大模型推理基础设施

第 3 讲一次推理请求的完整链路

张书豪

华中科技大学计算机学院

研究生课程课件

教学目标

核心结论

本讲分解一次生成请求的完整系统链路，建立面向时序与阶段的分析视角。

- 理解 API 到流式返回的关键阶段。
- 区分 prefill 与 decode 的资源特性。
- 找到排队点、状态生成点与资源回收点。

请求生命周期总览

核心结论

一次请求通常要经历接入、规范化、排队、执行、返回和回收六个阶段。

1. API 接入与参数校验
2. tokenizer 与输入规范化
3. 请求排队与调度
4. prefill 生成首轮 KV
5. decode 迭代输出 token
6. 流式返回与资源回收

为什么生命周期视角是基础

核心结论

只有先把请求穿过系统的完整路径画清楚，后续任何优化才知道自己在改哪一段。

- 生命周期视角帮助区分排队问题和执行问题。
- 生命周期视角帮助区分计算状态和长期状态对象。
- 生命周期视角帮助解释 TTFT 和 TPOT 为什么受不同阶段支配。

prefill 与 decode

核心结论

prefill 和 decode 在资源需求、控制目标和瓶颈形式上显著不同。

- prefill 更计算密集，直接决定首 token 延迟。
- decode 单轮较轻，但迭代频繁，更依赖状态驻留。
- 把二者混为同一任务，容易造成错误调度与资源竞争。

prefill 为什么敏感

核心结论

prefill 直接决定首 token 何时出现，因此它既是计算问题，也是用户体验问题。

- 长 prompt 会显著放大 prefill 开销。
- RAG 或工具注入会进一步增加 prompt 长度和波动。
- prefill 常常也是资源切分和阶段解耦的起点。

decode 为什么难管

核心结论

decode 每轮算得不重，但它持续依赖状态驻留和低抖动调度，因此更容易受系统噪声影响。

- 每轮都依赖已有 KV 和 batch 组织。
- 并发请求越多，轮次间调度和状态竞争越复杂。
- decode 是服务稳态控制的核心阶段。

系统中的潜在瓶颈点

核心结论

生命周期视角的价值，在于把不同阶段的瓶颈显式区分开。

- 队列等待与队头阻塞。
- prefill 对长 prompt 的容量和算力压力。
- decode 的状态驻留和频繁调度开销。
- 输出侧流式接口和回收延迟。

状态对象沿着哪条链路传播

核心结论

一次请求在生命周期中会持续生成和消耗多类状态对象。

- tokenizer 生成规范化输入状态。
- prefill 生成首轮 KV。
- decode 追加 KV 并消费队列与调度元数据。
- 返回阶段触发日志、统计和资源回收。

源码穿插：在 nano-vllm-hust 中跟踪请求路径

核心结论

nano-vllm-hust 保留了从请求进入到全部完成的完整主路径，足够支撑本讲的生命周期分析。

- nanovllm/engine/llm_engine.py 中的 add_request() 对应请求进入系统。
- 同一文件中的 step() 串联 schedule()、模型执行和 postprocess()。
- generate() 展示了从批量请求注入到全部完成的完整外层驱动循环。

源码穿插：课堂观察重点

核心结论

沿着 LLMEngine 的主路径阅读源码，可以将本讲的生命周期划分与真实实现逐段对应起来。

- **请求进入点**：LLMEngine.add_request() 中的 tokenizer 编码与 Sequence 构造。
- **阶段切换点**：step() 中 schedule() 返回的 is_prefill 标志。
- **完成与回收点**：postprocess() 更新状态并在完成时输出结果。

为什么生命周期视角重要

核心结论

很多“优化没生效”的根本原因，是优化点并不位于主导端到端时延的阶段。

- 局部 kernel 更快，可能仍然输给排队放大。
- 共享执行更强，可能仍然输给 KV 驻留边界。
- 需要先定位主导阶段，再定义优化对象。

生命周期分析中的典型误判

核心结论

将 GPU 利用率偏低直接视为主导问题，往往会忽略等待放大与状态竞争的真实影响。

- 设备不满载不一定说明算力不足。
- 可能是请求排队、KV 碎片或重组时机不合适。
- 生命周期分析的作用，就是避免过早跳到算子层结论。

课堂练习

核心结论

课堂练习可围绕请求时序图展开，以强化生命周期分析能力。

- 标出至少 4 个潜在瓶颈点。
- 标出至少 3 类状态对象的生成与释放时刻。
- 讨论哪个阶段最可能支配 TTFT，哪个阶段最可能支配 TPOT。

讲解：为什么生命周期是后续所有机制课的前提

核心结论

如果一次请求的阶段边界没画清楚，调度、KV、执行优化都会变成悬空讨论。

- 生命周期把一个请求拆成进入、规范化、排队、prefill、decode、返回、回收等阶段。
- 不同阶段的资源特征不同，因此瓶颈也不同；TTFT 和 TPOT 往往不是由同一阶段支配。
- 只有先把阶段画出来，后面才知道一个优化到底改了哪里、为什么可能生效。

讲解：prefill 与 decode 为什么必须分开教

核心结论

很多初学者把“生成”当成单一过程，但系统里最重要的区分之一就是 prefill 与 decode。

- prefill 更像一次性大计算，直接决定首 token 何时出现。
- decode 更像持续迭代的稳态过程，更容易受到调度、状态驻留和抖动影响。
- 课程后续很多机制都是围绕这一区分展开的，例如阶段解耦、continuous batching 和状态搬运。

例子：一次多轮对话的真实链路并不止于模型前向（1）

核心结论

请求生命周期要覆盖从输入组织到状态回收的完整过程，而不是只盯住模型执行。

进入前的准备

chat template、tokenizer 和 prompt 拼接本身就会引入时间与长度变化。

例子：一次多轮对话的真实链路并不止于模型前向（2）

核心结论

请求生命周期要覆盖从输入组织到状态回收的完整过程，而不是只盯住模型执行。

进入后的主路径

入队、调度、prefill、decode、输出组装会在不同对象之间多次转移控制权。

例子：一次多轮对话的真实链路并不止于模型前向（3）

核心结论

请求生命周期要覆盖从输入组织到状态回收的完整过程，而不是只盯住模型执行。

完成后的尾部动作

请求结束不代表系统工作结束，状态释放、统计更新和资源回收同样属于生命周期的一部分。

例子：chat template 为什么会影响生命周期分析（1）

核心结论

如果输入在真正进入 engine 前已经发生了长度膨胀，那么后面的阶段观察也会被一并放大。

用户看到的是一条自然语言消息

但系统里常常先被改写为带角色标记、特殊 token 和 generation prompt 的模板串。

例子：chat template 为什么会影响生命周期分析（2）

核心结论

如果输入在真正进入 engine 前已经发生了长度膨胀，那么后面的阶段观察也会被一并放大。

直接后果

prompt_len 变化会影响 prefill 代价、KV 初始规模和后续排队行为。

例子：chat template 为什么会影响生命周期分析（3）

核心结论

如果输入在真正进入 engine 前已经发生了长度膨胀，那么后面的阶段观察也会被一并放大。

教学意义

生命周期不是从“用户文本”开始算，而是从“真正进入系统的 token 序列”开始算。

代码例子：多轮对话在入口处如何被组织（1）

核心结论

生命周期课程里的代码例子要回答：输入是怎样变成请求对象的。

入口片段

```
prompts = [tokenizer.apply_chat_template(...)]
outputs = llm.generate(prompts, sampling_params)
```

代码例子：多轮对话在入口处如何被组织（2）

核心结论

生命周期课程里的代码例子要回答：输入是怎样变成请求对象的。

课堂应追问

‘generate()‘ 接到的是原始文本、token 序列，还是已经带运行时元数据的请求对象？

代码例子：多轮对话在入口处如何被组织（3）

核心结论

生命周期课程里的代码例子要回答：输入是怎样变成请求对象的。

系统含义

入口处的数据形态，决定了后面 state object 该如何承载长度、状态和 block table。

代码例子：Sequence 为什么是生命周期里的关键对象（1）

核心结论

如果不能说清 Sequence 承载了哪些字段，就很难讲明请求是如何跨阶段存活的。

课堂应关注的字段

```
status
num_cached_tokens
block_table
last_block_num_tokens
```

代码例子：Sequence 为什么是生命周期里的关键对象（2）

核心结论

如果不能说清 `Sequence` 承载了哪些字段，就很难讲明请求是如何跨阶段存活的。

对象视角

这些字段把“请求内容、调度状态、KV 元数据”压在了同一个生命周期对象上。

代码例子：Sequence 为什么是生命周期里的关键对象（3）

核心结论

如果不能说清 `Sequence` 承载了哪些字段，就很难讲明请求是如何跨阶段存活的。

结论

一次请求之所以是系统问题，关键就在于它不是一次函数调用，而是一个持续演化的状态对象。

例子：prefill 和 decode 为什么应该拆开看（1）

核心结论

生命周期讲义里最重要的区分，就是 prefill 与 decode 的资源特征不同。

prefill

一次性处理更长输入，计算更密集，也更容易直接抬高 TTFT。

例子：prefill 和 decode 为什么应该拆开看（2）

核心结论

生命周期讲义里最重要的区分，就是 prefill 与 decode 的资源特征不同。

decode

每轮 token 较少，但会跨更多轮持续存在，更容易与调度和状态驻留绑定在一起。

例子：prefill 和 decode 为什么应该拆开看（3）

核心结论

生命周期讲义里最重要的区分，就是 prefill 与 decode 的资源特征不同。

误区

如果把二者都写成“模型生成阶段”，就会把关键的性能来源混在一起。

课堂练习：用时序图复述一次请求（1）

核心结论

课程练习的目的不是画复杂图，而是检验你是否知道控制权何时转移。

练习一

在时序图中标出：请求进入、prefill 开始、decode 开始、输出完成、状态释放五个节点。

课堂练习：用时序图复述一次请求（2）

核心结论

课程练习的目的不是画复杂图，而是检验你是否知道控制权何时转移。

练习二

说明其中哪一个节点最适合插第一条日志，为什么。

课堂练习：用时序图复述一次请求（3）

核心结论

课程练习的目的不是画复杂图，而是检验你是否知道控制权何时转移。

练习三

用一句话回答：为什么生命周期图是后续调度和 KV 讨论的公共前提？

本讲总结

核心结论

理解单个请求如何穿过系统，是分析后续调度、KV 与执行优化问题的基础。

- 请求链路是系统问题的最小观察单元。
- 生命周期分析先于机制设计。

对应 Tutorial

核心结论

本讲对应 Tutorial 2。先独立作答，再对照下一页参考答案。

- 文件：build/tutorials/Tutorial_02_ 请求生命周期与 PrefillDecode.pdf
- 动手运行、日志记录与提交要求统一查看 build/experiments/ 和对应 experiment 源目录下的 assignment_spring-2026/。