密级状态：绝密(　　)　　秘密(　　)　　内部(　　)　　公开(√)

RKNN API For RKNPU User Guide

（技术部，图形计算平台中心）

文件状态： [　] 正在修改 [√] 正式发布	当前版本：	1.3.0
	作　　者：	HPC/NPU 团队
	完成日期：	2022-05-13
	审　　核：	熊伟
	完成日期：	2022-05-13

瑞芯微电子股份有限公司

Rockchips　Semiconductor　Co . , Ltd

更新记录

版本	修改人	修改日期	修改说明	核定人
v0.6.0	HPC 团队	2021-03-01	初始版本	熊伟
v0.7.0	HPC 团队	2021-04-22	删除输入通道转换流程说明	熊伟
v1.0.0	HPC 团队	2021-04-30	正式发布版本	熊伟
v1.1.0	HPC 团队	2021-08-13	1. 增加 rknn_tensor_mem_flags 标志 2. 增加输入/输出 tensor 原生属性的查询命令 3. 增加 NC1HWC2 的内存布局	熊伟
v1.2.0b1	NPU 团队	2021-12-04	1. 增加 RK3588/RK3588s 平台说明 2. 增加 rknn_set_core_mask 接口 3. 增加 rknn_dup_context 接口 4. 增加输入输出 API 详细说明	熊伟
v1.2.0	HPC 团队	2022-01-14	1. 增加关键字说明 2. 增加 NPU SDK 目录和编译说明 3. 增加调试方法章节 4. 增加 NATIVE_LAYOUT 中 C2 取值说明	熊伟
v1.3.0	NPU/HPC 团队	2022-05-13	1. 修复命名 destroy 变为 destroy 2. 增加 RV1106/RV1103 的使用说明 3. 增加 NATIVE_LAYOUT 的细节说明 4. 增加 C API 硬件平台支持说明 5. 增加 NPU 版本、利用率查询以及 NPU 电源手动开关的指令	熊伟

目 录

1 主要功能说明

RKNN SDK 为带有 RKNPU 的芯片平台提供编程接口，能够帮助用户部署使用 RKNN-Toolkit2 导出的 RKNN 模型，加速 AI 应用的落地。

2 硬件平台

本文档适用如下硬件平台：

RK3566、RK3568、RK3588、RK3588S、RV1103、RV1106

注：文档部分地方使用 RK356X 来统一表示 RK3566/RK3568，使用 RK3588 来统一表示 RK3588/RK3588S。

3 关键字说明

RKNN 模型：指运行在 RKNPU 上的文件，后缀名为.rknn。

连板推理：指通过 USB 口连接 PC 和开发板，调用 RKNN-Toolkit2 的接口运行模型。模型实际运行在开发板的 NPU 上。

HIDL：用于指定 Android HAL 和其用户之间的接口的一种接口描述语言。

CTS：全名兼容性测试套件，是谷歌提供的一个 Andorid 平台自动化测试套件。

VTS：全名供应商测试套件，是谷歌提供的一个 Andorid 平台自动化测试套件。

DRM：英文全名 Direct Rendering Manager，是一个主流的图形显示框架。

NATIVE_LAYOUT：指对于 NPU 运行时而言，通常性能表现最佳的计算机内存排列格式。

tensor：张量，通常是指维度大于等于四维的数据。

fd：文件描述符，被用来标识一块内存空间。

i8 模型：量化的 RKNN 模型，即以 8 位有符号整型数据运行的模型。

fp16 模型：非量化的 RKNN 模型，即以 16 位半精度浮点型数据运行的模型。

4 使用说明

4.1 RKNN SDK 开发流程

在使用 RKNN SDK 之前，用户首先需要使用 RKNN-Toolkit2 工具将用户的模型转换为 RKNN 模型。

得到 RKNN 模型文件之后，用户可以选择使用 C 接口在 RK356X/RK3588/RV1106/RV1103 平台开发应用，后续章节将说明如何在这些平台上基于 RKNN SDK 进行开发。

4.2 RKNN Linux 平台开发说明

4.2.1 Linux 平台 RKNN API 库

对于 RK356X/RK3588，SDK 库文件为<sdk>/rknpu2/runtime 下的 librknnrt.so；对于 RV1106/RV1103，SDK 库文件为<sdk>/rknpu2/runtime 下的 librknnmrt.so

4.2.2 EXAMPLE 使用说明

SDK 提供了 Linux 平台的 MobileNet 图像分类、SSD 目标检测、YOLOv5 目标检测示例。这些 Demo 能够为客户基于 RKNN SDK 开发自己的 AI 应用提供参考。Demo 代码位于<sdk>/rknpu2/examples 目录。下面以 RK356X 的 rknn_mobilenet_demo 为例，来讲解如何快速上手运行。

1） 编译 Demo

```
cd examples/rknn_mobilenet_demo
#设置 build-linux.sh 下的 GCC_COMPILER 为正确的编译器路径
./build-linux_RK356X.sh
```

2） 部署到 RK356X 设备

```
adb push install/rknn_mobilenet_demo_Linux /userdata/
```

3） 运行 Demo

```
adb shell
cd /userdata/rknn_mobilenet_demo_Linux/
export LD_LIBRARY_PATH=./lib
./rknn_mobilenet_demo model/RK356X/mobilenet_v1.rknn model/dog_224x224.jpg
```

4.3 RKNN ANDROID 平台开发说明

4.3.1 ANDROID 平台 RKNN API 库

Android 平台有两种方式来调用 RKNN API

1） 应用直接链接 librknnrt.so

2） 应用链接 Android 平台 HIDL 实现的 librknn_api_android.so

对于需要通过 CTS/VTS 测试的 Android 设备可以使用基于 Android 平台 HIDL 实现的 RKNN API。如果不需要通过 CTS/VTS 测试的设备建议直接链接使用 librknnrt.so，对各个接口调用流程的链路更短，可以提供更好的性能。

对于使用 Android HIDL 实现的 RKNN API 的代码位于 RK356x Android 系统 SDK 的 vendor/rockchip/hardware/interfaces/neuralnetworks 目录下。当完成 Android 系统编译后，将会生成一些 NPU 相关的库（对于应用只需要链接使用 librknn_api_android.so 即可），如下所示：

```
/system/lib/librknn_api_android.so
/system/lib/librknnhal_bridge.rockchip.so
/system/lib64/librknn_api_android.so
/system/lib64/librknnhal_bridge.rockchip.so
/vendor/lib64/rockchip.hardware.neuralnetworks@1.0.so
/vendor/lib64/rockchip.hardware.neuralnetworks@1.0-adapter-helper.so
/vendor/lib64/librknnrt.so
/vendor/lib64/hw/rockchip.hardware.neuralnetworks@1.0-impl.so
```

也可以使用如下命令单独重新编译生成以上的库

```
mmm vendor/rockchip/hardware/interfaces/neuralnetworks/ -j8
```

4.3.2 EXAMPLE 使用说明

目前 SDK 提供了 MobileNet 图像分类、SSD 目标检测、YOLOv5 目标检测示例。Demo 代码位于<sdk>/rknpu2/examples 目录。用户可以使用 NDK 编译 Android 命令行中执行的 demo。下面以 RK356X 下的 rknn_mobilenet_demo 为例来讲解在 Android 平台上该 demo 如何使用：

1）编译 Demo

```
cd examples/rknn_mobilenet_demo
#设置 build-android_RK356X.sh 下的 ANDROID_NDK_PATH 为正确的 NDK 路径
./build-android_RK356X.sh
```

2）部署到 RK356X 设备

```
adb push install/rknn_mobilenet_demo_Android /data/
```

3）运行 Demo

```
adb shell
cd /data/rknn_mobilenet_demo_Android/
export LD_LIBRARY_PATH=./lib
./rknn_mobilenet_demo model/RK356X/mobilenet_v1.rknn model/dog_224x224.jpg
```

以上 Demo 默认使用 librknnrt.so，如果开发者需要使用 librknn_api_android.so，可以将对应 Demo 下的 CMakeLists.txt 中链接 librknnrt.so 的地方修改为 librknn_api_android.so 即可，再按上述步骤编译运行。

4.4 RKNN C API

4.4.1 API 硬件平台支持说明

（1）RKNN C API 不同硬件平台支持如下：

	RKNN C API	RK356X	RK3588	RV1106/RV1103
1	rknn_init	√	√	√
2	rknn_set_core_mask	×	√	×
3	rknn_dup_context	√	√	×
4	rknn_destroy	√	√	√
5	rknn_query	√	√	√
6	rknn_inputs_set	√	√	×
7	rknn_run	√	√	√
8	rknn_wait	×	×	×
9	rknn_outputs_get	√	√	×
10	rknn_outputs_release	√	√	√
11	rknn_create_mem_from_mb_blk	×	×	×
12	rknn_create_mem_from_phys	√	√	×
13	rknn_create_mem_from_fd	√	√	×
14	rknn_create_mem	√	√	√
15	rknn_destroy_mem	√	√	√
16	rknn_set_weight_mem	√	√	×
17	rknn_set_internal_mem	√	√	×
18	rknn_set_io_mem	√	√	√

更详细的 RKNN C API 使用说明，请查看 4.4.3 API 详细说明章节

（2）rknn_query 函数不同平台支持的查询参数如下：

	rknn_query params	RK356X	RK3588	RV1106/RV1103
1	RKNN_QUERY_IN_OUT_NUM	√	√	√
2	RKNN_QUERY_INPUT_ATTR	√	√	√
3	RKNN_QUERY_OUTPUT_ATTR	√	√	√
4	RKNN_QUERY_PERF_DETAIL	√	√	×
5	RKNN_QUERY_PERF_RUN	√	√	×
6	RKNN_QUERY_SDK_VERSION	√	√	√
7	RKNN_QUERY_MEM_SIZE	√	√	×
8	RKNN_QUERY_CUSTOM_STRING	√	√	√
9	RKNN_QUERY_NATIVE_INPUT_ATTR	√	√	√
10	RKNN_QUERY_NATIVE_OUTPUT_ATTR	√	√	√

4.4.2 API 流程说明

目前在 RK356X/RK3588 上有两组 API 可以使用，分别是通用 API 接口和零拷贝流程的 API 接口，**RV1106/RV1103 只支持零拷贝流程的 API 接口**。两组 API 的主要区别在于，通用接口每次更新帧数据，需要将外部模块分配的数据拷贝到 NPU 运行时的输入内存，而零拷贝流程的接口会直接使用预先分配的内存（包括 NPU 运行时创建的或外部其他框架创建的，比如 DRM 框架），减少了内存拷贝的花销。当用户输入数据只有虚拟地址时，只能使用通用 API 接口；当用户输入数据有物理地址或 fd 时，两组接口都可以使用。

对于通用 API 接口，首先初始化 rknn_input 结构体，帧数据包含在该结构体中，使用 rknn_inputs_set 函数设置模型输入，等待推理结束后，使用 rknn_outputs_get 函数获取推理的输出，进行后处理。在每次推理前，更新帧数据。通用 API 调用流程如图 3-1 所示，黄色字体流程表示用户行为。

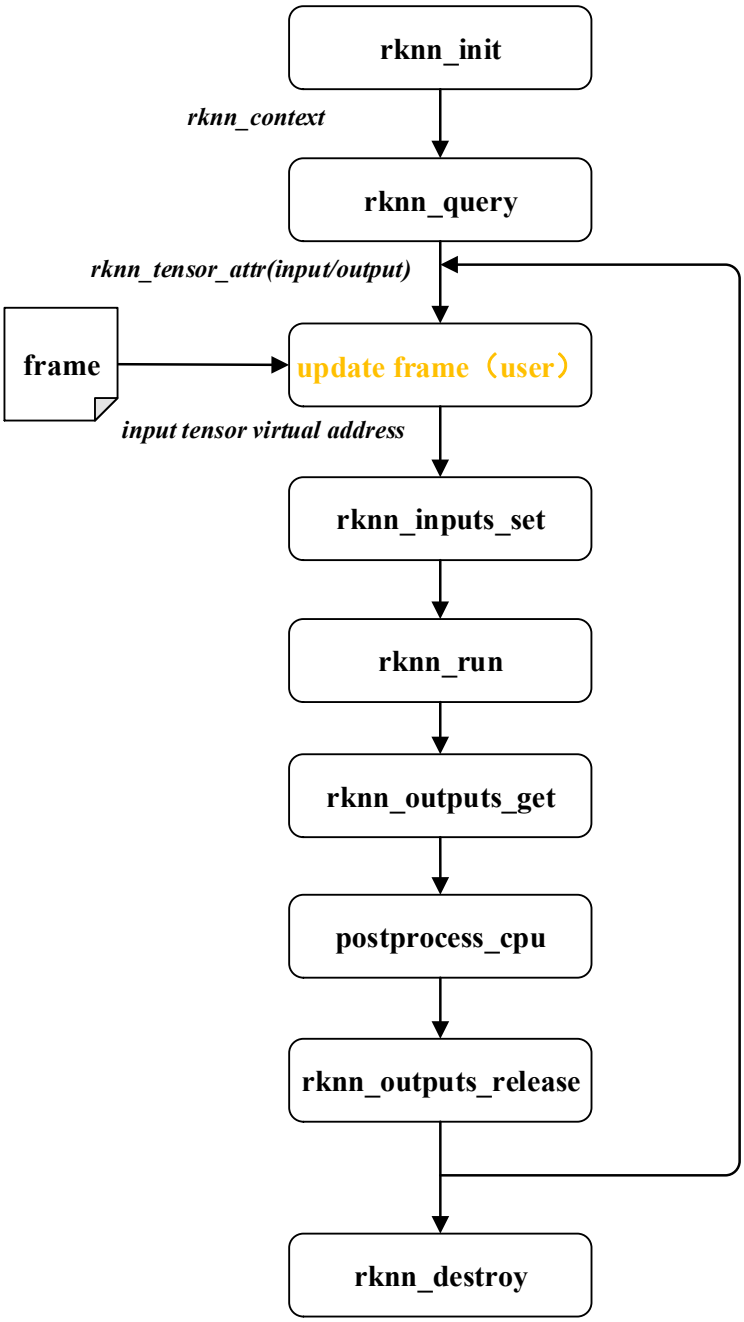

图 3-1 通用 API 接口调用流程

对于零拷贝 API 接口，在分配内存后使用内存信息初始化 rknn_tensor_memory 结构体，在推理前创建并设置该结构体，并在推理后读取该结构体中的内存信息。根据用户是否需要自行分配模型的模块内存（输入/输出/权重/中间结果）和内存表示方式（文件描述符/物理地址等）差异，有下列三种典型的零拷贝调用流程，如图 3-2 至图 3-4 所示，红色部分表示专为零拷贝加入的接口和数据结构，斜体表示接口调用之间传递的数据结构。**注意：目前 RV1106/RV1103 只支持如下方式的零拷贝实现。**

1）输入/输出内存由运行时分配

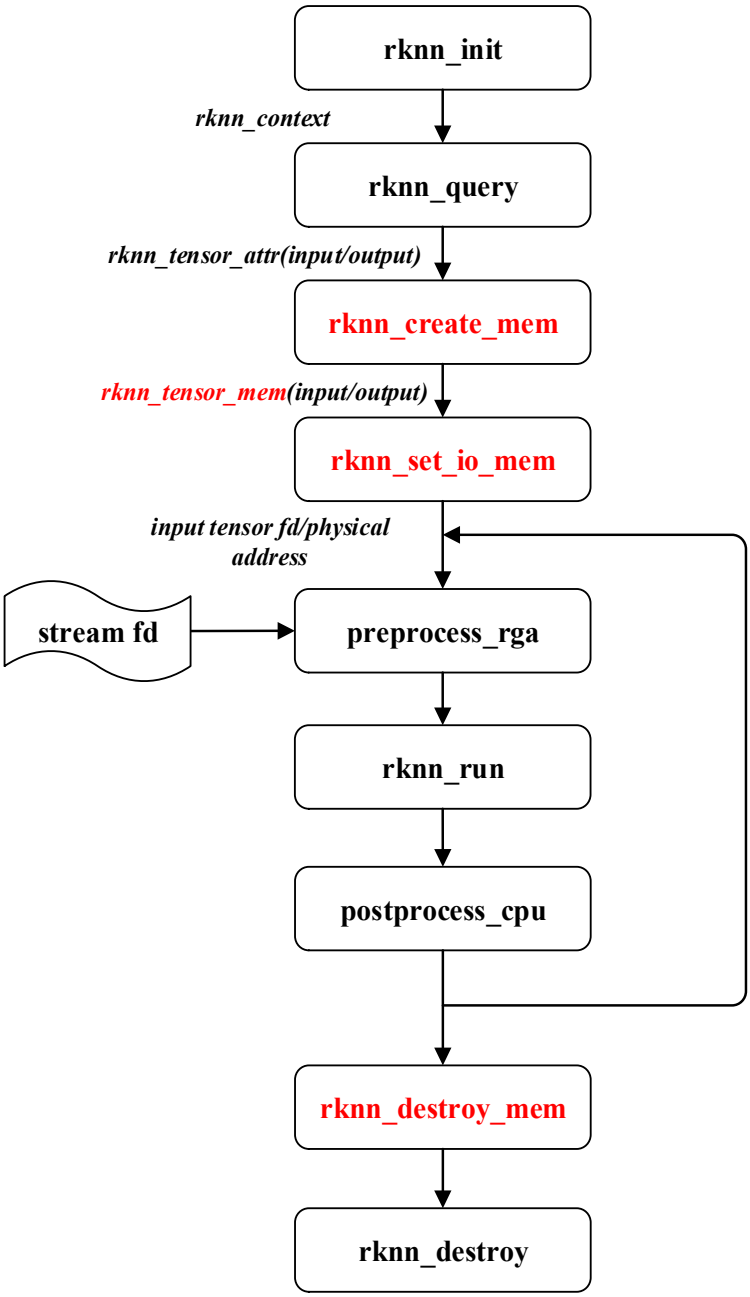

图 3-2 零拷贝 API 接口调用流程（输入/输出内部分配）

如图 3-2 所示，rknn_create_mem 接口创建的输入/输出内存信息结构体包含了文件描述符成员和物理地址，RGA 的接口使用到 NPU 分配的内存信息， preprocess_rga 表示 RGA 的接口，stream_fd 表示 RGA 的接口输入源的内存数据，postprocess_cpu 表示后处理的 CPU 实现。

2）输入/输出内存由外部分配

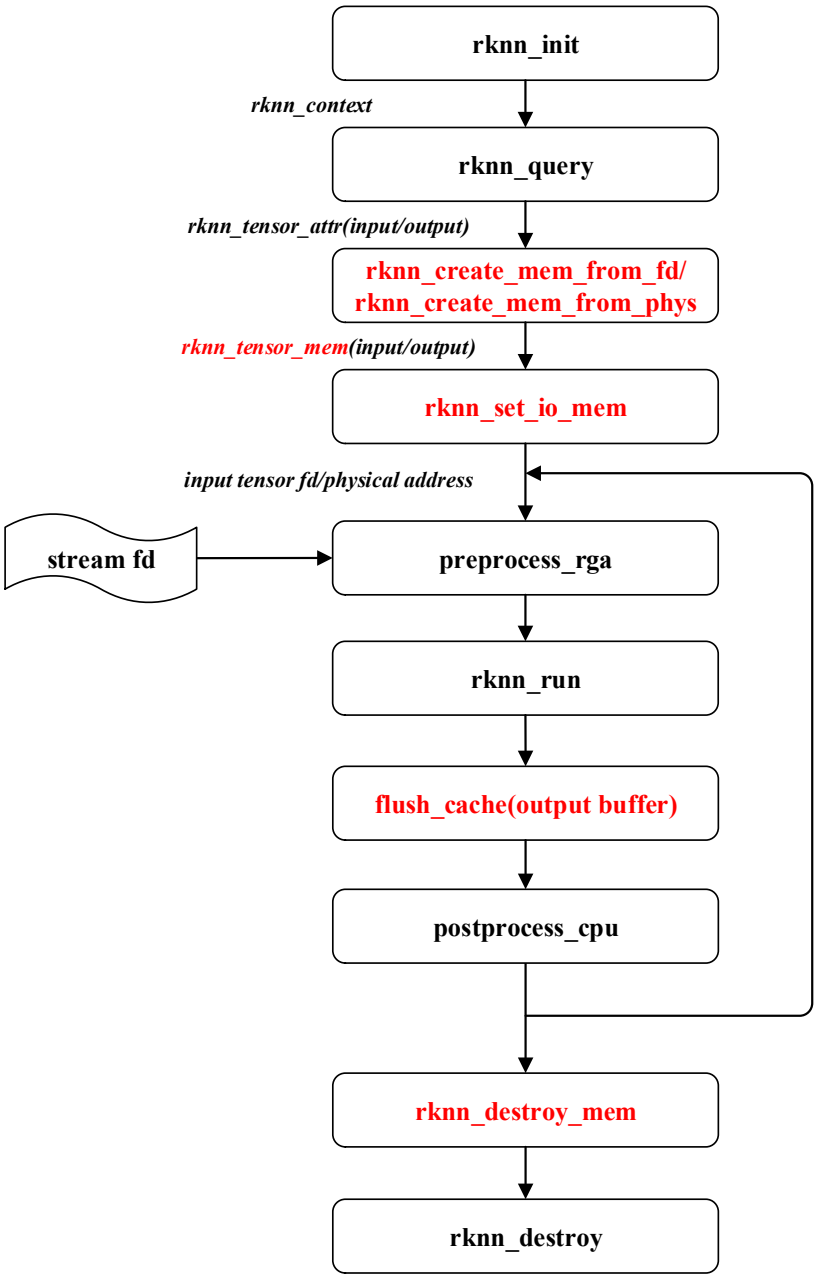

图 3-3 零拷贝 API 接口调用流程（输入/输出外部分配）

如图 3-3 所示，flush_cache 表示用户需要调用与分配的内存类型关联的接口来刷新输出缓存。

3）输入/输出/权重/中间结果内存由外部分配

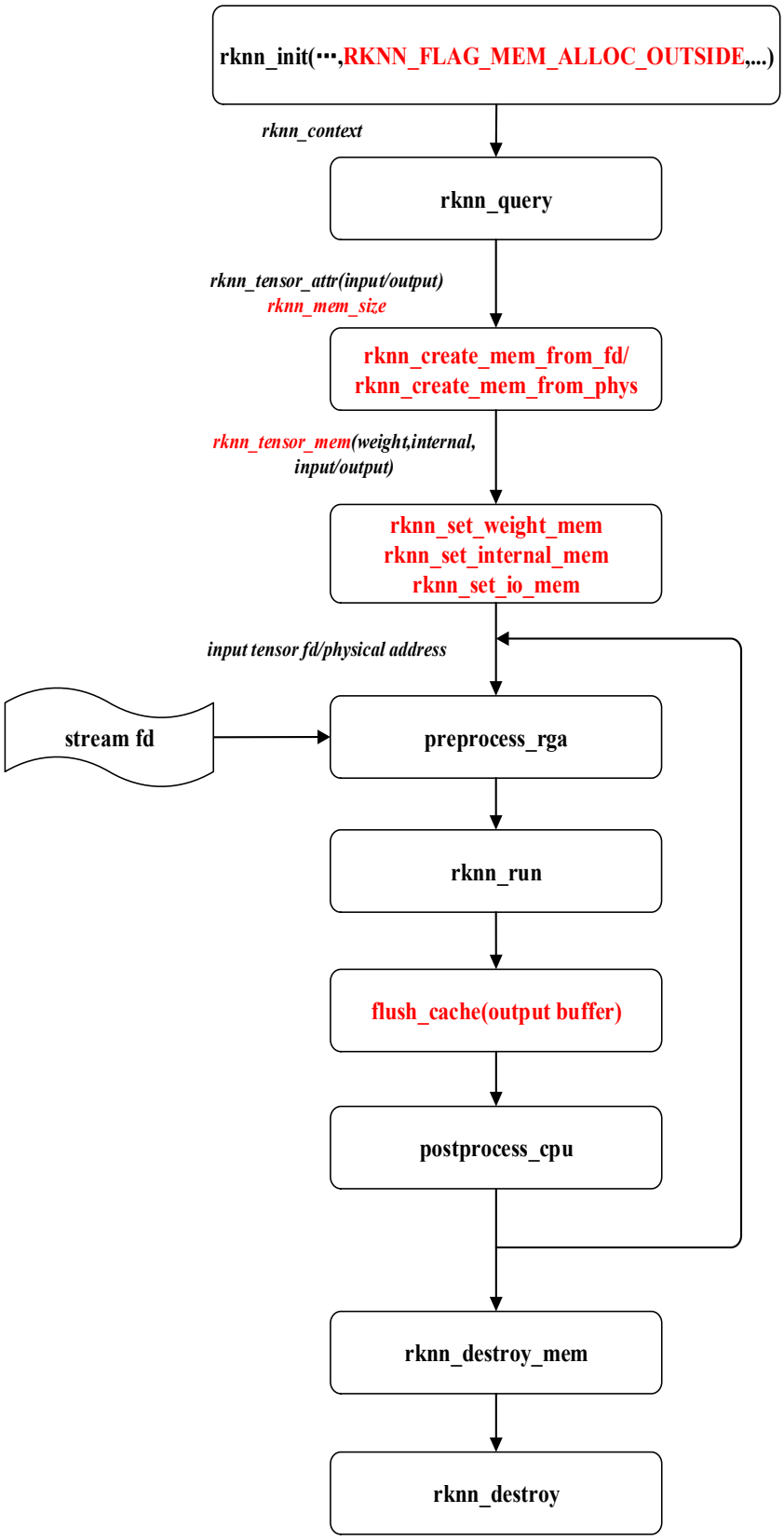

图 3-4 零拷贝 API 接口调用流程（输入/输出/权重/中间结果外部分配）

4.4.2.1 API 内部处理流程

在推理 RKNN 模型时，原始数据要经过输入处理、NPU 运行模型、输出处理三大流程。目前根据不同模型输入格式和量化方式，通用 API 接口内部会存在以下两种处理流程。（**注意：目前 RV1106/RV1103 只支持 int8 量化模型，且只支持通道数为 1 或 3 或 4 uint8 数据输入**）

1）int8 量化模型且输入通道数是 1 或 3 或 4

如图 3-5 所示，原始数据的处理流程经过优化。假设输入是 3 通道的模型，用户必须保证 R、G、B 三个通道的颜色顺序与训练模型时一致，归一化、量化和模型推理都会在 NPU 上运行，NPU 输出的数据排布格式和反量化过程在 CPU 上运行。

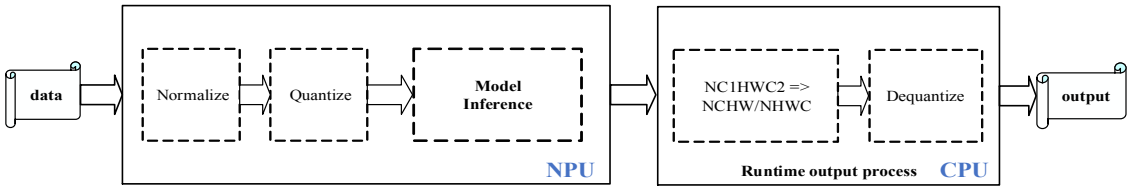

图 3-5 优化的数据处理流程

2）输入通道数是 2 或大于等于 4 的量化模型或非量化模型

对数据处理的流程如图 3-6 所示。对于数据的归一化、量化、数据排布格式转换、反量化均在 CPU 上运行，模型本身的推理在 NPU 上运行。此场景下，对于输入数据流程的处理效率会低于图 3-5 中优化的输入数据处理流程。

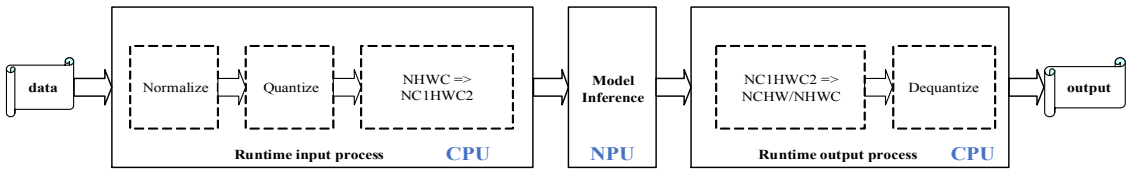

图 3-6 普通的数据处理流程

对于零拷贝的 API，API 内部流程只存在一种运行情形，如图 3-5 所示。零拷贝场景的条件如下：

1. 输入通道数是 1 或 3 或 4。

2. RK356X 输入的宽度是 8 像素对齐， RK3588 和 RV1106/RV1103 输入的宽度是 16 像素对齐。

3. int8 非对称量化模型。

4.4.2.2 量化和反量化

量化和反量化用到的量化方式、量化数据类型以及量化参数，可以通过 rknn_query 接口查询。

目前，RK356X/RK3588/RV1106/RV1103 只支持非对称量化，不支持动态定点量化，数据类型和量化方式组合包括：

- int8（非对称量化）
- int16（非对称量化，**暂未实现**）
- float16（**RV1106/RV1103 不支持**）

通常，归一化后的数据用 32 位浮点数保存，32 位浮点转换成 16 位浮点数请参考 IEEE-754 标准。假设归一化后的 32 位浮点数据是 D，下面介绍量化流程：

1）float32 转 int8（非对称量化）

假设输入 tensor 的非对称量化参数是 S_q，ZP，数据 D 量化过程表示为下式：

$$D_q = round(clamp(D / S_q + ZP, -128, 127))$$

上式中，clamp 表示将数值限制在某个范围。round 表示做舍入处理。

2）float32 转 int16（非对称量化）

假设输入 tensor 的非对称量化参数是 S_q、ZP，数据 D 量化过程表示为下式：

$$D_q = round(clamp(D / S_q + ZP, -32768, 32767))$$

反量化流程是量化的逆过程，可以根据上述量化公式反推出反量化公式，这里不做赘述。

4.4.3 API 详细说明

4.4.3.1 rknn_init

rknn_init 初始化函数将创建 rknn_context 对象、加载 RKNN 模型以及根据 flag 和 rknn_init_extend 结构体执行特定的初始化行为。

API	rknn_init
功能	初始化 rknn
参数	rknn_context *context：rknn_context 指针。函数调用之后，context 将会被赋值。
	void *model：RKNN 模型的二进制数据或者 RKNN 模型路径。
	uint32_t size：当 model 是二进制数据，表示模型大小，当 model 是路径，则设置为 0。
	uint32_t flag：特定的初始化标志。目前，仅支持如下标志： RKNN_FLAG_COLLECT_PERF_MASK：用于运行时查询网络各层时间。 RKNN_FLAG_MEM_ALLOC_OUTSIDE：用于表示模型输入、输出、权重、中间 tensor 内存全部由用户分配。 RKNN_FLAG_SHARE_WEIGHT_MEM：用于表示共享另个模型的 weight 权重
	rknn_init_extend：特定初始化时的扩展信息。没有使用，传入 NULL 即可。如果需要共享模型 weight 内存，则需要传入另个模型 rknn_context 指针
返回值	int 错误码（见 rknn 返回值错误码）。

示例代码如下：

```
rknn_context ctx;
int ret = rknn_init(&ctx, model_data, model_data_size, 0, NULL);
```

4.4.3.2 rknn_set_core_mask

rknn_set_core_mask 函数指定工作的 NPU 核心，该函数仅支持 RK3588 平台（包含三个 NPU 核心），在 RK356X/RV1106/RV1103 平台上设置会返回错误。

API	rknn_set_core_mask
功能	设置运行的 NPU 核心
参数	rknn_context context：设置运行核的 rknn_context 对象。
	rknn_core_mask core_mask：NPU 核心的枚举类型，目前有如下方式配置： **RKNN_NPU_CORE_AUTO**：表示自动调度模型，自动运行在当前空闲的 NPU 核上； **RKNN_NPU_CORE_0**：表示运行在 NPU0 核上 **RKNN_NPU_CORE_1**：表示运行在 NPU1 核上 **RKNN_NPU_CORE_2**：表示运行在 NPU2 核上 **RKNN_NPU_CORE_0_1**：表示同时工作在 NPU0、NPU1 核上（暂不支持） **RKNN_NPU_CORE_0_1_2**：表示同时工作在 NPU0、NPU1、NPU2 核上（暂不支持）
返回值	int 错误码（见 rknn 返回值错误码）。

示例代码如下：

```
rknn_context ctx;
rknn_core_mask core_mask = RKNN_NPU_CORE_0;
int ret = rknn_set_core_mask(ctx, core_mask);
```

4.4.3.3 rknn_dup_context

rknn_dup_context 生成一个指向同一个模型的新 context，可用于多线程执行相同模型时的权重复用，支持 RK356X/RK3588 芯片。**(RV1106/RV1103 平台暂不支持)**

API	rknn_dup_context
功能	生成同一个模型的两个 ctx，复用模型的权重信息
参数	rknn_context * context_in：rknn_context 指针。初始化后的 rknn_context 对象
	rknn_context * context_out：rknn_context 指针。生成新的 rknn_context 对象
返回值	int 错误码（见 rknn 返回值错误码）。

示例代码如下：

```
rknn_context ctx_in;
rknn_context ctx_out;
int ret = rknn_dup_context(&ctx_in, &ctx_out);
```

4.4.3.4 rknn_destroy

rknn_destroy 函数将释放传入的 rknn_context 及其相关资源。

API	rknn_destroy
功能	销毁 rknn_context 对象及其相关资源。
参数	rknn_context context：要销毁的 rknn_context 对象。
返回值	int 错误码（见 rknn 返回值错误码）。

示例代码如下：

```
rknn_context ctx;
int ret = rknn_destroy (ctx);
```

4.4.3.5 rknn_query

rknn_query 函数能够查询获取到模型输入输出信息、逐层运行时间、模型推理的总时间、

SDK 版本、内存占用信息、用户自定义字符串等信息。

API	rknn_query
功能	查询模型与 SDK 的相关信息。
参数	rknn_context context：rknn_context 对象。
	rknn_query_cmd cmd：查询命令。
	void* info：存放返回结果的结构体变量。
	uint32_t size：info 对应的结构体变量的大小。
返回值	int 错误码（见 rknn 返回值错误码）

当前 SDK 支持的查询命令如下表所示：

查询命令	返回结果结构体	功能
RKNN_QUERY_IN_OUT_NUM	rknn_input_output_num	查询输入输出 tensor 个数
RKNN_QUERY_INPUT_ATTR	rknn_tensor_attr	使用通用 API 接口时,查询输入 tensor 属性
RKNN_QUERY_OUTPUT_ATTR	rknn_tensor_attr	使用通用 API 接口时,查询输出 tensor 属性
RKNN_QUERY_PERF_DETAIL	rknn_perf_detail	查询网络各层运行时间，**需要调用 rknn_init 接口时，设置 RKNN_FLAG_COLLECT_PERF_MASK 标志才能生效**
RKNN_QUERY_PERF_RUN	rknn_perf_run	查询推理模型（不包含设置输入/输出）的耗时，单位是微秒
RKNN_QUERY_SDK_VERSION	rknn_sdk_version	查询 SDK 版本
RKNN_QUERY_MEM_SIZE	rknn_mem_size	查询分配给权重和网络中间 tensor 的内存大小

RKNN_QUERY_CUSTOM_STRING	rknn_custom_string	查询 RKNN 模型里面的用户自定义字符串信息
RKNN_QUERY_NATIVE_INPUT_ATTR	rknn_tensor_attr	使用零拷贝 API 接口时,查询原生输入 tensor 属性，它是 NPU 直接读取的模型输入属性
RKNN_QUERY_NATIVE_OUTPUT_ATTR	rknn_tensor_attr	使用零拷贝 API 接口时,查询原生输出 tensor 属性，它是 NPU 直接输出的模型输出属性
RKNN_QUERY_NATIVE_NC1HWC2_INPUT_ATTR	rknn_tensor_attr	使用零拷贝 API 接口时,查询原生输入 tensor 属性，它是 NPU 直接读取的模型输入属性与 RKNN_QUERY_NATIVE_INPUT_ATTR 查询结果一致
RKNN_QUERY_NATIVE_NC1HWC2_OUTPUT_ATTR	rknn_tensor_attr	使用零拷贝 API 接口时,查询原生输出 tensor 属性，它是 NPU 直接输出的模型输出属与 RKNN_QUERY_NATIVE_OUTPUT_ATTR 查询结果一致性
RKNN_QUERY_NATIVE_NHWC_INPUT_ATTR	rknn_tensor_attr	使用零拷贝 API 接口时,查询原生输入 tensor 属性与 RKNN_QUERY_NATIVE_INPUT_ATTR 查询结果一致
RKNN_QUERY_NATIVE_NHWC_OUTPUT_ATTR	rknn_tensor_attr	使用零拷贝 API 接口时,查询原生输出 NHWC tensor 属性，

各个指令用法的详细说明，如下：

1） 查询 SDK 版本

传入 RKNN_QUERY_SDK_VERSION 命令可以查询 RKNN SDK 的版本信息。其中需要先创建 rknn_sdk_version 结构体对象。

示例代码如下：

```
rknn_sdk_version version;
ret = rknn_query(ctx, RKNN_QUERY_SDK_VERSION, &version,
                 sizeof(rknn_sdk_version));
printf("sdk api version: %s\n", version.api_version);
printf("driver version: %s\n", version.drv_version);
```

2） 查询输入输出 tensor 个数

在 rknn_init 接口调用完毕后，传入 RKNN_QUERY_IN_OUT_NUM 命令可以查询模型输入输出 tensor 的个数。其中需要先创建 rknn_input_output_num 结构体对象。

示例代码如下：

```
rknn_input_output_num io_num;
ret = rknn_query(ctx, RKNN_QUERY_IN_OUT_NUM, &io_num,
                 sizeof(io_num));
printf("model input num: %d, output num: %d\n", io_num.n_input,
                 io_num.n_output);
```

3） 查询输入 tensor 属性(用于通用 API 接口)

在 rknn_init 接口调用完毕后，传入 RKNN_QUERY_INPUT_ATTR 命令可以查询模型输入 tensor 的属性。其中需要先创建 rknn_tensor_attr 结构体对象。(**注意：RV1106/RV1103 查询出来的 tensor 是原始输入 native 的 tensor**)

示例代码如下：

```
rknn_tensor_attr input_attrs[io_num.n_input];
memset(input_attrs, 0, sizeof(input_attrs));
for (int i = 0; i < io_num.n_input; i++) {
    input_attrs[i].index = i;
    ret = rknn_query(ctx, RKNN_QUERY_INPUT_ATTR, &(input_attrs[i]),
                     sizeof(rknn_tensor_attr));
}
```

4） 查询输出 tensor 属性(用于通用 API 接口)

在 rknn_init 接口调用完毕后，传入 RKNN_QUERY_OUTPUT_ATTR 命令可以查询模型输

出 tensor 的属性。其中需要先创建 rknn_tensor_attr 结构体对象。(**注意：RV1106/RV1103 查询出来的 tensor 是原始输出 native 的 tensor**)

示例代码如下：

```
rknn_tensor_attr output_attrs[io_num.n_output];
memset(output_attrs, 0, sizeof(output_attrs));
for (int i = 0; i < io_num.n_output; i++) {
        output_attrs[i].index = i;
        ret = rknn_query(ctx, RKNN_QUERY_OUTPUT_ATTR, &(output_attrs[i]),
                                        sizeof(rknn_tensor_attr));
}
```

5） 查询模型推理的逐层耗时

在 rknn_run 接口调用完毕后，rknn_query 接口传入 RKNN_QUERY_PERF_DETAIL 可以查询网络推理时逐层的耗时，单位是微秒。使用该命令的前提是，在 rknn_init 接口的 flag 参数需要包含 RKNN_FLAG_COLLECT_PERF_MASK 标志。

示例代码如下：

```
rknn_context ctx;
int ret = rknn_init(&ctx, model_data, model_data_size,
                            RKNN_FLAG_COLLECT_PERF_MASK, NULL);
...
ret = rknn_run(ctx,NULL);
...
rknn_perf_detail perf_detail;
ret = rknn_query(ctx, RKNN_QUERY_PERF_DETAIL, &perf_detail,
                            sizeof(perf_detail));
```

6） 查询模型推理的总耗时

在 rknn_run 接口调用完毕后，rknn_query 接口传入 RKNN_QUERY_PERF_RUN 可以查询上模型推理（不包含设置输入/输出）的耗时，单位是微秒。

示例代码如下：

```
rknn_context ctx;
int ret = rknn_init(&ctx, model_data, model_data_size, 0, NULL);
...
ret = rknn_run(ctx,NULL);
...
rknn_perf_run perf_run;
ret = rknn_query(ctx, RKNN_QUERY_PERF_RUN, &perf_run,
                 sizeof(perf_run));
```

7） 查询模型的内存占用情况

在 rknn_init 接口调用完毕后，当用户需要自行分配网络的内存时，rknn_query 接口传入 RKNN_QUERY_MEM_SIZE 可以查询模型的权重和网络中间 tensor 的内存（不包括输入和输出）占用情况。使用该命令的前提是在 rknn_init 接口的 flag 参数需要包含 RKNN_FLAG_MEM_ALLOC_OUTSIDE 标志。

示例代码如下：

```
rknn_context ctx;
int ret = rknn_init(&ctx, model_data, model_data_size,
                    RKNN_FLAG_MEM_ALLOC_OUTSIDE , NULL);
rknn_mem_size mem_size;
ret = rknn_query(ctx, RKNN_QUERY_MEM_SIZE, &mem_size,
                 sizeof(mem_size));
```

8） 查询模型里用户自定义字符串

在 rknn_init 接口调用完毕后，当用户需要查询生成 RKNN 模型时加入的自定义字符串，rknn_query 接口传入 RKNN_QUERY_CUSTOM_STRING 可以获取该字符串。

示例代码如下：

```
rknn_context ctx;
int ret = rknn_init(&ctx, model_data, model_data_size, 0, NULL);
rknn_custom_string custom_string;
ret = rknn_query(ctx, RKNN_QUERY_CUSTOM_STRING, &custom_string,
                 sizeof(custom_string));
```

9） 查询原始输入 tensor 属性(用于零拷贝 API 接口)

在 rknn_init 接口调用完毕后，传入 RKNN_QUERY_NATIVE_INPUT_ATTR 命令可以查询模型原生输入 tensor 的属性。其中需要先创建 rknn_tensor_attr 结构体对象。

示例代码如下：

```
rknn_tensor_attr input_attrs[io_num.n_input];
memset(input_attrs, 0, sizeof(input_attrs));
for (int i = 0; i < io_num.n_input; i++) {
    input_attrs[i].index = i;
    ret = rknn_query(ctx, RKNN_QUERY_NATIVE_INPUT_ATTR,
                     &(input_attrs[i]), sizeof(rknn_tensor_attr));
}
```

10）查询原始输出 tensor 属性(用于零拷贝 API 接口)

在 rknn_init 接口调用完毕后，传入 RKNN_QUERY_NATIVE_OUTPUT_ATTR 命令可以查询模型原生输出 tensor 的属性。其中需要先创建 rknn_tensor_attr 结构体对象。

示例代码如下：

```
rknn_tensor_attr output_attrs[io_num.n_output];
memset(output_attrs, 0, sizeof(output_attrs));
for (int i = 0; i < io_num.n_output; i++) {
    output_attrs[i].index = i;
    ret = rknn_query(ctx, RKNN_QUERY_NATIVE_OUTPUT_ATTR,
                     &(output_attrs[i]), sizeof(rknn_tensor_attr));
}
```

11) 查询原始输入 tensor 属性(用于零拷贝 API 接口)

在 rknn_init 接口调用完毕后，传入 RKNN_QUERY_NATIVE_NHWC_INPUT_ATTR 命令可以查询模型 NHWC 输入 tensor 的属性。其中需要先创建 rknn_tensor_attr 结构体对象。

示例代码如下：

```
rknn_tensor_attr input_attrs[io_num.n_input];
memset(input_attrs, 0, sizeof(input_attrs));
for (int i = 0; i < io_num.n_input; i++) {
    input_attrs[i].index = i;
    ret = rknn_query(ctx, RKNN_QUERY_NATIVE_NHWC_INPUT_ATTR,
                     &(input_attrs[i]), sizeof(rknn_tensor_attr));
}
```

12) 查询原始输出 tensor 属性(用于零拷贝 API 接口)

在 rknn_init 接口调用完毕后，传入 RKNN_QUERY_NATIVE_NHWC_OUTPUT_ATTR 命

令可以查询模型原生输出 tensor 的属性。其中需要先创建 rknn_tensor_attr 结构体对象。

示例代码如下：

```
rknn_tensor_attr output_attrs[io_num.n_output];
memset(output_attrs, 0, sizeof(output_attrs));
for (int i = 0; i < io_num.n_output; i++) {
        output_attrs[i].index = i;
        ret = rknn_query(ctx, RKNN_QUERY_NATIVE_NHWC_OUTPUT_ATTR,
                                        &(output_attrs[i]), sizeof(rknn_tensor_attr));
}
```

4.4.3.6 rknn_inputs_set

通过 rknn_inputs_set 函数可以设置模型的输入数据。该函数能够支持多个输入，其中每个输入是 rknn_input 结构体对象，在传入之前用户需要设置该对象。**（注：RV1106/RV1103 不支持这个接口）**

API	rknn_inputs_set
功能	设置模型输入数据。
参数	rknn_context context：rknn_context 对象。
	uint32_t n_inputs：输入数据个数。
	rknn_input inputs[]：输入数据数组，数组每个元素是 rknn_input 结构体对象。
返回值	int 错误码（见 rknn 返回值错误码）

示例代码如下：

```
rknn_input inputs[1];
memset(inputs, 0, sizeof(inputs));
inputs[0].index = 0;
inputs[0].type = RKNN_TENSOR_UINT8;
inputs[0].size = img_width*img_height*img_channels;
inputs[0].fmt = RKNN_TENSOR_NHWC;
inputs[0].buf = in_data;
inputs[0].pass_through = 0;

ret = rknn_inputs_set(ctx, 1, inputs);
```

4.4.3.7 rknn_run

rknn_run 函数将执行一次模型推理，调用之前需要先通过 rknn_inputs_set 函数或者零拷贝的接口设置输入数据。

API	rknn_run
功能	执行一次模型推理。
参数	rknn_context context：rknn_context 对象。
	rknn_run_extend* extend：保留扩展，当前没有使用，传入 NULL 即可。
返回值	int 错误码（见 rknn 返回值错误码）

示例代码如下：

```
ret = rknn_run(ctx, NULL);
```

4.4.3.8 rknn_wait

该接口用于非阻塞模式推理，**目前暂未实现**。

4.4.3.9 rknn_outputs_get

rknn_outputs_get 函数可以获取模型推理的输出数据。该函数能够一次获取多个输出数据。其中每个输出是 rknn_output 结构体对象，在函数调用之前需要依次创建并设置每个 rknn_output 对象。

对于输出数据的 buffer 存放可以采用两种方式：一种是用户自行申请和释放，此时 rknn_output 对象的 is_prealloc 需要设置为 1，并且将 buf 指针指向用户申请的 buffer；另一种是由 rknn 来进行分配，此时 rknn_output 对象的 is_prealloc 设置为 0 即可，函数执行之后 buf 将指向输出数据。**（注：RV1106/RV1103 不支持这个接口）**

API	rknn_outputs_get
功能	获取模型推理输出。
参数	rknn_context context：rknn_context 对象。
	uint32_t n_outputs：输出数据个数。
	rknn_output outputs[]：输出数据的数组，其中数组每个元素为 rknn_output 结构体对象，代表模型的一个输出。
	rknn_output_extend* extend：保留扩展，当前没有使用，传入 NULL 即可
返回值	int 错误码（见 rknn 返回值错误码）

示例代码如下：

```
rknn_output outputs[io_num.n_output];
memset(outputs, 0, sizeof(outputs));
for (int i = 0; i < io_num.n_output; i++) {
    outputs[i].index = i;
    outputs[i].is_prealloc = 0;
    outputs[i].want_float = 1;
}
ret = rknn_outputs_get(ctx, io_num.n_output, outputs, NULL);
```

4.4.3.10 rknn_outputs_release

rknn_outputs_release 函数将释放 rknn_outputs_get 函数得到的输出的相关资源。

API	rknn_outputs_release
功能	释放 rknn_output 对象。
参数	rknn_context context：rknn_context 对象。
	uint32_t n_outputs：输出数据个数。
	rknn_output outputs[]：要销毁的 rknn_output 数组。
返回值	int 错误码（见 rknn 返回值错误码）

示例代码如下：

```
ret = rknn_outputs_release(ctx, io_num.n_output, outputs);
```

4.4.3.11 rknn_create_mem_from_mb_blk

目前暂未实现。

4.4.3.12 rknn_create_mem_from_phys

当用户要自己分配内存让 NPU 使用时，通过 rknn_create_mem_from_phys 函数可以创建一个 rknn_tensor_mem 结构体并得到它的指针，该函数通过传入物理地址、虚拟地址以及大小，外部内存相关的信息会赋值给 rknn_tensor_mem 结构体。**（注：RV1106/RV1103 暂时不支持这个接口）**

API	rknn_create_mem_from_phys
功能	通过物理地址创建 rknn_tensor_mem 结构体并分配内存
参数	rknn_context context：rknn_context 对象。
	uint64_t phys_addr：内存的物理地址。
	void *virt_addr：内存的虚拟地址。
	uint32_t size：内存的大小。
返回值	rknn_tensor_mem*： tensor 内存信息结构体指针。

示例代码如下：

```
//suppose we have got buffer information as input_phys, input_virt and size
rknn_tensor_mem* input_mems [1];
input_mems[0] = rknn_create_mem_from_phys(ctx, input_phys, input_virt, size);
```

4.4.3.13 rknn_create_mem_from_fd

当用户要自己分配内存让 NPU 使用时，rknn_create_mem_from_fd 函数可以创建一个 rknn_tensor_mem 结构体并得到它的指针，该函数通过传入文件描述符 fd、偏移、虚拟地址以及大小，外部内存相关的信息会赋值给 rknn_tensor_mem 结构体。**（注：RV1106/RV1103 暂时不支持这个接口）**

API	rknn_create_mem_from_fd
功能	通过文件描述符创建 rknn_tensor_mem 结构体
参数	rknn_context context：rknn_context 对象。
	int32_t fd：内存的文件描述符。
	void *virt_addr：内存的虚拟地址，**fd 对应的首地址**。
	uint32_t size：内存的大小。
	int32_t offset：内存相对于文件描述符和虚拟地址的偏移量。
返回值	rknn_tensor_mem*： tensor 内存信息结构体指针。

示例代码如下：

```
//suppose we have got buffer information as input_fd, input_virt and size
rknn_tensor_mem* input_mems [1];
input_mems[0] = rknn_create_mem_from_fd(ctx, input_fd, input_virt, size, 0);
```

4.4.3.14 rknn_create_mem

当用户要 NPU 内部分配内存时，rknn_create_mem 函数可以创建一个 rknn_tensor_mem 结构体并得到它的指针，该函数通过传入内存大小，运行时会初始化 rknn_tensor_mem 结构体。

API	rknn_create_mem
功能	运行时内部创建 rknn_tensor_mem 结构体并分配内存
参数	rknn_context context：rknn_context 对象。
	uint32_t size：内存的大小。
返回值	rknn_tensor_mem*： tensor 内存信息结构体指针。

示例代码如下：

```
//suppose we have got buffer size
rknn_tensor_mem* input_mems [1];
input_mems[0] = rknn_create_mem(ctx, size);
```

4.4.3.15 rknn_destroy_mem

rknn_destroy_mem 函数会销毁 rknn_tensor_mem 结构体，用户分配的内存需要自行释放。

API	rknn_destroy_mem
功能	销毁 rknn_tensor_mem 结构体
参数	rknn_context context：rknn_context 对象。
	rknn_tensor_mem*：tensor 内存信息结构体指针。
返回值	int 错误码（见 rknn 返回值错误码）。

示例代码如下：

```
rknn_tensor_mem* input_mems [1];
int ret = rknn_destroy_mem(ctx, input_mems[0]);
```

4.4.3.16 rknn_set_weight_mem

如果用户自己为网络权重分配内存，初始化相应的 rknn_tensor_mem 结构体后，在调用 rknn_run 前，通过 rknn_set_weight_mem 函数可以让 NPU 使用该内存。

API	rknn_set_weight_mem
功能	设置包含权重内存信息的 rknn_tensor_mem 结构体
参数	rknn_context context：rknn_context 对象。
	rknn_tensor_mem*：权重 tensor 内存信息结构体指针。
返回值	int 错误码（见 rknn 返回值错误码）。

示例代码如下：

```
rknn_tensor_mem* weight_mems [1];
int ret = rknn_set_weight_mem(ctx, weight_mems[0]);
```

4.4.3.17 rknn_set_internal_mem

如果用户自己为网络中间 tensor 分配内存，初始化相应的 rknn_tensor_mem 结构体后，在调用 rknn_run 前，通过 rknn_set_internal_mem 函数可以让 NPU 使用该内存。

API	rknn_set_internal_mem
功能	设置包含中间 tensor 内存信息的 rknn_tensor_mem 结构体
参数	rknn_context context：rknn_context 对象。
	rknn_tensor_mem*：模型中间 tensor 内存信息结构体指针。
返回值	int 错误码（见 rknn 返回值错误码）。

示例代码如下：

```
rknn_tensor_mem* internal_tensor_mems [1];
int ret = rknn_set_internal_mem(ctx, internal_tensor_mems[0]);
```

4.4.3.18 rknn_set_io_mem

如果用户自己为网络输入/输出 tensor 分配内存，初始化相应的 rknn_tensor_mem 结构体后，在调用 rknn_run 前，通过 rknn_set_io_mem 函数可以让 NPU 使用该内存。

API	rknn_set_io_mem
功能	设置包含模型输入/输出内存信息的 rknn_tensor_mem 结构体。
参数	rknn_context context：rknn_context 对象。
	rknn_tensor_mem*：输入/输出 tensor 内存信息结构体指针。
	rknn_tensor_attr *: 输入/输出 tensor 的属性。
返回值	int 错误码（见 rknn 返回值错误码）。

示例代码如下：

```
rknn_tensor_attr output_attrs[1];
rknn_tensor_mem* output_mems[1];

ret = rknn_query(ctx, RKNN_QUERY_NATIVE_OUTPUT_ATTR, &(output_attrs[0]),
sizeof(rknn_tensor_attr));
output_mems[0] = rknn_create_mem(ctx, output_attrs[0].size_with_stride);
rknn_set_io_mem(ctx, output_mems[0], &output_attrs[0]);
```

4.4.4 RKNN 数据结构定义

4.4.4.1 rknn_sdk_version

结构体 rknn_sdk_version 用来表示 RKNN SDK 的版本信息，结构体的定义如下：

成员变量	数据类型	含义
api_version	char[]	SDK 的版本信息。
drv_version	char[]	SDK 所基于的驱动版本信息。

4.4.4.2 rknn_input_output_num

结构体 rknn_input_output_num 表示输入输出 tensor 个数，其结构体成员变量如下表所示：

成员变量	数据类型	含义
n_input	uint32_t	输入 tensor 个数
n_output	uint32_t	输出 tensor 个数

4.4.4.3 rknn_tensor_attr

结构体 rknn_tensor_attr 表示模型的 tensor 的属性，结构体的定义如下表所示：

成员变量	数据类型	含义
index	uint32_t	表示输入输出 tensor 的索引位置。
n_dims	uint32_t	Tensor 维度个数。
dims	uint32_t[]	Tensor 各维度值。
name	char[]	Tensor 名称。
n_elems	uint32_t	Tensor 数据元素个数。
size	uint32_t	Tensor 数据所占内存大小。
fmt	rknn_tensor_format	Tensor 维度的格式，有以下格式： **RKNN_TENSOR_NCHW** **RKNN_TENSOR_NHWC** **RKNN_TENSOR_NC1HWC2**
type	rknn_tensor_type	Tensor 数据类型，有以下数据类型： **RKNN_TENSOR_FLOAT32** **RKNN_TENSOR_FLOAT16** **RKNN_TENSOR_INT8** **RKNN_TENSOR_UINT8** **RKNN_TENSOR_INT16** **RKNN_TENSOR_UINT16** **RKNN_TENSOR_INT32** **RKNN_TENSOR_INT64** **RKNN_TENSOR_BOOL**

qnt_type	rknn_tensor_qnt_type	Tensor 量化类型，有以下的量化类型： **RKNN_TENSOR_QNT_NONE**：未量化； **RKNN_TENSOR_QNT_DFP**：动态定点量化； **RKNN_TENSOR_QNT_AFFINE_ASYMMETRIC**：非对称量化。
fl	int8_t	RKNN_TENSOR_QNT_DFP 量化类型的参数。
zp	int32_t	RKNN_TENSOR_QNT_AFFINE_ASYMMETRIC 量化类型的参数。
scale	float	RKNN_TENSOR_QNT_AFFINE_ASYMMETRIC 量化类型的参数。
stride	uint32_t	实际存储一行图像数据的像素数目，等于一行的有效数据像素数目 + 为硬件快速跨越到下一行而补齐的一些无效像素数目，单位是像素。
size_with_stride	uint32_t	实际存储图像数据所占的存储空间的大小（包括了补齐的无效像素的存储空间大小）。
pass_through	uint8_t	0 表示未转换的数据，1 表示转换后的数据，转换包括归一化和量化。

4.4.4.4 rknn_perf_detail

结构体 rknn_perf_detail 表示模型的性能详情，结构体的定义如下表所示：（RV1106/RV1103 暂不支持）

成员变量	数据类型	含义
perf_data	char*	性能详情包含网络每层运行时间，能够直接打印出来查看。
data_len	uint64_t	存放性能详情的字符串数组的长度。

4.4.4.5 rknn_perf_run

结构体 rknn_perf_run 表示模型的总体性能，结构体的定义如下表所示：（RV1106/RV1103 暂不支持）

成员变量	数据类型	含义
run_duration	int64_t	网络总体运行（不包含设置输入/输出）时间，单位是微秒。

4.4.4.6 rknn_mem_size

结构体 rknn_mem_size 表示初始化模型时的内存分配情况，结构体的定义如下表所示：

成员变量	数据类型	含义
total_weight_size	uint32_t	网络的权重占用的内存大小。
total_internal_size	uint32_t	网络的中间 tensor 占用的内存大小。

4.4.4.7 rknn_tensor_mem

结构体 rknn_tensor_mem 表示 tensor 的内存信息。结构体的定义如下表所示：

成员变量	数据类型	含义
virt_addr	void*	该 tensor 的虚拟地址。
phys_addr	uint64_t	该 tensor 的物理地址。
fd	int32_t	该 tensor 的文件描述符。
offset	int32_t	相较于文件描述符和虚拟地址的偏移量。
size	uint32_t	该 tensor 占用的内存大小。
flags	uint32_t	rknn_tensor_mem 的标志位，有以下标志： **RKNN_TENSOR_MEMORY_FALGS_ALLOC_INSIDE:** 表明 rknn_tensor_mem 结构体由运行时创建。 **用户不用关注该标志**。
priv_data	void*	内存的私有数据。

4.4.4.8 rknn_input

结构体 rknn_input 表示模型的一个数据输入，用来作为参数传入给 rknn_inputs_set 函数。结构体的定义如下表所示：

成员变量	数据类型	含义
index	uint32_t	该输入的索引位置。
buf	void*	输入数据的指针。
size	uint32_t	输入数据所占内存大小。
pass_through	uint8_t	设置为 1 时会将 buf 存放的输入数据直接设置给模型的输入节点，不做任何预处理。
type	rknn_tensor_type	输入数据的类型。
fmt	rknn_tensor_format	输入数据的格式。

4.4.4.9 rknn_output

结构体 rknn_output 表示模型的一个数据输出，用来作为参数传入给 rknn_outputs_get 函数，在函数执行后，结构体对象将会被赋值。结构体的定义如下表所示：

成员变量	数据类型	含义
want_float	uint8_t	标识是否需要将输出数据转为 float 类型输出。
is_prealloc	uint8_t	标识存放输出数据是否是预分配。
index	uint32_t	该输出的索引位置。
buf	void*	输出数据的指针。
size	uint32_t	输出数据所占内存大小。

4.4.4.10 rknn_init_extend

结构体 rknn_init_extend 表示初始化模型时的扩展信息，**目前仅 RK356X 支持使用**。结构体的定义如下表所示：

成员变量	数据类型	含义
ctx	rknn_context	已初始化的 rknn_context 对象。
reserved	uint8_t[128]	预留数据位。

4.4.4.11 rknn_run_extend

结构体 rknn_run_extend 表示模型推理时的扩展信息，**目前暂不支持使用**。结构体的定义如下表所示：

成员变量	数据类型	含义
frame_id	uint64_t	表示当前推理的帧序号。
non_block	int32_t	0 表示阻塞模式，1 表示非阻塞模式，非阻塞即 rknn_run 调用直接返回。
timeout_ms	int32_t	推理超时的上限，单位毫秒。
fence_fd	int32_t	用于非阻塞执行推理。**暂不支持**

4.4.4.12 rknn_output_extend

结构体 rknn_output_extend 表示获取输出的扩展信息，**目前暂不支持使用**。结构体的定义如下表所示：

成员变量	数据类型	含义
frame_id	int32_t	输出结果的帧序号。

4.4.4.13 rknn_custom_string

结构体 rknn_custom_string 表示转换 RKNN 模型时，用户设置的自定义字符串，结构体的定义如下表所示：

成员变量	数据类型	含义
string	char[]	用户自定义字符串。

4.4.5 输入输出 API 详细说明

4.4.5.1 通用输入输出 API（非零拷贝）

特别注意：RV1106/RV1103 不支持这套接口实现

1）rknn_inputs_set

调用该 API 时，可以设置 rknn_inputs_set 接口中的参数：rknn_input inputs[]来选择输入数据的类型大小等等。

对于输入的说明：pass_through 为 rknn_input 结构体中的成员；pass_through 如果设置为 1，则输入的数据直接输入到模型内进行运算，不进行如何转换;如果设置为 0，则输入数据根据设置的 tensor 的 fmt 和 type 进行相应的变换，再输入模型进行运算，注意，变换过程在 rknn api 内部自动处理。如表-1 所示，当模型为 int8 模型，走 pass_through 为 1 的模式，则输入的数据只能是 int8 的数据类型，并且当原始模型输入是 1,3,4 通道数的时候，数据排布方式为 NHWC，其他 channel 的数据排布方式为 NC1HWC2；走 pass_through 为 0 的模式，则输入数据排布方式只支持 NHWC 排布。RK356X/RK3588 支持的非零拷贝接口 NPU 的输入配置，如表 4-1 所示。

表 4-1 RK356X/RK358 非零拷贝接口 NPU 支持的输入配置

Input 数据类型	pass_through	输入 Channel 数	支持 input layout	备注
uint8_t	0	-	NHWC	
float32	0	-	NHWC	
int8	1	1,3,4	NHWC	仅支持 int8 模型
int8	1	非 1,3,4 通道	NC1HWC2	仅支持 int8 模型
float16	0	-	NHWC	
float16	1	1,3,4	NHWC	仅支持 fp16 模型
float16	1	非 1,3,4 通道	NC1HWC2	仅支持 fp16 模型

2）rknn_outputs_get

调用该 API 时，可以设置接口中 output 的 want_float 的参数来选择输入数据的类型。在 1.2.0 版本前的 rknn-toolkit2 工具转换 RKNN 模型时，默认将输出层的数据类型固定设为 float32，因此即使 want_float=0，非量化模型的结果仍旧 float32。在 1.2.0 含 1.2.0 版本后的 toolkit 转换的 rknn 模型，int8 模型输出即为 int8 数据类型；特殊的，如果 int8 模型的最后一层为 float16 输出时模型输出为 float16 数据类型，例如最后一层是 softmax 层；float16 模型则输出为 float16 数据类型。

对于输出说明：want_float 为 rknn_output 结构体中的成员。RK356X/RK3588 支持的非零拷贝接口 NPU 的输出配置，如表 4-2 所示。

表 4-2 RK356X/RK3588 非零拷贝接口 NPU 支持的输出配置

want_float	output 数据类型	支持 output layout	备注
1	float32	NCHW	
0	int8	NCHW	仅支持 int8 模型
	float16	NCHW	

4.4.5.2 零拷贝输入输出 API

rknn_create_mem、rknn_set_io_mem

输入时，设置的数据应该根据实际要放入零拷贝的 input 内存的数据，设置对应的大小、类型、

布局等参数。

对于输入的说明：pass_through 为 rknn_tensor_attr 结构体中的成员，pass_through 参数与 rknn_input 中的 pass_through 一致。RK356X/RK3588 支持的零拷贝接口 NPU 输入配置如表 4-3 所示，RV1106/RV1103 支持零拷贝接口 NPU 输入配置如表 4-4 所示。

表 4-3 RK356X/RK3588 零拷贝接口 NPU 支持的输入配置

Input 数据类型	pass_through	输入 Channel 数	支持 input layout	备注
uint8_t	0	-	NHWC	
float32	0	-	NHWC	
int8	1	1,3,4	NHWC	仅支持 int8 模型
int8	1	非 1,3,4 通道	NC1HWC2	仅支持 int8 模型
float16	0	-	NHWC	
float16	1	1,3,4	NHWC	仅支持 fp16 模型
float16	1	非 1,3,4 通道	NC1HWC2	仅支持 fp16 模型

表 4-4 RV1106/RV1103 零拷贝接口 NPU 支持的输入配置

Input 数据类型	pass_through	输入 Channel 数	支持 input layout	备注
uint8_t	0	-	NHWC	仅支持 int8 模型
int8	1	1,3,4	NHWC	暂不支持
int8	1	非 1,3,4 通道	NC1HWC2	暂不支持

输出时，当不是使用 NATIVE_LAYOUT 配置时，应该根据实际要获取的零拷贝的 output 内存的数据，设置对应的大小、类型、布局等参数。当使用 NATIVE_LAYOUT 配置时，建议直接采用默认配置，即通过 RKNN_QUERY_NATIVE_OUTPUT_ATTR 选项查询到的输出配置，数据内存大小采用 size_with_stride 的大小。RK356X/RK3588 支持的零拷贝接口 NPU 输出配置如表 4-5 所示，RV1106/RV1103 支持零拷贝接口 NPU 输出配置如表 4-6 所示。

表 4-5 RK356X/RK3588 零拷贝接口 NPU 支持的输出配置

output 数据类型	可支持 output layout	备注
float32	NCHW	
int8	NCHW	仅支持 int8 模型
int8	NC1HWC2	仅支持 int8 模型
float16	NCHW	
float16	NC1HWC2	仅支持 fp16 模型

表 4-6 RV1106/RV1103 零拷贝接口 NPU 支持的输出配置

output 数据类型	可支持 output layout	备注
float32	NHWC	暂不支持
int8	NHWC	暂不支持
int8	NC1HWC2	仅支持 int8 模型

4.4.5.3 NATIVE_LAYOUT 查询参数的说明

RKNN_QUERY_NATIVE_INPUT_ATTR

RKNN_QUERY_NATIVE_OUTPUT_ATTR

这两个查询选项用于查询输入输出的原始 tensor 属性。通常，对于 NPU 运行时而言，这个属性的性能表现最优。查询到的属性成员中，rknn_tensor_format 可能是 NC1HWC2，它是一种特殊的内存排布格式，地址由低到高顺序依次是 C2->W->H->C1->N，其中 C2 变化最快，N 变化最慢。C2 的取值不是固定的，而是与芯片平台和数据类型相关的参数，具体数值如下表：

表-5 不同芯片平台和数据类型下 C2 取值

平台	int8(量化)	float16（非量化）	int16（量化）
RK356X	8	4	4
RK3588	16	8	8
RV1106/RV1103	16	8	8

举例说明，假设在 RK356X 上，原始模型里内存布局为 1x1x1x32 的输出（NHWC），转换成

量化的 RKNN 模型后，使用 native 属性查询 RKNN 模型的输出时，输出会被转换成内存布局为 1x4x1x1x8 的 int8 数据类型输出（NC1HWC2），其中 C1=[32/8]=4, C2=8, []表示向上取整。以 RK356X 的 int8 数据在内存中的 NC1HWC2 排列为例，其中 C2=8 排布如图 4-1 所示。

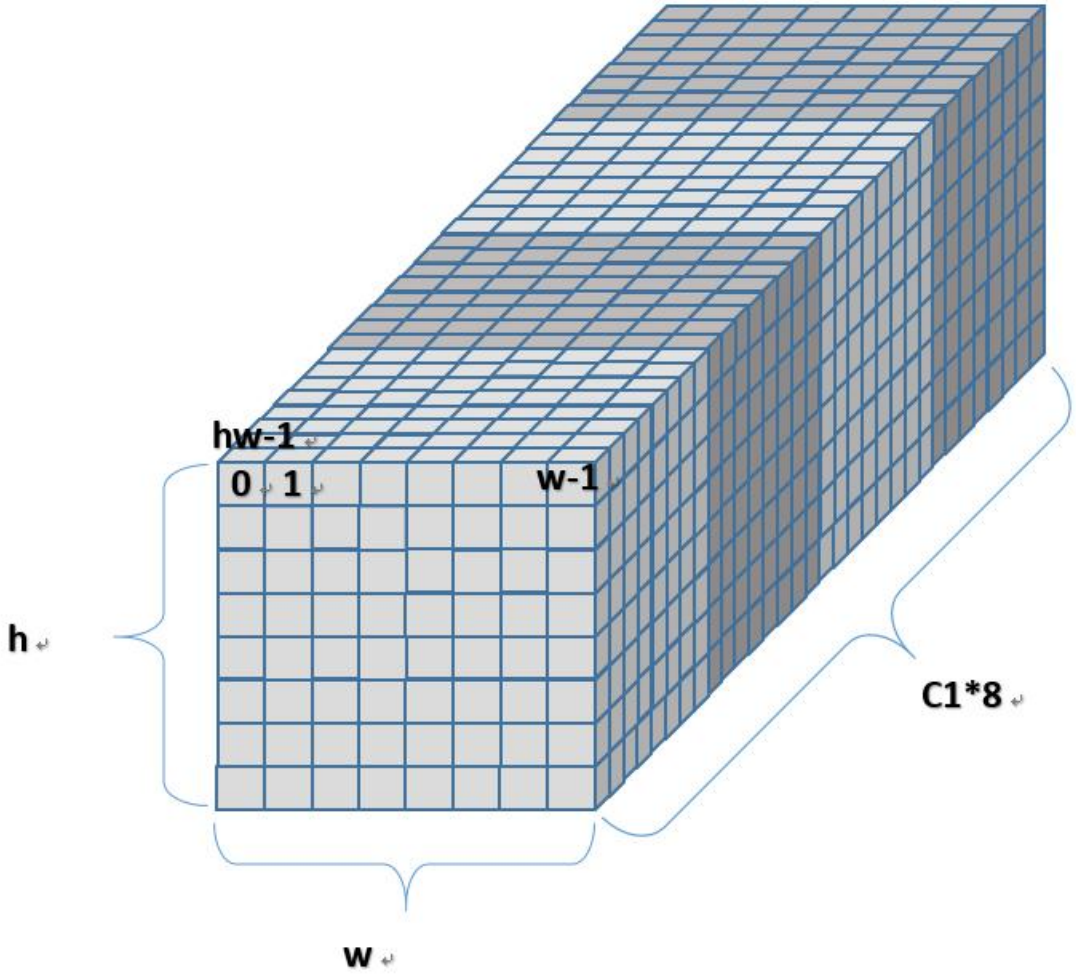

图 4-1 RK356X 的 int8 数据在内存中 NC1HWC2 的排列方式

（1） 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
   * hw_dst :  表示 NCHW 的 h*w 的值
   */
int NC1HWC2_to_NCHW(const int8_t* src, float* dst, int* dims, int channel, int hw_dst)
{
   int batch    = dims[0];
   int C1         = dims[1];
   int h          = dims[2];
   int w          = dims[3];
   int C2         = dims[4];
   int hw_src = w * h;
   for (int i = 0; i < batch; i++) {
      src = src + i * channel * hw_src;
      dst = dst + i * C1 * hw_dst * C2;
      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;
}
```

（2） 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
   * hw_dst :  表示 NHWC 的 h*w 的值
   */
int NC1HWC2_to_NHWC(const int8_t* src, float* dst, int* dims, int channel, int hw_dst )
{
   int batch    = dims[0];
   int C1        = dims[1];
   int h          = dims[2];
   int w          = dims[3];
   int C2        = dims[4];
   int hw_src = w * h;
   for (int i = 0; i < batch; i++) {
      src = src + i * channel * hw_src;
      dst = dst + i * C1 * hw_dst * 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 * align_stride + cur_w;
            for (int c = 0; c < channel; ++c) {
               int               plane = c / C2;
               const auto* src_c = plane * align_hw * C2 + src;
               int               offset = c % C2;
               dst[cur_h * w * channel + cur_w * channel + c] = src_c[C2 * cur_hw + offset];
            }
         }
      }
   }   return 0;
}
```

调用 rknn_query 查询后，运行时将最优性能配置填充到输入输出的 rknn_tensor_attr 结构体中，**如果用户需要的输入或输出配置不同于查询接口获取的 rknn_tensor_attr 结构体，可以对 rknn_tensor_attr 结构体进行对应修改，配置的信息要与上述的表-3 和表-4 相符合，特别注意：如果查询输出的数据类型是 uint8，用户想获取成 float32 类型输出，则 rknn_tensor_attr 结构体的 size 要修改成原 size 的四倍，同时其中的数据类型要修改成 RKNN_TENSOR_FLOAT32。**当使用上述两个命令查询 NATIVE_LAYOUT 后，应该使用零拷贝接口获取输入输出。

4.4.6 RKNN 返回值错误码

RKNN API 函数的返回值错误码定义如下表所示：

错误码	错误详情
RKNN_SUCC（0）	执行成功
RKNN_ERR_FAIL（-1）	执行出错
RKNN_ERR_TIMEOUT（-2）	执行超时
RKNN_ERR_DEVICE_UNAVAILABLE（-3）	NPU 设备不可用
RKNN_ERR_MALLOC_FAIL（-4）	内存分配失败
RKNN_ERR_PARAM_INVALID（-5）	传入参数错误
RKNN_ERR_MODEL_INVALID（-6）	传入的 RKNN 模型无效
RKNN_ERR_CTX_INVALID（-7）	传入的 rknn_context 无效
RKNN_ERR_INPUT_INVALID（-8）	传入的 rknn_input 对象无效
RKNN_ERR_OUTPUT_INVALID（-9）	传入的 rknn_output 对象无效
RKNN_ERR_DEVICE_UNMATCH（-10）	版本不匹配
RKNN_ERR_INCOMPATILE_OPTIMIZATION_LEVEL_VERSION（-12）	RKNN 模型设置了优化等级的选项，但是和当前驱动不兼容
RKNN_ERR_TARGET_PLATFORM_UNMATCH（-13）	RKNN 模型和当前平台不兼容

4.5 NPU SDK 说明

4.5.1 SDK 目录说明

RK356X 和 RK3588 平台 NPU SDK 包含了 API 使用示例程序、NPU 运行库、服务程序、文档。服务程序称为 rknn_server，是在开发板上常驻的服务进程，用于连板推理。开发者使用 USB 连接开发板并在 PC 上使用 rknn-toolkit2 的 python 接口运行模型时，依赖服务进程建立通信，安装使用方式参考 rknn_server_proxy.md。整体目录层次结构如下：

```
├── doc
├── examples                                # 示例程序
│   ├── 3rdparty
│   ├── rknn_api_demo                       # 零拷贝 API 示例程序
│   ├── rknn_mobilenet_demo
│   ├── rknn_multiple_input_demo
│   ├── rknn_ssd_demo
│   └── rknn_yolov5_demo
├── README.md
├── rknn_server_proxy.md                    # 板端服务程序说明（用于连板推理）
└── runtime
    ├── RK356X                              # RK356X 运行库和板端服务程序目录
    │   ├── Android
    │   │   ├── librknn_api
    │   │   └── rknn_server
    │   └── Linux
    │       ├── librknn_api
    │       └── rknn_server
    └── RK3588                              # RK3588 运行库和板端服务程序目录
        ├── Android
        │   ├── librknn_api
        │   └── rknn_server
        └── Linux
            ├── librknn_api
            └── rknn_server
    └── RV1106                              # RV1106/RV1103 运行库
        └── Linux
            └── librknn_api
```

4.6 调试方法

4.6.1 日志等级

NPU 的运行库会根据开发板上的系统环境变量输出一些日志信息或者生成文件，用于开发者调试和查看。日志等级设置方法和参数说明如下：

```
export RKNN_LOG_LEVEL=<level_number>
```

<level_number>代表日志等级，可以是 0-5 之间的数字，数字越大代表输出信息越多。不同数字对应的日志信息如下：

表-7 不同日志等级对应的输出信息

日志等级	输出信息
0	仅打印错误日志
1	打印错误和警告的日志
2	打印提示日志以及等级 1 日志
3	打印调试日志以及等级 2 日志
4	打印等级 3 日志以及打印每层信息，开启后会影响 rknn_run 接口的性能
5	打印等级 4 日志以及导出中间层数据，开启后会影响 rknn_run 接口的性能

举个例子，查看逐层性能信息，可以设置如下的环境变量：

```
export RKNN_LOG_LEVEL=4
```

当调试完成以后，可以恢复默认日志等级，恢复命令如下：

```
unset RKNN_LOG_LEVEL
```

4.6.2 性能调试

4.6.2.1 板端环境排查

通常，板子上的各个单元的频率是动态调频，这种情况下测试出来的模型性能会有波动。为了防止性能测试结果不一致，在性能评估时，建议固定板子上的相关单元的频率再做测试。相关单元的频率查看和设置命令如下：

1. CPU 调试命令

(1) 查看 CPU 频率

```
cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_cur_freq
或
cat /sys/kernel/debug/clk/clk_summary | grep arm
```

(2) 固定 CPU 频率（**需固件支持**）

```
# 查看 CPU 可用频率
cat /sys/devices/system/cpu/cpufreq/policy0/scaling_available_frequencies
408000 600000 816000 1008000 1200000 1416000 1608000 1704000
# 设置 CPU 频率，例如，设置 1.7GHz
echo userspace > /sys/devices/system/cpu/cpufreq/policy0/scaling_governor
echo 1704000 > /sys/devices/system/cpu/cpufreq/policy0/scaling_setspeed
```

2. 内存（DDR）调试命令

(1) 查看内存（DDR）频率

```
cat /sys/class/devfreq/dmc/cur_freq
或
cat /sys/kernel/debug/clk/clk_summary | grep ddr
```

(2) 固定内存（DDR）频率（**需固件支持**）

```
# 查看 DDR 可用频率
cat /sys/class/devfreq/dmc/available_frequencies
# 设置 DDR 频率，例如，设置 1560MHz
echo userspace > /sys/class/devfreq/dmc/governor
echo 1560000000 > /sys/class/devfreq/dmc/userspace/set_freq
```

3. NPU 调试命令

(1) 查看 NPU 频率

对于 RK356X:

```
cat /sys/kernel/debug/clk/clk_summary | grep npu
或
cat /sys/class/devfreq/fde40000.npu/cur_freq
```

对于 RK3588（**需固件支持**）:

```
cat /sys/class/devfreq/fdab0000.npu/cur_freq
```

对于 RV1106/RV1103:

```
cat /sys/kernel/debug/clk/clk_summary | grep npu
```

(2) 固定 NPU 频率

注意：在 npu 驱动 0.7.2 版本之后，需要先打开 npu 电源，才能进行频率设置

对于 RK356X:

```
# 查看 NPU 可用频率
cat /sys/class/devfreq/fde40000.npu/available_frequencies
# 设置 NPU 频率，例如，设置 1 GHz
echo userspace > /sys/class/devfreq/fde40000.npu/governor
echo 1000000000 > /sys/kernel/debug/clk/clk_scmi_npu/clk_rate
```

对于 RK3588（**需固件支持**）:

```
# 设置 NPU 频率，例如，设置 1GHz
echo 1000000000 > /sys/kernel/debug/clk/clk_npu_dsu0/clk_rate
```

对于 RV1106/RV1103（**不支持修改频率**）

4.6.2.2 NPU 支持查询设置项

NPU 驱动版本在 **0.7.2** 之后的，可通过节点查询 NPU 的版本、NPU 不同核心的利用率以及手动开关 npu 电源

（1）查询 NPU 驱动版本

```
cat /sys/kernel/debug/rknpu/driver_version
或
cat /proc/debug/rknpu/driver_version
```

（2）查询 NPU 利用率

```
cat /sys/kernel/debug/rknpu/load
或
cat /proc/debug/rknpu/load
```

（3）查询 NPU 电源状态

```
cat /sys/kernel/debug/rknpu/power
```

（4）打开 NPU 电源

```
echo on > /sys/kernel/debug/rknpu/power
```

（5）关闭 NPU 电源

```
echo offs > /sys/kernel/debug/rknpu/power
```