从PagedAttention到调度器：vLLM高效推理全链路揭秘@陶源20260105

创作者简介——@陶源

👋大家好，我是陶源

我是来自**ACG混合云部的一名RD实习生**，目前参与模型适配的相关工作，对大模型以及模型推理加速有一定的兴趣，希望能和大家多多交流～

🏷️技能标签：模型训推、vLLM推理加速

> ✅本文主要介绍vLLM，vLLM是高性能大模型推理框架，通过高效KV缓存管理与连续批处理提升吞吐、降低时延，适合高并发在线服务与批量推理场景。

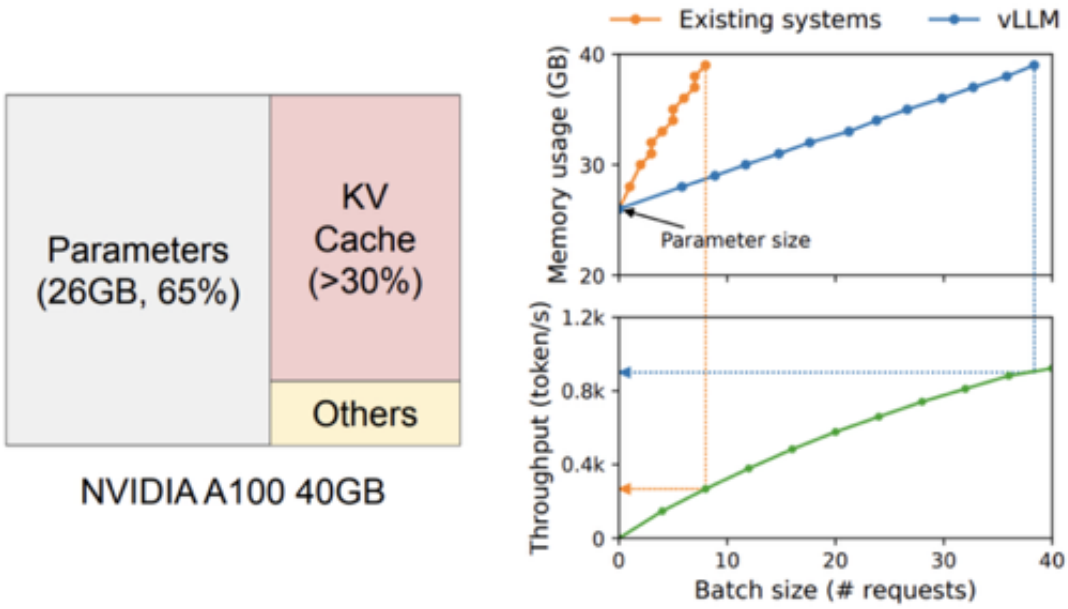
🎁欢迎星标本文、评论区点赞、投币、留言赢好礼 🔍点击此处了解更多AI+社区 🧪社区如流群

本文结构

1. vLLM简介

vLLM 是 UC Berkeley 等开源的一款**高吞吐、内存高效**的大模型推理与服务引擎。它的核心创新在于 **PagedAttention** 算法，将注意力层的 Key-Value 缓存（KV cache）以类似操作系统虚拟内存分页的方式管理。通过这一创新，vLLM 实现了远超传统方案的推理性能。下图展示了一个13B的模型在A100 40GB的gpu上做推理时的显存占用分配，从这张图中我们可以直观感受到推理中KV cache对显存的占用。**因此，如何优化KV cache，节省显存，提高推理吞吐量，就成了LLM推理框架需要解决的重点问题**。

> ℹ️ 本次分享针对vLLM v0进行，vLLM v1针对vLLM v0存在的一些问题进行了改进，分享过程中对部分内容会稍作提及，后续有机会也会为大家带来更详细的vLLM v1的分享～



vLLM 让大模型推理服务变得更“便宜”和“高效”

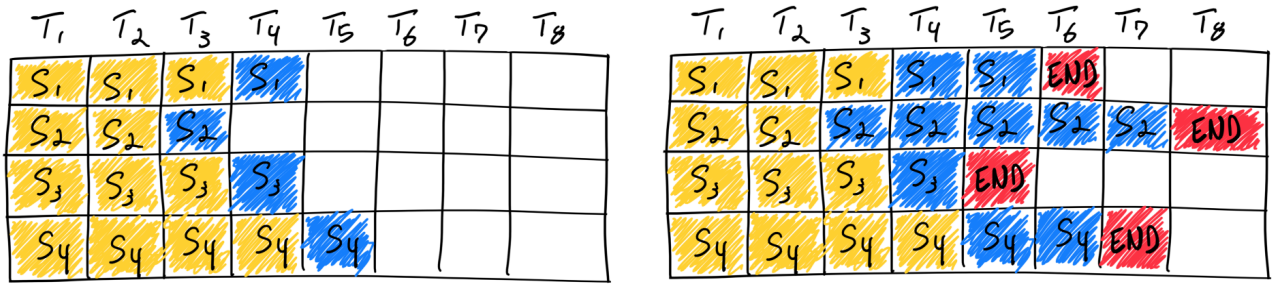
2. vLLM高效的关键

vLLM 的架构围绕提高 GPU 利用率和内存利用率而设计。其关键包括高效的推理流水线（**连续批处理调度**）和创新的内存管理策略（**PagedAttention**）。

2.1. 静态批处理与连续批处理

传统推理框架常采用**静态批处理**：

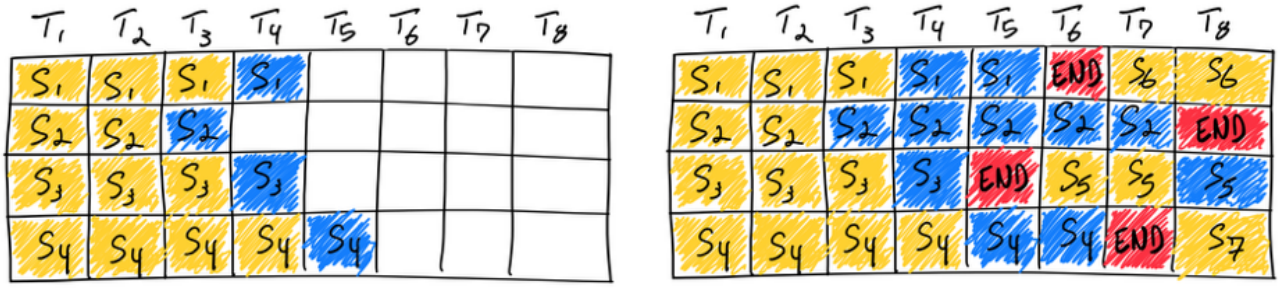
- **工作方式**：收集多个请求 -> 组成一个批次 -> 一次性输入模型 -> 所有请求同时计算 -> 所有请求完成后，释放整个批次，再处理下一批。
- **致命缺陷**：**木桶效应**。在生成任务中，每个请求的输入长度和输出长度（即需要生成的token数）都不同。一个请求可能生成10个token就结束了，而另一个需要100个token。快的请求必须等待慢的请求全部完成后，整个批次才能解散，导致GPU计算资源在每一步都有大量闲置。



静态批处理示意图

连续批处理调度在每个生成迭代（每产生一个新 token）的粒度上动态管理批次：

- 每次迭代时，调度器将当前所有未完成的请求组成批次，在 GPU 上并行推理一个 token。
- **只要某些请求完成了当前 token 的生成，调度器立即回收其占用的计算槽和显存页，下一迭代就可立即将新到达的请求补充进来，不用等待其他请求完成。**
- 通过上述机制，vLLM 的推理流水线能够持续地接纳新请求并生成输出。GPU 几乎始终保持忙碌状态，减少空闲和等待时间，大幅提升了整体吞吐量。



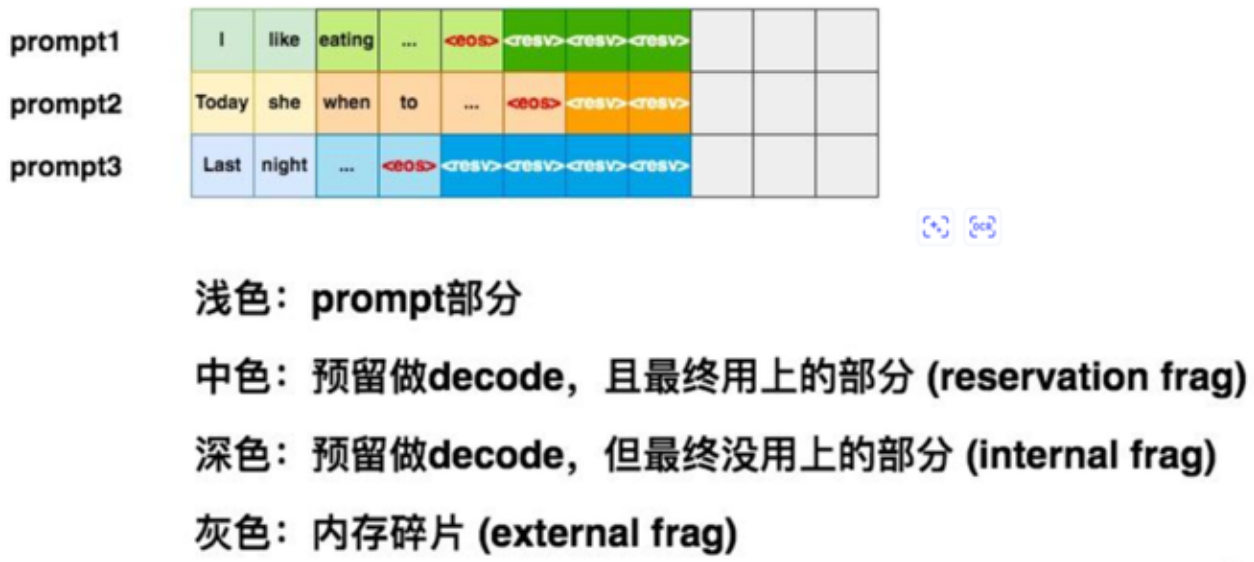
连续批处理示意图

2.2. 内存管理策略：PagedAttention

2.2.1 为KV cache分配存储空间的常规方式

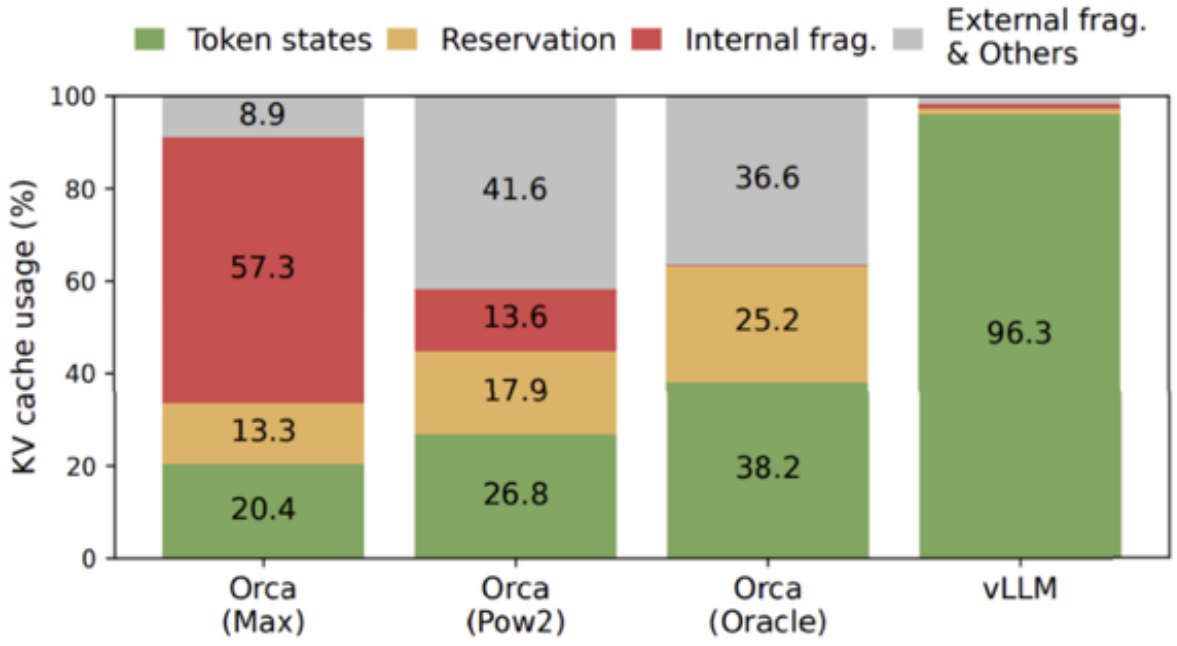
在常规的推理框架中，当我们的服务接收到一条请求时，它会为这条请求中的prompts分配gpu显存空间，其中就包括对KV cache的分配。**由于推理所生成的序列长度大小是无法事先预知的，所以大部分框架会按照(batch_size, max_seq_len)这样的固定尺寸，在gpu显存上预先为一条请求开辟一块连续的矩形存储空间。**

- 我们假设max_seq_len = 8，所以当第1条请求(prompt1)过来时，我们的推理框架为它安排了(1, 8)大小的连续存储空间。
- 当第2条请求（prompt2）过来时，同样也需要1块(1, 8)大小的存储空间。
- 但此时prompt1所在的位置上，只剩3个空格子了，所以它只能另起一行做存储。对prompt3也是同理。



常规kc cache的存储分配

观察整个KV cache排布，你会发现它们的毛病在于太过“静态化”。当你无法预知序列大小时，你为什么一定要死板地为每个序列预留KV cache空间呢？**为什么不能做得更动态化一些，即“用多少占多少”呢？**这样我们就能减少上述这些存储碎片，使得每一时刻推理服务能处理的请求更多，提高吞吐量，这就是vLLM在做的核心事情，我们先通过一张实验图来感受下vLLM在显存利用上的改进效果（VS 其它推理框架）：



vLLM让珍贵且昂贵的GPU内存“物尽其用”了

vLLM是通过什么技术，动态地为请求分配KV cache显存，提升显存利用率的？

为了解决KV缓存带来的内存消耗低效的问题，引入了PagedAttention。

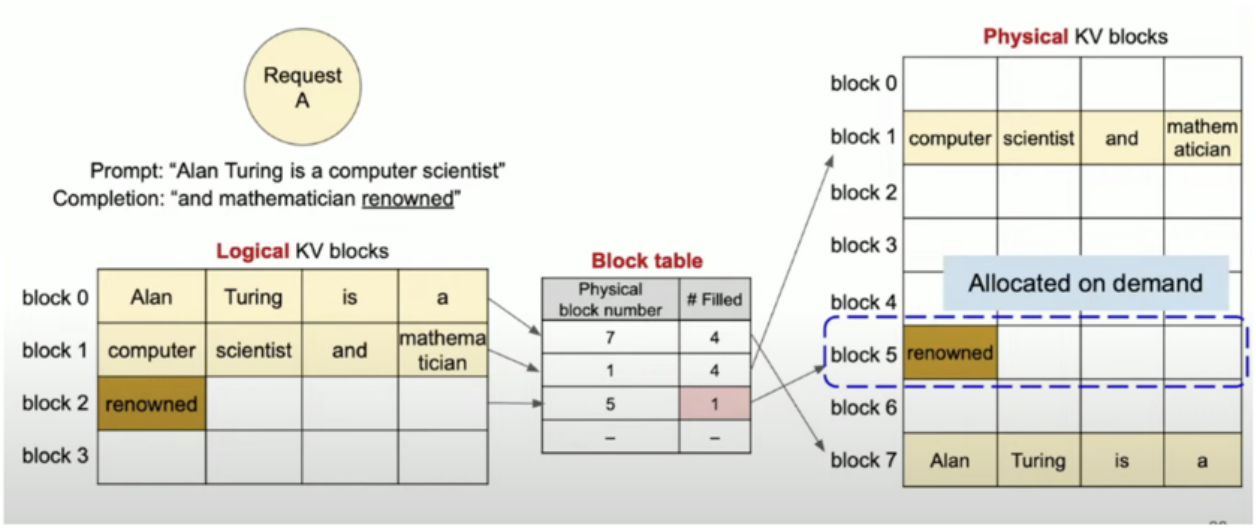
2.2.2. PagedAttention机制解析

vLLM 的 PagedAttention 从**操作系统的分页存储**获得灵感，彻底改变了 KV 缓存的管理方式：

- **分页存储思想**：不再为每个请求分配一整块连续内存，**而是将每个序列的 KV 缓存拆分为若干固定大小的小块（KV 页）。**每个块可存储预定数量（例如 N 个）token 的键/值向量。系统仅在需要时（生成新 token 时）按块分配内存，并通过一个块表（block table）维护序列的逻辑块到物理内存块的映射。
- **非连续存储**：得益于块表映射，序列的各块在物理显存中无需连续。**逻辑上每个序列的 token 仍按顺序划分到第1块、第2块...，但实际这些块可以分散在内存任意位置。**这样避免了请求间争夺大块连续内存导致的外部碎片。**通过这种方式，每个request都会认为自己在一个连续且充足的存储空间上操作，尽管物理上这些数据的存储并不是连续的。**
- **按需分配**：当序列生成新 token 时，vLLM 先尝试放入该序列当前的最后一个块；如果该块已满，则向 GPU 内存的**空闲块池**申请一个新物理块，映射为下一个逻辑块。**整个生成过程中，只分配实际需要的块，不预留整段空间，从而将内存浪费降至极低。**
- **高效释放**：**当请求结束或被中止，其占用的物理块会立即归还空闲池，无需执行大范围的显存释放操作，减少碎片整理开销。**由于所有块大小相同，释放后很容易被其它序列重新利用，也不会产生严重碎片。

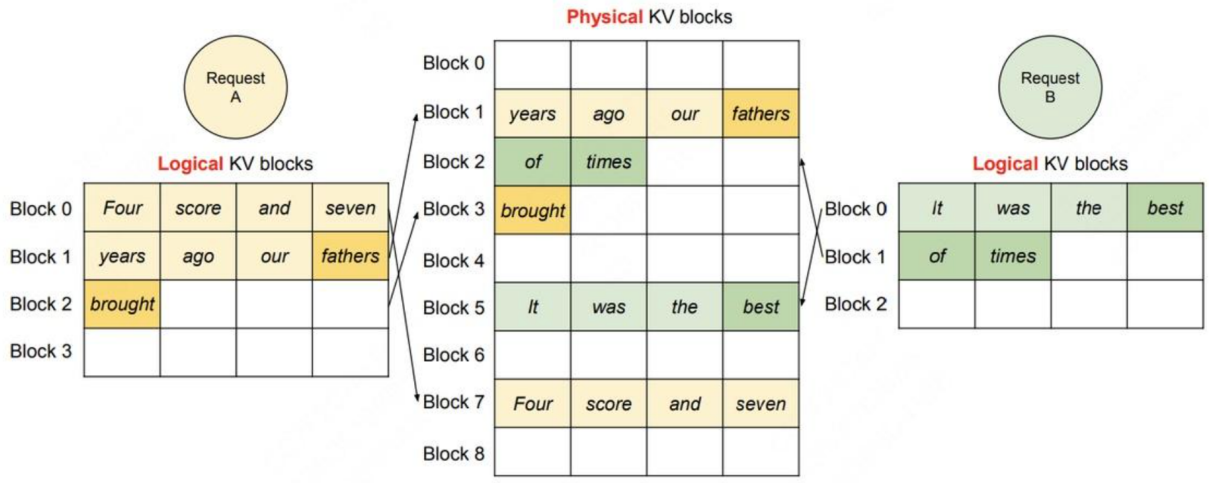
2.2.3 PagedAttention在不同场景下的运作

- 处理单个请求



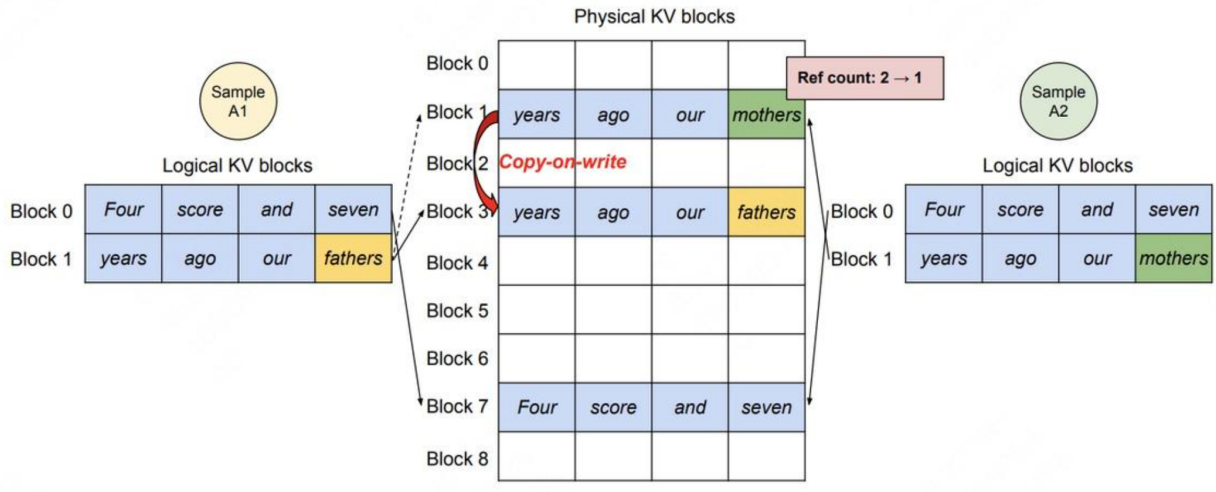
处理单个请求时，PagedAttention运作流程

- 处理多个不相同请求



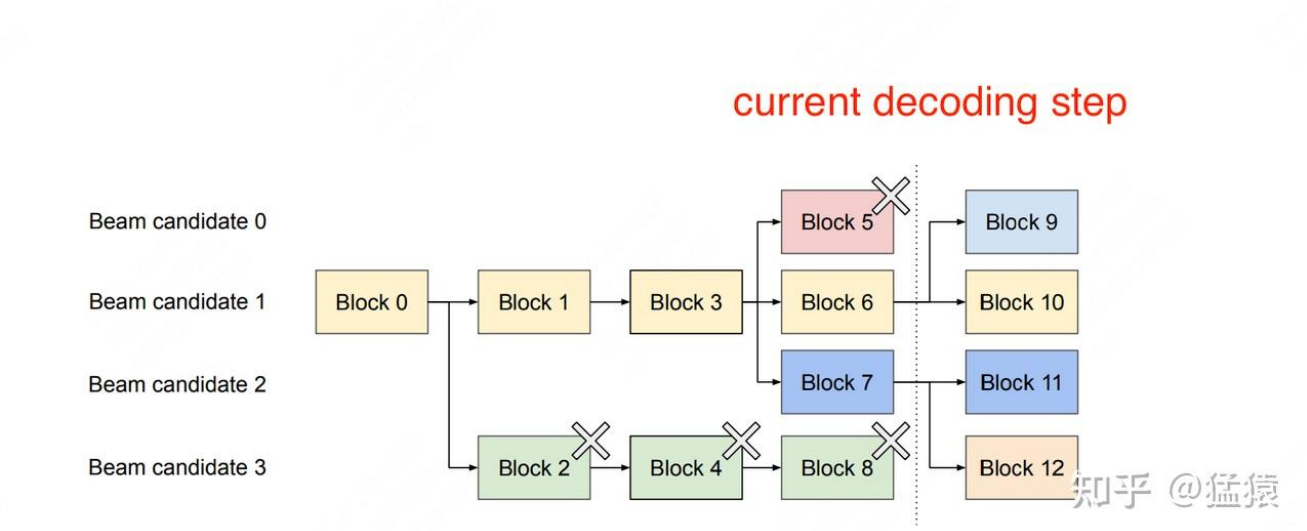
处理多个不相同请求时，PagedAttention运作流程

- 处理多个相同请求（Parallel Sampling）
  - Prefill阶段：
    - 逻辑块：逻辑块独立，每个Sample有各自的逻辑块。
    - 物理块：文字完全相同，共享物理空间。
  - Decode阶段：
    - 两个Sample分别进行推理，得到fathers和mothers。
    - 逻辑块中，两个Sample分别保存各自的逻辑块。
    - 物理块中，触发copy-on-write机制：由于fathers/mothers是两个完全不同的token，因此对物理块block1触发复制机制，即在物理内存上新开辟一块空间。此时物理块block1只和A2的逻辑块block1映射，将其ref count减去1；物理块block3只和A1的逻辑块block1映射，将其ref count设为1。
  - 总结起来，vLLM节省KV cache显存的核心思想是，对于相同数据对应的KV cache，能复用则尽量复用；无法复用时，再考虑开辟新的物理空间。



处理多个请求时，PagedAttention运作流程

- Beam Search：束搜索，这是LLM常用的deocde策略之一，即在每个decode阶段，我不是只产生1个token，而是产生top k个token（这里k也被称为束宽）。top k个token必然对应着此刻的top k个序列。我把这top k个序列喂给模型，假设词表的大小为|V|，那么在下一时刻，我就要在k*|V|个候选者中再选出top k，以此类推。不难想象每一时刻我把top k序列喂给模型时，它们的前置token中有大量的KV cache是重复的。



Beam Search

我们从右往左来看这张图。虚线位置表示“**当前decoding时刻**”，beam width = 4。**图中所有的block皆为逻辑块。**

- 因为beam width = 4，这意味着根据beam search算法，在当前阶段我们生成了top 4个概率最大的token（我们记这4个token为beam candidate 0/1/2/3），它们分别装在block5，block6，block7和block8中。
- 现在我们继续使用beam search算法做decoding，继续找出top 4个最可能的next token。经过我们的计算，这top 4 next token，有2个来自beam candidate 1，有2个来自beam candidate 2。因此我们在block6中引出block9和block10，用于装其中两个top 2 next token；对block7也是同理。
- 现在，block9/10/11/12中装的top 4 next token，就成为新的beam candidates，可以按照和上述一样的方式继续做beam search算法。而对于block5和block8，它们已经在beam search的搜索算法中被淘汰了，后续生成的token也不会和它们产生关系，所以可以清除掉这两个逻辑块，并释放它们对应的物理块的内存空间。
- 好，我们继续往左边来看这幅图。block3引出block5/6/7，block4引出block8，这意味着当前这4个top4 token，是上一个timestep下candidate1和candidate3相关序列生成的（candidate0和2的block没有画出，是因为它们所在的序列被beam search算法淘汰了，因此没有画出的必要）。由于block8已经被淘汰，所以block4也相继被淘汰，并释放对应的物理内存空间。

由此往左一路推，直到block0为止（block0代表着prompt，因此被beam seach中所有的序列共享）。这一路上，我们都根据最新时刻的beam search decoding结果，释

放掉不再被需要的逻辑块和对应的物理内存空间，达到节省显存的目的。

2.2.4. PagedAttention的优势

PagedAttention 带来了多方面的优势：

- **内存利用率大幅提升**：由于仅为实际生成的内容分配内存，并消除了跨请求的大块预留，GPU 内存碎片和浪费显著降低。这使得同等显存下可以并发容纳更多请求的 KV 缓存，从而支持更大的有效批量，提升吞吐。
- **支持长序列和大模型**：逻辑-物理分离后，vLLM 不需要提前为最大长度保留显存，只要有空闲块就能继续扩展序列长度。对于超长输入/输出或模型参数非常大的场景，PagedAttention 也能灵活调度内存，用完再申请新的块。**此外，vLLM 还支持在GPU显存不足时，将部分序列的KV块交换（swap）到CPU内存，并在其重新调度时通过重算快速恢复。**
- **内存共享与分支开销降低**：PagedAttention 天然支持KV 缓存共享。利用块表的灵活映射，不同序列可以指向相同的物理块实现内容共享。典型应用是在**并行采样**和**Beam Search**等生成分支场景：多个输出序列共享相同的输入 prompt 或共同前缀。vLLM 中这些序列的前缀部分可以共用同一批物理KV块，避免重复存储和计算。当分支真正发生差异（如Beam Search某条beam生成不同新token），仅对新增部分分配块即可；若共享块需要修改，则触发写时复制（Copy-on-Write）另给新块，保证线程安全。通过共享前缀，并行采样可减少约6%~30%的内存占用，Beam Search 减少约44%~66% 的KV内存。这带来明显性能收益：复杂采样算法的显存开销降低最多一半，令其在服务中更实用。

综上，PagedAttention 将传统 LLM 推理中低效的内存管理变成了类似操作系统虚拟内存的按需分页。几乎所有内存浪费被限制在每个序列**最后一个未填满的块**内。实验表明，这种改进让系统的GPU内存利用率接近最优，有效批量显著增加，从而带来吞吐量的大幅提升。PagedAttention 正是 vLLM 性能飞跃的“秘密武器”。

2.3. 调度与抢占

> 当采用动态分配显存的办法时，虽然明面上同一时刻能处理更多的prompt了，但因为没有为每个prompt预留充足的显存空间。**如果在某一时刻整个显存被打满了，而此时所有的prompt都没做完推理，那该怎么办?**

2.3.1. vLLM的总原则

先来先服务（FCFS），后来先抢占，gpu不够就先swap到cpu上。

- 后来先抢占

- 当一堆请求到达vLLM服务器，导致gpu显存不足时，vLLM会**暂停这堆请求中最后到达的那些请求的推理，同时将它们相关的KV cache从gpu上释放掉，以便为更早到达的请求留出足够的gpu空间，让它们完成推理任务**。如果不这样做的话，各个请求间相互争夺gpu资源，最终将导致没有任何一个请求能完成推理任务。等到先来的请求做完了推理，vLLM调度器认为gpu上有足够的空间了，就能恢复那些被中断的请求的执行了。

2.3.2. “后来先抢占”的实现

对于这些因gpu资源不足而被抢占的任务，vLLM要完成两件事：

- 暂停它们的执行，同时将与之相关的KV cache从gpu上释放掉
- 等gpu资源充足时，重新恢复它们的执行

针对这两件事，vLLM分别设计了**Swapping（交换策略）**和**Recomputation（重计算策略）**来解决。我们来细看这两个策略。

（1）Swapping

> 对于被抢占的请求，vLLM要将其KV cache从gpu上释放掉，那么：
> - **问题1：该释放哪些KV cache?**
> - **问题2：要把这些KV cache释放到哪里去?**

先看问题1。在vLLM中，一般采取的是all-or-nothing策略，即释放被抢占请求的所有block。

再来看问题2。对于这些被选中要释放的KV block，如果将它们直接丢掉，那未免过于浪费。**vLLM采用的做法是将其从gpu上交换（Swap）到cpu上**。这样等到gpu显存充足时，再把这些block从cpu上重载回来。

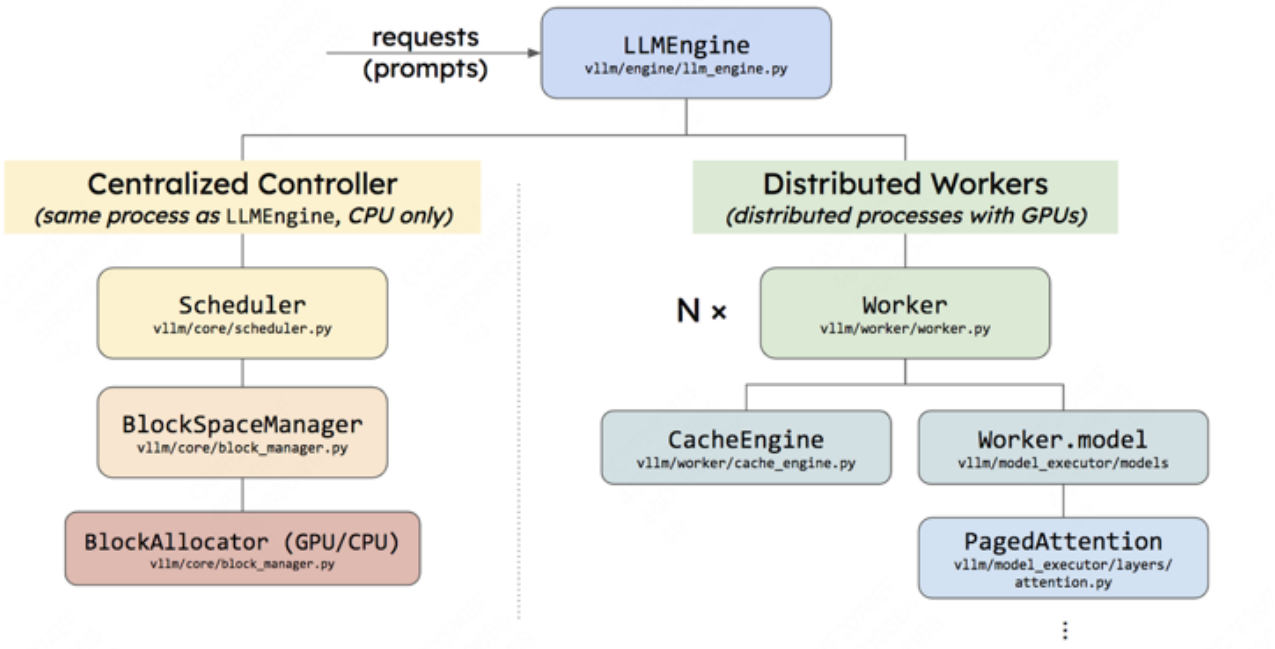
（2）Recomputation

知道了Swapping机制，重计算的过程也很好理解了：对于有些任务，当它们因为资源不足而被抢占时，可以不做swap，而是直接释放它们的物理块，把它们重新放入等待处理的队列中，等后续资源充足时再重新从prefill阶段开始做推理。

2.4. 总结

PageAttention为动态批处理提供了灵活、高效的内存管理基础，使其调度能力得以充分发挥；而动态批处理则将PageAttention节省的内存资源转化为实实在在的计算吞吐量提升。

3. vLLM整体架构设计

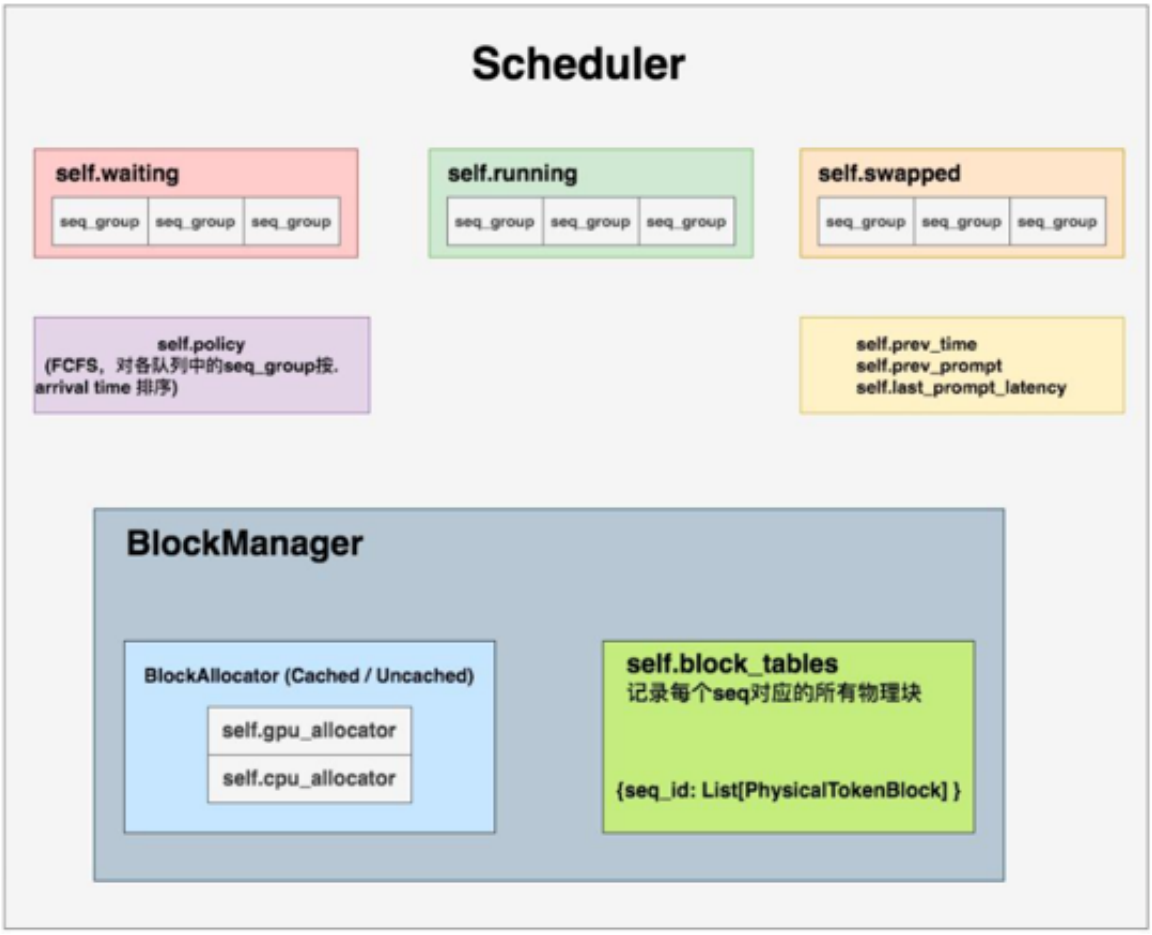


vLLM代码架构

3.1. Centralized Controller

Centralized Controller，也就是前文我们所说的调度器(Scheduler)。

- **调度器的主要作用就是，在每1个推理阶段，决定要把哪些数据送给模型做推理，同时负责给这些模型分配KV Cache物理块**。但要注意，它只是分配了物理块的id，而不是物理块本身。物理块的实际分配是模型在推理过程中根据物理块id来操作的，也就是CacheEngine做的事情。



Scheduler结构

self.policy：是vLLM自定义的一个Policy实例，目标是根据调度器总策略（FCFS，First Come First Serve，先来先服务）原则，对各个队列里的seq_group按照其arrival time进行排序。

self.prev_time：**上一次调度发起的时间点，初始化为0。**我们知道每执行1次推理阶段前，调度器都要做一次调度，这个变量存放的就是上次调度发起的时间点。

self.prev_prompt：**取值为True/False，初始化为False。若上一次调度时，调度器有从waiting队列中取出seq_group做推理，即为True，否则为False。**

self.last_prompt_latency：**记录“当前调度时刻（now） - 最后一次有从waiting队列中取数做推理的那个调度时刻”的差值**（并不是每一次调度时，调度器一定都会从waiting队列中取seq_group，它可能依旧继续对running队列中的数据做推理），初始化为0。

BlockManager：物理块管理器。这也是vLLM自定义的一个class。物理块管理器这个class下又维护着两个重要属性：

- **BlockAllocator**：物理块分配者，负责实际为seq做物理块的分配、释放、拷贝等操作。其下又分成self.gpu_allocator和self.cpu_allocator两种类型，分别管理gpu和cpu上的物理块。
- **self.block_tables**：负责维护每个seq下的物理块列表。

3.2. Distributed Workers

Distributed Workers，也就是分布式系统，你可以将每个worker理解成一块gpu。它的作用是将我们要使用的模型load到各块卡上，然后对Controller传来的数据做1次推理，返回相关结果。

- **Worker：在硬件上，它指gpu；在代码上，它指的是Worker实例**（每个gpu上的进程维护自己的Worker实例）。在每个Worker实例中又管控着如下两个重要实例：
  - **CacheEngine：负责管控gpu/cpu上的KV cache物理块**（调度器的block manager只负责物理块id的分配，CacheEngine则是根据这个id分配结果实打实地在管理物理块中的数据）
  - **Worker.model：负责加载模型，并执行推理。**PagedAttention的相关逻辑，就维护这个实例关联的代码下。

4. vLLM运作流程

当一条请求过来时，整个vLLM是怎么运作的呢?

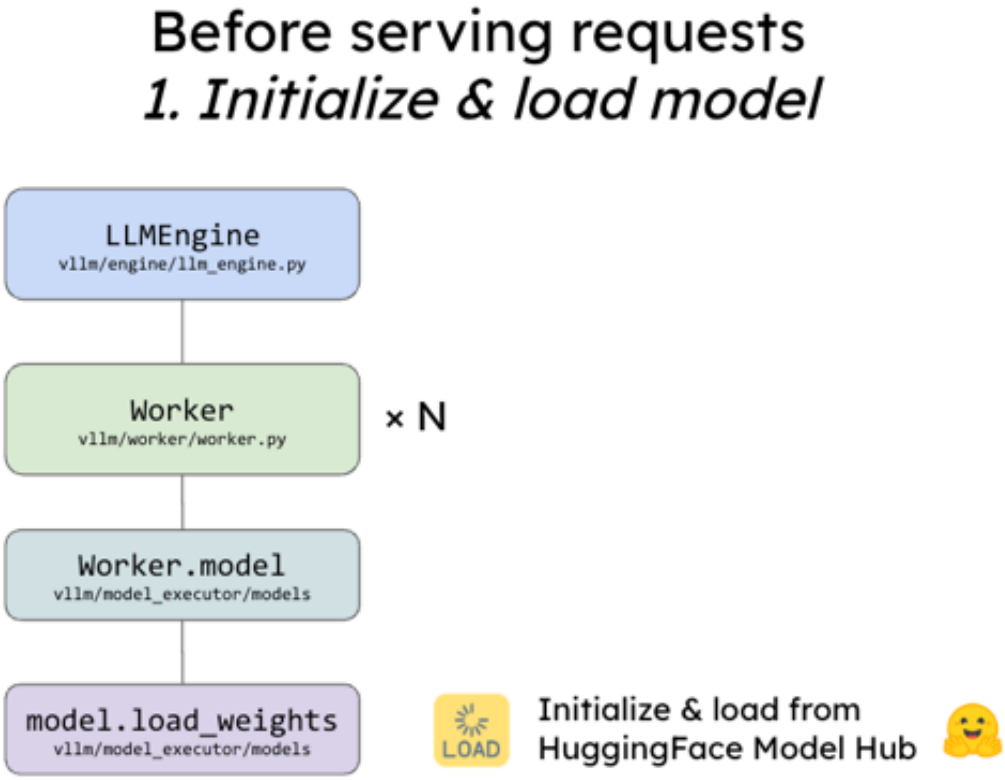
离线批处理进行调用 Python

```
from vllm import LLM, SamplingParams

if __name__ == "__main__":
    # Sample prompts.
    prompts = [
        "Hello, my name is",
        "The president of the United States is",
        "The capital of France is",
        "The future of AI is",
    ]
    # Create a sampling params object.
    sampling_params = SamplingParams(temperature=0.8, top_p=0.95)

    # 创建llm实例，在这个过程中也创建了llm实例下维护的llm_engine
    llm = LLM(model="facebook/opt-125m")
    # 执行offline batching推理，得到这批prompts的输出
    outputs = llm.generate(prompts, sampling_params)
    # 打印输出
    for output in outputs:
        prompt = output.prompt
        generated_text = output.outputs[0].text
        print(f"Prompt: {prompt!r}, Generated text:
{generated_text!r}")
```

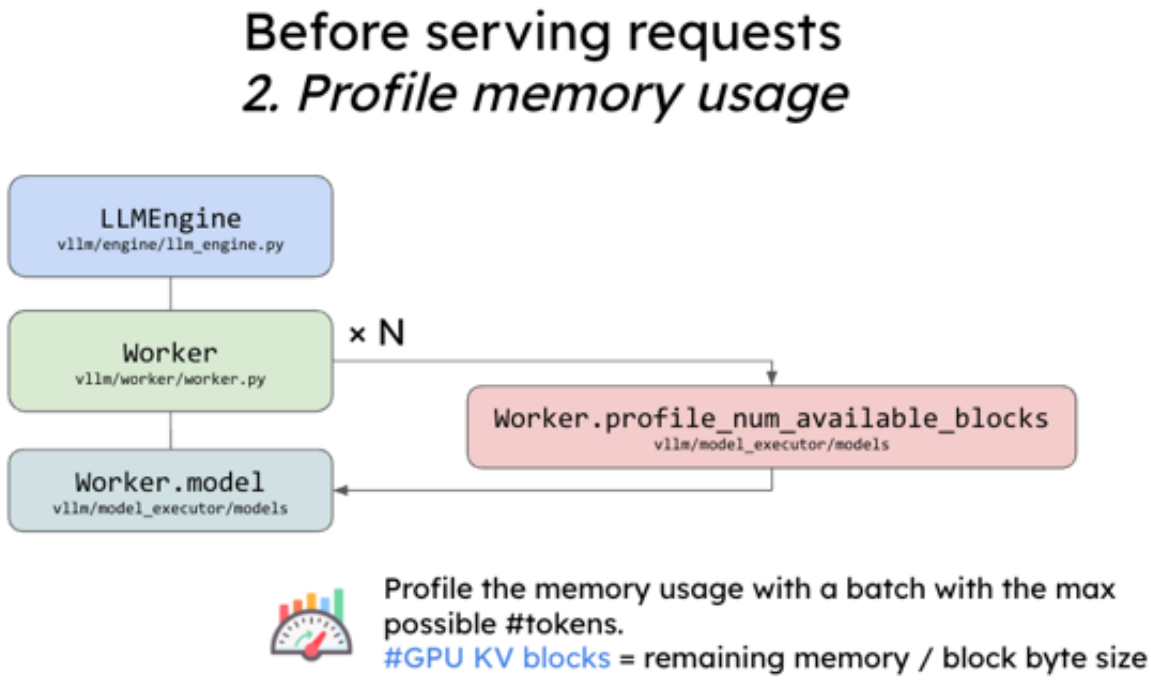
4.1. 加载模型




```python
    #
    ================================================================================
    ========
    # 初始化vLLM offline batched inference实例，并加载指定模型
    #
    ================================================================================
    ========
    llm = LLM(model="facebook/opt-125m")
```

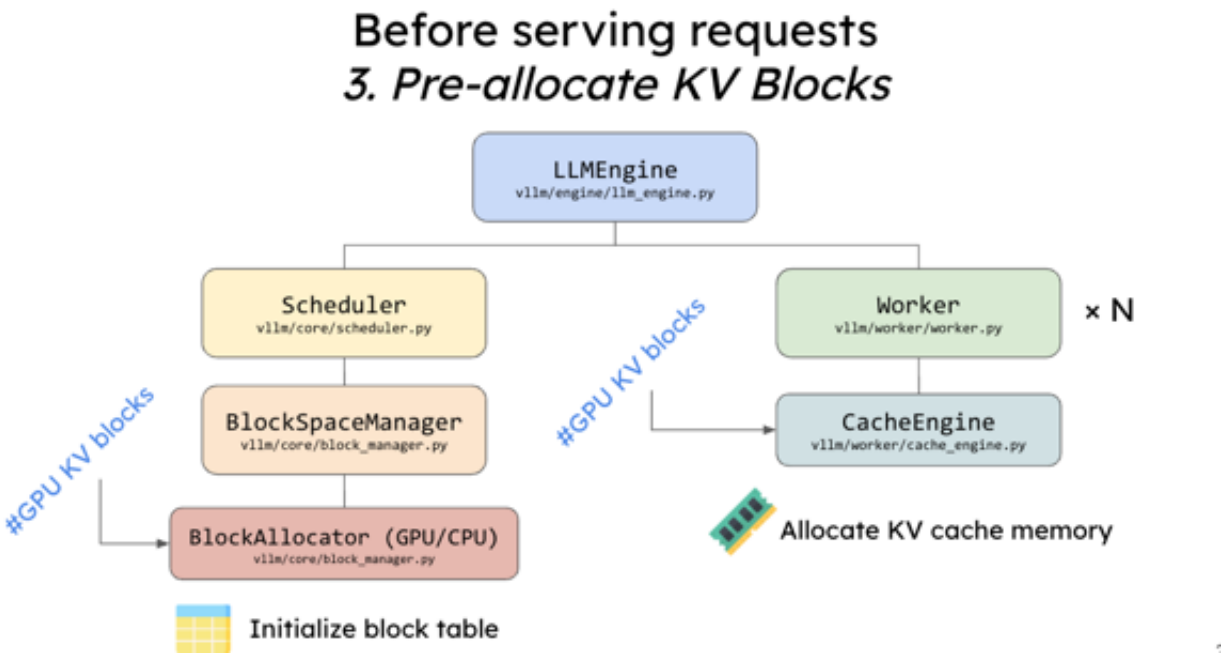
这里在做的事很直观：把base model加载到worker上。

4.2. 预分配显存



实例化了一个离线批处理的vLLM对象。其本质是实例化了一个内核引擎LLMEngine对象。在执行这个步骤时，LLMEngine会执行一次模拟实验（profiling），来判断需要在gpu上预留多少的显存空间给KV Cache block。vLLM管这个步骤叫profile_num_available_blocks。

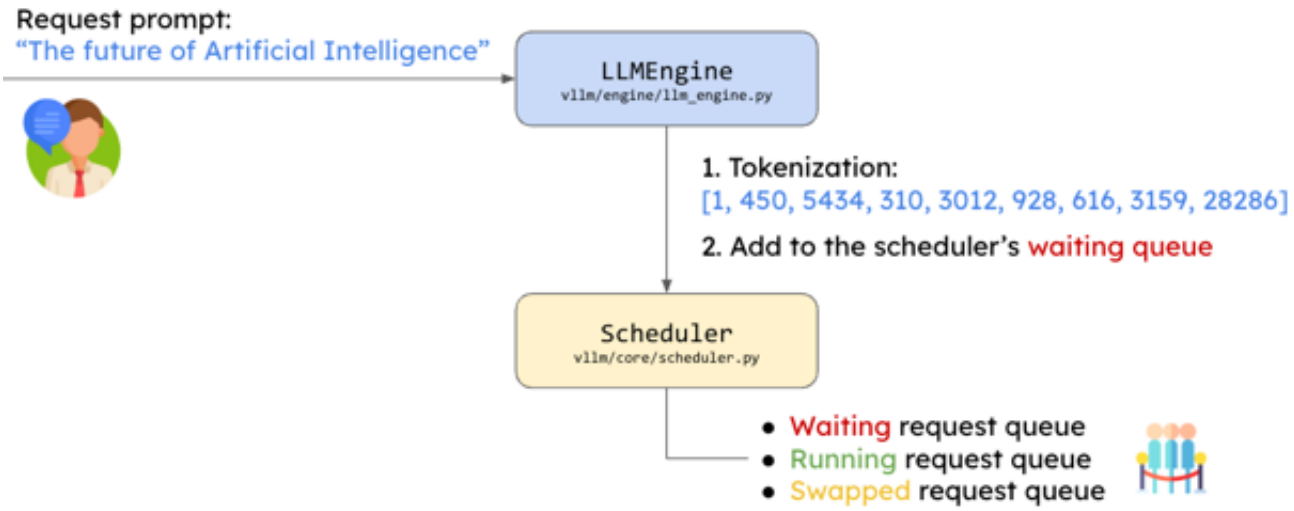
- 加载预分配的KV Cache到gpu上



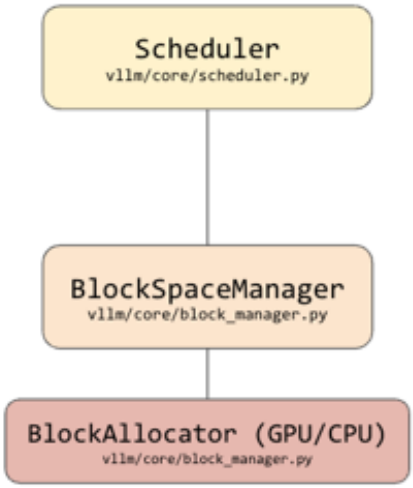
当我们确定好KV Cache block的大小后，我们就可以创建empty tensor，将其先放置到gpu上，实现显存的预分配。以后这块显存就是专门用来做KV Cache的了。也正是因为这种预分配，你可能会发现在vLLM初始化后，显存的占用比你预想地要多（高过模型大小），这就是预分配起的作用。

4.3. Scheduler调度

When requests arrive



At every step, the scheduler



Decide the set of requests to run at the current step.

- When there are free KV block memory
  waiting → running
- When no KV block memory available for new tokens:
  Swapping: running → swapped
  Recomputation: running → waiting

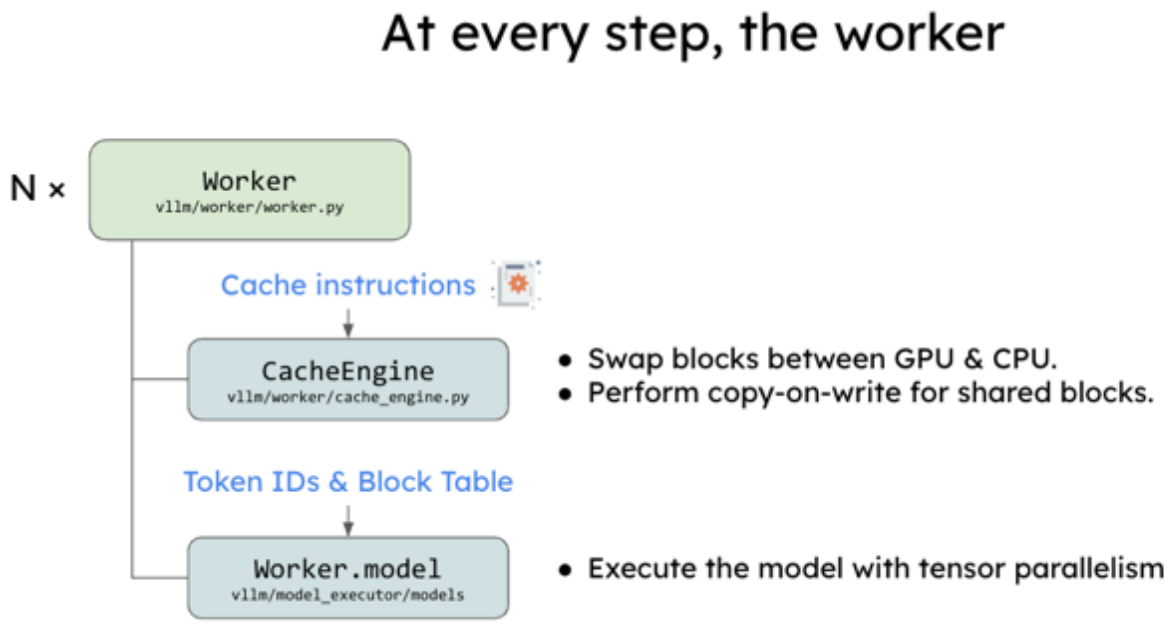
- Allocate space for new KV Cache.
- Handle block sharing.
- Swap/delete preempted KV blocks.

→ Cache instructions & Block table

- **self.waiting, self.running, self.swapped**：这三个都是python的deque()实例（双端队列，允许你从队列两侧添加或删除元素）。
  - **waiting队列用于存放所有还未开始做推理的seq_group**，“未开始”指连prefill阶段都没有经历过。所以waiting队列中的seq_group只有一个seq，即是原始的prompt。
  - **running队列用于存放当前正在做推理的seq_group。更准确地说，它存放的是上1个推理阶段被送去做推理的seq_group们**，在开始新一轮推理阶段时，调度器会根据本轮的筛选结果，更新running队列，即决定本轮要送哪些seq_group去做推理。
  - **swapped队列用于存放被抢占的seq_group**。若一个seq_group被抢占，调度器会对它执行swap或recomputation操作，分别对应着将它送去swapped队列或waiting队列。

- 请求发起：The future of Artifical Intelligence
- 到达LLMEngine
  - 分词
  - 加入调度序列：waiting queue
- 调度器Scheduler进行队列管理：筛选出需要运行的一组请求
  - 当KV块内存有空闲：waiting请求变为running
  - 当没有KV块内存用于生成新token
    - running变为swapping
    - running变为waiting
- BlockSpaceManager进行块空间管理：
  - 给新的KVCache分配空间
  - 处理块共享
  - 交换/删除被抢占的KV块
- BlockAllocator：当BlockSpaceManager下达“分配”或者“交换”命令时，块分配器在真实的物理内存上进行分配和回收。

4.4. 执行推理



Distributed Workers接受来自Scheduler的请求，分发到各个worker上去做推理。

- CacheEngine负责管理实际KVCache。
- Worker.model负责加载模型，进行推理。

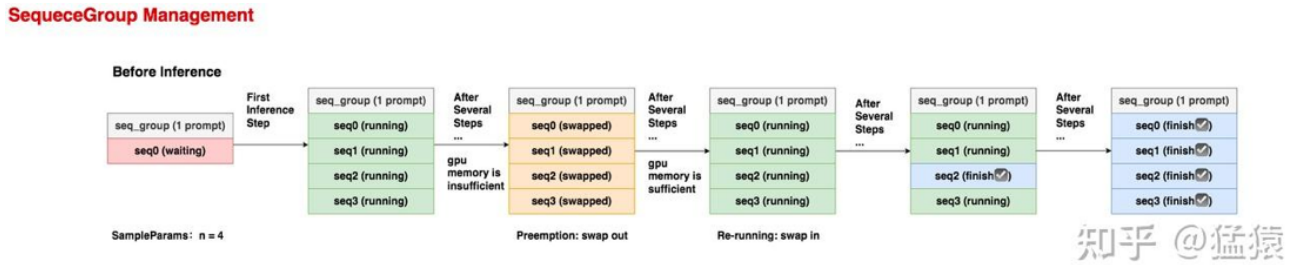
推理入口：`outputs = llm.generate(prompts, sampling_params)`

当我们调用`generate()`时，它实际做了两件事情：

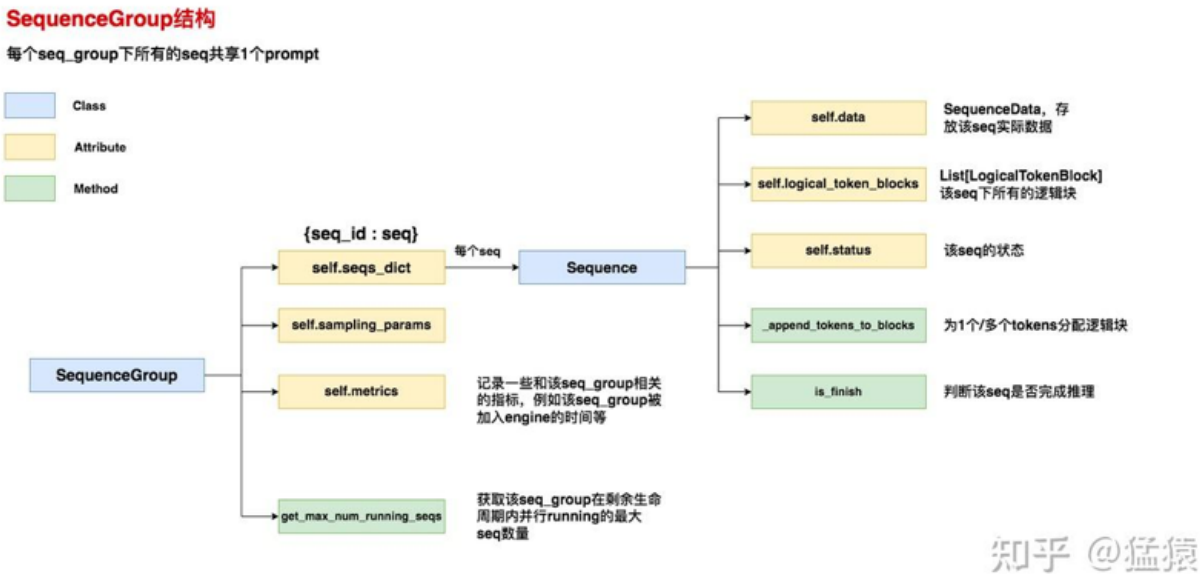
- `_add_request`：**将输入数据传给LLMEngine：**把每1个prompt包装成一个SequenceGroup对象（"1个prompt -> 多个outputs"的结构）。把包装成SequenceGroup对象的数据加入调度器（Scheduler）的waiting队列，等待处理。
- `_run_engine`：**执行推理：**只要调度器的waiting/running/swapped队列非空，我们就认为此时这批batch还没有做完推理，这时我们就会**调用LLMEngine的step()**函数，来完成1次调度以决定要送哪些数据去做推理。

SequenceGroup

> 可能出现"1个prompt -> 多个outputs"的情况。那是否能设计一种办法，对1个prompt下所有的outputs进行集中管理，来方便vLLM更好做推理呢?



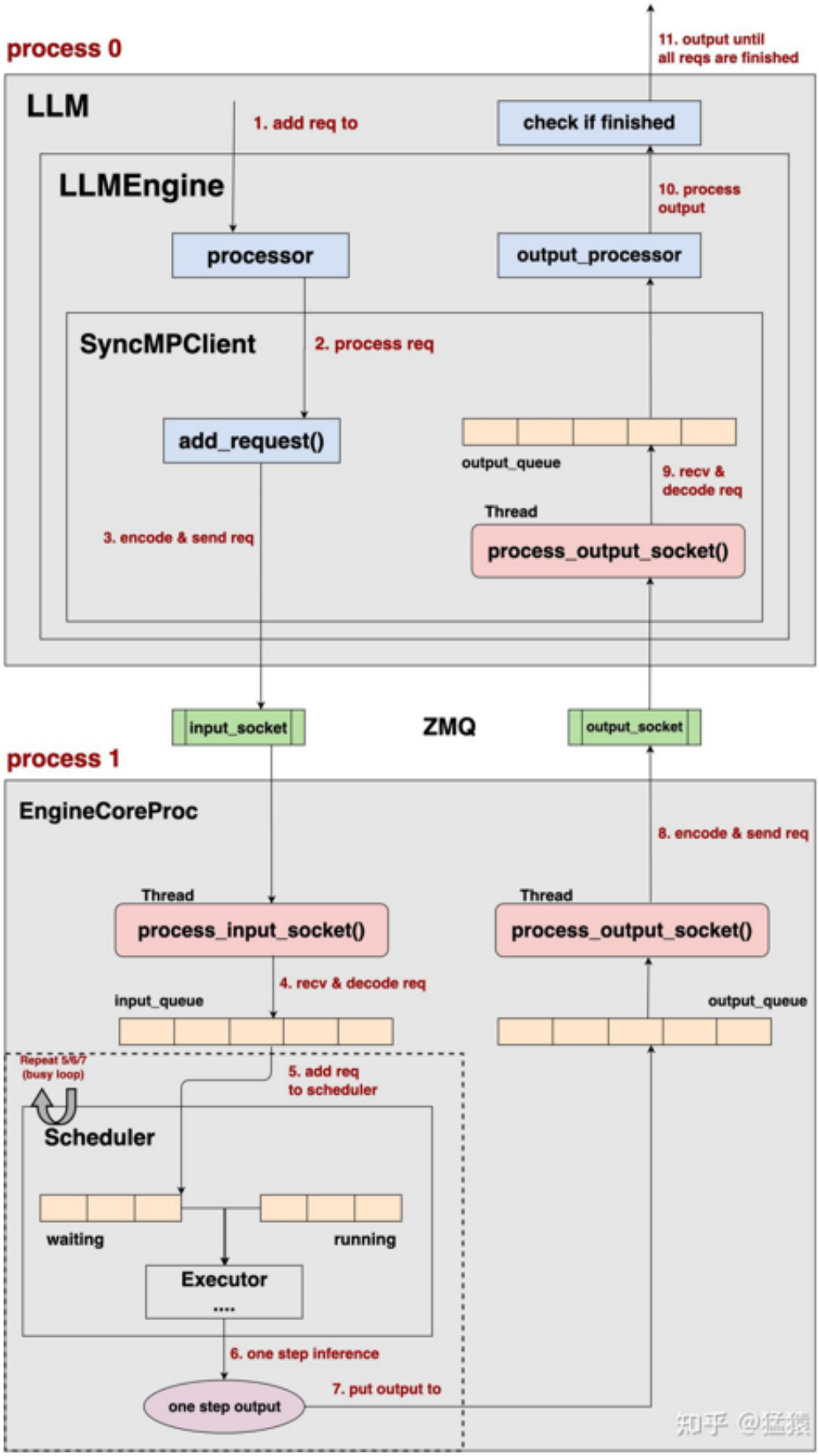
采样参数n=4,需要四种不同的output



SequenceGroup结构

到这里不妨思考一下，vLLM v0的整个运作流程有没有哪些可以优化的地方?

vLLM v1的优化，还是以offline batching为例:

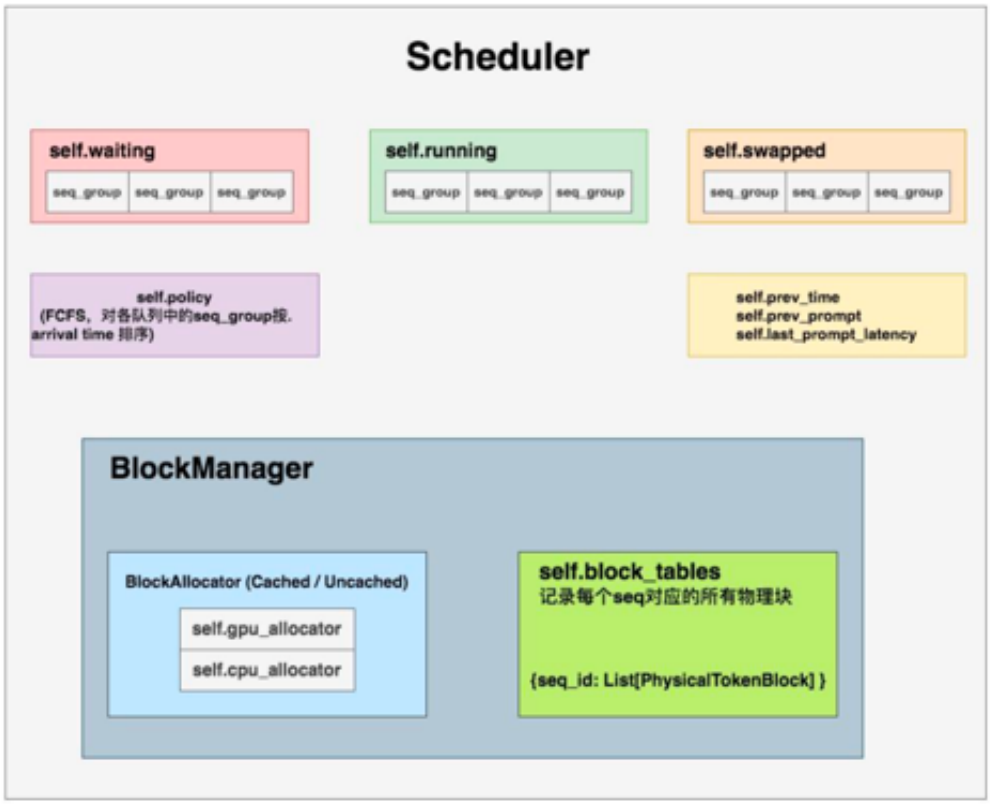


- vllm v1将`请求的pre-process和输出结果的post-process`与`实际的推理过程`拆分在2个不同的进程中(process0, process1)。
- Client负责`请求的pre-process和输出结果的post-process`，EngineCore负责`实际的推理过程`，不同进程间使用ZMQ来通信数据。

- 通过这样的进程拆分，在更好实现cpu和gpu运作的overlap的同时，也将各种模型复杂的前置和后置处理模块化，统一交给processor和output_processor进行管理。

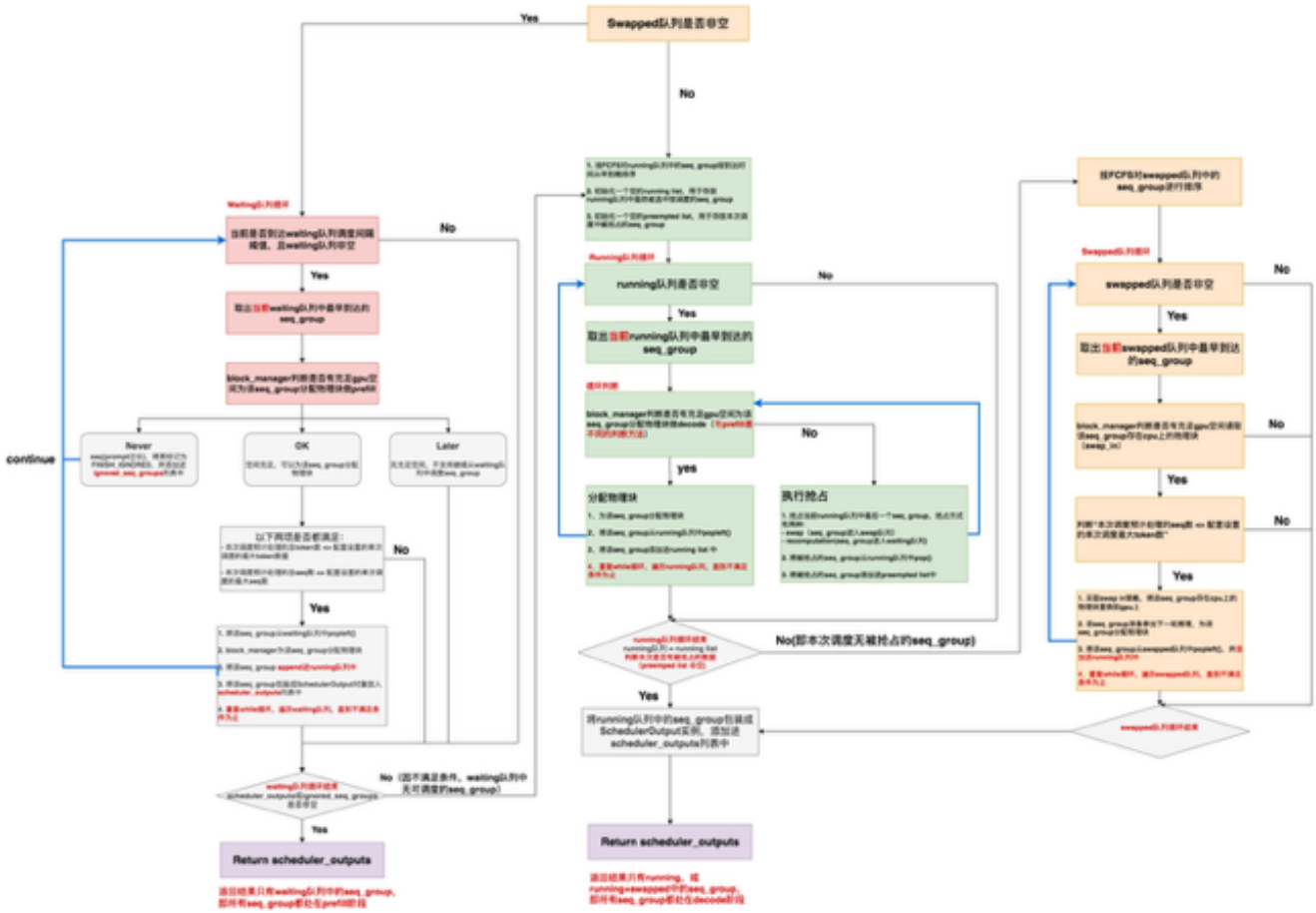
5. Scheduler

5.1. 调度器结构



Scheduler结构

5.2. 整体调度策略



调度策略

running队列中的seq_group不一定能继续在本次调度中被选中做推理，这是因为gpu上KV cache的使用情况一直在变动，以及waiting队列中持续有新的请求进来的原因。所以调度策略的职责就是要根据这些变动，对送入模型做推理的数据做动态规划。

总结来说：

- **如果当前swapped队列为空，**那就去检查是否能从waiting队列中调度seq_group，直到不满足调度条件为止（gpu空间不足，或waiting队列已为空等）。此时，1个推理阶段中，所有的seq_group都处在prefill阶段。
- **如果当前swapped队列非空，或者无法从waiting队列中调度任何seq_group时：**
  - 检查是否能从running队列中调度seq_group，直到不满足调度条件为止。
  - 若本次无新的被抢占的seq_group，且swapped队列非空，就检查是否能从swapped队列中调度seq_group，直到不满足调度条件为止。

此时，1个推理阶段中，所有的seq_group要么全来自running队列，要么来自running + swapped队列，它们都处在decode阶段。

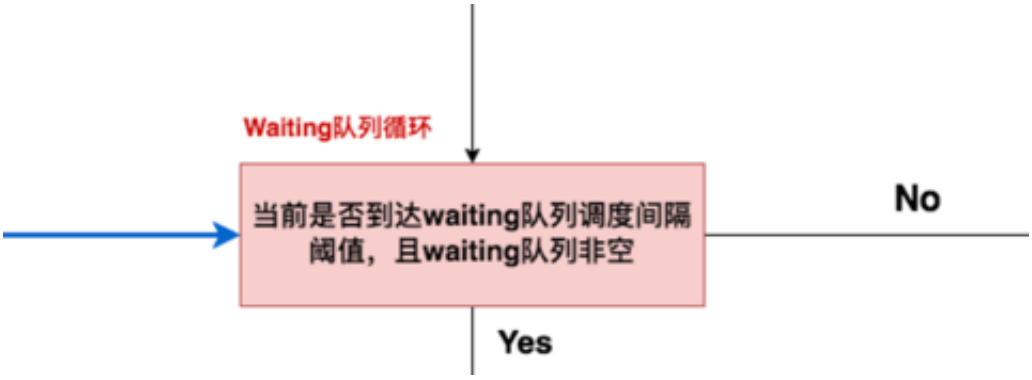
至此我们要记住vLLM调度中非常重要的一点：在1个推理阶段中，所有的seq_group要么全部处在prefill阶段。要么全部处在decode阶段。

> 你可能想问：**为什么要以swapped是否非空为判断入口呢?**

这是因为，如果当前调度步骤中swapped队列非空，说明在之前的调度步骤中这些可怜的seq_group因为资源不足被抢占，而停滞了推理。所以**根据FCFS规则，当gpu上有充足资源时，我们应该先考虑它们，而不是考虑waiting队列中新来的那些seq_group。**

同理，在图中你会发现，当我们进入对running队列的调度时（图中红色分支），我们会根据“**本次调度是否有新的被抢占的seq_group**”，来决定要不要调度swapped队列中的数据。这个理由也很简单：在本次调度中，我就是因为考虑到gpu空间不足的风险，我才新抢占了一批序列。既然存在这个风险，我就最好不要再去已有的swapped队列中继续调度seq_group了。

5.3. _passed_delay：判断调度waiting队列的时间点

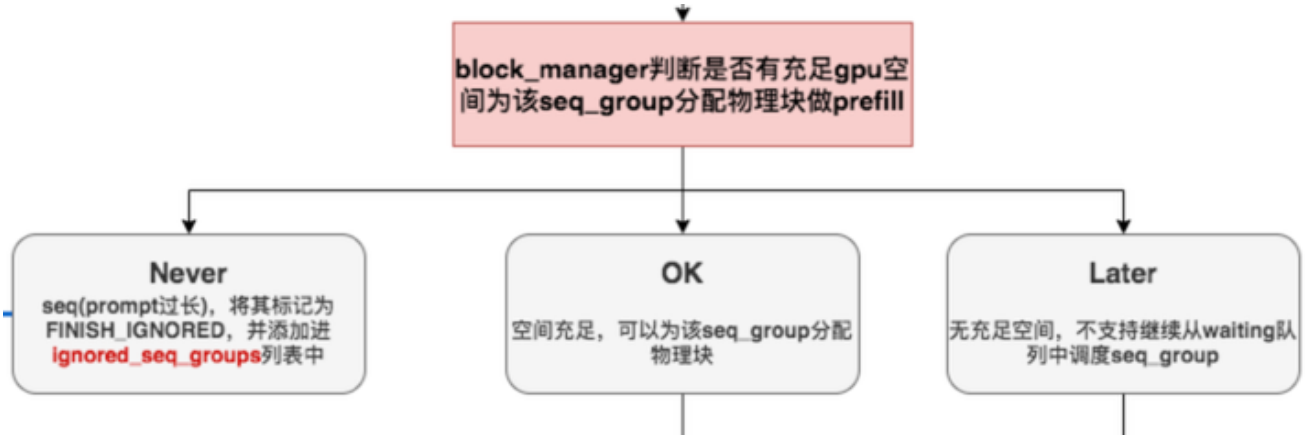


在5.1的流程图中，我们会看到**进入waiting循环的判断条件之一是：waiting队列是否达到调度间隔阈值**。这是个什么东西？又为什么要设置这样一个阈值呢？

我们知道模型在做推理时，waiting队列中是源源不断有seq_group进来的，一旦vLLM选择调度waiting队列，它就会停下对running/swapped中seq_group的decode处理，转而去做waiting中seq_group的prefill，也即**vLLM必须在新来的seq_group和已经在做推理的seq_group间取得一种均衡：既不能完全不管新来的请求，也不能耽误正在做推理的请求。所以“waiting队列调度间隔阈值”就是来控制这种均衡的：**

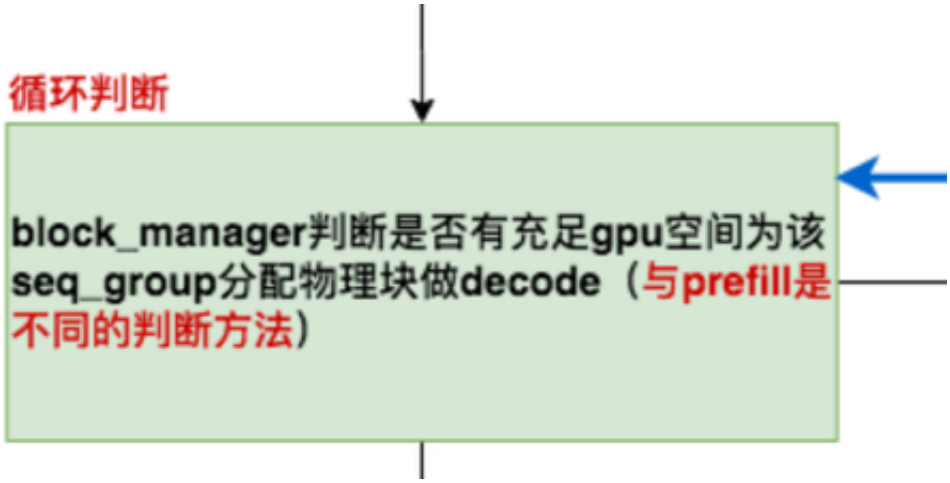
- **调度间隔设置得太小**，每次调度都只关心waiting中的新请求，这样发送旧请求的用户就迟迟得不到反馈结果。且此时waiting队列中积累的新请求数量可能比较少，不利于做batching，浪费了并发处理的能力。
- **调度间隔设置得太大**，waiting中的请求持续挤压，同样对vLLM推理的整体吞吐有影响。

5.4. can_allocate：能否为seq_group分配物理块做prefill



通过了调度时间阈值的判断条件，现在我们顺利从waiting中取出一个seq_group，我们将对它进行prefill操作。**所以这里我们必须先判断：gpu上是否有充足的空间为该seq_group分配物理块做prefill**，根据5.1中绘制的调度器结构，这个操作当然是由我们的self.block_manager来做。

5.5. can_append_slot：能否为seq_group分配物理块做decode



我们从running队列中调度seq_group时，我们也会判断是否能为该seq_group分配物理块。**但这时，我们的物理块空间是用来做decode的（给每个seq分配1个token的位置），而不是用来做prefill的（给每个seq分配若干个token的位置），所以这里我们采取的是另一种判断方法** `can_append_slot` **。**

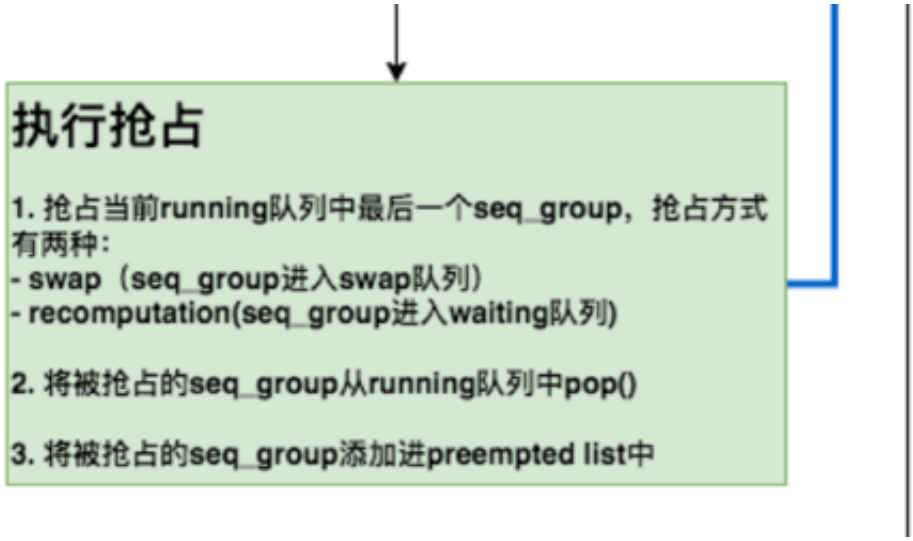
更具体来说，running队列中seq_group下的n个seqs在上1个推理阶段共生成了n个token。在本次调度中，我们要先为这n个token分配物理块空间，用于存放它们在本次调度中即将产生的KV值。我们知道：

- 当往1个seq的物理块上添加1个token时，可能有两种情况：
  - 之前的物理块满了，所以我新开1个物理块给它
  - 之前的物理块没满，我直接添加在最后一个物理块的空槽位上
  - **所以，对于1个seq来说，最坏的情况就是添加1个物理块；对于n个seqs来说，最坏的情况就是添加n个物理块（想想前面讲过的copy-on-write机制）**

- **对于1个seq_group，除了那些标记为“finish”的seq外，其余seqs要么一起送去推理，要么一起不送去推理。即它们是集体行动的**

所以，判断能否对一个正在running的seq_group继续做推理的最保守的方式，就是判断当前可用的物理块数量是否至少为n。

5.6. preempt：抢占策略



抢占策略的核心逻辑：

在若干个推理阶段后，gpu上的资源不够了，这个seq_group不幸被调度器抢占（preemption），它相关的KV block也被swap out到cpu上。此时所有seq的状态变为swapped。这里要注意，**当一个seq_group被抢占时，对它的处理有两种方式：**

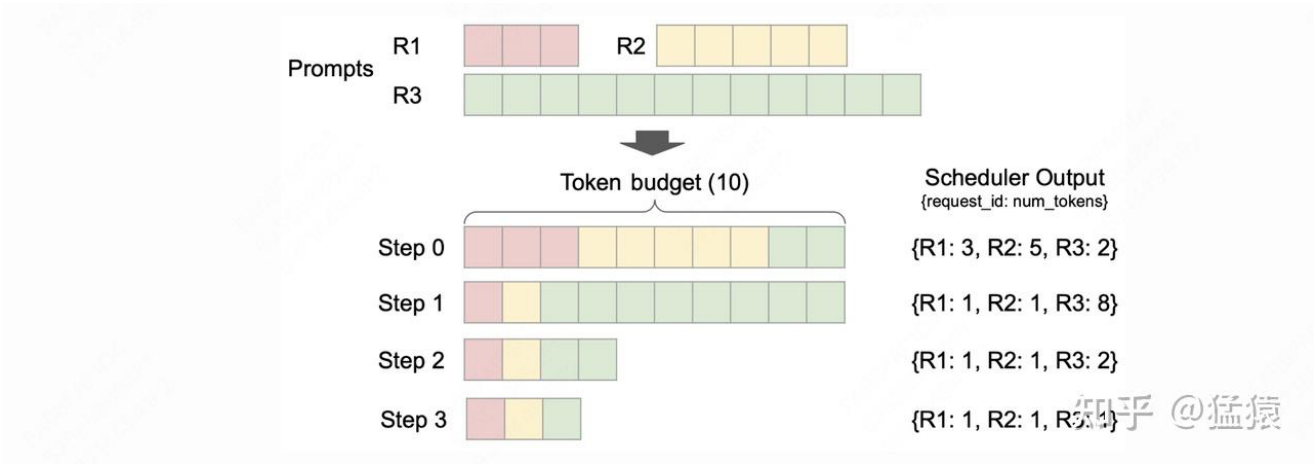
- **Swap：如果该seq_group剩余生命周期中并行运行的最大seq数量 > 1，此时会采取swap策略**，即把seq_group下【所有】seq的KV block从gpu上卸载到cpu上。（seq数量比较多，直接把算出的KV block抛弃，比较可惜）
- **Recomputation：如果该seq_group剩余生命周期中并行运行的最大seq数量 = 1，此时会采取recomputation策略**，即把该seq_group相关的物理块都释放掉，然后将它重新放回waiting队列中(**放在最前面**)。等下次它被选中推理时，就是从prefill阶段开始重新推理了，因此被称为“重计算”。（seq数量少，重新计算KV block的成本不高）

5.7. vLLM v1的改进

在V0的策略中，每次调度步骤要么全是做prefill的请求，要么全是做decode的请求，除此以外，调度器中维护着waiting，running，swapped三个队列，整体来说调度策略比较复杂。

在V1中，调度策略简化许多，**最主要的就是允许单次调度步骤中同时调度prefill和decode请求，同时调度器中只维护waiting和running队列。我们从vllm官方blog中来**

看下V1的整体调度思想



如图所示：

- 首先，**vllm会对单次调度步骤设置一个 `token_budget`，它用来决定每次调度中最多允许“计算”多少个token。**这个token_budget可以由用户通过 `scheduler_config.max_num_batched_tokens` 进行配置。在我们的图例中token_budget = 10
- 假设此时waiting队列中有3个R1，R2，R3三个请求（prompts），长度分别为3，5，12。请求已经按照到来的先后顺序排列好了。**Vllm v1依然采用的是FCFS原则（First come First serve），按先来后到的顺序处理请求。**
- step0：
  - 调度器开始执行调度。**根据某种策略**，调度器决定将【R1的3个token】，【R2的5个token】都算入本次调度步骤中。此时token_budget = 10 - 3 - 5 = 2。
  - R3有12个token需要计算，但本轮调度的token_budget只剩2个，所以只取R3的2个token加入本次调度中
  - R1，R2和R3都会从waiting队列转移到running队列上
- step1:
  - 调度器开始执行新一轮调度。此时R1和R2都做完了prefill，进入decode阶段，它们待计算的token都只有1个。R3仍在prefill阶段，并根据token_budget再送入8个token进入本轮调度
- 以此类推，可以发现，直到step3为止，最长的R3才做完了prefill，进入decode阶段。

6. BlockManager

6.1. BlockManager结构

BlockManager这个class下又维护着两个重要属性：

- `BlockAllocator`：**物理块分配者，负责实际为seq做物理块的分配、释放、拷贝等操作。**其下又分成`self.gpu_allocator`和`self.cpu_allocator`两种类型，分别管理gpu和cpu上的物理块。
- `self.block_tables`：**负责维护每个seq下的物理块列表，本质上它是一个字典，形式如`{seq_id: List[PhysicalTokenBlock]}`。**注意，这个字典维护着【所有】seq_group下seq的物理块，而不是单独某一个seq的。因为调度器是全局的，所以它下面的的BlockManager自然也是全局的。

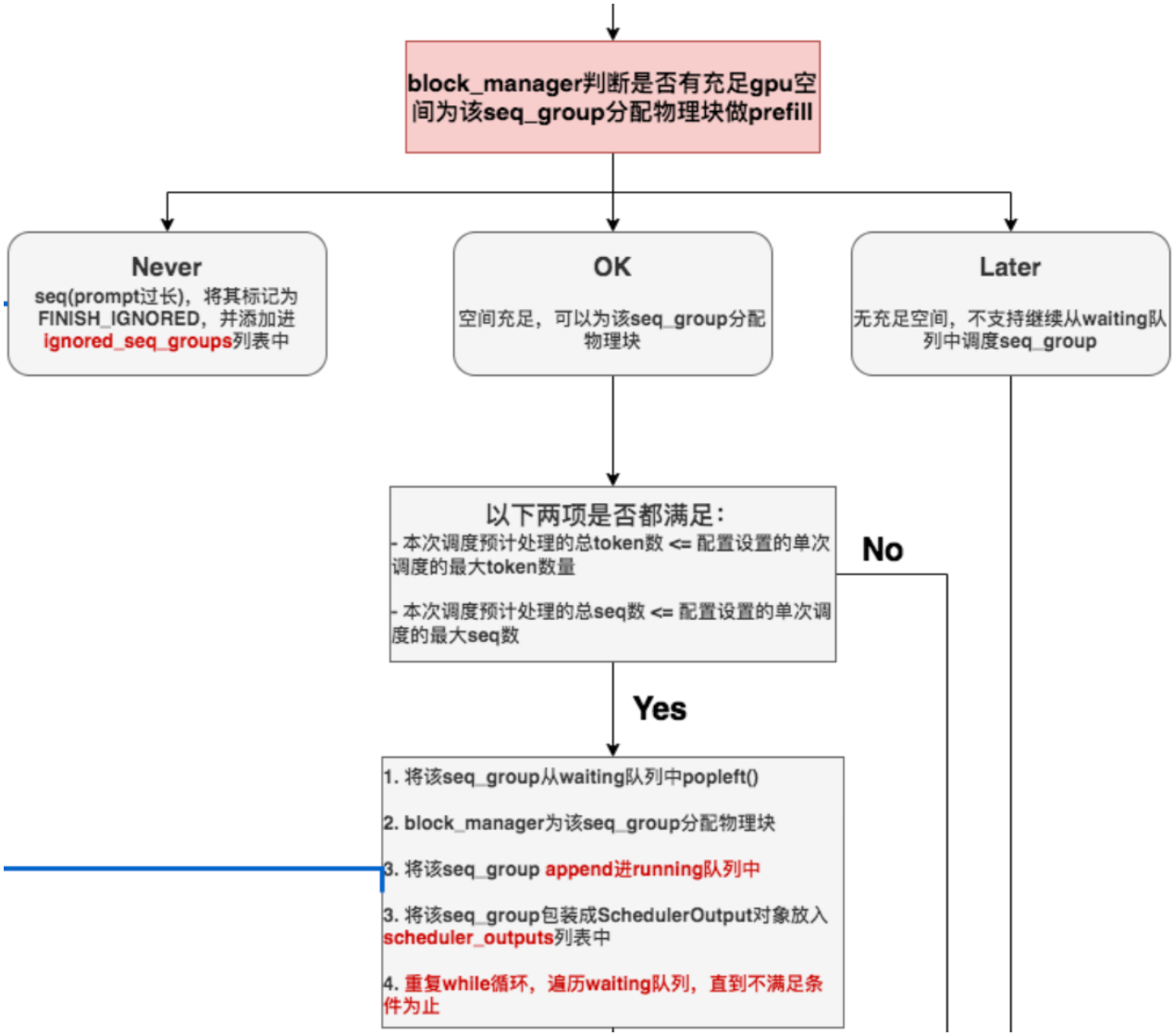
其中，BlockAllocator又分成两种类型：

- `CachedBlockAllocator`：**按照prefix caching的思想来分配和管理物理块。**在前面的介绍中，我们提过有些prompts中可能含有类似system message（例如，“假设你是一个能提供帮助的行车导航”）E）等prefix信息，带有这些相同prefix信息的prompt完全可以共享用于存放prefix的物理块，这样既节省显存，也不用再对prefix做推理。
- `UncachedBlockAllocator`：**正常分配和管理物理块，没有额外实现prefix caching的功能。**

6.2. UncachedBlockAllocator

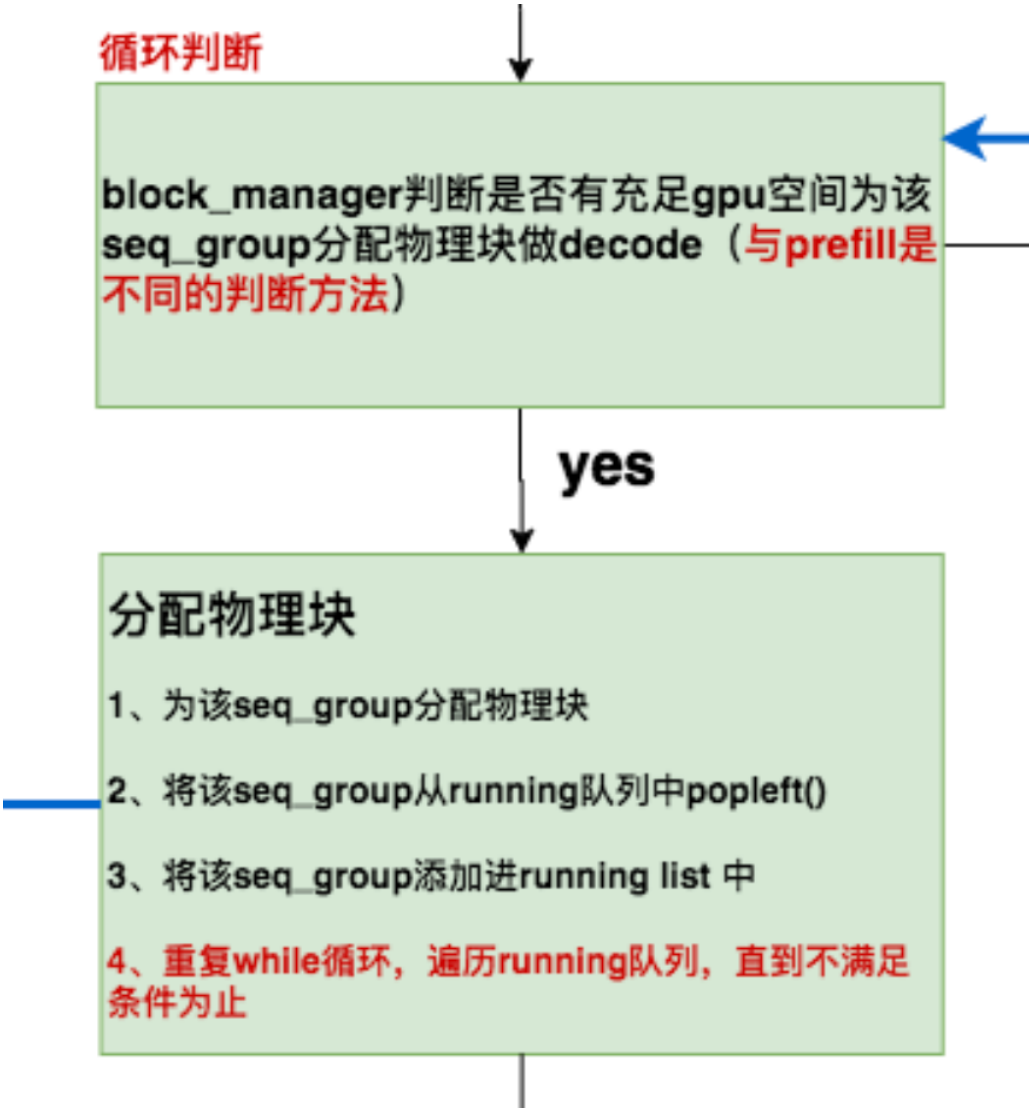
在整体调度策略的讲解中，我们明确了非常重要的一点：**在vllm的1个推理阶段，所有的seq_group要么一起做prefill，要么一起做decode。**这也意味着，**某次调度的结果，要么全部来自waiting队列（等待做prefill的），要么全部来自running或者running + swapped队列（等待做decode的）。**

6.2.1. 为waiting队列中的seq_group分配prefill需要的物理块



- 调用 `self.block_manager.can_allocate(seq_group)` 方法，**判断当前gpu上是否有充足的空间，能为当下这seq_group的prefill阶段分配充足的物理块**，用于装其KV Cache。
- 一旦我们认为当下空间充足，则调用 `self._allocate(seq_group)` 方法，**为waiting队列中的这个seq_group实际分配物理块**，这时我们就会运用到BlockAllocator，并且BlockAllocator的类型不同（即是否做prefix caching），allocate的方法也会不同。
- 这里我们做的只是**给定一种“物理块的分配方案”**，我们只是在制定这个seq_group可以使用哪些物理块，**但并没有实际往物理块中添加数据！“添加数据”这一步留到这1步推理实际开始时，由CacheEngine按照这个方案，往物理块中实际添加KV Cache。**

6.2.2. 为running/swapped队列中的seq_group分配decode需要的物理块



接下来我们考虑为running/swapped队列中的seq_group分配decode需要的物理块。

对于每个seq_group，在上1个推理阶段，我们对它下面的每个seq都产出了1个token。所以在这个推理阶段，我们判断能否为这些seq_group分配物理块时，我们也会分成两步：

- **调用 `self.block_manager.can_append_slot(seq_group)` 方法**，判断是否至少能为这个seq_group下的每个seq都分配1个空闲物理块。如果可以则认为能调度这个seq_group。
- **调用 `self._append_slot(seq_group, blocks_to_copy)` 方法**，实际分配物理块。

同样，在这里我们依然要强调，调度器中只是给出了物理块的分配方案，并没有实际往物理块中添加数据，添加数据这一步是CacheEngine照着这个方案来实际操作的。

6.3. CachedBlockAllocator

</> Bash

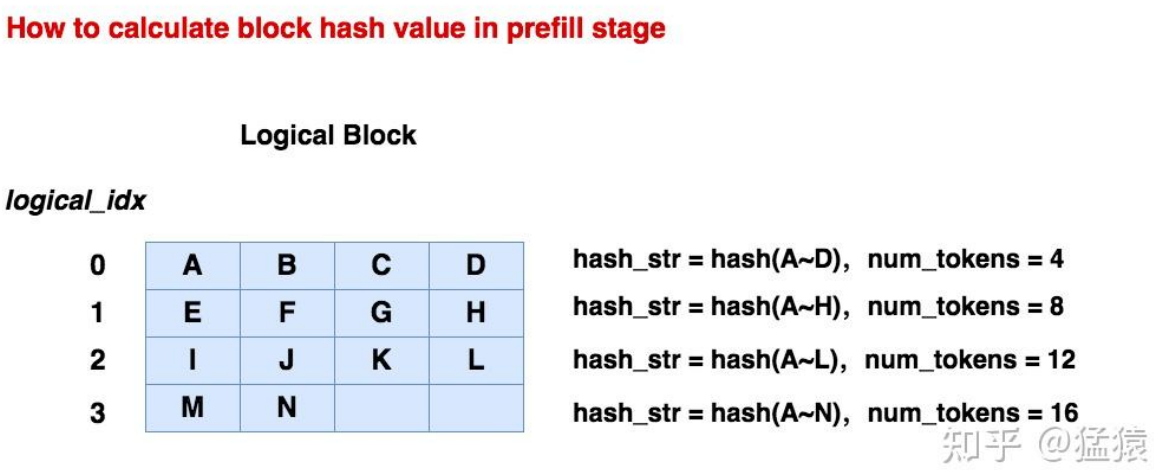
```
    #
    ================================================================================
    =======#
    ================================================================================
    =======
    # 如果做了prefix caching，即使用的是CachedBlockAllocator
    #
    ================================================================================
    =======
    elif self.enable_caching:
        block = self.gpu_allocator.allocate(
            seq.hash_of_block(logical_idx),
            seq.num_hashed_tokens_of_block(logical_idx))
```

6.3.1. hash值的计算

```
    block = self.gpu_allocator.allocate(
             seq.hash_of_block(logical_idx),
             seq.num_hashed_tokens_of_block(logical_idx))
```

`CachedBlockAllocator` **按照prefix caching的思想来分配和管理物理块**。在前面的介绍中，我们提过有些prompts中可能含有类似system message（例如，“假设你是一个能提供帮助的行车导航”）E）等prefix信息，带有这些相同prefix信息的prompt完全可以共享用于存放prefix的物理块，这样既节省显存，也不用再对prefix做推理。

hash值的计算就决定了这个prefix是否真的“相同”：当两个等待做prefill的seq拥有同样的hash值时，说明它们共享一样的prompt，这时就可以重复利用已有的KV cache。



hash的计算过程

```Bash
# 逻辑块0的计算:
num_tokens = 0 * 4 + 4 = 4
token_tuple = (A, B, C, D)  # 前4个token
hash_value = hash(( (A,B,C,D), lora_id ))

# 逻辑块1的计算:
num_tokens = 1 * 4 + 4 = 8
token_tuple = (A, B, C, D, E, F, G, H)  # 前8个token
hash_value = hash(( (A,B,C,D,E,F,G,H), lora_id ))

# 逻辑块2的计算:
num_tokens = 2 * 4 + 4 = 12
token_tuple = (A, B, C, D, E, F, G, H, I, J, K, L)
hash_value = hash((token_tuple, lora_id))

# 逻辑块3的计算
num_tokens = 3 * 4 + 4 = 16
token_tuple = (A, B, C, D, E, F, G, H, I, J, K, L, M, N)  # 实际只有14个
hash_value = hash((token_tuple, lora_id))
```

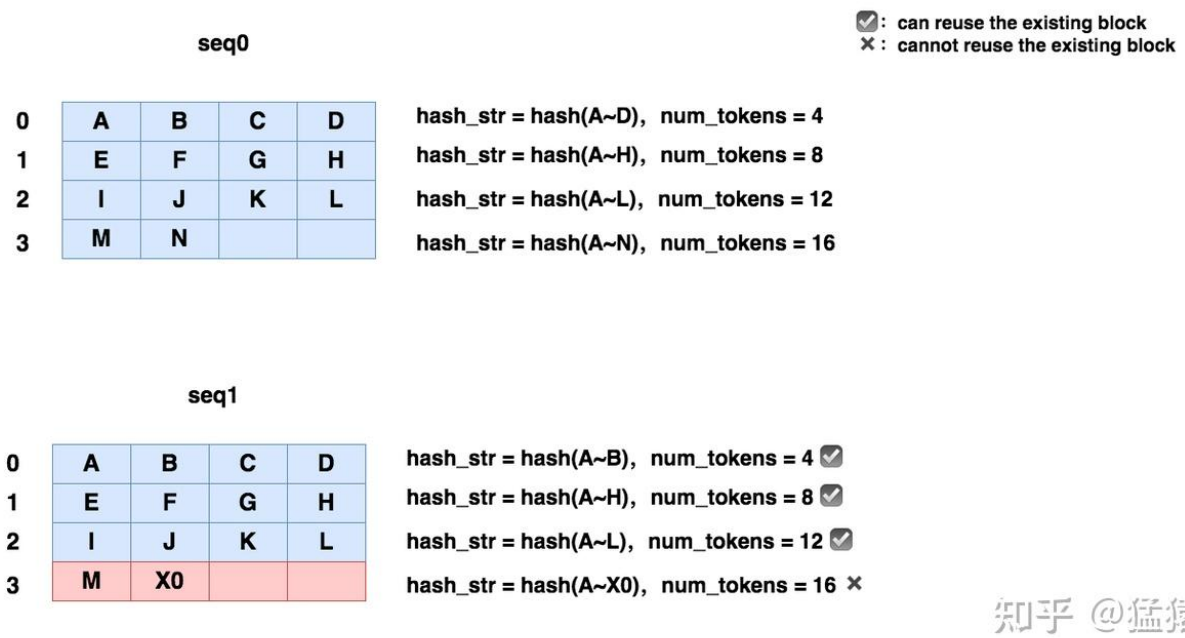
6.3.2. 使用evictor管理物理块分配细节

- 当一个物理块没有任何逻辑块引用时（例如一个seq刚做完整个推理），这时它理应被释放。**但是在prefix caching的前提下，我们的优化思想是**：即使这个物理块当前没有用武之地，可是如果不久之后来了一个新seq，它的prefix（例如system message）和这个物理块指向的内容高度一致，那么这个物理块就可以被重复使用，以此减少存储和计算开销。
- **所以，我们设置一个驱逐器（evictor）类，它的free_tables属性将用于存放这些暂时不用的物理块。**
- 此时，**该设备上全部可用的物理块 = *正在被使用/等待被使用的物理块数量* + *evictor的free_tables中的物理块数量***
- 在prefill阶段，当我们想创建一个物理块时，我们先算出这个物理块的hash值，然后**去free_tables中看**有没有可以重复利用的物理块，有则直接复用
- **如果没有可以重复利用的hash块，那这时我们先检查下这台设备剩余的空间是否够我们创建一个新物理块**。如果可以，就创建新物理块。
- 如果此时没有足够的空间创建新物理块，那么我们只好**从free_tables中驱除掉一个物理块**，为这个新的物理块腾出空间，**驱逐策略如下：**
  - 先根据**LRU（Least Recently Used）**原则，驱逐较老的那个物理块
  - 如果找到多个最后一次使用时间相同的老物理块，那么则**根据max_num_tokens原则**（max_num_tokens即为6.3.1图例中的num_tokens），驱逐其hash值计算中涵盖tokens最多的那个物理块。
  - **如果这些老物理块的LRU和max_num_tokens还是一致的话，那就从它们中随机驱逐一个**

6.3.3. prefix的标准

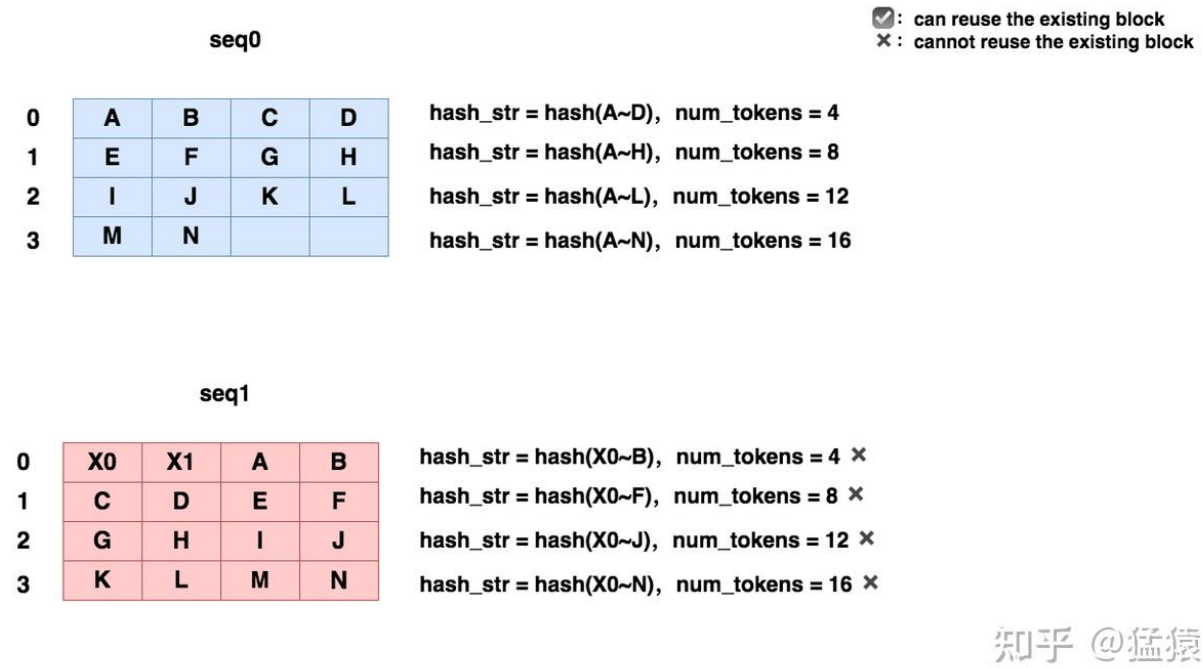
> 看到这里，也许有个想法一直在你脑中徘徊：“使用prefix caching，是不是就意味着两个seq的prompt必须完全一致，才可以重复利用物理块呢？”

下面我们再通过两个例子，帮助大家解答这个疑惑，也更好理解“prefix”的含义。



假设seq0现在做完了prefill，产出蓝色的4块物理块。现在进来一个seq1，我们想知道：seq1到底该怎么复用seq0的物理块?

- 当seq1刚进来时，我们先算好了它的逻辑块。现在要给每个逻辑块分配物理块。
- 对每个逻辑块，当我们决定是否要给它分配一个新的物理块时（一个新的物理块意味着占用了新的存储空间），我们先计算这个物理块的hash值。
- 按照这个流程，我们发现seq1的block0～2都可以复用seq0的（蓝色）
- 但是hash(seq1 block3) != hash(seq0 block3)，因此我们需要为seq1 block3（红色）开辟新空间。

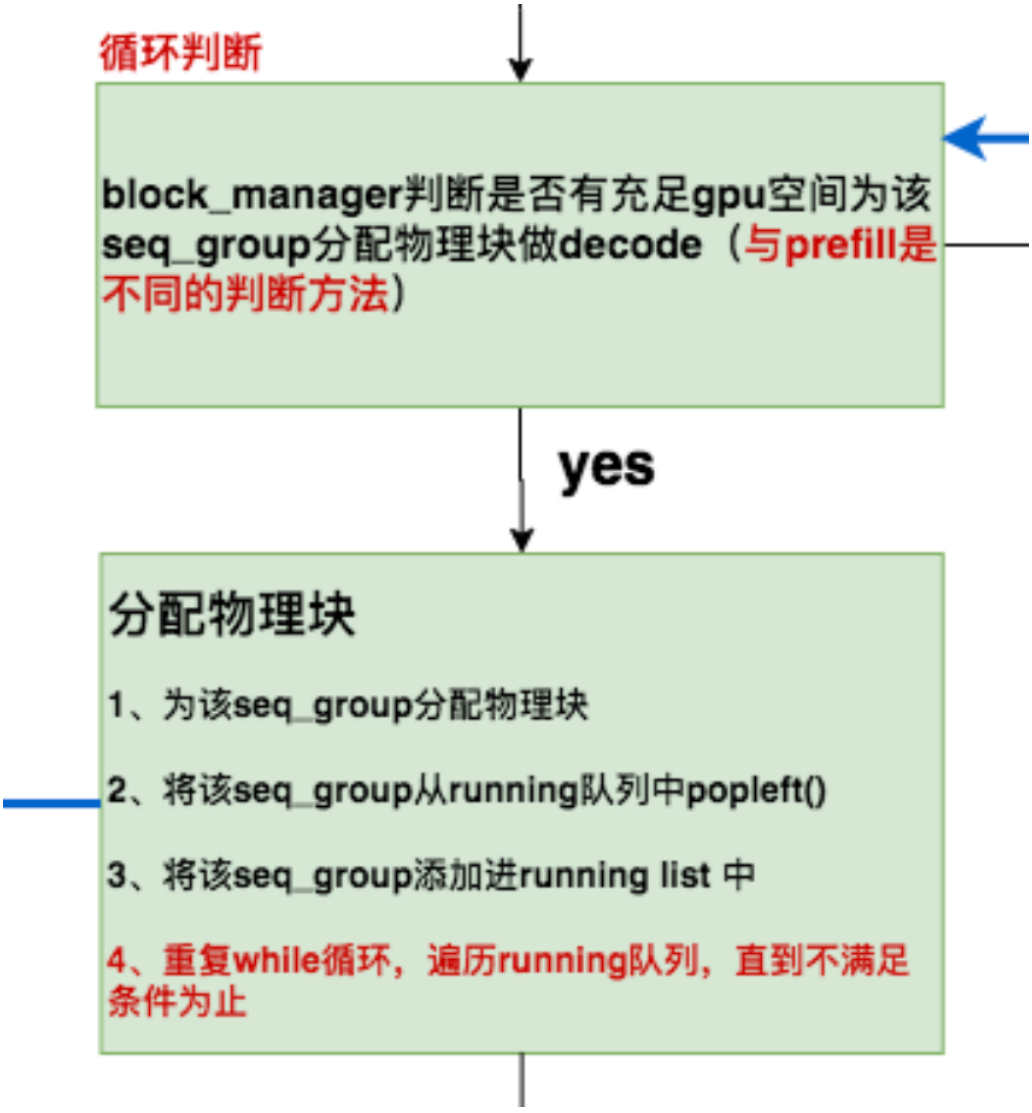


可以发现，尽管seq0和seq1的prompt的大部分内容是相同的，但是seq1依然不能复用seq0的prompt，**这是因为KV cache的计算也需要考虑位置编码的原因**。

通过例1和例2，你现在是否已更好了解我们只对prefix计算hash值的原因了呢？我们再小结一下vllm中hash值计算的一些要点：

- **vllm中，hash值的计算是block-level维度的**
- **vllm中，hash值的计算考虑了当前block及其之前所有block所维护的token值。这样做是为了找到最长可复用的prefix。**

6.3.4. decode阶段物理块的分配



在上面我们讲过UncachedBlockAllocator下为decode阶段分配物理块的方法（比较简单），**但是现在若使用CachedBlockAllocator，考虑物理块的复用问题时，情况就更复杂一些了。**

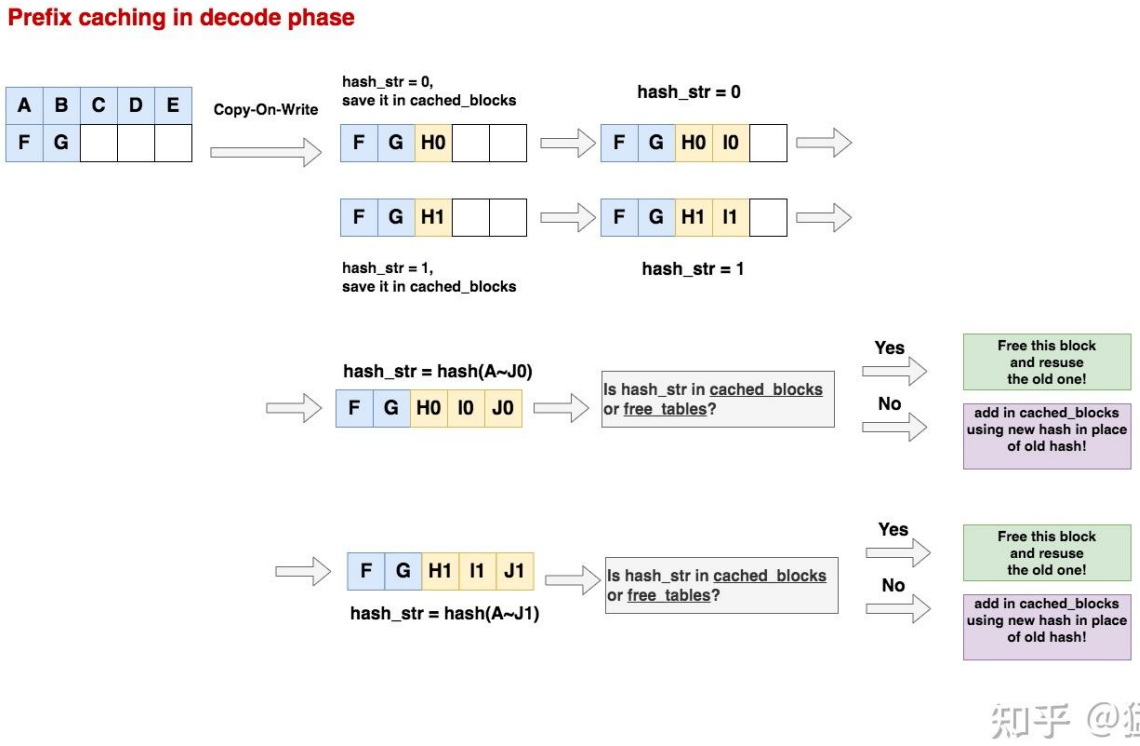
对于每个seq_group，在上1个推理阶段，我们对它下面的每个seq都产出了1个token。所以在这个推理阶段，我们判断能否为这些seq_group分配物理块时，我们也会分成两步：

- **（1）调用 `self.block_manager.can_append_slot(seq_group)` 方法**，判断是否至少能为这个seq_group下的每个seq都分配1个空闲物理块。如果可以则认为能调度这个seq_group。

- （2）调用 `self._append_slot(seq_group, blocks_to_copy)` 方法，实际分配物理块。

考虑下面这个使用parallel sampling做推理的例子，当n=2时，我们希望模型针对这一个prompt，产出2个推理序列结果：

```
# Parallel
Sampling("What is the meaning of life?",
SamplingParams(n=2, temperature=0.8, top_p=0.95,
frequency_penalty=0.1))
```



- **首先，seq正常做完prefill（蓝色部分），我们用黄色部分表示decode。**
- **开始做decode。根据copy-on-write机制**，FG所在的block1此时被两个子序列的逻辑块引用（ref_count = 2），所以它需要被拷贝一份。这样我们就得到2个物理块，用于装H0和H1。
- **在启动copy-on-write机制的同时，我们也要重新计算物理块的hash值**。和prefill阶段不同，在decode阶段，**当这个物理块还没满的时候，我们会给它附一个相互不重复的默认hash值**（`from itertools import count(), hash_str = next(iter(count()))`）。

- **我们会把附上hash值的物理块加入CachedBlockAllocator的 `cached_blocks` 属性中**（参见3.3节代码中的讲解），我们说过，这个属性用于记录当前正在被使用的物理块。
- 两个子序列继续做decode（风平浪静的美丽日子）
- **当一个子序列用完当前物理块的所有slots时（例如当子序列1生成J0后），我们再对这个物理块重新做hash计算，计算方式是hash(A~J0)。**
- **拿着这个new_hash，我们去cached_blocks（当前正在被使用的物理块列表）和free_tables（驱逐器的冷宫，曾经被使用的物理块列表）寻找。**看看这两者中是否存着相同hash值的物理块：
  - **如果找到可以复用的物理块，我们就释放当前这个物理块，复用旧物理块**
  - **如果没有找到可以复用的物理块，我们就把当前这个物理块的旧hash值从cached_blocks中释放掉，取而代之以新hash值。**

7. 参考文档

1. 猛猿：图解大模型计算加速系列之：vLLM核心技术PagedAttention原理
2. 猛猿：图解大模型计算加速系列：vLLM源码解析1，整体架构
3. 猛猿：图解大模型计算加速系列：vLLM源码解析2，调度器策略(Scheduler)
4. 猛猿：图解大模型计算加速系列：vLLM源码解析3，块管理器BlockManager（上篇）
5. 猛猿：图解大模型计算加速系列：vLLM源码解析3，Prefix Caching

🎁文末互动好礼🎁

1️⃣ 互动一

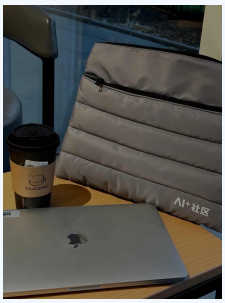
🤔在你的模型推理或服务部署实践中，**你最在意的是吞吐、时延，还是资源成本?**

🤔你是否用过 vLLM，或正在评估类似的推理框架？它解决了你哪些问题，又留下了哪些困惑?

2️⃣ 互动二

✍️**「周一创作者日」**已经陪大家走过了一段时间，新年新气象！为更好建设社区创作者专栏，欢迎各位在评论区留下你的直觉感受和建议，你的每一条反馈，都会成为我们栏目建设的重要参考💖

欢迎在评论区分享你的想法，我们将从留言中精选 **5 位走心分享的同学**，送出「AI+社区」电脑包，期待你的声音 👀✨



✨**恭喜以下五位同学获得AI＋社区电脑包**👜

@罗亮亮 @唐佳丽 @刘宇 @赵宗星 @v_李轩

随后会有运营同学拉群进行地址收集和奖品发放哦~