PR #40151 完整报告

vllm-project/vllm

[compile] Skip FX graph deserialiaztion on loading, further reducing warm compile time.

合并时间：2026-04-23 13:43

原文链接：http://prhub.com.cn/vllm-project/vllm/pull/40151

执行摘要

- 一句话：通过跳过 FX 图反序列化，将热编译时间降低至亚 2 秒级别。
- 推荐动作：建议技术管理者和工程师精读此 PR，重点关注 generate_execution_code_with_name 的设计决策，以及缓存反序列化的跳过逻辑。这些变更展示了如何通过代码生成优化编译性能，值得学习。

功能与动机

动机是进一步减少热编译时间，以提升模型加载和推理效率。PR body 中引用测试数据，显示热编译时间从数秒降至亚秒级别，例如 DeepSeek-V3.2 从 6.05 秒降至 0.27 秒（-95.5%）。作者指出这是跟进 PR #38657，利用最近添加的 Python 执行代码作为源，避免 FX 图序列化开销。

实现拆解

1. 代码生成逻辑重构：在 vllm/compilation/codegen.py 中，新增 generate_execution_code_with_name 函数，支持内联 torch.fx.GraphModule 子模块到生成的 Python 代码中，从而避免序列化整个图。关键符号包括 generate_execution_code_with_name 和 inlined_submods 列表。
2. 缓存和反序列化调整：在 vllm/compilation/caching.py 中，修改 VllmSerializableFunction 类的 __init__ 方法，允许 graph_module 参数为 bytes 类型，并在 deserialize_compile_artifacts 方法中跳过图的反序列化步骤，直接使用执行代码。添加 fake_mode 参数以支持回退路径。
3. 导入和依赖更新：在 vllm/compilation/backends.py 中，调整导入语句，从 codegen 模块直接导入 compile_execution_fn 和 generate_execution_code，简化代码结构。
4. 测试配套：本次变更未包含直接测试文件，但 PR body 提供了详细的性能测试数据，验证了热编译时间的改进。

关键文件：

- vllm/compilation/codegen.py（模块 编译模块；类别 source；类型 core-logic；符号 generate_execution_code, generate_execution_code_with_name）：核心变更文件，实现了代码生成逻辑的重构，支持内联子模块以避免 FX 图序列化。
- vllm/compilation/caching.py（模块 编译模块；类别 source；类型 data-contract；符号 VllmSerializableFunction）：修改了缓存反序列化逻辑，允许跳过 FX 图反序列化，直接使

用执行代码，减少加载时间。

- vllm/compilation/backends.py（模块 编译模块；类别 source；类型 dependency-wiring）：调整导入语句，简化代码结构，确保从 codegen 模块正确导入函数。

关键符号：generate_execution_code_with_name, VllmSerializableFunction.init, deserialize_compile_artifacts

关键源码片段

vllm/compilation/caching.py

修改了缓存反序列化逻辑，允许跳过 FX 图反序列化，直接使用执行代码，减少加载时间。

```
class VllmSerializableFunction(SerializableCallable):
    def __init__(
        self,
        graph_module: torch.fx.GraphModule | bytes, # 现在允许 bytes 类型，避免反序列化
        example_inputs: Sequence[Any],
        prefix: str,
        optimized_call: Callable[..., Any],
        is_encoder: bool = False,
        vllm_backend: Any | None = None,
        sym_tensor_indices: list[int] | None = None,
        aot_autograd_config: dict[str, Any] | None = None,
        execution_code: str | None = None,
        submod_names: list[str] | None = None,
    ) -> None:
        self.graph_module = graph_module # 不再断言为 GraphModule，支持直接传递 bytes
        self.example_inputs = example_inputs
        self.prefix = prefix
        self.optimized_call = optimized_call
        self.is_encoder = is_encoder
        self.shape_env = None
        self.vllm_backend = vllm_backend
        self.sym_tensor_indices = sym_tensor_indices
        self.execution_code = execution_code # 存储生成的执行代码
        self.submod_names = submod_names
        self._fake_mode: Any | None = None
        # 其他初始化逻辑 ...

    @classmethod
    def deserialize_compile_artifacts(cls, data: bytes) -> "VllmSerializableFunction":
        from torch._guards import TracingContext, tracing
        from torch.fx.experimental.symbolic_shapes import ShapeEnv

        state = pickle.loads(data)
        fake_mode = FakeTensorMode(shape_env=ShapeEnv())

        # 跳过 graph_module 的反序列化，直接加载 example_inputs
        state["example_inputs"] = GraphPickler.loads(state["example_inputs"], fake_mode)
```

```
        standalone_compile_artifacts = state.pop("standalone_compile_artifacts", None)
        sym_shape_indices_map = state.pop("sym_shape_indices_map", {})
        returns_tuple_map = state.pop("returns_tuple_map", {})

        # 使用执行代码重构函数，避免反序列化整个图
        if envs.VLLM_USE_MEGA_AOT_ARTIFACT:
            assert standalone_compile_artifacts is not None
            submod_names = state.get("submod_names")
            # 重构逻辑 ...
        # 回退路径：仅在需要时反序列化 graph_module
        state["graph_module"] = cls.deserialize_graph_module(state["graph_module"], fake_mode)
        state["graph_module"].recompile()
        # 其他逻辑 ...
```

评论区精华

review 中主要讨论了三个问题：

- 向后兼容性风险：gemini-code-assist[bot] 指出 state["submod_names"] 在旧缓存中可能缺失，导致 KeyError。zhxchen17 回应说缓存加载失败时会生成新缓存，因此保持当前行为，并添加注释解释。
- 导入缺失：gemini-code-assist[bot] 建议在生成代码中添加 import operator 以支持函数调用，但未在代码中直接采纳，可能已通过其他方式解决。
- 子模块绑定方式：gemini-code-assist[bot] 建议使用直接字典访问而非 .get() 以提供更清晰的错误信息，但 zhxchen17 解释使用 .get() 是故意的，用于处理内联子模块的占位符。
- 向后兼容性风险：submod_names 键缺失 (correctness): 决定不修改代码，添加注释解释行为，依赖缓存再生机制。
- 生成代码导入缺失：operator 模块 (correctness): 未在代码中直接采纳，可能已通过其他方式解决，但风险仍需关注。
- 子模块绑定方式：使用 .get() vs 直接访问 (design): 保持使用 .get()，并添加注释说明意图。

风险与影响

- 风险：技术风险包括：
  - 向后兼容性：旧缓存可能因缺少 submod_names 键而加载失败，但设计上会触发新缓存生成，风险可控。
  - 运行时错误：如果生成代码中缺少必要导入（如 operator），可能导致 NameError，但 review 中未显示修复，需关注。
  - 逻辑复杂性：内联子模块和跳过反序列化增加了代码复杂度，可能引入隐蔽的 bug，尤其是在边缘情况下。
- 影响：影响范围：
  - 用户：热编译时间大幅减少，提升模型启动速度，尤其对频繁加载模型的场景有益。
  - 系统：编译模块的核心路径变更，可能影响所有使用 vLLM 编译功能的模型推理。
  - 团队：代码结构简化，但需确保缓存兼容性和测试覆盖。

- 风险标记：向后兼容性风险，运行时导入缺失，核心路径变更

关联脉络

- PR #38657 [compile] Follow-up PR for Python *execution code optimization*: 本 PR 是跟进 PR #38657，利用其引入的 Python 执行代码作为真相来源，进一步优化编译时间。