CANN
6.3.RC2
软件安装指南 (开放态, Ascend 310P)

文档版本	01
发布日期	2023-08-03

华为技术有限公司

商标声明


和其他华为商标均为华为技术有限公司的商标。
本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为公司对本文档内容不做任何明示或暗示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

华为技术有限公司

地址： 深圳市龙岗区坂田华为总部办公楼 邮编：518129

网址： https://www.huawei.com

客户服务邮箱： support@huawei.com

客户服务电话： 4008302118

目 录

1 简介

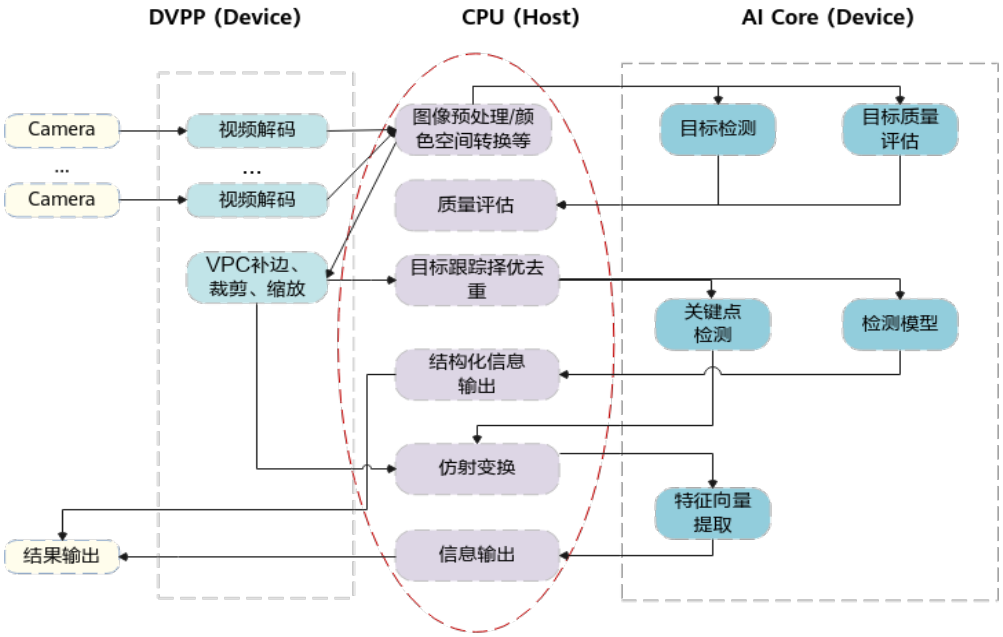
背景信息

Host和Device的概念说明如下：

- Host：是指与昇腾AI处理器所在硬件设备相连接的X86服务器、ARM服务器，利用昇腾AI处理器提供的NN（Neural-Network）计算能力完成业务。
- Device：是指安装了昇腾AI处理器的硬件设备，利用PCIe接口与服务器连接，为服务器提供NN计算能力。

标准形态，即Device为EP（Endpoint），通过PCIe（Peripheral Component Interconnect Express）配合Host进行工作，此时Device上的CPU资源仅能通过Host调用，相关推理应用程序运行在Host，Device只为服务器提供NN计算能力。例如推理应用的自定义预处理、后处理等操作在Host侧执行，DVPP图像预处理及模型推理在Device侧执行，典型业务处理流程如图1-1所示。

图 1-1 标准形态下业务处理流程样例



开放形态即客户可以利用Device侧Control CPU的通用算力，降低Host侧CPU的负载。即将上图中椭圆虚线框中的处理操作转移到Device侧的Control CPU，开放Device侧昇

腾AI处理器的编程能力，从而可降低对Host CPU的硬件要求，减少Host与Device之间的数据传输通信开销，提升整体处理性能。

须知

不支持开放形态与EP标准形态相互切换使用。

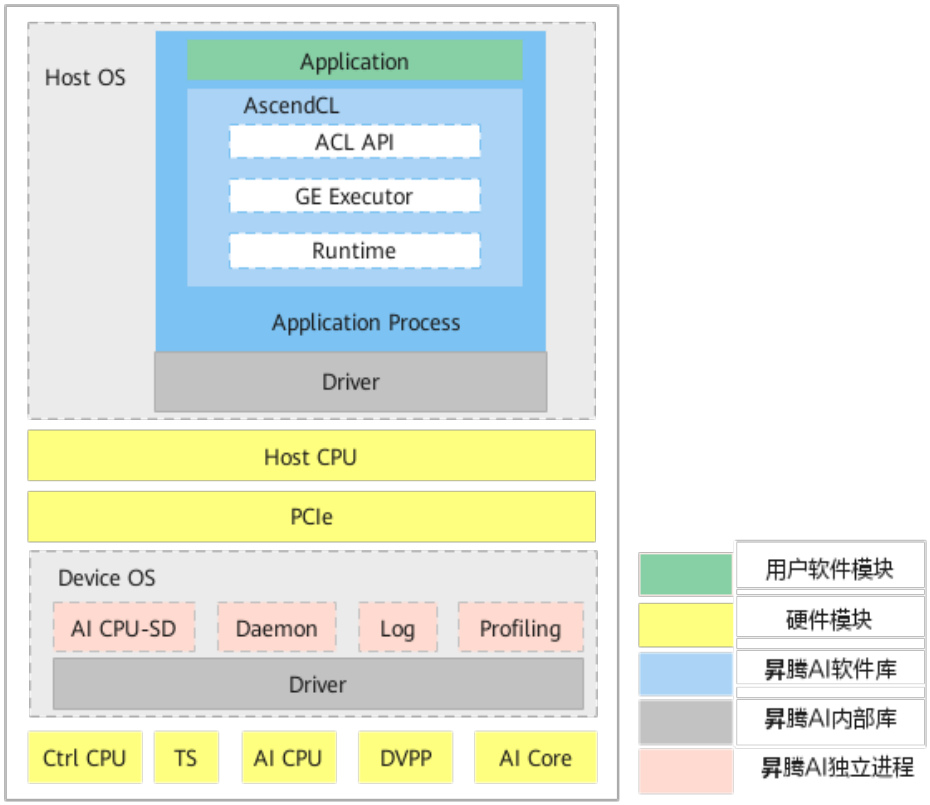
若进行了开放形态的改造，后续想使用EP标准形态，需要恢复Device侧的文件系统为标准态文件系统（例如：重装对应版本的Driver包，或者把历史备份的文件系统镜像包恢复，然后重启Host）。

标准形态与开放形态对比

- 标准形态

  标准形态的逻辑架构如图1-2所示。

  图 1-2 标准形态逻辑架构

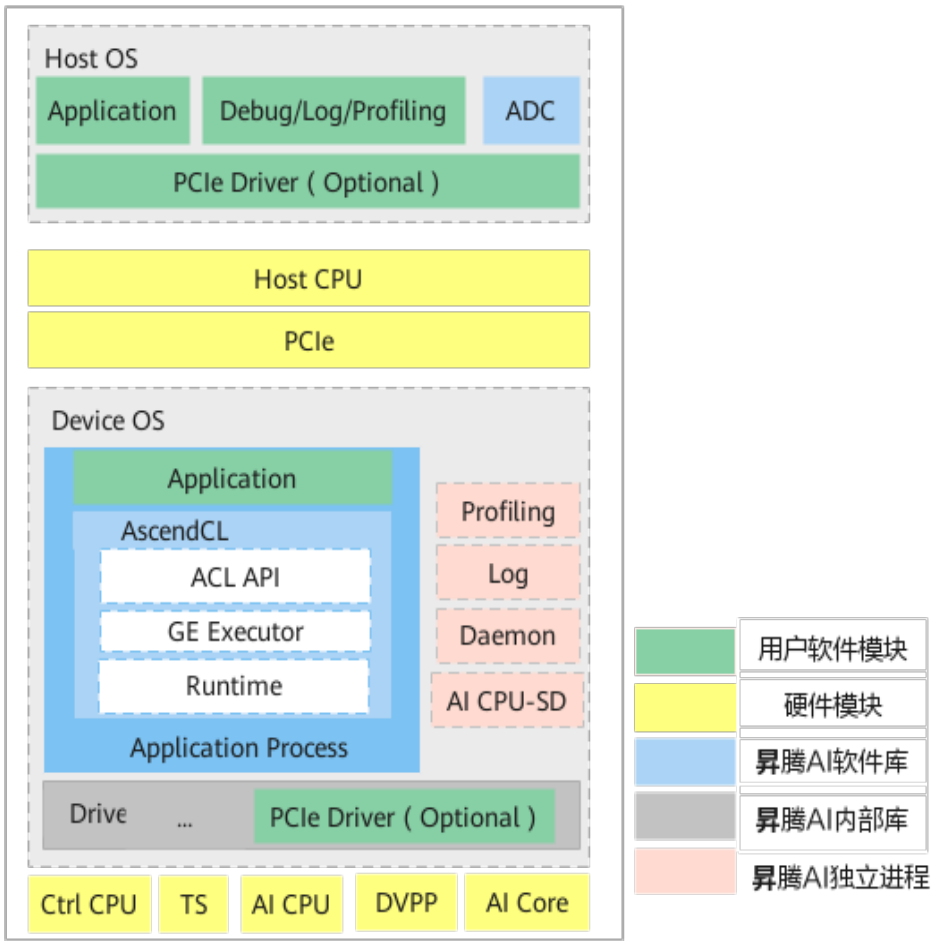
  

  标准形态下应用开发流程如下：

  a. 应用开发：Host调用AscendCL进行应用程序的开发，使用GCC进行Host侧应用程序的编译。
  b. 应用部署：在Host侧启动应用程序。
  c. 信息采集（可选）：在Host侧使用IDE启动Profiling任务，采集日志与Profiling信息。

- 开放形态

图 1-3 开放形态逻辑架构



开放形态下的应用开发流程如下：

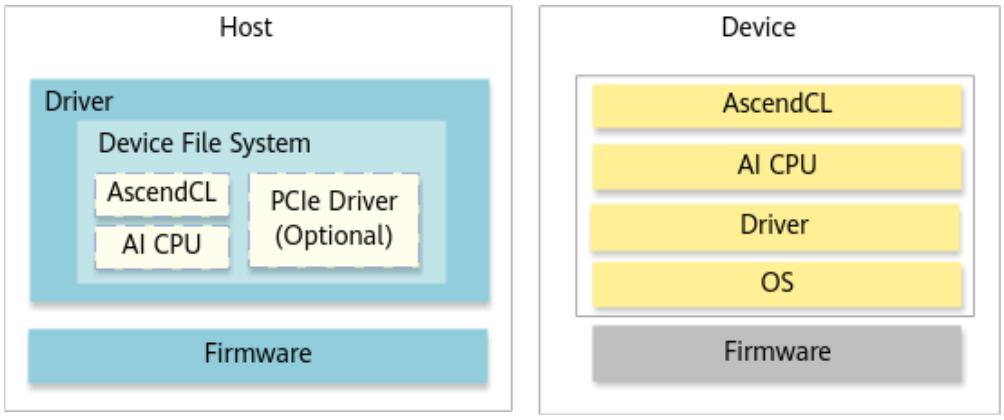
a. 定制Device侧文件系统镜像，主要包含将Runtime放置到文件系统，另外版本提供了驱动源码包，用户可自行定制驱动并进行驱动的源码编译。
b. 应用开发：对于Device侧应用程序，需要使用HCC（Huawei Compiler Collection：华为编译器）进行编译；对于Host侧应用程序，需要使用GCC进行编译。
c. 应用调试：将开发好的Device应用手工拷贝到Device进行调试。
d. 应用程序合入镜像：将调试好的Device应用合入镜像，重新打包文件系统。
e. 信息采集（可选）：采集日志与Profiling等相关信息。

2 安装方案

概述

开放场景下的运行环境为Device侧，Device侧的环境部署架构如图2-1所示。

图 2-1 环境部署架构



- Device的Firmware在设备启动时会从Host自动加载并进行更新。
- 用户只需要将AscendCL及AI CPU相关库文件部署到Host侧的Driver包的文件系统中，然后重新制作文件系统镜像。设备启动时，会自动加载Host侧的文件系统镜像到Device。
- 若用户需要自定义驱动，驱动定制后需要重新进行源码编译，将编译好的内核文件部署到文件系统中并重新制作文件系统镜像。设备启动时，会自动加载Host侧的文件系统镜像到Device。

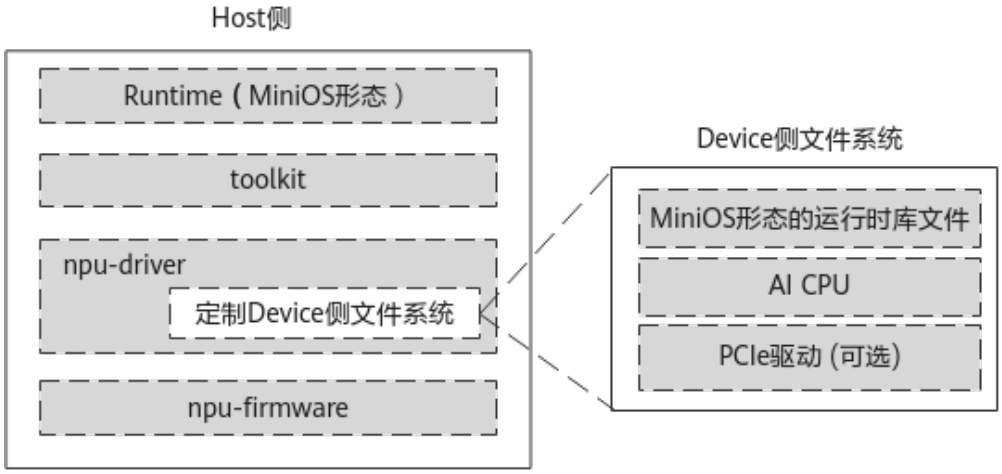
软件部署

开放形态软件部署主要分为Host侧环境部署及Device侧文件系统的定制。安装软件如图2-2所示，其中Device侧软件包说明如下。

- Host侧
  - npu-firmware：固件安装包。
  - npu-driver：驱动安装包。包含文件系统镜像包davinci_mini.cpio.gz。

  - toolkit：开发套件包。包含AI CPU算子包aicpu_kernels及HCC编译器，其中需要将AI CPU软件包中的相关库文件打入Device侧文件系统。
  - Runtime（MiniOS形态）：MiniOS形态的AI应用程序运行库。Host侧需安装开发态和运行态的MiniOS形态Runtime，其中运行态用于部署到Device侧的文件系统中。
- Device侧
  - MiniOS形态的运行时库文件：MiniOS形态的运行时组件包，包含在在Runtime软件包中。包括跟设备之间的交互、任务调度执行、图执行、数据传输、图像数据加速处理、Blas和短特征检索加速等相关接口和功能。
  - AI CPU：AI CPU相关库文件。包含了AI CPU算子的kernel实现及运行依赖的库文件。
  - PCIe驱动（可选）：用户不使用默认PCIe通信方式，需要自定义驱动源码时安装。

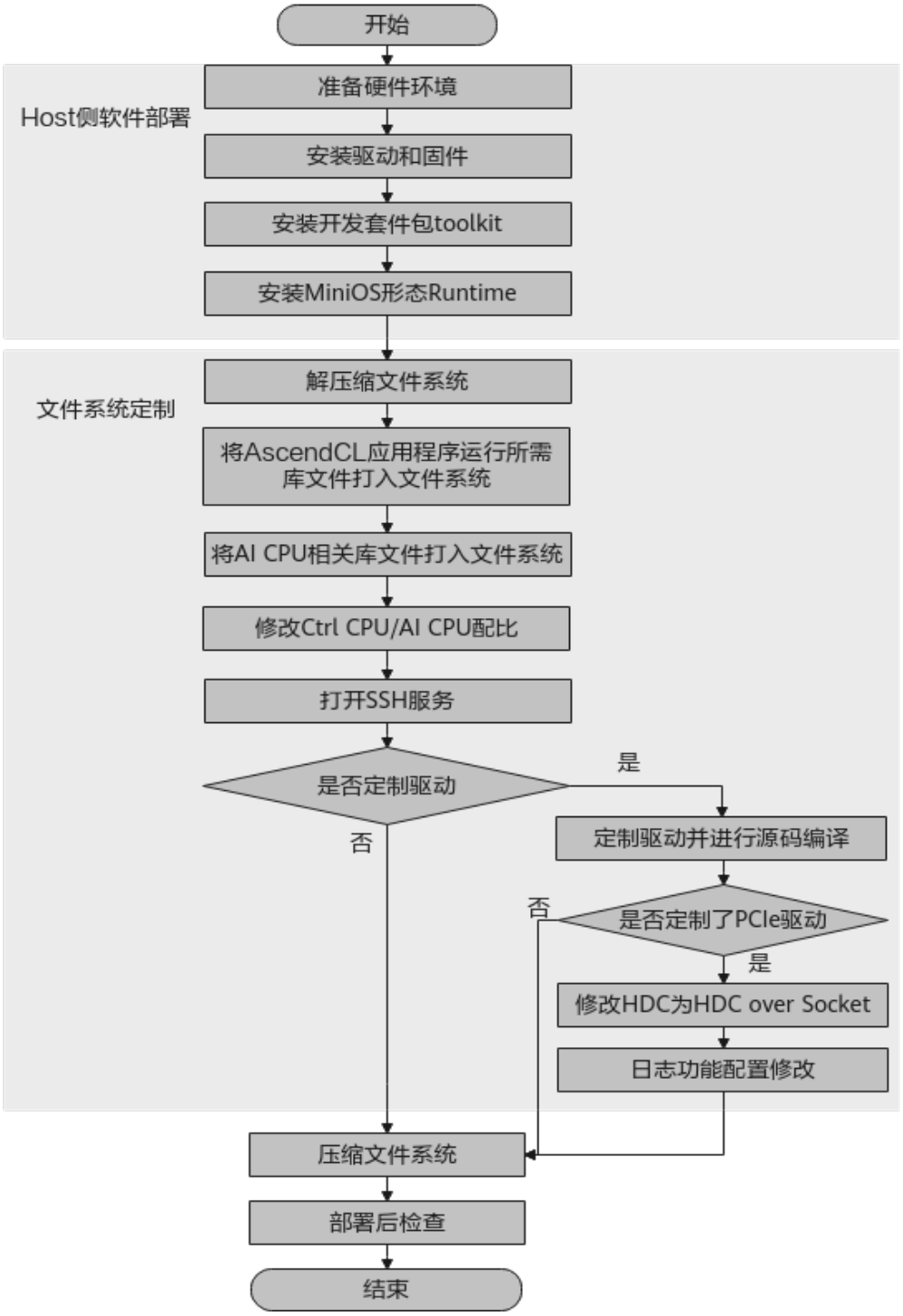
图 2-2 开放形态部署软件



3 部署流程

开放形态的部署流程如下图所示。

图 3-1 开放形态部署流程



步骤1 部署Host侧开发环境。

1. 准备硬件环境。
2. 安装驱动和固件。
3. 安装开发套件包toolkit。
4. 安装MiniOS形态Runtime。

步骤2 解压缩文件系统。

步骤3 文件系统修改。

1. 将MiniOS形态的AscendCL应用程序运行时所需文件打入文件系统，运行时所需文件在Runtime软件包中。
2. 将AI CPU相关库文件打入文件系统。
3. 修改AI CPU与Ctrl CPU的配比。
4. 修改系统启动脚本，打开SSH服务开关。
5. 是否需要定制驱动。
   - 若开发者无需定制驱动，至此文件系统修改完毕，后续可正常使用如下辅助功能：
     - 日志功能，请参见《**日志参考（开放态）**》。
     - 黑匣子功能，请参见《**黑匣子日志参考**》。
     - Profiling功能（昇腾AI任务性能分析），支持“通过调用acl.json文件方式采集Profiling数据”与“通过调用AscendCL API方式采集Profiling数据”两种方式，详细请参见《**CANN 开发工具指南**》（开放态）中的“性能分析工具”章节。

     下一步请执行**步骤4**进行文件系统的压缩即可。
   - 若开发者需要定制驱动。
     - 开发者可根据需要自行定制驱动源码包，定制后可参见**5.3.6 驱动源码编译**进行源码编译。

       若开发者定制了PCIe驱动，使用自己的PCIe驱动进行通信，源码编译完成后，还需要将HDC通信机制使用的通道由PCIe修改为Socket，将PCIe通道留给用户自己的PCIe驱动使用，详细修改方法请参见**5.3.7 （PCIe定制场景）修改HDC通信机制**。

       PCIe驱动定制后，原有Host与Device间通信机制发生变化，辅助功能的使用如下：
       - 日志功能无法通过日志服务slogd和驱动的hdc接口将日志信息转储到Host侧，所以此场景下日志需要进行打屏输出，用户可以对打屏日志做进一步处理，需要参见**5.3.8 （PCIe定制场景）日志功能配置**在制作文件系统时禁止slogd的启动。
       - 黑匣子功能与Profiling功能，与不定制PCIe驱动场景的使用方法相同，详细使用方法请参见《**黑匣子日志参考**》与《**CANN 开发工具指南**》（开放态）中的“性能分析工具”章节。

步骤4 压缩文件系统，并替换Device安装路径下的文件系统。

步骤5 进行Device侧环境检查。

----结束

4 Host 侧软件安装

软件包获取

进行环境搭建前，需要获取以下软件包。

组件	包名	简介	获取链接
CtrlCPU开放SDK	Ascend-cann-device-sdk_*<version>*_linux-*<arch>*.zip	包含Runtime（MiniOS形态）和PCIe驱动源码包。 • Runtime（MiniOS形态）：MiniOS形态的运行时组件包。包括跟设备之间的交互、任务调度执行、图执行、数据传输、图像数据加速处理、Blas和短特征检索加速等相关接口和功能。 • 驱动源码包：包含了PCIe源码，kernel镜像，以及文件系统签名及打包相关工具。	**获取链接**

获取CtrlCPU开放SDK包后，**需要将其解压，**解压后得到包如**表4-1**所示。

表 4-1 CtrlCPU 开放 SDK 包组件

组件	包名
Runtime（MiniOS形态）	**CANN-runtime-***<version>***-minios.aarch64.run**
驱动源码包	***<soc version>*-driver-***<version>***-minios.aarch64-src.tar.gz**

其中软件包中的{version}表示软件包版本号，<arch>表示操作系统架构，<soc version>昇腾AI处理器的版本。

安装步骤

步骤1 Host侧开发环境软件安装。

- 通过工具进行安装。请参见《**ascend-deployer用户指南**》或《**SmartKit Computing 用户指南**》安装固件包npu-firmware、驱动包npu-driver和开发套件包toolkit。
- 通过命令行方式安装。请参见《**CANN 6.3.RC2 软件安装指南**》安装驱动固件以及开发套件包。

步骤2 请将**软件包获取**的软件包以root用户或者Host侧规划的普通安装用户上传到Host侧，例如以root用户上传到Host侧的/usr/local/software/目录中。其中软件包中的***<version>***表示具体版本号。

步骤3 进入Host侧的/usr/local/software/目录中，在此目录中执行如下命令为run包增加可执行权限。

chmod +x CANN-runtime-*<version>*-minios.aarch64.run

步骤4 校验安装包。

./CANN-runtime-*<version>*-minios.aarch64.run --check

步骤5 安装Runtime软件包。

1. 安装开发环境所需的Runtime包。

   ./CANN-runtime-*<version>*-minios.aarch64.run --full --install-path=/usr/local/AscendMiniOs --pre-check

   /usr/local/AscendMiniOs是示例安装路径，用户可指定安装路径。

   说明

   如果用户同时安装Runtime开放态和标准态软件包，未防止覆盖安装，建议使用**--install-path=<path>**参数指定安装路径，例如--install-path=$HOME/AscendMiniOs。

2. 安装运行环境所需的Runtime包。

   ./CANN-runtime-*<version>*-minios.aarch64.run --run --install-path=/usr/local/AscendMiniOSRun --pre-check

   --run命令安装时必须指定目录，安装后的文件存储目录示例为“/usr/local/AscendMiniOSRun/”。

----结束

5 定制文件系统

5.1 概述

简介

此章节的目的是重新制作Device侧的文件系统，主要内容包含将运行时依赖文件打入文件系统；若客户需要自定义驱动，还需要重新进行源码编译；若定制了PCIe驱动，还需要对HDC通信方式进行更改，对日志功能的配置进行修改。

声明

以下路径是各软件以root用户安装的默认路径，本文以此路径为例来说明文件系统定制的操作步骤，请**务必**根据实际情况获取这些软件的实际安装路径，以便后续操作时使用：

- “/usr/local/Ascend/driver”为Driver组件的默认安装路径。
- ${INSTALL_DIR}：为CANN软件安装目录。例如若使用root用户安装的Ascend-cann-toolkit软件包，则安装后文件存储路径为：/usr/local/Ascend/ascend-toolkit/latest；若使用普通用户安装的Ascend-cann-toolkit软件包，则安装后文件存储路径为：$HOME/Ascend/ascend-toolkit/latest。

5.2 解压缩文件系统

去掉文件系统的签名头，并解压缩文件系统。

步骤1 执行以下命令切换到root用户，后续操作都需要以root用户执行。

su - root

步骤2 创建工作目录用于进行文件系统的修改，例如：/usr/local/filesys_modify。

mkdir /usr/local/filesys_modify

说明

下文文件系统制作目录都以/usr/local/filesys_modify为例。

步骤3 从/usr/local/Ascend/driver/device目录下获取Device侧的文件系统包Ascend310P.cpio.gz。

将/usr/local/Ascend/driver/device/Ascend310P.cpio.gz拷贝到用户修改文件系统工作目录下，例如/usr/local/filesys_modify。

cp /usr/local/Ascend/driver/device/Ascend310P.cpio.gz /usr/local/filesys_modify/

步骤4 进入/usr/local/filesys_modify目录，执行如下命令去掉文件系统的签名头。

dd if=Ascend310P.cpio.gz of=raw-Ascend310P.cpio.gz skip=33 bs=256

命令执行完成后，生成去头文件raw-Ascend310P.cpio.gz。

步骤5 执行如下命令解压文件系统。

gunzip raw-Ascend310P.cpio.gz

解压缩完成后在当前目录下生成文件系统文件raw-Ascend310P.cpio。

步骤6 在当前目录下创建tempdir文件夹，并进入tempdir目录。

mkdir tempdir

cd tempdir

步骤7 执行如下命令在tempdir目录下解压raw-Ascend310P.cpio文件系统，完成分包。

cpio -idmv < ../raw-Ascend310P.cpio

命令执行完成后，会在tempdir目录下生成Device侧的解压后的文件系统。

----结束

5.3 修改文件系统

5.3.1 使用前须知

- 修改文件系统前，请参见**3 部署流程**了解文件系统修改流程。
- **注意：**文件系统最大支持2G，修改文件系统时，压缩前文件系统大小不要超过此最大限制，否则会造成Device启动失败。

5.3.2 部署应用运行依赖文件

简介

开放形态下AscendCL应用程序运行在Device侧，所以需要将运行时依赖库文件以及配置文件打入文件系统，Host重启后，会将文件系统镜像自动加载到Device侧，为Device侧应用程序运行提供必要的依赖库。

操作步骤

步骤1 执行以下命令切换到root用户，后续操作都需要以root用户执行。

su - root

步骤2 在解压后文件系统目录下的home/HwHiAiUser/目录中创建相关目录，用于存放应用程序运行时依赖库文件以及配置文件。

mkdir *-p /usr/local/filesys_modify/*tempdir/home/HwHiAiUser/Ascend/lib64

mkdir -p *//usr/local/filesys_modify/*tempdir/home/HwHiAiUser/Ascend/data/platform_config

步骤3 将安装后的minios形态的运行时依赖库文件拷贝到**步骤2**创建的“/home/HwHiAiUser/Ascend”目录中。

cp -r /usr/local/AscendMiniOSRun/aarch64-linux/lib64/* */usr/local/filesys_modify/*tempdir/home/HwHiAiUser/Ascend/lib64/

步骤4 将Host侧相关配置文件拷贝到**步骤2**创建的“/home/HwHiAiUser/Ascend/data/platform_config”目录中。

cp /usr/local/Ascend/ascend-toolkit/latest/*x86_64-linux*/data/platform_config/Ascend310P*.ini */usr/local/filesys_modify/*tempdir/home/HwHiAiUser/Ascend/data/platform_config

若Host侧操作系统形态为aarch64，请将上述路径中的“x86_64-linux”修改为“aarch64-linux”。

步骤5 将文件系统中的/home/HwHiAiUser/Ascend文件夹的属组修改为HwHiAiUser用户组。

因为Device侧应用运行用户为HwHiAiUser用户，所以需要为相关库文件添加HwHiAiUser用户属组。

cd */usr/local/filesys_modify/*tempdir/home/HwHiAiUser/

chown -R 1000:1000 Ascend

说明

Device侧HwHiAiUser用户的用户ID和属组ID为1000。

步骤6 配置AscendCL应用运行所需环境变量。

修改文件系统的/home/HwHiAiUser/.bashrc文件，添加LD_LIBRARY_PATH的环境变量。

执行如下命令打开/home/HwHiAiUser/.bashrc文件：

vi */usr/local/filesys_modify/*tempdir/home/HwHiAiUser/.bashrc

在文件最后添加如下环境变量：

export LD_LIBRARY_PATH=/home/HwHiAiUser/Ascend/lib64

并执行**:wq**保存退出。

步骤7 （可选）若开发者需要将自己开发的应用程序部署到Device侧运行，也可以将应用程序可执行文件放到文件系统任一目录中。

应用程序请赋予HwHiAiUser用户的属组，执行赋权操作时，请使用HwHiAiUser用户的用户ID与属组ID，ID都为1000。

若需要实现不同的Device加载不同的业务包，请参见**7 Device侧业务包加载**进行开发。

说明

- 建议先参考《**应用软件开发指南（C&C++，开放态）**》中的“AscendCL样例使用指导”将使用HCC编译器编译好的应用程序拷贝到Device侧进行调试后，再将应用程序打入文件系统。
- 压缩前文件系统大小请不要超过2G，否则会造成Device启动失败。

----结束

5.3.3 部署 AI CPU 相关库文件

简介

开放形态下需要AI CPU相关库文件打入文件系统，Host重启后，会将文件系统镜像自动加载到Device侧，为Device侧AI CPU算子执行提供必要的依赖库。

操作步骤

步骤1 执行以下命令切换到root用户，后续操作都需要以root用户执行。

su - root

步骤2 解压缩“*{soc version}*-aicpu_syskernels.tar.gz”软件包。

1. 进入CANN软件安装后文件存储路径的“opp/*{soc version}*/aicpu”目录（例如普通用户默认安装路径为：$HOME/Ascend/ascend-toolkit/latest/opp/*{soc version}*/aicpu，root用户默认安装路径为：/usr/local/Ascend/ascend-toolkit/latest/opp/*{soc version}*/aicpu），执行如下命令去掉“*{soc version}*-aicpu_syskernels.tar.gz”的签名头。

   dd if=Ascend310P-aicpu_syskernels.tar.gz of=raw-Ascend310P-aicpu_syskernels.tar.gz skip=33 bs=256

   命令执行完成后，生成去头文件“raw-Ascend310P-aicpu_syskernels.tar.gz”。
2. 执行如下命令解压“raw-Ascend310P-aicpu_syskernels.tar.gz”。

   tar -zxvf raw-Ascend310P-aicpu_syskernels.tar.gz

   解压缩完成后在当前目录下生成文件夹“aicpu_kernels_device”。

步骤3 在当前目录下执行如下命令，替换解压后文件系统中“usr/lib64/aicpu_kernels”文件夹下库文件。

cp -rf aicpu_kernels_device/*.so */usr/local/filesys_modify/*tempdir/usr/lib64/aicpu_kernels/

步骤4 修改文件系统中“**usr/lib64/aicpu_kernels**”文件夹下文件的权限和属组。

1. 修改aicpu_kernels目录下的文件权限。

   cd */usr/local/filesys_modify/*tempdir/usr/lib64/aicpu_kernels

   chmod 440 *
2. 修改用户属组。

Device侧aicpu_kernels目录下所有文件的用户属组是HwHiAiUser，使用root用户执行**步骤3**拷贝命令时会改变此目录下部分文件属组，所以需执行如下命令更改aicpu_kernels下所有文件属组。

cd */usr/local/filesys_modify/*tempdir/

chown -R 1000:1000 usr/lib64/aicpu_kernels

说明

Device侧HwHiAiUser用户的用户ID和属组ID为1000。

----结束

5.3.4 修改 Ctrl CPU 和 AI CPU 配比

ARM CPU可划分为Ctrl CPU和AI CPU两部分，其中AI CPU负责执行不适合跑在AI Core中的算子（承担非矩阵类复杂计算），Ctrl CPU负责控制芯片的整体运行，开放形态下，可用于执行推理应用的自定义预处理、后处理等操作，用户可自行根据业务复杂度修改Ctrl CPU和AI CPU的配比。

昇腾310P AI处理器提供的Ctrl CPU和AI CPU的默认配比为1:7，此默认配比不适合开放形态下应用程序运行在Device上的场景。所以开放形态下，用户可通过修改驱动源码包{soc version}-driver-{software version}-minios.aarch64-src.tar.gz中的配置文件，修改Ctrl CPU和AI CPU的默认配比，详细的操作步骤如下：

步骤1 执行以下命令切换到root用户，后续操作都需要以root用户执行。

su - root

步骤2 解压**软件包获取**中的驱动源码包{soc version}-driver-{software version}-minios.aarch64-src.tar.gz。

例如在/usr/local/software/目录下执行如下解压命令：

tar -zxvf {soc version}-driver-{software version}-minios.aarch64-src.tar.gz

命令执行完成后会在/usr/local/software/目录下生成解压后的driver文件夹。

步骤3 在“driver/source/config/user_config/miniv2/config.h”配置文件中修改AI CPU和Ctrl CPU配比。

vi driver/source/config/user_config/miniv2/config.h

查找关键信息“**CPU_NUM_CONFIG_NAME**”，如下图所示：

```
static const struct user_config_item user_cfg_version_1[UC_ITEM_MAX_NUM] = {
#ifdef CFG_SOC_PLATFORM_MDC_V51
    {CPU_NUM_CONFIG_NAME, -1, UC_CPU_CFG_OFFSET, UC_CPU_CFG_SIZE, UC_AUTHORITY_USER_WR, "0x000000", NULL},
    {"ddr_ecc_enable", -1, UC_DDR_ECC_CFG_OFFSET, UC_DDR_ECC_SIZE, UC_AUTHORITY_USER_WR, "0x00",
        ddr_ecc_enable_check_para},
#else
    {CPU_NUM_CONFIG_NAME, -1, UC_CPU_CFG_OFFSET, UC_CPU_CFG_SIZE, UC_AUTHORITY_USER_WR, "0x010007", NULL},
```

红框中的第一个字节“01”代表Ctrl CPU的数量；第二个字节“00”代表Data CPU的数量，推理场景下保持默认值0，开发者无需关注；第三个字节“07”代表AI CPU的数量。

用户可通过修改以上三个字节，进行Ctrl CPU、Data CPU、AI CPU的数量配置，修改配置时请遵循如下约束：

- Ctrl CPU的数量 + Data CPU的数量 + AI CPU的数量 = 8
- Ctrl CPU的数量 >=1

- AI CPU的数量 >=1

说明

例如，开放形态下，Data CPU配置为0，Ctrl CPU与AI CPU的数量组合可以为4+4、5+3、6+2、7+1。

修改完成后，输入**:wq**保存退出。

步骤4 进行驱动源码编译，并替换文件系统中的内核镜像文件。

因为修改Ctrl CPU与AI CPU配比的操作对驱动源码包进行了修改，所以需要重新进行驱动源码包的编译，编译操作请参见**5.3.6 驱动源码编译**。

须知

- 用户也可以通过DCMI接口修改Ctrl CPU与AI CPU的配比。请注意，一旦通过DCMI配置Ctrl CPU与AI CPU配比后，以上默认配置就会失效，因为DCMI配置优先级高于config.h中默认配置，若想仍然使用默认配置，则需要清除DCMI配置。
- 通过DCMI修改Ctrl CPU与AI CPU配置可用使用DCMI的**dcmi_set_device_user_config**接口，清除DCMI配置可使用**dcmi_clear_device_user_config**接口，接口的详细使用方法可请参见《**DCMI API参考**》。
- 使用DCMI接口修改配置后，需要重启Host才会生效。

----结束

5.3.5 打开 SSH 服务

简介

Device的SSH服务默认处于关闭状态，开放态场景下，若需要开启Device侧的SSH服务，可参考本章节进行操作。

须知

- 若您制作文件系统时，未参考本章节修改系统启动脚本打开SSH服务，后续若需要打开，系统启动后，您可以参见**配置接口打开SSH服务**进行操作。
- Device的SSH服务处于关闭状态可提升系统安全性，如果需要打开SSH，请确保SSH登录密码的复杂度和机密性。如果SSH登录密码出现泄露，攻击者可侵入Device对系统进行篡改，并可能导致系统内部敏感信息泄漏、系统无法得到预期的运行结果或系统无法正常运行等安全事件。

操作步骤

您可以通过修改Device系统启动脚本“device_sys_init.sh”，实现SSH服务的开启，详细操作步骤如下：

步骤1 执行以下命令切换到root用户，后续操作都需要以root用户执行。

su - root

步骤2 切换到解压后的Device侧文件系统所在目录。

cd *[/usr/local/filesys_modify/tempdir](#)*

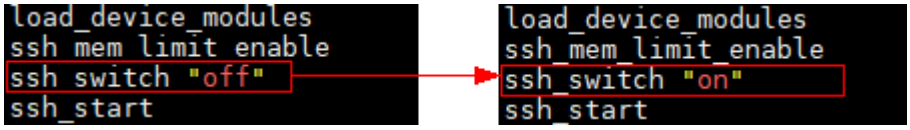
步骤3 执行如下命令为“device_sys_init.sh”脚本添加写权限。

cd var

chmod +w device_sys_init.sh

步骤4 在“device_sys_init.sh”文件中查找关键字**ssh_switch "off"**，并将“ssh_switch”后的“off”修改为“on”。

如下图所示：



SSH服务开启后，系统默认的SSH服务占用的内存大小限制为50MB，若用户想取消此限制，可参考如下操作：

注释掉“device_sys_init.sh”脚本中的“ssh_mem_limit_enable”，如下所示：

```
load_device_modules
#ssh_mem_limit_enable
ssh_switch "on"
ssh_start
```

须知

若用户在修改文件系统时未取消SSH服务占用内存大小的限制，后续Device启动后，可参见步骤5通过修改cgroup文件的方式取消SSH服务的内存大小限制。

步骤5 保存对“device_sys_init.sh”脚本的修改。

步骤6 执行如下命令取消“device_sys_init.sh”脚本的写权限。

chmod -w device_sys_init.sh

----结束

5.3.6 驱动源码编译

简介

发布包中提供了驱动源码包*{soc version}*-driver-*{software version}*-minios.aarch64-src.tar.gz，用户可自行根据需要进行驱动的定制或配置文件的修改。若用户对驱动源码包中的任何文件进行了定制修改，定制完成后，都需要参考本章节进行源码编译；若未对驱动源码包中的文件进行任何修改，则此章节可跳过。

源码编译依赖于HCC编译器，HCC编译器存在于toolkit安装包的toolchain目录下。

操作步骤

步骤1 执行以下命令切换到root用户，后续操作都需要以root用户执行。

su - root

步骤2 解压**软件包获取**中的驱动源码包*{soc version}*-driver-*{software version}*-minios.aarch64-src.tar.gz。

例如在/usr/local/software/目录下执行如下解压命令：

tar -zxvf *{soc version}*-driver-*{software version}*-minios.aarch64-src.tar.gz

若已执行过此操作，此步骤可跳过。

步骤3 进入驱动源码包解压后的driver目录，例如*/usr/local/software/driver*。

步骤4 关闭内核模块（KO）签名功能。

Device操作系统内核模块默认开启了签名功能，若用户自定义驱动源码，需要关闭内核模块的签名功能，操作步骤如下：

1. 进入*/usr/local/software/*driver/kernel/linux-5.10目录。

   cd /usr/local/software/driver/source/kernel/linux-5.10

2. 执行如下命令生成.config文件。

   make ARCH=arm64 CROSS_COMPILE=${INSTALL_DIR}/toolkit/toolchain/hcc/bin/aarch64-target-linux-gnu- eulerosv2r9_defconfig

   - ${INSTALL_DIR}请替换为CANN软件安装后文件存储路径，例如“$HOME/Ascend/ascend-toolkit/latest”。
   - 此命令依赖bison与flex，可使用如下命令进行安装：

     yum install bison flex

     或者

     apt-get install bison flex

     说明：请根据实际操作系统版本选择依赖的安装命令，如果执行命令时提示缺少其他依赖，请根据提示自行安装。

3. 修改内核配置选项。

   a. 执行如下命令，弹出内核配置界面。

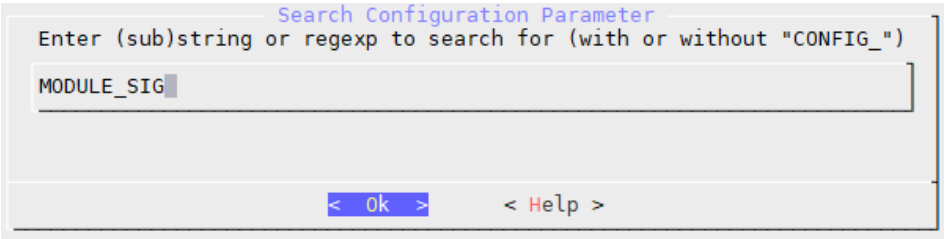
      make ARCH=arm64 CROSS_COMPILE=${INSTALL_DIR}/toolkit/toolchain/hcc/bin/aarch64-target-linux-gnu- menuconfig

      - ${INSTALL_DIR}请替换为CANN软件安装后文件存储路径。
      - 此命令依赖ncurses-devel，可使用如下命令进行安装：**yum install -y ncurses-devel** 或者 **apt-get install libncurses5-dev**

        说明：请根据实际操作系统版本选择依赖的安装命令，如果执行命令时提示缺少其他依赖，请根据提示自行安装。

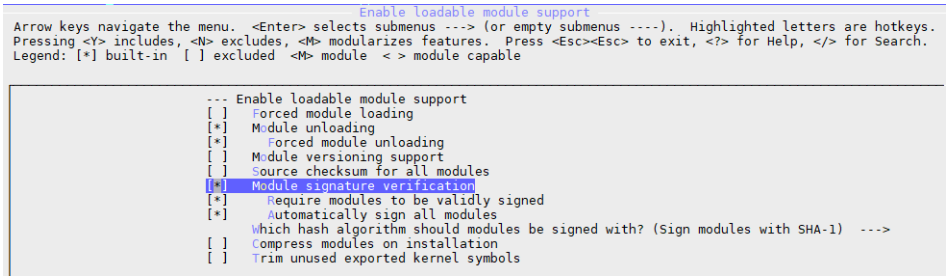
   b. 在配置界面中输入“/”，进入搜索界面，搜索“MODULE_SIG”。

      如下图所示：

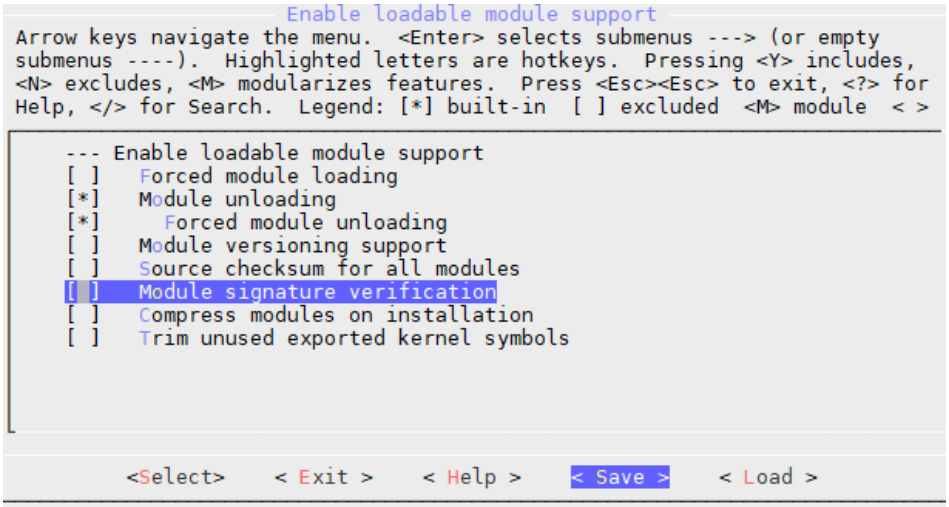


c. 选择“Ok”，进入搜索结果界面，在搜索结果界面中按数字键“1”跳转到CONFIG选项。

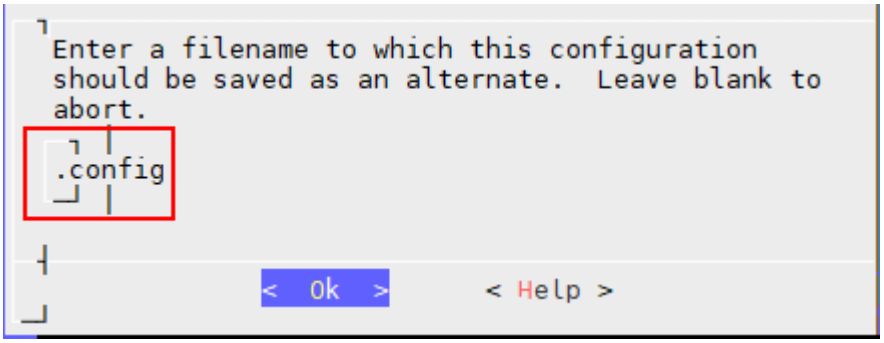
如下图所示：



按空格键关闭内核模块校验选项，[ ]里面存在“*”表示打开，[ ]空白表示关闭，关闭后示例如下所示：



d. 选择“Save”保存，进入配置文件名称界面，如下所示。



请注意，需要保持默认文件名“.config”，然后选择“Ok”。

e. 连续选择“Exit”，退出内核配置界面。

4. 替换“.config”文件。

   cp .config arch/arm64/configs/eulerosv2r9_defconfig

5. 执行如下命令重新编译生成内核镜像。

   make ARCH=arm64 CROSS_COMPILE=${INSTALL_DIR}/toolkit/toolchain/hcc/bin/aarch64-target-linux-gnu- Image

   ${INSTALL_DIR}请替换为CANN软件安装后文件存储路径。

   在弹出的回显信息中默认输入“y”,命令执行完成后，会在“kernel/linux-5.10/arch/arm64/boot”目录下生成内核镜像文件“Image”。

   说明

   提升编译内核镜像效率的方法如下：

   内核镜像编译支持-j[N]参数指定编译CPU核数，N表示使用N个CPU并发编译，不指定则使用1个cpu编译内核。N参数应根据实际编译环境CPU个数配置（可执行“cat /proc/cpuinfo | grep processor | wc -l”命令查看），比如编译环境有32个CPU，则可以使用20个CPU编译，命令示例如下所示：

   make ARCH=arm64 CROSS_COMPILE=${INSTALL_DIR}/toolkit/toolchain/hcc/bin/aarch64-target-linux-gnu- Image -j20

6. 为重新生成的内核镜像增加签名头。

   进入内核镜像文件所在目录。

   cd arch/arm64/boot/

   执行如下两条命令为内核镜像文件增加签名头：

   python3.7 */usr/local/software/*driver/source/vendor/hisi/tools/signtool/image_pack/esbc_header.py -raw_img Image -out_img header-Ascend310P.image -version 1.1.1.1.1 -nvcnt 0 -tag uimage -platform hi1910p

   python3.7 */usr/local/software/*driver/source/vendor/hisi/tools/signtool/image_pack/image_pack.py -raw_img header-Ascend310P.image -out_img Ascend310P.image -D -platform hi1910p

   - */usr/local/software/*为**步骤2**中{soc version}-driver-{software version}-minios.aarch64-src.tar.gz解压目录。
   - -raw_img：原始镜像文件路径，保持与示例一致即可。
   - -out_img：重新打包后的镜像文件输出路径，保持与示例一致即可。
   - -version：镜像版本号，格式为x.x.x.x.x用户可自定义。
   - -platform：请配置为hi1910p。
   - 其他参数请保持默认值即可。

   命令执行完成后，会在“kernel/linux-5.10/arch/arm64/boot”目录下生成增加签名头后的文件系统镜像文件Ascend310P.image。

7. 替换Driver安装目录下device文件夹下的的Ascend310P.image。

   须知

   替换前请先备份 /usr/local/Ascend/driver/device/目录下的Ascend310P.image文件到其他目录，防止内核镜像编译出错。

   a. 为Ascend310P.image文件添加写权限。

      chattr -i /usr/local/Ascend/driver/device/Ascend310P.image

b. 替换/usr/local/Ascend/driver/device/目录下的Ascend310P.image。

cp -rf /usr/local/software/driver/source/kernel/linux-5.10/arch/arm64/boot/Ascend310P.image /usr/local/Ascend/driver/device/

c. 取消Ascend310P.image文件的写权限。

chattr +i /usr/local/Ascend/driver/device/Ascend310P.image

8. 删除编译生成文件，为后续编译驱动源码做准备。

cd /usr/local/software/driver/source/kernel/linux-5.10

make mrproper

步骤5 在解压后的*/usr/local/software/driver*下的source目录，安装相关依赖并进行源码编译。

1. 进入驱动源码目录。

cd */usr/local/software/driver*/source

2. 编译驱动。

make driver_device product=mini CROSS_COMPILE=${INSTALL_DIR}/toolkit/toolchain/hcc/bin/aarch64-target-linux-gnu- KERNEL_DIR=*/usr/local/software*/driver/source/kernel/linux-5.10 KERNEL_DEFCONFIG=*/usr/local/software*/driver/source/kernel/linux-5.10/arch/arm64/configs/eulerosv2r9_defconfig build_device=true

- CROSS_COMPILE：为HCC交叉编译器所在目录，需要配置为绝对路径，其中${INSTALL_DIR}为CANN软件安装后文件存储路径，请根据实际路径替换。
- KERNEL_DIR：驱动源码文件存储路径，需要配置为绝对路径，*/usr/local/software/*请更换为驱动源码包所在目录。
- KERNEL_DEFCONFIG：驱动源码配置文件存储路径，需要配置为绝对路径，*/usr/local/software/*请更换为驱动源码包所在目录。
- 其他请保持不变。

说明

若进行源码编译时，一直循环提示类似如下信息：

```
scripts/kconfig/conf  --syncconfig Kconfig
make[4]: warning:  Clock skew detected.  Your build may be incomplete.
make[3]: warning:  Clock skew detected.  Your build may be incomplete.
make[2]: Warning: File '/usr/local/software/driver/source/kernel/linux-4.19/scripts/Makefile.ubsan' has modification time 38211501 s in the future
make[3]: Warning: File '/usr/local/software/driver/source/kernel/linux-4.19/arch/arm64/Makefile' has modification time 38211500 s in the future
make[4]: Warning: File '/usr/local/software/driver/source/kernel/linux-4.19/scripts/Makefile.host' has modification time 38211501 s in the future
  HOSTCC  scripts/basic/fixdep
```

请检查当前系统时间是否早于source/kernel/linux-5.10/scripts下的文件创建时间，若是，则请更改当前系统时间，然后重新进行编译操作。

命令执行成功后，会在source目录的out/device/文件夹下生成内核对象文件。

说明：

内核对象的源码都在driver/source/目录下。在driver/source/目录下通过**grep -r *xx.ko***命令，通过对应的module.mk文件，找到源码所在的文件夹，示例如下：

```
[root@localhost source]# grep -r -i drv_pcie.ko

… …

drivers/ai_sdk/arc/linux/kernel_space/src/drv_pcie_device/pcie_slave/module.mk:LOCAL_INSTALLED_KO_FILES := drv_pcie.ko
build/product_modules/device-ctrlcpu.mk:DRIVER_DEVICE_MODULES += drv_dtm.ko drv_nor_flash.ko drv_i2c.ko drv_e2prom.ko drv_pca6416.ko drv_
mng.ko drv_devmm.ko drv_pcie_hdc.ko drv_devdrv.ko ascend_event_sched.ko drv_dvpp.ko drv_log.ko drv_mdio.ko drv_gmac.ko drv_prof.ko drv_t
Binary file out/device/obj/drv_pcie_ko/scripts/mod/empty.o matches
out/device/obj/drv_pcie_ko/scripts/mod/devicetable-offsets.s:    .string "/usr/local/software/driver/source/out/device/obj/drv_pcie_ko"
out/device/obj/drv_pcie_ko/arch/arm64/kernel/asm-offsets.s:      .string "/usr/local/software/driver/source/out/device/obj/drv_pcie_ko"
```

则drv_pcie.ko对应的源码文件夹为drivers/ai_sdk/arc/linux/kernel_space/src/drv_pcie_device/pcie_slave。

步骤6 替换文件系统中的相关内核对象文件。

拷贝编译生成的*.ko到解压后的文件系统的var目录下。

cd */usr/local/software/driver*/source/out/device

cp -rf *.ko */usr/local/filesys_modify/tempdir*/var/

----结束

5.3.7 （PCIe 定制场景）修改 HDC 通信机制

简介

Host与Device之间的HDC（Host Device Communication）通信机制默认使用PCIe通道（HDC over PCIe），若用户不进行PCIe驱动定制，则可跳过本章节。

若用户自定义PCIe驱动，使用自定义的Host与Device之间的数据传输机制，则需要参考本章节将原有HDC修改为Socket通道（HDC over Socket），将PCIe通道留给用户自定义的PCIe驱动使用。

须知

Host的DCMI功能依赖HDC over PCIe，如果HDC修改为HDC over Socket，那么DCMI功能将无法使用。

操作步骤

步骤1 进入解压后文件系统的*/usr/local/filesys_modify*/tempdir/usr/local/etc/目录，以root用户执行如下命令创建hdcBasic.cfg文件。

vi hdcBasic.cfg

在hdcBasic.cfg文件中写入如下内容：

```
//TRANS_TYPE 0: USE SOCKET
//TRANS_TYPE 1: USE PCIe
TRANS_TYPE              = 0
SOCKET_SEGMENT          = 524288
PCIE_SEGMENT            = 524288
```

步骤2 执行**:wq**保存退出。

----结束

5.3.8 （PCIe 定制场景）日志功能配置

简介

PCIe驱动定制场景下，HDC修改为使用Socket通道，此种场景下，Device侧产生的日志无法通过日志服务slogd和驱动的HDC接口将日志信息转储到Host侧，所以此场景下日志进行打屏输出，用户可以对打屏日志做进一步处理。

日志打屏输出无需启动slogd和sklogd服务，所以需要修改Device侧的启动脚本，删除或注释掉slogd与sklogd服务相关内容。

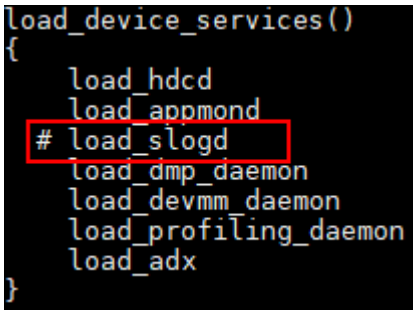
若用户不进行PCIe驱动定制，则可跳过本章节。

操作步骤

步骤1 以root用户执行以下操作，注释掉启动脚本（rcS文件）中的日志进程启动命令。

vi */usr/local/filesys_modify/tempdir/*var/device_sys_init.sh

步骤2 注释掉启动slogd的语句，如下所示：



步骤3 **执行:wq!**保存退出。

----结束

5.4 压缩文件系统

5.4.1 概述

用户在完成修改文件系统后，需按照本章节制作安全启动所需的密钥和证书，并将添加签名头的文件系统替换Device安装路径下的文件系统，确保设备能够安全可靠地启动。

说明

自CANN 6.0.RC1版本起，安全启动功能支持PSS填充模式，密钥长度变更为4096，若已参考以前版本文档进行配置，需参考本章节重新实现安全启动功能。

5.4.2 设置用户根证书信息

Device启动时会优先使用内置的华为证书校验文件系统，然后再使用用户证书进行校验。

- 若用户未对Device侧文件系统进行任何修改（即未执行**5.3 修改文件系统**的任何操作），此种情况下使用内置华为证书会校验通过，则不会再使用用户证书进行校验。
- 若Device侧文件系统有修改，则需要用户按照以下步骤上传用户根证书文件及CRL文件到Host指定目录/usr/local/Ascend/CMS，Device启动时需要使用用户证书对文件系统进行校验。

需要注意，一个芯片只需设置一次用户根证书信息(“user.xer”和“user.crl”)，若扩容芯片，需重新设置用户根证书信息。若芯片未变更，在用户根证书有效期内，用户可以一直使用已设置的根证书信息，实现安全启动功能。

操作步骤

步骤1 执行以下命令切换到root用户，后续操作都需要以root用户执行。

su - root

步骤2 获取用户根证书文件与用户CRL文件。参考**13.3.2.2 创建并激活根证书**生成的用户根证书文件**user.xer**和证书吊销列表文件**user.crl**。

步骤3 将用户根证书文件与用户CRL文件上传到Host侧指定目录。在Host侧驱动软件包（*{soc version}*-driver-*{software version}*-*{os.arch}*.run）的安装目录下创建CMS文件夹，执行如下命令在驱动默认安装路径“/usr/local/Ascend”创建CMS文件夹。

mkdir /usr/local/Ascend/CMS

步骤4 将根证书文件“user.xer”与证书吊销列表文件“user.crl”上传到上述步骤创建的CMS目录中，例如“ /usr/local/Ascend/CMS”。

步骤5 在Host侧使用upgrade-tool工具设置用户根证书信息。

- /usr/local/Ascend为Host侧驱动软件包的默认安装路径，若指定了安装路径，请根据实际情况替换。
- upgrade-tool工具的详细参数说明请参见《**Ascend 310P 23.0.RC2 npu-smi 命令参考（AI加速卡）**》。
- --device_index为设备编号，取值为“-1”时，指对当前host侧环境下所有的芯片进行烧录，若扩容芯片，需重新执行该步骤进行烧录。

/usr/local/Ascend/driver/tools/upgrade-tool --device_index -1 --user_cert --path /usr/local/Ascend/CMS/user.xer

----结束

5.4.3 部署修改后文件系统

操作步骤

步骤1 执行以下命令切换到root用户，后续操作都需要以root用户执行。

su - root

步骤2 解压**软件包获取**中的{soc version}-driver-{software version}-minios.aarch64-src.tar.gz。

解压后的driver/source/vendor/hisi/tools/signtool/image_pack目录中有加密与打包工具，用于文件系统压缩步骤。

例如在/usr/local/software/目录下执行如下解压命令：

tar -zxvf {soc version}-driver-{software version}-minios.aarch64-src.tar.gz

说明：如果进行**5.3.6 驱动源码编译**时已经解压了此源码包，则可跳过此步骤。

步骤3 进入*/usr/local/filesys_modify/*目录，并删除原有的raw-Ascend310P.cpio与Ascend310P.cpio.gz文件。

cd */usr/local/filesys_modify/*

rm raw-Ascend310P.cpio

rm Ascend310P.cpio.gz

步骤4 进入tempdir目录，并将tempdir目录中的内容打包成raw-Ascend310P.cpio.gz。

cd tempdir

find . | cpio -o -H newc | gzip > ../raw-Ascend310P.cpio.gz

命令执行完成后，会在上级目录filesys_modify目录下生成压缩后的文件系统raw-Ascend310P.cpio.gz。

步骤5 对文件系统镜像文件进行签名前预处理。

1. 切换到上一级的文件系统制作的filesys_modify目录下，并为文件系统增加签名头。

   cd *<u>/usr/local/filesys_modify</u>*

   python3 *<u>/usr/local/software/</u>*driver/source/vendor/hisi/tools/signtool/image_pack/esbc_header.py -raw_img raw-Ascend310P.cpio.gz -out_img header-Ascend310P.cpio.gz -version 1.1.1.1.1 -nvcnt 0 -tag initrd -platform hi1910p

   - */usr/local/software/*为**步骤2**中{soc version}-driver-{software version}-minios.aarch64-src.tar.gz解压目录。
   - -raw_img：原始镜像文件路径，保持与示例一致即可。
   - -out_img：重新打包后的镜像文件输出路径，保持与示例一致即可。
   - -version：镜像版本号，格式为x.x.x.x.x用户可自定义。
   - -platform：请配置为hi1910p。
   - 其他参数请保持默认值即可。

2. 执行如下命令，生成文件系统的摘要信息文件“**initrd.ini**”。

   digest=`sha256sum header-Ascend310P.cpio.gz | awk '{print $1}'`

   echo "initrd, ${digest};" > ./initrd.ini

步骤6 对摘要信息文件“**initrd.ini**”进行签名，详细操作请参见**13.3.2.3 生成开放态文件系统CMS签名文件**。

步骤7 将上述步骤生成的CMS签名文件initrd.ini.p7s和initrd.ini.crl拷贝到***<u>/usr/local/filesys_modify/</u>***目录。

步骤8 进入***<u>/usr/local/filesys_modify/</u>***目录，执行如下命令将文件系统和签名文件打包并增加签名头。

python3 *<u>/usr/local/software</u>*/driver/source/vendor/hisi/tools/signtool/image_pack/image_pack.py -raw_img ./header-Ascend310P.cpio.gz -out_img ./Ascend310P.cpio.gz -cms ./initrd.ini.p7s -ini ./initrd.ini -crl ./initrd.ini.crl --addcms -version 1.1.1.1.1 -platform hi1910p --pss

- */usr/local/software/*为**步骤2**中{soc version}-driver-{software version}-minios.aarch64-src.tar.gz解压目录。
- -raw_img：原始镜像文件路径，保持与示例一致即可。
- -out_img：重新打包后的镜像文件输出路径，保持与示例一致即可。
- -cms：CMS签名文件。
- -ini：签名文件的摘要信息。
- -crl：签名机构的吊销列表证书。
- -addcms：增加CMS签名的标志。

- -version：镜像版本号，格式为x.x.x.x.x用户可自定义。
- -platform：请配置为hi1910p。
- 其他参数请保持默认值即可。

命令执行完成后，会在filesys_modify目录下生成增加签名头后的文件系统镜像文件Ascend310P.cpio.gz。

步骤9 部署修改后的文件系统。替换Driver安装目录下device文件夹下的的Ascend310P.cpio.gz。

1. 为Ascend310P.cpio.gz文件添加写权限。

   chattr -i /usr/local/Ascend/driver/device/Ascend310P.cpio.gz

2. 替换/usr/local/Ascend/driver/device/目录下的Ascend310P.cpio.gz。

   须知

   替换前请先备份 /usr/local/Ascend/driver/device/目录下的Ascend310P.cpio.gz文件到其他目录，防止文件系统编译出错。

   cp -rf /*usr/local/filesys_modify/*Ascend310P.cpio.gz /usr/local/Ascend/driver/device/

3. 取消Ascend310P.cpio.gz文件的写权限。

   chattr +i /usr/local/Ascend/driver/device/Ascend310P.cpio.gz

文件系统替换完成后，为节省空间，可删除/usr/local/filesys_modify/下的tempdir文件夹。

步骤10 重启Host。

reboot

----结束

6 部署后检查

步骤1 重启后，使用root用户在Host侧执行如下命令，检查设备信息。

/usr/local/Ascend/driver/tools/upgrade-tool --device_index -1 --component -1 --version

输出示例如下所示，表示能正常获取Device侧设备信息，则表示Device正常启动。

```
{
Get component version(x.x.x.x.x) succeed for deviceId(0), componentType(xxx).
        {"device_id":xx, "component":xxx "version":xxx}
        {"device_id":xx, "component":xxx "version":xxx}
        ...
}
```

步骤2 登录Device端。

1. 在Host侧配置endvnic网卡的IP地址，IP地址需要与Device侧的默认IP地址（192.168.1.199）在同一网段。

   例如Host侧的endvnic网卡的IP地址配置为192.168.1.200。

   ifconfig endvnic 192.168.1.200

2. 在Host侧执行如下命令将Device默认IP地址192.168.1.199添加到信任IP地址列表中。

   ssh-keygen -f "/root/.ssh/known_hosts" -R 192.168.1.199

3. 执行如下命令，登录Device。

   ssh HwHiAiUser@192.168.1.199

   Device侧的HwHiAiUser用户的默认密码为“Huawei2012#”，首次登录要求修改密码。

   说明

   执行登陆命令时，提示Are you sure you want to continue connecting：

   The authenticity of host '192.168.1.199 (192.168.1.199)' can't be established.

   ECDSA key fingerprint is SHA256:4QpgDRdFWh+QhwvY +rSQbas8kE3gjhLTQ83rNCm8zhI.

   Are you sure you want to continue connecting (yes/no)?

   输入yes即可，随后就可以输入密码。

4. 登录Device后，执行如下命令切换到root用户。

   su - root

root用户的默认密码为Huawei12#$，首次登录要求修改密码。

说明

密码默认有效期为90天，您可以通过chage命令来设置用户的有效期，详情请参见**12.2 设置用户有效期**。

步骤3 在Device侧以root用户执行如下命令，查看运行的驱动信息。

lsmod |grep drv

如下图所示，则说明相关模块启动正常。

```
[root@(none) HwHiAiUser]# lsmod |grep drv
drv_spi                 20480  0
drv_osal               131072  0
drv_pwm                 20480  0
drv_upgrade             57344  0
drv_log                 69632  4
drv_prof               114688  0
drv_ts                  40960  0
drv_devmm              176128  2
drv_buff_module         24576  0
ascend_event_sched     139264  4 retr,drv_ts
drv_platform            40960  1 drv_log
drv_devdrv             466944  3 drv_log,drv_platform,drv_prof
drv_devmng             303104  21 drv_devdrv,drv_log,drv_devmm,retr,drv_platform,drv_ts,drv_prof
drv_davinci_intf        36864  0
drv_pkicms              36864  2 drv_devdrv,drv_upgrade
ascend_virtmng_dev      36864  2 drv_devmm,drv_devmng
ipc_drv                 94208  4 drv_devdrv,drv_log,drv_ts,drv_devmng
drv_pcie_hdc           196608  34
drv_pcie_vnic           45056  0
drv_pcie               258048  7 drv_devdrv,drv_pcie_vnic,drv_pcie_hdc,drv_devmm,drv_platform,ascend_virtmng_dev,drv_devmng
drv_dfm                 24576  3 drv_devdrv,drv_pcie,drv_devmng
drv_user_cfg            90112  2 drv_upgrade,drv_devmng
drv_ascend_ctl          16384  0
hsm_heartbeat           16384  2 drv_platform,drv_devmng
hsm_flash               16384  2 drv_user_cfg,drv_upgrade
hsm_firmware_update     20480  3 drv_pkicms,drv_upgrade,drv_devmng
drv_pm                  16384  0
mntn_bbox              274432  7 drv_devdrv,tzdriver,hsm_heartbeat,drv_dfm,drv_devmng
```

步骤4 在Device侧，切换到HwHiAiUser用户，查看AscendCL库文件是否在Device部署成功。

1. 执行如下命令，查看AscendCL库文件是否存在，且属组是否为HwHiAiUser。
   ls -l /home/HwHiAiUser/Ascend/lib64
2. 检查AscendCL库文件所在路径是否已添加到环境变量**LD_LIBRARY_PATH**。
   echo $LD_LIBRARY_PATH
   回显如下所示，代表runtime的环境变量设置成功。
   ```
   /home/HwHiAiUser/Ascend/lib64
   ```

说明

Device侧其他文件的修改，您都可以参考此方法进行确认，此处不再一一描述。

----结束

7 Device 侧业务包加载

7.1 简介

将用户开发的业务包部署到Device侧有以下两种方式：

- 修改Device侧的文件系统“**.cpio.gz”文件：将业务包部署到Device的文件系统中，并将此业务包的所属用户和用户组修改为HwHiAiUser，重新压缩并替换原有文件系统，然后重启Host，则业务包会自动加载到每个Device，可参见**步骤7**。

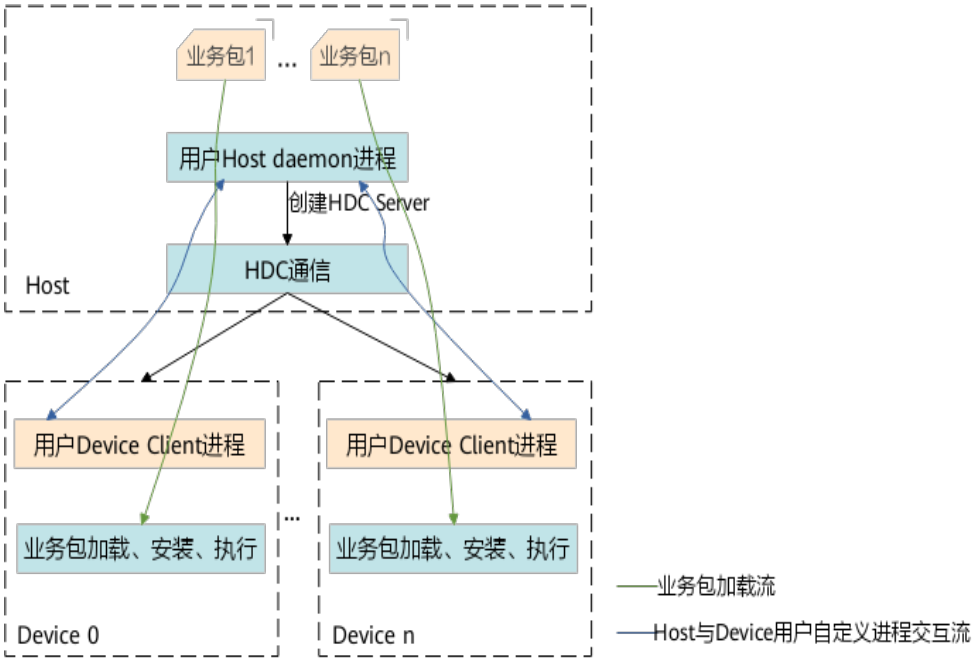
  此种方式适合所有Device上部署相同业务包的场景。
- 用户自定义实现Host与Device侧的相关进程，基于HDC实现Host与Device的消息交互与文件传输功能，实现系统启动时分别将不同的业务包加载到不同的Device。

  本章节详细介绍用户如何自定义实现将不同的业务包加载到不同的Device。

7.2 实现方案

Device加载不同业务包的总体实现方案如下图所示。

图 7-1 Device 加载业务包实现方案



用户需要自定义实现如下模块：

- 业务包。

  应用程序安装包，建议按照如下目录结构进行业务包的打包。

  ```
  业务包.tar.gz
  ├── package
  │   ├── xxx.so          //so文件
  │   ├── app             //可执行文件
  ├── scripts
  │   ├── install.sh      //安装脚本，把业务包的so文件和app可执行文件拷贝到指定目录
  │   ├── uninstall.sh    //卸载脚本，停止业务进程并删除安装文件
  │   ├── boot.sh         //业务启动脚本，启动业务进程
  ```

- Host侧的daemon进程，用于实现业务包的加载。
- Device侧的Client进程，负责进行业务包的安装和启动，此进程需要打包到Device侧的文件系统中。

说明

- Device侧业务包的安装用户请使用HwHiAiUser用户。
- Host的daemon进程与Device的client进程基于PCIe、HDC实现消息的交互及文件的传输，若用户自行修改了PCIe或HDC驱动代码导致PCIe HDC不可用，需要自行开发新的通信功能，新的通信功能需要支持Host与Device的消息通信和文件加载功能。

7.3 配置 H2D 文件传输白名单

简介

Device默认开启了H2D（Host to Device）文件传输时的白名单校验功能，对传输文件进行大小、存放目录及权限的校验，仅在白名单列表中的文件才允许使用HDC接口从Host传输到Device。

不同Device加载不同业务包的场景下，开发者需要调用HDC接口将业务包传输到Device，此种场景下，开发者需要参考本章节进行H2D文件传输白名单的配置。

操作步骤

H2D白名单校验文件为Device侧的“**/etc/hdcFileTransWhiteList.cfg**”文件，您需要在制作Device侧文件系统时修改此文件，文件系统的解压缩及压缩方法可参见**5 定制文件系统**，下面仅描述此文件的配置方法。

步骤1 切换到解压后的Device侧文件系统所在目录。

cd *[/usr/local/filesys_modify/tempdir](#)*

步骤2 执行如下命令为白名单校验文件“hdcFileTransWhiteList.cfg”添加写权限。

cd etc

chattr -i hdcFileTransWhiteList.cfg

chmod +w hdcFileTransWhiteList.cfg

步骤3 编辑白名单校验文件“hdcFileTransWhiteList.cfg”，追加需要进行传输的文件的校验规则。

须知

- 请在此文件中追加白名单校验规则，不允许修改文件中预置校验规则，否则会造成系统异常。
- 此文件中校验规则总数不能超过32条（含预置校验规则），多出的校验规则无法生效。
- 为避免文件被误修改，建议修改前先备份此文件。

请在此文件中追加需要进行传输的文件的校验规则，配置格式如下：

<文件名> <文件存储目的路径> <文件名匹配规则> <文件大小上限> <文件数量> <文件传输用户权限>

表 7-1 参数说明

参数	说明
文件名	字母、下划线、数字的组合，不能超过128个字节。

参数	说明
文件存储目的路径	● 配置为“*”，代表存储在Device侧可信路径下。 默认可信路径为：/home/HwHiAiUser/hdcd/devicex，devicex目录需要根据当前Device设备的id号确定，如果当前Device的设备ID是“0 ”，则为“device0”。 ● 配置为*/filepath*，代表存储在Device侧“可信路径/*filepath*”目录下。 其中*filepath*需要为已存在的路径，例如配置为“/temp”，则： 代表目的存储路径为：/home/HwHiAiUser/hdcd/devicex/temp，devicex目录需要根据当前Device设备的id号确定，如果当前Device的设备ID是“0 ”，则为“device0”。
文件名匹配规则	有如下取值，当前请开发者配置为“1”。 ● 1：代表全字匹配。 ● 0：代表部分匹配，部分匹配规则由于较为复杂，当前暂不建议开发者使用。
文件大小的上限	允许传输文件的大小的上限，单位为Byte。
文件的数量	全字匹配模式下，请配置为“1”，表示此条配置规则仅匹配一个文件，即“文件名”字段配置的文件。
文件传输用户权限	是否需要使用root用户进行传输，有以下两种取值： ● 1：是，需要使用root用户进行传输。 ● 0：否，不要求必须使用root用户进行传输，任意用户都可以。

配置示例：

```
service.tar.gz /service 1 55428800 1 0
```

步骤4 取消hdcFileTransWhiteList.cfg文件的写权限。

chmod -w hdcFileTransWhiteList.cfg

chattr +i hdcFileTransWhiteList.cfg

----结束

补充说明

开发者也可以通过重命名白名单校验文件（hdcFileTransWhiteList.cfg）的方式关闭H2D的校验功能，但H2D文件传输的白名单校验功能关闭后，从Host传输到Device的文件不会进行任何限制，需要用户自行控制文件数量与大小，否则会存在Device侧内存空间耗尽，导致应用程序申请不到内存的风险。

关闭H2D校验功能的参考步骤如下：

步骤1 切换到解压后的Device侧文件系统所在目录。

cd */usr/local/filesys_modify/tempdir*

步骤2 执行如下命令为白名单校验文件“hdcFileTransWhiteList.cfg”添加写权限。

cd etc

chattr -i hdcFileTransWhiteList.cfg

步骤3 重命名白名单校验文件。

mv hdcFileTransWhiteList.cfg hdcFileTransWhiteList.cfg_back

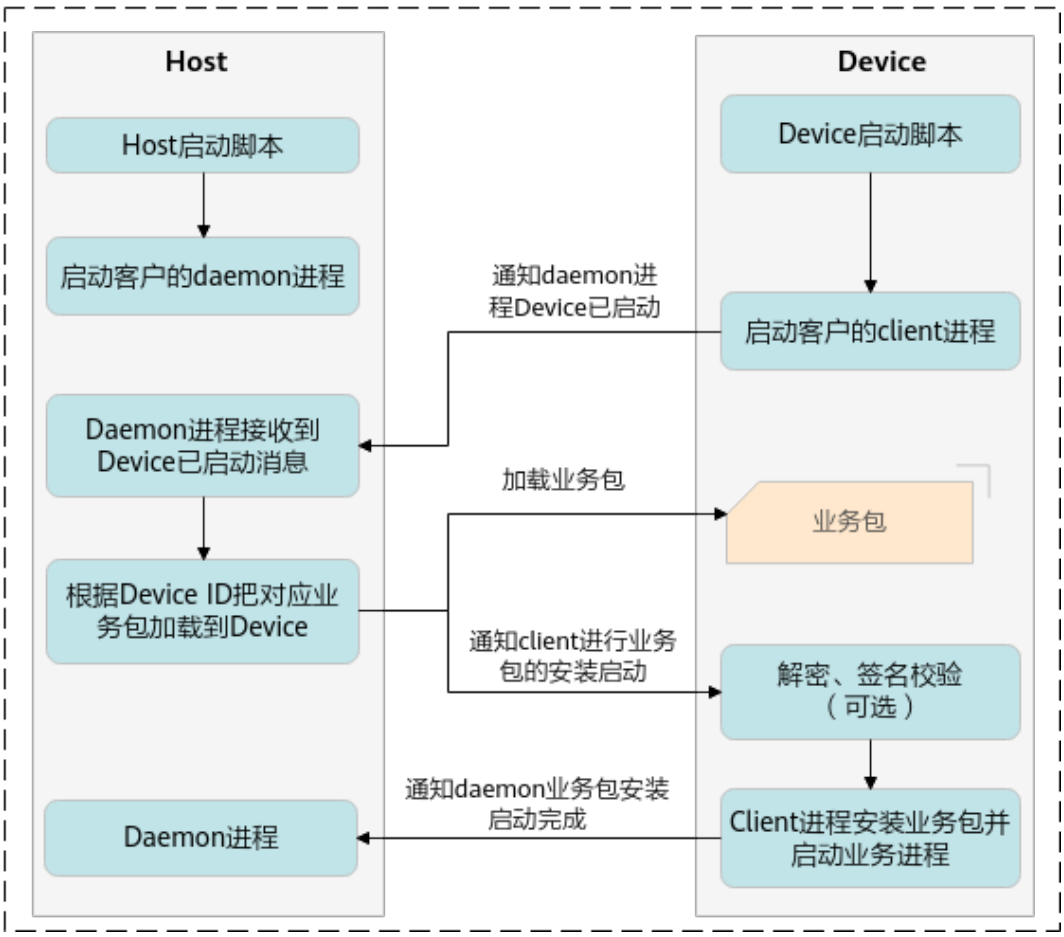
----结束

7.4 业务包加载

原理介绍

将不同的业务包加载到相应的Device的详细实现流程如下图所示。

图 7-2 Device 启动时业务包加载流程



1. Host启动时启动客户自定义实现的daemon进程，接收Device侧消息。
2. Device启动时，拉起客户自定义实现的client进程，client进程通过HDC连接到Host侧的客户daemon进程，并通知Host daemon进程Device已启动。

3. Host侧的daemon进程接收到Device已启动的消息后，通过HDC文件传输功能把相应的业务包拷贝到Device，并通知Device侧的client进程进行业务包的安装和启动。
4. Device侧client进程收到业务包安装消息后，进行业务包安装操作。

   首先进行业务包的解密和签名校验（可选，用户根据需要自行实现业务包的解密/签名校验），然后解压业务包，执行安装脚本进行安装，安装成功后执行启动脚本启动业务进程。
5. Device侧业务包安装完成后，通知Host侧daemon进程完成业务包的安装操作。

开发步骤

步骤1 业务包开发。

建议按照**业务包**中目录结构进行业务包的文件部署，包含库文件、可执行文件、安装脚本、卸载脚本以及业务进程启动脚本。

步骤2 Host侧daemon进程开发。

用户自定义实现Host侧的daemon进程，daemon进程需要包含如下功能：

- 通过HDC接收Device侧client进程发送的消息，并获取发送消息的Device ID。
- 当接收到Device侧client进程发送的Device已启动消息后，能够根据Device ID将业务包通过HDC的文件传输功能加载到Device侧；然后通知Device侧client进程进行业务包的安装。

详细代码示例请参见**10 HDC样例**。

步骤3 将Daemon进程启动脚本写入Host操作系统的启动脚本，实现Host启动时自动启动Daemon进程。

参考方法如下：

1. 在Host侧操作系统的/etc/init.d目录下创建一个Shell脚本，例如命名为Daemon_init.sh。
2. 在Daemon_init.sh中写入Daemon进程的启动命令。
3. 为Daemon_init.sh脚本添加可执行权限。

以上步骤即可以实现Daemon进程的开机启动。

步骤4 Device侧client进程开发。

用户自定义实现Device侧client进程，client进程需要包含如下功能：

- 通过HDC连接Host侧daemon进程，并发消息到daemon进程，并能接收Host侧daemon进程发送的消息。

  功能实现代码示例可参见**10 HDC样例**。
- 进行业务包的解密、签名校验（可选，用户根据需要自行实现业务包的解密/签名校验）。
- 调用业务包的安装脚本进行业务包的安装。
- 安装完成后，启动业务进程。

Client进程开发完成后，需要将其打包到Device侧文件系统中，文件系统的修改方法请参见**5 定制文件系统**。

步骤5 将client进程启动脚本写入Device系统启动脚本，实现Device启动时自动启动client进程。

您可以将client进程启动命令写入Device系统的rcS启动脚本中（**/etc/rc.d/init.d/rcS**文件）。

rcS文件需要通过修改Device侧文件系统的方式进行修改，文件系统的修改方法请参见**5 定制文件系统**。

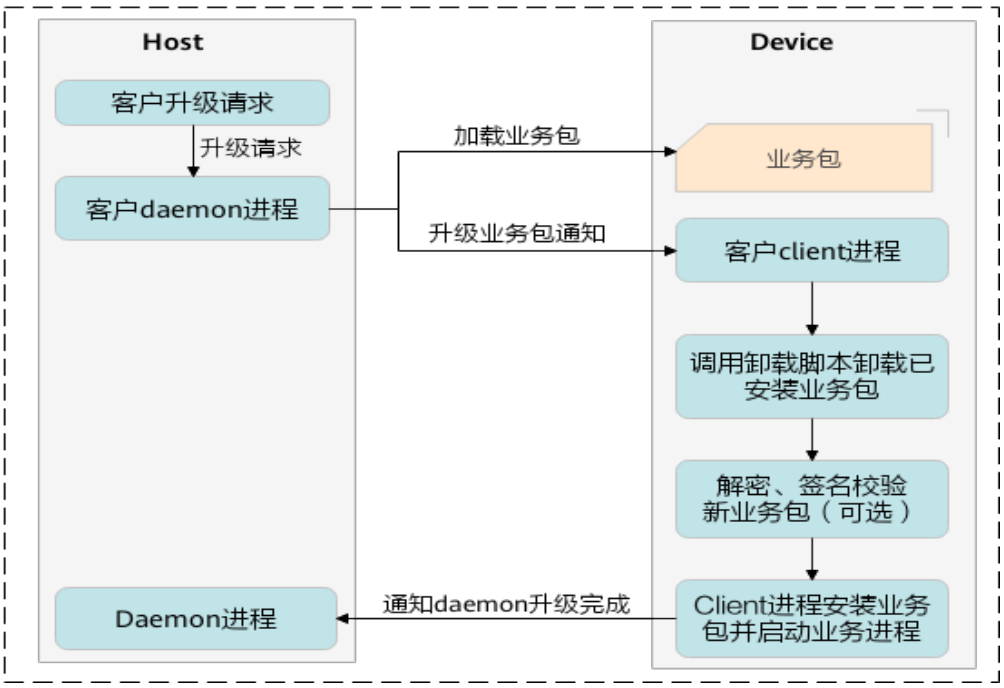
----结束

7.5 业务包升级

原理介绍

Device侧业务包升级的详细实现流程如下图所示。

图 7-3 业务包升级流程



1. 客户需要升级业务包时，首先通知Host侧的客户daemon进程进行业务包升级处理。
2. Daemon进程接收到升级请求后，首先通过HDC文件传输功能加载业务包到Device侧，然后发消息到Device侧的Client进程，通知Client进程进行业务包升级处理。
3. Device侧client进程接收到升级消息后，调用原业务包卸载脚本进行业务包卸载操作，包含：
   - 停止原业务进程。
   - 删除原业务包安装文件。
4. Device侧Client进程完成原业务包卸载操作后，进行新业务包安装操作。

首先进行业务包的解密和签名校验（可选，用户根据需要自行实现业务包的解密/签名校验），然后解压业务包，执行安装脚本进行安装，安装成功后执行启动脚本启动业务进程。

5. Device侧业务包安装完成后，通知Host侧daemon进程完成业务包的升级操作。

开发步骤

业务包升级流程的实现可参见**原理介绍**，业务包升级功能仅需要在业务包加载功能的基础上增加以下功能点：

- 客户端daemon进程增加对升级请求的处理，接收到升级请求后：
  a. 通过HDC加载业务包到Device。
  b. 并通知Device侧client进程进行业务包升级处理。
- Device侧client进程增加调用业务包卸载脚本的功能。

Host与Device间通过HDC进行消息通信及文件加载的代码示例可参见**10 HDC样例**。

8 使用 Memory Cgroup 机制实现业务内存资源控制

简介

昇腾AI软件栈提供了基于Cgroup（Control Groups）机制的内存限制功能，当业务进程的Cgroup达到特定的内存限额后（内存限额设置方法请参见**内存限额设置**），内核会尝试回收内存。

如果内存回收失败，业务进程会被挂起，后续有内存后再被唤醒，从而避免内存溢出（用户也可以参见**内存不足时应用退出设置**，实现内存回收失败时应用程序直接退出）。

添加业务进程到 Cgroup

开放形态下，用户需要自行在Device侧将业务应用进程添加到业务进程的cgroup中，添加方法如下：

以**root**用户执行如下命令：

echo *ServicePID* > /sys/fs/cgroup/memory/usermemory/cgroup.procs

ServicePID：业务应用程序的进程ID。

内存限额设置

- 内存限额的计算方法如下：

  limit = FreeMem - FreeCma - 预留大小

  - FreeMem：空闲内存，可通过“cat /proc/meminfo”命令获取“MemFree”的值，并将其单位转换为字节，即：MemFree*1024。
  - FreeCma：空闲的连续内存，可通过“cat /proc/meminfo”命令获取“CmaFree”的值，并将其单位转换为字节，即：CmaFree * 1024。
  - 预留大小：预留给不在user memory Cgroup中的服务进程使用，若设置太小可能会导致内存限额（limit）起不到限制作用，推荐设置为：昇腾AI处理器的个数*500*1024*1024 Byte。

- 内存限额的设置方法如下：

  echo limit > /sys/fs/cgroup/memory/usermemory/memory.limit_in_bytes

内存限额设置代码示例：


```
    free_mem=0
    MaxCpuID=`cat /proc/cpuinfo | grep "processor" | tail -1 | tr -cd "[0-9]"`
    free_cma=`cat /proc/meminfo | grep "CmaFree" | awk '{print $2}'`
    case ${MaxCpuID} in
    15)
        free_mem=$((free_mem + `cat /sys/devices/system/node/node1/meminfo | grep "Node 1 MemFree" |
awk '{print $4}'`))
        ;&
    7)
        free_mem=$((free_mem + `cat /sys/devices/system/node/node0/meminfo | grep "Node 0 MemFree" |
awk '{print $4}'`))
        ;;
    *)
        echo "modify usermem limit failed"
        return 0
        ;;
    esac

    ddr_node_num=$((MaxCpuID/8+1))
    mem_limit=$((free_mem * 1024 - ddr_node_num * 500 * 1024 * 1024 - free_cma * 1024))
    echo $((mem_limit)) > /sys/fs/cgroup/memory/usermemory/memory.limit_in_bytes
    echo $((mem_limit / 2)) > /sys/fs/cgroup/memory/usermemory/cust_aicpu/memory.limit_in_bytes
```

内存不足时应用退出设置

当业务进程的Cgroup达到特定的内存限额后，内核会尝试回收内存，如果内存回收失败，业务进程会被挂起。若用户不想一直等待，可执行如下操作，实现内存不足时应用程序自动退出。

步骤1 开启操作系统OOM Killer机制。

登录Device，以root用户执行如下命令启用enable_oom_killer，1表示启用，0表示禁用。

```
echo 1 > /proc/sys/vm/enable_oom_killer
```

步骤2 开启memory cgroup的OOM控制机制。

登录Device，以root用户执行如下命令启用memory cgroup的OOM机制，0表示“oom kill enable”，1表示“oom kill disable”。

```
echo 0 >  /sys/fs/cgroup/memory/usermemory/memory.oom_control
```

说明

memory Cgroup的kill enable功能生效的前提是系统enable_oom_killer开启。

----结束

9 HDC 接口参考

9.1 简介

为提供统一、高效的Host与Device间通信功能，昇腾AI软件栈提供HDC（Host Device Communication）模块。HDC对外提供类似Socket的接口，如Send/Recv/Close等，内部利用DMA（Direct Memory Access）加速，实现快速的内存搬移。

根据传输性能不同，HDC提供普通通道、快速通道两种解决方案。两种方案对比如下表所示：

通道类型	使用方法	性能	数据大小
普通通道	简单	一般	具体允许最大数据大小需要使用**drvHdcGetCapacity**接口查询。 允许的最大值范围：64Bytes~524224Bytes
快速通道	较复杂	高 （内存免拷贝）	最大512M Bytes

HDC接口头文件“ascend_hal.h”存储在{soc version}-driver-{software version}-minios.aarch64-src.tar.gz源码包中的“driver/source/inc/driver/”目录下。

须知

若使用GDB工具进行HDC接口的调试，GDB工具的版本需要为7.12及以后版本。

9.2 公共接口

9.2.1 drvHdcGetCapacity

函数功能

获取HDC的能力，可以获取链路类型以及普通通道数据块的大小。

函数原型

drvError_t drvHdcGetCapacity(struct drvHdcCapacity *capacity)

参数说明

参数名	输入/输出	说明
capacity	输出参数	包含链路类型及普通通道数据块允许最大值两个参数，详细参数说明可参见9.7.1 drvHdcCapacity。 • 链路类型：HDC over PCIe或者HDC over Socket。 普通通道的链路类型支持HDC over PCIe与HDC over Socket。 快速通道的链路类型仅支持HDC over PCIe。 • 普通通道数据块允许大小。 数据块大小决定了普通通道一次能收发的最大数据长度，一般在普通通道发送数据前调用该接口获取最大数据长度。

返回值说明

- 0：DRV_ERROR_NONE，成功
- 3：DRV_ERROR_INVALID_VALUE，参数错误

约束说明

在调用HDC的发送接口发送数据之前，需要通过该接口获取允许发送的最大数据长度。

9.2.2 drvHdcSetSessionReference

函数功能

设置HDC 会话（session）进程绑定，创建会话后，会话使用进程需调用该接口。例如进程A创建session，把该session给子进程B使用，则该接口在进程B中调用。

函数原型

drvError_t drvHdcSetSessionReference(HDC_SESSION session)

参数说明

参数名	输入/输出	说明
session	输入	已经创建的session。 类型：HDC_SESSION。

返回值说明

- 0：DRV_ERROR_NONE，成功
- 3：DRV_ERROR_INVALID_VALUE，参数错误
- 4：DRV_ERROR_INVALID_HANDLE，bind字符设备失败
- 17：DRV_ERROR_IOCRL_FAIL：ioctl命令失败。

约束说明

- 不支持孙子进程（即子进程的子进程）使用session。
- 如果创建session后不调用该接口，在进程异常后，若上层应用未做异常处理，HDC无法感知并释放对应session资源。

9.2.3 halHdcGetSessionInfo

函数功能

获取HDC会话的device id和fid信息，fid在算力虚拟化场景使用。

函数原型

drvError_t halHdcGetSessionInfo(HDC_SESSION session, struct drvHdcSessionInfo *info)

参数说明

参数名	输入/输出	说明
session	输入	获取会话信息的对象。 类型：HDC_SESSION。
info	输出	保存获取的信息。 类型：9.7.5 drvHdcSessionInfo。

返回值说明

- 0：DRV_ERROR_NONE，成功
- 3：DRV_ERROR_INVALID_VALUE，参数错误
- 4：DRV_ERROR_OPER_NOT_PERMITTED，无权限发访问
- 17：DRV_ERROR_IOCRL_FAIL：ioctl命令失败。

约束说明

需要在物理机场景下使用。

9.2.4 halHdcGetSessionAttr

函数功能

获取HDC会话创建者是否有root权限，以及HDC会话的运行环境、vfid等信息。

命令字在“enum drvHdcSessionAttr”中定义。运行环境、vfid等在虚拟化环境中使用。

函数原型

drvError_t halHdcGetSessionAttr(HDC_SESSION session, int attr, int *value)

参数说明

参数名	输入/输出	说明
session	输入	指定查询的会话对象。 类型：HDC_SESSION。

参数名	输入/输出	说明

attr	输入	命令字，取值范围如下：
		enum drvHdcSessionAttr {
		HDC_SESSION_ATTR_DEV_ID = 0, /* 获取设备id */
		HDC_SESSION_ATTR_UID = 1, /* 获取当前session进程是否有root权限 */
		HDC_SESSION_ATTR_RUN_ENV = 2, /* 获取当前环运行境类型 */
		HDC_SESSION_ATTR_VFID = 3, /* 获取vfid,虚拟化场景使用 */
		HDC_SESSION_ATTR_LOCAL_CREATE_PID = 4, /* 获取本端进程pid */
		HDC_SESSION_ATTR_PEER_CREATE_PID = 5, /* 获取对端进程pid */
		HDC_SESSION_ATTR_MAX
		};
		类型：int。

value	输出	● 当attr为HDC_SESSION_ATTR_RUN_ENV时，返回host的环境信息，类型如下： #define HDCDRV_SESSION_RUN_ENV_UNKNOW 0 /* 初始值 */ #define HDCDRV_SESSION_RUN_ENV_PHYSICAL 1 /* 表示物理环境 */ #define HDCDRV_SESSION_RUN_ENV_PHYSICAL_CONTAINER 2 /* 表示物理机中的容器环境 */ #define HDCDRV_SESSION_RUN_ENV_VIRTUAL 3 /* 表示虚拟机形态 */ #define HDCDRV_SESSION_RUN_ENV_VIRTUAL_CONTAINER 4 /* 表示虚拟机中容器场景 */ ● 当attr为HDC_SESSION_ATTR_DEV_ID时，返回session对应的设备ID。 ● 当attr为HDC_SESSION_ATTR_UID时，返回1为root权限，0为非root权限。 ● 当attr为HDC_SESSION_ATTR_VFID时，获取的是vfid。 ● 当attr为HDC_SESSION_ATTR_LOCAL_CREATE_PID时，获取的是本端进程的pid。 ● 当attr为HDC_SESSION_ATTR_PEER_CREATE_PID时，获取的是对端进程的pid。

返回值说明

- 0：DRV_ERROR_NONE，成功
- 3：DRV_ERROR_INVALID_VALUE，参数错误
- 17：DRV_ERROR_IOCRL_FAIL：ioctl命令失败。

约束说明

无

9.2.5 halHdcGetServerAttr

函数功能

获取HDC Server绑定的Device ID。

函数原型

hdcError_t halHdcGetServerAttr(HDC_SERVER server, int attr, int *value)

参数说明

参数名	输入/输出	说明
server	输入	指定的HDC Server对象。 类型：HDC_SERVER。
attr	输入	命令字，取值范围参考9.7.12 drvHdcServerAttr。 类型：int。
value	输出	获取的Device ID。 类型：int。

返回值说明

- 0：DRV_ERROR_NONE，成功
- 2：DRV_ERROR_INVALID_DEVICE，device id 入参值无效
- 3：DRV_ERROR_INVALID_VALUE，参数错误

约束说明

无

9.2.6 halHdcGetTransType

函数功能

获取当前HDC传输通道类型。

函数原型

hdcError_t halHdcGetTransType(enum halHdcTransType *transType)

参数说明

参数名	输入/输出	说明
transType	输出	获取传输通道类型。 类型：enum halHdcTransType。 transType = [HDC_TRANS_USE_SOCKET=0, HDC_TRANS_USE_PCIE=1]

返回值说明

- 0：DRV_ERROR_NONE，成功
- 3：DRV_ERROR_INVALID_VALUE，参数错误

约束说明

无

9.2.7 halHdcSetTransType

函数功能

设置当前进程的HDC传输通道类型。

函数原型

hdcError_t halHdcSetTransType(enum halHdcTransType transType)

参数说明

参数名	输入/输出	说明
transType	输入	指定当前进程的HDC传输通道类型。 类型：enum halHdcTransType。 transType = [HDC_TRANS_USE_SOCKET=0, HDC_TRANS_USE_PCIE=1]

返回值说明

- 0：DRV_ERROR_NONE，成功
- 3：DRV_ERROR_INVALID_VALUE，参数错误

- 4：DRV_ERROR_INVALID_HANDLE，句柄不合法或重复设置
- 8：DRV_ERROR_PARA_ERROR，参数错误
- 17：DRV_ERROR_IOCRL_FAIL，ioctl命令失败

约束说明

同一个进程只能在初始化时设置一次传输通道类型，重复设置会返回错误，且必须在调用其他所有接口之前调用该接口。

9.3 客户端接口

9.3.1 drvHdcClientCreate

函数功能

创建HDC Client进程，客户端必须执行。

函数原型

drvError_t drvHdcClientCreate(HDC_CLIENT *client, int maxSessionNum, int serviceType, int flag)

参数说明

参数名	输入/输出	说明
client	输出	创建成功的HDC Client句柄 类型：**HDC_CLIENT**。
maxSessionNum	输入	支持的最大会话数，及最大会话并发数，根据业务需要传入。 支持的会话数按Device分，类型：int，取值范围如下： 针对昇腾310P AI处理器 ，Device 侧创建client时maxSessionNum 取值范围：[1, 136)，Host侧支持的最大session数为64 x maxSessionNum；算力分组场景下，Device侧创建client时maxSessionNum 取值范围：[1, 136*16+8)，Host侧支持的最大seesion数为136*16+8。 针对AS31XM1X AI处理器，Host侧和Device侧支持的最大session数相同，在serviceType为0时最大session数为8，serviceType非0时最大session数为128。

参数名	输入/输出	说明
serviceType	输入	服务类型，详细参考9.7.10 drvHdcServiceType枚举类型。 类型：int。 **说明：** • 基于HDC接口进行业务开发的场景可以设置为“ 14”或者为预留服务类型“64~127”，一个Device上最多支持用户设置32个预留的服务类型。 • 若一个Device上起多个进程，每个进程使用的drvHdcServiceType不能相同。
flag	输入	预留参数，传入固定值“0”即可。 类型：int。

返回值说明

- 0：DRV_ERROR_NONE，成功
- 3：DRV_ERROR_INVALID_VALUE，参数错误
- 40：DRV_ERROR_MALLOC_FAIL，client申请内存失败

约束说明

无

9.3.2 drvHdcClientDestroy

函数功能

销毁HDC Client进程，不使用HDC后需要调用该接口释放资源。

函数原型

drvError_t drvHdcClientDestroy(HDC_CLIENT client)

参数说明

参数名	输入/输出	说明
client	输入	需要释放的HDC Client句柄。 类型：HDC_CLIENT。

返回值说明

- 0：DRV_ERROR_NONE，成功
- 3：DRV_ERROR_INVALID_VALUE，参数错误

- 23：DRV_ERROR_CLIENT_BUSY，session未释放

约束说明

释放Client之前需要保证其所有Session都已释放，否则会返回DV_ERROR_CLIENT_BUSY。

9.3.3 drvHdcSessionConnect

函数功能

发起HDC连接，服务端接收该链接请求后，该接口返回创建成功的会话（session）。

创建HDC Client进程后，且服务端处于监测状态时，即可调用该接口。

函数原型

drvError_t drvHdcSessionConnect(int peer_node, int peer_devid, HDC_CLIENT client, HDC_SESSION *session)

参数说明

参数名	输入/输出	说明
peer_node	输入	Device设备所在的节点的节点号，当前默认填写“0”。 类型：int。
peer_devid	输入	连接的设备的Device ID，范围：[0,64)。 类型：int。
client	输入	新创建的session对应的HDC Client句柄。 类型：**HDC_CLIENT**。
session	输出	创建好的session。 类型：**HDC_SESSION**。

返回值说明

- 0：DRV_ERROR_NONE，成功
- 2：DRV_ERROR_INVALID_DEVICE，设备号非法
- 3：DRV_ERROR_INVALID_VALUE，参数错误
- 6：DRV_ERROR_OUT_OF_MEMORY，申请session失败
- 19：DRV_ERROR_SOCKET_CONNECT：socket连接失败
- 35：DRV_ERROR_REMOTE_NOT_LISTEN，远端没监测

约束说明

无

9.3.4 drvHdcSessionConnectEx

函数功能

发起HDC连接，服务端接收该链接请求后，该接口返回创建成功的会话（session）。

创建HDC Client进程后，且服务端处于监测状态时，即可调用该接口。

函数原型

hdcError_t halHdcSessionConnectEx(int peer_node, int peer_devid, int peer_pid, HDC_CLIENT client, HDC_SESSION *pSession)

参数说明

参数名	输入/输出	说明
peer_node	输入	Device设备所在的节点的节点号，当前默认填写“0”。 类型：int。
peer_devid	输入	连接的设备的Device ID，范围：[0,64)。 类型：int。
peer_pid	输入	对端的进程pid号。 类型：int。
client	输入	新创建的session对应的HDC Client句柄。 类型：**HDC_CLIENT**。
session	输出	创建好的session。 类型：**HDC_SESSION**。

返回值说明

此接口的详细返回值如下：

- 0：DRV_ERROR_NONE，成功
- 2：DRV_ERROR_INVALID_DEVICE，设备号非法
- 3：DRV_ERROR_INVALID_VALUE，参数错误
- 6：DRV_ERROR_OUT_OF_MEMORY，申请session失败
- 19：DRV_ERROR_SOCKET_CONNECT，socket连接失败
- 34：DRV_ERROR_DEVICE_NOT_READY，等待设备响应
- 35：DRV_ERROR_REMOTE_NOT_LISTEN，远端没监测
- 44：DRV_ERROR_REMOTE_NO_SESSION，分配session失败

约束说明

无

9.3.5 drvHdcSessionClose

函数功能

会话使用完后，需要调用该接口关闭会话，释放资源。

函数原型

drvError_t drvHdcSessionClose(HDC_SESSION session)

参数说明

参数名	输入/输出	说明
session	输入	需要释放的session。 类型：HDC_SESSION。

返回值说明

- 0：DRV_ERROR_NONE，成功
- 3：DRV_ERROR_INVALID_VALUE，参数错误

约束说明

无

9.4 服务端接口

9.4.1 drvHdcServerCreate

函数功能

创建HDC Server进程，服务端必须执行，服务创建后，该服务默认进入监测状态。

函数原型

drvError_t drvHdcServerCreate(int devid, int serviceType, HDC_SERVER *pServer)

参数说明

参数名	输入/输出	说明
pServer	输出	创建成功的HDC Server。 类型：HDC_SERVER。

参数名	输入/输出	说明
devid	输入	设备的Device ID，取值范围：[0,64) 类型：int。
serviceType	输入	服务类型，详细定义请参见**9.7.10 drvHdcServiceType**。 类型：int。 **说明：** • 基于HDC接口进行业务开发的场景可以设置为“ 14”或者为预留服务类型“64~127”，一个Device上最多支持用户设置32个预留的服务类型。 • 若一个Device上起多个进程，每个进程使用的drvHdcServiceType不能相同。

返回值说明

- 0：DRV_ERROR_NONE，成功
- 2：DRV_ERROR_INVALID_DEVICE，设备号不合法
- 3：DRV_ERROR_INVALID_VALUE，参数错误
- 4：DRV_ERROR_INVALID_HANDLE，bind字符设备失败
- 17：DRV_ERROR_IOCRL_FAIL：ioctl命令失败
- 31：DRV_ERROR_SERVER_CREATE_FAIL，服务创建失败
- 34：DRV_ERROR_DEVICE_NOT_READY，等待设备响应
- 40：DRV_ERROR_MALLOC_FAIL，申请内存失败
- 61：DRV_ERROR_SERVER_HAS_BEEN_CREATED，Server已经创建成功

约束说明

无

9.4.2 drvHdcServerDestroy

函数功能

销毁HDC Server进程，服务进程退出或不再使用HDC Server时调用此接口释放资源。

函数原型

drvError_t drvHdcServerDestroy(HDC_SERVER server)

参数说明

参数名	输入/输出	说明
server	输入	需要销毁的HDC Server句柄。 类型：HDC_SERVER。

返回值说明

- 0：DRV_ERROR_NONE，成功
- 2：DRV_ERROR_INVALID_DEVICE，设备号非法
- 3：DRV_ERROR_INVALID_VALUE，参数错误
- 23：DRV_ERROR_CLIENT_BUSY，服务正在使用

约束说明

无

9.4.3 drvHdcSessionAccept

函数功能

类似socket Accept接口，调用该接口后，会阻塞，等待连接请求，连接请求过来后返回创建成功的会话（session）。

说明：此接口与socket Accept接口一样，需要尽量保持在阻塞状态，以免连接请求丢失。

函数原型

drvError_t drvHdcSessionAccept(HDC_SERVER server, HDC_SESSION *session)

参数说明

参数名	输入/输出	说明
server	输入	新创建的session所属的HDC Server。 类型：HDC_SERVER。
session	输出	创建的会话。 类型：HDC_SESSION。

返回值说明

- 0：DRV_ERROR_NONE，成功

- 2：DRV_ERROR_INVALID_DEVICE，设备号非法
- 3：DRV_ERROR_INVALID_VALUE，参数错误
- 22：DRV_ERROR_SOCKET_ACCEPT，连接失败
- 34：DRV_ERROR_DEVICE_NOT_READY，等待设备响应
- 40：DRV_ERROR_MALLOC_FAIL，内存分配失败

约束说明

此处会阻塞等待链接建立起来后，返回session指针。

9.4.4 drvHdcSessionClose

函数功能

会话使用完后，需要调用该接口关闭会话，释放资源。

函数原型

drvError_t drvHdcSessionClose(HDC_SESSION session)

参数说明

参数名	输入/输出	说明
session	输入	需要关闭的会话。 类型：**HDC_SESSION**。

返回值说明

- 0：DRV_ERROR_NONE，成功
- 3：DRV_ERROR_INVALID_VALUE，参数错误

约束说明

无

9.5 普通通道收发接口

9.5.1 drvHdcAllocMsg

函数功能

申请普通通道消息接收、发送所需描述符。

函数原型

drvError_t drvHdcAllocMsg(HDC_SESSION session, struct drvHdcMsg **ppMsg, int count)

参数说明

参数名	输入/输出	说明
session	输入	指定接收数据的session。 类型：HDC_SESSION。
ppMsg	输出	消息描述符指针，用于存放收发Buffer的地址和长度。 类型：**struct drvHdcMsg ***。
count	输入	消息描述符中的Buffer个数，当前只支持1个，固定传入1。 类型：int。

返回值说明

- 0：DRV_ERROR_NONE，成功
- 3：DRV_ERROR_INVALID_VALUE，参数错误
- 40：DRV_ERROR_MALLOC_FAIL，内存申请失败

约束说明

- 用户在收发数据前需要先申请消息描述符，使用完再释放消息描述符。
- 如果是用于发送，消息描述符中的Buffer需要由用户调用**drvHdcAddMsgBuffer**接口添加到消息描述符中，发送完成后由用户自行释放。
- 如果是用于offline场景接收，消息描述符中的Buffer在**drvHdcRecv**中申请并赋值，用户使用**drvHdcMsgGetBuffer**接口获取Buffer地址并使用，使用完调用**drvHdcFreeMsg**释放消息描述符和对应的消息指针。
- 如果是用于online场景接收，消息描述符中的Buffer在**drvHdcRecv**中直接将对端传递过来的Buffer地址赋值进去，用户使用**drvHdcMsgGetBuffer**接口获取Buffer地址并使用，使用完调用**drvHdcFreeMsg**释放消息描述符，对应的Buffer不做释放操作。

9.5.2 drvHdcFreeMsg

函数功能

释放普通通道用于收发消息的描述符。

函数原型

drvError_t drvHdcFreeMsg(struct drvHdcMsg *msg)

参数说明

参数名	输入/输出	说明
msg	输入	需要释放的消息描述符指针。 数据类型：9.7.7 drvHdcMsg。

返回值说明

- 0：DRV_ERROR_NONE，成功
- 3：DRV_ERROR_INVALID_VALUE，参数错误

约束说明

请参见约束说明。

9.5.3 drvHdcAddMsgBuffer

函数功能

消息发送之前，调用该接口，将要发送的数据buffer添加消息描述符中。

函数原型

drvError_t drvHdcAddMsgBuffer(struct drvHdcMsg *msg, char *pBuf, int len)

参数说明

参数名	输入/输出	说明
msg	输入	待操作的消息描述符指针。 数据类型：9.7.7 drvHdcMsg
pBuf	输入	待添加的数据buffer指针。 类型：char *。
len	输入	待添加的数据buffer有效数据长度，要求大于0。 类型：int。

返回值说明

- 0：DRV_ERROR_NONE，成功
- 3：DRV_ERROR_INVALID_VALUE，参数错误
- 6：DRV_ERROR_OUT_OF_MEMORY，没有可用的消息缓冲区

约束说明

请参见**约束说明**。

9.5.4 drvHdcGetMsgBuffer

函数功能

接收到消息后，从消息描述符中取出接收到的数据的内存地址及其长度。

函数原型

drvError_t drvHdcGetMsgBuffer(struct drvHdcMsg *msg, int index, char **pBuf, int *pLen)

参数说明

参数名	输入/输出	说明
msg	输入	待操作的消息描述符指针。 数据类型：**9.7.7 drvHdcMsg**。
pBuf	输出	接收到的数据的buffer指针。 类型：char *。
pLen	输出	接收到的buffer有效数据长度。 类型：int。
index	输入	接收到的buffer序列号，当前只支持1个，固定传0。 类型：int。

返回值说明

- 0：DRV_ERROR_NONE，成功
- 3：DRV_ERROR_INVALID_VALUE，参数错误

约束说明

请参见**约束说明**。

9.5.5 drvHdcReuseMsg

函数功能

重置消息描述符。

该接口会删除消息描述符中的数据buffer内存，但会保留消息描述符，以便消息描述符能重复使用。

调用该接口后就不需要单独释放数据buffer。

函数原型

drvError_t drvHdcReuseMsg(struct drvHdcMsg *msg)

参数说明

参数名	输入/输出	说明
msg	输入	待重用的消息描述符指针。 数据类型：**9.7.7 drvHdcMsg**。

返回值说明

- 0：DRV_ERROR_NONE，成功
- 3：DRV_ERROR_INVALID_VALUE，参数错误

约束说明

无

9.5.6 halHdcSend

函数功能

普通通道发送数据接口。

该接口会将上层传下来的Buffer地址和长度封装成消息发送到对端。

函数原型

hdcError_t halHdcSend(HDC_SESSION session, struct drvHdcMsg *pMsg, UINT64 flag, UINT32 timeout)

参数说明

参数名	输入/输出	说明
session	输入	指定发送数据的session。 类型：**HDC_SESSION**。
pMsg	输入	用于发送消息的描述符指针。 类型：**struct drvHdcMsg ***。

参数名	输入/输出	说明
flag	输入	HDC阻塞标志，有以下取值： ● 0：阻塞死等，如果发送通道阻塞则会等待通道空闲，死等。 ● 1：不阻塞，如果发送通道阻塞则不发送数据，立马返回。 ● 2：阻塞直到超时，如果发送通道阻塞则会等待通道空闲，直到超时。 建议使用“2”，用户可根据实际数据发送情况选择重发或者丢弃。 类型：UINT64。
timeout	输入	当“flag”取值为“2”时，此字段生效，用户可设置超时时间，单位毫秒。 此字段生效时，若timeout配置为“0”，系统默认设置超时时间3000ms， 类型：UINT32。

返回值说明

- 0：DRV_ERROR_NONE，成功
- 3：DRV_ERROR_INVALID_VALUE，参数错误
- 6：DRV_ERROR_OUT_OF_MEMORY，申请内存池失败
- 16：DRV_ERROR_WAIT_TIMEOUT，等待超时
- 25：DRV_ERROR_SOCKET_CLOSE，会话关闭
- 36：DRV_ERROR_NON_BLOCK，非阻塞没有数据
- 46：DRV_ERROR_OPER_NOT_PERMITTED，无权限访问

须知

halHdcSend接口返回后，用户就可以对内存进行复用或释放（halHdcSend接口底层有内存池进行数据的拷贝中转）。

约束说明

- 当返回值为“6：DRV_ERROR_OUT_OF_MEMORY”时，表示申请内存块失败，需等待一段时间资源被释放后，再重新发送数据。
- 当返回值为“16：DRV_ERROR_WAIT_TIMEOUT”时，表示等待超时时，需根据实际使用场景重新发送或者返回失败。
- 当返回值为“25：DRV_ERROR_SOCKET_CLOSE”时，表示会话关闭，如果是析构流程时返回的值，可以不打印错误打印。

9.5.7 halHdcRecv

函数功能

普通通道接收数据接口。

函数原型

hdcError_t halHdcRecv(HDC_SESSION session, struct drvHdcMsg *pMsg, int bufLen,UINT64 flag, int *recvBufCount, UINT32 timeout)

参数说明

参数名	输入/输出	说明
session	输入	指定接收数据的session。 类型：HDC_SESSION。
pMsg	输出	用于接收消息的描述符指针。 类型：drvHdcMsg。
flag	输入	HDC阻塞标志，有以下取值： • 0：阻塞死等，如果发送通道阻塞则会等待通道空闲，死等。 • 1：不阻塞，如果发送通道阻塞则不发送数据，立马返回。 • 2：阻塞直到超时，如果发送通道阻塞则会等待通道空闲，直到超时。
timeout	输入	当“flag”取值为“2”时，设置超时时间，单位毫秒。 类型：UINT32。
bufLen	输入	接收数据buffer的长度，单位为字节，预留未使用 类型：int。
recvBufCount	输出	接收到的数据buffer数量。 类型：int。

返回值说明

- 0：DRV_ERROR_NONE，成功
- 3：DRV_ERROR_INVALID_VALUE，参数错误
- 6：DRV_ERROR_OUT_OF_MEMORY，内存超限
- 16：DRV_ERROR_WAIT_TIMEOUT，等待超时
- 25：DRV_ERROR_SOCKET_CLOSE，会话关闭
- 26：DRV_ERROR_RECV_MESG，接收消息失败

- 36：DRV_ERROR_NON_BLOCK，非阻塞没有数据
- 40：DRV_ERROR_MALLOC_FAIL，内存分配失败
- 46：DRV_ERROR_OPER_NOT_PERMITTED，无权限访问

约束说明

该接口会自动根据接收数据长度申请内存并保存接收到的数据，用户取出数据后，可以调用drvHdcReuseMsg接口释放所有消息块资源，也可以调用drvHdcReuseMsg接口只释放数据buffer内存，保留消息块。

9.5.8 halHdcRecvEx

函数功能

普通通道接收数据接口，支持一次接收多个数据，并汇总到一块数据buffer内。

函数原型

hdcError_t halHdcRecvEx(HDC_SESSION session, struct drvHdcMsg *pMsg, int bufLen, int *recvBufCount, struct drvHdcRecvConfig *userConfig)

参数说明

参数名	输入/输出	说明
session	输入	指定接收数据的session。 类型：**HDC_SESSION**。
pMsg	输出	用于接收消息的描述符指针。 类型：**drvHdcMsg**。
bufLen	输入	该参数为预留参数，用于指定接收数据buffer的长度，单位为字节，当前未使用。 类型：int。
recvBufCount	输出	接收到的数据buffer数量。 类型：int。
userConfig	输入	传入用户相关配置 类型：**drvHdcRecvConfig**。

返回值说明

- 0：DRV_ERROR_NONE，成功
- 3：DRV_ERROR_INVALID_VALUE，参数错误
- 6：DRV_ERROR_OUT_OF_MEMORY，内存超限
- 16：DRV_ERROR_WAIT_TIMEOUT，等待超时

- 25：DRV_ERROR_SOCKET_CLOSE，会话关闭
- 26：DRV_ERROR_RECV_MESG，接收消息失败
- 36：DRV_ERROR_NON_BLOCK，非阻塞没有数据
- 40：DRV_ERROR_MALLOC_FAIL，内存分配失败
- 46：DRV_ERROR_OPER_NOT_PERMITTED，无权限访问

约束说明

该接口会自动根据接收数据长度申请内存并保存接收到的数据，用户取出数据后，可以调用**drvHdcReuseMsg**接口释放所有消息块资源，也可以调用**drvHdcReuseMsg**接口只释放数据buffer内存，保留消息块。该接口本身并不提供数据拆分功能，需要用户自行根据recvBufCount来拆分pMsg内的数据buffer。

9.5.9 drvHdcGetTrustedBasePath

函数功能

获取HDC文件传输可信路径，配合drvHdcSendFile接口使用。

例如：

针对昇腾310P AI处理器默认可信路径：

- Device为：/home/HwHiAiUser/hdcd/device0

  以上路径中的device0目录需要根据当前device设备的id号确定，如果当前Device的设备ID是“1 ”，路径最后一层即为“device1”。
- Host为：/var/hdcd。

函数原型

drvError_t drvHdcGetTrustedBasePath(int peer_node, int peer_devid, char *base_path, unsigned int path_len)

参数说明

参数名	输入/输出	说明
peer_node	输入	Device设备所在的节点的节点号，当前默认填写“0”。 类型：int
peer_devid	输入	设备的Device ID。 类型：int
base_path	输出	返回获取到的可信路径。 类型：char *
path_len	输入	获取到的可信路径的长度。 类型：int

返回值说明

- 0：DRV_ERROR_NONE，成功
- 3：DRV_ERROR_INVALID_VALUE，参数异常
- 6：DRV_ERROR_OUT_OF_MEMORY，路径对比或者拼接失败

约束说明

无

9.5.10 drvHdcSendFile

函数功能

HDC封装的文件传输接口，可发送文件到指定设备的指定路径。

函数原型

drvError_t drvHdcSendFile(int peer_node, int peer_devid, const char *file, const char *dst_path, void （*progress_notifier)(struct drvHdcProgInfo *))

参数说明

参数名	输入/输出	说明
peer_node	输入	Device设备所在的节点的节点号，当前默认填写“0”。 类型：int。
peer_devid	输入	从Host发往Device时，为设备的Device ID，范围：[0,64)。 从Device发往Host时，固定填写“0”。 类型：int。
file	输入	文件名，带相对路径或绝对路径的文件名。 类型：char *。
dst_path	输入	指定文件发送到接收端的路径（目的地址路径）；如果路径是目录（即不包含文件名），则文件发送到对端后保持文件名不变；否则文件发送到接收端后文件名改为路径中去除目录的部分。 文件接收路径需要取可信路径，可信路径请参见**9.5.9 drvHdcGetTrustedBasePath**。 类型：char *。
progress_notifier	输入	指定用户的回调处理函数。 当文件传输的进度增加至少百分之一时，文件传输协议会调用该接口。

返回值说明

- 0：DRV_ERROR_NONE，成功
- 3：DRV_ERROR_INVALID_VALUE，参数错误
- 52：DRV_ERROR_DST_PATH_ILLEGAL，目的地址错误
- 其他：失败

约束说明

无

9.6 快速通道收发接口

9.6.1 drvHdcMallocEx

函数功能

申请快速通道大页内存。

快速通道内存分“发送数据内存”“接收数据内存”“发送控制消息内存”“接收控制消息内存”四种，使用快速通道收发数据之前，根据使用场景需要提前申请内存。

函数原型

void *drvHdcMallocEx(enum drvHdcMemType mem_type, void *addr, unsigned int align, unsigned int len, int devid,unsigned int flag)

参数说明

参数名	输入/输出	说明
mem_type	输入	申请的内存类型 类型：**drvHdcMemType**。
addr	输出	申请快速通道内存成功后返回的虚拟地址 指定起始申请地址，从该地址后面找到一个可用页申请。 若传入默认值“NULL”，表示由系统指定。 类型：void *。
align	输入	对齐长度，仅支持4k（host+device）和2M（device侧），当前该参数预留，为后续扩展使用，保持默认值“0”即可。 类型：int。
len	输入	申请内存大小，取值范围：0~512M。 类型：int。

参数名	输入/输出	说明
devid	输入	设备的Device ID，取值范围：[0,64)。 类型：int。
flag	输入	内存申请标志，有以下取值： ● HDC_FLAG_MAP_VA32BIT：DVPP 4G限制内存空间的地址。 ● HDC_FLAG_MAP_HUGE：大页内存 ● 其他：普通内存 类型：int。

返回值说明

返回申请到的内存地址。

约束说明

调用内核函数申请物理内存，连续物理内存不足时会返回失败，如果是指定了DVPP用的4G限制内存范围内的空间申请，超出可能会申请失败，非DVPP请不要指定在该范围。

9.6.2 drvHdcFreeEx

函数功能

释放设备申请到的内存。

函数原型

drvError_t drvHdcFreeEx(enum drvHdcMemType mem_type, void *buf)

参数说明

参数名	输入/输出	说明
mem_type	输入	内存类型。 类型：**drvHdcMemType**。
buf	输入	申请到的内存地址。 类型：void *。

返回值说明

- 0：DRV_ERROR_NONE，成功

- 3：DRV_ERROR_INVALID_VALUE，参数错误
- 17：DRV_ERROR_IOCRL_FAIL，ioctl调用失败

约束说明

无

9.6.3 drvHdcDmaMap

函数功能

将快速通道内存与device进行映射，映射后该内存只能与对应的device通信。

如果**drvHdcMallocEx**接口申请内存时传入了正确的Device ID，则不需要单独调用该接口进行映射。

函数原型

drvError_t drvHdcDmaMap(enum drvHdcMemType mem_type, void *buf, int devid)

参数说明

参数名	输入/输出	说明
mem_type	输入	内存类型。 类型：**drvHdcMemType**。
buf	输入	申请到的内存地址。 类型：void。
devid	输入	设备的Device ID，取值范围：[0,64)。 类型：int。

返回值说明

- 0：DRV_ERROR_NONE，成功
- 3：DRV_ERROR_INVALID_VALUE，参数错误
- 17：DRV_ERROR_IOCRL_FAIL，ioctl调用失败

约束说明

无

9.6.4 drvHdcDmaUnMap

函数功能

将快速通道内存与Device ID解除映射。

函数原型

drvError_t drvHdcDmaUnMap(enum drvHdcMemType mem_type, void *buf)

参数说明

参数名	输入/输出	说明
mem_type	输入	内存类型。 类型：**drvHdcMemType**。
buf	输入	申请到的内存地址。 类型：void *。

返回值说明

- 0：DRV_ERROR_NONE，成功
- 3：DRV_ERROR_INVALID_VALUE，参数错误
- 17：DRV_ERROR_IOCRL_FAIL，ioctl调用失败

约束说明

无

9.6.5 drvHdcDmaReMap

函数功能

快速通道内存重映射，将内存从device A解映射，并重映射给device B。在重复使用大页内存给不同device发送数据情况下，可以调用该接口。

函数原型

drvError_t drvHdcDmaReMap(enum drvHdcMemType mem_type, void *buf, int devid)

参数说明

参数名	输入/输出	说明
mem_type	输入	内存类型。 类型：**drvHdcMemType**。
buf	输入	申请到的内存地址。 类型：void *。
devid	输入	设备的Device ID，取值范围：[0,64)。 类型：int。

返回值说明

- 0：DRV_ERROR_NONE，成功
- 3：DRV_ERROR_INVALID_VALUE，参数错误
- 17：DRV_ERROR_IOCRL_FAIL，ioctl调用失败

约束说明

无

9.6.6 halHdcFastSend

函数功能

HDC免拷贝快速发送消息接口。

函数原型

hdcError_t halHdcFastSend(HDC_SESSION session, struct drvHdcFastSendMsg msg, UINT64 flag, UINT32 timeout)

参数说明

参数名	说明
session	指定发送消息的session。 类型：**HDC_SESSION**。
msg	发送的消息。 类型：**drvHdcFastSendMsg**。
flag	阻塞标志。 ● 0：阻塞死等，如果发送通道阻塞则会等待通道空闲，死等。 ● 1：不阻塞，如果发送通道阻塞则不发送数据，立马返回。 ● 2：阻塞直到超时，如果发送通道阻塞则会等待通道空闲，直到超时。 类型：UINT64。
timeout	当flag取值为“2"时，设置阻塞超时时间，单位毫秒。 类型：UINT32。

返回值说明

- 0：DRV_ERROR_NONE，成功
- 3：DRV_ERROR_INVALID_VALUE，参数错误
- 16：DRV_ERROR_WAIT_TIMEOUT，发送超时
- 25：DRV_ERROR_SOCKET_CLOSE，session通道被关闭

- 27：DRV_ERROR_SEND_MESG，消息发送失败
- 46：DRV_ERROR_OPER_NOT_PERMITTED，无权限访问

须知

halHdcFastSend接口是异步发送接口，在发送数据时，底层通过DMA快速搬运，源地址一直都是使用的用户申请的内存地址。

因此，使用此接口时，建议用户对消息收发做二次管理，当对端接收到数据后，需要返回响应消息给发送端。在发送端收到响应消息后，用户就可以对内存进行释放或者复用。

约束说明

使用此接口时，需要提前通过HDC申请好内存。

halHdcFastSend、halHdcFastRecv接口，接收、发送数据buffer或控制消息buffer需同时给定。例如发送数据，则需要给定源数据大页内存地址和接收侧的目的大页内存地址。

针对AS31XM1X AI处理器，halHdcFastSend接口发送同一个地址的数据时，必须调用halHdcWaitMemRelease接口等待上一次发送完成，否则会造成数据覆盖和丢失。

9.6.7 halHdcFastRecv

函数功能

HDC免拷贝快速接收消息接口。

函数原型

hdcError_t halHdcFastRecv(HDC_SESSION session, struct drvHdcFastRecvMsg *msg, UINT64 flag, UINT32 timeout)

参数说明

参数名	说明
session	指定接收消息的session。 类型：HDC_SESSION。
msg	接收的消息。 类型：drvHdcFastSendMsg。
flag	阻塞标志。 • 0：默认值，阻塞。 • HDC_FLAG_NOWAIT：非阻塞。 类型：UINT64。

参数名	说明
timeout	当flag取值为“HDC_FLAG_NOWAIT"时，设置阻塞超时时间，单位毫秒。 类型：UINT32。

返回值说明

- 0：DRV_ERROR_NONE，成功
- 3：DRV_ERROR_INVALID_VALUE，参数错误
- 16：DRV_ERROR_WAIT_TIMEOUT，发送超时
- 25：DRV_ERROR_SOCKET_CLOSE，连接失败
- 26：DRV_ERROR_RECV_MESG，接收失败
- 46：DRV_ERROR_OPER_NOT_PERMITTED，无权限访问

约束说明

使用此接口时，需要提前通过HDC申请好内存。

halHdcFastSend、halHdcFastRecv接口，接收、发送数据buffer或控制消息buffer需同时给定。例如发送数据，则需要给定源数据大页内存地址和接收侧的目的大页内存地址。

9.6.8 halHdcRegisterMem

函数功能

HDC提供内存注册功能，允许用户向HDC注册外部申请的内存（如mBuf），用于快速通道传输。

函数原型

hdcError_t halHdcRegisterMem(signed int devid, enum drvHdcMemType mem_type, void *va, unsigned int len, unsigned int flag)

参数说明

参数名	输入/输出	说明
devid	输入	设备的Device ID，取值范围：[0,1]。 类型：int
mem_type	输入	注册的内存类型。 类型：**drvHdcMemType**。

参数名	输入/输出	说明
va	输入	输入的虚拟内存地址，来源为本进程其他接口申请（如mBuf），只支持已建好页表的内存，且要求4K对齐，同一个type内不允许重叠。 类型：void *。
len	输入	内存的长度，根据内存是大页还是小页，需要满足4K/2M对齐。 类型：unsigned int。
flag	输入	兼容/保留flag标志，目前填0即可

返回值说明

- 0：DRV_ERROR_NONE，成功
- 3：DRV_ERROR_INVALID_VALUE，参数错误
- 17：DRV_ERROR_IOCRL_FAIL，ioctl命令失败，详细error信息参见内核日志

约束说明

用户调用此接口的逻辑（主流程）应为：

1. 用户先调用mbuf接口halMbufAlloc等，并提取裸地址va;
2. 用户按需求调用halHdcRegisterMem将va注册（底层会完成dma和映射关系的发送）；
3. 用户调用halHdcSend/halHdcRecv交互获取彼此对端快速通道信息即va和len（data和ctrl数据）；
4. 用户构造速通道信息的send结构体（struct drvHdcFastSendMsg分为src/dst的data和ctrl数据，data和ctrl数据不全为null，dst的地址由步骤3获取）；
5. 用户调用halHdcFastSend发送数据;
6. 用户调用halHdcFastRecv接受数据（struct drvHdcFastRecvMsg分为data和ctrl数据，按需读取）。
7. 用户注册超过10M的内存时，必须使用huge page内存（2M page，page内连续）。

对同一个va地址重复注册会返回错误。

使用完的地址需要调用halHdcUnregisterMem接口进行解注册，否则会导致内存被pin住以及内存泄漏。

只支持在PCIe模式下调用。

9.6.9 halHdcUnregisterMem

函数功能

HDC提供内存解注册功能，允许用户将注册过的内存（如mBuf）解除注册。

函数原型

hdcError_t halHdcUnregisterMem(enum drvHdcMemType mem_type, void *va)

参数说明

参数名	输入/输出	说明
mem_type	输入	指定待解注册的内存的类型。 类型：**drvHdcMemType**。
va	输入	待解注册的虚拟内存地址，同一个type内不允许重叠。 类型：void *。

返回值说明

- 0：DRV_ERROR_NONE，成功
- 3：DRV_ERROR_INVALID_VALUE，参数错误
- 17：DRV_ERROR_IOCRL_FAIL，ioctl命令失败，详细error信息参见内核日志

约束说明

仅支持在PCIe模式下使用。

9.6.10 halHdcWaitMemRelease

函数功能

等待数据发送完成通知接口，在halHdcFastSend之后调用此接口，会阻塞直至send完成后返回，用于确保数据发送完成后进行内存的重复使用或释放。

函数原型

hdcError_t halHdcWaitMemRelease(HDC_SESSION session, int time_out, struct drvHdcFastRecvMsg *msg)

参数说明

参数名	输入/输出	说明
session	输入	指定接收消息的session。 类型：**HDC_SESSION**。
time_out	输入	设置阻塞超时时间，单位毫秒，-1 一直等待，0 立即返回，大于0表示超时时间。 类型：int。
msg	输出	发送端已完成dma搬移的内存的信息。 类型：**struct drvHdcFastRecvMsg**。

返回值说明

- 0：DRV_ERROR_NONE，正常返回
- 3：DRV_ERROR_INVALID_VALUE，参数错误
- 8：DRV_ERROR_PARA_ERROR，参数错误
- 16：DRV_ERROR_WAIT_TIMEOUT，超时
- 17：DRV_ERROR_IOCRL_FAIL，ioctl命令失败，详细error信息参见内核日志

约束说明

- 仅支持在PCIe模式使用。
- halHdcWaitMemRelease存储已搬移完的va数据的队列和session关联，且长度和快速通道recv接口相关配置关联，size=128，（在1个session发送128个va range之前执行halHdcWaitMemRelease取出数据），超过最大深度，在send处理，返回失败。
- 不支持多线程同时执行halHdcWaitMemRelease (与recv保持一致)。
- halHdcFastSend接口发送同一个地址的数据时，必须调用此接口等待上一次发送完成，否则会造成数据覆盖和丢失。

9.7 数据结构说明

9.7.1 drvHdcCapacity

功能

用于查看当前HDC支持的数据块大小以及通道类型（PCIe或者Socket）。

定义原型

```
struct drvHdcCapacity {
    enum drvHdcChanType chanType; /* 当前 hdc 的通道类型；0是socket，1是pcie */
```

```
    unsigned int maxSegment; /* HDC支持的最大数据块大小 */
};
```

9.7.2 HDC_SESSION

功能

一个 void * 的指针，指针指向的地方包含了建链之后的 session 信息。

定义原型

```
typedef void *HDC_SESSION;
```

成员介绍

- client端，指向的session信息如下：

```
struct hdcClientSession {
    struct hdcSession session; /* client 端 session 信息 */
    bool alloc; /* client 端 session 是否已经在使用中；true 表示被使用中，false 表示没有被使用中 */
    INT32 node; /* 当前设备所在的节点数；目前只支持一个设备，默认传 0 */
    INT32 devid; /* host侧看到的device侧的设备ID号 */
    UINT32 portno; /* 目前未使用 */
    UINT32 servaddr; /* 目前未使用 */
    struct hdcClientHead *client; /* client信息，请参见9.7.4 HDC_CLIENT */
};
```

- server端，指向的session信息如下：

```
struct hdcServerSession {
    struct hdcSession session; /* server 端 session 信息 */
    UINT32 deviceId; /*  */
    struct sockaddr_in clientAddr; /* socket模式下使用，记录client地址 */
    struct hdcServerHead *server; /* server信息，请参见9.7.3 HDC_SERVER */
};
```

- 上面两个结构体中，公共的 hdcSession 的定义如下所示：

```
struct hdcSession {
    UINT32 magic; /* hdc 魔术字，防止被踩 */
    UINT32 device_id; /* 当前设备的 device id 号 */
    INT32 sockfd; /* 当前 session 通道的 fd 号 */
    UINT32 type; /* 当前 session 通道是 client 还是 server；0 是 server，1 是 client */
    mmProcess bind_fd; /* 当前进程绑定的底层字符设备的 fd 号 */
};
```

9.7.3 HDC_SERVER

功能

一个 void * 的指针，指针指向的地方包含了 server 端信息。

定义原型

```
typedef void *HDC_SERVER;
```

成员介绍

```
struct hdcServerHead {
    UINT32 magic; /* hdc 魔术字，防止被踩 */
    INT32 serviceType; /* hdc server 端的服务类型，参考9.7.10 drvHdcServiceType */
    UINT32 sessionNum; /* 记录此 server 上当前共有多少个 session 通道在使用中 */
    UINT32 portno; /* socket 模式中使用，记录服务类型端口，pcie模式没有实际意义 */
    UINT32 servaddr; /* socket模式使用，记录server端地址 */
    mmSockHandle listenFd;/* 服务监测 fd；socket上指 socket 监测，pcie 上等于 device id */
    INT32 deviceId; /* 当前 server 所在的设备 id 号 */
```



```
    mmProcess bind_fd; /* 当前进程绑定的底层字符设备的 fd 号 */
    mmMutex_t mutex; /* server的锁 */
    struct hdcServerSession *pSession;/* 和 session 通道结构体互相包含 */
};
```

9.7.4 HDC_CLIENT

功能

一个 void * 的指针，指针指向的地方包含了 client 端信息。

定义原型

```
typedef void *HDC_CLIENT;
```

成员介绍

```
struct hdcClientHead {
    UINT32 magic; /* hdc 魔术字，防止被踩 */
    INT32 serviceType; /* hdc server 端的服务类型，请参见9.7.10 drvHdcServiceType */
    UINT32 flag; /* 当前无实际意义，赋值为0，方便以后扩展使用 */
    UINT32 maxSessionNum; /* client支持的最大的 session 通道数 */
    mmMutex_t mutex; /* client 锁 */
    struct hdcClientSession session[0]; /* 和 session 通道结构体互相包含 */
};
```

9.7.5 drvHdcSessionInfo

功能

获取当前 session 通道的 Device ID，以及 fid 信息（fid 可以确定当前 session 是在物理机上还是虚拟机上；如果在虚拟机上，可以根据 fid 定向到具体的虚拟机）等。

定义原型

```
struct drvHdcSessionInfo {
    unsigned int devid; /* 当前 session 所在的 device 设备号 */
    unsigned int fid; /* 当前 session 的 fid 号；用来区分物理机和虚拟机等 */
    unsigned int res[HDC_SESSION_INFO_RES_CNT]; /* 当前无实际意义，扩展预留 */
};
```

9.7.6 drvHdcMsgBuf

功能

HDC MSG Buffer结构体定义；可以知道消息的 buf 地址，以及消息长度。

定义原型

```
struct drvHdcMsgBuf {
    char *pBuf; /* msg 消息 buf 地址 */
    int len; /* 消息 buf 的长度 */
};
```

9.7.7 drvHdcMsg

功能

HDC MSG描述符结构体定义；包含消息的数量，以及消息 buf 的指针信息。

定义原型

```
struct drvHdcMsg {
    int count; /* 消息的数量，当前只支持一个，默认传 1 */
    struct drvHdcMsgBuf bufList[0]; /* 消息的结构体指针；当前默认支持一个 */
};
```

9.7.8 drvHdcMemType

功能

HDC支持的内存类型。

定义原型

```
enum drvHdcMemType {
    HDC_MEM_TYPE_TX_DATA = 0, /* 发送数据消息类型 */
    HDC_MEM_TYPE_TX_CTRL = 1, /* 发送控制消息类型 */
    HDC_MEM_TYPE_RX_DATA = 2, /* 接收数据消息类型 */
    HDC_MEM_TYPE_RX_CTRL = 3, /* 接收控制消息类型 */
    HDC_MEM_TYPE_DVPP = 4,    /* DVPP数据类型 */
    HDC_MEM_TYPE_ANY = 5，     /* 任意数据类型 */（仅适用于AS31XM1X AI处理器）
    HDC_MEM_TYPE_MAX          /* 总的数据类型支持数量 */
};
```

9.7.9 drvHdcFastSendMsg

功能

HDC 使用快速通道发送消息的描述符指针。

定义原型

```
struct drvHdcFastSendMsg {
    unsigned long long srcDataAddr; /* 源数据地址 */
    unsigned long long dstDataAddr; /* 目的数据地址 */
    unsigned long long srcCtrlAddr; /* 源控制消息地址 */
    unsigned long long dstCtrlAddr; /* 目的控制消息地址 */
    unsigned int dataLen;           /* 数据内容长度 */
    unsigned int ctrlLen;           /* 控制消息长度 */
};
```

9.7.10 drvHdcServiceType

功能

HDC 支持的service类型。

定义原型

```
enum drvHdcServiceType {
    HDC_SERVICE_TYPE_DMP = 0, /* dmp模块使用的服务类型 */
    HDC_SERVICE_TYPE_PROFILING = 1, /* profiling模块使用的服务类型 */
    HDC_SERVICE_TYPE_IDE1 = 2, /* ide模块使用的服务类型 */
    HDC_SERVICE_TYPE_FILE_TRANS = 3, /* 文件收发(hdcd模块)使用的服务类型 */
    HDC_SERVICE_TYPE_IDE2 = 4, /* ide模块使用的服务类型 */
    HDC_SERVICE_TYPE_LOG = 5, /* log模块使用的服务类型 */
    HDC_SERVICE_TYPE_RDMA = 6, /* rdma模块使用的服务类型 */
    HDC_SERVICE_TYPE_BBOX = 7, /* bbox黑匣子模块使用的服务类型 */
    HDC_SERVICE_TYPE_FRAMEWORK = 8, /* framework模块使用的服务类型 */
    HDC_SERVICE_TYPE_TSD = 9, /* TSD推理模块使用的服务类型 */
```



```
    HDC_SERVICE_TYPE_TDT = 10, /* TDT推理模块使用的服务类型 */
    HDC_SERVICE_TYPE_PROF = 11, /* prof服务类型 */
    HDC_SERVICE_TYPE_IDE_FILE_TRANS = 12, /* ide模块的数据收发服务类型 */
    HDC_SERVICE_TYPE_DUMP = 13, /* dump模块预留的服务类型 */
    HDC_SERVICE_TYPE_USER3 = 14, /* 供业务开发使用 */
    HDC_SERVICE_TYPE_DVPP = 15, /* dvpp模块服务类型 */
    HDC_SERVICE_TYPE_QUEUE = 16, /* 队列模块使用的服务类型 */
    HDC_SERVICE_TYPE_UPGRADE = 17, /* 升级使用的服务类型 */
    HDC_SERVICE_TYPE_USER_START = 64, /* 为客户预留的服务类型，64为第一个服务类型 */
    HDC_SERVICE_TYPE_USER_END = 127, /* 为客户预留的服务类型，127为最后一个服务类型 */
    HDC_SERVICE_TYPE_MAX /* 支持的最多的服务类型数量 */
};
```

说明

64~127为当前为客户预留的服务类型，一个Device上最多支持用户设置32个预留的服务类型，同一个Device上不同的进程需要使用不同的服务类型。

9.7.11 drvHdcProgInfo

功能

这个结构体是用来统计HDC传输文件时，传输文件的总大小，传输速度，以及耗时等信息；

定义原型

```
struct drvHdcProgInfo {
    char name[256]; /* 传输的文件的名称描述 */
    int progress; /* 传输完成的百分比 */
    long long int send_bytes; /* 传输的数据的大小 */
    long long int rate; /* 传输数据的速度 */
    int remain_time;/* 传输文件的耗时 */
};
```

9.7.12 drvHdcServerAttr

功能

针对 halHdcGetServerAttr 接口的一个枚举，主要是指定 halHdcGetServerAttr 接口所要完成的操作；根据类型，查询 server 端的对应的信息；

定义原型

```
enum drvHdcServerAttr {
    HDC_SERVER_ATTR_DEV_ID = 0, /* 获取server 端的设备 id */
    HDC_SERVER_ATTR_MAX /* 支持的类型总数量 */
};
```

9.7.13 drvHdcRecvConfig

功能

用户在使用**halHdcRecvEx**接口时，可通过drvHdcRecvConfig结构体传入用户自行设定的配置。

定义原型

```
struct drvHdcRecvConfig {
    UINT64 wait_flag; /* HDC阻塞标识，取值同halHdcRecv接口的参数flag */
```



```
    UINT32 timeout; /* 当“flag”取值为“2”时，需要设置超时时间，单位毫秒, 含义同halHdcRecv的参数timeout */
    int group_flag; /* HDC接收模式标识， 0表示hdc 1次取1个数据buffer， 1表示hdc 1次取多个数据buffer */
    int reserved_params1; /* 预留参数，暂不使用 */
    int reserved_params2; /* 预留参数，暂不使用 */
    int reserved_params3; /* 预留参数，暂不使用 */
    int reserved_params4; /* 预留参数，暂不使用 */
};
```

9.7.14 drvError_t

功能

HDC接口的返回值类型。

定义原型

```
typedef enum tagDrvError {
    DRV_ERROR_NONE = 0,                 /* 成功 */
    DRV_ERROR_NO_DEVICE = 1,            /* 无合法设备 */
    DRV_ERROR_INVALID_DEVICE = 2,       /* 设备号非法 */
    DRV_ERROR_INVALID_VALUE = 3,        /* 参数值非法 */
    DRV_ERROR_INVALID_HANDLE = 4,       /* 句柄不合法 */
    DRV_ERROR_INVALID_MALLOC_TYPE = 5, /* malloc类型无效 */
    DRV_ERROR_OUT_OF_MEMORY = 6,        /* 内存溢出 */
    DRV_ERROR_INNER_ERR = 7,            /* 内部错误 */
    DRV_ERROR_PARA_ERROR = 8,           /* 参数错误 */
    DRV_ERROR_UNINIT = 9,               /* driver未初始化 */
    DRV_ERROR_REPEATED_INIT = 10,       /* driver重复初始化 */
    DRV_ERROR_NOT_EXIST = 11,
    DRV_ERROR_REPEATED_USERD = 12,
    DRV_ERROR_BUSY = 13,                /* 任务已经在运行 */
    DRV_ERROR_NO_RESOURCES = 14,        /* 资源短缺 */
    DRV_ERROR_OUT_OF_CMD_SLOT = 15,
    DRV_ERROR_WAIT_TIMEOUT = 16,        /* 发送超时 */
    DRV_ERROR_IOCRL_FAIL = 17,          /* ioctl命令失败 */
    ...
}
```

完整定义可参见{soc version}-driver-{software version}-minios.aarch64-src.tar.gz源码包中的“driver/source/inc/driver/ascend_hal_error.h“文件。

9.7.15 hdcError_t

HDC接口返回值类型，定义如下所示：

```
typedef drvError_t hdcError_t;
```

关于drvError_t的详细说明，请参见**9.7.14 drvError_t**。

9.7.16 halHdcTransType

HDC通信类型，定义如下所示：

```
enum halHdcTransType {
    HDC_TRANS_USE_SOCKET = 0,
    HDC_TRANS_USE_PCIE = 1
};
```

9.7.17 drvHdcFastRecvMsg

HDC 使用快速通道接收消息的描述符指针。

```
struct drvHdcFastRecvMsg {
    unsigned long long dataAddr; /* 数据消息接收地址 */
    unsigned long long ctrlAddr; /* 控制消息接收地址 */
    unsigned int dataLen;          /* 数据内容长度 */
    unsigned int ctrlLen;         /* 控制消息长度 */
};
```

9.8 多进程场景使用注意事项

为了支持多进程业务，HDC在drvHdcServiceType类型为14的基础上，提供了扩充业务类型64~127供开发者使用，详情可参见**9.7.10 drvHdcServiceType**，开发者可以为不同的进程分配不同的serviceType。

每一个通信的客户端进程与服务端进程之间使用serviceType为标识进行配对，若开发者为不同的进程规划不同的serviceType，服务端可通过如下方式获取客户端的serviceType（不局限于以下列举方法）：

- 方式一，根据业务需要，规划不同的serviceType，并将规划的serviceType以常量写入代码中。
- 方式二，将serviceType写入文件，并将文件传递到对端，然后对端通过文件获取serviceType。
  a. 客户端创建HDC进程后，通过**drvHdcSessionConnect**接口获取当前进程的serviceType。
  b. 将获取到的serviceType写入文件，然后通过SCP方式或者**drvHdcSendFile**接口将文件发送到对端。

     需要注意，当前版本**drvHdcSendFile**接口仅支持将文件从Host发送到Device。
  c. 服务端通过文件获取客户端serviceType，然后通过调用**drvHdcSessionAccept**接口等待客户端的连接请求。

10 HDC 样例

10.1 样例代码

```
/*
 * Copyright (c) Huawei Technologies Co., Ltd. 2012-2020. All rights reserved.
 * Description:
 * Author: huawei
 * Create: 2020-10-01
 */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/types.h>
#include <unistd.h>

#include "ascend_hal.h"

/* ****************************************** */
/* 取值范围 [1-96]:                           */
/* 一个 device 最多支持 96 个 session 通道，*/
/* 创建 client 时，建议传参小于等于 96      */
/* ****************************************** */
#define MAX_SESSION_NUM 96

/* ************************************************************** */
/* 0：阻塞死等，如果收发通道阻塞则会等待通道空闲，死等。          */
/* 1：不阻塞，如果收发通道阻塞则不发送数据，立马返回。            */
/* 2：阻塞直到超时，如果收发通道阻塞则会等待通道空闲，直到超时。 */
/* 建议使用“2”，用户可根据实际数据发送情况选择重发或者丢弃。    */
/* ************************************************************** */
#define HDC_WAIT_TIMEOUT 2

/* ************************************************ */
/* 单位 ms；用户根据业务设置合适的收发通道超时时间值 */
/* ************************************************ */
#define HDC_SEND_RECV_WAIT_TIME 100

#define HDC_SEND_RECV_DATA_LEN 256
#define HDC_DST_PATH_LEN 256
#define HDC_CHANGE_FRAME_SIZE_TO_KB 1024

#define HDC_ERROR(fmt, ...) printf("Line: %04d. %s]" fmt "\n",  __LINE__, __func__, ## __VA_ARGS__)
```

```
#define HDC_INFO(fmt, ...) printf("[Line: %04d. %s]" fmt "\n",  __LINE__, __func__, ## __VA_ARGS__)

void usage()
{
    printf("-------------------------------------------------------------------------------------------\n");
    printf( "normal channel, usage:\n"
    "\t./hdc_demo 0 server/client(1/0) server_type(14) dev_id(0~64) size(KB) loop_times \n");
    printf( "fast channel, usage:\n"
    "\t./hdc_demo 1 server/client(1/0) server_type(14) dev_id(0~64) size(KB) loop_times \n");
    printf( "send file channel, usage:\n"
    "\t./hdc_demo 2 dev_id(0~64) file \n");
    printf("-------------------------------------------------------------------------------------------\n");
    printf( "normal channel, examples:\n"
    "\tserver:./hdc_demo 0 1 14 0 64 10\n"
    "\tclient:./hdc_demo 0 0 14 0 64 10\n");
    printf( "fast channel, examples:\n"
    "\tserver:./hdc_demo 1 1 14 0 64 10\n"
    "\tclient:./hdc_demo 1 0 14 0 64 10\n");
    printf( "send file channel, examples:\n"
    "\t./hdc_demo 2 0 /home/examples.txt \n");
    printf("-------------------------------------------------------------------------------------------\n");
}

int normal_channel_client(int devid, int type, int size, int count)
{
    struct drvHdcMsg *pMsg = NULL;
    HDC_SESSION Session = NULL;
    HDC_CLIENT client = NULL;
    char *buffer = NULL;
    int recv_cnt;
    int ret;

    ret = drvHdcClientCreate(&client, MAX_SESSION_NUM, type, 0);
    if (ret) {
        HDC_ERROR("drvHdcClientCreate failed, ret = %d", ret);
        return -1;
    }

    buffer = (char *)malloc(size);
    if (!buffer) {
        HDC_ERROR("malloc failed, ret = %d", ret);
        goto OUT;
    }

    while (count--) {
        ret = drvHdcSessionConnect(0, devid, client, &Session);
        if (ret) {
            HDC_ERROR("drvHdcSessionConnect failed, ret : %d, count : %d", ret, count);
            break;
        }

        ret = drvHdcSetSessionReference(Session);
        if (ret) {
            HDC_ERROR("drvHdcSetSessionReference failed, ret : %d, count : %d", ret, count);
            break;
        }

        ret = drvHdcAllocMsg(Session, &pMsg, 1);
        if (ret) {
            HDC_ERROR("drvHdcAllocMsg failed, ret : %d, count : %d", ret, count);
            break;
        }

        ret = halHdcRecv(Session, pMsg, size, 0, &recv_cnt, 0);
        if (ret) {
            HDC_ERROR("halHdcRecv failed, ret : %d, count : %d", ret, count);
            break;
        }
```



```
        ret = drvHdcReuseMsg(pMsg);
        if (ret) {
            HDC_ERROR("drvHdcReuseMsg failed, ret : %d, count : %d", ret, count);
            break;
        }

        ret = drvHdcAddMsgBuffer(pMsg, buffer, size);
        if (ret) {
            HDC_ERROR("drvHdcAddMsgBuffer failed, ret : %d, count : %d", ret, count);
            break;
        }

        ret = halHdcSend(Session, pMsg, 0, 0);
        if (ret) {
            HDC_ERROR("halHdcSend failed, ret : %d, count : %d", ret, count);
            break;
        }

        ret = drvHdcFreeMsg(pMsg);
        if (ret) {
            HDC_ERROR("drvHdcFreeMsg failed, ret : %d, count : %d", ret, count);
            break;
        }
        pMsg = NULL;

        ret = drvHdcSessionClose(Session);
        if (ret) {
            HDC_ERROR("drvHdcSessionClose failed, ret : %d, count : %d", ret, count);
            break;
        }
        Session = NULL;
    }

OUT:
    if (pMsg != NULL) {
        drvHdcFreeMsg(pMsg);
    }

    if (Session != NULL) {
        drvHdcSessionClose(Session);
    }

    if (buffer != NULL) {
        free(buffer);
    }

    if (client != NULL) {
        drvHdcClientDestroy(client);
    }

    return ret;
}

int normal_channel_server(int devid, int type, int size, int count)
{
    struct drvHdcMsg *pMsg = NULL;
    HDC_SESSION Session = NULL;
    HDC_SERVER server = NULL;
    char *buffer = NULL;
    int recv_len;
    int ret;

    ret = drvHdcServerCreate(devid, type, &server);
    if (ret) {
        HDC_ERROR("drvHdcServerCreate failed, ret = %d", ret);
        return -1;
    }

    buffer = (char *)malloc(size);
```



```
    if (!buffer) {
        HDC_ERROR("malloc failed, ret = %d", ret);
        goto OUT;
    }

    while (count--) {
        ret = drvHdcSessionAccept(server, &Session);
        if (ret) {
            HDC_ERROR("drvHdcSessionAccept failed, ret : %d, count : %d", ret, count);
            break;
        }

        ret = drvHdcSetSessionReference(Session);
        if (ret) {
            HDC_ERROR("drvHdcSetSessionReference failed, ret : %d, count : %d", ret, count);
            break;
        }

        ret = drvHdcAllocMsg(Session, &pMsg, 1);
        if (ret) {
            HDC_ERROR("drvHdcAllocMsg failed, ret : %d, count : %d", ret, count);
            break;
        }

        ret = drvHdcAddMsgBuffer(pMsg, buffer, size);
        if (ret) {
            HDC_ERROR("drvHdcAddMsgBuffer failed, ret : %d, count : %d", ret, count);
            break;
        }

        ret = halHdcSend(Session, pMsg, 0, 0);
        if (ret) {
            HDC_ERROR("halHdcSend failed, ret : %d, count : %d", ret, count);
            break;
        }

        ret = drvHdcReuseMsg(pMsg);
        if (ret) {
            HDC_ERROR("drvHdcReuseMsg failed, ret : %d, count : %d", ret, count);
            break;
        }

        ret = halHdcRecv(Session, pMsg, size, 0, &recv_len, 0);
        if (ret) {
            if (ret != DRV_ERROR_SOCKET_CLOSE)
                HDC_ERROR("halHdcRecv failed, ret : %d, count : %d", ret, count);
            break;
        }

        ret = drvHdcFreeMsg(pMsg);
        if (ret) {
            HDC_ERROR("drvHdcFreeMsg failed, ret : %d, count : %d", ret, count);
            break;
        }
        pMsg = NULL;

        ret = drvHdcSessionClose(Session);
        if (ret) {
            HDC_ERROR("drvHdcSessionClose failed, ret : %d, count : %d", ret, count);
            break;
        }
        Session = NULL;
    }

OUT:
    if (pMsg != NULL) {
        drvHdcFreeMsg(pMsg);
    }
```



```
    if (Session != NULL) {
        drvHdcSessionClose(Session);
    }

    if (buffer != NULL) {
        free(buffer);
    }

    if (server != NULL) {
        drvHdcServerDestroy(server);
    }

    return ret;
}

int normal_channel_test(int argc, char *argv[], void (*usage)(void))
{
    int server_type;
    int frame_size;
    int device_id;
    int loop_cnt;
    int mode;

    if (argc < 7) {
        usage();
        return -1;
    }

    mode = atoi(argv[2]); /* server or client */
    device_id = atoi(argv[3]);
    server_type = atoi(argv[4]);
    frame_size = atoi(argv[5]) * HDC_CHANGE_FRAME_SIZE_TO_KB; /* 将用户入参 frame 大小转化为 KB */
    loop_cnt = atoi(argv[6]);

    if (mode) {
        return normal_channel_server(device_id, server_type, frame_size, loop_cnt);
    } else {
        return normal_channel_client(device_id, server_type, frame_size, loop_cnt);
    }
}

int fast_channel_client(int devid, int type, int size, int count)
{
    unsigned long hostsendDataBuf;
    unsigned long hostrecvDataBuf;
    unsigned long hostsendCtrlBuf;
    unsigned long hostrecvCtrlBuf;
    struct drvHdcFastSendMsg smsg;
    struct drvHdcFastRecvMsg rmsg;
    struct drvHdcMsg  *pMsg = NULL;
    HDC_SESSION session = NULL;
    HDC_CLIENT client = NULL;
    void *sendDataBuf = NULL;
    void *recvDataBuf = NULL;
    void *sendCtrlBuf = NULL;
    void *recvCtrlBuf = NULL;
    char *sendbuffer = NULL;
    char *pBuf = NULL;
    int recvBufCount;
    int datalen;
    int ctrllen;
    int bufLen;
    int ret;

    ret = drvHdcClientCreate(&client, MAX_SESSION_NUM, type, 0);
    if (ret) {
        HDC_ERROR("drvHdcClientCreate failed, ret = %d", ret);
        goto _uninit;
    }
```

```
ret = drvHdcSessionConnect(0, devid, client, &session);
if (ret) {
    HDC_ERROR("drvHdcSessionConnect failed, ret : %d\n", ret);
    goto _destroy;
}

ret = drvHdcSetSessionReference(session);
if (ret) {
    HDC_ERROR("drvHdcSetSessionReference failed, ret : %d\n", ret);
    goto _close;
}

ret = drvHdcAllocMsg(session, &pMsg, 1);
if (ret) {
    HDC_ERROR("drvHdcSessionAccept failed, ret : %d\n", ret);
    goto _close;
}

sendbuffer = malloc(HDC_SEND_RECV_DATA_LEN);
if (sendbuffer == NULL) {
    HDC_ERROR("malloc fail.\n");
    goto _free;
}

datalen = size;
ctrllen = HDC_CHANGE_FRAME_SIZE_TO_KB; /*  */

/* 快速通道申请 buffer准备 */
sendDataBuf = drvHdcMallocEx(HDC_MEM_TYPE_TX_DATA, NULL, 0, datalen, devid, 0);
recvDataBuf = drvHdcMallocEx(HDC_MEM_TYPE_RX_DATA, NULL, 0, datalen, devid, 0);
sendCtrlBuf = drvHdcMallocEx(HDC_MEM_TYPE_TX_CTRL, NULL, 0, ctrllen, devid, 0);
recvCtrlBuf = drvHdcMallocEx(HDC_MEM_TYPE_RX_CTRL, NULL, 0, ctrllen, devid, 0);
if (!sendDataBuf || !recvDataBuf || !sendCtrlBuf || !recvCtrlBuf) {
    HDC_ERROR("drvHdcMallocEx failed, %p,%p,%p,%p\n",
        sendDataBuf, recvDataBuf, sendCtrlBuf, recvCtrlBuf);
    goto _free;
}

HDC_INFO("native fast channel info : datalen %d, ctrllen %d,  %p,%p,%p,%p",
    datalen, ctrllen, sendDataBuf, recvDataBuf, sendCtrlBuf, recvCtrlBuf);

*(((unsigned long *)sendbuffer) + 0) = (unsigned long)sendDataBuf;
*(((unsigned long *)sendbuffer) + 1) = (unsigned long)recvDataBuf;
*(((unsigned long *)sendbuffer) + 2) = (unsigned long)sendCtrlBuf;
*(((unsigned long *)sendbuffer) + 3) = (unsigned long)recvCtrlBuf;
*(((unsigned long *)sendbuffer) + 4) = datalen;
*(((unsigned long *)sendbuffer) + 5) = ctrllen;

ret = drvHdcAddMsgBuffer(pMsg, sendbuffer, HDC_SEND_RECV_DATA_LEN);
if (ret) {
    HDC_ERROR("drvHdcAddMsgBuffer failed, ret : %d\n", ret);
    goto _free;
}

/* 发送本端 快速通道的 buffer信息给到对端 */
ret = halHdcSend(session, pMsg, 0, 0);
if (ret) {
    HDC_ERROR("halHdcSend failed, ret : %d\n", ret);
    goto _free;
}

ret = drvHdcReuseMsg(pMsg);
if (ret) {
    HDC_ERROR("drvHdcReuseMsg failed, ret : %d\n", ret);
    goto _free;
}

ret = halHdcRecv(session, pMsg, HDC_SEND_RECV_DATA_LEN, 0, &recvBufCount, 0);
```



```
    if (ret) {
        HDC_ERROR("halHdcRecv failed, ret : %d\n", ret);
        goto _free;
    }

    ret = drvHdcGetMsgBuffer(pMsg, 0, &pBuf, &bufLen);
    if (ret) {
        HDC_ERROR("drvHdcGetMsgBuffer failed, ret : %d\n", ret);
        goto _free;
    }

    /* 获取对端快速通道信息 */
    hostsendDataBuf = *(((unsigned long *)pBuf) + 0);
    hostrecvDataBuf = *(((unsigned long *)pBuf) + 1);
    hostsendCtrlBuf = *(((unsigned long *)pBuf) + 2);
    hostrecvCtrlBuf = *(((unsigned long *)pBuf) + 3);
    datalen = (int)(*(((unsigned long *)pBuf) + 4));
    ctrllen = (int)(*(((unsigned long *)pBuf) + 5));

    HDC_INFO("remote fast channel info : datalen %d, ctrllen %d,  %lx,%lx,%lx,%lx",
        datalen, ctrllen, hostsendDataBuf, hostrecvDataBuf, hostsendCtrlBuf, hostrecvCtrlBuf);

    drvHdcFreeMsg(pMsg);
    pMsg = NULL;
    free(sendbuffer);
    sendbuffer = NULL;

    while (count--) {
        smsg.srcDataAddr = (unsigned long long)sendDataBuf;
        smsg.dstDataAddr = hostrecvDataBuf; /* 对端的接收buffer */
        smsg.srcCtrlAddr = (unsigned long long)sendCtrlBuf;
        smsg.dstCtrlAddr = hostrecvCtrlBuf; /* 对端的接收buffer */

        smsg.dataLen = datalen;
        smsg.ctrlLen = ctrllen;

        /* 用户可根据实际数据发送情况选择重发或者丢弃；参数使用参考具体接口描述 */
        ret = halHdcFastSend(session, smsg, HDC_WAIT_TIMEOUT, HDC_SEND_RECV_WAIT_TIME);
        if (ret) {
            HDC_ERROR("halHdcFastSend failed, ret : %d\n", ret);
            break;
        }

        ret = halHdcFastRecv(session, &rmsg, 0, 0);
        if (ret) {
            HDC_ERROR("halHdcFastRecv failed, ret : %d\n", ret);
            break;
        }
    }

    HDC_INFO("fast channel time delay client finish");

_free:
    if (sendbuffer) {
        free(sendbuffer);
    }

    if (sendDataBuf) {
        drvHdcFreeEx(HDC_MEM_TYPE_TX_DATA, sendDataBuf);
    }

    if (recvDataBuf) {
        drvHdcFreeEx(HDC_MEM_TYPE_RX_DATA, recvDataBuf);
    }

    if (sendCtrlBuf) {
        drvHdcFreeEx(HDC_MEM_TYPE_TX_CTRL, sendCtrlBuf);
    }
```



```
    if (recvCtrlBuf) {
        drvHdcFreeEx(HDC_MEM_TYPE_RX_CTRL, recvCtrlBuf);
    }

    if (pMsg) {
        drvHdcFreeMsg(pMsg);
    }

    sendbuffer = NULL;
    sendDataBuf = NULL;
    recvDataBuf = NULL;
    sendCtrlBuf = NULL;
    recvCtrlBuf = NULL;
    pMsg = NULL;
_close:
    drvHdcSessionClose(session);
_destroy:
    drvHdcClientDestroy(client);
_uninit:
    return ret;
}

int fast_channel_server(int devid, int type, int size, int count)
{
    unsigned long hostsendDataBuf;
    unsigned long hostrecvDataBuf;
    unsigned long hostsendCtrlBuf;
    unsigned long hostrecvCtrlBuf;
    struct drvHdcFastSendMsg smsg;
    struct drvHdcFastRecvMsg rmsg;
    struct drvHdcMsg *pMsg = NULL;
    HDC_SESSION session = NULL;
    HDC_SERVER server = NULL;
    void *sendDataBuf = NULL;
    void *recvDataBuf = NULL;
    void *sendCtrlBuf = NULL;
    void *recvCtrlBuf = NULL;
    char *sendbuffer = NULL;
    char *pBuf = NULL;
    int recvBufCount;
    int datalen;
    int ctrllen;
    int bufLen;
    int ret;

    ret = drvHdcServerCreate(devid, type, &server);
    if (ret) {
        HDC_ERROR("drvHdcServerCreate failed, ret = %d\n", ret);
        goto _uninit;
    }

    ret = drvHdcSessionAccept(server, &session);
    if (ret) {
        HDC_ERROR("drvHdcSessionAccept failed, ret : %d\n", ret);
        goto _destroy;
    }

    ret = drvHdcSetSessionReference(session);
    if (ret) {
        HDC_ERROR("drvHdcSetSessionReference failed, ret : %d\n", ret);
        goto _close;
    }

    ret = drvHdcAllocMsg(session, &pMsg, 1);
    if (ret) {
        HDC_ERROR("drvHdcSessionAccept failed, ret : %d\n", ret);
        goto _close;
    }
```



```
ret = halHdcRecv(session, pMsg, HDC_SEND_RECV_DATA_LEN, 0, &recvBufCount, 0);
if (ret) {
    HDC_ERROR("halHdcRecv failed, ret : %d\n", ret);
    goto _free;
}

ret = drvHdcGetMsgBuffer(pMsg, 0, &pBuf, &bufLen);
if (ret) {
    HDC_ERROR("drvHdcGetMsgBuffer failed, ret : %d\n", ret);
    goto _free;
}

/* 获取对端快速通道信息 */
hostsendDataBuf = *(((unsigned long *)pBuf) + 0);
hostrecvDataBuf = *(((unsigned long *)pBuf) + 1);
hostsendCtrlBuf = *(((unsigned long *)pBuf) + 2);
hostrecvCtrlBuf = *(((unsigned long *)pBuf) + 3);
datalen = (int)(*(((unsigned long *)pBuf) + 4));
ctrllen = (int)(*(((unsigned long *)pBuf) + 5));

HDC_INFO("remote fast channel info : datalen %d, ctrllen %d,  %lx,%lx,%lx,%lx",
    datalen, ctrllen, hostsendDataBuf, hostrecvDataBuf, hostsendCtrlBuf, hostrecvCtrlBuf);

/* 快速通道申请 buffer准备 */
sendDataBuf = drvHdcMallocEx(HDC_MEM_TYPE_TX_DATA, NULL, 0, datalen, devid, 0);
recvDataBuf = drvHdcMallocEx(HDC_MEM_TYPE_RX_DATA, NULL, 0, datalen, devid, 0);
sendCtrlBuf = drvHdcMallocEx(HDC_MEM_TYPE_TX_CTRL, NULL, 0, ctrllen, devid, 0);
recvCtrlBuf = drvHdcMallocEx(HDC_MEM_TYPE_RX_CTRL, NULL, 0, ctrllen, devid, 0);
if (!sendDataBuf || !recvDataBuf || !sendCtrlBuf || !recvCtrlBuf) {
    HDC_ERROR("drvHdcMallocEx failed, %p,%p,%p,%p\n",
        sendDataBuf, recvDataBuf, sendCtrlBuf, recvCtrlBuf);
    goto _free;
}

sendbuffer = malloc(HDC_SEND_RECV_DATA_LEN);
if (!sendbuffer) {
    HDC_ERROR("malloc fail.\n");
    goto _free;
}

*(((unsigned long *)sendbuffer) + 0) = (unsigned long)sendDataBuf;
*(((unsigned long *)sendbuffer) + 1) = (unsigned long)recvDataBuf;
*(((unsigned long *)sendbuffer) + 2) = (unsigned long)sendCtrlBuf;
*(((unsigned long *)sendbuffer) + 3) = (unsigned long)recvCtrlBuf;
*(((unsigned long *)sendbuffer) + 4) = datalen;
*(((unsigned long *)sendbuffer) + 5) = ctrllen;

HDC_INFO("native fast channel info : datalen %d, ctrllen %d,  %p,%p,%p,%p",
    datalen, ctrllen, sendDataBuf, recvDataBuf, sendCtrlBuf, recvCtrlBuf);

ret = drvHdcReuseMsg(pMsg);
if (ret) {
    HDC_ERROR("drvHdcReuseMsg failed, ret : %d\n", ret);
    goto _free;
}

ret = drvHdcAddMsgBuffer(pMsg, sendbuffer, HDC_SEND_RECV_DATA_LEN);
if (ret) {
    HDC_ERROR("drvHdcAddMsgBuffer failed, ret : %d\n", ret);
    goto _free;
}

/* 发送本端的  快速通道的 buffer信息给到对端 */
ret = halHdcSend(session, pMsg, 0, 0);
if (ret) {
    HDC_ERROR("halHdcSend failed, ret : %d\n", ret);
    goto _free;
}
```



```
    drvHdcFreeMsg(pMsg);
    pMsg = NULL;
    free(sendbuffer);
    sendbuffer = NULL;

    while (count--) {
        ret = halHdcFastRecv(session, &rmsg, 0, 0);
        if (ret) {
            HDC_ERROR("halHdcFastRecv failed, ret : %d\n", ret);
            break;
        }

        smsg.srcDataAddr = (unsigned long long)sendDataBuf;
        smsg.dstDataAddr = hostrecvDataBuf; /* 对端的接收buffer */
        smsg.srcCtrlAddr = (unsigned long long)sendCtrlBuf;
        smsg.dstCtrlAddr = hostrecvCtrlBuf; /* 对端的接收buffer */
        smsg.dataLen = datalen;
        smsg.ctrlLen = ctrllen;

        /* 用户可根据实际数据发送情况选择重发或者丢弃；参数使用参考具体接口描述 */
        ret = halHdcFastSend(session, smsg, HDC_WAIT_TIMEOUT, HDC_SEND_RECV_WAIT_TIME);
        if (ret) {
            HDC_ERROR("halHdcFastSend failed, ret : %d\n", ret);
            break;
        }
    }

    HDC_INFO("fast channel time delay server finish");

_free:
    if (sendbuffer) {
        free(sendbuffer);
    }

    if (sendDataBuf) {
        drvHdcFreeEx(HDC_MEM_TYPE_TX_DATA, sendDataBuf);
    }

    if (recvDataBuf) {
        drvHdcFreeEx(HDC_MEM_TYPE_RX_DATA, recvDataBuf);
    }

    if (sendCtrlBuf) {
        drvHdcFreeEx(HDC_MEM_TYPE_TX_CTRL, sendCtrlBuf);
    }

    if (recvCtrlBuf) {
        drvHdcFreeEx(HDC_MEM_TYPE_RX_CTRL, recvCtrlBuf);
    }

    if (pMsg) {
        drvHdcFreeMsg(pMsg);
    }

    sendbuffer = NULL;
    sendDataBuf = NULL;
    recvDataBuf = NULL;
    sendCtrlBuf = NULL;
    recvCtrlBuf = NULL;
    pMsg = NULL;
_close:
    drvHdcSessionClose(session);
_destroy:
    drvHdcServerDestroy(server);
_uninit:
    return ret;
}

int fast_channel_test(int argc, char *argv[], void (*usage)(void))
```



```
{
    int server_type;
    int frame_size;
    int device_id;
    int loop_cnt;
    int mode;

    if (argc < 7) {
        usage();
        return -1;
    }

    mode = atoi(argv[2]); /* server or client */
    device_id = atoi(argv[3]);
    server_type = atoi(argv[4]);
    frame_size = atoi(argv[5]) * HDC_CHANGE_FRAME_SIZE_TO_KB; /* 将用户入参 frame 大小转化为 KB */
    loop_cnt = atoi(argv[6]);

    if (mode) {
        return fast_channel_server(device_id, server_type, frame_size, loop_cnt);
    } else {
        return fast_channel_client(device_id, server_type, frame_size, loop_cnt);
    }
}

int hdc_send_file_to_device(int argc, char *argv[], void (*usage)(void))
{
    char dst_path[HDC_DST_PATH_LEN] = {0};
    char *file = NULL;
    int devid;
    int ret;

    if (argc < 4) {
        usage();
        return -1;
    }

    devid = atoi(argv[2]);
    file = argv[3];

    ret = drvHdcGetTrustedBasePath(0, devid, dst_path, HDC_DST_PATH_LEN);
    if (ret) {
        HDC_ERROR("drvHdcGetTrustedBasePath failed, ret : %d\n", ret);
        return -1;
    }

    ret = drvHdcSendFile(0, devid, file, dst_path, NULL);
    if (ret) {
        HDC_ERROR("drvHdcSendFile failed, ret : %d\n", ret);
        return -1;
    }

    HDC_INFO("send file success, dst path : %s", dst_path);

    return 0;
}

int main(int argc, char *argv[])
{
    int op_type;
    int ret = -1;

    if (argc < 2) {
        usage();
        return -1;
    }

    op_type = atoi(argv[1]);
    switch (op_type) {
```



```
        case 0:
            ret = normal_channel_test(argc, argv, usage);
            break;
        case 1:
            ret = fast_channel_test(argc, argv, usage);
            break;
        case 2:
            ret = hdc_send_file_to_device(argc, argv, usage);
            break;
        default:
            usage();
            break;
    }

    HDC_INFO("Test finish. ret = %d", ret);
    return ret;
}
```

说明

参见**10.2 样例编译**编译成可执行文件后（例如可执行文件的名字为hdc_demo），你可以通过./hdc_demo -h命令查看输入参数的详细含义，其中：

- “server/client（1/0）”：当前执行此程序的机器为服务端的话，则配置为“1”，客户端，则为“0”。
- “server_type”的详细含义请参见**9.7.10 drvHdcServiceType**。
- “size”：指传送的数据的大小，单位KB。
- “loop_times”：指收发超时次数。

10.2 样例编译

Host 侧应用编译

- 编译器：gcc。
- 依赖库文件：Host侧Driver包（*.run）安装路径下的“driver/lib64/driver/libascend_hal.so”，例如“/usr/local/Ascend/driver/lib64/driver/libascend_hal.so”。
- 头文件：“ascend_hal.h”，存储在{soc version}-driver-{software version}-minios.aarch64-src.tar.gz源码包中的“driver/source/inc/driver/”目录下，例如源码包加压缩到“/usr/local/software”目录下，则头文件存储路径为“/usr/local/software/driver/source/inc/driver”
- 编译命令：例如源文件为hdc_demo.c，编译命令示例如下：

```
gcc -lpthread -o hdc_demo hdc_demo.c -lascend_hal -L/usr/local/Ascend/driver/lib64/driver/ -I/usr/local/software/driver/source/inc/driver/
```

Device 侧应用编译

在Host侧使用HCC编译器进行交叉编译。

- HCC编译器存储路径：Toolkit安装路径下的“toolkit/toolchain/hcc/bin/aarch64-target-linux-gnu-gcc”。
- 依赖的库文件：使用**解压缩文件系统**中的库文件：“lib64/libascend_hal.so”。
- 头文件：“ascend_hal.h”，存储在{soc version}-driver-{software version}-minios.aarch64-src.tar.gz源码包中的“driver/source/inc/driver/”目录下，例如源码包加压缩到“/usr/local/software”目录下，则头文件存储路径为“/usr/local/software/driver/source/inc/driver”

- 编译命令：例如源文件为hdc_demo.c，编译命令示例如下：

```
aarch64-target-linux-gnu-gcc -lpthread -o hdc_demo hdc_demo.c -lascend_hal -lc_sec -lslog -lmmpa -ldevmmap -lstdc++ -L/文件系统解压路径/lib64 -L/文件系统解压路径/usr/lib64 -I/usr/local/software/driver/source/inc/driver/
```


11 安全加固

11.1 加固须知

本文中列出的安全加固措施为基本的加固建议项。用户应根据自身业务，重新审视整个系统的网络安全加固措施。用户应按照所在组织的安全策略进行相关配置，包括并不局限于软件版本、口令复杂度要求、安全配置（协议、加密套件、密钥长度等），权限配置、防火墙设置等。必要时可参考业界优秀加固方案和安全专家的建议。

11.2 Host 侧加固

11.2.1 禁止使用 root 帐户远程登录系统

root是Linux系统中的超级特权用户，具有所有Linux系统资源的访问权限。如果允许直接使用root账号登录Linux系统对系统进行操作，会带来很多潜在的安全风险。通常情况下，建议将在“/etc/ssh/sshd_config”文件中将“PermitRootLogin”参数设置为“no”，设置后root用户无法通过SSH登录到系统，增加了系统的安全性。如果需要使用root权限进行管理操作，可以通过其他普通用户登录系统后，再使用su或sudo命令切换到root用户进行操作。这样可以避免直接使用root用户登录系统，从而减少系统被攻击的风险。

11.2.2 账户 UID 安全加固

UID为0的账户具有系统最高权限，应保证只有一个，其中只有root是Linux系统中的超级特权用户，确保当前系统中只有root帐户uid=0，同时，各系统账号的UID必须是不同的，可以执行**id**查询用户UID。

11.2.3 内存地址随机化机制安全加固

ASLR（address space layout randomization）开启后能增强漏洞攻击防护能力，建议用户将/proc/sys/kernel/randomize_va_space里面的值设置为2，开启该功能。

11.2.4 禁止使用 SetUID 或 SetGID 的 shell 脚本

特殊权限的脚本有可能被恶意使用，给系统带来巨大的威胁，如非必要，建议禁止使用SUID、SGID脚本。

可以执行如下命令查找系统中SUID/SGID文件，查看是否必要存在。若无必要，去掉“s”位以取消文件的SetUID或SetGID权限，或删除文件。

```
find / -perm -2000 -exec ls -l {} \; -exec md5sum {} \;
find / -perm -4000 -exec ls -l {} \; -exec md5sum {} \;
```

11.2.5 无属主文件安全加固

用户可以执行**find / -nouser -o -nogroup**命令，查找系统中的无属主文件。根据文件的uid和gid创建相应的用户和用户组，或者修改已有用户的uid、用户组的gid来适配，赋予文件属主，避免无属主文件给系统带来安全隐患。

11.2.6 设置 umask

建议用户将主机（包括宿主机）和容器中的umask设置为027及其以上，提高安全性。

以设置umask为077为例，具体操作如下所示。

步骤1 以root用户登录服务器，编辑“/etc/profile”文件。

```
vim /etc/profile
```

步骤2 在“/etc/profile”文件末尾加上**umask 077**，保存并退出。

步骤3 执行如下命令使配置生效。

```
source /etc/profile
```

----结束

11.3 Device 侧加固

11.3.1 关闭 Device SSH 服务

Device的SSH服务默认处于关闭状态，处于关闭状态可提升系统安全性。若已开启SSH服务，可参考以下步骤关闭。

步骤1 创建 close_device_ssh.c，内容如下：

```
#include <stdlib.h>
#include <stdio.h>
#include "dsmi_common_interface.h" // DSMI相关接口所在头文件，存储路径为Host侧的/usr/local/Ascend/driver/include
int main()
{
    int ret;
    int dev_list[64] = {0};
    int dev_cnt = 0;
    const char config_name[20] = "ssh_status";
    unsigned int buf_size = 1;
    unsigned char buf = 0; // 0: disable 1:enable
    ret = dsmi_get_device_count(&dev_cnt);
    if (ret != 0) {
        printf("[%s] get dev_cnt test_fail value = %d \n", __func__, ret);
        return -1;
    }
```

```
    if (dev_cnt <= 0) {
        printf("[%s] get dev_cnt test_fail value = -1 , dev_cnt:%d \n", __func__, dev_cnt);
        return -1;
    }
    printf("[%s] dev_cnt:%d \n", __func__, dev_cnt);
    ret = dsmi_list_device(dev_list, dev_cnt);
    if (ret != 0) {
        printf("[%s] list device test_fail value = %d \n", __func__, ret);
        return -1;
    }
    for (int i = 0; i < dev_cnt; i++) {
        ret = dsmi_set_user_config(dev_list[i], config_name, buf_size, &buf);
        if (ret != 0) {
            printf("[%s, %d] dev_id:%d test_fail, value = -1 ret:%d \n", __func__, __LINE__, dev_list[i], ret);
            return -1;
        }
        printf("[%s, %d] dev_id:%d set %s:0x%x, buf_size:%d\n", __func__, __LINE__, dev_list[i],
config_name,
        buf, buf_size);
    }
    return 0;
}
```

步骤2 执行如下命令，对“close_device_ssh.c”文件进行编译。

```
gcc close_device_ssh.c /usr/local/Ascend/driver/lib64/driver/libdrvdsmi_host.so -L. -I/usr/local/Ascend/driver/
include -std=c99 -o close_device_ssh
```

/usr/local/Ascend表示Driver的默认安装路径，请根据实际情况替换。

步骤3 执行可执行文件“close_device_ssh”，关闭Device的SSH服务。

```
./close_device_ssh
```

步骤4 重启Host。

通过DSMI接口开启Device的SSH服务后，需要重启才能生效，请在Host侧执行如下命令进行重启操作。

```
reboot
```

----结束

11.3.2 开启 Host to Device 文件传输白名单校验功能

Device默认开启了H2D（Host to Device）文件传输时的白名单校验功能，对传输文件进行大小、存放目录及权限的校验，仅在白名单列表中的文件才允许使用HDC接口从Host传输到Device，可参考**7.3 配置H2D文件传输白名单**进行配置。

12 常用操作

12.1 修改 ssh 超时连接时间

Device侧配置的客户端与服务端无响应超时时间默认为300s，若用户想修改默认连接超时时间，可在制作Device侧文件系统时修改文件系统中的/etc/ssh目录下的sshd_config文件与/etc/profile文件。

- 不超时设置方法。
  a. 修改文件系统中的/etc/ssh目录下的sshd_config文件。
     vi *<u>/usr/local/filesys_modify/tempdir</u>*/etc/ssh/sshd_config
     - 取消“ClientAliveInterval”配置项的注释，并设置为0。
     - 取消“ClientAliveCountMax”配置项的注释，并设置为0。

     执行**:wq**保存退出。
  b. 修改文件系统中的/etc/profile文件。
     vi *<u>/usr/local/filesys_modify/tempdir</u>*/etc/profile
     修改“TMOUT”为0，如下所示：
     ```
     export TMOUT=0
     ```
     执行**:wq**保存退出。
- 固定超时时间设置方法。
  a. 修改文件系统中的/etc/ssh目录下的sshd_config文件。
     vi *<u>/usr/local/filesys_modify/tempdir</u>*/etc/ssh/sshd_config
     - 取消“ClientAliveInterval”配置项的注释，并设置为期望超时的固定时间，单位为秒。
     - 取消“ClientAliveCountMax”配置项的注释，并设置为0。

     执行**:wq**保存退出。

b. 修改文件系统中的/etc/profile文件。

vi */usr/local/filesys_modify/tempdir*/etc/profile

修改TMOUT为期望超时的固定时间（同etc/ssh/sshd_config文件中的ClientAliveInterval），单位为秒，如下所示：

```
export TMOUT=500
```

执行**:wq**保存退出。

说明

若制作文件系统时未按照如上配置进行设置，后续可直接在Device侧进行上述配置修改，修改后重启sshd进程，然后重新连接Device即可。

使用root用户重启sshd进程操作如下：

```
ps -ef | grep sshd
kill -9 <sshd_pid>
/usr/sbin/sshd &
```

说明：Device侧无文件编辑命令，可将Device上相关文件拷贝到其他环境进行编译，然后替换Device侧对应文件。

12.2 设置用户有效期

为保证用户的安全性，应设置用户的有效期，使用系统命令chage来设置用户的有效期。

命令为：

```
chage [-m mindays] [-M maxdays] [-d lastday] [-I inactive] [-E expiredate] [-W warndays] user
```

相关参数请参见表12-1。

表 12-1 设置用户有效期

参数	参数说明
-m	两次修改密码的最小间隔天数。设置为“0”表示任何时候都可以更改口令。
-M	口令保持有效的最大天数，为相对于上一次修改密码或者创建账号的天数。 设置为“-1”表示可删除这项口令的检测。设置为“99999”表示无限期。
-d	上一次更改的日期。
-I	停滞时期。过期指定天数后，设定密码为失效状态。
-E	用户到期的日期。超过该日期，此用户将不可用。 日期格式为：YYYY-MM-DD。
-W	用户口令到期前，提前收到警告信息的天数。
-l	列出当前的设置。由非特权用户来确定口令或帐户何时过期。

说明

- 表12-1只列举出常用的参数，用户可通过**chage --help**命令查询详细的参数说明。
- User必须填写，填写时请替换为具体用户，默认为root用户。
- Device重启后，Device的配置文件会恢复默认配置（默认账号的超期时间恢复为90天），若用户需要自定义过期时间，建议用户在5 定制文件系统时在启动脚本（例如：“/etc/rc.d/init.d/rcS”）中添加对应的chage命令重新配置过期时间。

举例说明：

- 修改用户HwHiAiUser的有效期为2020年12月01日，用户HwHiAiUser的口令在2020年12月1日过期
  ```
  chage -E 2020-12-01 HwHiAiUser
  ```
- 修改用户HwHiAiUser的有效期为90天，创建HwHiAiUser用户的天数或相对上一次修改密码的天数
  ```
  chage -M 90 HwHiAiUser
  ```

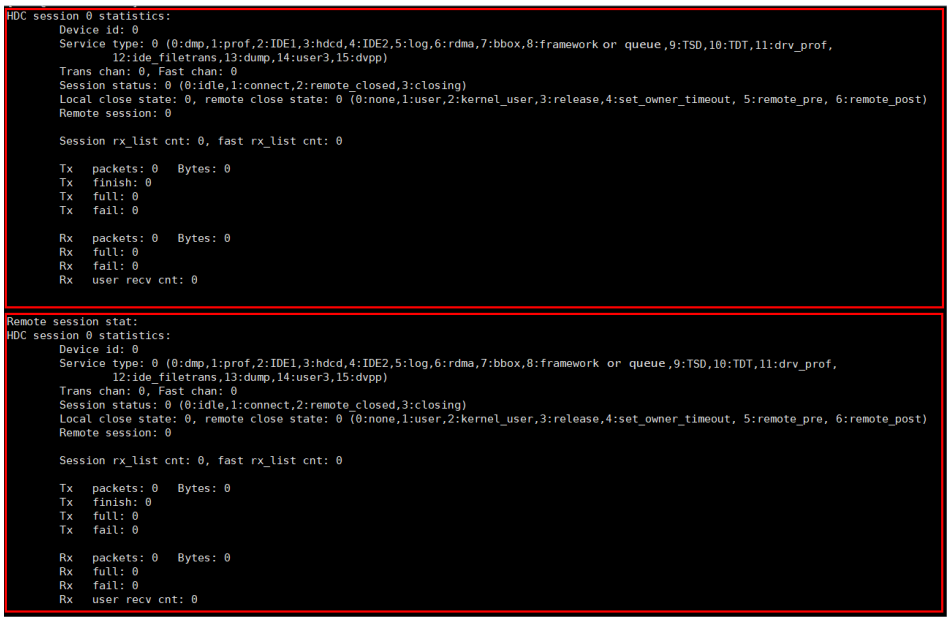
12.3 HDC 常用操作

以root用户登录Host或者Device，并进入目录“/sys/devices/virtual/hisi_hdc/hisi_hdc/hdc”，此目录下文件如下所示：

```
[root@(none) hdc]# ls
chan  chan_stat  dev  dev_stat  server  server_stat  session  session_stat
```

以下操作都在此目录下执行。

1. 通过“dev”查看收发包情况，以及内存池使用情况等。
   a. 输入要查询的设置的Device ID。
      echo *{Device ID}* > dev
      例如：echo 0 > dev
   b. 查看指定Device的统计信息。
      cat dev_stat
      例如：
      ```
      Total active session number: 10  // 表示当前dev上总的使用中session数为10个
      active session list：4，7，8  // 表示当前dev上使用中的session为 4,7,8
      ```
2. 通过“session”查看session的使用情况，以及收发统计信息。
   a. 输入要查询的session通道号。
      echo *{session 通道号}* > session
      session通道号可以通过1中查出的“active session list”获取。
      例如：echo 0 > session
   b. 查看指定session的统计信息。
      cat session_stat
      回显如下所示：



其中上半部分红框中信息为本端的session统计信息；下半部分红框中信息为对端的session统计信息。

关键字段解释如下所示：

- Trans chan：普通通道对应的通道号。
- Fast chan：快速通道对应的通道号。
- Session status：表示当前session状态。

  0:idle

  1:connect

  2:remote_closed

  3:closing

  Remote session: *161*，表示对端相连的session号是*161*。
- Local close state: 本端session通道被关闭的方式； remote close state: 对端session通道被关闭的方式。
- Session rx_list cnt：普通通道当前时刻缓存中的队列数量。
- fast rx_list cnt：快速通道当前时刻缓存中的队列数量。
- Tx：本端session通道发送情况。
- Rx：本端session通道接收情况。
- tx mem pool：本端发送的内存使用情况；rx mem pool：本端接收的内存使用情况。
- Huge mem pool's total num：大块内存总个数（每块内存的大小为512K）； remain num：当前空闲的大块内存个数。
- Small mem pool's total num：小块内存总个数（每块内存的大小为一个page size）；remain num：当前空闲的小块内存个数。

3. 通过“chan”查看收发信息以及队列调度信息。
   a. 输入要查询的chan通道号。
      echo *{chan通道号}* > chan
      chan通道号可以通过2中查出的“Trans chan”或者“Fast chan”获取。
      例如：echo 2 > chan
   b. 查看指定通道的收发统计信息。
      cat chan_stat
      例如：“full”表示队列满的次数，“fail”表示收发失败的次数。
4. 通过“server”查看统计信息。
   a. 输入要查询的服务类型。
      echo *{服务类型}* > server
      例如：echo 0 > server
   b. 查看指定服务类型的统计信息。
      cat server_stat

13 附录

13.1 使用 DSMI 接口打开 SSH 服务

简介

若您在制作文件系统时未通过修改系统启动脚本的方式打开SSH服务，系统启动后，您可参考本章节使用DSMI接口进行打开。

须知

Device的SSH服务处于关闭状态可提升系统安全性。

如果需要打开SSH，请确保SSH登录密码的复杂度和机密性。如果SSH登录密码出现泄露，攻击者可侵入Device对系统进行篡改，并可能导致系统内部敏感信息泄漏、系统无法得到预期的运行结果或系统无法正常运行等安全事件。

操作步骤

用户可通过调用DSMI接口“dsmi_set_user_config”打开Device侧的SSH服务，下面给出调用此DSMI接口的完整示例。

步骤1 在Host侧以root用户新建调用DSMI接口的源码文件，例如命名为“dsmi_open_ssh.c”。

touch dsmi_open_ssh.c

“dsmi_open_ssh.c”文件中代码示例如下：

```
#include <stdlib.h>
#include <stdio.h>
#include "dsmi_common_interface.h"  // DSMI相关接口所在头文件，存储路径为Host侧的/usr/local/Ascend/
```

```
driver/include

int main()
{
    int ret;
    int dev_list[64] = {0};
    int dev_cnt = 0;
    const char config_name[20] = "ssh_status";
    unsigned int buf_size = 1;
    unsigned char buf = 1;  // 0: disable 1:enable

    ret = dsmi_get_device_count(&dev_cnt);
    if (ret != 0) {
        printf("[%s] get dev_cnt test_fail value = %d \n", __func__, ret);
        return -1;
    }
    if (dev_cnt <= 0) {
        printf("[%s] get dev_cnt test_fail value = -1 , dev_cnt:%d \n", __func__, dev_cnt);
        return -1;
    }
    printf("[%s] dev_cnt:%d \n", __func__, dev_cnt);

    ret = dsmi_list_device(dev_list, dev_cnt);
    if (ret != 0) {
        printf("[%s] list device test_fail value = %d \n", __func__, ret);
        return -1;
    }

    for (int i = 0; i < dev_cnt; i++) {
         ret = dsmi_set_user_config(dev_list[i], config_name, buf_size, &buf);
         if (ret != 0) {
             printf("[%s, %d] dev_id:%d test_fail, value = -1 ret:%d \n", __func__, __LINE__, dev_list[i], ret);
             return -1;
         }
        printf("[%s, %d] dev_id:%d set %s:0x%x, buf_size:%d\n", __func__, __LINE__, dev_list[i], config_name,
buf, buf_size);
    }

    return 0;
}
```

步骤2 执行如下命令，对“dsmi_open_ssh.c”文件进行编译。

```
gcc dsmi_open_ssh.c /usr/local/Ascend/driver/lib64/driver/libdrvdsmi_host.so -L. -I/usr/local/Ascend/driver/
include  -std=c99 -o open_ssh_tool
```

说明

/usr/local/Ascend表示Driver组件的默认安装路径，请根据实际情况替换。

编译完成后，会生成可执行文件“open_ssh_tool”。

步骤3 执行可执行文件“open_ssh_tool”，开启Device的SSH服务。

```
./open_ssh_tool
```

步骤4 重启Host。

通过DSMI接口开启Device的SSH服务后，需要重启才能生效，请在Host侧执行如下命令进行重启操作。

```
reboot
```

步骤5 SSH服务开启后，系统默认的SSH服务占用的内存大小限制为50MB，若用户想取消此限制，可进行如下操作：

1. 以HwHiAiUser用户登录Device，然后切换到root用户。
2. 执行如下命令取消对SSH服务占用内存大小的限制。


```
cat /sys/fs/cgroup/memory/sshdmemory/cgroup.procs |xargs -n1 >>/sys/fs/
cgroup/memory/cgroup.procs;rmdir /sys/fs/cgroup/memory/sshdmemory/
```

需要注意，每次Host重启后，都会恢复默认50MB的SSH服务内存限额。

----结束

13.2 HCC 编译器说明

Toolkit包中提供了HCC编译器，用于进行Device侧驱动及应用程序的编译。HCC编译器存储在Toolkit安装路径的“toolchain/hcc”目录中，包含的主要二进制工具说明如下表所示。

表 13-1 HCC 编译器工具说明

工具名称	存储相对路径	功能说明及使用场景	风险分析	保留原因
gcc	hcc/bin/aarch64-target-linux-gnu-gcc	编译C语言实现的源文件，生成可执行文件。 用户编写Device侧代码时会使用该工具编译代码生成可执行文件。	用户只能用来编译源码生成可执行的二进制文件，无法获取其他运行态信息，实际风险小。	属于编译器配套基础二进制工具集。
cpp	hcc/bin/aarch64-target-linux-gnu-cpp	编译C++语言实现的源文件，生成可执行文件。 用户编写Device侧代码时会使用该工具编译代码生成可执行文件。	用户只能用来编译源码生成可执行的二进制文件，无法获取其他运行态信息，实际风险小。	
ld	hcc/bin/aarch64-target-linux-gnu-ld	创建动态链接的可执行文件时，将所有符号添加到动态符号表中。 用户编写Device侧代码时在编译源码的过程中会使用该工具链接中间文件生成完整的可执行文件。	用户使用链接器在编译过程中链接中间文件，无法获取其他运行态信息，实际风险小。	
readelf	hcc/bin/aarch64-target-linux-gnu-readelf	显示二进制文件的ELF标头包含的信息。 调试问题时可通过该工具查看算子库二进制文件的ELF头信息及各个段的信息。	用户仅仅可以查看二进制文件的段信息，无法获取其他运行态信息，实际风险小。	

工具名称	存储相对路径	功能说明及使用场景	风险分析	保留原因
addr2line	hcc/bin/aarch64-target-linux-gnu-addr2line	用来将程序地址转换成其所对应的程序源文件、代码行以及函数。	对于存在调试信息的可执行程序，可使用此工具将地址对应成源码，无法获取其他运行态信息，实际风险小。	
g++	hcc/bin/aarch64-target-linux-gnu-g++	编译C++语言实现的源文件，生成可执行文件。 用户编写Device侧代码时会使用该工具编译代码生成可执行文件。	用户只能用来编译源码生成可执行的二进制文件，无法获取其他运行态信息，实际风险小。	
gcov-dump	hcc/bin/aarch64-target-linux-gnu-gcov-dump	线下gcda和gcno文件的dump工具 ，用于打印出Device侧coverage文件内容。	用户只能用来查看gcda及gcno文件内容，无法获取其他运行态信息，实际风险小。	
ranlib	hcc/bin/aarch64-target-linux-gnu-ranlib	用于生成库文件中的.o文件索引。 用户将object文件添加到库文件中后，可使用此工具更新符号表索引。	用户将object文件添加到库文件中后，仅可使用此工具更新符号表索引，无法获取其他运行态信息，实际风险小。	
ar	hcc/bin/aarch64-target-linux-gnu-ar	创建、修改、解压静态库文件。 将object文件打包成库文件。	用户只能用来将目标文件打包成库文件，无法获取其他运行态信息，实际风险小。	
gcov-tool	hcc/bin/aarch64-target-linux-gnu-gcov-tool	线下gcda文件处理工具。 用户可使用此工具打印出Device侧gcda文件中计数器。	用户只能用来查看gcda文件中计数器，无法获取其他运行态信息，实际风险小。	
as	hcc/bin/aarch64-target-linux-gnu-as	将汇编文件转换成二进制文件。 用户可使用此工具将汇编文件转换成机器码二进制文件。	用户只能用来编译源码生成可执行的二进制文件，无法获取其他运行态信息，实际风险小。	

工具名称	存储相对路径	功能说明及使用场景	风险分析	保留原因
gcc-7.3.0	hcc/bin/aarch64-target-linux-gnu-gcc-7.3.0	编译C语言实现的源文件，生成可执行文件。 用户编写Device侧代码时会使用该工具编译代码生成可执行文件。	用户只能用来编译源码生成可执行的二进制文件，无法获取其他运行态信息，实际风险小。	
gprof	hcc/bin/aarch64-target-linux-gnu-gprof	程序性能分析工具，计算程序运行中各个函数消耗时间。 用户查看Device侧可执行文件的性能工具。	用户可以通过该工具修改ELF文件头，用于调试，风险可控。	
strings	hcc/bin/aarch64-target-linux-gnu-strings	从目标文件中列出可打印的字符串。 用户可以将目标文件中字符串打印出来。	用户只能将目标文件中字符串打印出来，无法获得其他运行态信息，实际风险小。	
c++	hcc/bin/aarch64-target-linux-gnu-c++	编译C++语言实现的源文件，生成可执行文件。用户编写Device侧代码时会使用该工具编译代码生成可执行文件。	用户只能用来编译源码生成可执行的二进制文件，无法获取其他运行态信息，实际风险小。	
gcc-ar	hcc/bin/aarch64-target-linux-gnu-gcc-ar	创建，修改，解压一个静态库文件。 用于将Device侧object文件打包成库文件。	用户只能用来将目标文件打包成库文件，无法获取其他运行态信息，实际风险小。	
strip	hcc/bin/aarch64-target-linux-gnu-strip	丢弃程序文件中的符号信息。 用户可以使用此工具将目标文件中的符号表或指定段删除。	用户只能将目标文件中的符号表或者指定段删除， 无法获得其他运行态信息，实际风险小。	
c++filt	hcc/bin/aarch64-target-linux-gnu-c++filt	转换C++符号为可识别的C符号。 将符号表中C++的符号转换成可识别的C符号。	用户只能用来将C++符号转换成可识别的C符号，实际风险小。	
gcc-nm	hcc/bin/aarch64-target-linux-gnu-gcc-nm	用于列出程序文件中的符号，符号是指函数或变量名等。 用户查看Device侧可执行文件的符号表。	用户查看可执行文件的符号表，无法获取其他运行态信息，实际风险小。	

工具名称	存储相对路径	功能说明及使用场景	风险分析	保留原因
ld.bfd	hcc/bin/aarch64-target-linux-gnu-ld.bfd	链接器，将多个对象和库文件组合成一个二进制文件，重新定位它们的数据并且捆绑符号索引。 用户编写Device侧代码时在编译源码的过程中会使用该工具链接中间文件生成完整的可执行文件。	用户使用链接器在编译过程中链接中间文件，无法获取其他运行态信息，实际风险小。	
gcc-ranlib	hcc/bin/aarch64-target-linux-gnu-gcc-ranlib	用于生成库文件中的.o文件索引。 用户将object文件添加到库文件中后，更新符号表索引。	用户将object文件添加到库文件中后，更新符号表索引，无法获取其他运行态信息，实际风险小。	
nm	hcc/bin/aarch64-target-linux-gnu-nm	用于列出程序文件中的符号，符号是指函数或变量名等。 用户可使用此工具查看Device侧可执行文件的符号表。	用户查看可执行文件的符号表，无法获取其他运行态信息，实际风险小。	
elfedit	hcc/bin/aarch64-target-linux-gnu-elfedit	用于更新ELF文件头及文件属性。 用户更新Device侧可执行文件的ELF文件头。	用户可以通过该工具修改ELF文件头，用于调试，风险可控。	
gcov	hcc/bin/aarch64-target-linux-gnu-gcov	用于打印文件覆盖率信息。 用户打印出Device侧插桩后可执行文件的覆盖率信息。	用户只能用来统计覆盖率信息，无法获取其他运行态信息，实际风险小。	
objcopy	hcc/bin/aarch64-target-linux-gnu-objcopy	用于对生成的程序文件进行一定的编辑。 用户编译、拷贝Device侧可执行文件，将目标文件从一种二进制格式复制或者翻译到另一种二进制格式。	用户编辑、拷贝可执行文件，无法获取其他运行态信息，实际风险小。	

工具名称	存储相对路径	功能说明及使用场景	风险分析	保留原因
run	/hcc/bin/aarch64-target-linux-gnu-run	模拟器，用于运行和调试程序。	用户仅可通过该工具获取调试信息，无法获取其他信息，实际风险小。	

13.3 CMS 签名

13.3.1 安装 cmssign 工具

检查 root 用户的 umask

如果root用户计划允许其他用户使用安装的Python等软件，请执行以下操作。否则，请确认umask符合用户所在组织的安全设置规范。

1. 以root用户登录待安装Linux环境。
2. 执行如下命令，检查root用户的umask值。

   umask
3. 如果umask不等于077，请执行如下操作配置，在该文件的最后一行添加umask 077后保存。
   a. 在任意目录下执行如下命令，打开**.bashrc**文件：

      vi ~/.bashrc

      在文件最后一行后面添加umask 077内容。
   b. 执行**:wq!**命令保存文件并退出。
   c. 执行**source ~/.bashrc**命令使其立即生效。

说明

建议在cmssign工具使用完毕后，将用户umask修改至符合安全规范。

安装步骤

步骤1 以安装用户登录需要运行cmssign工具的Linux服务器，并确保当前环境已安装Java。

步骤2 在用户的家目录下，执行如下命令建立并载入虚拟python环境（如果创建失败，请根据提示安装所需组件）。

python3 -m venv env

source ~/env/bin/activate

说明

当用户不需要使用cmssign工具时，可执行如下命令退出虚拟python环境：

deactivate

步骤3 执行如下命令升级pip版本。

python -m pip install --upgrade pip

步骤4 执行如下命令安装cmssign工具包。

python -m pip install cmssign==0.0.4

----结束

说明

- 安装完毕后，用户可选择执行如下命令，加固安装文件权限、确认可载入的python库路径是否符合预期、目录和文件的属主及权限是否符合预期等。

  chmod -R g-rwx,o-rwx ~/env

  python -c "import sys; print('\n'.join(sys.path))"

  find ~/env ! -user {用户名}

  ls -ld ~/env

  若执行**find**命令后返回的结果为空、执行**ls**命令显示该目录下其他用户不可读写，表示加固安装文件权限成功。
- 若已安装cmssign工具，可执行以下命令升级cmssign工具至最新版本。

  python3 -m pip install -U cmssign
- 数字签名文件使用的证书需使用安全的签名算法，例如使用无截断的sha2-256作为摘要和完整性验证算法、使用sha256_rsa作为签名摘要算法、使用安全的填充算法等。同时用户使用的算法、私钥长度、口令复杂度等需要符合所在组织的安全要求。

参数说明

使用cmssign工具时，用户根据实际需要选择对应参数。

使用**cmssign sign --help**命令查询的参数请参见**表13-2**所示。

表 13-2 cmssign sign 参数说明

选项	用途
--help | -h	查询帮助信息。
--signer	用于签名的证书文件，.der格式，需指定。
--key	用于签名的私钥文件，需指定。
--signerCA	用于签名的根证书文件，.der格式，需指定。
--signerCRL	用于签名的CRL文件，.der格式，需指定。
--rootCRL	用户根证书的CRL文件，.der格式，需指定。

选项	用途
--tssigner	用于时间戳签名的证书文件，.der格式，需指定。
--tskey	用于时间戳签名的私钥文件，需指定。
--tsCA	用于时间戳签名的根证书文件，.der格式，需指定。
--timestamp	用于设置签名文件的时间戳，可选。若未设置该参数，则默认使用系统时间戳。配置格式示例如下： '20220101'或'20220101123000'。
--in	需要进行签名的ini文件，需指定。
--out	用于保存cms签名的文件，若未配置，将默认为“*infile*.p7s”。

使用**cmssign combine --help**命令查询的参数请参见**表13-3**所示：

表 13-3 cmssign combine 参数说明

选项	用途
--help | -h	查询帮助信息。
--cmsfile	用于合成的cms签名文件，需指定。
--tsfile	用于合成的时间戳文件，需指定。
--out	用于保存合成后的cms签名的文件，需指定。

13.3.2 创建 CMS 签名

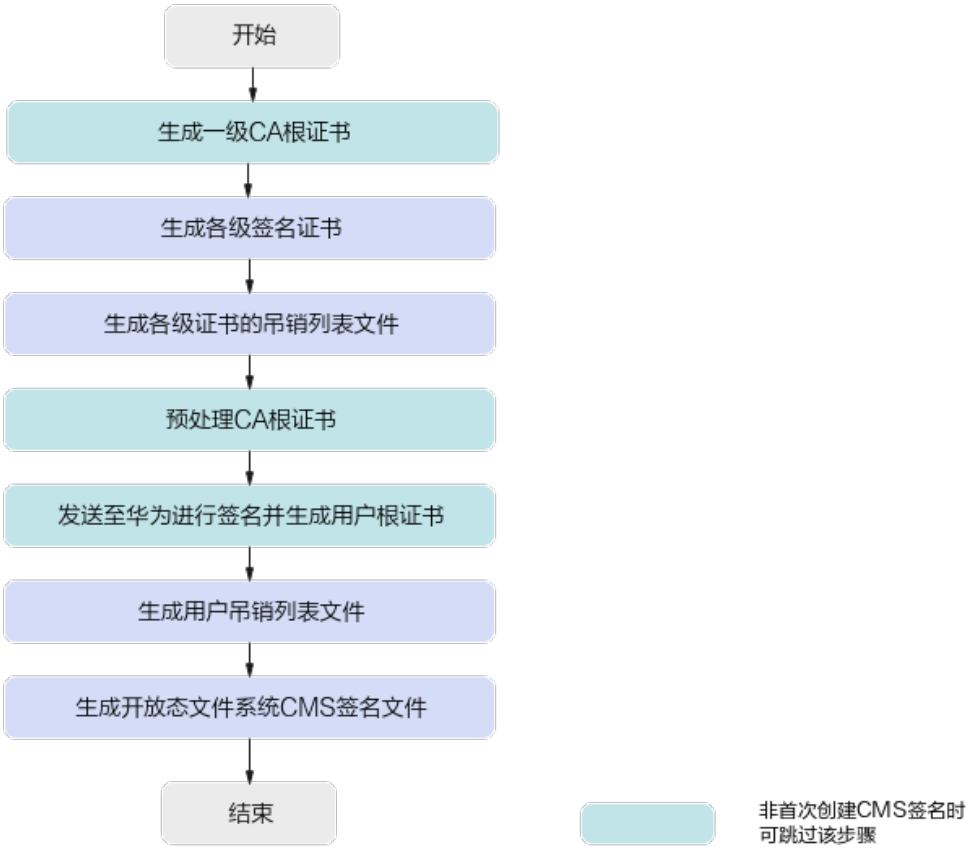
13.3.2.1 概述

使用Openssl命令行模拟华为公司签名平台的CMS签名，即执行**openssl**命令，创建根证书、二级CA签名证书、二级CA时间戳证书、三级签名证书和时间戳证书、证书吊销列表crl，生成用户根证书和吊销列表，并使用cmssign工具合成符合华为公司签名平台的签名，业务流程如**图13-1**所示。生产环境下，用户需妥善保管私钥。

注意

使用Openssl生成证书只能用于开发和测试阶段，正式商用发布需要客户构建PKI系统或CA系统（用于管理密钥和证书）。当前提供的签名方式仅做参考，客户可以根据自身情况使用PKI系统并结合CMS格式进行签名，同时客户需要自行管理产生的密钥和证书。

图 13-1 创建 CMS 签名的业务流程



1. 生成一级CA根证书，具体操作请参见**步骤3**。
2. 生成各级签名证书，具体操作请参见**步骤4**至**步骤7**。
3. 生成各级证书的吊销列表文件，具体操作请参见**步骤8**。
4. 预处理CA根证书生成摘要信息文件**usrcert.ini**，具体操作请参见**步骤9**。
5. 将摘要信息文件**usrcert.ini**发送至华为进行签名，并返回**usrcert.ini.cms**和**usrcert.ini.crl**文件，生成用户根证书**user.xer**，具体操作请参见**步骤10**。
6. 生成用户吊销列表文件**user.crl**文件，具体操作请参见**步骤11**。
7. 使用cmssign工具，生成开放态文件系统CMS签名文件，具体操作请参见**13.3.2.3 生成开放态文件系统CMS签名文件**。

13.3.2.2 创建并激活根证书

本章节将指导用户编写CMS签名依赖的证书文件，并激活用户根证书，生成用户根证书文件**user.xer**和证书吊销列表文件**user.crl**。建议将本章节生成的各类证书文件保存

至固定的文件夹，可以在CANN 6.0.RC1及更新版本中各类证书有效期内实现安全启动功能时一直使用。

说明

- 创建根证书前，用户需确保本地环境已安装符合所在组织安全要求的Openssl（Openssl版本需在1.1及以上，若版本不满足要求，请升级Openssl版本，建议安装最新且无漏洞版本）。
- 以下示例步骤中生成的各级CA证书、签名证书、吊销列表包含有效期信息，用户可以根据产品生命周期调整对应文件的有效期，若**步骤4**至**步骤8**生成的各类文件任一超出有效期，则需要重新生成各类证书文件和开放态文件系统CMS签名文件，并重新部署文件系统。

步骤1 登录Linux服务器并执行如下命令切换至root用户。

su - root

步骤2 进入Linux服务器任意目录，比如/root/sign，在该目录下创建并编写CMS签名依赖的各类证书。

步骤3 生成一级CA根证书（非首次创建CMS签名时，可跳过该步骤）。

1. 执行**vi x509_rootcaG2.cnf**命令生成并编写x509_rootcaG2.cnf文件，将如下内容加入到该文件中。

```
[ req ]
default_bits = 4096
distinguished_name = req_distinguished_name
prompt = no
string_mask = utf8only
x509_extensions = extensions

[ req_distinguished_name ]
C = CN
O = Test
CN = RootCA G2
#emailAddress = cmscbb@huawei.com

[ extensions ]
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid:always, issuer
basicConstraints = CA:TRUE
keyUsage = cRLSign, keyCertSign
```

   执行**:wq!**保存该文件。

2. 执行如下命令创建一级CA根证书。其中**-days 36500**表示一级CA根证书有效期为100年。

   openssl req -new -sigopt rsa_padding_mode:pss -sigopt rsa_pss_saltlen:-1 -nodes -utf8 -sha256 -days 36500 -batch -x509 -config x509_rootcaG2.cnf -outform PEM -out rootcaG2.pem -keyout rootca_priG2_pri.pem

   openssl x509 -in rootcaG2.pem -inform pem -outform der -out rootcaG2.der

步骤4 生成二级CA签名证书。

1. 执行**vi x509_signca.cnf**命令生成并编写x509_signca.cnf文件，将如下内容加入到该文件中。

```
[ req ]
default_bits = 4096
distinguished_name = req_distinguished_name
prompt = no
string_mask = utf8only
x509_extensions = extensions

[ req_distinguished_name ]
C = CN
```



```
O = Test
CN = Signing Certificate CA G2
#emailAddress = cmscbb@huawei.com

[ extensions ]
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid, issuer
basicConstraints = CA:TRUE
keyUsage = cRLSign, keyCertSign
extendedKeyUsage = codeSigning
```

执行**:wq!**保存该文件。

2. 执行如下命令创建二级CA签名证书。其中**-days 3650**表示二级CA签名证书有效期为10年。

   openssl genrsa -out signca_pri.pem 4096

   openssl req -new -config x509_signca.cnf -out signca.csr -key signca_pri.pem

   openssl x509 -req -in signca.csr -sigopt rsa_padding_mode:pss -sigopt rsa_pss_saltlen:-1 -CA rootcaG2.pem -CAkey rootca_priG2_pri.pem -CAcreateserial -out signca.pem -days 3650 -extensions extensions -extfile x509_signca.cnf

   openssl x509 -in signca.pem -inform pem -outform der -out signca.der

步骤5 生成二级CA时间戳证书。

1. 执行**vi x509_tsca.cnf**命令生成并编写x509_tsca.cnf文件，将如下内容加入到该文件中。

```
[ req ]
default_bits = 4096
distinguished_name = req_distinguished_name
prompt = no
string_mask = utf8only
x509_extensions = extensions

[ req_distinguished_name ]
C = CN
O = Test
CN = Timestamp Certificate CA G2
#emailAddress = cmscbb@huawei.com

[ extensions ]
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid:always, issuer
basicConstraints = CA:TRUE
keyUsage = cRLSign, keyCertSign
extendedKeyUsage = critical, timeStamping

[ tsa ]
default_tsa = tsa_config1

[ tsa_config1 ]
serial = ./serial
crypto_device = builtin
#certs = ./certs/tsa.pem
signer_digest = sha256
default_policy = 1.2.3.4.1
other_policies = 1.2.3.4.5.6, 1.2.3.4.5.7
digests = sha256, sha384, sha512
ess_cert_id_chain = no
ess_cert_id_alg = sha256
```

执行**:wq!**保存该文件。

2. 执行如下命令创建二级CA时间戳证书。其中**-days 3650**表示二级CA时间戳证书有效期为10年。

openssl genrsa -out tsca_pri.pem 4096

openssl req -new -config x509_tsca.cnf -out tsca.csr -key tsca_pri.pem

openssl x509 -req -sigopt rsa_padding_mode:pss -sigopt rsa_pss_saltlen:-1 -in tsca.csr -CA rootcaG2.pem -CAkey rootca_priG2_pri.pem -CAcreateserial -out tsca.pem -days 3650 -extensions extensions -extfile x509_tsca.cnf

openssl x509 -in tsca.pem -inform pem -outform der -out tsca.der

步骤6 生成三级签名证书。

1. 执行**vi x509_signcert.cnf**命令生成并编写x509_signcert.cnf文件，将如下内容加入到该文件中。

```
[ req ]
default_bits = 4096
distinguished_name = req_distinguished_name
prompt = no
string_mask = utf8only
x509_extensions = extensions

[ req_distinguished_name ]
C = CN
O = Test
CN = Signing Certificate G2
#emailAddress = cmscbb@huawei.com

[ extensions ]
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid, issuer
basicConstraints = CA:FALSE
keyUsage = critical, nonRepudiation, digitalSignature
extendedKeyUsage = codeSigning
```

   执行**:wq!**保存该文件。

2. 执行如下命令创建三级签名证书。其中**-days 3650**表示三级签名证书有效期为10年。

   openssl genrsa -out signcert_pri.pem 4096

   openssl req -new -config x509_signcert.cnf -out signcert.csr -key signcert_pri.pem

   openssl x509 -req -sigopt rsa_padding_mode:pss -sigopt rsa_pss_saltlen:-1 -in signcert.csr -CA signca.pem -CAkey signca_pri.pem -CAcreateserial -out signcert.pem -days 3650 -extensions extensions -extfile x509_signcert.cnf

   openssl x509 -in signcert.pem -inform pem -outform der -out signcert.der

步骤7 生成三级时间戳证书。

1. 执行**vi x509_tsa.cnf**命令生成并编写x509_tsa.cnf文件，将如下内容加入到该文件中。

```
[ req ]
default_bits = 4096
distinguished_name = req_distinguished_name
prompt = no
string_mask = utf8only
x509_extensions = extensions

[ req_distinguished_name ]
C = CN
O = Test
CN = Timestamp Certificate G2
#emailAddress = cmscbb@huawei.com
```



```
[ extensions ]
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid, issuer
basicConstraints = critical, CA:FALSE
keyUsage = critical, nonRepudiation, digitalSignature
extendedKeyUsage = critical, timeStamping

[ tsa ]
default_tsa = tsa_config1

[ tsa_config1 ]
serial = ./serial
crypto_device = builtin
#certs = ./certs/tsa.pem
signer_digest = sha256
default_policy = 1.2.3.4.1
other_policies = 1.2.3.4.5.6, 1.2.3.4.5.7
digests = sha256, sha384, sha512
ess_cert_id_chain = no
ess_cert_id_alg = sha256
```

执行**:wq!**保存该文件。

2. 执行如下命令创建三级时间戳证书。其中**-days 3650**表示三级时间戳证书有效期为10年。

   openssl genrsa -out ts_pri.pem 4096

   openssl req -new -config x509_tsa.cnf -out ts.csr -key ts_pri.pem

   openssl x509 -req -sigopt rsa_padding_mode:pss -sigopt rsa_pss_saltlen:-1 -in ts.csr -CA tsca.pem -CAkey tsca_pri.pem -CAcreateserial -out ts.pem -days 3650 -extensions extensions -extfile x509_tsa.cnf

   openssl x509 -in ts.pem -inform pem -outform der -out ts.der

步骤8 生成吊销列表文件。

须知

- **步骤3**至**步骤7**生成的各类文件需要在吊销列表的有效期内签发开放文件系统的CMS签名文件，否则将验证失败，需要重新生成CMS签名并部署文件系统。
- **default_days=365**表示颁发证书的有效期为365天，**default_crl_days=30**表示30天发布一次证书吊销列表。其中**default_days**需大于等于**default_crl_days**的值，用户可以根据产品生命周期自行调整**default_days**和**default_crl_days**字段，修改吊销列表文件的有效期。

1. 执行**vi x509_crl.cnf**命令生成并编写一级CA吊销列表所需x509_crl.cnf文件，将如下内容加入到该文件中。

```
[ ca ]
default_ca=CA_default
[ CA_default ]
database=./index
crlnumber=./crlnumber
private_key=./rootca_priG2_pri.pem
crl_extensions=crl_ext
default_days=365
default_crl_days=30
default_md=default
preserve=no
[ crl_ext ]
authorityKeyIdentifier=keyid
```

2. 执行**vi signcrl.cnf**命令生成并编写三级签名证书吊销列表所需signcrl.cnf文件，将如下内容加入到该文件中。


```
[ ca ]
default_ca=CA_default
[ CA_default ]
database=./index
crlnumber=./crlnumbersign
private_key=./signca_pri.pem
crl_extensions=crl_ext
default_days=365
default_crl_days=30
default_md=default
preserve=no
[ crl_ext ]
authorityKeyIdentifier=keyid
```

3. 执行**vi tscrl.cnf**命令生成并编写三级时间戳证书吊销列表所需tscrl.cnf文件，将如下内容加入到该文件中。

```
[ ca ]
default_ca=CA_default
[ CA_default ]
database=./index
crlnumber=./crlnumberts
private_key=./tsca_pri.pem
crl_extensions=crl_ext
default_days=365
default_crl_days=30
default_md=default
preserve=no
[ crl_ext ]
authorityKeyIdentifier=keyid
```

4. 执行如下命令生成吊销列表文件。

   touch index

   echo unique_subject=no > index.attr

   echo 01 > crlnumber

   echo 05 > crlnumbersign

   echo 09 > crlnumberts

   openssl ca -gencrl -key rootca_priG2_pri.pem -cert rootcaG2.pem -out crl.pem -config x509_crl.cnf -sigopt rsa_padding_mode:pss -sigopt rsa_pss_saltlen:-1

   openssl crl -in crl.pem -inform pem -outform der -out rootG2.crl

   openssl ca -gencrl -key signca_pri.pem -cert signca.pem -out signcrl.pem -config signcrl.cnf -sigopt rsa_padding_mode:pss -sigopt rsa_pss_saltlen:-1

   openssl crl -in signcrl.pem -inform pem -outform der -out sign.crl

   openssl ca -gencrl -key tsca_pri.pem -cert tsca.pem -out tscrl.pem -config tscrl.cnf -sigopt rsa_padding_mode:pss -sigopt rsa_pss_saltlen:-1

   openssl crl -in tscrl.pem -inform pem -outform der -out ts.crl

步骤9 用户的一级CA根证书rootcaG2.der需要华为进行CMS签名。进行签名前，需要执行如下命令对用户根证书进行预处理，生成摘要信息文件**usrcert.ini**（非首次创建CMS签名时，可跳过该步骤）。

python3 */usr/local/software*/driver/source/vendor/hisi/tools/signtool/image_pack/esbc_header.py -raw_img ./rootcaG2.der -out_img ./user.der -version 1.1.1.1.1 -nvcnt 0 -tag usrcert -platform hi1910p

digest=`sha256sum user.der | awk '{print $1}'`

echo "usrcert, ${digest};" > ./usrcert.ini

步骤10 将上述步骤生成的**usrcert.ini**发送至华为工程师获取签名文件**usrcert.ini.cms**和**usrcert.ini.crl**。之后执行如下命令生成用户根证书文件**user.xer**文件（非首次创建CMS签名时，可跳过该步骤）。

python3 */usr/local/software*/driver/source/vendor/hisi/tools/signtool/image_pack/image_pack.py -raw_img ./user.der -out_img ./user.xer -cms ./usrcert.ini.cms -ini ./usrcert.ini -crl ./usrcert.ini.crl --addcms -version 1.1.1.1.1 -platform hi1910p --pss

步骤11 执行如下命令生成用户吊销列表文件**user.crl**。

cat rootG2.crl sign.crl ts.crl signca.der tsca.der > user.crl

经过以上操作获取到用户根证书文件**user.xer**和用户吊销列表文件**user.crl**。

----结束

13.3.2.3 生成开放态文件系统 CMS 签名文件

本章节将以输入信息摘要文件**initrd.ini**为例，生成开放态文件系统CMS签名文件**initrd.ini.p7s**和签名机构的吊销列表证书**initrd.ini.crl**。在压缩文件系统过程中，**initrd.ini.p7s**和**initrd.ini.crl**需配套当前输入的**initrd.ini**使用。

步骤1 进入Linux服务器任意目录，比如/root/sign，在该目录下生成CMS签名文件和吊销列表。

步骤2 将**步骤5**生成的摘要信息initrd.ini文件拷贝到当前目录下。

步骤3 将**步骤2**创建的CMS签名依赖的各类证书拷贝到当前目录下。

步骤4 参考**13.3.1 安装cmssign工具**完成安装。

步骤5 载入Python虚拟环境，使用cmssign工具生成CMS签名文件。

执行如下命令，对initrd.ini文件进行CMS签名，合成带时间戳的CMS签名文件initrd.ini.p7s。

cmssign sign --signer signcert.der --key signcert_pri.pem --signerCA signca.der --signerCRL sign.crl --rootCRL rootG2.crl --tssigner ts.der --tskey ts_pri.pem --tsCA tsca.der --in initrd.ini

步骤6 执行如下命令，合成签名机构的吊销列表证书crl文件。

cat rootG2.crl sign.crl ts.crl signca.der tsca.der > initrd.ini.crl

完成以上步骤生成initrd.ini.p7s和initrd.ini.crl文件，完成CMS签名。

----结束

13.4 FAQ

13.4.1 PKCS 签名问题导致驱动或固件回退失败、npu-smi 命令异常或 davinci 设备无法启动

问题描述 1

驱动回退过程中出现图13-2所示报错信息，执行npu-smi命令异常，查看“ascend_install.log”日志信息，显示图13-3所示报错：

图 13-2 驱动回退报错

```
Verifying archive integrity...  100%   SHA256 checksums are OK. All good.
Uncompressing ASCEND DRIVER RUN PACKAGE  100%
[Driver] [2022-10-12 03:57:52] [INFO]Start time: 2022-10-12 03:57:52
[Driver] [2022-10-12 03:57:52] [INFO]LogFile: /var/log/ascend_seclog/ascend_install.log
[Driver] [2022-10-12 03:57:52] [INFO]OperationLogFile: /var/log/ascend_seclog/operation.log
[Driver] [2022-10-12 03:57:52] [INFO]base version is 22.0.3.b090.
[Driver] [2022-10-12 03:57:52] [WARNING]Do not power off or restart the system during the installation/upgrade
[Driver] [2022-10-12 03:57:53] [INFO]set username and usergroup, HwHiAiUser:HwHiAiUser
[Driver] [2022-10-12 03:57:54] [ERROR]The software signature verification failed because the signature mode used by the
s [], details in : /var/log/ascend_seclog/ascend_install.log
[Driver] [2022-10-12 03:57:54] [INFO]End time: 2022-10-12 03:57:54
root@localhost old_version_package]# npu-smi info
oot@localhost old_version_package]# npu-smi set -t pkcs-enable -d 0
```

图 13-3 驱动日志信息

```
[Driver] [2022-08-12 11:08:29] [INFO]drv_pcie_hdc_host.ko might not load, try to insmod it
[Driver] [2022-08-12 11:08:29] [INFO]driver in run packet could load ok, no need rebuild
[Driver] [2022-08-12 11:08:29] [INFO]upgradePercentage:10%
[Driver] [2022-08-12 11:08:29] [INFO]upgradePercentage:30%
[Driver] [2022-08-12 11:08:29] [INFO] crl file size check success
[Driver] [2022-08-12 11:08:29] [INFO]case two start: generate new images crl, compare newer and check cms
[Driver] [2022-08-12 11:08:29] [ERROR]PSS signature verification failed, stop install
[Driver] [2022-08-12 11:08:29] [ERROR]gen_cur_image_crl_compare_newer failed
[Driver] [2022-08-12 11:08:29] [ERROR]hw_crl_check failed.
[Driver] [2022-08-12 11:08:29] [ERROR]cur device crl check fail, stop upgrade
```

固件回退过程中出现图13-4所示报错，查看“ascend_install.log”日志信息，显示图13-5所示报错：

图 13-4 固件回退报错

```
Verifying archive integrity...  100%   SHA256 checksums are OK. All good.
Uncompressing ASCEND310P-HDK-NPU FIRMWARE RUN PACKAGE  100%
[Firmware] [2022-10-11 06:22:58] [INFO]Start time: 2022-10-11 06:22:58
[Firmware] [2022-10-11 06:22:58] [INFO]LogFile: /var/log/ascend_seclog/ascend_install.log
[Firmware] [2022-10-11 06:22:58] [INFO]OperationLogFile: /var/log/ascend_seclog/operation.log
[Firmware] [2022-10-11 06:22:59] [INFO]base version is 1.83.10.1.236.
[Firmware] [2022-10-11 06:22:59] [WARNING]Do not power off or restart the system during the installation/upgrade
[Firmware] [2022-10-11 06:23:08] [INFO]upgradePercentage: 0%
[Firmware] [2022-10-11 06:23:14] [ERROR]Firmware upgrade failed, details in : /var/log/ascend_seclog/ascend_install.log
[Firmware] [2022-10-11 06:23:14] [INFO]Failed to upgrade firmware package, please retry after uninstall and reboot!
[Firmware] [2022-10-11 06:23:14] [INFO]End time: 2022-10-11 06:23:14
```

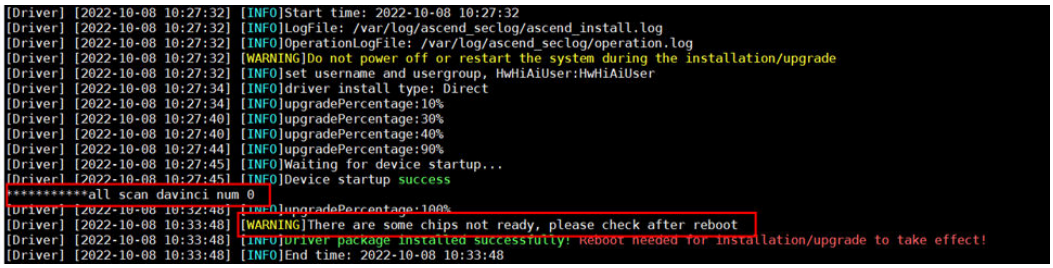
图 13-5 固件日志信息

```
[Firmware] [2022-10-11 06:23:14] [INFO]firmware upgrade log <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
[Firmware] [2022-10-11 06:23:14] [ERROR]firmware upgrade failed
[Firmware] [2022-10-11 06:23:14] [INFO]End time: 2022-10-11 06:23:14
```

问题描述 2

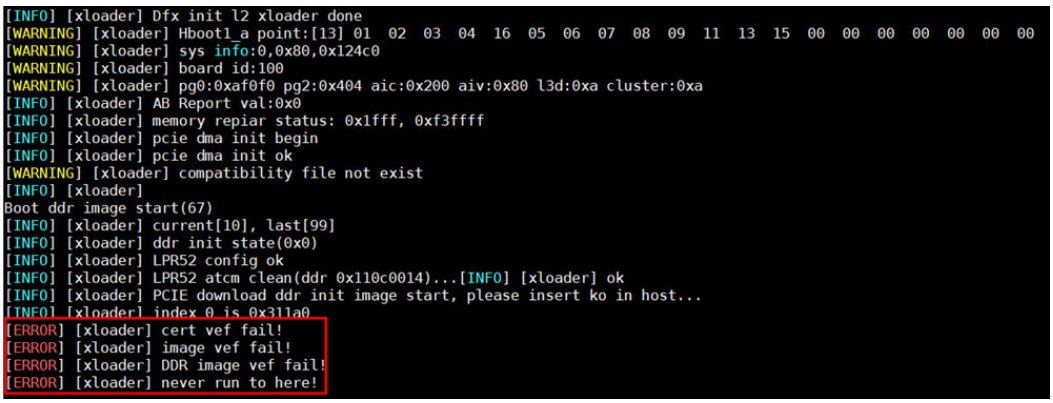
驱动安装过程中davinci设备无法启动，出现图13-6所示报错：

图 13-6 davinci 设备无法启动



使用**msnpureport -f** 收集device侧日志，查看“hisi_logs/device-0/20221008103618-286885000/snapshot/hdr.log”日志，出现图13-7所示校验失败日志信息：

图 13-7 校验失败



可能原因

签名校验失败。

解决方案

步骤1 安装NPU 22.0.3及以上版本驱动和配套固件（NPU驱动和固件包名称中包含的版本为6.0.RC1，但是部署驱动和固件后使用npu-smi命令查询获取的驱动版本为22.0.3，固件版本为1.83.10.1.X）。

步骤2 设置PKCS的状态为“使能”。执行如下命令：

npu-smi set -t pkcs-enable -d *0*

如出现如下回显，则表示设置成功。

```
[WARNING]: Enabling pkcs will raise safety risks.
    Status                  : OK
    Message                 : The pkcs-enable is set successfully
```

具体请参见《**Ascend 310P 23.0.RC2 npu-smi 命令参考（AI加速卡）**》。

步骤3 重新执行回退操作。

先安装22.0.2及之前版本的固件包，此时不能重启。再安装对应版本的驱动包，最后重启（NPU驱动和固件包名称中包含的版本为5.1.RC2，但是部署驱动和固件后使用npu-smi命令查询获取的驱动版本为22.0.2，固件版本为1.82.22.2.X）。

⚠ 注意

重启系统后，PKCS的状态会恢复为默认“禁用”状态，设置为“使能”状态后请直接执行回退操作。

步骤4 若回退至CANN 5.1.RC2版本及之前的自定义文件系统，回退版本时需同时将/usr/local/CMS下的两个文件修改为版本配套的证书文件，可参考**5.4.2 设置用户根证书信息**。

----结束