

# RK182X RKNN3 SDK 快速上手指南

文件标识：RK-JC-YF-416

发布版本：V1.0.0

日期：2026-1-1

文件密级：绝密 秘密 内部资料 公开

## 免责声明

本文档按“现状”提供，瑞芯微电子股份有限公司（“本公司”，下同）不对本文档的任何陈述、信息和内容的准确性、可靠性、完整性、适销性、特定目的性和非侵权性提供任何明示或暗示的声明或保证。本文档仅作为使用指导的参考。

由于产品版本升级或其他原因，本文档将可能在未经任何通知的情况下，不定期进行更新或修改。

## 商标声明

“Rockchip”、“瑞芯微”、“瑞芯”均为本公司的注册商标，归本公司所有。

本文档可能提及的其他所有注册商标或商标，由其各自拥有者所有。

## 版权所有 © 2025 瑞芯微电子股份有限公司

超越合理使用范畴，非经本公司书面许可，任何单位和个人不得擅自摘抄、复制本文档内容的部分或全部，并不得以任何形式传播。

瑞芯微电子股份有限公司

Rockchip Electronics Co., Ltd.

地址：福建省福州市铜盘路软件园A区18号

网址：[www.rock-chips.com](http://www.rock-chips.com)

客户服务电话：+86-4007-700-590

客户服务传真：+86-591-83951833

客户服务邮箱：[fae@rock-chips.com](mailto:fae@rock-chips.com)

## 前言

### 概述

本文档面向零基础用户，详细介绍如何使用RKNN3 Toolkit在PC端完成AI模型转换，并部署到搭载RK1820/1828协处理器的Rockchip开发板上。

### 读者对象

本文档主要适用于以下工程师：

- 技术支持工程师
- 软件开发工程师

### 修订记录

版本	修改人	修改日期	修改说明	核定人
V0.1.0	HPC	2025-05-13	初始版本	熊伟
V0.2.0	HPC	2025-08-15	1. 调整模型加载接口说明 2. 新增模型初始化接口说明 3. 调整模型部署流程图 4. 增加准备开发板 5. 增加环境准备 6. 增加运行示例程序 7. 修改开发流程 8. 增加常见问题	熊伟
V0.3.0	HPC	2025-09-10	更新 LLM 会话相关接口改动	熊伟
V0.3.0b0	HPC	2025-09-12	1. 修改文档中的一些错误 2. 增加附图 3. 增加OpenAI接口使用说明	熊伟
V0.4.0b0	HPC	2025-10-18	1. 增加RK1828的平台	熊伟
V0.5.0	HPC	2025-11-29	1. 增加同步输入、输出数据	熊伟
V1.0.0	HPC	2026-01-01	1. 更新rkllm3-server用法 2. 更新支持的模型框架 3. 更新模型评估方法	熊伟

# 目录

## RK182X RKNN3 SDK 快速上手指南

### 1 概述

1.1 RKNN3 SDK整体软件框架

1.2 支持的硬件平台

1.3 支持的操作系统

### 2 硬件环境准备

2.1 硬件清单

2.2 开发板和连接工具介绍

2.3 连接开发板

### 3 开发环境准备

3.1 下载RKNN3相关仓库

3.2 安装 rknn3-toolkit 环境

3.2.1 安装Python

3.2.1.1 安装Miniforge Conda

3.2.1.2 使用Miniforge Conda创建Python环境

3.2.2 安装rknn3-toolkit

3.2.3 验证安装

3.3 安装编译工具

3.3.1 安装CMake

3.3.2 安装编译器

3.3.2.1 确认开发板系统类型和架构

3.3.2.2 Android系统开发板安装NDK

3.3.2.3 Linux系统开发板安装GCC交叉编译器

3.4 运行示例程序

3.4.1 RKNN3 Model Zoo介绍

3.4.2 安装板端RKNPU3环境

3.4.3 运行常规模型

3.4.4 运行LLM

### 4 开发流程

4.1 CNN模型开发流程介绍

4.1.1 模型转换

4.1.1.1 模型转换流程

4.1.1.2 关键接口说明

4.1.1.2.1 创建、释放RKNN对象

4.1.1.2.2 模型配置

4.1.1.2.3 模型加载

4.1.1.2.4 模型构建

4.1.1.2.5 模型导出

4.1.2 模型评估

4.1.2.1 精度分析

4.1.2.2 性能评估

4.1.2.4 内存评估

4.1.3 模型部署

4.1.4 关键接口说明

4.1.4.1 上下文初始化和销毁

4.1.4.2 模型加载

4.1.4.3 模型初始化

4.1.4.4 模型属性查询

4.1.4.5 同步输入、输出数据

4.1.4.6 模型推理

## 4.2 LLM 模型开发流程介绍

- 4.2.1 模型转换
- 4.2.2 LLM模型部署
- 4.2.3 关键接口说明
  - 4.2.3.1 会话创建和销毁
  - 4.2.3.2 设置聊天模板
  - 4.2.3.3 注册回调函数
  - 4.2.3.4 会话推理
  - 4.2.3.5 终止会话
  - 4.2.3.6 清理 KV Cache

## 4.2.4 LLM Vision模型开发流程介绍

## 5 RKLLM3 Server

- 5.1 使用方法
- 5.2 快速上手
- 5.3 用CURL进行测试
- 5.4 API 端点
  - 5.4.1 GET `/health`: 返回健康检查结果
  - 5.5 OpenAI兼容的API端点
    - 5.5.1 GET `/v1/models`: OpenAI兼容的模型信息API
    - 5.5.2 POST `/v1/chat/completions`: OpenAI兼容的聊天补全API
    - 5.5.3 Function to encode the image
    - 5.5.4 Getting the Base64 string
  - 5.6 更多示例
    - 5.6.1 OAI-like API
    - 5.6.2 API 错误

## 6 常见问题

- 6.1 命令 adb devices 查看不到设备
- 6.2 rknn3\_find\_devices查不到设备
- 6.3 rknn3-model-zoo转换模型时找不到'py\_utils'
- 6.4 rknn3-model-zoo GRQ量化失败

## 7. 参考资料

- 7.1 模型转换
- 7.2 模型部署

# 1 概述

本文档介绍RKNN3 SDK及如何使用RKNN3 SDK开发并部署AI模型到RK1820/1828协处理器。RKNN3 SDK提供了完整的AI模型部署解决方案，包括PC端开发套件（RKNN3 Toolkit）、板端运行时库（RKNN3 Runtime）以及模型转换部署示例（RKNN3 Model Zoo）。本SDK支持RK1820/1828协处理器模式，即主控SoC通过PCIe/USB高速接口连接RK1820/1828协处理器。

- **主控SoC**: 作为系统核心，负责任务调度、资源分配和整体控制
- **RK1820/1828协处理器**: 作为AI计算加速单元，专注于高性能神经网络推理任务
- **PCIe/USB高速接口**: 实现主控与协处理器之间的低延迟、高带宽数据交互

## 1.1 RKNN3 SDK整体软件框架

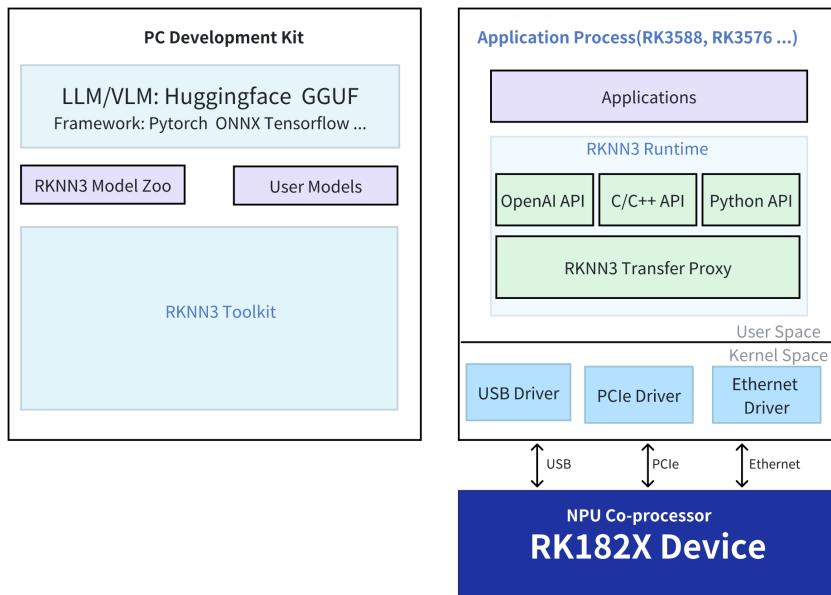


图1-1 RKNN3 SDK框图

软件框架主要包含两个核心组件：

### 1. PC端开发套件

在PC端，用户可通过RKNN3 Toolkit将PyTorch、ONNX等深度学习框架训练的模型转换为RKNN格式。RKNN3 Model Zoo提供了丰富的模型转换示例，涵盖多种AI模型类型：

- **CNN模型**: MobileNetV2、ResNet50、YOLOv5、YOLOv6、YOLOv8等
- **LLM模型**: Qwen 2.5 0.5B、1.5B、3B等
- **VLM模型**: FastVLM、Qwen2.5 VL、Qwen3 VL等

### 2. 开发板运行环境

模型转换完成后，可在开发板上使用RKNN3 API加载和运行RKNN模型。除RKNN3 API外，还支持OpenAI兼容API调用LLM模型。主要包含以下模块：

#### RKNN3 API

提供RKNN模型加载、推理、LLM模型推理及会话管理等核心功能。文件结构如下：

```
rknn3-api
├── include
│   └── float16.h
│   └── rknn3_api.h
├── Android
│   └── arm64-v8a
│       └── librknn3_api.so
└── Linux
    └── aarch64
        └── librknn3_api.so
```

开发时主要链接librknn3\_api.so库文件。

### **rkllm3-server**

提供OpenAI兼容API服务，支持文本和图片输入，暂不支持语音和视频输入。文件结构如下：

```
rkllm3-server
├── bin
│   ├── android_arm64-v8a
│   │   └── rkllm3-server
│   └── linux-aarch64
│       └── rkllm3-server
```

### **rknn3\_transfer\_proxy**

提供Host端（如RK3588、RK3576）与RK1820/1828协处理器间的通信接口，支持PCIe和USB连接。文件结构如下：

```
rknn3_transfer_proxy
├── android-arm64-v8a
│   └── rknn3_transfer_proxy
└── linux-aarch64
    └── rknn3_transfer_proxy
```

## **1.2 支持的硬件平台**

- RK3588/RK3576 + RK1820/1828协处理器

## **1.3 支持的操作系统**

- Android/Linux

## 2 硬件环境准备

本章介绍硬件环境准备，内容包括：

- 硬件清单
- 开发板和连接工具介绍
- 开发板连接方法

### 2.1 硬件清单

运行本文档示例程序需要准备以下硬件：

- 计算机 × 1
- RK1820/1828模组 × 1
- RK\_EVB10\_RK3588\_V10开发板 × 1
- USB-C数据线 × 1
- RK3588电源适配器 × 1

注意：以下章节中RK1820/1828模组简称RK1820/1828，RK\_EVB10\_RK3588\_V10开发板简称RK3588。

### 2.2 开发板和连接工具介绍

#### 1. RK1820/1828模组



图2-1 RK1820/1828模组

#### 2. RK\_EVB10\_RK3588\_V10开发板



图2-2 RK\_EVB10\_RK3588\_V10开发板

### 3. 连接开发板和计算机的数据线



图2-3 USB-C 数据线

### 4. 电源适配器



图2-4 RK3588电源适配器

## 2.3 连接开发板

下面以RK\_EVB10\_RK3588\_V10开发板搭载RK1820/1828为例说明如何快速开发：

1. 准备一台操作系统为 Ubuntu20.04 / Ubuntu22.04 的计算机。
2. 将RK1820/1828模组插入到RK\_EVB10\_RK3588\_V10开发板上，如下图所示：



图2-5 RK\_EVB10\_RK3588\_V10开发板搭载RK1820/1828

3. 将RK1820/1828/RK3588开发板上标识 debug 的端口通过数据线与计算机相连。（注意，由于硬件版本不同，开发板的数据线接口类型和位置可能会发生变化。）
4. 打开电源开关，等待开发板系统启动完成。
5. 查看开发板是否连接至计算机
  - 1) 检查RK3588是否连接成功

在计算机的终端窗口（命令行界面）中，执行以下命令：

```
# 安装 adb
sudo apt install adb

# 查询adb连接的设备
adb devices

# 连接成功时输出信息如下，其中13af7b28115662cd 为 RK3588 的设备 ID。
# 若无设备显示请参考第 5.1 章节进行排查。
List of devices attached
13af7b28115662cd device
```

- 2) 检查RK1820/1828是否连接成功

在计算机的终端窗口（命令行界面）中，执行以下命令：

```
# 进入RK3588终端
```

```
adb shell

# 若RK3588为安卓系统，进入rknn3_transfer_proxy安装路径:/vendor/bin
cd /vendor/bin

# 若RK3588为linux系统， rknn3_transfer_proxy安装路径:/usr/bin
cd /usr/bin

# 查询设备
./rknn3_transfer_proxy devices

# 参考输出如下
List of ntb devices attached
0000:01:00.0          b98e6c51      PCIE
```

注意：若找不到rknn3\_transfer\_proxy，请参考[3.4.2 安装板端RKNPU3环境](#)。

# 3 开发环境准备

本章介绍PC端开发环境安装配置，内容包括：

- 下载RKNN3相关仓库
- 安装rknn3-toolkit环境
- 安装编译工具
- 运行示例程序

## 3.1 下载RKNN3相关仓库

建议创建专门目录存放RKNN3仓库，例如创建名为 Projects 的文件夹，并将RKNN3相关仓库下载到该目录下。下载方式有SDK和Github两个种方式，以下重点介绍Github下载方式。

**注意：SDK和Github两个种下载方式的目录结构可能不一样。**

需要下载的仓库包括 rknn3-toolkit、 rknn3-model-zoo，参考命令如下：

```
# 新建 Projects 文件夹
mkdir Projects

# 进入该目录
cd Projects

# 下载 rknn3-toolkit 仓库
git clone https://github.com/airockchip/rknn3-toolkit

# 下载 rknn3-model-zoo 仓库
git clone https://github.com/airockchip/rknn3-model-zoo

# 整体目录结构如下:
Projects
├── rknn3-model-zoo
│   ├── 3rdparty
│   └── examples
└── rknn3-toolkit
    ├── doc
    ├── rknn3-toolkit
    ├── rknn3-toolkit-lite
    └── rknn3-runtime
        ├── rknn3-api
        ├── rknn3_transfer_proxy
        └── rkllm3-server
    └── ...

```

## 3.2 安装 rknn3-toolkit 环境

### 3.2.1 安装Python

如果系统中未安装**Python 3.10（推荐版本）**，或存在多个Python版本，建议使用Miniforge Conda创建独立的Python 3.10环境。

#### 3.2.1.1 安装Miniforge Conda

在终端中执行以下命令检查是否已安装Miniforge Conda，若已安装可跳过此步骤。

```
conda -v  
# 输出示例：conda 23.3.1，表示Miniforge conda版本为23.3.1  
# 如果提示 conda: command not found，则表示未安装Miniforge
```

如果未安装Miniforge Conda，可通过以下命令下载安装包：

```
wget -c https://github.com/conda-forge/miniforge/releases/download/25.3.0-1/Miniforge3-  
25.3.0-1-Linux-x86_64.sh
```

然后执行以下命令安装Miniforge Conda：

```
# 拷贝到用户目录  
cp /path/to/Miniforge3-25.3.0-1-Linux-x86_64.sh ~  
  
# 增加运行权限  
chmod 777 Miniforge3-25.3.0-1-Linux-x86_64.sh  
  
# 运行安装脚本  
bash Miniforge3-25.3.0-1-Linux-x86_64.sh
```

#### 3.2.1.2 使用Miniforge Conda创建Python环境

在终端中执行以下命令进入Conda base环境：

```
# 刷新环境变量  
source ~/.bashrc  
  
# 成功后，命令行提示符会变为：  
# (base) xxx@xxx:~$
```

执行以下命令创建名为toolkit3的Python 3.10环境：

```
conda create -n toolkit3 python=3.10
```

激活toolkit3环境，后续将在此环境中安装rknn3-toolkit：

```
conda activate toolkit3  
# 成功后，命令行提示符会变为：  
# (toolkit3) xxx@xxx:~$
```

**注意：**后续Python运行命令默认在toolkit3环境中执行。

### 3.2.2 安装rknn3-toolkit

激活toolkit3环境后，通过本地wheel包安装rknn3-toolkit：

```
# 进入 rknn3-toolkit 目录
cd Projects/rknn3-toolkit/rknn3-toolkit/packages

# 根据Python版本和处理器架构选择对应的requirements文件
# cpxxx为Python版本号，x.x.x为rknn3-toolkit版本号
pip install -r requirements_cpxxx-x.x.x.txt

# 安装rknn3-toolkit
# 根据Python版本和处理器架构选择对应的wheel安装包
# x.x.x为rknn3-toolkit版本号，cpxx为Python版本号
pip install rknn3_toolkit-x.x.x-cpxx-cpxx-manylinux_2_17_x86_64.manylinux2014_x86_64.whl
```

### 3.2.3 验证安装

在终端中执行以下命令验证rknn3-toolkit环境是否安装成功，若无报错则表示安装成功：

```
# 进入 Python 交互模式
python

# 导入 RKNN 类
from rknn.api import RKNN
```

## 3.3 安装编译工具

### 3.3.1 安装CMake

在终端中执行以下命令：

```
# 更新包列表
sudo apt update

# 安装 cmake
sudo apt install cmake
```

### 3.3.2 安装编译器

#### 3.3.2.1 确认开发板系统类型和架构

为便于描述，后续文档使用"板端"表示开发板端。

##### 1. 确认板端系统类型

在终端中执行以下命令：

```
adb shell getprop ro.build.version.release
```

如果输出为数字（Android系统版本号），则表示板端为Android系统：

```
adb shell getprop ro.build.version.release  
# 输出  
12
```

否则板端为Linux系统：

```
adb shell getprop ro.build.version.release  
# 输出  
/bin/bash: line 1: getprop: command not found
```

## 2. 确认板端系统架构

如果板端为Android系统，可在PC端执行以下命令查询系统架构：

```
adb shell getprop ro.product.cpu.abi
```

该命令输出示例，其中arm64-v8a表示ARM 64位架构、第八版本ABI。

```
adb shell getprop ro.product.cpu.abi  
# 输出  
arm64-v8a
```

如果板端为Linux系统，可在PC端执行以下命令查询系统架构：

```
adb shell uname -a
```

该命令输出示例，其中aarch64表示ARM 64位架构。

```
adb shell uname -a  
# 输出  
Linux rk3588-buildroot 6.1.118 #4 SMP Sat Jul 5 14:44:53 CST 2025 aarch64 GNU/Linux
```

### 3.3.2.2 Android系统开发板安装NDK

**注意：**本节适用于Android系统开发板，如果板端为Linux系统，请跳过此节。

- **NDK下载地址**（建议下载r19c版本）：[https://dl.google.com/android/repository/android-ndk-r19c-linux-x86\\_64.zip](https://dl.google.com/android/repository/android-ndk-r19c-linux-x86_64.zip)
- **解压软件包**

建议将NDK软件包解压到Projects文件夹中，位置如下：

```
Projects
├── rknn3-toolkit
├── rknn3-model-zoo
└── android-ndk-r19c # 此路径在后续编译RKNN3 C Demo时会用到
```

此时NDK编译器路径为Projects/android-ndk-r19c

### 3.3.2.3 Linux系统开发板安装GCC交叉编译器

**注意：**本节适用于Linux系统开发板，如果板端为Android系统，请跳过此节。

应用程序在主控上运行，根据主控SoC选择交叉编译工具。以RK3588 Linux系统为例：

- 交叉编译工具下载地址

```
https://releases.linaro.org/components/toolchain/binaries/6.3-2017.05/aarch64-linux-gnu/gcc-linaro-6.3.1-2017.05-x86\_64\_aarch64-linux-gnu.tar.xz
```

- 解压软件包

建议将GCC软件包解压到Projects文件夹中。以板端为64位系统的GCC软件包为例，存放位置如下：

```
Projects
├── rknn3-toolkit
├── rknn3-model-zoo
└── gcc-linaro-6.3.1-2017.05-x86_64_aarch64-linux-gnu # 此路径在后续编译RKNN3 C Demo时会用到
```

此时GCC编译器路径为 Projects/gcc-linaro-6.3.1-2017.05-x86\_64\_aarch64-linux-gnu/bin/aarch64-linux-gnu

## 3.4 运行示例程序

### 3.4.1 RKNN3 Model Zoo介绍

RKNN3 Model Zoo提供了丰富的示例代码，帮助用户快速在搭载RK1820/1828的RK3588开发板上运行各种常用模型。工程目录结构如下：

```
rknn3-model-zoo
├── 3rdparty # 第三方库
├── datasets # 数据集
├── tokenizer # 分词器
├── examples # 示例代码
├── utils # 常用方法，如文件操作，画图等
├── build-linux.sh # 用于目标为Linux 系统开发板的编译脚本
├── build-android.sh # 用于目标为Android 系统开发板的编译脚本
└── ...
```

其中examples目录包含常用模型示例，如Qwen2.5和YOLO等。每个模型示例提供Python模型转换代码和C/C++模型推理示例代码（为便于描述，后续用RKNN3 Python和RKNN3 C Demo表示）。

### 3.4.2 安装板端RKNPU3环境

RKNN3 C Demo需要安装RKNPU3环境并启动rknn3\_transfer\_proxy服务。以下是RKNPU3环境的两个核心组件：

- **rknn3\_transfer\_proxy**: 运行在RK3588开发板上的后台代理服务，通过PCIe/USB在RK3588与RK1820/1828之间传输数据
- **RK1820/1828 Runtime库 (librknn3\_api.so)**: 负责在系统中加载RKNN模型，并通过相应接口调用专用神经处理单元（NPU）执行RKNN模型推理操作

在终端中执行以下命令，通过adb将librknn3\_api.so和通信代理rknn3\_transfer\_proxy传输到RK3588指定位置：

```
# 进入rknn3-runtime目录
cd Projects/rknn3-toolkit/rknn3-runtime

# 如果是android系统，安装 rknn3-api/Android/arm64-v8a/librknn3_api.so
adb push rknn3-api/Android/arm64-v8a/librknn3_api.so /vendor/lib64/

# 如果是linux系统，安装 rknn3-api/Linux/aarch64/librknn3_api.so
adb push rknn3-api/Linux/aarch64/librknn3_api.so /usr/lib/

# 如果是android系统，安装 rknn3_transfer_proxy/android-arm64-v8a/rknn3_transfer_proxy
adb push rknn3_transfer_proxy/android-arm64-v8a/rknn3_transfer_proxy /vendor/bin/

# 如果是linux系统，安装 rknn3_transfer_proxy/linux-aarch64/rknn3_transfer_proxy
adb push rknn3_transfer_proxy/linux-aarch64/rknn3_transfer_proxy /usr/bin/

# 增加可运行权限
adb shell chmod +x /usr/bin/rknn3_transfer_proxy

adb shell sync
```

#### 重要提示：

1. 将程序推送到RK3588板子后，务必执行sync操作（PC端执行adb shell sync，或adb shell进入板子后执行sync命令）！
2. rknn3\_transfer\_proxy在RK3588可设置为开机自动启动，如未设置，需手动执行rknn3\_transfer\_proxy命令。

### 3.4.3 运行常规模型

以YOLOv6模型为例，其目录结构如下：

```
rknn3-model-zoo
├── examples
│   └── yolov6
│       ├── cpp # C/C++ Demo 示例代码
│       ├── model # 模型、测试图片等文件
│       └── python # 模型转换脚本
└── ...
```

#### 1. 准备模型

进入 `rknn3-model-zoo/examples/yolov6/model` 目录，运行 `download_model.sh` 脚本下载可用的YOLOv6 ONNX模型到当前 `model` 目录下。在终端中执行以下命令：

```
# 进入 yolov6/model 目录
cd Projects/rknn3-model-zoo/examples/yolov6/model

# 运行 download_model.sh 脚本，下载 yolov6 onnx 模型
# 例如，下载好的 onnx 模型存放路径为 model/yolov6n_rknn3.onnx
./download_model.sh
```

相关输出：

```
--2025-09-15 11:30:44--
https://ftrg.zbox.filez.com/v2/delivery/data/95f00b0fc900458ba134f8b180b3f7a1/examples/y
olov6/yolov6n_rknn3.onnx
Resolving ftrg.zbox.filez.com (ftrg.zbox.filez.com)... 180.184.171.46
Connecting to ftrg.zbox.filez.com (ftrg.zbox.filez.com)|180.184.171.46|:443...
connected.
HTTP request sent, awaiting response... 200
Length: 18644871 (18M) [application/octet-stream]
Saving to: './yolov6n_rknn3.onnx'

./yolov6n_rknn3.onnx                                     100%[=====>]
17.78M  2.48MB/s    in 7.9s

2025-09-15 11:30:53 (2.26 MB/s) - './yolov6n_rknn3.onnx' saved [18644871/18644871]
```

注意：

- (1) 文件名带“`rknn3`”后缀的模型是专门为 `RK1820/1828` 优化，将 `yolo` 后处理（解码、候选框筛选排序以及 `nms` 等）内置到 `RK1820/1828` 端进行计算，减少数据传输压力。用户也可以修改脚本中的模型文件名，改成不带“`rknn3`”后缀的模型，此时模型后处理需要在应用程序中完成。
- (2) 从以下链接下载完整的 PyTorch 示例工程：[pytorch示例](#)，下载完成后，参考其中的导出文档，按照步骤进行操作，即可生成适配 `RK1820/1828` 的优化 ONNX 模型。下载链接可以在 `download_model.sh` 中查看。

#### 2. 模型转换

进入 `rknn3-model-zoo/examples/yolov6/python` 目录，运行 `convert.py` 脚本，该脚本将原始的 ONNX 模型转成 RKNN 模型，在计算机的终端窗口（命令行界面）中，执行以下命令：

```
# 进入 yolov6/python 目录
cd Projects/rknn3-model-zoo/examples/yolov6/python

# 运行 convert.py 脚本，将原始的 ONNX 模型转成 RKNN 模型
# 用法: python3 convert.py onnx_model_path [platform] [dtype(optional)]
[output_rknn_path(optional)]
#       platform choose from [RK1820/1828]
python convert.py ../model/yolov6n_rknn3.onnx RK1820 i8
```

相关输出：

```
--> Loading model
I Loading : 100%|██████████| 142/142 [00:00<00:00,
24888.89it/s]
W load_onnx: Please note that some float16/float64 data types in the model have been
modified to float32!
done
--> Building model
I FoldConstant : 100%|██████████| 218/218 [00:00<00:00,
505.15it/s]
I OpFusing 0: 100%|██████████| 100/100 [00:00<00:00,
522.77it/s]
I OpFusing 1 : 100%|██████████| 100/100 [00:00<00:00,
344.86it/s]
I OpFusing 0 : 100%|██████████| 100/100 [00:00<00:00,
274.72it/s]
I OpFusing 1 : 100%|██████████| 100/100 [00:00<00:00,
261.63it/s]
I OpFusing 2 : 100%|██████████| 100/100 [00:00<00:00,
249.86it/s]
I OpFusing 0 : 100%|██████████| 100/100 [00:00<00:00,
231.67it/s]
I OpFusing 1 : 100%|██████████| 100/100 [00:00<00:00,
222.40it/s]
I OpFusing 2 : 100%|██████████| 100/100 [00:00<00:00,
220.71it/s]
I FoldConstant : 100%|██████████| 170/170 [00:00<00:00,
622.77it/s]
...
I rknn building ...
I Split shape 0 done
I Compile all models for 1 cores
module optimization: [======] 1/1
(100.0%) Total: 0s
register configuration: [======] 1/1
(100.0%) Total: 0s
memory optimization: [======] 1/1
(100.0%) Total: 0s
weight sharing: [======] 1/1 (100.0%)
Total: 0s
code generation: [======] 1/1 (100.0%)
Total: 1s
I rknn building done.
done
--> Export rknn model
done
```

### 3. 运行 RKNN3 C Demo

- 板端为 Android 系统

以 Android 系统 (arm64-v8a 架构) 的 RK3588 平台为例，需要使用 rknn3-model-zoo 目录下的 build-android.sh 脚本进行编译。在运行 build-android.sh 脚本之前，需要指定编译器的路径 ANDROID\_NDK\_PATH 为本地的 NDK 编译器路径。即在 build-android.sh 脚本中，需要加入以下代码：

```
# 添加到 build-android.sh 脚本的开头位置即可  
ANDROID_NDK_PATH=Projects/android-ndk-r19c
```

然后在 rknn3-model-zoo 目录下，在计算机的终端窗口中运行 build-android.sh 脚本，参考命令如下：

```
# 编译yolov6 C++推理程序，主控设备为rk3588因此target设为rk3588  
. ./build-android.sh -t rk3588 -a arm64-v8a -b Release -d yolov6
```

- 板端为 Linux 系统

以 Linux 系统 (aarch64 架构) 的 RK3588 平台为例，需要使用 rknn3-model-zoo 目录下的 build-linux.sh 脚本进行编译。在运行 build-linux.sh 脚本之前，需要配置 GCC\_COMPILER 环境变量。

在计算机的终端窗口中，进入 rknn3-model-zoo 目录，运行以下命令编译：

```
# 导入GCC 交叉编译器环境变量  
export GCC_COMPILER=Projects/gcc-linaro-6.3.1-2017.05-x86_64_aarch64-linux-gnu/bin/aarch64-linux-gnu  
  
# 编译yolov6 C++推理程序，主控设备为rk3588因此target设为rk3588  
. ./build-linux.sh -t rk3588 -a aarch64 -b Release -d yolov6
```

安装运行：

编译脚本在编译完成后会将编译好的可执行程序打包在 Projects/rknn3-model-zoo/install 目录下。使用以下命令将程序和模型传输到开发板 /data 目录：

```

# 若为linux系统，传输编译好的程序到 /data 目录
adb push Projects/rknn3-model-zoo/install/rk3588_linux_aarch/rknn_yolov6_demo /data/

# 若为android系统，传输编译好的程序到 /data 目录
adb push Projects/rknn3-model-zoo/install/rk3588_android_arm64-v8a/rknn_yolov6_demo
/data/

# 进入rk3588终端
adb shell

# 进入demo目录
cd /data/rknn_yolov6_demo

# 运行rknn_yolov6_demo
# 用法 ./rknn_yolov6_demo <model_path> <weight_path> <image_path> <core_mask>
./rknn_yolov6_demo ./model/yolov6n_rknn3.rknn ./model/yolov6n_rknn3.weight
./model/bus.jpg 1

```

相关输出：

```

model is NHWC input fmt
model input height=640, width=640, channel=3
origin size=640x640 crop size=640x640
input image: 640 x 640, subsampling: 4:2:0, colorspace: YCbCr, orientation: 1
--> inference model
scale=1.000000 dst_box=(0 0 639 639) allow_slight_change=1 _left_offset=0 _top_offset=0
padding_w=0 padding_h=0
rga_api version 1.10.1_[0]
Pre-process time: 2.09 ms
-->rknn_run
Inference time: 21.37 ms
Post-process time: 0.44 ms
Total time: 23.90 ms
--> inference model done
bus @ (97 140 557 440) 0.951
person @ (109 239 221 537) 0.940
person @ (213 238 287 511) 0.932
person @ (480 233 560 520) 0.924
person @ (78 326 117 515) 0.455
write_image path: out.png width=640 height=640 channel=3 data=0x7f885e801

```

### 3.4.4 运行LLM

运行llm主要步骤分为：1) 模型导出为onnx；2) 模型转换为rknn；3) 编译运行示例程序。

以Qwen2.5模型为例，其目录结构如下：

```
rknn3-model-zoo
├── examples
│   └── Qwen2_5
│       ├── data # 量化数据集
│       ├── cpp # 模型推理示例代码
│       └── python # 模型转换脚本
└── ...
```

1. 安装依赖环境，在计算机的终端窗口（命令行界面）中，执行以下命令：

```
cd Projects/rknn3-model-zoo
pip install -r requirements.txt
```

2. llm模型导出为onnx模型，在计算机的终端窗口（命令行界面）中，执行以下命令：

```
# 设置环境变量
export PYTHONPATH=Projects/rknn3-model-zoo/

# 进入Qwen2.5 python目录
cd Projects/rknn3-model-zoo/examples/Qwen2_5/python/

# 导出onnx模型和配置文件
python export_llm.py
```

运行成功后会在 `examples/Qwen2_5/model` 文件夹下生成后缀名为 `onnx` 的模型文件、后缀名为 `.config.pkl` 的配置文件。

3. 转换为 `rknn` 模型，在计算机的终端窗口（命令行界面）中，执行以下命令：

```
# 进入Qwen2.5 python目录
cd Projects/rknn3-model-zoo/examples/Qwen2_5/python/

# 转换为rknn模型
python export_rknn.py
```

相关输出：

```
I rknn building done.
done
--> Export RKNN model
done
```

4. 运行 RKNN3 C Demo

- 板端为 Android 系统

以 Android 系统 (arm64-v8a 架构) 的 RK3588 平台为例，需要使用 `rknn3-model-zoo` 目录下的 `build-android.sh` 脚本进行编译。在运行 `build-android.sh` 脚本之前，需要指定编译器的路径 `ANDROID_NDK_PATH` 为本地的 NDK 编译器路径。即在 `build-android.sh` 脚本中，需要加入以下代码：

```
# 添加到 build-android.sh 脚本的开头位置即可  
ANDROID_NDK_PATH=Projects/android-ndk-r19c
```

然后在 rknn3-model-zoo 目录下，在计算机的终端窗口中运行 build-android.sh 脚本，参考命令如下：

```
# 编译Qwen2.5 C++推理程序，主控设备为rk3588因此target设为rk3588  
. ./build-android.sh -t rk3588 -a arm64-v8a -b Release -d Qwen2_5
```

- 板端为 Linux 系统

以 Linux 系统 (aarch64 架构) 的 RK3588 平台为例，需要使用 rknn3-model-zoo 目录下的 build-linux.sh 脚本进行编译。在运行 build-linux.sh 脚本之前，需要配置 `GCC_COMPILER` 环境变量。

在计算机的终端窗口中，进入 `rknn3-model-zoo` 目录，运行以下命令编译：

```
# 导入GCC 交叉编译器环境变量  
export GCC_COMPILER=Projects/gcc-linaro-6.3.1-2017.05-x86_64_aarch64-linux-gnu/bin/aarch64-linux-gnu  
  
# 编译Qwen2.5 C++推理程序，主控设备为rk3588因此target设为rk3588  
. ./build-linux.sh -t rk3588 -a aarch64 -b Release -d Qwen2_5
```

安装运行：

编译脚本编译完成后会将编译好的可执行程序打包在 `Projects/rknn3-model-zoo/install` 目录下。使用以下命令将程序和模型传输到开发板 `/data` 目录：

```
# 若为linux系统，传输编译好的程序到 /data 目录  
adb push Projects/rknn3-model-zoo/install/rk3588_linux_aarch/rknn_Qwen2_5_demo /data/  
  
# 若为android系统，传输编译好的程序到 /data 目录  
adb push Projects/rknn3-model-zoo/install/rk3588_android_arm64-v8a/rknn_Qwen2_5_demo /data/  
  
# 进入rk3588终端  
adb shell  
  
# 进入demo目录  
cd /data/rknn_Qwen2_5_demo/  
  
# 运行rknn_qwen2_5_demo  
# 用法： ./rknn_qwen2_5_demo <model_path> <weight_path> <tokenizer_path> <embedding_path>  
<core_mask> <prompt>  
. ./rknn_qwen2_5_demo ./model/Qwen2.5-1.5B-Instruct.rknn \  
./model/Qwen2.5-1.5B-Instruct.weight \  
./model/Qwen2.5-1.5B-Instruct.tokenizer.gguf \  
./model/Qwen2.5-1.5B-Instruct.embed.bin 0xff \  
"who are you?"
```

# 4 开发流程

本章将介绍如何基于 RKNN3 SDK 快速完成模型转换，并在 RK3588/RK1820/1828 上部署。

## 4.1 CNN模型开发流程介绍

用户可参考如下流程图了解RKNN的整体开发步骤，流程主要分为3个部分：模型转换、模型评估和板端部署运行。

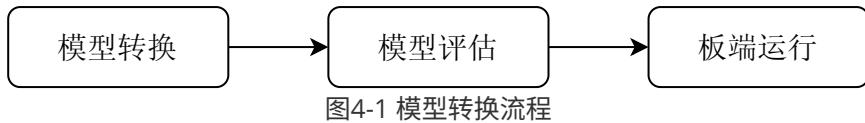


图4-1 模型转换流程

### 4.1.1 模型转换

在这一阶段，原始的深度学习模型会被转化为RKNN格式，以便在RK1820/1828上进行高效的推理。包括以下5个步骤：

- a. **获取原始模型**：获取或训练深度学习模型，建议使用主流框架，如ONNX、PyTorch或TensorFlow。
- b. **模型配置**：在RKNN3 Toolkit中进行必要的配置，如归一化参数、量化参数和目标平台等。
- c. **模型加载**：使用适当的加载接口将模型导入RKNN3-Toolkit，根据模型框架选择正确的加载接口。
- d. **模型构建**：通过 `rknn.build()` 接口构建RKNN模型，可选择是否进行量化，提高模型在硬件上推理的性能。
- e. **模型导出**：通过 `rknn.export_rknn()` 接口将RKNN模型导出，用于后续部署。

#### 4.1.1.1 模型转换流程

模型转换是 `rknn3-toolkit` 的核心功能，它允许用户将不同框架的深度学习模型转换为 RKNN 格式以在 RKNPU 上运行。本节将详细说明模型转换的流程，涉及的接口和注意事项。RKNN模型转换基本流程如下图所示：

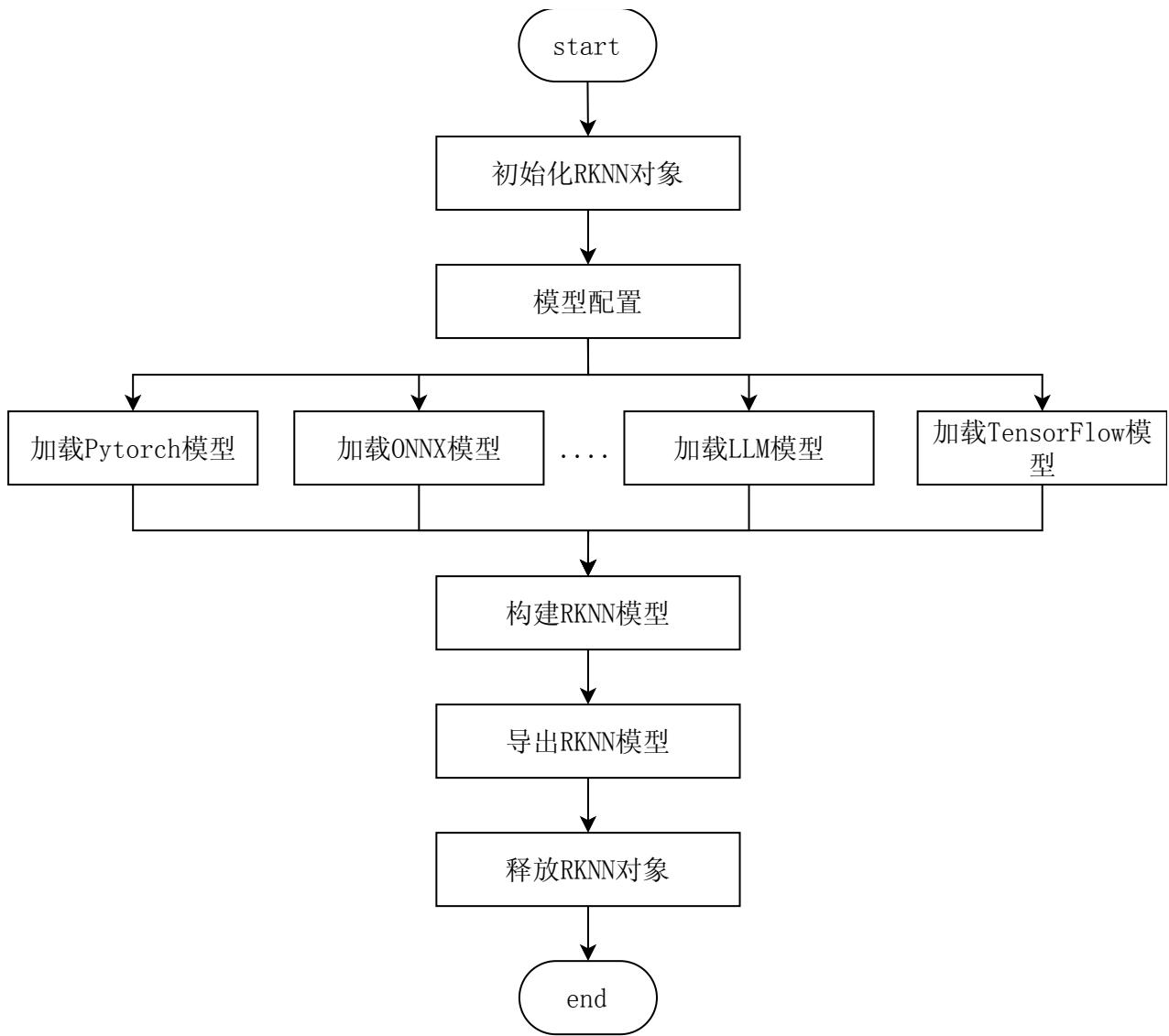


图4-2 模型转换流程

#### 4.1.1.2 关键接口说明

本节简要介绍常规模型转换流程中使用到的关键接口和重点参数。更详细的接口使用说明请参考<Rockchip\_RKNPU\_API\_Reference\_RKNN3\_Toolkit\_CN.pdf> 文档。

##### 4.1.1.2.1 创建、释放RKNN对象

在使用rknn3-toolkit的所有API接口时，都需要先调用 `RKNN()` 方法初始化 `RKNN` 对象，不再使用该对象时通过调用该对象的 `release()` 方法进行释放。

举例如下：

```

from rknn.api import RKNN
rknn = RKNN(verbose=True)
.....
rknn.release()

```

参数说明：

- `verbose`: 是否打印详细日志

#### 4.1.1.2.2 模型配置

在构建RKNN模型之前，需要先对模型进行通道均值、量化图片RGB2BGR转换、量化类型等的配置，这些操作可以通过 `config` 接口进行配置。

举例如下：

```
rknn.config(target_platform='RK1820', quantized_algorithm='normal',
            mean_values=[[0, 0, 0]], std_values=[[255, 255, 255]],
            input_attrs={'image_arrays': {'dtype': 'uint8', 'layout': 'NHWC'}})
```

主要参数说明：

- `target_platform`: 指定 RKNN 模型是基于哪个芯片平台生成的。
- `quantized_algorithm`: 计算每一层的量化参数时采用的量化算法，目前支持的量化算法包括 `normal`、`mmse`、`kl_divergence`、`grq` 及 `gdq`。默认值为 `normal`。
- `mean_values`: 输入的均值。
- `std_values`: 输入的归一化值。
- `input_attrs`: 用于设置推理时输入的属性，比如CNN图像模型期望输入RGB的数据，则需要设置输入的 `dtype` 为 `uint8`，`layout` 为 `NHWC`，其他类型的模型可以不设置。上述的 `image_arrays` 为模型输入的名称，需要根据实际模型进行修改。

#### 4.1.1.2.3 模型加载

rknn3-toolkit目前支持 `ONNX`、`PyTorch`、`TensorFlow`、`TensorFlow Lite` 等模型的加载转换。不同框架的模型需要调用对应接口进行加载。

本节以 ONNX 模型为例说明对应的加载接口，其他框架模型的加载接口请参考  
`<Rockchip_RKNPU_API_Reference_RKNN3_Toolkit_CN.pdf>` 文档。

rknn3-toolkit 提供 `load_onnx` 接口加载 ONNX 模型。

举例如下：

```
ret = rknn.load_onnx(model='../model/yolov6n.onnx')
```

主要参数说明：

- `model`: `ONNX` 模型文件路径。

#### 4.1.1.2.4 模型构建

rknn3-toolkit 提供 `build` 接口构建 RKNN 模型。

举例如下：

```
# Build model
print('--> Building model')
rknn.build(do_quantization=True, dataset=args.dataset_path)
if ret != 0:
    print('Build model failed!')
    exit(ret)
print('done')
```

主要参数说明：

- do\_quantization: 是否对模型进行量化。默认值为True。
- dataset: 用于量化校正的数据集。

#### 4.1.1.2.5 模型导出

rknn3-toolkit 提供 `export_rknn` 接口导出 RKNN 模型文件，用于模型部署。

举例如下：

```
ret = rknn.export_rknn(export_path='./yolov6.rknn')
```

主要参数说明：

- export\_path: 导出模型文件的路径。

注：导出模型包括两部分，一个是 `.rknn` 结尾的模型文件，一个是 `.weight` 结尾的权重文件，在后续部署时，这两个文件都会用到。

## 4.1.2 模型评估

该阶段的主要目标是评估模型推理结果的准确性、推理性能和内存占用等关键指标。

### 4.1.2.1 精度分析

比较量化模型与浮点模型推理结果之间的差异，以分析量化误差的来源。目前支持模拟器精度分析和连板精度分析。可以通过 `rknn.inference()` 接口的 `accuracy_analysis` 进行配置。

举例如下：

```
ret = rknn.init_runtime(target=platform, device_id='515e9b401c060c0b')

# Preprocess
image_src = cv2.imread(IMG_PATH)
img = preprocess(image_src)

# Inference and accuracy_analysis
outputs = rknn.inference(inputs=[img], accuracy_analysis=True)

# Postprocess
outputs = postprocess(outputs)
```

主要参数说明：

- inputs: 输入数据列表。

- accuracy\_analysis:是否启用精度分析

注: inputs 接收 nchw 数据格式的numpy输入。

#### 4.1.2.2 性能评估

主要分析模型在运行过程中的性能评估结果，包括算子的运行时间、周期和带宽。注意：性能评估需要在 rknn.config() 中配置 profile\_mode=True。

举例如下：

```
ret = rknn.init_runtime(target=platform, core_mask=0xff)
ret = rknn.eval_perf()
```

以RK1820、Qwen2.5-0.5B为例，执行eval\_perf后输出性能评估结果如下：

Op Time Ranking Table (Core 7)					
OpType	Calls	CPU(us)	NPU(us)	Total(us)	Ratio(%)
exMatMul	121	0	56357	56357	23.04
Add	120	0	54594	54594	22.32
exMatMulActivation	48	0	25881	25881	10.58
rcclScatter	50	0	23468	23468	9.60
rcclGather	49	0	23015	23015	9.41
exNorm	49	0	22701	22701	9.28
Reshape	50	0	22565	22565	9.23
exAttention	24	0	14645	14645	5.99
Transpose	1	0	461	461	0.19
Gather	1	0	447	447	0.18
OutputOperator	1	0	447	447	0.18
Total	0		244581	244581	100.00
...					
Op Time Ranking Table (Core 0)					
OpType	Calls	CPU(us)	NPU(us)	Total(us)	Ratio(%)
rcclReduce	147	0	68083	68083	21.60
exMatMul	121	0	55740	55740	17.68
Add	120	0	53837	53837	17.08
rcclBroadcast	56	0	26167	26167	8.30
exMatMulActivation	48	0	25648	25648	8.14
rcclScatter	50	0	23096	23096	7.33
rcclGather	49	0	22677	22677	7.19
exNorm	49	0	22479	22479	7.13
exAttention	24	0	12909	12909	4.09
Reshape	3	0	1346	1346	0.43
OutputProc	1	0	568	568	0.18

Transpose	1	0	458	458	0.15
Sub	1	0	457	457	0.14
Clip	1	0	456	456	0.14
Mul	1	0	455	455	0.14
Gather	1	0	439	439	0.14
OutputOperator	1	0	438	438	0.14
<hr/>					
Total		0	315253	315253	100.00
<hr/>					

#### 4.1.2.4 内存评估

获取模型的内存消耗评估结果。

示例代码：

```
ret = rknn.init_runtime(target=platform)
memory_detail = rknn.eval_memory()
```

内存评估结果如下：

```
===== Memory Usage Information =====
Device Memory:
System : 19.12 MB total, 9.76 MB free, 9.36 MB used ( 49.0%)
Node 0 : 319.50 MB total, 263.23 MB free, 56.27 MB used ( 17.6%)
Node 1 : 319.50 MB total, 265.15 MB free, 54.35 MB used ( 17.0%)
Node 2 : 299.90 MB total, 245.51 MB free, 54.39 MB used ( 18.1%)
Node 3 : 319.50 MB total, 265.15 MB free, 54.35 MB used ( 17.0%)
Node 4 : 299.90 MB total, 250.54 MB free, 49.36 MB used ( 16.5%)
Node 5 : 299.90 MB total, 249.03 MB free, 50.87 MB used ( 17.0%)
Node 6 : 319.50 MB total, 268.59 MB free, 50.91 MB used ( 15.9%)
Node 7 : 319.50 MB total, 268.63 MB free, 50.87 MB used ( 15.9%)

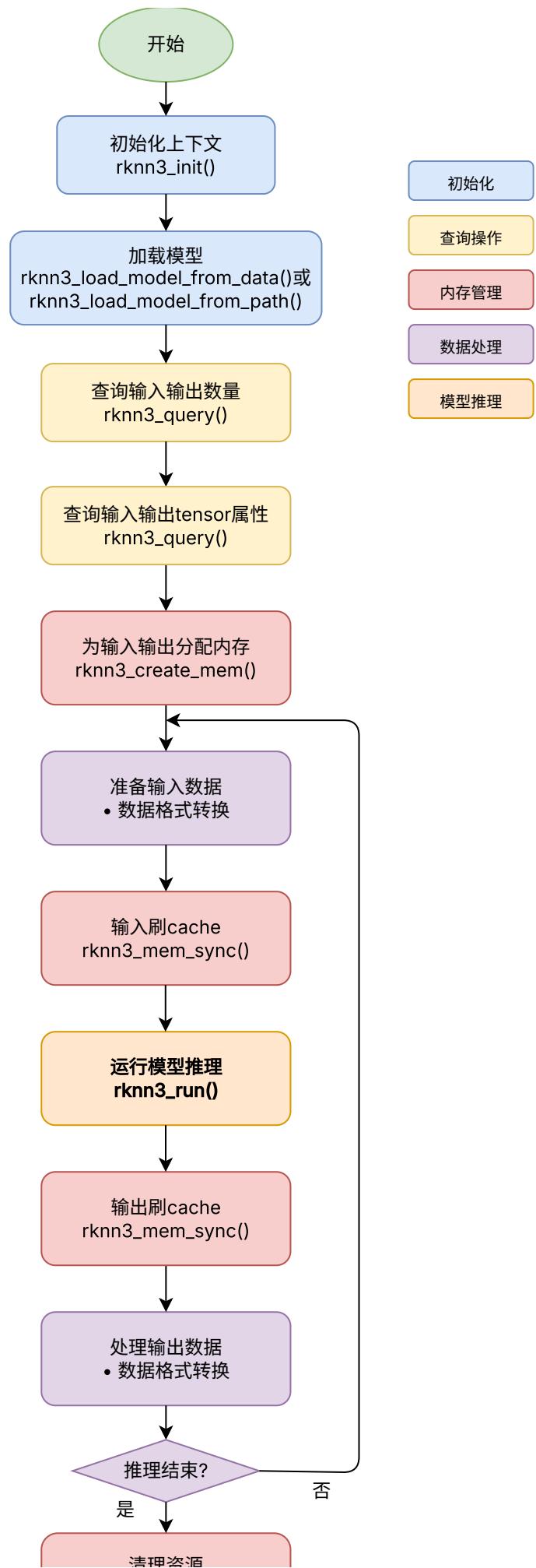
Per-Core Memory Allocation (MB):
Core Command Weight Internal KVCache Total
----- ----- -----
0 5.06 40.66 4.25 6.23 56.20
1 2.75 42.05 3.24 6.23 54.28
2 2.79 42.05 3.24 6.23 54.32
3 2.75 42.05 3.24 6.23 54.28
4 2.76 37.05 3.24 6.23 49.29
5 2.75 38.58 3.24 6.23 50.80
6 2.79 38.58 3.24 6.23 50.85
7 2.75 38.58 3.24 6.23 50.81
----- -----
Total 24.41 319.60 26.94 49.88 420.84
=====
```

### 4.1.3 模型部署

这个阶段涵盖了模型的实际部署和运行。它通常包括以下步骤：

- a. 模型初始化：加载RKNN模型，准备进行前处理。
- b. 模型前处理：加载待推理数据，准备进行推理。
- c. 模型推理：执行推理操作，将输入数据传递给模型并获取推理结果。
- d. 模型后处理：获取推理结果进行后处理，后处理结果传给应用端。
- e. 模型释放：在完成推理流程后，释放模型资源，以便其他任务使用RKNN模型。

参考流程图如下：



初始化

查询操作

内存管理

数据处理

模型推理

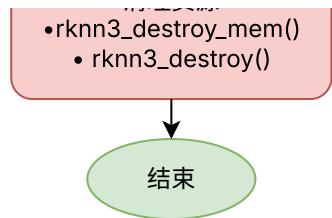


图4-3 CNN模型部署流程

#### 4.1.4 关键接口说明

本节简要说明CNN模型部署过程中使用到的 RKNN3 C API，详细的接口使用说明请参考 <Rockchip\_RKNPU\_API\_Reference\_RKNNRT3\_CN.pdf> 文档。

##### 4.1.4.1 上下文初始化和销毁

RKNN3 C API 提供 `rknn3_init` 接口初始化上下文信息，提供 `rknn3_destroy` 接口销毁上下文信息。

举例如下：

```

rknn3_context ctx;
int ret = rknn3_init(&ctx, nullptr);
if (ret != RKNN3_SUCCESS) {
    return -1;
}

.....
rknn3_destroy(ctx);

```

主要参数说明如下：

- `rknn3_context* context`: 指向将被初始化的RKNN上下文句柄的指针

##### 4.1.4.2 模型加载

RKNN3 C API 提供 `rknn3_load_model_from_path` 接口从文件路径加载 `RKNN` 模型到指定的上下文中。

举例如下：

```

// 加载模型
ret = rknn3_load_model_from_path(ctx, model_path, weight_path);
if (ret != RKNN3_SUCCESS) {
    printf("rknn3_load_model failed! ret=%d\n", ret);
    rknn3_destroy(ctx);
    return -1;
}

```

主要参数说明如下：

- `rknn3_context context`: `RKNN` 上下文句柄
- `const char* model_path`: `RKNN` 模型文件的路径
- `const char* weight_path`: `RKNN` 权重文件的路径

RKNN3 C API 同样提供 `rknn3_load_model_from_data` 接口从内存加载 `RKNN` 模型。该接口的主要参数说明如下：

- rknn3\_context context: RKNN 上下文句柄
- const void\* model\_data: RKNN 模型数据
- uint64\_t model\_size: RKNN 模型数据大小
- const void\* weight\_data: 权重数据
- uint64\_t weight\_size: 权重大小

#### 4.1.4.3 模型初始化

RKNN3 C API 提供 rknn3\_model\_init 接口完成模型初始化。

举例如下：

```
ret = rknn3_model_init(ctx, &config);
if (ret != RKNN3_SUCCESS) {
    printf("rknn3_model_init failed! ret=%d\n", ret);
    rknn3_destroy(ctx);
    return -1;
}
```

主要参数说明如下：

- rknn3\_context context: RKNN 上下文句柄
- rknn3\_config\* config: 模型初始化的配置参数

#### 4.1.4.4 模型属性查询

RKNN3 C API 提供 rknn3\_query 接口查询模型输入、输出个数，输入、输出属性等信息。

举例如下：

```

// 查询输入输出信息
rknn3_input_output_num io_num;
ret = rknn3_query(ctx, RKNN3_QUERY_IN_OUT_NUM, &io_num, sizeof(io_num));
if (ret != RKNN3_SUCCESS) {
    printf("rknn3_query input/output num failed! ret=%d\n", ret);
    rknn3_destroy(ctx);
    return -1;
}

// 查询输入tensor的attr
rknn3_tensor_attr input_attrs[io_num.n_input];
for (int i = 0; i < io_num.n_input; i++) {
    input_attrs[i].index = i;
    ret = rknn3_query(ctx, RKNN3_QUERY_INPUT_ATTR, input_attrs + i,
                      sizeof(rknn3_tensor_attr));
    if (ret != RKNN3_SUCCESS) {
        printf("rknn3_query input attr failed! ret=%d\n", ret);
        rknn3_destroy(ctx);
        return -1;
    }
    dump_tensor_attr(input_attrs + i);
}

// 查询输出tensor的attr
rknn3_tensor_attr output_attrs[io_num.n_output];
for (int i = 0; i < io_num.n_output; i++) {
    output_attrs[i].index = i;
    ret = rknn3_query(ctx, RKNN3_QUERY_OUTPUT_ATTR, output_attrs + i,
                      sizeof(rknn3_tensor_attr));
    if (ret != RKNN3_SUCCESS) {
        printf("rknn3_query output attr failed! ret=%d\n", ret);
        rknn3_destroy(ctx);
        return -1;
    }
    dump_tensor_attr(output_attrs + i);
}

```

主要参数说明如下：

- rknn3\_query\_cmd cmd: 查询命令类型
  - RKNN3\_QUERY\_IN\_OUT\_NUM: 查询输入和输出 tensor 的数量
  - RKNN3\_QUERY\_INPUT\_ATTR: 查询输入 tensor 的属性
  - RKNN3\_QUERY\_OUTPUT\_ATTR: 查询输出 tensor 的属性
- void\* info: 用于存储查询结果的缓冲区指针
- uint64\_t size: `info` 缓冲区的大小 (字节)

#### 4.1.4.5 同步输入、输出数据

RKNN3 C API 提供 `rknn3_mem_sync` 接口完成 host 和 RK182x 之间内存数据的同步功能，这个接口可以用来同步模型输入数据，也可以用来同步模型输出数据。

举例如下：

```
// sync inputs
for (int i = 0; i < io_num.n_input; i++)
{
    ret = rknn3_mem_sync(ctx, inputs[i].mem, RKNN3_MEMORY_SYNC_TO_DEVICE);
    if (ret != RKNN3_SUCCESS)
    {
        printf("rknn3_mem_sync input %d failed! ret=%d\n", i, ret);
        break;
    }
}

// sync outputs
for (int i = 0; i < io_num.n_output; i++)
{
    rknn3_mem_sync(ctx, outputs[i].mem, RKNN3_MEMORY_SYNC_FROM_DEVICE);
    if (ret != RKNN3_SUCCESS)
    {
        printf("rknn3_mem_sync output %d failed! ret=%d\n", i, ret);
        break;
    }
}
```

主要参数说明如下：

- `rknn3_tensor_mem* mem`: 待同步内存。
- `rknn3_mem_sync_mode mode`: 同步模式。如果是从 Host 端（如 RK3588）同步数据到 RK182X，将 `mode` 被设置成 `RKNN3_MEM_SYNC_TO_DEVICE`；如果要将内存数据从 RK182X 同步回 Host 端（如 RK3588）做进一步处理，将 `mode` 设置成 `RKNN3_MEM_SYNC_FROM_DEVICE`。

#### 4.1.4.6 模型推理

RKNN3 C API 提供 `rknn3_run` 接口完成模型推理，这个接口会阻塞直到推理结束。

举例如下：

```
// 运行模型
ret = rknn3_run(ctx, inputs, io_num.n_input, outputs, io_num.n_output);
if (ret != RKNN3_SUCCESS) {
    printf("rknn3_run failed! ret=%d\n", ret);
    rknn3_destroy(ctx);
    return -1;
}
```

主要参数说明如下：

- const rknn3\_tensor inputs[]: 包含输入数据的输入 `tensor` 数组
- uint32\_t n\_inputs: 输入 `tensor` 的数量
- rknn3\_tensor outputs[]: 用于存储推理结果的输出 `tensor` 数组
- uint32\_t n\_outputs: 输出 `tensor` 的数量

## 4.2 LLM 模型开发流程介绍

### 4.2.1 模型转换

LLM模型的开发流程与CNN模型有些差异，需要先将HuggingFace上的模型转换成ONNX模型，然后导出RKNN模型。整体流程如下：

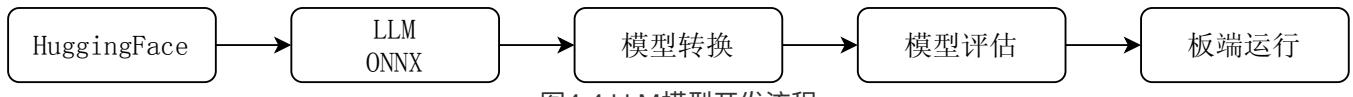


图4-4 LLM模型开发流程

从HuggingFace模型转成ONNX模型可以参考rknn3-model-zoo的《LLM模型适配教程.md》。下面以Qwen2.5 0.5B为例简要说明这一流程。相关代码可以参考rknn3-model-zoo下的 `examples/Qwen2_5/python/export_llm.py`

a. 构建LLM Torch模型，这一步对于大部分LLM都通用，没有特别修改

```
from transformers import AutoModelForCausalLM, AutoConfig
config = AutoConfig.from_pretrained(args.model_path, **kwargs)
model = AutoModelForCausalLM.from_pretrained(args.model_path, **kwargs)
```

b. 从AutoModel转ONNX

```
dummy_input = torch.zeros((1, in_len), dtype=torch.long)
attention_mask = torch.ones((1, in_len), dtype=torch.float)
position_ids = torch.arange(0, in_len, dtype=torch.long).unsqueeze(0)

inputs = (dummy_input, attention_mask, position_ids)
input_names = ["input_ids", "attention_mask", "position_ids"]
dynamic_axes = {}
if args.dynamic_shape:
    dynamic_axes.update({
        'input_ids': {1: 'sequence'},
        'attention_mask': {1: 'sequence'},
        'position_ids': {1: 'sequence'},
    })
torch.onnx.export(
    model,
    inputs,
    args.export_llm_path,
    export_params=True,
    opset_version=19,
    do_constant_folding=True,
    input_names=input_names,
    output_names=output_names,
    dynamic_axes=dynamic_axes,
)
```

这一步对大部分LLM模型也是通用的，但是需要注意的是inputs需要和原始的torch定义一致，有部分模型可能不是按照 `inputs = (dummy_input, attention_mask, position_ids)` 的顺序。

c. 导出LLM特有的配置，比如 `chat_template`、`vocab_size`、`hidden_size` 等，用户通常不需要修改这份代码。

```
export_llm_config(args.model_path, os.path.splitext(args.export_llm_path)[0] +  
.config.pkl', chat_context, prompt)
```

导出的 `xxx.config.pkl` 在下一步做模型转换的时候需要用到。

d. 导出tokenizer。

```
export_tokenizer(args.model_path, os.path.splitext(args.export_llm_path)[0] +  
.tokenizer.gguf')
```

tokenizer主要在板端推理时使用。rknn3-model-zoo使用的是<https://github.com/ggml-org/llama.cpp> 的 tokenizer。用户也可以使用自己的tokenizer。只需要在板端推理时，实现 `RKLLMCallback` 中的 `tokenizer_callback` 即可（具体使用参考《Rockchip\_RKNPU\_API\_Reference\_RKNNRT3\_CN》）。

e. 导出embedding

```
export_embed_weight(model.model.embed_tokens.weight, os.path.splitext(args.export_llm_path)  
[0] + '.embed.bin')
```

由于RK1820/1828的DRAM大小有限，因此将LLM模型中的embedding放到HOST处理。这里生成的 `xxx.embed.bin` 用于 `RKLLMCallback` 中的 `embed_callback`

f. 加载ONNX模型

```
rknn.config(target_platform='RK1820',  
           quantized_dtype='w4a16', quantized_algorithm='grq',  
           quantized_method='group32')  
  
ret = rknn.load_llm(model=args.onnx_path, config=args.config)
```

这里的model是上述b步骤导出的ONNX模型，config是步骤c导出的config.pkl

- `target_platform`: 目标芯片平台，要求是 RK1820/1828
- `quantized_dtype`: 量化类型，对应LLM模型要求是 `w4a16`
- `quantized_algorithm`: 量化算法，可选 `grq` 或者 `normal`，一般使用 `grq` 的量化精度更高，建议使用 `grq`
- `quantized_method`: 量化方法，建议使用 `'group32'`。

g. 量化并导出rknn模型

```
rknn.build(do_quantization=True, dataset=args.dataset_path)  
ret = rknn.export_rknn(args.rknn_path)
```

到这一步将生成板端运行所需要的所有的文件，包括 `xxx.rknn`、`xxx.weight`、`xxx.embed.bin`、`xxx.tokenizer.gguf`

## 4.2.2 LLM模型部署

LLM模型的部署与CNN模型部署存在较大差异，为了更好的管理LLM模型上下文，引入 `Session` 的概念。

`rknn3 session` 的主要特点如下：

- `rknn3 session` 基于 `rknn3 context` 之上构建，一个 `rknn3 context` 可以建立多个 `session`。
- 多个 `session` 共用权重、internal memory等，但 `KVCache` 及 `LoRA` 不复用。
- 多个 `session` 之间不能同时运行，同一时刻只能运行一个 `session`。

基于会话管理的 LLM 模型部署流程如下图所示：

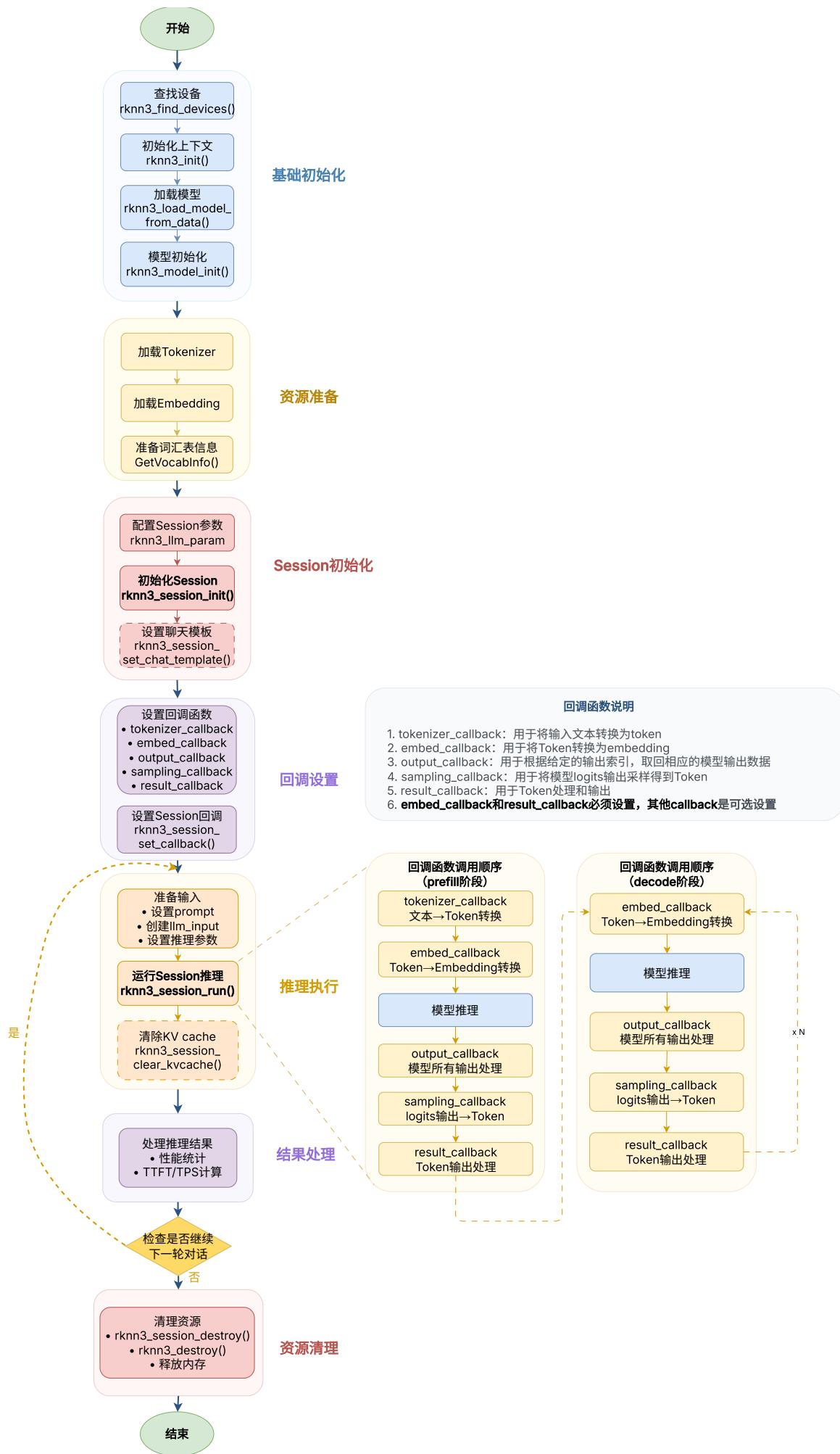


图4-5 基于会话管理的 LLM 模型部署流程

## 4.2.3 关键接口说明

### 4.2.3.1 会话创建和销毁

RKNN3 C API 提供 `rknn3_session_init` 接口初始化 RKNN 会话，提供 `rknn3_session_destroy` 接口销毁会话。

举例如下：

```
// 设置基础会话参数
rknn3_llm_param params
params.logits_name = {0};
params.logits_name = "logits"; //指定输出节点名称
params.max_context_len = 1024; //设置最大上下文长度，单位token
params.sampling_param.temperature = 1.0f; //设置生成的“随机性”程度，(0,1]
params.sampling_param.top_k = 1; //设置topk
params.sampling_param.top_p = 0.9; //设置topp
params.sampling_param.repeat_penalty = 1.1f; //设置重复惩罚系数
params.sampling_param.frequency_penalty = 0.0f; //频率惩罚参数
params.sampling_param.presence_penalty = 0.0f; //存在惩罚参数
params.vocab_info.vocab_size = vocab_info.vocab_size; //设置词表大小
params.vocab_info.n_special_eos_id = vocab_info.n_special_eos_id; //设置序列结束符个数
//设置EOS token id
memcpy(params.vocab_info.special_eos_id, vocab_info.special_eos_id,
sizeof(vocab_info.special_eos_id));

params.vocab_info.n_special_bos_id = vocab_info.n_special_bos_id; //设置序列开始符个数
//设置BOS token id
memcpy(params.vocab_info.special_bos_id, vocab_info.special_bos_id,
sizeof(vocab_info.special_bos_id));

params.vocab_info.linefeed_id = vocab_info.linefeed_id; //换行符token ID
params.vocab_info.ignore_eos_token = 0; //是否忽略 EOS token 继续推理

// 初始化会话
rknn3_session* session = rknn3_session_init(ctx, &params, 1);
if (!session) {
    printf("Failed to initialize test session\n");
    return -1;
}

.....
// 销毁会话
rknn3_session_destroy(session);
```

主要参数说明如下：

- `rknn3_llm_param* params`: 包含会话配置参数的 `rknn3_llm_param` 结构指针
- `int n_params`: 参数数量
- `rknn3_session* session`: RKNN会话指针

#### 4.2.3.2 设置聊天模板

RKNN3 C API 提供 `rknn3_session_set_chat_template` 设置LLM的聊天模板，包括系统提示、前缀和后缀。

举例如下：

```
const char* system_prompt = "<|im_start|>system\nYou are Qwen, created by Alibaba Cloud.\nYou are a helpful assistant.<|im_end|>\n";
const char* prompt_prefix = "<|im_start|>user\n";
const char* prompt_postfix = "<|im_end|>\n<|im_start|>assistant\n";

int ret;
// Set Chat Template
ret = rknn3_session_set_chat_template(llm_ctx->rknn_sess, system_prompt, prompt_prefix,
prompt_postfix);
if (ret < 0)
{
    printf("Failed to set chat template\n");
    goto out;
}
```

主要参数说明如下：

- `rknn3_session*` `session`: RKNN3会话句柄
- `const char*` `system_prompt`: 定义语言模型上下文或行为的系统提示
- `const char*` `prompt_prefix`: 聊天中用户输入前添加的前缀
- `const char*` `prompt_postfix`: 聊天中用户输入后添加的后缀

#### 4.2.3.3 注册回调函数

RKNN3 C API 提供 `rknn3_session_set_callback` 为RKNN3会话注册回调函数。

举例如下：

```
// LLM Callback
RKLLMCallback callback = {0};
// result 回调函数，负责接收并处理模型每次生成的 token 输出
callback.result_callback = result_callback;
callback.result_userdata = tokenizer;
// embedding 回调函数，负责将 token 转换为模型可处理的 embedding 向量
callback.embed_callback = embed_callback;
callback.embed_userdata = &embedding_info;
// tokenizer 回调函数，负责将输入文本转换为 token
callback.tokenizer_callback = tokenizer_callback;
callback.tokenizer_userdata = tokenizer;
// sampling 回调函数，负责对模型输出的 token 概率分布进行采样
callback.sampling_callback = sampling_callback;
callback.sampling_userdata = &embedding_info;

// LLM Set Callback
ret = rknn3_session_set_callback(llm_ctx->rknn_sess, &(callback));
if (ret < 0)
{
    printf("Failed to set callback\n");
    goto out;
}
```

主要参数说明如下：

- `rknn3_session*` session：RKNN3会话实例指针。
- `RKLLMCallback*` callback：包含回调函数的 `RKLLMCallback` 结构指针

回调函数具体设置请参考<Rockchip\_RKNPU\_API\_Reference\_RKNNRT3\_CN.pdf> 文档

#### 4.2.3.4 会话推理

RKNN3 C API 提供 `rknn3_session_run` 接口进行会话推理。

举例如下：

```
// 准备输入数据
int n_inputs = 1;
rknn3_llm_input inputs[n_inputs];
char *prompt = "Please tell me a story!";
rknn3_llm_infer_param param = {.keep_history = 0, .max_new_tokens = 128};
rknn3_llm_tensor input_tensor = {.name = NULL, .prompt = prompt, .embed = NULL, .tokens =
NULL, .n_tokens = 0};
rknn3_llm_input input;
input.input_type = RKNN3_LLM_INPUT_PROMPT;
input.llm_input = input_tensor;
inputs[0] = input;

ret = rknn3_session_run(session, inputs, n_inputs, &param);
if (ret != RKNN3_SUCCESS) {
    printf("session run failed with error: %d\n", ret);
    rknn3_session_destroy(session);
    return -1;
}
```

主要参数说明如下：

- `rknn3_session*` session：RKNN3会话句柄指针。
- `rknn3_llm_input inputs[]`：包含输入数据的输入 tensor 数组。
- `uint32_t n_inputs`：提供的输入 tensor 数量。
- `rknn3_llm_infer_param* param`：推理参数配置指针。

注：RKNN3 C API 同时提供 `rknn3_session_run_async` 接口实现异步推理功能。

#### 4.2.3.5 终止会话

举例如下：

```
ret = rknn3_session_stop(session);
if (ret != RKNN3_SUCCESS) {
    printf("session stop failed with error: %d\n", ret);
    rknn3_session_destroy(session);
    return -1;
}
```

主要参数说明如下：

- `rknn3_session*` session：RKNN3会话句柄指针。

#### 4.2.3.6 清理 KV Cache

RKNN3 C API 提供 `rknn3_session_clear_kvcache` 接口进行清理 KV Cache。

举例如下：

```
ret = rknn3_session_clear_kvcache(session, RKNN3_KVCACHE_CLEAR_ALL);
if (ret != RKNN3_SUCCESS)
{
    printf("session clear kvcache failed with error: %d\n", ret);
    rknn3_session_destroy(session);
    return -1;
}
```

主要参数说明如下：

- `rknn3_session* session`: RKNN3会话句柄指针。
- `rknn3_kvcache_clear_policy clear`: 清理 KV Cache 的策略，定义如何清理 KV Cache

#### 4.2.4 LLM Vision模型开发流程介绍

针对视觉多模态模型而言，除了导出其中的LLM模型外，还需要导出Vision模型，导出LLM模型的流程与4.2.2章节介绍的一样，下面主要介绍vision模型的导出及部署。



- 使用 `rknn3-model-zoo` 工程导出多模态模型时，会有多个 `.onnx` 后缀的模型，例如 `FastVLM` 模型会导出用于生成文本的 `FastVLM-llm.onnx` 模型和对图像做 embedding 的 `FastVLM-vision.onnx` 模型。转 RKNN 模型时，它们使用不同的加载接口，`FastVLM-llm.onnx` 模型使用 `load_llm` 接口加载，而 `FastVLM-vision.onnx` 模型使用 `load_onnx` 接口加载。具体示例如下：

```
# 构建视觉模型
rknn.config(target_platform='RK1820',
            quantized_dtype='w4a16', quantized_algorithm='normal',
            quantized_method='group32', core_num=8)
rknn.load_onnx(model='../../model/vision/FastVLM-vision.onnx')
rknn.build(do_quantization=True, dataset=args.dataset_path)
rknn.export_rknn('.../model/vision/FastVLM-vision.rknn')

# 构建llm模型
rknn.config(target_platform='RK1820',
            quantized_dtype='w4a16', quantized_algorithm='grq',
            quantized_method='group32')
rknn.load_llm(model='.../model/llm/FastVLM-llm.onnx',
              config='.../model/llm/FastVLM-llm.config.pkl')
rknn.build(do_quantization=True,
           dataset='.../data/llm/dataset.txt')
rknn.export_rknn('.../model/llm/FastVLM-llm.rknn')
```

视觉模型在板端推理时，使用的是普通RKNN3 API接口，具体使用可以参考4.1.4章节。推理完后将得到 `img_embeds`，在推理LLM模型时，将 `prompt` 及 `img_embeds` 同时传给LLM模型，得到最终的结果。示例如下：

```
// LLM Input
tensor.name          = "input_embeds";
tensor.prompt        = prompt;
tensor.image.image_embed = img_embeds;
if(rknn_app_ctx.vision.embeds_ndims == 2) {
    tensor.image.n_image_tokens = rknn_app_ctx.vision.embeds_shape[0];
    tensor.image.n_image      = 1;
} else {
    tensor.image.n_image_tokens = rknn_app_ctx.vision.embeds_shape[1];
    tensor.image.n_image      = rknn_app_ctx.vision.embeds_shape[0];
}
tensor.image.image_width   = rknn_app_ctx.vision.model_width;
tensor.image.image_height  = rknn_app_ctx.vision.model_height;
tensor.image.image_start   = "<|vision_start|>";
tensor.image.image_end     = "<|vision_end|>";
tensor.image.image_content = "<|image_pad|>";
tensor.enable_thinking = false;

inputs[0].input_type = RKNN3_LLM_INPUT_MULTIMODAL;
inputs[0].multimodal_input = tensor;

ret = rknn3_session_run(llm_ctx->rknn_sess, inputs, n_inputs, &llm_infer_param);
```

## 5 RKLLM3 Server

rkllm3-server是基于RKNPU3实现的一套基本的LLM Server。

功能:

- 基于RKNPU3的LLM推理
- OpenAI API 兼容的对话模板

### 5.1 使用方法

通用参数

参数	说明
-a, --alias STRING	设置模型别名（供REST API使用）
-c, --ctx-size N	提示词上下文大小（默认：4096，0 = 从模型加载）
-n, --predict, --n-predict N	要预测的token数量（默认：-1，-1 = 上下文大小）
-m, --model FNAME	RKNN LLM 模型路径
--weight FNAME	RKNN LLM 权重路径（可不设置）
--model2 FNAME	多模态模型时的vision模型路径
--weight2 FNAME	多模态模型时的vision权重路径（可不设置）
--model3 FNAME	多模态模型时的audio模型路径
--weight3 FNAME	多模态模型时的audio权重路径（可不设置）
--vocab FNAME	词汇表路径
--embed FNAME	embed的bin文件路径
--mel-filter FNAME	mel filters路径（用于audio模型）
--img-start STRING	多模态模型的图像输入前缀
--img-end STRING	多模态模型的图像输入后缀
--img-content STRING	多模态模型的图像输入的占位符
--audio-start STRING	多模态模型的语音输入前缀
--audio-end STRING	多模态模型的语音输入后缀
--audio-content STRING	多模态模型的语音输入的占位符
--img-width N	多模态模型的输入图像的宽（部分裁减过的模型需要设置，如qwen3_vl）

参数	说明
--img-height N	多模态模型的输入图像的高（部分裁减过的模型需要设置，如qwen3_vl）
--chat-template-file JINJA_TEMPLATE_FILE	设置自定义Jinja聊天模板（默认：使用模型元数据中的模板）
--embedding	是否是词嵌入模型

## 采样参数

参数	说明
--temp N	温度（默认：0.8）
--top-k N	top-k采样（默认：40, 0 = 禁用）
--top-p N	top-p采样（默认：0.9, 1.0 = 禁用）
--repeat-penalty N	惩罚重复token序列（默认：1.0, 1.0 = 禁用）
--presence-penalty N	重复alpha存在惩罚（默认：0.0, 0.0 = 禁用）
--frequency-penalty N	重复alpha频率惩罚（默认：0.0, 0.0 = 禁用）

## 示例专用参数

参数	说明
-h, --help, --usage	打印使用说明并退出
--host HOST	监听IP地址（默认：127.0.0.1）
--port PORT	监听端口（默认：8080）
-to, --timeout N	服务器读写超时时间（秒）（默认：600）
--device-id STRING	设备ID（多设备时需要指定具体的设备ID）
--log-level N	日志等级（默认：0）

## 5.2 快速上手

运行以下命令开启服务进程：

```

# LLM模型
./rkllm3-server -m qwen2.5-3b.rknn --vocab qwen2.5-3b.tokenizer.gguf --embed qwen2.5-
3b.embed.bin --host 0.0.0.0 --port 8080 -c 768 --n_predict 512 --repeat-penalty 1.1 --
presence-penalty 1.0 --frequency-penalty 1.0 --top-k 1 --top-p 0.8 --temp 0.8

# 多模态模型 (LLM+VISION)
./rkllm3-server -m Qwen2.5-VL-3B-llm.rknn --model2 Qwen2.5-VL-3B-vision.rknn --vocab
Qwen2.5-VL-3B-llm.tokenizer.gguf --embed Qwen2.5-VL-3B-llm.embed.bin --host 0.0.0.0 --port
8080 -c 768 --n_predict 512 --repeat-penalty 1.1 --presence-penalty 1.0 --frequency-penalty
1.0 --top-k 1 --top-p 0.8 --temp 0.8 --img-start "<|vision_start|>" --img-end "
<|vision_end|>" --img-content "<|image_pad|>" --img-width 392 --img-height 392

# 多模态模型 (LLM+VISION+AUDIO)
./rkllm3-server -m Qwen2.5-Omni-3B-llm.rknn --model2 Qwen2.5-Omni-3B-vision.rknn --model3
Qwen2.5-Omni-3B-audio.rknn --vocab Qwen2.5-Omni-3B-llm.tokenizer.gguf --embed Qwen2.5-Omni-
3B-llm.embed.bin --mel-filter mel_128_filters.txt --host 0.0.0.0 --port 8080 -c 768 --
n_predict 512 --repeat-penalty 1.1 --presence-penalty 1.0 --frequency-penalty 1.0 --top-k 1
--top-p 0.8 --temp 0.8 --img-start "<|vision_bos|>" --img-end "<|vision_eos|>" --img-content
"<|IMAGE|>" --audio-start "<|audio_bos|>" --audio-end "<|audio_eos|>" --audio-content "
<|AUDIO|>"

# 词嵌入模型
./rkllm3-server -m Qwen3-Embedding-4B.rknn --vocab Qwen3-Embedding-4B.tokenizer.gguf --embed
Qwen3-Embedding-4B.embed.bin --embedding

# 同时加载多个模型 (需要确保内存足够加载多个模型)
./rkllm3-server --params-file params.json

```

## 5.3 用CURL进行测试

使用 [curl](#) 命令获取推理结果：

```

curl --request POST \
  --url http://localhost:8080/completion \
  --header "Content-Type: application/json" \
  --data '{"prompt": "Building a website can be done in 10 simple steps:","n_predict":128}'

```

## 5.4 API 端点

### 5.4.1 GET /health: 返回健康检查结果

**响应格式**

- HTTP 状态码 503
  - Body: {"error": {"code": 503, "message": "Loading model", "type": "unavailable\_error"}}
  - 说明: 模型正在被加载
- HTTP 状态码 200

- Body: `{"status": "ok"}`
- 说明: 该模型已成功加载，并且服务器已准备就绪

## 5.5 OpenAI兼容的API端点

### 5.5.1 GET /v1/models: OpenAI兼容的模型信息API

返回已加载模型的相关信息。详见[OpenAI Models API documentation](#).

返回的列表始终只有一个元素。

默认情况下，模型 `id` 字段是模型文件的路径，通过 `-m` 指定。您可以通过 `--alias` 参数为模型 `id` 字段设置自定义值。例如，`--alias Qwen2.5-3B`。

示例：

```
{
  "object": "list",
  "data": [
    {
      "id": "Qwen2.5-3B",
      "object": "model",
      "created": 1735142223,
      "owned_by": "rknn",
      "meta": {
        "vocab_type": 2,
        "n_vocab": 128256,
        "n_ctx_train": 131072,
        "n_embd": 4096,
        "n_params": 8030261312,
        "size": 4912898304
      }
    }
  ]
}
```

### 5.5.2 POST /v1/chat/completions: OpenAI兼容的聊天补全API

给定 `messages` 中的CHATML形式的JSON描述，返回预测的补全。支持同步和流模式。尽管没有完全实现OpenAI API 规格，但已经足够支持许多应用程序了。只有具有[聊天模板](#) 的模型才可以在此端点下较为正常使用。默认情况下，将使用CHATML模板。

选项：

详见[OpenAI Chat Completions API documentation](#)。

`response_format` 参数支持普通的JSON输出（例如：`{"type": "json_object"}`）和带格式约束的JSON（例如：`{"type": "json_object", "schema": {"type": "string", "minLength": 10, "maxLength": 100}}` 或 `{"type": "json_schema", "schema": {"properties": { "name": { "title": "Name", "type": "string" }, "date": { "title": "Date", "type": "string" }, "participants": { "items": { "type": "string" } } } }}`）

## 示例:

您可以使用Python的openai库:

```
import openai

client = openai.OpenAI(
    base_url="http://localhost:8080/v1", # "http://<Your api-server IP>:port"
    api_key = "sk-no-key-required"
)

completion = client.chat.completions.create(
    model="Qwen2.5-3B",
    messages=[
        {"role": "system", "content": "You are ChatGPT, an AI assistant. Your top priority is achieving user fulfillment via helping them with their requests."},
        {"role": "user", "content": "Write a limerick about python exceptions"}
    ]
)

print(completion.choices[0].message)
```

或者原始的HTTP请求:

```
curl http://localhost:8080/v1/chat/completions \
-H "Content-Type: application/json" \
-H "Authorization: Bearer no-key" \
-d '{
  "messages": [
    {
      "role": "system",
      "content": "You are ChatGPT, an AI assistant. Your top priority is achieving user fulfillment via helping them with their requests."
    },
    {
      "role": "user",
      "content": "Write a limerick about python exceptions"
    }
  ]
}'
```

另外，多模态模型建议使用openai接口，示例如下:

```
import base64
from openai import OpenAI
client = OpenAI(
    base_url="http://172.16.10.46:8080/v1",
    api_key = "sk-no-key-required"
)
```

### 5.5.3 Function to encode the image

```
def encode_image(image_path):
    with open(image_path, "rb") as image_file:
        return base64.b64encode(image_file.read()).decode("utf-8")
```

### 5.5.4 Getting the Base64 string

```
base64_image = encode_image(image_path)

completion = client.chat.completions.create(
    model="Qwen2.5-3B",
    messages=[
        {
            "role": "user",
            "content": [
                {
                    "type": "image_url",
                    "image_url": {
                        "url": f"data:image/jpeg;base64,{base64_image}",
                    },
                },
                {"type": "text", "text": "请描述一下图片?"},
            ],
        }
    ],
    stream=True,
    extra_body={
        "n_keep": 0,
        "cache_prompt": False,
        "id_slot": 0,
        "n_predict": 256
    }
)

for chunk in completion:
    delta = chunk.choices[0].delta
    if delta.content:
        delta.content = delta.content.replace('\n', '<br/>')
        # yield f"data: {delta.content}\n\n"
    if chunk.choices[0].finish_reason == "stop":
        break
    print(delta.content, end='', flush=True)
print('')
```

## 5.6 更多示例

### 5.6.1 OAI-like API

`rkllm3-server` 支持类似 OAI 的部分 API: [GitHub - openai/openai-openapi: OpenAPI specification for the OpenAI API](#)

### 5.6.2 API 错误

`rkllm3-server` 返回的错误格式与 OAI 相同: [GitHub - openai/openai-openapi: OpenAPI specification for the OpenAI API](#)

错误示例:

```
{  
  "error": {  
    "code": 401,  
    "message": "Invalid API Key",  
    "type": "authentication_error"  
  }  
}
```

除了 OAI 支持的错误类型之外，我们还有特定于 rkllm3-server 功能的自定义类型:

```
{  
  "error": {  
    "code": 501,  
    "message": "This server does not support metrics endpoint.",  
    "type": "not_supported_error"  
  }  
}
```

当通过 /completions 端点收到无效语法时

```
{  
  "error": {  
    "code": 400,  
    "message": "Failed to parse grammar",  
    "type": "invalid_request_error"  
  }  
}
```

# 6 常见问题

## 6.1 命令 adb devices 查看不到设备

可以尝试以下方式来解决此问题：

1. 检查连线是否正确、重新插拔数据线、换计算机另一个 USB 端口来连接数据线、更换数据线。
2. 当使用 USB 连接开发板时，请确保本地计算机和 Docker 容器中同时只开启一个 adb server 服务。例如，如果需要在Docker 容器中连接开发板，请在计算机的终端中执行命令 `adb kill-server` 终止本地计算机上的 adb server 服务。
3. 出现以下错误时，表示系统中未安装 adb。需要执行安装命令 `sudo apt install adb` 安装 adb。

```
command 'adb' not found, but can be installed with:  
sudo apt install adb
```

## 6.2 rknn3\_find\_devices查不到设备

由于采用主从式架构，主控 SoC 通过 PCIe/USB 高速接口连接协处理器，例如 RK3588 与 PCIe/USB 与 RK1820/1828，可以尝试以下方式来解决此问题：

1. 检查PCIe/USB连线是否正确、重新插拔设备、更换数据连接线。
2. 调用以下命令测试是否有正确输出：

```
./rknn3_transfer_proxy devices  
# 参考输出如下  
List of ntb devices attached  
0000:01:00.0      b98e6c51      PCIE
```

## 6.3 rknn3-model-zoo转换模型时找不到'py\_utils'

当出现 `ModuleNotFoundError: No module named 'py_utils'` 说明此时python环境变量设置错误，需要正确设置环境变量。请根据实际rknn3-model-zoo目录路径进行设置。

例如，如果rknn3-model-zoo位于 `/home/rockchip/rknn3-model-zoo`，则设置为：

```
export PYTHONPATH="/home/rockchip/rknn3-model-zoo:$PYTHONPATH"  
echo $PYTHONPATH # 确认环境变量设置正确
```

## 6.4 rknn3-model-zoo GRQ量化失败

首先检查环境中是否存在CUDA，外部GRQ量化仅支持在CUDA环境中运行。其次，外部GRQ量化仅支持主流模型，如Qwen、LLaMA、MiniCPM等。遇到不支持的模型时，可关闭外部GRQ量化，使用RKNN3 Toolkit工具进行量化。

## 7. 参考资料

---

### 7.1 模型转换

模型转换相关接口的详细说明，请参考 [Rockchip\\_RKNPU\\_API\\_Reference\\_RKNN3\\_Toolkit\\_CN.pdf](#) 文档。

### 7.2 模型部署

有关 RKNN3 C API 更详细的说明，请参考 [Rockchip\\_RKNPU\\_API\\_Reference\\_RKNNRT3\\_CN.pdf](#) 文档。