

PR #38939 完整报告

sgl-project/sglang

[Rust] Use Dynamo native renderers when chat templates are missing

合并时间: 2026-09-15 08:29

原文链接: <http://prhub.com.cn/sgl-project/sglang/pull/38939>

执行摘要

本 PR 让 Rust 服务器在模型缺少 `chat_template` 时回退到 `dynamo_renderer::native_formatter_for`, 使 DeepSeek V4 等无模板模型首次可用 `/v1/chat/completions`。核心工作包括: Python 侧传递 `model_type` 契约、`OAISChatLikeRequest` 适配层支持 `chat_template_kwargs` 透传、以及 V4 流式推理的初始状态修正。依赖上 `dynamo-renderer` 升级到 5.1.2。整体属于 Rust 服务器 chat 链路的通用能力补齐, 不触及 Python 主调度与内核路径。

功能与动机

DeepSeek V4 的 checkpoint 不带 `chat_template`——其 prompt 由 Python 代码按架构构造 (`encoding_dsv4.py / resolve_chat_encoding_spec`)。Rust 服务器此前只有模板路径, `load_chat_formatter` 返回 `TemplateError::Missing` 后 `load_chat_support` 会直接禁用 `/v1/chat/completions` (PR body 原话: *Models such as DeepSeek V4 ship no chat_template, so the Rust frontend disabled /v1/chat/completions for them*)。

本 PR 的目标是把这种 "代码内建编码器" 能力接入 Rust 服务器: 模板缺失时委托给 Dynamo 的 `native_formatter_for`, 并用 Python 侧传递的 `model_type` 作为选择依据, 让 Rust 前端与 Python 前端在同一模型上获得一致的 chat 体验。

实现拆解

1. 契约扩展: Python → Rust 传递 `model_type` - `rust/sglang-server/src/message/config.rs`: `ModelConfig` 新增 `pub model_type: Option<String>`, `py_new` 签名同步增加关键字参数, 默认 `None`。 - `python/sglang/srt/rust_server/config.py`: `_build_server_args` 用 `getattr(mc.hf_config, "model_type", None)` 读取模型类型并传入 Rust 扩展。
2. 模板回退链: `native_formatter` 兜底 - `template_loader.rs`: `load_chat_formatter` 新增 `model_type` 参数; 定义 `native_formatter` 闭包, 仅在未指定 `--chat-template` 时调用 `native_formatter_for` (`model_type` 与 `model_path` 均转小写)。 - 回退路径: 无 config 文件时尝试原生编码器; config 存在但模板缺失 (`TemplateError::Missing`) 时也尝试原生编码器; 显式指定模板时保持原逻辑。 - `openai.rs` 的 `load_chat_support` 负责把 `model_config.model_type` 接入调用链。
3. 渲染接口升级: 适配 `OAISChatLikeRequest` - `Cargo.toml`: `dynamo-renderer` 5.0.0 → 5.1.2, 新增 `minijinja` 依赖; 新版本渲染要求 `trait` 对象且模板变量为 `minijinja::Value`。 - `template.rs`: 新增 `ChatTemplateKwargs` 与 `TemplateRequest` 包装结构体, 把

CreateChatCompletionRequest 委托给 trait 方法，并通过 chat_template_args() 返回额外变量。

- 请求层透传：chat_template_kwarg 进入模板 - chat.rs：新增 ChatRequest (serde(flatten) 展开标准请求并追加可选 chat_template_kwarg)，prepare_chat_request 与 formatter.render 均接收该参数。
- 流式推理初始化：V4 无起始标记场景 - chat.rs：计算 starts_in_reasoning——reasoning_parser 为 deepseek-v4 相关名称且 prompt 以 thinking 结尾时为 true，并透传至 chat_event_stream。 - reasoning.rs：ReasoningStreamSplitter 新增 starts_in_reasoning 构造参数，parser 惰性构建时调用 set_in_reasoning(true)，使首个流式帧直接按思考内容解析。
- 测试配套 - template.rs 新增 chat_template_kwarg_reach_the_template、missing_template_falls_back_to_native_formatter；reasoning.rs 新增 v4_streaming_separates_prefilled_reasoning。 - 无独立集成测试文件，依赖现有 Rust 测试套件与 CI 重跑验证（rust-server 组曾失败后重跑通过）。

关键源码片段

rust/sglang-server/src/api_server/openai/template.rs

核心改动文件：新增 ChatTemplateKwarg 类型与 TemplateRequest 适配层，实现 OAISChatLikeRequest trait，render 方法支持 kwarg 透传，并补充两个关键单元测试（kwarg 注入与 native formatter 回退）。

```
/// ChatFormatter 的渲染入口：HuggingFace 变体现在接受 OAISChatLikeRequest trait。  
/// 由于 dynamo-renderer 5.1.2 的渲染接口要求 trait 对象，而协议类型本身不携带  
/// chat_template_kwarg，因此用 TemplateRequest 包装层补上这块数据。
```

```
/// 额外变量集：Jinja 模板渲染时可通过 chat_template_kwarg 注入 thinking 等字段。  
pub type ChatTemplateKwarg = HashMap<String, serde_json::Value>;
```

```
/// 协议请求 + 额外变量的组合，实现 dynamo-renderer 定义的渲染接口。
```

```
struct TemplateRequest<'a> {  
    request: &'a CreateChatCompletionRequest,  
    kwarg: Option<&'a ChatTemplateKwarg>,  
}  
  
impl OAISChatLikeRequest for TemplateRequest<'_> {  
    fn model(&self) -> String {  
        self.request.model()  
    }  
    fn messages(&self) -> minijinja::Value {  
        self.request.messages()  
    }  
    fn typed_messages(&self) -> Option<&[ChatCompletionRequestMessage]> {  
        self.request.typed_messages()  
    }  
    fn tools(&self) -> Option<minijinja::Value> {
```

```

        self.request.tools()
    }
    fn tool_choice(&self) -> Option<minijinja::Value> {
        self.request.tool_choice()
    }
    fn response_format(&self) -> Option<minijinja::Value> {
        self.request.response_format()
    }
    fn reasoning_effort(&self) -> Option<minijinja::Value> {
        self.request.reasoning_effort()
    }
    fn should_add_generation_prompt(&self) -> bool {
        self.request.should_add_generation_prompt()
    }
    // 关键桥接：把 HTTP 层解析出的 chat_template_kwargs 交给模板引擎。
    fn chat_template_args(&self) -> Option<&ChatTemplateKwargs> {
        self.kwargs
    }
    fn extract_text(&self) -> Option<TextInput> {
        self.request.extract_text()
    }
    fn mm_processor_kwargs(&self) -> Option<&serde_json::Value> {
        self.request.mm_processor_kwargs()
    }
}

```

rust/sglang-server/src/api_server/openai/chat.rs

请求入口改造：新增 ChatRequest 包装结构体（serde flatten 支持 chat_template_kwargs），计算 starts_in_reasoning 并透传到流式推理分流器。

```

/// HTTP 请求包装：serde(flatten) 展开标准 OpenAI 请求，同时捕获
/// chat_template_kwargs（OpenAI 协议没有该字段，属于 SGLang 扩展）。
#[derive(Deserialize)]
struct ChatRequest {
    #[serde(flatten)]
    request: CreateChatCompletionRequest,
    chat_template_kwargs: Option<ChatTemplateKwargs>,
}

// 在 chat_completions 入口解析后传给 prepare_chat_request。
let (request, prompt) =
    match prepare_chat_request(&state, request, chat_template_kwargs.as_ref()).await {
        Ok(prepared) => prepared,
        Err(response) => return response,
    };

// V4 模型的 prefill 阶段已经输出了 " thinking" 起始标记，因此流式阶段
// 不会再出现新的 <think> 起始边界；必须告诉 reasoning splitter 从
// 思考状态开始，否则前序 token 会被误判为正常答案文本。

```

```

let starts_in_reasoning = matches!(
    reasoning_parser.as_deref(),
    Some("deepseek-v4" | "deepseek_v4" | "deepseekv4")
) && prompt.ends_with(" thinking");

// 该标记随后传入 chat_event_stream, 用于初始化每个 choice 的
// ReasoningStreamSplitter (见 reasoning.rs 的 starts_in_reasoning 字段)。

```

rust/sglang-server/src/api_server/openai/template_loader.rs

实现核心回退逻辑: native_formatter 闭包与三条回退路径, 决定无模板模型是否能加载 chat 支持。

```

pub(super) fn load_chat_formatter(
    config_file: Option<&str>, // tokenizer_config.json 路径
    model_path: Option<&str>, // 模型目录
    model_type: Option<&str>, // Python 侧传入的 HF model_type
    chat_template_arg: Option<&str>, // 命令行 --chat-template 参数
) -> Result<ChatFormatter, TemplateError> {
    // 1. 注册表名称优先: builtin 模板不依赖任何文件。
    if let Some(argument) = chat_template_arg
        && let Some(spec) = builtin_template(argument)
    {
        return Ok(ChatFormatter::Legacy(Box::new(LegacyFormatter { spec })));
    }

    // 2. 模型路径推断 legacy 会话模板, 与 Python 的 load_chat_template 对齐。
    if chat_template_arg.is_none()
        && let Some(model_path) = model_path
        && let Some(spec) = infer_legacy_template_from_model_path(model_path)
    {
        return Ok(ChatFormatter::Legacy(Box::new(LegacyFormatter { spec })));
    }

    // 3. 原生 formatter 回退: DeepSeek V4 等模型不带 chat_template,
    // Dynamo 内置了按架构编码的渲染器 (native_formatter_for),
    // 仅在未显式指定 --chat-template 时参与, 避免覆盖用户意图。
    let native_formatter = || match chat_template_arg {
        None => native_formatter_for(
            &model_type.map(str::to_lowercase),
            &model_path.unwrap_or_default().to_lowercase(),
        )
        .map(ChatFormatter::HuggingFace),
        Some(_) => None,
    };

    // 4. 没有 tokenizer_config.json 时只能依赖原生编码器。
    let Some(config_file) = config_file else {
        return native_formatter().ok_or(TemplateError::MissingConfig);
    };
}

```

```

// 5. 读取 tokenizer 配置交给 HF 渲染器。
let config_path = Path::new(config_file);
let config_text = read_to_string(config_path, "tokenizer config");
let mut config = parse_json(&config_text, config_path, "tokenizer config");

// 6. 未显式指定模板时：模板缺失（Missing）则回退原生编码器，
// 否则走正常 HuggingFace 模板渲染路径。
let Some(argument) = chat_template_arg else {
    return match formatter_from_config(&config) {
        Err(TemplateError::Missing) => native_formatter().ok_or(TemplateError::Missing),
        result => result,
    };
};

// 7. 显式指定 --chat-template 时的 jinja/JSON 文件处理逻辑保持不变。
// ...（原逻辑：校验路径、注入模板、构造 formatter）
}

```

评论区精华

本 PR 没有 review 评论，讨论集中在 issue 评论与 commit 演进：

```

rainj-me: /rerun-group rust-server、/rerun-test
test/registered/rust/test_run_rust_tests.py ...

```

说明 rust-server 组与核心端点测试曾失败，重跑后全部通过；合并前 hnyls2002 再次 [/tag-and-rerun-ci](#) 确认最终状态。

commit 迭代中值得注意的设计取舍：初始提交是 "为 DSV4-Flash 定制内置 encoder"，中途演进为 "通用 native formatter 回退" (a602b9d)，并删掉了 Python/Rust 渲染一致性测试 (04a52d5) ——作者选择把一致性责任收敛到 dynamo-renderer 的单一实现里，避免双端测试漂移。

风险与影响

1. dynamo-renderer 接口升级：5.1.2 将渲染接口 trait 化 (OAIDChatLikeRequest)，所有 HuggingFace 渲染路径都经过新的 TemplateRequest 适配层，存在渲染结果差异的回归面；dynamo-protocols 与 dynamo-tokenizers 也连带升级。
2. 字符串匹配脆弱性：prompt.ends_with(" thinking") 依赖 native formatter 输出格式的精确性，若 Dynamo 调整 V4 prompt 尾部格式，该判断会静默失效。
3. 行为变更：原本因无模板而被禁用 chat 的模型，只要 Dynamo 认识其架构就会自动启用 chat API，属于语义变化，需要留意不同架构的 prompt 质量。
4. 测试覆盖：新增测试均为 Rust 单元测试，缺少独立的集成 / 端到端测试文件验证真实 HTTP 请求下的行为。

关联脉络

本 PR 是 DeepSeek V4 支持线的一环：前序有 DSV4 的 KV 缓存时序修复 (PR#38409)、AMD 平台 fp8 双池 KV (PR#37413)、MI355X 部署文档 (PR#39396)。这些 PR 分别补齐了 DSV4 在调度、缓存、硬件后端与文档侧的能力，而本 PR 补齐的是 Rust 服务器前端的 chat 入口。整体上，SGLang 正在把 DSV4 从 Python 主路径向 Rust 服务器路径全面对齐。