密级状态：绝密(　　)　　秘密(　　)　　内部(　　)　　公开(√)

RKLLM SDK 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：	1.2.2
	作　　者：	HPC
	完成日期：	2025-9-29
	审　　核：	熊伟
	完成日期：	2025-9-29

瑞芯微电子股份有限公司

Rockchip Electronics Co., Ltd

更新记录

版本	修改人	修改日期	修改说明	核定人
V1.0.0	AI 组	2024-3-23	初始版本	熊伟
V1.0.1	AI 组	2024-5-8	1. 优化内存占用 2. 优化推理耗时 3. 优化量化精度 4. 增加 Gemma, Phi-3 等模型支持 5. 增加 Server 调用及中断接口	熊伟
V1.1.0	AI 组	2024-10-10	1. 支持分组量化 2. 支持加载 lora 模型联合推理 3. 支持 prompt_cache 的存储和预加载 4. 优化 init, prefill, decode 耗时 5. 支持 gguf 格式模型的转换 6. 添加 gdq 算法提升 4bit 量化精度 7. 增加分组和非分组混合量化 8. 增加 Llama3, Gemma2, MiniCPM3 等模型支持	熊伟
V1.1.4	AI 组	2025-2-7	1. 支持 GPTQ 模型转换 2. 增加 TeleChat2, MiniCPM-S 模型支持 3. 增加 update_rkllm 接口 4. 支持 MiniCPMV 和 Qwen2VL 中 llm 模型转换 5. 修复 MiniCPM3 等模型推理问题 6. 增加 DeepSeek_R1_Distill_Demo 7. 增加 Qwen2_VL_2B_Demo	熊伟
V1.2.0	AI 组	2025-4-8	1. 支持自定义模型转换 2. 支持 chat_template 配置 3. 支持多轮对话 4. 支持自动 prompt cache 复用 5. 支持 max_context 到 16k	熊伟

版本	修改人	修改日期	修改说明	核定人
			6. 支持 embedding flash 存储，减少内存占用 7. 新增 grq int4 量化算法 8. 支持 GPTQ-int8 模型转换 9. 支持 RK3562 平台 10. 新增支持 InternVL2、Janus、Qwen2.5-VL 等视觉多模态模型 11. 支持 CPU 核配置 12. 新增 Gemma3 支持 13. 新增 Python 3.9/3.11/3.12 支持	
V1.2.1	AI 组	2025-6-25	1. 新增 RWKV7、Qwen3、MiniCPM4 模型支持 2. 新增 RV1126B 平台支持 3. 支持 function calling 功能 4. 支持 cross attention 推理 5. 优化回调函数支持暂停推理 6. 支持多 batch 推理 7. 优化 kv cache 清除接口 8. 优化 chat template 解析，支持 thinking 模式选择 9. Server demo 添加 openai 格式 10. 增加模型推理性能数据返回 11. 支持 mrope 多模态位置编码 12. 新增量化优化算法用于提升量化精度	熊伟
V1.2.2	HPC	2025-9-29	1. 新增 Gemma3n 和 InternVL3 模型支持 2. 支持多实例推理 3. 支持 LongRoPE 4. 解决异步接口推理问题 5. 修复 chat template 解析问题 6. 优化推理性能 7. 优化多模态视觉模型 demo	熊伟

目 录

1 RKLLM 简介

1.1 RKLLM 工具链介绍

1.1.1 RKLLM-Toolkit 功能介绍

RKLLM-Toolkit 是为用户提供在计算机上进行大语言模型的量化、转换的开发套件。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 Hugging Face 和 GGUF 格式的大语言模型（Large Language Model, LLM）转换为 RKLLM 模型，目前支持的模型包括 LLaMA, Qwen, Qwen2, Qwen3, Phi-2, Phi-3, ChatGLM3, Gemma, Gemma2, Gemma3, Gemma3n, InternLM2, TeleChat2, MiniCPM-S, MiniCPM 和 MiniCPM3, MiniCPM4，转换后的 RKLLM 模型能够在 Rockchip NPU 平台上加载使用。

2）量化功能：支持将浮点模型量化为定点模型，目前支持的量化类型包括

a. w4a16;

b. w4a16 分组量化(支持的分组数为 32, 64, 128);

c. w8a8;

d. w8a8 分组量化(支持的分组数为 128, 256, 512);

1.1.2 RKLLM Runtime 功能介绍

RKLLM Runtime 主要负责加载 RKLLM-Toolkit 转换得到的 RKLLM 模型，并在板端通过调用 NPU 驱动在 Rockchip NPU 上加速 RKLLM 模型的推理。在推理 RKLLM 模型时，用户可以自行定义 RKLLM 模型的推理参数设置，定义不同的文本生成方式，并通过预先定义的回调函数不断获得模型的推理结果。

1.2 RKLLM 开发流程介绍

RKLLM 的整体开发步骤主要分为 2 个部分：模型转换和板端部署运行。

1）模型转换：

在这一阶段，用户提供的 Hugging Face 格式的大语言模型将会被转换为 RKLLM 格式，以便在 Rockchip NPU 平台上进行高效的推理。这一步骤包括：

a. 获取原始模型：1、开源的 Hugging Face 格式的大语言模型；2、自行训练得到的大语言模型，要求模型保存的结构与 Hugging Face 平台上的模型结构一致；3、GGUF 模型，目前仅支持 q4_0 和 fp16 类型模型；

b. 模型加载：通过 rkllm.load_huggingface()函数加载 huggingface 格式模型，通过 rkllm.load_gguf()函数加载 GGUF 模型；

c. 模型量化配置：通过 rkllm.build() 函数构建 RKLLM 模型，在构建过程中可选择是否进行模型量化来提高模型部署在硬件上的性能，以及选择不同的优化等级和量化类型。

d. 模型导出：通过 rkllm.export_rkllm() 函数将 RKLLM 模型导出为一个.rkllm 格式文件，用于后续的部署。

2）板端部署运行：

这个阶段涵盖了模型的实际部署和运行。它通常包括以下步骤：

a. 模型初始化：加载 RKLLM 模型到 Rockchip NPU 平台，进行相应的模型参数设置来定义所需的文本生成方式，并提前定义用于接受实时推理结果的回调函数，进行推理前准备。

b. 模型推理：执行推理操作，将输入数据传递给模型并运行模型推理，用户可以通过预先定义的回调函数不断获取推理结果。

c. 模型释放：在完成推理流程后，释放模型资源，以便其他任务继续使用 NPU 的计算资源。

以上这两个步骤构成了完整的 RKLLM 开发流程，确保大语言模型能够成功转换、调试，并最终在 Rockchip NPU 上实现高效部署。

1.3 适用的硬件平台

本文档适用的硬件平台主要包括：RK3576、RK3588、RK3562、RV1126B。

2 开发环境准备

在发布的 RKLLM 工具链压缩文件中，包含了 RKLLM-Toolkit 的 whl 安装包、RKLLM Runtime 库的相关文件以及参考示例代码，具体的文件夹结构如下：

```
doc
└────Rockchip_RKLLM_SDK_CN.pdf    # RKLLM SDK 中文说明文档
└────Rockchip_RKLLM_SDK_EN.pdf    # RKLLM SDK 英文说明文档

examples
├────multimodal_model_demo      # 多模态推理调用示例工程
├────rkllm_api_demo             # 板端 API 推理调用示例工程
└────rkllm_server_demo          # RKLLM-Server 部署示例工程

rkllm-runtime
|    └──── Android
|         └──── librkllm_api
|              └────arm64-v8a
|                   └──── librkllmrt.so  # 64 位 Runtime 库
|              └────armeabi-v7a
|                   └──── librkllmrt.so   # 32 位 Runtime 库
|              └────include
|                   └──── rkllm.h
|    └──── Linux
|         └──── librkllm_api
|              └────aarch64
|                   └──── librkllmrt.so  # 64 位 Runtime 库
|              └────armhf
|                   └──── librkllmrt.so  # 32 位 Runtime 库
|              └────include
|                   └──── rkllm.h

rkllm-toolkit
|    └──── packages
|         └────rkllm_toolkit-x.x.x-cp3xx-cp3xx-linux_x86_64.whl
|    └──── examples

rknpu-driver
└────rknpu_driver_x.x.x_xxxxxxx.tar.bz2

scripts
├────fix_freq_rk3576.sh   # RK3576 定频脚本
└────fix_freq_rk3588.sh   # RK3588 定频脚本
└────fix_freq_rk3562.sh   # RK3562 定频脚本
└────fix_freq_rv1126b.sh  # RV1126B 定频脚本
```

在本章中将会对 RKLLM-Toolkit 工具及 RKLLM Runtime 的安装进行详细的介绍，具体的使用方法请参考第 3 章中的使用说明。

2.1 RKLLM-Toolkit 安装

本节主要说明如何通过 pip 方式来安装 RKLLM-Toolkit，用户可以参考以下的具体流程说明完成 RKLLM-Toolkit 工具链的安装。

2.1.1 通过 pip 方式安装

2.1.1.1 安装 miniforge3 工具

为防止系统对多个不同版本的 Python 环境的需求，建议使用 miniforge3 管理 Python 环境。

检查是否安装 miniforge3 和 conda 版本信息，若已安装则可省略此小节步骤。

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

下载 miniforge3 安装包

```
wget -c https://mirrors.bfsu.edu.cn/github-release/conda-
forge/miniforge/LatestRelease/Miniforge3-Linux-x86_64.sh
```

安装 miniforge3

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

2.1.1.2 创建 RKLLM-Toolkit Conda 环境

进入 Conda base 环境

```
source ~/miniforge3/bin/activate # miniforge3 为安装目录
# (base) xxx@xxx-pc:~$
```

创建一个 Python3.8 版本（建议版本）名为 RKLLM-Toolkit 的 Conda 环境

```
conda create -n RKLLM-Toolkit python=3.8
```

进入 RKLLM-Toolkit Conda 环境

```
conda activate RKLLM-Toolkit
# (RKLLM-Toolkit) xxx@xxx-pc:~$
```

2.1.1.3 安装 RKLLM-Toolkit

在 RKLLM-Toolkit Conda 环境下使用 pip 工具直接安装所提供的工具链 whl 包，在安装过程中，安装工具会自动下载 RKLLM-Toolkit 工具所需要的相关依赖包。

```
pip3 install rkllm_toolkit-x.x.x-cp3xx-cp3xx-linux_x86_64.whl
```

若执行以下命令没有报错，则安装成功。

```
python
from rkllm.api import RKLLM
```

2.2 RKLLM Runtime 库的使用

在所发布的的 RKLLM 工具链文件中，包括包含 RKLLM Runtime 的全部文件：

1）librkllmrt.so：适用于板端进行模型推理的 RKLLM Runtime 库；

2）include/rkllm.h：与 librkllmrt.so 相对应的头文件，包含相关结构体及函数定义的说明；

在通过 RKLLM 工具链构建板端的部署推理代码时，需要注意对以上头文件及函数库的链接，从而保证编译的正确性。当代码在板端实际运行的过程中，同样需要确保以上函数库文件成功推送至板端，并通过以下环境变量设置完成函数库的声明：

```
export LD_LIBRARY_PATH=/path/to/your/lib
```

2.3 RKLLM Runtime 的编译要求

在使用 RKLLM Runtime 的过程中，需要注意 gcc 编译工具的版本。推荐使用交叉编译工具 gcc-arm-10.2-2020.11-x86_64-aarch64-none-linux-gnu；下载路径为：GCC_10.2 交叉编译工具下载地址。请注意，交叉编译工具往往向下兼容而无法向上兼容，因此不要使用 10.2 以下的版本。

若是选择使用 Android 平台，需要进行 Android 可执行文件的编译，推荐使用 Android NDK 工具进行交叉编译，下载路径为：Android_NDK 交叉编译工具下载地址，推荐使用 r21e 版本。

具体的编译方式也可以参考 examples/rkllm_api_demo/deploy 目录下的编译脚本。

2.4 芯片内核更新

由于所提供的 RKLLM 所需要的 NPU 内核版本较高，用户在板端使用 RKLLM Runtime 进行模型推理前，首先需要确认板端的 NPU 内核是否为 v0.9.8 版本，具体的查询命令如下：

```
# 板端执行以下命令，查询 NPU 内核版本
cat /sys/kernel/debug/rknpu/version

# 确认命令输出是否为：
# RKNPU driver: v0.9.8
```

若所查询的 NPU 内核版本低于 v0.9.8，请前往官方固件地址下载最新固件进行更新；

若用户所使用的为非官方固件，需要对内核进行更新；其中，RKNPU 驱动包支持两个主要

内核版本：kernel-5.10 和 kernel-6.1；对于 kernel-5.10，建议使用具体版本号 5.10.209，内核地址为 GitHub-rockchip-linux/kernelatdevelop-5.10；对于 kernel-6.1，建议使用具体版本号 6.1.84；用户可在内核根目录下的 Makefile 中确认具体版本号。内核的具体的更新步骤如下：

1) 下载压缩包 rknpu_driver_0.9.8_20241009.tar.bz2。

2) 解压该压缩包，将其中的 rknpu 驱动代码覆盖到当前内核代码目录(kernel/drivers/rknpu)。

3) 重新编译内核。

4) 将新编译的内核烧录到设备中。

补充说明：

若是发现编译内核的过程中，出现如图 2-1 的错误日志：

```
drivers/rknpu/rknpu_gem.c: In function 'rknpu_gem_mmap_pages':
drivers/rknpu/rknpu_gem.c:891:2: error: implicit declaration of function 'vm_flags_set' [-Werror=implicit-function-declaration]
  891 |  vm_flags_set(vma, VM_MIXEDMAP);
      |  ^~~~~~~~~~~
drivers/rknpu/rknpu_gem.c: In function 'rknpu_gem_mmap_buffer':
drivers/rknpu/rknpu_gem.c:988:2: error: implicit declaration of function 'vm_flags_clear' [-Werror=implicit-function-declaration]
  988 |  vm_flags_clear(vma, VM_PFNMAP);
      |  ^~~~~~~~~~~~~
cc1: all warnings being treated as errors
make[2]: *** [scripts/Makefile.build:273: drivers/rknpu/rknpu_gem.o] Error 1
make[1]: *** [scripts/Makefile.build:516: drivers/rknpu] Error 2
make[1]: *** Waiting for unfinished jobs....
  AR      drivers/iio/built-in.a
make: *** [Makefile:1929: drivers] Error 2
  MAKE KERNEL IMAGE FAILED.
```

图 2-1 内核编译错误

用户需要修改内核头文件 kernel/include/linux/mm.h，加入以下代码：

```
static inline void vm_flags_set(struct vm_area_struct *vma,
                                vm_flags_t flags)
{
      vma->vm_flags |= flags;
}

static inline void vm_flags_clear(struct vm_area_struct *vma,
                                  vm_flags_t flags)
{
      vma->vm_flags &= ~flags;
}
```

3 RKLLM 使用说明

3.1 模型转换

RKLLM-Toolkit 提供模型的转换、量化功能。作为 RKLLM-Toolkit 的核心功能之一，它允许用户将 Hugging Face 或 GGUF 格式的大语言模型转换为 RKLLM 模型，从而将 RKLLM 模型在 Rockchip NPU 上运行。本节介绍 RKLLM-Toolkit 对 LLM 模型的具体转换实现，以供用户参考。

3.1.1 RKLLM 初始化

在这一部分，用户需要先初始化 RKLLM 对象，这是整个工作流的第一步。在示例代码中使用 RKLLM()构造函数来初始化 RKLLM 对象。

3.1.2 模型加载

在 RKLLM 初始化完成后，用户需要调用 rkllm.load_huggingface()函数来传入模型的具体路径，RKLLM-Toolkit 即可根据对应路径顺利加载 Hugging Face 或 GGUF 格式的大语言模型，从而顺利完成后续的转换、量化操作，具体的函数定义如下：

表 3-1 load_huggingface 函数接口说明

函数名	load_huggingface
描述	用于加载开源的 Hugging Face 格式的大语言模型。
参数	***model:*** LLM 模型的文件路径，用于加载模型进行后续的转换、量化； ***model_lora:*** lora 权重的文件路径，转换时 model 必须指向相应的 base model 路径； ***device:*** 指定模型转换时使用的设备，支持 cuda 和 cpu 两个选项； ***dtype:*** 权重的数据类型，可选值包括 float32，float16 和 bfloat16；float16 和 bfloat16 可以减少显存占用，但会损失量化精度； ***custom_config:*** 自定义模型的配置文件，具体详见自定义模型转换章节； ***load_weight:*** 是否加载权重，设置为 False 时不会加载真实权重；
返回值	**0** 表示模型加载正常；**-1** 表示模型加载失败；

示例代码如下：

```
ret = rkllm.load_huggingface(
    model = './huggingface_model_dir',
    model_lora = './huggingface_lora_model_dir'
    )
if ret != 0:
    print('Load model failed!')
```

表 3-2 load_gguf 函数接口说明

函数名	load_gguf
描述	用于加载开源的 GGUF 格式的大语言模型，所支持的数值类型为 q4_0 和 fp16 两种，gguf 格式的 lora 模型也可以通过此接口加载转换为 rkllm 模型。
参数	***model:*** GGUF 模型文件路径；
返回值	**0** 表示模型加载正常；**-1** 表示模型加载失败；

示例代码如下：

```
ret = rkllm.load_gguf(model = './model-Q4_0.gguf')
if ret != 0:
    print('Load model failed!')
```

3.1.3 模型构建

用户在通过 rkllm.load_huggingface()函数完成原始模型的加载后，下一步就是通过 rkllm.build()函数实现对 RKLLM 模型的构建。构建模型时，用户可以选择是否进行量化，量化有助于减小模型的大小和提高在 Rockchip NPU 上的推理性能。rkllm.build()函数的具体定义如下：

表 3-3 build 函数接口说明

函数名	build
描述	用于构建得到 RKLLM 模型，并在转换过程中定义具体的量化操作。
参数	***do_quantization:*** 该参数控制是否对模型进行量化操作，建议设置为 True； ***optimization_level:*** 该参数用于设置是否进行量化精度优化，可选择的设置为{0, 1}，0 表示不做任何优化，1 表示进行精度优化，精度优化可能造成模型推理性能下降； ***quantized_dtype:*** 该参数用于设置量化的具体类型，目前支持的量化类型包括“w4a16”，“w4a16_g32”，“w4a16_g64”，“w4a16_g128”，“w8a8”，“w8a8_g128”，“w8a8_g256”，“w8a8_g512”，“w4a16”表示对权重进行 4bit 量化而对激活值不进行量化；“w4a16_g64”表示对权重进行 4bit 分组量化(group size=64)而对激活值不进行量化；“w8a8”表示对权重和激活值均进行 8bit 量化；“w8a8_g128”表示对权重和激活值均进行 8bit 分组量化(group size=128)；目前 RK3576 和 RV1126B 平台支持“w4a16”，“w4a16_g32”，“w4a16_g64”，“w4a16_g128”和“w8a8”五种量化类型，RK3588 支持“w8a8”，“w8a8_g128”，“w8a8_g256”，“w8a8_g512”四种量化类型，RK3562 支持“w8a8”，“w4a16_g32”，“w4a16_g64”，“w4a16_g128”和“w4a8_g32”量化类型；GGUF 模型的 q4_0 对应的量化类型为“w4a16_g32”；注意 group size 应能被线性层的输出维度整除，否则不支持! ***quantized_algorithm:*** 量化精度优化算法，可选择的设置包括“normal”、“grq”或“gdq”，所有量化类型均可选择 normal，而 gdq 和 grq 算法只支持 w4a16 及 w4a16 分组量化，且 gdq 和 grq 对算力要求高，必须使用 GPU 进行加速运算；grq 相比 gdq 算法速度更快且显存占用更低，4 bit 量化可优先尝试 grq 算法；支持利用 lora 模块进一步提高量化精度，在无其他 lora 模型导入的情况下，可以使用“normal_r[1~64]”、“grq_r[1~64]”或“gdq_r[1~64]”算法，例如“normal_r8”、“grq_r8”或“gdq_r8”，最终将会导出量化模型与 lora 模型，runtime 加载两个模型执行推理。 ***num_npu_core:*** 模型推理需要使用的 npu 核心数，“RK3576”可选项为[1,2],

	“RK3588”可选项为[1,2,3]，“RK3562”可选项为[1]，“RV1126B”可选项为[1]； ***extra_qparams:*** 使用 gdq/grq 算法会生成.qparams 后缀的量化权重缓存文件，将此参数设置为*.qparams 路径，可以重新进行模型导出； ***dataset:*** 用于量化校正数据集，格式为 json，内容示例如下，input 为问题，需要加上提示词，target 为回答，多条数据以{}字典形式保存在列表中： [{"input":"今天天气怎么样？","target":"今天天气晴。"},....] ***hybrid_rate:*** 分组和不分组混合量化比率(∈[0,1))，当量化类型为 w4a16/w8a8 时，会按比率分别混合 w4a16 分组/w8a8 分组类型来提高精度，当量化类型为 w4a16 分组/w8a8 分组类型时，会按比率分别混合 w4a16/w8a8 类型来提高推理性能，当 hybrid_rate 值为 0 时，不进行混合量化； ***target_platform:*** 模型运行的硬件平台，可选择的设置包括“RK3576”、“RK3588”、“RK3562”、“RV1126B”； ***max_context:*** 上下文长度的上限值，最大支持到 16384 且必须按 32 对齐；
返回值	**0** 表示模型转换、量化正常；**-1** 表示模型转换失败；

示例代码如下：

```
ret = rkllm.build(
       do_quantization=True,
       optimization_level=1,
       quantized_dtype='w8a8',
       quantized_algorithm="normal",
       num_npu_core=3,
       extra_qparams=None,
       dataset="quant_data.json",
       hybrid_rate=0,
       target_platform='rk3588')
if ret != 0:
   print('Build model failed!')
```

3.1.4 模型导出

用户在通过 rkllm.build()函数构建了 RKLLM 模型后，可以通过 rkllm.export_rkllm()函数将 RKNN 模型保存为一个.rkllm 文件，以便后续模型的部署。rkllm.export_rkllm()函数的具体参数定义如下：

表 3-4 export_rkllm 函数接口说明

函数名	export_rkllm
描述	用于保存转换、量化后的 RKLLM 模型，用于后续的推理调用。
参数	***export_path:*** 导出 RKLLM 模型文件的保存路径，lora 模型会自动保存为带_lora 后缀的 rkllm 模型；
返回值	**0** 表示模型成功导出保存；**-1** 表示模型导出失败；

示例代码如下：

```
ret = rkllm.export_rkllm(export_path = './model.rkllm')
if ret != 0:
    print('Export model failed!')
```

3.1.5 GPTQ 模型转换

用户除了使用上述工具中提供的量化算法进行模型转换，也可以先使用 AutoGPTQ 开源量化工具将浮点模型量化为 4bit/8bit 权重（需保存为 Hugging Face 格式），再转换为 RKLLM 模型。使用 AutoGPTQ 量化浮点模型时，需确保以下参数设置：

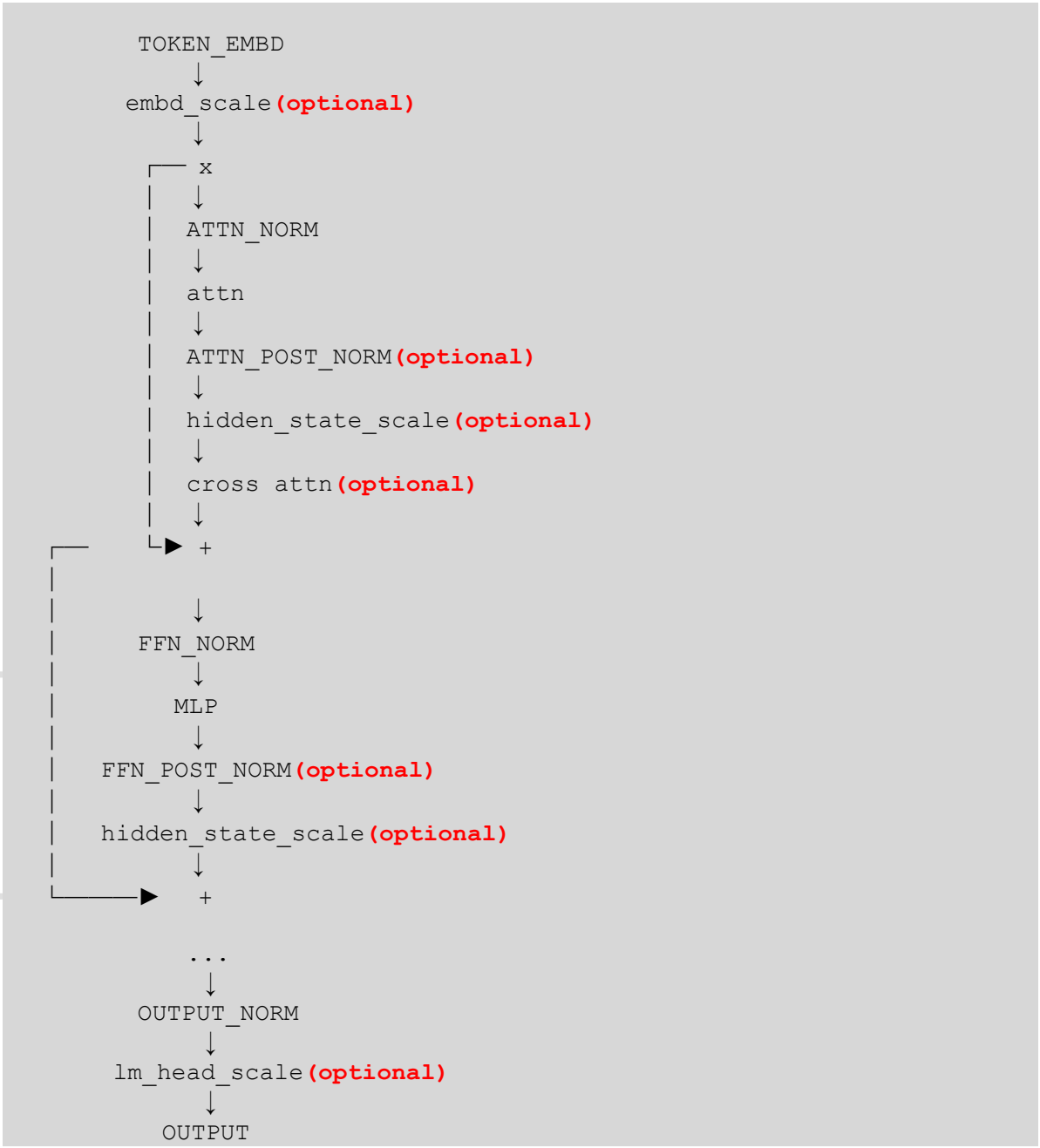
```
bits=4
sym=true
group_size=32/64/128
desc_act=false

bits=8
sym=true
group_size=128/256/512
desc_act=false
```

Hugging Face 格式的 GPTQ 模型转换为 rkllm 的示例代码如下：

```
modelpath = '/path/to/Model-Instruct-GPTQ-Int4'
llm = RKLLM()

ret = llm.load_huggingface(model=modelpath, model_lora = None,
device='cuda')
if ret != 0:
    print('Load model failed!')
    exit(ret)

# Build model
dataset = None
qparams = None
target_platform = "RK3576"
optimization_level = 1
quantized_dtype = "w4a16_g32" #w4a16_g64 or w4a16_g128
quantized_algorithm = "normal"
num_npu_core = 2

ret = llm.build(do_quantization=True,
optimization_level=optimization_level, quantized_dtype=quantized_dtype,
            quantized_algorithm=quantized_algorithm,
target_platform=target_platform, num_npu_core=num_npu_core,
extra_qparams=qparams, dataset=dataset)
if ret != 0:
    print('Build model failed!')
    exit(ret)

# Export rkllm model
ret =
llm.export_rkllm(f"./{os.path.basename(modelpath)}_{quantized_dtype}_{
target_platform}.rkllm")
if ret != 0:
    print('Export model failed!')
    exit(ret)
```

3.1.6 自定义模型转换

用户如果修改了模型结构或者名称，且修改后的整体架构如下，则可以使用自定义功能转换模型。



以 Qwen 模型为例，将 Qwen 模型文件 modeling_qwen.py 中相应的变量名称填入自定义配置文件中，如下所示：

```
{
    "BLOCKNAME": "QWenBlock",
    "TOKEN_EMBD": "wte",
    "ATTN_NORM": "ln_1",
    "ATTN_Q_NORM": "",
    "ATTN_K_NORM": "",
    "CROSS_ATTN_NORM": "",
    "CROSS_ATTN_Q": "",
    "ATTN_Q": "",
    "ATTN_K": "",
    "ATTN_V": "",
    "ATTN_QKV": "attn.c_attn",
    "ATTN_KV": "",
    "KV_CONTINUOUS": "true",
    "ATTN_OUT": "attn.c_proj",
    "CROSS_ATTN_OUT": "",
    "ATTN_POST_NORM": "",
    "FFN_NORM": "ln_2",
    "FFN_UP": "mlp.w1",
    "FFN_GATE": "mlp.w2",
    "ACT_TYPE": "silu",
    "FFN_DOWN": "mlp.c_proj",
    "FFN_POST_NORM": "",
    "OUTPUT_NORM": "ln_f",
    "OUTPUT": "lm_head"
}
```

其中，ACT_TYPE 可选项为["silu", "gelu", "relu", "fatrelu", "squarerelu", "swiglu"]共六种，ATTN_NORM 和 FFN_NORM 只支持 RMSNorm；

如果使用了 ATTN_QKV 或者 ATTN_KV，则必须确认权重是否可以拆分为连续的 K|V，如果连续则 KV_CONTINUOUS 设置为 true，例如 Qwen 模型的 c_attn 权重是连续存储，它可以按 Q,K,V 大小进行顺序切分，而 InternLM2 模型的 wqkv 权重是不连续存储，它的 Q,K,V 是按 head_dim 交错排布。

Qwen 1.8B 模型的 modeling_qwen.py 文件定义如下：

```
class QWenLMHeadModel(QWenPreTrainedModel):
     def __init__(self, config):
       super().__init__(config)
          self.transformer = QWenModel(config)
          self.lm_head = nn.Linear(config.hidden_size,
config.vocab_size, bias=False)

class QWenModel(QWenPreTrainedModel):
    _keys_to_ignore_on_load_missing = ["attn.masked_bias"]

    def __init__(self, config):
        super().__init__(config)
        self.wte = nn.Embedding(self.vocab_size, self.embed_dim)
        self.ln_f = RMSNorm(
            self.embed_dim,
            eps=config.layer_norm_epsilon,
        )
class QWenBlock(nn.Module):
    def __init__(self, config):
        super().__init__()
        hidden_size = config.hidden_size
        self.bf16 = config.bf16

        self.ln_1 = RMSNorm(
            hidden_size,
            eps=config.layer_norm_epsilon,
        )
        self.attn = QWenAttention(config)
        self.ln_2 = RMSNorm(
            hidden_size,
            eps=config.layer_norm_epsilon,
        )

        self.mlp = QWenMLP(config)
class QWenAttention(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.c_attn = nn.Linear(config.hidden_size, 3 *
self.projection_size)
        self.c_proj = nn.Linear(
            config.hidden_size, self.projection_size, bias=not
config.no_bias
        )
class QWenMLP(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.w1 = nn.Linear(
            config.hidden_size, config.intermediate_size // 2, bias=not
config.no_bias)
        self.w2 = nn.Linear(
            config.hidden_size, config.intermediate_size // 2, bias=not
config.no_bias)
        ff_dim_in = config.intermediate_size // 2
        self.c_proj = nn.Linear(ff_dim_in, config.hidden_size, bias=not
config.no_bias)
```

自定义模型的转换，包括支持 cross attention 的模型，我们提供了一个 huggingface 模型结构供参

考，自定义的配置文件如下：

```
{
    "BLOCKNAME": "CustomDecoderLayer",
    "TOKEN_EMBD": "embed_tokens",
    "ATTN_NORM": "input_layernorm",
    "ATTN_Q_NORM": "",
    "ATTN_K_NORM": "",
    "CROSS_ATTN_NORM": "cross_layernorm",
    "CROSS_ATTN_Q": "cross_attn.cross_q_proj",
    "ATTN_Q": "",
    "ATTN_K": "",
    "ATTN_V": "",
    "ATTN_QKV": "self_attn.qkv_proj",
    "ATTN_KV": "",
    "KV_CONTINUOUS": "true",
    "ATTN_OUT": "self_attn.o_proj",
    "CROSS_ATTN_OUT": "cross_attn.cross_o_proj",
    "ATTN_POST_NORM": "",
    "FFN_NORM": "post_attention_layernorm",
    "FFN_UP": "mlp.up_proj",
    "FFN_GATE": "mlp.gate_proj",
    "ACT_TYPE": "silu",
    "FFN_DOWN": "mlp.down_proj",
    "FFN_POST_NORM": "",
    "OUTPUT_NORM": "norm",
    "OUTPUT": "lm_head"
```

3.1.7 旧版本模型更新

由于 1.0.2 版本与 1.1 及之后版本差异较大，因此提供了 rkllm.update_rkllm()函数将 1.0.2 版本模型更新为最新版本，更新模型时无需执行上述模型加载和构建步骤，直接调用此接口进行更新，更新后模型量化类型等参数均未改变。rkllm.update_rkllm()函数的具体参数定义如下：

表 3-5 update_rkllm 函数接口说明

函数名	update_rkllm
描述	将 1.0.2 版本模型更新为最新版本
参数	***model:*** 1.0.2 版本的 rkllm 模型路径
返回值	**0** 表示模型更新成功；**-1** 表示模型更新失败；

示例代码如下：

```
ret = llm.update_rkllm(model = "./model_1.0.2version.rkllm")
if ret != 0:
    print('Load model failed!')
    exit(ret)
```

3.1.8 仿真精度评估

用户在通过 rkllm.build()函数构建了 RKLLM 模型后，可以通过 rkllm.get_logits()函数在 PC 端进行仿真精度评估，rkllm.get_logits()函数的具体参数定义如下：

表 3-6 get_logits 函数接口说明

函数名	get_logits
描述	用于 PC 端仿真精度评估。
参数	***inputs:*** 仿真输入格式与 huggingface 模型推理一样，示例如下： {“input_ids":"","top_k":1,...}
返回值	返回模型推理出的 logits 值；

使用此函数进行 wikitext 数据集 ppl 测试示例代码如下：

```
def eval_wikitext(llm):
    seqlen = 512
    tokenizer = AutoTokenizer.from_pretrained(
        modelpath,
        trust_remote_code=True
        )
    #Dataset download link:
    #https://huggingface.co/datasets/Salesforce/wikitext/tree/main/wiki
text-2-raw-v1
    testenc = load_dataset("parquet", data_files='./wikitext/wikitext-
2-raw-1/test-00000-of-00001.parquet', split='train')
    testenc = tokenizer(
        "\n\n".join(testenc['text']),
        return_tensors="pt").input_ids
    nsamples = testenc.numel() // seqlen
    nlls = []
    for i in tqdm(range(nsamples), desc="eval_wikitext: "):
        batch = testenc[:, (i * seqlen): ((i + 1) * seqlen)]
        inputs = {"input_ids": batch}
        lm_logits = llm.get_logits(inputs)
        if lm_logits is None:
            print("get logits failed!")
            return
        shift_logits = lm_logits[:, :-1, :]
        shift_labels = batch[:, 1:].to(lm_logits.device)
        loss_fct = nn.CrossEntropyLoss().to(lm_logits.device)
        loss = loss_fct(shift_logits.view(-1, shift_logits.size(-1)),
shift_labels.view(-1))
        neg_log_likelihood = loss.float() * seqlen
        nlls.append(neg_log_likelihood)
    ppl = torch.exp(torch.stack(nlls).sum() / (nsamples * seqlen))
    print(f'wikitext-2-raw-1-test ppl: {round(ppl.item(), 2)}')
```

3.1.9 仿真模型推理

用户在通过 rkllm.build()函数构建了 RKLLM 模型后，可以通过 rkllm.chat_model()函数在 PC 端进行仿真推理，rkllm.chat_model()函数的具体参数定义如下：

表 3-7 chat_model 函数接口说明

函数名	chat_model
描述	用于 PC 端仿真模型推理。
参数	***messages:*** 文本输入，需要加上相应提示词 ***args:*** 推理配置参数，比如 topk 等采样策略参数
返回值	返回模型推理结果；

示例代码如下：

```
args ={
    "max_length":128,
    "top_k":1,
    "temperature":0.8,
    "do_sample":True,
    "repetition_penalty":1.1
    }
mesg = "Human: 今天天气怎么样？\nAssistant:"
print(llm.chat_model(mesg, args))
```

以上的这些操作涵盖了 RKLLM-Toolkit 模型转换、量化的全部步骤，根据不同的需求和应用场景，用户可以选择不同的配置选项和量化方式进行自定义设置，方便后续进行部署。

3.2 板端推理

此章节介绍板端 Runtime 库 API 接口函数的调用流程，用户可以参考本节内容使用 C++构建代码，在板端实现对 RKLLM 模型的推理，获取推理结果。RKLLM 板端推理实现的调用流程如下：

1） 定义回调函数 callback()；

2） 定义 RKLLM 模型参数结构体 RKLLMParam；

3） rkllm_init()初始化 RKLLM 模型；

4） rkllm_run()进行模型推理；

5） 通过回调函数 callback()对模型实时传回的推理结果进行处理；

6） rkllm_destroy()销毁 RKLLM 模型并释放资源；

在本节的后续部分，该文档将详细介绍流程的各环节，并对其中的函数进行详细说明。

3.2.1 回调函数定义

回调函数是用于接收模型实时输出的结果。在初始化 RKLLM 时回调函数会被绑定，在模型推理过程中不断将结果输出至回调函数中，并且每次回调只返回一个 token。

示例代码如下，该回调函数将输出结果实时地打印输出到终端中：

```
int callback(RKLLMResult* result, void* userdata, LLMCallState state)
{
    if(state == LLM_RUN_NORMAL){
        printf("%s", result->text);
    }
    if (state == LLM_RUN_FINISH) {
        printf("finish\n");
    } else if (state == LLM_RUN_ERROR){
        printf("\run error\n");
    }
    return 0;
}
```

1）LLMCallState 是一个状态标志，其具体定义如下：

表 3-8 LLMCallState 状态标志说明

枚举定义	LLMCallState
描述	用于表示当前 RKLLM 的运行状态。
枚举值	***LLM_RUN_NORMAL***：表示 RKLLM 模型当前正在推理中； ***LLM_RUN_FINISH***：表示 RKLLM 模型已完成当前输入的全部推理； ***LLM_RUN_WAITING***：表示当前 RKLLM 解码出的字符不是完整 UTF8 编码，需等待与下一次解码拼接； ***LLM_RUN_ERROR***：表示 RKLLM 模型推理出现错误；

用户在回调函数的设计过程中，可以根据 LLMCallState 的不同状态设置不同的后处理行为。

3）RKLLMResult 是返回值结构体，其具体定义如下：

表 3-9 RKLLMResult 返回值结构体说明

结构体定义	RKLLMResult
描述	用于返回当前推理生成结果。
字段	***const char* text:*** 表示当前推理生成的文本内容； ***int32_t token_id:*** 表示当前推理生成的 token id； ***RKLLMResultLogits logits:*** 表示当前推理生成的 logits 信息，仅在 RKLLMInferMode 设置为 RKLLM_INFER_GET_LOGITS 时返回； ***RKLLMPerfStat perf:*** 表示当前推理结束时的性能统计数据，仅在 RKLLM_RUN_FINISH 状态时返回数据；

4）RKLLMResultLogits 是返回值结构体，其具体定义如下：

表 3-9 RKLLMResultLogits 返回值结构体说明

结构体定义	RKLLMResultLogits
描述	用于返回当前推理生成的 logits。
字段	***const float* logits:*** 表示当前推理生成的 logits； ***int vocab_size:*** 表示词表大小； ***int num_tokens:*** 表示返回的 token 数，用户可根据 vocab_size 与 num_tokens 计算出 logits 大小；

用户在回调函数的设计过程中，可以根据 RKLLMResult 中值设置不同的后处理行为。

3.2.2 参数结构体 RKLLMParam 定义

结构体 RKLLMParam 用于描述、定义 RKLLM 的详细信息，具体的定义如下：

表 3-10 RKLLMParam 结构体参数说明

结构体定义	RKLLMParam
描述	用于定义 RKLLM 模型的各项细节参数。
字段	***const char* model_path:*** 模型文件的存放路径； ***int32_t max_context_len:*** 设置推理时的最大上下文长度； ***int32_t max_new_tokens:*** 用于设置模型推理时生成 token 的数量上限； ***int32_t top_k:*** top-k 采样是一种文本生成方法，它仅从模型预测的概率最高的 k 个 token 中选择下一个 token。该方法有助于降低生成低概率或无意义 token 的风险。更高的值（如 100）将考虑更多的 token 选择，导致文本更加多样化；而更低的值（如 10）将聚焦于最可能的 token，生成更加保守的文本。默认值为 40； ***float top_p:*** top-p 采样，也被称为核心采样，是另一种文本生成方法，从累计概率至少为 p 的一组 token 中选择下一个 token。这种方法通过考虑 token 的概率和采样的 token 数量在多样性和质量之间提供平衡。更高的值（如 0.95）使得生成的文本更加多样化；而更低的值（如 0.5）将生成更加保守的文本。默认值为 0.9； ***float temperature:*** 控制生成文本随机性的超参数，它通过调整模型输出 token 的概率分布来发挥作用；更高的温度（如 1.5）会使输出更加随机和创造性，当温度较高时，模型在选择下一个 token 时会考虑更多可能性较低的选项，从而产生更多样和意想不到的输出；更低的温度（例 0.5）会使输出更加集中、保守，较低的温度意味着模型在生成文本时更倾向于选择概率高的 token，从而导致更一致、更可预测的输出；温度为 0 的极端情况下，模型总是选择最有可能的下一个 token，这会导致每次运行时输出完全相同；为了确保随机性和确定性之间的平衡，使输出既不过于单一和可预测，也不过于随机和杂乱，默认值为 0.8； ***float repeat_penalty:*** 控制生成文本中 token 序列重复的情况，帮助防止模型生成重复或单调的文本。更高的值（例如 1.5）将更强烈地惩罚重复，而较低的值（例如

	0.9）则更为宽容。默认值为 1.1； ***float frequency_penalty:*** 单词/短语重复度惩罚因子，减少总体上使用频率较高的单词/短语的概率，增加使用频率较低的单词/短语的可能性，这可能会使生成的文本更加多样化，但也可能导致生成的文本难以理解或不符合预期。设置范围为[-2.0, 2.0]，默认为 0； ***int32_t mirostat:*** 在文本生成过程中主动维持生成文本的质量在期望的范围内的算法，它旨在在连贯性和多样性之间找到平衡，避免因过度重复（无聊陷阱）或不连贯（混乱陷阱）导致的低质量输出；取值空间为{0, 1, 2}, 0 表示不启动该算法，1 表示使用 mirostat 算法，2 则表示使用 mirostat2.0 算法； ***float mirostat_tau:*** 选项设置 mirostat 的目标熵，代表生成文本的期望困惑度。调整目标熵可以控制生成文本中连贯性与多样性的平衡。较低的值将导致文本更加集中和连贯，而较高的值将导致文本更加多样化，可能连贯性较差。默认值是 5.0； ***float mirostat_eta:*** 选项设置 mirostat 的学习率，学习率影响算法对生成文本反馈的响应速度。较低的学习率将导致调整速度较慢，而较高的学习率将使算法更加灵敏。默认值是 0.1； ***bool skip_special_token:*** 是否跳过特殊 token 不输出,例如推理结束符号<EOS>等; ***bool is_async:*** 是否使用异步模式; ***const char* img_start:*** 选项设置多模态输入图像编码的起始标志符，在多模态输入模式下需要配置； ***const char* img_end:*** 选项设置多模态输入图像编码的终止标志符，在多模态输入模式下需要配置； ***const char* img_content:*** 选项设置多模态输入图像编码的内容标志符，在多模态输入模式下需要配置； ***n_keep:*** 清除 kv cache 时需要在开头保留的 cache 数量，多轮对话时所设置的 n_keep 值必须不小于 system_prompt 长度； ***RKLLMExtendParam extend_param:*** 控制推理的特殊参数；

表 3-11 RKLLMExtendParam 结构体参数说明

结构体定义	RKLLMExtendParam
描述	控制推理的特殊参数。
字段	***int32_t base_domain_id:*** 控制 RKLLM 模型从哪个 domain 开始初始化，默认为 0； ***int8_t embed_flash:*** 控制是否将模型词表存在 flash 中来节省内存，0 为关闭，1 为开启 ***int8_t enabled_cpus_num:*** 设置推理时使用的 CPU 数量，不同的芯片型号可设置范围不同。RK3588/3576 的可设置范围为 1~8, RK3562 的可设置范围为 1~4，默认设置为 4 ***uint32_t enabled_cpus_mask:*** 使用二进制掩码配置具体用哪些 CPU 核进行推理。在 rkllm.h 中预设了宏表示 CPU 编号，使用 CPU4|CPU5|CPU6|CPU7 方式配置 ***uint8_t n_batch:*** 设置并行推理，默认设置为 1 ***int8_t use_cross_attn:*** 设置是否启用交叉注意力

在实际的代码构建中，RKLLMParam 需要调用 rkllm_createDefaultParam()函数来初始化定义，并根据需求设置相应的模型参数。示例代码如下：

```
RKLLMParam param = rkllm_createDefaultParam();
param.model_path = "model.rkllm";
param.top_k = 1;
param.max_new_tokens = 256;
param.max_context_len = 512;
```

3.2.3 输入结构体定义

为适应不同的输入数据，定义了 RKLLMInput 输入结构体，目前可接受文本、图片和文本、Token id 以及编码向量四种形式的输入，具体的定义如下：

表 3-12 RKLLMInput 结构体参数说明

结构体定义	RKLLMInput
描述	用于接收不同形式的输入数据。
字段	***RKLLMInputType input_type:*** 输入模式； ***const char* role:*** 输入类型，可选["user", "tool"]； ***bool enable_thinking:*** Qwen3 模型是否使用 thinking 模式； ***union:*** 用于存储不同的输入数据类型，具体包含以下几种形式： ***- const char* prompt_input:*** 文本提示输入，用于传递自然语言文本； ***- RKLLMEmbedInput embed_input:*** 嵌入向量输入，表示已处理的特征向量； ***- RKLLMTokenInput token_input:*** Token 输入，用于传递分词后的 Token 序列； ***- RKLLMMultiModelInput multimodal_input:*** 多模态输入，可传递多模态数据，如图片和文本的联合输入。

表 3-13 RKLLMInputType 输入类型说明

枚举定义	RKLLMInputType
描述	用于表示输入数据类型。
枚举值	***RKLLM_INPUT_PROMPT:*** 表示输入数据是纯文本； ***RKLLM_INPUT_TOKEN:*** 表示输入数据是 Token id； ***RKLLM_INPUT_EMBED:*** 表示输入数据是编码向量； ***RKLLM_INPUT_MULTIMODAL:*** 表示输入数据是图片和文本；

当输入数据是纯文本时，使用 input_data 直接输入；当输入数据是 Token id、编码向量以及图片和文本时，RKLLMInput 需要搭配 RKLLMTokenInput, RKLLMEmbedInput 以及 RKLLMMultiModelInput 三个输入结构体使用，具体的介绍如下：

1）RKLLMTokenInput 是接收 Token id 的输入结构体，具体的定义如下：

表 3-14 RKLLMTokenInput 结构体参数说明

结构体定义	RKLLMTokenInput
描述	用于接收 Token id 数据。
字段	***int32_t* input_ids:** 输入 token ids 的内存指针； ***size_t n_tokens:*** 输入数据的 token 数量；

2）RKLLMEmbedInput 是接收编码向量的输入结构体，具体的定义如下：

表 3-15 RKLLMEmbedInput 结构体参数说明

结构体定义	RKLLMEmbedInput
描述	用于接收 Embedding 数据。
字段	***float* embed:*** 输入 token embedding 的内存指针； ***size_t n_tokens:*** 输入数据的 token 数量；

3）RKLLMMultiModelInput 是接收图片和文本的输入结构体，具体的定义如下：

表 3-16 RKLLMMultiModelInput 结构体参数说明

结构体定义	RKLLMMultiModelInput
描述	用于接收图片和文本多模态数据。
字段	***char* prompt:*** 输入文本的内存指针； ***float* image_embed:*** 输入图片 embedding 的内存指针； ***size_t n_image_tokens:*** 输入图片 embedding 的 token 数量； ***size_t n_image:*** 输入图片数量，支持输入连续多帧图片； ***size_t image_width:*** 输入图片计算 embedding 时的宽度，用于 mrope 计算参数； ***size_t image_height:*** 输入图片计算 embedding 时高度，用于 mrope 计算参数；

RKLLM 支持不同的推理模式，定义了 RKLLMInferParam 结构体，目前可支持在推理过程与预加载的 LoRA 模型联合推理，或保存 Prompt Cache 用于后续推理加速，具体的定义如下：

表 3-17 RKLLMInferParam 结构体参数说明

结构体定义	RKLLMInferParam
描述	用于定义不同的推理模式。
字段	***RKLLMInferMode mode:*** 推理模式，当支持 RKLLM_INFER_GENERATE 普通推理模式与 RKLLM_INFER_GET_LOGITS 额外获取 logits 值的推理模式； ***RKLLMLoraParam* lora_params:*** 推理时使用的 LoRA 的参数配置，用于在加载多个 LoRA 时选择需要推理的 LoRA，若无需加载 LoRA 则设为 NULL 即可； ***RKLLMPromptCacheParam* prompt_cache_params:*** 推理时使用 Prompt Cache 的参数配置，若无需生成 Prompt Cache 则设为 NULL 即可； ***keep_history:*** 推理时是否需要保留历史上下文，当多轮对话时需设置为 1；

表 3-18 RKLLMLoraParam 结构体参数说明

结构体定义	RKLLMLoraParam
描述	用于定义推理时使用 LoRA 的参数；
字段	***const char* lora_adapter_name:*** 推理时使用的 LoRA 名称

表 3-19 RKLLMPromptCacheParam 结构体参数说明

结构体定义	RKLLMPromptCacheParam
描述	用于定义推理时使用 Prompt Cache 的参数；
字段	***int save_prompt_cache:*** 是否在推理时保存 Prompt Cache, 1 为需要, 0 为不需要； ***const char* prompt_cache_path:*** Prompt Cache 保存路径，若未设置则默认保存到"./prompt_cache.bin"中；

使用 RKLLMPromptCacheParam 的示例如下：

```
//初始化并设置 Prompt Cache 参数 (如果需要使用 prompt cache)
RKLLMPromptCacheParam prompt_cache_params;
prompt_cache_params.save_prompt_cache = true; // 是否保存 prompt cache
prompt_cache_params.prompt_cache_path = "./prompt_cache.bin"; // 若需要保存 prompt cache, 指定 cache 文件路径
rkllm_infer_params.prompt_cache_params = &prompt_cache_params;
```

3.2.4 初始化模型

在进行模型的初始化之前，需要提前定义 LLMHandle 句柄，该句柄用于模型的初始化、推理和资源释放过程。注意，正确的模型推理流程需要统一这 3 个流程中的 LLMHandle 句柄对象。

在模型推理前，用户需要通过 rkllm_init()函数完成模型的初始化，具体函数的定义如下：

表 3-20 rkllm_init 函数接口说明

函数名	rkllm_init
描述	用于初始化 RKLLM 模型的具体参数及相关推理设置。
参数	***LLMHandle* handle:*** 将模型注册到相应句柄中，用于后续推理、释放调用； ***RKLLMParam* param:*** 模型定义的参数结构体； ***LLMResultCallback callback:*** 用于接受处理模型实时输出的回调函数；
返回值	**0** 表示初始化流程正常；**-1** 表示初始化失败；

示例代码如下：

```
LLMHandle llmHandle = nullptr;
rkllm_init(&llmHandle, &param, callback);
```

3.2.5 模型推理

用户在完成 RKLLM 模型的初始化流程后，即可通过 rkllm_run()函数进行模型推理，并可以通过初始化时预先定义的回调函数对实时推理结果进行处理；rkllm_run()的具体函数定义如下：

表 3-21 rkllm_run 函数接口说明

函数名	rkllm_run
描述	调用完成初始化的 RKLLM 模型进行结果推理；
参数	***LLMHandle handle:*** 模型初始化注册的目标句柄； ***RKLLMInput* rkllm_input:*** 模型推理的输入数据； ***RKLLMInferParam* rkllm_infer_params:*** 模型推理过程中的参数传递； ***void* userdata:*** 用户自定义的函数指针，默认设置为 NULL 即可；
返回值	**0** 表示模型推理正常运行；**-1** 表示调用模型推理失败；

3.2.6 模型中断

在进行模型推理时，用户可以调用 rkllm_abort()函数中断推理进程，具体的函数定义如下：

表 3-22 rkllm_ abort 函数接口说明

函数名	rkllm_ abort
描述	用于中断 RKLLM 模型推理进程。
参数	***LLMHandle handle:*** 模型初始化注册的目标句柄；
返回值	**0** 表示 RKLLM 模型中断成功；**-1** 表示模型中断失败；

示例代码如下：

```
// 其中 llmHandle 为模型初始化时传入的句柄
rkllm_abort(llmHandle);
```

3.2.7 释放模型资源

在完成全部的模型推理调用后，用户需要调用 rkllm_destroy()函数进行 RKLLM 模型的销毁，并释放所申请的 CPU、NPU 计算资源，以供其他进程、模型的调用。具体的函数定义如下：

表 3-23 rkllm_ destroy 函数接口说明

函数名	rkllm_ destroy
描述	用于销毁 RKLLM 模型并释放所有计算资源。
参数	***LLMHandle handle:*** 模型初始化注册的目标句柄；
返回值	**0** 表示 RKLLM 模型正常销毁、释放；**-1** 表示模型释放失败；

示例代码如下：

```
// 其中 llmHandle 为模型初始化时传入的句柄
rkllm_destroy(llmHandle);
```

3.2.8 LoRA 模型加载

RKLLM 支持在推理基础模型的同时推理 LoRA 模型，可以在调用 rkllm_run 接口前通过 rkllm_load_lora 接口加载 LoRA 模型。RKLLM 支持加载多个 LoRA 模型，每调用一次 rkllm_load_lora 可加载一个 LoRA 模型。具体的函数定义如下：

表 3-24 rkllm_load_lora 函数接口说明

函数名	rkllm_load_lora
描述	用于加载 LoRA 模型。
参数	***LLMHandle handle:*** 模型初始化注册的目标句柄； ***RKLLMLoraAdapter* lora_adapter:*** 加载 LoRA 模型时的参数配置;
返回值	**0** 表示 LoRA 模型正常加载；**-1** 表示模型加载失败；

表 3-25 RKLLMLoraAdapter 结构体参数说明

结构体定义	RKLLMLoraAdapter
描述	用于配置加载 LoRA 时的参数。
字段	***const char* lora_adapter_path:*** 待加载 LoRA 模型的路径； ***const char* lora_adapter_name:*** 待加载 LoRA 模型的名称，由用户自定义，用于后续推理时选择指定 LoRA； ***float scale:*** LoRA 模型在推理过程中对基础模型参数进行调整的幅度；

加载 LoRA 的示例代码如下：

```
RKLLMLoraAdapter lora_adapter;
memset(&lora_adapter, 0, sizeof(RKLLMLoraAdapter));
lora_adapter.lora_adapter_path = "lora.rkllm";
lora_adapter.lora_adapter_name = "lora_name";
lora_adapter.scale = 1.0;
ret = rkllm_load_lora(llmHandle, &lora_adapter);
if (ret != 0) {
    printf("\nload lora failed\n");
}
```

3.2.9 Prompt Cache 管理

在模型推理过程中，Prefill 阶段通常消耗大量的计算资源和时间，特别是在 Prompt 很长的情况下。为了加速这一过程，RKLLM 支持文件加载 Prompt Cache，通过复用缓存中的内容，可以显著减少 Prefill 阶段的耗时，从而提升整体推理效率。

在调用 rkllm_run 接口进行推理之前，请确保正确配置 prompt_cache_params 参数。这一步骤允许模型在推理结束后生成对应的 Prompt Cache 文件。当首次运行推理时，系统会自动生成一个 Prompt Cache 文件。该文件包含了 Prefill 阶段所需的中间结果，以便后续使用。在后续的

推理任务中，可以通过调用 rkllm_load_prompt_cache 接口加载之前生成的 Prompt Cache 文件。

具体的函数定义如下：

表 3-26 rkllm_load_prompt_cache 函数接口说明

函数名	rkllm_load_prompt_cache
描述	用于加载 Prompt Cache。
参数	***LLMHandle handle:*** 模型初始化注册的目标句柄，可见 3.2.4 初始化模型； ***const char* prompt_cache_path:*** 待加载 Prompt Cache 文件的路径;
返回值	**0** 表示 Prompt Cache 模型正常加载；**-1** 表示模型加载失败；

表 3-27 rkllm_release_prompt_cache 函数接口说明

函数名	rkllm_release_prompt_cache
描述	用于释放 Prompt Cache。
参数	***LLMHandle handle:*** 模型初始化注册的目标句柄，可见 3.2.4 初始化模型；
返回值	**0** 表示 Prompt Cache 模型正常释放；**-1** 表示模型释放失败；

注意：

RKLLM 会从头开始检测输入与 prompt_cache 中相同的部分，假设您的输入格式固定为 PROMPT_PREFIX + text + PROMPT_POSTFIX，您可以仅对 PROMPT_PREFIX 部分生成 Prompt Cache，加载后在后续的推理中即可复用这部分结果。

RKLLM 支持生成多个 Prompt Cache 文件。在需要使用不同的 Prompt Cache 时，只需加载对应的文件即可。当需要切换到另一个 Prompt Cache 文件或不再需要使用加载的 Prompt Cache 时，请显式调用 rkllm_release_prompt_cache 接口进行释放。

加载 Prompt Cache 的示例代码如下：

```
// 初始化并设置 Prompt Cache 参数,并调用 run 接口生成 prompt cache 文件
RKLLMPromptCacheParam prompt_cache_params;

// 是否保存 prompt cache
prompt_cache_params.save_prompt_cache = true;
// 若需要保存 prompt cache, 指定 cache 文件绝对路径
prompt_cache_params.prompt_cache_path = "/data/prompt_cache.bin";
rkllm_infer_params.prompt_cache_params = &prompt_cache_params;

rkllm_infer_params.mode = RKLLM_INFER_GENERATE;
```

```
rkllm_input.input_type = RKLLM_INPUT_PROMPT;
rkllm_input.prompt_input = (char *)prompt.c_str();
rkllm_run(llmHandle, &rkllm_input, &rkllm_infer_params, NULL);

// 加载 prompt cache 文件，减少 prefill 耗时
rkllm_load_prompt_cache(llmHandle, "./prompt_cache.bin");
if (ret != 0) {
    printf("\nload Prompt Cache failed\n");
}

rkllm_run(llmHandle, &rkllm_input, &rkllm_infer_params, NULL);
```

3.2.10 KV Cache 管理

RKLLM 支持手动清除 KV 缓存，可用于单轮和多轮对话。在调用清除缓存功能时，如果 keep_system_prompt 设置为 1，则会保留系统提示词（如果存在）；否则，将清空整个缓存。函数定义如下：

表 3-28 rkllm_clear_kv_cache 函数接口说明

函数名	rkllm_clear_kv_cache
描述	用于清除 kv cache。
参数	***LLMHandle handle:*** 模型初始化注册的目标句柄； ***keep_system_prompt:*** 是否保留系统提示词； ***start_pos:*** 需要删除的 kv cache 起始位 position id，包含 start_pos 位置，若无需使用请手动配置为 nullptr； ***end_pos:*** 需要删除的 kv cache 结束位 position id，不包含 end_pos 位置，若无需使用请手动配置为 nullptr； 注意： 1. 只有在单轮模式中使用暂停推理功能时才支持删除指定位置 kv cache； 2. 如果提供了特定的范围 [start_pos, end_pos)，则忽略 ***keep_system_prompt***，改为清除范围内的 kv cache；
返回值	**0** 表示 kv cache 清除成功；**-1** 表示清除失败；

3.2.11 Chat Template 设置

当用户使用文本输入时，RKLLM 会对默认文本进行前处理，前处理时会根据 Hugging Face 模型 tokenizer_config.json 文件中的 chat_template 字段，自动解析并应用提示词模板。如需自定义，可使用以下函数进行重置，其中，system_prompt 作为系统提示词，用于指导模型行为，prompt_prefix 为用户输入前缀，prompt_postfix 为用户输入后缀，具体的函数定义如下。当用户重置模板后，enable_thinking 选项将失效，用户需要在自定义的 prompt 中进行配置。

表 3-29 rkllm_set_chat_template 函数接口说明

函数名	rkllm_set_chat_template
描述	用于设置提示词。
参数	***LLMHandle handle:*** 模型初始化注册的目标句柄； ***system_prompt:*** 系统提示词； ***prompt_prefix:*** 用户输入前缀； ***prompt_postfix:*** 用户输入后缀；
返回值	**0** 表示 chat_template 设置成功；**-1** 表示设置失败；

3.2.12 Function Calling 设置

RKLLM 支持通过 Function Calling 实现模型与外部系统的结构化交互，扩展大模型能力边界，提升模型在知识补充、数据精确获取等任务中的表现。当启用 Function Calling 模式后，应用程序可将函数定义传入模型，模型根据用户提问判断是否需要调用函数，并根据设定的格式输出调用请求。应用程序根据模型意图调用相应函数并将结果返回，模型最终基于结果继续对话。RKLLM 支持的 Function Calling 设置函数定义如下：

表 3-30 rkllm_set_function_tools 函数接口说明

函数名	rkllm_set_function_tools
描述	Function Calling 配置，包括系统提示词、工具函数定义和工具响应标识
参数	***LLMHandle handle:*** 模型初始化注册的目标句柄； ***system_prompt:*** 系统提示词； ***tools:*** JSON 格式的函数定义字符串，描述可用函数的名称、功能和参数格式； ***tool_response_str:*** 工具函数调用结果的标识标签，用于与普通对话内容进行区分；
返回值	**0** 表示设置成功；**-1** 表示设置失败；

示例代码如下：

首先配置 tools

```
std::string system_prompt = "You are Qwen, created by Alibaba Cloud.
You are a helpful assistant.\n\nCurrent Date: 2024-09-30";
std::string 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"]
        }
    }
}
])";

rkllm_set_function_tools(llmHandle, system_prompt.c_str(),
tools.c_str(), "tool_response");
```

接下来，在推理过程中配合 rkllm_run 实现调用链：

```
// 用户提问
RKLLMInferParam rkllm_infer_params;
memset(&rkllm_infer_params, 0, sizeof(RKLLMInferParam));
rkllm_infer_params.mode = RKLLM_INFER_GENERATE;
rkllm_infer_params.keep_history = 0;

RKLLMInput rkllm_input;
rkllm_input.input_type = RKLLM_INPUT_PROMPT;
rkllm_input.enable_thinking = false;

rkllm_input.role = "user";
rkllm_input.prompt_input = "What's the temperature in San Francisco
now? How about tomorrow?";

rkllm_run(llmHandle, &rkllm_input, &rkllm_infer_params, NULL);

// 第一次调用 rkllm_run 会返回需要调用的工具函数名
<tool_call>
{"name": "get_current_temperature", "arguments": {"location": "San
Francisco"}}
</tool_call>
<tool_call>
{"name": "get_temperature_date", "arguments": {"location": "San
Francisco", "date": "2024-10-01"}}
</tool_call>

// 将工具调用结果返回给大模型，role 必须配置为 tool
rkllm_input.role = "tool";
rkllm_input.prompt_input = R"([
    {
        "temperature": 26.1,
        "location": "San Francisco",
        "unit": "celsius"
    },
    {
        "temperature": 25.9,
        "location": "San Francisco",
        "date": "2024-09-30",
        "unit": "celsius"
    }
])";

rkllm_run(llmHandle, &rkllm_input, &rkllm_infer_params, NULL);

// 最终结果
"The current temperature in San Francisco is 26.1°C. Tomorrow, the
temperature is expected to be 25.9°C."
```

3.2.13 Cross attention 设置

RKLLM 支持 cross attention 推理，用户可以通过如下函数将 encoder 生成的 K/V 缓存、掩码以及位置信息输入给解码器用于交叉注意力计算。cross attention 只支持自定义模型推理，模型的转换方式详见 3.1.6 章节。

表 3-31 rkllm_set_cross_attn_params 函数接口说明

函数名	rkllm_set_cross_attn_params
描述	用于配置交叉注意力参数
参数	***LLMHandle handle:*** 模型初始化注册的目标句柄； ***RKLLMCrossAttnParam* cross_attn_params:*** 交叉注意力参数；
返回值	**0** 表示设置成功；**-1** 表示设置失败；

表 3-32 RKLLMCrossAttnParam 结构体参数说明

结构体定义	RKLLMCrossAttnParam
描述	交叉注意力参数
字段	***- float* encoder_k_cache:*** encoder 输出的 k 值缓存指针， ***- float* encoder_v_cache:*** encoder 输出的 v 值缓存指针 ***- float* encoder mask:*** encoder 的注意力掩码 ***- int32_t* encoder_pos:*** encoder 输入 token 的位置信息 ***- int* num_tokens:*** encoder 的输入 token 数

3.2.14 多 batch 并行推理

RKLLM 支持同时推理多个 batch（建议 batch 数量不超过 8）。

使用两个 batch 进行推理的示例代码如下，主要为：

- 模型初始化时需要将 param.extend_param.n_batch 参数设置为 2；
- 使用多 batch 进行推理时，输入 RLLLMInput 和 callback 中的 RKLLMResult 均为 n_batch 大小的数组；
- callback 中所有 batch 推理结果同步返回，当某个 batch 返回的 token id 为负数时，表示该 batch 推理结束，所有 batch 都推理结束时才会终止推理；
- 在 callback 中处理返回的文本时，必须先判断返回的 text 是否为空指针，再进行赋值操作。

```
#include <string.h>
#include <unistd.h>
#include <string>
#include "rkllm.h"
#include <fstream>
#include <iostream>
#include <csignal>
#include <vector>

using namespace std;
LLMHandle llmHandle = nullptr;

std::string output_texts[10];

int callback(RKLLMResult *result, void *userdata, LLMCallState state)
{
    if (state == RKLLM_RUN_FINISH)
    {
        printf("\nrkllm run finish\n");
    } else if (state == RKLLM_RUN_ERROR) {
        printf("\nrkllm run error\n");
    } else if (state == RKLLM_RUN_NORMAL) {
        RKLLMResult batch1 = result[0];
        RKLLMResult batch2 = result[1];
        if (batch1.text) {
            output_texts[0] += batch1.text;
            printf("batch 0 %s\n", output_texts[0].c_str());
        }
        if (batch2.text) {
            output_texts[1] += batch2.text;;
            printf("batch 1 %s\n", output_texts[1].c_str());
        }
    }
    return 0;
}

int main(int argc, char **argv)
{
    if (argc < 4) {
        std::cerr << "Usage: " << argv[0] << " model_path
max_new_tokens max_context_len\n";
        return 1;
    }

    RKLLMParam param = rkllm_createDefaultParam();
    param.model_path = argv[1];
    param.top_k = 1;
    param.top_p = 0.95;
    param.temperature = 0.8;
    param.repeat_penalty = 1.1;
    param.frequency_penalty = 0.0;
    param.presence_penalty = 0.0;
    param.max_new_tokens = std::atoi(argv[2]);
    param.max_context_len = std::atoi(argv[3]);
    param.skip_special_token = true;
    param.extend_param.base_domain_id = 0;
    param.extend_param.embed_flash = 1;
    param.extend_param.n_batch = 2;
```

```
    int ret = rkllm_init(&llmHandle, &param, callback);
    if (ret == 0){
        printf("rkllm init success\n");
    } else {
        printf("rkllm init failed\n");
        exit_handler(-1);
    }

    RKLLMInput rkllm_input[2];
    memset(&rkllm_input, 0, sizeof(RKLLMInput)*2);
    RKLLMInferParam rkllm_infer_params;
    memset(&rkllm_infer_params, 0, sizeof(RKLLMInferParam));  // 将所有内容初始化为 0
    output_texts[0].clear();
    output_texts[1].clear();

    rkllm_infer_params.mode = RKLLM_INFER_GENERATE;
    rkllm_infer_params.keep_history = 0;

    rkllm_input[0].input_type = RKLLM_INPUT_PROMPT;
    rkllm_input[0].role = "user";
    rkllm_input[0].enable_thinking = false;
    rkllm_input[0].prompt_input = "上联：江边惯看千帆过";

    rkllm_input[1].input_type = RKLLM_INPUT_PROMPT;
    rkllm_input[1].role = "user";
    rkllm_input[1].enable_thinking = false;
    rkllm_input[1].prompt_input = "以咏梅为题目，帮我写一首古诗，要求包含梅花、白雪等元素。";

    printf("robot: ");
    rkllm_run(llmHandle, &rkllm_input[0], &rkllm_infer_params, NULL);

    rkllm_destroy(llmHandle);
    return 0;
}
```

3.2.15 模型暂停推理

RKLLM 支持在单轮模式中暂停推理，具体为在 callback 中 return 1。暂停推理时，kv cache 不会被清除，用户可以继续更改输入后，调用 rkllm_run 恢复推理。

3.2.16 模型性能数据回调

RKLLM 支持在推理结束时返回单次推理性能数据，包括 prefill 和 decode 推理总耗时、token 数量和内存占用，返回的结构体定义如下：

表 3-33 RKLLMPerfStat 结构体参数说明

结构体定义	RKLLMPerfStat
描述	返回模型推理性能数据
字段	***- float prefill_time_ms:*** prefill 阶段总耗时，单位 ms ***- int prefill_tokens:*** prefill 阶段总 token 数 ***- float generate_time_ms:*** generate 阶段总耗时 ***- int generate_tokens:*** generate 阶段生成的总 token 数 ***- float memory_usage_mb:*** 推理过程 VmHWM 内存大小，单位 MB

3.3 板端推理调用示例

本节中提供板端推理调用的 C++工程在 examples/rkllm_api_demo 路径下同步更新，同时提供了编译脚本，以便用户快速编译项目来完成 RKLLM 模型的板端推理调用。

3.3.1 示例工程的完整代码

推理调用的完整 C++代码示例位于工具链的 examples/rkllm_api_demo/deploy/src 文件中，其中 llm_demo.cpp 为大语言模型推理示例。该实现了包括模型初始化、模型推理、回调函数处理输出和模型资源释放等全部流程，用户可以参考相关代码进行自定义功能的实现。

3.3.2 示例工程的使用说明

examples/rkllm_api_demo/deploy 目录下不仅包含了对 RKLLM 模型推理调用的示例代码，同时包含了编译脚本 build-android.sh 和 build-linux.sh。本节将会对示例代码的使用进行简要说明，以使用 build-linux.sh 脚本为 Linux 系统编译可执行文件为例：

首先，用户需要自行准备交叉编译工具，注意在 2.3 中说明的编译工具版本推荐为 10.2 及以上，随后在编译过程前自行替换 build-linux.sh 中的交叉编译工具路径：

```
# 设置交叉编译器路径为本地工具所在路径
GCC_COMPILER_PATH=~/gcc-arm-10.2-2020.11-x86_64-aarch64-none-linux-gnu/bin/aarch64-none-linux-gnu
```

随后，用户即可使用 build-linux.sh 启动编译程序，在编译完成后，用户在

examples/rkllm_api_demo/deploy 目录下构建得到对应的 install/demo_Linux_aarch64/llm_demo 程序。后续将可执行文件、函数库文件夹 lib 及 RKLLM 模型（提前使用 RKLLM-Toolkit 工具完成转换、量化）推送至板端：

```
adb push install/demo_Linux_aarch64  /data
adb push /PC/path/to/your/rkllm/model /data/demo_Linux_aarch64
```

在完成以上步骤后，用户即可通过 adb 进入板端终端界面，并进入相应的 /data/demo_linux_aarch64 文件夹目录下，通过以下指令进行 RKLLM 的板端推理调用：

```
adb shell
cd /data/demo_linux_aarch64
export LD_LIBRARY_PATH=./lib  # 通过环境变量指定函数库路径
./llm_demo /path/to/your/rkllm/model 1024 2048
```

通过以上操作，用户即可进入示例推理界面，与板端模型进行推理交互，并实时获取 RKLLM 模型的推理结果。

3.3.3 板端推理性能及日志查看

若需要查看 RKLLM 在板端推理时的性能或者日志，可使用如下指令：

```
export RKLLM_LOG_LEVEL=1  //仅查看 TTFT、TPS 和内存数据
export RKLLM_LOG_LEVEL=2  //除了性能数据，还会打印 cache 长度等更多信息
```

打印出的性能数据如下图所示：

```
I rkllm: ------------------------------------------------------------------------------------
I rkllm:  Model init time (ms)  834.82
I rkllm: ------------------------------------------------------------------------------------
I rkllm:  Stage         Total Time (ms)  Tokens    Time per Token (ms)      Tokens per Second
I rkllm: ------------------------------------------------------------------------------------
I rkllm:  Prefill       306.31           13        23.56                    42.44
I rkllm:  Generate      1918.31          9         213.15                   4.69
I rkllm: ------------------------------------------------------------------------------------
I rkllm:  Peak Memory Usage (GB)
I rkllm:  0.52
I rkllm: ------------------------------------------------------------------------------------
```

图 3-1 RKLLM 在板端推理性能

3.4 板端 Server 的部署实现

在使用 RKLLM-Toolkit 完成模型转换并得到 RKLLM 模型后，用户可以使用该模型在 Linux 开发板上进行板端 Server 服务的部署，即在 Linux 设备上设置服务端并向局域网内的所有人暴露网络接口，其他人通过访问对应地址即可调用 RKLLM 模型进行推理，实现高效简洁的交互。本节将介绍两种不同的 Server 部署实现：

1）基于 Flask 搭建的 RKLLM-Server-Flask，用户能够在客户端通过 request 请求的方式实现与服务端间的 API 访问。在提供的 RKLLM-Server-Flask 示例中，包含普通对话示例以及支持 Function Calling 功能的单轮对话示例；

2）基于 Graio 搭建的 RKLLM-Server-Gradio，参考所提供的示例，用户能够快速实现网页服务端的搭建，进行可视化交互。此外，该示例中同样提供了 Gradio-API 接口的使用方式，便于用户进行二次开发；

以上所提及的两种 Server 实现的示例代码均位于 examples/rkllm_server_demo 目录下，其中包含了两种实现的具体代码、一键部署脚本及 API 接口调用示例代码，用户可以选择不同的示例进行参考并进行二次开发。具体目录说明如下所示：

```
examples/rkllm_server_demo
├────rkllm_server                  # 板端部署所需文件
│    └──── lib                        # RKLLM Runtime 库
│    └──── flask_server.py            # RKLLM-Server-Flask 部署示例代码
│    └──── gradio_server.py           # RKLLM-Server-Gradio 部署示例代码
├────build_rkllm_server_flask.sh  # RKLLM-Server-Flask 一键部署脚本
├────build_rkllm_server_gradio.sh # RKLLM-Server-Gradio 一键部署脚本
├────chat_api_flask.py            # RKLLM-Server-Flask 的 API 接口调用示例
├────chat_api_gradio.py           # RKLLM-Server-Gradio 的 API 接口调用示例
└────Readme.md
```

3.4.1 RKLLM-Server-Flask 部署示例介绍

RKLLM-Server-Flask 的部署示例中，主要使用了 Flask 框架来完成服务端的搭建，在客户端则通过 request 收发结构体数据来完成 API 访问的实现。

用户在调用 API 的过程中即是向服务端发送特定的结构体数据，其主要内容如下所示：

```
# 客户端调用 OpenAI-API 所发送的结构体数据
{
    "model": "No models available",
    "messages": [
      {
        "role": "system",
        "content": "You are a helpful assistant."
      },
      {
        "role": "user",
        "content": "Hello!"
      }
    ],
    "stream": False,
    "enable_thinking": False,
    "tools": None
}
```

其中，model 和 stream 指明了需要调用的具体模型和是否启动流式推理传输，而 messages 中的 content 数据即为重要的用户输入；enable_thinking 表示模型回答时是否执行深度思考（用户需要根据模型是否支持改功能来设置该选项），tools 为 Function Calling 功能中，用户自定义函数的功能描述（具体定义规则参考 3.4.1.3.2 小节），用户同样也要根据模型是否支持 Function Calling 功能来对此参数进行设置，不支持或者不使用需设置为 None。

而对于服务端回传的数据而言，所输出的结构体数据会根据是否选择流式推理传输而存在不同，非流式推理设置下所回传的数据内容如下所示：

```
# 非流式推理 stream = false 时服务端回传的结构体数据
{
  "id": "chatcmpl-123",
  "object": "chat.completion",
  "created": 1677652288,
  "model": "gpt-3.5-turbo-0125",
  "system_fingerprint": "fp_44709d6fcb",
  "choices": [{
    "index": 0,
    "message": {
      "role": "assistant",
      "content": "\n\nHello there, how may I assist you today?",
    },
    "logprobs": null,
    "finish_reason": "stop"
  }],
  "usage": {
    "prompt_tokens": 9,
    "completion_tokens": 12,
    "total_tokens": 21
  }
}
```

其中，回传的数据中最为重要的部分为 choices 内容下的 messages 内容，即模型所给出的推理结果；

与之不同的是，在设置流式推理传输时，服务端所回传的是一个 Response 对象，该对象中包含流式推理过程中不同时刻下模型的输出结果，每个时刻的数据内容如下：

```
# 流式推理 stream = true 时服务端回传的结构体数据
{
  "id": "chatcmpl-123",
  "object": "chat.completion.chunk",
  "created": 1677652288,
  "model": "gpt-3.5-turbo-0125",
  "system_fingerprint": "fp_44709d6fcb",
  "choices": [{
    "index": 0,
    "delta": {
      "role": "assistant",
      "content": "\n\nHello there, how may I assist you today?",
    },
    "logprobs": null,
    "finish_reason": "stop"
  }]
}
```

其中，用户在接受流式传输得到的数据后，需要关注部分为 choices 中的 delta 数据部分；此外，当 finish_reason 为空值 None 时表示模型正处于推理状态，数据还未完全生成，直到 finish_reason 返回停止符 stop 才表示流失推理结束；

在给出 RKLLM-Server-Flask 的部署示例代码及 API 访问示例中，可以看到对上述传输结构体的相同定义，保证了所部署的 RKLLM-Server-Flask 的一般性。在本节的后续部分中，将分别介绍服务端的一键部署脚本、服务端部署实现的重要设置以及用于客户端的访问 API 的脚本设计。

3.4.1.1 服务端：RKLLM-Server-Flask 的一键部署脚本

rkllm_server_demo 目录下的 build_rkllm_server_flask.sh 即为 RKLLM-Server-Flask 的一键部署脚本，该脚本能够帮助用户在 Linux 开发板上快速搭建 RKLLM-Server-Flask 服务端。在使用该脚本前，用户应注意以下事项：

1） 为开发板接入网线，并在 adb shell 下通过 ifconfig 指令查询该开发板的具体 IP，后续 RKLLM-Server-Flask 将在局域网内以该 IP 设置服务端并接受客户端的访问；

2） 用户需要提前完成 RKLLM 模型的顺利转换，并在执行一键部署脚本前，已将 RKLLM 模型推送至 Linux 板端；

用户可以在 PC 端（非开发板）直接通过调用 build_rkllm_server_flask.sh 脚本，快速地在 Linux 开发板上实现 RKLLM-Server-Flask 服务端的部署。

一键部署脚本 build_rkllm_server_flask.sh 脚本的具体用法如下：

```
./build_rkllm_server_flask.sh
    --workshop [RKLLM-Server Working Path]
    --model_path [Absolute Path of Converted RKLLM Model on Board] --
platform [Target Platform: RK3588/RK3576/RK3562/RV1126B]
    [--lora_model_path [Lora Model Path]]
    [--prompt_cache_path [Prompt Cache File Path]]
```

其中，workshop 参数指明 RKLLM-Server-Flask 在板端的后续工作路径；model_path 参数指明了已使用 RKLLM-Toolkit 转换得到 RKLLM 模型在板端的绝对路径，RKLLM-Server-Flask 在工作时将根据该路径读取 RKLLM 模型；platform 参数指明了当前使用的平台类型为 RK3588、RK3576、RK3562 或 RV1126B；lora_model_path 和 prompt_cache_path 则作为可选参数，当用户需要加载 Lora 模型或需要调用 Prompt 功能时可以通过两个参数指定具体的文件路径；

以下是一键部署脚本 build_rkllm_server_flask.sh 使用的简单示例：

```
./build_rkllm_server_flask.sh
    --workshop /path/to/workshop
    --model_path /path/to/model.rkllm
    --platform rk3588
```

在执行上述命令后后，一键部署脚本将进行对板端 Linux 环境的检查、自动安装 Flask 库、将 rkllm_server_demo/rkllm_server 下部署示例运行所需的相关文件推送至板端、并在预设的工作路径下索引 RKLLM 模型来完成 RKLLM-Server-Flask 的启动。当终端中出现下图中的“RKLLM Model has been initialized successfully!”信息后，即说明 RKLLM-Server-Flask 示例的成功部署。

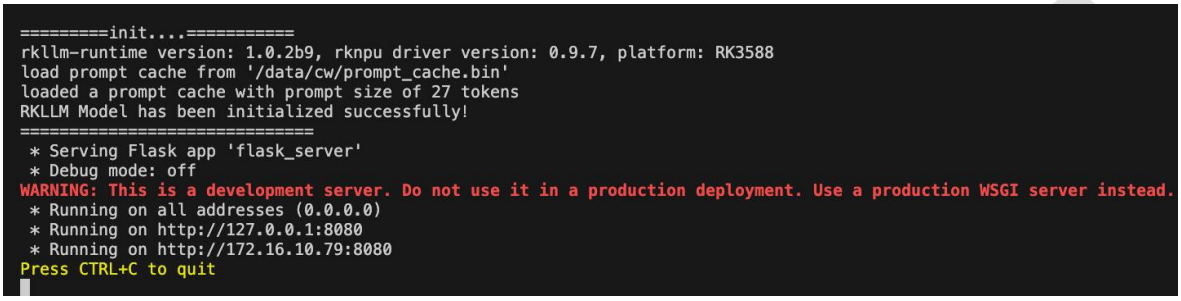


图 3-2 RKLLM-Server-Flask 成功部署后的终端信息

参考 build_rkllm_server_flask.sh 中的具体代码逻辑，用户可以了解 RKLLM-Server-Flask 示例的详细部署过程，这能够帮助用户更加灵活地进行自定义 Server 的部署实现。在此需要强调的是，在一键部署脚本的步骤 3 中，一键部署脚本自动将当前版本的 RKLLM Runtime 同步至 rkllm_server/lib/librkllmrt.so，保证了 flask_server.py 在运行时调用的是当前版本的 librkllmrt.so。

3.4.1.2 服务端：RKLLM-Server-Flask 部署实现介绍

本小节对 RKLLM-Server-Flask 部署示例的实现方式进行梳理介绍，帮助用户了解示例代码的构建逻辑，以便用户参考当前 RKLLM-Server-Flask 的实现进行二次开发。

在 RKLLM-Server-Flask 的部署示例中，主要基于 Flask 库完成了服务端的基本实现；此外，在 RKLLM 模型的推理上，选择使用 Python 中的 ctypes 库来完成对 RKLLM Runtime 动态库的直接调用。

在 rkllm_server/flask_server.py 的整体实现中，为了通过 ctypes 对 librkllmrt.so 进行调用，需要提前在 Python 中参考 librkllmrt.so 所对应的头文件 rkllm.h 完成相关定义，在 Flask 服务端接收用户发送的结构体数据后，调用相关函数实现 RKLLM 模型的推理。flask_server.py 的具体代码实现主要由以下步骤组成：

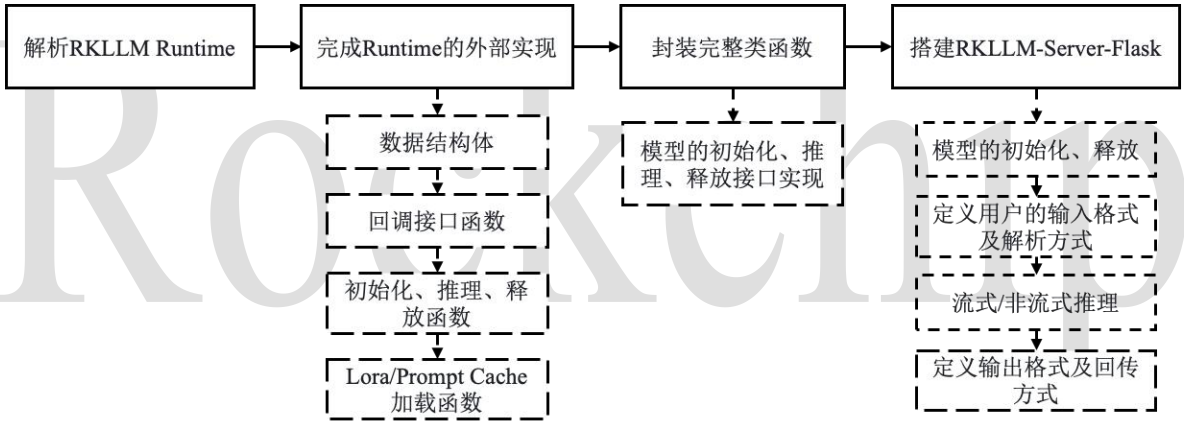


图 3-3 RKLLM-Server-Flask 部署实现流程介绍

1） 解析 RKLLM Runtime：设置动态库的路径，通过 ctypes 加载动态库 librkllmrt.so 实现对 RKLLM Runtime 的解析；

2） 完成 Runtime 的外部实现：在 Python 代码中，通过 ctypes 完成对 RKLLM Runtime 头文件中相关实现的外部定义，包括数据结构体定义，回调函数、初始化函数、推理函数、释放函数、Lora/Prompt Cache 加载函数的接口定义等；

3） 封装完整的类函数：基于 2）中所实现的各种 Runtime 数据类型及函数接口，封装完整的类函数，集成 RKLLM 的初始化、推理、释放等操作，便于后续调用；

4） 搭建 RKLLM-Sever-Flask：利用 3）中封装的完整类函数搭建 Flask 服务端，包括在 Flask 启动时根据用户指定参数加载 RKLLM 模型、定义用户输入的格式以及输入的解析方

式、定义流式/非流式推理的不同推理调用方式、定义推理输出的回传格式及回调函数的具体实现、RKLLM 的释放等；

以上各模块的具体实现组成了 rkllm_server/flask_server.py 的主体代码，从而完成了对 RKLLM-Server-Flask 示例的部署实现，用户可以修改其中对于 RKLLM 模型的初始化定义来实现不同的自定义模型。此外，用户同样可以参考以上 RKLLM-Server-Flask 部署示例，进行自定义 Server 的部署实现。

3.4.1.3 客户端：API 访问示例

在 rkllm_server_demo/chat_api_flask.py 中包含了普通对话示例(主要代码为 main_demo1)以及支持 Function Calling 功能单轮对话示例(主要代码为 main_demo2)。

3.4.1.3.1 客户端：API 访问示例 1——普通对话

普通对话功能即常见的问答对话，本小节对普通对话示例进行详细说明：

1) 定义 RKLLM-Server-Flask 的网络地址，用户需要根据 Linux 开发板的具体 IP、Flask 框架所设置的端口号及函数名称对访问目标进行设置；

```
# 设置 Server 服务器的地址
server_url = 'http://172.16.10.xx:8080/rkllm_chat'
```

2) 定义 API 访问的形式，可选非流式传输和流式传输，默认为非流式传输；

```
# 设置是否开启流式对话
main_demo1(is_streaming=True)
```

3) 定义会话对象，使用 requests.Session() 对 API 访问接口与服务端的通信过程进行相关设置，用户可根据实际开发需要进行自定义修改；

```
# 创建一个会话对象
session = requests.Session()
session.keep_alive = False  # 关闭连接池，保持长连接
adapter = requests.adapters.HTTPAdapter(max_retries=5)
session.mount('https://', adapter)
session.mount('http://', adapter)
```

4) 定义 API 调用过程中用于包装发送数据的结构体，用户对 RKLLM-Server-Flask 的访问将被包含在该结构体中；

```
# 准备要发送的数据
# model: 为用户在设置 RKLLM-Server 时定义的模型，此处并无作用
# messages: 用户输入的问题；支持在 messags 加入多个问题
# stream: 是否开启流式对话
# enable_thinking: 是否开启深度思考
# tools: 用户对自定义函数的描述

data = {
    "model": 'your_model_deploy_with_RKLLM_Server',
    "messages": [{"role": "user", "content": user_message}],
    "stream": is_streaming,
    "enable_thinking": False,
    "tools": None
}
```

5） 向 RKLLM-Server-Flask 服务端发送请求，并获取回传数据；

```
# 发送 POST 请求
responses = session.post(server_url, json=data, headers=headers,
stream=is_streaming, verify=False)
```

6） 对回传数据结构体进行解析；

```
# 解析响应
# 非流式传输
if not is_streaming:
       if responses.status_code == 200:
       print("Q:", data["messages"][-1]["content"])
       print("A:", json.loads(responses.text)["choices"][-
1]["message"]["content"])
    else:
       print("Error:", responses.text)

# 流式传输
else:
    if responses.status_code == 200:
       print("Q:", data["messages"][-1]["content"])
       print("A:", end="")
       for line in responses.iter_lines():
          if line:
             line = json.loads(line.decode('utf-8'))
             if line["choices"][-1]["finish_reason"] != "stop":
                print(line["choices"][-1]["delta"]["content"], end="")
                sys.stdout.flush()
    else:
       print('Error:', responses.text)
```

以上 1-6 的整体流程即为 RKLLM-Server-Flask 服务端的 API 访问方式，用户可根据上述示例代码进行自定义功能的开发。需要注意的是，用户一定要核对 RKLLM-Server-Flask 的确定网址，即确定的 IP 地址、端口号和服务端接受输入的函数接口；此外，当遇到特定的收发结构体需求时，用户可在服务端和客户端自定义所需的数据结构体，保证自定义功能的实现。

3.4.1.3.2 客户端：API 访问示例 2——支持 Function Calling 功能的单轮对话

LLM 对于训练数据中没有的信息，包括训练结束后发生的事情，它们并不了解。此外，它们通过概率方式学习，这意味着对于有固定规则集的任务，如数学计算，可能不够精确。为此，Function Calling 是指模型根据用户请求自动调用预定义的函数来获取更精确、实时的信息。

本小节以天气查询任务为例（场景：假设用户要询问 LLM 关于旧金山的当天温度和明日温度。通常，模型会无法回答这种实时性问题。但用户有两个函数，可以分别获取城市的当前温度和指定日期的温度，希望模型能够利用这两个函数，得到准确的答案），对 Function Calling 功能进行详细说明：

1） 定义 RKLLM-Server-Flask 的网络地址，用户需要根据 Linux 开发板的具体 IP、Flask 框架所设置的端口号及函数名称对访问目标进行设置；

```
# 设置 Server 服务器的地址
server_url = 'http://172.16.10.xx:8080/rkllm_chat'
```

2） 定义 API 访问的形式，可选非流式传输和流式传输，默认为非流式传输；

```
# 设置是否开启流式对话
main_demo2(is_streaming=True)
```

3） 定义会话对象，使用 requests.Session() 对 API 访问接口与服务端的通信过程进行相关设置，用户可根据实际开发需要进行自定义修改；

```
# 创建一个会话对象
session = requests.Session()
session.keep_alive = False  # 关闭连接池，保持长连接
adapter = requests.adapters.HTTPAdapter(max_retries=5)
session.mount('https://', adapter)
session.mount('http://', adapter)
```

4） 定义 llm 需要调用的函数；get_current_temperature 函数返回指定城市的当前温度，get_temperature_date 返回指定城市指定在指定日期的温度；

```
def get_current_temperature(location: str, unit: str = "celsius"):
      """
      Get current temperature at a location.

      Args:
         location: The location to get the temperature for, in the
format "City, State, Country".
         unit: The unit to return the temperature in. Defaults to
"celsius". (choices: ["celsius", "fahrenheit"])

      Returns:
         the temperature, the location, and the unit in a dict
      """
      return {
         "temperature": 26.1,
         "location": location,
         "unit": unit,
      }


def get_temperature_date(location: str, date: str, unit: str =
"celsius"):
      """
      Get temperature at a location and date.

      Args:
         location: The location to get the temperature for, in the
format "City, State, Country".
         date: The date to get the temperature for, in the format
"Year-Month-Day".
         unit: The unit to return the temperature in. Defaults to
"celsius". (choices: ["celsius", "fahrenheit"])

      Returns:
         the temperature, the location, the date and the unit in a
dict
      """
      return {
         "temperature": 25.9,
         "location": location,
         "date": date,
         "unit": unit,
      }
```

5） 定义对自定义函数的描述

在本示例中，TOOLS 中包含两个函数的描述，每个函数描述包含两个字段的 JSON object：

type：string，用于指定工具类型，目前仅"function"有效。function：object，详细说明了如何使用该函数。对于每个 function，它是一个具有三个字段的 JSON object：name：string 表示函数名称；description：string 描述函数用途；parameters：JSON Schema(https://json-schema.org/learn/getting-started-step-by-step)，用于指定函数接受的参数。请参阅链接以了解如何构建 JSON Schema。值得注意的字段包括 type、required 和 enum。

```
TOOLS = [
       {
           "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"],
               },
           },
       },
   ]
```

6) 定义 API 调用过程中用于包装发送数据的结构体，用户对 RKLLM-Server-Flask 的访问将被包含在该结构体中，开启对 LLM 的第一次调用；

```
# 准备要发送的数据
# model: 为用户在设置 RKLLM-Server 时定义的模型，此处并无作用
# messages: 用户输入的问题；本示例中仅支持一个问题，即单轮对话
# stream: 是否开启流式对话
# enable_thinking: 是否开启深度思考
# tools: 用户对自定义函数的描述

messages = [
    {"role": "system", "content": "You are Qwen, created by Alibaba
Cloud. You are a helpful assistant.\n\nCurrent Date: 2024-09-30"},
    {"role": "user",  "content": "What's the temperature in San
Francisco now? How about tomorrow?"},
    ]

data = {
        "model": 'your_model_deploy_with_RKLLM_Server',
        "messages": messages,
        "stream": False,
        "enable_thinking": False,
        "tools": TOOLS
    }

# Send a POST request
responses = session.post(server_url, json=data, headers=headers,
stream=False, verify=False)
```

第一次调用 LLM 的返回如下：即返回调用用户自定义函数的输入参数；

```
server_answer = json.loads(responses.text)["choices"][-
1]["message"]["content"]
matches = re.findall(r"<tool_call>\s*(\{.*?\})\s*</tool_call>",
''.join(server_answer), re.DOTALL)
print("server_answer:", server_answer, '\n')

'''
以下为 print 的输出内容
server_answer: <tool_call>
{"name": "get_current_temperature", "arguments": {"location": "San
Francisco"}}
</tool_call>
<tool_call>
{"name": "get_temperature_date", "arguments": {"location": "San
Francisco", "date": "2024-10-01"}}
</tool_call>
'''
```

7) 接下来，对 LLM 的返回解析，构建第二次调用 LLM 的输入，并开启对 LLM 的第二次调用：

```
result = [json.loads(match) for match in matches]
for function_call in result:
    messages.append({'role': 'assistant', 'content': '',
'function_call':function_call})

tool_calls = [{'function': result[i]} for i in range(len(result))]
function_call = []
for tool_call in tool_calls:
    if fn_call := tool_call.get("function"):
        fn_name: str = fn_call["name"]
        fn_args: dict = fn_call["arguments"]
        fn_res: str =
json.dumps(get_function_by_name(fn_name)(**fn_args))
        messages.append({'role': 'tool', 'name': fn_name,
'content':fn_res})

print("messages:", messages, '\n')


'''
以下为 print 的输出内容
messages: [{'role': 'system', 'content': 'You are Qwen, created by
Alibaba Cloud. You are a helpful assistant.\n\nCurrent Date: 2024-09-
30'}, {'role': 'user', 'content': "What's the temperature in San
Francisco now? How about tomorrow?"}, {'role': 'assistant', 'content':
'', 'function_call': {'name': 'get_current_temperature', 'arguments':
{'location': 'San Francisco'}}}, {'role': 'assistant', 'content': '',
'function_call': {'name': 'get_temperature_date', 'arguments':
{'location': 'San Francisco', 'date': '2024-10-01'}}}, {'role': 'tool',
'name': 'get_current_temperature', 'content': '{"temperature": 26.1,
"location": "San Francisco", "unit": "celsius"}'}, {'role': 'tool',
'name': 'get_temperature_date', 'content': '{"temperature": 25.9,
"location": "San Francisco", "date": "2024-10-01", "unit":
"celsius"}'}]
'''

data = {
"model": 'your_model_deploy_with_RKLLM_Server',
"messages": messages,
"stream": is_streaming,
"enable_thinking": False,
"tools": TOOLS
}

# Send a POST request
responses = session.post(server_url, json=data, headers=headers,
stream=is_streaming, verify=False)
```

8) 对第二次调用 LLM 得到的回传数据结构体进行解析，得到模型关于旧金山当前温度与明日温度的最终回答；

```
if not is_streaming:
       # Parse the response
       if responses.status_code == 200:
          print("A:", json.loads(responses.text)["choices"][-
1]["message"]["content"])
       else:
          print("Error:", responses.text)
else:
   if responses.status_code == 200:
       print("A:", end="")
       for line in responses.iter_lines():
       if line:
          line = json.loads(line.decode('utf-8'))
          if line["choices"][-1]["finish_reason"] != "stop":
             print(line["choices"][-1]["delta"]["content"], end="")
             sys.stdout.flush()
   else:
       print('Error:', responses.text)


'''
以下为 print 的输出内容
A:The current temperature in San Francisco is 26.1°C. Tomorrow, the
temperature is expected to be 25.9°C.
'''
```

以上 1-8 的整体流程即为 RKLLM-Server-Flask 服务端的 Function Calling 的使用示例，用户可根据上述示例代码进行自定义功能的开发。需要注意的是，用户需要确定所调用的 LLM 支持 Function Calling 功能。此外，对于一些能力较弱的模型，例如 Qwen3-0.6B，在第一次调用后，它无法准确返回用户自定义函数的输入参数，这将导致 Function Calling 功能的失败。

3.4.2 RKLLM-Server-Gradio 部署示例介绍

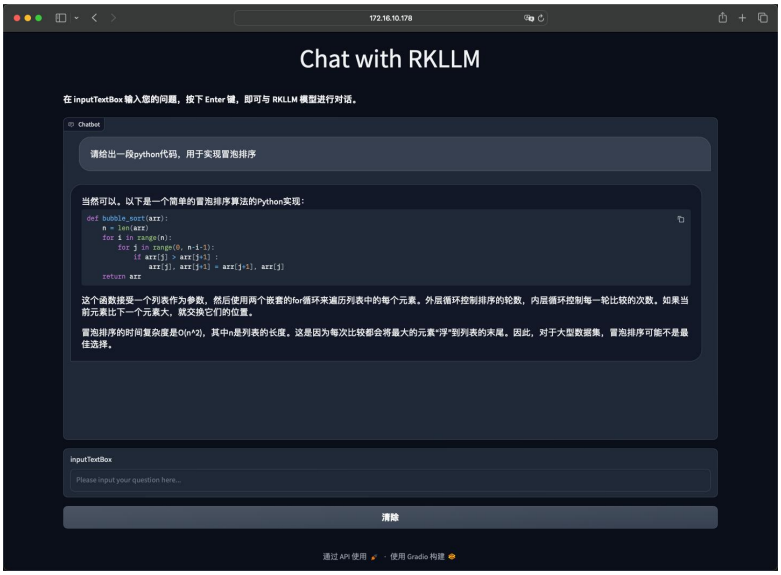


图 3-4 RKLLM-Server-Gradio 部署示例的外部访问页面

gradio 是一个简单易用的 Python 库，能够用于快速构建机器学习模型的交互式界面，本节将具体介绍如何在 Linux 设备上利用 gradio 进行 RKLLM-Server-Gradio 的快速部署实现，并在局域网内直接访问服务端来使用 RKLLM 模型推理，上图展示了 RKLLM-Server-Gradio 部署成功后的网页端示例。

在当前的 rkllm_server_demo 目录下，已包含了上图 3-3 所展示的 RKLLM-Server-Gradio 部署示例的具体实现代码。用户同样可以直接通过一键部署脚本 build_rkllm_server_gradio.sh 来实现 RKLLM-Server-Gradio 的快速搭建，并在部署成功后，选择通过网页端访问或是 API 访问的形式来调用 RKLLM 模型进行推理。

3.4.2.1 服务端：一键部署脚本

一键部署脚本 build_rkllm_server_gradio.sh 是为了方便用户在 Linux 开发板上快速搭建 RKLLM-Server-Gradio 服务端的部署脚本。与 RKLLM-Server-Flask 搭建过程一致，在使用一键部署脚本前，用户需要注意：

1） 为开发板接入网线，并在 adb shell 下通过 ifconfig 指令查询该开发板的具体 IP，后续 RKLLM-Server-Gradio 将在局域网内以该 IP 设置服务端并接受客户端的访问；

2） 用户需要提前完成 RKLLM 模型的顺利转换，并在执行一键部署脚本前，已将 RKLLM

模型推送至 Linux 板端；

一键部署脚本 build_rkllm_server_gradio.sh 的具体用法与 build_rkllm_server_flask.sh 类似：

```
./build_rkllm_server_gradio.sh
    --workshop [RKLLM-Server Working Path]
    --model_path [Absolute Path of Converted RKLLM Model on Board] --
platform [Target Platform: RK3588/RK3576/RK3562/RV1126B]
    [--lora_model_path [Lora Model Path]]
    [--prompt_cache_path [Prompt Cache File Path]]
```

相同地，workshop 参数指明 RKLLM-Server-Gradio 在板端的后续工作路径；model_path 参数指明了已使用 RKLLM-Toolkit 转换得到 RKLLM 模型在板端的绝对路径，RKLLM-Server-Gradio 在工作时将根据该路径读取 RKLLM 模型；platform 参数指明了当前使用的平台类型为 RK3588、RK3576、RK3562 或 RV1126B；lora_model_path 和 prompt_cache_path 则作为可选参数，当用户需要加载 Lora 模型或需要调用 Prompt 功能时可以通过两个参数指定具体的文件路径；

用户可以在 PC 端（非开发板）直接通过以下命令完成 build_rkllm_server_gradio.sh 脚本的简单调用，快速实现 RKLLM-Server-Gradio 示例的部署：

```
./build_rkllm_server_gradio.sh
    --workshop /user/data
    --model_path /user/data/model.rkllm
    --platform rk3588
```

在执行上述命令后后，一键部署脚本将进行对板端 Linux 环境的检查、自动安装 gradio 库、将 rkllm_server_demo/rkllm_server 下 RKLLM-Server-Gradio 示例运行所需的相关文件推送至板端、并在预设的工作路径下索引 RKLLM 模型来完成 RKLLM-Server-Gradio 的启动。当终端中出现下图中的“RKLLM Model has been initialized successfully!”信息后，即说明 RKLLM-Server 示例的成功启动。

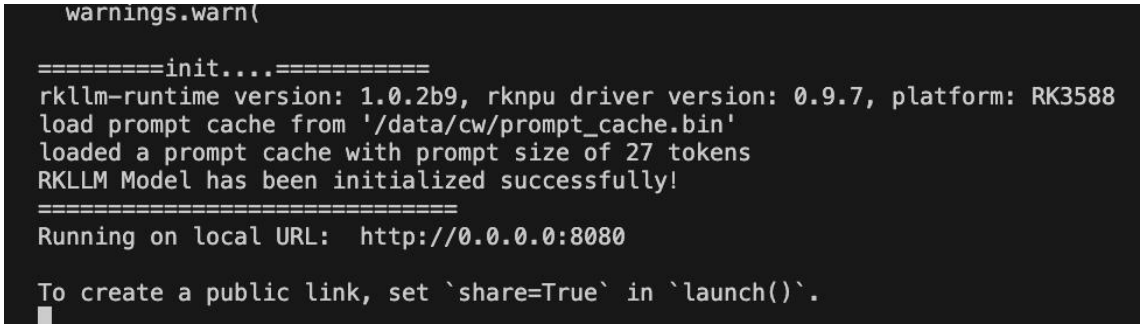


图 3-5 PC 端执行一键部署脚本后的终端信息

参考 build_rkllm_server_gradio.sh 中的具体代码，用户可以了解 RKLLM-Server-Gradio 示例的详细部署过程，这能够帮助用户更加灵活地进行自定义 Server 的部署实现。在此需要强调的是，

在 build_rkllm_server_gradio.sh 的步骤 3 中，一键部署脚本自动将当前版本的 RKLLM Runtime 同步至 rkllm_server/lib/librkllmrt.so，保证了 gradio_server.py 在运行时对 librkllmrt.so 的索引，用户在进行自定义 Server 的过程中需要注意对 librkllmrt.so 的调用。

3.4.2.2 服务端：RKLLM-Server-Gradio 部署实现介绍

RKLLM-Server-Gradio 与 RKLLM-Server-Flask 的部署实现的基本一致，同样是使用了 ctypes 库对 RKLLM Runtime 库进行直接调用，完成 RKLLM 模型的推理，具体的部署实现流程可参考下图所示：

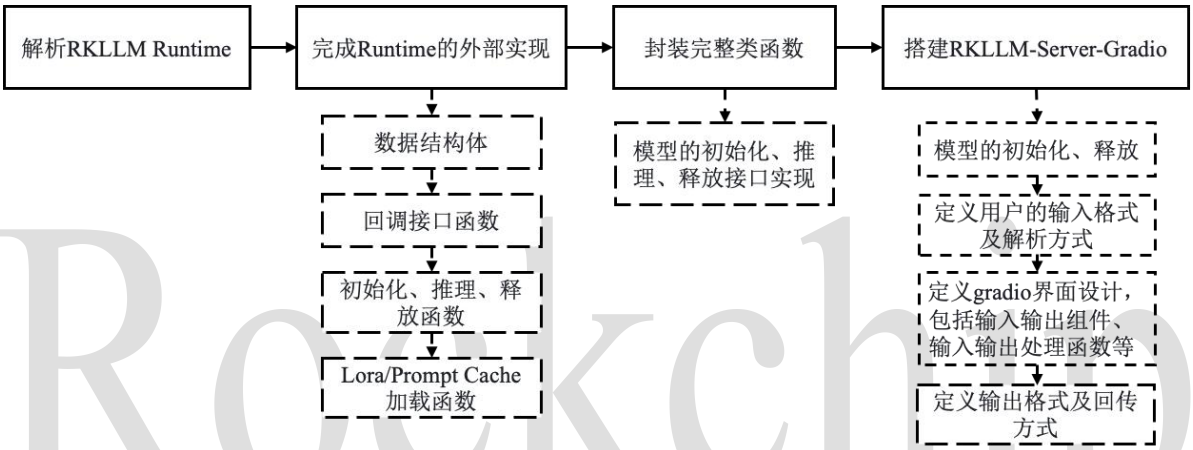


图 3-6 RKLLM-Server-Gradio 部署实现流程介绍

不同的是，RKLLM-Server-Gradio 选择使用 gradio 库来实现服务端的搭建并完成与客户端的通信，能够提供简单的网页端服务，这也要求在在 RKLLM-Server-Gradio 的是现在需要对 gradio 界面进行特定的处理：

1） gradio 函数库提供了输入输出数据的完整通信实现，因此无需再通过 Flask 来定义 Server 对于输入输出的复杂实现；

2） gradio 是基于不同的控件元素实现的部署框架，在使用中需要调用 gradio 的不同控件来完成界面设计，并为每个控件指定其触发条件、函数调用逻辑以及不同控件间的数据流传递逻辑；

用户可以参考 rkllm_server/gradio_server.py 的主体代码来了解 RKLLM-Server-Gradio 的具体实现，并通过修改其中对于 RKLLM 模型的初始化定义来实现不同的自定义模型。此外，以可以参考 RKLLM-Server-Gradio 部署示例的实现代码，进行自定义 Server 的部署实现。

3.4.2.3 客户端：RKLLM-Server-Gradio 使用说明

在 Linux 开发板上成功完成 RKLLM-Server-Gradio 服务端的部署后，用户可以通过“界面访问”和“API 调用”两种方式对 RKLLM-Server-Gradio 进行访问。

1） 界面访问。在通过一键部署脚本成功启动 RKLLM-Server-Gradio 后，用户即可在当前局域网下，通过任意一台电脑的浏览器直接访问“开发板 IP:8080”（如图 3-2 中的“172.16.10.xx:8080”）与 RKLLM 模型进行快速交互，实现效果可见图 3-2。在 RKLLM-Server-Gradio 的使用过程中，gradio 自动集成了 Markdown、HTML 等语法，能够自动匹配 RKLLM 模型所输出结果的格式，如代码段、Markdown 文本等；并且，该部署在 RKLLM-Server 的设置过程中启动了访问队列，当多个用户同时与 RKLLM-Server 进行交互时，将按照输入提交时间的顺序依次推理并返回；需要注意的是，当前用户对 RKLLM-Server 正处于推理状态（对话框处于高亮状态）时，将不再接受该用户的下一次输入，直到本次推理结束。

2） API 调用。rkllm_server_demo 目录下提供了 chat_api_gradio.py，用户在 PC 端安装 gradio_client（安装指令为：pip install gradio_client）后，即可脱离界面仅使用 api 接口与 RKLLM-Server 进行交互，实现效果如下图所示。此外，在使用 chat_api_gradio.py 前，同样需要注意修改该代码中的 IP 地址为开发板当前的 IP 地址，如以下代码段所示。

```
from gradio_client import Client
# 实例化 Gradio Client，用户需要根据自己部署的具体网址进行修改
client = Client("http://172.16.10.xx:8080/")
```

图 3-7 终端使用 API 调用访问

用户可以根据需求场景自行选择两种不同的客户端调用方式，如当需要在局域网内提供交互服务，推荐使用界面访问的方式；而当需要自定义对 RKLLM-Server-Gradio 的访问行为时，推荐使用 API 调用方式进行二次开发；

最后，需要强调的是，在 RKLLM-Server-Gradio 的实现当中，并没有实现类似 OpenAI-API

收发数据结构体的定义，因此该部署实现无法兼容 OpenAI-API 接口，用户在进行二次开发时需要参考 chat_api_gradio.py 中的具体函数实现方式完成开发；若是在开发中需要兼容 OpenAI-API 接口，请参考 RKLLM-Server-Flask 的实现。

4 相关资源

RKLLM：https://github.com/airockchip/rknn-llm

RKNN：https://github.com/airockchip/rknn-toolkit2