Billion-Context 解析

为什么长任务 Agent 不需要塞满百万上下文?

刘政东 · 2026-03-24

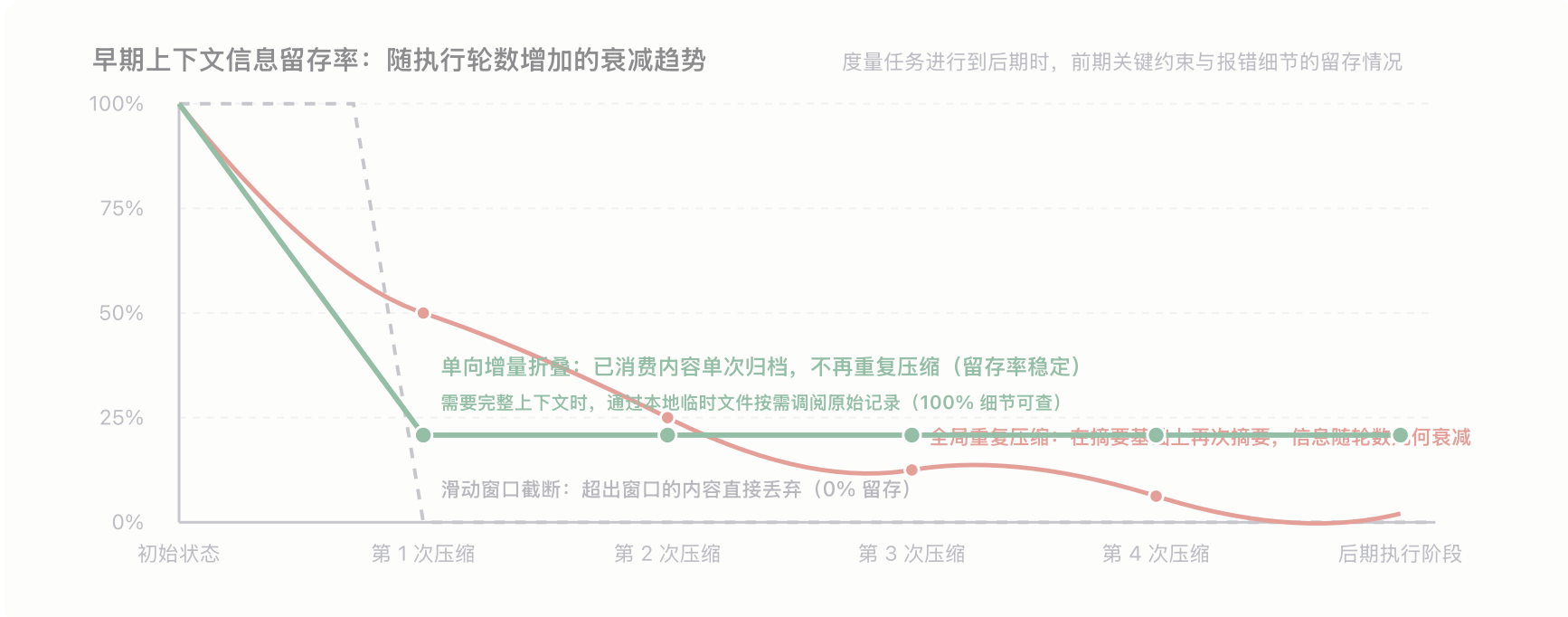
现状与假象

- 模型厂商把上下文窗口卷到 1M、2M
- 宣传宣称“能一口气读完几十本书”
- 但在实际工程场景中：
  - 跑自动化重构
  - 连续执行自动化单测
  - 多轮调试定位缺陷
- 轮数一多，Agent 依然频繁翻车、注意力涣散

长任务的真实痛点

1. **注意力稀释**
   - 上下文塞满上百次命令的冗长输出
   - 早期立下的架构约束与核心红线被模型遗漏
2. **延迟与成本飙升**
   - 每次推理都要全量 Prefill 十几万 Token
   - 账单坐火箭，首字延迟（TTFT）拉长至几十秒

两种常见解法的死穴

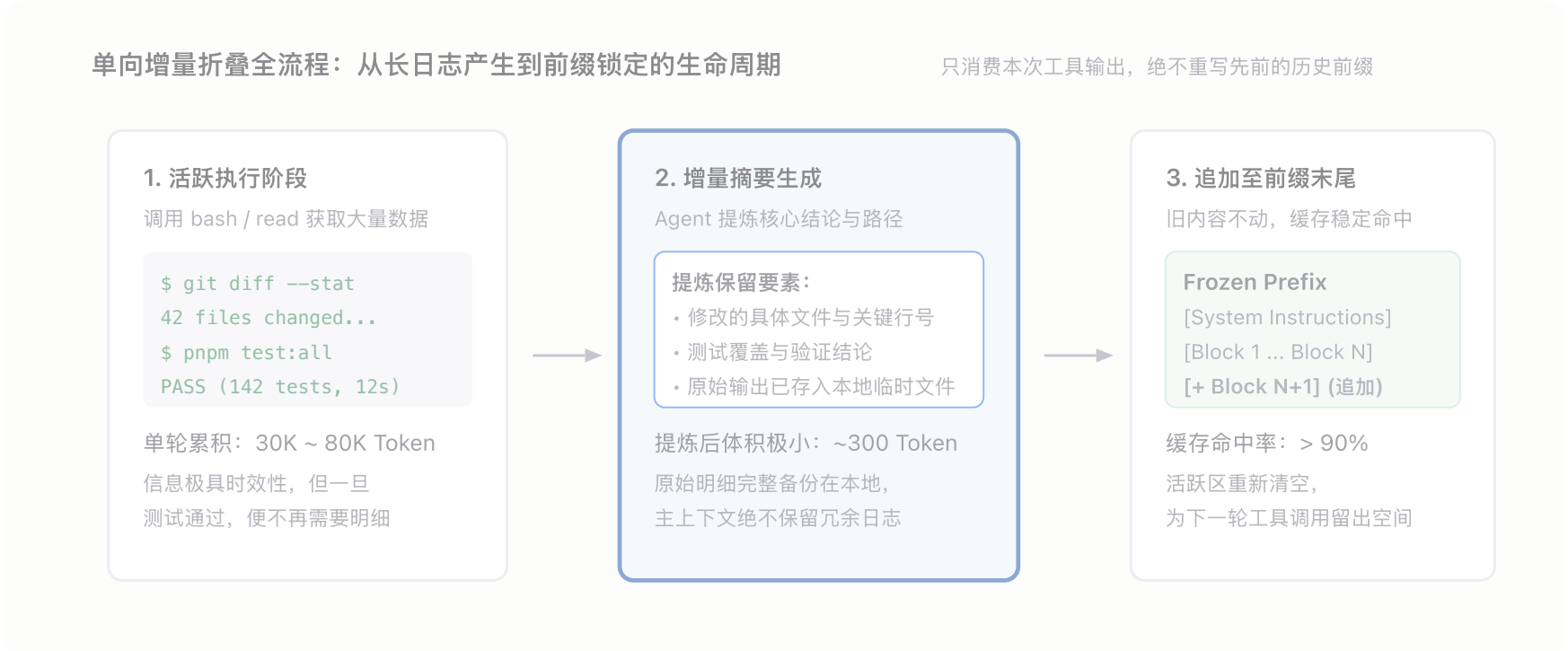


- **滑动窗口截断**：直接丢弃前文，任务规范与初始目标凭空消失
- **全局重复重写**：
  - 在摘要上再做摘要，细节几何衰减

核心解法：不可变前缀


上下文结构划分：不可变前缀区与动态活跃区
保护 Prompt 前缀不被修改，是稳定命中 API 缓存折扣的关键
固定指令区 (System Instructions & Tools)
Agent 身份、环境约束、工具函数定义（全生命周期保持严格不变）
100% 缓存命中
历史摘要区：不可变前缀 (The Frozen Prefix)
已结束轮次的工具输出与讨论，经单次提炼为只读摘要块（按时序单向追加）
Block 1: 架构与依赖调研
耗费 45K Token → 提炼为 300 词
Block 2: 核心代码重构
耗费 60K Token → 提炼为 450 词
前缀锁定 · 享 90% 缓存折扣
折叠线 (The Fold)
活跃工作区：动态交互 (The Active Window)
当前轮次的用户反馈、最新文件读取、单次 bash 运行输出、Agent 当前思考
在此处快速生成与读取，任务单元完成后，整段直接打包推入上方前缀区，不再回改
单次消耗 · 随轮次迭代

增量提炼流程



1. 活跃区执行命令，产生几万行输出
2. 提炼核心结论、修改的文件路径与测试状态

核心原语与物理锚点

- **三大核心标识**：
  - `m` (Message Ref)：单条消息的唯一物理坐标，单调递增不可变
  - `b` (Block ID)：折叠提炼后的独立摘要块
  - `t` (Tier / Token / Tail)：收敛层级、Token 计量、被影响的尾随 Token
- **为什么是 Compress-as-Anchor?**
  - **传统合成消息 (Synthetic)**：强行塞假系统消息，破坏角色契约且打碎缓存
  - **物理锚点 (Anchor)**：将 Agent 原生调用的 `compress` 原汁原味钉在时间线原位，100% 顺畅继承底层缓存

摘要的三级演进与封顶机制

- **Tier 1 现场工单** (~300t)：保留文件路径、函数名、改动行号与报错现场
- **Tier 2 架构决策** (~80t)：丢弃调试细节，只保留"最终结论、决策理由 Why 与避坑教训"
- **Tier 3 事实索引** (~20t)：降维成纯单行索引（一件事一句话，≤8 词）
- **为什么在 Tier 3 封顶?**
  - **语义物理极限**：再压就成了毫无工程价值的废话
  - **容量完全覆盖**：100 条 Tier 3 事实也仅 2,500 Token，足以支撑上千轮长期任务
  - **同级垃圾回收 (GC)**：淘汰废弃模块，合并同类项，永不越界膨胀

核心拷问：多层收敛在哪收？前缀坏不坏？

多层收敛的真实机制：收敛位置与前缀缓存重付（Re-Pay）代价

收敛发生在历史前缀区内部，前缀虽被破坏，但经由盈亏平衡迅速收回

收敛前：历史前缀中累积的大量底层碎片块 (Tier 1 Blocks)

Block b3 (300t) | Block b4 (400t) | ..... | Block b15 (350t) | 尾随内容 Tail: b16 及后续 (T Tokens)

compress({ startId: "b3", endId: "b15", summary: "..." })

收敛后：多个旧块被替换为单一的高阶里程碑块 (Tier 2)

Tier 2 块 (仅 ~200 Token)
高度提炼 b3~b15 的最终结论与架构状态，原碎片细节落盘至本地文件

⚠ **前缀哈希改变：缓存失效**
尾随的 T Tokens 需全额重新计费 (Re-Pay)

为什么值得破坏前缀？前缀缓存经济学（Breakeven Analysis）

1. 一次性重购代价 (One-Time Re-Pay)：尾随内容 T 重新写入 + 摘要生成输出，产生一次性开销 ΔC_1
2. 后续每轮永久节省 (Per-Turn Saving)：整段上下文砍掉几千 Token，后续每轮稳定省钱 Δs
3. **盈亏平衡轮数 $n^* = \Delta C_1 / \Delta s$：实测只需 2 ~ 9 轮即可收回成本，后续几十轮纯净赚上千万 Token**

最精妙的设计：按需落盘查阅

历史回溯机制对比：重灌上下文 vs 导出本地文件按需读取

当 Agent 在第 100 轮需要核查第 5 轮的具体报错时

常见做法：整段重新灌回上下文

将 50,000 Token 的原始记录重新解压进 Prompt

产生的问题：

1. 上下文瞬间暴涨 50K Token
2. 打碎已缓存的历史前缀，后续请求重新全量计费
3. 再次引发注意力稀释，模型更容易遗漏眼前任务

后果：成本飙升，后续推理延迟成倍增加

更好的方案：解压为本地文件按需读

将历史输出落盘到 /tmp/，仅读取所需片段

```
decompress({ blockId: "b5", toFile: "/tmp/b5.txt" })
read({ path: "/tmp/b5.txt", offset: 140, limit: 20 })
```

- 原始万行日志全部写在磁盘上，不占上下文
- 读回的只有关键的 20 行报错堆栈（仅 200 Token）

收益：前缀缓存完全不受影响，细节 100% 精确可查

- **传统死穴**：把历史原始数据重新解压塞回上下文，打碎前缀
- **Billion-Context 做法**：

工程师视角的启示

- **上下文是昂贵的 L1 Cache**，不是无限倾倒的垃圾桶
- **保护 Prompt 前缀**，就是保护响应延迟和调用成本
- **善用文件系统与本地工具**：
  - 与其把海量输出硬塞给大模型
  - 不如让它学会像人类工程师一样去查阅本地日志