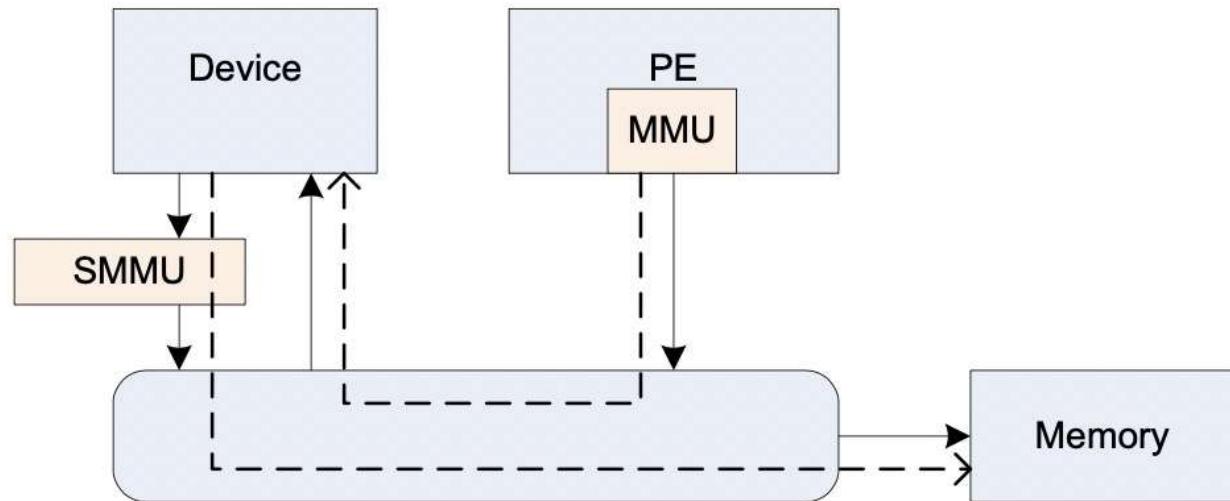


ARM SMMU的原理与IOMMU

2020-10-10 阅读 561

1: arm smmu的原理

1.1: smmu 基本知识



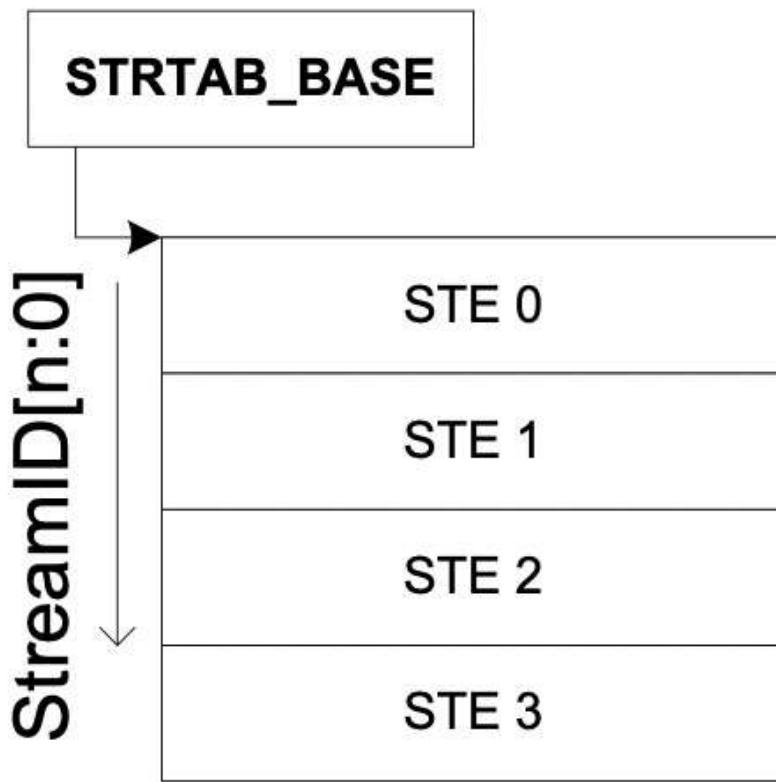
如上图所示，smmu 的作用和mmu 类似，mmu作用是替cpu翻译页表将进程的虚拟地址转换成cpu可以识别的物理地址。同理，smmu的作用就是替设备将dma请求的地址，翻译成设备真正能用的物理地址，但是当smmu bypass的时候，设备也可以直接使用物理地址来进行dma；

1.2: smmu 的数据结构

smmu的重要的用来dma地址翻译的数据结构都是放在内存中的，由smmu的寄存器保存着这些表在内存中的地址，首先就是StreamTable(STE)，这ste 表既包含stage1的翻译表结构也包含stage2的翻译结构，所谓stage1负责VA 到 PA的转换，stage2负责IPA到PA的转换。

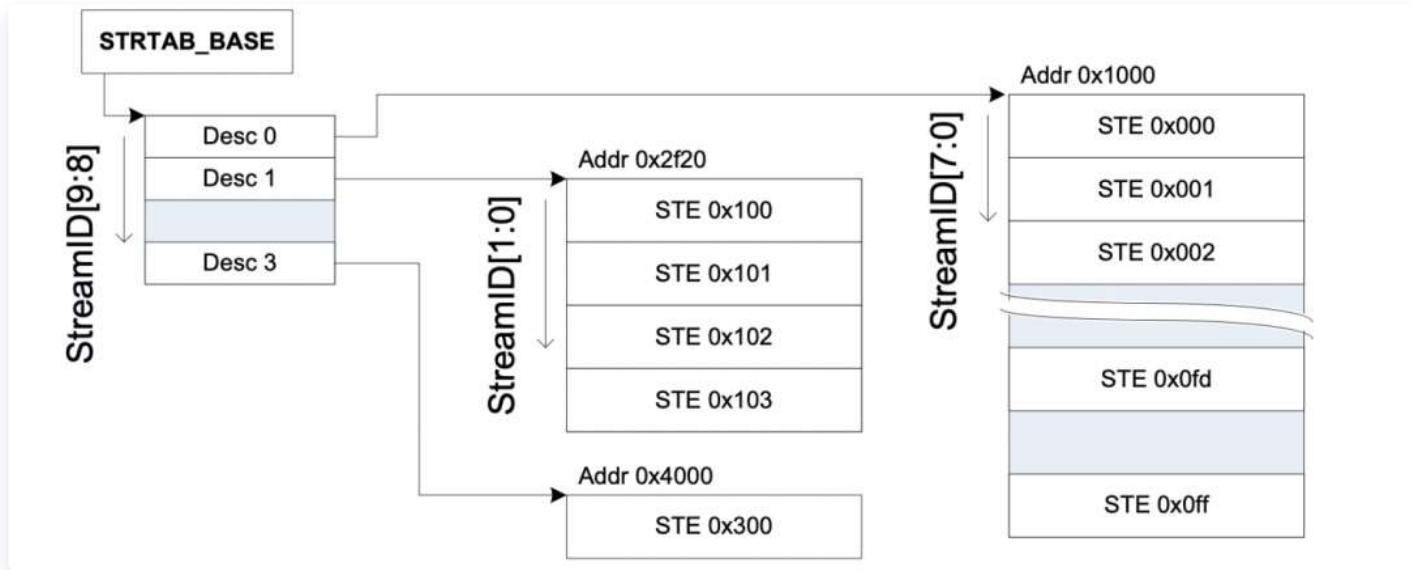
接下来我们重点看一下这个STE的结构，到底在内存中是如何组织的；

对smmu来说，一个smmu可以给很多个设备服务，所以，在smmu里面为了区分的对每个设备进行管理，smmu给每一个设备一个ste entry，那设备如何定位这个ste entry呢？对于一个smmu来说，我们给他所管理的每个设备一个唯一的device id，这个device id又叫 stream id；对于设备比较少的情况下，我们的smmu 的ste 表，很明显只需要是1维数组就可以了，如下图：



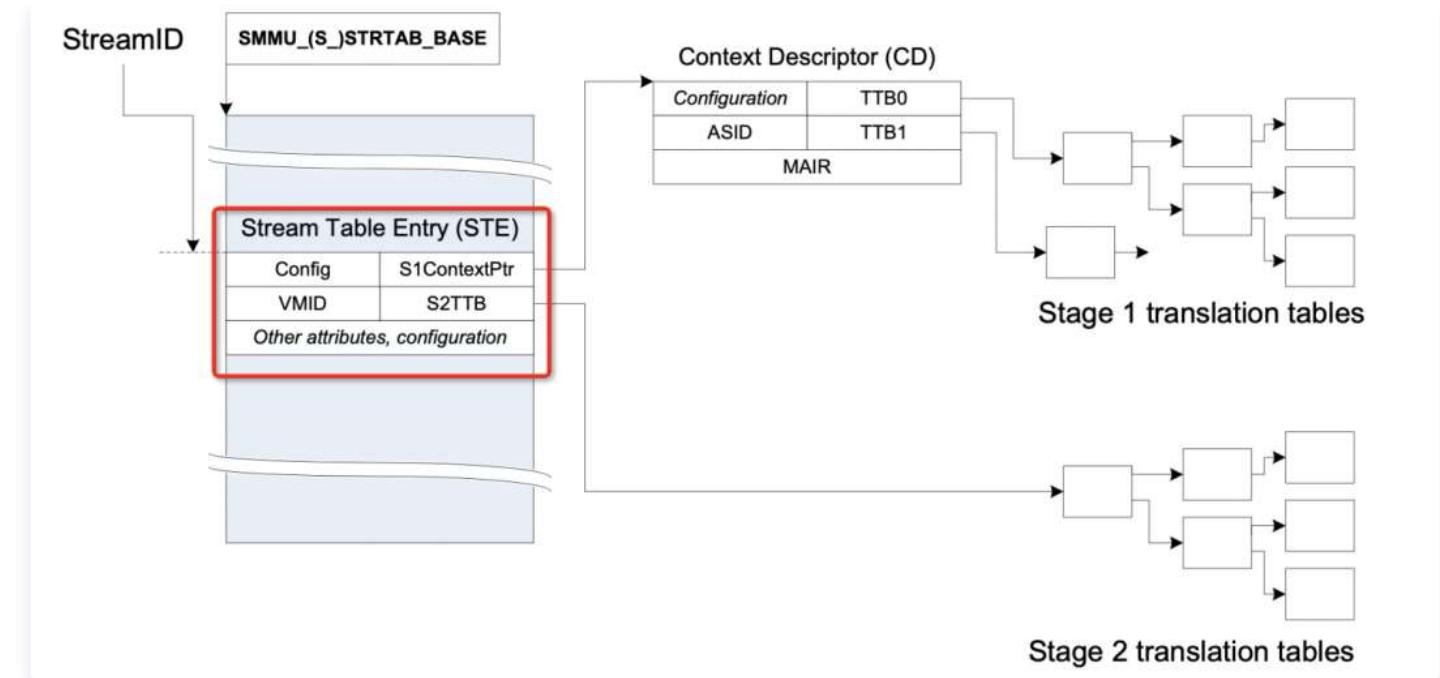
注意，这里ste采用线性表并不是真是由设备的数量来决定的，而是写在smmu 的ID0寄存器中的，也就是配置好了的，对于华为鲲鹏上的smmu基本不采用这种结构；

对于设备数量较多的情况下，我们为了 smmu 更加的皮实点，可以采用两层ste表的结构，如下图：



这里的结构其实很类似我们的mmu的页表了，在arm smmu v3 我们第一层的目录desc的目录结够，大小采用8 (STRTAB_SPLIT) 位，也就是stream id的高8位，stream id剩下的低位全部用来寻址第二层真正的ste entry；

介绍完了 smmu 中管理设备的ste的表的两种结构后，我们来看看这个ste表的具体结构是啥，里面有啥奥秘呢：



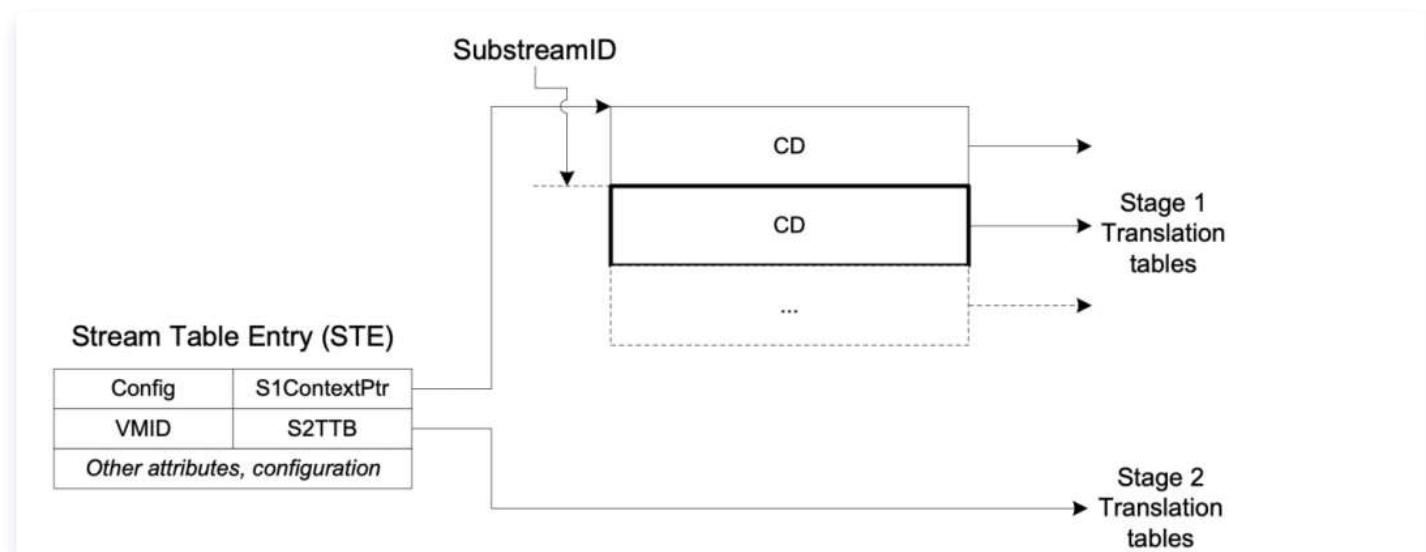
如上所示，红框中就是smmu中一个ste entry的全貌了，从红框中能看出来，这个ste entry同时管理了stage1 和 stage2的数据结构；其中config是表示ste有关的配置项，这个不需要理解也不需要记忆，不知道的查一下smmuv3 的手册即可，里面的VMID是指虚拟机ID,这里我们重点关注一下S1ContextPtr和S2TTB。

首先我们来说S1ContextPtr：

这个S1ContextPtr指向的一个Context Descriptor的目录结构，这张图为了好理解只画了一个，在我们arm中，如果没有虚拟机参与的话，无论是cpu还是smmu地址翻译都是从va->pa/iova->pa，我们称之为stage1，也就是不涉及虚拟，只是一阶段翻译而已。

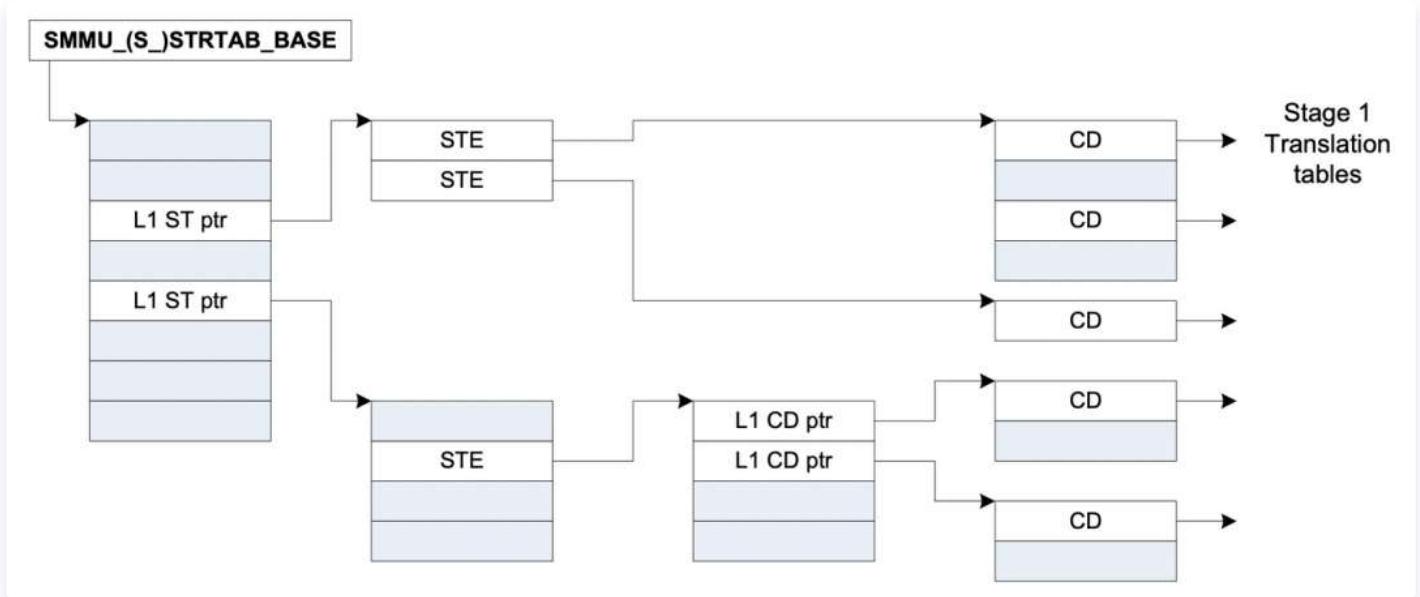
重要的CD表，读到这里，你是不是会问一个问题，在smmu中我们为何要使用CD表呢？原因是这样的，一个smmu可以管理很多设备，所以用ste表来区分每个设备的数据结构，每个设备一个ste表。那如果每个设备上跑了多个任务，这些任务又同时使用了不同的page table 的话，那咋管理呢？对不对？所以smmu 采用了CD表来管理每个page table；

看一看cd 表的查找规则：



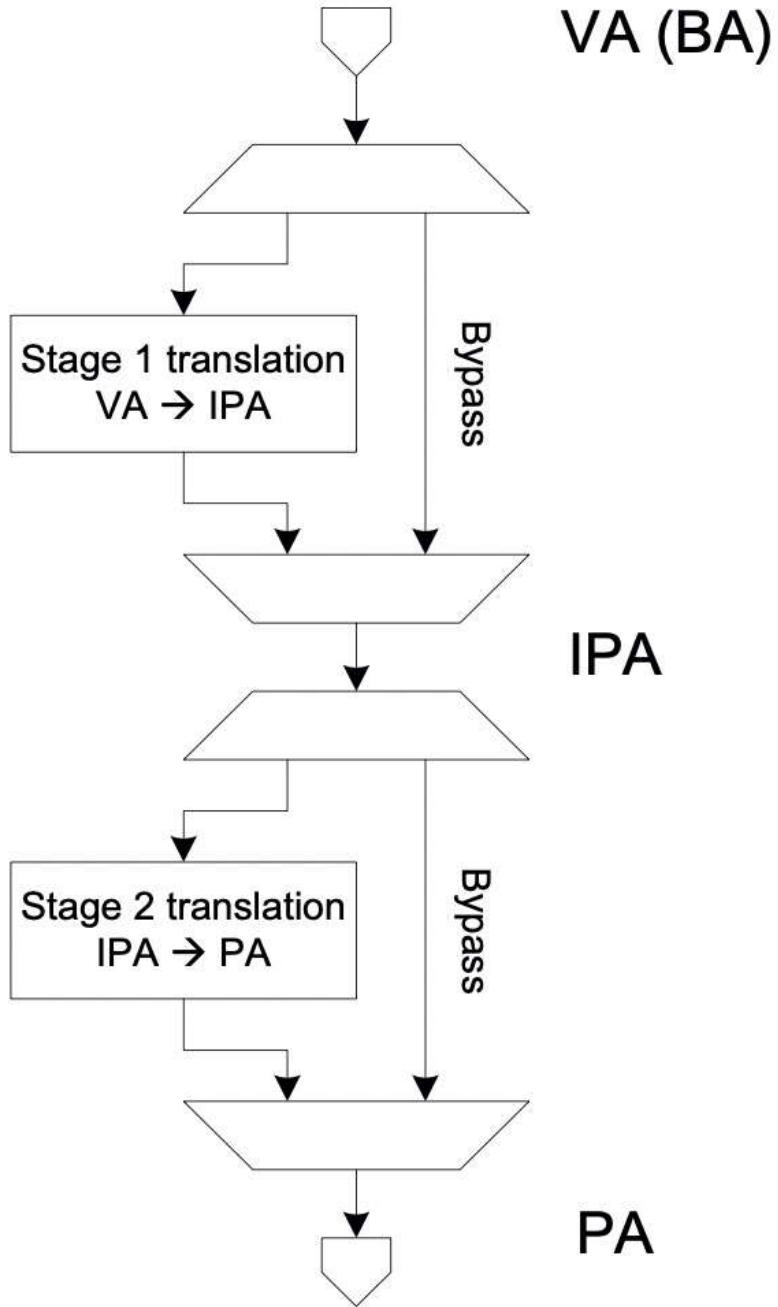
先说另外一个重要概念：SubstreamID(pasid)，这个叫substreamid又称之为pasid，也是非常简单的概念，既然有表了，那也得有id来协助查找啊，所以就出来了这个id，从这里也可以看出来，道理都一样，用了表了就有id

啊！



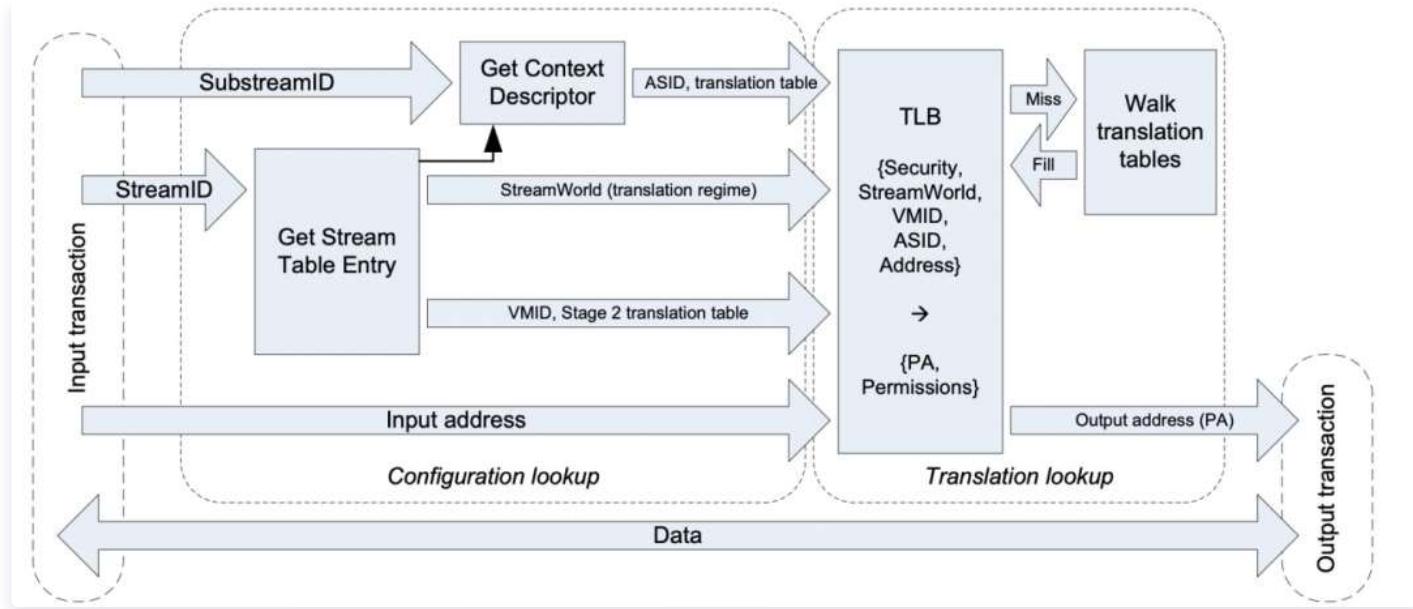
CD表，在smmu中也是可以是线性的或者两级的，这个都是在smmu 寄存器中配置好了的，由smmu驱动来读去，进行按对应的位进行分级，和ste表一样的原理；

介绍了两个基本的也重要的数据结构后我，smmu是在支持虚拟化的时候，可以同时进行stage1 和 stage2的翻译的，如下图所示：



当我们在虚拟机的guest中启用smmu的时候，smmu是需要同时开启stage1 和 stage2的，当然了，smmu 也是可以进行bypass的；

1.3:smmu的地址翻译流程



如上图，基本可以很明显的概括出了一个外设请求 smmu 的地址翻译的基本流程，当一个外设需要dma的物理地址的时候，开始请求smmu的地址翻译，这时候外设给 smmu 3个比较重要的信息，分别是：streamid：协助smmu找到管理外设的ste entry，substreamid：当找到ste entry后，协助smmu找到对应的cd 表，通过这两个id smmu 就可以找到对应的iopge table了，smmu找到page table 后结合外设提交过来的最后一个信息iova，即可开始进行地址翻译；

smmu 也有tlb的缓存，smmu首先会根据当前cd表中存放的asid来查查tlb缓存中有没有对应page table的缓存，这里其实和mmu找页表的原理是一样的，不过多解释了，很简单；

上图中的地址翻译还涉及到了stage2，这里不解释了，smmu涉及到虚拟化的过程比较复杂，这个有机会再解释；

2 smmu驱动与iommu框架

2.1：smmu v3驱动初始化

简单的介绍了上面的两个重要表以及smmu内部的基本的查找流程后，我们现在来看看在linux内核中，smmu驱动是如何完成初始化的过程，借着这个分析，我们看看smmu里的重要的几种队列：

smmuv3的在内核中的代码路径：drivers/iommu/arm-smmu-v3.c:

```

static int arm_smmu_device_probe(struct platform_device *pdev)
{
>-----int irq, ret;
>-----struct resource *res;
>-----resource_size_t ioaddr;
>-----struct arm_smmu_device *smmu;
>-----struct device *dev = &pdev->dev;
>-----bool bypass;

>-----smmu = devm_kzalloc(dev, sizeof(*smmu), GFP_KERNEL);
>-----if (!smmu) {
>----->----dev_err(dev, "failed to allocate arm_smmu_device\n");
>----->----return -ENOMEM;
>-----}
>-----smmu->dev = dev;

>-----if (dev->of_node) {
>----->----ret = arm_smmu_device_dt_probe(pdev, smmu);
>-----} else {
>----->----ret = arm_smmu_device_acpi_probe(pdev, smmu);
>----->----if (ret == -ENODEV)
>----->---->----return ret;
>-----}

>-----/* Set bypass mode according to firmware probing result */
>-----bypass = !!ret;

>-----/* Base address */
>-----res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
>-----if (resource_size(res) + 1 < arm_smmu_resource_size(smmu)) {
>----->----dev_err(dev, "MMIO region too small (%pr)\n", res);
>----->----return -EINVAL;
>-----}
>-----ioaddr = res->start;

```

上面是smmu驱动中初始化流程的前半部分，从中可以很容易看出来，内核中每个smmu都有一个结构体struct arm_smmu_device来管理，实际上初始化的流程就是在填充着个结构。看上图，首先就是从slub/slub中分配一个对象空间，随后一个比较重要的是函数

arm_smmu_device_dt_probe 和 arm_smmu_device_acpi_probe，这两函数会从dts中的smmu节点和acpi的smmu配置表中读取一些smmu中断等等属性；

随后调用函数platform_get_resource来从dts或者acpi表中读取smmu的寄存器的基地址，这个很重要，后续所有的初始化都是围绕着个配置来的；

```

-----irq = platform_get_irq_byname(pdev, "combined");
-----if (irq > 0)
----->-----smmu->combined_irq = irq;
-----else {
----->-----irq = platform_get_irq_byname(pdev, "eventq");
----->-----if (irq > 0)
----->----->-----smmu->evtq.q.irq = irq;

----->-----irq = platform_get_irq_byname(pdev, "priq");
----->-----if (irq > 0)
----->----->-----smmu->priq.q.irq = irq;

----->-----irq = platform_get_irq_byname(pdev, "gerror");
----->-----if (irq > 0)
----->----->-----smmu->gerr_irq = irq;
-----}
-----/* Probe the h/w */
-----ret = arm_smmu_device_hw_probe(smmu);
-----if (ret)
----->-----return ret;

-----/* Initialise in-memory data structures */
-----ret = arm_smmu_init_structures(smmu);
-----if (ret)
----->-----return ret;

-----/* Record our private device structure */
-----platform_set_drvdata(pdev, smmu);

-----/* Reset the device */
-----ret = arm_smmu_device_reset(smmu, bypass);
-----if (ret)
----->-----return ret;

```

继续看剩下的部分，开头很容易看出来，要读取smmu的几个中断号，smmu 硬件给软件消息有队列buffer，smmu 硬件通过中断的方式让smmu驱动从队列buffer中取消息，我们一一介绍：

第一个eventq中断，smmu的一个队列叫event队列，这个队列是给挂在smmu上的platform设备用的，当platform设备使用smmu翻译dma 的iova的时候，如果发生了一场smmu会首先将异常的消息填到event队列中，随

后上报一个eventq的中断给 smmu 驱动，smmu驱动接到这个中断后，开始执行中断处理程序，从event队列中将异常的消息读出来，显示异常；

另外一个priq中断时给pri队列用的，这个队列是专门给挂在smmu上的pcie类型的设备用的，具体的流程其实和event队列是一样的，这里不多解释了；

最后一个gerror中断，如果smmu 在执行过程中，发生了不可恢复的严重错误，smmu会报告一个gerror中断给smmu驱动，就不需要队列了，因为本身严重错误了，直接中断上来处理了；

完成了3个中断初始化后（具体的中断初始化映射流程，不在这里介绍，改天单独写个中断章节介绍），smmu 驱动此时已经完成了smmu管理结构的分配，以及smmu配置的读取，smmu的寄存器的映射，以及smmu中断的初始化，这些都搞完后，smmu驱动开始读取提前写死在 smmu 寄存器中的各种配置，将配置bit位读取出来放到struct arm_smmu_device的数据结构中，函数arm_smmu_device_hw_probe函数就负责读smmu的硬件寄存器；

当我们寄存器配置读取完毕后，这时候我们知道哪些信息呢？会有这个smmu支持二级ste还是一级的ste，二级的cd还有1级的cd，这个smmu支持的物理页大小，iova和pa的地址位数等等；这些头填在arm_smmu_device的features的字段里面；

基本信息读出来后，我们是不是可开始初始化数据结构了？答案是肯定的啦，看看函数arm_smmu_init_structures；

```
static int arm_smmu_init_structures(struct arm_smmu_device *smmu)
{
    int ret;

    ret = arm_smmu_init_queues(smmu);
    if (ret)
        return ret;

    return arm_smmu_init_stab(smmu);
}
```

从上面的数据结构初始化的函数可以看出来，smmu驱动主要负责初始化两种数据结构，一个stab(stream table的简写)，另外一个种是队列的内存分配和初始化；我们首先来看看队列的：

```

static int arm_smmu_init_queues(struct arm_smmu_device *smmu)
{
>-----int ret;

>-----/* cmdq */
>-----spin_lock_init(&smmu->cmdq.lock);
>-----ret = arm_smmu_init_one_queue(smmu, &smmu->cmdq.q, ARM_SMMU_CMDQ_PROD,
>----->>>-----|      ARM_SMMU_CMDQ_CONS, CMDQ_ENT_DWORDS);
>-----if (ret)
>----->----return ret;

>-----/* evtq */
>-----ret = arm_smmu_init_one_queue(smmu, &smmu->evtq.q, ARM_SMMU_EVTQ_PROD,
>----->>>-----|      ARM_SMMU_EVTQ_CONS, EVTQ_ENT_DWORDS);
>-----if (ret)
>----->----return ret;

>-----/* priq */
>-----if (!(smmu->features & ARM_SMMU_FEAT_PRI))
>----->----return 0;

>-----return arm_smmu_init_one_queue(smmu, &smmu->priq.q, ARM_SMMU_PRIQ_PROD,
>----->>>-----|      ARM_SMMU_PRIQ_CONS, PRIQ_ENT_DWORDS);
}

```

从上面可以看出来，smmu驱动主要初始化3个队列：cmdq, evtq, priq;这里不再进一步解释了，避免陷入函数细节分析；

最后我们来看看smmu 的strtab的初始化：

```
static int arm_smmu_init_strtab(struct arm_smmu_device *smmu)
{
    u64 reg;
    int ret;

    if (smmu->features & ARM_SMMU_FEAT_2_LVL_STRTAB)
        ret = arm_smmu_init_strtab_2lvl(smmu);
    else
        ret = arm_smmu_init_strtab_linear(smmu);

    if (ret)
        return ret;

    /* Set the strtab base address */
    reg = smmu->strtab_cfg.strtab_dma & STRTAB_BASE_ADDR_MASK;
    reg |= STRTAB_BASE_RA;
    smmu->strtab_cfg.strtab_base = reg;

    /* Allocate the first VMID for stage-2 bypass STEs */
    set_bit(0, smmu->vmid_map);
    return 0;
}
```

从上图可以看出来，首先判断我们需要初始化一级的还是二级的stream table，这里依据就是上面的硬件寄存器中读取出来的；

我们首先看看函数arm_smmu_init_strtab_linear 函数：

```

static int arm_smmu_init_strtab_linear(struct arm_smmu_device *smmu)
{
    void *strtab;
    u64 reg;
    u32 size;
    struct arm_smmu_strtab_cfg *cfg = &smmu->strtab_cfg;

    size = (1 << smmu->sid_bits) * (STRTAB_STE_DWORDS << 3);
    strtab = dmam_alloc_coherent(smmu->dev, size, &cfg->strtab_dma,
        GFP_KERNEL | __GFP_ZERO);
    if (!strtab) {
        dev_err(smmu->dev,
            "failed to allocate linear stream table (%u bytes)\n",
            size);
        return -ENOMEM;
    }
    cfg->strtab = strtab;
    cfg->num_l1_ents = 1 << smmu->sid_bits;

    /* Configure strtab_base_cfg for a linear table covering all SIDs */
    reg = FIELD_PREP(STRTAB_BASE_CFG_FMT, STRTAB_BASE_CFG_FMT_LINEAR);
    reg |= FIELD_PREP(STRTAB_BASE_CFG_LOG2SIZE, smmu->sid_bits);
    cfg->strtab_base_cfg = reg;

    arm_smmu_init_bypass_stes(strtab, cfg->num_l1_ents);
    return 0;
}

```

对于线性的stream table表来说smmu 驱动会将调用dma alloc接口将stream table 需要的所有空间都一把分配完毕了，并且将所有的ste entry项都给预先的初始化成bypass的模式，具体的就不深入看了，比较简单，设置bit；

随后我们来看看函数：

arm_smmu_init_strtab_2lvl;

```

static int arm_smmu_init strtab_2lvl(struct arm_smmu_device *smmu)
{
    void *strtab;
    u64 reg;
    u32 size, l1size;
    struct arm_smmu_strtab_cfg *cfg = &smmu->strtab_cfg;

    /* Calculate the L1 size, capped to the SIDSIZE. */
    size = STRTAB_L1_SZ_SHIFT - (ilog2(STRTAB_L1_DESC_DWORDS) + 3);
    size = min(size, smmu->sid_bits - STRTAB_SPLIT);
    cfg->num_l1_ents = 1 << size;

    size += STRTAB_SPLIT;
    if (size < smmu->sid_bits)
        dev_warn(smmu->dev,
                 "2-level strtab only covers %u/%u bits of SID\n",
                 size, smmu->sid_bits);

    l1size = cfg->num_l1_ents * (STRTAB_L1_DESC_DWORDS << 3);
    strtab = dma_alloc_coherent(smmu->dev, l1size, &cfg->strtab_dma,
                               GFP_KERNEL | __GFP_ZERO);
    if (!strtab) {
        dev_err(smmu->dev,
                "failed to allocate l1 stream table (%u bytes)\n",
                size);
        return -ENOMEM;
    }
    cfg->strtab = strtab;

    /* Configure strtab_base_cfg for 2 levels */
    reg = FIELD_PREP(STRTAB_BASE_CFG_FMT, STRTAB_BASE_CFG_FMT_2LVL);
    reg |= FIELD_PREP(STRTAB_BASE_CFG_LOG2SIZE, size);
    reg |= FIELD_PREP(STRTAB_BASE_CFG_SPLIT, STRTAB_SPLIT);
    cfg->strtab_base_cfg = reg;

    return arm_smmu_init_l1_strtab(smmu);
}

```

我们可以思考一个问题：我们真的需要将所有的ste entry都创造出来吗？很显然，不是的，smmu驱动的初始化正是基于这种原理，仅仅只会初始化第一级的ste目录项，其实这里就是类似页表的初始化了也只是先初始化了目录项；函数中dma alloc coherent就是负责分配第一级的目录项的，分配的大小是多大呢？我们可以看一下有一个关键的宏STRTAB_SPLIT，这个宏目前在smmu驱动中是8位，也就是预先会分配 2^8 个目录项，每个目录项的大小是固定的；

我们可以看到里面还调用了一个函数arm_smmu_init_l1_strtab函数，这里就是我们空间分配完了，总该给这些目录项给初始化一下吧，这里就不深入进去看了；

到此为止，我们已经将基本的数据结构初始化给简要的讲完了；我们接着看smmu驱动初始化的剩下的，见下图：

```
-----/* Reset the device */
-----ret = arm_smmu_device_reset(smmu, bypass);
-----if (ret)
----->-----return ret;

-----/* And we're up. Go go go! */
-----ret = iommu_device_sysfs_add(&smmu->iommu, dev, NULL,
----->----->-----|      "smmu3.%pa", &ioaddr);
-----if (ret)
----->-----return ret;

-----iommu_device_set_ops(&smmu->iommu, &arm_smmu_ops);
-----iommu_device_set_fwnode(&smmu->iommu, dev->fwnode);

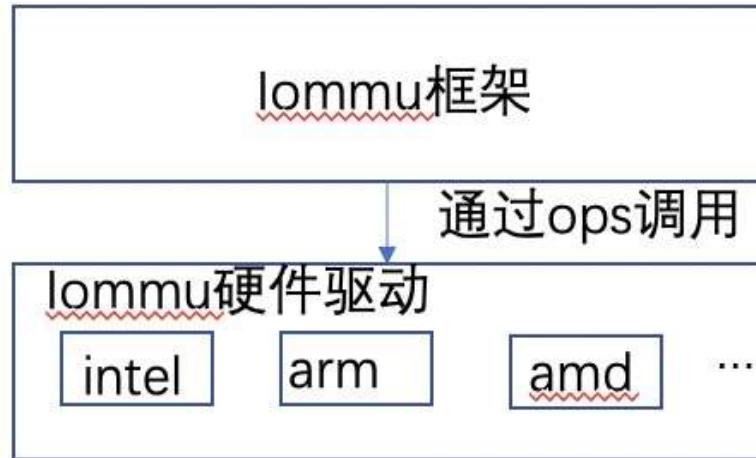
-----ret = iommu_device_register(&smmu->iommu);
-----if (ret) {
----->-----dev_err(dev, "Failed to register iommu\n");
----->-----return ret;
-----}
```

上图是smmu 驱动初始化的剩下的部分，我们可以看出来里面第一个函数是arm_smmu_device_reset，这个函数是干嘛的呢，我们前面是不是已经给这个smmu在内存中分配了几个队列和stream table的目录项？那这些数据结构的基址总该让smmu知道吧？这个函数就是将这些基地址给放到smmu的控制寄存器中的；当前我们需要的东西给初始化完后，smmu驱动接下来就是将smmu的基本数据结构注册到上层的iommu抽象框架里，让iommu结构能够调用到smmu，这个在后面再说。

2.2 smmu 与 iommu关系

2.2.1 两者的结构关系

smmu 和 iommu 是何种关系呢？在我们的硬件体系中，能够有能力完成设备iova 到 pa转换的有很多，例如有intel iommu, amd的iommu ,arm的smmu等等，不一一枚举了；那这些不同的硬件架构不会都作为一个独立的子系统，所以，在linux 内核中 抽象了一层 iommu 层，由iommu层给各个外部设备驱动提供结构，隐藏底层的不同的架构；如图所示：



由上图可以很明显的看出来，各个架构的smmu驱动是如何使如何和iommu框架对接的，iommu框架通过不同架构的ops来调用到底层真正的驱动接口；

我们可以问自己一个问题：底层的驱动是如何对接到上层的？

接下来我们来看看进入内核代码来帮我们解开疑惑；

```

>-----ret = iommu_device_sysfs_add(&smmu->iommu, dev, NULL,
>----->----->----->-----|      "smmu3.%pa", &ioaddr);
>-----if (ret)
>----->-----return ret;

>-----iommu_device_set_ops(&smmu->iommu, &arm_smmu_ops);
>-----iommu_device_set_fwnode(&smmu->iommu, dev->fwnode);

>-----ret = iommu_device_register(&smmu->iommu);
>-----if (ret) {
>----->-----dev_err(dev, "Failed to register iommu\n");
>----->-----return ret;
>-----}

#ifndef CONFIG_PCI
>-----if (pci_bus_type.iommu_ops != &arm_smmu_ops) {
>----->-----pci_request_acs();
>----->-----ret = bus_set_iommu(&pci_bus_type, &arm_smmu_ops);
>----->-----if (ret)
>----->----->-----return ret;
>-----}
#endif
#ifndef CONFIG_ARM_AMBA
>-----if (amba_bustype.iommu_ops != &arm_smmu_ops) {
>----->-----ret = bus_set_iommu(&amba_bustype, &arm_smmu_ops);
>----->-----if (ret)
>----->----->-----return ret;
>-----}
#endif
>-----if (platform_bus_type.iommu_ops != &arm_smmu_ops) {
>----->-----ret = bus_set_iommu(&platform_bus_type, &arm_smmu_ops);
>----->-----if (ret)
>----->----->-----return ret;
>-----}
>-----return 0;

```

如上图是smmu 驱动初始化的最后一部分，对于底层的每一个smmu结构在iommu框架层中都一有一个唯一的一个结构体表示： struct iommu_device， 上图中函数iommu_device_register所完成的任务就是将我们所初始化好的 iommu结构体给注册到iommu层的链表中，统一管理起来；最后我们根据smmu所挂载的是pcie外设，还是 platform外设，将和个smmu绑定到不同的总线类型上；

2.2.2 iommu的重要结构与ops

iommu 层通过ops来调用底层硬件驱动，我们来看看smmu v3硬件驱动提供了哪些ops call：

```

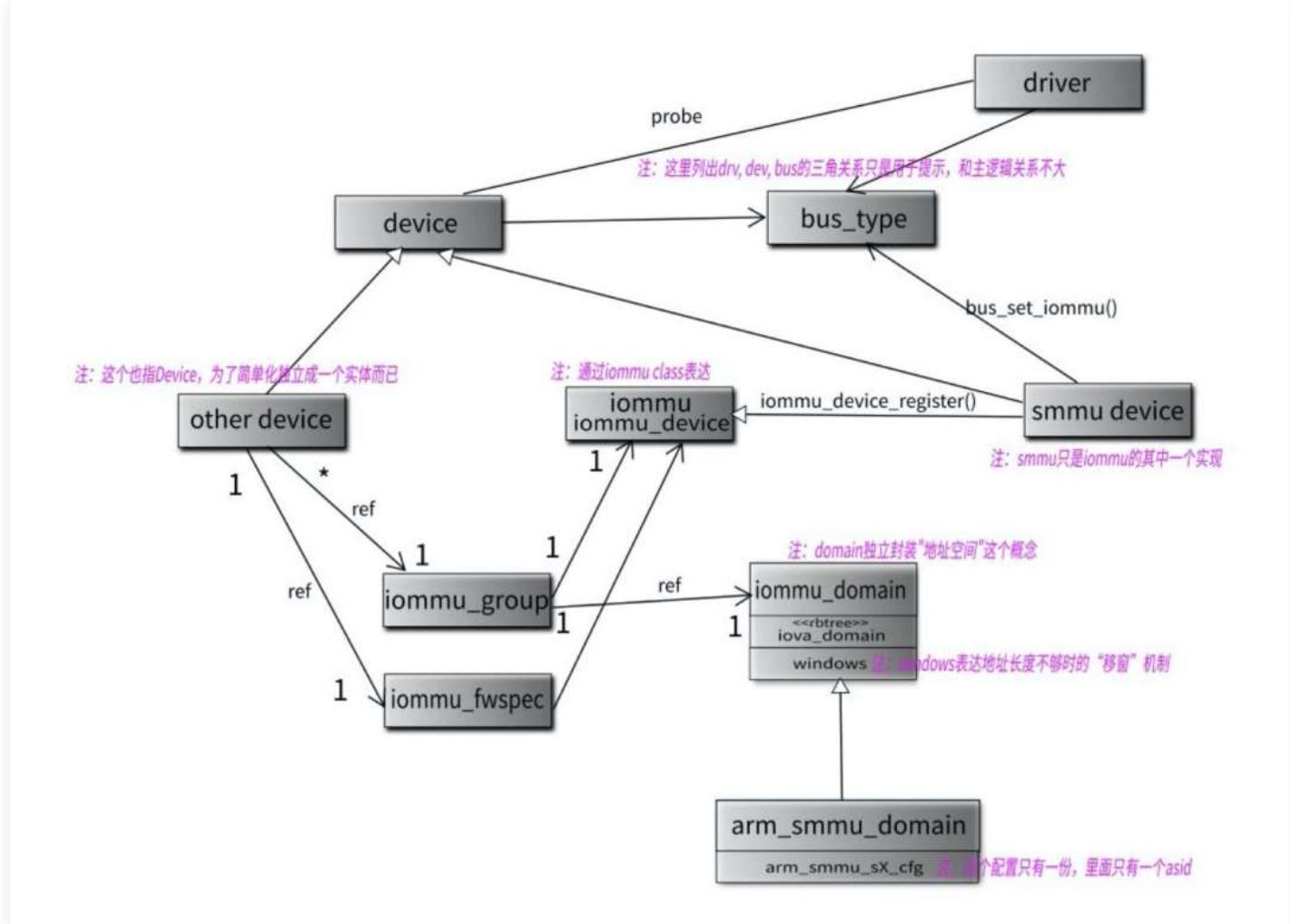
static struct iommu_ops arm_smmu_ops = {
>----- .capable>-----= arm_smmu_capable,
>----- .domain_alloc>----= arm_smmu_domain_alloc,
>----- .domain_free>----= arm_smmu_domain_free,
>----- .attach_dev>----= arm_smmu_attach_dev,
>----- .map>---->----= arm_smmu_map,
>----- .unmap>---->----= arm_smmu_unmap,
>----- .flush_iotlb_all>----= arm_smmu_flush_iotlb_all,
>----- .iotlb_sync>----= arm_smmu_iotlb_sync,
>----- .iova_to_phys>----= arm_smmu_iova_to_phys,
>----- .add_device>----= arm_smmu_add_device,
>----- .remove_device>----= arm_smmu_remove_device,
>----- .device_group>----= arm_smmu_device_group,
>----- .domain_get_attr>----= arm_smmu_domain_get_attr,
>----- .domain_set_attr>----= arm_smmu_domain_set_attr,
>----- .of_xlate>---->----= arm_smmu_of_xlate,
>----- .get_resv_regions>----= arm_smmu_get_resv_regions,
>----- .put_resv_regions>----= arm_smmu_put_resv_regions,
>----- .pgsize_bitmap>----= -1UL, /* Restricted during device attach */
};

```

上图就是smmu v3 硬件驱动提供的所有的调用函数；

既然到了iommu层，那我们也会涉及到两种概念的管理，一种是设备如何管理，另外一种是smmu 提供的io page table如何管理；

为了分别管理，这两种概念，iommu 框架提供了两种结构体，一个是 struct iommu_domain 这个结构抽象出了一个domain的结构，用来代表底层的arm_smmu_domain，其实最核心的是管理这个domian所拥有的io page table。另外一个是struct iommu_group这个结构是用来管理设备的，多个设备可以在一个iommu group中，以此来共享一个iopage table；我们看一个网络上的图即可很明白的表明其中的关系：



这张图中很明显的写出来smmu domian和 iommu的domain的关系，以及iommu group的作用；不再过多解释。

2.3 dma iova 与iommu

dma 和 iommu 息息相关，iommu的产生其实很大的原因就是避免dma的时候直接使用物理地址而导致的不安全性，所以就产生了iova，我们在调用dma alloc的时候，首先在io 的地址空间中分配你一个iova，然后在iommu所管理的页表中做好iova 和dma alloc时候产生的物理地址进行映射；外设在进行dma的时候，只需要使用iova即可完成dma动作；

那我们如何完成dma alloc的时候iova到pa的映射的呢？

dma_alloc -> __iommu_alloc_attrs

```

599 >-----> pgprot_t prot = __get_dma_pgprot(attrs, PAGE_KERNEL, coherent);
600 >-----> struct page **pages;
601
602 >-----> pages = iommu_dma_alloc(dev, iosize, gfp, attrs, ioprot,
603 >-----> >-----> handle, flush_page);
604 >-----> if (!pages)
605 >-----> >-----> return NULL;
606
607 >-----> addr = dma_common_pages_remap(pages, size, VM_USERMAP, prot,
608 >-----> >-----> >-----> _builtin_return_address(0));
609 >-----> if (!addr)
610 >-----> >-----> iommu_dma_free(dev, pages, iosize, handle);
611 >-----}
612 >-----return addr;
613 }
  
```

在__iommu_alloc_attrs函数中调用iommu_dma_alloc函数来完成iova和pa的分配与映射；

iommu_dma_alloc->__iommu_dma_alloc_pages，

首先会调用者个函数来完成物理页面的分配：

```
570 >-----count = PAGE_ALIGN(size) >> PAGE_SHIFT;
571 >-----pages = __iommu_dma_alloc_pages(dev, count, alloc_sizes >> PAGE_SHIFT,
572 >----->>>>>>-----gfp);
573 >-----if (!pages)
574 >----->-----return NULL;
575
576 >-----size = iova_align(iovad, size);
577 >-----iova = iommu_dma_alloc_iova(domain, size, dev->coherent_dma_mask, dev);
578 >-----if (!iova)
579 >----->-----goto out_free_pages;
580
581 >-----if (sg_alloc_table_from_pages(&sgt, pages, count, 0, size, GFP_KERNEL))
582 >----->-----goto out_free_iova;
583
584 >-----if (!(prot & IOMMU_CACHE)) {
585 >----->-----struct sg_mapping_iter miter;
586 >----->-----/*
587 >----->-----/* The CPU-centric flushing implied by SG_MITER_TO_SG isn't
588 >----->-----/* sufficient here, so skip it by using the "wrong" direction.
589 >----->-----*/
590 >----->-----sg_miter_start(&miter, sgt.sgl, sgt.orig_nents, SG_MITER_FROM_SG);
591 >----->-----while (sg_miter_next(&miter))
592 >----->-----flush_page(dev, miter.addr, page_to_phys(miter.page));
593 >----->-----sg_miter_stop(&miter);
594 >-----}
595
596 >-----if (iommu_map_sg(domain, iova, sgt.sgl, sgt.orig_nents, prot)
597 >----->-----< size)
598 >----->-----goto out_free_sg;
599
600 >-----*handle = iova;
601 >-----sg_free_table(&sgt);
602 >-----return pages;
603
604 out_free_sg:
605 >-----sg_free_table(&sgt);
606 out_free_iova:
607 >-----iommu_dma_free_iova(cookie, iova, size);
608 >-----
```

函数__iommu_dma_alloc_pages中完成的任务是页面分配，iommu_dma_alloc_iova完成的就是iova的分配，最后iommu_map_sg即可完成iova到pa的映射；

linux采用rb tree来管理每一段的iova区间，这其实和我们的虚拟内存的分配是类似的，我们的vma的管理也是这样的；

我们接下来来看看iova的释放过程，这个释放的过程，我们是可以看到看到strict 个 non-strict模式的最核心的区别：

```

static void __iommu_dma_unmap(struct iommu_domain *domain, dma_addr_t dma_addr,
>----->-----size_t size)
{
>-----struct iommu_dma_cookie *cookie = domain->iova_cookie;
>-----struct iova_domain *iovad = &cookie->iovad;
>-----size_t iova_off = iova_offset(iovad, dma_addr);

>-----dma_addr -= iova_off;
>-----size = iova_align(iovad, size + iova_off);

>-----WARN_ON(iommu_unmap_fast(domain, dma_addr, size) != size);
>-----if (!cookie->fq_domain)
>----->-----iommu_tlb_sync(domain);
>-----iommu_dma_free_iova(cookie, dma_addr, size);
}

```

老规矩，直接撸代码，我们看到dma的释放流程也是很简单的，首先将iova和pa进行解映射处理，然后将iova结构给释放掉；

看图中解映射的部分就是在iommu_unmap_fast流程中处理的就是调用iommu的unmap然后通过ops 调用到arm smmu v3驱动的 unmap函数： __iommu_dma_unmap->iommu_unmap_fast->(ops->unmap: arm_smmu_unmap)->arm_lpae_unmap；

我们进入函数arm_lpae_unmap中看看是干啥的，见下图：

```

603 >-----if (size == ARM_LPAE_BLOCK_SIZE(lvl, data)) {
604 >----->-----__arm_lpae_set_pte(ptep, 0, &iop->cfg);
605
606 >----->-----if (!iopte_leaf(pte, lvl)) {
607 >----->-----/* Also flush any partial walks */
608 >----->-----io_pgtable_tlb_add_flush(iop, iova, size,
609 >----->----->----->-----ARM_LPAE_GRANULE(data), false);
610 >----->-----io_pgtable_tlb_sync(iop);
611 >----->-----ptep = iopte_deref(pte, data);
612 >----->-----__arm_lpae_free_pgtable(data, lvl + 1, ptep);
613 >----->-----} else if (iop->cfg.quirks & IO_PGTABLE_QUIRK_NON_STRICT) {
614 >----->-----/*
615 >----->----- * Order the PTE update against queueing the IOVA, to
616 >----->----- * guarantee that a flush callback from a different CPU
617 >----->----- * has observed it before the TLBIALL can be issued.
618 >----->-----*/
619 >----->-----smp_wmb();
620 >----->-----} else {
621 >----->-----io_pgtable_tlb_add_flush(iop, iova, size, size, true);
622 >----->-----}
623
624 >----->-----return size;
625 >-----} else if (iopte_leaf(pte, lvl)) {
626 >----->-----/*
627 >----->----- * Insert a table at the next level to map the old region,
628 >----->----- * minus the part we want to unmap
629 >----->-----*/
630 >----->-----return arm_lpae_split_blk_unmap(data, iova, size, pte,
631 >----->----->----->----->-----lvl + 1, ptep);
632 >-----}
633
634 >-----/* Keep on walkin' */
635 >-----ptep = iopte_deref(pte, data);
636 >-----return __arm_lpae_unmap(data, iova, size, lvl + 1, ptep);

```

这个函数采用递归的方式来查找io page table的最后一项，当找到的时候，我们可注意看代码行613~622行，其中613~620行是当我们的iommu采用默认的non strict模式的时候，我们是不用立马对tlb进行无效化的；但是当我们采用strict模式的时候，我们还是会将tlb给刷新一下，调用函数io_pgtable_tlb_add_flush给smmu写入一个tlb无效化的指令；

那我们采用non-strict模式的时候是如何刷新tlb的呢？秘密就在函数iommu_dma_free_iova函数中见下图：

```

413 static void iommu_dma_free_iova(struct iommu_dma_cookie *cookie,
414 >----->-----dma_addr_t iova, size_t size)
415 {
416 >-----struct iova_domain *iovad = &cookie->iovad;
417
418 >-----/* The MSI case is only ever cleaning up its most recent allocation */
419 >-----if (cookie->type == IOMMU_DMA_MSI_COOKIE)
420 >----->-----cookie->msi_iova -= size;
421 >-----else if (cookie->fq_domain)>----/* non-strict mode */
422 >----->-----queue_iova(iovad, iova_pfn(iovad, iova),
423 >----->----->-----size >> iova_shift(iovad), 0);
424 >-----else
425 >----->-----free_iova_fast(iovad, iova_pfn(iovad, iova),
426 >----->----->-----size >> iova_shift(iovad));
427 }

```

我们可以看到，如果采用non-strict的模式的时候，我们是放到一个队列中的，当我们的队列满的时候，会调用函数iovad->flush_cb，

这个函数指针，最终会调用到函数：iommu_dma_flush_iotlb_all，来进行全局的tlb的刷新，smmu无需执行太多的指令了；

2.4 smmu和iommu的bypass

方式一：将iommu给彻底给bypass掉，linux提供了iommu.passthrough command line的选项，这个选项配置上后，dma默认不会走iommu，而是走传统的swiotlb方式的dma；

方式二：smmu v3的驱动默认支持驱动参数配置，disable_bypass，在系统中是默认关闭bypass的，我们可以通过这个来将某个smmu给bypass掉；

方式三：acpi或者dts中不配置相应的smmu节点，比较粗暴的办法。

3.smmu 的PMCG

ARM的SMMU提供了性能相关的统计寄存器(Performance Monitor Counter Groups - PMCG)，首先要确定使用的系统里有arm_smmuv3_pmu这个模块，或者它已经被编译进内核。

这个模块的代码在内核目录kernel/drivers/perf/arm_smmuv3_pmu.c,内核配置是：

CONFIG_ARM_SMMU_V3_PMU;

smmu pmcg 社区的patch 连接：

<https://lwn.net/Articles/784040/>

详细用法可以参见 社区pmcg的补丁文档，里面内容很简单。