DPDK线程创建源码梳理

1、线程类型

```
// 参照《dpdk文档略读》笔记
// 需要注意的是，DPDK 应用程序需要使用 -pthread 编译选项，以启用线程支持。

28、 dpdk中的“非eal线程”
其中，non-EAL pthread 有两类
    1、使用 rte_thread_register 调用注册，从而具有有效socketid的non-EAL pthread；
    2、未进行注册过， 属性为 LCORE_ID_ANY 的non-EAL pthread。
这类生成的线程，某些库将使用替代的唯一 ID（例如 TID），有些库不会受到影响，有些库可以工作但有限制（例如计时器和内存池库）。
    // 注意：：
    3、在实际应用的过程中：非EAL线程虽然可以和EAL线程一样，拥有隶属于自己的“线程缓存”，可以达到很好的效率。但是，它不提供“主 副线程”这样的操作模式，当你对非eal线程使用“主线程”所创建的线程池、队列等，副线程是没用权限取到的。

补充：
以下时dpdk中，各种线程类型的状态：
/**
 * The lcore role (used in RTE or not).
 */
enum rte_lcore_role_t {
    ROLE_RTE, // 对应 rte_eal_remote_launch
    ROLE_OFF, // 默认赋值
    ROLE_SERVICE, // 对应 七 中所创建的服务线程
    ROLE_NON_EAL, // 对应 rte_thread_create
};

##注意：rte_eal_remote_launch ()和 rte_thread_create ()都是DPDK中用于创建线程的函数，但它们有不同的用途和行为：
  rte_eal_remote_launch()：
    这个函数用于在DPDK环境中启动一个EAL线程。
    它启动的线程将在DPDK的EAL环境中执行，因此可以使用DPDK提供的各种API和功能。
    通常情况下，用于创建DPDK中的工作线程，执行数据包处理、网络协议栈等任务。
  rte_thread_create()：
    这个函数用于在普通的非-EAL环境下创建线程，它并不需要DPDK的初始化或者EAL环境。
    它创建的线程是普通的POSIX线程，可以使用标准的POSIX线程API进行操作。
    可以在DPDK外部的应用中使用，用于一般的多线程应用程序，而不依赖于DPDK的特定功能。
  总的来说，rte_eal_remote_launch()用于创建DPDK环境中的EAL线程，而rte_thread_create()则用于在非DPDK环境中创建普通的线程。


// 对于我们常见的dpdk启动程序中的"lcores"参数，其指定的lcore参数用于配置用于数据包处理的逻辑核心（Logical Cores）。它默认的创建的线程状态都是ROLE_RTE；
case OPT_LCORES_NUM:
        if (eal_parse_lcores(optarg) < 0) {
            EAL_LOG(ERR, "invalid parameter for --"
                OPT_LCORES);
            return -1;
        }
```

```
        |
        |
static int eal_parse_lcores(const char *lcores)
{
    // 参数字符串处理过后的lcore --- idx.
    for (idx = 0; idx < RTE_MAX_LCORE; idx++) {
            if (!CPU_ISSET(idx, &lcore_set))
                continue;
            set_count--;

            if (cfg->lcore_role[idx] != ROLE_RTE) {
                lcore_config[idx].core_index = count;
                cfg->lcore_role[idx] = ROLE_RTE; // 默认都设置为ROLE_RTE
                count++;
            }
            rte_memcpy(&lcore_config[idx].cpuset, &cpuset,
                    sizeof(rte_cpuset_t));
        }
}

// 而对于ROLE_NON_EAL，除 ROLE_RTE 与 ROLE_SERVICE 之外， 只要是ROLE_OFF(默认赋值)，则会将其类型设置为 ROLE_NON_EAL。
rte_thread_register() // 专用于创建NO_EAL线程。
        |
        |
eal_lcore_non_eal_allocate(void)
{
    struct rte_config *cfg = rte_eal_get_configuration();
    struct lcore_callback *callback;
    struct lcore_callback *prev;
    unsigned int lcore_id;

    rte_rwlock_write_lock(&lcore_lock);
    for (lcore_id = 0; lcore_id < RTE_MAX_LCORE; lcore_id++) {
        if (cfg->lcore_role[lcore_id] != ROLE_OFF)
            continue;
        cfg->lcore_role[lcore_id] = ROLE_NON_EAL;
        cfg->lcore_count++;
        break;
    }
    ..................
}
```

2、rte_eal_remote_launch 线程创建

2.1 线程创建

```
int rte_eal_remote_launch(lcore_function_t *f, void *arg, unsigned int
worker_id)
{
    int rc = -EBUSY;
    /* Check if the worker is in 'WAIT' state. Use acquire order
```

```
     * since 'state' variable is used as the guard variable.
     */
    if (rte_atomic_load_explicit(&lcore_config[worker_id].state,
            rte_memory_order_acquire) != WAIT)
        goto finish;

    // 保存线程所需参数。
    lcore_config[worker_id].arg = arg;
    // 确保在工作线程开始运行该函数之前完成所有内存操作。
    // 将工作线程函数f，原子存储到指定的原子变量中。
    rte_atomic_store_explicit(&lcore_config[worker_id].f, f,
rte_memory_order_release);
    // 开始唤醒工作线程
    rc = eal_thread_wake_worker(worker_id);

finish:
    rte_eal_trace_thread_remote_launch(f, arg, worker_id, rc);
    return rc;
}



## eal_thread_wake_worker 唤醒操作
// 如果（M-W）线程间的通信管道已经建立完毕，则可以唤醒该进程。反之，其将一直卡在下面的两处while里或返回错误。
{
    int m2w = lcore_config[worker_id].pipe_main2worker[1];
    int w2m = lcore_config[worker_id].pipe_worker2main[0];
    char c = 0;
    int n;

    /* 以下两个操作是为了
        1、确保rte_eal_init主进程的管道已经创建完毕;
        2、管道可以正常传递信息。
    */
    do {
        n = _write(m2w, &c, 1); // 向管道写入信息
    } while (n == 0 || (n < 0 && errno == EINTR));
    if (n < 0)
        return -EPIPE;

    do {
        n = _read(w2m, &c, 1); // 等待读取管道信息
    } while (n < 0 && errno == EINTR);
    if (n <= 0)
        return -EPIPE;
    return 0;
}
```

2.2 线程启动（eal_worker_thread_create)

```
rte_eal_init() // 1236行
        |
RTE_LCORE_FOREACH_WORKER(i) {
        /*
         * create communication pipes between main thread and children
         */
        // 创建线程交互管道。
        if (pipe(lcore_config[i].pipe_main2worker) < 0) // main线程 发送管道信息给工作线程 的管道
            rte_panic("Cannot create pipe\n");
        if (pipe(lcore_config[i].pipe_worker2main) < 0) // main线程 从 工作线程 接受管道信息 的管道
            rte_panic("Cannot create pipe\n");

        lcore_config[i].state = WAIT;

        /* create a thread for each lcore */
        ret = eal_worker_thread_create(i); // 开始创建指定core的工作线程
        if (ret != 0)
            rte_panic("Cannot create thread\n");

        /* Set thread_name for aid in debugging. */
        snprintf(thread_name, sizeof(thread_name),
            "dpdk-worker%d", i);
        rte_thread_set_name(lcore_config[i].thread_id, thread_name); // 设置线程名称

        ret = rte_thread_set_affinity_by_id(lcore_config[i].thread_id,  // // 设置线程亲和性
            &lcore_config[i].cpuset);
        if (ret != 0)
            rte_panic("Cannot set affinity\n");
    }


// 开始线程初始化设置
static int eal_worker_thread_create(unsigned int lcore_id)
{
    pthread_attr_t *attrp = NULL;
    void *stack_ptr = NULL;
    pthread_attr_t attr;
    size_t stack_size;
    int ret = -1;

    // 获取配置的“大页栈空间大小”。
    stack_size = eal_get_internal_configuration()->huge_worker_stack_size;
    if (stack_size != 0) {
        // ## 注意：
        // ##   此处的stack大小分配是根据lcore_id所对应NUMA节点上申请的，并设置 pthread 属性
        stack_ptr = rte_zmalloc_socket("lcore_stack", stack_size,
            RTE_CACHE_LINE_SIZE, rte_lcore_to_socket_id(lcore_id));
        if (stack_ptr == NULL) {
```

```
            rte_eal_init_alert("Cannot allocate worker lcore stack memory");
            rte_errno = ENOMEM;
            goto out;
        }

         // 线程属性初始化
        if (pthread_attr_init(&attr) != 0) {
            rte_eal_init_alert("Cannot init pthread attributes");
            rte_errno = EFAULT;
            goto out;
        }
        attrp = &attr;

         // 为每个线程创建其相应的 stack_size 栈空间大小。
         // 具体的大小配置，可以查看 eal_parse_huge_worker_stack 函数（可以参数命令配
置）。
        if (pthread_attr_setstack(attrp, stack_ptr, stack_size) != 0) {
            rte_eal_init_alert("Cannot set pthread stack attributes");
            rte_errno = EFAULT;
            goto out;
        }
    }

    // 创建线程。并将所需配置的属性attrp进行设置。
    if (pthread_create((pthread_t *)&lcore_config[lcore_id].thread_id.opaque_id,
            attrp, eal_worker_thread_loop, (void *)(uintptr_t)lcore_id) == 0)
        ret = 0;

out:
    if (ret != 0)
        rte_free(stack_ptr);
    if (attrp != NULL)
        pthread_attr_destroy(attrp);
    return ret;
}

// 我需要再次提醒一遍，对于上面创建的线程，dpdk会在 eal_parse_lcores 的处理过程中就将指定逻
辑核上的线程设置为 ROLE_RTE 类型。
```

3、rte_thread_register 线程创建

3.1 线程配置

```
int rte_thread_register(void)
{
    unsigned int lcore_id;
    rte_cpuset_t cpuset;

    /* EAL init flushes all lcores, we can't register before. */
    if (eal_get_internal_configuration()->init_complete != 1) {
        EAL_LOG(DEBUG, "Called %s before EAL init.", __func__);
        rte_errno = EINVAL;
```

```
        return -1;
    }
    // 对于NO-EAL线程，默认关闭不支持多线程操作。
    if (!rte_mp_disable()) {
        EAL_LOG(ERR, "Multiprocess in use, registering non-EAL threads is not
supported.");
        rte_errno = EINVAL;
        return -1;
    }
     // 设置线程亲和性
    if (rte_thread_get_affinity_by_id(rte_thread_self(), &cpuset) != 0)
        CPU_ZERO(&cpuset);
     // 为线程分配lcore_id
    lcore_id = eal_lcore_non_eal_allocate();
    if (lcore_id >= RTE_MAX_LCORE)
        lcore_id = LCORE_ID_ANY; // 如果lcore_id分配失败，则默认使用LCORE_ID_ANY，需要
注意的是，如果线程设置为LCORE_ID_ANY，那么其将可能无法使用dpdk中为每个lcore所设置的缓存空间，
导致线程（以抓包线程为例）性能下降。
    // 设置线程亲和性
    __rte_thread_init(lcore_id, &cpuset);
    if (lcore_id == LCORE_ID_ANY) {
        rte_errno = ENOMEM;
        return -1;
    }
    EAL_LOG(DEBUG, "Registered non-EAL thread as lcore %u.",
        lcore_id);
    return 0;
}
```

3.2 线程启动

```
// 示例
static uint32_t thread_loop(void *arg)
{
    struct thread_context *t = arg;
    unsigned int lcore_id;

    lcore_id = rte_lcore_id();
    if (lcore_id != LCORE_ID_ANY) {
        printf("Error: incorrect lcore id for new thread %u\n", lcore_id);
        t->state = Thread_ERROR;
    }

    // 只需要在指定线程的执行函数中调用 rte_thread_register 即可将函数注册为NO-EAL线程。
    if (rte_thread_register() < 0)
        printf("Warning: could not register new thread (this might be expected
during this test), reason %s\n",
            rte_strerror(rte_errno));
    lcore_id = rte_lcore_id();
    if ((t->lcore_id_any && lcore_id != LCORE_ID_ANY) ||
            (!t->lcore_id_any && lcore_id == LCORE_ID_ANY)) {
        printf("Error: could not register new thread, got %u while %sexpecting
%u\n",
            lcore_id, t->lcore_id_any ? "" : "not ", LCORE_ID_ANY);
```

```
        t->state = Thread_ERROR;
    }
    ……
}
```