Worker执行：Worker.execute_model()

注：这里采用的名词分划(partition)指的是执行当前函数的进程，对应了参与张量并行的一个GPU

Step 1：由于ray worker的调用没有传递参数，其序列组的metadata以及需要进行交换的块由driver_worker调用broadcast函数将这些数据广播给所有的ray worker，其主要通过调用torch.distributed.broadcast_object_list()函数实现，src为0号worker(driver worker)。

Step 2：调用CacheEngine的swap_in、swap_out以及copy函数进行块的交换操作，其中swap_in和swap_out均通过CacheEngine._swap()实现

Step 3：执行模型获得对应的输出

```
def execute_model(
    self,
    seq_group_metadata_list: Optional[List[SequenceGroupMetadata]] = None,
    blocks_to_swap_in: Optional[Dict[int, int]] = None,
    blocks_to_swap_out: Optional[Dict[int, int]] = None,
    blocks_to_copy: Optional[Dict[int, List[int]]] = None,
) -> Optional[SamplerOutput]:
    # Step 1: driver worker将序列组信息以及需要换入换出的块数据广播到所有的ray worker上
    if self.is_driver_worker:
        assert seq_group_metadata_list is not None
        num_seq_groups = len(seq_group_metadata_list)
        assert blocks_to_swap_in is not None
        assert blocks_to_swap_out is not None
        assert blocks_to_copy is not None
        block_swapping_info = [blocks_to_swap_in, blocks_to_swap_out, blocks_to_copy]
        broadcast_object_list([num_seq_groups] + block_swapping_info, src=0)
    else:
        recv_data = [None] * 4
        broadcast_object_list(recv_data, src=0)
        num_seq_groups = recv_data[0]
        block_swapping_info = recv_data[1:]
    # Step 2: 执行块的交换与复制，主要为调用CacheEngine对应的函数
    self.cache_swap(*block_swapping_info)
    # 如果没有输入序列，则直接返回空值
    if num_seq_groups == 0:
        return {}
    # Step 3: 调用model_runner的函数执行模型
    output = self.model_runner.execute_model(seq_group_metadata_list,self.gpu_cache)
    return output
```

- CacheEngine._swap()

  函数根据src_to_dst中记录的块号，将src(如换入时为GPU block)的key和value块交换到dst(如换入时为CPU block)中，并将对应的时间戳记录在cache流中(换入和换出不会同时进行，只需要一个event数组进行记录即可，copy操作不使用流进行记录)，后续调用event.wait等待记录的事件执行完毕

```
def _swap(
    self,
    src: List[KVCache],
    dst: List[KVCache],
    src_to_dst: Dict[int, int],
) -> None:
    # 在cache_stream的上下文下运行代码
    with torch.cuda.stream(self.cache_stream):
        for i in range(self.num_layers):
            src_key_cache, src_value_cache = src[i]
            dst_key_cache, dst_value_cache = dst[i]
            # 执行交换操作
            cache_ops.swap_blocks(src_key_cache, dst_key_cache, src_to_dst)
            cache_ops.swap_blocks(src_value_cache, dst_value_cache, src_to_dst)
            # 在cache_stream上记录事件，用于后续等待事件完成
            event = self.events[i]
            event.record(stream=self.cache_stream)
```

  该函数调用了cache_ops.swap_blocks函数(cpp函数)，其定义在vllm/csrc/cache_kernels.cu中，用于调用cuda的函数进行内存交换。

  Step 1：根据src和dst的device类型，决定cuda需要执行的内存拷贝类型(如cudaMemcpyDeviceToHost表示从GPU拷贝到CPU内存)，保存在memcpy_type中

Step 2：调用cudaMemcpyAsync函数执行拷贝操作，并将对应拷贝事件记录在CUDA stream中。

gpu_cache记录tensor的指针指向的应当是gpu内存的地址，而cpu_cache指向的应当是cpu的内存地址

```
void swap_blocks(
    torch::Tensor& src,
    torch::Tensor& dst,
    const std::map<int64_t, int64_t>& block_mapping) {
    // Step 1
    torch::Device src_device = src.device();
    torch::Device dst_device = dst.device();
    cudaMemcpyKind memcpy_type;
    if (src_device.is_cuda() && dst_device.is_cuda()) {
    // gpu的内存拷贝只能在同一个gpu上进行
        TORCH_CHECK(src_device.index() == dst_device.index(), "src and dst must be on the same GPU");
        memcpy_type = cudaMemcpyDeviceToDevice;
    } else if (src_device.is_cuda() && dst_device.is_cpu()) {
        memcpy_type = cudaMemcpyDeviceToHost;
    } else if (src_device.is_cpu() && dst_device.is_cuda()) {
        memcpy_type = cudaMemcpyHostToDevice;
    } else {
        TORCH_CHECK(false, "Invalid device combination");
    }
    // Step 2
    char *src_ptr = static_cast<char*>(src.data_ptr());
    char *dst_ptr = static_cast<char*>(dst.data_ptr());

    // 计算出每个块的大小，用于计算不同块的内存偏移量
    const int64_t block_size_in_bytes = src.element_size() * src[0].numel();
    // 设置当前CUDA上下文设备，如src采用0号GPU时，则设置CUDA上下文设备为0号GPU
    const at::cuda::OptionalCUDAGuard device_guard(src_device.is_cuda() ? src_device : dst_device);
    // CUDA stream记录拷贝事件
    const cudaStream_t stream = at::cuda::getCurrentCUDAStream();
    for (const auto& pair : block_mapping) {
        int64_t src_block_number = pair.first;
        int64_t dst_block_number = pair.second;
        // 计算出对应GPU和CPU块的偏移量，相当于块号*块的大小
        int64_t src_offset = src_block_number * block_size_in_bytes;
        int64_t dst_offset = dst_block_number * block_size_in_bytes;
        cudaMemcpyAsync(
          dst_ptr + dst_offset,
          src_ptr + src_offset,
          block_size_in_bytes, // 拷贝的数据量
          memcpy_type, // 拷贝类型
          stream, // 拷贝操作保存在当前CUDA流中
        );
    }
}
```

- CacheEngine.copy()

  函数根据src_to_dst中记录的块号，将src的key和value块拷贝到dst列表的所有块中

```
def copy(self, src_to_dsts: Dict[int, List[int]]) -> None:
    key_caches = [key_cache for key_cache, _ in self.gpu_cache]
    value_caches = [value_cache for _, value_cache in self.gpu_cache]
    cache_ops.copy_blocks(key_caches, value_caches, src_to_dsts)
```

该函数调用了cache_ops.copy_blocks函数，定义在vllm/csrc/cache_kernels.cu中，用于调用cuda的函数进行内存拷贝。注意这里传入的参数为gpu_cache，因此需要逐层考虑进行拷贝(python采用的是引用，key_cache和value_cache记录的都是对应层的数据的指针)。

疑问：函数转为张量前为什么要先进行int64的转换，是否是为了统一数据类型

```
void copy_blocks(
  std::vector<torch::Tensor>& key_caches,
  std::vector<torch::Tensor>& value_caches,
  const std::map<int64_t, std::vector<int64_t>>& block_mapping) {
  // 检测key和value的大小是否一致，检测采用的设备是否为cuda
  int num_layers = key_caches.size();
  TORCH_CHECK(num_layers == value_caches.size());
  if (num_layers == 0) {
    return;
```

```
  }
  torch::Device cache_device = key_caches[0].device();
  TORCH_CHECK(cache_device.is_cuda());
  // 创建数组，用强制转换将每层的指针保存在int64类型数组中，推测是为了避免不同数据类型带来的影响，并方便后续的数据传输
  int64_t key_cache_ptrs[num_layers];
  int64_t value_cache_ptrs[num_layers];
  for (int layer_idx = 0; layer_idx < num_layers; ++layer_idx) {
    key_cache_ptrs[layer_idx] = reinterpret_cast<int64_t>(key_caches[layer_idx].data_ptr());
    value_cache_ptrs[layer_idx] = reinterpret_cast<int64_t>(value_caches[layer_idx].data_ptr());
  }
  // 创建块映射关系，映射关系表中每两个元素为一对{src, dst}映射
  std::vector<int64_t> block_mapping_vec;
  for (const auto& pair : block_mapping) {
    int64_t src_block_number = pair.first;
    for (int64_t dst_block_number : pair.second) {
      block_mapping_vec.push_back(src_block_number);
      block_mapping_vec.push_back(dst_block_number);
    }
  }
  int64_t* block_mapping_array = block_mapping_vec.data();
  int num_pairs = block_mapping_vec.size() / 2;
  // 将存储指针的数组和块的映射关系表转为tensor并移动到GPU上
  torch::Tensor key_cache_ptrs_tensor = torch::from_blob(
    key_cache_ptrs, {num_layers}, torch::kInt64).to(cache_device);
  torch::Tensor value_cache_ptrs_tensor = torch::from_blob(
    value_cache_ptrs, {num_layers}, torch::kInt64).to(cache_device);
  torch::Tensor block_mapping_tensor = torch::from_blob(
    block_mapping_array, {2 * num_pairs}, torch::kInt64).to(cache_device);
  // 启动copy_blocks_kernel进行数据拷贝，注意这里grid和block讲述了线程块的排列方式，每个线程块负责某一块的拷贝，每个线程负责块的某个元素的拷贝。VLLM_DISPATCH_FLOATING_AND_BYTE_TYPES传入的第一个参数为拷贝的数据类型，将内核中定义的template转换为该数据类型
  const int numel_per_block = key_caches[0][0].numel();
  dim3 grid(num_layers, num_pairs);
  dim3 block(std::min(1024, numel_per_block));
  // cuda上下文转换为cache_device
  const at::cuda::OptionalCUDAGuard device_guard(cache_device);
  // 拷贝时间保存在当前的cuda流中，用于执行同步操作
  const cudaStream_t stream = at::cuda::getCurrentCUDAStream();
  VLLM_DISPATCH_FLOATING_AND_BYTE_TYPES(
    key_caches[0].scalar_type(), "copy_blocks_kernel", ([&] {
      vllm::copy_blocks_kernel<scalar_t><<<grid, block, 0, stream>>>(
        key_cache_ptrs_tensor.data_ptr<int64_t>(),
        value_cache_ptrs_tensor.data_ptr<int64_t>(),
        block_mapping_tensor.data_ptr<int64_t>(),
        numel_per_block);
    }));
}
```

该函数调用了vLLM自定义的内核copy_blocks_kernel，将多个copy操作(cudaMemcpyAsync)组织成批，并通过一次内核调用执行所有拷贝操作。其主要原因在于拷贝操作通常会在不连续的块上执行，导致会触发多次小的data movement操作，浪费了cuda kernel的启动时间(详见论文的implementation)。**非异步的CUDA内核函数在退出前会自动进行同步操作，等待所有线程完成计算**

```
// Grid:(num_layers, num_pairs), Block:(numel_per_block)
template<typename scalar_t>
__global__ void copy_blocks_kernel(
  int64_t* key_cache_ptrs,
  int64_t* value_cache_ptrs,
  const int64_t* __restrict__ block_mapping,
  const int numel_per_block) {
  // 根据block编号选择进行拷贝的层和块，一个线程块负责一层的一个块的拷贝
  const int layer_idx = blockIdx.x;
  const int pair_idx = blockIdx.y;
  // 将数据转换回原有的数据类型scalar_t，注意这里转换的是对应层的指针(ptrs的layer_idx层)
  scalar_t* key_cache = reinterpret_cast<scalar_t*>(key_cache_ptrs[layer_idx]);
  scalar_t* value_cache = reinterpret_cast<scalar_t*>(value_cache_ptrs[layer_idx]);
  // 当前线程块负责拷贝的src块号和dst块号
  int64_t src_block_number = block_mapping[2 * pair_idx];
  int64_t dst_block_number = block_mapping[2 * pair_idx + 1];
  // src和dst块相对于key_cache的偏移量
  const int64_t src_block_offset = src_block_number * numel_per_block;
  const int64_t dst_block_offset = dst_block_number * numel_per_block;
  for (int i = threadIdx.x; i < numel_per_block; i += blockDim.x) {
    // 每个线程负责拷贝的元素的偏移量，每个线程负责一个元素的拷贝
    int64_t src_offset = src_block_offset + i;
```

```
    int64_t dst_offset = dst_block_offset + i;
    key_cache[dst_offset] = key_cache[src_offset];
  }
  for (int i = threadIdx.x; i < numel_per_block; i += blockDim.x) {
    // value块与key块的拷贝同理
    int64_t src_offset = src_block_offset + i;
    int64_t dst_offset = dst_block_offset + i;
    value_cache[dst_offset] = value_cache[src_offset];
  }
}
```

- ModelRunner.execute_model()

  Step 1：调用prepare_input_tensors()函数，准备输入到模型的输入tensor和位置tensor，并得到输入的元数据和采样的元数据

  Step 2：执行一次模型的forward函数，得到模型的输出结果

  Step 3：执行模型的sample函数，采样获得下一个token

```
def execute_model(
    self,
    seq_group_metadata_list: Optional[List[SequenceGroupMetadata]],
    kv_caches: List[Tuple[torch.Tensor, torch.Tensor]],
) -> Optional[SamplerOutput]:
    # Step 1
    input_tokens, input_positions, input_metadata, sampling_metadata = (
        self.prepare_input_tensors(seq_group_metadata_list))
    # Step 2
    if input_metadata.use_cuda_graph:
        graph_batch_size = input_tokens.shape[0]
        model_executable = self.graph_runners[graph_batch_size]
    else:
        model_executable = self.model
    hidden_states = model_executable(
        input_ids=input_tokens,
        positions=input_positions,
        kv_caches=kv_caches,
        input_metadata=input_metadata,
    )
    # Step 3
    output = self.model.sample(
        hidden_states=hidden_states,
        sampling_metadata=sampling_metadata,
    )
    return output
```

  1. **prepare_input_tensors()**

     driver worker根据序列组处于prompt还是decoding阶段(根据调度器的代码，我们知道序列组中的序列只会处于其中一个阶段)，分别调用不同的prepare函数从序列组的元数据信息中提取出输入token等信息，并同时提取出采样需要的元数据，通过broadcast_object_list函数将所有的信息广播给所有的ray worker

```
def prepare_input_tensors(
    self,
    seq_group_metadata_list: Optional[List[SequenceGroupMetadata]],
) -> Tuple[torch.Tensor, torch.Tensor, InputMetadata, SamplingMetadata]:
    if self.is_driver_worker:
        is_prompt = seq_group_metadata_list[0].is_prompt
        # 根据序列组所处的阶段进行不同的输入准备
        if is_prompt:
            (input_tokens, input_positions, input_metadata,
             prompt_lens) = self._prepare_prompt(seq_group_metadata_list)
        else:
            (input_tokens, input_positions, input_metadata
             ) = self._prepare_decode(seq_group_metadata_list)
            prompt_lens = []
        sampling_metadata = self._prepare_sample(seq_group_metadata_list,
                                                 prompt_lens)
        def get_size_or_none(x: Optional[torch.Tensor]):
            return x.size() if x is not None else None
        # 将输入数据广播给所有的ray worker。这里首先广播各个数据的数据量，并根据数据量是否为0决定是否广播某项数据(如块表、上下文长度等)
        py_data = {
```

```python
            "input_tokens_size":
            input_tokens.size(),
            "input_positions_size":
            input_positions.size(),
            "is_prompt":
            input_metadata.is_prompt,
            "slot_mapping_size":
            get_size_or_none(input_metadata.slot_mapping),
            "max_context_len":
            input_metadata.max_context_len,
            "context_lens_size":
            get_size_or_none(input_metadata.context_lens),
            "block_tables_size":
            get_size_or_none(input_metadata.block_tables),
            "use_cuda_graph":
            input_metadata.use_cuda_graph,
            "selected_token_indices_size":
            sampling_metadata.selected_token_indices.size(),
        }
        broadcast_object_list([py_data], src=0)
        broadcast(input_tokens, src=0)
        broadcast(input_positions, src=0)
        if input_metadata.slot_mapping is not None:
            broadcast(input_metadata.slot_mapping, src=0)
        if input_metadata.context_lens is not None:
            broadcast(input_metadata.context_lens, src=0)
        if input_metadata.block_tables is not None:
            broadcast(input_metadata.block_tables, src=0)
        broadcast(sampling_metadata.selected_token_indices, src=0)
    else:
        # ray worker根据各个数据的size接收不同的数据
        receving_list = [None]
        broadcast_object_list(receving_list, src=0)
        py_data = receving_list[0]
        input_tokens = torch.empty(*py_data["input_tokens_size"],
                                   dtype=torch.long,
                                   device="cuda")
        broadcast(input_tokens, src=0)
        input_positions = torch.empty(*py_data["input_positions_size"],
                                      dtype=torch.long,
                                      device="cuda")
        broadcast(input_positions, src=0)
        if py_data["slot_mapping_size"] is not None:
            slot_mapping = torch.empty(*py_data["slot_mapping_size"],
                                       dtype=torch.long,
                                       device="cuda")
            broadcast(slot_mapping, src=0)
        else:
            slot_mapping = None
        if py_data["context_lens_size"] is not None:
            context_lens = torch.empty(*py_data["context_lens_size"],
                                       dtype=torch.int,
                                       device="cuda")
            broadcast(context_lens, src=0)
        else:
            context_lens = None
        if py_data["block_tables_size"] is not None:
            block_tables = torch.empty(*py_data["block_tables_size"],
                                       dtype=torch.int,
                                       device="cuda")
            broadcast(block_tables, src=0)
        else:
            block_tables = None
        selected_token_indices = torch.empty(
            *py_data["selected_token_indices_size"],
            dtype=torch.long,
            device="cuda")
        broadcast(selected_token_indices, src=0)
        input_metadata = InputMetadata(
            is_prompt=py_data["is_prompt"],
            slot_mapping=slot_mapping,
            max_context_len=py_data["max_context_len"],
            context_lens=context_lens,
            block_tables=block_tables,
```

```python
            use_cuda_graph=py_data["use_cuda_graph"],
        )
        sampling_metadata = SamplingMetadata(
            seq_groups=None,
            seq_data=None,
            prompt_lens=None,
            selected_token_indices=selected_token_indices,
            categorized_sample_indices=None,
            perform_sampling=False,
        )

    return input_tokens, input_positions, input_metadata, sampling_metadata
```

- **_prepare_prompt()**

当序列组处于prompt阶段时，调用该函数获得所有序列的输入数据、位置数据、输入元数据以及prompt的长度

```python
def _prepare_prompt(
    self,
    seq_group_metadata_list: List[SequenceGroupMetadata],
) -> Tuple[torch.Tensor, torch.Tensor, InputMetadata, List[int]]:
    assert len(seq_group_metadata_list) > 0
    # 输入tokens，包含了每条序列的prompt的token id
    input_tokens: List[List[int]] = []
    # 每条序列的token的位置信息
    input_positions: List[List[int]] = []
    # token在物理块上的位置信息
    slot_mapping: List[List[int]] = []
    # 每条序列的prompt的长度
    prompt_lens: List[int] = []
    # 逐个序列组进行数据解析
    for seq_group_metadata in seq_group_metadata_list:
        assert seq_group_metadata.is_prompt
        # 获得某个序列组的所有序列的id，由于序列都处于prompt阶段，取第一条序列的id即可
        seq_ids = list(seq_group_metadata.seq_data.keys())
        assert len(seq_ids) == 1
        seq_id = seq_ids[0]
        # 获得序列组的prompt token、prompt长度以及输入位置信息
        seq_data = seq_group_metadata.seq_data[seq_id]
        prompt_tokens = seq_data.get_token_ids()
        prompt_len = len(prompt_tokens)
        prompt_lens.append(prompt_len)
        input_tokens.append(prompt_tokens)
        input_positions.append(list(range(prompt_len)))
        # 做内存profiling时，块表还没有生成，这里随机生成一个slot的映射
        if seq_group_metadata.block_tables is None:
            slot_mapping.append([_PAD_SLOT_ID] * prompt_len)
            continue
        # 构建每个token到块中token slot的映射。以某个token在8号位置，块大小为5，其所在的物理块为10，则该token所在的
token slot为10*5+8%5=53号(token生成的KV cache应当会保存在对应的token slot上)
        slot_mapping.append([])
        block_table = seq_group_metadata.block_tables[seq_id]
        # 如果采用sliding_window时，前面的token在块中的内容会被后面相应位置的token覆盖，它们对应的槽为-1(表示没有映射
到块上)
        start_idx = 0
        if self.sliding_window is not None:
            start_idx = max(0, prompt_len - self.sliding_window)
        for i in range(prompt_len):
            if i < start_idx:
                slot_mapping[-1].append(_PAD_SLOT_ID)
                continue
            block_number = block_table[i // self.block_size]
            block_offset = i % self.block_size
            slot = block_number * self.block_size + block_offset
            slot_mapping[-1].append(slot)
    # 为了保障tensor都有相同的长度，需要对小于最大长度的序列进行padding(slot_mapping用-1进行padding，说明这些token
没有被放入块中)
    max_prompt_len = max(prompt_lens)
    input_tokens = _make_tensor_with_pad(input_tokens,max_prompt_len,
                                         pad=0,dtype=torch.long)
    input_positions = _make_tensor_with_pad(input_positions,max_prompt_len,
                                            pad=0,dtype=torch.long)
    slot_mapping = _make_tensor_with_pad(slot_mapping,max_prompt_len,
```

```
                                     pad=_PAD_SLOT_ID,dtype=torch.long)
    # 构建输入的元数据，这里只需要记录token到块的映射位置即可(context实际上都在input中)
    input_metadata = InputMetadata(
        is_prompt=True,
        slot_mapping=slot_mapping,
        max_context_len=None,
        context_lens=None,
        # prompt阶段KV cache还没有被填入块表中
        block_tables=None,
        use_cuda_graph=False,
    )
    return input_tokens, input_positions, input_metadata, prompt_lens
```

- **_prepare_decode()**

当序列组处于decode阶段时，调用该函数获得所有序列的输入数据、位置数据、输入元数据

疑问：input_tokens为什么只插入最后一个token的id，推测是只有最后一个token的KV cache还没有被计算放到对应的token slot中

```
def _prepare_decode(
    self,
    seq_group_metadata_list: List[SequenceGroupMetadata],
) -> Tuple[torch.Tensor, torch.Tensor, InputMetadata]:
    assert len(seq_group_metadata_list) > 0
    # 输入的token，在decode阶段为最后一个生成的token
    input_tokens: List[List[int]] = []
    # 最后一个生成的token的位置信息
    input_positions: List[List[int]] = []
    # 最后一个生成的token在物理块中的位置
    slot_mapping: List[List[int]] = []
    # 每条序列的上下文长度
    context_lens: List[int] = []
    # 每条序列使用的物理块表(每个List包含了序列所有的物理块号)
    block_tables: List[List[int]] = []
    # 逐个序列组进行解析
    for seq_group_metadata in seq_group_metadata_list:
        assert not seq_group_metadata.is_prompt
        # 获得序列组所有的id，与prompt阶段不同，由于不同序列的数据不同，需要逐个进行解析
        seq_ids = list(seq_group_metadata.seq_data.keys())
        for seq_id in seq_ids:
            seq_data = seq_group_metadata.seq_data[seq_id]
            # 获得最后一个token以及对应的位置信息，加入到input列表中
            generation_token = seq_data.get_last_token_id()
            input_tokens.append([generation_token])
            seq_len = seq_data.get_len()
            position = seq_len - 1
            input_positions.append([position])
            # 获得序列生成采用的上下文信息的长度。没有采用滑动窗口时，采用完整的序列(包括prompt、已生成的所有token)来进行输出生成(context在后续执行时如何使用？)
            context_len = seq_len if self.sliding_window is None else min(
                seq_len, self.sliding_window)
            context_lens.append(context_len)
            # 获得最后一个token的KV cache应当放置在的slot位置，保存在slot_mapping中
            block_table = seq_group_metadata.block_tables[seq_id]
            block_number = block_table[position // self.block_size]
            block_offset = position % self.block_size
            slot = block_number * self.block_size + block_offset
            slot_mapping.append([slot])
            if self.sliding_window is not None:
                sliding_window_blocks = (self.sliding_window //
                                         self.block_size)
                block_table = block_table[-sliding_window_blocks:]
            block_tables.append(block_table)
    # 计算当前batch的输入序列数量，如果小于capture graph时采用的batch大小且context长度小于捕获时支持的最大长度，则可以利用已构建好的计算图加快推理效率
    batch_size = len(input_tokens)
    max_context_len = max(context_lens)
    use_captured_graph = (
        not self.model_config.enforce_eager
        and batch_size <= _BATCH_SIZES_TO_CAPTURE[-1]
        and max_context_len <= self.max_context_len_to_capture)
    if use_captured_graph:
```

```
        # 如果要利用已构建好的计算图进行计算，需要对序列进行padding，与已经构建好计算图的batch size相对应
        graph_batch_size = _get_graph_batch_size(batch_size)
        assert graph_batch_size >= batch_size
        for _ in range(graph_batch_size - batch_size):
            input_tokens.append([])
            input_positions.append([])
            slot_mapping.append([])
            context_lens.append(1)
            block_tables.append([])
        batch_size = graph_batch_size
    # 将输入数据等转化为相应的tensor，便于后续输入模型执行运算
    input_tokens = _make_tensor_with_pad(input_tokens,max_len=1,pad=0,
                                         dtype=torch.long,device="cuda")
    input_positions = _make_tensor_with_pad(input_positions,max_len=1,pad=0,
                                            dtype=torch.long,device="cuda")
    slot_mapping = _make_tensor_with_pad(slot_mapping,max_len=1,
                                         pad=_PAD_SLOT_ID,dtype=torch.long,
                                         device="cuda")
    context_lens = torch.tensor(context_lens,dtype=torch.int,device="cuda")

    if use_captured_graph:
        # 当利用已构建好的计算图时，由于构建计算图时已经生成了对应的块表，因此只需要将scheduler计算得到的块表的内容填入到相应的位置中即可。回顾：block_table为一个序列，每个元素代表了对应逻辑块映射到的物理块号
        input_block_tables = self.graph_block_tables[:batch_size]
        for i, block_table in enumerate(block_tables):
            if block_table:
                input_block_tables[i, :len(block_table)] = block_table
        block_tables = torch.tensor(input_block_tables, device="cuda")
    else:
        # 当不利用已构建好的计算图时，则重新生成一个tensor存放块表，为了所有输入长度的一致性，需要将context小于max_context_len的序列的块表补充0到max_context_len
        block_tables = _make_tensor_with_pad(
            block_tables,
            max_len=max_context_len,
            pad=0,
            dtype=torch.int,
            device="cuda",
        )
    # 构建输入的元数据，包括各个序列的长度、token在块上的映射位置等
    input_metadata = InputMetadata(
        is_prompt=False,
        slot_mapping=slot_mapping,
        max_context_len=max_context_len,
        context_lens=context_lens,
        block_tables=block_tables,
        use_cuda_graph=use_captured_graph,
    )
    return input_tokens, input_positions, input_metadata
```

- **_prepare_sample()**

  该函数获得采样的元数据，用于指导模型执行时的采样操作

  疑问：在prompt阶段，prompt_logprobs什么时候才会是None，这代表了什么情况，为什么此时要做相应的操作

  疑问：prepare_sample中准备的sampling元数据如何用于后续的操作，这里的selected_token_indices、categorized_sample_indices分别表示什么意思(categorized_sample_indices后续没有传给其他的ray worker，ray worker只用到了selected_token_indices作为sampling元数据)

```
def _prepare_sample(
    self,
    seq_group_metadata_list: List[SequenceGroupMetadata],
    prompt_lens: List[int],
) -> SamplingMetadata:
    # 每个序列组的序列id以及采样参数
    seq_groups: List[Tuple[List[int], SamplingParams]] = []
    #
    selected_token_indices: List[int] = []
    #
    selected_token_start_idx = 0
    #
    categorized_sample_indices = {t: [] for t in SamplingType}
    #
    categorized_sample_indices_start_idx = 0
```

```python
    # prompt阶段时，所有的prompt都被填充到了最大长度作为输入
    max_prompt_len = max(prompt_lens) if prompt_lens else 1
    for i, seq_group_metadata in enumerate(seq_group_metadata_list):
        seq_ids = list(seq_group_metadata.seq_data.keys())
        sampling_params = seq_group_metadata.sampling_params
        seq_groups.append((seq_ids, sampling_params))

        if seq_group_metadata.is_prompt:
            assert len(seq_ids) == 1
            prompt_len = prompt_lens[i]
            # prompt部分不需要进行采样，令采样开始的指针跳过这部分。
            if sampling_params.prompt_logprobs is not None:
                categorized_sample_indices_start_idx += prompt_len - 1
            # 将该序列组采样开始的idx记录到对应采样方法的数组中
            categorized_sample_indices[
                sampling_params.sampling_type].append(
                    categorized_sample_indices_start_idx)
            categorized_sample_indices_start_idx += 1
            # 当存在prompt_logprobs时，sampling将所有的prompt token都考虑在内，而不是只考虑最后一个token
            if sampling_params.prompt_logprobs is not None:
                selected_token_indices.extend(
                    range(selected_token_start_idx,
                          selected_token_start_idx + prompt_len - 1))
            selected_token_indices.append(selected_token_start_idx +
                                          prompt_len - 1)
            selected_token_start_idx += max_prompt_len
        else:
            num_seqs = len(seq_ids)
            # 每个序列都只需要考虑最后一个token，sampling按序考虑所有序列的token
            selected_token_indices.extend(
                range(selected_token_start_idx,
                      selected_token_start_idx + num_seqs))
            selected_token_start_idx += num_seqs
            categorized_sample_indices[
                sampling_params.sampling_type].extend(
                    range(categorized_sample_indices_start_idx,
                          categorized_sample_indices_start_idx + num_seqs))
            categorized_sample_indices_start_idx += num_seqs
    # 将对应的序列转换为tensor类型
    selected_token_indices = _async_h2d(selected_token_indices,
                                        dtype=torch.long,
                                        pin_memory=not self.in_wsl)
    categorized_sample_indices = {
        t: _async_h2d(seq_ids, dtype=torch.int, pin_memory=not self.in_wsl)
        for t, seq_ids in categorized_sample_indices.items()
    }
    # 所有序列组中的序列数据，包括prompt、输出的token id以及累积的logprob
    seq_data: Dict[int, SequenceData] = {}
    for seq_group_metadata in seq_group_metadata_list:
        seq_data.update(seq_group_metadata.seq_data)

    sampling_metadata = SamplingMetadata(
        seq_groups=seq_groups,
        seq_data=seq_data,
        prompt_lens=prompt_lens,
        selected_token_indices=selected_token_indices,
        categorized_sample_indices=categorized_sample_indices,
    )
    return sampling_metadata
```

2. 以llama模型为例，下面介绍模型的forward执行过程。首先回顾forward函数的各个输入信息及其shape

   input_ids：所有序列的输入token id。prompt阶段时，其shape为(seq_group_num, max_prompt_len)；decode阶段时，其shape为(seq_num, 1)

   positions：所有序列的输入token的位置，shape与input_ids相同

   kv_caches：保存在GPU中的KV cache信息，List长度为隐藏层数量，每个元素KVCache为(Tensor, Tensor)。每个Tensor保存在GPU内存中，其中key的shape为(GPU块数量**num_gpu_blocks**, 当前GPU被划分到的head数量**num_heads**(总head数//参与张量并行的GPU数), 每个head的大小**head_size//16**(16为元素大小), 块的大小**block_size**, **16**), value的shape为(GPU块数量**num_gpu_blocks**, 当前GPU被划分到的head数量**num_heads**, 每个head的大小**head_size**, 块的大小**block_size**, **16**) input_metadata：数据包括is_prompt(是否处于prompt阶段)、slot_mapping(每个token对应物理块中的位置，表示为block_number * self.block_size + block_offset，shape与input一致)、max_context_len(最大的上下文长度，在prompt阶段为None)、context_len(每个序列的上下文长度，在prompt阶段为None)、block_tables(每个序列使用的物理块表，prompt下None，非prompt下长度为序列数量)、use_cuda_graph(是否使用已构建的计算图)

疑问：对weight_loader的一个疑惑是，其输入的Parameter参数是什么，如何得到的

```
def LlamaForCausalLM.forward(
    self,
    input_ids: torch.Tensor,
    positions: torch.Tensor,
    kv_caches: List[KVCache],
    input_metadata: InputMetadata,
) -> torch.Tensor:
    # 调用Llama模型的forward函数
    hidden_states = self.model(input_ids, positions, kv_caches,input_metadata)
    return hidden_states
```

查看Llama模型的各层的初始化，其主要包括Embedding块、Decoder块以及norm块的初始化，其中Decoder块共有num_hidden_layers层。执行forward函数时，模型对输入进行Embedding，并执行num_hidden_layers层Decoder块(注意其输入)，最后执行一次均方层归一化RMSNorm(LayerNorm的改进)，得到对应的输出

```
def __init__(
    self,
    config: LlamaConfig,
    linear_method: Optional[LinearMethodBase] = None,
) -> None:
    # embedding层
    self.embed_tokens = VocabParallelEmbedding(
        config.vocab_size,
        config.hidden_size,
    )
    # decoding层
    self.layers = nn.ModuleList([
        LlamaDecoderLayer(config, linear_method)
        for _ in range(config.num_hidden_layers)
    ])
    # Norm层，rms_norm_eps是为了避免归一化时的除0错误
    self.norm = RMSNorm(config.hidden_size, eps=config.rms_norm_eps)

def forward(
    self,
    input_ids: torch.Tensor,
    positions: torch.Tensor,
    kv_caches: List[KVCache],
    input_metadata: InputMetadata,
) -> torch.Tensor:
    hidden_states = self.embed_tokens(input_ids)
    residual = None
    for i in range(len(self.layers)):
        layer = self.layers[i]
        hidden_states, residual = layer(
            positions,
            hidden_states,
            kv_caches[i],
            input_metadata,
            residual,
        )
    hidden_states, _ = self.norm(hidden_states, residual)
    return hidden_states
```

Embedding层的定义如下，由于其采用了张量并行来计算，需要注意权重矩阵的划分和计算结果的合并

```
def __init__(self,
             num_embeddings: int, # 字典大小vocabulary size
             embedding_dim: int, # 隐藏层大小hidden_size
             params_dtype: Optional[torch.dtype] = None):
    super().__init__()
    # 设置输入输出的大小以及执行张量并行的GPU数量
    self.num_embeddings = num_embeddings
    # 该函数将字典大小扩展到64的倍数
```

```
        self.num_embeddings_padded = pad_vocab_size(num_embeddings)
        self.embedding_dim = embedding_dim
        if params_dtype is None:
            params_dtype = torch.get_default_dtype()
        self.tp_size = get_tensor_model_parallel_world_size()
        # 将权重矩阵进行划分，每个GPU根据自己的rank获得Eembedding权重矩阵开始行和终止行的idx
        self.vocab_start_index, self.vocab_end_index = (
            vocab_range_from_global_vocab_size(
                self.num_embeddings_padded, get_tensor_model_parallel_rank(),
                self.tp_size))
        # 每个划分需要处理的行数
        self.num_embeddings_per_partition = (self.vocab_end_index -
                                             self.vocab_start_index)
        # 初始化权重矩阵，对于每个划分(GPU)而言，其行为划分到的行数，列为隐藏层大小
        self.weight = Parameter(
            torch.empty(self.num_embeddings_per_partition,
                        self.embedding_dim,
                        device=torch.cuda.current_device(),
                        dtype=params_dtype))
        # parallel_dim：分发张量的维度
        set_weight_attrs(self.weight, {
            "parallel_dim": 0,
            "weight_loader": self.weight_loader
        })

    def weight_loader(self, param: Parameter, loaded_weight: torch.Tensor):
        parallel_dim = param.parallel_dim
        assert loaded_weight.shape[parallel_dim] == self.num_embeddings
        loaded_weight = loaded_weight[self.vocab_start_index:self.
                                      vocab_end_index]
        param[:loaded_weight.shape[0]].data.copy_(loaded_weight)

    def forward(self, input_):
        if self.tp_size > 1:
            # 采用张量并行时，需要将没分配到当前划分的token id设置为0，这些词汇不在当前的设备执行embedding计算。且由于矩阵大小的原因，这些token的id应当减去vocab_start_index(将每个token看成只有id位置为1其他位置为0的长度为vocab_size的向量即可理解)
            input_mask = ((input_ < self.vocab_start_index) |
                          (input_ >= self.vocab_end_index))
            masked_input = input_.clone() - self.vocab_start_index
            masked_input[input_mask] = 0
        else:
            masked_input = input_
        # 执行embedding操作
        output_parallel = F.embedding(masked_input, self.weight)
        # 将没有划分到的部分的输出设置为0
        if self.tp_size > 1:
            output_parallel[input_mask, :] = 0.0
        # 将输出归约到所有执行张量并行的GPU上，执行的是求和操作(没分配到的部分输出设置为0的原因)
        output = tensor_model_parallel_all_reduce(output_parallel)
        return output
```

LlamaDecoderLayer(decoding层)的定义如下，其主要包含了四个模块：LlamaAttention、LlamaMLP以及两个RMSNorm(类似于LayerNorm)。其forward执行顺序为RMSNorm->LlamaAttention->RMSNorm->LlamaMLP(与传统的transformer不同的是这里的LayerNorm提前到了Attention和MLP之前)

```
    def __init__(
        self,
        config: LlamaConfig,
        linear_method: Optional[LinearMethodBase] = None,
    ) -> None:
        super().__init__()
        # 设置隐藏层大小，实际上就是每个token向量的长度
        self.hidden_size = config.hidden_size
        # Relative Position Encoding(ROPE)，用于初始化相对位置编码和类型的参数
        rope_theta = getattr(config, "rope_theta", 10000)
        rope_scaling = getattr(config, "rope_scaling", None)
        # 设置可支持的最大序列长度
        max_position_embeddings = getattr(config, "max_position_embeddings",8192)
        # Attention层初始化
        self.self_attn = LlamaAttention(
            hidden_size=self.hidden_size,
            num_heads=config.num_attention_heads,
            num_kv_heads=config.num_key_value_heads,
```

```
            rope_theta=rope_theta,
            rope_scaling=rope_scaling,
            max_position_embeddings=max_position_embeddings,
            linear_method=linear_method,
        )
        # MLP层初始化
        self.mlp = LlamaMLP(
            hidden_size=self.hidden_size,
            intermediate_size=config.intermediate_size,
            hidden_act=config.hidden_act,
            linear_method=linear_method,
        )
        # Norm层初始化
        self.input_layernorm = RMSNorm(config.hidden_size, eps=config.rms_norm_eps)
        self.post_attention_layernorm = RMSNorm(config.hidden_size,
                                                eps=config.rms_norm_eps)

    def forward(
        self,
        positions: torch.Tensor,
        hidden_states: torch.Tensor,
        kv_cache: KVCache,
        input_metadata: InputMetadata,
        residual: Optional[torch.Tensor],
    ) -> Tuple[torch.Tensor, torch.Tensor]:
        # Self Attention
        if residual is None:
            # 第一层Decoding层的残差为输入本身，关于残差的定义和计算可以详见RMSNorm的forward
            residual = hidden_states
            hidden_states = self.input_layernorm(hidden_states)
        else:
            hidden_states, residual = self.input_layernorm(
                hidden_states, residual)
        hidden_states = self.self_attn(
            positions=positions,
            hidden_states=hidden_states,
            kv_cache=kv_cache,
            input_metadata=input_metadata,
        )
        # Fully Connected
        hidden_states, residual = self.post_attention_layernorm(
            hidden_states, residual)
        hidden_states = self.mlp(hidden_states)
        return hidden_states, residual
```

- **LlamaAttention**

  LlamaAttention主要包含四个模块，分别为计算qkv向量QKVParallelLinear、向输入加入position embedding、执行attention操作PagedAttention以及对attention的输出进行线性映射RowParallelLinear。LlamaAttention的输入中的KV cache记录了当前隐藏层key和value向量

```
def __init__(
    self,
    hidden_size: int,
    num_heads: int,
    num_kv_heads: int, # 默认与num_heads一致，也可能为1.目前先考虑一致的情况
    rope_theta: float = 10000,
    rope_scaling: Optional[Dict[str, Any]] = None,
    max_position_embeddings: int = 8192,
    linear_method: Optional[LinearMethodBase] = None,
) -> None:
    super().__init__()
    # 初始化各个部分的维度
    self.hidden_size = hidden_size
    tp_size = get_tensor_model_parallel_world_size()
    self.total_num_heads = num_heads
    assert self.total_num_heads % tp_size == 0
    # 每个分划被分到的attention head数量
    self.num_heads = self.total_num_heads // tp_size
    self.total_num_kv_heads = num_kv_heads
    if self.total_num_kv_heads >= tp_size:
        # 如果KV head数量大于分划数量，则它应当是分划数的整数倍
        assert self.total_num_kv_heads % tp_size == 0
```

```
        else:
            # 如果KV head数量肖于分划数量，则对KV head进行复制分配给各个分划
            assert tp_size % self.total_num_kv_heads == 0
        self.num_kv_heads = max(1, self.total_num_kv_heads // tp_size)
        # 每个head的维度为总维度除以head的数量
        self.head_dim = hidden_size // self.total_num_heads
        # 每个分划的q和kv的维度
        self.q_size = self.num_heads * self.head_dim
        self.kv_size = self.num_kv_heads * self.head_dim
        # scaling常数，在执行完attention操作后除以该数
        self.scaling = self.head_dim**-0.5
        self.rope_theta = rope_theta
        self.max_position_embeddings = max_position_embeddings
        # 计算出对应的Q、K、V向量
        self.qkv_proj = QKVParallelLinear(
            hidden_size,
            self.head_dim,
            self.total_num_heads,
            self.total_num_kv_heads,
            bias=False,
            linear_method=linear_method,
        )
        # 在attention计算完成后，通过线性变换获得各个输出token的概率，需要执行reduce操作。这里input_is_parallel=True的原因是每个分划分别计算了各自的attention层，已经完成了结果的划分
        self.o_proj = RowParallelLinear(
            # input_size会在初始化时根据tp_size划分，该层实际使用的权重矩阵大小为(self.total_num_heads * self.head_dim // tp_size, hidden_size)
            self.total_num_heads * self.head_dim,
            hidden_size,
            bias=False,
            linear_method=linear_method,
        )
        # 进行position embedding操作
        self.rotary_emb = get_rope(
            self.head_dim,
            rotary_dim=self.head_dim,
            max_position=max_position_embeddings,
            base=rope_theta,
            rope_scaling=rope_scaling,
        )
        # 用PagedAttention机制计算attention
        self.attn = PagedAttention(self.num_heads,
                                   self.head_dim,
                                   self.scaling,
                                   num_kv_heads=self.num_kv_heads)

    def forward(
        self,
        positions: torch.Tensor,
        hidden_states: torch.Tensor,
        kv_cache: KVCache,
        input_metadata: InputMetadata,
    ) -> torch.Tensor:
        qkv, _ = self.qkv_proj(hidden_states)
        # 从qkv的concate结果中根据大小解析出query、key和value向量
        q, k, v = qkv.split([self.q_size, self.kv_size, self.kv_size], dim=-1)
        q, k = self.rotary_emb(positions, q, k)
        # 从KVCache对象中解析出当前层的key和value的cache
        k_cache, v_cache = kv_cache
        attn_output = self.attn(q, k, v, k_cache, v_cache, input_metadata)
        output, _ = self.o_proj(attn_output)
        return output
```

QKVParallelLinear基于ColumnParallelLinear(该层的定义在MLP部分介绍，主要区别在于weight_loader)，对Query、Key和Value向量的计算并行化，输出为Query、Key和Value向量concate之后的结果，因此权重矩阵的长度为三个向量长度的和。每个Query、Key和Value向量都包含了total_num_heads或total_num_kv_heads个向量，代表了不同head计算出的QKV结果。并行计算完成后同样不需要进行gather操作，因为每个分划只对自己划分到的head执行attention操作

```
    def __init__(
        self,
        hidden_size: int,
        head_size: int,
        total_num_heads: int,
```

```python
        total_num_kv_heads: Optional[int] = None,
        bias: bool = True,
        skip_bias_add: bool = False,
        params_dtype: Optional[torch.dtype] = None,
        linear_method: Optional[LinearMethodBase] = None,
    ):
        # 输入向量的维度，即embedding层的输出维度
        self.hidden_size = hidden_size
        # 每个head的维度
        self.head_size = head_size
        self.total_num_heads = total_num_heads
        if total_num_kv_heads is None:
            total_num_kv_heads = total_num_heads
        # 默认kv head数量与query head数量一致
        self.total_num_kv_heads = total_num_kv_heads
        # 计算出各个分划被分配到的head的数量
        tp_size = get_tensor_model_parallel_world_size()
        self.num_heads = divide(self.total_num_heads, tp_size)
        # 如果kv head为1，则需要对其进行复制
        if tp_size >= self.total_num_kv_heads:
            self.num_kv_heads = 1
            self.num_kv_head_replicas = divide(tp_size,
                                               self.total_num_kv_heads)
        else:
            self.num_kv_heads = divide(self.total_num_kv_heads, tp_size)
            self.num_kv_head_replicas = 1
        input_size = self.hidden_size
        # 该层的输出维度为Query、Key、Value向量的所有head的维度之和。注意output维度，进入到ColumnParallelLinear进行初
始化时才进行划分。
        output_size = (self.num_heads +
                       2 * self.num_kv_heads) * tp_size * self.head_size
        super().__init__(input_size, output_size, bias, False, skip_bias_add,
                         params_dtype, linear_method)
    # 获取QKV计算中采用的权重矩阵，实际上是完整的QKV层的权重矩阵，包含了所有的head(没进行划分)
    def weight_loader(self,
                      param: Parameter,
                      loaded_weight: torch.Tensor,
                      loaded_shard_id: Optional[str] = None):
        param_data = param.data
        output_dim = getattr(param, "output_dim", None)
        if loaded_shard_id is None:
            if output_dim is None:
                assert param_data.shape == loaded_weight.shape
                param_data.copy_(loaded_weight)
                return
            shard_offsets = [
                # 分别计算得到Q、K、V对应的权重矩阵的起始位置和输出维度
                ("q", 0, self.total_num_heads * self.head_size),
                ("k", self.total_num_heads * self.head_size,
                 self.total_num_kv_heads * self.head_size),
                ("v", (self.total_num_heads + self.total_num_kv_heads) *
                 self.head_size, self.total_num_kv_heads * self.head_size),
            ]
            packed_dim = getattr(param, "packed_dim", None)
            for shard_id, shard_offset, shard_size in shard_offsets:
                # 如果进行了量化，则需要调整分划的大小和偏移量
                if packed_dim == output_dim:
                    shard_size = shard_size // param.pack_factor
                    shard_offset = shard_offset // param.pack_factor
                # 在完整的权重矩阵中根据起始位置和偏移量得到Q、K、V的对应部分，分别进行划分
                loaded_weight_shard = loaded_weight.narrow(
                    output_dim, shard_offset, shard_size)
                self.weight_loader(param, loaded_weight_shard, shard_id)
            return

        tp_rank = get_tensor_model_parallel_rank()
        assert loaded_shard_id in ["q", "k", "v"]
        if output_dim is not None:
            if loaded_shard_id == "q":
                # 注意这里用的是num_heads，即total_num_heads // tp_size，是划分后的起始位置和权重矩阵长度
                shard_offset = 0
                shard_size = self.num_heads * self.head_size
            elif loaded_shard_id == "k":
                shard_offset = self.num_heads * self.head_size
```

```python
                shard_size = self.num_kv_heads * self.head_size
            elif loaded_shard_id == "v":
                shard_offset = (self.num_heads +
                                self.num_kv_heads) * self.head_size
                shard_size = self.num_kv_heads * self.head_size
            # 如果进行了量化，则需要调整分划的大小和偏移量
            packed_dim = getattr(param, "packed_dim", None)
            if packed_dim == output_dim:
                shard_size = shard_size // param.pack_factor
                shard_offset = shard_offset // param.pack_factor
            param_data = param_data.narrow(output_dim, shard_offset,
                                           shard_size)
            shard_id = tp_rank // self.num_kv_head_replicas
            start_idx = shard_id * shard_size
            loaded_weight = loaded_weight.narrow(output_dim, start_idx,
                                                 shard_size)
        else:
            ignore_warning = getattr(param, "ignore_warning", False)
            if not ignore_warning:
                logger.warning(
                    "Loading a weight without `output_dim` attribute in "
                    "QKVParallelLinear, assume the weight is the same "
                    "for all partitions.")
        assert param_data.shape == loaded_weight.shape
        param_data.copy_(loaded_weight)
```

get_rope操作根据传入了rope_scaling类型，并基于输入初始化一个RotaryEmbedding层(或者其子类，如LinearScalingRotaryEmbedding)返回给LlamaAttention，用于进行位置编码操作。RotaryEmbedding层首先初始化一个cos_sin_cache，里面缓存了各个position的position encoding结果。在forward函数中，RotaryEmbedding调用rotary_embedding_kernel将位置信息加入到Key和Value向量中(_forward函数以python的形式实现了该操作)

- **PagedAttention**

  PagedAttention层为vLLM的核心设计，采用分页的机制进行KV cache的管理，实现内存的高效利用。其初始化操作主要涉及一些赋值操作，初始化head数量、head维度、滑动窗口大小等信息。进入forward函数，其首先对Query、Key、Value向量的shape进行调整，得到

```python
'''
初始Query、Key、Value向量的shape为
query: shape = [batch_size, seq_len, num_heads * head_size]
key: shape = [batch_size, seq_len, num_kv_heads * head_size]
value: shape = [batch_size, seq_len, num_kv_heads * head_size]
'''
batch_size, seq_len, hidden_size = query.shape
query = query.view(-1, self.num_heads, self.head_size)
key = key.view(-1, self.num_kv_heads, self.head_size)
value = value.view(-1, self.num_kv_heads, self.head_size)
'''
调整后Query、Key、Value向量的shape为
query: shape = [batch_size*seq_len, num_heads, head_size]
key: shape = [batch_size*seq_len, num_kv_heads, head_size]
value: shape = [batch_size*seq_len, num_kv_heads, head_size]
'''
```

如果key和value向量已经被计算得到，则将它们保存在KV cache中，主要调用了内核函数执行操作。在这一步，prompt或者decoding阶段生成的新的token的key value向量会被存储在cache中

疑问：key cache和value cache的形状为什么不一样，为什么block要放在head之后(可能的原因是每个head的计算需要取出其对应的所有的block，这样的组织可以连续取出一个head对应的所有block)

```cpp
void reshape_and_cache(
  torch::Tensor& key, // [num_tokens, num_heads, head_size]
  torch::Tensor& value, // [num_tokens, num_heads, head_size]
  torch::Tensor& key_cache, // [num_blocks,num_heads,head_size/x,block_size,x]
  torch::Tensor& value_cache, // [num_blocks,num_heads,head_size,block_size]
  torch::Tensor& slot_mapping,  // [num_tokens]
  const std::string& kv_cache_dtype)
{
  // 初始化各项参数
  int num_tokens = key.size(0);
  int num_heads = key.size(1);
  int head_size = key.size(2);
```

```
  int block_size = key_cache.size(3);
  int x = key_cache.size(4);
  // stride的值想到与num_heads * head_size
  int key_stride = key.stride(0);
  int value_stride = value.stride(0);
  // 设置每个线程块负责一个token的内容的复制
  dim3 grid(num_tokens);
  // 每个线程块的线程数量，每个线程负责一个值的内容的复制
  dim3 block(std::min(num_heads * head_size, 512));
  // 设置当前的上下文cuda设备
  const at::cuda::OptionalCUDAGuard device_guard(device_of(key));
  const cudaStream_t stream = at::cuda::getCurrentCUDAStream();
  if (kv_cache_dtype == "auto") {
    if (key.dtype() == at::ScalarType::Float) {
      // CALL_RESHAPE_AND_CACHE函数作用在于对key和value cache的数据类型进行转换，并利用完成类型转换的数据调用
reshape_and_cache_kernel核函数，进行数据存储
      CALL_RESHAPE_AND_CACHE(float, float, false);
    } else if (key.dtype() == at::ScalarType::Half) {
      CALL_RESHAPE_AND_CACHE(uint16_t, uint16_t, false);
    } else if (key.dtype() == at::ScalarType::BFloat16) {
      CALL_RESHAPE_AND_CACHE(__nv_bfloat16, __nv_bfloat16, false);
    }
  } else if (kv_cache_dtype == "fp8_e5m2") {
    if (key.dtype() == at::ScalarType::Float) {
      CALL_RESHAPE_AND_CACHE(float, uint8_t, true);
    } else if (key.dtype() == at::ScalarType::Half) {
      CALL_RESHAPE_AND_CACHE(uint16_t, uint8_t, true);
    } else if (key.dtype() == at::ScalarType::BFloat16) {
      CALL_RESHAPE_AND_CACHE(__nv_bfloat16, uint8_t, true);
    }
  } else {
    TORCH_CHECK(false, "Unsupported data type of kv cache: ", kv_cache_dtype);
  }
}

template<typename scalar_t, typename cache_t, bool is_fp8_e5m2_kv_cache>
__global__ void reshape_and_cache_kernel(
  const scalar_t* __restrict__ key, // [num_tokens, num_heads, head_size]
  const scalar_t* __restrict__ value, // [num_tokens, num_heads, head_size]
  cache_t* __restrict__ key_cache, // [num_blocks, num_heads, head_size/x, block_size, x]
  cache_t* __restrict__ value_cache, // [num_blocks, num_heads, head_size, block_size]
  const int64_t* __restrict__ slot_mapping, // [num_tokens]
  const int key_stride, // num_heads*head_size，表示一个token的大小
  const int value_stride,
  const int num_heads,
  const int head_size,
  const int block_size,
  const int x) {

  const int64_t token_idx = blockIdx.x;
  const int64_t slot_idx = slot_mapping[token_idx];
  if (slot_idx < 0) {
    // 如果当前的token没有被分配块slot(值为-1)，忽略
    return;
  }

  // 计算出当前token所在的块以及应当放置在块的哪个slot(每个块有block_size个slot)
  const int64_t block_idx = slot_idx / block_size;
  const int64_t block_offset = slot_idx % block_size;

  // 一个token的key或value向量的元素数量，for循环事实上只执行一次或者零次
  const int n = num_heads * head_size;
  for (int i = threadIdx.x; i < n; i += blockDim.x) {
    // 拷贝的数据在key和value向量中所在的位置
    const int64_t src_key_idx = token_idx * key_stride + i;
    const int64_t src_value_idx = token_idx * value_stride + i;
    // 当前idx位于的head号
    const int head_idx = i / head_size;
    const int head_offset = i % head_size;
    // 留空，x这里代表的是什么，有什么意义
    const int x_idx = head_offset / x;
    const int x_offset = head_offset % x;
    // 计算出需要进行拷贝的值在key cache中的位置
    const int64_t tgt_key_idx
```

```
    = block_idx * num_heads * (head_size / x) * block_size * x
    + head_idx * (head_size / x) * block_size * x
    + x_idx * block_size * x
    + block_offset * x
    + x_offset;
    // 计算处需要拷贝的值在value cache中的位置
    const int64_t tgt_value_idx
    = block_idx * num_heads * head_size * block_size
    + head_idx * head_size * block_size
    + head_offset * block_size
    + block_offset;
    // 获得需要拷贝的值，并在后续拷贝到KV cache对应的位置上
    scalar_t tgt_key = key[src_key_idx];
    scalar_t tgt_value = value[src_value_idx];
    if constexpr (is_fp8_e5m2_kv_cache) {
#ifdef ENABLE_FP8_E5M2
      key_cache[tgt_key_idx] = fp8_e5m2_unscaled::vec_conversion<uint8_t, scalar_t>(tgt_key);
      value_cache[tgt_value_idx] = fp8_e5m2_unscaled::vec_conversion<uint8_t, scalar_t>(tgt_value);
#else
      assert(false);
#endif
    } else {
      key_cache[tgt_key_idx] = tgt_key;
      value_cache[tgt_value_idx] = tgt_value;
    }
  }
}
```

如果当前的序列组位于prompt阶段，由于该阶段不需要用到KV cache和PagedAttention机制(prompt的key和value刚刚计算出来，可以直接使用)，因此vLLM直接用xformer调用相关的库memory_efficient_attention_forward计算attention的结果

vLLM在prompt阶段引入注意力偏置，在注意力机制中引入偏置或约束，改变注意力权重的计算方式，以使模型在关注输入数据时更倾向于特定的方面或特征，从而实现以下功能

- 指导模型关注特定的信息或特征，以提高任务性能，适应特定的数据分布或任务要求
- 约束注意力范围，控制模型关注的上下文窗口大小，避免过度关注长距离的依赖关系或无关信息
- 帮助语言模型处理超过训练样本的context长度

```
if self.num_kv_heads != self.num_heads:
    # 如果num_kv_heads=1，则需要将key和value复制为num_queries_per_kv个副本，表示每个key和value处理
num_queries_per_kv个head
    query = query.view(query.shape[0], self.num_kv_heads,
                       self.num_queries_per_kv, query.shape[-1])
    key = key[:, :,None, :].expand(key.shape[0], self.num_kv_heads,
                              self.num_queries_per_kv,
                              key.shape[-1])
    value = value[:, :, None, :].expand(value.shape[0],
                                   self.num_kv_heads,
                                   self.num_queries_per_kv,
                                   value.shape[-1])
# 引入注意力偏置
if input_metadata.attn_bias is None:
    if self.alibi_slopes is None:
        attn_bias = BlockDiagonalCausalMask.from_seqlens(
            [seq_len] * batch_size)
        if self.sliding_window is not None:
            # 只考虑滑动窗口的部分
            attn_bias = attn_bias.make_local_attention(self.sliding_window)
        input_metadata.attn_bias = attn_bias
    else:
        input_metadata.attn_bias = _make_alibi_bias(
            self.alibi_slopes, self.num_kv_heads, batch_size,
            seq_len, query.dtype)

if self.alibi_slopes is None:
    # 形状变为[1, batch_size*seq_len, num_heads, head_size]
    query = query.unsqueeze(0)
    key = key.unsqueeze(0)
    value = value.unsqueeze(0)
else:
    # 将Q、K、V向量的形状变化为[batch_size, seq_len, num_heads, head_size]
    query = query.unflatten(0, (batch_size, seq_len))
    key = key.unflatten(0, (batch_size, seq_len))
```

```
    value = value.unflatten(0, (batch_size, seq_len))

out = xops.memory_efficient_attention_forward(
    query,
    key,
    value,
    attn_bias=input_metadata.attn_bias,
    p=0.0,
    scale=self.scale,
    op=xops.fmha.MemoryEfficientAttentionFlashAttentionOp[0] if
    (is_hip()) else None,
)
output = out.view_as(query)
```

如果当前的序列组位于decode阶段，vLLM则采用设计的PagedAttention机制，利用存储在GPU块中的KV cache和新的token的Q、K、V向量，计算attention结果。

疑问：这里调用PagedAttention V1 或者 V2的差异在哪里，为什么采用这样的判定条件

```
def _paged_attention(
    query: torch.Tensor,
    key_cache: torch.Tensor,
    value_cache: torch.Tensor,
    input_metadata: InputMetadata,
    num_kv_heads: int,
    scale: float,
    alibi_slopes: Optional[torch.Tensor],
) -> torch.Tensor:
    # output、query、key、value的维度一致，都为(batch_size, num_heads, head_size)
    # 剩余token的key和value都已经保存在了KV cache中
    output = torch.empty_like(query)
    block_size = value_cache.shape[3]
    num_seqs, num_heads, head_size = query.shape
    max_num_partitions = (
        (input_metadata.max_context_len + _PARTITION_SIZE - 1) //
        _PARTITION_SIZE)
    #
    use_v1 = input_metadata.max_context_len <= 8192 and (
        max_num_partitions == 1 or num_seqs * num_heads > 512)
    if use_v1:
        # Run PagedAttention V1.
        ops.paged_attention_v1(
            output,
            query,
            key_cache,
            value_cache,
            num_kv_heads,
            scale,
            input_metadata.block_tables,
            input_metadata.context_lens,
            block_size,
            input_metadata.max_context_len,
            alibi_slopes,
        )
    else:
        #
        assert _PARTITION_SIZE % block_size == 0
        tmp_output = torch.empty(
            size=(num_seqs, num_heads, max_num_partitions, head_size),
            dtype=output.dtype,
            device=output.device,
        )
        exp_sums = torch.empty(
            size=(num_seqs, num_heads, max_num_partitions),
            dtype=torch.float32,
            device=output.device,
        )
        max_logits = torch.empty_like(exp_sums)
        ops.paged_attention_v2(
            output,
            exp_sums,
            max_logits,
            tmp_output,
```

```
                query,
                key_cache,
                value_cache,
                num_kv_heads,
                scale,
                input_metadata.block_tables,
                input_metadata.context_lens,
                block_size,
                input_metadata.max_context_len,
                alibi_slopes,
            )
        return output
```

PagedAttentionV1的主要定义如下

疑问：share_mem_size是如何进行设置的；线程块大小等设置的原因是什么

```
template<
  typename T,
  int BLOCK_SIZE,
  int NUM_THREADS = 128>
void paged_attention_v1_launcher(
  torch::Tensor& out,
  torch::Tensor& query,
  torch::Tensor& key_cache,
  torch::Tensor& value_cache,
  int num_kv_heads,
  float scale,
  torch::Tensor& block_tables,
  torch::Tensor& context_lens,
  int max_context_len,
  const c10::optional<torch::Tensor>& alibi_slopes) {
  // query、key、value的shape为[num_blocks, num_heads, head_size]
  int num_seqs = query.size(0);
  int num_heads = query.size(1);
  int head_size = query.size(2);
  // 每个序列的块表都被扩展到了最大块数
  int max_num_blocks_per_seq = block_tables.size(1);
  // query等向量中token的步幅，此时第i个token的offset为i * q_stride
  int q_stride = query.stride(0);
  // 块的步幅，留意cache的shape[num_blocks,num_heads,head_size/x,block_size,x]
  int kv_block_stride = key_cache.stride(0);
  // head的步幅
  int kv_head_stride = key_cache.stride(1);
  // 通常一个warp计算一个块的结果。当block_size较小时，划分线程组使得warp能处理多个块
  int thread_group_size = MAX(WARP_SIZE / BLOCK_SIZE, 1);
  assert(head_size % thread_group_size == 0);
  // 将数据转换对应类型的数据指针
  const float* alibi_slopes_ptr = alibi_slopes ?
    reinterpret_cast<const float*>(alibi_slopes.value().data_ptr()):nullptr;
  T* out_ptr = reinterpret_cast<T*>(out.data_ptr());
  T* query_ptr = reinterpret_cast<T*>(query.data_ptr());
  T* key_cache_ptr = reinterpret_cast<T*>(key_cache.data_ptr());
  T* value_cache_ptr = reinterpret_cast<T*>(value_cache.data_ptr());
  int* block_tables_ptr = block_tables.data_ptr<int>();
  int* context_lens_ptr = context_lens.data_ptr<int>();
  // NUM_THREADS为一个线程块中线程的数量，他们划分为了NUM_WARPS个warp运行，默认为128
  constexpr int NUM_WARPS = NUM_THREADS / WARP_SIZE;
  // 填满所有的块所需要的token数量，这里为什么要这样设置线程块需要的shared_mem_size
  int padded_max_context_len = DIVIDE_ROUND_UP(max_context_len, BLOCK_SIZE) * BLOCK_SIZE;
  int logits_size = padded_max_context_len * sizeof(float);
  int outputs_size = (NUM_WARPS / 2) * head_size * sizeof(float);
  int shared_mem_size = std::max(logits_size, outputs_size);
  // 每个线程块处理某条序列的某个head的计算
  dim3 grid(num_heads, num_seqs, 1);
  dim3 block(NUM_THREADS);
  const at::cuda::OptionalCUDAGuard device_guard(device_of(query));
  const cudaStream_t stream = at::cuda::getCurrentCUDAStream();
  switch (head_size) {
    ... // 这里省略，实际上就是调用paged_attention_kernel进行计算操作
  }
}
```

PagedAttentionV2的主要定义如下

```
template<
  typename T,
  int BLOCK_SIZE,
  int NUM_THREADS = 128,
  int PARTITION_SIZE = 512>
void paged_attention_v2_launcher(
  torch::Tensor& out,
  torch::Tensor& exp_sums,
  torch::Tensor& max_logits,
  torch::Tensor& tmp_out,
  torch::Tensor& query,
  torch::Tensor& key_cache,
  torch::Tensor& value_cache,
  int num_kv_heads,
  float scale,
  torch::Tensor& block_tables,
  torch::Tensor& context_lens,
  int max_context_len,
  const c10::optional<torch::Tensor>& alibi_slopes) {
  int num_seqs = query.size(0);
  int num_heads = query.size(1);
  int head_size = query.size(2);
  int max_num_blocks_per_seq = block_tables.size(1);
  int q_stride = query.stride(0);
  int kv_block_stride = key_cache.stride(0);
  int kv_head_stride = key_cache.stride(1);

  int thread_group_size = MAX(WARP_SIZE / BLOCK_SIZE, 1);
  assert(head_size % thread_group_size == 0);

  // NOTE: alibi_slopes is optional.
  const float* alibi_slopes_ptr = alibi_slopes ?
    reinterpret_cast<const float*>(alibi_slopes.value().data_ptr())
    : nullptr;

  T* out_ptr = reinterpret_cast<T*>(out.data_ptr());
  float* exp_sums_ptr = reinterpret_cast<float*>(exp_sums.data_ptr());
  float* max_logits_ptr = reinterpret_cast<float*>(max_logits.data_ptr());
  T* tmp_out_ptr = reinterpret_cast<T*>(tmp_out.data_ptr());
  T* query_ptr = reinterpret_cast<T*>(query.data_ptr());
  T* key_cache_ptr = reinterpret_cast<T*>(key_cache.data_ptr());
  T* value_cache_ptr = reinterpret_cast<T*>(value_cache.data_ptr());
  int* block_tables_ptr = block_tables.data_ptr<int>();
  int* context_lens_ptr = context_lens.data_ptr<int>();

  constexpr int NUM_WARPS = NUM_THREADS / WARP_SIZE;
  int max_num_partitions = DIVIDE_ROUND_UP(max_context_len, PARTITION_SIZE);
  int logits_size = PARTITION_SIZE * sizeof(float);
  int outputs_size = (NUM_WARPS / 2) * head_size * sizeof(float);

  // For paged attention v2 kernel.
  dim3 grid(num_heads, num_seqs, max_num_partitions);
  int shared_mem_size = std::max(logits_size, outputs_size);
  // For paged attention v2 reduce kernel.
  dim3 reduce_grid(num_heads, num_seqs);
  int reduce_shared_mem_size = 2 * max_num_partitions * sizeof(float);

  dim3 block(NUM_THREADS);
  const at::cuda::OptionalCUDAGuard device_guard(device_of(query));
  const cudaStream_t stream = at::cuda::getCurrentCUDAStream();
  switch (head_size) {
    // NOTE(woosuk): To reduce the compilation time, we only compile for the
    // head sizes that we use in the model. However, we can easily extend this
    // to support any head size which is a multiple of 16.
    case 64:
      LAUNCH_PAGED_ATTENTION_V2(64);
      break;
    case 80:
      LAUNCH_PAGED_ATTENTION_V2(80);
      break;
    case 96:
      LAUNCH_PAGED_ATTENTION_V2(96);
```

```
        break;
      case 112:
        LAUNCH_PAGED_ATTENTION_V2(112);
        break;
      case 128:
        LAUNCH_PAGED_ATTENTION_V2(128);
        break;
      case 256:
        LAUNCH_PAGED_ATTENTION_V2(256);
        break;
      default:
        TORCH_CHECK(false, "Unsupported head size: ", head_size);
        break;
    }
  }
```

- **LlamaMLP**

以下为LlamaMLP的构造以及其forward函数，其包含了两个线性层运算和一个激活函数计算。根据Megatron-LM论文中的介绍，对于第一层线性层$Y = XA$的计算，对权重矩阵的列进行划分得到$[A_1, A_2]$，执行并行计算$[XA_1, XA_2]$，得到结果$[Y_1, Y_2]$。将$[Y_1, Y_2]$通过激活函数后，再对第二层线性层的权重矩阵B的行进行划分得到，并行计算得到结果$[Y_1B_1, Y_2B_2]$。最后将结果进行合并，即可得到最终的并行结果

```
def __init__(
    self,
    hidden_size: int,
    intermediate_size: int,
    hidden_act: str, # 采用的非线性激活函数
    linear_method: Optional[LinearMethodBase] = None,
) -> None:
    super().__init__()
    self.gate_up_proj = MergedColumnParallelLinear(
        hidden_size, [intermediate_size] * 2,
        bias=False,
        linear_method=linear_method)
    self.down_proj = RowParallelLinear(intermediate_size,
                                       hidden_size,
                                       bias=False,
                                       linear_method=linear_method)
    if hidden_act != "silu":
        raise ValueError(f"Unsupported activation: {hidden_act}. "
                         "Only silu is supported for now.")
    # 激活函数设置为SilluAndMul()
    self.act_fn = SiluAndMul()

def forward(self, x):
    # 输入大小为hidden_size，输出大小为2*intermediate_size
    gate_up, _ = self.gate_up_proj(x)
    # 输入大小为2*intermediate_size，输出大小为intermediate_size。详见Silu的定义
    x = self.act_fn(gate_up)
    # 输入大小为intermediate_size，输出大小为hidden_size
    x, _ = self.down_proj(x)
    return x
```

ColumnParallelLinear层

将权重矩阵A进行划分得到$A = [A_1, \ldots, A_p]$，不需要对输入矩阵X进行划分，得到结果$[Y_1, \ldots, Y_p] = [XA_1, \ldots, XA_p]$

```
def __init__(
    self,
    input_size: int, # hidden_size，输入向量的长度
    output_size: int, # 2*intermediate_size，输出向量的长度
    bias: bool = True, # False，是否采用偏移调整模型预测能力
    gather_output: bool = False, # 计算完成后是否执行all-gather操作
    skip_bias_add: bool = False, # 是否跳过结果与bias相加的步骤
    params_dtype: Optional[torch.dtype] = None, # 模型采用的数据
    linear_method: Optional[LinearMethodBase] = None, # 采用的线性量化方案
):
    super().__init__()
    # 保存各项参数
```

```
    self.input_size = input_size
    self.output_size = output_size
    self.gather_output = gather_output
    self.skip_bias_add = skip_bias_add
    if params_dtype is None:
        params_dtype = torch.get_default_dtype()
    self.params_dtype = params_dtype
    if linear_method is None:
        linear_method = UnquantizedLinearMethod()
    self.linear_method = linear_method
    # 根据参与张量并行的GPU数量，将权重矩阵和输出划分，得到每个划分计算的列数
    tp_size = get_tensor_model_parallel_world_size()
    self.output_size_per_partition = divide(output_size, tp_size)
    # 生成一个行为output_size_per_partition，列为input_size的权重矩阵。其输入维度input_dim为1(input_size)，输出维度output_dim为0(output_size_per_partition)
    self.linear_weights = self.linear_method.create_weights(
        self.input_size, self.output_size_per_partition, self.input_size,
        self.output_size, self.params_dtype)
    # 设置向模型注册可学习的权重参数weight，并设置weight权重的加载器为weight_loader
    for name, weight in self.linear_weights.items():
        if isinstance(weight, torch.Tensor):
            self.register_parameter(name, weight)
            set_weight_attrs(weight, {"weight_loader": self.weight_loader})
    # Llama模型中没有用到bias，暂时不考虑bias的影响。bias主要调整模型的灵活性和预测能力，相当于将线性方程修改为Y = AX + bias
    if bias:
        self.bias = Parameter(
            torch.empty(self.output_size_per_partition,
                        device=torch.cuda.current_device(),
                        dtype=params_dtype))
        set_weight_attrs(self.bias, {
            "output_dim": 0,
            "weight_loader": self.weight_loader,
        })
    else:
        # 设置bias为None。register_parameter作用为向模型注册一个名为bias的参数
        self.register_parameter("bias", None)
# 根据当前进程的rank获得对应的权重矩阵，并拷贝到linear_weights
def weight_loader(self, param: Parameter, loaded_weight: torch.Tensor):
    tp_rank = get_tensor_model_parallel_rank()
    output_dim = getattr(param, "output_dim", None)
    param_data = param.data
    if output_dim is not None:
        shard_size = param_data.shape[output_dim]
        start_idx = tp_rank * shard_size
        # narrow对完整的权重矩阵在output_dim进行切片操作
        loaded_weight = loaded_weight.narrow(output_dim, start_idx, shard_size)
    assert param_data.shape == loaded_weight.shape
    param_data.copy_(loaded_weight)

def forward(self, input_):
    bias = self.bias if not self.skip_bias_add else None
    # 执行矩阵乘法，令输入与当前分划的权重矩阵相乘，得到当前分划的输出结果。注意调用F.linear时，权重张量的维度应该是(output_features, input_features)
    output_parallel = self.linear_method.apply_weights(
        self.linear_weights, input_, bias)
    if self.gather_output:
        # 需要时聚合所有分划的输出结果
        output = tensor_model_parallel_all_gather(output_parallel)
    else:
        output = output_parallel
    output_bias = self.bias if self.skip_bias_add else None
    return output, output_bias
```

MergedColumnParallelLinear层(该层的分划操作比较绕)

该层为Llama实际使用的层，与ColumnParallelLinear基本一致，其主要区别在于其权重矩阵和加载有不同。输入的output_sizes为一个序列，在载入权重矩阵时将多个需要的权重子矩阵连接到了一起。在为每个分划分配权重矩阵时，需要逐个子矩阵进行遍历并将矩阵相应的部分添加到分划的权重矩阵的对应位置中(在Llama中对于每个分划，其output_sizes为2*intermediate_size // tp_size，即获取了一个2*intermediate_size大小的权重矩阵并逐个矩阵地将计算使用的权重分配给各个分划)

```
def __init__(
    self,
```

```python
        input_size: int,
        output_sizes: List[int],
        bias: bool = True,
        gather_output: bool = False,
        skip_bias_add: bool = False,
        params_dtype: Optional[torch.dtype] = None,
        linear_method: Optional[LinearMethodBase] = None,
    ):
        self.output_sizes = output_sizes
        tp_size = get_tensor_model_parallel_world_size()
        assert all(output_size % tp_size == 0 for output_size in output_sizes)
        super().__init__(input_size, sum(output_sizes), bias, gather_output,
                         skip_bias_add, params_dtype, linear_method)

    def weight_loader(self,
                      param: Parameter,
                      loaded_weight: torch.Tensor,
                      loaded_shard_id: Optional[int] = None):
        param_data = param.data
        output_dim = getattr(param, "output_dim", None)
        # 如果没有id，则需要将连接后的权重矩阵进行划分，得到多个子权重矩阵，它们的输出大小等于output_size
        if loaded_shard_id is None:
            if output_dim is None:
                assert param_data.shape == loaded_weight.shape
                param_data.copy_(loaded_weight)
                return
            current_shard_offset = 0
            shard_offsets = []
            # 根据output_sizes将连接后的权重矩阵逐个划分成多个子矩阵
            for i, output_size in enumerate(self.output_sizes):
                shard_offsets.append((i, current_shard_offset, output_size))
                current_shard_offset += output_size
            packed_dim = getattr(param, "packed_dim", None)
            for shard_id, shard_offset, shard_size in shard_offsets:
                # 如果对权重矩阵进行了量化处理，需要调整分划的大小和偏移量
                if packed_dim == output_dim:
                    shard_size = shard_size // param.pack_factor
                    shard_offset = shard_offset // param.pack_factor
                loaded_weight_shard = loaded_weight.narrow(
                    output_dim, shard_offset, shard_size)
                self.weight_loader(param, loaded_weight_shard, shard_id)
            return

        assert loaded_shard_id < len(self.output_sizes)
        tp_rank = get_tensor_model_parallel_rank()
        tp_size = get_tensor_model_parallel_world_size()
        if output_dim is not None:
            # 分划的权重矩阵中，应当赋值当前子矩阵的权重的起始位置(每个子矩阵给某个分划分配的权重数量为
    output_sizes[loaded_shard_id] // tp_size)
            shard_offset = sum(self.output_sizes[:loaded_shard_id]) // tp_size
            # 当前子矩阵应当分配给当前分划的列数
            shard_size = self.output_sizes[loaded_shard_id] // tp_size
            # 如果对权重矩阵进行了量化处理，需要调整分划的大小和偏移量
            packed_dim = getattr(param, "packed_dim", None)
            if packed_dim == output_dim:
                shard_size = shard_size // param.pack_factor
                shard_offset = shard_offset // param.pack_factor
            # 分划自己的权重矩阵也需要取出相应的部分进行赋值操作
            param_data = param_data.narrow(output_dim, shard_offset,
                                           shard_size)
            # 子权重矩阵中，分配给当前分划的起始位置
            start_idx = tp_rank * shard_size
            # 子权重矩阵中获取分划对应的位置进行后续赋值
            loaded_weight = loaded_weight.narrow(output_dim, start_idx,
                                                 shard_size)
        else:
            ignore_warning = getattr(param, "ignore_warning", False)
            if not ignore_warning:
                logger.warning(
                    "Loading a weight without `output_dim` attribute in "
                    "MergedColumnParallelLinear, assume the weight is "
                    "the same for all partitions.")
        assert param_data.shape == loaded_weight.shape
        param_data.copy_(loaded_weight)
```

RowParallelLinear层

对权重矩阵的行进行划分，并将其与经过列划分后的输入相乘，得到结果。最终的运行结果通过一次all-reduce操作将各个分划的计算结果进行相加合并

```
"""
The linear layer is defined as Y = XA + b. A is parallelized along
its first dimension and X along its second dimension as:
          | A_1 |
          | .   |
      A = | .   |        X = [X_1, ..., X_p]
          | .   |
          | A_p |
"""
def __init__(
    self,
    input_size: int, # intermediate_size，权重矩阵的行数
    output_size: int, # hidden_size，权重矩阵的列数
    bias: bool = True, # 线性运算是否采用偏移量，Llama中为False
    input_is_parallel: bool = True, # 判断输入是否已经在不同GPU间完成了划分
    skip_bias_add: bool = False, # 是否跳过加上偏移量的计算
    params_dtype: Optional[torch.dtype] = None,
    reduce_results: bool = True, # 是否聚合运算结果
    linear_method: Optional[LinearMethodBase] = None,
):
    super().__init__()
    # 保存初始化的参数
    self.input_size = input_size
    self.output_size = output_size
    self.input_is_parallel = input_is_parallel
    self.reduce_results = reduce_results
    if params_dtype is None:
        params_dtype = torch.get_default_dtype()
    self.params_dtype = params_dtype
    self.skip_bias_add = skip_bias_add
    self.linear_method = linear_method
    # 对行进行划分，得到分配给各个分划的行数
    self.tp_size = get_tensor_model_parallel_world_size()
    self.input_size_per_partition = divide(input_size, self.tp_size)
    if linear_method is None:
        linear_method = UnquantizedLinearMethod()
    # 生成一个行为output_size，列为input_size_per_partition的权重矩阵
    self.linear_weights = self.linear_method.create_weights(
        self.input_size_per_partition, self.output_size, self.input_size,
        self.output_size, self.params_dtype)
    for name, weight in self.linear_weights.items():
        if isinstance(weight, torch.Tensor):
            self.register_parameter(name, weight)
            set_weight_attrs(weight, {"weight_loader": self.weight_loader})
    # 如果不聚合结果时，直接向矩阵乘法结果增加偏移量会出现问题(行划分后运行结果需要执行加法操作，直接与bias相加会导致每个分划的计算结果都与bias相加，导致出错)
    if not reduce_results and (bias and not skip_bias_add):
        raise ValueError("When not reduce the results, adding bias to the "
                         "results can lead to incorrect results")

    if bias:
        self.bias = Parameter(
            torch.empty(self.output_size,
                        device=torch.cuda.current_device(),
                        dtype=params_dtype))
        set_weight_attrs(self.bias, {
            "output_dim": 0,
            "weight_loader": self.weight_loader,
        })
    else:
        self.register_parameter("bias", None)

def weight_loader(self, param: Parameter, loaded_weight: torch.Tensor):
    # 根据当前分划的rank获取权重矩阵相应部分，并拷贝到param.data中
    tp_rank = get_tensor_model_parallel_rank()
    input_dim = getattr(param, "input_dim", None)
    param_data = param.data
    if input_dim is not None:
        shard_size = param_data.shape[input_dim]
```

```python
            start_idx = tp_rank * shard_size
            loaded_weight = loaded_weight.narrow(input_dim, start_idx,
                                                 shard_size)
        assert param_data.shape == loaded_weight.shape
        param_data.copy_(loaded_weight)

    def forward(self, input_):
        # 如果输入还没有被划分到不同GPU上，则需要根据当前分划的rank获得相应位置的输入
        if self.input_is_parallel:
            input_parallel = input_
        else:
            tp_rank = get_tensor_model_parallel_rank()
            splitted_input = split_tensor_along_last_dim(
                input_, num_partitions=self.tp_size)
            input_parallel = splitted_input[tp_rank].contiguous()
        # 执行矩阵乘法操作获得当前分划的输出
        output_parallel = self.linear_method.apply_weights(
            self.linear_weights, input_parallel)
        # 需要集合计算结果时，调用all_reduce操作，将所有分划的计算结果相加
        if self.reduce_results and self.tp_size > 1:
            output_ = tensor_model_parallel_all_reduce(output_parallel)
        else:
            output_ = output_parallel
        if not self.skip_bias_add:
            output = output_ + self.bias if self.bias is not None else output_
            output_bias = None
        else:
            output = output_
            output_bias = self.bias
        return output, output_bias
```