PR #6055 完整报告

verl-project/verl

[rollout] feat: improve error messages for malformed tool calls

合并时间：2026-04-24 10:31

原文链接：http://prhub.com.cn/verl-project/verl/pull/6055

执行摘要

- 一句话：改进 ToolAgentLoop 工具调用错误信息，提供更具可操作性的反馈
- 推荐动作：该 PR 值得精读，尤其是 _call_tool 方法的拆分逻辑——展示了如何将一个 catching all 的异常处理重构为细粒度、信息丰富的错误反馈，这对 LLM agent 的 self-correction 能力至关重要。建议合并后关注 review 中提出的 finally 块风险，并考虑后续改进。

功能与动机

PR body 中明确指出：'Split the catch-all exception handler in _call_tool() into specific error cases so the LLM receives actionable feedback for self-correction'。原代码将所有异常统一捕获为 'Error when executing tool: {e}'，导致 LLM 无法区分是函数名错误、参数解析错误还是工具执行错误，从而无法自我纠正。

实现拆解

变更核心在 tool_agent_loop.py 的 _call_tool 方法，将原先单一 try-except 结构拆分为三个阶段，每个阶段都有独立的错误处理：

1. 工具名称验证：在 tool_call.name 不在 active_tools 中时，立即返回包含可用工具列表的错误消息，例如：Unknown function 'calculater'. Available tools: ['calculator', 'search']。
2. 参数 JSON 解析验证：将 json.loads 从主 try 块移到独立的 try-except 中，专门捕获 json.JSONDecodeError 和 TypeError，返回包含工具名的错误消息：Invalid JSON in arguments for 'calculator': ...。
3. 工具创建与执行：保留原有创建和执行逻辑，但错误消息改为 Error executing tool '{tool_name}': {e}，明确包含工具名称，方便 LLM 定位问题。

配套新增了完整的 CPU 单元测试文件 test_call_tool_on_cpu.py，通过 mock 模拟 ToolAgentLoop，覆盖所有新添加的错误路径。

关键文件：

- verl/experimental/agent_loop/tool_agent_loop.py（模块 智能体循环；类别 source；类型 core-logic；符号 _call_tool）：核心变更文件：拆分 _call_tool 方法的错误处理逻辑，提供细粒度错误消息

- tests/experimental/agent_loop/test_call_tool_on_cpu.py（模块 测试；类别 test；类型 test-coverage；符号 FakeFunctionCall, FakeAgentData, FakeTool, FakeFailingTool）：配套新增的纯 CPU 单元测试，覆盖所有新增错误路径，无需 GPU 即可运行

关键符号：_call_tool

关键源码片段

verl/experimental/agent_loop/tool_agent_loop.py

核心变更文件：拆分 _call_tool 方法的错误处理逻辑，提供细粒度错误消息

```
# verl/experimental/agent_loop/tool_agent_loop.py

async def _call_tool(
    self, tool_call: FunctionCall, tools_kwargs: dict[str, Any], agent_data: AgentData
) -> tuple[ToolResponse, float, dict]:
    """Call tool and return tool response."""
    active_tools = getattr(agent_data, "_active_tools", self.tools)

    # 阶段 1: 验证函数名是否存在
    tool_name = tool_call.name
    if tool_name not in active_tools:
        available = list(active_tools.keys())
        msg = f"Unknown function '{tool_name}'. Available tools: {available}"
        logger.warning(msg)
        return ToolResponse(text=msg), 0.0, {} # 返回包含可用工具列表的提示

    # 阶段 2: 验证参数是否为合法 JSON
    try:
        tool_args = json.loads(tool_call.arguments)
    except (json.JSONDecodeError, TypeError) as e:
        msg = f"Invalid JSON in arguments for '{tool_name}': {e}"
        logger.warning(msg)
        return ToolResponse(text=msg), 0.0, {} # 指出哪个工具的参数格式错误

    # 阶段 3: 创建并执行工具
    tool, instance_id = None, None
    try:
        tool = active_tools[tool_name]
        kwargs = tools_kwargs.get(tool_name, {})
        instance_id, _ = await tool.create(create_kwargs=kwargs.get("create_kwargs", {}))
        tool_execution_response, tool_reward, res = await tool.execute(
            instance_id, tool_args, agent_data=agent_data
        )
    except Exception as e:
        logger.warning(f"Error executing tool '{tool_name}': {e}")
        return ToolResponse(text=f"Error executing tool '{tool_name}': {e}"), 0.0, {} #
        明确包含工具名
    finally:
```

```
            if tool and instance_id:
                await tool.release(instance_id) # 注意：release 异常会掩盖原始错误

        # 后续截断逻辑不变 ...
        tool_response_text = tool_execution_response.text
        if tool_response_text and len(tool_response_text) > self.max_tool_response_length:
            if self.tool_response_truncate_side == "left":
                tool_response_text = tool_response_text[: self.max_tool_response_length] + "...
                (truncated)"
            elif self.tool_response_truncate_side == "right":
                tool_response_text = "(truncated)..." + tool_response_text[-self.max_tool_response_
                length:]
            else:
                length = self.max_tool_response_length // 2
                tool_response_text = tool_response_text[:length] + "...(truncated)..." + tool_response_
                text[-length:]

        tool_response_kwargs = {"text": tool_response_text}
        # 多媒体数据附加大致不变 ...
        return ToolResponse(**tool_response_kwargs), tool_reward, res
```

tests/experimental/agent_loop/test_call_tool_on_cpu.py

配套新增的纯 CPU 单元测试，覆盖所有新增错误路径，无需 GPU 即可运行

```
# tests/experimental/agent_loop/test_call_tool_on_cpu.py ( 新增 )

"""Unit tests for ToolAgentLoop._call_tool error handling (no GPU required)."""

import unittest
from dataclasses import dataclass, field
from typing import Any
from unittest.mock import MagicMock

from verl.tools.schemas import ToolResponse


@dataclass
class FakeFunctionCall:
    """模拟函数调用，用于测试"""
    name: str
    arguments: str


@dataclass
class FakeAgentData:
    """模拟agent数据"""
    tools_kwargs: dict = field(default_factory=dict)

class FakeTool:
    """模拟执行成功的工具"""
```

```
    def __init__(self, name: str):
        self.name = name

    async def create(self, create_kwargs=None):
        return "instance_1", ToolResponse()

    async def execute(self, instance_id, parameters, **kwargs):
        return ToolResponse(text=f"OK: {parameters}"), 1.0, {}

    async def release(self, instance_id):
        pass


class FakeFailingTool(FakeTool):
    """模拟执行时抛出异常的工具"""
    async def execute(self, instance_id, parameters, **kwargs):
        raise RuntimeError("database connection failed")


def _make_tool_agent_loop(tools: dict[str, Any]):
    """创建最小ToolAgentLoop mock实例，绑定真实的_call_tool方法"""
    from verl.experimental.agent_loop.tool_agent_loop import ToolAgentLoop

    mock = MagicMock(spec=ToolAgentLoop)
    mock.tools = tools
    mock.max_tool_response_length = 10000
    mock.tool_response_truncate_side = "left"
    mock._call_tool = ToolAgentLoop._call_tool.__get__(mock, ToolAgentLoop) # 绑定真实方法
    return mock


class TestCallToolErrorHandling(unittest.IsolatedAsyncioTestCase):
    """测试_call_
    tool的六个用例：正常调用、未知函数名、无效JSON、空参数、None参数、工具执行失败"""

    def setUp(self):
        self.tools = {
            "calculator": FakeTool("calculator"),
            "search": FakeTool("search"),
        }
        self.loop = _make_tool_agent_loop(self.tools)
        self.agent_data = FakeAgentData()

    async def test_valid_tool_call(self):
        # 验证正常调用返回 reward=1.0
        tool_call = FakeFunctionCall(name="calculator", arguments='{"a": 3, "b": 5}')
        response, reward, _ = await self.loop._call_tool(tool_call, {}, self.agent_data)
        assert reward == 1.0
        assert "OK" in response.text
```

```
    async def test_unknown_function_name(self):
        # 验证未知函数名消息中包含可用工具列表
        tool_call = FakeFunctionCall(name="calculater", arguments='{"a": 3}')
        response, reward, _ = await self.loop._call_tool(tool_call, {}, self.agent_data)
        assert reward == 0.0
        assert "Unknown function" in response.text
        assert "calculater" in response.text
        assert "calculator" in response.text
        assert "search" in response.text

    async def test_invalid_json_arguments(self):
        # 验证无效 JSON 参数错误包含工具名
        tool_call = FakeFunctionCall(name="calculator", arguments="{a: 3}")
        response, reward, _ = await self.loop._call_tool(tool_call, {}, self.agent_data)
        assert reward == 0.0
        assert "Invalid JSON" in response.text
        assert "calculator" in response.text

    # 其余测试类似 ...
```

评论区精华

> 评论者：gemini-code-assist[bot]（高优先级）— 在 tool_agent_loop.py 第 471 行（finally 块附近）指出：except Exception 块中返回错误后，finally 块中如果 tool.release() 再抛出异常，会掩盖原始错误。建议将 release 调用用独立的 try-except 包裹。 结论：该问题未在 PR 中得到解决，当前实现仍然存在此风险。

PR 作者没有回应此评论，且 wuxibin89 直接批准了 PR。

- finally 块中 tool.release() 异常可能掩盖原始错误 (correctness): PR 未采纳该建议，当前实现仍存在此风险，但 release 方法通常简单，实际触发概率低。

风险与影响

- 风险：
  1. finally 块异常掩盖风险：正如 review 中指出的，若 tool.release() 在执行过程中抛出异常，会覆盖原始工具执行错误，导致 LLM 看到的是 release 失败信息而非真正的执行错误。但 release 通常是轻量操作（如返回数据库连接），风险较低。
  2. 回归风险低：变更仅针对错误路径的拆分，正常执行路径逻辑不变；新测试覆盖了主要错误情况。
  3. 兼容性风险低：错误消息格式改变，但下游（如 agent loop）仅检查 reward 是否为 0，不解析文本内容，因此不影响决策逻辑。
- 影响：
  - 对用户（LLM agent）：正面影响。现在工具调用失败时 LLM 能收到具体错误信息，例如是函数名拼写错误、参数格式错误还是工具内部错误，从而能够自我纠正，提高成功率。
  - 对系统：几乎没有影响。变更仅影响错误路径，且为纯 CPU 操作。

- 对团队：提供了可复用的错误处理模式，并附带完善的单元测试，便于后续维护。
- 风险标记：finally 块异常掩盖风险，未处理的 *release* 异常

关联脉络

- PR #6074 [BREAKING] [env] refactor: *deprecate* verl/interactions: 同一文件 tool_agent_loop.py 在发版前有更广泛的废弃和迁移动作，本 PR 是 tool_agent_loop 功能增强的一部分。
- PR #6072 [veomni] feat: *enable* VeOmni *engine* for on-policy distillation: 同样涉及 *agent*_loop 模块，VeOmni 引擎的蒸馏功能也使用了 tool_agent_loop。