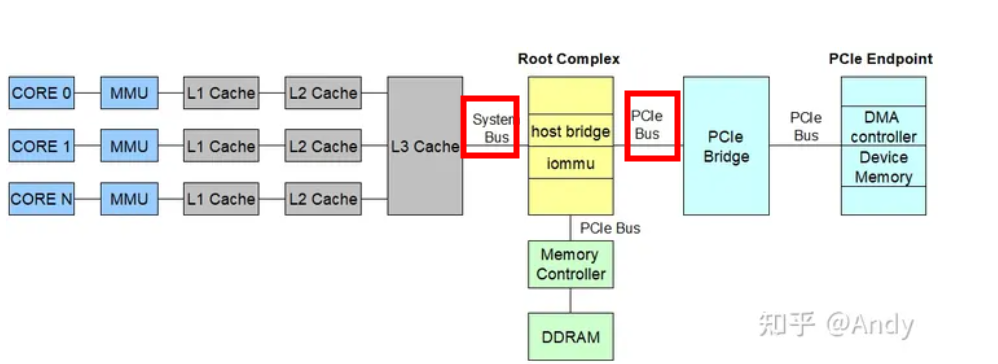
# 一文读懂 内存DMA 及 设备内存控制



## 序言

对于PCIe 设备（PCIe Endpoint）来说，其和CPU CORE、DRAM 的交互，主要涉及两种类型的内存访问：

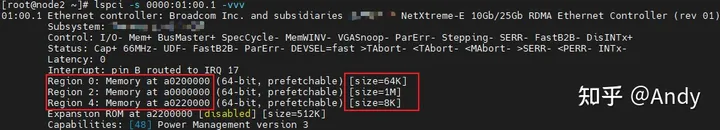
* 设备内存访问：PCIe 设备的 Device Memory（设备内存）的访问，例如CPU 需要读写配置 PCIe 网卡或显卡的 寄存器（设备内存）。发起者是CPU，响应者是设备。
* DMA内存访问：PCIe 设备需要 DMA 读写 主机的 DRAM 内存，例如 网卡收到数据报文后，需要将其上报主机，则通过 PCIe Endpoint 中的 DMA 控制器 进行 DMA 写 主机的 DRAM 内存。发起者是设备DMA控制器，响应者主机内存DRAM，不需要CPU的参与。

以CPU CORE 发起设备内存读访问为例。从上图的左侧开始，CPU CORE 执行程序指令时，发出内存读写访问的地址是虚拟地址，接着MMU（内存管理单元或称为地址翻译单元更好） 将该虚拟地址转换为物理地址。如果物理地址访问的数据不在Cache中，则通过 Root Complex 中的 host bridge，将物理地址转换为总线地址，最后基于总线地址访问设备内存空间（Device Memory）。本文将基于上图的计算机系统架构，追本溯源，详细的解释设备内存 和 DMA内存 的访问细节。

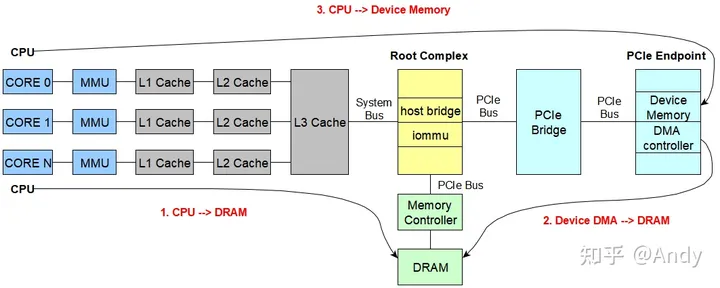
## 重要概念

对于本文架构图中涉及的主要概念解释如下，如果不想被概念搞晕，建议暂时滤过，等不清楚时再回查即可。

* System Bus：系统总线，用于CPU Package 内部子系统之间的连接，例如连接L3 Cache 和 Root Complex。
* Cache：X86 的 Cache 通常分为三级，各个CPU CORE 有自己的 L1和L2 Cache，所有CPU CORE 共享 L3 Cache。Intel I7 L3 为 8MB，L2 256KB，L1 共64KB，所以Cache Size从 L1 到L3是递增。对于程序指令或数据都可以加载到Cache，加载的单位是 Cache Line，通常为 64B。
* Root Complex：其中包括了两个重要组件，一个是 PCIe Root Bridge，通常称为 Host Bridge，其 Bus：Device. Function 为 00:00.0；另一个是 iommu，其用于将设备（如PCIe Endpoint 网卡）发起DMA读写DRAM使用的总线地址，转换为DRAM内存的 物理地址。
* Memory Controller：DRMA 内存控制器，通常包含在CPU package 中，用于控制对内存的访问。
* PCIe bus：PCIe 总线，用于连接 Host Bridge 到 PCIe Switch，PCIe Bridge 以及 PCIe Endpoint。
* DMA Controller：DMA 控制器，其通常位于 PCIe Endpoint 内，用于设备对 DRAM 中数据进行 ”DMA 读”或“DMA 写”。
* Device Memory：设备内存，用于对设备进行控制，如配置设备、获取设备状态等；在系统对PCIe 总线进行深度优先扫描时，会根据设备的 Bar 空间大小（Device Memory Size，如下面的网卡例如，Bar 0，2，4 Size 分别为64K，1M，8K），分配 iomem 中映射的物理地址，下文会提到 iomem 的细节。



## 内存访问



如前面所所述，计算机内部主要涉及两种内存访问。一种是 物理内存DRAM，另一种是 设备内存（如网卡，显卡等PCIe设备）。

内存访问方向

以上两种类型内存的访问，主要包括上图中三个方向：

* CPU -> DRAM: 从CPU 读写物理内存中的数据或程序指令；
* Device DMA -> DRAM: 设备的DMA 控制器发出针对DRAM的 DMA 读写，例如网卡从物理内存中DMA读取数据包进行发包。
* CPU -> Device Memory: 主机CPU 发出针对设备的读写，例如主机需要初始化设置显卡寄存器，让其开始工作。

### 内存访问过程

* A. CPU访问物理内存（CPU -> DRAM）

CPU使用虚拟地址发出内存读写请求

MMU将虚拟地址转为DRAM的物理地址

如果访问数据的物理地址在Cache 有缓存数据，则直接从Cache 读取即可

如果Cache 中没有缓存，则通过内存控制器，访问DRMA（当然这涉及到页表的管理，不在本文的讨论范围），如果有兴趣，强烈建议深入阅读《深入理解计算机系统》

* B. 设备访问物理内存（Device DMA -> DRAM）

外设使用总线地址，发起针对DRAM的DMA读写访问

IOMMU负责将总线地址转为物理地址

通过内存控制器，访问DRMA

* C. 设备内存访问（CPU -> Device Memory）

CPU使用虚拟地址发出内存读写请求

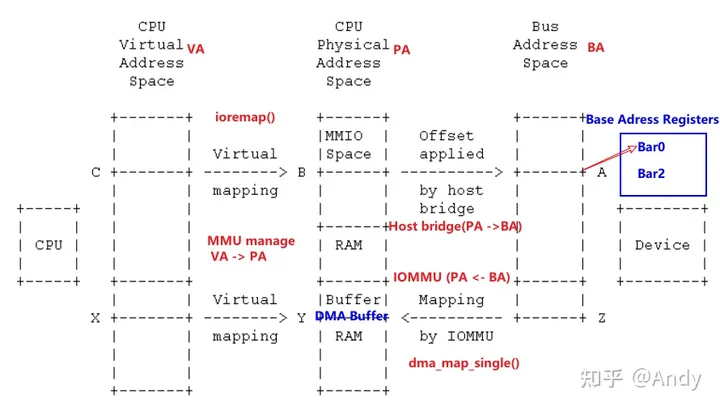
MMU将虚拟地址转为设备内存的物理地址（通过 cat /proc/iomem 可以查看到）

Host Bridge 将物理地址转为总线地址

通过PCIe总线寻址到设备，并进行设备内存读写

总结：访问物理内存时，可以认为 虚拟地址 和 总线地址 都是Virtual的地址，都需要通过一个地址转换硬件（CPU侧是MMU，device侧是IOMMU），将虚拟地址转换为物理内存DRAM的实际物理地址。而访问设备内存时，通过Host Bridge将物理地址转换为设备内存的总线地址。

地址空间



虚拟地址：CPU Virtual Address Space（VA）

虚拟地址 被主机CPU用于访问物理内存，或者设备（PCIe EP Device)的 Bar 空间的内存（寄存器）。每个进程都有相同的虚拟地址空间（例如32位Linux系统，最大支持4G的地址空间，但只有高地址3GB用于进程的虚拟地址空间，低地址或顶端的1G给内核使用）。CPU执行指令访问内存使用的地址是虚拟地址，然后通过MMU、TLB 及 Page Tables 将虚拟地址转换为内存的物理地址。

物理地址：CPU Physical Address Space（PA）

我们可以认为物理地址空间，就是所有硬件的内存映射空间（MMIO- memory map I/O）。可将物理内存RAM看成一种特殊的MMIO空间。OS在给硬件设备编址时，其会看到物理内存 以及设备内存（通过PCIe Bar 空间映射的设备寄存器），所以进行了统一的物理地址的编址。如下，低地址给RAM使用，而PCI 设备内存通常位于高地址。

物理地址空间的编址 可以通过 cat /proc/iomem 查看：

[root@node2 ~]# cat /proc/iomem

00000000-00000fff : Reserved

00001000-0005cfff : System RAM // System RAM：物理内存

0005d000-0005dfff : Reserved

0005e000-0009ffff : System RAM

......

00100000-3fffffff : System RAM // Kernel Space

01000000-01c00ea0 : Kernel code

01e00000-02214fff : Kernel rodata

02400000-026294bf : Kernel data

02893000-02bfffff : Kernel bss

......

8f800000-dfffffff : PCI Bus 0000:00 // PCI Root Bridge Bus 0 下挂PCIe 设备内存的总大小

8f800000-8f9fffff : PCI Bus 0000:02

8fa00000-8fbfffff : PCI Bus 0000:02

8fc00000-8fdfffff : PCI Bus 0000:04

8fe00000-8fffffff : PCI Bus 0000:04

90000000-9fffffff : 0000:00:02.0

90000000-902fffff : BOOTFB

a0000000-a02fffff : PCI Bus 0000:01 // PCIe bus 01 下挂PCIe 设备内存的总大小

a0000000-a00fffff : 0000:01:00.1 // PCIe 网卡 Bar 2 空间（设备内存）映射的物理内存

a0000000-a00fffff : bnxt\_en

a0100000-a01fffff : 0000:01:00.0

a0100000-a01fffff : bnxt\_en

a0200000-a020ffff : 0000:01:00.1 // PCIe 网卡 Bar 0 空间（设备内存）映射的物理内存

a0200000-a020ffff : bnxt\_en

a0210000-a021ffff : 0000:01:00.0

a0210000-a021ffff : bnxt\_en

a0220000-a0221fff : 0000:01:00.1 // PCIe 网卡 Bar 1 空间（设备内存）映射的物理内存

a0220000-a0221fff : bnxt\_en

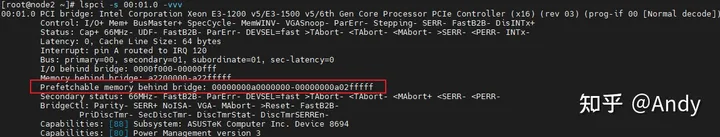
a0222000-a0223fff : 0000:01:00.0

a0222000-a0223fff : bnxt\_en

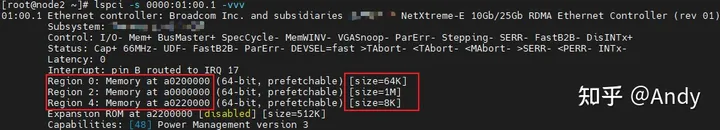
a0224000-a0243fff : 0000:01:00.1

a0244000-a0263fff : 0000:01:00.1

在作者使用这台主机上，物理地址范围 a0000000-a02fffff 对应: PCI Bus 0000:01。该Bus位于PCI Bridge 和 PCI Endpoint之间（可以回到文章开始的图查看），所以可以通过命令 （ lspci -s 00:01.0 -vvv ）查看到 Bus 上一级 Bridge 的信息，得到Bridge下挂设备的总地址空间，即下图中可以被CPU访问预取的 Memory 范围：00000000a0000000-00000000a02fffff



而在该PCI Bridge下，下挂PCIe Endpoint的其中一个Function（01:00.1），其 Bar2 对应物理地址范围：a0000000-a00fffff : 0000:01:00.1（Bus:Device.Function)，即为 PCIe 网卡 Bar 2（下图Region 2）空间（设备内存）映射的物理地址空间。



### 总线地址：Bus Address Space（BA）

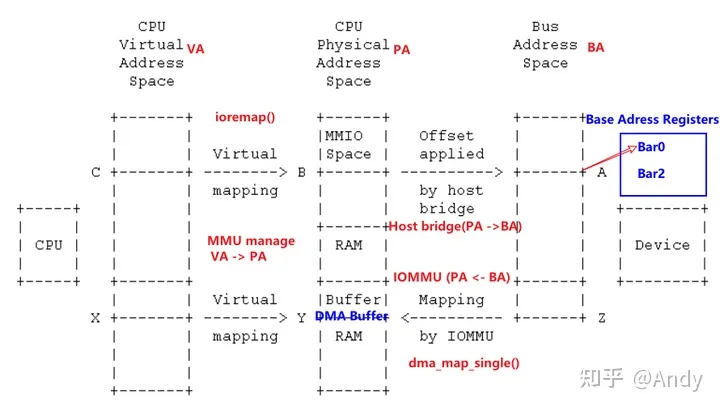
总线地址是为 终端设备读写DMA 内存 或者 主机读写设备配置空间（PCIe Bar）使用的总线地址空间的地址（PCI 总线）。

我们在调用【dma = dma\_map\_single(device, buf, size, DMA\_TO\_DEVICE)】，将数据 buf（虚拟地址） 建立 dma地址映射给device 访问时，返回的类型为 dma\_addr\_t 的 dma 地址，就是 bus address。该地址可以传给设备用于DMA的物理内存数据读取。

**在一些系统中，总线地址和物理地址是一样的**（我们可以调用 phy = virt\_to\_phys(buf) 得到虚拟地址buf 对应的物理地址 phy， 确认 phy 和 dma 地址是否相同）。Host Bridge和 IOMMU可以实现物理地址和总线地址的任意映射。

注意：从设备的角度，不管是访问设备内存，或者设备发起DMA读写，都是使用的总线地址。

### 访问过程详解



#### CPU 读取设备内存

1. 首先，CPU需要通过ioremap，将虚拟地址A映射到物理地址空间 MMIO 中物理机地址B；
2. 然后，CPU发起 ioread、iowrite 请求，发起时用的虚拟地址C，接着MMU转换为物理地址B；
3. 最后，PCIe Host Bridge 将物理地址B转换为总线地址A，通过总线地址A对Bar空间进行读写。

注意：对Bar Register 的读写基于 PCIe的 Configuration TLP 操作。

代码示例：

/\* CPU ioremap \*/

struct pci\_dev \*pdev;

u8 \_\_iomem \*hw\_addr = ioremap(pci\_resource\_start(pdev, 0),

pci\_resource\_len(pdev, 0));

u32 val = 10;

u32 reg\_offset = 0;

/\* 将 10 写入hw\_addr[reg\_offset] \*/

writel(val, &hw\_addr[reg\_offset]);

value = readl(&hw\_addr[reg]);

#### Device DAM 访问物理内存

首先，CPU基于虚拟机地址X，通过MMU，将数据写入物理地址Y对应的 物理内存中；

然后，CPU 调用如dma\_map\_single 这样的API，将总线地址Z 映射到 虚拟地址X和物理地址Y。

最后，Device 通过总线地址Z，发出DMA读请求（PCIe Memeory Read），接着IOMMU将总线地址翻译为物理地址Y 去 读写物理内存 DMA Buffer.

编程示例：

struct device \*dev; /\* device for DMA mapping \*/

struct sk\_buff \*skb；

dma = dma\_map\_single(dev, skb->data, size, DMA\_TO\_DEVICE);

/\* tx\_desc 是网卡发包时用到的描述符，

\* 硬件设备可以通过DMA 访问到描述符绑定的dma地址，然后硬件可以基于该DMA地址发起内存访问

\*/

tx\_desc->read.buffer\_addr = cpu\_to\_le64(dma);

### DMA映射编程

常用的DMA映射有两种类型，一种是一致性DMA映射（consistent DMA mapping）；另一种是流式DMA映射（streaming DMA mapping）。理解这两种常用的 DMA映射类型，是编程的基础。

#### 一致性DMA映射

一致性DMA 映射关闭了 L1/L2/L3 Cache。首先，当 CPU 写入数据时，则会直接放入内存，而不会在Cache进行缓存，所以设备可以立即DMA读取到CPU写入的数据；其次，当设备DMA写入数据到内存后，则CPU可以立即读取到该变化的数据，而不会读取Cache中的脏数据，因为Cache关闭了。

Consistent DMA mapping的常用场景：

网卡驱动程序 和 网卡DMA控制器往往是通过内存中的一些描述符（形成环）进行交互（描述符中包括收发数据包用到的大块DMA内存地址），这些保存描述符的memory，需要被主机CPU和网卡DMA控制器频繁的读写，并且在任何一端写，都需要另一端立即可以访问到，所以通常采用Consistent DMA mapping比较方便。

2. SCSI硬件适配器上的DMA 与 主存中的一些数据结构（mailbox command）进行交互，这些保存mailbox command的memory一般采用Consistent DMA mapping。

代码示例

如下调用 dma\_alloc\_coherent 分配一块一致性DMA内存（取自intel ixgbe驱动 linux-4.18.0-348\drivers\net\ethernet\intel\ixgbe\ixgbe.h & linux-4.18.0-348\drivers\net\ethernet\intel\ixgbe\ixgbe\_main.c）：

struct ixgbe\_ring {

struct device \*dev; /\* device for DMA mapping \*/

void \*desc; /\* descriptor ring memory \*/

dma\_addr\_t dma; /\* phys. address of descriptor ring \*/

unsigned int size; /\* length in bytes \*/

u16 count; /\* amount of descriptors \*/

}

/\*\*

\* ixgbe\_setup\_tx\_resources - allocate Tx resources (Descriptors)

\* @tx\_ring: tx descriptor ring (for a specific queue) to setup

\*

\* Return 0 on success, negative on failure

\*\*/

int ixgbe\_setup\_tx\_resources(struct ixgbe\_ring \*tx\_ring)

{

struct device \*dev = tx\_ring->dev;

// 返回是 DMA地址 对应的虚拟地址 （tx\_ring->desc）

tx\_ring->desc = dma\_alloc\_coherent(dev, // 进行DMA映射的PCI设备对应的device

tx\_ring->size, // 描述符环对应的总大小（bytes）

&tx\_ring->dma, // 输出：DMA 地址

GFP\_KERNEL);

...

}

参考内核 Document/core-api/dma-api.rst

void \*

dma\_alloc\_coherent(struct device \*dev, size\_t size,

dma\_addr\_t \*dma\_handle, gfp\_t flag)

Consistent memory is memory for which a write by either the device or

the processor can immediately be read by the processor or device

without having to worry about caching effects.

#### 流式DMA映射

因为一致性DMA关闭了Cache，虽然使用带来了方便，但是会牺牲数据读写的性能。例如Intel当前的CPU package，支持在 PCIe Endpoint DMA 访问内存时，将数据放入到L3 Cache，然后DMA 控制器从L3 Cache 读取数据（这是对我们DMA控制器直接访问DRAM内存常识的挑战）。而这些硬件的优化，在使用流式DMA影射时，可以充分发挥性能的优势。

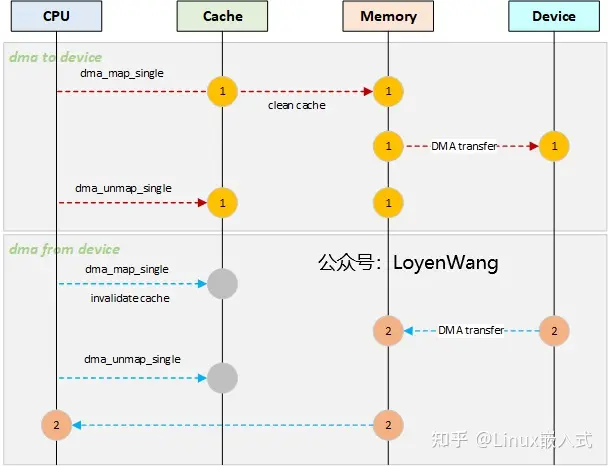
streaming DMA mapping的常用场景：

网卡进行数据传输使用的DMA buffer，发送数据是，主机准备好DMA Buffer，由硬件DMA读取；接受数据时，也是主机准备好DMA Buffer，由硬件DMA 写入接受到的数据。

文件系统中的各种数据buffer，这些buffer中的数据最终要被读写到SCSI设备上去

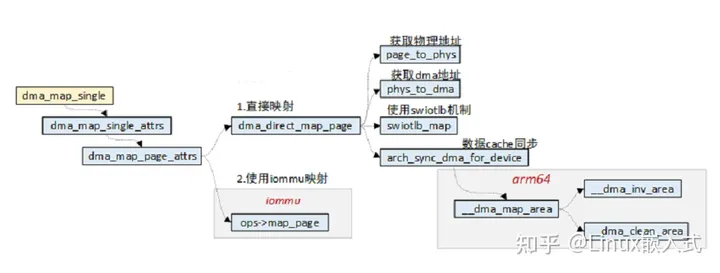
* streaming mapping：对应于non-coherence设备；
* 通常在单次DMA传输时进行map，在传输完成后进行unmap（除非调用了dma\_sync\_XXX()函数）；
* non-coherence设备，由于buffer不与其他数据共享cache line，通常会work better；

先看一下数据一致性问题：



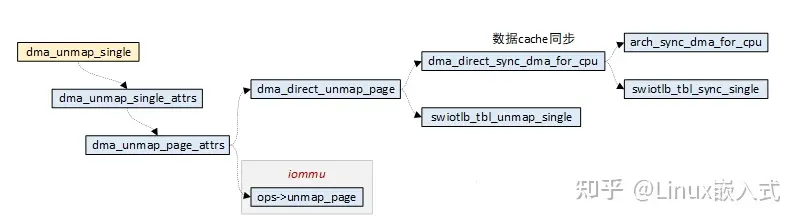
* dma to device时，需要将cache中的数据flush到memory中；
* dma from device时，需要先将cache中的数据invalidate掉，避免CPU读取的是原来的数据；

dma\_map\_single函数如下：

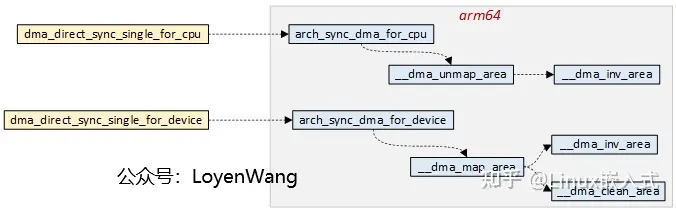


* map操作时存在两种方式，直接映射或使用iommu来完成映射；

dma\_unmap\_single是逆操作：



从上述函数中可以看到，最终都会调用到arch相关的cache操作，这个与体系结构是强相关的，以arm64为例：



* 代码示例1

如下dma\_map\_single 将 虚拟地址 skb->data 指向的内存，映射到dma 总线地址上，然后将dma 地址写入描述符给硬件DMA读取 skb->data 指向的内存数据。

需要注意的是，skb->data 指向的数据区域需要是连续物理内存。内核采用带k字的分配函数（如kmalloc）即得到连续的物理内存，而使用v开头的分配函数，如vmalloc，则不能保证。

struct device \*dev; /\* device for DMA mapping \*/

struct sk\_buff \*skb；

dma = dma\_map\_single(dev, skb->data, size, DMA\_TO\_DEVICE);

/\* tx\_desc 是网卡发包时用到的描述符，

\* 硬件设备可以通过DMA 访问到描述符绑定的dma地址，然后硬件可以基于该DMA地址发起内存访问

\*/

tx\_desc->read.buffer\_addr = cpu\_to\_le64(dma);

* 代码示例2

如下调用dev\_alloc\_pages 分配一个page，再调用 dma\_map\_page\_attrs 将该页映射dma 地址，最后调用dma\_sync\_single\_range\_for\_device 将页面的数据同步给设备访问：

struct page \*page;

dma\_addr\_t dma;

struct device \*dev; /\* device for DMA mapping \*/

int page\_size = PAGE\_SIZE;

/\* alloc new page for storage \*/

page = dev\_alloc\_pages(0);

/\* map page for use \*/

dma = dma\_map\_page\_attrs(dev, page, 0,

page\_size,

DMA\_FROM\_DEVICE,

DMA\_ATTR\_SKIP\_CPU\_SYNC);

/\* sync the buffer for use by the device \*/

dma\_sync\_single\_range\_for\_device(dev, dma,

0, page\_size,

DMA\_FROM\_DEVICE);

总结

本文结合计算机系统的架构，我们从内存访问的角度，介绍了各种地址空间（虚拟、物理、总线）的概念。以及物理内存和设备内存访问三个方向（CPU->DRAM, CPU -> Device Memory, Device DMA -> DRAM)。最后介绍了DMA 映射编程常用的方法，如果觉得不过瘾，建议继续研究作者参考的内核 Document。

# PCIe MSI-X 中断编程



## 1. 前言

当前数据中心服务器，CPU基本都是基于PCIE总线和各种设备（例如，内存、显卡和网卡等）相连。而各种PCIE设备采用 MSIX（Message Signaled Interrupt eXtended - 基于消息的信号中断扩展）将中断信号发送给CPU。我们知道MSI最多支持32个中断向量号，而MSI-X中断向量数目最大为2048。那我们编程时如何获取、申请、控制这些中断向量资源呢，本文将为你揭开神秘的面纱。

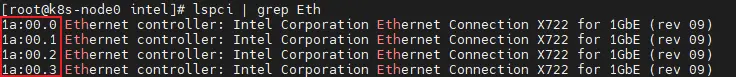
## 2. MSI-X Capability

基于lscpi 命令可以轻易获取到当前系统中PCI 设备的能力，而MSI-X 能力是PCIE设备众多能力中的一个。下面将基于网卡设备查看其 MSI-X Capability配置。

### 2.1 Capability配置

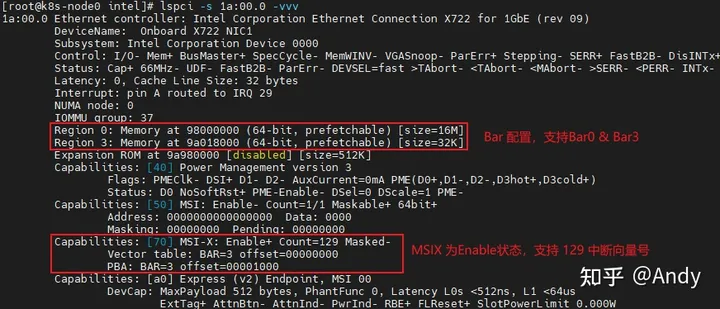
查看系统中的PCIE网卡设备

# lspci | grep Eth



查看 PCIE 设备 1a:00.0 的MSI-X Capability

# lspci -s 1a:00.0 -vvv



lspci -s BDF -vvv 显示pcie设备配置空间信息

从上图PCIE设备的配置空间信息可以看出，该PCIE设备支持两个Bar地址空间（Region 0 & Region 3），

Bar0：总线地址起始位置为 0x98000000， 总大小时 16MB

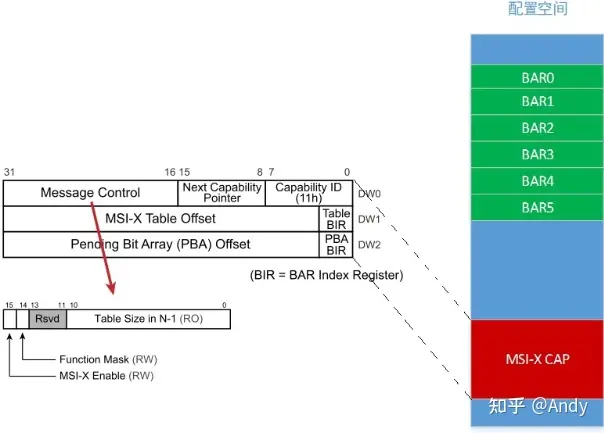
Bar3：总线地址起始位置为 0x9a018000， 总大小时 32KB

MSI-X ：中断能力 为 Enable 状态，支持 129 中断向量号

MSI-X Vector Table：位于 Bar3，起始地址或 offset 为 0

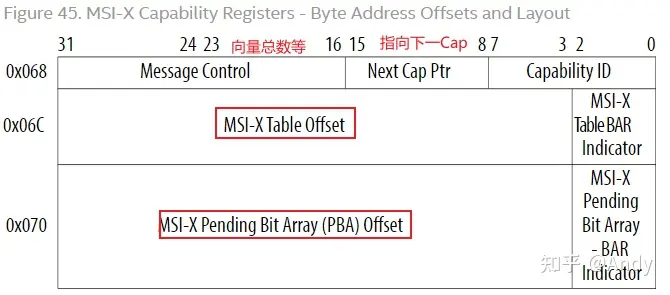
MSI-X PBA (pending table）：也位于Bar3，但起始地址为 0x00001000

下图展示了MSIX Capability在配置空间中的位置，其大小为3个DWORD，即12字节：

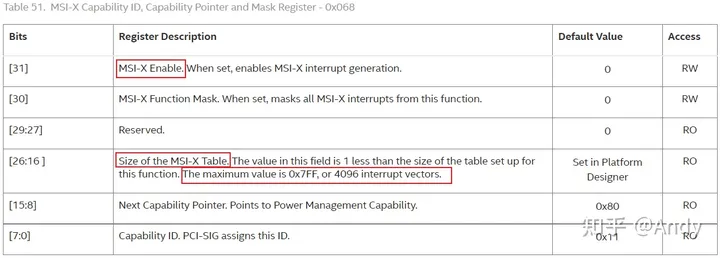


### 2.2 Capability标准

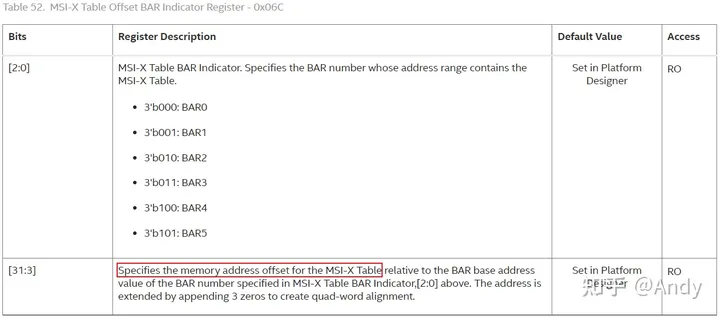
上图第二个红框关于MSI-X Capability的标准如下：



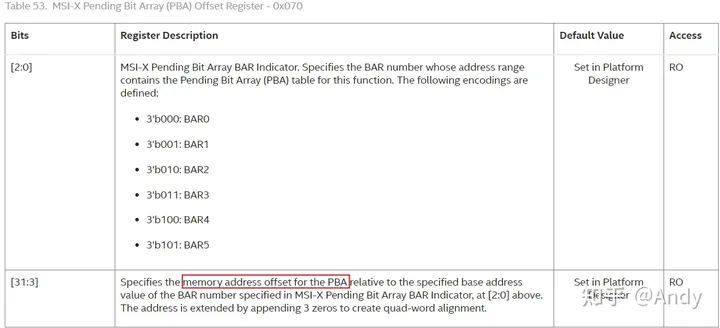
第1个DWORD 0x068位置：关键是 16~26 bit，存放着 Vector Table 的Size，最大为0x7FF，共11位，所以可知一个 PCIE function 最多只能支持 2的11次方，即2048个中断号。



第2个DWORD 0x06C位置：3~31 位是 MSI-X Table位于Bar空间的起始位置



第3个DWORD 0x06D位置：3~31 位是 MSI-X PBA位于Bar空间的起始位置



如何获取MSIX Capability的信息呢？下面将基于内核代码解释读取第一个DOWRD 中 Size of the MSI-X Table（即中断向量总数 Vector Number ）的过程。

### 2.3 编程获取Capability

可以基于接口 pci\_read\_config\_word 来读取PCIE EP（end point）设备的配置空间信息。下面代码用该接口读取MSIX Capability 的第一个DWORD 中的Message Control（16~31 位），从而获取到当前 PCIE EP 真实支持中断号数量。例如上面PCIE 网卡设备的 num\_vectors 为130（129+1）。

#include <linux/pci.h>

/\* MSI-X registers (in MSI-X capability) \*/

#define PCI\_MSIX\_FLAGS 2 /\* Message Control \*/ #define PCI\_MSIX\_FLAGS\_QSIZE 0x07FF /\* Table size \*/

struct pci\_dev \*pdev = pci\_dev;

u16 msix\_config;

int num\_vectors;// pdev 为 PCIE 设备，pdev->msix\_cap为设备配置空间中MSIX Capability的起始位置（PCI\_MSIX\_FLAGS表示读取Message Control）// 读取到的 message control 寄存器放置于 msix\_config 变量 pci\_read\_config\_word(pdev, pdev->msix\_cap + PCI\_MSIX\_FLAGS, &msix\_config);

num\_vectors = ((msix\_config & PCI\_MSIX\_FLAGS\_QSIZE) + 1);

pr\_info("MSIX: num\_vectors=%d\n", num\_vectors);

也可以基于PCI Driver提供的现有接口 pci\_msix\_vec\_count （drivers/pci/msi.c)，来获取给定PCIE设备支持的 中断向量数目(vector number)。

/\*\* \* pci\_msix\_vec\_count - return the number of device's MSI-X table entries \* @dev: pointer to the pci\_dev data structure of MSI-X device function \* This function returns the number of device's MSI-X table entries and \* therefore the number of MSI-X vectors device is capable of sending. \* It returns a negative errno if the device is not capable of sending MSI-X \* interrupts. \*\*/int pci\_msix\_vec\_count(struct pci\_dev \*dev){

u16 control;

if (!dev->msix\_cap)

return -EINVAL;

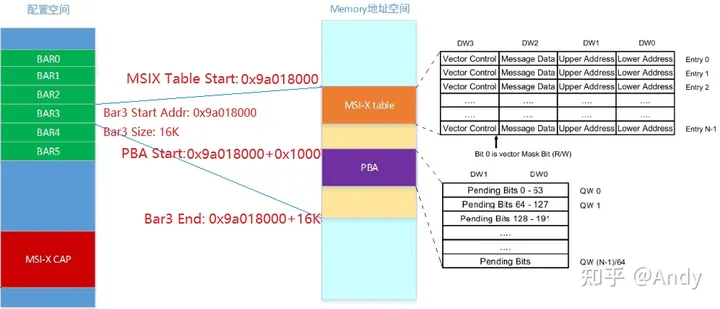
pci\_read\_config\_word(dev, dev->msix\_cap + PCI\_MSIX\_FLAGS, &control);

return msix\_table\_size(control);}

#define msix\_table\_size(flags) ((flags & PCI\_MSIX\_FLAGS\_QSIZE) + 1)

### 3. MSI-X Table

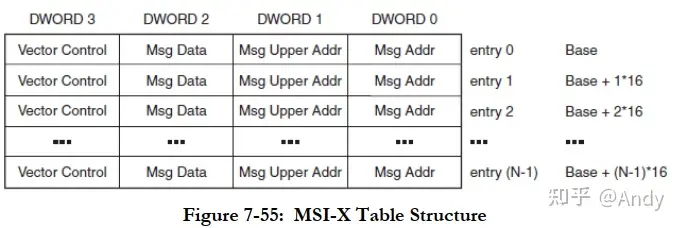
MSIX Table 中存放着所有PCIE 设备的中断向量号，如下图MSIX Table位于BAR3 起始地址 0x9a018000 处，该地址对应PCIE EP 设备中的Memory 地址空间，该Memory 空间是PCIE 设备内部的寄存器空间。



### 3.1 Table 结构

MSIX Table 中每一条Entry 代表一个中断向量，Msg Data 中包括了中断向量号，Msg Addr 中通常包含了多核CPU用于处理 中断的 Local APIC 编号。

从MSI-X table 的结构可以看出，每个entry占用4个DWORD，即16bytes，所以访问第N个Entry的地址是：n\_entry\_address = base address[BAR] + 16 \* n





MSIX Table的结构中每一条Entry的中断信息，是什么时候产生的呢？接下来将从内核代码的角度进行分析。

### 3.2 Table初始化

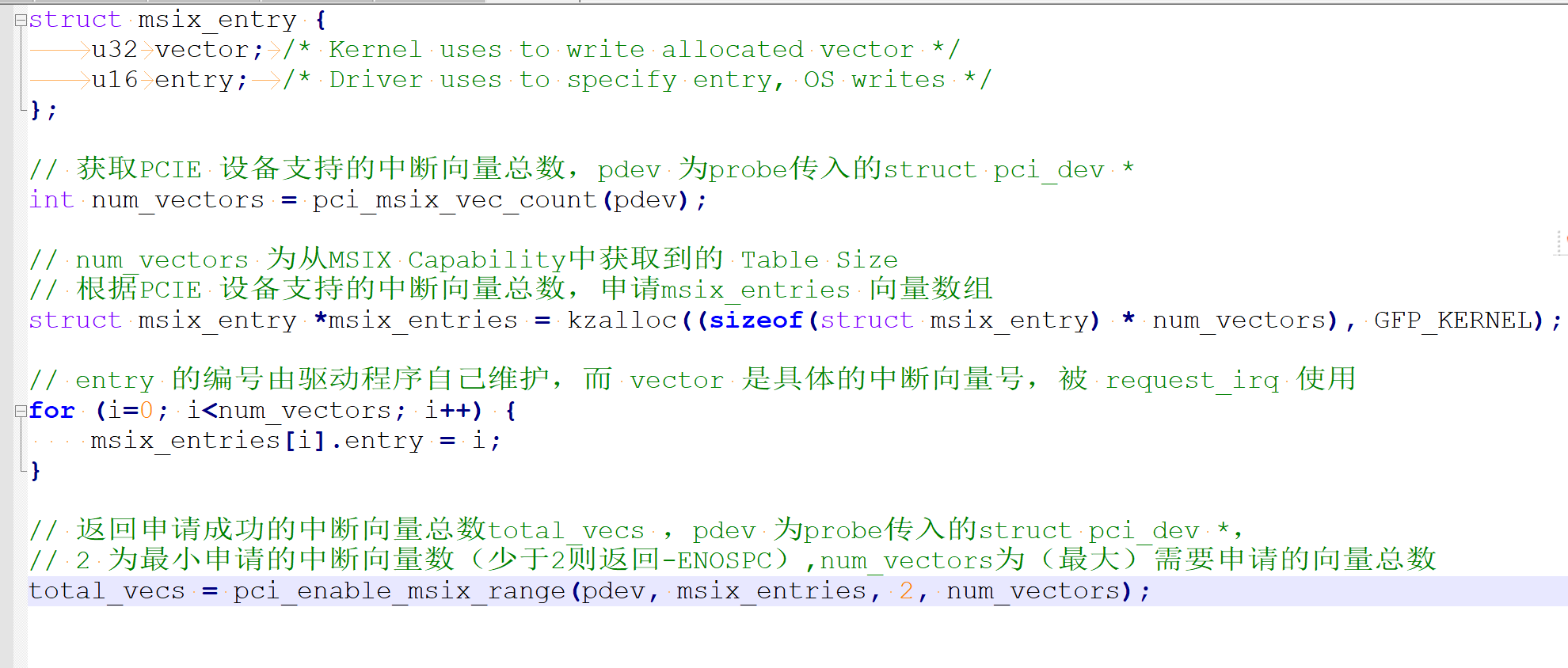
如果你熟悉PCIE 驱动，内核会调用pci\_driver 注册的 probe 函数，而probe 函数会负责中断向量的申请：

如果你熟悉PCIE 驱动，内核会调用pci\_driver 注册的 probe 函数，而probe 函数会负责中断向量的申请：

第一步：调用 pci\_msix\_vec\_count 获取当前PCIE 设备支持的中断向量总数

第二步：调用pci\_enable\_msix\_range 分配 MSIX Table 中每一个中断向量 Entry，获得软件可以使用的所有中断向量号，即msix\_entry 的 vector 成员。

下面代码详细解释了pci\_enable\_msix\_range 如何使用：



下面代码深入分析pci\_enable\_msix\_range 在内核中的实现，你会发现MSIX Table Entry如何被写入：

probe

|--- pci\_msix\_vec\_count

|--- pci\_enable\_msix\_range (\_\_pci\_enable\_msix\_range)

|---\_\_pci\_enable\_msix

|--- msix\_capability\_init

|--- pci\_msi\_setup\_msi\_irqs

|--- arch\_setup\_msi\_irqs

// 调用硬件（如X86）相关的接口获得IRQ Domain信息，Domain负责将硬件中断ID映射到软件的IRQ Number（vector）

|--- native\_setup\_msi\_irqs

|--- [ msi\_domain\_alloc\_irqs ]

[ msi\_domain\_alloc\_irqs ]

|--- irq\_domain\_activate\_irq ( \_\_irq\_domain\_activate\_irq )

|--- msi\_domain\_activate ( domain->ops->activate )

|--- irq\_chip\_write\_msi\_msg

|--- pci\_msi\_domain\_write\_msg (data->chip->irq\_write\_msi\_msg)

|--- " \_\_pci\_write\_msi\_msg "

// 在函数 [ msi\_domain\_alloc\_irqs ] 中循环每一个中断号，最终调用 \_\_pci\_write\_msi\_msg

int msi\_domain\_alloc\_irqs(struct irq\_domain \*domain, struct device \*dev,

int nvec)

{

int i, ret, virq;

for\_each\_msi\_entry(desc, dev) {

virq = desc->irq;// 中断号

irq\_data = irq\_domain\_get\_irq\_data(domain, desc->irq);

ret = irq\_domain\_activate\_irq(irq\_data, can\_reserve);

}

}

// " \_\_pci\_write\_msi\_msg " 负责将每一个中断向量 Entry 写入 MSIX Table

void \_\_pci\_write\_msi\_msg(struct msi\_desc \*entry, struct msi\_msg \*msg)

{

struct pci\_dev \*dev = msi\_desc\_to\_pci\_dev(entry);

if (dev->current\_state != PCI\_D0 || pci\_dev\_is\_disconnected(dev)) {

/\* Don't touch the hardware now \*/

} else if (entry->msi\_attrib.is\_msix) {

void \_\_iomem \*base = pci\_msix\_desc\_addr(entry);

// 将message address 和 message data 写入 MSIX table

writel(msg->address\_lo, base + PCI\_MSIX\_ENTRY\_LOWER\_ADDR);

writel(msg->address\_hi, base + PCI\_MSIX\_ENTRY\_UPPER\_ADDR);

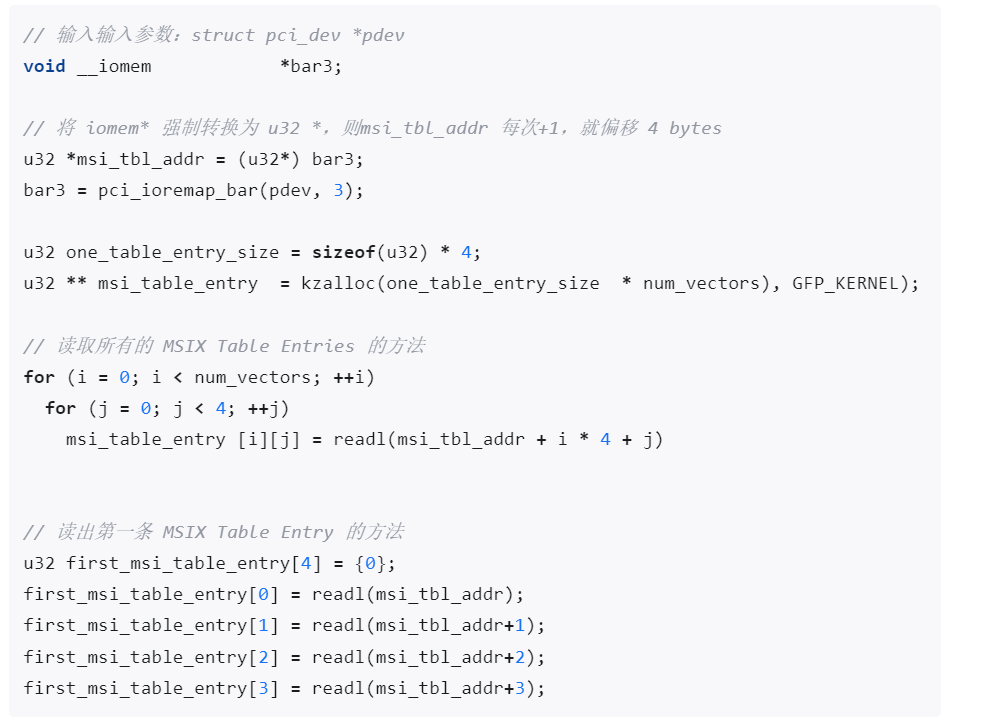
writel(msg->data, base + PCI\_MSIX\_ENTRY\_DATA);

}

}

### 3.3 Table访问

MSIX Table Entry 的访问，需要借助于pci\_ioremap\_bar（ioremap), 将PCIE 设备Bar空间对应的设备内存（即PCIE终端设备上的Register空间）映射到主机的\_\_iomem 类型虚拟地址，才可以被驱动程序访问。下面代码给出了MSIX Table的读取方法。



下图展示了ioremap 用到的相关地址信息以及 map 过程：

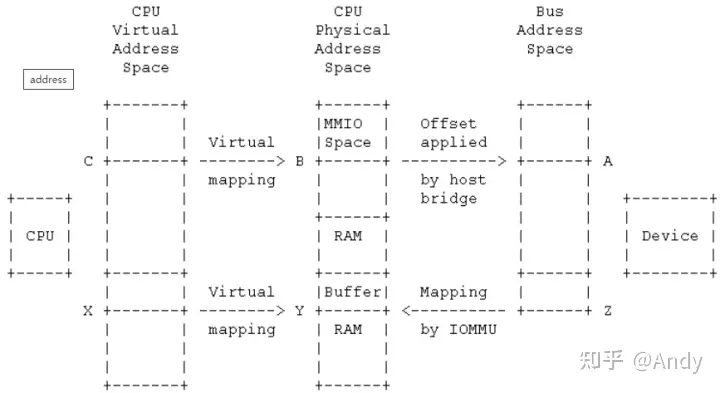
A 地址为 PCI Bus Address，表示在PCI总线上的地址，CPU并不能通过总线地址A（位于BAR范围内，例如上面网卡设备 bar3 的起始总线地址是0x9a018000 ）直接访问总线上的PCI设备。

B 地址是 物理地址，位于物理内存的MMIO Space，可以通过 " cat /proc/iomem | grep -i pci " 查找到所有PCIE 设备的 物理地址空间范围

C 地址为虚拟地址，可以直接被CPU使用，即被驱动程序访问。

PCI Host Bridge 负责进行 物理地址B 和 总线地址A 之间的转换

ioremap 负责将 物理地址B 映射成 虚拟地址 C，这样CPU就可以通过 readl，writel 来访问总线地址A位于的总线地址空间了



4. Capability、Bar、MSI-X Table 关系图



左侧展示了MSI-X CAP 对应的数据结构信息，其存储在PCIE终端设备的配置空间

右侧展示了 MSI-X table，PBA 对应的具体数据结构信息，其存储于PCIE设备的片上内存（寄存器）

中间部分展示了 MSIX Table 和 PBA 对应的PCI总线地址起始位置，以及总共的IO Memory大小（16K）