DPDK网卡设备INIT源码解析

一、dpdk的-a/b参数逻辑

补充 rte_bus_register 总链表**rte_bus_list的初始化**:

```
struct rte_pci_bus rte_pci_bus = {
    .bus = {
        .scan = rte_pci_scan, // pci扫描回调函数
         // 扫描 PCI 总线的内容，并为所有已注册的驱动程序调用probe()函数，这些驱动程序在其id_table 中具有已发现设备的匹配条目。如果供应商/设备 ID 匹配，则调用给定设备的所有已注册驱动程序的probe() 函数。 如果初始化失败，则返回 < 0；如果没有找到该设备的驱动程序，则返回 1。
        .probe = pci_probe, // 为pci设备绑定uio驱动时会被调用
        .cleanup = pci_cleanup,
        .find_device = pci_find_device,
        .plug = pci_plug,
        .unplug = pci_unplug,
        .parse = pci_parse, // 处理获取pci地址
        .devargs_parse = rte_pci_devargs_parse,
        .dma_map = pci_dma_map,
        .dma_unmap = pci_dma_unmap,
        .get_iommu_class = rte_pci_get_iommu_class,
        .dev_iterate = rte_pci_dev_iterate,
        .hot_unplug_handler = pci_hot_unplug_handler,
        .sigbus_handler = pci_sigbus_handler,
    },
    .device_list = TAILQ_HEAD_INITIALIZER(rte_pci_bus.device_list), // 初始化device_list的bus链表。
    .driver_list = TAILQ_HEAD_INITIALIZER(rte_pci_bus.driver_list),
};
        |
// 此处注册的(bus).name就是"pci"， 并将 rte_pci_bus.bus(注意：只是bus部分的回调) 插入 rte_bus_list 链表中
RTE_REGISTER_BUS(pci, rte_pci_bus.bus);
        |
rte_bus_register(struct rte_bus *bus)
{
    RTE_VERIFY(bus);
    RTE_VERIFY(rte_bus_name(bus) && strlen(rte_bus_name(bus)));
    /* A bus should mandatorily have the scan implemented */
    RTE_VERIFY(bus->scan);
    RTE_VERIFY(bus->probe);
    RTE_VERIFY(bus->find_device);
    /* Buses supporting driver plug also require unplug. */
    RTE_VERIFY(!bus->plug || bus->unplug);

    TAILQ_INSERT_TAIL(&rte_bus_list, bus, next);
    EAL_LOG(DEBUG, "Registered [%s] bus.", rte_bus_name(bus));
}
```

存储对应的-a/b参数的pci设备:

第一步：将参数信息存储与devopt_list中；

```
// -a 即是只使用指定的pci
#define OPT_DEV_ALLOW          "allow"
    OPT_DEV_ALLOW_NUM       = 'a',
// -b 即是不使用指定的pci
#define OPT_DEV_BLOCK          "block"
    OPT_DEV_BLOCK_NUM      = 'b',


// 链表记录相应的 pci 设备
rte_eal_init() // +1007行  之所以标注行数，是为了与下面的区分前后逻辑
eal_parse_args(int argc, char **argv)
eal_parse_common_option()
    switch (opt) {
    case 'b':
        if (eal_option_device_add(RTE_DEVTYPE_BLOCKED, optarg) < 0) // 所指定的类型
不一样 BLOCKED
            return -1;
        break;

    case 'a':
        if (eal_option_device_add(RTE_DEVTYPE_ALLOWED, optarg) < 0) // 所指定的类型
不一样 ALLOWED
            return -1;
        break;


// 将指定pci设备加入 devopt_list。
static int
eal_option_device_add(enum rte_devtype type, const char *optarg)
{
    struct device_option *devopt;
    size_t optlen;
    int ret;

    optlen = strlen(optarg) + 1;
    devopt = calloc(1, sizeof(*devopt) + optlen);
    if (devopt == NULL) {
        EAL_LOG(ERR, "Unable to allocate device option");
        return -ENOMEM;
    }

    devopt->type = type; // 设置类型
    ret = strlcpy(devopt->arg, optarg, optlen);
    if (ret < 0) {
        EAL_LOG(ERR, "Unable to copy device option");
        free(devopt);
        return -EINVAL;
    }
    TAILQ_INSERT_TAIL(&devopt_list, devopt, next); // 插入链表
    return 0;
}
```

开始对-a、b存储的的pci设备进行处理：

第二步：将devopt_list中存储的pci信息，转存于devargs_list链表下

```
rte_eal_init            // +1028行

# 第二步：
eal_option_device_parse // dpdk参数处理
    RTE_TAILQ_FOREACH_SAFE(devopt, &devopt_list, next, tmp) {
        if (ret == 0) {
            ret = rte_devargs_add(devopt->type, devopt->arg);
            if (ret)
                EAL_LOG(ERR, "Unable to parse device '%s'",
                    devopt->arg);
        }
        TAILQ_REMOVE(&devopt_list, devopt, next);
        free(devopt);
    }
        |
        |
rte_devargs_add
     rte_devargs_parse // 存储相关的 pci 名称等参数信息
        {
            // bus = TAILQ_FIRST(&rte_bus_list);
            // 根据dpdk启动时多遍历的全部 bus 设备，然后根据 dev pci名称，来查询其对应的
bus属性，其中就包括了标识其设备类型的name，比如后面所调用指定的"pci"类型。
            bus = rte_bus_find(bus, bus_name_cmp, dev);
            da->bus = bus;
        }
     TAILQ_INSERT_TAIL(&devargs_list, devargs, next); // 将其存储到 devargs_list 中
```

pci设备扫描注册填充：

第三步：对系统进行整体bus设备扫描，并根据扫描结果，对比判断devargs_list中pci设备是否进行读取填充。

```
## 补充devargs_list的轮询查找流程
// 查询devargs_list链表中所存pci设备（名称 + 类型）。
pci_devargs_lookup(const struct rte_pci_addr *pci_addr)
{
    struct rte_devargs *devargs;
    struct rte_pci_addr addr;

    // 轮询devargs_list的"pci" devargs
    RTE_EAL_DEVARGS_FOREACH("pci", devargs) {
        devargs->bus->parse(devargs->name, &addr);
        if (!rte_pci_addr_cmp(pci_addr, &addr)) // 对比pci总线地址是否一致
            return devargs;
    }
}
        |
// rte_devargs_next的调用是通过宏定义 RTE_EAL_DEVARGS_FOREACH 来操作的
```

```
#define RTE_EAL_DEVARGS_FOREACH(busname, da) \
    for (da = rte_devargs_next(busname, NULL); \
         da != NULL; \
         da = rte_devargs_next(busname, da)) \
        |
// 轮询调用 rte_devargs_next 来获取rte_devargs
struct rte_devargs* rte_devargs_next(const char *busname, const struct
rte_devargs *start)
{
    struct rte_devargs *da;
    if (start != NULL)
        da = TAILQ_NEXT(start, next);
    else
        da = TAILQ_FIRST(&devargs_list); //开始循环处理devargs_list+
    while (da != NULL) {
        if (busname == NULL ||
            (strcmp(busname, da->bus->name) == 0))
            return da;
        da = TAILQ_NEXT(da, next);
    }
    return NULL;
}



#第三步：根据参数-ab的指定进行设备是否读取判断。

rte_eal_init // +1061行
rte_bus_scan()
    // 轮询rte_bus_list链表
    TAILQ_FOREACH(bus, &rte_bus_list, next) {
        ret = bus->scan(); // rte_pci_scan() 的调用
    }
        |
// dpdk在进行设备扫描时，会进行相应的忽略，不再继续 pci_scan_one 扫描填充操作
## 即是上述 rte_pci_bus 中注册的 scan 回调函数。
int rte_pci_scan(void)
{
    ……
    dir = opendir(rte_pci_get_sysfs_path()); // 读取pci设备总路径
    while ((e = readdir(dir)) != NULL) {
         // 判断是否忽略该pci设备
         // -a 返回 false ，继续执行后续 pci_scan_one;
         // -b 默认返回 true,于是忽略该设备
        if (rte_pci_ignore_device(&addr)) // 具体描述如下
            continue;
        if (pci_scan_one(dirname, &addr) < 0) // 对pci地址位addr的设备进行扫描读取填充
相关信息。
            goto error;
    }
    ……
}
        |
// 开始对-a或者-b参数pci设备的判断
bool rte_pci_ignore_device(const struct rte_pci_addr *pci_addr)
```

```
{
    struct rte_devargs *devargs = pci_devargs_lookup(pci_addr);

    switch (rte_pci_bus.bus.conf.scan_mode) {
    case RTE_BUS_SCAN_ALLOWLIST:
        if (devargs && devargs->policy == RTE_DEV_ALLOWED)
            return false;
        break;
    case RTE_BUS_SCAN_UNDEFINED:
    case RTE_BUS_SCAN_BLOCKLIST:
        if (devargs == NULL || devargs->policy != RTE_DEV_BLOCKED)
            return false;
        break;
    }
    return true;
}
```

二、dpdk的rte_eth_rx_burst内核逻辑

```
# 补充
/*
 *与每个 RX 队列相关的结构。
 volatile：当两个线程都要用到某一个变量且该变量的值会被改变时，应该用volatile 声明，该关键字的作用是防止优化编译器把变量从内存装入CPU 寄存器中。
 */
struct i40e_rx_queue {
    struct rte_mempool *mp; /**< 填充 RX 环的 mbuf 池 */
    volatile i40e_rx_desc *rx_ring;/**< RX环虚拟地址*/
    uint64_t rx_ring_phys_addr;  /**< RX环DMA地址 */
    struct i40e_rx_entry *sw_ring; /**< RX软环地址 */
    uint16_t nb_rx_desc;  /**< RX 描述符数量 */
    uint16_t rx_free_thresh;  /**< 可容纳的最大可用 RX 描述 */
    uint16_t rx_tail;  /**< tail 当前值 */
    uint16_t nb_rx_hold;  /**< 持有的空闲 RX 描述的数量 */
    struct rte_mbuf *pkt_first_seg; /**<当前数据包的第一段*/
    struct rte_mbuf *pkt_last_seg; /**<当前数据包的最后一段*/
    struct rte_mbuf fake_mbuf;  /**< 虚拟 mbuf */
#ifdef RTE_LIBRTE_I40E_RX_ALLOW_BULK_ALLOC
    uint16_t rx_nb_avail;  /**< 准备好的分阶段数据包数量 */
    uint16_t rx_next_avail;/**<下一个阶段数据包的索引*/
    uint16_t rx_free_trigger;  /**< 触发 rx 缓冲区分配 */
    structrte_mbuf *rx_stage[RTE_PMD_I40E_RX_MAX_BURST *2];
#endif

    uint16_t rxrearm_nb;     /**< 剩余待重新武装的数量 */
    uint16_t rxrearm_start;  /**<我们开始重新武装的idx */
    uint64_t mbuf_initializer;  /**< 初始化 mbuf 的值 */

    uint16_t port_id;  /**<设备端口ID */
    uint8_t crc_len;  /**< 0 如果 CRC 被剥离，则为 4 */
    uint8_t fdir_enabled;  /**< 0 如果 FDIR 禁用，1 当启用 */
```

```
    uint16_t queue_id;  /**< RX队列索引 */
    uint16_t reg_idx;  /**< RX队列寄存器索引*/
    uint8_t drop_en;  /**<如果不为0，则设置寄存器位*/
    volatile uint8_t *qrx_tail;  /**< 尾部寄存器地址 */
    struct i40e_vsi *vsi; /**< 该队列所属的VSI */
    uint16_t rx_buf_len;  /*数据包缓冲区大小 */
    uint16_t rx_hdr_len;  /*标头缓冲区大小 */
    uint16_t max_pkt_len;  /*最大数据包长度 */
    uint8_t hs_mode;  /*标头分割模式 */
    bool q_set;  /**< 指示rx队列是否已配置 */
    bool rx_deferred_start; /**< 不要在开发启动时启动此队列 */
    uint16_t rx_using_sse;  /**<flag 表示 vPMD 用于 rx */
    uint8_t dcb_tc;          /**< rx 队列的流量类别 */
    uint64_t offloads;  /**< RTE_ETH_RX_OFFLOAD_*的 Rx 卸载标志 */
    const struct rte_memzone *mz;
};
```

2.1 调用逻辑

```
static inline uint16_t rte_eth_rx_burst(uint16_t port_id, uint16_t queue_id,
                            struct rte_mbuf **rx_pkts, const uint16_t nb_pkts)
{
    // 获取指向队列数据的指针
    p = &rte_eth_fp_ops[port_id];
    qd = p->rxq.data[queue_id];

    nb_rx = p->rx_pkt_burst(qd, rx_pkts, nb_pkts);
}

### 不同的网卡，注册的rx回调函数不一样，我们还是以i40e网卡为例：
// dpdk一般会以"卡类型_set_rx_function"命名对应网卡的收包设置函数。
i40e_set_rx_function
{
    dev->rx_pkt_burst = i40e_recv_scattered_pkts_vec;
    dev->rx_pkt_burst = i40e_recv_pkts_vec;
    dev->rx_pkt_burst = i40e_recv_pkts_bulk_alloc;
    dev->rx_pkt_burst = dev->data->scattered_rx ? i40e_recv_scattered_pkts :
i40e_recv_pkts;
}

// 其中就包括了相应的矢量收包函数，此处我们以基础收包函数 40e_recv_pkts_bulk_alloc 为例说明
static uint16_t i40e_recv_pkts_bulk_alloc(void *rx_queue,
              struct rte_mbuf **rx_pkts,
              uint16_t nb_pkts)
{
    uint16_t nb_rx = 0, n, count;
    if (unlikely(nb_pkts == 0))
        return 0;

     // 如果我们配置的收包数是小于32.
    if (likely(nb_pkts <= RTE_PMD_I40E_RX_MAX_BURST))
        return rx_recv_pkts(rx_queue, rx_pkts, nb_pkts);

    // 默认配置burst收包数为32个，如果超过，则循环收包
    while (nb_pkts) {
```

```
        n = RTE_MIN(nb_pkts, RTE_PMD_I40E_RX_MAX_BURST); //一次只收32个包
        count = rx_recv_pkts(rx_queue, &rx_pkts[nb_rx], n);
        nb_rx = (uint16_t)(nb_rx + count);
        nb_pkts = (uint16_t)(nb_pkts - count);
        if (count < n)
            break;
    }
    return nb_rx;
}

// rx_recv_pkts收包处理
// rxq指向网卡的收包队列
static inline uint16_t
rx_recv_pkts(void *rx_queue, struct rte_mbuf **rx_pkts, uint16_t nb_pkts)
{
    # 第一步 获取环形队列中的数据包地址，并赋值rxq.
    /* // 将收包地址，指向网卡收包队列的环形队列中。// 环形收包队列由
i40e_dev_rx_queue_setup 建立。
        rxq = x_queue;
        rxep = &rxq->sw_ring[rxq->rx_tail];

        for (j = 0; j < I40E_LOOK_AHEAD; j++) // I40E_LOOK_AHEAD可以查看下面备注描述
            rxq->rx_stage[i + j] = rxep[j].mbuf;
    */
    nb_rx = (uint16_t)i40e_rx_scan_hw_ring(rxq);

    # 第二步 将rxq队列收包地址赋值给用户传递进来的二级指针**rx_pkts
    // 开始填充返回值(收包填充)。
    i40e_rx_fill_from_stage(rxq, rx_pkts, nb_pkts);
    {
        struct rte_mbuf **stage = &rxq->rx_stage[rxq->rx_next_avail];
        for (i = 0; i < nb_pkts; i++)
            rx_pkts[i] = stage[i];
        rxq->rx_next_avail = (uint16_t)(rxq->rx_next_avail + nb_pkts); // 记录下次
收包循环ring位置
    }
}
```

2.2 I40E_LOOK_AHEAD

备注

在DPDK的Intel 40 Gigabit Ethernet驱动程序（i40e）中，I40E_LOOK_AHEAD是一个用于配置收包处理的选项，其作用是控制驱动程序在接收数据包时是否进行提前预取（look ahead）。

预取是一种优化技术，用于在处理器需要访问内存中的数据时，提前将数据加载到处理器的缓存中。在网络数据包处理中，如果驱动程序能够预取将要处理的数据包，可以有效地减少等待数据包到达处理器的延迟，从而提高系统的处理性能。

I40E_LOOK_AHEAD选项允许用户配置i40e驱动程序是否在接收数据包时执行预取操作。通常，预取会导致更多的内存访问和处理器资源消耗，因此在某些情况下可能会影响性能。通过配置I40E_LOOK_AHEAD选项，用户可以根据应用程序的需求和系统的实际情况来平衡性能和资源消耗。

具体来说，I40E_LOOK_AHEAD选项的值可以是以下之一：

I40E_LOOKAHEAD_ON：启用预取功能，驱动程序将在接收数据包时执行预取操作。

I40E_LOOKAHEAD_OFF：禁用预取功能，驱动程序将不会执行预取操作。

根据具体的网络应用和系统配置，用户可以根据性能需求选择适当的选项来配置I40E_LOOK_AHEAD。

2.3 i40e_dev_rx_queue_setup

补充说明：i40e_dev_rx_queue_setup i40e_dev_tx_queue_setup

```
// 在这系列函数下，都会建立对应网卡的收包环形队列，例如：
 /* 分配建立 i40e_rx_queue 结构 Allocate the rx queue data structure */
rxq = rte_zmalloc_socket("i40e rx queue",
              sizeof(struct i40e_rx_queue),
              RTE_CACHE_LINE_SIZE,
              socket_id);
/* 分配建立收包队列 Allocate the software ring. */
len = (uint16_t)(nb_desc + RTE_PMD_I40E_RX_MAX_BURST);
rxq->sw_ring =
    rte_zmalloc_socket("i40e rx sw ring",
                sizeof(struct i40e_rx_entry) * len,
                RTE_CACHE_LINE_SIZE,
                socket_id);
```

这样的话，后续网卡的收包将存储于其对应建立的环形队列中，并且由对应的收包函数，从中获取数据包地址（而不是拷贝）。

备注：i40e_dev_rx_queue_setup 函数的调用，实际是存于 staic const struct eth_dev_ops i40e_eth_dev_ops 其中的默认回调 rx_queue_setup 函数。

其在设备启动初始化时：// 具体的调用逻辑可以查看三中的描述。

```
eth_i40e_dev_init()
    dev->dev_ops = &i40e_eth_dev_ops; // 配置到了设备属性中
```

所以，如果我们需要对网卡就行自适应的修改和调整，我们可以使用

```
    rte_eth_dev_configure();
    rte_eth_rx_queue_setup();
    rte_eth_dev_start();
```

此系类函数，对相关的rx属性进行主动修改，但如果你进一步查看 函数，你会发现，其内部也会调用

```
    ret = (*dev->dev_ops->rx_queue_setup)(dev, rx_queue_id, nb_rx_desc,
                          socket_id, &local_conf, mp);
```

只不过其对应的参数都变成了用户所自行配置的结果。

2.4 RTE_PMD_I40E_RX_MAX_BURST

注意：

1、在DPDK中，收包环形队列（也称为Ring队列）不会自动销毁收到的数据包。相反，环形队列中的数据包由应用程序在处理完毕后手动进行销毁或重用。

通常情况下，当应用程序从环形队列中取出数据包进行处理后，可以选择性地将数据包销毁或者重新使用。这取决于应用程序的需求和设计。

销毁数据包：应用程序在处理完数据包后，可以调用相关函数来释放该数据包所占用的内存。通常，DPDK提供了函数或者接口来进行数据包的销毁操作，比如rte_pktmbuf_free()函数用于释放数据包占用的内存资源。

```
rte_pktmbuf_free(pkt);
```

重用数据包：在某些情况下，应用程序可能需要将数据包重新放入到环形队列中，以便后续的处理或传输。在这种情况下，应用程序可以直接将数据包重新放入到环形队列中，而无需销毁它。

```
rte_ring_enqueue(ring, pkt);
```

总的来说，DPDK并不会自动管理数据包的生命周期，而是由应用程序负责在适当的时机对数据包进行销毁或者重用。这样的设计允许应用程序对数据包的处理过程进行灵活的控制，以满足不同场景下的性能和功能需求。

2、如果收包环形队列中的数据包没有及时被处理和销毁，就会导致环形队列溢出，进而导致丢包。

```
// 在上述距离的i40e中，环形队列大小就是 len 的大小。
```

收包环形队列的大小是有限的，一旦队列满了，新到达的数据包就无法放入队列中，从而导致丢包。因此，及时处理和释放环形队列中的数据包是至关重要的。

为了避免丢包，应用程序通常需要根据系统的处理能力和环境负载情况来调整环形队列的大小，以确保队列足够大，能够容纳预期的数据包数量，并且及时地处理队列中的数据包。

另外，对于高速网络环境下的数据包处理，及时的数据包处理和释放也是确保系统性能的关键因素之一。因此，对于DPDK应用程序来说，合理管理环形队列中的数据包是非常重要的一部分。

3、RTE_PMD_I40E_RX_MAX_BURST 是一个 DPDK 中的宏定义，用于配置 Intel 40 Gigabit Ethernet 驱动程序（i40e）在一次 RX 操作中最多处理的数据包数量。它的作用是限制在一次收包操作中从网络适配器接收的数据包数量。

在 DPDK 中，为了提高数据包处理的效率，通常会使用批处理技术来一次性处理多个数据包，而不是逐个处理。RTE_PMD_I40E_RX_MAX_BURST 宏定义的值可以用来指定 i40e 驱动程序在一次 RX 操作中最多处理的数据包数量。它决定了每次从网络适配器接收数据包时一次性处理的最大数据包数量。

默认情况下，RTE_PMD_I40E_RX_MAX_BURST 的值通常是较小的，以避免过多的数据包聚集在单个处理周期内，导致延迟增加。通常，这个值会根据系统的处理能力、应用程序的需求以及网络负载等因素进行调整。

通过调整 RTE_PMD_I40E_RX_MAX_BURST 的值，可以优化收包操作的性能和效率，以适应不同的网络环境和应用场景。增大该值可以提高数据包处理的吞吐量，但同时也会增加处理延迟。

三、关联设备注册与读取回调

第一步： i40e网卡设备注册（rte_pci_bus.driver_list）

```
// 以下还是以i40e网卡为例子
// 让我们思考一个问题，那些pci网卡（绑定的网卡设备）是在什么时候进行的相关（以二中rx的收包属性为例）设置？
/** Helper for PCI device registration from driver (eth, crypto) instance */
#define RTE_PMD_REGISTER_PCI(nm, pci_drv) \
RTE_INIT(pciinitfn_ ##nm) \
{\
    (pci_drv).driver.name = RTE_STR(nm);\
```

```
    rte_pci_register(&pci_drv); \
} \

// i40e网卡默认的相关配置
static struct rte_pci_driver rte_i40e_pmd = {
    .id_table = pci_id_i40e_map,
    .drv_flags = RTE_PCI_DRV_NEED_MAPPING | RTE_PCI_DRV_INTR_LSC,
    .probe = eth_i40e_pci_probe, // 其中就时i40e网卡初始化、属性配置的关键。
eth_i40e_dev_init 也在其中。
    .remove = eth_i40e_pci_remove,
};
#### 注意：上面的函数就是在后续的函数scan中将会进行各自调用的 probe 回调，以及 remove 回调。

// 注册i40e网卡的pci模块。
RTE_PMD_REGISTER_PCI(net_i40e, rte_i40e_pmd);
    rte_pci_register();

// 这个链表很熟悉，其实就是 一 中的全局变量：struct rte_pci_bus rte_pci_bus。
// 它对list进行了初始化，并且保证后续设备init时，将相关的注册加入到该链表中
void rte_pci_register(struct rte_pci_driver *driver)
{
    TAILQ_INSERT_TAIL(&rte_pci_bus.driver_list, driver, next);
}

## 对于 .probe 函数作用的大致描述：
/*
在 DPDK 中，.probe 回调函数是一种设备驱动程序的注册方式，用于初始化和注册设备。它是一个在
DPDK 初始化过程中调用的回调函数，用于发现和识别物理设备，并初始化 DPDK 库中的设备抽象结
构。.probe 回调函数的作用主要包括以下几个方面：
 1、设备发现和识别：
    在初始化过程中，DPDK 驱动程序会调用注册的 .probe 回调函数，用于发现和识别物理设备。
    通过访问设备的寄存器或者其他方式，驱动程序可以识别连接到系统的物理设备，并确定其类型和配置。
2、设备初始化：
    一旦设备被发现和识别，.probe 回调函数还负责初始化设备并为其分配资源。
    这可能包括初始化设备的硬件寄存器、分配内存、配置中断等操作，以便设备可以正常工作。
3、注册设备：
    在设备被发现、识别和初始化后，.probe 回调函数通常会将设备注册到 DPDK 库中，以便后续的操作
可以通过 DPDK API 来访问设备。
    通过注册设备，应用程序可以使用 DPDK 提供的各种功能和接口来操作设备，例如接收和发送数据包、
配置设备参数等。
.probe 回调函数通常是在 DPDK 初始化阶段中调用的，用于初始化和注册设备。它是设备驱动程序的重要
部分，负责管理设备的发现、识别和初始化过程，以及将设备注册到 DPDK 库中，使其可以被应用程序使用。
*/
```

第二步： 网卡probe初始化函数的调用：pci_probe_all_drivers

```
// 我们可以根据上述的 rte_pci_bus.driver_list 来查看调用位置：
#define FOREACH_DRIVER_ON_PCIBUS(p) \
    RTE_TAILQ_FOREACH(p, &(rte_pci_bus.driver_list), next)
// 初始化全部的pci设备。
static int pci_probe_all_drivers(struct rte_pci_device *dev)
{
    struct rte_pci_driver *dr = NULL;
    FOREACH_DRIVER_ON_PCIBUS(dr) {
        rc = rte_pci_probe_one_driver(dr, dev); // 初始化位置
```

```
        if (rc < 0)
            /* negative value is an error */
            return rc;
        if (rc > 0)
            /* positive value means driver doesn't support it */
            continue;
        return 0;
    }
    return 1;
}

// 而在对应的 rte_pci_probe_one_driver 函数中：
static int rte_pci_probe_one_driver(struct rte_pci_driver *dr, struct
rte_pci_device *dev)
{
    ..................
    /* call the driver probe() function */
    ret = dr->probe(dr, dev); // 本次 eth_i40e_pci_probe 函数的调用。
    ..................
}
```

```
## 注意：
    pci_probe_all_drivers 函数的调用，其实是 struct rte_pci_bus rte_pci_bus 的
bus.probe == rte_pci_scan 回调函数来调用。
```

第三步： pci_probe_all_drivers的调用时机

```
rte_eal_init() // +1281行
int rte_bus_probe(void)
{
    TAILQ_FOREACH(bus, &rte_bus_list, next) {
        if (!strcmp(rte_bus_name(bus), "vdev")) {
            vbus = bus;
            continue;
        }
        ret = bus->probe(); // bus回调函数调用位置
        if (ret)
            EAL_LOG(ERR, "Bus (%s) probe failed.",
                rte_bus_name(bus));
    }
    return 0;
}


/* 总体的逻辑就是：
    1、各类不同 bus 总线类型的设备进行 RTE_REGISTER_BUS 注册，然后存储于 rte_bus_list
中；
    2、然后由 rte_bus_probe 调用并轮询bus设备。
    3、各种类(以pci为例)的bus设备，开始调用自己的 bus.probe 回调函数，来轮询
rte_pci_bus.driver_list 链表。
    4、在轮询条件在，系统bus种类下的各个设备开始调用 probe 回调函数，用以进行相应的设备初始
化、rx等属性配置等。
```

四、网卡设备处理流程总结

```
// 我们总结一下上述的调用时机，以 rte_eal_init 为时间尺。
rte_eal_init()
    eal_parse_args // 1007行： 对与pci设备相关的参数进行处理，存于 devopt_list
    eal_option_device_parse // 1028行： devopt_list 转换为 devargs_list
    rte_bus_scan // 1061行： 扫描全局bus设备，关联devargs_list，对其中需要“特殊处理”的pci设备进行指定操作。
    rte_bus_probe // 1281行： 开始对剩余设备进行初始化处理。
```