DPDK关键技术

赵剑川　2014.11.26

概要

DPDK是 INTEL提供的提升数据面报文快速处理速率的应用程序开发包，它主要利用以下几个方面的支持特点来优化报文处理过程，从而加快报文处理速率：

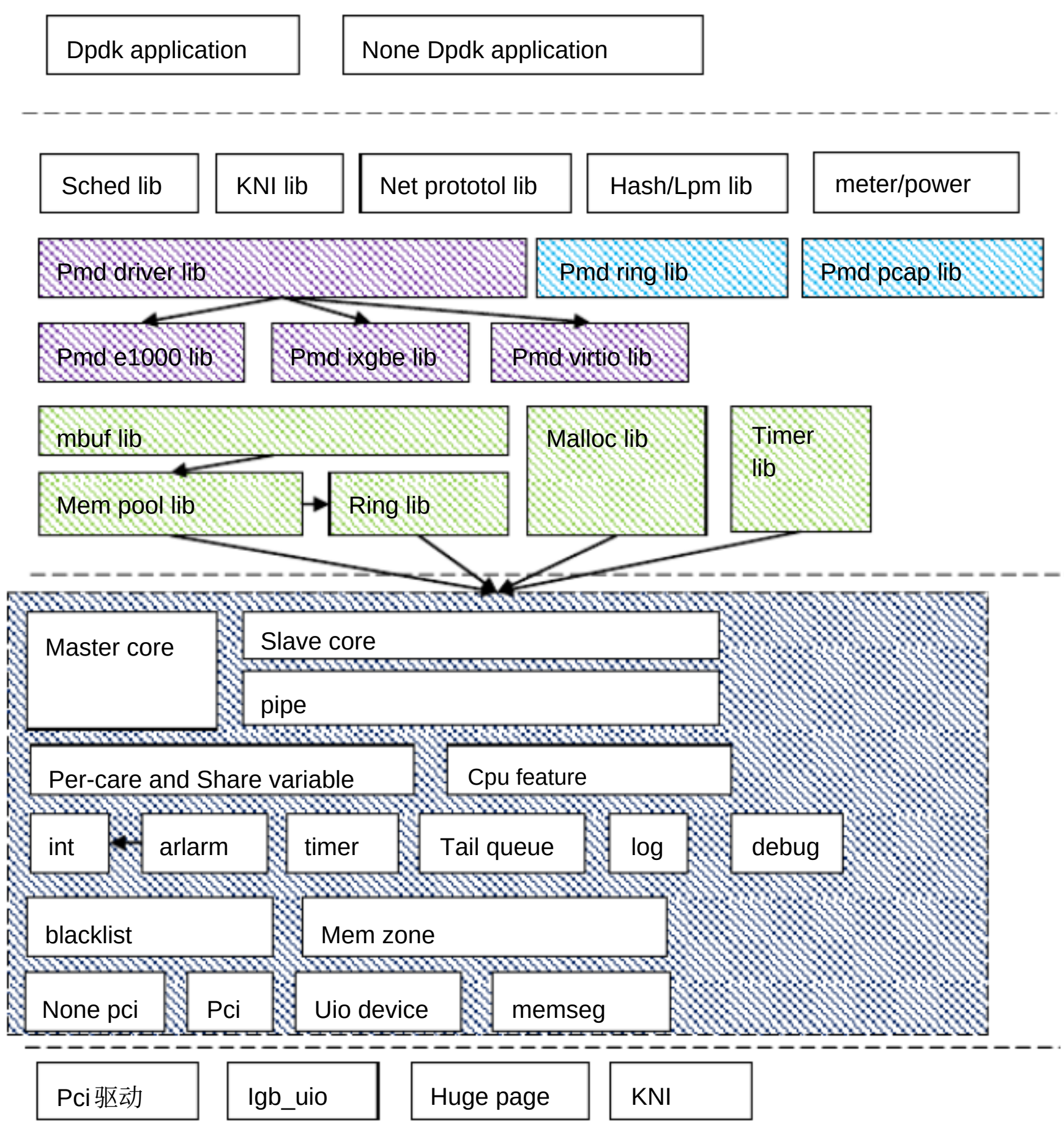
1、 使用大页缓存支持来提高内存访问效率。

2、 利用 UIO支持， 提供应用空间下驱动程序的支持， 也就是说网卡驱动是运行在用户空间的，减下了报文在用户空间和应用空间的多次拷贝。

3、 利用 LINUX亲和性支持， 把控制面线程及各个数据面线程绑定到不同的 CPU核，节省了线程在各个 CPU核来回调度。

4、 提供内存池和无锁环形缓存管理，加快内存访问效率。

整个 DPDK系统由许多不同组件组成，各组件为应用程序和其它组件提供调用接口，其结构图如下图所示：



Dpdk 各系统组件结构图

1、 环境抽象层： 为 DPDK其它组件和应用程序提供一个屏蔽具体平台特性的统一接口, EAL 提供的功能主要有： DPDK 加载和启动；支持多核或多线程执行类型； CPU 核亲和性处理；原子操作和锁操作接口；时钟参考； PCI 总线访问接口；跟踪和调试接口； CPU 特性采集接口；中断和告警接口等。

2、 堆内存管理组件（ Malloc lib）：堆内存管理组件为应用程序提供从大页内存分配堆内存的接口。当需要分配大量内存小块时（如用于存储列表中每个表项指针的内存） ，使用这些接口可以减少 TLB缺页。

3、 环缓冲区管理组件： 环缓冲区管理组件为应用程序和其它组件提供一个无锁的多生产者多消费者 FIFO队列 API。

4、 内存池管理组件：为应用程序和其它组件提供分配内存池的接口，内存池是一个由固定大小的多个内存块组成的内存容器，可用于存储相同对像实体，如报文缓存块等。内存池由内存池的名称（一个字符串）来唯一标识，它由一个环缓中区和一组核本地缓存队列组成，每个核从自已的缓存队列分配内存块，当本地缓存队列减少到一定程度时，从内存环缓冲区中申请内存块来补充本地队列。

5、 网络报文缓存块管理组件：提供应用程序创建和释放用于存储报文信息的缓存块的接口，这些 MBUF 存储在一内存池中。提供两种类型的 MBUF，一种用于存储一般信息，一种用于存储报文数据。

6、 定时器组件：提供一些异步周期执行的接口（也可以只执行一次） ，可以指定某个函数在规定的时间异步的执行，就像 LIBC中的 timer 定时器，但是这里的定时器需要应用程序在主循环中周期调用 rte_timer_manage 来使定时器得到执行，使用起来没有那么方便。定时器组件的时间参考来自 EAL层提供的时间接口。

除了以上六个核心组件外， DPDK还提供以下功能：

1、 以太网轮询模式驱动（ PMD）架构：把以太网驱动从内核移到应用层，采用同步轮询机制而不是内核态的异步中断机制来提高报文的接收和发送效率。

2、 报文转发算法支持： Hash 库和 LPM 库为报文转发算法提供支持。

3、 网络协议定义和相关宏定义： 基于 FreeBSD IP 协议栈的相关定义如： TCP、UDP 、SCTP 等协议头定义。

4、 报文 QOS 调度库：支持随机早检测、流量整形、严格优先级和加权随机循环优先级调度等相关 QOS 功能。

5、 内核网络接口库（ KNI ）：提供一种 DPDK 应用程序与内核协议栈的通信的方法，类似普通 LINUX 的 TUN/TAP 接口，但比 TUN/TAP 接口效率高。每个物理网口可以虚拟出多个 KNI 接口。

以下分章节对各个组件单元进行详细分析。

日志系统篇：

1、 全局日志变量 rte_logs

```
struct rte_logs rte_logs = {
        .type = ~0,
        .level = RTE_LOG_DEBUG,
        .file = NULL,
};
```

该变量用于存储日志文件的的 FILE指针、日志打印级别、要记录的日志类型。

2、 日志类型：

```
/*  系统内部日志类型    */
#define RTE_LOGTYPE_EAL      0x00000001 /**< Log related to eal. */
#define RTE_LOGTYPE_MALLOC 0x00000002 /**< Log related to malloc. */
#define RTE_LOGTYPE_RING     0x00000004 /**< Log related to ring. */
#define RTE_LOGTYPE_MEMPOOL 0x00000008 /**< Log related to mempool. */
#define RTE_LOGTYPE_TIMER   0x00000010 /**< Log related to timers. */
#define RTE_LOGTYPE_PMD       0x00000020 /**< Log related to poll mode driver. */
#define RTE_LOGTYPE_HASH     0x00000040 /**< Log related to hash table. */
```

#define RTE_LOGTYPE_LPM　　0x00000080 /**< Log related to LPM. */
#define RTE_LOGTYPE_KNI　　0x00000100 /**< Log related to KNI. */
#define RTE_LOGTYPE_ACL　　0x00000200 /**< Log related to ACL. */
#define RTE_LOGTYPE_POWER　0x00000400 /**< Log related to power. */
#define RTE_LOGTYPE_METER　0x00000800 /**< Log related to QoS meter. */
#define RTE_LOGTYPE_SCHED 0x00001000　/**<　Log related　to　QoS port　scheduler.
*/
/*　用户可自定义的日志类型　　*/
#define RTE_LOGTYPE_USER1　0x01000000 /**< User-defined log type 1. */
#define RTE_LOGTYPE_USER2　0x02000000 /**< User-defined log type 2. */
#define RTE_LOGTYPE_USER3　0x04000000 /**< User-defined log type 3. */
#define RTE_LOGTYPE_USER4　0x08000000 /**< User-defined log type 4. */
#define RTE_LOGTYPE_USER5　0x10000000 /**< User-defined log type 5. */
#define RTE_LOGTYPE_USER6　0x20000000 /**< User-defined log type 6. */
#define RTE_LOGTYPE_USER7　0x40000000 /**< User-defined log type 7. */
#define RTE_LOGTYPE_USER8　0x80000000 /**< User-defined log type 8. */

3、日志级别

/* ：Can't use 0, as it gives compiler warnings */
#define RTE_LOG_EMERG　　1U　/**< System is unusable.　　*/
#define RTE_LOG_ALERT　　2U　/**< Action must be taken immediately. */
#define RTE_LOG_CRIT　　3U　/**< Critical conditions.　　*/
#define RTE_LOG_ERR　　4U　/**< Error conditions.　　*/
#define RTE_LOG_WARNING　5U　/**< Warning conditions.　　*/
#define RTE_LOG_NOTICE　6U　/**< Normal but significant condition. */
#define RTE_LOG_INFO　　7U　/**< Informational.　　*/
#define RTE_LOG_DEBUG　　8U　/**< Debug-level messages.　　*/

4、改写系统日志文件

/**
* @param f
　*　　文件流指针，　可以是　NULL，如果是　NULL，系统使用默认日志文件流，　　如串口或　syslog.
rte_eal_log_init(const char *id, int facility)　初始化时，已经把默认日专文件流设置为　　syslog
　* @return
　*　　- 0 on success.
　*　　- Negative on error.
　*/
int rte_openlog_stream(FILE *f);

5、设计日志打印级别

void rte_set_log_level(uint32_t level);

6、使能某个日志类型，使能之后，可以记录该类型的日志信息。

void rte_set_log_type(uint32_t type, int enable);

7、取得当前核中刚刚处理的日志消息类型和级别

每个核处理日志消息时，会记录本该消息的类型和级别到一个变量中。

int rte_log_cur_msg_loglevel(void);
int rte_log_cur_msg_logtype(void);

8、使能　LOG存储记录

void rte_log_set_history(int enable);

9、存储或显示历史记录，这个接口，日志是在标准输出中显示的

int rte_log_add_in_history(const char *buf, size_t size);
void rte_log_dump_history(void);

10、打印一条日志，这里会根据类型和级别，判断是否打印，如果打印，则打印到初始化时所设定的文件中。

#define RTE_LOG(l, t, ...)　　\
　　(void)(((RTE_LOG_ ## l <= RTE_LOG_LEVEL) &&　　\

```
        (RTE_LOG_ ## l <= rte_logs.level) &&            \
        (RTE_LOGTYPE_ ## t & rte_logs.type)) ?          \
      rte_log(RTE_LOG_ ## l,                            \
          RTE_LOGTYPE_ ## t, # t ": " __VA_ARGS__) :    \
      0)
```

11、日志打印机制：

```
#define RTE_LOG(l, t, ...)                              \
    (void)(((RTE_LOG_ ## l <= RTE_LOG_LEVEL) &&         \
        (RTE_LOG_ ## l <= rte_logs.level) &&            \
        (RTE_LOGTYPE_ ## t & rte_logs.type)) ?          \
      rte_log(RTE_LOG_ ## l,                            \
          RTE_LOGTYPE_ ## t, # t ": " __VA_ARGS__) :    \
      0)
```

上面调用先判断类型和级别，看是否需要记录，如果不需要，就什么都不做，如果需要就调用：

rte_log(uint32_t level, uint32_t logtype, const char *format, ...)-----

```
rte_vlog(__attribute__((unused)) uint32_t level,__attribute__((unused)) uint32_t logtype,const char *format, va_list ap)
{
    int ret;
    FILE *f = rte_logs.file;
    unsigned lcore_id;

    /*  记录正在打印的日志的类型和级别      */
    lcore_id = rte_lcore_id();
    log_cur_msg[lcore_id].loglevel = level;
    log_cur_msg[lcore_id].logtype = logtype;

    ret = vfprintf(f, format, ap);/*  把日志输出到初始化时定义的文件流中      */
    fflush(f);
    return ret;
}
```

初始化时定义的文件流设置如下（发）：

```
/*
 * set the log to default function, called during eal init process,
 * once memzones are available.
 */
int
rte_eal_log_init(const char *id, int facility)
{
FILE *log_stream;

log_stream = fopencookie(NULL, "w+", console_log_func);
if (log_stream == NULL)
    return -1;

openlog(id, LOG_NDELAY | LOG_PID, facility);

/* 这里把默认的  LOG文件流定义为  fopencookie  所设置的文件流  */
if (rte_eal_common_log_init(log_stream) < 0)
    return -1;
```

```
        return 0;
}
```

Fopencookie 文件流的操作接口定义如下（ lib\librte_eal\linuxapp\ealeal_log.c ）：

```
static cookie_io_functions_t console_log_func = {
        .read  = console_log_read,
        .write = console_log_write,/*  这就是 vfprintf 调用时，真正的写操作接口  */
        .seek  = console_log_seek,
        .close = console_log_close
};
```

下面就是写日志时的真正操作了：

```
static ssize_t
console_log_write(__attribute__((unused)) void *c, const char *buf, size_t size)
{
        char copybuf[BUFSIZ + 1];
        ssize_t ret;
        uint32_t loglevel;

        /*  先把日志志信息记录在历史记录中  */
        rte_log_add_in_history(buf, size);

        /*  再把日志输出到标准输出  */
        ret = fwrite(buf, 1, size, stdout);
        fflush(stdout);

        /* truncate message if too big (should not happen) */
        if (size > BUFSIZ)
                size = BUFSIZ;

        /* Syslog error levels are from 0 to 7, so subtract 1 to convert */
        loglevel = rte_log_cur_msg_loglevel() - 1;
        memcpy(copybuf, buf, size);
        copybuf[size] = '\0';

        /*  最后，还把信息输出到系统日志中  */
        syslog(loglevel, "%s", copybuf);

        return ret;
}
```

HUGPAGE的使用：

为什么内存要分页？实践过程中，有这样的问题：程序需要使用 4G 内存空间，而物理内存又小于 4G，导致程序不得不降低内存占用。为解决些问题，引入 MMU，MMU 核心思想是利用虚拟地址代替物理地址，即 CPU使用虚拟地址， MMU 负责把虚拟地址转成物理地址

内存分页是基于 MMU 的一种内存管理机制，这种机制，从结构上保证了访问内存的高效，使 OS能支持非连续内存的分配。

为了访问更快，硬件上引入 TLB（页表寄存器缓冲），但 TLB有限，为了减少 MISS，引入大页

大页的设置，

```
echo 128 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
mount -t hugetlbfs –o pagesize=2M nodev /mnt/hugepages-fs
```

内存初始化：

1、 映射文件总结

DPDK中，有两个核间共享的全局变量，分别是总内存配置信息和页表数组信息，系统中所有核都可以对这两个全局变量进行访问，为了使每个核访问相同变量时的变量地址是一致的，采用映射相同文件的方式来访问相同的变量地址。

总内存配置信息映射文件 /var/run/.rte_config

rte_config.mem_config 指向总内存配置信息，每个核通过映射 /var/run/.rte_config 文件来使 mem_config 指向相同的物理内存。用 struct rte_mem_config 结构来存储总内存配置信息，其主要包括以下信息：

内存分段信息：总共可存储 256 个连续的内存段，保存在 struct rte_memseg memseg[RTE_MAX_MEMSEG]数组中。这个数组在运行过程中不会变化。

空闲内存段信息 free_memseg：这个一开始就是内存分段的信息，随着运行过程中内存的使用情况，这个信息会做相应的调整。

内存域数组 memzone：这个存储了系统中已分配的内存域，内存域从 free_memseg 中分配得到，并按分配的先后顺序保存在这个数组中，每一个内存域有一个唯一的名字与其对应。

总共可存储 2560 个域。 rte_memzone 是 DPDK内存管理最终向客户程序提供的基础接口，通过 ret_memzone_reverse 可以获取基于 DPDK HUGEPAGE的属于同一个物理 CPU的物理内存连续且虚拟内存也连续的一块地址。 rte_ring/rte_malloc/rte_mempool 等组件就是依赖于 rte_memzone 组件实现的。

页表数组信息映射文件 /var/run/.rte_hugepage_info:

DPDK中，使用到的内存所对应的页用一个 Struct hugepage 表示，系统中会使用很多内存页，把所有内存页的信息放到一个 Struct hugepage 结构数组进行管理，这里暂时把这个数据叫做页表项数组吧！

保存页表项数据的内存也是采用共享内存形式，通过映射文件 /var/run/.rte_hugepage_info 来保证各核所访问的内存是相同的。

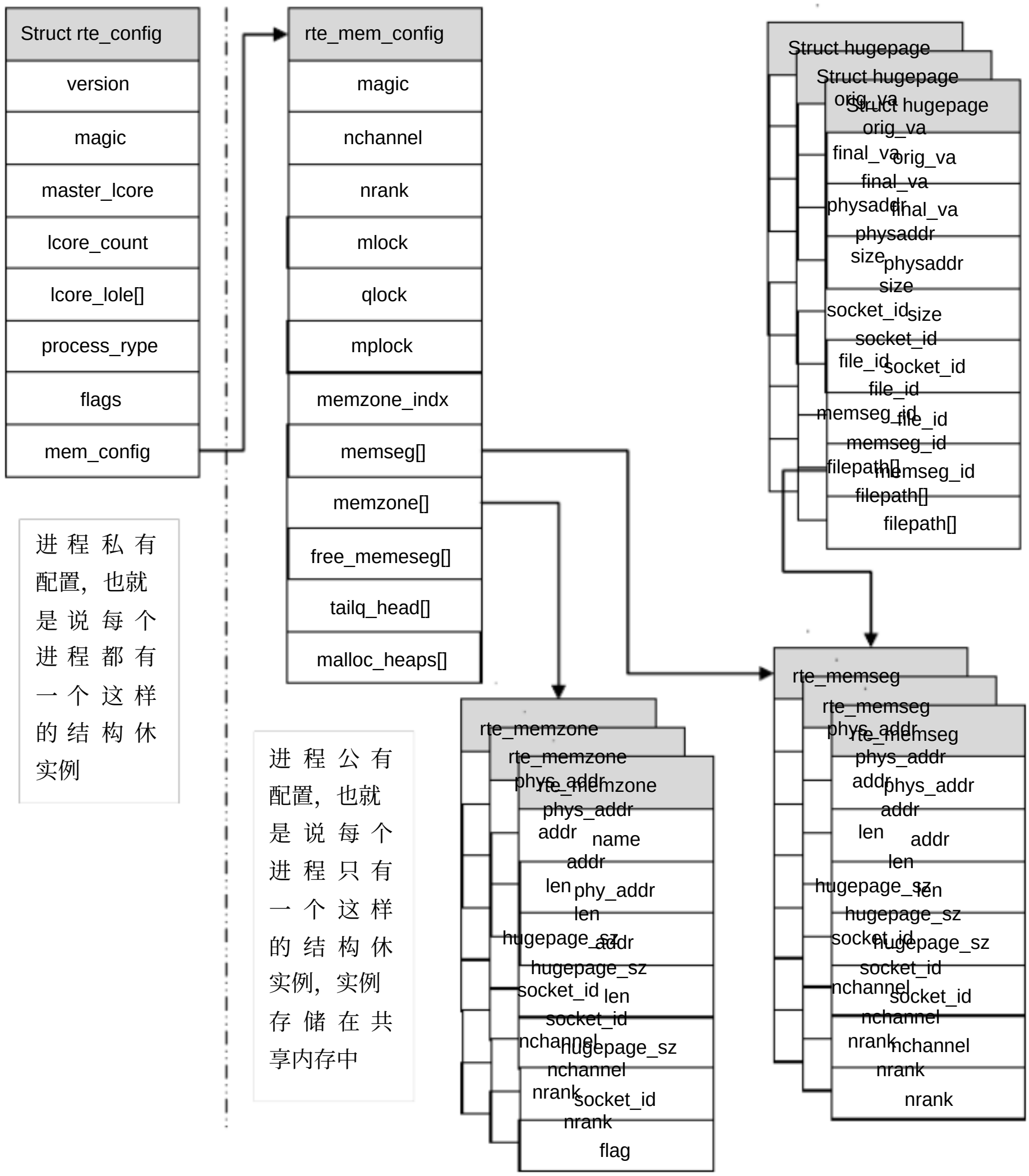
页表项信息记录了每一页所对应的物理地址、 虚拟地址、 该页的大小、 该页所属的 socket_id、该页所挂载到的文件名称，等等。页表项信息结构如下：

```
struct hugepage {
        void *orig_va;           /**< virtual addr of first mmap() orig=1     时放在这里  */
        void *final_va;          /**< virtual addr of 2nd mmap()      虚拟开始地址   */
        uint64_t physaddr;      /**< physical addr    物理开始地址   */
        size_t size;             /**< the page size    该页的大小   */
        int socket_id;           /**< NUMA socket ID */
        int file_id;             /**< the '%d' in HUGEFILE_FMT  在  map_all_hugepages 时的顺序  */
        /* 该页所属的内存段，一个内存段包含的页是相连续的          */
        int memseg_id;           /**< the memory segment to which page belongs */
        /* 该大页缓存所挂载到的目录下的文件       */
        char filepath[MAX_HUGEPAGE_PATH]; /*rtemap%s*< path to backing file on filesystem */
};
```

其中 orig_va 在初始化完内存之后全为 NULL，不再使用。

页挂载路径 /mn/huge/rtesmp_%fileid

由上述页表项信息结构知道，每一个页都挂载到 hugepage 文件系统中的一个文件中上，文件路径名格式是： /mn/huge/rtesmp_%fileid



各结构关系图

2、 共享内存子系统的初始化

DPDK的共享内存是使用系统中的大页内存来实现的，对共享内存的管理，其实就是管理这些大页内存， 所谓共享内存初始化的主要工作， 就是把系统中的大页内存的信息 （如内存大小，相应的物理地址等）采集并存储在 DPDK自已所设计的一系例结构体变量上，通过这些结构体变量， DPDK可以按照自已的方式来管理系统中的内存分配和申请。

DPDK通过以下结构来管理共享内存：

系统大页文件系统信息结构： 用于存储采集到的系统中的各种大页文件系统的页面大小和页面数。 系统中可能有不同页面大小的大页文件系统， 如页面大小为 2M（32 位机） 或 1G（64 位机），路径 /sys/kernel/mm/hugepages/ 保存了系统中所有大页面文件系统的信息。 DPDK用

数组 internal_config.hugepage_info[MAX_HUGEPAGE_SIZES]来保存所有不同页面大小的HUGETLBFS。

```
struct hugepage_info {
	size_t hugepage_sz;	/**<  页面大小  */
	const char *hugedir;	/**<  目录/MNT/HUGE //dir where hugetlbfs is mounted */
	// 一个 NUMA NODE 包含多个 socket，一个 socket 包含多个核，可以把一个 socket 看成
	// 是一个物理 CPU，同一个 socket 中的内存是共享的，
	// 这里表示不同物理 CPU 上的页数
	uint32_t num_pages[RTE_MAX_NUMA_NODES];
			/**< number of hugepages of that size on each socket	*/
	int lock_descriptor;	/**< file descriptor for hugepage dir	大页目录的文件锁  */
};
```

页信息结构：用于保存页面的物理地址、映射到进程中的虚拟地址、页面大小、页面所属物理 CPU、页面对应的 MMAP 文件等信息。DPDK用这个结构体数组来保存 DPDK使用到的所有页面的信息。我们在这里把这个结构体数组称为页表，每一个表项表示一个页。DPDK把页表保存到了共享内存映射文件 /var/run/.rte_hugepage_info ，不同的进程通过访问这个文件，都可以得到这个全局的页表。

```
struct hugepage {
	void *orig_va;	/**< virtual addr of first mmap()	只在页面信息建立过程中使用,	*/
	void *final_va;	/**< virtual addr of 2nd mmap()	页面信息建立后的最终虚拟地址	*/
	uint64_t physaddr;	/**< physical addr */
	size_t size;	/**< the page size	该页的大小  */
	int socket_id;	/**< NUMA socket ID	页面所属物理 CPU 号 */
	int file_id;	/**< the '%d' in HUGEFILE_FMT  在 map_all_hugepages 时页被发现的的顺序 */
	/* 该页所属的内存段，一个内存段包含的页是物理地址和虚拟地址都是相连续的	*/
	int memseg_id;	/**< the memory segment to which page belongs */
	/* 该大页缓存所挂载到的目录下的文件	*/
	char filepath[MAX_HUGEPAGE_PATH]; /*rtemap%s*< path to backing file on filesystem */
};
```

这里要说明的是，DPDK为了提高内存性能，对地址做了优化，尽量（这里是说尽量，要看程序地址空间是否足够用）把物理地址上连续的页映射到连续的虚拟地址上。DPDK还用了另外一个优化，那就是把相同的物理地址映射到每个进程的虚拟地址也保持一样，这样共享内存的虚拟地址就可以在不同的进程中传递了。

共享内存配置信息结构体：DPDK把大页面内存映射到自己的地址空间并保存在页表项数组，这时只是取得了要管理的内存，之后通过另个一个结构来管理所取得的内存，这个结构如下：

```
struct rte_mem_config {
	volatile uint32_t magic;	/**< Magic number - Sanity check. */
	/* memory topology */
	uint32_t nchannel;	/**< Number of channels (0 if unknown). */
	uint32_t nrank;	/**< Number of ranks (0 if unknown). */
	rte_rwlock_t	mlock;	/* 访问 free_memseg 的锁 *<  only used by memzone LIB for thread-safe. */
	rte_rwlock_t qlock;	/**< used for tailq operation for thread safe. */
	rte_rwlock_t mplock;	/**< only used by mempool LIB for thread-safe. */
	/* 内存域索引，表示当前系统中已分配了多少个内存域	*/
	uint32_t memzone_idx; /**< Index of memzone */
	/* memory segments and zones */
	struct rte_memseg memseg[RTE_MAX_MEMSEG];	/**< Physmem descriptors. */
```

```
        struct rte_memzone memzone[RTE_MAX_MEMZONE]; /**< Memzone descriptors. */
        /* Runtime Physmem descriptors. */
        struct rte_memseg free_memseg[RTE_MAX_MEMSEG];
        // 存储了系统中的所有队列的队列头指针
        struct rte_tailq_head tailq_head[RTE_MAX_TAILQ]; /**< Tailqs for objects */
        /* 各 socket 上对应的堆，    rte_malloc  从这里分配内存   */
        /* Heaps of Malloc per socket */
        struct malloc_heap malloc_heaps[RTE_MAX_NUMA_NODES];
} __attribute__((__packed__));
```

其中，struct rte_memseg memseg[RTE_MAX_MEMSEG]就是用来管理　DPDK所映射的所有大页面内存的结构体数组，　该数组每一个成员代表一个物理地址和虚拟地址都连续的一块内存分片。也就是说，　DPDK映射到的大页面内存中，可能存在几个页面的物理地址是连续的，如果程序空间足够大，就可以把这几个页面映射到连续的虚拟地址空间上，　　DPDK把边几个地址上连续页面所对应的内存视为一个内存段，用一个　　struct　rte_memseg　结构来保存这个内存段的信息，如起始物理地址、起始虚拟地址、长度等信息，　　struct rte_memseg　结构如下：

```
struct rte_memseg {
        phys_addr_t phys_addr;          /**< Start physical address. */
        union {
                void *addr;              /**< Start virtual address. */
                uint64_t addr_64;       /**< Makes sure addr is always 64 bits */
        };
        size_t len;                      /**< Length of the segment. */
        /* 属于哪种页面大小的大页面文件系统的内存        */
        size_t hugepage_sz;             /**< The pagesize of underlying memory */
        int32_t socket_id;                /**< NUMA socket ID.   所属物理   CPU 号*/
        uint32_t nchannel;               /**< Number of channels. */
        uint32_t nrank;                  /**< Number of ranks. */
} __attribute__((__packed__));
```

综上所述，　DPDK的内存初始化工作，主要是将　　hugetlbfs　配置的大内存页，根据其映射的物理地址是否连续、属于哪个　　socket 等信息，有效的组织起来，为后续管理提供便利。

3、 memzone 子系统的初始化

DPDK中，有多个功能模块，如　　RING 模块等，都要用到共享内存，　　DPDK把为每个功能模块分配的内存保存在用内存域来表示，　　保存在　memzone 数组中。 每个　memzone 有其自已的名字字符串表示。

另外，为了实现内存的快速访问，　　DPDK不直接从　memseg 数组中分配内存，因为　　memseg 中每个内存的起始地址和结束地址不一定是缓冲线对齐的，　　这样不利于内存的快速访问。　因此，　DPDK用 struct　rte_memseg　free_memseg[RTE_MAX_MEMSEG]数组来管理内存缓冲线对齐后的内存分片。各功能模块申请内存时，都从这个　　free_memseg　数组中申请内存，把申请到的内存保存到　memzone 中。

内存申请接口如下：

```
const struct rte_memzone *
rte_memzone_reserve(const char *name, size_t len, int socket_id,
                    unsigned flags)
```

已申请的内存域查询接口：

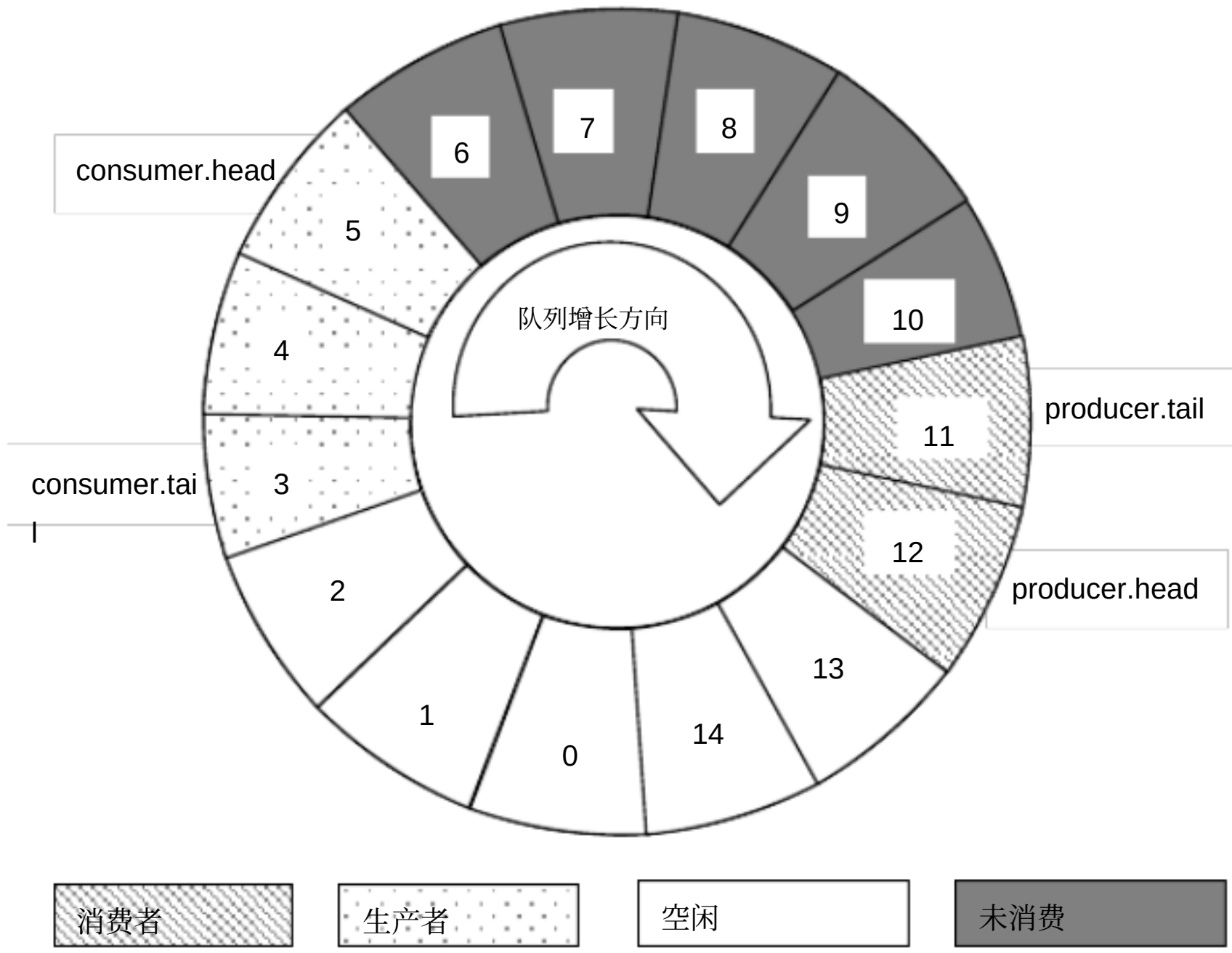
```
Void rte_memzone_dump(void)
```

多生产者多消费者的无锁环形队列

DPDK设计了一种无锁环形队列，这种无锁队列适用于单生产者单消费者、单生产者多消费

者、多生产者单消费者、多生产者多消费者等多种情形。
无锁环形队列的结构体如下：

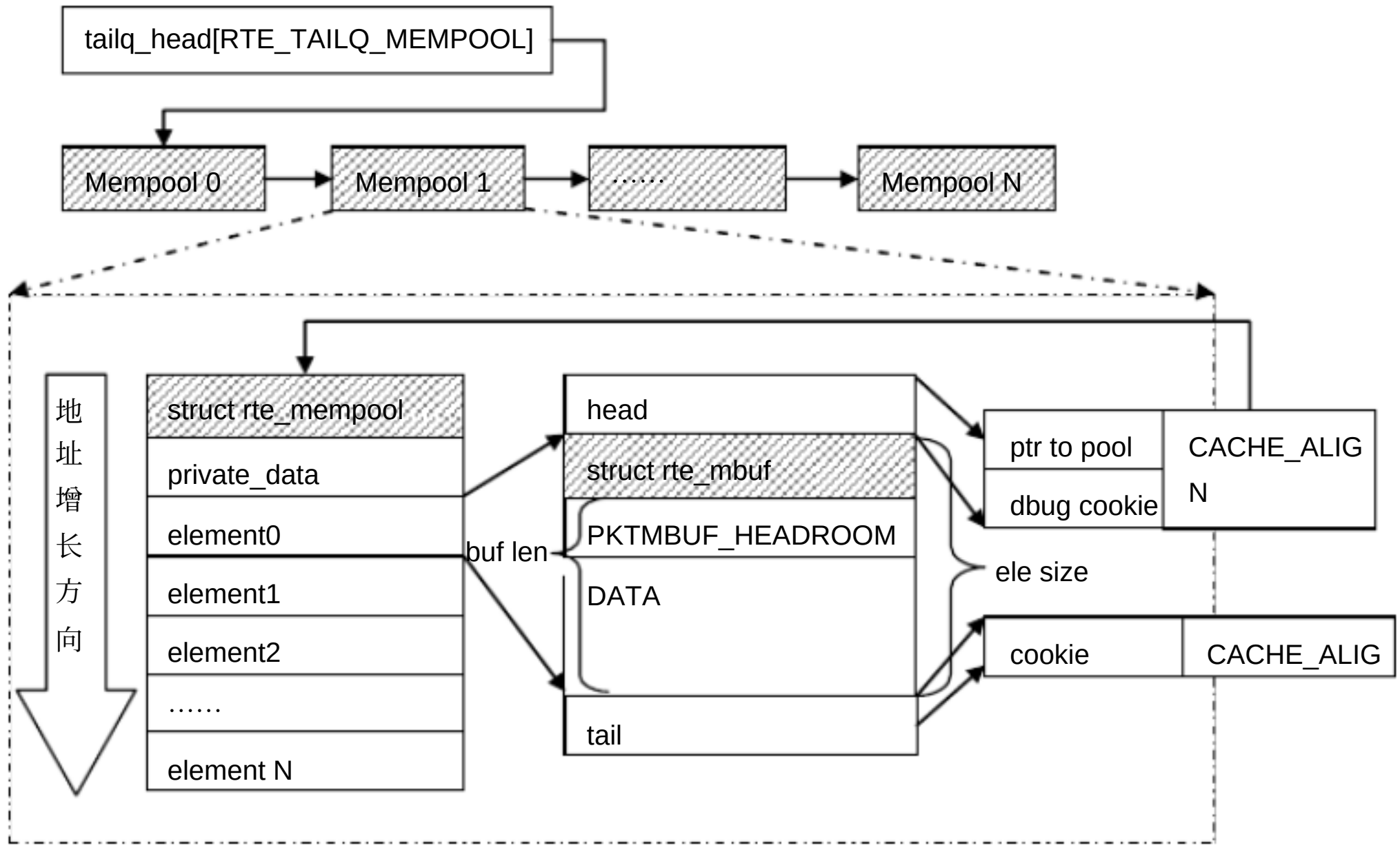
```
struct rte_ring {
        TAILQ_ENTRY(rte_ring) next;          /**< Next in list.    用于链接到全局环形队列列表中       */
        char name[RTE_RING_NAMESIZE];      /**< Name of the ring. */
        int flags;                    /**< Flags supplied at creation.    标志位，如生产者消费者模式      */

        /** Ring producer status.    生产者的状态   */
        struct prod {
                // 可供消费的   items（条目）总数警介线，等于      (size-1)表示不设置警介线
                uint32_t watermark;             /**< Maximum items before EDQUOT. */
                // 表示是否多生产者
                uint32_t sp_enqueue;           /**< True, if single producer. */
                // 环形队列的大小、生产者与消费者的        size  和  mask 是一样的
                uint32_t size;                    /**< Size of ring. */
                uint32_t mask;                    /**< Mask (size-1) of ring. */
                // 生产者的头指针和尾指针，其实生产完成后，这两个值都是指队列尾
                volatile uint32_t head;      /**< Producer head. */
                volatile uint32_t tail;       /**< Producer tail. */
        } prod __rte_cache_aligned;

        /** Ring consumer status. */
        struct cons {
                // 是否多消费者
                uint32_t sc_dequeue;           /**< True, if single consumer. */
                uint32_t size;                    /**< Size of the ring. */
                uint32_t mask;                    /**< Mask (size-1) of ring. */
                // 消费者头尾指针，其实消费完成后，这两个值都是指向队列头
                volatile uint32_t head;      /**< Consumer head. */
                volatile uint32_t tail;       /**< Consumer tail. */
#ifdef RTE_RING_SPLIT_PROD_CONS
        } cons __rte_cache_aligned;
#else
        } cons;
#endif

#ifdef RTE_LIBRTE_RING_DEBUG
        struct rte_ring_debug_stats stats[RTE_MAX_LCORE];
#endif
        // 队列中保存的所有对象
        void * ring[0] __rte_cache_aligned; /**< Memory space of ring starts here.
                                             * not volatile so need to be careful
                                             * about compiler re-ordering */
};
consumerconsumer
```



如图，环形队列长度为 15，队列的增长方向是顺时钟方向，也就说，生产者以顺时钟方向往队列中放东西，而消费者同样以顺时钟方向从队列中取东西。当生产者需要往队列中存数据时，先用生产者队列头指针加上需要存放数据的数量得到新的生产者头指针，然后从生产者尾指针开始向队列中逐个存入数据，直到所有要存入的数据都存入完成后，把生产者尾指针加上所生产的数据量得到新的生产者尾指针；当有消费者需要取数据时，先把原消费者头指针加上打算消费者的数量得到新的消费者头指针，然后从消费者尾部逐个取出数据，直到完成所要提取的数据量，最后把消费者尾指针加上所消费的数量得到新的消费者尾指针。

内存池管理

DPDK把相同大小的内存块用一个称为内存池的数据结构来管理，每一个内存块称为 MBUF。当程序需要使用到相应大小的内存块时，可以向相应的内存池申请内存。
所有内存池都统一放到一个内存池列表中，这个列表的表头就存储在全局共享内存中的列表数组中： rte_config.mem_config->tailq_head[RTE_TAILQ_MEMPOOL] 。内存池用字符串名称唯一标识。内存池拓朴结构图如下：



以 pktmbuf 为例的内存池拓朴图

DPDK用一个 struct rte_mempool 结构体来管理一个内存池，其结构组成如下：

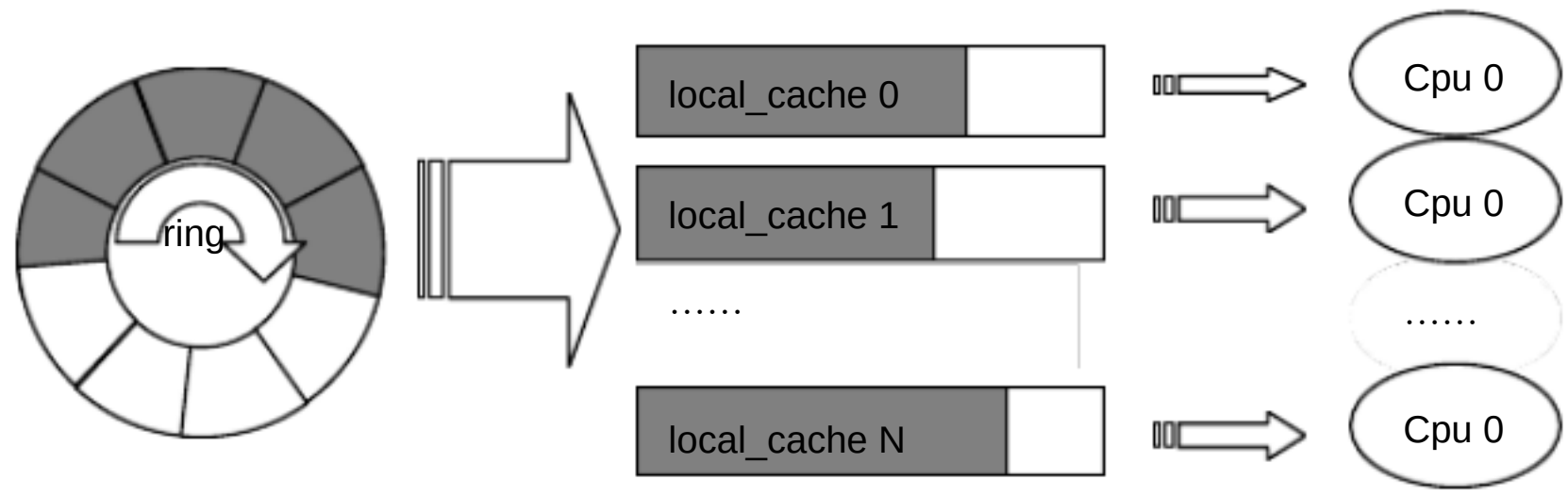
```
struct rte_mempool {
        TAILQ_ENTRY(rte_mempool) next;     /**< Next in list. */

        char name[RTE_MEMPOOL_NAMESIZE]; /**< Name of mempool. */
        struct rte_ring *ring;                  /**< Ring to store objects.    保存对象的全局环    */
        phys_addr_t phys_addr;                  /**< Phys. addr. of mempool struct. */
        int flags;                              /**< Flags of the mempool. */
        // 内存池里面有多少个内存块
        uint32_t size;                          /**< Size of the mempool. */
        // 本地缓存队列长度，当本地缓存小于要分配的数量时，
        // 要从全局环中取出足够多的内存填补本地缓存，使是分配
        // 完指定数目的内存后，本地缓存长度等于         cache_size
        uint32_t cache_size;                    /**< Size of per-lcore local cache. */
        // 当本地缓冲队列长度大于这个值时，把大于         cache_size的部分写回到
        // 全局环缓冲区中
        uint32_t cache_flushthresh;             /**< Threshold before we flush excess elements. */

        // 内存池里面每一个内存块单元有大小，指从内存
        // 单元起始地址到结束地址之间的大小，不包含头尾
        uint32_t elt_size;                      /**< Size of an element. */
        // 内存单元起始地址之前的头部大小
        uint32_t header_size;                   /**< Size of header (before elt). */
        // 内存单元结束地址之后的尾部大小
        unsigned private_data_size;             /**< Size of private data. */

#if RTE_MEMPOOL_CACHE_MAX_SIZE > 0
```

```
        /** Per-lcore local cache.  每个核一个本地池，这样申请内存时，不需要互斥操作就可以申请，当本地池小于某个值时，才从全局中申请到本地池中，再从本地池可申请      */
        struct rte_mempool_cache local_cache[RTE_MAX_LCORE];
#endif

#ifdef RTE_LIBRTE_MEMPOOL_DEBUG
        /** Per-lcore statistics. */
        struct rte_mempool_debug_stats stats[RTE_MAX_LCORE];
#endif
}   __rte_cache_aligned;
```

由以上结构可以看出， 它是基于无锁环形队列来管理内存块的； 另外， 由于所有核都可能向内存池申请或释放内存块， 这样就可能存在多核之间的竞争， 当申请或释放动作比较频繁时，可能由于竞争导致出现申请释放时间延迟的可能。

为了尽可能的降低多核之间的况争频率， DPDK为每一个核分配了一个本地私有内存队列，并保证每个队列至少有一定数目的内存以供本址 CPU使用，当 CPU核申请或释放内存时，都只对本 CPU 的本地私有队列进行申请，只有本地队列长度太小时，才向全局环队列申请内存。这样就大大减少了 CPU 访问全局环形队列的频率，使多核竞争概率降到一个很小的值，有效避免了时间延迟。

其管理结构拓朴图如下：



用 rte_mempool_create(const char *name, unsigned n, unsigned elt_size,

```
                unsigned cache_size, unsigned private_data_size,
                rte_mempool_ctor_t *mp_init, void *mp_init_arg,
                rte_mempool_obj_ctor_t *obj_init, void *obj_init_arg,
                int socket_id, unsigned flags)
```

可以创建一个内存池，参数解释如下：

const char *name， 内存池的字符串名称

unsigned n，内存池总的内存块数量

unsigned elt_size, 对于报文件缓存的 MBUF 内存池，这个是三个长度的和： sizeof(struct rte_mbuf) + RTE_PKTMBUF_HEADROOM+DATA_len

unsigned cache_size，本地缓存队列长度， 当本地缓存小于要分配的数量时， 要从全局环中取出足够多的内存填补本地缓存，使分配完指定数目的内存后，本地缓存长度等于 cache_size

unsigned private_data_size, 内存池私有数据部分的长度，对于报文缓存池，这个私有数据用下面结构来存储，它表示缓存区数据部分的长度（包含 RTE_PKTMBUF_HEADROOM），其大小是在创建报文缓存区内存池时设置的，默认是 2048 + RTE_PKTMBUF_HEADROOM

```
struct rte_pktmbuf_pool_private {
        uint16_t mbuf_data_room_size; /**< Size of data space in each mbuf.*/
};
```

rte_mempool_ctor_t *mp_init, 内存池初始化函数，对于报文缓存区是 rte_pktmbuf_pool_init ，只是对内存池私有数据部分做初始化，也就是初始化 mbuf_data_room_size

void *mp_init_arg, 内存池初始化函数的参数
rte_mempool_obj_ctor_t *obj_init, 对象初始化函数，对于报文缓存区是 rte_pktmbuf_init
void *obj_init_arg, 对象初始化函数的参数
int socket_id,
unsigned flags

MBUF 管理
DPDK设置两种类型的 MBUF，一种是报文缓存类型，一种是控制信息类型，其宏定义为：

```
enum rte_mbuf_type {
        RTE_MBUF_CTRL,  /**< Control mbuf. */
        RTE_MBUF_PKT,   /**< Packet mbuf. */
};
```

MBUF 的结构可参考内存池拓朴结构图，其结构定义如下：

```
struct rte_mbuf {
        struct rte_mempool *pool; /**< Pool from which mbuf was allocated. */
        //MBUF 的缓存开始地址，从 struct rte_mbuf 开始但不包含 struct rte_mbuf
        void *buf_addr;                 /**< Virtual address of segment buffer. */
        phys_addr_t buf_physaddr; /**< Physical address of segment buffer. */
        //MBUF 内存长度， buf_len=mp->elt_size - sizeof(struct rte_mbuf)
        uint16_t buf_len;               /**< Length of segment buffer. */
#ifdef RTE_MBUF_SCATTER_GATHER
        /** 这个结构用于引用计数
          * 16-bit Reference counter.
          * It should only be accessed using the following functions:
          * rte_mbuf_refcnt_update(), rte_mbuf_refcnt_read(), and
          * rte_mbuf_refcnt_set(). The functionality of these functions (atomic,
          * or non-atomic) is controlled by the CONFIG_RTE_MBUF_REFCNT_ATOMIC
          * config option.
          */
        union {
                rte_atomic16_t refcnt_atomic;       /**< Atomically accessed refcnt */
                uint16_t refcnt;                    /**< Non-atomically accessed refcnt */
        };
#else
        uint16_t refcnt_reserved;           /**< Do not use this field */
#endif
        uint8_t type;           /**< Type of mbuf. MBUF 的类型 RTE_MBUF_CTRL或 RTE_MBUF_PKT, */
        uint8_t reserved;                   /**< Unused field. Required for padding. */
        uint16_t ol_flags;                  /**< Offload features. */

        //DPDK 的 MBUF 只有这两种情况使用
        union {
                struct rte_ctrlmbuf ctrl;
                struct rte_pktmbuf pkt;//  网络数据报文的信息
        };
} __rte_cache_aligned;
```

定时器（ ALARM）子系统
编程中经常需在指定某个操作异步的延时一段时间后才执行， 注意， 这里的异步是指在延时等待的时候， 主流程可以同时执行其它操作， 不需要等待。 本文件把这种异步延时操作称为定时器任务。 系统可能同时有很多异步延时操作， 也就是说， 可能在某段时间内会同时存在多个定时器任务。

DPDK把所有定时器任务按等待时间顺序从小到大保存在一个定时器链表中，每个链表结点表示一个定时任务，结点组织结构如下

```
struct alarm_entry {
        LIST_ENTRY(alarm_entry) next;//用于链接到链表的指针
        struct timeval time;//   设定任务将来执行的时间点
        rte_eal_alarm_callback cb_fn;//  将来要执行的任务，用一个函数来表示
        void *cb_arg;//  函数的参数
        volatile int executing;//  任务是否正在执行
};
```

在 DPDK 初始化时，通过函数 timerfd_create 创建一个文件描述符，当定时器链表中有至少一个定时器时， DPDK通过调用函数 timerfd_settime 来通知内核，让内核在将来某个时间点产生信号，当内存产生信号时， timerfd_create 所创建的文件描述符变为可读，这时 DPDK就可以遍历链表了。

那么 DPDK 是怎么读取 timerfd_create 所创建的文件描述符的呢？它通过创建一个线程（这个线程 DPDK称之为中断线程，后面中断子系统会做介绍）来专门监听文件描述符的变化，当文件描述符变为可读时，就可以执行遍历定时器链表的操作了。

中断子系统（ DPDK第一个功能线程 ）

DPDK 是专门用来处理网络报文的，它通过 open 函数打开网口对应的 UIO 设备接口得到一个文件符，通过 read()函数，读这个文件描述符来检查 UIO 设备是否有读事件产生。当网卡中有中断产生时，内核 UIO 模块的中断处理函数会产生一个读事件，从而唤醒用户空间的 read()函数，使得 DPDK调用的 read()函数返回。这样，内核就把硬件中断通知到了应用空间的 read()函数了。 因此，read 一个 UIO 设备，就相当于一个监听硬件是否有中断产生的过程。另外，对 UIO 节点进行 write 操作，可以实现关闭和使能 UIO 设备中断的功能，也就是说，对 UIO 设备节点读写，可以达到响应和控制中断的目的。

DPDK 把前面一章中定器时器子系统的 timerfd 文件描述符和本章的 UIO 文件描述符统一称为中断源。可以这么说， timerfd 是内核定时器软中断在应用空间的表现形式，而 UIO 节点是 UIO 物理设备中断在应用空间的表现形式。当 timerfd 文件描述符或 UIO 文件描述符可读时，表示有中断发生。中断源用下面 struct rte_intr_source 结构体描述：

```
struct rte_intr_handle {
        int fd;                                  /**< file descriptor     中断所对应的文件描述符     */
        enum rte_intr_handle_type type;       /**< handle type    系统中只有    UIO 或定时器中断类型    */
};
struct rte_intr_source {
        TAILQ_ENTRY(rte_intr_source) next;
        //
        struct rte_intr_handle intr_handle; /**< interrupt handle */
        // 一个中断源可能有多个回调函数
        struct rte_intr_cb_list callbacks;      /**< user callbacks */
        // 为 1 表示中断已产生，正在被处理，不能
        // 删除中断源
        uint32_t active;
};
```

所有中断源都保存在中断源列表 intr_sources 中，DPDK中只有两种中断源， 用户也可以其于此结构再添加中断源， 添加中断源的方法是通过调用 rte_intr_callback_register 向中断子系统注册中断源。

DPDK 专门创建一个子线程用于监听所有中断源，检查是否有中断发生，如果有，则调用 rte_intr_callback_register 注册的中断处理进程。

DPDK并不是采用通常的 select()方法来监听中断，而是用一种内核新技术，叫做 epoll 的方

法，该方法只要用 epoll_create 创建一个 epoll 对像（这个对像用文件描述符表示），然后把所有要监听的中断源所对应的文件描述符用 EPOLL_CTL_ADD 命令加入到 epoll 对像中，最后调用 epoll_wait，等待中断事件发生，epoll_wait 返回时，会返回当前 epoll 对像中所有发生的 epoll 事件，并保存到 epoll_event 结构体数组中，通过 epoll_event 结构体数组可以找到发生中断的文件描述符所对应的中断源，从而执行中断源对应的回调函数。

时间子系统

DPDK这了实现更精准的时间延时，提供了一个微秒级的时间延时接口：

Void rte_delay_us(unsigned us)

调用这个接口，进行会挂起指定的时间后被重新唤醒。

该接口有两种可选的时钟源，分别的时间戳计算器（TSC）和高精度定时器（HPET），TSC是系统中的一个 64 位寄存器，每过一个 CPU时间，TSC计数会加 1；HPET是一种通过访问设备文件 /dev/hpet 来实现计时的一种精度更高的计时方法，这里暂时不做研究。

我们看一下基于 TSC时钟的定时是怎么实现的。

```
void
rte_delay_us(unsigned us)
{
	const uint64_t start = rte_get_timer_cycles();//	取得当前 CPU 周期
	const uint64_t ticks = (uint64_t)us * rte_get_timer_hz() / 1E6;//	把 us 转化成 CPU 周期数
	while ((rte_get_timer_cycles() - start) < ticks)//	判断消耗的 CPU周期数
		rte_pause();
}
```

从上面可知，DPDK先把要延时的时间换算成 CPU 周期数，然后不停的查看 PCU周期计数器，看计数是否到达设定的值，如果是，则表示定时到期，返回。

查看 CPU周期的方法分两种，就是前面说的 TSC和 HPET，查看 CPU周期的接口如下：

```
static inline uint64_t
rte_get_timer_cycles(void)
{
	switch(eal_timer_source) {
	case EAL_TIMER_TSC:
		return rte_rdtsc();//	用 rdtsc 汇编指令从寄存器中读取
	case EAL_TIMER_HPET:
#ifdef RTE_LIBEAL_USE_HPET
		return rte_get_hpet_cycles();//	使用 HPET读取 CPU周期。
#endif
	default: rte_panic("Invalid timer source specified\n");
	}
}
```

系统初始化时，默认设置使用 TSC时钟：

```
int
rte_eal_timer_init(void)
{
	eal_timer_source = EAL_TIMER_TSC;//设置时间源全局变量
	set_tsc_freq();
	return 0;
}
```

PCI抽象子系统

初始化

通常，网卡都是基于 PCI总线的，所以，DPDK用一个 PCI层来管理系统中所有 PCI总线设备（当然，如果不使用 PCI 接口的网卡，可在配置中把 PCI 子系统关闭掉，配置变量是 internal_config.no_pci ）。

PCI子系统初始化时，主要是完成以下任务：

1、 初始化三个列表的头指针：

PCI设备列表： struct pci_device_list device_list struct pci_driver_list driver_list;// 用于保存扫描到的所有 PCI 设备， PCI 设备是一种基于 PCI 总线接口的设备，如 CPI 接口网卡设备，不是PCI本身的设备。

PCI设备驱动列表： ;// 用于保存所有已注册的 PCI 设备驱动，在调用 rte_XXX_pmd_init 接口加载相应的网卡驱动时，会把网卡驱动以 PCI 驱动的形式，向这个列表注册一个 PCI驱动。

UIO 资源列表： uio_res_list，这个实际是把 uio_res_list 指向 mcfg->tailq_head[RTE_TAILQ_PCI]中。

2、 扫描出系统所有 PCI 设备，从 PCI设备 SYS文件系统中获取 PCI 设备的参数信息，填写

PCI设备结构体：

```
struct rte_pci_device {
        TAILQ_ENTRY(rte_pci_device) next;           /**< Next probed PCI device. */
        struct rte_pci_addr addr;                   /**< PCI location. */
        struct rte_pci_id id;                       /**< PCI ID. */
        struct rte_pci_resource mem_resource[PCI_MAX_RESOURCE]; /**<PCI Memory Resource */
        struct rte_intr_handle intr_handle;         /**< Interrupt handle */
        const struct rte_pci_driver *driver;        /**< Associated driver PCI 设备的驱动， 这里是网卡驱动 */
        uint16_t max_vfs;                   /**< sriov enable if not zero     是否支持 sriov 功能 */
        int numa_node;                      /**< NUMA node connection */
        unsigned int blacklisted:1;                 /**< Device is blacklisted */
};
```

这些设备信息从 /sys/bus/pci/devices/DeviceADDR/ 目录中的相应文件获取：

设备厂商信息：

/sys/bus/pci/devices/DeviceADDR/subsystem_device

/sys/bus/pci/devices/DeviceADDR/subsystem_vendor

/sys/bus/pci/devices/DeviceADDR/vendor

/sys/bus/pci/devices/DeviceADDR/device

是否支持 sriov 功能，在虚拟化技术上用， 1 为支持， 0 为不支持

/sys/bus/pci/devices/DeviceADDR/max_vfs

PCI资源信息，如内存，中断， IO 地址等：

/sys/bus/pci/devices/DeviceADDR/resource ，其内容格式如下：

资源开始物理地址	资源结束物理地址	标志，如内存、中断、 IO 等
0x00000000d8940000	0x00000000d895ffff	0x0000000000140204
0x0000000000000000	0x0000000000000000	0x0000000000000000
0x00000000d8900000	0x00000000d890ffff	0x0000000000140204
0x0000000000000000	0x0000000000000000	0x0000000000000000
0x00000000000020c0	0x00000000000020ff	0x0000000000040101
0x0000000000000000	0x0000000000000000	0x0000000000000000
0x00000000dc410000	0x00000000dc41ffff	0x000000000004e200

3、 探寻设备所使用的驱动

在运行时环境（ RTE）初始化时，只是把扫描到的 PCI设备加到 PCI设备列表 device_list 中，这时 PCI 设备还没有对应的驱动，需要等到 RTE初始化完成，且进行 pmd （PCI 设备驱动）初始化后，调用 rte_eal_pci_probe （）来进行设备驱动探寻操作。

在扫描 PCI设备时，还无法得到中断

另外，下面这几个信息在 pci_scan（）里面还没有填写

struct rte_intr_handle intr_handle; /**< Interrupt handle 在扫描 PCI 设备时，调用函数 pci_uio_map_resource 初始化了中断句柄 */

const struct rte_pci_driver *driver; /**< Associated driver 在 rte_eal_pci_probe_one_driver 里，探寻到设备驱动后，会把这个指针关联到相应的驱动中 */

unsigned int blacklisted:1; /**< Device is blacklisted */

PCI设备驱动属性文件作用：

bind ：向这个文件写入 PCI设备的地址（如： 0000:02:05.0），可以把对应的 PCI设备绑定到这个驱动中。

Unbind ：向这个文件写入 PCI 设备的地址（如： 0000:02:05.0），可以把对应的 PCI 设备与这个驱动分离，也就是不使用这个驱动。

new_id ：向这个文件写入 PCI 设备的厂商 ID 和设备 ID 信息等（如 vendor/device/subvendor/subdevice/class/class_mask/driver_data 中的至少前两个信息），可以使这个驱动支持指定的设备类型。

remove_id ：与 new_id 相反。

uevent ：向这个文件写入 :add,remove,change,move,online,offline 命令，如写入 add 命令，这样可以向 udevd 发送一条 netlink 消息，让它再重新一遍相关的 udev 规则文件很有用。

网口驱动层

所有网口都用一个 struct rte_eth_dev 表示，DPDK最多默认支持 32 个网口，所有网口的 struct rte_eth_dev 保存在一个数组中： rte_eth_devices[nb_ports]

rte_eth_dev_callback_register

可以注册多个中断回调函数，当网卡产生连接状态变化的中断时 ，分别执行 struct rte_eth_dev-> 回调列表中的所有函数

QOS

流的参数为：接收端口号、发送端口号、接收核、调度核、发送核

流的数量最大为： RTE_MAX_LCORE^2=32

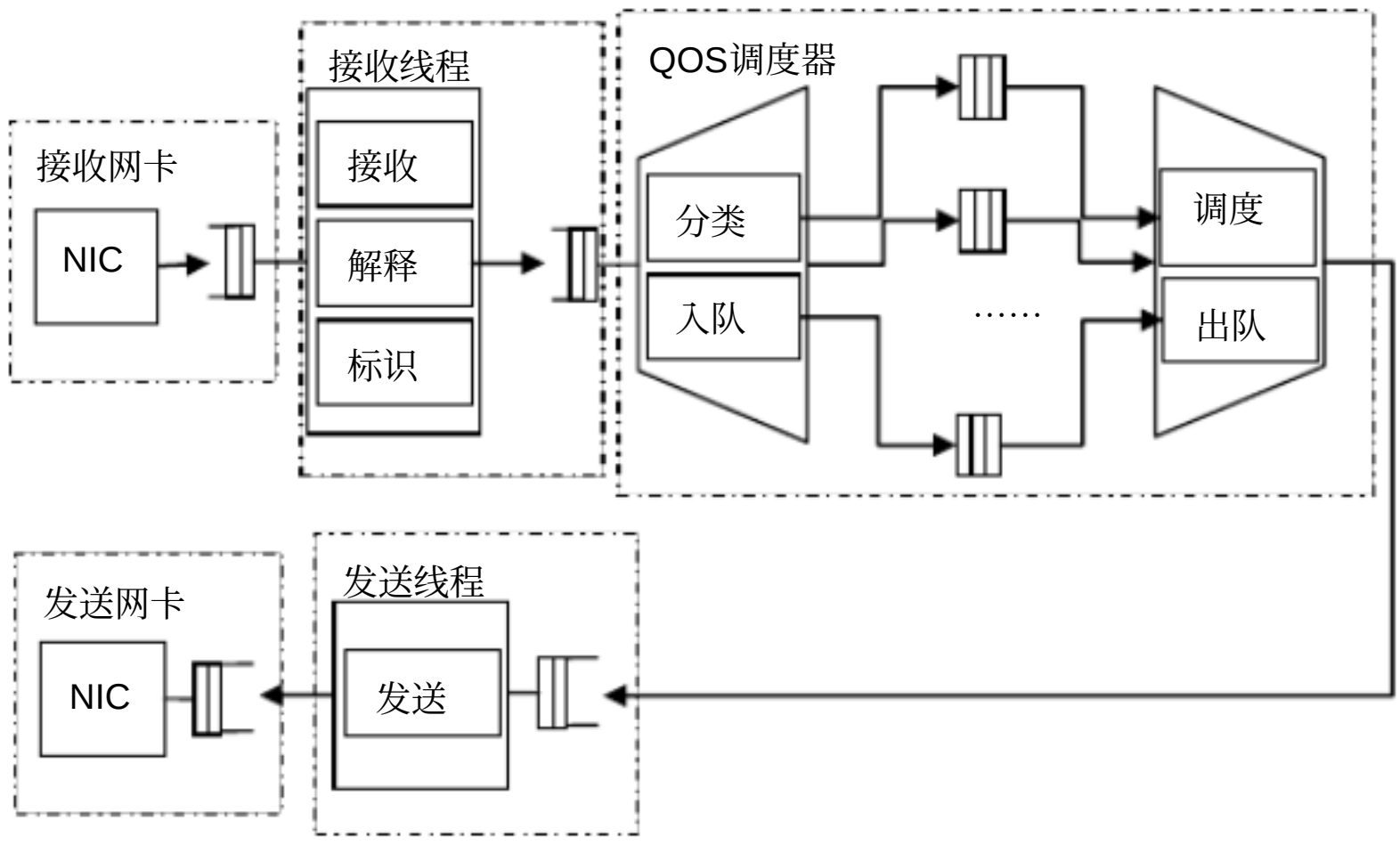
端口号最大为： 32，

每个流的接收或发送端口要不一样，也就是说不能出现一个端口有两个流在接收或者发送

每个流的接收核与调度核不能同一个，也就是说系统一定至少要有两个核，一个用于接收，一个用于调度和发送

多个流可以同一个核接收，同一个核发送，

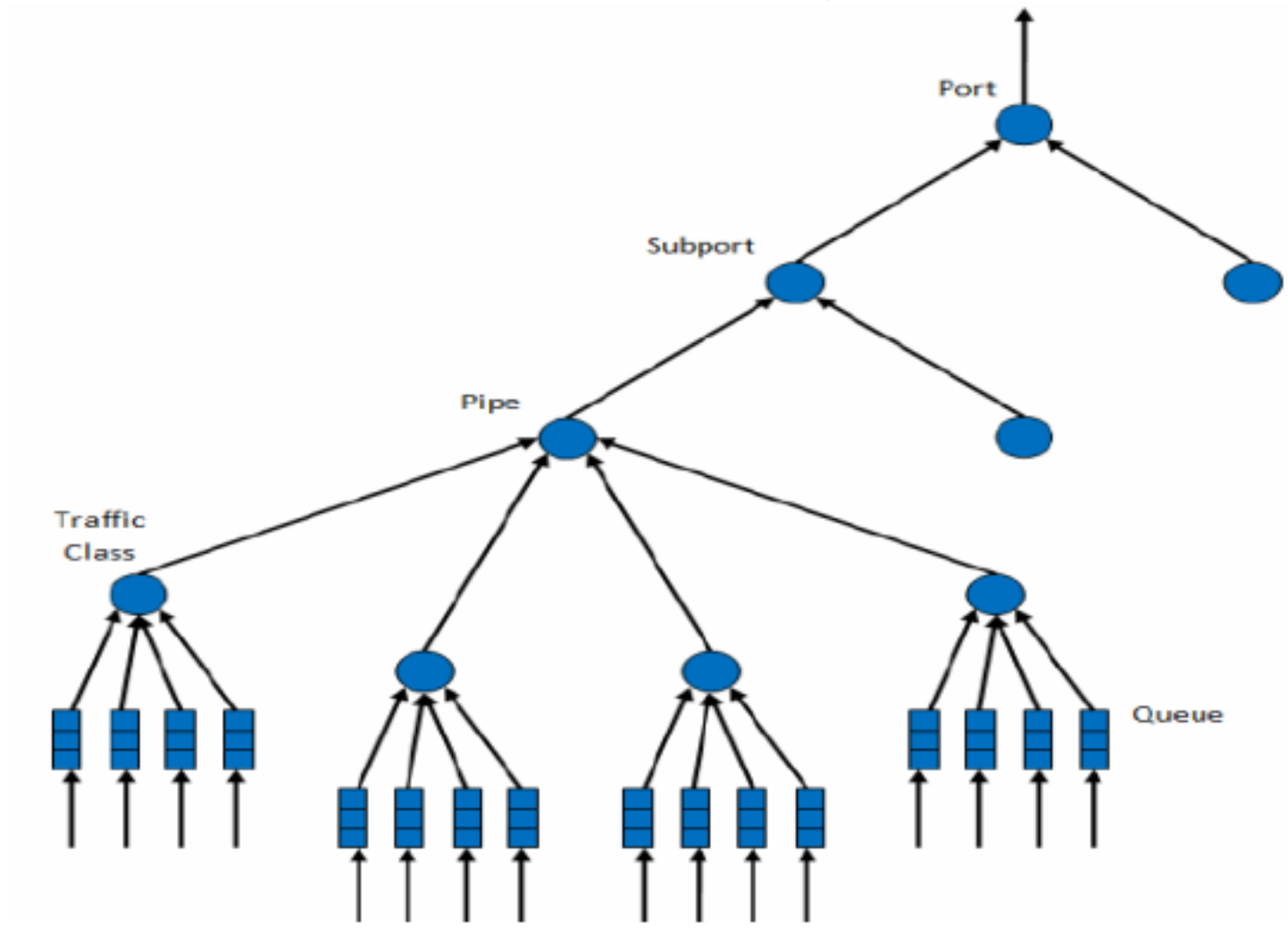
下面是一个使用 DPDK QOS功能来处理网络报文实例流程图：



使用 DPDK QOS功能处理网络报文流程图

由上图可见， DPDK的 QOS调度机制是通过设置多个队列用于缓存接收到的不同类型的报文，发送报文的时候，按照设定的 QOS策略决定从哪个队列中读取报文进行发送。

DPDK把所有 QOS队列用一个分级的树形结构来描述和管理，如图：



其中， Port 代表物理网口； 每个 Port 下最多 64 个 Subport（默认 8 个） ，每一个 Subport 可以代表一组用户； 每个 Subport 节点下最多有 1M 个 Pipe 节点（默认 4K），一个节点可以代表一组用户中的某个用户；每个 Pipe 节点下有 4 个 TC(traffic class)节点，每一个节点代表用户中的某一种流最类型，如数据、语音、媒体等业务流量类型，其中每一种 TC 可以有不同的速率、 延迟和斗动要求； 每一个 TC节点下有 4 个 Queue 节点， 每个 Queue 节点代表一个调度队列。

QOS 调度过程分析

1、接收：接收网卡把收到的报文 DMA 到自已的硬件接收环缓冲区中，接收线程从硬件的环缓冲区中取出报文，解释报文类型，把类型标识到报文和管理结构 rte_mbuf->pk. Hash. sched 中，其实就是标识报文的 subport, pipe, traffic_class, queue, color，最后把报文暂存在一个接收队列中，等待调度线程把报文调度到相应的 QOS队列。

2、入队：调度线程从接收队列中取出一定数量的报文，根据每个报文的标识，找到相应的 QOS队列，把报文分配到相应的 QOS队列中，然后到 BIT 图中激活该 QOS 队列，如果配置了随机早检测机制，入队前需要先做随机早检测，如果检测结果为丢弃则报文不入队。 RED 算法是什么，下面会专门分析。

3、出队：调度线程把一定数目的报文入队后，采用相应的调度策略，从 QOS队列中选择一个队列， 然后从所先择的队列中把报文取出保存到发送队列中， 等待发送线程发送。 调度策略是什么，下面会专门分析。

4、发送：发送线程从发送队列中读取报文，把报文发送到外出网卡的缓冲区中。

随机早检测的方法：

QOS队列的长度是固定的， 当某个类型的报文以很高的速度到达系统， 而系统的处理速速又跟不上时， QOS队列很快就会因为满而无法缓存该类型的报文，这时报文将会被丢。

但这种尾部丢包机制存在一些典型问题： 1、容易产生全局同步现象； 2、对突发流量存在偏见。

采用随机早检测可以解决上面存在的问题，其方法如下：

1、 颜色标记：在每种流量类型（ TC）中，设置 3 种颜色来标记报文，这样，把可以同一类型的流量再细分为不同的子类型，从而使不同的子类型分别用不同的 RED门限。比如，对于多媒体报文，可以分为 TCP握手报文、 TCP数据报文、其它报文等，这样，可以把 TCP握手报文的门限提高，保证 TCP握手报文不容易被丢弃。

2、 RED配置参数设置和算法

```
/**
 * RED configuration parameters
 */
struct rte_red_config {
        // 平均队列长度在这两个值之间随机丢弃
        uint32_t min_th;        /**< min_th scaled in fixed-point format */
        uint32_t max_th;        /**< max_th scaled in fixed-point format */
        // 计算丢弃概率所使用的比较值
        uint32_t pa_const; /**< Precomputed constant value used for pa calculation (scaled in
fixed-point format) */
        uint8_t maxp_inv;      /**< maxp_inv */
        // 当前队列长度在计算平均队列长度时的权重         ,<0,选   2^(-n), 方便左右移位计算
        uint8_t wq_log2;        /**< wq_log2 */
};
/**
 * RED run-time data
 */
struct rte_red {
        // 平均队列长度   ,这个长度是放大了的    ,不是实际队列长度的数量级
        uint32_t avg;             /**< Average queue size (avg), scaled in fixed-point format */
        // 从上次标志报文不丢弃到当前的入队列数目
        uint32_t count;          /**< Number of packets since last marked packet (count) */
        uint64_t q_time;        /**< Start of the queue idle time (q_time) */
};
```

通过下面算法计算平均队列长度：

avg_new = (1 - wq) * avg + wq * q

=》 avg = avg + q * wq - avg * wq （1）

其中， 等号左边的 avg_new 表示所要计算的平均队列长度； 等号右边的 avg 表示旧的平均队列长度； q 表示当前队列长度； wq 表示当前队列长度在计算时的权重，通常为 0 到 1 之间，为了方便左右移位计算，通常取 2^(-n)， wq 为 1 表示计算时不考虑历史平均值。

由于上述算法（ 1）中的 avg * wq 涉及小数运算，程序中可以把等式两边同时乘以一个数，如 2^(N+n)，其中 N>n，这样，就变成了：

avg *2^(N+n)= avg*2^(N+n) + q * wq*2^(N+n) - avg *2^(N+n)* wq

使 avg_s= avg *2^(N+n)，则变成：

avg_s = avg_s + q * 2^N - avg_s * 2^(-n)

这样平均队列长度的计算变变成了左右移位计算。

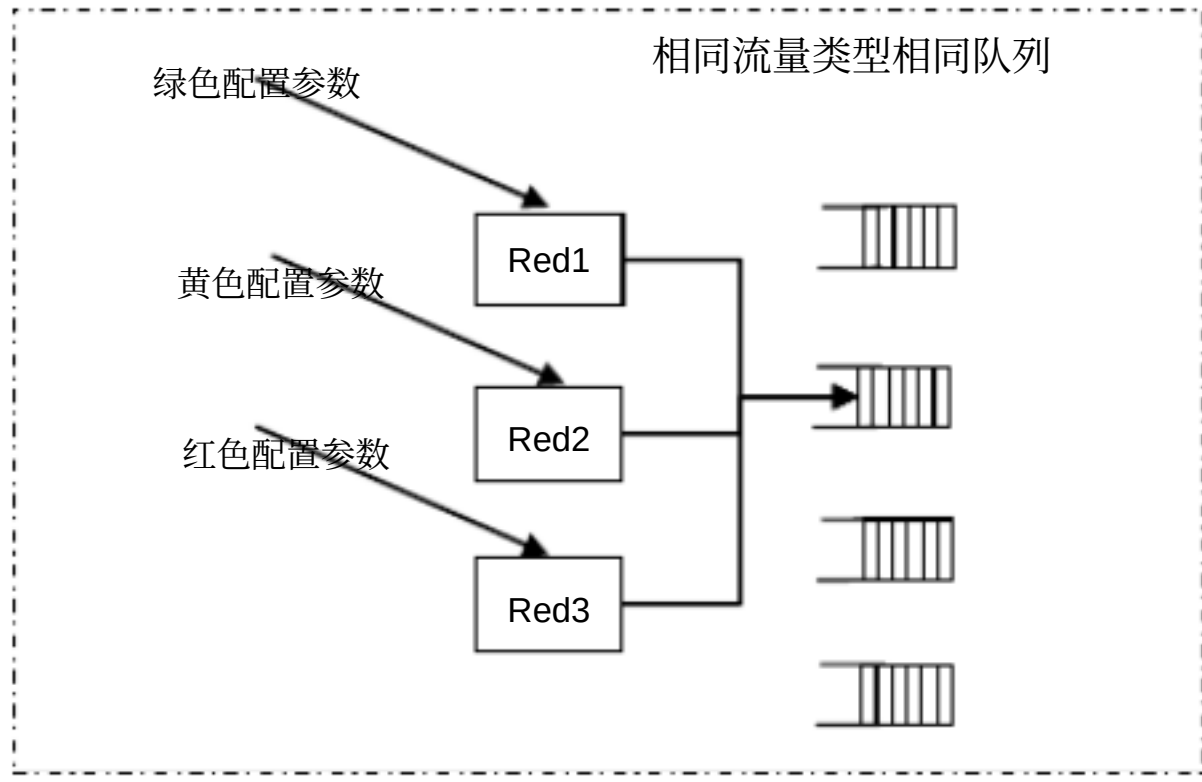
当 avg 值小于 min_th 时，报文不用丢弃，当大于 max_th 时，报文被丢弃；当在 min_th 和 max_th 之间时，报文随机丢弃，随机丢度的算法如下：

1、 计算累计连续不丢弃报文的个数 pa_num

2、 计算平均队列长度与最小门限的差值

3、 用前面计算所得到的两个值相乘，所得到的值 pa_num_count 与 pa_const，如果大于 pa_const，则丢弃；如果小于 pa_const，则用随机值 rte_red_rand_val 来计算是否丢弃，公式如下： rte_red_rand_val/ （pa_const-pa_num_count ），如果小于 pa_num 则不丢弃，如果大于 pa_num 则丢弃。

随机早检测示意图如下：



队列调度方法：

队列的调度就是选择一个 QOS队列来发送报文。 DPDK对于所有 PIPE采用轮询的方法调度，也就是说，每个 PIPE之间是平等的，没有先后之分；对于同一 PIPE中的 4 个 TC，采用严格优先级的方法；而对于同一 TC的 4 个调度队列，采用 WRR的方法。

为了加速内存数据的读取， DPDK 分几个阶段来读取 QOS 队列中的报文，从 QOS 队列中调度出一个报文用于发送的步骤是：

1、 扫描 BIT 图，从 BIT 图中选择下一个活动的 PIPE（PIPE指中至少有一个队列有报文在等待发送），把该 PIPE的 struct rte_sched_pipe 结构的内容预取到物理缓存中。

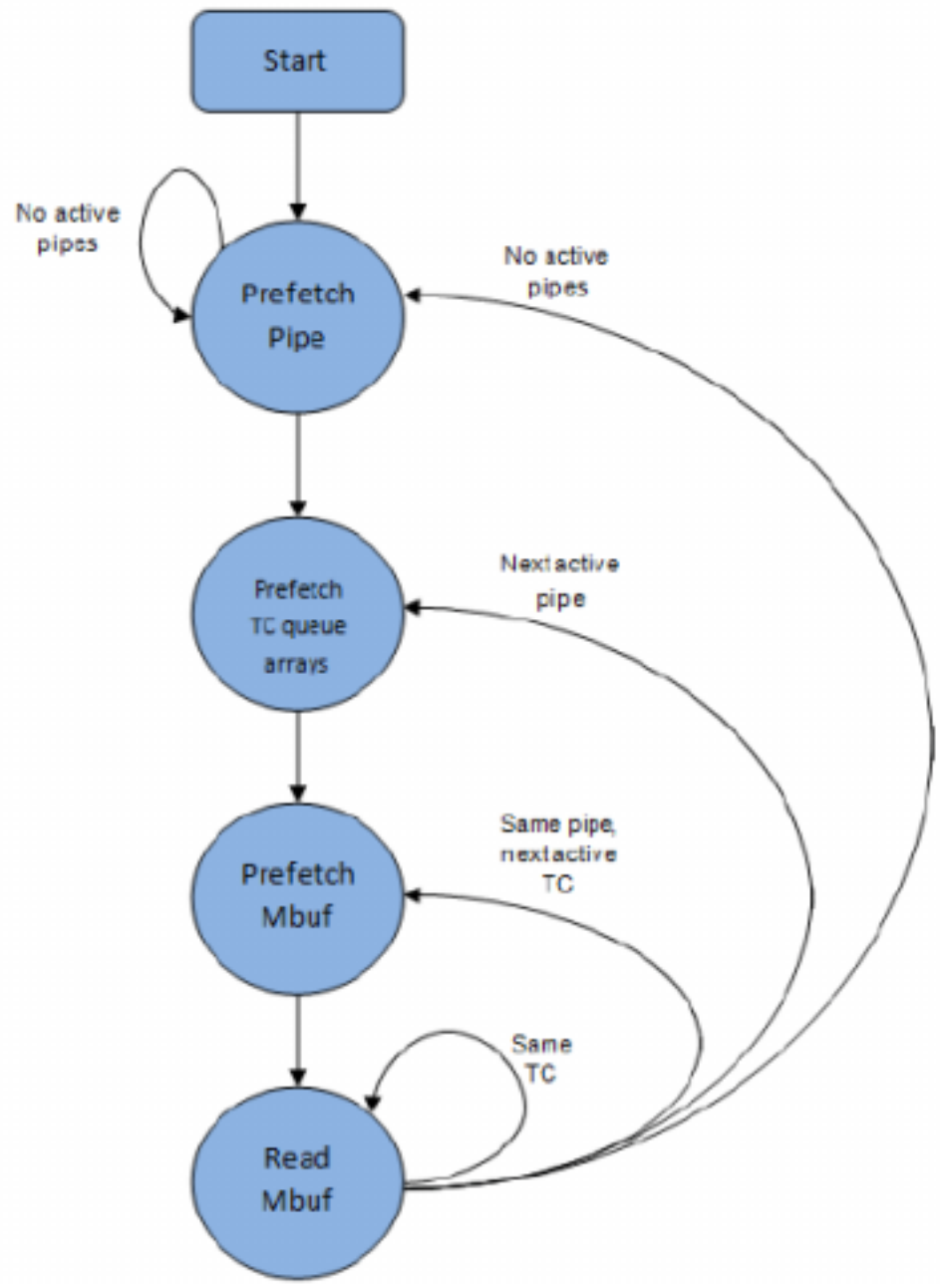
由于下一步读取队列头指针时要用到每个队列的当前读写位置结构体（struct rte_sched_queue），所以这一步也顺便把当前 TC中 4 个队列读写位置结构体数组的内容预取到缓存中。

2、 读取 PIPE数据结构，更新当新 PIPE和其所在的子端口的令牌桶的令牌数。选择该 PIPE的第一个 TC（严格优先级），在所选择的 TC中用 WRR 算法选取一个调度队列。预取所选择的 TC中 4 个 QOS队列的头指针（只从读指针 queue[i]->qr 开始预取）。这个队列其实是一个数组，保存的是报文描述符结构体的地址。

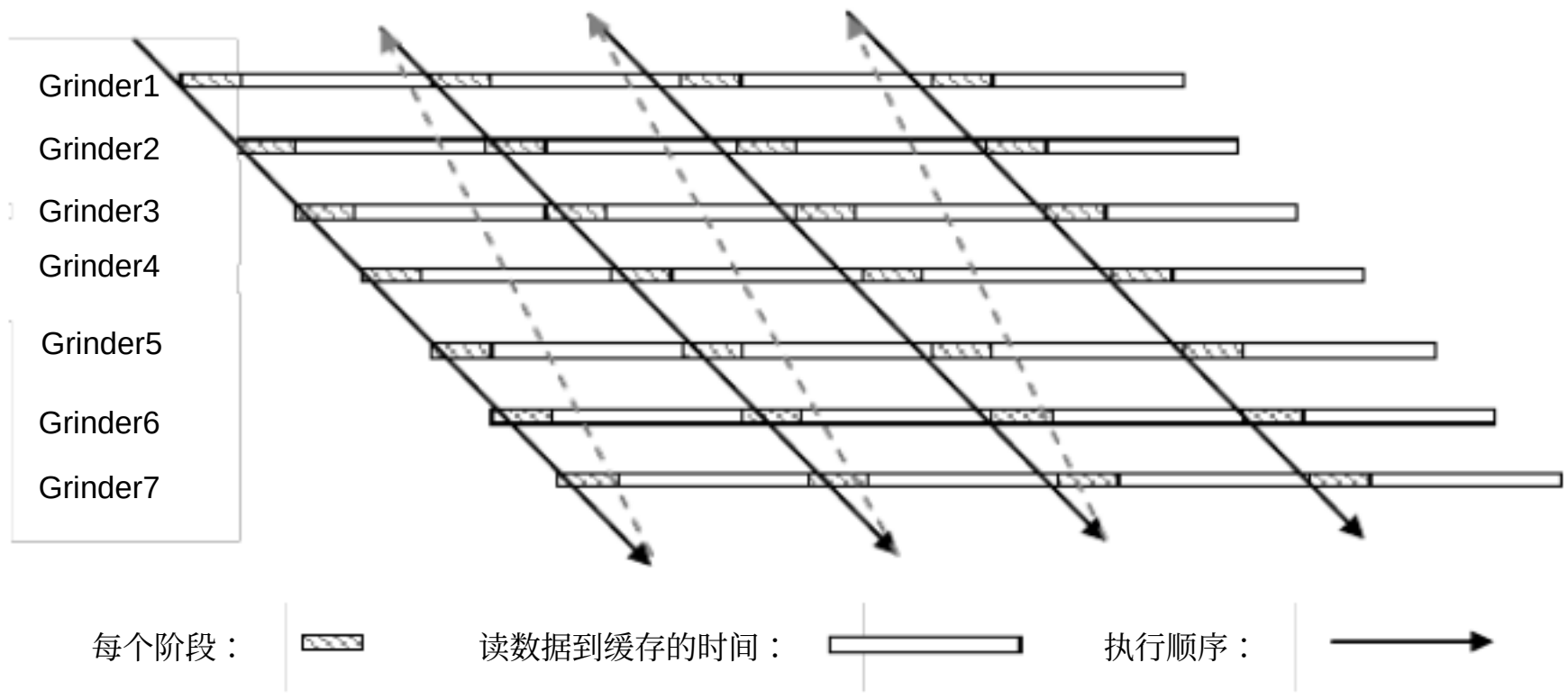
3、 从当前 QOS队列中读出队头元素，得到要发送的报文描述符结构体（ struct rte_mbuf ）的地址，预取这个地址的内容到缓存。

4、 读报文描述符结构体（ struct rte_mbuf ），从中读取报文长度，根据令牌数决定报文是否可以发送，如果不能发送，放弃当前 TC，因为当前 TC 已经达到流量上限，需要重新找一个 TC来发送。 回到第三步； 如果当前 PIPE的所有 TC 都没有可以发送的报文， 则重新读出一个 PIPE，回到第二步； 如果当前 PIPE也没有可以发送的报文，则回到第一步， 重新从 BIT 图选取一个组 PIPE。

整个过程如下图所示：



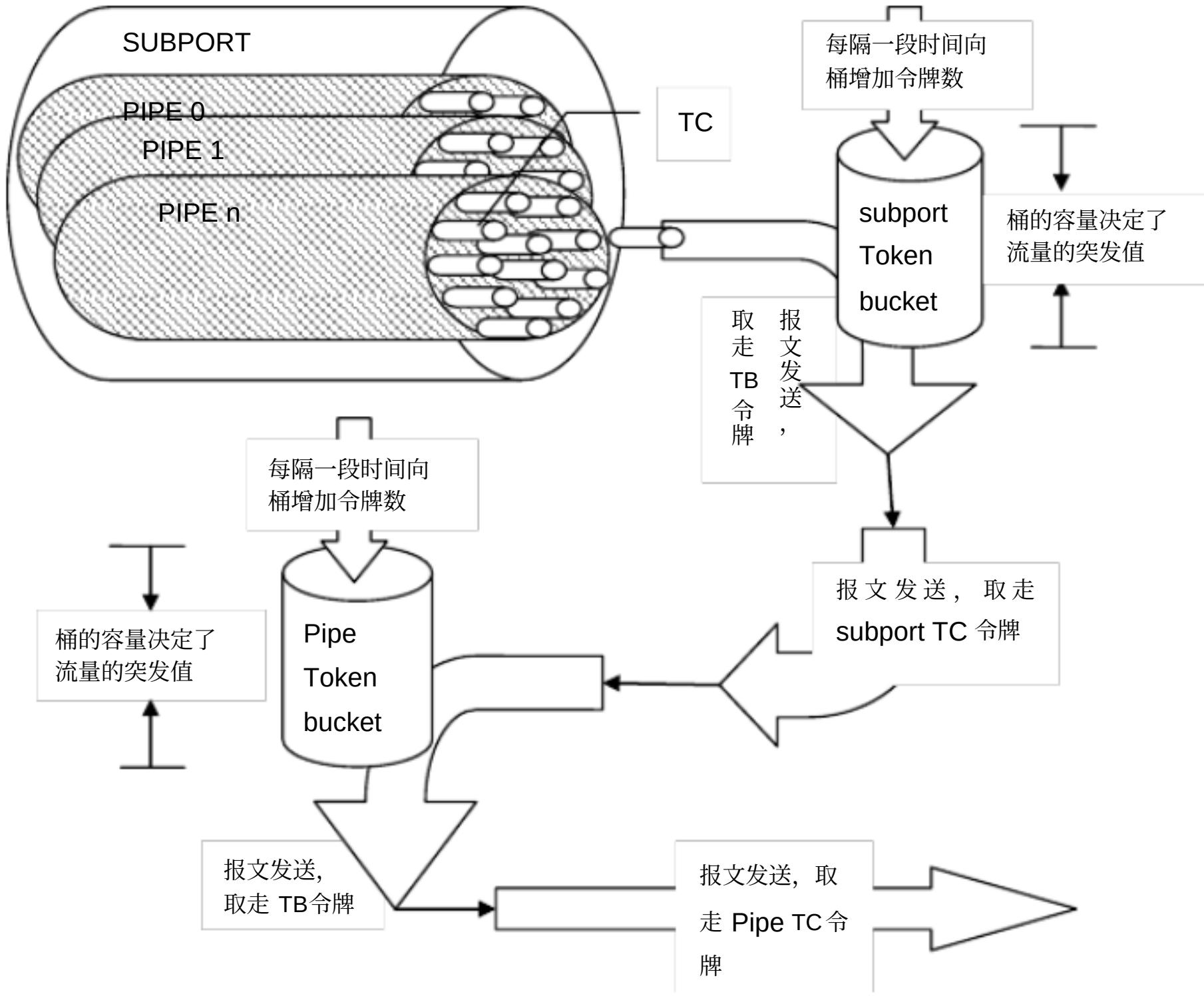
上面 4 个阶段中， 前一阶段是为了后一阶段读取数据做内存预取准备工作， 每个阶段都需要等待前一阶段内存准备好后才能执行， 最后一个阶段， 每发送一个报文， 都要重新预取下一个报文的内存，这样，阶段之间就有一个时间空闲，如果 CPU 连续地执行上面 4 个阶段，则预取内存的效果没有得到体现。 为了有效利用 CPU，在预取内存的同时做其它操作， DPDK 设计了 8 个 grinder，每一个 grinder 包括上面 4 个阶段的步骤，轮留执行 8 个 grinder，每个 grinder 只执行一个阶段， 这样，可以保证每个 grinder 阶段执行时，本阶段所要访问的数据已经在缓存中准备好了。从下图可以直观看到 CPU没有把时间浪费在数据据上。



采用我个 grinder 分阶段执行的好处还在于实现了各 PIPE之间的随机轮询， 谁先准备好数据则先选择谁。

当选择了一个 PIPE，则在 PIPE 中按照 TC的顺序选择 TC；而当选择了一个 TC 后，则在 TC 的队列按 WRR 选择队列。

WRR 的实现：
流量限制和整形的实现：流量的限制是在报文从 QOS 队列出列之前，通过令牌桶的机制实现的，DPDK为每一个子端口和每个 PIPE设置了一个令牌桶， 每个令牌桶中设置有效令牌数；另外在每个子端口和 PIPE中还设置了不同流量类型（ TC）的有效令牌数。
令牌桶中的有效令牌数：用于限制本子端口或 PIPE当前时刻可以通过的总字节数。
TC有效令牌数：限制本子端口或 PIPE的本 TC类型流量当前时刻可能通过的总字节数。
报文出列之前， 先要判断报文的长度是否小于这 4 个令牌数的最小值， 如果是， 则报文可以出列；如果不是，则报文不可以出列，需要选择其它流量类型的队列来读取报文并出列。

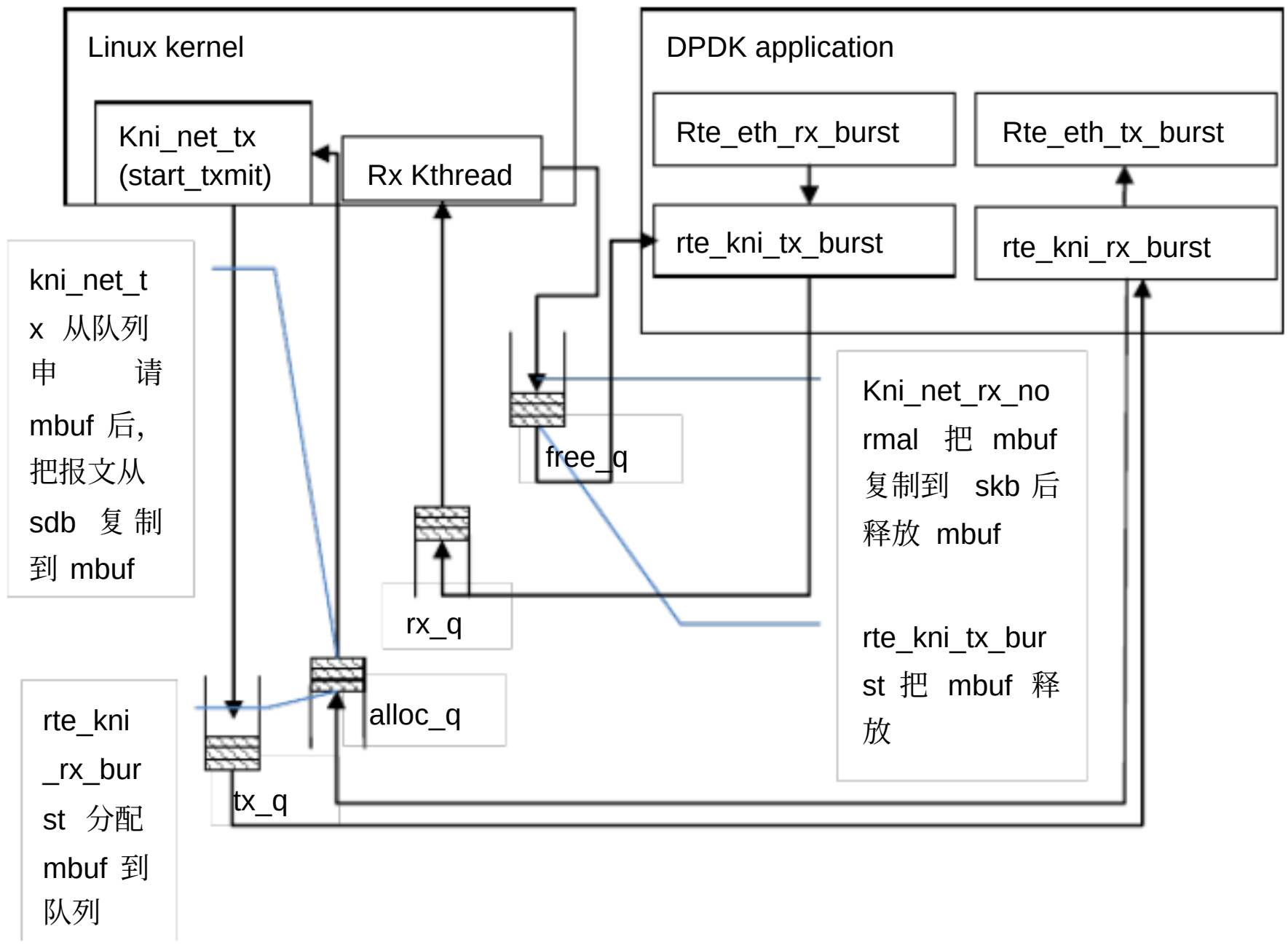


限速的最小单位是 TC

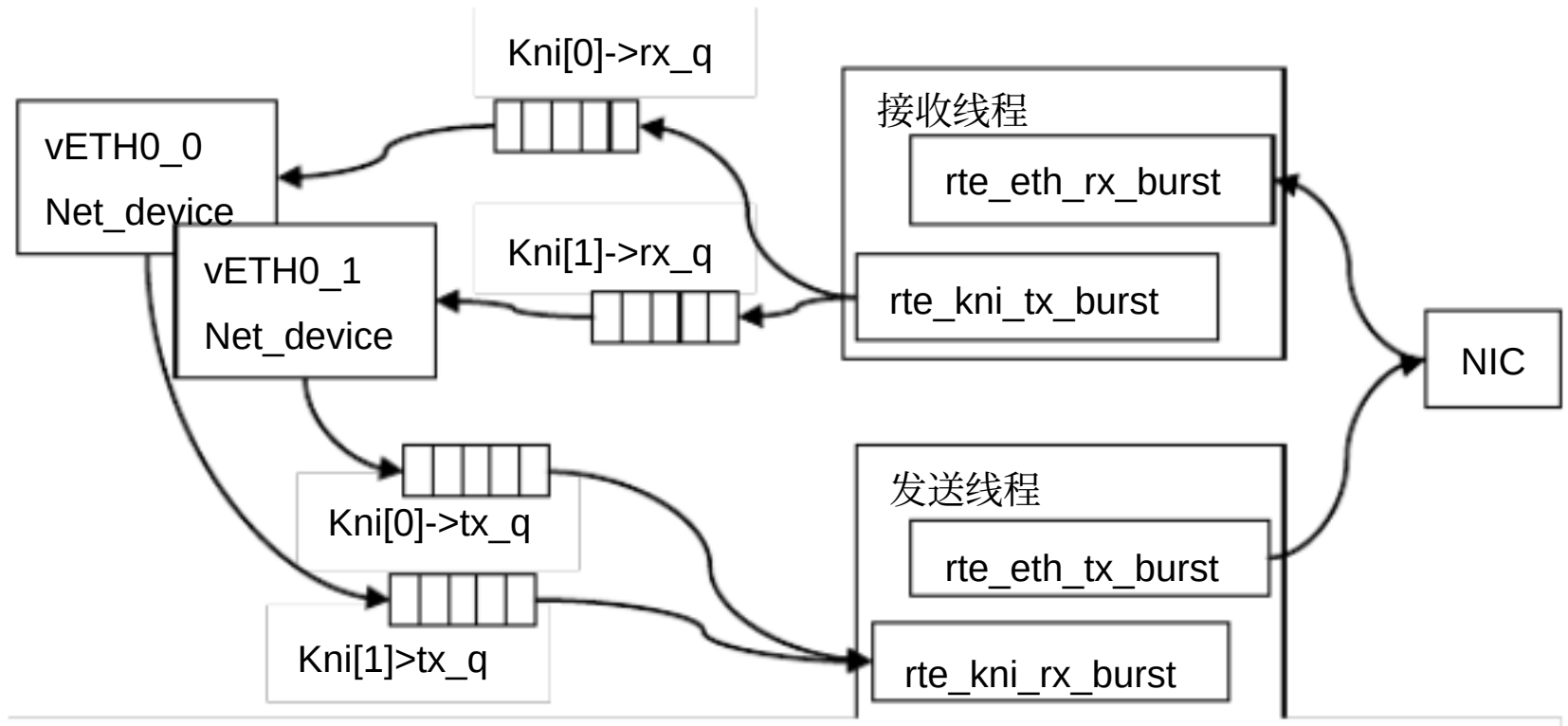
因为令牌桶结构体是放在 PIPE结构体中的， 而 grinder 的第二个阶段已经把 PIPE结构体数据加载到缓存中了，所在 DPDK 在 grinder 的第二个阶段更新有效令牌数而不是按一定的频率更新， 这也可以避免频繁更新令牌数导致 CPU利用率降低的问题。 由于 DPDK是在最后一个阶段才判断令牌数是否足够用，为以保证了判断令牌数之前令牌已经更新完成。 （其实为什么不在判断的时候才更新令牌数呢？ ）

KNI
内核线程数可以支持 single、 multiple ，默认是 single
单线程接收时， OPEN时统一创建一个全局内核接收线程 kni_kthread
多线程接收时，把每个接口的接收线程放在网口各自的 struct kni_dev-》pthread 里面，多线程接收时，全局变量 multiple_kthread_on 为 1
Loopback 模式， lo_mode_none 、 lo_mode_fifo 、 lo_mode_fifo_skb 默认是 lo_mode_none

每种模式的接收方式不一样，分别对应三个不同的接收函数（用一个统一的全局变量接口 kni_net_rx_func 指向三个不同函数）：
kni_net_rx_normal
kni_net_rx_lo_fifo
kni_net_rx_lo_fifo_skb
只有 kni_net_rx_normal 是真正把接收到的报文发送到应用层，其它两种方式是把报文在本虚接口中回环出去



KNI 的应用实中，在一个物理网卡上虚拟出多个虚拟设备，每个虚拟设备用于接收不同的业务流，但是实例中并没有做业务流的分类，只是简单把从物理网卡上接收到的报文轮流发送给不同的虚拟设备而已，其业务流程如下：



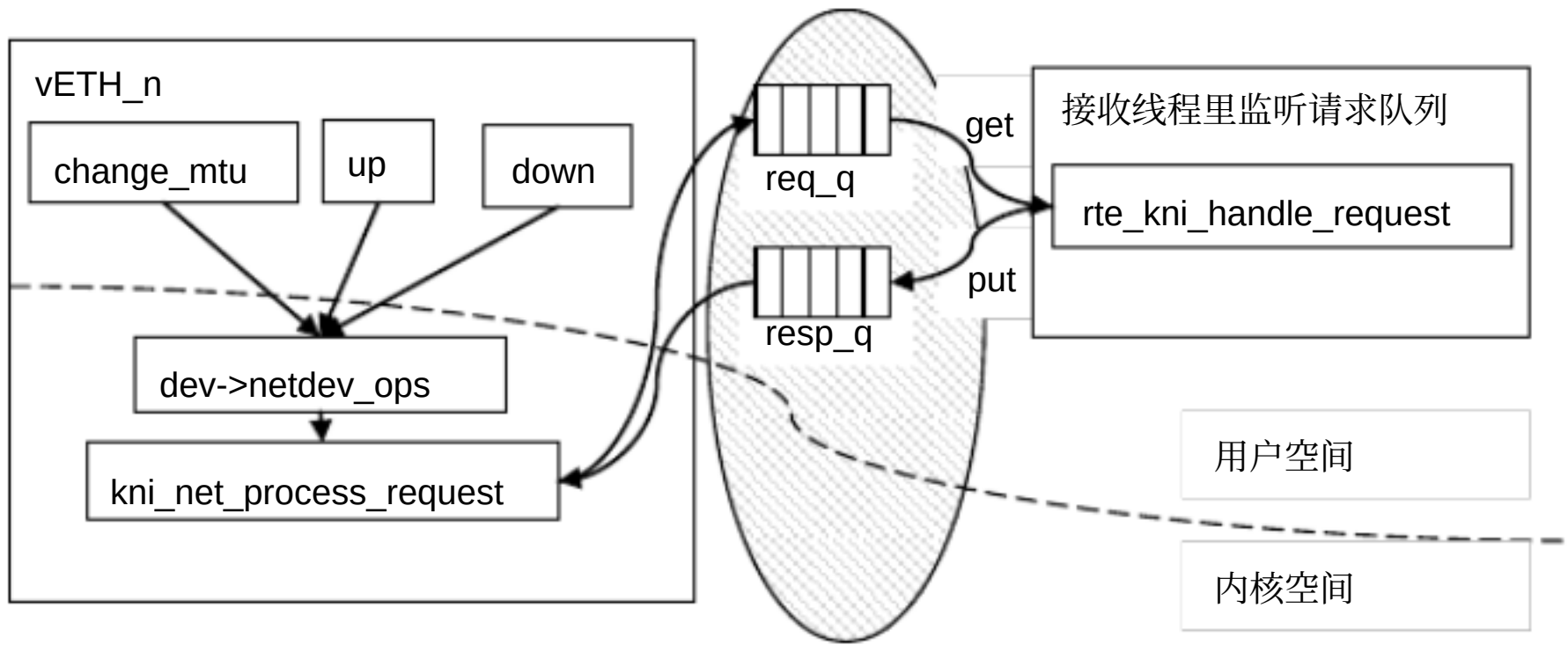
一个 struct rte_kni 代表一个虚拟设备，该实例只是把每个物理网口的报文，每批发送到不同的队列

KNI 实例流程图

1、 每一个物理网口分别用一个 DPDK 应用程序接收线程和一个 DPDK 应用程序发送线程专门处理。

2、 每个接收线程和发送线程只是简单转发报文。

3、 每个接收线程还负责处理内核的接口配置请求，如： MTU、 UP、 DOWN 接口的配置请求是通用（非 DPDK）应用程序发出，同内核放入请求队列中，等待 DPDK 应用线程处理（因为网卡驱动是在 DPDK 应用程序中实现的，所以对网卡的操作还是在应用程序中调用执行） ，其处理结构如下图：



每个虚拟设备都会有配置请求，一个 struct rte_kni 代表一个虚拟设备

DPDK为数据平面开发提供一个环境抽象层 （EAL）以及基于 EAL之上的一系列应用程序接口库和应用程序驱动库。其总体组件结构图如下图，其中绿色组件为系统的核心组件：

定时器系统

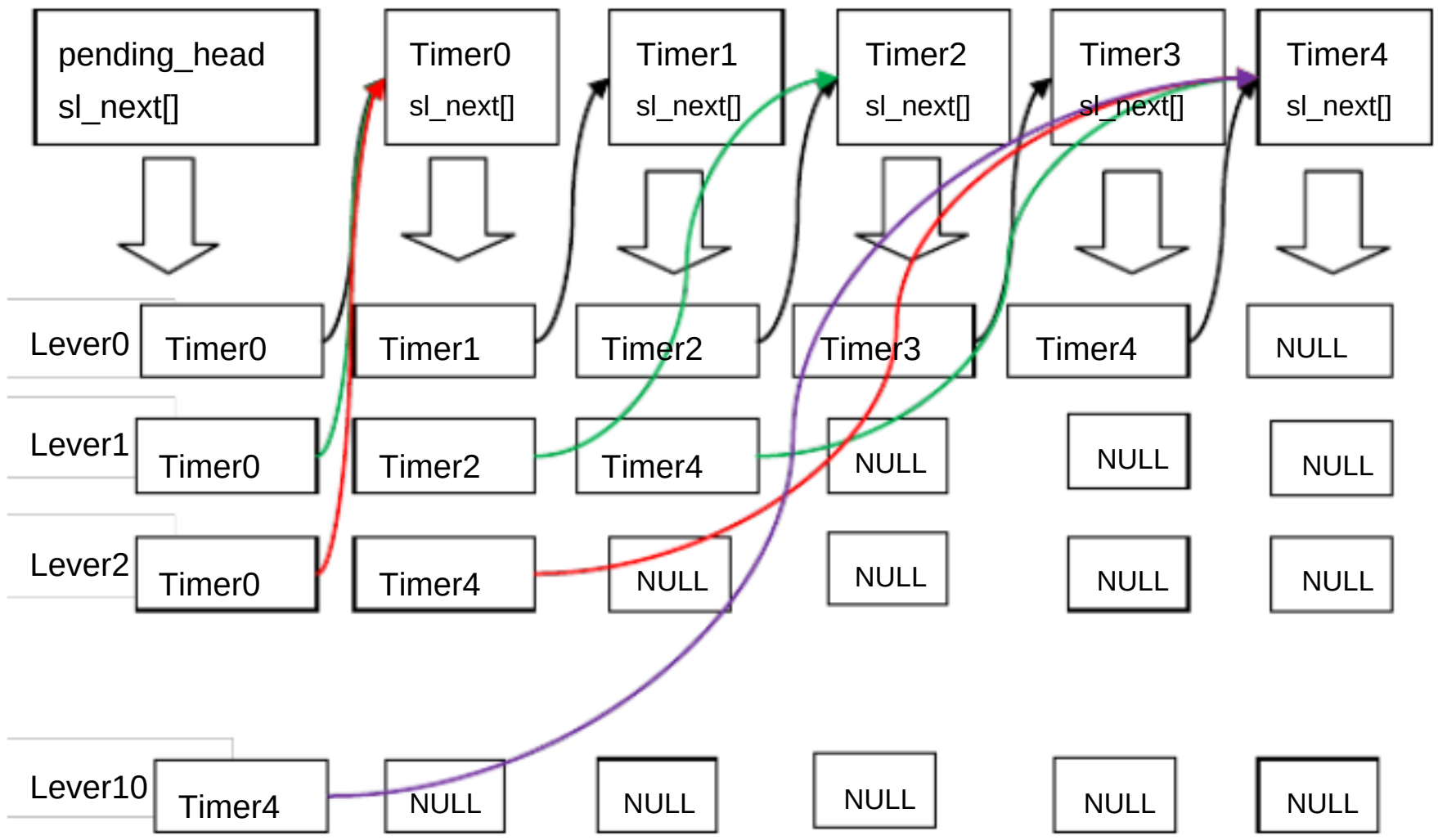
定时器系统提供一些异步周期执行的接口， 可以指定某个函数在规定的时间异步的执行， 就像 LIBC 中的 timer 定时器，但是这里的定时器需要应用程序在主循环中周期调用 rte_timer_manage 来使定时器得到执行，使用起来没有那么方便。

DPDK和定时器实现中，为每个核创建一个定时器列表，用于保存本核要执行的定时器。每个核的定时器列表用 10 个级别的队列来连接定时器， 每个定时器有 （1⁄4）^N 的概率落在第 N 个队列中。也就是说，对于队列 0，所有的定时器都可以连到队列 0；而对于队列 1，只

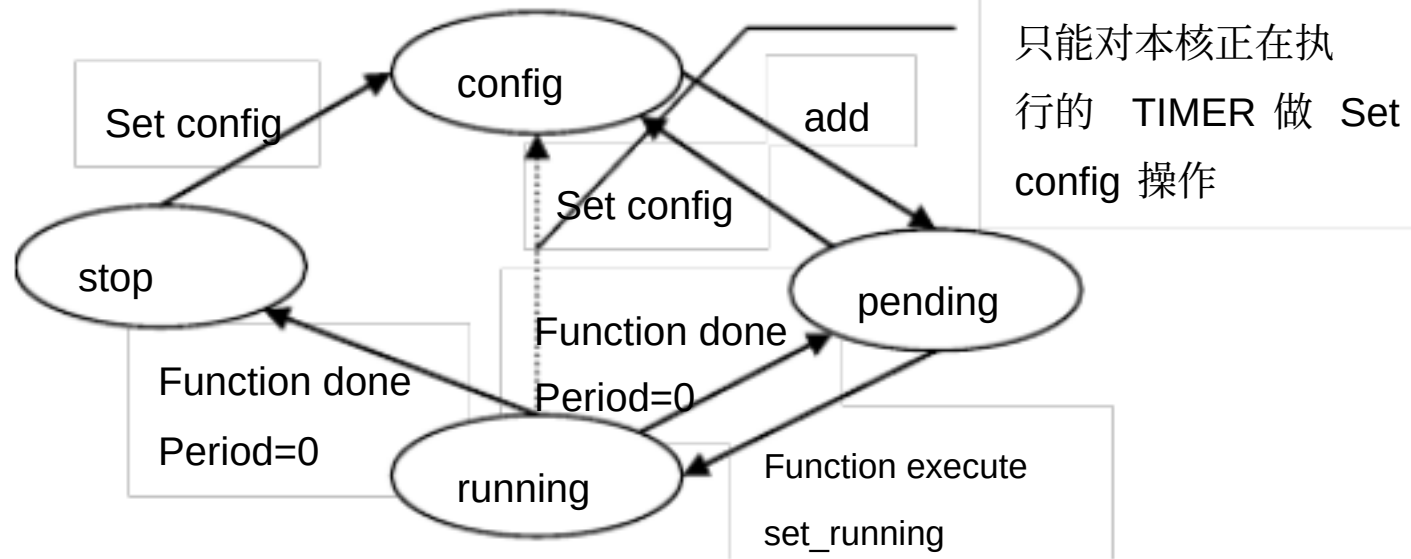
有 1/4 的定时器可以连接到队列 1；以些类推……
定时器结构如下：

```
struct rte_timer
{
        uint64_t expire;                /**< Time when timer expire.      下一次到期执行时间     */
        struct rte_timer *sl_next[MAX_SKIPLIST_DEPTH];
        volatile union rte_timer_status status; /**< Status of timer. */
        uint64_t period;                /**< Period of timer (0 if not periodic).       周期 */
        rte_timer_cb_t *f;              /**< Callback function.     回调函数  */
        void *arg;                      /**< Argument to callback function.       函数参数  */
};
```

其中， 每个定时器中的 sl_next[MAX_SKIPLIST_DEPTH] 就是用来表示本定时器在每个级别队列中的下一个定时器的指针，这样，通过每个定时器的 sl_next[N] 的值，就可以把定时器串连到不同的队列中了（这是跳表结构） 。
可以想像， 如果只有一个级别， 那么 sl_next[MAX_SKIPLIST_DEPTH]就只有一个元素了， 这样，定时器队列就和普通的链表结构一样只有一个 next 指针了。使用 10 个队列，就有 10 个 next 指针。
单个核的定时器队列结构如下图所示：

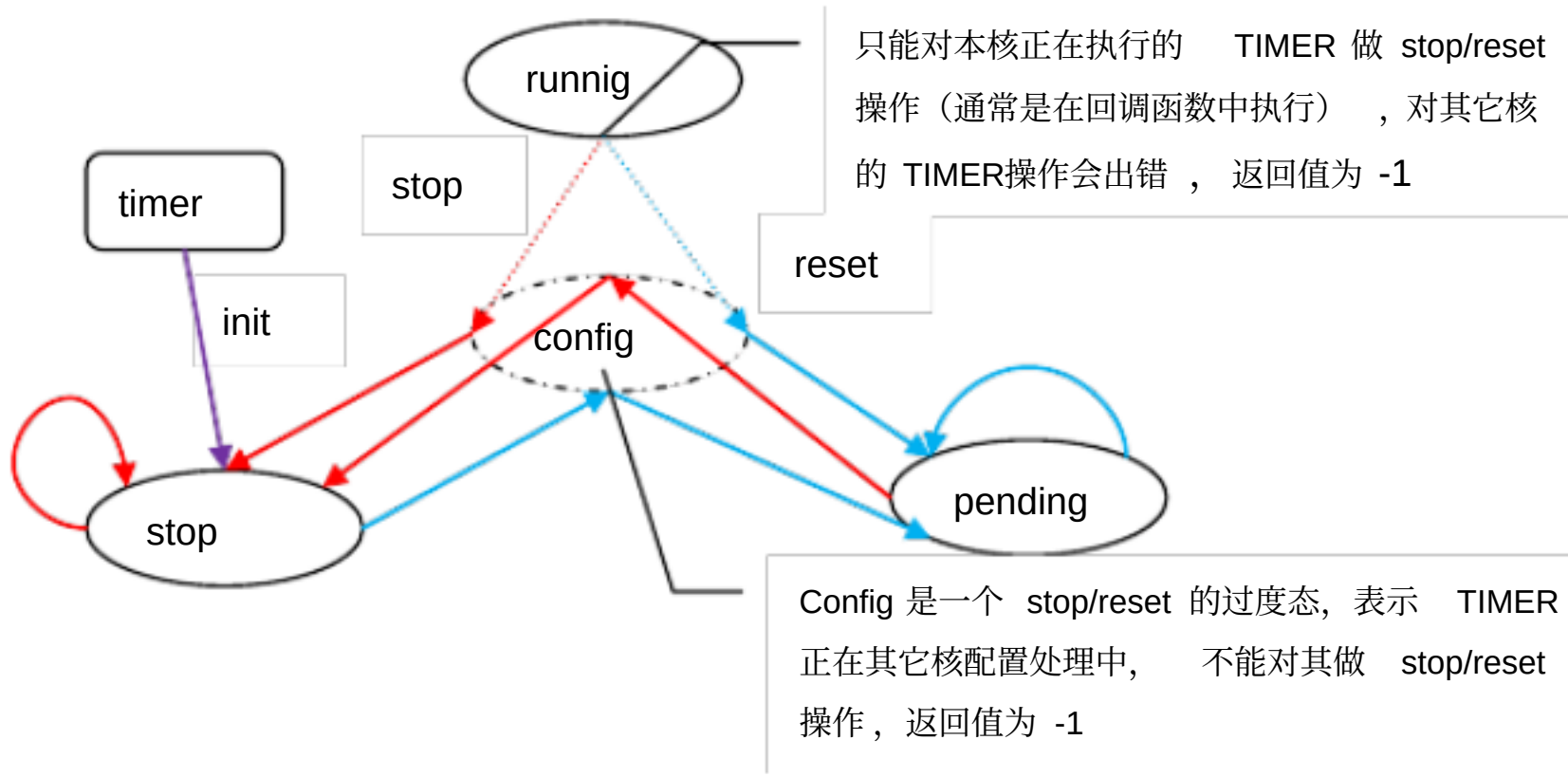


单个核的定时器列表结构图



内部状态转换图

应用程序调用 TIMER 库的 init/stop/reset 接口做定时器操作，其外部状态转换图如下：



外部状态转换图

Qos_sched –c 3 –n 2 -- --pfc “0,1,0,1” –mst 2 –cfg profile.cfg