

PR #3149 完整报告

radixark/miles

ci: repeat a failed test's last output in the suite summary

合并时间: 2026-09-10 02:28

原文链接: <http://prhub.com.cn/radixark/miles/pull/3149>

执行摘要

- 一句话: 在 CI 测试套件失败摘要中重复显示失败测试的最后输出, 提升错误诊断效率。
- 推荐动作: 本 PR 是一个高质量、有明确问题驱动的基础设施改进。它通过精确的量化分析定义了问题, 并通过设计精巧、内存受控的缓冲区机制和全面的单元测试 (包括使用真实数据回放) 来解决问题。虽然它不涉及核心业务逻辑或架构变更, 但其对 CI 可观测性和开发效率的提升是显著且直接的。建议精读其实现, 特别是 `_drain_child_output` 和 `_failure_tail` 的流式处理与缓冲区管理逻辑, 以及测试中模拟大日志和验证通知器窗口的技巧。

功能与动机

根据 PR body 描述, 问题根源是 GPU 测试套件在某个测试失败后仍继续运行, 产生大量输出 (例如日志大小达 153MB), 导致失败测试的 traceback 距离日志末尾达 56.6MB。这使得无论是手动查看日志末尾还是 CI Lark 通知器, 都只能看到失败测试的文件名和退出码 (如 "exit code 255"), 无法直接获取失败原因。为了改善这一开发者体验和自动化诊断效率, 需要确保失败测试的关键输出信息在日志摘要中可见。

实现拆解

1. 核心逻辑引入: 在 `tests/ci/ci_utils.py` 中新增 `_drain_child_output` 函数和 `_failure_tail` 函数。前者以 64KB 为单位读取子进程的 stdout 管道, 将数据直通写入 runner 的 stdout, 同时将数据块追加到一个最大长度为 `FAILURE_TAIL_LINES * 8` 的 deque 缓冲区中, 并通过滑动窗口机制确保缓冲区总大小不超过 `FAILURE_TAIL_BYTES` (8KB)。后者将缓冲区中的字节块拼接、截取最后 8KB, 解码为文本并过滤空行, 最后返回最后 60 行 (`FAILURE_TAIL_LINES`)。
2. 运行器集成: 在 `run_unittest_files` 函数中, 为每次文件运行初始化一个新的 `output_tail` 缓冲区。在 `run_one_file` 的两种模式 (`capture_output=True/False`) 中, 都将子进程的输出导向 `subprocess.PIPE`。对于 `capture_output=False` 的场景 (CUDA 阶段), 调用 `_drain_child_output` 进行流式处理; 对于 `capture_output=True` 的场景, 在逐行日志记录的同时, 也将每一行编码后追加到 `output_tail`。
3. 摘要输出增强: 修改 `failed_tests` 列表的存储格式, 从 `(filename, reason)` 扩展为 `(filename, reason, tail)`, 其中 `tail` 是通过 `_failure_tail(output_tail)` 生成的失败测试尾部摘要。在最终打印 FAILED: 摘要时, 遍历 `failed_tests` 列表, 为每个失败测试额外打印其尾部摘要内容。

4. 测试覆盖：新增 tests/ci/test/test_failure_tail.py 文件，包含 5 个关键测试用例：验证 drain 函数能无损传递所有字节；验证大流（40MB）下缓冲区大小被限制（<1MB）；验证 _failure_tail 能正确保留最后若干行并去除空行；通过一个实际失败的测试用例验证摘要内容包含错误信息且长度小于通知器窗口（33.6KB）；验证在远超通知器窗口的超大日志（40MB+）下，摘要仍能捕获错误信息。

关键文件：

- tests/ci/ci_utils.py（模块 测试运行器；类别 test；类型 core-logic；符号 _drain_child_output, _failure_tail, run_unittest_files, FAILURE_TAIL_BYTES）：核心实现文件。引入了失败测试输出尾部捕获（_drain_child_output, _failure_tail）和摘要打印逻辑，并修改了 run_unittest_files 函数以集成此功能。
- tests/ci/test/test_failure_tail.py（模块 失败尾部测试；类别 test；类型 test-coverage；符号 _CapturedStdout, drain, test_draining_passes_every_byte_through_untouched, test_draining_holds_a_bounded_tail_of_a_large_stream）：新增的测试文件，为失败测试输出捕获功能提供了全面、多角度的单元测试覆盖，是验证该功能正确性的关键。

关键符号：_drain_child_output, _failure_tail, run_unittest_files

关键源码片段

tests/ci/ci_utils.py

核心实现文件。引入了失败测试输出尾部捕获（_drain_child_output, _failure_tail）和摘要打印逻辑，并修改了 run_unittest_files 函数以集成此功能。

在 tests/ci/ci_utils.py 中引入的两个核心函数，用于捕获和格式化失败测试的最后输出。

FAILURE_TAIL_BYTES = 8_192 # 保留最后 8KB 的字节数据

FAILURE_TAIL_LINES = 60 # 最终显示的摘要行数上限

```
def _drain_child_output(stream, tail: deque) -> None:
```

```
    """Pass a child's output straight through, keeping only its last bytes for the failure summary.
    """
```

```
    held = 0 # 当前 deque 中保留的字节数
```

```
    # 以 64KB 为单位读取子进程的 stdout 管道，避免逐行处理的性能开销。
```

```
    while chunk := stream.read(65_536):
```

```
        # 将数据块直通写入 runner 自身的 stdout，保持日志流完整。
```

```
        sys.stdout.buffer.write(chunk)
```

```
        sys.stdout.buffer.flush()
```

```
        # 追加到用于保留尾部的 deque 缓冲区。
```

```
        tail.append(chunk)
```

```
        held += len(chunk)
```

```
        # 通过滑动窗口机制，确保缓冲区总大小不超过 FAILURE_TAIL_BYTES。
```

```
        while tail and held - len(tail[0]) >= FAILURE_TAIL_BYTES:
```

```
            held -= len(tail.popleft())
```

```
def _failure_tail(chunks: deque) -> str:
```

```
    # 将 deque 中的字节块拼接，取最后 8KB，解码为文本。
```

```

text = b"".join(chunks)[-FAILURE_TAIL_BYTES:].decode("utf-8", errors="replace")
# 过滤掉空白行。
lines = [line for line in text.splitlines() if line.strip()]
# 返回最后 FAILURE_TAIL_LINES (60) 行。
return "\n".join(lines[-FAILURE_TAIL_LINES:])

```

tests/ci/test/test_failure_tail.py

新增的测试文件，为失败测试输出捕获功能提供了全面、多角度的单元测试覆盖，是验证该功能正确性的关键。

tests/ci/test/test_failure_tail.py 中的关键测试函数，模拟真实场景验证功能。

常量，与 ci_failure_analysis.py 中下载的日志窗口大小一致。

NOTIFIER_TAIL_BYTES = 33_600

```

def test_a_failing_test_explains_itself_inside_the_notifier_window(tmp_path, monkeypatch,
caplog):
    """验证一个失败的测试能在通知器窗口内解释自身。"""
    # 创建一个会打印大量噪音然后失败的测试脚本。
    Path(tmp_path / "test_boom.py").write_text(
        "for index in range(5000):\n"
        "    print(f'noise {index}')\n"
        "raise AssertionError('the reactor exploded at step 42')\n"
    )
    monkeypatch.chdir(tmp_path)
    with caplog.at_level(logging.INFO):
        # 运行该测试，并启用 continue_on_error 以获取完整日志。
        run_unittest_files([CITestFile(name="test_boom.py")], timeout_per_file=120, continue_on_
error=True)
    # 断言日志中包含失败信息。
    assert "FAILED: test_boom.py returned exit code 1" in caplog.text
    # 提取 "FAILED:" 标记后的摘要部分。
    _, separator, summary = caplog.text.rpartition("\nFAILED:\n")
    assert separator, "the runner printed no failure summary"
    # 断言摘要中包含具体的错误信息。
    assert "the reactor exploded at step 42" in summary
    # 断言摘要总长度小于通知器窗口，确保能被捕获。
    assert len(summary) < NOTIFIER_TAIL_BYTES

def test_the_summary_survives_a_log_far_larger_than_the_notifier_window():
    """验证在超大日志下，摘要仍能捕获错误信息。"""
    # 生成约 40MB 的 "progress" 行，然后附加一个错误。
    body = b"".join(f"progress step {index}\n".encode() for index in range(400_000))
    body += b"AssertionError: the reactor exploded\n"
    _, tail = drain(body)
    rendered = _failure_tail(tail)
    # 断言错误信息被捕获在渲染后的摘要中。

```

```
assert "AssertionError: the reactor exploded" in rendered
# 断言渲染后的摘要长度不超过 FAILURE_TAIL_BYTES (8KB)。
assert len(rendered.encode()) <= FAILURE_TAIL_BYTES
# 断言原始日志体远大于通知器窗口。
assert len(body) > 100 * NOTIFIER_TAIL_BYTES
```

评论区精华

PR未引发技术讨论，仅有自动化机器人（[claude\[bot\]](#)）的审查提示和人工审查者（[guapisolo](#)）的批准。讨论重点在于 PR body 中清晰的问题陈述、量化分析（日志大小与 traceback 距离）以及通过详细单元测试（包括使用真实夜间构建日志进行回放验证）进行的方案验证。

- CI 测试失败后日志可读性 (design): 方案被接受并合并。通过在 `ci_utils.py` 中实现固定大小的输出尾部缓冲区（8KB）和摘要打印，并辅以全面的单元测试（包括真实日志回放），解决了问题。无公开讨论。

风险与影响

- 风险：性能影响：变更将原本不捕获输出（`stdout=None, stderr=None`）的非重试模式改为通过 `subprocess.PIPE` 捕获输出，并进行流式处理。尽管使用了 64KB 的分块读取（避免了逐行处理的 Python 和日志开销），但仍在进程间增加了管道通信。对于输出量极大的 GPU 测试套件（如 153MB 日志），这可能会略微增加 I/O 开销，但缓冲区管理（固定大小的 `deque`）确保了内存使用是可控的。正确性风险：`_failure_tail` 函数在解码时使用了 `errors="replace"`，这对于处理可能包含非 UTF-8 字节的测试输出是安全的。缓冲区重置逻辑（在 `run_one_file` 顶部）确保了每个测试文件的输出是独立的，避免了信息混杂。唯一未完全验证的场景是：一个真实的 GPU 硬件故障在集成环境下的完整链路（失败 -> 通知器），但通过日志形状回放测试已覆盖了核心逻辑。兼容性：变更修改了 `failed_tests` 列表的内部数据结构（从二元组变为三元组），影响范围被严格限制在 `run_unittest_files` 函数内部及其日志打印逻辑，不影响外部接口。
- 影响：开发者体验：显著改善。CI 日志尾部现在直接包含失败测试的关键错误信息（如断言错误），无需再滚动数百兆字节日志。自动化工具：CI Lark 通知器（`ci_failure_analysis.py`）的 `extract_log_evidence` 函数现在有更大概率在 33.6KB 的日志尾部窗口内提取到决定性的错误行，从而能提供更准确的失败原因分析。CI 可靠性：通过更早、更清晰地暴露失败原因，有助于加快问题定位和修复速度。新增的单元测试增强了 CI 基础设施代码本身的可靠性。资源使用：引入了固定的额外内存开销（每个测试运行 8KB 缓冲区），并可能略微增加 I/O 延迟（由于管道重定向），但这是为了换取更好的可诊断性而进行的权衡，被认为是可接受的。
- 风险标记：非捕获模式性能影响，依赖日志格式

关联脉络

- PR #3152 ci(lark): explain each failed job on the CI status card: 本 PR (3149) 直接改善了 CI Lark 通知器（`ci_failure_analysis.py`）的输入数据质量。通过确保失败测试的 traceback 更靠近日志末尾，通知器的 `extract_log_evidence` 函数有更高概率在固定的尾部窗口（如 33.6KB）内提取到决定性的错误信息，从而提供更准确的失败原因分析。两者共

同构成了 CI 失败诊断能力的提升链路。