odp Documentation

Release 0.99

lockless

May 26, 2017

Contents

CHAPTER 1

OpenDataPlane (ODP) 用户指南

摘要

本文档旨在指导新的ODP应用程序开发人员。 有关ODP的更多详细信息，请参见 ODP 主页。

Fig. 1.1: Overview of a system running ODP applications

ODP是一份API规范，为高性能网络应用程序的实现提供平台独立性、自动硬件加速和CPU扩展。 本文档介绍如何充分利用API的优势来编写应用程序。

简介

Fig. 1.2: OpenDataPlane Components

ODP API 规范

ODP由三个各自独立又相关联的组件组成。首先，ODP是一个抽象的API规范，用于描述数据面应用程序的功能模型。 该规范涵盖了许多常见数据面应用程序的编程需求，如接收、操作、传输数据包，但是并不要求这些功能具体如何实现。 这是非常有意义的，正因为ODP API没有一个优选实施例，它们允许在支持ODP实现的各种平台上运行ODP，并进行各种创新开发。 为了实现这一目标，ODP API使用了抽象数据类型描述，其定义留给ODP的实现者。 例如，ODP数据包由类型为 `odp_packet_t` 的抽象句柄引用，与数据包相关的API会引用此类型的参数。 这就是为什么 `odp_packet_t` 不是ODP API的一部分，这是实现者的责任。

ODP API 特性

- 开源，开放贡献，BSD-3许可
- 供应商与平台中立

- 以应用为中心，涵盖数据面应用的功能需求
- 通过指定ODP的功能行为确保可移植性
- 由应用程序作者和平台实施者共同公开界定
- 架构可以有效在各种平台上实现
- 由Linaro集团赞助、管理和维护

ODP API 实现

其次，ODP由这个API规范的多个实现组成，每个实现都针对特定的目标平台进行定制。 ODP的具体实现确定每个ODP抽象类型在该平台上的标识方式及每个ODP API是如何实现的。 在某些平台上，ODP API将使用某些具有加速API功能的特殊执行来实现。 另一方面，硬件协处理器引擎可能会完全卸载API，以便CPU可以很少或者完全不参与执行。 但是，不论什么情况，应用程序都会看到相同的功能行为，而不依赖于给定平台选择如何实现它。 通过允许每个平台自由地确定如何以最佳方式实现每个API的指定功能行为，ODP运行应用程序充分利用每个平台的独特功能， 而不需要具有该平台的专业知识，或者关心如何最好的将应用程序调整到特定平台。 后一种考虑在网络功能虚拟化（NFV）环境中尤为重要，其中，应用将在其他人选择的目标平台上运行。

ODP 实现特性

- 通用实现并不适用于所有平台，支持多个实现使得ODP适应于平台之间的差异性。
- 任何人都可以根据自己的平台创建一个ODP实现
- 每个实现的开发和维护由拥有者决定
  - 根据业务需求确定开源和闭源
  - 具有独立的发布周期和服务流
- 允许在平台的ODP API实现上使用HW和SW的新特性

参考实现

为了轻松开始在新平台上实现ODP，ODP提供了许多实现方案，作为参考。 其中两个主要的实现是 `odp-linux` 和 `odp-dpdk` 。

odp-linux

odp-linux 实现是仅依赖于Linux系统API的纯SW实现。作为ODP的功能模型，它能方便的将ODP引导到任何支持Linux内核的平台上。

odp-dpdk

odp-dpdk 实现是使用DPDK作为SW加速器的纯SW实现。特别地，odp-dpdk为使用NIC的系统提供卓越的IO性能，这允许ODP应用程序充分利用 DPDK支持的各种NIC设备驱动程序。

*odp*参考实现特性

- 开源，开放贡献，BSD-3许可
- 提供ODP在新平台上的轻松引导
- 实施者可根据需要自由借用或定制代码
- 是否来自参考实现，实施者可以保留对其实现的完全控制。

ODP 验证套件

第三，为了保证不同ODP实现之间的一致性，ODP包含一个验证套件，用于验证ODP的任何给定实现是否忠实的提供每个API指定的功能行为。 作为单独的开源组件，应用程序编写者，系统集成商和平台提供商都可以使用验证套件来确认任何所谓ODP实现是否符合ODP API规范。

*odp*验证套件特性

- 与ODP API规范同步
- 由LNG维护和分配
- 开源，开放贡献，BSD-3许可
- 是确保所有ODP实现中的应用程序可移植性的关键
- 测试ODP实现是否符合ODP API的指定功能行为
- 用户和供应商可以随时运行，以验证ODP的实现

ODP API SPEC 版本

作为不断发展的标准，ODP API SPEC以增量版本发布，ODP相应的实现以及验证API一致性的验证套件与此版本号关联。 ODP版本号使用一个标准的三级数字（major.minor.fixlevel）进行指定，根据级别所表示的变化程度递增。

- fixlevel级别的增量表示规范的说明或其他不影响规范的语法或语义的次要更改。API规范中的这种变化预计将是罕见的。
- minor级别的增量代表引入新的API或功能功能，或者对其指定API的语法或功能行为的更改，因此可能需要应用程序源代码更改。 在规范的每个版本的发行说明中都对这些更改进行了详细的记录。
- major级别的增量代表了很大的结构性变化，这些变化最有可能需要一定程度的应用程序源代码更改，再次在该版本的发行说明中记录。

ODP 实现版本

ODP实现可以随意使用他们希望的任何发布命名/编号约定，只要清楚给定的版本实现了什么级别的ODP API。 推荐使用相同的三级编号方案，其中major和minor对应于ODP API相应级别，fixlevel表示与该API级别实现相关联的实现定义的服务级别。 LNG提供的ODP参考实现遵循这个原则。

ODP 测试套件版本

ODP验证测试套件遵循这些相同的命名规则。major和minor对应于套件验证的API级别，fixlevel代表该API级别的验证套件本身的服务级别。

ODP 设计目标

ODP组件结构有三个主要目标。

第一个是应用程序在各种平台上的可移植性。 这些平台在处理器指令集架构，应用处理核心数目和类型，内存组织以及平台特定的硬件加速和卸载功能都不一样。 ODP应用程序可以从一个一致实现方便转到另一个，最多只需要进行重新编译。

第二，ODP旨在允许数据面应用程序充分利用平台的特性，包括专门的硬件加速器，而无需专门的编程。这是通过将API规范与其在各个平台上的实现分开来实现的。 由于每个平台以对该平台最佳的方式实现每个ODP API，所以应用程序自动获得这种优化的优点，而不需要显式编程。

第三，ODP旨在允许应用程序自动扩展以支持多核心架构。 这是使用基于事件的编程模型完成的，该模型允许将应用程序写入独立于可用于实现应用程序功能的处理核心数量。 结果是写入此模型的应用程序不需要重新设计，因为它支持从4到40核到400核心。

文档组织

本文档分成以下几个部分。

第一部分介绍了ODP应用程序，ODP API组件区域及其相关抽象数据类型的高级概述。这一部分在概念层面介绍ODP API。

第二部分提供了ODP支持的编程模型的教程，特别注意事件模型，因为它代表了大多数ODP应用程序的首选结构。 这一部分基于第一部分介绍的概念，并展示了ODP应用程序的结构，以便最好地实现前面提到的三个ODP设计目标。

第三部分提供了主要ODP API组件的更详细的概述，旨在作为每个API的完整参考规范的伴侣。 后者旨在由ODP应用程序员以及实现者使用，以了解每个API的精确语法和语义。

ODP 应用程序和报文流

数据面应用程序从根本上说涉及接收、检查、操作和传输数据包。 数据面的显著特点是这些应用程序主要关注ISO堆栈（L2和L3）的最底层，并且他们具有非常高的性能要求。 ODP旨在为这些应用程序提供便携式框架。

从最上层看，ODP应用程序是使用一个或多个ODP API的程序。由于ODP是一个编程框架，而不是编程环境， 所以，应用程序可以自由地使用其他非ODP API。

ODP应用程序的作用和运行方式各不相同，但总体来说具有如下特点：

1. 它们被组织成一个或多个并行执行的线程。
2. 这些线程使用各种同步机制来传达和协调其活动。
3. 它们从一个或多个数据包I/O接口接收数据包。
4. 他们检查，转换或以其他方式处理数据包。
5. 它们将数据包传输到一个或多个数据包I/O接口。

ODP应用程序的顶层视图如下所示：

Fig. 1.3: ODP Application Packet Flow Overview

数据包到达并从由PktIO抽象表示的网络接口接收（RX）。 然后，它们直接转到由ODP线程轮询的队列，或者可以通过分类器Classification和排序，进入到代表各个流的队列。 接下来就可以通过调度器Scheduler将这些队列分派到应用程序线程。

线程，术语描述为可以调用各种ODP API来处理数据包内容。 对于输出处理，报文可以通过直接排队到PKTIO输出队列，或者可以在TX之前将其交给TM进行QoS处理。 注意，输出接口可以在loopback模式下工作，在这种情况下，发送给他们的数据包被重新回环到输入端进行“第二次”处理。 例如，传入的IPSec报文在解密之前无法正确分类，因此必须回环回来进行第二次处理。一旦解密，其实际内容可见，则可以将其分配到对应flow上。

需要注意的是，上述图表中唯一需要操作的是黄色部分。 这里显示的其他内容都由ODP提供，可供任何ODP应用程序使用。 这代表了数据面应用程序的“机械”，并且被构造为允许写入ODP API的应用程序可以在提供ODP实现的每个平台上进行移植和优化，而无需额外的工作。

ODP API 组件

ODP程序围绕几个概念进行构建，每个开发者都应该熟悉这些概念。 主要的概念主要的是：线程、事件、队列、池、共享内存、缓冲区、数据包、PktIO、定时器和同步器。

线程（Thread）

线程是ODP中的基本编程单元。ODP应用程序被组织成执行设计工作的线程集合。 ODP线程可能或者可能不会与其他线程共享内存，这取决于具体实现。 线程有两种类型：控制线程和工作线程，他们由抽象类型 `odp_thread_type_t` 表示。

控制线程是组织工作线程工作的监督线程。 而工作线程则负责执行应用程序的主要逻辑，它采用的是RTC模型。 特别的是，工作线程一般运行于专用的处理核心上，特别是在多核心的处理环境中。 但是，如果需要，给定的实现可以在单核上运行（通常在较小和较低性能目标环境上）。

除了线程类型，线程还具有关联属性，例如，线程掩码和调度程序组，确定他们可以在哪里运行，以及他们可以处理的工作类型。

事件（Event）

事件是线程执行工作的过程。事件可以表示新的工作，如需要处理的数据包到达，或者他们可以表示异步执行的请求完成。 事件还可以表示通知时间，或应用程序感兴趣的各个组件状态的更改。 事件有一个描述代表它的事件类型。 线程可以创建新事件，或消耗由他们处理的事件，或者可以对事件执行进一步处理，然后将事件传递给另一个组件以进行其他处理。 对事件的引用是通过抽象类 `odp_event_t` 句柄实现的。提供了专用函数将他们转换成由事件表示的适当类型的特定句柄。

队列（Queue）

队列是保存事件的消息传递通道。 事件可以通过入队操作添加到队列中，或者通过出队操作从队列中删除。 队列的端点将根据使用方式而有所不同。 队列有两种主要类型：轮询和调度，这将在引入事件模型时更详细的讨论。 队列也可能具有关联的上下文，这表示所有使用它的事件的持久状态。 这些状态是允许线程对事件进行有状态处理以及无状态处理。

队列由抽象类型 `odp_event_t` 表示。

池（Pool）

池是元素存储的共享内存区域。Pool代表了事件及其他东西的后备存储。 池通常在应用程序初始化和终止期间内创建和销毁，在程序处理过程中被调用。 池可以由专门的ODP组件或专门的应用程序使用，或者两者共享使用。 池具有描述他们包含的元素的关联类型。两个最重要的池类型是缓冲区（buffer）和数据包（packet）。

池由抽象类型 `odp_pool_t` 表示。

共享内存（Shared Memory）

共享内存表示在线程之间共享的原始存储块。 他们是池的构建块，但是如果需要，可以直接由ODP应用程序使用。

共享内存由抽象数据 `odp_shm_t` 表示。

缓冲区（Buffer）

Buffer是ODP组件和应用程序用于实现其功能的固定大小的共享存储块。 Buffer包含0个或多个字节的应用程序数据以及提供有关Buffer信息的系统维护Metadata（例如buffer大小，从哪个pool分配）。 Metadata是一个重要的ODP概念，因为他允许任意数量的辅助信息与ODP对象相关联。 大多数ODP对象都有相关联的元数据，并且该元数据通过访问器函数进行操作，该函数用作此数据信息的getter和setter。 Getter操作允许应用程序读取Metadata，而Setter操作允许应用程序写入Metadata。 请注意，一些Meatdata本质上是只读的，因此没有提供Setter操作。 当对象具有多个Metadata时，每个Metadata都有自己关联的Getter和Setter访问操作。

Buffer由抽象数据类型 `odp_buffer_t` 表示。

数据包（Packet）

Packet是指通过IO接口接收和发送，并表示数据面应用程序操作的基本数据。 Packet来自 `ODP_POOL_PACKET` 类型的Pool。 与简单的Buffer不同，ODP Packet具有丰富的语义，允许以复杂的方式进行检查和操作，这些将在后面描述。 Packet还支持丰富的Metadata以及User Metadata。 User Metadaa允许应用程序将确定的副信息量与每个Packet相关联以供自己使用。

Packet由抽象类型 `odp_packet_t` 表示。

报文接口（PktIO）

PktIO是ODP表示IO接口的方式。 PktIO对象是能够接收（RX）和发送（TX）报文的逻辑端口。 这可以由基础平台作为集成功能直接支持，或者可以表示通过PCIE或其他总线连接的设备。

PktIO由抽象类型 `odp_pktio_t` 表示。

时间（Time）

时间API用于测量应用程序的时间间隔和跟踪事件流程，并提供了访问时间源的便利方式。 事件API由两个主要部分组成：Local time API和Global time API。

局部时间（Local time）

Local time API被设计为在一个线程内使用，并且可以比Global time API更快。 由于时间一致性不能保证，Local time API不能在线程间使用。 本地时间戳是调用线程本地的，不能与其他线程共享。 当前本地时间可以通过接口 `odp_time_local()` 获取。

全局时间（Global time）

Global time API被设计为用于跟踪线程之间的时间。 所以，全局时间戳可以在线程之间共享。 当前全局时间可以通过接口 `odp_time_global()` 获取。

时间API包括随时间运行的功能，例如 `odp_time_diff()` `odp_time_sum()` 和 `odp_time_cmp()` 。 时间转换函数 `odp_time_to_ns()` , `odp_time_local_from_ns()` , `odp_time_global_from_ns()` 。 要获取时间源使用接口 `odp_time_local_res()` , `odp_time_global_res()` 。 要等待，使用接口 `odp_time_wait_ns()` 和 `odp_time_wait_until()` 在线程繁忙期间循环等待 。

`odp_time_t` 类型表示本地或全局时间戳。

定时器（Timer）

Timer指应用程序如何衡量和相应时间的流逝。 Timer从具有自己的抽象类型 `odp_timer_pool_t` 的专用池（定时器池）中申请出来。 应用程序可能同时具有许多定时器活动，并可将其设置为使用相对或绝对时间。 当定时器到期时，他们会创建类型为 `odp_timeout_t` 的事件，这些事件作为定时器到期的通知。

同步器（Synchronizer）

并行运行的多个线程通常需要各种同步服务，使得他们以可靠和协调的方式运行。 ODP提供了一组丰富的locks、barries，和类似的同步原语，以及用于表示各种类型的原子变量的抽象类型。 ODP事件模型还使用队列来避免在许多情况下显式加锁的需要。 这些内容将在下一节讨论。

ODP 组件

基于ODP概念，ODP提供了与ODP应用程序的工作流程相关的多个组件。 这些组件包括分类器，调度程序和流量管理器等。 这些组件与数据包处理的三个主要阶段有关：接收，处理和发送。

分类器(Classifier)

分类器提供了一套控制数据包接收（RX）处理的API。

Fig. 1.4: ODP Receive Processing with Classifier

Classfier提供两个逻辑相关的服务：

- **Packet parsing** : 检查并从接收的数据包中提取结构信息
- **Packet classification** : 将模式匹配规则PMR应用于解析结果，以将传入的数据包分配给服务等级CoS

综合起来，这些允许进入的数据包被分类成流，这些流是具有共同处理要求的逻辑上相关的数据包系列。虽然许多数据面应用程序执行无状态报文处理（如简单转发），但是也有很多数据面程序执行有状态报文处理。 流的状态信息与这些报文组的状态相关。

CoS确定属于某条流的数据包的两个属性：

- 收到报文时存储的池
- 报文将被添加进去的处理队列

ODP支持的PRMs允许基于报文字段值组合（tuples）的流分类。分类的主要优点是在许多平台上，这些功能是以硬件方式执行的， 这意味着当数据包正在被接收，而ODP应用程序没有任何明确处理的情况下，分类是以线速发生的。

Note: Classfier的使用是可选的。如果选择使用，应用程序可以通过直接轮询的方式直接从对应PktIO输入队列接收数据包。

Fig. 1.5: ODP Scheduler and Event Processing

调度器(Scheduler)

Scheduler提供了一套控制可扩展事件处理的API。

Scheduler负责选择和调度一个或多个事件到请求的线程。 事件选择基于几个因素，涉及包含可调度事件的队列和进行 `odp_schedule()` 或 `odp_schedule_multi()` 调用的线程。

ODP队列具有调度优先级（scheduling priority），可以确定应相对于其他队列上的事件，当前队列上的紧急事件如何处理。 队列还具有与他们相关联的调度程序组的ID，他们必须与调用程序的线程的关联调度程序组线程掩码想匹配。 这允许事件被分类处理，并且具有专门用于处理某一类事件的线程。 线程可以动态地加入或离开调度程序组，从而允许应用程序的相应更快地满足需求。

当线程从shceduler接收到事件时，它又可以通过异步操作的ODP API（如加密处理）来调用其他处理引擎。 当这样的处理完成时，完成的事件被添加回可调度队列中，这个事件可以被再次调度回线程，以使用异步操作的结果继续处理。

线程本身可以将事件排入队列，以供其他线程进行下游处理，从而允许应用程序自身结构最大化并发处理。

流量管理(Traffic Manager)

Traffic Manager提供了一整套API用于控制数据包输出的流量整形和服务质量QoS处理。

Fig. 1.6: ODP Transmit processing with Traffic Manager

报文处理的最后阶段是发包。 此时，应用程序有几个选择。 与RX处理一样，应用程序可以将数据包直接发送到PktIO TX队列进行直接传输。 但是，通常情况下，应用陈旭需要对包括作为发送处理的一部分流的分组执行流量整形和相关服务质量QoS处理。 为了满足这一需求，ODP提供了一套流量管理API，允许编码建立仲裁器，整形器等，以控制输出数据包处理实现所需要的QoS目标。 而且，这样做的优点是在许多平台上，流量管理功能都是硬件实现的，允许透明的卸载这个功能。

ODP 应用程序编码结构

头文件结构

应用程序编程时仅需要包含 ‘‘ include/odp_api.h‘‘ 文件，这个文件包含了 `platform/<implementation name>/include/odp/api` 文件， 以在该平台上提供API的完整定义。 定义API行为的doxygen文档都包含在公共APi文件中，接口实现在每个平台目录中。 如果 ‘‘ #define‘‘ 不直接对用户可见的话，通常可以适当的访问函数来替代。

Users include structure

```
./
├- include/
|   ├- odp/
|   |   └- api/
|   |       └- spec/
|   |           └- The Public API and the documentation.
```

```
|   |
|   |
|   ├- odp_api.h   This file should be the only file included by the
|   |               application.
```

Initialization

ODP应用程序需要优雅退出，所有程序必须在关闭ingress，情况所有队列等情况下才能执行终结函数退出程序。

Startup

ODP应用程序必须调用的第一个API是 `odp_init_global()` 。 这需要两个指针。 第一个是 `odp_init_t` ，包含平台独立且可移植的ODP初始化数据， 第二个是 `odp_platform_init_t` ，用于平台特定的数据。

调用 `odp_init_global()` 建立ODP API框架，这个操作需要在调用其他API之前执行。 每个应用程序只调用一次。 全局初始化完成之后，每个线程依次调用“odp_init_local()“ 。 这为该线程建立本地ODP线程上下文，这个操作必须在该线程调用其他ODP API前执行。 线程类型是 `ODP_THREAD_WORKER` 或 `ODP_THREAD_CONTROL` 。

Shutdown

关闭操作是初始化过程的反响逻辑，在调用 `odp_term_global()` 之前，每个线程调用 `odp_term_local` 来终止ODP。

应用程序初始化和终止时序

ODP应用程序遵循以下一般结构流程:

Fig. 1.7: ODP Application Structure Flow Diagram

公共约定

许多ODP API与参数和返回值类型共享常见的约定。本节重点介绍一些比较常见和常用的惯例。

句柄和特殊指示器

ODP资源通过具有抽象类型 `odp_resource_t` 的句柄表示。 所以池有类型为 `odp_pool_t` 的句柄表示，队列由类型 `odp_queue_t` 表示等等。 每个这样的类型都有一个不同类型的 `ODP_RESOURCE_INVALID` 用于指示一个句柄没有合法引用资源。 资源通常通过 `odp_resource_create()` API来创建，该API返回一个代表创建对象的类型为 `odp_resource_t` 的句柄。 如果资源耗尽，则无法创建，此时返回的句柄为 `ODP_RESOURCE_INVALID` 。 无效资源并不一定代表错误逻辑。 例如，ODP_EVENT_INVALID响应于从队列中获取事件的 `odp_queue_deq()` 调用，知识表示队列为空。

涉及范围

除非在API中特别注明，否则所有ODP资源都是ODP应用程序的全局资源，无论它是作为一个进程还是多个进程运行。 因此，ODP句柄在ODP应用程序中具有通用意义，但在应用程序范围之外没有任何意义。

资源和名称

许多ODP资源对象（如池和队列）在创建时支持与ODP对象关联的应用程序指定的字符串名称。 此名称有两个目的：文档和查找。 查找功能对于允许分为多个进程的ODP应用程序来获取公共资源的句柄特别有用。

应用程序可移植性注意事项

ODP旨在支持创建可轻松在多个平台上运行的可移植数据面应用程序，同时充分利用运行平台的硬件加速功能。 本节讨论应用程序开发人员在使用ODP时应考虑的问题。

首先，应该注意的是，可移植性不是绝对的，也不是一个单值属性。 虽然任何应用程序可以从一个平台移植到另一个平台，但真正的问题是：以什么样的代价？ 这个开销可以从两个维度上衡量：移植所需的工作量和移植导致的性能差异。 理想情况下，应用程序应该在平台之间以最小的工作量完成移植，且具有最小的性能影响。 虽然ODP旨在支持这一理想目标，但每个应用程序都必须评估其移植所要达到的目标以及如何使用ODP来实现这一目标。

可移植性和共存性

由于ODP提供的是编程框架而不是编程环境，因此它可以与其他框架提供的API一起工作，同时具有最小的干扰。 因此，当我们谈到ODP环境中的可移植性时，我们必须说明应用程序中使用ODP API的那些部分的可移植性。 如果应用程序使用非ODP API，则在评估整个应用程序的可移植性时必须考虑这些其他的API。 对于许多应用程序，将某些非可移植的代码隔离到应用程序的几个区域就足够了，结果是应用程序比没有使用ODP的应用程序更方便移植。 特别是在处理在生产环境中运行的现有应用程序时，可能会以增量的方式引入ODP，结果是随着时间的推移应用程序变得更加具有可移植性。

源代码 vs 二进制文件移植

ODP旨在支持源代码和二进制文件的可移植性。 源代码可移植是ODP API规范本身所固有的。 写入ODP API规范的任何应用程序将在任何符合ODP规范的ODP实现之间进行源代码的移植，最多只需要重新编译。 这是因为ODP API不会暴露实现细节或可能因平台而异的内部结构。

对于共享通用指令集架构（ISA）的平台，ODP还可以通过应用二进制接口（ABI）的规范提供二进制可移植性。 这在网络功能虚拟化（NFV）环境中特别有用。在NFV中，可以在一个平台上开发和编译数据平面应用程序以进行分发，然后通过NFV Orchestrator功能部署在许多不同的平台上。

ODP应用程序配置文件

为了帮助满足这些需求，ODP提供了两个不同的应用程序配置文件，旨在表征不同类型的数据平面应用程序的需求：嵌入式配置文件和云端配置文件。

嵌入式配置文件

当希望程序针对特定平台，希望在该平台上实现最佳性能，且源代码可移植性足够，那么就可以使用嵌入式配置文件。 如果这样的应用程序需要支持多个平台，那么它们只需要针对该平台的ODP实现进行重新编译。

嵌入式应用程序通常使用从git存储库下载的ODP副本，以便可以根据应用程序的精确需求进行配置。 要指定应用程序的平台，使用嵌入的配置文件：

```
./configure --enable-abi-compat=no ...
```

应该用作ODP配置选项的一部分。 这允许应用程序使用内联形式的ODP API来在此平台上提供最佳性能，并可能包括一些其他的优化。 这个结果是得到一个二进制文件，可以在给定的目标平台上实现最高性能，并可以通过重新编译移植到其他平台。

云端配置文件

相比直线，ODP云端配置文件旨在支持希望与平台无关的应用程序，并在共享此ABI的所有平台上进行二进制兼容。 Linux配置中包含的任何ODP实现将被配置为云配置文件，因此在针对分布式ODP副本（通过sudo apt-get install或等效命令安装的ODP）进行编译时，不需要额外的操作。

当使用从repo中下载的ODP副本时，配置时将使用云配置文件：

```
./configure --enable-abi-compat=yes ...
```

Note: 请注意，–enable-abi-compat = yes是默认值，因此不需要指定。除非为此选项指定了no，否则结果将是旨在在云配置文件中运行的应用程序。

ABI特性

ABI由几个约定组成，确保根据一个ODP实现编译的程序可以在另一个具有可能非常不同的ODP实现但不需要重新编译的平台上运行。 这些约定包括：

- 一组函数调用约定，定义函数如何调用其他函数，传递参数和接收返回的结果。这些通常由操作系统（如Linux）指定，并独立于ODP。
- 避免使用任何ODP API的内联扩展。这可以确保不同的ODP实现可以维护其不同的内部构件，而这些差异对于应用程序是可见的。
- 共享此ABI定义的所有ODP实现使用的ODP抽象数据类型的大小和一致性的协议。 这意味着，例如，odp_packet_t句柄的大小在ABI的所有成员中是相同的。 由于这些句柄是不透明的，所以ODP实现之间的结构不一样，因为应用程序从不引用这些可能不同的内部结构。

Note: 请注意，ABI定义存在于特定指令集架构（ISA）中，如x86-64或AArch64。 二进制文件不能直接在ISA之间进行连接，需要重新编译。

每个ODP实现将确定其支持的ABI定义（如果有的话）。 当在ABI compabitilty模式下编译ODP实现时，生成的二进制文件与共享此ABI的所有其他ODP实现自动二进制兼容。 例如，对于x86-64 ISA，odp-linux和odp-dpdk实现都是常见的ABI。

共享内存

分配共享内存

可以通过调用 odp_shm_reserve() API创建共享内存块。 该API调用需要传入共享内存块名称、块大小、对齐要求和可选标志参数。它返回一个 ‘‘ odp_shm_t‘‘ 结构体。 块大小和对齐以字节为单位。

Note: 提供的名称不一定是唯一的，即，在保留不同的块时，可以使用相同的名称。

创建一个共享内存块代码如下：

```
#define ALIGNMENT 128
#define BLKNAME "shared_items"

odp_shm_t shm;
uint32_t shm_flags = 0;

typedef struct {
...
} shared_data_t;

shm = odp_shm_reserve(BLKNAME, sizeof(shared_data_t), ALIGNMENT, shm_flags);
```

获取共享内存块地址

上述API调用返回的 odp_shm_t 句柄检索指定创建的共享内存块的地址（在调用者的ODP线程虚拟地址空间中）。

获取贡献内存块的代码如下：

```
shared_data_t *shared_data;
shared_data = odp_shm_addr(shm);
```

接口 odp_shm_addr() 返回的地址仅在ODP线程空间中有效。虽然 odp_shm_t 句柄可以在ODP线程之间共享，并且在任何线程中保持有效， 但 odp_shm_addr() 返回的地址可能根据线程空间而不同（对于同一个shm块），因此不应该在ODP线程之间共享。 举个例子，在两个ODP线程之间通过IPC传送shm句柄是正确的，并且让这些线程都执行各自的 odp_shm_addr() 来获取共享内存块的地址。 但是如果直接 odp_shm_addr() 返回的地址从一个ODP线程发送到另一个ODP线程则可能会失败（该地址在接收端的地址空间中可能没意义）。

即使调用 odp_shm_addr() 的线程与原来调用 odp_shm_reserve() 的线程不一致， odp_shm_addr() 返回的地址仍然保证根据在块创建时提供的对齐要求对齐。

所有共享内存块在任何ODP线程寻址空间中都是连续的，address~address+size，其中size是共享内存块的大小，这个空间是可读写的，映射了整个共享内存块。

内存行为

默认情况下，ODP线程被假定为缓存一致性系统：对共享内存块执行的任何更改都将保证最终对共享此内存块的其他ODP线程可见。 然而，ODP中并没有共享内存上任何操作相关联的隐式内存屏障，也就是说当ODP线程执行的改变对另一个ODP线程是不可见的， 故使用共享内存块的程序需要执行ODP提供的一些内存屏障来保证ODP线程之间共享数据的一致性。

如果ODP线程具有单独的虚拟空间（ODP线程被实现为进程），则给定的共享内存块映射到不同的ODP线程虚拟地址空间上各个ODP线程各不相同。 但是， ODP_SHM_SINGLE_VA 标志可以在 odp_shm_reserve() 调用时使用，以保证所有ODP线程的地址唯一性，无论其实现或创建的时间如何。

根据名称查找

上面讲过，共享内存块指针可以通过IPC机制在ODP线程之间传递，然后执行API来获取当前ODP线的地址。 获取已创建的共享内存块的指针的更简单的方法是通过接口 ‘‘ odp_shm_lookup()‘‘ 调用来实现。 但是，这需要ODP线程来提高共享内存块的名称，假如没有找到对应名称的内存块，则返回 ODP_SHM_INVALID 。 当使用相同名称保存多个共享内存块时，查找接口将返回任意这些块指针中的一个。

根据名称查找块指针和地址:

```
#define BLKNAME "shared_items"

odp_shm_t shm;
shared_data_t *shared_data;

shm = odp_shm_lookup(BLKNAME);
if (shm != ODP_SHM_INVALID) {
        shared_data = odp_shm_addr(shm);
        ...
}
```

释放共享内存块

使用 odp_shm_free() API调用执行释放共享内存块。 该接口需要共享内存块指针作为参数。 允许任何ODP线程在共享内存块上执行 odp_shm_free() ，即申请和释放共享内存块的线程可能不同。 共享内存块应该只被释放一次，一旦释放，共享内存块就不应该被任何ODP线程再次引用。

释放共享内存块:

```
if (odp_shm_free(shm) != 0) {
        ...//handle error
}
```

与外部程序共享内存

ODP提供了与ODP实例外部的实体共享内存的方法:

与外部非ODP线程共享内存块是通过在调用 odp_shm_reserve() 时设置 ‘‘ ODP_SHM_PROC‘‘ 标志来实现的。 这些非ODP线程如何检索共享内存块依赖于具体的实现和操作系统。

与外部ODP实例（运行于同一个操作系统）共享内存块是通过调用 odp_shm_reserve() 时设置 ‘‘ ODP_SHM_EXPORT ‘‘ 标志来实现的。 在ODP实例A中使用此标志创建的内存块可以通过在ODP实例B上使用 odp_shm_import() 接口映射到远程ODP实例B上（在相同操作系统中）。

ODP实例间共享内存: instance A

```
odp_shm_t shmA;
shmA = odp_shm_reserve("memoryA", size, 0, ODP_SHM_EXPORT);
```

ODP实例间共享内存: instance B

```
odp_shm_t shmB;
odp_instance_t odpA;

/* get ODP A instance handle by some OS method */
odpA = ...

/* get the shared memory exported by A:
shmB = odp_shm_import("memoryA", odpA, "memoryB", 0, 0);
```

Note: 每个ODP实例的范围限制shmA和shmB（您不能在其所属的ODP实例之外使用它们）。 另请注意，两个ODP实例必须在完成后调用odp_shm_free（）。

共享内存创建标志

`odp_shm_reserve()` API的最后一个参数是一组ORed标志。当前支持如下几种标志：

ODP_SHM_PROC

当给出此标志时，分配的共享内存将在ODP之外变得可见。 非ODP线程（例如通常的linux进程或linux线程）将能够使用本机（非ODP）OS调用（如shm_open（）和mmap（对于linux））来访问内存。 每个ODP实施应提供关于在该特定平台上如何完成此映射的描述。

ODP_SHM_EXPORT

当给出此标志时，分配的共享内存将在同一个操作系统上运行的其他ODP实例变得可见。 想要看到此导出内存的其他ODP实例应使用 `odp_shm_import()` ODP函数。

ODP_SHM_SW_ONLY

该标志指示ODP共享内存将仅由ODP应用软件使用：没有硬件（如DMA或其他加速器）访问内存。 这个内存不会涉及其他的ODP调用（ODP调用可能隐含地涉及到HW，这取决于ODP的实现），除了 `odp_shm_lookup()` 和 `odp_shm_free()`。 ODP实现可以使用该标志作为性能优化的提示，或者也可以忽略该标志。

ODP_SHM_SINGLE_VA

该标志用于保证共享内存被映射的地址的唯一性：没有该标志，给定的内存块可能会被不同的ODP线程映射到不同的虚拟地址（假设目标具有虚拟地址）。 这意味着 `odp_shm_addr()` 返回的值在不同的线程中是不同的。 设置此标志保证共享此内存块的所有ODP线程将在在所有ODP线程上调用 `odp_shm_addr()` 返回相同的值。 注意，ODP实现可能会限制可以分配这个标志的内存数目。

队列

队列是ODP提供的基本事件排序机制，所有的ODP程序都显式或隐式地使用队列。 队列是通过API `odp_queue_create()` 来创建的，该API返回一个类型为 `odp_queue_t` 的指针，该指针用于所有使

用该队列的API调用。 每个Queue具有两种ODP类型中的一个，POLL和SCHED，用于指示如何使用它们。 POLL队列由ODP应用程序直接管理，而SCHED队列使用ODP调度器提供自动可扩展调度和同步服务。

POLL queues 操作

```
odp_queue_t poll_q1 = odp_queue_create("poll queue 1", ODP_QUEUE_TYPE_POLL, NULL);
odp_queue_t poll_q2 = odp_queue_create("poll queue 2", ODP_QUEUE_TYPE_POLL, NULL);
...
odp_event_t ev = odp_queue_deq(poll_q1);
...do something
int rc = odp_queue_enq(poll_q2, ev);
```

关键区别是，在POLL队列中，事件出队操作是应用程序负责的，而SCHED队列中的事件出队则是ODP调度程序完成的。

SCHED queues 操作

```
odp_queue_param_t qp;
odp_queue_param_init(&qp);
odp_schedule_prio_t prio = ...;
odp_schedule_group_t sched_group = ...;
qp.sched.prio = prio;
qp.sched.sync = ODP_SCHED_SYNC_[NONE|ATOMIC|ORDERED];
qp.sched.group = sched_group;
qp.lock_count = n; /* Only relevant for ordered queues */
odp_queue_t sched_q1 = odp_queue_create("sched queue 1", ODP_QUEUE_TYPE_SCHED, &qp);

...thread init processing

while (1) {
        odp_event_t ev;
        odp_queue_t which_q;
        ev = odp_schedule(&which_q, <wait option>);
        ...process the event
}
```

当使用sched queue时，发送者选择一个目标队列，将事件发送到队列中。 发送者不知道哪个ODP线程（哪个core）或硬件加速器将处理这个事件，但是队列中的所有事件最终将被调度和处理。

可以看出，可以在SCHED队列创建时指定队列的其他属性，它们控制调度程序如何处理其中包含的事件，这些属性包括组，优先级和同步类。

组

调度器的工作是从最高优先级的SCHED queue中返回下一个事件给调用者。SCHED queue必须是调用者有资格接收的队列。 这个是由队列创建时设置的队列调度器组及调用者的调度器组掩码来指定的。 调度器组由 `odp_scheduler_group_t` 类型的指针表示，并由 `odp_scheduler_group_create()` 接口创建。 ODP预定义了多个调度器组，包括 `ODP_SCHED_GROUP_ALL` `ODP_SCHED_GROUP_WORKER` 及 `ODP_SCHED_GROUP_CONTROL` 。 应用程序可以自由创建其他调度器组，线程可以使用 `odp_scheduler_group_join()` 和 `odp_scheduler_group_leave()` 加入或离开调度器组。

优先级

数据结构 `odp_queue_param_t` 的prio字段指定队列调度的优先级，即如何选择符合条件的调度器组中的队列进行调度。 队列的默认调度优先级是 `NORMAL` 但可以根据需求设置成 `HIGHEST` 或 `LOWEST` 。

同步

除了在多核心环境中为ODP应用提供自动可扩展性的调度功能之外，调度程序的另一个主要功能是提供事件同步服务，大大简化并行处理环境中的应用程序编程。 队列的SYNC模式决定调度程序如何处理来自同一队列的多个事件的同步处理。 ODP支持三种类型的队列调度器同步区：并行，原子和有序。

并行队列

指定 `ODP_SCHED_SYNC_NONE` 模式的SCHED队列在处理事件的方式上是不受限制的。

Fig. 1.8: Parallel Queue Scheduling

在并行队列中保存的所有事件都有资格同时进行调度，并且它们之间的所有必需的同步都是应用程序负责的。 源自并行队列的事件也因此具有最高的吞吐率，但是它们也可能涉及大量的应用程序工作。 在上图中，四个线程正在调用 `odp_schedule()` 来获取要处理的事件。 调度程序已经将来自第一个队列的三个事件并行分配给三个线程。 第四个线程正在处理来自第三个队列的单个事件。 第二个队列可能是空的，优先级较低，或者不是在与调度程序服务的任何线程匹配的调度程序组中。

原子队列

原子队列简化事件同步，因为每一次只有一个线程可以处理给定原子队列中的事件。 由于锁定是由调度程序隐式完成的，因此原子队列调度的事件可以被锁定。 请注意，如果使用 `odp_schedule_multi()` ，调用者可能会从同一个原子队列接收一个或多个事件。 在这种情况下，这些多个事件都共享相同的原子调度上下文。

Fig. 1.9: Atomic Queue Scheduling

在这个例子中，无论在一个原子队列中可能有多少个事件，每次只有一个调用线程可以接收它的调度事件。 这里两个线程处理来自两个不同原子队列的事件。 请注意，不同的原子队列之间不存在来自同一原子队列的事件之间的同步。 与原子队列相关联的队列上下文将持续到下次调用调度程序或直到应用程序通过调用 `odp_schedule_release_atomic()` 显式释放它。 请注意，虽然原子队列简化了编程，但原子队列的串行性质可能会削弱扩展性。

有序队列

有序队列同时提供了并行队列的可扩展性和原子队列的易同步性。

Fig. 1.10: Ordered Queue Scheduling

当从有序队列调度事件时，调度程序从队列中并行分配多个事件到不同的线程， 并且调度程序还可以确保输出队列上的这些事件的相对顺序与其起始有序队列的序列相同。

与原子队列一样，与有序队列相关联的排序保证是指源自同一队列的事件，而不是源自不同队列的事件。 因此，在该图中，三个线程分别来自第一有序队列的处理事件5,3和4。 不管这些线程如何完成处理，这些事件将以它们的输出队列的原始相对顺序显示。

有序保证

无论事件是否被发送到不同的输出队列，相对顺序都将被保留。 例如，如果某些事件被发送到输出队列A，而其他事件被发送到输出队列B，则这些输出队列上的事件将仍然与它们的发起队列具有相同的相对顺序。 类似地，如果处理消耗事件，使得对于它们中的一些（例如，作为IP片段重新组合处理的一部分）不发出输出，其他事件仍然将相对于这些序列间隙被正确地排序。 最后，如果针对给定的顺序排入多个事件（例如，作为MTU注意事项的分组分割处理的一部分），则这些事件中的每一个将占据目标输出队列中的发起者的序列。 在这种情况下，这些事件的相对顺序将按照线程 `odp_queue_enq()` 调用它们的顺序。

与有序队列调度事件相关联的有序上下文将持续到下一个调度程序调用，或者直到调用 `odp_schedule_release_ordered()` 的显式释放。 该调用可以用作性能咨询，线程不再需要为当前上下文订购保证。 因此，当前调度程序上下文中的任何后续排队将被视为线程在并行队列上下文中运行。

顺序锁定

调度器处理有序队列的另一个强大功能是顺序锁。 与每个有序队列相关联的多个顺序锁是在队列创建时由lock_count参数指定的。 顺序锁提供了有效的方式来在顺序上下文中执行顺序处理。 例如，相关顺序5,6和7的假定事件由三个不同的线程并行执行。 顺序锁将使这些线程能够同步，以便它们可以在其起始队列顺序中执行一些关键部分。 每个有序队列支持的顺序锁的数量取决于具体实现（可通过 `odp_config_max_ordered_locks_per_queue()` API查询）。 如果实现支持多个有序锁，则这些锁可用于保护给定有序上下文中的不同有序临界区。

小结：有序队列

要了解这些注意事项如何组合在一起，请考虑以下代码： Processing with Ordered Queues

```
void worker_thread()
        odp_init_local();
        ...other initialization processing

        while (1) {
                ev = odp_schedule(&which_q, ODP_SCHED_WAIT);
                ...process events in parallel
                odp_schedule_order_lock(0);
                ...critical section processed in order
                odp_schedule_order_unlock(0);
                ...continue processing in parallel
                odp_queue_enq(dest_q, ev);
        }
}
```

这段代码表示了在有序队列上运行的典型工作线程的简化结构。 并行处理多个事件，并使用有序队列确保它们以与发起的顺序相同的顺序放置在dest_q上。 在并行处理的同时，使用有序锁可以使关键部分在整个并行流程中按顺序进行处理。 当一个线程到达 `odp_schedule_order_lock()` 调用时，它等待直到所有先前事件的锁定的锁定顺序已经解决，然后进入临界区。 `odp_schedule_order_unlock()` 调用释放关键部分，并允许下一个订单输入。

队列调度小结

有序和并行队列由于并行事件处理而提高了原子队列的吞吐量，但要求应用程序采取措施确保上下文数据同步（如果需要）。

报文处理

ODP应用程序旨在处理数据包。 为了帮助处理数据包，使得应用程序能够操作数据包数据及其元数据。 数据包通过各自实现的抽象数据类型 `odp_packet_t` 来引用。

当报文到达源 `odp_pktio_t` 并且被应用程序直接或间接（通过调度队列）接收时，报文对象被创建。 当他们通过相关联的的传输队列传输到 `odp_pktio_t` 时，他们可能被隐式释放，或者通过调用 `odp_packet_free()` 来释放。

有时，应用程序可能直接发起一个数据包，或者通过从现有数据包导出新的数据包，ODP提供了对应的API以实现这些处理。 应用程序创建的数据包可以通过回环口重新送回并进行分类，或者应用程序可以根据需要进行自己的解析。

与数据包相关联的各种属性（如解析结果）存储在元数据中，ODP提供了对应的API以允许应用程序操作并修改这些信息。

数据包结构和概念

报文是由符合诸如以太网架构格式的八位字节序组成，可以通过 ODP PKTIO抽象接口进行接收和发送。 数据包长度是指数据包的字节数。 ODP中数据包的数据是通过偏移来引用的，因为他们反映了数据包的逻辑内容和结构，而与特定的ODP实现和如何存储数据无关。

这些概念如下图所示：

Fig. 1.11: ODP Packet Structure

报文数据包括0个或多个报头，0个或多个字节的payload，再接0个或多个报尾。 这里显示的是允许应用程序检查和检索数据包的各个部分并操作其结构的各种API。

为了支持数据包操作，预定义了headroom和tailroom与数据包相关联。 可以通过操作这些区域来调整数据包。 典型的数据包处理包括通过数据包接收时调用 `odp_pull_head()` 从数据包中剥离报头，数据包发送时调用 `odp_push_head()` 插入新的报头。 注意，因为headroom和tailroom表示保留的区域，因此这些区域在通过相关的push操作成为报文的一部分之前，不能被ODP应用程序寻址或直接使用。 类似的，通过pull操作删除的字节就不能被访问了。

报文段和寻址

ODP平台使用一系列方法和技术来有效存储和处理数据包。 这些技术各个平台各不相同，因此为了保证可移植性，ODP提供一些数据包引用的约定。

ODP API通过抽象数据结构 `odp_packet_t` 来引用数据包对象。 描述数据包的系统元数据的各种位与数据包相关联。 通过参考元数据，ODP应用程序通过最小化检查报文数据的需求来加速报文处理。 这是因为，通过解析和分类功能来填充元数据，该功能与通过ODP调度程序呈现给应用程序之前发生的入口处理相耦合。

当ODP应用程序需要检查数据报内容时，它通过API调用来获取数据包的访问地址，该API调用可用于应用程序的一致性访问。 为了确保可移植性，ODP应用程序假定底层实现将数据包存储在预定于的实现及大小可管理的段中。 这表示应用程序可以通过正常的存储器访问来引用数据包的连续可寻址部分。 ODP提供API，允许应用程序按照需要以有效和便携的方式对数据包进行操作。 通过将这些与数据包提供的元数据结合，ODP应用程序可以完全独立于平台的方式运行，同时在支持ODP的各种平台上实现最佳性能。

报文段寻址及元数据的关系如下图所示：

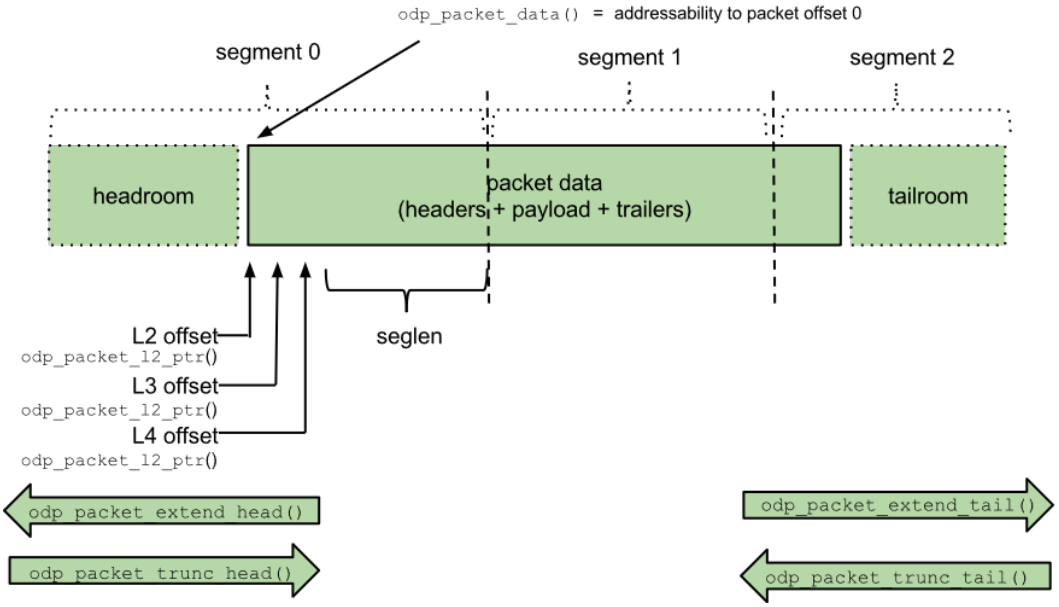

Fig. 1.12: ODP Packet Segmentation

报文元数据在解析阶段设置，标识报文中各种报头的起始偏移量。 报文本身被存储为由ODP实现管理的区域的一系列段。 段0是数据包的第一个段，并且通常是数据包的headroom和报头所在的位置。 根据报文长度的不同，附加段可以是报文的一部分，并且包含报文剩余的有效载荷和尾部。 应用程序不需要关注段，除非当前应用程序需要对报文进行寻址。 因此，例如，如果应用程序进行类似 `odp_packet_l4_ptr()` 的调用来获取数据包4层头的地址时，从该调用返回的长度就是从4层头部开始的可连续寻址的长度。 这是因为以下字节占用了不同的段，并且可能存储在不同地方。 为了获得对这些字节的访问，应用程序只需要寻址到该偏移量，并且能够寻址占用下一个段的数据包字节等。 请注意，任何数据包可寻址性调用的返回长度始终是剩余数据包长度或其包含段的大小中较小的。 因此，上图中的段2的映射将返回仅扩展到分组结尾的长度，因为剩余的字节是为数据包保留的尾部的一部分，并且应用程序不可用，直到可用通过适当的API调用。

不仅PUSH/PULL API允许应用程序对当前段结构中的数据包执行高效操作，ODP还提供了允许段添加/删除的API。 `odp_packet_extend_head()` 及 `odp_packet_trunc_head()` API允许在数据包开始处插入或删除段，而 `odp_packet_extend_tail()` 及 `odp_packet_trunc_tail()` 允许在数据包尾部添加删除段。 通过向数据包添加一个或多个段来扩展数据包，可以允许实现自定义长度的数据包。 截断数据包会删除一个或多个数据段，以缩小数据包的大小。

元数据处理

如上所述，作为报文接收阶段分类处理的一部分，报文元数据通常是由解析器设置。 需要注意的是，作为处理数据包的一部分，应用程序可能会更改此元数据，以反映报文内容和结构的变化。 虽然更改元数据可能会影响一些ODP API，更改元数据旨在记录应用程序对数据包的更改，但本身不会导致进行这些更改。 例如，如果应用程序通过使用 `odp_packet_l3_offset_set()` API来更改L3偏移量，那么后续对 `odp_packet_l3_ptr()` 的调用将返回一个从该更改的偏移开始的地址。 更改属性，如 `odp_packet_has_udp_set()` 这种操作本身不会将非UDP数据包变成有效的UDP数据包。 预估应用程序更改数据包时应该谨慎，以确保生成的元数据更改正确反映应用程序对数据包的修订。

报文操作

ODP报文操作API主要分成两类：不改变数据包段结构的以及可能会改变数据包段结构的。 上面已经举过这样的例子。 PUSH/PULL API允许对数据包headroom/tailroom进行处理，这不会导致数据包原有分段的改变。 而EXTEND/TRUNC API提供相同的功能，但是潜在的，可能会将添加段到数据包，或者从数据包删除段。

具有执行功能类似的两类API的原因是在大多数操作实现中，不改变报文段结构的操作将比那些改变段结构的操作更加高效。 为了解决这个问题，可能涉及数据包段更改的API总是将 `odp_packet_t` 作为返回值。 应用程序应该使用这个新返回的指针。

为了使以这种方式操作数据包的应用程序能够最有效地运行，这些API的返回值遵循标准约定。 通常，小于零的返回值表示错误，输入数据包将不会有变化。 返回值为零表示成功，但也表示任何缓存的数据包的可寻址性仍然有效。 大于零的返回值也表示成功，但潜在地改变了数据包可寻址性。 例如，如果应用程序先前通过 `odp_packet_l3_ptr()` API获得了数据包的第3层头的可寻址性，则返回值为0将意味着应用程序可能会继续使用该指针来访问L3头， 而返回值大于零意味着应用程序应该重新发出该调用以重新获得可寻址性，因为报文段可能已经改变，因此旧指针可能不再有效。

报文拷贝

数据包最简单的操作是制作数据包的全部或部分副本。 `odp_packet_copy()` 和 `odp_packet_copy_part()` API用于返回包含现有数据包的整体或选定部分的新数据包。 请注意，这些操作还指定新数据包申请的报文池。

报文数据拷贝及移动

ODP提供了几个API，使得数据包的部分可以复制到存储区域、另一个数据包或单个数据包中，如下所示：

Fig. 1.13: ODP Packet Data Copying and Moving Operations

当源或目的地址是ODP数据包时，这些API提供边界检查。 这意味着，数据必须在 `0..odp_packet_len()-1` 的偏移范围内。 对于涉及内存区域的操作，调用方负责确保 `odp_packet_copy_to/from_mem()` 引用的内存区域有效。

在单个数据包中处理数据时，提供了两个类似的API: `odp_packet_copy_data()` 和 `odp_packet_move_data()` 。 其中，移动操作更为通用，并且即使在源和目的数据区域重叠时也可以使用移动操作。 仅当调用者知道这两个区域不重叠时，才能使用复制操作，且此时复制更高效。 当处理重叠的存储区域时，`odp_packet_move_data()` 操作就好像是源区域首先被复制到不重叠的单独存储区域，然后再从该区域复制到目标区域。