Billion-Context 解析：为什么真正的长任务 Agent，不需要也不该塞满百万上下文?

模型窗口卷到 1M、2M 之后，我们在实际跑自动化重构、长链条调试等任务时，Agent 为何依然频繁变笨？本文深入拆解超长上下文的注意力稀释机制，并以 billion-context（ACP）为样本，剖析不可变前缀、Compress-as-Anchor 物理锚点、多层收敛中真实的前缀破坏与缓存重购（Re-Pay）经济学模型，以及三级摘要演化与按需文件回溯的系统设计。

16 分钟阅读

这两年各大模型厂商把上下文窗口从 8K、32K 一路卷到了 1M 甚至 2M，宣传语里总是写着“能一口气读完几十本书”。

但只要你在实际工程里让 Agent 跑过真实的重型任务——比如修复一个跨越十几个文件的重构、自动跑测试、根据几十次报错反复调试代码——你就会发现一个尴尬的事实：**窗口虽然装得下，但只要对话轮数一多，Agent 马上就开始变笨、原地打转，甚至把最初立下的规矩忘得干干净净。更别提账单上的 Token 费用像坐火箭一样飙升，每回一次消息都要等上半分钟。**

最近在各类前沿 Agent 框架中以 ACP（Agent Context Protocol）等形态落地的 billion-context 机制，给出了一套完全不同的工程解法。今天我们不谈虚头巴脑的概念，像一线系统工程师那样，把超长上下文的底层痛点、常见做法的死穴，以及 billion-context 在**不可变前缀、物理锚点、多层收敛中真实的前缀破坏与缓存重购（Re-Pay）经济学**上的精妙权衡，彻底拆解透彻。

真实世界里的长任务，到底在痛什么?

先看一个真实的场景：你让 Agent 重构一个老项目的认证模块，要求它保持向下兼容、写完跑单元测试、根据报错自行修复。

Agent 开始干活：

1. 跑了一次全局搜索，终端吐出 200 个匹配结果（消耗 15K Token）；
2. 读了 5 个核心文件，代码又占了 20K Token；

3. 改了几处代码，跑了一次 `pytest`，报了 8 个错误，堆栈信息刷了几百行（再加 10K Token）；
4. 它修了两个错，又跑了一次测试……

才刚刚来回 10 轮，上下文就已经堆了十几万 Token。

此时系统会立刻在底层遭遇两个硬伤：

1. 注意力稀释与有效上下文衰退（Attention Dilution）

从 Transformer 的数学机理来看，Softmax 注意力权重的总和恒等于 1。当上下文中塞满了成千上万行已经过时的构建日志、废弃的代码片段时，注意力权重被迫在大量噪声中归一化，分母急剧膨胀。

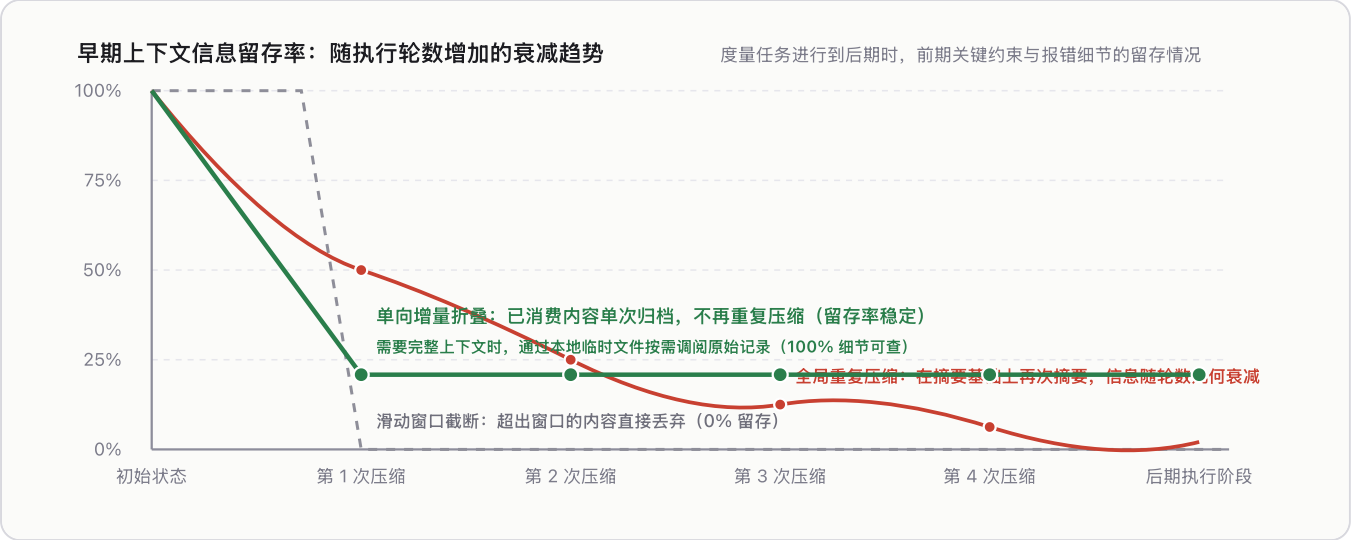
其结果就是**关键约束的权重被稀释**：你在第一轮明确强调的“禁止修改公共接口签名”，到了第 15 轮，模型可能只分配了极微弱的注意力给这句话，顺手就把签名改坏了；甚至在第 30 轮遇到报错时，它会开始在两个互不兼容的方案之间反复打转、原地自愈失败。大窗口给的是“容量”，却不是“注意力质量”。

2. 首字延迟（TTFT）与推理算力的无底洞

每一次让大模型生成新 Token，服务端都必须对当前整段上下文做一次全量 Prefill。尽管模型具备 KV Cache，但如果上下文毫无节制地膨胀到 10 万、20 万 Token，每次交互哪怕模型只输出一句“正在修改”，你也必须为十几万 Token 的二次计算等待数秒至数十秒。在交互密集的自动化 Agent 里，这种延迟是完全不可接受的。

为什么常见的两类解法都有死穴?

面对上下文膨胀，目前市面上绝大多数框架采取的处理手段非常原始，代价也极高。



早期上下文信息留存率对比

做法 A：滑动窗口直接截断（Sliding Window Truncation）

这是最朴素的做法：设置一个固定窗口（如只保留最近 20 条消息），超出部分直接从队列头部切掉。

这种方式在闲聊场景下感知不明显，但在工程 Agent 中是灾难性的。你在最开始告知的“系统架构约束”、“环境配置细节”、“核心业务红线”，都会在截断的一瞬间彻底蒸发。Agent 会丧失状态机的全局一致性，变成一个只有“短期记忆”的盲人，开始反复踩前几轮已经踩过的坑。

做法 B：全局重新总结（Global Compaction）

很多流行框架在上下文快满时，会唤起一次模型，让它“把以上全部对话总结成一段摘要”，然后用这段摘要直接替换掉前面的所有历史。

这看似保住了信息，但埋下了两个极其隐蔽的系统级暗坑：

1. **信息熵损耗的几何放大**： 在一段长达数万字的原始执行记录上做一次总结，至少丢掉 80% 的具体行号、精确配置与调用参数；当任务推进到第 50 轮，框架对“上一次的总结 + 新内容”再做第二次总结。如此循环三次，早期的关键事实就被彻底稀释成了“此前对数据库进行了重构”这样毫无工程执行价值的废话。
2. **彻底打碎前缀缓存（Prompt Caching）**： 这是最昂贵的隐性成本。目前主流云端模型（Anthropic Claude、DeepSeek、OpenAI）都上线了基于精确前缀匹配的 Prompt Caching 机制：**只要上下文的前半段文本保持一字不差，命中部分的读取费用通常只有原价的 10%~20%，首字延迟也能降低近一个数量级。** 如果你采用全局重新总结，意味着你修改了上下文最开头的历史文本。整段 Prompt 的前缀哈希瞬间全变，原先建立的几万乃至十几万 Token 的 KV Cache 瞬间清零！后续的每一次请求都必须以全额全价重新从头 Prefill。

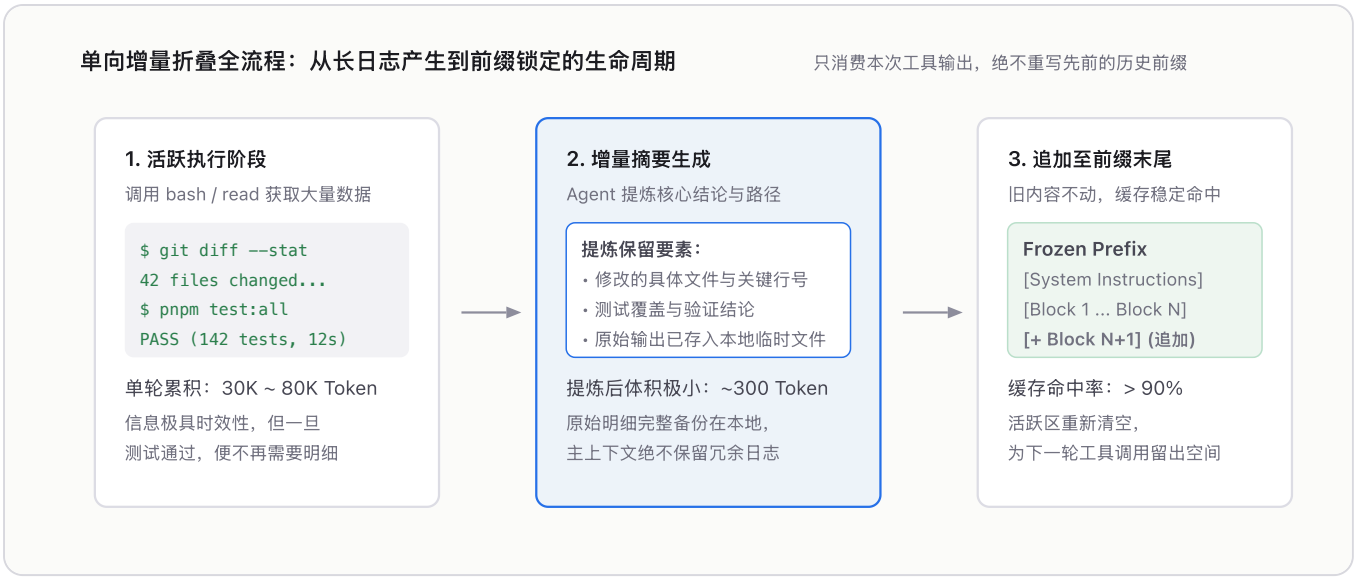
增量折叠与不可变前缀（The Fold）

针对上面的死穴，billion-context 的底层核心思路是：**将上下文划分为明确的分区，把已确认的历史作为不可变前缀锁定，只允许单向追加，严禁随意回头修改。**

上下文结构划分：不可变前缀与活跃区

系统在内部划定了一条“折叠线（The Fold）”，将 Prompt 严格切分为两个区域：

1. **不可变前缀区（The Frozen Prefix）**：
    - **固定指令层**：Agent 的 System Prompt、环境定义、工具元数据。这部分在整个会话生命周期中字符完全静止，100% 稳定命中服务商的根缓存；
    - **已折叠历史块（Blocks）**：已经执行完毕并被提炼的历史片段（Block 1, Block 2.....）。它们按时间顺序单向向后追加，一旦落入前缀区，**默认不修改、不重写、不移位**。
2. **动态活跃工作区（The Active Window）**：
    - 处于折叠线之下，包含当前正在进行的 1~2 轮交互：用户的新指令、Agent 发起的 `bash`、`read` 工具调用，以及工具返回的原始长日志。



单向增量折叠全流程

增量折叠的运行闭环

当 Agent 在活跃区里完成了一个阶段性的工作（比如跑完了一次单元测试，并分析出了修复方案），这一批长达数万 Token 的工具输出就完成了它的历史使命。

此时，Agent 主动调用 `compress` 工具，**仅针对活跃区里这段刚刚消费完毕的内容**做一次单向增量提炼：

- 提炼出的摘要非常紧凑（通常 200~400 Token），只包含核心结论、修改的文件路径与关键行号；
- 生成的新摘要块（Block $N+1$）直接挂载在折叠线上方的前缀末尾；
- 活跃区内被提炼的原始长日志被移出主上下文（但在本地持久化保留）。

关键点在于：每一次日常折叠完全发生在尾部。 前面已有的 System Prompt 和历史摘要块（Block 1 ~ Block N）没有发生哪怕一个字符的改变。新请求的前半段能够完整命中上一次在云端建立的 KV Cache，缓存命中率通常稳定在 90% 以上！既释放了宝贵的上下文窗口，又最大化保护了缓存折扣。

关键实现考量：为什么是 Compress-as-Anchor 而不是合成伪造消息？

在把历史折叠掉时，绝大多数框架会选择**伪造消息（Synthetic Message Injection）**：把原始消息删掉，强行注入一条假的 `{ role: "user", content: "[系统自动总结...]" }`。

但 billion-context 彻底否决了这种做法，选择了 **Compress-as-Anchor（以工具调用为物理锚点）**：

1. **守护大模型角色状态机（Role Invariants）**：云端 API 严格要求消息交替轮替。凭空硬插伪造消息极易触发模型 API 报错；

2. **成为不可变前缀的物理锚点**：Agent 自己调用的 `call: compress(...)` 本身就是一次合规的工具调用。折叠后，海量原始日志被移出，但**这次工具调用以及它参数里的摘要文本，原汁原味地钉在时间轴原位，成为一个不可移动的物理锚点（Anchor）**。下一次交互时，新内容直接往它后面挂载，100% 顺畅继承底层 KV Cache，模型更不会对消息来源产生认知幻觉。

核心拷问：多层收敛是在哪收的？前缀到底会不会被破坏？

看到这里，任何有架构经验的工程师都会立刻提出一个尖锐的问题：

> **“如果一直只在尾部追加 Tier 1 摘要块，任务跑上几百轮，前缀里岂不是会堆几十上百个摘要块？如果要收敛成更高阶的里程碑，这个收敛到底是在哪里收的？收完了，前缀难道不会被破坏吗？”**

这是一个直击 billion-context 底层设计的灵魂问题。我们直接看系统的真实实现。

多层收敛的真实机制与缓存重付

1. 收敛到底发生在哪里?

在系统中，收敛分两种完全不同的路径：

- **日常折叠（Tier 1 Fold）**：收在**活跃区前端（折叠线边缘）**。它处理的是刚刚在尾部产生的新日志，新生成的块直接追加在前缀末尾，**不修改历史，前缀完全不坏**。
- **多层收敛（Tier 2 / Tier 3 Distillation）**：收在**历史前缀区内部**！ 当长任务跑了 80 轮，历史区里已经积攒了 15 个细碎的 Tier 1 块（`b1 ~ b15`）时，系统会触发多层收敛机制。Agent 会发起一次跨块压缩：

  ```
  compress({ content: [{ startId: "b3", endId: "b15", summary: "..." }] })
  ```

  这次操作的作用目标是**已经存在于前缀中间的连续历史块**。系统将 `b3 ~ b15` 这一串细碎的底层操作记录移除，替换为一个概括宏观阶段的高阶里程碑块（Tier 2 Block）。

2. 收完了前缀是否会被破坏?

答案是：会破坏，而且是 100% 必然破坏！

大模型服务商的 Prompt Caching 采用的是**严格的自顶向下前缀匹配（Exact Prefix Match）**。计算哈希是从第一个 Token 开始逐字向后计算的。 当你把前缀中间的 `b3 ~ b15` 替换成了一个新的 Tier 2 块，从 `b3` 所在的位置开始，后续所有 Token（包括后面的 `b16` 以及活跃区的所有新内容）的绝对位置和前缀哈希全部发生了改变。

在 GPU 显存里，从改动点往后的所有已缓存 KV 全部失效！

3. 既然会破坏前缀，为什么系统还要做多层收敛?

既然破坏前缀会导致缓存失效，为什么不干脆永远不收敛，一直单向追加 Tier 1?

因为**物理缓存虽然省钱，但无法解决模型的“认知带宽”极限**。 如果连续跑 100 轮，即使每个 Tier 1 块只有 300 Token，100 个碎片块堆在上下文里也有整整 30,000 Token。更要命的是，上下文里充斥着上百个琐碎的细节描述（“某轮改了 A 文件”、“某轮修了 B 接口”）。面对如此多细碎的历史片段，大模型的注意力再次涣散，它根本无法一眼看清宏观的架构演化阶段，最终导致决策失误。

所以多层收敛是一个必然的选择：**它必须在“保住物理缓存”与“保住模型注意力清晰度”之间找到平衡。**

4. 前缀缓存经济学（Breakeven Analysis）

billion-context 的精妙之处在于，它并没有天真地把前缀当成绝对不可触碰的死规矩，而是**建立了一套精确的“前缀缓存经济学数学模型”，对每一次收敛进行盈亏平衡测算**。

在系统的底层源码中，每一次折叠都会精确计算以下指标：

- S：被折叠掉的原始历史块总 Token 数；
- σ **(sigma)**：提炼后生成的高阶摘要 Token 数；
- T：折叠点之后、被连带破坏前缀缓存的尾随内容 Token 数（Tail Tokens）；
- w **(Cache Write Price)**：缓存写入或未命中单价（基准全价 1.0）；
- r **(Cache Read Price)**：缓存命中单价（基准折扣价 0.1，即 1 折）；
- q **(Output Price)**：模型输出 Token 单价（基准 4.0，模型生成摘要的费用）。

当多层收敛替换前缀中的旧块时，系统产生了两笔账：

账目一：一次性重购成本（One-Time Re-Pay Cost，ΔC_1）

因为前缀被破坏，改动点之后的尾随内容 T 在下一次请求中无法命中缓存，必须以写入单价 w 重新全额计算，再加上模型生成摘要本身的输出开销：

$$\Delta C_1 = (w - r)T + q \cdot \sigma - r \cdot S$$

这就是破坏前缀需要向模型厂商缴纳的“一次性重付罚款”。在系统账单报表中，这一项被明确列为 `compress re-pay`。

账目二：后续每轮的永久节省（Per-Turn Saving，Δs）

虽然交了罚款，但多层收敛把原本几十个旧块的庞大体积彻底抹平了，全局上下文永久净减少了 $(S - \sigma)$ 个 Token。 在随后的每一轮交互中，只要新前缀重新建立起缓存，每次请求都少算 $(S - \sigma)$ 个 Token，按命中单价 r 计算，每轮稳定省钱：

$$\Delta s = (S - \sigma) \cdot r$$

盈亏平衡轮数（Breakeven Turns，n^*）

将一次性重购成本除以每轮节省金额，就得到了极其关键的**盈亏平衡轮数**：

$$n^* = \frac{\Delta C_1}{\Delta s} = \frac{(w - r)T + q \cdot \sigma - r \cdot S}{(S - \sigma) \cdot r}$$

只有当两次多层收敛之间的执行轮数 $k \geq n^*$ 时，这次破坏前缀在财务和算力上才是真正净赚的（PAID BACK）！

5. 实测账单：真实运行中的数据印证

在真实系统的运行报表（`acp_cache`）中，我们可以看到真实的收敛核算结果：

```
GRAND LEDGER
  total input    14.71M tok
  total cached   12.76M tok  (hit 86.8%)
  miss breakdown (input − cached = 1.95M tok):
    new content     277.3K tok  (fresh append — not an invalidation)
    compress re-pay 26.5K tok   (4 folds — re-billed prefix)
    ttl/other       1.64M tok   (TTL expiry / network idle)

FOLD ECONOMICS
  gross saved 10.12M tok · repay cost 26.5K tok · summary cost 1850 tok → net
+10.10M tok
  verdict: 3 PAID BACK / 0 NOT PAID BACK
  #1: S=43.0K  σ=978  T=11.5K  ΔC₁=9.9K  Δs=4.2K/turn  n∗=2.4  k=68 → PAID BACK
  #2: S=14.9K  σ=424  T=15.0K  ΔC₁=13.7K Δs=1.4K/turn  n∗=9.5  k=34 → PAID BACK
```

看 `#1` 的这组真实数据：

- 一次收敛消除了 43K Token 的旧块，生成了 978 Token 摘要，连带影响了 11.5K Token 的尾随前缀；
- 破坏前缀导致的一次性 Re-Pay 代价约为 9,947 Token，而收敛后每轮交互稳定节省 4,205 Token；
- 盈亏平衡轮数 $n^* = 2.4$ **轮**！
- 也就是说，**只要后续任务继续跑 3 步，这次破坏前缀交的“过路费”就被完全赚了回来**！
- 实际上，该任务在收敛后继续推进了 68 轮（$k = 68 \gg 2.4$），整个生命周期不仅没有因为破坏前缀亏损，反而净省下了超过 1000 万 Token 的巨额开销！

摘要的三级演化：从现场工单到极简索引

为了把控好每一次收敛的信息保真度，系统定义了严格的 Tier 1 ⟶ Tier 2 ⟶ Tier 3 三级演进体系。

假设你在让 Agent 做一个耗时很长的任务：**“重构认证模块，将老旧的 RSA 签名算法升级为 ED25519，并跑通全套单测”**。

它在上下文中的三级演进过程非常典型：

Tier 0：原始现场（约 65,000 Token）

- 终端打印了 5 个源码文件的全文（20K Token）；
- 跑了 3 次单测，报了 8 个跨文件错误堆栈，刷了几百行报错（35K Token）；
- 中间改错了代码、撤回、重新搜索……

Tier 1：现场工单（约 300 Token）

- **定位**：给**当前具体干活**用的。丢弃冗长日志，但保留**具体文件路径、函数名、改动行号、测试状态**。
- **内容示例**：

> - 修改了 `auth/jwt.py:45-88` 中的 `verify_token` 函数签名，更新密钥加载逻辑；
> - 初次运行 `pytest tests/test_auth.py` 时报错 `ValueError: Invalid key header`，排查发现是测试 mock 证书格式不匹配，在 `tests/conftest.py:12` 修复了公钥编码；
> - `pytest tests/test_auth.py` 14 个测试用例全部通过。

Tier 2：架构里程碑（约 80 Token）

- **定位**：给**下一步宏观决策**看的。任务推进到第 50 步，堆积了十几个 Tier 1 时触发：**彻底丢弃具体的调试过程、报错堆栈和行号，只保留“最终结论、设计决策理由（Why）、避坑教训”**。
- **内容示例**：

> - **认证模块迁移**：已将签名算法从 RSA 升级为 ED25519；
> - **核心决策**：选择保持 `verify_token` 外部接口签名不变，内部向下兼容老 Token，避免影响下游 12 个微服务；
> - **避坑经验**：测试环境 Mock 证书必须使用带 Header 的标准 PEM 格式，不可使用裸 Base64。

Tier 3：极简事实索引（约 20 Token）

- **定位**：任务推进到了 150 步以上，连决策理由也嫌长。系统做极度浓缩：**连“决策理由”和“经验教训”都扔掉，降维成纯单行事实索引（Lookup Index）**，一件事一句话（不超过 8 个单词）。
- **内容示例**：

> - `Auth-v2 迁移完成 — ED25519 升级上线且向下兼容`
> - `Bug #402 修复 — 证书加载解析异常`

核心追问：Tier 3 还能不能继续压缩成 Tier 4?

源码中对于收敛层级有明确的封顶边界：

```
const outputTier = isBlockBoundary ? Math.min(3, targetTier + 1) : 1;
```

系统在设计上把层级严格封顶在 Tier 3，绝不再往上建 Tier 4。

为什么不再往上压?

1. **语义密度的物理极限**： Tier 3 已经被压缩成了最极限的单行事实索引（“某模块 — 某成果”）。如果还要继续强行压缩成 Tier 4，它就只能变成“重构了系统”、“修了 Bug”，连涉及什么模块、达成了什么效果都被抹杀了。这会导致**不可逆的语义崩塌**，模型彻底丧失寻址索引能力。
2. **长任务容量已经足够支撑“无限对话”**： 一条 Tier 3 事实只有约 25 个 Token。哪怕一个长任务跑了上千轮、产生了 100 个大阶段里程碑，这 100 条 Tier 3 索引加在一起也仅仅占用 **2,500 Token**！在现代模型的上下文窗口里，2,500 Token 的开销几乎可以忽略不计（不足 1%），根本不需要冒着语义丢失的风险升到 Tier 4。
3. **同级垃圾回收（Tier 3 GC）**： 如果 Tier 3 积累过多，系统不升阶，而是在同级做淘汰：自动剔除已被后续版本作废（`[OBSOLETE]`）的陈旧索引，或将同一模块的多条记录合并为一行。

深入系统骨架：核心组织原语与关键要素

在 billion-context（ACP）架构中，系统通过一套严密的标识原语和要素来驱动整个生命周期：

1. 三大核心标识：`m` / `b` / `t`

标识	全称	典型示例	物理含义与底层职责
`m`	Message Ref（消息引用）	`m00001`，`m00150`	**单条消息的唯一物理坐标**。在会话生命周期内全局单调自增（以 `m` 开头 + 5 位数字）。每一轮 User 输入、Assistant 回复、Tool Call 与 Tool Result 装配进 Prompt 时被打上标签，作为只读锚点供折叠精确定位。
`b`	Block ID（摘要块标识）	`b1`，`b3`，`b15`	**折叠提炼后的独立摘要块**。一段连续的 `m` 原始日志被提炼后，原始消息移出，原地挂载一个轻量的摘要块 `b`。它既是底层摘要，也是更高阶收敛的输入目标。
`t`	Tier / Token / Tail	`t2`，`300t`，`T=11.5K`	根据上下文承担三项职责：**Tier（收敛层级）**（标明 `t2` / `t3`）；**Token 计量**（如 `300t`）；**Tail Tokens (T)**（前缀经济学中被破坏缓存的尾随 Token 数）。

2. 系统核心要素全景

- **The Fold（折叠线）**：切分不可变前缀（享受 90%+ 缓存折扣）与动态活跃工作区。
- **Protected Ranges（受保护区）**：活跃区底部的最新用户输入和执行中的工具调用，严禁折叠，避免破坏上下文一致性。
- **Compress-as-Anchor（工具调用锚点）**：不使用伪造合成消息，以原生 `compress` 工具调用作为前缀物理锚点。
- **Decompress-to-File（解压缩落盘）**：历史原始记录不回填进 Prompt，按需解压到 `/tmp/` 本地文件，模型通过带偏移量的 `read` 工具切片查阅。

历史回溯：用文件系统替代“重新塞回上下文”

在长任务中，还有一个所有做上下文管理的人都逃不开的经典矛盾：**如果任务推进到第 150 轮，模型突然需要核对第 8 轮某次构建报错里的具体错误堆栈，而这个报错早就在 Tier 1 折叠中被摘要掉了，怎么办？**

很多系统的第一反应是提供一个“解压（Decompress）”功能，把当年被折叠的几万字原始日志重新灌回 Prompt。

但这种做法直接会导致系统崩溃：

1. 刚刚辛苦收敛腾出来的上下文，瞬间又被几万 Token 撑爆；
2. 已经稳定运行的前缀缓存再次被撕得粉碎；
3. 模型重新陷入海量日志的注意力泥潭。

billion-context 给出的解法非常干脆，极具经典 Unix 哲学的味道：**把解压内容落盘为本地文件（Decompress-to-File）。**

历史回溯机制对比

当 Agent 需要调阅历史细节时，它的标准执行流程是：

1. **导出文件**：调用解压工具，参数指定将 Block 的原始内容写入本地磁盘：

```
decompress({ blockId: "b3", toFile: "/tmp/restore-b3.txt" })
```

2. **按需分片读取**：系统将当年被折叠的原始输出原汁原味地写入本地临时文件，而在主上下文里，Agent 接着调用日常读取代码的 `read` 工具，配合 `offset` 和 `limit`，只读取包含关键报错的 20 行：

```
read({ path: "/tmp/restore-b3.txt", offset: 120, limit: 20 })
```

这一招彻底解开了“细节回溯”与“上下文膨胀”的死结：

- 几万字的冗长日志在磁盘上躺着，**主上下文仅仅增加了 200 个 Token 的精准切片**；
- 正在享受缓存优惠的历史前缀**毫发无损**，缓存命中率纹丝不动；
- 模型却像一位经验丰富的工程师查阅系统日志一样，获得了 100% 精确、未经任何模型二手总结失真的原始第一手现场。

总结：给 Agent 开发者的系统设计反思

跳出具体的工具实现，billion-context 给我们做复杂 Agent 系统架构带来了几条极其深刻的工程启示：

1. **上下文不是倾倒垃圾的水桶，而是极其昂贵的 CPU L1 缓存**： 盲目迷信厂商宣传的百万 Token 窗口，就像试图把全量数据库直接硬塞进 CPU 缓存行一样荒谬。主上下文必须始终保持极高的信噪比。
2. **前缀缓存是一门精密的经济学**： 在追求上下文精简的过程中，永远要把云端服务商的 Prompt Caching 计费模型计入考量。盲目地全局重写是巨额浪费，而基于盈亏平衡（Breakeven Analysis）的有节奏收敛，才是工业级的系统设计。
3. **不要在 Prompt 里重复发明操作系统**： 文件系统、管道、临时目录已经发展了半个世纪，具备极致成熟的索引、分片、寻址与持久化能力。把长文本、海量日志留给文件系统，让模型通过精确的工具去检索和查阅，远比把一切都交给神经网络的前向传播要可靠、低成本得多。

原文 liuzhengdong.babelgo.cn/billion-context/ · 演示版 liuzhengdong.babelgo.cn/billion-context/talk/