PR #7204 完整报告

verl-project/verl

[rollout] fix: decode per-turn LLM tokens in traces

合并时间：2026-07-31 11:33

原文链接：http://prhub.com.cn/verl-project/verl/pull/7204

执行摘要

- 一句话：修复多轮 rollout trace 缺失逐轮解码文本
- 推荐动作：值得精读。它展示了如何用最小 API 面修复一个真实用户问题：字段回退解析、输出拷贝保护、按配置注入依赖这三个手法都有可复用性。与未合并的 #5588 对照阅读，可以理解「窄修复 vs 通用 API」的取舍逻辑。若你的团队依赖 agentic RL 的可视化分析，建议尽快合入升级。

功能与动机

issue #3515 报告在 agentic RL 示例（gsm8k_multiturn_grpo + SGLang async + MLflow）中设置 token2text=True 后仍然看不到 prompt_text。PR body 指出根因：原解码器只从被 trace 函数的输出上读取 prompt_ids/response_ids，并要求被装饰对象自带 tokenizer；而 LLMServerClient.generate 的 prompt_ids 是入参、返回 TokenOutput.token_ids，client 并不拥有 AgentLoop 的 tokenizer。因此用户在多轮 agent 场景下无法在 trace 中看到解码文本。

实现拆解

1. 重构解码辅助逻辑（verl/utils/rollout_trace.py）：把原来嵌套在 rollout_trace_op 内部的 add_token2text 闭包提升为模块级函数 _trace_field、_trace_output_copy、_add_token2text。_add_token2text 先校验实例上有可调用的 tokenizer.decode，再按「输出优先、输入回退」的顺序解析字段：prompt_ids 依次看 output.prompt_ids、inputs.prompt_ids；response_ids 依次看 output.response_ids、output.token_ids。
2. 输出拷贝保护：_trace_output_copy 对 Pydantic 模型用 model_dump()、对 dict 用 dict()、对普通对象用 vars() 拷贝，再把解码结果写入拷贝，保证原返回结果不被修改；若无法拷贝则原样返回。
3. 统一三个后端路径：weave、mlflow、trackio 的分支都改为调用 _add_token2text(self, inputs, result)，保持共享转换路径，避免后端行为分叉。
4. 注入 tokenizer（verl/experimental/agent_loop/agent_loop.py）：AgentLoopWorker.__init__在 trace_config 的 token2text 为 True 时执行 self.llm_client.tokenizer = self.tokenizer，让装饰器能拿到解码器。
5. 测试与文档配套：tests/utils/test_rollout_trace_on_cpu.py 新增 mock_mlflow_client fixture、FakeTokenizer、GeneratedTokenOutput 模型和 LLMGenerateTraceClass，新增 test_token2text_decodes_llm_generate_input_and_output 断言 span.set_outputs 收

到解码字段且返回值不变；docs/advance/rollout_trace.rst 更新说明，明确 prompt_text/response_text 会同时出现在完整轨迹与逐轮 LLM 调用上。

关键文件：

- verl/utils/rollout_trace.py（模块 轨迹工具；类别 source；类型 core-logic；符号 _trace_field, _trace_output_copy, _add_token2text, rollout_trace_op）：核心解码逻辑所在文件：新增 _trace_field/_trace_output_copy/_add_token2text，将原装饰器内部闭包提升为模块级函数，并支持从 inputs 回退取 prompt_ids、从 token_ids 回退取 response_ids，是本次修复的主干。
- tests/utils/test_rollout_trace_on_cpu.py（模块 单元测试；类别 test；类型 test-coverage；符号 mock_mlflow_client, GeneratedTokenOutput, FakeTokenizer, LLMGenerateTraceClass）：新增针对 MLflow span 的 CPU 单测，覆盖 LLMServerClient.generate 输入输出解码场景，并验证返回值未被污染。
- verl/experimental/agent_loop/agent_loop.py（模块 Agent 循环；类别 source；类型 core-logic；符号 AgentLoopWorker.init）：在 token2text 开启时把 AgentLoop tokenizer 注入 llm_client，是让 per-turn generate 可解码的前提。
- docs/advance/rollout_trace.rst（模块 文档；类别 docs；类型 documentation）：同步更新 rollout trace 文档，说明 prompt_text/response_text 现在同时出现在完整轨迹与逐轮 LLM 调用上。

关键符号：_add_token2text, _trace_field, _trace_output_copy, rollout_trace_op, AgentLoopWorker.init, test_token2text_decodes_llm_generate_input_and_output

关键源码片段

verl/utils/rollout_trace.py

核心解码逻辑所在文件：新增 _trace_field/_trace_output_copy/_add_token2text，将原装饰器内部闭包提升为模块级函数，并支持从 inputs 回退取 prompt_ids、从 token_ids 回退取 response_ids，是本次修复的主干。

```
# 从输出对象中按字段名取值，兼容 Pydantic 对象与普通 dict
def _trace_field(value, field_name):
    if isinstance(value, dict):
        return value.get(field_name)
    return getattr(value, field_name, None)


# 为输出做一次拷贝，避免向原始返回结果注入额外字段：
# - Pydantic 模型用 model_dump()，防止 vars() 拿到内部 __dict__ 引用
# - dict 直接 copy，普通对象取 vars() 的拷贝
# - 无法拷贝时返回 None，由调用方决定原样返回
def _trace_output_copy(output):
    if isinstance(output, BaseModel):
        return output.model_dump()
    if isinstance(output, dict):
        return dict(output)
```

```
    if hasattr(output, "__dict__"):
        return dict(vars(output))
    return None


# 核心解码逻辑：优先用输出上的 prompt_ids/response_ids（旧完整轨迹布局），
# 否则回退到 LLMServerClient.generate 的逐轮调用布局——
# prompt_ids 在 inputs 里，响应 token 在输出的 token_ids 里。
async def _add_token2text(instance, inputs, output):
    tokenizer = getattr(instance, "tokenizer", None)
    decode = getattr(tokenizer, "decode", None)
    if not callable(decode):
        # 没有可用的解码器时保持原样返回，不阻断 trace
        return output

    prompt_ids = _trace_field(output, "prompt_ids")
    if prompt_ids is None:
        prompt_ids = inputs.get("prompt_ids")
    response_ids = _trace_field(output, "response_ids")
    if response_ids is None:
        response_ids = _trace_field(output, "token_ids")

    # 既没有输入 prompt 也没有输出 token，无需解码
    if prompt_ids is None and response_ids is None:
        return output

    output_copy = _trace_output_copy(output)
    if output_copy is None:
        return output

    loop = get_event_loop()
    # 解码放到线程池执行，避免阻塞事件循环
    if prompt_ids is not None:
        output_copy["prompt_text"] = await loop.run_in_executor(None, decode, prompt_ids)
    if response_ids is not None:
        output_copy["response_text"] = await loop.run_in_executor(None, decode, response_ids)
    return output_copy
```

tests/utils/test_rollout_trace_on_cpu.py

新增针对 MLflow span 的 CPU 单测，覆盖 LLMServerClient.generate 输入输出解码场景，并验证返回值未被污染。

```
class GeneratedTokenOutput(BaseModel):
    # 模拟 LLMServerClient.generate 的返回结构：只有 token_ids，没有 response_ids
    token_ids: list[int]


class FakeTokenizer:
    # 简单的确定性解码器，便于断言解码后的文本内容
```

```
    def decode(self, token_ids):
        return "decoded:" + ",".join(str(token_id) for token_id in token_ids)


class LLMGenerateTraceClass:
    # 复刻异步 agentic RL 路径：prompt_ids 是入参，返回 TokenOutput
    def __init__(self):
        self.tokenizer = FakeTokenizer()

    @rollout_trace_op
    async def generate(self, *, prompt_ids):
        return GeneratedTokenOutput(token_ids=[3, 4])


async def test_token2text_decodes_llm_generate_input_and_output(mock_mlflow_client):
    # 验证 MLflow span 的 outputs 里被补上 prompt_text/response_text，
    # 同时函数返回值仍然是原始 Pydantic 对象（不被污染）。
    _, mock_span = mock_mlflow_client
    RolloutTraceConfig.init(
        project_name="my-project",
        experiment_name="my-experiment",
        backend="mlflow",
        token2text=True,
    )

    instance = LLMGenerateTraceClass()
    result = await instance.generate(prompt_ids=[1, 2])

    assert result == GeneratedTokenOutput(token_ids=[3, 4])
    mock_span.set_inputs.assert_called_once_with({"prompt_ids": [1, 2]})
    mock_span.set_outputs.assert_called_once_with(
        {
            "token_ids": [3, 4],
            "prompt_text": "decoded:1,2",
            "response_text": "decoded:3,4",
        }
    )
```

verl/experimental/agent_loop/agent_loop.py

在 token2text 开启时把 AgentLoop tokenizer 注入 llm_client，是让 per-turn generate 可解码的前提。

```
        trace_config = self.rollout_config.trace
        if trace_config.get("token2text", False):
            # rollout_trace_op 作用在 LLM client 上，因此需要把 AgentLoop
            # 持有的 tokenizer 注入 client，逐轮 generate 才能解码文本。
            self.llm_client.tokenizer = self.tokenizer
        RolloutTraceConfig.init(
            self.rollout_config.trace.project_name,
```

```
            self.rollout_config.trace.experiment_name,
            trace_config.get("backend"),
            trace_config.get("token2text", False),
            trace_config.get("max_samples_per_step_per_worker", None),
        )
```

评论区精华

本 PR 的 Review 评论为空，wuxibin89 直接批准合并；issue 评论仅有 CLA 和 Gemini Code Assist 的自动提示。核心设计取舍写在 PR body 中：作者对比了未合并的 #5588（通用可配置字段映射 API + 全局 tokenizer fallback），明确选择「更窄的修复」——只为既有 rollout 字段做回退解码，不新增公共 API、不引入全局 tokenizer 状态。这种取舍保证了配置和装饰器 API 完全不变，也让 reviewer 能快速确认风险边界。

- LLM client tokenizer 注入方式 (design): 采用按需属性注入，保持 LLM client 接口不变；风险是注入字段非正式 API，未来 client 类型变化时可能失效。
- 窄修复 vs 通用字段映射 API (design): 选择窄修复，避免公共 API 与全局状态，降低维护负担；对应地放弃了 #5588 的通用扩展性。

风险与影响

- 风险： 1) 核心 trace 路径改动（rollout_trace.py）：三个后端都经过同一解码函数，若 _trace_output_copy 对某种输出类型返回 None，将静默跳过解码，不会报错但不满足预期； 2) 输出拷贝语义：Pydantic 模型用 model_dump() 深拷贝，dict 与 vars 拷贝为浅拷贝，嵌套敏感字段可能被共享； 3) tokenizer 注入依赖属性赋值（agent_loop.py 中 self.llm_client.tokenizer = self.tokenizer），不属于 client 正式接口，未来若 client 类型变化可能失效； 4) 解码通过线程池 run_in_executor 执行，token 较长时增加线程调度开销； 5) 两个真实 Weave/MLflow 集成测试默认跳过，真实 tokenizer 与后端行为未被 CI 覆盖。
  - 影响：用户侧：配置了 token2text 且使用 MLflow/Weave/Trackio 的 agentic RL 用户，将能在每个 LLMServerClient.generate 的 span 中看到 prompt_text/response_text，无需修改任何 yaml 或代码；系统侧：trace span 输出体积增加（新增两个文本字段），解码工作在线程池执行，对主事件循环无阻塞；团队侧：rollout trace 的字段解析逻辑被收敛到模块级函数，后续维护者需要理解「输出优先、输入回退」的双布局规则。
  - 风险标记：核心 trace 路径变更，tokenizer 属性注入，解码线程池开销，真实后端集成测试跳过

关联脉络

- 暂无明显关联 PR