

# RKNN3 用户手册

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

本文介绍Rockchip RKNN3 SDK开发。

### 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
V1.0.0	HPC/NN	2026-01-01	1.初版	熊伟

---

## 目录

### RKNN3 用户手册

#### 1. RKNN3简介

- 1.1 适用的硬件平台
- 1.2 关键字说明
- 1.3 RKNN3工具链介绍
  - 1.3.1 RKNN3软件栈整体介绍
  - 1.3.2 RKNN3-Toolkit功能介绍
  - 1.3.3 RKNN3 API以及Runtime功能介绍
- 1.4 开发流程介绍
  - 1.4.1 CNN模型开发流程介绍
  - 1.4.2 LLM模型开发流程介绍
    - 1.4.2.1 LLM模型转换
      - 1.4.2.1.1 导出ONNX模型
      - 1.4.2.1.2 导出RKNN模型
    - 1.4.2.2 LLM模型部署
  - 1.4.3 VLM模型开发流程介绍
    - 1.4.3.1 VLM模型转换
    - 1.4.3.2 VLM模型部署

#### 2. 开发环境准备

- 2.1 RKNN3 Toolkit安装
  - 2.1.1 通过Docker方式安装
    - 2.1.1.1 安装Docker工具
    - 2.1.1.2 镜像准备
    - 2.1.1.3 查询镜像信息
    - 2.1.1.4 运行镜像
    - 2.1.1.5 运行Demo
  - 2.1.2 通过Pip方式安装
    - 2.1.2.1 安装Python环境
    - 2.1.2.2 安装Miniforge工具
    - 2.1.2.3 创建RKNN3-Toolkit Conda环境
    - 2.1.2.4 安装RKNN3 Toolkit
- 2.2 硬件环境准备
  - 2.2.1 硬件清单
  - 2.2.2 开发板和连接工具介绍
  - 2.2.3 硬件连接
- 2.3 RKNN3 Model Zoo安装
  - 2.3.1 下载仓库
  - 2.3.2 指定编译器
  - 2.3.3 安装依赖
  - 2.3.4 RKNPU3环境准备
    - 2.3.4.1 rknn3\_transfer\_proxy版本确认
    - 2.3.4.2 NPU内核Driver版本确认
    - 2.3.4.3 Runtime库版本确认
    - 2.3.4.4 安装和更新

#### 2.4 示例

- 2.4.1 YOLOv6模型部署示例
  - 2.4.1.1 模型转换
  - 2.4.1.2 板端部署运行
- 2.4.2 Qwen 2.5 3B部署示例
  - 2.4.2.1 模型转换
  - 2.4.2.2 板端部署运行

#### 3. RKNN3使用说明

- 3.1 非LLM模型转换
  - 3.1.1 RKNN初始化及对象释放
  - 3.1.2 模型转换配置
  - 3.1.3 模型加载接口介绍
  - 3.1.4 构建RKNN模型
  - 3.1.5 导出RKNN模型
- 3.2 LLM模型转换
  - 3.2.1 huggingface模型转换
    - 3.2.1.1 导出 ONNX 模型
    - 3.2.1.2 导出 config 文件
    - 3.2.1.3 导出 tokenizer 文件
    - 3.2.1.4 导出 embedding 文件
  - 3.2.2 LLM模型转换配置
  - 3.2.3 LLM模型加载
- 3.3 模型评估
  - 3.3.1 模型推理和精度分析
  - 3.3.2 模型性能评估

3.3.3 模型内存评估														
3.4 板端C API推理	3.4.1 CNN/VIT等CNN模型推理	3.4.1.1 初始化上下文	3.4.1.2 加载模型	3.4.1.3 模型初始化	3.4.1.4 查询模型属性	3.4.1.5 创建张量内存	3.4.1.6 准备输入数据	3.4.1.7 内存同步	3.4.1.8 执行推理	3.4.1.9 处理输出数据	3.4.1.10 释放资源	3.4.1.11 完整示例流程	3.4.1.12 注意事项	3.4.1.13 调试工具
3.4.2 LLM模型推理	3.4.2.1 session 初始化及销毁	3.4.2.2 回调函数定义	3.4.2.3 LLM 模型推理执行	3.4.2.4 LLM 模型推理中断	3.4.2.5 Chat Template 设置	3.4.2.6 LLM 运行状态查询	3.4.2.7 KVCache 策略设置	3.4.2.8 KVCache 清理	3.4.2.9 KVCache 导入与导出	3.4.2.10 Function Calling 设置	3.4.2.11 LLM 参数设置			
3.5 RKLLM3 Server介绍	3.5.1 主要功能说明	3.5.2 使用方法	3.5.3 快速上手	3.5.4 API 端点	3.5.4.1 GET <code>/health</code> : 返回健康检查结果									
3.5.5 OpenAI兼容的API端点	3.5.5.1 GET <code>/v1/models</code> : OpenAI兼容的模型信息API	3.5.5.2 POST <code>/v1/chat/completions</code> : OpenAI兼容的聊天补全API	3.5.5.3 POST <code>/v1/embeddings</code> : OpenAI兼容的词嵌入API											
3.5.6 API 错误														
3.6 板端Python API推理	3.6.1 系统依赖说明	3.6.2 工具安装	3.6.3 基本使用流程	3.6.4 运行参考示例	3.6.5 RKNN3 Toolkit Lite API详细说明	3.6.5.1 RKNNLite初始化及对象释放	3.6.5.2 加载RKNN模型	3.6.5.3 初始化运行时环境	3.6.5.4 模型推理					
4. RKNN3高级功能	4.1 数据排列格式	4.2 NPU多核配置	4.2.1 当前多核运行限制	4.2.2 模型转换时的核心配置	4.2.3 运行时核心掩码									
4.3 动态Shape	4.3.1 动态Shape功能介绍	4.3.2 生成动态Shape的RKNN模型	4.3.3 C API部署											
4.4 RKNN3 Session高级用法	4.4.1 LLM 输入类型设置	4.4.1.1 prompt 输入	4.4.1.2 token 输入	4.4.1.3 embed 输入	4.4.1.4 multimodal 输入	4.4.1.5 aux 输入	4.4.2 Callback 设置	4.4.2.1 tokenizer callback	4.4.2.2 embed callback	4.4.2.3 result callback	4.4.2.4 sampling callback			

4.4.2.5 output callback
4.4.3 Function Calling 设置
4.4.4 KVCache 导入与导出
4.5 KVCache管理
4.5.1 KV Cache 量化
4.6 Cacheable内存一致性
4.6.1 Cacheable 内存同步方向
4.6.2 同步 Cacheable 内存
4.7 LLM 模型适配
4.7.1 ONNX 模型适配
4.7.2 tokenizer说明
4.8 自定义CPU后处理算子
4.8.1 插件接口定义
4.8.2 插件实现步骤
步骤 1：定义算子结构体
步骤 2：实现关键回调函数
步骤 3：实现注册入口函数
4.8.3 编译插件
4.8.4 插件加载与使用
4.8.5 YOLOv5 插件示例
4.8.6 注意事项与限制
5. 量化说明
5.1 量化介绍
5.1.1 量化定义
5.1.2 量化计算原理
5.1.3 量化误差
5.1.4 线性对称量化和线性非对称量化
5.1.5 Per-Layer、Per-Channel和Group量化
5.1.6 量化算法
5.2 量化配置
5.2.1 量化数据类型
5.2.2 量化算法建议
5.2.3 量化校正数据集建议
5.2.4 量化配置方法
5.3 外部GRQ量化说明
5.4 导入外部量化模型说明
5.4.1 导出 ONNX 模型与 <code>config.pkl</code>
5.4.2 量化模型配置要求
5.4.3 转换为 RKNN 模型
6 精度排查
6.1 模拟器精度排查
6.1.1 模拟器FP16精度
6.1.2 模拟器量化精度
6.2 Runtime精度排查
6.2.1 连板精度
6.2.2 Runtime精度
7 性能优化
7.1 模型性能优化前期分析流程
7.1.1 环境条件与配置检查
7.1.2 部署过程耗时分析
7.2 模型性能分析
7.2.1 获取Profile信息
7.2.2 Session运行LLM模型时，获取Profile信息
7.2.3 分析逐层耗时
7.2.4 分析CPU算子影响
7.2.5 分析NPU算子性能瓶颈（目前暂不支持，后续版本增加）
7.3 量化加速
7.4 图级别优化
7.4.1 非NPU OP通过图变换实现NPU化
7.4.2 利用硬件Fuse特性设计网络或图优化
7.4.3 算法等效变换或者子图单OP化
7.4.4 算子等效进行“同类项合并”、“提取公因式”
7.5 算子级别优化
7.5.1 面向DDR性能优化的OP尺寸设计（非强制）
7.5.2 高利用率模型算子的设计
7.5.3 子图融合的匹配
8. 内存使用优化
8.1 模型运行时内存组成及分析方法介绍
8.1.1 RKNN模型运行时内存组成
8.1.2 模型内存分析方法
8.2 Internal内存复用
9. 常见问题

9.1 NPU环境准备问题  
9.2 工具安装问题  
9.3 模型转换常用参数说明  
9.4 模型加载问题  
    9.4.1 ONNX模型转换常见问题  
    9.4.2 Pytorch模型转换常见问题  
    9.4.3 TensorFlow模型转换常见问题  
    9.4.4 其它  
9.5 模型量化问题  
9.6 模型转换问题  
9.7 模拟器推理及连板推理的说明  
9.8 模型评估常见问题  
9.9 C API使用常见问题

# 1. RKNN3简介

---

## 1.1 适用的硬件平台

本文档适用如下硬件平台：

- RK1820
- RK1828

## 1.2 关键字说明

- **主控SoC**: 运行用户应用的主处理器（如RK3588/RK3576），通过PCIe/USB与协处理器（如RK1820/RK1828）通信。
- **协处理器**: 由主控SoC调度，通过USB/PCIe等接口参与计算的专用加速单元，主要用于特定计算任务。
- **RKNN模型**: 运行在协处理器NPU上的模型文件，包括`.rknn`（模型结构）和`.weight`（权重数据）两个文件。
- **基础API**: 面向CNN/ViT等通用视觉模型的推理接口，包括模型加载、属性查询、内存管理、推理执行等基础功能。
- **Session API**: 面向LLM大语言模型的专用接口，提供会话管理、KV Cache、LoRA、采样控制等高级功能。
- **连板推理**: 连板推理中，PC 经过主控SoC接入 RK1820/RK1828 协处理器，推理由 PC 侧 RKNN3-Toolkit 发起，模型在协处理器 NPU 上执行，用于模型验证和精度分析。
- **RKNN3 Runtime**: 运行在RK182X上的运行时环境，负责模型执行、内存管理、多核调度等。
- **Tensor (张量)**: 表示多维数组的数据结构，包含数据内存（`rknn3_tensor_mem`）和属性信息（`rknn3_tensor_attr`）。
- **数据类型 (dtype)**: 张量元素的数据类型，如FP32、FP16、INT8、UINT8等。
- **数据布局 (layout)**: 张量在内存中的排列方式，如NCHW、NHWC、NC1HWC2等。
- **NC1HWC2**: NPU优化的内存布局格式，将通道维度分块存储（FP16为16子通道，INT8为32子通道），提升NPU访问效率。
- **量化模型**: 使用INT8等低精度数据类型的模型，需要scale和zero\_point参数进行量化/反量化。
- **非量化模型**: 使用FP16数据类型的模型，无需量化参数。
- **核心掩码 (core\_mask)**: 用于指定模型运行在哪些NPU核心上，如0x3表示使用core 0和core 1。
- **动态Shape**: 同一模型支持多组输入尺寸组合，运行时可通过`rknn3_set_shape`切换。
- **KV Cache**: 大语言模型中缓存的key-value注意力状态，用于加速自回归生成。
- **LoRA**: 低秩适配（Low-Rank Adaptation），一种高效的模型微调方法。

## 1.3 RKNN3工具链介绍

### 1.3.1 RKNN3软件栈整体介绍

RKNN3软件栈帮助用户将主流深度学习模型快速、高效地部署到协处理器上。RKNN3软件栈整体架构图如图1-1所示：

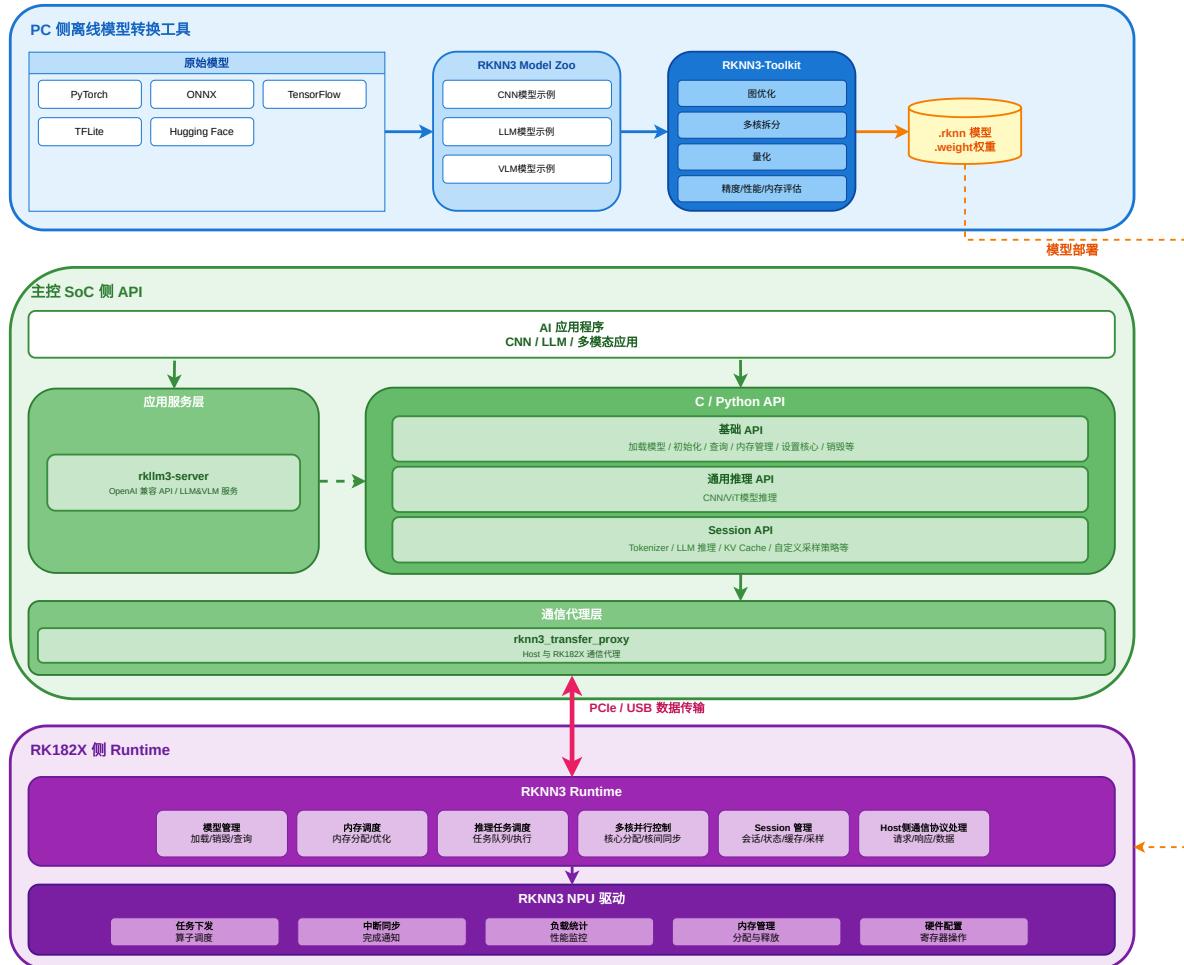


图 1-1 RKNN3 软件栈整体架构

RKNN3软件栈整体架构分为"PC侧离线模型转换工具 + 主控SoC侧API + RK182X侧Runtime & NPU驱动"三大部分，各个部分的功能说明如下：

#### 1. PC侧：提供完整的模型转换示例与工具链，包括：

- **RKNN3 Model Zoo：** 提供丰富的模型转换与部署示例，涵盖多种AI模型类型：
  - **CNN模型：** MobileNetV2、ResNet50、YOLOv5、YOLOv6、YOLOv8等视觉识别与检测模型。
  - **LLM模型：** Qwen2.5 0.5B、1.5B、3B、Qwen3 0.6B、1.7B、4B等大语言模型。
  - **VLM模型：** FastVLM、Qwen2.5 VL、Qwen3 VL等视觉语言多模态模型。
- **RKNN3-Toolkit：** 模型转换核心工具，将ONNX/PyTorch/TensorFlow等模型转换为 .rknn 和 .weight 格式，并完成图优化、多核拆分、量化、精度分析与可视化分析等功能。对于LLM/VLM等大语言模型，需要先通过RKNN3 Model Zoo中的导出脚本将模型转换为 ONNX格式，再使用RKNN3-Toolkit转换为 .rknn 和 .weight 格式完成部署。

#### 2. 主控SoC侧：提供RKNN3 API接口层与应用服务，包括：

- **rkllm3-server：** 提供OpenAI兼容API服务，支持通过标准的OpenAI API接口调用LLM和VLM模型，支持文本、图片输入和语音（暂不支持视频输入）。该服务器简化了大模型的集成和调用流程，方便与现有应用生态对接。
- **RKNN3 API：** 核心接口层，支持C和Python两种编程语言接口，按照功能类型分成以下不同系列的API：
  - **基础 API：** 模型加载、初始化、查询、内存管理、设置运行核心、销毁等基础操作。
  - **通用推理 API：** 面向CNN/ViT等通用视觉模型的推理接口。
  - **Session API：** 面向LLM的推理接口，支持KV Cache、自定义采样策略等功能。
- **rknn3\_transfer\_proxy：** 提供主控SoC与RK1820/1828协处理器之间的通信代理服务，支持PCIe和USB两种连接方式。

#### 3. RK182X侧：运行完整的推理软件栈，包括：

- **RKNN3 Runtime**: 负责模型管理、内存调度、推理任务调度、多核并行控制，以及Session会话管理等操作。
- **RKNN3 NPU 驱动**: 负责任务下发、中断同步、负载统计等底层操作。
- **通信机制**: 主控SoC与RK182X通过PCIe/USB互联，API层与Runtime层之间采用内部的通信协议进行数据交互和控制指令传递。

在RK182X典型部署形态中，应用运行在主控SoC上，通过RKNN3 API控制RK1820/RK1828协处理器上的模型加载与推理执行，实现高效的AI算力扩展。用户可以选择直接使用RKNN3 API进行模型推理，或通过rknn3-server使用OpenAI兼容接口调用大模型服务。

### 1.3.2 RKNN3-Toolkit功能介绍

RKNN3-Toolkit是为用户提供在计算机上进行模型转换、推理和性能评估的开发套件，RKNN3-Toolkit的主要框图如下：

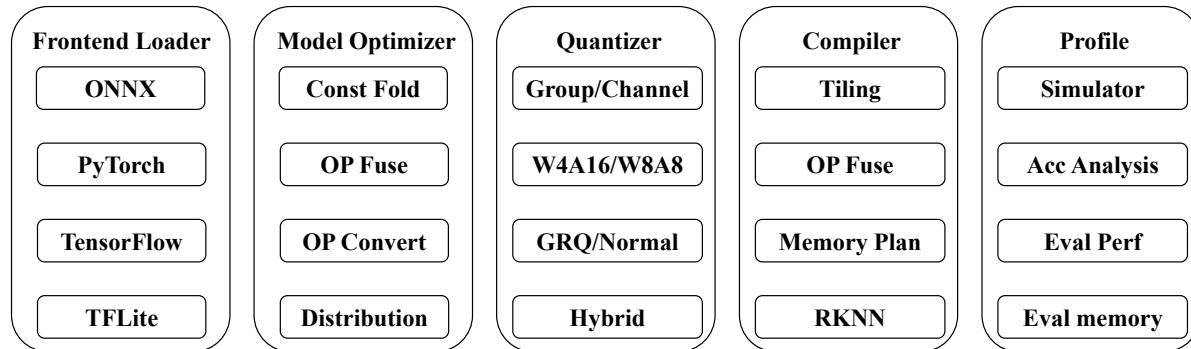


图 1-2 RKNN3-Toolkit 软件框图

通过该工具提供的Python接口可以便捷地完成以下功能：

1. 模型转换：支持将 PyTorch、ONNX、TensorFlow、TensorFlow Lite 等主流框架的模型转换为 RKNN 模型。
2. 量化功能：支持将浮点模型量化为定点模型，量化类型包括 w4a16、w6a16、w8a8、w8a16，量化方法支持 per-channel、per-group 和 per-layer。
3. 模型推理：支持将 RKNN 模型部署到指定的 NPU 设备上进行推理或在 PC 端通过模拟器仿真 NPU 运行 RKNN 模型获取推理结果。
4. 性能和内存评估：支持在指定的 NPU 设备上运行 RKNN 模型，对模型在真实硬件环境下的性能表现和内存占用情况进行评估。
5. 量化精度分析：提供量化模型各层推理结果与浮点模型推理结果之间的余弦距离和欧氏距离，用于分析量化误差来源，辅助提升量化模型精度。

### 1.3.3 RKNN3 API以及Runtime功能介绍

RKNN3提供基础API和Session API两类接口，运行在主控SoC侧，通过PCIe/USB与RK182X上的Runtime通信。RK182X上的Runtime负责模型执行、内存管理、多核调度与NPU驱动交互。

#### RK182X Runtime核心功能

- **任务调度**: 接收主控SoC侧API请求，调度推理任务到NPU核心，管理多模型并发与多核负载均衡。
- **内存管理**: 管理协处理器 DRAM，处理张量内存分配、释放与缓存同步。
- **多核支持**: 支持RK182X多达8个NPU核心的绑核调度与并行推理。
- **LLM会话管理**: 维护KV Cache状态、LoRA、采样状态等会话上下文。
- **性能监控**: 统计推理耗时、Prefill/Decode耗时等指标。

#### 基础 API

面向CNN/ViT等通用视觉模型的推理接口，主要功能包括：

- **模型生命周期管理**: `rknn3_init` 初始化上下文、`rknn3_load_model_from_path/data` 加载模型与权重、`rknn3_model_init` 配置运行参数（核心掩码、优先级、超时等）、`rknn3_destroy` 释放资源。
- **模型属性查询**: 通过`rknn3_query` 接口查询输入/输出个数 (`RKNN3_QUERY_IN_OUT_NUM`)、张量属性 (`shape`、`dtype`、`layout`、量化参数等)、NPU核心数 (`RKNN3_QUERY_CORE_NUMBER`)、SDK版本、动态Shape配置、设备内存信息等。
- **内存管理**: `rknn3_create_mem` 分配张量内存（支持指定核心）；`rknn3_mem_sync` 实现CPU与设备间的缓存同步；通过`rknn3_tensor` 结构绑定内存与张量属性。
- **多核与并发**: 通过`rknn3_config.run_core_mask` 配置核心掩码（支持指定特定核心）；`rknn3_run` 同步推理、`rknn3_run_async + rknn3_wait` 异步推理。
- **动态Shape支持**: 查询模型支持的所有Shape组合，通过`rknn3_set_shape` 运行时切换输入尺寸。
- **调试与诊断**: `rknn3_dump_features` 导出逐层特征用于精度分析；`rknn3_profile_ops` 输出逐层算子性能统计；`rknn3_profile_mem` 查看各NPU核心内存占用；`rknn3_find_devices` 枚举可用设备。

#### Session API

面向LLM（大语言模型）的专用接口，提供会话化推理能力：

- **会话管理**: `rknn3_session_init` 创建会话（配置vocab、embedding维度、最大上下文长度等）、`rknn3_session_destroy` 销毁会话。
- **KV Cache管理**: 支持 `recurrent` 和 `normal` 两种缓存策略；`rknn3_session_set_kvcache_policy` 设置策略、`rknn3_session_clear_kvcache` 清理缓存（可保留system prompt）、`rknn3_session_load/save_kvcache` 加载/保存缓存状态。
- **LoRA**: `rknn3_session_enable/disable_lora` 动态启用/禁用LoRA，支持多任务微调场景。目前暂不支持。
- **采样控制**: 配置 `top-k`、`top-p`、`temperature`、`repeat_penalty` 等采样参数，控制生成质量与多样性。
- **多模态输入**: 支持文本prompt、token序列、embedding向量、图像/音频/视频等多模态输入。
- **回调机制**: 支持结果回调（逐token返回）、采样回调（自定义采样逻辑）、tokenizer回调（自定义分词器）、embedding回调、output回调等。
- **聊天模板**: `rknn3_session_set_chat_template` 配置system prompt、用户输入前缀/后缀等对话格式。

## 1.4 开发流程介绍

### 1.4.1 CNN模型开发流程介绍

CNN/ViT 等视觉模型的典型开发流程分为三阶段：

1. **模型转换 (PC侧)**
  - 准备原始模型 (ONNX/PyTorch/TensorFlow等)
  - 在RKNN3-Toolkit中配置归一化、量化与目标平台等参数
  - 通过 `rknn.build()` 完成模型构建，可开启量化以获得更高性能与更低内存占用
  - 通过 `rknn.export_rknn()` 导出 `.rknn` 和 `.weight` 模型文件
2. **模型评估 (PC连板)**
  - 通过USB/PCIe连接主控SoC与RK182X开发板，进行连板推理
  - 验证功能正确性，完善前后处理逻辑
  - 精度分析：通过 simulator 或连板推理方式，分析并获取量化误差及板端推理误差的来源。
  - 性能分析：评估推理时算子耗时，优化运行核心调度
3. **板端部署运行 (主控SoC+RK182X)**
  - 在主控SoC应用中加载 `.rknn` 和 `.weight` 文件
  - 初始化RKNN3 Runtime，配置多核掩码等运行参数
  - 实现输入前处理（数据类型转换、布局转换）和输出后处理（反量化、业务逻辑）
  - 集成到实际业务流程，实现端到端推理

### 1.4.2 LLM模型开发流程介绍

LLM模型的开发流程与CNN模型存在一些差异，主要体现在对HuggingFace模型的特殊处理，以及LLM模型在板端部署时所使用的API也与CNN模型的不同。这里重点介绍LLM模型转换和LLM模型部署流程。

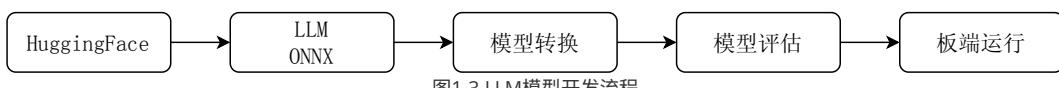


图1-3 LLM模型开发流程

#### 1.4.2.1 LLM模型转换

LLM模型转换主要包括两个步骤：首先将HuggingFace模型导出为ONNX格式，再将ONNX模型转换为RKNN格式。关于如何将HuggingFace模型转为ONNX，当前协处理器对模型输入输出的定义无特殊限制。但为了在硬件上达到更优的推理性能，根据协处理器的硬件特性，RKNN3 构建了一套 LLM 推理接口及流程，该流程要求 LLM 模型在导出 ONNX 时需要符合一定的要求，具体要求如下：

##### 部署示例生成文件说明

rknn3-model-zoo提供经典模型转换和板端部署示例。转换脚本主要包含 `export_llm.py`、`export_vision.py`、`export_rknn.py` 三部分，对应的功能及生成的文件如下：

1. `export_llm.py` 从 HugggingFace 格式的 LLM文件导出 ONNX模型和相关配置文件，生成文件含义如下
  - **model\_llm.onnx**: 适配RKNN3-Toolkit的ONNX模型文件，其中默认使用外部grq量化算法，模型本身自带量化参数。当不使用外部 grq 量化算法时，可使用RKNN3-Toolkit工具在模型转换过程中进行量化。
  - **model\_llm.config.pkl**: 使用RKNN3-Toolkit转换LLM模型需要的配置文件，其中包括chat template、vocab size、hidden size等描述模型规格的参数，以及使用外部grq量化算法的相关信息。

- **model\_llm.embed.bin**: 模型Embedding层的权重，采用Float16存储。当前Embedding层的操作集中在主控端，主控端基于 Token Ids 查询到对应的 Embeds 数据并传输给协处理器。当模型的 Embedding 层操作比较复杂时，可以没有 model\_llm.embed.bin 文件，此时用户需要在应用业务逻辑中自行管理处理 Embedding 操作。
- **model\_llm.tokenizer.gguf**: 模型Tokenizer文件。RKNN3板端部署时默认支持Tokenizer功能，需要使用对应Tokenizer文件进行初始化。当使用自定义Tokenizer时可忽略此文件。

2.export\_vision.py支持从HuggingFace导出Vision的ONNX模型，具体包括：

- **model\_vision.onnx**: 适配RKNN3-Toolkit的ONNX模型文件，不支持使用外部量化。

3.export\_rknn.py支持将ONNX模型转换为RKNN，具体包括：

- **model.rknn**: RKNN模型文件。
- **model.weight**: RKNN模型的权重文件。

#### ONNX 格式下LLM模型的输入输出限制及含义

ONNX必须包含且只包含以下4个输入

- **input\_ids**: 输入的 token id，维度为 [1, sequence]，数据类型为 int64。
- **attention\_mask**: 输入的因果推理mask，维度为 [1, sequence]，数据类型为 float32。主要用于控制固定输入shape情况下、补齐空输入对推理的影响。
- **position\_ids**: 配置输入的位置编码 id，控制 rope 的采样点。
- **logit\_id\_to\_keep**: 保留对应 id 的 logits 输出结果，主要用于 prefill 阶段保留最后一个有效 logits 输出，减少冗余的计算、内存开销。

ONNX模型输出:

- 输出为单个 logit 的输出。

#### KV Cache

RKNN3-Toolkit 在加载 LLM 模型时，会自动根据 Attention op 构筑 KV Cache 的内部管理逻辑，包含推理性、Cache量化等优化，**无需用户自行管理 KV Cache**。功能启用时，要求 ONNX 模型不能有 KV Cache 输入输出，通常可将 llm 模型 config 中的 use\_cache 配置为 False 进行模型导出，此时导出 ONNX 模型不会有 KV Cache 输入输出。

#### 支持多种输入 Shape

- RKNN3 暂不支持实时动态 shape，目前支持静态多组 shape，用户需提前指定模型应用中所需的 shape。例如在 LLM 模型的支持场景中，prefill 阶段需要采用长度为 N 的输入进行推理，以获取更快的推理性；在 generate 阶段采用长度为 1 的输入进行因果推理。
- 通过在 `rknn.load_llm` 接口配置 `seq_len` 参数，例如配置 `seq_len = [1, 64]`，生成输入 token 长度为 64 的 prefill 模型、以及输入 token 长度为 1 的 decoder 模型

##### 1.4.2.1.1 导出ONNX模型

1. 在将HuggingFace模型转成 ONNX 模型时，需要先构建 LLM Torch 模型，然后再将 Torch 模型转成 ONNX 模型。在 rknn3-model-zoo 工程中，这部分过程代码封装在 `causal_llm_to_onnx` 接口中，示例代码：

```
causal_llm_to_onnx(model, args)
```

其核心代码如下：

```

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

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。

2. 导出 config 文件。配置文件是使用 RKNN3-Toolkit 转换 RKNN 模型需要的配置文件，其中包括 chat template、vocab size、hidden size 等描述模型规格的参数，以及使用外部 grq 量化算法的相关信息。示例代码：

```

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

```

3. 导出tokenizer文件。tokenizer 文件包含了分词和词表等信息，即对应 LLM 模型的 tokenizer.json 信息。示例代码：

```

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

```

4. 导出embedding文件。embedding 文件包含了 LLM 模型中的 embedding 数据 (Float16 类型)，即词表向量。词表向量按照 Torch 模型中的 embedding 权重的原始排布存储，即 shape 信息为 (vocab\_size, embedding\_dim)。示例代码：

```

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

```

#### 1.4.2.1.2 导出RKNN模型

##### 1. 加载ONNX模型

针对 LLM 模型，`rknn.config` 新增了 `llm_config` 参数，该参数为 Python 字典类型。用户可先载入默认配置，再根据实际需求修改其中的相关选项。LLM 模型加载目前只支持 ONNX 格式的模型文件，用户可以通过 `rknn.load_llm` 接口加载 LLM 模型、开启 rknnpu 工具链针对 LLM 模型的一系列优化功能。`rknn.config()` 和 `rknn.load_llm()` 接口详细说明参见[3.1.2 模型转换配置](#)、[3.2.3 LLM模型加载](#)。示例代码：

```

from rknn.api import RKNN, DEFAULT_RKNN_LLM_CONFIG
rknn = RKNN(verbose=True)
my_config = DEFAULT_RKNN_LLM_CONFIG.copy()
print('LLM config is:', my_config)
rknn.config(target_platform='rk1820', llm_config=my_config)

# args.onnx_path 是 ONNX 模型文件路径，例如 './Qwen2.5-3B.onnx'
# args.config 是 config 文件路径，例如 './Qwen2.5-3B.config.pkl'
ret = rknn.load_llm(model=args.onnx_path, config=args.config)

```

## 2. 量化并导出RKNN模型

用户通过 `rknn.build()` 接口构建了 RKNN 模型后，可以通过 `rknn.export_rknn()` 接口导出RKNN模型和权重文件（`.rknn` 和 `.weight` 后缀），以便后续模型的部署。示例代码：

```
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的所有文件，包括 `xxx.rknn`、`xxx.weight`、`xxx.embed.bin`、`xxx.tokenizer.gguf`。

### 1.4.2.2 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 `Session` 管理的 LLM 模型部署流程如下图所示：

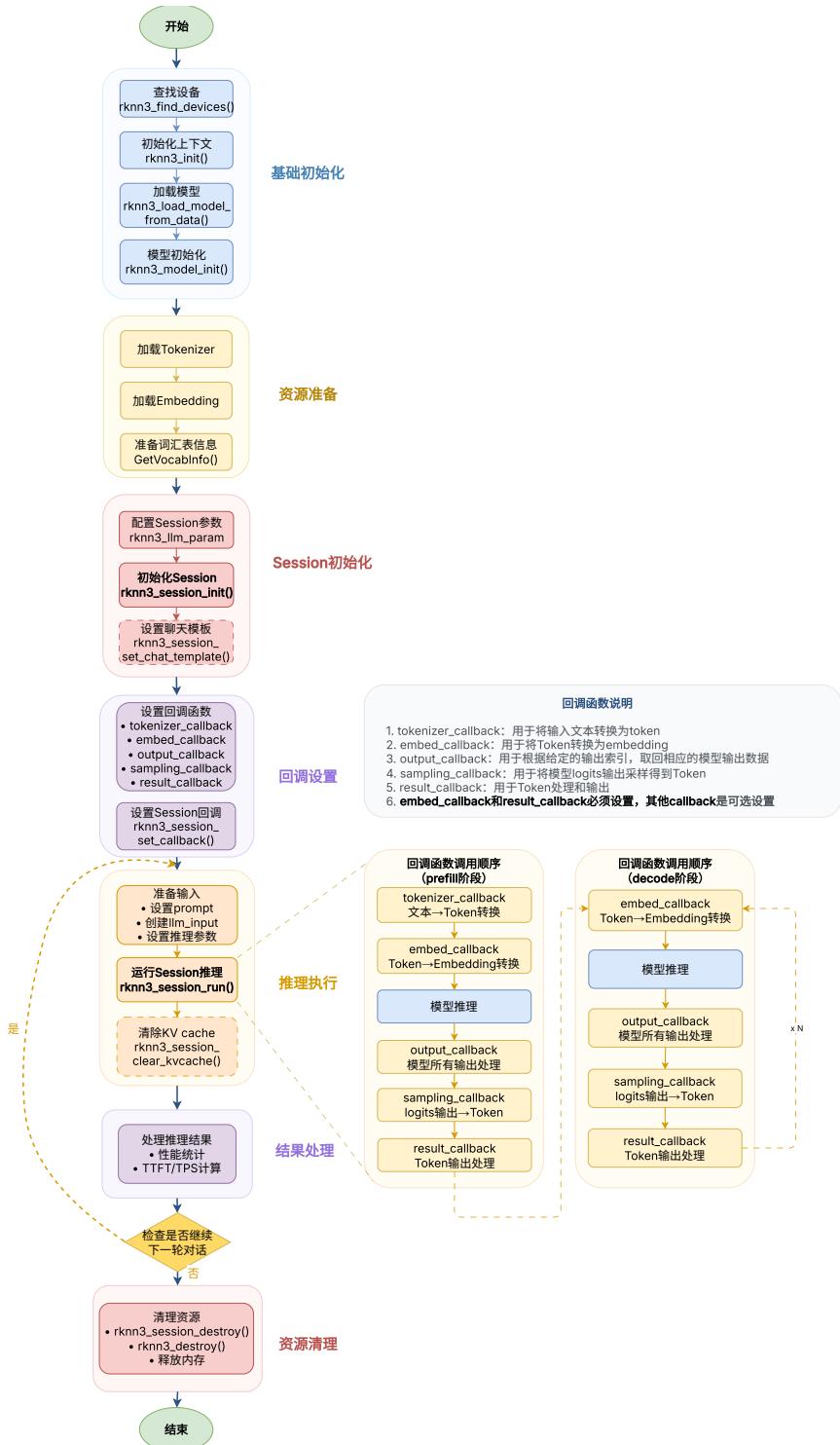
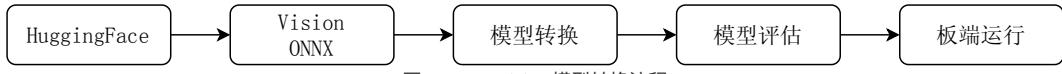


图1-4 基于 Session 管理的 LLM 模型部署流程

### 1.4.3 VLM模型开发流程介绍

#### 1.4.3.1 VLM模型转换

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，LLM部分的导出流程与前述一致，以下将重点介绍 Vision 模型的导出方法。



从HuggingFace模型转成ONNX模型可以参考rknn3-model-zoo的Qwen2\_5\_VL示例，具体步骤如下：

- 构建VLM Torch模型，这一步对于大部分VLM都通用

```
from transformers import AutoConfig, Qwen2_5_VLForConditionalGeneration

config = AutoConfig.from_pretrained(args.model_path, **kwargs)
model = Qwen2_5_VLForConditionalGeneration.from_pretrained(args.model_path, **kwargs)
```

- 注意：某些模型为了兼容RKNN推理框架，可能对其原始模型代码进行了定制化修改。因此，并非所有模型都直接通过transformers库加载。例如，Qwen2\_5\_VL添加num\_logits\_to\_keep输入和Qwen3\_VL引入deepstack的修改等，都需在原始代码中插入相应改动。这类定制化模型通常需要使用修改后的本地代码加载，而非直接调用Hugging Face的标准接口。

- 从VLM的Torch模型转ONNX，对应export\_qwen2\_vl\_vision接口，内部核心代码如下：

```
def export_qwen2_vl_vision(model, args, patch_size=14):
    class qwen2vl(torch.nn.Module):
        def __init__(self, vl_m, in_h, in_w, patch_size):
            super(qwen2vl, self).__init__()
            self.vl_m = vl_m

            self.grid_t = 1
            self.merge_size = 2
            self.channel = 3
            self.temporal_patch_size = 2
            self.patch_size = patch_size

            align_size = self.merge_size * self.patch_size
            align_h = (in_h + align_size - 1) // align_size * align_size
            align_w = (in_w + align_size - 1) // align_size * align_size
            self.in_h = align_h
            self.in_w = align_w

        def forward(self, pixel, grid_thw):
            patches = pixel.repeat(self.temporal_patch_size, 1, 1, 1)
            h = pixel.shape[2]
            w = pixel.shape[3]
            patches = patches.reshape(self.grid_t, self.temporal_patch_size, self.channel,
                                     h//self.merge_size//self.patch_size, self.merge_size, self.patch_size,
                                     w//self.merge_size//self.patch_size, self.merge_size, self.patch_size)
            patches = patches.permute(0, 3, 6, 4, 7, 2, 1, 5, 8)
            flatten_patches = patches.reshape(self.grid_t * h//self.patch_size * w//self.patch_size,
                                             self.channel * self.temporal_patch_size * self.patch_size * self.patch_size)
            return self.vl_m(flatten_patches, grid_thw)

        in_h = args.img_h_size
        in_w = args.img_w_size
        model = qwen2vl(model, in_h, in_w, patch_size)

        fake_input = torch.randn(1, 3, in_h, in_w)
        grid_thw = torch.tensor([[1, in_h//model.patch_size, in_w//model.patch_size]], dtype=torch.int64)

        torch.onnx.export(model, (fake_input, grid_thw), args.export_vision_path, input_names=['pixel',
                                              'grid_thw'], dynamic_axes={'pixel': {2: 'height', 3: 'width'}}, opset_version=17)
```

- 注意：不同VLM的视觉模块在前处理和后处理逻辑上存在差异。若需支持其他模型，需为其编写适配的前后处理接口。

#### 1.4.3.2 VLM模型部署

VLM 模型的部署通常分为两个独立部分：vision模型的部署和LLM模型的部署。在使用 rknn3-model-zoo 工程导出多模态模型时，会生成多个 `.onnx` 文件。例如，FastVLM 模型会导出以下两个文件：

- FastVLM-vision.onnx：用于图像特征提取的vision编码器部分。
- FastVLM-llm.onnx：用于文本生成的 LLM 部分；

在转换为 RKNN 模型时，这两部分需使用不同的加载接口：

- vision模型使用 `load_onnx` 接口；
- LLM 模型使用专用的 `load_llm` 接口。

转RKNN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《Rockchip\_RKNPU\_API\_Reference\_RKNNRT3》。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);
```

## 2. 开发环境准备

### 2.1 RKNN3 Toolkit安装

本章节提供两种RKNN3-Toolkit安装方式，通过Docker或pip方式安装，用户可根据需求选择任意一种方式进行安装。RKNN3 Toolkit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。

下载参考命令如下：

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

# 工程整体目录结构如下：
├── rknn3-toolkit
│   ├── doc
│   ├── rknn3-toolkit
│   │   ├── decker
│   │   └── packages
│   ...
│   ├── rknn3-toolkit-lite
│   └── rknn3-runtime
│       ├── rknn3-api
│       ├── rknn3_transfer_proxy
│       └── rkllm3-server
└── ...
```

#### 2.1.1 通过Docker方式安装

##### 2.1.1.1 安装Docker工具

已安装Docker工具的用户可跳过此步骤，未安装的用户请根据官方手册进行安装。Docker官方安装手册链接：<https://docs.docker.com/engine/install/>。

注意事项：需要将用户添加到docker用户组。

```
# 创建docker用户组
sudo groupadd docker

# 把当前用户加入docker用户组
sudo usermod -aG docker $USER

# 更新激活docker用户组
newgrp docker

# 验证不需要sudo执行docker命令
docker run hello-world
```

正确运行结果如下：

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
719385e32844: Pull complete
Digest: sha256:88ec0acaa3ec199d3b7eaf73588f4518c25f9d34f58ce9a0df68429c5af48e8d
Status: Downloaded newer image for hello-world:latest
Hello from Docker
```

##### 2.1.1.2 镜像准备

本节介绍两种RKNN3-Toolkit镜像环境准备方式，可任选一种方式进行操作。

###### 1) 通过Dockerfile创建镜像环境

在RKNN3-Toolkit工程中docker/docker\_file文件夹下，提供了构建RKNN3-Toolkit开发环境的Dockerfile文件，用户通过Docker命令创建镜像，如下所示。

```
# 注意：以下 xx 和 x.x.x 代表版本号，请根据实际数值进行替换
cd docker/docker_file/ubuntu_22_04_cp310
docker build -f Dockerfile_ubuntu_22_04_for_cp310 -t rknn3_toolkit:x.x.x-cp310.
```

## 2) 加载已打包完整开发环境的 Docker 镜像

在RKNN3-Toolkit工程中docker/docker\_image文件夹下，提供了 rknn3-toolkit-x.x.x-cp310-docker.tar.gz 镜像文件，用户通过 Docker命令导入镜像，如下所示。

```
# 注意：以下 xx 和 x.x.x 代表版本号，请根据实际数值进行替换  
cd docker/docker_image  
docker load -i rknn3-toolkit-x.x.x-cp310-docker.tar.gz
```

### 2.1.1.3 查询镜像信息

创建或加载镜像成功后，查看Docker的镜像信息。

```
docker images
```

相应的RKNN3-Toolkit镜像信息显示。

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
rknn3-toolkit	x.x.x-cp310	xxxxxxxxxxxx	1 hours ago	5.89GB

### 2.1.1.4 运行镜像

执行以下命令运行Docker镜像，运行后将进入镜像的bash环境。

```
docker run -t -i --privileged -v /dev:/dev rknn3-toolkit:x.x.x-cpxx /bin/bash
```

将 examples 文件夹代码映射进 Docker 环境，可通过附加 `-v` 参数实现。

```
docker run -t -i --privileged -v /dev:/dev -v /rknn3-toolkit/examples:/examples rknn3-toolkit:x.x.x-cp310 /bin/bash
```

### 2.1.1.5 运行Demo

```
cd examples/onnx/yolov5  
python test.py
```

脚本运行成功后结果如下。

```
class      score      xmin, ymin, xmax, ymax  
-----  
person    0.881      [ 209, 244, 286, 506]  
person    0.864      [ 478, 238, 559, 526]  
person    0.829      [ 110, 238, 230, 534]  
person    0.323      [ 79, 353, 122, 517]  
bus       0.705      [ 92, 131, 555, 465]  
Save results to result.jpg!
```

## 2.1.2 通过Pip方式安装

### 2.1.2.1 安装Python环境

若已安装Python环境，则可省略此步骤。

```
sudo apt-get update  
sudo apt-get install python3 python3-dev python3-pip  
sudo apt-get install libxslt1-dev zlib1g zlib1g-dev libglib2.0-0 libsm6 libgl1-mesa-glx libprotobuf-dev gcc
```

若想在本地环境（非Conda虚拟环境）中安装 RKNN3-Toolkit 工具，请在安装好Python环境后跳至[步骤2.1.2.4](#)。

### 2.1.2.2 安装Miniforge工具

如果系统中同时有多个版本的Python环境，建议使用Miniforge管理Python环境。  
检查是否安装Miniforge和版本信息，若已安装 Miniforge，可跳过本小节。

```
conda -V  
# 提示 conda: command not found 则表示未安装miniforge conda  
# 提示 例如版本 conda 23.9.0
```

下载Miniforge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
```

安装Miniforge

```
chmod 777 Miniforge3-25.3.0-1-Linux-x86_64.sh  
bash Miniforge3-25.3.0-1-Linux-x86_64.sh
```

### 2.1.2.3 创建RKNN3-Toolkit Conda环境

进入Conda base环境

```
# 创建并激活环境  
conda create -n toolkit3 python=3.10  
conda activate toolkit3
```

### 2.1.2.4 安装RKNN3 Toolkit

通过本地 wheel 包安装

```
# 请根据不同的 python 版本及处理器架构, 选择不同的 requirements 文件:  
# cpxxx为Python版本号, x.x.x为rknn3-toolkit版本号  
pip install -r packages/requirements_cpxxx-x.x.x.txt  
  
# 安装 RKNN3 Toolkit  
# 请根据不同的 python 版本及处理器架构, 选择不同的 wheel 安装包文件:  
# 其中 x.x.x 是 RKNN3 Toolkit 版本号, cpox 是 python 版本号  
pip install packages/x86_64/rknn3_toolkit-x.x.x-cpox-cpox-manylinux_2_17_x86_64.manylinux2014_x86_64.whl
```

若执行以下命令没有报错，则安装成功。

```
python3  
>>> from rknn.api import RKNN
```

## 2.2 硬件环境准备

### 2.2.1 硬件清单

- 计算机 × 1 (Ubuntu20.04/Ubuntu22.04)
- 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

### 2.2.2 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

### 2.2.3 硬件连接

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[2.3.4 RKNPU3环境准备](#)。

## 2.3 RKNN3 Model Zoo安装

RKNN3 Model Zoo提供了丰富的模型转换示例，涵盖CNN、LLM、VLM等多种AI模型类型。

### 2.3.1 下载仓库

在计算机的终端窗口（命令行界面）中，执行以下命令：

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

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  
│   ├── yolov5          # YOLOv5示例  
│   ├── yolov6          # YOLOv6示例  
│   ├── yolov8          # YOLOv8示例  
│   └── Qwen2_5          # Qwen2.5 LLM示例  
|  
└── ...  
├── utils               # 常用工具  
├── build-linux.sh      # Linux编译脚本  
└── build-android.sh    # Android编译脚本  
└── requirements.txt    # python环境依赖库
```

### 2.3.2 指定编译器

- 板端为 Android 系统

在 rknn3-model-zoo 工程下的 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=/opt/android-ndk-r19c
```

- 板端为 Linux 系统

在 rknn3-model-zoo 工程下的 build-linux.sh 脚本中，需要加入以下代码：

```
# 导入GCC 交叉编译器环境变量  
GCC_COMPILER=/opt/gcc-linaro-6.3.1-2017.05-x86_64_aarch64-linux-gnu/bin/aarch64-linux-gnu
```

有关cmake、ndk/gcc交叉编译器的下载和安装方法可参考《Rockchip\_RK182X\_Quick\_Start\_RKNN3\_SDK》。

### 2.3.3 安装依赖

在计算机的终端窗口（命令行界面）中，执行以下命令：

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

### 2.3.4 RKNPU3环境准备

要在RK1820/RK1828上运行C Demo，要求在RK3588上安装：rknn3\_transfer\_proxy、RK1820/1828 Runtime库（librknn3\_api.so）。

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

如果RK3588板端没有安装 rknn3\_transfer\_proxy、Runtime 库，或者rknn3\_transfer\_proxy和 Runtime 库的版本不一致，都需要重新安装或更新。

#### 2.3.4.1 rknn3\_transfer\_proxy版本确认

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
```

查询结果：

```
I NPUTTransfer(2970): Starting RKN3 Transfer Proxy, Transfer version x.x.x, devid = -1, pid = 2970:1341
```

X.X.X表示版本号，例如0.5.0。

#### 2.3.4.2 NPU内核Driver版本确认

有些功能或者算子的性能优化项与内核驱动版本有关，较新的内核驱动版本可支持最新的底层性能优化。所以请检查是否使用到较新的内核驱动版本。

驱动版本随开机启动日志打印如下：

```
[02_0000000630] RKNPU Driver: vx.x.x (20251204), core number: 8
```

x.x.x 表示版本号，例如0.5.0。

#### 2.3.4.3 Runtime库版本确认

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
strings /vendor/lib64/librkn3_api.so | grep -i "librkn3_api version"

# 若RK3588为linux系统
strings /usr/lib/librkn3_api.so | grep -i "librkn3_api version"
```

查询结果：

```
librkn3_api version: x.x.x (1e13fde build@2025-11-01T17:08:25)
```

x.x.x 表示版本号，例如0.5.0。

#### 2.3.4.4 安装和更新

查询rknn3\_transfer\_proxy和 Runtime 库版本，若版本号不一致请更新至同一版本。

##### 1. 安装和更新

在计算机的终端窗口（命令行界面）中，执行以下命令：

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

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

# 如果是android系统，安装 rknn3-api/Android/arm64-v8a/librkn3_api.so
adb push rknn3-api/Android/arm64-v8a/librkn3_api.so /vendor/lib64/

# 如果是linux系统，安装 rknn3-api/Linux/aarch64/librkn3_api.so
adb push rknn3-api/Linux/aarch64/librkn3_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。

## 2.4 示例

### 2.4.1 YOLOv6模型部署示例

以YOLOv6模型为例，其目录结构如下：

```

rknn3-model-zoo
├── examples
│   ├── yolov6      # YOLOv6示例
│   │   ├── 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
│   └── ...

```

#### 2.4.1.1 模型转换

##### 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：

```

cd rknn3-model-zoo/examples/yolov6/model

# 下载YOLOv6 ONNX模型
./download_model.sh

```

**注意：**

文件名带“rknn3”后缀的模型例如“yolov6n\_rknn3.onnx”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. 转换为RKNN模型

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 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

```

转换后的模型保存路径为： rknn3-model-zoo/examples/yolov6/model，模型文件为 yolov6n\_rknn3.rknn 以及 yolov6n\_rknn3.weight。

#### 2.4.1.2 板端部署运行

##### 1. 编译C Demo

在计算机的终端窗口（命令行界面）中，执行以下命令：

```
# 进入rknn3-model-zoo
cd rknn3-model-zoo

# 板端为 Android 系统，编译yolov6 C++推理程序
./build-android.sh -t rk3588 -a arm64-v8a -b Release -d yolov6

# 板端为 Linux 系统，编译yolov6 C++推理程序
./build-linux.sh -t rk3588 -a aarch64 -b Release -d yolov6
```

## 2. 推送可执行文件到板端

脚本在完成编译后会将生成的可执行程序打包在 `rknn3-model-zoo/install` 目录下。在计算机的终端窗口（命令行界面）中，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 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 rknn3-model-zoo/install/rk3588_android_arm64-v8a/rknn_yolov6_demo /data
```

## 3. 板端运行

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

# 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
```

## 4. 查看结果

运行 `rknn_yolov6_demo` 在终端会打印以下信息：

```
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
```

从板端拉取到本地查看，在本地电脑的终端中，执行以下命令：

```
adb pull /data/rknn_yolov6_demo/out.png .
```

输出结果图片如下所示：

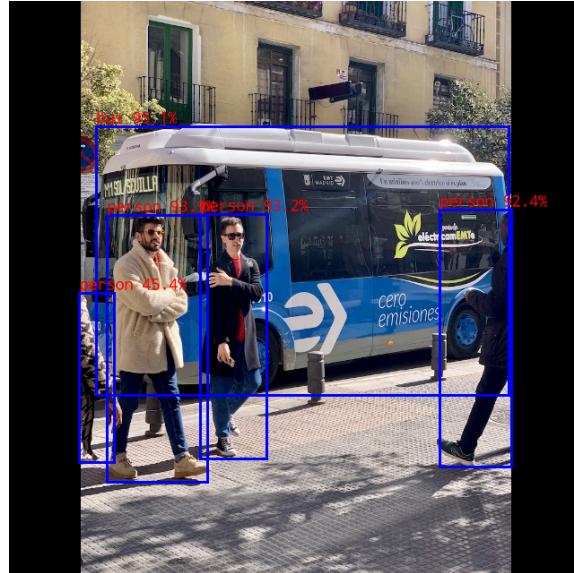


图2-6 RKNN C demo可视化结果

## 2.4.2 Qwen 2.5 3B部署示例

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
└── ...
...
```

### 2.4.2.1 模型转换

#### 1. 导出ONNX模型

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：

```
# 进入Qwen2.5目录
cd 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：

- Qwen2.5-3B-Instruct.onnx - ONNX模型文件
- Qwen2.5-3B-Instruct.config.pkl - 配置文件
- Qwen2.5-3B-Instruct.tokenizer.gguf - 分词器
- Qwen2.5-3B-Instruct.embed.bin - embedding权重

#### 2. 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 rknn3-model-zoo/examples/Qwen2_5/python/

# 转换为rknn模型
python export_rknn.py
```

#### 2.4.2.2 板端部署运行

##### 1. 编译C Demo

在计算机的终端窗口（命令行界面）中，执行以下命令：

```
# 板端为 Android 系统，编译Qwen2.5 C++推理程序  
./build-android.sh -t rk3588 -a arm64-v8a -b Release -d Qwen2_5  
  
# 板端为 Linux 系统，编译Qwen2.5 C++推理程序  
./build-linux.sh -t rk3588 -a aarch64 -b Release -d Qwen2_5
```

##### 2. 推送可执行文件到板端

脚本在完成编译后会将生成的可执行程序打包在 rknn3-model-zoo/install 目录下。在计算机的终端窗口（命令行界面）中，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 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 rknn3-model-zoo/install/rk3588_android_arm64-v8a/rknn_Qwen2_5_demo /data
```

##### 3. 板端运行

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  
  
# 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?"
```

## 3. RKNN3使用说明

### 3.1 非LLM模型转换

本节将重点介绍RKNN3-Toolkit的非LLM模型转换功能。模型转换是RKNN3-Toolkit的核心功能之一，它允许用户将各种不同框架的深度学习模型转换为RKNN格式以在RKNPU上运行。用户可参考如下模型转换流程图以理解如何进行模型转换。

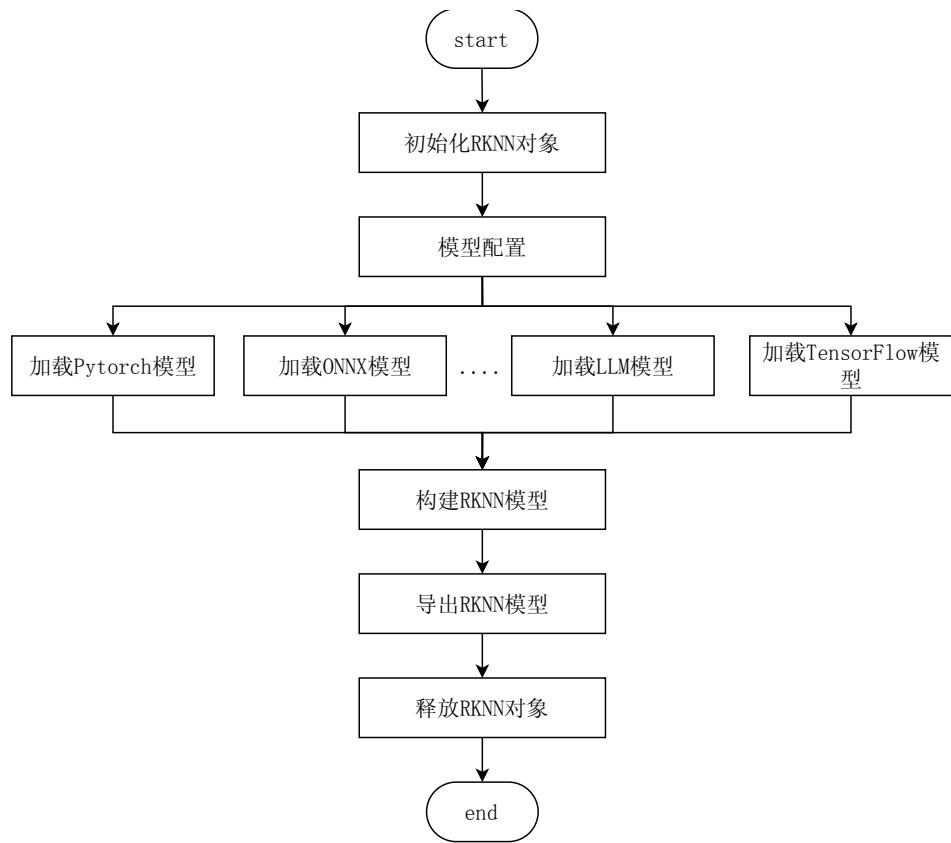


图3-1 模型转换流程图

目前RKNN3-Toolkit支持多个主流深度学习框架的模型转换，包括：

- TensorFlow（推荐版本为1.12.0~2.8.0）
- TensorFlow Lite（推荐版本为Schema version = 3）
- ONNX（推荐版本为1.10.0~1.19.0）
- PyTorch（推荐版本为2.0.0~2.7.0）

用户可以使用上述框架训练所得模型或获取预训练模型，将他们转换成RKNN格式，以遍更高效地在RKNPU平台上部署和推理。

### 3.1.1 RKNN初始化及对象释放

在这一阶段，用户需要先初始化RKNN对象，这是整个工作流程的第一步：

初始化RKNN对象：

- 使用 `RKNN()` 构造函数来初始化RKNN对象，用户可以传入参数 `verbose` 和 `verbose_file`。
- `verbose` 参数决定是否在屏幕上显示详细日志信息。
- 如果设置了 `verbose_file` 参数并且 `verbose` 为 `True`，日志信息还将写入到指定的文件中。

示例代码：

```
rknn = RKNN(verbose=True, verbose_file='./mobilenet_build.log')
```

当完成所有的RKNN相关的操作后，用户需要释放资源，这是整个工作流程的最后一步：

- 使用 `release()` 接口释放RKNN对象占用的资源。

示例代码：

```
rknn.release()
```

### 3.1.2 模型转换配置

在模型转换之前，用户需要进行一些配置以确保模型转换的正确性和性能。配置参数通过 `rknn.config()` 接口设置，包括归一化参数、量化方法、目标平台等。下面列出了一些常用的配置参数：

- `mean_values` 和 `std_values`: 用于设置输入的均值和归一化值。这些值在量化过程中使用，且C API推理阶段图片不再做均值和归一化值，减少部署耗时。
- `quant_img_RGB2BGR`: 用于控制量化阶段加载的量化校正图像是否需要进行RGB到BGR的转换，默认值为 `False`。该配置只在量化时生效，模型部署阶段不会生效，需要用户在处理输入数据时提前做好相应转换。注意：`quant_img_RGB2BGR = True` 时输入数据的处理顺序为先做RGB2BGR转换（用户自行完成）再做归一化操作（运行时内部完成）。
- `target_platform`: 用于指定RKNN模型的目标平台，支持RK1820、RK1828。
- `quantized_algorithm`: 用于指定计算每一层量化参数时采用的量化算法，支持 `grq`、`gdq`、`normal`、`mmse` 或 `kl_divergence`，默认算法为 `normal`。
- `quantized_method`: 用于指定计算每一层的量化参数时采用的量化方法，支持 `layer`、`channel` 或 `group{size}`，默认为 `channel`。
- `optimization_level`: 通过修改模型优化等级，可以关掉部分或全部模型转换过程中使用到的优化规则。该参数值为0时关闭所有优化选项。
- `quantized_dtype`: 用于指定量化类型，目前支持的量化类型有 `w8a8`、`w4a16`，该参数必须显性设置。
- `input_attrs`: 模型部署时的输入属性，默认值为 {}。格式为：`{input_name: {attr_name: attr_value, ...}, ...}`，其中 `attr_name` 的可选项为 `dtype`, `layout` 等。
- `dynamic_input`: 支持动态输入，根据用户指定的多组输入shape，来模拟动态输入的功能，默认值为 `None`。

更具体的 `rknn.config()` 接口配置请参考 [Rockchip\\_RKNPU\\_API\\_Reference\\_RKNN3\\_Toolkit\\_CN](#) 手册。

示例代码：

```
rknn.config(
    mean_values=[[103.94, 116.78, 123.68]],
    std_values=[[58.82, 58.82, 58.82]],
    quant_img_RGB2BGR=False,
    input_attrs={'pixel': {'dtype': 'uint8', 'layout': 'NHWC'}},
    target_platform='rk1820')
```

### 3.1.3 模型加载接口介绍

用户需要使用相应的加载接口导入不同框架模型文件。RKNN3-Toolkit提供了不同的加载接口，包括TensorFlow、TensorFlow Lite、ONNX、PyTorch等，下面是各种框架的模型加载接口的简要介绍：

- TensorFlow 模型加载接口：
  - 使用 `rknn.load_tensorflow()` 接口加载TensorFlow模型。
  - 需要提供TensorFlow模型文件（.pb后缀）路径、输入节点名、输入节点的形状以及输出节点名。

示例代码：

```
ret = rknn.load_tensorflow(tf_pb='./ssd_mobilenet_v1_coco_2017_11_17.pb',
                           inputs=['Preprocessor/sub'],
                           outputs=['concat', 'concat_1'],
                           input_size_list=[[1, 300, 300, 3]])
```

- TensorFlow Lite 模型加载接口：
  - 使用 `rknn.load_tflite()` 接口加载TensorFlow Lite模型。
  - 需要提供TensorFlow Lite模型文件（.tflite后缀）路径。

示例代码：

```
ret = rknn.load_tflite(model='./mobilenet_v1.tflite')
```

- ONNX 模型加载接口：
  - 使用 `rknn.load_onnx()` 接口加载ONNX模型。
  - 需要提供 ONNX 模型文件（.onnx 后缀）路径。

示例代码：

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

- PyTorch 模型加载接口：
  - 使用 `rknn.load_pytorch()` 接口加载PyTorch模型。
  - 需要提供PyTorch模型文件（.pt后缀）路径，模型必须是torchscript格式的。

示例代码：

```
ret = rknn.load_pytorch(model='./resnet18.pt', input_size_list=[[1, 3, 224, 224]])
```

用户可以根据不同框架的模型选择合适的接口进行加载，确保模型转换的正确性。

### 3.1.4 构建RKNN模型

用户加载原始模型后，下一步就是通过 `rknn.build()` 接口构建RKNN模型。构建模型时，用户可以选择是否进行量化，量化可以减小模型的大小和提高在RKNPU3上的性能。`rknn.build()` 接口参数如下：

- `do_quantization`: 该参数控制是否对模型进行量化，建议设置为 `True`。
- `dataset`: 该参数指定用于量化校准的数据集，数据集的格式是文本文件。

dataset.txt示例：

```
./imgs/ILSVRC2012_val_00000665.JPG  
./imgs/ILSVRC2012_val_00001123.JPG  
./imgs/ILSVRC2012_val_00001129.JPG  
./imgs/ILSVRC2012_val_00001284.JPG  
./imgs/ILSVRC2012_val_00003026.JPG  
./imgs/ILSVRC2012_val_00005276.JPG
```

示例代码：

```
ret = rknn.build(do_quantization=True, dataset='./dataset.txt')
```

### 3.1.5 导出RKNN模型

用户通过 `rknn.build()` 接口构建了 RKNN 模型后，可以通过 `rknn.export_rknn()` 接口导出RKNN模型文件（`.rknn` 后缀），以便后续模型的部署。`rknn.export_rknn()` 接口参数如下：

- `export_path`: 导出模型文件的路径。

示例代码：

```
ret = rknn.export_rknn(export_path='./mobilenet_v1.rknn')
```

这些接口涵盖了RKNN3-Toolkit模型转换阶段，根据不同的需求和应用场景，用户可以选择不同的模型配置和量化算法进行自定义设置，方便后续进行部署。

## 3.2 LLM模型转换

本章节所涉及的 LLM (Large Language Model) 模型，指的是基于 Transformer 架构构建、并包含 Attention 机制的神经网络模型。这类模型在推理过程中通常需要对历史上下文进行缓存，以避免重复计算。在 RKNPU3 中，这部分用于缓存上下文的内存被称为 KV Cache。

针对上述 LLM 模型特性，为了提升推理性能并降低模型使用和部署的复杂度，RKNPU3 工具链在 Attention 模块中内嵌并支持 KV Cache 管理机制，并提供一系列面向 LLM 推理场景的定制化优化能力。本章节将围绕这些优化功能，详细介绍其配置方法以及开启方式。

目前 RKNPU3 允许模型带有显性的 KV Cache 输入及输出，但为了性能及内存的优化，我们建议用户尽可能采用 RKNPU3 内嵌 KV Cache 的功能。

### 3.2.1 huggingface模型转换

RKNN3 构建了一套 LLM 推理接口及流程，该流程要求 LLM 模型在导出 ONNX 时需要符合一定的要求，对于 LLM 模型，需要将 tokenizer 和 embedding 过程从模型中拆分出来，并保存模型的配置文件。

LLM 模型的转换依赖 rknn3-model-zoo 工程，不同的 LLM 模型的模型转换代码可能有差异，但整个转换流程主要可以分成 4 步：导出 ONNX 模型、导出 tokenizer 文件、导出 embedding 文件、导出 config 文件。

#### 3.2.1.1 导出 ONNX 模型

在将 LLM 模型转成 ONNX 模型时，需要先构建 LLM Torch 模型，然后再将 Torch 模型转成 ONNX 模型。在 rknn3-model-zoo 工程中，我们将这部分过程代码封装在 `causal_llm_to_onnx` 接口中，示例代码：

```
causal_llm_to_onnx(model, args)
```

其核心代码如下：

```

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

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,
)

```

### 3.2.1.2 导出 config 文件

config 文件是使用 RKNN3-Toolkit 转换 RKNN 模型需要的配置文件，其中包括 chat template、vocab size、hidden size 等描述模型规格的参数，以及使用外部 grq 量化算法的相关信息。

示例代码：

```

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

```

### 3.2.1.3 导出 tokenizer 文件

tokenizer 的作用是将文本转成 token id、将 token id 转成文本以及解析词表信息。在板端推理时需要用到模型对应的 `tokenizer.gguf` 文件以及 `libtokenizer.a` 库。

- `tokenizer.gguf`: `tokenizer.gguf` 文件在 LLM 模型转换时，通过 `export_tokenizer` 接口导出，其中包含了分词和词表等信息，即对应 LLM 模型的 `tokenizer.json` 信息。
- `libtokenizer.a`: `libtokenizer.a` 库负责将 `tokenizer.gguf` 文件中的词表信息进行解析，并提供 `文本转 token id` 和 `token id 转文本` 等接口。`rknn3-model-zoo` 使用的是 `https://github.com/ggml-org/llama.cpp` 的 tokenizer，在 `rknn3-model-zoo` 的路径为 `rknn3_model_zoo/tokenizer`。如果有调整 tokenizer 库的需要，可以修改 `rknn3_model_zoo/tokenizer/src` 中的代码，并重新编译出新的 `libtokenizer.a`。

导出 `tokenizer.gguf` 文件的示例代码：

```

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

```

### 3.2.1.4 导出 embedding 文件

embedding 的作用是将 token id 转成词表向量。embedding 文件包含了 LLM 模型中的 embedding 数据 (Float16 类型)，即词表向量。词表向量按照 Torch 模型中的 embedding 权重的原始排布存储，即 shape 信息为 (vocab\_size, embedding\_dim)。

示例代码：

```

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

```

### 3.2.2 LLM模型转换配置

针对 LLM 模型，`rknn.config` 新增了 `llm_config` 参数，该参数为 Python 字典类型。用户可先载入默认配置，再根据实际需求修改其中的相关选项。

```
from rknn.api import RKNN, DEFAULT_RKNN_LLM_CONFIG
rknn = RKNN(verbose=True)
my_config = DEFAULT_RKNN_LLM_CONFIG.copy()
print('LLM config is:', my_config)
rknn.config(target_platform='rk1820', llm_config=my_config)
```

默认配置具体如下：

```
DEFAULT_RKNN_LLM_CONFIG = {
    'attention_config': [
        {
            'mask_name': 'attention_mask',
            'belong_to_this_config': True,
            'kvcache_buffer_len': 1024,
            'kvcache_dtype': 'Float16',
            'attention_type': 'FullAttention',
            'sliding_window_size': -1,
            'position_name': 'position_ids',
            'max_position_embeddings': 8192,
            'mrope_type': None,
            'mrope_section': None,
            'mrope_new_id_name': None,
        }
    ],
    'llm_head': [
        {
            'name': 'auto',
            'heads_in_model': 1,
            'device': 'RK1820',
            'quantized_dtype': 'w6a16',
            'quantized_method': 'group32',
        }
    ],
    'vocab': [
        {
            'index_name': 'auto',
            'multiple_embeddings_in_model': False,
            'tie_embeddings': False,
            'store_on_compute_device': False,
        }
    ],
    'keep_one_logit': [
        {
            'output_name': None,
            'idx_name': 'num_logits_to_keep',
            'axis': None,
        }
    ],
}
```

LLM config 主要有4个字典子项，分别为 `attention_config`, `llm_head`, `vocab`, `keep_one_logit`。

**attention\_config**: 用于配置 KV Cache 缓存机制、配置 Attention op 的推理机制，接受类型为列表，列表中允许存在多个字典指定多种 Attention 结构的具体参数。Attention\_config 的字典定义如下：

配置	功能	数据类型:默认值	备注
mask_name	指定 mask 名称，在后续流程中 Attention op 会通过识别 mask 输入来确定该 Attention op 所属的相关配置。 该参数不可缺省！	String: 'attention_mask'	不用类型的 Attention 结构不允许共享同一个 mask 输入
kvcache_buffer_len	指定 KV Cache 缓存长度。推理时若上下文长度超过设定长度时，遵循先进先出的原则，进行循环覆盖。配置数值越大时，KV Cache 占用内存越大，长文本任务的表现理论上会更好。	Int:1024	数值要求32对齐

配置	功能	数据类型:默认值	备注
kvcache_dtype	指定上下文保存及计算数据类型，影响推理性能及缓存内存占用大小。 内存占用上, Float16 > Int8_to_FP16 > Int4_to_FP16	String: 'Float16'	支持 Float16, Int8_to_FP16, Int4_to_FP16
attention_type	指定 Attention 类型	String: 'FullAttention'	支持 FullAttention, SlidingAttention. (当前版本暂不支持 SlidingAttention)
sliding_window_size	指定 Sliding Attention 的滑窗长度。当 attention 类型为 FullAttention 时，此参数不生效	Int: -1	/
position_name	指定 position id 输入的名字，用于提取位置编码特征值。对于早期的模型，可能不存在位置编码，此时允许缺省，配置为 None.	String: 'position_ids'	不同类型的 Attention 结构不允许共享同一个 position_ids 输入
max_position_embeddings	指定位置编码的最大长度。推理时若上下文长度超过设定长度时，会抛出错误。	Int: 8192	数值要求32对齐
mrope_type	当 LLM 模型为特殊的 mrope 类型时，可以配置并开启 mrope 功能。目前支持 ['Qwen2.5-VL', 'Qwen3-VL'] 类型。注意，开启 mrope 时必须配置 position_name，且这个 position_name 是单维的输入，开启 mrope 功能后 toolkit 会自动生成一个 [seq_len, 3] 维度的输入	Strings: None	/
mrope_section	配置 mrope section 信息。对于 Qwen2.5-VL、Qwen3-VL 模型，可在模型的配置 config.json 中找到对应的配置值。	list: None	/
mrope_new_id_name	配置 mrope 新生成的 position 名称。对于当前支持的 ['Qwen2.5-VL', 'Qwen3-VL']，会生成维度为 [seq_len, 3] 的新输入。	Strings: None	/

**lilm\_head:** 用于配置并开启 LLM 模型 Head 层的优化行为。(Head层通常为单一一个参数量较大的Linear层)。暂不支持多个 LLM Head 层，后续会完善并支持。

配置名	功能	数据类型:默认值	备注
name	配置 lilm head 层的权重名称	String: auto	目前仅支持 auto，暂不支持指定
device	配置 lilm head 层的计算设备	String: rk1820	支持 rk1820, rk3588, rk3576
quantized_dtype	配置 lilm head 层的量化类型。lilm head 层规模较大，采用量化可以有效降低内存占用大小	String: w6a16	支持 w16a16, w4a16, w6a16, w8a8. 当平台为 rk3576 或 rk3588 时，会被强制更改为 w8a8.
quantized_method	配置 lilm head 层的量化方式。	String: group32	支持 layer, channel, group{Size}. 当平台为 rk3576 或 rk3588 时，会被强制更改为 channel.

**vocab:** 用于配置并开启 LLM 模型对字典层的优化行为。(当前暂未实装该参数对应的功能)

配置名	功能	数据类型:默认值	备注
index_name	配置 vocab 层的 index 输入名称	String: auto	目前仅支持 auto，暂不支持指定

配置名	功能	数据类型:默认值	备注
tie_embedding	配置 vocab 层的 tie embedding 参数, tie_embedding 为 True 时意味着 vocab 和 llm_head 共享同一个数组。开启后则不再需要额外保存 vocab, 但由于 llm_head 存在量化行为, 反量化为 float16 的 embedding 时会降低推理性能。	Bool: False	当前版本暂不支持
store_on_compute_device	配置 vocab 是否保存在计算设备上。vocab 保存在计算设备上时可以减少每轮 LLM 推理所需要的传输耗时、但会大幅增加计算设备的内存占用。	Bool: False	当前版本暂不支持

**keep\_one\_logit:** 用于配置并开启对输出的指定轴进行取单值行为（插入gather op）。对于大多数 LLM 模型，在输出采样时只需要对最后一个有效token输出进行采样即可。以输入token 序列长度为 100 为例，则输出的第100个序列为有效输出，其他输出均被舍弃。故可以将这个舍弃行为提前到 llm\_head 前面，减少 llm head 线性 Linear 层的计算量、减少计算过程中所需要的内存占用。

配置名	功能	数据类型:默认值	备注
output_name	指定输出名称	String: None	为None 时则不生效
idx_name	生成 gather indices 输入的名称，并加入到模型的输入	String: num_logit_to_keep	/
axis	指定 gather 生效轴	Int: None	/

### 3.2.3 LLM模型加载

LLM 模型加载目前只支持 ONNX 格式的模型文件，用户可以通过 `rknn.load_llm` 接口加载 LLM 模型、开启 rknnpu 工具链针对 LLM 模型的一系列优化功能。

API 名称	<b>load_llm</b>
功能	加载 ONNX 模型、开启 LLM 模型的优化功能
参数	model: ONNX 模型的路径
	config: LLM原始模型相关配置文件路径, 配置内容包括但不限于 tokenizer 信息、system prompt 信息
	seq_lens: 指定 LLM 模型转为 RKNN 模型后, 单次推理支持的输入token数
返回值	int: 返回0表示成功, 非0表示失败
注意	当前 rknnpu 工具链针对 LLM 模型的优化及配置参数, 请参考 3.2.2 与 4.7 章节

示例：

```
ret = rknn.load_llm(model='./Qwen2.5-3B.onnx',
                     config='./Qwen2.5-3B.config.pkl',
                     seq_lens=[1, 128])
```

## 3.3 模型评估

### 3.3.1 模型推理和精度分析

原始模型转换为RKNN模型后，可在模拟器或开发板上进行推理，对推理结果进行后处理，检验其是否正确。若推理结果不正确，可以对量化模型进行精度分析和查看前后处理流程的正确性。模型推理阶段使用到的主要接口相关参数说明如下：

运行时初始化接口 `rknn.init_runtime()`：

API 名称	<b>init_runtime</b>
功能	初始化运行时环境
参数	target: 目标硬件平台。默认值 None，推理在模拟器上进行。如果要获取板端实际推理结果，则该参数需要填入相应的平台 (RK1820、RK1828)。

<b>API 名称</b>	<b>init_runtime</b>
	device_id: 设备编号。默认值为 None。若设置target，则选择所设置ID对应的设备进行推理。如果电脑连接多台设备连板推理时，需要指定填入相应的设备ID。若通过网络adb连接设备进行模型推理，则需要用户手动执行命令 adb connect [IP] 连接网络设备后再填入设备编号，通常为 [IP] 或 [IP:Port] 的形式。使用 adb devices 命令可以列出所有已连接设备
	core_mask: 模型运行时的核心数，默认值为-1。使用模拟器推理时忽略该参数，板端实际推理时需要指定核心数，对于 RK1820、RK1828平台核心数范围为0x1-0xff
<b>返回 值</b>	int: 返回0表示成功，非0表示失败

推理接口 `rknn.inference()`:

<b>API 名 称</b>	<b>inference</b>
<b>功能</b>	对RKNN模型进行前向推理
<b>参数</b>	inputs: 输入数据列表，格式为ndarray
	data_format: 输入数据的layout列表，只对4维的输入有效，格式为“NCHW”或“NHWC”。默认值为 None，表示所有输入的 layout都为 NHWC
	accuracy_analysis: 是否开启精度分析，默认为False
	output_dir: 分析结果保存目录，默认值为'./snapshot'
<b>返回值</b>	results: 推理结果，类型是list，列表成员是ndarray

示例:

```
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)
```

精度分析结果如图3-2。

layer_name	simulator_error				runtime_error			
	entire		single		entire		single_sim	
	cos	euc	cos	euc	cos	euc	cos	euc
[Input] images	1.00000	0.0	1.00000	0.0				
[exDataConvert] images_int8	1.00000	2.5780	1.00000	2.5780				
[Conv] 128	1.00000	2.5780	1.00000	2.5780	1.00000	2.5780	1.00000	0.0
[Conv] 286								
[Relu] 131	0.99977	37.674	0.99977	37.674	0.99977	37.682	1.00000	2.0193
[Conv] 132								
[Relu] 133	0.99949	53.789	0.99960	47.679	0.99949	53.790	1.00000	3.2487
...								
[Sigmoid] 283_int8	0.99903	3.6972	0.99995	0.8721				
[exDataConvert] 283	0.99903	3.6972	0.99996	0.7372	0.99908	3.5866	1.00000	0.1702
[Conv] 280								
[Sigmoid] output_int8	0.99934	6.3987	0.99995	1.7267				
[exDataConvert] output	0.99934	6.3987	0.99996	1.4713	0.99932	6.4315	1.00000	0.3788

图3-2 精度分析结果

完整的精度分析包括模拟器精度分析结果和板端精度分析结果，同时模拟器和板端的精度分析结果都分为完整模型推理结果对比和逐层推理结果对比。详细说明如下：

- simulator\_error:
  - entire: 从输入层到当前层 simulator 结果与 golden 结果对比的余弦距离和欧氏距离。

- single: 当前层使用 golden 输入时, simulator 结果与 golden 结果对比的余弦距离和欧氏距离。
- runtime\_error:
  - entire: 从输入层到当前层板端实际推理结果与 golden 结果对比的余弦距离和欧氏距离。
  - single: 当前层使用 golden 输入时, 板端当前层实际推理结果与 golden 结果对比的余弦距离和欧氏距离。

**注意事项:**

1.通过 `rknn.load_rknn()` 加载RKNN模型后, 因RKNN模型缺失原始模型信息, 因此无法使用模型精度分析功能。

2.指定`target`和`core_mask`后即为RK182X板端精度分析, 否则为模拟器精度分析

### 3.3.2 模型性能评估

接口 `rknn.eval_perf()` 会输出模型的性能评估结果, `log_level` 参数表示打印性能信息等级, 默认值为0。具体等级分为0: 仅打印汇总表; 1: 打印每个 Op 的详细信息表和汇总表; 2: 打印每个 Op 的详细信息包括时间、周期和带宽及汇总表。`dynamic_idx` 参数表示动态输入的索引值, 仅在动态shape中有效。默认值为0。注意: 性能评估需要在 `rknn.config()` 中配置 `profile_mode=True`。

示例代码:

```
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
Total	0	315253	315253	100.00	

表3-1 性能评估结果各字段说明

字段	详细说明
ID	算子编号
OpType	算子类型
Call	单帧内该算子运行次数
CPU(us)	单帧内该算子在 CPU 上的总耗时
NPU(us)	单帧内该算子在 NPU 上的总耗时
Total(us)	单帧内该算子的总耗时
Ratio(%)	单帧内该算子的总耗时与单帧总耗时的比值

### 3.3.3 模型内存评估

接口 `rknn.eval_memory()` 可以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
=====
```

内存评估结果各字段说明如下：

表3-2 内存评估结果各字段说明

字段	详细说明
Core	使用的核心数
Command	寄存器配置及相关控制信息的内存占用 (MB)
Internal	模型中间 Tensor 的内存占用 (MB)
Weight	模型权重 Tensor 的内存占用 (MB)
KVCache	模型 KV Cache 的内存占用 (MB)

字段	详细说明
Total	模型整体内存总占用 (MB)

## 3.4 板端C API推理

### 3.4.1 CNN/ViT等CNN模型推理

本节介绍RKNN3基础C API的调用流程，适用于CNN/ViT等通用视觉模型。

典型调用顺序：初始化上下文 → 加载模型 → 模型初始化 → 查询属性 → 创建内存 → 准备输入 → 推理执行 → 处理输出 → 释放资源。

以CNN/ViT等视觉模型为例，C API的推理流程如下图：

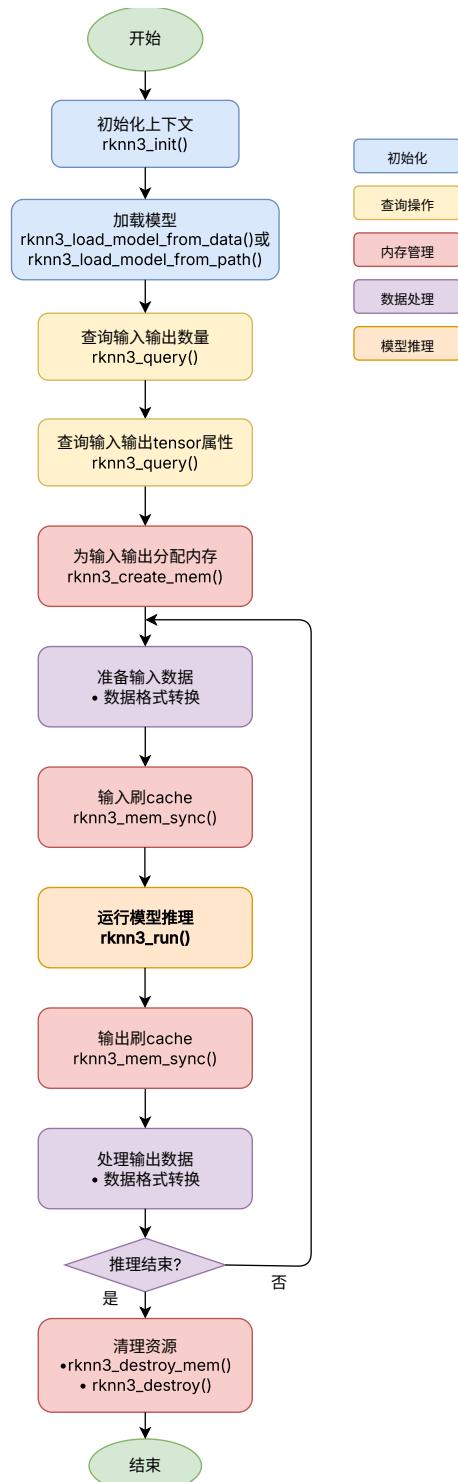


图 3-3 RKNN3 C API 推理流程

### 3.4.1.1 初始化上下文

```
int rknn3_init(rknn3_context* context, rknn3_init_extend* init_extend);
```

接口说明：

API 名称	<b>rknn3_init</b>
功能	创建RKNN3运行时上下文，是所有后续操作的前提
参数	rknn3_context* context: 输出参数，返回创建的上下文句柄

API 名称	<b>rknn3_init</b>
	rknn3_init_extend* init_extend: 可选参数, 用于指定设备ID (多设备场景) 或传入NULL使用默认设备
返回值	int: 返回0表示成功, 非0表示失败

示例代码:

```
rknn3_context ctx;
int ret = rknn3_init(&ctx, NULL);
if (ret != RKNN3_SUCCESS) {
    printf("rknn3_init failed! ret=%d\n", ret);
    return -1;
}
```

#### 3.4.1.2 加载模型

```
int rknn3_load_model_from_path(rknn3_context context, const char* model_path, const char* weight_path);
```

接口说明:

API 名称	<b>rknn3_load_model_from_path</b>
功能	从文件路径加载RKNN模型文件 (.rknn) 和权重文件 (.weight) 到Runtime
参数	rknn3_context context: RKNN3运行时上下文
	const char* model_path: 模型文件路径
	const char* weight_path: 权重文件路径
返回值	int: 返回0表示成功, 非0表示失败

```
int rknn3_load_model_from_data(rknn3_context context, const void* model_data, uint64_t model_size,
                               const void* weight_data, uint64_t weight_size);
```

接口说明:

API 名称	<b>rknn3_load_model_from_data</b>
功能	从内存数据加载RKNN模型和权重到Runtime
参数	rknn3_context context: RKNN3运行时上下文
	const void* model_data: 模型数据指针
	uint64_t model_size: 模型数据大小
	const void* weight_data: 权重数据指针
	uint64_t weight_size: 权重数据大小
返回值	int: 返回0表示成功, 非0表示失败

示例代码:

```
// 从文件加载
FILE* fp = fopen(model_path, "rb");
fseek(fp, 0, SEEK_END);
int model_len = ftell(fp);
void* model = malloc(model_len);
fseek(fp, 0, SEEK_SET);
fread(model, 1, model_len, fp);
fclose(fp);

// 加载权重 (同样方式)
// ...

ret = rknn3_load_model_from_data(ctx, model, model_len, weight_data, weight_len);
```

```

if (ret != RKNN3_SUCCESS) {
    printf("rknn3_load_model_from_data failed! ret=%d\n", ret);
    return -1;
}

```

### 3.4.1.3 模型初始化

```
int rknn3_model_init(rknn3_context context, rknn3_config* config);
```

接口说明：

API 名称	<b>rknn3_model_init</b>
功能	配置模型运行参数并初始化，完成后可释放模型和权重数据
参数	rknn3_context context: RKNN3运行时上下文
	rknn3_config* config: 配置参数，包含以下字段： - run_core_mask: NPU核心掩码，控制模型运行在哪些核心上 - priority: 运行优先级 - run_timeout: 运行超时时间（毫秒），0表示无超时 - user_mem_weight/internal/sram: 是否使用用户分配的内存
返回值	int: 返回0表示成功，非0表示失败

示例代码：

```

// 查询核心数
uint32_t core_num = 0;
ret = rknn3_query(ctx, RKNN3_QUERY_CORE_NUMBER, &core_num, sizeof(core_num));

// 自动生成核心掩码
uint32_t core_mask = 0;
for (uint32_t i = 0; i < core_num; i++) {
    core_mask |= (1 << i);
}

rknn3_config config;
memset(&config, 0, sizeof(config));
config.run_core_mask = core_mask;

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

// 初始化完成后释放模型数据
free(model);
free(weight_data);

```

### 3.4.1.4 查询模型属性

```
int rknn3_query(rknn3_context context, rknn3_query_cmd cmd, void* info, uint64_t size);
```

接口说明：

API 名称	<b>rknn3_query</b>
功能	查询模型的输入/输出信息、版本、性能等
参数	rknn3_context context: RKNN3运行时上下文
	rknn3_query_cmd cmd: 查询命令类型
	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）
返回值	int: 返回0表示成功，非0表示失败

常用查询命令：

- `RKNN3_QUERY_IN_OUT_NUM`：查询输入/输出张量个数
- `RKNN3_QUERY_INPUT_ATTR`：查询输入张量属性
- `RKNN3_QUERY_OUTPUT_ATTR`：查询输出张量属性
- `RKNN3_QUERY_CORE_NUMBER`：查询NPU核心数量
- `RKNN3_QUERY_DYNAMIC_SHAPE_CONFIG`：查询完整的动态形状配置
- `RKNN3_QUERY_DYNAMIC_SHAPE_INFO`：查询所有支持的形状组合
- `RKNN3_QUERY_SDK_VERSION`：查询SDK版本

张量属性 (`rknn3_tensor_attr`) 重要字段：

- `index`：张量索引（查询前需设置）
- `name`：张量名称
- `n_dims`：维度数量
- `shape[RKNN3_MAX_DIMS]`：各维度大小
- `n_elems`：元素总数
- `aligned_size`：对齐后的内存大小（字节）
- `dtype`：数据类型（FP32/FP16/INT8/UINT8等）
- `layout`：数据布局（NCHW/NHWC/NC1HWC2等）
- `qnt_type`：量化类型
- `qnt_info.scale/zero_point`：量化参数
- `core_id`：所属核心ID

示例代码：

```
// 查询输入/输出个数
rknn3_input_output_num io_num;
ret = rknn3_query(ctx, RKNN3_QUERY_IN_OUT_NUM, &io_num, sizeof(io_num));

printf("input tensors:\n");
rknn3_tensor inputs[io_num.n_input];
for (uint32_t i = 0; i < io_num.n_input; i++) {
    inputs[i].attr = (rknn3_tensor_attr*)malloc(sizeof(rknn3_tensor_attr));
    inputs[i].attr->index = i;
    ret = rknn3_query(ctx, RKNN3_QUERY_INPUT_ATTR, inputs[i].attr, sizeof(rknn3_tensor_attr));

    // 打印属性信息
    printf("  input[%d]: name=%s, shape=[%d,%d,%d,%d], dtype=%s, layout=%s\n",
        i, inputs[i].attr->name,
        inputs[i].attr->shape[0], inputs[i].attr->shape[1],
        inputs[i].attr->shape[2], inputs[i].attr->shape[3],
        rknn3_get_type_string(inputs[i].attr->dtype),
        rknn3_get_layout_string(inputs[i].attr->layout));
}

// 输出张量查询方式相同
```

### 3.4.1.5 创建张量内存

```
rknn3_tensor_mem* rknn3_create_mem(rknn3_context context, uint64_t size, int32_t core_id,
                                     rknn3_mem_alloc_flags flags);
```

接口说明：

API 名称	<code>rknn3_create_mem</code>
功能	为输入/输出张量分配内存
参数	<code>rknn3_context context</code> : RKNN3运行时上下文 <code>uint64_t size</code> : 内存大小（使用 <code>aligned_size</code> ）

API 名称	<code>rknn3_create_mem</code>
	<code>int32_t core_id: 目标核心ID (从 attr-&gt;core_id 获取)</code>
	<code>rknn3_mem_alloc_flags flags: 内存标志 (RKNN3_FLAG_MEMORY_CACHEABLE 是带Cache, RKNN3_FLAG_MEMORY_NON_CACHEABLE 是不带Cache)</code>
返回值	<code>rknn3_tensor_mem*: 成功时返回内存对象指针, 失败时返回NULL</code>

示例代码：

```
// 为输入张量创建内存
for (uint32_t i = 0; i < io_num.n_input; i++) {
    inputs[i].mem = rknn3_create_mem(ctx,
        inputs[i].attr->aligned_size,
        inputs[i].attr->core_id,
        RKNN3_FLAG_MEMORY_CACHEABLE);
    if (inputs[i].mem == NULL) {
        printf("Failed to create input memory %d\n", i);
        return -1;
    }
}

// 为输出张量创建内存 (方式相同)
```

### 3.4.1.6 准备输入数据

根据模型的输入数据类型和布局，需要进行相应的数据转换，假定已准备好输入类型是FP32，布局是NCHW的数据，则通常需要以下两个步骤转换成NPU的输入数据格式：

数据类型转换：

- FP32 → FP16：使用 `fp32_to_fp16` 转换
- FP32 → INT8：量化转换 `(value / scale) + zero_point`
- UINT8 → FP16/INT8：根据layout进行转换

布局转换：

- **NCHW**：标准布局，直接拷贝或类型转换
- **NHWC**：需要从NCHW转换为NHWC
- **NC1HWC2**：NPU优化布局，需要特殊转换，以RK182X为例：
  - FP16：16个子通道分块 (C2=16)
  - INT8：32个子通道分块 (C2=32)

示例代码：

```
// 假设输入数据为FP32 NCHW格式
float* input_data = ...; // 从文件或内存加载
rknn3_tensor_attr* input_attr = inputs[0].attr;
void* dst = inputs[0].mem->virt_addr;

if (input_attr->layout == RKNN3_TENSOR_NCHW) {
    if (input_attr->dtype == RKNN3_TENSOR_FLOAT16) {
        // FP32 → FP16
        for (int i = 0; i < input_attr->n_elems; i++) {
            ((float16*)dst)[i] = fp32_to_fp16(input_data[i]);
        }
    } else if (input_attr->dtype == RKNN3_TENSOR_INT8) {
        // FP32 → INT8 量化
        float scale = input_attr->qnt_info.scale;
        int32_t zp = input_attr->qnt_info.zero_point;
        for (int i = 0; i < input_attr->n_elems; i++) {
            ((int8_t*)dst)[i] = (int8_t)(input_data[i] / scale + zp);
        }
    }
} else if (input_attr->layout == RKNN3_TENSOR_NC1HWC2) {
    // 需要调用专用的布局转换函数
    // NCHW_fp32_to_NC1HWC2_fp16/int8(...)
```

```
}
```

#### 3.4.1.7 内存同步

```
int rknn3_mem_sync(rknn3_context context, rknn3_tensor_mem* mem, rknn3_mem_sync_mode mode);
```

接口说明：

API 名称	<code>rknn3_mem_sync</code>
功能	同步CPU和NPU之间的缓存数据
参数	<code>rknn3_context context</code> : RKNN3运行时上下文 <code>rknn3_tensor_mem* mem</code> : 要同步的内存对象指针 <code>rknn3_mem_sync_mode mode</code> : 同步模式
返回值	int: 返回0表示成功，非0表示失败

同步模式：

- `RKNN3_MEMORY_SYNC_TO_DEVICE`: CPU → NPU (输入数据写入后)
- `RKNN3_MEMORY_SYNC_FROM_DEVICE`: NPU → CPU (推理完成后读取输出)
- `RKNN3_MEMORY_SYNC_BIDIRECTIONAL`: 双向同步

示例代码：

```
// 输入数据同步到设备
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);
        return -1;
    }
}
```

#### 3.4.1.8 执行推理

```
int rknn3_run(rknn3_context context, const rknn3_tensor inputs[], uint32_t n_inputs,
              rknn3_tensor outputs[], uint32_t n_outputs);
```

接口说明：

API 名称	<code>rknn3_run</code>
功能	执行同步推理，阻塞等待完成
参数	<code>rknn3_context context</code> : RKNN3运行时上下文 <code>const rknn3_tensor inputs[]</code> : 输入张量数组 <code>uint32_t n_inputs</code> : 输入张量数量 <code>rknn3_tensor outputs[]</code> : 输出张量数组 <code>uint32_t n_outputs</code> : 输出张量数量
返回值	int: 返回0表示成功，非0表示失败

示例代码：

```
// 同步推理
struct timespec start, end;
clock_gettime(CLOCK_MONOTONIC, &start);

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);
    return -1;
}
```

```

}

clock_gettime(CLOCK_MONOTONIC, &end);

float elapsed = ((end.tv_sec - start.tv_sec) * 1000 * 1000 +
                 (end.tv_nsec - start.tv_nsec) / 1000) / 1000.0;
printf("Inference time: %.3f ms\n", elapsed);

// 输出数据同步到CPU
for (int i = 0; i < io_num.n_output; i++) {
    rknn3_mem_sync(ctx, outputs[i].mem, RKNN3_MEMORY_SYNC_FROM_DEVICE);
}

```

### 3.4.1.9 处理输出数据

假定应用要得到FP32类型，NCHW布局的数据，根据NPU输出张量的类型和布局，需要进行反量化和布局转换：

**数据类型转换：**

- FP16 → FP32：使用 `fp16_to_fp32` 转换
- INT8 → FP32：反量化 `(value - zero_point) * scale`

**布局转换：**

- NC1HWC2 → NCHW：调用专用转换函数

**示例代码：**

```

for (uint32_t i = 0; i < io_num.n_output; i++) {
    rknn3_tensor_attr* output_attr = outputs[i].attr;
    void* output_mem = outputs[i].mem->virt_addr;

    // 分配FP32输出缓冲区
    float* output_fp32 = (float*)malloc(output_attr->n_elems * sizeof(float));

    if (output_attr->layout == RKNN3_TENSOR_NCHW) {
        if (output_attr->dtype == RKNN3_TENSOR_FLOAT16) {
            // FP16 → FP32
            for (int j = 0; j < output_attr->n_elems; j++) {
                output_fp32[j] = fp16_to_fp32(((float16*)output_mem)[j]);
            }
        } else if (output_attr->dtype == RKNN3_TENSOR_INT8) {
            // INT8 → FP32 反量化
            float scale = output_attr->qnt_info.scale;
            int32_t zp = output_attr->qnt_info.zero_point;
            for (int j = 0; j < output_attr->n_elems; j++) {
                output_fp32[j] = (((int8_t*)output_mem)[j] - zp) * scale;
            }
        }
    } else if (output_attr->layout == RKNN3_TENSOR_NC1HWC2) {
        // 调用NC1HWC2_to_NCHW转换函数
        // NC1HWC2_fp16_to_NCHW_fp32(...) 或 NC1HWC2_int8_to_NCHW_fp32(...)
    }

    // 后续业务逻辑处理...
    free(output_fp32);
}

```

### 3.4.1.10 释放资源

```
int rknn3_destroy_mem(rknn3_context context, rknn3_tensor_mem* mem);
```

**接口说明：**

API 名称	<code>rknn3_destroy_mem</code>
功能	释放为输入/输出张量分配的内存
参数	rknn3_context context: RKNN3运行时上下文
	rknn3_tensor_mem* mem: 要释放的内存对象指针
返回值	int: 返回0表示成功，非0表示失败

```
int rknn3_destroy(rknn3_context context);
```

接口说明：

API 名称	rknn3_destroy
功能	销毁RKNN3运行时上下文并释放资源
参数	rknn3_context context: RKNN3运行时上下文
返回值	int：返回0表示成功，非0表示失败

示例代码：

```
// 释放输入/输出内存
for (uint32_t i = 0; i < io_num.n_input; i++) {
    if (inputs[i].attr) free(inputs[i].attr);
    if (inputs[i].mem) rknn3_destroy_mem(ctx, inputs[i].mem);
}
for (uint32_t i = 0; i < io_num.n_output; i++) {
    if (outputs[i].attr) free(outputs[i].attr);
    if (outputs[i].mem) rknn3_destroy_mem(ctx, outputs[i].mem);
}

// 销毁上下文
rknn3_destroy(ctx);
```

### 3.4.1.11 完整示例流程

以下是一个完整的推理流程概览：

```
// 1. 初始化
rknn3_context ctx;
rknn3_init(&ctx, NULL);

// 2. 加载模型
rknn3_load_model_from_data(ctx, model, model_len, weight, weight_len);

// 3. 模型初始化（配置核心掩码）
rknn3_config config = {0};
config.run_core_mask = 0x3; // 使用core 0和1
rknn3_model_init(ctx, &config);

// 4. 查询属性
rknn3_input_output_num io_num;
rknn3_query(ctx, RKNN3_QUERY_IN_OUT_NUM, &io_num, sizeof(io_num));

// 5. 创建输入/输出张量和内存
rknn3_tensor inputs[io_num.n_input];
rknn3_tensor outputs[io_num.n_output];
// ... 查询属性、创建内存 ...

// 6. 准备输入数据（包含布局转换、类型转换）
// ... 填充inputs[i].mem->virt_addr ...

// 7. 同步输入到设备
for (int i = 0; i < io_num.n_input; i++)
    rknn3_mem_sync(ctx, inputs[i].mem, RKNN3_MEMORY_SYNC_TO_DEVICE);

// 8. 执行推理
rknn3_run(ctx, inputs, io_num.n_input, outputs, io_num.n_output);

// 9. 同步输出到CPU
for (int i = 0; i < io_num.n_output; i++)
    rknn3_mem_sync(ctx, outputs[i].mem, RKNN3_MEMORY_SYNC_FROM_DEVICE);

// 10. 处理输出（包含反量化、布局转换）
// ... 读取outputs[i].mem->virt_addr ...
```

```
// 11. 释放资源  
rknn3_destroy(ctx);
```

#### 3.4.1.12 注意事项

1. **内存对齐**: 使用 `aligned_size` 而非 `n_elems * sizeof(dtype)` 分配内存
2. **核心ID**: 创建内存时需使用 `attr->core_id`, 确保内存分配在正确的核心上
3. **缓存同步**: 使用可缓存内存 (`RKNN3_FLAG_MEMORY_CACHEABLE`) 时, 必须正确调用 `rknn3_mem_sync`
4. **布局转换**: NC1HWC2布局需要特殊处理, 注意FP16/INT8的子通道数不同
5. **量化参数**: INT8模型需要使用 `qnt_info.scale` 和 `zero_point` 进行量化/反量化
6. **错误处理**: 所有API调用均需检查返回值 (`RKNN3_SUCCESS` 表示成功)
7. **资源释放顺序**: 先释放tensor内存, 最后释放上下文

#### 3.4.1.13 调试工具

导出中间特征:

```
int rknn3_dump_features(rknn3_context context,  
                        const rknn3_tensor inputs[], uint32_t n_inputs,  
                        rknn3_tensor outputs[], uint32_t n_outputs,  
                        const char* dump_dir);
```

接口说明:

API 名称	<code>rknn3_dump_features</code>
功能	逐层导出中间特征到目标文件夹, 保存为 <code>.npy</code> 文件, 便于对比分析精度问题。
参数	<code>rknn3_context context</code> : RKNN3运行时上下文 <code>const rknn3_tensor inputs[]</code> : 输入张量数组 <code>uint32_t n_inputs</code> : 输入张量数量 <code>rknn3_tensor outputs[]</code> : 输出张量数组 <code>uint32_t n_outputs</code> : 输出张量数量 <code>const char* dump_dir</code> : 中间结果保存目标文件夹
返回值	<code>int</code> : 返回0表示成功, 非0表示失败

逐层算子性能分析:

```
int rknn3_profile_ops(rknn3_context context,  
                      const rknn3_tensor inputs[], uint32_t n_inputs,  
                      rknn3_tensor outputs[], uint32_t n_outputs,  
                      uint32_t log_level);
```

接口说明:

API 名称	<code>rknn3_profile_ops</code>
功能	打印逐层算子性能信息, 包含耗时、Cycle和带宽等统计, 并输出汇总表。
参数	<code>rknn3_context context</code> : RKNN3运行时上下文 <code>const rknn3_tensor inputs[]</code> : 输入张量数组 <code>uint32_t n_inputs</code> : 输入张量数量 <code>rknn3_tensor outputs[]</code> : 输出张量数组 (可为NULL) <code>uint32_t n_outputs</code> : 输出张量数量 (为0时使用内部输出) <code>uint32_t log_level</code> : 0仅打印逐算子的耗时汇总; 1打印逐算子耗时表+汇总; 2打印逐算子耗时/Cycle/带宽+汇总
返回值	<code>int</code> : 返回0表示成功, 非0表示失败

内存占用分析:

```
int rknn3_profile_mem(rknn3_context context);
```

接口说明：

API 名称	<b>rknn3_profile_mem</b>
功能	打印各NPU核心内存占用信息，包含命令/权重/内部/KVCache内存及设备内存统计。
参数	rknn3_context context: RKNN3运行时上下文
返回值	int: 返回0表示成功，非0表示失败

### 3.4.2 LLM模型推理

#### 3.4.2.1 session 初始化及销毁

```
rknn3_session* rknn3_session_init(rknn3_context context, rknn3_llm_param* params, int n_params);
```

接口说明：

API	<b>rknn3_session_init</b>
功能	使用指定参数初始化新的RKNN3会话。
参数	rknn3_context context: 用于会话的RKNN3上下文指针
	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
返回值	rknn3_session* 如果成功返回会话句柄，如果初始化失败则返回NULL

```
int rknn3_session_destroy(rknn3_session* session);
```

接口说明：

API	<b>rknn3_session_destroy</b>
功能	销毁RKNN3会话并释放相关资源。
参数	rknn3_session* session: 要销毁的RKNN3会话指针
返回值	int: 返回0表示成功，非0表示失败
注意	调用此函数后，会话指针变为无效，不应再使用

示例代码：

```
// 初始化用于LLM推理的参数结构体
rknn3_llm_param params;
memset(&params, 0, sizeof(rknn3_llm_param));

// 指定推理输出节点的名称
params.logits_name = (char *)"logits";
// 设置最大上下文长度，单位token
params.max_context_len = 1024;
// temperature采样参数，影响生成的随机性
params.sampling_param.temperature = 1.0f;
// top-k采样
params.sampling_param.top_k = 1;
// top-p采样 (nucleus采样)
params.sampling_param.top_p = 0.9;
// 重复惩罚系数，降低重复内容的生成概率
params.sampling_param.repeat_penalty = 1.1f;
// 频率惩罚，惩罚出现频率高的token
params.sampling_param.frequency_penalty = 0.0f;
// 存在惩罚，惩罚已出现过的token
params.sampling_param.presence_penalty = 0.0f;
// 设置词表大小
```

```

params.vocab_info.vocab_size = vocab_info.vocab_size;
// 设置序列结束 token id, 控制推理何时结束
params.vocab_info.n_special_eos_id = vocab_info.n_special_eos_id;
memcpy(params.vocab_info.special_eos_id, vocab_info.special_eos_id, sizeof(vocab_info.special_eos_id));
// 设置序列开始 token id
params.vocab_info.n_special_bos_id = vocab_info.n_special_bos_id;
memcpy(params.vocab_info.special_bos_id, vocab_info.special_bos_id, sizeof(vocab_info.special_bos_id));
// 设置换行符token ID
params.vocab_info.linefeed_id = vocab_info.linefeed_id;
// 是否忽略序列结束 token 继续推理
params.vocab_info.ignore_eos_token = 0;

rknn3_session *session = rknn3_session_init(ctx, &params, 1);
if (!session) {
    printf("rknn3_session_init failed!\n");
    return -1;
}

...
ret = rknn3_session_destroy(session);
if (ret != 0) {
    printf("rknn3_session_destroy failed!, ret = %d\n", ret);
    return -1;
}

```

### 3.4.2.2 回调函数定义

```
int rknn3_session_set_callback(rknn3_session* session, RKLLMCallback* callback);
```

接口说明：

API	rknn3_session_set_callback
功能	为RKNN3会话设置回调函数
参数	rknn3_session* session: RKNN3会话实例指针
	RKLLMCallback* callback: 包含回调函数的RKLLMCallback结构指针
返回值	int: 返回0表示成功，非0表示失败

示例代码：

```

// LLM Callback
RKLLMCallback callback;
memset(&callback, 0, sizeof(RKLLMCallback));

// result 回调函数, 用于处理每步推理得到的token (如将token转换为文本)
callback.result_callback = result_callback;
callback.result_userdata = tokenizer;
// embedding 回调函数, 用于将token转换为embedding向量
callback.embed_callback = embed_callback;
callback.embed_userdata = &embedding_info;
// tokenizer 回调函数, 用于将文本转换为token
callback.tokenizer_callback = tokenizer_callback;
callback.tokenizer_userdata = tokenizer;
// sampling 回调函数, 用于从logits中采样得到token id
callback.sampling_callback = sampling_callback;
callback.sampling_userdata = &embedding_info;
// output 回调函数, 用于处理模型输出 tensors (用户自定义处理输出 tensors, 如作为其他模型输入等)
callback.output_callback = output_callback;
callback.output_userdata = &embedding_info;
callback.output_tensors = output_tensors;
callback.n_output_tensors = n_output_tensors;

// LLM Set Callback
ret = rknn3_session_set_callback(session, &callback);
if (ret != 0)
{
    printf("rknn3_session_set_callback failed, ret = %d\n", ret);
}

```

```
    return -1;
}
```

### 3.4.2.3 LLM 模型推理执行

```
int rknn3_session_run(rknn3_session* session, rknn3_llm_input inputs[], uint32_t n_inputs,
rknn3_llm_infer_param* param);
```

接口说明：

API	<code>rknn3_session_run</code>
功能	使用提供的输入和参数运行推理。
参数	<code>rknn3_session* session</code> : RKNN3会话句柄指针 <code>rknn3_llm_input inputs[]</code> : 包含输入数据的输入 tensor 数组 <code>uint32_t n_inputs</code> : 提供的输入 tensor 数量 <code>rknn3_llm_infer_param* param</code> : 推理参数配置指针
返回值	<code>int</code> : 返回0表示成功，非0表示失败

```
int rknn3_session_run_async(rknn3_session* session, rknn3_llm_input inputs[], uint32_t n_inputs,
rknn3_llm_infer_param* param);
```

接口说明：

API	<code>rknn3_session_run_async</code>
功能	为大型语言模型会话异步运行推理。
参数	<code>rknn3_session* session</code> : RKNN3会话句柄指针 <code>rknn3_llm_input inputs[]</code> : 模型的输入 tensor 数组 <code>uint32_t n_inputs</code> : 输入 tensor 数量 <code>rknn3_llm_infer_param* param</code> : 推理参数配置指针
返回值	<code>int</code> : 返回0表示成功，非0表示失败
注意	此函数对给定的LLM会话执行异步推理。它允许使用提供的输入和参数对模型进行非阻塞执行

示例代码：

```
// 准备输入数据
int n_inputs = 1;
rknn3_llm_input inputs[n_inputs];
rknn3_llm_input input;
rknn3_llm_tensor input_tensor;
rknn3_llm_infer_param param;
memset(&inputs, 0, sizeof(rknn3_llm_input) * n_inputs);
memset(&input, 0, sizeof(rknn3_llm_input));
memset(&input_tensor, 0, sizeof(rknn3_llm_tensor));
memset(&param, 0, sizeof(rknn3_llm_infer_param));

char *prompt = "Please explain the basic concept of relativity";
param = {.keep_history = 0, .max_new_tokens = max_new_tokens};
input_tensor = {.name = NULL, .prompt = prompt, .embed = NULL, .tokens = NULL, .n_tokens = 0, .enable_thinking = false};
input.input_type = RKNN3_LLM_INPUT_PROMPT;
input.llm_input = input_tensor;
inputs[0] = input;

ret = rknn3_session_run(session, inputs, n_inputs, &param);
// ret = rknn3_session_run_async(session, inputs, n_inputs, &param);
if (ret != 0) {
    printf("rknn3_session_run failed, ret = %d\n", ret);
```

```
    return -1;
}
```

#### 3.4.2.4 LLM 模型推理中断

```
int rknn3_session_stop(rknn3_session* session);
```

接口说明：

API	rknn3_session_stop
功能	终止当前会话推理。
参数	rknn3_session* session：RKNN3会话指针
返回值	int：返回0表示成功，非0表示失败

示例代码：

```
ret = rknn3_session_stop(session);
if (ret != 0) {
    printf("rknn3_session_stop failed, ret = %d\n", ret);
    return -1;
}
```

#### 3.4.2.5 Chat Template 设置

```
int rknn3_session_set_chat_template(rknn3_session* session, const char* system_prompt, const char*
prompt_prefix, const char* prompt_postfix);
```

接口说明：

API	rknn3_session_set_chat_template
功能	设置LLM的聊天模板，包括系统提示、前缀和后缀。
参数	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
返回值	int：返回0表示成功，非0表示失败

示例代码：

```
const char* system_prompt = "<|im_start|>system\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";
ret = rknn3_session_set_chat_template(session, system_prompt, prompt_prefix, prompt_postfix);
if (ret != 0) {
    printf("rknn3_session_set_chat_template failed, ret = %d\n", ret);
    return -1;
}
```

#### 3.4.2.6 LLM 运行状态查询

```
int rknn3_session_query_state(rknn3_session* session, RKLLMRunState* state);
```

接口说明：

API	rknn3_session_query_state
功能	查询RKNN3会话的当前状态。
参数	rknn3_session* session：要查询的RKNN3会话指针

<b>API</b>	<b>rknn3_session_query_state</b>
	RKLLMRunState* state：用于存储查询运行状态的指针。如可查询最大推理token数(n_max_tokens)，当前已推理token数(n_total_tokens)，当前会话prefill阶段处理的token数(n_prefill_tokens)，当前会话generate(decode)阶段生成的token数(n_decode_tokens)，KV Cache 策略(kvcache_policy)。
<b>返回值</b>	int：返回0表示成功，非0表示失败

示例代码：

```
RKLLMRunState state;
memset(&state, 0, sizeof(RKLLMRunState));
ret = rknn3_session_query_state(session, &state);
if (ret != 0) {
    printf("rknn3_session_query_state failed, ret = %d\n", ret);
    return -1;
}
```

#### 3.4.2.7 KVCache 策略设置

```
int rknn3_session_set_kvcache_policy(rknn3_session* session, rknn3_kvcache_policy policy,
rknn3_kvcache_policy_param* param);
```

接口说明：

<b>API</b>	<b>rknn3_session_set_kvcache_policy</b>
<b>功能</b>	设置RKNN3会话的KV Cache策略。
<b>参数</b>	rknn3_session* session：RKNN3会话句柄
	rknn3_kvcache_policy policy：要设置的 KV Cache 策略。RKNN3_KVCACHE_POLICY_RECURRENT 表示 KV Cache 自动复写。RKNN3_KVCACHE_POLICY_NORMAL 表示只使用最大 max_context_len 长度的 KV Cache，超过后停止推理。默认是 RKNN3_KVCACHE_POLICY_RECURRENT
	rknn3_kvcache_policy_param* param：指定 KV Cache 策略的参数。目前提供策略 RKNN3_KV CACHE_POLICY_RECURRENT 策略下保留不被复写的长度 (n_keep 和 n_keep_aligned)
<b>返回值</b>	int：返回0表示成功，非0表示失败

示例代码：

```
ret = rknn3_session_set_kvcache_policy(session, RKNN3_KV CACHE_POLICY_RECURRENT, nullptr);
if (ret != 0) {
    printf("rknn3_session_set_kvcache_policy failed, ret = %d\n", ret);
    return -1;
}
```

#### 3.4.2.8 KVCache 清理

```
int rknn3_session_clear_kvcache(rknn3_session* session, rknn3_kvcache_clear_policy clear);
```

接口说明：

<b>API</b>	<b>rknn3_session_clear_kvcache</b>
<b>功能</b>	根据指定策略清除RKNN3会话的KV Cache。
<b>参数</b>	rknn3_session* session：RKNN3会话句柄指针
	rknn3_kvcache_clear_policy clear：清除KV Cache的策略，定义如何清除缓存。RKNN3_KV CACHE_CLEAR_ALL 表示 KV Cache 全部清除。RKNN3_KV CACHE_KEEP_SYSTEM_PROMPT 表示保留 system prompt 对应的 KV Cache，剩下的清除。
<b>返回值</b>	int：返回0表示成功，非0表示失败

示例代码：

```

ret = rknn3_session_clear_kvcache(session, RKNN3_KVCACHE_CLEAR_ALL);
if (ret != 0) {
    printf("rknn3_session_clear_kvcache failed, ret = %d\n", ret);
    return -1;
}

```

### 3.4.2.9 KVCache 导入与导出

```
int rknn3_session_save_kvcache(rknn3_session* session, char* kvcache_path);
```

接口说明：

API	<code>rknn3_session_save_kvcache</code>
功能	将KV Cache保存到指定路径。
参数	<code>rknn3_session* session</code> : RKNN3会话指针
	<code>char* kvcache_path</code> : 保存KV cache的路径
返回值	int: 返回0表示成功，非0表示失败

```
int rknn3_session_load_kvcache(rknn3_session* session, const char* kvcache_path, int64_t size);
```

接口说明：

API	<code>rknn3_session_load_kvcache</code>
功能	从指定路径加载KV Cache。
参数	<code>rknn3_session* session</code> : RKNN3会话指针
	<code>const char* kvcache_path</code> : KV Cache文件的路径
	<code>int64_t size</code> : KV Cache文件路径的长度，即 <code>strlen(kvcache_path)</code>
返回值	int: 返回0表示成功，非0表示失败

示例代码：

```

ret = rknn3_session_save_kvcache(session, "kvcache.bin");
if (ret != RKNN3_SUCCESS) {
    printf("rknn3_session_save_kvcache failed, ret = %d\n", ret);
    return -1;
}

...

ret = rknn3_session_load_kvcache(session, "kvcache.bin", strlen("kvcache.bin"));
if (ret != RKNN3_SUCCESS) {
    printf("rknn3_session_load_kvcache failed, ret = %d\n", ret);
    return -1;
}

```

### 3.4.2.10 Function Calling 设置

```
int rknn3_session_set_function_tools(rknn3_session* session, const char* tools);
```

接口说明：

API	<code>rknn3_session_set_function_tools</code>
功能	设置RKNN3会话支持的Function Calling工具，用于配置和启用调用外部函数的信息。
参数	<code>rknn3_session* session</code> : RKNN3会话句柄指针
	<code>const char* tools</code> : Function Calling描述字符串，JSON或特定格式，描述工具信息，如函数名、参数等

<b>API</b>	<b>rknn3_session_set_function_tools</b>
<b>返回值</b>	int：返回0表示成功，非0表示失败

示例代码：

```
char *function_tools = R"([{"type": "function", "function": {"name": "get_current_temperature", "description": "Get current temperature at a location.", "parameters": {"type": "object", "properties": {"location": {"type": "string", "description": "The location to get the temperature for, in the format \\\"City, State, Country\\\"."}, "unit": {"type": "string", "enum": ["celsius", "fahrenheit"], "description": "The unit to return the temperature in. Defaults to \\\"celsius\\\"."}}, "required": [{"location"]}}}, {"type": "function", "function": {"name": "get_temperature_date", "description": "Get temperature at a location and date.", "parameters": {"type": "object", "properties": {"location": {"type": "string", "description": "The location to get the temperature for, in the format \\\"City, State, Country\\\"."}, "date": {"type": "string", "description": "The date to get the temperature for, in the format \\\"Year-Month-Day\\\"."}, "unit": {"type": "string", "enum": ["celsius", "fahrenheit"], "description": "The unit to return the temperature in. Defaults to \\\"celsius\\\"."}}, "required": [{"location", "date"}]}]);"

ret = rknn3_session_set_function_tools(session, function_tools);
if (ret != 0) {
    printf("rknn3_session_set_function_tools failed, ret = %d\n", ret);
    return -1;
}
```

### 3.4.2.11 LLM 参数设置

```
int rknn3_session_set_llm_param(rknn3_session* session, rknn3_llm_param* params, int n_params);
```

接口说明：

<b>API</b>	<b>rknn3_session_set_llm_param</b>
<b>功能</b>	配置LLM相关参数（如采样参数、词表信息等），支持在Session初始化后更新LLM相关参数。
<b>参数</b>	<p>rknn3_session* session: RKNN3会话句柄</p> <p>rknn3_llm_param* params: LLM参数数组指针（每个logits输出对应一个LLM参数）</p> <p>int n_params: LLM参数数量（等于logits输出数量）</p>
<b>返回值</b>	int 状态码：见 <a href="#">5.2. 状态码</a>
<b>注意</b>	params需按输出顺序依次设置；如模型有多个logits输出，需对应配置

示例代码：

```
rknn3_llm_param params;
memset(&params, 0, sizeof(rknn3_llm_param));
params.logits_name = (char *)"logits";
params.max_context_len = 1024;
params.sampling_param.temperature = 1.0f;
params.sampling_param.top_k = 1;
params.sampling_param.top_p = 0.9;
params.sampling_param.repeat_penalty = 1.1f;
params.sampling_param.frequency_penalty = 0.0f;
params.sampling_param.presence_penalty = 0.0f;
params.vocab_info.vocab_size = vocab_info.vocab_size;
params.vocab_info.n_special_eos_id = vocab_info.n_special_eos_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;
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;
params.vocab_info.ignore_eos_token = 0;

ret = rknn3_session_set_llm_param(session, &params, 1);
if (!session) {
    printf("rknn3_session_set_llm_param failed, ret = %d\n", ret);
    return -1;
}
```

## 3.5 RKLLM3 Server介绍

### 3.5.1 主要功能说明

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

### 3.5.2 使用方法

通用参数

参数	说明
<code>-a, --alias STRING</code>	设置模型别名（供REST API使用）
<code>-c, --ctx-size N</code>	提示词上下文大小（默认：4096, 0 = 从模型加载）
<code>-n, --predict, --n-predict N</code>	要预测的token数量（默认：-1, -1 = 上下文大小）
<code>-m, --model FNAME</code>	RKNN LLM 模型路径
<code>--weight FNAME</code>	RKNN LLM 权重路径（可不设置）
<code>--model2 FNAME</code>	多模态模型时的vision模型路径
<code>--weight2 FNAME</code>	多模态模型时的vision权重路径（可不设置）
<code>--model3 FNAME</code>	多模态模型时的audio模型路径
<code>--weight3 FNAME</code>	多模态模型时的audio权重路径（可不设置）
<code>--vocab FNAME</code>	词汇表路径
<code>--embed FNAME</code>	embed的bin文件路径
<code>--mel-filter FNAME</code>	mel filters路径（用于audio模型）
<code>--img-start STRING</code>	多模态模型的图像输入前缀
<code>--img-end STRING</code>	多模态模型的图像输入后缀
<code>--img-content STRING</code>	多模态模型的图像输入的占位符
<code>--audio-start STRING</code>	多模态模型的语音输入前缀
<code>--audio-end STRING</code>	多模态模型的语音输入后缀
<code>--audio-content STRING</code>	多模态模型的语音输入的占位符
<code>--img-width N</code>	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）
<code>--img-height N</code>	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）
<code>--chat-template-file JINJA_TEMPLATE_FILE</code>	设置自定义Jinja聊天模板（默认：使用模型元数据中的模板）
<code>--embedding</code>	是否是词嵌入模型

采样参数

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

参数	说明
--frequency-penalty N	重复alpha频率惩罚（默认：0.0, 0.0 = 禁用）

#### 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）

### 3.5.3 快速上手

#### 1. 开启服务进程

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
```

加载多个模型时，params.json的基本格式如下：

```
{
  "host": "127.0.0.1",
  "port": 8080,
  "timeout": 30,
  "models": {
    "Qwen2.5-0.5B": {
      "alias": "Qwen2.5-0.5B",
      "model": "Qwen2.5-0.5B-Instruct.rknn",
      "weight": "Qwen2.5-0.5B-Instruct.weight",
      "model2": "",
      "weight2": "",
      "model3": "",
      "weight3": "",
      "vocab": "Qwen2.5-0.5B-Instruct.tokenizer.gguf",
      "embed": "Qwen2.5-0.5B-Instruct.embed.bin",
      "mel-filter": "",
      "ctx-size": 1024,
      "max-tokens": 2048
    }
  }
}
```

```

    "predict": 512,
    "temp": 0.8,
    "top-k": 1,
    "top-p": 0.8,
    "repeat-penalty": 1.1,
    "presence-penalty": 1.0,
    "frequency-penalty": 1.0,
    "img-start": "",
    "img-end": "",
    "img-content": "",
    "audio-start": "",
    "audio-end": "",
    "audio-content": "",
    "qwenvl": false,
    "chat-template-file": "",
    "embedding": false
},
"Qwen3-0.6B": {
    "alias": "Qwen3-0.6B",
    "model": "Qwen3-0.6B.rknn",
    "weight": "Qwen3-0.6B.weight",
    "model2": "",
    "weight2": "",
    "model3": "",
    "weight3": "",
    "vocab": "Qwen3-0.6B.tokenizer.gguf",
    "embed": "Qwen3-0.6B.embed.bin",
    "mel-filter": "",
    "ctx-size": 1024,
    "predict": 512,
    "temp": 0.8,
    "top-k": 1,
    "top-p": 0.8,
    "repeat-penalty": 1.1,
    "presence-penalty": 1.0,
    "frequency-penalty": 1.0,
    "img-start": "",
    "img-end": "",
    "img-content": "",
    "audio-start": "",
    "audio-end": "",
    "audio-content": "",
    "qwenvl": false,
    "chat-template-file": "",
    "embedding": false
}
}
}

```

注: 使用--params-file参数, 将忽略命令行的其余参数, 仅params.json内的参数有效。

另外, RKLLM3 Server除了提供常规的rkllm3-server二进制可执行程序外, 还提供了librkllm3-server.so的使用方式, 方便将server集成到应用中, 用法如下:

```

#include "rkllm3-server.h"
#include <thread>
#include <stdio.h>

const char *json = R"({
    "host": "127.0.0.1",
    "port": 8080,
    "timeout": 30,
    "models": {
        "Qwen2.5-0.5B": {
            "alias": "Qwen2.5-0.5B",
            "model": "Qwen2.5-0.5B-Instruct.rknn",
            "weight": "Qwen2.5-0.5B-Instruct.weight",
            "model2": "",
            "weight2": "",
            "model3": "",
            "weight3": ""
        }
    }
})";

```

```

        "vocab": "Qwen2.5-0.5B-Instruct.tokenizer.gguf",
        "embed": "Qwen2.5-0.5B-Instruct.embed.bin",
        "mel-filter": "",
        "ctx-size": 1024,
        "predict": 512,
        "temp": 0.8,
        "top-k": 1,
        "top-p": 0.8,
        "repeat-penalty": 1.1,
        "presence-penalty": 1.0,
        "frequency-penalty": 1.0,
        "img-start": "",
        "img-end": "",
        "img-content": "",
        "audio-start": "",
        "audio-end": "",
        "audio-content": "",
        "qwenvl": false,
        "chat-template-file": "",
        "embedding": false
    }
}
});

// server子线程的状态回调函数
void status_callback(void* userdata, ServerStatus status) {
    switch (status) {
    case SERVER_MODEL_INITED:
        printf("SERVER_MODEL_INITED\n");
        break;
    case SERVER_EXITED:
        printf("SERVER_EXITED\n");
        break;
    case SERVER_ERROR:
        printf("SERVER_ERROR\n");
        break;
    default:
        printf("UNKNOW\n");
        break;
    }
}

int main() {
    StatusCallback callback = {0};
    callback.status_callback = status_callback;
    callback.status_userdata = NULL;

    int server_result = 0;
    std::thread server_thread([&]() {
        // start_server会一直阻塞直至stop_server被调用,
        // 因此这边要创建子线程来执行start_server
        server_result = start_server(json, &callback);
    });

    printf("Main thread: Waiting 300 seconds ... \n");
    std::this_thread::sleep_for(std::chrono::seconds(60*5));

    printf("Main thread: Stopping server...\n");
    stop_server();

    printf("Main thread: server_thread.join ...\n");
    server_thread.join();

    return 0;
}

```

const char \*json 字符串内的"model", "weight", "model2", "weight2", "model3", "weight3", "vocab", "embed", "mel-filter"即可以是文件路径, 也可以是文件的fd句柄(解决Android系统下文件权限的问题), 如下:

```

const char *json = R"({
    "host": "127.0.0.1",

```

```

"port": 8080,
"timeout": 30,
"models": {
  "Qwen2.5-0.5B": {
    "alias": "Qwen2.5-0.5B",
    "model": "3",
    "weight": "4",
    "model2": "",
    "weight2": "",
    "model3": "",
    "weight3": "",
    "vocab": "5",
    "embed": "6",
    "mel-filter": ""
  },
  ...
}
}
});
```

## 2. 用curl进行测试

使用 [curl](#)。

```

curl --request POST \
--url http://localhost:8080/v1/chat/completions \
--header "Content-Type: application/json" \
--data '{"messages": [{"role": "user", "content": "Hello!"}], "n_predict": 128}'
```

## 3.5.4 API 端点

### 3.5.4.1 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

## 3.5.5 OpenAI兼容的API端点

### 3.5.5.1 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](#).

返回的列表可以有多个元素，对应多个模型。

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_embed": 4096,
        "n_params": 8030261312,
        "size": 4912898304
      }
    }
  ]
}
```

```
        }
    ]
}
```

### 3.5.5.2 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" }, "title": "Participants", "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)
```

或者使用原始的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"
)

# Function to encode the image
def encode_base64(media_path):
```

```

with open(media_path, "rb") as media_file:
    return base64.b64encode(media_file.read()).decode("utf-8")

image_path = "demo.jpg"
audio_path = "demo.wav"      # 如存在音频输入

# Getting the Base64 string
base64_image = encode_base64(image_path)
base64_audio = encode_base64(audio_path)      # 如存在音频输入

completion = client.chat.completions.create(
    model="Qwen2.5-VL",
    messages=[
        {
            "role": "user",
            "content": [
                {
                    "type": "image_url",
                    "image_url": {
                        "url": f"data:image/jpeg;base64,{base64_image}",
                    },
                },
                {
                    "type": "input_audio",
                    "input_audio": {
                        "data": f"{base64_audio}",
                        "format": "wav",
                    }
                },  # 如存在音频输入
                { "type": "text", "text": "请描述一下图片?" },
            ],
        },
    ],
    stream=True,
    extra_body={
        "n_predict": 256,
        "chat_template_kwargs": { "enable_thinking": False }      # Thinking类模型(如Qwen3)可以通过此方式关闭thinking输出
    }
)

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

```

### 3.5.5.3 POST /v1/embeddings: OpenAI兼容的词嵌入API

参见 [OpenAI Embeddings API documentation](#).

示例:

```

import openai

client = openai.OpenAI(
    base_url="http://localhost:8080/v1",
    api_key = "sk-no-key-required"
)

response = client.embeddings.create(
    model="Qwen3-Embedding-4B",
    input="The food was delicious and the waiter...",
    encoding_format="float"
)

print(response.data[0].embedding)

```

### 3.5.6 API 错误

`rkllm3-server` 返回的错误格式与 OAI 相同: <https://github.com/openai/openai-openapi>

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 /v1/chat/completions 端点收到无效语法时

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

## 3.6 板端Python API推理

RKNN3-Toolkit Lite为用户提供协处理器模型推理的Python接口，方便用户使用Python语言进行AI大模型应用开发或者前期模型验证。

注意：在使用RKNN3-Toolkit Lite开发AI应用前，需要通过RKNN3-Toolkit将各深度学习框架导出的模型转成RKNN模型。模型转换详细方法请参考[3.1章节](#)。

### 3.6.1 系统依赖说明

使用RKNN3-Toolkit Lite需满足以下运行环境要求：

表3-4 RKNN3-Toolkit Lite运行环境

操作系统版本	Debian 10 / 11 / 12 (aarch64)
Python版本	3.9 / 3.10 / 3.11 / 3.12
Python依赖库	'transformers'、'numpy'、'jinja2'

### 3.6.2 工具安装

在RK3588/RK3576等芯片端上通过 `pip3 install` 命令安装 RKNN3-Toolkit Lite。安装步骤如下：

- 如果系统中没有安装 `python3/pip3` 等程序，请先通过 `apt-get` 方式安装，参考命令如下：

```
sudo apt-get update  
sudo apt-get install -y python3 python3-dev python3-pip gcc
```

注意：安装部分依赖模块时，需要编译源码，此时将用到 `python3-dev` 和 `gcc`，因此该步骤将这两个包也一起安装，避免后面安装依赖模块时编译失败。

- 安装依赖模块：`opencv-python` 和 `numpy`，参考命令如下：

```
sudo apt-get install -y python3-opencv  
sudo apt-get install -y python3-numpy
```

或者

```
pip3 install opencv-python  
pip3 install numpy
```

注意：

1. RKNN3-Toolkit Lite本身并不依赖 `opencv-python`，但是在示例中需要使用该模块对图像进行处理。
2. 在Debian10固件上通过 `pip3` 安装 `numpy` 可能失败，建议用上述方法安装。
  - 安装RKNN3-Toolkit Lite  
各平台的安装包都放在SDK的 `rknn3-toolkit-lite/packages` 文件夹下。进入 `packages` 文件夹，执行以下命令安装RKNN3-Toolkit Lite：

```
# Python 3.9  
pip3 install rknn3_toolkit_lite-x.y.z-cp39-cp39-linux_aarch64.whl  
# Python 3.10  
pip3 install rknn3_toolkit_lite-x.y.z-cp310-cp310-linux_aarch64.whl  
# Python 3.11  
pip3 install rknn3_toolkit_lite-x.y.z-cp311-cp311-linux_aarch64.whl  
# Python 3.12  
pip3 install rknn3_toolkit_lite-x.y.z-cp312-cp312-linux_aarch64.whl
```

### 3.6.3 基本使用流程

使用RKNN3-Toolkit Lite部署RKNN模型的基本流程如下图所示：

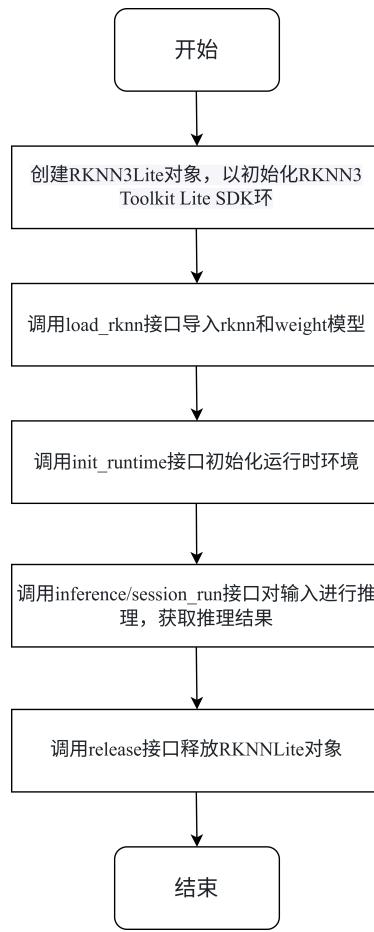


图3-5 RKNN3-Toolkit Lite基本使用流程

注意：

1. 在调用 `inference` 或者 `session_run` 接口进行推理之前，需要准备输入数据并进行相应的预处理。如果是纯LLM需要获取 `input_ids` 数据，若是多模态模型则要获取 `embeds` 输入，然后根据输入信息设置推理接口中的各项参数。
2. 在调用推理接口后，通常需要对推理结果进行相应的处理，以完成应用程序相关功能。

### 3.6.4 运行参考示例

在 `SDK/rknn3-toolkit-lite/examples` 目录，提供了一个基于RKNN3-Toolkit Lite开发的VLM应用。该应用使用RKNN3-Toolkit Lite接口加载Qwen2.5VL-3B RKNN模型和Tokenizer，进行输入图片和Prompt的推理，并输出分析结果。

运行该示例的步骤：

1. 准备一块安装有RKNN3-Toolkit Lite的开发板；
2. 将 `SDK/rknn3-toolkit-lite/examples` 目录推到开发板上；
3. 在开发板上进入 `examples/Qwen2.5VL` 目录，执行如下命令运行该示例：

```
python test.py \
--rknn_vision_path XXX/Qwen2.5-VL-3B-vision.rknn \
--rknn_llm_path XXX/Qwen2.5-VL-3B-llm.rknn \
--tokenizer_path XXX/Tokenizer_Path \
--embed_path XXX/Qwen2.5-VL-3B-llm.embed.bin
```

注意：请确保所指定的 RKNN 模型路径与配套权重文件路径一致，否则可能导致推理失败。

示例运行结果（部分）：

```
--> Running vision model
这张图片展示了一个太空场景。背景中可以看到一个巨大的行星，可能是月球或火星的表面，上面有一些坑洞和山脉。天空中有许多星星点缀其中。

前景中有一个穿着宇航服的人躺在地上休息。宇航服看起来是白色的，并且有多个口袋和按钮。这个人似乎在放松，手里拿着一瓶饮料（可能是啤酒），旁边还有一个小盒子。地面上还有一些物品散落着，包括一个梯子、一块石头和其他一些不明物体。

整个场景给人一种太空探索或月球任务的感觉，但同时也带有一种轻松的氛围，因为宇航员看起来像是在休息和享受片刻的放松时间。

-----Finished-----
```

注意：由于模型版本、输入图像或运行环境的差异，实际输出结果可能与上述示例略有不同。

### 3.6.5 RKNN3 Toolkit Lite API详细说明

本章节将详细说明RKNN3-Toolkit Lite提供的所有API的用法。

#### 3.6.5.1 RKNNLite初始化及对象释放

在使用 RKNN3-Toolkit Lite 时，需要先调用 `RKNN3Lite()` 方法创建一个 `RKNN3Lite` 对象，并在使用完毕后调用该对象的 `release()` 方法释放相关资源。如果加载的是大语言模型（LLM），则需在初始化时设置参数 `llm_mode=True`。

具体说明如下：

API	RKNN3Lite
描述	RKNNLite初始化
参数	<code>llm_mode</code> : 是否为LLM模式
	<code>verbose</code> : 是否在终端中打印详细日志
	<code>verbose_file</code> : 日志文件路径，如果指定，将日志写入该文件
返回值	返回0表示成功，非0表示失败

此外，初始化 `RKNN3Lite` 对象时，可通过 `verbose` 和 `verbose_file` 参数控制日志输出行为：

- `verbose=True` 表示在终端中打印详细日志；
- 若同时指定了 `verbose_file`（例如 `'./inference.log'`），则日志信息还会被写入该文件中。

举例如下：

```
# 将详细日志信息输出到屏幕，并写到inference.log文件中
rknn_lite = RKNN3Lite(verbose=True, verbose_file='./inference.log')
# 只在屏幕打印详细日志信息
rknn_lite = RKNN3Lite(verbose=True)
# 初始化LLM模型
rknn_lite = RKNN3Lite(llm_mode=True)
...
rknn_lite.release()
```

### 3.6.5.2 加载RKNN模型

将转换好的RKNN模型通过 `load_rknn` 接口加载到runtime中。

API	<code>load_rknn</code>
描述	加载RKNN模型
参数	<code>model_path</code> : RKNN模型路径
	<code>weight_path</code> : RKNN模型路径
返回值	返回0表示成功，非0表示失败

举例如下：

```
ret = rknn_lite.load_rknn(model_path='mobilenetv2-12.rknn', weight_path='mobilenetv2-12.weight')
```

### 3.6.5.3 初始化运行时环境

在模型推理之前，必须先初始化运行时环境。

API	<code>init_runtime</code>
描述	初始化运行时环境。
参数	<code>target</code> ：指定的协处理器，目前支持RK1820/RK1828 <code>core_mask</code> ：npu核心掩码，协处理器共包含 8 个 NPU 核心，该参数通过位掩码指定启用的核心。例如 0x0f(即二进制 0b00001111)，表示使用低4位npu核心。注意该参数需要和转rknn模型的core_mask保持一致，否则会报错 <code>llm_args</code> ：LLM 模型的配置参数字典 <code>llm_callback</code> ：LLM模型的回调函数可用于流式推理 <code>llm_embed_path</code> ：embedding词表路径，用于初始化LLMGetEmbedCallback 所需的参数 <code>device_id</code> ：设备id,用于区分多个设备，如果只连接了一个设备可以置为None
返回值	0: 初始化运行时环境成功； -1: 初始化运行时环境失败。

注：

- `llm_args` 目前可配置的参数及其含义如下：
  - `top_k`：采样时考虑的最高概率词汇的数量，整数类型，默认值根据模型而定。
  - `top_p`：核采样概率阈值，浮点数类型，取值范围 [0, 1]，用于控制采样的多样性。
  - `temperature`：控制随机性的温度参数，浮点数类型，通常大于 0，越小生成越确定。
  - `repeat_penalty`：重复惩罚系数，浮点数类型，用于减少重复生成的惩罚。
  - `frequency_penalty`：频率惩罚，浮点数类型，基于词汇出现频率的惩罚。
  - `presence_penalty`：存在惩罚，浮点数类型，基于词汇是否出现过的惩罚。
  - `vocab_size`：词汇表大小，整数类型，表示模型词汇表的总大小。
  - `special_bos_id`：开始标记 (Beginning of Sequence) 的ID，整数类型。
  - `special_eos_id`：结束标记 (End of Sequence) 的ID，整数类型。
  - `linefeed_id`：换行符的ID，整数类型。
  - `skip_special_token`：是否跳过特殊标记，布尔类型，True 表示跳过。
  - `ignore_eos_token`：是否忽略结束标记，布尔类型，True 表示忽略。
  - `keep_history`：是否保持对话历史，布尔类型，True 表示保持。
  - `max_new_tokens`：最大新生成标记数，整数类型，控制生成文本的最大长度。
  - `logits_name`：logits 输出的名称，字符串类型，用于指定输出层名称。
  - `max_context_len`：最大上下文长度，整数类型，表示模型能处理的上下文最大长度。
- `llm_callback` 主要包含5个回调函数，用于处理LLM推理过程中的不同阶段：
  - `LLMResultCallback`：处理推理结果的回调函数，用于接收和处理生成的文本结果。

- `LLMSamplingCallback`: 采样回调函数，用于自定义采样策略或处理采样过程。
  - `LLMTokenizerCallback`: 分词器回调函数，用于处理输入文本的分词和编码。
  - `LLMGetEmbedCallback`: 获取嵌入回调函数，用于获取输入的嵌入向量。
  - `LLMGetLastHiddenLayerCallback`: 获取最后隐藏层回调函数，用于获取模型的最后隐藏层输出。
- 注意：这些回调函数的具体实现和参数与 API 手册中提供的函数和参数一一对应，用户可根据需要自定义实现。
- `device_id` 可以通过 `get_devices_id` 函数获取所有协处理器的设备ID。

传统模型举例如下：

```
...
# 获取device id
device_id = rknn_lite.get_devices_id()
# 初始化运行时环境
ret = rknn_lite.init_runtime(target='rk1820', core_mask=0x01, device_id = device_id[0])
if ret != 0:
    print('Init runtime environment failed')
    exit(ret)
```

大模型举例如下：

```
ARGS = [{"max_new_tokens":256, "top_k":1, "repeat_penalty":1.1, "special_eos_id": 151645, ...}]
callback = RKLLMCallback()
...

ret = rknn_lite.init_runtime(target='rk1820', core_mask=0xff, llm_args=ARGS, llm_callback=callback)
if ret != 0:
    print('Init runtime environment failed')
    exit(ret)
```

### 3.6.5.4 模型推理

传统模型的inference接口详细说明

API	<code>inference</code>
描述	运行传统模型推理。
参数	<code>inputs</code> ：输入数据列表，每个元素为numpy.ndarray。
	<code>data_format</code> ：数据格式列表，每个元素为字符串，可选 'undefined'、'nhwc' 或 'nchw'。'nhwc' 和 'nchw' 仅对4维输入有效。对于非四维输入，应设为 undefined 或留空。默认值为 None。
返回值	输出数据列表，每个元素为numpy.ndarray；失败时返回None。

大模型的session\_run接口详细说明

API	<code>session_run</code>
描述	运行LLM模型推理，支持流式输出。
参数	<code>inputs</code> ：输入数据列表，用于多模态模型（如图像、音频等）。类型为列表，元素可以是 RKNN3Image、RKNN3Audio、RKNN3Video 或 RKNN3AuxTensor 等专用类型。
	<code>prompt</code> ：文本提示，作为LLM模型的输入。
	<code>embeds</code> ：输入嵌入向量，用于跳过tokenization的场景。类型为numpy.ndarray，通常为3D张量 (batch_size, sequence_length, hidden_size)。注意：prompt 和 embeds 互斥，不能同时提供。
返回值	返回两个值： <ul style="list-style-type: none"> <li>• <code>ret</code>：0 表示推理成功，-1 表示失败；</li> <li>• 第二个值为性能统计信息列表，格式为 <code>[n_decode_tokens, n_prefill_tokens, llm_start_time, llm_end_time]</code>。</li> </ul>

传统模型推理代码参考如下：

```

...
# 运行推理
outputs = rknn_lite.inference(inputs=[img])
if outputs is None:
    print('Inference failed')
    exit(-1)
# 处理输出
print(outputs)

```

LLM模型推理代码参考如下：

```

...
prompt = "介绍一下LLM模型的工作原理。"
ret, [n_decode_tokens, n_prefill_tokens, llm_start_time, llm_end_time] = rknn_llm.session_run(prompt=prompt)
if ret != 0:
    print('RKNN llm inference failed!')
    exit(ret)

```

VLM模型推理代码参考如下：

```

...
# 获取视觉模型的输出并转换为所需格式
outputs = np.float16(np.expand_dims(rknn_vision.inference(inputs=[feature])[0], 0))

prompt = "<image>请描述图片内容"
inputs = []
llm_input = RKNN3Image()
llm_input.image_embed = outputs.ctypes.data_as(ctypes.POINTER(Float16))
llm_input.n_image_tokens = outputs.shape[1]
llm_input.n_image = outputs.shape[0]
llm_input.image_width = 392
llm_input.image_height = 392
llm_input.image_start = "<|vision_start|>".encode('utf-8')
llm_input.image_end = "<|vision_end|>".encode('utf-8')
llm_input.image_content = "<|image_pad|>".encode('utf-8')
inputs.append(llm_input)

# 运行LLM推理
ret, [n_decode_tokens, n_prefill_tokens, llm_start_time, llm_end_time] = rknn_llm.session_run(inputs=inputs,
prompt=prompt)
if ret != 0:
    print('RKNN LLM inference failed!')
    exit(ret)

```

## 4. RKNN3高级功能

### 4.1 数据排列格式

目前RKNN的数据排列格式主要有以下四种，`NHWC`、`NCHW`、`NC1HWC2`、`UNDEFINE`。

其中`NHWC`和`NCHW`的数据排布为深度学习常见数据排列方式，本章节不做额外说明，重点讲述RKNPU硬件专用的`NC1HWC2`数据格式的存储以及转换。

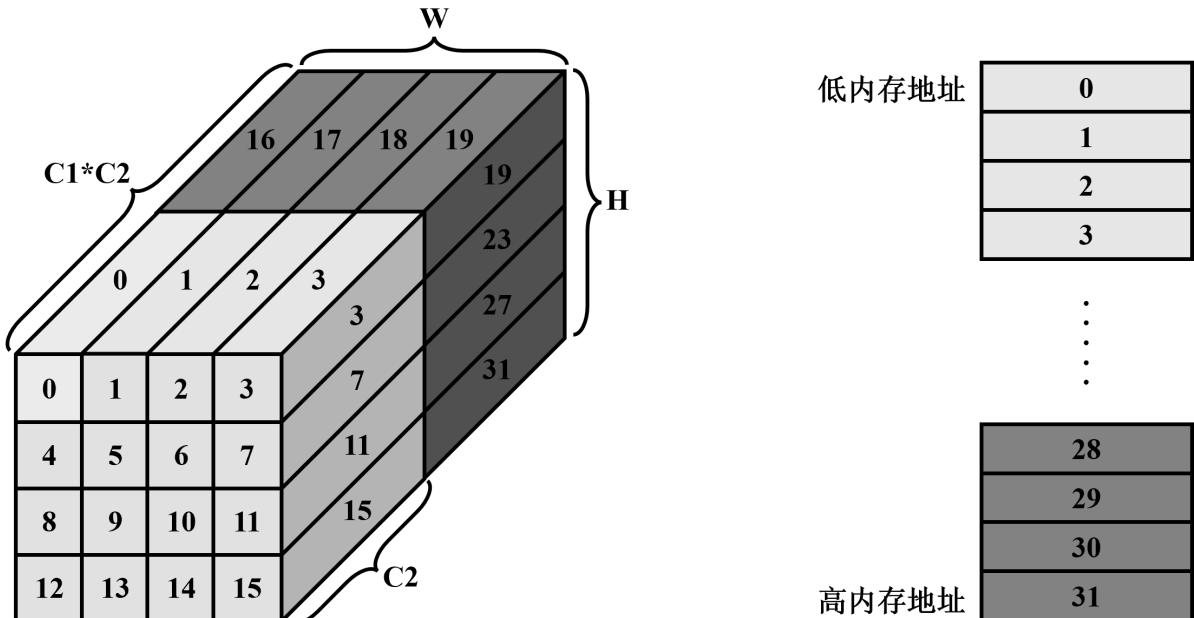


图4-1 RKNPU NC1HWC2数据排布与存储

如图4-1所示，数字0代表一笔数据，即一次存放C2个数据，其中C2是由平台决定的，不同硬件平台的C2的规则约束由表5-1所示，C1为C/C2的上取整值。NC1HWC2数据存放的顺序与图中数值增长的顺序一致，先存放0-15的数据，再存放16-31的数据。以C2为8为例，当feature为(1,13,4,4)，对应的NC1HWC2为(1,2,4,4,8)，此时C2为8，C1为2，feature在内存中在16-31排放的数据中，对应的每个C2数据块只有前5个数据有效，剩下的3个数据是额外补的对齐数据。

表4-1 不同硬件平台对应的C2值

	RK1820
int8	32
float16	16

接下来重点介绍NC1HWC2数据排列转NCHW和NHWC数据在内存中的变化过程。

以feature (1, 13, 2, 2), C2为8为例，数据在内存排布中的转换，根据前文的对齐要求可知feature(1, 13, 2, 2) 对应的NC1HWC2 为(1, 2, 2, 2, 8) , NC1HWC2 的存储如下图所示，红色部分为额外对齐的无效数据。

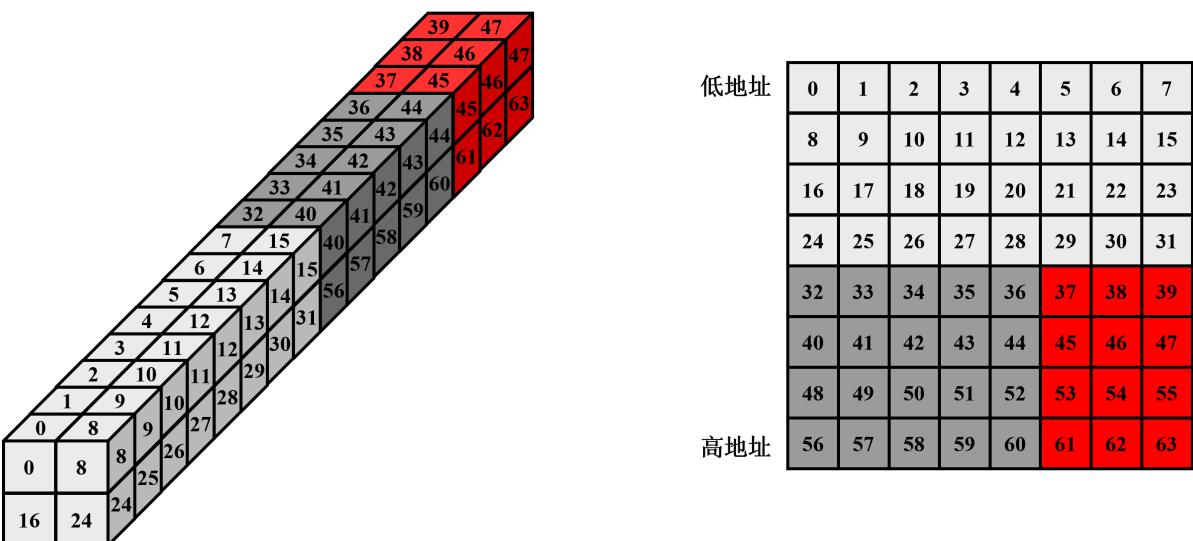


图4-2 NC1HWC2数据排布展开

移除无效数据转成NCHW即(1, 13, 2, 2)数据，在内存中的排布如下：

低地址	0	8	16	24
	1	9	17	25
	2	10	18	26
	3	11	19	27
	4	12	20	28
	5	13	21	29
	6	14	22	30
	7	15	23	31
	32	40	48	56
	33	41	49	57
	34	42	50	58
	35	43	51	59
高地址	36	44	52	60

图4-3 NCHW 数据排布

移除无效数据转成 NHWC 即 (1, 2, 2, 13)数据，在内存中的排布如下：

低地址	0	1	2	3	4	5	6	7	32	33	34	35	36
	8	9	10	11	12	13	14	15	40	41	42	43	44
	16	17	18	19	20	21	22	23	48	49	50	51	52
高地址	24	25	26	27	28	29	30	31	56	57	58	59	60

图4-4 NHWC 数据排布

转换示例代码：

NC1HWC2 转 NCHW：以int8数据排列的 NC1HWC2 转成int8数据排列的 NCHW 如下所示：

```

/*
 *src: 表示NC1HWC2输入tensor的地址
 *dst: 表示NCHW输出tensor的地址
 *dims: 表示NC1HWC2的shape信息
 *channel: 表示NCHW输入的C的值
 * h : 表示NCHW的h的值
 * w: 表示NCHW的w的值
 */
int NC1HWC2_to_NCHW(const int8_t* src, int8_t* dst, int* dims, int channel, int h, int w)
{
    int batch = dims[0];
    int C1 = dims[1];
    int C2 = dims[4];
    int hw_src = dims[2] * dims[3];
    int hw_dst = h * w;
    for (int i = 0; i < batch; i++) {
        src = src + i * C1 * hw_src * C2;
        dst = dst + i * channel * hw_dst;
        for (int c = 0; c < channel; ++c) {
            int plane = c / C2;
            const int8_t* src_c = plane * hw_src * C2 + src;
            int offset = c % C2;
            for (int cur_h = 0; cur_h < h; ++cur_h)
                for (int cur_w = 0; cur_w < w; ++cur_w) {
                    int cur_hw = cur_h * w + cur_w;
                    dst[c * hw_dst + cur_h * w + cur_w] = src_c[C2 * cur_hw + offset];
                }
        }
    }
}

```

```
    return 0;
}
```

NC1HWC2 转 NHWC：以int8数据排列的 NC1HWC2 转成int8数据排列的 NHWC 如下所示：

```
/*
 *src: 表示NC1HWC2输入tensor的地址
 *dst: 表示NCHW输出tensor的地址
 *dims: 表示NC1HWC2的shape信息
 *channel: 表示NHWC输入的C的值
 * h : 表示NCHW的h的值
 * w: 表示NCHW的w的值
 */
int NC1HWC2_to_NHWC(const int8_t* src, int8_t* dst, int* dims, int channel, int h, int w)
{
    int batch = dims[0];
    int C1 = dims[1];
    int C2 = dims[4];
    int hw_src = dims[2] * dims[3];
    int hw_dst = h * w;
    for (int i = 0; i < batch; i++) {
        src = src + i * C1 * hw_src * C2;
        dst = dst + i * channel * hw_dst;
        for (int cur_h = 0; cur_h < h; ++cur_h) {
            for (int cur_w = 0; cur_w < w; ++cur_w) {
                int cur_hw = cur_h * dims[3] + cur_w;
                for (int c = 0; c < channel; ++c) {
                    int plane = c / C2;
                    const auto* src_c = plane * hw_src * C2 + src;
                    int offset = c % C2;
                    dst[cur_h * w * channel + cur_w * channel + c] = src_c[C2 * cur_hw + offset];
                }
            }
        }
    }
    return 0;
}
```

## 4.2 NPU多核配置

本章节将详细介绍多核NPU的配置方法，以提高模型的推理效率。

注意：多核运行适用于网络层计算量较大的网络，对小网络提升幅度较小，甚至可能因为部分算子在单核多核之间切换（该切换需CPU介入）而导致性能下降。

### 4.2.1 当前多核运行限制

- CNN模型只支持单核推理，不支持模型拆分成多核推理，但支持多batch多核。
- LLM/VLM/ViT模型支持多核推理。
- 不支持同一时刻推理两个及以上的多核模型，但如果多核模型是串行推理，则是支持的。
- 支持同一时刻推理多个单核模型。
- 支持同一时刻推理一个多核模型加多个单核模型。
- 用户需要在转换模型时指定该模型在板端推理时占用的核心数，在板端推理时，指定的core mask需要与转换模型时指定的core num 匹配。

### 4.2.2 模型转换时的核心配置

通过RKNN3-Toolkit的 `config()` 接口中的 `core_num` 参数设置模型运行的 NPU 核心。

参数	详细说明
core_num	core_num：模型需要用到的NPU核心数。RK1820的NPU核心数范围为1到8，默认值为0表示自动 注意： 1. Transformer类模型可以根据需要设置使用的NPU核心数，其他模型（如CNN模型）建议设为1。 2. 自动模式下，在使用load_llm加载的模型会使用8个NPU核心，其余情况使用1个NPU核心。

示例如下：

```
# 视觉模型核心配置示例
rknn.config(target_platform='RK1820',
            quantized_dtype='w4a16',
            quantized_algorithm='normal',
            quantized_method='group32',
            core_num=8) # 指定使用8个核心
```

### 4.2.3 运行时核心掩码

如果使用Python作为应用程序开发语言，可以通过RKNN3-Toolkit的`init_runtime()`接口中的`core_mask`参数设置模型运行的NPU核心。该参数的详细说明如下表：

参数	详细说明
<code>core_mask</code>	模型连板推理的运行核心，如果连板推理时，需要指定该参数，支持从0x1到0xff的掩码值。默认值为-1

示例如下：

```
## Python
.....
# Init runtime environment
print('--> Init runtime environment')
ret = rknn.init_runtime(target='rk1820', core_mask=0xff)
if ret != 0:
    print('Init runtime environment failed!')
    exit(ret)
```

如果使用C/C++作为应用程序开发语言，可以调用`rknn3_model_init()`接口，通过配置`rknn3_config`的`run_core_mask`设置模型执行的核心掩码。`rknn3_config`的成员变量详细说明如下表：

成员变量	数据类型	含义
<code>priority</code>	<code>int32_t</code>	运行优先级
<code>run_timeout</code>	<code>uint32_t</code>	运行超时时间（毫秒），0表示无超时
<code>run_core_mask</code>	<code>uint32_t</code>	模型执行的核心掩码
<code>user_mem_weight</code>	<code>uint8_t</code>	权重内存是否由用户分配
<code>user_mem_internal</code>	<code>uint8_t</code>	内部内存是否由用户分配
<code>user_sram</code>	<code>uint8_t</code>	SRAM内存是否由用户分配
<code>reserved</code>	<code>uint8_t</code>	保留字段

使用C/C++ API设置模型运行NPU核心，参考代码如下：

```
// C++
rknn3_config config;
memset(&config, 0, sizeof(config));
config.run_core_mask = core_mask;

//Init RKNN Model
ret = rknn3_model_init(ctx, &config);
if (ret < 0) {
    printf("rknn_model_init failed! ret=%d\n", ret);
    return ret;
}
```

## 4.3 动态Shape

### 4.3.1 动态Shape功能介绍

动态shape是指模型输入数据的形状在运行时可以改变。它可以帮助处理输入数据大小不固定的情况，增加模型的灵活性。在之前仅支持静态shape的RKNN模型情况下，如果用户需要使用多个输入shape，传统的做法是生成多个RKNN模型，在模型部署时初始化多个上下文分别执行推理，而在引入动态shape后，用户可以只保留一份与静态shape RKNN模型大小接近的动态shape RKNN模型，并使用一个上下文进行推理，从而节省Flash占用和DDR占用，动态shape在图像处理和序列模型推理中具有重要的作用，它的典型应用场景包括：

- 序列长度改变的模型，常见于NLP模型，例如BERT, GPT
- 空间维度变化的模型，例如分割和风格迁移
- 带Batch模型，Batch维度上变化
- 可变输出数量的目标检测模型

### 4.3.2 生成动态Shape的RKNN模型

本节介绍使用RKNN3-Toolkit的Python接口生成动态shape的RKNN模型的步骤：

#### 1. 确认模型支持动态shape

如果模型文件本身不是动态shape, RKNN3-Toolkit支持扩展成动态shape的RKNN模型。首先，用户要确认模型本身不存在限制动态shape的算子或子图结构，例如，常量的形状无法改变，RKNN3-Toolkit工具在转换过程会报错，如果遇到不支持动态shape扩展的情况，用户要根据报错信息，修改模型结构，重新训练模型以支持动态shape。建议使用原始模型本身就是动态shape的模型。

#### 2. 设置需要使用的输入形状

由于NPU硬件特性，动态shape RKNN模型不支持输入形状任意改变，要求用户设置有限个输入形状。对于多输入的模型，每个输入的shape个数要相同。Python代码示例如下：

```
# Python
dynamic_shapes = [
    [[1, 3, 224, 224]], # set the first shape for all inputs
    [[1, 3, 192, 192]], # set the second shape for all inputs
    [[1, 3, 160, 160]], # set the third shape for all inputs
]
# Pre-process config
rknn.config(mean_values=[[0, 0, 0]], std_values=[[255, 255, 255]], dynamic_input=dynamic_shapes)
```

上述接口配置会生成支持3个shape分别是[1,3,224,224]、[1,3,192,192]和[1,3,160,160]的动态shape RKNN模型。

`dynamic_input` 中的shape与原始模型框架的layout一致。例如，对于相同的224x224大小的RGB图片做分类，TensorFlow/TFLite模型输入是[1,224,224,3]，而ONNX模型输入是[1,3,224,224]。

#### 3. 量化

在设置好输入shape后，如果要做量化，则需要设置量化矫正集数据。工具会读取用户设置的最大分辨率输入做量化（是所有输入尺寸之和的最大的一组shape）。例如，模型有两个输入，一个输入shape分别是[1,224]和[1,112]，另一个输入shape分别[1,40]和[1,80]，第一组shape所有输入尺寸之和是 $1 \times 224 + 1 \times 40 = 264$ ，第二组shape所有输入尺寸之和是 $1 \times 112 + 1 \times 80 = 192$ ，第一组shape所有输入尺寸之和更大，因此使用两个输入分别以[1,224]和[1,40]的shape做量化。

- 如果量化矫正集是jpg/png图片格式，用户可以使用不同的分辨率的图片做量化，因为工具会对图片使用opencv的resize方法缩放到最大分辨率后做量化。
- 如果量化矫正集是npy格式，则用户必须使用最大分辨率输入的shape。量化后，模型内所有shape在运行时使用同一套量化参数进行推理。

#### 4. 推理评估或精度分析

动态shape RKNN模型做推理或做精度分析时，用户必须提供第2步中设置的其中一组shape的输入。接口使用上与静态shape RKNN模型场景一致，此处不做赘述。

### 4.3.3 C API部署

得到动态shape RKNN模型后，接着使用RKNPU3 C API进行部署。使用API部署动态shape RKNN模型的流程如下图所示：

图4-5 动态shape输入API调用流程

加载动态 Shape 的 RKNN 模型后，可以在运行时灵活切换输入的形状。通过 `rknn_query()` 可以获取模型的输入输出数量、动态 Shape 的配置信息以及所有可用的 Shape 列表。每个输入支持的 Shape 信息以 `rknn3_shape_info` 返回，包含 Shape ID、输入输出个数和布局等信息。模型默认使用 Shape ID 0，如果不需要改变 Shape，直接调用 `rknn3_create_mem()` 创建输入输出内存，再调用 `rknn3_mem_sync()` 将输入数据同步到设备端即可。

若需要切换 Shape，则需要先调用 `rknn3_destroy_mem()` 释放之前分配的输入输出内存，再调用 `rknn3_set_shape()` 设置新的 Shape ID，然后重新执行 `rknn3_create_mem()` 和 `rknn3_mem_sync()` 以创建适配新 Shape 的内存并同步数据。准备完成后，即可通过 `rknn3_run()` 执行推理，并使用 `rknn3_mem_sync()` 将输出数据同步回主机端。

#### 1. 初始化

调用 `rknn3_init()`、`rknn3_load_model_from_path()`、`rknn3_model_init()` 接口加载并初始化动态shape RKNN模型，对于动态shape RKNN模型，在初始化上下文时有如下限制：

- 不支持权重共享功能（带 `RKNN_FLAG_SHARE_WEIGHT_MEM` 标志的初始化）。
- 不支持上下文复用功能（具体说明见 `rknn3_dup_context` 接口）。

## 2. 查询RKNN模型支持的输入shape组合

初始化成功后，通过 `rknn_query()` 可以获取模型的输入输出数量、动态 Shape 的配置信息以及所有可用的 Shape 列表。每个输入支持的 Shape 信息以 `rknn3_shape_info` 返回，包含 Shape ID、输入输出个数和布局等信息。C代码示例如下：

```
// Get Model Input Output Number
rknn3_input_output_num io_num;
ret = rknn3_query(ctx, RKNN3_QUERY_IN_OUT_NUM, &io_num, sizeof(io_num));
if (ret < 0)
{
    printf("rknn_query fail! ret=%d\n", ret);
    return ret;
}
printf("model input num: %d, output num: %d\n", io_num.n_input, io_num.n_output);

rknn3_shape_config shape_config;
ret = rknn3_query(ctx, RKNN3_QUERY_DYNAMIC_SHAPE_CONFIG, &shape_config, sizeof(rknn3_shape_config));
if (ret < 0)
{
    printf("rknn_query fail! ret=%d\n", ret);
    return ret;
}
printf("dynamic shape config: n_shapes=%d, current_shape_id=%d\n",
       shape_config.n_shapes, shape_config.current_shape_id);

rknn3_shape_info shape_infos[shape_config.n_shapes];

for (uint32_t i = 0; i < shape_config.n_shapes; i++) {
    shape_infos[i].shape_id      = i;
    shape_infos[i].input_attrs   = (rknn3_tensor_attr*)malloc(sizeof(rknn3_tensor_attr) * io_num.n_input);
    shape_infos[i].output_attrs = (rknn3_tensor_attr*)malloc(sizeof(rknn3_tensor_attr) * io_num.n_output);

    if (!shape_infos[i].input_attrs || !shape_infos[i].output_attrs) {
        printf("Failed to allocate memory for shape attributes\n");
        rknn3_destroy(ctx);
        return -1;
    }
}

ret = rknn3_query(ctx, RKNN3_QUERY_DYNAMIC_SHAPE_INFO, shape_infos, sizeof(rknn3_shape_info) *
shape_config.n_shapes);
if (ret < 0)
{
    printf("rknn_query fail! ret=%d\n", ret);
    return ret;
}

// print all shape infos
for (uint32_t i = 0; i < shape_config.n_shapes; i++) {
    printf("Shape %d (ID: %d)%s:\n", i, shape_infos[i].shape_id, shape_infos[i].is_default ? " [Default]" :
": ");

    for (uint32_t j = 0; j < shape_infos[i].n_inputs; j++) {
        rknn3_tensor_attr* attr = &shape_infos[i].input_attrs[j];
        printf("  Input %d (%s): [%s, %s]\n", attr->index, attr->name);
        for (uint32_t k = 0; k < attr->n_dims; k++) {
            printf("%d%s", attr->shape[k], (k < attr->n_dims - 1) ? ", " : "");
        }
        printf("] Aligned size: %lu bytes\n", attr->aligned_size);
    }
}
```

注意：对于多输入的模型，所有输入的shape按顺序一一对应，例如，有两个输入、多种shape的RKNN模型，第一个输入的第一个shape与第二个输入的第一个shape组合有效，不存在交叉的shape组合。例如，模型有两个输入A和B，A的shape分别是[1,224]和[1,112]，B的shape分别[1,40]和[1,80]，此时，只支持以下两组输入shape的情况：

- A shape = [1,224],B shape=[1,40]
- A shape = [1,112],B shape=[1,80]

## 3. 设置输入shape

模型默认使用 Shape ID 0，在输入数据shape发生改变时，需要调用 `rknn3_set_shape()` 接口传入 Shape ID修改输入shape。

```
int rknn3_set_shape(rknn3_context context, int32_t shape_id);
```

接口说明：

API 名称	<b>rknn3_set_shape</b>
功能	为动态输入设置模型形状
参数	rknn3_context context: RKNN3运行时上下文
	int32_t shape_id: 要设置的形状的ID
返回值	int: 返回0表示成功, 非0表示失败

示例：

```
int shapeID = 0;
ret = rknn3_set_shape(ctx, shapeID);
if (ret < 0)
{
    printf("rknn3_set_shape fail! ret=%d\n", ret);
    return ret;
}
```

在设置输入shape后，可以再次调用 rknn\_query 查询当前设置成功的输入和输出shape，C代码示例如下：

```
// Get Model Input Info
printf("input tensors:\n");
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 < 0)
    {
        printf("rknn_query fail! ret=%d\n", ret);
        return ret;
    }
    dump_tensor_attr(&(input_attrs[i]));
}

// Get Model Output Info
printf("output tensors:\n");
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 < 0)
    {
        printf("rknn_query fail! ret=%d\n", ret);
        return ret;
    }
    dump_tensor_attr(&(output_attrs[i]));
}
```

#### 4. 分配内存

确定好输入的shape后，需要调用 `rknn3_create_mem()` 创建输入输出内存，C代码示例如下：

```
// Set to context
app_ctx->inputs = (rknn3_tensor*)malloc(io_num.n_input * sizeof(rknn3_tensor));
app_ctx->outputs = (rknn3_tensor*)malloc(io_num.n_output * sizeof(rknn3_tensor));
app_ctx->rkn3_ctx = ctx;
app_ctx->io_num = io_num;
for (int i = 0; i < app_ctx->io_num.n_input; i++) {
    app_ctx->inputs[i].mem = rknn3_create_mem(ctx, input_attrs[i].aligned_size, input_attrs[i].core_id,
                                                RKNN3_FLAG_MEMORY_CACHEABLE);
    app_ctx->inputs[i].attr = (rknn3_tensor_attr*)malloc(sizeof(rknn3_tensor_attr));
    memcpy(app_ctx->inputs[i].attr, &(input_attrs[i]), sizeof(rknn3_tensor_attr));
}
```

```

for (int i = 0; i < app_ctx->io_num.n_output; i++) {
    app_ctx->outputs[i].mem = rknn3_create_mem(ctx, output_attrs[i].aligned_size,
output_attrs[i].core_id,
        RKNN3_FLAG_MEMORY_CACHEABLE);
    app_ctx->outputs[i].attr = (rknn3_tensor_attr*)malloc(sizeof(rknn3_tensor_attr));
    memcpy(app_ctx->outputs[i].attr, &(output_attrs[i]), sizeof(rknn3_tensor_attr));
}

```

若shape没有改动则这些内存可以复用到程序结束，若shape有改动或者程序结束时需要调用 `rknn3_destroy_mem()` 释放之前分配的输入输出内存，C代码示例如下：

```

for (int i = 0; i < app_ctx->io_num.n_input; i++) {
    if (app_ctx->inputs && app_ctx->inputs[i].mem) {
        rknn3_destroy_mem(app_ctx->rknn_ctx, app_ctx->inputs[i].mem);
    }
    if (app_ctx->inputs && app_ctx->inputs[i].attr != NULL) {
        free(app_ctx->inputs[i].attr);
        app_ctx->inputs[i].attr = NULL;
    }
}
if (app_ctx->inputs) {
    free(app_ctx->inputs);
    app_ctx->inputs = NULL;
}
for (int i = 0; i < app_ctx->io_num.n_output; i++) {
    if (app_ctx->outputs && app_ctx->outputs[i].mem) {
        rknn3_destroy_mem(app_ctx->rknn_ctx, app_ctx->outputs[i].mem);
    }
    if (app_ctx->outputs && app_ctx->outputs[i].attr != NULL) {
        free(app_ctx->outputs[i].attr);
        app_ctx->outputs[i].attr = NULL;
    }
}

```

## 5. 推理

在设置好当前输入shape后并分配好内存后，在推理之前需要调用 `rknn3_mem_sync` 将数据同步到RK1820/RK1828，推理后同样需要掉用 `rknn3_mem_sync` 将数据从设备同步回主机，C代码示例如下：

```

// Set Input Data
memcpy(app_ctx->inputs[0].mem->virt_addr, (uint8_t*)img.virt_addr, img.size);

// Sync inputs
for (int i = 0; i < app_ctx->io_num.n_input; i++)
{
    printf("-->rknn3_mem_sync input[%d]\n", i);
    ret = rknn3_mem_sync(app_ctx->rknn_ctx, app_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);
        goto out;
    }
}

// Run
printf("-->rknn_run\n");
ret = rknn3_run(app_ctx->rknn_ctx, app_ctx->inputs, app_ctx->io_num.n_input, app_ctx->outputs, app_ctx->io_num.n_output);
if (ret < 0)
{
    printf("rknn_run fail! ret=%d\n", ret);
    goto out;
}

// Sync outputs
for (int i = 0; i < app_ctx->io_num.n_output; i++)
{
    printf("-->rknn3_mem_sync output[%d]\n", i);
    ret = rknn3_mem_sync(app_ctx->rknn_ctx, app_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);
        goto out;
    }
}

```

```
        goto out;
    }
}
```

## 4.4 RKNN3 Session高级用法

### 4.4.1 LLM 输入类型设置

#### 4.4.1.1 prompt 输入

用户可以直接输入自然语言文本，通过设置为 prompt 输入类型，实现向模型输入原始文本，由 RKNN3 推理库完成文本到 token id 的分词等转换流程。该方式适合无需自行处理 tokenizer 的场景，推荐大部分通用用例采用。下面介绍 prompt 类型输入的具体构建方法。

示例代码：

```
rknn3_llm_input llm_inputs[1];
rknn3_llm_input input;
rknn3_llm_tensor input_tensor;
memset(&input, 0, sizeof(rknn3_llm_input));
memset(&input_tensor, 0, sizeof(rknn3_llm_tensor));

char *prompt = "Please explain the basic concept of relativity";
input_tensor = {.name = NULL, .prompt = prompt, .embed = NULL, .tokens = NULL, .n_tokens = 0, .enable_thinking = false};
input.input_type = RKNN3_LLM_INPUT_PROMPT;
input.llm_input = input_tensor;
llm_inputs[0] = input;
```

#### 4.4.1.2 token 输入

用户可自行准备 token 数据，并通过设置为 token 输入类型，实现向模型直接输入 token id 序列。这种方式适用于已经完成 tokenizer(分词)处理，希望跳过文本到 token id 转换过程的场景。下面介绍 token 类型输入的具体构建方法。

示例代码：

```
rknn3_llm_input llm_inputs[1];
rknn3_llm_input input;
rknn3_llm_tensor input_tensor;
memset(&input, 0, sizeof(rknn3_llm_input));
memset(&input_tensor, 0, sizeof(rknn3_llm_tensor));

int32_t tokens[6] = {151628, 151629, 151630, 151631, 151632, 151633};
uint64_t n_tokens = 6;
input_tensor = {.name = NULL, .prompt = NULL, .embed = NULL, .tokens = tokens, .n_tokens = n_tokens,
.enable_thinking = false};
input.input_type = RKNN3_LLM_INPUT_TOKEN;
input.llm_input = input_tensor;
llm_inputs[0] = input;
```

#### 4.4.1.3 embed 输入

用户可自行准备 embedding 数据，并通过设置为 embed 输入类型，实现向模型直接输入嵌入向量。该方式尤其适用于无需传统“文本->token id->embedding”流程的模型。下面介绍 embed 类型输入的具体构建方法。

示例代码：

```
rknn3_llm_input llm_inputs[1];
rknn3_llm_input input;
rknn3_llm_tensor input_tensor;
memset(&input, 0, sizeof(rknn3_llm_input));
memset(&input_tensor, 0, sizeof(rknn3_llm_tensor));

uint64_t n_tokens = 6;
int embedding_dim = 1024;
float16* embed = (float16*)malloc(n_tokens * embedding_dim * sizeof(float16));
if (!embed) {
    printf("malloc embed failed!\n");
    return -1;
}

input_tensor = {.name = NULL, .prompt = NULL, .embed = embed, .tokens = NULL, .n_tokens = n_tokens,
.enable_thinking = false};
```

```

input.input_type = RKNN3_LLM_INPUT_EMBED;
input.llm_input = input_tensor;
llm_inputs[0] = input;

```

#### 4.4.1.4 multimodal 输入

当模型具备多模态输入能力时，可以使用 multimodal 输入类型。目前支持的模态包括 image、audio 和 video。以 Qwen/Qwen2.5-Omni-3B 模型为例，以下示范如何同时构建包含 image、audio 和 video 的多模态输入。如果仅需支持部分模态（如仅 image），可只设置对应模态的字段，未涉及的模态字段不设置。

示例代码：

```

rknn3_llm_input llm_inputs[1];
rknn3_llm_input input;
rknn3_llm_multimodal_tensor input_tensor;
memset(&input, 0, sizeof(rknn3_llm_input));
memset(&input_tensor, 0, sizeof(rknn3_llm_multimodal_tensor));

int embedding_dim = 1024;
// size: n_image * n_image_tokens * embedding_dim * sizeof(float16)
float16* vision_output = (float16*)malloc(1 * 196 * embedding_dim * sizeof(float16));
if (!embed) {
    printf("malloc vision_output failed!\n");
    return -1;
}

// size: n_audio * n_audio_tokens * embedding_dim * sizeof(float16)
float16* audio_output = (float16*)malloc(1 * 75 * embedding_dim * sizeof(float16));
if (!embed) {
    printf("malloc audio_output failed!\n");
    return -1;
}

// size: n_video * n_frame_per_video * n_frame_tokens *embedding_dim * sizeof(float16)
float16* video_output = (float16*)malloc(1 * 2 * 196 * embedding_dim * sizeof(float16));
if (!embed) {
    printf("malloc video_output failed!\n");
    return -1;
}

input_tensor.name = NULL;
input_tensor.prompt = "You need to do three things: 1.<image>Describe the content of this image; 2.<audio>Transcribe this audio; 3.<video>Describe the content of this video.";
input_tensor.image.image_embed = vision_output;
input_tensor.image.n_image_tokens = 196;
input_tensor.image.n_image = 1; // The number of "<image>" tags in the prompt
input_tensor.image.image_width = 392;
input_tensor.image.image_height = 392;
input_tensor.image.image_start = (char*)"<|vision_bos|>";
input_tensor.image.image_end = (char*)"<|vision_eos|>";
input_tensor.image.image_content = (char*)"<|vision_pad|>";

input_tensor.audio.audio_embed = audio_output;
input_tensor.audio.n_audio_tokens = 75;
input_tensor.audio.n_audio = 1; // The number of "<audio>" tags in the prompt
input_tensor.audio.audio_start = (char*)"<|audio_bos|>";
input_tensor.audio.audio_end = (char*)"<|audio_eos|>";
input_tensor.audio.audio_content = (char*)"<|AUDIO|>";

input_tensor.video.video_embed = video_output;
input_tensor.video.n_frame_tokens = 196;
input_tensor.video.n_frame_per_video = 2;
input_tensor.video.n_video = 1; // The number of "<video>" tags in the prompt
input_tensor.video.video_start = (char*)"<|vision_eos|>";
input_tensor.video.video_end = (char*)"<|vision_eos|>";
input_tensor.video.video_content = (char*)"<|VIDEO|>";
input_tensor.video.frame_width = 392;
input_tensor.video.frame_height = 392;

input.input_type = RKNN3_LLM_INPUT_MULTIMODAL;
input.multimodal_input = input_tensor;

```

```
llm_inputs[0] = input;
```

#### 4.4.1.5 aux 输入

部分模型除了支持常规的 LLM 输入类型外，还会增加一些特殊的辅助输入。例如，Qwen/Qwen3-VL 系列模型引入了 3 个 deepstack 辅助输入。针对这类情况，需要利用 aux 辅助输入类型协同主输入共同构建完整的推理输入，实现扩展功能的灵活支持。下面以 Qwen/Qwen3-VL-2B-Instruct 模型为例，介绍 aux 输入的配置方法。

示例代码：

```
rknn3_llm_input llm_inputs[4]; // 1 multimodal + 3 aux
rknn3_llm_input input;
rknn3_llm_multimodal_tensor input_tensor;
rknn3_llm_input aux_input0;
rknn3_llm_input aux_input1;
rknn3_llm_input aux_input2;
rknn3_aux_tensor aux_input_tensor0;
rknn3_aux_tensor aux_input_tensor1;
rknn3_aux_tensor aux_input_tensor2;
memset(&input, 0, sizeof(rknn3_llm_input));
memset(&input_tensor, 0, sizeof(rknn3_llm_multimodal_tensor));
memset(&aux_input0, 0, sizeof(rknn3_llm_input));
memset(&aux_input1, 0, sizeof(rknn3_llm_input));
memset(&aux_input2, 0, sizeof(rknn3_llm_input));
memset(&aux_input_tensor0, 0, sizeof(rknn3_aux_tensor));
memset(&aux_input_tensor1, 0, sizeof(rknn3_aux_tensor));
memset(&aux_input_tensor2, 0, sizeof(rknn3_aux_tensor));

int embedding_dim = 2048;
// size: n_image * n_image_tokens * embedding_dim * sizeof(float16)
float16* vision_output = (float16*)malloc(1 * 144 * embedding_dim * sizeof(float16));
if (!embed) {
    printf("Malloc vision_output failed!\n");
    return -1;
}

input_tensor.name = NULL;
input_tensor.prompt = "<image>Describe the content of this image";
input_tensor.image.image_embed = vision_output;
input_tensor.image.n_image_tokens = 144;
input_tensor.image.n_image = 1; // The number of "<image>" tags in the prompt
input_tensor.image.image_width = 384;
input_tensor.image.image_height = 384;
input_tensor.image.image_start = (char*)<|vision_bos|>;
input_tensor.image.image_end = (char*)<|vision_eos|>;
input_tensor.image.image_content = (char*)<|vision_pad|>;
input.input_type = RKNN3_LLM_INPUT_MULTIMODAL;
input.multimodal_input = input_tensor;
llm_inputs[0] = input;

// size: n_image * n_image_tokens * vision_embed_dim * sizeof(float16)
aux_input_tensor0.mem = rknn3_create_mem(ctx, 1 * 144 * embedding_dim * sizeof(float16), 0,
RKNN3_FLAG_MEMORY_CACHEABLE);
if (!aux_input_tensor0.mem) {
    printf("Fail to create aux_input_tensor0.mem!\n");
    return -1;
}
aux_input0.input_type = RKNN3_LLM_INPUT_AUX;
aux_input0.aux_input = aux_input_tensor0;
llm_inputs[1] = aux_input0;

// size: n_image * n_image_tokens * vision_embed_dim * sizeof(float16)
aux_input_tensor1.mem = rknn3_create_mem(ctx, 1 * 144 * embedding_dim * sizeof(float16), 0,
RKNN3_FLAG_MEMORY_CACHEABLE);
if (!aux_input_tensor1.mem) {
    printf("Fail to create aux_input_tensor1.mem!\n");
    return -1;
}
aux_input1.input_type = RKNN3_LLM_INPUT_AUX;
aux_input1.aux_input = aux_input_tensor1;
llm_inputs[2] = aux_input1;
```

```

// size: n_image * n_image_tokens * vision_embed_dim * sizeof(float16)
aux_input_tensor2.mem = rknn3_create_mem(ctx, 1 * 144 * embedding_dim * sizeof(float16), 0,
RKNN3_FLAG_MEMORY_CACHEABLE);
if (!aux_input_tensor2.mem) {
printf("Fail to create aux_input_tensor2.mem!\n");
return -1;
}
aux_input2.input_type = RKNN3_LLM_INPUT_AUX;
aux_input2.aux_input = aux_input_tensor2;
llm_inputs[3] = aux_input2;

```

## 4.4.2 Callback 设置

### 4.4.2.1 tokenizer callback

`tokenizer_callback` 用于将输入文本编码为 token 序列，一般由 tokenizer (分词器) 实现。用户需实现该回调函数，将字符串（如 prompt）转为所需 token id 形式。

参数说明：

- `userdata`：用户自定义参数，通常为 tokenizer 实例指针
- `text`：待编码的文本内容
- `text_len`：文本长度
- `tokens`：存储输出 token id 的数组
- `n_tokens_max`：`tokens` 能容纳的最大 token 数，值为 `text_len + 2`

返回值：

- 返回实际编码得到的 token 数
- 若编码失败则返回值  $\leq 0$

示例代码：

```

/**
 * @brief tokenizer_callback 回调，用户实现，将文本编码为 tokens
 *
 * @param userdata      指向 tokenizer 实例的指针
 * @param text          输入文本
 * @param text_len      文本长度
 * @param tokens        输出 token id 数组
 * @param n_tokens_max tokens 最多可容纳数量，值为 text_len + 2
 * @return int          返回实际编码的 token 数，失败时返回 ≤ 0
 */
int tokenizer_callback(void* userdata, const char* text, int32_t text_len, int32_t* tokens, int32_t n_tokens_max)
{
    // 从用户传入的 userdata 获取 tokenizer 实例
    Tokenizer* tokenizer = (Tokenizer*)userdata;

    // 调用 tokenizer 的分词方法进行编码
    int n_tokens = tokenizer->Tokenize(text, text_len, tokens, n_tokens_max);
    if (n_tokens <= 0) {
        printf("tokenizer failed for '%s'\n", text);
        return n_tokens;
    }

    // 返回编码获得的 token 数
    return n_tokens;
}

...

// 设置 tokenizer_callback 回调
callback.tokenizer_callback = tokenizer_callback;
callback.tokenizer_userdata = tokenizer;

```

#### 4.4.2.2 embed callback

`embed_callback` 用于根据 token 序列获取其对应的 embedding（词向量）数据。一般情况下，用户需要实现此回调来检索每个输入 token 的 embedding 并填充到 embedding buffer 中。

参数说明：

- `userdata`：用户自定义参数，通常为 `embedding_info` 结构体指针，包含 embedding 数据信息
- `tokens`：输入的 token id 数组
- `num_tokens`：tokens 的数量
- `embed`：embedding buffer，需按顺序存放每个 token 的 embedding
- `len`：embedding buffer 的总字节长度（应等于 `num_tokens * embedding_dim * sizeof(float16)`）

返回值：

- 成功返回 0，失败返回 < 0

示例代码：

```
/**  
 * @brief embed_callback 回调，用户实现，按 tokens 获取对应的 embedding  
 *  
 * @param userdata 指向 embedding_info 结构体的指针  
 * @param tokens 输入的 token id 数组  
 * @param num_tokens token 数量  
 * @param embed embedding buffer，按顺序存放每个 token 的 embedding  
 * @param len embedding buffer 总字节数，等于 num_tokens * embedding_dim * sizeof(float16)  
 * @return int 成功返回 0，失败返回 < 0  
 */  
int embed_callback(void* userdata, int32_t* tokens, uint64_t num_tokens, void* embed, uint64_t len)  
{  
    struct embedding_info* embed_info = (struct embedding_info*)userdata;  
  
    // 检查 buffer 大小是否匹配  
    if (len != num_tokens * embed_info->embedding_dim * sizeof(float16)) {  
        printf("invalid embed buffer!\n");  
        return -1;  
    }  
  
    // 依次查找每个 token 的 embedding 向量，并 copy 到输出 buffer  
    for (uint64_t n = 0; n < num_tokens; n++) {  
        memcpy((unsigned char*)embed + n * embed_info->embedding_dim * sizeof(float16),  
               embed_info->embedding_data + tokens[n] * embed_info->embedding_dim,  
               embed_info->embedding_dim * sizeof(float16));  
    }  
  
    return 0;  
}  
  
...  
  
// 设置 embed_callback 回调  
callback.embed_callback = embed_callback;  
callback.embed_userdata = &embedding_info;
```

#### 4.4.2.3 result callback

`result callback` 用于在模型推理过程中输出推理结果，通常用于将每步新生成的 token 转换为字符串并输出到终端。用户可以根据需要自定义 token 的展示或处理逻辑。需要注意的是，该回调函数在一次推理中会被多次调用，每次输出一个或多个新生成的 token，并有对应的状态（state）提示用户推理的阶段。

参数说明：

- `userdata`：用户自定义参数，通常为 Tokenizer 对象指针（或其他用于解码 token 的上下文）
- `result`：指向 `RKLLMResult` 的结构体，包含本次调用生成的 token id 数组及数量等信息
- `state`：指示推理当前状态，可为 `RKLLM_RUN_NORMAL`（正常生成）、`RKLLM_RUN_ERROR`（推理失败）、`RKLLM_RUN_FINISH`（推理结束）、`RKLLM_RUN_WAITING`（等待补全 UTF-8 字符）、`RKLLM_RUN_MAX_NEW_TOKEN_REACHED`（生成 token 数量达到上限）、`RKLLM_RUN_STOP`（用户主动终止）

返回值：

- 成功返回 0，失败返回 < 0

示例代码：

```
/**
 * @brief result_callback 回调，在每次生成 token 后被调用
 *
 * @param userdata      通常为 Tokenizer 对象指针
 * @param result        指向 RKLLMResult 结构体，包含当前 step 输出的 token
 * @param state         当前推理状态
 * @return int          成功返回 0，失败返回 < 0
 */
int result_callback(void* userdata, RKLLMResult* result, LLMCallState state)
{
    Tokenizer* tokenizer = (Tokenizer*)userdata;

    if (state == RKLLM_RUN_NORMAL) {
        std::string piece;
        if (result->num_tokens == 1) {
            // 单 token 时直接转换
            piece = tokenizer->TokenToPiece(result->token_ids[0]);
        } else {
            // 多 token 时解码为字符串
            piece = tokenizer->Decode(result->token_ids, result->num_tokens);
        }
        // 输出 token 文本
        printf("%s", piece.c_str());
    } else if (state == RKLLM_RUN_ERROR) {
        // 推理过程中发生错误
        printf("\n\nError occurred during inference\n");
    } else if (state == RKLLM_RUN_FINISH) {
        // 推理任务正常完成
        printf("\n\nFinished\n");
    } else if (state == RKLLM_RUN_WAITING) {
        // 模型在等待更多输入以补全一个完整的 UTF-8 字符
        printf("\n\nWaiting for UTF-8 encoded character\n");
    } else if (state == RKLLM_RUN_MAX_NEW_TOKEN_REACHED) {
        // 达到用户配置的生成最大 token 数
        printf("\n\nMax new token reached\n");
    } else if (state == RKLLM_RUN_STOP) {
        // 用户主动终止
        printf("\n\nStop\n");
    }

    fflush(stdout);
    return 0;
}

...
// 设置 result_callback 回调
callback.result_callback = result_callback;
callback.result_userdata = tokenizer;
```

#### 4.4.2.4 sampling callback

`sampling callback` 用于在生成 token 时自定义采样策略。用户可根据自己的需求，实现 argmax、top-k、top-p (nucleus sampling)、temperature scaling 等多种采样方式。该回调使得模型输出 token 时更具灵活性和可控性。

参数说明：

- `userdata`：用户自定义参数，通常为 `embedding_info` 结构体指针，包含 `embedding` 数据信息
- `logits`：模型当前 step 输出的原始 logits 分数，类型为 `float16` 指针，长度为词表大小（vocab size）
- `logits_name`：当前 logits tensor 名称字符串，方便区分多输出场景

返回值：

- 采样得到的 token id

示例代码（以 argmax 采样为例）：

```
/**
```

```

 * @brief 找到最大概率 (argmax) 对应的 token index
 *
 * @param data    logits 分数指针 (float16类型)
 * @param size    词表长度
 * @return int    最大值对应的 index
 */
int argmax(const float16 *data, int size)
{
    if (size <= 0 || !data)
        return -1;

    int max_id = 0;
    for (int i = 1; i < size; i++)
    {
        if (fp16_to_fp32(data[i]) > fp16_to_fp32(data[max_id]))
        {
            max_id = i;
        }
    }
    return max_id;
}

/**
 * @brief sampling_callback 回调，用户可自定义采样策略
 *
 * @param userdata      用户自定义参数，通常为 embedding_info 结构体指针，包含 embedding 数据信息
 * @param logits         当前 step 的 logits 指针 (float16)
 * @param logits_name   当前 logits 的 tensor 名称
 * @return int           输出选中 token 的 id，失败返回 < 0
 */
int sampling_callback(void *userdata, float16 *logits, char *logits_name)
{
    struct embedding_info *embed_info = (struct embedding_info *)userdata;
    int size = embed_info->vocab_size;
    int max_id = argmax(logits, size);

    return max_id;
}

...

// 设置 sampling_callback 回调
callback.sampling_callback = sampling_callback;
callback.sampling_userdata = &embedding_info;

```

#### 4.4.2.5 output callback

`output callback` 允许用户在每次模型推理输出后，取回模型的输出 tensor（如 logits 或其他输出）。通过自定义 output callback，并在 callback 结构体中进行配置，用户即可灵活获取所需的输出 tensor 数据，实现个性化后处理逻辑。

参数说明：

- `userdata`：用户自定义参数，通常为 `embedding_info` 结构体指针或其他自定义上下文
- `output_tensors`：指向 `rknn3_tensor` 结构体数组，包含本次推理输出的所有 tensor
- `n_output_tensors`：`output_tensors` 数组的元素个数（即输出 tensor 数量）
- `state`：`output_callback` 的状态

返回值：

- 成功返回 0，失败返回 < 0

示例代码：

```

 /**
 * @brief output_callback 回调，在每次模型输出后被调用
 *
 * @param userdata      用户自定义参数，通常为 embedding_info 结构体指针或其他自定义上下文
 * @param output_tensors 指向 rknn3_tensor 结构体数组，包含当前 step 的输出 tensor
 * @param n_output_tensors   输出 tensor 数量
 * @param state          output_callback 的状态
 * @return int           成功返回 0，失败返回 < 0
 */

```

```

/*
int output_callback(void* userdata, rknn3_tensor* output_tensors, uint32_t n_output_tensors,
LLMOutputCallbackState state)
{
    // 此处示例每个输出 tensor 仅打印前10个元素
    printf("output_callback: state = %d\n", state);
    for (int i = 0; i < n_output_tensors; i++) {
        printf("output tensor %d, name: %s\n", i, output_tensors[i].attr->name);
        for (int j = 0; j < 10; j++) {
            printf(" [%d] = %f\n", j, fp16_to_fp32(((float16*)output_tensors[i].mem->virt_addr)[j]));
        }
    }
    return 0;
}

...

// 输出 tensor 相关对象的准备
rknn3_tensor output_tensors[1];
memset(output_tensors, 0, sizeof(output_tensors));
int n_output_tensors      = 1;
int output_tensors_index[1] = {0}; // 指定取回 index=0 的输出 tensor

for (int i = 0; i < n_output_tensors; i++) {
    output_tensors[i].attr = (rknn3_tensor_attr*)malloc(sizeof(rknn3_tensor_attr));
    output_tensors[i].attr->index = output_tensors_index[i];
    ret = rknn3_query(ctx, RKNN3_QUERY_OUTPUT_ATTR, output_tensors[i].attr, sizeof(rknn3_tensor_attr));
    if (ret < 0) {
        printf("rknn_query failed! ret=%d\n", ret);
        return -1;
    }
    output_tensors[i].mem = rknn3_create_mem(ctx, output_tensors[i].attr->aligned_size, output_tensors[i].attr-
>core_id, RKNN3_FLAG_MEMORY_CACHEABLE);
}
}

...

// 设置 output_callback 回调
callback.output_callback = output_callback;
callback.output_userdata = &embedding_info;
callback.output_tensors = output_tensors;
callback.n_output_tensors = n_output_tensors;

...

// 释放输出 tensor 相关对象
for (int i = 0; i < n_output_tensors; i++) {
    if (output_tensors[i].attr) {
        free(output_tensors[i].attr);
    }
    if (output_tensors[i].mem) {
        rknn3_destroy_mem(ctx, output_tensors[i].mem);
    }
}
}

```

#### 4.4.3 Function Calling 设置

Function Calling（函数调用）功能允许用户在推理过程中通过模型输出结果来触发预设的函数工具，从而实现与外部系统的数据交互或动作执行。例如，可以配置天气查询、日程提醒等外部调用。

用户需要先通过调用 `rknn3_session_set_function_tools` 向模型设置一组 Function Calling 工具（以 JSON 格式描述），指定每个工具的函数名称、参数定义等信息。推理时，模型会根据会话模版和用户设置的 Function Calling 工具判断是否需要发起函数调用，并在输出结果中提供函数调用请求。用户只需解析模型输出的调用内容，实现实际函数调用，并将结果反馈给模型（此时需设置 `role = "tool"`，模型会对函数调用结果进行处理，并返回最终的结果），便可轻松实现与外部系统的数据交互和复杂功能扩展。以下为关键步骤代码示例，完整示例请参考：`rknn3_session_test_demo/src/rknn_session_test_function_call.cpp`。

示例代码：

```
// 设置一组 Function Calling 工具
```

```

char *function_tools = R"([{"type": "function", "function": {"name": "get_current_temperature", "description": "Get current temperature at a location.", "parameters": {"type": "object", "properties": {"location": {"type": "string", "description": "The location to get the temperature for, in the format \"City, State, Country\"."}, "unit": {"type": "string", "enum": ["celsius", "fahrenheit"], "description": "The unit to return the temperature in. Defaults to \"celsius\"."}}, "required": ["location"]}}], {"type": "function", "function": {"name": "get_temperature_date", "description": "Get temperature at a location and date.", "parameters": {"type": "object", "properties": {"location": {"type": "string", "description": "The location to get the temperature for, in the format \"City, State, Country\"."}, "date": {"type": "string", "description": "The date to get the temperature for, in the format \"Year-Month-Day\"."}, "unit": {"type": "string", "enum": ["celsius", "fahrenheit"], "description": "The unit to return the temperature in. Defaults to \"celsius\"."}}, "required": ["location", "date"]}}})";

ret = rknn3_session_set_function_tools(session, function_tools);
if (ret != 0) {
    printf("rknn3_session_set_function_tools failed, ret = %d\n", ret);
    return -1;
}

rknn3_llm_input llm_inputs[1];
rknn3_llm_input input;
rknn3_llm_infer_param llm_infer_param;
rknn3_llm_tensor input_tensor;
memset(&input, 0, sizeof(rknn3_llm_input));
memset(&llm_infer_param, 0, sizeof(rknn3_llm_infer_param));
memset(&input_tensor, 0, sizeof(rknn3_llm_tensor));

// 第一次推理，模型生成函数调用请求，输出结果存于 model_answer 中
char *prompt = "what's the temperature in San Francisco now? How about tomorrow? Current Date: 2024-09-30.";
llm_infer_param = {.keep_history = 0, .max_new_tokens = max_new_tokens};
input_tensor = {.name = NULL, .prompt = prompt, .embed = NULL, .tokens = NULL, .n_tokens = 0, .enable_thinking = false};
input.role = "user";
input.input_type = RKNN3_LLM_INPUT_PROMPT;
input.llm_input = input_tensor;
llm_inputs[0] = input;

ret = rknn3_session_run(session, llm_inputs, 1, &llm_infer_param);
if (ret != 0) {
    printf("rknn3_session_run failed, ret = %d\n", ret);
    return -1;
}

...

// 解析模型输出的调用内容
std::vector<std::string> tool_call_list = extract_tool_call_list(model_answer);
// 实现实际函数调用，返回结果存于 tool_prompt 中
std::string tool_prompt = get_tool_call_results_json(tool_call_list);

// 第二次推理，模型会对函数调用结果进行处理，并返回最终的结果
llm_infer_param = {.keep_history = 0, .max_new_tokens = max_new_tokens};
input_tensor = {.name = NULL, .prompt = tool_prompt.c_str(), .embed = NULL, .tokens = NULL, .n_tokens = 0, .enable_thinking = false};
input.role = "tool";
input.input_type = RKNN3_LLM_INPUT_PROMPT;
input.llm_input = input_tensor;
llm_inputs[0] = input;

ret = rknn3_session_run(session, llm_inputs, 1, &llm_infer_param);
if (ret != 0) {
    printf("rknn3_session_run failed, ret = %d\n", ret);
    return -1;
}

```

#### 4.4.4 KVCache 导入与导出

KVCache 导入/导出功能允许用户在模型多轮推理过程中，快速保存某一轮推理后的 KVCache 状态，在后续推理时再进行恢复，适用于长对话场景下的模型中断续算或跨会话保存上下文，满足多场景应用资源复用和推理加速需求。（注意：KVCache 状态和当前 session 创建时使用的模型及运行库高度关联，不同模型、不同版本运行库的 KVCache 状态不能混用。）

接口说明：

- `int rknn3_session_load_kvcache(rknn3_session* session, const char* kvcache_path, int64_t size);`：将当前 session 的 KVCache 保存到文件 `kvcache_path`。
- `int rknn3_session_save_kvcache(rknn3_session* session, char* kvcache_path);`：从文件 `kvcache_path` 加载 KVCache 到当前 session 中。

常见典型使用流程为：第一轮推理完成后用 `rknn3_session_save_kvcache` 导出 KVCache；在第二轮推理开始前用 `rknn3_session_load_kvcache` 导入 KVCache。示例代码：

```

for (int i = 0; i < 2; i++) {
    // 第二轮推理前先从文件导入 KVCache，以衔接或者复用上一轮的上下文
    if (i == 1) {
        ret = rknn3_session_load_kvcache(session, "kvcache.bin", strlen("kvcache.bin"));
        if (ret != RKNN3_SUCCESS) {
            printf("rknn3_session_load_kvcache failed, ret = %d\n", ret);
            return -1;
        }
    }

    // 执行模型推理
    ret = rknn3_session_run(session, llm_inputs, llm_n_inputs, &llm_infer_param);
    if (ret != RKNN3_SUCCESS) {
        printf("rknn3_session_run failed, ret = %d\n", ret);
        return -1;
    }

    // 第一轮推理完成后导出当前 KVCache 到文件
    if (i == 0) {
        ret = rknn3_session_save_kvcache(session, "kvcache.bin");
        if (ret != RKNN3_SUCCESS) {
            printf("rknn3_session_save_kvcache failed, ret = %d\n", ret);
            return -1;
        }
    }
}

```

## 4.5 KVCache管理

### 4.5.1 KV Cache 量化

#### KV Cache 量化介绍

在大规模语言模型推理过程中，KV Cache（Key/Value Cache）用于存储历史的注意力键值，以避免重复计算，从而提高推理速度。随着序列长度增长，KV Cache 的内存占用会快速增加。为了减少 KV Cache 的存储带宽与内存访问开销，可以采用量化方式将其从 FP16/FP32 转换为 INT8 或更低位宽表示。但由于 KV Cache 数值分布随时间逐 token 动态变化，如果对整段 KV 使用统一的量化参数，会导致量化误差累积从而影响推理精度。因此通常采用分组量化（Group Quantization）来降低精度损失。

#### 分组量化定义

KV Cache 分组量化（Group Quant）将 KV Cache 中每个通道或token按分组数（group\_size）进行分块（group），并为每个分组独立计算量化比例因子（scale）与零点（zero point, zp），从而减少量化误差。分组量化分为 Channel 分组量化 和 Token 分组量化两种方式，即沿着不同的方向做分组量化，两种量化方式如下图所示：

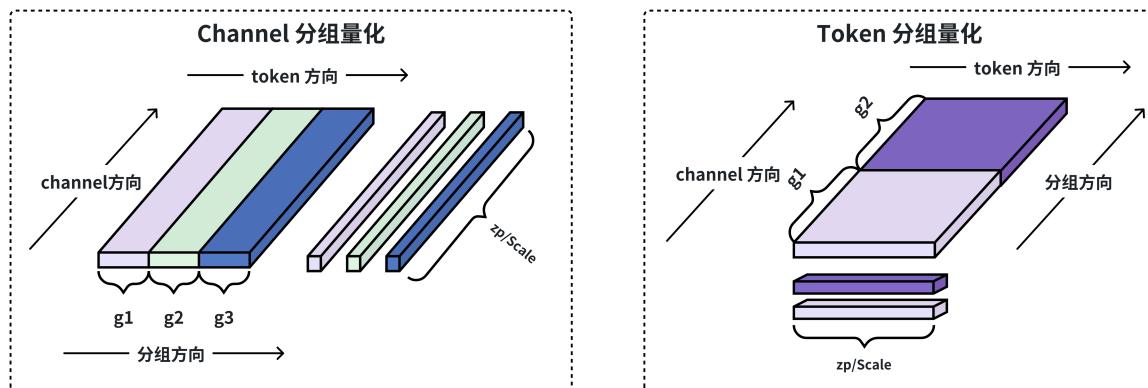


图4-6 分组量化示意图

Channel 分组量化：沿着 token 方向进行分组，对每个分组内的 channel 上的每个点计算一组 zp 和 scale。

Token 分组量化：沿着 channel 方向进行分组，对每个 token 上的每个 channel 分组计算一组 zp 和 scale。

量化计算采用了线性非对称计算，计算原理详见《5.量化说明》，区别在于量化零点采用了 Float8 或者 Float16 类型。

在 RKNN3 中，Key 采用了 Channel 分组量化方式，Value 采用了 Token 分组量化方式。

#### 分组量化支持情况

目前支持的分组量化类型与具体规格如下表所示：

量化类型	分组数	量化方式	KV Cache 量化数据类型	zp 数据类型	scale 数据类型	反量化数据类型
Int4_to_F8	32	线性非对称	INT4	Float8(E4M3)	Float16	Float8(E4M3)
Int4_to_F16	16	线性非对称	INT4	Float16	Float16	Float16
Int8_to_F16	16	线性非对称	INT8	Float16	Float16	Float16

量化类型以 src\_to\_dst 表示，其中 src 表示量化后的 KV Cache 数据类型，dst 表示反量化后参与运算的数据类型。

## 4.6 Cacheable 内存一致性

在 RKNN3 SDK 中，应用程序（运行在 SoC 上）与 NPU（运行在 RK182X 协处理器上）可能会同时访问同一块带有 Cache 属性的共享内存。由于 SoC 和 RK182X 协处理器拥有独立的 Cache 与内存管理机制，如果数据未正确进行 Cache 同步，会导致 SoC 与协处理器读取到的数据不一致，从而引发模型推理错误或数据异常。因此，必须在合适的时机调用 `rknn3_mem_sync` 对 cacheable 内存进行同步，保证两侧访问 DDR 数据的一致性。

本章节将介绍 cacheable 内存同步的方向以及如何使用 `rknn3_mem_sync` 完成同步操作。

### 4.6.1 Cacheable 内存同步方向

当 SoC 与 RK182X 协处理器访问同一块 cacheable 内存时，正确的同步方向取决于“哪一侧更新数据、哪一侧读取数据”。典型情况如下：

- SoC 写数据 → RK182X 读取数据

当应用程序在 SoC 侧写入数据（例如模型输入数据等），写操作通常会先写入 SoC 本地 Cache，而不是立即写入到设备端 DDR。

如果 RK182X 协处理器随后直接从 DDR 读取该内存，则可能读取旧数据。因此需要在 RK182X 访问该内存前执行同步，使 SoC Cache 中的数据刷新到设备端 DDR。

此时同步方向为：**RKNN3\_MEMORY\_SYNC\_TO\_DEVICE**，表示数据从 SoC 刷新至设备（RK182X）侧可见的 DDR。

- RK182X 写数据 → SoC 读取数据

当 NPU 或协处理器在推理过程中将输出结果写回共享内存时，这些写操作会停留在 RK182X 的 Cache 中，而不是实时写回 SoC 端 DDR。

若 SoC 随后读取该共享内存，可能会读到旧数据，因此需要在 SoC 访问前进行同步，将设备侧 Cache 刷新到 SoC 侧 DDR。

此时同步方向为：**RKNN3\_MEMORY\_SYNC\_FROM\_DEVICE**，表示数据从设备（RK182X）同步到 SoC 可见的 DDR。

### 4.6.2 同步 Cacheable 内存

明确同步方向后，即可调用 `rknn3_mem_sync` 对 cacheable 内存进行同步。接口定义请参考 [3.4.1.7 内存同步](#) 章节。

其中 `rknn3_mem_sync_mode` 枚举定义：

```
/**  
 * @brief Memory synchronization modes for rknn3_mem_sync function  
 */  
typedef enum _rknn3_mem_sync_mode  
{  
    /* the mode used for consistency of device access after CPU accesses data. */  
    RKNN3_MEMORY_SYNC_TO_DEVICE = 0x1,  
  
    /* the mode used for consistency of CPU access after device accesses data. */  
    RKNN3_MEMORY_SYNC_FROM_DEVICE = 0x2,  
  
    /* the mode used for consistency of data access between device and CPU in both directions. */  
    RKNN3_MEMORY_SYNC_BIDIRECTIONAL =  
        RKNN3_MEMORY_SYNC_TO_DEVICE | RKNN3_MEMORY_SYNC_FROM_DEVICE,  
} rknn3_mem_sync_mode;
```

各枚举意义：

- `RKNN3_MEMORY_SYNC_TO_DEVICE`：适用于 SoC 写数据 → RK182X 读取的情况。
- `RKNN3_MEMORY_SYNC_FROM_DEVICE`：适用于 RK182X 写结果 → SoC 读取的情况。

- `RKNN3_MEMORY_SYNC_BIDIRECTIONAL`: 同时包含 `TO_DEVICE` 与 `FROM_DEVICE`, 适用于方向不确定或内存双向使用的情况。

#### 使用建议

1. 当确定数据流向时, 推荐使用单方向同步, 可减少多余 Cache 刷新操作, 提高推理性能。
2. 在从 SoC 向设备发送输入数据时, 调用 `rknn3_mem_sync(..., RKNN3_MEMORY_SYNC_TO_DEVICE)`。
3. 在从设备读取输出数据时, 调用 `rknn3_mem_sync(..., RKNN3_MEMORY_SYNC_FROM_DEVICE)`。

## 4.7 LLM 模型适配

本章节阐述了如何加载 LLM 模型以及 LLM 模型转换时的特殊配置。本章节将详细解释其中的相关概念, 以及用户应该如何将自己的 LLM 模型调整到符合 RKNPU3 工具链的相关要求。

### 4.7.1 ONNX 模型适配

RKNPU3 工具链针对 LLM 模型的推理需求, 从运行效率、内存节省方面出发, 提供了以下优化配置。

功能	优化行为	收益
Attention 配置	配置 KV Cache 缓存机制、划窗推理机制	简化模型调用流程, 提升推理性能、减少 KV Cache 内存占用
LLM head 配置	配置 LLM head 的存储位置, 量化精度	提升推理精度
LLM vocab 配置	配置 LLM vocab 的存储位置, tie embedding 优化	减少 LLM head 字典的内存占用
keep one logit 配置	配置是否对输出的指定轴进行取单值操作	减少 Internal 内存占用、减少无效计算量

除“Attention 配置”为强制开启项, 其他选项用户可根据实际需求在 `rknn.config` 接口中进行配置, 并使用 `rknn.load_llm` 接口加载模型以开启这些优化。

当前 `rknn.load_llm` 接口加载 ONNX 模型需满足以下基本要求:

- **模型支持动态 Shape:** ONNX 模型需原生支持动态 Shape 推理。若模型可通过 ONNX Runtime 加载并成功执行动态 Shape 推理, 则认为满足此条件。如模型不支持动态 Shape, 后续流程可能出现异常。
- **注意力结构规范:** 模型中所有注意力 (Attention) 结构需符合经典定义, 并需提供具有注意力掩码 (`attention_mask`) 语义的输入张量。经典注意力结构定义为:  $\text{Attention}(Q, K, V, Mask) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}} + Mask\right)V$

如图4-7, 是经典 LLM 模型的简化推理流程, 接下来从各个环节出发介绍各优化项。

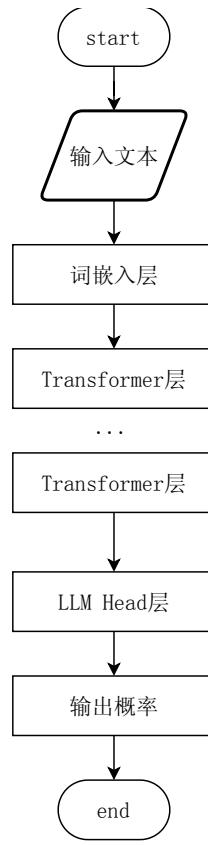


图4-7 LLM模型简化流程

#### LLM Vocab

词嵌入层是大型语言模型的第一个处理层，其功能是将输入的词元（Token）映射为对应的数值向量表示，以便为后续神经网络计算提供结构化输入。该层本质上不执行算术运算，其行为是基于预定义的词表（vocabulary）执行查表与内存拷贝。

在当前的 RKNPU3 工具链中，默认将整个词表驻留于主机（Host）内存。在推理过程中，当计算设备（Device）需要查询词表时，工具链会通过主机与设备间的通信接口，动态传输所需的词表子集，从而降低计算设备上的内存占用。后续版本的 RKNPU3 工具链将进一步支持以下优化策略：

- 将完整的词表预置于计算设备内存，以消除查询时的通信延迟。适用于占用内存较小的 LLM 模型。
- 支持嵌入层权重绑定（Tie Embedding），即允许语言模型头（LLM Head）层的权重与词嵌入层共享。该特性可将两个组件的权重融合为单一的存储对象，以进一步优化内存使用效率。

---

#### Attention OP - 算子定义与匹配

RKNPU3 工具链通过识别并匹配 ONNX 计算图中符合经典自注意力（Attention）结构的算子组合，将其转换为专用的 exAttention 算子。此转换需满足以下条件：

- 注意力计算必须符合 Transformer 架构中的经典注意力结构。
- 计算过程中产生的中间张量不得被计算图中的其他算子所使用。
- 模型的输入输出定义中，不应显式包含与该注意力结构对应的键值缓存（Key-Value Cache）输入或输出端口。

如需外部管理 KV 缓存，则应通过 `rknn.load_onnx` 接口加载模型。在此模式下，工具链将不会执行 Transformer 相关的图优化转换。

- 支持混合注意力结构，即允许模型内同时存在采用外部管理 KV 缓存的注意力模块，以及使用 RKNPU3 内部 KV 缓存管理机制的注意力模块，此时仍使用 `rknn.load_llm` 接口加载模型。

---

#### Attention OP - RoPE 融合支持

exAttention 算子支持将旋转位置编码（Rotary Positional Embedding, RoPE）的计算过程融合至算子内部。融合后，算子可在发生KV cache 循环复写的注意力计算场景中，对 RoPE 的计算结果执行数值修正，提高推理精度。启用 RoPE 融合需满足以下条件：

- RoPE 的位置信息应仅由 `position_id` 输入决定。
- 支持以下两种 RoPE 实现方式：
  1. 位置编码在推理时由 `position_id` 实时计算生成，不依赖于其他预存输入。

2. 模型已预算算并存储了 RoPE 位置编码矩阵。该矩阵的维度需不小于 `rknn.config` 中为该注意力结构所配置的 `max_position_embeddings` 参数。
- 位置编码矫正仅对普通 rope 有效。开启 mrope 功能且触发 KV cache 循环覆写时，目前暂无位置编码矫正效果。

#### Attention OP - 多注意力结构配置

若模型中存在多种参数或配置不同的注意力结构（例如，RoPE 配置、最大推理长度、KV 缓存管理方式、是否启用滑动窗口等存在差异），需遵循以下配置规则：

- 必须在 `rknn.config` 中为每种独立的注意力结构分别进行定义和配置。
- 每种注意力结构必须具备独立的 `attention_mask` 输入端口，且该输入在模型中必须对应不同的张量名称，不同结构的 `attention_mask` 端口不得共享。

例如，即使模型中两种结构的掩码值完全相同，也需在模型输入中提供两个独立的、不同名称的 `attention_mask` 张量。
- 每种注意力结构可独立选择是否配置 RoPE。若配置 RoPE，其位置信息应通过独立的输入张量（通常为 `position_id`）决定。

若多个注意力结构均需启用 RoPE，则模型必须提供对应数量的、独立的 `position_id` 输入端口，不得在不同结构间复用同一 `position_id` 输入。

#### LLM Head

LLM Head 是大型语言模型（LLM）架构中的关键组件，通常指模型的最后一层。其核心功能是将最终的隐藏状态（hidden states）通过线性变换映射为词汇表规模上的概率分布，从而预测下一个待生成的词元。在实现上，该层通常表现为一个矩阵乘法（MatMul）层，其权重矩阵的维度与词汇表大小及模型隐藏层维度直接相关。

以 Qwen3-4B 模型为例，其 LLM Head 在 float16 精度下约占 740MB 存储空间。当前工具链支持对 LLM Head 层单独配置量化类型，用户可根据实际任务在推理精度与内存占用之间进行权衡。此外，也支持将 LLM Head 指定运行在 rk3588 或 rk3576 等协处理器设备上，从而在可接受的一定推理速度损耗下，有效减少主计算设备上的内存占用。

#### Keep\_one\_logit

在典型的因果语言模型（如 Qwen2ForCausalLM、LlamaForCausalLM）推理过程中，Prefill 阶段（即首次输入多个词元进行前向计算）的后处理仅需对序列最后一个位置的 logit 进行采样，即可得到下一个预测词元。其余位置的 logit 在后续推理中不会被使用，但依然参与计算并占用内存资源。

为此，可在模型架构中（在进入 LLM Head 层计算之前）插入一个 Gather 算子，使其仅提取并保留序列最后一个位置的 logit，从而显著减少 LLM Head 层的计算量及内存占用。

说明：目前此优化功能仅支持在 LLM Head 层的输入来自 MatMul 或 Gemm 算子时启用，其他情况暂不兼容。

#### 4.7.2 tokenizer说明

tokenizer 的作用是将文本转成 token id、将 token id 转成文本以及解析词表信息。在板端推理时需要用到模型对应的 `tokenizer.gguf` 文件以及 `libtokenizer.a` 库。

- `tokenizer.gguf`: `tokenizer.gguf` 文件在 LLM 模型转换时，通过 `export_tokenizer` 接口导出，其中包含了词表信息，即对应 LLM 模型的 `tokenizer.json` 信息。
- `libtokenizer.a`: `libtokenizer.a` 库负责将 `tokenizer.gguf` 文件中的词表信息进行解析，并提供 `文本转 token id` 和 `token id 转文本` 等接口。rknn3-model-zoo 使用的是 <https://github.com/ggml-org/llama.cpp> 的 tokenizer，在 rknn3-model-zoo 的路径为 `rknn3_model_zoo/tokenizer`。如果有调整 tokenizer 库的需要，可以修改 `rknn3_model_zoo/tokenizer/src` 中的代码，并重新编译出新的 `libtokenizer.a`。

## 4.8 自定义CPU后处理算子

RKNN3 提供了自定义算子扩展机制，允许用户实现自定义的后处理算子（Postprocess Plugin）在 RK182x 协处理器的 CPU 核心上执行。该机制主要用于将模型的后处理计算从主控 SoC 迁移到 RK182x 协处理器端，从而实现以下优势：

- **减少数据传输开销**：避免将大量原始特征图数据从协处理器传回主控端
- **降低主控端负载**：将后处理计算任务卸载到协处理器端的 CPU 核心
- **简化应用开发**：统一的插件接口，便于快速集成

典型的应用场景包括目标检测模型（YOLOv5/v6/v8）的边界框解码、NMS（非极大值抑制）等后处理操作。

#### 4.8.1 插件接口定义

自定义后处理算子需要实现 `rknn3_custom_op` 结构体定义的接口。该结构体在 `rknn3_api.h` 中定义如下：

```
typedef struct _rknn3_custom_op
{
    const char *op_type;           // 算子类型名称
    rknn3_op_plugin_type plugin_type; // 插件类型
    rknn3_op_target_type target;   // 执行后端 (CPU/NPU)
    uint32_t version;             // 版本号
    const char *author;            // 作者信息
    const char *description;       // 描述信息

    // 生命周期回调函数
    int (*init)(rknn3_custom_op_context *op_ctx);      // [可选] 初始化
    int (*prepare)(rknn3_custom_op_context *op_ctx,
                   rknn3_tensor *inputs, uint32_t n_inputs,
                   rknn3_tensor *outputs, uint32_t n_outputs); // [可选] 准备
    int (*compute)(rknn3_custom_op_context *op_ctx,      // [必需] 计算
                   rknn3_tensor *inputs, uint32_t n_inputs,
                   rknn3_tensor *outputs, uint32_t n_outputs);
    int (*deinit)(rknn3_custom_op_context *op_ctx);     // [可选] 反初始化

    // 后处理插件专用接口
    int (*get_output_num)(rknn3_custom_op_context *op_ctx); // [可选] 获取输出数量
    int (*get_attrs)(rknn3_custom_op_context *op_ctx,        // [可选] 获取输入输出属性
                    rknn3_tensor_attr *inputAttrs, uint32_t n_inputs,
                    rknn3_tensor_attr *outputAttrs, uint32_t n_outputs);
} rknn3_custom_op;
```

关键数据结构说明：

- `rknn3_custom_op_context`: 算子上下文，包含 RKNN3 context 句柄、私有数据指针等
- `rknn3_tensor`: 张量结构，包含内存信息 (`rknn3_tensor_mem`) 和属性信息 (`rknn3_tensor_attr`)
- `rknn3_tensor_attr`: 张量属性，包括 shape、dtype、layout、量化参数等

插件类型枚举：

```
typedef enum _rknn3_op_plugin_type
{
    RKNN3_OP_PLUGIN_TYPE_POSTPROCESS = 0, // 后处理插件
    RKNN3_OP_PLUGIN_TYPE_CUSTOM_OP = 1,    // 自定义算子 (暂不支持)
} rknn3_op_plugin_type;

typedef enum _rknn3_op_target_type
{
    RKNN3_OP_TARGET_TYPE_CPU = 1, // CPU 后端
} rknn3_op_target_type;
```

#### 4.8.2 插件实现步骤

自定义算子插件开发流程如图4-8所示：

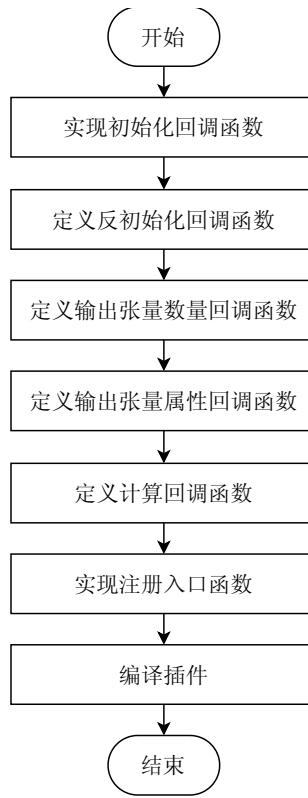


图4-8 自定义算子插件开发流程

插件实现需要完成以下关键函数：

#### 步骤 1：定义算子结构体

```

static rknn3_custom_op yolo_postprocess_op = {
    .op_type = "YoloPostprocess",
    .plugin_type = RKNN3_OP_PLUGIN_TYPE_POSTPROCESS,
    .target = RKNN3_OP_TARGET_TYPE_CPU,
    .version = 1,
    .author = "Rockchip",
    .description = "YOLO Postprocess Plugin",

    .init = yolo_postprocess_plugin_init,           // 初始化
    .compute = yolo_postprocess_plugin_compute, // 计算 (必需)
    .deinit = yolo_postprocess_plugin_deinit, // 反初始化
    .get_output_num = yolo_postprocess_plugin_get_output_num,
    .getAttrs = yolo_postprocess_plugin_getAttrs,
};

```

#### 步骤 2：实现关键回调函数

```

// 初始化函数：分配私有数据
static int yolo_postprocess_plugin_init(rknn3_custom_op_context *op_ctx)
{
    // 分配并初始化私有数据
    rknn3_yolo_postprocess_context *pp_ctx = malloc(sizeof(rknn3_yolo_postprocess_context));
    memset(pp_ctx, 0, sizeof(rknn3_yolo_postprocess_context));
    pp_ctx->conf_threshold = 0.25;
    pp_ctx->nms_threshold = 0.45;

    // 查询模型输入尺寸并保存
    // ... 通过 rknn3_query 获取模型信息

    op_ctx->priv_data = pp_ctx;
    return 0;
}

// 计算函数：执行后处理逻辑
static int yolo_postprocess_plugin_compute(rknn3_custom_op_context *op_ctx,

```

```

        rknn3_tensor *inputs, uint32_t n_inputs,
        rknn3_tensor *outputs, uint32_t n_outputs)
{
    rknn3_yolo_postprocess_context *pp_ctx = op_ctx->priv_data;

    // 访问输入: inputs[i].mem->virt_addr, inputs[i].attr
    // 写入输出: outputs[j].mem->virt_addr
    int result_count = post_process(pp_ctx, inputs, n_inputs, outputs, n_outputs);

    return (result_count >= 0) ? 0 : -1;
}

// 获取输出属性: 定义输出张量的 shape、dtype 等
static int yolo_postprocess_plugin_get_attrs(rknn3_custom_op_context *op_ctx,
                                              rknn3_tensor_attr *inputAttrs, uint32_t n_inputs,
                                              rknn3_tensor_attr *outputAttrs, uint32_t n_outputs)
{
    // ...
    outputAttrs[0].n_dims = 3;
    outputAttrs[0].shape[0] = inputAttrs[0].shape[0]; // batch
    outputAttrs[0].shape[1] = 256; // 最大检测数
    outputAttrs[0].shape[2] = 6; // (score, class_id, x1, y1, x2, y2)
    outputAttrs[0].dtype = RKNN3_TENSOR_FLOAT32;
    outputAttrs[0].aligned_size = outputAttrs[0].shape[0] * 256 * 6 * sizeof(float);
    return 0;
}

// 反初始化函数: 释放资源
static int yolo_postprocess_plugin_deinit(rknn3_custom_op_context *op_ctx)
{
    if (op_ctx->priv_data) {
        free(op_ctx->priv_data);
        op_ctx->priv_data = NULL;
    }
    return 0;
}

```

### 步骤 3：实现注册入口函数

```

static rknn3_custom_op* registered_ops[] = { &yolo_postprocess_op, NULL };

// 插件入口函数（函数名固定）
rknn3_custom_op* rknn3_register_custom_ops_plugin(int op_index)
{
    if (op_index < 0 || registered_ops[op_index] == NULL) {
        return NULL;
    }
    return registered_ops[op_index];
}

```

### 4.8.3 编译插件

编译要求：

- **语言：**必须使用纯 C 语言（不支持 C++ 类、模板、命名空间等）
- **编译器下载地址：** RK182X-GCC <https://console.zbox.filez.com/l/103Dro> 提取码: rknn

编译脚本示例：

```

#!/bin/bash

RKNUPU3_INCLUDE="../../../../3rdparty/rknpu3/include/"
CC="riscv64-unknown-elf-gcc"
STRIP="riscv64-unknown-elf-strip"

# RK182x CPU 核心专用编译选项
CFLAGS="-mcpu=c908v -mrvv-v0p10-compatible -march=rv64gcv -mabi=lp64d \
-O2 -g2 -mcmodel=medany -fpic -fno-plt \
-D_POSIX_C_SOURCE=1 -I${RKNUPU3_INCLUDE}"

LD_FLAGS="-mcpu=c908v -O2 -g2 -Wl,-r,-z,max-page-size=1024 \

```

```

-fpic -nostartfiles -nostdlib -static-libgcc -e 0"

# 编译
${CC} -o rknn3_custom_op.o -c rknn3_custom_op.c ${CFLAGS}
${CC} -o yolov5_postprocess.o -c yolov5_postprocess.c ${CFLAGS}
${CC} -o libpostprocess_yolov5_rk182x.so ${LD_FLAGS} *.o
${STRIP} --strip-unneeded -R .hash libpostprocess_yolov5_rk182x.so

```

#### 4.8.4 插件加载与使用

自定义算子插件使用流程如图4-9所示：

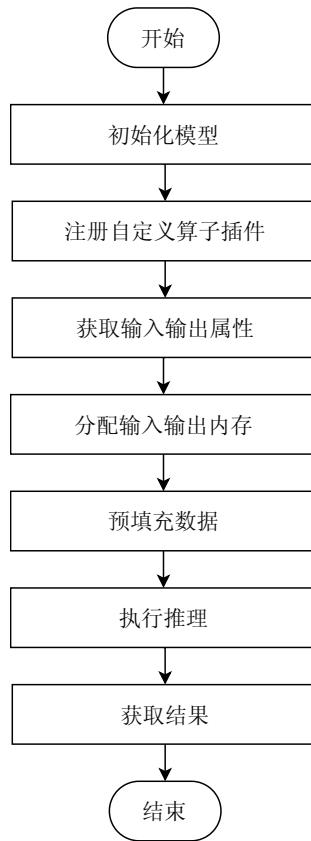


图4-9 自定义算子插件使用流程

API 接口：

```

int rknn3_register_custom_ops_plugins(rknn3_context ctx,
                                       const char* plugin_path,
                                       int64_t size);

```

使用流程：

```

// 1. 初始化并加载模型
rknn3_init(&ctx, NULL);
rknn3_load_model_from_path(ctx, "yolov5.rknn", "yolov5.weight");
rknn3_model_init(ctx, &config);

// 2. 加载后处理插件（必须在 model_init 之后）
const char* plugin_path = "./libpostprocess_yolov5_rk182x.so";
ret = rknn3_register_custom_ops_plugins(ctx, plugin_path, strlen(plugin_path));
if (ret != RKNN3_SUCCESS) {
    printf("Plugin load failed! ret=%d\n", ret);
    return -1;
}

// 3. 查询插件输出信息
rknn3_input_output_num io_num;
rknn3_query(ctx, RKNN3_QUERY_POSTPROCESS_IN_OUT_NUM, &io_num, sizeof(io_num));

rknn3_tensor_attr output_attrs[io_num.n_output];
for (int i = 0; i < io_num.n_output; i++) {

```

```

        output_attrs[i].index = i;
        rknn3_query(ctx, RKNN3_QUERY_POSTPROCESS_OUTPUT_ATTR,
                     &output_attrs[i], sizeof(rknn3_tensor_attr));
    }

// 4. 执行推理（框架自动调用插件）
rknn3_run(ctx, inputs, n_inputs, outputs, n_outputs);

// 5. 读取插件输出结果（已包含后处理后的检测框）
float* result = (float*)outputs[0].mem->virt_addr;
// result 格式: [N, 256, 6] (score, class_id, x1, y1, x2, y2)

```

关键点：

- 插件必须在 `rknn3_model_init` 之后、`rknn3_run` 之前加载
- 使用 `RKNN3_QUERY_POSTPROCESS_*` 查询插件的输入输出属性
- `rknn3_run` 自动调用插件，无需额外操作

#### 4.8.5 YOLOv5 插件示例

在 `rknn3-model-zoo/examples/yolov5/cpp/libpostprocess_rk182x/` 目录提供了完整的 YOLO 后处理插件实现。

目录结构：

```

libpostprocess_rk182x/
├── rknn3_custom_op.c      # 插件主入口
├── yolov5_postprocess.c  # YOLOv5 后处理 (Anchor-based)
├── yolov8_postprocess.c  # YOLOv8/v6 后处理 (Anchor-free)
├── postprocess.h          # 头文件
├── build.sh               # 编译脚本
└── prebuilt/              # 预编译库
    ├── libpostprocess_yolov5_rk182x.so
    ├── libpostprocess_yolov6_rk182x.so
    └── libpostprocess_yolov8_rk182x.so

```

插件配置：

```

#define MAX_OBJ_NUM     256 // 最大检测数
#define OBJ_CLASS_NUM   80  // 类别数 (COCO)
#define NMS_THRESH       0.45 // NMS 阈值
#define BOX_THRESH       0.25 // 置信度阈值

```

编译方法：

```

cd rknn3-model-zoo/examples/yolov5/cpp/libpostprocess_rk182x
./build.sh yolov5      # 编译 YOLOv5 插件
./build.sh yolov8      # 编译 YOLOv8/v6 插件

```

输出格式：

- Shape: `[N, 256, 6]`
- 每个检测框: `(score, class_id, x1, y1, x2, y2)`

#### 4.8.6 注意事项与限制

语言要求：

- ⚠ 必须使用纯 C 语言，不支持 C++ 特性（类、模板、命名空间等）
- 编译使用 `gcc`（不是 `g++`）

编译工具链：

- 必须使用 `riscv64-unknown-elf-gcc`（针对 RK182x CPU 核心）
- 编译选项参考 `examples/yolov5/cpp/libpostprocess_rk182x/build.sh`

接口要求：

- 必须实现 `rknn3_register_custom_ops_plugin(int op_index)` 入口函数
- `getAttrs` 及 `compute`、`getOutputNum` 函数必需，`init/deinit` / 根据需要实现

内存管理：

- 输入张量由框架管理（只读）
- 输出张量由框架根据 `get_attrs` 分配
- `init` 中分配的资源必须在 `deinit` 中释放

常见错误：

错误	原因	解决方法
插件加载失败	路径错误或文件不存在	检查 <code>.so</code> 文件路径和权限
推理失败	<code>get_attrs</code> 输出属性错误	检查 <code>shape</code> 、 <code>dtype</code> 、 <code>aligned_size</code>
运行时崩溃	内存越界或空指针	添加边界检查和参数校验
编译错误	使用了 C++ 特性	确保纯 C 代码，使用 <code>gcc</code> 编译

参考资料：

- 完整示例：`rknn3-model-zoo/examples/yolov5/cpp/libpostprocess_rk182x/`

## 5. 量化说明

### 5.1 量化介绍

#### 5.1.1 量化定义

模型量化是指将深度学习模型中的浮点参数和操作转换为定点表示，如FP32转换为INT8等。量化能够降低内存占用，实现模型压缩和推理加速，但会造成一定程度的精度损失。

#### 5.1.2 量化计算原理

以线性非对称量化为例，浮点数量化为有符号定点数的计算原理如下：

$$x_{int} = \text{clamp}\left(\lfloor \frac{x}{s} \rfloor + z; -2^{b-1}, 2^{b-1} - 1\right) \quad (5-1)$$

其中  $x$  为浮点数， $x_{int}$  为量化定点数， $\lfloor \cdot \rfloor$  为四舍五入运算， $s$  为量化比例因子， $z$  为量化零点， $b$  为量化位宽，如INT8数据类型中  $b$  为8； $\text{clamp}$  为截断运算，具体定义如下：

$$\text{clamp}(x; a, c) = \begin{cases} a, & x < a, \\ x, & a \leq x \leq c, \\ c, & x > c, \end{cases} \quad (5-2)$$

从定点数转换为浮点数称为反量化过程，具体定义如下：

$$x \approx \hat{x} = s(x_{int} - z) \quad (5-3)$$

设量化范围为  $(q_{\min})$ ，截断范围为  $(c_{\min})$ ，量化参数  $s$  和  $z$  的计算公式如下：

$$s = \frac{q_{\max} - q_{\min}}{c_{\max} - c_{\min}} = \frac{q_{\max} - q_{\min}}{2^b - 1} \quad (5-4)$$

$$z = c_{\max} - \lfloor \frac{q_{\max}}{s} \rfloor \text{ 或 } z = c_{\min} - \lfloor \frac{q_{\min}}{s} \rfloor \quad (5-5)$$

其中截断范围是根据量化的数据类型决定，例如INT8的截断范围为(-128, 127)；量化范围根据不同的量化算法确定。

#### 5.1.3 量化误差

量化会造成模型一定程度的精度丢失。根据公式(5-1)可知，量化误差来源于舍入误差和截断误差，即  $\lfloor \cdot \rfloor$  和  $\text{clamp}$  运算。四舍五入的计算方式会产生舍入误差，误差范围为  $(-\frac{1}{2}s, \frac{1}{2}s)$ 。当浮点数  $x$  过大，比例因子  $s$  过小时，容易导致量化定点数超出截断范围，产生截断误差。理论上，比例因子  $s$  的增大可以减小截断误差，但会造成舍入误差的增大。因此为了权衡两种误差，需要设计合适的比例因子和零点，来减小量化误差。

#### 5.1.4 线性对称量化和线性非对称量化

线性量化中定点数之间的间隔是均匀的，例如INT8线性量化将量化范围均匀等分为256个数。线性对称量化中零点是根据量化数据类型确定并且零点  $z$  位于量化定点数范围上的中心对称点，例如INT8中零点为0。线性非对称量化中零点根据公式(5-5)计算确定并且零点  $z$  一般不在量化定点数范围上的中心对称点。

对称量化是非对称量化的简化版本，理论上非对称量化能够更好的处理数据分布不均匀的情况，因此实践中大多采用非对称量化方案。

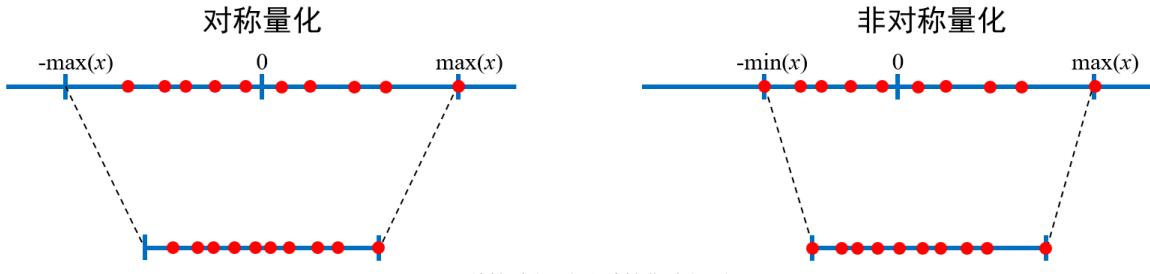


图5-1 线性对称量化和线性非对称量化

### 5.1.5 Per-Layer、Per-Channel和Group量化

Per-Layer量化将网络层的所有通道作为一个整体进行量化，所有通道共享相同的量化参数。Per-Channel量化将网络层的各个输出通道独立进行量化，每个通道有自己的量化参数。Group量化是在Per-Channel量化的基础上，对输入通道进行分组，每个分组独立进行量化。

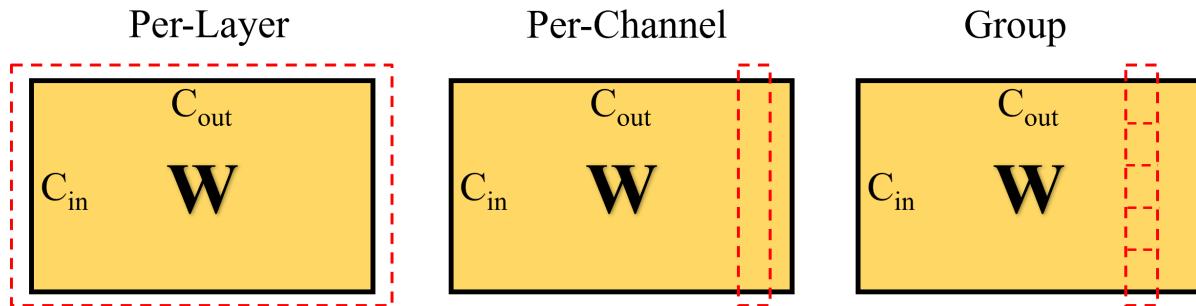


图5-2 Per-Layer量化、Per-Channel量化和Group量化

注意：RKNN3-Toolkit中Group量化和Per-Channel量化中只针对权重进行Per-Channel量化，激活值和中间值仍为Per-Layer量化且Group量化仅支持在a16的条件下。

### 5.1.6 量化算法

量化比例因子 $s$ 和零点 $z$ 是影响量化误差的关键参数，而量化范围的求解对量化参数起到决定性作用。以下介绍三种关于量化范围求解的算法，Normal, KL-Divergence和MMSE。

Normal量化算法是通过计算浮点数中的最大值和最小值直接确定量化范围的最大值和最小值。从[5.1.2 量化计算原理](#)，Normal量化算法不会产生截断误差，但对异常值很敏感，因为大异常值可能会导致舍入误差过大。

$$q_{\min} = \min \mathbf{V} \quad (5-6)$$

$$q_{\max} = \max \mathbf{V} \quad (5-7)$$

其中 $V$ 为浮点数Tensor。

KL-Divergence量化算法计算浮点数和定点数的分布，通过调整不同的阈值来更新浮点数和定点数的分布，并根据KL散度最小化两个分布的相似性来确定量化范围的最大值和最小值。KL-Divergence量化算法通过最小化浮点数和定点数之间的分布差异，能够更好地适应非均匀的数据分布并缓解少数异常值的影响。

$$\arg \min_{q_{\min}, q_{\max}} H(\Psi(\mathbf{V}), \Psi(\mathbf{V}_{int})) \quad (5-8)$$

其中 $H(\cdot, \cdot)$ 为KL散度计算公式， $\Psi(\cdot)$ 为分布函数，将对应数据计算为离散分布， $V_{int}$ 为量化定点数Tensor。

MMSE量化算法通过最小化浮点数与量化反量化后浮点数的均方误差损失，确定量化范围的最大值和最小值，在一定程度上缓解大异常值带来的量化精度丢失问题。由于MMSE量化算法的具体实现是采用暴力迭代搜索近似解，速度较慢，内存开销较大，但通常会比Normal量化算法具有更高的量化精度。

$$\arg \min_{q_{\min}, q_{\max}} \|\mathbf{V} - \widehat{\mathbf{V}}(q_{\min}, q_{\max})\|_F^2 \quad (5-9)$$

其中 $\widehat{\mathbf{V}}(q_{\min}, q_{\max})$ 为 $\mathbf{V}$ 的量化、反量化形式， $\|\cdot\|_F$ 为F范数。

## 5.2 量化配置

### 5.2.1 量化数据类型

RKNN3-Toolkit支持的量化数据类型w8a8和w4a16，其中w8、w4代表权重量化为int8、int4类型；a8、a16代表激活量化为int8、float16类型。

## 5.2.2 量化算法建议

Normal量化算法运行速度快，适用于一般场景。

KL-Divergence量化算法运行速度略慢于Normal量化算法，对于存在非均匀分布的部分模型能够改善量化精度，部分场景下能够缓解少数异常值造成的量化精度丢失问题。

MMSE量化算法运行速度较慢，内存消耗大，相比KL\_Divergence量化算法能够更好的缓解异常值造成的量化精度丢失问题。对于量化友好的模型可尝试使用MMSE量化算法来提高量化精度，因为在多数场景下MMSE量化精度要高于Normal和KL-Divergence量化算法。

默认情况下使用Normal量化算法，当遇到量化精度问题时可尝试使用KL-Divergence和MMSE量化算法。

针对a16系列算法，RKNN3-Toolkit提供了高精度的GRQ量化算法。

## 5.2.3 量化校正数据集建议

量化校正数据集用于计算激活值的量化范围，在选择量化校正数据集时应覆盖模型实际应用场景的不同数据分布，例如对于分类模型，量化校正数据集应包含实际应用场景中不同类别的图片。一般推荐量化校正数据集数量为20-200张，可根据量化算法的运行时间适当增减。需要注意的是，Normal量化在a16的情况下不需要设置量化数据集，其余量化算法在增加量化校正数据集数量会增加量化算法的运行时间但不一定能提高量化精度。

## 5.2.4 量化配置方法

RKNN3-Toolkit中量化的配置方法在 `rknn.config()` 和 `rknn.build()` 接口实现。其中量化方法配置由 `rknn.config()` 接口实现，量化开关和校正集路径的选择由 `rknn.build()` 接口实现。

`rknn.config()` 接口包含以下相关量化配置项：

1. `quantized_dtype`：选择量化类型，目前可选 `w8a8`、`w16a16`。
2. `quantized_algorithm`：选择量化算法，包括Normal，KL-Divergence、MMSE量化、GRQ、GDQ算法。可选值为 `normal`，`kl_divergence`、`mmse`、`grq`、`gdq`，默认为 `normal`。
3. `quantized_method`：选择Per-Layer、Per-Channel量化和Group量化。可选值为 `layer`、`channel`、`group{size}`，其中size代表分组数，默认为 `channel`。

`rknn.build()` 接口包含以下相关量化配置项：

1. `do_quantization`：是否开启量化，默认为 `False`。
2. `dataset`：量化校正数据集的路径，默认为空。

目前支持文本文件格式，用户可以把用于校正的图片(jpg或png格式)或npy文件路径放到一个.txt文件中。文本文件里每一行为一条路径信息，如：

```
a.jpg  
b.jpg
```

如有多个输入，则每个输入对应的文件用空格隔开，如：

```
a0.jpg a1.jpg  
b0.jpg b1.jpg
```

## 5.3 外部GRQ量化说明

RKNN3-Toolkit提供基于PyTorch的外部GRQ量化，用于通过权重微调和量化参数计算提升指定模型的量化精度。量化结果以 `.safetensors` 保存并可转换为 ONNX（示例见 RKNN3-Model-Zoo）。该功能依赖CUDA环境，且当前仅支持部分模型。外部GRQ量化会将微调后的权重与量化参数保存为 `.safetensors` 格式，再借助RKNN3-Model-Zoo工程转换为对应的ONNX文件及 `config.pkl`，以用于后续的RKNN模型转换。通常情况下，外部GRQ生成的ONNX模型在量化精度上优于直接使用RKNN3-Toolkit内置的GRQ。外部GRQ的使用示例如下：

```
from rknn.utils.grq import grq_quantize

ret = grq_quantize(model_path='Qwen/Qwen2.5-3B-Instruct', dataset_path='dataset.json',
save_dir='./grq_model/', group=32)
if ret:
    print("GRQ SUCCESS")
else:
    print("GRQ Failed")
```

接口包含以下相关量化配置项：

1. `model_path`：模型文件路径或名称。
2. `dataset_path`：量化输入的数据集。

3. `save_dir`：GRQ导出的.safetensor模型文件路径。
4. `group`：量化分组数，指定为-1时是Per-Channel量化。

## 5.4 导入外部量化模型说明

RKNN3-Toolkit 支持导入 **外部 AutoGPTQ 量化的 ONNX 模型**，目前仅支持 **w4a16 (4-bit 权重 + 16-bit 激活)** 的量化形式。

整体流程如下：

1. 使用 AutoGPTQ 对模型进行量化，并将权重及量化参数保存为 `.safetensors` 格式
2. 借助 **RKNN3-Model-Zoo** 工程，将量化模型转换为 **ONNX 模型** 和 `config.pkl` 配置文件
3. 通过 `load_llm` 接口将模型转换为 **RKNN 模型**

### 5.4.1 导出 ONNX 模型与 config.pkl

以下示例代码来自

`RKNN3-Model-Zoo/examples/Qwen2_5/python/export_llm.py`

```
model = AutoModelForCausalLM.from_pretrained(autogptq_model_path, **kwargs)

# Export LLM to ONNX
causal_llm_to_onnx(model, args)

# Export LLM configuration
export_llm_config(
    autogptq_model_path,
    config_path,
    chat_context,
    prompt
)
```

函数说明

- `causal_llm_to_onnx`
  - 用于将 AutoGPTQ 量化后的 LLM 转换为 ONNX 模型。
  - `model`: AutoGPTQ 加载的模型实例
  - `args`: 模型导出相关参数（如输出路径等）
- `export_llm_config`
  - 用于生成模型转换所需的配置信息文件 `config.pkl`，其中包含量化参数和推理相关配置。
  - `autogptq_model_path`: AutoGPTQ 模型路径
  - `config_path`: `config.pkl` 文件保存路径
  - `chat_context`: 会话模板
  - `prompt`: 提示词

### 5.4.2 量化模型配置要求

对于 AutoGPTQ 量化模型，在 `config.pkl` 中必须包含如下量化参数：

```
llm_config["q_params"] = {
    'bits': 4,           # 权重量化位数，目前仅支持 4-bit
    'sym': False,        # 是否使用对称量化
    'group_size': 32,    # 分组大小，支持 32 / 64 / 128 / -1 (per-channel)
}
```

注：

1. 当前 RKNN3-Toolkit 量化模型仅支持 4-bit 权重量化、
2. `group_size` 必须与 AutoGPTQ 量化时的设置保持一致
3. `sym` 通常为 `False`（非对称量化）

### 5.4.3 转换为 RKNN 模型

以下示例代码来自

RKNN3-Model-Zoo/examples/Qwen2\_5/python/export\_rknn.py

```
# Create RKNN object
rknn = RKNN(verbose=True)

# pre-process config
print('--> config model')
rknn.config(target_platform='rk1820',
            quantized_dtype='w4a16', quantized_algorithm='grq', quantized_method='group32',
            )
print('done')

# Load model
print('--> Loading model')
ret = rknn.load_llm(model=args.onnx_path, config=args.config) # 外部量化模型需要添加量化配置信息
if ret != 0:
    print('Load model failed!')
    exit(ret)
print('done')

# 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')

#Export rknn model
print('--> Export RKNN model')
ret = rknn.export_rknn(args.rknn_path)
if ret != 0:
    print('Export rknn failed!')
    exit(ret)
print('done')

rknn.release()
```

## 6 精度排查

模型精度问题排查一般从两个方面进行，一是模拟器精度排查，二是板端Runtime精度排查。模拟器推理结果正确是板端Runtime推理正确的前提，所以需优先保证模拟器推理结果正确，再进行板端Runtime精度问题的排查。

因此本章节将针对**模拟器精度排查**以及**Runtime精度排查**两个方面给出排查建议以及处理方案。下图为具体排查步骤：

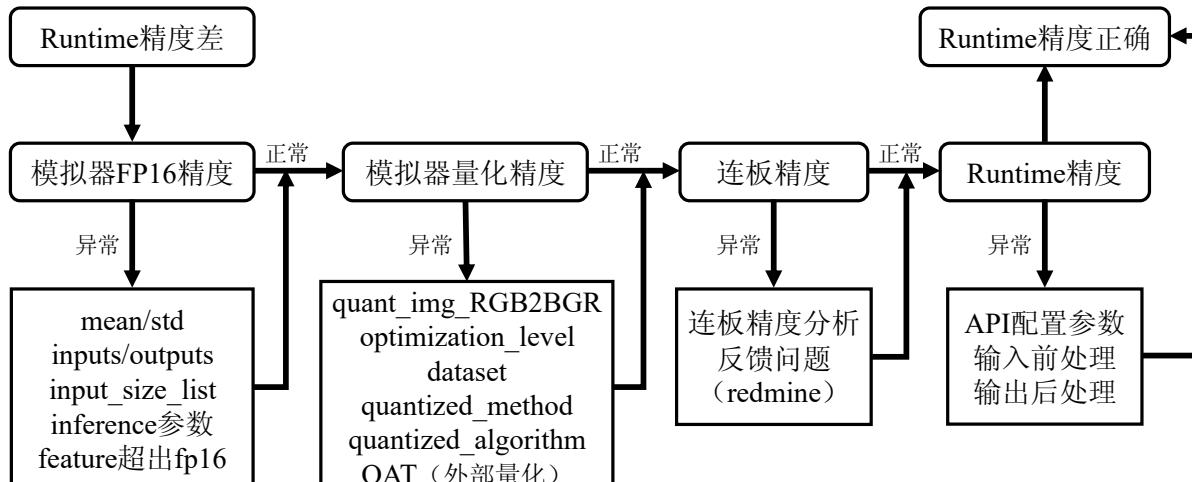


图6-1 精度排查步骤

- 模拟器精度排查，主要分为**模拟器FP16精度**和**模拟器量化精度**排查两个子步骤（上图深色框部分）。
- Runtime精度排查，主要分为**连板精度**和**Runtime精度**排查两个子步骤（上图浅色框部分）。

判断精度可以使用余弦距离作为基本的判断依据，但最终量化对精度的影响仍需要在数据集上验证。

## 6.1 模拟器精度排查

模拟器推理结果正确是板端Runtime推理正确的前提，所以需优先保证模拟器推理结果正确。

RKNN3-Toolkit上的模拟器推理根据模型是否量化分为**FP16推理**和**量化推理**。FP16推理结果正确是量化推理的结果正确的前提，因此当存在量化推理精度问题时，优先验证FP16推理的正确性，再排查量化推理的精度问题。

### 6.1.1 模拟器FP16精度

RKNPU3目前不支持FP32的计算方式，因此模拟器在不开启量化的情况下，默认是FP16的运算类型，所以只需要在使用 `rknn.build()` 接口时，将 `do_quantization` 参数设置为 `False`，即可以将原始模型转换为FP16的RKNN模型，接着调用 `rknn.init_runtime(target=None)` 和 `rknn.inference()` 接口进行FP16模拟推理并获取输出结果。

如果FP16推理输出结果错误，则可以进行以下排查：

- **配置错误**

模型的配置信息主要集中在 `rknn.config()` 接口里，同时在其他的API里也有少数的配置信息可能影响FP16的精度，主要参数如下：

**mean\_values / std\_values**: 模型的归一化参数，一般原始模型的输入归一化操作是放在模型的输入预处理里实现的，但RKNN模型在推理时可以包含该归一化的操作（在开启量化后，对量化校正数据集也会先进行归一化操作），因此在原始模型有归一化步骤时，要确保该参数和原始模型使用的归一化参数一致。

**input\_attrs**: 模型的输入属性，RKNN模型在推理时可以对输入属性进行修改，因此在修改输入属性后要确保输入和配置的属性一致

**input\_size\_list**: `rknn.load_tensorflow()`、`rknn.load_pytorch()` 和 `rknn.load_onnx()` 接口的输入节点shape信息，如果配置错误会导致错误的推理结果。

**inputs / outputs**: `rknn.load_tensorflow()` 和 `rknn.load_onnx()` 接口的输入和输出节点名称，如果配置错误会导致错误的推理结果。

**inference接口参数**: `rknn.inference()` 接口的输入参数，主要包括 `inputs` 和 `data_format`。

- 一般在 Python 环境下，图像数据都是通过 `cv2.imread()` 读取的。此时需要注意，`cv2.imread()` 读取的图像格式为 BGR。如果原始模型的输入为 BGR，则不需要做 RGB 顺序的调整；而如果原始模型的输入为 RGB，则需要调用 `cv2.cvtColor(img, cv2.COLOR_BGR2RGB)` 将图像数据转为 RGB。另外，通过 `cv2.imread()` 读取的图像的 shape 维度为 3 维，但一般模型的输入 shape 为 4 维，因此需要调用 `np.expand_dims(img, 0)` 将输入 shape 扩展为 4 维，之后才可以传给 `rknn.inference()` 接口进行推理。通过 `cv2.imread()` 读取的图像的 layout 为 NHWC，`data_format` 的默认值也是 NHWC，因此不需要设置 `data_format` 参数。
- 如果模型的输入数据不是通过 `cv2.imread()` 读取，此时必须清楚知道输入数据的 layout 并设置正确的 `data_format` 参数，同时也要确保输入数据的 shape 和原始模型一致。如果输入数据是图像数据，也要确保其 RGB 顺序与原始模型一致。

参数配置的检查是很重要的环节，是很多用户出现FP16推理输出结果错误的主要原因。具体排查步骤如下：

1. 使用原始模型在原始推理框架下进行推理，并将推理结果保存下来。
2. 使用RKNN3-Toolkit对原始模型进行转换并推理，此时需要使用与前一步骤里同样的输入数据，并设置FP16的推理方式（`rknn.build()` 的 `do_quantization` 设为 `False`），同时 `rknn.init_runtime()` 的 `target` 参数设为 `None`，以调用RKNN3-Toolkit的模拟器进行推理，同样将推理的结果保存下来。
3. 对比两次推理的结果，如果结果较为一致（可以用余弦距离来判断一致性），说明上述的配置都没有问题。
4. 如果结果不一致，检查上述参数是否正确。

如果确认上述参数配置无误，结果仍然不一致，则有可能是模型中间的Tensor超出FP16表达范围导致。

- **超出FP16表达范围**

模型的中间Tensor在由FP32转为FP16后，可能会出现运算溢出的问题。因为一般模型的推理数据类型是FP32，如果推理时模型中的Tensor有数值超过FP16表达范围（-65504~65504），则该Tensor就会溢出，导致模型推理结果异常。

对于溢出问题，可以通过配置 `rknn.inference()` 接口中的 `accuracy_analysis` 参数进行模拟器FP16精度分析，如果分析结果中的 `simulator_error` 的 `entire` 列或 `single` 列出现异常值（出现‘inf’等的字样），则可能出现了FP16溢出。此时可以尝试修改模型结构来保证模型中的所有Tensor不会出现FP16溢出（如添加一些BN层等）。

如果确认上述参数配置无误，并且也不是FP16溢出，但结果仍然不一致，则可能是模拟器内部实现问题，请将该模型的复现文件提供给瑞芯微NPU团队进行分析解决。

### 6.1.2 模拟器量化精度

在排除FP16精度问题后，可以对模型进行量化（使用 `rknn.build()` 接口时，将 `do_quantization` 参数设置为 `True`），然后通过调用接口 `rknn.init_runtime(target=None)` 和 `rknn.inference(accuracy_analysis=True)` 进行模拟器量化精度分析。

如果在分析结果中，发现 `simulator_error` 的 `entire` 列精度下降的比较厉害，并且 `simulator_error` 的 `single` 没有发现有哪层精度下降特别多的情况，则主要从以下几个方面进行排查：

- **配置错误**

与FP16推理的配置问题类似，错误的配置也会导致量化推理精度问题，因此在保证FP16推理正确的配置基础上，仍然要对以下量化配置参数进行检查。

**quant\_img\_RGB2BGR**: 表示在加载量化图像时是否需要先做RGB2BGR的操作，更多详细信息见 `quant_img_RGB2BGR` 参数说明，该参数务必和训练时的图像通道顺序保持一致，在配置错误时也会导致量化精度下降比较多。

**optimization\_level**: 优化等级的选择，默认为3，表示速度优先，这种情况下会开启一些对提升性能有益，但却会略微影响到精度的优化规则，将该配置调小（如改为0），则会禁用这些优化规则。

**dataset**: `rknn.build()` 接口的量化校正数据集配置，用于在量化过程中，计算每个Tensor合适的量化参数（scale / zero\_point）。如果选择了和实际部署场景差异较大的校正集，则可能会出现精度下降的问题，此外校正集的数量过多或过少都会影响精度（一般选择20~200张）。

若量化精度差，可以通过以下方式进行排查校准：

1. 直接进行量化推理，然后检查推理的结果与原始模型在原始推理框架下推理的结果进行比较，如果结果差异不是很大，则可以认为 `quant_img_RGB2BGR` 和 `dataset` 参数基本无误。
2. 若量化推理与原始模型推理结果差异大，可以按以下步骤依次进行排查校准：
  - 如原始模型输入的图像格式是BGR，此时可以修改 `quant_img_RGB2BGR` 为 `True`，关于模型输入的RGB顺序，其实从前面FP16推理精度验证步骤的输入数据的处理代码中可以得知输入数据的RGB顺序。
  - 可以先使用一张图像进行量化（`dataset.txt`中只留一行），推理时也使用这张图像进行推理，如果此时单张图像的精度提升较多，则说明先前使用的量化校正数据集选择不佳，可以重新选择与部署场景较吻合的图片（如果提升并不明显，则可能不是 `dataset` 的问题）。
  - 如原先只使用一张图像进行量化（`dataset.txt`中只有一行），此时可以尝试使用更多的图像进行量化，可以提高到20~200张左右。

经过上述配置排查之后，应该不会出现量化结果完全错误的情况，如果出现完全错误的情况，请重新检查上述的配置。在确认配置无误的情况下，如果模型的精度还是不够，可以尝试修改量化方法和算法等相关配置。

#### • 量化方法和量化算法

有些模型本身对量化并不友好，此时可以尝试切换不同的量化方法和量化算法。目前量化方法主要有五种，分别是layer、channel和group{size}，可通过 `rknn.config()` 接口里的 `quantized_method` 参数进行设置（默认是channel）。量化算法主要分为五种，分别是normal，kl\_divergence、mmse、grq和gdq，可通过 `rknn.config()` 接口里的 `quantized_algorithm` 参数进行设置（默认是normal）。步骤如下：

1. w8a8量化下如原先使用的是layer的量化方法，可以改为channel的量化方法，一般情况下，channel的量化方法精度比layer的量化方法精度会高许多；w4a16量化下如原先使用的是channel的量化方法，可以改为group{size}的量化方法；
2. w8a8量化下如量化方法已经是channel，但精度还是无法满足要求，此时可以将量化算法由normal改为kl\_divergence或mmse，这种方式会导致量化的时间大幅增加，但会带来比normal更好的精度表现，同时运行时的性能并不会受到影响；w4a16量化下可以将量化算法改为grq或gdq；

如果使用上述方法后，从分析结果中仍然发现 `simulator_error` 的 `entire` 列精度还是不好，并且 `simulator_error` 的 `single` 列有部分层精度掉的比较多，这可能是这些层的权重数值分布不好，导致量化后会出现精度下降较多的情况，说明此模型不适合量化。

## 6.2 Runtime精度排查

在模拟器精度正常的情况下，仍可能在板端C API部署时出现推理结果异常。出现这种问题的原因一般有三种：第一种是板端的Runtime的bug导致；第二种是调用RKNPU3的C API编程时接口没有正确使用导致；第三种是模型的前后处理不正确导致。

当遇到这种问题时，可以先通过连板功能快速排查是否是板端Runtime的bug导致，如果连板没有问题，再排查C API部署的问题。

### 6.2.1 连板精度

1. 在配置好连板调试环境的情况下，将开发板通过USB连接到电脑上，然后使用RKNN3-Toolkit进行连板推理（设置 `rknn.init_runtime()` 的 `target` 和 `core_mask` 参数，如 `target='rk1820'`、`core_mask=0xff`），并检查推理结果是否大致正确（因为模拟器并没有严格模拟NPU硬件，所以结果可能与模拟器并没有完全一致）。
2. 如果上述步骤里的推理结果与模拟器推理结果差异较大，则可以初步确定板端的Runtime存在bug，此时可以使用接口 `rknn.inference(accuracy_analysis=True)` 进行连板精度分析，精度分析完后会输出每层的分析结果。注：开启逐层精度分析功能需要在 `rknn.config` 中配置 `profile_mode=True`。
3. 检查分析结果中的 `runtime_error` 的 `single_sim` 列，如其cos余弦距离偏低或euc欧氏距离偏高（显示黄色或红色），从而导致 `runtime_error` 的 `entire` 列与 `simulator_error` 的 `entire` 列差异越来越大，则可能Runtime在实现该层时有出现精度丢失或异常的问题，此时可以将该分析结果以及复现的模型反馈给瑞芯微NPU团队进行修复。

### 6.2.2 Runtime精度

如果连板精度没有问题，但精度仍然有问题，则问题可能出在用户调用RKNN3的C API进行编程的C/C++代码本身，这时用户需要仔细检验下RKNN3的C API的接口配置等是否配置正确，以及模型的前处理和后处理流程是否正确（需要与模拟器端的流程完全一致）。可以按照以下步骤查看：

1. 检查输入配置和数据

查看C API的输入是否配置正确。例如，RKNN3-Toolkit在转换RKNN模型时已经配置均值和方差，则在C/C++代码中不需要做归一化。对于3通道的输入，通道顺序与模型训练时设置的输入通道是否一致；确认 `rknn3_create_mem()` 接口创建的内存大小、输入数据格式等属性以及最终传入的输入的正确性。

## 2. 检查输出配置和数据

在确保输入正确后，查看代码中输出属性和内存是否配置正确，确认 `rknn3_create_mem()` 接口创建的内存大小以及输出数据格式等属性是否正确。

## 7 性能优化

模型部署时，用户在跑通结果后，会有进一步的性能优化需求，此章节将从完整的模型性能优化流程来介绍如何调优。并展开介绍用户最常做的操作：模型性能分析，图级别优化，算子级别优化。完整优化流程如下图所示：

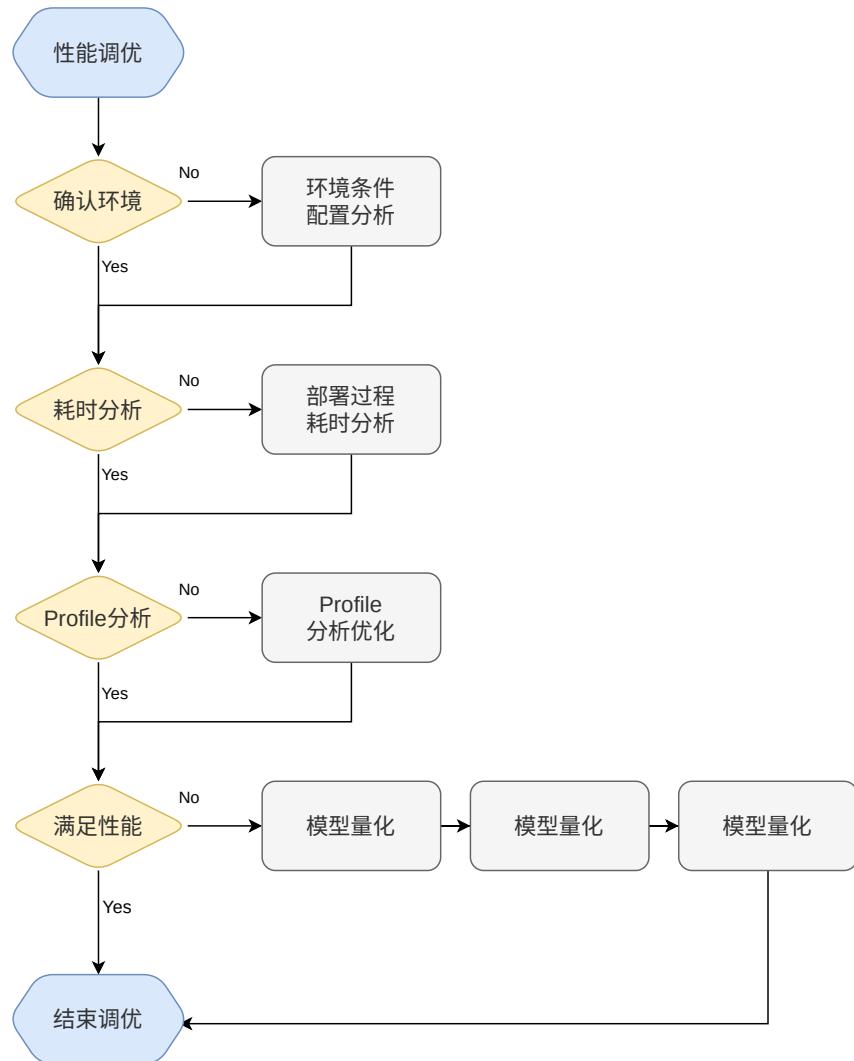


图7-1 模型性能优化流程

### 7.1 模型性能优化前期分析流程

#### 7.1.1 环境条件与配置检查

在所有性能分析及优化之初，应该优先确定测试环境的基准，只有在基准相同时，所测性能数据才是有意义的。未定频时测试同一模型的推理性能波动幅度较大，即使采用多次取平均所得的耗时数据仍不准确。

查询和设置测试环境的条件和配置有如下几个方面：

- **查询和设置DDR频率**

暂无，固件中已经定频

- **查询和设置CPU、NPU频率**

定频频率直接影响运行速率，频率越高性能相对越好。频率变化与性能提升不是线性关系。频率增加相对来说功耗也会增加。定频命令参考如下：

```
opp_dump      # 查询当前 CPU、NPU频率
srv_set_rate 1000 # 设置当前CPU频率，单位为MHz
npu_set_rate 1000 # 设置当前NPU频率，单位为MHz
```

```
# 查询频率
msh /data>opp_dump # 输入查询命令
srv opp table:
cur=10000000000,min=0,max=UINT64_MAX,req=1000000000 # cur=1000000000 当前cpu频率
rate=1000000000 volt=875000 len=0                      # 以下是可选配置cpu频率档位，rate为频率，volt为电压
```

```

rate=11000000000 volt=900000 len=64
rate=12000000000 volt=900000 len=52
rate=13000000000 volt=966000 len=52
rate=13100000000 volt=966000 len=48
npu opp table:
cur=10000000000,min=0,max=UINT64_MAX,reg=1000000000 # cur=1000000000 当前npu频率
rate=267000000 volt=850000 len=0 # 以下是可选配置npu频率档位, rate为频率, volt为电压
rate=400000000 volt=850000 len=0
rate=500000000 volt=850000 len=156
rate=600000000 volt=850000 len=108
rate=700000000 volt=850000 len=76
rate=800000000 volt=850000 len=52
rate=850000000 volt=850000 len=40
rate=900000000 volt=875000 len=40
rate=950000000 volt=912500 len=40
rate=1000000000 volt=950000 len=40
rate=1010000000 volt=950000 len=40
# 设置频率
msh /data>npu_set rate 1000      # 输入设置NPU命令
ACLK RKNN LOCSYS rate = 1000 MHz # 已设置NPU频率为1000MHz
msh /data>srv set rate 1000      # 输入设置CPU命令
CLK CORE rate = 1000 MHz         # 已设置CPU频率为1000MHz

```

- 检查NPU内核Driver版本

有些功能或者算子的性能优化项与内核驱动版本有关，较新的内核驱动版本能应用到最新的底层优化实现。所以请检查是否使用到较新的内核驱动版本。

驱动版本随开机启动日志打印如下：

```
I NPUTTransfer(2970): Starting RKNN3 Transfer Proxy, Transfer version x.x.x, devid = -1, pid = 2970:1341
```

x.x.x代表版本号，例如0.5.0。

- 检查NPU的负载

NPU的负载为单位时间内NPU执行任务的时间占比。负载能够反应NPU的繁忙程度，如果查询到的负载较低，则表明NPU等待任务提交的时间较长，需要检查数据输入输出拷贝用时以及应用程序前后处理进行优化等等。或者在应用程序中使用多线程处理方式来提升NPU负载。

另外需要注意的是，NPU的负载不代表实际的MAC利用率，MAC利用率反应的是执行中任务在NPU硬件单元中的执行效率。

在主机（如RK3588、RK3576）上查询NPU负载的命令如下：

```
rknn-smi info -w
```

Device(Idx)	Chip(Idx)	Power(mW)	Temp(C)	CPU(%)	NPU(%)	Memory(%)	Health
0	0	3300	49	2%	15%	20%	OK
0	0	15449	49	1%	99%	20%	OK
0	0	15272	49	0%	99%	20%	OK
0	0	15201	49	0%	99%	20%	OK
0	0	15177	49	0%	99%	20%	OK
0	0	15166	49	0%	99%	20%	OK
0	0	15201	49	0%	99%	20%	OK
0	0	14938	49	0%	99%	20%	OK
0	0	15414	49	0%	99%	20%	OK
0	0	15272	50	0%	99%	20%	OK
0	0	15473	49	0%	99%	20%	OK
0	0	15628	50	0%	99%	20%	OK
0	0	15829	49	0%	99%	20%	OK
0	0	15960	50	0%	99%	20%	OK
0	0	15853	49	0%	99%	20%	OK
0	0	16103	50	0%	99%	20%	OK
0	0	15961	49	0%	99%	20%	OK

图7-1 NPU负载

## 7.1.2 部署过程耗时分析

整个部署程序的推理部分耗时的占用有如下三个方面：用户应用程序耗时、输入输出数据传输耗时、模型推理耗时。分析各环节耗时占比能够直观的确定优化重点。测定这些步骤的耗时可以在应用程序上通过打时间戳的方式获得。

- **用户应用程序耗时**

用户应用程序耗时主要指推理过程中非NPU相关的耗时，一般来说主要是数据的前处理、后处理和逻辑代码的耗时。这部分耗时由用户全权控制。用户在发现应用程序的耗时占总体耗时非常高时，除精简优化代码以外，可以尝试将一部分操作通过专用硬件加速。

例如将一些矩阵乘加操作，采用Matmul API接口来调用NPU辅助执行计算。

- **输入输出传输耗时**

用户进行部署时，RK1820是作为协处理器存在的，需要与主机（如RK3588、RK3576）进行输入输出的数据传输，然后开始推理运算。需要减少该部分耗时，应该选择采用模型查询出的输入输出tensor对应的数据类型和排布类型。

- **推理耗时**

NPU执行推理的耗时，该部分耗时直接体现部署模型的耗时。受推理模型规模、编译优化版本影响。RKNN的LOG打印的推理耗时在不同 RKNN\_LOG\_LEVEL 等级时不一样，因为LOG打印存在一定的耗时。一般在查看单帧推理耗时时，设置的LOG等级为1，并且跑多次取平均为准。

## 7.2 模型性能分析

### 7.2.1 获取Profile信息

当需要了解模型推理逐层耗时情况时，首先需要模型转换的时候，设置rknn.config(profile\_mode=True, ...），其次在运行程序前输入以下指令打印详细信息：

```
# 在主机 RK3588/RK3576 中要建立 console 转发接口，然后直接对rk1820传输命令
rknn-console rk1820

# 直连 RK1820 时则跳过上一步，直接接下一步设置 RKNN_LOG_LEVEL 即可

# 设置 RKNN_LOG_LEVEL=4
uart_irq_set_owner node2
rknn_set_log_level 4
uart_irq_set_owner subsoc
rknn_srv_model_test model.rknn model.weight input.npy output.npy 0xff 1
```

性能分析报告信息如下（仅截出性能相关部分）

```
RKNN3: -----
RKNN3: Operator Time Consuming Ranking Table (core 0)
RKNN3: OpType          CallNum   CPU(us)   NPU(us)   Total(us)   Ratio(%)
RKNN3: exGlu            5          0        705        705       32.09%
RKNN3: Conv              25         0        586        586       26.67%
RKNN3: Reshape           21         0        442        442       20.12%
RKNN3: exSDPAttention    5          0        262        262       11.93%
RKNN3: Transpose         3          0        75          75       3.41%
RKNN3: Concat             1          0        32          32       1.46%
RKNN3: InputOperator     1          0        28          28       1.27%
RKNN3: OutputProc         1          0        24          24       1.09%
RKNN3: Split              1          0        23          23       1.05%
RKNN3: OutputOperator    1          0        20          20       0.91%
RKNN3: -----
RKNN3: Total(core 0) CPU:0   NPU:2197   All:2197 us
RKNN3: inference: 60405 us

RKNN3: MSG_RUN: 64691 us
```

图7-2 性能分析表

```

RKNN3: NPU core:0 task:0 blk:0 sub:0 op:InputOperator:136_shape4 time:28 us
RKNN3: NPU core:0 task:0 blk:0 sub:1 op:Transpose:Slice_31_2split_reshape_13_256_1_1 time:27 us
RKNN3: NPU core:0 task:0 blk:0 sub:2 op:Split:Slice_31_2split time:23 us
RKNN3: NPU core:0 task:0 blk:0 sub:3 op:Conv:MatMul_73_2gemm_2conv time:23 us
RKNN3: NPU core:0 task:0 blk:0 sub:4 op:Reshape:226_shape4_MatMul_101_tp_rs time:21 us
RKNN3: NPU core:0 task:0 blk:0 sub:5 op:Conv:MatMul_66_2gemm_2conv time:23 us
RKNN3: NPU core:0 task:0 blk:0 sub:6 op:Reshape:227_shape4_MatMul_99_rs time:21 us
RKNN3: NPU core:0 task:0 blk:0 sub:7 op:Conv:MatMul_58_2gemm_2conv time:23 us
RKNN3: NPU core:0 task:0 blk:0 sub:8 op:Reshape:209_shape4_MatMul_99_tp_rs time:21 us
RKNN3: NPU core:0 task:0 blk:0 sub:9 op:exSDPAttention:MatMul_101_2sdpa time:52 us
RKNN3: NPU core:0 task:0 blk:0 sub:10 op:Reshape:MatMul_108_2gemm_2conv_reshape1 time:21 us
RKNN3: NPU core:0 task:0 blk:0 sub:11 op:Conv:MatMul_108_2gemm_2conv time:23 us
RKNN3: NPU core:0 task:0 blk:0 sub:12 op:Conv:MatMul_167_2gemm_2conv time:23 us
RKNN3: NPU core:0 task:0 blk:0 sub:13 op:Reshape:341_shape4_MatMul_195_tp_rs time:21 us
RKNN3: NPU core:0 task:0 blk:0 sub:14 op:Conv:MatMul_160_2gemm_2conv time:23 us
RKNN3: NPU core:0 task:0 blk:0 sub:15 op:Reshape:342_shape4_MatMul_193_rs time:21 us
RKNN3: NPU core:0 task:0 blk:0 sub:16 op:Conv:MatMul_152_2gemm_2conv time:23 us
RKNN3: NPU core:0 task:0 blk:0 sub:17 op:Reshape:324_shape4_MatMul_193_tp_rs time:21 us
RKNN3: NPU core:0 task:0 blk:0 sub:18 op:exSDPAttention:MatMul_195_2sdpa time:53 us
RKNN3: NPU core:0 task:0 blk:0 sub:19 op:Reshape:MatMul_202_2gemm_2conv_reshape1 time:21 us
RKNN3: NPU core:0 task:0 blk:0 sub:20 op:Conv:MatMul_202_2gemm_2conv time:23 us
RKNN3: NPU core:0 task:0 blk:0 sub:21 op:Conv:MatMul_261_2gemm_2conv time:23 us
RKNN3: NPU core:0 task:0 blk:0 sub:22 op:Reshape:456_shape4_MatMul_289_tp_rs time:21 us
RKNN3: NPU core:0 task:0 blk:0 sub:23 op:Conv:MatMul_254_2gemm_2conv time:23 us
RKNN3: NPU core:0 task:0 blk:0 sub:24 op:Reshape:457_shape4_MatMul_287_rs time:21 us
RKNN3: NPU core:0 task:0 blk:0 sub:25 op:Conv:MatMul_246_2gemm_2conv time:23 us

```

图7-3 逐层性能

可以针对总体性能Profile和逐层性能Profile快速定位想要的信息。并根据数据来制定后续不同侧重的优化策略。获得Profile后可以进行如下分析：分析逐层耗时，找出高耗时算子。下文将详细讨论。

### 7.2.2 Session运行LLM模型时，获取Profile信息

rknn3\_session\_demo 工程中，执行 `./run.sh perf` 后可以打印如下performance信息：

```

=====
Performance Summary
=====
Model Context | Rope Length | Input Tokens | New Tokens | TTFT(ms) | TPOT(ms) | Prefill TPS | Decode TPS
1024          | 8192        | 128         | 128        | 26.43     | 7.36      | 4842.43    | 135.81
=====
```

名词解释：

- Model Context：模型上下文长度。
- RoPE Length：旋转位置编码长度。
- Input Tokens：输入 Token 数。
- New Tokens：生成 Token 数。
- TTFT：从请求开始到第一个 Token 生成完成所消耗的时间。
- TPOT：Decode 阶段平均每生成一个 Token 所需的时间。
- Prefill TPS：Prefill 阶段模型每秒处理的 Token 数。
- Decode TPS：Decode 阶段模型每秒生成的 Token 数。

### 7.2.3 分析逐层耗时

如下图中，可以找出耗时高的算子，优先优化高耗时算子。

图7-4-1 高耗时算子性能分析

```

RKNN3: -----
RKNN3: Operator Time Consuming Ranking Table (core 0)
RKNN3: OpType          CallNum  CPU(us)  NPU(us)  Total(us)  Ratio(%)
RKNN3: exGlu           5        0         705       705       32.15%
RKNN3: Conv             25       0         584       584       26.63%
RKNN3: Reshape          21       0         441       441       20.11%
RKNN3: exSDPAttention   5        0         259       259       11.81%
RKNN3: Transpose        3        0         77        77        3.51%
RKNN3: Concat            1        0         32        32        1.46%
RKNN3: InputOperator    1        0         28        28        1.28%
RKNN3: OutputProc       1        0         24        24        1.09%
RKNN3: Split             1        0         23        23        1.05%
RKNN3: OutputOperator   1        0         20        20        0.91%
RKNN3: -----
RKNN3: Total(core 0) CPU:0  NPU:2193 All:2193 us
RKNN3: inference: 60405 us

RKNN3: MSG_RUN: 64689 us

```

图7-4-2 高耗时算子性能分析

值得说明的是，高耗时算子不一定是低效算子，某些算子是高算力消耗的，高耗时的算子如果Mac利用率很高时，应该考虑能否降低此算子的尺寸规模以减少耗时。但当利用率很低的高耗时算子出现时，这类算子就是优化重点。

#### 7.2.4 分析CPU算子影响

如下图中，可以看到某些高耗时的算子是运行在CPU上的，将这些CPU算子NPU化将可以极大改善高耗时影响。一般来说，用户的大部分性能优化问题都会在将CPU算子NPU化后得到解决。因此要重点注意CPU算子的耗时情况。

Operator Time Consuming Ranking Table						
OpType	CallNumber	CPUTime(us)	GPUTime(us)	NPUTime(us)	TotalTime(us)	TimeRatio(%)
ConvRelu	125	0	0	54613	54613	42.09%
ConvExSwish	35	0	0	20047	20047	15.45%
ReduceMax	7	16104	0	0	16104	12.41%
ArgMax	7	14833	0	0	14833	11.43%
Concat	36	0	0	10464	10464	8.06%
Conv	25	0	0	3685	3685	2.84%
exSoftmax13	1	0	0	1916	1916	1.48%
Resize	11	0	0	1893	1893	1.46%
Sigmoid	12	0	0	1595	1595	1.23%
Reshape	14	388	0	699	1087	0.84%
Mul	7	0	0	1065	1065	0.82%
Add	7	0	0	1046	1046	0.81%
MaxPool	9	0	0	784	784	0.60%
OutputOperator	32	569	0	0	569	0.44%
ConvSigmoid	1	0	0	40	40	0.03%
InputOperator	1	9	0	0	9	0.01%

图7-5 CPU算子性能分析

一般来说算子运行在非NPU上的原因有如下几种：

- 算子尺寸超限（查询OpList文档的算子尺寸限制）
- 算子尚未支持在NPU上运算（查询OpList是否支持该算子，可以在Github工程上提Issue）
- NPU硬件限制无法支持（是否可以算法等效成其他NPU可支持的其他实现）

## 7.2.5 分析NPU算子性能瓶颈(目前暂不支持, 后续版本增加)

考虑NPU算子的高耗时问题时, 可以根据DDR Cycles/NPU Cycles/Total Cycles这三栏来判断该算子耗时的理论瓶颈是带宽瓶颈还是算力瓶颈。这里的DDR Cycles是根据NPU频率换算过后的数据, 指该层算子读写数据换算成NPU频率下所需的Cycle数, 因此可以直接与NPU Cycle比较。

如下图所示:

ID	OpType	DataType	Target	InputShape	OutputShape	DDR Cycles	NPU Cycles	Total Cycles	Time(us)	MacUsage(%)	Task Number	Regcmd	Size	Rv
0	InputOperate	UINT8	CPU	\	(1,3,300,300)	0	0	0	12 \	0	0	0	0	
1	Conv	UINT8	NPU	(1,3,300,300)(1,3,300,300)		261543	1582	261543	585	0.3	0	0	0	1
2	Split	INT8	CPU	(1,3,300,300)(1,1,300,300).(1		0	0	0	6677 \	0	0	0	0	
3	Conv	INT8	NPU	(1,1,300,300)(1,8,150,150)		275553	17226	275553	1556	1.23	0	0	0	1
4	Conv	INT8	NPU	(1,1,300,300)(1,8,150,150)		275553	17226	275553	1505	1.27	0	0	0	1
5	Conv	INT8	NPU	(1,1,300,300)(1,8,150,150)		275553	17226	275553	1573	1.22	0	0	0	1
6	Concat	INT8	CPU	(1,8,150,150)(1,24,150,150)		0	0	0	4684 \	0	0	0	0	1
7	Conv	INT8	NPU	(1,24,150,150)(1,64,150,150)		329723	67500	329723	684	10.96	0	0	0	2
8	Clip	INT8	NPU	(1,64,150,150)(1,64,150,150)		438387	0	438387	729 \	0	0	0	0	2
9	MaxPool	INT8	NPU	(1,64,150,150)(1,64,75,75)		275490	0	275490	725 \	0	0	0	0	1
10	Conv	INT8	NPU	(1,64,75,75),((1,64,75,75)		110676	45000	110676	281	17.79	0	0	0	
11	Clip	INT8	NPU	(1,64,75,75),((1,64,75,75)		109676	0	109676	275 \	0	0	0	0	
12	Conv	INT8	NPU	(1,64,75,75),((1,192,75,75)		253595	1215000	1215000	1456	92.72	0	0	0	1
13	Clip	INT8	NPU	(1,192,75,75)(1,192,75,75)		328817	0	328817	599 \	0	0	0	2	
14	MaxPool	INT8	NPU	(1,192,75,75)(1,192,38,38)		206599	0	206599	535 \	0	0	0	1	
15	AveragePool	INT8	CPU	(1,192,38,38)(1,192,38,38)		0	0	0	16159 \	0	0	0	0	
16	Conv	INT8	NPU	(1,192,38,38)(1,32,38,38)		50564	17328	50564	192	10.03	0	0	0	
17	Clip	INT8	NPU	(1,32,38,38),(1,32,38,38)		14169	0	14169	121 \	0	0	0	0	
18	Conv	INT8	NPU	(1,192,38,38)(1,64,38,38)		58609	34656	58609	193	19.95	0	0	0	
19	Clip	INT8	NPU	(1,64,38,38),(1,64,38,38)		28233	0	28233	132 \	0	0	0	0	
20	Conv	INT8	NPU	(1,64,38,38),(1,96,38,38)		43855	155952	155952	304	57	0	0	0	
21	Clip	INT8	NPU	(1,96,38,38),(1,96,38,38)		42297	0	42297	152 \	0	0	0	0	
22	Conv	INT8	NPU	(1,96,38,38),(1,96,38,38)		55095	233928	233928	399	65.14	0	0	0	
23	Clip	INT8	NPU	(1,96,38,38),(1,96,38,38)		42297	0	42297	153 \	0	0	0	0	
24	Conv	INT8	NPU	(1,192,38,38)(1,64,38,38)		58609	34656	58609	144	26.74	0	0	0	

图7-6 NPU算子性能瓶颈分析

- 其中第三层 DDR Cycles 远大于 NPU Cycles时, 说明该层读写数据花费Cycle数量远大于运算所需Cycle数量, 所以该Conv瓶颈来自带宽。
- 其中第十二层 DDR Cycles 远小于 NPU Cycles时, 说明该层读写数据花费Cycle数量远小于运算所需Cycle数量, 所以该Conv瓶颈来自算力。

目前仅统计Conv算子的NPU Cycles, 其他算子类型的NPU Cycles统计未来版本会逐步补充。

## 7.3 量化加速

模型量化能大幅降低模型的运算规模, 节约带宽消耗。其次采用量化的模型会在量化运算单元中执行运算, 在算力上, 量化运算单元比非量化运算单元的算力规模更大, 理论算力更高。因此采用量化进行加速优化, 能够从带宽和算力两个维度实现性能提升。模型量化的具体使用方式详见[量化说明](#)。

## 7.4 图级别优化

模型的图级别优化是最容易从整体角度去统筹优化模型的方法。在分析出耗时占比较高的算子或图区域后, 我们可以有多种不同的方式去改造图进而达成优化的目的。图优化主要以节省多余算子、非NPU OP的NPU化、面向硬件高效率算子改造等为目标。这些目标有可能有些时候是存在矛盾的, 例如为了非NPU OP的NPU化, 可能需要额外多出几个算子, 看似违背了节省多余算子的目标, 但总体推理性提升, 便是有意义的。

在RKNN-Toolkit3工具链中, 软件栈在转换模型的过程中已经会进行一定程度上的图优化。但这一过程不是万能和尽善尽美的, 有些未被考虑的场景仍然会出现冗余的操作, 用户可以根据本节介绍的一些思路来进行预先性的图优化。以下仅作为每一种优化方法的介绍, 不是强制固定, 实际场景需要灵活运用。

### 7.4.1 非NPU OP通过图变换实现NPU化

对于非NPU OP, 可以做一些等效的图变换来替换成NPU可支持的算子, 以达成NPU化的优化目的。

以下图shufflenetv2\_0.5模型为例, 将其中channel shuffle操作改为卷积近似替换。weight数值为0/1, 可以达成重排数据的效果。

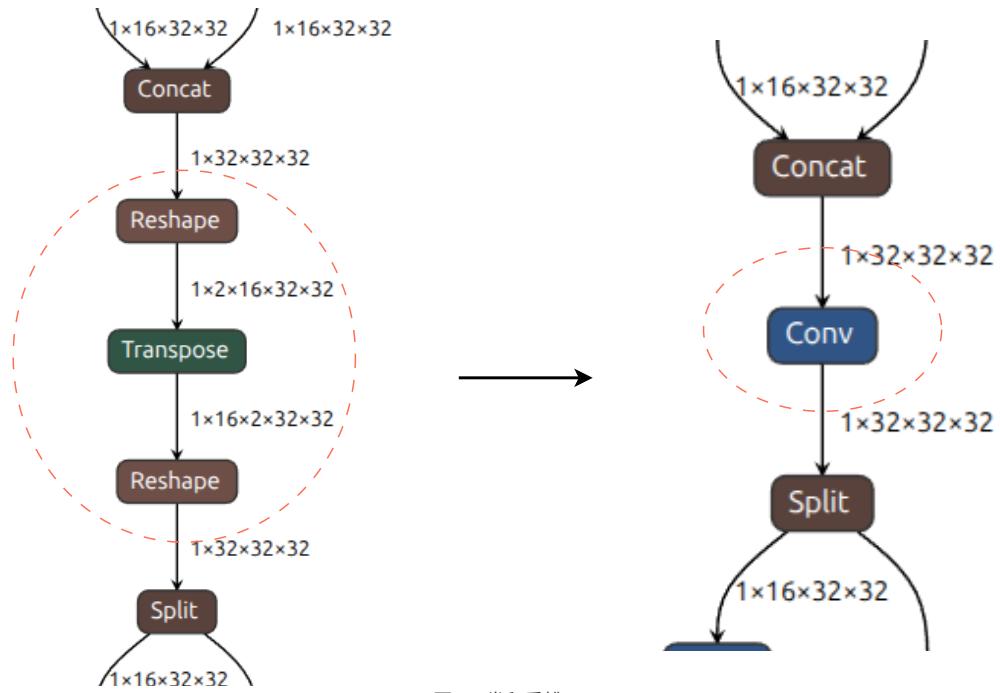


图7-7 卷积重排

#### 7.4.2 利用硬件Fuse特性设计网络或图优化

NPU支持一些算子组合进行融合，可以适当调整部分算子的运算流程，以适应NPU的融合规则实现算子融合优化。

尽管RKNN软件栈会有一定程度的图优化，但无法做到全面覆盖到所有情况。某些特殊情况下出现了理论上可融合简化，但最终图优化未能融合的图结构，用户可以算法上手动调整以快速解决该优化问题。

例如下图，在不改变计算正确性的情况下，通过调整Transpose与Clip算子的顺序，使得Conv与Clip融合运算，提高了性能。

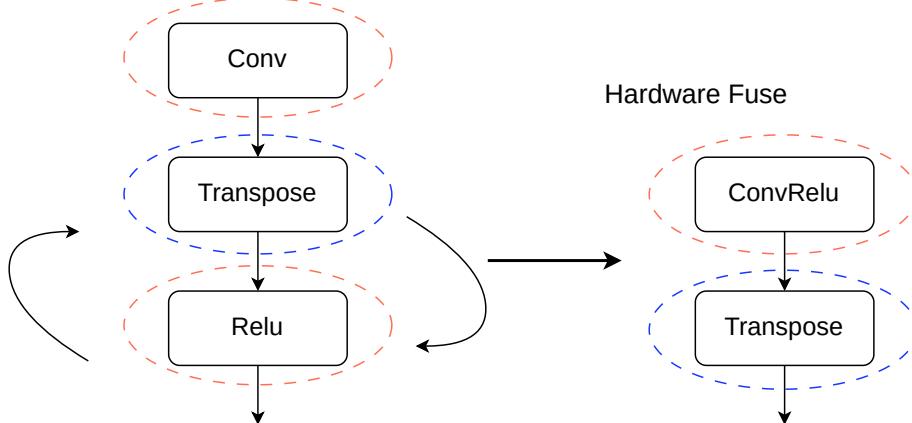


图7-8 算子优化/融合

利用融合规则设计，融合规则如下：

已支持的融合规则	未来计划支持的融合规则
Conv+Relu	Activation+Add(Mul)
Conv+PReLU(LeakyRelu)	Add(Mul)+Activation
Conv+Clip	Conv+Mul
Conv+Sigmoid(Tanh/Elu/Silu...)	Conv+Activation+Mul
Conv+Add	Conv+Activation+Pooling
Conv+Activation+Add	Conv+Activation+Add(Mul)+Pooling

### 7.4.3 算法等效变换或者子图单OP化

由于算法实现设计上，不会优先考虑具体部署的性能因素，往往容易在局部图上产生出复杂冗余的图结构。可以通过算法等效的方式，将某个区域的子图单op化，减少算子计算步骤，达成优化目的。

例如下图为Yolov5-nano 等效图变换，将若干复杂的Slice取数融合到Conv中，形成一个新的Conv，极大简化了图结构。

方案来源：<https://github.com/ultralytics/yolov5/issues/4825>

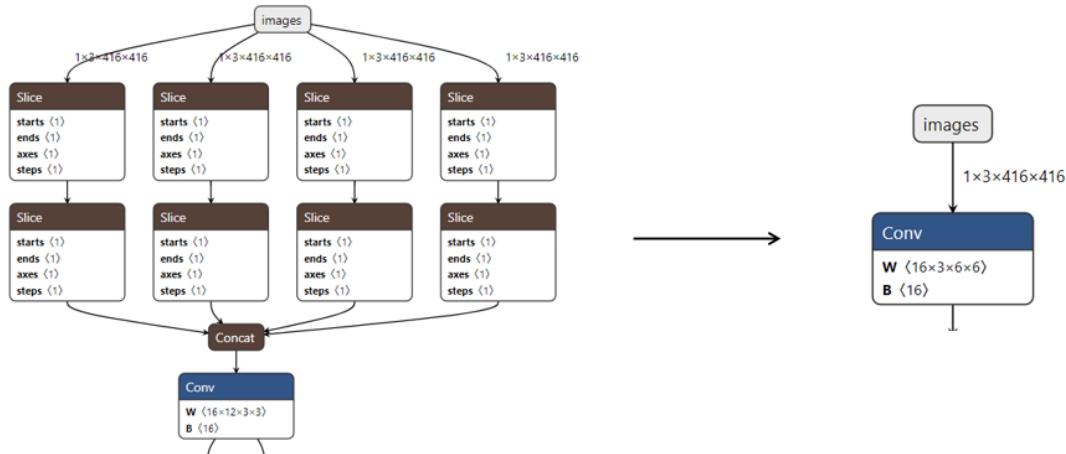


图7-9 算子等效图变换

### 7.4.4 算子等效进行“同类项合并”、“提取公因式”

某些算子连续多次运算时，可以简省合并为同一个算子，如Reshape、Transpose、Slice、部分Add/Mul/Sub/Div等。

例如下图可以通过简单调整图顺序以达成同类算子合并目的。

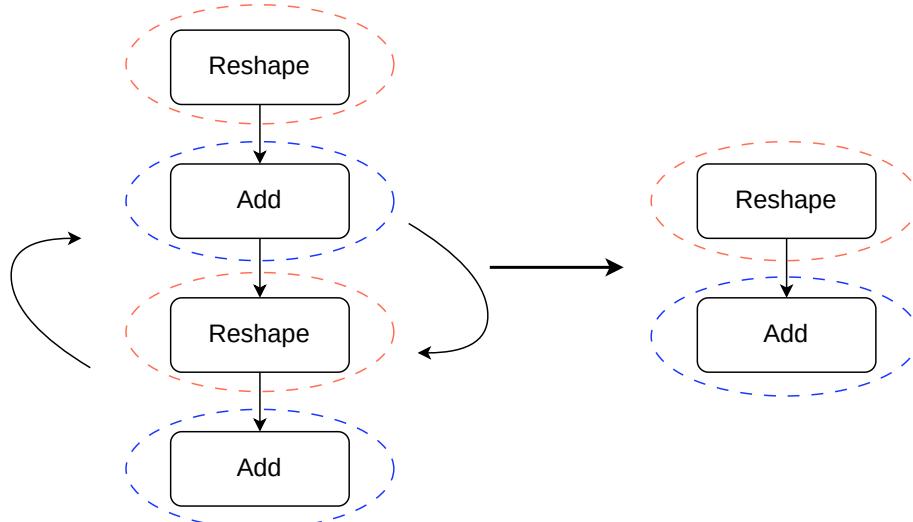


图7-10 同类项算子合并

某些图结构有一些共有部分的同类型操作可以调整顺序以提取成单一操作。

例如下图可以通过调整算子顺序将重复性的同类算子单独提取出来只执行一次操作。

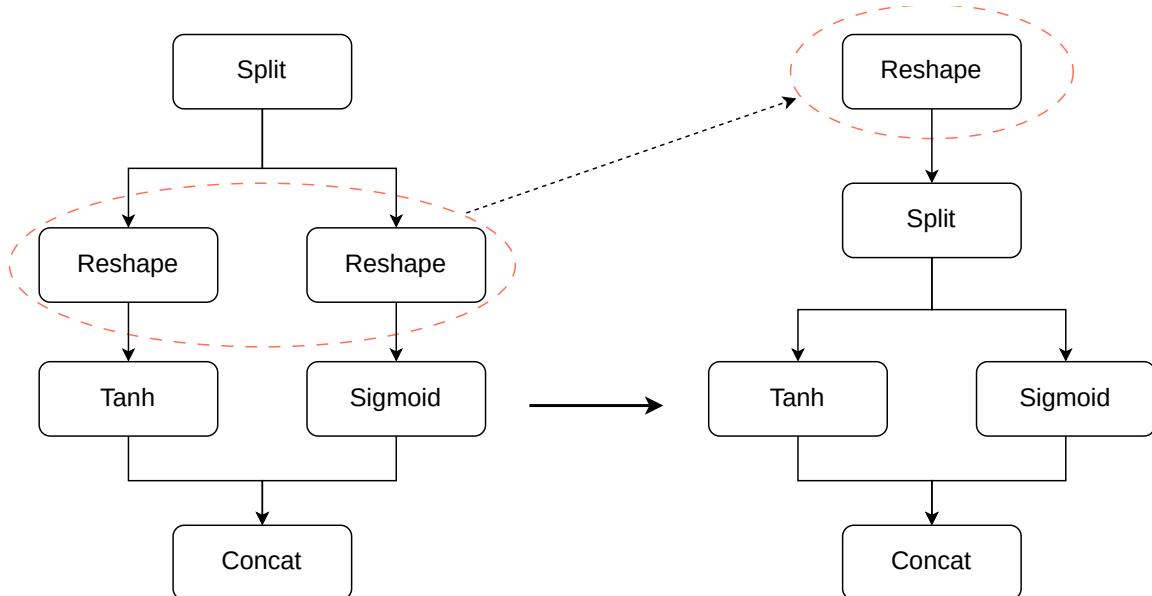


图7-11 重复性算子合并

## 7.5 算子级别优化

模型的算子级别优化是针对性比较强的细节优化阶段，该优化指对于某些特定算子进行具体改造设计，以达到进一步地提升性能。算子的主要方向是以硬件实现效率最高的尺寸规格来进行尺寸设计。例如某些算子尺寸规模相似，对齐与非对齐的运行耗时可能差别巨大，差别的原因在于硬件对于部分非对齐尺寸的算子会需要额外的冗余操作来保证正确性，因此算子的尺寸设计对于模型性能也能起到很大的影响，用户可以根据如下一些思路来进行算子优化。

### 7.5.1 面向DDR性能优化的OP尺寸设计（非强制）

在一些对齐尺寸下，除NPU运算效率更高外，对于DDR的读写也更友好，同等带宽条件下，更友好的读写会提升DDR的带宽效率，从而达到更好的性能。以下列出一些对于DDR读写更友好的尺寸规则，这些规则不强制。

- Channel按对齐量对齐

对齐表格如下所示：

表7-1 RK1820/RK1828

Dtype	Conv		Depthwise Conv	Other OP
	Input Channel	Output Channel	Input & Output Channel	
Int8	32	32	64	32
Int16	32	16	32	16
Float16	32	16	32	16
BFloat16	32	16	32	16

- Height \* Width > 1时，Height \* Width 4对齐
- 同等规模的算子，Width大Height小的尺寸，面向DDR读写更友好。如下图所示，右图卷积效率高于左图卷积

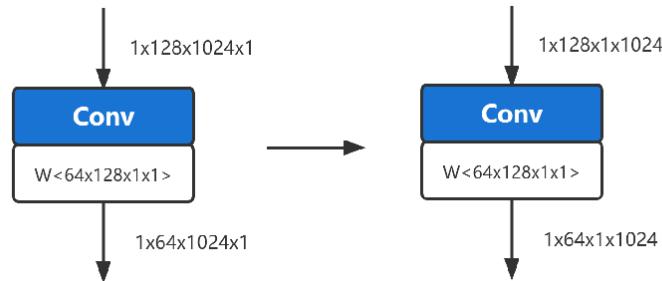


图7-12 同等规格卷积对比

### 7.5.2 高利用率模型算子的设计

要在模型设计上整体提高算力的利用率，一般要尽量避免低效算子，以及选择容易跑出更高利用率的算子尺寸设计。这里主要列出一些可以尽可能避免的低效算子、讨论卷积尺寸与利用率的关系。

- 规避低效算子原则设计

模型中尽量减少低效算子的使用，低效算子主要有三类，如表7-5所示

表7-5 三类低效算子

数据搬运类	尺寸变换类	非ReLU类激活函数
Transpose	Resize	Sigmoid
Reshape	Tile	Tanh
Split	Pooling	Softplus
Concat	Pad	Hardswish

- 卷积尺寸与利用率关系的讨论

由于卷积的性能会受到算力和带宽的双重影响，在评估性能时常采用MAC利用率来说明硬件算力发挥程度。

卷积尺寸同时与硬件计算效率和带宽读写效率相关，进而影响到最终的MAC利用率，因此这里讨论卷积尺寸与利用率关系，作为用户设计模型的性能参考因素。以下根据经验数据来作为一个大致的参考：

以下名词注释：KH (kernel height), KW (kernel width), KC (kernel channel), type\_bytes (权重位宽除以8), Ksize (KW或KH), Kstride (KW或KH方向上的stride)

- 卷积的输入输出Tensor的Channel符合对齐要求时（见8.4.1中对齐表格数据），利用率更高。
- 输入Tensor的 Channel < 256 时利用率相对较高，当Channel > 512以后，随着Channel增大，利用率会逐渐下降。
- 权重尺寸上， KH\* KW \* KC \* type\_bytes < 6K Bytes 时利用率相对较高。当超过一定大小后利用率会明显下降。
- Ksize / Kstride 的比值越大，利用率相对更高。例如 Ksize=3, Kstride=1优于Ksize=2, Kstride=1
- 输出Tensor的Height \* Width < 32 时利用率下降。
- 输出Tensor的Channel越大，利用率越高。

以上讨论仅是考察独立尺寸影响利用率的因素，实际部署模型里的卷积所呈现出的实测利用率则是诸多因素综合后的结果，开发者如果对某一卷积性能不够满意，希望通过提升利用率以优化其性能，可以参考上述尺寸与利用率关系的讨论进行针对性调整。

### 7.5.3 子图融合的匹配

RKNN软件栈会将某些特定的图关系匹配成自定义算子，如下图所示，如果没有被融合成对应的算子，可以检查是否因连接关系不同导致没有匹配上。

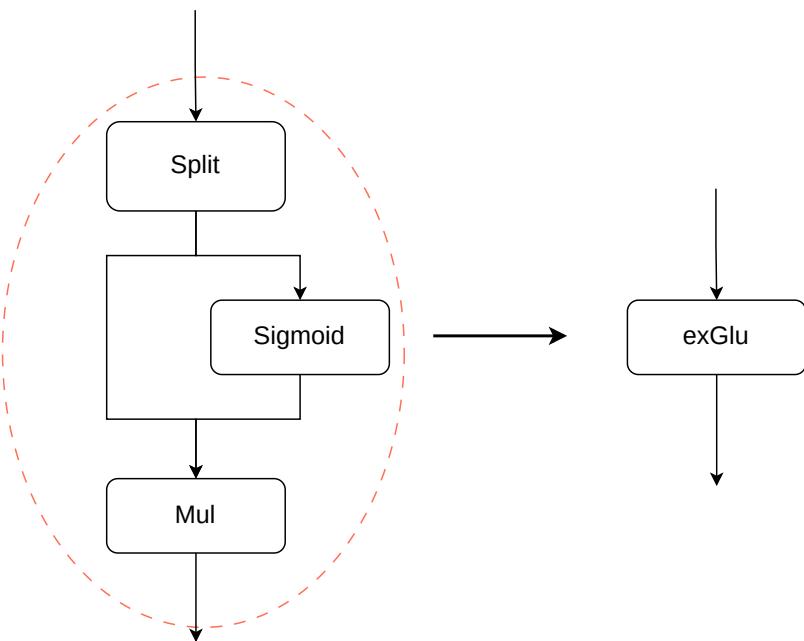


图7-13 Glu子图融合

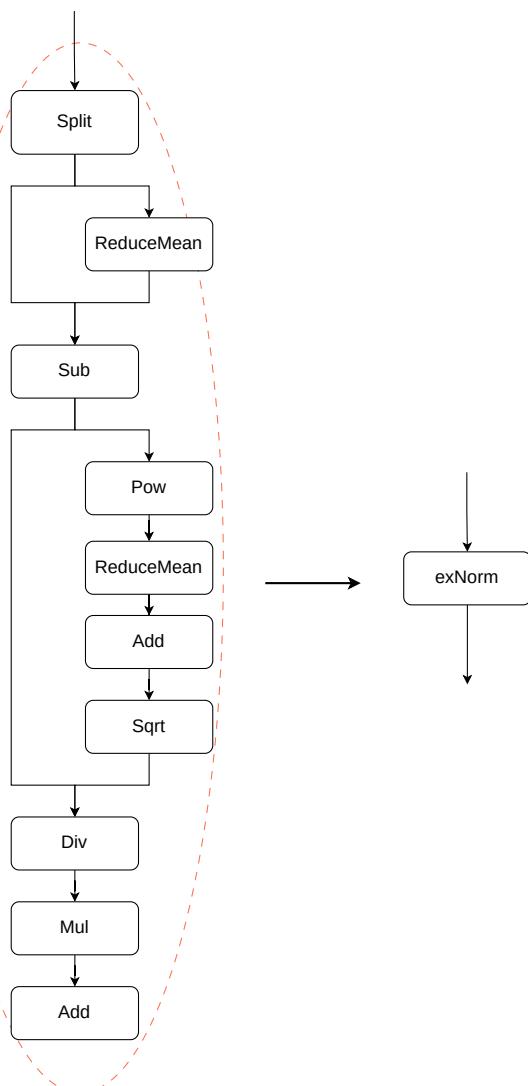


图7-14 exNorm子图融合

目前已经支持的子图融合规则有：

- Split + Sigmoid + Mul -> GLU

- ReduceMean + Sub + Pow + ReduceMean + Add + Sqrt + Div (+ Mul + Add) -> exNorm
- MatMul + Add + Softmax + MatMul -> exSDPAttention

## 8. 内存使用优化

### 8.1 模型运行时内存组成及分析方法介绍

#### 8.1.1 RKNN模型运行时内存组成

RKNN 模型运行时内存主要由权重、internal tensor、寄存器配置、输入输出tensor四种组成，若是大语言模型则要增加一个KVCache内存。运行时的内存通常是在 rknn3\_model\_init 和 rknn3\_session\_init 的时候创建完成。

#### 8.1.2 模型内存分析方法

在 rknn3\_model\_init() 或者 rknn3\_session\_init() 接口调用完毕后，用户就可以查看模型分配的相关内存。通过调用 rknn3\_query 接口，传入 RKNN3\_QUERY\_CORE\_MEM\_SIZE 即可查询模型在各个npu核心的权重和internal的内存(不包括输入和输出)的占用情况。

以下是示例代码：

```
rknn3_core_mem_size* core_mem_sizes;
uint32_t core_num = 0;
...
ret = rknn3_init(&ctx, NULL);
if (ret < 0) {
{
    printf("rknn_init fail ret=%d\n", ret);
    return ret;
}

// Load RKNN Model
ret = rknn3_load_model_from_path(ctx, model_path, weight_path);
if (ret < 0) {
    printf("rknn_load_model failed! ret=%d\n", ret);
    return ret;
}

//Init RKNN Model
ret = rknn3_model_init(ctx, &config);
if (ret < 0) {
    printf("rknn_model_init failed! ret=%d\n", ret);
    return ret;
}

//查询模型的core_num
ret = rknn3_query(app_ctx->vision.rknn_ctx, RKNN3_QUERY_CORE_NUMBER, &core_num, sizeof(core_num));
if (ret < 0) {
    printf("rknn3_query failed! ret=%d\n", ret);
    return ret;
}

//获取各个核心上模型占用weight和internal内存大小
rknn_mem_size core_mem_sizes;
ret = rknn3_query(app_ctx->vision.rknn_ctx, RKNN3_QUERY_CORE_MEM_SIZE, core_mem_sizes,
sizeof(rknn3_core_mem_size) * core_num);
if (ret < 0) {
    printf("rknn3_query core memory size failed! ret=%d\n", ret);
    return ret;
}

for (int i = 0; i < core_num; i++) {
    printf("core id=%d weight size: %d, internal size: %d\n", core_mem_sizes[i].core_id,
core_mem_sizes[i].weight_size, core_mem_sizes[i].internal_size);
}
```

### 8.2 Internal内存复用

由于协处理器上的内存资源十分有限，在部署多模态模型时，若前置的数据处理模型（如视觉模型）与大语言模型采用串行推理方式，则可以在两者之间复用同一段 Internal 内存，从而显著提升内存利用率，降低整体内存占用。以VLM模型为例，在视觉模型完成前向推理、提取图像特征并传递给 LLM 后，其占用的 Internal 内存可以被复用，供后续 LLM 的中间计算使用。这种“时间分片 + 空间复用”的策略在一定程度上可以缓解内存瓶颈问题。

以下是一个 VLM 实现 Internal 内存复用的示例代码：

```

...
int ret = -1;

uint32_t core_num_vision = 0;
uint32_t core_num_llm = 0;
//获取vision模型的core_num
ret = rknn3_query(app_ctx->vision.rknn_ctx, RKNN3_QUERY_CORE_NUMBER, &core_num_vision,
sizeof(core_num_vision));
if (ret < 0) {
    printf("rknn3_query failed! ret=%d\n", ret);
    return ret;
}
//获取llm模型的core_num
ret = rknn3_query(app_ctx->llm.rknn_ctx, RKNN3_QUERY_CORE_NUMBER, &core_num_llm, sizeof(core_num_llm));
if (ret < 0) {
    printf("rknn3_query failed! ret=%d\n", ret);
    return ret;
}

rknn3_core_mem_size* core_mem_sizes_vision = (rknn3_core_mem_size*)malloc(sizeof(rknn3_core_mem_size) *
core_num_vision);
if (!core_mem_sizes_vision) {
    printf("Failed to allocate memory for core_mem_sizes_vision\n");
    return ret;
}
rknn3_core_mem_size* core_mem_sizes_llm = (rknn3_core_mem_size*)malloc(sizeof(rknn3_core_mem_size) *
core_num_llm);
if (!core_mem_sizes_llm) {
    printf("Failed to allocate memory for core_mem_sizes_llm\n");
    return ret;
}
//获取vision在各个npu核心上模型占用weight和internal内存大小
ret = rknn3_query(app_ctx->vision.rknn_ctx, RKNN3_QUERY_CORE_MEM_SIZE, core_mem_sizes_vision,
sizeof(rknn3_core_mem_size) * core_num_vision);
if (ret < 0) {
    printf("rknn3_query core memory size failed! ret=%d\n", ret);
    return ret;
}
//获取llm在各个npu核心上模型占用weight和internal内存大小
ret = rknn3_query(app_ctx->llm.rknn_ctx, RKNN3_QUERY_CORE_MEM_SIZE, core_mem_sizes_llm,
sizeof(rknn3_core_mem_size) * core_num_llm);
if (ret < 0) {
    printf("rknn3_query core memory size failed! ret=%d\n", ret);
    return ret;
}

int llm_to_vision[core_num_llm];
for (int i = 0; i < core_num_llm; i++) { llm_to_vision[i] = -1; }

// 遍历视觉模型的核心和 LL M 的核心, 寻找运行在相同npu核心上的情况
int core_num_same = 0;
for (int i = 0; i < core_num_vision; i++) {
    for (int j = 0; j < core_num_llm; j++) {
        if (core_mem_sizes_vision[i].core_id == core_mem_sizes_llm[j].core_id) {
            uint64_t internal_size = std::max(core_mem_sizes_vision[i].internal_size,
core_mem_sizes_llm[j].internal_size);
            core_mem_sizes_vision[i].internal_size = internal_size;
            core_mem_sizes_llm[j].internal_size = internal_size;
            core_num_same++;
            llm_to_vision[j] = i;
            break;
        }
    }
}

// 为所有实际需要分配的 Internal 内存块指针申请存储空间, 总共需要分配的 Internal 内存块数 = 视觉模型数量 + LL M 数量 - 重叠的核心数 (重用的部分)
app_ctx->n_internal_mems = core_num_vision + core_num_llm - core_num_same;
app_ctx->internal_mems = (rknn3_tensor_mem**)calloc(app_ctx->n_internal_mems, sizeof(rknn3_tensor_mem));
if (!app_ctx->internal_mems) {
    printf("Failed to allocate memory for app_ctx->internal_mems array\n");
    return -1;
}

```

```

}

// 为视觉模型每个核心分配指针数组（用于后续设置模型 internal 内存）
rknn3_tensor_mem** internal_mems_vision = (rknn3_tensor_mem**)calloc(core_num_vision,
sizeof(rknn3_tensor_mem));
if (!internal_mems_vision) {
    printf("Failed to allocate memory for internal_mems_vision array\n");
    return -1;
}
// 为 LLM 每个核心分配指针数组（用于后续设置模型 internal 内存）
rknn3_tensor_mem** internal_mems_llm = (rknn3_tensor_mem**)calloc(core_num_llm, sizeof(rknn3_tensor_mem));
if (!internal_mems_llm) {
    printf("Failed to allocate memory for internal_mems_llm array\n");
    return -1;
}

// 分配内存并初始化视觉模型的 internal 内存块
int idx = 0;
for (uint32_t i = 0; i < core_num_vision; i++) {
    internal_mems_vision[i] = rknn3_create_mem(app_ctx->vision.rknn_ctx,
core_mem_sizes_vision[i].internal_size, core_mem_sizes_vision[i].core_id, RKNN3_FLAG_MEMORY_CACHEABLE);
    if (!internal_mems_vision[i]) {
        return -1;
    }
    printf("Created user internal memory for core %d: size=%lu, virt_addr=%p, phys_addr=0x%lx\n",
core_mem_sizes_vision[i].core_id,
internal_mems_vision[i]->size, internal_mems_vision[i]->virt_addr, internal_mems_vision[i]->phys_addr);
    app_ctx->internal_mems[idx++] = internal_mems_vision[i];
}

// 为 LLM 分配 internal 内存（复用或新分配）
for (uint32_t i = 0; i < core_num_llm; i++) {
    if (llm_to_vision[i] != -1) {
        // 复用：LLM 核心与视觉模型核心在同一物理核心上，直接指向已分配的内存
        internal_mems_llm[i] = internal_mems_vision[llm_to_vision[i]];
        continue;
    }
    // 不满足则需要为该 LLM 核心单独分配内存，这里使用 vision.rknn_ctx 分配，是为了后续统一释放（避免重复释放问题）
    internal_mems_llm[i] = rknn3_create_mem(app_ctx->vision.rknn_ctx, core_mem_sizes_llm[i].internal_size,
core_mem_sizes_llm[i].core_id, RKNN3_FLAG_MEMORY_CACHEABLE);
    if (!internal_mems_llm[i]) { // 都使用vision.rknn_ctx分配，方便后面释放
        return -1;
    }
    printf("Created user internal memory for core %d: size=%lu, virt_addr=%p, phys_addr=0x%lx\n",
core_mem_sizes_llm[i].core_id,
internal_mems_llm[i]->size, internal_mems_llm[i]->virt_addr, internal_mems_llm[i]->phys_addr);
    // 将新分配的内存块也加入全局管理数组
    app_ctx->internal_mems[idx++] = internal_mems_vision[i];
}

// 将分配好的 internal 内存设置给视觉模型
ret = rknn3_set_internal_mem(app_ctx->vision.rknn_ctx, internal_mems_vision, core_num_vision);
if (ret < 0) {
    printf("rknn3_set_internal_mem failed! ret=%d\n", ret);
    return ret;
}
// 将分配好的 internal 内存（含复用部分）设置给 LLM 模型
ret = rknn3_set_internal_mem(app_ctx->llm.rknn_ctx, internal_mems_llm, core_num_llm);
if (ret < 0) {
    printf("rknn3_set_internal_mem failed! ret=%d\n", ret);
    return ret;
}

// 释放临时分配的指针数组（internal 内存块本身由 rknn3_create_mem 申请，需后续 rknn3_destroy_mem 释放）
free(internal_mems_vision);
free(internal_mems_llm);
free(core_mem_sizes_vision);
free(core_mem_sizes_llm);

return ret;

```

## 9. 常见问题

### 9.1 NPU环境准备问题

在 RK182X 或配套 NPU 平台进行模型部署与推理时，**rknn3-toolkit**、**Proxy** 与 **Runtime** 版本是否一致直接影响模型导出、量化、推理的兼容性。如果出现以下情况，必须先检查 NPU 相关环境是否匹配：

#### 1. 切换到新的开发机器 / Docker 环境时

#### 2. 更新或重新安装 RKNN-ToolKit3 后

Toolkit 升级后必须确认 rknn3\_transfer\_proxy 和 Runtime 也同步更新，否则可能会出现模型转化正常但设备上跑不起来的问题。

#### 3. 从其他人共享的项目/SDK 拉取代码或模型时

团队成员使用的环境不同步时，最容易出现版本差异导致的推理不兼容。

#### 4. 运行错误或加载失败时

当前部署运行建议RKNN3-Toolkit、rknn3\_transfer\_proxy和Runtime库版本为同一版本，版本查询方式如下：

```
# 查询rknn3-toolkit版本  
pip list | grep rknn3-toolkit  
  
# 查询rknn模型转换时使用的rknn3-toolkit版本  
strings xxxx.rknn | grep rknn3-toolkit  
  
# 查询Runtime 库版本  
strings /usr/lib/librknn3_api.so | grep -i "librknn_api version"  
  
# 查询rknn3_transfer_proxy版本  
/usr/bin/rknn3_transfer_proxy
```

### 9.2 工具安装问题

#### • RKNN3-ToolKit依赖的环境限制太严格，导致无法成功安装

在所有依赖库都已安装、但部分库的版本和要求不匹配时，可以尝试在安装指令后面加上"no-deps"参数，取消安装Python库时的环境检查。如：

```
pip install rknn3-toolkit*.whl --no-deps
```

#### • PyTorch依赖说明

RKNN3-Toolkit的PyTorch模型加载功能，依赖于PyTorch。PyTorch的模型分为浮点模型和已量化模型（包含QAT及PTQ量化模型）。

对于PyTorch 1.6.0导出的模型，建议将RKNN3-Toolkit依赖的PyTorch版本降级至1.6.0以免出现加载失败的问题。

对于已量化模型（QAT、PTQ），我们推荐使用PyTorch 1.10~1.13.1导出模型，并将RKNN3-Toolkit依赖的PyTorch版本升级至1.10~1.13.1。

另外在加载PyTorch模型时，建议导出原模型的PyTorch版本，要与RKNN3-Toolkit依赖的PyTorch版本尽量一致。  
使用推荐的PyTorch版本。

#### • TensorFlow依赖说明

RKNN3-Toolkit的TensorFlow模型加载功能依赖于TensorFlow。由于TensorFlow各版本之间的兼容性一般，其他版本可能会造成RKNN3-Toolkit模型加载异常，所以在加载TensorFlow模型时，建议导出原模型的TensorFlow版本，要与RKNN3-Toolkit依赖的TensorFlow版本一致。

对于TensorFlow版本引发的问题，通常会体现在"rknn.load\_tensorflow()"阶段，且出错信息会指向依赖的TensorFlow路径。

推荐使用的TensorFlow版本为2.6.2或2.8.0。

#### • RKNN3-ToolKit安装包命名规则

以0.5.0版本的发布件为例，RKNN3-Toolkit wheel包命令规则如下：

```
rknn3_toolkit-0.5.0-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl
```

- rknn3\_toolkit: 工具名称。
- 0.5.0: 版本号。
- cp310: 适用的Python版本，例如cp38-cp38表示适用的Python版本是3.8。
- manylinux2014\_x86\_64: 系统类型和CPU架构。

#### • RKNN3-ToolKit是否有ARM Linux版本

RKNN3-Toolkit没有ARM Linux版

## 9.3 模型转换常用参数说明

本章节主要覆盖模型转换阶段常用参数的使用说明。

- 根据模型确定参数

模型转换时，`rknn.config()` 和 `rknn.build()` 接口会影响模型转换结果。

`rknn.load_onnx()`, `rknn.load_tensorflow()` 指定输入输出节点，会影响模型转换结果。

`rknn.load_pytorch()`, `rknn.load_tensorflow()` 指定输入的尺寸大小会影响模型转换结果。

可以参考以下基本步骤进行模型转换：

- 准备量化数据，提供dataset.txt文件。
- 确定模型要使用的NPU平台，如RK1820、RK1828等，并填写 `rknn.config()` 接口中的 `target_platform` 参数。
- 当输入是3通道的图像，且量化数据采用的是图片格式（如jpg、png格式）时，需要确认模型的输入是RGB还是BGR，以决定 `rknn.config()` 接口中 `quant_img_RGB2BGR` 参数的值。
- 确认模型训练时候的归一化参数，以决定 `rknn.config` 接口中的 `mean_values` 和 `std_values` 参数的值。
- 确认模型输入的尺寸信息，填入load接口相应参数中，如 `rknn.load_onnx()` 接口中的 `input_size_list` 参数。
- 确认模型要量化比特数，以决定 `rknn.config()` 接口中的 `quantized_dtype` 参数的值。不对模型进行量化或加载的是已量化模型时可以忽略此步骤。
- 确认模型量化时使用的量化算法，以决定 `rknn.config()` 接口中 `quantized_algorithm` 参数的值。不对模型进行量化或加载已量化模型时可以忽略此步骤。
- 确认是否对模型进行量化，以决定 `rknn.build()` 接口中 `do_quantization` 参数的值。选择对模型进行量化时，需要额外填写 `rknn.build()` 接口中的dataset参数，指定量化正数据。

- RKNN模型的跨平台兼容性

RK1820、RK1828平台使用的模型互相之间是不兼容的。

- 量化校正数据的格式及要求

量化校正数据的格式有两种选择，一种是图片格式（jpg, png），RKNN3-Toolkit会调用OpenCV接口进行读取；另一种是npy格式，RKNN3-Toolkit会调用numpy接口进行读取。对于非RGB/BGR图片输入的模型，建议使用numpy的npy格式提供量化数据。

- 多输入模型dataset.txt文件的填写方式

模型量化需要用dataset.txt文件指定量化数据的路径。规则为一行作为一组输入，模型存在多输入时，多个输入写在同一行，并用空格隔开。

如单输入模型，使用两组量化数据：

```
sampleA.npy  
sampleB.npy
```

如三个输入的模型，两组量化数据按如下方式填写：

```
sampleA_in0.npy sampleA_in1.npy sampleA_in2.npy  
sampleB_in0.npy sampleB_in1.npy sampleB_in2.npy
```

- 确认`rknn.config()`的`quant_img_RGB2BGR`参数

采用图片（jpg, png）作为量化数据时，需要考虑设置`quant_img_RGB2BGR`参数。

模型采用RGB图片进行训练时，则`quant_img_RGB2BGR`参数设为False或不设置。且在使用Python inference接口或RKNPU3 C API进行推理时，输入RGB图片。

模型采用BGR图片进行训练时，则`quant_img_RGB2BGR`参数设为True。但在使用Python inference接口或RKNPU3 C API进行推理时，同样需要输入BGR图片（`quant_img_RGB2BGR`只会影响从量化校正数据集读入的图像）。

若量化数据采用numpy的npy格式，则建议不要使用`quant_img_RGB2BGR`参数，避免产生使用混乱的问题。

- `rknn.config()`的`mean_values`和`std_values`和`quant_img_RGB2BGR`的计算顺序问题

因为`quant_img_RGB2BGR`只控制在量化过程中读取校正集图像时是否要进行转换通道，并不会影响其他的步骤。因此对于RKNN3-Toolkit的inference接口及 RKNPU3 C API，对输入数据都只先进行减均值（`mean_values`）、再除标准差（`std_values`）的操作，并没有通道转换的操作。

- 模型是非3通道输入或多输入时，`rknn.config()`的`mean_values`和`std_values`的设置问题

`mean_values` 和 `std_values` 的设置格式是一致的。以 `mean_values` 为例子。

假设输入有N个通道，则 `mean_values` 的值为 `[[channel_1, channel_2, channel_3, channel_4, ..., channel_n]]`。

存在多输入时，则 `mean_values` 的值为 `[[channel_1, channel_2, channel_3, channel_4, ..., channel_n], [channel_1, channel_2, channel_3, channel_4, ..., channel_n], [channel_1, channel_2, channel_3, channel_4, ..., channel_n]]`，示例如下：

```
rknn.config(mean_values=[[128, 128, 128, 128]],
            std_values=[[1, 1, 1, 1]],
            target_platform='rk1820',
            core_num=1)
```

- 量化参数矫正算法和量化图片数量的选取

RKNN3-Toolkit中量化算法（`rknn.config()` 的 `quantized_algorithm`）参数提供五种算法进行参数矫正，分别为normal，mmse，kl\_divergence，gdq及grq，默认使用normal。normal为常规的量化参数矫正算法；而mmse会迭代中间层的计算结果，对权重数值进行一定范围的裁剪，以获得更高的推理精度。使用mmse不一定能提升量化精度，但相比normal方式，量化时会占用更多的内存、耗费更长的模型转换时间；使用kl\_divergence量化算法所用时间会比normal多一些，但比mmse会少很多，在某些场景下（feature分布不均匀时）可以得到较好的改善效果。gdq量化算法只在w4a16下有效，能有效的提高w4a16的权重量化精度，推荐量化数据集量为200张以上。因对算力要求高，须使用GPU进行加速运算。grq量化算法只在w4a16下有效，也能有效的提高w4a16的权重量化精度，推荐量化数据集量为200张以上。同样因对算力要求高，须使用GPU进行加速运算，但grq相比gdq算法速度更快且显存占用更低，因此可优先尝试grq算法。

建议常规模型先使用normal算法，如果量化效果不佳，可尝试使用mmse或kl\_divergence算法。使用normal或kl\_divergence算法时，推荐给出20-200组数据进行量化。使用mmse量化时，推荐使用20-50组数据进行量化。

- 关于LLM/VLM/VIT模型量化

建议LLM/VLM/VIT模型进行量化时 `rknn.config()` 参数采用 `quantized_dtype='w4a16'`, `quantized_method='group32'` 进行量化  
建议LLM模型优先尝试grq算法（`rknn.config()` 的 `quantized_algorithm`）。

- 量化模型与非量化模型，推理时输入输出的差异

调用通用RKNPU3 C API时（输入数据的数据类型（如uint8数据，float数据）与模型的量化与否没有关系。输出数据的数据类型可以选择自动  
处理成float16格式，也可以选择直接输出模型推理结果，此时数据类型与输出节点的数据类型一致。

## 9.4 模型加载问题

### 9.4.1 ONNX模型转换常见问题

- 加载模型时出现“Error parsing message”报错

转换ONNX模型时，提示加载失败：

```
E load_onnx: Catch exception when loading onnx model: /rknn_resnet_demo/resnet50v2.onnx!
E lod_onnx: Traceback (most recent call last):
E load_onnx: File "rknn/api/rknn_base.py", line 1094, in rknn.api.rknn_base.RKNNBase.load_onnx
E load_onnx: File "/usr/local/lib/python3.6/dist-packages/onnx/__init__.py", line 115, in load_model
...
E load_onnx: google.protobuf.message.DecoderError: Error parsing message
```

原因可能是ONNX模型损坏导致（如没下载全），需要重新下载该模型，并确保其MD5值正确

- 是否支持动态的输入shape

可以通过 `rknn.config()` 的 `dynamic_input` 参数进行动态输入shape的仿真

- 自定义输出节点时报错

`rknn.load_onnx()` 时传入 `outputs` 参数进行模型的裁剪，但报如下错误：

```
E load_onnx: the '378' in outputs=['378', '439', '500'] is invalid!
```

日志提示输出节点378是无效的，因此outputs参数需设置正确的输出节点名称。

### 9.4.2 Pytorch模型转换常见问题

- 加载Pytorch模型时出现`torch._C`没有`_jit_pass_inline`属性的错误

错误日志如下：

```
'torch._C' has no attribute '_jit_pass_inline'
```

请将PyTorch升级到1.6.0或之后的版本。

- Pytorch模型的保存格式

目前只支持 `torch.jit.trace()` 导出的模型。`torch.save()` 接口仅保存权重参数字典，缺乏网络结构信息，无法被正常导入并转成RKNN模型。

- 转换时遇到PytorchStreamReader失败的错误

详细错误如下：

```
E Catch exception when loading pytorch model: ./mobilenet0.25_Final.pth!
E Traceback (most recent call last):
.....
E cpp_module = torch._C.import_ir_module(cu, f, map_location, extra_files)
E RuntimeError: [enforce fail at inline container.cc:137]. PytorchStreamReader failed reading zip archive:
failed
finding central directory frame .....
```

出错原因是输入的PyTorch模型没有网络结构信息。通常pth只有权重，并没有网络结构信息。对于已保存的模型权重文件，可以通过初始化对应的网络

结构，再使用net.load\_state\_dict()加载pth权重文件。最后通过torch.jit.trace()接口将网络结构和权重参数固化成一个pt文件。得到torch.jit.trace()处理过以后的pt文件，就可以用rknn.load\_pytorch()接口将其转为RKNN模型。

- 转换时遇到**KeyError**的错误

错误日志如下：

```
E Traceback (most recent call last):
.....
E KeyError: 'aten::softmax'
```

出现形如KeyError: 'aten::xxx'的错误信息时，表示该算子当前版本还不支持。RKNN3-Toolkit在每次版本升级时都会修复此类bug，请使用最新版本的RKNN3-Toolkit试试。

- 转换时遇到**"Syntax error in input! LexToken(xxx)"**的错误

错误日志如下：

```
WARNING: Token 'COMMENT' defined, but not used
WARNING: There is 1 unused token
!!!!!! Illegal character !!!
Syntax error in input! LexToken(NAMED_IDENTIFIER, 'fc', 1, 27)
!!!!!! Illegal character !!!
```

该错误的原因有很多种，请按照以下顺序排查：

- 1) 未继承torch.nn.module创建网络。请继承torch.nn.module基类来创建网络，然后再用torch.jit.trace()生成pt文件。
- 2) 更新RKNN3-Toolkit版本，torch建议使用1.6.0, 1.9.0, 1.10.0或1.13.1版本。

### 9.4.3 TensorFlow模型转换常见问题

- **Tensorflow1.x模型报错**

使用rknn.load\_tensorflow()接口加载tensorflow1.x模型如出现报错提示：

```
E load_tensorflow: Catch exception when loading tensorflow model: ./yolov3_mobilenetv2.pb!
E load_tensorflow: Traceback (most recent call last):
.....
E load_tensorflow: tensorflow.python.framework.errors_impl.InvalidArgumentError: Node
'MobilenetV2/expanded_conv/depthwise/BatchNorm/cond/Switch_1' expects to be colocated with unknown node
'MobilenetV2/expanded_conv/depthwise/BatchNorm/moving_mean'
E load_tensorflow: During handling of the above exception, another exception occurred:
E load_tensorflow: Traceback (most recent call last):
E load_tensorflow: File "rknn/api/rknn_base.py", line 990, in rknn.api.rknn_base.RKNNBase.load_tensorflow
.....
E load_tensorflow: return func(*args, **kwargs)
E load_tensorflow: File "/usr/local/lib/python3.6/dist-packages/tensorflow/python/framework/importer.py",
line
431, in import_graph_def
E load_tensorflow: raise ValueError(str(e))
E load_tensorflow: ValueError: Node 'MobilenetV2/expanded_conv/depthwise/BatchNorm/cond/Switch_1' expects
to be colocated with unknown node 'MobilenetV2/expanded_conv/depthwise/BatchNorm/moving_mean'
```

建议：

- 如当前安装的是1.x的TensorFlow，请安装2.x的TensorFlow。
- 更新RKNN3-Toolkit / RKNPU3至最新版本。

- **TransformGraph类似的报错**

TensorFlow的模型转成RKNN时报错：

```

Traceback (most recent call last):
File "test.py", line 80, in <module>
    input_size_list=[[1, 368, 368, 3]])
File "/usr/local/lib/python3.6/site-packages/rknn/api/rknn.py", line 68, in load_tensorflow
    input_size_list=input_size_list, outputs=outputs)
File "rknn/api/rknn_base.py", line 940, in rknn.api.rknn_base.RKNNBase.load_tensorflow
File "/usr/local/lib/python3.6/dist-packages/tensorflow/tools/graph_transforms/_init__", line 51, in
    TransformGraph.transforms_string, status)
File "/usr/local/lib/python3.6/dist-packages/tensorflow/python/framework/errors_impl.py". ;ome 548, in
    __exit__
C_api.TF_GetCode(self.status.status)
Tensorflow.python.framework.error_impl.InvalidArgumentError: Beta input to batch norm has bad shape: [24]

```

原因：

- 1) 该模型直接调用TensorFlow原生的TransformGraph类进行优化时，也会报上面的错误（RKNN3-Toolkit里同样会调用TransformGraph进行优化，因此也会报同样的错误）。
- 2) 可能是模型生成时的TensorFlow版本与目前安装的版本已经不兼容了。建议：使用1.14.0的TensorFlow版本重新生成该模型，或者寻找其他框架的同类型模型。

- "Shape must be rank 4 but is rank 0"报错

加载pb模型时：

```

rknn.load_tensorflow(tf_pb='./model.pb',
    inputs=['X', 'Y'],
    outputs=['generator/xs'],
    input_size_list=1, INPUT_SIZE, INPUT_SIZE, 3)

```

会产生报错：

```

E load_tensorflow: Catch exception when loading tensorflow model: ./model.pb!
E load_tensorflow: Traceback (most recent call last):
E load_tensorflow: File "/usr/local/lib/python3.6/dist-packages/tensorflow/python/framework/importer.py",
line
427, in import_graph_def
E load_tensorflow: graph._c_graph, serialized, options) # pylint: disable=protected-access
E load_tensorflow: tensorflow.python.framework.errors_impl.InvalidArgumentError: Shape must be rank 4 but
is
rank 0 for 'generator/conv2d_3/Conv2D' (op: 'Conv2D') with input shapes: [], [7,7,3,32].

```

原因可能是该模型是多输入模型，`rknn.load_tensorflow()` 的 `input_size_list` 没按规范填写。可以参考：

```

rknn.load_tensorflow(tf_pb='./conv_128.pb',
    inputs=['input1', 'input2', 'input3', 'input4'],
    outputs=['output'],
    input_size_list=[[1, 128, 128, 3], [1, 128, 128, 3],
    [1, 128, 128, 3], [1, 128, 128, 1]])

```

#### 9.4.4 其它

- 加载模型出错时的排查步骤

首先确认原始深度学习框架是否可以加载该模型并进行正确的推理，检查原始模型是否有问题。

其次请将RKNN3-Toolkit升级到最新版本。如果模型有RKNN3-Toolkit不支持的层（或OP），通过打开调试日志开关，在日志中可以看到哪一个算子是RKNN3-Toolkit不支持的。

如果仍无法解决，请将使用的RKNN3-Toolkit版本和详细的错误日志反馈给瑞芯微NPU开发团队。

### 9.5 模型量化问题

- 量化对模型体积的影响

分两种情况，当导入的模型是量化的模型时，`rknn.build()` 接口的 `do_quantization=False` 会使用该模型里面的量化参数。当导入的模型是浮点的模型时，`do_quantization=False` 不会做量化的操作，但是会把权重从FP32转成FP16，这块几乎不会有精度损失。这两种情况都减少了模型权重的体积，从而使得整个模型占用空间变小。

- 模型量化时，图片是否需要和模型输入的尺寸一致

不需要。RKNN3-Toolkit会自动对这些图片进行缩放处理。但是缩放操作也可能使图片信息发生改变，对量化精度产生一定影响，所以最好使用尺寸相近的图片。如果是非图像格式的校正数据，如numpy格式，则需要与模型输入的shape一致。

- 量化校正数据集是否需要根据`rknn_batch_size`参数进行修改

不需要。

`rknn.build()` 的 `rknn_batch_size` 参数只会修改最后导出的RKNN模型的batch维（由1改为`rknn_batch_size`），并不会影响量化阶段的流程，因此量化校正数据集还是按照 batch 为1的方式来设置即可。

- **模型量化时，程序运行一段时间后被kill掉或程序卡住**

在模型量化过程中，RKNN3-Toolkit会申请较多的系统内存，有可能造成程序被kill掉或卡住。

解决方法：增加电脑内存或增大虚拟内存（交换分区）。

## 9.6 模型转换问题

- **常见转换bug报错的问题**

尝试将RKNN3-Toolkit更新至最新版本。

- **怎么判断算子RKNN是否支持**

直接进行模型的转换，如果不支持会有相关提示，也可参考官方文档。

- **转换时提示算子不支持**

建议：

- 1) 可尝试更新RKNN3-Toolkit / RKNPU3至最新版本。
- 2) 修改模型，采用等效算子来替代。

- **rknn.config()的mean\_values报错提示**

设置 mean/std 为：

```
rknn.config(mean_values=[[128, 128, 128]], std_values=[[128, 128, 128]])
```

时转换模型报错：

```
--> Loading model
transpose_input for input_1: shape must be rank 4, ignored
E load_tflite: The len of mean_values ([[128, 128, 128]]) for input 0 is wrong, expect 32!
```

原因可能是模型的输入不是3通道图像数据（例如输入shape是1x32，非图像数据），此时：

- 需要根据输入通道个数来设置 `mean_values / std_values`。
  - 如果模型不需要指定 `mean/std`，`rknn.config()` 可以不设置 `mean_values / std_values`（`mean/std`一般只对图像输入有效）。
- **模型存在4维以上Op时报错（如5维或6维）**

RKNN目前暂不支持4维以上的OP，可以手工将这些节点去掉。

- **"Not support input data type 'float16'"报错**

目前RKNN3-Toolkit还不支持float16的权重类型的Pytorch模型，需改为float32。

- **动态图相关报错**

转换模型时，如果出现以下类似报错：

```
E build: ValueError: The Op of 'NonZero' is not support! it will cause the graph to be a dynamic graph!
```

说明包含该OP的模型为动态图，需要手动修改模型，用其他OP替换或将其移除。

- **RKNN模型大小问题**

模型转换结束后，可能存在转换出来的RKNN模型比原始模型大的现象，甚至跟模型的输入shape也有关系，这种现象是正常的。因为RKNN模型里不仅仅包含权重和图结构信息，还会有很多NPU的寄存器配置信息，并且为了提高运行效率，可能也会做OP的拆解等操作，这些都会导致RKNN模型变大。

## 9.7 模拟器推理及连板推理的说明

- **术语说明**

模拟器推理：RKNN3-Toolkit 在 Linux x86\_64 平台提供模拟器功能，可以在没有开发板的情况下进行模型推理，获取推理结果。（该功能输出结果未必与连板或板端一致，更推荐使用连板推理或板端推理）。

连板推理：指在开发板已连接电脑的情况下，调用 RKNN3-Toolkit 的 Python API 推理模型，获取推理结果。

板端推理：指在开发板上调用 RKNPU3 的 C API 接口推理模型，获取推理结果。

- **模拟器推理结果与连板推理结果不一致**

发生此情况时，可能意味着板端的结果不正确。

由于硬件和驱动的差异，模拟器不保证可以和板端获取一模一样的结果。但如果差异实在太大，可以将问题反馈给RK的NPU团队进行分析。

- **连板推理的工作原理**

使用连板推理时，RKNN3-Toolkit会与板端进行通信，通信时会将模型、模型的输入由PC端传至板端，随后调用RKNPU3 C API进行模型推理，板端推理完成后将结果回传至PC端。

- **连板推理与板端推理结果有差异**

连板推理是基于RKNPU3 C API实现的，理论上连板推理结果会与RKNPU3 C API推理结果一致。当这两者出现较大差异时，请确认输入的预处理、数据类型、数据的排布方式（NCHW，NHWC）是否有差异。

需指出，如差异很小且发生在小数点后3位及之后的数值上，则属于正常现象。差异可能产生在使用不同的库读取图片、转换数据类型等步骤上。

- **板端推理的速度比连板推理更快**

由于连板推理存在额外的数据拷贝、传输过程，会导致连板推理的性能不如板端的RKNPU3 C API 推理性能。因此，NPU实际推理性能以RKNPU3 C API的推理性能为准。

- **涉及连板调试、连板推理功能时，获取详细的错误日志**

连板推理时，模型的初始化、推理等操作主要在开发板上完成，此时日志信息主要产生在板端上。为了获取具体的板端调试信息，可以通过串口进入开发板操作系统。然后执行以下两条命令设置获取日志的环境变量。保持串口窗口不要关闭，再进行连板调试，此时板端的错误信息就会显示在串口窗口上：

```
export RKNN_LOG_LEVEL=5
```

- **连板推理数据的排布方式说明**

`rknn.inference` 在连板推理时输入数据的排布方式可选择为NCHW或NHWC，即 `data_format='NCHW'` 或 `data_format='NHWC'`，默认为 NHWC。因为 RKNN3-Toolkit 中在转换时ONNX模型的4维输入默认都是 NCHW的格式，在使用`rknn`进行`inference`时输入排布方式默认为NHWC，内部会进行对应的格式转换，如果在`inference`中设置的 `data_format='NCHW'`，则输入数据不进行默认的NCHW到NHWC的转换。

## 9.8 模型评估常见问题

- **量化模型精度不及预期**

参考本文档的第6章节。

- **支持哪些框架的已量化模型**

支持TensorFlow、TensorFlow Lite和PyTorch框架的已量化模型。

- **Runtime出现"Invalid RKNN format"报错**

Runtime上出现以下报错：

```
Loading model ...
E RKNN: [06:28:39.048] parseRKNN from buffer: Invalid RKNN format!
E RKNN: [06:28:39.049] rknn_init, load model failed!
rknn_init error ret=-1
```

原因：

- 1) 可能是模型转换时的 `rknn.config()` 的 `target_platform` 没有设置对，或没有设置。
- 2) Runtime版本与RKNN3-Toolkit不兼容。

建议：

- 1) 设置正确的 `target_platform`。
- 2) RKNN3-Toolkit与Runtime要一起更新到同一个版本。

- **rknn.inference()耗时与rknn.eval\_perf()理论速度不一致**

因为 `rknn.inference()` 使用PC + adb的方式进行连板推理，存在着一些固定的数据传输开销，因此与 `rknn.eval_perf()` 理论速度不一致。对于更真实的帧率，建议直接在开发板上使用RKNPU3 C API进行测试。

- **rknn.inference()对多batch的支持**

可以在构建RKNN模型时就指定输入图片的数量，详细用法参考RKNN3-Toolkit API手册中关于 `rknn.build` 接口的说明。另外，当 `rknn_batch_size` 大于1（如等于4时），Python里推理的调用要由：

```
outputs = rknn.inference(inputs=[img])
```

修改为：

```
img = np.expand_dims(img, 0)
img = np.concatenate((img, img, img, img), axis=0)
outputs = rknn.inference(inputs=[img])
```

- **运行多个RKNN模型**

运行两个或多个模型时，需要创建多个RKNN对象。一个RKNN对象对应一个模型，类似一个上下文。每个模型在各自的上下文里初始化模型，推理，获取推理结果，互不干涉。这些模型在NPU上 推理时是串行进行的。

- **模型推理的耗时非常长，而且得到的结果错误**

如果推理耗时超过20s，且结果错误，这通常是NPU出现了NPU Hang的BUG。如果遇到该问题，可以尝试更新RKNN3-Toolkit / RKNPU3到最新版本。

- **模型输入为3维情况下，连板推理结果错误**

模型的输入为3维情况下，如出现Simulator的仿真结果正确，但连板推理结果错误的情况。

原因可能是当前NPU的输入3维支持还不完善，后面会完善3维的支持。

建议：

- 1) 先将模型输入改为4维。
- 1) 更新RKNN3-Toolkit / RKNPU3至最新版本进行尝试。

- **连板推理结果错误，并且每次都不一致**

ONNX模型转RKNN后，用Simulator的仿真结果正确，并且每次结果都一致。但在连板推理时结果 错误，并且每次都不一致。这种问题可能是板端NPU内核驱动bug导致，此时需要更新板端的NPU内 核驱动，并且需要一并更新最新的RKNN3-Toolkit / RKNPU3。

- **模型存在较多的Resize OP时，出现精度下降问题**

当ONNX模型里存在较多的Resize OP时，转换为RKNN后出现精度下降。可能的原因是：

- 1) 精度下降是因为NPU目前还不支持硬件级别 `Resize`（后续会支持），转换工具会将 `Resize` 转为 `ConvTranspose`，会导致一点点的精度丢失。
- 2) 如模型有多个串联的 `Resize`，则可能会累积了太多误差导致精度下降比较多。

建议：

- 1) 目前尽量避免 `Resize` 的使用（如将 `Resize` 改为 `ConvTranspose` 再进行训练）
- 2) 可以在 `rknn.config()` 里加入 `optimization_level=2` 的参数，此时 `Resize` Op会走CPU，精度不会掉，但会导致性能下降。

- **do\_quantization设为False以后推理结果都为nan**

`rknn.build()` 接口中的 `do_quantization` 设为 `True` 时推理结果没有异常，但设为 `False` 以后推理结果就都变 为 `nan` 了。原因可能是 `do_quantization=False` 时，RKNN模型的运算类型是fp16的，但该模型的中间层（如卷积）输出的范围可能超出了fp16 (65536) 的范围（如-51597~75642）。

建议：训练的时候需要保证中间层的输出不超过fp16的表达范围（一般通过添加BN层来解决该问题）。

- **QAT模型与RKNN模型结果不一致**

在Pytorch框架下使用QAT训练了一个分类模型并转为RKNN模型，对该模型使用Pytorch和RKNN分 别进行推理，发现得到的结果不一样，原因可能是Pytorch的推理没有设置 `engine='qnnpack'`，因为 RKNN的推理方式与 qnnpack 更为贴近。

- **怎么获取模型运行时候内存占用率**

可以使用 `rknn.eval_memory()` 接口，输出的日志里有个Total项，就是总的占用大小。

- **性能评估时，开启或关闭`rknn.init_runtime()`的`perf_debug`参数，性能数据的差异**

开启 `perf_debug` 时，为了收集每层的信息，会添加一些调试代码，并且可能禁用一些并行的机制，因此耗时比 `perf_debug=False` 时多一些。开启 `perf_debug` 的主要作用是看模型中是否有耗时占比比较多的层，以此为依据来设计优化方案。

- **环境用的docker，之前连板推理正常，重启docker后，推理时卡在初始化环境阶段？**

因为docker重启时 `rknn3_transfer_proxy` 类似于异常退出的状态，导致开发板上的RKNN Server无法检测 到上端连接已经断开，这时需要重启下开发板，重置RKNN Server的连接状态。

## 9.9 C API使用常见问题

- **`rknn3_create_mem`如何创建合适的大小的内存？**

对于输入而言，一般原则是：`rknn3_create_mem()` 使用 `rknn3_tensor_attr` 的 `aligned_size` 和 `core_id` 进行分配内存

- **输入数据如何填充？**

对于四维形状输入，`fmt=NHWC`，即数据填充顺序为 `[batch, height, width, channel]`。非四维输入形状，`fmt=UNDEFINED`，按照模型的原始形状填充数据。

- **出现"Meet unsupport xxx operator"错误如何处理？**

在板端运行demo出现类似的报错时，一般是板端的 Runtime (`librknn3_api.so`) 不支持该算子。建议用户 先更新RKNN 相关工具链到最新版本，再重新转换模型，并在板端重跑demo，或者通过redmine上报给RKNN 团队。

- 模型加载出现，类似 **core\_mask ff is not match with npu core number 1!**

出现这种错误是由于在调用 `rknn3_model_init` 时传入的参数 `run_core_mask` 与模型转换时配置的 `core_num` 不一致导致的，按正确的配置运算核心数即可。