PR #6189 完整报告

verl-project/verl

[tool] feat: simpler function-based tool registration

合并时间：2026-05-08 11:43

原文链接：http://prhub.com.cn/verl-project/verl/pull/6189

执行摘要

- 一句话：新增 @function_tool 装饰器实现基于函数的工具注册。
- 推荐动作：值得精读，尤其关注新工具注册的设计模式（正交路径、自动 schema 推断、命名空间隔离）。对扩展 agent 框架的开发者有参考价值。建议结合测试文件理解使用方式。P.S. 作者计划后续将工具加载移至 AgentLoopWorker，可追踪新 PR。

功能与动机

PR #6189 旨在提供一种更简洁的注册 verl 工具的方式，避免编写 BaseTool 子类和 YAML 配置文件。开发者只需编写纯 Python 函数并添加 @function_tool 装饰器，即可让 LLM 调用该工具。PR 描述中指出这是与现有 YAML 驱动流程正交的新路径，可独立落地。

实现拆解

1. 新增 verl/tools/utils/function_tool.py，定义 FunctionTool dataclass 和 function_tool 装饰器。装饰器支持裸用（无括号）和带名称，通过 transformers.utils.get_json_schema 自动推断参数 schema，并支持手动提供 schema。FunctionTool 封装了调用接口，同步调用通过 asyncio.to_thread 异步执行，避免阻塞事件循环。
2. 在 verl/experimental/agent_loop/agent_loop.py 中新增 FunctionToolListWrap dataclass，用于在 Ray Actor 间传递函数工具列表。同时修改 ToolAgentLoop.__init__，接收 function_tools 参数，将其与原生工具合并，并检查名称冲突。
3. 在 verl/workers/config/rollout.py 的 multi_turn 配置中新增 function_tool_path 字段，指向用户定义的函数工具文件。AgentLoopWorker 会加载该文件并将工具列表注入 ToolAgentLoop。
4. 配套提供三个测试文件：test_function_tool_on_cpu.py 测试装饰器注册、schema 推断、类型映射（如 int | float 输出 number，Literal 输出 enum）；test_mixed_tools_on_cpu.py 测试原生工具与函数工具的共存加载与名称冲突检测；function_tool_examples.py 提供示例工具集（含同步和异步函数）。
5. 更新 docs/sglang_multiturn/multiturn.rst 文档，添加函数工具的使用说明和示例。

关键文件：

- verl/tools/utils/function_tool.py（模块 工具模块；类别 source；类型 core-logic；符号 FunctionTool, function_tool, call, load_function_tools_from_path）：核心新增文件，定义了 FunctionTool dataclass 和 function_tool 装饰器，实现函数式工具注册、schema 推

断和调用分发。

- tests/tools/test_function_tool_on_cpu.py（模块 单元测试；类别 test；类型 test-coverage；符号 test_decorator_registers_with_inferred_schema, test_int_float_union_emits_number_type, test_int_literal_emits_int_enum, test_load_and_call）：主要单元测试文件，全面测试装饰器注册、schema 推断、类型映射、加载与调用等核心功能。
- tests/tools/test_mixed_tools_on_cpu.py（模块 集成测试；类别 test；类型 test-coverage；符号 test_native_only_loader, test_function_only_loader, test_mixed_loader_no_collision, test_mixed_loader_with_collision）：测试原生工具与函数工具共存场景，验证命名冲突检测和合并加载逻辑。
- verl/experimental/agent_loop/tool_agent_loop.py（模块 Agent 循环；类别 source；类型 core-logic；符号 ToolAgentLoop.init, ToolAgentLoop._call_tool）：修改核心：ToolAgentLoop 集成函数工具的加载、合并与调用分发。
- verl/experimental/agent_loop/agent_loop.py（模块 Agent 循环；类别 source；类型 core-logic；符号 FunctionToolListWrap）：新增 FunctionToolListWrap 用于 Ray Actor 间传递函数工具列表。

关键符号：FunctionTool, function_tool, FunctionTool.call, normalize_function_tool_return, load_function_tools_from_path, _build_schema_from_fn, ToolAgentLoop.init, ToolAgentLoop._call_tool, FunctionToolListWrap.init

关键源码片段

verl/tools/utils/function_tool.py

核心新增文件，定义了 FunctionTool dataclass 和 function_tool 装饰器，实现函数式工具注册、schema 推断和调用分发。

```
# 版权所有 2025 Bytedance Ltd.
"""轻量级函数式工具注册。"""
from __future__ import annotations
import asyncio, inspect, logging, os, sys
from dataclasses import dataclass
from typing import Any, Callable, Optional
from transformers.utils import get_json_schema
from verl.tools.schemas import OpenAIFunctionToolSchema, ToolResponse

logger = logging.getLogger(__file__)
logger.setLevel(os.getenv("VERL_LOGGING_LEVEL", "WARN"))

# 全局注册表
FUNCTION_TOOL_REGISTRY: dict[str, FunctionTool] = {}
_LOADED_FUNCTION_TOOL_PATHS: dict[str, list[FunctionTool]] = {}

@dataclass
class FunctionTool:
```

```
    """Agent 循环依赖的载体对象。"""
    name: str
    fn: Callable[..., Any]
    tool_schema: OpenAIFunctionToolSchema
    is_async: bool = False

    async def call(self, parameters: dict[str, Any]) -> Any:
        """调用底层函数，同步函数通过线程池转换。"""
        if self.is_async:
            return await self.fn(**parameters)
        return await asyncio.to_thread(self.fn, **parameters)

def function_tool(
    name: Optional[str | Callable] = None, *,
    schema: Optional[OpenAIFunctionToolSchema | dict] = None,
):
    """注册 Python 函数为 verl 工具。支持裸装饰器 @function_tool。"""
    def _make_decorator(tool_name_override: Optional[str]):
        def decorator(fn: Callable):
            tool_name = tool_name_override or fn.__name__
            if isinstance(schema, OpenAIFunctionToolSchema):
                built_schema = schema
            elif isinstance(schema, dict):
                built_schema = OpenAIFunctionToolSchema.model_validate(schema)
            else:
                # 使用 transformers 自动推断 Schema
                built_schema = _build_schema_from_fn(fn, tool_name)
            entry = FunctionTool(
                name=tool_name, fn=fn, tool_schema=built_schema,
                is_async=inspect.iscoroutinefunction(fn),
            )
            # 检查名称冲突
            existing = FUNCTION_TOOL_REGISTRY.get(tool_name)
            if existing is not None and existing.fn is not fn:
                raise ValueError(
                    f"函数工具 '{tool_name}' 已被注册到 "
                    f"{existing.fn.__module__}.{existing.fn.__qualname__}; "
                    f"拒绝覆盖 {fn.__module__}.{fn.__qualname__}."
                )
            FUNCTION_TOOL_REGISTRY[tool_name] = entry
            logger.info("注册函数工具 '%s' 来自 %s.%s", tool_name, fn.__module__, fn.__qualname__
            )
            return fn
        return decorator

    if callable(name) and schema is None: # 裸装饰器
        return _make_decorator(None)(name)
    return _make_decorator(name)
```

```
# ... （其余辅助函数省略）
```

评论区精华

主要讨论：

- wuxibin89 要求支持裸装饰器 @function_tool（无括号），已实现。
- wuxibin89 要求添加 async 函数工具示例，已添加 fetch_url。
- gemini-code-assist[bot] 建议使用 transformers.utils.get_json_schema 替代手写 schema 推断，作者采纳。
- gemini-code-assist[bot] 建议 Union[int, float] 应映射为 number 类型，已通过 schema 宽松校验解决。
- wuxibin89 建议将工具加载移至 AgentLoopWorker 避免重复，作者答复会另开 PR 重构。
- 使用 transformers.utils.get_json_schema 替代手写 schema 推断 (design): 作者采纳，用 get_json_schema 替换了手动构建 schema 的代码。
- 支持 @function_tool 裸装饰器（无括号） (design): 添加了 if callable(name) and schema is None 分支，检测到 callable 时自动作为裸装饰器处理。
- 添加 async 函数工具示例 (testing): 在 function_tool_examples.py 中添加了 fetch_url 异步函数工具。
- 将工具加载移至 AgentLoopWorker 避免重复加载 (design): 暂时在 ToolAgentLoop 中加载，待后续 PR 重构。

风险与影响

- 风险：
  1. 命名冲突风险：函数工具与原生工具名称冲突时触发断言，但若用户未测试，可能导致运行时才发现。
  2. Schema 推断依赖：依赖 Google 风格 docstring 和类型注解，格式错误时 get_json_schema 抛出异常，影响启动。
  3. 性能开销：同步工具通过 asyncio.to_thread 调用，在高并发场景下可能有线程切换开销。
  4. 安全风险：function_tool_path 动态加载用户 Python 文件并执行，存在代码注入风险，应只用于受信任来源。
  5. 配置复杂度：新增 function_tool_path 配置项，需用户理解与 tool_config_path 的区分，错误配置可能导致工具缺失。
- 影响：
  - 用户侧：工具开发者受益，无需编写 YAML 和 BaseTool 子类，但需遵循 Google 风格 docstring 保证 schema 推断正确。
  - 系统侧：不影响现有 YAML 工具流程，两者可共存。ToolAgentLoop 在初始化时合并两个来源的工具列表，并确保名称唯一。对性能无显著影响，工具加载仅在 worker 启动时执行一次。
  - 团队侧：新增代码约 1675 行，但主要集中在新文件和测试，维护成本可控。

- 风险标记：配置复杂度，安全风险：*exec* 用户代码，依赖 docstring 格式，命名冲突可能，线程切换开销

关联脉络

- PR #5978 [related] previous similar PR: PR body 中提及搜索到类似 PR #5978，参考了其设计。