# PR #26815 完整报告

sgl-project/sglang

Add a deterministic token oracle and production write-input assertion

合并时间：2026-05-31 09:57

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/26815

## 执行摘要

- 一句话：新增确定性 token oracle 与 write-input 断言
- 推荐动作：该 PR 设计清晰，分层合理，值得精读。特别是 Token Oracle 的确定性哈希选择、perturb 与断言分离的测试策略、以及通过环境变量精细控制功能启用的思路，可作为类似可观测性功能的参考模板。

## 功能与动机

PR body 提到需要让 kv-canary 的 write-input 断言能够验证模型消费了正确的 input_ids/positions，而前提是有一个确定性的 token oracle 来预先知道每个请求的采样结果。此外，通过扰动机制可以证明断言检查链路在真实运行时是活跃的。

## 实现拆解

1. 确定性 Token Oracle 核心：在 python/sglang/srt/kv_canary/token_oracle/oracle.py 中定义 TokenOracle 协议和 HashOracle 实现，使用 SplitMix64 哈希将 (generalized_req_id, position) 确定性地映射到 token_id。
2. TokenOracleManager 与采样器：在 oracle_manager.py 实现 TokenOracleManager，负责填充预期输入（fill_expected_inputs）和生成下一 token（sample_next_tokens）；在 sampler.py 中注册 "token_oracle" 采样后端，替换标准采样逻辑，并通过 install_oracle_sampler 注入依赖。
3. Perturb 扰动机制：在 next_token_swap.py 中实现 maybe_perturb_swap_next_tokens，以指定概率随机交换批内两个请求的采样 token，仅影响采样输出，不碰 KV 路径，用于验证 write-input 断言能检测到输入不一致。
4. Write-Input 断言集成：在 single_forward_manager/manager.py 中通过 _should_enable_write_input_assert_for_launch 判断是否启用断言（默认关闭，且对 EAGLE draft 解码步骤跳过），利用 TokenOracleManager.fill_expected_inputs 填充预期值，在写入内核执行后比较实际 input_ids/positions。
5. 环境变量与配置：通过 server_args.py 注册 "token_oracle" 采样后端，在 environ.py 新增 SGLANG_KV_CANARY_ENABLE_TOKEN_ORACLE、SGLANG_KV_CANARY_ENABLE_WRITE_INPUT_ASSERT、SGLANG_KV_CANARY_PERTURB_NEXT_TOKEN_SWAP_PROB、SGLANG_KV_CANARY_PERTURB_WARMUP_STEPS 等变量；CanaryConfig 中新增

enable_write_input_assert 配置项。

6. 测试覆盖：包括 oracle 单元测试、mock 连线测试、token oracle manager 行为测试、perturb e2e 测试、EAGLE positions 回归测试等。

关键文件：

- python/sglang/srt/kv_canary/token_oracle/oracle.py（模块 KVCache；类别 source；类型 core-logic；符号 TokenOracle, expected_tokens, HashOracle, _splitmix64_tensor）：定义 TokenOracle 协议和 HashOracle 实现，是确定性 token 映射的核心算法。
- python/sglang/srt/kv_canary/token_oracle/oracle_manager.py（模块 KVCache；类别 source；类型 dependency-wiring；符号 TokenOracleManager, init, fill_expected_inputs, sample_next_tokens）：管理 TokenOracle 实例，提供 fill_expected_inputs 和 sample_next_tokens 两个核心方法，并处理不同 forward mode 下 generalized_req_id 的扩展。
- python/sglang/srt/kv_canary/token_oracle/sampler.py（模块 KVCache；类别 source；类型 dependency-wiring；符号 install_oracle_sampler, _OracleSampler, init, forward）：将 TokenOracleManager 注册为采样后端，替换标准采样逻辑，并集成 next-token-swap 扰动。
- test/registered/mock_model/test_self_unit_canary_mock_wiring.py（模块 单元测试；类别 test；类型 test-coverage；符号 _StubForwardBatch, _scalar_expected_token, TestFillExpectedInputs, test_sample_next_tokens_uses_next_position）：测试 TokenOracleManager 核心逻辑（fill_expected_inputs, sample_next_tokens），覆盖多种 forward mode。
- test/registered/mock_model/test_self_unit_oracle.py（模块 单元测试；类别 test；类型 test-coverage；符号 _signed_to_unsigned_i64, _call, TestHashOracle, test_hash_oracle_is_deterministic_for_same_inputs）：验证 HashOracle 的确定性和范围正确性，并对比 tensor 实现与标量 SplitMix64 参考实现。
- python/sglang/srt/kv_canary/perturb/next_token_swap.py（模块 KVCache；类别 source；类型 core-logic；符号 NextTokenSwapConfig, from_env, _get_config, maybe_perturb_swap_next_tokens）：实现 next-token-swap 扰动，用于测试断言链路活性，是测试基础设施的一部分。
- python/sglang/srt/kv_canary/single_forward_manager/manager.py（模块 KVCache；类别 source；类型 dependency-wiring；符号 _should_enable_write_input_assert_for_launch）：整合 write-input 断言到前向传播管理器中，对接 TokenOracleManager 和写内核。
- python/sglang/srt/kv_canary/token_oracle/install.py（模块 KVCache；类别 source；类型 core-logic；符号 install_token_oracle_from_env）：提供环境变量驱动的安装入口，简化外部调用。
- test/registered/kv_canary/test_self_e2e_pr_25015.py（模块 端到端测试；类别 test；类型 test-coverage；符号 _EaglePositionsBase, setUpClass, test_pr_25015_eagle_positions, TestEaglePositionsMisalignRegression）：回归测试 EAGLE positions 对齐问题，确保 token oracle 与 positions 配合正确。
- test/registered/mock_model/test_self_unit_install.py（模块 单元测试；类别 test；类型 test-coverage；符号 _make_server_args, TestInstallTokenOracleFromEnv,

test_install_token_oracle_from_env_disabled_returns_none,test_install_token_oracle_from_env_enabled_registers_oracle_backend）：测试install_token_oracle_from_env 的开关逻辑和正确性。

- test/registered/mock_model/test_self_unit_oracle_torch_vs_ref.py（模块 单元测试；类别 test；类型 test-coverage；符号 TestHashOracleTorchVsRef, test_hash_oracle_matches_scalar_splitmix64_ref, test_hash_oracle_batched_matches_scalar_splitmix64_ref）：对比 PyTorch 实现的 SplitMix64 与标量参考实现，确保数值一致。
- test/registered/kv_canary/test_self_unit_token_oracle.py（模块 单元测试；类别 test；类型 test-coverage；符号 TestTokenOracleManager, setUp, test_fill_expected_inputs_expands_draft_extend_generalized_req_ids_per_token）：测试 TokenOracleManager 的扩展逻辑（draft_extend 模式）。

关键符号：TokenOracleManager.fill_expected_inputs, TokenOracleManager.sample_next_tokens, HashOracle.expected_tokens, _OracleSampler.forward, maybe_perturb_swap_next_tokens, install_oracle_sampler, install_token_oracle_from_env, _should_enable_write_input_assert_for_launch

## 关键源码片段

### python/sglang/srt/kv_canary/token_oracle/oracle.py

定义 TokenOracle 协议和 HashOracle 实现，是确定性 token 映射的核心算法。

```
from __future__ import annotations
from dataclasses import dataclass
from typing import Protocol

import torch

class TokenOracle(Protocol):
    """Deterministic (generalized_req_id, position) -> token_id mapping."""
    def expected_tokens(
        self, *, generalized_req_ids: torch.Tensor, positions: torch.Tensor
    ) -> torch.Tensor: ...

@dataclass(frozen=True, slots=True, kw_only=True)
class HashOracle:
    """token_id = splitmix64(generalized_req_id XOR position) % vocab_size."""
    vocab_size: int

    def expected_tokens(
        self, *, generalized_req_ids: torch.Tensor, positions: torch.Tensor
    ) -> torch.Tensor:
        # 输入广义请求 ID 和位置，XOR 后经过 SplitMix64 哈希，再取模词表大小
        x = generalized_req_ids.to(torch.int64) ^ positions.to(torch.int64)
        x = _splitmix64_tensor(x)
        return _uint64_mod(x, self.vocab_size).to(torch.int32)
```

```
_C1: int = -4658895280553007687 # 0xBF58476D1CE4E5B9 作为有符号 int64
_C2: int = -7723592293110705685 # 0x94D049BB133111EB 作为有符号 int64

def _splitmix64_tensor(x: torch.Tensor) -> torch.Tensor:
    # 对张量逐元素执行 SplitMix64 混合步骤
    x = (x ^ _logical_shr(x, 30)) * _C1
    x = (x ^ _logical_shr(x, 27)) * _C2
    x = x ^ _logical_shr(x, 31)
    return x

def _logical_shr(x: torch.Tensor, n: int) -> torch.Tensor:
    # 逻辑右移，用于 PyTorch 有符号整数的无符号移位
    return (x >> n) & ((1 << (64 - n)) - 1)

def _uint64_mod(x: torch.Tensor, mod: int) -> torch.Tensor:
    # 正确处理 PyTorch 有符号 int64 取模，使其行为如同无符号
    offset = (1 << 64) % mod
    base = x % mod
    correction = (x < 0).to(x.dtype) * offset
    return (base + correction) % mod
```

## python/sglang/srt/kv_canary/token_oracle/oracle_manager.py

管理 TokenOracle 实例，提供 fill_expected_inputs 和 sample_next_tokens 两个核心方法，并处理不同 forward mode 下 generalized_req_id 的扩展。

```
from __future__ import annotations
from typing import TYPE_CHECKING

import torch
from sglang.srt.kv_canary.expected_inputs import ExpectedInputs
from sglang.srt.kv_canary.token_oracle.oracle import TokenOracle

if TYPE_CHECKING:
    from sglang.srt.model_executor.forward_batch_info import ForwardBatch

class TokenOracleManager:
    def __init__(self, *, oracle: TokenOracle) -> None:
        self.oracle = oracle

    def fill_expected_inputs(
        self, *, forward_batch: "ForwardBatch", expected_inputs_out: ExpectedInputs
    ) -> None:
        # 从 forward batch 获取当前输入和位置
        positions = forward_batch.positions
        input_ids = forward_batch.input_ids
        num_tokens = int(input_ids.shape[0])
        if num_tokens == 0:
            return
```

```
        # 构建每个 token 对应的广义请求 ID（考虑 bootstrap room 和不同 forward mode 的展开）
        generalized_req_ids = _build_generalized_req_id_per_token(
            forward_batch=forward_batch, num_tokens=num_tokens,
            generalized_req_ids_per_row=select_generalized_req_ids(
                vanilla_req_ids=forward_batch.rids_int,
                bootstrap_room_ids_int=forward_batch.bootstrap_room_ids_int,
            ),
        )

        # extend 模式下实际 input_ids 就是预期 token；decode 模式通过 oracle 确定
        if forward_batch.forward_mode.is_extend():
            expected_tokens = input_ids
        else:
            expected_tokens = self.oracle.expected_tokens(
                generalized_req_ids=generalized_req_ids,
                positions=positions.to(torch.int64),
            )

        # 将预期值和实际位置写入输出缓冲区
        expected_inputs_out.tokens[:num_tokens].copy_(expected_tokens.to(torch.int64))
        expected_inputs_out.positions[:num_tokens].copy_(positions.to(torch.int64))

    def sample_next_tokens(
        self, *, generalized_req_ids: torch.Tensor, logits_positions: torch.Tensor
    ) -> torch.Tensor:
        # 采样下一 token 时位置加 1，因为预测的是下一个位置
        return self.oracle.expected_tokens(
            generalized_req_ids=generalized_req_ids,
            positions=logits_positions.to(torch.int64) + 1,
        )

# 其余辅助函数（_build_generalized_req_id_per_token、_expand_uniform、select_generalized_
req_ids）省略
```

## python/sglang/srt/kv_canary/token_oracle/sampler.py

将 TokenOracleManager 注册为采样后端，替换标准采样逻辑，并集成 next-token-swap 扰动。

```
from __future__ import annotations
from typing import TYPE_CHECKING, List

import torch
from sglang.srt.kv_canary.perturb.next_token_swap import maybe_perturb_swap_next_tokens
from sglang.srt.kv_canary.token_oracle.oracle import TokenOracle
from sglang.srt.kv_canary.token_oracle.oracle_manager import (
    TokenOracleManager, select_generalized_req_ids,
)
from sglang.srt.layers.sampler import Sampler, register_sampler_backend
```

```
if TYPE_CHECKING:
    from sglang.srt.layers.logits_processor import LogitsProcessorOutput
    from sglang.srt.sampling.sampling_batch_info import SamplingBatchInfo

def install_oracle_sampler(*, oracle: TokenOracle) -> TokenOracleManager:
    """注册 token_oracle 采样后端并返回 manager 供外部使用。"""
    manager = TokenOracleManager(oracle=oracle)
    register_sampler_backend(
        "token_oracle",
        lambda: _OracleSampler(token_oracle_manager=manager),
    )
    return manager

class _OracleSampler(Sampler):
    def __init__(self, *, token_oracle_manager: TokenOracleManager) -> None:
        super().__init__()
        self._token_oracle_manager = token_oracle_manager

    def forward(
        self,
        logits_output: "LogitsProcessorOutput",
        sampling_info: "SamplingBatchInfo",
        return_logprob: bool,
        top_logprobs_nums: List[int],
        token_ids_logprobs: List[List[int]],
        positions: torch.Tensor,
    ) -> torch.Tensor:
        # 使用 sampling_info 的 rids_int 作为 generalized_req_id 来源
        vanilla_req_ids = sampling_info.rids_int
        if vanilla_req_ids is None:
            raise RuntimeError(
                "_OracleSampler.forward: generalized_req_id source tensor is None; "
                "token oracle requires a per-forward generalized_req_id source tensor "
                "(set in ForwardBatch.init_new when SGLANG_KV_CANARY_ENABLE_TOKEN_
                ORACLE=1)"
            )
        # 通过 oracle manager 获取下一 token（基于当前位置 +1）
        batch_next_token_ids = self._token_oracle_manager.sample_next_tokens(
            generalized_req_ids=select_generalized_req_ids(
                vanilla_req_ids=vanilla_req_ids,
                bootstrap_room_ids_int=sampling_info.bootstrap_room_ids_int,
            ),
            logits_positions=positions,
        )

        # 可选扰动：交换两个请求的 token 以验证断言链路
        batch_next_token_ids = maybe_perturb_swap_next_tokens(batch_next_token_ids)
        return batch_next_token_ids
```

## 评论区精华

- 暂无高价值评论线程

## 风险与影响

- 风险：所有新功能均默认关闭，对现有推理路径无影响。启用时风险包括：1）确定性哈希算法（SplitMix64）的正确性依赖，如果存在未发现的 bug 可能导致错误 passed；2）采样后端替换可能与其他自定义采样器冲突；3）write-input 断言增加了每次前向传播的额外计算开销（但仅比较张量，开销很低）；4）perturb 扰动仅用于测试，若误在生产环境启用可能引入随机错误。
- 影响：对用户无感知，所有功能默认关闭。对系统增加了 kv-canary 的可观测性能力，使开发者能够端到端验证 KV 缓存写入的正确性。对团队提供了健壮的测试手段（perturb）来证明断言机制有效。影响范围限于 kv-canary 子系统，不涉及核心推理路径。
- 风险标记：默认关闭无影响，确定性哈希算法正确性依赖，采样后端替换可能冲突，Perturb 仅用于测试

## 关联脉络

- PR #26816 [kv-canary] Add the KV-canary perturb framework for fault-injection self-tests: 本 PR 的 next-token-swap 扰动构建在 #26816 引入的 perturb 框架之上。