Worker初始化

- **初始化worker对象：LLMEngine.init_workers()**

  在单机情况下，vLLM初始化本地的worker作为drive worker，不初始化其他的ray worker(其他非本机的worker保存在self.workers)。此时，初始化操作主要分为三个阶段

  Step 1：初始化driver_worker为Worker类的对象，并设置is_driver_worker=True对其进行标记

  Step 2：令driver worker初始化模型运行环境

  Step 3：令driver worker载入模型

```python
def _init_workers(self):
    from vllm.worker.worker import Worker
    assert self.parallel_config.world_size == 1, (
        "Ray is required if parallel_config.world_size > 1.")
    # Step 1
    self.workers: List[Worker] = []
    distributed_init_method = f"tcp://{get_ip()}:{get_open_port()}"
    self.driver_worker = Worker(
        self.model_config,
        self.parallel_config,
        self.scheduler_config,
        local_rank=0,
        rank=0,
        distributed_init_method=distributed_init_method,
        is_driver_worker=True,
    )
    # Step 2
    self._run_workers("init_model")
    # Step 3
    self._run_workers("load_model")
```

  1. **Worker.init()**

     设置当前worker的参数，包括模型参数(model_config)、并行参数(parallel_config)、调度器参数(scheduler_config)、**层级(rank和local_rank，local_rank为当前worker对应的本地GPU索引，rank为当前worker在分布式环境中的编号)**、初始化方法(distributed_init_method)、是否为driver(is_driver_worker)。此外，其初始化了一个ModelRunner对象，里面包含了模型执行的相关信息以及模型执行的方法(部分初始化操作需要在profiling后才设置)

  2. **Worker.init_model()**

     初始化模型运行环境，包括设置环境变量、设置运行的设备、检查模型支持的数据类型、初始化分布式环境和种子。初始化分布式环境时，如果当前进程已经完成了环境的初始化，则验证初始化的分布式环境进程数量与parallel_config设置的进程数量相等；否则调用torch.distributed.init_process_group函数，定义分布式环境中进程间的通信方式、进程通过哪个IP和端口建立通信(通常为driver的IP和某个端口)。

```python
def init_model(self) -> None:
    # NCCL为了保障数据安全，在执行all_reduce操作时会在RECORD_STREAMS保存input tensor直到同      步完成，导致内存使用量增加。该环境变量的设置避免了这一点
    os.environ["TORCH_NCCL_AVOID_RECORD_STREAMS"] = "1"
    # 该环境变量会导致图构建时出现问题
    os.environ.pop("NCCL_ASYNC_ERROR_HANDLING", None)
    # 设置当前线程使用的设备为local_rank号GPU
    self.device = torch.device(f"cuda:{self.local_rank}")
    torch.cuda.set_device(self.device)
    # 检查GPU是否支持给定的模型数据类型
    _check_if_gpu_supports_dtype(self.model_config.dtype)
    # 初始化分布式环境
    _init_distributed_environment(self.parallel_config, self.rank,
                                  self.distributed_init_method)
    # 初始化模型的种子，保证模型执行的一致性
    set_random_seed(self.model_config.seed)

def _init_distributed_environment(
    parallel_config: ParallelConfig,
    rank: int,
    distributed_init_method: Optional[str] = None,
) -> None:
```

```
    if torch.distributed.is_initialized():
        torch_world_size = torch.distributed.get_world_size()
        if torch_world_size != parallel_config.world_size:
            raise RuntimeError(...)
    elif not distributed_init_method:
        raise ValueError(...)
    else:
    # distributed_init_method的例子为tcp://{driver_ip}:{get_open_port()}，表示采用tcp        利用driver的某个开放端口建立通信
        torch.distributed.init_process_group(
            backend="nccl",
            world_size=parallel_config.world_size,
            rank=rank,
            init_method=distributed_init_method,
        )
    torch.distributed.all_reduce(torch.zeros(1).cuda())
    initialize_model_parallel(parallel_config.tensor_parallel_size,
                              parallel_config.pipeline_parallel_size)
```

完成分布式环境初始化后，将调用initialize_model_parallel函数，根据给定的tensor模型并行采用的gpu数量和pipeline模型并行采用的模型数量，初始化模型分布式运行时采用的运行组，并用全局变量记录组的情况以及进程自身所在的pipeline组

疑问：不同的并行组之间运行的内容不同，还是同一组之间运行的内容不同。具体运行时是如何对模型在进行划分分配到组实现张量和流水线并行的

```
def initialize_model_parallel(
    tensor_model_parallel_size: int = 1,
    pipeline_model_parallel_size: int = 1,
) -> None:
    assert torch.distributed.is_initialized()
    world_size: int = torch.distributed.get_world_size()
    if (world_size !=
            tensor_model_parallel_size * pipeline_model_parallel_size):
        raise RuntimeError(...)
    # 根据进程数量(GPU数量)以及组的大小计算出两个组的数量
    num_tensor_model_parallel_groups: int = (world_size //
                                             tensor_model_parallel_size)
    num_pipeline_model_parallel_groups: int = (world_size //
                                               pipeline_model_parallel_size)
    rank = torch.distributed.get_rank()
    # 构建tensor并行组
    global _TENSOR_MODEL_PARALLEL_GROUP
    assert _TENSOR_MODEL_PARALLEL_GROUP is None, (
        "tensor model parallel group is already initialized")
    for i in range(num_tensor_model_parallel_groups):
        ranks = range(i * tensor_model_parallel_size,
                      (i + 1) * tensor_model_parallel_size)
        group = torch.distributed.new_group(ranks)
        if rank in ranks:
            _TENSOR_MODEL_PARALLEL_GROUP = group
    # 构建pipeline并行组
    global _PIPELINE_MODEL_PARALLEL_GROUP
    global _PIPELINE_GLOBAL_RANKS
    assert _PIPELINE_MODEL_PARALLEL_GROUP is None, (
        "pipeline model parallel group is already initialized")
    for i in range(num_pipeline_model_parallel_groups):
        ranks = range(i, world_size, num_pipeline_model_parallel_groups)
        group = torch.distributed.new_group(ranks)
        if rank in ranks:
            _PIPELINE_MODEL_PARALLEL_GROUP = group
            _PIPELINE_GLOBAL_RANKS = ranks
```

3. **Worker.load_model()->ModelRunner.load_model()->get_model()**

调用model_executor/model_loader.py中的get_model函数，将模型及其权重载入，并将模型保存在ModelRunner的model成员变量中。其中hf_config指的是hugging face config，里面包含了模型的超参数、输入输出等信息

```
def get_model(model_config: ModelConfig) -> nn.Module:
    # 获取模型的类，定义在model_executor/models中，通过importlib动态导入
    model_class = _get_model_architecture(model_config.hf_config)

    linear_method = None
```

```
    # 从模型的量化配置中获取对应的线性量化方案，用于指导模型的初始化。模型量化的主要功能在于建立一        种浮点数据和定点数据间的映射关系，使得以较小的精度损失代价获得了较大的内存和计算效率收益
    if model_config.quantization is not None:
        quant_config = get_quant_config(model_config.quantization,
                                        model_config.model,
                                        model_config.hf_config,
                                        model_config.download_dir)
        # 要求设备计算能力达到quantilization的最低要求
        capability = torch.cuda.get_device_capability()
        capability = capability[0] * 10 + capability[1]
        if capability < quant_config.get_min_capability():
            raise ValueError(...)
        supported_dtypes = quant_config.get_supported_act_dtypes()
        if model_config.dtype not in supported_dtypes:
            raise ValueError(...)
        # 获得模型的线性量化方案，如AWQ
        linear_method = quant_config.get_linear_method()

    with _set_default_torch_dtype(model_config.dtype):
        # 创建一个模型实例，其基于model_class类，利用hf_config和线性量化方案初始化
        with torch.device("cuda"):
            model = model_class(model_config.hf_config, linear_method)
        if model_config.load_format == "dummy":
            # dummy格式的模型采用随机的参数，作者标注才用这个方法是为了更精确的性能评估
            initialize_dummy_weights(model)
        else:
            # 加入模型的参数，不同模型定义了自己的载入参数的方法
            model.load_weights(model_config.model, model_config.download_dir,
                               model_config.load_format, model_config.revision)

    # 设置模型为评估模式，避免模型计算梯度、执行dropout层等，获得稳定可靠的输出
    return model.eval()
```

- **执行profiling，获得各个worker的可用块信息，并初始化KV cache：LLMEngine._init_cache()**

  该函数评估可用的内存总量并计算出最大可分配的GPU块和CPU块数量，并基于此初始化各个worker的KV cache。当存在多个worker时，该函数取所有worker可用块的最小值，保证所有worker的块的分配

```
def _init_cache(self) -> None:
    # 各个worker调用profile_num_available_blocks函数获得所有worker可用块的数量
    num_blocks = self._run_workers(
        "profile_num_available_blocks",
        block_size=self.cache_config.block_size,
        gpu_memory_utilization=self.cache_config.gpu_memory_utilization,
        cpu_swap_space=self.cache_config.swap_space_bytes,
        cache_dtype=self.cache_config.cache_dtype,
    )
    # 可用块设置为所有worker的最小值
    num_gpu_blocks = min(b[0] for b in num_blocks)
    num_cpu_blocks = min(b[1] for b in num_blocks)
    logger.info(f"# GPU blocks: {num_gpu_blocks}, "
                f"# CPU blocks: {num_cpu_blocks}")
    if num_gpu_blocks <= 0:
        raise ValueError("...")
    # 计算出可支持的最大的序列长度
    max_seq_len = self.cache_config.block_size * num_gpu_blocks
    if self.model_config.max_model_len > max_seq_len:
        raise ValueError(...)
    self.cache_config.num_gpu_blocks = num_gpu_blocks
    self.cache_config.num_cpu_blocks = num_cpu_blocks
    # 初始化KV cache并预热模型.
    self._run_workers("init_cache_engine", cache_config=self.cache_config)
    self._run_workers("warm_up_model")
```

  1. **Worker.profile_num_available_blocks()**

     Worker通过令模型执行一次prefill阶段以获得模型运行时的内存峰值(prefill阶段模型往往需要更大的内存，因为其同时考虑的token数量更大，考虑所有token之间的矩阵乘法运算。而autoregressive只考虑单个token与其他token的向量-矩阵运算)。基于此，Worker计算出可以用于存储KV cache的GPU和CPU的物理块的数量

```
def profile_num_available_blocks(
    self,
```

```python
    block_size: int,
    gpu_memory_utilization: float,
    cpu_swap_space: int,
) -> Tuple[int, int]:
    # 清空GPU内的缓存以释放被占用的GPU内存
    torch.cuda.empty_cache()
    # 令模型执行一次prefill阶段的运行，获得模型运行时使用的峰值内存
    self.model_runner.profile_run()
    # 计算出当前设备执行模型时利用的的最大内存总量
    torch.cuda.synchronize()
    free_gpu_memory, total_gpu_memory = torch.cuda.mem_get_info()
    peak_memory = total_gpu_memory - free_gpu_memory
    # 获取每个块占用的内存总量，这里get_cache_block_size为类函数，实际上还没有创建类对象
    cache_block_size = CacheEngine.get_cache_block_size(
        block_size, self.model_config, self.parallel_config)
    # 计算出gpu和cpu块的数量
    num_gpu_blocks = int(
        (total_gpu_memory * gpu_memory_utilization - peak_memory) //
        cache_block_size)
    num_cpu_blocks = int(cpu_swap_space // cache_block_size)
    num_gpu_blocks = max(num_gpu_blocks, 0)
    num_cpu_blocks = max(num_cpu_blocks, 0)
    # 避免profiling的运行影响GPU内存
    torch.cuda.empty_cache()
    return num_gpu_blocks, num_cpu_blocks
```

- **ModelRunner.profile_run()**

  ModelRunner用配置中最大可支持的序列数量和token数量生成相应数量的随机输入序列，并支持top-k采样，执行模型的prefill阶段，以获得模型运行时所需要的最大内存(这里先不详细说明execute_model和kv_caches的运行)

```python
def profile_run(self) -> None:
    # 支持top-k采用以反映真实的内存使用量
    vocab_size = self.model_config.get_vocab_size()
    sampling_params = SamplingParams(top_p=0.99, top_k=vocab_size - 1)
    # 生成最大可支持的序列数量和token数量的输入进行profiling，获得内存使用量峰值
    max_num_batched_tokens = self.scheduler_config.max_num_batched_tokens
    max_num_seqs = self.scheduler_config.max_num_seqs
    seqs: List[SequenceGroupMetadata] = []
    for group_id in range(max_num_seqs):
        seq_len = (max_num_batched_tokens // max_num_seqs +
                   (group_id < max_num_batched_tokens % max_num_seqs))
        seq_data = SequenceData([0] * seq_len)
        seq = SequenceGroupMetadata(
            request_id=str(group_id),
            is_prompt=True,
            seq_data={group_id: seq_data},
            sampling_params=sampling_params,
            block_tables=None,
        )
        seqs.append(seq)
    # 利用上述随机输入序列执行模型
    num_layers = self.model_config.get_num_layers(self.parallel_config)
    kv_caches = [(None, None)] * num_layers
    self.execute_model(seqs, kv_caches)
    torch.cuda.synchronize()
    return
```

- **CacheEngine.get_cache_block_size()**

  计算出每个block所占的大小，其中一个token所占的KV cache大小为 $2 * num\_heads * head\_size * num\_layers * dtype\_size$，分别表示张量并行下当前GPU设备分配的head数量(num_heads)、每个key向量的元素数量(head_size)、隐藏层的数量(num_layers，每个隐藏层存储一个cache)。不同模型在元素数量和head数量的计算上可能有不同

```python
def get_cache_block_size(
    block_size: int,
    model_config: ModelConfig,
    parallel_config: ParallelConfig,
) -> int:
    head_size = model_config.get_head_size()
```

```python
        num_heads = model_config.get_num_kv_heads(parallel_config)
        num_layers = model_config.get_num_layers(parallel_config)

        key_cache_block = block_size * num_heads * head_size
        value_cache_block = key_cache_block
        total = num_layers * (key_cache_block + value_cache_block)
        dtype_size = _get_dtype_size(model_config.dtype)
        return dtype_size * total
```

2. **Worker.init_cache_engine()**

   初始化cache的配置，为每层分配GPU和CPU cache空间，并设置model_runner的块大小。这里分配的GPU cache事实上就是用于后续计算的KV cache

   疑问：这里的allocate_gpu_cache创建了对应的torch向量，它保存在了CPU内存中，这是否影响了CPU内存的可换入换出空间

```python
def init_cache_engine(self, cache_config: CacheConfig) -> None:
    self.cache_config = cache_config
    self.cache_engine = CacheEngine(self.cache_config, self.model_config,
                                    self.parallel_config)
    self.cache_events = self.cache_engine.events
    self.gpu_cache = self.cache_engine.gpu_cache
    self.model_runner.set_block_size(self.cache_engine.block_size)

# CacheEngine.__init__()调用CacheEngine.allocate_gpu_cache()函数分配GPU cache
def allocate_gpu_cache(self) -> List[KVCache]:
    gpu_cache: List[KVCache] = []
    # (num_heads,head_size,block_size,x)
    key_block_shape = self.get_key_block_shape()
    # 与key有些不同，shape为(num_heads,head_size,block_size)，对后续有什么影响
    value_block_shape = self.get_value_block_shape()
    for _ in range(self.num_layers):
        key_blocks = torch.empty(
            # *表示拆开对应的tuple
            size=(self.num_gpu_blocks, *key_block_shape),
            dtype=self.dtype,
            device="cuda",
        )
        value_blocks = torch.empty(
            size=(self.num_gpu_blocks, *value_block_shape),
            dtype=self.dtype,
            device="cuda",
        )
        gpu_cache.append((key_blocks, value_blocks))
    return gpu_cache

# CacheEngine.__init__()初始化event用于后续的synchronization
self.events = [torch.cuda.Event() for _ in range(self.num_layers)]
```

3. **Worker.warm_up_model()->ModelRunner.capture_model()->CUDAGraphRunner.capture()**

   疑问：捕获模型的计算图如何对后续的运行造成影响，是否与graph_runner有关

```python
def warm_up_model(self) -> None:
    # 捕获模型的计算图，以进行模型优化和编译。计算图支持的最大长度有限制，当一个序列长度大于
model_config.max_context_len_to_capture时，转为eager模式运行
    if not self.model_config.enforce_eager:
        self.model_runner.capture_model(self.gpu_cache)
    # 重新设置种子避免模型初始化和profiling的影响
    set_random_seed(self.model_config.seed)

def capture_model(self, kv_caches: List[KVCache]) -> None:
    # 生成随机的输入序列用于进行capture操作
    max_batch_size = max(_BATCH_SIZES_TO_CAPTURE)
    input_tokens = torch.zeros(max_batch_size, 1, dtype=torch.long).cuda()
    input_positions = torch.zeros(max_batch_size, 1,dtype=torch.long).cuda()
    slot_mapping = torch.empty(max_batch_size, 1, dtype=torch.long).cuda()
    slot_mapping.fill_(_PAD_SLOT_ID)
    context_lens = torch.ones(max_batch_size, dtype=torch.int32).cuda()
    block_tables = torch.from_numpy(self.graph_block_tables).cuda()
    # 从大batch到小batch进行遍历，以减少内存使用量
    for batch_size in reversed(_BATCH_SIZES_TO_CAPTURE):
```

```python
        input_metadata = InputMetadata(
            is_prompt=False,
            slot_mapping=slot_mapping[:batch_size],
            max_context_len=self.max_context_len_to_capture,
            context_lens=context_lens[:batch_size],
            block_tables=block_tables[:batch_size],
            use_cuda_graph=True,
        )
        graph_runner = CUDAGraphRunner(self.model)
        graph_runner.capture(
            input_tokens[:batch_size],
            input_positions[:batch_size],
            kv_caches,
            input_metadata,
            memory_pool=self.graph_memory_pool,
        )
        # 后面的capture操作可以利用大的batch生成的内存池建图，减少内存使用
        self.graph_memory_pool = graph_runner.graph.pool()
        self.graph_runners[batch_size] = graph_runner

def capture(self, input_ids: torch.Tensor, positions: torch.Tensor, kv_caches: List[KVCache],
input_metadata: InputMetadata, memory_pool) -> None:
    assert self.graph is None
    # 执行一遍模型保证捕获的计算图不包含初始的内核启动部分
    self.model(
        input_ids,
        positions,
        kv_caches,
        input_metadata,
    )
    torch.cuda.synchronize()
    # 捕获计算图以进行计算优化提升执行效率
    self.graph = torch.cuda.CUDAGraph()
    with torch.cuda.graph(self.graph, pool=memory_pool):
        hidden_states = self.model(
            input_ids,
            positions,
            kv_caches,
            input_metadata,
        )
    torch.cuda.synchronize()
    # 保存输入和输出的缓存
    self.input_buffers = {
        "input_ids": input_ids,
        "positions": positions,
        "kv_caches": kv_caches,
        "slot_mapping": input_metadata.slot_mapping,
        "context_lens": input_metadata.context_lens,
        "block_tables": input_metadata.block_tables,
    }
    self.output_buffers = {"hidden_states": hidden_states}
    return
```