

一本书是怎么被写出来的

lab 写书服务端调度代码手册

王建硕

费曼式写法 · 由多个 AI 代理撰写与互相审校

导读

lab 写书服务端调度代码手册

先别急着打开那 1770 行的 `server.ts`。先看一个数字：从你在 App 里点下「写书」，到一本书出现在书架上，中间要经过 **八步、三条引擎腿、两次落盘登记、一次扣费和至多一次退款**——而负责调度这一切的代码，主体都集中在 `server.ts` 的写书段落和 `book-legs.ts`、`inflight.ts` 两个小模块里。

这本书就是这些代码（连同它的测试和发版脚本）的手册。读者设定只有一种：要维护这套系统的工程师——包括三个月后已经忘了为什么这样写的作者自己。

这套系统解决什么问题

「写一本书」对 lab 来说是一个坏脾气的任务：跑几十分钟到几小时、烧的是别人家的订阅配额、中途可能被发版重启杀死、而用户付的是真金白银的算力。普通的「起个后台任务跑完回调」在这里处处踩坑：

- 引擎配额会半夜打穿——所以有三条腿的降级链（第四、五章）；
- 进程会死在终点线上——所以有在飞登记和重启守卫（第六章）；
- 钱收了就必须有交代——所以有早登记、幂等退款和 `_unmatched` 兜底（第七、八章）。

怎么读

顺序读：第一章给全景，之后每章拆一段，事故簿（第十章）收尾。每章固定四段——**这段代码在哪、它做什么、为什么这样写、怎么排查**——急用的时候可以直接翻每章最后一节，照命令表查。

所有代码引用都逐字对照过源码；所有函数名、常量名、日志前缀都可以在仓库里 `grep` 到。书里讲的每个机制背后都有一次真的事故，日期和书名都在源码注释里写着——如果哪天你改动了某段代码，记得回头看看它挡过的那次事故。

大白话

大白话：这不是一本「架构设计书」，是一本「出事了翻哪页」的手册。系统设计里没有多少深思熟虑，大多是事故驱动的补丁——但每个补丁都钉了测试、留了注释、写清了为什么。维护它的正确姿势是尊重这些补丁的来历。

第一章 · 全景：一单的完整生命周期

这本书讲的是一个约 1770 行的 `server.ts`，外加两个小模块 `book-legs.ts` 和 `inflight.ts`。它们合起来是 lab 的「[写书服务端调度](#)」：App 里点一下「写书」，到一本书出现在书架上，中间发生的每一步都在这几个文件里。第一章先把整条链走完一遍，后面九章逐段拆开。

这段代码在哪

- `src/server.ts`：主体。HTTP 路由在文件末尾的 `createServer` 回调里，写书相关的是 `POST /api/book → handleBook()` 这条线。
- `src/book-legs.ts`：换腿判据（第五章）。
- `src/inflight.ts`：在飞登记（第六章）。

它做什么：一单的生命周期

用户在 App 里填一段「种子」（一个词、一句话或一篇文章），点提交。服务端收到后并不等书写完才回话——`POST /api/book` 是 [fire-and-forget](#)（点火即忘：验完资格立刻返回 202，活儿在本进程后台慢慢跑完）。源码注释里写得很直白：「与 `/api/chat` 的『断线即中止』相反——app 提交完种子就可以关掉」。

完整链条是八步：

步骤	函数	日志行（ <code>journalctl</code> 里 <code>grep</code> 得到）
1. 收单 验资格 160 算力)	<code>handleBook() → chargeBook()</code> （扣	无日志，失败直接回 402/401
2. 开工 登记	<code>runBookJob() → writeInflight()</code>	登记失败只留 <code>[inflight] write failed</code>
3. 装提 示词开 跑	<code>startBookJob()</code>	<code>[book] start scope=... job=... seed=...</code>
4. 三腿 链跑引 擎	<code>runBookEngine() → runLeg()</code>	<code>[book] leg 1/3 = kimi- compat(\${BOOK_CLAUDE_MODEL})</code>
5. 引擎 边写边 发 done	子进程里的 agent 照 skill 跑 <code>build.mjs</code>	（在子进程日志，不在本服务）
6. 中途 早登记	30 秒轮询 <code>findBookByJobId()</code>	<code>[book] early-registered slug=... scope=...</code>
7. 引擎 返回销 档	<code>removeInflight()</code>	<code>[book] done scope=... thread=...</code>
8. 收尾 登记	<code>patchThreadEntry() / registerBookPost() / notifyBookDone()</code>	<code>[book] registered → [book] community- post → [book] push</code>

第 4 步的「三腿链」是这套系统的核心：腿（leg）指一个能跑写书任务的引擎凭据——Kimi 兼容端点、本机 Codex 订阅、lab 自己的 Claude 订阅，三条腿按序降级，一条倒下换一条。装配提示词、跑引擎、销档的骨架代码长这样：

```
const rec: InflightCreate = {
  kind: "create", jobId: randomUUID(), seed, scope, author, auth, startedAt: Date.now(), attempts: 1,
};
// 先登记再开跑。登记失败只留日志：登记是为了崩溃恢复，不能反过来挡住正常开工。
writeInflight(INFLIGHT_DIR, rec).catch((e) => console.error("[inflight] write failed", e));
startBookJob(rec);
```

第 5 步发生在子进程里，不在本文件：被拉起的写书 agent 读 skill、逐章写稿、逐章调
build.mjs done 把章节发到 R2。所以即使服务端进程在第 6 步之后被杀，已发布的章节也
永远在线上——这是「边写边发」换来的抗断点能力。

为什么这样写

- **202 立刻返回**：一本书要写几十分钟到几小时，没有任何 HTTP 连接值得等这么久。App 拿 202 关页面，书写完靠 APNs 推送叫用户回来。
- **扣费即准入**：没有排队、没有每日限额。注释原文：「扣成功才开写。数量限制全部取消——算力就是闸门」。简化了一整个准入子系统。
- **登记先于开跑**：writeInflight 在 startBookJob 之前。万一进程在引擎跑完前死掉，重启后能靠盘上的登记把这一单续上（第六章展开）。但登记失败不挡开工——崩溃恢复是保险，不能反过来成为单点故障。
- **收尾全是「尽力而为」**：登记、书帖、推送、退款每一个都 catch 住只留日志。书写完了是既成事实，收尾的任何一个 HTTP 调用失败都不该让整单看起来失败。

怎么排查

一条订单从头跟到尾，只需要一条命令：

```
journalctl -u claudes-agent | grep '\[book\]'
```

健康的一单，日志顺序是 start → leg → early-registered → done → registered →
community-post → push。缺哪一段，问题就在哪两个环节之间：

- 有 start 没有 leg：提示词装配后引擎没起来，看有没有 [book] job failed 异常。
- 有 leg 没有 early-registered：引擎跑了但 30 秒轮询一直找不到 book.json——agent 没按约定建工作目录，或 jobId 没写进 book.json。
- 有 done 带 ERROR=：引擎认输。后面应紧跟 [book] refund ref=... 和 [admin-push] 写书任务失败。
- 有 done 没有 registered：找不到 slug，看 bookmeta/ 目录下有没有 _unmatched-
<jobId>.json（第八章）。

第二章 · 入口与门禁：/api/book

这一章拆开链条的第一步：HTTP 入口 `handleBook()` 和它唯一的门禁 `chargeBook()`。核心设计一句话：**没有独立的权限系统，扣费就是准入。**

这段代码在哪

- `src/server.ts` 的 `handleBook()`：路由 `POST /api/book` 的处理函数。
- `src/server.ts` 的 `chargeBook()`：把用户 `bearer` 转发给 `agent worker` 的 `BOOK_CHARGE_URL`（默认 `https://jianshuo.dev/agent/usage/book-charge`）。
- `src/server.ts` 的 `fetchAuthorName()`：署名解析。

它做什么

`handleBook()` 的分支逻辑，按执行顺序：

条件	结果	说明
seed 为空	400 empty seed	seed 先 <code>trim().slice(0, 20000)</code> ，防巨型 body
bearer 不是 Bearer ... 开头	401 bad token	在 <code>chargeBook()</code> 里判
charge 返回非 200 或 <code>!body.ok</code>	原样透传状态码（如 402）	402 的 body 带 <code>need_suanli / suanli</code> ，App 据此弹充值
<code>payload.dry === true</code> 且 charge 通过	200 返回 charge 结果	只验余额不扣费，开跑前的预检
全部通过	202 {ok, charged_suanli, suanli}	同步起 <code>runBookJob()</code> ，不等待

扣费调用本身是一次转发——lab 不自己算钱，钱的事全在 `agent worker`：

```
async function chargeBook(
  auth: string | undefined,
  extra: { seed?: string; slug?: string; kind?: "revise" },
  dry: boolean,
): Promise<{ status: number; body: any }> {
  if (!auth?.startsWith("Bearer ")) return { status: 401, body: { error: "bad token" } };
  try {
    const r = await fetch(BOOK_CHARGE_URL, {
      method: "POST",
      headers: { Authorization: auth, "Content-Type": "application/json" },
      body: JSON.stringify({ ...extra, seed: extra.seed?.slice(0, 200), dry }),
    });
    const body = await r.json().catch(() => ({}));
    return { status: r.status, body };
  } catch {
    return { status: 502, body: { error: "charge unreachable" } };
  }
}
```

注意三个细节：

- **seed 只转前 200 字**：扣费服务只需要知道这是什么单，不需要全文——全文已经落到 lab 本地，没必要在网络上传两遍。
- **worker 不可达返回 502**，而不是放行。宁可暂时拒客，不免费开写。
- **定价不在本文件**：一口价 160 算力写在 agent worker 的 `usage.js` 的 `BOOK_SUANLI` 里，注释原话是「一口价扣 160 算力（真源在 `agent/src/usage.js` 的 `BOOK_SUANLI`）」。改价格不用动 lab。

署名在扣费成功后才解析：App 显式传的 `author` 优先；没传就用同一枚 `bearer` 去拉提交者的 `CLAUDE.json` 取 `profile.name`：

```
const author =
  String(payload?.author ?? "").trim().slice(0, 20) ||
  (await fetchAuthorName(req.headers.authorization));
runBookJob(seed, String(charge.body.scope ?? ""), author, req.headers.authorization);
```

`fetchAuthorName()` 拉不到就返回空串——空的意思是「这本书不署名」。注释里有一条硬规矩：「绝不回落到任何默认人名」。

为什么这样写

- **认证外包给 worker**：lab 不存用户表、不内置任何密钥（注释：「app 里零内置密钥」）。用户身份的唯一证明就是他手里那枚 VoiceDrop bearer（`anon_*` 或 `session JWT`），lab 拿它去问 worker「这是谁、余额够不够」。bearer 既认人又扣钱，一次调用办完两件事。
- **Caddy 对 `/api/book` 豁免 `basic_auth`**：lab 的网页聊天靠 Caddy 一层全局密码挡住外人，但 App 的写书请求走自己的 bearer，不该再吃那道密码。所以 Caddyfile 单独豁免这个路径，认证责任整体下移给 `chargeBook()`。
- **没有并发闸门**：写书路径没有自己的并发上限——`MAX_CONCURRENT_RUNS`（默认 2）只拦 `/api/chat`。写书的闸门就是算力：每单先扣 160，账户没钱自然进不来。这是 2026-08-10 用「扣费即准入」替换掉「须有成文文章 + 每日限额」旧门槛后的结果。

怎么排查

症状	先看
App 报「算力不足」	正常流程——402 的 body 里有 <code>need_suanli</code> ，让用户充值即可
点了写书没反应、无 [book] start 日志	手动复现： <code>curl -X POST localhost:8787/api/book -H "Authorization: Bearer <token>" -d '{"seed": "测试", "dry": true}'</code> ，看返回哪一档
全部 401	bearer 过期，让用户在 App 里重新登录
全部 502 <code>charge unreachable</code>	lab 到 <code>jianshuo.dev</code> 的出站网络断了，查 VPS 出口
<code>dry</code> 通、真扣失败	worker 端 <code>dry</code> 与真扣路径分叉，去查 agent worker 日志

第三章 · 提示词装配

写书 agent 不是人，它读到的全部世界就是一段提示词。这一章讲 `startBookJob()` 怎么把这段提示词拼出来——四块料：通用前言 `CODEX_BOOK_PREAMBLE`、署名、登记字段 (`jobId/owner`)、以及续跑时的补充提示。

这段代码在哪

- `src/server.ts` 的常量 `CODEX_BOOK_PREAMBLE`。
- `src/server.ts` 的 `startBookJob()` 前半段：拼 `prompt`。
- `src/server.ts` 的常量 `RESUME_HINT` 与函数 `resumeAfterRestartHint()`：两段名字像、用途不同的续跑提示。

它做什么

前言是写死的全文，一字未改过（2026-08-20 定稿至今）：

```
const CODEX_BOOK_PREAMBLE =
  `你是跑在服务器上的自动写书代理（可跑 bash、读写文件、联网）。\n` +
  `先完整阅读 ${SKILLS_DIR}/wjs-voicedrop-writing-book/SKILL.md 并严格照做；` +
  `按它第 0 步选定书的类型后，把对应写作 skill 的 SKILL.md（同在 ${SKILLS_DIR}/ 下）也完整读进来再动笔。\n` +
  `你没有并行子代理，也没有 Workflow——skill 里说 spawn 写手/评审 subagent 的地方，一律由你自己分步串行扮演：` +
  `写完一章，抛开写作时的思路，按该类型的评审维度独立重读打分并把意见落盘 reviews/NN.json；` +
  `不过就照 must_fix 重写（最多 3 轮），过审立刻 build.mjs done 发布，绝不攒到最后。\n` +
  `其余约定（工作目录、book.json、边写边发、断点续跑、封面用 /opt/claude-agent/bin/paint）一律照 skill 执行。`;
```

上面的注释解释了为什么需要这段：「codex 是单代理循环，没有 Claude 的 skill 装载/并行 subagent/Workflow——skill 里这三样都要在 prompt 里翻译成『自己读文件 + 自己分步扮演角色』」。skill 文件是给人和 Claude 写的，前言负责把它「转译」成单代理能执行的版本。同一份前言也喂给 claude 腿——三条腿引擎无关，提示词完全一致。

前言之后，`startBookJob()` 拼上这单的专属字段：

```
const byline = author
  ? `作者署名「${author}」——book.json 的 author 字段用这个名字，封面/页脚照此署名。`
  : `提交者没有留名字——book.json 不写 author 字段，全书不署名（不要默认署任何人名）。`;

const prompt =
  `${CODEX_BOOK_PREAMBLE}\n\n` +
  `任务：写一本书。${byline}\n` +
  `本次写书任务号 jobId=「${jobId}」——建 book.json 时把它原样写进顶层 "jobId" 字段（登记簿要靠它对号，别漏）。\n` +
  `(scope ? `这本书的产权归属 owner=「${scope}」——建 book.json 时把它原样写进顶层 "owner" 字段（谁能在线修改这本书` +
  `种子：\n${seed}` +
  `(resumed ? resumeAfterRestartHint(jobId) : "");`;
```

逐字段看：

注入内容	形式	服务端怎么用它
author	署名语句，有名字/没名字两种措辞	只进 book.json 的 author ；没名字时明确叮嘱「不要默认署任何人名」
jobId	「原样写进顶层 "jobId" 字段」	收尾时 findBookByJobId() 靠它反查 slug；退款 ref=jobId 靠它幂等

注入内容	形式	服务端怎么用它
owner	「原样写进顶层 "owner" 字段」	修书的主人校验以此为准；30 秒轮询还有兜底补写（第七章）
seed	全文	不进 prompt 之外的任何地方（扣费只带前 200 字）

最后是续跑提示词——系统里有两段，常被混淆：

	RESUME_HINT	resumeAfterRestartHint()
触发时机	换腿：runBookEngine 里第 2 条腿起追加	重启：inflight 孤儿续跑时追加（attempts > 1）
原因	上一条腿配额/凭据倒了，书可能写了一半	进程被杀，书可能「已经很完整」
措辞强度	「先查有没有半成品」	更具体：用 build.mjs status 对账，明确「不要重画已有的插图、不要重写已过审的章节」
带不带 jobId	不带	带（写书单），帮 agent 在 workspace 里认出自己的目录
	<pre>function resumeAfterRestartHint(jobId?: string): string { return (` \n\n补充（本单是服务器重启后的续跑）：上一次尝试被服务重启打断，很可能已经写了一部分甚至接近完成—— +     `动笔前先列一下 \${WORKSPACE}/ 下本单的工作目录` + (jobId ? `（book.json 里 jobId=「\${jobId}」的那个）` :     `，用 build.mjs status 看每章是 done / 待发(有稿) / 待写：有稿的直接发布，缺的补写，缺封面就补封面。` +     `**不要另起一本新书、不要换 slug、不要重画已有的插图、不要重写已过审的章节。**`); }</pre>	

为什么这样写

- **提示词即接口**：jobId/owner 不靠服务端事后修数据库，而是「嘱咐 agent 写进 book.json」。因为书的真源就是文件系统上的 book.json——服务端、build.mjs、修书模式全都读它，一处是真源处处是真源。
- **两段续跑提示词分开写**：9/8《嘟嘟和山上的灯塔》被发版 restart 杀掉时 14 页 14 图全在、只差发布，如果续跑提示词只说「接着写」，引擎很可能把 14 张图重画一遍。所以重启续跑的措辞专门加了一整排「不要」。换腿续跑没这个强度——换腿时书通常还残得多。
- **没名字要明说**：「不要默认署任何人名」是防 LLM 的本能——不写这句，它会自己编一个署名。

怎么排查

- 想看成形的提示词全文：写书单在 codex 腿时去 `$CODEX_HOME/sessions/YYYY/MM/DD/rollout-*.jsonl` 找第一条 `user_message`；lab 侧栏的会话列表（`summarizeCodex()`）也能直接点开看。
- agent 没写 jobId/owner 导致对不上号：先看第七章的 30 秒轮询有没有打 `[book] injected defaults` ——owner 有兜底，jobId 没有（jobId 漏写只能落 `_unmatched`，第八章）。
- 改提示词后想验证：起一本测试书，`journalctl -u claude-agent -f | grep '\[book\] start'` 确认开跑，再翻 rollout 第一条消息。

第四章 · 三条腿

「腿」是能跑一整本书的一套引擎凭据。当前三条：**kimi**（注入 Kimi 的 Anthropic 兼容端点）、**codex**（本机的 Codex CLI，走 ChatGPT 订阅）、**claude**（lab 自己的 Claude 订阅 OAuth）。`runBookEngine()` 把它们串成降级链：第一条倒了换第二条，全倒才算失败。

这段代码在哪

- `src/server.ts`：`runBookEngine()`（链）、`runLeg()`（分发）、`runClaudeExec()`（kimi/claude 两条腿共用的实现）、`runCodexExec()`（codex 腿）。
- `src/book-legs.ts`：`parseLegs()` 解析 `BOOK_LEGS`，`availableLegs()` 按凭据过滤。判据 `shouldTryNextLeg()` 是第五章的主角。

它做什么

链的主体只有一个循环：

```
const legs = availableLegs(BOOK_LEGS, {
  kimi: Boolean(BOOK_ANTHROPIC_BASE_URL),
  claude: Boolean(process.env.CLAUDE_CODE_OAUTH_TOKEN),
});
if (!legs.length) return runCodexExec(prompt, onThread); // 全没配：保底还是老路
let last: CodexOutcome = { ok: false, threadId: "", reply: "", error: "no leg ran" };
for (let i = 0; i < legs.length; i++) {
  const leg = legs[i];
  console.log(`[book] leg ${i + 1}/${legs.length} = ${legLabel(leg)}`);
  last = await runLeg(leg, i === 0 ? prompt : prompt + RESUME_HINT, onThread);
  if (last.ok) {
    if (i > 0) console.log(`[book] leg ${legLabel(leg)} 写成（前 ${i} 条腿用不了）`);
    return last;
  }
  if (!shouldTryNextLeg(last.error)) {
    console.log(`[book] leg ${legLabel(leg)} 失败，但不是配额/凭据问题，不换腿 → ${last.error.slice(0, 120)}`);
    return last;
  }
  console.log(
    `[book] leg ${legLabel(leg)} 用不了（配额满或凭据失效）` +
    `(i + 1 < legs.length ? ` → 换 ${legLabel(legs[i + 1])}` : "（已是最后一条腿）")
  );
}
return last;
```

三条腿的启动方式与环境变量：

腿	实现	关键 env	凭据来源
		注入	
kimi	<code>runClaudeExec(prompt, onThread, true)</code>	<code>ANTHROPIC_BASE_URL / ANTHROPIC_API_KEY</code> （来自 <code>BOOK_ANTHROPIC_*</code> ），删掉 <code>CLAUDE_CODE_OAUTH_TOKEN</code>	Kimi 兼容端点 key
claude	<code>runClaudeExec(prompt, onThread, false)</code>	不注入，吃 lab 自己的 <code>CLAUDE_CODE_OAUTH_TOKEN</code> ，模型退回 <code>MODEL</code>	lab 的 Claude 订阅

腿	实现	关键 env	凭据来源
codex	<code>runCodexExec()</code>	CODEX_HOME （独立凭据链）； BOOK_CODEX_MODEL 非空才加 <code>-m</code>	<code>\$HOME/.codex</code> 的 ChatGPT 登录

klm 腿的注入有一个容易漏的细节：第三方端点只认自家模型名，所以要把 Claude Code 内部所有档位都映射过去：

```
if (useCompat) {
  env.ANTHROPIC_BASE_URL = BOOK_ANTHROPIC_BASE_URL;
  env.ANTHROPIC_API_KEY = BOOK_ANTHROPIC_API_KEY;
  delete env.CLAUDE_CODE_OAUTH_TOKEN; // 订阅 token 在会场抢道，明确让位给兼容端点
  // 第三方端点只认自家模型名：把 Claude Code 内部各档位（含子任务的 haiku 档）
  // 全部映射到同一个模型，否则内部小任务会拿 claude-* 模型名打 Kimi 端点报错。
  env.ANTHROPIC_MODEL = BOOK_CLAUDE_MODEL;
  env.ANTHROPIC_DEFAULT_FABLE_MODEL = BOOK_CLAUDE_MODEL;
  env.ANTHROPIC_DEFAULT_OPUS_MODEL = BOOK_CLAUDE_MODEL;
  env.ANTHROPIC_DEFAULT_SONNET_MODEL = BOOK_CLAUDE_MODEL;
  env.ANTHROPIC_DEFAULT_HAIKU_MODEL = BOOK_CLAUDE_MODEL;
  env.CLAUDE_CODE_SUBAGENT_MODEL = BOOK_CLAUDE_MODEL;
  env.CLAUDE_CODE_EFFORT_LEVEL = process.env.BOOK_CLAUDE_EFFORT ?? "high";
}
```

codex 腿是 spawn 一个子进程、逐行解析 `--json` 事件流：

```
const args = ["exec", "--json", "-s", "danger-full-access", "-C", WORKSPACE, "--skip-git-repo-check"];
if (BOOK_CODEX_MODEL) args.push("-m", BOOK_CODEX_MODEL);
args.push(prompt);
const child = spawn(CODEX_BIN, args, {
  cwd: WORKSPACE,
  env: { ...process.env, CODEX_HOME },
  stdio: ["ignore", "pipe", "pipe"],
});
```

事件解析只认三类：thread.started（或 session.created）拿线程号；

item.completed 且 item.type === "agent_message" 的最后一条文本是答复；

turn.failed / error 记错误。进程退出时 ok = code === 0 && !!reply ——退出码 0 但没有答复，也算失败。

claude 腿还有一道文字识别：CLI 正常退出但第一轮就是 API 错误时，result 的 subtype 仍是 success，所以要按答复文本再判一次：

```
if (ok && /Failed to authenticate|API Error: \d{3}/i.test(reply.slice(0, 300))) {
  ok = false;
  error = reply.slice(0, 200);
}
```

为什么这样写

- **换腿是整本重跑**：三条腿的会话互不互通，换腿不能续上一条腿的上下文，只能靠工作目录里的半成品 + RESUME_HINT （第三章）。注释承认代价：「前一腿烧掉的轮数收不回来，这是换腿的固有代价」。
- **两条腿共用一个 runClaudeExec**：klm 和 claude 的差别只是「注不注入兼容端点」一个布尔（injectCompat）。订阅端点不认第三方模型名，所以 claude 腿必须把模型退回 MODEL ——注释：「订阅端点不认 k3 这类第三方名」。

- **codex 的三个坑都有注释**: flags 必须紧跟子命令后 (无 resume)、`--skip-git-repo-check` (workspace 不是 git repo)、`stdin: "ignore"` (否则等 EOF 挂住)。缺任何一个, 腿会以莫名其妙的方式死掉。
- **BOOK_RUN_DIR 独立目录**: claude 腿的 cwd 不是 WORKSPACE, 是独立的 bookrun/。注释记了原因: WORKSPACE 项目积累的 auto-memory (含旧书内容) 会自动注入请求, 触发 Kimi 风控 400 high risk (2026-08-24 二分定位实锤)。每次开跑还会先清空 bookrun 项目的 memory 目录。
- **错误不被覆盖**: catch 里 `if (!error)` 才赋值——SDK 常在 result 之后补抛一个笼统的「process exited with code 1」, 会把已经识别出的 403 配额错误盖掉, 换腿判别随之瞎掉。

怎么排查

- 看腿怎么换的: `journalctl -u claude-agent | grep '\[book\] leg'`。一行一个决定: 开跑、写成、不换腿、换哪条。
- claude 腿的逐轮成本: `grep 'claude-engine done'` 看 `turns=` 与 `cost=`。
- 手动探 codex 腿是否能跑: `cd /opt/claude-agent/workspace && codex exec --skip-git-repo-check "说一句你好"` ——复现的正是 `runCodexExec` 的最小形态。
- 超时兜底是 `BOOK_TIMEOUT_MS` (默认 3 小时), 超时被 SIGKILL 的错误文本是「书没写完就超时了 (180 分钟兜底)」。

第五章 · 换腿判据

换不换腿，整个系统只看一个函数的返回值：`shouldTryNextLeg(error)`。输入是上一条腿的错误文本，输出是一个布尔——**true = 这条腿废了，换下一条**；**false = 书本身写坏了，认输**。全部判据是三个正则数组，钉在 `book-legs.ts` 里。

这段代码在哪

- `src/book-legs.ts`：`QUOTA_PATTERNS`、`MODEL_REJECTED_PATTERNS`、`NOT_QUOTA_PATTERNS` 三个数组 + `shouldTryNextLeg()`。
- `test/book-legs.test.mjs`：所有模式对应的单测，错误串全部抄自事故当天 `journalctl` 的真实日志。

抽成独立文件、不 `import server.ts` 的原因写在文件头：「判别与顺序都抽在这里，是为了能脱离 `server.ts`（`import` 即起服务）单测」。

它做什么

判定顺序是先否定、后肯定——先看「明确不是配额」，再看模型被拒，最后看配额/凭据：

```
export function shouldTryNextLeg(error: string | undefined | null): boolean {
  const s = String(error ?? "");
  if (!s) return false;
  if (NOT_QUOTA_PATTERNS.some((re) => re.test(s))) return false;
  if (MODEL_REJECTED_PATTERNS.some((re) => re.test(s))) return true;
  return QUOTA_PATTERNS.some((re) => re.test(s));
}
```

三类「该换腿」的模式：

类别	代表模式	对应的真实错误
① 配额满	<code>/usage limit/i</code> 、 <code>/quota/i</code> 、 <code>/rate[_-]?limit/i</code> 、 <code>/\b429\b/</code> 、 <code>/api error: 403/i</code> 、 <code>/overloaded/i</code>	Kimi 「You've reached your 5-hour usage limit」；Kimi 兼容端点把配额耗尽报成 403
② 凭据失效	<code>/not logged in/i</code> 、 <code>/\b401\b/</code> 、 <code>/invalid.{0,12}api[_-]?key/i</code> 、 <code>/credit balance is too low/i</code> 、 <code>/refresh[_-]?token/i</code> 、 <code>/token.{0,24}(revoked invalidated)/i</code> 、 <code>/session has ended/i</code>	codex 「Not logged in」；refresh token 被吊销
③ 模型被账号拒	<code>/model.{0,40}does not exist/i</code> 、 <code>/do not have access to it/i</code> 、 <code>/model.{0,40}is not supported/i</code> 、 <code>/model[_-]?not[_-]?found/i</code>	404 「The model `gpt-5.5` does not exist...」；400 「'gpt-5.6' model is not supported...」

「不该换腿」的否定模式只有两个，但位置最高：

```
const NOT_QUOTA_PATTERNS: RegExp[] = [
  /error_max_turns/i,
  /high risk/i,
];
```

`error_max_turns` 是书写太长撞轮数上限, `high risk` 是 Kimi 内容风控——两者换腿重跑必然重蹈覆辙。注释:「这些词出现时, 即便文本里混进了上面某个词也不换腿」。单测里就钉了一个混合串: `"error_max_turns (quota fine)"` 必须判 `false`。

为什么这样写

- **模式必须锚在 `model` 一词附近**: ③ 的四条模式都带着 `model.{0,40}` 这类窗口。不锚定的话, 任何 `ENOENT` (「config file does not exist」) 或「功能不支持」(「compression is not supported」) 都会被当成换腿理由, 白烧一份额度。单测专门有一组「『不存在/不支持』只认模型那一种, 别误伤文件与功能」。
- **刻意不写裸 `/token/`**: 上下文超长的报错是「250000 tokens > 200000 maximum」「`max_tokens exceeded`」「`token limit reached`」——全含 `token` 二字, 但它们是书写坏了, 不是凭据问题。所以凭据模式写的是 `refresh[_]?token` 和 `token.{0,24}` (`revoked|invalidated`) 这种带动作的。单测有一组就叫「上下文超长不是凭据问题, 别被 `token` 二字带偏」。
- **宁可宽松**: 文件注释说清了误判的不对称——「漏判=退回『一条腿倒下全线瘫痪』的老样子, 误判=多花一条腿重跑一次。前者伤用户, 后者只费自己的额度」。所以模式宁宽勿窄。
- **每条模式都有生日**: `refresh token` 那两条是 2026-09-02 傍晚《易经在说什么》裁出来的(事故原文见第十章); 模型被拒那四条是 2026-09-07 五单退款裁出来的。模式不是设计出来的, 是事故喂出来的——所以单测的错误串坚持抄日志原文, 不自己编。

怎么排查

- 怀疑「该换没换」: 从日志抄出失败那行的 `ERROR=` 文本, 在仓库里先 `npm run build`, 然后一行验证: `node --input-type=module -e "import {shouldTryNextLeg} from './dist/book-legs.js'; console.log(shouldTryNextLeg('把错误串贴这'))"` (`-e` 默认按 CJS 解析, 不加 `--input-type=module` 顶层 `import` 会报语法错)。
- 改模式前先跑测试: `npm test` (先 `tsc build` 再 `node --test`)。新增一条模式, 必须同时在 `test/book-legs.test.mjs` 里加「该换」和「不该换」两组例子。
- 日志里 `[book] leg ...` 失败, 但不是配额/凭据问题, 不换腿 → 后面跟的就是被判 `false` 的错误原文, 直接拿去对模式。

第六章 · 在飞登记

书是在本进程后台跑的，`systemctl restart` 一下子进程全死。在飞登记 (inflight) 是应对这件事的机制：每单开工时在 `inflight/` 目录落一个 JSON，引擎跑完立刻删。于是「启动时目录里还有文件 = 上次没跑完的孤儿」，服务一起来就把它续上。

这段代码在哪

- `src/inflight.ts`：纯文件操作模块
(`writeInflight / removeInflight / listInflight / canRetry / inflightId`)，不 `import server.ts`，可单测。
- `src/server.ts`：`resumeInflightJobs()` (启动续跑)、`giveUpInflight()` (放弃一单)、`INFLIGHT_DIR` (默认 `/opt/claude-agent/inflight`)。
- `deploy.sh`：发版时的重启守卫。
- `test/inflight.test.mjs`：钉「崩溃恢复靠的那几条性质」。

它做什么

登记里存的是「重跑这一单的全部再现参数」：

字段	写书单 (InflightCreate)	修书单 (InflightRevise)
标识	<code>jobId</code>	<code>slug + entryTs</code>
内容	<code>seed</code>	<code>instruction</code>
共有	<code>scope</code> 、 <code>author</code> 、 <code>auth</code> (下单 <code>bearer</code>)、 <code>startedAt</code> 、 <code>attempts</code>	

落盘的写法是两步：先写临时文件再 `rename`，权限 `0600`：

```
export async function writeInflight(dir: string, rec: Inflight): Promise<void> {
  await mkdir(dir, { recursive: true, mode: 0o700 });
  const p = inflightPath(dir, rec);
  const tmp = p + ".tmp";
  await writeFile(tmp, JSON.stringify(rec, null, 2) + "\n", { mode: 0o600 });
  await rename(tmp, p);
}
```

生命周期对应的日志：

时刻	动作	日志
开工	<code>writeInflight</code> (失败只留日志，不挡开工)	(静默)
引擎返回	<code>removeInflight</code> —— 成败都销档	(静默，后续是 <code>[book] done</code>)
启动后 5 秒	<code>resumeInflightJobs()</code> 扫目录	<code>[inflight]</code> 发现 <code>N</code> 个上次没跑完的任务 / <code>[inflight]</code> 续跑 ... (第 <code>N</code> 次)
超过次数封顶	<code>giveUpInflight()</code> ：标 <code>failed</code> + 退款 + 告警 + 销档	<code>[inflight]</code> ... 已拉起 <code>N</code> 次仍未完成，放弃

次数封顶只有两行：

```
/** 原跑 + 续跑各算一次；第 3 次不再试——两次都没跑完，多半不是「被打断」而是书本身有问题。 */
export const INFLIGHT_MAX_ATTEMPTS = 2;
```

`resumeInflightJobs()` 的关键顺序是「先记次数再开跑」—— `rec.attempts += 1` 先落盘再 `startBookJob(rec)`。这一跑再被杀，下次启动读到的 `attempts` 已经是加过的，不会无限重跑。

`deploy.sh` 那一半守卫是 `bash`：

```
busy="$(inflight_list)"
if [ -n "$busy" ] && [ "${FORCE_RESTART:-0}" != "1" ]; then
  if [ "${WAIT_FOR_IDLE:-0}" = "1" ]; then
    echo "► 有书在跑，等空窗（最多 ${MAX_WAIT_MIN} 分钟）："; echo "$busy" | sed 's/^/ /'
    ...
  else
    echo "✕ 有书在跑，不重启（代码已同步，正在跑的进程仍是旧版）："
    echo "$busy" | sed 's/^/ /'
    exit 2
  fi
fi
```

注意设计：**代码同步永远做，restart 才看 inflight**。`rsync` 不影响运行中的进程；非空时退出码 2 并列出在跑的书，给两条出路—— `WAIT_FOR_IDLE=1` 每 30 秒看一次、最多等 `MAX_WAIT_MIN`（默认 90）分钟；`FORCE_RESTART=1` 立刻杀（新代码会把被杀的书续跑，但前一条腿烧掉的轮数收不回）。

为什么这样写

- **tmp + rename**：rename 在 POSIX 里是原子的，读者要么看到旧文件要么看到新文件，永远读不到半截 JSON。
- **bearer 落盘是被迫的**：退款、推送、书帖登记都要用下单那枚用户 bearer 认人，不存就续不完整。文件头注释交代了这笔账：「文件 0600、放在 `ReadWritePaths=/opt/claude-agent` 沙箱内、与 `.env` 里的 OAuth token 和 `.codex/auth.json` 同一信任边界，跑完即删——可接受」。systemd 单元的 `ProtectSystem=strict + ReadWritePaths=/opt/claude-agent` 保证 `inflight/` 恰好在仅有的可写区内。
- **档档在收尾之前**：引擎一返回就删登记，再去做登记簿/推送这些慢活。注释：「登记留着反而会在收尾中途重启时把一本已退款/已发布的书再跑一遍」。宁可漏续跑（损失一次收尾），不可重复跑（重复扣费的观感）。
- **封顶 2 次的理由**：注释原话——「两次都没跑完，多半不是『被打断』而是书本身有问题」。无限重跑会把一本写不成的书反复烧钱。
- **坏文件不拖累别人**：`listInflight()` 对坏 JSON、错形状、`.tmp` 半截文件一律跳过——一个手写坏的文件不该拖住别的孤儿。单测逐条钉死。

怎么排查

要确认什么	命令
现在有没有书在跑	<code>ls /opt/claude-agent/inflight/</code> （空 = 空闲，可安全 restart）
这单是谁、第几次	<code>cat /opt/claude-agent/inflight/*.json</code> （0600，需 <code>claude-agent</code> 身份或 <code>root</code> ）
启动续跑了没有	<code>journalctl -u claude-agent grep '\[inflight\]'</code>
发版被守卫拦了	看 <code>deploy.sh</code> 输出的书名清单；确认后可 <code>WAIT_FOR_IDLE=1 ./deploy.sh</code>

第七章 · 30 秒轮询与早登记

写书 job 起跑时还不知道 slug——slug 是建筑师（写书 agent）中途起的。这一单要在中途就有了主人、能在 App 里看到「写书中」，服务端就得自己去把 slug 找出来。办法是一个 30 秒的轮询：setInterval 每 30 秒拿 jobId 去工作目录里反查 book.json。

这段代码在哪

- src/server.ts 的 findBookByJobId() / findSlugByJobId()：jobId → book.json 的反查。
- src/server.ts 的 startBookJob() 中段：reg = setInterval(..., 30000) 那个轮询闭包。

它做什么

反查扫三个目录——WORKSPACE、BOOK_RUN_DIR (bookrun)、/tmp (旧约定)：

```
async function findBookByJobId(jobId: string): Promise<{ dir: string; book: any } | null> {
  for (const root of [WORKSPACE, BOOK_RUN_DIR, "/tmp"]) {
    let dirs: string[];
    try {
      dirs = await readdir(root);
    } catch {
      continue;
    }
    for (const d of dirs) {
      try {
        const j = JSON.parse(await readFile(join(root, d, "book.json"), "utf8"));
        if (j?.jobId === jobId && typeof j?.slug === "string" && SLUG_RE.test(j.slug))
          return { dir: join(root, d), book: j };
      } catch {
        /* not a book dir */
      }
    }
  }
  return null;
}
```

每个目录尝试读 book.json，读到不是书目录（无此文件、JSON 坏、没 jobId）就跳过；命中要求三个条件同时成立：jobId 相等、slug 是字符串、且过 SLUG_RE (/^[a-z0-9][a-z0-9-]{0,62}\$/ ——防提示词里被注入奇怪路径)。

轮询命中后做四件事：

1. **clearInterval**——只登记一次。
2. **slug 回填 inflight 登记**：续跑/放弃时省一次反查，也让 deploy.sh 的守卫清单能报出书名。
3. **确定性兜底注入**：type === "childrens" 且没写 hidden → 补 "hidden": true（绘本缺省不上架）；没写 owner → 补下单人 scope。改完写回 book.json，日志 [book] injected defaults slug=... hidden=... owner=...。
4. **早登记 bookmeta**：往对话线 push 一条 {kind:"create", status:"running"} 条目，日志 [book] early-registered slug=... scope=...。

```

const reg = setInterval(async () => {
  try {
    const hit = await findBookByJobId(jobId);
    if (!hit) return;
    clearInterval(reg);
    const slug: string = hit.book.slug;
    earlySlug = slug;
    if (rec.slug !== slug) { rec.slug = slug; writeInflight(INFLIGHT_DIR, rec).catch(() => {}); }
    ...
    const meta = (await readBookMeta(slug)) ?? { slug, scope, author, createdAt: startedAt, thread: [
      if (!meta.thread.some((e) => e.ts === startedAt)) {
        meta.thread.push({ ts: startedAt, kind: "create", instruction: seed.slice(0, 4000), ..., status:
          await writeBookMeta(meta);
          console.log(`[book] early-registered slug=${slug} scope=${scope}`);
        }
      } catch {
        /* 下一轮再试 */
      }
    ], 30000);
    reg.unref?.();
  }
}

```

为什么这样写

- **为什么必须中途登记**：注释记着一次实锤——「只在收尾登记的话，进程中途死掉这本书就没有主人（2026-08-20 部署重启掐死在跑任务，实锤丢过一次）」。扣了费的书中途死掉，没有 early-registered 就查不到它属于谁。slug 一出现（建筑师落 book.json，约 1 分钟内）就登记，把「没主人的窗口」从整个写作周期压到一分钟。
- **为什么有了 skill 约定还要服务端兜底注入**：skill 和提示词（第三章）都叮嘱 agent 写 owner、绘本写 hidden，但 LLM 是概率机器，「叮嘱了」不等于「一定写」。服务端这一层是确定性的：检查了一定有。而且 build.mjs 每次发布都从盘上重读 book.json，注入的值不会被后续写作冲掉。
- **为什么用 startedAt 对号**：thread 条目以 ts === startedAt 去重。续跑沿用原单的 startedAt（第六章），所以重跑不会登记出第二条 create 条目。
- **扫 /tmp 是历史包袱**：注释明说「旧约定；PrivateTmp 下与本进程同命名空间，job 刚结束时还在」。现在的工作目录一律在 WORKSPACE，但老代码路径留着不碍事。

怎么排查

- App 里长时间看不到「写书中」：grep 有没有 [book] early-registered。没有就是轮询一直没命中——看书的目录建在哪（ls /opt/claude-agent/workspace/ | grep book-，再 grep jobId .../book-*/book.json）。
- owner 没写进 book.json：应有 [book] injected defaults 日志；没有的话查 SLUG_RE 是否拦了 slug。
- 命中过但 App 仍看不到进度：去查 bookmeta 的上传（第八章），readBookMeta 读的是 R2 公开 URL，边缘缓存 5 分钟。

第八章 · 收尾与钱

引擎返回之后，`startBookJob()` 里那个 `async` 闭包进入收尾：登记对话线、登记书帖、推送、失败则退款告警。这一章同时讲清钱的回路（预扣与退款）和登记簿的存储（为什么真源在 R2 而不是本地）。

这段代码在哪

- `src/server.ts`：`readBookMeta()` / `writeBookMeta()` / `patchThreadEntry()`（对话线存储）；`registerBookPost()` / `notifyBookDone()` / `notifyAdmin()`（三个外向调用）；`refundBook()`（退款）；`startBookJob()` 收尾段（`_unmatched` 落档）。

它做什么

对话线的存储模型。`bookmeta` 是一本书的「对话线」——创建与历次修改的流水账（`{slug, scope, author, createdAt, thread:[...]}`）。真源是 R2 的 `books/<slug>/_src/bookmeta.json`，本地 `bookmeta/` 目录只剩两个用途：老条目的读回退（读到即懒迁移上 R2）和 `_unmatched` 落档。写路径：

```
function writeBookMeta(meta: BookMeta): Promise<void> {
  const p = metaWriteChain.then(async () => {
    const body = JSON.stringify(meta, null, 2) + "\n";
    try {
      const tok = await publisherToken();
      if (!tok) throw new Error("no publisher token");
      const r = await fetch(`${FILES_API}/upload/books/${meta.slug}/_src/bookmeta.json`, {
        method: "PUT",
        headers: { Authorization: `Bearer ${tok}`, "Content-Type": "application/json" },
        body,
      });
      if (!r.ok) throw new Error(`upload ${r.status}`);
      await unlink(metaPath(meta.slug)).catch(() => {}); // R2 落定后清掉本地旧件
    } catch (e) {
      console.error(`[bookmeta] R2 upload failed slug=${meta.slug}:`, ...);
      await mkdir(BOOKMETA_DIR, { recursive: true });
      await writeFile(metaPath(meta.slug), body);
    }
  });
  metaWriteChain = p.catch(() => {});
  return p as Promise<void>;
}
```

三个要点：`metaWriteChain` 把写串行化（防 create 收尾与 revise 并发时「读-改-写」互相覆盖）；上传失败回落本地文件（宁可暂留 VPS 也绝不丢条目）；R2 落定后删本地旧件（防回退读到陈旧线）。读路径反过来：先 fetch R2 公开 URL（带 `?_=${Date.now()}` 穿透书页路由的 5 分钟边缘缓存——App 轮询 history 要看到实时进度），失败才读本地。

收尾的三个外向调用，全部尽力而为、失败只留日志：

调用	认人方式	干什么
<code>registerBookPost(auth, slug)</code>	用户 bearer	把书 upsert 成社区一等帖 (<code>share_id "book-</code>

调用	认人方式	干什么
		<slug>"), 赞/回应/推荐与普通帖同权
notifyBookDone(auth, slug, title)	用户 bearer	给主人发 APNs 「书写好了」; 只有 ok 才推
notifyAdmin(title, body)	发布账号 token (worker 校验 scope==ADMIN_SCOPE)	失败/放弃立刻推管理员

顺序有讲究：先 registerBookPost 再 notifyBookDone —— 「推送里的书立刻可在社区看到」。失败不推用户：「钱的事人工处理，别用推送吓人」。

退款。预扣一口价的书没写成，用下单时那枚 bearer 调 worker 的 book-refund：

```
async function refundBook(  
  auth: string | undefined,  
  extra: { ref: string; kind?: "revise" },  
) : Promise<void> { ...  
  console.log(`[book] refund ref=${extra.ref} kind=${extra.kind ?? "book"} status=${r.status} ${bod
```

幂等靠 ref：写书 ref=jobId、修书 ref=slug#ts，worker 端同 ref 只退一次（返回 deduped）。startBookJob() 里有两处退款调用——正常失败分支（if (!ok)）和 catch 分支——但它们互斥：try 走完就不进 catch，注释原话「双保险不会双退」。第二处的存在理由是引擎抛异常（超时/崩溃）时 ok 分支根本走不到。

找不到 slug 的兜底：引擎返回了（甚至 ok）但 findSlugByJobId 还是找不到——全书没一个地方承认这个 jobId。这时落 bookmeta/_unmatched-<jobId>.json，内含 jobId/scope/author/instruction 全文，留人工对号。注释：「绝不让一本已扣费的书没有主人记录」。

为什么这样写

- **真源放 R2 是因为 VPS 会重装：**注释记着 2026-08-23 的迁移——本地文件会随 VPS 重装丢失，R2 天然持久，而且公开 URL 读、files API 写，不需要新基础设施。
- **写走发布账号、读走公开 URL：**上传用 publisherToken()（与 build.mjs 同源的 ~/.config/voicedrop/credentials）——所有书由同一个发布账号发布，存储位置分不出下单人，这正是 bookmeta 必须存在的原因（下单人只在扣费那一刻可知）。
- **收尾全幂等：**patchThreadEntry 按 ts 对号、退款按 ref 幂等、书帖是 upsert。收尾中途进程死掉重来一遍，每一步都安全。

怎么排查

症状	命令 / 位置
书写完了但 直接探真源：	curl -s
App 没有对 "https://jianshuo.dev/voicedrop/books/<slug>/_src/bookmeta.json"；没有	
话线 再看本地	/opt/claude-agent/bookmeta/<slug>.json
怀疑上传失败	journalctl -u claude-agent grep '\[bookmeta\]' (R2 upload failed)
用户说没收到推送	grep '\[book\] push' 看 status; 4xx 多半是 bearer 过期
要对一笔退款	grep '\[book\] refund' —— refunded=160 是退了，deduped 是重复请求被幂等拦下

症状

命令 / 位置

done 了但
没有 `ls /opt/claude-agent/bookmeta/_unmatched-*`，拿 jobId 人工对号
registered

第九章 · 修书：/api/book/revise

书写完不是终点。主人在 App 里可以持续提修改指令，走 `POST /api/book/revise`。修书与写书共用引擎链、在飞登记、对话线，但入口逻辑独立：`handleBookRevise()`。它的设计重点是把「主人校验」放在「扣费」之前——**不是主人，一分钱不动**。

这段代码在哪

- `src/server.ts` 的 `handleBookRevise()`：路由处理。
- `src/server.ts` 的 `runReviseJob()`：修书 job 本体。

它做什么

入口的分支顺序（每一道都是故意的）：

1. **400 校验**：slug 过 `SLUG_RE`；非 dry 时 instruction 不能为空（截 4000 字）。
2. **dry 探路**：`chargeBook(auth, {slug, kind:"revise"}, true)`——验 token、验余额、拿 scope，**不扣费**。探路失败直接透传状态码。
3. **主人校验**：owner 真源 = 线上 `_src/book.json` 的 owner，退回对话线的 scope。owner 存在且 $\neq$ 请求者 $\rightarrow$ **403 not-owner**；owner 不存在 $\rightarrow$ 只有发布账号本人（`publisherScope()`）且书在线（`bookExistsOnline()`）才放行，否则 404。
4. **补建对话线**：meta 不存在就当场建一条（老书没有 bookmeta 的场景）。
5. **同书单飞**：`meta.thread.some(e => e.status === "running")` $\rightarrow$ **409 busy**。
6. **dry 就此返回** 200（App 预检走到这里为止）。
7. **真扣费**：同一 URL，`dry=false`，预扣 40 算力。
8. **落 thread 条目 + inflight 登记 + 开跑**，202 返回。

主人校验那段的原文：

```
const srcBook = await fetchSrcBook(slug);
let meta = await readBookMeta(slug);
const owner = String(srcBook?.owner ?? "") || String(meta?.scope ?? "");
if (owner) {
  if (owner !== requester) {
    res.writeHead(403, json).end(JSON.stringify({ error: "not-owner" }));
    return;
  }
} else {
  const pub = await publisherScope();
  if (!requester || !pub || requester !== pub || !(await bookExistsOnline(slug))) {
    res.writeHead(404, json).end(JSON.stringify({ error: "no-book" }));
    return;
  }
}
```

`runReviseJob()` 与写书的差别在提示词和收尾：

- **fresh session, 不 resume**：注释——「不 resume 写书旧线程（背着整段历史只多花钱）——每次修改都是全新 codex exec，以工作目录/线上成书这些『文件』为真源」。原书写书时的几十轮对话不肯进来；引擎自己去读文件。

- **署名冻结**: byline 叮嘱「这本书署名『X』, 改动不要动署名」。署名取自 `bookAuthor(srcBook, meta)` ——线上 `book.json` 优先 (修书可能改过署名), 登记簿只是创建期快照。
- **成功刷新书帖**: `registerBookPost(auth, slug)` ——标题/hidden/章节数可能变了。
- **失败退 40**: `refundBook(auth, {ref: `${slug}#${entryTs}`, kind: "revise"})` ——`ref` 用条目时间戳, 与 `inflight` 的修书 id (`slug#entryTs`) 同一套, 幂等对得上。

为什么这样写

- **403 先于扣费**: 先 `dry` 拿 `scope` 比对、再真扣——主人校验失败时用户余额一分不动。如果反过来先扣再验, 每个乱试别人 `slug` 的请求都得走一次退款。
- **409 单飞防并发改书**: 两个修改同时跑, 写回 `book.json` / 章节文件会互相覆盖。对话线里任何一条 `status:"running"` 就拒绝新单。
- **404 而不是 403 给外人**: `owner` 缺失的老书路径上, 非发布账号拿到的是 404 `no-book`——不透露「这本书存在但不是你的」。
- **dry 探路复用扣费通道**: 不另造一个「验余额」接口, `dry=true` 的 `charge` 就是它。一个接口两种模式, 少一处分叉。

怎么排查

症状	先看
主人修自己的书被 403	探真源: <code>curl -s "https://jianshuo.dev/voicedrop/books/<slug>/_src/book.json" grep owner</code> , 与 <code>whoami</code> 的 <code>scope</code> 对比; 边缘缓存 5 分钟, 代码里已带 <code>?_=</code> 穿透, 手动 <code>curl</code> 记得也带
409	
busy	看对话线里有没有卡在 <code>running</code> 的条目: <code>curl -s ".../_src/bookmeta.json"</code> ; 有 = 上次一直不修书收尾没 <code>patch</code> 成功 (查 <code>[revise] done</code> 与 <code>[bookmeta]</code> 日志)
解	
修书改了但没生效	<code>journalctl -u claude-agent grep '\[revise\]'</code> 看 <code>start/done</code> ; <code>done</code> 带 <code>ERROR</code> 则已自动退 40
怀疑重复扣费	<code>grep '\[book\] refund'</code> 看 <code>kind=revise</code> 的 <code>ref</code> 是否 <code>deduped</code>

第十章 · 事故簿与诊断手册

这套系统里没有哪个机制是「设计」出来的——每一条都对应一次真烧过钱或真丢过书的事故。这一章把事故按时间排开，每条注明它催生了哪段代码；后半张是诊断命令表，出事时照表查。

这段代码在哪

事故不在代码里，在代码的**注释**里。读 `server.ts` 的注释就是读事故簿——每条注释都带日期。

它做什么：事故时间线

日期	事故	催生的机制
2026-08-20	部署重启掐死在跑任务，一本已扣费的书中途死掉、没有主人记录	30 秒轮询早登记（第七章）；同日写书引擎换到 Codex CLI（ <code>runCodexExec</code> 诞生）
2026-08-23	一个 121MB 的 codex rollout 把「readFile 整读」打爆 node 堆，systemd 重启连带杀死正在跑的写书子进程（嘟嘟和小考拉那单）	所有 jsonl 读取改流式逐行 <code>jsonlLines()</code> ，单行超 2MB 丢弃
2026-08-23	VPS 重装风险显性化	bookmeta 真源迁 R2，本地只留回退（第八章）
2026-08-24	workspace 项目的 auto-memory 把旧书内容注入请求，Kimi 风控 400 high risk（二分定位实锤）	claude 腿 cwd 改独立 <code>BOOK_RUN_DIR</code> ，每单清空该项目 memory（第四章）
2026-08-31	《同一个月亮》绘本 turns=81 撞 80 轮上限：14 张图全画完，卡在没上传，线上只有字没有图	<code>BOOK_MAX_TURNS</code> 提到 160
2026-09-02 上午	单腿吃 Kimi，7 单 11 分钟内涌入打穿 5 小时配额，08:24-09:37 九本书全灭，每本跑几十轮才倒	三腿降级链 <code>BOOK_LEGS</code> + <code>shouldTryNextLeg()</code> （第四、五章）
2026-09-02 傍晚	《易经在说什么》：codex 腿报「refresh token was revoked」，判据一个都不匹配，第三条腿没试就整单失败	凭据失效类模式补 <code>/refresh[_-]? token/i</code> 等四条（第五章）
2026-09-07	42 单里 5 单：kimi 撞配额正常换到 codex，codex 没设 <code>BOOK_CODEX_MODEL</code> 吃了 CLI 默认的 gpt-5.5 被账号拒，判据只认配额/凭据 → 判成「书写坏了」→ 第三条腿一次没试就退款	<code>MODEL_REJECTED_PATTERNS</code> （400 与 404 两种措辞都认）+ 钉死 <code>BOOK_CODEX_MODEL</code>
2026-09-08	《嘟嘟和山上的灯塔》14 页 14 图全写完、差一步发布，被一次发版 <code>systemctl restart</code> 杀在终点线上；当天两次发版都没人看有没有任务在跑	在飞登记 <code>inflight.ts</code> + 启动续跑 + <code>deploy.sh</code> 重启守卫（第六章）
2026-09-10	莫兰迪《在瓶子之间》：codex 死在 <code>build.mjs</code> <code>done</code> 中途，目录停在 7/8	（无需新代码）验证了「每章即发」的价值——前 7 章永远在线，只损失最后一章

读法：每一条机制的注释里都写着「如果没有它会怎样」。改代码前先把对应事故读一遍——删掉一条判据、一个登记、一个守卫，就等于把那次事故重新邀请回来。

为什么这样写

- **注释带日期、带书名、带原文**：模式数组里每条正则后面跟着事故当天的原话（如 `/refresh[_-]?token/i` 后的注释引了 `codex` 的完整报错）。下次有人想「简化」这条正则时，先看它挡过什么。
- **单测用真实错误串**：`book-legs.test.mjs` 文件头写明「这些错误串全部抄自 2026-09-02 事故当天 `journalctl` 的真实日志」。自己编的串会和真实世界漂移。
- **事故的修复都偏向「不依赖人记得」**：灯塔事故的根因不是「没人看」，而是「需要人看」。修复是把「看」变成 `deploy.sh` 的退出码 2。

怎么排查：诊断命令表

场景	命令
跟一单的全生命周期	<code>journalctl -u claude-agent grep '\[book\]'</code>
看三腿链怎么换的	<code>journalctl -u claude-agent grep '\[book\] leg'</code>
有没有书在跑（发版前必看）	<code>ls /opt/claude-agent/inflight/</code>
启动续跑/放弃的记录	<code>journalctl -u claude-agent grep '\[inflight\]'</code>
修书链路	<code>journalctl -u claude-agent grep '\[revise\]'</code>
退款流水	<code>journalctl -u claude-agent grep '\[book\] refund'</code>
某本书还差哪几章	<code>node ~/.claude/skills/wjs-voicedrop-writing-book/build.mjs status book-<slug></code>
验证一条错误该不该换腿	<code>node --input-type=module -e "import {shouldTryNextLeg} from './dist/book-legs.js'; console.log(shouldTryNextLeg('错误串'))"</code> （先 <code>npm run build</code> ）
探 <code>codex</code> 腿死活	<code>cd /opt/claude-agent/workspace && codex exec --skip-git-repo-check "说一句你好"</code>
跑全部单测	<code>npm test</code> （先 <code>tsc build</code> ，再 <code>node --test</code> ）
服务起不来	<code>journalctl -u claude-agent -n 200 找 [uncaughtException] / [unhandledRejection]</code>

最后一行留给读者：这本书本身就是这条流水线写出来的——你读到的每一章，都经历过第二到第九章描述的每一个环节。它既是文档，也是证据。