SMO??

SMO ????????

> 设计层面的 "为什么"。算子表 / 投影律 / 良构律 / 范例 在 `SMO指令.md`（由 `spec/smo_spec.yaml` 自动渲染）。两文档零重叠。

??SMO ???

SMO（Semantic Meta Operator） = 把自然语言投影为 类型化图 的形式系统。

```
自然语言句子                SMO 算式（中间形式）              Typed graph（最终消费形态）
──────────────              ──────────────────────            ──────────────────────────────

他能够使用想象力        他 > 能力(使用) > 想象力        proposition(verb=使用,
                                                                  slots.agent=他,
                                                                  gates.modality=能力,
                                                                  slots.theme=想象力)
```

人读自然语言；下游图扩散读 graph；SMO 在中间作桥梁 ——把汉语句子的语义形式化得足够紧，让下游 graph parser 机械抽取节点和边。

??????

```
第一层 · 三个算子                  第二层 · 5 个 scope（按作用维度）          第三层 · 6 元原则
──────────────────                 ──────────────────────────────────        ──────────────────

 :    挂载（M : H → H'）            slot    → prop.slots.<role>              M1 不双重编码
 >    驱动（concept 之间）          gate    → prop.gates.{modality,...}      M2 槽语义保真
 ( )  封装 + 函数式 op(args)        conn    → logic_prop                     M3 作用域不交错
                                    junct   → junct[T] 节点                  M4 算子函数式
```

mod → modifier 边 M5 canonical ≠ 内容词

M6 省略要级联

整张关系字典都是 第二层的填表 ； 4 投影律（ PL1-PL4）和 3 良构律（ WL1-WL3）都是 第三层 **6** 元原则在不同 **scope** 上的实例化 。

?????? scope??? letter class

v1 的 11 个 letter class（ B/C/D/E/F/G/H/I/J/K/L）是按 "中文功能词的细分语义 "切的； v2 的 5 个 scope 是按 "在 typed graph 上作用的字段维度 "切的。后者更本质：

- syntax 形态由 scope 决定： slot 必 `r(C):V`， gate 必 `r(V)`， conn 必 `r(P, Q)`， junct 必 `r(X, Y, ...)`， mod 按 target 选挂载方式。
- graph 投影机械可推： scope 决定字段位置， canonical 名决定字段值。
- 类内细分（如 modality / polarity / quantifier / tense 都是 gate）只用于语义路由，不影响 syntax。

把 11 类合到 5 类后，跨切面规则数量大约从 30+ 降到 4——因为原 P-rule 多数是 "某 class 不能这样用 " 的特化形式， scope 一统就退化成 PL1 的子情况。

?? `:` ?????????

`:` 是单一修饰挂载算子： `M : H -> H'`（ M 挂到 H 上，输出与 H 同型）。合法 (M, H) 对仅三种模式（详见 `SMO指令.md §1.1`）：

- **NP-attribution**： concept/cluster/prop 修饰 concept/cluster（包括主动定语从句）
- **Slot-mount**： scope=slot 的关系挂在 verb/prop 上
- **Cluster-mod**： scope=mod & target=cluster 的关系挂在 cluster 上

新增挂载形态 = 表里加一行，不需要新 W 规则。早期 fix1-fix20 反复补 W-N 的根因是把 `:` 当成多种独立操作；统一为 (M, H) 表后这类反复就消失了。

?? `>` ?????

`>` 仅承载值（ concept/cluster/prop），不能是关系名。原因： graph 投影时 `A > V > O` → `proposition(verb=V, slots.agent=A, slots.theme=O)`， A/V/O 都是节点 ID，关系名混入会破坏整一性。

所有关系都用函数式 `r(args)` 承载，不入 `>` 中段。这是 PL1（M4 函数式纯净）的根据。

??surface ?????v2 ?????

v1 在算式里既写 surface 也写 canonical（`了` ↔ `完成`），多义 surface 还要带 `[canonical]` 标签。**v2** 退役这套机制：

- 算式里只写 **canonical**（`完成(走)` 而非 `了(走)`）
- surface 退到关系字典的 `surfaces` 字段，只在回写方向用作 readback 候选
- 多义 surface 在投影方向直接由 LLM 用上下文消歧，不再需要标签

代价：失去原句字面词偏好（v1 保留 `了/已经/已`，v2 默认只用首项）。换来的是：

项	v1	v2
算子表条目	70+ canonical × 80+ surface	70 canonical（surface 只是 readback hint）
多义消歧	14 行 multi_surfaces 表 + `[canonical]` 语法	无
LLM 提示词	~6k token	~3k token
P-rule 数	27	4（PL1-PL4）
Lint E-code 数	25+	3 良构律 + 类型检查

兼容性：parser 仍接受旧式 surface 写法（`了(通过)` `都[全称]`），demo 暂不需要重写。新生成的算式应优先 canonical 形式。

???? = ????

```
LLM:  自然语言 → SMO 算式 → 自然语言
                                ↑
                          应与原句语义近似
```

所有规则（PL/WL/M）都是辅助工具。最终判定是回译保真：投影出来的算式让 LLM 回写回中文，应跟原句语义近似。

推论：

- 回译不一致 = **LLM** 转换错误，不是 SMO 算式错误。
- 形式良构 ≠ 语义正确 —— `S > V > 0` 良构但 V 选错 canonical 时回译就崩。
- **lint** 不能保证正确，只能保证良构；语义错只能靠回译人审或 demo 基准比对捕捉。

回译没有自动化　　metric（中文有多种合法变体），是人审　　+ demo 基准的混合校验。

??Graph schema??????

SMO 算式的最终消费方是　　typed graph:

```
Concept 节点:    canonical / role / vector
Action prop:     verb + slots.<role> + gates.{polarity,modality,quantifier,tense,voice}
Logic prop:      antecedent + consequent + connector_class
边类型:          slot:<role> / modifier / within_cluster / coref / seq
```

每个 SMO 关系 → graph 字段的映射由 `spec/smo_spec.yaml` 中关系的 `scope` 与 `role`/`gate_field` 决定，机械投影。完整对照见 `SVO语义检索的系统化方案.md §2.1`。

下游扩散检索消费的是 graph，不是 SMO 文本。所以 SMO 设计的最高目标是 **"投出来的 graph 字段干净 " + "回译能保真 "**——其它原则都为这两件事服务。

???????

9.1 ?????????

文件	作用	形态
`SMO手册.md`（本文件）	设计原理 / why	手写
`SMO指令.md`	关系字典 / 投影律 / 良构律 / 范例	自动渲染
`spec/smo_spec.yaml`	单一数据源	手编 yaml
`spec/render/instruction.py`	yaml → `SMO指令.md` 渲染器	手写代码
`spec/lint/{parser,validator}.py`	SMO 算式 parser + 校验器	手写代码
`fix/fix*.md`	design log（不是规范）	手写
`demo/*.txt`	NL ↔ SMO 配对回归基准	人工标注

派生关系:

```
spec/smo_spec.yaml
    ├──→ SMO指令.md                （自动渲染）
    ├──→ spec/lint/validator   （直接读 yaml 校验算式）
```

```
└──→ 未来: parser / graph 投影
```

9.2 ????

```
改原理 / 加新论证                    →  改本文件（手写）
改关系字典 / 投影律 / lint      →  改 spec/smo_spec.yaml → python -m spec.build.cli render
改 lint 实现                          →  改 spec/lint/validator.py
任何改动后                            →  python -m spec.build.cli lint demo
```

9.3 fix*.md ? design log

`fix/fix1.md` ~ `fix/fix25.md` 是历史档案 ——记录 "为什么从 A 改到 B"的设计辨论。它们 不是当前规范，规范在本文件 + `SMO指令.md` + `spec/smo_spec.yaml` 三处一致表达。

新增 fix-N 的目的是留下 "为什么这样改 + 影响了哪些 demo 行 / 哪些 lint"的决策档案，规则改 yaml 即可。

? · ????

- 第一次读: 本文件全篇（约 15 分钟） → `SMO指令.md` §三 关系字典 + §七 范例（ 10 分钟）
- 投影时查询: `SMO指令.md` §三 关系字典 + §四 投影律
- 实现 parser / graph 投影: `spec/smo_spec.yaml` + 本文件 §八 graph schema
- 修订规范: 本文件 §九

Revision #1
Created 2026-05-29 11:07:31 UTC by Colin
Updated 2026-05-29 11:07:31 UTC by Colin