**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

```
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()**

  Step 1：调用自身的_schedule函数得到调度器输出，其输出包括了被调度的序列组(scheduled_seq_groups)、是否为prompt阶段(prompt_run)、一次iteration同时执行的请求数量(num_batched_tokens)、需要执行换入换出复制操作的块(blocks_to_swap_in、blocks_to_swap_out、blocks_to_copy)、被忽略的序列组(ignored_seq_groups)

  Step 2：生成seq_group的元数据，记录了各个seq_group的请求id、是否为prompt阶段、正在运行的序列的数据(Dict[seq_id, SequenceData])、正在运行的序列的块表、序列组中共同的已被计算的块、seq_group的状态(通常不用)

  ```
  def schedule(self) -> Tuple[List[SequenceGroupMetadata], SchedulerOutputs]:
      # Step 1
      scheduler_outputs = self._schedule()
      now = time.time()

      # Step 2
      seq_group_metadata_list: List[SequenceGroupMetadata] = []
      for seq_group in scheduler_outputs.scheduled_seq_groups:
          # 设置首次被调用的时间
          seq_group.maybe_set_first_scheduled_time(now)

          seq_data: Dict[int, SequenceData] = {}
          block_tables: Dict[int, List[int]] = {}

          # 获得正在运行的序列的序列数据和块表
          for seq in seq_group.get_seqs(status=SequenceStatus.RUNNING):
              seq_id = seq.seq_id
              seq_data[seq_id] = seq.data
              block_tables[seq_id] = self.block_manager.get_block_table(seq)
              # 设置序列的块最后被访问的时间
              self.block_manager.access_all_blocks_in_seq(seq, now)

          seq_group_metadata = SequenceGroupMetadata(
              request_id=seq_group.request_id,
  ```

```
            is_prompt=scheduler_outputs.prompt_run,
            seq_data=seq_data,
            sampling_params=seq_group.sampling_params,
            block_tables=block_tables,
            lora_request=seq_group.lora_request,
            # 计算出seq_group中所有共享的已被计算的块
            computed_block_nums=self.block_manager.
            get_common_computed_block_ids(seq_group),
            state=seq_group.state,
        )
        seq_group_metadata_list.append(seq_group_metadata)
    return seq_group_metadata_list, scheduler_outputs
```

**_schedule()**

Step 1：当交换队列中不存在请求时，尝试将等待的请求换入运行(存在交换出去的请求时不准新的等待请求进入内存运行)。

Step 2：该步骤逐条请求查看等待队列中的请求是否满足换入的要求，对每条请求进行的判断包括：请求prompt的长度要小于调度器的限制；调用can_allocate函数判断是否有足够的剩余空间换入请求(如果返回值为LATER，由于先进先出的原则，直接退出循环，表示第一条请求无法被换入时不考虑后续请求的换入；如果返回值为NEVER，则表示请求不会满足换入条件)。对于不满足换入条件的请求，将其标记为ignored的终止态，加入到ignored列表中。此外，vLLM还控制了一次性换入的请求的token数量(控制该batch的token数量)、正在运行的最大请求数量以及当前batch请求的padding token数量不超过调度器配置中的限制值，避免后续频繁地换入和换出。对于满足要求的请求，调用allocate函数为它们分配内存

Step 3：这一步判断是否有waiting的请求被换入，这些请求处于prefill阶段。当存在waiting的请求时，优先令他们运行结束prefill阶段，再与其他running请求一起执行autoregressive阶段。

Step 4：判断当前的剩余空闲块能否容纳所有running阶段的请求的执行，如果没有时选择请求抢占其空间。首先遍历所有序列组，调用BlockSpaceManager.can_append_slot()函数，判断剩余空闲块能否容纳序列组中的所有序列的token(这里采用了简单的方式，即序列数量少于空闲块数量即可，而不是细致地判定每个块中是否有空的token位置)。

如果可以容纳序列组所有的序列，调用append_slot函数为每个序列的新的token分配位置；如果无法容纳序列组所有的序列，则调用_preemt函数逐出优先级最低(到达时间最晚)的序列组(如果当前序列组优先级最低，则将本序列组逐出，并break出循环)

Step 5：如果当前iteration没有序列被逐出，则根据FIFO顺序尝试将换出的序列换回。调度器遍历swapped队列，并逐个判断序列组是否有足够的空间将其换入(判定条件为序列组包含的所有块+为每个序列分配一个空闲块<=gpu中现有的空闲块)。对于可换入的序列组，调用_swap_in函数将其块换入，并调用_append_slot函数为序列组中的所有序列的新token分配一个槽位(为新的块分配空间，旧的块存在共享时进行复制，旧的块满时重新计算哈希)

Step 6：生成对应的调度器输出

```
def _schedule(self) -> SchedulerOutputs:
    # Blocks that need to be swapped or copied before model execution.
    blocks_to_swap_in: Dict[int, int] = {}
    blocks_to_swap_out: Dict[int, int] = {}
    blocks_to_copy: Dict[int, List[int]] = {}

    # Fix the current time.
    now = time.monotonic()

    # Step 1
    if not self.swapped:
        ignored_seq_groups: List[SequenceGroup] = []
        scheduled: List[SequenceGroup] = []
        # The total number of sequences on the fly, including the
        # requests in the generation phase.
        num_curr_seqs = sum(seq_group.get_max_num_running_seqs()
                            for seq_group in self.running)
        curr_loras = set(
            seq_group.lora_int_id
            for seq_group in self.running) if self.lora_enabled else None
        seq_lens: List[int] = []

        leftover_waiting_sequences = deque()
        # Step 2
        while self.waiting:
            seq_group = self.waiting[0]
            waiting_seqs = seq_group.get_seqs(
                status=SequenceStatus.WAITING)
            assert len(waiting_seqs) == 1, (
                "Waiting sequence group should have only one prompt "
                "sequence.")
            # 留意到这里直接取的是sequence长度而不是prompt，原因在于recompute的序列也被放入了                waiting队列中，其prompt相当于原来的prompt和所有已生成的token
```

```
            num_prompt_tokens = waiting_seqs[0].get_len()
            if num_prompt_tokens > self.prompt_limit:
                logger.warning(
                    f"Input prompt ({num_prompt_tokens} tokens) is too long"
                    f" and exceeds limit of {self.prompt_limit}")
                for seq in waiting_seqs:
                    seq.status = SequenceStatus.FINISHED_IGNORED
                ignored_seq_groups.append(seq_group)
                self.waiting.popleft()
                continue

            # If the sequence group cannot be allocated, stop.
            can_allocate = self.block_manager.can_allocate(seq_group)
            if can_allocate == AllocStatus.LATER:
                break
            elif can_allocate == AllocStatus.NEVER:
                logger.warning(
                    f"Input prompt ({num_prompt_tokens} tokens) is too long"
                    f" and exceeds the capacity of block_manager")
                for seq in waiting_seqs:
                    seq.status = SequenceStatus.FINISHED_IGNORED
                ignored_seq_groups.append(seq_group)
                self.waiting.popleft()
                continue

            lora_int_id = 0
            if self.lora_enabled:
                lora_int_id = seq_group.lora_int_id
                if (lora_int_id > 0 and lora_int_id not in curr_loras
                        and len(curr_loras) >= self.lora_config.max_loras):
                    leftover_waiting_sequences.appendleft(seq_group)
                    self.waiting.popleft()
                    continue

            new_seq_lens = seq_lens + [num_prompt_tokens]
            num_batched_tokens = len(new_seq_lens) * max(new_seq_lens)
            if (num_batched_tokens >
                    self.scheduler_config.max_num_batched_tokens):
                break

            num_new_seqs = seq_group.get_max_num_running_seqs()
            if (num_curr_seqs + num_new_seqs >
                    self.scheduler_config.max_num_seqs):
                break

            num_paddings = num_batched_tokens - sum(new_seq_lens)
            if num_paddings > self.scheduler_config.max_paddings:
                break
            seq_lens = new_seq_lens

            if lora_int_id > 0:
                curr_loras.add(lora_int_id)
            self.waiting.popleft()
            self._allocate(seq_group)
            self.running.append(seq_group)
            num_curr_seqs += num_new_seqs
            scheduled.append(seq_group)
        self.waiting.extendleft(leftover_waiting_sequences)

        # Step 3
        if scheduled or ignored_seq_groups:
            scheduler_outputs = SchedulerOutputs(
                scheduled_seq_groups=scheduled,
                prompt_run=True,
                num_batched_tokens=len(seq_lens) *
                max(seq_lens) if seq_lens else 0,
                blocks_to_swap_in=blocks_to_swap_in,
                blocks_to_swap_out=blocks_to_swap_out,
                blocks_to_copy=blocks_to_copy,
                ignored_seq_groups=ignored_seq_groups,
            )
            return scheduler_outputs

    # Step 4
```

```
# 先进先出的原则，按照到达时间顺序对请求进行排序
self.running = self.policy.sort_by_priority(now, self.running)

running: Deque[SequenceGroup] = deque()
preempted: List[SequenceGroup] = []
while self.running:
    seq_group = self.running.popleft()
    while not self.block_manager.can_append_slot(seq_group):
        # 驱逐优先级最低的序列组，没有则驱逐当前序列组
        if self.running:
            victim_seq_group = self.running.pop()
            self._preempt(victim_seq_group, blocks_to_swap_out)
            preempted.append(victim_seq_group)
        else:
            self._preempt(seq_group, blocks_to_swap_out)
            preempted.append(seq_group)
            break
    else:
        self._append_slot(seq_group, blocks_to_copy)
        running.append(seq_group)
self.running = running

# Step 5
self.swapped = self.policy.sort_by_priority(now, self.swapped)
# 判断是否有序列被逐出
if not preempted:
    num_curr_seqs = sum(seq_group.get_max_num_running_seqs()
                        for seq_group in self.running)
    curr_loras = set(
        seq_group.lora_int_id
        for seq_group in self.running) if self.lora_enabled else None

    leftover_swapped = deque()

    while self.swapped:
        seq_group = self.swapped[0]
        lora_int_id = 0
        if self.lora_enabled:
            lora_int_id = seq_group.lora_int_id
            if (lora_int_id > 0 and lora_int_id not in curr_loras
                    and len(curr_loras) >= self.lora_config.max_loras):
                leftover_swapped.appendleft(seq_group)
                self.swapped.popleft()
                continue

        # 没有足够的空闲块分配给当前seq_group，退出循环
        if not self.block_manager.can_swap_in(seq_group):
            break

        # 保障运行序列的总数不会超过调度器支持的最大数量
        num_new_seqs = seq_group.get_max_num_running_seqs()
        if (num_curr_seqs + num_new_seqs >
                self.scheduler_config.max_num_seqs):
            break

        if lora_int_id > 0:
            curr_loras.add(lora_int_id)
        self.swapped.popleft()
        self._swap_in(seq_group, blocks_to_swap_in)
        self._append_slot(seq_group, blocks_to_copy)
        num_curr_seqs += num_new_seqs
        self.running.append(seq_group)

    self.swapped.extendleft(leftover_swapped)

# Step 6
# 每次iteration序列只生成一个token，因此组织成批的token数量与序列数量一致
num_batched_tokens = sum(
    seq_group.num_seqs(status=SequenceStatus.RUNNING)
    for seq_group in self.running)

# 这里prompt_run设置为了False，表示执行的是autoregressive阶段
scheduler_outputs = SchedulerOutputs(
    scheduled_seq_groups=self.running,
```

```
        prompt_run=False,
        num_batched_tokens=num_batched_tokens,
        blocks_to_swap_in=blocks_to_swap_in,
        blocks_to_swap_out=blocks_to_swap_out,
        blocks_to_copy=blocks_to_copy,
        ignored_seq_groups=[],
    )
    return scheduler_outputs
```

- **BlockSpaceManager.can_allocate(SequenceGroup)**

  该函数用于判定序列组是否能够被换入GPU中运行，**做出的假设是seq_group中的所有seq都共享了相同的prompt**(因此这里的num_required_blocks仅仅等于第一条序列需要的块数量)。如果序列需要的块大于了整个gpu可分配的块，则永不调度该序列，否则当gpu中有足够空闲空间时，可以将序列换入运行

```
def can_allocate(self, seq_group: SequenceGroup) -> AllocStatus:
    seq = seq_group.get_seqs(status=SequenceStatus.WAITING)[0]
    num_required_blocks = len(seq.logical_token_blocks)

    if self.block_sliding_window is not None:
        num_required_blocks = min(num_required_blocks,
                                  self.block_sliding_window)
    num_free_gpu_blocks = self.gpu_allocator.get_num_free_blocks()

    if (self.num_total_gpu_blocks - num_required_blocks <
            self.watermark_blocks):
        return AllocStatus.NEVER
    if num_free_gpu_blocks - num_required_blocks >= self.watermark_blocks:
        return AllocStatus.OK
    else:
        return AllocStatus.LATER
```

- **_allocate()->BlockSpaceManager.allocate()**

  调用BlockSpaceManager.allocate(SequenceGroup)，并修改序列组中的所有序列状态为running。**allocate函数同样假设了换入的SequenceGroup有相同的前缀**，因此只需要对某条序列进行内存分配，并将其块表复制给所有其他的序列即可。该函数进一步调用了BlockAllocator.allocate函数，进行块的分配。**留意到这里采用了hash_of_block函数来进行哈希，这个函数这里只会作用到prompt或者已经填满的块上，而prompt长度不会变，这里计算出的哈希值不会发生变化。如果块没有满，则会用default_hash_ctr来取哈希值**

  **疑问：这里的block_sliding_window表示的是什么，为什么要这么做**

  **疑问：采用default_hash_ctr来取哈希值时，后续如何利用哈希值获取对应的物理块**

```
def allocate(self, seq_group: SequenceGroup) -> None:
    # NOTE: Here we assume that all sequences in the group have the same
    # prompt.
    seq = seq_group.get_seqs(status=SequenceStatus.WAITING)[0]

    # Allocate new physical token blocks that will store the prompt tokens.
    num_prompt_blocks = len(seq.logical_token_blocks)

    block_table: BlockTable = []
    for logical_idx in range(num_prompt_blocks):
        if (self.block_sliding_window is not None
                and logical_idx >= self.block_sliding_window):
            block = block_table[logical_idx % self.block_sliding_window]
        else:
            block = self.gpu_allocator.allocate(
                seq.hash_of_block(logical_idx),
                seq.num_hashed_tokens_of_block(logical_idx))
        block_table.append(block)

    # Assign the block table for each sequence.
    for seq in seq_group.get_seqs(status=SequenceStatus.WAITING):
        self.block_tables[seq.seq_id] = block_table.copy()
```

- **BlockAllocator.allocate -> BlockAllocator.allocate_block**

  当不支持caching操作时，allocator直接调用allocate_block函数进行块的分配，其主要操作为当设备上存在物理块没有被分配过时，直接分配序号最小的一个未分配过的物理块；当设备上的所有物理块都被分配过，则调用evictor的evict函数分配一个物理块

  当支持caching操作时，allocator尝试从cached表中找到当前哈希值缓存的物理块。如果没有找到，则分配一个新的物理块

  **疑问：什么时候会出现调用allocate函数时对应哈希值被cached_blocks表记录的情况，可能是某个序列已经被逐出，但是它的物理块数据没有被其他的序列占用(可能是其他序列有完全相同的prompt或者序列)**

```
def allocate_block(self, block_hash: int, num_hashed_tokens: int) -> PhysicalTokenBlock:
    if self.current_num_blocks == self.num_blocks:
        block = self.evictor.evict()
        block.block_hash = block_hash
        block.num_hashed_tokens = num_hashed_tokens
        return block
    block = PhysicalTokenBlock(device=self.device,
                               block_number=self.current_num_blocks,
                               block_size=self.block_size,
                               block_hash=block_hash,
                               num_hashed_tokens=num_hashed_tokens)
    self.current_num_blocks += 1
    return block

def allocate(self, block_hash: Optional[int] = None, num_hashed_tokens: int = 0) -> PhysicalTokenBlock:
    # If caching is disabled, just allocate a new block and return it
    if not self.enable_caching:
        block = self.allocate_block(next(self.default_hash_ctr),
                                    num_hashed_tokens)
        block.ref_count += 1
        return block

    if block_hash is None:
        block_hash = next(self.default_hash_ctr)
    if block_hash in self.evictor:
        assert block_hash not in self.cached_blocks
        block = self.evictor.remove(block_hash)
        assert block.ref_count == 0
        self.cached_blocks[block_hash] = block
        block.ref_count += 1
        assert block.block_hash == block_hash
        return block
    if block_hash not in self.cached_blocks:
        self.cached_blocks[block_hash] = self.allocate_block(
            block_hash, num_hashed_tokens)
    block = self.cached_blocks[block_hash]
    assert block.block_hash == block_hash
    block.ref_count += 1
    return block
```

- **Evictor.evict()**

  不同的Evictor有不同的驱逐方案，这里只举LRU的例子进行分析。该方案主要选择的是最长时间未被访问到的块，将其内容进行替换。该函数并不负责在没有空闲块时将序列逐出腾出内存空间，仅仅只是在空闲块中选择最合适的分配给请求。

  LRU方案首先找到最久未被访问到的块(记录为last_accessed最小)，并在其中找到前缀token数量最多的块，将其返回

```
def evict(self) -> PhysicalTokenBlock:
    free_blocks: List[PhysicalTokenBlock] = list(self.free_table.values())
    if len(free_blocks) == 0:
        raise ValueError("No usable cache memory left")

    # 找到最长时间未被访问的块
    lowest_timestamp = free_blocks[0].last_accessed
    for block in free_blocks:
        if block.last_accessed < lowest_timestamp:
            lowest_timestamp = block.last_accessed

    least_recent: List[PhysicalTokenBlock] = []
    for block in free_blocks:
        if block.last_accessed == lowest_timestamp:
            least_recent.append(block)
```

```
        # 找到前缀长度最长的块
        highest_num_hashed_tokens = 0
        for block in least_recent:
            if block.num_hashed_tokens > highest_num_hashed_tokens:
                highest_num_hashed_tokens = block.num_hashed_tokens

        evicted_block: Optional[PhysicalTokenBlock] = None
        for block in least_recent:
            if block.num_hashed_tokens == highest_num_hashed_tokens:
                evicted_block = block
                break

        assert evicted_block is not None

        del self.free_table[evicted_block.block_hash]

        evicted_block.computed = False
        return evicted_block
```

- **_append_slot()->BlockSpaceManager.append_slot()->_allocate_last_physical_block()**

  append_slot()函数为新的token分配一个物理槽。其首先判定序列是否为新的token分配了新的逻辑块，如果有则创建新的物理块并建立逻辑块到新的物理块的映射(调用_allocate_last_physical_block函数)，否则检查最后一个逻辑块的引用数量。

  如果最后一个逻辑块引用数量大于1，说明当前逻辑块被多个序列引用，需要为当前序列复制一个新的物理块并代替块表中原来的物理块的引用(Copy on Write策略)；如果最后一个逻辑块引用数量为1，则判断其是否已满，如果满了则为其计算新的哈希值(_maybe_promote_last_block函数调用hash_of_block函数)，使得该块可以被其他序列共享(否则块的哈希值是用的default，其计算出来的数不会重复。在块完成后，其哈希值是固定的，可以被查找得到)

  将需要进行复制的块以[src, [dst]]的形式存储在blocks_to_copy中，用以指导后续的复制操作。

```
def _append_slot(self, seq_group: SequenceGroup,blocks_to_copy: Dict[int, List[int]],) -> None:
    for seq in seq_group.get_seqs(status=SequenceStatus.RUNNING):
        ret = self.block_manager.append_slot(seq)
        if ret is not None:
            src_block, dst_block = ret
            if src_block in blocks_to_copy:
                blocks_to_copy[src_block].append(dst_block)
            else:
                blocks_to_copy[src_block] = [dst_block]

def append_slot(self,seq: Sequence) -> Optional[Tuple[int, int]]:
    """Allocate a physical slot for a new token."""
    logical_blocks = seq.logical_token_blocks
    block_table = self.block_tables[seq.seq_id]
    if len(block_table) < len(logical_blocks):
        assert len(block_table) == len(logical_blocks) - 1

        if (self.block_sliding_window
                and len(block_table) >= self.block_sliding_window):
            block_table.append(block_table[len(block_table) %
                                           self.block_sliding_window])
        else:
            new_block = self._allocate_last_physical_block(seq)
            block_table.append(new_block)
            return None

    last_block = block_table[-1]
    assert last_block.device == Device.GPU
    if last_block.ref_count == 1:
        # 如果最后一个块满了，为其重新计算哈希值，使其能够被共享
        new_block = self._maybe_promote_last_block(seq, last_block)
        block_table[-1] = new_block
        return None
    else:
        new_block = self._allocate_last_physical_block(seq)
        block_table[-1] = new_block
        self.gpu_allocator.free(last_block)
        return last_block.block_number, new_block.block_number

def _allocate_last_physical_block(self, seq: Sequence) -> PhysicalTokenBlock:
    block_hash: Optional[int] = None
```

```
    # 留意到只有当最后一个块是满的时，才会调用hash_of_block，否则由于token发生变化，同个块的哈希值可能发生变化。在block_hash为None时，allocate函数会自动分配哈希值
    if (self._is_last_block_full(seq)):
        block_hash = seq.hash_of_block(len(seq.logical_token_blocks) - 1)
    num_hashed_tokens = seq.num_hashed_tokens_of_block(
        len(seq.logical_token_blocks) - 1)
    new_block = self.gpu_allocator.allocate(block_hash, num_hashed_tokens)
    if block_hash is None:
        assert new_block.ref_count == 1
    return new_block
```

- **_preempt()**

  根据驱逐策略选择相应的驱逐操作。如果没有给定驱逐策略，则根据序列组的序列数量选择操作(只有一条序列时选择recompute，否则选择swapping)

  当采用recompute策略时，调用BlockSpaceManager.free()函数将序列组所有的物理块释放，将序列转换为waiting状态，并放到等待队列的最左侧(此时，被逐出的序列组相当于未被调度的状态，只是其逻辑块表和sequence包含了已生成的所有token，根据FIFO规则放到了waiting队列最前面)

  当采用swap策略时，将序列组的所有正在运行的序列调整为SWAPPED状态，计算出所有需要换出的块以及其对应的在CPU上的映射，并将序列组加入swapped队列中。如果CPU中没有足够的块分配给换出的序列组，则报出运行错误。所有的被换出的块的到CPU块的映射关系以blocks_to_swap_out函数返回

```
def _preempt(self,
    seq_group: SequenceGroup,
    blocks_to_swap_out: Dict[int, int],
    preemption_mode: Optional[PreemptionMode] = None,
) -> None:
    if preemption_mode is None:
        if seq_group.get_max_num_running_seqs() == 1:
            preemption_mode = PreemptionMode.RECOMPUTE
        else:
            preemption_mode = PreemptionMode.SWAP
    if preemption_mode == PreemptionMode.RECOMPUTE:
        self._preempt_by_recompute(seq_group)
    elif preemption_mode == PreemptionMode.SWAP:
        self._preempt_by_swap(seq_group, blocks_to_swap_out)
    else:
        raise AssertionError("Invalid preemption mode.")

def _preempt_by_recompute(self, seq_group: SequenceGroup) -> None:
    seqs = seq_group.get_seqs(status=SequenceStatus.RUNNING)
    assert len(seqs) == 1
    for seq in seqs:
        seq.status = SequenceStatus.WAITING
        self.block_manager.free(seq)
    self.waiting.appendleft(seq_group)

def _preempt_by_swap(self,seq_group: SequenceGroup,blocks_to_swap_out: Dict[int, int]) -> None:
    self._swap_out(seq_group, blocks_to_swap_out)
    self.swapped.append(seq_group)

def _swap_out(self,seq_group: SequenceGroup,blocks_to_swap_out: Dict[int, int],) -> None:
    if not self.block_manager.can_swap_out(seq_group):
        raise RuntimeError(
            "Aborted due to the lack of CPU swap space. Please increase "
            "the swap space to avoid this error.")
    mapping = self.block_manager.swap_out(seq_group)
    blocks_to_swap_out.update(mapping)
    for seq in seq_group.get_seqs(status=SequenceStatus.RUNNING):
        seq.status = SequenceStatus.SWAPPED

def swap_out(self, seq_group: SequenceGroup) -> Dict[int, int]:
    # 建立起GPU上被换出的块到CPU块的映射关系
    mapping: Dict[PhysicalTokenBlock, PhysicalTokenBlock] = {}
    for seq in seq_group.get_seqs(status=SequenceStatus.RUNNING):
        new_block_table: BlockTable = []
        block_table = self.block_tables[seq.seq_id]

        for gpu_block in block_table:
            if gpu_block in mapping:
            # 当前块是个被共享的块，直接更新对应块的引用次数ref_count
```

```
                cpu_block = mapping[gpu_block]
                cpu_block.ref_count += 1
            else:
            # 当前的块未被转换，从cpu_allocator中分配一个新的块，采用的哈希值与GPU的块相同
                cpu_block = self.cpu_allocator.allocate(
                    gpu_block.block_hash, gpu_block.num_hashed_tokens)
                mapping[gpu_block] = cpu_block
            new_block_table.append(cpu_block)
            # 释放被换出的GPU块，主要为将它们加入到evitor的空闲块中记录
            self.gpu_allocator.free(gpu_block)
        self.block_tables[seq.seq_id] = new_block_table

    block_number_mapping = {
        gpu_block.block_number: cpu_block.block_number
        for gpu_block, cpu_block in mapping.items()
    }
    return block_number_mapping
```

- **_swap_in()->BlockSpaceManager.swap_in()**

  换入和换出操作类似，均用mapping记录CPU块到新分配的物理块的映射，并将换入的序列状态转为RUNNING，将对应的序列组放入running队列中

```
def _swap_in(
    self,
    seq_group: SequenceGroup,
    blocks_to_swap_in: Dict[int, int],
) -> None:
    mapping = self.block_manager.swap_in(seq_group)
    blocks_to_swap_in.update(mapping)
    for seq in seq_group.get_seqs(status=SequenceStatus.SWAPPED):
        seq.status = SequenceStatus.RUNNING

def swap_in(self, seq_group: SequenceGroup) -> Dict[int, int]:
    mapping: Dict[PhysicalTokenBlock, PhysicalTokenBlock] = {}
    for seq in seq_group.get_seqs(status=SequenceStatus.SWAPPED):
        new_block_table: BlockTable = []
        block_table = self.block_tables[seq.seq_id]

        for cpu_block in block_table:
            if cpu_block in mapping:
            # 当前的块是被共享的块，增加ref_count即可
                gpu_block = mapping[cpu_block]
                gpu_block.ref_count += 1
            else:
            # 当前的块未被共享，为其分配一个新的gpu块
                gpu_block = self.gpu_allocator.allocate(
                    cpu_block.block_hash, cpu_block.num_hashed_tokens)
                mapping[cpu_block] = gpu_block
            new_block_table.append(gpu_block)
            # 释放CPU中对应的块
            self.cpu_allocator.free(cpu_block)
        self.block_tables[seq.seq_id] = new_block_table

    block_number_mapping = {
        cpu_block.block_number: gpu_block.block_number
        for cpu_block, gpu_block in mapping.items()
    }
    return block_number_mapping
```

- **_run_workers()**

  函数利用ray在所有worker上运行指定的方法。在step中，其操作所有worker执行"execute_model"方法，传递的参数包括seq_group_metadata_list(每一项包含seq_group的id、是否为prompt阶段、各条序列的数据、采样参数以及块表)、blocks_to_swap_in(需要从cpu换入的块)、blocks_to_swap_out(需要从gpu换出的块)、blocks_to_copy(需要进行拷贝的块)

  **疑问：worker和drive worker在执行上区别在哪里**

```
def _run_workers(
    self,
    method: str,
    *args,
```

```
    driver_args: Optional[List[Any]] = None,
    driver_kwargs: Optional[Dict[str, Any]] = None,
    max_concurrent_workers: Optional[int] = None,
    **kwargs,
) -> Any:

    if max_concurrent_workers:
        raise NotImplementedError(
            "max_concurrent_workers is not supported yet.")

    # 令所有ray worker执行对应的method函数，得到输出结果。ray worker不接收块表、metadata等信息，这些等待由driver worker将这些参数进行广播
    ray_worker_outputs = [
        worker.execute_method.remote(method, *args, **kwargs)
        for worker in self.workers
    ]

    if driver_args is None:
        driver_args = args
    if driver_kwargs is None:
        driver_kwargs = kwargs

    # 调用本地的worker(保存为driver_worker)执行method函数
    driver_worker_output = getattr(self.driver_worker,
                                   method)(*driver_args, **driver_kwargs)

    # 获得drive worker以及其他所有的ray worker的运行结果
    if self.workers:
        ray_worker_outputs = ray.get(ray_worker_outputs)

    return [driver_worker_output] + ray_worker_outputs
```

- **_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prompt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

```
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
    # child=parent的序列表示已在seq_group中的序列，不需要分配块表
    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)

    # 由于child属于finished时，它没有被分配块表，因此只需要回收parent的资源
    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$P(x_i|x_0, x_1, \ldots, x_{i-1})$)**

```
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)
```