

PR #2845 完整报告

radixark/miles

[multi-lora v2 6/7] Tinker gateway runtime and entry point

合并时间: 2026-09-16 16:26

原文链接: <http://prhub.com.cn/radixark/miles/pull/2845>

执行摘要

- 一句话: 新增 Tinker 网关运行时与 `serve` 入口
- 推荐动作: 值得精读, 是本轮 multi-LoRA v2 栈中最靠近「真实执行链路」的一篇。阅读顺序建议: 先看 `miles/tinker/runtime.py` 的 `_pad_to_dp_multiple` / `_build_train_data` / `_execute_batch` 三件套, 理解 `datum` 契约与 `sample_index` 重组规则; 再看 `serve_tinker.py` 的 `serve()`, 重点体会任务监督 (`asyncio.wait(FIRST_COMPLETED)`)、`dispose` 顺序与 `GatewayConfig` 的推导; 最后扫 `miles/utils/arguments.py` 的 `entry` 维度, 确认对既有 `train` 入口没有隐性行为回归。需要重点关注的设计决策有四条: 训练命令经 object store 传递并让 worker 自行按 DP 切分、用零掩码副本做 DP 补齐、显式失败透传与未知失败终止网关的失败分类边界、以及 `NaN` / `MASK_LOGPROB` 的 wire 约定。建议精读时反向确认 `TrainerCell.execute(kill_on_failure=True)` 与 Tinker 服务错误分类是否已完全对齐, 并确认 `_build_train_data` 依赖首条 `datum` 探测可选键的假设在真实 SDK 用法下是否成立。

功能与动机

PR body 的表述是「Connect the protocol service to Miles through `MilesBackend` and `serve_tinker.py`」: dataset 驱动的 multi-LoRA v1 后端已被移除, 改由 Tinker 协议承载——客户端对 adapter slot 显式提交 `forward_backward` / `optim_step`, 服务端不再拥有数据集与调度 (`docs/advanced/lora.md` 的改动原文: "Multi-LoRA training is served through the Tinker protocol: `serve_tinker.py` turns a miles deployment into a training service where each client drives its own adapter slot with explicit `forward_backward` / `optim_step` operations (the server owns no datasets or schedules)"). 协议层与槽位机制在前序切片中已完成, 本 PR 要解决的是「最后一公里」: 训练命令如何落到 Megatron actor、采样如何经 router 命中不可变 adapter、失败如何分类并终止服务。由于 `serve` 场景没有 rollout/GBS 语义, 还必须为参数解析新增 `entry` 维度, 否则 `--rollout-batch-size` 等训练必填项会阻塞网关启动。

实现拆解

1. 入口与参数契约 (`serve_tinker.py` + `miles/utils/arguments.py`): `parse_args` 新增 `entry` 形参, 断言只接受 `"train"` / `"serve"` 并写入 `args.entry`; `add_miles_arguments` 用 `parser.set_defaults(entry="train")` 兜底, 保证既有调用方行为不变。`--rollout-batch-size` 从 `required=True` 降级为 `default=None`, 其「必填」语义下沉到 `miles_validate_args` 中

新增的 `args.entry == "train"` 分支，使 `serve` 入口可以不带任何 rollout 语义启动。新增 `--tinker-server-port`（默认 10613）、`--tinker-base-model`（默认 `--hf-checkpoint`）、`--tinker-checkpoint-root`（默认 `<save>/tinker`）三个网关参数。

2. `serve()` 的启动编排 (`serve_tinker.py`)：函数开头集中做前置断言——必须开启 multi-LoRA、`--load` 必须等于 `--hf-checkpoint`（训练与推理共用同一份冻结基座）、`checkpoint` 根目录必须可推导；随后从 HF config 取 `max_position_embeddings`，并在设置 `--max-tokens-per-gpu` 时与 `trainer` 的 padding 块 (`tensor_model_parallel_size × data_pad_size_multiplier`) 取小，得到 `GatewayConfig.max_tokens_per_datum`。启动顺序为 `worker manager` → `object store` (`contribute_segment=False`) → `InferenceController` → `TrainerController(role="actor", with_ref=False, inference_controller=None, rollout_executor=None)` → 组装 `GatewayConfig` 与 `TinkerService(MilesBackend(...))`；`target_modules` 先经 `convert_target_modules_to_hf` 归一化，再推导 `trains_attn` / `trains_mlp` / `trains_unembed`。`dp_size` 由 `actor_num_nodes × actor_num_gpus_per_node ÷ (TP × PP × CP)` 算出并注入 `MilesBackend`。最后 `service.run()` 与 `uvicorn` 的 `server.serve()` 用 `asyncio.wait(..., return_when=FIRST_COMPLETED)` 互相监督，任一方结束都会取消另一方，并在正常完成路径上按 `inference controller` → `trainer` 顺序 `dispose`。`__main__` 强制 `use_dynamic_global_batch_size=True` 与 `delay_split_train_data_by_dp=True`，因为一条训练命令就是一个 batch。
3. 训练路径 (`miles/tinker/runtime.py` 的 `MilesBackend`)：`_build_train_data` 把 slot datum 列表翻译成 `miles` 的 `train_data`——`tokens` / `target_tokens` / `loss_masks` / `response_lengths` / `total_lengths` / `sample_indices` / `adapter_slots` / `dynamic_global_batch_size`，并通过 `DATUM_TO_BATCH_KEYS` 选择性带上 `loss_weights` / `advantages` / `rollout_log_probs`；`_pad_to_dp_multiple` 在 batch 构造前用「最后一条 datum 的零掩码副本」把样本数补齐到 `dp_size` 的整数倍。`_call_trainer` 把 `train_data` 放进 `object store` (`ROLLOUT_DATA_VALUE_SPEC`) 后以 `data_ref` 调用 `trainer` RPC，`finally` 中回收引用；`_execute_batch` 收齐各 rank 的 `per_datum` 结果后按 `sample_index` 重组，只返回调用方真实提交的 datum，显式 `error` 结果原样透传。slot 生命周期 RPC (`load_slot` / `unload_slot` / `save_slot` / `export_slot`) 统一用 `_slot_failure` 取出首个非 `None` 的失败结果，`optim_step` 取 `worker_results[0]`，`trainer_dead()` 转发到 `TrainerController.has_errored_cell()`。
4. 采样路径 (`MilesBackend.sample` / `_generate_request`)：把 payload 编译成 SGLang `/generate` 请求并用 `asyncio.gather` 开 `num_samples` 份；`_with_sample_seed` 在调用方指定 `seed` 时按 `index` 递增，避免 `N` 条样本重复。`stop` 语义按 Tinker 定义区分处理——`[]` 映射为 `ignore_eos=True`、整数列表映射为 `stop_token_ids`、其余映射为 `stop`。LoRA 请求带上不可变的 adapter 名 `lora_path`，并通过 `lora_backfill_paths` 让引擎自行回填被驱逐的磁盘版本。`_prompt_logprobs` / `_topk_prompt_logprobs` 把 SGLang 的 `meta_info` 翻译成 Tinker 期望的形状，缺失位置用 `NaN` 和 `(0, MASK_LOGPROB)` 占位；`_to_sequence` 把 `finish_reason.type == "abort"` 直接转成 `error`，避免截断样本被客户端当成成功结果继续训练。
5. 失败契约与配套改动（本 PR 无新增测试）：`miles/ray/train/group.py` 新增 `TrainerController.has_errored_cell()`，让网关能感知 cell 已被 kill；

miles/backends/megatron_utils/model.py 把 get_optimizer_param_scheduler 的返回类型放宽为 OptimizerParamScheduler | None，与 serve 入口没有 rollout 迭代的调用路径对齐（docstring 的 Returns 描述未同步更新）；miles/tinker/server/encoding.py 在 render_result 的 sample 分支把 NaN 转成 JSON 的 null、剥掉 MASK_LOGPROB 占位、整行被掩码时输出 None；docs/advanced/lora.md 把多 LoRA 描述重写为「由 Tinker 协议承载、客户端驱动自己的 adapter slot」，并删除了过时的 Internals 清单。测试与示例全部落在栈内 #2849，本 PR 的验证叙述（checkpoint 根目录拒绝探测、slot checkpoint 续训、pre-commit）都在栈级 body 中给出。

关键文件：

- miles/tinker/runtime.py（模块 网关运行时；类别 source；类型 core-logic；符号 _pad_to_dp_multiple, _build_train_data, MilesBackend, MilesBackend._execute_batch）：新增的核心运行时：MilesBackend 定义 gateway datum 到 miles train_data 的翻译契约、DP 补齐、对象存储传递训练命令、per-datum 输出重组与 router 采样，是整个 Tinker 网关的边界层。
- serve_tinker.py（模块 服务入口；类别 source；类型 entrypoint；符号 serve, main）：独立 serve 入口：把 worker manager、object store、InferenceController、TrainerController、TinkerService 与 uvicorn 编排起来，并定义了服务循环与 HTTP server 的互相监督及 dispose 顺序。
- miles/utils/arguments.py（模块 参数解析；类别 source；类型 configuration；符号 parse_args, get_miles_extra_args_provider, miles_validate_args, add_lora_arguments）：引入 entry 维度与 --tinker-* 参数：把 --rollout-batch-size 从 argparse 必填改为按 entry 校验，使 serve 入口可以不带 rollout 语义启动，是共享解析层的行为变化点。
- miles/tinker/server/encoding.py（模块 协议编码；类别 source；类型 data-contract；符号 render_result, _tensor_json）：补完 sample 响应的 wire 编码：把 runtime 的 NaN 占位转为 JSON null，剥掉 MASK_LOGPROB 占位，保证 SDK 侧拿到合法 JSON。
- miles/ray/train/group.py（模块 训练控制器；类别 source；类型 core-logic；符号 TrainerController.has_errored_cell）：新增 TrainerController.has_errored_cell()，为网关提供「训练 cell 是否已被 kill」的判定入口，是失败契约对齐的关键钩子。
- miles/backends/megatron_utils/model.py（模块 模型后端；类别 source；类型 data-contract；符号 get_optimizer_param_scheduler）：把 get_optimizer_param_scheduler 的返回类型放宽为 OptimizerParamScheduler | None，与 serve 入口下没有 rollout 迭代语义的调用路径对齐。
- docs/advanced/lora.md（模块 文档；类别 docs；类型 documentation）：把 multi-LoRA 章节重写为 Tinker 协议叙事（serve_tinker.py 作为服务入口、槽位机制在 lora/ 下），并删除过时的 Internals 清单，是对外说明的唯一同步文档。

关键符号：_pad_to_dp_multiple, _build_train_data, MilesBackend.init, MilesBackend.trainer_dead, MilesBackend._execute_batch, MilesBackend._call_trainer, MilesBackend.sample, MilesBackend._generate_request, MilesBackend._slot_failure, _to_sequence, _prompt_logprobs, serve, parse_args, TrainerController.has_errored_cell, render_result, get_optimizer_param_scheduler

关键源码片段

miles/tinker/runtime.py

新增的核心运行时：**MilesBackend** 定义 gateway datum 到 miles train_data 的翻译契约、DP 补齐、对象存储传递训练命令、per-datum 输出重组与 router 采样，是整个 Tinker 网关的边界层。

```
# internal datum key -> trainer batch key: Tinker datum 键到 miles 训练 batch 键的映射
DATUM_TO_BATCH_KEYS = {"weights": "loss_weights", "advantages": "advantages", "sampling_logprobs": "rollout_log_probs"}
```

```
def _pad_to_dp_multiple(slot_datums: list, dp_size: int) -> list:
    """Equalize DP shares with zero-mask datums; their outputs are dropped after the loss pass.
    """
    # 一条训练命令必须均分到整个 DP 组，否则 worker 侧分片不齐会直接卡死
    remainder = len(slot_datums) % dp_size
    if remainder == 0:
        return slot_datums
    slot, datum = slot_datums[-1]
    # 副本复用最后一条 datum 的 tokens，只把 padding 标记打开
    filler = dict(datum) | {"padding": True}
    return slot_datums + [(slot, filler)] * (dp_size - remainder)
```

```
def _build_train_data(slot_datums: list) -> dict:
    """Miles response_lengths select label positions here, including prompt targets."""
    datums = [datum for _, datum in slot_datums]
    train_data = {
        "tokens": [datum["tokens"] for datum in datums],
        "target_tokens": [datum["target_tokens"] for datum in datums],
        # padding datum 的 loss_mask 全 0，所以它的 loss/logprob 会在重组时被丢弃
        "loss_masks": [[0 if datum.get("padding") else 1] * datum["target_len"] for datum in datums],
        "response_lengths": [datum["target_len"] for datum in datums],
        "total_lengths": [len(datum["tokens"]) for datum in datums],
        # sample_index 是跨 DP rank 重新组装 per-datum 输出的唯一键
        "sample_indices": list(range(len(datums))),
        # adapter_slots 让不同的 LoRA slot 共享同一次前反向
        "adapter_slots": [slot for slot, _ in slot_datums],
        # 一条命令就是一个 batch，绕开 miles 固定 global batch size 的切分逻辑
        "dynamic_global_batch_size": len(datums),
    }
    # 可选键以首条 datum 为准：Tinker 的 loss_fn 决定 weights/advantages 是否存在
    for datum_key, batch_key in DATUM_TO_BATCH_KEYS.items():
        if datum_key in datums[0]:
            train_data[batch_key] = [datum[datum_key] for datum in datums]
    return train_data
```

```

class MilesBackend:
    # ... __init__ / trainer_dead / load_slot / unload_slot / save_slot / export_slot 略 ...

    async def _execute_batch(
        self, method: str, batch_id: int, slot_datums: list, loss_fn: str, loss_fn_config: dict
    ) -> list[dict] | dict:
        # DP 补齐必须在构造 batch 之前完成, padding 的位置决定后续 index 区间
        train_data = _build_train_data(_pad_to_dp_multiple(slot_datums, self.dp_size))
        train_data["loss_fn"] = loss_fn
        train_data["loss_fn_config"] = loss_fn_config
        worker_results = await self._call_trainer(method, batch_id, train_data)
        by_index: dict[int, dict] = {}
        for worker_result in worker_results:
            if "error" in worker_result:
                # actor 侧的显式失败原样透传给网关, 由网关判定是否属于用户错误
                return worker_result
            for datum_output in worker_result["per_datum"]:
                index = int(datum_output["sample_index"])
                if index not in by_index: # 重复输出与 padding 只保留第一份
                    by_index[index] = {
                        "loss": float(datum_output["loss"]),
                        "logprobs": datum_output["logprobs"].tolist(),
                    }
        # 只取调用方真实提交的 datum, 尾部 padding 的 index 落在区间之外
        return [by_index[index] for index in range(len(slot_datums))]

    async def _call_trainer(self, method: str, batch_id: int, train_data: dict) -> list:
        store = object_store.get_instance()
        # 训练命令先落对象存储, worker 自己按 DP 切分并就地读取, driver 不做二次分发
        data_ref = store.put(value=train_data, value_spec=ROLLOUT_DATA_VALUE_SPEC)
        try:
            return await getattr(self.trainer, method)(batch_id=batch_id, data_ref=data_ref)
        finally:
            store.remove(data_ref) # 成功与失败都要回收, 否则对象存储持续增长

```

serve_tinker.py

独立 serve 入口: 把 worker manager、object store、InferenceController、TrainerController、TinkerService 与 uvicorn 编排起来, 并定义了服务循环与 HTTP server 的互相监督及 dispose 顺序。

```

async def serve(args):
    # 前置校验集中在最前: 任何一条不满足都在拉起 GPU 栈之前失败
    assert args.multi_lora, "serve_tinker requires --multi-lora-n-adapters > 0"
    assert args.load == args.hf_checkpoint, "Tinker trainers and engines must load the same frozen HF base"
    checkpoint_root = args.tinker_checkpoint_root or (args.save and f"{args.save}/tinker")
    assert checkpoint_root, "set --tinker-checkpoint-root (or --save to derive <save>/tinker)"
    hf_config = load_hf_config(args.hf_checkpoint)
    # 单条 datum 的上限同时受模型上下文与 trainer 的最小 padding 块约束

```

```

max_tokens_per_datum = hf_config.max_position_embeddings
if args.max_tokens_per_gpu is not None:
    # The trainer pads each packed microbatch to this multiple.
    pad_size = args.tensor_model_parallel_size * args.data_pad_size_multiplier
    trainer_token_limit = args.max_tokens_per_gpu // pad_size * pad_size
    max_tokens_per_datum = min(max_tokens_per_datum, trainer_token_limit)
assert max_tokens_per_datum > 0, "trainer token budget must fit at least one padding block"

# ... configure_logger / init_http_client / launch_worker_manager / object_store.init_instance
略 ...

trainer = TrainerController(
    args=args,
    role="actor",
    with_ref=False,
    with_opd_teacher=False,
    # 采样走 SGLang router, 训练侧不持有推理控制器也不参与权重更新窗口
    inference_controller=None,
    rollout_executor=None,
)
await trainer.init()

# target modules 先归一化成 HF 名, 再据此推导 LoRA 覆盖范围
target_modules = set(convert_target_modules_to_hf(args.target_modules))
config = GatewayConfig(
    base_model=args.tinker_base_model or args.hf_checkpoint,
    n_slots=args.multi_lora_n_adapters,
    checkpoint_root=checkpoint_root,
    vocab_size=hf_config.vocab_size,
    max_tokens_per_datum=max_tokens_per_datum,
    lora_alpha=args.lora_alpha,
    max_lora_rank=args.lora_rank,
    trains_attn=bool(target_modules & {"q_proj", "k_proj", "v_proj", "o_proj"}),
    trains_mlp=bool(target_modules & {"gate_proj", "up_proj", "down_proj"}),
    trains_unembed="lm_head" in target_modules,
)
# ... router_url / dp_size / TinkerService(MilesBackend(trainer, router_url, dp_size=dp_size))
略 ...

# supervise both: dispatcher 崩溃必须把 HTTP server 一起带走,
# 否则 /healthz 仍在应答, 而所有训练 future 永久挂起
service_task = asyncio.create_task(service.run())
server_task = asyncio.create_task(server.serve())
try:
    done, _ = await asyncio.wait({service_task, server_task}, return_when=asyncio.FIRST_COMPLETED)
    for task in done:
        task.result() # 在 serve() 内重新抛出子任务异常
finally:

```

```

for task in (service_task, server_task):
    task.cancel()
    with suppress(asyncio.CancelledError):
        await task

# 顺序与 train.py 一致：先推理控制器，再 trainer；只覆盖正常完成路径
await inference_controller.dispose()
await trainer.dispose()

if __name__ == "__main__":
    args = parse_args(entry="serve")
    # 命令一次只提交一个 work unit，其大小就是 batch size
    args.use_dynamic_global_batch_size = True
    args.delay_split_train_data_by_dp = True
    asyncio.run(serve(args))

```

miles/tinker/server/encoding.py

补完 sample 响应的 wire 编码：把 runtime 的 NaN 占位转为 JSON null，剥掉 MASK_LOGPROB 占位，保证 SDK 侧拿到合法 JSON。

```

if op == "sample":
    rendered = {"type": "sample", "sequences": result["sequences"]}
    if result.get("prompt_logprobs") is not None:
        # runtime 用 NaN 表示「该位置没有可评分的 token」；JSON 不允许 NaN，转成 null
        rendered["prompt_logprobs"] = [
            None if math.isnan(logprob) else logprob for logprob in result["prompt_logprobs"]
        ]
    if result.get("topk_prompt_logprobs") is not None:
        topk = result["topk_prompt_logprobs"]
        # runtime 用 (0, MASK_LOGPROB) 填充不足 k 的位置，编码层负责剥离占位符
        rendered["topk_prompt_logprobs"] = [
            [
                (token_id, logprob)
                for token_id, logprob in zip(ids, probs, strict=True)
                if (token_id, logprob) != (0, MASK_LOGPROB)
            ]
            or None # 整行都被掩码时输出 null，而不是空列表
            for ids, probs in zip(topk["token_ids"], topk["logprobs"], strict=True)
        ]
    return rendered

```

评论区精华

本轮 review 由 nanjiangwill 给出「三项正确性 / 可用性关注 + 两项显式非阻塞的生命周期建议」，作者逐条以“Addressed:”回复，最终由 yushengsu-thu approve。

1) 训练 cell 被杀与服务失败语义错位（最重要）：reviewer 指出

`TrainerController._execute_slots()` 调用 `TrainerCell.execute()` 时用的是默认

`kill_on_failure=True`, 「any actor exception marks the only trainer cell errored and kills its workers. `TinkerService` instead treats batch failures as recoverable and calls `zero_grads()` before continuing」, 结果是清理逻辑对着一个已死的 cell 执行、派发随之停摆; 创建 / 加载失败同样会让服务「活着但没有可用的 trainer」。结论是补齐契约: 显式 actor 失败结果透传给网关, 未知训练失败沿用 `TrainerCell` 边界并由 supervisor 终止当前单 cell 服务, CUDA/NCCL 类故障保持 cell-fatal。

2) `abort` finish reason 被吞成成功样本: reviewer 指出 `_to_sequence()` 把所有非 `length` 的 finish reason 都映射为 `stop`, 「SGLang returns `type=\"abort\"`, and miles' normal rollout path maps that to `Sample.Status.ABORTED`; here it becomes a successful partial sample that the client can train on」, 建议对 `abort` 抛错。最终实现改为返回 `{"error": "the engine aborted this sample; resubmit the request"}`。

3) 服务循环与 HTTP server 的监督: `service.run()` 与 `uvicorn` 现在一起被监督, 「completion or failure of either task tears down the other」, 避免 dispatcher 已崩溃而 `/healthz` 仍在应答、所有训练 future 永久挂起。

4) 非阻塞项一: 把 `checkpoint_root` 的计算与校验提前到 `launch_worker_manager()` 之前, 避免配置非法时先拉起 worker 与 GPU 栈再断言失败——最终 `serve()` 开头的断言顺序已按要求前置。

5) 非阻塞项二: 建议在 `finally` 中 dispose `trainer` 与 `inference_controller` (停掉 watcher、ticker 与推理 server 客户端)。这一条被有意保留: PR body 明确写「Controller lifecycle: match `train.py` by disposing the inference controller, then the trainer, after normal completion of the serving tasks. No `AsyncExitStack` or initialization/exception cleanup guarantees are introduced」, 即只覆盖正常完成路径。

- 训练 cell 被 kill 与 Tinker 服务「可恢复失败」语义错位 (correctness): 按 reviewer 建议补齐契约: 显式 actor 失败结果经 `_slot_failure` / `error` 字段透传给网关, 未知训练失败沿用 `TrainerCell` 边界并由 supervisor 终止当前单 cell 服务, CUDA/NCCL 类故障保持 cell-fatal; 实现上新增 `TrainerController.has_errored_cell()` 与 `MilesBackend.trainer_dead()` 作为判定入口。
- `abort` finish reason 被映射为成功样本 (correctness): 改为对 `abort` 返回 `{"error": "the engine aborted this sample; resubmit the request"}`, 让请求在服务层结算为失败而不是静默返回截断序列。
- `service` 循环与 `uvicorn` 的互相监督 (design): `service.run()` 与 `server.serve()` 现在通过 `asyncio.wait(..., return_when=FIRST_COMPLETED)` 共同被监督, 并在 `finally` 中取消、等待两个任务。
- `checkpoint_root` 校验与资源初始化顺序 (design): 已采纳: `serve()` 开头的断言顺序前置, `checkpoint` 根目录与 `--load == --hf-checkpoint` 的检查都在 worker 初始化之前执行。
- 控制器 dispose 是否放进 `finally` (design): 有意保留现状: PR body 明确说明「Controller lifecycle: match `train.py` by disposing the inference controller, then the trainer, after normal completion of the serving tasks. No `AsyncExitStack` or initialization/exception cleanup guarantees are introduced」, 即只覆盖正常完成路径, 异常路径的清理由进程退出兜底。

风险与影响

- 风险：可用性：单 cell 服务的故障半径。serve_tinker.py 只为 actor 建一份 cell，而 TrainerCell.execute() 默认 kill_on_failure=True；一次 CUDA/NCCL 级故障会经 trainer_dead() → has_errored_cell() 终止整个网关。多租户并发下，一个租户触发的训练故障会连带中断其他租户。

正确性：_build_train_data 的同构假设。可选键只用首条 datum 探测 (if datum_key in datums[0])：第一条没有 weights 而后继 datum 有时，这些键会被静默丢弃；反之则抛 KeyError 直接变成 500，而不是 UserInputError 之类的结构化错误。

正确性：结果重组缺少缺项检查。_execute_batch 结尾是 [by_index[index] for index in range(len(slot_datums))], 若某个 rank 结果缺失或 sample_index 冲突，会抛 KeyError 而不是返回 {"error": ...}；同一 sample_index 的重复输出也只保留第一份，缺少显式日志。

性能：DP 补齐的无效算力。_pad_to_dp_multiple 最多复制 dp_size - 1 条样本，且复用的是最后一条 datum 的 token 长度；DP 越大、单命令批量越小时，padding 带来的无效 forward/backward 比例越高。

兼容性：参数校验时机后移。--rollout-batch-size 不再是 argparse 必填。

miles_validate_args 中 if args.num_steps_per_rollout is not None: 一段位于 entry 判断之前，serve 入口若误传 --num-steps-per-rollout 会以 None * ... 的 TypeError 失败，而不是可读的断言信息。

契约分散：logprob wire 格式。NaN 与 MASK_LOGPROB 的占位约定同时存在于 miles/tinker/runtime.py 与 miles/tinker/server/encoding.py，走 protobuf (proto_codec) 的路径需要各自实现同样语义，存在不一致风险。

启动门槛。assert args.load == args.hf_checkpoint 要求 serve 只能从冻结 HF 基座起，不能像 train 那样从 miles checkpoint 恢复；按训练习惯传 --load 的用户会在启动即失败。

测试覆盖。本 PR 未新增测试文件，MilesBackend 的翻译逻辑与 serve() 的监督 / 生命周期行为只能通过 #2849 的网关测试与 e2e 间接覆盖，栈级 body 也标注「Fresh hosted CPU/NVIDIA CI is pending」。

- 影响：对用户 / 调用方：Tinker SDK 客户端第一次可以真正连到一个自带训练与采样的 miles 部署——serve_tinker.py 是唯一入口，端口 (--tinker-server-port)、对外基座名 (--tinker-base-model)、checkpoint 根目录 (--tinker-checkpoint-root) 都是显式契约，tinker:// 路径语义由此确定。

对系统：新增一条与 train.py 平行的 serve 入口，通过 args.entry 隔离两者的参数校验；serve() 只创建 actor、with_ref=False、inference_controller=None，采样经 router 走磁盘回填，不参与权重更新窗口，因此不会改变既有训练链路的权重同步路径。

对共享层：miles/utils/arguments.py 是所有入口共用的参数解析，--rollout-batch-size 由 required=True 改为可选属于全局行为变化（主要影响报错时机与文案）；

miles/ray/train/group.py 新增方法、miles/backends/megatron_utils/model.py 放宽注解都是向后兼容的增量改动。

对团队 / 演进：本 PR 完成 7 篇栈的第 6 环，MilesBackend 定义的 datum↔train_data 契约将成为后续所有 Tinker 协议扩展的边界；第 7 环 #2849 负责示例、SDK 客户端与测试套件，本 PR 的复用点（测试文件、pr-2849 Docker 镜像）都指向它。

- 风险标记：核心路径变更，失败语义耦合 (kill_on_failure)，缺少本 PR 测试覆盖，参数校验时机后移，DP 补齐带来额外算力，wire 契约分散在两处

关联脉络

- PR #2849 [multi-lora v2 7/7] Tinker gateway example and tests: 直接叠加在本 PR 之上的第 7 篇：提供 serve_qwen3_30b_a3b_tinker.py 启动器、SDK 客户端与网关测试套件；PR body 指出 checkpoint 回归测试挂在该 PR 的测试文件里，运行环境也复用 pr-2849 Docker 镜像，本 PR 的功能验证主要依赖它。
- PR #2844 （标题未在上下文中提供）：PR body 明确写「Stacked on #2844」，本 PR 直接构建在其之上，属于同一 multi-lora v2 栈的相邻切片。
- PR #2840 （标题未在上下文中提供）：PR body 说明「Per-slot optimizer ownership is established in #2840」，本 PR 的 MilesBackend.optim_step 与 slot 生命周期 RPC 依赖其建立的 per-slot 优化器归属语义。
- PR #3198 （标题未在上下文中提供）：整个栈的最终集成点：PR body 多次以「the final #3198 tree exactly matches the reviewed implementation」「broad GPU selection remains on #3198」指代它，说明本 PR 的最终形态与 #3198 的评审树一致。