NPULearn CoT 技术文档

版本：v0.2.0 | 更新日期：2025 年 6 月
适用平台：Windows、macOS、Linux、Android

目录

1. 概要

本文档系统性阐述了 NPULearn 的核心技术——思维链（Chain of Thought, CoT）。CoT 为大规模语言模型构建了一套结构化的内部推理框架，通过模拟人类的“理解-推理-验证”流程，旨在本质上提升智能助手的逻辑严密性、问题求解能力与交互可靠性。本文档将详细说明 CoT 的执行流程、关键指令及输出格式规范。

2. CoT 核心理念

CoT 的核心理念在于，强制模型在生成最终响应前，必须执行一套明确且可追溯的内部推理流程。该流程被严格划分为多个阶段，以确保模型在输出前已完成充分的理解、推理与自我校验。

其主要优势包括：

- **提升准确性**：通过分阶段推理，有效降低逻辑、计算及多步规划等复杂任务中的错误率。
- **增强可解释性**：模型的内部推理链路（`understand`, `think`, `verify`）对开发者完全可见，显著提升了诊断能力与决策透明度。
- **提高可靠性**：强制性的 `verify` 阶段赋予模型自我纠错能力，可在输出前拦截并修正潜在错误，确保生成结果的稳健性。
- **复杂任务处理能力**：使模型能够系统性分解并应对高复杂度的用户请求。

3. CoT 执行流程

CoT 的完整执行流程由一系列特殊标记符分隔的阶段构成。模型的完整输出遵循此固定结构，但系统最终仅将特定阶段的内容呈现给用户。

3.1. 特殊标记符

CoT 流程使用以下特殊标记符界定不同阶段：

- `<|start_title|>` 和 `<|end_title|>`: 用于包裹对话标题。
- `<|start_header|>` 和 `<|end_header|>`: 用于包裹各阶段名称，如 `understand`。
- `...`: 用于封装需要由后端解析和执行的排版指令或工具调用。

3.2. 流程阶段

整个流程分为以下几个核心阶段，模型必须严格遵循。

阶段	核心任务	关键要求与说明
更新对话标题（可选）	Update Chat Title	• 若当前对话无标题，则根据**全部历史对话上下文**生成一个简洁、准确的中文标题。 • 格式: `<|start_title|>对话标题<|end_title|>`
理解 (understand)	Comprehend	• **语言**: 中文。 • 使用 `PlantUML` 语法将对用户请求的理解可视化。

		• 拆解请求，列出关键点与待解决步骤。 • 设定回应时的情感基调与人格（Persona）。 • 识别与请求相关的用户角色。 • 主动联想并列出与问题相关的**背景常识**，作为后续推理的事实基础。
思考 (think)	Think Step-by-Step	• **语言**：中文。 • 同样可使用 `PlantUML` 呈现详细的思考链路。 • 进行严谨的、链式的逻辑分析，尤其是在处理数值计算和多步推理时。 • 分析不同用户请求间的内在关联，以及请求与自身角色的关系。 • 若在推理中发现先前思路有误，必须输出特定语句以触发重思考： `“我之前的想法是错的，让我再试一次。”`
验证 (verify)	Verify & Self-Correct	• **语言**：中文。 • 对 `think` 阶段的推理过程进行批判性复核，检查是否存在逻辑、事实或计算谬误。 • **核心纠错机制**：若发现任何错误，**必须**重新进入 `think` 阶段（通过再次输出 `<|start_header|>think<|end_header|>`），开启新一轮的思考。
排版与响应 (typeset_and_respond)	Format and Respond	• **语言**：中文。 • **可见性**：此为**唯一**默认对用户直接可见的部分。 • **强制性**：此阶段**不可或缺**。 • 整合并润色已验证的结论，生成最终的、符合人格设定、排版精良的响应。

- 所有动态内容必须通过 `tool_code` 指令生成。

4. 排版与工具调用 (Typesetting & Tool Calls)

4.1. 设计理念与实现

为支持富文本及动态内容生成，模型在 `typeset_and_respond` 阶段需采用特定的工具调用语法。该语法借鉴了业界先进模型（如 Gemini）对工具调用的支持，并经多轮迭代，已成为本软件 CoT 框架的标准实现。

所有工具调用必须封装于 `tool_code` 代码块内，并遵循如下格式：
`print(default_api.<function_name>(<args>))`

- `function_name`：需调用的功能名称，如 `send_image`。
- `args`：以 `key=value` 形式提供的参数列表，参数间以逗号 `,` 分隔。

4.2. 后端解析

为实现对该语法的精确解析，NPULearn 后端集成了基于 XLang-Rust 的解析器。该解析器可将 `tool_code` 内的指令转换为抽象语法树（AST），并交由前端或相关模块进行渲染。

技术选型说明：理想情况下可引入 `RustPython` 在沙箱环境中直接执行代码，但考虑到项目复杂性及依赖问题，当前版本暂未采用该方案。

4.3. 示例

发送一张图片：

tool_code
```
print(default_api.send_image(url="https://example.com/image.jpg"))
```

渲染 Markdown 表格：

tool_code
```
print(default_api.render_markdown(content="| Header | ... |\n|---|---|\n| Cell | ...
|"))
```

4.4. 多重调用

单个 `tool_code` 块内可包含多个指令，以分号 `;` 分隔。

tool_code
```
print(default_api.add_text(content="...")); print(default_api.send_image(url="..."))
```

兼容性说明：此处的 `;` 是自定义分隔符，非标准 Python 语法。为提升易用性，新版解析器已支持自动切分不带分号的多重表达式，但仍推荐使用分号以保证代码的明确性与向后兼容性。

5. 完整输出示例

以下是一个完整的、包含所有 CoT 阶段的内部输出样例。

text
```
<|start_title|>CoT技术文档说明<|end_title|>
```

```
<|start_header|>understand<|end_header|>
@startuml
actor User
participant Model as "NPULearn"
User -> Model: "帮我完善这个CoT说明文档"
note right of Model
- 核心需求：基于现有材料，完成一份CoT技术文档。
- 关键内容：解释CoT工作原理、流程、特殊标记和tool_code语法。
- 角色定位：技术文档撰写助手。
- 背景常识：技术文档应结构清晰、用词准确、包含示例。
end note
@enduml
<|start_header|>think<|end_header|>
@startuml
participant Model as "NPULearn"
note right of Model
1.  **规划文档结构**:
    - 概要 -> 核心理念 -> 执行流程 -> 工具调用 -> 示例 -> 提取规则 -> 效果总结。结构逻辑清晰。
2.  **内容填充**:
    - [理念] 阐述CoT的四大优势。
    - [流程] 使用表格清晰化每个阶段的任务和要求。
    - [工具调用] 说明语法、后端实现和多重调用规则。
    - [示例] 构造一个贴切的完整例子。
3.  **格式化**:
    - 全文使用中文，遵循用户提供的Typst格式。
    - 使用block美化代码示例区。
end note
@enduml
<|start_header|>verify<|end_header|>
@startuml
participant Model as "NPULearn"
note right of Model
- [覆盖度检查] Rust代码中的关键概念（如各阶段要求、纠错机制）均已覆盖。
- [结构检查] 文档结构层层递进，符合技术写作范式。
- [示例检查] 示例具有代表性，清晰展示了完整流程。
- [结论] 逻辑无误，可以生成最终响应。
end note
@enduml
<|start_header|>typeset_and_respond<|end_header|>
好的，非常乐意帮您完善这份NPULearn CoT技术文档。
// 优化：将自引用回复替换为更通用的示例
// (此处为优化后的Typst文档正文)
#heading("1. 概要")
本文档详尽阐述了 NPULearn 的核心技术——思维链...
```

6. 响应提取规则

在系统后端，CoT 的推理过程（如 `understand`, `think` 等）与最终响应在逻辑上相互隔离。系统通过如下规则从模型完整输出中提取面向用户的响应：

1. 定位**最后一次**出现的 `<|start_header|>typeset_and_respond<|end_header|>` 标记。

2. 该标记之后、下一个 `<|start_header|>` 标记之前（或至输出末尾）的所有内容，即被视为最终响应。

该机制确保最终呈现给用户的内容，均为模型经过多轮校验与排版优化后的结果。前端 UI 可选用折叠区域等方式，为开发者或高级用户提供完整推理链路的可视化选项，但该部分内容默认不直接展示。

7. 效果与价值

引入 CoT 框架后，NPULearn 在多个关键指标上获得了显著的性能提升：

- **逻辑与事实准确性**：在处理逻辑陷阱问题（如“草莓(strawberry)中有几个 r?”）和需要事实核查的场景时，准确率大幅提高，有效克服了由训练数据偏差引发的“幻觉”现象。
- **鲁棒性**：强制的 `理解` 和 `验证` 步骤显著降低了对用户意图的误判率，使 AI 的响应更加稳健和贴切。
- **可解释性与用户信任**：通过开放模型的思考过程，不仅为开发者提供了透明的调试路径，也增强了用户对 AI 决策的理解与信任。

上述改进共同将 AI 的可靠性与可信赖度提升至新的水平。

8. 缺陷

当前 CoT 设计不建议与 `Tool Calling` 并用。由于大语言模型的输出精度有限，模型可能会将 `tool_code` 内容与普通函数调用混淆，导致严重的输出格式错误等问题。

9. 未来发展方向

当前 NPULearn CoT 框架为模型的响应提供了可靠的逻辑基础。然而，其“一次性”生成完整思考链的模式，在处理需要连续、多步工具调用的复杂任务时，仍有其局限性。未来的发展核心，是将 CoT 框架从一个响应生成器，演进为一个能够自主规划、行动、观察和适应的流式自主代理（Streaming Autonomous Agent）。

该演进蓝图借鉴了 ReAct (Reason+Act) 模式的深度优化，旨在实现更高的响应效率与更强的任务处理能力。

9.1. 从单次思考链到代理循环

未来的核心架构将由线性的“理解-思考-验证-响应”流程，演进为动态、事件驱动的代理循环。

- **核心理念**：模型不再一次性完成所有推理，而是在循环中逐步推进。每一步均为“思考-行动”单元，可根据中间结果动态调整后续策略。
- **状态管理**：代理的完整“记忆”与“状态”通过对话历史（History）承载。系统可在历史中动态注入工具执行结果等伪系统消息，从而精确引导模型在循环中的每一步行为。

9.2. 流式处理与实时中断

为实现极致的用户体验与系统效率，未来交互模式将全面采用流式处理。

- **工作机制**：系统实时监控模型输出的文本流。一旦检测到模型意图调用工具（即生成 `tool_code` 块），系统将立即中断（Cancel）当前 API 请求，而非等待其自然结束。
- **核心优势**：
  1. **即时响应**：用户可几乎实时看到 AI 的行动意图，无需等待完整推理过程，显著降低感知延迟。
  2. **资源效率**：中断机制可避免为生成无关文本而产生的额外 Token 消耗与计算开销。

9.3. 思维模型深化

为提升复杂任务中的逻辑严密性，模型的推理过程将被强制拆分为两个层级、目标明确的阶段。

- **战术分析**（`analysis_result`）：每次工具调用后，模型需首先进行一次狭义的战术分析，其唯一目标为评估该次工具调用结果，并决定下一个直接行动。例如："获取股价成功，下一步用此数据调用分析工具"。
- **战略综合**（`final_thought`）：当所有必要信息收集完毕后，模型需进入全局、战略性的综合思考阶段，整合所有循环中获得的信息，规划最终呈现给用户的答案结构、内容与语气。

该分层设计可有效避免模型在复杂步骤中迷失方向，确保决策连贯性与最终答案质量。

9.4. 安全与控制

赋予 AI 更高自主权的同时，必须配套更为严格的安全与控制机制。

- **循环成本控制**：引入 `max_cycle_cost` 机制，为代理循环次数设置上限。该机制是防止因模型逻辑错误或任务无法收敛导致无限循环与资源耗尽的关键"熔断器"。
- **强输出格式约束**：在系统提示词中，对最终 `response` 块的内容格式（如仅允许一组安全、语义化的 HTML 标签）进行严格限定，确保代理最终输出可预测、可解析且安全，便于前端直接渲染。

通过上述四个维度的演进，NPULearn CoT 框架将由"思考者"转变为高效、智能且安全的"行动者"，以应对更为复杂和开放的真实世界任务，并显著提升智能学习助手的综合能力。

9.5. 实现

上述基于 `ReAct` 模式的 Agent 已由作者实现，详见：AutoGemini。未来可能考虑将其引入 NPULearn，以替代基础 CoT。