

PR #2844 完整报告

radixark/miles

[multi-lora v2 5/7] Tinker gateway protocol face

合并时间: 2026-09-16 16:25

原文链接: <http://prhub.com.cn/radixark/miles/pull/2844>

执行摘要

- 一句话: 新增 Tinker 网关协议层与每模型调度器, 纯进程内无 runtime 依赖
- 推荐动作: 值得精读, 尤其是三处: core/service.py 的失败分级与 _close_model 的 slot quarantine 语义、core/scheduler.py 的“arrival 选 seed + pack_key 打包 + optim barrier 合并”三件套、以及 review 线程本身——它展示了一个很典型的踩坑模式: 给外部 SDK 写服务端时, request_id 必须由 SDK 的 (session/seq) 幂等键派生, 自己随机生成的 id 在客户端重试下必然重复占资源。阅读时建议对照 #2845 的 runtime 接线和 #2849 的测试, 确认 review 中“Non-blocking”的几条 (完整 outcome map 校验、loss_fn_config_v2 优先级) 在最终实现里是否真的闭环。

功能与动机

Tinker SDK 客户端 (tinker 0.26.2) 需要一个可直连的网关: 既要遵守 SDK 的 wire 契约 (pydantic 请求模型 + protobuf 二进制 + zstd 压缩), 又要在单节点上把多个租户、多个 LoRA adapter (slot) 的训练与采样请求按 seq 顺序正确落到同一个 trainer 上。PR body 明确说这些协议层“do not import miles.ray or miles.backends; runtime wiring follows in #2845”, 也就是先把可单测、可复用的协议与调度语义定下来; 同时 body 强调失败隔离的边界: request-local 的 forward-only/sampling 失败只结束该请求, checkpoint load/save/export 的执行失败会终止共享 trainer cell 与服务, 而“automatic recovery and stream replay are not implemented”。

实现拆解

1) 定义内部数据契约与配置 (miles/tinker/core/types.py)。新增 CommandOp 枚举 (6 个 op) 并为每个 op 挂 is_batch() (只有 FORWARD_BACKWARD/FORWARD_ONLY 进 BatchUnit) 与 requires_model_close_on_failure() (FORWARD_BACKWARD/OPTIM_STEP/LOAD_STATE 失败要关模型); LOSS_INPUT_KEYS 把 wire 的 logprobs 映射到内部 sampling_logprobs, LOSS_FN_INPUTS 声明每个 loss_fn 必读的输入, 从而在进入共享 batch 之前就拒绝缺字段的 datum; GatewayConfig 集中承载配额 (max_datums_per_request、max_tokens_per_batch、max_samples_per_request、max_lora_rank)、lease_timeout_s=300、batch_token_budget=262144; Command.arrival 是全局提交序, ModelRecord 用 request_id_by_seq 做幂等表, SessionRecord/SamplingSessionRecord 保存 models_by_seq 与 samples_by_seq。datum 的约定写在模块 docstring 里: tokens =

model_input + target_tokens[-1:], target_len 计全部 label 位置。

2) 每模型严格顺序 + 跨模型打包的调度器 (core/model_queue.py、core/scheduler.py) 。ModelRequestQueue 只用 last_enqueued_seq_id 与 pending_by_seq 做顺序闸门, submit() 只放行 seq 连续的指令; open_batch_run() 返回队首连续的 batch run (遇到 barrier 或带 validation_error 的请求即停), ready_barrier() 只在前面 batch 全部完成后才返回头 barrier, 因此“同一模型的先后顺序”是队列层面的硬保证而不是调度器的启发式。RequestScheduler.schedule_next() 用全局 arrival 在 datum 与 barrier 之间选 seed (barrier 的 arrival 更早则优先), _pack_batch() 以 pack_key()=(op, loss_fn, sorted(loss_fn_config.items())) 为等价类、在 batch_token_budget 内跨模型跨租户打包 datum, _merge_barriers() 把不同模型的 OPTIM_STEP 合成一次 trainer 调用、其它 barrier 保持单条执行。这样做的收益是 GPU 批利用率与 optimizer 调用次数, 代价是打包失败时会放大到同 batch 的所有请求。

3) 服务编排与失败策略 (core/service.py、core/future.py) 。TinkerService.run() 是单事件循环: 每轮在 _trainer_lock 内先结算 ready_rejections (validation_error 结成为 user 失败), 再 schedule_next() 并执行 _run_batch()/_run_barrier(), 随后检查 backend.trainer_dead() 并主动退出; finally 里 cancel 并 gather 掉 sweeper、_create_tasks、_sample_tasks, 且用 return_exceptions=True 保证清理失败不会覆盖真正的退出原因。_observe_background_task 把后台任务异常写进 _background_error 并唤醒主循环。create_model 走两阶段: 先在锁外占 slot、建 future 与 ModelRecord、注册 ModelRequestQueue, 再由 _run_create_model 在锁内调 backend.load_slot(); _close_model 幂等, 负责移除队列、失败所有挂起请求、把原因写入有界 (4*n_slots) 的 _close_reasons, 并只在 unload_slot 成功后才把 slot 放回 free_slots。RequestFutureStore 提供 create/resolve/fail/get/request_id_for_retry, _FINISHED_TTL_S=3600, 过期后的重试由 request_id_for_retry 直接返回一个“result expired after retention”的终态失败, 不重放后端工作。

4) 协议边界与输入校验 (miles/tinker/server/app.py、server/encoding.py、server/proto_codec.py、core/input_validation.py、core/utils.py) 。app.py 用 _tenant() 从 X-API-Key 或 Bearer token 建立租户身份, 统一把 UserInputError 映射为 400、OwnershipError 映射为 403, retrieve_future 对 PENDING 的 future 做 30s 长轮询 (避免 SDK 无退避地空转), 并按 Accept 头在 JSON 与 protobuf 之间选择渲染器。encoding.py 把“pin 住的 SDK pydantic 模型”当作 wire 契约 (validate_against_sdk), decode_command 在信封可识别时把内容错误降级为携带 validation_error 的 payload, 使该请求仍能按 seq 在正确位置结成为 user 失败; build_datum/tensor_data_to_list 统一 1-D 与 CSR 稀疏输入。proto_codec.py 处理 zstd 解压、ForwardBackwardRequest 解析 (loss_fn_config_v2 优先)、dense/sparse_csr tensor 解码与结果二进制编码。input_validation.py 承担模型布局一致性 (train_attn/mlp/unembed 必须与服务器启动参数一致)、rank 区间、token id 区间、loss_fn 必读输入与长度对齐、seq_id >= 1、checkpoint 路径段白名单与元数据兼容性; utils.py 的 parse_tinker_path/resolve_checkpoint_dir 用 segment 白名单 + realpath 前缀断言防路径逃逸, read_checkpoint_metadata 用 sha256(tenant) 的 tenant_digest 做跨租户归属校验, resolve_sampler_checkpoint 明确拒绝从训练 checkpoint 采样。

5) 测试、配置与部署配套：本 PR 不含任何测试文件，也无可配置开关；三个新增的 init.py 只是包骨架。作者在 body 中说明回归测试放在 #2849 (tests/fast/tinker/*、tests/e2e/lora/test_slot_checkpoint_resume.py)，且“This implements failure handling and isolation; automatic recovery and stream replay are not implemented”。这意味着本 PR 定义的行为契约在合入时没有自动化验证网，只能依赖 review 与后继 PR 覆盖。

关键文件：

- miles/tinker/core/service.py (模块 服务编排；类别 source；类型 core-logic；符号 TinkerService, init, run, _observe_background_task)：网关的中枢：TinkerService 串起会话、模型、future 与 trainer 调度，定义了失败分级（请求级失败 / 模型流终止 / 服务退出）和 slot 生命周期，是 review 讨论的主战场（8 条意见中 6 条指向这里）。
- miles/tinker/core/scheduler.py (模块 请求调度；类别 source；类型 core-logic；符号 DatumRef, BatchUnit, BarrierUnit, RequestScheduler)：调度核心：DatumRef/BatchUnit/BarrierUnit 与 RequestScheduler.schedule_next 定义“先做谁、哪些可以合批、哪些 barrier 可以合并”，是跨模型跨租户利用率与顺序正确性的分界点。
- miles/tinker/core/model_queue.py (模块 模型队列；类别 source；类型 data-contract；符号 PendingRequest, record_output, pack_key, is_batch_op)：顺序闸门与完成度记账：ModelRequestQueue 用 last_enqueued_seq_id 保证每个模型内部严格 seq 顺序，PendingRequest.record_output/pack_key 决定了输出回填与可打包等价类。
- miles/tinker/core/input_validation.py (模块 输入校验；类别 source；类型 validation；符号 validate_model_config, validate_batch_payload, validate_sample_payload, validate_seq_id)：把配额、布局一致性与 loss 输入契约集中到一处，使 core 与 executor 都能在进 batch 前拒绝坏数据；review 中 seq_id 边界问题即由 validate_seq_id 统一收口。
- miles/tinker/core/types.py (模块 核心类型；类别 source；类型 data-contract；符号 CommandOp, is_batch, requires_model_close_on_failure, LOSS_INPUT_KEYS)：全部模块共享的契约：CommandOp 的 is_batch/requires_model_close_on_failure 直接决定失败策略，GatewayConfig 承载配额与租约参数，ModelRecord 的幂等表支撑重试语义。
- miles/tinker/core/future.py (模块 请求结果；类别 source；类型 core-logic；符号 RequestFuture, RequestFutureStore, create, resolve)：请求结果与重试语义的载体：RequestFutureStore 提供 resolve/fail 的“首次结算生效”语义与 TTL 过期后的终态失败，避免重试重放后端工作。
- miles/tinker/core/utils.py (模块 检查点工具；类别 source；类型 checkpoint；符号 parse_tinker_path, resolve_checkpoint_dir, build_checkpoint_metadata, read_checkpoint_metadata)：checkpoint 路径与归属的唯一入口：tinker:// 解析、路径逃逸防护、META.json 读取与 tenant_digest 比对，决定跨租户 checkpoint 能否被加载或采样。
- miles/tinker/server/encoding.py (模块 JSON 编解码；类别 source；类型 data-contract；符号 validate_against_sdk, validate_create_model, decode_command, _decode_command)：JSON wire 契约的实现：用 pin 住的 SDK pydantic 模型做校验、把解码错误降级成携带 validation_error 的 payload、并统一 datum 构造与结果渲染。

- miles/tinker/server/proto_codec.py (模块 二进制协议; 类别 source; 类型 data-contract; 符号 maybe_decompress, decode_forward_backward_request, _decode_forward_backward, _decode_tensor) : protobuf 边界的实现: zstd 解压、ForwardBackwardRequest 解析、loss_fn_config_v2 优先策略与 dense/sparse_csr 张量解码, 直接决定 SDK 二进制路径能否被兼容。
- miles/tinker/server/app.py (模块 HTTP 接口; 类别 source; 类型 entrypoint; 符号 _tenant, build_app, _user_error, _ownership_error) : 网关对外入口: 路由表、_tenant 鉴权、400/403 错误映射、retrieve_future 长轮询与 protobuf 渲染分支都在此会合。
- miles/tinker/__init__.py (模块 包骨架; 类别 source; 类型 package-scaffold) : 新增包骨架, 使 miles.tinker 成为可导入的命名空间 (无逻辑)。
- miles/tinker/core/__init__.py (模块 包骨架; 类别 source; 类型 package-scaffold) : 新增包骨架, 承载 service/scheduler/model_queue/future/types/input_validation/utils (无逻辑)。
- miles/tinker/server/__init__.py (模块 包骨架; 类别 source; 类型 package-scaffold) : 新增包骨架, 承载 app/encoding/proto_codec (无逻辑)。

关键符号: TinkerService.run, TinkerService.create_model, TinkerService._run_create_model, TinkerService._close_model, TinkerService.submit, TinkerService._run_batch, TinkerService._run_barrier, TinkerService._dispatch_barrier_op, TinkerService._observe_background_task, RequestScheduler.schedule_next, RequestScheduler._pack_batch, RequestScheduler._merge_barriers, ModelRequestQueue._enqueue_contiguous_commands, ModelRequestQueue.open_batch_run, PendingRequest.record_output, RequestFutureStore.request_id_for_retry, validate_batch_payload, validate_model_config, validate_seq_id, build_datum, decode_command, decode_forward_backward_request, _decode_tensor, resolve_sampler_checkpoint

关键源码片段

miles/tinker/core/scheduler.py

调度核心: DatumRef/BatchUnit/BarrierUnit 与 RequestScheduler.schedule_next 定义“先做谁、哪些可以合批、哪些 barrier 可以合并”, 是跨模型跨租户利用率与顺序正确性的分界点。

```
@dataclass
class BatchUnit:
    """One forward pass on the trainer: datums packed from any number of requests."""

    op: CommandOp # FORWARD_BACKWARD | FORWARD_ONLY
    loss_fn: str | None
    loss_fn_config: dict | None
    datums: list[DatumRef] # 每条指向 request.datums[local_index], 输出按同一顺序写回

@dataclass
class BarrierUnit:
```

```
"""One barrier call after all preceding batch requests in its model queue finish."""
```

```
op: CommandOp # OPTIM_STEP | SAVE_STATE | LOAD_STATE | SAVE_WEIGHTS_FOR_SAMPLER
```

```
entries: list[tuple[ModelRequestQueue, PendingRequest]]
```

```
class RequestScheduler:
```

```
def schedule_next(self) -> BatchUnit | BarrierUnit | None:
```

```
    """Arrival order selects the seed; compatible work may overtake intervening requests."""
```

```
    datums = self._ready_datums()
```

```
    barriers = self._ready_barriers()
```

```
    # 全局 arrival 决定谁先出发: batch 与 barrier 之间也按 arrival 比较
```

```
    datum_seed = min(datums, key=lambda ref: ref.arrival) if datums else None
```

```
    barrier_seed = min(barriers, key=lambda e: e[1].command.arrival) if barriers else None
```

```
    if datum_seed is None and barrier_seed is None:
```

```
        return None
```

```
    if barrier_seed is not None and (datum_seed is None or barrier_seed[1].command.arrival < datum_seed.arrival):
```

```
        return self._merge_barriers(barrier_seed, barriers)
```

```
    return self._pack_batch(datum_seed, datums)
```

```
def _pack_batch(self, seed: DatumRef, datums: list[DatumRef]) -> BatchUnit:
```

```
    pack_key = seed.request.pack_key() # (op, loss_fn, sorted(loss_fn_config)) 等价类
```

```
    compatible = sorted(
```

```
        (ref for ref in datums if ref.request.pack_key() == pack_key),
```

```
        key=lambda ref: (ref.arrival, ref.local_index),
```

```
    )
```

```
    picked: list[DatumRef] = []
```

```
    tokens = 0
```

```
    for ref in compatible:
```

```
        datum_tokens = len(ref.datum["tokens"])
```

```
        # 只受 token 预算约束: 跨模型、跨租户的 datum 可以进同一个 BatchUnit
```

```
        if picked and tokens + datum_tokens > self.batch_token_budget:
```

```
            break
```

```
        picked.append(ref)
```

```
        tokens += datum_tokens
```

```
    for ref in picked:
```

```
        ref.request.num_issued_datums += 1 # 记账放在打包时, 重试不会重复发 datum
```

```
    command = seed.request.command
```

```
    return BatchUnit(
```

```
        op=command.op,
```

```
        loss_fn=command.payload.get("loss_fn"),
```

```
        loss_fn_config=command.payload.get("loss_fn_config"),
```

```
        datums=picked,
```

```
    )
```

```
def _merge_barriers(
```

```
    self,
```

```

seed: tuple[ModelRequestQueue, PendingRequest],
barriers: list[tuple[ModelRequestQueue, PendingRequest]],
) -> BarrierUnit:
    op = seed[1].command.op
    if op == CommandOp.OPTIM_STEP:
        # 不同模型的 optim barrier 合并成一次 trainer 调用
        entries = [(model_queue, barrier) for model_queue, barrier in barriers if barrier.
                    command.op == op]
    else:
        entries = [seed] # save/load/save_weights_for_sampler 等 barrier 仍单条执行
    return BarrierUnit(op=op, entries=entries)

```

miles/tinker/server/proto_codec.py

protobuf 边界的实现: zstd 解压、ForwardBackwardRequest 解析、loss_fn_config_v2 优先策略与 dense/sparse_csr 张量解码, 直接决定 SDK 二进制路径能否被兼容。

```

def _decode_forward_backward(message) -> dict:
    """ForwardBackwardRequest proto -> 内部 payload (含 datum 归一化) 。”
    datums = []
    for index, datum in enumerate(message.data):
        tokens: list[int] = []
        for chunk in datum.model_input:
            if chunk.WhichOneof("chunk") != "encoded_text":
                raise UserInputError(f"unsupported model_input chunk type: {chunk.
                    WhichOneof('chunk')}")
            tokens.extend(np.frombuffer(chunk.encoded_text.tokens, dtype=np.int32).tolist())
        inputs = {name: _decode_tensor(name, tensor) for name, tensor in datum.loss_fn_inputs.
            items()}
        datums.append(build_datum(tokens, inputs, index))

    loss_fn_config = dict(message.loss_fn_config)
    # Tinker SDK 的 v2 protobuf config 同时支持 number 与 text 取值:
    # 数值会双写到两张表, 字符串只写 v2, 因此 v2 存在时必须以它为准,
    # 否则像 dro(beta) 之外的字符串设置会被静默丢弃
    if message.loss_fn_config_v2:
        loss_fn_config = {}
        for name, value in message.loss_fn_config_v2.items():
            kind = value.WhichOneof("value")
            if kind is None:
                raise UserInputError(f"loss_fn_config[{name!r}]: missing number or text value")
            loss_fn_config[name] = getattr(value, kind)

    return {
        "model_id": message.model_id,
        "seq_id": message.seq_id,
        "datums": datums,
        "loss_fn": message.loss_fn,
        "loss_fn_config": loss_fn_config,
    }

```

评论区精华

8 条 review 意见全部来自 nanjiangwill，主题高度集中：不是代码风格，而是“外部 SDK 会重试，所以服务端必须自己提供幂等键”。作者在后续版本中把幂等键改为 SDK 请求体自带的 (tenant, session_id, model_seq_id) 与 (sampling_session_id, seq_id)，并引入 validate_seq_id 统一拒绝非法 seq_id；body 里也补充了“creation/training/sampling 的重试若结果已过期会拿到稳定终态失败，而不是重放后端工作”。本 PR 已获 yushengsu-thu approve，但 8 条意见中有若干是“Non-blocking”，说明作者按建议做了结构性修正而非逐条打补丁。

- create_model 必须按 (tenant, session_id, model_seq_id) 幂等 (correctness): 最终 head 先 _session_for(tenant, session_id) 校验父会话，validate_seq_id(model_seq_id) 后用 session.models_by_seq 缓存 (request_id, model_id) 并复用，重试不再重复占槽或重复 load_slot；结果过期时由 request_id_for_retry 生成终态失败。
- load_slot 失败后不能直接把 slot 放回 free_slots (correctness): 最终实现用 record.slot_initialized 标记“后端已触碰该 slot”，_close_model 里只有 unload_slot 成功才 free_slots.add，失败则记日志并保留占用 (quarantine)；trainer 已死时直接 return，不再归还。
- create_sampling_session 的绑定信息与创建去重 (correctness): head 版把租户、session_id、model_path 都放进 SamplingSessionRecord，并在 SessionRecord 里用 sampling_sessions_by_seq 做去重；base_model 兼容性下沉到 resolve_sampler_checkpoint，在采样时按 checkpoint 元数据校验。
- 采样请求的 seq_id 丢失导致重试重复生成 (correctness): head 版在 decode_sample_request 里保留 seq_id，SamplingSessionRecord.samples_by_seq 以 (sampling_session_id, seq_id) 保留 (request_id, sequence_ids)，重试返回已保留的结果，过期后给终态失败。
- seq_id <= 0 或非整数会让 future 永远 pending (correctness): head 版引入 validate_seq_id(value, name, minimum=1)，在 create_model 与 submit 的入口统一拒绝，并在注释里写明“请求 seq_id 从 1 开始，幂等键从 0 开始”的差异。
- 优化器清理失败时的 poison 状态管理 (correctness): head 版已经不存在 _poisoned_slots —— 规划层被重写为 model_queue + scheduler 架构，同类问题由“barrier 执行失败即结束该模型的流并由 _close_model 隔离 slot”覆盖。
- barrier 结果集必须先整体校验再结算 (correctness): head 版把结果产出与结算分离：_dispatch_barrier_op 先返回与 entries 一一对应的 outcomes 列表（单条异常在内部被收敛为 user 错误），_run_barrier 再统一 _finish_request；_close_model 与 remove_model_queue 也保持幂等。
- protobuf loss_fn_config 的 v2 兼容与优先级 (correctness): head 版的 _decode_forward_backward 在 message.loss_fn_config_v2 非空时重建整个字典，用 WhichOneof("value") 取出 number 或 text，缺失取值则显式报错；作者在 body 中补充了 SDK 0.26.2 对 legacy/v2 优先级、零值与空字符串的探针验证。

风险与影响

- 风险：1) 无随行测试：13 个新增文件对应 0 个测试文件，幂等、租约回收、barrier 合并这些语义在合入时没有回归网；作者把验证放到 #2849，等于让协议层先裸奔合入。2) 跨租户打包的失败放大：_pack_batch 只按 (op, loss_fn, loss_fn_config) 判兼容，跨模型跨租户共用一次 forward，_fail_batch 会把该 BatchUnit 里所有请求一起判失败（body 把跨租户打包写成设计目标，但隔离性只在时延与失败域上被弱化）。3) 进程内状态无持久化：futures、sessions、leases、幂等表、未保存的梯度累积都在进程内，future 结果 1 小时后由 request_id_for_retry 转为终态失败（core/future.py 的 _FINISHED_TTL_S），SDK 的长重试窗口可能踩到这条线。4) SDK 版本耦合：encoding.py 在 import 时物化 ADAM_PARAM_DEFAULTS = tinker_types.AdamParams().model_dump(), proto_codec.py 依赖 tinker_public_pb2 的字段布局，升 tinker 版本会直接改变服务端接受的契约。5) 路径与鉴权：checkpoint 归属靠 tenant_digest 而不是签名，API Key 明文即租户标识；resolve_checkpoint_dir 的逃逸防护除了 segment 白名单还依赖 assert，在 python -O 下会被剥离（白名单仍在，风险有限）。6) 长轮询带来的连接占用：retrieve_future 允许挂起 30 秒，高并发 client 会积累大量未完成请求。7) 后端契约违反是 fail-fast：_run_batch 的 assert len(outputs) == len(refs) 与 trainer_dead() 检查会直接终止 service.run()，这是 body 声明的设计（共享 trainer cell 失败即退出），但对上层运维意味着需要外部重启与连接拒绝策略。
- 影响：对用户：Tinker SDK 客户端可以不改代码指向该网关，获得会话 / 心跳、模型创建、训练指令、save/load、save_weights_for_sampler 与采样等全部端点，并支持 JSON 与 protobuf 两种传输；但 ttl_seconds、user_metadata、lora_config.seed、从训练 checkpoint 采样等能力被显式拒绝，需要在文档与错误信息里对齐预期。对系统：这是 multi-lora v2 的协议面，把“谁先训练、哪些 datum 能合批、失败后谁被终止”这些决策从 runtime 里解耦出来，#2845 直接在其上接 ExecutorBackend 与 serve 入口，任何契约变化都会波及 6/7 与 7/7。对团队：本 PR 事实上定义了网关模块的核心不变量（幂等键来自 SDK、slot 只有 unload 成功才可复用、checkpoint 归属靠 tenant_digest、失败策略按 op 分级），后续所有改动都要围绕这些不变量评审。
- 风险标记：无随行测试覆盖，进程内状态无持久化，跨租户打包失败放大，SDK 版本契约耦合，长轮询占连接，路径防护含 assert

关联脉络

- PR #2845 [multi-lora v2 6/7] Tinker gateway runtime and entry point: 直接后继：本 PR body 明说 “These protocol layers do not import miles.ray or miles.backends; runtime wiring follows in #2845”，runtime.py 与 serve_tinker.py 会实现这里定义的 ExecutorBackend 契约（load_slot/unload_slot/forward_backward/forward_only/optim_step/checkpoint 等）。
- PR #2849 [multi-lora v2 7/7] Tinker gateway example and tests: 本 PR 把测试全部留给了 7/7: tests/fast/tinker/core/test_service.py 与 tests/e2e/lora/tinker_gateway.py 覆盖排序、去重、batch/barrier 规划、失败隔离、checkpoint 归属、JSON/protobuf 契约与 HTTP 错误，正是本 PR 未随行的那部分验证；其 body 还提到复用 core/utils.py 做 checkpoint 断言。
- PR #3219 fix: align MTP loss masks with input tokens before context-parallel slicing: 概念关联：两者都在把“loss 输入必须与 token 位置严格对齐”固化为契约；本 PR 用

target_len 与 LOSS_FN_INPUTS 长度校验在网关切面拒绝错位数据，#3219 则在训练侧统一 input-aligned 掩码契约，属于同一类数据契约治理。