

2026年10月更新版

Loop Engineering

别再问我什么是循环工程

从一句句喂prompt，到设计让agent自己干活的系统

The New Layer Above the Harness

文档版本：v261020

发布时间：2026-10-20 (build #0)

涵盖内容：定义 · 演化 · 解剖 · 六零件 · 评判器 · 真实案例 · 代价 · 上手

花叔

花叔

本手册包含的产品信息、功能描述和定价可能随时变化，请以各产品官方文档为准。

目录

CONTENTS

第一部分 · 它是什么

§01 别再问了：Loop Engineering到底是什么

§02 三级跳：从Prompt到Context到Loop

第二部分 · 它怎么转

§03 一个循环的五个动作

§04 六个零件：搭一个Loop需要什么

§05 生成器与评判器：为什么写代码的AI不能给自己打分

第三部分 · 它在哪跑、要什么代价

§06 让循环在你睡觉时跑：Stripe的Minions，和调度怎么选

§07 代价：验证债、理解债、认知投降、token失控

第四部分 · 你怎么开始

§08 当工程师，不只是按下启动键

§09 今天就动手：搭你的第一个Loop

§01 别再问了：Loop Engineering到底是什么

Stop Asking: What Loop Engineering Really Is

又一个XX Engineering。这一次的词讲的是：你退到循环外面去。

又来一个

Prompt Engineering的课还在卖，Context Engineering的墨迹没干，Harness Engineering我写过一本书。现在Loop Engineering又来了。

过去一年「XX Engineering」冒出来一串，造词速度快赶上模型迭代。你想翻白眼，我理解。

但这一个有点不一样。前面几个词都还假设你坐在键盘前，一句句指挥agent。Loop engineering把这个假设删掉了：你从干活的位置上退出来，在循环外面，负责造那个循环。

一句话定义

命名这个词、把它写成文章的人，是Addy Osmani。他在Google待了十四年多，先做Chrome的开发者体验，写这篇文章时是Google Cloud AI的总监；9月他去了Anthropic，做Claude Code。他给的定义很短：

原文：Loop engineering is replacing yourself as the person who prompts the agent. You design the system that does it instead.

翻成中文：循环工程，就是把那个负责prompt agent的人从你自己换成一套系统。你不再亲自一句句喂，你设计那套替你喂的系统。你写的东西，从给agent的话，变成了一个会自动给agent发话的东西。

同一段里他紧接着写了两句保留：这事还早，他自己是怀疑的（I'm skeptical），token成本得非常小心。

一周之内，三个人说到一起

这个词是2026年6月那一周，几拨人几乎同时撞到了同一件事，然后才有了名字。

传播最广的一条，是Peter Steinberger（OpenClaw的作者）6月7日发的推（UTC时间，东八区是6月8日凌晨），截至10月20日有850万浏览。

原文：Here's your monthly reminder that you shouldn't be prompting coding agents anymore. You should be designing loops that prompt your agents.

每月提醒一次：你不该再去prompt编程agent了，你该去设计那些会prompt你agent的循环。「monthly reminder」说明这话他不是第一次说，Addy文章里引的也是同一意思的另一个措辞，前面没有这句「每月提醒」。

几乎同时，**Boris Cherny**（Anthropic的Claude Code负责人）也在说同一件事。这句我能追到的出处是Addy文章里的引用，没找到他本人的原帖。

原文：I don't prompt Claude anymore. I have loops running that prompt Claude and figuring out what to do. My job is to write loops.

我不再prompt Claude了。我让一堆循环跑着，由它们去prompt Claude、去琢磨该干什么。我的工作，就是写循环。

给它命名的是**Addy Osmani**。6月7日他在个人博客把这件事写成文章，标题就叫Loop Engineering，顺手把Steinberger和Cherny的话都引了进去，第二天同步到Substack，6月9日又发了一版X长文，截至10月20日约240万浏览。从几个人各自在说，到有了名字，前后就一周。

三个人说的是同一个动作：你设计的对象，从agent的一次行为，变成了驱动agent的整个系统。就像当年的vibe coding，有了名字，一种做法才变得能讨论。

从「你prompt agent」到「你设计prompt agent的系统」

旧世界：你坐在那儿，一句句prompt agent。它干完一件，停下，等你下一句。你是这个循环里的人肉时钟，每一拍都得你来敲。

新世界：你设计一套东西，让它自己一拍一拍地敲。它到点自己醒，自己分出小帮手去干活，自己把结果喂给下一轮。你在循环外面看着它转。Addy管这一层叫「harness的上一层楼」（§ 02）。

6月之后

6月之后能追到一手出处的几件事：

时间	谁	做了什么
6月30日	Anthropic官方博客，Delba de Oliveira、Michael Segner	《Loop engineering: Getting started with loops》，给了厂商自己的定义和四类分法
7月9日	Addy Osmani	X长文《Own the Outer Loop》：agent跑内循环，工程师守外循环（\$08会用到）
7月17日	IBM Think	写了一篇《What Is Loop Engineering?》科普
8月22日	arXiv 2608.21884，Lulla、Treude、Baltes等五位	一篇探索性的预印本研究（审稿中），挖了36,710个仓库
9月	Addy Osmani	加入Anthropic，做Claude Code

Anthropic那篇的定义：

原文：loops as agents repeating cycles of work until a stop condition is met

循环，就是agent一轮轮重复干活，直到满足某个停止条件。Addy的定义讲人的位置，Anthropic的定义讲循环什么时候停。

他们还按「你把哪一样交出去」把循环分成四类：

类型	你交出去的是	Claude Code里对应
Turn-based（按轮）	检查：每轮你还在，但不用你逐行看	普通对话
Goal-based（按目标）	停止条件：跑到条件满足为止	<code>/goal</code>
Time-based（按时间）	触发：到点自己醒	<code>/loop</code> （本机）、 <code>/schedule</code> （云端）
Proactive（主动）	prompt本身：它自己决定该干什么	<code>/schedule</code> 、 <code>/goal</code> 、skills、dynamic workflows、auto mode组合

那篇论文是去GitHub上数了一遍。作者先读了一圈网上关于loop的文章，归纳出这些文章推荐的做法：有触发器拉起的agent运行、机器能判断的停止条件、持久化的状态文件、验证用的子agent、token预算、明确的转人工节点。然后去GitHub上挖，启发式规则匹配出256个仓库，人工确认其中217个真有自主循环在跑。这些仓库都提交了循环的配置，但几乎没有一个提交了那些文章里反复强调的状态文件。

§02 三级跳：从Prompt到Context到Loop

The Triple Jump: From Prompt to Context to Loop

四层栈，一层管一件事。Prompt、Context、Loop是三次跳，Harness夹在Context和Loop中间，是那块起跳板。

四层栈

过去两年AI圈造的那串XX Engineering，是一层层叠起来的，每层管一件更大的事。

层	管什么	核心问题
Prompt engineering	写好一次的提示词	我该告诉模型什么
Context engineering	这一刻窗口里放什么	检索什么、摘要什么、清掉什么
Harness engineering	单次运行的武装	给哪些工具、允许哪些动作、什么算完成
Loop engineering	在harness之上调度	怎么让它自己一遍遍跑起来

从一句话，到一个窗口，到一次运行，最后到一整套自动跑的循环。

第一层：Prompt

最底层，也是大家最熟的。你斟酌措辞、给例子、定角色、调语气。它的边界很清楚：就是这一次，你递给模型的那段文字。递进去，模型吐出来，结束。

这层的麻烦在于，它假设你每次都在场，每次都亲自递。一个prompt解决一件事，下一件还得你再写一个。

第二层：Context

往上一层，问题从「我说什么」变成「此刻这个窗口里，该装哪些东西，模型才解得动这道题」。检索什么资料、把长文摘成什么样、哪些过时信息要清出去。它管模型当下能看到的整个视野。

这层之所以单独成一层，是因为模型的窗口有限，塞错东西比塞少东西更糟。给它一窗户的噪音，再好的提示词也白搭。

第三层：Harness

再往上，到harness engineering。这层管的是agent跑一次，身上得带什么装备：给它哪些工具、允许它碰哪些动作、何时加载上下文、失败了怎么收场、什么状态才算完成。

这层我在另一本书里整本展开过。它把agent的单次运行武装好，但不负责让这次运行自动重来。

第四层：Loop

前三层做完，你手里有了一个装备齐全、能漂亮跑完一次的agent。但它跑完就停了，等你再敲下一拍。Loop这层，把「等你敲」这件事也自动化掉。

原文： Loop engineering sits one floor above the harness. The harness but it runs on a timer, it spawns little helpers, and it feeds itself.

循环工程，坐在harness的上一层楼。还是那套harness，只不过它在定时器上跑，会孵化小帮手，还会自己喂自己。

上一层楼多了什么

Addy那句话里有三个动词。

在定时器上跑。 harness武装的是一次运行，得你来触发。loop给它装了个定时器，到点自己醒、自己开工。

孵化小帮手。 一个loop转起来，会自己分出子agent去干并行的活，比如一个起草修改，另一个专门挑刺审查。harness关心单个agent怎么武装，loop关心一群agent怎么协作、谁监督谁。

自己喂自己。 loop跑出来的结果，会被它自己当成下一轮的输入。昨天发现的问题写进一个文件，今天醒来先读这个文件，接着干。它有了跨越单次对话的记忆。

Addy的原话是：自动化才让一个loop成为真正的loop，而不只是你手动跑过一次的某次运行（原文见 § 04零件一）。

Anthropic 6月那篇博客的四类分法，也能套在这层楼上：按轮、按目标、按时间、主动，差别在于你把检查、停止条件、触发、prompt本身里的哪一样交给了系统（§ 01有表）。

为什么要分这么清

因为每一层的失败方式不一样，拦它的检查也得装在不同的地方。

prompt错了，你当场就看见，改一句就行。context装错了，模型答偏，你也容易察觉。到了loop这层，它在你睡觉时自己跑、自己改你没看过的代码、自己把错误也一并喂给下一轮，出问题你可能几天后才发现。

§03 一个循环的五个动作

The Five Moves of One Loop

把一个loop转一圈拆开看，里面只有五个动作：发现、交付、验证、持久化、调度。

循环这个词容易让人想偏，一听就以为是一段反复执行的代码，转个不停。loop engineering里的循环每转一圈，都在干一件具体的活：发现一件值得做的事，把它交给agent，验证结果对不对，把状态存下来，然后决定下一步。少一个，loop就转不起来，或者转起来也是空转。

一个例子

Addy在文章里描述了一种他常用的形态，原话是「Here is one shape I keep using」：一个每天早上在仓库上跑的triage（分诊）loop。

早上一个automation自动跑起来。它调用一个triage skill，去读昨天CI跑挂的测试、还开着的issue、最近的commit，把分诊结果写进一个markdown文件或者Linear看板。每个值得动手的发现，开一个隔离的worktree。一个子agent起草修复，第二个子agent对照项目的skill和测试做审查。connector自动开PR、更新ticket。处理不了的进收件箱等人。状态文件留着，第二天接着这里跑。

动作一：发现

loop转一圈，第一件事是搞清楚这一圈该干什么。在Addy的例子里，这一步是triage skill干的：CI失败、open issues、最近commits，合起来就是昨天到今天系统里出了哪些值得管的事。

要让agent自己去找活。你要是每天早上手动列一遍今天修哪几个bug，那loop还是靠你启动的，没省下你。

Addy强调，automation里调用的是skill，不是一大段指令：你触发一个 `$skill-name`，别往一个没人会去更新的定时任务里贴一大面墙的指令。发现逻辑变了，改skill就行。

发现这一步做得好不好，决定了整个loop的质量上限。它找出来的活要是没价值，后面四个动作都白做。

动作二：交付

发现了活，得把它交到干活的agent手里，这一步叫handoff。Addy的做法是，每个值得做的发现单独开一个隔离的worktree。worktree是git的一个机制，让多个agent在各自的工作目录里改代码，互不踩脚（§ 04零件二）。

交付之后，真正干活的是子agent。一个起草修复，写出第一版改动。任务切得越干净，后面验证和合并就越省事：一个发现对应一个worktree，对应一份明确的改动，谁出问题都好定位。

动作三：验证

第一个子agent起草修复之后，第二个子agent来审查。它对照的是项目的skill和测试，指令跟第一个不一样，有时候连模型都换一个。写代码的那个agent给自己的作业打分时太手软（原话见 § 04零件五），换一个专门挑刺的，才抓得住第一个自我说服放过去的问题。

Addy在讲子agent的那一节写得很直接：

原文： the loop runs while you are not watching, so a verifier you actually trust is the only reason you can walk away.

循环是在你不看的时候跑的，所以一个你真信得过的验证者，是你能走开的唯一理由。用我自己的话说：验证这一步，就是往循环里放一个能说「不」的东西（§ 05）。

动作四：持久化

验证通过，结果得落到一个能活过这次对话的地方。Addy用的是两样东西：改动本身通过connector自动开PR、更新ticket，落进代码仓库和工单系统；处理不了的发现进triage收件箱，等人工。还有一个贯穿始终的状态文件，记着转到哪了。

上下文一清空，agent就忘干净了，写进markdown或者看板的东西不会忘。没有持久化，loop每转一圈都像第一次睁眼；有了它，今天没干完的，明天能接着干。

§ 01那篇arXiv论文发现，真在跑循环的仓库几乎都没把状态文件提交进版本控制。我读这句的理解是：持久化必须有，但不一定非得是一份commit进仓库的markdown；要紧的是下一轮醒来时能读到它，而且你自己也能读到它。

动作五：调度

Addy的triage loop早上自动跑，不用他每天去点；状态文件让今天没处理完的发现留到明天，明天那一圈接着这里开始。调度，把「我手动跑了一次」变成了「它自己一圈一圈在转」。

没有调度，就得你每天记着去跑一遍，哪天忘了就断了。

五个动作一张表

动作	干什么	在triage loop里
发现	自己找出这圈该做的事	skill读CI失败、issue、commit
交付	把任务隔离着交给agent	每个发现开一个worktree
验证	换个agent说「不」	第二个子agent对照测试审查
持久化	把状态写到对话之外	开PR、收件箱、状态文件
调度	让它一圈圈自动转	早上automation自动跑

§04 六个零件：搭一个Loop需要什么

Six Parts: What It Takes to Build a Loop

Addy的原话是「A loop needs five things and then one place to remember stuff」，五样东西，再加一个记东西的地方，我把它合成六个零件。

零件和动作对得上：发现靠skill，交付靠worktree，验证靠子agent，持久化靠memory，调度靠automation。

零件一：Automations（自动化/调度）

Automation是那个让loop自己动起来的東西。它挂在一个时间表或者触发器上，到点了就自动发现、自动分诊，不用人在旁边盯着按。它排第一，因为它是「循环」这个词的根。

原话： Automations are what make a loop an actual loop and not just one run you did once.
自动化，才是让循环成为真正循环的东西，而不只是你跑过一次的那一次。

automation里该触发的是一个skill，不是一整面墙的指令。你贴一大段prompt进定时任务，没人会再去更新它；触发一个有名字的skill，逻辑变了改skill就行。

调度的形态不止一种：有的跑在你自己机器上，机器得开着；有的跑在云端，机器关了也照转。怎么选在§06。

零件二：Worktrees（git worktree隔离）

Worktree是git自带的机制，让你在同一个仓库里开出多个互相独立的工作目录。每个agent在自己的worktree里改代码，互不干扰。loop一旦同时跑多个agent，文件写冲突就是迟早的事。

原话： Two agents writing the same file is the exact same headache as two engineers committing to the same lines and nobody talked to each other first.
两个agent同时写一个文件，跟两个工程师事先谁也没打招呼、就往同几行代码上提交，是一模一样的头疼。

这个零件的价值随并行规模放大。一个loop同时只跑一个agent，worktree可有可无；你想让它早上同时修五个bug，五个agent各改各的，没有隔离就是五份改动搅在一起，最后谁也不知道哪行是谁写的。

零件三：Skills（用SKILL.md复用知识）

Skill是把项目知识固化成一个文件（SKILL.md），让agent每次都能直接取用，不必每一圈都重新推一遍上下文。

不固化，你就得在每次运行里重新告诉agent这个项目是怎么回事、规矩是什么、坑在哪。这种反复交代的成本，Addy有一篇文章专门讲，叫意图债（intent debt）。把一次想清楚的东西写下来，之后每圈都省掉重新推导。

零件四：Plugins & Connectors（MCP连接器）

Connector是让loop接到外部世界的接口。issue tracker、数据库、staging环境的API、Slack，都靠它连进来，技术底座是MCP（Model Context Protocol）。

原话： A loop that can only see the filesystem is a tiny loop.

一个只能看见文件系统的循环，是个很小的循环。

Codex和Claude Code都走MCP，同一个MCP服务器两边都能挂。持久化那一步里，connector自动开PR、更新ticket，就是这个零件在收尾。只接文件系统，它能看的就是几个目录；接上issue tracker，它知道还有哪些活没干；接上Slack，它能把结果告诉你。每多接一个外部系统，loop能自主完成的链条就长一截。

零件五：Sub-agents（生成者和评判者分开）

子agent的关键用法，是把写东西的和评判东西的拆成两个。一个agent负责生成，另一个指令不同、有时模型也不同的agent负责挑刺。同一个agent既当运动员又当裁判，裁判会偏心。

原话： The model that wrote the code is way too nice grading its own homework. A second agent with different instructions and sometimes a different model catches the stuff the first one talked itself into.

写代码的那个模型，给自己的作业打分时太客气了。换一个指令不同、有时模型也不同的agent，能抓住第一个自我说服放过去的东西。

Anthropic Labs团队的Prithvi Rajasekaran在做长时间运行的应用时也得出过同一个结论：把一个独立的评判者调得足够挑剔，比让生成者对自己的作品狠下心来，容易得多（§ 05）。

零件六：Memory（持久化状态）

第六个零件，Addy在正文里把它叫the memory，他的表格里这一行写作State。它是loop活在单次对话之外的记忆，一份markdown文件，或者一块Linear看板。agent的上下文窗口一清空，它就什么都不记得了，loop要想今天接着昨天跑，记忆就得落到磁盘上。

原话： the model forgets everything between runs so the memory has to be on disk and not in the context. The agent forgets, the repo doesnt.

模型在两次运行之间什么都会忘，所以记忆得待在磁盘上，不能待在上下文里。agent会忘，仓库不会。

上下文是agent这一轮能看到的東西，会随着窗口刷新被冲掉。memory是写在磁盘上的东西，跨轮、跨天都还在，需要时再读回来。

六零件速查表

零件	是什么	对应动作	一句话原话
Automations	挂在时间表、触发器上自动跑	调度	make a loop an actual loop
Worktrees	隔离并行agent的工作目录	交付	same headache as two engineers
Skills	固化项目知识、还意图债	发现	fire \$skill-name instead of a wall of instructions
Connectors	MCP接外部系统	持久化、发现	only see the filesystem is a tiny loop
Sub-agents	生成者与评判者分离	验证	too nice grading its own homework
Memory	磁盘上的持久状态	持久化	the agent forgets, the repo doesnt

§05 生成器与评判器：为什么写代码的AI不能给自己打分

Generator and Evaluator

让写代码的agent评价自己的产出，它多半会夸。这一章讲怎么在循环里放一个独立的评判器，以及Claude Code和Codex已经把它做成了哪个命令。

它总夸自己

你让一个agent写完一段代码，再让它评价自己写得怎么样。它会怎么回答？

Anthropic Labs团队的Prithvi Rajasekaran在一篇官方工程博客里写过他的观察，场景是做长时间运行的应用：

原文： when asked to evaluate work they've produced, agents tend to respond by confidently praising the work—even when, to a human observer, the quality is obviously mediocre.

让agent评价自己产出的东西，它往往会自信地夸一通，哪怕在人看来质量明显很一般。

写代码的那个上下文里，已经塞满了它为什么这么写的理由。它回头看自己的代码，看到的是一路推导过来的自我说服。自己写的文章回头读，也是越读越顺。

放到loop里，这个毛病会被放大。loop值钱，是因为它能无人值守地跑很多轮。如果每轮这一版行不行都由刚写完的agent自己拍板，跑得越久，攒下的自我表扬越多，离真实质量越远。

改一个怀疑论者，比改一个谦虚的作者容易

Rajasekaran的结论是：

原文： tuning a standalone evaluator to be skeptical turns out to be far more tractable than making a generator critical of its own work.

调一个独立的评判器，让它去怀疑，比让生成器自我批判要容易得多。

在prompt里加一句请严格一点，作者也跳不出自己的视角。换一个agent，给它完全不同的指令，让它从零开始看这段代码，它没参与写，就没有那套自我说服。

这套思路他说借鉴的是GAN：

原文： I designed a multi-agent structure with a generator and evaluator agent.

生成对抗网络（GAN）里，一个网络负责造，一个网络负责挑刺，两个对着干，造的那个越来越像真的。搬到 agent 上，就是一个 generator 写，一个 evaluator 审，结构上把写和判断写得好不好分开。Addy 那句「写代码的模型给自己批作业太客气」，说的是同一件事（原话见 § 04 零件五）。

评判器要会动手，不只是会读

光换个 agent 还不够。evaluator 如果只是把代码读一遍，它判断的还是代码看起来对不对，没碰到跑起来对不对。

Rajasekaran 给 evaluator 接上了 Playwright MCP，让它像真人 QA 那样去用这个东西。前端实验里，评判器自己导航页面、截图；后面的全栈实验里，它像用户一样把应用点一遍，测 UI 功能、API 端点和数据库状态。

判断的依据就从读代码觉得没问题，变成了点过按钮、看过页面、截了图。

写 evaluator 指令时，社区里常见一种往最坏处校准的说法：默认「这段代码是坏的，除非被证明能跑」（assume the code is broken until proven otherwise）。这句是社区里的说法，不是官方的，方向和 Rajasekaran 一致：评判器的默认态度是怀疑。

落到产品：/goal 把评判器用在了停止条件上

Claude Code 里有一个命令把这个结构做成了产品原语，叫 /goal，2.1.139（2026年5月）加进来的。

用法是给 agent 一个条件，让它一直跑到条件满足为止。关键在由谁来判定满足了没有。官方文档的写法是：

官方文档： After each turn, a model checks whether the condition holds. If the model judges it not yet met, Claude starts another turn instead of returning control to you. ... completion is decided by a fresh model rather than the one doing the work.

每跑完一轮，一个模型来检查条件成立没有。它判断还没满足，Claude 就再跑一轮，不把控制权交还给你。是否完成，由一个全新的模型判定。这个模型默认是你配置里那个小而快的模型，底层实现是一个只在本会话生效、基于 prompt 的 Stop hook。

10月我在一个小仓库里实际跑了一次。仓库是让 agent 搭的练手仓库，里面只有一个 calc.py，留了两个 bug：加法写成了减法，判断偶数写成了判断奇数。测试文件有三条测试，跑起来两条失败。然后我输入：

```
/goal python3 -m unittest -v 显示全部测试通过 (OK)，并且只改 calc.py、不改测试文件，or stop after 6 turns
```

```
claude — demo-repo

Claude Code v2.1.287
Sonnet 5.5 · Claude Max
~/Documents/写作/10-橙皮书/Loop-Engineering/实测-202610/demo-repo

> /goal python3 -m unittest -v 显示全部测试通过 (OK), 并且只改 calc.py、不改测试文件, or stop after 6 turns
└─ Goal set: python3 -m unittest -v 显示全部测试通过 (OK), 并且只改 calc.py、不改测试文件, or stop after 6 turns

• 目标收到: 让测试全部通过, 只改 calc.py。先跑一下测试看现状。

• Running python3 -m unittest -v 2>&1 | tail -40; cat calc.py
└─ $ python3 -m unittest -v 2>&1 | tail -40; cat calc.py

+ Zigzagging... (13s · ↓ 107 tokens)

◎ /goal active (12s)

>

▶▶ auto mode on (shift+tab to cycle) · esc to interrupt · ← for agents
```

Claude Code 2.1.287, 模型Sonnet 5.5。回车后先回显一行Goal set, 右下角出现◎ /goal active和计时, 它立刻开始第一轮: 先跑测试看现状。中间空白的终端行我裁掉了。

24秒, 一轮就结束了。它改了 `calc.py` 里的两处, 跑测试, 3条全过, 然后汇报git status里只有`calc.py`是被追踪文件的改动。最后一行是评判器给的结论: Goal achieved。

```
claude — demo-repo

4
5 def is_even(n):

• Update(calc.py)
└─ Added 1 line, removed 1 line
3
4
5 def is_even(n):
6 - return n % 2 == 1
6 + return n % 2 == 0
7
8
9 def clamp(x, lo, hi):

Ran 1 shell command

• 目标达成: python3 -m unittest -v 3 个测试全部通过, 输出 OK。

只改了 calc.py 里两处, 没动测试文件:
- add: a - b 改成 a + b
- is_even: n % 2 == 1 改成 n % 2 == 0

git status 里只有 calc.py 是被追踪文件的改动, 另外两个 __pycache__/ 目录是运行测试产生的, 本来就未追踪。

✓ Goal achieved (24s · 1 turn · 731 tokens)

* Baked for 24s · done 8:40

> commit 这次修改

▶▶ auto mode on (shift+tab to cycle) · ← for agents
```

Goal achieved (24s · 1 turn · 731 tokens)这一行出自评判器。

按Ctrl+O打开详细记录, 能看到评判器给出的理由:

```
claude — Ctrl+O 详细记录

只改了 calc.py 里两处，没动测试文件：
- add: a - b 改成 a + b
- is_even: n % 2 == 1 改成 n % 2 == 0

git status 里只有 calc.py 是被追踪文件的改动，另外两个 __pycache__/ 目录是运行测试产生的，本来就没追踪。

✓ Goal achieved (24s · 1 turn · 731 tokens)
Goal: python3 -m unittest -v 显示全部测试通过 (OK)，并且只改 calc.py、不改测试文件，or stop after 6 turns
Reason: Transcript shows test output 'Ran 3 tests in 0.000s\n\nOK' indicating all tests passed. Git status shows only 'M calc.py' as modified tracked file (test files untouched). Condition satisfied: tests pass AND only calc.py modified.

* Baked for 24s · done 8:40

Showing detailed transcript · ctrl+o to toggle · ⌕ scroll · v to open in vi · ? for shortcuts verbose
```

测试输出里有Ran 3 tests ... OK，git status里只有M calc.py。两个条件，它各找了一条证据。

理由的第一个词是Transcript。评判器自己不跑命令、不读文件，它只能看干活的agent在对话里亮出来的东西。文档原话是「It doesn't run commands or read files independently, so write the condition as something Claude's own output can demonstrate」。

所以条件要写成对话里能被证明的样子。我写了只改calc.py，它就去跑了一次git status把证据摆出来；你要是写「代码质量要好」，评判器在对话里找不到能证明它的东西，就只能凭感觉。

文档里还写了几条：

- **三种判定**：没满足（继续跑）、满足（清掉目标，记一条achieved）、不可能（评判器认定条件永远达不到，自动清掉并写明原因）。
- **给它一个刹车**：在条件里直接写 `or stop after 20 turns` 这样的轮数或时间条款，评判器每轮按对话判断。我上面那条写的是6轮。
- **手动停**：`/goal clear`，别名stop、off、reset、none、cancel都行；条件最长4,000字符；恢复会话时还没达成的目标会跟着恢复。
- **评判的花费很小**：我这次实测的/usage里，Sonnet干活花了0.59美元，小模型那一行是0.054美元。小模型那一行里不只有评判器，Claude Code的一些后台杂活也走它，单独拆不出来。

```
Claude Code 2.1.287 · /usage

Settings Status Config Usage Stats

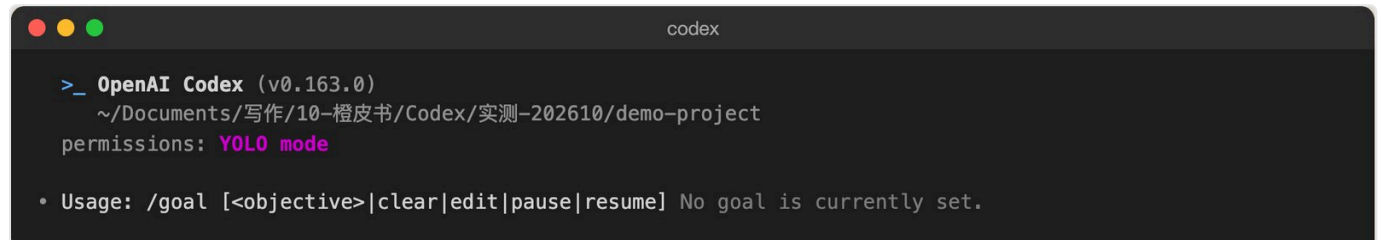
Session

Total cost:          $0.64
Total duration (API): 1m 30s
Total duration (wall): 7m 23s
Total code changes:  7 lines added, 4 lines removed
Usage by model:
  claude-sonnet-5-5:  3.6k input, 3.8k output, 1.8m cache read, 45.5k cache write ($0.59)
  claude-haiku-4-5:   923 input, 95 output, 0 cache read, 26.4k cache write ($0.0541)
Prompt cache (main): 21 requests · 97% of input tokens from cache · no misses · warm (1h TTL, last activity 41s ago)
```

这次/goal加/loop实测的整场会话账单。Total cost是按API价折算的参考数，我用的是订阅。

maker-checker（生产者-检查者）是银行的老规矩：大额转账，录入的人和复核的人得是两个人。经手的人会信任自己的操作，复核才有意义。/goal把这条规矩塞进了agent的循环里，每一轮干完，都得过一道不是它自己的检查。

Codex也有/goal，命令名一样。它是0.128.0（2026年4月30日）加进来的，现在支持设置、查看、edit、pause、resume、clear，我本机跑 `codex features list`，goals一项标着stable。



```
codex
>_ OpenAI Codex (v0.163.0)
  ~/Documents/写作/10-橙皮书/Codex/实测-202610/demo-project
permissions: YOLO mode

• Usage: /goal [<objective>|clear|edit|pause|resume] No goal is currently set.
```

Codex CLI 0.163.0里直接输入/goal，会打印用法：设置目标，或者clear、edit、pause、resume。

别把/loop当成这个。/loop是按时间间隔重跑，属于调度，跟跑到条件满足为止是两码事。

§06 让循环在你睡觉时跑：Stripe的Minions，和调度怎么选

Stripe's Minions, and Where a Loop Runs

一个公开、能追溯到一手出处的大规模案例，加上几种调度方式各自靠什么、能不能在你关机后继续跑。

Stripe的Minions：每周1,300多个PR

把loop推到企业规模，最值得看的是Stripe的Minions。

数字先放这：每周合并1,300多个PR，全部由Minions产出，人审过，但没有一行是手写的。这是Stripe工程师Alistair Gray在Stripe官方博客上两篇长文里写的，第一篇2026年2月9日，第二篇2月19日；第一篇时还是1,000多个。这是2月的数，之后的我没查到。

触发方式很轻。一次典型的运行从一条Slack消息开始，到一个CI通过、等人审的PR结束，中间不用人插手。除了Slack，命令行、网页界面也能发起，Stripe内部的文档平台、feature flag平台、工单系统也都接了它。比如CI发现一个不稳定的测试，会自动开一张工单，工单上就有一个按钮，点一下让Minion去修。

LLM醒来之前，系统先用确定性的代码把上下文备齐：消息里有看起来相关的链接，就先把对应的MCP工具跑一遍，把内部文档、工单详情、构建状态、Sourcegraph代码搜索的结果拉进来。Stripe为此建了一个内部的中心MCP服务器，叫Toolshed，第一篇写的是400多个工具，第二篇已经是将近500个。

让LLM自己去找上下文，它可能搜错、可能漏、可能为了凑答案现编。Stripe把找资料这种能写死规则的活，从LLM手里拿走，交给确定性代码去办。

它的agent内核是Block开源的goose的一个fork，2024年底分出来，之后按Minions的需要一路改。

第二篇里有个概念叫blueprint（蓝图）。它把工作流写成代码，像一台状态机：有些节点是确定性的，只跑代码不调模型，比如「跑配置好的linter」「推送改动」；有些节点交给agent自由发挥，比如「实现任务」「修CI失败」。agent写完代码，linter一定会跑，它跳不过去。CI最多跑两轮：第一次推送没过，让Minion修一次再推，还不过就交还给人。

沙箱用的是devbox，就是Stripe工程师自己写代码用的那种AWS EC2机器。一池子devbox提前预热好，10秒内能起一台；它们跟生产环境和互联网隔离，所以Minion在里面跑不需要人逐条批权限。Stripe用的是运维那句老话：devbox是「cattle, not pets」，是一群随时能换的牛，不是养着的宠物。坏了不修，直接起一台新的。

§ 04讲并行靠git worktree，Stripe在第一篇里明说，worktree这种做法「wouldn't scale at Stripe」，他们靠的是一任务一台devbox。worktree是目录级的隔离，devbox是机器级的隔离。个人和小团队用worktree够了，规模大到Stripe这样，隔离单位就从目录变成了机器。

环节	谁来做
触发	人（Slack、命令行、网页、内部工具里的按钮）
备上下文	确定性代码先跑一遍MCP工具（非LLM）
写代码、修CI	agent节点
跑linter、推送	确定性节点（agent跳不过）
审这1,300个PR	人

最后一行，这1,300个PR仍然由工程师审，时间从写挪到了审。

关机以后靠什么跑

本地的/loop也好，桌面的定时任务也好，它们都要你的机器开着。机器一关，loop就停了。要关机也跑，得靠别的东西：Cloud Routines，或者GitHub Actions的schedule触发。

	Cloud Routines	Desktop定时任务	/loop
跑在哪	Anthropic云	你的机器	你的机器
需要开机吗	否	是	是
需要开着会话吗	否	否	是
最小间隔	1小时	1分钟	1分钟
怎么建	claude.ai/code/routines，或会话里 /schedule	桌面版里新建	会话里 /loop
能看本地文件吗	否（fresh clone）	能	能

要频繁、要看本地文件，用本地/loop，代价是机器得开着。要关机也跑、不依赖本地状态，上云，代价是最小间隔一小时、每次都是干净的clone。

举两个场景。你想让一个loop每分钟盯一下本地正在跑的开发服务器，那只能是本地/loop，云端看不到你机器里那个进程，间隔也下不到一分钟。反过来，你想让它每天凌晨三点扫一遍仓库的open issue、该提PR提PR，这种活不该绑在你的笔记本上，笔记本会合盖、会断电、会被你带出门。这时候用GitHub Actions的schedule触发或者Cloud Routines，它们跑在一台一直醒着的机器上。

本地/loop适合你在场时让它替你多跑几轮，云端调度才管你不在场的时候。

Cloud Routines现在仍标着research preview（研究预览），要用claude.ai的订阅账号登录才有 /schedule；触发器除了定时，还能是一个API调用或者GitHub事件；每个账号定时运行每小时最多100次。

排程本身没什么神秘。按文档，`/loop`给了间隔就换算成标准的cron表达式，五个字段，最小粒度一分钟；循环任务7天后自动过期，最后触发一次然后删掉自己；嫌它烦，设个 `CLAUDE_CODE_DISABLE_CRON=1` 就全关了。它还有一点抖动：循环任务可能比整点晚到，最多晚间隔的一半、不超过30分钟。

不过我10月实际跑的时候，看到的跟文档不一样。我在Claude Code 2.1.287里输入 `/loop 1m` 加一句检查待办的prompt，它说循环已启动，每1分钟触发一次，30分钟后到期。我问它用什么机制实现的，它先调了一次 `CronList`，结果是 `No scheduled jobs`，然后告诉我：这个循环是用Monitor工具起的一个后台shell脚本，每60秒往外打一行带prompt的标记把它唤醒，设了30分钟超时。



```
Claude Code 2.1.287 · 问它循环怎么实现的

● 目前只有一个循环在跑。

1. 每分钟查 TODO.md 的循环
  - 机制：用 Monitor 工具起的后台 shell 循环，任务 ID 是 b4fv0hm1a。脚本是 while true; do sleep 60; echo 'AGENT_LOOP_TICK_todo {...}'; done。
  - 工作方式：每 60 秒往 stdout 打一行带 prompt 的标记。这一行会作为通知把我唤醒，我再按里面的 prompt 去读 TODO.md。
  - 到期时间：监控设了 30 分钟超时，大约在启动后 30 分钟自动结束，到期会通知我，需要的话再续。
  - 停止方法：想提前停，让我用 TaskStop 终止它。
2. 其他机制，目前都没有在跑
  - Cron: CronList 返回「No scheduled jobs」，没有用 CronCreate 建过任务。
  - ScheduleWakeup: 没用过。这个只服务于不带间隔的动态 /loop，你的 /loop 带了 1m，走的是固定间隔。
  - /goal 的 Stop hook: 条件是「测试全部通过且只改 calc.py」，达成后已自动清除，不再生效。

补充一点：这个循环只存在于当前会话里。会话关闭后它就没了，不会像系统 cron 那样在后台继续跑。

* Sautéed for 11s · done 8:46 · 1 monitor still running

› 停掉这个循环

>> auto mode on · 1 monitor · ← for agents · ↓ to manage
```

它自己说的实现方式。文档里Monitor只出现在「不给间隔、让Claude自己定节奏」那种`/loop`里，我给了1m，它还是用了Monitor。

文档写的是7天过期的cron任务，我这一次（2.1.287，Sonnet 5.5）看到的是30分钟超时的后台脚本，你的机器上会出现哪一种，我没法保证。所以起完一个`/loop`，问它一句用什么机制跑的、什么时候到期，别默认它会替你跑满7天。

Codex的定时任务叫Scheduled tasks，归在ChatGPT桌面app里，Codex就在这个app里面。每个任务可以选在本地项目里跑，也可以开一个新的worktree跑；结果出现在侧栏的Scheduled里；自定义周期用的是日历标准里的RRULE写法，比如 `RRULE:FREQ=MONTHLY;BYMONTHDAY=1;BYHOUR=9;BYMINUTE=0`，不是cron。Codex CLI和IDE扩展里没有这个管理界面。想关机也跑，有ChatGPT网页和手机端的定时任务（碰不到你电脑上的文件夹），还有按Gmail、Slack、GitHub事件触发的任务，这一种只在网页和手机端、符合条件的套餐里有，不能和定时同时设。另外，Codex Cloud任务也能在你电脑睡着时继续干活，它是另一套功能，不归Scheduled tasks管。这块我只读了文档，没在桌面app里实际建过。

有几个流传很广的数字，你看到时最好谨慎对待。比如「Anthropic约九成Claude Code由它自己写」「Nubank迁移百万行ETL提效12倍」这类，多是二手综述，我没追到一手出处。

§07 代价：验证债、理解债、认知投降、token失控

The Costs

循环替你干活，也在替你欠账。这一章讲四笔：验证债、理解债、认知投降、token失控，以及工具里已经有的几道上限。

循环自己跑，也在自己犯错

原文： A loop running unattended is also a loop making mistakes unattended.

一个没人看着的循环，也是一个没人看着犯错的循环。

第一笔账叫验证债。你让循环自动开PR、自动改代码、自动合并，每一步都省了你的时间。省下来的时间变成了一堆还没人验证过的产出，堆在那儿等你还。

这笔债不会立刻找你。比如一个循环一晚上改了很多文件，测试全绿，PR全开，看起来漂亮极了。问题藏在测试没覆盖到的地方，藏在能跑和对之间的缝里，攒到某个上线的早上一起冒出来。

Addy在同一段里接着写：

原文： The whole reason you split the verifier sub-agent from the maker is to make the loop's "its done" mean something, and even then "done" is a claim and not a proof.

把验证的子agent和干活的分开，全部意义就在于让循环说的「做完了」有点分量；即便这样，「做完了」也只是一个声明，不是证明。

防它的办法 § 05讲过：装一个跟干活的不是同一个的评判者，让它会动手验证，把完成条件写成能被证明的样子。剩下那部分「声明」和「证明」之间的距离，得人来补。

你以为你懂这个repo，其实你不懂了

第二笔账更隐蔽，Addy管它叫理解债（comprehension debt）。

循环每天替你写一堆你没写过的代码。代码能跑、测试能过、PR能合，表面上一切正常。背后慢慢变质的，是你对这个项目的理解。

原文： The faster the loop ships code you did not write, the bigger the gap between what exists and what you actually get.

循环交付你没写的代码越快，实际存在的东西和你真正理解的东西之间，差距就越大。这句前面Addy还有一句：Your understanding still rots if you allow it，你允许的话，你的理解照样会烂掉。

读代码比写代码无聊，而循环把写的部分全包了。你不再被迫一行行读过去，自然就不读了。代码库在长大，你脑子里的那张地图却停在三个月前。等出事的那天，你打开文件，像在看别人的项目。

这个差距平时不报警。项目跑得好好的，你也觉得自己挺懂这套东西，直到某个bug钻进一个你从没读过的角落，循环修不动了，得你亲自上。

防它，就得定期读循环的产出，别只看测试过了就合。挑几个改动，自己讲一遍它干了什么、为什么这么改。讲不出来，就是你的地图该更新了。

循环跑顺了，人就懒得有意见了

第三笔账是前两笔的心理版本，Addy叫它认知投降（cognitive surrender）。

原文：When the loop runs itself its very tempting to stop having an opinion and just take whatever it gives back.

当循环自己跑起来，你会很想停止有自己的判断，它给什么你就收什么。

循环越可靠，你越容易把判断这件事整个外包出去。持续有判断很累，每个PR都要自己想一遍对不对，比直接点合并累多了。循环跑得越稳，你就越倾向于相信它，看得越少，判断的能力就越生疏。

循环可以替你执行，主意还得你拿。它递回来的东西，你至少得有能力说一句「这个不对」。判断不必每次都改，但能力得在。

账单可能比你想象的厚很多

最后一笔是直接打在账单上的钱，叫token失控。一个自主跑的循环，消耗有多少，很难提前算准。

原文：you absolutely have to be careful about token costs (usage patterns can vary wildly if you are token rich or poor)

你必须非常小心token成本。括号里那半句是说：用量会剧烈波动，取决于你是token富人还是token穷人。同一个循环，富人跑得起，穷人可能一夜醒来发现账单爆了。这句在Addy文章开头那一段，紧跟着「这事还早，我是怀疑的」。

你写的是逻辑，跑出来的是次数，次数乘以单价才是钱。一个bug让它空转一整晚，你睡醒看到的是一张陌生的账单。

上线前给循环设几个硬上限，到顶就停。工具里已经有现成的开关：

上限	怎么设
轮数	Claude Code的 <code>/goal</code> 条件里直接写 <code>or stop after 20 turns</code> ，评判器每轮按对话判断；无人值守用 <code>claude -p</code> 跑时，加 <code>--max-turns</code>
花费	<code>claude -p</code> 加 <code>--max-budget-usd</code> ，子agent的花费也算在里面，到顶后不能再派新的子agent，还在跑的后台子agent会被停掉
时长	本机定时循环有过期时间：按文档cron任务7天；我实测那个Monitor循环是30分钟
子agent	同时在跑的子agent默认最多20个（ <code>CLAUDE_CODE_MAX_CONCURRENT_SUBAGENTS</code> ，2.1.217起），嵌套默认最多3层；一个会话累计派多少个，已经不设上限
看账	<code>/usage</code> 按模型拆开这场会话的花费，也按skill、子agent、MCP拆最近的用量； <code>/goal</code> 不带参数会显示已经跑了几轮、用了多少token
Codex	<code>/goal</code> 遇到用量上限会自动暂停；0.148.0起可以给goal配token预算；0.151.0起，嵌套子agent的token也算进根目标的预算

Anthropic 6月那篇《Getting started with loops》也专门列了省token的建议，我觉得最实用的两条是：确定性的活用脚本做，别让模型做；检查的间隔要跟被检查的东西变化的频率对上。它还说，模型和推理强度的选择，是决定一个循环花多少钱的几根最大的杠杆之一。

§08 当工程师，不只是按下启动键

Stay the Engineer

同一个循环，两个人造，结果可以截然相反。

同样的循环，相反的结局

原文：Two people can build the exact same loop and get completely opposite results. One uses it to move faster on work they understand deeply. The other uses it to avoid understanding the work at all. The loop doesn't know the difference. You do.

两个人造出一模一样的循环，得到的结果可以完全相反。一个人用它，在自己吃透的活上跑得更快；另一个人用它，好让自己压根不用去理解这份活。循环分不出这两种人，你分得出。

一个人用循环，是为了在自己已经吃透的事情上跑得更快。代码他读得懂，方向他拿得准，循环只是替他把手动的活包了。循环帮他扩大的，是他本来就有的判断。

另一个人用同样的循环，是为了不必再去理解。看不懂没关系，循环会写；判断不了没关系，循环会合。他用循环把理解这件麻烦事整个绕过去了。

不稀缺的和稀缺的

循环让一件事变得极其便宜：生成。代码、方案、PR、修复，都能批量造出来，几乎不要钱、不要等。AI写代码已经很便宜了。

那什么还稀缺？

判断力。知道哪个方案是对的、哪行代码该拦下来、哪个产出虽然能跑但根上错了。

循环能生成一百个选项，也可以替你选，但它选的依据是看起来合理。它把所有不需要判断的活都拿走了，剩下的全是判断。

过去你判断失误，顶多自己手写错一段代码，影响有限，还来得及发现。现在你判断失误，循环会批量地、不知疲倦地把这个错误执行一百遍，它不会停下来问你「这真的对吗」。

循环不懂你为什么造它

还有一层，关于谁来设计这个循环。

循环会忠实地执行你给的逻辑。但它不懂你当初为什么要造它，不懂你真正想要的是什么，不懂哪些地方你其实是想自己把关的。

循环的边界在哪、哪里该留一个人工的卡点、哪里该让它自己跑，这些决定没人能从循环里读出来。它们只活在造它的人的脑子里，得被事先想清楚、再一条条写进去。漏掉的那部分，循环不会替你补，它会照着你写的跑，连同你忘了写的那些缺口一起跑。

如果你抱着赶紧把自己解放出去的心态造它，你会把检查都省掉，因为检查碍事。如果你还打算在这儿当工程师，你会主动给它留几道你能说不的关口，哪怕那意味着它慢一点。

内循环交给agent，外循环归你

7月9日，Addy在X上发了一篇长文，标题叫《Own the Outer Loop》，截至10月20日有130万浏览，接着Loop Engineering那篇往下讲。

他的分法是：agent跑内循环，调查、实现、验证、再来一遍；工程师守外循环，也就是对这套系统负责。为了把负责说具体，他用了三个词：

词	他的意思
Quality（质量）	放系统出去之前装上的所有检查。检查会产出证据
Verdict（裁决）	工作进入下游系统之前那个最终决定：放行、拦下、改道、收窄范围、加护栏，还是直接拒绝
Answerability（可解释）	有人问起时，你能说清楚为什么

裁决那一条，他的原话是：

原文： The model may write the line, but the Verdict is mine.

代码可以是模型写的，但放不放行，是我的决定。

这篇长文里他也列了三项隐性成本：认知投降、认知债（cognitive debt），还有一项新的，他叫编排税（orchestration tax）：现在开一堆agent很容易，但你的注意力没法跟着并行。认知债和 § 07 的理解债不是同一个词：理解债说的是代码和你的理解之间的差距，认知债说的是你自己怎么解决问题的理解和记忆在流失。

他在文末写了一句：

原文： An agent can write it. But before it reaches users, someone must explain why it should exist, why it's safe enough to be part of production, and what they will do when it is wrong.

agent可以把它写出来。但在它到达用户之前，得有一个人说得清：它为什么该存在，为什么它安全到可以上线，以及它出错时你打算怎么办。

造循环，但要像一个打算留下来的人

Addy在Loop Engineering那篇里，写两个人那段之前，先给自己降了降温：去搭你的循环吧，但直接prompt你的agent也照样有效，关键是找到平衡（prompting your agents directly is also effective. It's all about finding the right balance）。

文章结在这一句：

原文： Build the loop. But build it like someone who intends to stay the engineer, not just the person who presses go.

造那个循环。但要像一个打算继续当工程师的人去造它，而不是一个只负责按下启动键的人。

§09 今天就动手：搭你的第一个Loop

Build Your First Loop Today

从一行 `/loop` 起步，一层层加上分诊、状态文件、评判器、worktree和云端调度，最后给一张检查清单。

别想着一步到位

很多人一听loop engineering，脑子里立刻浮现Stripe那种每周1,300多个PR的流水线，然后觉得离自己太远，放弃了。

Stripe那套是终点。第一个loop小到几乎不像个系统就行，一个会定时帮你看一眼的小东西。

我的建议是，从一个 `/loop` 定时任务开始。它是Claude Code自带的命令，门槛最低，跑起来就知道是怎么回事。

第一步：跑一个/loop

`/loop` 是2026年3月的2.1.71版加进Claude Code的，作用就一个：按时间间隔，把同一件事重跑一遍。它有三种形态，你按需要挑。

三种形态：

`/loop 5m check the deploy`：固定5分钟一次

`/loop check the deploy`：Claude自己决定节奏（1分钟到1小时之间），每轮结束会告诉你下一轮隔多久、为什么

`/loop`：裸跑，执行项目里 `.claude/loop.md` 的内容；没有的话找 `~/.claude/loop.md`；都没有就跑内置的维护任务

时间单位是s/m/h/d，最小1分钟。它是会话级的，跑在你本机，会话关了它就停。

我10月在一个练手的小仓库里跑了一遍。仓库里有一个 `TODO.md`，写着一条待办「给clamp补一个下界测试」。我输入：

```
/loop 1m 读 TODO.md: 如果有没打勾的待办，挑第一条做掉、跑一遍测试、在 TODO.md 里打勾；如果全部打勾了，只回复一句「没有待办」
```

它先当场跑了一轮，把那条待办做掉、打了勾，然后每分钟醒一次。待办清空之后，每一轮就只回一句「没有待办」，三四秒结束：

```
Claude Code 2.1.287 · /loop 1m

Ran 1 shell command

● 循环已启动，每 1 分钟触发一次（监控任务 b4fv0hm1a，30 分钟后到期，到期会通知，需要的话再续）。启动时已经先跑了一轮：

- TODO.md 里唯一的待办「给 clamp 补一个下界测试」已完成。我在 tests/test_clamp 里加了 clamp(-5, 0, 10) == 0，并在
  TODO.md 里打了勾。
- 测试 3/3 通过，输出 OK。

第一次定时触发会在约 1 分钟后到来。待办已经全部打勾，所以之后每个 tick 我只会回复「没有待办」，直到你往 TODO.md
里加新条目。

* Sautéed for 16s · done 8:42 · 1 monitor still running

● Monitor event: AGENT_LOOP_TICK_todo {"prompt": "读 TODO.md: 如果有没打勾的待办，挑第一条做掉、跑一遍测试、在 TODO.md ...

Read 1 file

● 没有待办。

* Churned for 4s · done 8:43 · 1 monitor still running

● Monitor event: AGENT_LOOP_TICK_todo {"prompt": "读 TODO.md: 如果有没打勾的待办，挑第一条做掉、跑一遍测试、在 TODO.md ...

Searched for 1 pattern

● 没有待办。

* Sautéed for 3s · done 8:44 · 1 monitor still running

> 往 TODO.md 加一条新待办

▶ auto mode on · 1 monitor · ← for agents · ↓ to manage
```

Claude Code 2.1.287。每一轮开头那行Monitor event: AGENT_LOOP_TICK_todo就是闹钟，后面跟着它重读TODO.md、回复「没有待办」。

然后我在另一个终端里往 `TODO.md` 追加了一条「给is_even补一个负数测试」，什么也没跟它说。下一分钟，它自己读到了这条，补了两条断言，跑测试，打勾：

```
Claude Code 2.1.287 · /loop 1m

11     self.assertFalse(is_even(7))
12 +   self.assertTrue(is_even(-4))
13 +   self.assertFalse(is_even(-3))
14
15     def test_clamp(self):
16         self.assertEqual(clamp(15, 0, 10), 10)

● Update(TODO.md)
└─ Added 1 line, removed 1 line
   1 # 待办
   2
   3 - [x] 给 clamp 补一个下界测试
   4 -- [ ] 给 is_even 补一个负数测试
   4 +- [x] 给 is_even 补一个负数测试

Ran 1 shell command

● TODO.md 里出现了新待办「给 is_even 补一个负数测试」，已做完。

- 我在 test_is_even 里加了 is_even(-4) 为真、is_even(-3) 为假两条断言，并在 TODO.md 里打了勾。
- 测试 3/3 通过，输出 OK。

循环继续运行，TODO.md 里的待办全部打勾后，后续 tick 我只回复「没有待办」。

* Cogitated for 10s · done 8:45 · 1 monitor still running

> 提交这次改动

▶ auto mode on · 1 monitor · ← for agents · ↓ to manage
```

我没在对话里说一个字，新待办是从文件里进来的。

我这一次看到的实现是Monitor起的后台脚本，30分钟到期，跟文档写的7天cron不一样（§ 06有截图），到期了想续就再起一次。停掉它，跟它说「把这个循环停掉」就行；固定间隔的cron任务也可以问「what scheduled tasks do I have?」看列表再取消。空转的那些轮次，每轮只花几秒钟。

第二步：让它读CI和issue，先做triage

光重跑一句话不算loop，得让它读点东西、做点判断。最朴素的起点是一个triage（分诊）任务。

你给它一个prompt，让它每天早上去看三样东西：昨天的CI有没有失败、有哪些新开的issue、最近的commit改了什么。看完，挑出值得处理的，列个清单。

到这一步，你已经有了loop最核心的两个零件：一个定时触发的发现机制，加一个会替你过一遍信息的分诊脑子。Addy把定时触发这一块叫automation（§ 04零件一）。

第三步：加一个状态文件，让它有记忆

分诊跑完，结果存哪？别让它留在对话窗口里，关掉就没了。开一个markdown文件，把每次的发现、处理到哪一步都写进去。

这个文件就是loop的记忆，它得落在磁盘上，不能只活在上下文里，提不提交进git都行（§ 03）。有了它，今天没处理完的事，明天这个loop醒来还能接着干。

不想用裸markdown的话，换成Linear看板、issue列表也行，道理一样：状态要活在单次对话之外。

第四步：加一个evaluator

Claude Code里现成的工具是 `/goal` (2.1.139起)：它跑到某个条件满足为止，判断条件成立的，是另一个没参与干活的模型。写条件记三条（实跑的命令和截图见 § 05）：

- 写成对话里能被证明的样子。测试全部通过可以，代码质量好不行。
- 把你不想让它碰的东西写进去。我写了不改测试文件，它就主动跑了git status证明自己没碰测试。
- 带一个停止条款。结尾加上 `or stop after 6 turns`，跑不出来也会停。

Codex也有 `/goal`，命令名一样，还多了edit、pause、resume。

第五步：加worktree，让它并行

前面四步搭完，你已经有一个能自己跑的单线loop了。下一步，是让它能同时干好几件事。

办法是git worktree，同一个仓库开出几份互相独立的工作目录。Claude Code里有三个入口，别混了：

入口	给谁开	开在哪
<code>claude --worktree 名字</code> （或 <code>-w</code> ）	你自己新开的这个会话	<code>.claude/worktrees/名字/</code> ，分支 <code>worktree-名字</code>
<code>claude --bg "任务"</code> ，或在 <code>claude agents</code> 里派任务	后台会话	改文件之前自动进自己的worktree
子agent定义里写 <code>isolation: worktree</code>	子agent	每个子agent一个临时worktree，没改动就自动删

`claude agents` 是一屏看所有后台会话的视图，哪个在跑、哪个等你、哪个做完了，目前还是研究预览。记得把 `.claude/worktrees/` 加进 `.gitignore`。

隔离开，它们才能并行而不打架（§ 04零件二）。

第六步：离开电脑也跑

到这儿你的loop还绑在本机上。要关机也跑，现在有三条路：

路	要什么	注意
Cloud Routines (/schedule 或 claude.ai/code/routines)	claude.ai订阅登录、GitHub仓库	研究预览；最小间隔1小时；每次是干净的clone；除了定时，还能用API调用或GitHub事件触发
Desktop定时任务	桌面版开着、电脑开着	能读本地文件；可以勾选每次在独立worktree里跑
GitHub Actions的schedule	仓库和Actions额度	跟Claude Code无关的老办法，最稳

Routines的官方文档里列了几个现成的用例：夜里整理backlog、告警进来自动开修复PR、按GitHub事件做代码审查、部署后校验、查文档和代码有没有对不上。不知道第一个云端loop做什么，就从这里挑一个。Routines会在你的账号里建云端对象，这块我没建，表里的信息来自文档。

工具速查

上面用的命令都是Claude Code的。Codex能做同样的事，入口和叫法有的一样、有的不一样。这张表按官方文档和我本机的版本（Claude Code 2.1.287、Codex CLI 0.163.0）整理：

能力	Claude Code	Codex
按时间重跑	<code>/loop</code> （本机、会话内）	ChatGPT桌面app里的Scheduled tasks，自定义周期用RRULE
跑到条件满足	<code>/goal</code>	<code>/goal</code> （0.128.0起），带edit、pause、resume
并行隔离	<code>--worktree</code> 、后台会话自动worktree、子agent的 <code>isolation:worktree</code>	桌面app内置worktree，任务可选本地项目或新worktree
子agent	<code>.claude/agents/</code> ；Agent Teams是实验功能，默认关，要设 <code>CLAUDE_CODE_EXPERIMENTAL_AGENT_TEAMS=1</code>	<code>.codex/agents/</code> 里的TOML定义
外部连接	MCP、Plugins	MCP、Plugins
调用skill	SKILL.md， <code>/skill-name</code>	SKILL.md， <code>\$skill-name</code>
关机也跑	Cloud Routines（研究预览）	ChatGPT网页和手机端的定时任务、事件触发任务、Codex Cloud

Cursor也有对应的东西：Cloud agents、Automations，9月10日还出了Projects（测试版），一个只做规划、不写代码的协调agent，把活派给云端agent去跑。这块我只看了它的changelog，没实测。

第一个loop检查清单

动手之前，对着这七条过一遍。缺哪条，你的loop就在那条上有风险。

要素	问自己
发现源	它定时去读什么？（CI、issue、commit、收件箱）
状态文件	跨轮的记忆落在哪个磁盘文件上？
evaluator	有没有一个独立的、不是干活那个agent的检查？
隔离	并行的agent是不是各自一个worktree？
停止条款	跑不出来时，它几轮、多久之后会自己停？（ <code>/goal</code> 里的 <code>or stop after N turns</code> 、循环的到期时间）
token上限	设没设花费的天花板？跑飞了谁拦得住？会不会用 <code>/usage</code> 看账？
人工复核点	哪一步停下来等你看一眼，而不是一路自动到底？

Loop Engineering

别再问我什么是循环工程



微信搜一搜

🔍 花叔

花叔 · AI Native Coder

我一行代码都不会写，却用AI做出了AppStore付费榜Top 1的小猫补光灯，写了多本技术书。

所有产品全部AI写的，我只负责想清楚要做什么。我始终相信：AI生成内容不稀缺，稀缺的是判断力。

B站：花叔v · **公众号：花叔** · **X：花叔** · **YouTube** · **GitHub** · **官网**

Created by 花叔 · 2026年6月初版 · 10月更新
所有橙皮书免费下载：huasheng.ai/orange-books