# 关于DPDK Cryptodev，你不得不明白的几点！

原创 DPDK开源社区 2017-03-20

作者 张帆


安全已经成为组建服务器的必备需求之一，然而对网络报文的保护，如加密和认证运算等涉及非常复杂的数学运算，对服务器CPU或专用加速器等有非常高的要求。网络应用工程师在对网络报文的安全保障方面存在如下待解决的难题：

- 当今的以太网接口带宽已经达到100Gbits/s甚至更大，如何高效且低延迟地完成如此大量数据的加/解密及认证运算成为网络应用工程师最大的难题之一。
- 代码可复用难题。即便对同一类crypto算法，软件实现和硬件实现有很大的不同，导致应用在crypto实现的软件及硬件之间切换非常困难。
- 优化难题。crypto实现的优化能确保服务器的安全及专用加速器等硬件的高效利用，然而这一部分的难度较大，再加上各类crypto引擎优化方式的不同以及以上的代码复用问题，可谓雪上加霜。

## DPDK CRYPTODEV简介

DPDK的CRYPTODEV软件库及相应的驱动库就是为了解决以上难题而生。DPDK CRYPTODEV是DPDK的一个软件库，更是涵盖软/硬件CRYPTODEV引擎驱动的完整框架（目前暂不支持非对称加密算法）。它将提供不同算法的各类软硬件CRYPTODEV引擎的统一API，并对用户隐藏各类已高度优化的crypto实现细节。与之相对应的是DPDK的CRYPTO轮询模式驱动集，包含多种链式crypto/认证操作的实现。最后，DPDK CRYPTODEV还拥有非对称入队及出队的统一实现，来保证对硬件crypto操作效率的最优化。

DPDK CRYPTODEV目前支持的crypto算法有：

加密算法:
- AES-CBC/CTR, 128, 192, 和256 bit
- SNOW3G UEA2, Kasumi F8, NULL

认证算法：
- MD5_HMAC, SHA1, SHA224, SHA256, SHA384, 以及SHA512
- AES-XCBC
- SNOW3G UIA2, KASUMI F9, NULL

加密-认证算法：

- AES-GCM 128, 192,及256bit

DPDK CRYPTODEV使用了两种不同的形式来支持以上罗列的crypto算法: 软件PMD，使用特殊指令集优化的软件crypto实现，无需额外硬件，但某些PMD因使用了特殊指令集（如AES，AVX，SSE等指令集）对CPU架构有一定要求。每一个软件PMD仅支持一种或几种加密或认证算法；硬件PMD，要求系统加装了Intel QuickAssist (QAT) DH895xxC加速器。目前，QAT PMD支持除了NULL之外所有的加密和认证算法，且拥有更大的吞吐量。

## DPDK CRPYTODEV实现细节

和ETHDEV一样，DPDK CRYPTODEV将每一个加密引擎，不管是硬件PMD还是虚拟的软件PMD，抽象成一个设备（rte_cryptodev），记录了该设备支持的算法，最大队列的大小，最大支持缓存，以及enqueue和dequeue函数指针等信息。并对所有设备提供了通用的API，该API包含操作一个加密引擎所需的所有函数，如对设备的创建（仅针对软件PMD），初始化，开始或停止，enqueue或dequeue等。用户调用该API时，API的内部实现会调用对象PMD的具体实现来完成相应操作。此举有若干好处：

- 用户只需关注所用算法的应用而无需了解PMD实现的细节。
- 用户若想将他的代码的操作对象在软件PMD和硬件加速PMD之间转换，他无需修改或仅需极小修改（通常用于适配不同PMD支持算法的区别）就能实现。
- 甚至于，同样的代码可在物理系统与虚拟系统间随意转换。

用户应用和PMD之间的关系如图1所示。

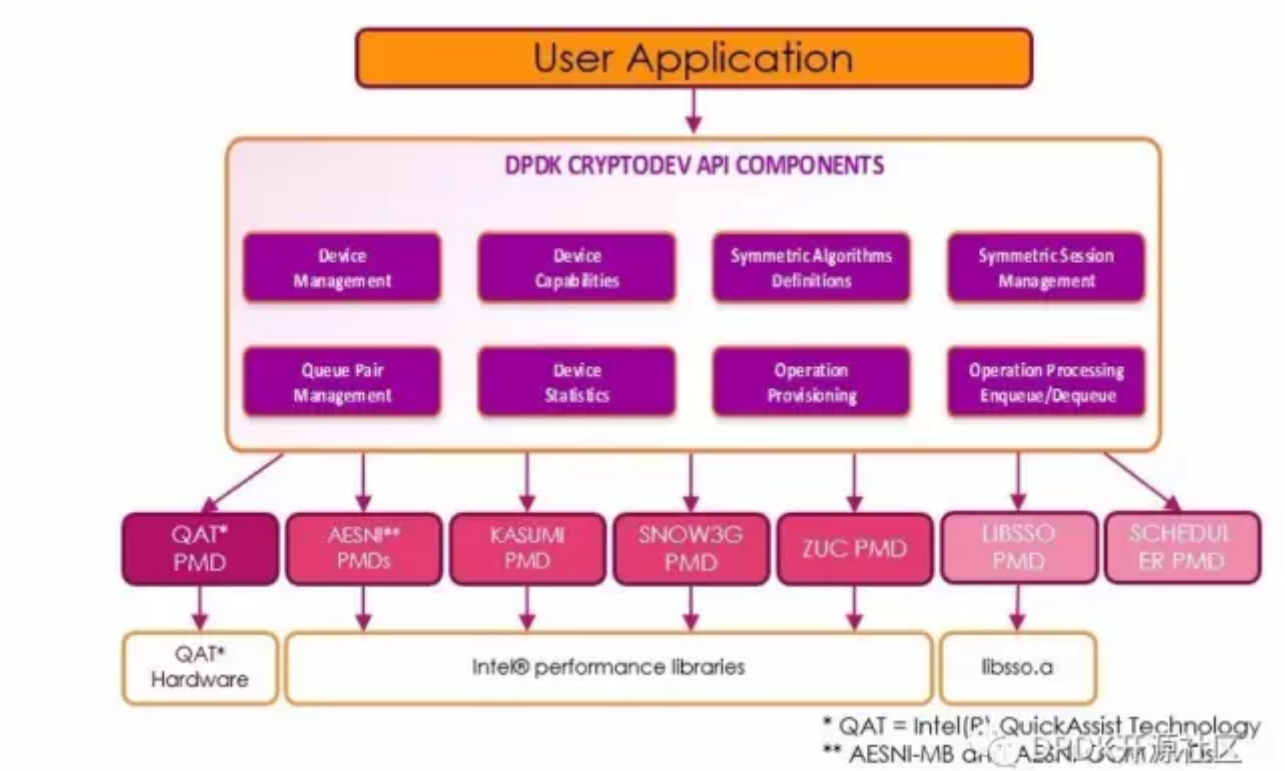


图1：DPDK CRYPTODEV系统结构图

DPDK Cryptodev的工作流程如图2所示。这里先不详细说明图里每一处的详情了，请大家在看到最后时，再回头看一下这幅图，印象能更深刻一些。

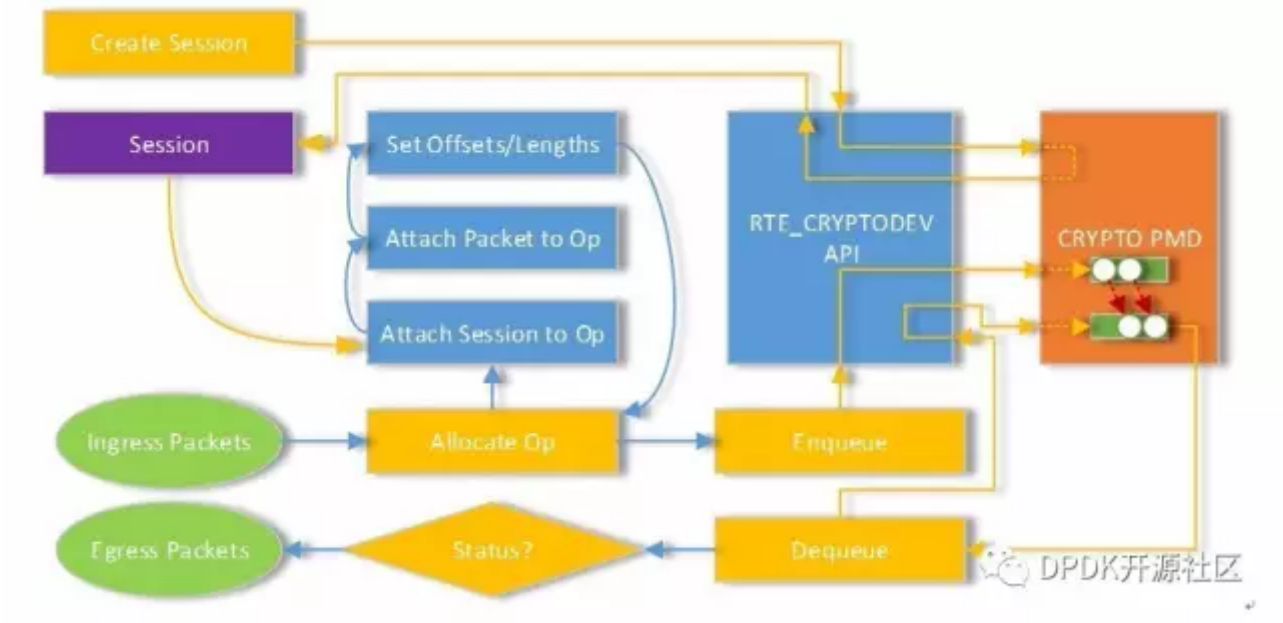



图2. DPDK Cryptodev 工作流程图

所有DPDK CRYPTODEV的API均罗列于lib/rte_cryptodev/rte_cryptodev.h中。介于篇幅限制这里不一一介绍，有兴趣的读者请参考源码和DPDK编程指南。我们来说一下DPDK CRYPTODEV的工作流程，在其中会介绍一些API的用法。

## 初始化

不管使用哪种PMD，用户首先需要在编译DPDK前修改dpdk/config/common_base文件下的选项来启用该PMD(缺省模式下除了null PMD，其他PMD均被禁用)。如要使用AESNI_MB，需要在该文件找到如下行：

CONFIG_RTE_LIBRTE_PMD_AESNI_MB=n

然后将“n”修改成“y”再编译DPDK。

同时，为了能编译成功，用户可能需要下载某些必要的库并使用宏指定其位置。具体信息请参考 http://dpdk.org/doc/guides/cryptodevs/index.html。

编译成功后，若使用软件PMD，用户可选择在DPDK的EAL命令行中使用vdev关键字指定，如以下的EAL命令行：

```
./l2fwd-crypto -l 11 -n 4 --vdev
"crypto_aesni_mb_pmd,name=aesni_mb0,max_nb_queue_pairs=2,max_nb_sessions=1024,socket_id=0"
```

以上命令行可为DPDK应用在socket 0上创建一个名为aesni_mb0的AESNI_MB的软件PMD，其拥有2个队列对，支持最大会话数为1024个。Cryptodev也和DPDK其他的虚拟设备vdev一样，支持在代码中动态创建该vdev。比如，以下代码和以上命令行有相同作用：

```
rte_eal_vdev_init("cryptodev_aesni_mb_pmd",
        "name=aesni_mb1,max_nb_queue_pairs=2,max_nb_sessions=1024,socket_id=0")
```

但当用户使用硬件加速（QAT）PMD时则必须将其和使用DPDK的NI一样绑定至IGB_UIO核心模块。和NIC一样，可使用whitelist或blacklist等EAL选项来过滤掉某些PMD的加载。初始化完成后，所有PMD均自动完成加载。关于Cryptodev和vdev和QAT初始化选项更详细的介绍，请参照DPDK编程指南。

## 设备配置

对设备的配置分为2个API，一个是设备配置，一个是队列对配置。

配置设备的参数被保存在如下结构中：

```
struct rte_cryptodev_config {
    int socket_id; /* 所在的socket */
    uint16_t nb_queue_pairs; /*队列对数量*/
    struct { /* 会话MEMPOOL的配置 */
        uint32_t nb_objs;    /* MEMPOOL的大小 */
        uint32_t cache_size; /* 每内核缓存的大小*/
    } session_mp;
};
```

这里需要说明的是，nb_queue_pairs需要小于或等于目标设备所能支持的最大队列对数量，session_mp的参数也需要在设备所支持最大会话数量范围之内酌情设定。每个设备的这些限制都保存在struct rte_cryptodev_info 结构之内。该结构内容如下：

```
struct rte_cryptodev_info {
    const char *driver_name; /* 驱动名 */
    enum rte_cryptodev_type dev_type; /*设备类型，也是PMD类型 */
    struct rte_pci_device *pci_dev; /* PCI设备信息句柄 */
    uint64_t feature_flags; /* 特性标志位，设备支持的特性 */
```

```
        const struct rte_cryptodev_capabilities *capabilities; /* crypto能力，包括支持的算法，
        数据，密钥长度等 */
    unsigned max_nb_queue_pairs; /* 最大队列对数量 */
    struct {
        unsigned max_nb_sessions; /* 最大会话数量 */
    } sym;
};
```

使用rte_cryptodev_info_get(dev_id, &dev_info)可获得某设备的具体信息内容。

要配置设备，需要申明并赋值一个rte_cryptodev_config结构，并将其和设备ID一并作为参数传递给rte_cryptodev_configure()函数中。该函数API如下：

```
extern int
rte_cryptodev_configure(uint8_t dev_id, struct rte_cryptodev_config *config);
```

## Queue Pair (队列对)配置

然后我们需要配置设备的队列对Queue Pair。与DPDK NIC的queue有所不同的是，DPDK Crytodev设备全部使用非对称操作模式，即对某批crypto保工作的提交并不期待在函数执行完成时就能完成。设备会从输入Queue里拿到工作需求，处理并完成后放进输出Queue里。当应用调用取出操作函数时，驱动会从输出Queue中提取尽可能多的获取已完成的工作。这样做能将Enqueue和Dequeue的开销均匀地分配到每个工作中，同时还能保证硬件卸载的提速功能能得到最大的发挥。我们把这个输入和输出Queue对称为队列对。

不同的设备支持的最大队列对数量不同。Qat是一个VF有2条Queue Pair（不可变），而软件PMD则是8条（在EAL选项中可以修改）。多Queue Pair可供多个线程使用而不会在Dequeue时拿到别的线程Enqueue的工作，保证系统延展性。

配置Queue Pair则相对简单，只需指定它所在的SOCKET和Mempool的大小即可。

```
extern int
rte_cryptodev_queue_pair_setup(
        uint8_t dev_id, /* 设备号 */
        uint16_t queue_pair_id, /* 队列对号 */
        const struct rte_cryptodev_qp_conf *qp_conf, /* 配置参数 */
        int socket_id);
```

其中struct rte_cryptodev_qp_conf 仅包含一个属性，及能存放多少个工作。

## 创建会话

创建会话的过程主要是告知设备自己想进行的crypto操作信息。如是想做AES-CBC的加密和HMAC-SHA1的Authentication生成工作等。想要知道目标设备所能进行的操作种类，用户可参考DPDK用户指南，或动态分析rte_cryptodev_info_get()函数所获得的info.capability结构指针。

想要创建会话，需要调用函数

```
struct rte_cryptodev_sym_session * rte_cryptodev_sym_session_create(
    uint8_t dev_id, struct rte_crypto_sym_xform *xform);
```

rte_crypto_sym_xform 结构如图3所示。其中，type指的是crypto类型，类型包括Crypto，以及Authentication。rte_crypto_cipher_xform和struct rte_crypto_auth_xform包含crypto和认证所需的各项参数，参数较多在这里就不一一介绍了，请参看rte_crypto_sym.h源码。rte_crypto_sym_xform 结构中的next指针是为了指定链式操作所设置的。某些Cryptodev支持单一操作，如加密或Authentication验证，有些还支持链式操作，如AES-CBC加密 + HMAC-SHA1生成，或HMAC-SHA256验证 + AES-CTR解密。使用该指针指向下一个xform即可实现链式操作的配置。

成功创建会话后，函数会传递回一个rte_cryptodev_sym_session结构的指针，用户需保存该指针留作后用。如果目标设备不支持指定的操作，则会返回一个空指针。

## 操作句柄的管理和获取

DPDK Cryptodev的最终操作对象是Crypto Operation，我们这里成为操作句柄。crypto密保所需的信息，如算法，会话，密钥，数据等的都需在操作句柄中被指定。和队列对一样，我们需要创建一个内存池来获取操作句柄。创建内存池的函数为：

```
extern struct rte_mempool *
rte_crypto_op_pool_create(
        const char *name, /*唯一名*/
        enum rte_crypto_op_type type, /*类型，目前仅支持对称类型 */
        unsigned nb_elts, /*内存池大小*/
        unsigned cache_size, /*缓存大小*/
        uint16_t priv_size, /*私有数据大小，没有可设为0*/
        int socket_id);
```

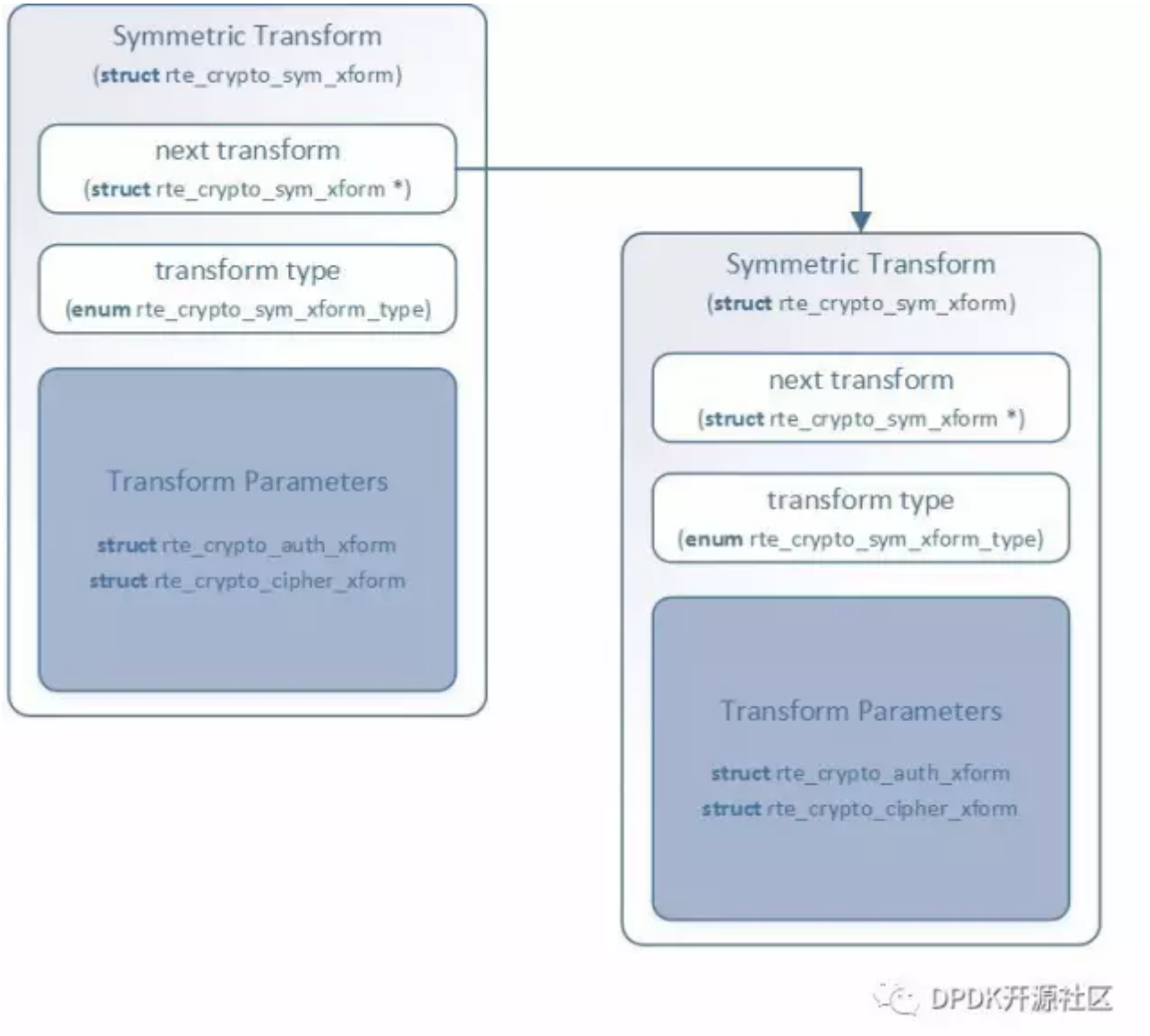



图3. rte_crypto_sym_xform 结构

函数运行成功后会返回一个rte_mempool指针，然后可使用rte_crypto_op_alloc()或rte_crypto_op_bulk_alloc()函数从该内存池中取出一个或多个操作句柄。

然后我们则需要配置每个句柄，链接会话（使用 rte_crypto_op_attach_sym_session（ ）函数即可，配置对象数据，密钥的指针，长度，以及物理地址等。详细配置方法请参照l2fwd_crypto范例程序的源码。然后我们就能将其传递给设备让其开始作业了。

## 操作句柄入队和出队

和向DPDK的Ethdev传递网络帧Mbuf一样，向Cryptodev设备提交作业的方式也是异步形式，以批为单位进行Enqueue和Dequeue。Enqueue和Dequeue的API如下：

```
uint16_t rte_cryptodev_enqueue_burst(
        uint8_t dev_id, /* 设备ID */
        uint16_t qp_id, /* 队列对ID */
```

```
        struct rte_crypto_op **ops, /* 操作句柄数组指针 */
        uint16_t nb_ops /* 操作句柄的个数 */);

uint16_t rte_cryptodev_dequeue_burst(
        uint8_t dev_id, /* 设备ID */
        uint16_t qp_id, /* 队列对ID */
        struct rte_crypto_op **ops, /* 操作句柄数组指针 */
        uint16_t nb_ops /* 操作句柄的个数 */);
```

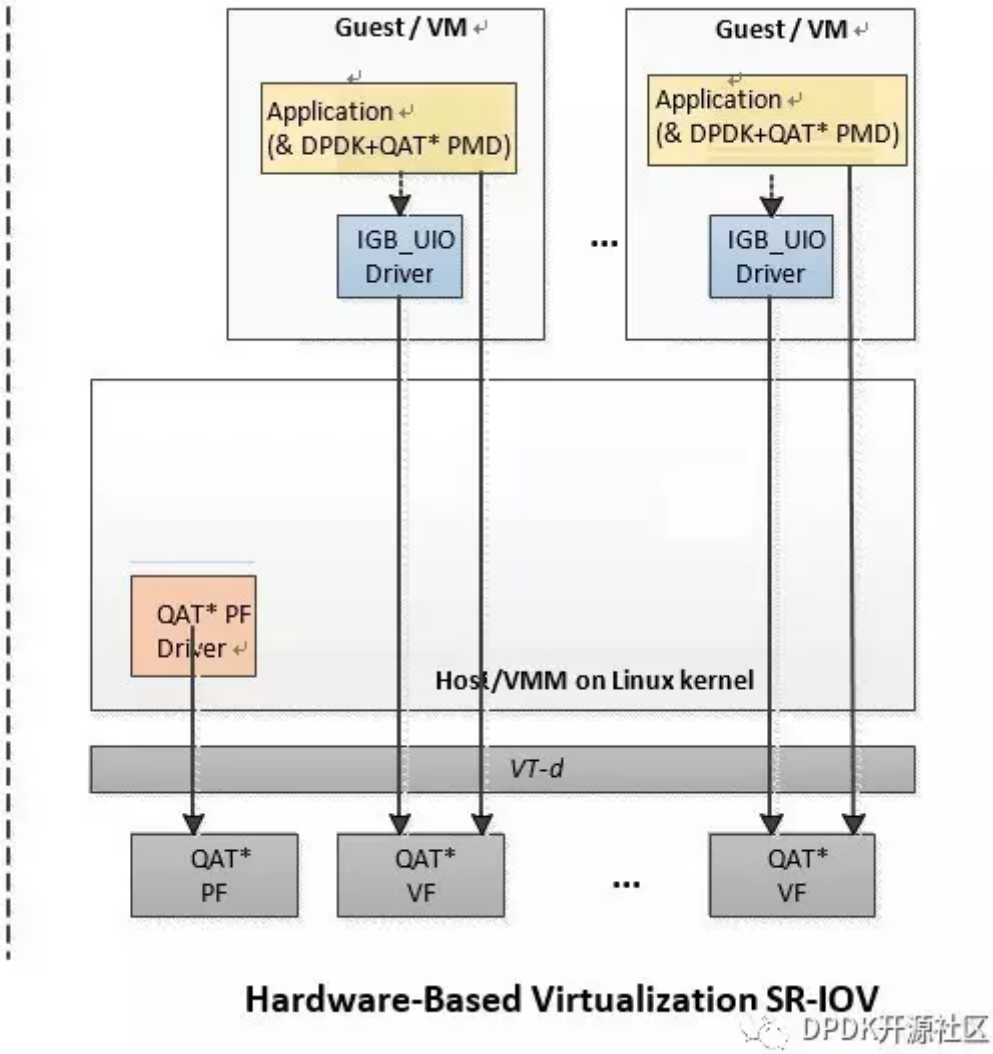
单次Enqueue和Dequeue的开销较大，所以我们希望通过一次尽量传递更多的工作数量来减少Enqueue和Dequeue的次数。但是也不能一次传递太多，因为这样一来设备和Queue Pair的缓存得有足够大来存放更多的操作数，二来操作延迟也将被放大不少。可以看到，不管是DPDK中的L2fwd_crypto范例还是Cryto性能测试App, nb_ops都被设为不多余32。

## QAT PMD的PF和VF使用场景

DPDK Cryptodev的QAT PMD拥有PF和VF两种使用场景，为多虚拟机高效共用同一个QAT硬件设计了一个非常方便的解决方案。

当将一个QAT设备由DPDK Cryptodev驱动时，系统可见两种驱动，其中一类是PF驱动，运行于VMM或HOST上，可用于管理所有的VF。第二类是VF，VF可以有很多个，每个都拥有2个Queue Pair，可分配给VM直接访问。VF可被VM直接检测并绑定在DPDK的IGB_UIO驱动之上，这样每个VM都会拥有一个虚拟的QAT硬件并可像在HOST一样直接进行操作。如图4显示。

这样做的好处自不必说，同样的应用实现可在虚拟机和物理机之间随意切换且无需修改代码，不会损失QAT的性能。




## 总结

在这篇文章我们介绍了DPDK Cryptodev，包括如何使用它。DPDK Cryptodev还很年轻，从诞生到现在刚2年时间，还有很多地方需要改进。希望这篇文章能帮助您更好地了解DPDK Cryptodev，也希望您能多多提问，让DPDK Cryptodev更加完善。若想了解更多，请登录dpdk.org网站或者发邮件到dev@dpdk.org或user@dpdk.org参与讨论。非常感谢！

注1：DPDK CRYPTO编程指南：
http://dpdk.org/doc/guides/prog_guide/cryptodev_lib.html

作者简介：张帆，爱尔兰利莫里克大学计算机网络信息学博士，湖南省湘潭大学兼职教授，现为英特尔公司爱尔兰分部网络软件工程师。近年专著有Comparative Performance and Energy Consumption Analysis of Different AES Implementations on a Wireless Sensor Network Node等。发表SCI/EI 检索国际期刊及会议论文3 篇。目前主要从事英特尔DPDK 在SDN应用方面的扩展研究工作。

## “DPDK开源社区”精品文章

玩物志 | 什么！DPDK在盒子里？

基于virtio-user的新exception path方案

DPDK Release 17.02

Hyperscan Release 4.4.0

DPDK Release 16.11

无锁队列详细分解——Lock与Cache，到底有没有锁？

从计算机架构师的角度看DPDK性能

欢迎搭乘Hyperscan号极速列车~

无锁队列详细分解 — 顶层设计

VMware Player 搭建DPDK实验平台

Qemu/Kvm 搭建DPDK实验平台技术贴：利用DPDK加速容器网络

END


干货满满的公众号

长按指纹识别二维码关

阅读原文 投诉