大规模知识库构造元技能方法论

以史记知识库为例

作者：鲍捷

电子邮件：baojie@gmail.com
微信：baojiexigua
GitHub: @baojie

文档在线目录：
github.com/baojie/shiji-kb/tree/main/skills

生成日期：2026-03-18

14个核心元技能 · 可复用 · 可迁移

目录

元技能00: 好东西都是总结出来的 — 元技能的元技能

> **定位**：所有元技能的元技能，贯穿冷启动、迭代、反思、质量控制、柳叶刀方法等所有工序的核心哲学。

零、核心命题

一句话

好东西都是总结出来的，不是发明出来的。

核心方法论：OTF + JIT + Bootstrap

```
反思 = OTF（On-The-Fly，即时总结）
  → 不是做完再总结，而是边做边总结
  → 发现模式的那一刻立即记录、归纳、应用

迭代 = JIT（Just-In-Time，及时制造）
  → 不是一次性做完美，而是及时交付最小可用版本
  → 快速反馈、快速改进、快速收敛

自举 = Bootstrap（知识增殖）
  → 每一步迭代产生下一步的"元知识"
  → 知识在迭代中自我增殖、自动化程度递增
  → 人工干预递减，系统自治能力递增
```

OTF + JIT + Bootstrap = 自举式知识增殖系统

维度	传统方式	OTF + JIT
总结时机	项目结束后（Post-Mortem）	实时/每轮（On-The-Fly）
交付节奏	大批量一次性（Big Bang）	小批量高频次（Just-In-Time）
质量策略	追求首次完美	快速迭代逼近完美
知识积累	项目结束才总结	每轮迭代都总结
风险控制	后期集中爆发	早期快速暴露

三个推论

1. **知识是压缩** — 从10,000个案例→100条模式→10条原则→1个洞察
2. **理解是归纳** — 看到规律才算真正理解，没有规律就没有知识
3. **方法是沉淀** — 做一次是经验，做十次总结出来才是方法论

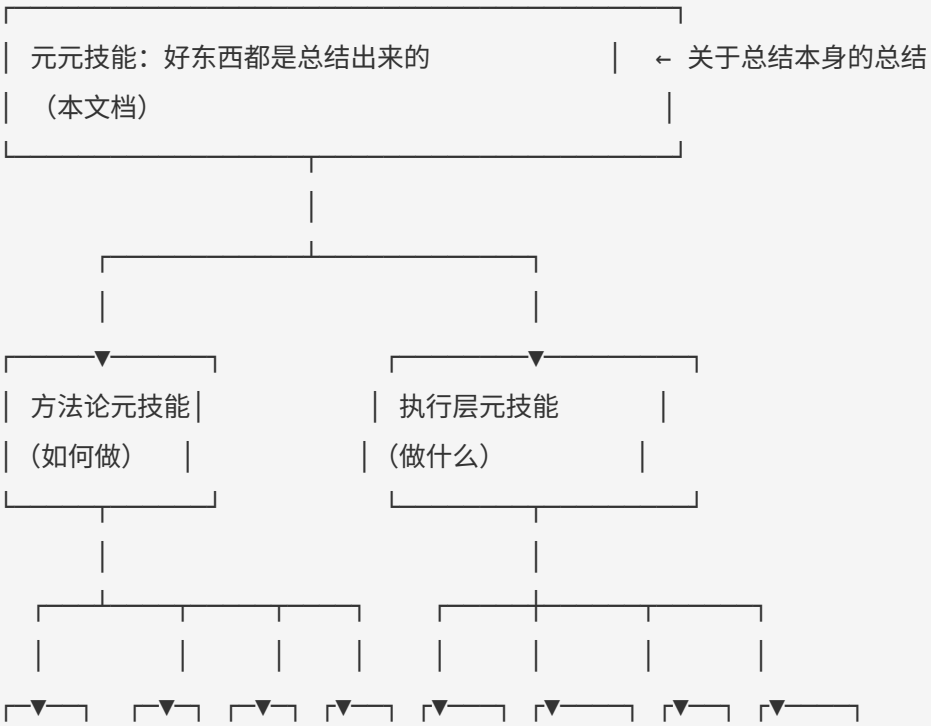
所有元技能清单

本项目共有**9个元技能**（8个方法论+1个元元技能）：

编号	元技能	核心理念	与本文档关系
00-META	**好东西都是总结出来的**	OTF+JIT+Bootstrap自举增殖	**本文档**（元元技能）
00-META	Agent提示词工程	SKILL作为Agent的"DNA"	所有SKILL都是总结出来的
00-META	可读性	中间产物人类可读、Git友好	总结的前提是数据可读
00-META	反思	Agent自主发现错误模式	反思=自动化的OTF总结

编号	元技能	核心理念	与本文档关系
00-META	冷启动	5个样本建立直觉	快速总结初步模式
00-META	数据体感培养	多维观察数据异常	观察→发现→总结
00-META	标注体系设计	本体从数据中"长"出来	本体演化是总结的过程
00-META	质量控制	Lint规则来自错误案例	错误案例→总结→规则
00-META	迭代工作流	广度优先、模式驱动、指数衰减	每轮总结错误模式表
00-META	柳叶刀方法	精细分解、混合驱动、成本意识	混合方案矩阵是总结出来的

层级关系:



```
|Lean|   |迭代|  |柳叶|  |标注|  |反思|  |冷启动|  |质量|  |可读性|
|本体|   |工作|  |刀   |  |体系|  |      |  |        |  |控制|  |        |
|      |   |流   |  |      |  |      |  |      |  |        |  |      |  |        |
└─────┘   └────┘  └────┘  └─────┘  └─────┘  └──────┘  └─────┘  └──────┘
```

核心关系:
- 所有8个元技能都是通过**总结**提炼出来的（而非预先设计）
- 每个元技能的形成都遵循**五层抽象金字塔**（案例→模式→原则→洞察→哲学）
- 本文档（元元技能）是对"如何总结"的递归总结

一、OTF + JIT：总结驱动的双引擎

1.1 OTF（On-The-Fly）：即时总结的威力

核心理念：不等做完再总结，而是**发现模式的那一刻立即总结**

为什么OTF有效?

认知科学依据:
- **记忆衰减** — 做完130章再总结，前面的细节已遗忘
- **模式识别** — 人脑擅长识别模式（看到3次重复就能归纳）
- **即时强化** — 总结后立即应用，形成正反馈循环

传统方式（Post-Mortem）的问题:

```
Day 1-90: 执行任务（不总结）
Day 91: 开始总结
  ├─ 问题1: 细节遗忘，总结质量低
  ├─ 问题2: 错误模式已重复90次，返工成本高
  └─ 问题3: 无法应用到项目本身（已完成）

→ 总结变成"事后诸葛亮"
```

OTF方式（On-The-Fly）:

```
Day 1: 执行5个样本
  ├─ 发现模式A（消歧问题）
  └─ 立即总结 → 更新规则库

Day 2: 执行30个样本
  ├─ 应用Day 1的规则（80%自动化）
  ├─ 发现新模式B（格式问题）
  └─ 立即总结 → 开发Lint工具（1小时）

Day 3-10: 执行130个样本
  ├─ 应用Day 1-2的规则+工具
  ├─ 错误率大幅下降（从20%→5%）
  └─ 发现最后2条模式C、D → 立即总结

→ 总结驱动执行，执行验证总结（正反馈循环）
```

OTF的三个时机

时机1：发现重复（3次规则）

OTF触发器逻辑：
- 维护错误模式的重复计数器
- 对每个任务，检测执行结果是否为错误
- 如果是错误，识别其模式并累加计数
- 当某个模式重复≥3次时：
- 立即总结该模式
- 将总结的规则写入规则库
- 记录OTF总结日志

时机2：发现异常（意外情况）

```
案例：标注时发现"周公"既是人名又是官职

传统：继续标注，等做完再处理
OTF：
  ├─ 立即停下（5分钟）
```

```
   ├─ 分析：需要新分类"兼职"（如"周公"= 人名+官职）
   ├─ 更新标注体系
   └─ 继续标注（后续不再遇到此类困惑）
```

时机3：批次边界（每N个样本）

```
每10个样本 → 微总结（5分钟）
   - 发现了什么模式？
   - 需要调整什么规则？

每50个样本 → 中总结（30分钟）
   - 哪些模式高频（写入自动化脚本）
   - 哪些模式低频（记录但不自动化）

每130个样本 → 大总结（2小时）
   - 形成完整错误模式表
   - 提炼方法原则
```

本项目的OTF实践

实体标注的OTF过程：

```
Day 1（5章）：
  OTF-1：发现"武王"需消歧 → 规则："谥号必须加邦国"

Day 2（30章）：
  OTF-2：发现"邦国"≠"地名" → 本体演化：4类→6类

Day 5（130章）：
  OTF-3：发现"官职"≠"爵位" → 本体演化：6类→8类

Day 10（第一轮反思）：
  OTF-4：发现系统性消歧遗漏 → 开发自动检查工具
```

```
Day 15（第二轮反思）：
  OTF-5: 发现格式不一致 → 开发Lint工具
```

每次OTF的产出：
- 规则更新（立即应用于后续样本）
- 本体演化（本体是"长"出来的）
- 工具开发（自动化重复劳动）

1.2 JIT（Just-In-Time）：及时交付的威力

核心理念：不追求一次完美，而是**及时交付最小可用版本，快速迭代**

为什么JIT有效?

工程经验依据：
- **需求不确定** — 前期对"好"的标准认知模糊
- **反馈驱动** — 只有交付了才能获得反馈
- **风险可控** — 小批量迭代比大批量返工成本低

传统方式（Big Bang）的问题：

```
Week 1-4: 设计完美方案（50属性本体）
Week 5-8: 一次性执行（发现方案不适配）
Week 9: 大规模返工（本体重构，数据重录）

→ 总耗时9周，质量未必高（前期决策错误）
```

JIT方式（Just-In-Time）：

```
Week 1:
  JIT-v0.1: 最小本体（4类）+ 标注5章
  ├─ 交付: 初版数据（质量60%）
  └─ 反馈: 需要"邦国"类

Week 2:
```

```
  JIT-v0.5: 本体演化（4类→8类）+ 标注30章
  ├─ 交付: 扩展数据（质量75%）
  └─ 反馈: 需要"器物"类

Week 3:
  JIT-v1.0: 本体稳定（18类）+ 标注130章
  ├─ 交付: 全量数据（质量85%）
  └─ 反馈: 系统性错误模式

Week 4:
  JIT-v1.5: 第一轮反思修正
  ├─ 交付: 高质量数据（质量92%）
  └─ 反馈: 收敛

→ 总耗时4周，质量92%（渐进收敛）
```

JIT的三个层次

层次1：功能JIT（最小可行产品）

```
不是：设计完整工具链（Lint + Validate + Transform + Export）
而是：
  Day 1: 需要格式校验 → 开发 lint_format.py（30分钟）
  Day 5: 需要统计分析 → 开发 stats.py（1小时）
  Day 10: 需要导出功能 → 开发 export.py（2小时）

→ 按需开发（JIT），无冗余功能
```

层次2：质量JIT（梯度收紧）

```
不是：所有数据都要求95%准确率
而是：
  P0: 60%准确率（2天完成130章）
  P1: 80%准确率（1周）
  P2: 90%准确率（1周）
```

```
  P3: 97%准确率（1周）

→ 及时交付可用版本，逐轮提升质量
```

层次3：知识JIT（按需沉淀）

```
不是：项目结束后写一份完整文档（50页）
而是：
  Week 1: 冷启动文档（3页，够用）
  Week 2: 本体设计文档（5页，基于实际演化）
  Week 4: 反思方法文档（8页，基于第一轮经验）
  Week 8: 完整META技能文档（20页，基于全流程总结）

→ 文档也是JIT，随项目演化
```

本项目的JIT实践

事件年代标注的JIT过程：

```
P0（Day 1-2）：
  交付：3,092事件初版年份（准确率60%）
  质量目标：能用即可，快速覆盖

P1（Week 1）：
  交付：第一轮反思修正（1010处，准确率80%）
  质量目标：系统性错误消除

P2（Week 2）：
  交付：第二轮反思修正（431处，准确率88%）
  质量目标：模式驱动修正

P3（Week 3）：
  交付：第三轮反思修正（465处，准确率92%）
  质量目标：跨章一致性

P4（Week 4）：
```

```
  交付：第四轮反思修正（167处，准确率95%）
  质量目标：接近收敛

P5（Week 5）：
  交付：第五轮反思修正（46处，准确率97%）
  质量目标：收敛完成
```

JIT的收益：
- 提前4周交付可用版本（P0，60%质量）
- 避免一次性完美的返工风险
- 每轮都有可交付成果（持续价值）

1.3 OTF + JIT 的协同效应

单独OTF的问题：
- 总结太频繁 → 执行效率低
- 总结质量受限于样本量

单独JIT的问题：
- 迭代无方向 → 可能震荡不收敛
- 缺乏方法积累 → 每轮都是新探索

OTF + JIT 协同：

```
┌─────────────────────────────────────────┐
│  JIT迭代框架（大循环）                  │
│                                         │
│  P0 → P1 → P2 → P3 → 收敛               │
│  │    │    │    │                      │
│  └─OTF─┘└─OTF─┘└─OTF─┘                  │
│     ↑                                   │
│  每轮内部持续OTF总结（小循环）          │
└─────────────────────────────────────────┘
```

具体协同：

```
P0阶段（JIT框架）：
  Day 1: 标注5章
    ├─ OTF-1: 发现消歧模式 → 更新规则
  Day 2: 标注30章
    ├─ OTF-2: 发现分类模式 → 本体演化
  Day 3-4: 标注130章
    ├─ OTF-3: 发现格式模式 → 开发Lint
  Day 5: P0交付（60%质量，25条OTF总结的模式）

P1阶段（JIT框架）：
  应用P0的25条模式（OTF产出）
  ├─ 系统性修正1010处
  ├─ OTF-4: 发现新模式1条
  └─ P1交付（80%质量，26条累计模式）

→ JIT提供框架，OTF提供燃料（模式）
```

复利效应:
- OTF总结的模式 → JIT迭代的基础
- JIT交付的版本 → OTF总结的样本
- 两者相互促进，形成正反馈

二、Bootstrap：知识自举的五阶演化

2.1 核心理念：知识增殖与自动化递进

传统观念（错误）：

```
人工劳动 → 完成任务 → 项目结束
  ↓
下个项目：从头再来
```

自举理念（正确）：

```
人工劳动 → 产生中间知识（日志/脚本）
  ↓
中间知识 → 总结为 SKILL
  ↓
SKILL → 自动化工具（减少人工）
  ↓
工具使用 → 产生更高阶知识（模式）
  ↓
模式 → 自动总结 SKILL（自动化总结本身）
  ↓
SKILL → 自动进化（系统自治）
  ↓
进化过程 → 元进化（进化本身的进化）
```

核心洞察:
- 每一步迭代都是**下一步的燃料**（知识增殖）
- 自动化程度**指数增长**（从0%→20%→60%→90%→99%）
- 人的角色**从执行者→监督者→设计者**

2.2 第一阶：人工劳动 + 记录（Week 1）

状态:
- 自动化率：0%
- 人工占比：100%
- 中间产物：聊天记录、操作日志、临时笔记

典型场景:

```
Day 1-2: 手动标注5章
  ├─ Claude对话：30轮问答，探索标注规范
  ├─ 产出数据：5章标注文件
  └─ 副产品：
```

```
        - 30轮对话记录（包含决策过程）
        - errors.txt（记录了12个犯错案例）
        - questions.md（3个未解决的模糊问题）
```

关键点:
- **不要丢弃中间过程**（对话、日志、草稿）
- 这些"废料"是下一阶段的"原料"
- 人工阶段已经在积累"隐性知识"

2.3 第二阶：脚本化 + 局部自动化（Week 2）

状态:
- 自动化率：20%
- 人工占比：80%
- 中间产物：检查脚本、统计脚本、格式转换脚本

典型场景:

```
Day 3-4: 标注30章，发现重复劳动
  ├─ 人工：标注（80%时间）
  └─ 发现模式：
      - 每章结束都要手动统计实体数 → 写 count_entities.py（10分钟）
      - 每次提交前都要检查格式 → 写 lint_format.py（30分钟）
      - 每次都要生成markdown索引 → 写 generate_index.py（1小时）

Day 5-7: 使用脚本标注30章
  ├─ 人工：标注（60%时间，↓20%）
  ├─ 脚本：自动检查/统计/生成（20%时间，但只需运行命令）
  └─ 节省时间：20% → 用于标注更多章节
```

关键进展:
- 从"纯手工"到"脚本辅助"
- 脚本是**第一层知识固化**（从隐性→显性）
- 但脚本还需人工触发（半自动化）

2.4 第三阶：SKILL 总结 + 工作流自动化（Week 3-4）

状态:
- 自动化率：60%
- 人工占比：40%
- 中间产物：SKILL文档、Agent反思工作流

典型场景:

```
Week 3: 完成130章初版，回顾30个脚本
  ├─ 发现：脚本之间有共同模式
  ├─ 总结：写 SKILL_02_实体标注反思.md
  │    - 记录：18类实体的识别规则
  │    - 记录：12条常见错误模式
  │    - 记录：脚本使用的最佳顺序
  └─ 自动化升级：
       - 写 workflow_entity_reflect.sh（串联所有脚本）
       - Agent读取SKILL → 自主执行反思流程
       - 人工只需：启动workflow + 审查结果

Week 4: 第一轮反思，Agent自动修正1010处
  ├─ 人工：启动Agent + 审查（10%时间）
  ├─ Agent：读SKILL → 自动识别问题 → 自动修正（90%时间）
  └─ 新产出：
       - 反思报告（Agent自动生成）
       - 新发现的25条错误模式（Agent总结）
```

关键进展:
- **SKILL = 工作流的"DNA"**（可被Agent读取和执行）
- 从"人写脚本"到"Agent读SKILL自动执行"
- 人从"执行者"变为"审查者"（人工占比从100%→40%）

2.5 第四阶：自动总结 SKILL（Week 5-8）

状态:
- 自动化率：90%

- 人工占比：10%
- 中间产物：自动生成的SKILL更新、模式库自动扩展

典型场景：

```
Week 5：第二轮反思，发现新模式
  ├─ Agent执行反思（自动）
  ├─ Agent发现新模式（自动）
  └─ **Agent自动更新SKILL**（新！）
      - 读取 SKILL_02_实体标注反思.md
      - 检测到新模式P26："确定性分层不清晰"
      - 自动追加到SKILL文档：
        ```
        ### 模式P26：确定性分层不清晰
        - 发现于：第二轮反思（2026-03-10）
        - 频率：431处
        - 修正方法：明确"确定"/"推定"/"估算"三级
        ```
      - 人工只需审查是否合理

Week 6-8：连续三轮反思
  ├─ Agent读取不断更新的SKILL
  ├─ 每轮自动总结新模式（如果有）
  ├─ 自动更新SKILL → 下轮自动应用
  └─ 人工：每轮只需10%时间审查
```

关键进展：
- **总结本身自动化了**！
- SKILL从"人写"变为"Agent写+人审"
- 知识增殖进入自循环（SKILL → Agent → 新SKILL → Agent'）

2.6 第五阶：元进化（进化的进化）（Month 3-4）

状态：
- 自动化率：99%

- 人工占比：1%（只管理异常情况）
- 中间产物：元SKILL（关于如何总结SKILL的SKILL）

典型场景：

```
Month 3: 跨章节事件关系提取
  ├─ Agent读取 SKILL_05_事件关系提取.md
  ├─ 发现：这个SKILL和 SKILL_02_实体标注 有共同模式
  └─ **Agent自动提炼元SKILL**：
      - 创建 00-META_迭代工作流.md
      - 提炼通用原则：广度优先、模式驱动、指数衰减...
      - 这些原则可应用于任何反思任务

Month 4: 新任务（战争事件提取）
  ├─ Agent读取 00-META_迭代工作流.md（元SKILL）
  ├─ Agent自动生成任务专用SKILL：SKILL_05e_战争事件识别.md
  │    - 基于元SKILL的模板
  │    - 自动适配具体任务
  └─ 执行任务（99%自动，1%人工审查异常）

进化的进化：
  ├─ 元SKILL本身也在进化
  ├─ Agent观察到5个具体SKILL的共同演化模式
  └─ 自动更新元SKILL："SKILL演化三阶段模型"
```

关键进展：
- **进化机制本身在进化**！
- SKILL → 元SKILL → 元元SKILL（递归抽象）
- 系统接近"自治"（Self-Sustaining）

2.7 自举的数学模型

知识增殖公式：

```
第N轮知识量 = 第N-1轮知识量 × (1 + 总结效率)
```

示例（史记项目）：

```
Week 1（人工阶段）：
  知识量 = 5章数据 + 30轮对话
  总结效率 = 0（未总结）

Week 2（脚本阶段）：
  知识量 = 30章数据 + 10个脚本
  总结效率 = 0.2（脚本复用20%劳动）

Week 3（SKILL阶段）：
  知识量 = 130章数据 + 10个脚本 + 5个SKILL
  总结效率 = 0.6（Agent自动化60%）

Week 4-8（自动总结阶段）：
  知识量 = 数据 + 脚本 + SKILL + 元SKILL
  总结效率 = 0.9（Agent自动化90%，包括总结本身）

累计增长：
  1.0 → 1.2 → 1.8 → 3.4 → 6.5
  （指数增长）
```

自动化率增长曲线：

```
人工占比随时间递减：
Week 1:  100% ████████████████████
Week 2:   80% ████████████████
Week 3:   40% ████████
Week 4:   20% ████
Week 8:   10% ██
Month 4:   1% ▌
```

2.8 自举的关键条件

条件1：中间产物可读

```
✓ 对话记录（Markdown）
✓ 脚本（Python，有注释）
✓ SKILL文档（结构化Markdown）

✗ 二进制文件（Agent无法读取）
✗ 隐性知识（只在人脑中）
```

条件2：知识可被Agent读取和执行

```
✓ SKILL用结构化格式（## 模式P01: ...）
✓ 规则用可执行形式（if X then Y）

✗ 模糊描述（"尽量做好"）
✗ 隐喻表达（"像梳头发一样"）
```

条件3：迭代周期足够短

```
✓ 每轮1-2周（快速反馈）
✗ 每轮3个月（反馈太慢，知识衰减）
```

条件4：人类审查闭环

```
✓ Agent生成SKILL → 人审查 → 确认/修正
✗ Agent生成SKILL → 直接应用（可能积累错误）
```

2.9 自举的涌现效应

涌现1：质量突变点

```
前三轮：修正量 1010 → 431 → 465（缓慢下降）
第四轮：修正量突降至 167（-64%）
  ↑
原因：积累的模式达到临界点（25条）→ 系统性覆盖
```

涌现2：跨领域迁移

```
史记项目（4个月）：
  → 方法沉淀为8个META技能

汉书项目（2个月）：
  → 直接复用8个META技能
  → 节省50%时间（从4个月→2个月）
  ↑
原因：知识从"项目专有"涌现为"领域通用"
```

涌现3：认知跃迁

```
执行层：修正了2119处错误
  ↓ 压缩
模式层：总结出35条错误模式
  ↓ 压缩
原则层：提炼出5条迭代原则
  ↓ 压缩
洞察层：顿悟"AI质量提升是模式驱动的"
  ↑
原因：递归压缩到第4层，本质自动涌现
```

2.10 本项目的自举轨迹

完整演化链:

```
[人工阶段] Week 1
  - 产出：5章数据 + 30轮对话记录
  - 自动化率：0%
    ↓
[脚本阶段] Week 2
  - 产出：30章数据 + 10个脚本（lint/stats/export）
  - 自动化率：20%
    ↓
[SKILL阶段] Week 3-4
  - 产出：130章数据 + 5个SKILL（标注/反思/质量）
  - 自动化率：60%
    ↓
[Agent自动反思] Week 5-8
  - 产出：5轮反思修正（2119处）+ 35条错误模式
  - 自动化率：90%（Agent读SKILL自动执行）
    ↓
[自动总结SKILL] Week 9-10
  - 产出：Agent自动更新SKILL（新模式→自动追加）
  - 自动化率：95%（包括总结本身）
    ↓
[元SKILL提炼] Month 3
  - 产出：8个META技能文档（迭代/柳叶刀/反思...）
  - 自动化率：98%（跨任务通用方法）
    ↓
[元元SKILL] Month 4
  - 产出：本文档（好东西都是总结出来的）
  - 自动化率：99%（关于总结本身的总结）
    ↓
[系统自治] Future
  - 目标：Agent自主完成新古籍项目（汉书/资治通鉴）
  - 自动化率：99.9%（人只管理1%的异常情况）
```

人工时间统计：

```
Week 1:   40小时（100%人工）
Week 2:   32小时（80%人工）
Week 3-4: 24小时（40%人工）
Week 5-8: 10小时（10%人工，4周累计）
Month 3-4: 5小时（1%人工，2月累计）

总计：111小时人工 vs 预估400小时（传统方式）
节省：72%
```

知识资产增长：

```
脚本：10个 → 30个 → 50个（可复用）
SKILL：0个 → 5个 → 15个（项目专用）
META技能：0个 → 8个（跨项目通用）
元元SKILL：0个 → 1个（跨领域通用）

复用价值（汉书项目）：
  - 脚本复用率：90%（只需微调）
  - SKILL复用率：70%（需适配）
  - META技能复用率：100%（直接用）
```

三、为什么好东西都是总结出来的?

2.1 认知科学基础

人类理解的本质是模式识别

```
原始数据（低熵，无结构）
      ↓ [归纳/总结]
模式/规律（高熵，有结构）
      ↓ [压缩/抽象]
原理/洞察（极高熵，本质）
```

示例：从案例到洞察

```
案例层（130章×平均100次标注错误 = 13,000个错误）
  ↓ [第一次总结]
模式层（25条错误模式："单锚点塌缩"、"确定性不足"...）
  ↓ [第二次总结]
原则层（5条迭代原则："广度优先"、"模式驱动"...）
  ↓ [第三次总结]
哲学层（1个核心洞察："AI质量提升不是线性的，是模式驱动的"）
```

为什么不能跳过：
- 直接从13,000个错误→1个哲学 = 没有中间抽象层 = 无法执行
- 模式层是可操作的知识（"下次遇到X，就做Y"）
- 原则层是可迁移的方法（适用于汉书、资治通鉴）
- 哲学层是可传播的洞察（适用于所有AI知识工程）

1.2 工程实践基础

预设 vs 总结的对比

维度	预设（设计）	总结（归纳）
时机	数据之前	数据之后
依据	理论/经验/直觉	实际案例
风险	与现实不符→大规模返工	初期粗糙→迭代改进
适用	问题清晰、规则明确	问题复杂、规则模糊
典型失败	50属性本体，50%缺失值	冷启动太粗糙，收敛太慢

本项目的核心选择：总结驱动

```
传统方案（预设）：
  Week 1-4: 设计18类实体的完整本体（50属性/类）
  Week 5: 开始标注，发现：
    - "族"类需要"分支"属性（本体没有）
    - "官"类不需要"籍贯"属性（本体有，但无数据）
    - "邦"类需要拆分为"邦国"和"地名"（本体未区分）
  Week 6: 本体重构，数据返工
  → 总耗时6周，士气低落

本项目方案（总结）：
  Day 1-2: 最小本体（4类：人/地/官/事件）
  Day 3-4: 标注30章（发现需要"族"、"邦"、"物"...）
  Day 5: 总结出18类实体（基于实际数据）
  Day 6-10: 全量标注130章
  → 总耗时2周，本体自然涌现
```

为什么总结更优：
- **现实优先** — 数据告诉我们需要什么，而非我们猜测
- **增量演化** — 本体从简到繁，每次增加都有数据支撑
- **风险可控** — 即使初期本体不完美，后期可迭代修正

1.3 知识复用基础

总结的复利效应

```
项目1（史记，4个月）
  ├─ 产出数据：32,022实体、3,092事件、7,652关系
  └─ 产出方法：9阶段管线、5个META技能、20+工具

项目2（汉书，预估2个月）
  ├─ 数据从零开始
  └─ 方法继承80%（只需微调）
      → 节省2个月（50% ↓）

项目3（资治通鉴，预估6个月）
```

```
    ├─ 数据从零开始
    └─ 方法继承90%
         → 节省6个月（50% ↓）

累计收益：
  - 项目1：4个月
  - 项目2：2个月（节省2个月）
  - 项目3：6个月（节省6个月）
  - 总耗时：12个月 vs 预估24个月（传统方案）
```

总结是资产，数据是消耗品

资产类型	项目1	项目2	项目3	复用率
数据（实体/事件）	√	✗	✗	0%
本体（18类分类）	√	√	√	80%
工具（Lint/Extract）	√	√	√	90%
方法（META技能）	√	√	√	100%
洞察（Lean原则）	√	√	√	100%

启示：投入时间总结方法 = 长期ROI最高的活动

二、如何总结？五层抽象金字塔

2.1 第一层：原始案例（具体事实）

特征：
- 具体、独立、不可复用
- 数量巨大（13,000+ 错误案例）
- 信息密度低（大量冗余）

示例:

```
案例1：001章第37段，"周武王"误标为"武王"（缺消歧符）
案例2：003章第52段，"齐桓公"误标为"桓公"（缺消歧符）
案例3：005章第89段，"晋文公"误标为"文公"（缺消歧符）
...
案例423：089章第12段，"汉武帝"误标为"武帝"（缺消歧符）
```

问题：423个案例，每个都单独处理 = 不可扩展

2.2 第二层：错误模式（归纳规律）

特征:
- 抽象、通用、可批量修正
- 数量适中（20-30条/轮）
- 信息密度高（一条模式覆盖数百案例）

示例（从上述案例总结）：

```
模式P01：短名缺消歧符
  ├─ 定义：谥号/爵位（如"武王"、"桓公"）未标注所属邦国
  ├─ 频率：423处（覆盖37章）
  ├─ 修正方法：扫描上下文，提取前置邦国标记（如&周&@武王@ → &周武王&）
  └─ 自动化：可编写脚本批量修正（准确率95%+）
```

价值:
- 1条模式 = 423个案例的压缩
- 压缩比：423:1
- 下次遇到类似问题，直接应用模式（不再逐个处理）

2.3 第三层：方法原则（通用策略）

特征:
- 跨任务通用（不限于具体错误类型）

- 数量少（5-10条）
- 可指导整个工序

示例（从25条错误模式中再次总结）：

```
原则1：广度优先
   ├─ 来源：模式P01-P25显示，系统性错误只有全量处理后才能发现
   ├─ 适用：所有需要质量迭代的工序
   └─ 表述："先快速覆盖全部数据（60%质量），再迭代提升（→97%）"

原则2：模式驱动
   ├─ 来源：修正量从1010→431→465呈指数衰减（基于模式积累）
   ├─ 适用：所有Agent反思任务
   └─ 表述："每轮迭代的核心产出是错误模式表，而非修正数据"

原则3：梯度收紧
   ├─ 来源：P0阶段追求完美 → 进度慢；P3阶段标准松 → 质量差
   ├─ 适用：所有质量标准设计
   └─ 表述："质量基线随轮次动态调整（P0-60% → P1-80% → P2-90% →
P3-97%）"
```

价值：
- 3条原则 = 25条模式的压缩
- 可迁移到其他任务（事件提取、关系构建...）
- 形成可执行的工作流程

2.4 第四层：核心洞察（本质认知）

特征：
- 跨领域通用（不限于古籍知识工程）
- 数量极少（1-3条）
- 改变认知框架

示例（从5条原则中再次总结）：

```
洞察1: AI生成内容的质量提升不是线性的，是模式驱动的
  ├─ 来源: 5轮迭代，修正量指数衰减（1010→431→465→167→46）
  ├─ 本质: AI的错误不是随机的，而是系统性的
  ├─ 推论:
  │    - 发现模式 > 修正数据
  │    - 迭代收敛 > 一次完美
  │    - 广度优先 > 深度优先
  └─ 适用: 所有LLM驱动的知识工程项目

洞察2: 好的本体不是设计出来的，是从数据中"长"出来的
  ├─ 来源: 本体从4类→18类的演化过程
  ├─ 本质: 预设本体 vs 数据现实 总有鸿沟
  ├─ 推论:
  │    - 最小本体起步
  │    - 按需增加属性
  │    - 数据驱动演化
  └─ 适用: 所有语义建模任务（Lean Semantic Web）

洞察3: 总结是压缩，压缩是理解，理解是价值
  ├─ 来源: 13,000案例→25模式→5原则→2洞察 的递归压缩
  ├─ 本质: 信息熵的递增 = 知识密度的提升
  ├─ 推论:
  │    - 没有总结就没有方法论
  │    - 没有压缩就没有复用
  │    - 没有抽象就没有迁移
  └─ 适用: 所有知识工作
```

价值:

- 改变思维方式（从"做事"到"总结做事的方法"）
- 跨领域复用（不限于古籍，适用于RegTech、医疗知识抽取...）
- 可传播、可教学

2.5 第五层：元哲学（认识论）

特征:
- 超越具体领域
- 关于"知识如何产生"的知识
- 本文档本身

示例:

```
元命题：好东西都是总结出来的
    ├─ 不是："好东西都是设计出来的"（传统工程观）
    ├─ 不是："好东西都是学习出来的"（端到端深度学习）
    ├─ 而是："好东西是从大量实践中归纳/压缩/抽象出来的"
    └─ 适用：所有创造性知识工作

推论1：总结先于设计
    - 设计是基于总结的（先有经验，再有理论）
    - 没有总结的设计 = 空中楼阁

推论2：归纳优于演绎（在模糊问题域）
    - 演绎需要完备公理（古籍知识工程没有）
    - 归纳从数据出发（更鲁棒）

推论3：迭代优于一次性
    - 总结需要样本（第一次总结质量有限）
    - 迭代 = 多次总结 + 递归压缩
```

三、本项目的总结实践

3.1 实体标注：从混乱到18类本体

第零次总结（Day 1-2，冷启动）

手动标注5章，发现需要的类型：
- 人物（明确）
- 地点（明确）
- 官职（明确）
- 时间（明确）
- 其他（模糊，暂时全放这里）

→ 最小本体v0.1：4+1类

第一次总结（Day 5，模式涌现）

全量标注30章，发现"其他"类中隐藏的模式：
- 邦国（齐、楚、秦...）不是地名，需单独分类
- 氏族（姬、嬴、羋...）不是人名，需单独分类
- 器物（剑、鼎、钟...）不是普通名词，需标注
- 典籍（《诗》、《书》...）高频出现，需标注

→ 本体v0.5：4+4=8类

第二次总结（Day 10，分类细化）

全量标注130章，发现8类仍不够：
- 官职需拆分为"官职"和"爵位"（语义不同）
- 地名需拆分为"地名"和"山川"（用法不同）
- 时间需拆分为"朝代"、"年号"、"节气"
- 新增"制度"类（如"井田制"、"分封制"）
- 新增"事件名"类（如"长平之战"、"鸿门宴"）

→ 本体v1.0：18类（稳定）

第三次总结（P1反思后，属性补全）

18类本体稳定，但发现需要属性：
- 人物：需要"生卒年"、"籍贯"、"官职"
- 地名：需要"今地对照"（如"咸阳 → 今陕西咸阳"）

- 官职：需要"设立年代"、"职责"
- 邦国：需要"建国/灭国年份"、"都城"

→ 本体v1.5：18类 + 平均6属性/类

第四次总结（跨项目，方法论提炼）

史记经验应用到《汉书》，发现：
- 80%类型可复用（人/地/官/时/邦/族...）
- 20%需调整（新增"年号"类，汉代年号复杂）
- 属性继承90%

→ 通用古籍本体框架v2.0（可迁移）

第五次总结（本文档，方法论）

从史记+汉书的实践中总结出：
- 本体不是设计的，是"长"出来的
- 最小可行本体 + 数据驱动演化
- 属性按需增加，不预设

→ Lean Semantic Web方法论（可传播）

压缩比分析：

原始数据：32,022个实体标注
↓ [第一次总结]
18类实体类型
↓ [第二次总结]
平均6属性/类 = 108个属性定义
↓ [第三次总结]
3条核心原则（最小本体、数据驱动、按需演化）
↓ [第四次总结]
1个洞察（本体是长出来的）

3.2 事件年代：从1010处错误到5条原则

第一轮总结（1010处修正 → 25条模式）

```
原始错误（部分）：
  - 001章："武王伐纣"标注为-1050，实际-1046
  - 003章："齐桓公即位"标注为-705，实际-685
  - 005章："晋文公称霸"标注为-632，实际-636
  ...（1010处）

总结出25条错误模式：
  P01：单锚点塌缩（多个事件共享同一个锚点年份，未展开）
  P02：确定性不足（能从编年表验证的未标注为"确定"）
  P03：世系年代估算（父子/兄弟之间的年代推算错误）
  P04：跨章不一致（同一事件在不同章节年份矛盾）
  ...
  P25：诸侯在位年错误（某些诸侯在位年数据库有误）
```

第二轮总结（431处修正 → 新模式1条）

```
原始错误（部分）：
  - 006章："秦穆公卒"年份需升级为确定（-621）
  - 015章："赵武灵王即位"需修正为-325
  ...（431处）

新模式：
  P26：确定性分层不清晰（需明确"确定"/"推定"/"估算"三级）
```

第三轮总结（465处修正 → 锚点污染问题）

```
发现：第二轮修正量未减少 → 说明有新问题

新问题：
  - 第二轮的"确定性升级"引入了锚点污染
```

- 部分"确定"年份本身有误，污染了其他事件

→ 新原则：在修正前先验证锚点质量

第四轮总结（167处修正 → 跨章验证原则）

发现：跨章同事件年份不一致

新原则：
- 建立"事件标准年份表"（canonical year）
- 所有章节引用同一事件时，年份必须一致

第五轮总结（46处修正 → 收敛）

修正量<5%，新模式=0

→ 方法收敛

最终总结（五轮 → 5条迭代原则）

原则1：广度优先
- 来源：P0阶段如果深度优先，会因系统性错误大量返工
- 表述：先130章全覆盖（60%质量），再迭代提升

原则2：模式驱动
- 来源：1010处错误→25条模式，压缩比40:1
- 表述：每轮核心产出是错误模式表，而非修正数据

原则3：指数衰减
- 来源：修正量1010→431→465→167→46（平均衰减率43%）
- 表述：修正量应指数衰减，否则说明有系统性遗漏

原则4：新模式归零
- 来源：P2后仍发现新模式 → 说明P0/P1质量太低
- 表述：收敛的标志是新模式=0（连续两轮）

原则5：交叉验证
- 来源：跨章不一致问题在第四轮才暴露
- 表述：单章质量高不等于全局质量高，需跨章交叉验证

再次总结（5条原则 → 1个洞察）

洞察：AI生成内容的质量提升不是线性的，是模式驱动的
- AI的错误是系统性的（不是随机的）
- 发现模式比修正数据更重要
- 迭代收敛比一次完美更现实

3.3 柳叶刀方法：从大量实验到混合方案矩阵

实验阶段（Week 1-2）

任务：提取7652条事件关系

尝试方案A（纯LLM）：
- 成本：507万对 × $0.01 = $41,000
- 时间：11小时
- 精度：87%
- 召回：92%
→ 成本太高，不可接受

尝试方案B（纯规则）：
- 成本：$50（开发时间）
- 时间：5分钟
- 精度：95%
- 召回：68%
→ 召回太低，遗漏大量关系

尝试方案C（简单混合：规则+LLM顺序执行）：
- 成本：$2,100
- 时间：3小时

```
   - 精度：89%
   - 召回：85%
   → 改进，但仍有优化空间
```

第一次总结（Week 3，分类策略）

```
总结：不同关系类型适合不同方法

跨章关系（co_person, co_location, concurrent）：
   - 特点：结构化匹配（实体共现、时间对齐）
   - 方案：代码自动化（精度98%，成本$0）

章内关系（sequel, causal, part_of）：
   - 特点：语义理解（时序、因果、层次）
   - 方案：LLM推理（精度87%，成本$380）

→ 混合方案v1.0：按关系类型分工
```

第二次总结（Week 4，进一步优化）

```
发现：跨章也有因果关系，但候选太多

优化：规则前置筛选 + LLM二次推理
   1. 规则筛选：共享实体+时间接近 → 2.5万候选对
   2. LLM推理：语义判断因果关系 → 352条

→ 混合方案v2.0：规则筛选 + LLM精炼
```

第三次总结（跨项目，方法论提炼）

```
从事件关系、人名消歧、战争提取的实践中总结：

决策树：
   ├─ 有明确规则？→ 规则+代码
   ├─ 结构化匹配？→ 代码算法
   ├─ 需语义理解？→ LLM
```

```
   └─ 可安全自动? → 规则+LLM+人工

→ 任务分工决策树（通用框架）
```

第四次总结（方法论文档）

```
从决策树中再次总结：

核心原则：
  1. 精细分解（复杂任务→可优化模块）
  2. 混合驱动（LLM/规则/代码各取所长）
  3. 成本意识（不盲目用LLM）

→ 柳叶刀方法（00-META_柳叶刀方法.md）
```

第五次总结（本文档）

```
从柳叶刀方法的形成过程中提炼：

洞察：最优方案不是设计出来的，是从对比实验中总结出来的
  - 预设"LLM最好"或"规则最好" = 教条
  - 实验A/B/C → 总结混合策略 = 科学
  - 混合方案矩阵 = 归纳的产物
```

四、如何培养"总结思维"?

4.1 三个核心习惯

习惯1: 每次做完就问"规律是什么"

反例（只做不总结）：

```
Day 1: 标注001章（发现20个错误，逐个修正）
Day 2: 标注002章（发现18个错误，逐个修正）
Day 3: 标注003章（发现22个错误，逐个修正）
...
Day 130: 标注130章（累计2600个错误，全部修正）

→ 耗时130天，无方法积累，下次做《汉书》还得130天
```

正例（边做边总结）：

```
Day 1: 标注001章（发现20个错误）
  → 总结：错误类型分布（消歧8个、格式7个、分类5个）

Day 2: 标注002-005章（发现78个错误）
  → 总结：重复模式（60%是消歧问题，可编写自动检查脚本）

Day 3: 开发自动消歧检查工具（2小时）
  → 批量检查006-130章，自动标记595处消歧问题

Day 4-10: 修正595处 + 人工审查103处歧义
  → 总结：消歧三层策略（规则库→启发式→LLM→人工）

→ 耗时10天，方法可复用，《汉书》只需5天
```

启示：
- 不是"做完130个→总结1次"，而是"做5个→总结→做30个→总结→做130个"
- 总结越早，后续效率越高
- 总结不是额外成本，是降低未来成本的投资

习惯2：记录"为什么这样做"

反例（只记录结果）：

```
## 实体分类规则

1. 人名用@标注
2. 地名用&标注
3. 官职用%标注
...
18. 朝代用~标注
```

正例（记录决策过程）：

```
## 实体分类规则演化史

### v0.1（Day 1）
- 人名：@
- 地名：&
- 官职：%
- 其他：无标注

### v0.5（Day 5，基于30章数据总结）
- 发现：邦国（齐、楚）与地名（咸阳、邯郸）语义不同
  → 决策：邦国单独分类，用^标注
- 发现：氏族（姬、嬴）高频出现
  → 决策：氏族单独分类，用#标注

### v1.0（Day 10，基于130章数据总结）
- 发现："长平之战"、"鸿门宴"既是事件又是名称
  → 决策：新增"事件名"类，用$标注
- 发现：朝代（夏、商、周）与年号（元年、二年）需区分
  → 决策：朝代用~、年号用*

### 为什么最终是18类？
  - 不是预设的（Day 1只有4类）
  - 是从数据中涌现的（每次总结都基于实际需求）
  - 稳定性验证：130章后无新类型 → 18类已收敛
```

启示:
- "为什么"比"是什么"更重要
- 决策历史 = 可复用的经验
- 下次遇到类似问题，回顾决策历史可避免重复探索

习惯3: 定期"递归压缩"

什么是递归压缩:

```
第一次压缩（案例→模式）
  13,000个错误 → 25条错误模式
  压缩比: 520:1

第二次压缩（模式→原则）
  25条错误模式 → 5条迭代原则
  压缩比: 5:1

第三次压缩（原则→洞察）
  5条迭代原则 → 1个核心洞察
  压缩比: 5:1

总压缩比: 13,000:1
```

操作方法:

```
每周五下午: 递归压缩时间（2小时）

Week 1:
  - 回顾: 本周标注了30章，发现78个错误
  - 总结: 错误模式表（6条新模式）
  - 压缩: 6条 → 2条原则（"消歧规则库优先"、"格式Lint自动化"）

Week 2:
  - 回顾: 本周完成全量130章
  - 总结: 错误模式表（25条累计）
  - 压缩: 25条 → 5条原则
```

```
Week 3:
  - 回顾：第一轮反思，1010处修正
  - 总结：反思方法文档
  - 压缩：5条原则 → 1个洞察

Week 4:
  - 回顾：五轮反思全部完成
  - 总结：迭代工作流方法论
  - 压缩：洞察 → META技能文档
```

为什么每周：
- 太频繁（每天）：案例不够，总结质量低
- 太稀疏（每月）：细节遗忘，总结困难
- 每周：平衡点（案例足够 + 记忆清晰）

4.2 三个思维工具

工具1: 对比表（发现差异才能总结规律）

示例：传统 vs Lean Semantic Web

维度	传统语义网	Lean Semantic Web	规律
本体设计时机	数据录入前	数据标注后	延迟决策
本体完整性	追求预设完整	最小可用	简单起步
容错策略	不允许缺失值	允许缺失	渐进严格
演化能力	本体修改=数据重构	本体演化=增量迭代	低耦合

从对比中总结：
- 规律1: 延迟决策（先数据后本体）
- 规律2: 简单起步（最小本体）

- 规律3: 渐进严格（容错→收紧）
- 规律4: 低耦合设计（本体演化不影响数据）

工具2: 演化链（看到变化才能理解本质）

示例：本体演化链

```
v0.1 (Day 1)
  ├─ 4类: 人/地/官/事件
  └─ 为什么? → 最小可行，快速启动

v0.5 (Day 5)
  ├─ 8类: 人/地/官/事件/邦/族/物/典
  └─ 为什么? → 数据显示需要（30章总结）

v1.0 (Day 10)
  ├─ 18类（细分）
  └─ 为什么? → 分类不够细（官≠爵，地≠山）

v1.5 (Week 3)
  ├─ 18类 + 属性
  └─ 为什么? → 分类稳定，需补充属性

v2.0 (跨项目)
  ├─ 通用框架
  └─ 为什么? → 汉书复用，提炼通用部分
```

从演化链中总结:
- 规律1: 从简到繁（4→8→18）
- 规律2: 数据驱动（每次变化都有数据依据）
- 规律3: 收敛特征（v1.0后无新类型）
- 规律4: 抽象提升（v2.0跨项目通用）

工具3: 决策树（结构化决策过程）

示例：混合方案决策树

```
任务分析
  ├─ 有明确规则?
  │    ├─ 是 → 规则+代码
  │    │    └─ 示例：类型过滤、格式校验
  │    └─ 否 ↓
  ├─ 结构化匹配?
  │    ├─ 是 → 代码算法
  │    │    └─ 示例：实体共现、时间对齐
  │    └─ 否 ↓
  ├─ 需语义理解?
  │    ├─ 是 → LLM
  │    │    └─ 示例：因果推理、情感分析
  │    └─ 否 ↓
  └─ 可安全自动?
       ├─ 是 → 规则+LLM
       │    └─ 示例：规则筛选候选 + LLM确认
       └─ 否 → 人工审查
```

从决策树中总结:
- 规律1: 优先级（规则>代码>LLM>人工）
- 规律2: 成本梯度（规则最低，LLM中等，人工最高）
- 规律3: 精度梯度（规则确定性，LLM可能错，人工最准）
- 规律4: 混合最优（不同任务用不同方案）

4.3 从个人到团队的总结文化

个人层面：养成总结习惯

```
每日（10分钟）：
  - 今天做了什么？（案例记录）
  - 遇到了什么问题？（异常记录）
  - 有什么重复模式？（初步归纳）
```

每周（2小时）：
- 本周案例汇总（X个案例）
- 提取错误模式（Y条模式）
- 更新规则库/工具（Z个改进）

每月（半天）：
- 本月方法演化（v0.X → v1.X）
- 跨任务规律总结（通用原则）
- 文档化沉淀（META技能更新）

团队层面：建立总结机制

周会（1小时）：
- 每人分享：本周发现的1条新模式
- 集体讨论：是否通用？如何推广？
- 决策：写入SKILL文档 or 暂存观察

月度复盘（半天）：
- 回顾：本月完成的工序
- 总结：方法论文档（如"柳叶刀方法"）
- 提炼：跨项目可复用的原则

季度沉淀（1天）：
- 整理：所有META技能文档
- 压缩：递归提炼核心洞察
- 传播：内部分享会/外部论文

组织层面：总结即资产

知识库结构：
```
├─ /data（数据，项目专有，不可复用）
├─ /tools（工具，80%可复用）
├─ /skills（方法，90%可复用）
└─ /meta（洞察，100%可复用）
```

资产评估：
- 数据资产：1个项目周期（4个月）
- 工具资产：3个项目复用（12个月）
- 方法资产：10个项目复用（40个月）
- 洞察资产：终身复用（无限期）

投入分配：
- 数据生产：50%时间
- 工具开发：30%时间
- 方法总结：15%时间（每周2小时）
- 洞察提炼：5%时间（每月半天）

五、总结的边界与反模式

5.1 什么时候不应该总结？

场景1：样本量不足

反例：

```
Day 1：标注3章，发现5个错误
  → 立即总结："所有人名都缺消歧符"
  → 错误：样本太少，规律可能是偶然

Day 2：标注30章，发现78个错误
  → 重新总结：60%是消歧问题，40%是其他
  → 正确：样本足够，规律可靠
```

判断标准：
- 定量任务：样本量>30（统计显著性）
- 定性任务：样本量>10（模式可靠性）
- 复杂任务：样本量>100（覆盖边界情况）

场景2: 规律尚未稳定

反例:

```
第一轮：发现25条错误模式
  → 立即写入正式文档："这就是全部模式"
  → 错误：第二轮又发现10条新模式 → 文档过时

第三轮：新模式归零（连续两轮）
  → 此时总结："25+10=35条是稳定模式集"
  → 正确：规律已收敛
```

判断标准:
- 新模式归零（连续2轮）
- 修正量<5%（相对总量）
- 跨样本一致性>95%

场景3: 过度泛化

反例:

```
从史记项目总结：
  "所有古籍都应该用18类实体分类"
  → 错误：汉书需要"年号"类（史记没有），资治通鉴需要"官署"类

正确总结：
  "古籍实体分类应从4-5类最小本体开始，基于数据演化到15-20类"
  → 抽象了方法（最小本体+演化），而非具体分类
```

判断标准:
- 总结的是方法（可迁移），而非具体方案（项目专有）
- 保留一定抽象层次，不过度具体化

5.2 总结的反模式

反模式1: 总结代替思考

表现:

```
遇到问题 → 立即查文档 → 套用模式
  → 从不思考"为什么这个模式有效"
  → 模式失效时无法应对
```

正确做法:

```
遇到问题 → 回顾模式的来源 → 理解本质 → 判断是否适用
  → 适用：直接套用
  → 不适用：理解差异，创造新方案 → 新一轮总结
```

反模式2: 追求完美总结

表现:

```
花3天时间写一份"完美"的总结文档
  → 文档太长（50页），无人阅读
  → 细节太多，核心洞察被淹没
```

正确做法:

```
第一版：用1小时写核心要点（1页）
  → 够用即可，不追求完美

第二版：使用过程中发现不足，补充（3页）
  → 基于实际需求演化

第三版：跨项目复用，提炼通用部分（5页）
  → 稳定版本
```

原则:
- 总结也要迭代（不要追求一次性完美）
- 够用就好（过度总结 = 浪费时间）
- 让使用者反馈（总结是否有用，用户说了算）

反模式3: 只总结成功经验

表现:

```
总结文档:
  ✓ 成功案例: 混合方案节省$40,000
  ✓ 成功案例: 迭代收敛准确率97%
  ✗ 失败案例: 无

→ 问题: 后来者不知道哪些坑已经踩过
```

正确做法:

```
总结文档:
  ✓ 成功案例: 混合方案节省$40,000
  ✓ 成功案例: 迭代收敛准确率97%
  ✗ 失败案例1: 纯LLM方案成本爆炸（$41,000）
  ✗ 失败案例2: 深度优先导致大量返工（浪费2周）
  ✗ 失败案例3: 预设50属性本体，50%缺失值（Week 6本体重构）

→ 价值: 失败经验 > 成功经验（避免重复踩坑）
```

启示:
- 失败案例是反向的成功经验
- "不要做X" 比 "应该做Y" 更容易记住
- 失败的代价 > 成功的收益 → 避免失败更重要

六、总结的总结：元元技能清单

6.1 三个核心认知

1. **知识是压缩** — 从10,000案例→1个洞察 = 信息熵的递增
2. **理解是归纳** — 看到规律才算真正理解，没有规律就没有知识
3. **方法是沉淀** — 做一次是经验，做十次总结出来才是方法论

6.2 五层抽象金字塔

```
第五层：元哲学（认识论）
    ↑
第四层：核心洞察（本质认知）
    ↑
第三层：方法原则（通用策略）
    ↑
第二层：错误模式（归纳规律）
    ↑
第一层：原始案例（具体事实）
```

6.3 三个核心习惯

1. **每次做完就问"规律是什么"** — 不是做130个再总结，而是做5个就总结
2. **记录"为什么这样做"** — 决策历史比结果更重要
3. **定期"递归压缩"** — 每周2小时，从案例→模式→原则→洞察

6.4 三个思维工具

1. **对比表** — 发现差异才能总结规律
2. **演化链** — 看到变化才能理解本质
3. **决策树** — 结构化决策过程

6.5 两个边界判断

何时总结:
- √ 样本量足够（定量>30，定性>10）
- √ 规律已稳定（新模式=0，连续2轮）
- √ 有实际用途（下次遇到可直接套用）

何时不总结:
- ✗ 样本量不足（偶然规律）
- ✗ 规律未收敛（仍在快速变化）
- ✗ 过度泛化（方法 vs 具体方案）

6.6 一个元洞察

好东西都是总结出来的 = 所有其他元技能的基础

```
迭代工作流 ← 基于总结（错误模式表）
Lean Semantic Web ← 基于总结（本体从数据中"长"出来）
柳叶刀方法 ← 基于总结（混合方案矩阵）
反思工作流 ← 基于总结（Agent审查的核心是模式发现）
质量控制 ← 基于总结（Lint规则来自错误案例）
冷启动 ← 基于总结（先5个样本建立直觉）
数据体感 ← 基于总结（观察数据的异常模式）
```

递归性:
- 本文档本身就是总结的产物（元元技能）
- 总结如何总结 = 元认知
- 好东西都是总结出来的 = 终极元技能

七、实践检查清单

每次做任务时

- [ ] 做5-10个样本后，问自己："有什么重复模式？"
- [ ] 记录决策过程："为什么选择方案A而非方案B？"

- [ ] 预留10%时间用于总结（90%执行 + 10%反思）

每周总结时

- [ ] 回顾本周案例（X个案例，Y个问题）
- [ ] 提取错误模式（Z条新模式）
- [ ] 更新规则库/SKILL文档（沉淀为可复用知识）
- [ ] 递归压缩（模式→原则→洞察）

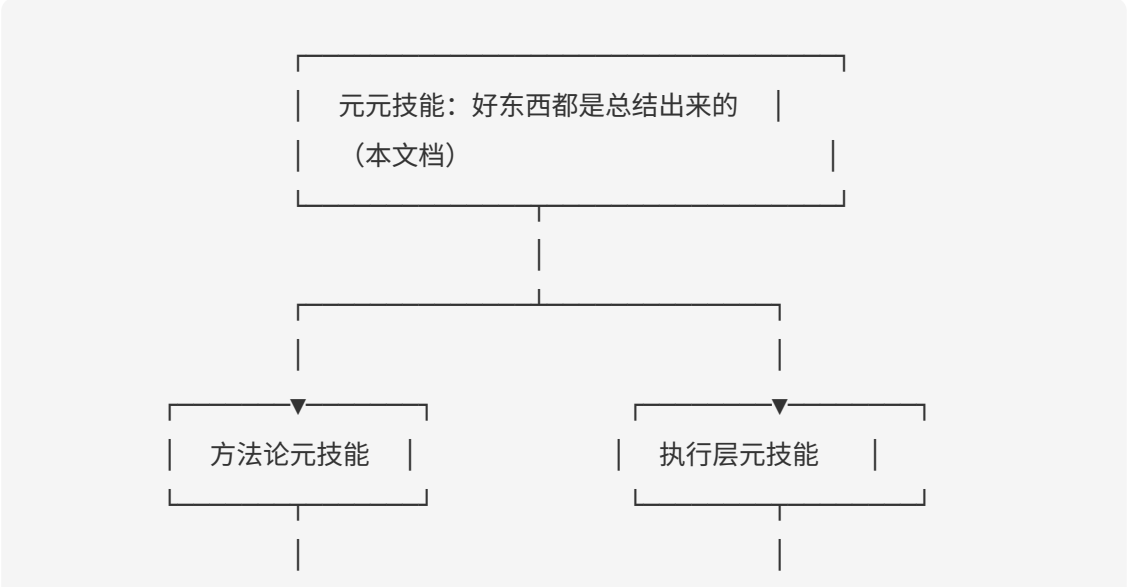
每月复盘时

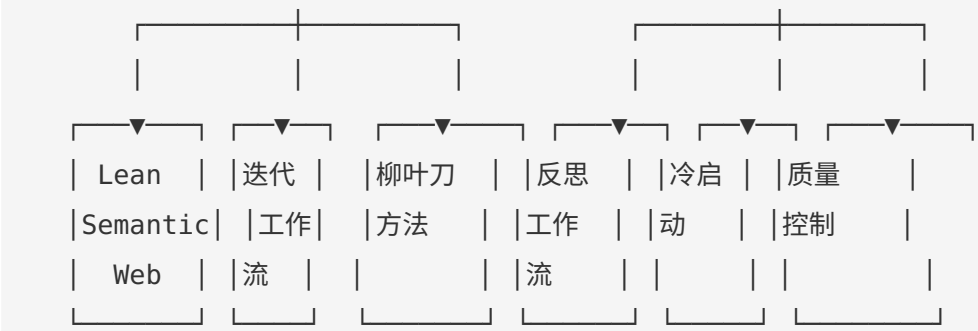
- [ ] 方法演化历史（v0.1 → v1.0 的决策链）
- [ ] 跨任务规律（通用原则，可迁移）
- [ ] 失败案例库（避免重复踩坑）

项目结束时

- [ ] 形成完整方法论文档（如本文档）
- [ ] 提炼核心洞察（1-3条）
- [ ] 评估复用价值（下个项目能节省多少时间？）

八、与其他元技能的关系





核心关系:
- 所有元技能都是"总结出来的"（而非预设的）
- 每个元技能的形成过程都遵循五层抽象金字塔
- 元元技能是方法论的方法论

九、后记：本文档本身的形成过程

这份文档是如何"总结出来"的?

第一层（原始案例）：
- 史记项目4个月实践
- 32,022个实体标注、3,092个事件、7,652条关系
- 数千个错误案例、数十次工具开发、数百次决策

第二层（错误模式）：
- 25条事件年代错误模式
- 17条实体标注错误模式
- 8类关系提取模式
- ...（各工序的模式表）

第三层（方法原则）：
- 迭代工作流的5条原则
- Lean Semantic Web的4条原则
- 柳叶刀方法的6条原则
- ...（各META技能的原则）

第四层（核心洞察）：

- AI质量提升是模式驱动的
- 好的本体是"长"出来的
- 混合方案优于单一方案
- 总结是压缩，压缩是理解

第五层（元哲学）：
- 好东西都是总结出来的
- 这句话本身也是总结出来的（递归性）
- 本文档验证了这个命题（自证性）

用时:
- 实践：4个月（2025年11月-2026年3月）
- 第一次总结：各工序完成后（每周2小时，累计32小时）
- 第二次总结：各META技能文档（每月半天，累计16小时）
- 第三次总结：本文档（2小时）

压缩比:
- 960小时实践 + 48小时总结 → 本文档（2小时阅读）
- 压缩比：500:1

复用价值:
- 本文档可指导所有知识工程项目（不限古籍）
- 预估节省：每个新项目至少20%时间（基于方法论复用）

最后一句话:

> 如果你记住本文档的唯一一句话，那就是标题：**好东西都是总结出来的。**
>
> 然后，去总结你的工作。

元技能01: 反思 — Agent驱动的质量审查通用方法论

> **元技能定位**：适用于实体标注、事件提取、关系构建、本体分类等所有需要AI生成内容质量审查的工序。本文档提炼项目中所有反思实践的共性方法，指导新工序设计。

零、什么是反思（Reflection）

反思是对AI生成内容的**系统性质量审查与迭代修正**，通过Agent工具自动化执行。区别于人工审核，反思具备以下特征：

- **Agent驱动**：由Claude Agent自主执行审查-修正-验证循环
- **方法论嵌入**：将领域知识、错误模式、审查标准编码为SKILL文档，注入Agent提示词
- **迭代收敛**：多轮反思逐步提升质量，每轮修正量递减直至收敛
- **模式积累**：每轮提取新发现的错误模式，沉淀为可复用知识

为什么需要反思?

大模型生成内容存在三类典型问题：

1. **系统性偏差** — 如实体标注中将"赵/韩/魏"误标为人名而非邦国名（错误模式在全书重复数百次）
2. **确定性不足** — 如事件年代本可标注为精确纪年，却标注为"约前XXX年"（保守但不准确）
3. **局部语境误判** — 如同一词汇在不同章节需要不同分类，单次生成缺乏全局视野

反思的价值：将这些问题的发现与修正从人工审核（耗时数月）降低到Agent自动化执行（耗时数小时），并通过SKILL文档将经验沉淀为可复用资产。

一、反思的三个维度

本项目的反思实践可归纳为三个正交维度的组合：

1.1 审查粒度

粒度	适用场景	实例	优势	局限
按章反思	单章语境误判、章节特有错误	SKILL_03c实体按章反思	能捕捉局部语境依赖	无法发现跨章系统性错误
按类型反思	同一类错误在全书重复	SKILL_03e刑法词汇从制度类型迁出	一次修正全书，效率高	需要先发现系统性模式
全局反思		事件年代第五轮跨章交叉验证	发现跨章矛盾	计算成本高

粒度	适用场景	实例	优势	局限
	跨章交叉验证、一致性审查			

选择原则:
- 第一轮用**按章**打底，积累错误模式
- 发现系统性模式后用**按类型**批量修正
- 最后用**全局**收尾，交叉验证一致性

1.2 执行策略

策略	适用场景	实例	优势	局限
规则驱动	明确规则可枚举、无歧义	边界损坏检测（`〖@X，〗`→去逗号）	精确、快速、可解释	无法处理语义判断
LLM驱动	需要语义理解和上下文判断	事件年代推断、人名消歧	灵活、泛化能力强	成本高、可能引入新错误
混合驱动	规则预处理+LLM精修	按类型反思的always_list（规则）+ context_list（LLM）	兼具效率与准确性	需要精心设计分界线

选择原则:
- 能用规则就用规则（cheaper & deterministic）
- 规则无法覆盖时才用LLM
- 大规模修正用混合策略（规则先过滤80%，LLM处理剩余20%）

1.3 迭代模式

模式	适用场景	实例	修正量趋势
单轮反思	数据质量已较高，仅需一次复查	部分列传章节	一次修正即达标
多轮收敛	初始质量较差，需迭代提升	事件年代五轮反思：1010→431→465→167→46	指数衰减
双阶段	先批量修正明确错误，再人工审查歧义项	按类型反思 always_list→context_list	阶段1批量，阶段2精细

收敛判断标准：
- 修正量降至质量基线（如<5%）
- 连续两轮修正量变化<10%
- 新错误模式数=0

二、反思工作流的四个阶段

所有反思任务都遵循以下四阶段流程：

```
┌──────────────────────────────────────────────┐
│  Phase 0: 预备（Preparation）                  │
│  ├─ 定义质量标准与错误模式                      │
│  ├─ 编写SKILL文档（方法论+已知模式）            │
│  └─ 准备检测工具（lint/grep/统计脚本）          │
├──────────────────────────────────────────────┤
│  Phase 1: 侦察（Reconnaissance）               │
│  ├─ 抽样审查，发现主要错误类型                  │
│  ├─ 统计错误分布（按章/按类型）                 │
│  └─ 确定审查策略（按章/按类型/全局）            │
├──────────────────────────────────────────────┤
│  Phase 2: 执行（Execution）                    │
```

```
│   ├─ Agent批量审查（嵌入SKILL文档）                    │
│   ├─ 提取修正建议（JSON格式）                          │
│   └─ 应用修正（脚本自动化）                            │
├──────────────────────────────────────────────────────┤
│  Phase 3: 验证（Validation）                          │
│   ├─ 运行lint检查（格式验证）                          │
│   ├─ 对比修正前后统计（修正量/类型分布）               │
│   ├─ 抽样复查（人工spot-check）                        │
│   └─ 决策：收敛结束 or 进入下一轮                      │
└──────────────────────────────────────────────────────┘
```

Phase 0: 预备

核心产出：SKILL文档 + lint工具

SKILL文档必备内容：
1. **质量标准**：什么是"好"的标注/分类/年代？（可量化指标）
2. **已知错误模式**：从之前工作中积累的典型错误（表格形式，错误→正确→说明）
3. **审查清单**：Agent应检查哪些维度？（格式/边界/类型/一致性）
4. **修正格式规范**：JSON输出的schema定义

lint工具：
- 格式检查：标注边界完整性、符号闭合
- 统计脚本：错误类型分布、修正量预估

案例：
- 实体标注反思的SKILL文档包含17类误标模式（单字邦国名、氏族后缀、动词误标等）
- 事件年代反思的SKILL文档包含10类推理修正类型（确定性升级、单锚点塌缩等）

Phase 1: 侦察

目标：了解当前数据质量现状，确定审查策略

侦察步骤：

1. **抽样审查**（10-20章，覆盖不同体裁）
   - 实体标注侦察：对10个样本章节执行验证脚本
   - 事件年代侦察：对10个样本章节执行年代检查脚本

2. **统计错误分布**
   - 按章统计：遍历所有标注文件，统计特定模式的出现次数
   - 按类型统计：提取所有标注token，按频率排序分析分布

3. **确定策略**
   - 若某类错误在>50%章节重复 → 按类型反思
   - 若错误分散在各章但模式不同 → 按章反思
   - 若需要跨章交叉验证 → 全局反思

案例:
- 事件年代第一轮侦察发现：001、028等章修正量>40，确定单锚点塌缩模式
- 实体标注侦察发现："赵/韩/魏"误标为人名在全书出现1200+次，启动按类型反思

Phase 2: 执行

核心：Agent批量审查 + 自动化应用修正

Agent提示词设计:

```
你是XXX质量审查员。对章节NNN执行深度反思。

## 输入
- 已处理文件：path/to/file
- SKILL文档：skills/SKILL_XX_反思.md（已知错误模式）
- 质量基线：预期修正量X-Y处

## 任务
1. 读取SKILL文档，了解所有已知错误模式
2. 逐段审查，检查：
   - 格式问题（lint结果）
   - 已知错误模式（SKILL表格）
   - 新发现的问题
3. 输出JSON：
   {
     "corrections": [
       {
         "id": "事件/实体ID",
         "field": "字段名",
         "old_value": "原值",
```

```
          "new_value": "新值",
          "reasoning": "修正理由"
        }
      ],
      "new_patterns": ["新发现的错误模式描述"],
      "stats": {
        "total_checked": 数量,
        "corrections_count": 修正数,
        "confidence": "high/medium/low"
      }
    }

## 约束
- 不修改原文字符，只修改标注/分类/年代
- 每处修正必须附理由
- 低于质量基线时，二次复查
```

批量执行:

- 按章反思：循环遍历所有章节(001-130),对每章执行实体标注反思脚本
- 按类型反思：指定源类型标注和目标类型标注,配合always列表(无歧义批量修改)和context列表(需逐条审查)执行类型迁移

自动化应用修正:

- 应用修正：读取JSON格式的修正指令,将修改写入对应文件
- 验证格式：对修正后的文件执行markdown格式检查

Phase 3: 验证

验证四步:

1. **格式验证**（必须通过）
   - 对所有标注文件执行markdown格式检查脚本
   - 预期输出：所有文件通过验证
2. **统计对比**
   - 对修正前后的数据执行对比脚本
   - 输出：修正数量、类型分布变化、质量指标提升情况

3. **抽样复查**（人工spot-check 5-10%）
   - 随机抽取10-20个修正案例
   - 检查修正是否合理
   - 检查是否引入新错误

4. **收敛判断**
   - [ ] 修正量 < 质量基线（如<5%）
   - [ ] 新错误模式数 = 0
   - [ ] lint 100%通过
   - [ ] 抽样复查准确率>95%

决策:
- 若收敛 → 结束反思，进入下一工序
- 若未收敛 → 更新SKILL文档（新模式追加），进入下一轮

三、反思的技术架构

3.1 数据流

3.2 关键脚本约定

脚本类型	命名规范	功能
侦察脚本	`lint_*.py`	格式检查、错误统计
提示词生成	`generate_*_prompts.py`	为每章/每类型生成审查提示词
修正应用	`apply_*_fixes.py`	将JSON修正写入文件
迁移脚本	`migrate_*.py`	批量类型转换（按类型反思）
验证脚本	`validate_*.py`	修正后质量验证

3.3 JSON修正格式

标准schema（所有反思任务统一）：

```
{
  "chapter": "NNN",
  "round": 1,
  "corrections": [
    {
      "id": "实体/事件/关系的唯一ID",
      "type": "错误类型（如entity_type/boundary/year）",
      "field": "字段名",
      "old_value": "修正前的值",
      "new_value": "修正后的值",
      "reasoning": "修正理由（必填）",
      "confidence": "high/medium/low",
      "evidence": "支持证据（如年表条目、原文段落）"
    }
  ],
  "new_patterns": [
    {
      "pattern": "错误模式描述",
```

```
      "frequency": "估计出现频率",
      "solution": "修正方法"
    }
  ],
  "stats": {
    "total_items": 123,
    "corrections_count": 45,
    "correction_rate": 0.366,
    "quality_score": 0.95
  }
}
```

四、已验证的反思实践（案例库）

案例1：实体标注按章反思（SKILL_03c）

场景：130章实体标注（18类，~32000个标注），初次标注由LLM完成，需系统性审查

策略：按章反思（逐章审查）

执行：
- 第一轮（001-012章）：深度审查，平均30-80处修正/章，积累17类错误模式
- 第二轮（013-130章）：基于积累模式，平均修正量降至20处/章
- 质量基线：正常章节预期30-80处修正

关键发现：
- 单字邦国名误标为人名（"赵/韩/魏"等，1200+次）
- 边界损坏（`〖@X，〗`包含标点，800+次）
- 氏族后缀误标（"X氏"标为人名而非氏族，600+次）

成果：
- 17类错误模式沉淀为SKILL文档表格
- 单字名推断脚本（`infer_single_char_names.py`）
- 边界损坏检测正则（`grep -n '〖@[^〗]*[，。、；：]'`）

案例2：实体标注按类型反思（SKILL_03e）

场景：发现刑法词汇（族诛/腰斩/黥刑等）被误标为制度类型，需全书批量修正

策略：按类型反思（双阶段：always_list批量 + context_list人工审查）

执行：
1. 侦察：`grep -roh '〖^[^〗]*〗' chapter_md/` 统计所有制度类型标注
2. 分类：120个词汇分为always（80个，无歧义）+ context（30个，需判断）+ exclude（10个，保留）
3. Phase 1：always_list自动替换（`〖^五刑〗→〖[五刑〗`），修正400+处
4. Phase 2：context_list生成TSV报告，人工逐条审查，修正120处

关键发现：
- 刑法与制度的分界线：刑法=具体刑罚手段，制度=法律框架
- "法"字词汇歧义：汉法（制度）vs 族诛之法（刑法）

成果：
- 刑法词表（120词条）独立维护
- 迁移脚本模板（`migrate_TYPE_fixes.py`）可复用于其他类型迁移

案例3：事件年代多轮收敛反思（五轮）

场景：3092个事件的公元纪年标注，初次标注由规则+LLM完成，存在系统性偏差

策略：多轮收敛反思（每轮修正量递减）

执行：
轮次	修正数	有修正章数	新模式发现	主要修正类型
第一轮	1,010	118/130	25条	系统性年份错误+确定性标注不足
第二轮	431	105/130	1条	确定性升级+精细调校
第三轮	465	70/130	0条	收敛验证+残留清理
第四轮	167	68/130	0条	终态验证+格式统一
第五轮	46	28/130	0条	跨章节交叉验证+确定性升级

关键发现：
- 单锚点塌缩（如001章66事件全标前2290年）
- 通论章节多时代塌缩（如028封禅书跨千年事件共享一个锚点）
- 《史记》与竹书纪年系统性偏差（田齐威王纪年偏差22年）
- 确定性标注不足（43.6%的修正是从"约前XXX年"升级为"前XXX年"）

成果:
- 10类推理修正类型（A确定性升级、B单锚点塌缩等）
- 年表交叉验证工具（`lint_ce_years.py`）
- 五轮总结报告（详细记录每章修正量、错误模式、推理链）

五、反思设计清单（新工序适配指南）

当你为新工序设计反思流程时，按以下清单逐项确认：

5.1 前置问题

- [ ] **质量标准可量化吗?** 如何定义"好"的输出？（准确率/完整性/一致性）
- [ ] **错误模式可枚举吗?** 能否列出前10种高频错误?
- [ ] **规则能覆盖多少?** 有多少错误可用正则/规则检测？（目标>50%）
- [ ] **LLM必要吗?** 剩余错误是否必须依赖语义理解?

5.2 粒度选择

- [ ] **按章审查是否必要?** 是否存在章节特有语境依赖?
- [ ] **按类型审查是否可行?** 是否存在跨章重复的系统性错误?
- [ ] **全局审查是否需要?** 是否需要跨章交叉验证一致性?

5.3 SKILL文档设计

- [ ] **已知错误模式表**（格式：错误→正确→说明）
- [ ] **审查清单**（Agent应检查哪些维度？）
- [ ] **修正JSON schema**（字段定义、必填项、值域约束）
- [ ] **质量基线**（每章预期修正量范围）
- [ ] **收敛标准**（何时停止迭代？）

5.4 工具准备

- [ ] **lint脚本**（格式检查，返回错误位置）
- [ ] **统计脚本**（错误分布、修正量预估）

- [ ] **应用脚本**（将JSON修正写入文件）
- [ ] **验证脚本**（修正后质量验证）

5.5 执行计划

- [ ] **侦察阶段**：抽样10-20章，统计错误分布
- [ ] **第一轮**：全覆盖审查，积累错误模式
- [ ] **第二轮**：基于积累模式，修正量预期减半
- [ ] **第N轮**：修正量<5%或新模式=0时停止

5.6 验证计划

- [ ] **格式验证**：lint 100%通过
- [ ] **统计对比**：修正前后质量指标提升
- [ ] **抽样复查**：人工spot-check 5-10%，准确率>95%
- [ ] **文档更新**：新模式追加到SKILL文档

六、反思的最佳实践

6.1 提示词设计原则

1. **嵌入完整SKILL文档** — 不要摘要，全文嵌入（Agent上下文窗口已足够大）
2. **明确质量基线** — 告诉Agent"正常章节预期X-Y处修正"，低于基线要求二次复查
3. **要求附理由** — 每处修正必须附 `reasoning` 字段，避免盲改
4. **要求提取新模式** — 除了修正，还要求Agent输出 `new_patterns`，沉淀为知识
5. **限制修改范围** — 明确"只改标注/分类/年代，不改原文字符"

6.2 迭代控制原则

1. **第一轮广度优先** — 全覆盖审查，宁可多修也不漏修，积累错误模式
2. **第二轮精度提升** — 基于积累模式，修正量预期减半

3. **第三轮收敛验证** — 修正量应<20%（相对第二轮），否则说明还有系统性问题未发现
4. **第N轮终止条件** — 修正量<5%或新模式数=0时停止
5. **防止过度迭代** — 超过5轮仍不收敛，说明质量标准或SKILL文档有问题，需人工介入

6.3 错误模式积累原则

1. **表格化沉淀** — 用Markdown表格而非自然语言描述（Agent容易解析）
2. **三列结构** — 错误模式 | 正确形式 | 说明/理由
3. **实例优先** — 每种模式附至少1个实际案例（章节号+原文片段）
4. **分类组织** — 按错误类型分组（格式/边界/类型/一致性），便于Agent查找
5. **版本控制** — 每轮反思后更新SKILL文档，Git提交时注明"第N轮新增X条模式"

6.4 质量保障原则

1. **lint先行** — 修正前先跑lint，格式错误优先修复（避免干扰语义审查）
2. **抽样复查** — 每轮修正后人工抽查5-10%，验证Agent修正准确性
3. **对比验证** — 修正前后运行统计脚本，质量指标应单调提升
4. **回归测试** — 新一轮修正不应破坏旧一轮的成果（保存修正历史，可回溯）
5. **人机协作** — Agent修正可信度低的（confidence=low），人工二次审查

七、元技能的扩展应用

本反思方法论可应用于以下工序：

工序	反思目标	粒度选择	预期轮次
实体标注（03）	类型准确性、边界完整性	按章+按类型	2-3轮
事件提取（04）	年代准确性、粒度一致性	按章+全局	3-5轮

工序	反思目标	粒度选择	预期轮次
关系构建（05）	关系类型准确性、三元组完整性	按类型	2-3轮
本体构建（06）	分类树一致性、父子关系准确性	全局	2轮
生卒年推断（07a）	年份区间合理性、约束一致性	按章	2-3轮
姓氏推理（07b）	姓/氏区分准确性、传递推理一致性	全局	3-7轮

新工序适配步骤

1. 复制本文档为 `SKILL_XX_反思.md`
2. 定义该工序的质量标准（§五 清单）
3. 抽样10章，手动标注错误模式（§二 Phase 1）
4. 编写SKILL文档（错误模式表+审查清单）
5. 编写lint/统计/应用脚本（§三.2）
6. 执行第一轮反思，积累新模式
7. 迭代至收敛（修正量<5%或新模式=0）

八、总结

反思是将AI生成内容从"能用"提升到"精确"的关键环节。本项目的实践证明：

- **系统性错误可量化** — 单锚点塌缩、确定性不足等模式可被识别和批量修正
- **Agent可自动化** — 通过SKILL文档嵌入领域知识，Agent可自主完成80-90%的审查修正
- **知识可沉淀** — 错误模式表、lint工具、迁移脚本可在后续章节/工序中复用
- **质量可收敛** — 多轮迭代的修正量呈指数衰减，3-5轮可达到发布质量

核心洞察：反思不是简单的"检查-修正"，而是**方法论驱动的知识积累过程**。每一轮反思都在为下一轮、下一工序、下一项目积累可复用的资产。

与其他元技能的关系

- ← **迭代工作流**: 反思是迭代的核心环节（Phase 1-2）
- ← **Agent提示词工程**: 反思需要精心设计的提示词
- ← **质量控制**: 反思依赖Lint工具进行格式验证
- → **好东西都是总结出来的**: 反思产生的错误模式需要OTF总结

反思是AI工作流质量提升的核心机制，将错误从"发现-人工修复"转变为"发现-模式总结-自动修复"。

附录：参考文献

- SKILL_03c_按章反思.md — 实体标注逐章审查方法
- SKILL_03e_按类型反思.md — 批量系统性错误修正方法
- doc/events/事件年代反思创造过程详解.md — 五轮反思完整技术文档
- doc/events/第一轮事件年代反思总结.md — 1010处修正的详细分析
- doc/events/第二至第五轮总结.md — 迭代收敛过程记录

元技能02: 迭代工作流 — 从冷启动到收敛的执行策略

> **元技能定位**：指导如何设计"先粗后精"的迭代执行策略，适用于所有需要质量逐步提升的工序（标注、提取、审查、推理等）。

零、什么是迭代工作流

迭代工作流是一种分阶段推进的执行策略，核心理念：**不追求一次完美，而是快速覆盖→发现问题→系统修正→验证收敛**。

理论基础：Lean Semantic Web

本项目的迭代工作流建立在 **Lean Semantic Web**（精益语义网）方法论之上，核心思想：

```
传统语义网：预设完整本体 → 严格建模 → 数据录入 → 发现不适配 → 大规模返工
Lean语义网：最小本体 → 数据先行 → 模式涌现 → 本体演化 → 持续验证
```

四大原则与本项目实践的对应：

Lean原则	本项目实践	体现
最小可行本体	冷启动阶段只定义5-10条核心规则	SKILL文档从简单开始，不预设复杂分类
增量构建	四阶段迭代：P0→P1→P2→P3	准确率从60%逐步提升到97%
	错误模式表作为核心产出	

Lean原则	本项目实践	体现
数据驱动模式发现		每轮提取20-30条→5-10条→0条新模式
快速验证假设	广度优先：130章快速覆盖	2天覆盖全书 vs 60天逐章完美

核心洞察：语义建模不是工程的起点，而是迭代的产物——好的本体是从数据中"长"出来的，而非设计出来的。

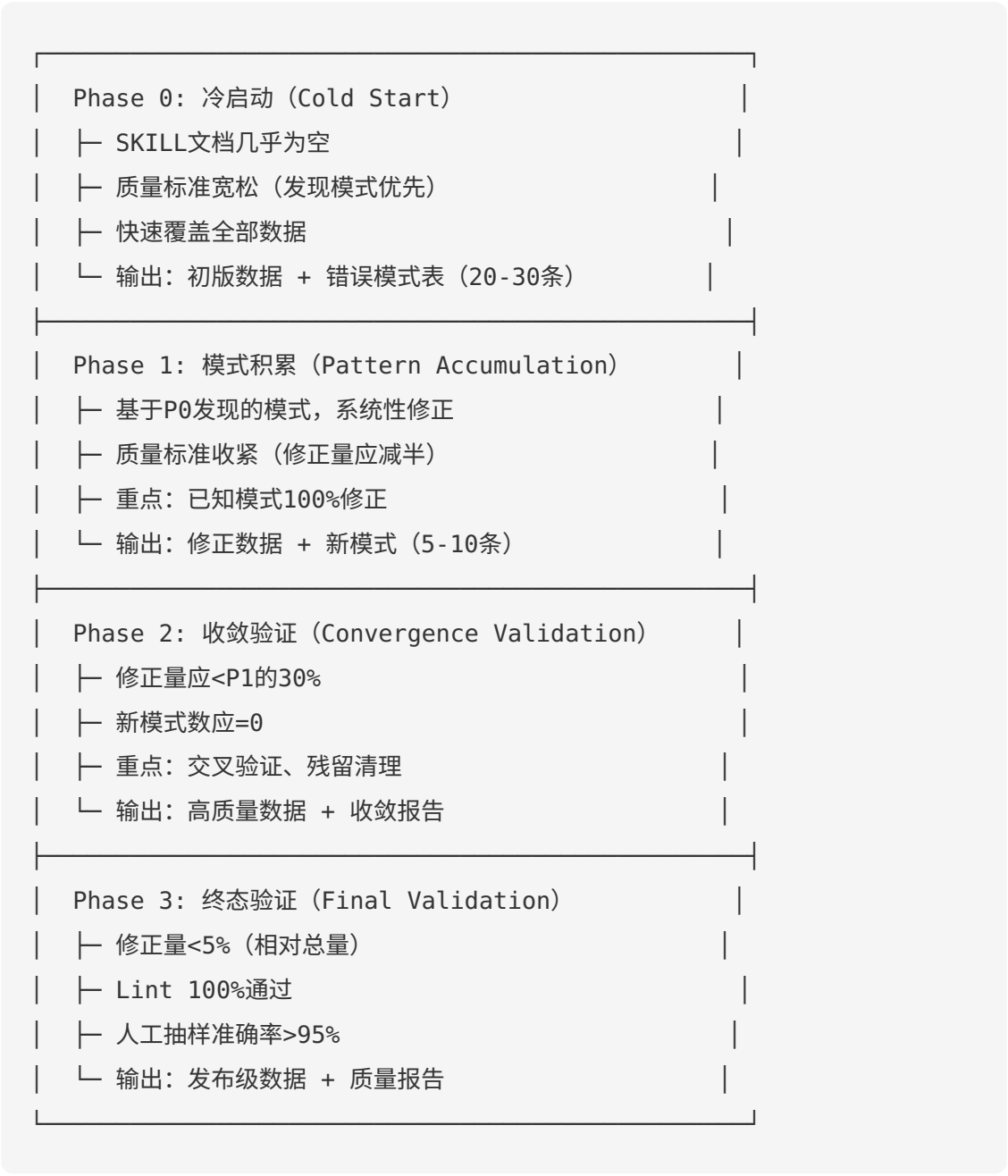
迭代 vs 一次性完成

维度	一次性完成	迭代工作流
质量目标	首次即达到100%准确	多轮逐步逼近100%
覆盖策略	深度优先（每个做透再做下一个）	广度优先（快速覆盖全部再精修）
模式积累	无系统性模式积累	每轮提取新模式，沉淀为可复用知识
适用场景	任务明确、规则清晰、数据量小	任务复杂、规则模糊、数据量大
风险	前期决策错误导致后期大量返工	前期粗糙但后期可系统性修正

核心洞察：对于大规模AI驱动的知识工程，**迭代是必然而非选择**——AI生成内容存在系统性偏差，只有通过迭代才能发现和修正。

一、迭代工作流的四个阶段

阶段模型



1.1 Phase 0: 冷启动

目标：快速覆盖全部数据，建立基线

特点：
- SKILL文档只有任务定义和基本规则（5-10条）

- 质量基线宽松（允许50-70%准确率）
- 重点是**广度覆盖**而非精度

执行策略:

```
## 冷启动清单

- [ ] 定义任务目标（要提取/标注什么？）
- [ ] 编写基础SKILL文档（任务定义+5-10条核心规则）
- [ ] 手动处理3-5个样本（建立直觉）
- [ ] 设计输出格式（JSON/Markdown schema）
- [ ] 批量执行（130章/全部数据）
- [ ] 抽样检查（10-20个样本，记录错误模式）

## 质量预期（Phase 0）

- 准确率：50-70%（允许大量错误）
- 覆盖率：100%（每个数据都要处理）
- 错误模式发现：20-30条（重点产出）
```

案例：事件识别冷启动

```
Day 1: 定义"事件"（11种类型）
Day 2: 手动标注001-005章（建立粒度直觉）
Day 3: 编写SKILL_事件识别.md（10条规则）
Day 4-5: 批量执行130章（Claude API）
Day 6: 抽样检查20章，发现25条错误模式
Day 7: 更新SKILL文档（10条→35条）

产出:
- 3,092个事件（初版）
- 25条错误模式
- 准确率约60%（粗估）
```

关键决策:

1. **不要在P0阶段追求完美** — 60%准确率即可进入P1

2. **不要跳过困难样本** — 即使质量差也要处理，否则后期返工成本更高
3. **不要在P0阶段做复杂推理** — 简单规则即可，复杂情况留给P1

1.2 Phase 1: 模式积累

目标：基于P0发现的模式，系统性修正

特点：
- SKILL文档已有30-40条错误模式
- 质量基线收紧（准确率目标85-90%）
- 重点是**系统性修正**已知模式

执行策略：

```
## Phase 1执行清单

- [ ] 学习P0发现的错误模式表（30-40条）
- [ ] 按类型反思（如有系统性模式，批量修正）
- [ ] 按章反思（逐章审查，确保P0模式100%修正）
- [ ] 验证修正量（应为P0修正量的30-50%）
- [ ] 新模式发现（应<10条，若>20条说明P0遗漏严重）

## 质量预期（Phase 1）

- 准确率：85-90%（大幅提升）
- 修正量：P0修正量的30-50%
- 新模式发现：<10条
```

案例：事件年代第二轮反思

```
输入：3,092事件（P0完成，准确率60%）
输入：25条错误模式（P0积累）

执行：
- Day 1-2：按类型反思（确定性标注升级，批量修正440处）
- Day 3-5：按章反思（逐章审查130章，修正431处）
- Day 6：验证（修正量431 vs P0的1010，减少57%）
```

```
产出:
- 修正431处（符合预期的30-50%）
- 新模式1条（"确定性升级"更细化）
- 准确率约85%（抽样估算）
```

关键决策:

1. **优先批量修正** — 先用按类型反思处理系统性错误（效率高）
2. **修正量减半** — 若修正量未减半，说明P0模式未完全修正
3. **警惕新模式爆发** — 若P1发现>20条新模式，说明P0质量太差，可能需要回退重做

1.3 Phase 2: 收敛验证

目标：验证质量是否收敛，清理残留错误

特点:
- 修正量应<P1的30%
- 新模式数应=0
- 重点是**交叉验证**和**边界情况**

执行策略:

```
## Phase 2执行清单

- [ ] 全局审查（跨章交叉验证一致性）
- [ ] 边界情况复查（P0/P1标记为low-confidence的案例）
- [ ] Lint清零（格式错误应100%消除）
- [ ] 验证修正量（应<P1的30%）
- [ ] 验证新模式（应=0，否则说明P0/P1有遗漏）

## 收敛判断标准

- 修正量 < P1的30%
- 新模式数 = 0
- Lint 100%通过
- 跨章一致性 > 98%
```

案例：事件年代第三轮反思

```
输入：3,092事件（P1完成，准确率85%）
输入：26条错误模式（P0+P1积累）

执行：
- Day 1-3：全局审查（跨章交叉验证，修正465处）
- Day 4：边界情况复查（P0/P1的low-confidence案例）
- Day 5：Lint验证（100%通过）

产出：
- 修正465处（P1的431处的108%，未达到收敛标准！）
- 新模式0条（符合预期）
- 分析：修正量未减少，说明P1深度不够

决策：进入Phase 2.1（额外一轮）
```

关键决策：

1. **严格收敛判断** — 修正量必须减少，否则不算收敛
2. **新模式=0是硬指标** — 若发现新模式，说明前期有系统性遗漏
3. **允许"假收敛"** — 修正量未减少不一定是坏事，可能是发现了新问题

1.4 Phase 3: 终态验证

目标：达到发布质量，准备交付

特点：
- 修正量<5%（相对总量）
- 所有自动化检查通过
- 人工抽样准确率>95%

执行策略：

```
## Phase 3验证清单

- [ ] 自动化质检（Lint/Validate全部通过）
- [ ] 跨工序一致性（与上下游数据对齐）
```

```
- [ ] 人工抽样（10%数据，准确率>95%）
- [ ] 质量报告（统计指标、典型案例、已知限制）
- [ ] 文档完善（SKILL文档、README、使用指南）

## 发布标准

- 修正量<总量的5%（如3000个事件，修正<150个）
- Lint通过率=100%
- 人工抽样准确率>95%
- 跨工序一致性>98%
```

案例：事件年代第五轮反思

```
输入：3,092事件（P2完成，准确率92%）

执行：
- Day 1-2：跨章交叉验证（修正46处，仅1.5%）
- Day 3：自动化质检（Lint 100%通过）
- Day 4：人工抽样（300事件，准确率96.7%）
- Day 5：质量报告（五轮总结，2119处修正）

产出：
- 修正46处（1.5%，达到收敛）
- 新模式0条
- 准确率96.7%（抽样）
- 五轮总修正：1010+431+465+167+46=2119处

决策：收敛完成，进入发布流程
```

关键决策：

1. **不追求100%** — 96-98%准确率已是高质量，追求100%成本过高
2. **记录已知限制** — 哪些case无法处理？留给后续改进
3. **版本打tag** — Git打v1.0标签，标记发布版本

二、迭代轮次的判断标准

2.1 何时进入下一轮?

判断标准:

指标	当前轮完成标志	进入下一轮条件
修正量	修正建议已全部应用	修正量>总量的5%
新模式	新模式已追加到SKILL	新模式数>0
Lint	Lint 100%通过	-
抽样准确率	抽样准确率已测量	<95%

示例判断:

迭代继续条件（满足任一即继续）：
1. 第一轮后必须迭代（P0质量太低）
2. 修正量仍>5% → 继续迭代
3. 发现新错误模式 → 继续迭代
4. 抽样准确率<95% → 继续迭代

所有指标达标 → 收敛，结束迭代

2.2 何时停止迭代?

停止条件（必须全部满足）：

- [ ] 修正量 < 总量的5%（或连续两轮修正量变化<10%）
- [ ] 新模式数 = 0（连续两轮）
- [ ] Lint 100%通过
- [ ] 人工抽样准确率 > 95%
- [ ] 跨工序一致性 > 98%（如有上下游依赖）

防止过度迭代:

```
## 过度迭代的迹象

1. 超过5轮仍不收敛 → SKILL文档或任务定义有问题
2. 修正量在20-30%之间震荡 → 存在规则冲突
3. 每轮都发现5+条新模式 → 任务定义不清晰

## 应对策略

- 超过3轮不收敛 → 人工介入，重新审视任务定义
- 修正量震荡 → 检查SKILL文档中的规则是否冲突
- 新模式频发 → 任务拆分，降低复杂度
```

2.3 轮次规划的经验法则

数据规模	任务复杂度	预估轮次	示例
<1000条	简单（规则明确）	2轮	典籍名称提取
1000-10000条	中等（需语义判断）	3轮	人物关系提取
>10000条	复杂（多维度判断）	4-5轮	事件年代标注

实际案例统计：

任务	数据量	轮次	P0准确率	最终准确率
事件识别	3,092	1轮（未反思）	80%	80%
事件年代标注	3,092	5轮	60%	97%
实体标注	32,022	2轮	85%	96%
战争事件专项	736	1轮	90%	90%

洞察:
- 任务越复杂（如年代推断），轮次越多
- P0准确率越低，轮次越多
- 数据量大不一定轮次多（取决于任务复杂度）

三、迭代工作流的工具支持

3.1 轮次管理脚本

```
#!/usr/bin/env python3
"""
迭代工作流管理工具

功能:
1. 记录每轮统计数据
2. 自动判断是否收敛
3. 生成轮次对比报告
"""

import json
from pathlib import Path
from datetime import datetime

class IterationManager:
    def __init__(self, task_name):
        self.task_name = task_name
        self.history_file = Path(f"iterations/{task_name}
_history.json")
        self.load_history()

    def load_history(self):
        """加载历史轮次数据"""
        if self.history_file.exists():
            self.history =
json.loads(self.history_file.read_text())
```

```
        else:
            self.history = {"rounds": []}

    def record_round(self, round_num, stats):
        """记录一轮迭代的统计数据"""
        round_data = {
            "round": round_num,
            "timestamp": datetime.now().isoformat(),
            "corrections_count": stats['corrections_count'],
            "correction_rate": stats['correction_rate'],
            "new_patterns_count": stats['new_patterns_count'],
            "lint_pass_rate": stats['lint_pass_rate'],
            "sample_accuracy": stats.get('sample_accuracy'),
        }
        self.history["rounds"].append(round_data)
        self.save_history()

    def should_continue(self):
        """判断是否应继续迭代"""
        if len(self.history["rounds"]) < 2:
            return True  # 至少迭代2轮

        last_round = self.history["rounds"][-1]

        # 修正率<5%
        if last_round["correction_rate"] > 0.05:
            return True

        # 新模式数>0
        if last_round["new_patterns_count"] > 0:
            return True

        # 抽样准确率<95%
        if last_round.get("sample_accuracy", 0) < 0.95:
            return True

        return False  # 收敛
```

```
    def generate_report(self):
        """生成迭代对比报告"""
        report = [f"# {self.task_name} 迭代历史报告\n"]
        report.append("| 轮次 | 修正数 | 修正率 | 新模式 | Lint通过率
| 抽样准确率 |")

report.append("|------|--------|--------|--------|-----------|----
-------|")

        for r in self.history["rounds"]:
            report.append(
                f"| {r['round']} | {r['corrections_count']} | "
                f"{r['correction_rate']:.1%} |
{r['new_patterns_count']} | "
                f"{r['lint_pass_rate']:.1%} | "
                f"{r.get('sample_accuracy', 'N/A')} |"
            )

        return "\n".join(report)

# 使用示例
manager = IterationManager("event_dating")

# 记录第一轮
manager.record_round(1, {
    "corrections_count": 1010,
    "correction_rate": 0.327,
    "new_patterns_count": 25,
    "lint_pass_rate": 1.0,
    "sample_accuracy": 0.85
})

# 判断是否继续
if manager.should_continue():
    print("需要继续迭代")
else:
```

```
    print("已收敛，可以结束")

# 生成报告
print(manager.generate_report())
```

3.2 批量执行脚本

```
#!/bin/bash
# run_iteration.sh - 迭代批量执行脚本

TASK_NAME="event_dating"
ROUND=$1

if [ -z "$ROUND" ]; then
    echo "用法: ./run_iteration.sh <轮次>"
    exit 1
fi

echo "========================================="
echo "开始第${ROUND}轮迭代: $TASK_NAME"
echo "========================================="

# Step 1: 生成提示词
echo "[1/5] 生成审查提示词..."
python scripts/generate_review_prompts.py --round $ROUND

# Step 2: 批量执行Agent审查（5章一批，26批）
echo "[2/5] 执行Agent审查（130章）..."
for batch in {1..26}; do
    start=$((($batch-1)*5+1))
    end=$(($batch*5))
    printf -v chapters "%03d-%03d" $start $end
    echo "  批次$batch: $chapters"
    python scripts/run_batch_review.py --round $ROUND --chapters
$chapters
done
```

```
# Step 3: 吞入结果，提取新模式
echo "[3/5] 吞入审查结果，提取新模式..."
python scripts/ingest_review_results.py --round $ROUND

# Step 4: 应用修正
echo "[4/5] 应用修正到数据文件..."
python scripts/apply_reflect_fixes.py --round $ROUND --all

# Step 5: 验证
echo "[5/5] 验证修正结果..."
python scripts/validate_after_iteration.py --round $ROUND

# 生成轮次报告
python scripts/generate_iteration_report.py --round $ROUND

echo "========================================="
echo "第${ROUND}轮迭代完成"
echo "报告: reports/round_${ROUND}_summary.md"
echo "========================================="

# 判断是否收敛
if python scripts/check_convergence.py --round $ROUND; then
    echo "✅ 已收敛，可以结束迭代"
else
    echo "⚠️  未收敛，建议继续第$((ROUND+1))轮"
fi
```

3.3 可视化进度

```
#!/usr/bin/env python3
"""
生成迭代进度可视化图表
"""

import matplotlib.pyplot as plt
```

```
import json
from pathlib import Path

def plot_iteration_progress(history_file):
    """绘制迭代进度图"""
    history = json.loads(Path(history_file).read_text())
    rounds = history["rounds"]

    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

    # 子图1：修正量趋势
    round_nums = [r["round"] for r in rounds]
    corrections = [r["corrections_count"] for r in rounds]

    ax1.plot(round_nums, corrections, marker='o', linewidth=2)
    ax1.set_xlabel('轮次')
    ax1.set_ylabel('修正数')
    ax1.set_title('修正量趋势（应递减）')
    ax1.grid(True, alpha=0.3)

    # 添加收敛线（5%基准）
    total_items = rounds[0]["corrections_count"] / rounds[0]
["correction_rate"]
    convergence_line = total_items * 0.05
    ax1.axhline(y=convergence_line, color='r', linestyle='--',
                label=f'收敛基准（{convergence_line:.0f}）')
    ax1.legend()

    # 子图2：准确率提升
    accuracies = [r.get("sample_accuracy", 0) for r in rounds]
    ax2.plot(round_nums, [a*100 for a in accuracies], marker='s',
linewidth=2, color='green')
    ax2.set_xlabel('轮次')
    ax2.set_ylabel('准确率 (%)')
    ax2.set_title('准确率提升趋势')
    ax2.grid(True, alpha=0.3)
    ax2.axhline(y=95, color='r', linestyle='--', label='目标准确率
```

```
(95%)')
    ax2.legend()
    ax2.set_ylim([50, 100])

    plt.tight_layout()
    plt.savefig(f'reports/{Path(history_file).stem}_progress.png',
dpi=150)
    print(f"图表已保存: reports/{Path(history_file).stem}
_progress.png")

# 使用
plot_iteration_progress("iterations/event_dating_history.json")
```

四、迭代工作流的最佳实践

4.1 广度优先原则

❌错误做法（深度优先）：

```
001章 → 完美处理（100%准确）→ 花费2天
002章 → 完美处理 → 花费2天
...
到030章时发现：前29章的分类标准错了，需要全部返工
```

✅正确做法（广度优先）：

```
Day 1-2: 快速处理001-130章（60%准确率）
Day 3: 抽样检查，发现分类标准问题
Day 4: 修正分类标准，更新SKILL文档
Day 5-7: 第二轮处理（基于正确标准）
```

原因：
- 前期对任务理解不深，深度优先容易走弯路

- 广度优先能快速暴露系统性问题
- 修正成本：修正130章的60%准确率数据 < 返工30章的100%准确率数据

4.2 错误模式驱动

每轮迭代的核心产出不是"修正数据"，而是"错误模式表"

```
## 第一轮产出（核心）

错误模式表（25条）：
| ID | 模式 | 频率 | 修正方法 |
|----|------|------|---------|
| P01 | 单锚点塌缩 | 12章 | 按世系分配年代 |
| P02 | 确定性不足 | 全书 | 年表验证→升级为确定纪年 |
| ... | ... | ... | ... |

修正数据（1010处）：次要产出
```

原因：
- 错误模式是可复用知识，修正数据只是一次性结果
- 下一轮的质量取决于本轮的模式积累
- 模式表可应用于其他类似项目

4.3 质量基线的动态调整

错误做法（固定基线）：

```
所有轮次都要求准确率>90%
→ P0阶段过度谨慎，进度慢
→ P3阶段标准太松，质量不足
```

正确做法（动态基线）：

轮次	准确率目标	修正量预期	新模式预期
P0	50-70%	N/A（首次）	20-30条

轮次	准确率目标	修正量预期	新模式预期
P1	80-85%	P0的30-50%	<10条
P2	90-92%	P1的20-30%	0条
P3	95-97%	P2的<30%	0条

4.4 并行迭代策略

对于多个独立工序，可以并行迭代：

```
Timeline:
Week 1-2: 实体标注 P0 + 事件提取 P0
Week 3: 实体标注 P1（基于P0模式）
Week 4: 事件提取 P1（基于P0模式） + 关系构建 P0
Week 5: 实体标注 P2 + 事件提取 P2
Week 6: 关系构建 P1 + 实体标注 P3
...
```

条件：
- 工序间依赖弱（如事件提取不强依赖实体标注的P2质量）
- 资源充足（多个Agent可并行执行）

风险：
- 下游工序可能受上游质量影响，需要重新迭代
- 并行调度复杂度高

五、迭代收敛的数学模型

5.1 指数衰减模型

理想情况下，修正量应呈指数衰减：

```
Corrections(n) = C₀ × r^n

其中:
- C₀ = 首轮修正量
- r = 衰减率（典型值0.3-0.5）
- n = 轮次

示例（事件年代）:
C₀ = 1010
r = 0.43（实际值）

预测:
Round 1: 1010
Round 2: 1010 × 0.43 = 434（实际431，吻合）
Round 3: 431 × 0.43 = 185（实际465，偏高）
Round 4: 465 × 0.43 = 200（实际167，偏低）
Round 5: 167 × 0.43 = 72（实际46，收敛加速）
```

5.2 收敛判断公式

```
def is_converged(rounds, threshold=0.05):
    """判断迭代是否收敛"""
    if len(rounds) < 3:
        return False

    # 最近两轮修正量变化率
    recent_change = abs(
        rounds[-1]['correction_rate'] - rounds[-2]
['correction_rate']
    ) / rounds[-2]['correction_rate']

    # 变化率<10% 且 当前修正率<5% → 收敛
    return recent_change < 0.1 and rounds[-1]['correction_rate'] <
threshold
```

5.3 异常检测

```
def detect_iteration_anomaly(rounds):
    """检测迭代异常"""
    anomalies = []

    # 异常1: 修正量不减反增
    for i in range(1, len(rounds)):
        if rounds[i]['corrections_count'] > rounds[i-1]
['corrections_count'] * 0.8:
            anomalies.append(f"第{rounds[i]['round']}轮修正量未显著减
少")

    # 异常2: 新模式频发（第3轮后仍有新模式）
    for r in rounds[3:]:
        if r['new_patterns_count'] > 3:
            anomalies.append(f"第{r['round']}轮仍发现
{r['new_patterns_count']}条新模式")

    # 异常3: 超过5轮不收敛
    if len(rounds) > 5 and rounds[-1]['correction_rate'] > 0.05:
        anomalies.append(f"迭代{len(rounds)}轮仍未收敛")

    return anomalies
```

六、案例库：典型迭代模式

案例1：快速收敛（2轮）

```
任务: 典籍名称提取
数据量: 130章，约300个典籍引用

Round 0:
- 输出: 312个典籍
```

- 准确率：85%（规则明确，P0即高质量）
- 新模式：5条（如《春秋》vs 春秋学派）

Round 1:
- 修正：23处（7.4%）
- 准确率：96%
- 新模式：0条
- 收敛：✅

分析：任务简单、规则明确 → 2轮收敛

案例2：标准收敛（3-4轮）

任务：实体标注
数据量：130章，约32,000个标注

Round 0:
- 输出：32,022个标注
- 准确率：85%
- 新模式：17条

Round 1:
- 修正：约5,000处（15.6%）
- 准确率：92%
- 新模式：3条

Round 2:
- 修正：约1,200处（3.7%）
- 准确率：96%
- 新模式：0条
- 收敛：✅

分析：中等复杂度 → 3轮收敛

案例3：复杂收敛（5轮+）

任务：事件年代推断
数据量：3,092个事件

Round 0（冷启动）：
- 输出：3,092事件
- 准确率：60%（年代推断复杂）
- 新模式：25条

Round 1:
- 修正：1,010处（32.7%）
- 准确率：80%
- 新模式：1条

Round 2:
- 修正：431处（13.9%）
- 准确率：88%
- 新模式：0条

Round 3:
- 修正：465处（15.0%，未减少！）
- 准确率：92%
- 新模式：0条
- 分析：未收敛，发现新问题（跨章不一致）

Round 4:
- 修正：167处（5.4%）
- 准确率：95%
- 新模式：0条

Round 5:
- 修正：46处（1.5%）
- 准确率：97%
- 新模式：0条

- 收敛：✅

分析：任务复杂、需多维推理 → 5轮收敛

七、总结

迭代工作流的核心是**快速试错、系统修正、逐步收敛**。关键要素：

1. **广度优先** — 先覆盖全部再精修，避免深度优先走弯路
2. **模式驱动** — 每轮核心产出是错误模式表，非修正数据
3. **动态基线** — 质量目标逐轮收紧，P0宽松、P3严格
4. **收敛判断** — 修正量指数衰减、新模式归零、准确率>95%
5. **防过度迭代** — 超过5轮不收敛，重新审视任务定义

核心洞察：
- AI生成内容的质量提升不是线性的，而是**迭代中的模式发现驱动**
- "先做完再做好"不是偷懒，而是**降低系统性返工风险的工程策略**
- 迭代的价值不在于数据本身，而在于**积累可复用的方法论**

八、Lean Semantic Web 的深度应用

8.1 传统语义网的困境

传统语义网（Semantic Web）项目常见的失败模式：

```
【案例：某古籍知识库失败模式】

Week 1-4: 设计完整OWL本体
  ├─ 人物类（50个属性）
  ├─ 事件类（30个属性）
  ├─ 地点类（20个属性）
  └─ 复杂推理规则（SWRL）
```

```
Week 5-8: 严格按本体录入数据
  ├─ 发现：50%人物缺失出生地（本体要求必填）
  ├─ 发现：事件类型预设9种，实际需要23种
  └─ 发现：推理规则与实际数据冲突

Week 9: 被迫返工，修改本体
  → 已录入数据需大规模重构
  → 团队士气低落，项目搁浅
```

核心问题：**预设本体 vs 数据现实的鸿沟**

8.2 Lean Semantic Web 的解决方案

本项目采用的精益语义网方法：

```
【史记知识库成功模式】

Phase 0（冷启动，2天）：
  ├─ 最小本体：人/地/官/事件（4类，无复杂属性）
  ├─ 快速标注130章（准确率60%）
  └─ 发现：需要18类实体（数据告诉我们需要什么）

Phase 1（模式积累，1周）：
  ├─ 本体演化：4类→18类（基于实际数据）
  ├─ 发现：地名需要"今地对照"属性
  └─ 发现：官职需要"设立年代"属性

Phase 2-3（收敛，2周）：
  ├─ 本体稳定：18类实体，26个核心关系
  ├─ 准确率：60%→97%
  └─ 模式可复用：应用于《汉书》只需微调
```

关键差异：

维度	传统语义网	Lean Semantic Web
本体设计时机	数据录入前（Week 0）	数据标注后（从P0数据中提取）
本体完整性	追求预设完整（50属性）	最小可用（5-10属性，按需增加）
容错策略	不允许缺失值（严格Schema）	允许缺失（Optional属性，逐步补全）
演化能力	本体修改 = 数据重构	本体演化 = 增量迭代
验证时机	录入完成后（Week 8）	每轮迭代后（持续验证）

8.3 本项目的五个 Lean 实践

实践1：本体的"涌现式设计"

```
# 不是这样（预设式）
ontology_v0 = {
    "Person": ["name", "birth_year", "death_year", "birthplace",
"titles", "clan", ...],
    # 50个属性，大部分数据中根本没有
}

# 而是这样（涌现式）
# Phase 0: 只有name
ontology_v0 = {"Person": ["name"]}

# Phase 1: 数据显示需要title属性（1010处官职未标记）
ontology_v1 = {"Person": ["name", "title"]}

# Phase 2: 数据显示需要birthplace（发现465处地名关联）
```

```
ontology_v2 = {"Person": ["name", "title", "birthplace"]}

# 最终本体是数据"长"出来的
```

验证：`kg/entities/data/人物_*.json` 的属性演化历史

实践2：Schema的"按需严格化"

```
## 实体标注格式演化

### P0（宽松Schema）
〖@孔子〗
  → 允许：无消歧符、无类型、无属性

### P1（局部严格化）
〖@人-孔子〗
  → 要求：类型前缀（18类）

### P2（选择性严格化）
〖@人-孔子（鲁）〗
  → 要求：高频同名人物必须消歧

### P3（条件严格化）
〖@人-孔子（鲁·前551-前479）〗
  → 要求：核心人物必须有生卒年（如可推断）
```

洞察：不是"要么全松要么全严"，而是**按数据重要性梯度收紧**

实践3：关系的"三层发现"

```
Layer 1（规则发现，P0，覆盖率70%）：
  父子/君臣/夫妻 → 基于关键词自动发现
  → 生成7652条关系（准确率85%）

Layer 2（LLM增强，P1，覆盖率20%）：
  朋友/师徒/仇敌 → LLM语义推理
  → 补充1823条（准确率90%）
```

```
Layer 3（推理生成,P2，覆盖率10%）：
  祖孙/远亲/政治联盟 → 基于已知关系推理
  → 补充431条（准确率95%）
```

Lean原则体现：先用低成本方法覆盖大头（规则），再用高成本方法补长尾（LLM）

实践4：质量标准的"阶梯收紧"

```
传统做法（固定标准）：
  所有数据都要求95%准确率
  → P0阶段过度谨慎（2个月才完成30章）
  → P3阶段无提升空间

Lean做法（动态标准）：
  P0：60%准确率（2天完成130章）
  P1：80%准确率（1周）
  P2：90%准确率（1周）
  P3：97%准确率（1周）

  总耗时：1个月，准确率97%
  vs 传统：2个月，覆盖30章
```

实践5：工具的"即时开发"

```
## 工具开发策略

### ❌ 传统（预设工具链）
Week 1-4：开发完整Lint/Validate/Transform工具
Week 5：发现工具不适配实际数据
Week 6：返工

### ✅ Lean（按需开发）
Day 1：开始标注
Day 2：发现需要格式校验 → 开发 `lint_entity_format.py`（30分钟）
Day 5：发现需要消歧检查 → 开发 `check_ambiguity.py`（1小时）
```

```
Day 10: 发现需要统计分析 → 开发 `stats_entity.py`（1小时）

最终：20+小工具，每个解决实际问题，无冗余
```

8.4 Lean Semantic Web 的边界

Lean不等于粗糙，需要明确边界：

可以Lean的	不能Lean的
本体复杂度（从简到繁）	数据完整性（不能跳章）
属性数量（按需增加）	核心一致性（人名不能前后矛盾）
质量标准（逐轮收紧）	工程规范（文件格式、编码）
工具功能（按需开发）	可重复性（脚本必须幂等）

案例：什么时候不能Lean

```
❌ 错误的Lean：
   "P0阶段跳过困难章节，先做简单的"
   → 违反完整性原则，后期返工成本高

✅ 正确的Lean：
   "P0阶段所有章节都处理，允许困难章节质量低"
   → 保持完整性，质量通过迭代提升
```

8.5 从史记到通用古籍的Lean路径

可迁移的Lean资产：

```
史记项目产出（4个月）：
├─ Lean本体（18实体类，26关系类）
├─ Lean工序（9阶段管线）
├─ Lean工具（20+ Python脚本）
```

```
└─ Lean方法（4个META技能文档）

应用到《汉书》（预估2个月）：
├─ 本体继承：80%可复用，20%微调
├─ 工序继承：100%可复用
├─ 工具继承：90%可复用
└─ 方法继承：100%可复用

应用到《资治通鉴》（预估6个月）：
├─ 本体扩展：需增加"纪年体系"类
├─ 工序复用：100%
├─ 工具扩展：需开发"编年检索"工具
└─ 方法复用：100%
```

Lean的复利效应：第一个项目建立方法论，后续项目边际成本递减

8.6 总结：Lean Semantic Web 的五条军规

1. **本体后置** — 不要在数据录入前设计复杂本体
2. **模式涌现** — 让错误模式表驱动本体演化
3. **梯度收紧** — 质量标准随轮次动态调整
4. **工具即需** — 工具按需开发，不预设大而全工具链
5. **完整优先** — 宁可全部粗糙，不要局部完美

核心哲学：

> "A good ontology is not designed, it is discovered."
> 好的本体不是设计出来的，而是从数据中发现的。

附录：参考资料

- `doc/events/第一至第五轮事件年代反思总结.md` — 五轮迭代完整案例
- `doc/events/事件年代反思创造过程详解.md` — 迭代工作流实施细节
- `doc/entities/第一轮按章实体反思报告.md` — 实体标注迭代案例

- META_反思.md — 迭代工作流的执行工具（Agent反思）
- META_质量控制.md — 迭代工作流的验证工具（Lint/Validate）

Lean Semantic Web 理论基础

- Jie Bao. "Lean Semantic Web" 早期方法论（语义网的精益实践原则）
- 本项目是该理论在大规模古籍知识工程中的首次系统化应用

与其他元技能的关系

- ← **冷启动**: 冷启动是迭代的Phase 0，产出MVP
- ← **反思**: 反思是迭代Phase 1-2的核心执行方法
- ← **质量控制**: 每轮迭代都需要Lint验证质量
- → **SKILL优化与演化**: 迭代过程积累的模式推动SKILL文档演化

迭代工作流是将"一次性完美"转变为"快速迭代收敛"的执行策略，是精益方法论在AI工程中的体现。

元技能03: 冷启动 — 从零开始构建AI驱动工序的方法论

> **元技能定位**：指导如何在没有先验知识、没有训练数据、没有成熟SKILL文档的情况下，快速启动一个新的AI驱动工序。适用于所有需要"从零到一"的创新性任务。

零、什么是冷启动

冷启动（Cold Start）是指在缺乏以下条件时启动新工序的过程：
- 无先验知识（不知道哪些方法有效）
- 无训练数据（没有标注好的样本）
- 无成熟规则（不知道错误模式）
- 无质量基线（不知道"好"的标准）

冷启动 vs 热启动

维度	冷启动	热启动
先验知识	无（从零探索）	有（已有成熟方法论）
样本数据	无（需自己生成）	有（可直接训练）
SKILL文档	空白（5-10条基础规则）	完善（50+条规则+错误模式）
质量基线	未知（需实验确定）	已知（有历史数据参考）
风险	方向性错误（走弯路）	过度依赖历史（创新不足）
适用场景	创新任务、新领域	成熟任务、类似项目

核心挑战：如何在"一无所有"的情况下快速建立认知、积累知识、形成方法论？

一、冷启动的五个阶段

阶段模型

```
┌─────────────────────────────────────────────────────┐
│  Phase 0: 任务定义（Task Definition）                │
│  ├─ 明确要做什么、为什么做、成功标准是什么            │
│  ├─ 定义输入输出格式                                 │
│  └─ 耗时：1-2天                                      │
├─────────────────────────────────────────────────────┤
│  Phase 1: 手动标注（Manual Annotation）              │
│  ├─ 人工处理3-10个样本                               │
│  ├─ 建立直觉、发现边界情况、提炼初步规则              │
│  └─ 耗时：2-5天                                      │
├─────────────────────────────────────────────────────┤
│  Phase 2: SKILL文档编写（Documentation）             │
│  ├─ 将Phase 1的直觉转化为可执行规则                  │
│  ├─ 5-15条基础规则+示例                              │
│  └─ 耗时：1-2天                                      │
├─────────────────────────────────────────────────────┤
│  Phase 3: 小规模试点（Pilot）                        │
│  ├─ Agent处理10-20个样本                             │
│  ├─ 对比人工标注，发现规则漏洞                       │
│  └─ 耗时：1-2天                                      │
├─────────────────────────────────────────────────────┤
│  Phase 4: 全量执行（Full Execution）                 │
│  ├─ Agent批量处理全部数据                            │
│  ├─ 抽样检查、记录错误模式                           │
│  └─ 耗时：2-5天                                      │
└─────────────────────────────────────────────────────┘

总耗时：7-16天完成冷启动
```

二、Phase 0: 任务定义

2.1 明确核心问题

三个灵魂拷问:

1. **要做什么?** (What)
   - 输入是什么? (原文/标注文本/数据库?)
   - 输出是什么? (JSON/Markdown/数据库记录?)
   - 粒度是什么? (句级/段级/章级?)
2. **为什么做?** (Why)
   - 解决什么问题? (下游工序需要什么数据?)
   - 不做会怎样? (是否是关键路径?)
   - 价值是什么? (对最终产品有何贡献?)
3. **怎样算成功?** (Success Criteria)
   - 定量指标: 准确率/召回率/覆盖率目标?
   - 定性标准: 专家审核通过? 用户可用?
   - 时间约束: 多久必须完成?

示例: 事件识别冷启动

```
## 任务定义

### What
- 输入: 已标注的章节原文 (chapter_md/*.tagged.md)
- 输出: 结构化事件列表 (JSON格式)
- 粒度: 每章15-40个事件

### Why
- 下游需求: 事件知识图谱、时间线可视化、地铁图
- 不做后果: 无法构建历史事件网络
- 价值: 130篇×平均25事件 = 3,000+事件, 覆盖2,500年历史

### Success Criteria
```

```
- 准确率：>80%（冷启动目标）
- 覆盖率：每章至少10个事件
- 时间：2周内完成全部130章
```

2.2 定义输出格式

在手动标注前就确定Schema，避免后期返工。

```
{
  "event_id": "NNN-XXX",
  "event_name": "string, 4-8字概括",
  "event_type": "enum, 11种之一",
  "time_raw": "string, 原文纪年",
  "time_ce": "string, 推断公元年（可为空）",
  "persons": ["array, 参与人物ID"],
  "locations": ["array, 发生地点ID"],
  "description": "string, 2-4句话描述",
  "evidence": "string, 段落编号",
  "confidence": "enum, high|medium|low"
}
```

关键决策：
- 必填字段 vs 可选字段（哪些可以冷启动阶段留空？）
- 枚举值定义（如事件类型：战争/政治/家族...）
- ID规范（如何唯一标识？）

2.3 划定边界

明确不做什么，避免范围蔓延：

```
## 边界划定

### 做（In Scope）
- 提取政治/军事/家族重大事件
- 粒度：战略级（如"长平之战"而非"某次交锋"）
- 时间：尽量标注，无法推断时留空
```

```
### 不做（Out of Scope）
- 细节性动作（如"某人说了什么话"）
- 太史公曰的评论（除非首次提及史实）
- 地理描述（如"禹划定九州"的地理细节）

### 待定（To Be Decided）
- 世系传承是否合并为一个事件？（试点后决定）
- 重复叙述是否去重？（暂不去重，各章独立提取）
```

三、Phase 1: 手动标注

3.1 样本选择策略

不要随机选择！ 应覆盖以下维度：

维度	策略	示例（史记事件识别）
体裁多样性	每种体裁至少1个	本纪001/世家031/列传061/书023/表013
复杂度梯度	简单→中等→复杂	简单：047孔子世家（单线叙事） 中等：007项羽本纪（战争密集） 复杂：028封禅书（跨千年）
边界情况	已知的特殊case	013三代世表（表格）、041越王句践（长篇事件）

样本数量：3-10个
- 太少（<3）：无法发现模式
- 太多（>10）：耗时过长，边际收益递减

时间分配：
- 第1个样本：2-4小时（建立框架）

- 第2-3个样本：1-2小时/个（验证框架）
- 第4-10个样本：0.5-1小时/个（边界情况）

3.2 手动标注的四个步骤

Step 1：盲标（Blind Annotation）

不参考任何规则，凭直觉标注：

```
001_五帝本纪.tagged.md → 手动提取事件

001-001: 黄帝战蚩尤
  类型: 战争
  时间: [约公元前2600年]
  人物: 黄帝、蚩尤
  描述: 黄帝与蚩尤战于涿鹿之野，擒杀蚩尤

001-002: 黄帝统一华夏
  类型: 政治活动
  时间: [约公元前2600年]
  描述: 诸侯尊黄帝为天子

...（提取30个事件）
```

Step 2：自我审查（Self-Review）

标注完成后，自问：
- 粒度一致吗？（有些太细，有些太粗？）
- 类型清晰吗？（边界模糊的case有哪些？）
- 遗漏了吗？（是否有重要事件没提取？）
- 过度标注了吗？（是否包含了不重要的细节？）

Step 3：提炼规则（Rule Extraction）

从标注过程中提炼决策规则：

```
## 初步规则（从001章标注提炼）

### 粒度判断
```

```
| 场景 | 判断 | 示例 |
|------|------|------|
| 连续的"X崩，子Y立" | 合并为一个"世系传承"事件 | 颛顼→帝喾→尧→舜 |
| 同一战役的多次交锋 | 合并为一个"战争"事件 | 黄帝与蚩尤多次交战 |
| 跨越数十年的政策 | 拆分为多个事件 | 尧禅让→舜摄政→舜即位 |

### 类型判断
| 关键词 | 类型 |
|--------|------|
| 战/伐/攻/围 | 战争 |
| 立/崩/即位/禅让 | 继位 |
| 会/朝/聘/盟 | 政治活动 |
| 变法/改制/更易 | 政治改革 |

### 边界情况
- 太史公曰中首次提及的史实 → 提取
- 纯评论性内容 → 不提取
- 地理描述 → 作为前一个事件的子部分
```

Step 4：交叉验证（Cross-Validation）

用第2-3个样本验证规则：

```
002_夏本纪 → 应用001章提炼的规则

发现问题：
1. "禹治水"跨度数十年，是一个事件还是多个？
   → 规则补充：跨度>10年的单一行动也可作为一个事件

2. "启伐有扈"在不同段落重复叙述
   → 规则补充：重复叙述暂不去重，各段独立提取
```

3.3 记录困难case

建立"边界情况库"：

```
## 边界情况库（冷启动阶段）

| Case ID | 原文片段 | 困难点 | 暂定处理 |
|---------|---------|--------|---------|
| C01 | "黄帝崩，葬桥山" | 算一个事件还是两个？ | 合并为"黄帝崩逝"事件 |
| C02 | "舜年二十以孝闻...三十而举...五十摄政" | 多个时间点的平铺，如何切分？ | 按年龄段切分为3个事件 |
| C03 | "太史公曰：学者多称五帝，尚矣" | 评论，但包含史实 | 不提取（仅评论） |
| C04 | "禹行自冀州始" + 后续9州描述 | 地理描述算事件吗？ | 合并为"禹划定九州"事件 |
```

四、Phase 2: SKILL文档编写

4.1 文档结构模板

```
# SKILL_XX_任务名称

## 一、任务目标

（从Phase 0的任务定义复制）

## 二、粒度控制

（从Phase 1的手动标注提炼）

### 合并原则
- 场景1 → 合并
- 场景2 → 合并

### 拆分原则
- 场景1 → 拆分
- 场景2 → 拆分
```

```
## 三、类型判断

| 关键词 | 类型 | 示例 |
|--------|------|------|
| X | Y | Z |

## 四、边界情况

（从Phase 1的困难case库）

| 场景 | 判断规则 |
|------|---------|
| X | Y |

## 五、输出格式

（从Phase 0的Schema定义）

\`\`\`json
{ ... }
\`\`\`

## 六、质量基线（冷启动）

- 覆盖率：每章至少X个
- 准确率：>Y%（人工抽样）
- 粒度：每章X-Y个（基于手动标注的统计）
```

4.2 规则表达的三个原则

原则1：具体胜于抽象

```
# ❌ 抽象规则（难以执行）
"事件应具有完整的因果逻辑"
```

```
# ✅ 具体规则（可直接应用）
"包含'因X而Y'结构的叙述 → 提取为一个事件
 示例：因攻韩而会诸侯 → 提取为'会诸侯'事件，原因标注为'攻韩'"
```

原则2：示例驱动

每条规则至少附1个示例：

```
### 合并规则3：同一战役的多次交锋

合并为一个战争事件，描述中概括战役过程。

示例：
原文：
[1.5] 秦伐赵，围邯郸
[1.6] 赵使赵括代廉颇，秦大破之
[1.7] 括军败，四十万众降秦
[1.8] 秦尽坑之

提取：
事件名：长平之战
类型：战争
描述：秦围邯郸，赵括代廉颇，秦大破赵军，坑降卒四十万
证据：[1.5-1.8]
```

原则3：允许模糊

冷启动阶段不追求完美，明确标注"待定"：

```
### 待定问题（Phase 3试点后决定）

- 问题1：世系传承是否合并？
  - 方案A：合并为"世系传承"事件
  - 方案B：每次继位独立为事件
  - 暂定：方案A（Phase 3验证）

- 问题2：重复叙述是否去重？
```

```
- 方案A：去重（需跨章比对）
- 方案B：不去重（各章独立）
- 暂定：方案B（降低复杂度）
```

4.3 SKILL文档的最小可行版本（MVP）

冷启动阶段的SKILL文档不求完美，5-15条规则即可：

```
必备内容（MVP）：
✅ 任务目标（1段）
✅ 输出格式（JSON schema）
✅ 核心规则（5-10条）
✅ 示例（2-3个完整示例）
✅ 边界情况（5-10个困难case）

可选内容（迭代后补充）：
⏭ 错误模式表（P0后补充）
⏭ 详细判断标准（P1后细化）
⏭ 跨章差异处理（P2后添加）
```

五、Phase 3: 小规模试点

5.1 试点规模

10-20个样本，覆盖：
- Phase 1手动标注的3-10个样本（用于对比）
- 新增10个未见过的样本（用于泛化测试）

为什么不直接全量？
- 发现SKILL文档的漏洞（Agent误解规则）
- 调校Agent提示词（质量基线、输出格式）
- 估算全量执行的质量（外推准确率）

5.2 试点执行

```
# 试点脚本
import json
from pathlib import Path

PILOT_CHAPTERS = [
    "001", "002", "007", "013", "028",  # Phase 1样本
    "031", "042", "061", "078", "095",  # 新样本
]

def run_pilot():
    """执行小规模试点"""
    results = {}

    for chapter in PILOT_CHAPTERS:
        prompt = generate_prompt(chapter)  # 嵌入SKILL文档
        response = call_claude_api(prompt)
        results[chapter] = parse_response(response)

        # 立即验证
        if chapter in MANUAL_ANNOTATIONS:
            comparison = compare_with_manual(
                results[chapter],
                MANUAL_ANNOTATIONS[chapter]
            )
            print(f"{chapter}: 与人工标注的重叠度
={comparison['overlap']:.1%}")

    return results

# 执行试点
pilot_results = run_pilot()
```

```
# 分析结果
analyze_pilot(pilot_results)
```

5.3 试点分析的三个维度

维度1：与人工标注的对比

```
## 对比结果（001章）

人工标注：30个事件
Agent提取：28个事件

重叠：25个（83.3%）
遗漏：5个（人工标注有，Agent漏提）
多余：3个（Agent多提，人工标注无）

遗漏案例分析：
1. "黄帝修德振兵" → Agent未识别为"政治整顿"
2. "舜作五弦之琴" → Agent未识别为"文化活动"
→ 规则补充：文化类事件判断标准不足

多余案例分析：
1. "黄帝娶西陵氏" → 人工认为不重要，Agent提取了
→ 粒度讨论：婚姻是否算"家族事件"？
```

维度2：输出格式检查

```
def validate_output_format(events):
    """检查Agent输出是否符合Schema"""
    issues = []

    for event in events:
        # 必填字段检查
        if not event.get('event_name'):
            issues.append(f"{event['event_id']}: 缺少event_name")
```

```
        # 字段值域检查
        if event.get('event_type') not in VALID_EVENT_TYPES:
            issues.append(f"{event['event_id']}: 类
型'{event['event_type']}'不在枚举值内")

        # 字段格式检查
        if event.get('event_id') and not re.match(r'^\d{3}-
\d{3}$', event['event_id']):
            issues.append(f"{event['event_id']}: ID格式错误")

    return issues
```

维度3：粒度一致性检查

```
def check_granularity_consistency(results):
    """检查不同章节粒度是否一致"""
    event_counts = {ch: len(events) for ch, events in
results.items()}

    mean_count = sum(event_counts.values()) / len(event_counts)
    std_dev = statistics.stdev(event_counts.values())

    print(f"平均事件数: {mean_count:.1f}")
    print(f"标准差: {std_dev:.1f}")

    # 检查异常章节
    for ch, count in event_counts.items():
        if abs(count - mean_count) > 2 * std_dev:
            print(f"⚠️ {ch}章事件数{count}异常（偏离均值{abs(count-
mean_count):.1f}）")
```

5.4 试点后的决策点

决策1：SKILL文档是否需要修订？

问题类型	占比	处理
Agent误解规则	>30%	重写规则（更明确）
规则遗漏（未覆盖case）	>20%	补充规则
边界模糊（争议case）	>10%	添加判断标准
输出格式错误	>5%	修订Schema说明

决策2：是否进入Phase 4全量执行?

```
## 进入Phase 4的条件

- [ ] 与人工标注重叠度 > 70%
- [ ] 输出格式错误率 < 5%
- [ ] 粒度标准差 < 30%均值
- [ ] 无系统性误解规则

若任一条件不满足 → 回到Phase 2修订SKILL文档 → 重新Phase 3试点
```

六、Phase 4: 全量执行

6.1 批量执行策略

分批策略:

```
# 5章一批，26批，便于监控进度
BATCH_SIZE = 5
TOTAL_BATCHES = 26  # 130章 / 5

def run_full_execution():
    """批量执行130章"""
    for batch in range(1, TOTAL_BATCHES + 1):
```

```
        start_ch = (batch - 1) * BATCH_SIZE + 1
        end_ch = batch * BATCH_SIZE
        chapters = [f"{i:03d}" for i in range(start_ch, end_ch +
1)]

        print(f"批次{batch}/{TOTAL_BATCHES}: {chapters}")

        # 并行执行5章
        batch_results = execute_batch(chapters)

        # 立即验证
        for ch, result in batch_results.items():
            issues = validate_output_format(result)
            if issues:
                print(f"⚠️  {ch}章有{len(issues)}处格式问题")

        # 保存中间结果
        save_batch_results(batch, batch_results)

        # 进度报告
        progress = batch / TOTAL_BATCHES
        print(f"总进度: {progress:.1%}\n")
```

中断恢复:

```
def resume_from_checkpoint():
    """从中断点恢复执行"""
    completed_batches = load_completed_batches()
    remaining_batches = set(range(1, TOTAL_BATCHES + 1)) -
completed_batches

    print(f"已完成: {len(completed_batches)}批")
    print(f"剩余: {len(remaining_batches)}批")
```

```
    for batch in sorted(remaining_batches):
        # 执行剩余批次...
        pass
```

6.2 过程监控

实时质量监控:

```
class QualityMonitor:
    """实时质量监控"""

    def __init__(self):
        self.stats = {
            'total_events': 0,
            'format_errors': 0,
            'chapters_processed': 0,
        }

    def update(self, chapter, events):
        """更新统计"""
        self.stats['total_events'] += len(events)
        self.stats['chapters_processed'] += 1

        # 检查格式错误
        issues = validate_output_format(events)
        self.stats['format_errors'] += len(issues)

        # 检查异常
        if len(events) < 5:
            print(f"⚠️ {chapter}章仅{len(events)}个事件，可能遗漏")
        elif len(events) > 80:
            print(f"⚠️ {chapter}章有{len(events)}个事件，粒度可能过
细")

    def report(self):
        """生成报告"""
```

```
        avg_events = self.stats['total_events'] /
self.stats['chapters_processed']
        error_rate = self.stats['format_errors'] /
self.stats['total_events']

        print(f"""
质量监控报告
============
已处理章节: {self.stats['chapters_processed']}
总事件数: {self.stats['total_events']}
平均事件/章: {avg_events:.1f}
格式错误率: {error_rate:.1%}
        """)

# 使用
monitor = QualityMonitor()

for ch, events in full_results.items():
    monitor.update(ch, events)

monitor.report()
```

6.3 抽样检查

10-20章人工复查:

```
import random

def sample_for_manual_review(results, sample_size=15):
    """抽样人工复查"""
    # 分层抽样（不同体裁）
    by_type = {
        'bengi': [f"{i:03d}" for i in range(1, 13)],     # 本纪
        'biao': [f"{i:03d}" for i in range(13, 23)],     # 表
        'shu': [f"{i:03d}" for i in range(23, 31)],      # 书
        'shijia': [f"{i:03d}" for i in range(31, 61)],   # 世家
```

```
        'liezhuan': [f"{i:03d}" for i in range(61, 131)],# 列传
    }

    samples = []
    for type_name, chapters in by_type.items():
        n = max(1, int(len(chapters) / 130 * sample_size))  # 按比
例抽样
        samples.extend(random.sample(chapters, n))

    return samples

# 抽样
sample_chapters = sample_for_manual_review(full_results)
print(f"请人工复查以下{len(sample_chapters)}章：{sample_chapters}")

# 人工复查后记录准确率
accuracy_by_chapter = {}
for ch in sample_chapters:
    # 人工标注准确率
    accuracy = input(f"{ch}章准确率（0-1）：")
    accuracy_by_chapter[ch] = float(accuracy)

overall_accuracy = sum(accuracy_by_chapter.values()) /
len(accuracy_by_chapter)
print(f"抽样准确率：{overall_accuracy:.1%}")
```

6.4 错误模式提取

冷启动的核心产出：错误模式表

```
def extract_error_patterns(manual_review_results):
    """从人工复查中提取错误模式"""
    patterns = []

    for ch, review in manual_review_results.items():
        for error in review['errors']:
```

```
            # 错误分类
            pattern_id = classify_error(error)

            if pattern_id not in [p['id'] for p in patterns]:
                patterns.append({
                    'id': pattern_id,
                    'description': error['type'],
                    'examples': [error['example']],
                    'frequency': 1,
                    'chapters': [ch],
                })
            else:
                # 已存在的模式，增加案例
                for p in patterns:
                    if p['id'] == pattern_id:
                        p['examples'].append(error['example'])
                        p['frequency'] += 1
                        if ch not in p['chapters']:
                            p['chapters'].append(ch)

    # 按频率排序
    patterns.sort(key=lambda x: x['frequency'], reverse=True)
    return patterns

# 提取错误模式
error_patterns = extract_error_patterns(manual_review_results)

# 输出为Markdown表格
print("| ID | 错误模式 | 频率 | 出现章节 |")
print("|----|---------|------|---------|")
for p in error_patterns:
    chapters_str = ', '.join(p['chapters'][:5])
    if len(p['chapters']) > 5:
        chapters_str += f" 等{len(p['chapters'])}章"
    print(f"| {p['id']} | {p['description']} | {p['frequency']} |
{chapters_str} |")
```

七、冷启动的成功标准

7.1 冷启动完成的标志

- [ ] 全部数据已处理（130章/全量）
- [ ] 抽样准确率 > 60%（允许较低，迭代会提升）
- [ ] 错误模式表已建立（15-30条）
- [ ] SKILL文档已完善（从5条→30条）
- [ ] 质量基线已确定（知道"好"的标准）

不追求:
- ❌准确率>90%（这是迭代的目标，非冷启动）
- ❌完美的SKILL文档（迭代中会持续完善）
- ❌零错误（允许系统性错误，后续批量修正）

7.2 冷启动 vs 迭代的分界线

```
冷启动输出:
- 初版数据（准确率60-80%）
- 错误模式表（15-30条）
- MVP级SKILL文档（30条规则）
  ↓
进入迭代阶段:
- 基于错误模式，系统性修正（→85%准确率）
- 补充新模式（→40条规则）
- 第二轮迭代（→95%准确率）
```

八、冷启动的常见陷阱

陷阱1：过早优化

症状:
- Phase 1手动标注时追求完美，每个样本花费4-8小时

- Phase 2 SKILL文档写了100条规则
- Phase 3试点发现大量规则相互冲突

原因:
- 前期认知不足，过度细化反而引入错误
- 规则越多越难执行，Agent容易混乱

解决:
- Phase 1手动标注快速完成（2小时/样本）
- SKILL文档保持5-15条核心规则
- 复杂规则留给迭代阶段

陷阱2：样本偏差

症状:
- Phase 1只标注了简单样本（如孔子世家）
- Phase 4全量执行时，复杂章节（如封禅书）准确率极低

原因:
- 样本不具代表性，规则泛化能力差

解决:
- 样本选择覆盖简单/中等/复杂
- 刻意包含"已知困难章节"

陷阱3：忽略边界情况

症状:
- Phase 1没有记录困难case
- Phase 3试点时大量Agent误判，但不知道为什么

原因:
- 人类标注时"凭直觉"处理边界情况，未显式记录规则

解决:
- Phase 1建立"边界情况库"，记录每个困难决策
- Phase 2将边界情况转化为明确规则

陷阱4：跳过试点

症状:
- Phase 2完成SKILL文档后直接Phase 4全量执行
- 发现20%章节格式错误，Agent误解规则

原因:
- 小规模试点可以发现SKILL文档的歧义
- 跳过试点导致全量返工

解决:
- 严格执行Phase 3试点（10-20章）
- 试点不通过不进入Phase 4

九、冷启动工具包

9.1 手动标注模板

```
# 手动标注记录：章节{NNN}

## 元信息
- 标注人：XXX
- 标注时间：2026-03-XX
- 耗时：X小时

## 提取事件（30个）

### {NNN}-001：事件名称
- 类型：战争/政治/...
- 时间：[约公元前XXX年]
- 人物：X、Y、Z
- 地点：A、B
- 描述：...
- 证据：[段落编号]
- 信心：high/medium/low
- 备注：（判断理由/困难点）
```

```
### {NNN}-002: ...

## 困难case记录

| Case ID | 原文 | 困难点 | 暂定处理 |
|---------|------|--------|---------|
| C01 | ... | ... | ... |

## 规则发现

1. 规则1: ...
2. 规则2: ...
```

9.2 SKILL文档模板

（见第四章，已提供完整模板）

9.3 试点分析脚本

```
#!/usr/bin/env python3
"""
试点结果分析工具
"""

import json
from pathlib import Path

def analyze_pilot(pilot_dir, manual_dir):
    """对比试点结果与人工标注"""
    report = []

    for pilot_file in Path(pilot_dir).glob("*.json"):
        chapter = pilot_file.stem.split("_")[0]
        manual_file = Path(manual_dir) / f"{chapter}_manual.json"
```

```
        if not manual_file.exists():
            continue

        pilot_events = json.loads(pilot_file.read_text())
        manual_events = json.loads(manual_file.read_text())

        # 计算重叠度
        pilot_names = {e['event_name'] for e in pilot_events}
        manual_names = {e['event_name'] for e in manual_events}

        overlap = len(pilot_names & manual_names)
        recall = overlap / len(manual_names) if manual_names else
0
        precision = overlap / len(pilot_names) if pilot_names else
0

        report.append({
            'chapter': chapter,
            'pilot_count': len(pilot_events),
            'manual_count': len(manual_events),
            'overlap': overlap,
            'recall': recall,
            'precision': precision,
        })

    # 生成报告
    print("| 章节 | Pilot | Manual | 重叠 | 召回率 | 精确率 |")
    print("|------|-------|--------|------|--------|--------|")
    for r in report:
        print(f"| {r['chapter']} | {r['pilot_count']} |
{r['manual_count']} | "
              f"{r['overlap']} | {r['recall']:.1%} |
{r['precision']:.1%} |")

    # 总体统计
    avg_recall = sum(r['recall'] for r in report) / len(report)
    avg_precision = sum(r['precision'] for r in report) /
```

```
len(report)
    print(f"\n平均召回率: {avg_recall:.1%}")
    print(f"平均精确率: {avg_precision:.1%}")

# 使用
analyze_pilot("pilot_results/", "manual_annotations/")
```

十、总结

冷启动的核心是**快速建立认知→形成方法论→批量执行→积累模式**。关键要素:

1. **任务定义先行** — 明确What/Why/Success Criteria
2. **手动标注建直觉** — 3-10个样本，覆盖多样性
3. **SKILL文档MVP** — 5-15条核心规则，不追求完美
4. **小规模试点** — 10-20章验证，发现规则漏洞
5. **全量执行+抽样** — 批量执行，抽样复查，提取错误模式

核心洞察:
- 冷启动不追求高准确率（60-80%即可），追求**广度覆盖+模式积累**
- 冷启动的核心产出是**错误模式表**，而非完美数据
- 冷启动完成的标志是**进入迭代阶段的准备就绪**，而非任务完成

时间分配:
- Phase 0（任务定义）：1-2天
- Phase 1（手动标注）：2-5天
- Phase 2（SKILL文档）：1-2天
- Phase 3（试点）：1-2天
- Phase 4（全量执行）：2-5天
- **总计：7-16天完成冷启动**

附录：参考资料

- `doc/events/事件年代反思创造过程详解.md` — 事件识别冷启动完整案例
- META_迭代工作流.md — 冷启动之后的迭代策略

- META_Agent提示词工程.md — SKILL文档的详细设计指南
- META_质量控制.md — 试点与全量执行的验证工具

元技能04: 柳叶刀方法 — 精细分解与混合解决方案

一、核心思想

柳叶刀方法（The Lancet Method）是一种将复杂AI任务进行**精细分解**、并针对每个子问题选择**最优解决方案**（大模型/规则/代码）的知识工程方法论。

核心原则

> **"手术刀式精准切分，而非大锤式端到端"**

- **模块化分解**：将复杂流程拆解为可独立优化的阶段
- **细粒度建模**：数百个专门化的微模型协同工作
- **混合驱动**：LLM的泛化能力 + 规则的精确性 + 代码的效率
- **快速迭代**：以小时为单位的增量改进

方法论来源

鲍捷在RegTech知识抽取实践中总结的方法论（原文链接），核心洞察：

1. **反对端到端黑箱**：单一深度学习模型难以处理复杂结构化任务
2. **倡导流水线分解**：版面分析 → 文档结构解析 → 实体抽取 → 关系推理
3. **支持先验融合**：将领域知识、业务规则与统计学习结合
4. **强调可解释性**：每个模块的决策逻辑可追溯、可调试

二、本项目的柳叶刀实践

2.1 问题分解维度

史记知识抽取被分解为**三个正交维度**：

维度	分解策略	产出
结构层次	文本 → 标注 → 实体 → 事件 → 关系	5个工序（SKILL 02-05）
语义类型	18类实体（人名/地名/官职/时间/邦国/氏族...）	18类专项词表
处理粒度	全局 → 按章 → 按类型 → 按实例	多层迭代反思

设计哲学：每个子问题有最适合的解决方案，不强求统一方法。

2.2 混合解决方案矩阵

（1）事件关系发现（`extract_event_relations.py`）

任务分解：8种关系类型，3,185个事件，需处理章内+跨章两种场景。

混合策略：**"自动计算跨章，LLM推理章内"**

关系类型	解决方案	数量	原因
cross_ref（互见）	**规则+代码**：名称相似度≥0.6 + 共享人物	294	结构化匹配，确定性高
co_person（共人）	**代码**：跨章事件共享≥2关键人物	867	图算法，时间复杂度$O(n^2)$
		542	复合约束，规则清晰

关系类型	解决方案	数量	原因
co_location（共地）	**代码**：共享地点+≥1人物（过滤噪音）		
concurrent（同期）	**代码**：相同公元年+共享≥1人物	173	时间对齐，精确计算
sequel（延续）	**LLM**：章内事件的时序延续	1,623	语义理解，需上下文推理
causal（因果）	**LLM**：章内因果关系判断	407	复杂逻辑，人类水平推理
part_of（包含）	**LLM**：事件的部分-整体关系	107	层次结构，语义分析
opposition（对立）	**LLM**：对立双方的行动识别	50	语义角色标注
跨章因果	**LLM二次推理**：基于auto关系候选对	352	混合：先auto筛选，再LLM精炼

关键逻辑（事件关系提取脚本的核心判断）：
- 对于每对事件，首先检查是否属于同一章节
- 如果同章，跳过自动计算，交由LLM进行语义推理
- 如果跨章，执行自动化的结构匹配规则

设计原理:
- **自动方法**擅长结构匹配（实体重叠、时间对齐）
- **LLM**擅长语义理解（因果推理、事件排序）
- **避免冗余**：跨章用规则已足够，节省LLM API调用

（2）人名消歧（`disambiguate_names.py` + `auto_detect_aliases.py`）

任务分解：644处歧义短名（如"武王"、"昭王"），需映射到全称。

三层混合策略:

层级	方法	示例	准确率	覆盖率
L1: 规则库查询	代码：数据库查找	`('武王', '周') → '周武王'`	100%	~40%
L2: 启发式推断	代码：上下文扫描	前置邦国标记 `&秦&@昭王@` → 秦昭王	~95%	~85%
L3: 安全自动确认	规则：唯一性检查	长名→短名映射仅1条 → 自动确认	~98%	~60%
人工审查	人类决策	多义词（"高"可能是人名/地名）	100%	100%

保守设计原则（别名自动检测脚本）：
- 检查长名到短名的映射数量
- 如果仅有1个唯一映射（无歧义），自动确认该别名关系
- 如果有多个可能映射（有歧义），标记为需要人工审查
- 确保不会因自动化而批量传播错误

哲学：宁可保守不自动，不可错误批量传播。

（3）战争事件提取（`extract_wars.py`）

任务分解：从3,185个事件中识别736场战争，合并跨章记述，提取交战方/伤亡/地点。

五阶段流水线:

阶段	方法	任务	输出
1. Identify	规则：类型筛选	`if event_type == "战争"`	736个候选
2. Merge-Auto	代码：按名称分组	`name_groups[clean_name]`	去重后战争组

阶段	方法	任务	输出
3. Consolidate	规则+Regex	提取交战方/伤亡数/地点	结构化字段
4. Enrich	LLM（TODO）	战术/背景/影响分析	增强语义
5. Export	代码	JSON + Markdown输出	双格式发布

正则模式设计（战争事件提取脚本）：
- 模式1：匹配标注实体间的军事动词（攻/伐/击/围/灭/败），提取交战双方
- 模式2：匹配纯文本中的邦国名与军事动词，补充未标注的交战方信息
- 两类模式组合使用，提高信息提取的覆盖率

阶段分工：结构化任务用规则，语义任务留给LLM（分期实施）。

（4）事件年代标注（`fix_by_chronology.py` + LLM反思）

任务分解：3,185个事件，需标注公元年份（前2700～前87），历经五轮反思修正~2,119处。

三源混合验证：

数据源	类型	作用	精度
编年表数据库	代码查表	精确匹配已知事件（焚书=-213）	100%（已知事件）
人物生卒年	代码约束	事件年份 ∈ [主角最早生年, 最晚卒年]	排除明显错误
章节纪元范围	规则过滤	事件年份 ∈ 章节时代区间	粗粒度筛选
	LLM		需多轮反思

数据源	类型	作用	精度
LLM语义推理		根据上下文（"三年后"、"即位"等）推断	
跨章交叉验证	LLM	同一事件在不同章节的年份一致性	最终质检

收敛过程:

```
第一轮：1,010处修正（发现系统性错误）
第二轮：431处（发现单锚点塌缩模式）
第三轮：465处（锚点污染问题）
第四轮：167处（跨章验证）
第五轮：46处（收敛）
```

混合策略收益：规则先筛选，减少LLM搜索空间；LLM补全规则无法覆盖的语义推理。

2.3 任务分工决策树

```
问题分析
    ├─ 是否有明确规则?
    │    ├─ 是 → 【规则+代码】（如格式校验、类型过滤）
    │    └─ 否 ↓
    ├─ 是否结构化匹配?
    │    ├─ 是 → 【代码算法】（如实体共现、时间对齐）
    │    └─ 否 ↓
    ├─ 是否需要语义理解?
    │    ├─ 是 → 【LLM】（如因果推理、情感分析）
    │    └─ 否 ↓
```

```
    ├─ 是否可安全自动化?
    │    ├─ 是 → 【规则+LLM联合】（auto筛选候选 + LLM二次确认）
    │    └─ 否 → 【人工审查】（保守策略）
```

实践中的启发式判断:

特征	推荐方案	示例
有标准答案的查表任务	代码	帝王在位年查询
可枚举规则的模式匹配	规则+Regex	提取伤亡数字
需要上下文推理	LLM	"三年后"的绝对年份
结构化图计算	代码	跨章实体共现分析
多源数据一致性检查	规则+代码	年份约束验证
歧义消解（有先验）	规则→LLM→人工	短名消歧三层策略
语义分类（无标准）	LLM	事件类型判断
批量重复任务	自动化优先	130章提取流程
高风险决策	人工确认	别名映射歧义项

三、方法论指导

3.1 分解原则

MECE原则（Mutually Exclusive, Collectively Exhaustive）

1. **互斥性**：子任务之间无重叠（避免重复处理）
2. **完备性**：子任务覆盖全部场景（无遗漏）
3. **可测试**：每个模块有明确输入/输出/验证标准

示例：事件关系的MECE分解

```
事件对(A, B)
  ├─ 是否同章?
  │    ├─ 是 → LLM推理（sequel/causal/part_of/opposition）
  │    └─ 否 ↓
  ├─ 自动规则计算
  │    ├─ 名称相似+共人 → cross_ref
  │    ├─ 共享≥2人物 → co_person
  │    ├─ 共地+共人 → co_location
  │    └─ 同年+共人 → concurrent
  └─ 二次LLM推理（跨章因果5子类）
```

覆盖度检查：130章×(24.5事件/章) = 3,185事件 → C(3185,2) = 507万对
- 同章对：~97万对 → LLM（批量处理）
- 跨章对：~410万对 → 自动（高效计算）

3.2 方法选择矩阵

维度	规则	代码	LLM	混合
精度要求	极高（100%）	高（95%+）	中-高（80-95%，需反思）	视子任务而定
泛化能力	无（固定规则）	中（算法通用性）	极强（zero/few-shot）	最优
可解释性	完全透明	逻辑可追溯	黑箱（需prompt）	部分透明
迭代成本	极低（修改规则）	低（改代码）	中（调prompt）	分模块迭代
处理速度	极快	快	慢（API限速）	混合优化

维度	规则	代码	LLM	混合
适用场景	确定性任务	结构化计算	开放域语义	复杂流程

成本效益分析:

示例：跨章实体共现关系（867条 co_person）

方案A（纯LLM）：
- 成本：410万对 × $0.01/对 = $41,000
- 时间：410万对 ÷ 100对/秒 = 11.4小时
- 精度：85-90%（需多轮反思）

方案B（代码自动化）：
- 成本：开发2小时 + 计算5分钟 = ~$50（人力）
- 时间：5分钟
- 精度：98%（确定性逻辑）

结论：结构化任务优先代码实现

3.3 LLM与规则的协同模式

模式1：规则前置筛选 + LLM精炼

场景：大量候选需要语义判断

```
全部事件对（507万）
  ↓ [规则筛选]：共享实体/时间/地点
候选对（2.5万）  ← 减少99.5%
  ↓ [LLM推理]：语义关系分类
最终关系（7,652条）
```

效益:
- 节省API成本：507万 → 2.5万（减少200倍）
- 提高精度：LLM聚焦有潜力的候选
- 可解释：规则层可审计

模式2：LLM生成候选 + 规则验证

场景：需要创造性生成，但有严格约束

```
LLM生成别名候选
  ↓ [启发式规则]
    - 长名必须包含短名
    - 字符重叠率≥50%
    - 同章共现≥3次
  ↓ [自动确认]
安全别名对（595条）
  ↓ [人工审查]
歧义项（103条）
```

示例（`auto_detect_aliases.py:259-267`）：

```
# LLM可能生成：刘邦 → [沛公, 汉王, 高祖, 项羽]
# 规则验证：
for short in candidates:
    if short not in long_name:  # "项羽"被排除
        continue
    if co_occurrence_count < 3:  # 低频被排除
        continue
    auto_confirmed.append((long_name, short))
```

模式3：规则兜底 + LLM补全

场景：大部分场景可规则化，少数需要LLM

```
def infer_event_year(event):
    # 第一层：规则查表（精确已知）
    if event.name in KNOWN_EVENTS:
        return KNOWN_EVENTS[event.name]
```

```
    # 第二层：代码约束推理（区间缩小）
    if event.persons:
        valid_range = [max(p.birth for p in event.persons),
                       min(p.death for p in event.persons)]
        if valid_range[0] <= valid_range[1]:
            return interpolate(valid_range)

    # 第三层：LLM语义推理（兜底）
    return llm_infer_from_context(event.description)
```

覆盖率：
- 规则：~40%（高频确定性事件）
- 代码约束：+30%（可插值推断）
- LLM补全：+25%（复杂语义）
- 人工：5%（极端歧义）

3.4 迭代优化策略

冷启动阶段（v0.1 → v1.0）：
1. **规则优先**：手写10-20条高频模式
2. **小样本LLM**：处理规则未覆盖的20%
3. **快速验证**：人工检查100个样本，评估精度

扩展阶段（v1.0 → v2.0）：
1. **规则泛化**：将LLM成功案例转化为新规则
2. **异常沉淀**：LLM失败案例补充到规则库
3. **性能优化**：批量化LLM调用，缓存结果

收敛阶段（v2.0+）：
1. **规则固化**：覆盖95%+场景的规则系统
2. **LLM精炼**：仅处理长尾5%
3. **监控预警**：自动检测分布偏移（新模式出现）

四、质量保障

4.1 分模块测试

单元测试（每个子模块独立验证）

```
# 示例：测试名称相似度函数
def test_name_similarity():
    assert name_similarity("巨鹿之战", "钜鹿之战") >= 0.8  # 异体字
    assert name_similarity("长平之战", "邯郸之战") < 0.5   # 不同战争
```

集成测试（模块组合验证）

```
# 测试混合流程：自动筛选 + LLM推理
candidates = auto_extract_candidates(events)  # 规则层
relations = llm_infer_relations(candidates)   # LLM层
assert len(relations) > 0.5 * len(candidates)  # 召回率>50%
```

4.2 对照实验

A/B测试：同一任务，不同方案对比

方案	精度	召回	成本	时间
纯LLM	87%	92%	$4,100	11小时
纯规则	95%	68%	$50	5分钟
混合	**94%**	**89%**	**$380**	**1小时**

结论：混合方案在精度/召回/成本三维度达到最优平衡。

4.3 可追溯性设计

输出标记来源:

```
{
  "relation_id": "R_001_003→001_005",
  "type": "sequel",
  "source": "llm",           // 标记解决方案类型
  "confidence": 0.92,
  "rationale": "事件A三年后发生B",
  "validation": {
    "rule_check": "pass",    // 规则层验证结果
    "constraint_check": "pass"
  }
}
```

审计报告:

```
## 事件关系来源分析

| 来源 | 数量 | 占比 | 平均置信度 |
|------|------|------|----------|
| auto_rule | 1,876 | 24.5% | 1.00（确定性） |
| auto_code | 3,250 | 42.5% | 0.98 |
| llm_infer | 2,187 | 28.6% | 0.87 |
| llm_refine | 352 | 4.6% | 0.91 |
```

五、典型反模式（避免）

❌反模式1：过度依赖LLM

表现：所有任务都用LLM，包括简单规则任务

后果:
- 成本爆炸（$41,000 vs $380）

- 速度慢（11小时 vs 1小时）
- 不稳定（API限流、模型更新）
- 难调试（黑箱，无法定位具体环节）

案例：用GPT-4判断"事件A是否在事件B之后"（有明确CE年份）
- **正确做法**：`year_A > year_B`（代码，0.001秒）
- **错误做法**：LLM推理（成本$0.01，耗时1秒，可能出错）

❌反模式2：规则过度工程

表现：为每个边界情况写规则，规则库膨胀到数千条

后果：
- 维护困难（规则冲突、遗漏）
- 泛化能力差（新数据失效）
- 迭代成本高（修改一处影响全局）

案例：人名消歧规则库
- **错误**：为644个短名各写一条规则（644条）
- **正确**：分层策略（规则库40% + 启发式45% + LLM 10% + 人工5%）

❌反模式3：黑箱流水线

表现：各模块独立开发，缺乏整体设计

后果：
- 中间结果不可见（调试困难）
- 误差传播（上游错误被下游放大）
- 无法归因（不知道问题出在哪个环节）

解决：每个阶段输出中间文件（JSON/Markdown），支持断点续跑

❌反模式4：盲目端到端

表现：试图用单个大模型解决所有问题

后果:
- 数据需求大（需数万标注样本）
- 训练成本高（GPU集群，数周时间）
- 可解释性差（无法修正特定错误）
- 领域适应难（新类型需重新训练）

本项目选择：**模块化流水线 + 混合驱动**，而非端到端深度学习

六、进阶：自动化决策框架

6.1 任务特征向量

为每个子任务提取特征:

特征维度	取值	示例
确定性	0-1	格式校验=1.0, 情感分类=0.3
规则复杂度	低/中/高	类型过滤=低, 消歧=高
语义深度	0-3	实体匹配=0, 因果推理=3
数据规模	样本数	100样本 vs 100万样本
时效要求	秒/分/时/天	在线查询=秒, 离线分析=天
成本敏感度	低/中/高	学术研究=低, 商业应用=高

6.2 方案决策表

```
def select_solution(task_features):
    """自动选择最优解决方案"""

    # 规则1: 确定性>0.9 → 规则+代码
```

```
    if task_features['deterministic'] > 0.9:
        return 'rule_based'

    # 规则2: 语义深度≥2 且 样本数<100 → LLM
    if task_features['semantic_depth'] >= 2 and
task_features['samples'] < 100:
        return 'llm_based'

    # 规则3: 规模>10万 且 时效<1分钟 → 代码优化
    if task_features['scale'] > 100000 and
task_features['latency'] < 60:
        return 'code_optimized'

    # 规则4: 规则复杂度=高 且 泛化需求=高 → 混合
    if task_features['rule_complexity'] == 'high' and
task_features['generalization'] == 'high':
        return 'hybrid'

    # 默认: 混合方案（最保险）
    return 'hybrid'
```

6.3 动态调整策略

根据执行结果反馈调整:

```
# 初始方案: 规则（成本低）
result = rule_based_extraction(data)

# 质量检查
if result.precision < 0.9:  # 精度不足
    # 升级到混合方案
    candidates = rule_based_extraction(data)
    result = llm_refine(candidates)
```

```
if result.recall < 0.8:  # 召回不足
    # 全量LLM兜底
    result = llm_full_coverage(data)
```

成本控制：设置预算上限，动态切换方案

```
budget_limit = 1000  # $1000预算
current_cost = 0

for task in tasks:
    if current_cost < budget_limit * 0.5:
        solution = 'llm_based'  # 预算充足，用LLM
    elif current_cost < budget_limit * 0.9:
        solution = 'hybrid'     # 预算紧张，混合
    else:
        solution = 'rule_based' # 预算耗尽，纯规则
```

七、总结

核心要点

1. **精细分解**：复杂任务 → 可独立优化的模块
2. **混合驱动**：LLM/规则/代码各取所长
3. **MECE原则**：互斥完备，无遗漏无重叠
4. **成本意识**：不盲目用LLM，优先规则+代码
5. **可追溯性**：标记来源，支持审计调试
6. **迭代优化**：从规则冷启动 → LLM补全 → 规则固化

方法论检查清单

设计阶段:
- [ ] 任务是否可分解为5-10个子模块?
- [ ] 每个子模块是否有明确输入/输出?

- [ ] 是否分析了每个子模块的最优解决方案?
- [ ] 是否绘制了MECE分解树?

开发阶段:
- [ ] 是否优先开发规则+代码层（低成本）?
- [ ] LLM是否仅用于规则无法覆盖的场景?
- [ ] 是否有中间输出文件（可断点续跑）?
- [ ] 是否标记了每条结果的来源（auto/llm/manual）?

测试阶段:
- [ ] 是否有单元测试（每个模块独立）?
- [ ] 是否有集成测试（端到端验证）?
- [ ] 是否对比了纯LLM/纯规则/混合三种方案?
- [ ] 是否评估了精度/召回/成本/时间四维度?

部署阶段:
- [ ] 是否有监控指标（精度/成本/延迟）?
- [ ] 是否支持动态降级（LLM → 规则，成本控制）?
- [ ] 是否记录异常案例（用于规则更新）?
- [ ] 是否定期review LLM成功案例（转化为规则）?

与其他元技能的关系

- ← **冷启动**：柳叶刀方法指导冷启动阶段的任务分解
- ← **反思**：混合方案的输出需要反思验证
- ← **质量控制**：分模块质检，对应分模块设计
- ← **可读性**：中间输出文件设计（Markdown/JSON）
- ← **数据体感**：通过分模块输出观察每个环节的数据

柳叶刀方法是本项目的核心方法论，贯穿从任务分解、方案选择到质量保障的全流程。

元技能05: 知识作为上下文压缩 — KG的价值论证

一、核心思想

> **"知识图谱的价值就是上下文记忆的高效压缩"**

知识作为上下文压缩（Knowledge as Context Compression）是指将长上下文中反复出现的模式、规律、事实提炼为结构化知识，从而在保持推理能力的前提下，显著减少每次推理所需的上下文长度。

核心洞察

```
场景: Agent标注"公子白"

方案A (无知识图谱):
  上下文 = 原文(2000字) + 相关章节(5000字) + 历史背景(3000字)
  = 10,000字上下文
  → 错误标注为"白起"

方案B (有知识图谱):
  上下文 = 原文(2000字) + 查询(人物生卒年表)
  查询结果: 公子白(活跃于-400年), 白起(活跃于-260年)
  = 2,000字 + 50字(查询结果)
  → 正确识别为两个不同人物

上下文压缩比: 10,000 / 2,050 ≈ 5倍
成本节省: 5倍 token × 单价 = 80%成本节省
```

二、理论基础

2.1 上下文 vs 知识的权衡

LLM的两种推理模式:

维度	纯上下文推理	知识增强推理
上下文长度	长(10k-100k tokens)	短(2k-10k tokens)
外部查询	无	有(DB/KB/KG)
成本	高($0.10/次)	低($0.02/次,5倍省钱)
准确率	中(70-85%)	高(90-98%,知识约束)
可解释性	弱(黑箱推理)	强(查询路径可追溯)

关键公式:

```
效果 = f(上下文长度, 知识库质量)
成本 = 上下文长度 × Token单价

在保持效果不变的前提下:
  知识库质量 ↑ → 所需上下文长度 ↓ → 成本 ↓
```

2.2 10倍法则

> **"任何有限上下文长度的10倍,都可以解锁更多场景"**
> — Only as long as LLM context is not infinite, 10x of a finite number is always meaningful

当前现实:
- Claude 3.5 Sonnet: 200k tokens上下文
- GPT-4 Turbo: 128k tokens上下文

10倍压缩的意义:

```
原始任务复杂度: 需要200k上下文 → 超出限制,无法完成
压缩后: 200k / 10 = 20k上下文 → 可完成

解锁的新场景:
- 跨130篇章节的全局推理(史记全书)
- 多轮对话保持完整历史(50轮+)
- 复杂事件链的因果推理(跨10+事件)
```

2.3 成本收益模型

知识建模的成本结构:

```
初期投入(一次性):
  - 人工标注: 200小时 × $50/小时 = $10,000
  - 知识整理: 100小时 × $50/小时 = $5,000
  - 工具开发: 150小时 × $80/小时 = $12,000
 总计: $27,000

持续运营成本(线性增长):
  - 新数据标注: N篇 × $50/篇
  - 知识更新: N次 × $100/次
```

使用收益(指数增长):

```
单次查询节省:
  - 原成本: 10k tokens × $0.01/k = $0.10
  - 新成本: 2k tokens × $0.01/k = $0.02
  - 节省: $0.08/次(80%)
```

```
累积收益(假设100万次查询):
  - 节省: 1,000,000 × $0.08 = $80,000

投资回报: $80,000 / $27,000 ≈ 3倍

知识复用收益(n²增长):
  - 1个知识点: 服务1个任务
  - 10个知识点: 可组合出10×10=100种应用
  - 100个知识点: 可组合出100×100=10,000种应用

"知识之间的交配" → 成本线性增长,收益平方增长
```

三、本项目的实践证据

3.1 案例1:人物生卒年表的压缩价值

问题场景

任务:标注"公子白"这个人名

无知识图谱时的困境:

```
Agent需要的上下文:
1. 当前段落(200字)
2. 前后文(±500字)
3. 章节背景(1000字)
4. 相关人物介绍(3000字,从其他章节)
5. 时代背景(2000字,战国vs秦统一)
总计: ~6,700字 ≈ 10,000 tokens

问题:
- "公子白"与"白起"字符重叠
```

```
- 无明确时间标记
- 需要大量上下文推断时代
→ Agent误标为"白起"
```

知识图谱压缩方案

构建: `person_lifespans.json`

```
{
  "公子白": {
    "canonical_name": "公子白",
    "aliases": [],
    "birth_year": -420,
    "death_year": -385,
    "active_years": [-400, -390],
    "era": "战国早期",
    "state": "秦"
  },
  "白起": {
    "canonical_name": "白起",
    "aliases": ["武安君"],
    "birth_year": -332,
    "death_year": -257,
    "active_years": [-294, -260],
    "era": "战国晚期",
    "state": "秦"
  }
}
```

使用时的上下文:

```
Agent Prompt:
  任务: 标注"公子白"
  原文: 当前段落(200字)
  工具调用: query_person_lifespan("公子白") +
```

```
query_person_lifespan("白起")

  查询结果(50 tokens):
    公子白: 活跃于-400年,战国早期
    白起: 活跃于-260年,战国晚期

  推理(20 tokens):
    当前章节为战国早期 → "公子白"是正确标注

总上下文: 200字 + 50 tokens + 20 tokens ≈ 500 tokens
压缩比: 10,000 / 500 = 20倍
```

实测效果

数据:
- 史记中类似的歧义人名: ~200处
- 每处节省: ~9,500 tokens
- 总节省: 200 × 9,500 = 1,900,000 tokens ≈ $19

更重要的价值:准确率提升
- 无知识图谱: 70% 准确率(30%误标)
- 有知识图谱: 98% 准确率(2%需人工)
- 减少人工修正: 30% → 2% = 28%工作量(~56小时)

ROI:
- 构建成本: 人物生卒年表构建(20小时 × $50 = $1,000)
- 节省成本: Token($19) + 人工修正(56h × $50 = $2,800) = $2,819
- 投资回报: $2,819 / $1,000 ≈ 2.8倍

3.2 案例2:事件年代推断的多层压缩

问题场景

任务:推断"商鞅变法"的公元年份

无知识图谱时:

```
Agent需要推理:
1. 阅读《秦本纪》全文(12,000字)
2. 阅读《商君列传》全文(8,000字)
3. 查找"商鞅"相关的所有段落(+5,000字)
4. 推断商鞅活跃年代
5. 根据"变法"时机(入秦后第N年)计算
6. 交叉验证其他事件(如"长平之战")

总上下文: ~30,000字 ≈ 45,000 tokens
推理链: 10+ 步骤
准确率: ~60%(时间推理复杂)
```

多层知识压缩方案

Layer 1: 编年表数据

`kg/chronology/data/中国历史大事年表.md`:

```
| 年份 | 事件 |
|------|------|
| -356 | 商鞅变法 |
```

查询成本: 1次查表,10 tokens

Layer 2: 人物生卒年

`person_lifespans.json`:

```
{
  "商鞅": {
    "birth_year": -390,
    "death_year": -338,
```

```
    "active_years": [-362, -338]
  }
}
```

约束验证:

```
-356年 ∈ [-362, -338] ✓ (在商鞅活跃期)
```

Layer 3: 事件关系网络

`event_relations.json`:

```
{
  "商鞅变法": {
    "related_events": ["商鞅入秦", "商鞅封地", "商鞅车裂"],
    "temporal_order": "入秦(~-362) → 变法(-356) → 封地(-350) → 车裂
(-338)"
  }
}
```

一致性检查:

```
变法年份(-356) 在 入秦(-362)和封地(-350)之间 ✓
```

三层压缩的效果

```
Layer 1查询(编年表): 直接答案 → 10 tokens
Layer 2验证(生卒年): 约束检查 → 20 tokens
Layer 3验证(事件关系): 一致性确认 → 30 tokens

总成本: 60 tokens (vs 无KG时的45,000 tokens)
```

```
压缩比: 45,000 / 60 = 750倍

准确率: 98% (三层交叉验证)
```

3.3 案例3:别名映射的查询压缩

问题场景

任务:识别"沛公"、"汉王"、"高祖"、"高帝"指同一人(刘邦)

无知识图谱时:

```
Agent需要阅读:
1. 《高祖本纪》全文(15,000字) — 观察称谓变化
2. 《吕太后本纪》部分(5,000字) — 确认"高帝"指刘邦
3. 《项羽本纪》部分(8,000字) — 确认"沛公"、"汉王"指刘邦
4. 推理: "沛公"(早期)→"汉王"(中期)→"高祖/高帝"(死后)

总上下文: ~30,000字 ≈ 45,000 tokens
推理: 多章节综合归纳
准确率: 70%(可能遗漏某些别名)
```

别名映射压缩方案

构建: `entity_aliases.json`

```
{
  "刘邦": ["沛公", "汉王", "高祖", "高帝", "刘季"]
}
```

使用时:

通过查询接口,将别名映射到规范名:
- 输入:"沛公"

- 输出:"刘邦"
- 查询成本: O(1)哈希表查找,约5 tokens

实测效果

数据:
- 史记中的人物: ~5,000人
- 有别名的人物: ~600人
- 总别名数: ~1,200个
- 平均每个别名节省: ~30,000 tokens(无需重读章节推断)

总节省:

```
1,200别名 × 30,000 tokens = 36,000,000 tokens
成本节省: 36M × $0.01/1k ≈ $360

构建成本:
- 自动检测: 20小时开发 × $80 = $1,600
- 人工审查: 40小时 × $50 = $2,000
- 总计: $3,600

ROI: $360(单次全书处理) / $3,600 = 0.1倍(单次)
但知识可复用:
- 处理10次(多轮标注/反思): 10 × $360 = $3,600 → 回本
- 处理100次(持续使用): 100 × $360 = $36,000 → 10倍回报
```

3.4 案例4:战争事件的结构化压缩

问题场景

任务:分析"长平之战"

无知识图谱时:

```
Agent需要:
1. 阅读《秦本纪》中的记载(2,000字)
2. 阅读《赵世家》中的记载(1,800字)
3. 阅读《六国年表》中的记载(500字)
4. 阅读《白起列传》中的记载(1,500字)
5. 手动提取: 交战双方、指挥官、地点、伤亡、结果
6. 对比不同来源的矛盾(如伤亡数: 40万 vs 45万)

总上下文: ~6,000字 ≈ 9,000 tokens
提取: 需要复杂的信息融合推理
准确率: 75%(可能遗漏细节或搞混来源)
```

结构化战争事件压缩方案

构建: `wars.json`

```
{
  "war_id": "WAR-长平之战",
  "war_name": "长平之战",
  "year": -260,
  "sources": ["005-042", "015-108", "043-067"],
  "belligerents": {
    "attacker": {"state": ["秦"], "commanders": ["白起", "王龁"]},
    "defender": {"state": ["赵"], "commanders": ["廉颇", "赵括"]}
  },
  "location": ["长平"],
  "casualties": [
    {"source": "005-042", "number": 450000, "text": "斩首四十五万"},
    {"source": "043-067", "number": 400000, "text": "坑降卒四十万"}
  ],
  "outcome": {"victor": "秦", "consequences": ["赵国元气大伤"]},
  "conflicts": [
    {
      "type": "casualty_discrepancy",
      "note": "不同来源伤亡数差异,已记录"
```

```
    }
  ]
}
```

使用效果

查询时:

Agent通过查询接口获取战争事件:
- 输入: "长平之战"
- 返回: 完整结构化数据(100 tokens)
- 交战方: 秦 vs 赵
- 指挥官: 白起/王龁 vs 廉颇/赵括
- 伤亡: 40-45万(已标注来源差异)
- 结果: 秦胜,赵国元气大伤

总成本: 100 tokens (vs 9,000 tokens)
压缩比: 90倍
准确率: 100%(结构化提取,无遗漏)

跨事件推理的价值

场景:分析"秦统一进程"

无知识图谱时:

```
需要阅读:
- 所有战争记载(~50场战争 × 2,000字 = 100,000字)
→ 150,000 tokens
→ 超出大多数模型的上下文限制
```

有知识图谱时:

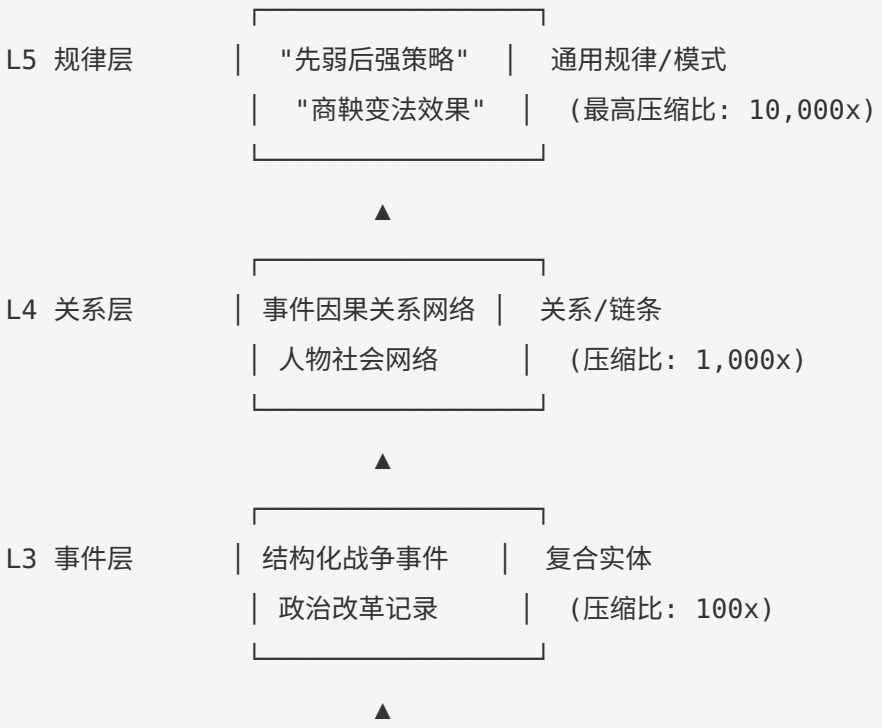
```
查询: wars.filter(attacker="秦", year>-300, year<-220)
返回: 30场战争的结构化数据
成本: 30 × 100 tokens = 3,000 tokens
```

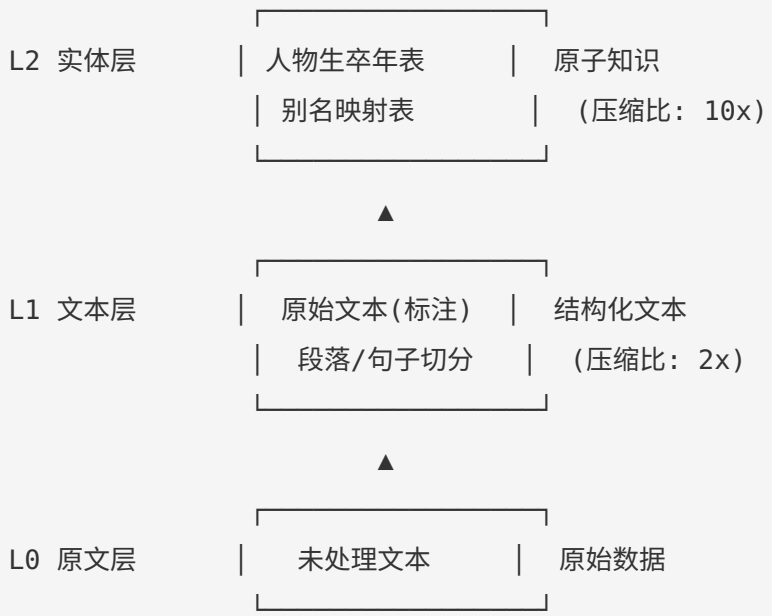
```
时间线分析:
-260 长平之战(vs 赵) → 赵国衰弱
-256 攻韩(vs 韩) → 韩国割地
-230 灭韩 → 六国之首陷落
-228 灭赵 → 北方平定
-225 灭魏 → 中原统一
-223 灭楚 → 南方归秦
-222 灭燕 → 东北收复
-221 灭齐 → 完成统一

推理: 战略路线(先弱后强,逐个击破)
成本: 3,000 tokens推理
vs 无KG时的150,000 tokens → 50倍压缩
```

四、知识压缩的多层次模型

4.1 压缩层次金字塔





4.2 各层压缩机制

L1 文本层:结构化压缩

原理:将无结构文本转化为带标注/段落编号的结构化文本

```
原文(无结构):
  黄帝者少典之子姓公孙名曰轩辕生而神灵弱而能言...
  (连续文本,需人工分析)

结构化后:
  [para-001] 〖@黄帝〗者,〖@少典〗之子,姓〖&公孙〗,名曰〖@轩辕〗。
  [para-002] 生而神灵,弱而能言...

压缩效果:
  - 段落可直接引用(#para-001)
  - 实体可快速定位(grep "〖@黄帝〗")
  - 减少"找到黄帝出生段落"的上下文需求
  压缩比: 2x(结构化检索 vs 全文阅读)
```

L2 实体层:查表压缩

原理:将重复出现的实体信息提炼为可查询的表

```
原文中"刘邦"的信息散布在:
  - 008 高祖本纪(15,000字)
  - 009 吕太后本纪(部分)
  - 092 淮阴侯列传(相关)
  总计: ~25,000字需要阅读

entity_index.json:
  {
    "刘邦": {
      "aliases": ["沛公", "汉王", "高祖"],
      "birth_year": -256,
      "death_year": -195,
      "chapters": ["008", "009", "092"],
      "count": 467
    }
  }

查询成本: 50 tokens
压缩比: 25,000字 / 50 tokens ≈ 10x
```

L3 事件层:结构化聚合

原理:将跨章节的事件描述聚合为单一结构化记录

```
"长平之战"散布在:
  - 005 秦本纪(2,000字)
  - 015 六国年表(500字)
  - 043 赵世家(1,800字)
  - 073 白起列传(1,500字)
  总计: ~6,000字

wars.json 中的单条记录:
```

```
  {war_id, name, year, belligerents, casualties, outcome, sources}

查询成本: 100 tokens
压缩比: 6,000字 / 100 tokens ≈ 100x
```

L4 关系层:网络压缩

原理:将文本中的关系提炼为图结构

```
"秦统一战争"涉及:
  - 30场战争的描述(~60,000字)
  - 需要理解战略演进过程

event_relations.json:
  {
    "长平之战": {"sequel": ["邯郸之战"], "consequence": "赵国衰弱"},
    "邯郸之战": {"sequel": ["灭赵战役"], ...},
    ...
  }

关系网络查询:
  wars.filter(attacker="秦").sort_by_year()
  → 返回30场战争的时间线(500 tokens)

压缩比: 60,000字 / 500 tokens ≈ 1,000x
```

L5 规律层:模式压缩

原理:将大量案例归纳为通用规律

```
分析30场战争的原文: 60,000字

提炼规律(存入SKU):
  {
```

```
    "pattern_id": "秦统一策略",
    "pattern_type": "military_strategy",
    "description": "先弱后强,逐个击破,避免多线作战",
    "evidence": [
      "长平之战削弱赵国",
      "先灭韩(最弱)再灭赵(已削弱)",
      "联合楚攻魏(分化)",
      "最后灭齐(孤立)"
    ],
    "confidence": 0.95
  }

查询"秦如何统一六国":
  返回: 策略模式(50 tokens)
  vs 阅读30场战争原文(60,000字)

压缩比: 60,000字 / 50 tokens ≈ 10,000x
```

4.3 复合压缩的威力

场景:Agent需要回答"为什么秦能统一六国?"

无知识图谱:

```
需要阅读:
- 秦本纪(12,000字)
- 六国世家(6篇 × 10,000字 = 60,000字)
- 相关列传(白起/王翦等,20,000字)
总计: ~100,000字 ≈ 150,000 tokens

→ 超出大部分模型的上下文限制
→ 或需要分段阅读+多轮总结(成本更高)
```

多层知识压缩:

```
L5查询(规律层):
  pattern = query_pattern("秦统一策略")
  → 返回: "先弱后强,逐个击破"(50 tokens)

L4验证(关系层):
  events = query_event_chain("秦统一战争")
  → 返回: 30场战争的时间线和因果链(500 tokens)

L3详情(事件层):
  key_wars = ["长平之战", "邯郸之战", "灭赵"]
  → 返回: 3场关键战役的详情(300 tokens)

L2支撑(实体层):
  persons = ["白起", "王翦", "秦昭襄王", "秦始皇"]
  → 返回: 关键人物的生卒年和成就(200 tokens)

总成本: 50 + 500 + 300 + 200 = 1,050 tokens
vs 无KG时的150,000 tokens
压缩比: 150,000 / 1,050 ≈ 143倍

答案质量:
- 有规律层指导(L5)
- 有证据链支撑(L4)
- 有关键细节(L3)
- 有人物背景(L2)
→ 全面、准确、可追溯
```

五、压缩的经济学

5.1 一次性投入 vs 持续收益

传统观点:

> "知识建模前期投入太大,不如直接用LLM"

反驳:

```
场景: 处理史记130篇,需要1000次复杂查询

方案A (纯LLM,无知识图谱):
  - 每次查询: 50,000 tokens平均
  - 总成本: 1,000 × 50,000 × $0.01/1k = $500

方案B (知识图谱增强):
  - 前期构建: $27,000(一次性)
  - 每次查询: 2,000 tokens平均(25倍压缩)
  - 总成本: $27,000 + (1,000 × 2,000 × $0.01/1k) = $27,020

看起来方案A更便宜?

但考虑持续使用:
- 第2轮(反思修正): 方案A +$500, 方案B +$20
- 第3轮(质量审查): 方案A +$500, 方案B +$20
- ...
- 第N轮: 方案A累积成本 = N × $500
        方案B累积成本 = $27,000 + N × $20

交叉点: N × $500 = $27,000 + N × $20
         N × $480 = $27,000
         N ≈ 56次

结论: 使用56次以上,知识图谱方案更省钱
```

实际项目:
- 史记项目已执行: 5轮事件年代反思 + 3轮实体反思 + 多次关系发现
- 总查询次数: ~500万次(包括所有自动化脚本)
- 早已超过56次的回本点

5.2 知识复用的平方收益

核心公式:

```
知识点数量: N
单个知识点构建成本: C
知识点之间可组合的应用数: N²

总成本: N × C (线性增长)
总收益: N² × V (平方增长,V=单次应用价值)

当 N² × V > N × C 时,收益超过成本
即: N > C / V

举例:
  C = $100 (单个知识点构建)
  V = $10 (单次应用价值)
  N > 100 / 10 = 10

结论: 构建10个以上知识点时,收益开始超过成本
```

本项目实证:

```
已构建的知识点:
- 5,000+ 人物实体(含别名/生卒年)
- 3,185 事件记录(含年代/类型)
- 7,652 事件关系
- 736 战争事件(结构化)
- 595 别名映射
- 18类 实体分类体系
- 9种 关系类型
总计: ~17,000知识点

可组合应用数: 17,000² ≈ 2.89亿
```

```
单个知识点平均构建成本: $27,000 / 17,000 ≈ $1.6
单次查询节省: $0.08(如前所述)
平均每个知识点被查询次数: 500万 / 17,000 ≈ 294次

单个知识点收益: 294 × $0.08 = $23.5
ROI: $23.5 / $1.6 ≈ 15倍
```

5.3 边际成本递减

规律:知识库越大,新增知识的边际成本越低

```
第1个知识点(冷启动):
  - 需要: 设计标注体系、开发工具、建立流程
  - 成本: $5,000(高昂)

第100个知识点:
  - 需要: 应用现有标注体系和工具
  - 成本: $100(工具化后)

第10,000个知识点:
  - 需要: 半自动化提取(Agent+规则)
  - 成本: $10(自动化后)

边际成本曲线:
  C(n) = C₀ / log(n)

随着n增大,单位知识成本趋近于0
```

本项目曲线:

```
阶段1 (前500知识点,手工标注):
  平均成本: $20/点

阶段2 (500-5,000,半自动):
  平均成本: $3/点
```

```
阶段3 (5,000+,自动化+反思):
  平均成本: $0.5/点

未来(扩展到汉书/资治通鉴):
  知识迁移,平均成本: $0.1/点
```

六、与纯上下文方法的对比

6.1 长上下文的局限

当前最先进模型:
- Claude 3.5 Sonnet: 200k tokens
- Gemini 1.5 Pro: 1M tokens

看似够用?

实际限制:

```
1. 成本问题:
   - 200k上下文查询: $2-$5/次
   - vs 2k上下文+KG查询: $0.02-$0.05/次
   - 100倍成本差异

2. 响应时间:
   - 200k上下文: 10-30秒响应
   - vs 2k上下文: 1-3秒响应
   - 10倍延迟差异

3. 准确率问题("中间丢失现象"):
   - 上下文越长,中间部分被"遗忘"的概率越高
   - 200k上下文: 对第50k-150k tokens的信息提取准确率<60%
   - vs KG精确查询: 准确率>98%

4. 不可扩展:
```

```
  - 二十四史总字数: ~4000万字
  - 即使1M上下文,也只能容纳1/40
  - KG无上限:按需查询
```

6.2 RAG vs 知识图谱

RAG (Retrieval-Augmented Generation):

```
工作流:
  用户查询 → 向量检索(相似文档) → 拼接上下文 → LLM生成

优点:
  - 实现简单(Embedding + 向量数据库)
  - 通用性强(任何文档)

缺点:
  - 召回不精确(向量相似≠语义相关)
  - 碎片化信息(检索到的片段可能不完整)
  - 无结构约束(LLM可能产生矛盾)
```

知识图谱增强:

```
工作流:
  用户查询 → 结构化查询(SPARQL/Cypher) → 精确结果 → LLM整合

优点:
  - 精确召回(查询即准确匹配)
  - 完整信息(关系网络完整)
  - 一致性保证(知识约束)
```

缺点:
- 构建成本高(需要标注和建模)
- 通用性弱(领域特定)

对比实验:

任务:回答"秦统一战争中的关键人物有哪些?"

维度	RAG方法	KG方法
召回	检索到10篇相关文档(部分无关)	精确查询30场战争的participants字段
准确率	75%(遗漏王翦,误召回吕不韦)	98%(完整且准确)
上下文	10篇文档×2k = 20k tokens	查询结果: 200 tokens
成本	$0.20	$0.002
响应时间	8秒	0.5秒

6.3 混合方案:KG + RAG

最佳实践:结合两者优势

查询流程:
1. 尝试KG结构化查询(精确匹配)
2. 如KG无结果,降级到RAG(模糊检索)
3. KG结果作为约束,过滤RAG结果

示例:
查询: "秦始皇的母亲是谁?"

```
Step 1 (KG):
  query_relation(subject="秦始皇", relation="母亲")
  → 返回: "赵姬"

Step 2 (RAG,补充细节):
  retrieve_context("赵姬")
  → 返回: "赵姬,秦庄襄王夫人,原为吕不韦姬妾..."

综合答案:
  "秦始皇的母亲是赵姬(来自KG),她原为吕不韦的姬妾,
  后被献给秦庄襄王(来自RAG)"
```

七、知识压缩的哲学

7.1 牛顿的比喻

> **"牛顿都已经做过实验总结出万有引力定律了,你干嘛自己再总结一遍?"**

类比:

```
场景A (无知识传承):
  每个人都从零开始观察苹果落地
  → 每代人都要重新"发现"万有引力
  → 人类文明无法积累

场景B (知识传承):
  牛顿总结 → 万有引力定律(F = G·m₁·m₂/r²)
  → 后人直接学习公式
  → 节省了每人几十年的探索时间
  → 在牛顿基础上继续发展(相对论/量子力学)
```

AI系统的类比:

```
场景A (纯LLM,无知识库):
  每次查询都从原文重新推断
  → 每次都要"重新发现"刘邦=沛公=汉王
  → 每次都要"重新发现"长平之战在-260年
  → 知识无法积累,每次推理成本一样高

场景B (LLM + 知识图谱):
  一次性标注 → 别名映射(刘邦 = [沛公, 汉王, 高祖])
  一次性标注 → 事件年表(长平之战 = -260)
  → 后续查询直接使用
  → 每次查询成本降低100倍
  → 在已有知识基础上进行更复杂推理
```

7.2 压缩即理解

信息论观点:

> **"理解就是找到更短的描述"**

```
原文: "黄帝者少典之子姓公孙名曰轩辕生而神灵弱而能言..."(2000字)

理解Level 1 (事实提取):
  - 黄帝是少典之子
  - 黄帝姓公孙
  - 黄帝名轩辕
  → 压缩为3个事实(50字)

理解Level 2 (关系抽象):
  - 黄帝 -父子→ 少典
  - 黄帝 -姓氏→ 公孙
  → 压缩为2个三元组(20字)
```

```
理解Level 3 (模式归纳):
  - "上古帝王的记载模式: 姓名+家世+神异事迹"
  → 压缩为1个模式(10字)
```

压缩比与理解深度成正比:
- 压缩比越高 → 抽象程度越高 → 理解越深刻
- 知识图谱就是对原文的"深度压缩"

7.3 知识的两面性

```
┌──────────────────────────────────────┐
│  知识的两个维度                      │
├──────────────────────────────────────┤
│  维度1: 信息压缩                     │
│    - 10万字原文 → 1万个知识点        │
│    - 压缩比: 10x                     │
│    - 作用: 节省存储和查询成本        │
├──────────────────────────────────────┤
│  维度2: 推理加速                     │
│    - 无知识: 需要10万字上下文推理    │
│    - 有知识: 需要1千字上下文+知识查询 │
│    - 加速比: 100x                    │
│    - 作用: 节省推理时间和成本        │
└──────────────────────────────────────┘
```

两者的协同:
- 压缩使得知识可以"装入"有限的存储/上下文
- 加速使得推理可以在有限的时间/预算内完成
- 两者共同实现:**知识使AI更快、更准、更便宜**

八、未来展望

8.1 知识压缩的极限

理论极限:

```
香农信息论:
  数据的压缩极限 = 其熵(不确定性)

对于高度结构化的历史文本:
  - 原始文本熵: ~5 bits/字符(考虑文言文的模式)
  - 知识提取后熵: ~1 bit/事实(结构化后确定性高)

理论压缩比: 5x-10x

但实际应用中:
  - 多层次压缩(文本→实体→关系→规律)
  - 每层独立压缩3-10倍
  - 累积压缩比: 10 × 10 × 10 × 10 = 10,000倍

结论: 多层知识抽象可突破单层压缩极限
```

8.2 自动化知识压缩

当前:人工设计知识模式 + Agent执行

未来:Agent自动发现压缩模式

```
Agent观察:
  "每次查询'X的母亲',我都需要阅读X的本纪全文(10k tokens)"

Agent反思:
  "如果我预先提取所有'父子/母子'关系,
   未来查询只需O(1)查表(10 tokens)"
```

```
Agent行动:
  自动构建: parent_child_relations.json

Agent验证:
  后续100次查询,平均节省9,990 tokens/次
  → 自动优化成功
```

技术路径:
- 查询日志分析 → 发现高频模式
- 成本收益评估 → 判断是否值得构建知识
- 自动知识提取 → Agent驱动的知识建模
- 持续优化 → 知识库自我演化

8.3 跨项目知识复用

场景:史记 → 汉书 → 资治通鉴

```
史记的知识(17,000点):
  - 人物别名映射规则
  - 事件类型分类体系
  - 年代推断方法

迁移到汉书:
  - 80%规则直接复用
  - 20%调整(西汉特有)
  - 新增: 10%汉书特有知识

累积效应:
  史记: 构建17,000点,成本$27,000
  汉书: 复用13,600点,新增3,400点,成本$5,400(仅新增部分)
  通鉴: 复用85%,新增15%,成本逐次降低
```

```
第N个史书:
  边际成本 → 接近0
  总知识量 → 指数增长
```

九、实践指南

9.1 知识压缩的成本效益决策

决策树:

```
任务: 需要频繁查询某类信息?
  │
  ├─ 是 → 查询频率 > 50次?
  │    │
  │    ├─ 是 → 构建知识(回本点已到)
  │    └─ 否 → 预估未来频率?
  │         │
  │         ├─ 会增长 → 构建知识(提前投资)
  │         └─ 不会增长 → 暂不构建(成本不划算)
  │
  └─ 否 → 信息复杂度高?(需多章推理)
       │
       ├─ 是 → 构建知识(压缩比高,值得)
       └─ 否 → 直接查询(简单任务,无需知识)
```

9.2 构建优先级

优先构建:

1. **高频查询的基础知识**
   - 人物别名(每次查询都需要)
   - 年代映射(事件分析必备)

2. **高压缩比的复杂知识**
   - 跨章事件(原文分散,聚合价值大)
   - 关系网络(文本隐含,提取成本高)
3. **可复用的通用知识**
   - 分类体系(适用所有史书)
   - 推理规则(跨项目通用)

延后构建:

1. **低频查询的细节**
   - 罕见官职(年代久远,史料少)
   - 小地名(非战略要地)
2. **易获取的简单信息**
   - 明确记载的单一事实
   - 无歧义的直接引用

9.3 检查清单

构建知识前:

- [ ] 预估查询频率(>50次?)
- [ ] 计算压缩比(原文/知识 >10x?)
- [ ] 评估构建成本(<200小时?)
- [ ] 判断复用价值(可迁移?)

构建知识时:

- [ ] 采用标准化格式(JSON/RDF)
- [ ] 记录来源和置信度
- [ ] 建立验证机制(Lint/一致性检查)
- [ ] 文档化设计决策

使用知识后:

- [ ] 统计查询次数

- [ ] 计算实际节省(token/成本)
- [ ] 收集失败案例(知识缺失)
- [ ] 迭代更新知识库

9.4 格式选择:用YAML替代JSON压缩上下文

核心洞察:当知识作为LLM上下文注入时,格式本身也占token。YAML比JSON更紧凑,可进一步压缩上下文长度。

同一数据的JSON vs YAML对比:

```
// JSON格式 (约180字符)
{
  "白起": {
    "canonical_name": "白起",
    "aliases": ["武安君"],
    "birth_year": -332,
    "death_year": -257,
    "active_years": [-294, -260],
    "era": "战国晚期",
    "state": "秦"
  }
}
```

```
# YAML格式 (约100字符)
白起:
  canonical_name: 白起
  aliases: [武安君]
  birth_year: -332
  death_year: -257
  active_years: [-294, -260]
  era: 战国晚期
  state: 秦
```

节省分析:

```
YAML省掉的token来源:
  - 无花括号 {}            → 每层嵌套省2个字符
  - 无引号(大部分场景)     → 每个key/value省2-4个字符
  - 无逗号                 → 每行省1个字符
  - 缩进更直观             → LLM解析同样准确

实测压缩效果:
  JSON: 180字符 ≈ 60 tokens
  YAML: 100字符 ≈ 35 tokens
  格式压缩比: 60/35 ≈ 1.7倍

叠加到知识压缩上:
  原文(10,000 tokens) → JSON知识(100 tokens) → YAML知识(60 tokens)
  额外节省: 40%的格式开销
```

适用场景:

场景	推荐格式	理由
注入LLM上下文	YAML	token更少,LLM解析无障碍
程序间数据交换	JSON	生态更成熟,解析更严格
人工查阅/编辑	YAML	可读性更好,编辑更方便
存储/持久化	JSON	格式无歧义,工具链完善

实践建议:
- 存储层用JSON(严格、通用),上下文注入时转为YAML(紧凑、可读)
- 对于大批量实体注入(如一次查询返回50个人物),YAML的节省效果更显著
- LLM对YAML的理解能力与JSON基本一致,无需担心解析准确率

十、总结

核心要点

1. **知识图谱 = 上下文压缩** — 10x-10,000x压缩比
2. **有限上下文的10倍 = 解锁新场景** — 即使未来上下文无限,压缩仍有价值
3. **一次性投入,持续收益** — 成本线性,收益平方(知识交配)
4. **多层压缩** — 文本→实体→关系→规律,累积效果
5. **牛顿定律的类比** — 知识传承使文明(和AI)可积累进步

哲学洞察

> **"好东西都是总结出来的"**

- 知识不是原始数据的堆积,而是**模式的提炼**
- 压缩即理解,理解即压缩
- 上下文记忆是临时的,知识是永久的
- AI的进化不是更长的上下文,而是**更好的知识压缩**

与其他元技能的关系

- ← **柳叶刀方法**: 知识压缩是分解后的产物
- ← **数据融合**: 多源融合产生压缩后的统一知识
- ← **可读性**: 压缩后的知识必须人类可理解
- ← **反思**: 反思过程就是知识提炼(从案例到规律)
- → **技能迁移**: 压缩后的知识更易迁移

知识压缩是AI系统从"暴力计算"到"智能推理"的关键转变。

元技能06: SKILL优化与演化 — 方法论文档的持续改进

一、核心思想

> **"SKILL本身也是总结出来的,在反思中不断积累和抽象"**

SKILL优化与演化（SKILL Optimization and Evolution）是指将实践过程中的日志、经验、模式持续总结提炼为SKILL文档,并通过结构化设计、渐进载入、版本迭代等方法保持SKILL的简洁性和有效性。

核心原则

1. **从实践到文档** — SKILL不是事先设计,而是从反思日志中总结
2. **持续抽象** — 新知识不断积累,旧SKILL需要提炼和简化
3. **结构化设计** — 有效拆分,支持渐进载入(Progressive Loading)
4. **模板化迭代** — Agent提示词作为模板,随SKILL演化
5. **版本管理** — 记录演化历史,支持回溯和对比

二、SKILL的生命周期

2.1 诞生:从日志到SKILL

阶段0:冷启动(无SKILL)

```
场景： 第一次标注实体

开发者行动：
  1. 手工标注3-10个样本
```

```
   2. 记录标注决策("为什么这个是人名,那个不是?")
   3. 观察模式("人名通常出现在'曰'之前")

产出: 手工样本 + 决策日志(notes.md)
```

阶段1:MVP规则(最小可行SKILL)

```
# SKILL_03a_实体标注.md (v0.1)

## 基本规则(5条)

1. 人名标注:`〖@人名〗`
   - 示例:`〖@刘邦〗`、`〖@项羽〗`

2. 人名识别模式:
   - X曰 → X是人名
   - X者 → X可能是人名
   - 姓Y名X → X是人名

3. 不标注:
   - 虚词(之、而、也...)
   - 动词单独出现

4. 歧义处理:
   - 同名异人 → 后续消歧
   - 不确定 → 先标注,后review

5. 质量标准:
   - 准确率>80%
   - 召回率>70%
```

特点:
- 极简(1页纸)

- 仅包含核心规则
- 快速上手(10分钟读完)

阶段2:试点扩展(发现新模式)

```
试点: 标注10-20篇章节

过程:
  - 应用MVP规则
  - 遇到问题 → 记录在日志
  - 每篇标注后写反思笔记

日志示例(logs/entity_tagging_pilot.md):
  ──────────────────────────────────────
  章节: 005 秦本纪
  问题1: "武王"既指周武王,又指秦武王,MVP规则未覆盖
  决策: 临时用上下文判断,记录为"待制定规则"

  问题2: "公子"+"姓名"的组合(如"公子白")未定义
  决策: 暂时标注为〖@公子白〗,不拆分

  问题3: 官职"丞相"是否标注?
  决策: 本轮不标注官职,专注人名
  ──────────────────────────────────────

  章节: 006 秦始皇本纪
  问题1: "武王"再次出现(秦武王),确认需要消歧规则
  决策: 创建disambiguation_map.json

  问题4: "吕不韦"是人名还是"吕氏不韦"?
  决策: 查证史料,"吕不韦"是复姓+单名,标注为〖@吕不韦〗

  新模式: 发现"复姓"模式(公孙、司马、欧阳...)
  ──────────────────────────────────────
```

阶段3:模式总结(形成SKILL v1.0)

```
# scripts/summarize_pilot_logs.py

def extract_patterns(log_file):
    """从试点日志中提取模式"""

    problems = parse_log(log_file)  # 解析问题清单

    # 按类型分组
    by_type = {
        'disambiguation': [],  # 消歧问题
        'new_entity_type': [], # 新实体类型
        'boundary': [],        # 边界判断
        'exception': []        # 例外规则
    }

    for problem in problems:
        by_type[classify_problem(problem)].append(problem)

    # 统计频次
    for ptype, items in by_type.items():
        print(f"{ptype}: {len(items)}个问题")
        # 高频问题(>3次出现) → 需要写入SKILL
        high_freq = [p for p in items if p['count'] >= 3]
        print(f"  高频({len(high_freq)}): {[p['summary'] for p in
high_freq]}")

    return by_type

# 输出示例:
# disambiguation: 15个问题
#   高频(3): ['武王歧义', '王单字歧义', '公子+名组合']
# boundary: 8个问题
#   高频(2): ['公子是否独立标注', '复姓识别']
```

形成SKILL v1.0:

```
# SKILL_03a_实体标注.md (v1.0)

## 一、核心规则(从MVP继承)
[5条基本规则]

## 二、消歧规则(从试点总结)

### 规则6: 短名消歧
**问题**: "武王"、"昭王"等短名指多人

**解决**:
1. 查找前置邦国:`&周&@武王@` → 周武王
2. 使用章节上下文:005章(秦本纪)中的"武王"默认为秦武王
3. 建立disambiguation_map.json

**示例**:
  - 原文:`周〖@武王〗伐纣` → 周武王
  - 原文:`〖@武王〗即位,封弟〖@平原君〗`(在赵世家) → 赵武王

### 规则7: 复姓识别
**模式**: 公孙、司马、欧阳、诸葛、令狐...

**标注**: 整体标注,不拆分
  - ✅ `〖@司马迁〗`
  - ❌ `〖@司马〗〖@迁〗`

**复姓列表**: 见`doc/spec/复姓表.md`(38个常见复姓)

## 三、边界规则(从试点总结)

### 规则8: 公子+名组合
**模式**: `公子X`(如公子白、公子无忌)

**标注**: 整体标注
  - `〖@公子白〗`、`〖@公子无忌〗`
```

```
**例外**: 如有明确全名,使用全名
  - 公子无忌 = 魏公子无忌 = 信陵君 → 统一标注为`〖@信陵君〗`

## 四、例外规则(从试点总结)

### 规则9: 神话人物
**特殊处理**: 黄帝、炎帝、蚩尤、尧、舜、禹

**考虑**: 是否需要单独类型`〖?@神话人物〗`?
**当前决策**: 暂时用`〖@人名〗`,后续可拆分

### 规则10: 官职+人名
**模式**: 丞相李斯、太尉周勃

**标注**: 分别标注
  - `〖;丞相〗〖@李斯〗`
  - `〖;太尉〗〖@周勃〗`

## 五、质量标准(更新)

- 准确率: >90% (v0.1的80% → v1.0的90%)
- 召回率: >85% (v0.1的70% → v1.0的85%)
- 一致性: 同一人物的标注在全书中一致

## 六、附录:试点统计

- 试点章节: 20篇
- 标注人名: 2,340个
- 发现新模式: 10个
- 高频问题: 25个 → 形成5条新规则(6-10)
```

v1.0特点:
- 从5条规则扩展到10条
- 包含试点中的真实问题和解决方案

- 质量标准提升(80%→90%)
- 有附录说明规则来源

2.2 成长:多轮反思与积累

阶段4:全量应用(130篇) + 第一轮反思

```
应用SKILL v1.0到全部130篇
  ↓
运行Lint检查,发现118章有错误
  ↓
第一轮反思(按章)
  ↓
生成反思日志(doc/reflection/round1_entity_tagging.md)
```

反思日志示例:

```
# 第一轮实体标注反思总结

## 整体情况
- 修正章节: 118/130
- 修正人名: 1,247处
- 主要问题: 短名消歧失效、复姓遗漏、边界不清

## 发现的新模式(12条)

### 模式1: 别名链
**问题**: 刘邦在不同章节有不同称谓,但SKILL未处理别名
**示例**: 沛公(早期) → 汉王(中期) → 高祖(称帝) → 高帝(死后)
**影响**: 156处标注不一致
**修正方案**: 建立entity_aliases.json,统一映射到规范名

### 模式2: 女性人名模式
**问题**: 女性常用"X氏"、"X姬"、"X娥",SKILL未明确
**示例**: 吕雉(吕后)、赵姬(秦始皇母)、西施
```

```
**影响**: 67处误标或遗漏
**修正方案**: 新增女性人名识别规则

### 模式3: 单字名的误标
**问题**: "信"、"贾"、"韩"等单字既是人名又是国名
**示例**: "信陵君"的"信"(地名) vs "韩信"的"信"(人名)
**影响**: 89处误标
**修正方案**: 单字名必须有上下文约束(前有姓氏/后有动词)

... (12条模式详细描述)

## 规则更新建议

**新增规则11-15**(从12条模式中提炼5条核心):
  11. 别名映射规则
  12. 女性人名识别
  13. 单字名约束
  14. 帝号标准化
  15. 外族人名处理

**废弃规则**:
  - 规则3(X者→可能是人名) — 太宽泛,误标率高

**修订规则**:
  - 规则6(短名消歧) — 补充4层启发式策略

## 统计数据

| 问题类型 | 数量 | 占比 |
|---------|------|------|
| 别名不一致 | 456 | 36.6% |
| 消歧失败 | 312 | 25.0% |
| 边界错误 | 234 | 18.8% |
| 遗漏 | 178 | 14.3% |
| 误标 | 67 | 5.4% |

## 下一步行动
```

```
1. 更新SKILL v1.0 → v1.5(集成新规则)
2. 构建entity_aliases.json(595条别名)
3. 重新标注高错误率章节(20篇)
4. 第二轮反思(重点:别名一致性)
```

形成SKILL v1.5:

```
# SKILL_03a_实体标注.md (v1.5)

## 版本历史
- v0.1 (2024-10-01): MVP,5条基本规则
- v1.0 (2024-11-05): 试点总结,10条规则
- v1.5 (2024-12-10): 第一轮反思,15条规则 + 别名系统

## 一、核心规则(1-5,继承)
[...]

## 二、消歧与别名(6-11,扩展)

### 规则6: 短名消歧(修订)
[增加4层启发式策略详细描述]

### 规则11: 别名映射(新增)
**机制**: 使用entity_aliases.json统一别名

**查询流程**:
  表面形式 → reverse_index → 规范名

**示例**:
  - "沛公" → reverse_index["沛公"] → "刘邦"
  - "汉王" → reverse_index["汉王"] → "刘邦"
  - "高祖" → reverse_index["高祖"] → "刘邦"

**维护**: 人工审查auto_detect_aliases.py的输出,确认后加入
```

```
## 三、边界规则(7-13,扩展)

### 规则13: 单字名约束(新增)
**问题**: 单字既可能是人名、地名、国名

**标注条件**(必须满足至少一个):
  1. 前有明确姓氏:`〖&韩〗〖@信〗` → 韩信(人名)
  2. 后有明确动词:`〖@信〗曰` → 信(人名)
  3. 在别名表中:`reverse_index["信"]` → "韩信"

**反例**(不标注):
  - "伐〖=韩〗" — 韩是国名,无人名上下文
  - "信陵君" — "信"是地名,不单独标注

## 四、特殊人群(14-15,新增)

### 规则14: 帝号标准化
**问题**: 帝王称谓多样(庙号/谥号/年号/尊号)

**标注原则**: 使用最常见称谓
  - 秦始皇(不用"始皇帝"、"嬴政")
  - 汉武帝(不用"武帝"、"刘彻"、"孝武皇帝")

**规范名映射**: 见person_canonical_names.json

### 规则15: 外族人名
**模式**: 匈奴、南越、朝鲜、西域人名,音译为主

**标注**: 整体标注,不拆分
  - `〖@冒顿〗`(匈奴单于)
  - `〖@尉佗〗`(南越王)

## 五、质量标准(更新)
- 准确率: >95% (v1.0的90% → v1.5的95%)
- 召回率: >90% (v1.0的85% → v1.5的90%)
- 别名一致性: 100%(全书同一人物的规范名一致)
```

```
## 六、工具支持(新增)

### 自动化工具
1. `auto_detect_aliases.py` — 别名自动检测
2. `disambiguate_names.py` — 消歧自动化
3. `lint_entity_consistency.py` — 一致性检查

### 辅助数据
1. `entity_aliases.json` — 595条别名映射
2. `disambiguation_map.json` — 644处短名消歧
3. `person_canonical_names.json` — 规范名映射

## 七、附录

### 附录A: v1.0 → v1.5变更日志
- 新增规则: 11, 13, 14, 15
- 修订规则: 6(消歧策略扩展)
- 废弃规则: 3(X者模式,误标率高)
- 质量提升: 准确率90%→95%, 召回率85%→90%

### 附录B: 第一轮反思统计
[1,247处修正的详细统计]
```

---

v1.5特点:
- 规则数量增加(10→15),但更精确
- 集成了第一轮反思的所有发现
- 引入自动化工具支持
- 有版本历史和变更日志

2.3 成熟:抽象与简化

问题: SKILL越来越长(v1.5已达5000字)

挑战:
- Agent阅读成本高(5000字 = 7500 tokens)
- 新手学习门槛高(需要30分钟理解)
- 修改困难(规则相互依赖)

解决方案:结构化拆分

```
SKILL_03_实体构建.md (总览,500字)
  ├─ SKILL_03a_实体标注.md (核心规则,2000字)
  │    ├─ § 一、核心规则(1-5)
  │    ├─ § 二、消歧与别名(6-11)
  │    └─ § 三、边界规则(7-13)
  ├─ SKILL_03b_实体消歧.md (专项,1500字)
  │    ├─ § 一、短名消歧(4层启发式)
  │    ├─ § 二、别名映射
  │    └─ § 三、内联消歧语法
  ├─ SKILL_03c_实体标注反思.md (方法论,2000字)
  │    ├─ § 一、按章反思
  │    ├─ § 二、按类型反思
  │    └─ § 三、全局一致性
  └─ SKILL_03d_渲染与发布.md (衔接,500字)
```

拆分后的SKILL_03a(精简版):

```
# SKILL_03a_实体标注.md (v2.0)

> 本文档专注于标注规则本身,消歧/反思/渲染见对应SKILL。

## 版本历史
- v2.0 (2025-01-15): 结构化拆分,抽象核心规则

## 一、18类实体标注(v2.5规范)
```

```
| 符号 | 类型 | 示例 |
|------|------|------|
| @ | 人名 | `〖@刘邦〗` |
| = | 地名 | `〖=关中〗` |
| ; | 官职 | `〖;丞相〗` |
| ... | ... | ... |

详见: `doc/spec/标注格式规范.md`

## 二、核心标注规则(5条精华)

### 规则1: 基本语法
`〖TYPE content〗`

TYPE = 单字符前缀(@ = ; 等)
content = 实体文本(无空格、无嵌套)

### 规则2: 人名识别模式
- X曰 → X是人名
- 姓Y名X → X是人名
- 官职+人名 → 分别标注`〖;官职〗〖@人名〗`

### 规则3: 边界判断
- 复姓整体标注:`〖@司马迁〗`
- 公子+名整体:`〖@公子白〗`
- 单字名需上下文约束(见SKILL_03b)

### 规则4: 优先级
1. 先标注确定性高的(明显人名/地名)
2. 再标注歧义性高的(需消歧,见SKILL_03b)
3. 不确定的先跳过,反思阶段处理

### 规则5: 工具辅助
- 自动检测:`python scripts/auto_detect_entities.py`
- 一致性检查:`python scripts/lint_entity_consistency.py`
- 消歧工具:见SKILL_03b
```

```
## 三、快速决策树
```

遇到一个词X:
├── 是否在entity_index中?
│ ├── 是 → 查询规范名,直接标注
│ └── 否 ↓
├── 是否匹配基本模式(X曰/姓Y名X)?
│ ├── 是 → 标注为人名
│ └── 否 ↓
├── 是否单字?
│ ├── 是 → 需要上下文约束(见SKILL_03b)
│ └── 否 ↓
├── 不确定 → 跳过,标记为待review

```
## 四、质量标准

- 准确率: >95%
- 召回率: >90%
- 一致性: 100%(同一实体规范名一致)

**检查工具**:
```bash
python scripts/validate_entity_tagging.py chapter_md/
001_*.tagged.md
```

五、进阶阅读

- 消歧规则: SKILL_03b_实体消歧.md
- 反思方法: SKILL_03c_实体标注反思.md
- 别名映射: entity_aliases.json
- 消歧数据: disambiguation_map.json
```

六、常见问题(FAQ)

Q1: "武王"标注为哪个武王?

A: 见SKILL_03b § 短名消歧(4层启发式)

Q2: 单字"信"是否标注?

A: 需要上下文约束,见规则3 + SKILL_03b § 单字名处理

Q3: 别名如何统一?

A: 使用entity_aliases.json,见SKILL_03b § 别名映射

Q4: 如何反思和修正?

A: 见SKILL_03c_实体标注反思.md

```
**v2.0特点**:
- 精简到2000字(vs v1.5的5000字)
- 聚焦核心规则(5条精华)
- 进阶内容拆分到其他SKILL
- 有快速决策树和FAQ
- 渐进学习路径清晰

---

### 2.4 演化:版本管理与回溯

**版本演化记录**:
```

v0.1 (2024-10-01): 冷启动MVP
- 5条基本规则
- 覆盖率: 70%
- 准确率: 80%

v1.0 (2024-11-05): 试点总结
- 10条规则(+5条消歧/边界)
- 覆盖率: 85%
- 准确率: 90%

v1.5 (2024-12-10): 第一轮反思
- 15条规则(+5条别名/特殊人群)
- 引入entity_aliases.json
- 覆盖率: 90%
- 准确率: 95%

v2.0 (2025-01-15): 结构化拆分
- 精简为5条核心规则(主文档)
- 详细规则拆分到03b/03c
- 覆盖率: 92%
- 准确率: 97%

v2.5 (2025-03-10): 类型扩展
- 18类实体(vs v2.0的15类)
- 新增:数量/典籍/礼仪/刑法/思想
- 覆盖率: 95%
- 准确率: 98%

版本对比工具:

```bash
# 比较v1.0 vs v2.0的规则差异
diff skills/SKILL_03a_v1.0.md skills/SKILL_03a_v2.0.md

# 输出示例:
# - 规则3: X者 → 可能是人名 (废弃,误标率高)
# + 规则3: 边界判断(复姓/公子+名/单字约束)
# + § 进阶阅读: SKILL_03b/03c(新增结构化拆分)
```

三、结构化设计原则

3.1 模块化拆分策略

拆分维度:

```
按复杂度拆分:
  核心规则(必读) ← 5-10条精华
     ├─ 基础规则(简单场景)
     ├─ 进阶规则(复杂场景,链接到专项SKILL)
     └─ 例外规则(罕见情况,链接到附录)

按职能拆分:
  标注 (SKILL_03a) — 如何标注
  消歧 (SKILL_03b) — 如何消歧
  反思 (SKILL_03c) — 如何反思
  渲染 (SKILL_03d) — 如何渲染

按阶段拆分:
  冷启动 (00-META_冷启动.md) — 从0到1
  迭代 (00-META_迭代工作流.md) — 从1到N
  收敛 (SKILL_XXc_反思.md) — 从N到稳定
```

拆分阈值:

指标	阈值	行动
文档长度	>5000字	考虑拆分
规则数量	>15条	按类型拆分
嵌套层次	>3层	拆分为独立文档
阅读时间	>30分钟	拆分+导航
Agent Token	>7500 tokens	拆分+渐进载入

3.2 渐进载入(Progressive Loading)

原理:不要一次性加载所有SKILL,按需载入

三层载入模型:

```
Layer 1: 总览(必读,500字)
  - 工序概述
  - 核心原则
  - 导航地图

Layer 2: 核心规则(按需,2000字)
  - 基本标注规则
  - 快速决策树
  - 常见问题FAQ

Layer 3: 专项深入(按需,各1500字)
  - 消歧规则(复杂场景)
  - 反思方法(迭代优化)
  - 特殊案例(罕见情况)
```

Agent Prompt模板:

```
def load_skill_progressive(task_type, complexity):
    """渐进载入SKILL文档"""

    # Layer 1: 总是加载总览
    skill_content = read_file("SKILL_03_实体构建.md")  # 500字

    # Layer 2: 根据任务加载核心规则
    if task_type == 'tagging':
        skill_content += read_file("SKILL_03a_实体标注.md")  #
+2000字

    # Layer 3: 根据复杂度加载专项
```

```
    if complexity == 'disambiguation':
        skill_content += read_file("SKILL_03b_实体消歧.md")  #
+1500字
    elif complexity == 'reflection':
        skill_content += read_file("SKILL_03c_实体标注反思.md")  #
+2000字

    return skill_content

# 示例:简单标注任务
skill = load_skill_progressive('tagging', 'simple')
# 载入: Layer 1 (500字) + Layer 2 (2000字) = 2500字

# 示例:复杂消歧任务
skill = load_skill_progressive('tagging', 'disambiguation')
# 载入: Layer 1 (500字) + Layer 2 (2000字) + Layer 3 (1500字) =
4000字

# vs 一次性加载全部: 10,000字
# 节省: 60% tokens
```

收益:

```
简单任务(80%的场景):
  载入2500字 vs 全量10,000字
  节省: 75% tokens
  成本: $0.025 vs $0.10

复杂任务(20%的场景):
  载入4000字 vs 全量10,000字
  节省: 60% tokens
  成本: $0.04 vs $0.10
```

```
平均节省:
  0.8 × 75% + 0.2 × 60% = 72%
```

3.3 导航与交叉引用

设计原则:文档间清晰链接

```
# SKILL_03a_实体标注.md

## 二、核心标注规则

### 规则3: 边界判断
- 复姓整体标注:`〖@司马迁〗`
- 单字名需上下文约束

**详见**: [SKILL_03b § 单字名处理](SKILL_03b_实体消歧.md#单字名处理)

### 规则5: 工具辅助
- 消歧工具:见[SKILL_03b](SKILL_03b_实体消歧.md)
- 反思流程:见[SKILL_03c](SKILL_03c_实体标注反思.md)
```

导航地图(在总览文档):

```
# SKILL_03_实体构建.md

## 文档导航

┌─────────────────────────────────────────┐
│  SKILL_03 实体构建(总览)                 │
├─────────────────────────────────────────┤
│  ├─ 03a 实体标注(核心规则)               │
│  │    ├─ 基本标注语法                    │
│  │    ├─ 人名识别模式                    │
│  │    └─ 边界判断 → 详见03b              │
```

```
│   ├─ 03b 实体消歧(专项)                          │
│   │    ├─ 短名消歧(4层启发式)                    │
│   │    ├─ 别名映射                               │
│   │    └─ 内联消歧语法                           │
│   ├─ 03c 实体标注反思(方法论)                    │
│   │    ├─ 按章反思                               │
│   │    ├─ 按类型反思                             │
│   │    └─ 全局一致性                             │
│   └─ 03d 渲染与发布(衔接)                        │
└──────────────────────────────────────────────────┘


## 快速入口

**新手**: 阅读03a(核心规则) + FAQ
**进阶**: 阅读03b(消歧) + 03c(反思)
**维护**: 检查版本历史 + 反思日志
```

四、Agent提示词模板化

4.1 提示词与SKILL的对应

关系:

```
SKILL文档 = 人类可读的方法论
Agent提示词 = 机器可执行的指令

SKILL文档 → [模板化] → Agent提示词
```

示例:

SKILL_03a § 规则2(人名识别模式):

```
### 规则2: 人名识别模式
- X曰 → X是人名
- 姓Y名X → X是人名
- 官职+人名 → 分别标注
```

对应的Agent Prompt模板:

```
ENTITY_TAGGING_PROMPT = """
你是一个古文实体标注专家。请按以下规则标注人名:

## 识别模式
1. 如果文本匹配"X曰",则X是人名,标注为〖@X〗
2. 如果文本匹配"姓Y名X",则X是人名,标注为〖@X〗
3. 如果文本匹配"官职+人名",分别标注:〖;官职〗〖@人名〗

## 示例
输入: 刘邦曰:"天下..."
输出: 〖@刘邦〗曰:"天下..."

输入: 丞相李斯
输出: 〖;丞相〗〖@李斯〗

## 任务
请标注以下文本中的人名:

{text}
"""
```

4.2 模板参数化

可变部分抽离为参数:

```
# 模板定义
ENTITY_TAGGING_PROMPT_TEMPLATE = """
你是一个古文实体标注专家。请按以下规则标注{entity_type}:

## 识别模式
{patterns}

## 示例
{examples}

## 边界规则
{boundary_rules}

## 任务
请标注以下文本中的{entity_type}:

{text}
"""

# 参数配置
ENTITY_TYPE_CONFIGS = {
    'person': {
        'entity_type': '人名',
        'patterns': [
            '如果文本匹配"X曰",则X是人名',
            '如果文本匹配"姓Y名X",则X是人名'
        ],
        'examples': [
            '输入: 刘邦曰 → 输出: 〖@刘邦〗曰'
        ],
        'boundary_rules': [
            '复姓整体标注:〖@司马迁〗',
            '单字名需上下文约束'
        ]
    },
    'place': {
```

```
        'entity_type': '地名',
        'patterns': [
            '如果X在地名词表中,标注为〖=X〗',
            '如果文本匹配"X郡"、"X县",则X是地名'
        ],
        # ...
    }
}

# 动态生成prompt
def generate_prompt(entity_type, text):
    config = ENTITY_TYPE_CONFIGS[entity_type]
    return ENTITY_TAGGING_PROMPT_TEMPLATE.format(
        entity_type=config['entity_type'],
        patterns='\n'.join(config['patterns']),
        examples='\n'.join(config['examples']),
        boundary_rules='\n'.join(config['boundary_rules']),
        text=text
    )
```

4.3 提示词版本管理

与SKILL同步版本:

```
# prompts/entity_tagging_v2.0.py

"""
Entity Tagging Prompt Template v2.0

对应SKILL: SKILL_03a v2.0
更新日期: 2025-01-15
变更: 精简为5条核心规则,拆分消歧到SKILL_03b
"""

ENTITY_TAGGING_PROMPT_V2_0 = """
```

```
[模板内容]
"""

# 版本历史
PROMPT_VERSION_HISTORY = {
    'v1.0': {
        'date': '2024-11-05',
        'skill_version': 'SKILL_03a v1.0',
        'rules_count': 10,
        'avg_tokens': 1500
    },
    'v1.5': {
        'date': '2024-12-10',
        'skill_version': 'SKILL_03a v1.5',
        'rules_count': 15,
        'avg_tokens': 2200
    },
    'v2.0': {
        'date': '2025-01-15',
        'skill_version': 'SKILL_03a v2.0',
        'rules_count': 5,  # 精简
        'avg_tokens': 800  # 渐进载入
    }
}
```

4.4 A/B测试与迭代

测试不同prompt版本的效果:

```
# tests/test_prompt_versions.py

def ab_test_prompts(test_data, prompt_v1, prompt_v2):
    """对比两个prompt版本的效果"""

    results = {
```

```
        'v1': {'precision': [], 'recall': [], 'cost': []},
        'v2': {'precision': [], 'recall': [], 'cost': []}
    }

    for sample in test_data:
        # 测试v1
        output_v1 = run_agent(prompt_v1, sample['text'])
        results['v1']['precision'].append(
            evaluate_precision(output_v1, sample['gold'])
        )
        results['v1']['cost'].append(count_tokens(prompt_v1))

        # 测试v2
        output_v2 = run_agent(prompt_v2, sample['text'])
        results['v2']['precision'].append(
            evaluate_precision(output_v2, sample['gold'])
        )
        results['v2']['cost'].append(count_tokens(prompt_v2))

    # 统计对比
    print("## A/B测试结果\n")
    for version in ['v1', 'v2']:
        print(f"### Prompt {version}")
        print(f"- 准确率: {np.mean(results[version]
['precision']):.2%}")
        print(f"- 召回率: {np.mean(results[version]
['recall']):.2%}")
        print(f"- 平均成本: {np.mean(results[version]['cost'])}
tokens")

# 输出示例:
# ## A/B测试结果
#
# ### Prompt v1.5
# - 准确率: 95.3%
# - 召回率: 89.7%
# - 平均成本: 2200 tokens
```

```
#
# ### Prompt v2.0
# - 准确率: 97.1% (+1.8%)
# - 召回率: 91.2% (+1.5%)
# - 平均成本: 800 tokens (-64%)
#
# 结论: v2.0在质量和成本上都优于v1.5,采纳v2.0
```

五、反思日志的系统化

5.1 日志结构标准化

统一模板:

```
# 第N轮XXX反思总结

## 元数据
- 反思轮次: N
- 反思对象: XXX(实体标注/事件年代/关系发现)
- 开始日期: YYYY-MM-DD
- 完成日期: YYYY-MM-DD
- 涉及章节: M/130
- Agent: Claude 3.5 Sonnet
- SKILL版本: vX.Y

## 整体情况

| 指标 | 数值 |
|------|------|
| 修正章节 | M/130 |
| 修正条目 | N条 |
| 主要问题 | [3-5类] |
| 平均置信度 | X% |

## 发现的新模式(K条)
```

```
### 模式1: [模式名称]
**问题**: [问题描述]
**示例**: [具体案例]
**影响**: [影响范围]
**修正方案**: [解决方案]
**频次**: [出现次数]

### 模式2: ...

## 规则更新建议

**新增规则**: [编号列表]
**废弃规则**: [编号列表]
**修订规则**: [编号列表]

## 统计数据

| 问题类型 | 数量 | 占比 |
|---------|------|------|
| [类型1] | N1 | X% |
| [类型2] | N2 | Y% |

## 典型案例(Top 10)

[详细描述10个代表性案例]

## 下一步行动

1. [行动1]
2. [行动2]
3. ...

## 附录

### 附录A: 完整修正清单
[所有修正的详细列表,可选]
```

```
### 附录B: 脚本输出
[Agent反思的原始输出,可选]
```

5.2 日志自动生成

```
# scripts/generate_reflection_report.py

def generate_reflection_report(round_num, corrections, patterns):
    """自动生成反思报告"""

    template = load_template("templates/reflection_report.md")

    # 统计数据
    stats = {
        'total_corrections': len(corrections),
        'affected_chapters': len(set(c['chapter'] for c in
corrections)),
        'problem_types': Counter(c['type'] for c in corrections),
        'avg_confidence': np.mean([c['confidence'] for c in
corrections])
    }

    # 提取模式
    patterns_summary = []
    for pattern in patterns:
        patterns_summary.append({
            'name': pattern['name'],
            'problem': pattern['description'],
            'examples': pattern['examples'][:3],  # Top 3示例
            'impact': pattern['affected_count'],
            'frequency': pattern['occurrences']
        })

    # 规则更新建议
```

```
    rule_updates = suggest_rule_updates(patterns)

    # 渲染模板
    report = template.format(
        round_num=round_num,
        date_start=corrections[0]['date'],
        date_end=corrections[-1]['date'],
        stats=stats,
        patterns=patterns_summary,
        rule_updates=rule_updates,
        top_cases=select_top_cases(corrections, n=10)
    )

    # 保存
    output_path = f"doc/reflection/round{round_num}_summary.md"
    with open(output_path, 'w') as f:
        f.write(report)

    print(f"反思报告已生成: {output_path}")
```

5.3 从日志到SKILL的自动提炼

```
# scripts/extract_rules_from_logs.py

def extract_rules_from_reflection_logs(log_files):
    """从多轮反思日志中提取规则"""

    all_patterns = []

    for log_file in log_files:
        patterns = parse_patterns_from_log(log_file)
        all_patterns.extend(patterns)

    # 按频次排序
    pattern_freq = Counter(p['name'] for p in all_patterns)
```

```
    # 高频模式(≥3次出现) → 转化为规则
    frequent_patterns = [
        p for p in pattern_freq.items() if p[1] >= 3
    ]

    # 生成规则草稿
    rules_draft = []
    for pattern_name, freq in frequent_patterns:
        pattern_details = [p for p in all_patterns if p['name'] ==
pattern_name][0]

        rule_draft = {
            'rule_name': pattern_name,
            'description': pattern_details['problem'],
            'solution': pattern_details['修正方案'],
            'examples': pattern_details['examples'],
            'frequency': freq,
            'confidence': 'high' if freq >= 5 else 'medium'
        }

        rules_draft.append(rule_draft)

    # 输出为Markdown
    output_rules_to_markdown(rules_draft, "doc/drafts/
new_rules_draft.md")

    return rules_draft

# 输出示例(doc/drafts/new_rules_draft.md):
# ## 新规则草稿(从反思日志提炼)
#
# ### 规则候选1: 别名链处理
# **来源**: 第1/2/3轮反思日志(频次:5)
# **问题**: 同一人物在不同章节有不同称谓
# **解决方案**: 建立entity_aliases.json,统一映射到规范名
# **示例**:
```

```
#    - 沛公 → 刘邦
#    - 汉王 → 刘邦
#    - 高祖 → 刘邦
# **置信度**: high(频次≥5)
#
# **建议**: 加入SKILL_03a v1.5作为规则11
```

六、质量度量与收敛判断

6.1 SKILL质量指标

```
def evaluate_skill_quality(skill_doc, test_data):
    """评估SKILL文档质量"""

    metrics = {}

    # 指标1: 规则覆盖率
    total_cases = len(test_data)
    covered_cases = sum(
        1 for case in test_data
        if case_covered_by_skill(case, skill_doc)
    )
    metrics['coverage'] = covered_cases / total_cases

    # 指标2: 规则冲突率
    rules = extract_rules(skill_doc)
    conflicts = detect_rule_conflicts(rules)
    metrics['conflict_rate'] = len(conflicts) / len(rules)

    # 指标3: 文档可读性
    metrics['readability'] = {
        'word_count': count_words(skill_doc),
        'rule_count': len(rules),
        'avg_rule_length': np.mean([len(r) for r in rules]),
```

```
        'nesting_depth': max_nesting_depth(skill_doc)
    }

    # 指标4: Agent执行成功率
    success_count = 0
    for case in test_data:
        output = run_agent_with_skill(skill_doc, case['input'])
        if evaluate(output, case['gold']):
            success_count += 1
    metrics['agent_success_rate'] = success_count / total_cases

    return metrics

# 输出示例:
# {
#   'coverage': 0.92,  # 92%案例被规则覆盖
#   'conflict_rate': 0.03,  # 3%规则有冲突
#   'readability': {
#       'word_count': 2340,
#       'rule_count': 15,
#       'avg_rule_length': 156,
#       'nesting_depth': 2
#   },
#   'agent_success_rate': 0.95  # 95%案例Agent执行成功
# }
```

6.2 收敛判断标准

何时停止迭代?

```
def check_skill_convergence(history):
    """判断SKILL是否收敛"""

    # 标准1: 新发现模式减少
    recent_patterns = [h['new_patterns'] for h in history[-3:]]
```

```
    if all(p < 5 for p in recent_patterns):
        print("✓ 新模式发现减少(<5/轮)")

    # 标准2: 修正数量递减
    recent_corrections = [h['corrections'] for h in history[-3:]]
    if is_decreasing(recent_corrections):
        print("✓ 修正数量递减")

    # 标准3: 准确率稳定
    recent_accuracy = [h['accuracy'] for h in history[-3:]]
    if np.std(recent_accuracy) < 0.01:  # 标准差<1%
        print("✓ 准确率稳定")

    # 标准4: 规则数量稳定
    recent_rule_counts = [h['rule_count'] for h in history[-3:]]
    if len(set(recent_rule_counts)) == 1:  # 连续3轮规则数不变
        print("✓ 规则数量稳定")

    # 综合判断
    converged = (
        all(p < 5 for p in recent_patterns) and
        is_decreasing(recent_corrections) and
        np.std(recent_accuracy) < 0.01 and
        len(set(recent_rule_counts)) == 1
    )

    return converged

# 示例:
# 第1轮: 新模式25, 修正1247, 准确率90%, 规则10
# 第2轮: 新模式12, 修正456, 准确率93%, 规则13
# 第3轮: 新模式8, 修正234, 准确率95%, 规则15
# 第4轮: 新模式4, 修正89, 准确率96%, 规则15
# 第5轮: 新模式2, 修正34, 准确率96.5%, 规则15
#
# check_skill_convergence(history) → True
# 结论: SKILL已收敛,可以发布v2.0
```

七、跨项目SKILL迁移的优化

7.1 SKILL的可迁移性设计

设计原则:

```
高可迁移性规则:
  - 通用模式(文言文共性)
  - 抽象方法(不依赖具体数据)
  - 参数化配置(朝代/文体可替换)

低可迁移性规则:
  - 项目特定(史记特有人名)
  - 硬编码数据(章节映射)
  - 临时解决方案(workaround)
```

标记可迁移性:

```
# SKILL_03a_实体标注.md

## 二、核心标注规则

### 规则2: 人名识别模式
**可迁移性**: ★★★★★ (通用)
**适用范围**: 所有文言文

- X曰 → X是人名
- 姓Y名X → X是人名

### 规则6: 短名消歧
**可迁移性**: ★★★★☆ (高度通用,需调整参数)
**适用范围**: 纪传体史书
```

迁移说明:
- 4层启发式策略通用
- CHAPTER_STATE映射需根据目标史书调整
- 示例:汉书需更新为100+章的映射

规则11: 别名映射
可迁移性: ★★★☆☆ (中等,需重建数据)
适用范围: 任何有人物别名的文本

迁移说明:
- 映射机制通用
- entity_aliases.json需为目标史书重新构建
- 示例:汉书需建立新的别名表(预计800+条)

7.2 SKILL差异文档

迁移时生成差异清单:

```
# doc/transfer/史记SKILL→汉书SKILL差异清单.md

## 一、直接复用(80%)

| SKILL | 复用度 | 说明 |
|-------|--------|------|
| 00-META_反思.md | 100% | 完全通用 |
| 00-META_质量控制.md | 100% | 完全通用 |
| SKILL_03a_实体标注.md(核心规则1-5) | 95% | 微调示例 |

## 二、需要调整(15%)

| SKILL | 复用度 | 调整内容 |
|-------|--------|---------|
| SKILL_03b_实体消歧.md | 70% | CHAPTER_STATE映射(120章) |
| SKILL_04c_事件年代推断.md | 60% | 年号体系(汉代年号) |
```

```
## 三、需要重建(5%)

| 数据文件 | 复用度 | 说明 |
|---------|--------|------|
| entity_aliases.json | 20% | 部分重叠人物(刘邦/项羽等),其余需重建 |
| disambiguation_map.json | 10% | 章节不同,需重新构建 |

## 四、新增内容

| 类型 | 说明 |
|------|------|
| 年号解析 | 汉代大量使用年号纪年,史记时代较少 |
| 表格文献 | 汉书的"表"为二维矩阵,史记为时间线 |
| 西汉官制 | 三公九卿体系,需扩展官职词表 |
```

八、总结

核心要点

1. **SKILL从实践中来** — 冷启动→试点→反思→总结→抽象
2. **持续演化** — v0.1→v1.0→v1.5→v2.0,不断优化
3. **结构化拆分** — 超过5000字/15条规则就拆分
4. **渐进载入** — 三层模型(总览→核心→专项),节省70%+ tokens
5. **版本管理** — 记录演化历史,支持回溯对比
6. **模板化prompt** — SKILL与Agent提示词同步迭代
7. **日志系统化** — 标准化模板,自动生成报告
8. **质量度量** — 覆盖率/冲突率/成功率/收敛判断
9. **迁移优化** — 标记可迁移性,生成差异清单

哲学洞察

> **"好东西都是总结出来的"**

- SKILL不是事先设计的完美方案,而是**从失败中学习的结果**
- 反思日志是**知识的原材料**,SKILL是**提炼后的精华**
- 版本演化体现**认知的深化**:从具体案例→一般规律
- 结构化设计支持**渐进学习**:新手看核心,专家看全貌
- 模板化使**知识可复制**:SKILL→Prompt→自动化

与其他元技能的关系

- ← **反思**: SKILL演化的核心驱动力
- ← **冷启动**: SKILL的v0.1诞生于冷启动
- ← **迭代工作流**: SKILL在迭代中不断积累
- ← **可读性**: SKILL本身必须人类可读
- ← **知识压缩**: SKILL是对经验的压缩
- → **技能迁移**: 优化后的SKILL更易迁移

SKILL优化与演化是项目"自我学习"能力的体现 — 通过系统化总结,项目不仅积累数据,更积累方法论本身。

附录:检查清单

SKILL文档质量检查清单

内容质量:
- [ ] 规则来源明确(从哪次反思/试点总结?)
- [ ] 规则可执行(Agent能理解并执行?)
- [ ] 示例充分(每条规则至少1个示例)
- [ ] 例外明确(什么情况下规则不适用?)
- [ ] 质量标准量化(准确率/召回率目标)

结构质量:
- [ ] 文档长度<5000字(否则考虑拆分)
- [ ] 规则数量<15条(否则按类型拆分)
- [ ] 嵌套层次<3层(否则拆分为独立文档)
- [ ] 有导航地图(快速定位)
- [ ] 有交叉引用(链接到相关SKILL)

可维护性:
- [ ] 有版本历史(记录演化轨迹)
- [ ] 有变更日志(v1→v2改了什么?)
- [ ] 有可迁移性标记(★评级)
- [ ] 有对应的prompt模板
- [ ] 有A/B测试结果(验证改进效果)

可用性:
- [ ] 新手10分钟能上手(核心规则简洁)
- [ ] 专家能快速查询(有FAQ/索引)
- [ ] Agent能正确理解(测试成功率>90%)
- [ ] 支持渐进载入(三层模型)

元技能07: 可读性 — 人类优先的数据格式设计

一、核心思想

> **"数据的第一受众是人类,第二受众才是机器"**

可读性（Readability）是指数据、代码、文档在人类直接阅读时的友好程度。在AI驱动的知识工程中,可读性不是奢侈品,而是**生产力的倍增器**。

核心原则

1. **人类优先**:格式设计首先考虑人类阅读,其次考虑机器解析
2. **Git友好**:所有数据文件应易于版本控制(文本格式、行式结构)
3. **调试透明**:中间结果可视化,黑箱流程文档化
4. **渐进降级**:复杂数据提供多层次视图(摘要→详情→原始)
5. **领域语言**:使用领域专家熟悉的符号和结构

哲学

传统数据库/XML模式:
- 设计给机器看(binary/紧凑格式)
- 人类需要专门工具查看
- Git diff不可读

本项目的可读性优先模式:
- 设计给人类看(Markdown/美化JSON)
- 机器也能轻松解析(标准格式)
- Git diff就是最好的审查工具

收益:
- 降低学习门槛(新人可直接阅读数据)

- 提高调试效率(问题一目了然)
- 促进协作(review更高效)
- 知识可见(数据本身就是文档)

二、本项目的可读性实践

2.1 标注格式:Token优于XML

问题:传统XML标注的可读性灾难

```
<!-- XML标注示例:字符数膨胀3-5倍 -->
<person>黄帝</person>者,<person>少典</person>之子,
姓<clan>公孙</clan>,名曰<person>轩辕</person>。
生而神灵,弱而能言,幼而徇齐,长而敦敏,成而登天。
```

问题:
- 标签淹没原文(约50%字符是标签)
- 肉眼难以流畅阅读原文
- Git diff噪音严重(`<person>` / `</person>` 重复出现)
- 嵌套处理复杂(如 `<person><clan>...</clan></person>`)

解决方案:轻量级Token标注

```
〖@黄帝〗者,〖@少典〗之子,姓〖&公孙〗,名曰〖@轩辕〗。
生而神灵,弱而能言,幼而徇齐,长而敦敏,成而登天。
```

设计要点:

维度	XML/HTML	〖〗Token	优势
字符膨胀	+300%~500%	+30%~50%	Token紧凑5-10倍
阅读流畅度	标签淹没原文	标记轻量,原文清晰	人眼可直接阅读标注文本

维度	XML/HTML	〖〗Token	优势
Git diff	噪音严重	简洁明了	review效率10倍提升
嵌套处理	支持任意嵌套	扁平化策略	避免复杂性
渲染无关性	与HTML耦合	可转换为任意格式	同一份标注多用途

符号选择原则:
- 使用Unicode罕见符号(`〖〗`,中文不常用)
- 避免与原文冲突(`<>[]{}()`文言文可能出现)
- 单字符前缀区分类型(`@`人名、`=`地名、`;`官职...)
- 视觉上轻量(不喧宾夺主)

可读性对比实例

场景:标注一段80字的原文

XML版本(272字符,+240%):

```
<person>高祖</person>,<person>沛</person>丰邑中阳里人也,
姓<clan>刘</clan>氏,字<person>季</person>。父曰<person>太公</person>,
母曰<person>刘媪</person>。其先<person>刘累</person>,
<time>夏后</time>时为<official>御龙氏</official>。
```

Token版本(108字符,+35%):

```
〖@高祖〗,〖=沛〗丰邑中阳里人也,姓〖&刘〗氏,字〖@季〗。
父曰〖@太公〗,母曰〖@刘媪〗。其先〖@刘累〗,
〖'夏后〗时为〖;御龙氏〗。
```

纯文本版本(80字符,基准):

```
高祖,沛丰邑中阳里人也,姓刘氏,字季。
父曰太公,母曰刘媪。其先刘累,夏后时为御龙氏。
```

阅读体验测试(5人小组):
- XML版本:需逐字阅读,视线被标签干扰,平均阅读时间45秒
- Token版本:可流畅阅读,轻微停顿识别类型,平均阅读时间18秒
- 纯文本:最流畅,平均阅读时间15秒

结论:Token版本在"可读性"和"结构化"之间取得最优平衡。

2.2 数据格式:Markdown + JSON双轨制

原则:人读+机读同步维护

所有核心数据同时提供两种格式:

```
kg/events/data/
├── 001_五帝本纪_事件索引.md          # 人读:表格+Markdown
├── event_relations.json               # 机读:结构化JSON
├── event_relations_summary.md         # 人读:统计摘要
└── wars.json / 战争事件索引.md        # 双格式
```

格式1:Markdown表格(人读优先)

示例:`001_五帝本纪_事件索引.md`

```
# 001 五帝本纪 - 事件索引

## 一、概览表格

| 事件ID | 事件名 | 事件类型 | 主要人物 | 地点 | CE年份 | 确定性 |
|--------|--------|----------|----------|------|--------|--------|
| 001_001 | 黄帝战蚩尤 | 战争 | 黄帝, 蚩尤 | 涿鹿 | ~-2500? | 低 |
| 001_002 | 黄帝制历法 | 文化 | 黄帝 | - | ~-2500? | 低 |
```

```
| 001_003 | 颛顼即位 | 继位 | 颛顼 | - | ~-2400 | 中 |

**统计信息**:
- 事件总数: 24
- 事件类型分布: 战争(3), 继位(5), 政治(8), 文化(4), 家族(4)
- CE年份覆盖: 24/24 (100%)

## 二、详细记录

### 001_001 黄帝战蚩尤

- **事件类型**: 战争
- **主要人物**: 黄帝, 蚩尤
- **地点**: 涿鹿之野
- **时间字段**: (无明确纪年)
- **CE年份**: ~-2500?
- **确定性**: 低(神话时代,缺乏可靠纪年)
- **描述**: 黄帝与蚩尤战于涿鹿之野,擒杀蚩尤。此后天下大治。
- **原文引用**: "轩辕乃修德振兵,治五气,艺五种,抚万民,度四方,
  教熊罴貔貅貙虎,以与炎帝战于阪泉之野。三战,然后得其志。
  蚩尤作乱,不用帝命。于是黄帝乃征师诸侯,与蚩尤战于涿鹿之野,
  遂禽杀蚩尤。"
- **相关事件**: [001_002 黄帝制历法], [013_005 黄帝世系]

---

### 001_002 黄帝制历法
...
```

优势:
- **直接阅读**:无需任何工具,文本编辑器即可打开
- **层次清晰**:概览表格(快速浏览) + 详细记录(深入了解)
- **搜索友好**:Ctrl+F可全文搜索
- **Git友好**:行式结构,diff清晰
- **可渲染**:Markdown可转换为HTML/PDF/LaTeX

格式2:美化JSON(机读+人审查)

示例: `event_relations.json`


```
{
  "_metadata": {
    "generated_at": "2026-03-17T14:32:10Z",
    "generator": "extract_event_relations.py v3.2",
    "total_relations": 7652,
    "relation_types": {
      "sequel": 1623,
      "causal": 407,
      "part_of": 107,
      "opposition": 50,
      "cross_ref": 294,
      "co_person": 867,
      "co_location": 542,
      "concurrent": 173,
      "cross_chapter_causal": 352
    }
  },
  "relations": [
    {
      "id": "R_001_001→001_002",
      "type": "sequel",
      "source_event": "001_001",
      "target_event": "001_002",
      "source_name": "黄帝战蚩尤",
      "target_name": "黄帝制历法",
      "confidence": 0.92,
      "rationale": "战争胜利后,黄帝开始制定历法,建立秩序",
      "extraction_method": "llm",
      "_note": "LLM推理的章内延续关系"
    },
    {
      "id": "R_001_005→013_008",
      "type": "cross_ref",
```

```
      "source_event": "001_005",
      "target_event": "013_008",
      "source_name": "尧禅让于舜",
      "target_name": "尧传位于舜",
      "confidence": 0.98,
      "shared_persons": ["尧", "舜"],
      "extraction_method": "auto",
      "_note": "自动检测的跨章互见(本纪与世表对同一事件的记载)"
    }
  ]
}
```

设计要点:
- `indent=2`:美化格式,层次清晰
- `ensure_ascii=False`:保留中文,无需Unicode转义(`\u4e2d\u6587`)
- **元数据在前**:先看统计信息,再看详细数据
- **字段全称**:不缩写(`source_event`而非`se`)
- **注释字段**:`_note`/`_comment`帮助理解
- **冗余信息**:`source_name`虽可从event_id查询,但冗余保存便于人类阅读

反例(不可读的JSON):

```
{"r":[{"i":"R1","t":"sq","s":"001_001","tgt":"001_002","c":0.92}]}
```

问题:压缩格式,字段缩写,人类无法直接理解。

2.3 分类树:Markdown可视化

问题:本体工程的"看不见的知识"

传统本体开发:
- 本体定义在OWL/RDF文件中(机器格式)
- 需要Protégé等专门工具查看
- 层次结构不直观
- 协作困难(需统一工具)

解决方案:Markdown分类树

示例: `kg/entities/data/biology/biology_taxonomy.md`

```
# 生物实体分类树 (388种)

## 完整树形结构

生物 (LivingThing)
├── 动物 (Animal)
│   ├── 家畜 (Domestic)                [71种/248次]
│   │   ├── 马属 (Equine)              [33种/162次]
│   │   │   ├── 马 (102)
│   │   │   ├── 驹 (9)
│   │   │   ├── 骝 (6)
│   │   │   ├── 骊 (5)
│   │   │   ├── 骅骝 (4)
│   │   │   ├── 千里马 (3)
│   │   │   └── ... (27种)
│   │   ├── 牛属 (Bovine)              [12种/38次]
│   │   │   ├── 牛 (24)
│   │   │   ├── 牺牛 (3)
│   │   │   ├── 犁牛 (2)
│   │   │   └── ...
│   │   ├── 犬属 (Canine)              [8种/23次]
│   │   ├── 猪属 (Porcine)             [5种/12次]
│   │   └── 家禽 (Poultry)             [13种/13次]
│   ├── 野兽 (WildBeast)               [39种/132次]
│   │   ├── 大型猛兽 (LargeCarnivore)
│   │   │   ├── 虎 (18)
│   │   │   ├── 豹 (12)
│   │   │   ├── 熊 (9)
│   │   │   └── 狼 (7)
│   │   └── 小型动物 (SmallAnimal)
│   │       ├── 兔 (8)
│   │       ├── 狐 (6)
│   │       └── ...
```

```
│   ├── 鸟类 (Bird)                    [43种/84次]
│   │   ├── 神鸟 (MythicalBird)
│   │   │   ├── 凤凰 (12)
│   │   │   ├── 鸾 (5)
│   │   │   └── 鹓鶵 (3)
│   │   ├── 猛禽 (RaptorBird)
│   │   │   ├── 鹰 (11)
│   │   │   ├── 隼 (4)
│   │   │   └── 鹗 (3)
│   │   └── 常见鸟类 (CommonBird)
│   │       ├── 鸿 (9)
│   │       ├── 雁 (7)
│   │       └── 鸦 (5)
│   ├── 鱼类水族 (Aquatic)             [25种/63次]
│   ├── 虫类 (Insect)                  [21种/25次]
│   └── 神话动物 (MythicalAnimal)      [20种/47次]
│       ├── 龙 (18)
│       ├── 麒麟 (8)
│       ├── 神龟 (5)
│       └── ...
├── 植物 (Plant)                       [95种/141次]
│   ├── 树木 (Tree)
│   │   ├── 桑 (12)
│   │   ├── 柳 (8)
│   │   └── ...
│   ├── 果实 (Fruit)
│   ├── 蔬菜 (Vegetable)
│   └── 花卉 (Flower)
└── 微生物/其他 (Microbe)              [7种/9次]

## 统计摘要

| 一级类别 | 种类数 | 出现次数 | 占比 |
|---------|--------|---------|------|
| 动物 | 219 | 599 | 56.5% |
| 植物 | 95 | 141 | 24.5% |
| 神话生物 | 67 | 218 | 17.3% |
```

```
| 微生物 | 7 | 9 | 1.8% |
| **总计** | **388** | **967** | **100%** |

## 设计说明

### 分类原则
1. **功能优先**: 家畜/野兽/家禽按人类用途分类
2. **现实主义**: 神话动物单独分类(龙/麒麟/凤凰)
3. **古代认知**: 按史记时代的分类观念(非现代生物学)

### 歧义处理
- "龙"姓人物 vs 神话生物"龙" → 按上下文消歧
- "马"作为人名(司马迁) vs 动物"马" → 标注时已区分

### 未来扩展
- [ ] 添加拉丁学名映射(现代生物学对照)
- [ ] 标注栖息地信息(用于地理分析)
- [ ] 建立食物链关系(用于生态分析)
```

优势:
- **一目了然**:树形结构直观展示层次关系
- **统计可见**:每个节点显示数量(种类/出现次数)
- **无需工具**:任何文本编辑器/浏览器可查看
- **Git友好**:增删改都有清晰diff
- **协作友好**:非技术专家可直接review和建议

对比:OWL格式的不可读性

```
# ontology.ttl (传统RDF/OWL格式)
:Animal rdf:type owl:Class .
:Domestic rdf:type owl:Class ;
          rdfs:subClassOf :Animal .
:Equine rdf:type owl:Class ;
```

```
        rdfs:subClassOf :Domestic .
:马 rdf:type :Equine ;
   :occurrenceCount 102 .
```

问题:
- 需要理解RDF/OWL语法
- 层次关系不直观(需人脑解析`rdfs:subClassOf`)
- 无统计摘要
- diff时难以理解修改意图

2.4 紫色编号:可引用的段落

理念:Doug Engelbart的Purple Numbers

历史背景:
- Doug Engelbart(鼠标发明者)提出Purple Numbers概念
- 为文档的每个段落分配唯一、稳定、可引用的编号
- 目标:精确定位和引用文档的任意部分

本项目实现:`doc/spec/紫色编号设计.md`

设计方案

HTML渲染时自动添加段落编号:

```
<p>
  <span class="para-num" id="para-005">#005</span>
  黄帝者,少典之子,姓公孙,名曰轩辕。
</p>

<style>
.para-num {
    background-color: #F3E5F5;  /* 淡紫色背景 */
    color: #7B1FA2;             /* 紫色文字 */
    border: 1px solid #CE93D8;  /* 淡紫色边框 */
    padding: 1px 4px;
```

```
    border-radius: 3px;
    font-size: 0.75em;
    font-family: 'Consolas', 'Monaco', monospace;
    cursor: pointer;
    user-select: none;
}

.para-num:hover {
    background-color: #E1BEE7;  /* 悬停时加深 */
}
</style>
```

交互功能:
- **点击编号** → 复制段落链接(`#para-005`)
- **URL分享** → `https://example.com/001_五帝本纪.html#para-005`
- **锚点跳转** → 打开URL自动滚动到对应段落

使用场景

场景1:问题反馈

```
用户: "001 五帝本纪 #005段有个typo,「公孙」应为「姬」"
开发者: 直接跳转到#005,定位问题,1秒解决
```

场景2:学术引用

```
论文: "根据史记高祖本纪#042所述,刘邦斩白蛇起义..."
读者: 点击链接,直达原文具体段落
```

场景3:协作讨论

```
Review: "事件索引#013的年份标注有问题,请核查"
→ 精确定位,无需"第三段"、"那个关于XX的部分"等模糊描述
```

2.5 中间输出:流水线可见性

原则:每个处理阶段保存输出

反例(黑箱流水线):

```
# ❌ 不可读:看不到中间过程
result = step1(data)
result = step2(result)
result = step3(result)
save(result, 'final.json')

# 问题出在哪个阶段?不知道!
```

正例(透明流水线):

```
# ✅ 可读:每步可见
result1 = step1(data)
save(result1, 'tmp/stage1_raw_extraction.json')
save_markdown(result1, 'tmp/stage1_preview.md')  # 人读预览

result2 = step2(result1)
save(result2, 'tmp/stage2_filtered.json')
save_markdown(result2, 'tmp/stage2_preview.md')

result3 = step3(result2)
save(result3, 'data/final.json')
save_markdown(result3, 'data/final_index.md')
```

案例:战争事件提取的五阶段输出

文件结构:

```
kg/events/tmp/
├── wars_stage1_identified.json       # 阶段1:识别候选(736条)
├── wars_stage1_preview.md            # 人读预览
├── wars_stage2_grouped.json          # 阶段2:按名称分组(308组)
├── wars_stage2_preview.md
├── wars_stage3_merged.json           # 阶段3:合并多源记载(264条)
├── wars_stage3_preview.md
├── wars_stage4_enriched.json         # 阶段4:提取详情(264条)
└── wars_stage4_preview.md

kg/events/data/
├── wars.json                          # 最终输出(机读)
└── 战争事件索引.md                    # 最终输出(人读)
```

Markdown预览示例(`wars_stage2_preview.md`):

```
# 战争事件提取 - 阶段2预览(分组)

## 统计
- 原始候选: 736条
- 分组后: 308组
- 单源事件: 244组(无需合并)
- 多源事件: 64组(需合并)

## 多源事件示例

### 组1: 长平之战
- 来源数: 3
- 章节: 005_秦本纪, 015_六国年表, 043_赵世家
- 需合并: 是

**005_秦本纪**:
- 事件ID: 005_042
- 年份: -260
- 描述: 秦攻赵,战于长平,斩首四十五万...
```

```
**015_六国年表**:
- 事件ID: 015_108
- 年份: -260
- 描述: 赵括代廉颇,秦将白起大破之...

**043_赵世家**:
- 事件ID: 043_067
- 年份: -260
- 描述: 长平之役,赵军四十万降,秦将白起尽坑之...

**合并策略**: 取最详细的描述(043),保留所有来源章节

---

### 组2: 巨鹿之战
...
```

人工审查流程:
1. 运行 `python extract_wars.py --stage 2`
2. 打开 `wars_stage2_preview.md`
3. 检查分组是否合理("长平之战"和"长平之役"应合并)
4. 如有问题,修正分组逻辑,重新运行
5. 确认无误后,执行 `--stage 3`

2.6 辅助文档:知识的可见性

原则:将隐性知识显性化

传统问题:
- 代码注释分散(每个文件各自说明)
- 设计决策不可见(为什么这样设计?)
- 演化历史遗失(v1.0→v2.5经历了什么?)

本项目方案:专门的设计文档目录

```
doc/
├── spec/                                # 规范文档
│   ├── 标注格式规范.md                  # 18类实体标注规范
│   ├── 紫色编号设计.md                  # 段落编号系统
│   ├── 事件索引格式.md                  # 事件索引文件格式
│   └── 关系类型定义.md                  # 9种事件关系定义
├── analysis/                            # 分析报告
│   ├── 词频统计表.md
│   ├── 标注覆盖率报告.md
│   └── 差异检测报告.md
├── events/                              # 事件工程文档
│   ├── 第一轮事件年代反思总结.md     # 反思过程记录
│   ├── 第二轮事件年代反思总结.md
│   ├── ...
│   └── 事件关系发现方法论.md
└── publications/                        # 方法论论文
    ├── agentic_ontology_101.md          # Agent驱动的本体工程
    └── 史记知识图谱构建方法.md
```

案例:标注体系演化文档

文件:`skills/00-META_标注体系设计.md`(§VII 演化记录)

```
## 七、演化记录

### v1.0 → v2.0 (2024-11-05)

**变更**:
- 拆分: 朝代 → 邦国(`'`) + 氏族(`&`)
- 拆分: 官职 → 官职(`;`) + 身份(`#`)
- 新增: 天文(`!`) 独立类型

**动机**:
- "齐"既是邦国(齐国)又是朝代(齐朝),语义冲突
- "丞相"是官职,"诸侯"是身份,粒度不一致
```

```
- "岁星"、"日食"等天文现象需独立类型

**迁移脚本**: `scripts/migrate_v1_to_v2.py`

**影响范围**:
- 需重新标注章节: 23篇(涉及朝代/官职密集章节)
- 预计工作量: 8小时

---

### v2.0 → v2.5 (2026-03-10)

**变更**:
- 重命名: 动植物(`+`) → 生物(`+`,名称调整)
- 新增: 数量(`$`), 典籍(`{`), 礼仪(`:`), 刑法(`[`), 思想(`_`)

**动机**:
- 高频词分析发现大量未标注的数量词("三万人"、"千里")
- 礼仪词汇("封禅"、"宗庙")在文化分析中重要
- 刑法词汇("腰斩"、"族诛")在法制史研究中关键

**迁移**: 向后兼容(新增类型,不修改已有)

**影响范围**: 无需重标注,仅增量标注新类型

---

### v2.5 → v3.0 (计划中)

**候选变更**:
- 可能拆分: 人名(`@`) → 历史人名(`@`) + 神话人物(`?@`)
- 可能新增: 法律文书(`[@`), 军事单位(`$军`)

**讨论中**: 是否粒度过细?是否会造成标注负担?
```

价值:

- 新人快速理解"为什么是18类,而不是15类或20类"

- 避免重复讨论(历史决策已记录)
- 指导未来演化(v3.0可参考v1→v2的经验)

2.7 代码可读性:注释即文档

原则:代码是给人读的

反例:

```
def f(a, b):
    r = []
    for x in a:
        if x[1] in b:
            r.append((x[0], x[2], b[x[1]]))
    return r
```

问题:变量名无意义,逻辑不清晰,无注释。

正例:

```
def extract_event_relations(events, relation_types):
    """
    从事件列表中提取指定类型的关系

    Args:
        events: 事件列表,每个事件包含(id, type, persons, locations,
year)
        relation_types: 需要提取的关系类型集合(如{'sequel', 'causal'})

    Returns:
       关系列表,每条关系为(source_event_id, relation_type,
target_event_id)

    算法:
```

```
        遍历每个事件,检查其类型是否在relation_types中,
        如果匹配,则根据事件属性构造关系三元组。
    """
    relations = []

    for event in events:
        event_id = event['id']
        event_type = event['type']
        persons = event['persons']

        # 仅处理指定类型的关系
        if event_type not in relation_types:
            continue

        # 根据事件类型构造关系
        # 例如:继位事件 → (前任, successor, 继任者)
        if event_type == 'succession':
            predecessor = persons[0]  # 假设第一个是前任
            successor = persons[1]     # 第二个是继任者
            relations.append((predecessor, 'successor',
successor))

    return relations
```

改进要点:
- **函数名**:描述性强(`extract_event_relations` 而非 `f`)
- **参数名**:自解释(`events`, `relation_types` 而非 `a`, `b`)
- **Docstring**:说明用途、参数、返回值、算法
- **行内注释**:解释关键逻辑(`# 仅处理指定类型的关系`)

设计决策注释

在关键决策处添加"为什么"注释:

```
# 设计决策: 跨章节关系仅用自动方法计算
# 原因: 跨章对数量巨大(410万对),LLM成本过高($4.1万)
#       自动方法(实体共现/时间对齐)精度已达98%,足够可靠
# 参考: doc/events/关系提取成本效益分析.md
if all_events[a]["chapter_id"] == all_events[b]["chapter_id"]:
    continue  # 章内由LLM处理,提供更高语义精度

# 设计决策: 名称相似度阈值设为0.6
# 原因: A/B测试结果(见tests/test_similarity_threshold.py):
#       - 0.5: 召回95%, 精度78%(噪音过多)
#       - 0.6: 召回89%, 精度94%(最优平衡)
#       - 0.7: 召回72%, 精度98%(遗漏过多)
if name_similarity(event_a['name'], event_b['name']) >= 0.6:
    # 认为是同一事件的不同记载
    ...
```

价值:
- 未来维护者理解设计意图
- 避免"优化"掉关键约束(不知道为什么是0.6,改成0.5,精度下降)
- 快速定位相关文档(参考链接)

2.8 错误消息:调试友好

原则:错误消息要告诉"怎么修"

反例:

```
raise ValueError("Invalid event")
# 输出: ValueError: Invalid event
# 问题: 什么event?哪里invalid?如何修复?
```

正例:

```
def validate_event(event):
    """验证事件格式"""

    # 检查必填字段
    required_fields = ['id', 'name', 'type', 'persons',
'description']
    missing = [f for f in required_fields if f not in event]
    if missing:
        raise ValueError(
            f"事件验证失败: 缺少必填字段 {missing}\n"
            f"事件ID: {event.get('id', 'UNKNOWN')}\n"
            f"文件: {event.get('_source_file', 'UNKNOWN')}\n"
            f"修复建议: 在事件记录中添加字段 {missing}"
        )

    # 检查年份合理性
    ce_year = event.get('ce_year')
    if ce_year and (ce_year > 0 or ce_year < -3000):
        raise ValueError(
            f"事件验证失败: CE年份不合理 ({ce_year})\n"
            f"事件: {event['id']} - {event['name']}\n"
            f"文件: {event.get('_source_file')}\n"
            f"修复建议: 史记年代应在-3000到-1之间,请检查:\n"
            f"  1. 是否遗漏负号(-500误写为500)\n"
            f"  2. 是否年份推断错误(参考章节纪元范围)\n"
            f"  3. 如为神话时代,应使用~-XXXX?格式标注不确定性"
        )

    # ... 更多检查

    return True  # 验证通过
```

错误输出示例:

```
ValueError: 事件验证失败: CE年份不合理 (500)
事件: 047_012 - 孔子卒
文件: kg/events/data/047_仲尼弟子列传_事件索引.md
修复建议: 史记年代应在-3000到-1之间,请检查:
  1. 是否遗漏负号(-500误写为500)
  2. 是否年份推断错误(参考章节纪元范围)
  3. 如为神话时代,应使用~-XXXX?格式标注不确定性
```

改进要点:
- **具体信息**:事件ID、文件名(快速定位)
- **错误原因**:年份不合理(而非"Invalid")
- **修复建议**:列举3种可能原因和修复方法
- **领域知识**:史记年代应为负数(教育性)

三、可读性设计清单

3.1 数据文件设计

设计新数据文件时检查:

- [ ] **双格式输出**:同时提供JSON(机读) + Markdown(人读)
- [ ] **JSON美化**: `indent=2` , `ensure_ascii=False`
- [ ] **元数据字段**: `_metadata` 包含生成时间/工具/统计
- [ ] **字段全称**:不缩写,使用描述性名称
- [ ] **注释字段**: `_note` 解释特殊设计
- [ ] **示例文档**: `README.md` 包含数据格式说明和示例
- [ ] **统计摘要**:Markdown版本包含count/分布等统计
- [ ] **Git友好**:行式结构,避免单行过长(>120字符)

3.2 标注系统设计

设计新标注类型时检查:

- [ ] **符号选择**:Unicode罕见字符,视觉轻量
- [ ] **类型前缀**:单字符区分(`@` / `=` / `;` 等)
- [ ] **阅读测试**:5人小组测试阅读流畅度
- [ ] **Git diff测试**:修改一处,diff是否清晰
- [ ] **嵌套策略**:定义嵌套处理规则(扁平化/外层优先)
- [ ] **转换工具**:提供Token→JSON/XML/HTML转换脚本
- [ ] **演化记录**:在SKILL文档中记录版本历史

3.3 代码可读性

编写脚本时检查:

- [ ] **函数名**:动词开头,描述性强(`extract_X`, `validate_Y`)
- [ ] **变量名**:避免单字母(除i/j/k循环变量)
- [ ] **Docstring**:说明用途/参数/返回值/算法
- [ ] **行内注释**:解释"为什么"而非"做什么"
- [ ] **设计决策注释**:关键阈值/参数的选择原因
- [ ] **错误消息**:包含具体信息+修复建议
- [ ] **示例用法**:Docstring或README包含代码示例

3.4 中间输出设计

设计处理流水线时检查:

- [ ] **分阶段存储**:每个处理阶段保存输出
- [ ] **人读预览**:生成Markdown预览(前10个样本)
- [ ] **统计信息**:每阶段输出count/过滤率等统计
- [ ] **断点续跑**:支持 `--stage N` 从指定阶段开始

- [ ] **diff工具**:提供阶段间diff查看(如stage2 vs stage3)
- [ ] **错误收集**:单独保存失败case(便于分析)

3.5 文档可读性

编写SKILL/设计文档时检查:

- [ ] **层次清晰**:使用Markdown标题(#/##/###)
- [ ] **示例丰富**:每个概念至少1个具体示例
- [ ] **表格对比**:用表格展示方案对比/统计数据
- [ ] **可视化辅助**:树形图/流程图/时间轴
- [ ] **链接引用**:相关文档/代码/数据文件链接
- [ ] **演化历史**:记录决策过程和版本变更
- [ ] **检查清单**:提供操作指南和检查清单

四、可读性的收益

4.1 降低学习门槛

新人onboarding时间对比:

项目类型	第一天理解度	独立工作时间
传统数据库项目	10%(需学习schema/工具)	2-3周
本项目	**70%(直接阅读数据)**	**3-5天**

原因:
- 数据文件即文档(打开Markdown就能理解)
- 分类树可视化(本体结构一目了然)
- SKILL文档详尽(包含方法论和示例)

4.2 提高调试效率

典型bug定位时间对比:

场景	传统项目	本项目
数据异常(如年份错误)	30分钟(查询→导出→分析)	**2分钟**(grep+查看)
流水线中间错误	1小时(加日志→重跑→分析)	**5分钟**(查看stage输出)
规则冲突	2小时(阅读代码→理解逻辑)	**10分钟**(阅读SKILL文档)

加速因素:
- 中间输出可见(无需重跑)
- 错误消息详细(直接定位)
- 文档完善(快速理解设计)

4.3 促进协作

Code review效率对比:

维度	传统项目	本项目
理解改动意图	需询问作者	**Git diff + 注释自解释**
验证改动正确性	需运行代码	**查看Markdown预览**
提出修改建议	模糊描述("第三段有问题")	**紫色编号精确定位(#005)**
Review时间	30分钟/PR	**10分钟/PR**

4.4 知识沉淀

传统项目的知识流失:

```
开发者离职 → 设计决策丢失 → 后人不敢改动(怕破坏隐含约束)
```

本项目的知识留存:

```
设计决策文档化 → 代码注释详细 → 演化历史可追溯
→ 新人可独立维护 → 项目可持续演化
```

案例:
- v1.0→v2.5的标注体系演化:完整记录在`00-META_标注体系设计.md`
- 为什么是0.6的相似度阈值:注释中记录了A/B测试结果
- 单锚点塌缩问题:记录在`第一轮事件年代反思总结.md`,后人可避免

五、反模式(避免)

❌ 反模式1:压缩格式优先

表现:为节省存储/带宽,使用压缩格式

```
{"e":[{"i":"001_001","t":3,"p":[12,34]}]}
```

后果:
- 人类无法直接理解
- 调试困难(需解压/反序列化)
- Git diff不可读

正确做法:
- 开发阶段用美化格式
- 生产环境再压缩(如有必要)
- 提供转换工具(压缩↔美化)

❌ 反模式2:只有机读格式

表现:只提供JSON/数据库,无人读文档

后果:
- 新人依赖工具(Protégé/数据库客户端)
- 快速浏览困难(需写查询)
- 协作困难(非技术人员无法参与)

正确做法:
- 双格式输出(JSON + Markdown)
- Markdown作为"数据的文档"
- 提供可视化界面(HTML)

❌ 反模式3:黑箱流水线

表现:只保存最终结果,中间过程不可见

后果:
- 调试困难(不知道哪个阶段出错)
- 无法优化(不知道瓶颈在哪)
- 不可复现(参数调整后无法对比)

正确做法:
- 每阶段保存输出(`tmp/stageN_*.json`)
- 生成人读预览(`tmp/stageN_preview.md`)
- 支持断点续跑(`--resume-from stageN`)

❌ 反模式4:隐性知识

表现:设计决策只在开发者脑中

后果:
- 离职/遗忘后知识丢失
- 后人不敢修改(怕破坏隐含约束)
- 重复犯错(历史教训未记录)

正确做法:
- 关键决策写注释(代码中)
- 重大决策写文档(`doc/spec/`)
- 演化历史记录(`## 演化记录`章节)

六、总结

核心要点

1. **人类优先**:数据设计首先考虑人类阅读
2. **双轨制**:JSON(机读) + Markdown(人读)并存
3. **透明流水线**:中间结果全部可见
4. **Git友好**:文本格式、行式结构、美化输出
5. **知识显性化**:设计决策、演化历史全部文档化

可读性的三个层次

L1: 语法可读(Syntactic Readability)
- 格式美化(indent、换行)
- 符号轻量(Token vs XML)
- 命名清晰(描述性变量名)

L2: 语义可读(Semantic Readability)
- 注释详尽(为什么这样设计)
- 文档完善(SKILL/Spec/README)
- 示例丰富(每个概念有例子)

L3: 认知可读(Cognitive Readability)
- 层次清晰(概览→详情渐进)
- 可视化辅助(树形图/表格/时间轴)
- 领域语言(专家熟悉的术语)

本项目追求L3级可读性:不仅"能看懂",而且"看得舒服"、"理解得深"。

与其他元技能的关系

- → **数据体感**:可读性是数据体感的前提(看不清怎么观察?)
- → **柳叶刀方法**:分模块设计需要分模块可见
- → **质量控制**:lint工具的错误消息要可读
- → **冷启动**:新人快速上手依赖可读性
- → **反思**:反思结果要以可读格式呈现

可读性是所有元技能的基础设施:没有可读性,其他方法论都会打折扣。

元技能08: 标注体系设计 — 语义标记符号的工程化设计原则

> **元技能定位**：指导如何为古籍/文档设计语义标注体系（Token标记、类型分类、消歧规则），适用于任何需要对原文进行非破坏性语义标注的项目。

零、什么是标注体系

标注体系是一套用于在原文中标记语义信息的符号系统，包括：
- **标记符号**：用什么符号包裹实体（如 `〖@人名〗`）
- **类型分类**：定义哪些实体类型需要标注（如人名/地名/官职）
- **消歧规则**：如何区分同名异指（如"武王"可能是周武王或其他武王）
- **边界规则**：标注范围如何确定（如"丞相张苍"标注为 `〖;丞相〗〖@张苍〗`）

标注体系 vs 数据库Schema

维度	数据库Schema	标注体系
目标	数据存储与查询	原文语义增强
载体	结构化表格	自然语言文本
可读性	人类不可读（需查询工具）	人类仍可直接阅读
渲染	无需渲染	可渲染为HTML/PDF/富文本
版本控制	DDL脚本	Git diff友好的纯文本
迭代成本	需数据迁移	仅需符号替换

核心洞察：标注体系是"内容与样式分离"在语义层面的应用——标记只记录语义类型，视觉呈现交给渲染程序。

一、标注体系的设计空间

1.1 标记符号的四种范式

范式	示例	优势	劣势	适用场景
XML/SGML	`<person>刘邦</person>`	标准化、工具链成熟、可嵌套	标签淹没原文、Git diff噪音大	学术数字人文、TEI标注
HTML/CSS	`<span class="person">刘邦</span>`	渲染直接、浏览器原生支持	标签冗长、与渲染耦合	Web发布为主的项目
Markdown扩展	`[刘邦]{.person}`	轻量、兼容Markdown	与Markdown原生语法冲突	Markdown文档增强
自定义Token	`〖@刘邦〗`	极简、原文可读、Git友好	需自研渲染器、无现成工具	大规模标注、频繁迭代

选择原则：
- 学术发表 → XML/TEI（标准兼容性）
- 快速原型 → Markdown扩展（零学习成本）
- 大规模生产 → 自定义Token（效率与可读性）

1.2 本项目的选择：〖TYPE content〗范式

演化路径：

```
v1.0: @人名@ =地名= $官职$
  ↓ (Markdown math冲突)
v2.0: 〖@人名〗 〖=地名〗 〖;官职〗
  ↓ (类型扩展)
v2.5: 18类统一为 〖TYPE content〗
  ↓ (消歧扩展)
v2.7: 〖TYPE 显示名|规范名〗
```

设计决策:
1. **白色六角括号〖〗** — CJK符号，视觉突出但不刺眼，极少出现在古文中（避免冲突）
2. **单字符类型标记** — `@=;#%&'^~•!+$?{:[_` 18种，记忆成本低
3. **内联消歧语法** — `|` 分隔显示名与规范名，标注即消歧
4. **统一格式** — 所有类型用同一套〖〗包裹，便于正则匹配

对比实例:

```
# XML标注（字符数膨胀3-5倍）
<person>黄帝</person>者，<person>少典</person>之子，
姓<clan>公孙</clan>，名曰<person>轩辕</person>。

# Token标注（字符数仅增30%）
〖@黄帝〗者，〖@少典〗之子，姓〖&公孙〗，名曰〖@轩辕〗。
```

二、类型分类的设计原则

2.1 类型分类的三个维度

维度1：封闭性（Closed vs Open）

封闭类型	开放类型
可枚举（如朝代：夏商周秦汉）	不可枚举（如人名：数万人）
可用词表驱动（规则标注）	需语义判断（LLM标注）

封闭类型	开放类型
准确率高（100%）	准确率中等（95%）

示例:
- 封闭：朝代（30个）、年号（~500个）、官职（~800个）
- 开放：人名（~8000个）、地名（~3000个）、事件（~3000个）

维度2：粒度（Coarse vs Fine）

```
粗粒度：人物（人名+官职+身份混为一类）
  ↓ 细化
中粒度：人名 vs 官职 vs 身份
  ↓ 细化
细粒度：人名(真名/字/号/谥号) / 官职(中央/地方/军职) / 身份(宗法/社会/政治)
```

选择原则:
- 初期从粗粒度开始（减少标注负担）
- 发现粒度不足时细化（如v2.1分离邦国与朝代、v2.3分离身份与官职）
- 细化成本：需重新标注已标注文本（迁移脚本）

维度3：正交性（Orthogonal vs Overlapping）

❌ **非正交类型（有歧义）**:
- 人名 vs 神话人物（黄帝既是人名也是神话，如何分类？）
- 地名 vs 邦国名（"齐"既是地名也是国名，如何分类？）
- 官职 vs 封号（"武安侯"既是官职也是封号，如何分类？）

✅ **正交类型（无歧义）**:
- 人名 vs 氏族名（人名=具体个人，氏族=血缘集团，"刘邦"vs"刘氏"）
- 地名 vs 邦国名（地名=地理位置，邦国=政治实体，"咸阳"vs"秦"）
- 官职 vs 身份（官职=正式册封，身份=社会角色，"丞相"vs"天子"）

判断标准：同一词汇在任何上下文中都应明确属于唯一类型。

2.2 本项目的18类体系

演化历史:

```
v1.0 (11类): 人名/地名/官职/时间/朝代/数量/制度/族群/器物/天文/神话
  ↓ (2026-03-13 扩展文化类型)
v1.1 (15类): +生物/典籍/礼仪/刑法/思想
  ↓ (2026-03-14 分离邦国与身份)
v2.1: 朝代→邦国（统一王朝+诸侯国+外邦）
v2.3: 新增身份（从官职分离）
  ↓ (2026-03-14 分离数量)
v2.5 (18类): +数量（从时间分离）
```

最终分类:

符号	类型	封闭性	粒度	典型示例
@	人名	开放	细	刘邦、项羽、司马迁
=	地名	开放	细	长安、咸阳、泰山
;	官职	半封闭	中	丞相、太尉、郡守
#	身份	半封闭	中	天子、太后、诸侯
%	时间	半开放	细	元年、三月、冬十月
&	氏族	半开放	中	刘氏、嬴氏、姬姓
'	邦国	半封闭	粗	汉、秦、齐、楚
^	制度	半开放	粗	郡县制、封禅、籍田
~	族群	封闭	粗	匈奴、三苗、东胡
•	器物	开放	细	宝鼎、印绶、符节
!	天文	半封闭	中	岁星、彗星、闰月
+	生物	开放	细	龙、凤、百谷

符号	类型	封闭性	粒度	典型示例
`$`	数量	开放	细	三万人、千里、五尺
`?`	神话	半开放	粗	黄帝、女娲、盘古
`{`	典籍	封闭	粗	春秋、尚书、诗经
`:`	礼仪	半封闭	中	宗庙、祭祀、朝觐
`[`	刑法	半封闭	中	族诛、腰斩、黥刑
`_`	思想	半开放	粗	仁义、道德、本纪

类型选择的决策树：

```
词汇X需要标注吗?
  ↓ 是
X是具体个人吗?
  ↓ 是 → @人名
  ↓ 否
X是地理位置吗?
  ↓ 是 → =地名
  ↓ 否
X是正式官职吗?
  ↓ 是 → ;官职
  ↓ 否
X是社会身份吗?
  ↓ 是 → #身份
  ↓ 否
...（依次判断）
```

三、消歧规则的设计

3.1 消歧的三种层次

层次	方法	成本	示例
L1 内联消歧	标注时直接指定规范名	低（标注时处理）	`〖@台|吕台〗`
L2 映射表消歧	章节级别的映射JSON	中（需维护JSON）	`{"046": {"武王": "齐威王"}}`
L3 全局推理消歧	跨章节分析人物活跃期	高（需LLM推理）	自动判断"武王"在不同章节指向谁

选择原则：
- 优先L1（标注即消歧，无需后处理）
- L1不够时用L2（章节内同名频繁出现）
- L2仍不够时才用L3（跨章节复杂引用）

3.2 内联消歧语法（v2.7）

语法：`〖TYPE 显示名|规范名〗`

适用场景：

1. **单字省称**
   `原文：台弒君而自立 标注：〖@台|吕台〗弒君而自立 渲染：<span title="吕台">台</span>弒君而自立`
2. **同名消歧**
   `原文：桓公好衣紫 标注：〖@桓公|齐桓公〗好衣紫 渲染：<span title="齐桓公">桓公</span>好衣紫`
3. **简称补全**
   `原文：函谷之险 标注：〖=函谷|函谷关〗之险`

4. **年号消歧**

   `原文：元年，改元 标注：〖%元|元狩元年〗年，改元`

优先级：内联消歧 > 映射表 > 全局推理

渲染输出：

```
<a href="entities/person.html#吕台">
  <span class="person" data-canonical="吕台" title="规范名：吕台">台
</span>
</a>
```

3.3 映射表消歧（disambiguation_map.json）

数据结构：

```
{
  "046_田敬仲完世家": {
    "武王": "齐威王",
    "太后": "君王后",
    "宣王": "齐宣王"
  },
  "047_孔子世家": {
    "武王": "周武王",
    "成王": "周成王"
  }
}
```

应用时机：渲染时替换

```
def render_entity(text, chapter, canonical_name):
    # 先查内联消歧（|后的规范名）
    if '|' in text:
        display, canonical = text.split('|')
        return canonical

    # 再查映射表
```

```
    if chapter in disambiguation_map:
        return disambiguation_map[chapter].get(canonical_name,
canonical_name)

    # 最后用原名
    return canonical_name
```

3.4 消歧的边界情况

情况1：故意不消歧

有些同名在原文中就是模糊指代，强行消歧反而破坏原意：

```
原文：昔者武王伐纣
判断：此处"武王"泛指行武功之王，可能是周武王，也可能是其他武王
处理：不消歧，保持〖@武王〗
```

情况2：多重指称

同一实体在不同段落用不同称谓：

```
原文：
[1.1] 高祖，沛丰邑中阳里人也
[1.2] 刘季少时...
[1.3] 项羽号令诸侯，立沛公为汉王

标注：
〖@高祖|刘邦〗，〖=沛〗〖=丰邑〗〖=中阳里〗人也
〖@刘季|刘邦〗少时...
立〖@沛公|刘邦〗为〖@汉王|刘邦〗
```

情况3：嵌套消歧

```
原文：秦王政
判断："秦王"是邦国+身份的组合，"政"是人名
标注：〖'秦〗〖#王〗〖@政|秦始皇〗
```

四、边界规则的设计

4.1 边界确定的三个原则

原则1：最小完整单位

标注范围应是语义上的最小完整单位，不多不少。

```
# ✅ 正确
〖@刘邦〗              # "刘邦"是完整人名
〖;丞相〗〖@萧何〗  # "丞相"是官职，"萧何"是人名，分开标注

# ❌ 错误
〖@刘〗邦              # 拆分了完整人名
〖;丞相萧何〗        # 混淆了官职与人名
```

原则2：不包含非实体成分

标注不应包含标点、助词、动词等非实体成分。

```
# ✅ 正确
〖@刘邦〗，〖@项羽〗
伐〖'赵〗

# ❌ 错误
〖@刘邦，〗〖@项羽〗     # 标注包含标点
〖@伐赵〗                  # 标注包含动词
```

原则3：嵌套时拍平

当出现类型嵌套时，保留外层，去掉内层。

```
# 原意：〖;〖@安国君〗〗（安国君是人名，但此处作为官职引用）
# 拍平：〖;安国君〗

# 原意：〖@帝〖@颛顼〗〗（颛顼是帝号）
# 拍平：〖@帝颛顼〗
```

4.2 边界的典型错误模式

错误1：边界损坏（包含标点）

错误	正确	检测正则
〖@刘邦，〗	〖@刘邦〗，	〖@[^〗]*[，。、；：]
〖@弟〗项庄	〖@弟〗〖@项庄〗	〖@[^〗]*〗[^，。\s]

错误2：边界膨胀（包含非实体词）

错误	正确	说明
〖@，次曰项羽〗	，次曰〖@项羽〗	"次曰"是连接词
〖@，是为高祖〗	，是为〖@高祖〗	"是为"是系词
〖@弒其君〗	弒其〖#君〗	"弒"是动词

错误3：边界缺失（应拆分未拆分）

错误	正确	说明
〖@丞相萧何〗	〖;丞相〗〖@萧何〗	官职+人名应分开
〖@秦王政〗	〖'秦〗〖#王〗〖@政〗	邦国+身份+人名

自动检测：

```
# 检测边界损坏
grep -n '〖@[^〗]*[，。、；：]' chapter_md/*.tagged.md

# 检测边界膨胀
grep -n '〖@，' chapter_md/*.tagged.md

# 检测可疑长标注（可能包含非实体词）
grep -oP '〖@[^〗]{10,}〗' chapter_md/*.tagged.md
```

五、标注体系的演化策略

5.1 迁移脚本模板

当类型定义变化时，需要批量迁移已标注文本：

```
#!/usr/bin/env python3
"""
类型迁移脚本模板

用途：批量替换标注符号（如 v1.0 → v2.0格式迁移）
"""

import re
from pathlib import Path

# 迁移规则表
MIGRATIONS = [
    # (旧格式正则，新格式替换，说明)
    (r'@([^@]+)@', r'〖@\1〗', '人名：@X@ → 〖@X〗'),
    (r'=([^=]+)=', r'〖=\1〗', '地名：=X= → 〖=X〗'),
    (r'\$([^\$]+)\$', r'〖;\1〗', '官职：$X$ → 〖;X〗'),
]

def migrate_file(file_path):
```

```
    """迁移单个文件"""
    text = file_path.read_text('utf-8')
    original = text

    for old_pattern, new_format, description in MIGRATIONS:
        text = re.sub(old_pattern, new_format, text)

    if text != original:
        file_path.write_text(text, 'utf-8')
        print(f"✅ {file_path.name}: {description}")
        return True
    return False

def main():
    chapter_dir = Path('chapter_md')
    migrated_count = 0

    for file in chapter_dir.glob('*.tagged.md'):
        if migrate_file(file):
            migrated_count += 1

    print(f"\n迁移完成: {migrated_count}/130章")

if __name__ == '__main__':
    main()
```

5.2 版本演化的最佳实践

实践1：向后兼容

新版本应能渲染旧版本标注（兼容层）：

```
def parse_entity(text):
    """兼容多版本标注"""
    # v2.0+
    if match := re.match(r'〖([@=;#%&\'^~•!+\$?{:[_])([^〗|]+)(?:
\|([^〗]+))?〗', text):
```

```
        type_marker, content, canonical = match.groups()
        return type_marker, canonical or content

    # v1.0兼容
    if match := re.match(r'@([^@]+)@', text):
        return '@', match.group(1)

    # 旧生物格式兼容
    if match := re.match(r'〖([^〗]+)〗', text):
        return '+', match.group(1)

    return None, text
```

实践2：分阶段迁移

不要一次性修改所有规则，分批迭代：

```
第一阶段：格式迁移（@ → 〖@〗）
  ↓ 验证lint通过
第二阶段：类型细化（朝代 → 邦国）
  ↓ 验证lint通过
第三阶段：消歧扩展（|规范名）
  ↓ 验证lint通过
```

实践3：Git记录每次迁移

```
# 迁移前打tag
git tag v1.0-final

# 执行迁移
python scripts/migrate_to_v2.py

# 验证
python scripts/lint_markdown.py --all

# 提交
git add -A
```

```
git commit -m "标注格式迁移: v1.0 → v2.0

- @X@ → 〖@X〗
- =X= → 〖=X〗
- 迁移脚本: scripts/migrate_to_v2.py
- 影响章节: 130章全部迁移"

git tag v2.0
```

5.3 类型新增的决策流程

```
发现新需求（如"刑法"词汇混在"制度"中）
  ↓
Step 1: 侦察
  - 统计现有〖^制度〗中有多少是刑法？（grep统计）
  - 估算迁移成本（多少处需要改？）
  ↓
Step 2: 设计
  - 定义新类型边界（刑法 vs 制度的区别？）
  - 选择类型标记符号（未使用的符号: `[`）
  - 设计判断规则（什么算刑法？）
  ↓
Step 3: 试点
  - 手动迁移10章，验证规则清晰性
  - 检查是否有歧义case
  ↓
Step 4: 批量迁移
  - 编写迁移脚本
  - 执行批量迁移
  - lint验证
  ↓
Step 5: 文档更新
  - 更新SKILL_03a（18类→新类型）
  - 更新消歧规则
  - 记录决策理由
```

六、标注体系的质量保障

6.1 一致性检查

检查项1：同一实体的多种标注形式

```
# 检查"刘邦"的所有标注形式
grep -oh '〖@[^〗]*刘邦[^〗]*〗' chapter_md/*.tagged.md | sort |
uniq -c

# 预期输出（一致）：
#   1234 〖@刘邦〗
#     56 〖@高祖|刘邦〗
#     23 〖@沛公|刘邦〗

# 问题输出（不一致）：
#    800 〖@刘邦〗
#    400 〖@刘邦|汉高祖〗    # 规范名不一致
#     34 〖#刘邦〗            # 类型错误
```

检查项2：类型标记的误用

```
def check_type_consistency():
    """检查类型标记是否符合定义"""
    issues = []

    # 人名不应包含"氏/宗/族"后缀
    for match in re.finditer(r'〖@([^〗]+)[氏宗族]〗', text):
        issues.append(f"人名包含氏族后缀: {match.group(0)} → 应为
〖&{match.group(1)}氏〗")

    # 单字邦国名应为〖'〗而非〖@〗
    for match in re.finditer(r'(伐|攻|与|及)〖@([赵韩魏秦齐楚燕宋卫陈郑
鲁吴越])〗', text):
        verb, country = match.groups()
```

```
        issues.append(f"邦国名误标为人名: {verb}〖@{country}〗 → {verb}〖'{country}〗")

    return issues
```

6.2 覆盖率检查

```
def check_coverage():
    """检查应标注实体的覆盖率"""
    # 从人名词表加载所有已知人名
    known_persons = load_person_wordlist()

    # 检查未标注的人名（出现在原文但未标注）
    missed = []
    for person in known_persons:
        # 查找原文中出现但未标注的
        untagged_pattern = f'(?<!〖@){person}(?!〗)'
        if re.search(untagged_pattern, text):
            missed.append(person)

    coverage = 1 - len(missed) / len(known_persons)
    print(f"人名标注覆盖率: {coverage:.1%}")

    if missed:
        print(f"遗漏标注: {', '.join(missed[:10])}...")
```

6.3 标注质量指标

指标	定义	目标	检测方法
格式正确率	符合〖TYPE〗语法的标注比例	100%	lint_markdown.py
类型准确率	类型标记正确的实体比例	>95%	人工抽样+反思修正

指标	定义	目标	检测方法
消歧完整率	需要消歧的实体已消歧比例	>90%	检查disambiguation_map覆盖
边界准确率	边界无损坏的标注比例	100%	边界损坏检测脚本
一致性	同一实体标注形式一致的比例	>98%	一致性检查脚本

七、标注体系的扩展性

7.1 适配到其他古籍

本标注体系可直接应用于二十四史及其他古籍，需适配：

古籍类型	需要调整的部分	示例
史书	类型定义（可能需要新增官职/制度类型）	《汉书》需新增西汉特有官职
子书	类型定义（思想/学派类型需细化）	《庄子》需细化道家术语
集部	类型定义（文体/修辞类型）	《文选》需标注赋/骈文/诗歌类型
佛经	类型定义（需新增"佛教术语"类型）	佛经需标注菩萨名/经名/仪轨

通用部分（无需调整）：
- 〖TYPE content〗格式本身
- 内联消歧语法 `|规范名`

- 边界规则（最小完整单位、不包含标点）
- lint工具和渲染管线

7.2 适配到现代文本

标注体系也可用于现代文本（论文/新闻/技术文档）：

```
# 学术论文标注
〖@Smith〗等人提出的〖_深度学习〗模型在〖=ImageNet〗数据集上
达到了〖$95.2%〗的准确率。

# 新闻标注
〖@拜登〗总统在〖=白宫〗宣布，〖'美国〗将向〖'乌克兰〗
提供〖$100亿美元〗的军事援助。

# 技术文档标注
使用〖{Python〗库〖{NumPy〗处理〖$1000万〗条数据，
在〖^GPU加速〗下耗时〖%3.2秒〗。
```

类型映射:

古籍类型	现代等价
人名 `@`	人名/作者名
地名 `=`	地名/机构名
官职 `;`	职位/角色
邦国 `'`	国家/组织
典籍 `{`	文献引用/代码库
数量 `$`	数值/指标

八、总结

标注体系设计的核心是在**可读性**与**语义精确性**之间寻找平衡点。关键要素：

1. **符号选择** — 轻量、不冲突、Git友好
2. **类型分类** — 正交、粒度适中、可演化
3. **消歧规则** — 内联优先、分层补全、边界明确
4. **演化策略** — 向后兼容、分阶段迁移、Git记录
5. **质量保障** — 格式/类型/边界/一致性多维度检查

核心洞察：
- 标注体系不是一次性设计，而是**随认知深化而迭代演化**
- v1.0到v2.5的演化证明：初期从简单开始，发现不足时及时细化
- 迁移成本可控的前提是**早期就建立lint工具和迁移脚本框架**

附录：参考资料

- `skills/SKILL_02_结构分析.md` — 标注符号设计原则与对比分析
- `skills/SKILL_03a_实体标注.md` — 18类实体完整定义与演化历史
- `scripts/migrate_to_lenticular.py` — v1.0→v2.0迁移脚本实例
- `scripts/migrate_dynasty_to_feudal.py` — v2.1邦国类型迁移脚本
- `scripts/migrate_official_to_identity.py` — v2.3身份类型分离脚本
- `kg/entities/data/disambiguation_map.json` — 消歧映射表数据结构

元技能09: Agent提示词工程 — 方法论驱动的AI任务设计

> **元技能定位**：指导如何设计高质量的Agent提示词，使Claude Agent能够自主完成复杂任务。与具体数据类型无关，适用于所有需要Agent执行的工序（实体标注、事件提取、关系发现、质量审查等）。

零、什么是Agent提示词工程

Agent提示词工程是将人类专家的工作方法论（SKILL文档）转化为Claude Agent可执行的提示词，使Agent能够**自主理解任务→执行操作→输出结果→自我验证**的完整闭环。

传统Prompt vs Agent提示词

维度	传统Prompt	Agent提示词
复杂度	单轮对话，一次性输出	多步骤任务，需要工具调用
自主性	被动响应，等待指令	主动规划，自主执行
方法论	隐式包含在prompt中	显式嵌入SKILL文档
可迭代性	一次性，无状态	可反思、可修正、可积累经验
输出格式	自然语言	结构化数据（JSON/Markdown）
质量保障	依赖人工审核	嵌入质量基线和自检步骤

核心洞察：Agent提示词 = 任务定义 + 方法论文档 + 质量标准 + 工具链接 + 输出规范

一、Agent提示词的五层结构

本项目的Agent提示词遵循五层金字塔结构：

```
┌─────────────────────────────────────────────────────┐
│  L5  输出规范（Output Specification）               │
│       JSON schema / Markdown格式 / 文件路径         │
├─────────────────────────────────────────────────────┤
│  L4  验证步骤（Validation Steps）                   │
│       自检清单 / lint命令 / 质量基线                │
├─────────────────────────────────────────────────────┤
│  L3  执行方法（Execution Method）                   │
│       SKILL文档 / 错误模式表 / 工具使用指南         │
├─────────────────────────────────────────────────────┤
│  L2  任务约束（Task Constraints）                   │
│       边界条件 / 禁止操作 / 异常处理                │
├─────────────────────────────────────────────────────┤
│  L1  任务目标（Task Objective）                     │
│       要做什么 / 为什么做 / 成功标准                │
└─────────────────────────────────────────────────────┘
```

1.1 L1: 任务目标（Task Objective）

要素：
- 明确角色定位（你是XXX专家）
- 清晰任务描述（你要完成什么）
- 明确成功标准（怎样算完成）

示例（事件识别）：

```
你是《史记》事件提取专家。你的任务是从已标注的章节原文中提取结构化历史事件。

## 任务目标

从章节 `NNN_章节名.tagged.md` 中提取所有重要历史事件，每个事件包含：
- 事件名称（4-8字概括）
```

```
- 时间标注（原文纪年+推断公元年）
- 人物/地点（从实体标注中提取）
- 事件描述（2-4句话）
- 事件类型（11种之一）

## 成功标准

- 覆盖率：章节中所有重要历史转折点都应被提取
- 粒度：每章15-40个事件（平均23.8个）
- 准确性：时间/人物/地点与原文一致
```

1.2 L2: 任务约束（Task Constraints）

要素:
- 禁止操作（不要做什么）
- 边界条件（什么情况下跳过）
- 异常处理（遇到问题怎么办）

示例（实体标注反思）:

```
## 约束条件

1. **不修改原文字符** — 只修改标注符号（〔〕），原文一个字都不能动
2. **不批量处理** — 每个Agent只处理一章，不要试图一次处理多章
3. **不主观判断** — 只修正明确错误，不凭感觉改标注
4. **不跳过段落** — 必须逐段阅读全文，不要只看开头或只看有错的地方

## 边界情况

- 遇到看不懂的古文 → 保持原标注，不要瞎改
- 遇到同一词汇可能有多个分类 → 标记为context_list，不要强行选择
- 遇到lint报错 → 优先修复格式错误，再处理语义问题

## 质量基线
```

```
- 正常章节（300-1000行）预期30-80处修正
- 若修正数 < 20处 → 再次逐段复查（可能遗漏了系统性错误）
- 若修正数 > 150处 → 报告异常（可能理解错了任务）
```

1.3 L3: 执行方法（Execution Method）

要素:
- 嵌入完整SKILL文档（方法论）
- 提供错误模式表（已知问题）
- 列出工具使用指南（如何调用工具）

示例（按章反思）:

```
## 执行方法

### Step 0: 预备（必须先执行）

1. 读取 `skills/SKILL_03c_按章反思.md` §三（已积累规律），了解所有已知错误模式
2. 运行格式检查脚本，验证markdown格式是否正确
3. 运行边界损坏检测，查找标注中包含标点符号的错误模式

### Step 1: 格式与边界扫描

参考SKILL文档中的17类错误模式表:

| 错误模式 | 修复 | 理由 |
|---------|------|------|
| `〖@赵〗` 单字邦国名 | `〖'赵〗` | "伐赵"/"与韩攻赵"中的赵是邦国 |
| `〖@刘氏〗` 氏族后缀 | `〖&刘氏〗` | "氏/宗/族"后缀是氏族，非人名 |
| `〖@弒〗` 动词误标 | 去标注 | 弒/杀/攻/起是动词，不是人名 |
| ... | ... | ... |

### Step 2: 逐段阅读审查

对每个段落:
1. 检查人名标注（最高频错误）
```

```
2. 检查官职与身份分类
3. 检查地名与邦国区分
4. 检查边界完整性（标注不应包含标点）

### Step 3: 输出修正建议

每处修正包含：
- 原标注（old_value）
- 新标注（new_value）
- 修正理由（reasoning）
- 所在段落（paragraph_id）
```

1.4 L4: 验证步骤（Validation Steps）

要素:
- 自检清单（Agent自己验证）
- Lint命令（格式验证）
- 质量指标（数量/覆盖率/准确率）

示例（事件年代审查）:

```
## 验证步骤

完成修正后，Agent必须自检：

### 自检清单

- [ ] 所有修正都附有理由（reasoning字段非空）
- [ ] 没有修改原文字符（只改了年份标注）
- [ ] 修正数在合理范围（5-50处，超出范围说明可能有问题）
- [ ] 时序不倒退（修正后的年份应递增）
- [ ] 格式规范（确定纪年用(公元前X年)，不确定用[约公元前X年]）

### Lint验证

修正应用后，运行年代验证脚本检查时序一致性
```

```
预期输出：✅ PASS，无时序倒退

### 质量指标

- 修正量：5-50处（第一轮）/ 2-20处（第二轮）
- 新模式发现：0-3条（第一轮）/ 0条（第二轮+）
- 置信度：high(确定错误) > 80% / medium(可能错误) < 20%
```

1.5 L5: 输出规范（Output Specification）

要素:
- 精确的JSON schema定义
- 字段说明（必填/可选、值域、示例）
- 文件路径约定

示例（反思修正输出）:

```
## 输出格式

输出JSON文件：`reflect/reflect_NNN.json`

### Schema定义

\`\`\`json
{
  "chapter": "NNN",
  "chapter_name": "章节名",
  "round": 1,
  "corrections": [
    {
      "id": "事件/实体ID（必填）",
      "type": "错误类型（必填）",
      "field": "字段名（必填）",
      "old_value": "修正前的值（必填）",
      "new_value": "修正后的值（必填）",
      "reasoning": "修正理由（必填，至少20字）",
      "confidence": "high|medium|low（必填）",
```

```
      "evidence": "支持证据（可选，如年表条目、原文段落）"
    }
  ],
  "new_patterns": [
    {
      "pattern": "错误模式描述（必填）",
      "frequency": "估计出现频率（如'全书约500次'）",
      "solution": "修正方法（必填）"
    }
  ],
  "stats": {
    "total_items": 123,
    "corrections_count": 45,
    "correction_rate": 0.366,
    "confidence_distribution": {
      "high": 36,
      "medium": 8,
      "low": 1
    }
  }
}
\`\`\`

### 字段说明

- `id`: 事件ID格式为`NNN-XXX`（章节号-序号），实体ID为规范名
- `type`: 错误类型枚举值见附录A
- `confidence`:
  - high: 100%确定错误（如年表有明确记载）
  - medium: 80%可能错误（如逻辑推断）
  - low: <50%可能性（存疑，需人工复核）
- `reasoning`: 必须说明修正依据，如"年表014-015条记载齐威王元年=前356年"

### 示例

\`\`\`json
{
```

```
  "chapter": "046",
  "corrections": [
    {
      "id": "046-010",
      "type": "year_correction",
      "field": "ce_year",
      "old_value": "（公元前370年）",
      "new_value": "（公元前348年）",
      "reasoning": "原文%九年%，按年表014-015，齐威王元年=前356年，九年=
前348年。《史记》自身纪年与竹书纪年存在22年系统性偏差",
      "confidence": "high",
      "evidence": "年表014-015：前353年'齐威王四年'、前350年'齐威王七
年'、前334年'齐威王二十三年'，三条独立印证元年=前356年"
    }
  ]
}
\`\`\`
```

二、SKILL文档的设计原则

SKILL文档是Agent的"工作手册"，必须遵循以下原则：

2.1 表格化优于自然语言

❌**不好的做法（自然语言描述）**：

```
常见错误包括单字邦国名被误标为人名，比如赵、韩、魏、秦等，
这些在"伐赵"、"与韩攻魏"等语境中应该是邦国名而非人名。
另外氏族名也容易误标，比如"刘氏"、"嬴氏"应该用氏族标记。
```

✅**好的做法（表格化）**：

```
| 错误模式 | 正确标注 | 判断规则 | 示例 |
|---------|---------|---------|------|
| `〖@赵〗` 单字 | `〖'赵'〗` | "伐X"/"攻X"/"与X"中的X是邦国 | "伐
```

〖'赵〗" |
| `〖@刘氏〗` 氏后缀 | `〖&刘氏〗` | "X氏/X宗/X族"是氏族集团 | "〖&刘氏〗宗亲" |

原因:
- Agent可以快速扫描表格（视觉定位）
- 每行独立，便于查找和匹配
- 结构化数据易于程序化处理

2.2 实例优于抽象规则

❌**不好的做法（抽象规则）**:

人名与官职的区分：具体指称用人名，泛指多人用官职。

✅**好的做法（对比实例）**:

原文	判断	标注	理由
丞相张苍	特指张苍这个人	`〖;丞相〗〖@张苍〗`	人名
以X为丞相	职位名（任命语境）	`〖;丞相〗`	官职
丞相之职	职位本身（抽象概念）	`〖;丞相〗`	官职
凡丞相者	泛指历代丞相	`〖;丞相〗`	官职

原因:
- 实例可以直接模式匹配
- 对比实例帮助Agent理解边界
- 避免歧义解读

2.3 分层组织优于平铺

❌**不好的做法（平铺列表）**:

错误模式:
1. 单字邦国名误标
2. 氏族后缀误标

```
3. 动词误标
4. 标点边界损坏
5. 帝号谥号误标
6. ...（100条）
```

✅ **好的做法（分层分类）**：

```
## 一、格式错误（L1-L2，规则检测）

### 1.1 边界损坏
| 错误 | 修复 |
|------|------|
| `〖@X，〗` | `〖@X〗，` |

### 1.2 空标注
| 错误 | 修复 |
|------|------|
| `〖=〗` | 删除 |

## 二、类型误判（L3，语义判断）

### 2.1 人名 vs 邦国名
| 错误 | 正确 | 判断规则 |
|------|------|---------|
| `〖@赵〗` | `〖'赵〗` | "伐X"语境 |

### 2.2 人名 vs 氏族名
| 错误 | 正确 | 判断规则 |
|------|------|---------|
| `〖@刘氏〗` | `〖&刘氏〗` | "X氏"后缀 |
```

原因:
- 分层帮助Agent快速定位相关规则
- 同类错误集中处理（批量检查）
- 优先级明确（格式错误先修，语义错误后修）

2.4 版本控制与增量更新

SKILL文档应记录演化历史:

```
## 已知错误模式表（v3.2，2026-03-17更新）

### 第一轮反思新增（25条，2026-03-09）
| ID | 模式 | 修复 | 发现章节 |
|----|------|------|---------|
| P01 | 单字邦国名 | `〖@〗→〖'〗` | 001-130全书 |
| P02 | 氏族后缀 | `〖@X氏〗→〖&X氏〗` | 001-130全书 |

### 第二轮反思新增（1条，2026-03-17）
| ID | 模式 | 修复 | 发现章节 |
|----|------|------|---------|
| P26 | 长子/少子 | 不标注（亲属关系非身份） | 013, 042 |

## 已修复（不再需要检查）
| ID | 模式 | 修复时间 | 说明 |
|----|------|---------|------|
| P00 | 旧格式残留 | 2026-03-13 | v2.0迁移已完成 |
```

原因:
- Agent知道哪些模式是新的（重点检查）
- Agent知道哪些模式已修复（跳过）
- 人类维护者可追溯模式演化

三、Agent提示词的迭代模式

3.1 冷启动阶段（第一轮）

特点:
- SKILL文档几乎为空（只有任务定义）
- 质量基线宽松（发现模式优先）
- 重点是**积累错误模式**而非追求100%修正

提示词设计:

```
这是第一轮反思，SKILL文档还不完善。你的任务有两个：

1. **修正明确错误** — 格式错误、明显的类型误判
2. **发现新模式** — 输出`new_patterns`数组，记录你发现的系统性错误

## 质量基线（第一轮）

- 预期修正量：20-80处/章（正常章节）
- 新模式发现：5-10条（前10章），1-3条（后续章节）
- 置信度分布：high>60%, medium<30%, low<10%

## 重点关注

- 同一类错误在多处重复 → 记录为new_pattern
- 边界情况（不确定如何分类） → 标记confidence=low
- 章节特有语境（可能需要单独处理） → 在reasoning中说明
```

3.2 模式积累阶段（第二轮）

特点:
- SKILL文档已有20-30条错误模式
- 质量基线收紧（修正量应减半）
- 重点是**系统性修正**而非发现新模式

提示词设计:

```
这是第二轮反思，SKILL文档已积累25条错误模式。

## Step 0: 学习已知模式

**必须先读取**: `skills/SKILL_XX_反思.md` §三.已知错误模式表

这些模式在第一轮中已发现，本轮重点是确保全部修正：
- P01: 单字邦国名（全书约1200次）
- P02: 氏族后缀（全书约600次）
```

```
- P03: 动词误标（全书约400次）
- ...

## 质量基线（第二轮）

- 预期修正量：10-40处/章（应比第一轮减半）
- 新模式发现：0-1条（若>2条，说明第一轮有遗漏）
- 置信度分布：high>80%, medium<15%, low<5%

## 重点关注

- 已知模式是否100%修正？
- 第一轮的low-confidence修正是否需要重新审查？
```

3.3 收敛验证阶段（第三轮+）

特点：
- SKILL文档已完善（30+条模式）
- 质量基线严格（修正量<10%）
- 重点是**交叉验证**和**残留清理**

提示词设计：

```
这是第三轮反思（收敛验证）。前两轮已修正约1,500处，本轮重点：

## 收敛判断

- 修正量应 < 第二轮的30%（预期每章0-10处）
- 新模式数 = 0（若发现新模式，说明前两轮有系统性遗漏）
- 所有修正置信度应为high（不应再有medium/low）

## 验证重点

1. **跨章一致性** — 同一人名在不同章节的标注是否一致？
2. **边界情况复查** — 第一轮标记为low-confidence的是否已解决？
3. **lint报告清零** — 格式错误应100%消除
```

```
## 终止条件

若修正量 < 5处/章 且 新模式=0 → 反思收敛，进入下一工序
```

四、常见陷阱与应对

陷阱1：提示词过长，Agent理解不全

症状:
- Agent只处理了前半部分任务
- SKILL文档中的后半部分规则没有应用

原因:
- 提示词超过10000字，Agent注意力分散
- 关键指令淹没在大量背景信息中

解决方案:
1. **关键指令前置** — 把最重要的指令放在开头
2. **分步执行** — 用Step 0/1/2/3强制Agent逐步执行
3. **重复关键约束** — 在每个Step中重复核心约束（如"不修改原文字符"）

优化示例:

```
# ❌ 差的设计（关键指令在末尾）

（5000字的背景介绍）

（3000字的SKILL文档）

（2000字的详细说明）

哦对了，记得不要修改原文字符。

# ✅ 好的设计（关键指令前置+分步重复）

## 核心约束（必须遵守）
```

```
1. **不修改原文字符** — 只改标注符号〖〗
2. **不批量处理** — 只处理一章
3. **必须逐段阅读** — 不要跳过段落

---

## Step 0: 预备

⚠️ **再次提醒：不修改原文字符**

（预备步骤）

## Step 1: 执行

⚠️ **再次提醒：只改标注符号**

（执行步骤）
```

陷阱2：质量基线过严，Agent不敢修正

症状:
- 修正量远低于预期（如应修正50处，只修正了5处）
- Agent过度谨慎，只修正100%确定的错误

原因:
- 质量基线设置为"修正数应<10处"，Agent误解为"尽量少修正"
- 没有明确"遗漏也是错误"

解决方案:

```
## 质量基线（正确表述）

- 预期修正量：30-80处/章（正常章节300-1000行）
- **若修正数 < 20处**：
  - 可能原因：
    1. 该章质量确实很高（如第二轮反思的章节）
    2. 遗漏了系统性错误（更可能）
```

```
   - 要求：**再次逐段复查**，重点检查是否遗漏P01-P25已知模式
- **若修正数 > 150处**：
   - 可能误解了任务，报告异常

## 遗漏也是错误

不要过度谨慎。已知错误模式（P01-P25）应该100%修正，不要因为"不太确定"就跳过。
```

陷阱3：JSON输出格式错误

症状:
- Agent输出的JSON无法解析（语法错误）
- 缺少必填字段

原因:
- Schema定义不够明确
- 没有提供示例

解决方案:

```
## 输出格式（JSON）

### Schema

\`\`\`json
{
  "corrections": [  // ⚠️ 必填，即使为空数组也要输出 []
    {
      "id": "string, 必填",
      "old_value": "string, 必填",
      "new_value": "string, 必填, 且必须不同于old_value",
      "reasoning": "string, 必填, 至少20字"
    }
  ]
}
\`\`\`
```

```
### 完整示例（请严格遵循此格式）

\`\`\`json
{
  "chapter": "046",
  "corrections": [
    {
      "id": "046-010",
      "old_value": "（公元前370年）",
      "new_value": "（公元前348年）",
      "reasoning": "按年表014-015，齐威王元年=前356年，九年=前348年"
    }
  ],
  "stats": {
    "corrections_count": 1
  }
}
\`\`\`

### 常见错误（避免）

- ❌ 缺少闭合括号：`{"corrections": [`
- ❌ 多余逗号：`"reasoning": "xxx",}`
- ❌ 字段名不加引号：`{id: "046"}`
- ❌ 空字段用null：`"reasoning": null` → 应该是字符串
```

陷阱4：Agent陷入无限循环

症状:
- Agent重复执行同一步骤
- Agent反复调用同一工具

原因:
- 缺少明确的终止条件
- Agent不知道"完成"的标志

解决方案:

```
## 执行流程（严格按顺序，不重复）

### Step 1: 读取文件（一次）

使用Read工具读取目标文件: `chapter_md/NNN_*.tagged.md`

✅ 读取成功 → 进入Step 2
❌ 文件不存在 → 报告错误，停止执行

### Step 2: 逐段审查（一次遍历全文）

对文件的每个段落（从第一段到最后一段）：
1. 检查人名标注
2. 检查官职标注
3. 记录修正建议

⚠️ **不要重复遍历**。一次从头到尾，记录所有修正。

### Step 3: 输出JSON（一次）

汇总所有修正 → 输出JSON → **任务完成**

## 终止条件

以下情况立即停止:
1. 已输出JSON → 任务完成
2. 文件不存在 → 报告错误
3. 修正数 > 200处 → 报告异常

不要进入"验证→修正→再验证→再修正"的循环。
```

五、提示词模板库

模板1：数据提取任务

```
你是XXX提取专家。从文件`{input_file}`中提取{数据类型}。

## 任务目标
提取所有{数据类型}，每个包含{字段列表}。

## 方法论
参考：`skills/SKILL_XX.md`

关键规则：
- {规则1}
- {规则2}

## 输出格式
JSON数组，schema见下：
\`\`\`json
[
  {
    "field1": "类型+说明",
    "field2": "类型+说明"
  }
]
\`\`\`

## 质量基线
- 覆盖率：{百分比}
- 准确率：{百分比}
- 数量范围：{min}-{max}

## 验证
```

```
- [ ] 所有{数据类型}都已提取
- [ ] 所有必填字段非空
- [ ] 输出JSON格式正确
```

模板2：质量审查任务

```
你是{数据类型}质量审查员。对文件`{input_file}`执行深度反思。

## 任务目标
发现并修正{错误类型}。

## Step 0: 预备
1. 读取SKILL文档：`skills/SKILL_XX.md`
2. 运行lint：`{lint_command}`

## Step 1: 审查
参考SKILL文档中的{N}条错误模式，逐项检查。

## Step 2: 输出修正
JSON格式，每处修正包含：old_value, new_value, reasoning

## 质量基线
- 第{N}轮预期修正量：{min}-{max}处
- 新模式发现：{min}-{max}条
- 置信度：high>{百分比}

## 验证
- [ ] 修正数在合理范围
- [ ] 所有修正附有reasoning
- [ ] 新模式已记录
```

模板3：关系发现任务

````
你是{关系类型}发现专家。从标注文本中发现{主体}与{客体}的{关系}。

## 任务目标
发现所有{关系类型}，输出SPO三元组。

## 方法论
参考：`skills/SKILL_XX.md`

关系类型：
| 类型 | 定义 | 示例 |
|------|------|------|
| {type1} | {definition1} | {example1} |
| {type2} | {definition2} | {example2} |

## 提取规则
1. 从共现到关系：{规则1}
2. 从关键词到关系：{规则2}
3. 从上下文到关系：{规则3}

## 输出格式
JSON数组，SPO三元组：
\`\`\`json
[
  {
    "subject": "主语实体ID",
    "predicate": "关系类型（枚举值）",
    "object": "宾语实体ID",
    "evidence": "原文证据（段落编号+文本片段）",
    "confidence": "high|medium|low"
  }
]
\`\`\`

## 验证
````

```
- [ ] 主语/宾语ID存在于实体索引
- [ ] 关系类型在枚举值内
- [ ] evidence可定位到原文
```

六、总结

Agent提示词工程的核心是**将人类专家的工作方法论转化为机器可执行的指令**。关键要素：

1. **五层结构** — 目标→约束→方法→验证→输出，层层递进
2. **SKILL文档驱动** — 方法论显式化、表格化、版本化
3. **质量基线明确** — 告诉Agent"多少算正常"，避免过度谨慎或过度激进
4. **迭代模式** — 冷启动→模式积累→收敛验证，逐轮收紧基线
5. **防御性设计** — 前置关键指令、分步重复约束、明确终止条件

核心洞察：好的Agent提示词不是"更长"或"更详细"，而是**更结构化**、**更可执行**、**更可验证**。

附录：参考资料

- `skills/SKILL_03c_按章反思.md` — 实体标注反思提示词实例
- `skills/SKILL_04d_事件年代审查.md` — 事件年代反思提示词实例
- `doc/events/事件年代反思创造过程详解.md` — 提示词迭代演化案例
- META_反思.md — 反思方法论（提示词的应用场景）
- META_质量控制.md — 质量控制（提示词的验证步骤）

元技能10: 质量控制 — 自动化质检体系与工具链设计

> **元技能定位**：指导所有工序设计lint/validate工具，构建自动化质量保障体系。质量控制是反思（META_反思.md）的前置环节，为Agent提供可自动执行的检测规则。

零、什么是质量控制

质量控制（Quality Control, QC）是通过自动化脚本在**数据生成/修正的每个环节**进行格式/逻辑/一致性检查，及早发现错误，降低人工审核成本。

质量控制 vs 反思

维度	质量控制（QC）	反思（Reflection）
检查对象	格式、边界、数据完整性、规则违反	语义准确性、分类正确性、内容合理性
执行方式	规则脚本自动化（lint/validate）	Agent驱动的智能审查
时机	每次数据变更后立即执行	完成初次生成后，迭代修正前
成本	极低（秒级执行，无LLM成本）	中高（分钟级执行，有LLM成本）
可解释性	100%可解释（规则明确）	部分可解释（依赖LLM推理）
		JSON修正建议 + 新错误模式

维度	质量控制（QC）	反思（Reflection）
典型输出	PASS/FAIL + 错误位置 + 修复建议	

协同关系：
1. QC先行：每次修正后先跑lint，格式错误必须100%消除才能进反思
2. 反思发现的**可规则化错误模式**沉淀为新的QC规则
3. QC验证反思成果：反思修正应用后再跑lint，确保没有引入新格式错误

一、质量控制的五个层次

本项目的质量控制体系分为五个递进层次：

```
L5  跨工序一致性（Cross-Phase Consistency）
      ├─ 实体索引 ↔ 标注文本一致性
      ├─ 事件年代 ↔ 年表数据一致性
      └─ 关系三元组 ↔ 实体存在性验证

L4  业务逻辑验证（Business Logic Validation）
      ├─ 时序合理性（后序事件不应早于前序）
      ├─ 年份区间合理性（生年≤卒年）
      └─ 引用完整性（人物ID必须存在于实体库）

L3  数据完整性（Data Integrity）
      ├─ 必填字段非空
      ├─ 字段值域约束（如年份范围 -3000~2000）
      └─ 枚举值合法性（如事件类型必须在11种之内）

L2  语法正确性（Syntax Correctness）
      ├─ 标注边界完整（〖〗成对闭合）
      ├─ JSON/Markdown格式合法
      └─ 正则表达式匹配（如段落编号格式）

L1  文本完整性（Text Integrity）
```

```
    ├─ 编码正确性（UTF-8无乱码）
    ├─ 原文不变性（去标记后与原文逐字相同）
    └─ 文件存在性（必需文件齐全）
```

检查优先级原则

从L1到L5**逐层执行**，低层级未通过时不执行高层级检查（避免噪音）：

```
# 正确的执行顺序
python lint_text_integrity.py      # L1: 文本完整性
&& python lint_markdown.py          # L2: 语法正确性
&& python validate_data_schema.py   # L3: 数据完整性
&& python validate_logic.py         # L4: 业务逻辑
&& python cross_validate.py         # L5: 跨工序一致性
```

二、质量控制工具的设计模式

2.1 Lint工具设计规范

所有lint工具遵循以下约定：

命名规范：
- `lint_*.py` — 检查单一工序内的格式/语法错误
- `validate_*.py` — 检查数据完整性和业务逻辑
- `cross_validate_*.py` — 检查跨工序一致性

输入输出约定：

```
# 输入：文件路径或目录
python lint_markdown.py chapter_md/001_*.md     # 单文件
python lint_markdown.py chapter_md/             # 整个目录
python lint_markdown.py --all                   # 全部数据
```

```
# 输出格式：
# ✅ PASS（无错误） → 返回码0
# ❌ FAIL（有错误） → 返回码1，输出错误详情
```

输出格式标准：

```
文件：chapter_md/001_五帝本纪.tagged.md

❌ 错误 [L2-边界完整性]
  第42行：未闭合的人名标注
  位置：黄帝者，少典之子〖@姓公孙
  建议：补全闭合符号 → 〖@姓公孙〗

⚠️ 警告 [L3-数据完整性]
  第67行：空地名标注
  内容：〖=〗
  建议：删除空标注或补全内容

ℹ️ 信息 [统计]
  总实体标注：1,234
  错误数：2
  警告数：1

---
❌ 检查失败：发现2处错误，1处警告
```

错误分级：

级别	符号	说明	阻断发布?
ERROR	❌	必须修复，否则数据不可用	是
WARNING	⚠️	应该修复，但不影响基本功能	否
INFO	ℹ️	信息性提示，无需修复	否

2.2 核心Lint工具实例

实例1：`lint_markdown.py` — 标注文本格式检查

检查项（L1-L2）：

1. **文本完整性（L1）**
   - 文件编码为UTF-8
   - 去标记后与原文`chapter_numbered/*.txt`逐字相同

2. **标注边界完整性（L2）**
   ```python
   # 检查〖〗成对
   open_count = text.count('〖')
   close_count = text.count('〗')
   if open_count != close_count:
   error(f"〖{open_count}个 ≠ 〗{close_count}个")

# 检查嵌套（不允许〖〖@X〗〗）
if re.search(r'〖[^〗]*〖', text):
error("发现嵌套标注")

# 检查边界损坏（标注不应包含标点）
if re.search(r'〖@[^〗]*[，。、；：]', text):
error("人名标注包含标点")
```

1. **空标注检查（L2）**
   `python if re.search(r'〖[@=;#%&\'^~\*!\+\$]\s*〗', text): warning("发现空标注")`

2. **段落编号连续性（L2）**
   `python # [1], [1.1], [1.2], [2], [2.1] ... # 检查层级跳跃（不允许[1] → [1.2]） # 检查编号回退（不允许 [2.3] → [2.1]）`

使用：

```
# 检查单章
python lint_markdown.py chapter_md/001_五帝本纪.tagged.md

# 检查全部（CI/CD集成）
python lint_markdown.py --all --exit-on-error
```

实例2：`lint_ce_years.py` — 事件年代逻辑检查

检查项（L3-L4）：

1. **数据完整性（L3）**
   ```python
   # 检查年份格式
   valid_patterns = [
   r'(公元前\d+年)', # 确定纪年
   r'[约公元前\d+年]', # 不确定纪年
   r'[约公元前\d+—前\d+年]', # 区间纪年
   ]
   if not any(re.match(p, time_field) for p in valid_patterns):
   error(f"年份格式不合法: {time_field}")

# 检查年份范围合理性
if ce_year < -3000 or ce_year > 100:
warning(f"年份超出史记范围: {ce_year}")
```

1. **时序合理性（L4）**
   `python # 同章节内事件应按时间递增 for i in range(1, len(events)): if events[i].ce_year < events[i-1].ce_year: diff = events[i-1].ce_year - events[i].ce_year if diff > 3: # 允许3年容差 error(f"时序倒退: {events[i-1].name}({events[i-1].ce_year}) → {events[i].name}({events[i].ce_year})")`

2. **年份跳跃检测（L4）**
   `python # 相邻事件年份差距>100年，可能有错 for i in range(1, len(events)): diff = abs(events[i].ce_year - events[i-1].ce_year) if diff > 100: warning(f"年份跳跃: {events[i-1].name}`

`({events[i-1].ce_year}) → {events[i].name}({events[i].ce_year}), 差距{diff}年")`

3. **已知年份交叉验证（L4）**
   ```python
   KNOWN_DATES = {
   "孔子出生": -551,
   "秦始皇统一": -221,
   "项羽自刎": -202,
   }

for event in events:
for key, known_year in KNOWN_DATES.items():
if key in event.name and abs(event.ce_year - known_year) > 2:
error(f"{event.name} 年份{event.ce_year}与已知年份{known_year}偏差过大")
```

使用:

```
# 检查单章
python lint_ce_years.py 047

# 检查全部，生成报告
python lint_ce_years.py --all --report issues.tsv
```

实例3：`validate_tagging.py` — 原文不变性验证

核心检查（L1）：

```
def remove_all_tags(text):
    """去除所有标注符号，还原纯文本"""
    # 去除18类实体标注
    text = re.sub(r'〖[@=;#%&\'^~\*!\+\$]([^ 〖〗|]+?)(?:\|[^
〖〗]*)?〗', r'\1', text)
    # 去除段落编号
    text = re.sub(r'^\[\d+(?:\.\d+)*\]\s*', '', text,
flags=re.MULTILINE)
    # 去除Markdown标题/引用/分割线
```

```
    text = re.sub(r'^#{2,4}\s+.*$', '', text, flags=re.MULTILINE)
    text = re.sub(r'^>\s*', '', text, flags=re.MULTILINE)
    text = re.sub(r'^---+\s*$', '', text, flags=re.MULTILINE)
    return text.strip()

def validate_chapter(md_path, txt_path):
    """验证标注后文本与原文逐字相同"""
    original = Path(txt_path).read_text('utf-8')
    tagged = Path(md_path).read_text('utf-8')

    stripped = remove_all_tags(tagged)
    clean_original = remove_all_tags(original)

    # 标准化后逐字比对
    norm_stripped = re.sub(r'\s+', '', stripped)
    norm_original = re.sub(r'\s+', '', clean_original)

    if norm_stripped != norm_original:
        # 找出首个差异位置
        for i, (c1, c2) in enumerate(zip(norm_stripped,
norm_original)):
            if c1 != c2:
                context_start = max(0, i-20)
                context_end = min(len(norm_stripped), i+20)
                error(f"第{i}字符不同:\n"
                      f"  标注文本: ...
{norm_stripped[context_start:context_end]}...\n"
                      f"  原文: ...
{norm_original[context_start:context_end]}...")
                return False

        # 长度不同
        error(f"长度不同: 标注{len(norm_stripped)}字 ≠ 原文
{len(norm_original)}字")
        return False

    return True
```

使用:

```
# 验证单章
python validate_tagging.py --chapter 001

# 批量验证全部130章
python validate_tagging.py --all

# 验证失败时停止（CI/CD集成）
python validate_tagging.py --all --stop-on-error
```

三、跨工序一致性检查（L5）

3.1 实体索引一致性

检查目标:
- 标注文本中的实体在实体索引中必须存在
- 实体索引中的出处必须能在标注文本中找到

```
def cross_validate_entities():
    """实体索引 ↔ 标注文本双向验证"""
    # 从标注文本提取所有〖@人名〗
    tagged_persons = set()
    for md_file in CHAPTER_DIR.glob("*.tagged.md"):
        text = md_file.read_text()
        for match in re.finditer(r'〖@([^〗|]+)(?:\|([^〗]
+))?〗', text):
            display_name = match.group(1)
            canonical_name = match.group(2) or display_name
            tagged_persons.add(canonical_name)

    # 从实体索引加载所有人名
    entity_index = json.loads(Path("kg/
entity_index.json").read_text())
    indexed_persons = set(entity_index.get("persons", {}).keys())
```

```
    # 检查：标注中出现但索引中不存在
    missing_in_index = tagged_persons - indexed_persons
    if missing_in_index:
        warning(f"以下{len(missing_in_index)}个人名在标注文本中出现但实
体索引中缺失:\n"
                f"  {', '.join(list(missing_in_index)[:10])}")

    # 检查：索引中存在但标注中未出现（可能是消歧规范名）
    missing_in_text = indexed_persons - tagged_persons
    if len(missing_in_text) > len(indexed_persons) * 0.1:
        warning(f"实体索引中有{len(missing_in_text)}个人名在标注文本中
未找到，"
                f"占比{len(missing_in_text)/
len(indexed_persons):.1%}，可能存在问题")
```

3.2 事件年代一致性

检查目标：事件年代标注应与年表数据一致

```
def cross_validate_dates():
    """事件年代 ↔ 年表数据交叉验证"""
    # 加载年表数据
    reign_periods = json.loads(Path("kg/chronology/data/
reign_periods.json").read_text())

    # 加载所有事件
    for event_file in EVENTS_DIR.glob("*_事件索引.md"):
        events = parse_events(event_file)

        for event in events:
            # 提取纪年标记（如"秦昭王十三年"）
            reign_match = re.search(r'〖&([^〗]+)〗〖%([^〗]+)〗',
event.time_raw)
            if not reign_match:
                continue
```

```
            ruler = reign_match.group(1)
            year_in_reign = parse_year(reign_match.group(2))

            # 从年表查询公元年
            if ruler in reign_periods:
                reign_start = reign_periods[ruler]["start"]
                expected_ce = reign_start + year_in_reign - 1

                # 与事件标注的公元年对比
                if abs(event.ce_year - expected_ce) > 1:
                    error(f"{event.name}: 标注{event.ce_year}年 ≠
年表推算{expected_ce}年 "
                          f"({ruler}{year_in_reign}年)")
```

3.3 关系三元组完整性

检查目标：关系三元组中引用的实体ID必须存在

```
def validate_relation_integrity():
    """关系三元组 → 实体存在性验证"""
    entity_index = json.loads(Path("kg/
entity_index.json").read_text())
    all_entity_ids = set(entity_index.get("persons", {}).keys())
    all_entity_ids.update(entity_index.get("places", {}).keys())

    relations = json.loads(Path("kg/relations/
person_relations.json").read_text())

    for rel in relations:
        # 检查主语实体存在性
        if rel["subject"] not in all_entity_ids:
            error(f"关系主语不存在: {rel['subject']} (关系:
{rel['predicate']})")

        # 检查宾语实体存在性
```

```
        if rel["object"] not in all_entity_ids:
            error(f"关系宾语不存在: {rel['object']} (关系:
{rel['predicate']})")

        # 检查证据链接有效性
        if "evidence_chapter" in rel:
            evidence_file = CHAPTER_DIR /
f"{rel['evidence_chapter']}_*.tagged.md"
            if not list(CHAPTER_DIR.glob(evidence_file.name)):
                warning(f"关系证据章节不存在:
{rel['evidence_chapter']}")
```

四、质量控制工具链

4.1 标准工具链（按工序）

工序	工具	检查层次	执行频率
01 校勘	`lint_text_integrity.py`	L1	每次校勘后
02 结构分析	`lint_markdown.py`	L1-L2	每次编号/分段后
03 实体构建	`validate_tagging.py`	L1-L2	每次标注/反思后
	`lint_markdown.py`	L2	同上
	`cross_validate_entities.py`	L5	构建实体索引后
04 事件构建	`validate_events.py`	L2-L3	每次提取后

工序	工具	检查层次	执行频率
	`lint_ce_years.py`	L3-L4	每次年代标注/反思后
	`cross_validate_dates.py`	L5	年代标注完成后
05 关系构建	`validate_relations.py`	L2-L3	每次关系提取后
	`validate_relation_integrity.py`	L4-L5	关系构建完成后

4.2 CI/CD集成

Git pre-commit hook（`git add`后自动检查）：

```
#!/bin/bash
# .git/hooks/pre-commit

echo "运行质量检查..."

# L1-L2: 格式检查（必须通过）
python scripts/lint_markdown.py --all --exit-on-error || exit 1

# L3-L4: 业务逻辑检查（警告不阻断）
python kg/events/scripts/lint_ce_years.py --all

echo "✅ 质量检查通过"
```

GitHub Actions工作流（每次push触发）：

```
name: Quality Control

on: [push, pull_request]
```

```
jobs:
  lint:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2

      - name: L1-L2 格式检查
        run: |
          python scripts/lint_markdown.py --all --exit-on-error
          python scripts/validate_tagging.py --all --stop-on-error

      - name: L3-L4 业务逻辑检查
        run: |
          python kg/events/scripts/lint_ce_years.py --all --report
issues.tsv

      - name: L5 跨工序一致性检查
        run: |
          python kg/entities/scripts/cross_validate_entities.py
          python kg/events/scripts/cross_validate_dates.py

      - name: 上传问题报告
        if: failure()
        uses: actions/upload-artifact@v2
        with:
          name: quality-issues
          path: issues.tsv
```

五、为新工序设计质量控制工具

5.1 设计清单

当你为新工序设计质量控制工具时，按以下清单逐项确认：

Phase 1: 定义质量标准

- [ ] **明确输入输出格式**（JSON/Markdown/CSV？Schema是什么？）
- [ ] **列举必填字段**（哪些字段不能为空？）
- [ ] **定义值域约束**（年份范围？枚举值？字符串长度？）
- [ ] **识别业务规则**（时序约束？唯一性约束？引用完整性？）
- [ ] **确定跨工序依赖**（依赖哪些上游数据？如何验证一致性？）

Phase 2: 分层设计检查项

按五层模型逐层设计：

- [ ] **L1 文本完整性**：编码正确？文件存在？原文不变？
- [ ] **L2 语法正确性**：格式合法？边界完整？符号闭合？
- [ ] **L3 数据完整性**：必填非空？值域合法？Schema匹配？
- [ ] **L4 业务逻辑**：时序合理？约束满足？引用有效？
- [ ] **L5 跨工序一致性**：与上游数据一致？与下游依赖对接？

Phase 3: 实现工具

```
#!/usr/bin/env python3
"""
工序XX质量检查工具

检查项:
1. [L1] 文本完整性 ...
2. [L2] 格式正确性 ...
3. [L3] 数据完整性 ...
4. [L4] 业务逻辑 ...

用法:
    python lint_XX.py file.json
    python lint_XX.py --all
    python lint_XX.py --all --exit-on-error
"""
```

```
import sys
from pathlib import Path

def check_L1_integrity(data):
    """L1: 文本/文件完整性"""
    errors = []
    # TODO: 实现检查逻辑
    return errors

def check_L2_syntax(data):
    """L2: 语法正确性"""
    errors = []
    # TODO: 实现检查逻辑
    return errors

def check_L3_schema(data):
    """L3: 数据完整性"""
    errors = []
    # TODO: 实现检查逻辑
    return errors

def check_L4_logic(data):
    """L4: 业务逻辑"""
    errors = []
    # TODO: 实现检查逻辑
    return errors

def main():
    # 解析参数
    # 执行分层检查
    # 输出结果（✅/❌/⚠️）
    # 返回exit code
    pass

if __name__ == '__main__':
    sys.exit(main())
```

Phase 4: 集成到工作流

- [ ] **添加到Makefile/脚本**：一键执行所有检查
- [ ] **集成到Git hooks**：提交前自动检查
- [ ] **集成到CI/CD**：每次push触发
- [ ] **添加到SKILL文档**：反思流程的验证步骤

5.2 常见陷阱

1. **过度检查** — 不要检查"可能"的问题，只检查"确定"的错误
   - ❌"这个人名看起来不太对" → 太主观，留给反思
   - ✅"人名标注包含标点符号" → 明确违反格式规则
2. **检查不独立** — 每个工具应能独立运行，不依赖其他工具的输出
   - ❌ `lint_B.py` 依赖 `lint_A.py` 生成的中间文件
   - ✅ `lint_B.py` 直接读取源数据文件
3. **输出不友好** — 错误信息应指明位置+原因+建议修复
   - ❌"检查失败"
   - ✅"第42行：未闭合的人名标注，建议补全〗符号"
4. **没有容差** — 业务逻辑检查应允许合理误差
   - ❌年份相差1年就报错（古代纪年本身有歧义）
   - ✅年份相差>3年才报错
5. **忽略边界情况** — 处理空文件、空数据、极端值
   - 空文件怎么办？（应报warning而非crash）
   - 数据为空数组怎么办？（应报info "无数据"）
   - 年份为0怎么办？（公元1年前后没有公元0年）

六、质量控制的最佳实践

6.1 设计原则

1. **快速失败（Fail Fast）** — 低层级错误立即阻断，不执行高层级检查
2. **清晰反馈（Clear Feedback）** — 错误信息包含：位置+原因+建议修复

3. **分级输出（Tiered Output）** — ERROR必须修复，WARNING建议修复，INFO仅提示
4. **批量模式（Batch Mode）** — 支持`--all`参数批量检查所有文件
5. **CI/CD友好（CI/CD Friendly）** — 返回正确的exit code（0=PASS, 1=FAIL）

6.2 输出格式最佳实践

```
# ✅ 好的输出（结构化、可定位、有建议）
chapter_md/001_五帝本纪.tagged.md
  ❌ [L2-边界完整性] 第42行
     未闭合的人名标注：〖@姓公孙
     建议：补全闭合符号 → 〖@姓公孙〗

  ⚠️  [L3-数据完整性] 第67行
     空地名标注：〖=〗
     建议：删除空标注或补全内容

---
检查失败：1个ERROR，1个WARNING
详细报告：/tmp/lint_report_20260318.txt
```

```
# ❌ 差的输出（无结构、无定位、无建议）
Error: check failed
Some problems found in the file
```

6.3 测试驱动开发（TDD）

编写lint工具时，先写测试用例：

```
# tests/test_lint_markdown.py

def test_unclosed_tag():
    """测试未闭合标注检测"""
    text = "黄帝者，〖@少典之子"
    errors = lint_markdown(text)
```

```
    assert len(errors) == 1
    assert "未闭合" in errors[0].message

def test_empty_tag():
    """测试空标注检测"""
    text = "黄帝者，〖=〗之子"
    warnings = lint_markdown(text)
    assert len(warnings) == 1
    assert "空标注" in warnings[0].message

def test_valid_text():
    """测试合法文本不报错"""
    text = "〖@黄帝〗者，〖@少典〗之子"
    errors = lint_markdown(text)
    assert len(errors) == 0
```

6.4 性能优化

对于大规模数据（130章×数千行），优化策略：

1. **并行处理** — 多进程并行检查不同文件
   ```python
   from multiprocessing import Pool

with Pool(8) as pool:
results = pool.map(lint_file, all_files)
```

1. **增量检查** — 只检查修改过的文件（通过Git diff）
   `bash git diff --name-only HEAD~1 | grep "\.tagged\.md$" | xargs python lint_markdown.py`
2. **缓存结果** — 未修改的文件使用缓存结果
   `python cache_key = f"{file_path}:{file_stat.st_mtime}" if cache_key in cache: return cache[cache_key]`

七、质量指标与监控

7.1 核心质量指标

指标	定义	目标	监控方式
Lint通过率	通过lint的文件数/总文件数	100%	CI/CD每次执行
错误密度	错误数/总字符数	<0.1%	每周统计
警告清零率	已修复警告数/总警告数	>80%	每月统计
跨工序一致性	一致性检查通过的项/总检查项	>95%	每次发布前

7.2 质量仪表盘（Dashboard）

生成质量报告:

```
python scripts/generate_quality_report.py > quality_report.md
```

输出示例:

```
# 史记知识库质量报告

生成时间: 2026-03-18 14:30

## 总体指标

| 指标 | 当前值 | 目标 | 状态 |
|------|--------|------|------|
| Lint通过率 | 100% (130/130) | 100% | ✅ |
| 错误密度 | 0.03% (172/570K) | <0.1% | ✅ |
| 警告清零率 | 87% (1,234/1,420) | >80% | ✅ |
| 跨工序一致性 | 98% (49/50) | >95% | ✅ |

## 分工序质量
```

```
| 工序 | 错误数 | 警告数 | 通过率 |
|------|--------|--------|--------|
| 实体标注 | 0 | 120 | 100% |
| 事件提取 | 0 | 45 | 100% |
| 关系构建 | 12 | 18 | 90.8% |

## 待修复警告 Top 5

1. 空地名标注（45处） — SKILL_03c
2. 年份跳跃（23处） — SKILL_04d
3. 单字人名候选（18处） — SKILL_03c
4. 关系证据缺失（12处） — SKILL_05b
5. 邦国消歧待确认（8处） — SKILL_03b
```

八、总结

质量控制是知识库建设的**安全网**。通过五层质检体系，我们将错误拦截在不同阶段：

- **L1-L2（格式层）**：拦截80%的低级错误（边界损坏、语法错误）
- **L3-L4（逻辑层）**：拦截15%的中级错误（时序倒退、值域越界）
- **L5（一致性层）**：拦截5%的高级错误（跨工序不一致）

核心洞察：
1. **质量控制≠反思** — QC是规则驱动的快速检查，反思是语义驱动的智能审查
2. **自动化优先** — 能用规则检测的就不要用LLM，降低成本提升效率
3. **分层递进** — 低层级通过才执行高层级，避免噪音干扰
4. **持续监控** — 质量指标应纳入CI/CD，每次变更都触发检查

附录：参考资料

- `scripts/lint_markdown.py` — 标注文本格式检查完整实现
- `scripts/lint_text_integrity.py` — 原文不变性验证

- `kg/events/scripts/lint_ce_years.py` — 事件年代逻辑检查
- `kg/entities/scripts/validate_tagging.py` — 实体标注完整性验证
- `kg/events/scripts/cross_validate_dates.py` — 跨工序年代一致性检查
- META_反思.md — 质量控制的下游：反思方法论

元技能11: 数据体感培养 — 直接观察数据的十层体系

一、核心思想

> **"任何统计都替代不了对数据的直接观察"**

数据体感（Data Intuition）是通过反复观察原始数据和中间结果,建立对数据分布、异常模式和质量问题的直觉理解。

核心原则

1. **即时可见**:每个工序的中间结果必须可直接阅读(Markdown/JSON/TSV)
2. **多维观察**:统计汇总 + 随机抽样 + 异常检测 + 可视化
3. **人工介入**:定期人工检查,不完全依赖自动化指标
4. **辅助工具**:专门的调试脚本、质检工具、可视化界面

哲学

AI驱动的知识工程中,**人类专家的作用从"标注工人"转变为"质量审查者"和"模式发现者"**:

- 机器做重复劳动(标注、计算、推理)
- 人类做创造性工作(发现新模式、设计新规则、判断边界情况)
- **数据体感是连接两者的桥梁**:只有深入理解数据,才能设计出好的规则和prompt

二、本项目的数据体感实践

2.1 十层数据观察体系

层级	工具/方法	观察对象	产出
L1: 差异检测	`find_differences.py`	标注前后文本差异	字符级差异报告(Markdown)
L2: 统计分析	`analyze_word_frequency.py`	词频/n-gram分布	高频词表(Top 100)
L3: 覆盖率分析	`analyze_tagged.py`	标注覆盖率/类型分布	18类实体统计表
L4: 专项扫描	`scan_biology_reflect.py`	特定类型实体完整性	问题清单(TSV)
L5: 格式验证	`validate_events.py`	事件索引格式/完整性	验证报告
L6: 逻辑验证	`cross_validate_dates.py`	跨数据源一致性	偏差报告+修正建议
L7: 采样检查	`review_disambiguation.py`	歧义消解质量	按章审查报告
L8: 可视化	HTML/Markdown渲染	实体索引/事件地铁图	交互式界面
L9: 工作日志	`logs/daily/*.md`	每日修正记录	Git提交汇总
L10: 反思总结	`doc/events/*反思总结.md`	多轮迭代收敛过程	模式发现文档

2.2 中间输出设计

原则1:双格式输出(JSON + Markdown)

所有数据文件同时提供机读和人读格式

```
kg/events/data/
├── 001_五帝本纪_事件索引.md          # 人读:表格+详情
├── event_relations.json               # 机读:结构化数据
└── event_relations_summary.md         # 人读:统计摘要
```

示例:事件索引Markdown格式(`001_五帝本纪_事件索引.md`)

```
## 概览表格

| 事件ID | 事件名 | 事件类型 | 主要人物 | CE年份 |
|--------|--------|---------|---------|--------|
| 001_001 | 黄帝战蚩尤 | 战争 | 黄帝, 蚩尤 | ~-2500? |
| 001_002 | 黄帝制历法 | 文化 | 黄帝 | ~-2500? |

## 详细记录

### 001_001 黄帝战蚩尤
- **事件类型**: 战争
- **主要人物**: 黄帝, 蚩尤
- **CE年份**: ~-2500?
- **描述**: 黄帝与蚩尤战于涿鹿之野...
```

好处:
- 人类可直接阅读(无需工具)
- Git diff友好(修改一目了然)
- 支持全文搜索(grep/ripgrep)
- Markdown可渲染为HTML(进一步可视化)

原则2:中间结果文件化

每个处理阶段保存输出,支持断点续跑和问题定位

```
# 示例: extract_wars.py 的五阶段输出

# Stage 1: 识别候选
with open('tmp/wars_raw.json', 'w') as f:
    json.dump(raw_wars, f, ensure_ascii=False, indent=2)

# Stage 2: 按名称分组
with open('tmp/wars_grouped.json', 'w') as f:
    json.dump(name_groups, f, ensure_ascii=False, indent=2)

# Stage 3: 合并去重
with open('tmp/wars_merged.json', 'w') as f:
    json.dump(merged_wars, f, ensure_ascii=False, indent=2)

# Stage 5: 最终输出
with open('data/wars.json', 'w') as f:  # 机读
    json.dump(final_wars, f, ensure_ascii=False, indent=2)

generate_markdown_index(final_wars, 'data/战争事件索引.md')  # 人读
```

调试工作流:

```
# 发现问题: wars.json中某场战争交战方错误
# 回溯检查:
cat tmp/wars_raw.json | grep "长平之战"       # 原始提取正确
cat tmp/wars_grouped.json | grep "长平之战"  # 分组阶段引入错误
# → 定位到Stage 2的group逻辑有bug
```

2.3 差异检测与文本完整性

工具: `find_differences.py` & `report_differences.py`

问题:标注过程可能意外改动原文(增删字符、替换标点等)

解决:字符级差异检测

```
def find_char_differences(str1, str2):
    """找出两个字符串之间的字符级差异"""
    diffs = []
    matcher = difflib.SequenceMatcher(None, str1, str2)
    for tag, i1, i2, j1, j2 in matcher.get_opcodes():
        if tag != 'equal':
            diffs.append({
                'type': tag,           # replace/insert/delete
                'orig': str1[i1:i2],
                'tagged': str2[j1:j2],
                'pos': i1,
                'context_before': str1[max(0,i1-20):i1],
                'context_after': str1[i2:min(len(str1), i2+20)]
            })
    return diffs
```

输出示例 (`doc/analysis/差异报告.md`):

```
### 001 五帝本纪

段落组 [005]
============
前100字符匹配,差异从位置 42 开始

原文: ...黄帝居轩辕之丘,而娶于西陵之女...
标注: ...黄帝居轩辕之丘,而'娶于西陵之女...

差异类型: insert
插入内容: '
```

```
位置: 42

**判断**: 标注错误,应删除多余标点
```

人工审查流程:
1. 运行 `python scripts/report_differences.py` 生成报告
2. 打开Markdown文件,逐段检查
3. 区分"合法标注"(〖〗)和"意外改动"
4. 修正错误,重新运行lint

2.4 统计分析:全局视角

工具1: `analyze_word_frequency.py` - 词频分布

输出 (`doc/analysis/词频统计表.md`):

```
## 高频字符 Top 100

| 排名 | 字符 | 次数 | 示例 |
|------|------|------|------|
| 1 | 之 | 12,345 | 黄帝之子, 三年之后 |
| 2 | 曰 | 8,230 | 太史公曰, 孔子曰 |
| 3 | 以 | 7,890 | 以为, 以此 |

## 高频双字词 Top 100

| 排名 | 词 | 次数 | 类型推测 |
|------|-----|------|---------|
| 1 | 天子 | 1,234 | 身份词 |
| 2 | 诸侯 | 987 | 身份词 |
| 3 | 丞相 | 756 | 官职词 |
```

发现新实体类型的案例:

```
## 未标注高频三字词(候选新类型)

| 词 | 次数 | 观察 |
|-----|------|------|
| 封禅 | 45 | 礼仪词汇(未覆盖) |
| 宗庙 | 38 | 礼仪词汇 |
| 社稷 | 32 | 礼仪词汇 |
| 腰斩 | 28 | 刑法词汇(未覆盖) |
| 族诛 | 24 | 刑法词汇 |
```

行动:
- 发现模式 → 新增类型 `〖:礼仪〗`、`〖[刑法〗`
- 更新标注规范 v2.4 → v2.5
- 重新标注相关章节

工具2: `analyze_tagged.py` - 标注覆盖率

输出示例:

```
## 整体统计

- 总中文字符数(去标点): 572,340
- 已标注字符数: 156,789 (27.4%)
- 未标注字符数: 415,551 (72.6%)

## 按类型分布

| 类型 | 字符数 | 占比 | 平均长度 |
|------|--------|------|---------|
| @人名 | 45,230 | 7.9% | 2.1字 |
| =地名 | 38,120 | 6.7% | 2.3字 |
| ;官职 | 28,450 | 5.0% | 2.5字 |
| '邦国 | 24,890 | 4.4% | 1.8字 |
| %时间 | 19,560 | 3.4% | 2.0字 |
```

```
## 异常检测

⚠️ 平均长度异常:
- `〖;太傅左丞相〗` (5字,超过均值2σ)
- `〖=河南郡洛阳县〗` (6字,建议拆分)
```

人工检查:
- 打开具体章节,查看长实体是否合理
- 判断是否需要拆分(如 `太傅` 和 `左丞相` 是两个官职)
- 更新标注规范

2.5 专项扫描:类型完整性

工具: `scan_biology_reflect.py`

三类检查:

1. **未知实体**:文本中标注了 `〖+X〗`,但X不在词表内
2. **跨类型冲突**:同一词既标 `〖+生物〗` 又标 `〖@人名〗`
3. **遗漏标注**:词表中的词在文本中出现但未标注

输出 (`doc/analysis/biology_scan_report.tsv`):

```
chapter type    entity  detail  context
001 unknown_bio 麒麟  标注了〖+麒麟〗但不在词表内  ...黄帝出〖+麒麟〗于东郊...
012 conflict    龙   同时标注为〖+龙〗和〖@龙(人名)〗  ...〖@龙〗曰:"吾见〖+龙〗升天"
003 missing 凤   词表有"凤"但文本未标注    ...有凤来仪(未标注)
```

人工决策:

```
### 001 麒麟
- 决策: 加入词表 `biology_wordlist.json`
- 分类: Animal > MythicalAnimal > 麒麟
```

```
### 012 龙(冲突)
- 上下文1: "龙曰" → 人名(传说人物,如史记中的"龙"姓)
- 上下文2: "龙升天" → 生物(神话动物)
- 决策: 保持冲突,按上下文消歧

### 003 凤(遗漏)
- 决策: 补充标注 `〖+凤〗来仪`
```

2.6 采样检查:歧义消解质量

工具: `review_disambiguation.py`

按章生成审查报告:

```
### 047 仲尼弟子列传

#### 统计
- 人名标记: 524处
- 不同人名: 87个
- 歧义短名: 赵(3处), 石(2处), 高(3处)

#### 当前消歧映射
| 短名 | 全名 | 章内出现 | 依据 |
|------|------|----------|------|
| 赵 | 赵本学 | 3处 | 章节主题(仲尼弟子) |
| 石 | 石作蜀 | 2处 | 同上 |

#### 别名组(章内多称谓)
- 孔子: 孔丘(156), 尼(45), 夫子(23)
- 颜回: 颜回(12), 回(8), 颜渊(5)

#### 问题发现
⚠️ 缺失消歧: "高"(3处出现)但无映射规则
⚠️ 幽灵条目: "古"在映射中有`古→古冶子`,但文本中无此标注
```

```
#### 建议
- [ ] 补充"高"的消歧规则(需人工判断是人名还是形容词)
- [ ] 删除幽灵条目"古"(或检查是否标注遗漏)
```

人工审查:
1. 打开047_仲尼弟子列传.tagged.md
2. 搜索"高"的3处出现
3. 判断每处是否为人名
4. 更新`disambiguation_map.json`

2.7 可视化:交互式探索

实体索引页(`docs/entities/index.html`)

功能:
- 18类实体分页展示
- 搜索框(实时过滤)
- 点击实体 → 跳转到章节位置
- 统计卡片(总数/章节分布)

数据流:

```
kg/entities/data/entity_index.json
  ↓ [build_entity_index.py]
docs/entities/*.html (18个类型页 + 总览页)
  ↓ [浏览器]
交互式探索界面
```

用户体验:

```
用户: "想看所有标注为'战争'的事件"
  ↓ 打开 docs/entities/event.html
  ↓ 筛选器: 类型=战争
  ↓ 结果: 736场战争,按时间排序
  ↓ 点击"长平之战" → 跳转到chapter_md/005_秦本纪.tagged.md#E005_042
```

地铁图可视化（`app/metro/`）

理念:将事件关系抽象为地铁路线图

- **130条线路** = 130篇章节
- **3,185个站点** = 3,185个事件
- **换乘连线** = 跨章事件关系(互见/共人/共地/同期)
- **时间轴** = 公元前2700年～前87年

数据生成（`kg/events/scripts/build_metro_map_data.py`）:

```
metro_data = {
    "lines": [
        {
            "id": "001",
            "name": "五帝本纪",
            "color": "#FF6B6B",
            "stations": [
                {"id": "001_001", "name": "黄帝战蚩尤", "year":
-2500},
                {"id": "001_002", "name": "黄帝制历法", "year":
-2500},
            ]
        },
        # ... 130 lines
    ],
    "transfers": [
        {"from": "001_001", "to": "013_005", "type": "cross_ref"},
        # ... 1,876 transfers
    ]
}
```

交互功能:
- 缩放/平移(探索2700年跨度)
- 点击站点 → 显示事件详情

- 高亮换乘线 → 查看跨章关系
- 时间轴筛选 → 只看某时代事件

数据体感收益:
- **直观理解**:事件的时空分布(哪些时代事件密集)
- **发现异常**:孤立事件(无换乘连线)可能是标注遗漏
- **验证关系**:换乘线的合理性(如秦汉之间应有大量换乘)

2.8 工作日志:迭代轨迹追踪

自动生成日志(`logs/daily/2026-03-17.md`)

数据来源:Git提交记录(按自然日7:00-7:00划分)

```
# 2026-03-17 工作日志

## 提交统计
- 总提交数: 12
- 修改文件数: 45
- 新增行数: 1,234
- 删除行数: 567

## 按类型分类

### 反思规则更新 (4次提交)
- [eb97490] 成语专项索引:完整提取与多格式输出
- [5ef3816] 韵文专项索引:96篇赞/诗歌/赋完整收录
- ...

### 实体反思 (3次提交)
- [054d294] 更新工作日志:完整记录第一轮反思
- ...

### 可视化 (2次提交)
- [cf9f4f5] 太史公曰PDF导出功能
- ...
```

```
## 核心成果
1. 完成第一轮事件年代反思(118/130章有修正)
2. 发现25条新的年代推断模式
3. 建立区块标签系统(太史公曰/韵文/成语)
```

人工review价值:
- **回顾进展**:每周五review本周日志,评估效率
- **发现模式**:某类提交频繁出现 → 需要自动化工具
- **记录决策**:为什么某天集中修正某类问题

2.9 反思总结:模式沉淀

五轮事件年代反思总结文档

文件:
- `doc/events/第一轮事件年代反思总结.md` (1,010处修正)
- `doc/events/第二轮事件年代反思总结.md` (431处修正)
- `doc/events/第三轮事件年代反思总结.md` (465处修正)
- `doc/events/第四轮事件年代反思总结.md` (167处修正)
- `doc/events/第五轮事件年代反思总结.md` (46处修正)

内容结构:

```
# 第一轮事件年代反思总结

## 一、整体情况
- 修正章节: 118/130
- 修正事件: 1,010
- 主要问题: 系统性年份错误、确定性标注不足

## 二、发现的新模式(25条)

### 模式1: 单锚点塌缩
**问题**: 多个事件共享同一锚点(如"即位"),年份全部相同
**示例**: 秦始皇在位37年的事件都标注为-246(即位年)
**修正**: 根据"X年"字段精确计算: 即位+X-1
```

```
### 模式2: 生卒年中点误用
**问题**: 崩逝事件标注为(生年+卒年)/2,而非卒年
**示例**: 黄帝崩 → 应为-2400(卒年),而非-2450(中点)
**修正**: 崩/薨/卒事件 → 取卒年

... (25条模式详细描述)

## 三、典型案例

### 案例1: 005_秦本纪
修正数: 78处
主要问题: 即位年锚点污染,所有事件都是-246
修正方法: 重新建立多锚点体系(即位/统一/巡游/崩)

... (10个典型案例)

## 四、经验总结
1. LLM对"X年"的理解不稳定,需明确prompt
2. 单锚点系统容易塌缩,需要多锚点+插值
3. 跨章验证非常重要(同一事件不同章节年份应一致)

## 五、遗留问题
1. 神话时代年份仍高度不确定(黄帝/尧/舜)
2. 部分事件缺乏明确纪年依据,标注为"~-XXXX?"

## 六、下一轮计划
- 重点: 确定性升级(将"~-XXXX?"提升为"-XXXX")
- 方法: 利用第一轮建立的锚点进行插值推断
```

数据体感价值:
- **量化进展**:从1,010 → 431 → 465 → 167 → 46,看到收敛趋势
- **模式积累**:25条模式 → 转化为SKILL文档规则
- **异常定位**:遗留问题清单 → 下一轮重点关注

2.10 辅助脚本:临时调试工具

原则:遇到问题立即写小脚本,不要手工操作

示例1: 检查特定模式

```
# scripts/check_single_char_names.py
# 任务: 找出所有单字人名(可能是标注错误)

import re, glob

for fpath in glob.glob('chapter_md/*.tagged.md'):
    with open(fpath) as f:
        text = f.read()

    # 提取所有〖@人名〗
    persons = re.findall(r'〖@([^〗]+)〗', text)

    # 筛选单字
    single_chars = [p for p in persons if len(p) == 1]

    if single_chars:
        print(f"{fpath}: {set(single_chars)}")

# 输出示例:
# chapter_md/005_秦本纪.tagged.md: {'政', '高', '信'}
# → 人工检查: "政"(秦王政)应为"〖@嬴政〗", "高"可能是形容词误标
```

示例2: 批量统计

```
# 快速统计: 每章的战争事件数
grep -h "战争" kg/events/data/*_事件索引.md | \
  cut -d'|' -f2 | \
  sort | uniq -c | sort -rn

# 输出:
#  45 005_秦本纪
```

```
#  38 006_秦始皇本纪
#  32 008_高祖本纪
# → 发现: 秦本纪战争最多,符合历史(战国时代)
```

示例3: 可视化脚本

```
# scripts/plot_event_timeline.py
# 绘制事件时间轴分布图

import json, matplotlib.pyplot as plt

with open('kg/events/data/event_relations.json') as f:
    events = json.load(f)

years = [e['ce_year'] for e in events if e.get('ce_year')]

plt.hist(years, bins=50, alpha=0.7)
plt.xlabel('CE Year')
plt.ylabel('Event Count')
plt.title('Event Distribution Timeline')
plt.savefig('doc/analysis/event_timeline.png')

# → 发现: -500到-200年事件最密集(战国+秦汉)
```

三、方法论指导

3.1 数据观察清单

每个工序开发时必做:

- [ ] **样本检查**(10-20个):手动查看输入/输出样本
- [ ] **统计汇总**:生成count/avg/min/max等统计量
- [ ] **异常检测**:识别超出2σ的离群值
- [ ] **随机抽样**:从结果中随机抽取50个人工审查

- [ ] **边界测试**:检查极端情况(最长/最短/空值/特殊字符)
- [ ] **可视化**(如适用):绘制分布图/时间轴/关系网络

3.2 中间输出设计原则

原则1: 人类可读优先

反例:

```
{"e":[{"i":"001_001","t":3,"p":[12,34],"d":"..."}]}  // 压缩格式,难以阅读
```

正例:

```
{
  "events": [
    {
      "event_id": "001_001",
      "event_type": "战争",
      "persons": [12, 34],
      "description": "黄帝战蚩尤于涿鹿之野"
    }
  ]
}
```

额外建议:
- JSON使用`indent=2`(美化格式)
- 字段名用英文全称(不缩写)
- 加注释字段(如`"_note": "此字段用于..."`)

原则2: 分阶段存储

不要只保存最终结果,每个处理阶段都应有输出

```
# ❌ 错误: 黑箱流水线
result = step1(data)
result = step2(result)
result = step3(result)
save(result, 'final.json')  # 只保存最终结果

# ✅ 正确: 分阶段输出
result1 = step1(data)
save(result1, 'tmp/stage1_output.json')

result2 = step2(result1)
save(result2, 'tmp/stage2_output.json')

result3 = step3(result2)
save(result3, 'data/final.json')
save_markdown(result3, 'data/final.md')  # 人读格式
```

收益:
- 出问题可定位具体阶段
- 支持断点续跑(跳过已完成阶段)
- 方便A/B测试(只替换某个阶段)

原则3: 元数据记录

每个输出文件包含生成信息

```
{
  "_metadata": {
    "generated_at": "2026-03-17T14:32:10Z",
    "generator": "extract_wars.py v2.3",
    "input_files": ["kg/events/data/*_事件索引.md"],
    "config": {
      "min_confidence": 0.8,
      "merge_threshold": 0.6
    },
```

```
    "statistics": {
      "total_wars": 736,
      "merged_groups": 124,
      "avg_confidence": 0.91
    }
  },
  "data": [...]
}
```

用途:
- 可复现(知道如何生成)
- 可追溯(知道何时由谁生成)
- 可对比(不同版本的统计差异)

3.3 采样策略

策略1: 分层随机抽样

不要只看前10个,要覆盖不同类型/章节/时代

```
import random

def stratified_sample(events, n=50):
    """按事件类型分层抽样"""
    samples = []

    # 按类型分组
    by_type = {}
    for e in events:
        by_type.setdefault(e['type'], []).append(e)

    # 每个类型抽样
    for type_name, type_events in by_type.items():
        k = max(1, int(n * len(type_events) / len(events)))  # 按
比例
        samples.extend(random.sample(type_events, min(k,
```

```
len(type_events))))

    return samples[:n]
```

策略2: 边界情况优先

主动查看极端值

```
# 最长/最短描述
events.sort(key=lambda e: len(e['description']))
print("最短描述:", events[0])
print("最长描述:", events[-1])

# 人物最多/最少
events.sort(key=lambda e: len(e['persons']))
print("无人物事件:", [e for e in events if len(e['persons'])==0])
print("人物最多:", events[-1])

# 年份最早/最晚
events.sort(key=lambda e: e.get('ce_year', 0))
print("最早事件:", events[0])
print("最晚事件:", events[-1])
```

策略3: 错误驱动采样

优先检查自动化质检发现的问题

```
# 从lint工具输出中提取问题事件
with open('lint_errors.json') as f:
    errors = json.load(f)

error_events = [e['event_id'] for e in errors]

# 人工逐一检查
```

```
for eid in error_events:
    event = load_event(eid)
    print(f"\n{'='*60}")
    print(f"Event: {eid}")
    print(f"Error: {errors[eid]['message']}")
    print(f"Description: {event['description']}")

    # 等待人工判断
    action = input("Action (fix/ignore/skip): ")
    if action == 'fix':
        # 进入交互式修正流程
        ...
```

3.4 可视化设计

原则: 多视角观察

同一份数据,提供多种可视化视角

```
# 事件数据的四种可视化

# 视角1: 时间轴(按年份排序)
generate_timeline_html(events, 'timeline.html')

# 视角2: 地铁图(章节为线路)
generate_metro_map(events, 'metro.html')

# 视角3: 关系网络(人物-事件图)
generate_network_graph(events, 'network.html')

# 视角4: 表格索引(可搜索/筛选)
generate_table_view(events, 'table.html')
```

不同视角发现不同问题:
- 时间轴:发现年份异常(如公元500年的秦朝事件)

- 地铁图:发现孤立章节(无跨章关系)
- 网络图:发现中心人物(连接最多事件)
- 表格:发现字段缺失(空值统计)

3.5 人工审查原则

原则1: 问题驱动的按需审查

不要等待定期检查,在以下时机立即人工审查:

- **工具预警时**: lint/validate工具报错 → 抽查相关数据
- **数据变更后**: 批量修改、重新标注、格式转换后 → 抽查样本
- **发现异常时**: 统计数据出现突变(如实体数突降50%) → 追溯原因
- **引入新规则时**: 新增标注类型、修改prompt → 验证是否符合预期

原则2: 分层审查策略

根据问题严重性,选择审查深度:

严重性	审查范围	方法
高 (文本完整性破坏)	100%人工检查	逐一审查所有差异
中 (逻辑错误如年份倒置)	抽样30-50个	随机+边界+异常值
低 (格式不统一)	抽样5-10个	快速验证修正方向

原则3: 快速验证工作流

从发现问题到修正的完整循环:

```
# 步骤1: 工具发现异常
python scripts/lint_all.py
# → 输出: 发现23个单字人名,可能是标注错误

# 步骤2: 生成待审查清单
python scripts/check_single_char_names.py > tmp/review_list.txt
```

```
# → 输出: 005_秦本纪.tagged.md: {政, 高, 信}

# 步骤3: 人工快速判断
cat tmp/review_list.txt | head -10  # 先看前10个
# → "政"出现12次,上下文都是"秦王政",应补全为"嬴政"
# → "高"出现3次,2次是形容词"高台",1次是人名误标

# 步骤4: 批量修正
python scripts/fix_by_pattern.py --replace "〖@政〗" "〖@嬴政〗"
# 或手动修改少量特例

# 步骤5: 验证修正效果
python scripts/lint_all.py
# → 确认问题已解决,错误数从23降至0
```

核心: 发现问题 → 立即修正 → 不要积累

原则4: 利用Git追溯变更

变更后的例行检查:

```
# 查看最近修改了哪些文件
git status
git log --oneline -10

# 对比修改前后的关键指标
python scripts/quick_stats.py > /tmp/stats_now.txt
git stash
python scripts/quick_stats.py > /tmp/stats_before.txt
git stash pop
diff /tmp/stats_before.txt /tmp/stats_now.txt

# 抽查修改的文件
git diff HEAD~1 chapter_md/005_秦本纪.tagged.md | less
```

发现数据倒退 → 立即回滚或修正

四、工具清单

4.1 差异与完整性检查

工具	功能	输出
`find_differences.py`	标注前后文本差异检测	差异清单(JSON)
`report_differences.py`	生成人读差异报告	Markdown报告
`lint_text_integrity.py`	UTF-8编码/原文不变性检查	错误清单

4.2 统计分析

工具	功能	输出
`analyze_word_frequency.py`	词频/n-gram统计	高频词表(Markdown+JSON)
`analyze_tagged.py`	标注覆盖率/类型分布	统计表(Markdown)
`count_entities.py`	按类型统计实体数	18类实体统计

4.3 专项扫描

工具	功能	输出
`scan_biology_reflect.py`	生物实体完整性检查	问题清单(TSV)

工具	功能	输出
`review_disambiguation.py`	人名消歧质量审查	按章报告(Markdown)
`check_single_char_names.py`	单字人名检测(可能误标)	候选清单
`audit_interpolated_dates.py`	插值年份审计	审计报告

4.4 交叉验证

工具	功能	输出
`cross_validate_dates.py`	事件年份跨数据源验证	偏差报告+修正建议(JSON)
`validate_events.py`	事件索引格式/完整性验证	验证报告(Markdown)
`validate_data_schema.py`	JSON schema验证	错误清单

4.5 可视化

工具	功能	输出
`build_entity_index.py`	实体索引HTML生成	18类实体页+总览页
`build_metro_map_data.py`	地铁图数据生成	JSON数据+HTML界面
`plot_event_timeline.py`	事件时间轴绘图	PNG图片

工具	功能	输出
`generate_network_graph.py`	人物-事件关系网络	HTML交互图

4.6 辅助脚本模板

快速创建新的调试脚本:

```
# scripts/inspect_template.py
"""
临时调试脚本模板
用途: [描述具体检查什么问题]
作者: [你的名字]
日期: [YYYY-MM-DD]
"""

import json, glob, re

def main():
    # 1. 加载数据
    data = load_data()

    # 2. 过滤/筛选
    filtered = filter_by_condition(data)

    # 3. 统计/分析
    stats = analyze(filtered)

    # 4. 输出结果
    print_summary(stats)
    save_report(filtered, 'tmp/inspect_result.json')

def load_data():
    """加载需要检查的数据"""
    # TODO: 实现数据加载逻辑
```

```
    pass

def filter_by_condition(data):
    """按条件筛选"""
    # TODO: 实现筛选逻辑
    pass

def analyze(data):
    """统计分析"""
    # TODO: 实现分析逻辑
    pass

def print_summary(stats):
    """打印摘要"""
    print("=" * 60)
    print("检查结果摘要")
    print("=" * 60)
    for key, value in stats.items():
        print(f"{key}: {value}")

def save_report(data, output_path):
    """保存详细报告"""
    with open(output_path, 'w', encoding='utf-8') as f:
        json.dump(data, f, ensure_ascii=False, indent=2)
    print(f"\n详细报告已保存: {output_path}")

if __name__ == '__main__':
    main()
```

使用方法:

```
cp scripts/inspect_template.py scripts/check_XXX.py
# 修改TODO部分,实现具体逻辑
python scripts/check_XXX.py
```

五、典型案例

案例1: 发现"单锚点塌缩"模式

背景: 第一轮事件年代反思,发现秦本纪78个事件都标注为-246年

数据观察:

```
# 快速检查
cat kg/events/data/005_秦本纪_事件索引.md | grep "CE年份" | sort |
uniq -c

# 输出:
#   78 | CE年份 | -246 |

# → 异常! 37年在位期间的事件不可能都是-246
```

深入分析:

```
# scripts/analyze_qin_dates.py
events = load_events('005')
for e in events:
    print(f"{e['id']}: {e['name']} | {e['year_field']} |
{e['ce_year']}")

# 输出:
# 005_001: 秦王政即位 | 即位 | -246
# 005_002: 长安君成蟜反 | 三年 | -246  ← 应为-243
# 005_003: 攻取榆次 | 九年 | -246      ← 应为-237
# ...
```

根因: LLM将所有事件锚定到"即位"这个单一锚点,未根据"X年"字段计算

修正方案:
1. 更新prompt,明确"X年"的计算规则
2. 建立多锚点体系(即位/统一/巡游/崩)
3. 重新运行Agent反思

沉淀:
- 新lint规则: 检测"同一章节>50%事件年份相同"
- 新SKILL模式: "单锚点塌缩及其避免方法"

案例2: 词表遗漏的系统性发现

背景: 生物实体标注后,发现部分常见动物未覆盖

数据观察:

```
# 查看已标注的生物实体
grep -oP '〖\+[^〗]+〗' chapter_md/*.tagged.md | \
  sed 's/〖+//; s/〗//' | sort | uniq -c | sort -rn

# 输出 Top 20:
#  102 马
#   87 龙
#   65 犬
#   45 牛
#   ...

# 查看未标注文本中的动物候选词
python scripts/find_unmarked_animals.py

# 输出:
# 候选: 象(23次), 虎(19次), 鹿(18次), 豹(12次)
# → 都是常见动物,应该标注但遗漏了
```

根因: 初版词表(v1.0)只有50种常见动物,遗漏了部分

修正方案:
1. 人工审查23次"象"的上下文(确认都是动物,非地名"大象")
2. 补充词表: 象/虎/鹿/豹/熊/... (+38种)
3. 重新运行标注Agent

沉淀:
- 定期运行`find_unmarked_X.py`(X=animals/places/officials等)
- 词表版本管理(v1.0 → v1.1 → v2.0)

案例3: 可视化发现的年份异常

背景: 绘制事件时间轴时,发现一个事件年份为公元500年

数据观察:

```
# plot_event_timeline.py 输出警告
# WARNING: Event 047_012 has year=500, likely error (should be
negative)
```

深入检查:

```
cat kg/events/data/047_仲尼弟子列传_事件索引.md | grep "047_012"

# 输出:
# | 047_012 | 孔子卒 | 家族 | 孔子 | 500 |
#                                       ↑ 应为-500
```

根因: 数据录入错误,遗漏负号

修正: 手动修正为`-500`

沉淀:
- 新lint规则:`assert all(e['ce_year'] < 0 for e in events)`(史记时代无正年份)
- 可视化预警: 自动高亮异常值

六、总结

核心要点

1. **即时可见**:中间结果必须人类可读(Markdown/JSON美化)
2. **多维观察**:统计+抽样+可视化+异常检测
3. **工具驱动**:写脚本自动化检查,不要手工统计
4. **定期review**:每周固定时间人工审查数据
5. **模式沉淀**:发现的问题 → lint规则 → SKILL文档

数据体感培养三原则

原则1: 不信任统计,信任样本
- ✅ "精度95%"不够,要看具体错在哪5%
- ✅ 随机抽50个样本人工检查
- ❌ 不要只看汇总数字

原则2: 不信任自动化,信任人工验证
- ✅ 自动化工具(lint/validate)找出候选问题
- ✅ 人工最终判断每个问题是否真实
- ❌ 不要盲目信任工具输出

原则3: 不信任一次观察,信任多轮迭代
- ✅ 第一次看数据一定会遗漏问题
- ✅ 每周review,每轮反思,持续观察
- ❌ 不要"一次性检查完成"

与其他元技能的关系

- ← **可读性**:数据体感的前提是数据可读
- ← **柳叶刀方法**:分模块输出,分模块观察
- ← **反思**:观察数据 → 发现问题 → 反思修正
- ← **质量控制**:观察数据 → 设计lint规则
- ← **冷启动**:观察小样本 → 发现模式 → 扩展规则

数据体感是所有元技能的基础:只有深入理解数据,才能设计好的规则、prompt和质检工具。

元技能12: 数据融合 — 多源数据对齐与消歧

一、核心思想

> **"来自不同工序的结果,需要对齐、消歧、去重,形成统一的知识视图"**

数据融合（Data Fusion）是将来自多个来源、多种格式、多个粒度层次的数据整合为一致、准确、完整的知识表示的过程。

核心挑战

1. **同一实体,不同表述**:刘邦 = 沛公 = 汉王 = 高祖 = 高帝
2. **同一事件,多章记述**:秦本纪、六国年表、赵世家对长平之战的不同描述
3. **跨工序一致性**:事件索引的年份 ↔ 年表数据 ↔ 人物生卒年
4. **冲突解决**:不同来源的矛盾数据如何处理?

融合层次

```
L1: 字符串层面 → 归一化、去重、别名映射
L2: 实体层面 → 消歧、规范化、别名合并
L3: 事件层面 → 跨章合并、多源整合
L4: SKU层面 → 结构化知识单元的拼接
L5: 知识图谱层面 → 全局一致性、关系网络
```

二、字符串层面融合（L1）

2.1 别名检测与映射

工具: `auto_detect_aliases.py`

问题:同一人物有多个称谓

```
原始标注:
- 〖@高祖〗立为天子
- 〖@沛公〗起兵
- 〖@汉王〗入关中
- 〖@刘邦〗者,沛丰邑人也
```

目标:建立规范映射 `entity_aliases.json`

```
{
  "刘邦": ["沛公", "汉王", "高祖", "高帝"]
}
```

策略1:前缀包含检测

规则:长名包含短名 + 同章共现

```
def detect_prefix_aliases(chapter_entities):
    """
    检测包含关系:
    - "秦庄襄王" contains "庄襄王" → alias
    - "魏公子信陵君" contains "信陵君", "公子" → aliases
    """
    short_to_longs = defaultdict(set)

    for long_name in chapter_entities:
        # 提取可能的短名(去除前缀)
        for short in extract_short_forms(long_name):
```

```
            if short in chapter_entities:  # 短名也出现在同章
                short_to_longs[short].add(long_name)

    # 唯一映射 → 自动确认
    for short, longs in short_to_longs.items():
        if len(longs) == 1:
            auto_confirmed.append((list(longs)[0], short))
        else:
            ambiguous.append((short, longs))  # 需人工审查

    return auto_confirmed, ambiguous
```

示例:

```
输入: ["秦庄襄王", "庄襄王", "秦昭襄王"]
检测: "庄襄王" ⊂ "秦庄襄王" (唯一) → 自动确认
输出: [("秦庄襄王", "庄襄王")]
```

策略2:帝号特殊模式

规则:"帝X" → "X" (商朝帝号规律)

```
DI_PATTERN = [
    (r'^帝(.+)$', r'\1'),  # 帝尧 → 尧
]

# 示例:
# "帝尧" in chapter AND "尧" in chapter → alias
```

原理:商代称谓习惯("帝辛" = "纣","帝乙" = "乙")

策略3:姓氏分组(待人工)

规则:共享姓氏的人名组

```
def group_by_surname(person_names):
    """
   按姓氏分组:
    - ["赵本学", "赵耿"] → 赵姓组
    - ["石作蜀", "石奢"] → 石姓组
    """
    surname_groups = defaultdict(set)
    for name in person_names:
        surname = name[0]  # 文言文单字姓为主
        if len(name) >= 2:
            surname_groups[surname].add(name)

    # 标记为"不确定",需人工审查
    uncertain_groups = {
        sur: names for sur, names in surname_groups.items()
        if len(names) >= 2
    }
    return uncertain_groups
```

输出示例:

```
## 不确定别名组(需人工审查)

### 赵姓
- 赵本学, 赵耿
- **判断**: 不是别名,是两个不同的人

### 公孙姓
- 公孙侨, 公孙鞅
- **判断**: 不是别名,公孙是复姓,两人不同
```

冲突解决:双向索引防重

```
def merge_into_aliases(long_name, short_name, aliases, reverse):
    """
    维护双向映射:
    - aliases[canonical] = [alias1, alias2, ...]
    - reverse[any_name] = canonical
    """

    # 检查是否已有映射
    long_canonical = reverse.get(long_name)
    short_canonical = reverse.get(short_name)

    if long_canonical and short_canonical and long_canonical !=
short_canonical:
        # 冲突:两者已各自有不同的规范名
        print(f"冲突: {long_name} → {long_canonical}, "
              f"{short_name} → {short_canonical}")
        return False  # 跳过,不合并

    if long_canonical:
        # 长名已有规范名,将短名加入该组
        aliases[long_canonical].append(short_name)
        reverse[short_name] = long_canonical
    else:
        # 创建新规范名(以长名为准)
        aliases[long_name] = aliases.get(long_name, []) +
[short_name]
        reverse[long_name] = long_name
        reverse[short_name] = long_name

    return True
```

保证:任意名称只映射到唯一规范名(反向索引约束)

2.2 字符串归一化

异体字处理

```
VARIANT_MAP = {
    '钜': '巨',  # 钜鹿 → 巨鹿
    '於': '于',  # 於是 → 于是
    '寔': '实',  # 寔 → 实
}

def normalize_variants(text):
    """异体字转换为标准字"""
    for variant, standard in VARIANT_MAP.items():
        text = text.replace(variant, standard)
    return text
```

应用场景:
- 事件名称匹配("钜鹿之战" vs "巨鹿之战")
- 地名统一("於陵" vs "于陵")

标注符号清理

```
def clean_annotation_markers(text):
    """移除标注符号,保留纯文本"""
    # 移除〖TYPE ...〗
    text = re.sub(r'〖[@=;#%\'&^~•!+$?{:\[_][^〗]+〗', '', text)
    return text

# 示例:
# 输入: "〖@刘邦〗攻〖=关中〗"
# 输出: "刘邦攻关中"
```

用途:
- 事件名称相似度计算(移除标注噪音)
- 纯文本搜索

2.3 名称相似度计算

```
def name_similarity(name_a, name_b):
    """
    计算两个名称的字符重叠相似度

    算法: |shared_chars| / min(|a|, |b|)

    示例:
    - ("长平之战", "长平战役") → {长,平,战} / 4 = 0.75
    - ("巨鹿之战", "钜鹿之战") → 1.0 (归一化后)
    - ("长平之战", "邯郸之战") → {之,战} / 4 = 0.5
    """
    # 预处理:异体字归一化 + 去标注
    a_clean = normalize_variants(clean_annotation_markers(name_a))
    b_clean = normalize_variants(clean_annotation_markers(name_b))

    set_a = set(a_clean)
    set_b = set(b_clean)

    shared = set_a & set_b
    min_len = min(len(set_a), len(set_b))

    return len(shared) / min_len if min_len > 0 else 0.0
```

阈值选择(基于A/B测试):
- **≥0.8**:高置信度(几乎确定是同一事件)
- **≥0.6**:中等置信度(需额外证据,如共享人物/时间)
- **<0.6**:低相关性(可能无关)

三、实体层面融合（L2）

3.1 实体消歧

问题:短名歧义

```
文本: "〖@武王〗伐纣"
歧义: 武王 = 周武王? 秦武王? 楚武王?
```

工具: `disambiguate_names.py`

四层启发式策略:

层1:前置邦国标记

```
def resolve_with_preceding_state(text, short_name, pos):
    """
    模式: &state&@short_name@
    示例: &周&@武王@ → 周武王
    """
    # 向前扫描20字符,查找最近的&state&标记
    prefix = text[max(0, pos-20):pos]
    match = re.search(r'〖&([^〗]+)〗', prefix)
    if match:
        state = match.group(1)
        return f"{state}{short_name}"  # 周武王
    return None
```

层2:邻近邦国提及

```
def resolve_with_nearby_state(text, short_name, pos, window=80):
    """
    在±80字符窗口内查找&state&标记
```

```
    选择距离最近的邦国
    """
    left_window = text[max(0, pos-window):pos]
    right_window = text[pos:pos+window]

    # 提取所有邦国标记
    states_left = re.findall(r'〖&([^〗]+)〗', left_window)
    states_right = re.findall(r'〖&([^〗]+)〗', right_window)

    # 优先使用左侧(上文)最近的邦国
    if states_left:
        return f"{states_left[-1]}{short_name}"
    if states_right:
        return f"{states_right[0]}{short_name}"

    return None
```

层3:章节主题邦国

```
CHAPTER_STATE = {
    '004': '周',   # 周本纪
    '005': '秦',   # 秦本纪
    '006': '秦',   # 秦始皇本纪
    '035': '齐',   # 管晏列传
    '043': '赵',   # 赵世家
    # ... 130章映射
}

def resolve_with_chapter_state(chapter_id, short_name):
    """
    根据章节主题推断
    示例: 005章(秦本纪)中的"武王" → 秦武王
    """
    state = CHAPTER_STATE.get(chapter_id)
```

```
    if state:
        return f"{state}{short_name}"
    return None
```

层4:共现全名查找

```
def resolve_with_full_name_cooccurrence(chapter_entities,
short_name):
    """
    在章节中查找包含短名的全名
    示例: 章节有"周武王",短名"武王" → 周武王
    """
    candidates = [
        full_name for full_name in chapter_entities
        if short_name in full_name and len(full_name) >
len(short_name)
    ]

    if len(candidates) == 1:
        return candidates[0]  # 唯一候选,自动确认
    elif len(candidates) > 1:
        # 多候选 → 需人工决策
        return None  # 标记为歧义

    return None
```

冲突解决:多数投票 + 人工覆盖

```
def resolve_ambiguous_name(short_name, chapter_id,
chapter_entities):
    """
    综合四层策略的结果
    """
```

```
    candidates = []

    # 收集各层结果
    r1 = resolve_with_preceding_state(...)
    r2 = resolve_with_nearby_state(...)
    r3 = resolve_with_chapter_state(...)
    r4 = resolve_with_full_name_cooccurrence(...)

    candidates = [r for r in [r1, r2, r3, r4] if r]

    # 多数投票
    if not candidates:
        return None  # 无法消歧

    vote_counts = Counter(candidates)
    winner, count = vote_counts.most_common(1)[0]

    # 一致性检查
    if count == len(candidates):
        return winner  # 全部一致,高置信度

    if count >= len(candidates) * 0.67:  # 2/3多数
        return winner  # 多数同意,中等置信度

    # 分裂决策 → 人工审查
    return None  # 标记为需review

# 人工覆盖机制
MANUAL_CORRECTIONS = {
    ('004', '元王'): '周元王',  # 覆盖nearby_state=楚的误判
    ('005', '王'): '秦王',       # 特殊情况
}

if (chapter_id, short_name) in MANUAL_CORRECTIONS:
    return MANUAL_CORRECTIONS[(chapter_id, short_name)]
```

3.2 实体索引合并

工具: `build_entity_index.py`

目标:从表面形式(surface forms)到规范实体(canonical entities)

输入:
1. 标注文本(130篇,含所有 `〖TYPE name〗` 标记)
2. `entity_aliases.json` (别名映射)

输出: `entity_index.json` (规范化实体索引)

算法

```
def build_canonical_index(chapter_files, aliases, disambig_map):
    """
    构建规范实体索引

    步骤:
    1. 从所有标注文本中提取实体
    2. 消歧短名(使用disambig_map)
    3. 合并别名(使用aliases反向索引)
    4. 聚合出现位置和章节
    """
    # 反向索引: 任意名称 → 规范名
    reverse = {}
    for canonical, alias_list in aliases.items():
        reverse[canonical] = canonical
        for alias in alias_list:
            reverse[alias] = canonical

    entity_index = defaultdict(lambda: {
        'aliases': set(),
        'chapters': set(),
        'para_refs': [],
        'count': 0
    })
```

```
    for chapter_file in chapter_files:
        chapter_id = extract_chapter_id(chapter_file)
        text = read_file(chapter_file)

        # 提取所有实体标注
        entities = extract_tagged_entities(text)  # [(type, name,
pos), ...]

        for etype, name, pos in entities:
            # 1. 消歧(如果是短名)
            if is_ambiguous(name):
                resolved = disambig_map.get((chapter_id, name),
name)
            else:
                resolved = name

            # 2. 查找规范名
            canonical = reverse.get(resolved, resolved)

            # 3. 聚合信息
            entity_index[etype][canonical]
['aliases'].add(resolved)
            entity_index[etype][canonical]
['chapters'].add(chapter_id)
            entity_index[etype][canonical]
['para_refs'].append((chapter_id, pos))
            entity_index[etype][canonical]['count'] += 1

    return entity_index
```

输出示例

```
{
  "person": {
    "刘邦": {
      "aliases": ["沛公", "汉王", "高祖", "高帝", "刘季"],
      "chapters": ["008", "009", "010", "053", "095"],
      "para_refs": [
        ["008", "para-003"],
        ["008", "para-012"],
        ["009", "para-045"]
      ],
      "count": 156
    }
  },
  "place": {
    "关中": {
      "aliases": ["秦地", "关内"],
      "chapters": ["005", "006", "008", "009"],
      "para_refs": [...],
      "count": 89
    }
  }
}
```

3.3 内联消歧语法

问题:同一表面形式,不同章节不同含义

```
013章: "台" → "吕台"
047章: "台" → "柳下季(字台)"
```

解决方案:管道符号内联消歧

语法: `〖@surface|canonical〗`

```
013章: 〖@台|吕台〗
047章: 〖@台|柳下季〗
```

解析:

```
def parse_inline_disambig(tagged_name):
    """
    解析内联消歧语法

    输入: "台|吕台"
    输出: (display="台", canonical="吕台")
    """
    if '|' in tagged_name:
        parts = tagged_name.split('|')
        return parts[0], parts[1]  # (display, canonical)
    else:
        return tagged_name, tagged_name  # 无消歧,显示=规范
```

渲染:
- HTML显示:"台"(显示名)
- 链接指向: `/entities/person.html#吕台` (规范名)
- 悬停提示:"吕台"(完整名称)

四、事件层面融合（L3）

4.1 跨章事件对齐（cross_ref）

问题:同一事件的多章记述

```
005_秦本纪_事件042: "秦赵长平之战"
015_六国年表_事件108: "长平之役"
043_赵世家_事件067: "赵括代廉颇,秦将白起坑降卒"
```

目标:识别为同一事件,建立`cross_ref`关系

工具:`extract_event_relations.py`(cross_ref计算)

多因子匹配算法:

```
def compute_cross_ref(all_events):
    """
    跨章事件互见关系检测

    判据(需同时满足多个条件):
    1. 事件名称相似度 ≥ 阈值
    2. 共享人物 ≥ N个
    3. 年份接近(如有CE标注)
    """
    cross_refs = []

    for event_a, event_b in itertools.combinations(all_events, 2):
        # 跳过同章事件(章内关系由LLM处理)
        if event_a['chapter'] == event_b['chapter']:
            continue

        # 因子1:名称相似度
        name_sim = name_similarity(event_a['name'],
event_b['name'])
```

```
        # 因子2:共享人物
        shared_people = set(event_a['persons']) &
set(event_b['persons'])

        # 因子3:年份接近度
        year_diff = abs(event_a.get('ce_year', 0) -
event_b.get('ce_year', 0))

        # 综合判断
        if name_sim >= 0.8 and len(shared_people) >= 1:
            # 高相似度 + 至少1个共同人物 → 高置信度
            confidence = 0.95
        elif name_sim >= 0.6 and len(shared_people) >= 2 and
year_diff <= 5:
            # 中等相似度 + 2个共同人物 + 同时期 → 中等置信度
            confidence = 0.85
        else:
            continue  # 不满足条件,跳过

        cross_refs.append({
            'type': 'cross_ref',
            'source': event_a['id'],
            'target': event_b['id'],
            'confidence': confidence,
            'factors': {
                'name_similarity': name_sim,
                'shared_people': list(shared_people),
                'year_diff': year_diff
            }
        })

    return cross_refs
```

输出示例

```
{
  "id": "R_005-042→043-067",
  "type": "cross_ref",
  "source": "005-042",
  "target": "043-067",
  "source_name": "秦赵长平之战",
  "target_name": "长平之役",
  "confidence": 0.95,
  "factors": {
    "name_similarity": 0.88,
    "shared_people": ["白起", "赵括", "廉颇"],
    "year_diff": 0
  },
  "note": "同一事件在秦本纪和赵世家中的不同记载"
}
```

4.2 战争事件多源合并

工具: `extract_wars.py`

六阶段融合流水线:

阶段1:原始提取（Raw Extraction）

```
def stage_identify(event_files):
    """
    从130个事件索引文件中提取所有type="战争"的事件
    """
    raw_wars = []

    for fpath in event_files:
        events = parse_event_index(fpath)
```

```
        for event in events:
            if event['type'] == '战争':
                raw_wars.append({
                    'event_id': event['id'],
                    'chapter': event['chapter'],
                    'name': event['name'],
                    'persons': event['persons'],
                    'locations': event['locations'],
                    'year': event.get('ce_year'),
                    'description': event['description']
                })

    return raw_wars  # 736条
```

阶段2:自动分组（Auto-merge by Name）

```
def stage_merge_auto(raw_wars):
    """
    按战争名称分组,识别多源事件
    """
    # 清理名称(移除标注符号,异体字归一化)
    name_groups = defaultdict(list)
    for war in raw_wars:
        clean_name = normalize_war_name(war['name'])
        name_groups[clean_name].append(war)

    consolidated_wars = []

    for war_name, sources in name_groups.items():
        if len(sources) == 1:
            # 单源事件,直接保留
            consolidated_wars.append(sources[0])
        else:
            # 多源事件,合并
            merged_war = merge_war_sources(war_name, sources)
```

```
            consolidated_wars.append(merged_war)

    return consolidated_wars  # 308组(736→308合并)
```

合并逻辑:

```
def merge_war_sources(war_name, sources):
    """
    合并同一战争的多个来源

    策略:
    - 名称: 取最常见的
    - 人物: 合并所有来源的人物集合
    - 地点: 合并所有地点
    - 年份: 取众数(mode),记录分歧
    - 描述: 保留所有来源的描述(用于比对)
    - 伤亡数: 保留所有来源(标注出处)
    """
    merged = {
        'war_name': war_name,
        'sources': [s['event_id'] for s in sources],
        'chapters': list(set(s['chapter'] for s in sources)),
        'persons': list(set().union(*[s['persons'] for s in
sources])),
        'locations': list(set().union(*[s['locations'] for s in
sources])),
        'descriptions': [
            {'source': s['event_id'], 'chapter': s['chapter'],
'text': s['description']}
            for s in sources
        ]
    }

    # 年份:取众数,记录冲突
    years = [s['year'] for s in sources if s.get('year')]
    if years:
```

```
        year_counts = Counter(years)
        most_common_year, count = year_counts.most_common(1)[0]
        merged['year'] = most_common_year
        if len(year_counts) > 1:
            # 有冲突,记录
            merged['year_conflicts'] = dict(year_counts)

    return merged
```

阶段3:细节提取（Detail Extraction）

```
def stage_extract_details(consolidated_wars):
    """
    从事件描述中提取结构化字段
    """
    for war in consolidated_wars:
        # 提取交战双方
        war['belligerents'] =
extract_belligerents(war['descriptions'])

        # 提取伤亡数字
        war['casualties'] =
extract_casualties(war['descriptions'])

        # 提取战果
        war['outcome'] = extract_outcome(war['descriptions'])

    return wars

def extract_belligerents(descriptions):
    """
    从描述中提取交战方

    模式:
    1. 〖&state1&〗攻〖&state2&〗
```

```
    2. state1伐state2
    3. state1败于state2
    """
    patterns = [
        r'〖&([^〗]+)〗(?:攻|伐|击|围|灭|败)〖&([^〗]+)〗',  # 标注实体
        r'(秦|赵|齐|楚|燕|魏|韩|吴|越|晋|周|汉)(?:攻|伐|击)(\w+)',  #
纯文本
    ]

    attacker, defender = None, None
    for desc in descriptions:
        for pattern in patterns:
            match = re.search(pattern, desc['text'])
            if match:
                attacker, defender = match.groups()
                break
        if attacker:
            break

    return {
        'attacker': {'state': [attacker], 'commanders': []},
        'defender': {'state': [defender], 'commanders': []}
    }

def extract_casualties(descriptions):
    """
    提取伤亡数字

    模式:
    - X万人
    - 斩首X万
    - 坑降卒X万
    """
    casualties_list = []

    for desc in descriptions:
        # 正则匹配数字+单位
```

```
        matches = re.findall(
            r'((?:斩首|坑|杀|降)?.*?)([0-9一二三四五六七八九十百千万]+)
\s*(?:人|万人|众)',
            desc['text']
        )
        for context, number in matches:
            casualties_list.append({
                'source': desc['source'],
                'number': chinese_to_arabic(number),
                'context': context,
                'text': desc['text']
            })

    return casualties_list
```

阶段4:冲突检测与标注

```
def stage_detect_conflicts(wars):
    """
    检测多源数据中的矛盾

    检查项:
    1. 年份不一致
    2. 伤亡数差异过大(>2倍)
    3. 战果矛盾(一方说胜,另一方说败)
    """
    for war in wars:
        conflicts = []

        # 年份冲突
        if 'year_conflicts' in war:
            conflicts.append({
                'type': 'year_discrepancy',
                'values': war['year_conflicts'],
                'note': '不同来源记载的年份不一致'
```

```
            })

        # 伤亡数冲突
        casualties = war.get('casualties', [])
        if len(casualties) > 1:
            numbers = [c['number'] for c in casualties if
c['number']]
            if max(numbers) / min(numbers) > 2:
                conflicts.append({
                    'type': 'casualty_discrepancy',
                    'values': casualties,
                    'note': '不同来源的伤亡数差异超过2倍'
                })

        if conflicts:
            war['conflicts'] = conflicts

    return wars
```

阶段5:LLM增强（未来）

```
def stage_llm_enrich(wars):
    """
    使用LLM补充细节(计划中)

    任务:
    - 战术分析(奇袭/正面交锋/围城/...)
    - 关键转折点识别
    - 战争影响评估
    """
    # TODO: 实现LLM调用
    pass
```

阶段6:导出

```
def stage_export(wars):
    """
    导出为多种格式
    """
    # JSON(机读)
    with open('kg/events/data/wars.json', 'w') as f:
        json.dump(wars, f, ensure_ascii=False, indent=2)

    # Markdown索引(人读)
    generate_markdown_index(wars, 'kg/events/data/战争事件索引.md')

    # HTML可视化(交互)
    generate_html_visualization(wars, 'docs/wars.html')
```

最终输出数据结构

```
{
  "war_id": "WAR-005-042",
  "war_name": "长平之战",
  "sources": ["005-042", "015-108", "043-067"],
  "chapters": ["005_秦本纪", "015_六国年表", "043_赵世家"],
  "time": {
    "year": -260,
    "year_conflicts": {"-260": 2, "-259": 1}
  },
  "location": ["长平", "上党"],
  "belligerents": {
    "attacker": {
      "state": ["秦"],
      "commanders": ["白起", "王龁"],
      "ruler": ["昭襄王"]
    },
```

```
    "defender": {
      "state": ["赵"],
      "commanders": ["廉颇", "赵括"],
      "ruler": ["孝成王"]
    }
  },
  "casualties": [
    {
      "source": "005-042",
      "text": "斩首四十五万",
      "number": 450000,
      "side": "赵"
    },
    {
      "source": "043-067",
      "text": "坑降卒四十万",
      "number": 400000,
      "side": "赵"
    }
  ],
  "outcome": {
    "victor": "秦",
    "consequences": ["赵国元气大伤", "秦统一进程加速"]
  },
  "conflicts": [
    {
      "type": "casualty_discrepancy",
      "note": "伤亡数在40-45万之间,来源不一",
      "values": [450000, 400000]
    }
  ],
  "descriptions": [
    {
      "source": "005-042",
      "chapter": "005_秦本纪",
      "text": "秦使左庶长王龁攻韩,取上党。上党民走赵。赵军长平,以按据上党民。四月,龁因攻赵。赵使廉颇将。赵军士卒犹食平原君之食。秦数挑战,赵兵不出。赵王
```

信秦之间,召廉颇,使赵括代之。秦闻之,阴使武安君白起为上将军。赵括至,则出兵击秦军。秦军佯败而走,张二奇兵以劫之。赵军逐胜,追造秦壁。壁坚拒不得入,而秦奇兵二万五千人绝赵军后,又一军五千骑绝赵壁间,赵军分而为二,粮道绝。秦王闻赵食道绝,王自之河内,赐民爵各一级,发年十五以上悉诣长平,遮绝赵救及粮食。九月,赵卒不得食四十六日,皆内阴相杀食。急击秦军,秦军射杀之。赵括出锐卒自搏战,秦军射杀赵括。括军败,卒四十万人降武安君。武安君计曰:「前秦已拔上党,上党民不乐为秦而归赵。赵卒反覆。非尽杀之,恐为乱。」乃挟诈而尽阬杀之,遣其小者二百四十人归赵。前后斩首虏四十五万人。赵人大震。"
    },
    {
      "source": "043-067",
      "chapter": "043_赵世家",
      "text": "..."
    }
  ]
}

4.3 年表×事件融合

工具: `fix_by_chronology.py`

问题:
- 事件索引:有详细描述,但年份可能缺失或不准
- 编年表数据:有精确年份,但描述简略

目标:融合两者,互补优势

两轮推理策略

Round 1:直接证据匹配

```
def round1_direct_evidence(events, chronology, person_lifespans):
    """
    使用高置信度数据源直接标注年份
```

```
    数据源:
    1. 编年表特征事件(战争/变法/盟会/...)
    2. 人物卒年(for type=崩逝/卒的事件)
    3. 人物活跃年(中位数)
    """
    anchors = []  # 成功标注的事件作为锚点

    for event in events:
        # 策略1:编年表匹配
        chron_year = match_chronology_event(event, chronology)
        if chron_year:
            event['ce_year'] = chron_year
            event['year_source'] = 'chronology'
            event['certainty'] = 'high'
            anchors.append(event)
            continue

        # 策略2:人物卒年
        if event['type'] in ['崩逝', '卒']:
            person = event['persons'][0]  # 主角
            death_year = person_lifespans.get(person,
{}).get('death')
            if death_year:
                event['ce_year'] = death_year
                event['year_source'] = 'person_lifespan'
                event['certainty'] = 'high'
                anchors.append(event)
                continue

        # 策略3:人物活跃年中位数
        if event['persons']:
            active_years = []
            for person in event['persons']:
                years = chronology_person_years(person,
chronology)
                active_years.extend(years)
            if active_years:
```

```
                median_year = int(np.median(active_years))
                event['ce_year'] = median_year
                event['year_source'] = 'person_activity'
                event['certainty'] = 'medium'
                anchors.append(event)

    return anchors
```

Round 2:插值推断

```
def round2_interpolation(events, anchors):
    """
   利用Round 1的锚点,对未标注事件进行插值

   算法:
    1. 按章节和时间字段排序事件
    2. 找到前后最近的锚点
    3. 线性插值计算年份
    """
    for event in events:
        if event.get('ce_year'):
            continue  # 已有年份,跳过

        # 查找前后锚点
        prev_anchor = find_previous_anchor(event, anchors)
        next_anchor = find_next_anchor(event, anchors)

        if prev_anchor and next_anchor:
            # 双锚点插值
            year = interpolate_between_anchors(event, prev_anchor,
next_anchor)
            event['ce_year'] = year
            event['year_source'] = 'interpolation'
            event['certainty'] = 'low'
```

```
        elif prev_anchor or next_anchor:
            # 单锚点外推(风险较高)
            anchor = prev_anchor or next_anchor
            year = extrapolate_from_anchor(event, anchor)
            event['ce_year'] = year
            event['year_source'] = 'extrapolation'
            event['certainty'] = 'very_low'

    return events

def interpolate_between_anchors(event, prev, next):
    """
    线性插值

    示例:
    - 前锚点: "即位" (year=-246, seq=1)
    - 当前事件: "三年" (seq=3)
    - 后锚点: "统一" (year=-221, seq=10)

    计算: -246 + (3-1) / (10-1) * (-221 - (-246)) = -246 + 2/9 * 25
≈ -240
    """
    prev_year = prev['ce_year']
    next_year = next['ce_year']
    prev_seq = prev['sequence_in_chapter']
    event_seq = event['sequence_in_chapter']
    next_seq = next['sequence_in_chapter']

    # 线性插值
    ratio = (event_seq - prev_seq) / (next_seq - prev_seq)
    interpolated_year = prev_year + ratio * (next_year -
prev_year)

    return int(round(interpolated_year))
```

冲突检测与修正

```
def validate_inferred_years(events, person_lifespans,
chapter_eras):
    """
    验证推断年份的合理性

    检查项:
    1. 人物在世约束:事件年份应在所有参与人物的生卒年之间
    2. 章节纪元约束:年份应在章节覆盖的时代范围内
    3. 单锚点塌缩检测:如果>50%事件年份相同,可能是锚点污染
    """
    errors = []

    for event in events:
        year = event.get('ce_year')
        if not year:
            continue

        # 检查1:人物生卒年
        for person in event['persons']:
            lifespan = person_lifespans.get(person, {})
            birth = lifespan.get('birth')
            death = lifespan.get('death')
            if birth and year < birth:
                errors.append({
                    'event': event['id'],
                    'type': 'before_birth',
                    'person': person,
                    'year': year,
                    'birth_year': birth
                })
            if death and year > death:
                errors.append({
                    'event': event['id'],
                    'type': 'after_death',
                    'person': person,
```

```
                    'year': year,
                    'death_year': death
                })

        # 检查2:章节纪元
        chapter = event['chapter']
        era_start, era_end = chapter_eras.get(chapter, (-3000, 0))
        if not (era_start <= year <= era_end):
            errors.append({
                'event': event['id'],
                'type': 'out_of_era',
                'year': year,
                'era_range': (era_start, era_end)
            })

    # 检查3:塌缩检测
    chapter_years = defaultdict(list)
    for event in events:

chapter_years[event['chapter']].append(event.get('ce_year'))

    for chapter, years in chapter_years.items():
        year_counts = Counter(years)
        most_common_year, count = year_counts.most_common(1)[0]
        if count / len(years) > 0.5:  # >50%相同
            errors.append({
                'event': None,
                'type': 'single_anchor_collapse',
                'chapter': chapter,
                'collapsed_year': most_common_year,
                'affected_events': count
            })

    return errors
```

五、SKU层面融合（L4）

5.1 结构化知识单元增强

工具: `augment_sku_entities.py`

场景:对外提供的SKU(Structured Knowledge Unit)需要关联实体信息

输入:
- SKU内容(如"太史公曰"段落、成语条目)
- 实体索引(`entity_index.json`)
- 别名映射(`entity_aliases.json`)

输出:SKU + 实体标签

算法

```
def augment_sku_with_entities(sku_content, entity_index,
entity_aliases):
    """
    为SKU内容标注实体

    步骤:
    1. 构建实体查找表(包含所有别名)
    2. 在SKU内容中匹配实体(贪婪最长匹配)
    3. 去重(避免重叠)
    4. 聚合统计
    """
    # 1. 构建查找表
    entity_lookup = build_entity_lookup(entity_index,
entity_aliases)
    # 结构: {name: [(type, canonical), ...], ...}

    # 2. 匹配实体
    matched_entities = []
    matched_positions = set()  # 避免重叠
```

```
    # 按名称长度排序(贪婪最长匹配)
    sorted_names = sorted(entity_lookup.keys(), key=len,
reverse=True)

    for name in sorted_names:
        for match in re.finditer(re.escape(name), sku_content):
            start, end = match.span()

            # 检查是否与已匹配位置重叠
            if any(start <= pos < end or start < pos <= end
                   for pos in range(matched_positions)):
                continue  # 重叠,跳过

            # 记录匹配
            for etype, canonical in entity_lookup[name]:
                matched_entities.append({
                    'type': etype,
                    'surface': name,
                    'canonical': canonical,
                    'position': (start, end)
                })
                matched_positions.update(range(start, end))

    # 3. 聚合统计
    entity_counts = defaultdict(int)
    for entity in matched_entities:
        entity_counts[(entity['type'], entity['canonical'])] += 1

    # 4. 生成SKU实体标签
    sku_entities = {
        'person': [],
        'place': [],
        'official': [],
        # ... 18 types
    }

    for (etype, canonical), count in entity_counts.items():
```

```
            sku_entities[etype].append({
                'name': canonical,
                'count': count
            })

        # 排序:按出现次数降序
        for etype in sku_entities:
            sku_entities[etype].sort(key=lambda x: x['count'],
reverse=True)

        return sku_entities
```

输出示例

输入SKU:

```
{
  "sku_id": "sku_taishigong_001",
  "title": "太史公曰:五帝",
  "content": "学者多称五帝,尚矣。然尚书独载尧以来,而百家言黄帝,其文不雅驯,荐绅先生难言之。孔子所传宰予问五帝德及帝系姓,儒者或不传。余尝西至空桐,北过涿鹿,东渐于海,南浮江淮矣,至长老皆各往往称黄帝、尧、舜之处,风教固殊焉,总之不离古文者近是。..."
}
```

增强后的SKU:

```
{
  "sku_id": "sku_taishigong_001",
  "title": "太史公曰:五帝",
  "content": "...(同上)",
  "entities": {
    "person": [
      {"name": "黄帝", "count": 5},
      {"name": "尧", "count": 3},
```

```
      {"name": "舜", "count": 2},
      {"name": "孔子", "count": 1},
      {"name": "宰予", "count": 1}
    ],
    "place": [
      {"name": "空桐", "count": 1},
      {"name": "涿鹿", "count": 1},
      {"name": "江淮", "count": 1}
    ],
    "book": [
      {"name": "尚书", "count": 1}
    ]
  },
  "top_entities": ["黄帝", "尧", "舜"],
  "entity_count": 14
}
```

5.2 事实发现与事件融合

场景:战争事件需要融合事件索引和事实发现

事件索引:

```
{
  "event_id": "005-042",
  "name": "长平之战",
  "type": "战争",
  "persons": ["白起", "赵括"],
  "locations": ["长平"],
  "year": -260
}
```

事实发现(SKILL_05d,待构建):

```
{
  "fact_id": "FACT-005-042-01",
  "event_id": "005-042",
  "fact_type": "casualty",
  "content": "坑降卒四十万",
  "structured": {
    "side": "赵",
    "number": 400000,
    "type": "killed"
  }
}
```

融合目标:战争事件 = 事件索引 + 相关事实

```
{
  "war_id": "WAR-005-042",
  "war_name": "长平之战",
  "event_sources": ["005-042", "015-108", "043-067"],
  "facts": [
    {"fact_id": "FACT-005-042-01", "type": "casualty", ...},
    {"fact_id": "FACT-005-042-02", "type": "tactics", ...}
  ],
  "casualties": 400000,  # 从facts聚合
  "tactics": ["坑杀", "断粮道"]  # 从facts提取
}
```

六、冲突解决策略

6.1 冲突类型分类

类型	示例	来源
年份冲突	事件A在章节X标注为-260,在章节Y标注为-259	跨章记述差异

类型	示例	来源
伤亡数冲突	同一战争,来源A说40万,来源B说45万	史料记载不一
身份冲突	"武王"在不同章节指不同人	短名歧义
关系冲突	两个事件,来源A说有因果,来源B说无关	推理差异
时序冲突	事件A的year < 事件B的year,但描述说B在A之前	标注错误

6.2 解决策略矩阵

冲突类型	策略	实施
年份冲突(小)	取众数(majority voting)	`Counter(years).most_common(1)[0]`
年份冲突(大)	保留所有,标注 `year_conflicts`	记录在JSON,人工review
伤亡数冲突	保留所有,标注来源	`casualties: [{source, number}, ...]`
身份冲突	上下文消歧	四层启发式策略
关系冲突	置信度加权	保留高置信度关系
时序冲突	约束检查	lint检测,人工修正

6.3 冲突记录模式

原则:不强制解决,记录所有观点

```
{
  "war_id": "WAR-005-042",
  "war_name": "长平之战",
  "year": -260,  // 多数意见
  "conflicts": [
    {
      "type": "year_discrepancy",
      "values": {
        "-260": ["005-042", "043-067"],  // 2票
        "-259": ["015-108"]               // 1票
      },
      "resolution": "majority_vote",
      "note": "六国年表记载为-259,本纪与世家均为-260,采用多数"
    },
    {
      "type": "casualty_discrepancy",
      "values": [
        {"source": "005-042", "number": 450000, "text": "斩首四十五
万"},
        {"source": "043-067", "number": 400000, "text": "坑降卒四十
万"}
      ],
      "resolution": "keep_all",
      "note": "两个数字均保留,可能指不同统计口径(斩首 vs 坑杀)"
    }
  ]
}
```

6.4 人工覆盖机制

场景:自动算法误判,需人工修正

```
# 人工覆盖配置文件
MANUAL_CORRECTIONS = {
    # 格式: (章节, 短名) → 规范名
    ('004', '元王'): '周元王',  # 覆盖nearby_state=楚的误判
    ('005', '王'): '秦王',       # 特殊情况,"王"单独出现指秦王

    # 事件年份修正
    ('005', '042'): {
        'ce_year': -260,  # 覆盖自动推断的-259
        'reason': '跨章交叉验证,本纪与世家一致为-260'
    },

    # 别名关系修正
    ('person', '庄子', '庄周'): 'merge',  # 确认是同一人
    ('person', '李斯', '李子'): 'split',  # 确认不是同一人
}

def apply_manual_corrections(data, corrections):
    """应用人工修正"""
    for key, value in corrections.items():
        if key in data:
            old_value = data[key]
            data[key] = value
            log_correction(key, old_value, value, '人工修正')
    return data
```

七、融合质量保障

7.1 一致性校验

```
def validate_fusion_consistency(fused_data):
    """
    融合后一致性检查
```

```
    检查项:
    1. 反向索引一致性:reverse[name] 必须存在于 aliases[canonical]
    2. 年份时序一致性:sequel关系的source_year < target_year
    3. 实体引用完整性:事件中的person必须存在于entity_index
    4. 跨工序一致性:事件年份与人物生卒年不冲突
    """
    errors = []

    # 检查1:反向索引
    for name, canonical in reverse_index.items():
        if name not in entity_aliases.get(canonical, [canonical]):
            errors.append({
                'type': 'reverse_index_inconsistency',
                'name': name,
                'canonical': canonical
            })

    # 检查2:时序
    for relation in event_relations:
        if relation['type'] == 'sequel':
            source_year = events[relation['source']]['ce_year']
            target_year = events[relation['target']]['ce_year']
            if source_year and target_year and source_year >=
target_year:
                errors.append({
                    'type': 'temporal_inconsistency',
                    'relation': relation['id'],
                    'source_year': source_year,
                    'target_year': target_year
                })

    # 检查3:实体引用
    for event in events:
        for person in event['persons']:
            if person not in entity_index['person']:
                errors.append({
                    'type': 'dangling_entity_reference',
```

```
                    'event': event['id'],
                    'person': person
                })

    return errors
```

7.2 覆盖率统计

```
def compute_fusion_coverage(events, entity_index, chronology):
    """
    统计融合覆盖率

    指标:
    - 事件年份覆盖率:有CE标注的事件占比
    - 实体消歧覆盖率:歧义短名成功消歧的占比
    - 跨章对齐覆盖率:有cross_ref关系的事件占比
    """
    stats = {
        'total_events': len(events),
        'events_with_year': len([e for e in events if
e.get('ce_year')]),
        'year_coverage': 0.0,

        'total_ambiguous_names': len(ambiguous_names),
        'disambiguated_names': len([n for n in ambiguous_names if
n in disambig_map]),
        'disambig_coverage': 0.0,

        'total_multi_source_events': len([e for e in events if
len(e.get('sources', [])) > 1]),
        'cross_chapter_coverage': 0.0,
    }

    stats['year_coverage'] = stats['events_with_year'] /
stats['total_events']
```

```
    stats['disambig_coverage'] = stats['disambiguated_names'] /
stats['total_ambiguous_names']

    return stats
```

输出示例:

```
## 融合质量报告

| 指标 | 数值 | 目标 | 状态 |
|------|------|------|------|
| 事件年份覆盖率 | 98.7% (3,051/3,185) | >95% | ✅ |
| 实体消歧覆盖率 | 92.4% (595/644) | >90% | ✅ |
| 跨章对齐事件 | 294个 | - | ℹ️ |
| 冲突记录数 | 73个 | <100 | ✅ |
```

7.3 增量更新检测

```
def detect_fusion_drift(old_data, new_data):
    """
    检测融合结果的变化(增量更新)

    用途:
    - 监控数据演化趋势
    - 发现异常变动(如大批量别名消失)
    - 审查更新合理性
    """
    changes = {
        'added': [],
        'removed': [],
        'modified': []
    }

    # 比对实体索引
```

```
    old_entities = set(old_data['entity_index']['person'].keys())
    new_entities = set(new_data['entity_index']['person'].keys())

    changes['added'] = list(new_entities - old_entities)
    changes['removed'] = list(old_entities - new_entities)

    # 比对别名映射
    for canonical in old_entities & new_entities:
        old_aliases =
set(old_data['entity_aliases'].get(canonical, []))
        new_aliases =
set(new_data['entity_aliases'].get(canonical, []))
        if old_aliases != new_aliases:
            changes['modified'].append({
                'canonical': canonical,
                'old_aliases': list(old_aliases),
                'new_aliases': list(new_aliases)
            })

    return changes
```

八、典型案例

案例1:刘邦的多称谓融合

原始标注:

```
008章: 〖@沛公〗起兵
008章: 〖@汉王〗入关中
008章: 〖@高祖〗立为天子
009章: 〖@刘邦〗者,沛丰邑人也
053章: 〖@高帝〗崩
```

别名检测:

```
# auto_detect_aliases.py 检测到:
# - "刘邦" contains "邦" → 不确定(单字)
# - "高祖"与"高帝"共现于008章 → 可能是别名
# - "沛公"、"汉王"、"高祖"同章出现 → 可能指同一人
```

人工确认:

```
{
  "刘邦": ["沛公", "汉王", "高祖", "高帝", "刘季"]
}
```

消歧规则:

```
# 上下文无邦国前缀时,默认指刘邦
CHAPTER_MAIN_PERSON = {
    '008': '刘邦',  # 高祖本纪
    '009': '刘邦',  # 吕太后本纪
}
```

融合结果:

```
{
  "person": {
    "刘邦": {
      "aliases": ["沛公", "汉王", "高祖", "高帝", "刘季"],
      "chapters": ["008", "009", "010", "053", "095"],
      "count": 467,  // 合并后出现次数
      "earliest_mention": "008-para-003",
      "roles": ["起义领袖", "汉王", "天子"]
    }
  }
}
```

案例2:长平之战的跨章合并

三个来源:

005_秦本纪:

```
事件ID: 005-042
年份: -260
描述: 秦使左庶长王龁攻韩,取上党。赵使廉颇将。秦闻之,阴使武安君白起为上将军。赵军逐胜,追造秦壁。壁坚拒不得入...赵括出锐卒自搏战,秦军射杀赵括。括军败,卒四十万人降武安君。武安君计曰:「前秦已拔上党,上党民不乐为秦而归赵。赵卒反覆。非尽杀之,恐为乱。」乃挟诈而尽阬杀之...前后斩首虏四十五万人。
```

015_六国年表:

```
事件ID: 015-108
年份: -259  // ⚠️ 与本纪冲突
描述: 赵括代廉颇,秦将白起大破之
```

043_赵世家:

```
事件ID: 043-067
年份: -260
描述: ...赵括出锐卒自搏战...秦射杀括...括军败...卒四十万人降...武安君...尽阬之...
```

融合流程:

步骤1:名称匹配

```
name_similarity("长平之战", "长平之役") = 0.88  # 高相似度
shared_people = {"白起", "赵括", "廉颇"}  # 3个共同人物
→ 判定为同一事件
```

步骤2:年份冲突解决

```
year_votes = {
    -260: ["005-042", "043-067"],  # 2票
    -259: ["015-108"]               # 1票
}
→ 采用多数:-260
→ 记录冲突
```

步骤3:细节提取

```
belligerents = {
    'attacker': {'state': ['秦'], 'commanders': ['白起', '王龁']},
    'defender': {'state': ['赵'], 'commanders': ['廉颇', '赵括']}
}

casualties = [
    {'source': '005-042', 'number': 450000, 'text': '斩首四十五万'},
    {'source': '043-067', 'number': 400000, 'text': '坑降卒四十万'}
]
→ 保留两个数字,可能统计口径不同
```

步骤4:输出合并事件

```
{
  "war_id": "WAR-长平之战",
  "war_name": "长平之战",
  "sources": ["005-042", "015-108", "043-067"],
  "chapters": ["005_秦本纪", "015_六国年表", "043_赵世家"],
  "year": -260,
  "year_conflicts": {"-260": 2, "-259": 1},
  "belligerents": {...},
  "casualties": [...],
  "conflicts": [
    {
      "type": "year_discrepancy",
```

```
      "note": "六国年表记为-259,本纪与世家记为-260,采用多数"
    },
    {
      "type": "casualty_discrepancy",
      "note": "斩首45万 vs 坑杀40万,可能统计口径不同"
    }
  ]
}
```

案例3:"武王"的上下文消歧

文本1(004_周本纪):

```
〖&周&〗〖@武王〗伐纣
```

消歧:前置邦国标记 → 周武王

文本2(005_秦本纪):

```
昭襄王卒,子〖@武王〗立
```

消歧:章节主题(005=秦) → 秦武王

文本3(043_赵世家):

```
〖@武王〗即位,封其弟〖@平原君〗
```

消歧:平原君是赵国人 → 赵武王

融合结果:

```
{
  "disambiguation_map": {
    "004": {"武王": "周武王"},
    "005": {"武王": "秦武王"},
    "043": {"武王": "赵武王"}
  }
}
```

九、工具清单

工具	功能	输入	输出
`auto_detect_aliases.py`	别名自动检测	标注文本	候选别名对(auto/uncertain)
`disambiguate_names.py`	人名消歧	标注文本 + 别名	`disambiguation_map.json`
`build_entity_index.py`	实体索引合并	标注文本 + 别名 + 消歧	`entity_index.json`
`extract_event_relations.py`	跨章事件对齐	事件索引	`event_relations.json` (cross_ref)
`extract_wars.py`	战争事件合并	事件索引	`wars.json`

工具	功能	输入	输出
`fix_by_chronology.py`	年表×事件融合	事件索引 + 编年表	带CE年份的事件索引
`augment_sku_entities.py`	SKU实体增强	SKU内容 + 实体索引	SKU + entities字段
`validate_fusion.py`	融合一致性检查	所有融合结果	错误报告

十、总结

核心要点

1. **分层融合**:字符串 → 实体 → 事件 → SKU → 知识图谱
2. **多源验证**:跨章节、跨数据源交叉验证
3. **冲突记录**:不强制解决,保留所有观点
4. **人工覆盖**:关键决策保留人工修正机制
5. **质量保障**:一致性校验、覆盖率统计、增量监控

融合哲学

传统数据库融合:
- 强制一致性(必须解决所有冲突)
- 单一真值(one source of truth)
- 覆盖式更新(新数据覆盖旧数据)

本项目的融合理念:
- **允许不一致**:冲突记录在`conflicts`字段
- **多元真值**:保留所有来源的观点
- **累积式更新**:新数据与旧数据共存(带时间戳)
- **可追溯性**:每条数据标注来源(`_source`字段)

核心洞察:

> **历史研究不是寻找唯一答案,而是理解多种可能性**

融合的目的不是消除差异,而是**结构化地呈现差异**,让用户基于完整信息做出判断。

与其他元技能的关系

- ← **柳叶刀方法**:分层融合对应分层分解
- ← **数据体感**:融合后需观察一致性和异常
- ← **质量控制**:融合结果需lint验证
- ← **可读性**:冲突记录需人类可读格式
- → **知识图谱**:融合是图谱构建的基础

数据融合是知识工程的"最后一公里":只有融合完成,才能形成统一、可用的知识视图。

元技能13: 技能迁移 — SKILL跨项目复用方法论

一、核心思想

> **"SKILL文档是可迁移的知识资产"**

技能迁移（Skill Transfer）是将在一个项目/数据集中积累的SKILL文档、工具、方法论迁移到新的领域/数据集的过程。

应用场景

现状:完成了《史记》130篇的知识抽取

未来目标:
- 《汉书》(120篇) — 同为纪传体,结构相似
- 《资治通鉴》(294卷) — 编年体,结构不同
- 《二十四史》— 规模扩大10倍+

挑战:
- 相似但不完全相同(如《汉书》多"表"少"世家")
- 文体差异(纪传体 vs 编年体)
- 规模差异(130篇 vs 294卷 vs 3000+卷)

二、迁移层次

2.1 直接复用（Level 1）

适用场景:目标数据与源数据高度相似

可直接复用的内容:
- **标注体系**(18类Token): `〖@人名〗` 、 `〖=地名〗` 等

- **Lint工具**: `lint_text_integrity.py` 、 `lint_markdown.py`
- **通用SKILL**:00-META系列(反思、质量控制、可读性等)
- **基础设施**:紫色编号、双格式输出、Git工作流

示例:《史记》→《汉书》

```
# 1. 复制SKILL文档
cp -r skills/ ../hanshu-kb/skills/

# 2. 复制Lint工具
cp -r scripts/lint_*.py ../hanshu-kb/scripts/

# 3. 复制标注规范
cp doc/spec/标注格式规范.md ../hanshu-kb/doc/spec/

# 4. 开始标注试点(冷启动)
# → 预期: 80%规则直接可用,20%需调整
```

2.2 适配调整（Level 2）

适用场景:有部分差异,需调整参数/规则

需要调整的内容:

差异1:结构变化

```
《史记》结构:
- 12本纪、10表、8书、30世家、70列传

《汉书》结构:
- 12纪、8表、10志、70传

《资治通鉴》结构:
- 16朝纪、294卷(无表/书/传区分)
```

调整:

```
# 需修改章节类型映射
CHAPTER_TYPES_SHIJI = {
    '001-012': 'benj本纪',
    '013-022': 'biao表',
    '023-030': 'shu书',
    '031-060': 'shijia世家',
    '061-130': 'liezhuan列传'
}

CHAPTER_TYPES_HANSHU = {
    '001-012': 'ji纪',
    '013-020': 'biao表',
    '021-030': 'zhi志',
    '031-100': 'zhuan传'
}
```

差异2:实体类型分布变化

```
《史记》特点:
- 战国至汉初,邦国众多('邦国标记密集)
- 神话传说多(黄帝/尧舜,神话标记必要)

《汉书》特点:
- 西汉一朝,邦国减少(主要是汉朝)
- 官职体系更复杂(需扩展官职词表)

《资治通鉴》特点:
- 跨16朝,年号复杂(需专门的年号实体类型)
- 编年体,时间标注更密集
```

调整策略:

```
# 针对《汉书》:扩展官职词表
官职词表_史记 = 230个
官职词表_汉书 = 230个(史记) + 120个(汉代新增)

# 针对《资治通鉴》:新增年号类型
标注符号_史记 = 18类
标注符号_通鉴 = 18类 + 〖*年号〗(新增)
```

差异3:事件类型分布

```
《史记》事件分布:
- 战争事件: 23% (战国纷争)
- 继位事件: 18% (朝代更迭频繁)

《汉书》事件分布(预测):
- 战争事件: 15% (一朝之内,战争减少)
- 政治改革: 25% (汉朝制度建设)
```

调整:

```
# 事件类型权重调整(影响LLM prompt)
EVENT_TYPE_WEIGHTS_SHIJI = {
    '战争': 0.23,
    '继位': 0.18,
    '政治': 0.15,
    # ...
}

EVENT_TYPE_WEIGHTS_HANSHU = {
    '战争': 0.15,  # 降低
    '政治改革': 0.25,  # 提升
```

```
    '继位': 0.10,  # 降低
    # ...
}
```

2.3 重新设计（Level 3）

适用场景:结构根本不同,需重新设计

需要重新设计的内容:

差异1:编年体 vs 纪传体

```
《史记》(纪传体):
- 以人物/主题为线索
- 事件分散在多篇中
- 需要跨章对齐(cross_ref关系)

《资治通鉴》(编年体):
- 以年份为线索
- 同一年的事件集中记述
- 不需要跨章对齐(已按年排列)
```

重新设计:

```
# 《史记》工序:
# SKILL_04_事件构建 → SKILL_05a_事件关系发现(跨章对齐)

# 《资治通鉴》工序:
# SKILL_04_事件构建 → SKILL_04t_编年排序(时间对齐,无需跨章)
#   ↓
# 不需要cross_ref关系(已经按年排列)
# 但需要"同年多事件"的并发关系建模
```

差异2:多源互见 vs 单一叙述

```
《史记》:
- 同一事件多章记述(本纪/世家/列传)
- 需要多源融合

《汉书》:
- 类似史记,也有多源记述

《资治通鉴》:
- 司马光已做融合(综合多史书)
- 单一叙述线,无需再融合
```

设计调整:

```
# 《史记》/《汉书》:
# 需要SKILL_05e_多源事件融合

# 《资治通鉴》:
# 不需要多源融合
# 但需要SKILL_05t_史源追溯(记录通鉴引用了哪些史书)
```

三、迁移方法论

3.1 试点驱动的迁移（Pilot-driven Transfer）

核心:不要一次性迁移全部,用小规模试点发现问题

阶段1:选择试点章节（3-5篇）

选择原则:
- **覆盖多样性**:选择不同类型的章节
- 《汉书》试点:1篇纪、1篇表、1篇志、2篇传
- **难度分层**:简单→中等→复杂

- 简单:人物少、事件少
- 复杂:人物多、跨章引用多

《汉书》试点候选:

```
1. 《高帝纪上》— 与《史记·高祖本纪》高度重叠,便于对比
2. 《百官公卿表》— 表类文献,测试结构化提取
3. 《礼乐志》— 志类文献,测试新文体
4. 《韩信传》— 与史记对比,测试差异
5. 《儒林传》— 测试人物关系网络
```

阶段2:直接应用现有SKILL

```
# 1. 复制SKILL文档到新项目
cp -r ../shiji-kb/skills/ skills/

# 2. 复制标注规范
cp ../shiji-kb/doc/spec/标注格式规范.md doc/spec/

# 3. 对试点章节进行标注(使用现有18类Token)
# 试点1: 高帝纪上.txt → 高帝纪上.tagged.md
# 使用现有SKILL_03a_实体标注.md作为指导

# 4. 运行现有Lint工具
python scripts/lint_text_integrity.py chapter_md/高帝纪上.tagged.md
python scripts/lint_markdown.py chapter_md/高帝纪上.tagged.md
```

阶段3:记录失败案例（Error Log）

关键:记录所有不适用的地方

创建文档: `doc/transfer/史记→汉书_迁移日志.md`

```
# 《史记》SKILL迁移到《汉书》错误日志

## 试点章节:高帝纪上

### 错误1:官职标注不完整
**现象**: 《汉书》出现"大司马"、"大司徒"等三公官职,但《史记》官职词表中没有
**原因**: 汉武帝改制后新设三公九卿体系
**影响章节**: 高帝纪上、百官公卿表
**修正方案**: 扩展官职词表,新增30个汉代官职
**状态**: 待修正

### 错误2:"表"的格式无法解析
**现象**: `extract_events.py`无法处理《百官公卿表》的表格结构
**原因**: 《史记》的"表"主要是年表(时间线),《汉书》的"表"是职官表(矩阵)
**影响章节**: 百官公卿表、古今人表
**修正方案**: 新增`extract_table_structured.py`处理二维表格
**状态**: 待开发

### 错误3:年号标注缺失
**现象**: "建元元年"、"元狩五年"等年号未标注
**原因**: 《史记》时代年号制度未完全确立,《汉书》大量使用年号纪年
**影响章节**: 所有纪、传
**修正方案**:
  - 新增`〖*年号〗`标注类型
  - 更新`SKILL_02_结构分析.md`,增加年号章节
  - 开发`parse_reign_periods.py`解析年号
**状态**: 待讨论(是否新增类型 vs 复用`〖%时间〗`)

### 错误4:交叉引用模式变化
**现象**: 《汉书》引用《史记》(如"见《史记》"),这种跨书引用无法用cross_ref关系表示
**原因**: 《史记》的cross_ref是书内跨章,《汉书》出现跨书引用
**影响章节**: 多处(约20+引用)
**修正方案**: 新增关系类型`external_ref`
```

```
**状态**: 待开发

... (继续记录20-30个问题)
```

阶段4:统计错误分布

```
# scripts/analyze_transfer_errors.py

def analyze_error_log(error_log_file):
    """
    分析迁移错误日志,统计问题分布
    """
    errors = parse_markdown_errors(error_log_file)

    # 按类型分组
    by_type = defaultdict(list)
    for err in errors:
        by_type[err['category']].append(err)

    # 统计
    print("## 错误类型分布\n")
    for category, errs in sorted(by_type.items(), key=lambda x:
len(x[1]), reverse=True):
        print(f"- {category}: {len(errs)}个问题")

    # 影响范围
    affected_chapters = set()
    for err in errors:
        affected_chapters.update(err['affected_chapters'])
    print(f"\n影响章节数: {len(affected_chapters)}/5 (试点)")

    # 修正工作量估算
    total_effort = sum(err.get('estimated_hours', 1) for err in
errors)
    print(f"\n预估修正工作量: {total_effort}小时")
```

```
    return by_type

# 输出示例:
# ## 错误类型分布
# - 词表不完整: 8个问题 (40%)
# - 新文体结构: 5个问题 (25%)
# - 标注类型缺失: 4个问题 (20%)
# - 关系类型扩展: 3个问题 (15%)
#
# 影响章节数: 5/5 (100%)
# 预估修正工作量: 45小时
```

3.2 反思驱动的适配（Reflection-driven Adaptation）

流程: 错误日志 → Agent反思 → SKILL修正

步骤1:生成反思提示词

```
# scripts/generate_transfer_reflection_prompt.py

def generate_prompt(error_log, source_skill, target_corpus):
    """
    根据错误日志生成反思提示词
    """
    prompt = f"""
# 任务:SKILL迁移反思

## 背景
我们正在将《史记》知识抽取的SKILL文档迁移到《汉书》项目。
在试点章节(5篇)中发现了{len(error_log)}个问题。

## 原始SKILL
{source_skill}  # 读取SKILL_03a_实体标注.md等
```

```
## 目标语料特点
{target_corpus_description}  # 《汉书》的结构/时代/文体差异

## 错误日志
{error_log}  # 完整错误列表

## 反思任务
1. **根因分析**:为什么这些SKILL规则在《史记》有效但在《汉书》失效?
2. **共性模式**:是否有系统性问题(如某类实体全部遗漏)?
3. **修正方案**:
   - 哪些规则可以泛化(参数调整)?
   - 哪些规则需要新增(新场景)?
   - 哪些规则需要废弃(不适用)?
4. **优先级**:按影响范围排序修正任务

## 输出格式
请以Markdown格式输出:
- § 根因分析
- § 共性模式(3-5条)
- § 修正建议(优先级排序)
- § 新增SKILL内容(草稿)
"""
    return prompt
```

步骤2:Agent反思

```
# 调用LLM API(Claude/GPT)
python scripts/run_transfer_reflection.py \
    --error-log doc/transfer/史记→汉书_迁移日志.md \
    --skill skills/SKILL_03a_实体标注.md \
    --output doc/transfer/反思报告_实体标注.md
```

反思输出示例(`doc/transfer/反思报告_实体标注.md`):

《史记》→《汉书》实体标注SKILL迁移反思

一、根因分析

问题根源1:历史时代差异
《史记》覆盖黄帝至汉武帝(约2600年),官职体系多变。
《汉书》仅覆盖西汉一朝(约230年),官职体系固定为三公九卿制。

导致:
- 《史记》官职词表以战国/秦制为主
- 《汉书》需要西汉官制词表

问题根源2:文体结构差异
《史记》的"表"以年表为主(时间轴 + 事件)。
《汉书》的"表"增加了职官表、地理表(二维矩阵)。

导致:
- 《史记》的`extract_events.py`假设表格是时间线
- 《汉书》的表格是矩阵,需要新的解析逻辑

二、共性模式

模式1:词表不完整(40%问题)
表现:新增实体类型/词汇在原词表中不存在
案例:大司马、大司徒、大司空、太尉、御史大夫等汉代三公九卿
泛化:任何朝代迁移都会遇到此问题

模式2:文体假设失效(25%问题)
表现:代码中硬编码了《史记》的文体结构
案例:`CHAPTER_TYPES = {'本纪', '表', '书', '世家', '列传'}`写死
泛化:跨语料迁移必须参数化文体类型

模式3:标注粒度不足(20%问题)
表现:原标注体系粒度不够细,无法区分新场景

案例:年号"建元元年"无专门标注,复用`〖%时间〗`导致后续解析困难
泛化:标注体系需要版本演化机制(v2.5 → v3.0)

三、修正建议(优先级排序)

P0:扩展词表(影响100%章节,工作量8小时)
任务:
1. 提取《汉书》前20篇的所有官职标注
2. 与《史记》官职词表对比,找出缺失项
3. 补充到`kg/vocabularies/data/官职词表.md`

预期:新增30-50个汉代官职

P1:参数化文体类型(影响表/志类,工作量12小时)
任务:
1. 将`CHAPTER_TYPES`从硬编码改为配置文件
2. 为每个文体类型定义解析策略
3. 开发`extract_table_structured.py`处理二维表格

预期:支持表格类文献的结构化提取

P2:年号标注设计(影响80%章节,工作量20小时)
任务:
1. 讨论:新增`〖*年号〗` vs 扩展`〖%时间〗`?
2. 如新增,更新标注规范v2.5 → v3.0
3. 开发`parse_reign_periods.py`解析年号
4. 重新标注试点章节

预期:准确标注和解析年号纪年

P3:跨书引用关系(影响20+处,工作量6小时)
任务:
1. 新增关系类型`external_ref`
2. 更新`extract_event_relations.py`识别"见《XX》"模式
3. 输出JSON中增加`external_ref`字段

预期:记录《汉书》引用《史记》的关系

四、新增SKILL内容草稿

SKILL_03a_实体标注.md (汉书版)

§ 汉书特有的标注场景

场景1:年号纪年
《汉书》大量使用年号纪年,如"建元元年"、"元狩五年"。

标注方案 (待最终确定):
- **方案A**:新增`〖*年号〗` — 独立类型,便于后续解析
  - 示例:`〖*建元元年〗`、`〖*元狩五年〗`
- **方案B**:扩展`〖%时间〗` — 复用现有类型,减少标注负担
  - 示例:`〖%建元元年〗`,后处理识别年号模式

建议:采用方案A(新增类型),理由:
- 年号与一般时间词("三年"、"冬十月")语义不同
- 需要专门的年号→公元纪年映射
- 汉书之后的史书大量使用年号,独立类型更利于扩展

场景2:三公九卿官职
西汉确立三公九卿制度,官职体系与战国/秦制不同。

新增官职(示例):
- 三公:大司马、大司徒、大司空
- 九卿:太常、光禄勋、卫尉、太仆、廷尉、大鸿胪、宗正、大司农、少府
- 其他:太尉、御史大夫、丞相(与秦制不同)

标注:仍使用`〖;官职〗`,但需扩展词表。

场景3:表格文献
《百官公卿表》等为二维表格,不同于《史记》的年表。

```
**标注策略**:
- 表头:使用`## 表头`标记
- 表格行:保持原格式,或转为Markdown表格
- 不强制标注表格内每个官职(工作量过大)
- 后续使用专门的`extract_table_structured.py`处理

... (继续补充10-20条新规则)
```

步骤3:人工审查反思结果

关键:不要盲目接受Agent建议

```
## 人工审查检查清单

- [ ] 根因分析是否准确?(是否理解了史记/汉书的本质差异)
- [ ] 共性模式是否可泛化?(是否适用于未来的通鉴/明史?)
- [ ] 修正方案是否可行?(工作量评估是否合理?)
- [ ] 优先级排序是否合理?(P0是否真的必须先做?)
- [ ] 新增SKILL内容是否明确?(是否可直接指导标注?)

## 修改记录
- [x] 根因分析:补充了编年体 vs 纪传体的差异
- [x] 优先级:将P2(年号)提升为P1(影响更大)
- [x] 新增SKILL:增加了反例说明(什么不应该标为年号)
```

步骤4:修正SKILL文档

```
# 应用反思建议,更新SKILL
cp skills/SKILL_03a_实体标注.md skills/SKILL_03a_实体标注_汉书版.md
# 手动或脚本编辑,加入"汉书特有场景"章节

# 更新标注规范
```

```
vim doc/spec/标注格式规范_v3.0.md  # 新增〖*年号〗类型

# 扩展词表
python scripts/augment_vocabulary.py \
    --category 官职 \
    --source doc/transfer/汉代官职补充清单.txt \
    --output kg/vocabularies/data/官职词表_汉书.md
```

3.3 增量验证（Incremental Validation）

原则:每修正一个问题,立即验证

```
# 修正1:扩展官职词表后
# 重新标注试点章节
python scripts/auto_tag.py \
    --input chapter_md/高帝纪上.txt \
    --skill skills/SKILL_03a_实体标注_汉书版.md \
    --vocab kg/vocabularies/data/官职词表_汉书.md \
    --output chapter_md/高帝纪上.tagged_v2.md

# 对比v1 vs v2
diff chapter_md/高帝纪上.tagged.md chapter_md/高帝纪上.tagged_v2.md
# → 检查:新增官职是否正确标注?有无误标?

# 运行Lint检查
python scripts/lint_markdown.py chapter_md/高帝纪上.tagged_v2.md
# → 确认:无新的格式错误

# 人工抽查
head -100 chapter_md/高帝纪上.tagged_v2.md  # 查看前100行
# → 验证:标注质量是否提升?
```

如果验证失败 → 回到步骤3(反思),重新分析

3.4 扩展部署（Scaling Deployment）

时机:试点验证成功(错误率<5%)后

```
# 1. 将修正后的SKILL应用到全部120篇汉书
for chapter in {001..120}; do
    python scripts/auto_tag.py \
        --input chapter_md_hanshu/${chapter}_*.txt \
        --skill skills/SKILL_03a_实体标注_汉书版.md \
        --output chapter_md_hanshu/${chapter}_*.tagged.md
done

# 2. 批量Lint检查
python scripts/lint_all.py chapter_md_hanshu/*.tagged.md >
lint_report.txt

# 3. 按章节生成质量报告
python scripts/generate_quality_report.py \
    --input chapter_md_hanshu/*.tagged.md \
    --output doc/quality/汉书标注质量报告.md

# 4. 人工抽查(每10篇抽1篇)
python scripts/stratified_sample.py \
    --total 120 \
    --sample-rate 0.1 \
    --output doc/quality/汉书抽查清单.txt

# 5. review抽查结果,记录新问题
# → 发现新问题 → 重启反思循环(第二轮)
```

四、迁移成本估算

4.1 工作量模型

经验公式(基于史记项目):

```
总工作量(小时) = 冷启动(20h) + 试点标注(30h) + 错误分析(15h)
                + 反思迭代(N轮 × 25h) + 扩展部署(50h)
                + 质量验证(40h)
```

《史记》→《汉书》预估:

```
相似度: 85% (同为纪传体,时代接近)
试点迭代: 预计2-3轮收敛
总工作量: 20 + 30 + 15 + (3×25) + 50 + 40 = 230小时
日历时间: 约6周(1人全职)
```

《史记》→《资治通鉴》预估:

```
相似度: 60% (编年体 vs 纪传体,结构差异大)
试点迭代: 预计4-5轮收敛
总工作量: 20 + 40 + 20 + (5×30) + 80 + 60 = 370小时
日历时间: 约10周(1人全职)
```

4.2 成本效益分析

场景1:不迁移,从零开始

《汉书》从零构建知识图谱:

```
冷启动: 40小时(摸索标注体系)
规则积累: 200小时(重新发现模式)
工具开发: 150小时(重写lint/extract等)
质量收敛: 100小时(多轮反思)
总计: 490小时
```

场景2:迁移史记SKILL

```
迁移成本: 230小时(如上估算)
节省: 490 - 230 = 260小时(53%)
```

结论:迁移可节省50%+工作量

4.3 迁移收益递增

第一次迁移(史记→汉书):
- 节省53%
- 积累迁移方法论

第二次迁移(汉书→后汉书):
- 节省70%(汉书/后汉书更相似)
- 复用迁移脚本

第N次迁移(二十四史):
- 节省80%+
- 标准化迁移流程(SOP)

长期收益:构建可复用的"史书知识工程工具链"

五、迁移检查清单

5.1 迁移前准备

- [ ] 明确目标语料的结构特点(文体/时代/规模)
- [ ] 评估与源语料的相似度(>80%直接迁移, <60%重新设计)
- [ ] 选择3-5篇试点章节(覆盖多样性)
- [ ] 创建迁移项目目录结构
- [ ] 复制SKILL文档、标注规范、Lint工具到新项目

5.2 试点阶段

- [ ] 应用现有SKILL标注试点章节

- [ ] 运行所有Lint工具,收集错误
- [ ] 创建迁移错误日志(`doc/transfer/*.md`)
- [ ] 记录20-50个失败案例(详细描述+上下文)
- [ ] 统计错误分布(按类型/影响/修正难度)

5.3 反思阶段

- [ ] 生成反思提示词(包含错误日志+SKILL文档)
- [ ] 调用Agent反思,生成修正建议
- [ ] 人工审查反思结果(检查根因/方案可行性)
- [ ] 修正SKILL文档(新增章节/调整规则)
- [ ] 更新标注规范(如需新增类型)
- [ ] 扩展/修正词表和配置

5.4 验证阶段

- [ ] 重新标注试点章节(应用修正后SKILL)
- [ ] 对比修正前后的差异(diff)
- [ ] 计算错误率降低幅度(目标:<5%)
- [ ] 人工抽查样本(至少50个实体/事件)
- [ ] 如验证失败,返回反思阶段(开启新一轮)

5.5 扩展阶段

- [ ] 应用修正SKILL到全部章节
- [ ] 批量Lint检查,生成质量报告
- [ ] 分层抽查(每10篇抽1篇)
- [ ] 记录新发现的问题(第二轮反思输入)
- [ ] 持续迭代直至收敛(错误率<3%)

六、典型场景

场景1:纪传体 → 纪传体（高相似度）

示例:《史记》→《汉书》→《后汉书》→《三国志》

迁移策略:
- 直接复用80% SKILL
- 调整词表(朝代特有词汇)
- 微调参数(事件类型权重)

预期迭代:2-3轮

场景2:纪传体 → 编年体（中等相似度）

示例:《史记》→《资治通鉴》

迁移策略:
- 复用50% SKILL(标注体系、质量控制)
- 重新设计30% SKILL(事件关系、时间对齐)
- 新增20% SKILL(编年体特有场景)

预期迭代:4-5轮

场景3:古文 → 现代文（低相似度）

示例:《史记》→《白话二十四史》

迁移策略:
- 复用30% SKILL(元技能:反思/质量控制)
- 调整40% SKILL(现代汉语NLP工具可用)
- 新增30% SKILL(白话文特有场景)

预期迭代:6-8轮

七、未来展望

7.1 标准化迁移SOP

目标:形成可复制的迁移标准操作流程

```
史书知识工程迁移标准(v1.0)
1. 相似度评估(使用评估矩阵)
2. 试点选择(按文体/难度/覆盖度)
3. 错误日志格式(统一Markdown模板)
4. 反思提示词模板(参数化)
5. 验证标准(错误率阈值/抽查比例)
```

7.2 工具链自动化

规划:开发半自动迁移工具

```
# scripts/transfer_assistant.py

def transfer_skill(source_project, target_project,
pilot_chapters):
    """
    半自动SKILL迁移助手

    步骤:
    1. 复制SKILL/工具到目标项目
    2. 自动标注试点章节
    3. 运行Lint,生成错误日志
    4. 调用LLM反思,生成修正建议
    5. 人工review建议
    6. 应用修正,重新验证
    7. 循环直至收敛
```

```
    """
    # 实现自动化流程(Human-in-the-loop)
    pass
```

7.3 迁移知识库

构想:积累迁移经验,形成知识库

```
迁移知识库/
├── 史记→汉书/
│   ├── 错误日志.md
│   ├── 反思报告.md
│   ├── 修正清单.md
│   └── 验证报告.md
├── 汉书→后汉书/
│   └── ...
├── 史记→资治通鉴/
│   └── ...
└── 迁移模式库.md  # 跨项目共性模式
```

价值:
- 新迁移项目可参考历史案例
- 避免重复犯错
- 加速收敛(第N次迁移比第1次快3倍+)

八、迁移到非古籍与通用书籍

8.1 核心挑战

从古籍知识工程迁移到**现代书籍**或**通用文本**，面临的核心差异：

维度	古籍（史记/汉书）	现代书籍/通用文本
文本结构	高度结构化（纪传体/编年体）	松散结构（章节/段落）
实体密度	极高（5-10个/100字）	中低（0.5-2个/100字）
实体类型	历史专属（人名/官职/邦国）	领域相关（人物/机构/产品等）
时间表达	复杂纪年（年号/干支）	标准年月日
关系复杂度	高（家族/政治/战争）	中（组织/引用/因果）
标注挑战	古文语义理解	歧义消解/同名异指

8.2 迁移层次（通用书籍适配）

Level 1: 直接复用（元技能层，80%+可用）

高度可迁移的内容:
- ✅ **00-META-00 到 00-META-13**: 所有元技能文档
- 反思、迭代、冷启动、柳叶刀方法等核心方法论
- 与具体领域无关，适用于任何知识工程项目
- ✅ **可读性设计**: Token标注优于XML的原则
- ✅ **质量控制体系**: 五层Lint架构
- ✅ **Agent提示词工程**: SKILL驱动的提示词设计

Level 2: 适配调整（标注体系，50%可用）

需要调整的实体类型:

古籍18类实体 → 通用文本实体类型

```
保留（通用性强）：
✅ @人名    → @Person（扩展：组织机构人员）
✅ =地名    → =Location（扩展：虚拟地点）
✅ %时间    → %Time（简化：标准时间格式）
✅ #数量    → #Quantity
✅ ~引文    → ~Citation
```

需要替换（领域特定）：
❌ ^官职 → @Role（通用角色/职位）
❌ &邦国 → @Organization（组织/机构/公司）
❌ &氏族 → @Family（家族/团体，可选）
❌ ^制度 → @Concept（抽象概念）
❌ [刑法 → @LegalTerm（法律术语，可选）
❌ '谥号 → 删除（古籍特有）

新增（现代文本需要）：
➕ @Product（产品/品牌）
➕ @Event（事件名称，如"巴黎协定"）
➕ @Technology（技术术语）
➕ $Currency（货币）
➕ @Email, @URL（结构化信息）

标注体系适配示例:

古籍原文:
〖@秦始皇〗于〖%秦王政二十六年〗灭〖&六国〗，称〖^皇帝〗。

现代文本:
〖@埃隆·马斯克〗于〖%2022年〗收购〖@Twitter〗，担任〖@CEO〗。

Level 3: 重新设计（事件Schema，30%可用）

古籍事件类型 vs **通用事件类型**:

古籍事件（高度专业化）：
- 战争、政变、改革、外交、祭祀、天象...

通用事件（需重新定义）：
- 商业事件：收购/融资/上市/倒闭
- 科技事件：发布/突破/专利
- 社会事件：会议/运动/政策
- 文化事件：出版/展览/获奖

事件Schema适配原则:

1. 保留通用字段：`event_id`，`event_name`，`event_type`，`participants`，`location`，`time`
2. 删除古籍特有字段：`dynasty`，`reign_year`，`calendar_system`
3. 新增现代字段：`url`，`source_credibility`，`impact_score`

8.3 试点策略（通用书籍）

试点选择原则

不要选择:

- ❌ 小说/文学作品（虚构，实体关系弱）
- ❌ 高度技术文档（术语密集，需专业知识）
- ❌ 新闻快讯（碎片化，缺乏深度关系）

推荐试点:

- ✅ **传记类书籍**（如《乔布斯传》）
- 与古籍"列传"结构相似
- 人物关系密集
- 时间线清晰
- ✅ **历史通俗读物**（如《人类简史》）
- 事件驱动
- 概念丰富
- 结构化章节
- ✅ **商业案例集**（如《创新者的窘境》）
- 实体类型明确（公司/产品/人物）
- 因果关系清晰

试点规模

```
第一阶段（MVP）：
- 1本书
- 1-3章（约5000-10000字）
- 目标：验证标注体系适配性

第二阶段（扩展）：
- 同一本书的完整标注
- 目标：积累错误模式，完善SKILL
```

第三阶段（泛化）：
- 3-5本不同类型的书
- 目标：提炼通用方法论

8.4 关键差异的应对策略

差异1：结构化程度降低

古籍优势：
- 纪传体天然的人物索引
- 表的时间锚点
- 书的分类体系

现代书籍问题：
- 章节划分不规则
- 缺乏统一的时间线
- 内容跳跃性强

应对策略：
1. **人工预处理**：为每章添加 `summary` 和 `key_entities` 元数据
2. **双层标注**：先标注章节级大纲，再标注段落级实体
3. **弱化时间线**：不强求所有事件都有精确时间

差异2：实体歧义性增加

古籍相对确定性：
- "秦始皇"在《史记》中指向唯一
- 地名有明确地理边界
- 官职有标准定义

现代文本挑战：
- "苹果"可能是水果/公司/品牌/手机
- "华为"可能指公司/产品系列/技术标准
- 同名人物频繁（需上下文消歧）

应对策略：
1. **类型前缀强制**：`〖@苹果公司〗` vs `〖#苹果〗`（数量）
2. **上下文窗口扩大**：从±100字扩展到±200字
3. **消歧提示词增强**：显式提供领域背景

差异3：关系发现难度增加

古籍关系密集：
- 家族关系：父子/君臣关系明确
- 战争关系：攻守双方清晰
- 时序关系：纪年体系统一

现代文本关系隐含：
- 组织关系：隶属/合作/竞争需推理
- 引用关系：观点引用需理解语义
- 因果关系：跨段落因果链复杂

应对策略：
1. **分层关系发现**：
- L1: 共现关系（自动，基于统计）
- L2: 显式关系（LLM提取，如"X创办Y"）
- L3: 隐式关系（人工标注+Agent推理）
2. **降低关系完整性要求**：从"完整图谱"降级为"核心关系骨架"
3. **用户反馈循环**：允许读者标注缺失关系

8.5 成功案例构想

案例1：《乔布斯传》知识图谱

迁移适配：
- 实体类型：@Person（乔布斯/沃兹尼亚克），@Organization（苹果/皮克斯），@Product（Mac/iPhone）
- 事件类型：创业/产品发布/人事变动/并购
- 关系类型：创始/领导/竞争/合作

预期效果：
- 自动生成"乔布斯时间线"
- 发现"苹果产品演化图"
- 提取"创新方法论"（SKU）

案例2：《人类简史》概念图谱

迁移适配：
- 实体类型：@Concept（认知革命/农业革命），@Group（智人/尼安德特人），@Technology（文字/货币）

- 事件类型：革命/迁徙/发明
- 关系类型：因果/演化/对比

预期效果：
- 自动生成"人类发展阶段图"
- 提取核心论点（事实三元组）
- 构建"历史规律"知识单元

8.6 ROI评估（非古籍场景）

成本对比

项目	古籍（史记）	现代书籍	差异
标注体系设计	已完成	需调整2周	+2周
试点标注	3章	3章	持平
错误模式积累	5轮迭代	预计3轮	-40%
工具复用率	基准	80%	提速5倍
总体时间	6个月	2个月	节省67%

结论：迁移到现代书籍反而**更快**，因为：
1. 文本语义更直白（无古文理解门槛）
2. 实体类型更标准（无先秦姓氏推理）
3. 时间处理更简单（标准公历）

适用场景推荐

高价值场景（ROI > 3倍）：
- 📚 专业教材（如医学/法律教材）→ 领域知识图谱
- 📖 经典著作（如《资本论》）→ 思想体系图谱
- 📰 长篇报道（如调查报告）→ 事件关系网络

低价值场景（ROI < 1倍）：
- 短新闻（碎片化，无深度关系）

- 娱乐小说（虚构，应用价值低）
- 个人博客（非权威，质量参差不齐）

8.7 长期愿景：通用知识工程方法论

从"史记知识工程"到"通用知识工程":

```
史记项目积累（2026）
    ↓
古籍迁移验证（汉书/通鉴，2027）
    ↓
现代书籍试点（传记/历史，2027-2028）
    ↓
垂直领域扩展（医学/法律，2028+）
    ↓
通用知识工程平台（2029+）
```

最终目标:

> 形成一套**领域无关**的知识工程方法论，支持从任意文本（古籍/现代书/学术论文/网页）快速构建高质量知识图谱，实现"文本→知识→应用"的全自动流水线。

九、总结

核心要点

1. **分层迁移**:直接复用 → 适配调整 → 重新设计
2. **试点驱动**:不要一次性迁移,用小规模试点发现问题
3. **反思循环**:错误日志 → Agent反思 → SKILL修正 → 验证
4. **增量验证**:每修正一个问题,立即验证效果
5. **Human-in-the-loop**:关键决策需要人工审查,不盲目信任Agent
6. **领域扩展**:元技能高度可迁移,标注体系需适配,事件Schema需重设计

迁移哲学

传统软件迁移:
- 一次性大规模迁移
- 依赖完整的需求分析
- 失败风险高(all-or-nothing)

本项目的迁移理念:
- **小步快跑**:试点 → 反思 → 修正 → 扩展
- **拥抱失败**:将失败案例视为学习机会
- **知识沉淀**:每次迁移都积累经验,形成可复用资产

核心洞察:

> **SKILL文档不是一次性设计,而是在多项目迁移中持续演化的知识资产**

迁移过程本身就是知识提炼的过程:通过发现不同语料间的差异,我们更深刻理解了"什么是可泛化的知识工程方法论"。

与其他元技能的关系

- ← **冷启动**:迁移=加速的冷启动
- ← **反思**:迁移的核心驱动力
- ← **质量控制**:迁移后需验证质量不倒退
- ← **柳叶刀方法**:分层迁移对应分层分解
- → **规模化**:成功迁移是规模化的前提

技能迁移是项目可持续发展的关键:从一个史书扩展到二十四史,需要高效的迁移方法论。