# SCML 2025 供应链管理联赛高级代理架构蓝皮书：HRL-X 混合残差学习系统的设计与实施

## 1. 执行摘要与战略背景：从启发式规则到认知型架构的范式跃迁

供应链管理联赛（Supply Chain Management League, SCML）作为国际自动协商代理竞赛（ANAC）中最具挑战性的赛道之一，长期以来被视为检验多智能体系统（Multi-Agent Systems, MAS）在复杂经济环境中适应能力的试金石。随着 2025 年标准联赛（Standard League）规则的重大迭代，竞赛的本质发生了根本性的转变。特别是由于非易腐库存（Non-Perishable Inventory）机制的引入、期货合约（Future Contracts）交易的开放以及更为严苛的短缺惩罚（Shortfall Penalties）体系的确立 2，SCML 已不再仅仅是一个考察单次博弈谈判技巧的竞技场，而是演变成了一个要求代理具备长周期库存规划、跨期风险对冲以及多线程并发资源协调能力的综合性企业资源规划（ERP）模拟环境。

回顾 2024 年及之前的赛果，以 PenguinAgent 和 AS0 为代表的基于规则的启发式代理（Heuristic Agents）凭借其稳健的成本控制逻辑和确定性的风险规避策略，占据了排行榜的前列 2。然而，这种“硬编码”的智慧在面对 2025 年日益复杂的动态市场时显露出了明显的局限性。具体而言，启发式代理缺乏对非平稳市场环境的自适应能力，难以利用长尾分布中的投机机会（如跨期库存套利），且无法有效处理由期货交易带来的状态空间指数级膨胀问题。为了突破这一策略天花板，学术界与工业界开始将目光投向更具认知深度的强化学习（Reinforcement Learning, RL）架构。

本研究报告深入探讨了针对 SCML 2025 环境定制的“混合残差学习架构”（Hybrid Residual Learner, HRL-X）。该架构并非单一算法的简单堆砌，而是一个有机的控制论系统，它通过 **分层强化学习（HRL）** 解决时间尺度的抽象与长期规划问题，利用 **决策 Transformer（Decision Transformer, DT）** 处理博弈过程中的序列依赖与对手建模，并借助 **离线强化学习（Offline RL）** 中的 "Reward-on-the-Line" (ROL) 机制解决高风险环境下的探索安全与样本效率问题 2。

本报告将提供一份详尽无遗的技术蓝图，旨在为 LitaAgent-HRL 的开发提供代码级的实施方案。不同于常规的概要设计，本文档将深入 TensorFlow/PyTorch 的具体实现层面，详细阐述 HRL-X 架构中每一个组件的数学原理、张量流向、工程实现细节以及完整的训练路线图。我们将通过严谨的数学建模，剖析 SCML 2025 环境下的部分可观察马尔可夫决策过程（POMDP）特性，并论证 HRL-X 如何通过分层控制、残差学习和全局注意力机制，系统性地解决库存耦合、并发协商和探索风险这三大核心难题。

## 2. 环境数学建模：高维 POMDP 下的深度耦合博弈

在着手具体的代码实现之前，必须透彻理解 LitaAgent-HRL 所处的数学环境。SCML 2025 本质上是一个高维、随机、部分可观察的马尔可夫博弈，我们需要将其形式化为一个 POMDP 元组 $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \Omega, \mathcal{O}, \gamma \rangle$。

### 2.1 状态空间的非完全可观察性与信念重构

全局状态 $\mathcal{S}$ 包含了上帝视角下的所有市场信息：每一个竞争对手 $i \in \{1, \dots, N\}$ 的私有库存水平 $I\_t^i$、资金状况 $B\_t^i$、生产线闲置率、隐藏的生产成本函数 $C^i(\cdot)$，以及未来 $T$ 天的真实市场外生需求曲线 $D(p, t)$。显然，对于单一代理 LitaAgent-HRL 而言，这些关键信息是完全屏蔽的 4。代理所能依赖的仅是极其有限的观测空间 $\Omega$。这包括自身的私有状态（如当前库存 $I\_t$、资金余额 $B\_t$）以及部分带有噪声或滞后性的公共市场状态（如公告板上的交易价格指数 $\mathcal{M}\_t$）4。

这种严重的信息不对称导致了环境的 **非平稳性（Non-Stationarity）**。从 LitaAgent-HRL 的局部视角看，环境的转移概率 $\mathcal{T}(s'|s, a)$ 似乎随时间动态变化，因为对手的策略 $\pi^{-i}$ 在不断演变，而这些变化无法被直接观测。因此，我们的架构必须具备 **信念状态重构（Belief State Reconstruction）** 的能力。这就决定了我们在 L3 层必须采用具备序列建模能力的架构（如 Decision Transformer），即通过观测到的历史交互序列 $\mathcal{H}\_t$（包括对手的历史出价、响应时间、让步幅度），推断出隐藏状态 $z\_t$（如对手的急迫程度或底线价格）1。

### 2.2 2025 规则引发的“三重耦合难题”

LitaAgent-HRL 的核心设计逻辑旨在解决由 2025 年新规则引发的三重深度耦合问题。理解这些耦合是理解 HRL-X 分层设计的关键。

#### 2.2.1 库存-时间的跨期耦合 (Inventory-Time Coupling)

非易腐库存规则的引入彻底改变了库存管理的逻辑。在旧规则下，库存每日清零，代理只需关注当天的供需平衡（Just-In-Time）。而在新规则下，库存可以累积，这引入了 **跨期套利（Inter-temporal Arbitrage）** 的可能性。代理可以在原材料价格低廉时（$t$ 时刻）大量囤积，承担持有成本，待市场价格回升后（$t+H$ 时刻）生产并出售成品。这种决策具有极长的因果链条：今日的买入动作可能旨在满足十天后的市场需求。这意味着奖励信号在时间维度上极度稀疏且延迟，传统的扁平化 RL 难以处理这种长达数百步的信用分配（Credit Assignment）问题。因此，必须引入分层强化学习（HRL），利用高层管理者（L2）进行跨天规划 4。

#### 2.2.2 并发-资源的竞争耦合 (Concurrency-Resource Coupling)

SCML 2025 强调并发协商。代理必须同时在多个线程中与上游供应商和下游采购商进行谈判。这些看似独立的谈判线程实际上通过共享的资源池（有限的库存空间、有限的现金流、有限的产能）紧密耦合。

* **资金耦合**：如果在上游过度激进地锁定原材料，可能导致现金流枯竭，无法支付下游交易的违约金或生产成本。
* 库存耦合：上游买入的原料如果没有下游订单消化，将导致爆仓，产生高额仓储费。  
  单一线程的最优解（例如以极低价格买入大量原料）如果缺乏全局视角的配合，可能对整体效用是灾难性的。这要求架构中必须包含一个 全局协调器（L4 Global Coordinator），利用注意力机制在不同线程间动态分配权重 1。

#### 2.2.3 风险-探索的生存耦合 (Risk-Exploration Coupling)

取消现货市场并引入严厉的短缺惩罚，使得环境具有极端的风险不对称性。对于 RL 代理而言，这意味着 **探索（Exploration）** 的代价极其高昂。在训练初期，基于 $\epsilon$-greedy 或高熵策略的随机探索极易触发连续违约，导致代理破产并提前终止回合。这种“死亡陷阱”使得代理难以收集到成功的正样本，容易陷入“不敢交易”的次优局部极小值。因此，必须引入 **安全护盾（L1 Safety Shield）**，通过确定性规则强制约束探索边界 5。

## 3. HRL-X 总体架构拓扑与数据流设计

针对上述数学模型与挑战，HRL-X 架构被设计为一个集成了规则控制、分层规划、序列建模与多智能体协调的四层混合控制系统。本章概述其拓扑结构与数据流向。

### 3.1 架构层级定义

| **层级** | **组件名称** | **核心技术** | **时间尺度** | **核心职责** |
| --- | --- | --- | --- | --- |
| **L1** | **安全护盾 (Safety Shield)** | 启发式规则 (Penguin Logic) | 实时 (Real-time) | 生成硬性动作掩码 $\mathcal{M}\_{safe}$，提供基准动作 $a\_{base}$，防止破产与违约。这是系统的“宪法”。 |
| **L2** | **战略管理者 (Strategic Manager)** | PPO (Proximal Policy Optimization) | 天 (Daily) | 基于宏观状态设定每日战略目标向量 $g\_t$（买卖量与限价）。负责跨期规划与库存势能管理。 |
| **L3** | **残差执行者 (Residual Actor)** | 决策 Transformer (Decision Transformer) | 轮 (Round) | 基于微观协商历史与目标 $g\_t$，输出相对于基准的动作残差 $\Delta a$。负责具体的谈判博弈。 |
| **L4** | **全局协调器 (Global Coordinator)** | 注意力机制 (Centralized Attention) | 轮 (Round) | 处理并发谈判，计算各线程的重要性权重 $\alpha\_k$，调节 L3 的激进程度，解决资源争夺。 |

### 3.2 张量计算图与数据流

在每一决策步 $t$，数据流向如下：

1. **观测解析**：原始观测 $o\_t$ 被拆解。宏观特征（总库存、资金、天数、未来合约列表）流向 L2；微观特征（各线程的报价序列、剩余时间）流向 L3。
2. **L1 安全计算**：L1 根据当前 $I\_t, B\_t$ 及已签合约，计算物理约束。输出动作掩码张量 mask\_tensor 和基准动作张量 base\_action。此过程不涉及梯度计算。
3. **L2 战略规划**（仅在每日开始时激活）：L2 接收宏观特征，通过 MLP 网络输出高斯分布参数，采样得到当天的目标向量 goal\_vector（如：今日需买入 50 单位，最高价 100）。此向量被广播（Broadcast）给所有 L3 实例。
4. **L3 微观编码**：每个活跃线程的 L3 决策 Transformer 接收各自的历史序列 history\_seq 和 goal\_vector。Transformer 输出当前线程的隐状态 hidden\_state。
5. **L4 全局协调**：L4 接收所有线程的 hidden\_state 集合。通过多头自注意力（Multi-Head Self-Attention）计算各线程的注意力权重 attention\_weights。
6. 动作合成：L3 结合 hidden\_state 和 L4 反馈的 attention\_weights，输出残差动作 delta\_action。最终动作合成公式为：  
     
   $$a\_{final} = \text{Clip}(a\_{base} + \Delta a, \mathcal{M}\_{safe})$$  
     
   此动作被发送至 SCML 模拟器执行。

## 4. L1 安全基底层：确定性逻辑与代码实现

L1 安全护盾是 HRL-X 的基石。它不涉及梯度更新，而是通过硬编码的领域知识确保代理在任何时候都不会采取自杀式行动。我们借鉴了 2024 年冠军 PenguinAgent 的核心逻辑 5，并将其封装为可嵌入计算图的张量操作。

### 4.1 核心逻辑：Penguin 范式的数学表达

安全护盾的核心任务是计算 **可行且安全** 的动作边界。

约束一：最大安全买入量 ($Q\_{max\\_buy}$)

为了防止爆仓（导致仓储费剧增），买入量必须受限于剩余库容。考虑到已签署但未交付的入库量 $I\_{incoming}$ 和已承诺发货的出库量 $O\_{committed}$：

$$Q\_{max\\_buy} = C\_{total} - I\_{current} - I\_{incoming} + O\_{committed}$$

任何 $q > Q\_{max\\_buy}$ 的买入 Offer 都是非法的 2。

约束二：最小必要买入量 ($Q\_{min\\_buy}$)

为了防止违约（Shortfall Penalty），必须确保有足够的原料来生产已承诺的订单：

$$Q\_{min\\_buy} = \max(0, O\_{committed} - I\_{current} - I\_{incoming})$$

这是满足当前合约的底线 2。

约束三：破产保护价格 ($P\_{limit}$)

为了防止现金流断裂，总支出不能超过当前可用资金减去保留金（Reserve）：

$$P\_{limit}(q) = \frac{B\_t - \text{Reserve}}{q}$$

基准动作 ($a\_{base}$)

L1 不仅提供约束，还提供一个“及格”的动作建议。基于 PenguinAgent 的保守策略，基准动作通常是：买入量 $q = Q\_{min\\_buy}$，价格 $p = \text{Cost}\_{production} \times (1 + \text{Margin}\_{min})$。这为残差学习提供了一个极其稳健的起点 5。

### 4.2 TensorFlow 自定义层实现方案

我们将 L1 封装为一个标准的 Keras Layer。注意，虽然其内部逻辑不可微，但它可以作为张量预处理步骤，确保输入 L3 的数据已经包含了明确的边界信息。

Python

import tensorflow as tf  
  
class SafetyMaskingLayer(tf.keras.layers.Layer):  
 """  
 L1: Safety Shield Layer.  
 负责计算动作掩码与基准动作，不含可训练参数。  
 实现了 PenguinAgent 的核心库存与资金约束逻辑。  
 """  
 def \_\_init\_\_(self, warehouse\_capacity, bankruptcy\_reserve=1000.0, \*\*kwargs):  
 super(SafetyMaskingLayer, self).\_\_init\_\_(\*\*kwargs)  
 self.capacity = tf.constant(warehouse\_capacity, dtype=tf.float32)  
 self.reserve = tf.constant(bankruptcy\_reserve, dtype=tf.float32)  
  
 def call(self, inputs):  
 # inputs shape: (batch\_size, 5)  
 # [current\_inventory, wallet\_balance, incoming\_committed, outgoing\_committed, market\_price]  
 inventory = inputs[:, 0]  
 wallet = inputs[:, 1]  
 incoming = inputs[:, 2]  
 outgoing = inputs[:, 3]  
 market\_price = inputs[:, 4]  
  
 # --- 1. 计算约束边界 ---  
   
 # 预测库存峰值：当前 + 在途 - 拟出库  
 # 注意：这里做保守估计，假设出库可能失败，故计算最大买入时不扣减 outgoing 可能更安全，  
 # 但 Penguin 逻辑允许扣减以最大化流转。  
 projected\_inventory = inventory + incoming - outgoing  
   
 # Max Buy: 剩余空间  
 # 使用 ReLU 确保非负  
 max\_buy\_qty = tf.nn.relu(self.capacity - projected\_inventory)  
   
 # Min Buy: 缺口 (Shortfall prevention)  
 min\_buy\_qty = tf.nn.relu(outgoing - (inventory + incoming))  
  
 # Max Spend Price per unit (simplified for tensor op)  
 # 实际操作中价格上限通常依赖于数量，这里计算一个针对单单位的理论上限  
 max\_price\_total = tf.nn.relu(wallet - self.reserve)  
  
 # --- 2. 生成基准动作 (Baseline Action) ---  
   
 # 策略：只买必须的 (Min Buy)，如果无需买则为 0。这是 Penguin 的 JIT 核心。  
 base\_buy\_qty = min\_buy\_qty  
   
 # 策略：基准价格设为市场价。  
 base\_buy\_price = market\_price  
   
 # 组合基准动作向量 [qty, price]  
 baseline\_action = tf.stack([base\_buy\_qty, base\_buy\_price], axis=1)  
  
 # --- 3. 生成掩码边界 (Bounds) ---  
 # 返回 [min\_q, max\_q, max\_total\_spend]  
 bounds = tf.stack([min\_buy\_qty, max\_buy\_qty, max\_price\_total], axis=1)  
  
 return baseline\_action, bounds  
  
 def compute\_output\_shape(self, input\_shape):  
 return [(input\_shape, 2), (input\_shape, 3)]

**实现洞察**：

* 使用 tf.nn.relu 替代 tf.maximum(0, x) 以保持计算图的简洁性。
* bounds 张量并不直接作为 Softmax 的 mask，而是作为后续 L3 输出层 tf.clip\_by\_value 的参数。这是一种针对连续动作空间的 **软掩码机制 (Soft Masking)**，不同于离散动作空间的 Logit Masking。这种设计允许残差网络在安全边界内自由探索，但物理上阻止越界 4。

## 5. L2 战略管理者：分层 PPO 代理设计与实现

L2 管理者是 HRL-X 的大脑，负责跨天的时间抽象（Temporal Abstraction）。它解决的是“今天该采取何种库存姿态”的问题，而不关心具体的谈判话术。

### 5.1 状态与动作空间定义

管理者状态空间 $\mathcal{S}\_{mgr}$：

我们需要构建能够反映宏观供需和长期趋势的特征向量：

* **库存势能特征**：$\Phi(s) = I\_{total} \times P\_{avg}$。反映当前持仓的总市值。
* **资金健康度**：$B\_t / B\_{initial}$。
* **期货承诺向量**：这是一个关键设计。它是一个长度为 $H$（如 10 天）的向量，表示未来每天的净合约量。这需要通过 1D 卷积层（Conv1D）进行特征提取，以感知未来的供需峰谷（如“三天后有一大批原料到货”）4。
* **市场趋势**：过去 10 天的市场均价与成交量。

管理者动作空间（目标向量 $g\_t$）：

管理者输出的是一个指导性的目标向量，而非直接的控制指令：

$$g\_t = [Q\_{target\\_buy}, P\_{limit\\_buy}, Q\_{target\\_sell}, P\_{limit\\_sell}]$$

* $Q\_{target\\_buy}$：今日计划总买入量。
* $P\_{limit\\_buy}$：买入最高限价（超出此价 L3 将极其谨慎）。  
  这些目标通过 $R\_{intrinsic}$（内在奖励）与 L3 的行为挂钩。

### 5.2 PPO 算法适用性与代码实现

对于 L2，我们选择 **PPO (Proximal Policy Optimization)** 算法。原因在于管理者的决策频率低（每天一次），样本相对稀缺，PPO 的 Clip 机制能防止策略剧烈震荡，且天然支持连续的高斯分布输出。

以下是使用 TensorFlow 构建 PPO Actor-Critic 网络的实现方案。特别注意 **Conv1D** 的使用，这是处理期货合约时间序列的关键。

Python

import tensorflow as tf  
import tensorflow\_probability as tfp  
  
class ManagerPPOAgent(tf.keras.Model):  
 def \_\_init\_\_(self, state\_dim, action\_dim=4, clip\_ratio=0.2):  
 super(ManagerPPOAgent, self).\_\_init\_\_()  
 self.clip\_ratio = clip\_ratio  
 self.action\_dim = action\_dim  
   
 # --- 特征提取层 ---  
 # 专门处理期货向量的卷积层：感知未来的供需波峰  
 # filters=16, kernel\_size=3 意味着它能捕捉 3 天内的局部供需模式  
 self.future\_conv = tf.keras.layers.Conv1D(filters=16, kernel\_size=3, activation='relu')  
 self.flatten = tf.keras.layers.Flatten()  
   
 # 共享特征层  
 self.common\_dense = tf.keras.layers.Dense(128, activation='relu')  
   
 # --- Actor 网络 (策略) ---  
 self.actor\_dense = tf.keras.layers.Dense(64, activation='relu')  
 # 输出均值和对数标准差  
 self.actor\_out = tf.keras.layers.Dense(action\_dim \* 2)   
   
 # --- Critic 网络 (价值) ---  
 self.critic\_dense = tf.keras.layers.Dense(64, activation='relu')  
 self.critic\_out = tf.keras.layers.Dense(1)  
  
 def call(self, inputs):  
 # inputs: {'scalars': (batch, n), 'futures': (batch, H, 1)}  
 scalars = inputs['scalars']  
 futures = inputs['futures']  
   
 # 处理期货序列特征  
 fut\_feat = self.future\_conv(futures)  
 fut\_feat = self.flatten(fut\_feat)  
   
 # 拼接标量特征与时序特征  
 concat\_state = tf.concat([scalars, fut\_feat], axis=-1)  
 x = self.common\_dense(concat\_state)  
   
 # Actor 前向传播  
 a\_x = self.actor\_dense(x)  
 actor\_params = self.actor\_out(a\_x)  
 mean, log\_std = tf.split(actor\_params, 2, axis=-1)  
   
 # 限制 log\_std 防止数值不稳定 (Clip log\_std)  
 # 这是一个关键的工程技巧，防止标准差爆炸导致 NaN  
 log\_std = tf.clip\_by\_value(log\_std, -20, 2)  
 std = tf.exp(log\_std)  
   
 # 构建高斯分布  
 dist = tfp.distributions.Normal(mean, std)  
 action = dist.sample()  
   
 # Critic 前向传播  
 c\_x = self.critic\_dense(x)  
 value = self.critic\_out(c\_x)  
   
 # 返回动作，对数概率（用于 PPO Loss 计算），以及状态价值  
 return action, dist.log\_prob(action), value  
  
 def compute\_loss(self, old\_log\_probs, returns, advantages, inputs, actions):  
 """  
 PPO Loss Function Implementation  
 """  
 new\_actions, new\_log\_probs, values = self.call(inputs)  
   
 # 1. Critic Loss (MSE)  
 critic\_loss = tf.reduce\_mean(tf.square(returns - values))  
   
 # 2. Actor Loss (Clipped Surrogate Objective)  
 ratio = tf.exp(new\_log\_probs - old\_log\_probs)  
 surr1 = ratio \* advantages  
 surr2 = tf.clip\_by\_value(ratio, 1.0 - self.clip\_ratio, 1.0 + self.clip\_ratio) \* advantages  
 actor\_loss = -tf.reduce\_mean(tf.minimum(surr1, surr2))  
   
 # 3. Entropy Bonus (Exploration)  
 entropy\_loss = -tf.reduce\_mean(new\_log\_probs) # 近似熵  
   
 total\_loss = actor\_loss + 0.5 \* critic\_loss - 0.01 \* entropy\_loss  
 return total\_loss

## 6. L3 残差执行者：决策 Transformer 与残差学习实现

L3 是系统的战术引擎，负责在每一轮（Round）的微观博弈中与对手周旋。我们采用 **决策 Transformer (Decision Transformer, DT)** 结合 **残差学习（Residual Learning）** 的方案 1。

### 6.1 为什么要用 Decision Transformer？

SCML 的协商过程本质上是一个序列决策问题。对手的每一次出价、每一次犹豫（响应时间）、每一次让步，都构成了上下文（Context）。传统的 MLP 或 LSTM 在处理长序列依赖（Long-term Dependency）和因果推理方面能力有限。DT 通过 **自注意力机制（Self-Attention）**，能够直接关注到序列中关键的历史节点（例如对手在 Deadline 前的突然降价），从而捕捉其隐性的心理状态（Belief State），这正是解决 POMDP 问题的关键 4。

### 6.2 残差学习机制： **$A = A\_{base} + \Delta A$**

为了解决 RL 代理在初期探索时的不稳定性，L3 并不直接输出绝对动作，而是输出相对于 L1 基准动作的偏置（Residual/Delta）。

$$A\_{final} = \text{Clip}(A\_{base} + \Delta A, \text{Bounds})$$

* **训练初期**：网络权重接近随机，$\Delta A \approx 0$，代理表现等同于 PenguinAgent（及格线）。
* **训练后期**：网络学会了在何时微调价格以榨取更多利润（如 $\Delta p = +5$）或为了成交而快速让步（$\Delta p = -2$）。

### 6.3 决策 Transformer 实现代码

我们需要实现一个因果 GPT 风格的 Transformer。核心创新在于将 L2 的 Goal 注入到 Transformer 的每一个时间步中，作为条件控制。

Python

class ResidualDecisionTransformer(tf.keras.Model):  
 def \_\_init\_\_(self, d\_model=128, n\_heads=4, n\_layers=2, max\_len=20, action\_dim=2):  
 super().\_\_init\_\_()  
 self.d\_model = d\_model  
   
 # --- 嵌入层 ---  
 # 状态嵌入：将报价历史 [price, qty, time] 映射到隐空间  
 self.state\_emb = tf.keras.layers.Dense(d\_model)  
 # 目标嵌入：将 L2 Goal 映射到隐空间  
 self.goal\_emb = tf.keras.layers.Dense(d\_model)  
 # 位置编码：标准的 Transformer 位置嵌入  
 self.pos\_emb = tf.keras.layers.Embedding(max\_len, d\_model)  
 self.dropout = tf.keras.layers.Dropout(0.1)  
   
 # --- Transformer Blocks ---  
 self.blocks =  
 for \_ in range(n\_layers):  
 self.blocks.append({  
 'attn': tf.keras.layers.MultiHeadAttention(num\_heads=n\_heads, key\_dim=d\_model),  
 'ln1': tf.keras.layers.LayerNormalization(),  
 'ffn': tf.keras.Sequential(),  
 'ln2': tf.keras.layers.LayerNormalization()  
 })  
   
 # --- 残差输出头 ---  
 # 输出 [delta\_qty, delta\_price]  
 # 使用 Tanh 限制残差范围在 [-1, 1] 之间，然后通过 scaling 放大  
 self.action\_head = tf.keras.layers.Dense(action\_dim, activation='tanh')  
   
 # 可学习的缩放因子，控制残差幅度。初始化为较小值，逐步放开。  
 # 例如初始化为 ，表示数量可微调 +/- 5，价格可微调 +/- 10  
 self.residual\_scale = tf.Variable([5.0, 10.0], trainable=True, dtype=tf.float32)  
  
 def call(self, inputs, training=False):  
 # inputs: {'history': (B, T, F), 'goal': (B, G\_dim), 'baseline': (B, A\_dim)}  
 history = inputs['history']  
 goal = inputs['goal']  
 baseline = inputs['baseline']  
   
 seq\_len = tf.shape(history)[1]  
   
 # 1. 嵌入  
 x = self.state\_emb(history) # (B, T, d\_model)  
   
 # 2. 融合 Goal (将 L2 的目标注入到每一个时间步)  
 # 这是一种 Conditional Transformer 设计  
 g = self.goal\_emb(goal) # (B, d\_model)  
 g = tf.expand\_dims(g, 1) # (B, 1, d\_model)  
 x = x + g # 广播相加  
   
 # 3. 位置编码  
 positions = tf.range(start=0, limit=seq\_len, delta=1)  
 x = x + self.pos\_emb(positions)  
 x = self.dropout(x, training=training)  
   
 # 4. 因果掩码 (Causal Mask)  
 # 确保 t 时刻只能看到 0...t，防止穿越未来  
 mask = 1 - tf.linalg.band\_part(tf.ones((seq\_len, seq\_len)), -1, 0)  
   
 # 5. Transformer Pass  
 for block in self.blocks:  
 # Self Attention  
 attn\_out = block['attn'](query=x, value=x, key=x, attention\_mask=mask)  
 x = block['ln1'](x + attn\_out)  
 # FFN  
 ffn\_out = block['ffn'](x)  
 x = block['ln2'](x + ffn\_out)  
   
 # 6. 取最后一个时间步的隐状态  
 last\_hidden = x[:, -1, :] # (B, d\_model)  
   
 # 7. 计算残差  
 delta = self.action\_head(last\_hidden) \* self.residual\_scale  
   
 # 8. 合成 (这里仅计算合成前的 raw 值，Clip 在外部模型循环中做)  
 final\_raw = baseline + delta  
   
 # 返回 final\_raw 用于损失计算，返回 last\_hidden 供 L4 使用  
 return final\_raw, delta, last\_hidden

## 7. L4 全局协调器：基于注意力机制的并发调度

SCML 2025 的核心痛点在于并发协商中的资源冲突。L4 协调器不直接产生动作，而是通过调节 L3 的隐状态或输出权重来解决冲突 2。

### 7.1 "广播-过滤" 协议与集中式注意力

我们实施 **“广播-过滤”（Broadcast-Filter）** 协议。代理在每天开始时向所有潜在供应商发起请求（Broadcast），然后在协商过程中动态决定哪些交易值得成交（Filter）。L4 负责 Filter 步骤。

集中式注意力机制原理

协调器接收所有活跃线程 L3 Transformer 输出的隐状态向量集合 $\{h\_1, h\_2, \dots, h\_K\}$。它采用 多头自注意力机制（Multi-Head Self-Attention） 来计算每个线程的重要性权重 $\alpha\_k$：

$$Q = W\_q S\_{global}, \quad K\_k = W\_k h\_k$$

$$\alpha = \text{Softmax}\left(\frac{Q K^T}{\sqrt{d\_k}}\right)$$

* $S\_{global}$：全局状态（库存、资金）。
* $\alpha\_k$：线程 $k$ 的注意力权重。

**逻辑解释**：如果 $S\_{global}$ 显示原料库存告急，且线程 A 是唯一能提供该原料的供应商（其 $h\_A$ 编码了“我有货”的特征），$Q$ 与 $K\_A$ 的点积将极大，导致 $\alpha\_A \to 1$。

### 7.2 策略修正与资源倾斜实现

计算出的权重 $\alpha\_k$ 如何影响决策？我们将其作为一种 **调制信号** 反馈给 L3。在 L3 的 call 方法中，我们可以修改最后一步的残差计算逻辑：

Python

# 修改 L3 输出逻辑以接受 L4 的调制  
# alpha\_k (B, 1) 来自 L4  
# 如果 alpha\_k 很高，放大残差，允许更激进的出价 (Aggressive)  
# 如果 alpha\_k 很低，缩小残差或施加负偏置，导致出价极低 (Conservative/Stall)  
  
delta = self.action\_head(last\_hidden) \* self.residual\_scale \* (1 + alpha\_k)

或者，更直接地，L4 输出一个 **Gate 值** $\in $，如果 Gate < 0.5，则强制覆盖 L3 的动作为 Reject 或 Wait。这实际上实现了资源的动态分配。

## 8. 训练实施方案：从取证到自我博弈的四阶段课程

训练 LitaAgent-HRL 是一个系统工程，我们制定了严格的四阶段课程学习计划。

### 8.1 阶段 0：行为克隆与热启动 (Forensic Phase & Warm Start)

目标：解决 RL 冷启动问题，获得一个性能等同于 PenguinAgent 的神经网络副本。

数据源：SCML 2023-2024 冠军日志。使用 Python 脚本解析 negotiations.csv。

L2 数据重构（Hindsight Labeling）：

由于 PenguinAgent 没有显式的 Goal，我们需要通过 取证逆向演绎（Forensic Inverse-Deduction） 来生成标签。

* 如果 PenguinAgent 在某天实际上买入了 100 单位，最高价 50。
* 我们就假定它当天的 Goal 是 ``。
* 构建样本对：(State\_DayStart, Goal\_Reconstructed)。

L3 数据重构：

直接使用每一轮的 (History, Offer) 对作为样本。

**实施算法**：使用 **Behavior Cloning (BC)** 预训练 L2 和 L3 网络。

### 8.2 阶段 1：离线强化学习与 ROL 算法 (Offline RL Phase)

问题：BC 会模仿 PenguinAgent 的错误。我们需要它做得比 PenguinAgent 更好。

算法：Reward-on-the-Line (ROL) 2。

**实施细节**：

1. **集合一致性 (Ensemble Agreement)**：训练 $N=5$ 个 Decision Transformer 网络。
2. 不确定性惩罚：对于任意状态动作对 $(s, a)$，计算 $N$ 个网络 Q 值的方差。如果方差过大（说明该动作在数据集中未出现，属 OOD），则在 Loss 中施加惩罚。  
     
   $$L\_{ROL} = \| a\_{pred} - a\_{expert} \|^2 + \lambda\_{var} \text{Var}(Q\_{ensemble}(s, a))$$
3. **优势过滤**：只模仿那些最终带来正利润的谈判序列。

### 8.3 阶段 2：分层联合微调 (Hierarchical Co-training)

环境设置：SCML 2025 标准模拟器。

算法：MAPPO (Multi-Agent PPO)。

奖励函数工程 ($R\_{total}$)：

为了解决稀疏奖励，我们设计了复合奖励函数：

$$R\_t = R\_{profit} + \lambda\_1 R\_{liquidity} - \lambda\_2 R\_{risk} + \lambda\_3 R\_{intrinsic}$$

* 势能函数 ($R\_{profit}$)：引入势能函数 $\Phi(s) = I\_{inventory} \times P\_{market\\_avg}$ 来衡量库存价值。  
    
  $$R\_{shaped} = (B\_{t+1} - B\_t) + \gamma \Phi(s\_{t+1}) - \Phi(s\_t)$$  
    
  这解决了“买入即亏损（现金减少）”的短视问题，让代理学会投资 1。
* **流动性奖励 ($R\_{liquidity}$)**：只要达成交易，给予微小正奖励 $\epsilon$，防止策略冻结。
* **内在一致性奖励 ($R\_{intrinsic}$)**：惩罚 L3 执行量与 L2 目标量的偏差。

**训练循环伪代码**：

Python

def train\_online\_mappo():  
 # 1. 收集轨迹  
 trajectories = run\_episode(agent, env)  
   
 # 2. L2 更新 (Daily Step)  
 # 计算 L2 的 Returns (基于 Daily Profit + Potential)  
 l2\_loss = ppo\_update(l2\_agent, trajectories['macro'])  
   
 # 3. L3/L4 更新 (Round Step)  
 # 计算 L3 的 Returns (基于 Deal Profit + Intrinsic Reward)  
 # L4 与 L3 共享 Critic 或联合训练  
 l3\_loss = ppo\_update(l3\_agent, trajectories['micro'])  
   
 return l2\_loss, l3\_loss

### 8.4 阶段 3：全压力对抗自博弈 (Adversarial Self-Play)

目标：达到超人水平（Superhuman Performance）。

对手池 (Opponent Pool)：

* 静态对手：PenguinAgent, AS0 (基准)。
* 动态对手：历史版本的 LitaAgent-HRL (Self-Play)。

**实施**：随着训练进行，定期将当前的 LitaAgent-HRL 保存并存入对手池。代理在训练时，其对手从池中随机采样。这迫使代理不断战胜“昨天的自己”，逼近纳什均衡点，确保持续进化并防止策略循环 4。

## 9. 结论

本报告提供的 HRL-X 架构方案，通过 **L1 安全护盾** 解决了探索风险，通过 **L2 战略层** 解决了跨期库存规划，通过 **L3 残差 DT** 解决了微观序列博弈，并通过 **L4 全局协调** 解决了并发资源争夺。结合详细的 TensorFlow 代码实现与 ROL-MAPPO 混合训练流程，该方案为开发 SCML 2025 冠军级代理提供了完备的理论与工程基础。这不仅仅是对规则代理的升级，而是向具身认知与从容应对复杂经济系统迈出的关键一步。

#### 引用的著作

1. HRL-X 研究：强化学习问题解决
2. PenguinAgent 目标与谈判机制
3. HRL-X 代理实现与数据收集
4. SCML 代理开发技术架构
5. L1-L4 层设计与离线强化学习