注：本文档按照step函数的运行顺序递归地进行函数说明，记录vLLM在推理时的运行方案。其中核心的组件scheduler和tokenizer需要详细说明其成员和运行方式，worker的运行代码将在其他文档说明。LLMEngine中还包含了其他如记录状态、加入请求等操作，需要时在其他文档中进行说明

LLMEngine.step()

执行一次iteration并返回新产生的token，其主要运行流程为

Step 1：调度在下一个iteration需要执行的序列(可能会进行抢占或重排)以及获得需要执行操作的tokens block(包括块的换入、换出和复制等)

Step 2：通知所有workers执行模型并产生输出

Step 3：处理模型输出，包括解码输出、更新被调度的序列组(beam_search等)、释放执行完成的序列组

```python
def step(self) -> List[RequestOutput]:
    # Step 1
    seq_group_metadata_list, scheduler_outputs = self.scheduler.schedule()

    if not scheduler_outputs.is_empty():
        # Step 2
        all_outputs = self._run_workers(
            "execute_model",
            driver_kwargs={
                "seq_group_metadata_list": seq_group_metadata_list,
                "blocks_to_swap_in": scheduler_outputs.blocks_to_swap_in,
                "blocks_to_swap_out": scheduler_outputs.blocks_to_swap_out,
                "blocks_to_copy": scheduler_outputs.blocks_to_copy,
            },
            use_ray_compiled_dag=USE_RAY_COMPILED_DAG)

        # Only the driver worker returns the sampling results.
        output = all_outputs[0]
    else:
        output = []

    # Step 3
    return self._process_model_outputs(output, scheduler_outputs)
```

- **scheduler.schedule()**

- **_run_workers()**

- **_process_model_outputs()**

  Step 1：如果prefix caching为True，则序列组中所有序列中的block都被标记为完成，使得新加入的请求不会再对其进行重新计算

  Step 2：处理每个序列组中的输出，主要为调用_process_sequence_group_outputs函数

  Step 3：将已经完成的序列组占有的块释放

  Step 4：从被调度的序列组和prompt长度大于模型支持长度的序列组(ignored_seq_groups)中获取请求的输出，得到一个包含RequestOutput对象的列表

  ```python
  def _process_model_outputs(self, output: SamplerOutput,scheduler_outputs: SchedulerOutputs) ->
  List[RequestOutput]:
      now = time.time()

      scheduled_seq_groups = scheduler_outputs.scheduled_seq_groups

      # Step 1
      if self.cache_config.enable_prefix_caching:
          for seq_group in scheduled_seq_groups:
              self.scheduler.mark_blocks_as_computed(seq_group)

      # Step 2
      for seq_group, outputs in zip(scheduled_seq_groups, output):
          self._process_sequence_group_outputs(seq_group, outputs)

      # Step 3
  ```

```
    self.scheduler.free_finished_seq_groups()

    # Step 4
    request_outputs: List[RequestOutput] = []
    for seq_group in scheduled_seq_groups:
        seq_group.maybe_set_first_token_time(now)
        request_output = RequestOutput.from_seq_group(seq_group)
        request_outputs.append(request_output)
    for seq_group in scheduler_outputs.ignored_seq_groups:
        request_output = RequestOutput.from_seq_group(seq_group)
        request_outputs.append(request_output)

    if self.log_stats:
        self.stat_logger.log(self._get_stats(scheduler_outputs))

    return request_outputs
```

1. **Scheduler.mark_blocks_as_computed(seq_group)**

   调用链路：

   Scheduler.mark_blocks_as_computed() -> **BlockSpaceManager.mark_blocks_as_computed()** -> **BlockSpaceManager.compute_full_blocks_in_seq()**

   具体操作：将序列当前所有的块都标记为已完成。这里采用reversed的方式从后往前遍历，避免冗余检查

   ```
   def compute_full_blocks_in_seq(self, seq: Sequence):
       if seq.seq_id not in self.block_tables:
           return
       max_full_block = seq.get_len() // self.block_size - 1
       block_table = self.block_tables[seq.seq_id]
       if max_full_block == -1:
           return
       for i in reversed(range(max_full_block)):
           if block_table[i].computed:
               break
           block_table[i].computed = True
   ```

2. **_process_sequence_group_outputs(seq_group, outputs)**

   Step 1：从序列组中随机选择一条序列，逐个token对prompt的token_id进行解码的带对应的text。这里利用了all_token_ids[:i]的原因应当是需要考虑position encoding以及前面的内容对token进行解码

   疑问：这里看得不是很懂，主要在prompt_logprobs上，每个prompt_logprobs的元素表示的是什么。推测每个元素的Dict都只包含了一个prompt token的Logprob

   Step 2：将所有的序列输出与对应的父序列用字典映射到一起，得到parent_child_dict

   Step 3：根据父序列(parent sequence)所包含的child数量，决定执行的操作。当该序列没有产生新的输出时，意味着该序列在未来不再会被使用，修改其状态为finished并释放空间；当序列产生超过一个输出时，将多余的输出转换为新的序列(调用fork函数)，并记录其与父序列的关系。最后一个sample的输出将放入父序列中继续进行后续的计算

   Step 4：对output中新产生的token以及logprob进行解码，并检查当前序列是否已经完成输出(主要为检查序列是否以预设值的终止序列或者终止符结尾，或者序列长度是否超过了最大长度)

   Step 5：对于非beam search的情况，vLLM将所有的子序列都分支为新的序列，其主要通过调用scheduler的folk函数实现(后面会详细说明)。同时，该步骤将已完成的序列进行资源回收，但这些序列仍旧会保存在序列组中作为可能的输出

   疑问：由于新的token只被分配了逻辑块，这里的folk函数只需要采用物理块表的深拷贝操作。其物理块的分配什么时候进行

   Step 6：对于beam search的情况，vLLM首先在已完成的序列中选择最优的beam_width条序列，将新完成的序列加入了sequence group中，将beam分数低的序列从sequence group中移出

   Step 7：对正在运行的序列根据他们的beam分数进行排序

   Step 8：判断是否需要提前结束beam search，主要调用_check_beam_search_early_stopping函数，对比正在运行的最优序列和已完成的最差序列的分数，当最差序列的分数已经大于当前正在运行的序列的最大可获得分数，那么提前结束beam search

   Step 9：如果停止beam search进程，则将所有正在运行的序列放入unselected_child_seqs中表示不再需要；如果不暂停，则选择前beam_width条序列继续运行，丢弃其余的序列

   Step 10：将被选到的新生成的序列(selected_child_seq中)进行folk操作，拷贝其父序列的物理块表，并加入到序列组seq group中；将已经完成的序列进行资源回收；对没有被选到的序列进行资源回收，并将它们移出seq_group(unselected_child_seq)

   ```
   def _process_sequence_group_outputs(self, seq_group: SequenceGroup,
                                           outputs: SequenceGroupOutput) -> None:
   ```

```
# Step 1
prompt_logprobs = outputs.prompt_logprobs
if prompt_logprobs is not None:
    seq = next(iter(seq_group.seqs_dict.values()))
    all_token_ids = seq.get_token_ids()
    for i, prompt_logprobs_for_token in enumerate(prompt_logprobs):
        self._decode_logprobs(seq, seq_group.sampling_params,
                              prompt_logprobs_for_token,
                              all_token_ids[:i])
    seq_group.prompt_logprobs = prompt_logprobs

# Step 2
samples = outputs.samples
parent_seqs = seq_group.get_seqs(status=SequenceStatus.RUNNING)
existing_finished_seqs = seq_group.get_finished_seqs()
parent_child_dict = {
    parent_seq.seq_id: []
    for parent_seq in parent_seqs
}
for sample in samples:
    parent_child_dict[sample.parent_seq_id].append(sample)
# List of (child, parent)
child_seqs: List[Tuple[Sequence, Sequence]] = []

# Step 3
for parent in parent_seqs:
    child_samples: List[SequenceOutput] = parent_child_dict[
        parent.seq_id]
    if len(child_samples) == 0:
        parent.status = SequenceStatus.FINISHED_ABORTED
        seq_group.remove(parent.seq_id)
        self.scheduler.free_seq(parent)
        continue
    for child_sample in child_samples[:-1]:
        new_child_seq_id = next(self.seq_counter)
        child = parent.fork(new_child_seq_id)
        child.append_token_id(child_sample.output_token,
                              child_sample.logprobs)
        child_seqs.append((child, parent))
    last_child_sample = child_samples[-1]
    parent.append_token_id(last_child_sample.output_token,
                           last_child_sample.logprobs)
    child_seqs.append((parent, parent))

# Step 4
for seq, _ in child_seqs:
    self._decode_sequence(seq, seq_group.sampling_params)
    self._check_stop(seq, seq_group.sampling_params)

# Step 5
if not seq_group.sampling_params.use_beam_search:
    for seq, parent in child_seqs:
        if seq is not parent:
            seq_group.add(seq)
            if not seq.is_finished():
                self.scheduler.fork_seq(parent, seq)
    for seq, parent in child_seqs:
        if seq is parent and seq.is_finished():
            self.scheduler.free_seq(seq)
    return

selected_child_seqs = []
unselected_child_seqs = []
beam_width = seq_group.sampling_params.best_of # 选择最优的beam_width个结果
length_penalty = seq_group.sampling_params.length_penalty

# Step 6
existing_finished_seqs = [(seq, None, False)
                          for seq in existing_finished_seqs]
new_finished_seqs = [(seq, parent, True) for seq, parent in child_seqs
                     if seq.is_finished()]
all_finished_seqs = existing_finished_seqs + new_finished_seqs
# 根据beam分数对已完成的sequence进行排序
all_finished_seqs.sort(key=lambda x: x[0].get_beam_search_score(
    length_penalty=length_penalty, eos_token_id=x[0].eos_token_id),
                       reverse=True)
for seq, parent, is_new in all_finished_seqs[:beam_width]:
    if is_new:
        # 新的高分序列将其加入到sequence group中
        selected_child_seqs.append((seq, parent))
for seq, parent, is_new in all_finished_seqs[beam_width:]:
    if is_new:
        # 新的低分序列不会被加入到sequence group中
        unselected_child_seqs.append((seq, parent))
    else:
        # 原本已完成的低分序列从sequence group中移出
        seq_group.remove(seq.seq_id)

# Step 7
running_child_seqs = [(seq, parent) for seq, parent in child_seqs
                      if not seq.is_finished()]
running_child_seqs.sort(key=lambda x: x[0].get_beam_search_score(
    length_penalty=length_penalty, eos_token_id=x[0].eos_token_id),
                        reverse=True)

# Step 8
if len(running_child_seqs) == 0:
    stop_beam_search = True
elif len(all_finished_seqs) < beam_width:
    stop_beam_search = False
else:
    best_running_seq = running_child_seqs[0][0]
    current_worst_seq = all_finished_seqs[beam_width - 1][0]
    stop_beam_search = self._check_beam_search_early_stopping(
        seq_group.sampling_params.early_stopping,
        seq_group.sampling_params, best_running_seq, current_worst_seq)

# Step 9
if stop_beam_search:
    unselected_child_seqs.extend(running_child_seqs)
else:
    selected_child_seqs.extend(running_child_seqs[:beam_width])
    unselected_child_seqs.extend(running_child_seqs[beam_width:])

# Step 10
for seq, parent in selected_child_seqs:
    if seq is not parent:
        seq_group.add(seq)
        if not seq.is_finished():
            self.scheduler.fork_seq(parent, seq)

for seq, parent in selected_child_seqs:
    if seq is parent and seq.is_finished():
        self.scheduler.free_seq(seq)

for seq, parent in unselected_child_seqs:
    if seq is parent:
        seq_group.remove(seq.seq_id)
        self.scheduler.free_seq(seq)
```

- Scheduler.fork_seq()

  调用链路：**Scheduler.fork_seq()->Blockspacemanager.fork_seq()**

  子序列从父序列复制而来，目前共享已有的物理块，因此直接将父序列的物理块表复制给子序列即可

  ```
  def fork(self, parent_seq: Sequence, child_seq: Sequence) -> None:
      src_block_table = self.block_tables[parent_seq.seq_id]
      self.block_tables[child_seq.seq_id] = src_block_table.copy()
      for block in set(src_block_table):
          block.ref_count += 1
  ```

- 关于_check_beam_search_early_stopping参考的指标sampling_params.early_stopping

  ```
  early_stopping: True表示一旦有beam_width个输出序列则停止生成；False表示当剩余序列的分数很可能不如当前已完成序列时停止
  生成，很不可能表示为当前序列分数已经小于已完成序列的最差分数；Never表示只有剩余序列的分数一定不如当前已完成序列时停止生成
  ```

3. **scheduler.free_finished_seq_groups()**

将标记为已完成的序列组从调度器的running队列中移除

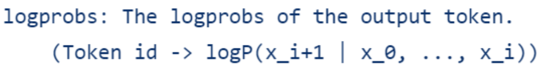
```
def free_finished_seq_groups(self) -> None:
    self.running = deque(seq_group for seq_group in self.running
                         if not seq_group.is_finished())
```

4. **RequestOutput.from_seq_group(seq_group)**

该函数主要从seq_group和seq类中获取序列的输出信息，并以RequestOutput的形式返回

这里不理解的点：logprobs和output_logprobs指的是什么？

应当是输出token时计算出的概率(对于第i个token，其logprob的值为$P(x_i|x_0,x_1,...,x_{i-1})$)



```
logprobs: The logprobs of the output token.
    (Token id -> logP(x_i+1 | x_0, ..., x_i))
```

Step 1：根据序列组中的序列数量以及采样参数，获取前n条最合适的序列。如果是采用beam search方案，选择score最高的n条序列；否则选择累计logprob最高的n条序列

Step 2：产生输出，其主要为调用CompletionOutput的构造函数得到完整输出。这里说明了需要根据采样参数中的logprobs参数，选择是否将logprobs值加入到输出中

Step 3：增加prompt序列，完成时间等信息，调用RequestOutput本身的构造函数(这里cls表示自身的构造函数)得到序列组的输出结果

```
def from_seq_group(cls, seq_group: SequenceGroup) -> "RequestOutput":
    seqs = seq_group.get_seqs()
    # Step 1
    if len(seqs) == 1:
        top_n_seqs = seqs
    else:
        n = seq_group.sampling_params.n
        if seq_group.sampling_params.use_beam_search:
            sorting_key = lambda seq: seq.get_beam_search_score(
                seq_group.sampling_params.length_penalty)
        else:
            sorting_key = lambda seq: seq.get_cumulative_logprob()
        sorted_seqs = sorted(seqs, key=sorting_key, reverse=True)
        top_n_seqs = sorted_seqs[:n]

    # Step 2
    # NOTE: We need omit logprobs here explicitly because the sequence
    # always has the logprobs of the sampled tokens even if the
    # logprobs are not requested.
    include_logprobs = seq_group.sampling_params.logprobs
    outputs = [
        CompletionOutput(seqs.index(seq), seq.output_text,
                         seq.get_output_token_ids(),
                         seq.get_cumulative_logprob(),
                         seq.output_logprobs if include_logprobs else None,
                         SequenceStatus.get_finished_reason(seq.status))
        for seq in top_n_seqs
    ]

    # Step 3
    prompt = seq_group.prompt
    prompt_token_ids = seq_group.prompt_token_ids
    prompt_logprobs = seq_group.prompt_logprobs
    finished = seq_group.is_finished()
    finished_time = time.time() if finished else None
    seq_group.set_finished_time(finished_time)
    return cls(seq_group.request_id,
               prompt,
               prompt_token_ids,
               prompt_logprobs,
               outputs,
               finished,
               seq_group.metrics,
               lora_request=seq_group.lora_request)
```