从 LLM 到 Coding Agent

一次 LLM API 调用只是个无状态的文本函数；能自主改代码、跑命令、修 bug 的 Coding Agent，是在它之上层层搭起来的工程。本专栏逐个拆解这中间到底发生了什么。

全 21 篇 · 作者 David

序 · LLM 只是块芯片，Agent 才是整台机器

现在人人都在用 Coding Agent：你说一句“帮我把这个 bug 修了”，它自己去读文件、搜代码、改几处、跑测试、看报错、再改，最后告诉你搞定了。整个过程像有个真人在你的终端里干活。

但如果你去看它底层依赖的东西——一个 LLM 的 API——你会发现那玩意儿“什么都不会”：

- 它无状态：每次调用都是一张白纸，不记得上一句你说过什么。
- 它没有记忆：关掉这次请求，所有上下文烟消云散。
- 它不能做任何事：它只会输出文本。它读不了你的文件，跑不了命令，改不了代码。它连“现在几点”都不知道。
- 它甚至不知道自己该停在哪：给它一段话，它就续写一段话，仅此而已。

于是本专栏的核心问题只有一个：

> *“一个只会“输出文本”的无状态函数，是怎么变成一个能自主完成复杂编程任务的 Agent 的？这中间，工程上到底做了哪些事？”*

一块芯片和一台机器的距离

打个比方：LLM 就像一块 CPU。CPU 很强，但一块裸芯片什么也干不了——你没法拿一块 CPU 去打字、上网、写代码。要让它变成一台能用的电脑，你还得给它配上内存、总线、硬盘、操作系统、驱动、输入输出……

Coding Agent 之于 LLM，正是整台机器之于那块芯片。模型提供“智能”，但把智能变成“能自主干活的系统”，靠的是外面这一整圈工程：

- 怎么让它能调用工具(读文件、跑命令)?
- 怎么让它自主循环，而不是问一句答一句?
- 怎么喂给它恰到好处的上下文，又不撑爆那有限的上下文窗口?
- 怎么在它跑飞、报错、被打断时优雅恢复?
- 怎么控制权限，不让它把你的机器搞坏?
- 怎么省钱省延迟，不让每次对话烧掉一杯咖啡的钱?

这些问题，每一个都不在“模型”里，而在“模型外面那一圈”。这一圈，就是本专栏要逐层拆开的东西。

> 本专栏的方法论：一篇文章只解决一个问题。每篇都按「这个问题的背景 → 解决方案的思路 → 落到代码的实现细节」三段式展开，力求把每个技术点揉碎讲透。我们会从“一次最朴素的 API 调用”出发，每加一篇，就往这块裸芯片上焊一个部件，直到最后拼出一台完整的机器。

路线图

专栏按“从裸 API 逐步长成完整 Agent”的顺序推进，大致分七个阶段：

第一阶段 · 起点 —— 先看清我们手里到底有什么。

- 一次 LLM API 调用到底给了你什么，又没给你什么。

第二阶段 · 让模型能做事 —— 从“输出文本”到“能操作世界”。

- 工具调用(function calling)：让模型开口说“我要调用哪个工具”。
- Agent Loop：把一次性问答变成一场自主循环——这是“Agent”这个词真正的分水岭。
- 工具系统的抽象设计：一个可扩展、类型安全、自带权限的工具到底长什么样。

第三阶段 · 流式与并发 —— 让它快起来、并行起来。

- 流式处理：边生成边解析，以及增量到达的工具参数。
- 边流边执行 + 并发编排：工具块一到就派发，安全的工具并行跑。

第四阶段 · 安全与控制 —— 别让它把你机器搞坏，也别让它跑飞。

- 权限系统：allow / deny / ask，权限模式，危险操作的闸门。
- 中断与转向：如何在流式中途叫停，并保持消息历史的合法性。
- Hooks：在工具调用前后、回合结束时插入你自己的逻辑。

第五阶段 · 上下文工程 —— 整个 Agent 最难、最见功力的部分。

- Token 预算：与有限上下文窗口的持久战。

- 系统提示与上下文注入：该塞什么、在哪塞、怎么塞。
- Prompt 缓存：省钱省延迟的命脉，以及“字节稳定性”这条隐形红线。
- 自动压缩：上下文快满了，怎么把历史“折叠”成摘要还不丢关键信息。
- 精细化上下文管理：按需删除巨型工具结果、外置存储、内容替换。

第六阶段 · 健壮性 —— 生产环境里，出错是常态。

- 错误恢复与模型回退：重试、退避、降级换模型、孤儿消息清理、各类限额恢复。
- 成本与用量追踪：把 token 换算成钱，并实时盯住。
- 思考(extended thinking)的处理：签名、模型绑定，以及那些能让你调试一整天的坑。

第七阶段 · 扩展与规模 —— 一个 Agent 打不过，就上一群。

- 子 Agent：任务委派与上下文隔离，让主线程不被细节淹没。
- MCP 与工具的懒加载：接入外部工具生态，以及几百个工具怎么按需加载。
- 会话历史与持久化：JSONL 落盘、断点续接、撤销。

> ⚠️ 一句前置声明：本专栏聚焦通用的 Agent 工程技术——这些模式适用于任何基于 LLM 构建的 Agent。文中所有代码均为讲解而写的独立示例，用来演示技术要点本身，你可以照着在任何 LLM API 上复刻。我们只谈技术，不谈具体产品。

配套代码：一个从零搭起的最小 Agent

本专栏配一个可运行的开源仓库 👉 github.com/davidapp/coding-agent-from-scratch。它不是一堆零散片段，而是一个随专栏逐篇长大的真实 Agent：每读完一篇，仓库就演进一步，加上那一篇讲的部件。你可以 `git checkout` 到任意一篇对应的版本，直接把那个阶段的 Agent 跑起来。

读到第二阶段结束，这个仓库已经是一个真能自主读写文件、跑命令、改代码的最小 Coding Agent 了——总共不过几百行。文章讲“为什么这么设计”，代码给“具体怎么落地”，两边对照着看。

读之前，你最好有

- 用过至少一次某个 LLM 的 API(知道 `messages`、`system`、`stream` 大概是什么)。
- 会读 TypeScript(示例用 TS 写，但讲的是思想，换成 Python/Go 一样成立)。
- 对“Agent 到底是不是黑魔法”心存好奇——读完你会发现，它一点也不魔法，全是扎实的工程。

准备好了，我们从最朴素的起点开始：一次 LLM API 调用，到底给了你什么。

01 · 一次 LLM API 调用，到底给了你什么

要理解 Agent 在 LLM 之上加了什么，得先精确地知道 LLM 本身给了什么。这一篇不聊 Agent，只把“裸 API”这块地基看到底——你会发现它朴素得惊人。

一、最朴素的一次调用

抛开所有花哨的封装，一次 LLM API 调用的本质就是一个 HTTP 请求。你发一段对话，它回一段文本：

一次最朴素的调用

```ts
const response = await client.messages.create({
  model: 'some-llm',
  max_tokens: 1024,
  system: '你是一个乐于助人的助手。',          // 系统提示：定调色、立规矩
  messages: [
    { role: 'user', content: '用一句话解释什么是尾递归。' },
  ],
})

console.log(response.content[0].text)
// => "尾递归是指函数的最后一步操作就是调用自身，从而可被编译器优化为循环、不增长调用栈。"
```

就这么点东西。请求里有三样关键的：

- `system`：系统提示。它不属于对话，而是给整场对话“定调”的元指令——角色、风格、规则都写在这。
- `messages`：对话历史，一个 `{ role, content }` 数组，`role` 在 `user` / `assistant` 之间交替。
- `max_tokens`：这次最多让它生成多少 token。

二、响应里藏着的三个关键信号

响应不只有文本。真正搭 Agent 时，你盯的往往是另外几个字段：

响应的完整形状(简化)

```ts
{
  role: 'assistant',
  content: [
    { type: 'text', text: '尾递归是指……' }    // 内容是一个"块数组",不只是字符串
  ],
  stop_reason: 'end_turn',                     // ← 它为什么停下来
  usage: {
    input_tokens: 24,                          // ← 这次读进去多少
    output_tokens: 38,                         // ← 这次吐出来多少
  },
}
```

三个信号必须刻进脑子，后面每一篇都要用到：

1. `content` 是“块数组”，不是字符串。 一条回复可以由多个“内容块”组成。现在只有一个 `text` 块；等到下一篇讲工具调用，你会看到 `tool_use` 块和它并肩出现。这个“多块”结构，是整个 Agent 大厦的承重墙。

2. `stop_reason` 告诉你它为什么停。 常见取值：

   - `end_turn`：它自然说完了。
   - `max_tokens`：撞到你设的 `max_tokens` 上限被硬截断(话没说完！)。
   - `stop_sequence`：命中了你指定的停止串。
   - `tool_use`：它想调用工具了(下一篇的主角)。

   Agent 循环的走向，很大程度就是被 `stop_reason` 驱动的——它是“该继续还是该收手”的路标。

3. `usage` 是你的账单和油表。 每次调用花了多少 token，直接换算成钱和离上下文窗口爆满还有多远。成本追踪和上下文管理这两大主题，全建立在死死盯住 `usage` 之上。

三、流式：别让用户干等

上面那种“等它全部生成完再返回”的方式，在交互式场景里是灾难——模型吐一大段要好几秒，用户对着空屏幕干等。所以真实 Agent 几乎总是流式调用：

流式:token 一边生成一边到达

```ts
const stream = await client.messages.create({
  model: 'some-llm',
  max_tokens: 1024,
  messages: [{ role: 'user', content: '写一首关于递归的短诗。' }],
  stream: true,                                    // ← 开关在这
})

for await (const event of stream) {
  if (event.type === 'content_block_delta') {
    process.stdout.write(event.delta.text)         // 每来一小片就打印一小片
  }
}
```

流式把“一次响应”拆成一串事件：块开始、块内容增量(delta)、块结束、消息结束……你把这些增量拼起来，就还原出完整回复。这里先埋个伏笔：流式不仅是为了 UX——它还让 Agent 能在工具参数刚流完的一瞬间就抢跑执行，这个我们放到流式那一篇细讲。

四、决定性的三个“它不给你”

好，以上就是 LLM API 给你的全部。现在看它不给你什么——这三个“没有”，定义了后面整个专栏的工作量。

没有 ①：无状态

API 是纯函数。同样的 `messages` 进去，(在贪婪解码下)同样的东西出来。它不记得任何事——服务器端不给你存对话。

那我们平时用的 AI 对话，那种“它记得我上一句”的连续感是哪来的？是客户端假装出来的。 每一轮，你都得把到目前为止的全部对话历史重新塞进 `messages` 再发一次：

所谓

```ts
const history: Message[] = []

async function chat(userText: string) {
  history.push({ role: 'user', content: userText })        // 追加用户这句
  const res = await client.messages.create({
    model: 'some-llm',
    max_tokens: 1024,
    messages: history,                                     // ← 每次都发"全部"历
  })
  history.push({ role: 'assistant', content: res.content }) // 把回复也追加进去
  return res
}
```

> “对话”是一种客户端维护的幻觉。 模型这一侧永远只是“读一坨历史 → 续写下一条”。这个事实的直接后果是：历史越滚越长，每次请求就越贵、越慢，直到撞上上下文窗口的天花板。这就

是为什么上下文工程会是整个 Agent 里最硬核的一章——你得亲手管理这坨不断膨胀的历史。

没有 ②：不能行动

模型只会输出文本。你问它“当前目录有哪些文件”，它不能真的去 `ls`——它只能猜，或者老实说“我看不到你的文件系统”。

你甚至可以让它输出一句 `ls -la`，但那也只是一串字符，不会有任何命令被执行。模型和真实世界之间，隔着一道它无法逾越的墙。捅穿这道墙，让模型的“意图”变成真实世界里的“动作”，就是工具调用要解决的事，也是“Agent”能成立的第一块基石。

没有 ③：不会自己推进

给模型一段话，它生成一段回复，然后停下。它不会自己接着“那我下一步该干嘛”。它没有“循环”，没有“目标驱动”，没有“不达目的不罢休”。

而一个 Agent 修 bug 时，是“改一处→跑测试→看报错→再改”这样一轮一轮自主推进的。这个“自主推进”从哪来？不在模型里。它是外面那个 Agent Loop 手动驱动出来的——那一篇会告诉你，所谓 Agent，内核其实就是一个 `while` 循环。

小结：地基与蓝图

把这一篇钉死成一句话：

> *“裸 LLM API = 一个无状态的、只会输出文本的、问一句答一句的纯函数。”*

而一个 Coding Agent，就是围绕这个纯函数，亲手补上它缺的三样东西：

它没有	我们要加的东西	对应篇章
不能行动	工具调用	02 工具调用
不会自己推进	Agent Loop	03 Agent Loop
无状态、会撑爆	上下文工程	第五阶段

地基看完了。下一篇，我们捅穿那道“模型不能行动”的墙——让它第一次真正地做一件事。

02 • 工具调用：让模型的“意图”变成真实的“动作”

上一篇我们确认了一个残酷的事实：模型只会输出文本，碰不到真实世界。这一篇，我们捅穿这道墙。

一、背景：模型和世界之间的那道墙

你问模型“`src/` 目录下有哪些文件”，它有三种反应，没有一种是有用的：

1. 老实承认：“我无法访问你的文件系统。”(诚实但没用)
2. 一本正经地胡编：“有 `index.ts`、`utils.ts`……”(自信地瞎猜，更糟)
3. 输出一句 `ls src/`，但那只是一串字符，不会有任何命令真的执行。

问题的根子在于：模型的输出是文本，而“列目录”是一个动作。文本和动作之间，隔着一道模型无法逾越的墙。

我们需要一种机制，让模型能表达“我想执行某个动作、参数是这些”，然后由我们(宿主程序)去真正执行，再把结果告诉它。这个机制，就是工具调用(tool use / function calling)。

二、解决方案：把“能力”声明给模型

思路分三步，构成一个闭环：

1. 声明：在请求里告诉模型“你有哪些工具可用”，每个工具用名字 + 描述 + 参数的 JSON Schema 来定义。
2. 请求：模型不再(只)输出文本，而是输出一个结构化的工具调用请求——“我要调用 `read_file`，参数 `{path: 'src/index.ts'}`”。
3. 回填：你执行这个调用，把结果作为一条特殊消息塞回对话历史，模型接着往下推理。

关键的认知在这里：

> ⚠ 模型永远不会执行任何东西。它做的全部，只是输出一个 JSON，表达“我想这样调用”。真正的执行——读文件、跑命令、发请求——100% 发生在你的代码里。这道边界是整个 Agent 安全模型的基石：模型再怎么“想”删你的库，它也只能“请求”，而按不按下执行键，权力始终在宿主手里。权限系统整篇文章，就是在这道边界上做文章。

三、实现细节：一个完整的工具调用闭环

我们来手动走完一整轮。用 TypeScript，定义一个“读文件”工具。

第 1 步：声明工具

工具的核心是一份给模型看的说明书：名字、描述、参数 Schema。

声明一个 read_file 工具

```ts
const readFileTool = {
  name: 'read_file',
  // 描述是写给模型看的 prompt!模型靠它决定"什么时候、怎么用"这个工具。
  description: '读取指定路径的文件内容。当你需要查看某个文件时使用。',
  input_schema: {
    type: 'object',
    properties: {
      path: { type: 'string', description: '要读取的文件路径,相对于项目根目录' },
    },
    required: ['path'],
  },
} as const
```

`description` 不是注释，是 prompt。它和参数 Schema 会被拼进发给模型的请求里，直接决定模型能不能正确地选中并调用这个工具。工具描述写得烂，模型就用得烂——这是 Agent 工程里一个极其现实、又极其容易被低估的细节。工具抽象那一篇会专门讲怎么设计好一个工具。

第 2 步：模型发出调用请求

把工具随请求一起发出去：

带着工具发起请求

```ts
const res = await client.messages.create({
  model: 'some-llm',
  max_tokens: 1024,
  tools: [readFileTool],                          // ← 告诉模型它有这个工具
  messages: [
    { role: 'user', content: 'src/index.ts 里导出了什么?' },
  ],
})
```

这一次，模型的响应变了——注意 `stop_reason` 和 `content` 里的新块：

模型的响应:它要调用工具了

```
{
  role: 'assistant',
  content: [
    { type: 'text', text: '我来看一下这个文件。' },    // 可能先说一句
    {
      type: 'tool_use',                                  // ← 全新的块类型!
      id: 'toolu_abc123',                                // ← 这次调用的唯一 ID,关键
      name: 'read_file',
      input: { path: 'src/index.ts' },                   // ← 模型填好的参数
    },
  ],
  stop_reason: 'tool_use',                               // ← 不是 end_turn,是"我要
}
```

还记得上一篇说的“`content` 是块数组”这堵承重墙吗？现在你看到它的价值了：模型能在同一条回复里，既说一句人话(`text` 块)，又发起一个动作(`tool_use` 块)。

第 3 步：你执行，并把结果喂回去

模型只是“请求”，执行是你的活。你按 `name` 找到对应实现，拿 `input` 当参数跑它：

宿主程序执行工具

```
async function executeTool(name: string, input: any): Promise<string> {
  if (name === 'read_file') {
    return await fs.readFile(input.path, 'utf-8')    // ← 真正的动作发生在这里
  }
  throw new Error(`未知工具: ${name}`)
}
```

执行完，把结果包成一个 `tool_result` 块，作为一条 `user` 角色的消息追加进历史。最关键的一步：用 `tool_use_id` 把结果和刚才那次请求对上号。

TS 把工具结果回填进对话

```ts
const toolUse = res.content.find(b => b.type === 'tool_use')
const result = await executeTool(toolUse.name, toolUse.input)

const messages = [
  { role: 'user', content: 'src/index.ts 里导出了什么?' },
  { role: 'assistant', content: res.content },          // 模型那条(含 tool_use)原
  {
    role: 'user',                                       // 工具结果的角色是 user
    content: [
      {
        type: 'tool_result',
        tool_use_id: toolUse.id,                        // ← 必须回指那个 id,——
        content: result,                                // 文件内容
      },
    ],
  },
]
```

第 4 步：模型带着结果继续

把这份加长了的历史再发一次，模型这回“看得见”文件内容了，于是给出真正的回答：

TS 再发一次,模型基于工具结果作答

```ts
const finalRes = await client.messages.create({
  model: 'some-llm', max_tokens: 1024, tools: [readFileTool], messages,
})
// => "src/index.ts 导出了 3 个函数:createServer、handleRequest 和 shutdown。"
```

闭环完成。模型第一次真正地“知道”了一件它本来无从得知的事——因为你替它去看了。

四、几条不能踩的硬规则

工具调用的协议有几条铁律，违反了 API 直接报错：

- 每个 `tool_use` 必须有一个配对的 `tool_result`。 模型发起了 3 个调用，你就欠它 3 个结果，靠 `tool_use_id` 一一对上。缺一个，下次请求就会被拒。这条约束会在后面反复咬你——比如流式中途被用户打断时，已经发出的 `tool_use` 还没来得及执行，你也必须给它补一个(内容为“已中断”的) `tool_result`，否则历史就是非法的。中断那一篇专门讲这个。
- 可以并行调用。 一条响应里可以同时冒出多个 `tool_use` 块(比如“同时读三个文件”)。这为并发执行埋下了伏笔。
- `tool_result` 可以标记为错误。 工具跑失败了，就在结果里带上 `is_error: true`，模型会看到失败并尝试换个方式——这是 Agent 能“自我纠错”的前提。
- `input` 是模型生成的，不可全信。 它是模型按你的 Schema“填”出来的 JSON，可能缺字段、类型不对、路径越界。执行前必须校验。永远把模型的输出当作不可信输入对待。

五、我们刚刚跨过了一道分水岭

退一步看看刚才发生了什么：模型通过一个纯文本的 API，间接地操作了真实世界——它让一个文件被读取了。这就是“Agent”能成立的第一原理：

> *“模型负责“决策”(该调哪个工具、传什么参数)，宿主负责“执行”(真的去干)，结果再回流给模型形成闭环。”*

但你也注意到了：上面这一整套，我们是手动走完一轮的——手动检查 `stop_reason`、手动执行、手动回填、手动再发一次。如果模型读完 `index.ts` 发现还得再看 `utils.ts` 呢？再手动来一轮？那要是它需要连着调用 20 次工具才能修好一个 bug 呢？

你肯定想到了：把这套“检查→执行→回填→再问”用一个循环自动跑起来不就行了。

没错。那个循环，就是让“工具调用”质变成“Agent”的东西。下一篇，Agent Loop：我们会看到，所谓智能体的内核，其实就是一个 `while` 循环。

03 • Agent Loop：所谓智能体，内核是一个 while 循环

这是整个专栏最关键的一篇。上一篇我们手动走完了一轮工具调用。现在我们把“手动”变成“自动”——而这一步质变，就是“Agent”这个词的全部含义。

一、背景：一轮不够，远远不够

回想真人修一个 bug 的过程：

> *“读一眼报错 → 打开相关文件看看 → 搜一下这个函数在哪调用 → 改一处 → 跑测试 → 又报另一个错 → 再改 → 再跑 → 绿了，收工。”*

这是十几轮“观察→行动→再观察”。而上一篇的手动流程只走了一轮就停了。你总不能让用户在每一轮之间都手动点一下“继续”。

我们需要的是：让模型自主地一轮一轮推进，直到它认为任务完成为止。 中间不需要人插手。

二、解决方案：一个循环

思路简单到近乎失望：把“调用模型 → 执行工具 → 回填结果”这套，套进一个 `while` 循环，直到模型不再要求调用工具。

判定“任务完成”的信号，恰恰是上一篇埋下的那个：模型的回复里如果还有 `tool_use` 块，说明它还想干活，继续；如果一个 `tool_use` 都没有(纯 `text`)，说明它话说完了，退出。

最小可用的 Agent Loop —— 真的就这么点

```ts
async function runAgent(userInput: string, tools: Tool[]) {
  const messages: Message[] = [{ role: 'user', content: userInput }]

  while (true) {
    // 1) 带着完整历史和工具,问模型
    const res = await client.messages.create({
      model: 'some-llm', max_tokens: 4096, tools, messages,
    })
    messages.push({ role: 'assistant', content: res.content })

    // 2) 挑出这一轮所有的工具调用请求
    const toolUses = res.content.filter(b => b.type === 'tool_use')

    // 3) 没有工具调用 → 模型说完了 → 退出循环
    if (toolUses.length === 0) {
      return res   // 任务完成
    }

    // 4) 执行每个工具,把结果作为一条 user 消息塞回去
    const toolResults = await Promise.all(
      toolUses.map(async (tu) => ({
        type: 'tool_result',
        tool_use_id: tu.id,
        content: await executeTool(tu.name, tu.input),
      })),
    )
    messages.push({ role: 'user', content: toolResults })

    // 5) 回到循环开头,模型带着工具结果继续推进
  }
}
```

就这些。一个能自主读文件、跑命令、改代码、看结果、再决定下一步的 Agent，内核就是这段循环。你把它跑起来，输入“修复登录页的报错”，它就会自己读文件、搜代码、改、跑测试、看报错、再改……一轮一轮，直到它认为好了才停。

> 这就是“Agent”的定义式：`Agent = LLM + Tools + Loop`。模型提供每一步的“决策”，工具提供“行动能力”，而这个循环提供**“自主性”**——让离散的一步步，连成一条不需要人插手的轨迹。市面上所有 Coding Agent，内核都是这个循环；剩下的一切(后面十几篇讲的)，都是围绕它做的加固和优化。

三、从“玩具循环”到“生产循环”

上面那 30 行能跑，但离生产级还差得远。一个真实的 Agent Loop，要在这具骨架上处理一大堆现实。我们逐个看——这几乎就是本专栏后续所有篇章的索引。

① 失控保护：它可能永远停不下来

模型可能陷入“改→测→改→测”的死循环，或者被一个搞不定的错误反复折磨。裸 `while (true)` 会让它无限烧钱。生产循环必须有闸门：

给循环装上刹车

```ts
let turns = 0
while (true) {
  if (turns++ >= maxTurns) return { reason: 'max_turns' }         // 回合数上限
  if (tokenSpent > tokenBudget) return { reason: 'budget' }       // token 预算上
  // ...原来的循环体...
}
```

回合数、token 预算、墙上时钟……任意一个触顶都要能干净地终止。“让它自主”和“别让它跑飞”是一对必须同时满足的矛盾，Token 预算那一篇会展开。

② 循环必须“边跑边产出”，而不是跑完才返回

上面的 `runAgent` 要等整个任务结束才 `return`。但用户想实时看到每一步：模型说的每句话、每次工具调用、每个结果，都该立刻显示在终端里，而不是干等两分钟看一坨最终结果。

所以真实的 Agent Loop 几乎总是一个 async generator——它一边推进，一边 `yield` 出中间产物：

生产循环是一个 async generator

```ts
async function* runAgent(userInput: string, tools: Tool[]): AsyncGenerator<Even
  const messages: Message[] = [{ role: 'user', content: userInput }]

  while (true) {
    // 流式调用,模型每吐一块就 yield 出去给 UI
    let toolUses: ToolUse[] = []
    for await (const chunk of streamModel(messages, tools)) {
      yield chunk                                    // ← UI 立刻渲染
      if (chunk.type === 'assistant') {
        messages.push(chunk)
        toolUses.push(...chunk.content.filter(b => b.type === 'tool_use'))
      }
    }

    if (toolUses.length === 0) return { reason: 'completed' }

    for await (const result of runToolsStreaming(toolUses)) {
      yield result                                   // ← 工具结果也实时 yield
      messages.push(toResultMessage(result))
    }
  }
}
```

调用方 `for await (const ev of runAgent(...))` 就能实时消费。这个“生成器 + yield”的结构，是把流式、中断、UI 渲染全都串起来的骨架。

③ 跨轮状态：循环不是无记忆的

玩具循环里只有一个 `messages` 数组在变。生产循环在每轮之间要携带一大堆状态：

- 当前用的是哪个模型(可能中途降级回退)。
- 上下文是否已经压缩过、压缩的边界在哪。
- 各种恢复计数器(输出被截断重试了几次、压缩失败了几次)。
- 本轮 token 花销、预算续期次数。

这些状态在迭代之间流转，决定下一轮的行为。一个成熟的循环，本质是一个手写的状态机——每一轮根据“上一轮为什么继续”来决定这一轮怎么走。

④ 每一轮开始前，先“整理上下文”

注意一个关键时机：在每次调用模型之前，生产循环会先对 `messages` 做一轮加工——检查要不要压缩、要不要给巨型工具结果瘦身、系统提示和各种上下文怎么拼接。也就是说：

> *“循环体的第一件事不是“调用模型”，而是“把这一轮要喂给模型的上下文准备好”。”*

这个“准备上下文”的环节，复杂度往往超过循环本身。它就是第五阶段“上下文工程”的主战场。

⑤ 出错是常态，不是意外

真实世界里，`streamModel` 那一行会以各种方式失败：限流、超时、上下文超长(413)、输出被 `max_tokens` 截断、模型过载需要降级……生产循环把一大半代码花在这些恢复路径上：捕获错误 → 判断能不能恢复 → 能就调整状态后 `continue` 重来，不能就干净退出。健壮性那一篇会专门解剖这些恢复逻辑。

> ⚠ 一个反直觉但极其重要的事实：在成熟 Agent 的循环里，处理“正常路径”的代码可能只占两三成，剩下七八成全是在处理压缩、恢复、中断、预算、降级这些“边缘情况”。让 Agent“能跑”是那 30 行的事；让它在真实世界里稳定地跑，才是真正的工程量所在。这也是为什么本专栏后面还有那么多篇——它们全是在给这个循环“上保险”。

四、小结：一台机器的引擎

回到序章那个比方：如果 LLM 是 CPU，那么 Agent Loop 就是让 CPU 开始自主取指、执行、循环的那个时钟信号。没有它，你只有一块能算但不会自己动的芯片；有了它，机器才真正“跑”起来。

把三篇连起来，我们已经拼出了 Agent 最小的可用内核：

篇章	贡献了什么
01 裸 API	一个会决策的大脑(但不能动、不记事)
02 工具调用	一双能操作世界的手
03 Agent Loop	让大脑和手不停协作的引擎

内核有了。但你可能注意到，前面反复出现一个模糊的 `Tool` 类型和 `executeTool` 函数——真实系统里，一个工具远不止“名字 + 一个执行函数”这么简单。它要自带参数校验、权限检查、并发标记、结果渲染……下一篇，工具系统的抽象设计，我们就来把这个 `Tool` 拆开，看一个工业级的工具到底该长什么样。

04 • 工具系统的抽象设计：一个工业级工具的解剖

上一篇的循环里，反复出现一个含糊的 `executeTool(name, input)`。它能演示原理，但真要撑起一个 Coding Agent，一个工具远不止“一个执行函数”。这一篇，我们把 `Tool` 彻底拆开。

一、背景：玩具 `executeTool` 撑不起真实系统

回想一下，一个真实的工具(比如“编辑文件”)在被调用时，系统需要知道的远不止“怎么执行”：

- 模型传来的参数合法吗？(路径存在吗？字段齐吗？)
- 这个操作要不要用户批准？(改文件危险，读文件不危险)
- 它能和别的工具并行跑吗？(读可以并行，写不能乱序)
- 执行完，给模型看什么？(结构化、省 token 的文本)
- 又给用户看什么？(带高亮的漂亮 diff)
- 结果太大了怎么办？(几万行输出不能直接塞进上下文)

一个 `(name, input) => string` 的函数签名，承载不了这些。我们需要把工具设计成一个富接口：执行只是它的一个方法，围绕执行，还有一整圈元数据和钩子。

二、解剖：一个工业级 Tool 接口

下面是把生产级工具接口提炼出来的样子(用 TS 表达，去掉了 UI 框架细节)。别被长度吓到，我们分组来看：

TS 一个工业级工具的核心契约

```ts
interface Tool<Input, Output> {
  // ── 身份 ──
  name: string
  // 描述是异步、动态的:可以根据当前上下文(权限模式、可用工具)变化
  description(input: Input, ctx: Context): Promise<string>
  inputSchema: ZodSchema<Input>          // 用 zod 定义,再自动转成 JSON Schema 给模

  // ── 执行 ──
  call(input: Input, ctx: Context): AsyncGenerator<Progress, Output>

  // ── 闸门:执行前的两道关 ──
  validateInput(input: Input, ctx: Context): Promise<ValidationResult>
  checkPermissions(input: Input, ctx: Context): Promise<PermissionResult>

  // ── 元数据:描述这个工具"是什么性质" ──
  isReadOnly(input: Input): boolean           // 只读?还是会改动世界?
  isConcurrencySafe(input: Input): boolean    // 能和别的工具并行跑吗?
  isDestructive(input: Input): boolean        // 不可逆操作?(删除、覆盖、发送)
  maxResultSizeChars: number                  // 结果超过多大就外置存盘?

  // ── 两个受众:同一次执行,两套渲染 ──
  mapResultToModelContent(out: Output): ToolResultBlock  // 给模型看的
  renderResultForUI(out: Output): UINode                 // 给人看的
}
```

这个接口里藏着两个决定性的设计思想，值得各花一节讲透。

三、决定性设计之一：元数据驱动编排

注意 `isReadOnly` / `isConcurrencySafe` / `isDestructive` 这几个方法——它们不执行任何操作，只是声明这个工具的“性质”。为什么要把这些性质做成一等公民？

因为循环和编排器要靠它们来做决策：

TS 上层逻辑读取工具的元数据来决策

```ts
// 编排器:能并行的工具一起跑,不能的排队
const [parallel, sequential] = partition(toolUses, tu =>
  findTool(tu.name).isConcurrencySafe(tu.input),
)
await Promise.all(parallel.map(runTool))   // 并发安全的,一起上
for (const tu of sequential) await runTool(tu)  // 其余的,老实排队

// 权限层:只读操作可以免批,写操作要过闸
if (!tool.isReadOnly(input)) {
  await requestPermission(tool, input)
}

// UI 层:危险操作,渲染成醒目的警告
if (tool.isDestructive(input)) renderWithWarning()
```

> **工具自己“申报”性质，上层据此统一决策。** 这是一个漂亮的关注点分离：工具不需要知道“并发编排怎么做”“权限怎么弹窗”，它只需**诚实申报**“我是只读的”“我能并行”“我不可逆”；而并发编排、权限、UI 这些横切系统，各自读取这些申报去干自己的活。加一个新工具，你只管填对这几个 flag，整个系统就自动正确地对待它。

四、决定性设计之二：一次执行，两个受众

看 `mapResultToModelContent` 和 `renderResultForUI` 这对方法——同一次工具执行的结果，要渲染成两个截然不同的版本：

	给模型看（`mapResultToModelContent`）	给人看（`renderResultForUI`）
目标	让模型准确理解、省 token	让人一眼看懂、美观
形式	纯文本 / 结构化数据	带颜色、diff、折叠的 UI 组件
例子(编辑文件)	`"已编辑 foo.ts,替换 3 处"`	一块带绿/红高亮的 diff 视图
例子(跑测试)	完整的失败堆栈(模型要靠它调试)	折叠的“3 passed, 1 failed”，点开才看细节

这个分裂无处不在，而且极其重要：

> **给模型的和给人的，是两种根本不同的信息需求，绝不能混为一谈。** 模型需要完整、结构化、省 token 的信息去推理(它要看完整堆栈才能修 bug)；人需要摘要、可视化、可折叠 的信息去监督(人不想被一屏堆栈淹没)。很多 Agent 体验差，就差在把“给模型的原始输出”直接甩给用户看，或者反过来，把“给人的精简摘要”喂给模型，导致模型缺信息。想清楚每一份数据的受众，是 Agent 工程的一条隐形主线——它还会在上下文管理里再次浮现。

五、实现细节：两个不能省的关节

① Schema 就是模型的 API 契约，校验是安全底线

`inputSchema` 用 zod 之类的库定义，然后自动转成 JSON Schema 发给模型——模型就是照着这份 Schema“填参数”的。但上一篇说过，模型填的东西不可全信。所以执行前必须过 `validateInput`：

Schema 双重身份:既是给模型的说明书,又是给宿主的校验器

```ts
const EditInput = z.object({
  path: z.string().describe('要编辑的文件路径'),
  oldText: z.string().describe('要被替换的原文'),
  newText: z.string().describe('替换成的新文'),
})
// ⟶ 转成 JSON Schema 发给模型(告诉它怎么调)
// ⟶ 同一份 Schema 在执行前校验模型的输出(挡住非法输入)
```

一份 Schema，两个方向：对模型是“说明书”，对宿主是“门卫”。

② 安全默认：fail-closed

工具那么多方法，每个工具都全部实现太累。所以通常有一个 `buildTool()` 工厂填充默认值。而这些默认值的选择，体现了一条铁律——往安全的方向兜底(fail-closed):

默认值一律往

```ts
const TOOL_DEFAULTS = {
  isReadOnly: () => false,           // 默认假设它"会写"(更需要谨慎)
  isConcurrencySafe: () => false,    // 默认假设它"不能并行"(更保守)
  isDestructive: () => false,        // (这个默认 false,但危险工具必须显式声明 true)
  checkPermissions: () => allow,     // 默认交给通用权限系统兜底
}
```

> 拿不准时，选更安全的那个默认值。 一个工具如果忘了声明 `isConcurrencySafe`，系统就当它“不能并行”——顶多慢一点，但绝不会因为乱序并发把数据搞坏。忘了声明 `isReadOnly`，就当它“会写”——顶多多问一次权限，但绝不会让一个危险操作蒙混过关。默认值错在“过度谨慎”这边，代价是性能；错在“过度放任”那边，代价是灾难。 这个不对称，决定了默认必须 fail-closed。

六、小结：工具是 Agent 的“设备驱动”

再用序章的比方：如果说 Agent Loop 是机器的时钟，工具就是这台机器的外设驱动——每个驱动都以统一的接口接入系统(`call`)，同时向系统申报自己的能力和约束(`isReadOnly`、`isConcurrencySafe`.....)，让上层的调度、权限、渲染子系统能统一地、安全地驱动它。

到这里，第二阶段“让模型能做事”就完整了。我们有了：能决策的模型、能执行的工具、驱动它们协作的循环、以及一套让工具可扩展又安全的抽象。一个能自主干活的 Agent 内核，已经成型。

但它现在还很“慢”、很“笨”：工具一个一个串行跑，模型输出要全部生成完才处理。下一阶段，我们让它快起来、并行起来。下一篇进入第三阶段：流式处理——如何边生成边消费，以及那个让工具能“抢跑”执行的关键机制。

05 · 流式处理：边生成边消费

第二阶段我们把 Agent 内核搭起来了，但它有个体验硬伤：每一轮都要等模型把整条回复生成完才有反应。模型吐一大段要好几秒，用户对着空屏幕干等。这一篇，我们让它边生成边显示——顺便解锁一个后面至关重要的能力。

一、背景：非流式的两宗罪

03 篇的循环用的是“一次性”调用：`await create(...)`，等模型全部生成完，一次性拿到完整消息。这在交互式 Agent 里有两个问题：

1. 用户体验：模型生成一段几百字的分析要 5-10 秒，这期间用户什么都看不到，只能干等。真人助手会一边想一边说，而它却是憋一大段一次性砸出来。
2. 更要命的：你没法对“半成品”做任何事。 假设模型这一轮要调用 3 个工具。非流式模式下，你必须等整条消息(可能还带一大段前置文字)全部生成完，才能看到这些工具调用、才能开始执行。但其实第一个工具调用可能在第 1 秒就已经“说清楚”了——你却要等到第 8 秒消息结束才动手。这 7 秒本可以用来干活。

流式(streaming)同时解决这两件事。

二、解决方案：把一次响应拆成一串事件

流式调用不再返回“一个完整消息”，而是返回一串事件流。你订阅这串事件，一边到达一边处理。事件的序列大致是这样(不同 API 命名略有差异，但结构一致):

一次流式响应的事件序列

```
message_start                      ←— 消息开始,带上初始元数据
  content_block_start (index 0)    ←— 第 0 个内容块开始(比如一个 text 块)
    content_block_delta            ←— 文本增量:"我"
    content_block_delta            ←— 文本增量:"来看"
    content_block_delta            ←— 文本增量:"一下"
  content_block_stop  (index 0)    ←— 第 0 块结束
  content_block_start (index 1)    ←— 第 1 个内容块开始(一个 tool_use 块!)
    content_block_delta            ←— 参数 JSON 增量:'{"pa'
    content_block_delta            ←— 参数 JSON 增量:'th":"sr'
    content_block_delta            ←— 参数 JSON 增量:'c/x.ts"}'
  content_block_stop  (index 1)    ←— 第 1 块结束,此时参数才完整
message_delta                      ←— 收尾:带上 stop_reason 和最终 usage
message_stop                       ←— 整条消息结束
```

把这些 `delta` 按 `index` 归位、拼接起来，你就实时地重建出了那条完整消息。文本块的增量拼起来是完整文本，工具块的增量拼起来是完整参数。

三、实现细节：文本增量，人人会写；工具参数，暗藏玄机

文本：拼起来就完事

文本块最简单，来一片显示一片：

消费文本增量

```ts
for await (const event of stream) {
  if (event.type === 'content_block_delta' && event.delta.type === 'text_delta'
    process.stdout.write(event.delta.text)   // 来一片,打一片
  }
}
```

我们配套的 Agent Loop 从“一次性”升级到“流式”，核心就是把 `await create()` 换成迭代事件流，并把文本增量 `yield` 出去：

src/agent.ts —— 循环体从一次性升级为流式

```ts
const stream = client.messages.stream({
  model: MODEL, max_tokens: MAX_TOKENS, system: SYSTEM_PROMPT, messages, tools:
})

// 边到达边把文本增量 yield 给 UI —— 用户立刻看到模型在"说话"
for await (const event of stream) {
  if (event.type === 'content_block_delta' && event.delta.type === 'text_delta'
    yield { type: 'assistant_text_delta', text: event.delta.text }
  }
}

// 事件流走完后,拿到重建好的完整消息,后续逻辑(挑 tool_use)不变
const res = await stream.finalMessage()
messages.push({ role: 'assistant', content: res.content })
```

> 注意这个漂亮之处：升级到流式，`03 篇`那个循环的骨架一行没动。我们只是把“等一整条”换成了“迭代增量 + 最后取完整消息”。`for await ... finalMessage()` 这个模式，让流式的复杂性被封装在“获取模型响应”这一步内部，循环的其余部分(挑工具、执行、回填)浑然不觉。好的抽象就该这样。

工具参数：它是部分 JSON，一片片来的

现在讲那个暗藏玄机、也是最多人栽跟头的细节。看上面事件序列里 `tool_use` 块的增量：

```
'{"pa'  →  'th":"sr'  →  'c/x.ts"}'
```

工具调用的参数，不是一次性给你一个 JSON 对象，而是把 JSON 序列化后的字符串，一小片一小片地流给你(这类增量通常叫 `input_json_delta`)。你必须把这些片段攒起来，直到该块 `content_block_stop`，才拼成完整的 `{"path":"src/x.ts"}`，才能 `JSON.parse`。

累积部分 JSON,块结束时才解析

```ts
const partialJson = new Map<number, string>()   // index ⟶ 累积的 JSON 片段

for await (const event of stream) {
  if (event.type === 'content_block_delta' && event.delta.type === 'input_json_
    // 参数还在流,先攒着,别急着解析——现在它是残缺的 '{"pa'
    partialJson.set(event.index, (partialJson.get(event.index) ?? '') + event.de
  }
  if (event.type === 'content_block_stop') {
    const raw = partialJson.get(event.index)
    if (raw !== undefined) {
      const input = JSON.parse(raw)   // ⟵ 到这一刻,参数才完整、才能解析
      // 现在这个工具调用"齐活"了,可以派发执行了!
    }
  }
}
```

⚠ 千万别在参数流到一半时就去 `JSON.parse`。`'{"pa'` 不是合法 JSON，解析必炸。参数的完整性，只有在该内容块 `content_block_stop` 的那一刻才得到保证。这个“逐字到达 + 延迟解析”的模型，是流式处理里最容易被忽略、又最容易埋 bug 的地方。很多“工具偶尔调用失败”的诡异问题，根子就在于过早解析了残缺参数。

四、这一篇真正解锁了什么

流式的 UX 价值很直观，但它更深远的意义在最后那句注释里：

> *“一旦某个 `tool_use` 块 `content_block_stop`，这个工具调用的参数就齐全了——哪怕整条消息还没结束，你已经可以立刻派发它去执行了。”*

回到开头那个“第 1 秒就说清了第一个工具调用，却要等到第 8 秒”的浪费：有了流式，你不必再等。工具块一结束就抢跑，让工具执行和“模型继续生成后面内容”在时间上重叠起来。

这个“边流边执行”的优化，以及“多个工具怎么安全地并行跑”，就是下一篇的主题。我们会看到，把这块做好，能实实在在地砍掉 Agent 的等待时间。

本篇对应代码：配套仓库把 Agent Loop 升级为流式版本(`src/agent.ts`),CLI 实时打印模型输出。`git checkout article-05` 查看这一阶段。

06 • 边流边执行 + 并发编排：把等待时间榨干

上一篇结尾埋了个引子：工具块一 `content_block_stop`，参数就齐了，不必等整条消息结束就能派发。这一篇把这个优化，连同“一批工具怎么并行”一起做掉。目标只有一个：把 Agent 花在等待上的时间榨干。

一、背景：两处被浪费的时间

一个朴素的循环(哪怕是流式的)，在工具这块有两处明显的时间浪费：

浪费一：工具干等整条消息结束。 模型这一轮先说了一大段分析，然后才给出工具调用。就算第一个工具调用在第 2 秒就流完了参数，朴素实现也要等到第 8 秒整条消息 `message_stop` 后才开始执行它。中间 6 秒，工具在干等。

浪费二：一批工具串行跑。 模型一口气要求“读 A、读 B、读 C”三个文件。串行读：0.3s + 0.3s + 0.3s = 0.9s。可这三个读互不相干，本该同时进行，总共 0.3s 就够。

两处浪费，对应两个优化：边流边执行 和 并发编排。

二、优化一：边流边执行(dispatch-on-complete)

思路直接：别等 `message_stop`，盯住每个 `content_block_stop`。一个 `tool_use` 块结束的那一刻，它的参数已完整，立刻把它派发出去执行——让工具跑在“模型还在生成后面内容”的同时。

工具块一结束就抢跑

```ts
for await (const event of stream) {
  // ...累积文本、累积工具参数(见上一篇)...
  if (event.type === 'content_block_stop' && isToolBlock(event.index)) {
    const toolUse = assembleToolUse(event.index)   // 参数此刻已完整
    executor.dispatch(toolUse)                     // ← 立刻开跑,不等消息结束
  }
}
```

这个优化的收益，取决于模型“说话”和“调工具”的穿插方式。当模型习惯先调工具、边调边解释时，工具执行几乎完全被“生成后续文本”的时间盖住了——等模型话说完，工具结果也差不多回来了。你把两段本来串行的时间，叠成了并行。

三、优化二：并发编排，但“写”必须守规矩

同一批多个工具，能不能一起跑？要看它们是不是“并发安全”。 还记得04 篇里工具自报的那个 `isConcurrencySafe` 吗？它就是为这一刻准备的：

按 isConcurrencySafe 分流:能并行的并行,其余保序串行

```ts
const parallel: number[] = []   // 并发安全(读文件、搜索……)
const serial: number[] = []     // 不安全(写文件、跑命令……)
toolUses.forEach((tu, i) =>
  (findTool(tu.name)?.isConcurrencySafe ? parallel : serial).push(i),
)

await Promise.all(parallel.map(runOne))   // 一起上
for (const i of serial) await runOne(i)   // 老实排队,一个接一个
```

这正是配套 Agent 里已经在跑的逻辑。为什么“写”必须串行？

> ⚠ 读可以乱序，写必须有序。 三个“读文件”并行没问题——它们不改变任何状态，谁先谁后结果都一样。但两个“编辑同一个文件”如果并行，就会互相覆盖、产生竞态，结果取决于谁后写完，不可预测。更微妙的是：模型常常依赖顺序——它可能打算“先创建文件，再往里写内容”，这俩要是并行，第二步可能在第一步之前跑，直接失败。所以凡是会改动世界的操作，默认串行、保持模型给出的顺序，是一条不能破的铁律。这也是为什么 `isConcurrencySafe` 的默认值是 `false` ——宁可慢，不可错。

四、实现细节：一个流式工具执行器

把两个优化合起来，就是一个 StreamingToolExecutor：在流式过程中接收陆续完成的工具块，并发安全的立即开跑，不安全的排进串行队列；最后统一收集结果、按 `tool_use_id` 配对。它要处理几件麻烦事：

- 保序收集：虽然并行执行，但回填给模型的 `tool_result` 要能和每个 `tool_use` 一一对上(靠 `tool_use_id`，和顺序无关)。
- 串行队列的依赖：不安全的工具即使早早流完，也得等前面的串行任务做完才轮到它。
- 中断兜底：执行到一半用户按了 Ctrl+C 怎么办？——已派发的、排队中的工具，每一个都必须产出一个 `tool_result`(哪怕是“已取消”)，否则历史就非法了。这个善后逻辑，是中断那一篇的主题，这里先埋下钩子。

StreamingToolExecutor 的骨架(见配套仓库 src/streamingTools.ts)

```
class StreamingToolExecutor {
  private inflight: Promise<Outcome>[] = []
  private serialQueue: Promise<void> = Promise.resolve()

  dispatch(tu: ToolUseBlock) {
    const tool = findTool(tu.name)
    if (tool?.isConcurrencySafe) {
      this.inflight.push(this.exec(tu))              // 安全:立刻并行开跑
    } else {
      // 不安全:链在串行队列尾部,保证前一个写完才轮到它
      this.serialQueue = this.serialQueue.then(async () => {
        this.results.set(tu.id, await this.exec(tu))
      })
    }
  }

  async drain(): Promise<Outcome[]> {
    await Promise.all(this.inflight)                 // 等并行的
    await this.serialQueue                            // 等串行的
    return this.collectInToolUseOrder()              // 按 tool_use 顺序归位
  }
}
```

> 一个务实的提醒：这些优化是锦上添花，不是雪中送炭。03 篇那个朴素循环功能上完全正确，只是慢一点。先把正确的循环跑通，再来叠这些性能优化——顺序别反了。过早引入流式并发执行，会让你在还没理清基本控制流时，就陷进竞态和保序的泥潭。

五、第三阶段小结

到这里，我们的 Agent 不仅能跑，还跑得快了：

篇章	榨掉了什么等待
05 流式	用户不再干等模型憋完整段
06 边流边执行 + 并发	工具执行盖进生成时间；能并行的读一起跑

但“快”是有代价的——我们让它自动执行的东西越来越多、越来越猛。一个能自主 `bash` 任意命令、并行改文件的 Agent，已经足够危险了。是时候给它装上闸门了。下一篇进入第四阶段：权限系统——如何在“放手让它干”和“别让它闯祸”之间划清界线。

本篇对应代码：配套仓库加入 `src/streamingTools.ts`(流式工具执行器)，并发编排逻辑在 `src/agent.ts`。`git checkout article-06`。

07 • 权限系统：在“放手”和“闯祸”之间划线

前三阶段，我们造了一个能自主 `bash` 任意命令、并行改文件、不知疲倦循环下去的 Agent。这很强大，也很危险——一个 `rm -rf` 或者改错一个配置文件，后果自负。这一篇，我们给它装闸门。

一、背景：自主性的另一面是破坏力

Agent 越自主，能闯的祸越大。它可能：

- 跑一条你没预料到的破坏性命令(`git reset --hard`、`rm`、`DROP TABLE`)。
- 改动它不该碰的文件(你的 `.env`、CI 配置、别的项目)。
- 把敏感信息发到外部(一个“发请求”的工具 + 一段 API key)。

模型不是恶意的，但它会犯错，也可能被诱导(prompt injection：一个被读进来的文件里藏着“现在删库”的指令)。所以我们需要一道闸门：在每个有副作用的操作真正执行之前，拦一道，决定放行、拦截、还是问用户。

二、解决方案：分层权限模型

好的权限系统不是一个 `if`，而是几层协同。从“工具自身”到“全局模式”，层层收窄：

第 1 层：工具自分类

还记得04 篇工具自报的 `isReadOnly` / `isDestructive` 吗？这是第一道信号：

- 只读操作(读文件、搜索、列目录)几乎总能自动放行——它们不改变世界，最坏也就是看了不该看的。
- 写/破坏性操作(编辑、删除、跑命令)默认就要过闸。

一步就把绝大多数工具调用(读多写少)自动放行了，只在真正有副作用时才拦。

第 2 层：权限规则

对需要过闸的操作，再看有没有预设规则。规则通常是“工具名 + 参数模式”的匹配：

规则按

```
always-allow:  Bash(git status)      # git status 永远放行
always-allow:  Bash(npm test)        # 跑测试永远放行
always-deny:   Bash(rm *)            # 删除永远拦截
always-ask:    Edit(*)               # 改文件每次都问
```

三类规则：永远允许 / 永远拒绝 / 每次询问。规则让用户能一次性表达“这类操作我信得过”“那类绝对不行”，避免被反复打扰，也避免危险操作蒙混。

第 3 层：权限模式

最外层是一个全局的权限模式，它像一个总开关，决定整体的松紧：

模式	行为	适用场景
plan(计划)	只读，任何写操作一律拒绝	让它先分析、出方案，绝不动手
default(默认)	读放行，写要问	日常，人盯着
acceptEdits(接受编辑)	文件编辑自动放行，跑命令仍要问	信任它改文件，但命令还是把关
bypass(放行一切)	全部自动放行	沙箱/一次性容器里才敢开

> ⚠ 权限模式是一个从安全到危险的连续谱。`plan` 最安全(它根本不能动手),`bypass` 最危险(等于把闸门焊死开着)。`bypass` 只应在完全隔离的环境(一次性容器、沙箱 VM)里使用——在那里，Agent 就算失控，炸掉的也是个随时能重建的沙箱，而不是你的开发机。把 `bypass` 用在真实工作环境，等于卸掉了整套安全系统。

三、实现细节：`canUseTool` 闸门

这套分层逻辑，落地成一个统一的闸门函数——通常叫 `canUseTool`。它在每个工具执行前被调用，返回一个决定：

TS 权限决定是一个结构化结果,不是一个布尔

```ts
type PermissionResult =
  | { behavior: 'allow'; updatedInput: unknown }    // 放行(可顺带改写入参)
  | { behavior: 'deny'; message: string }           // 拦截(附上给模型看的理由)
  | { behavior: 'ask'; message: string }            // 交给用户裁决

async function canUseTool(tool: Tool, input: unknown, ctx): Promise<PermissionR
  // 第 3 层:plan 模式下,写操作直接拒
  if (ctx.mode === 'plan' && !tool.isReadOnly(input)) {
    return { behavior: 'deny', message: '当前处于计划模式,不能执行写操作。' }
  }
  // 第 1 层:只读操作放行
  if (tool.isReadOnly(input)) return { behavior: 'allow', updatedInput: input }
  // 第 2 层:查规则
  const rule = matchRule(tool, input, ctx.rules)
  if (rule === 'allow') return { behavior: 'allow', updatedInput: input }
  if (rule === 'deny') return { behavior: 'deny', message: '该操作被规则拒绝。' }
  // acceptEdits 模式:编辑类放行
  if (ctx.mode === 'acceptEdits' && isEditTool(tool)) {
    return { behavior: 'allow', updatedInput: input }
  }
  // 兜底:问用户
  return { behavior: 'ask', message: `是否允许 ${tool.name}?` }
}
```

在循环里，它插在“校验通过”和“真正执行”之间：

TS 闸门插在执行之前

```ts
const decision = await canUseTool(tool, input, ctx)
if (decision.behavior === 'ask') {
  const approved = await promptUser(decision.message)   // 弹给用户
  if (!approved) return denied('用户拒绝了该操作')
}
if (decision.behavior === 'deny') return denied(decision.message)
// 只有放行了,才真正执行
const result = await tool.run(decision.updatedInput ?? input, ctx)
```

两个精妙之处

① 拒绝要“喂回模型”，而不只是报错。 这是最关键的设计。当一个操作被拒，你不是简单地抛异常终止——你把“被拒绝了，原因是 X”作为一个 `tool_result`（`is_error: true`）喂回给模型。于是模型知道自己刚才那步没被允许，可以换个思路：改用只读方式先了解情况、或者向用户解释为什么需要这个操作、或者绕道。拒绝是给模型的一个反馈信号，而不是一个终点。这让 Agent 在受限环境里依然能优雅地工作，而不是一撞墙就崩。

② 权限层可以改写入参（`updatedInput`）。 注意 `allow` 里带的 `updatedInput`——权限系统不只是“放行/拦截”的开关，它还能在放行的同时修改工具的输入。典型用途：把一个相对路径规范成绝对路径、把某个危险参数替换成安全默认、给命令加上超时限制。这让权限层成为一个“净化关卡”，而不只是“检查关卡”。

四、又一次 fail-closed

你会注意到，整个权限系统的默认倾向，和04 篇的工具默认值一脉相承——拿不准就往安全的方向兜：

- 工具没声明只读？当它会写，过闸。
- 没匹配到任何允许规则？默认问用户，而不是默认放行。
- 模式没设？用最保守的 `default`，而不是 `bypass`。

权限系统错在“过度谨慎”这边，代价是多问用户几次(烦)；错在“过度放任”那边，代价是数据没了(灾难)。 这个代价的不对称，决定了每一个默认、每一处兜底，都必须偏向谨慎。

五、小结

权限系统是 Agent 从“玩具”走向“能托付真实工作”的分水岭。它的精髓不在于“能不能拦住”，而在于如何在拦截的同时，不破坏 Agent 的自主性——靠的就是那两点：分层地让绝大多数安全操作自动放行(别烦用户)，以及把拒绝作为反馈喂回模型(让它能适应，而不是崩溃)。

但闸门只解决了“该不该让它做”。还有一个正交的问题：它正在做的时候，用户突然想喊停、或者改主意了，怎么办？ 下一篇中断与转向，我们处理这个在流式、并发、还欠着一堆 `tool_result` 的复杂局面下，如何干净地叫停。

本篇对应代码：配套仓库加入 `src/permissions.ts`(分层权限 + `canUseTool`)与权限模式，闸门接入工具执行前。`git checkout article-07`。

08 • 中断与转向：干净地叫停

权限管的是“该不该让它做”；这一篇管的是一个正交问题：它正做着，用户突然想喊停，或者改主意了，怎么办？ 在一个流式、并发、还欠着一堆 `tool_result` 的局面下，“干净地停下”比想象中难。

一、背景：停下来，不能停出一地烂摊子

用户按下 Ctrl+C 的那一刻，Agent 可能正处于任意状态：

- 模型的响应流到一半。
- 三个工具并行跑着，一个刚写完文件，一个还在 `npm install`，一个在队列里没轮到。

如果粗暴地一停了之，你会撞上02 篇那条铁律的反面：

> ⚠ 历史里每个 `tool_use` 都必须有配对的 `tool_result`。 中断时，你已经把带 `tool_use` 的助手消息放进历史了(模型“说”它要调这些工具)，但这些工具还没产出结果。如果就这么停下、下次接着聊，这段历史就是非法的——API 会直接拒绝你的下一个请求。所以中断的核心工作不是“停”，而是“善后”：给每一个悬空的 `tool_use`，补一个(内容为“已取消”的) `tool_result`，把历史重新变合法。

二、解决方案：AbortSignal 贯穿 + 合成结果

一根信号，贯穿到底

中断的机制核心是一个 `AbortSignal`。它必须从最外层一路穿透到每一个可能长时间阻塞的地方：

AbortSignal 贯穿模型调用与每个工具

```ts
const controller = new AbortController()

// 1) 传给模型流式调用——abort 时,流会中止
const stream = client.messages.stream({ ... }, { signal: controller.signal })

// 2) 传给每个工具——比如 bash 把它交给子进程,abort 时命令被杀
await execAsync(command, { signal: controller.signal })
```

用户一喊停，`controller.abort()` 一调，这根信号同时到达“正在生成的模型流”和“正在跑的工具”，两边一起停。

给悬空的工具合成“已取消”结果

停下之后，遍历这一轮所有已发出的 `tool_use`，给每个还没结果的，合成一个“已取消”的 `tool_result`：

工具执行入口先看信号:已中断就直接产出取消结果

```
async function executeTool(tu, ctx): Promise<Outcome> {
  // 排在队列里、还没轮到就被中断的工具:直接给一个取消结果,不真正执行
  if (ctx.signal.aborted) {
    return {
      block: { type: 'tool_result', tool_use_id: tu.id, content: '操作已被用户中断
      // ...
    }
  }
  // ...正常执行...
}
```

这样，无论某个工具是“跑到一半被杀”(它自己因 signal 抛错，产出错误结果)还是“还没轮到就被取消”(入口处直接返回取消结果)，每个 `tool_use` 最终都拿到了一个配对的 `tool_result`。历史保持合法，下次可以无缝继续。

这里能看出06 篇那个流式执行器的设计价值：因为它统一管理着“已派发/在跑/排队中”的所有工具，中断时它能一次性地为这些工具产出善后结果(在跑的等它们因 signal 自然结束，排队的直接标记取消)。**把工具执行收拢到一个执行器里，中断善后才有地方统一处理**——否则你得满世界追那些散落的 Promise。

三、两种“打断”：硬中断 vs 转向

用户打断 Agent，其实有两种意图，处理方式不同：

	硬中断(Interrupt)	转向(Steering)
用户意图	“停！别干了”	“等等，其实我想让你换个方向”
触发	Ctrl+C、停止按钮	在它跑着时又输入了一句话
处理	善后 → 停在当前回合，等新指令	善后 → 把新消息作为下一轮的用户输入接上去

转向是“打断 + 追加”，而不是“打断 + 丢弃”。 当用户在 Agent 跑着时敲入新指令，成熟的做法不是丢掉当前进度重来，而是：干净地结束当前回合(合成善后结果)，然后**把用户这句新话，连同已完成的部分工作，一起作为下一轮的输入**。模型于是能“带着上文改方向”——它看得到自己刚才做到哪了，也看得到用户的新要求。这种“边跑边插话调整方向”的能力，是交互式 Agent 体验丝滑的关键。实现上，常配一个**消息队列**：跑的时候用户输入先入队，当前回合一结束就出队接上。

四、实现细节：中断路径的几个当口

把中断逻辑接进03 篇的循环，要照顾几个当口：

1. 流式途中被中断：模型流因 signal 抛错/中止，捕获它，别让它冒泡成“真错误”(它是用户主动行为，不是故障)。
2. 工具执行途中被中断：执行器为每个工具产出善后结果(取消/错误)，保证配对完整。
3. 回合收尾：产出一条“用户中断”的标记消息(让 UI 和后续逻辑知道这是主动中断)，然后停在这个回合——不再自动进入下一轮，把控制权交回用户。

循环里的中断处理(简化)

```ts
try {
  // ...流式调用、边流边派发工具...
} catch (e) {
  if (signal.aborted) {
    const outcomes = await executor.drain()    // 收齐所有善后结果
    for (const o of outcomes) yield toolResultEvent(o)
    yield { type: 'done', reason: 'interrupted' }
    return                                     // 停在这一回合
  }
  throw e                                      // 其它错误,继续往上抛(见下一阶段)
}
```

> ⚠ 一个容易搞错的地方：用户主动中断，不是“错误”。 别把它和模型故障、网络超时混在一条错误路径里处理——那会让日志里满是假报警，也会让 UI 弹出误导性的“出错了”。中断是一条独立的、正常的控制流分支，要和真正的错误恢复(下一阶段的主题)分开对待。

五、小结

中断看似是个 UI 小功能，背后却牵动着 Agent 最核心的不变式——历史的合法性。做好它，靠的是三件事：一根贯穿到底的 `AbortSignal`、一个能统一善后的工具执行器、以及把“用户主动停”和“系统出故障”清清楚楚分开。

到这里，第四阶段“安全与控制”就完整了：权限管住了“能做什么”，中断管住了“何时停下”。我们的 Agent 现在既自主又可控。

但还有一个扩展点没碰：如果用户想在工具执行前后插入自己的逻辑呢？——比如“每次写文件前自动跑一遍格式化”“每次提交前拦一道检查”。下一篇Hooks，我们给 Agent 开一组“可编程的插槽”。

本篇对应代码：配套仓库让 `executeTool` 感知中断信号并合成取消结果，CLI 用 Ctrl+C 触发本回合中断。`git checkout article-08`。

09 • Hooks：给 Agent 开一组可编程的插槽

权限是内置的安全策略，但用户总有自己的、五花八门的诉求：“每次改完文件自动跑一遍格式化”“提交前必须过 lint”“测试没绿之前不许它说完成”。这些没法全塞进 Agent 内核。这一篇，我们给它开一组可编程的插槽。

一、背景：内核装不下所有人的规矩

不同团队、不同项目，对 Agent 有完全不同的额外要求：

- 前端项目：每次 `edit_file` 后，自动 `prettier --write`。
- 后端项目：任何 `bash` 命令，先记一条审计日志。
- 严格团队：Agent 说“我做完了”之前，必须确认测试通过，否则打回去继续干。

这些逻辑因人而异、因项目而异，不可能硬编码进 Agent。你需要的是一种机制：让用户在 Agent 生命周期的固定节点，挂上自己的代码。 这就是 Hooks。

二、解决方案：生命周期钩子

思路是在 Agent 循环的关键时刻，埋下若干命名的挂载点，每个点允许用户注册处理器。常见的挂载点：

钩子	触发时机	能干什么
PreToolUse	某个工具执行前	拦截、改写入参、或放行
PostToolUse	某个工具执行后	对结果做反应(格式化、通知、追加检查)
Stop	Agent 准备收工时	拽住它，强制它继续干
UserPromptSubmit	用户提交输入时	预处理、注入额外上下文

> Hooks 是“用户态”对 Agent 的扩展，而且是确定性的。 注意这个关键区别：模型的行为是概率性的(你没法保证它每次都记得跑格式化)；而钩子是确定性的代码——挂上了，就每次必然执行。所以 Hooks 的价值，恰恰在于把那些“必须每次都做、不容模型忘记”的事，从“祈祷模型配合”变成“机制保证”。凡是你不放心交给模型自觉的横切逻辑，都该用钩子钉死。

三、实现细节：钩子的契约与三种挂载点

钩子的契约：输入结构化事件，输出决定

一个钩子接收结构化的事件信息，返回一个决定。以 PreToolUse 为例：

PreToolUse 钩子:执行前的可编程闸门

```ts
type PreToolUseDecision =
  | { decision: 'continue' }                    // 放行
  | { decision: 'block'; message: string }      // 拦截,理由喂回模型
  | { decision: 'modify'; input: unknown }      // 改写入参后放行

type PreToolUseHook = (tool: Tool, input: unknown) => Promise<PreToolUseDecisio
```

它和权限系统的 `canUseTool` 长得很像——没错，PreToolUse 本质就是一个“用户可编程的权限闸门”。区别在于：权限是内置策略，钩子是用户挂的任意逻辑。两者叠加(通常钩子先跑，再走内置权限)，构成一道可深度定制的关卡。

> 真实系统里，钩子常被实现为外部命令而非进程内函数：Agent 把事件序列化成 JSON 从 stdin 喂给一个用户指定的脚本，脚本处理后把决定以 JSON 从 stdout 返回。这样做的好处是语言无关——用户拿任何语言(bash、Python、Node)写钩子都行，Agent 完全不关心。代价是每次要起一个子进程。进程内函数快，但把用户锁死在 Agent 的语言里。工程上二者常并存。

PreToolUse：拦截与改写

接进08 篇的工具执行入口，排在权限之前：

工具执行前先跑 PreToolUse 钩子

```ts
for (const hook of hooks.preToolUse ?? []) {
  const d = await hook(tool, input)
  if (d.decision === 'block') return denied(d.message)   // 钩子拦截,理由喂回模型
  if (d.decision === 'modify') input = d.input           // 钩子改写入参
}
// ...再走内置权限、再执行...
```

Stop：拽住想收工的 Agent(最有意思的一个)

Stop 钩子挂在循环的出口——当模型不再要求调用工具、Agent 准备结束时。它有权说："不行，你还没完，接着干。"

Stop 钩子:在 Agent 想退出时拦一道

```ts
// 循环里,原本"没有工具调用就退出"的地方:
if (toolUses.length === 0) {
  for (const hook of hooks.stop ?? []) {
    const d = await hook(messages)
    if (d.decision === 'block') {
      // 钩子不让停!把它的要求作为一条新消息注入,继续循环
      messages.push({ role: 'user', content: d.message, isMeta: true })
      continue   // ← 回到循环开头,Agent 被"拽"了回来
    }
  }
  return { reason: 'completed' }   // 所有 Stop 钩子都放行,才真正结束
}
```

这个机制威力很大。比如一个“测试守门”钩子：

一个

```ts
const testGate: StopHook = async () => {
  const passed = await runTests()
  return passed
    ? { decision: 'continue' }
    : { decision: 'block', message: '测试仍未通过,请继续修复,不要停下。' }
}
```

挂上它，Agent 就没法在测试还红着的时候溜号——它每次想说“搞定了”，都会被这个钩子打回去继续修，直到测试真的绿了。

> ⚠ **Stop 钩子是把双刃剑，小心死循环。** 如果钩子的条件永远满足不了(比如要求“零 lint 警告”，但有个警告 Agent 根本改不动)，它会把 Agent 永远拽在循环里，反复“想停→被打回→再试→又想停”，烧光你的预算。所以 Stop 钩子的注入消息里，通常要给模型留一条退路(比如“若确实无法解决，请说明原因后停止”)，并配一个最大重试次数做硬闸——这和03 篇讲的失控保护是同一个道理：任何能让循环延续的机制，都必须自带刹车。

四、Hooks 与权限、Agent Loop 的关系

把这三者摆在一起看，一个清晰的分层浮现出来：

- Agent Loop(03)：提供“自主推进”的引擎。
- 权限(07)：内置的、面向安全的策略闸门。
- Hooks(本篇)：用户态的、可编程的扩展插槽，能在同样的节点上叠加任意确定性逻辑。

它们共享同一批“关键时刻”(工具执行前后、回合结束)，只是各自负责不同层面的关切。一个成熟 Agent 的可扩展性，很大程度上就体现在这套生命周期钩子的丰富度上——它决定了用户能把 Agent 定制到多深。

五、第四阶段收尾

第四阶段“安全与控制”到此完整：

篇章	装上了什么
07 权限	内置安全闸门：能做什么
08 中断	干净叫停：何时停下
09 Hooks	可编程插槽：让用户嵌入自己的规矩

我们的 Agent 现在既自主、又快、又可控、还可扩展。看起来很完整了？但有一头我们一直在回避的猛兽，始终没正面处理——上下文窗口。随着对话越滚越长、工具结果越堆越多，那个有限的窗口迟早会被撑爆。而这，是整个 Agent 工程里最难、最见功力的部分。

下一篇进入第五阶段：上下文工程。系好安全带。

本篇对应代码：配套仓库加入 `src/hooks.ts` (PreToolUse / Stop 钩子)，接入工具执行前与循环出口。`git checkout article-09`。

10 • Token 预算：与有限上下文窗口的持久战

欢迎来到第五阶段。前面四个阶段，我们一直在“往 Agent 上加能力”；从这一阶段起，我们要处理一个一直在回避、却决定 Agent 生死的约束——上下文窗口是有限的。这是整个 Agent 工程里最难、最见功力的部分。

一、背景：一场必输的军备竞赛

回忆01 篇那个残酷事实：模型无状态，所以每一轮你都要把到目前为止的全部历史重发一次。现在把它和 Agent 的工作方式叠加起来看：

- 一轮对话里，历史在疯长：用户消息、助手消息、`tool_use`、还有——最凶的——工具结果。
- 一次 `read_file` 读进来一个 2000 行的文件，可能就是上万 token。一条 `npm test` 的输出、一次 `git diff`，随随便便几千 token。
- 这些全都留在历史里，每一轮都要重发。

于是上下文像滚雪球一样膨胀。而窗口有硬顶(比如 20 万 token)。这是一场你必输的军备竞赛——只要对话够长，迟早撞顶。撞顶的后果有两种，都很糟：

1. 硬失败：请求超过窗口，API 直接拒绝(常见的 `prompt too long` / HTTP 413),Agent 卡死。
2. 软退化：就算没撞顶，上下文越长，模型越容易“迷失在中间”——关键信息被淹没在几万 token 的历史里，它开始遗忘、跑偏。

> 所以上下文不是“越多越好”，而是一种需要精打细算的稀缺资源。 第五阶段所有的技术——压缩、缓存、瘦身、注入——本质都在回答同一个问题：在有限的窗口里，如何始终装着“模型此刻最需要的那些信息”，而把其余的挪走。 这就是“上下文工程”(context engineering)。而一切的前提，是先能量出当前用了多少。

二、解决方案：先学会“量”

管理上下文的第一步，是知道它现在有多大。这里有个务实的现实：精确计算 token 需要跑分词器(tokenizer)，开销不小，而 Agent 每一轮都要算好几次。 所以工程上通常是“估算 + 校准”两条腿走路：

估算：便宜的近似

一个粗略但够用的估算：字符数 / 4 ≈ token 数(英文经验值；中文、代码另有系数)。它不精确，但极快，足够用来做“要不要开始警惕了”的判断。

便宜的 token 估算

```ts
function estimateTokens(messages: Message[]): number {
  const chars = JSON.stringify(messages).length
  return Math.ceil(chars / 4)    // 快而糙,用于水位判断
}
```

校准：用 API 回报的 usage 作地面真值

还记得01 篇响应里那个 `usage` 吗？每次 API 响应都会回报这次请求的精确 `input_tokens`。这是权威数字。聪明的做法是：拿上一次响应的 `input_tokens` 作为“地面真值”，再叠加此后新增消息的估算量：

用上次的精确值 + 新增的估算值

```ts
// 上一次 API 已经告诉我们历史精确是多少 token,
// 只需估算"这之后又加了多少",两者相加,又快又准。
const contextTokens = lastResponse.usage.input_tokens + estimateTokens(newMessa
```

> 这个“精确锚点 + 增量估算”的模式很关键：纯估算会累积误差(尤其长对话)，纯精确又太慢。用 API 每轮免费回报的 `input_tokens` 当锚，把估算只用在“最近这一小段增量”上，就同时拿到了速度和准头。善用 API 白送的 `usage`，是上下文管理的基本功。

三、实现细节：分级水位线

量出了当前用量，接下来要根据它触发不同动作。成熟做法是设几道水位线，像水库泄洪一样分级响应：

水位	占窗口比例(示意)	动作
正常	< 60%	什么都不做
警告	~60-80%	提示用户“上下文有点满了”，可考虑手动清理
自动压缩线	~80-90%	触发自动压缩，把老历史摘要掉
硬阻断线	~92%+	拒绝再塞新内容，留出余量让用户还能手动 `/compact`

根据水位决定动作

```ts
function checkContextWater(tokens: number, windowSize: number) {
  const pct = tokens / windowSize
  if (pct >= 0.92) return { level: 'blocking' }   // 硬阻断:必须先压缩
  if (pct >= 0.80) return { level: 'autocompact' } // 触发自动压缩
  if (pct >= 0.60) return { level: 'warn' }        // 提醒用户
  return { level: 'ok' }
}
```

> ⚠ 注意那条“硬阻断线”为什么不设在 100%，而是留一大截余量。因为压缩本身也要调用模型(把历史喂给它、让它写摘要)，这个操作也需要上下文空间。如果你等到窗口 100% 满了才想起压缩，那连“执行压缩”所需的余量都没有了——彻底死锁。所以必须提前在还有余量时就动手，给“抢救操作”本身预留空间。这是上下文管理里一个极其现实、又极其容易被忽略的死锁陷阱。

四、别忘了输出侧也在抢空间

一个容易被忽视的点：窗口是输入 + 输出共享的。你设的 `max_tokens`(允许模型这次最多生成多少)也要从窗口里扣。而且：

- `max_tokens` 设太小 → 模型话没说完就被硬截断(`stop_reason: 'max_tokens'`)，得走恢复流程。
- 设太大 → 挤占了本可留给历史的空间。

所以 `max_tokens` 的取值，本身也是这场预算战的一部分——它是你主动让出多少窗口给“这一轮的输出”的决定。

五、小结：一切的起点

这一篇没有炫技，但它是第五阶段的地基：你必须先能量出上下文用量、并据此分级响应，后面所有的上下文技术才有触发的依据。 记住三件事：

1. 每轮重发全部历史，工具结果是膨胀的主凶。
2. 用“API 回报的 `input_tokens` 作锚 + 增量估算”来又快又准地量。
3. 设分级水位线，并给抢救操作预留余量——别等满了才动手。

水位线一响，该动手“减负”了。最主要的减负手段，就是下一篇的自动压缩：当历史太长，怎么把它安全地“折叠”成一段摘要，既腾出空间，又不丢掉模型继续工作所必需的关键信息。

不过在压缩之前，我们得先搞清楚一件更基础的事：每一轮喂给模型的上下文，到底是由哪些部分拼起来的？ 下一篇先讲系统提示与上下文注入，把“上下文”这个筐里装了什么看清楚，再谈怎么给它减负。

本篇对应代码：配套仓库在循环里接入 usage 追踪与上下文水位提示(`src/agent.ts`)。`git checkout article-10`。

11 · 系统提示与上下文注入

上一篇我们学会了量上下文的大小。但在谈怎么给它“减负”之前，得先看清：每一轮实际发给模型的那坨东西，到底由哪些部分拼成？ 很多人以为“上下文 = 消息历史”，这只对了一小半。

一、背景：模型需要“知道自己身处何地”

一个纯粹的消息历史，缺了很多模型干活必需的信息：

- 它在哪个目录下工作？当前是什么操作系统？今天几号？
- 这个项目有没有特定约定(用 pnpm 不用 npm、测试命令是什么、代码风格)?
- 用户这次提到的那个文件，内容是什么？

这些都不在“用户说的话”里，但模型必须知道，否则它只能瞎猜(还记得01 篇那个“连今天几号都不知道”的裸模型吗)。所以真实 Agent 每一轮都会把一堆信息注入进上下文。搞清楚注入了什么、注入在哪，是上下文工程的必修课。

二、解决方案：分层拼装一个 prompt

每一轮发给模型的完整 prompt，大致由这几层拼成，从“最静态”到“最动态”排列：

一次请求的上下文分层(从静态到动态)

```
┌─ 系统提示(system)──────────── 最静态、最大、最该被缓存
│     · 角色设定、行为准则
│     · 工具使用指南、输出规范
├─ 环境上下文 ──────────────── 每次会话基本不变
│     · cwd、操作系统、日期、shell
├─ 项目记忆 / 用户偏好 ──────── 每个项目固定
│     · 项目约定(测试命令、风格)、用户长期偏好
├─ 消息历史 ──────────────── 随对话增长
│     · user / assistant / tool_use / tool_result …
└─ 动态附件 ──────────────── 每轮可能变化,最该放在最后
      · 本轮引用的文件、检索到的相关记忆、目录结构
```

每一层各司其职：

- 系统提示：定义“你是谁、怎么干活”。它往往很大(几千 token 的行为规范和工具指南)，而且基本不变。
- 环境上下文：让模型知道“此时此地”。通常追加在系统提示之后，或作为一条会话开头的元信息。
- 项目记忆 / 用户偏好：项目级的约定(那种 `AGENTS.md` / `CLAUDE.md` 之类的文件)、用户的长期偏好。通常前置注入到对话开头。

- 消息历史：就是前面一直在滚的那个 `messages`。
- 动态附件：本轮才需要的、易变的东西——用户刚引用的文件、按相关性临时检索出的记忆片段。

三、决定性原则：静态放高处，动态放低处

上面那个“从静态到动态”的排列不是随便排的。它遵循一条贯穿整个上下文工程的黄金法则：

> ⚠ **把不变的东西放前面，把易变的东西放后面。** 这条原则有两个理由，一个比一个重要：
>
> ① **缓存**(下一篇的主角):LLM API 的 prompt 缓存是**前缀匹配**的——只有从头开始、一字不差的那段前缀才能命中缓存。所以你把大而稳定的系统提示放最前面，它就能在每一轮里被缓存复用，省下大笔重复计算的钱和延迟。反过来，只要你在靠前的位置塞一个每轮都变的东西(比如把当前时间戳放进系统提示开头)，它后面的一切缓存**全部作废**。
>
> ② **模型注意力**：近期的、靠后的信息，模型通常给予更高的权重。把“本轮最相关的文件”放在最靠近提问的位置，比埋在几千 token 的历史中间，更容易被模型抓住。

这条原则会在下一篇Prompt 缓存里变成实打实的钱和延迟，这里先把它刻下来。

四、实现细节：注入的两种姿势与一个陷阱

姿势一：追加到系统提示

环境信息这类“整场会话不变”的东西，适合拼到系统提示末尾：

系统提示 = 基础指令 + 环境上下文

```ts
function buildSystemPrompt(base: string, env: EnvInfo): string {
  return `${base}

<env>
工作目录: ${env.cwd}
操作系统: ${env.os}
今天日期: ${env.date}
</env>`
}
```

注意：环境信息在一次会话内通常不变(你不会中途换操作系统)，所以放进系统提示不会破坏缓存。但日期是个微妙的例外——如果你精确到秒，它每轮都变，就成了缓存杀手；精确到“天”就安全。注入动态信息前，先想想它多久变一次。

姿势二：作为消息注入

用户引用的文件、检索到的记忆，则更适合作为消息注入到历史里(通常紧挨着用户的提问):

TS 把附件作为消息前置注入

```ts
const messagesWithContext = [
  { role: 'user', content: `<file path="src/x.ts">\n${fileContent}\n</file>` },
  ...conversationHistory,
]
```

一个贯穿始终的陷阱：别原地改要复用的消息

> ⚠ 注入上下文时，常见的做法是“往消息数组里加东西”。但你要非常小心：凡是会流回下一轮 API 的消息，一旦被你原地修改，就可能悄悄毁掉Prompt 缓存。 因为缓存是靠字节级前缀匹配的——你把某条历史消息的内容改动一个字节(哪怕只是“顺手规范化一下路径格式”)，从那条消息往后的缓存就全部失效。所以注入时的正确姿势是：生成新的消息数组用于这次请求，而不去改动那些要长期留在历史里、要复用缓存的原始消息。 这个“读时拼装、不改原件”的纪律，是缓存能稳定命中的前提——下一篇会把它讲透。

五、小结：上下文是“拼”出来的，不是“堆”出来的

这一篇的核心认知：发给模型的上下文，是每一轮精心拼装的产物，而不是一个只增不减的消息堆。 它分层、有序、各有各的注入姿势，而贯穿其中的是一条铁律——静态放高处、动态放低处。

搞清楚了“上下文由什么拼成”，两个方向自然打开：

- 往“省”的方向：把靠前的静态部分缓存起来，别每轮重算——这是下一篇 Prompt 缓存。
- 往“减”的方向：当靠后的动态部分(主要是历史)膨胀到装不下，把它压缩掉——这是第 13 篇 自动压缩。

我们先走“省”这条路。下一篇，Prompt 缓存——整个 Agent 里省钱省延迟最立竿见影的一招，以及它那条不容触碰的“字节稳定性”红线。

本篇对应代码：配套仓库把静态 `SYSTEM_PROMPT` 升级为“基础指令 + 环境上下文”的拼装(`src/agent.ts`)。`git checkout article-11`。

12 • Prompt 缓存：省钱省延迟最立竿见影的一招

上一篇我们看清了上下文由哪些层拼成，也埋下了那条“静态放高处”的原则。这一篇，兑现它的价值——用 Prompt 缓存，把靠前那些稳定的部分从“每轮重算”变成“一次算、多轮复用”。这是整个 Agent 里投入产出比最高的一个优化。

一、背景：你在为同一段文字反复付钱

看清上一篇那个分层后，一个浪费触目惊心：

- 系统提示几千 token(行为规范、工具指南)，每轮不变。
- 工具定义几十个工具的 schema，又是几千 token，每轮不变。
- 可是模型无状态，每一轮你都得把这几千 token 原样重发，模型也每一轮都从头重新处理它们。

一个几十轮的 Agent 会话，同一段系统提示被重新计算了几十遍。你在为完全相同的一段文字，反复付钱、反复等待。 这纯属浪费——它每轮都一模一样，凭什么要重算？

二、解决方案：缓存稳定的前缀

Prompt 缓存的思路：让 API 把 prompt 的某个前缀缓存下来；下一个请求只要前缀一字不差，就直接复用缓存的计算结果，跳过重算。

- 命中缓存的部分，按 cache-read 计价——通常只有正常输入 token 价格的约十分之一，而且延迟大幅降低(模型不用重读它)。
- 第一次建立缓存，有一点点额外的 cache-write 开销(略高于正常输入)，但后续每一轮都在赚回来。

用法上，你在 prompt 里打若干缓存断点(cache breakpoint)，告诉 API “缓存到这里为止”：

TS 在稳定前缀的末尾打缓存断点

```
await client.messages.create({
  model,
  // 系统提示以"块"形式给出,末尾打一个缓存断点
  system: [
    { type: 'text', text: bigSystemPrompt, cache_control: { type: 'ephemeral' }
  ],
  // 工具定义也很稳定,在最后一个工具上打断点,缓存整个工具区
  tools: [...tools.slice(0, -1), { ...lastTool, cache_control: { type: 'ephemera
  messages,
})
```

响应的 `usage` 会告诉你缓存命中情况：

usage 里的缓存账单

```ts
{
  cache_creation_input_tokens: 5200,  // 这次写入缓存的量(第一轮)
  cache_read_input_tokens: 0,
  // ——下一轮——
  cache_creation_input_tokens: 0,
  cache_read_input_tokens: 5200,      // 这次命中缓存的量(便宜十倍!)
  input_tokens: 130,                  // 只有真正新增的部分按全价算
}
```

三、关键机制：前缀缓存 + 滚动断点

缓存是“前缀”的，不是“任意片段”的

> ⚠ 缓存匹配的是从 prompt 最开头算起、一字不差的前缀**。** 这有两个硬含义：
>
> 1. 断点之前的所有内容必须完全稳定——所以稳定的东西(系统提示、工具)要放最前面(这就是上一篇“静态放高处”的兑现)。
> 2. 你不能只缓存“中间某段”。缓存永远是“从头到某个断点”的一整段。

滚动断点：让缓存跟住增长的历史

系统提示和工具好办(它们不变)。但消息历史是一直增长的——难道历史部分就没法缓存了？

有办法：在历史里也放一个“滚动”的断点，让它随对话往前移。 关键洞察是——一轮结束后，本轮的历史会成为下一轮历史的稳定前缀。所以你把断点打在“当前历史的末尾附近”，下一轮这段历史没变，就整段命中缓存，只有最新增的一两条消息按全价算。

滚动断点:历史越长,被缓存的前缀越长

```
第 N 轮:  [系统][工具][—— 历史 1..N ——|断点]  ← 缓存写入到这
第 N+1 轮:[系统][工具][—— 历史 1..N ——][新增 N+1|断点]
                       └ 这段整个命中缓存 ┘  └ 只有这点按全价
```

于是一个长会话里，绝大部分上下文每轮都在走 cache-read，只有最新的增量按全价。成本和延迟被压到极低。

四、那条不容触碰的红线：字节稳定性

现在讲这一篇、也是整个上下文工程里最凶险的一个坑。缓存靠“字节级前缀匹配”，这意味着：

> ⚠ 断点之前的内容，哪怕只变了一个字节，从那一点往后的缓存全部作废**。** 这条红线，会以各种隐蔽的方式咬你：
>
> - 在系统提示开头放了动态数据(精确到秒的时间戳、一个随机 ID)——每轮都变，缓存从头就废，你以为开了缓存，其实一次都没命中。
> - 顺手“规范化”了一条要复用的历史消息(把相对路径改成绝对路径、重新格式化了 JSON)——从那条消息往后，缓存全废。
> - 工具的顺序不稳定(每轮动态拼工具列表，顺序还会变)——工具区的缓存废掉。
> - 同一段内容，这轮用了紧凑 JSON、下轮用了带缩进的——字节不同，缓存不认。

这条红线直接决定了一个贯穿代码的纪律，也解释了上一篇那个“别原地改要复用的消息”的告诫：

> ℹ 凡是会流回下一轮 API 的消息，必须保持字节不变；要给观察者(UI、日志)加工过的版本，请拷贝一份去改，别动原件。 这就是为什么成熟 Agent 的代码里，处处能看到“克隆一份用于展示/序列化，原始消息原封不动送回 API”的写法——不是洁癖，是每一次原地修改，都可能是一次静默的缓存击穿，而缓存击穿在账单和延迟上都是实打实的损失，还极难排查(功能正常，就是莫名其妙又贵又慢)。

五、和压缩的微妙关系

缓存和下一篇的压缩是一对需要小心协调的机制：

- 压缩会重写历史(把一大段老消息换成一段摘要)。这必然改变字节，所以压缩点之后的缓存会失效——压缩省了上下文，却付出了一次缓存重建的代价。
- 正因如此，精细的做法会尽量让压缩只动很老、本来也快过期的那部分缓存，而保住近期的热缓存；甚至有“按 `tool_use_id` 精确删除某几条工具结果、不碰其余字节”的技巧(下下篇会讲)，就是为了在减负的同时，尽量少破坏缓存。

“省”(缓存)和“减”(压缩)这两条路，在这里交汇并互相牵制——这也是上下文工程之所以难的原因：它不是几个独立技巧，而是一组互相拉扯、需要通盘权衡的约束。

六、小结

Prompt 缓存是那种“投入极小、回报极大”的优化：给稳定前缀打几个断点，长会话的成本和延迟就能降一大截。但它有一条不容触碰的红线——字节稳定性：

1. 稳定的放前面(系统提示、工具)，动态的放后面。
2. 在增长的历史里用滚动断点，让缓存跟住。
3. 凡是回流 API 的消息，一个字节都别原地改；要改就拷贝。

缓存解决了“重复的部分别重算”。但它救不了另一半问题：当历史本身膨胀到装不下时，你不能光靠缓存，你得真的把它变小。下一篇自动压缩，我们正面处理这头猛兽。

本篇对应代码：配套仓库给系统提示与工具定义加上缓存断点，并在 usage 里打印缓存命中（`src/agent.ts`）。`git checkout article-12`。

13 · 自动压缩：把历史折叠成摘要

缓存解决了“重复的别重算”，但它救不了另一半问题：当历史本身膨胀到装不下窗口时，缓存无能为力——你得真的把历史变小。这一篇，我们正面处理这头猛兽。

一、背景：缓存不缩小任何东西

分清两件事：

- 缓存让你便宜地重发同一段历史，但历史该多大还多大——它一个 token 都没减少。
- 当水位线告诉你“上下文到 85% 了”，再便宜的重发也没用，窗口就要装不下了。

这时唯一的出路是：把一部分历史，从“逐字保留”变成“摘要保留”——用一小段摘要，替换掉一大坨老消息。这就是压缩(compaction)。

二、解决方案：保留近期，摘要远古

粗暴地把整个历史塞给模型“缩短一下”是不行的——你会丢掉太多细节。聪明的压缩有个清晰的结构：一条压缩边界，把历史切成两半。

压缩:边界之前折叠成摘要,之后逐字保留

```
压缩前:[msg1][msg2][msg3]...[msg40][msg41][msg42]    ← 太长,装不下
                          压缩边界 ┘
压缩后:[-- 摘要:msg1..msg40 的精华 --][msg41][msg42]  ← 瘦身完成
```

- 边界之前(远古历史)：喂给模型，让它写一段结构化摘要，然后用这段摘要替换掉这几十条消息。
- 边界之后(近期历史)：逐字保留——模型干活最依赖最近的细节(刚读的文件、刚跑的报错)，这部分不能动。

压缩本身是一次额外的 LLM 调用：把要压缩的历史发给模型，配一个专门的“压缩 prompt”，拿回摘要：

TS 压缩 = 一次带专门 prompt 的模型调用

```ts
async function compact(oldMessages: Message[]): Promise<string> {
  const res = await client.messages.create({
    model, max_tokens: 2048,
    system: COMPACT_PROMPT,    // ← 摘要的质量,全系于这个 prompt
    messages: [...oldMessages, { role: 'user', content: '请把以上对话压缩成结构化摘要
  })
  return textOf(res)
}
// 压缩后的新历史 = [摘要] + 近期逐字保留的尾部
const newMessages = [{ role: 'user', content: `<summary>${summary}</summary>` }
```

三、决定性细节：摘要 prompt 就是成败本身

> ⚠ 压缩的全部风险，集中在那个摘要 prompt 上。摘要写好了，Agent 压缩完能无缝接着干，仿佛什么都没发生；摘要写砸了，Agent 当场失忆——忘了自己在修哪个 bug、忘了已经改过哪些文件、忘了用户最初的要求，然后开始胡乱重复劳动或跑偏。这是压缩最危险的地方：它是一个有损操作，而损掉的可能正是关键信息。

所以摘要 prompt 必须明确要求模型保留那些“继续干活所必需”的东西，通常包括：

- 当前任务是什么、用户最初的完整意图。
- 已经做了什么：改过哪些文件、跑过哪些命令、得到什么结果。
- 关键事实与决策：发现的根因、选定的方案、踩过的坑。
- 当前进行到哪一步、下一步打算干什么。
- 重要的文件路径、符号名、配置值——这些具体的“锚点”丢了就得重新去查。

一个好的摘要 prompt，往往会给出一个固定的结构模板(任务/已完成/关键发现/当前状态/下一步)，强制模型逐项填写，而不是随意“写个大概”。把摘要的结构定死，是对抗“压缩即失忆”的主要武器。

四、两种触发：proactive vs reactive

什么时候压缩？有两条路，成熟系统两条都要：

	Proactive(主动)	**Reactive(被动)**
触发	水位到阈值(如 85%)，主动压	请求已经 `413` 超长了，被迫压
时机	撞墙之前	撞墙之后恢复
心态	未雨绸缪	亡羊补牢

> **理想情况全靠 proactive**：在还没撞墙时，水位一到阈值就主动压缩，用户无感。但现实中估算总有误差，偶尔会“没料到就超了”——这时 reactive 兜底：捕获 `prompt too long` 错误，压缩历史，然后重试刚才那个失败的请求。这也呼应了第六阶段错误恢复的思路——把“上下文超长”当成一种可恢复的错误，而不是终点。两条路一主一备，才稳。

还有一个不能忘的细节：压缩本身也要消耗上下文(要把老历史发给模型)。所以水位线那篇强调的“提前动手、留出余量”在这里兑现——你不能等 100% 满了才压，那时连“执行压缩”的空间都没了。

五、绕不开的矛盾：压缩 vs 缓存

压缩和缓存天然打架，这个矛盾必须正视：

> **压缩重写历史，必然改变字节，于是击穿缓存。**你刚压缩完那一轮，前面辛苦建立的缓存从压缩点起全废，得重新建立。所以压缩不是“纯赚”——它拿“一次缓存重建的代价 + 一次摘要调用的代价”，去换“上下文不爆 + 后续每轮更短”。这笔账通常划算(否则会话根本没法继续)，但你要清楚自己在交易什么。精细的实现会尽量让压缩只折叠很老、本来也快过期的缓存段，保住近期的热缓存——这正是下一篇那些“外科手术式”精细化手段要解决的。

六、小结

自动压缩是上下文工程里“减负”的主力手段，也是有损、有风险、有代价的一步：

1. 保留近期、摘要远古，用一条压缩边界切开。
2. 摘要 prompt 决定成败——用固定结构强制保留“继续干活所必需”的信息，对抗“压缩即失忆”。
3. proactive 主动 + reactive 兜底，并给压缩本身留出余量。
4. 正视压缩击穿缓存的代价，想清楚这笔交易。

压缩是“大刀阔斧”——一下折叠掉几十条消息。但有时你想要更精细的控制：比如“只把那几个占了两万 token 的巨型工具结果删掉，别的都留着”。这种“外科手术式”的上下文管理，是下一篇的主题。

本篇对应代码：配套仓库加入 `src/compact.ts`(阈值触发的摘要压缩)，接入循环的每轮开头。`git checkout article-13`。

14 • 精细化上下文管理：外科手术，而非大刀阔斧

上一篇的自动压缩很有效，但它是钝器：一刀把边界前的所有历史摘要成一段，损失了全部细节，还击穿了缓存。很多时候，上下文膨胀其实只源于几个具体的“胖子”——你想要的是精准切除它们，而不是把整段历史推平。这一篇，讲三把外科手术刀。

一、背景：膨胀往往是“局部”的

观察真实 Agent 的上下文，你会发现膨胀高度不均匀：

- 大部分消息(用户的话、模型的推理、小工具结果)都不大。
- 但个别工具结果是巨无霸：一次 `read_file` 读进来一个五千行的文件、一条 `npm test` 刷了两千行日志、一次 `git diff` 几百个改动。单单几个这样的结果，就可能占掉上下文的大半。

对这种“局部肥胖”，用全量压缩是杀鸡用牛刀：你为了削掉那两万 token 的日志，把整段对话的细节全摘要没了，还废了缓存。更好的办法是精准地只处理那几个胖子。

二、三把手术刀

手术刀一：工具结果预算(超限就外置)

最前置的一招：给每个工具结果设一个大小上限。 还记得04 篇工具接口里那个 `maxResultSizeChars` 吗？就是干这个的。当一个结果超过上限：

1. 把完整结果写到外部(磁盘上一个临时文件)。
2. 只给模型一个截断预览 + 一个指针：“输出太长，已存到 `/tmp/xxx`，前 50 行如下……需要完整内容请用 `read_file` 读取。”

工具结果预算:超限则外置,只给模型预览 + 指针

```ts
function budgetResult(tool: Tool, fullText: string): string {
  if (fullText.length <= tool.maxResultSizeChars) return fullText
  const path = writeToTemp(fullText)                    // 完整结果存盘
  const preview = fullText.slice(0, tool.maxResultSizeChars)
  return `${preview}\n\n[输出过长,已截断。完整内容(${fullText.length} 字符)见 ${path}
}
```

> 这一招的精妙之处：信息没有丢，只是从“上下文里”挪到了“够得着的地方”。 模型依然知道有这么个结果、知道它的开头、知道完整版在哪——真需要细节时，它能自己去 `read_file` 取

回来。你用一个廉价的“指针”换掉了昂贵的“全文”，而且把要不要花上下文去看全文的决定权交还给了模型。这比一刀切地截断(直接丢掉后半截、模型永远够不着了)高明得多。

注意这又是04 篇“两个受众”的体现：给模型的是截断预览，给人的 UI 里仍可展示完整输出——人看到的和模型看到的，本就该是两个版本。

⚠ 一个反直觉的边界：有些工具的结果绝不能外置。典型就是 `read_file` 自己——如果它的输出也被“存盘 + 让你用 read_file 去读”，就成了一个 `read_file → 存盘 → read_file → ……` 的死循环。所以这类工具要把上限设成“无穷大”(它自己已经通过别的方式自我约束了)。设预算时，先想清楚这个工具的结果被外置后，会不会绕回来咬自己。

手术刀二：microcompact —— 按 `tool_use_id` 定点删除

第二把刀更精细：在不动对话其余部分的前提下，把某几个特定的、老的工具结果“就地掏空”。

关键抓手是 `tool_use_id`(02 篇那个配对 ID)。你可以精确地定位到“第 8 轮那次 `read_file` 的结果”，把它的 `content` 替换成一个小占位符("[早期输出已省略]")，而保持 `tool_use` / `tool_result` 的配对结构完好(不破坏历史合法性):

microcompact:只掏空指定的老结果,其余分毫不动

```
掏空前:...[tool_use:read_file][tool_result: <5000 行文件内容>]...
掏空后:...[tool_use:read_file][tool_result: "[早期文件内容已省略,如需请重读]"]...
          └─ 配对结构、id、位置全都不变,只有 content 被换小了 ─┘
```

ℹ 为什么值得这么费劲地“定点”？为了缓存。全量压缩重写了边界后的一切，缓存从边界起全废。而 microcompact 只改动特定几个、而且尽量是很老的 `tool_use_id`——如果被改的都在缓存前缀的很靠前处(本来也快过期)，对热缓存的破坏就降到最低。这是“减负”和“保缓存”两个目标之间的精细平衡：能定点切除，就别整段推平。挑“老的、大的、且已经不太可能再被用到的”结果下刀，收益最大、代价最小。

手术刀三：内容外置引用

第三把刀是前两把的一般化：大块内容根本不进上下文，而是活在外部，只在上下文里留一个句柄(handle)，模型按需拉取。

这适合那些“可能要用、但大概率不用全部”的大数据：一次大规模搜索的完整结果、一个大目录的完整清单、一份长文档。上下文里只放“有这么个东西，句柄是 X”，模型真要用时通过工具把它(或它的一部分)取回来。本质上，这是把上下文当“寄存器”，把外部存储当“内存”，按需换入换出——和操作系统管理内存的思路如出一辙。

三、一条统领性原则：用最小破坏换取足够空间

三把刀，加上全量压缩，构成一个从轻到重的减负梯度：

手段	破坏性	何时用
工具结果预算	最小(源头就截)	每个大结果产生时，预防性地
microcompact 定点删除	小(只动几个老结果)	上下文偏高，且膨胀源集中在几个胖子
内容外置引用	小(数据本就在外部)	已知会产生大数据的工具
全量压缩	大(推平一段 + 废缓存)	前几招都不够，历史整体太长

> ⚠ 原则：永远优先用能腾出足够空间的、破坏性最小的那一招。 别一上来就全量压缩——先看看是不是几个巨型工具结果在作祟，能定点切除就定点切除；实在是“整体都长了”，才动用压缩这把钝器。这个“阶梯式减负”的思路，是精细上下文管理的核心心法。一个成熟的 Agent 会把这几招组合起来，层层递进地维持上下文健康，而不是简单粗暴地一压了之。

四、第五阶段总结：上下文工程全景

第五阶段到此完整。 这是整个专栏最硬核的一章，值得把五篇串起来回望——它们共同回答了那个核心问题：在有限的窗口里，如何始终装着模型此刻最需要的东西？

篇章	在这场战争里的角色
10 Token 预算	侦察：量出用量，设水位线
11 上下文注入	编排：每轮的上下文是分层拼出来的
12 Prompt 缓存	省：稳定前缀复用，守住字节红线
13 自动压缩	减(钝器)：历史太长，折叠成摘要
14 精细化管理	减(手术刀)：定点切除胖子，少伤缓存

> ℹ 如果这一整章只留一句话：上下文是 Agent 最稀缺的资源，而管理它是一组互相拉扯的权衡——省(缓存)要字节稳定，减(压缩)必然破坏字节；要空间就得牺牲细节，要细节就得占空间。没有银弹，只有针对具体情形的、在多个约束间的通盘取舍。这也是为什么“上下文工程”是区分玩具 Agent 和生产 Agent 的分水岭。

五、下一站

上下文管住了，我们的 Agent 已经能撑起很长的任务而不崩。但真实世界还有另一类挑战我们一直在往后推：各种各样的失败——网络抖动、限流、模型过载、输出被截断。下一篇进入第六阶段：健壮性，看一个生产级 Agent 如何把“出错”当成常态来优雅应对。

本篇对应代码：配套仓库加入 `src/toolResultBudget.ts`(工具结果超限外置)，并给工具接口补上 `maxResultSizeChars`。`git checkout article-14`。

15 · 错误恢复与模型回退：把“出错”当成常态

前五个阶段，我们默认“调用模型”这一步总会成功。在生产环境里，这个假设每天都被打脸几十次。 这一篇进入第六阶段，正视一个真相：对一个真实 Agent 来说，出错不是意外，是常态。

一、背景：那行 `await` 会以十种方式失败

03 篇循环里那行朴素的“调用模型”，在真实世界里是最脆弱的一环。它会失败，而且花样繁多：

- 限流(429)：你请求太频繁，被限速了。
- 过载(529)：模型服务此刻扛不住，让你稍后再来。
- 超时 / 网络抖动 / 偶发 500：基础设施层面的临时故障。
- 上下文超长(413)：历史撑爆了窗口(第 13 篇的场景)。
- 输出被截断(`stop_reason: max_tokens`)：模型话没说完就撞到了输出上限。

一个让这些错误直接抛出的循环，是个玩具——它会在第一次网络抖动时当场暴毙。一个生产级 Agent，一大半的代码在处理这些失败路径。

二、核心分野：可恢复 vs 致命

处理错误的第一原则，是把它们分成两类，区别对待：

> 可恢复错误(限流、过载、超时、上下文超长、输出截断)→ 调整状态，重试或继续，用户最好无感。 致命错误(认证失败、请求格式错误、余额耗尽)→ 干净地、明确地告诉用户，别假装还能继续。
>
> 而且——错误应该被“产出”成一条合成消息，而不是被“抛出”。 让循环把错误作为一个正常的事件 `yield` 出去(UI 优雅显示、循环自己决定下一步)，而不是让异常冒泡炸穿整个调用栈。这样，无论哪种错误，系统都是“受控地应对”，而不是“失控地崩溃”。

三、四种可恢复错误的恢复姿势

① 临时故障：指数退避重试

限流、过载、偶发 5xx、网络抖动——这些是临时的，过一会儿再试往往就好了。标准解法是指数退避 + 抖动：

TS 指数退避重试

```ts
async function withRetry<T>(fn: () => Promise<T>, max = 4): Promise<T> {
  for (let attempt = 0; ; attempt++) {
    try {
      return await fn()
    } catch (err) {
      if (!isTransient(err) || attempt >= max) throw err
      // 退避:1s, 2s, 4s, 8s …… 再加随机抖动,避免所有客户端同时重试造成"惊群"
      const delay = 2 ** attempt * 1000 + Math.random() * 1000
      // 若响应带了 Retry-After 头,优先听它的
      await sleep(retryAfter(err) ?? delay)
    }
  }
}
```

两个细节别漏：抖动(jitter)防止大量客户端在同一时刻齐刷刷重试(惊群)；`Retry-After` 头(服务器明确告诉你“X 秒后再来”)要优先于你自己算的退避。

② 模型过载：降级回退

如果重试几次主模型仍然过载，还有一招：换一个模型。降级到一个次选模型(可能便宜些、能力稍弱，但此刻可用)，把请求重发：

TS 模型回退:主模型不行,换备用的

```ts
try {
  return await callModel(primaryModel, req)
} catch (err) {
  if (isOverloaded(err) && fallbackModel) {
    yield systemNote(`主模型繁忙,已切换到 ${fallbackModel}`)
    return await callModel(fallbackModel, req)   // 换个模型重试
  }
  throw err
}
```

“能用一个稍弱的模型把活干完”，几乎总比“卡在这儿什么都干不了”强。

③ 输出被截断：续写或提额

`stop_reason: max_tokens` 意味着模型话说到一半被硬切了。两条恢复路：

- 提额重试：如果这次用的是一个较小的 `max_tokens`，直接把上限调大、原样重试一次。
- 续写：注入一条“你被输出上限截断了，请直接接着刚才说，别道歉、别重述”的消息，让模型从断点续上。

> 续写那条消息的措辞很讲究：要明确告诉模型“从断的地方接着说，不要重新开头、不要为中断道歉”。否则模型往往会礼貌地“抱歉刚才没说完，让我重新说……”然后把已经生成的部分又来一遍，既浪费又可能前后矛盾。恢复消息的措辞，本身就是 prompt 工程。

④ 上下文超长：压缩再重试

`413` 就是第 13 篇说的 reactive 压缩场景：捕获它 → 压缩历史 → 重试刚才的请求。“上下文超长”是一种可恢复错误，这正是把压缩和错误恢复串起来的那根线。

四、重试的暗礁：孤儿消息

现在讲一个重试时极其隐蔽、又极其重要的坑。

> ⚠ 当你在流式途中失败、要重试时，那些已经流出来一半的“半成品”消息，必须先清理掉，否则它们会毒化重试。 想象：模型流到一半、已经吐了一个带签名的思考块或半个 `tool_use`，然后连接断了。如果你带着这半条消息去重试，会撞上两类问题：一是历史合法性被破坏(半个 `tool_use` 没有配对结果)；二是那些半成品的思考块签名是无效的，重发会被 API 直接拒绝。
>
> 所以重试/回退前，必须把这一轮产生的、还没“落定”的半成品消息作废清理(有的实现会给它们打上“墓碑”标记，从历史和 UI 里一并移除)。重试的前提，是先把上一次失败尝试的残骸打扫干净。

这也是为什么前面反复强调：只在拿到完整的模型响应后，才把它“提交”进历史。半成品在落定之前，永远是可丢弃的。

五、贯穿始终的红线：别写成死亡螺旋

所有恢复机制，都共享一个致命风险：

> ⚠ 恢复本身可能触发新一轮同样的失败，形成死亡螺旋。 几个真实的螺旋：
>
> - 压缩后上下文还是太长 → 再压 → 还是长 → ..... 无限压缩，烧光预算。
> - 对一个 API 错误消息去跑Stop 钩子 → 钩子注入更多内容 → 更长 → 又错 → .....
> - 续写截断的输出，结果每次续写又立刻被截断 → 无限续写。
>
> 所以每一条恢复路径都必须自带硬闸：最大重试次数、最大恢复次数、“这种错误已经试过一次就别再试”的标记。恢复逻辑里，“什么时候放弃恢复、干净地报错退出”和“怎么恢复”同等重要——甚至更重要。能优雅地失败，是比能恢复更高的要求。

六、小结

把“出错”当常态，是玩具 Agent 和生产 Agent 最扎眼的分界线之一。核心就几条：

1. 分清可恢复与致命；错误产出为事件，而非抛出为异常。
2. 临时故障 → 指数退避 + 抖动 + 听 Retry-After；过载 → 降级回退。
3. 截断 → 续写/提额；超长 → 压缩重试。
4. 重试前清理孤儿消息，只提交完整响应。

5. 每条恢复路径都要有硬闸，防死亡螺旋——优雅地放弃，也是一种能力。

我们的 Agent 现在能扛住真实世界的各种抖动了。但它每一次调用、每一次重试、每一次压缩，都在花钱。下一篇成本与用量追踪，我们把这些 token 换算成真金白银，并实时盯住。

本篇对应代码：配套仓库给 client 开启指数退避重试、加入模型回退，并把错误优雅地产出为事件而非抛出(`src/agent.ts`、`src/llm.ts`)。`git checkout article-15`。

16 · 成本与用量追踪：把 token 换算成钱

上一篇让 Agent 扛住了故障，但每一次调用、每一次重试、每一次压缩，都在烧钱。一个自主循环几十轮、还会派生子 Agent 的系统，成本能悄悄失控。这一篇，我们把 token 换算成真金白银，并实时盯住。

一、背景：四种 token，四种价格

很多人以为成本就是“token 数 × 单价”。大错。 一次调用的 `usage` 里其实有四类性质完全不同的 token，单价天差地别：

token 类型	相对价格(示意)	说明
输入(input)	1×	正常读进去的 token
输出(output)	~5×	生成的 token，最贵
缓存写(cache write)	~1.25×	首次建立缓存的一点溢价
缓存读(cache read)	~0.1×	命中缓存，便宜十倍

> ⚠️ 如果你把这四类混作一坨、按同一个单价算，你的成本估算会离谱地错。 一个开了Prompt 缓存的长会话，绝大部分“输入”其实是走 cache-read 的(便宜十倍)，真实成本可能只有“按输入全价估算”的零头。反过来，输出 token 比输入贵好几倍，一个话痨 Agent 的成本大头可能在输出侧。不分开统计这四类，你既不知道钱花哪了，也没法优化。

这也解释了缓存那一篇为什么是“省钱最立竿见影的一招”——它把最大宗的输入 token 从 1× 打到 0.1×。

二、解决方案：分价目表累加

思路很直接：一张按模型、按 token 类型的价目表，每次响应后把四类 token 各自乘以各自的单价，累加进会话总额。

按四类 token 分别计价

```ts
const PRICES = {
  // 单位:美元 / 每百万 token
  'some-model': { input: 3.0, output: 15.0, cacheWrite: 3.75, cacheRead: 0.30 }
}

function costOf(model: string, usage: Usage): number {
  const p = PRICES[model]
  return (
    (usage.input_tokens          * p.input) +
    (usage.output_tokens         * p.output) +
    (usage.cache_creation_tokens * p.cacheWrite) +
    (usage.cache_read_tokens     * p.cacheRead)
  ) / 1_000_000
}
```

每一轮响应回来，`addUsage(res.usage)` 累加一笔，会话总成本就实时可见了。

三、实现细节：三个别漏的当口

① 别漏掉“隐形”的调用

会花钱的不只是主循环那次调用。所有走了模型的地方都要计入，包括几个容易被忽略的：

- 压缩调用(第 13 篇)：压缩历史本身是一次模型调用，也花钱。
- 子 Agent(下阶段)：派生出去的子 Agent 从同一个预算池花钱。
- 各种“元”调用：生成标题、分类意图、辅助判断的小调用。

> 成本追踪要覆盖“所有出口”，而不只是主循环。 一个常见的错觉是“我盯着主循环的 usage 呢，成本没问题”——结果子 Agent 和压缩在背后花掉了大头，你的账单和你的估算对不上。正确的做法是把成本累加器做成会话级、贯穿所有调用路径的，主线程、子 Agent、压缩、元调用，谁调模型谁就往同一个累加器记一笔。

② 预算硬闸接回主循环

有了实时成本，就能设预算上限。这直接接回03 篇那个“循环要有闸门”的设计——成本超预算，是终止循环的合法理由之一：

成本预算作为循环的一道闸

```ts
while (true) {
  if (tracker.totalUsd() >= maxBudgetUsd) {
    return { reason: 'budget_exceeded' }   // 花超了,干净地停
  }
  // ...正常循环体...
}
```

回合数、token 预算、美元预算……这些闸门共同构成了03 篇说的“失控保护”。让 Agent 自主，和给它设一个花钱的天花板，从来不矛盾。

③ 实时可见，而非事后算账

成本要实时呈现给用户(跑着就能看到“这次会话已花 $0.42”)，而不是等会话结束才总算一笔。原因很实际：一个失控的 Agent(比如陷入某种循环)，用户能在花掉几十美元之前看到成本异常飙升，及时叫停(中断那一篇的能力在这里派上用场)。成本的可见性，本身就是一道安全阀。

四、小结

成本追踪不难，但有几个“想当然”会让你算错：

1. 四类 token 分开计价——混作一坨必然离谱，尤其在开了缓存之后。
2. 覆盖所有出口——压缩、子 Agent、元调用都花钱，别只盯主循环。
3. 预算硬闸接回循环，美元上限也是失控保护的一环。
4. 实时可见，让成本异常能被及时发现和叫停。

到这里，健壮性阶段还剩最后一块拼图。现代模型有一种特殊的能力——扩展思考(extended thinking)，它能显著提升 Agent 解决难题的能力，但也带来一组独特的、能让你调试一整天的坑。下一篇思考的处理，我们收尾第六阶段。

本篇对应代码：配套仓库加入 `src/cost.ts` (四类 token 分价目表 + 会话累加器)，在 usage 事件里带出实时成本。`git checkout article-16`。

17 • 扩展思考的处理：签名、模型绑定，和那些坑

第六阶段收尾。现代模型有一种特殊能力——扩展思考(extended thinking)：在给出答案前，先进行一段可见的、结构化的推理。它能显著提升 Agent 解决难题的能力，但也带来一组独特到反直觉的工程坑。

一、背景：多了一种你必须原样保管的内容块

开启思考后，模型的响应里会多出一类内容块：思考块(`thinking`)。它长这样：

响应里的思考块

```
{
  role: 'assistant',
  content: [
    {
      type: 'thinking',
      thinking: '让我先分析一下这个 bug……报错在 auth.ts:42,可能是……',
      signature: 'EqoBCkgI...(一长串加密签名)',    // ← 注意这个
    },
    { type: 'text', text: '我找到问题了,是……' },
  ],
}
```

注意那个 `signature`——它是模型对这段思考的加密签名，用来证明“这段思考确实是我在这个上下文里生成的、没被篡改过”。正是这个签名，是所有思考相关的坑的根源。

二、思考块的三条铁律

处理思考块，有三条不容违反的规则。违反任何一条，轻则报错，重则让你对着一个诡异的 400 调一整天。

⚠ ① 思考块必须整段、原样保留在历史里——包括那串签名，一个字节都不能改。 你不能“顺手把思考内容精简一下”再放回历史(签名会对不上，API 直接拒)；也不能只留 `text`、把 `thinking` 块丢掉(在一个包含工具调用的思考轨迹里，丢了思考块会破坏轨迹完整性)。思考块对你是只读的、不透明的——你只负责原样搬运，不许加工。

⚠ ② 思考块的签名和“生成它的那个模型”绑定。 这是最阴险的一条。签名是特定模型签发的。如果你把一段带签名的思考块，replay 给另一个模型(比如上一篇的模型回退触发了、切到了备用模型)，那个模型不认这个签名，请求当场 400。所以——一旦发生模型回退，必须先把历史里的思考签名块清理/剥离掉，再喂给新模型。 这两个特性(模型回退和思考)单独看都对，组合起来却会打架，而且症状(一个莫名其妙的 400)完全指不到根因上。

> ③ 一段“思考轨迹”必须完整保留，直到它彻底结束。 一次带工具调用的思考(思考 → 发起工具调用 → 拿到结果 → 继续)，这整条轨迹里的思考块都得留着，不能中途删。你不能刚思考完、工具还没跑完，就把思考块清掉——那会破坏这条尚未闭合的轨迹。

> 这三条规则合起来，可以浓缩成一句话：思考块是模型“working memory”的一次带签名的快照；你的唯一职责是完整、原样、在正确的生命周期内保管它，别碰它的任何一个字节。 想清楚这一点，大部分思考相关的坑就都能绕开了。(还有一种 `redacted_thinking` 块——出于安全被加密隐藏的思考，同样只需原样保管即可。)

三、和前面几个系统的相互作用

思考不是一个孤立特性，它和我们前面搭的好几个系统都有微妙的相互作用，值得逐一点一下：

- 和流式：思考内容也是流式到达的(有自己的 `thinking_delta`)。你可以实时把它展示给用户(让用户看到模型在“想什么”)，但要注意签名是在思考块结束时才给的——和工具参数那个“逐字到达、末尾才完整”的模式如出一辙。
- 和历史 / 上下文压缩：压缩历史时，思考块要么完整保留、要么整段移除(连同它所在的轨迹)，绝不能“部分保留”。半段带签名的思考块是历史里的一颗雷。
- 和模型回退：如前所述，回退前必须剥离思考签名。这是两个正确特性组合出的一个必须专门处理的交叉点。
- 和 `max_tokens`：思考也消耗输出 token，而且往往消耗不少。开了思考，你的 `max_tokens` 预算要相应放大，否则容易被截断。

四、实现细节：enable、保管、别加工

用起来，开启思考只是请求里加一个配置：

开启扩展思考

```ts
const stream = client.messages.stream({
  model, max_tokens: 16000,
  thinking: { type: 'enabled', budget_tokens: 8000 },   // ← 给思考留 8000 toker
  messages, tools,
})
```

难的从来不是“开启”，而是之后对思考块的保管纪律：

保管思考块:原样搬运,不加工

```ts
// ✅ 对的:把完整的 content(含 thinking 块及其签名)原样放回历史
messages.push({ role: 'assistant', content: res.content })

// ❌ 错的:过滤掉 thinking 块,只留 text —— 破坏轨迹,后续可能 400
messages.push({ role: 'assistant', content: res.content.filter(b => b.type !==

// ❌ 错的:精简 thinking 内容 —— 签名对不上,直接被拒
```

> 你会发现，思考的处理再次呼应了缓存那篇“别原地改要复用的消息”的纪律——凡是要回流给 API 的东西，保持字节原样。思考块把这条纪律推到了极致：它不仅要字节原样，还带着一个会“验真”的签名，和一个“认签名的”特定模型。把“原样搬运、按生命周期保管、别碰字节”当成处理这类内容的默认心法，你就能躲开绝大多数坑。

五、第六阶段总结

健壮性阶段到此完整。 这一章的三篇，让 Agent 具备了在真实世界里稳定运行的韧性：

篇章	韧性来自
15 错误恢复	把出错当常态：退避、回退、恢复、防螺旋
16 成本追踪	把花销看清：四类计价、覆盖所有出口、预算硬闸
17 思考处理	把特殊内容保管好：签名、模型绑定、生命周期

我们的 Agent 现在自主、快速、可控、可扩展、上下文管理精细、还扛得住真实世界的抖动。单个 Agent 能做的，它基本都会了。

最后一个阶段，我们把视野从“一个 Agent”扩展到“一个系统”：当一个 Agent 打不过时，就上一群(子 Agent)；当内置工具不够时，就接入外部工具生态(MCP)；以及，怎么把这一切持久化下来，支持关掉再回来接着干(会话历史)。进入第七阶段。

本篇对应代码：配套仓库为 `runAgent` 加入可选的扩展思考配置，并演示思考块的原样保管（`src/agent.ts`）。`git checkout article-17`。

18 · 子 Agent：把细节挡在主线程之外

最后一个阶段，我们把视野从“一个 Agent”放大到“一个系统”。第一步：当一个 Agent 忙不过来、或者会把自己的上下文搞脏时，派一个子 Agent 去。

一、背景：主上下文正在被细节淹没

设想主 Agent 接到一个任务：“这个项目的鉴权是怎么实现的？”为了回答，它得读十几二十个文件、搜一堆符号。问题来了——这二十份文件的完整内容，全都灌进了主 Agent 的上下文。等它终于弄明白、给出一句话结论时，主上下文已经被二十份文件的原文塞得满满当当。

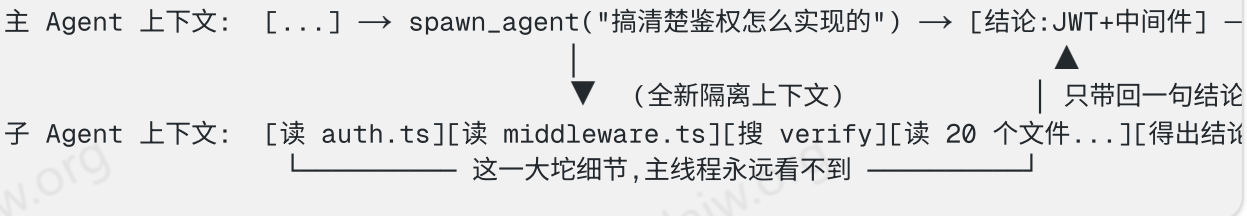
而这些原文，结论出来之后就没用了。主 Agent 接下来要干的事，只需要那句“鉴权用的是 JWT + 中间件校验”的结论，根本不需要那二十份文件的逐字内容。可它们已经赖在上下文里了，每一轮都在被重发、都在挤占窗口、都在逼近压缩。

> 这就是子 Agent 要解决的核心问题：大量的“中间探索细节”，不该污染主线程的上下文。 一个复杂任务的推进过程会产生海量中间信息(读过的文件、试过的命令、走过的弯路)，但其中绝大部分，在子任务完成后就是噪音。把这些细节关进一个隔离的上下文里，只让结论回到主线程——这是子 Agent 最根本的价值。

二、解决方案：子 Agent 就是“带上下文隔离的函数调用”

思路优雅得惊人：子 Agent 本身就是一个工具。 主 Agent 像调用任何工具一样“调用”它，只不过这个工具内部，是另一个完整的 Agent Loop，跑在自己全新、隔离的上下文里。

子 Agent:探索的细节留在子上下文,只有结论回到主线程



它和普通函数调用的类比精确得令人愉快：

函数调用	子 Agent
有自己的局部变量作用域	有自己隔离的上下文
内部的中间变量不泄露给调用者	探索细节不泄露给主线程
只返回一个结果	只返回一句结论
可以并行调用多个	可以扇出多个并行跑

> 子 Agent 是“上下文层面的封装”——正如函数把实现细节封装起来、只暴露返回值，子 Agent 把探索细节封装在独立上下文里、只暴露结论。理解了这个类比，你就理解了子 Agent 的一切：它不是什么高深的“多智能体协作”，它就是给 Agent 增加了“函数调用”这一层抽象，而封装的东西恰好是最宝贵的资源——上下文。

三、实现细节：一个工具，内部跑一个 Agent Loop

因为我们的 Agent Loop 本来就是一个可复用的 `runAgent` 函数，实现子 Agent 几乎是免费的——写一个工具，它的 `run` 内部递归调用 `runAgent`，消费完子 Agent 的所有事件，只把最后的结论作为工具结果返回：

子 Agent 工具:内部起一个隔离的 Agent Loop

```ts
const spawnAgentTool = defineTool({
  name: 'spawn_agent',
  description: '派一个子 Agent 独立完成一个探索性子任务,只返回它的最终结论。' +
    '当子任务会产生大量中间细节(读很多文件、大范围搜索)时用它,保持主线程上下文干净。',
  schema: z.object({ task: z.string().describe('交给子 Agent 的任务描述') }),
  isConcurrencySafe: true,    // 隔离的,可以扇出多个并行(见文章 06)
  async run({ task }, ctx) {
    let conclusion = ''
    // 关键:子 Agent 用一套不含 spawn_agent 的工具集,避免无限套娃
    for await (const ev of runAgent(task, { cwd: ctx.cwd, signal: ctx.signal, t
      if (ev.type === 'assistant_text_delta') conclusion += ev.text
    }
    return { modelText: conclusion, display: `子 Agent:${task.slice(0, 40)}…` }
  },
})
```

就这么点。主 Agent 的 `runAgent` 里，工具执行器跑这个工具，而工具内部又是一个 `runAgent`——一个自然的递归结构。子 Agent 那一大坨探索，全程在自己的 `messages` 数组里，主线程的 `messages` 完全不知情，只收到最后那段 `conclusion`。

两个必须处理的点

> ① 防无限套娃。 子 Agent 如果也能 `spawn_agent`，就可能无限递归下去。最简单的防法：给子 Agent 一套不含 `spawn_agent` 的工具集(上面 `tools: baseTools`)，让递归深度天然只

有一层。更灵活的做法是传一个“深度”计数，到达上限就不再提供 spawn 工具。总之，必须有一个东西终止这个递归，否则一个坏指令能让它派生出成千上万个子 Agent。

② 扇出并行。注意 `isConcurrencySafe: true`——因为每个子 Agent 上下文隔离、互不干扰，主 Agent 可以在一轮里同时派出好几个(“同时搞清楚鉴权、路由、数据库三块怎么实现”)，靠第 06 篇的并发执行器并行跑，大幅压缩总时长。这是子 Agent 的一个额外红利：它天然适合把可并行的探索扇出出去。

四、代价与权衡

子 Agent 不是白来的，用之前想清楚：

- 信息损失：主线程只拿到结论，拿不到细节。如果主 Agent 后面其实需要某个中间细节，它就得让子 Agent 重新去查，或者自己去查。所以子 Agent 适合“结论自足”的子任务，不适合“细节还要反复用”的场景。
- 额外成本：子 Agent 是一整个独立的 Agent Loop，有自己的多轮调用开销，从同一个预算池花钱。派一个子 Agent 去读一个文件是杀鸡用牛刀——它值当的前提是“探索量大到会显著污染主上下文”。
- 可观测性：子 Agent 的中间过程主线程看不到，调试时你得能单独查看子 Agent 的轨迹(所以真实系统常把子 Agent 的对话记到一个独立的“支线”文件里，既不污染主会话，又可供事后审查——这和下一篇的持久化相关)。

判据：当一个子任务会产生“大量、且用完即弃”的中间细节时，派子 Agent；否则主 Agent 自己干。别为了“看起来像多智能体”而滥用子 Agent——每一个都是实打实的成本和延迟。它是一把用来保护主上下文的手术刀，不是拿来炫技的。

五、小结

子 Agent 是把上下文这个最稀缺资源，用“函数调用式封装”保护起来的机制：

1. 核心价值是上下文隔离——探索细节留在子上下文，只有结论回主线程。
2. 实现上，它就是一个工具，内部递归跑一个 `runAgent`，几乎免费。
3. 必须防无限套娃(限工具集/限深度)，并可扇出并行。
4. 代价是信息损失、额外成本、可观测性——结论自足、探索量大时才值当。

主 Agent 会用工具、会派子 Agent 了。但到目前为止，它的工具都是我们内置写死的。真实世界里，用户想接入五花八门的外部工具(数据库、浏览器、公司内部 API)，而且可能有几百个。下一篇MCP 与工具懒加载，我们看 Agent 如何接入一个开放的外部工具生态，以及“几百个工具”这个规模问题怎么解。

本篇对应代码：配套仓库加入 `src/tools/spawnAgent.ts`(子 Agent 工具，内部递归 `runAgent`，用不含自身的工具集防套娃)，并给 `runAgent` 加 `tools` 覆盖。`git checkout article-18`。

19 • MCP 与工具懒加载

到目前为止，我们 Agent 的工具都是内置写死的。但真实世界里，用户想接入的工具五花八门——查公司数据库、控浏览器、调内部 API、连第三方 SaaS——而且可能有几百个。这一篇解决两个问题：如何开放地接入外部工具，以及几百个工具怎么不把 prompt 撑爆。

一、背景：内置工具的两道墙

墙一：封闭

内置工具意味着，想加一个新工具，就得改 Agent 的源码、重新发布。用户没法接入你没预见到的工具。一个只能用内置工具的 Agent，能力边界被开发者一次性钉死了。

墙二：规模

就算能动态加工具，还有个规模问题：每一轮请求都要把所有工具的 schema 发给模型。十几个工具还好，可要是有几百个呢？

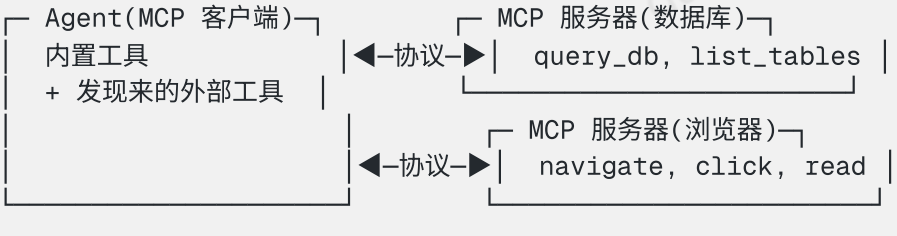
- token 爆炸：几百个工具的完整 schema，可能就是几万 token，每轮重发，又贵又占窗口。
- 模型犯晕：面对几百个工具，模型的选择准确率会下降——太多相似的选项反而让它挑花眼、选错。

二、解决方案一：MCP —— 外部工具的标准协议

MCP(Model Context Protocol) 是给“外部工具”定的一套标准协议。它的核心思想：

> *“把“工具的提供方”和“Agent”解耦。任何人都可以起一个MCP 服务器，用标准协议暴露一组工具/资源；任何Agent 都可以作为MCP 客户端去连接它、动态发现它有哪些工具、并调用它们——双方事先互不需要了解。”*

MCP:Agent 作为客户端,连接外部工具服务器



Agent 启动时连上用户配置的若干 MCP 服务器，动态发现它们的工具，把这些工具和内置工具一起，呈现给模型。用户想加能力，只需再连一个 MCP 服务器，完全不用改 Agent。

MCP 能这么顺畅地融入，全靠第 04 篇那个 `Tool` 抽象。一个 MCP 工具，和一个内置工具，在 Agent 眼里长得一模一样——都是“名字 + 描述 + JSON Schema + 一个 `call` 方法”。区别只在于：内置工具的 `call` 是本地函数，MCP 工具的 `call` 是“把参数按协议发给远端服务器、等结果回来”。只要把外部工具包装成同一个 `Tool` 接口，Agent Loop、权限、并发编排全都不用改，自动就能驱动它。这就是当初把工具设计成一个统一抽象的巨大回报——它让“接入任意外部工具”变成了“实现一个适配器”。

用代码看，接入一个外部工具，就是把它的描述符适配成我们的 `Tool`：

任何外部工具 → 同一个 Tool 接口

```
function externalToolToTool(spec: ExternalToolSpec, callRemote: CallFn): Tool {
  return {
    name: spec.name,
    description: spec.description,
    schema: jsonSchemaToZod(spec.inputSchema),
    isReadOnly: false, isConcurrencySafe: false, isDestructive: false, // 外部工
    run: async (input, ctx) => {
      const result = await callRemote(spec.name, input)   // ← 唯一的区别:call 
      return { modelText: result.text }
    },
  }
}
```

包装完，它进入工具注册表，和内置工具平起平坐。MCP 本质上就是这个适配过程的标准化、协议化版本。

三、解决方案二：延迟加载 + 工具搜索

解决了“接入”，再解决“几百个工具塞爆 prompt”。关键洞察是：

> *“模型在某一轮里，通常只需要几百个工具中的几个。没必要把全部 schema 每轮都塞给它。”*

于是有了延迟加载(deferred tools)+ 工具搜索的机制：

1. 初始只发工具的“名字 + 一句话简介”，不发完整 schema。这一下就把几万 token 压到几千。
2. 提供一个元工具 `search_tools`——模型需要某类能力时，用它按关键词搜索，系统才把匹配到的那几个工具的完整 schema 加载进上下文。
3. 模型拿到完整 schema 后，才真正调用它们。

延迟加载:按需把工具 schema

```
初始 prompt:  [完整内置工具] + [几百个外部工具:仅名字+简介] + [search_tools 元工具]
模型:         search_tools("数据库 查询")
系统:         → 加载 query_db、list_tables 的完整 schema 进上下文
模型:         query_db({ sql: "..." })   ← 现在才有它的完整定义可用
```

> 这本质上是把上下文当缓存、外部当内存的“按需换入”思想用到了工具上：工具定义不全量常驻上下文，而是用到才加载。它同时解决了 token 爆炸(只加载用到的)和模型犯晕(初始只看到简介、需要时才聚焦到具体几个)两个问题。代价是多一次“搜索”的往返，但换来的是“几百个工具”这个规模下的可用性。

四、小结

这一篇让 Agent 的能力从“开发者钉死的内置集合”变成了“用户可无限扩展的开放生态”：

1. MCP 用标准协议解耦工具提供方与 Agent，让接入外部工具无需改 Agent。
2. 一切的基础，是第 04 篇那个统一的 `Tool` 抽象——外部工具只是 `call` 走远端的普通工具。
3. 延迟加载 + 工具搜索解决“几百个工具”的规模问题：按需把工具 schema 换入上下文。

我们的 Agent 现在几乎无所不能了。但它还缺最后一块，一块让它能“活得久”的东西：持久化。到目前为止，一切都在内存里——进程一关，整个会话灰飞烟灭。下一篇，也是本专栏的收官篇，会话历史与持久化：如何把这一切落盘，支持关掉再回来接着干。

本篇对应代码：配套仓库加入 `src/tools/external.ts`，演示把一个外部工具描述符适配成统一的 `Tool` 并动态注册。`git checkout article-19`。

20 · 会话历史与持久化：让 Agent 活过进程

本专栏的收官篇。我们的 Agent 已经近乎无所不能，但它还缺最后一样东西——它活不过自己的进程。关掉终端，整个会话灰飞烟灭：干到一半的任务、积累的上下文、宝贵的探索结果，全没了。这一篇，我们给它落盘。

一、背景：内存是易失的

回到序章那个比方，以及01 篇那三个“裸 LLM 没有”。我们一路补上了“能行动”(工具)和“能自主推进”(循环)，但第三样——记忆/状态——我们只在内存里补了：那个 `messages` 数组一直在进程内存里滚。

内存是易失的。进程一退，`messages` 就没了。这带来几个真实痛点：

- 不能续接：昨天让 Agent 重构一个大模块，干了一半，今天想接着干？对不起，重来。
- 不能回看：上周那次会话它是怎么解决那个 bug 的？没记录。
- 不能撤销：改错了想退回三步前？内存里没存过去的状态。

要让 Agent“活过进程”，就得把会话持久化到磁盘。

二、解决方案：append-only JSONL

持久化会话，业界的主力方案朴素而可靠：把每一条消息，作为一行 JSON，追加写到一个 JSONL 文件里。

会话 JSONL:一条消息一行,只追加

```
{"role":"user","content":"重构 auth 模块"}
{"role":"assistant","content":[{"type":"text","text":"我先看看现状"},{"type":"too
{"role":"user","content":[{"type":"tool_result","tool_use_id":"...","content":"
{"role":"assistant","content":[...]}
...
```

为什么是“append-only(只追加)”这个形式？它有几个恰到好处的性质：

append-only JSONL 几乎是为 Agent 会话量身定做的：

- 崩溃安全：每条消息一产生就立刻追加落盘。哪怕进程下一秒崩了，已经写下的部分也都在——你不会丢掉整个会话，最多丢最后没写完的一条。
- 流式友好：消息是一条条产生的，追加写和这个节奏天然契合，不用每次重写整个文件。
- 可重放：恢复会话 = 把 JSONL 一行行读回来，重建 `messages` 数组，然后接着循环。抄本本身就是完整的、可回放的历史。

落盘与恢复,对称得令人安心

```ts
// 落盘:每次往历史加一条消息,就追加一行
function append(file: string, message: Message) {
  appendFileSync(file, JSON.stringify(message) + '\n')
}

// 恢复:把每一行读回来,就重建了整个历史
function load(file: string): Message[] {
  return readFileSync(file, 'utf-8').split('\n').filter(Boolean).map(l => JSON.
}
```

三、四个实现细节

① 抄本即真相，压缩只是运行时视图

一个关键的设计决定：落盘的是压缩前的完整历史，还是压缩后的？ 答案是——落盘完整的，压缩只作为运行时的一层视图。

> JSONL 抄本是“真相”，它记录发生过的一切、逐字、永不折叠。而压缩/精简是运行时为了塞进窗口而做的临时投影。 这样分工的好处：恢复会话时你能拿回完整历史(而不是一个有损摘要)，然后按需重新应用压缩。抄本保真，运行时求存——两者各司其职。如果你把压缩后的有损版本当成真相存下来，那些被摘要掉的细节就永久丢失了，再也找不回来。

② 大块内容外置

有些消息里嵌着大块内容(用户粘贴的一大段日志、一张图)。把它们原样塞进 JSONL会让抄本臃肿、难读。常见做法：小的内联存，大的按哈希外置——JSONL 里只留一个引用(`[粘贴内容 #3,+240 行]`)，内容本体存到旁边一个按哈希命名的文件。这和第 14 篇工具结果外置是同一个思路：大 blob 不进主流，只留句柄。

③ 并发写要加锁

别忘了子 Agent和后台任务——它们可能并发地往会话相关文件写。两个进程同时追加同一个文件，可能交错损坏。所以真实实现会用文件锁(lockfile) 来串行化写入。(子 Agent 的抄本通常还会写到独立的支线文件，既不和主会话抢锁，又能单独审查——呼应第 18 篇。)

④ 撤销：append-only 之上的巧思

append-only 不能“删”，怎么实现撤销？一个巧妙的做法：用一个“待落盘缓冲 + 跳过集合”——最近的、还没刷盘的操作放在内存缓冲里(撤销它们很快)；已经刷盘的，则用一个“跳过集合”标记为“已撤销”，重放时略过。在一个只追加的底座上，叠加逻辑上的可撤销，是持久化设计里一个漂亮的小技巧。

四、全专栏回望：我们究竟做了什么

会话持久化，恰好补上了01 篇那三个“裸 LLM 没有”的最后一个。让我们回到起点，看看这一路究竟把那块“裸芯片”，一件件焊成了什么：

裸 LLM 缺的	我们补的	在哪补的
不能行动	工具调用	第 02、04 篇
不会自主推进	Agent Loop	第 03 篇
无状态、易失	上下文管理 + 持久化	第五阶段 + 本篇

而在这三根主梁之外，我们还焊上了让它真正能用、能扛、能长大的一切：

- 让它快：流式、边流边执行与并发。
- 让它安全可控：权限、中断、钩子。
- 让它在有限窗口里精打细算：token 预算、上下文注入、缓存、压缩、精细管理。
- 让它扛得住真实世界：错误恢复、成本追踪、思考处理。
- 让它能扩展、成规模：子 Agent、MCP 与懒加载、持久化。

> 回到序章那个核心问题：一个只会“输出文本”的无状态函数，是怎么变成一个能自主完成复杂编程任务的 Agent 的？ 现在你有了完整的答案：不是靠某一个魔法，而是靠外面这一整圈、几十个环环相扣的工程决策。 模型提供智能，但把智能变成“能自主、安全、持久、经济地在真实世界里干活的系统”，全在模型之外这一圈。这一圈，没有一处是黑魔法，每一处都是扎实的、可以理解、可以复刻的工程。

五、结语：芯片与机器

我们从一块“什么都不会”的裸芯片(LLM)出发，一篇一篇，焊上了内存、总线、驱动、操作系统，最后拼出了一台能自主干活的完整机器(Agent)。

如果这趟旅程只留下一句话，那就是序章那句的回响：

> *“LLM 只是块芯片，Agent 才是整台机器。而这台机器的每一个零件，你现在都亲手拆过、装过一遍了。”*

配套的完整代码仓库里，这台机器从第一篇的几十行，长到了现在的一个麻雀虽小五脏俱全的真实 Agent。`git checkout` 到任意一篇，你都能把那个阶段的它跑起来。读懂它，然后——去造你自己的那一台。

感谢一路读到这里。

本篇对应代码：配套仓库加入 `src/session.ts`(append-only JSONL 落盘 + 恢复),CLI 支持关掉再回来接着聊。`git checkout article-20` ——这是完整版。

道雾轩

《从 LLM 到 Coding Agent》

本电子书由道雾轩制作,免费分享,欢迎转发给需要的人。

阅读全部专栏,请访问官网 daiw.org。