

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：	1.0.1
	作 者：	AI 组
	完成日期：	2024-5-8
	审 核：	熊伟
	完成日期：	2024-5-8

瑞芯微电子股份有限公司

Rockchip Electronics Co., Ltd

(版本所有,翻版必究)

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

Rockchip

目 录

1 RKLLM 简介.....	5
1.1 RKLLM 工具链介绍.....	5
1.1.1 RKLLM-Toolkit 功能介绍.....	5
1.1.2 RKLLM Runtime 功能介绍.....	5
1.2 RKLLM 开发流程介绍.....	5
1.3 适用的硬件平台.....	6
2 开发环境准备.....	7
2.1 RKLLM-Toolkit 安装.....	7
2.1.1 通过 pip 方式安装	8
2.2 RKLLM Runtime 库的使用.....	8
2.3 RKLLM Runtime 的编译要求.....	9
2.4 芯片内核更新.....	9
3 RKLLM 使用说明.....	11
3.1 模型转换.....	11
3.1.1 RKLLM 初始化	11
3.1.2 模型加载	11
3.1.3 RKLLM 模型的量化构建	12
3.1.4 导出 RKLLM 模型	12
3.2 板端推理实现.....	13
3.2.1 回调函数定义	13
3.2.2 参数结构体 RKLLMParam 定义.....	15
3.2.3 初始化模型	17
3.2.4 模型推理	17
3.2.5 模型中断	18

3.2.6 释放模型资源	19
3.3 板端推理调用示例.....	19
3.3.1 示例工程的完整代码	19
3.3.2 示例工程的使用说明	22
3.4 板端 Server 的部署实现	22
3.4.1 RKLLM-Server-Flask 部署示例介绍.....	23
3.4.2 RKLLM-Server-Gradio 部署示例介绍	34
4 相关资源.....	39

Rockchip

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 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, Phi-2, Phi-3, ChatGLM3, Gemma, InternLM2 和 MiniCPM，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 w4a16 和 w8a8。

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 RK3576/RK3588 板端通过调用 NPU 驱动在 Rockchip NPU 上实现 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 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：获取 Hugging Face 格式的大语言模型；或是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。
- b. 模型加载：通过 `rkllm.load_huggingface()` 函数加载原始模型。

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。

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 说明文档
rkllm-runtime
├── examples
│   ├── rkllm_api_demo          # 板端推理调用示例工程
│   └── rkllm_server_demo       # RKLLM-Server 部署示例工程
├── runtime
│   ├── Android
│   │   ├── librkllm_api
│   │   │   └── arm64-v8a
│   │   │       ├── librkllmrt.so # RKLLM Runtime 库
│   │   │       └── include
│   │   │           └── rkllm.h    # Runtime 头文件
│   └── Linux
│       ├── librkllm_api
│       │   └── aarch64
│       │       ├── librkllmrt.so # RKLLM Runtime 库
│       │       └── include
│       │           └── rkllm.h    # Runtime 头文件
rkllm-toolkit
├── examples
│   └── huggingface
│       └── test.py
├── packages
│   ├── md5sum.txt
│   └── rkllm_toolkit-x.x.x-cp38-cp38-linux_x86_64.whl
rknpu-driver
└── rknpu_driver_0.9.6_20240322.tar.bz2
```

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-cp38-cp38-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

在所公开的 RKLLM 工具链文件中，包括包含 RKLLM Runtime 的全部文件：

1) lib/librkllmrt.so: 适用于 RK3576/RK3588 板端进行模型推理的 RKLLM Runtime 库;

2) include/rkllm_api.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 RK3576/RK3588 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 RK3576/RK3588 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 rkllm/rkllm-runtime/examples/rkllm_api_demo 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 v0.9.6 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: v0.9.6
```

若所查询的 NPU 内核版本低于 v0.9.6, 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 5.10.198, 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 6.1.57; 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_0.9.6_20240322.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。
- 3) 重新编译内核。
- 4) 将新编译的内核烧录到设备中。

Rockchip

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 格式的大语言模型转换为 RKLLM 模型，从而将 RKLLM 模型在 Rockchip NPU 上加载运行。本节将重点介绍 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：

```
rkllm = RKLLM()
```

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 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: 存放 LLM 模型文件的文件夹路径，用于加载模型进行后续的转换、量化；
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')
if ret != 0:
    print('Load model failed!')
```

3.1.3 RKLLM 模型的量化构建

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 3-2 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	<p>do_quantization: 该参数控制是否对模型进行量化操作，建议设置为 <code>True</code>；</p> <p>optimization_level: 该参数用于设置是否进行量化精度优化，可选的设置包括 {0, 1}，0 表示不做任何优化，1 表示进行精度优化，精度优化有可能会造成模型推理性能下降；</p> <p>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” 和 “w8a8”，“w4a16” 表示对权重进行 4bit 量化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；目前 rk3576 平台支持 “w4a16” 和 “w8a8” 两种量化类型，rk3588 仅支持 “w8a8” 量化类型；</p> <p>target_platform: 模型运行的硬件平台，可选的设置包括 “rk3576” 或 “rk3588”；</p>
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',  
    target_platform='rk3588')  
if ret != 0:  
    print('Build model failed!')
```

3.1.4 导出 RKLLM 模型

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 3-3 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；
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!')
```

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 板端推理实现

此章节介绍通用 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：

```
void callback(RKLLMResult* result, void* userdata, LLMCallState state)
{
    if(state == LLM_RUN_NORMAL){
        printf("%s", result->text);
        for (int i=0; i<result->num; i++) {
            printf("token_id: %d logprob: %f", result->tokens[i].id,
                result->tokens[i].logprob);
        }
    }
    if (state == LLM_RUN_FINISH) {
        printf("finish\n");
    } else if (state == LLM_RUN_ERROR){
        printf("\run error\n");
    }
}
```

1) LLMCallState 是一个状态标志，其具体定义如下：

表 3-4 LLMCallState 状态标志说明

变量名	LLMCallState
描述	用于表示当前 RKLLM 的运行状态。
参数	无
返回值	0, <i>LLM_RUN_NORMAL</i> , 表示 RKLLM 模型当前正在推理中; 1, <i>LLM_RUN_FINISH</i> , 表示 RKLLM 模型已完成当前输入的全部推理; 2, <i>LLM_RUN_ERROR</i> , 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；

2) RKLLMResult 是返回值结构体，其具体定义如下：

表 3-5 RKLLMResult 返回值结构体说明

变量名	RKLLMResult
描述	用于返回当前推理生成结果。
参数	无
返回值	0, <i>text</i> , 表示当前推理生成的文本内容; 1, <i>tokens</i> , 表示当前推理生成的 token 位置上概率最大的前 top 个 token，其中每个 token 包含 id 和 logprob 两个属性; 2, <i>num</i> , 表示 <i>tokens</i> 中的数量;

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 3-6 RKLLMParam 结构体参数说明

变量名	RKLLMParam
描述	用于定义 RKLLM 模型的各项细节参数。
参数	<p><i>const char* model_path:</i> 模型文件的存放路径；</p> <p><i>int32_t num_npu_core:</i> 模型推理时使用的 NPU 核心数量，若芯片平台为“rk3576”可配置的范围为[1, 2]，若为“rk3588”可配置的范围则为[1, 3]；</p> <p><i>bool use_gpu:</i> 是否使用 GPU 进行 prefill 加速，该选项默认为 false；</p> <p><i>int32_t max_context_len:</i> 设置 prompt 的上下文大小；</p> <p><i>int32_t max_new_tokens:</i> 用于设置模型推理时生成 token 的数量上限；</p> <p><i>int32_t top_k:</i> 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。这种方法有助于减少生成低概率或无意义 token 的风险。更高的 top-k 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；</p> <p><i>float top_p:</i> 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 top-p 值（如 0.95）使得生成的文本更加多样化；而更低的值（如 0.5）将生成更加集中和保守的文本。默认值为 0.9；</p> <p><i>float temperature:</i> 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</p>

	<p>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；</p> <p><i>float repeat_penalty:</i> 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；</p> <p><i>float frequency_penalty:</i> 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；</p> <p><i>int32_t mirostat:</i> 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；</p> <p><i>float mirostat_tau:</i> 选项设置 mirostat 的目标熵，代表生成文本的期望困惑度值。调整目标熵可以让你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；</p> <p><i>float mirostat_eta:</i> 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；</p> <p><i>bool logprobs:</i> 选项设置是否返回输出 token 的对数概率值和 token_id；</p> <p><i>int32_t top_logporbs:</i> 选项设置返回概率最高的 token 数量，每个 token 附带对应的对数概率和 token_id，使用此参数时 logprobs 必须设置为 True；</p>
返回值	无

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.num_npu_core = 3;  
param.top_k = 1;  
param.max_new_tokens = 256;  
param.max_context_len = 512;
```

3.2.3 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，只有统一这 3 个流程中的 LLMHandle 句柄对象，才能够正确完成模型的推理流程。

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 3-7 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: 模型定义的结构体，详见 3.2.2 参数结构体 RKLLMParam 定义 ； LLMResultCallback callback: 用于接受处理模型实时输出的回调函数，详见 3.2.1 回调函数定义 ；
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);
```

3.2.4 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 3-8 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	<p>LLMHandle handle: 模型初始化注册的目标句柄，可见 3.2.3 初始化模型；</p> <p>const char* prompt: 模型推理的输入 prompt，即用户的“问题”，注意需要在问题前后加上预设的 prompt，保证推理正确性，详见示例代码；</p> <p>void* userdata: 用户自定义的函数指针，默认设置为 NULL 即可；</p>
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；

模型推理的示例代码如下：

```
// 提前定义 prompt 前后的文本预设值
#define PROMPT_TEXT_PREFIX "<|im_start|>system You are a helpful assistant. <|im_end|> <|im_start|>user"
#define PROMPT_TEXT_POSTFIX "<|im_end|><|im_start|>assistant"

// 定义输入的 prompt 并完成前后 prompt 的拼接
string input_str = "把这句话翻译成英文：RK3588 是新一代高端处理器，具有高算力、低功耗、超强多媒体、丰富数据接口等特点";
input_str = PROMPT_TEXT_PREFIX + input_str + PROMPT_TEXT_POSTFIX;

// 调用模型推理接口，其中 llmHandle 为模型初始化时传入的句柄
rkllm_run(llmHandle, input_str.c_str(), NULL);
```

3.2.5 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 3-9 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	<p>LLMHandle handle: 模型初始化注册的目标句柄，可见 3.2.3 初始化模型；</p>
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);
```

3.2.6 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 3-10 `rkllm_destroy` 函数接口说明

函数名	<code>rkllm_destroy</code>
描述	用于销毁 RKLLM 模型并释放所有计算资源。
参数	<i>LLMHandle handle</i> : 模型初始化注册的目标句柄，可见 3.2.3 初始化模型 ；
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);
```

3.3 板端推理调用示例

本节中提供板端推理调用的 C++ 工程在 `rkllm-runtime/examples/rkllm_api_demo` 路径下同步更新，同时提供了编译脚本，以使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

以下为推理调用的完整 C++ 代码示例，与工具链文件中的 `rkllm_api_demo/src/main.cpp` 相同，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。

```
// Copyright (c) 2024 by Rockchip Electronics Co., Ltd. All Rights Reserved.
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//     http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
```

```
// See the License for the specific language governing permissions and
// limitations under the License.

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

#define PROMPT_TEXT_PREFIX "<|im_start|>system You are a helpful
assistant. <|im_end|> <|im_start|>user"
#define PROMPT_TEXT_POSTFIX "<|im_end|><|im_start|>assistant"

using namespace std;
LLMHandle llmHandle = nullptr;
void exit_handler(int signal)
{
    if (llmHandle != nullptr)
    {
        {
            cout << "程序即将退出" << endl;
            LLMHandle _tmp = llmHandle;
            llmHandle = nullptr;
            rkllm_destroy(_tmp);
        }
        exit(signal);
    }
}

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

int main(int argc, char **argv)
{
    if (argc != 2)
    {
        printf("Usage:%s [rkllm_model_path]\n", argv[0]);
        return -1;
    }
    signal(SIGINT, exit_handler);
    string rkllm_model(argv[1]);
    printf("rkllm init start\n");
```

```
//设置参数及初始化
RKLLMParam param = rkllm_createDefaultParam();
param.model_path = rkllm_model.c_str();
param.target_platform = "rk3588";
param.num_npu_core = 3;
param.top_k = 1;
param.max_new_tokens = 256;
param.max_context_len = 512;
param.logprobs = false;
param.top_logprobs = 5;
rkllm_init(&llmHandle, param, callback);
printf("rkllm init success\n");

vector<string> pre_input;
pre_input.push_back("把下面的现代文翻译成文言文：到了春风和煦，阳光明媚的时候，湖面平静，没有惊涛骇浪，天色湖光相连，一片碧绿，广阔无际；沙洲上的鸥鸟，时而飞翔，时而停歇，美丽的鱼游来游去，岸上与小洲上的花草，青翠欲滴。");
pre_input.push_back("以咏梅为题目，帮我写一首古诗，要求包含梅花、白雪等元素。");
pre_input.push_back("上联：江边惯看千帆过");
pre_input.push_back("把这句话翻译成英文：RK3588 是新一代高端处理器，具有高算力、低功耗、超强多媒体、丰富数据接口等特点");
cout << "\n*****可输入以下问题对应序号获取回答/或自定义输入*****\n" << endl;
for (int i = 0; i < (int)pre_input.size(); i++)
{
    cout << "[" << i << "]" " " << pre_input[i] << endl;
}
cout << "\n*****\n" << endl;

string text;
while (true)
{
    std::string input_str;
    printf("\n");
    printf("user: ");
    std::getline(std::cin, input_str);
    if (input_str == "exit")
    {
        break;
    }
    for (int i = 0; i < (int)pre_input.size(); i++)
    {
        if (input_str == to_string(i))
        {
            input_str = pre_input[i];
            cout << input_str << endl;
        }
    }
    string text = PROMPT_TEXT_PREFIX + input_str +
PROMPT_TEXT_POSTFIX;
    printf("robot: ");
    rkllm_run(llmHandle, text.c_str(), NULL);
}
rkllm_destroy(llmHandle);
return 0;
}
```

3.3.2 示例工程的使用说明

rkllm_api_dem 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 rkllm_api_dem 目录下构建得到对应的 build/build_linux_aarch64_Release/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 build/build_linux_aarch64_Release/llm_demo /userdata/llm
adb push ../../runtime/Linux/librkllm_api/aarch64/librkllmrt.so
/userdata/llm/lib
adb push /PC/path/to/your/rkllm/model /board/path/to/your/rkllm/model
```

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/userdata/llm 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 /userdata/llm
export LD_LIBRARY_PATH=./lib # 通过环境变量指定函数库路径
taskset f0 ./llm_demo /path/to/your/rkllm/model
```

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。

3.4 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，特别设置了与 OpenAI-API 接口相同的收发结构体形式，便于用户在已有的开发基础上实现快速替换；

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 rkllm-runtime/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：

```
rkllm-runtime/examples/rkllm_server_demo
├── rkllm_server                                # 板端部署所需文件
│   ├── lib                                    # RKLLM Runtime 库
│   ├── fix_freq_rk3576.sh                     # rk3576 定频脚本
│   ├── fix_freq_rk3588.sh                     # rk3588 定频脚本
│   ├── 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。在 RKLLM-Server-Flask 的服务端和客户端实现中，特别考虑了 OpenAI-API 的调用方式，示例代码通过设置与 OpenAI-API 相同的数据结构体来保证实现的一致性，使得用户在完成 RKLLM-Server-Flask 的部署后，能够在已有的 OpenAI-API 开发基础上仅替换访问接口来快速实现服务迁移。

参考 [OpenAI-API 使用文档](#) 可知，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,
}
```

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；

而对于服务端回传的数据而言，OpenAI-API 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，最为重要的部分即为 **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“stop”才表示流式推理结束；

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，能够帮助已使用过 OpenAI-AIP 的用户实现快速替换。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）直接通过以下命令调用 `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：

```
# build_rkllm_server_flask.sh [目标平台:rk3588/rk3576] [工作路径] [已转换
的 rkllm 模型在板端的绝对路径]
build_rkllm_server_flask.sh rk3588 /path/to/workshop /path/to/model
```

在执行上述命令后，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 3-1 中的“RKLLM 初始化成功！”信息后，即说明 RKLLM-Server-Flask 示例的成功启动。

```
=====
RKLLM初始化成功!
=====
* Serving Flask app 'flask_server'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment.
Use a production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:8080
* Running on http://172.16.10.178:8080
Press CTRL+C to quit
█
```

图 3-1 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，以使用户参考当前 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。本节将对其中的主要代码进行说明介绍：

1) 设置动态库路径, 通过 ctypes 链接 RKLLM Runtime 的函数库 librkllmrt.so;

```
# 链接 RKLLM Runtime 函数库
llm_lib = ctypes.CDLL('lib/librkllmrt.so')
```

2) 定义 librkllmrt.so 动态库中的结构体, 实现 Python 代码与 C++函数库间的链接;

```
# 参考 rkllm.h 声明结构体定义
class Token(ctypes.Structure):
    _fields_ = [
        ("logprob", ctypes.c_float),
        ("id", ctypes.c_int32)
    ]

class RKLLMResult(ctypes.Structure):
    _fields_ = [
        ("text", ctypes.c_char_p),
        ("tokens", ctypes.POINTER(Token)),
        ("num", ctypes.c_int32)
    ]

class RKNNllmParam(ctypes.Structure):
    _fields_ = [
        ("model_path", ctypes.c_char_p),
        ("num_npu_core", ctypes.c_int32),
        ("max_context_len", ctypes.c_int32),
        ("max_new_tokens", ctypes.c_int32),
        ("top_k", ctypes.c_int32),
        ("top_p", ctypes.c_float),
        ("temperature", ctypes.c_float),
        ("repeat_penalty", ctypes.c_float),
        ("frequency_penalty", ctypes.c_float),
        ("presence_penalty", ctypes.c_float),
        ("mirostat", ctypes.c_int32),
        ("mirostat_tau", ctypes.c_float),
        ("mirostat_eta", ctypes.c_float),
        ("logprobs", ctypes.c_bool),
        ("top_logprobs", ctypes.c_int32),
        ("use_gpu", ctypes.c_bool)
    ]
```

3) 定义回调函数, 并完成与动态库间的函数链接, 不同的是, 在该回调函数中需要调用一些全局变量 global_text, global_state 和 split_byte_data, 这些变量是为了实现将 RKLLM 推理输出同步到 Flask 框架中进行流式输出的重要设计, 其中 split_byte_data 是为了解决回调函数所返回的文本中出现的编码中断问题;

```
# 定义全局变量, 用于保存回调函数的输出, 便于在 gradio 界面中输出
global_text = []
global_state = -1
split_byte_data = bytes(b"") # 用于保存分割的字节数据
```

```
# 定义回调函数
def callback(result, userdata, state):
    global global_text, global_state, split_byte_data
    if state == 0:
        # 保存输出的 token 文本及 RKLLM 运行状态
        global_state = state
        # 需要监控当前的字节数据是否完整，不完整则进行记录，后续进行解析
        try:
            global_text.append((split_byte_data +
result.contents.text).decode('utf-8'))
            print((split_byte_data + result.contents.text).decode('utf-
8'), end='')
            split_byte_data = bytes(b'')
        except:
            split_byte_data += result.contents.text
            sys.stdout.flush()
    elif state == 1:
        # 保存 RKLLM 运行状态
        global_state = state
        print("\n")
        sys.stdout.flush()
    else:
        print("run error")

# Python 端与 C++端的回调函数连接
callback_type = ctypes.CFUNCTYPE(None, ctypes.POINTER(RKLLMResult),
ctypes.c_void_p, ctypes.c_int)
c_callback = callback_type(callback)
```

4) 定义 Python 端的 RKLLM 类，其中包括了对动态库中 RKLLM 模型的初始化、推理及释放操作；其中，该部署示例直接进行了 RKLLM 模型的具体定义，用户可以自行修改初始化

init 部分的具体的参数；此外，当用户在实现自定义 Server 的过程中，可以选择更为合适的

定义方式进行替换，只需保持 Python 在函数定义时需要参考头文件给出相匹配的定义即可；

```
# 定义 RKLLM_Handle_t 和 userdata
RKLLM_Handle_t = ctypes.c_void_p
userdata = ctypes.c_void_p(None)

# 设置提示文本
PROMPT_TEXT_PREFIX = "<|im_start|>system You are a helpful assistant.
<|im_end|> <|im_start|>user"
PROMPT_TEXT_POSTFIX = "<|im_end|><|im_start|>assistant"

# 定义动态库中的 RKLLM 类
class RKLLM(object):
    def __init__(self, model_path, target_platform):
        rknnllm_param = RKNNllmParam()
        rknnllm_param.model_path = bytes(model_path, 'utf-8')
        if target_platform == "rk3588":
            rknnllm_param.num_npu_core = 3
        elif target_platform == "rk3576":
            rknnllm_param.num_npu_core = 2
```

```

rknnllm_param.max_context_len = 320
rknnllm_param.max_new_tokens = 512
rknnllm_param.top_k = 1
rknnllm_param.top_p = 0.9
rknnllm_param.temperature = 0.8
rknnllm_param.repeat_penalty = 1.1
rknnllm_param.frequency_penalty = 0.0
rknnllm_param.presence_penalty = 0.0
rknnllm_param.mirostat = 0
rknnllm_param.mirostat_tau = 5.0
rknnllm_param.mirostat_eta = 0.1
rknnllm_param.logprobs = False
rknnllm_param.top_logprobs = 5
rknnllm_param.use_gpu = True
self.handle = RKLLM_Handle_t()

self.rkllm_init = rkllm_lib.rkllm_init
self.rkllm_init.argtypes = [ctypes.POINTER(RKLLM_Handle_t),
ctypes.POINTER(RKNNllmParam), callback_type]
self.rkllm_init.restype = ctypes.c_int
self.rkllm_init(ctypes.byref(self.handle), rknnllm_param,
c_callback)

self.rkllm_run = rkllm_lib.rkllm_run
self.rkllm_run.argtypes = [RKLLM_Handle_t,
ctypes.POINTER(ctypes.c_char), ctypes.c_void_p]
self.rkllm_run.restype = ctypes.c_int
self.rkllm_destroy = rkllm_lib.rkllm_destroy
self.rkllm_destroy.argtypes = [RKLLM_Handle_t]
self.rkllm_destroy.restype = ctypes.c_int

def run(self, prompt):
    prompt = bytes(PROMPT_TEXT_PREFIX + prompt +
PROMPT_TEXT_POSTFIX, 'utf-8')
    self.rkllm_run(self.handle, prompt, ctypes.byref(userdata))
    return

def release(self):
    self.rkllm_destroy(self.handle)

```

- 5) RKLLM 模型的初始化，通过调用以上定义的 RKLLM 类别完成 RKLLM 的初始化，并通
过该 RKLLM 类进行模型的推理及释放；

```

# 初始化 RKLLM 模型
print("=====init....=====")
sys.stdout.flush()
target_platform = args.target_platform
model_path = args.rkllm_model_path
rkllm_model = RKLLM(model_path, target_platform)
print("RKLLM 初始化成功! ")
print("=====")
sys.stdout.flush()

```

- 6) 使用 Flask 搭建 RKLLM-Server-Flask，完成对客户端传回的输入结构体的解析、处理，并
通过此前设计的全局变量 global_text，global_state 和 split_byte_data 将 RKLLM 模型的推理输
出链接至 Flask 框架中，最后将数据以特定的结构体包装并发送至客户端；其中，该部分包

含了非流式推理和流式推理的两部分设计，用于响应用户的不同访问需求；

```
# 创建一个函数用于接受用户使用 request 发送的数据
@app.route('/rkllm_chat', methods=['POST'])
def receive_message():
    # 链接全局变量，获取回调函数的输出信息
    global global_text, global_state
    global is_blocking

    # 如果服务器正在阻塞状态，则返回特定响应
    if is_blocking or global_state==0:
        return jsonify({'status': 'error', 'message': 'RKLLM_Server is
busy! Maybe you can try again later.'}), 503

    # 加锁
    lock.acquire()
    try:
        # 设置服务器为阻塞状态
        is_blocking = True

        # 获取 POST 请求中的 JSON 数据
        data = request.json
        if data and 'messages' in data:
            # 重置全局变量
            global_text = []
            global_state = -1

            # 定义返回的结构体
            rkllm_responses = {
                "id": "rkllm_chat",
                "object": "rkllm_chat",
                "created": None,
                "choices": [],
                "usage": {
                    "prompt_tokens": None,
                    "completion_tokens": None,
                    "total_tokens": None
                }
            }

            if not "stream" in data.keys() or data["stream"] == False:
                # 在这里处理收到的数据
                messages = data['messages']
                print("Received messages:", messages)
                for index, message in enumerate(messages):
                    input_prompt = message['content']
                    rkllm_output = ""
                    # 创建模型推理的线程
                    model_thread =
threading.Thread(target=rkllm_model.run, args=(input_prompt,))
                    model_thread.start()

                    # 等待模型运行完成，定时检查模型的推理线程
                    model_thread_finished = False
```

```

        while not model_thread_finished:
            while len(global_text) > 0:
                rkllm_output += global_text.pop(0)
                time.sleep(0.005)

            model_thread.join(timeout=0.005)
            model_thread_finished = not
model_thread.is_alive()

        rkllm_responses["choices"].append(
            {"index": index,
             "message": {
                 "role": "assistant",
                 "content": rkllm_output,
             },
             "logprobs": None,
             "finish_reason": "stop"
            }
        )
    return jsonify(rkllm_responses), 200
else:
    # 在这里处理收到的数据
    messages = data['messages']
    print("Received messages:", messages)
    for index, message in enumerate(messages):
        input_prompt = message['content']
        rkllm_output = ""

        def generate():
            # 创建模型推理的线程
            model_thread =
threading.Thread(target=rkllm_model.run, args=(input_prompt,))
            model_thread.start()

            # 等待模型运行完成, 定时检查模型的推理线程
            model_thread_finished = False
            while not model_thread_finished:
                while len(global_text) > 0:
                    rkllm_output = global_text.pop(0)

                    rkllm_responses["choices"].append(
                        {"index": index,
                         "delta": {
                             "role": "assistant",
                             "content": rkllm_output,
                         },
                         "logprobs": None,
                         "finish_reason": "stop" if global_state
== 1 else None,
                        }
                    )
                    yield f"{json.dumps(rkllm_responses)}\n\n"

                model_thread.join(timeout=0.005)
                model_thread_finished = not
model_thread.is_alive()

            return Response(generate(), content_type='text/plain')

```

```

        else:
            return jsonify({'status': 'error', 'message': 'Invalid JSON
data!'}), 400
    finally:
        # 释放锁
        lock.release()
        # 将服务器状态设置为非阻塞
        is_blocking = False

# 启动 Flask 应用程序
app.run(host='0.0.0.0', port=8080, threaded=True, debug=False)

```

以上对各模块的介绍即构成了 `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` 中展示了对于上述 RKLLM-Server-Flask 服务端的 API 访问示例，用户在进行自定义功能的开发的过程中，只需参考该 API 访问示例使用对应的收发结构体进行数据的包装、解析即可；由于其中的收发数据结构体均参考 OpenAI-API 的设计，因此对于此前已使用 OpenAI-API 进行开发的用户而言，仅需替换相应的网络接口即可。本小节将对其中的重要代码进行说明：

- 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://172.16.10.166:8080/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
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 messages 加入多个问题
# stream: 是否开启流式对话，与 OpenAI 接口相同
data = {
    "model": 'your_model_deploy_with_RKLLM_Server',
    "messages": [{"role": "user", "content": user_message}],
    "stream": is_streaming
}
```

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) 定义对回传数据结构体的解析方式，由于 RKLLM-Server-Flask 将参考 OpenAI-API 回传结构体的形式，在实际使用中，需要进行解析操作；

```
# 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。

3.4.2 RKLLM-Server-Gradio 部署示例介绍

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，以下的图 3-2 展示了 RKLLM-Server-Gradio 部署成功后的网页端示例：

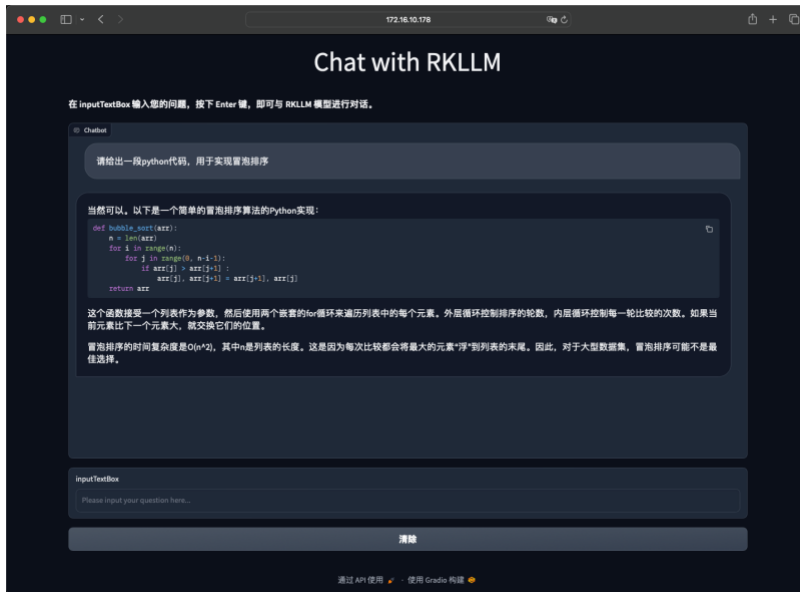


图 3-2 RKLLM-Server 部署示例的外部访问页面

在当前的 `rkllm_server_demo` 目录下，已包含了上图 3-2 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：

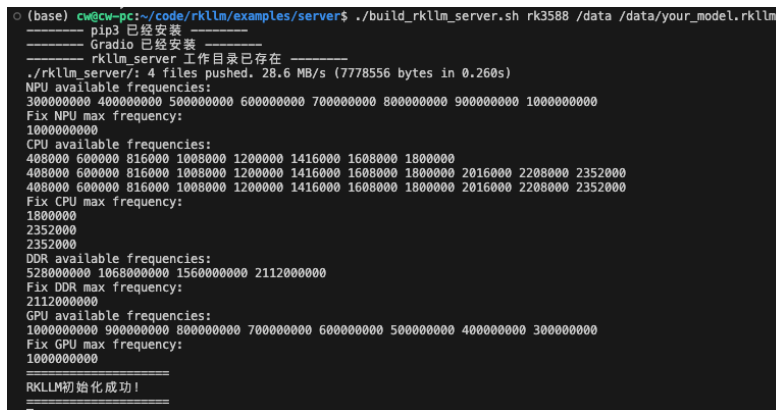
- 1) 为开发板接入网线，并在 `adb shell` 下通过 `ifconfig` 指令查询该开发板的具体 IP，后续 RKLLM-Server-Gradio 将在局域网内以该 IP 设置服务端并接受客户端的访问；
- 2) 用户需要提前完成 RKLLM 模型的顺利转换，并在执行一键部署脚本前，已将 RKLLM

模型推送至 Linux 板端；

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

```
# build_rkllm_server_gradio.sh [目标平台:rk3588/rk3576] [工作路径] [已转换的 rkllm 在板端的绝对路径]
build_rkllm_server_gradio.sh rk3588 /path/to/workshop /path/to/model
```

在执行上述命令后，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 3-3 中的“RKLLM 初始化成功！”信息后，即说明 RKLLM-Server 示例的成功启动。



```
(base) cw@cw-pc:~/code/rkllm/examples/server$ ./build_rkllm_server.sh rk3588 /data /data/your_model.rkllm
=====
pip3 已经安装
=====
Gradio 已经安装
=====
rkllm_server 工作目录已存在
=====
./rkllm_server/: 4 files pushed, 28.6 MB/s (7778556 bytes in 0.260s)
NPU available frequencies:
300000000 400000000 500000000 600000000 700000000 800000000 900000000 1000000000
Fix NPU max frequency:
1000000000
CPU available frequencies:
408000 600000 816000 1008000 1200000 1416000 1608000 1800000
408000 600000 816000 1008000 1200000 1416000 1608000 1800000 2016000 2208000 2352000
408000 600000 816000 1008000 1200000 1416000 1608000 1800000 2016000 2208000 2352000
Fix CPU max frequency:
1800000
2352000
2352000
DDR available frequencies:
528000000 1068000000 1560000000 2112000000
Fix DDR max frequency:
2112000000
GPU available frequencies:
1000000000 900000000 800000000 700000000 600000000 500000000 400000000 300000000
Fix GPU max frequency:
1000000000
=====
RKLLM初始化成功!
=====
```

图 3-3 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；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；

- 1) 设计接收用户输入的处理函数以及进行流式打印的输出函数，后续在 `gradio` 界面的设计中将直接调用此处的输入输出函数：

```
# 记录用户输入的 prompt
def get_user_input(user_message, history):
    history = history + [[user_message, None]]
    return "", history

# 获取 RKLLM 模型的输出并进行流式打印
def get_RKLLM_output(history):
    # 链接全局变量，获取回调函数的输出信息
    global global_text, global_state
    global_text = []
    global_state = -1

    # 创建模型推理的线程
    model_thread = threading.Thread(target=rkllm_model.run,
    args=(history[-1][0],))
    model_thread.start()

    # history[-1][1]表示当前的输出对话
    history[-1][1] = ""

    # 等待模型运行完成，定时检查模型的推理线程
    model_thread_finished = False
    while not model_thread_finished:
        while len(global_text) > 0:
            history[-1][1] += global_text.pop(0)
            time.sleep(0.005)
            # gradio 在调用 then 方法自动将 yield 返回的结果推进行输出
            yield history

        model_thread.join(timeout=0.005)
        model_thread_finished = not model_thread.is_alive()
```

- 2) 调用 `gradio` 库进行网页端界面的基本设计，并链接此前定义的输出输出处理函数，实现快速简洁的用户交互功能：

```
# 创建 gradio 界面
with gr.Blocks(title="Chat with RKLLM") as chatRKLLM:
    gr.Markdown("<div align='center'><font size='70'> Chat with RKLLM</font></div>")
    gr.Markdown("### 在 inputTextBox 输入您的问题，按下 Enter 键，即可与 RKLLM 模型进行对话。")
    # 创建一个 Chatbot 组件，用于显示对话历史
    rkllmServer = gr.Chatbot(height=600)
    # 创建一个 Textbox 组件，让用户输入消息
    msg = gr.Textbox(placeholder="Please input your question here...",
    label="inputTextBox")
    # 创建一个 Button 组件，用于清除聊天历史
    clear = gr.Button("清除")
```

```
# 将用户输入的消息提交给 get_user_input 函数，并且立即更新聊天历史
# 然后调用 get_RKLLM_output 函数，进一步更新聊天历史
# queue=False 参数确保这些更新不会被放入队列，而是立即执行
msg.submit(get_user_input, [msg, rkllmServer], [msg, rkllmServer],
queue=False).then(get_RKLLM_output, rkllmServer, rkllmServer)
# 当点击清除按钮时，执行一个空操作（lambda: None），并且立即清除聊天历史
clear.click(lambda: None, None, rkllmServer, queue=False)

# 启用事件队列系统
chatRKLLM.queue()
# 启动 Gradio 应用程序
chatRKLLM.launch()
```

以上对各模块的介绍即构成了 rkllm_server/gradio_server.py 的主体代码，从而完成了对 RKLLM-Server-Gradio 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，可以参考以上 RKLLM-Server-Gradio 部署示例的实现，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“172.16.10.178:8080”）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 3-4 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://172.16.10.169:8080/")
```

```
○ (base) [redacted] /rkllm/rkllm-runtime/examples/rkllm_server_demo$ python chat_api.py
=====
在终端中输入您的问题，即可与 RKLLM 模型进行对话....
=====
请输入您的问题：请给出福州的介绍
Loaded as API: http://172.16.10.102:8080/ ✓
Q: 请给出福州的介绍
A: 福州，福建省省会，位于中国东南沿海的大陆架上，东临太平洋，西隔台湾海峡与台湾岛相望；北濒东海，南界巴士海峡与菲律宾群岛相对。全市总面积13247平方千米，其中市区面积6580平方千米。福州市区气候温暖湿润，四季分明，是著名的“鱼米之乡”。福州是中国东南沿海重要的港口城市和风景名胜区，拥有众多的文化遗产和自然景观。
请输入您的问题： █
```

图 3-4 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>

Rockchip