

Classification Level: Top Secret() Secret() Internal() Public(✓)

RKLLM SDK User Guide

(Graphic Computing Platform Center)

Mark:	Version:	1.0.1
[] Changing	Author:	AI Group
[✓] Released	Completed Date:	8/May/2024
	Reviewer:	Vincent
	Reviewed Date:	8/May/2024

瑞芯微电子股份有限公司

Rockchip Electronics Co., Ltd.

(Copyright Reserved)

Revision History

Version	Modifier	Date	Modify description	Reviewer
V1.0.0	AI Group	2024-3-23	Initial version	Vincent
V1.0.1	AI Group	2024-5-8	1. Optimize memory usage. 2. Optimize inference time. 3. Optimize quantization accuracy. 4. Support Gemma, Phi-3 and other models. 5. Add Server call and interrupt interface.	Vincent

Table of contents

1 Introduction to RKLLM.....	5
1.1 Introduction to RKLLM Toolchain.....	5
1.1.1 Introduction to RKLLM-Toolkit.....	5
1.1.2 Introduction to RKLLM Runtime	5
1.2 Introduction to the RKLLM Development Process	5
1.3 Applicable Hardware Platforms.....	6
2 Development Environment Preparation	7
2.1 Installation of RKLLM-Toolkit	7
2.1.1 Installation via Pip.....	8
2.2 Introduction for RKLLM Runtime Usage.....	9
2.3 Compilation Requirements for RKLLM Runtime	9
2.4 Kernel Update.....	9
3 RKLLM Instruction Guide	11
3.1 Model Conversion.....	11
3.1.1 RKLLM Initialization.....	11
3.1.2 Loading Models	11
3.1.3 Quantization Construction for RKLLM Model	12
3.1.4 Exporting RKLLM Model.....	13
3.2 Inference Implementation in Board-side.....	13
3.2.1 Define Callback Function	14
3.2.2 Define RKLLMParam	15
3.2.3 Initialize Model.....	18
3.2.4 Inference Model	19
3.2.5 Interrupt Model Inference.....	19
3.2.6 Release Model.....	20

3.3 Board-side Inference Example	20
3.3.1 Complete Code of Example Project	21
3.3.2 Instructions for Example Project	23
3.4 Implementation of Board-side Server	24
3.4.1 Deployment Example of RKLLM-Server-Flask	25
3.4.2 Deployment Example of RKLLM-Server-Gradio	36
4 Reference.....	42

1 Introduction to RKLLM

1.1 Introduction to RKLLM Toolchain

1.1.1 Introduction to RKLLM-Toolkit

RKLLM-Toolkit is a development kit that provides users with the ability to quantify and convert large language models on the PC. Through the Python interface provided by the tool, the following functions can be conveniently accomplished:

1) Model Conversion: It supports the conversion of Large Language Model (LLM) in Hugging Face format to RKLLM model, and the supported models include LLaMA, Qwen, Qwen2, Phi-2, Phi-3, ChatGLM3, Gemma, InternLM2 and MiniCPM. ChatGLM3, Gemma, InternLM2 and MiniCPM. The converted RKLLM models can be loaded and used on Rockchip NPU platform.

2) Quantization Function: Supports quantization of floating-point models into fixed-point models. Currently supported quantization types include w4a16 and w8a8.

1.1.2 Introduction to RKLLM Runtime

The RKLLM Runtime is primarily responsible for loading the RKLLM models obtained from the RKLLM-Toolkit conversion and executing inference via the Rockchip NPU drivers embedded in the RK3576/RK3588 series. Throughout the RKLLM model inference process, users retain the autonomy to specify inference parameter configurations, define various text generation methodologies, and seamlessly retrieve inference results from the model via pre-defined callback functions.

1.2 Introduction to the RKLLM Development Process

The overall development process of RKLLM primarily comprises two phases: model conversion in PC and deployment and execution in board-side:

1) Model Conversion in PC:

During this phase, the provided Hugging Face formatted LLM is converted into RKLLM format for efficient inference on the Rockchip NPU platform. This step includes:

a. Model Acquisition: Obtain the large language model in Hugging Face format, either directly from the Hugging Face platform or one trained independently, ensuring consistency in model structure preservation.

b. Model Loading: Load the original model using the `rkllm.load_huggingface()` function.

c. Model Quantization Configuration: Utilize the `rkllm.build()` function to construct the RKLLM model, with the option to perform model quantization for enhanced hardware deployment performance. Additionally, choose different optimization levels and quantization types during the construction process.

d. Model Export: Export the RKLLM model as a `.rkllm` format file using the `rkllm.export_rkllm()` function for subsequent deployment.

2) Deployment and Execution in Board-side:

This phase encompasses the actual deployment and execution of the model, typically involving the following steps:

a. Model Initialization: Load the RKLLM model onto the Rockchip NPU platform, configure model parameters accordingly to define the desired text generation methods, and pre-define callback functions to receive real-time inference results, preparing for inference.

b. Model Inference: Execute the inference operation by passing input data to the model and running model inference. Users can continuously retrieve inference results through pre-defined callback functions.

c. Model Release: After completing the inference process, release the model resources to allow other tasks to utilize the computational resources of the NPU.

These two steps constitute the complete RKLLM development process, ensuring successful conversion, debugging, and ultimately efficient deployment of the LLMs on the Rockchip NPU.

1.3 Applicable Hardware Platforms

The hardware platforms to which this document applies mainly include: RK3576, RK3588

2 Development Environment Preparation

The RKLLM toolchain zip file contains the whl installer for the RKLLM-Toolkit, the associated files of the RKLLM Runtime library and reference sample code. The specific folder structure is shown below:

```
doc
├── Rockchip_RKLLM_SDK_CN.pdf      # RKLLM SDK Documentation (Chinese)
├── Rockchip_RKLLM_SDK_CN.pdf      # RKLLM SDK Documentation (English)
rkllm-runtime
├── examples
│   ├── rkllm_api_demo             # Board-side Inference Example
│   └── rkllm_server_demo          # RKLLM-Server Deployment Example
├── runtime
│   ├── Android
│   │   ├── librkllm_api
│   │   │   └── arm64-v8a
│   │   │       ├── librkllmrt.so  # RKLLM Runtime Library
│   │   │       └── include
│   │   │           └── rkllm.h    # Header File
│   └── Linux
│       ├── librkllm_api
│       │   └── aarch64
│       │       ├── librkllmrt.so  # RKLLM Runtime Library
│       │       └── include
│       │           └── rkllm.h    # Header File
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
```

This chapter provides detailed instructions for installing the RKLLM-Toolkit and RKLLM Runtime. For specific usage instructions, please refer to the instructions in Chapter 3.

2.1 Installation of RKLLM-Toolkit

This section mainly describes how to install the RKLLM-Toolkit with pip install command. Users can refer to the following detailed instructions to complete the installation of the RKLLM-Toolkit toolchain.

2.1.1 Installation via Pip

2.1.1.1 Installation of Miniforge3

To avoid the need for multiple versions of Python environments, it is recommended to use miniforge3 to manage your Python environments.

Check if miniforge3 is installed and the version information of conda. If it is already installed, you can skip this section.

```
conda -V
# If "conda: command not found" is displayed, it indicates that conda
is not installed.
# For example, version information could be displayed as conda 23.9.0
```

Download the miniforge3 installation package.

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

Install miniforge3.

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

2.1.1.2 Create RKLLM-Toolkit Conda Environment

Activate the Conda base environment.

```
source ~/miniforge3/bin/activate # directory of miniforge3
# (base) xxx@xxx-pc:~$
```

Create a Conda environment named RKLLM-Toolkit with Python version 3.8 (recommended).

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

Activate the RKLLM-Toolkit Conda environment.

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

2.1.1.3 Installing RKLLM-Toolkit

In the RKLLM-Toolkit Conda environment, use the pip command to install the provided toolchain .whl package directly. During the installation process, the installer will automatically download the necessary dependencies for the RKLLM-Toolkit tools.

```
pip3 install rkllm_toolkit-x.x.x-cp38-cp38-linux_x86_64.whl
```

If executing the following command does not result in any errors, the installation is successful.

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


2.2 Introduction for RKLLM Runtime Usage

Within the RKLLM toolchain files provided, the following components are included for the RKLLM runtime:

- 1) lib/librkllmrt.so: RKLLM Runtime library suitable for RK3576/RK3588 series.
- 2) include/rkllm_api.h: Corresponding header file to librkllmrt.so, containing descriptions of related structures and function definitions.

When constructing deployment inference code for RK3576/RK3588 series using the RKLLM toolchain, it is critical to ensure proper linkage to the above header file and function library to ensure compilation correctness. During the actual runtime of the code on RK3576/RK3588 boards, it is equally important to ensure successful transfer of the above function library files to the board and complete library declaration through the following environment variable settings:

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

2.3 Compilation Requirements for RKLLM Runtime

When utilizing RKLLM Runtime, it's essential to pay attention to the version of the gcc compilation tool. We recommend the cross-compilation tool [gcc-arm-10.2-2020.11-x86_64-aarch64-none-linux-gnu](#). Please note that cross-compilation tools often have backward compatibility but not upward compatibility, so refrain from using versions below 10.2.

If you choose to use the Android platform, and you need to compile Android executable files, we recommend using the Android NDK tool for cross-compilation. You can download it from the following link: [Android NDK Cross-Compilation Tool Download Link](#). We recommend using version r21e.

You can also refer to the specific compilation methods in the compilation scripts located in the rkllm/rkllm-runtime/examples/rkllm_api_demo directory.

2.4 Kernel Update

Due to the requirement of a higher version of the NPU kernel for the provided RKLLM, users need to verify that the NPU kernel on the board is version v0.9.6 before using RKLLM Runtime for model inference. The specific query command is as follows:

```
# Execute the following command to query the NPU kernel version.  
cat /sys/kernel/debug/rknpu/version  
  
# Confirm that the command output is as follows:  
# RKNPU driver: v0.9.6
```

If the queried NPU kernel version is lower than v0.9.6, please proceed to the official firmware address to download the latest firmware for updating.

For users using non-official firmware, it's necessary to update the kernel. The RKNPU driver package supports two main kernel versions: kernel-5.10 and kernel-6.1. For kernel-5.10, it is recommended to use a specific version at [GitHub-rockchip-linux/kernelatdevelop-5.10](https://github.com/rockchip-linux/kernelatdevelop-5.10). For kernel 6.1, it is recommended to use a specific version such as 6.1.57. Users can confirm the specific version number in the Makefile in the kernel's root directory. The specific steps to upgrade the kernel are as follows:

- 1) Download the [rknpu_driver_0.9.6_20240322.tar.bz2](#).
- 2) Unzip the compressed file and overwrite the RKNPU driver code into the current kernel code directory.
- 3) Recompile the kernel.
- 4) Flash the newly compiled kernel to the device.

3 RKLLM Instruction Guide

3.1 Model Conversion

RKLLM-Toolkit provides model conversion and quantization functionalities. As one of the core features of RKLLM-Toolkit, it allows users to convert LLMs in Hugging Face format into RKLLM models, enabling the deployment and execution of RKLLM models on Rockchip NPU. This section will focus on the specific implementation of model conversion by RKLLM-Toolkit for LLMs, serving as a reference for users.

3.1.1 RKLLM Initialization

In this section, users need to initialize the RKLLM object, which is the first step in the entire workflow. In the example code, use the RKLLM() constructor function to initialize the RKLLM object.

```
rkllm = RKLLM()
```

3.1.2 Loading Models

After initializing RKLLM, users need to call the rkllm.load_huggingface() function to pass the specific path of the model. RKLLM-Toolkit will then successfully load the large language model in Hugging Face format based on the corresponding path, enabling subsequent conversion and quantization operations. The specific function definition is as follows:

Table 3-1 Interface Specification for the load_huggingface Function

Fuctionom	load_huggingface
Introduction	Used to load open-source LLMs in Hugging Face format.
Parameters	model: The path where the LLM files are stored, used for loading the model for subsequent conversion and quantization.
Returns	0 indicates successful model loading; -1 indicates model loading failure.

The example code is as follows:

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

3.1.3 Quantization Construction for RKLLM Model

After loading the original model through the `rkllm.load_huggingface()` function, the next step is to build the RKLLM model using the `rkllm.build()` function. During model conversation, users can choose whether to perform quantization, which helps reduce the model size and improve inference performance on Rockchip NPU. The specific definition of the `rkllm.build()` function is as follows:

Table 3-2 Interface Specification for the build Function

Fuctionom	build
Introduction	Used to construct the RKLLM model and define specific quantization operations during the conversion process.
Parameters	<p><i>do_quantization</i>: This parameter controls whether to perform quantization operations on the model, and it is recommended to set it to True.</p> <p><i>optimization_level</i>: This parameter is used to set whether to perform quantization precision optimization. The available settings are {0, 1}, where 0 means no optimization and 1 means precision optimization. Precision optimization may cause a decrease in model inference performance.</p> <p><i>quantized_dtype</i>: This parameter is used to set the specific type of quantization. Currently supported quantization types include</p>
Returns	<p>0 indicates successful model conversion and quantization;</p> <p>-1 indicates model conversion failure.</p>

The example code is as follows:

```
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 Exporting RKLLM Model

After constructing the RKLLM model using the `rkllm.build()` function, users can save the RKNN model as a `.rkllm` file for subsequent model deployment using the `rkllm.export_rkllm()` function. The specific parameter definition of the `rkllm.export_rkllm()` function is as follows:

Table 3-3 Interface Specification for the `export_rkllm` Function

Fuctionom	<code>export_rkllm</code>
Introduction	Used to save the converted and quantized RKLLM model for subsequent inference calls.
Parameters	<i>export_path</i> : The file path for saving the exported RKLLM model file.
Returns	0 indicates successful export and saving of the model; -1 indicates model export failure.

The example code is as follows:

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

The above operations cover all steps of model conversion and quantization in the RKLLM-Toolkit. Depending on different requirements and application scenarios, users can choose different configuration options and quantization methods for customised settings, which facilitates subsequent deployment.

3.2 Inference Implementation in Board-side

This chapter introduces the usage of the general API interface functions. Users can refer to the content of this chapter to construct C++ code and implement inference of RKLLM models on the board to obtain inference results. The RKLLM board-side inference implementation is as follows:

- 1) Define the callback function `callback()`.
- 2) Define the RKLLM model parameter structure `RKLLMParam`.
- 3) Initialize the RKLLM model with `rkllm_init()`.
- 4) Perform model inference with `rkllm_run()`.
- 5) Process the real-time inference results returned by the callback function `callback()`.

6) Destroy the RKLLM model and release resources with `rkllm_destroy()`.

In the subsequent parts of this chapter, the document will provide detailed explanations of each step in the process and provide detailed explanations of the functions involved.

3.2.1 Define Callback Function

The callback function is used to receive real-time output results from the RKLLM model. It is bound during the initialization of RKLLM and continuously outputs results to the callback function during the RKLLM model inference process, returning only one token each time.

Here is an example code snippet where the callback function prints the output results in real-time to the terminal:

```
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 is a status flag, and its specific definition is as follows:

Table 3-4 Explanation of LLMCallState Status Flags

Variable Name	LLMCallState
Introduction	Used to indicate the current running state of RKLLM.
Parameters	0 , <i>LLM_RUN_NORMAL</i> , indicates that the RKLLM model is currently inferencing; 1 , <i>LLM_RUN_FINISH</i> , indicates that the RKLLM model has completed inference; 2 , <i>LLM_RUN_ERROR</i> , indicates that an error has occurred during inference;
Returns	None

During the design process of the callback function, users can set different post-processing

behaviors based on the different states of LLMCallState.

2) RKLLMResult is a return value structure, and its specific definition is as follows:

Table 3-5 Explanation of RKLLMResult Structure

Variable Name	RKLLMResult
Introduction	Used to return the current inference-generated result.
Parameters	<p>0, <i>text</i>, indicates the text content generated by the current inference;</p> <p>1, <i>tokens</i>, indicates the top tokens with the highest probability at the current token position, where each token includes two attributes: <i>id</i> and <i>logprob</i>;</p> <p>2, <i>num</i>, indicates the number of tokens in the result;</p>
Returns	None

During the design process of the callback function, users can set different post-processing behaviors based on the values in RKLLMResult.

3.2.2 Define RKLLMPParam

The RKLLMPParam structure is used to describe and define the detailed information of RKLLM. The specific definition is as follows:

Table 3-6 Explanation of RKLLMPParam Structure

Variable Name	RKLLMPParam
Introduction	Used to define the detailed parameters of the RKLLM model.
Parameters	<p><i>const char* model_path</i>: the path to the RKLLM model file;</p> <p><i>int32_t num_npu_core</i>: the number of NPU cores used during model inference; The "rk3576" platform has configurable range [1, 2]; while the "rk3588" is [1, 3];</p> <p><i>bool use_gpu</i>: whether to use GPU for prefill acceleration, default option is false;</p> <p><i>int32_t max_context_len</i>: the context size of the prompt;</p> <p><i>int32_t max_new_tokens</i>: the maximum number of generated tokens in inferencing;</p>

	<p><i>int32_t top_k:</i> top-k sampling is a text generation method that selects the next token only from the top-k tokens with the highest probabilities predicted by the model. This method helps reduce the risk of generating low-probability or meaningless tokens. A higher top-k value (e.g., 100) will consider more token choices, resulting in more diverse text generation, while a lower value (e.g., 10) will focus on the most probable tokens, generating more conservative text. The default value is 40;</p> <p><i>float top_p:</i> top-p sampling, also known as nucleus sampling, is another text generation method that selects the next token from a group of tokens with cumulative probabilities of at least p. This method balances diversity and quality by considering the probabilities of tokens and the number of sampled tokens. A higher top-p value (e.g., 0.95) results in more diverse text generation, while a lower value (e.g., 0.5) generates more focused and conservative text. The default value is 0.9;</p> <p><i>float temperature:</i> a hyperparameter that controls the randomness of generated text by adjusting the probability distribution of output tokens. A higher temperature (e.g., 1.5) makes the output more random and creative. When the temperature is high, the model considers more options with lower probabilities when selecting the next token, resulting in more diverse and unexpected outputs. A lower temperature (e.g., 0.5) makes the output more focused and conservative. Lower temperatures mean that the model is more likely to choose high-probability tokens, resulting in more consistent and predictable outputs. In the extreme case of a temperature of 0, the model always chooses the most probable next token, resulting in identical outputs every time. To balance randomness and determinism and ensure that the output is neither overly uniform and predictable nor overly random and chaotic, the default value is 0.8;</p> <p><i>float repeat_penalty:</i> controls the occurrence of token sequence repetitions in the generated text, helping to prevent the model from generating repetitive or</p>
--	---

	<p>monotonous text. A higher value (e.g., 1.5) imposes a stronger penalty on repetitions, while a lower value (e.g., 0.9) is more lenient. The default value is 1.1;</p> <p><i>float frequency_penalty</i>: a factor for penalizing word/phrase repetition, reducing the probability of using words/phrases with higher frequencies overall and increasing the likelihood of using those with lower frequencies. This may lead to more diversified generated text, but could also result in text that is difficult to understand or not as expected. The range is [-2.0, 2.0], with a default value of 0;</p> <p><i>int32_t mirostat</i>: an algorithm actively maintaining the quality of generated text within the expected range during the text generation process. It aims to find a balance between coherence and diversity, avoiding low-quality output caused by excessive repetition (boredom trap) or incoherence (confusion trap). The values space is {0, 1, 2}, where 0 indicates not activating the algorithm, 1 indicates using the mirostat algorithm, and 2 indicates using the mirostat 2.0 algorithm;</p> <p><i>float mirostat_tau</i>: an option setting the target entropy for mirostat, representing the expected perplexity value of the generated text. Adjusting the target entropy allows to control the balance between coherence and diversity in the generated text. Lower values will result in more concentrated and coherent text, while higher values will lead to more diversified text, possibly with lower coherence. The default value is 5.0;</p> <p><i>float mirostat_eta</i>: an option setting the learning rate for mirostat, which influences the algorithm's responsiveness to feedback on generated text. A lower learning rate will result in slower adjustment, while a higher learning rate will make the algorithm more sensitive. The default value is 0.1;</p> <p><i>bool logprobs</i>: an option setting whether to return the log probability values and token_ids of output tokens;</p> <p><i>int32_t top_logporbs</i>: an option setting the number of tokens with the highest probabilities to return, with each token accompanied by its corresponding log</p>
--	---

	probability and token_id. When using this parameter, logprobs must be set to True.
Returns	None

In actual code construction, RKLLMParam needs to call the rkllm_createDefaultParam() function to initialize its definition, and set the corresponding model parameters according to requirements. Sample code is as follows:

```
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 Initialize Model

Before initializing the model, it is necessary to define the LLMHandle handle in advance. This handle is used for the initialization, inference, and resource release processes of the model. It's important to note that only by unifying the LLMHandle handle object across these three processes can the inference process of the model be completed correctly.

Prior to model inference, users need to complete the model initialization through the rkllm_init() function. The specific function definition is as follows:

Table 3-7 Interface Specification for the rkllm_init Function

Fuctionom	rkllm_init
Introduction	Used to initialize the parameters and related inference settings for RKLLM model.
Parameters	<p>LLMHandle* handle: register the model to the corresponding handle for subsequent inference and release calls;</p> <p>RKLLMParam param: the parameter structure defined for the model;</p> <p>LLMResultCallback callback: a callback function used to receive and process real-time outputs from the model;</p>
Returns	<p>0 indicates that the initialization process is normal;</p> <p>-1 indicates initialization failure;</p>

The example code is as follows:

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

3.2.4 Inference Model

After completing the initialization process of the RKLLM model, users can perform model inference using the `rkllm_run()` function. Real-time inference results can be processed using the callback function predefined during initialization. The specific function definition of `rkllm_run()` is as follows:

Table 3-8 Interface Specification for the `rkllm_run` Function

Fuctionom	rkllm_run
Introduction	Used to performing result inference using the initialized RKLLM model.
Parameters	LLMHandle handle: the target handle registered during model initialization; const char* prompt: the input prompt for model inference, which is the user's "question". Ensure to add the predefined prompt before and after the question to ensure the correctness of the inference; void* userdata: the user-defined function pointer, typically set to NULL by default;
Returns	0 indicates that the model inference runs normally; -1 indicates a failure in calling the model inference;

The example code is as follows:

```
// Predefined text values for the prompt before and after  
#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"  
  
// Define the input prompt and complete the concatenation  
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;  
  
// Call the model inference interface  
rkllm_run(llmHandle, input_str.c_str(), NULL);
```

3.2.5 Interrupt Model Inference

During model inference, users can call the `rkllm_abort()` function to interrupt the inference process.

The specific function definition is as follows:

Table 3-9 Interface Specification for the rkllm_abort Function

Fuctionom	rkllm_ abort
Introduction	Used to interrupt the RKLLM model inference process.
Parameters	LLMHandle handle: the target handle registered during model initialization;
Returns	0 indicates successful interruption of the model; -1 indicates a failure to interrupt the model.

The example code is as follows:

```
// llmHandle is the the target handle registered
rkllm_abort(llmHandle);
```

3.2.6 Release Model

After completing all model inference calls, users need to call the rkllm_destroy() function to destroy the RKLLM model and release the CPU, GPU, and NPU computing resources allocated, for use by other processes or models. The specific function definition is as follows:

Table 3-10 Interface Specification for the rkllm_destory Function

Fuctionom	rkllm_ destroy
Introduction	Used to destroy the RKLLM model and release all computing resources.
Parameters	LLMHandle handle: the target handle registered during model initialization;
Returns	0 indicates successful destruction and release of the RKLLM model; -1 indicates a failure in releasing the model.

The example code is as follows:

```
// llmHandle is the the target handle registered
rkllm_destory(llmHandle);
```

3.3 Board-side Inference Example

The directory, rkllm-runtime/examples/rkllm_api_demo, is include the C++ project for the inference on the board-site, and which is synchronously updated with this documentation. Furthermore, compilation

scripts are provided for users to facilitate the compilation of the project and the completion of the board-side inference of the RKLLM model.

3.3.1 Complete Code of Example Project

The complete C++ code example for the inference call is provided below. It is identical to the main.cpp file in the toolchain's rkllm_api_demo/src directory. The code implements the entire process, including model initialization, inference, the handling of output with callback functions, and the release of model resources. Those wishing to implement custom functionalities may find it helpful to refer to the relevant code.

```
// 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 Instructions for Example Project

Under the directory of rkllm_api_demo, not only does it contain sample code for invoking the RKLLM model inference, but also includes compilation scripts build-android.sh and build-linux.sh. This section will provide a brief explanation of how to use the sample code, taking compiling executable files for Linux systems as an example using the build-linux.sh script:

Firstly, users need to prepare cross-compilation tools on their own, noting that the recommended version of the compilation tools in section 2.3 is 10.2 or above. Subsequently, before the compilation process, users should replace the path to the cross-compilation tools in build-linux.sh themselves:

```
# Set the path of the cross-compiler
GCC_COMPILER_PATH=~/.gcc-arm-10.2-2020.11-x86_64-aarch64-none-linux-
gnu/bin/aarch64-none-linux-gnu
```

Subsequently, users can use `build-linux.sh` to initiate the compilation process. Upon completion of the compilation, users will obtain the corresponding `llm_demo` program, which will be installed in the `build/build_linux_aarch64_Release/llm_demo` directory. Following this, the executable file, library folder, and the RKLLM model (previously converted and quantized using the RKLLM-Toolkit tool) should be pushed to the board-side.

```
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
```

After completing the above steps, users can enter the terminal interface of the board using `adb`, and navigate to the corresponding `/userdata/llm` directory. Then, the inference of RKLLM model on the board can be invoked using the following command:

```
adb shell
cd /userdata/llm
export LD_LIBRARY_PATH=./lib
taskset f0 ./llm_demo /path/to/your/rkllm/model
```

With the above operations, users can enter the example inference interface, interact with the board-side model for inference, and obtain real-time inference results from the RKLLM model.

3.4 Implementation of Board-side Server

After using RKLLM-Toolkit to convert the model and obtain the RKLLM model, users can deploy server-side services on Linux development boards. This involves setting up a server on a Linux device and exposing network interfaces to everyone in the local area network. Subsequently, the RKLLM model can be accessed by other users via the specified address, thus facilitating efficient and concise interaction. This section will introduce two different server deployment implementations.

1) RLM-Server-Flask based on Flask: Users can achieve API access between the client and server using request requests. In the provided RKLLM-Server-Flask example, the send-receive structure is specially set to be the same as the OpenAI-API interface, facilitating quick replacement for users on existing development bases.

2) RKLLM-Server-Gradio based on Gradio: By reference to the provided example, users can rapidly

construct a web server for visual interaction. Furthermore, the example illustrates the utilisation of the Gradio API interface, which enables users to undertake secondary development.

Both examples of server implementations mentioned above are located in the rkllm-runtime/examples/rkllm_server_demo directory. This directory contains specific code for both implementations, one-click deployment scripts, and API interface calling examples. Users can choose different examples for reference and further development. The directory structure is as follows:

```
rkllm-runtime/examples/rkllm_server_demo
├── rkllm_server                # Board-side Deployment Required Files
│   ├── lib                    # RKLLM Runtime
│   ├── fix_freq_rk3576.sh     # Script for RK3576 Fixed Frequency
│   ├── fix_freq_rk3588.sh     # Script for RK3588 Fixed Frequency
│   ├── flask_server.py        # RKLLM-Server-Flask Example
│   └── gradio_server.py       # RKLLM-Server-Gradio Example
├── build_rkllm_server_flask.sh # One-click Deployment Script -Flask
├── build_rkllm_server_gradio.sh # One-click Deployment Script -Gradio
├── chat_api_flask.py           # API Interface Example -Flask
├── chat_api_gradio.py          # API Interface Example -Gradio
└── Readme.md
```

3.4.1 Deployment Example of RKLLM-Server-Flask

In the deployment example of RKLLM-Server-Flask, the main focus is on using the Flask framework to set up the server-side. On the client-side, data is transmitted using the request-response structure to achieve API access. In both the server and client implementations of RKLLM-Server-Flask, special consideration is given to the calling method of the OpenAI-API. The example code ensures consistency by setting up data structures identical to those used in the OpenAI-API. This allows users to seamlessly migrate their services by simply replacing the access interface after deploying RKLLM-Server-Flask, leveraging their existing development foundations in OpenAI-API development.

According to the [OpenAI-API usage documentation](#), users send specific data structures to the server during the API call process. The main contents are as follows:

```
{
  "model": "No models available",
  "messages": [
    {
      "role": "system",
      "content": "You are a helpful assistant."
    },
    {
      "role": "user",
      "content": "Hello!"
    }
  ],
  "stream": false,
}
```

Among them, "model" and "stream" specify the specific model to be called and whether to initiate streaming inference transmission, while the "content" data in "messages" is the crucial user input.

As for the data returned by the server, the structure of the data output by the OpenAI-API varies depending on whether streaming inference transmission is selected. The data content returned under non-streaming inference settings is as follows:

```
{
  "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
  }
}
```

When streaming inference transmission is not selected, the most important part is the "messages" content under the "choices" section, which represents the inference results provided by the model.

In contrast, when streaming inference transmission is enabled, the server returns a Response object, which includes the output results of the model at different points in time during the streaming inference process. The data content at each moment is as follows:

```
{
  "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"
  }]
}
```

After receiving the data from streaming transmission, users need to focus on the "delta" data section within the "choices" part. Additionally, when "finish_reason" is empty (None), it indicates that the model is still in the inference state, and the data has not been fully generated yet. It's only when "finish_reason" returns "stop" that the streaming inference is considered finished.

In the provided deployment example code and API access examples for RKLLM-Server-Flask, you can see identical definitions of the transmission data structure, ensuring the generality of the deployed RKLLM-Server-Flask. This facilitates quick replacement for users who have previously used OpenAI-API. In the subsequent sections of this chapter, we will separately introduce the one-click deployment script for the server, important settings for server deployment implementation, and the script design for client-side API access.

3.4.1.1 Server-side: One-click Deployment Script for RKLLM-Server-Flask

The one-click deployment script for RKLLM-Server-Flask is named `build_rkllm_server_flask.sh` and is located in the `rkllm_server_demo` directory. This script helps users quickly set up the RKLLM-Server-Flask server on a Linux development board. Before using this script, users should note the following:

- 1) Ensure that the development board is connected to the network via an Ethernet cable. Use the "ifconfig" command in the adb shell to determine the specific IP address of the development board. The RKLLM-Server-Flask will then set up the server with this IP address within the local area network and accept client access.

- 2) Users should have successfully converted the RKLLM model beforehand. Before executing the

one-click deployment script, ensure that the RKLLM model has been pushed to the Linux board.

Users can directly invoke the `build_rkllm_server_flask.sh` script from their PC (not the development board) using the following command to quickly deploy the RKLLM-Server-Flask server on the Linux development board:

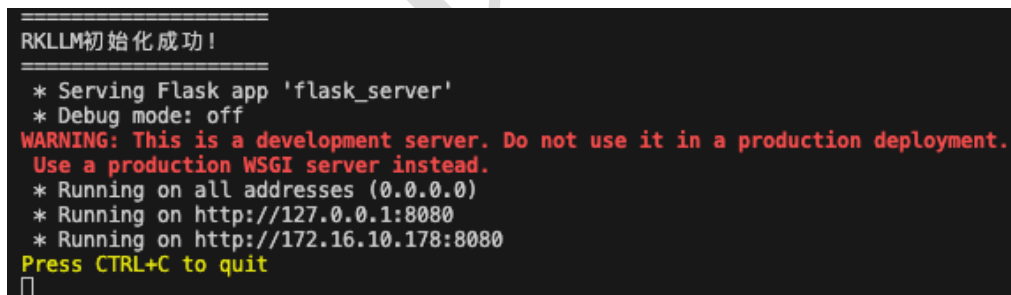
```
# build_rkllm_server_flask.sh [target platform: rk3588/rk3576]
[working directory] [path of the RKLLM model on the board]

build_rkllm_server_flask.sh rk3588 /path/to/workshop /path/to/model
```

After executing the above command, the one-click deployment script will perform the following steps:

- 1) Check the Linux environment on the board.
- 2) Automatically install the Flask library if not already installed.
- 3) Push the necessary files under `rkllm_server_demo/rkllm_server` to the board.
- 4) Index the RKLLM model in the preset working directory for RKLLM-Server-Flask.

Once you see the message "RKLLM initialization successful!" in the terminal, it indicates that the RKLLM-Server-Flask example has been successfully launched.



```
=====  
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  
□
```

Figure 3-1 Successful deployment of RKLLM-Server-Flask in terminal

By referring to the specific code logic in `build_rkllm_server_flask.sh`, users can understand the detailed deployment process of the RKLLM-Server-Flask example. This enables users to customize the deployment implementation of their server more flexibly. It is important to emphasize that in step 3 of the one-click deployment script, the script automatically synchronizes the current version of RKLLM Runtime to `rkllm_server/lib/librkllmrt.so`. This ensures that `flask_server.py` calls the current version of `librkllmrt.so` during runtime.

3.4.1.2 Server-side: Introductions for RKLLM-Server-Flask Example

In this section, we will outline and introduce the implementation approach of the deployment example for RKLLM-Server-Flask, helping users understand the construction logic of the example code for potential secondary development.

The deployment example of RKLLM-Server-Flask primarily relies on the Flask library to achieve the basic implementation of the server. Additionally, for RKLLM model inference, the ctypes library in Python is chosen to directly call the RKLLM Runtime library.

In the overall implementation of `rklm_server/flask_server.py`, in order to call `librkllmrt.so` via ctypes, it's necessary to define relevant structures in Python based on the header file `rklm.h` corresponding to `librkllmrt.so` beforehand. After the Flask server receives the struct data sent by users, it calls relevant functions to perform inference with the RKLLM model. This section will provide explanations and introductions to the main code segments.

- 1) Set the library path and link the RKLLM Runtime library `librkllmrt.so` through ctypes.

```
llm_lib = ctypes.CDLL('lib/librkllmrt.so')
```

- 2) Define the structures in the `librkllmrt.so` library and establish the linkage between Python and the C++ function library.

```
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) Define callback functions and establish function linkage with the dynamic library. However, in this instance, the callback function is required to make calls to three global variables: `global_text`, `global_state`, and `split_byte_data`. These variables are of critical importance for ensuring synchronisation between the output of RKLLM inference and the Flask framework for streaming output. Among these variables, `split_byte_data` has been designed to address the potential issue of encoding interruptions that may arise in the text returned by the callback function.

```
# Define global variables to store the output of the callback function
global_text = []
global_state = -1
split_byte_data = bytes(b"") # Used to store segmented byte data

def callback(result, userdata, state):
    global global_text, global_state, split_byte_data
    if state == 0:
        # Save the output token text and the RKLLM runtime state
        global_state = state
        # Monitor whether the current byte data is complete
        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:
        global_state = state
        print("\n")
        sys.stdout.flush()
    else:
        print("run error")

# Connecting Callback Functions between Python and C++
callback_type = ctypes.CFUNCTYPE(None, ctypes.POINTER(RKLLMResult),
ctypes.c_void_p, ctypes.c_int)
c_callback = callback_type(callback)
```

4) Define the RKLLM class in Python, which includes initialization, inference, and release operations for the RKLLM model in the dynamic library. In this deployment example, the RKLLM model is directly defined. Users can modify the specific parameters in the initialization (`init`) part as needed. Additionally,

when implementing a custom server, users can choose a more suitable definition method for replacement.

Just ensure that Python provides matching definitions when referencing header files during function definitions.

```
# Define RKLLM_Handle_t and userdata
RKLLM_Handle_t = ctypes.c_void_p
userdata = ctypes.c_void_p(None)

# Set up prompt text
PROMPT_TEXT_PREFIX = "<|im_start|>system You are a helpful assistant.
<|im_end|> <|im_start|>user"
PROMPT_TEXT_POSTFIX = "<|im_end|><|im_start|>assistant"

# Define the RKLLM class in the dynamic library
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) Initialize the RKLLM model by invoking the defined RKLLM class, and utilize this RKLLM class for inference and release of the model.

```
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) Using Flask, build RKLLM-Server-Flask to parse and process the input struct sent back from the client. Utilize the previously designed global variables `global_text`, `global_state`, and `split_byte_data` to link the inference output of the RKLLM model to the Flask framework. Finally, wrap the data in a specific struct and send it to the client. This part includes two designs: non-streaming inference and streaming inference, to respond to different user access requirements.

```
# Create function to receive data by users using the request module
@app.route('/rkllm_chat', methods=['POST'])
def receive_message():
    # Link global variables to retrieve output information
    global global_text, global_state
    global is_blocking

    # Return a specific response if the server is in a blocking state
    if is_blocking or global_state==0:
        return jsonify({'status': 'error', 'message': 'RKLLM_Server is
busy! Maybe you can try again later.'}), 503

    lock.acquire() # Lock
    try:
        # Set the server to a blocking state
        is_blocking = True

        # Retrieve JSON data from a POST request
        data = request.json
        if data and 'messages' in data:
            # Reset global variables
            global_text = []
            global_state = -1
```



```
# Define the structure of the returned struct
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:
    # Process the received data
    messages = data['messages']
    print("Received messages:", messages)
    for index, message in enumerate(messages):
        input_prompt = message['content']
        rkllm_output = ""
        # Create a thread for model inference
        model_thread =
threading.Thread(target=rkllm_model.run, args=(input_prompt,))
        model_thread.start()

        # periodically check the model's inference thread
        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:
    messages = data['messages']
    print("Received messages:", messages)
    for index, message in enumerate(messages):
        input_prompt = message['content']
        rkllm_output = ""

        def generate():
            model_thread =
threading.Thread(target=rkllm_model.run, args=(input_prompt,))
            model_thread.start()
```

```

        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:
        lock.release()
        # Set the server status to non-blocking
        is_blocking = False

# Start the Flask application.
app.run(host='0.0.0.0', port=8080, threaded=True, debug=False)

```

The above descriptions of each module constitute the main body of rkllm_server/flask_server.py, completing the deployment implementation of the RKLLM-Server-Flask example. Users can modify the initialization definition of the RKLLM model to implement different custom models. Additionally, users can also refer to the above RKLLM-Server-Flask deployment example to implement custom server deployments.

3.4.1.3 Client-side: API Access Example

In rkllm_server_demo/chat_api_flask.py, an API access example for the aforementioned RKLLM-Server-Flask server is demonstrated. When developing custom functionalities, users only need to refer to this API access example and utilize corresponding send-receive structures for data wrapping and parsing. Since the send-receive data structures in this example follow the design of the OpenAI-API, users who have previously developed using the OpenAI-API only need to replace the corresponding network

interfaces. This section will provide explanations for the important code segments:

1) Define the network address of RKLLM-Server-Flask. Users need to set the target address based on the specific IP of the Linux development board, the port number set by the Flask framework, and the function name for access.

```
server_url = 'http://172.16.10.166:8080/rkllm_chat'
```

2) Define the form of API access, with options for non-streaming transmission and streaming transmission, defaulting to non-streaming transmission.

```
is_streaming = True
```

3) Define the session object, using requests.Session() to configure the communication process between the API access interface and the server. Users can customize modifications according to their actual development needs.

```
session = requests.Session()
session.keep_alive = False # Close the connection pool
adapter = requests.adapters.HTTPAdapter(max_retries=5)
session.mount('https://', adapter)
session.mount('http://', adapter)
```

4) Define the structure used to wrap the data sent during API calls. User access to RKLLM-Server-Flask will be encapsulated within this structure.

```
# Prepare the data to be sent
# model: the model defined by the user when setting up RKLLM-Server,
# which has no effect here
# messages: the questions input by the user; supports adding multiple
# questions to messages
# stream: whether to enable streaming dialogue, same as the OpenAI
# interface
data = {
    "model": 'your_model_deploy_with_RKLLM_Server',
    "messages": [{"role": "user", "content": user_message}],
    "stream": is_streaming
}
```

5) Send a request to the RKLLM-Server-Flask server and retrieve the returned data.

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

6) Define the parsing method for the returned data structure. Since RKLLM-Server-Flask follows the format of the returned struct from the OpenAI-API, parsing operations are required in actual usage.

```
# Parse the response
# Non-streaming transmission
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)

# Streaming transmission
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)
```

The overall process from steps 1 to 6 represents the API access method for the RKLLM-Server-Flask server. Users can develop custom functionalities based on the example code provided above. It is important for users to verify the specific address of the RKLLM-Server-Flask, namely the IP address, port number, and the function interface that the server accepts input from. Additionally, when encountering specific requirements for send-receive structures, users can customize the required data structures on both the server and client sides to ensure the implementation of custom functionalities.

3.4.2 Deployment Example of RKLLM-Server-Gradio

Gradio is a simple and easy-to-use Python library used for quickly building interactive interfaces for machine learning models. In this section, we will specifically introduce how to quickly deploy RKLLM-Server-Gradio on a Linux device using Gradio, and directly access the server within the local network to perform RKLLM model inference. The following Figure 3-2 shows an example of the web interface after successfully deploying RKLLM-Server-Gradio:

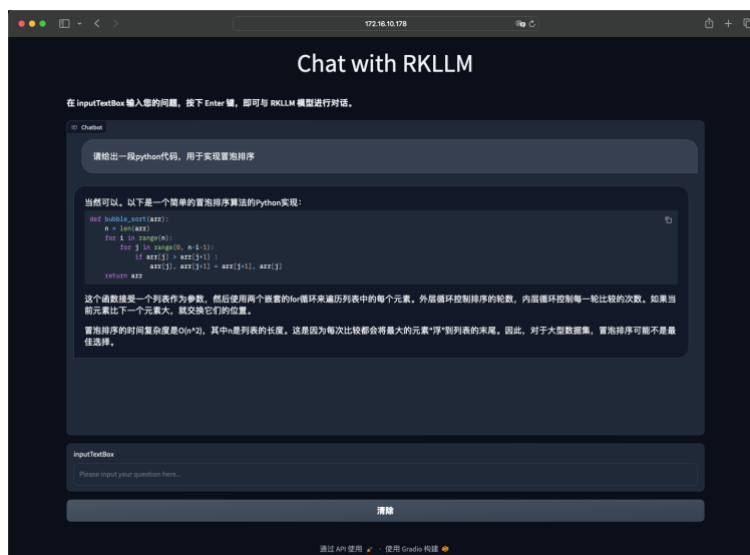


Figure 3-2 External access page for RKLLM-Server deployment example

In the current `rkllm_server_demo` directory, the specific implementation code for the RKLLM-Server-Gradio deployment example shown in Figure 3-2 is included. Users can also directly use the one-click deployment script `build_rkllm_server_gradio.sh` to quickly set up RKLLM-Server-Gradio. After successful deployment, users can choose to access the RKLLM model for inference either through the web interface or via API access.

3.4.2.1 Server-side: Introductions for RKLLM-Server-Gradio Example

The one-click deployment script `build_rkllm_server_gradio.sh` is designed to facilitate the quick setup of RKLLM-Server-Gradio on a Linux development board. Similar to the setup process for RKLLM-Server-Flask, users should pay attention to the following points before using the one-click deployment script:

- 1) Ensure that the development board is connected to the network via an Ethernet cable. Use the `ifconfig` command in the adb shell to query the specific IP address of the development board. RKLLM-Server-Gradio will be set up as a server within the local network using this IP address to accept client access.

- 2) Users need to complete the smooth conversion of the RKLLM model and have pushed the RKLLM model to the Linux board before executing the one-click deployment script.

Users can directly call the `build_rkllm_server_gradio.sh` script on the PC side (not on the development board) using the following command to quickly deploy the RKLLM-Server-Gradio example:

```
# build_rkllm_server_gradio.sh [target platform: rk3588/rk3576]
[working directory] [path of the RKLLM model on the board]

build_rkllm_server_gradio.sh rk3588 /path/to/workshop /path/to/model
```

After executing the above command, the one-click deployment script will perform the following steps:

- 1) Check the Linux environment on the board.
- 2) Automatically install the Gradio library if not already installed.
- 3) Push the necessary files under rkllm_server_demo/rkllm_server to the board.
- 4) Index the RKLLM model in the preset working directory for RKLLM-Server-Gradio.

Once you see the message "RKLLM initialization successful!" in the terminal, it indicates that the RKLLM-Server-Flask example has been successfully launched.

```
(base) ~:~/code/rkllm/examples/server$ ./build_rkllm_server.sh rk3588 /data/your_model.rkllm
===== pin3 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 0.268s)
NPU available frequencies:
300000000 400000000 500000000 600000000 700000000 800000000 900000000 1000000000
Fix NPU max frequency:
1000000000
CPU available frequencies:
400000 600000 816000 1008000 1200000 1416000 1608000 1800000
400000 600000 816000 1008000 1200000 1416000 1608000 1800000 2016000 2208000 2352000
400000 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!
```

Figure 3-3 Successful deployment of RKLLM-Server-Gradio in terminal

By referencing the specific code in build_rkllm_server_gradio.sh, users can understand the detailed deployment process of the RKLLM-Server-Gradio example. This can help users deploy custom servers more flexibly. It is important to emphasize that in step 3 of build_rkllm_server_gradio.sh, the one-click deployment script automatically synchronizes the current version of RKLLM Runtime to rkllm_server/lib/librkllmrt.so. This ensures that gradio_server.py indexes librkllmrt.so when running, and users need to pay attention to the invocation of librkllmrt.so when customizing the server.

3.4.2.2 Server-side: Introductions for RKLLM-Server-Gradio Example

The deployment implementation of RKLLM-Server-Gradio is essentially similar to RKLLM-Server-Flask. It also uses the ctypes library to directly call the RKLLM Runtime library to perform RKLLM model inference. However, RKLLM-Server-Gradio chooses to use the gradio library to build the server and

communicate with the client, providing a simple web-based service.

1) The design involves creating functions to handle user input and to perform streaming printing of output. These input and output functions will be directly called in the gradio interface design.

```
# Record the prompt input by the user
def get_user_input(user_message, history):
    history = history + [[user_message, None]]
    return "", history

# Get the output of the RKLLM model and perform streaming printing
def get_RKLLM_output(history):
    # Link global variables to retrieve output information
    global global_text, global_state
    global_text = []
    global_state = -1

    # Create a thread for model inference
    model_thread = threading.Thread(target=rkllm_model.run,
    args=(history[-1][0],))
    model_thread.start()

    # history[-1][1] represents the current output dialogue
    history[-1][1] = ""

    # periodically check the model's inference thread
    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)
            yield history

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

2) Invoke the gradio library to design the basic layout of the web interface, and link the previously defined input and output processing functions to achieve fast and simple user interaction functionality.

```
with gr.Blocks(title="Chat with RKLLM") as chatRKLLM:
    gr.Markdown("<div align='center'><font size='70'> Chat with RKLLM  
</font></div>")
    gr.Markdown("### Enter your question in the inputTextBox, then  
press Enter to start a conversation with the RKLLM model.")
    # Create a Chatbot component to display the conversation history
    rkllmServer = gr.Chatbot(height=600)
    # Create a Textbox component for users to input messages
    msg = gr.Textbox(placeholder="Please input your question here...",
    label="inputTextBox")
    # Create a Button component to clear the chat history
    clear = gr.Button("Clear")
```

```
# Submit the input to get_user_input function and update history.
# Then, call the get_RKLLM_output function to update chat history
# Use the queue=False parameter to ensure executed immediately.
msg.submit(get_user_input, [msg, rkllmServer], [msg, rkllmServer],
queue=False).then(get_RKLLM_output, rkllmServer, rkllmServer)
# execute a no-operation (lambda: None) and immediately clear
clear.click(lambda: None, None, rkllmServer, queue=False)

# Enable the event queue system
chatRKLLM.queue()
# Start the Gradio application
chatRKLLM.launch()
```

The explanations of each module above constitute the main code body of `rkllm_server/gradio_server.py`, completing the deployment of the RKLLM-Server-Gradio example. Users can modify the initialization definition of the RKLLM model to implement different custom models. Additionally, users can refer to the deployment example of RKLLM-Server-Gradio provided above to implement custom server deployments.

3.4.2.3 Client: RKLLM-Server-Gradio Usage Instructions

After successfully deploying the RKLLM-Server-Gradio on a Linux development board, users can access it via two methods: "Interface Access" and "API Access".

1) Interface Access: Upon successfully starting the RKLLM-Server-Gradio with the one-click deployment script, users can directly access the RKLLM model for quick interaction by opening any web browser on a computer within the current local area network and navigating to "board_IP:8080" (e.g., "172.16.10.178:8080" as shown in Figure 3-2). Gradio automatically integrates Markdown, HTML, and other syntaxes, adapting to the format of the RKLLM model's output results, such as code snippets and Markdown text. Additionally, during the setup of RKLLM-Server, an access queue is initiated. When multiple users interact with the RKLLM-Server simultaneously, the inputs are processed and returned in the order they were submitted. It's important to note that when a user's interaction with the RKLLM-Server is in the inference state (i.e., the dialogue box is highlighted), the server will not accept the user's next input until the current inference is completed.

2) API Access: In the `rkllm_server_demo` directory, `chat_api_gradio.py` is provided. After installing `gradio_client` on the PC (using the command: `pip install gradio_client`), users can interact with the RKLLM-

Server solely through the API interface without relying on the graphical interface, as shown in Figure 3-4.

Before using chat_api_gradio.py, it's important to modify the IP address in the code to match the current IP address of the development board, as shown in the following code.

```
from gradio_client import Client
client = Client("http://172.16.10.169:8080/")
```



```

(base) /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：

```

Figure 3-4 Access the RKLLM-Server-Gradio via API calls in terminal

Users can choose between the two client invocation methods based on their specific needs. For instance, when providing interactive services within a local area network, it's recommended to use the interface access method. On the other hand, if customizing access behaviors to RKLLM-Server-Gradio is required, it's advisable to use API Access for further development.

Lastly, it's important to note that in the implementation of RKLLM-Server-Gradio, there isn't a definition of data structures for sending and receiving data similar to OpenAI-API. Therefore, this deployment implementation is not compatible with the OpenAI-API interface. When conducting further development, users should refer to the specific function implementation in chat_api_gradio.py. If compatibility with the OpenAI-API interface is required, please refer to the implementation of RKLLM-Server-Flask.

4 Reference

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

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

Rockchip