ClaudeStation

Token 优化与对话压缩 PRD

Prompt Cache 策略 · 增量式对话压缩 · Compaction API 集成

版本: 2.0　日期: 2026-02-16　状态: 设计完成

1 背景与问题分析

1.1 当前架构的 Token 流向

ClaudeStation 每次 API 调用的输入结构如下：

[1] System Prompt + Project Documents → 稳定前缀，通过 cache_control 缓存

[2] 对话历史 → 每轮都在增长，当前未缓存也未压缩

[3] 当前用户消息 + 附件 → 每轮变化，不可缓存

当前的核心问题是：区域 [2] 随对话轮次线性增长，在 40 轮对话后可达 30K-50K tokens，每一轮都全量重发。这是最大的 token 浪费点。

1.2 前期讨论中的关键纠正

在此前的讨论中，有几个重要的事实需要纠正：

纠正 1：1 小时缓存的写入成本是 2.0x，不是 1.25x

之前的讨论中将 persistent 和 ephemeral 的写入成本混淆。实际官方定价：

TTL	写入成本乘数	读取成本乘数	语法
5 分钟 (默认)	1.25x	0.1x	{"type": "ephemeral"}
1 小时	2.0x	0.1x	{"type": "ephemeral", "ttl": "1h"}

关键发现：1 小时缓存的写入成本是 5 分钟的 1.6 倍（2.0x vs 1.25x）。只有当用户在一小时内会多次超过 5 分钟空闲时，1 小时 TTL 才划算。

纠正 2：心跳保活的收益微乎其微

之前讨论的“每 4 分 50 秒发送心跳”策略，实际成本分析：

- **心跳请求必须发送完整的缓存前缀**，否则无法触发 cache read。这意味着每次心跳需要支付缓存读取费用（10%）+ 最小输出费用。
- **2 次心跳约 20% vs 重建 25%**，只省了 5%。考虑实现复杂度和边缘情况（心跳期间用户发消息的竞态等），不值得实现。

结论：放弃心跳机制。让缓存自然过期，下次对话时自动重建。

纠正 3：Anthopic 已提供官方 Compaction API

2026 年 1 月，Anthopic 发布了 Compaction API（beta: compact-2026-01-12），实现服务端自动对话压缩。这意味着：

- **服务端压缩无额外 token 成本：**API 自动在接近上下文窗口极限时触发压缩，不需要客户端发送额外的压缩请求。
- **但触发时机较晚：**默认在接近极限时才触发，而不是主动优化。对于成本敏感的用户，在对话中期就应该开始压缩。

结论：采用“客户端主动压缩 + Compaction API 兜底”的混合策略。

纠正 4：分层压缩方案的成本误判

之前讨论的三层压缩方案声称“总成本 6,500 vs 12,000，节省 46%”，但这个计算有问题：压缩成本

3,000 tokens 没有考虑输出价格是输入价格的 5 倍（以 Sonnet 为例）。实际上用主模型做压缩的成本远高于预估。正确做法是用 Haiku 做压缩。

2 最优方案设计

2.1 核心设计原则

1. **缓存前缀最大化：**把尽可能多的稳定内容放入缓存前缀，享受 0.1x 读取价格。
2. **未缓存区域最小化：**通过压缩历史对话，减少每轮实际发送的未缓存 token 数。
3. **压缩成本最低化：**用 Haiku（$1/$5）而非主模型做压缩，增量式压缩而非批量压缩。
4. **精度保持最大化：**最近 N 轮对话保留全文，只压缩较老的历史。

2.2 分层上下文架构

每次 API 调用的输入结构分为四层：

层次	内容	缓存状态	变化频率	成本
缓存层 1	System Prompt + 项目文档	cache_control: ephemeral	极少变化	写入 1.25x，读取 0.1x
缓存层 2	对话摘要 (Rolling Summary)	第二个 cache breakpoint	每 ~10 轮更新	写入 1.25x，读取 0.1x
未缓存层	最近 8-10 轮完整对话	无缓存	每轮变化	基础输入价 1.0x
当前消息	用户新消息 + 附件	无缓存	每轮变化	基础输入价 1.0x

这是整个方案的核心创新：将对话摘要作为第二个 cache breakpoint。因为摘要每 10 轮才更新一次，在这 10 轮内它也被缓存读取（只付 0.1x）。而旧方案的 30K+ 历史 tokens 全部以 1.0x 价格发送。

2.3 增量式滚动摘要算法

与之前讨论的"分层渐进压缩"不同，本方案采用更简单高效的增量式滚动摘要：

触发条件：当未压缩的对话轮次超过 N 轮（默认 N=10）时触发压缩。

压缩方式：取最早的 K 轮（默认 K=5）的对话，调用 Haiku 生成摘要，将摘要追加到现有的 Rolling Summary 中。

关键特性：每次只压缩 5 轮（而非全部历史），压缩成本固定且可预测。无论对话多长，每次压缩的输入量始终约 3K tokens。

算法流程：

5. 用户发送第 11 条消息（超过 N=10 的阈值）
6. 取对话历史中第 1-5 轮（最早的 K=5 轮）
7. 调用 Haiku："请将以下对话压缩为简洁摘要，保留关键决策、代码片段、数据点"
8. 将摘要追加到 Rolling Summary，删除已压缩的原始消息
9. 继续正常处理用户的第 11 条消息，此时历史只有 Summary + 5 轮全文
10. 用户发送第 16 条消息时，重复以上过程

3 成本分析模型

3.1 场景假设

以下分析基于典型使用场景：

- 模型：Claude Sonnet 4.5（$3/$15 per MTok）
- 项目文档：50K tokens（系统提示词 + 文档）
- 每轮对话：用户 300 tokens，Claude 回复 600 tokens
- 总计 50 轮对话，发送节奏正常（每轮间隔 < 5 分钟）

3.2 方案对比

指标	方案 A: 当前简单截断	方案 B: 增量摘要+缓存	方案 C: 网页版 Claude.ai
第 50 轮发送量	50K(缓存)+45K(历史)	50K(缓存)+2K(摘要缓存)+9K(近期)	无控制，全由平台决定
缓存读取量	50K @ 0.1x	52K @ 0.1x	无缓存
未缓存量	45K @ 1.0x	9K @ 1.0x	全部 @ 1.0x
第 50 轮输入成本	$0.150	$0.043	无法计算
压缩摆擦成本	无	~$0.005/次	无
50 轮累计总输入成本	~$3.60	~$1.35	约$5.00+
节省比例 (vs 方案 A)	基线	62% 节省	-

方案 B 节省 62% 的关键原因：第 50 轮时，方案 A 要发送 45K 未缓存的历史 tokens，而方案 B 只发送 2K 摘要（缓存的）+ 9K 近期对话。差距随对话轮次增加线性扩大。

3.3 压缩摆擦成本详细分析

每次压缩操作的真实成本（使用 Haiku $1/$5 per MTok）：

- 输入：5 轮对话原文 ~3K tokens + 压缩指令 200 tokens = 3.2K tokens
- 输出：摘要 ~500 tokens
- **单次压缩成本：**3.2K × $1/MTok + 500 × $5/MTok = $0.0032 + $0.0025 = $0.0057
- 每 10 轮触发一次压缩，50 轮共触发 4 次，总压缩成本：4 × $0.0057 = $0.023

压缩摆擦仅占总节省金额的 ~1%。这证明用 Haiku 做压缩是成本效益极高的选择。

3.4 Cache TTL 选择决策树

基于 50K tokens 项目文档，Sonnet 定价：

场景	5 分钟 TTL	1 小时 TTL	推荐
持续对话，从不超过 5 分钟空闲	初始 $0.1875，续期	初始 $0.30，续期	5 分钟

	$0.015/轮	$0.015/轮	(更便宜)
偶尔超过 5 分钟 (1 次/小时)	初始+重建 = $0.375	$0.30	1 小时 (整体更便宜)
经常超过 5 分钟 (3 次/小时)	$0.1875 × 4 = $0.75	$0.30	1 小时 (明显更便宜)

实现建议：默认使用 5 分钟。在 Settings 中提供“1 小时缓存”选项，附说明：“如果你经常中断超过 5 分钟，开启此选项可节省成本”。

4 详细需求规格

4.1 Feature 1: 滚动摘要压缩系统

4.1.1 数据模型

在 conversations 表中新增字段：

字段	类型	说明
rolling_summary	TEXT	当前累积的对话摘要
summary_up_to_msg_id	TEXT	摘要覆盖到哪一条消息的 ID
summary_token_count	INTEGER	摘要的 token 数
compress_after_turns	INTEGER DEFAULT 10	多少轮后触发压缩，用户可配置

4.1.2 压缩提示词模板

发送给 Haiku 的压缩请求：

```
system: "You are a conversation summarizer. Output ONLY the summary, no preamble."
user: "请将以下对话压缩为简洁摘要。规则：\n1. 保留所有关键决策和结论\n2. 保留代码片段、数据点、技术细节\n3. 保留用户偏好和约束条件\n4. 删除客套、闲聊、重复内容\n5. 摘要长度控制在 500 tokens 以内\n\n 现有摘要: {existing_summary}\n\n 新对话内容: {conversation_turns}"
```

4.1.3 触发逻辑伪代码

```
def should_compress(conversation):
    uncompressed = messages_after(conversation.summary_up_to_msg_id)
    return len(uncompressed) > conversation.compress_after_turns * 2

def compress(conversation):
    oldest_5_turns = uncompressed[:10]  # 5 pairs
    summary = call_haiku(COMPRESS_PROMPT, oldest_5_turns, conversation.rolling_summary)
    conversation.rolling_summary = summary
    conversation.summary_up_to_msg_id = oldest_5_turns[-1].id
    # DB: mark compressed messages, don't delete (for export)
```

4.1.4 在 ContextBuilder 中的集成

修改 context_builder.py 的 build() 方法，将摘要作为第二个 cache breakpoint：

```
system_content = [
    {"type": "text", "text": system_prompt + docs, "cache_control": {"type": "ephemeral"}},
    {"type": "text", "text": f"<summary>{rolling_summary}</summary>", "cache_control": {"type": "ephemeral"}}
]
```

注意：Anthropic 允许在一个请求中设置最多 4 个 cache breakpoint。摘要作为第二个 breakpoint，当摘要在连续多轮中不变时，它会被 cache read（只付 0.1x）。一旦摘要更新，仅摘要部分的 cache 需要重建，而第一个 breakpoint（system+docs）的 cache 不受影响。

4.2 Feature 2: Compaction API 兑底集成

4.2.1 作用

客户端压缩在对话中期主动优化成本，但如果用户的单次消息非常长（如粘贴大量代码），仍可能接近上下文极限。Compaction API 作为安全网，在接近 200K 上限时自动压缩，避免 prompt_too_long 错误。

4.2.2 实现方式

在 ClaudeClient.stream_message() 的 API 调用参数中添加：

```
kwargs["betas"] = ["compact-2026-01-12"] kwargs["context_management"] = { "edits": [{"type": "compact_20260112", "trigger": {"type": "input_tokens", "value": 160000}}] }
```

触发阈值设为 160,000（200K 的 80%）。客户端压缩在 ~60K 时就开始工作，所以 Compaction API 通常不会触发。它只在异常情况（如巨大的单次输入）下触发。

4.3 Feature 3: Cache TTL 可配置

4.3.1 配置项

在 Settings 对话框的 General 标签页新增：

- **Cache TTL 下拉框：**“5 分钟 (默认，推荐持续对话)” / “1 小时 (经常中断时更便宜)”
- **压缩阈值滑块：**“压缩触发轮次”，范围 5-30，默认 10

4.3.2 存储

将 TTL 配置存入 projects.settings_json：

```
{"cache_ttl": "5m", "compress_after_turns": 10}
```

4.4 Feature 4: Token 仪表盘增强

4.4.1 Cache 命中率显示

在右侧面板的 Token Stats 区域增加：

- **Cache 命中率：**基于最近 10 次 API 调用的 cache_read_tokens / (cache_read_tokens + input_tokens)。显示为百分比和进度条。
- **压缩节省量：**显示“本次对话已压缩 X 次，节省约 Y tokens”
- **累计节省金额：**对比“如果无缓存/压缩的假设成本” vs “实际成本”

4.4.2 模型切换时的成本提示

当用户通过顶部工具栏切换模型时，在状态栏显示：

```
“切换到 Opus 4.6 — 项目文档 50K tokens，首次调用 cache 写入成本约 $0.31”
```

这是纯信息展示，不弹窗不打断工作流。

5 与网页版 Claude.ai 的系统性对比

网页版 Claude.ai 对话的核心缺点是：用户无法控制 token 经济性。API 使用则完全透明可控。

能力	Claude.ai 网页版	ClaudeStation 当前版	ClaudeStation 优化后 (v2)
Prompt Caching	无（每轮全量重新处理）	单层 ephemeral 5min	双层 breakpoint + TTL 可配
对话压缩	自动 (95%时触发)	无，仅截断	增量式 Haiku 压缩 + Compaction 兜底
压缩时机	被动（快爆时才触发）	无	主动（每 10 轮即压缩）
压缩成本	无额外成本（含在订阅中）	-	~$0.006/次 (Haiku)
压缩控制	无控制	-	可配置触发阈值
Token 透明度	无（看不到用量）	每条消息显示	每条 + 累计 + 节省金额 + 命中率
模型灵活性	可切换，但无成本提示	可切换	切换时显示 cache 重建成本
文档缓存	需重复上传（Projects 除外）	本地存储+cache	本地存储+双层 cache
50 轮对话估算成本	~$5.00+ (按 API 等价计算)	~$3.60	~$1.35

ClaudeStation v2 相比网页版节省约 73% 的 token 成本，核心优势来自：(1) Prompt Cache 使稳定前缀只付 10%；(2) 增量压缩使历史体积保持在低位；(3) 摘要也被缓存，进一步降低读取成本。

6 实现优先级与路线图

优先级	功能	工作量	节省影响
P0 (必做)	滚动摘要压缩系统	3-4 天	最大：降低每轮历史体积 80%+
P0 (必做)	摘要作为第二 cache breakpoint	0.5 天	进一吹降低摘要读取成本 90%
P1 (重要)	Cache TTL 可配置	1 天	中等：避免不必要的 cache 重建
P1 (重要)	Cache 命中率仪表盘	1 天	间接：帮助用户优化使用习惯
P2 (改进)	Compaction API 兜底	0.5 天	安全网：避免极端情况报错
P2 (改进)	模型切换成本提示	0.5 天	信息性：帮助用户做决策
P2 (改进)	compress_after_turns UI 配置	0.5 天	次要：灵活性

6.1 不做与可做的事项

- **不做心跳保活**：收益微薄（5%），实现复杂度高，边缘情况多。
- **不做 ModelRecommender**：基于关键词的规则引擎不可靠，且会让用户感觉工具在替自己做决定。
- **不做发送前确认弹窗**：高频使用场景下每次发消息都弹窗是灾难性的 UX。改为状态栏静默显示。
- **是否做 Embedding 检索**：需要 PyTorch 等重依赖（200MB+），可以做，只要能节省 tokens，信息更精确。
- **可以做 Cache 状态预测表**：本地时钟预测服务端 TTL 天然不精确，真正可靠的信号是 API 响应中的 cache_read_input_tokens 是否 > 0。

7 关键实现细节

7.1 压缩质量保障

压缩的核心风险是信息丢失。以下措施确保质量：

11. **压缩提示词明确保留优先级：**关键决策 > 代码片段 > 数据点 > 用户偏好 > 闲聊内容。
12. **原始消息不删除：**数据库中保留全部原始消息，仅在构建 API 请求时使用摘要替代。导出时可输出完整历史。
13. **摘要长度控制：**每次压缩产生的摘要不超过 500 tokens，累积 5 次压缩后摘要约 2000-2500 tokens——这远小于它所替代的 25 轮原始对话（~15K tokens）。
14. **摘要增长监控：**如果 Rolling Summary 超过 3000 tokens，对摘要本身再做一次压缩（“摘要的摘要”）。

7.2 边界情况处理

场景	处理方式
对话不足 10 轮	不触发压缩，全文发送
压缩调用 Haiku 失败	记录日志，跳过压缩，以全文继续发送
用户修改了系统提示词	第一个 cache breakpoint 失效重建，第二个不受影响
用户上传/删除文档	同上，第一个 breakpoint 重建
用户切换模型	所有 cache 失效重建，状态栏显示重建成本
摘要更新后 cache 变化	仅第二个 breakpoint 重建，第一个不受影响
Compaction API 触发	记录日志告警，状态栏显示“服务端已自动压缩”

7.3 可配置参数汇总

参数	默认值	范围	位置
COMPRESS_AFTER_TURNS	10	5-30	config.py + Settings UI
COMPRESS_BATCH_SIZE	5	3-10	config.py+ Settings UI
MAX_SUMMARY_TOKENS	500	200-1000	config.py+ Settings UI
SUMMARY_RECOMPRESS_THRESHOLD	3000	2000-5000	config.py+ Settings UI
CACHE_TTL	"5m"	"5m" / "1h"	Settings UI+ Settings UI
COMPACTION_TRIGGER	160000	100000-180000	config.py+ Settings UI
RECENT_TURNS_KEPT	10	5-20	config.py+ Settings UI

8 验收标准

15. **成本验证**：在 50 轮对话场景下，累计输入 token 成本对比无压缩方案节省 50%+。
16. **Cache 命中率**：在正常对话节奏下，cache_read_input_tokens 占总缓存前缀 token 的 90%+。
17. **压缩透明性**：压缩对用户不可见（无弹窗、无延迟感），仅在状态栏显示“已压缩”。
18. **对话质量**：压缩后 Claude 仍能回忆早期对话中的关键决策和技术细节。
19. **导出完整性**：Markdown 导出包含所有原始消息（非压缩版）。
20. **容错性**：压缩失败时不影响主流程，自动回退到全文发送。