寒武纪 1H8 边缘智能平台

用户手册 V1.2.1

版本修订记录

名称	寒武纪 1H8 边缘智能平台用户手册		
版本号	V1.2.1		
创建日期	2019.01.09		
创建人	徐友庆		
更新历史			
修改时间	修改人	版本号	修改内容
2019.01.09	徐友庆	V1.1.0	基于 V3.0 版本创建
2019.01.30	周亿文	V1.1.1	完善开发板测试过程部分
2019.02.02	周亿文	V1.1.2	增加网络模型重训练过程部分
2019.03.20	周亿文	V1.1.3	基于 V7.9，增加定点量化过程部分
2019.03.21	周亿文	V1.1.4	基于 V7.9，增加示例程序部分
2019.04.02	周亿文	V1.2.0	1. 拆分原来第三章软件平台为开发板的使用和宿主机开发两章； 2. 宿主机的开发中增加了大量内容：sdk 软件包介绍/应用程序的移植/交叉编译，详细说明了 1H8 整个应用程序移植开发的过程。
2019.07.20	徐友庆	V1.2.1	增加详细的操作与使用方法 增加开发环境配置 调整目录结构，完善细节

service：yqxu@hchsmart.com

目 录

第一章 平台介绍

寒武纪 1H8 边缘智能平台是一款集成寒武纪 1H8 智能终端处理器 IP、可用于边缘端人工智能计算的开发平台，包含硬件开发平台 HBoard，以及配套的软件开发包 SDK。

平台概述

寒武纪 1H8 边缘智能平台是基于 HBoard 硬件开发平台的一套边缘端人工智能应用的解决方案，它是以寒武纪 1H8 智能终端处理器为核心的人工智能计算平台，寒武纪 1H8 智能终端处理器以寒武纪 1H8 IP 核为核心计算单元，ARM Cortex A7 为主控单元的人工智能处理器芯片，可实现语音识别、人脸识别、目标识别等典型前端人工智能类应用。

平台特性

1. 低功耗、高性能
   寒武纪 1H8 是一款应用于嵌入式终端的人工智能处理器，主频可达 1GHz，典型功耗不超过 5W。可进行复杂的深度学习网络模型推理计算，整型运算能力可达 0.8Tops。

2. 支持多种深度学习网络模型
   支持多种深度学习网络模型，如常见的 AlexNet，GoogLeNet，VGG，MobileNet，MTCNN，ResNet，RCNN 系列，YOLO 系列，SSD 等，可支持 Caffe 和 TensorFlow 两个深度学习框架的模型。

3. 丰富的外部接口
   与其它嵌入式产品类似，HBoard 支持丰富的外围接口，像常见的 GPIO，PWM，UART，I2C，SPI，USB，WIFI，AUDIO，VIDEO，SD，Ethernet 等。

4. 快速定制
   HBoard 采用核心板（HCore）+ 底板的设计方式，在不修改核心板的情况下，只需要简单修改底板外围接口，即可针对不同应用场景进行快速定制。

5. 国产自主可控
   1H8 智能终端处理器以寒武纪 1H8 IP 核为核心计算单元，核心技术由中科寒武纪公司掌握，该公司由中科院计算所孵化而出，是目前全球智能芯片领域的引领者。1H8 是寒武纪第二代智能终端 IP 产品，专为机器视觉应用而设计，与初代产品 1A 相比，在同样的处理能力下具有更低的功耗和更小的面积。

平台架构

此平台的总体架构如图 1-1 所示，主要可以分成四层，硬件层、系统层、API 层和应用层。
硬件层：由 HBoard 开发板构成，主要包含核心板 HCore 以及外围接口底板。
系统层：嵌入式 Linux 操作系统，并包含相关的驱动程序。
API 层：可以分成两类接口，一类为调用 1H8 核相关的深度学习类接口；另一类为调 Video/Audio/WiFi 等其他通用接口。
应用层：可以分成两类应用，一类为人工智能类应用，如图像分类、目标检测、语音识别等；另一类为通用应用，如摄像头实时视频获取、音频播放、网络传输等。

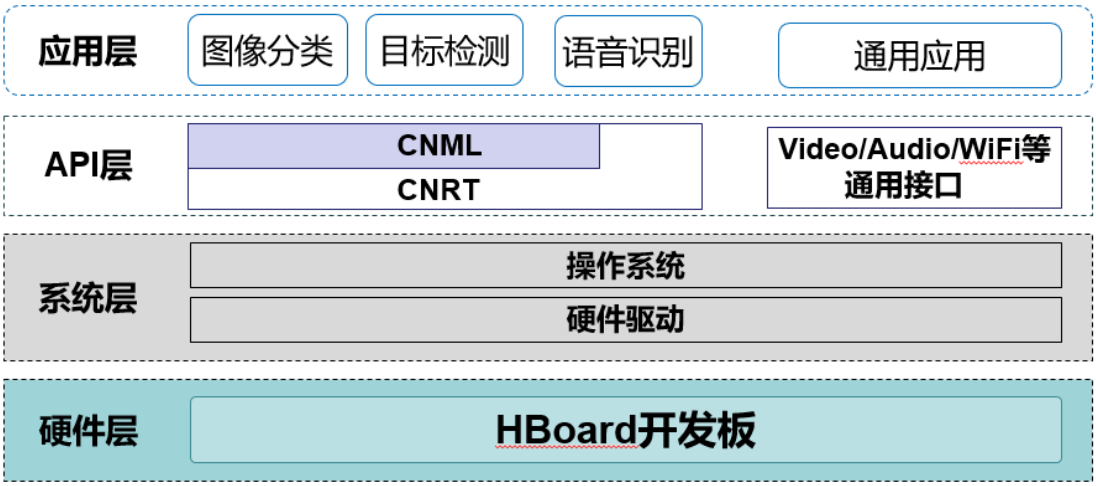


图 1-1 平台架构

开发模式

由于应用程序的开发需要在 Linux 操作系统上进行，因此，可通过两种途径来构建开发环境，第一种是通过 Window＋Linux 虚拟机的方式；第二种直接使用 Linux 物理机的方式，两种方式均可。对于初学者、Linux 操作不熟练的用户推荐使用第一种方式，对于 Linux 的熟练操作用户推荐使用第二种方式。

HBoard 可独立运行嵌入式 Linux 操作系统，并运行相应的应用程序，宿主机可通过 Debug 串口或以太网口与 HBoard 进行交互，便于控制程序的运行与调试等操作。

Window+Linux 虚拟机

由于初学者对 Linux 的操作并不熟练，可先采用 Windows + Linux 虚拟机的开发模式，部分操作可在 Windows 系统下进行，方便操作。虚拟机安装桌面 Linux 操作系统（如 Ubuntu16.04 Desktop），便于程序开发。

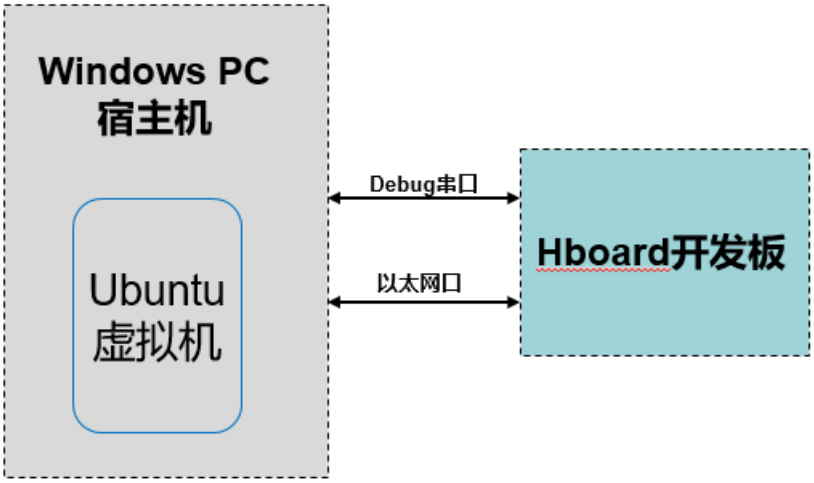


图 1-2 Windows+Linux 虚拟机开发模式

Linux 物理机

Linux 物理机是指在实体机上直接安装 Linux 操作系统，推荐使用 Ubuntu16.04 系统，因为后续的环境配置及程序开发均在此版本的系统下进行的。

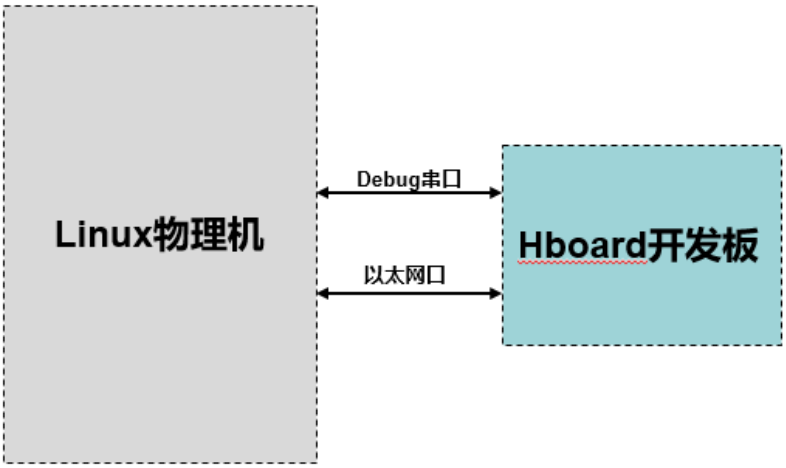


图 1-3 Linux 物理机开发模式

第二章 硬件介绍

此部分主要介绍边缘智能平台所依赖的硬件 HBoard 开发板，包括硬件总体架构以及其他各硬件模块的详细信息。

硬件总体架构

HBoard 硬件平台采用核心板（HCore）+ 底板的设计方式，其中核心板 HCore 由一颗包含寒武纪 1H8 智能终端处理器 IP 和 ARM Cortex A7 的处理器芯片 HChip、128MB 的 DDR3 内存存储器、128MB 的外部 Nand Flash 存储器、512MB 的外部 eMMC 存储器（可选）以及 WIFI 芯片组成。底板则是包含一些常见的外围接口，如 MIPI Camera，Audio，USB，SD，Ethernet，SPI，I2C，UART，GPIO，PWM 等，这些外部接口可以根据不同的应用场景进行快速定制，硬件总体架构如图 2-1 所示。

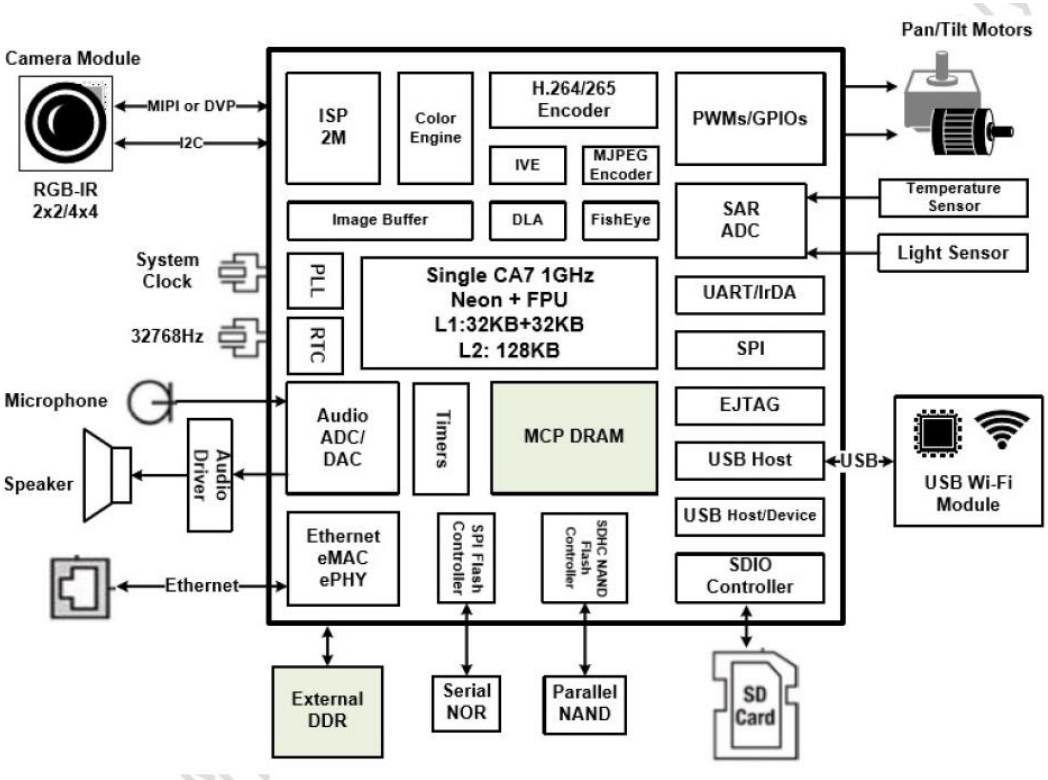


图 2-1 HBoard 硬件总体架构

核心板

核心板（如图 2-2 所示）主要包括五个模块，HChip 处理器芯片，DDR3 内存颗粒，外部存储器 Nand Flash，外部存储器 eMMC（可选）以及 WIFI 芯片。其中，HChip 处理器为集成寒武纪 1H8 智能终端处理器 IP、ARM Cortex A7 以及视频编解码为一体的智能终端处理器；DDR3 内存的容量为 128MB，数据位宽为 16bit；外部存储器 Nand Flash 的容量为 128MB；WIFI 芯片支持 IEEE 802.11b/g/n 标准和 1T1R 工作模式，具体配置如表 2-1 所示：

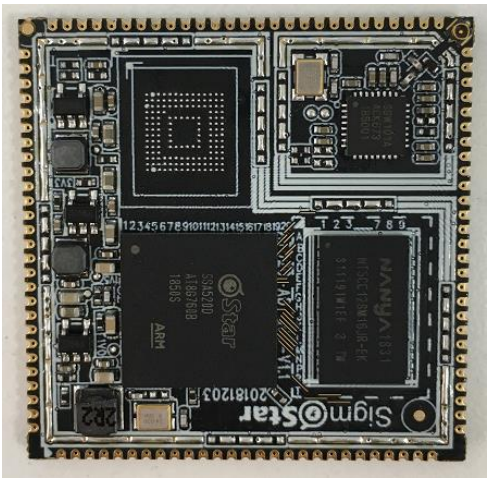


图 2-2 核心板

表 2-1 核心板硬件配置

配置项	说明
处理器	HChip，单核，主频 1GHz
内存	128MB 16bit DDR3
外存 1	128MB SPI Nand Flash，背面
外存 2	512M eMMC，可选
WIFI	支持 IEEE 802.11b/g/n 标准的 1T1R 工作模式

底板

底板提供了丰富的外部接口，包括 GPIO，PWM，UART，I2C，SPI，USB，AUDIO，VIDEO，SD，Ethernet 等，这些接口可以根据不同应用场景进行定制。外部接口的详细类型与数量如表 2-2 所示：

表 2-2 底板外部接口

接口类型	接口说明	数量
GPIO	标准输入输出	6
UART	串口	2
I2C		4
SPI	SPI 串行接口	1
ADC	模数转换器	2
DAC	数模转换器	2
USB	USB Host 接口	2
SDIO		1
Ethernet	10Mbps/100Mbps 以太网	1
VIDEO	● 4K2KP15+720P15+VGAP15 H265 encoder ● 5MP30+720P30+VGAP30 H265 encoder ● 4K2KP15+720P15+VGAP15 H264 encoder ● 5MP30+720P30+VGAP30 H264 encoder	/
AUDIO	● PDM 数字 MIC，最大支持 4 MIC 阵列 ● 模拟硅麦，最大支持 2 MIC 阵列	/

	● 4 欧 2 瓦 Speaker	
LCD	● 支持 TTL 显示屏 ● 最高支持 1280x720/60p	1

硬件实物

HBoard 硬件实物如图 2-3 所示，各部分描述如表 2-3 所示。

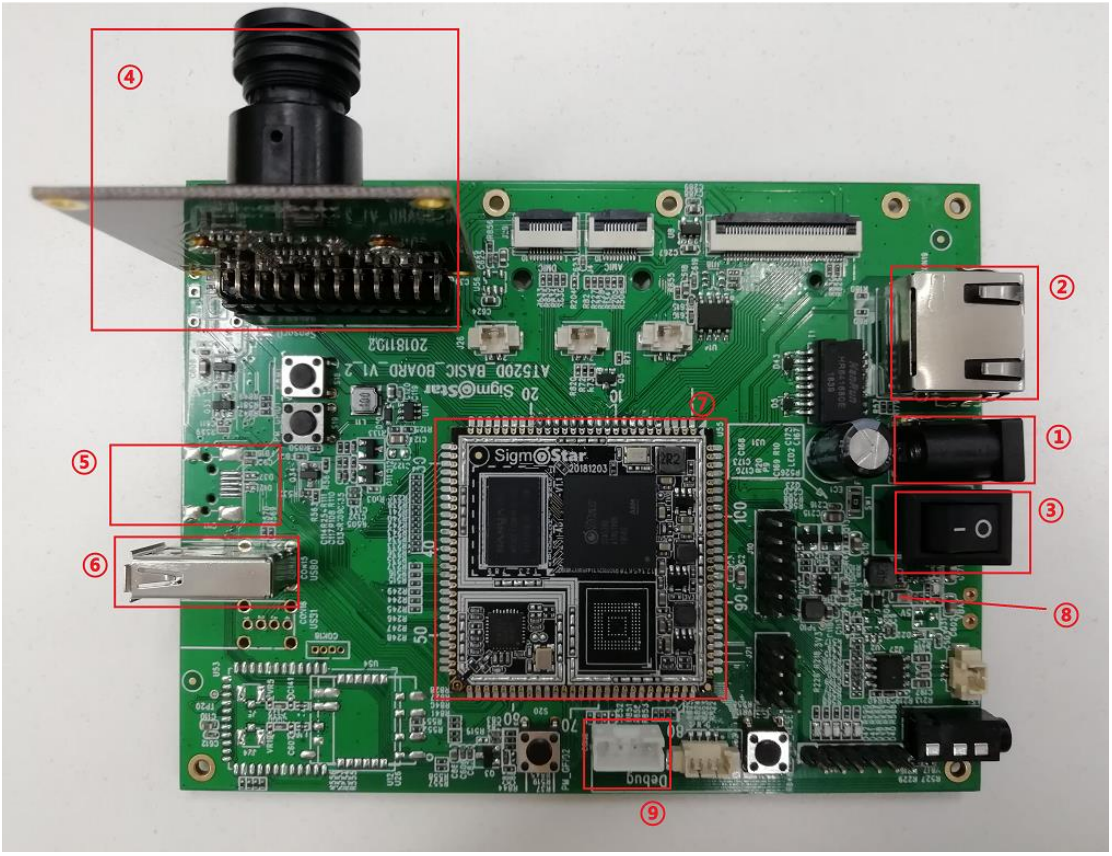


图 2-3 HBoard 硬件实物

表 2-3 HBoard 硬件模块介绍

序号	模块	描述
①	电源接口	为开发板供电(请使用配备的 12V 2A 电源适配器)
②	Enternet 接口	RJ45 以太网口
③	电源开关	电源开关打开时，电源指示灯常亮
④	摄像头	用于实时视频信号采集
⑤	SD 卡(背面)	用于数据的存储
⑥	USB 接口	用于数据传输
⑦	核心板	承担开发板的计算工作
⑧	电源指示灯	电源开关③打开时，指示灯常亮
⑨	串口	调试串口，用于操作开发板，通过串口工具与宿主机相连

第三章 操作与使用方法

此部分主要介绍如何操作与使用此开发平台的具体方法，包括电源线、调试线的连接方法，登录操作基本方法，以及如何运行一个示例程序的方法等。

电源及接线

① 使用配备的 12V/2A 电源适配器与 HBoard 开发板的电源接口相连。
② 使用网口连接时，网线一端连接 HBoard 开发板网口，另一端连接电脑网口。
③ 使用 Debug 串口连接时，串口线与 HBoard 串口槽口（如图 2-3）连接。
④ 打开电源开关，电源指示灯常亮。

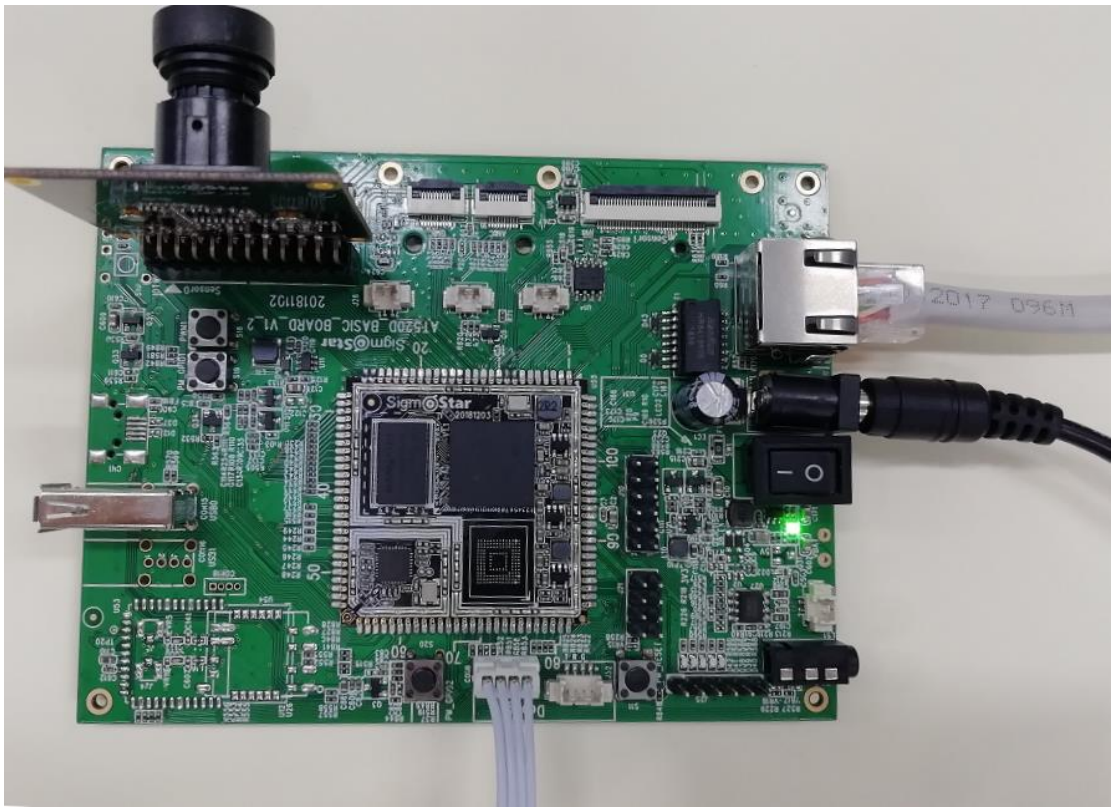


图 3-1 开发板连线

登录到开发板

此部分主要介绍登录到开发板的具体配置操作方法，提供 Debug 串口和以太网两种方式。其中，Debug 串口主要用于异常情况下系统级的调试时使用，正常操作可以通过 telnet 方式。

Debug 串口登录

此部分详细介绍如何在 Windows 系统和 Linux 系统下通过 Debug 串口登录到开发板的

具体操作方法，包括如何安装 USB 转串口工具驱动程序，以及配置串口参数的方法。

windows 系统

使用随开发板附带的串口线与开发板上的串口槽相连，打开 Windows 上的设备管理器，在端口一栏查看该串口在 Windows 上的端口号(示例中为 COM6)，如图 3-2 所示。如果端口处显示为黄色感叹号，请先安装 USB 转串口驱动程序。

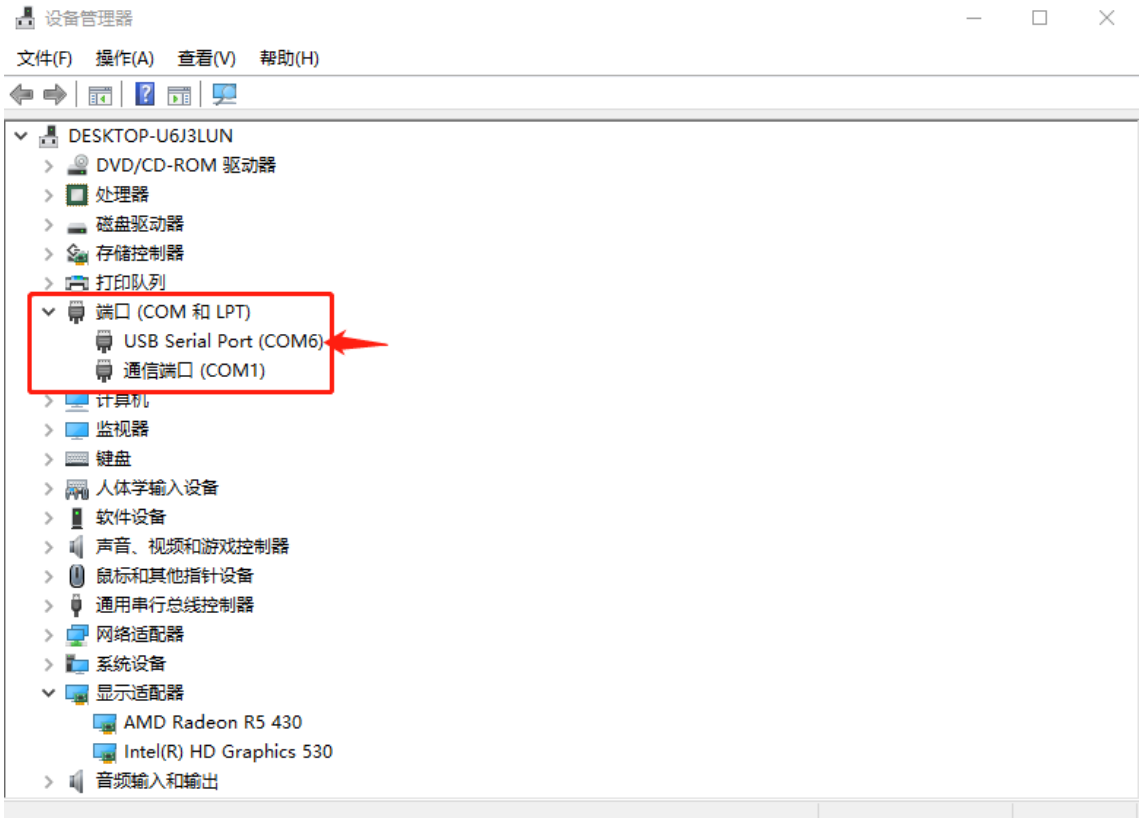


图 3-2 Windows 设备管理器

打开 putty 工具，按照如下配置后点击 Open 即可

- Connection type：Serial
- Serial line：COM6
- Speed：115200

注：可在 Saved Sessions 设置会话名称，点击 save 保存会话，避免每次登录时重新配置。

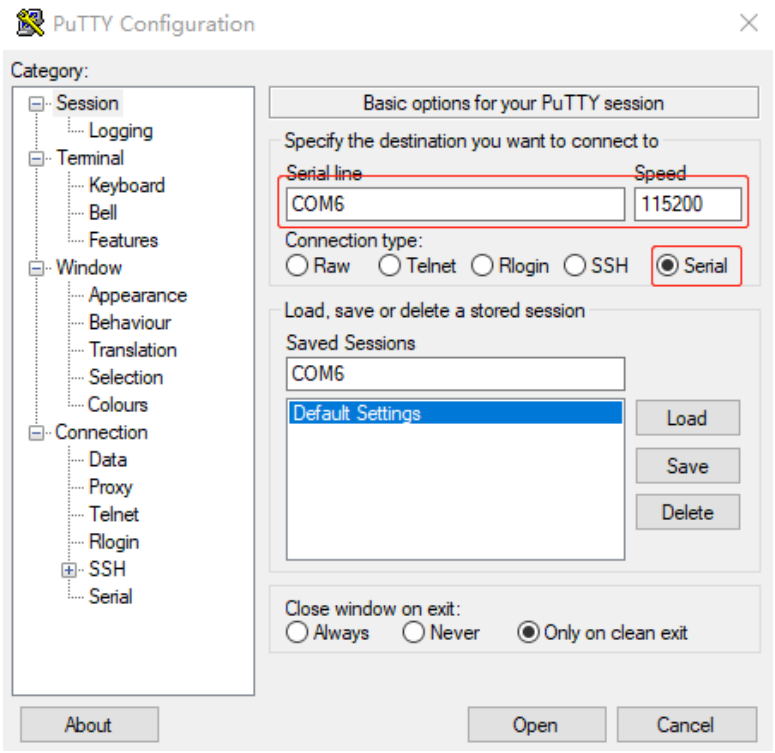
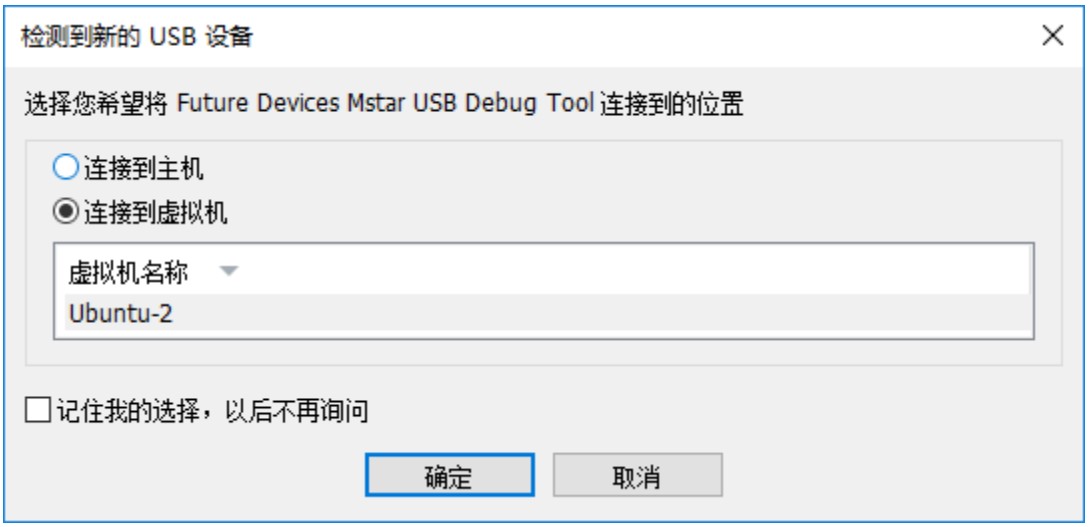


图 3-3 Putty 串口设置

Linux 系统

① 将开发板的串口连接到宿主机的 USB 接口，选择连接到虚拟机。



② 命令行执行如下命令，查看串口工具是否可用

```
ls /dev | grep ttyUSB*
```

```
hchsmart@hchsmart-virtual-machine:~$ ls /dev | grep ttyUSB*
ttyUSB0
ttyUSB1
```

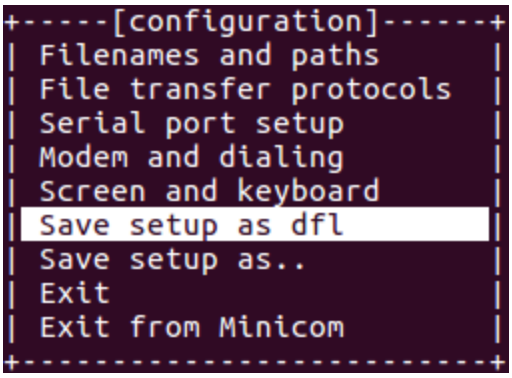
如果出现如上图所示信息则代表串口工具可用。

③ 命令行执行以下命令选择 Serial-port-setup（串口配置）修改 minicom 的配置：

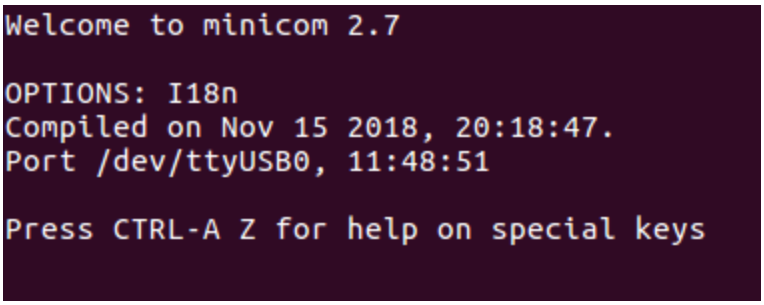
```
sudo minicom –s
```

将设备名修改为 ttyUSB1，波特率、数据位和停止位设置为 115200 8N1，硬件流控制和软件流控制都选择 No，修改后的配置如下图所示：

④ 按 enter 键并选择 save setup as dfl 保存配置，重启 minicom：



⑤ 执行 sudo minicom，上面显示 online 代表已连接。



Telnet 网络登录

此部分详细介绍如何在 Windows 系统和 Linux 系统下通过网络登录到开发板的具体操作方法，包括如何配置虚拟机及宿主机 IP 地址的方法。

Windows 系统

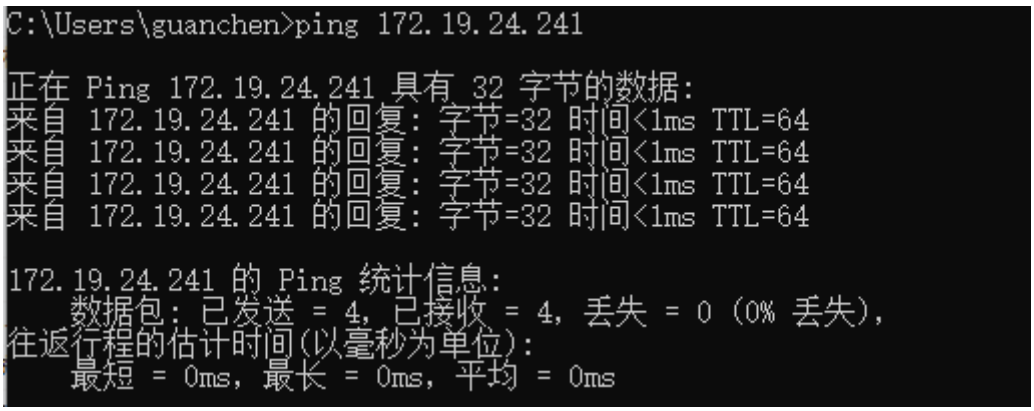
在使用 Telnet 命令之前需要先配置 Windows 系统的网络，使其与 HBoard 开发板处于同

一网段，其中，HBoard 开发板默认 IP 为 172.19.24.241。具体设置方法如下：

1）打开网络连接，右键单击以太网→属性

2）双击 IPv4 网络协议进行设置，将 IP 地址设置为 172.19.24.0 网段，点击确定。

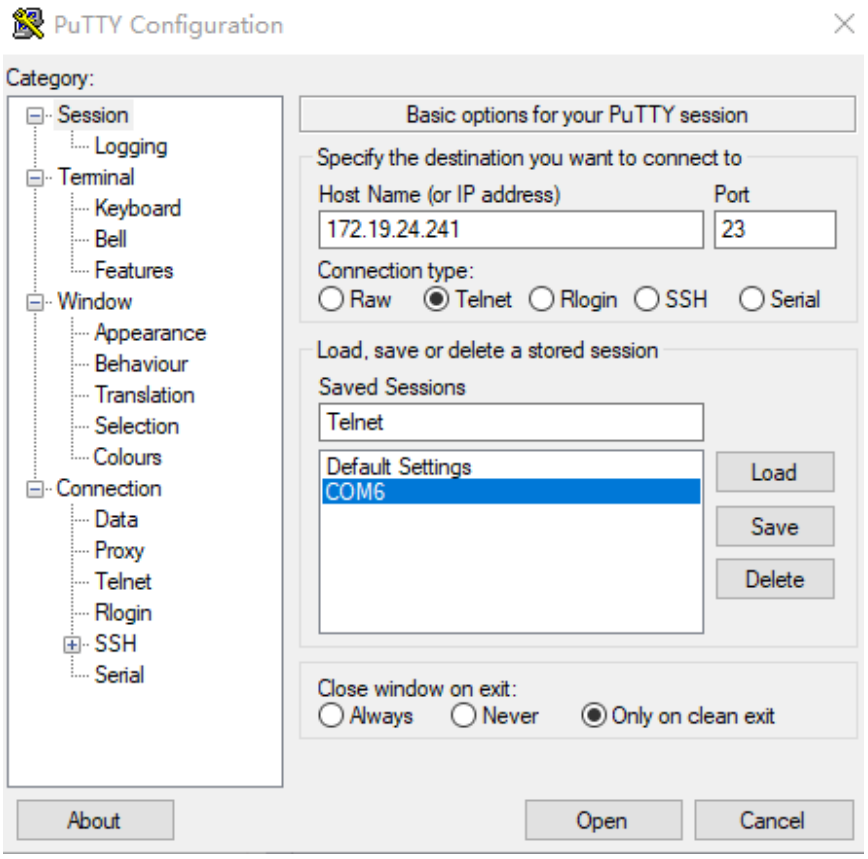
3）打开命令提示符执行 ping 172.19.24.241 命令，查看 pc 能否 ping 通开发板。



4）打开 putty 工具，按照如下参数配置，然后点击 open 即可。

- Connection type：Telnet

- Host Name (or IP address)：172.19.24.241



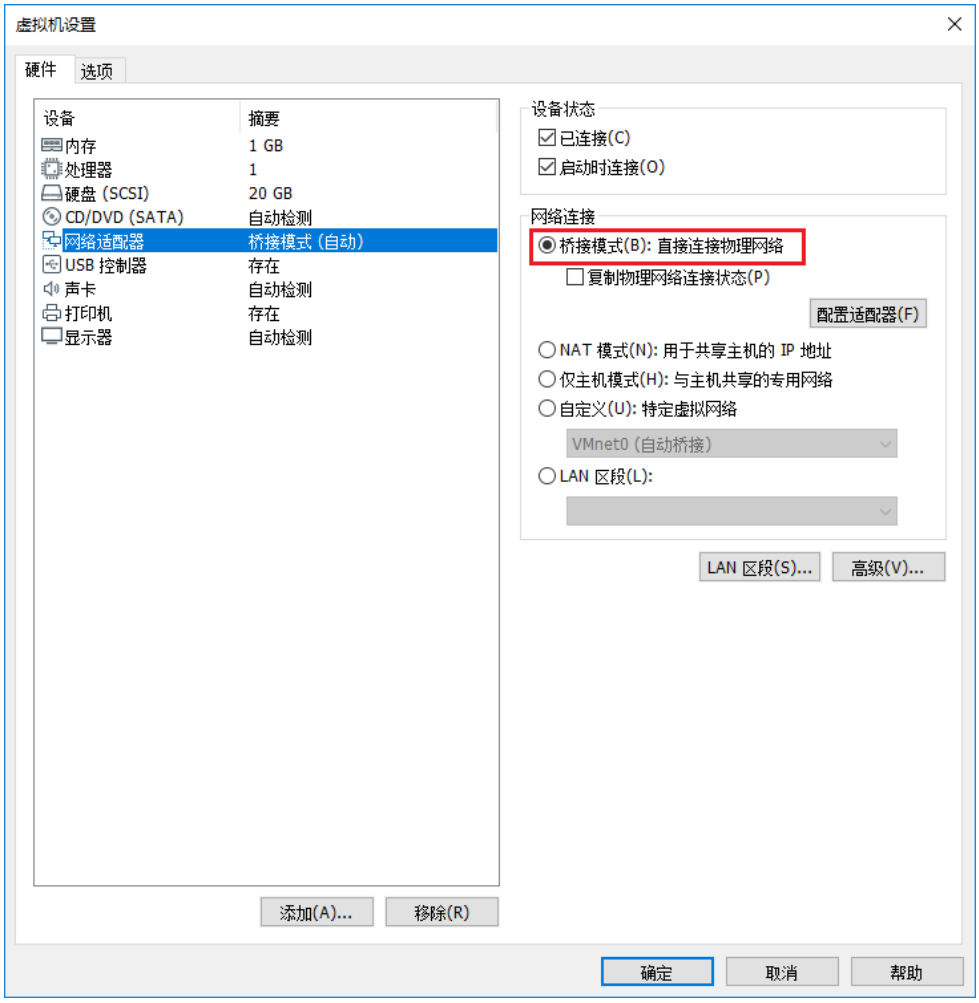
Linux 系统

在设置 Linux 系统之前需先保证宿主机 Windows 的 IP 已设置正确，并将虚拟的网络模式设置为 Bridge 桥接模式。具体设置方法如下：

1） Windows IP 设置如上

2） 虚拟机网络模式设置

打开虚拟机设置->网络适配器，将“网络连接”设置为“桥接模式”，然后点击“确认”，如下图所示。



3） Ubuntu IP 设置

将虚拟机 IP 地址与开发板（默认设置为 172.19.24.x 网段）设置在同一网段，使用 ifconfig 命令设置。

```
$ sudo ifconfig eth0 172.19.24.200 up
```

注：此设置结果为临时 IP，重新打开虚拟将失效，需重新设置，注意 IP 冲突。

① 执行 ping 命令

在虚拟机命令行中执行如下命令，查看虚拟机能否 ping 通开发板。

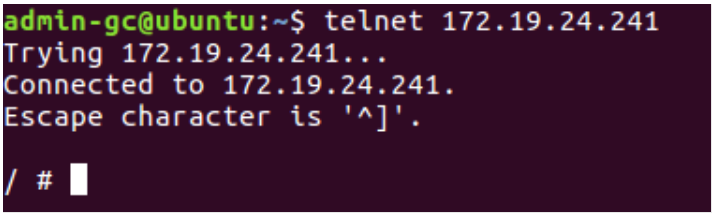
```
$ping 172.19.24.241
```

如果 ping 通，结果如下图所示。

```
admin-gc@ubuntu:~$ ping 172.19.24.241 -c10
PING 172.19.24.241 (172.19.24.241) 56(84) bytes of data.
64 bytes from 172.19.24.241: icmp_seq=1 ttl=64 time=0.562 ms
64 bytes from 172.19.24.241: icmp_seq=2 ttl=64 time=0.578 ms
64 bytes from 172.19.24.241: icmp_seq=3 ttl=64 time=0.463 ms
64 bytes from 172.19.24.241: icmp_seq=4 ttl=64 time=0.536 ms
64 bytes from 172.19.24.241: icmp_seq=5 ttl=64 time=0.529 ms
64 bytes from 172.19.24.241: icmp_seq=6 ttl=64 time=0.595 ms
64 bytes from 172.19.24.241: icmp_seq=7 ttl=64 time=1.14 ms
64 bytes from 172.19.24.241: icmp_seq=8 ttl=64 time=0.500 ms
64 bytes from 172.19.24.241: icmp_seq=9 ttl=64 time=0.610 ms
64 bytes from 172.19.24.241: icmp_seq=10 ttl=64 time=0.586 ms
```

4） Telnet 到 HBoard

```
$ telnet 172.19.24.241     #注:IP 地址会根据硬件平台的不同而变化
```



演示示例

实时人脸检测

此示例是运行在 HBoard 平台上的一个实时人脸识别的演示程序，它会在系统启动后默认执行。其实质是在后台运行一个 RTSP 服务器（程序是/customer/sample_fr_fast），它通过 HBoard 开发板自带的摄像头实时获取视频数据，并结合深度学习算法对视频中的人脸进行实时检测，用户可通过客户端 VLC 播放器观看到实时处理效果，具体操作步骤如下：

1. 启动 HBoard
   用网线将 HBoard 与安装好 windows 操作系统的 PC 机(windows 7/8/10)相连，接入 12V 电源，打开电源开关启动 HBoard

2. 设置 PC 机 IP 地址
   将 PC 机 IP 地址与 HBoard 设置在同一网段，HBoard 默认为 172.19.24.x 网段，具体的设置方法如上一节。设置完成后打开 cmd 控制台，执行 ping 172.19.24.241 判断 PC 与 HBoard 的网络是否连通。

3. 在 PC 机上打开 VLC 播放器

请预先安装好 VLC 播放器，然后打开媒体→网络串流，输入如下命令后点击“播放”按钮

```
rtsp://172.19.24.241/main_stream    #为具体开发板 IP，根据实际情况填写
```

图 3-4 VLC 打开网络流

4. 然后即可得到实时的视频流效果，如图 3-5 所示。

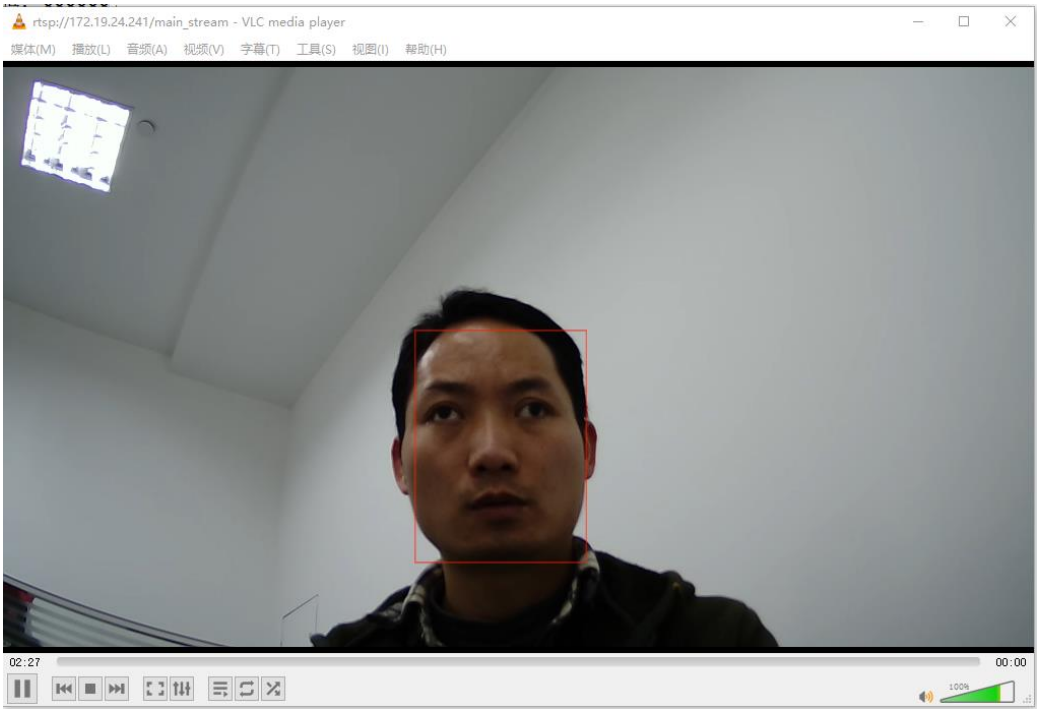


图 3-5 实时人脸检测

第四章 宿主机程序开发

本章主要介绍如何在宿主机上开发深度学习应用程序的流程及具体方法，主要包括SDK 软件包介绍，应用程序的移植过程，程序的交叉编译方法。

SDK 软件包

SDK（Software Development Kit）软件开发工具包，主要为进行软件程序开发所需要的一整套软件工具，包括.h 头文件、.so 库文件，示例程序源码，编译好的可执行示例程序，模型文件，交叉编译工具链，以及相关的 Shell 脚本，cmake 规则等。
SDK 根目录内容如图 4-1 所示：

```
zhou@ubuntu:~/work/1h8/HCAI1H_Devkit_Realease_V7.9$
zhou@ubuntu:~/work/1h8/HCAI1H_Devkit_Realease_V7.9$ tree -L 2
.
├── build_app_test.sh -> ./common/scripts/build_app_test.sh
├── common
│   ├── bin
│   ├── include
│   ├── lib
│   ├── scripts
│   └── toolchain
├── HCAI1H_SDK
│   ├── app
│   ├── data
│   └── offline_generate
├── HCAI1H_Test
│   └── caffe_offline
└── Version

12 directories, 2 files
zhou@ubuntu:~/work/1h8/HCAI1H_Devkit_Realease_V7.9$
```

图 4-1 SDK 目录结构

build_app_test.sh 是一个软链接，指向./common/scripts/build_app_test.sh,用于编译应用程序
common 目录下交叉编译器，头文件、依赖库，以及 Caffe 工具
HCAI1H_SDK 目录下存放应用程序源码及测试工具源码和网络模型原始文件
HCAI1H_Test 目录下存放各种 caffe 离线网络模型的测试文件
Version 是一个文本文件，记录当前 SDK 的版本

应用程序移植

此部分主要介绍宿主机上的应用程序移植方法，即如何把一个第三方深度学习应用程序在 Hboard 开发板上运行起来。归纳起来主要有三个过程：

1. 8bit 模型生成

   由于 8bit 神经网络模型能够显著减少模型占用的存储空间和处理带宽，目前 1H8 只支持 8bit 模型，而在其它平台（如 CPU 或 GPU）上训练出来的模型（以 caffe 为例，模型后缀为.caffemodel）大多为 float32 或 float16，因此需要将 FP32 或 FP16 格式的模型转换成 8bit 格式的模型。

2. 离线模型生成
   将.caffemodel 格式的模型转化为寒武纪支持的.cambricon 格式的离线模型。

3. 推理程序开发
   为了完成相应的功能（如图像分类或目标识别），需要编写一个应用程序来完成输入图片预处理、模型加载与推理计算、输出结果后处理等一系列操作。

8bit 模型生成

8bit 模型生成有两种方式，一种是直接量化；另一种是重训练。由于重训练的过程较为复杂与耗时，除非在对精度有非常严苛要求的情况下才采用重训练的方式，一般都推荐采用直接量化的方式。

直接量化

一般情况下，可以直接量化生成 fix8 网络模型的.prototxt 网络文件，即依据当前模型的权重和输入等参数，将其直接转换为 8bit 表示，但难以避免的会造成精度损失，这时可以引入 scale 参数，对数据分布进行相应的缩放，使其与 8bit 数据的值接近，从而提高数据表示精度。

使用 SDK 提供的量化工具可以从原有的模型中计算得到 8bit 信息，生成新的.prototxt 文件，用户可以通过修改.ini 格式的配置文件进行量化操作。

1. 修改 convert_fix8.ini 配置文件

[model] 节点：
original_models_path： 待量化的原始网络 eg: {YOUR_PATH}/ssd.prototxt
save_model_path： 量化后的网络 eg: {YOUR_PATH}/ssd.fix8.prototxt

[data] 节点：
images_folder_path： 图片路径 eg: {YOUR_PATH}/ssd_file_list
images_db_path： lmdb 的路径 eg: {YOUR_PATH}/ssd_lmdb
used_images_num： 迭代的次数。当网络的 batchsize 为 1，该值为 8 时，共迭代 8 次，读取 8 张图片；当网络的 batchsize 为 4，该值为 8 时，共迭代 8 次，读取 32 张图片。
注：image_folder_path 和 images_db_path 一次只能使用其中一个，不能两个同时指定使用。

[weights] 节点：
original_weights_path： 待量化网络的.caffemodel 权重文件

[preprocess] 节点：
mean： 均值
std： 缩小倍数，如 mobilenet 中的 0.017，实际意义是让输入乘以 0.017 倍。这个 std 参数会在生成的 fix8 网络的第一层卷积处，默认是 1。
scale：输入图像的高度和宽度，如: 224, 224。当用到 images_folder_path（图片）的时候，也就是输入层是 ImageData 层，对应 new_height 和 new_width 这两个参数，代表的是网络的输入层的高度和宽度。

crop： 裁剪的高度和宽度，如： 224, 224。当用到 images_db_path（LMDB）的时候，也就是输入层是 Data 层，对应 crop_size 的值。crop: 224, 224 代表的是网络输入层的高度和宽度。

[config] 节点：
fix8_op_list： 8bit 算子列表
use_firstconv： 是否使用到第一层卷积

convert_fix8.ini 文件示例如下，其中 path 要根据自己环境中的路径修改，使用绝对路径

```
： convert_fix8.ini
[model]
;blow two are lists, depending on framework
original_models_path = /path/ssd_deploy.prototxt
save_model_path = /path/ssd_deploy.fix8.prototxt
;input_nodes = input_node_1
;output_nodes = output_node_1, output_node_2

[data]
;only one should be set for below two
;images_db_path = /path/data/ilsvrc12_val_lmdb/
images_folder_path = /path/datasets/imagenet/file_list
used_images_num = 1

[weights]
original_weights_path = /path/ssd_deploy.caffemodel

[preprocess]
mean = 104,117,123
std = 0.017
scale = 300, 300
crop = 300, 300

[config]
fix8_op_list = Conv, FC, LRN
use_firstconv = 1
```

2. 量化操作

a）设置环境变量

```
$ export LD_LIBRARY_PATH=${SDK_ROOT}/common/lib/build.x86_64
```

其中，${SDK_ROOT}为 SDK 在实际环境中的绝对路径

b）Caffe 工具中带有量化工具，直接执行

```
$ cd common/bin/build.x86_64/caffe/tools/
$ ./generate_fix8_pt –ini_file convert_fix8.ini
```

在指定的 save_model_path 目录下得到量化后的 *ssd_deploy.fix8.prototxt* 文件。

```
I0626 11:37:44.939666 27362 net.cpp:1592] Ignoring source layer conv7_2_mbox_loc_flat
I0626 11:37:44.940017 27362 net.cpp:1592] Ignoring source layer conv7_2_mbox_conf_perm
I0626 11:37:44.940026 27362 net.cpp:1592] Ignoring source layer conv7_2_mbox_conf_flat
I0626 11:37:44.940030 27362 net.cpp:1592] Ignoring source layer conv7_2_mbox_priorbox
I0626 11:37:44.940114 27362 net.cpp:1592] Ignoring source layer conv8_2_mbox_loc_perm
I0626 11:37:44.940121 27362 net.cpp:1592] Ignoring source layer conv8_2_mbox_loc_flat
I0626 11:37:44.940488 27362 net.cpp:1592] Ignoring source layer conv8_2_mbox_conf_perm
I0626 11:37:44.940500 27362 net.cpp:1592] Ignoring source layer conv8_2_mbox_conf_flat
I0626 11:37:44.940505 27362 net.cpp:1592] Ignoring source layer conv8_2_mbox_priorbox
I0626 11:37:44.940582 27362 net.cpp:1592] Ignoring source layer pool6_mbox_loc_perm
I0626 11:37:44.940588 27362 net.cpp:1592] Ignoring source layer pool6_mbox_loc_flat
I0626 11:37:44.940830 27362 net.cpp:1592] Ignoring source layer pool6_mbox_conf_perm
I0626 11:37:44.940861 27362 net.cpp:1592] Ignoring source layer pool6_mbox_conf_flat
I0626 11:37:44.940865 27362 net.cpp:1592] Ignoring source layer pool6_mbox_priorbox
I0626 11:37:44.940868 27362 net.cpp:1592] Ignoring source layer mbox_loc
I0626 11:37:44.940871 27362 net.cpp:1592] Ignoring source layer mbox_conf
I0626 11:37:44.940876 27362 net.cpp:1592] Ignoring source layer mbox_priorbox
I0626 11:37:44.940878 27362 net.cpp:1592] Ignoring source layer mbox_loss
I0626 11:37:44.944378 27362 net.cpp:1549] reshaping...
I0626 11:37:46.079771 27362 generate_fix8_pt.cpp:834] Output file is model/ssd_float16_dense.fix8.prototxt, iteration: 2
```

图 4-2 量化操作

离线模型生成

SDK 工具中带有生成离线模型的工具，执行如下命令：

```
$ cd common/bin/build.x86_64/caffe/tools/
$ export LD_LIBRARY_PATH=${SDK_ROOT}/common/lib/build.x86_64
$ ./caffe genoff –model/path/ssd_deploy.fix8.prototxt –weights
        /path/ ssd_deploy.caffemodel –mname ssd_fix8 –mcore 1H8
```

其中，-mname 指定生成.cambricon 格式的模型的名称，-mcore 指定模型运行的目标平台。运行成功后生成的离线模型位于 common/bin/build.x86_64/caffe/tools/路径下，如下图所示

```
I0626 11:45:12.824452 27464 net.cpp:1889] bottom 2 shape: 1 2 1176 (2352)
I0626 11:45:12.824457 27464 net.cpp:1889] bottom 3 shape: 1 2 384 (768)
I0626 11:45:12.824461 27464 net.cpp:1889] bottom 4 shape: 1 2 96 (192)
I0626 11:45:12.824466 27464 net.cpp:1889] bottom 5 shape: 1 2 24 (48)
I0626 11:45:12.824472 27464 net.cpp:1891] top 0 shape: 1 2 15792 (31584)
I0626 11:45:12.824478 27464 net.cpp:1887] Layer name: priorbox_concat_reshape
I0626 11:45:12.824483 27464 net.cpp:1889] bottom 0 shape: 1 2 15792 (31584)
I0626 11:45:12.824488 27464 net.cpp:1891] top 0 shape: 1 2 3948 4 (31584)
I0626 11:45:12.824494 27464 net.cpp:1887] Layer name: priorbox_concat_permute
I0626 11:45:12.824499 27464 net.cpp:1889] bottom 0 shape: 1 2 3948 4 (31584)
I0626 11:45:12.824504 27464 net.cpp:1891] top 0 shape: 1 3948 2 4 (31584)
I0626 11:45:12.826004 27464 fusion.cpp:44] [Fusion] reset...
I0626 11:45:12.826038 27464 subnet.hpp:161] subnet[1] fusing...
I0626 11:45:12.826205 27464 fusion.cpp:87] [Fusion] setFusionIO (size: 1, 1)...
I0626 11:45:12.826217 27464 fusion.cpp:74] [Fusion] compiling...0x1acc300
I0626 11:45:17.253916 27464 net.cpp:498] Offline model generated!
********************** Offline model information BEGIN **********************
file name : ssd.cambricon
model name: ssd
model details as follow
[On CPU] subnet[0] layers : 0(data)
[On MLU] [call via func "subnet0"] subnet[1] layers : 1(conv1_1) 2(relu1_1) 3(conv1_
lu3_1) 13(conv3_2) 14(relu3_2) 15(conv3_3) 16(relu3_3) 17(pool3) 18(conv4_1) 19(relu
5_2) 28(relu5_2) 29(conv5_3) 30(relu5_3) 31(pool5) 32(fc6) 33(relu6) 34(fc7) 35(relu
onv7_2) 43(conv7_2_relu) 44(conv8_1) 45(conv8_1_relu) 46(conv8_2) 47(conv8_2_relu) 4
) 53(fc7_mbox_conf) 54(conv6_2_mbox_loc) 55(conv6_2_mbox_conf) 56(conv7_2_mbox_loc)
box_conf) 62(detection_out)
                  func "subnet0"  inputs: data,
                  func "subnet0" outputs: detection_out,
********************** Offline model information END **********************
execution time: 4.4496e+06 us
```

图 4-3 离线模型生成

推理程序开发

在 Hboard 开发板上运行示例，需要基于寒武纪运行时库（Cambricon Neuware Runtime Library，CNRT），程序开发的主要流程如下：

a. 初始化 CNRT 运行库和设备
b. 加载离线模型文件并且提取 Function
c. 获取 Function 的输入和输出数据描述符
d. 在 CPU 内存上为输入和输出数据分配空间，并准备好输入输出数据
e. 在 1H8 上分配数据空间，并将 CPU 内存数据拷入 1H8 内存
f. 启动硬件计算
g. 拷出输出数据
h. 释放资源

示例程序如下:

```
/*
* 1H8 Test sample
*/
int main(int argc, char * argv)
{
    //1. 初始化 CNRT 运行库和设备
    cnrtInit(0);
    unsigned dev_num;
    cnrtGetDeviceCount( & dev_num);
    if (dev_num == 0) return NULL;
    cnrtDev_t dev;
    cnrtGetDeviceHandle( & dev, 0);
    cnrtSetCurrentDevice(dev);

    //2. 从离线模型文件 ssd.cambricon 中加载 Model ，再从 Model 中提取对应的
        Function
    cnrtModel_t model;
    cnrtLoadModel( & model, “ssd.cambricon”);
    cnrtFunction_t function;
    cnrtCreateFunction( &function);
    cnrtExtractFunction( &function, model, “ssd”);

    //3. 获取 Function 的输入和输出数据描述符以及输入、输出的数目
    int inputNum,
    outputNum;
    cnrtDataDescArray_t inputDescS,
    outputDescS;
    cnrtGetInputDataDesc( & inputDescS, &inputNum, function);
    cnrtGetOutputDataDesc( & outputDescS, &outputNum, function);
```

```
//4. 在 CPU 内存上为输入和输出数据分配空间，并准备好输入输出数据
void * *inputCpuPtrS = (void * *) malloc(sizeof(void * ) * inputNum);
void * *outputCpuPtrS = (void * *) malloc(sizeof(void * ) * outputNum);
for (int i = 0; i < inputNum; i++) {
    int dataCount;
    cnrtDataDesc_t dataDesc = &(inputDescS[i]);
    // 为输入数据描述符设置数据格式和维度顺序
    cnrtSetHostDataLayout(dataDesc, CNRT_UINT8, CNRT_NCHW);

    // 获取输入数据量
    cnrtGetHostDataCount(dataDesc, &dataCount);
    // 申请输入数据存放空间，并给输入数据赋值
    float * cpuPtr = (float * ) malloc(sizeof(float) * dataCount);
    for (int j = 0; j < dataCount; j++) {
        cpuPtr[j] = 1;
    }
    inputCpuPtrS[i] = (void * ) cpuPtr;
}

for (int i = 0; i < outputNum; i++) {
    int dataCount;
    cnrtDataDesc_t dataDesc = &(outputDescS[i]);
    // 为输出数据描述符设置数据格式和维度顺序
    cnrtSetHostDataLayout(dataDesc, CNRT_FLOAT32, CNRT_NCHW);
    // 获取输出数据量
    cnrtGetHostDataCount(dataDesc, &dataCount);
    // 申请输出数据存放空间
    float * cpuPtr = (float * ) malloc(sizeof(float) * dataCount);
    outputCpuPtrS[i] = (void * ) cpuPtr;
}

//5. 在 1H8 内存上为输入和输出数据分配空间
void * *input1H8PtrS;
void * *output1H8PtrS;
cnrtMallocByDescArray( & input1H8PtrS, inputDescS, inputNum);
cnrtMallocByDescArray( & output1H8PtrS, outputDescS, outputNum);

//将输入数据从 CPU 内存拷入 1H8 内存
cnrtMemcpyByDescArray(input1H8PtrS, inputCpuPtrS, inputDescS, inputNum,
                      CNRT_MEM_TRANS_DIR_HOST2DEV);

//准备 Function 的参数（ 1H8 内存上的输入、输出数据地址），创建 Stream
void * param[2] = {
```

```
            input1H8PtrS[0],
            output1H8PtrS[0]
        };
        cnrtDim3_t dim = {
            1,
            1,
            1
        };
        cnrtStream_t stream;
        cnrtCreateStream( & stream);

        //6. 启动硬件计算
        cnrtInvokeFunction(function, dim, param, CNRT_FUNC_TYPE_BLOCK, stream, NULL);
        cnrtSyncStream(stream);

        //7. 将计算结果从 1H8 内存拷出到 CPU 内存
        cnrtMemcpyByDescArray(outputCpuPtrS, output1H8PtrS, outputDescS, outputNum,
                              CNRT_MEM_TRANS_DIR_DEV2HOST);
        //8. 释放资源
        for (int i = 0; i < inputNum; i++) {
            free(inputCpuPtrS[i]);
        }

        for (int i = 0; i < outputNum; i++) {
            free(outputCpuPtrS[i]);
        }

        free(inputCpuPtrS);
        free(outputCpuPtrS);
        cnrtFreeArray(input1H8PtrS, inputNum);
        cnrtFreeArray(output1H8PtrS, outputNum);
        cnrtDestroyStream(stream);
        cnrtDestroyFunction(function);
        cnrtUnloadModel(model);
        cnrtDestroy();

        return 0;
}
```

程序交叉编译

开发好的推理程序要运行在 Hboard 开发板上，需要使用开发板对应的交叉编译器对程序进行编译后才能拷贝到 HBoard 开发板上运行。

交叉编译器配置

宿主机基于 Ubuntu 16.04 64 位操作系统，开发环境的配置主要分为编译时需要的工具、依赖库的安装和交叉编译器的配置。

1) 需要安装的工具、依赖库如下:

```
$ sudo apt-get update
$ sudo apt-get install –y cmake lib32z1-dev
```

注: 安装 lib32z1-dev 是为了在宿主机(64 位系统)上兼容 32 位依赖库及其工具。

2) 配置交叉编译器

```
$ vi ~/.bashrc
```

在最后一行添加如下内容:

```
export PAHT=$PATH:${SDK}/common/toolchain/arm-linux-gnueabihf-4.8.3-201404/bin
```

其中${SDK}要修改成自己宿主机中 sdk 存放的根目录的绝对路径。

修改后 wq 保存退出，执行如下命令使其生效:

```
$ source ~/.bashrc
```

在宿主机终端上输入命令：arm-linux-gnueabihf-gcc –v, 出现 gcc version xxx 表明交叉编译器已配置成功，如图 4-4 所示

```
zhou@ubuntu:~$
zhou@ubuntu:~$ vi ~/.bashrc
zhou@ubuntu:~$ source ~/.bashrc
zhou@ubuntu:~$
zhou@ubuntu:~$
zhou@ubuntu:~$ arm-linux-gnueabihf-gcc -v
Using built-in specs.
COLLECT_GCC=arm-linux-gnueabihf-gcc
COLLECT_LTO_WRAPPER=/home/zhou/work/1h8/Mango_Devkit/common/toolchain/gcc-linaro-6.3.1-2017.05-x86_64_arm-linux-gnueabi
hf/bin/../libexec/gcc/arm-linux-gnueabihf/6.3.1/lto-wrapper
Target: arm-linux-gnueabihf
Configured with: '/home/tcwg-buildslave/workspace/tcwg-make-release/builder_arch/amd64/label/tcwg-x86_64-build/target/a
rm-linux-gnueabihf/snapshots/gcc.git~linaro-6.3-2017.05/configure' SHELL=/bin/bash --with-mpc=/home/tcwg-buildslave/wor
kspace/tcwg-make-release/builder_arch/amd64/label/tcwg-x86_64-build/target/arm-linux-gnueabihf/_build/builds/destdir/x8
6_64-unknown-linux-gnu --with-mpfr=/home/tcwg-buildslave/workspace/tcwg-make-release/builder_arch/amd64/label/tcwg-x86_
64-build/target/arm-linux-gnueabihf/_build/builds/destdir/x86_64-unknown-linux-gnu --with-gmp=/home/tcwg-buildslave/wor
kspace/tcwg-make-release/builder_arch/amd64/label/tcwg-x86_64-build/target/arm-linux-gnueabihf/_build/builds/destdir/x8
6_64-unknown-linux-gnu --with-gnu-as --with-gnu-ld --disable-libmudflap --enable-lto --enable-shared --without-included
-gettext --enable-nls --disable-sjlj-exceptions --enable-gnu-unique-object --enable-linker-build-id --disable-libstdcxx
-pch --enable-c99 --enable-clocale=gnu --enable-libstdcxx-debug --enable-long-long --with-cloog=no --with-ppl=no --with
-isl=no --disable-multilib --with-float=hard --with-fpu=vfpv3-d16 --with-mode=thumb --with-tune=cortex-a9 --with-arch=a
rmv7-a --enable-threads=posix --enable-multiarch --enable-libstdcxx-time=yes --enable-gnu-indirect-function --with-buil
d-sysroot=/home/tcwg-buildslave/workspace/tcwg-make-release/builder_arch/amd64/label/tcwg-x86_64-build/target/arm-linux
-gnueabihf/_build/sysroots/arm-linux-gnueabihf --with-sysroot=/home/tcwg-buildslave/workspace/tcwg-make-release/builder
_arch/amd64/label/tcwg-x86_64-build/target/arm-linux-gnueabihf/_build/builds/destdir/x86_64-unknown-linux-gnu/arm-linux
-gnueabihf/libc --enable-checking=release --disable-bootstrap --enable-languages=c,c++,fortran,lto --build=x86_64-unkno
wn-linux-gnu --host=x86_64-unknown-linux-gnu --target=arm-linux-gnueabihf --prefix=/home/tcwg-buildslave/workspace/tcwg
-make-release/builder_arch/amd64/label/tcwg-x86_64-build/target/arm-linux-gnueabihf/_build/builds/destdir/x86_64-unknow
n-linux-gnu
Thread model: posix
gcc version 6.3.1 20170404 (Linaro GCC 6.3-2017.05)
zhou@ubuntu:~$
```

图 4-4 配置交叉编译器

交叉编译

由于程序需要用到系统的依赖库并配置一些环境变量，所以推荐使用 SDK 自带的基于 cmake 目录结构模式进行交叉编译。

1) 将自己的源码拷贝到编译目录下:

```
$ cp demo_offline.cpp HCAI1H_Devkit_Realease_V7.9/HCAI1H_SDK/app/offline_test/
```

2) 修改 CMakeLists.txt，增加自己程序(demo_offline)的编译项

```
$ vi HCAI1H_Devkit_Realease_V7.9/HCAI1H_SDK/app/CMakeLists.txt
```

在第 35 行后面增加一行:

add_executable(demo_offline_fix8 offline_test/demo_offline.cpp)

```
ADD_EXECUTABLE(cnmlTest ${TEST_SRC})
target_link_libraries(cnmlTest cnml cnrt)

add_executable(runtime_test ${RUNTIME_TEST_SRC})
target_link_libraries(runtime_test cnrt stdc++)


# Add the offline_test test cases
include_directories("${OpenCV_DIR}/include")
link_directories("${OpenCV_DIR}/lib")

add_executable(offline_test_fix8 offline_test/classfication_cnrt_offline_fix8.cpp)
add_executable(offline_BGRA_test_fix8 offline_test/classification_offline_bgra.cpp)
add_executable(test_forward offline_test/test_forward_offline.cpp)
add_executable(faster-rcnn_offline_fix8 offline_test/faster-rcnn_offline.cpp)
add_executable(pvanet_offline_fix8 offline_test/pvanet_offline.cpp)
add_executable(yolov2_offline_fix8 offline_test/yolov2_offline.cpp)
add_executable(ssd_offline_fix8 offline_test/ssd_offline.cpp)
add_executable(rfcn_offline_fix8 offline_test/rfcn_offline_fix8.cpp)
add_executable(ssh_offline_fix8 offline_test/ssh_offline_fix8.cpp)
add_executable(facenet_offline_fix8 offline_test/facenet_offline.cpp)
add_executable(label_image_offline_fix8 offline_test/label_image_offline.cpp)
add_executable(demo_offline_fix8 offline_test/demo_offline.cpp)

target_link_libraries(offline_test_fix8 opencv_imgcodecs opencv_imgproc opencv_core opencv_highgui pthread stdc++ cnrt)
target_link_libraries(offline_BGRA_test_fix8 opencv_imgcodecs opencv_imgproc opencv_core opencv_highgui pthread stdc++ cnrt)
target_link_libraries(test_forward pthread stdc++ cnrt)
```

图 4-5 增加 demo 程序编译项

在第 48 行后面增加一行:

target_link_libraries(demo_offline_fix8 opencv_imgcodecs opencv_imgproc opencv_core opencv_highgui opencv_videoio glog gflags protobuf pthread stdc++ cnrt)

```
add_executable(offline_test_fix8 offline_test/classfication_cnrt_offline_fix8.cpp)
add_executable(offline_BGRA_test_fix8 offline_test/classification_offline_bgra.cpp)
add_executable(test_forward offline_test/test_forward_offline.cpp)
add_executable(faster-rcnn_offline_fix8 offline_test/faster-rcnn_offline.cpp)
add_executable(pvanet_offline_fix8 offline_test/pvanet_offline.cpp)
add_executable(yolov2_offline_fix8 offline_test/yolov2_offline.cpp)
add_executable(ssd_offline_fix8 offline_test/ssd_offline.cpp)
add_executable(rfcn_offline_fix8 offline_test/rfcn_offline_fix8.cpp)
add_executable(ssh_offline_fix8 offline_test/ssh_offline_fix8.cpp)
add_executable(facenet_offline_fix8 offline_test/facenet_offline.cpp)
add_executable(label_image_offline_fix8 offline_test/label_image_offline.cpp)
add_executable(demo_offline_fix8 offline_test/demo_offline.cpp)

target_link_libraries(offline_test_fix8 opencv_imgcodecs opencv_imgproc opencv_core opencv_highgui pthread stdc++ cnrt)
target_link_libraries(offline_BGRA_test_fix8 opencv_imgcodecs opencv_imgproc opencv_core opencv_highgui pthread stdc++ cnrt)
target_link_libraries(test_forward pthread stdc++ cnrt)
target_link_libraries(faster-rcnn_offline_fix8 opencv_imgcodecs opencv_imgproc opencv_core opencv_highgui pthread stdc++ cnrt)
target_link_libraries(pvanet_offline_fix8 opencv_imgcodecs opencv_imgproc opencv_core opencv_highgui pthread stdc++ cnrt)
target_link_libraries(ssd_offline_fix8 opencv_imgcodecs opencv_imgproc opencv_core opencv_highgui opencv_videoio glog gflags protobuf pthread stdc++ cnrt)
target_link_libraries(yolov2_offline_fix8 opencv_imgcodecs opencv_imgproc opencv_core opencv_highgui protobuf pthread stdc++ cnrt)
target_link_libraries(rfcn_offline_fix8 opencv_imgcodecs opencv_imgproc opencv_core opencv_highgui opencv_videoio glog gflags protobuf pthread stdc++ cnrt)
target_link_libraries(ssh_offline_fix8 opencv_imgcodecs opencv_imgproc opencv_core opencv_highgui opencv_videoio glog gflags protobuf pthread stdc++ cnrt)
target_link_libraries(facenet_offline_fix8 opencv_imgcodecs opencv_imgproc opencv_core opencv_highgui opencv_videoio glog gflags protobuf pthread stdc++ cnrt)
target_link_libraries(label_image_offline_fix8 opencv_imgcodecs opencv_imgproc opencv_core opencv_highgui opencv_videoio glog gflags protobuf pthread stdc++ cnrt)
target_link_libraries(demo_offline_fix8 opencv_imgcodecs opencv_imgproc opencv_core opencv_highgui opencv_videoio glog gflags protobuf pthread stdc++ cnrt)
```

图 4-6 增加 demo 程序链接项

3) 编译

$ cd HCAI1H_Devkit_Realease_V7.9/

$ ./build_app_test.sh

出现编译进度 100%表明编译成功，能看到 demo_offline_fix8 已完成编译，如下图

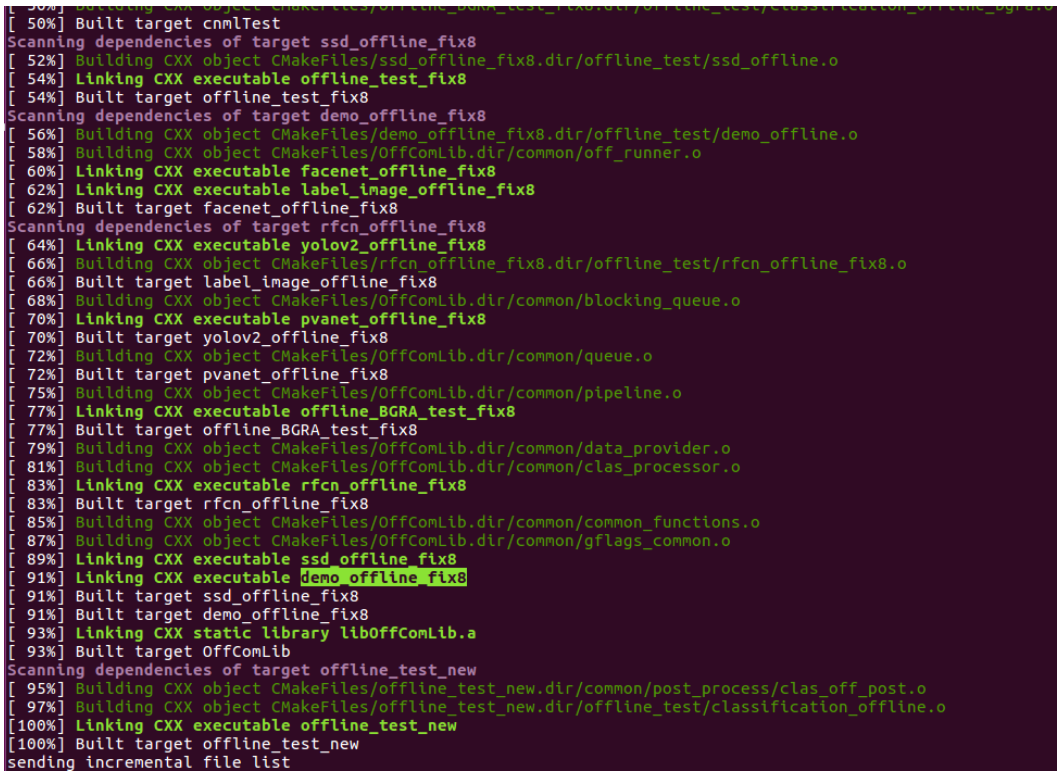


图 4-7 demo 程序编译过程

可以在 HCAI1H_Devkit_Realease_V7.9/HCAI1H_SDK/app/app/目录下查看已经成功生成了我们自己的 demo 程序，如下图:

```
zhou@ubuntu:~/work/1h8$
zhou@ubuntu:~/work/1h8$ ls HCAI1H_Devkit_Realease_V7.9/HCAI1H_SDK/app/app/
cnmlTest                label_image_offline_fix8  offline_test_new     ssd_offline_fix8
demo_offline_fix8       libOffComLib.a            pvanet_offline_fix8  ssh_offline_fix8
facenet_offline_fix8    offline_BGRA_test_fix8    rfcn_offline_fix8    test_forward
faster-rcnn_offline_fix8  offline_test_fix8       runtime_test         yolov2_offline_fix8
zhou@ubuntu:~/work/1h8$
```

图 4-8 编译好的 demo 程序

示例程序运行

此部分介绍在登录到开发板后，如何运行一个示例程序的具体方法与步骤。

NFS 安装与配置

为了避免频繁地通过 SD 卡拷贝文件到开发板上，减少操作的复杂性，我们采用 NFS 的方式将整个 SDK 挂载到开发板某个目录下来进行操作。在使用 NFS 之前需要在宿主机上先进行安装与配置，宿主机为 Linux 操作系统，以 Ubuntu16.04 为例，操作方法如下：

1. 在宿主机安装 NFS Server

```
$ sudo apt-get install nfs-kernel-server
```

2. 配置 NFS Server 根目录

修改/etc/exports 文件，增加如下内容，其中 SERVER_ROOT 为 NFS Server 的根路径

```
${SERVER_ROOT} *(rw,sync,no_root_squash,no_all_squash)
```

3. 启动 NFS Server

```
$ sudo /etc/init.d/nfs-kernel-server restart
```

关闭后台程序

系统启动时会默认在后台运行一个示例程序（sample_fr_fast），该程序会占用 IPU 资源，如需要运行其他使用 IPU 的程序，需要先关闭该程序，以防资源冲突。关闭后台程序有两种方法，一种是手动 kill 掉后台程序，另外一种是修改启动脚本。

1) 手动 kill 程序

执行 ps 命令查看进程 ID，然后通过 kill 命令将其杀掉，具体操作如下图所示:

```
#ps | grep sample
#kill -9 290              //290 为具体进程 ID，以查询结果为准。
```

```
/ #
/ # ps | grep sample
  290 root       0:28 ./sample_fr_fast
  335 root       0:00 grep sample
/ # kill -9 290
/ #
```

图 4-9 kill 后台进程

2) 修改启动脚本

由于手动 kill 程序是一次性的，如果经常要运行离线模型程序，那么修改启动脚本就可以一劳永逸。

```
#vi /customer/run.sh
```

在后台程序前面增加#，屏蔽启动，保存退出重启开发板。

```
#!/bin/sh

ifconfig eth0 172.19.24.241

## libcnrt..............
## export ENABLE_CNRT_PRINT_INFO=true

## CSpotter...license......
date -s "2019-01-01 12:00"

## onvif............
route add -net 239.255.255.250 netmask 255.255.255.255 eth0

## ......core......
ulimit -c unlimited
echo $PWD"/core-%e-%p-%t" > /proc/sys/kernel/core_pattern

export LD_LIBRARY_PATH=/lib:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH=$PWD:$LD_LIBRARY_PATH

cd $PWD;
#./sample_fr_fast &
~
```

4-10 屏蔽启动脚本

程序运行

以 SDK 包中提供的 ssd_offline_fix8 离线应用程序为例，执行如下命令

#mount -t nfs -o nolock -o tcp 172.19.24.200:${HOST_SDK_PATH} /mnt

此命令是将宿主机上的 HOST_SDK_PAT 路径挂载到开发板/mnt 目录下。其中，HOST_SDK_PATH 必须为 NFS 服务器中 SERVER_ROOT 的一个子目录或者为同一目录。

注：172.19.24.200 为宿主机 IP 地址，可根据实际情况修改

#cd /mnt/HCAI1H_Test/caffe_offline/ssd

#./run.sh

```
/ #
/ # cd /mnt/HCAI1H_Test/caffe_offline/ssd/
/mnt/HCAI1H_Test/caffe_offline/ssd # ls
jpg                 run.sh              ssd_image_file_list
/mnt/HCAI1H_Test/caffe_offline/ssd # ./run.sh
Prepare to run the ssd.cambricon offline model test!!
Run the ssd.cambricon offline model test!!
The test command line is: ../app/ssd_offline_fix8 -offlinemodel ../offline_models/ssd/ssd.cambricon -output_dir ./ -fil
e_list ./ssd_image_file_list -confidence_threshold 0.6
[cnrtInfo]:cnrtInit SUCCESS!
[cnrtInfo]:MLU device open success
[cnrtInfo]:Set Device descriptor: 513008.
[cnrtInfo]:Set Device ordinal: 0.
load file: ../offline_models/ssd/ssd.cambricon
[cnrtInfo]:static memory used: 37305888 Bytes
[cnrtInfo]:stack memory used: 14832384 Bytes
[cnrtInfo]:max memory used: 52756480 Bytes
[cnrtInfo]:max memory used: 52756480 Bytes
input blob num is 1
shape 1
shape 3
shape 300
shape 300
output shape 1
output shape 200
output shape 1
output shape 6
there are 1 figures in ./ssd_image_file_list
326----------ipuMemcpy: copy input----------
[cnrtInfo]:hardware time: 192335.000000 us
Forward execution time: 249692 us
343----------ipuMemcpy----------
./ssd_001763.jpg 8 0.983887 283 119 494 384
./ssd_001763.jpg 12 0.907715 2 20 328 365
ssd_detection() execution time: 536292 us
[cnrtInfo]:Cnrt total warning count: 0
[cnrtInfo]:Cnrt total error count: 0
[cnrtInfo]:Cnrt mlu device destruction has been completed
End the ssd.fix8 offline model test Done!!
/mnt/HCAI1H_Test/caffe_offline/ssd #
```

图 4-11 ssd_offline_fix8 应用程序执行过程

注：如果运行时出现“[cnrtError]: No MLU can be found! No device found”这样的提示信息，表示 IPU 资源被占用，请执行上一节中的操作，关闭后台程序。

图 4-12 ssd_offline_fix8 处理前后对比

ssd_offline_fix8 应用程序处理完成后的结果如图 5 所示，可以识别出图片中左侧目标为狗的概率为 0.91，右侧目标为猫的概率为 0.98。