 VM 之前必须先使用 VMPTRLD 指令加载 VMCS 指针，VMCS 指针指向需要进入的 VM 所对应的 VMCS 区域。

在 VMX operation 模式下，处理器会自动维护 4 个指针值：

- **VMXON pointer**，指向 VMXON 区域，有时也可以被称为 **VMX 指针**。
- **current-VMCS pointer**，这是当前所使用的 VMCS 区域指针，VMCS pointer 可以有多个，但在一个时刻里，只有唯一一个 current-VMCS pointer。
- **executive-VMCS pointer** 及 **SMM-transfer VMCS pointer**。

Executive-VMCS 指针与 SMM-transfer VMCS 指针使用在开启 SMM dual-monitor treatment（SMM 双重监控处理）机制下。在这个机制下，executive-VMCS 指针既可以指向 VMXON 区域，也可以指向 VMCS 区域。SMM-transfer VMCS 指针指向切入 SMM 模式后使用的 VMCS 区域。

2.2.3 检测 VMX 支持

软件应通过检查 CPUID.01H:ECX[5].VMX 位来确定是否支持 VMX 架构，该位为 1 时表明处理器支持 VMX 架构。可以实现如代码片段 2-1 所示的独立的函数来检测。

代码片段 2-1：

```
;-------------------------------------------------
; support_intel_vmx()
; input:
;       none
; output:
;       1 - support, 0 - unsupport
; 描述:
;       1) 检查是否支持 Intel VT-x 技术
;-------------------------------------------------
support_intel_vmx:
        ;;
        ;; 检查 CPUID.01H:ECX[5].VMX 位
        ;;
        bt DWORD [gs: PCB.FeatureEcx], 5
        setc al
        movzx eax, al
        ret
```

或者在进入 VMX operation 模式前进行 VMX 支持的检查，第 2.5 节将介绍更多关于“VMX 支持能力检查”的描述。

2.2.4 开启 VMX 进入允许

要开启 VMX operation 模式，必须先开启 CR4.VMXE 控制位，该控制位也表明处理器允许使用 VMXON 指令，但其他的 VMX 指令则必须在进入 VMX operation 模式后才能使用。

代码片段 2-2：

```
        ;;
        ;; 检测是否支持 VMX
        ;;
        bt DWORD [ebp + PCB.FeatureEcx], 5
        mov eax, STATUS_UNSUCCESS
```

```
        jnc vmx_operation_enter.done

        ;;
        ;; 开启 VMX operation 允许
        ;;
        REX.Wrxb
        mov eax, cr4
        REX.Wrxb
        bts eax, 13                                ; CR4.VMEX = 1
        REX.Wrxb
        mov cr4, eax
```

上面是实现进入 VMX operation 模式前的一段代码，在检查处理器支持 VMX 后，置 CR4.VMEX[13]位将允许进入 VMX operation 模式。此时，可以执行 VMXON 指令。在 CR4.VMEX=0 时，执行 VMXON 指令将会产生#UD 异常。

代码中的 REX.Wrxb 定义为一个宏，在定义了__X64 符号的情况下有效，它嵌入了一个 REX prefix 字节，也就是 48H 字节，在 64 位代码下使用 64 位的操作数（关于这方面的知识请参考第 1.2 节）。

2.3　VMX operation 模式

前面提及，在 VMX 架构下，处理器支持一种新的 VMX operation 模式。VMXON 指令只能在开启允许进入 VMX operation 模式后才能使用，其余的 VMX 指令只能在 VMX operation 模式下使用。VMX operation 模式里又分为两个操作环境，以支持 VMM 与 VM 软件的运行。

- VMX **root** operation
- VMX **non-root** operation

VMX 模式的 **root** 与 **non-root** 环境可以理解为：VMX 管理者与 guest 用户使用的环境。因此，VMM 运行在 VMX root operation 环境下，VM 则运行在 VMX non-root operation 环境下。

从 root 环境切换到 non-root 环境被称为“**VM-entry**”，反之从 non-root 环境切换回 root 环境被称为“**VM-exit**”。当软件运行在 VMX root operation 环境时，处理器的 **CPL**（Current Privilege Level）必须为 0，拥有最高的权限，可以访问所有的资源，包括新引进的 VMX 指令。

在 VMX non-root operation 环境里，当前的 CPL 值不必为 0。根据 VMM 的相应设置，guest 软件的访问权限受到了限制，部分指令的行为也会发生改变。在 VMX non-root operation 模式下，guest 软件执行任何一条 VMX 指令（除 VMFUNC 指令外）都会导致被称为“**VM-exit**”的行为发生。另外，软件执行 MOV 指令对 CR0 寄存器进行设置时，写 CR0 寄存器的行为将发生改变，导致 CR0 寄存器的值允许写入或被拒绝修改。

2.3.1 进入 VMX operation 模式

如前面所述，在启用处理器的虚拟化机制前，必须先开启 VMX 模式的允许（开启 CR4.VMXE 位），表明允许执行 VMXON 指令进入 VMX operation 模式。

在执行 VMXON 指令切换到 VMX operation 模式之前，还需要做一系列的准备工作，下面的代码片段摘自 lib\Vmx\VmxInit.asm 文件里的 vmx_operation_enter 函数，这里简要地了解一下。

代码片段 2-3：

```
    ;;
    ;; 检查是否已经进入了 VMX root operation 模式
    ;;
    test DWORD [ebp + PCB.ProcessorStatus], CPU_STATUS_VMXON
    jnz vmx_operation_enter.done
    ;;
    ;; 检测是否支持 VMX
    ;;
    bt DWORD [ebp + PCB.FeatureEcx], 5
    mov eax, STATUS_UNSUCCESS
    jnc vmx_operation_enter.done

    ;;
    ;; 开启 VMX operation 允许
    ;;
    REX.Wrxb
    mov eax, cr4
    REX.Wrxb
    bts eax, 13                                ; CR4.VMEX = 1
    REX.Wrxb
    mov cr4, eax

    ;;
    ;; 更新指令状态，允许执行 VMX 指令
    ;;
    or DWORD [ebp + PCB.InstructionStatus], INST_STATUS_VMX

    ;;
    ;; 初始化 VMXON 区域
    ;;
    call initialize_vmxon_region
    cmp eax, STATUS_SUCCESS
    jne vmx_operation_enter.done

    ;;
    ;; 进入 VMX root operation 模式
    ;; 1) operand 是物理地址 pointer
    ;;
    vmxon [ebp + PCB.VmxonPhysicalPointer]

    ;;
    ;; 检查 VMXON 指令是否执行成功
    ;; 1) 当 CF = 0 时，VMXON 执行成功
    ;; 1) 当 CF = 1 时，返回失败
    ;;
    mov eax, STATUS_UNSUCCESS
```

```
    jc vmx_operation_enter.done
    jz vmx_operation_enter.done
```

上面代码的处理如下：

（1）首先检查处理器是否已经进入 VMX operation 模式。是则直接返回，否则进行下一步。处理器 PCB（处理器控制块）中的 ProcessorStatus 的值记录着当前处理器的状态。

（2）检测是否支持 VMX 架构，这里并不在其他地方使用独立的函数进行检查。

（3）开启 CR4.VMXE 控制位，表明允许进入 VMX operation 模式。详见第 2.2.4 节。

（4）其中重要的一步是调用 initialize_vmxon_region 函数来初始化 VMXON 区域。

（5）接着执行 VMXON 指令，提供一个 VMXON 区域的物理指针作为操作数，这个指针被称为 VMXON 指针，注意这个指针使用的是物理地址。

（6）紧接着对 VMXON 指令进行检查看是否执行成功，分别检查 CF 与 ZF 标志位，当 CF=1 或 ZF=1 时，表明操作失败。

VMXON 指令执行成功后，表明处理器此时已经进入 VMX operation 模式，也就是已经处于 root 环境，行使 VMM 的管理职能。

2.3.2　进入 VMX operation 的制约

执行 VMXON 指令的基本要求是：需要开启 CR4.VMXE 位，不能在实模式、virtual-8086 以及兼容模式下执行，否则将产生#UD 异常；不能在非 0 级权限下执行，否则将产生#GP 异常。

2.3.2.1　IA32_FEATURE_CONTROL 寄存器

IA32_FEATURE_CONTROL 寄存器也影响着 VMXON 指令的执行，这个寄存器的 bit 0 为 lock 位，bit 1 与 bit 2 分别是 SMX 模式的 **inside** 与 **outside** 位。Inside 位指示允许在 SMX 内使用 VMX，outside 位指示允许在 SMX 外使用 VMX（这是开启 VMX 模式最基本的条件）。

（1）当 lock 位为 0 时，执行 VMXON 指令将产生#GP 异常。因此，执行 VMXON 指令前必须确保 lock 位为 1 值，也就是锁上 IA32_FEATURE_CONTROL 寄存器。上锁后如果对 IA32_FEATURE_CONTROL 寄存器进行写操作，将产生#GP 异常。

（2）Enable VMX inside SMX 位（bit 1），指示当处理器处于 SMX（Safer Mode Extensions）模式时，允许开启 VMX operation 模式。当 **inside=0 且处于 SMX 模式**时，执行 VMXON 指令将引发#GP 异常。

（3）Enable VMX outside SMX 位（bit 2），指示允许在 SMX 模式之外开启 VMX operation 模式。当 **outside=0 且不在 SMX 模式**时，执行 VMXON 指令将引发#GP 异常。

只有支持 VMX 与 SMX 模式时，才允许对 bit 1 置位（即 CPUID.01H:ECX[6:5]=11B）。在支持 VMX 模式时，才允许对 bit 2 置位（即

CPUID.01H:ECX[5]=1）。

注意：“Enable VMX outside SMX”位需要置位，这是运行 VMX 模式的基本条件。bit 1 与 bit 2 允许同时置位，此时表示无论是 SMX 模式内还是模式外都可以开启 VMX 模式。

下面的代码片段来自\lib\system_data_manage.asm 里的 get_vmx_global_data 函数，用来收集和分析 VMX 的数据：

代码片段 2-4：

```
        ;;
        ;; 关于 IA32_FEATURE_CONTROL.lock 位：
        ;; 1) 当 lock = 0 时，执行 VMXON 产生 #GP 异常
        ;; 2) 当 lock = 1 时，写 IA32_FEATURE_CONTROL 寄存器产生 #GP 异常
        ;;

        ;;
        ;; 下面将检查 IA32_FEATURE_CONTROL 寄存器
        ;; 1) 当 lock 位为 0 时，需要进行一些设置，然后锁上 IA32_FEATURE_CONTROL
        ;;
        mov ecx, IA32_FEATURE_CONTROL
        rdmsr
        bts eax, 0                                              ; 检查 lock 位，并上锁
        jc get_vmx_global_data.@7

        ;; lock 未上锁时：
        ;; 1) 对 lock 置位（锁上 IA32_FEATURE_CONTROL 寄存器）
        ;; 2) 对 bit 2 置位（启用 enable VMXON outside SMX）
        ;; 3) 如果支持 enable VMXON inside SMX，对 bit 1 置位
        ;;
        mov esi, 6          ; enable VMX outside SMX = 1, enable VMX inside SMX = 1
        mov edi, 4          ; enable VMX outside SMX = 1, enable VMX inside SMX = 0

        ;;
        ;; 检查是否支持 SMX 模式
        ;;
        test DWORD [gs: PCB.FeatureEcx], CPU_FLAGS_SMX
        cmovz esi, edi
        or eax, esi
        wrmsr


get_vmx_global_data.@7:

        ;;
        ;; 假如使用 enable VMX inside SMX 功能，
        ;; 则根据 IA32_FEATURE_CONTROL[1] 来决定是否必须开启 CR4.SMXE
        ;; 1) 本书例子中没有开启 CR4.SMXE
        ;;
%ifdef ENABLE_VMX_INSIDE_SMX
        ;;
        ;; ### step 7: 设置 Cr4FixedMask 的 CR4.SMXE 位 ###
        ;;
        ;; 再次读取 IA32_FEATURE_CONTROL 寄存器
        ;; 1) 检查 enable VMX inside SMX 位（bit1）
        ;;   1.1) 如果是 inside SMX（即 bit1 = 1），则设置 CR4FixedMask 标志位的相应位
        ;;
```

```
        rdmsr
        and eax, 2                  ; 取 enable VMX inside SMX 位的值（bit1）
        shl eax, 13                 ; 对应 CR4 寄存器的 bit 14 位（即 CR4.SMXE 位）
                                    ; 在 Cr4FixedMask 里设置 enable VMX inside SMX 位的值
        or DWORD [ebp + PCB.Cr4FixedMask], eax

%endif
```

这段代码检查 IA32_FEATURE_CONTROL 寄存器的 lock 位是否为 1。不为 1 时，需要上锁，并对“Enable VMX outside SMX”置位，同时根据 CPUID.01H:EDX[6].SMX 位来设置“Enable VMX inside SMX”位。

在定义了 ENABLE_VMX_INSIDE_SMX 符号时，接下来分析是否启用“Enable VMX inside SMX”，如果是，那么 Cr4FixedMask 的值需要添加 CR4.SMXE 位（需要为 1 值），也就是必须要开启 SMX 模式。本书中没有开启 CR4.SMXE 位。

2.3.2.2　CR0 与 CR4 固定位

进入 VMX operation 模式前，CR0 与 CR4 寄存器也必须要满足 VMX operation 模式运行的条件：

（1）CR0 寄存器需要满足 **PG**（bit 31）、**NE**（bit 5），以及 **PE**（bit 0）位为 1。也就是需要开启分页机制的保护模式，x87 FPU 数字异常的处理模式需要使用 native 模式。

（2）CR4 寄存器需要开启 VMXE 位（bit 13）。

这些位在 IA32_VMX_CR0_FIXED0 寄存器里属于 Fixed to 1（固定为 1）位（参见第 2.5.10 节），否则执行 VMXON 指令将会产生#GP 异常。由于 CR0.PG 与 CR0.PE 位必须为 1，因此 VMX operation 不能从实模式和非分页的保护模式里进入。如果需要在 SMX 模式里进入 VMX 模式，那么 CR0.SMXE 位（bit 14）也需要为 1 值。

在 VMCS 区域的字段设置里，也有很多类似这样的制约条件存在，某些位必须为 1 值，某些位必须为 0 值，某些位可以为 0 值也可以为 1 值。例如，上面的 CR0 与 CR4 寄存器列举的位必须为 1 值。因此，软件需要检测制约条件而进行相应的设置，这些设置也可能随着处理器架构的发展而有所改变。

上面所述的 CR0 与 CR4 寄存器固定位从第 1 代支持 VMX 的处理器开始就被确定了，后面我们将会看到使用 IA32_VMX_CR0_FIXED0 与 IA32_VMX_CR0_FIXED1 对 CR0 寄存器的固定位进行检测，使用 IA32_VMX_CR4_FIXED0 与 IA32_VMX_CR4_FIXED1 对 CR4 寄存器固定位进行检测。

下面一段代码来自 lib\Vmx\VmxInit.asm 里的 initialize_vmxon_region 函数，执行对 CR0 与 CR4 寄存器固定位的设置。

代码片段 2-5：

```
        ;;
        ;; 读 CR0 当前值
        ;;
        REX.Wrxb
        mov ecx, cr0
```

```
        mov ebx, ecx

        ;;
        ;; 检查 CR0.PE 与 CR0.PG 是否符合 fixed 位，这里只检查低 32 位值
        ;; 1) 对比 Cr0FixedMask 值（固定为 1 值），不相同则返回错误码
        ;;
        mov eax, STATUS_VMX_UNEXPECT                ; 错误码（超出期望值）
        xor ecx, [ebp + PCB.Cr0FixedMask]           ; 与 Cr0FixedMask 值异或，检查是否相同
        js initialize_vmxon_region.done             ; 检查 CR0.PG 位是否相等
        test ecx, 1
        jnz initialize_vmxon_region.done            ; 检查 CR0.PE 位是否相等

        ;;
        ;; 如果 CR0.PE 与 CR0.PG 位相符，设置 CR0 其他位
        ;;
        or ebx, [ebp + PCB.Cr0Fixed0]               ; 设置 Fixed 1 位
        and ebx, [ebp + PCB.Cr0Fixed1]              ; 设置 Fixed 0 位
        REX.Wrxb
        mov cr0, ebx                                ; 写回 CR0
        ;;
        ;; 直接设置 CR4 fixed 1 位
        ;;
        REX.W
        mov ecx, cr4
        or ecx, [ebp + PCB.Cr4FixedMask]            ; 设置 Fixed 1 位
        and ecx, [ebp + PCB.Cr4Fixed1]              ; 设置 Fixed 0 位
        REX.W
        mov cr4, ecx
```

代码设置流程如下：

（1）检查 CR0 寄存器的 PE 与 PG 位是否满足条件（在开启分页的保护模式下），通过与 Cr0FixedMask 值进行比较，如果 PG 和 PE 位与 Cr0FixedMask 中的 PG 和 PE 值不同，则失败返回。此错误不能在当前函数修复。

（2）如果 CR0 满足上面的基本条件，则直接设置 CR0 的 Fixed to 1（固定 1）位与 Fixed to 0（固定 0）位。这里的 Cr0Fixed0 与 Cr0Fixed1 的值来自于 IA32_VMX_CR0_FIXED0 与 IA32_VMX_CR0_FIXED1 寄存器。

（3）CR4 可以直接设置，通过“**或**”上 Cr4FixedMask 值，然后“**与**”上 Cr4Fixed1 值，得到最终的 CR4 的值。

CR0 与 CR4 寄存器设置的算法是一样的，通过“**或**”固定为 1 值，然后“**与**”固定为 0 值，得出最后需要的值。

根据 CR0 寄存器的限制，处理器在 VMX operation 模式（包括 **VMX root-operation** 与 **VMX non-root operation** 模式）必须要开启分页保护模式，那么 guest 软件必须运行在分页保护模式，在进入 guest 端时，处理器会检查是否符合这个要求。当不符合这个制约条件时，将产生 VM-entry 失败。如果需要 guest 软件执行实模式代码，那么可以选择进入 virtual-8086 模式。

后续的处理器可能会支持在 VMX non-root operation 模式下使用实模式，软件需要检测是否支持“**unrestricted guest**”（不受限制的 guest）功能。在支持并开启“unrestricted

guest”功能后，guest 软件允许运行在**实模式**或者**未分页的保护模式**。

在这种情况下，进入 guest 时，处理器将不会检查 CR0.PE 与 CR0.PG 控制位（参见第 4.5.1 节），而忽略 CR0 的 Fixed to 1（固定为 1）中的相应位。但是，如果 CR0.PG 为 1，则 CR0.PE 必须为 1。

2.3.2.3 A20M 模式

还有一个制约就是：当处理器处于 A20M 模式（即 A20 线 mask 模式），也不允许进入 VMX operation 模式，执行 VMXON 指令将引发#GP 异常。A20M 模式会屏蔽 A20 地址线产生所谓的“wrapping”现象，从而模拟 8086 处理器的 1M 地址内访问行为。

例子的 boot 阶段就调用了 FAST_A20_ENABLE 宏来开启 A20 地址线（即 A20 线 unmask），这个宏实现在 inc\cpu.inc 文件里。

2.3.3 设置 VMXON 区域

在虚拟化平台中，可能只有一份 VMM 存在，但可以有多份 VM 实例存在。每个 VM 需要有对应的 VMCS（虚拟机控制结构）区域来控制，而 VMM 本身也需要一个 VMXON 区域来进行一些记录或者维护工作。

```
vmxon [ebp + PCB.VmxonPhysicalPointer]
```

如上面代码所示，执行 VMXON 指令，需要提供一个 VMXON 指针作为操作数，这个指针是物理地址，指向 VMXON 区域。

2.3.3.1 分配 VMXON 区域

在执行 VMXON 指令进入 VMX operation 模式前，需要分配一块物理内存区域作为 VMXON 区域。这块物理内存区域需要对齐在 4K 字节边界上。VMXON 区域的大小和内存 cache 类型可以通过检查 IA32_VMX_BASIC 寄存器来获得。

代码片段 2-6：

```
;;
;; 分配 VMXON region
;;
call get_vmcs_access_pointer                ; edx:eax = pa:va
REX.Wrxb
mov [ebp + PCB.VmxonPointer], eax
REX.Wrxb
mov [ebp + PCB.VmxonPhysicalPointer], edx
```

如上代码所示，在初始化 VMXON 区域阶段 initialize_vmxon_region 函数里调用 get_vmcs_access_pointer 来分配 VMXON 区域，返回的物理地址保存在 PCB.VmxonPhysicalPointer 值里，虚拟地址保存在 PCB.VmxonPointer 值里。

2.3.3.2 VMXON 区域初始设置

VMXON 区域的首 8 个字节结构与 VMCS 区域是一样的，首 4 个字节为 VMCS ID

值，下一个 DWORD 位置是 VMX-abort indicator 字段，存放 VMX-abort 发生后的 ID 值，如图 2-3 所示。

Byte Offset	Contents
0	Bits 30:0: VMCS revision identifier Bit 31: shadow-VMCS indicator
4	VMX-abort indicator
8	VMCS data (implementation-specific format)

图 2-3

这个 **VMCS ID** 值可以从 IA32_VMX_BASIC[31:0]来获得，执行 VMXON 指令前必须将这个 VMCS ID 值写入 VMXON 区域首 DWORD 位置。如果 VMCS ID 值与处理器当前 VMX 版本下的 VMCS ID 值不符，则产生 VMfailInvalid 失败（参见第 2.6.2 节），此时 CF=1，指示 VMCS 指针无效。

代码片段 2-7：

```
;;
;; 设置 VMCS region 信息
;;
REX.Wrxb
mov ebx, [ebp + PCB.VmxonPointer]
mov eax, [ebp + PCB.VmxBasic]              ; 读取 VMCS revision identifier 值
mov [ebx], eax                             ; 写入 VMCS ID
```

上面代码来自 initialize_vmxon_region()函数，从 PCB.VmxBasic 里读取 32 位的 revision ID 值，然后写入 VMXON region 的首 4 个字节位置。PCB.VmxBasic 值在 stage1 阶段执行 get_vmx_global_data 函数时，从 IA32_VMX_BASIC 寄存器里读取。

注意：由于 VMXON 区域是物理内存区域，那么在设置 VMXON 区域之前，必须将 VMXON 区域映射到一个虚拟地址，然后通过虚拟地址进行写入操作。

代码中的 PCB.VmxonPointer 就是 VMXON 区域对应的虚拟地址指针，而 VmxonPhysicalPointer 则表示物理地址指针，这个物理指针使用在 VMXON 指令里。

2.3.4 退出 VMX operation 模式

处理器在 VMX operation 模式里不允许关闭 CR4.VMXE 位，只能在 VMX operation 模式外进行关闭，软件执行 VMXOFF 指令将退出 VMX operation 模式。

代码片段 2-8：

```
;;
;; 检查是否开启 VMX 模式
;;
test DWORD [ebp + PCB.ProcessorStatus], CPU_STATUS_VMXON
jz vmx_operation_exit.done
vmxoff
;;
;; 检查是否成功
```

```
        ;; 当 CF = 0 且 ZF = 0 时，VMXOFF 执行成功
        ;;
        mov eax, STATUS_VMXOFF_UNSUCCESS
        jc vmx_operation_exit.done
        jz vmx_operation_exit.done

        ;;
        ;; 下面关闭 CR4.VMXE 标志位
        ;;
        REX.Wrxb
        mov eax, cr4
        btr eax, 13
        REX.Wrxb
        mov cr4, eax

        ;;
        ;; 更新指令状态
        ;;
        and DWORD [ebp + PCB.InstructionStatus], ~INST_STATUS_VMX
        ;;
        ;; 更新处理器状态
        ;;
        and DWORD [ebp + PCB.ProcessorStatus], ~CPU_STATUS_VMXON
```

上面的代码片段来自 lib\VMX\VmxInit.asm 里的 vmx_operation_exit 函数，执行退出 VMX operation 工作：

（1）通过处理器状态标志检查当前是否处于 VMX operation 模式。

（2）执行 VMXOFF 指令退出 VMX operation 模式。

（3）关闭 CR4.VMXE 位。

（4）更新处理器状态标志位与指令状态标志位。

在执行 VMXOFF 指令后，必须检查指令是否成功。当 CF=1 时，指示当前的 VMCS 指针无效，当 ZF=1 时，指示 VMXOFF 指令执行遇到错误。只有当 CF 与 ZF 标志同时为 0 时，才表示 VMXOFF 是成功的！

在 VMX 开启 SMM dual-monitor treatment（SMM 双重监控处理）机制的情况下，必须先关闭 SMM 双重监控处理机制，才能使用 VMXOFF 指令关闭 VMX 模式。否则 VMXOFF 指令将会失败。

2.4　VMX operation 模式切换

进入 VMX operation 模式后，VMM 运行在 root 环境里，而 VM 需要运行在 non-root 环境里。虚拟化平台中经常会发生从 VMM 进入 VM，或者从 VM 返回到 VMM 的情况。有两个术语来描述它们之间的切换。

- **VM entry**（VM 进入）：从 VMX root operation 切换到 VMX non-root operation 就是 VM entry，表示从 VMM 进入到 VM 执行 guest 软件。
- **VM exit**（VM 退出）：从 VMX non-root operation 切换到 VMX root operation 就是

VM exit，表示从 VM 返回到 VMM。VMM 接管工作，VM 失去处理器控制权。

VM 的 entry 与 exit，顾名思义，是以 VM 的角度来定义。如果处理器要执行 guest 软件，那么 VMM 需要发起 VM-entry 操作，切换到 VMX non-root operation 模式执行。VM 会获得控制权，直到某些引发 VM exit 的事件发生。

VM-exit 发生后，处理器控制权重新回到 VMM。VMM 设置“VM exit”退出的条件是基于虚拟化处理器目的。因此，VMM 需要检查 VM 遇到了什么事件退出，从而虚拟化某些资源，返回一个虚拟化后的结果给 guest 软件，然后再次发起 VM-entry，切入 VM 让 guest 软件继续执行。

首次进入 VM 和退出后再次进入 VM 恢复执行，使用不同的指令进行。VMLAUNCH 指令发起首次进入 VM，VMRESUME 指令恢复被中断的 VM 执行。

图 2-4 摘自《Intel64 and IA-32 Architectures Software Developer's Manual》手册 Vol.3C23-2 页的图，很清晰地描述了 VMM 与 VM（guest）之间的切换关系。利用 VM entry 与 VM exit 行为，VMM 在多个 VM 之间来回切换。这个过程类似于 OS 对进程的调度，而每个 VM 就像一个进程，VMM 其中的一个职能类似于 OS 进程调度器。

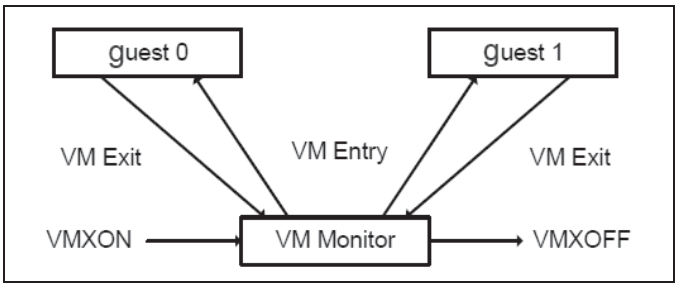

图 2-4

2.4.1 VM entry

首次进入 VM 环境，VMM 使用 VMLAUNCH 指令发起，VMLAUNCH 指令隐式地使用当前 VMCS 指针作为操作数。在发起 VM entry 操作前，VMM 需要对 VM 环境进行一些必要的配置工作，当前的 VMCS 指针（参见第 3.1.2 节）也必须先装载，下面是进入 VM 的简单流程。

（1）分配一个物理内存区域作为 VMCS 区域，这个区域需要在 4K 字节边界上对齐，并且需要满足内存的 cache 类型。区域的大小以及支持的 cache 类型，需要在 IA32_VMX_BASIC 寄存器里查询获得。

（2）写入 VMCS ID 值到这个 VMCS 区域的首 4 个字节里，这个 VMCS ID 值同样需要在 IA32_VMX_BASIC 寄存器里得到。

（3）提供这个 VMCS 的物理指针作为操作数，执行 VMCLEAR 指令，这个操作将设置 VMCS 的 launch 状态为“**clear**”，处理器置当前 VMCS 指针为

FFFFFFFF_FFFFFFFFh 值。VMCS 的 launch 状态和当前指针值由处理器动态维护，《Intel 手册》没明确说明它们存放在哪里，但推断应该是存放在 VMXON 区域内（进入 VMX operation 模式时被 VMXON 指令使用）。

（4）提供这个 VMCS 物理指针作为操作数，执行 VMPTRLD 指令，将装载这个 VMCS 指针作为当前的 VMCS 指针，处理器会维护这个 VMCS 指针。

（5）对 VMCS 区域的初始化设置，执行一系列的 VMWRITE 指令，将必要的数据写入当前 VMCS 指针指向的区域内（current-VMCS 区域）。VMWRITE 指令需要提供 VMCS 区域的一个 Index 值，这个 Index 值将用来指向 VMCS 区域的数据位置（参见第 3.3 节）。

（6）在第 5 步初始化 VMCS 区域完成后，执行 VMLAUNCH 指令进入 VM 环境，成功后处理器将切换到 VMX non-root operation 模式运行。guest 软件执行的进入点由 VMCS 初始化时写入 guest state area（guest 状态区域）的 RIP 字段里（参见第 3.8 节）。

需要注意的是，每次执行 VMX 指令都需要检查 CF 与 ZF 标志，确定指令是否成功，除非你能确保指令是成功的。例如，执行一系列的 VMREAD 与 VMWRITE 指令来读写 VMCS 区域，需要确保当前 VMCS 指针是正确有效的，并且提供的 Index 值在 VMCS 区域是存在的，这样大可不必在每次读写执行后进行烦琐的检查。

当 VM exit 产生后，VMM 需要重新进入 VM 环境，那么需要使用 VMRESUME 指令来恢复 VM 的运行。VMRESUME 指令也是隐式使用当前 VMCS 指针作为操作数，但是 VMRESUME 指令执行的假定前提是由于 VM 退出后再次进入 VM。因此，VMCS 的 launch 状态必须为“**launched**”，区别于首次进入 VM 时的 launch 状态为“**clear**”（执行 VMCLEAR 指令后的结果）。

2.4.2 VM exit

在 VM 中，guest 软件无法知道自己是否处于 VM 之中（Intel 保证 guest 软件没有任何途径可以检测）。因此，guest 软件不可能主动放弃控制权进行“VM exit”操作。

只有 guest 软件遇到一些**无条件 VM exit 事件**或者 **VMM 的设置引发 VM exit 的条件**发生，VM 才在不知不觉中失去了控制，VMM 将接管工作。guest 软件也无法知道自己什么时候发生了 VM exit 行为。

导致 VM exit 发生的三大类途径如下。

（1）执行无条件引发 VM exit 的指令，包括 CPUID、GETSEC、INVD 与 XSETBV 指令，以及所有 VMX 指令（除了 VMFUNC 指令外）。

（2）遇到无条件引发 VM exit 的**未被阻塞**的事件。例如，INIT 信号、SIPI 消息等。

（3）遇到 VMM 设置引发 VM exit 的条件，包括执行某些指令或者遇到某些事件发生。譬如，VMM 设置了“HLT exiting”条件，而 guest 软件执行了 HLT 指令而引发 VM

exit。又如，VM 遇到了 external-interrupt 的请求，VMM 设置了“external-interrupt exiting”条件而导致 VM exit。

一些事件的发生能无条件地导致 VM exit 发生。例如，triple fault（三重 fault 异常）事件和接收到 INIT 信号与 SIPI 信号，以及在使用 EPT（扩展页表）机制的情况下，遇到了 EPT violation（EPT 违例）或 EPT misconfiguration（EPT 配置不当）发生也能引发 VM exit。

后面将讲到，INIT 信号与 SIPI 信号在 virtual processor（虚拟处理器）的一些状态下能被阻塞，这里所说的“**虚拟处理器**”是指进入 VM 环境后的处理器，因为它的状态及资源可以被 VMM 设置，而呈现出一个**虚拟**的概念，并非指虚拟的处理器。虚拟处理器在 VM entry 完成后，它的活动状态被加载为 VMCS 区域的“guest state area”内的 Activity state 字段设置的状态值（参见第 4.17 节）。

虚拟处理器在 wait-for-SIPI 状态下 INIT 信号将被阻塞，不会产生 VM-exit。而 SIPI 信号能导致 VM exit 也仅仅当虚拟处理器处于 wait-for-SIPI 状态之下时才有效。INIT 与 SIPI 信号在 VMX root-operation 模式下都被阻塞（也就是在 VMM 下），它们被忽略。

在其他状态下（包括 active、HLT 以及 shutdown 状态），虚拟处理器接收到 INIT 信号将无条件产生 VM-exit。当发生 triple fault 事件退出时，VMM 可以选择将虚拟处理器置为 shutdown 状态。

在 OS 里的进程执行中，当时间片用完，需要被切换出控制权。而在 VT-x 技术里，除了因 guest 遇到可引发 VM exit 产生的事件而导致 VM exit 外，类似地，VMM 也可以为每个 VM 设置一个“时间片”，时间片用完导致 VM exit 发生。

新近的 VMX 架构也引入了 **VMX-preemption timer** 机制，VMM 设置了启用“activate VMX-preemption timer”功能，提供一个“VMX-preemption timer value”作为计数值，在 VM entry 切换进行时，计数值就开始递减 1，当这个计数值减为 0 时引发 VM exit。

这个 preemption timer 计数值递减的步伐依赖于 TSC 与 IA32_VMX_MISC[4:0]值，我们将在后面探讨。

2.4.3 SMM 双重监控处理下

在 SMM dual-monitor treatment 机制下，VMX 定义了另外两类的 VM exit 与 VM entry，它们是“**SMM VM-exit**”与“**VM-entry that return from SMM**”（从 SMM 返回中进入）。

- **SMM VM-exit**，可以从 VMM（VMX root-operation）或者 VM（VMX non-root operation）中产生 VM 退出行为，然后进入 SMM 模式执行被称为“**SMM-transfer Monitor**”（切入 SMM 监控者）的代码。
- **VM-entry that return from SMM**，将从 SMM 模式退出，然后返回到原来的

VMM 或 VM 中继续执行。

这个 SMM 双重监控处理是使用 VMM 的两端代码：VMX 端以及 SMM 端。也就是说，SMM 模式下也有 VMM 代码在运行。当发生 SMI（System manage interrupt）请求时，在 SMM 双重监控处理机制下，VMM 将从 **VMX 模式切入 SMM 模式**，然后执行 SMM 模式里的代码。

VMM 在 VMX 端的代码被称为“**Executive monitor**”，在 SMM 端的代码被称为“**SMM-transfer monitor**”。执行在 VMX 端时使用的区域被叫作“**executive VMCS**”，而 SMM 端使用的是“SMM-transfer VMCS”。

2.5 VMX 能力的检测

VMX 架构中的许多功能可能在不同的处理器架构里有不同的支持，新加入的 VMX 特性也可能在新近的处理器才支持，允许使用较早前的处理器不支持的一些非常实用的功能，比如前面所说的 VMX-preemption timer 功能。因此，VMM 需要检测当前 VMX 架构下有什么能力，进行相应的设置。

*提醒：本书在书写时主要参照的是《Intel 开发者手册》2012 年 8 月版本，order number 为：**325462-044US**。VMX 中的一些特性在笔者的电脑上并不支持，敬请读者注意。*

2.5.1 检测是否支持 VMX

软件需要检测处理器是否支持 VMX 架构，使用 CPUID.01H:ECX.VMX[5]位来进行检测（详见第 2.2.3 节）。

2.5.2 通过 MSR 组检查 VMX 能力

VMX 架构提供了众多项目能力的检测，包括：VMX 的基本信息、杂项信息、VPID 与 EPT 能力，还有对 VMCS 内的 control fields（控制字段）允许设置的位。而控制字段的位允许被置为 1 时，代表着处理器拥有这个能力。

譬如，secondary processor-based control 字段的 bit 7 是“Unrestricted guest”位，当它允许被置 1 时，表明处理器支持 unrestricted guest（不受限制的 guest 端）功能。反之，则表示处理器不支持该功能。

VMX 的这些能力的检测提供在几组共 14 个 MSR（Model Specific Register）里，如表 2-1 所示，除了这些，还有 4 个扩展了的 TRUE 系列寄存器，详见第 2.5.4 节和第 2.5.5 节。

表 2-1

序号	寄 存 器	描　　述
1	IA32_VMX_BASIC	提供 VMX 基本信息
2	IA32_VMX_PINBASED_CTLS	决定 Pin-based 控制字段
3	IA32_VMX_PROCBASED_CTLS	决定 Processor-based 控制字段
4	IA32_VMX_EXIT_CTLS	决定 VM-exit 控制字段
5	IA32_VMX_ENTRY_CLTS	决定 VM-entry 控制字段
6	IA32_VMX_MISC	提供 VMX 杂项信息
7	IA32_VMX_CR0_FIXED0	决定 CR0 的固定位
8	IA32_VMX_CR0_FIXED1	
9	IA32_VMX_CR4_FIXED0	决定 CR4 的固定位
10	IA32_VMX_CR4_FIXED1	
11	IA32_VMX_VMCS_ENUM	决定 VMCS Index 值
12	IA32_VMX_PROCEBASED_CTLS2	决定 secondary Pin-based 控制字段
13	IA32_VMX_EPT_VPID_CAP	决定 EPT 及 VPID 能力
14	IA32_VMX_VMFUNC	决定 VM-function 控制字段

软件通过读取这些寄存器的值来确定，在本书例子中，对这些寄存器的读取在 stage1 阶段执行的 get_vmx_global_data 函数里完成。

代码片段 2-9：

```
;-------------------------------------------------------
; get_vmx_global_data()
; input:
;       none
; output:
;       none
; 描述:
;       1) 读取 VMX 相关信息
;       2) 在 stage1 阶段调用
;-------------------------------------------------------
get_vmx_global_data:
        push ecx
        push edx

        ;;
        ;; VmxGlobalData 区域
        ;;
        mov edi, [gs: PCB.PhysicalBase]
        add edi, PCB.VmxGlobalData

        ;;
        ;; ### step 1: 读取 VMX MSR 值 ###
        ;; 1) 当 CPUID.01H:ECX[5]=1 时，IA32_VMX_BASIC 到 IA32_VMX_VMCS_ENUM 寄存器有效
        ;; 2) 首先读取 IA32_VMX_BASIC 到 IA32_VMX_VMCS_ENUM 寄存器值
        ;;

        mov esi, IA32_VMX_BASIC
```

```
get_vmx_global_data.@1:
       mov ecx, esi
       rdmsr
       mov [edi], eax
       mov [edi + 4], edx
       inc esi
       add edi, 8
       cmp esi, IA32_VMX_VMCS_ENUM
       jbe get_vmx_global_data.@1

       ;;
       ;; ### step 2: 接着读取 IA32_VMX_PROCBASED_CTLS2 ###
       ;; 1) 当 CPUID.01H:ECX[5]=1，并且 IA32_VMX_PROCBASED_CTLS[63] = 1 时，
       ;; IA32_VMX_PROCBASED_CTLS2 寄存器有效
       ;;
       test DWORD [gs: PCB.ProcessorBasedCtls + 4], ACTIVATE_SECONDARY_CONTROL
       jz get_vmx_global_data.@5

       mov ecx, IA32_VMX_PROCBASED_CTLS2
       rdmsr
       mov [gs: PCB.ProcessorBasedCtls2], eax
       mov [gs: PCB.ProcessorBasedCtls2 + 4], edx

       ;;
       ;; ### step 3: 接着读取 IA32_VMX_EPT_VPID_CAP
       ;; 1) 当 CPUID.01H:ECX[5]=1，IA32_VMX_PROCBASED_CTLS[63]=1，并且
       ;; IA32_PROCBASED_CTLS2[33]=1 时，IA32_VMX_EPT_VPID_CAP 寄存器有效
       ;;
       test edx, ENABLE_EPT
       jz get_vmx_global_data.@5

       mov ecx, IA32_VMX_EPT_VPID_CAP
       rdmsr
       mov [gs: PCB.EptVpidCap], eax
       mov [gs: PCB.EptVpidCap + 4], edx

       ;;
       ;; ### step 4: 读取 IA32_VMX_VMFUNC  ###
       ;; 1) IA32_VMX_VMFUNC 寄存器仅在支持 "enable VM functions" 1-setting 时有效，因此需要检
测是否支持!
       ;; 2) 检查 IA32_VMX_PROCBASED_CTLS2[45] 是否为 1 值
       ;;
       test DWORD [gs: PCB.ProcessorBasedCtls2 + 4], ENABLE_VM_FUNCTION
       jz get_vmx_global_data.@5

       mov ecx, IA32_VMX_VMFUNC
       rdmsr
       mov [gs: PCB.VmFunction], eax
       mov [gs: PCB.VmFunction + 4], edx


get_vmx_global_data.@5:

       ;;
       ;; ### step 5: 读取 4 个 VMX TRUE capability 寄存器 ###
       ;;
       ;; 如果 bit55 of IA32_VMX_BASIC 为 1，支持 4 个 capability 寄存器：
       ;; 1) IA32_VMX_TRUE_PINBASED_CTLS  = 48Dh
       ;; 2) IA32_VMX_TRUE_PROCBASED_CTLS = 48Eh
```

```
        ;; 3) IA32_VMX_TRUE_EXIT_CTLS      = 48Fh
        ;; 4) IA32_VMX_TRUE_ENTRY_CTLS     = 490h
        ;;
        bt DWORD [gs: PCB.VmxBasic + 4], 23
        jnc get_vmx_global_data.@6

        mov BYTE [gs: PCB.TrueFlag], 1                              ; 设置 TrueFlag 标志位
        ;;
        ;; 如果支持 TRUE MSR，那么就更新下面的 MSR:
        ;; 1) IA32_VMX_PINBASED_CTLS
        ;; 2) IA32_VMX_PROCBASED_CTLS
        ;; 3) IA32_VMX_EXIT_CTLS
        ;; 4) IA32_VMX_ENTRY_CTLS
        ;; 用 TRUE MSR 的值替代上面的 MSR!
        ;;
        mov ecx, IA32_VMX_TRUE_PINBASED_CTLS
        rdmsr
        mov [gs: PCB.PinBasedCtls], eax
        mov [gs: PCB.PinBasedCtls + 4], edx
        mov ecx, IA32_VMX_TRUE_PROCBASED_CTLS
        rdmsr
        mov [gs: PCB.ProcessorBasedCtls], eax
        mov [gs: PCB.ProcessorBasedCtls + 4], edx
        mov ecx, IA32_VMX_TRUE_EXIT_CTLS
        rdmsr
        mov [gs: PCB.ExitCtls], eax
        mov [gs: PCB.ExitCtls + 4], edx
        mov ecx, IA32_VMX_TRUE_ENTRY_CTLS
        rdmsr
        mov [gs: PCB.EntryCtls], eax
        mov [gs: PCB.EntryCtls + 4], edx


get_vmx_global_data.@6:
        ;;
        ;; ### step 6: 设置 CR0 与 CR4 的 mask 值（固定为 1 值）
        ;; 1) Cr0FixedMask = Cr0Fixed0 & Cr0Fixed1
        ;; 2) Cr4FixedMask = Cr4Fixed0 & Cr4Fxied1
        ;;
        mov eax, [gs: PCB.Cr0Fixed0]
        mov edx, [gs: PCB.Cr0Fixed0 + 4]
        and eax, [gs: PCB.Cr0Fixed1]
        and edx, [gs: PCB.Cr0Fixed1 + 4]
        mov [gs: PCB.Cr0FixedMask], eax                             ; CR0 固定为 1 值
        mov [gs: PCB.Cr0FixedMask + 4], edx
        mov eax, [gs: PCB.Cr4Fixed0]
        mov edx, [gs: PCB.Cr4Fixed0 + 4]
        and eax, [gs: PCB.Cr4Fixed1]
        and edx, [gs: PCB.Cr4Fixed1 + 4]
        mov [gs: PCB.Cr4FixedMask], eax                             ; CR4 固定为 1 值
        mov [gs: PCB.Cr4FixedMask + 4], edx

        ;;
        ;; 关于 IA32_FEATURE_CONTROL.lock 位:
        ;; 1) 当 lock = 0 时，执行 VMXON 产生 #GP 异常
        ;; 2) 当 lock = 1 时，写 IA32_FEATURE_CONTROL 寄存器产生 #GP 异常
        ;;

        ;;
        ;; 下面将检查 IA32_FEATURE_CONTROL 寄存器
```

```
        ;; 1) 当 lock 位为 0 时，需要进行一些设置，然后锁上 IA32_FEATURE_CONTROL
        ;; 2) 检查 IA32_FEATURE_CONTROL 的 bit1 与 bit2 位，决定 Cr4FixedMask 值
        ;;
        mov ecx, IA32_FEATURE_CONTROL
        rdmsr
        bts eax, 0                                              ; 检查 lock 位，并上锁
        jc get_vmx_global_data.@7

        ;; lock 未上锁时：
        ;; 1) 对 lock 置位（锁上 IA32_FEATURE_CONTROL 寄存器）
        ;; 2) 对 bit 2 置位（启用 enable VMXON outside SMX）
        ;; 3) 对 bit 1 清位（关闭 enable VMXON inside SMX）
        ;; 因此，此时 VMXON 指令需要在 outside SMX 下执行!
        ;;
        bts eax, 2                                          ; 置 enable VMX outside SMX = 1
        btr eax, 1                                          ; 清 enable VMX inside SMX = 0
        wrmsr

get_vmx_global_data.@7:
        ;;
        ;; ### step 7: 设置 Cr4FixedMask 的 CR4.SMXE 位 ###
        ;;
        ;; 再次读取 IA32_FEATURE_CONTROL 寄存器
        ;; 1) 检查 enable VMX inside SMX 位（bit1）
        ;; 2) 判断 VMX operation 是执行在 inside SMX 还是 outsie SMX
        ;;    2.1) 如果是 inside SMX（即 bit1 = 1），则设置 CR4FixedMask 位的相应位
        ;;
        ;; 如果 enable VMX inside SMX 位的值为 1，则表明必须开启 CR4.SMXE 位（即开启 SMX 模
式）
        ;;
        rdmsr
        and eax, 2                      ; 取 enable VMX inside SMX 位的值（bit1）
        shl eax, 13                     ; 对应在 CR4 寄存器的 bit 14 位（即 CR4.SMXE 位）
        or DWORD [ebp + PCB.Cr4FixedMask], eax     ; 设置 enable VMX inside SMX 位的值


get_vmx_global_data.@8:
        ;;
        ;; ### step 8: 查询 Vmcs 以及 access page 的内存 cache 类型 ###
        ;; 1) VMCS 区域内存类型
        ;; 2) VMCS 内的各种 bitmap 区域，access page 内存类型
        ;;
        mov eax, [gs: PCB.VmxBasic + 4]
        shr eax, 50-32                              ; 读取 IA32_VMX_BASIC[53:50]
        and eax, 0Fh
        mov [gs: PCB.VmcsMemoryType], eax


get_vmx_global_data.@9:
        ;;
        ;; ### step 9: 检查 VMX 所支持的 EPT page memory attribute ###
        ;; 1) 如果支持 WB 类型则使用 WB，否则使用 UC
        ;; 2) 在 EPT 设置 memory type 时，直接或上 [gs: PCB.EptMemoryType]
        ;;
        mov esi, MEM_TYPE_WB                                    ; WB
        mov eax, MEM_TYPE_UC                                    ; UC
        bt DWORD [gs: PCB.EptVpidCap], 14
        cmovnc esi, eax
        mov [gs: PCB.EptMemoryType], esi
```

```
get_vmx_global_data.done:
        pop edx
        pop ecx
        ret
```

这个 get_vmx_global_data 函数实现在 lib\system_data_manage.asm 文件里，在 stage1 阶段期间执行的 update_processor_basic_info 函数内部将调用这个函数来收集 VMX 的相关信息。

共分为 9 个步骤读取 VMX 的相关信息，其中一个额外的工作是设置前面第 2.3.2.1 节里提及的 IA32_FEATURE_CONTROL 寄存器。

2.5.3 例子 2-1

示例 2-1：列举出其中一个处理器 VMX 提供的能力信息

在继续讲解 VMX 能力之前我们先做个例子，将收集的 VMX 相关信息报告出来，例子主体代码在 chap02\ex2-1\ex.asm 文件里，如下所示。

代码片段 2-10：

```
        call get_usable_processor_index                         ; 获取可用的处理器 index 值
        mov esi, eax                                            ; 目标处理器为获取的处理器
        mov edi, TargetCpuVmxCapabilities                       ; 目标代码
        mov eax, signal                                         ; signal
        call dispatch_to_processor_with_waitting                ; 调度到目标处理器执行

        ;;
        ;; 等待 CPU 重启
        ;;
        call wait_esc_for_reset


;-----------------------------------------------
; TargetCpuVmxCapabilities()
; input:
;       none
; output:
;       none
; 描述：
;       1) 调度执行的目标代码
;-----------------------------------------------
TargetCpuVmxCapabilities:
        call update_system_status                               ; 更新系统状态
        call println

        ;;
        ;; 打印 VMX capabilities 信息
        ;;
        call dump_vmx_capabilities
        ret

signal  dd 1
```

这段代码找到一个可用的处理器 Index 值后，get_usable_processor_index 函数一般从

1 开始查询处理器是否可用。因此，一般会返回 1 值，也就是 CPU1。然后调度 TargetCpuVmxCapabilities 函数给这个处理器执行，TargetCpuVmxCapabilities 函数的工作只是简单地打印 CPU1 的 VMX 能力信息。dump_vmx_capabilities 函数代码实现在 lib\vmx\VmxDump.asm 文件里。

调度函数 dispatch_to_processor_with_waitting 需要提供一个信号指针，调用者需要等待目标处理器完成 TargetCpuVmxCapabilities 函数后才返回。它需要使用这个信号值进行等待。

使用命令 build -D__X64 编译该例子生成相应的映像文件后，在 VMware 里加载 demo.img（软盘映像）运行，该例子运行在 stage3 阶段。我们按下**<F2>键**切换到 CPU1，结果如图 2-5 所示。

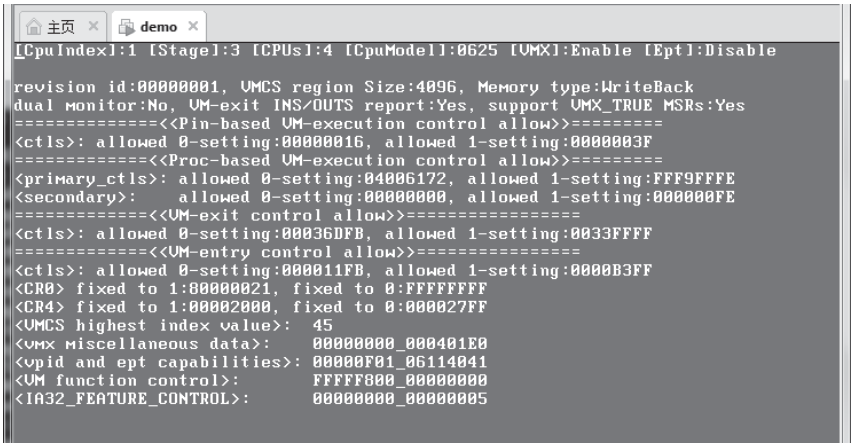

图 2-5

如图 2-5 所示，当前运行的 CpuIndex 值为 1（表示运行的是 CPU1），运行在 Stage3 阶段，已经进入了 VMX 模式，有 4 个逻辑处理器。

注意：这些 VMX 能力信息（capability information）是 VMware 提供的，并不代表真实机器具有的 VMX 能力。在下面的 VMX 能力信息里，VMCS ID 为 00000001h，VMCS 区域的大小为 4K，支持 VMX TRUE 寄存器，接下来是各个控制位的 allowed 0-setting 与 allowed 1-setting 位值，CR0 与 CR4 寄存器的固定位及杂项寄存器的值。在笔者的电脑上，VMCS 区域的大小为 1K，VMCS ID 值为 0FH。

VMCS index 的最大值为 45，IA32_FEATURE_CONTROL 寄存器的值为 05H，表示 lock 位上锁，使用了“VMX in outside SMX”模式。

2.5.4　基本信息检测

IA32_VMX_BASIC 寄存器用来检测 VMX 的基本能力信息，如图 2-6 所示。

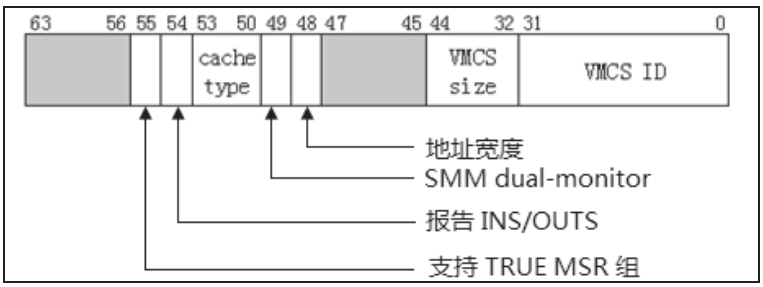

图 2-6

bits 31:0 为 VMCS ID 值，在初始化 VMXON 及 VMCS 区域时，需要用 VMCS ID 值来设置首 DWORD 位置。Bits 44:32 指示 VMCS 及 VMXON 区域的大小，这些区域大小以 1K 为单位，最高支持 4K 字节，并且地址需要在 4K 字节边界上。

bit 48 指示 VMXON 区域、VMCS 区域以及 VMCS 内所引用的区域（例如 I/O bitmap 区域、MSR bitmap 区域等）的宽度。为 1 时这些物理地址基址限定在 32 位内，为 0 时，物理地址宽度在 MAXPHYADDR 值内。在支持 64 位的平台上，bit 48 为 0 值。

bit 49 为 1 时，表明支持 SMM 及 SMI 的 dual-monitor treatment 功能。bit 54 为 1 时，表明支持当 VM-exit 是由于 INS 或 OUTS 指令而引发时，在 VMCS 的“VM-exit instruction information”字段里记录相应的信息。

bit 55 位为 1 时，表示支持 4 个 TRUE 寄存器，这 4 个 TRUE 寄存器将影响最终的 VMCS 中的某些控制位，如表 2-2 所示。

表 2-2

影响的 VMCS 字段	bit 55 = 0	bit 55 = 1
Pin-Based control	IA32_VMX_PINBASED_CTLS	IA32_VMX_TRUE_PINBASED_CTLS
Processor-Based control	IA32_VMX_PROCBASED_CTLS	IA32_VMX_TRUE_PROCBASED_CTLS
VM-exit contorl	IA32_VMX_EXIT_CTLS	IA32_VMX_TRUE_EXIT_CTLS
VM-entry control	IA32_VMX_ENTRY_CTLS	IA32_VMX_TRUE_ENTRY_CTLS

bit 55 的值决定由谁来控制这些 VMCS 字段固定位的设置。当 bit 55 为 0 时，这些 VMCS 字段的固定位值只需要分别受到 IA32_VMX_PINBASED_CTLS、IA32_VMX_PROCBASED_CTLS、IA32_VMX_EXIT_CTLS 及 IA32_VMX_ENTRY_CTLS 寄存器影响。为 1 时，则受对应的 TRUE 寄存器影响。

bit 53:50 这 4 位组成一个值，它指示 VMCS 区域以及 VMCS 区域内所引用的区域（例如 I/O bitmap，MSR bitmap 等）所支持的内存 cache 类型。

- 为 0 值时，支持 Uncacheable（UC）类型
- 为 6 值时，支持 WriteBack（WB）类型

VMX 目前只支持这两种 cache 类型，其他值（1~5 以及 7~15）都没使用（WC、WT、WP 及 UC-类型不支持）。在前面的 get_vmx_global_data 函数代码中读取了这个

cache 类型值，然后保存在 PCB.VmcsMemoryType 里。

IA32_VMX_BASIC 寄存器的其余位（bits 47:45 以及 bits 63:56）是保留位，为 0 值。

2.5.5　允许为 0 以及允许为 1 位

在表 2-2 里列举的两组寄存器（在 IA32_VMX_BASIC[55]=1 时使用 TRUE 寄存器），共 8 个寄存器，它们的结构和使用方法是一致的，结构如图 2-7 所示。

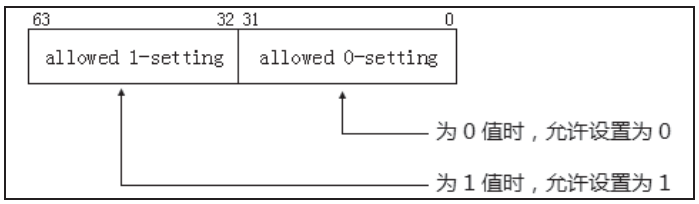

图 2-7

这些 64 位的寄存器低 32 位是 allowed 0-setting（允许设置为 0）位值，高 32 位是 allowed 1-setting（允许设置为 1）位值。这两组 32 位值对应一个 VMCS 区域的 control field（控制字段）值，这些控制字段是 32 位值，用法如下所示。

- bits 31:0 的用法：当其中的位为 0 值时，对应的控制字段相应的位允许为 0 值。
- bits 63:32 的用法：当其中的位为 1 值时，对应的控制字段相应的位允许为 1 值。

举个例子说明，在图 2-5 的运行结果里显示，IA32_VMX_TRUE_PINBASED_CTLS 的低 32 位值为 00000016h，高 32 位为 0000003Fh。它对应控制 Pin-based control 字段，那么表明：

（1）00000016h，Pin-based control 字段除了 bit 1、bit 2 以及 bit 4（值为 16h）不能为 0 值外，其他位都可以设置为 0 值。

（2）0000003Fh，Pin-based control 字段只允许 bit 0 到 bit 5 位可以设置 1 值，其余位必须为 0 值。

那么，形成 Pin-based control 字段的设置要求如图 2-8 所示。

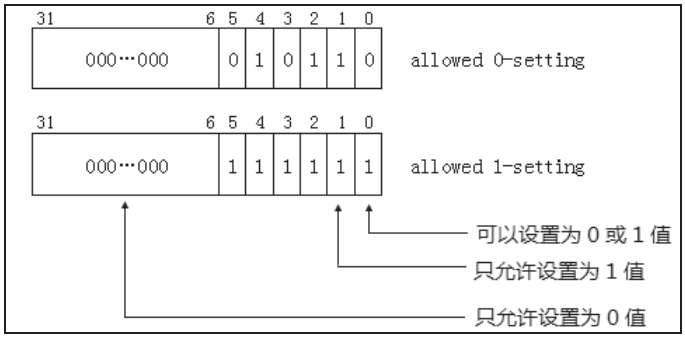

图 2-8

从上面的 allowed 0-setting 与 allowed 1-setting 的设置来看，bit 0 既可以为 0 值，也可以为 1 值。而 bit 1 只允许为 1 值。那么代表着 Pin-based control 字段的合法设置如下。

（1）bit 0、bit 3 以及 bit 5 可以设为 0 或 1 值。

（2）bit 1、bit 2 以及 bit 4 只能设为 1 值。

（3）bits 31:6 只能设为 0 值。

推广开来，表 2-2 中的 8 个寄存器，都使用相同的设置原理，不同的是设置的目标字段不一样。我们看到对于一个控制字段的设置：“**某些位必须为 1，某些位必须为 0**”（它们属于保留位）。

注意：这些必须为 0 值的保留位被称为“**default0**”位，必须为 1 值的保留位被称为“**default1**”位。这与通常接触到的数据结构中的“**保留位必须为 0 值**”有些区别（例如 PTE 中的保留位）。

当控制字段的保留位不符合这些 default0 和 default1 值，在 VM entry 操作时，字段的检查会失败，从而导致 VM entry 失败。

2.5.5.1 决定 VMX 支持的功能

前面所述，控制字段中必须为 0 及必须为 1 值的位都是保留位。当一个位可以设置为 1 值时，表明处理器将支持该项功能，也就是**非保留位**指示支持该位对应的功能。

以 Pin-based VM execution control 字段为例，它的 bit 6 是“activate VMX-preemption timer”功能，只有 bit 6 允许设置为 1 值时，才代表处理器支持 VMX-preemption timer 功能。在图 2-8 中，我们看到，bit 6 只能设置为 0 值。因此，这个处理器并不支持这项功能。

又如，该字段的 bit 0 为“external-interrupt exiting”功能位，而该位可以设置为 0 或 1 值，因而表明处理器支持该项功能。

2.5.5.2 控制字段设置算法

根据前面所述的 VMX 能力寄存器 allowed 0-setting 与 allowed 1-setting 位，对 VMCS 区域中相应的控制字段设置，可以分两步进行。首先根据自身的设计需求设置相应位，这是一个初始值。然后使用 allowed 0-setting 与 allowed 1-setting 位合成最终值。合成最终值的算法如下。

（1）Result1 = 初始值“**OR**”allowed 0-setting 值

（2）Result2 = Result1“**AND**”allowed 1-setting 值

这个 Result2 就是最终值，这个值确保满足在 VM-entry 时处理器对该控制字段的检查。下面的代码片段是对 Pin-based VM-execution control 字段的设置。

代码片段 2-11：

```
    ;;
    ;; 设置 Pin-based 控制域:
    ;; 1) [0]  - external-interrupt exiting: Yes
```

```
        ;; 2) [3]  - NMI exiting: Yes
        ;; 3) [5]  - Virtual NMIs: No
        ;; 4) [6]  - Activate VMX preemption timer: Yes
        ;; 5) [7]  - process posted interrupts: No
        ;;
        mov eax, EXTERNAL_INTERRUPT_EXITING | NMI_EXITING |
ACTIVATE_VMX_PREEMPTION_TIMER
        or eax, [ebp + PCB.PinBasedCtls]                        ; OR  allowed 0-setting
        and eax, [ebp + PCB.PinBasedCtls + 4]                   ; AND allowed 1-setting

        ;;
        ;; 写入 Pin-based VM-execution control 值
        ;;
        mov [ExecutionControlBufBase + EXECUTION_CONTROL.PinControl], eax
```

代码中对 Pin-based VM-execution control 字段的初始值设为 **49H**，然后依据上面的算法合成最终值。这是必须进行的一步，否则在执行 VM-entry 操作时，将由于检查 Pin-based VM-execution control 字段 default 位不满足要求而失败。

2.5.6　VM-execution 控制字段

在 VMCS 区域的三个 VM-execution control 字段需要检查 VMX 能力寄存器来确认 default0 与 default1 位。如下所示：

- Pin-based VM-execution control 字段
- primary processor-based VM-execution control 字段
- secondary processor-based VM-execution control 字段

在表 2-2 中列举了这些控制字段所支持的能力由其对应的寄存器提供。当 IA32_VMX_BASIC[55]为 1 时使用 TRUE 系列寄存器。此时，pin-based VM-execution control 字段的值由 IA32_VMX_TRUE_PINBASED_CTLS 寄存器来决定设置，primary processor-based VM-execution control 字段的值由 IA32_VMX_TRUE_PROCBASED_CTLS 寄存器来决定设置，但 secondary processor-based VM-execution control 字段没有对应的 TRUE 寄存器，它只由 IA32_VMX_PROCBASED_CTLS2 寄存器来决定设置。

2.5.6.1　Pin-based VM-execution control 字段

当 IA32_VMX_BASIC 的 bit 55 为 0 时，IA32_VMX_PINBASED_CTLS 寄存器决定 Pin-based VM-execution control 字段的设置。

当 IA32_VMX_BASIC 的 bit 55 为 1 时，忽略 IA32_VMX_PINBASED_CTLS 寄存器，Pin-based VM-execution control 字段由 IA32_VMX_TRUE_PINBASED_CTLS 寄存器来决定设置。

这两个 MSR 的用法是一样的，分为低 32 位与高 32 位。具体用法如下所示。

（1）bits 31:0（allowed 0-setting）：某位为 0 时，Pin-based VM-execution control 字段相应的位允许设为 0 值。

（2）bits 63:32（allowed 1-setting）：某位为 1 时，Pin-based VM-execution control 字段相应的位允许设为 1 值。

Pin-based VM-execution control 字段属于 32 位值，因此 MSR 的高 32 位（allowed 1-setting）也是对应 Pin-based VM-execution control 字段的 bits 31:0。

2.5.6.2 primary processor-based VM-execution control 字段

当 IA32_VMX_BASIC 的 bit 55 为 0 时，IA32_VMX_PROCBASED_CTLS 寄存器决定 primary processor-based VM-execution control 字段的设置。

当 IA32_VMX_BASIC 的 bit 55 为 1 时，忽略 IA32_VMX_PROCBASED_CTLS 寄存器，primary processor-based VM-execution control 字段由 IA32_VMX_TRUE_PROCBASED_CTLS 寄存器来决定设置。

这两个 MSR 的用法是一样的，分为低 32 位与高 32 位。具体用法如下所示。

（1）bits 31:0（allowed 0-setting）：某位为 0 时，primary processor-based VM-execution control 字段相应的位允许设为 0 值。

（2）bits 63:32（allowed 1-setting）：某位为 1 时，primary processor-based VM-execution control 字段相应的位允许设为 1 值。

primary processor-based VM-execution control 字段也是属于 32 位值，MSR 的高 32 位（allowed 1-setting）也是对应 primary processor-based VM-execution control 字段的 bits 31:0。

2.5.6.3 secondary processor-based VM-execution control 字段

由于 secondary processor-based VM-execution control 字段没有对应的 TRUE 寄存器。因此，它只受到 IA32_VMX_PROCBASED_CTLS2 寄存器的影响。

（1）bits 31:0（allowed 0-setting）：某位为 0 时，secondary processor-based VM-execution control 字段相应的位允许设为 0 值。

（2）bits 63:32（allowed 1-setting）：某位为 1 时，secondary processor-based VM-execution control 字段相应的位允许设为 1 值。

secondary processor-based VM-execution control 字段也是属于 32 位值，高 32 位的 allowed 1-setting 也是对应 secondary processor-based VM-execution control 字段的 bits 31:0。

2.5.7 VM-exit control 字段

当 IA32_VMX_BASIC 的 bit 55 为 0 时，IA32_VMX_EXIT_CTLS 寄存器决定 VM-exit control 字段的设置。

当 IA32_VMX_BASIC 的 bit 55 为 1 时，忽略 IA32_VMX_EXIT_CTLS 寄存器，VM-exit control 字段由 IA32_VMX_TRUE_EXIT_CTLS 寄存器来决定设置。

这两个 MSR 的用法是一样的，分为低 32 位与高 32 位。具体用法如下所示。

（1）bits 31:0（allowed 0-setting）：某位为 0 时，VM-exit control 字段相应的位允许设为 0 值。

（2）bits 63:32（allowed 1-setting）：某位为 1 时，VM-exit control 字段相应的位允许设为 1 值。

VM-exit control 字段属于 32 位值，MSR 的高 32 位（allowed 1-setting）也是对应 VM-exit control 字段的 bits 31:0。

2.5.8　VM-entry control 字段

当 IA32_VMX_BASIC 的 bit 55 为 0 时，IA32_VMX_ENTRY_CTLS 寄存器决定 VM-entry control 字段的设置。

当 IA32_VMX_BASIC 的 bit 55 为 1 时，忽略 IA32_VMX_ENTRY_CTLS 寄存器。VM-entry control 字段由 IA32_VMX_TRUE_ENTRY_CTLS 寄存器来决定。

这两个 MSR 的用法是一样的，分为低 32 位与高 32 位。具体用法如下所示。

（1）bits 31:0（allowed 0-setting）：某位为 0 时，VM-entry control 字段相应的位允许设为 0 值。

（2）bits 63:32（allowed 1-setting）：某位为 1 时，VM-entry control 字段相应的位允许设为 1 值。

VM-entry control 字段属于 32 位值，MSR 的高 32 位（allowed 1-setting）也是对应 VM-entry control 字段的 bits 31:0。

2.5.9　VM-function control 字段

当 VMX 支持“enable VM functions”功能时，将提供 IA32_VMX_VMFUNC 寄存器来决定 VM-function control 字段哪些位可以置位。与其他控制字段不同，VM-function control 字段是 64 位值。

当下面的条件满足时，才支持 IA32_VMX_VMFUNC 寄存器：

（1）CPUID.01H:ECX[5]=1，表明支持 VMX 架构。

（2）IA32_VMX_PROCBASED_CTLS[63]=1，表明支持 IA32_VMX_PROCBASED_CTLS2 寄存器。

（3）IA32_VMX_PROCBASED_CTLS2[45]=1，表明支持“enable VM functions”功能。

IA32_VMX_VMFUNC 寄存器没有 allowed 0-setting 位，只有 allowed 1-setting 位，用法如下所示。

bits 63:0（allowed 1-setting）：某位为 1 时，VM-function control 字段相应的位允许设为 1 值。

VM-function control 字段没有对应的 TRUE 寄存器，由 IA32_VMX_VMFUNC 寄存器最终决定设置。

2.5.10 CR0 与 CR4 的固定位

CR0 与 CR4 寄存器分别对应各自的 FIXED0 和 FIXED1 能力寄存器，如表 2-3 所示。

表 2-3

作　　用	CR0 寄存器	CR4 寄存器
fixed to 1	IA32_VMX_CR0_FIXED0	IA32_VMX_CR4_FIXED0
fixed to 0	IA32_VMX_CR0_FIXED1	IA32_VMX_CR4_FIXED1

编号为 0 的 FIXED 寄存器指示固定为 1 值，编号为 1 的 FIXED 寄存器指示固定为 0 值。需要这两个寄存器（FIXED0 和 FIXED1）来控制，是由于在 64 位模式下，CR0 与 CR4 寄存器是 64 位的。因此，一个用来描述固定为 1 值，一个用来描述固定为 0 值。

- 当 FIXED0 寄存器的位为 1 值时，CR0 和 CR4 寄存器对应的位必须为 1 值。
- 当 FIXED1 寄存器的位为 0 值时，CR0 和 CR4 寄存器对应的位必须为 0 值。

VMX 架构还保证了另一种现象：**当 FIXED0 寄存器的位为 1 时，FIXED1 寄存器相应的位也必定返回 1 值。FIXED1 寄存器的位为 0 时，FIXED0 寄存器相应的位也必定为 0 值**。

举例来说：IA32_VMX_CR4_FIXED0 的 bit 13 为 1 值，那么 IA32_VMX_CR4_FIXED1 的 bit 13 也会保证为 1 值。IA32_VMX_CR4_FIXED1 的 bit 20 为 0 值，IA32_VMX_CR4_FIXED0 的 bit 20 也保证为 0 值。

另外注意：如果 FIXED0 寄存器的某位为 0，而 FIXED1 寄存器对应的该位为 1 时，表明这个位允许设为 0 或者 1 值。例如，FIXED0 的 bit 1 为 0，而 FIXED1 的 bit 1 为 1 时，则 CR0 或 CR4 寄存器的 bit 1 可以为 0，也可以为 1。实际上，它与前面的控制字段保留位为 default0 与 default1 值有相似之处。

以图 2-5 的运行结果为例，注意，结果里只输出了低 32 位，因此以低 32 位为例说明。CR0 寄存器对应的 IA32_VMX_CR0_FIXED0 的值为 80000021h，IA32_VMX_CR0_FIXED1 的值为 FFFFFFFFh。CR4 寄存器对应的 IA32_VMX_CR4_FIXED0 值为 00002000h，IA32_VMX_CR4_FIXED1 的值为 000027FFh。

那么说明以下几种现象：

（1）CR0 寄存器的 bit 0、bit 5 以及 bit 31 必须为 1 值（FIXED0 与 FIXED1 寄存器的这些位都为 1）。也就是 CR0.PE、CR0.NE 以及 CR0.PG 位必须为 1，表示必须要开启分页的保护模式，并且使用 native 的 x87 FPU 浮点异常处理方法。

（2）CR4 寄存器的 bit 13 必须为 1 值（FIXED0 与 FIXED1 寄存器的 bit 13 都为 1）。也就是 CR4.VMEX 位必须为 1，表示必须开启 VMX 模式的许可。

（3）CR0 寄存器的其余位可为 0 或 1 值。

（4）CR4 寄存器的 bits 12:11 和 bits 31:14 必须为 0 值。

CR0 与 CR4 寄存器的固定位是 VMX 架构的制约条件之一，详见第 2.3.2 节所述。

2.5.10.1　CR0 与 CR4 寄存器设置算法

CR0 与 CR4 寄存器的设置和前面所说的控制字段方法是一样的。

（1）Result1 = 初始值“**OR**”FIXED0 寄存器

（2）Result2 = Result1“**AND**”FIXED1 寄存器

这个 Result2 就是 CR0 与 CR4 寄存器的最终合成值，也有下面的算法：

（1）FIXED0“AND”FIXED1 可以找到哪些位必须为 1 值。

（2）FIXED0“OR”FIXED1 可以找到哪些位必须为 0 值。

代码片段 2-12：

```
    ;;
    ;; 检查 CR0.PE 与 CR0.PG 是否符合 fixed 位，这里只检查低 32 位值
    ;; 1）对比 Cr0FixedMask 值（固定为 1 值），不相同则返回错误码
    ;;
    mov eax, STATUS_VMX_UNEXPECT                ; 错误码（超出期望值）
    xor ecx, [ebp + PCB.Cr0FixedMask]           ; 与 Cr0FixedMask 值异或，检查是否相同
    js initialize_vmxon_region.done             ; 检查 CR0.PG 位是否相等
    test ecx, 1
    jnz initialize_vmxon_region.done            ; 检查 CR0.PE 位是否相等

    ;;
    ;; 如果 CR0.PE 与 CR0.PG 位相符，设置 CR0 其他位
    ;;
    or ebx, [ebp + PCB.Cr0Fixed0]               ; 设置 Fixed 1 位
    and ebx, [ebp + PCB.Cr0Fixed1]              ; 设置 Fixed 0 位
    REX.Wrxb
    mov cr0, ebx                                ; 写回 CR0

    ;;
    ;; 直接设置 CR4 fixed 1 位
    ;;
    REX.W
    mov ecx, cr4
    or ecx, [ebp + PCB.Cr4FixedMask]                ; 设置 Fixed 1 位
    and ecx, [ebp + PCB.Cr4Fixed1]                  ; 设置 Fixed 0 位
    REX.W
    mov cr4, ecx
```

上面的代码是在进入 VMX 模式前对 CR0 和 CR4 寄存器进行设置。首先，检查 CR0 寄存器的初始值是否满足 CR0.PE=1 和 CR0.PG=1，表示当前处于分页保护模式。然后“**或**”FIXED0 寄存器，以及“**与**”FIXED1 寄存器值。

CR4 寄存器“或”Cr4FixedMask 值，这个值记录的是哪些位必须为 1，使用 Cr4FixedMask 而不是 FIXED0 寄存器是因为：Cr4FixedMask 也记录着是否需要开启 CR4.SMXE 位。

2.5.11 VMX 杂项信息

IA32_VMX_MISC 寄存器提供一些 VMX 的杂项信息，如图 2-9 所示。

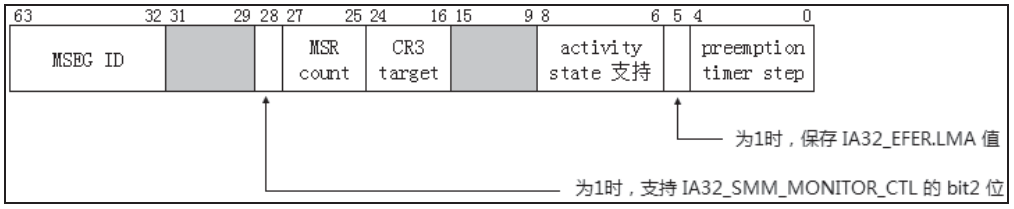

图 2-9

bits 4:0 提供一个 *X* 值，当 TSC 值的 bit *X* 改变时，VMX-preemption timer count 计数值将减 1。假如这个 *X* 值是 5，那么表示当 TSC 的 bit 5 发生改变（0 变 1 或 1 变 0）时，preemption timer count 计数值减 1。也就是说，TSC 计数 32 次时，preemption timer count 值减 1。

bit 5 为 1 时，表示当发生 VM-exit 行为时，将保存 IA32_EFER.LMA 的值在 VM-entry control 字段的“IA-32e mode guest”位里。只有当 VMX 支持“unrestricted guest”（不受限制的 guest）功能时，这个位才为 1 值。

bits 8:6 是一个 mask 位值，提供虚拟处理器 inactive 状态值的支持度，有下面的几个 inactive 状态值：

- bit 6 为 1 时，支持 HLT 状态
- bit 7 为 1 时，支持 shutdown 状态
- bit 8 为 1 时，支持 wait-for-SIPI 状态

只有这些 inactive 状态被支持时，才允许在 guest state 区域 activity state 字段设置相应的 inactive 状态值。bits 8:6 一般会返回 7（全部支持）。

bits 24:16 指示支持的 CR3-target 值的数量，一般会返回 4 值，表示支持 4 个 CR3 寄存器目标值。bits 27:25 返回一个 *N* 值，这个 *N* 值用来计算出在 MSR 列表（VM-exit MSR-load、VM-exit MSR-store 以及 VM-entry MSR-load 列表）里推荐的 MSR 最大个数。计算方法是：个数=（*N*+1）×512。一般会返回 0 值，表示列表里推荐最多支持 512 个 MSR。

bit 28 为 1 时，表示支持 IA32_SMM_MONITOR_CTL 寄存器的 bit 2 位能被设为 1 值。这个 bit 2 置位时，表示执行 VMXOFF 指令时，SMI 能被解开阻塞。一般情况下 VMXOFF 指令的执行将阻塞 SMI 请求。

bits 63:32 提供一个 MSEG ID 值，这个 ID 值用于 SMM 双重监控处理机制下。在初始化 SMM-transfer Monitor 所使用的 MSEG 区域头部时需要使用该 ID 值。

2.5.12　VMCS 区域字段 index 值

IA32_VMX_VMCS_ENUM 寄存器提供 VMCS 区域内字段的最高 index 值。在 VMCS 区域内含有若干字段，每个字段都有相应的 encode 值。

访问 VMCS 的字段时，需要将 encode 值作为 VMREAD 和 VMWRITE 指令的操作数，然后执行 VMREAD 和 VMWRITE 指令，对相应的字段进行读写访问。

如图 2-10 所示，IA32_VMX_VMCS_ENUM 寄存器的 bits 9:1 提供 encode 内最高的 index 值，以图 2-5 运行结果为例，这个最高 index 值为 45。

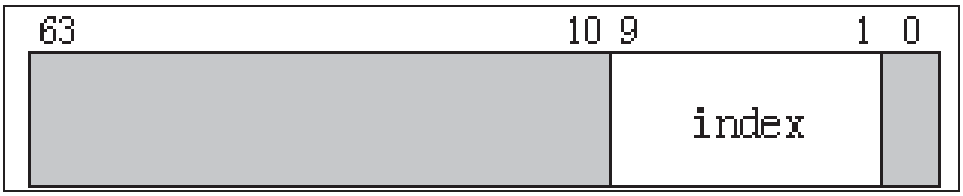

图 2-10

2.5.13　VPID 与 EPT 能力

IA32_VMX_EPT_VPID_CAP 寄存器提供两方面的能力检测，包括 EPT（扩展页表）所支持的能力，以及 EPT 页面 cache（TLBs 及 paging-structure cache）的能力。

当下面的条件满足时，才支持 IA32_VMX_EPT_VPID_CAP 寄存器。

（1）CPUID.01H:ECX[5]=1，表明支持 VMX 架构。

（2）IA32_VMX_PROCBASED_CTLS[63]=1，表明支持 IA32_VMX_PROCBASED_CTLS2 寄存器。

（3）IA32_VMX_PROCBASED_CTLS2[33]=1，表明支持“enable EPT”位。

如图 2-11 所示是 IA32_VMX_EPT_VPID_CAP 寄存器的结构。

对于 EPT 能力，bit 0 为 1 时，允许在 EPT 页表项里的 bits 2:0 使用 **100b**（execute-only 页）属性。bit 6 为 1 时，表明支持 4 级页表结构。bit 16 为 1 时支持使用 2M 页，bit 17 为 1 时支持使用 1G 页。最后，bit 21 为 1 时支持在页表项里使用 dirty 标志。

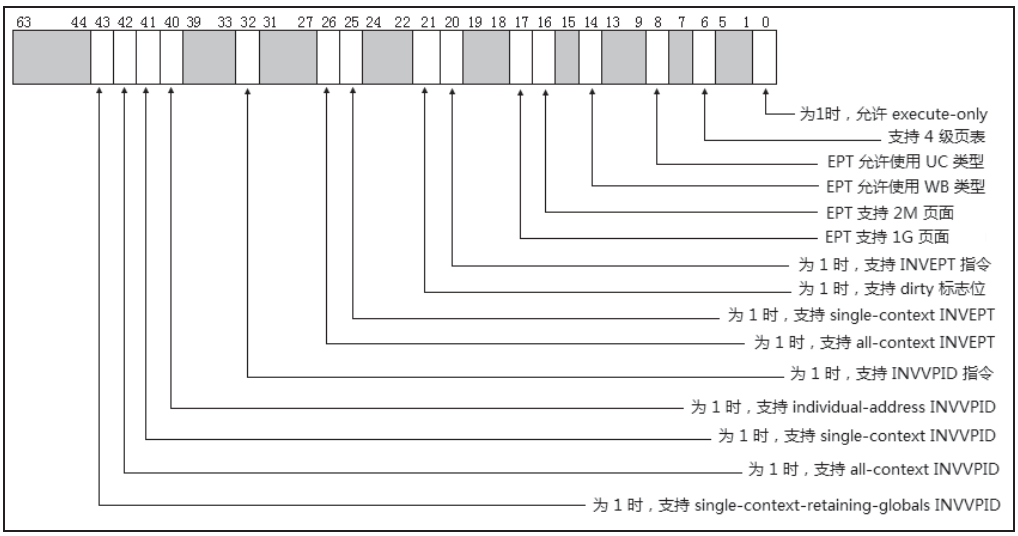

图 2-11

bit 8 为 1 时，允许在 EPTP 字段的 bits 2:0 里设为 UC 类型（值为 0），而 bit 14 为 1 时，允许在 EPTP 的 bits 2:0 里设置为 WB 类型（值为 6）。参见第 4.4.1.3 节。

对于 EPT cache 的能力，bit 20 为 1 时支持 INVEPT 指令，bit 32 为 1 时支持 INVVPID 指令。INVEPT 指令支持的刷新类型由 bit 25 和 bit 26 检测。bit 25 为 1 时，支持 single-context 刷新类型，bit 26 为 1 时，支持 all-context 刷新类型。

INVVPID 指令支持的刷新类型由 bits 43:40 检测。bit 40 为 1 时，支持 individual-address 类型（type 值为 0）。bit 41 为 1 时，支持 single-context 类型（type 值为 1）。bit 42 为 1 时，支持 all-context 类型（type 值为 2）。bit 43 为 1 时，支持 single-context-retaining-globals 类型（type 值为 3）。

2.6 VMX 指令

VMX 架构提供 13 条 VMX 指令，负责管理 4 个职能。如表 2-4 所示。

表 2-4

管理范围	指　令	描　述
VMCS 区域管理	VMPTRLD	加载一个 VMCS 指针作为 current-VMCS 指针
	VMPTRST	保存 current-VMCS 指针在内存里
	VMCLEAR	清 VMCS 状态，并置 current-VMCS 指针为 FFFFFFFF_FFFFFFFFh 值
	VMREAD	读 current-VMCS 字段值
	VMWRITE	写 current-VMCS 字段值

续表

管理范围	指　令	描　述
VMX 模式管理	VMXON	进入 VMX operation 模式
	VMXOFF	退出 VMX operation 模式
	VMLAUNCH	发起 VM-entry 操作，进入 VM
	VMRESUME	恢复 VM 执行
Cache 刷新	INVEPT	刷新 EPT 的 TLBs 和 paging-structure caches
	INVVPID	刷新 EPT 的 TLBs 和 paging-structure caches
调用服务例程	VMCALL	guest 调用 VMM 的服务例程，并退出 VM
	VMFUNC	guest 调用为 VM 准备的服务例程，不退出 VM

2.6.1　VMX 指令执行环境

VMX 架构对 CR0 和 CR4 寄存器的设置有基本的限制要求（详见第 2.3.2.2 节），即需要开启分页保护模式以及 CR4.VMEX=1。下面是 VMX 指令执行的基本环境。

（1）除了 VMXON 指令可以在进入 VMX operation 模式前执行，其他指令必须执行在 VMX operation 模式里。否则，将产生#UD 异常。

（2）不能在实模式、virtual-8086 模式以及 compatibility 模式下执行。否则，将产生#UD 异常。除了在支持并开启“unrestricted guest”功能后，在 guest 的非分页或非保护模式环境里可以执行 VMFUNC 指令外。

（3）所有 VMX 指令必须在 root 环境里执行（除了 VMFUNC 指令可以在 non-root 环境里执行外）。否则，将产生 VM-exit 行为。而 VMFUNC 指令执行在 root 环境里，将产生#UD 异常。

（4）除了 VMFUNC 指令外，所有 VMX 指令必须执行在 0 级权限里。否则，将产生#GP 异常。

VMXON 指令是唯一在 VMX operation 模式外可执行的指令。VMXON 指令在 root 内执行会产生下面所说的 VMfailValid 失败。在 non-root 内执行则会产生 VM-exit 行为。而 VMFUNC 指令是唯一 non-root 内正常执行的指令，如果在 root 内执行则会产生#UD 异常。

2.6.2　指令执行的状态

除了可能产生异常外（例如#GP、#UD 异常等），每条 VMX 指令的执行还可能会产生 3 个状态，分别表示为 VMsucceed、VMfailInvalid 以及 VMfailValid：

（1）**VMsuccess**：指令执行成功。指令会清所有的 eflags 寄存器标志位，例如 CF 与 ZF 标志。

（2）**VMfailInvalid**：表示因 current-VMCS 指针无效而失败。当 current-VMCS 指针

或 VMCS 内的 ID 值是无效时，VMX 指令会置 CF 标志位，指示执行失败。

（3）**VMfailValid**：表示遇到某些原因而失败。例如，VMREAD 指令读一个 VMCS 区域内不存在的字段时，VMREAD 指令会置 ZF 标志位，指示执行失败。

在这样的一个情形下，例如，当 current-VMCS 指针为初始值（FFFFFFFF_FFFFFFFFh），表明没有加载 VMCS 指针（使用 VMPTRLD 指令），执行 VMREAD 指令就会产生 VMfailInvalid 失败。

一般的情况下都需要检查 CF 与 ZF 标志位，以确定指令是否执行成功。VMfailInvalid 与 VMfailValid 失败的不同之处是：VMfailValid 失败会在当前 VMCS 内的 **VM-instruction error field**（指令错误字段）记录失败指令的编号。而 VMfailInvalid 是由于遇到无效的 VMCS 区域（典型地，VMCS ID 错误）或 current-VMCS 指针无效，因此不可能在 VMCS 内记录错误的信息。

2.6.3 VMfailValid 事件原因

VMX 指令产生的 VMfailValid 失败会对应一个编号，这个编号就记录在 VM-instruction error 字段里。这个字段是 VM-exit 信息区域内的其中一个 32 位字段。如表 2-5 所示。

表 2-5

错误编号	描　　述
1	VMCALL 指令执行在 root 环境
2	VMCLEAR 指令操作数是无效的物理地址
3	VMCLEAR 指令操作数是 VMXON 指针
4	VMLAUNCH 执行时，current-VMCS 状态是非“clear”
5	VMRESULT 执行时，current-VMCS 状态是非“launched”
6	VMRESULT 在 VMXOFF 指令后
7	VM-entry 操作时，current-VMCS 含有无效的 VM-execution 控制字段
8	VM-entry 操作时，current-VMCS 含有无效的 host-state 字段
9	VMPTRLD 操作数是无效的物理地址
10	VMPTRLD 操作数是 VMXON 指针
11	VMPTRLD 执行时，VMCS 内的 VMCS ID 值不符
12	VMREAD/VMWRITE 指令读/写 current-VMCS 内不存在的字段
13	VMWRITE 指令试图写 current-VMCS 内的只读字段
14	
15	VMXON 指令执行在 VMX root operation 里
16	VM-entry 操作时，current-VMCS 内含有无效的 executive-VMCS 指针
17	VM-entry 操作时，current-VMCS 内使用了非“launched”状态的 executive-VMCS
18	VM-entry 操作时，current-VMCS 内的 executive-VMCS 指针不是 VMXON 指针

续表

错误编号	描　　述
19	执行 VMCALL 指令切入 SMM-transfer monitor 时，VMCS 非“clear”状态
20	执行 VMCALL 指令切入 SMM-transfer monitor 时，VMCS 内的 VM-exit control 字段无效
21	
22	执行 VMCALL 指令切入 SMM-transfer monitor 时，MSEG 内的 MSEG ID 值无效
23	在 SMM 双重监控处理机制下执行 VMXOFF 指令
24	执行 VMCALL 指令切入 SMM-transfer monitor 时，MESG 内使用了无效的 SMM monitor 特征
25	进行“VM-entry that return from SMM”操作时，在 executive VMCS 中遇到了无效的 VM-execution control 字段
26	VM-entry 操作时，VMLAUNCH 或 VMRESUME 被 MOV-SS/POP-SS 阻塞
27	
28	INVEPT/INVVPID 指令使用了无效的操作数

表 2-5 列出了所有可能发生的 VMfailValid 原因，编号 14，21 以及 27 没有使用。其中有数个失败产生于 VM-entry 和 SMM 双重监控处理机制下的切入 SMM-transfer monitor 时。

2.6.4　指令异常优先级

如前面所述，如果 VMX 指令非正常执行，会出现下面三种情况之一。

（1）产生异常，可能产生的异常为：#UD，#GP，#PF 或者#SS。

（2）产生 VM-exit 行为。

（3）产生 VMfailInvalid 失败或者 VMfailValid 失败。

VMfailInvalid 与 VMfailValid 失败是在指令允许执行的前提下（即执行指令不会产生异常及 VM-exit），它会发生在 root 环境里。VMFUNC 指令执行在 non-root 环境不会产生失败，只可能产生#UD 异常或者 VM-exit。

在开启“unrestricted guest”功能后并进入实模式的 guest 软件里执行 VMX 指令时，一个#UD 异常的优先级高于 VM-exit 行为。或者一个由于 CPL 非 0 而引发的#GP 异常，优先级高于 VM-exit 行为（参见第 5.6 节）。

2.6.5　VMCS 管理指令

有五条 VMX 指令涉及 VMCS 区域的管理，如表 2-4 所示。下面将介绍它们的用法。

2.6.5.1 VMPTRLD 指令

VMPTRLD 指令从内存中加载一个 64 位的物理地址作为 **current-VMCS pointer**（当前 VMCS 指针），这个当前 VMCS 指针由处理器内部记录和维护，除了 VMXON、VMPTRLD 和 VMCLEAR 指令需要提供 VMXON 或 VMCS 指针作为操作数外，其他的指令都是在 current-VMCS 上操作。

```
vmptrld [gs: PCB.GuestA]            ; 加载 64 位 VMCS 地址
```

注意：这条指令读取的是 64 位物理地址值，即使在 32 位环境下。如上面代码所示，在 PCB.GuestA 里保存着一个 VMCS 区域的物理地址值，执行这条指令将更新 **current-VMCS pointer** 值。在指令未执行成功时，current-VMCS pointer 将维持原有值。

处理器会根据提供的 VMCS 指针，进行下面的检查事项：

（1）物理地址是否超过**物理地址宽度**，并且需要在 4K 地址边界上。当 IA32_VMX_BASIC[48]为 1 时，物理地址宽度在 32 位内，否则为 MAXPHYADDR 值（参见第 2.5.4 节）。

（2）VMCS 指针是否为 VMXON 指针。

（3）目标 VMCS 区域首 DWORD 值是否符合 VMCS ID 值。

当上面检查不通过时，如果原来已经存在 current-VMCS pointer，则会产生 **VMfailValid** 失败，current-VMCS pointer 将维持不变，并且在 current-VMCS 的指令错误字段里记录错误号是 11（如表 2-5 所示）。指示：“执行 VMPTRLD 时，VMCS 内的 VMCS ID 值不符”。否则，产生 VMfailInvalid 失败。

2.6.5.2 VMPTRST 指令

VMPTRST 将 64 位的 current-VMCS pointer 保存在提供的内存中，如下面示例。

```
vmptrst [gs: PCB.VmcsPhysicalPointer]          ; 保存 64 位 VMCS 地址
```

VMPTRST 指令不会产生 VMfailValid 或 VMfailInvalid 失败，除了产生异常或 VM-exit 外，它总是成功的。

2.6.5.3 VMCLEAR 指令

VMCLEAR 指令根据提供的 64 位 VMCS 指针，对目标 VMCS 区域进行一些初始化工作，并将目标 VMCS 的状态设置为“**clear**”。这个初始化工作，其中有一项是将处理器内部维护的关于 VMCS 的数据写入目标 VMCS 区域内。这部分 VMCS 数据可能与处理器相关，这是正确使用 VMCS 的前提。

在使用 VMLAUNCH 指令进入 VM 时，current-VMCS 的 launch 状态必须为“clear”。因此，当 VMCLEAR 指令对目标 VMCS 进行初始化后，使用 VMPTRLD 指令将目标 VMCS 加载为 current-VMCS，同时也更新 current-VMCS pointer 值。如下面代码示例。

```
vmclear [gs: PCB.GuestA]                ; 对 GuestA 进行初始化，置“clear”
jc @1
jz @1
```

```
vmptrld [gs: PCB.GuestA]                ; 将 GuestA 加载为 current-VMCS
jc @1
jz @1
```

执行上述操作后，在后续工作里对 GuestA 区域进行配置，然后就可以使用 VMLAUNCH 指令进入 VM 执行。

VMCLEAR 指令也会对目标 VMCS 指针进行与 VMPTRLD 指令相同的检查（参见第 2.6.5.1 节）。如果在执行 VMCLEAR 指令前已经加载过当前 VMCS 指针，并且 VMCLEAR 指令的目标 VMCS 是 current-VMCS，则会设置 current-VMCS pointer 为 FFFFFFFF_FFFFFFFFh 值。

2.6.5.4　VMREAD 指令

VMREAD 指令需要提供一个 32 位的 VMCS 字段 ID 值放在寄存器里作为源操作数。目标操作数是寄存器或者内存操作数，将读取到的相应字段值放在目标寄存器或者内存地址中。

VMREAD 指令在 32 位模式下固定使用 32 位的操作数，在 64 位模式下固定使用 64 位操作数。而 VMCS 字段有不同的宽度，例如，存在 16 位字段和 **natural-width** 类型的字段。natural-width 类型的字段在支持 64 位架构的处理器上是 64 位，在不支持 64 位架构的处理器上是 32 位。

为了适应不同宽度的字段，VMREAD 指令使用下面的读取原则：

- 如果 VMCS 字段的 size 小于目标操作数 size，则 VMCS 字段值读入目标操作数后，目标操作数的高位部分清 0。
- 如果 VMCS 字段的 size 大于目标操作数 size，则 VMCS 字段的低位部分读入目标操作数，VMCS 字段的高位部分忽略。

由于 VMCS 字段 ID 值为 32 位，在 64 位模式下源操作数的 bits 63:32 必须为 0。否则会因为尝试读取一个不支持的 VMCS 字段而产生 VMfailValid 失败。

```
vmread [ebp + EXIT_INFO.ExitReason], eax
vmread ebx, eax
```

上面的代码中，VMCS 字段 ID 值放在源操作数 EAX 里，读到的值放在内存及 EBX 寄存器里。VMCS 的字段 ID 也被称为“**字段 encode**”，由几个部分组成，如图 2-12 所示。

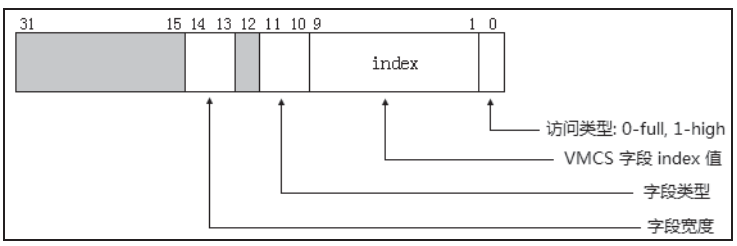

图 2-12

在前面第 2.5.12 节所提到的 IA32_VMX_VMCS_ENUM 寄存器（图 2-10）的 bits 9:1 提供最大的 VMCS 区域字段 index 值。这个最大 index 值就是指图 2-12 字段编码中的 index 最大值。关于字段编码的详细说明在第 3.3.1 节里描述。

在 inc\vmx.inc 文件里提供了两个宏，用来读取 VMCS 字段：**DoVmRead** 宏读取 VMCS 字段放在内存 buffer 中，**GetVmcsField** 宏读取 VMCS 字段值放在 EAX 寄存器中。如代码片段 2-13 所示。

代码片段 2-13：

```
;-----------------------------------------------
; DoVmRead
; input:
;       %1 - vmcs ID
;       %2 - target value (memory)
; output:
;       %2 - target value
;-----------------------------------------------
%macro DoVmRead 2
%if __BITS__ == 64
        mov rax, %1
        vmread %2, rax
%else
       mov eax, %1
       vmread %2, eax
%endif
%endmacro
;-----------------------------------------------
; GetVmcsField
; input:
;       %1 - VMCS ID
; output:
;       eax - field value
;-----------------------------------------------
%macro GetVmcsField    1
%if __BITS__ == 64
       mov rsi, %1
       vmread rax, rsi
%else
       mov esi, %1
       vmread eax, esi
%endif
%endmacro
```

DoVmRead 宏有两个参数，第 1 个参数是字段 ID 值，第 2 个是目标操作数。GetVmcsField 宏有 1 个参数，提供字段 ID 值，而输出固定使用 EAX 寄存器。

那么，就可以像下面的示例，使用 DoVmRead 或 GetVmcsField 宏来读取 VM-exit 信息区域里的 Exit-reason 字段。

```
        DoVmRead        EXIT_REASON, [ebp + EXIT_INFO.ExitReason]
        GetVmcsField    EXIT_REASON
```

VMREAD 指令操作在 current-VMCS 区域。因此，处理器会检查：（1）current-VMCS 指针是否无效（例如，为 FFFFFFFF_FFFFFFFFh 值）。（2）提供的字段 ID 值是

否有效，即 VMCS 中是否存在此字段。

在第 2.6.5.3 节里提到，当使用 VMCLEAR 指令对 current-VMCS 初始化时，当前 VMCS 指针就会变为 FFFFFFFF_FFFFFFFFh 值，此时执行 VMREAD 指令会产生 **VMfailInvalid** 失败。而当不存在 VMCS 字段时，则会产生 **VMfailValid** 失败。

2.6.5.5 VMWRITE 指令

VMWRITE 指令的源操作数是寄存器或者内存操作数，提供需要写入 VMCS 字段的值。目标操作数是寄存器，32 位的字段 ID 值提供在目标寄存器操作数里。

VMWRITE 指令在 32 位模式下固定使用 32 位的操作数，在 64 位模式下固定使用 64 位操作数。而 VMCS 字段有不同的宽度，例如存在 16 位字段和 natural-width 类型的字段。natural-width 类型的字段在支持 64 位架构的处理器上是 64 位，在不支持 64 位架构的处理器上是 32 位。

为了适合不同宽度的字段，WMWRITE 指令使用下面的写入原则：

- 如果写入值的 size 小于 VMCS 字段的 size，则值写入 VMCS 字段后，VMCS 字段的高位部分清 0。
- 如果写入值的 size 大于 VMCS 字段的 size，则忽略写入值的高位部分，低位部分写入 VMCS 字段里。

由于 VMCS 字段 ID 值为 32 位，在 64 位模式下目标操作数的 bits 63:32 必须为 0。否则会因为尝试写入一个不支持的 VMCS 字段而产生 VMfailValid 失败。

下面的代码中，写入数据放在 EBX 寄存器或者内存操作数，写入由 EAX 寄存器内的字段 ID 值指示的 VMCS 字段里。

```
        vmwrite eax, ebx
        vmwrite eax, [ebp + GUEST_STATUE.Rip]
```

与 VMREAD 指令稍微不同，VMWRITE 指令还会检查目标的 VMCS 字段**是否可写**。如果对一个 Read-Only 类型的字段进行写操作，将产生 VMfailValid 失败，错误编号为 13。

代码片段 2-14：

```
;-----------------------------------------------
; DoVmWrite
; input:
;       %1 - vmcs ID
;       %2 - value (memory)
; output:
;       none
;-----------------------------------------------
%macro DoVmWrite 2
%if __BITS__ == 64
        mov rax, %1
        vmwrite rax, %2
%else
       mov eax, %1
```

```
        vmwrite eax, %2
%endif
%endmacro
;-----------------------------------------------
; SetVmcsField
; input:
;       %1 - VMCS ID
;       %2 - value
; output:
;       none
;-----------------------------------------------
%macro SetVmcsField    2
%if __BITS__ == 64
        mov rsi, %1

%if %2 == eax
        mov rdi, rax
%elif %2 == ecx
        mov rdi, rcx
%elif %2 == edx
        mov rdi, rdx
%elif %2 == ebx
        mov rdi, rbx
%elif %2 == esp
        mov rdi, rsp
%elif %2 == ebp
        mov rdi, rbp
%elif %2 == esi
        mov rdi, rsi
%elif %2 == edi
        mov rdi, rdi
%else
        mov rdi, %2
%endif
        vmwrite rsi, rdi
%else
        mov esi, %1
        mov edi, %2
        vmwrite esi, edi
%endif
%endmacro
```

为了使用方便，在 inc\vmx.inc 文件里也实现了两个宏：**DoVmWrite** 与 **SetVmcsField**，用来写 VMCS 字段。DoVmWrite 宏与 DoVmRead 宏相对应，SetVmcsField 宏与 GetVmcsField 宏相对应。这里的 SetVmcsField 宏相对复杂一些，额外使用了 8 个条件判断，目的是当这个宏使用在 64 位模式下保证正确。

2.6.6 VMX 模式管理指令

此类指令共有 4 条，分别为：VMXON、VMXOFF、VMLAUNCH 以及 VMRESUME 指令。此类指令在启用 SMM 的“dual-monitor treatment”功能后，情况会变得复杂些。

2.6.6.1 VMXON 指令

执行 VMXON 指令成功后进入处理器的 VMX root-operation 模式，并且初始化 current-VMCS pointer 为 FFFFFFFF_FFFFFFFFh 值。在 root 环境下，**INIT** 与 **SIPI** 信号会无条件地被阻塞，也不允许进入 **A20M** 模式。因此，进入 VMX operation 前必须**启用 A20 地址线**，关闭地址的“wrapped”机制。

VMXON 指令的内存操作数里需要提供 64 位 VMXON 区域物理地址（另见第 2.3.3 节描述）。处理器会检查：（1）VMXON 指针是否超过物理地址宽度并且在 4K 边界上对齐。（2）VMXON 区域的头 DWORD 值是否符合 VMCS ID。这些检查不通过时产生 VMfailInvalid 类型失败。

当执行 VMXON 指令时，如果处理器已经进入到 VMX root operation 模式，将会产生 VMfailValid 失败，指令错误编号是表 2-5 上所列的 15（VMXON 指令执行在 VMX root operation 里）。指令执行在 non-root 环境里则会产生 VM-exit。

2.6.6.2 VMXOFF 指令

VMXOFF 指令无操作数，指令执行成功后将关闭处理器的 VMX operation 模式。如果开启了 SMM 双重监控处理机制，需要首先关闭 SMM 双重监控处理机制后再执行 VMXOFF 指令才能退出 VMX operation 模式，否则将产生 VMfailValid 类型失败。

关闭 SMM 双重监控处理机制做法是：在 root 模式里执行 VMCALL 指令切入 STM（SMM-transfer monitor）后，将当前 SMM-transfer VMCS 内 VM-entry 控制字段的“deactivate dual-monitor treatment”位置 1，再执行 VMRESUME 指令返回到 executive monitor。

2.6.6.3 VMLAUNCH 指令

VMLAUNCH 指令被使用于首次发起 VM-entry 操作。因此 current-VMCS 的状态必须为“**clear**”，否则产生 VMfailValid 类型失败。在发起 VM entry 操作前的一系列动作里，需要使用第 2.6.5.3 节里描述的 VMCLEAR 指令将目标 VMCS 状态置为“clear”状态，然后使用 VMPTRLD 指令加载该 VMCS 为 current-VMCS。

VM entry 操作是一个很复杂的过程，分为三个阶段进行，每个阶段会进行一些必要的检查。检查不通过时，这三个阶段的后续处理也不同，如下所示。

（1）执行基本检查。包括：

- 可能产生的指令异常（#UD 或者#GP 异常）。
- current-VMCS pointer 是否有效。
- 前面所说的当前 VMCS 状态是否为“**clear**”。
- 执行的 VMLAUNCH 指令是否被“**MOV-SS**”指令阻塞。

（2）对 VMCS 内的 VM-execution、VM-exit、VM-entry 以及 host-state 区域的各个字段进行检查。

（3）对 VMCS 的 guest-state 区域的各个字段进行检查。

在第 1 步里的检查失败后将执行 VMLAUNCH 指令的下一条指令。当 current-VMCS pointer 无效（如 FFFFFFFF_FFFFFFFFh）时产生 VMfailInvalid 失败。当 current-VMCS 为**非“clear”**或者执行的 VMLAUNCH 指令被“MOV-SS”指令阻塞时，产生 VMfailValid 失败。产生异常时转入执行异常处理例程。

这一步里，有一个需要关注的“blocking by MOV-SS”阻塞状态，在下面的情形里：

```
mov ss, ax                    ; 更新 SS 寄存器，或者执行 pop ss 指令
vmlaunch                      ; vmlaunch 有“blocking by MOV-SS”阻塞状态
```

上述情形，当 VMLAUNCH 或 VMRESUME 指令执行在 MOV-SS 或 POP-SS 指令后，那么就会产生 VMfailValid 失败，产生的指令错误号为 26，指示“VMLAUNCH 或 VMRESUME 指令将被 MOV-SS 阻塞”。

在第 2 步的检查通不过时，也产生 VMfailValid 失败。指令错误号为 7，指示“VM-entry 操作时，current-VMCS 内含有无效的控制字段”。控制字段包括：VM-execution、VM-exit 以及 VM-entry 区域内的控制字段。也可能产生的指令错误号为 8，指示“VM-entry 操作时，current-VMCS 内含有无效的 host-state 区域字段”（如表 2-5 所示）。失败后也执行 VMLAUNCH 指令的下一条指令。

在第 3 步里，当检查通不过时会产生 VM-exit 行为。处理器在 Exit reason 字段里记录产生 VM-exit 原因的一个编号。它的编号是 33，指示“由于无效的 guest-state 字段而产生 VM-entry 失败”。处理器会加载 host-state 的 context 信息，然后转入执行 host 的入口点。

2.6.6.4 VMRESUME 指令

VMRESUME 指令执行与 VMLAUNCH 相同的动作，但是 VMRESUME 指令使用于在 VM-exit 产生后再次进入 VM。因此，VMRESUME 指令检查 current-VMCS 的状态是否为“**launched**”，否则产生 VMfailValid 失败。指令错误编号为 5，指示“VM-entry 操作时，current-VMCS 状态为非 launched”。

2.6.6.5 返回到 executive monitor

在启用 SMM 双重监控处理机制的情况下，有一种 VM-entry 操作叫做“VM-entry that return from SMM”，直译为“从 SMM 退出进入 VM”。注意，这种情形**只会发生在 VMRESUME 指令**里。

这是两个 monitor 之间的切换，这种情形是从 **SMM-transfer monitor** 切换到 **executive monitor**。在这种情况下，在前面第 2.6.6.3 节所述的第 2 步检查里，处理器还会进行另一些检查（这里的 current-VMCS 等于 SMM-transfer VMCS），如下所示。

（1）检查在 current-VMCS 内的 executive-VMCS pointer 字段的 executive-VMCS 指针是否有效。否则产生 VMfailValid 失败，错误编号为 16，指示：“current-VMCS 含有无效的 executive-VMCS 指针”。

（2）如果 current-VMCS 内的 VM-entry 字段的“deactivate dual-monitor treatment”位为 1，检查 executive-VMCS 指针是否为 VMXON 指针。否则产生 VMfailValid 失败，错误编号为 18，指示：“current-VMCS 内的 executive-VMCS 指针不是 VMXON 指针”。

（3）如果 executive-VMCS 指针不是 VMXON 指针，那么 executive-VMCS 指针在切换到 executive monitor 后，会被加载为新的 current-VMCS 指针。此时 executive-VMCS 的状态必须为“**launched**”，否则产生 VMfailValid 失败，错误编号为 17，指示：“executive-VMCS 的状态为非 launched 状态”。

（4）当 executive-VMCS 属于 VMXON region 时（即返回到 root），不会检查 VM-execution 控制字段。当 executive-VMCS 不是 VMXON region 时（即返回到 non-root），处理器对 executive-VMCS 的 VM-execution 控制字段执行所有检查（参见第 4.4.1 节）。检查不通过时产生 VMfailValid 失败，错误编号为 25，指示“executive-VMCS 内含有无效的 VM-execution 控制字段”。

（5）当 executive-VMCS 属于 VMXON region 时，executive-VMCS 不能指示“注入一个事件”（除了 pending MTF VM-exit 事件外），参见第 3.6.3 节。

（6）当 executive-VMCS 属于 VMXON region 时（返回 root），所有 guest-state 区域的字段检查基于 VM-execution 控制字段为 0 的情况下，activity state 字段不能指示为 wait-for-SIPI 状态。当 executive-VMCS 不是 VMXON 时（返回 non-root），guest-state 区域字段的检查基于 VM-execution 控制字段相应的设置。

上面的检查失败后，会执行 SMM-transfer monitor 中 VMRESUME 指令的下一条指令。

2.6.7　cache 刷新指令

VMX 架构下提供了两条用于刷新 cache 的指令：INVEPT 与 INVVPID 指令。它们都对 TLBs 与 paging-structure caches 进行刷新。关于 TLBs 与 paging-structure caches 的描述可以参考《x86/x64 体系探索及编程》第 11.6 节，或者《Intel 开发人员手册》Vol3A 第 4.10 节。

在 VMX 架构下实现了三类映射途径下的 TLB caches 与 paging-structure caches 信息，它们是：

（1）**linear mapping**，当 EPT 机制未启用时（或者在 VMX root opeation 模式下），这类 cache 信息用来缓存 linear address 到 physical address 的转换（详见第 6.2.1 节）。

（2）**guest-physical mapping**，当 EPT 机制启用时，这类 cache 信息用来缓存 guest-physical address 到 host-physical address 的转换（详见第 6.2.2 节）。

（3）**combined mapping**，当 EPT 机制启用时，这类 cache 信息结合了 linear address 和 guest-physical address 到 host-physical address 的转换。也就是缓存了 linear address 到 host-physical address 的转换信息（详见第 6.2.3 节）。

这几类 cache 信息，我们将在后面的篇章里进行探讨（参见第 6.2 节）。INVEPT 与 INVVPID 指令的不同之处就是：刷新的 cache 信息，以及刷新的 cache 域。

2.6.7.1 INVEPT 指令

在启用 EPT 机制时，可以使用 INVEPT 指令对“GPA 转换 HPA”而产生的相关 cache 进行刷新。它根据提供的 EPTP 值（EPT pointer）来刷新 guest-physical mapping（guest-physical address 转换到 host-physical address）和 combined mapping（linear address 转换到 host-physical address）产生的 cache 信息。

```
invept rax, [InveptDescriptor]    ; 刷新 TLBs 及 paging-structure caches
```

在上面的指令示例里，rax 寄存器提供 **INVEPT type**，指示使用何种刷新方式。而内存操作数里存放着 INVEPT 描述符，EPTP 值提供在这个 INVEPT 描述符里，如图 2-13 所示。

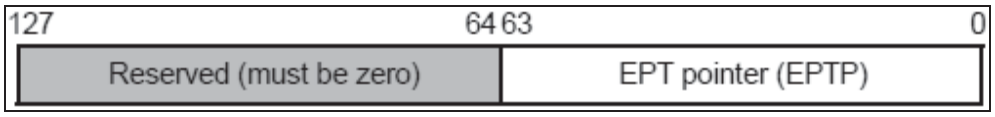

图 2-13

INVEPT 描述符的结构，共 16 个字节，bits 63:0 存放 EPTP 值。EPTP 值的 bits *N*–1:12 指向 EPT 结构的 PML4T 表，INVEPT 指令将 EPTP[*N*–1:12]所引伸出来的层级转换表结构作为依据进行刷新 cache，这个 EPTP 字段 bits *N*–1:12（*N*=MAXPHYADDR）提供的值被称为“**EP4TA**”（EPT PML4T address，扩展页表的 PML4T 地址）。

INVEPT 指令支持两种刷新类型：single-context 与 all-context。详见第 6.2.6.4 节所述。

（1）当 type 值为 1 时，使用 **single-context** 刷新方式。处理器刷新 INVEPT 描述符里提供的 EP4TA 值所对应的 guest-physical mapping 与 combined mapping 信息。

（2）当 type 值为 2 时，使用 **all-context** 刷新方式。处理器刷新所有 EP4TA 值对应的 guest-physical mapping 与 combined mapping 信息。也就是说，此时将忽略 INVEPT 描述符。

另外需要特别注意的是：处理器也刷新**所有 VPID 与 PCID 值**所对应的 combined mappings 信息。软件应该要查询处理器的 INVEPT 指令是否支持上述的 type 值。

前面第 2.5.13 节描述了 INVEPT 指令支持的刷新类型，当使用不支持的类型时产生 VMfailValid 失败，错误编号为 28，指示“无效的 INVEPT/INVVPID 操作数”。

2.6.7.2 INVVPID 指令

INVVPID 指令依据提供的 VPID 值对 linear mapping 及 combined mapping 的 cache 信息进行刷新。也就是 INVVPID 指令可以刷新 EPT 机制**启用**或者**未启用**时的线性地址到物理地址转换而产生的 cache 信息。

```
invvpid rax, [InvvpidDescriptor]  ; 刷新 TLBs 及 paging-structure caches
```

INVVPID 指令也需要在寄存器操作数里提供 **INVVPID type** 值，在内存操作数里提供 INVVPID 描述符。INVVPID 描述符结构如图 2-14 所示。

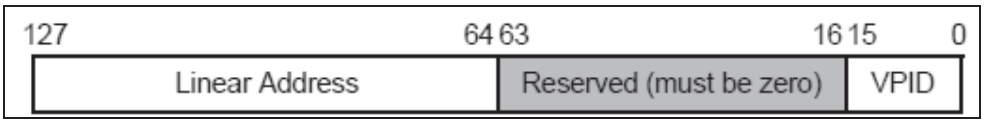

图 2-14

INVVPID 描述符的 bits 127:64 提供线性地址，bits 15:0 提供 VPID 值。INVVPID 指令依据这两个值进行刷新。INVVPID 指令支持 4 个刷新类型（详见第 6.2.6.3 节所述）。

- 当 type 值为 0 时，使用 **individual-address** 刷新方式。指令将刷新目标 VPID，所有 PCID 以及所有 EP4TA 域内与目标线性地址匹配的 cache 信息，具体为：① 匹配描述符内提供的目标线性址与目标 VPID 值。② 所有 PCID 域下对应的 linear mappings 与 combined mappings 信息。③ 所有 EP4TA 域下对应的 combined mappings 信息。
- 当 type 值为 1 时，使用 **single-context** 刷新方式。指令将刷新目标 VPID，所有 PCID 以及所有 EP4TA 域的 cache 信息，具体为：① 匹配描述符内提供目标 VPID 值。② 所有 PCID 域下对应的 linear mappings 与 combined mappings 信息。③ 所有 EP4TA 域下对应的 combined mappings 信息。
- 当 type 值为 2 时，使用 **all-context** 刷新方式。指令将刷新默认 VPID 值（0000H）外的所有 VPID，所有 PCID 以及所有 EP4TA 域的 cache 信息，具体为：① 所有 VPID 值（除了 0000H）。② 所有 PCID 域下对应的 linear mappings 与 combined mappings 信息。③ 所有 EP4TA 域下对应的 combined mappings 信息。
- 当 type 值为 3 时，使用 **single-context-retaining-global** 刷新方式。指令行为与类型 2 的 single-context 刷新方式相同，除了保留 global 的转换表外。

在所有的刷新方式里，都不能刷新 VPID 值为 0000H 的 cache 信息。否则产生 VMfailValid 失败，错误编号为 28，指示“无效的 INVEPT/INVVPID 操作数”。

软件也应该使用第 2.5.13 节里描述的方式，查询当前处理器支持哪种刷新类型。如果提供不支持的刷新类型，也同样产生编号为 28 的 VMfailValid 失败。

2.6.8　调用服务例程指令

VMX 架构提供了两个调用服务例程指令：VMCALL 与 VMFUNC 指令。它们服务的对象不同，VMCALL 指令使用在 VMM 里，而 VMFUNC 指令使用在 VM 里。

2.6.8.1　VMCALL 指令

利用 VMCALL 指令可以实现 SMM 的 dual-monitor treatment（SMM 双重监控处理）机制。VMCALL 指令在 non-root 里执行将会产生 VM-exit 行为，但在 root 环境里执行 VMCALL 指令，当满足检查条件时，在 VMM 里产生被称为“**SMM VM-exit**”的退出行

为，从而切换到 SMM 模式的 SMM-transfer monitor 里执行。这个 SMM-transfer monitor 入口地址提供在 MSEG 区域头部（由 IA32_SMM_MONITOR_CTL[31:12]提供）。

在 VMX root operation 里执行 VMCALL 指令，除了可能产生异常（#UD 或#GP）外，有两种可能：（1）指令失败（VMfailInvalid 或 VMfailValid）。（2）产生“SMM VM-exit”，激活 SMM 双重监控处理功能。

IA32_SMM_MONITOR_CTL 寄存器的 bit 0 为 valid 位。只有当 bit 0 为 1 时，才允许使用 VMCALL 指令通过切入 SMM-transfer monitor 执行来激活 SMM 双重监控处理机制。否则将产生 VMfailValid 失败，指示“VMCALL 指令执行在 VMX root operation 模式里”。

2.6.8.2 VMFUNC 指令

VMFUNC 指令是唯一能在 non-root 环境里使用的 VMX 指令。当允许并且设置 secondary processor-based control 字段的“enable VM functions”位为 1 时，允许 VM 里执行 VMFUNC 指令调用服务例程，否则将产生#UD 异常。注意，VMFUNC 指令在 VMX root operation 模式里执行也会产生#UD 异常。

执行 VMFUNC 指令前，在 eax 寄存器里放入功能号。然而，在执行某个功能号时，也需要在 VM-functions control 字段的相应位置位。VM-functions control 字段是一个 64 位值，因此 VMX 架构最多只支持 0 至 63 的功能号。如果提供的功能号大于 63，则会产生#UD 异常。

当启用“enable VM functions”功能，提供一个功能号给 eax 寄存器执行 VMFUNC 指令，但 VM-functions control 字段（参见第 3.5.20 节）相应位为 0 值，则会产生 VM-exit 行为。在成功执行 VMFUNC 指令的情况下不会产生 VM-exit 行为。

当前 VMX 架构下只实现了一个功能号 0，它是“**EPTP switching**”服务例程（参见第 6.1.11 节）。使用它则需要在 VM-functions control 字段的 bit 0 进行置位。软件使用前需要通过第 2.5.9 节描述的方法来检测是否支持该项功能。

为了支持 EPT switching 服务例程，VMX 架构添加了一个 EPTP-list address 字段（参见第 3.5.21 节），提供 512 个 EPT 值供切换。使用时需要在 EAX 寄存器放入 0 号功能，在 ECX 寄存器放入 EPT-list entry 的编号。当 ECX 的值大于 511 时将产生 VM-exit。