# PR #19545 完整报告

sgl-project/sglang

feat(observability): add OpenTelemetry tracing for speculative decoding

合并时间：2026-04-17 14:01

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/19545

## 执行摘要

- 一句话：为推测解码管道添加 OpenTelemetry 追踪，覆盖 EAGLE 和 NGRAM 工作器的 draft、verify 和 accept 阶段。
- 推荐动作：建议技术管理者和工程师精读此 PR，特别是 req_time_stats.py 中新增的追踪方法设计和 set_time_batch 的使用模式，这些展示了如何将 OpenTelemetry 集成到高性能推理管道中，同时保持低开销。关注设计决策如 trace_only 参数和事件放置时机，对于构建可观测性功能有借鉴价值。

## 功能与动机

根据 PR body 和关联 Issue #13511，推测解码的各个阶段目前没有 instrumentation，使得诊断性能瓶颈变得困难。这是追踪路线图（#13511）的一部分，旨在填补这一空白，提升系统的可观测性。

## 实现拆解

1. 扩展追踪配置：在 python/sglang/srt/observability/req_time_stats.py 中，新增 RequestStage 枚举值 SPEC_DRAFT（level 2）、SPEC_VERIFY（level 2）和 SPEC_DRAFT_EXTEND（level 3），并添加对应的追踪方法（如 set_spec_draft_start_time），这些方法使用 time.perf_counter() 记录时间戳并调用 trace_slice 生成追踪片段。
2. 集成到 EAGLE 工作器：在 python/sglang/srt/speculative/eagle_worker.py 中导入 set_time_batch 和 get_global_tracing_enabled，在 forward_batch_generation 方法的 draft、verify 和 draft_extend 阶段调用 set_time_batch（例如 set_time_batch(batch.reqs, "set_spec_draft_start_time", trace_only=True)），并通过循环为每个请求设置 accepted_tokens 事件。
3. 集成到 NGRAM 工作器：在 python/sglang/srt/speculative/ngram_worker.py 中类似地导入并集成追踪逻辑，在 forward_batch_generation 方法的关键点添加 set_time_batch 调用和事件处理。
4. 确保零开销设计：所有追踪逻辑都通过 get_global_tracing_enabled() 检查，并在 set_time_batch 中新增 trace_only 参数，当追踪禁用时快速返回，避免对核心路径的性能影响。

5. 无测试或配置配套改动：本次变更专注于源码级集成，没有修改测试文件或部署配置，但需确保与现有追踪系统兼容。

关键文件：

- python/sglang/srt/observability/req_time_stats.py（模块 可观测性；类别 source；类型 core-logic；符号 set_spec_draft_start_time, set_spec_draft_end_time, set_spec_verify_start_time, set_spec_verify_end_time）：核心追踪逻辑变更，新增 speculative decode 的配置和追踪方法，是 PR 的基石。
- python/sglang/srt/speculative/eagle_worker.py（模块 推测解码；类别 source；类型 dependency-wiring）：EAGLE 工作器集成追踪，在 draft、verify 和 draft_extend 阶段添加 span，是功能的关键集成点。
- python/sglang/srt/speculative/ngram_worker.py（模块 推测解码；类别 source；类型 dependency-wiring）：NGRAM 工作器集成追踪，类似 EAGLE 工作器，确保两种推测解码算法都有完整的追踪支持。

关键符号：set_spec_draft_start_time, set_spec_draft_end_time, set_spec_verify_start_time, set_spec_verify_end_time, set_spec_draft_extend_start_time, set_spec_draft_extend_end_time, set_time_batch

## 关键源码片段

### python/sglang/srt/observability/req_time_stats.py

核心追踪逻辑变更，新增 speculative decode 的配置和追踪方法，是 PR 的基石。

```
# 在RequestStage枚举中新增speculative decode的追踪配置
SPEC_DRAFT = RequestStageConfig(
    "spec_draft",
    level=2, # 追踪级别2，用于细粒度性能分析
)

SPEC_VERIFY = RequestStageConfig(
    "spec_verify",
    level=2,
)

SPEC_DRAFT_EXTEND = RequestStageConfig(
    "spec_draft_extend",
    level=3, # 级别3，提供更详细的追踪信息
)

# 新增的追踪方法：记录draft阶段开始时间
class SchedulerReqTimeStats(ReqTimeStatsBase):
    # 添加字段存储时间戳
    spec_draft_start_time: float = 0.0
    spec_verify_start_time: float = 0.0
    spec_draft_extend_start_time: float = 0.0

    def set_spec_draft_start_time(self, ts=None):
```

```
        if ts is None:
            ts = time.perf_counter() # 使用高精度计时器获取当前时间
        self.spec_draft_start_time = ts # 存储开始时间戳

    def set_spec_draft_end_time(self, ts=None):
        if ts is None:
            ts = time.perf_counter()
        stage = RequestStage.SPEC_DRAFT
        # 调用trace_slice生成追踪片段，记录从开始到结束的时间区间
        self.trace_slice(stage, self.spec_draft_start_time, ts)

    def set_spec_verify_end_time(self, ts=None, accepted_tokens: int = 0):
        if ts is None:
            ts = time.perf_counter()
        stage = RequestStage.SPEC_VERIFY
        # 在追踪片段中添加accepted_tokens属性，记录验证阶段接受的令牌数
        self.trace_slice(stage, self.spec_verify_start_time, ts, {"accepted_tokens": accepted_
        tokens})

# 扩展set_time_batch函数以支持trace_only参数
# 当trace_only为True且全局追踪禁用时，快速返回以避免开销
def set_time_batch(reqs: List[Any], set_func: str, trace_only: bool = False):
    if reqs is None or len(reqs) == 0:
        return
    if trace_only and not get_global_tracing_enabled():
        return # 仅用于追踪时，检查全局开关
    ts = time.perf_counter()
    for req in reqs:
        method = getattr(req.time_stats, set_func) # 动态调用对应的set_*_time方法
        method(ts)
```

# 评论区精华

review 中，sufeng-buaa 提出关键建议：

- 设计统一化：建议使用现有的 set_time_batch 模式来统一时间设置，并添加 trace_only 参数以优化性能（"You can add a default bool parameter trace_only to set_time_batch"）。
- 事件放置正确性：指出事件应放在 trace 结束前，否则可能丢失（"The event should be placed before the trace end"）。
- 代码质量：提醒移除废弃代码行以通过 lint 检查（"Sorry, this deprecated line needs to be removed"）。 作者接受了所有建议，在后续提交中应用补丁并重构代码，最终获得批准。
- 使用 set_time_batch 模式统一时间设置 (design): 作者采纳建议，在后续提交中重构代码，使用 set_time_batch 并添加 trace_only 参数。
- 事件放置时机以避免数据丢失 (correctness): 作者调整代码，确保事件在 trace 结束前记录。
- 移除废弃代码以通过 lint 检查 (style): 作者移除废弃行，保持代码整洁。

## 风险与影响

- 风险：性能风险：追踪逻辑可能增加 CPU 开销，但通过 get_global_tracing_enabled() 门控和 trace_only 参数在禁用时避免额外成本，最小化影响。 正确性风险：新增的 set_spec_*_time 方法在 req_time_stats.py 中需确保时间戳处理正确，特别是 trace_slice 调用可能因参数传递错误导致追踪数据不准确。 兼容性风险：与现有追踪模式（如 TraceReqContext）集成需一致，但基于现有设计降低了风险。 测试覆盖不足：没有直接对应的测试变更，可能隐藏回归 bug，需依赖现有测试套件验证。
- 影响：用户影响：启用 OpenTelemetry 追踪后，用户可以获得推测解码阶段的详细时间线，帮助诊断性能瓶颈，提升调试效率。 系统影响：在追踪启用时增加少量 CPU 开销（主要来自时间戳记录和 span 管理），但设计上避免了主要推理路径的负担；追踪数据可通过现有渠道导出，增强系统可观测性。 团队影响：工程师能更轻松地分析推测解码性能，支持优化工作，同时展示了如何在高性能系统中优雅集成追踪功能。
- 风险标记：性能开销门控，核心路径变更，缺少测试覆盖

## 关联脉络

- 暂无明显关联 PR