Hypothesis 工具核心代码模块分析报告

组员：张忠云 陈思敏

分工：张忠云负责模块 1、模块 2、模块 4

陈思敏负责模块 2、模块 3、模块 4

完成日期：2026.1.5

目录

1 工具与核心模块基础梳理

1.1 工具基础信息

Hypothesis 是一种面向 Python 语言的性质测试（Property-Based Testing, PBT）自动化测试工具，其核心思想是由用户描述程序应当满足的通用性质（property），再由工具自动生成大量测试输入对这些性质进行验证。与传统基于固定测试用例的单元测试方法不同，Hypothesis 通过自动化输入生成的方式，能够覆盖更广泛的输入空间，从而更容易发现隐藏在边界条件和极端情况下的程序错误。

Hypothesis 最初起源于学术界对性质测试与测试用例自动生成技术的研究成果，后发展为成熟的开源工程项目，并在工业界和学术界得到广泛应用。该工具由社区持续维护与迭代，当前版本更新频繁，功能稳定，生态完善。Hypothesis 采用 Mozilla Public License 2.0（MPL-2.0）开源协议，允许在遵循协议要求的前提下进行修改和商业使用。

在运行环境方面，Hypothesis 主要支持 Python 语言，通常与 pytest 等主流测试框架配合使用，适用于算法验证、数据结构测试、系统逻辑校验等多种场景。其主要应用目标并非形式化地“证明程序正确性”，而是通过大规模、自动化的动态测试手段，提高发现错误的概率和效率，属于基于性质测试的技术范畴。

1.2 核心模块定位

本次调研聚焦“当 Hypothesis 找到违反性质的输入后，如何对该输入做约减（shrinking）”。

从源码结构上，shrinking 逻辑主要位于：

https://github.com/HypothesisWorks/hypothesis/blob/master/hypothesis-python/src/hypothesis/internal/conjecture/shrinker.py，包含 Shrinker 核心实现（约减调度、

候选变换、排序规则等）。

https://github.com/HypothesisWorks/hypothesis/blob/master/hypothesis-python/src/hypothesis/internal/conjecture/engine.py：Conjecture 引擎在运行测试时导入并调用 Shrinker，管理“生成-失败- 约减”的整体流程与限额（例如最大 shrink 次数、最大 shrink 时间等）。

在 engine.py 中可以看到引擎导入 Shrinker, sort_key，并设置了与 shrink 相关的关键全局常量，例如 MAX_SHRINKS = 500 、MAX_SHRINKING_SECONDS = 300（超过时提前终止约减并给出警告）。

1.3 模块功能关联

Hypothesis 的输入约减模块对应本课程中讲授的基于性质测试核心技术。

在 PBT 方法中，测试的重点不在于人工构造具体测试用例，而是由开发者描述程序应满足的通用性质，并由工具自动生成大量输入进行动态验证。当发现违反性质的输入时，首次得到的失败用例往往规模较大、结构复杂，不利于理解和调试。输入约减模块正是 PBT 测试流程中的关键组成部分，其作用是在保持失败性质不变的前提下，对失败输入进行自动化简化，生成更小、更直观的反例。

因此，该模块体现了 PBT 中“反例最小化”和“测试用例规约”的核心思想，使性质测试不仅能够发现错误，还能提供易于分析的最小失败用例，是基于性质测试技术工程化落地的重要支撑。

2 核心代码模块原理分析

2.1 技术原理复盘

2.1.1 目标及意义

Hypothesis 性质测试的优势是自动生成输入覆盖大量边界情况，但它带来一个现实问题：第一次发现的反例往往很大、很乱（例如一个很长的列表、很复杂的嵌套结构），直接拿去调试不利于定位根因。Shrinking 的目标就是：在保证仍能触发同一失败的前提下，将失败输入不断简化，最终得到最小（或足够小）

且可读的反例。

2.1.2 输入和输出

从整体设计上看，输入约减模块并不直接处理用户层面的 Python 数据结构，而是作用于失败用例在内部表示层面的生成过程信息。

输入：

1）一个已确认违反性质的失败测试用例；

2）与该失败用例对应的输入生成轨迹（即测试输入在生成过程中的选择序列或中间表示）；

3）测试函数本身，用于对候选输入进行重新执行与验证。

输出：

1）一个经过约减后的最小失败用例；

2）约减过程中的统计信息（如成功约减次数、耗时、停止原因等），用于调试与性能控制。

2.1.3 基本流程

Hypothesis 的输入约减模块是在性质测试执行过程中发现失败之后被触发的一个独立阶段，其整体流程可以概括为“发现失败 → 记录生成轨迹 → 受限搜索 → 输出最小反例”。

首先，Hypothesis 按照用户定义的性质与输入生成策略运行性质测试，自动生成测试输入并执行被测函数或断言。当某一次执行过程中出现断言失败或运行时异常时，测试引擎会确认该输入违反性质，并将其判定为失败用例。

在确认失败后，Hypothesis 并不会立即终止测试，而是记录该失败用例在内部的生成轨迹信息，即输入在生成过程中所经历的一系列选择决策（如 choice sequence 或中间表示节点）。这些生成轨迹为后续的输入约减提供了基础。

随后，测试流程进入 shrinking 阶段。引擎首先对约减过程设置工程化约束条件，包括最大约减次数（如 MAX_SHRINKS）以及最大约减时间（如 MAX_SHRINKING_SECONDS），以避免在复杂场景中产生不可控的运行开销。

在约减阶段中，shrinking 模块会基于当前失败输入对应的生成轨迹，反复提出一系列“更小”的候选输入。这些候选由多种约减规则生成，其目标是在输入规模、数值范围或结构复杂度上逐步简化原始失败用例。

对于每一个候选输入，Hypothesis 都会重新运行测试函数进行动态验证。如果候选输入在实际执行中仍然违反性质，并且在规模或复杂度上优于当前失败用例，则该候选会被接受，并作为新的基准继续进行约减；如果候选无法复现失败，则会被直接丢弃并回退到当前基准输入。

上述过程会不断迭代，直到无法生成更优的失败候选，或达到预设的次数/时间限制。最终，shrinking 模块输出一个仍能稳定复现错误的最小失败用例，并同时给出约减过程的统计信息，例如停止原因（搜索空间耗尽、达到限额或无更优候选）等。

通过这一流程，Hypothesis 将原本可能复杂且难以理解的失败输入，自动规约为简洁、直观的反例，从而显著提升基于性质测试在实际调试与错误定位中的实用价值。

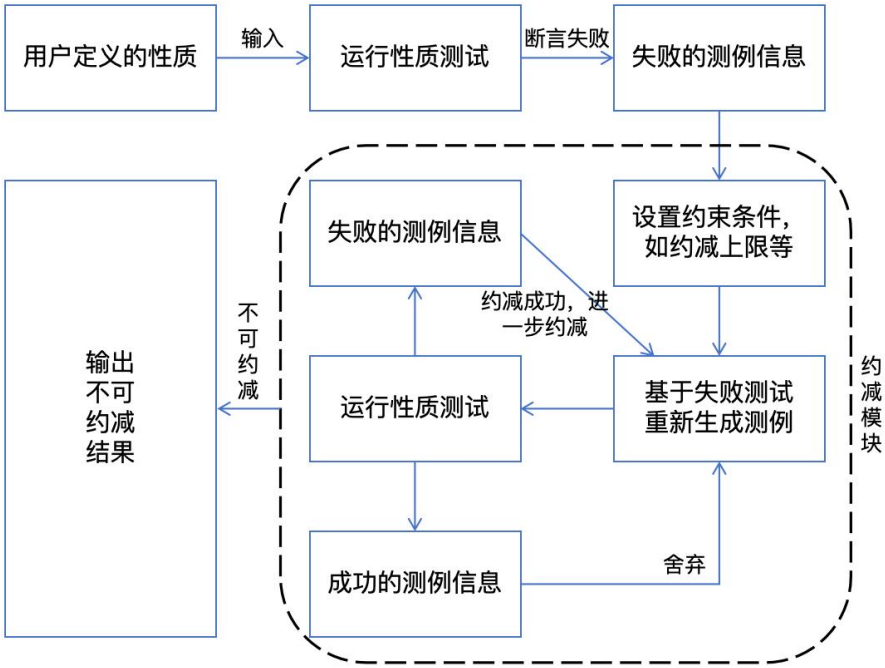


图 1 Hypothesis 运行流程

2.2 源码阅读与逻辑拆解

2.2.1 输入

Shrinking 模块在进入执行阶段时，主要接收以下三类输入信息：

1）已确认违反性质的失败测试用例

该输入由测试引擎在性质测试阶段确认，并在创建 Shrinker 对象时以 initial 参数形式传入。该对象通常为 ConjectureResult，并被初始化为当前的 shrink_target，作为后续约减的起点。

2）失败用例对应的输入生成轨迹

输入生成轨迹并非独立参数，而是封装在失败用例对象内部，主要包括：choices：测试输入的选择值序列；nodes：包含类型、取值及约束信息的 ChoiceNode 序列；spans：反映输入结构层次关系的区间信息。

这些数据共同描述了测试输入的生成过程，为 shrinking 阶段构造更小候选输入提供基础。

3）测试函数执行能力

Shrinker 并不直接持有用户定义的测试函数，而是通过 ConjectureRunner 引擎对象间接调用。所有候选输入的验证均通过 engine.cached_test_function(...) 完成，从而保证测试执行与缓存机制的一致性。

2.2.2 处理

Shrinking 的核心处理流程由 Shrinker.shrink() 方法触发，其内部逻辑可划分为以下几个阶段：

1）初始化与粗粒度约减

调用 initial_coarse_reduction() 对输入结构进行一次性预处理，用于消除明显冗余或结构性复杂的生成路径，为后续精细约减创造条件。

2）多策略 shrink pass 迭代执行

Shrinker 内部维护了一组预定义的 shrink pass（如 minimize_individual_choices、reorder_spans、try_trivial_spans 等）。greedy_shrink() 通过调用 fixate_shrink_passes(self.shrink_passes)，反复执行这些 shrink pass，直到在当前 shrink_target 上无法再产生改进。

3）候选生成与验证循环

每个 shrink pass 在执行过程中，会基于当前 nodes 构造新的候选输入，并通过 consider_new_nodes(...) 触发验证流程：

候选输入首先被送入 cached_test_function(...)；

该函数调用引擎重放测试函数；

若候选仍违反性质且在 sort_key() 顺序上更小，则通过 update_shrink_target(...) 更新当前最优失败用例。

4）终止条件判断

Shrinking 在以下情况下终止：

所有 shrink pass 均无法进一步改进当前失败用例；

连续多次调用未产生有效缩减，触发 StopShrinking；

引擎层达到时间或调用次数限制。

2.2.3 输出

Shrinking 模块的最终输出包括：

1）最小失败测试用例

即 shrink_target 中保存的测试输入，其在保持违反性质的前提下，在长度和复杂度上达到局部最小。

2）辅助调试信息（可选）

若启用 explain 模式，Shrinker 会调用 explain()，生成关于输入中关键片段（slice）的说明信息，用于帮助用户理解哪些输入部分对失败行为具有决定性影响。

2.2.4 shrink 模块关键函数调用关系

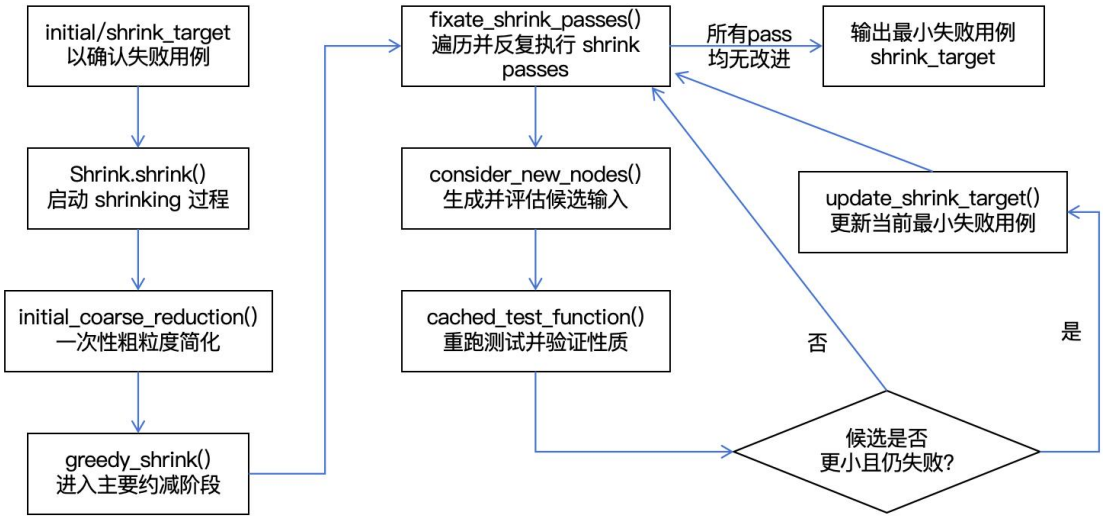


图 2 代码调用过程

2.3 前沿技术关联

Hypothesis 的输入约减模块体现了多项与前沿软件测试技术相关的设计思想。其基于输入生成轨迹（choices、nodes、spans）进行结构化反例最小化，体现了语义感知与结构化测试的理念；通过 shortlex 顺序作为约减目标函数，将 shrinking 过程转化为有序搜索问题，与搜索型测试和启发式优化方法相契合；同时，模块采用多种 shrink pass 并结合停滞检测机制进行自适应调度，体现了资源受限搜索与自适应测试策略的工程化实践。

3 调试验证与效果复现

步骤 1：环境准备（安装）

1. 安装运行环境：
   - Python 3.9.6
   - pytest 8.4.2
   - hypothesis 6.141.1
2. 安装测试依赖：pip install pytest hypothesis

步骤 2：构建简单被测对象

1. 编写一个简单的函数 my_abs()，用于模拟绝对值计算。

(1) 构造一个简单函数作为被测对象，用于避免复杂系统对测试工具分析过程的干扰。

(2) 该函数在输入为 0 时存在人为引入的错误，用于观察 Hypothesis 的错误发现能力。

(3) @given(st.integers())

让 Hypothesis 自动生成“任意整数”作为参数 x

(4) assert result >= 0

用于测试性质而非单纯的样例结果，比如 my_abs(3) == 3

test_abs.py 如下：

```
from hypothesis import given
from hypothesis import strategies as st
from abs_bug import my_abs

@given(st.integers())#
def test_abs_non_negative(x):
    result = my_abs(x)
    assert result >= 0
```

步骤 3：执行指定测试文件

1. 测试案例 abs_bug.py 文件：

```
def my_abs(x: int) -> int:
    if x < 0:
        return -x
    if x == 0:
        return 1   # 故意引入 bug
    return x
```

输入：pytest work_sm/test_abs.py

结果：pytest 成功收集并运行 1 个测试用例，输出为：

```
collected 1 item
work_sm/test_abs.py . [100%]
1 passed
```

```
(venv) (base) simin@simindeMacBook-Pro lab4 % pytest work_sm/test_abs.py
======================================== test session starts ========================================
platform darwin -- Python 3.9.6, pytest-8.4.2, pluggy-1.6.0
rootdir: /Users/simin/Desktop/ss/lab4-simin98/lab4
plugins: hypothesis-6.141.1
collected 1 item

work_sm/test_abs.py .                                                                          [100%]

========================================= 1 passed in 0.20s =========================================
```

2. 设置测试不通过的案例 abs_wrong；

```
def my_abs(x: int) -> int:
    if x < 0:
        return -x
    if x == 0:
        return -1   #明显违反性质的 bug。"
    return x
```

```
(venv) (base) simin@simindeMacBook-Pro lab4 % pytest work_sm/test_abs.py

======================================== test session starts ========================================
platform darwin -- Python 3.9.6, pytest-8.4.2, pluggy-1.6.0
rootdir: /Users/simin/Desktop/ss/lab4-simin98/lab4
plugins: hypothesis-6.141.1
collected 1 item

work_sm/test_abs.py F                                                                          [100%]

============================================= FAILURES ==============================================
_____________________________________ test_abs_non_negative _________________________________________

    @given(st.integers())
>   def test_abs_non_negative(x):

work_sm/test_abs.py:6:
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

x = 0

    @given(st.integers())
    def test_abs_non_negative(x):
        result = my_abs(x)
>       assert result >= 0
E       assert -1 >= 0
E       Falsifying example: test_abs_non_negative(
E           x=0,
E       )
E       Explanation:
E           These lines were always and only run by failing examples:
E               /Users/simin/Desktop/ss/lab4-simin98/lab4/work_sm/abs_wrong.py:5

work_sm/test_abs.py:8: AssertionError
===================================== short test summary info =======================================
FAILED work_sm/test_abs.py::test_abs_non_negative - assert -1 >= 0
========================================= 1 failed in 0.27s =========================================
```

3. Hypothesis 在自动生成大量整数输入的过程中发现测试失败，并通过 shrink 机制逐步简化失败输入，最终定位到最小失败用例 x = 0。

输入查看失败断点：

pytest -x --pdb work_sm/test_abs.py

```
(venv) (base) simin@simindeMacBook-Pro lab4 % pytest -x --pdb work_sm/test_abs.py

========================================= test session starts ==========================================
platform darwin -- Python 3.9.6, pytest-8.4.2, pluggy-1.6.0
rootdir: /Users/simin/Desktop/ss/lab4-simin98/lab4
plugins: hypothesis-6.141.1
collected 1 item

work_sm/test_abs.py F
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> traceback >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

    @given(st.integers())
>   def test_abs_non_negative(x):

work_sm/test_abs.py:6:
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

x = 0

    @given(st.integers())
    def test_abs_non_negative(x):
        result = my_abs(x)
>       assert result >= 0
E       assert -1 >= 0
E       Falsifying example: test_abs_non_negative(
E           x=0,
E       )
E       Explanation:
E           These lines were always and only run by failing examples:
E               /Users/simin/Desktop/ss/lab4-simin98/lab4/work_sm/abs_wrong.py:5

work_sm/test_abs.py:8: AssertionError
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> entering PDB >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> PDB post_mortem (IO-capturing turned off) >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
> /Users/simin/Desktop/ss/lab4-simin98/lab4/work_sm/test_abs.py(8)test_abs_non_negative()
-> assert result >= 0
```

进入 Pdb 后查看关键信息：

输入：where

p x

p result

输出：

```
(Pdb) p x
0
(Pdb) p result
-1
```

测试失败后进入 pdb，调用栈显示从 pytest → Hypothesis wrapper → test_abs.py → my_abs；在断点处观察变量 x=0，result=-1，验证失败由 abs_wrong.py 第 5 行分支触发，与预期一致。

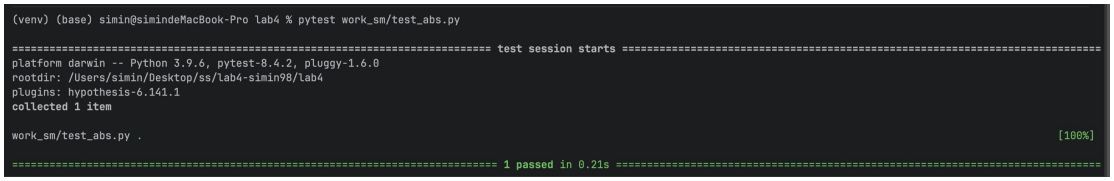
根据 Hypothesis 提供的最小失败用例，对被测函数进行修复，构建 abs_true.py。

```
def my_abs(x: int) -> int:
    if x < 0:
        return -x
    return x
```

重新运行测试：

输入：pytest work_sm/test_abs.py

结果通过



实验展示了 Hypothesis 在性质测试中的完整工作流程：自动生成输入 → 发现失败 → shrink 定位最小失败用例 → 修复缺陷 → 验证性质重新成立。

4 优缺点与技术落地反思

4.1 核心模块优缺点分析（Shrinking 输入约减模块）

4.1.1 优点

1.显著降低调试成本：自动生成最小可复现反例

性质测试往往能快速触发失败，但首次失败输入可能规模大、结构复杂，不利于定位根因。

Shrinking 模块通过“保持失败性质不变 + 尽可能简化输入”的策略，将失败用例自动规约为更小、更直观的反例。结合本次复现，Hypothesis 最终收敛到 Falsifying example: x=0，并在 PDB 中观察到 x=0, result=-1，且失败路径精确指向 abs_wrong.py:5 的错误分支。这类“最小反例 + 精确定位”的输出，直接提升了从失败现象到根因定位的效率。

2.结构化约减：利用生成轨迹（choices/nodes/spans）而非盲目删减

从源码拆解看，Shrinking 不直接对用户层的 Python 结构做粗暴裁剪，而是基于失败用例对应的生成轨迹信息（choices、nodes、spans）进行结构化候选

生成与验证。这种做法的优势在于：

(1) 候选输入与生成过程一致，减少“无效候选”比例；

(2) 对嵌套结构、组合结构等复杂输入更友好；

(3) 约减过程可被组织为多个 shrink pass 的组合，具备工程可维护性和可扩展性。

3.可控的工程化边界：次数/时间限额保证可用性

引擎层（engine.py）为 shrinking 设置了明确的“资源上限”，如 MAX_SHRINKS 与 MAX_SHRINKING_SECONDS。这意味着 shrinking 在最坏情况下不会无限运行，能在复杂输入/复杂性质场景下保持可控。对实际工程而言，这种“上限约束”是性质测试落地的关键：否则测试阶段容易出现不可接受的长尾耗时。

4.多策略 shrink pass + 停滞检测：在局部最小化上更稳健

Shrinker 通过一组 shrink pass（如对 choice 的逐项最小化、对 spans 重排或尝试更“平凡”的结构等）进行迭代式贪心约减，并配合停滞检测（例如无法进一步改进触发停止）避免无效循环。从工程角度看，这是“启发式搜索 + 终止条件”的经典组合，在不追求全局最优的前提下，以较低成本换取可用的局部最优反例。

5.与执行引擎一致的验证机制：缓存与重放降低重复成本

Shrinking 的每个候选都需要“重新执行测试函数以验证仍失败”。源码中通过 engine.cached_test_funct

4.1.2 局限性

1.性能开销不可避免：shrinking 本质是“受限搜索 + 反复重跑测试”

Shrinking 的核心操作是：生成候选 → 重跑测试 → 判断是否仍失败且更小。对于成本高的被测函数（例如涉及 IO、网络、复杂算法、随机性或外部依赖），候选验证会造成显著耗时；即使引擎设置了最大次数/最大时间，仍可能出现“测试耗时大、且最终反例不够小”的情况。换言之，shrinking 的质量与成本强依赖于“单次测试执行成本”和“性质定义是否稳定”。

2.约减结果是“局部最小”而非全局最小，且与排序规则强相关

shrinking 通常以 shortlex（短者优先、再按字典序）一类排序规则（sort_key）作为“更小”的判定依据，追求的是可读性强、规模更小的局部最小反例。这种目标函数对调试非常实用，但也意味着：

(1) 最终反例不保证是全局最小；

(2) 不同策略、不同 pass 顺序可能导致不同但等价的最小反例；

(3) 对某些结构化输入，“长度最小”未必等同于“语义上最直观”。

3.适用场景存在前提：被测逻辑需尽量确定性，外部依赖会干扰 shrink Hypothesis 依赖“候选输入重放时仍能稳定复现失败”才能收敛。若被测程序存在显著非确定性（时间、随机数、并发竞态、外部服务返回等），会造成：

(1) 候选结果不确定，shrinking 难以推进；

(2) shrink 可能提前停止，输出反例不稳定；

(3) 调试证据弱化（难以建立稳定因果链）。

因此，shrinking 更适合纯函数、数据结构、算法性质验证、以及可控环境下的系统逻辑校验。

4.对使用者提出抽象要求：性质（property）若定义不当，会导致误用或“无效测试”

性质测试不是“自动证明正确性”。如果性质定义过弱（例如仅检查返回值非空），即使 shrunk 输入很小也无法发现真实 bug；如果性质定义过强或与真实需求不一致，则可能产生“假失败/误报”或把合法行为当成错误。本次示例中性质“结果应非负”足够清晰，因此能得到高质量反例；但在工程场景中，性质抽象不当会直接降低工具收益。

4.2 技术落地反思

许多分析/验证工具能指出“存在问题”，但缺少可读、可复现的证据链，导致工程落地困难。Hypothesis shrinking 的关键价值在于：把失败从“抽象结论”转化为“最小可复现反例”，并与调用栈/断点/变量观察自然衔接（本次通过 pytest -x --pdb 的调试流程形成闭环）。对同类工具而言，提升落地能力的抓手往往不是“更强的检测能力”，而是“更好的反例与解释机制”。

将复杂过程拆为可组合策略（passes），利于扩展与维护。Shrinker 采用多

shrink pass 的设计，把“如何让输入更小”的策略拆解成可组合、可调度的单元。

性质测试与反例最小化天然有“搜索爆炸”风险。Hypothesis 将预算约束上移到引擎层，使得工具在复杂输入下仍具有“可预期的最坏开销”。同类工具在落地时也通常需要类似设计：给搜索/优化/约简过程设定预算与早停规则。当前 shrinking 的目标主要是“更短、更小”。在复杂结构输入场景，可考虑引入更贴近语义可读性的目标函数，优先消除对失败无贡献的结构片段；对对候选验证进行批处理或并行（在可控副作用/可重入条件下），或更细粒度地复用中间执行结果等方式提高效率。

5 参考文献

[1]D. R. MacIver, “Compositional shrinking,” Hypothesis, Dec. 8, 2016. [Online].Available:https://hypothesis.works/articles/compositional-shrinking/

[2]D. R. MacIver et al., “Hypothesis: The property-based testing library for Python,” GitHub repository, accessed Jan. 8, 2026. [Online]. Available: https://github.com/HypothesisWorks/hypothesis

[3]A. Zeller and R. Hildebrandt, “Simplifying and isolating failure-inducing input,”IEEE Transactions on Software Engineering,vol. 28, no. 2, pp. 183–200, Feb. 2002, doi: 10.1109/32.988498.

[4]D. R. MacIver, “Integrated shrinking,” Hypothesis. [Online]. Available: https://hypothesis.works/articles/integrated-shrinking/. Accessed: Jan. 8, 2026.