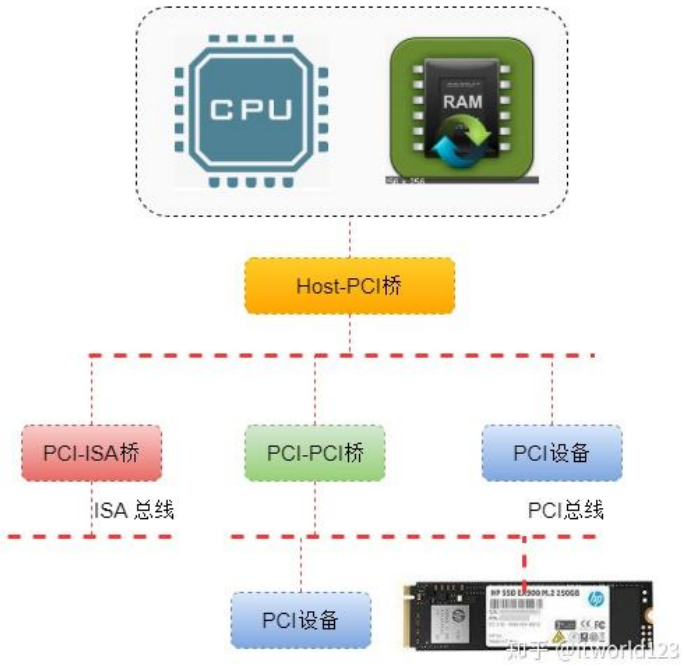
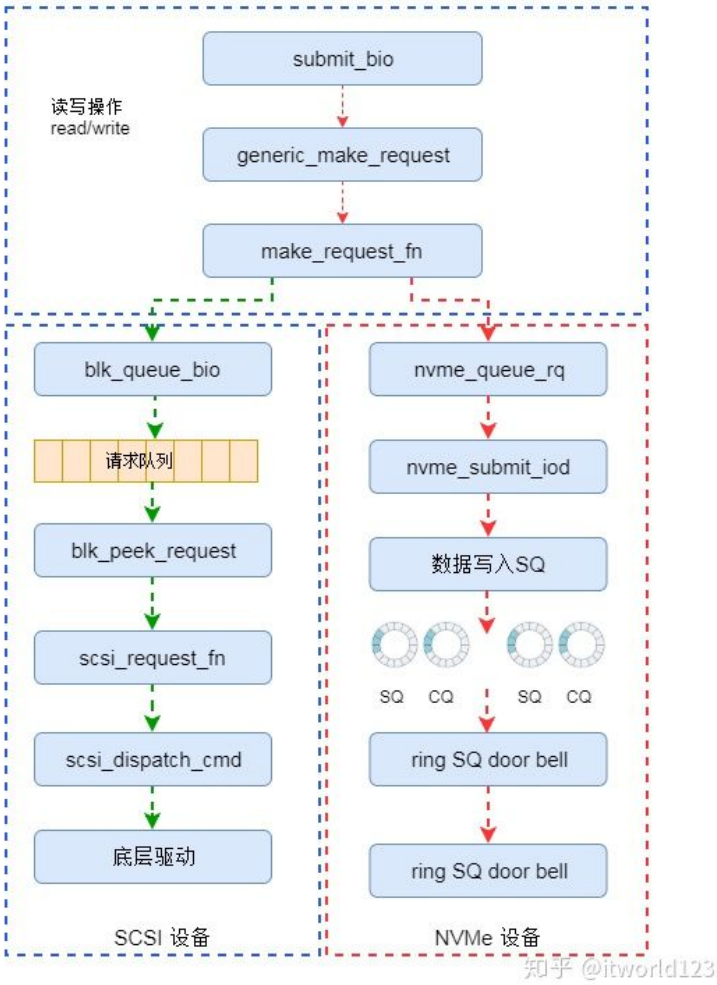
SSD

- ref
  - https://github.com/andyBrake/andyBrake.github.io/blob/master/doc/nvme.md
    - **NVMe驱动分析**

- BAT以及Google，Facebook等有着数量庞大的服务器，每家都是10万台级别，这注定了他们大概率不会从**存储厂商**购买昂贵的服务器，而是倾向于**自己研发廉价的服务器和存储设备来建设数据中心**
- SSD主控（FPGA）通过 **若干个通道**（channel）并行操作多块**FLASH颗粒**，可以提高底层带宽
  - 这也是为啥说SSD内部并行度很高，可以理解为就是**堆出来的**
  - **路宽了，单位时间能够传输的数据自然就多了**
  - 我们说SSD是多通道的，也主要是指SSD内部有能够**并行access的Flash**
  - 我们说的底层带宽指的就是Flash的带宽
- SSD内部
  - SSD对外暴露的是一个**连续的逻辑地址空间LBA**，说白了就是一个数组（数组元素是固定大小的块），其实包括HDD也是这样的
    - 其次，其内部**均**有逻辑地址到物理地址的**map table**
      - HDDs maintain an indirectiontable to handle bad block
        - 原来 HDD 也不是那么简单的逻辑
      - SSD 显然是要维护一个 **逻辑地址到物理地址的** 映射的
  - The Flash Translation Layer (FTL) in SSD performs an additional translation from logical blockaddresses (**LBA**) to physical page numbers (**PBN**)
- 面试成果所了解的一些SSD固件的info
  - 其实只有一点比较关键，那就是描述符由谁fetch下去，面试的结果看起来IO的描述符是完全硬件fetch下去的，随后软件做一些解析并发起真正数据的IO流程

NVMe 协议





- https://zhuanlan.zhihu.com/p/72234187
  - NVMe 协议除了可以走 PCIe 通道外，还可以走 FC 或 IB
    - 这个可以类比 HTTP 等用户态协议可以通过 TCP/UDP 底层协议实现是一个道理
      - 尽管 HTTP 协议貌似不支持 UDP，**仅仅是举一个例子**
    - **硬件的连通性是基础**
- NVMe 协议官方文档
  - https://nvmexpress.org/wp-content/uploads/NVMe-NVM-Express-2.0a-2021.07.26-Ratified.pdf
- SSD 型号的性能是有规范的，比如 intel P4500
  - https://www.intel.cn/content/www/cn/zh/products/sku/99029/intel-ssd-dc-p4500-series-4-0tb-2-5in-pcie-3-1-x4-3d1-tlc/specifications.html
    - 有这样一段话比较有意义
      - ref
        - https://www.intel.com/content/www/us/en/products/docs/memory-storage/solid-state-drives/ssd-dc-p4500-brief.html
      - Optimized for Storage Efficiency Across a Range of Workloads
        - This cloud-inspired SSD is built with an entirely new NVMe controller, optimized for read intensive workloads, and designed to maximize CPU utilization
        - With controller support for **up to 128 queues**, the Intel® SSD DC P4500 Series helps **minimize the risk of idle CPU cores** and performs most effectively on Intel platforms with Intel® Xeon® processors. The queue pair-to-CPU core mapping supports high drive count and also supports multiple SSDs scaling on Intel platforms
        - With the Intel® SSD DC P4500 Series, data centers can increase users, add more services, and perform more workloads per server, or quickly repartition to adapt to conditions. Now you can store more and know more
  - 若干指标如下
    - 顺序读取（最高）3200 MB/s
    - 顺序写入（最高）1800 MB/s
    - Random Read (100% Span) 645000 IOPS (4K Blocks)
    - Random Write (100% Span) 48000 IOPS (4K Blocks)

        - **带宽看顺序读写，IOPS看随机IO**
            - IOPS主要考察的是延迟，带宽检查路是否**足够宽**，延迟考察路是否**足够平**
            - **一次随机IO的延迟越低，则单位时间内的IOPS表现越好**
        - 接口 PCIe 3.1 x4, NVMe
            - PCIe 3.1 4 lane 的理论带宽是 4GB/s。这里完全可以 cover 住 SSD 底层 flash 的带宽
            - 底层带宽上去了，4 lane 的还不够用
- 明确一点，SSD的**队列深度**与**队列（指一对）数量**是SSD**出厂的时候就确定好的**
- An NVM Express queue. Each device has **at least two**
    - `one for admin commands/one for I/O commands`
    - 这个可以理解为设备出厂应该遵从的规范，至少2个
    - 按照NVMe SPEC协议中的标准，硬件最多支持64K个队列
        - 即IO队列最多65535个，admin队列1个
    - 但是由于硬件队列的增加会给NVMe SSD带来功耗的增加，所以不同的厂商在设计硬件队列个数时的考量是不同的，比如 **intel P3600支持32个队列**， **intel最新的P4500支持16384个**， **但是SUMSUNG PM963却只支持到8个**
        - 那么当CPU的个数超过硬件的队列个数，就会出现多个CPU共用一个硬件队列的情况，对性能就会产生影响
            - 因为软件在处理的时候，最终队列的数量是 `min(CPUs, queues)`
        - ref
            - https://www.sohu.com/a/305204622_467784
- 与其它块设备类似，NVMe设备初始化完成后会在/dev目录下出现一个文件。NVMe设备会出现一个形如 **nvmeXnY** 的设备文件
    - 如果对设备继续分区，那么设备文件形如 **nvmeXnYpZ**
- 队列深度的意思不言自明
    - **即可以在端口队列中等待服务的I/O请求数量**
        - **I/O SQ/CQ 深度可达64K，Admin SQ/CQ 深达 4K**
    - 使用SAS和SATA，排队的I/O请求数量很容易成为瓶颈。为了避免I/O请求由于超出队列深度而失败
        - 你必须创建许多HDD的LUN，以便所有I/O都能够快速进行
        - 说白了就是在**堆量来并行**
- NVMe发展的过程就是不断给SSD**开绿灯**的过程
    - **不需要合并 bio 这种常规操作**
        - **因为它支持随机写，而且速度快**
    - 每一个**NVMeController 允许最多65535个IO队列和一个Admin队列**
        - Admin队列在设备初始化之后随即创建,包括一个发送队列和一个完成队列
        - 其它的IO队列则是由Admin队列中发送的控制命令来产生的

NVMe驱动

起因

```
01:00.0 Non-Volatile memory controller: Samsung Electronics Co Ltd Device a80a
        Subsystem: Samsung Electronics Co Ltd Device a801
        Kernel driver in use: nvme
        Kernel modules: nvme
02:00.0 Non-Volatile memory controller: Samsung Electronics Co Ltd Device a80a
        Subsystem: Samsung Electronics Co Ltd Device a801
        Kernel driver in use: leapionvme
        Kernel modules: nvme
```

- 需要做一个nvme控制器（**NVMe协议本身的内容已经被卸载到硬件上了，最外面只提供了一个简单的寄存器**）的驱动，所以需要对NVMe驱动有比较详细的理解，所以就照着我们之前的套路，先把nvme驱动单独拉出来编译成模块，然后把本机的nvme ssd（**实验盘**）与内核驱动解绑，然后再重新绑定到我们自己的驱动（**leapioNVMe**）上
    - 中间遇到一个疑惑的地方（后面才发现真的是在**瞎想**），我自己的**系统盘**与**实验盘**是**一模一样**的，我再重新绑定我自己驱动的时候会不会把我的系统盘搞炸了，显然不会，因为现在系统盘已经与内核的nvme驱动绑定在一起了，即使我自己的驱动挂到pcie总线上然后去遍历设备的时候，即使这个时候会发现系统盘，但是系统盘已经有驱动绑定了，不会再绑定到我自己的驱动上的，否则全部都乱套了，所以**放开用**

驱动源码分析一波

```
/* nvme 驱动 */
static struct pci_driver nvme_driver = {
        .name           = "nvme",
        .id_table       = nvme_id_table,
        .probe          = nvme_probe,
    /* ... */
}

/* 一个 PCIe 设备是不是 NVMe 设备是通过这个表来识别的，好在能够生产SSD的厂商并不多 */
static const struct pci_device_id nvme_id_table[] =
{
    /* Intel 750/P3500/P3600/P3700 */
        { PCI_VDEVICE(INTEL, 0x0953), .driver_data = NVME_QUIRK_STRIPE_SIZE | NVME_QUIRK_DEALLOCATE_ZEROES, },
    /* Intel P3520 */
        { PCI_VDEVICE(INTEL, 0x0a53), .driver_data = NVME_QUIRK_STRIPE_SIZE | NVME_QUIRK_DEALLOCATE_ZEROES, },
    /* ... */
    /* Qemu emulated controller */
    { PCI_VDEVICE(INTEL, 0x5845), .driver_data = NVME_QUIRK_IDENTIFY_CNS | NVME_QUIRK_DISABLE_WRITE_ZEROES, },
    /* ... */
    /* #define PCI_CLASS_STORAGE_EXPRESS        0x010802，匹配原则也不全是厂商号与设备号，这个express class也可以 */
    { PCI_DEVICE_CLASS(PCI_CLASS_STORAGE_EXPRESS, 0xffffff) },
    { 0, }
}
```

module_init(nvme_core_init); —— NVMe的host驱动中有两个module_init，先描述一下不涉及pcie的这个

- `module_init(nvme_core_init);` —— `drivers/nvme/host/core.c`
  - `static int __init nvme_core_init(void)` —— 反正一顿全局变量的**初始化**，**不涉及PCIe**
    - `nvme_wq = alloc_workqueue("nvme-wq", WQ_UNBOUND | WQ_MEM_RECLAIM | WQ_SYSFS, 0);`
    - `nvme_reset_wq = alloc_workqueue("nvme-reset-wq", WQ_UNBOUND | WQ_MEM_RECLAIM | WQ_SYSFS, 0);`
    - `nvme_delete_wq = alloc_workqueue("nvme-delete-wq", WQ_UNBOUND | WQ_MEM_RECLAIM | WQ_SYSFS, 0);`
    - `result = alloc_chrdev_region(&nvme_ctrl_base_chr_devt, 0, NVME_MINORS, "nvme");`
      - Allocates a range of char device numbers
    - `nvme_class = class_create(THIS_MODULE, "nvme");`
      - `/sys/class` 目录下创建 `nvme`
    - `nvme_subsys_class = class_create(THIS_MODULE, "nvme-subsystem");`
      - `/sys/class` 目录下创建 `nvme-subsystem`
    - `result = alloc_chrdev_region(&nvme_ns_chr_devt, 0, NVME_MINORS, "nvme-generic");`
      - Allocates a range of char device numbers
    - `nvme_ns_chr_class = class_create(THIS_MODULE, "nvme-generic");`
      - `/sys/class` 目录下创建 `nvme-generic`
- 解释一下原因
  - `drivers/nvme/host` 目录下实际上长这样
    - `core.c`
    - `fc.c`
    - `fabrics.c`
    - `rdma.c`
    - `pci.c`
    - `...`
  - 其实写成这样才**好理解**，大家都是NVMe协议，只不过走的**链路不同**而已
    - PCIe就可以理解为我们的本地盘
      - 这也是我们目前主要所关注的，**不要觉得PCIe是本地的就区别对待**
    - 其它或多或少是其它机器的
  - 所以 `core.c` 中的内容其实是**不同链路nvme协议**所需的**公共部分**

重点check与PCIe相关的部分

- module_init(nvme_init);
  - return pci_register_driver(&nvme_driver); —— **熟悉的套路**

```
struct nvme_dev {
    /* ... */
    struct nvme_queue *queues;  /* 该设备所拥有的所有队列，队列id即qid */
    struct nvme_ctrl ctrl;  /* 每设备一个，而且不是设备指针的形式提供的 */
    unsigned int nr_allocated_queues;  /* 队列的数量，这个比较复杂，下文专门讲一下 */
    u32 q_depth;  /* 该设备的队列深度 */
    void __iomem *bar; /* nvme bar空间的基址 */
    u32 __iomem *dbs; /* dbs(doorbell sq 0 tail)的基址 */
    /* ... */
};
/* nvme_dev代表的是一个nvme设备，队列是有id的，一个设备所拥有的所有队列实际上是一个数组 */
/* 并且admin队列的数组 id=0 */
struct nvme_queue *nvmeq = &dev->queues[qid];
struct nvme_queue *adminq = &dev->queues[0];
/* nvme_queue在kernel中的数据结构，代表的是一对sq/cq */
/* 可以看到host是能够修改sq_tail_db与cq_head_db */
/* 而SSD controller则负责修改cq_tail_db与sq_head_db */
/* SSD controller一般是FPGA，甚至是Arm核心，与host CPU是平等的关系，应该按照平等关系理解 */
struct nvme_queue {
    struct nvme_dev *dev; /* 只是为了能够反引回去 */
    dma_addr_t sq_dma_addr; /* sq entry 在内存中的物理地址 */
        dma_addr_t cq_dma_addr; /* cq entry 在内存中的物理地址 */
        u16 sq_tail; /* host负责更新 */
        u16 cq_head; /* host负责更新 */
    /* ... */
};
```

- nvme_probe(struct pci_dev *pdev, const struct pci_device_id *id)
  - struct nvme_dev *dev; —— 明显与SCSI host不同的一点，一次probe得到的结果是**nvme设备**，而非**控制器**
  - dev = kzalloc_node(sizeof(*dev), GFP_KERNEL, node);
    - **把最关键的dev分配出来**
  - dev->nr_allocated_queues = nvme_max_io_queues(dev) + 1;
    - return num_possible_cpus();
      - 这里的基本的原则就是每一个CPU有一个IO队列，然后额外还有一个admin队列
        - 当然这个结果在后面与硬件交互的过程中会update，**以硬件的实际情况为准**
  - dev->queues = kcalloc_node(dev->nr_allocated_queues, sizeof(struct nvme_queue), GFP_KERNEL, node);
    - **直接分配nvme_queue，即NVMe的队列**
  - dev->dev = get_device(&pdev->dev); && pci_set_drvdata(pdev, dev); —— **常规操作**
  - result = nvme_dev_map(dev); —— **map bar空间，我手边的两个ssd都只有一个16K的bar空间**
    - pci_request_mem_regions(pdev, "nvme");
    - nvme_remap_bar(dev, NVME_REG_DBS + 4096) —— **NVME_REG_DBS = 0x1000(4096)** —— **SQ 0 Tail Doorbell**
      - if (size > pci_resource_len(pdev, 0)) —— pci_resource_len(pdev, 0) 就是**lspci**所看到的**16K**
        - return -ENOMEM;
      - if (dev->bar) iounmap(dev->bar); —— 怪不得**函数名**叫**remap**
      - dev->bar = ioremap(pci_resource_start(pdev, 0), size); —— 这个**size=8k**
        - 这里不会说把16K全部的map，仅仅是这里**指定的8K**
      - dev->bar_mapped_size = size; —— 这个 size=8K ，即**第一次**是**8K**
      - dev->dbs = dev->bar + NVME_REG_DBS; —— NVME_REG_DBS = 0x1000 /* SQ 0 Tail Doorbell */
        - **DBS是从bar空间的4K处开始**
  - INIT_WORK(&dev->ctrl.reset_work, nvme_reset_work); && INIT_WORK(&dev->remove_work, nvme_remove_dead_ctrl_work);
    - 初始化工作项，工作队列都是在 core.c 的 module_init 中创建的
  - result = nvme_setup_prp_pools(dev); —— **physical region page**
    - dev->prp_page_pool = dma_pool_create("prp list page", dev->dev, NVME_CTRL_PAGE_SIZE, NVME_CTRL_PAGE_SIZE, 0); —— 创建一个DMA的**大**池子，**4k大小，4k对齐**的池子
    - dev->prp_small_pool = dma_pool_create("prp list 256", dev->dev, 256, 256, 0); —— 再创建一个DMA的**小**池子，**256字节，256字节对齐的池子**

        - **Optimisation for I/Os between 4k and 128k**
    - 关于**PRP**的解释
        - PRP: physical region page, a PRP entry is a pointer to a physical memory page
        - PRP list is a set of PRP entries in a single page of contiguous memory
    - `quirks |= NVME_QUIRK_SHARED_TAGS; /* Prevent tag overlap between queues */` —— **quirks怪癖**, 对于一些特殊的盘有相应的补丁操作，其实就是flag
    - `quirks |= check_vendor_combination_bug(pdev);` —— 继续**更新怪癖**
    - `alloc_size = nvme_pci_iod_alloc_size();` —— **iod means io descriptors**
        - `size_t npages = max(nvme_pci_npages_prp(), nvme_pci_npages_sgl());` —— 这里的结果是**1**
        - `return sizeof(__le64 *) * npages + sizeof(struct scatterlist) * NVME_MAX_SEGS;`
            - `sizeof(__le64 *) = 8` / `sizeof(struct scatterlist) = 32` / `#define NVME_MAX_SEGS 127`
            - 所以**它**就是 `8*1 + 32*127 = 4072`
    - `WARN_ON_ONCE(alloc_size > PAGE_SIZE);` —— 这个值不能大于4096，大于之后会有**警告**
    - `dev->iod_mempool = mempool_create_node(1, mempool_kmalloc, mempool_kfree, (void *) alloc_size, GFP_KERNEL, node);`
        - **创建iod的内存池**
    - `result = nvme_init_ctrl(&dev->ctrl, &pdev->dev, &nvme_pci_ctrl_ops, quirks);` —— **详见下文**
        - **Initialize a NVMe controller structures. This needs to be called during earliest initialization so that we have the initialized structured around during probing**
    - **分隔符**
    - `dev_info(dev->ctrl.device, "pci function %s\n", dev_name(&pdev->dev));` —— 关键的里程碑**打印一下**
    - **分隔符**
    - `nvme_reset_ctrl(&dev->ctrl);` —— **详见下文**
    - `async_schedule(nvme_async_probe, dev);` —— **详见下文**
        - **等待reset_work，scan_work结束，释放ctrl**
    - `return 0;`

```
static const struct nvme_ctrl_ops nvme_pci_ctrl_ops = {
        .name                   = "pcie",
        .module                 = THIS_MODULE,
        .flags                  = NVME_F_METADATA_SUPPORTED |
                                  NVME_F_PCI_P2PDMA,
        .reg_read32             = nvme_pci_reg_read32,
        .reg_write32            = nvme_pci_reg_write32,
        .reg_read64             = nvme_pci_reg_read64,
        .free_ctrl              = nvme_pci_free_ctrl,
        .submit_async_event     = nvme_pci_submit_async_event,
        .get_address            = nvme_pci_get_address,
};
```

- `int nvme_init_ctrl(struct nvme_ctrl *ctrl, struct device *dev, const struct nvme_ctrl_ops *ops, unsigned long quirks)`
    - `ret = ida_simple_get(&nvme_instance_ida, 6, 7, GFP_KERNEL);` —— **Allocate an unused ID**
        - 这种写法我得到的nvme设备是**nvme6n1**（**块设备**）—— **6**是控制器的**设备号**
    - `ctrl->instance = ret;`
    - `device_initialize(&ctrl->ctrl_device);` —— **init device structure**
    - `ctrl->device = &ctrl->ctrl_device;`
    - `ret = dev_set_name(ctrl->device, "nvme%d", ctrl->instance);`
        - 这个名字我认识哈， `nvme6` ，它是一个**字符设备**
            - **6**是这个**字符设备**的**从**设备号
        - 块设备的**主从设备号**和这里没有关系
    - `cdev_init(&ctrl->cdev, &nvme_dev_fops);`
    - `ret = cdev_device_add(&ctrl->cdev, ctrl->device);` —— 这里直接创建了一个**字符设备**，这个是字符设备的**add**

```
/* NVMe寄存器 */
enum {
        NVME_REG_CAP    = 0x0000,       /* Controller Capabilities */
        NVME_REG_VS     = 0x0008,       /* Version */
        NVME_REG_INTMS  = 0x000c,       /* Interrupt Mask Set */
        NVME_REG_INTMC  = 0x0010,       /* Interrupt Mask Clear */
        NVME_REG_CC     = 0x0014,       /* Controller Configuration */
        NVME_REG_CSTS   = 0x001c,       /* Controller Status */
        NVME_REG_NSSR   = 0x0020,       /* NVM Subsystem Reset */
        NVME_REG_AQA    = 0x0024,       /* Admin Queue Attributes */
        NVME_REG_ASQ    = 0x0028,       /* Admin SQ Base Address */
        NVME_REG_ACQ    = 0x0030,       /* Admin CQ Base Address */
        NVME_REG_CMBLOC = 0x0038,       /* Controller Memory Buffer Location */
        NVME_REG_CMBSZ  = 0x003c,       /* Controller Memory Buffer Size */
        NVME_REG_BPINFO = 0x0040,       /* Boot Partition Information */
        NVME_REG_BPRSEL = 0x0044,       /* Boot Partition Read Select */
        NVME_REG_BPMBL  = 0x0048,       /* Boot Partition Memory Buffer
                                         * Location
                                         */
        NVME_REG_CMBMSC = 0x0050,       /* Controller Memory Buffer Memory
                                         * Space Control
                                         */
        NVME_REG_PMRCAP = 0x0e00,       /* Persistent Memory Capabilities */
        NVME_REG_PMRCTL = 0x0e04,       /* Persistent Memory Region Control */
        NVME_REG_PMRSTS = 0x0e08,       /* Persistent Memory Region Status */
        NVME_REG_PMREBS = 0x0e0c,       /* Persistent Memory Region Elasticity
                                         * Buffer Size
                                         */
        NVME_REG_PMRSWTP = 0x0e10,      /* Persistent Memory Region Sustained
                                         * Write Throughput
                                         */
        NVME_REG_DBS    = 0x1000,       /* SQ 0 Tail Doorbell */
};


static const struct blk_mq_ops nvme_mq_admin_ops = {
        .queue_rq       = nvme_queue_rq,
        .complete       = nvme_pci_complete_rq,
        .init_hctx      = nvme_admin_init_hctx,
        .init_request   = nvme_init_request,
        .timeout        = nvme_timeout,
};

static const struct blk_mq_ops nvme_mq_ops = {
        .queue_rq       = nvme_queue_rq,
        .complete       = nvme_pci_complete_rq,
        .commit_rqs     = nvme_commit_rqs,
        .init_hctx      = nvme_init_hctx,
        .init_request   = nvme_init_request,
        .map_queues     = nvme_pci_map_queues,
        .timeout        = nvme_timeout,
        .poll           = nvme_poll,
};
```

- `nvme_reset_ctrl(&dev->ctrl);`
  - `if (!nvme_change_ctrl_state(ctrl, NVME_CTRL_RESETTING))` —— 这里需要NVMe控制器的状态变为**NVME_CTRL_RESETTING**才能后续的reset操作
    - `return -EBUSY;`
  - `queue_work(nvme_reset_wq, &ctrl->reset_work);` —— 所以这里应该找的是 `ctrl->reset_work` 对应的**逻辑**，即**nvme_reset_work**（**稍后执行**）
    - `static void nvme_reset_work(struct work_struct *work)` —— 大量的队列初始化其实都丢到了**reset逻辑**中
      - `result = nvme_pci_enable(dev);` —— PCIe的enabe在这里**才**被执行
        - `pci_enable_device_mem(pdev);`
        - `pci_set_master(pdev);`
        - `dma_set_mask_and_coherent(dev->dev, DMA_BIT_MASK(dma_address_bits));`
        - `if (readl(dev->bar + NVME_REG_CSTS) == -1)` —— 要读**状态寄存器**，不能是 `-1`
          - `result = -ENODEV;`

- `result = pci_alloc_irq_vectors(pdev, 1, 1, PCI_IRQ_ALL_TYPES);` —— 回过头来看中断的时候才把这个补上，需要深入了解一下
  - 就是为PCIe设备分配中断向量用的，第二个参数与第三个参数分别是 `min_vecs/max_vecs` ，即用户需要的分配的**最小向量数**与**最大向量数**，其返回值是**分配的向量数量**
    - 这里的操作确实只是分配
- `dev->q_depth = min_t(u32, NVME_CAP_MQES(dev->ctrl.cap) + 1, io_queue_depth);`
  - 我的实验盘 **(Non-Volatile memory controller: Samsung Electronics Co Ltd Device a80a)** 的这个值 `NVME_CAP_MQES(dev->ctrl.cap) + 1 = 16384`
    - **这个深度还是有点夸张的**
- `dev->dbs = dev->bar + 4096;`
  - 门铃寄存器的地址是bar的**基址+4096**
- `nvme_map_cmb(dev);` —— NVMe **CMB**是NVMe SSD上的一块内存空间，可以通过**PCIe BAR**的方式暴露到主机内存空间中，并可**由主机直接读写**
  - 我手里这块SSD是没有支持cmd空间的

- `result = nvme_pci_configure_admin_queue(dev);` —— 配置NVMe的**管理队列**，这个地方开始**读写寄存器**了
  - `struct nvme_queue *nvmeq;`
  - `nvme_remap_bar(dev, db_bar_size(dev, 0));` —— 这里会把之前的unmap掉，**再重新map**，这个size的大小有变化
  - `dev->subsystem = readl(dev->bar + NVME_REG_VS) >= NVME_VS(1, 1, 0) ? NVME_CAP_NSSRC(dev->ctrl.cap) : 0;` —— 这里有读**版本寄存器**
  - `if (dev->subsystem && (readl(dev->bar + NVME_REG_CSTS) & NVME_CSTS_NSSRO))`
    - `writel(NVME_CSTS_NSSRO, dev->bar + NVME_REG_CSTS);` —— 可能会写**状态寄存器**
  - `result = nvme_disable_ctrl(&dev->ctrl);` —— **disable之后**才才能开始**配置admin queue**
    - `ctrl->ctrl_config &= ~NVME_CC_SHN_MASK;`
    - `ctrl->ctrl_config &= ~NVME_CC_ENABLE;`
    - `ret = ctrl->ops->reg_write32(ctrl, NVME_REG_CC, ctrl->ctrl_config);`
    - `return nvme_wait_ready(ctrl, ctrl->cap, false);`
  - `result = nvme_alloc_queue(dev, 0, NVME_AQ_DEPTH);` —— `#define NVME_AQ_DEPTH 32` —— **0号队列就是admi队列对**，**深度32**，深度本身也是可以配置的
    - `struct nvme_queue *nvmeq = &dev->queues[qid];` —— 对于admin queue，qid=0，表示第一个队列，非工作队列
    - `nvmeq->sqes = qid ? dev->io_sqes : NVME_ADM_SQES;` ，qid=0意味着admin队列
    - `nvmeq->q_depth = depth;`
    - `nvmeq->cqes = dma_alloc_coherent(dev->dev, CQ_SIZE(nvmeq), &nvmeq->cq_dma_addr, GFP_KERNEL);`
      - admin的完成队列位于内存， `#define CQ_SIZE(q) ((q)->q_depth * sizeof(struct nvme_completion))`
        - 很直观，就是**队列深度 x 每一个complete项的大小**
    - `nvme_alloc_sq_cmds(dev, nvmeq, qid);`
      - `nvmeq->sq_cmds = dma_alloc_coherent(dev->dev, SQ_SIZE(nvmeq), &nvmeq->sq_dma_addr, GFP_KERNEL);`
        - sq的entry也是位于内存的， `#define SQ_SIZE(q) ((q)->q_depth << (q)->sqes)`
          - 如果sq的深度是128，则 `SQ_SIZE(nvmeq)=8192` ，每一个entry的大小是64字节， `64*128=8192`
    - `nvmeq->cq_head = 0;`
    - `nvmeq->q_db = &dev->dbs[qid * 2 * dev->db_stride];`
      - admin的qid=0，**stride步幅=1**
    - `nvmeq->qid = qid;`
    - `dev->ctrl.queue_count++;`
  - `nvmeq = &dev->queues[0];`
  - `aqa = nvmeq->q_depth - 1;`
  - `aqa |= aqa << 16;`
  - `writel(aqa, dev->bar + NVME_REG_AQA);` —— **NVME_REG_AQA = 0x0024, Admin Queue Attributes**
    - 相当于配置**admin sq**队列的**深度**
  - `lo_hi_writeq(nvmeq->sq_dma_addr, dev->bar + NVME_REG_ASQ);` —— **NVME_REG_ASQ = 0x0028, Admin SQ Base Address**
  - `lo_hi_writeq(nvmeq->cq_dma_addr, dev->bar + NVME_REG_ACQ);` —— **NVME_REG_ACQ = 0x0030, Admin CQ Base Address**
  - **分隔符**，说白了就是在**配置一些寄存器**
  - `result = nvme_enable_ctrl(&dev->ctrl);`
    - `ctrl->ctrl_config |= (NVME_CTRL_PAGE_SHIFT - 12) << NVME_CC_MPS_SHIFT;`
    - `...`
    - `ctrl->ctrl_config |= NVME_CC_ENABLE;`

        - `ret = ctrl->ops->reg_write32(ctrl, NVME_REG_CC, ctrl->ctrl_config);` —— **NVME_REG_CC = 0x0014, Controller Configuration**
        - `return nvme_wait_ready(ctrl, ctrl->cap, true);` —— 等待**控制器ok**
    - `nvmeq->cq_vector = 0;` —— 这个值后面申请中断相关内容时会用到，**这个值是0**
    - `nvme_init_queue(nvmeq, 0);` —— **内存数据**的一些**初始化**行为
        - `nvmeq->sq_tail = 0;`
        - `nvmeq->cq_head = 0;`
        - `nvmeq->q_db = &dev->dbs[qid * 2 * dev->db_stride];`
        - `memset((void *)nvmeq->cqes, 0, CQ_SIZE(nvmeq));`
        - `...`
        - `wmb();`
    - `result = queue_request_irq(nvmeq);` —— **中断处理详见下文**
        - `if (use_threaded_interrupts)`
            - `return pci_request_irq(pdev, nvmeq->cq_vector, nvme_irq_check, nvme_irq, nvmeq, "nvme%dq%d", nr, nvmeq->qid);`
        - `else`
            - `return pci_request_irq(pdev, nvmeq->cq_vector, nvme_irq, NULL, nvmeq, "nvme%dq%d", nr, nvmeq->qid);`
    - `set_bit(NVMEQ_ENABLED, &nvmeq->flags);`
- `result = nvme_alloc_admin_tags(dev);`
    - `dev->admin_tagset.ops = &nvme_mq_admin_ops;` —— 这个ops是从块层出来之后继续向下的**ops**
    - `dev->admin_tagset.nr_hw_queues = 1;` —— 硬件队列的数量是**1**，admin队列确实只有一个
    - `dev->admin_tagset.queue_depth = NVME_AQ_MQ_TAG_DEPTH;` —— **NVME_AQ_MQ_TAG_DEPTH是在32的基础上减去了2，30左右**
    - `blk_mq_alloc_tag_set(&dev->admin_tagset);` —— 就是在分配资源标记**tag**
    - `dev->ctrl.admin_q = blk_mq_init_queue(&dev->admin_tagset);` —— 把**request_queue**这个对象分配出来
- `dev->ctrl.max_hw_sectors = min_t(u32, NVME_MAX_KB_SZ << 1, dma_max_mapping_size(dev->dev) >> 9);` —— 单次IO的数据总量（**扇区**为单位）
- `dev->ctrl.max_segments = NVME_MAX_SEGS;` —— **sg中的项数**
- `nvme_change_ctrl_state(&dev->ctrl, NVME_CTRL_CONNECTING);`
    - **Introduce CONNECTING state from nvme-fc/rdma transports to mark the initializing procedure here**
- `result = nvme_init_ctrl_finish(&dev->ctrl);` —— 这里有第一条cmd的发送，即 `nvme_admin_identify=0x06` ，返回关于controller与namespace能力和状态的数据结构 —— **2KB**
    - `ret = nvme_init_identify(ctrl);` —— 读取identify信息，并更新到内存数据结构
        - `ret = nvme_identify_ctrl(ctrl, &id);`
            - `struct nvme_command c = { };` —— 构造第一条nvme command
            - `int error;`
            - `c.identify.opcode = nvme_admin_identify;`
            - `c.identify.cns = NVME_ID_CNS_CTRL;`
            - `*id = kmalloc(sizeof(struct nvme_id_ctrl), GFP_KERNEL);`
            - `error = nvme_submit_sync_cmd(dev->admin_q, &c, *id, sizeof(struct nvme_id_ctrl));` —— **id其实是buffer，数据写回的地方，后面sizeof是bufferlen**
                - `return __nvme_submit_sync_cmd(q, cmd, NULL, buffer, bufflen, 0, NVME_QID_ANY, 0, 0);` —— **Returns 0 on success. If the result is negative, it's a Linux error code; if the result is positive, it's an NVM Express status code**
                - `req = nvme_alloc_request_qid(q, cmd, flags, qid);` —— **分配request并初始化**
                - `if (buffer && bufflen)`
                - `ret = blk_rq_map_kern(q, req, buffer, bufflen, GFP_KERNEL);` —— **map kernel data to a request, for passthrough requests**
                - `ret = nvme_execute_rq(NULL, req, at_head);`
                - `status = blk_execute_rq(disk, rq, at_head);`
                - `blk_execute_rq_nowait(bd_disk, rq, at_head, blk_end_sync_rq);`
                - `rq->rq_disk = bd_disk;`
                - `rq->end_io = done;`
                - `blk_account_io_start(rq);`
                - `blk_mq_sched_insert_request(rq, at_head, true, false);` —— 相当于是去了趟**块层**，SCSI的前几条命令也是这样的

- `ret = q->mq_ops->queue_rq(hctx, &bd);` —— nvme_queue_rq
  - `nvme_check_ready(&dev->ctrl, req, true);`
  - `ret = nvme_setup_cmd(ns, req);` —— **初始化过程中，这个地方没写啥**
  - `if (blk_rq_nr_phys_segments(req))`
    - `ret = nvme_map_data(dev, req, cmnd);`
  - `blk_mq_start_request(req);`
  - `nvme_submit_cmd(nvmeq, cmnd, bd->last);` —— 这下面是最经常说的敲门铃 —— **Copy a command into a queue and ring the doorbell**
    - `spin_lock(&nvmeq->sq_lock);`
    - `memcpy(nvmeq->sq_cmds + (nvmeq->sq_tail << nvmeq->sqes), cmd, sizeof(*cmd));`
      - **sq_tail初始化为0，sq_cmds就是cmd_slot**
    - `if (++nvmeq->sq_tail == nvmeq->q_depth) nvmeq->sq_tail = 0;` —— **如果绕完一圈了，就回到ring来**
      - `++nvmeq->sq_tail`，**即sq_tail自增1**
    - `nvme_write_sq_db(nvmeq, write_sq);`
      - `writel(nvmeq->sq_tail, nvmeq->q_db);`
        - 向尾指针寄存器中编写的是sq_tail的编号
    - `spin_unlock(&nvmeq->sq_lock);`
  - `if (blk_rq_is_poll(rq)) blk_rq_poll_completion(rq, &wait);` —— **反正就是要等待同步完成，就是wait在某一个变量上，中断会修改这个变量**
  - `else if (hang_check) while (!wait_for_completion_io_timeout(&wait, hang_check * (HZ/2)));`
  - `else wait_for_completion_io(&wait);`
- `if (nvme_req(rq)->flags & NVME_REQ_CANCELLED) return -EINTR;`
- `if (nvme_req(rq)->status) return nvme_req(rq)->status;`
- `return blk_status_to_errno(status);`

- 前面执行到操作可以理解为iocfacts类似的操作，读取硬件的一些基本消息用来初始化内存数据结构，这里浅作为分隔符存在
- `ctrl->max_hw_sectors = min_not_zero(ctrl->max_hw_sectors, max_hw_sectors);`
- `nvme_set_queue_limits(ctrl, ctrl->admin_q);`

- 接下来是一系列的配置操作，**可能会发送cmd**
- `ret = nvme_init_non_mdts_limits(ctrl);` —— Maximum Data Transfer Size
- `ret = nvme_configure_apst(ctrl);` —— Autonomous Power State Transitio
- `ret = nvme_configure_timestamp(ctrl);`
- `ret = nvme_configure_directives(ctrl);`
- `ret = nvme_configure_acre(ctrl);`

- `result = nvme_setup_io_queues(dev);`
  - `struct nvme_queue *adminq = &dev->queues[0];`
  - `nr_io_queues = dev->nr_allocated_queues - 1;`
  - `result = nvmqe_set_queue_count(&dev->ctrl, &nr_io_queues);`
    - `status = nvme_set_features(ctrl, NVME_FEAT_NUM_QUEUES, q_count, NULL, 0, &result);`
      - result 的 **某几位** 会包含 SSD 中队列的个数
    - `nr_io_queues = min(result & 0xffff, result >> 16) + 1;`
      - qemu 模拟的 SSD 的队列个数默认是 64，后面又 set 到 128 后跑了一下
        - num_queues=128
      - two example
        - result = 4063294
          - hex(4063294) is 0x3e003e
          - **0x3e is 62**
          - **63 + admi = 64**
        - result = 8257662
          - hex(8257662) is 0x7e007e
          - **0x7e is 126**
          - **126 + admi = 127**
    - `*count = min(*count, nr_io_queues);`
      - 如果用户指定了超级大的 队列数量，超过了 SSD 所能支持的数量

                - 那么这里的 count 只能等于 SSD 所能支持的最大队列数量
            - 如果 CPU 只有 3 个 core，但是 SSD 有 127 个 IO 队列可以供选择，但是其实也用不了那么多啊
                - 所以 count 最终的结果就是 3
            - 这里返回的 count 就是最终 **软件中 队列对** 的数量
        - do
            - size = db_bar_size(dev, nr_io_queues);
            - result = nvme_remap_bar(dev, size);
                - **nvme 队列的 bar 空间也要 map 一下**
            - --nr_io_queues;
        - while (1);
        - result = nvme_setup_irqs(dev, nr_io_queues);
            - **admin queue and each non-polled I/O queue 都需要中断**
            - 每一个队列都需要 **独立的中断**
        - result = nvme_create_io_queues(dev);
            - **for 循环中调用 nvme_alloc_queue**
    - nvme_dev_add(dev);
        - dev->tagset.ops = &nvme_mq_ops;
        - dev->tagset.nr_hw_queues = dev->online_queues - 1;
            - **SSD的队列数量是128，CPU的数量是15，所以最终结果是16**
        - dev->tagset.queue_depth = min_t(unsigned int, dev->q_depth, BLK_MQ_MAX_DEPTH) - 1;
            - **队列深度 1023**
        - ret = blk_mq_alloc_tag_set(&dev->tagset);
            - **软硬队列的 map 依据**
                - $25 = {mq_map = [34m0xffff888004245d80[m, nr_queues = 16, queue_offset = 0}
                - $26 = {mq_map = [34m0xffff888004245d40[m, nr_queues = 0, queue_offset = 0}
    - nvme_start_ctrl(&dev->ctrl); —— 启动NVMe设备，整个reset过程结束
        - nvme_start_keep_alive(ctrl);
        - nvme_enable_aen(ctrl);
        - if (ctrl->queue_count > 1)
            - nvme_queue_scan(ctrl);
            - nvme_start_queues(ctrl);
- async_schedule(nvme_async_probe, dev);

nvme 设备应该有一个 add disk 的过程 一个关键的函数就是 nvme_alloc_ns

- 重点关注一下队列这部分是如何分配的
- SSD 所提供的这个队列就是 nvme 协议中这个硬件队列的上下文 **hcxt**

```
// 一个 nvme 的 namespace 对应一个 独立的 nvme_dev，独立 的 disk 对象等等
// 分区不会增加 namespace，每一个 namespace 都是可以分区的
struct nvme_ns {
    ...
        struct nvme_ctrl *ctrl;  // 每一个 nvme_dev 与一个 ctrl 对应
        struct request_queue *queue;  // 这个数据结构会描述 软队列/硬队列 等等 request queues 是与设备挂钩的
        struct gendisk *disk;
        struct nvm_dev *ndev;
    ...
};
```

- nvme_scan_work
    - nvme_scan_ns_list
        - nvme_validate_or_alloc_ns(ctrl, nsid);
            - nvme_alloc_ns(ctrl, nsid, &ids);
- nvme_alloc_ns
    - struct nvme_ns *ns;
    - struct gendisk *disk;
    - ns->queue = blk_mq_init_queue(ctrl->tagset);

- tips
  - `ctrl->tagset` 包含 软硬队列的 map 关系以及 队列深度等info，这个 info 是 **取决于设备** 的
  - 先把 **虚拟机的核数** 升上去，否则单核有些东西测试不到，直接把 numa 也一步到位
- `return blk_mq_init_queue_data(set, NULL);`
  - `uninit_q = blk_alloc_queue(set->numa_node);`
    - `q = kmem_cache_alloc_node(blk_requestq_cachep, GFP_KERNEL | __GFP_ZERO, node_id);`
  - `q = blk_mq_init_allocated_queue(set, uninit_q, false);`
    - tips
      - 这个队列不需要电梯，说白了就是 **不需要调度**，所有的 IO request 都是 **直接丢到 SSD 队列中** 的
    - `q->mq_ops = set->ops;`
      - **nvme_mq_ops**
    - `blk_mq_alloc_ctxs(q);`
      - `ctxs->queue_ctx = alloc_percpu(struct blk_mq_ctx);`
        - 分配 **软件队列**，**每 CPU 一个**
    - `blk_mq_realloc_hw_ctxs(set, q);`
      - tips
        - 根据 set 来分配硬件队列上下文
        - map 的话，正常是能够保证每一个 Core 都有一个硬件队列，如果 CPU 的核心数量如果大于 SSD 能够提供的队列数量
          - **则 map 负责把软队列 map 到硬队列上，确实会有 share 的情况**
      - `for (i = 0; i < set->nr_hw_queues; i++)`
        - `hctx = blk_mq_alloc_and_init_hctx(set, q, i, node);`
    - `blk_mq_init_cpu_queues(q, set->nr_hw_queues);`
    - `blk_mq_add_queue_tag_set(set, q);`
    - `blk_mq_map_swqueue(q);`
- `disk->fops = &nvme_bdev_ops;`
- `disk->private_data = ns;`
- `disk->queue = ns->queue;`
- `device_add_disk(ctrl->device, ns->disk, nvme_ns_id_attr_groups);`

NVMe 协议的 IO 过程

```
static const struct block_device_operations nvme_bdev_ops = {
        .owner          = THIS_MODULE,
        .ioctl          = nvme_ioctl,
        .compat_ioctl   = nvme_compat_ioctl,
        .open           = nvme_open,
        .release        = nvme_release,
        .getgeo         = nvme_getgeo,
        .report_zones   = nvme_report_zones,
        .pr_ops         = &nvme_pr_ops,
};

blk_qc_t submit_bio_noacct(struct bio *bio)
{
        if (!bio->bi_bdev->bd_disk->fops->submit_bio)
                return __submit_bio_noacct_mq(bio);
        return __submit_bio_noacct(bio);
}
```

- 这个地方之前的理解有错，我 **想当然** 的以为 SSD 是有 bio->bi_bdev->bd_disk->fops->submit_bio
  - 但是自己跑过一次之后才知道 SSD 也是走的 mq 的路子

    - 确实是自己之前理解有偏差了，但是示意图中 **NVMe 协议部分** 也是会部分经过一下 mq 的，尽管最终的结果是 bypass 了，但是并不是 **直接由文件系统到驱动** 的
- 把 SSD nvme 协议这里的 **单独拉出来** 看一下
- `__submit_bio_noacct_mq(bio);`
    - `ret = blk_mq_submit_bio(bio);`
        - 同样需要将 bio 转变为 request，以 insert 到 SSD 所对应的 struct request_queue 中的 传输单元
            - 并不是每一次的 **bio提交** 都会转换成 **submit cmd**，有 缓存 **去尽量多的合并相邻的 bio**
        - f2fs 的存盘策略的表现是这样的，有2条分支被涉及到
            - `plug = blk_mq_plug(q, bio);`
            - `if (unlikely(is_flush_fua))`
                - `blk_insert_flush(rq);`
                - `blk_mq_run_hw_queue(data.hctx, true);`
                    - 开始走 NVMe 协议的那一套了
            - `else if (plug && (q->nr_hw_queues == 1 || q->mq_ops->commit_rqs || !blk_queue_nonrot(q)))`
                - tips
                    - 先用 **plug机制** 替文件系统兜底
                        - 用于尽可能合并 bio。f2fs 是 append 型的文件系统
                - `if (request_count >= BLK_MAX_REQUEST_COUNT || (last && blk_rq_bytes(last) >= BLK_PLUG_FLUSH_SIZE))`
                    - `blk_flush_plug_list(plug, false);`
                        - 刷到 ctx 中，即 **软件队列** 中
                - `blk_add_rq_to_plug(plug, rq);`

SSD中断

```
           CPU0       CPU1       CPU2       CPU3       CPU4       CPU5       CPU6       CPU7
 24:         10          0          0          0          0          0          0          0   nvme0q0,nvme0q1
 25:         10          0          0          0          0          0          0          0   nvme1q0,nvme1q1
 26:          0         10          0          0          0          0          0          0   nvme1q2
 ...                                                                                          ...
 27:          0          0          0          0          0          0          0         10   nvme1q8
 26:          0         10          0          0          0          0          0          0   nvme0q2
 ...                                                                                          ...
 27:          0          0          0          0          0          0          0         10   nvme0q8
```

- 可以通过 `cat /proc/interrupts | grep nvme | wc -l` 来看与nvme协议相关的中断
- **10.xxx.203** 有2块SSD，8核心，所以一共能够看到16个中断
    - admi队列的中断与第一个io队列的中断都是绑定在CPU0上的
    - 中断也有亲和性，只会中断自己的CPU
- `cd /proc/irq`
    - 可以去看被分配的中断号的信息

NVMe 协议 from 蛋蛋

- 上面的分析已经将 rq 写入了设备队列，下面先把 NVMe 协议理一下，再继续分析代码部分
    - **基本目标就是队列中的 rq 到 nvme_submit_cmd 的过程**

rq 到 nvme_submit_cmd

- `blk_mq_run_hw_queue(data.hctx, true);`
    - `...`

    - `__blk_mq_run_hw_queue(hctx);`
        - tips
            - **Send pending requests to the hardware**
        - `blk_mq_sched_dispatch_requests(hctx);`
            - `__blk_mq_sched_dispatch_requests(hctx)`
                - `blk_mq_dispatch_rq_list(hctx, &rq_list, 0)`
                    - `ret = q->mq_ops->queue_rq(hctx, &bd);`
- `static blk_status_t nvme_queue_rq(struct blk_mq_hw_ctx *hctx, const struct blk_mq_queue_data *bd)`
    - `struct request *req = bd->rq;`
    - `struct nvme_command cmnd;`
    - `ret = nvme_setup_cmd(ns, req, &cmnd);`
    - `blk_mq_start_request(req);`
    - `nvme_submit_cmd(nvmeq, &cmnd, bd->last);`
        - tips
            - 参数
                - nvmeq: The queue to use
                - cmd: The command to send
                - write_sq: whether to write to the SQ doorbell
        - `memcpy(nvmeq->sq_cmds + (nvmeq->sq_tail << nvmeq->sqes), cmd, sizeof(*cmd));`
            - **cmd copy到sq队列中**
        - `++nvmeq->sq_tail;`
            - **tail指针有对应的add**
        - `nvme_write_sq_db(nvmeq, write_sq);`
            - `if (!write_sq)`
                - 如果本次提交不需要update门铃寄存器，则SSD Controller也不会有动作，只是**简单的入队操作**
            - `else`
                - tips
                    - Host更新这个寄存器的同时，也是在告诉SSD Controller有新命令了，需要你去取
                - `writel(nvmeq->sq_tail, nvmeq->q_db);`
                    - nvmeq->sq_tail: 数据
                    - nvmeq->q_db: 需要写入的虚拟地址
                        - 这个是 SSD 某队列的 IO 端口映射后的虚拟地址
- 到此为止，SSD Controller负责执行cmd
    - 即数据IO等操作，即数据IO是SSD Controller主动完成的，而非是Host CPU
        - 其实讲道理这个数据的搬运是**DMA**做的，**但是一定不需要打扰的CPU**
            - DMA在硬件上
    - 对NVMe/PCIe来说，SSD收到Write命令后，**通过PCIe去Host的内存数据所在位置读取数据**
        - DMA也依赖的是PCIe
        - 然后把这些数据写入到闪存中，同时得到LBA与闪存位置的映射关系
    - SSD根据LBA，查找映射表，找到对应闪存物理位置，然后读取闪存获得数据。数据从闪存读上来以后，对NVMe/PCIe来说，SSD会通过PCIe把数据写入到Host指定的内存中。这样就完成了Host对SSD的读访问
        - 无论是读/写，数据搬运都是DMA完成的
- 并且可以看到Host CPU在写完sq队列后，负责更新sq_tail_db寄存器（**门铃寄存器，host敲SSD门，快递达到请取用**）
    - 可以理解为SSD Controller的sq_tail_db寄存器被修改后
        - 会触发对应的中断
        - 或者SSD Controller内部会poll sq_tail_db寄存器的变化
- 然后SSD Controller执行完毕sq队列中的cmd之后，SSD Controller会修改sq队列的head db寄存器
    - 这个寄存器是map到Host CPU的地址空间中的，即该寄存器对Host的SSD Controller是可见的
- 所以Host CPU与SSD Controller之间是**协作的对等关系**
    - 对于sq队列
        - Host CPU生产cmd后更新tail
        - SSD Controller消费cmd后更新head
            - 所以Host CPU是没法写sq的head的
- **继续**
- SSD Controller消费cmd并更新head后，需要通知Host这些CMD完事了

    - 如果Host CPU有task是while poll等待IO完成的，这个通知是必然的
    - 先看一眼函数调用栈
        - nvme_complete_rq在echo将数据写入后，要过比较久才能触发真正的IO过程，通过fsync来解决
            - **fsync file**
    - **顺便瞅一眼中断是从哪里来的**
        - `nvme_complete_rq`
- `common_interrupt` in `arch/x86/include/asm/idtentry.h`
    - `DEFINE_IDTENTRY_IRQ(common_interrupt)` in `arch/x86/kernel/irq.c`
        - `handle_irq(desc, regs);`
            - ...
            - `retval = __handle_irq_event_percpu(desc, &flags);` in `kernel/irq/handle.c`
                - `res = action->handler(irq, action->dev_id);`
                - `static irqreturn_t nvme_irq(int irq, void *data)`
                - `nvme_process_cq(nvmeq)`
                - `nvme_handle_cqe(nvmeq, nvmeq->cq_head);`
                - `nvme_pci_complete_rq(req);`
                - `nvme_complete_rq(req);`
- SSD Controller将sq队列中的cmd消化完毕之后，需要通知Host CPU
    - 假设消化了4条cmd。则SSD Controller会在CQ队列中增加4条完成的entry，并且更新CQ队列的tail指针寄存器
- 下一步是SSD Controller发起的MSI中断
    - 这个地方不确定是不是**更新CQ队列的tail指针寄存器**导致的MSI中断，**在时序上是符合的**
- Host CPU 执行之前注册的**中断handler nvme_irq**
    - 主要目的是消化CQ队列，**并更新 CQ 队列的 head 指针 以及 head 的 db 寄存器**
    - 可能会调用上层注册好的 **call_back 函数**
- **中断回调这部分详见 IO的中断回调部分**

队列	tail DB	head DB
sq	host CPU	SSD Controller
cq	SSD Controller	host CPU

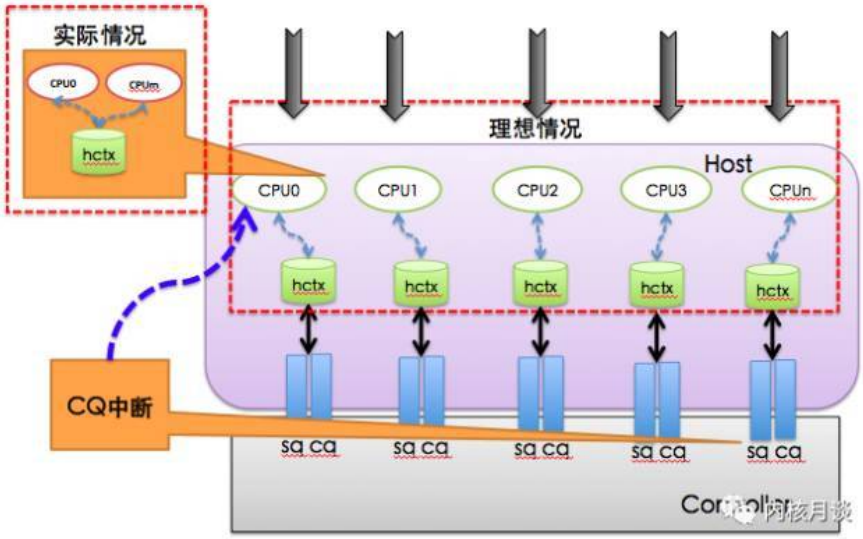
蛋蛋 总结的 NVMe 协议

1. host与SSD之间的通信协议
2. NVMe协议与PCIe协议的关系可以理解为HTTP与TCP之间的关系，当然HTTP也可以由UDP来完成，就像NVMe协议可以由其它协议完成一样
    - 但是NVMe协议与PCIe协议组合是最牛逼的
3. host 写 IO端口 的 doorbell 寄存器（寄存器直接 map 到 CPU 地址空间的）来通知 SSD 控制器取指令
4. SSD 控制器通过 MSI 中断（写指定内存以触发中断）通知 host 完成了一些 NVMe cmd
    - **不确定是否是通过写 cq 的 tail 尾指针寄存器所触发的 MSI 中断**
5. Host端每个Core可以有一个或者多个SQ，但只有一个CQ。给每个Core分配一对SQ/CQ好理解，为什么一个Core中还要多个SQ呢
    - 一是多线程
    - 二是QoS需求，什么是QoS？Quality of Service，服务质量。脑补一个场景，蛋蛋一边看小电影，同时迅雷在后台下载小电影，由于电脑配置差，看个小电影都卡。蛋蛋最讨厌看小电影的时候卡顿了，因为你刚刚燃起的激情会被那个缓冲浇灭。所以，蛋蛋不要卡顿！怎么办？
        - NVMe建议，**你设置两个SQ**，一个赋予 **高优先级**， **一个低优先级**，把看小电影所需的命令放到高优先级的SQ，迅雷下载所需的命令放到低优先级的SQ，这样，你那破电脑就能把有限的资源优先满足你看小电影了。至于迅雷卡不卡，下载慢不慢，这个时候已经不重要了。能让蛋蛋舒舒服服的看完一个小电影，就是好的QoS
6. NVMe的主要目的是解决一个问题
    - **延迟**
        - 延迟影响的指标是 IOPS
    - 以往的 AHCI 协议 被设计为起到一个 **翻译** 的作用
    - 名词解释

- HBA: host bus adapter
  - HBA 是 PCIe 协议连进来的，然后 HBA 再去接 SATA 的 SSD 或 HDD
- AHCI: Advanced Host Controller Interface
  - HBA 上跑的是一个 AHCI 协议，所以跑着 AHCI 协议的 HBA 会将 CMD from host 给到 SATA 的 storage
  - ACHI 理解为 SATA 上的数据传输协议

有一篇文章讲 NVMe 角度不一样

- ref
  - https://cloud.tencent.com/developer/article/1442277



1. MQ 架构对性能的最大提升在于 **将锁的粒度按照硬件队列来拆分**
2. 队列如果太深的话，延迟就会增大
3. 使用direct IO的业务大多在应用本身就已经做了一层cache，不依赖OS提供的page cache
4. linux kernel中的block layer通过REQ_SYNC与~REQ_SYNC这两种不同的标志来区分这两类IO
5. 中断绑定CPU，建议下发的SQ的CPU与响应的CQ的CPU保持一致，这样各自CPU来处理自己的事情，互相业务与中断不干扰

图解PCIe Gen4&5总线和NVMe SSD测试