Introduction	Used for simulate model inference on the PC.
Parameters	***messages:*** The text input, which needs to include the appropriate prompts. ***args:*** Inference configuration parameters, such as sampling parameters like top_k.
Returns	The logits values inferred by the model.

The example code is as follows:

```
args ={
    "max_length":128,
    "top_k":1,
    "temperature":0.8,
    "do_sample":True,
    "repetition_penalty":1.1
    }

mesg = "Human: How’s the weather today?\nAssistant:"
print(llm.chat_model(mesg, args))
```

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 that 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<reuslt->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 3-8 Explanation of LLMCallState Status Flags

Definition	LLMCallState
Introduction	Used to indicate the current running state of RKLLM.
Enumeration Values	***LLM_RUN_NORMAL:*** indicates that the RKLLM model is currently inferencing; ***LLM_RUN_FINISH:*** indicates that the RKLLM model has completed inference; ***LLM_RUN_WAITING:*** indicates that the currently decoded character from RKLLM is not a complete UTF-8 encoding and needs to be concatenated with the next decoding result; ***LLM_RUN_ERROR:*** indicates that an error has occurred during inference;

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 3-9 Explanation of RKLLMResult Structure

Definition	RKLLMResult
Introduction	Used to return the current inference-generated result.
Struct Fields	***const char* text:*** indicates the text content generated by the current inference; ***int32_t token_id:*** indicates the token_id generated by the current inference; ***RKLLMResultLogits logits:*** Represents the logits information generated during the current inference, only returned when RKLLMInferMode is set to RKLLM_INFER_GET_LOGITS;

3) RKLLMResultLogits is a return value structure, and its specific definition is as follows:

Table 3-9 Explanation of RKLLMResultLogitsStructure

Definition	RKLLMResultLogits
Introduction	Used to return the current inference-generated logits.
Struct Fields	***const float* logits:*** indicates the logits generated by the current inference; ***int vocab_size:***indicates the token_id generated by the current inference; ***int num_tokens:*** Indicates the number of tokens returned; users can calculate the size of the logits based on vocab_size and num_tokens;

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 RKLLMParam

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

Table 3-10 Explanation of RKLLMParam Structure

Definition	RKLLMParam
Introduction	Used to define the detailed parameters of the RKLLM model.

Struct Fields	***const char* model_path:*** the path to the RKLLM model file; ***int32_t num_npu_core:*** the number of NPU cores used during model inference; The "rk3576" platform has configurable range [1, 2]; while the "rk3588" is [1, 3]; ***bool use_gpu:*** whether to use GPU for prefill acceleration, default option is false; ***int32_t max_context_len:*** the maximum context length during inference; ***int32_t max_new_tokens:*** the maximum number of generated tokens in inferencing; ***int32_t top_k:*** 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; ***float top_p:*** 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; ***float temperature:*** 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； ***float repeat_penalty:*** controls the occurrence of token sequence repetitions in the generated text, helping to prevent the model from generating repetitive or 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; ***float frequency_penalty:*** 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; ***int32_t mirostat:*** 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; ***float mirostat_tau:*** 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; ***float mirostat_eta:*** 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; ***bool skip_special_token:*** whether to skip special tokens and not output them, such as the end-of-sequence token <EOS>. ***bool is_async:*** whether to use asynchronous mode. ***const char img_start*:*** option to set the start marker for multimodal input image encoding, which needs to be configured in multimodal input mode. ***const char img_end*:*** option to set the end marker for multimodal input image encoding, which needs to be configured in multimodal input mode. ***const char img_content*:*** option to set the content marker for multimodal input image encoding, which needs to be configured in multimodal input mode. ***RKLLMExtendParam extend_param:*** the special parameters for controlling inference. ***n_keep:*** The number of cache entries to retain at the beginning when clearing the KV cache. In multi-turn conversations, the n_keep value must be at least as long as the system_prompt length.

Table 3-11 Explanation of RKLLMExtendParam Structure

Definition	RKLLMExtendParam
Introduction	The special parameters for controlling inference.
Struct Fields	***int32_t base_domain_id:*** controls from which domain the RKLLM model starts initialization, default is 0. ***int8_t embed_flash:*** Controls whether to store the model's vocabulary in flash memory to save memory. Set to 0 to disable and 1 to enable. ***int8_t enabled_cpus_num:*** Sets the number of CPUs to use for inference. The range varies depending on the chip model. For RK3588/3576, the range is 1-8; for RK3562, the range is 1-4, with the default set to 4.

	uint32_t enabled_cpus_mask: Uses a binary mask to configure which specific CPU cores are used for inference. In rkllm.h, macros are predefined to represent CPU numbers, and you can configure them using the format CPU4 | CPU5 | CPU6 | CPU7.

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

3.2.3 Definite Input Struct

To accommodate different input data types, the RKLLMInput input struct is defined, which currently accepts four types of input: text, image and text, token IDs, and encoded vectors. The specific definition is as follows:

Table 3-12 Explanation of RKLLMInput Structure

Definition	RKLLMInput
Introduction	Used to receive different forms of input data.
Struct Fields	***RKLLMInputType input_type:*** Input mode; ***union:*** Used to store different input data types, specifically including the following forms: ***const char prompt_input*:*** Text prompt input, used to pass natural language text; ***RKLLMEmbedInput embed_input:*** Embedding vector input, representing processed feature vectors; ***RKLLMTokenInput token_input:*** Token input, used to pass the tokenized token sequence; ***RKLLMMultiModelInput multimodal_input:*** Multimodal input, which can pass

	multimodal data, such as combined input of images and text;

Table 3-13 Explanation of RKLLMInputType Structure

Definition	RKLLMInputType
Introduction	Used to represent the type of input data.
Enumeration Values	***RKLLM_INPUT_PROMPT:*** Indicates that the input data is plain text. ***RKLLM_INPUT_TOKEN:*** Indicates that the input data is token IDs. ***RKLLM_INPUT_EMBED:*** Indicates that the input data is encoded vectors. ***RKLLM_INPUT_MULTIMODAL:*** Indicates that the input data consists of images and text.

When the input data is plain text, it can be directly input using input_data. When the input data consists of token IDs, encoded vectors, or images and text, the RKLLMInput must be used in conjunction with the RKLLMTokenInput, RKLLMEmbedInput, and RKLLMMultiModelInput structures. The specific introduction is as follows:

1) RKLLMTokenInput is the input struct that receives token IDs. The specific definition is as follows:

Table 3-14 Explanation of RKLLMTokenInput Structure

Definition	RKLLMTokenInput
Introduction	Used to receive token_id data.
Struct Fields	***int32_t input_ids*:*** Memory pointer for the input token IDs. ***size_t n_tokens:*** The number of tokens in the input data.

2) RKLLMEmbedInput is the input struct that receives encoded vectors. The specific definition is as follows:

Table 3-15 Explanation of RKLLMEmbedInput Structure

Definition	RKLLMEmbedInput
Introduction	Used to receive embedding data.

Struct Fields	***float embed*:*** Memory pointer for the input token embeddings. ***size_t n_tokens:*** The number of tokens in the input data.

3) RKLLMMultiModelInput is the input struct that receives images and text. The specific definition is as follows:

Table 3-16 Explanation of RKLLMMultiModelInput Structure

Definition	RKLLMMultiModelInput
Introduction	Used to receive images and text data.
Struct Fields	***char prompt*:*** Memory pointer for the input text. ***float image_embed*:*** Memory pointer for the input image embeddings. ***size_t n_image_tokens:*** The number of tokens for the input image embeddings.

Here is an example of pure text input code:

```
#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"

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;

RKLLMInput rkllm_input;
rkllm_input.input_data = (char*)input_str.c_str();
rkllm_input.input_type = RKLLM_INPUT_PROMPT;

RKLLMInferParam rkllm_infer_params;
memset(&rkllm_infer_params, 0, sizeof(RKLLMInferParam));
rkllm_infer_params.mode = RKLLM_INFER_GENERATE;
```

An example code for image and text multimodal input is as follows. Note that the prompt for multimodal input needs to include the <image> placeholder to indicate where the image encoding should be inserted:

```
#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"

RKLLMInput rkllm_input;
string input_str = "<image>Please describe the image shortly.";
input_str = PROMPT_TEXT_PREFIX + input_str + PROMPT_TEXT_POSTFIX;
rkllm_input.multimodal_input.prompt = (char*)input_str.c_str();

rkllm_input.multimodal_input.n_image_tokens = 256;
int rkllm_input_len = multimodal_input.n_image_tokens * 3072;
rkllm_input.multimodal_input.image_embed = (float
*)malloc(rkllm_input_len * sizeof(float));
FILE *file;
file = fopen("models/image_embed.bin", "rb");
fread(rkllm_input.multimodal_input.image_embed, sizeof(float),
rkllm_input_len, file);
fclose(file);

rkllm_input.input_type = RKLLM_INPUT_MULTIMODAL;
RKLLMInferParam rkllm_infer_params;
memset(&rkllm_infer_params, 0, sizeof(RKLLMInferParam));
rkllm_infer_params.mode = RKLLM_INFER_GENERATE;
```

RKLLM supports different inference modes and defines the RKLLMInferParam structure. It currently supports joint inference with preloaded LoRA models during the inference process, or saving a Prompt Cache for subsequent inference acceleration. The specific definition is as follows:

Table 3-17 Explanation of RKLLMInferParam Structure

Definition	RKLLMInferParam
Introduction	Used to define different inference modes.
Struct Fields	***RKLLMInferMode mode:*** Inference mode, supporting RKLLM_INFER_GENERATE normal inference mode and RKLLM_INFER_GET_LOGITS additional logits retrieval inference mode. ***RKLLMLoraParam lora_params*:*** Parameter configuration for the LoRA used during inference, used to select which LoRA to infer when multiple LoRAs are loaded. Set to NULL if LoRA is not needed. ***RKLLMPromptCacheParam prompt_cache_params*:*** Parameter configuration for using the Prompt Cache during inference. Set to NULL if Prompt Cache generation is not needed.

	keep_history: Indicates whether to retain the historical context during inference. Set to 1 for multi-turn conversations.

Table 3-18 Explanation of RKLLMLoraParam Structure

Definition	RKLLMLoraParam
Introduction	Used to define the parameters for using LoRA during inference.
Struct Fields	***const char lora_adapter_name*:*** The name of the LoRA used during inference.

Table 3-19 Explanation of RKLLMPromptCacheParam Structure

Definition	RKLLMPromptCacheParam
Introduction	Used to define the parameters for using Prompt Cache during inference.
Struct Fields	***int save_prompt_cache:*** Indicates whether to save the Prompt Cache during inference. 1 means it is required, and 0 means it is not. ***const char prompt_cache_path*:*** Path to save the Prompt Cache. If not set, it defaults to "./prompt_cache.bin".

Here is an example of using RKLLMPromptCacheParam for inference:

```
// 1. Initialize and set LoRA parameters (if needed)
RKLLMLoraParam lora_params;
// Specify the LoRA model name
lora_params.lora_adapter_name = "test";
// 2. Initialize and Set Prompt Cache Parameters(if needed)
RKLLMPromptCacheParam prompt_cache_params;
// Enable saving Prompt Cache
prompt_cache_params.save_prompt_cache = true;
// Specify cache file path
prompt_cache_params.prompt_cache_path = "./prompt_cache.bin";
rkllm_infer_params.mode = RKLLM_INFER_GENERATE;
rkllm_infer_params.lora_params = &lora_params;
rkllm_infer_params.prompt_cache_params = &prompt_cache_params;
```

3.2.4 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 3-20 Interface Specification for the rkllm_init Function

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

The example code is as follows:

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

3.2.5 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 3-21 Interface Specification for the rkllm_run Function

Fuctiom	rkllm_run
Introduction	Used to performing result inference using the initialized RKLLM model.
Parameters	***LLMHandle handle:*** the target handle registered during model initialization. ***RKLLMInput rkllm_input*:*** Input data for model inference. For details, see section

	3.2.3 on input structure definition. ***RKLLMInferParam rkllm_infer_params*:*** Parameter passing during the model inference process. For details, see section 3.2.3 on input structure definition. ***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:

```
#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"

// Predefined text values for the prompt before and after
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;

// Define the input prompt and complete the concatenation
RKLLMInferParam rkllm_infer_params;
memset(&rkllm_infer_params, 0, sizeof(RKLLMInferParam));
rkllm_infer_params.mode = RKLLM_INFER_GENERATE;
// 1. Initialize and set LoRA parameters (if needed)
RKLLMLoraParam lora_params;
lora_params.lora_adapter_name = "test";
// 2. Initialize and Set Prompt Cache Parameters(if needed)
RKLLMPromptCacheParam prompt_cache_params;
prompt_cache_params.save_prompt_cache = true;
prompt_cache_params.prompt_cache_path = "./prompt_cache.bin";
rkllm_infer_params.mode = RKLLM_INFER_GENERATE;
// rkllm_infer_params.lora_params = &lora_params;
// rkllm_infer_params.prompt_cache_params = &prompt_cache_params;
rkllm_infer_params.lora_params = NULL;
rkllm_infer_params.prompt_cache_params = NULL;

RKLLMInput rkllm_input;
rkllm_input.input_type = RKLLM_INPUT_PROMPT;
rkllm_input.prompt_input = (char *)text.c_str();

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

3.2.6 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 3-22 Interface Specification for the rkllm_abort Function

Fuctiom	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);
```

3.2.7 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 3-23 Interface Specification for the rkllm_destory Function

Fuctiom	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_destroy(llmHandle);
```

3.2.8 Load LoRA Model

RKLLM supports running LoRA models simultaneously with the base model during inference. Before invoking the rkllm_run interface, you can load a LoRA model via the rkllm_load_lora interface. RKLLM allows loading multiple LoRA models; each call to rkllm_load_lora loads one LoRA model. The specific function definition is as follows:

Table 3-24 Interface Specification for the rkllm_load_lora Function

Fuctiom	rkllm_load_lora
Introduction	Used to load LoRA model for the base model.
Parameters	***LLMHandle handle:*** The target handle registered during model initialization. See section 3.2.4 on initializing the model. ***RKLLMLoraAdapter lora_adapter*:*** Parameter for loading the LoRA model.
Returns	**0** indicates the LoRA model was successfully loaded. **-1** indicates the model loading failed.

Table 3-25 Explanation of RKLLMLoraAdapter Structure

Definition	RKLLMLoraAdapter
Introduction	Used to configure parameters when loading a LoRA model.
Struct Fields	***const char* lora_adapter_path:*** The path to the LoRA model to be loaded. ***const char* lora_adapter_name:*** The name of the LoRA model to be loaded, defined by the user, used to select the specified LoRA during inference. ***float scale:*** The degree to which the LoRA model adjusts the base model parameters during inference.

Here is an example Code for Loading LoRA:

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

3.2.9 Load Prompt Cache

During the model inference process, the Prefill stage typically consumes a significant amount of computational resources and time, especially when the Prompt is long. To accelerate this process,

RKLLM supports loading Prompt Cache from files. By reusing cached content, you can significantly reduce the time spent in the Prefill stage, thereby improving overall inference efficiency.

Before invoking the rkllm_run interface for inference, ensure that the prompt_cache_params parameters are correctly configured. This step allows the model to generate the corresponding Prompt Cache file after inference. When running inference for the first time, the system will automatically generate a Prompt Cache file. This file contains the intermediate results required for the Prefill stage, which can be reused in subsequent tasks. In subsequent inference tasks, you can load previously generated Prompt Cache files by calling the rkllm_load_prompt_cache interface.

The specific function definitions are as follows:

Table 3-26 Interface Specification for the rkllm_load_prompt_cache Function

Fuctiom	rkllm_load_prompt_cache
Introduction	Used to load Prompt Cache file.
Parameters	***LLMHandle handle:*** The target handle registered during model initialization (refer to section 3.2.4 Initialization Model). ***const char* prompt_cache_path:*** The path to the Prompt Cache file to be loaded.
Returns	**0** indicates the Prompt Cache file was successfully loaded. **-1** indicates loading failed.

Table 3-27 Interface Specification for the rkllm_release_prompt_cache Function

Fuctiom	rkllm_release_prompt_cache
Introduction	Used to release the Prompt Cache.
Parameters	***LLMHandle handle:*** The target handle registered during model initialization (refer to section 3.2.4 Initialization Model).
Returns	**0** indicates that the Prompt Cache model was successfully released. -1 indicates that the model release failed.

Note:

RKLLM will detect the parts of the input that are the same as those in the prompt_cache from the beginning. If your input format is fixed as PROMPT_PREFIX + text + PROMPT_POSTFIX, you can generate the Prompt Cache for just the PROMPT_PREFIX part. After loading, you can reuse this part of the result in subsequent inferences.

RKLLM supports generating multiple Prompt Cache files. When different Prompt Cache files are needed, you can simply load the corresponding file. If you need to switch to another Prompt Cache file or no longer need the loaded Prompt Cache, please explicitly call the rkllm_release_prompt_cache interface to release it.

Here is an example Code for Loading Prompt Cache:

```
// Initialize and set the Prompt Cache parameters, then call the run
interface to generate the Prompt Cache file.

RKLLMPromptCacheParam prompt_cache_params;
// Whether to save the prompt cache
prompt_cache_params.save_prompt_cache = true;
// If you need to save the prompt cache, specify the absolute path of
the cache file.
prompt_cache_params.prompt_cache_path = "/data/prompt_cache.bin";
rkllm_infer_params.prompt_cache_params = &prompt_cache_params;

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

// Load the prompt cache file to reduce prefill time.
rkllm_load_prompt_cache(llmHandle, "./prompt_cache.bin");
if (ret != 0) {
    printf("\nload Prompt Cache failed\n");
}

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

3.2.10 KV Cache Management

RKLLM supports manual clearing of the KV cache, which can be used for both single-turn and multi-turn dialogues. When invoking the cache clearing function, if keep_system_prompt is set to 1, the system prompt (if present) will be retained; otherwise, the entire cache will be cleared.

The function definition is as follows:

Table 3-28 Interface Specification for the rkllm_clear_kv_cache Function

Fuctiom	rkllm_clear_kv_cache
Introduction	Used to clear the KV cache.
Parameters	***LLMHandle handle:*** The target handle registered during model initialization (refer to section 3.2.4 Initialization Model). ***keep_system_prompt:*** Whether to retain the system prompt (1 to retain, 0 to clear).
Returns	0: Indicates successful clearing of the KV cache. -1: Indicates failure to clear the KV cache.

3.2.11 Chat Template Settings

During model conversion, RKLLM automatically parses the chat_template field from the Hugging Face model's tokenizer_config.json file. In this field, system_prompt serves as the system prompt to guide model behavior, prompt_prefix acts as the prefix before user input, and prompt_postfix acts as the suffix after user input. During inference, RKLLM automatically applies the parsed prompt template. If modifications are needed, you can use the following function to reconfigure these settings.

The specific function definition is as follows:

Table 3-29 Interface Specification for the rkllm_set_chat_template Function

Fuctiom	rkllm_set_chat_template
Introduction	Used to set the prompt template.
Parameters	***LLMHandle handle:*** The target handle registered during model initialization (refer to section 3.2.4 Initialization Model). ***system_prompt:*** The system prompt guiding the model behavior. ***prompt_prefix:*** The prefix before user input. ***prompt_postfix:*** The suffix after user input.

Returns	0: Indicates the success of the chat template. -1: Indicates failure to set the chat template.

3.3 Board-side Inference Example

The directory, examples/DeepSeek-R1-Distill-Qwen-1.5B_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 inference calls is located in the example/DeepSeek-R1-Distill-Qwen-1.5B_Demo/deploy/src directory of the toolchain. The llm_demo.cpp file serves as an example of large language model inference. These examples include the full process, including model initialization, inference, handling outputs with callback functions, and releasing model resources. Users can refer to the relevant code to implement custom functionality.

3.3.2 Instructions for Example Project

Under the directory of examples/DeepSeek-R1-Distill-Qwen-1.5B_Demo/deploy, 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 installed in the install/demo_linux_aarch64/llm_demo directory 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 install/demo_Linux_aarch64  /data
adb push /PC/path/to/your/rkllm/model /data/demo_Linux_aarch64
```

After completing the above steps, users can enter the terminal interface of the board using adb, and navigate to the corresponding /data/demo_linux_aarch64 directory. Then, the inference of RKLLM model on the board can be invoked using the following command:

```
adb shell
cd /data/demo_linux_aarch64
export LD_LIBRARY_PATH=./lib
./llm_demo /path/to/your/rkllm/model 1024 2048
```

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.

3.3.3 Monitor inference performance and Log Viewing

To monitor the inference performance of RKLLM on the board like the above figure, you can use the command:

```
export RKLLM_LOG_LEVEL=1
```

```
I rkllm: ------------------------------------------------------------------------------------
I rkllm:  Stage         Total Time (ms)  Tokens    Time per Token (ms)      Tokens per Second
I rkllm: ------------------------------------------------------------------------------------
I rkllm:  Init          1430.74          /         /                        /
I rkllm:  Prefill       98.41            10        9.84                     101.61
I rkllm:  Generate      641.28           9         71.25                    14.03
I rkllm: ------------------------------------------------------------------------------------
I rkllm:  Memory Usage (GB)
I rkllm:  2.04
I rkllm: ------------------------------------------------------------------------------------
```

Figure 3-1 RKLLM inference performance logs on the hardware platform

This will display the number of tokens processed and the inference time for both the Prefill and Generate stages after each inference, as shown in the figure below. This information will help you evaluate the performance by providing detailed logging of how long each stage of the inference process takes.If you need to view more detailed logs, such as the tokens after encoding the prompt, you can use the following command:

```
export RKLLM_LOG_LEVEL=2
```

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 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:

```
examples/rkllm_server_demo
├———rkllm_server          # Board-side Deployment Required Files
│    └——— lib                    # RKLLM Runtime
│    └——— 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 on a development board) to quickly deploy the RKLLM-Server-Flask server on a Linux development board.

The specific usage of the one-click deployment script build_rkllm_server_flask.sh is as follows:

```
./build_rkllm_server_flask.sh
   --workshop [RKLLM-Server Working Path]
   --model_path [Absolute Path of Converted RKLLM Model on Board] --
platform [Target Platform: rk3588/rk3576]
   [--lora_model_path [Lora Model Path]]
   [--prompt_cache_path [Prompt Cache File Path]]
```

The `workshop` parameter specifies the subsequent working directory of RKLLM-Server-Flask on the board. The `model_path` parameter indicates the absolute path of the RKLLM model on the board, which was converted using RKLLM-Toolkit, and RKLLM-Server-Flask will read the model from this path during operation. The `platform` parameter specifies the current platform type, either rk3588 or rk3576. The `lora_model_path` and `prompt_cache_path` parameters are optional and can be used to

specify file paths when the user needs to load a Lora model or use the prompt feature.

The following is a simple example of how to use the one-click deployment script `build_rkllm_server_flask.sh`:

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

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 Model has been initialized successfully!" in the terminal, it indicates that the RKLLM-Server-Flask example has been successfully launched.

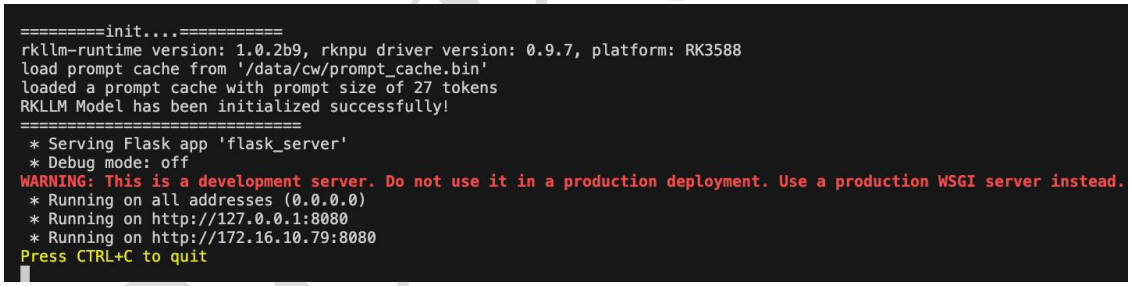

Figure 3-2 Successful deployment of RKLLM-Server-Flask in terminal

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 rkllm_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 rkllm.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. The specific code implementation of flask_server.py mainly consists of the following steps:

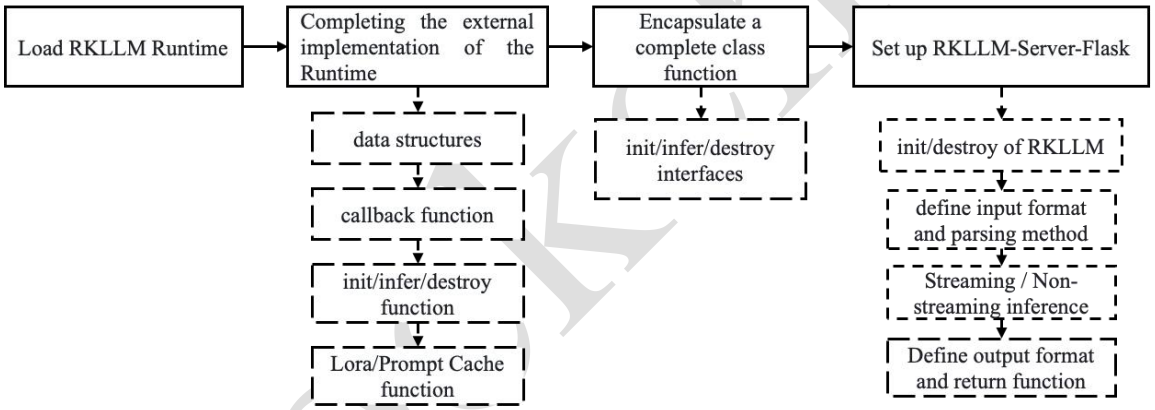

Figure 3-3 Overview of the RKLLM-Server-Flask Deployment Implementation Process

1) Load RKLLM Runtime: Set the path for the dynamic library and load the dynamic library librkllmrt.so using ctypes to achieve the analysis of the RKLLM Runtime.

2) Completing the external implementation of the Runtime: In the Python code, use ctypes to complete the external definitions of relevant implementations from the RKLLM Runtime header files, including definitions of data structures, callback functions, initialization functions, inference functions, destory functions, and loading functions for Lora/Prompt Cache.

3) Encapsulate a complete class function: Based on the various Runtime data types and function interfaces implemented in step 2), encapsulate complete class functions that integrate RKLLM's initialization, inference, destory, and other operations for easier subsequent calls.

4) Set up RKLLM-Server-Flask: Using the complete class functions encapsulated in step 3, build the Flask server, including loading the RKLLM model based on user-specified parameters at Flask startup, defining the format of user input and the method for parsing the input, differentiating between streaming/non-streaming inference call methods, defining the return format for inference output and the specific implementation of callback functions, and handling RKLLM release.

The specific implementations of the above modules form the main body of the code in rkllm_server/flask_server.py, thereby completing the deployment of the RKLLM-Server-Flask example. Users can modify the initialization definitions for the RKLLM model to implement different custom models. Additionally, users can refer to the RKLLM-Server-Flask deployment example for implementing their own custom server deployment.

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 shows an example of the web interface after successfully deploying RKLLM-Server-Gradio:

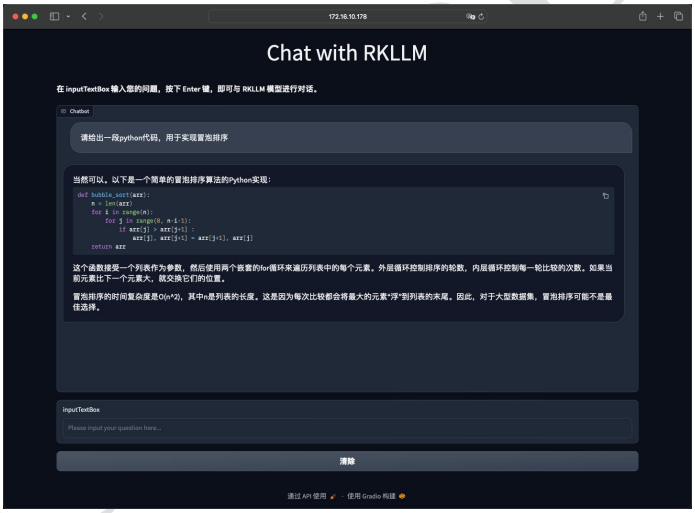

Figure 3-4 External access page for RKLLM-Server deployment example

In the current rkllm_server_demo directory, the specific implementation code for the RKLLM-Server-Gradio deployment example shown in Figure 3-3 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.

The usage of the one-click deployment script build_rkllm_server_gradio.sh is similar to that of build_rkllm_server_flask.sh:

```
./build_rkllm_server_gradio.sh
    --workshop [RKLLM-Server Working Path]
    --model_path [Absolute Path of Converted RKLLM Model on Board] --
platform [Target Platform: rk3588/rk3576]
    [--lora_model_path [Lora Model Path]]
    [--prompt_cache_path [Prompt Cache File Path]]
```

Similarly, the workshop parameter specifies the working directory for RKLLM-Server-Gradio on the device; the model_path parameter indicates the absolute path of the RKLLM model on the device, which was converted using RKLLM-Toolkit, and RKLLM-Server-Gradio will read the model from this path during operation; the platform parameter specifies the platform type being used, such as rk3588 or rk3576; lora_model_path and prompt_cache_path are optional parameters, allowing the user to specify the file paths for loading a Lora model or utilizing the Prompt feature if needed.

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

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 Model has been initialized successfully!" in the terminal, it indicates that the RKLLM-Server-Gradio example has been successfully launched.

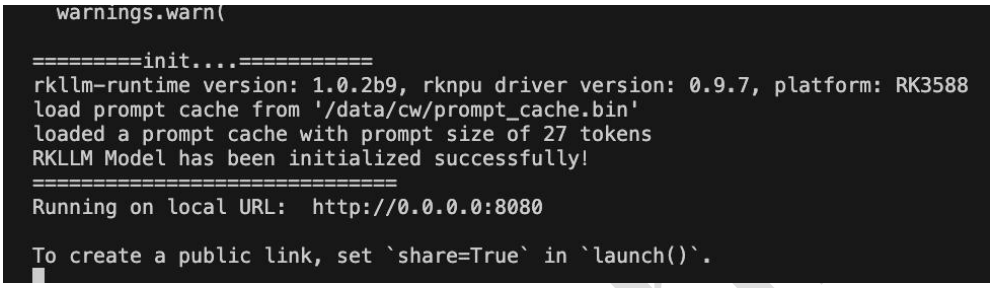

Figure 3-5 Successful deployment of RKLLM-Server-Gradio in terminal

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 similar to RKLLM-Server-Flask. It also uses the ctypes library to directly call the RKLLM Runtime library to perform RKLLM model inference. The specific deployment implementation process can be referenced in Figure below:

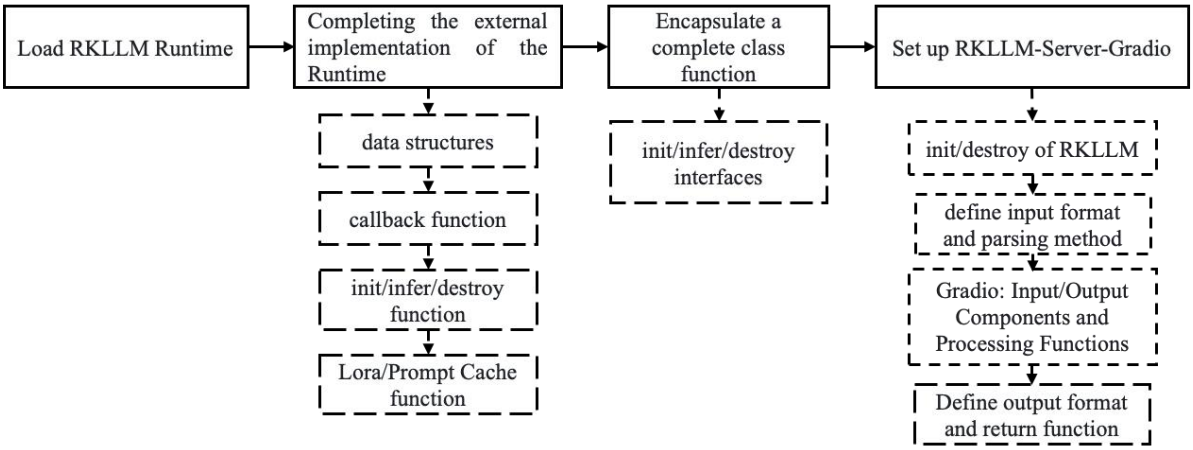

Figure 3-6 Overview of the RKLLM-Server-Flask Deployment Implementation Process

The difference is that RKLLM-Server-Gradio chooses to use the gradio library to implement the server setup and complete communication with the client, providing a simple web-based service. This requires specific handling of the Gradio interface in RKLLM-Server-Gradio as follows:

1) The Gradio function library provides complete communication for input and output data, so there is no need to define complex input-output implementations for the server via Flask.

2) Gradio is a deployment framework based on different control elements. During usage, it is necessary to call various Gradio components to complete the interface design and specify the trigger conditions, function call logic, and the data flow logic between different components for each element.

Users can refer to the main code in rkllm_server/gradio_server.py to understand the specific implementation of RKLLM-Server-Gradio, and by modifying the initialization definitions for the RKLLM model within it, they can implement different custom models. Additionally, users can refer to the deployment example of the RKLLM-Server-Gradio to deploy their own custom server.

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 following Figure. 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/")
```

Figure 3-7 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