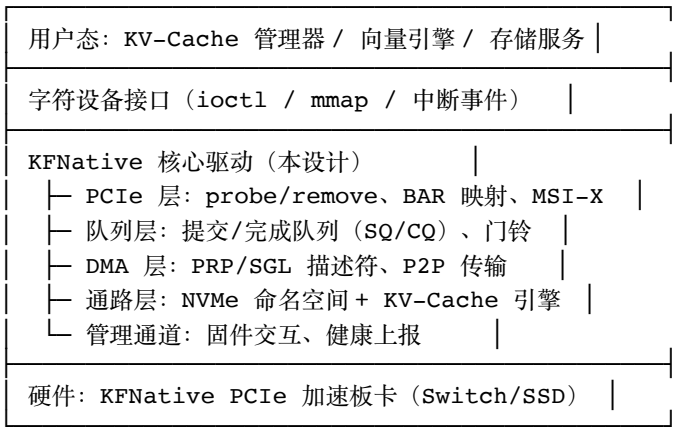
KFNative™ 驱动与固件设计文档

撰写日期：2026-09-12 作者：周舟

第 1 章 驱动架构与 PCIe BAR 映射

1.1 驱动总体架构

KFNative™ 驱动采用 Linux 内核 PCIe 驱动模型，层次划分为：



- **设备模型**：板卡作为单一 PCIe Function 枚举（SR-IOV 场景可启用 VF 供 K8s 多租户，设计推算）；BAR0 为控制寄存器区，BAR2 为队列/门铃区，BAR4 为可选共享缓存（CMB，对齐白皮书 5.1.5 企业盘 CMB 特性）。
- **与固件分工**：驱动负责主机侧资源管理与请求下发；板卡固件（第 4 章）负责队列仲裁、DMA 调度与 NVMe 命令转发。

1.2 PCIe BAR 布局（设计推算）

BAR	空间	内容
BAR0	4 KB（控制）	版本、能力、复位、中断状态/掩码、健康寄存器
BAR2	1 MB（队列区）	SQ/CQ 门铃、队列配置、完成队列头尾指针
BAR4	可选 64 MB（CMB）	板载缓存/共享内存，用于 KV 页暂存与 NVMe CMB

寄存器访问用 `readl/writel/ioread32`，关键结构在 `kfnative.h` 定义；门铃写入采用 `wmb()` 保证与 DMA 的可见性顺序。

1.3 probe 流程（C 骨架）

```
#include <linux/pci.h>
#include <linux/interrupt.h>
#include <linux/dma-mapping.h>

#define KFN_DRV_NAME "kfnative"
#define KFN_BAR_CTRL 0
#define KFN_BAR_QUEUE 2
#define KFN_BAR_CMB 4

static const struct pci_device_id kfn_pci_ids[] = {
    { PCI_DEVICE(PCI_VENDOR_ID_KFN, 0x4B46) }, /* 设计推算 VID/DID */
    { 0 }
};
MODULE_DEVICE_TABLE(pci, kfn_pci_ids);

struct kfn_dev {
    struct pci_dev *pdev;
    void __iomem *bar_ctrl;
    void __iomem *bar_queue;
    void __iomem *bar_cmb;
    struct kfn_queue *sqs;    /* 提交队列数组 */
    struct kfn_queue *cqs;    /* 完成队列数组 */
    struct kfn_msix *msix;    /* MSI-X 向量 */
    struct kfn_dma_pool *dma;
    spinlock_t lock;
};

static int kfn_probe(struct pci_dev *pdev, const struct pci_device_id *id)
{
    struct kfn_dev *kf;
    int err;
```

```
    kf = kzalloc(sizeof(*kf), GFP_KERNEL);
    if (!kf)
        return -ENOMEM;
    pci_set_drvdata(pdev, kf);
    kf->pdev = pdev;
    spin_lock_init(&kf->lock);

    err = pci_enable_device(pdev);
    if (err)
        goto err_free;
    pci_set_master(pdev);

    err = pci_request_regions(pdev, KFN_DRV_NAME);
    if (err)
        goto err_disable;

    err = pci_set_dma_mask(pdev, DMA_BIT_MASK(64));
    if (err)
        err = pci_set_dma_mask(pdev, DMA_BIT_MASK(32));
    if (err)
        goto err_regions;

    kf->bar_ctrl = pci_iomap(pdev, KFN_BAR_CTRL, 0);
    kf->bar_queue = pci_iomap(pdev, KFN_BAR_QUEUE, 0);
    kf->bar_cmb = pci_iomap(pdev, KFN_BAR_CMB, 0);
    if (!kf->bar_ctrl || !kf->bar_queue)
        goto err_unmap;

    /* 复位板卡并等待固件就绪（见第 4 章固件握手） */
    if (kfn_fw_reset_and_ready(kf))
        goto err_unmap;

    /* 建立队列与 MSI-X（见第 2 章） */
    err = kfn_setup_queues(kf);
    if (err)
        goto err_unmap;
    err = kfn_setup_msix(kf);
    if (err)
        goto err_queues;

    /* 注册字符设备 / misc 设备 */
    err = kfn_misc_register(kf);
    if (err)
        goto err_msix;

    dev_info(&pdev->dev, "KFNative ready, BARs mapped\n");
    return 0;

err_msix:
    kfn_teardown_msix(kf);
err_queues:
    kfn_teardown_queues(kf);
err_unmap:
    if (kf->bar_cmb) pci_iounmap(pdev, kf->bar_cmb);
    if (kf->bar_queue) pci_iounmap(pdev, kf->bar_queue);
    if (kf->bar_ctrl) pci_iounmap(pdev, kf->bar_ctrl);
    pci_release_regions(pdev);
```

```
err_disable:
    pci_disable_device(pdev);
err_free:
    kfree(kf);
    return -ENODEV;
}
```

probe 关键决策：先使能设备与 DMA（64 位掩码，回落 32 位），再映射 BAR；固件握手成功后才建队列与中断；任一环节失败按逆序回滚，避免资源泄漏。

1.4 remove 与电源管理

- `kfn_remove`：注销 misc 设备 → 关中断 → 清队列 → 固件停服 → iounmap → release_regions → disable_device → kfree。
- 电源管理：`kfn_suspend/resume` 保存/恢复队列头尾指针与门铃状态；板卡低功耗模式由固件配合（第 4 章电源状态机，设计推算）。

> 本章结论：驱动采用标准 PCIe 模型，以"BAR0 控制 + BAR2 队列 + BAR4 CMB"三区布局支撑控制、队列与缓存；probe 骨架实现使能-映射-握手-建队列-中断-注册的完整流程，错误路径逆序回滚。

第 2 章 队列管理与中断处理

2.1 队列模型：NVMe 风格 SQ/CQ

KFNative 板卡管理多组 SSD，队列设计对齐 NVMe 语义（协议背景：NVMe 支持最多 65,535 队列、MSI-X 最多 2048 向量），并为 KV-Cache 增加轻量命令类型：

- **I/O 队列组**：每 SSD 命名空间 1 组 SQ/CQ（深度 256–1024，设计推算），支持多队列分摊（对齐 NVMe 多队列）。
- **KV 队列组**：承载 KV-Cache 读写命令（`KV_PUT/KV_GET/KV_DEL`），走与 I/O 相同的门铃-完成机制，命令格式为自定义扩展（设计推算）。
- **管理队列**：0 号队列组承担固件配置、拓扑/健康查询。

门铃机制：主机写 BAR2 门铃（尾指针）通知固件有新命令；固件写 CQ 头指针 + 中断通知主机完成。顺序语义用 `wmb()`（写门铃前）与 `dma_rmb()`（读 CQ 前）保证。

2.2 队列建立（C 骨架）

```
struct kfn_queue {
    void *sq_buf;                   /* 提交队列 DMA 缓冲 */
    dma_addr_t sq_dma;
    u32 sq_depth;
    u32 sq_head;                    /* 主机侧视角 */
    u32 sq_tail;                    /* 门铃值 */
    void *cq_buf;                   /* 完成队列 DMA 缓冲 */
    dma_addr_t cq_dma;
    u32 cq_depth;
    u32 cq_head;
    u32 cq_tail;
    u16 qid;
    u16 irq_vec;                    /* 对应 MSI-X 向量 */
};

/* 队列命令字（设计推算，自定义扩展） */
struct kfn_cmd {
    __u8 opcode;                    /* KFN_OP_NVME / KFN_OP_KV_PUT / ... */
    __u8 flags;
    __u16 cid;                      /* 命令 ID */
    __u32 nsid;                     /* 命名空间/SSD ID */
    __u64 lba;                      /* 块地址（NVMe）或 KV key 指针（KV） */
    __u64 kv_key;
    __u64 dma_addr;                 /* 主机/设备侧数据地址 */
    __u32 len;
    __u32 rsvd;
} __packed;

#define KFN_DB_OFF(qid)   (0x100 + (qid) * 8)   /* 门铃寄存器偏移，设计推算 */

int kfn_queue_create(struct kfn_dev *kf, struct kfn_queue *q, u16 qid,
                     u16 depth, u16 irq_vec)
{
    int err;
    q->sq_buf = dma_alloc_coherent(&kf->pdev->dev, depth * sizeof(struct
kfn_cmd),
                                   &q->sq_dma, GFP_KERNEL);
    if (!q->sq_buf)
        return -ENOMEM;
    q->cq_buf = dma_alloc_coherent(&kf->pdev->dev, depth * sizeof(struct
kfn_cq_entry),
                                   &q->cq_dma, GFP_KERNEL);
    if (!q->cq_buf) {
        dma_free_coherent(&kf->pdev->dev, depth * sizeof(struct kfn_cmd),
                          q->sq_buf, q->sq_dma);
        return -ENOMEM;
    }
    q->qid = qid; q->sq_depth = q->cq_depth = depth;
    q->sq_head = q->sq_tail = q->cq_head = q->cq_tail = 0;
    q->irq_vec = irq_vec;

    /* 配置固件队列（写 BAR0 配置寄存器 + 门铃） */
    err = kfn_fw_config_queue(kf, qid, depth, q->sq_dma, q->cq_dma, irq_vec);
    if (err) {
```

```
        dma_free_coherent(&kf->pdev->dev, depth * sizeof(struct kfn_cq_entry),
                          q->cq_buf, q->cq_dma);
        dma_free_coherent(&kf->pdev->dev, depth * sizeof(struct kfn_cmd),
                          q->sq_buf, q->sq_dma);
        return err;
    }
    return 0;
}

/* 提交一条命令：写 SQ 尾 + 门铃 */
void kfn_submit(struct kfn_dev *kf, struct kfn_queue *q, struct kfn_cmd *cmd)
{
    unsigned long flags;
    struct kfn_cmd *slot = (struct kfn_cmd *)q->sq_buf + (q->sq_tail & (q-
>sq_depth - 1));

    spin_lock_irqsave(&kf->lock, flags);
    *slot = *cmd;                   /* 拷贝命令字 */
    wmb();                          /* 保证命令可见后再写门铃 */
    q->sq_tail++;
    writel(q->sq_tail, kf->bar_queue + KFN_DB_OFF(q->qid));
    spin_unlock_irqrestore(&kf->lock, flags);
}
```

2.3 MSI-X 中断注册（C 骨架）

```
static irqreturn_t kfn_irq_handler(int irq, void *data)
{
    struct kfn_queue *q = data;
    u32 done = 0;
    struct kfn_cq_entry *ent = (struct kfn_cq_entry *)q->cq_buf + (q->cq_head &
(q->cq_depth - 1));

    dma_rmb();                      /* 读 CQ 前保证 DMA 写入可见 */
    while (ent->status & KFN_CQ_VALID) {
        kfn_complete_one(q, ent); /* 分发到 NVMe/KV 完成处理 */
        ent->status = 0;            /* 清有效位，归还槽位 */
        q->cq_head++;
        done++;
        ent = (struct kfn_cq_entry *)q->cq_buf + (q->cq_head & (q->cq_depth -
1));
        dma_rmb();
    }
    /* 写 CQ 头指针（固件据此回收） */
    writel(q->cq_head, kf_bar_queue(q) + KFN_CQH_OFF(q->qid));
    if (done)
        kfn_wakeup_waiters(q);      /* 唤醒等待 ioctl 的用户态 */
    return IRQ_HANDLED;
}

int kfn_setup_msix(struct kfn_dev *kf)
{
    int i, nvec, err;
    nvec = pci_alloc_irq_vectors(kf->pdev, 1, KFN_MAX_IRQ, PCI_IRQ_MSIX);
    if (nvec < 0)
```

```
        return nvec;

    for (i = 0; i < kf->nr_queues; i++) {
        int irq = pci_irq_vector(kf->pdev, i);
        err = request_irq(irq, kfn_irq_handler, 0, KFN_DRV_NAME, &kf->qs[i]);
        if (err)
            goto fail;
    }
    return 0;
fail:
    while (--i >= 0)
        free_irq(pci_irq_vector(kf->pdev, i), &kf->qs[i]);
    return err;
}
```

中断策略：每队列 1 个 MSI-X 向量（对齐 NVMe 多队列/MSI-X 能力）；中断处理只做"取完成 → 分发 → 写头指针"，无锁批量回收；用户态经 poll/epoll 事件等待完成，避免忙轮询。

2.4 中断节流与 NVMe 多队列协同

- 高 IO 并发时可启用**中断合并**（interrupt coalescing）按完成量/超时触发，降低 CPU 开销（设计推算：默认合并窗口 32 完成或 50 μs）。
- 与 NVMe 原生多队列协同：每 SSD 命名空间可建多条队列（对齐 NVMe 65,535 队列能力），驱动按 CPU 亲和性分散队列与中断，支撑 Tb/s 级 IO 目标（白皮书第 5 章）。

> **本章结论**：队列层采用 NVMe 风格 SQ/CQ + 门铃/完成语义，为块 IO 与 KV 扩展命令统一服务；中断用 MSI-X 每队列一向量、批量回收 + 中断合并，配合 NVMe 多队列实现高并发低开销提交路径。

第 3 章 DMA 通路与 NVMe / KV-Cache 数据通路

3.1 DMA 描述符与 P2P 通路

KFNative 数据面核心是 **GPU↔SSD 的 P2P DMA**：数据不经主机内存/网卡，完全在 Extended PCIe 内流动（对齐架构文档第 3.3 节微秒级路径）。

描述符机制（设计推算）：支持 PRP（Physical Region Page，对齐 NVMe PRP）与 SGL（Scatter-Gather List）两种形态：

- **PRP**：连续小传输，页粒度（4 KB 对齐），用于 KV 页换入换出；

- **SGL**：非连续大数据传输，用于权重流式加载（GPT-3 175B 单层约 3.6 GB，需多 SGL 段，白皮书 2.4 节示例）。

P2P 实现：驱动经 `pci_p2pdma_add_resource` 将 GPU 的 P2P 内存注册到板卡 DMA 域；固件按描述符直接把 NVMe 数据写到 GPU 显存地址（GPUDirect Storage 思路，经 PCIe 实现，设计推算）。

3.2 DMA 骨架（C）

```
struct kfn_dma_req {
    struct kfn_queue *q;
    dma_addr_t src;                /* 源（SSD 数据面或 GPU P2P 地址） */
    dma_addr_t dst;                /* 目标 */
    u32 len;
    u32 nsid;
    u64 lba;
    int dir;                       /* DMA_FROM_DEVICE / DMA_TO_DEVICE */
    struct completion *done;
};

/* 描述符：PRP/SGL 通用头（设计推算） */
struct kfn_sgl {
    __le64 addr;
    __le32 len;
    __le32 flags;                  /* 含 P2P 标记 */
};

/* 发起一次 SSD→GPU 的流式加载（权重/KV 页） */
int kfn_issue_dma(struct kfn_dev *kf, struct kfn_dma_req *r)
{
    struct kfn_cmd cmd = {0};

    cmd.opcode = (r->dir == DMA_FROM_DEVICE) ? KFN_OP_NVME_READ :
KFN_OP_NVME_WRITE;
    cmd.nsid = r->nsid;
    cmd.lba = r->lba;
    cmd.len = r->len;
    cmd.dma_addr = r->dst;         /* GPU P2P 地址（经 pci_p2pdma 映射） */
    /* KV 场景改走 KFN_OP_KV_GET，dma_addr 为 KV 页地址 */
    kfn_submit(kf, r->q, &cmd);
    return 0;
}

/* 异步完成回调：由第 2 章 irq_handler 触发 */
void kfn_complete_dma(struct kfn_dev *kf, struct kfn_dma_req *r, int status)
{
    if (r->done)
        complete(r->done);
    /* 记录统计：带宽、延迟、重试率（供有效带宽评估） */
}

/* 同步封装（供内核路径简单使用） */
int kfn_dma_sync(struct kfn_dev *kf, struct kfn_dma_req *r, int timeout_ms)
```

```
{
    DECLARE_COMPLETION_ONSTACK(done);
    r->done = &done;
    kfn_issue_dma(kf, r);
    if (!wait_for_completion_timeout(&done, msecs_to_jiffies(timeout_ms)))
        return -ETIMEDOUT;
    return 0;
}
```

3.3 NVMe 数据通路

- **命名空间直通**：板卡固件把每 SSD 命名空间暴露给驱动，驱动侧按 NVMe Admin/IO 命令集下发（协议背景）；NVMe 原生多队列（最多 65,535 队列）由第 2 章队列层承载。
- **CMB（Controller Memory Buffer）**：对齐企业盘 CMB 特性（白皮书 5.1.5），驱动可将 SQ/CQ 或数据暂存放于 CMB，减少 PCIe 往返（设计推算：默认将管理队列放入 CMB）。
- **SR-IOV**：企业盘支持 SR-IOV（白皮书 5.1.5），板卡可为 K8s 多租户暴露 VF，租户直接访问命名空间（对应服务层 K8s CSI，见架构文档 2.3 节）。

3.4 KV-Cache 数据通路

KV-Cache 分页管理在用户态 KV-Cache 管理器（架构文档 2.2 节），驱动提供页级原语：

原语	语义	**DMA** 路径
KV_PUT(page_id, gpu_addr, len)	将 GPU 侧 KV 页写入配对 SSD	GPU(P2P) → Switch → SSD
KV_GET(page_id, gpu_addr, len)	从配对 SSD 读回 KV 页	SSD → Switch → GPU(P2P)
KV_DEL(page_id)	释放 KV 页	仅控制面
KV_SYNC(page_id, peer_id)	相邻 SSD 交换（训练 Ring 中转）	SSD → Switch → 相邻 SSD

流水线重叠：驱动以"异步提交 + 多描述符预取"支撑推理逐层加载与计算重叠（架构文档 3.1 节：加载 35 ms 被 180 ms 计算隐藏，设计推算）；固件支持多命令并行调度，DMA 请求按优先级（KV 热点 > 权重流 > 后台重建）。

本章结论：DMA 层以 PRP/SGL 描述符 + P2P 地址映射支撑 GPU↔SSD 直通；NVMe 通路复用命名空间/多队列/CMB/SR-IOV 生态特性；KV-Cache 通路以页级 PUT/GET/DEL/SYNC 原语实现专有缓存与相邻交换，为流式推理与 Ring 训练提供底层搬运能力。

第 4 章 固件功能模块与代码框架

4.1 固件功能模块划分

板卡固件（运行于 Switch 管理处理器/板载 MCU，设计推算）分为六大模块：

模块	职责
队列仲裁器	轮询 SQ、按优先级调度命令到 NVMe 引擎/KV 引擎
NVMe 引擎	转发 NVMe 命令到目标 SSD，聚合完成
KV 引擎	执行 KV_PUT/GET/DEL/SYNC，维护页表与配对映射
DMA 调度器	分配 PCIe 带宽，执行 PRP/SGL 拆解与 P2P 转发
Fabric 管理代理	拓扑发现、路由表缓存、健康/错误上报（与《Lattice Fabric 互联算法》协同）
电源/时钟管理	低功耗状态机、时钟门控、温度监控

固件-驱动握手协议（设计推算）：驱动 probe 写 `FW_RESET` → 固件执行自检 → 写 `FW_READY` + 版本/能力 → 驱动配置队列 → 建立管理通道（0 号队列）。详见第 1 章 `kfn_fw_reset_and_ready`。

4.2 固件状态机与命令流程（C 骨架，示意运行于板卡侧）

```
/* ---- 板卡固件侧代码（示意） ---- */
#define KFN_FW_CMD_QUEUE_DEPTH 512

struct kfn_fw {
    struct kfn_cmd sq[KFN_FW_CMD_QUEUE_DEPTH];  /* 主机提交队列镜像 */
    struct kfn_cq_entry cq[KFN_FW_CMD_QUEUE_DEPTH];
```

```
    u32 sq_tail;          /* 门铃写指针（主机侧视角） */
    u32 cq_head;          /* 固件完成写指针 */
    u32 cq_tail;          /* 主机已回收指针 */
    struct kfn_fw_ssd ssds[KFN_MAX_SSD];
    struct kfn_fw_kv kv;  /* KV 页表 */
};

enum fw_state { FW_RESET, FW_SELF_TEST, FW_READY, FW_SUSPEND, FW_FAILED };

/* 主循环：收命令→ 仲裁→ 执行→ 完成 */
void kfn_fw_run(struct kfn_fw *fw)
{
    for (;;) {
        u32 new_tail = readl(fw->db_regs + DB_SQ_TAIL);   /* 读门铃 */
        while (fw->sq_tail != new_tail) {
            struct kfn_cmd *c = &fw->sq[fw->sq_tail & (KFN_FW_CMD_QUEUE_DEPTH -
1)];
            if (!(c->flags & KFN_CMD_VALID))
                break;
            switch (c->opcode) {
            case KFN_OP_NVME_READ:
            case KFN_OP_NVME_WRITE:
                kfn_fw_ssd_exec(fw, c);       /* 经NVMe 引擎下发SSD */
                break;
            case KFN_OP_KV_PUT:
            case KFN_OP_KV_GET:
                kfn_fw_kv_exec(fw, c);        /* KV 页表+ DMA */
                break;
            case KFN_OP_KV_SYNC:
                kfn_fw_kv_sync(fw, c);        /* 相邻SSD 交换 */
                break;
            default:
                kfn_fw_admin_exec(fw, c);
                break;
            }
            c->flags &= ~KFN_CMD_VALID;       /* 归还槽位 */
            fw->sq_tail++;
        }
        /* 写完成队列+ 触发MSI-X（聚合） */
        if (kfn_fw_flush_cq(fw))
            kfn_fw_raise_irq(fw);
        kfn_fw_idle_power(fw);                /* 空闲时钟门控 */
    }
}

/* KV 引擎：页表查找+ DMA 调度（示意） */
int kfn_fw_kv_exec(struct kfn_fw *fw, struct kfn_cmd *c)
{
    struct kfn_fw_kv_page *pg = kfn_fw_kv_lookup(fw, c->kv_key);
    if (!pg)
        return kfn_fw_cq_error(fw, c, KFN_ERR_KV_MISS);

    /* 按配对映射定位SSD（设计推算：页表维护key→(ssd, lba)） */
    if (c->opcode == KFN_OP_KV_GET)
        kfn_fw_dma(fw, pg->ssd_id, pg->lba, c->dma_addr, c->len, 0);
    else /* PUT */
```

```
        kfn_fw_dma(fw, pg->ssd_id, pg->lba, c->dma_addr, c->len, 1);
    return 0;
}
```

4.3 健康、错误与固件升级

- **错误上报**：固件经 AER（PCIe 高级错误报告，协议背景）与私有管理消息上报错误分类（可恢复/不可恢复）；驱动侧记录并在 `dmesg` 输出。
- **健康监控**：温度/电压/功耗传感器周期上报 BMC（管理通道，见原理图模块），超阈值触发降速/低功耗状态机。
- **固件升级**：支持分级启动（A/B 分区），驱动经管理通道下发固件镜像，升级失败自动回退（设计推算）。

4.4 驱动-固件接口汇总（寄存器/消息表，设计推算）

接口	方向	用途
BAR0 FW_RESET / FW_READY	驱动→固件 / 固件→驱动	握手
BAR0 能力/版本寄存器	固件→驱动	能力协商
BAR2 门铃（SQ Tail）	驱动→固件	提交命令
BAR2 CQ 头指针	驱动→固件	回收完成
管理队列消息	双向	配置/健康/升级
MSI-X 向量	固件→驱动	完成/事件中断

本章结论：固件以"队列仲裁 + NVMe 引擎 + KV 引擎 + DMA 调度 + Fabric 代理 + 电源管理"六模块实现板卡侧执行；骨架代码覆盖主循环、KV 页表执行与完成/中断流程；配合驱动侧 BAR 映射、队列门铃、MSI-X 中断与 PRP/SGL DMA，构成完整的 KFNative™ 驱动-固件软件栈。