匠人学院
JR ACADEMY

2026 实战版

让多个 AI Agent 组队，干你一个人干不完的活

CrewAI 多 Agent 实战手册

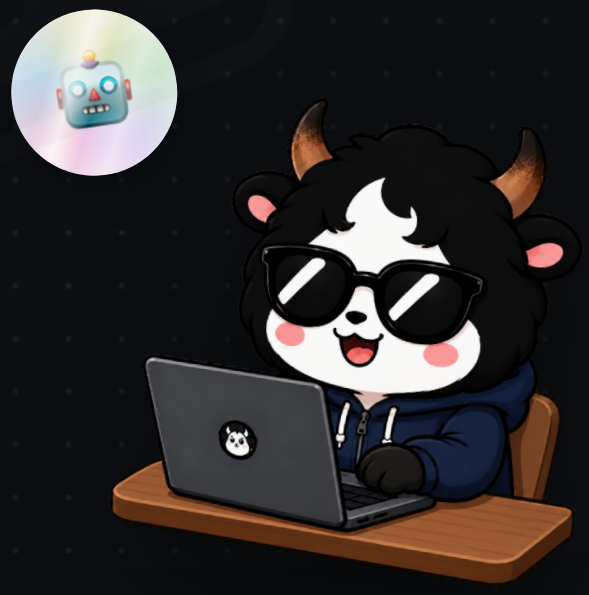
50k+ Star 开源多智能体框架。从安装到生产上线：Agent 角色设计、Task 链、Flow 编排、Memory、多模型混用、FastAPI 封装、Docker 容器化、断点续跑——8 章全有真实代码可复制。

Python 3.10+ 即装即用

多 Agent 自动协作

FastAPI + Docker 生产部署

一个人干不完的调研分析，让 Crew 来



jiangren.com.au · 在线版随官网更新 /wiki/crewai-guide

AI 来匠人 ✦ CrewAI 多 Agent 手册 ✦ JIANGREN.COM.AU ✦ 让多个 AI 组队干活 ✦

CONTENTS

目录（8 章）

📖 这本手册怎么读

这本书和官网 wiki《CrewAI 实战手册》同源（jiangren.com.au/wiki/crewai-guide，免费、不用注册，官网那份持续更新）。完全没碰过 CrewAI 的，第 1、2 章搞清楚概念、5 分钟跑通；做过 LLM 应用、想上多 Agent 的，第 3、4 章是核心——Agent 角色设计、Task 链、Flow 编排、Memory 全在这；已经有 Crew 想上生产的，第 6、7 章讲量化测试、FastAPI 封装、Docker 化、断点续跑，第 8 章讲怎么给 Agent 挂私有文档做企业内部知识问答。代码全可复制，配置直接用。

💬 装的时候卡住了？扫码进群问

群里每周分享真实的 CrewAI 项目模板和 Multi-Agent 实战案例。

CHAPTER 01

🤖 CrewAI 是什么：让多个 AI Agent 组队干活

CrewAI 一句话介绍

CrewAI 是一个开源 Python 框架，让你把多个 AI Agent 组成一支"团队"，每个 Agent 有自己的角色、目标和工具，协同完成复杂任务。它由 João Moura 在 2024 年初创建，截至 2026 年 4 月已有 **50k+ GitHub Stars**，是目前增长最快的多智能体框架。

CrewAI 架构示意

你可以把 CrewAI 理解为"AI 版的项目经理"：你定义好团队成员（Agent）、分配任务（Task）、选择协作模式（Crew），然后一键启动，Agent 们自己协调、自己执行、自己交付结果。

核心架构

CrewAI 的设计非常直观，三个核心概念就能上手：

- **Agent**：一个有角色（role）、目标（goal）、背景故事（backstory）的 AI 实体，可以绑定工具。
- **Task**：一件具体的事，指定交给哪个 Agent 做，期望输出什么。
- **Crew**：把 Agent 和 Task 组合在一起的容器，决定执行顺序（顺序执行 or 层级委派）。

在这之上，CrewAI 还有 **Flow**（编排多个 Crew 的工作流）和 **Memory**（跨任务记忆），后面章节展开。

和其他框架有什么不同

特性	CrewAI	LangGraph	AutoGen	Dify
设计思路	角色扮演团队	图状态机	对话式多 Agent	可视化拖拽
上手难度	低，3 个概念	高，要理解图	中等	最低，不用写代码
灵活度	中高	最高	中	低
Python 代码量	少，YAML 配置	多	中	几乎不写
适合场景	快速搭多 Agent 应用	复杂状态流转	群体讨论决策	非技术人员搭 AI 应用

实际选择很简单：

- 你是 Python 开发者，想快速搭一个多 Agent 系统 → **CrewAI**
- 你需要精确控制每一步状态流转和错误处理 → **LangGraph**
- 你想让多个 Agent 像开会一样讨论 → **AutoGen**
- 你不写代码，想拖拽搭应用 → **Dify / Coze**

谁适合用 CrewAI

- **Python 开发者**：CrewAI 是纯 Python 框架，写几十行代码就能跑起一个多 Agent 系统
- **自动化工程师**：内容生产、数据分析、竞品调研这类重复性工作，用 Crew 编排最省事
- **AI 应用开发者**：需要在产品里嵌入多 Agent 能力，CrewAI 的 API 设计比较干净
- **想学 AI Agent 原理的人**：角色 / 任务 / 协作的模型很容易理解，适合入门多智能体概念

不太适合的场景：完全不会 Python 的同学建议先看 Dify 或 Coze；只需要单次 LLM 调用的简单任务没必要上多 Agent。

CHAPTER 02

🚀 快速上手：安装 CrewAI 并跑通第一个 Crew

环境准备

CrewAI 要求 **Python 3.10 ~ 3.13**，包管理用的是 uv（Rust 写的超快 Python 包管理器，CrewAI CLI 内部依赖它）。

```
# 确认 Python 版本
python3 --version  # 需要 3.10+

# 安装 CrewAI（会自动安装 uv）
pip install crewai

# 带内置工具包一起装
pip install 'crewai[tools]'

# 验证
crewai version
```

安装完你会得到一个 `crewai` 命令行工具，它能帮你生成项目骨架、跑 Crew、训练 Agent。

创建第一个项目

CrewAI 提供脚手架命令，一行搞定项目结构：

```
crewai create crew my_research_team
cd my_research_team
```

生成的目录结构：

```
my_research_team/
├── pyproject.toml          # 项目依赖
├── .env                    # API Key 放这里
├── README.md
├── src/
│   └── my_research_team/
│       ├── __init__.py
│       ├── main.py         # 入口：kickoff()
│       ├── crew.py         # Crew 定义（@CrewBase 装饰器）
│       ├── config/
│       │   ├── agents.yaml # Agent 配置
│       │   └── tasks.yaml  # Task 配置
│       └── tools/
│           └── custom_tool.py
```

CrewAI 的一个设计亮点是 **YAML 配置 + Python 代码分离**：Agent 和 Task 的角色描述写在 YAML 里，编排逻辑写在 Python 里。运营同事改 YAML 调角色描述，开发同事改 Python 调逻辑，互不干扰。

配置 Agent 和 Task

打开 `config/agents.yaml`，定义你的 Agent 团队：

```yaml
researcher:
  role: "Senior Tech Researcher"
  goal: "找到关于 {topic} 最新、最准确的信息"
  backstory: >
    你是一个资深技术研究员，擅长从海量信息中提取关键洞察。
    你的研究结果会被团队其他成员引用，所以准确性至关重要。

writer:
  role: "Tech Blog Writer"
  goal: "把研究结果写成一篇通俗易懂的中文技术博客"
  backstory: >
    你是一个技术博客写手，文风直接不啰嗦。
    你的读者是有一定基础的开发者，不需要过度解释基础概念。
```

再打开 `config/tasks.yaml`：

```yaml
research_task:
  description: >
    对 {topic} 做一次全面调研，覆盖：最新进展、核心功能、
    和竞品的区别、社区评价。输出结构化的调研报告。
  expected_output: "一份包含 10 个要点的调研报告，每个要点附信息来源"
  agent: researcher

writing_task:
  description: >
    基于调研报告，写一篇 1500 字的中文技术博客。
    要求：有代码示例、有对比表格、语气轻松但专业。
  expected_output: "一篇可以直接发布的 Markdown 格式博客文章"
  agent: writer
```

配置 API Key

在 `.env` 文件里填入你的 LLM API Key：

```bash
# 默认用 OpenAI
OPENAI_API_KEY=sk-xxxxxxxx

# 也可以用 Anthropic Claude
ANTHROPIC_API_KEY=sk-ant-xxxxxxxx
```

CrewAI 默认使用 GPT-4o，但你可以在 Agent 配置里指定任何支持的模型（后面高级章节会讲多模型混用）。

启动你的第一个 Crew

```
# 安装依赖
crewai install

# 运行!
crewai run
```

终端会实时打印每个 Agent 的思考过程和输出：

```
[Researcher] 🔍 Starting research on "CrewAI framework"...
[Researcher] I'll search for the latest information...
[Researcher] ✅ Task completed. Found 10 key insights.

[Writer] ✍️ Writing blog post based on research...
[Writer] ✅ Blog post completed. 1,523 words.
```

第一次跑可能要等 1–2 分钟（取决于你的 LLM 响应速度）。输出会保存在终端里，你也可以在 `crew.py` 里配置输出到文件。

常见新手问题

Q: 必须用 OpenAI 吗? 不是。CrewAI 支持 OpenAI、Anthropic、Google Gemini、本地 Ollama 模型等。后面第四章会讲怎么切换。

Q: `crewai create crew` 和 `crewai create flow` 有什么区别? `crew` 创建一个 Agent 团队项目；`flow` 创建一个包含多个 Crew 的工作流项目。新手先从 `crew` 开始。

Q: YAML 里的 `{topic}` 是什么? 占位符。运行时通过 `crew.kickoff(inputs={"topic": "CrewAI"})` 传入实际值。

CHAPTER 03

🧩 核心功能：Agent、Task、Crew、Tool 深度拆解

Agent 的全部配置项

CrewAI 的 Agent 不只是"一个 prompt"——它有角色设定、工具绑定、委派能力和记忆。除了 YAML 配置，你也可以直接用 Python 定义：

```
from crewai import Agent
from crewai_tools import SerperDevTool, ScrapeWebsiteTool

researcher = Agent(
    role="Senior Tech Researcher",
    goal="找到最准确、最新的技术信息",
    backstory="你是一个有 10 年经验的技术分析师...",
    tools=[SerperDevTool(), ScrapeWebsiteTool()],
    llm="gpt-4o",           # 可以每个 Agent 用不同模型
    max_iter=5,              # 最多重试 5 次
    memory=True,             # 开启记忆
    allow_delegation=True,   # 允许把子任务委派给其他 Agent
    verbose=True             # 打印思考过程
)
```

几个关键参数说明：

- **`backstory`** 不是装饰——它直接影响 Agent 的行为。一个"谨慎的安全审计员"和一个"激进的增长黑客"面对同样的任务会给出完全不同的答案。
- **`allow_delegation`** 设为 True 后，Agent 发现自己搞不定的事会自动找团队里更合适的 Agent 帮忙。
- **`max_iter`** 防止 Agent 陷入死循环，5–10 是比较合理的值。

Task 的设计要点

Task 是你给 Agent 下达的具体指令。写好 Task 的关键是 **`expected_output` 要具体**：

```python
from crewai import Task

research_task = Task(
    description="调研 {topic} 的最新动态，包括版本更新、社区反馈、竞品对比",
    expected_output="结构化 Markdown 报告，包含 10 个要点，每点附来源链接",
    agent=researcher,
    output_file="research_report.md"  # 自动保存到文件
)

# Task 之间可以传递上下文
writing_task = Task(
    description="基于调研报告撰写技术博客",
    expected_output="1500 字中文技术博客，Markdown 格式",
    agent=writer,
    context=[research_task]  # 自动获取 research_task 的输出作为输入
)
```

`context` 参数是 CrewAI 的精华之一：你不需要手动把上一个 Task 的结果复制粘贴给下一个 Agent，框架自动传递。

Crew 的两种执行模式

顺序执行（Sequential）——Task 按定义顺序逐个执行：

```python
from crewai import Crew, Process

crew = Crew(
    agents=[researcher, writer],
    tasks=[research_task, writing_task],
    process=Process.sequential,  # 先研究，再写文章
    verbose=True
)

result = crew.kickoff(inputs={"topic": "CrewAI framework"})
print(result.raw)  # 最终输出
```

层级执行（Hierarchical）——自动分配一个 Manager Agent 来调度任务：

```python
crew = Crew(
    agents=[researcher, writer, editor],
    tasks=[research_task, writing_task, review_task],
    process=Process.hierarchical,  # Manager 自动决定谁先做什么
    manager_llm="gpt-4o"
)
```

层级模式适合任务之间有复杂依赖、需要动态调度的场景。

内置工具清单

`crewai[tools]` 自带一批开箱即用的工具：

工具	用途	需要的 API Key
`SerperDevTool`	Google 搜索	SERPER_API_KEY
`ScrapeWebsiteTool`	抓取网页内容	无
`FileReadTool`	读取本地文件	无
`DirectoryReadTool`	列出目录文件	无
`PDFSearchTool`	搜索 PDF 内容	无
`CodeInterpreterTool`	执行 Python 代码	无
`GithubSearchTool`	搜索 GitHub 仓库	GITHUB_TOKEN

自定义工具

写一个自定义工具只需要 `@tool` 装饰器：

```
from crewai.tools import tool

@tool("Calculate Token Cost")
def calculate_cost(model: str, input_tokens: int, output_tokens: int) -> str:
    """根据模型和 token 数量计算 API 调用费用"""
    prices = {
        "gpt-4o": (0.0025, 0.01),
        "claude-sonnet-4-6": (0.003, 0.015),
    }
    if model not in prices:
        return f"未知模型: {model}"
    input_price, output_price = prices[model]
    cost = (input_tokens / 1000) * input_price + (output_tokens / 1000) * output_price
    return f"预估费用: ${cost:.4f}"
```

把工具传给 Agent 就行，Agent 会根据 Task 需要自动决定是否调用：

```
cost_analyst = Agent(
    role="Cost Analyst",
    goal="分析和优化 AI API 调用成本",
    backstory="...",
    tools=[calculate_cost]
)
```

CrewAI 的工具系统兼容 LangChain Tools，如果你之前写过 LangChain 工具可以直接复用。

🤖 学多 Agent 系统是 AI Engineer 的核心技能。想知道怎么把 CrewAI 项目写进简历、通过 AI 岗位面试？Amelia 帮你做简历诊断：

匠人学院™
JR ACADEMY

Amelia

IT Career Consultant

在澳洲 IT 业务咨询领域拥有10年丰富经验和专业知识。对 IT 行业的发展趋势和技术需求有着深入的了解，擅长提供转码就业问题解决方案，能够准确把握最新的技术趋势和就业需求，为您提供最具前瞻性的建议。

探索IT领域，点亮你的科技未来

提供问题解决方案　工作内推机会

IT就业群提供　IT技术提升服务

300+
已帮助300+人成功就业

1000+
帮助1000+学员答疑解惑

200+
已帮助数百余人成功转码

CHAPTER 04

⚡ 进阶技巧：Flow 编排、Memory 记忆、多模型混用

Flow：编排多个 Crew 的工作流

当你的项目复杂到一个 Crew 搞不定时，**Flow** 登场。Flow 用事件驱动的方式把多个 Crew 串起来，支持条件分支、并行执行、状态共享。

```
from crewai.flow.flow import Flow, listen, start, router
from pydantic import BaseModel

class ContentState(BaseModel):
    topic: str = ""
    research: str = ""
    article: str = ""
    quality_score: float = 0.0

class ContentPipeline(Flow[ContentState]):
    @start()
    def pick_topic(self):
        self.state.topic = "CrewAI 2026 新特性"
        return self.state.topic

    @listen(pick_topic)
    def do_research(self, topic):
        # 这里可以 kickoff 一个 research_crew
        result = research_crew.kickoff(inputs={"topic": topic})
        self.state.research = result.raw
        return self.state.research

    @router(do_research)
    def check_quality(self, research):
        # 根据研究质量决定下一步
        if len(research) > 2000:
            return "write"    # 质量够，去写文章
        return "redo"         # 内容太少，重新研究

    @listen("write")
    def write_article(self):
        result = writing_crew.kickoff(inputs={
            "topic": self.state.topic,
            "research": self.state.research
        })
        self.state.article = result.raw

    @listen("redo")
    def redo_research(self):
        # 换一个更强的模型重新研究
        result = deep_research_crew.kickoff(inputs={"topic": self.state.topic})
        self.state.research = result.raw
```

`@start()` 标记入口，`@listen()` 监听上一步完成事件，`@router()` 做条件路由。这套模式比写一堆 if-else 干净得多。

Memory：让 Agent 跨任务记住东西

CrewAI 支持三种记忆：

记忆类型	作用	存储方式
Short-term	当前 Crew 运行期间的上下文	内存
Long-term	跨多次运行的经验积累	本地 SQLite
Entity	记住特定实体的信息（人名、项目名等）	本地 SQLite

开启方式：

```
crew = Crew(
    agents=[researcher, writer],
    tasks=[research_task, writing_task],
    memory=True,        # 开启 short-term + long-term
    embedder={          # 指定 embedding 模型（用于记忆检索）
        "provider": "openai",
        "config": {"model": "text-embedding-3-small"}
    }
)
```

Long-term Memory 的实际用途：你的 Crew 第一次跑完后，Agent 会记住哪些策略有效、哪些无效。下次再跑同类任务，它会优先采用之前成功的策略。这对反复运行的自动化任务（比如每日新闻整理）特别有用。

多模型混用

一个 Crew 里不同 Agent 可以用不同的 LLM——贵的模型做关键决策，便宜的做简单任务：

```python
researcher = Agent(
    role="Researcher",
    goal="...",
    backstory="...",
    llm="claude-sonnet-4-6"     # 研究用 Claude，推理能力强
)

writer = Agent(
    role="Writer",
    goal="...",
    backstory="...",
    llm="gpt-4o-mini"           # 写作用便宜模型就够了
)

reviewer = Agent(
    role="Reviewer",
    goal="...",
    backstory="...",
    llm="ollama/llama3"         # 审校用本地模型，零成本
)
```

本地模型用 Ollama 接入，不需要 API Key，适合对数据隐私有要求的场景。

实战案例：竞品调研 Crew

一个完整的竞品分析 Crew：

```python
from crewai import Agent, Task, Crew, Process
from crewai_tools import SerperDevTool, ScrapeWebsiteTool

search = SerperDevTool()
scrape = ScrapeWebsiteTool()

# Agent 团队
scout = Agent(
    role="Market Scout",
    goal="找到 {product} 的所有竞品及其最新动态",
    backstory="你专门追踪 SaaS 市场动态，嗅觉敏锐",
    tools=[search, scrape]
)

analyst = Agent(
    role="Competitive Analyst",
    goal="对比 {product} 和竞品的功能、定价、用户评价",
    backstory="你擅长结构化分析，报告总是有理有据",
    tools=[search]
)

strategist = Agent(
    role="Strategy Advisor",
    goal="基于竞品分析给出可执行的产品建议",
    backstory="你是产品策略顾问，建议必须落地可执行"
)

# 任务链
scan_task = Task(
    description="搜索 {product} 的主要竞品，每个竞品收集：官网、核心功能、定价、最近更新",
    expected_output="竞品清单表格，至少 5 个竞品",
    agent=scout
)

compare_task = Task(
    description="对比 {product} 和每个竞品的优劣势",
    expected_output="详细对比矩阵 + 每个维度的结论",
    agent=analyst,
    context=[scan_task]
)

advise_task = Task(
    description="给出 3 条具体的产品改进建议",
    expected_output="每条建议包含：要做什么、为什么、预期效果、优先级",
    agent=strategist,
    context=[compare_task],
    output_file="competitive_report.md"
)

# 组队启动
crew = Crew(
    agents=[scout, analyst, strategist],
    tasks=[scan_task, compare_task, advise_task],
    process=Process.sequential,
    verbose=True
)
```

```
result = crew.kickoff(inputs={"product": "CrewAI"})
```

这个 Crew 跑一次大约消耗 30k–50k tokens（视搜索结果量而定），用 GPT-4o 大概 $0.15–0.30，用 Claude Sonnet 差不多。

CHAPTER 05

❓ 常见问题：定价、踩坑、选型指南

定价：开源免费 + 企业版付费

CrewAI 的核心框架是 **MIT 开源**，本地跑完全免费。你只需要为 LLM API 调用付费（OpenAI、Anthropic 等的费用）。

CrewAI 同时提供托管平台 crewai.com：

方案	价格	包含
Free	$0/月	50 次 Crew 执行，基础监控
Pro	按量付费	更多执行次数，详细日志，团队协作
Enterprise	联系销售	私有部署，SSO，专属支持

```
# 本地跑 = 框架免费 + LLM API 费用
# 以 GPT-4o 为例，一个 3-Agent Crew 单次运行大约：
# - 输入 ~20k tokens × $0.0025/1k = $0.05
# - 输出 ~10k tokens × $0.01/1k  = $0.10
# - 合计约 $0.15/次
# 用 Ollama 本地模型则完全免费
```

常见踩坑和解决方法

1. Agent 陷入循环，反复输出同样的内容

这是最常见的问题。解决方法：

```
agent = Agent(
    role="...",
    goal="...",
    backstory="...",
    max_iter=5,              # 限制最大迭代次数
    max_retry_limit=2        # 工具调用失败最多重试 2 次
)
```

同时检查 `expected_output` 是否足够具体——模糊的目标容易让 Agent 反复尝试。

2. Token 费用超预期

多 Agent 协作时 token 消耗是单次调用的 3–10 倍，因为每个 Agent 都有自己的 system prompt + 上下文。控制方法：

- 简单任务用便宜模型（`gpt-4o-mini`、`claude-haiku-4-5-20251001`）
- 关键决策才用强模型
- 设置 `max_iter` 避免无限循环
- 用 `verbose=True` 观察 token 实际消耗

3. `ModuleNotFoundError: No module named 'crewai_tools'`

安装时漏了 tools 包：

```
pip install 'crewai[tools]'
```

4. Agent 之间互相推诿，任务没人做

通常是 `allow_delegation=True` 加上角色描述不清晰导致的。解决方法：给每个 Agent 明确的、不重叠的职责范围，或者在关键 Agent 上设 `allow_delegation=False` 。

5. Ollama 本地模型连不上

确保 Ollama 服务在跑，然后配置：

```
agent = Agent(
    role="...",
    goal="...",
    backstory="...",
    llm="ollama/llama3.1"  # 格式：ollama/模型名
)
```

选型决策树

不确定该不该用 CrewAI？按这个流程走：

```
你的任务需要多个 AI 角色协作吗？
├── 不需要，一个 LLM 调用就够 → 直接用 API，不用框架
└── 需要
    ├── 你会写 Python 吗？
    │   ├── 不会 → 用 Dify 或 Coze（可视化拖拽）
    │   └── 会
    │       ├── 需要精确控制状态流转？ → LangGraph
    │       └── 想快速搭建、角色分工清晰？ → CrewAI ✅
    └── 补充
        ├── 预算有限 → CrewAI + Ollama 本地模型（零 API 费）
        └── 企业级需求 → CrewAI Enterprise 或 LangGraph + LangSmith
```

CrewAI vs 直接写 Prompt Chain

有人会问："我用 Python 自己调 API，写几个函数串起来不也行吗？"

可以，但 CrewAI 帮你处理了这些你迟早要自己写的东西：

- **Agent 角色隔离**：每个 Agent 有独立的 system prompt 和上下文，不会互相污染
- **自动重试和容错**：工具调用失败自动重试，Agent 卡住自动切换策略
- **委派机制**：Agent 发现自己不擅长的任务可以自动转给更合适的队友
- **Memory 和学习**：跨运行记忆，越用越好
- **监控和调试**：verbose 日志、callback hooks、token 消耗追踪

学习资源

资源	链接	说明
官方文档	docs.crewai.com	最权威，更新最快
GitHub 仓库	github.com/crewAIInc/crewAI	源码 + examples 目录
YouTube 官方	搜索 "CrewAI tutorial"	João Moura 亲自讲解
社区论坛	community.crewai.com	提问和看别人的项目
CrewAI 认证	crewai.com	官方认证课程，10 万+ 开发者参加

上手建议：先跑通第二章的入门项目，然后改 Agent 角色和 Task 描述来做你自己的场景，比看再多教程都有用。

CHAPTER 06

🧪 测试与调试：用 crewai test 量化 Crew 表现

为什么多 Agent 比单模型更难测

调试单个 LLM 调用，你只需要看输入输出。调试一个 Crew，你面对的是：

- 哪个 Agent 输出了错误信息?
- Task 描述不清还是 Agent backstory 有问题?
- 是这次 LLM 返回异常，还是每次都差?

CrewAI 提供了一套完整的测试工具链来回答这些问题。

crewai test：一条命令得到量化评分

`crewai test` 是最快的质量基线工具。它会将你的 Crew 运行 N 次，然后用一个评审 LLM 给每个任务打分（1–10）。

```
# 跑 3 次，用 gpt-4o-mini 作为评审（便宜，够用）
crewai test -n 3 -m gpt-4o-mini
```

输出示例：

```
Task                | Run 1 | Run 2 | Run 3 | Avg
──────────────────────────────────────────────────
search_task         |   7.5 |   8.0 |   7.0 | 7.5
compare_task        |   6.0 |   7.5 |   6.5 | 6.7
advise_task         |   8.5 |   9.0 | 🔴 Err | 7.8*
──────────────────────────────────────────────────
Crew Overall Score  |  7.3  |  8.2  |  6.8  | 7.4
Total Execution Time |  142s | 128s  | 99s   |
```

重点看两个信号：

1. **方差大的任务**（Run 1 和 Run 3 差 2 分以上）→ 该任务描述不够稳健，Agent 理解不一致
2. **持续低分任务**（平均 < 6）→ 说明 expected_output 定义不清，或者 Agent backstory 不匹配

你也可以在代码里调用 `crew.test()`：

```
from crewai import Crew, Process

crew = Crew(
    agents=[scout, analyst, strategist],
    tasks=[scan_task, compare_task, advise_task],
    process=Process.sequential
)

# n_iterations=3, openai_model_name 用便宜的
crew.test(n_iterations=3, openai_model_name="gpt-4o-mini")
```

crewai train：用人工反馈训练 Agent

发现某个 Agent 输出总是方向跑偏？用 `crewai train` 收集人工评分，把这些评分作为示例喂给 Agent：

```
crewai train -n 5 -m gpt-4o-mini
```

运行期间，每次执行后系统会提示你对 Agent 输出评分（1–10）并提供文字反馈。训练结果保存在本地的 `trained_agents_data.pkl`，下次 `crew.kickoff()` 时自动加载。

```
# 启动时自动应用训练数据（默认行为）
crew.kickoff(inputs={"product": "CrewAI"})

# 禁用训练数据（想用原始表现对比时）
crew.kickoff(inputs={"product": "CrewAI"}, reset_memory_before_execution=True)
```

> **何时该 train？** 至少跑 `test` 发现某个任务平均分 < 7，才值得开始训练。直接 train 而不先测基线，无法判断有没有提升。

verbose 模式：按行追踪 Agent 思考链

生产前必过的关：开 `verbose=True` 看完整的 ReAct 循环。

```
crew = Crew(
    agents=[scout, analyst, strategist],
    tasks=[scan_task, compare_task, advise_task],
    verbose=True          # 打印每个 Agent 的思考 + 行动 + 观察
)
```

输出每个步骤长这样：

```
[Scout] Thought: I need to search for competitors of CrewAI...
[Scout] Action: Search("CrewAI competitors 2026")
[Scout] Observation: Found LangGraph, AutoGen, Dify...
[Scout] Final Answer: Here are the top 5 competitors...
```

看到 Agent 陷入循环（同一个 Action 出现 3 次以上），立即检查：

- 工具是否真的返回了有用内容？
- `expected_output` 是否让 Agent 知道什么时候该停？

crewai replay：回放失败的任务

长跑 Crew（10 个任务以上）跑到第 8 步挂了？不需要从头来：

```
# 列出上次运行的所有任务 ID
crewai log-tasks-outputs

# 从第 8 个任务（task_id 是 7，从 0 开始）重新跑
crewai replay -t 7
```

`replay` 会复用之前任务的输出作为上下文，只重跑你指定的任务及其后续。这在调试耗时 / 耗钱的 Crew 时省大量成本。

DeepEval：自动化评估 + CI 集成

`crewai test` 是手动触发，要进 CI 流水线就需要 DeepEval：

```
pip install deepeval crewai
```

```
from deepeval.integrations.crewai import DeepEvalCrewObserver
from deepeval.metrics import AnswerRelevancyMetric, FaithfulnessMetric

# 注册观察器——一行代码，不改任何 Crew 逻辑
DeepEvalCrewObserver()

# 定义评估指标
metrics = [
    AnswerRelevancyMetric(threshold=0.7),
    FaithfulnessMetric(threshold=0.8)
]

# 正常 kickoff，DeepEval 自动捕获所有 span
result = crew.kickoff(inputs={"product": "CrewAI"})

# 查看评估报告
# deepeval test run 或登录 confident-ai.com 看可视化
```

CI 里加这一步，每次 PR 合并前自动验证 Crew 表现没有退步。

调试速查表

症状	先检查	解决方向
Agent 输出与预期方向完全不同	`backstory` 和 `goal` 是否清晰	重写 Agent 定义，加具体约束
每次结果差异很大	`expected_output` 是否太模糊	在 expected_output 里加格式要求
工具调用失败	工具 API key / 网络是否正常	先单独测试工具，加 `max_retry_limit=3`
任务 N+1 没有用到任务 N 的结果	`context=[task_n]` 有没有加	显式声明任务依赖
Crew 跑了 30 分钟没结果	`max_iter` 是否太高	设 `max_iter=5`，用 replay 调试

🧪 **会跑 crewai test、能量化 Agent 质量，这就是生产级 AI Engineer 的水准。想把这些搭进真实项目再聊转行路线？**

匠人学院™
JR ACADEMY

Rain

Senior IT Career Consultant

在澳洲近20年，Master of HRM，15年企业HR及猎头招聘工作经验，熟悉中国及澳洲就业市场情况。尤其在IT领域中拥有超过6年的咨询及招聘经验。

是你拿下offer的最佳辅助！

毕业生求职做规划　辅助在职人士跳槽晋升

已帮助数千余人成功就业　免费简历诊断

3000+	1200+	1000+
3000+咨询案例	1200+辅助成功斩获offer	1000+简历诊断

http://linkedin.com/in/rainqiu

CHAPTER 07

🚢 生产上线：FastAPI 封装 + Docker 容器化 + 断点续跑

从脚本到服务

本地 `crew.kickoff()` 是同步阻塞的，跑完才返回。真实业务里你需要：

1. **HTTP 触发**：让后端系统、定时任务、前端按钮能调用 Crew
2. **异步执行**：Crew 跑 2 分钟，API 不能阻塞 2 分钟
3. **任务状态追踪**：用户需要知道"还在跑"还是"已完成"
4. **失败恢复**：网络抖动或 LLM 超时不应该让整个流程重来

这一章把这四件事都解决掉。

用 FastAPI 封装 Crew

项目结构：

```
my_crew/
├── crew.py           # Crew 定义
├── api.py            # FastAPI 应用
├── Dockerfile
└── docker-compose.yml
```

crew.py（复用第三章的竞品分析 Crew）：

```python
from crewai import Agent, Task, Crew, Process
from crewai_tools import SerperDevTool

def build_crew() -> Crew:
    search = SerperDevTool()

    scout = Agent(
        role="Market Scout",
        goal="找到 {product} 的所有竞品及最新动态",
        backstory="你专门追踪 SaaS 市场动态",
        tools=[search],
        max_iter=5
    )
    analyst = Agent(
        role="Competitive Analyst",
        goal="对比 {product} 和竞品的功能、定价",
        backstory="你擅长结构化分析",
        tools=[search],
        max_iter=5
    )

    scan_task = Task(
        description="搜索 {product} 主要竞品，每个收集官网、功能、定价",
        expected_output="竞品清单表格，至少 5 个竞品",
        agent=scout
    )
    compare_task = Task(
        description="对比 {product} 和每个竞品的优劣势",
        expected_output="详细对比矩阵",
        agent=analyst,
        context=[scan_task]
    )

    return Crew(
        agents=[scout, analyst],
        tasks=[scan_task, compare_task],
        process=Process.sequential
    )
```

api.py：

```python
import asyncio
import uuid
from fastapi import FastAPI, BackgroundTasks
from pydantic import BaseModel

app = FastAPI()

# 内存 job 状态（生产换 Redis）
jobs: dict[str, dict] = {}

class KickoffRequest(BaseModel):
    product: str

@app.post("/kickoff")
async def kickoff(req: KickoffRequest, background_tasks: BackgroundTasks):
    job_id = str(uuid.uuid4())
    jobs[job_id] = {"status": "running", "result": None}

    async def run():
        try:
            crew = build_crew()
            # kickoff_async() 不阻塞事件循环
            result = await crew.kickoff_async(inputs={"product": req.product})
            jobs[job_id] = {"status": "done", "result": result.raw}
        except Exception as e:
            jobs[job_id] = {"status": "error", "error": str(e)}

    background_tasks.add_task(run)
    return {"job_id": job_id}

@app.get("/status/{job_id}")
def status(job_id: str):
    return jobs.get(job_id, {"status": "not_found"})

@app.get("/health")
def health():
    return {"status": "ok"}
```

前端轮询 `/status/{job_id}` 直到 `status == "done"`，再拿 `result`。

同时跑多个 Crew：kickoff_for_each_async

如果你的场景是"给 10 个产品各跑一次竞品分析"，用 `kickoff_for_each_async`：

```python
from crewai import Crew

products = ["CrewAI", "LangGraph", "AutoGen", "Dify", "Coze"]
inputs_list = [{"product": p} for p in products]

crew = build_crew()
# 并发跑，自动管理事件循环
results = await crew.kickoff_for_each_async(inputs=inputs_list)

for product, result in zip(products, results):
    print(f"{product}: {result.raw[:200]}")
```

串行跑 5 个可能需要 10 分钟，并发只需要 2–3 分钟（取决于 LLM 限速）。

断点续跑：restore_from_state_id（CrewAI 1.14.5+）

Flow 跑到一半因为网络超时崩了？不用从头来。Flow 支持持久化 State，用 `restore_from_state_id` 从某个快照分叉：

```python
from crewai.flow.flow import Flow, listen, start, persist
from pydantic import BaseModel

class PipelineState(BaseModel):
    topic: str = ""
    research: str = ""
    article: str = ""

class ContentPipeline(Flow[PipelineState]):
    @start()
    @persist   # 每步执行后自动保存 state
    def pick_topic(self):
        self.state.topic = "CrewAI 2026 生产部署"

    @listen(pick_topic)
    @persist
    def do_research(self, topic):
        result = research_crew.kickoff(inputs={"topic": topic})
        self.state.research = result.raw

    @listen(do_research)
    @persist
    def write_article(self):
        result = writing_crew.kickoff(inputs={
            "topic": self.state.topic,
            "research": self.state.research
        })
        self.state.article = result.raw

pipeline = ContentPipeline()

# 正常跑，拿到 state_id
result = await pipeline.kickoff_async()
state_id = pipeline.state_id   # 保存下来

# 崩了？从某个已保存的快照恢复，分配新 state_id
recovery_pipeline = ContentPipeline()
result = await recovery_pipeline.kickoff_async(
    restore_from_state_id=state_id  # 从该快照加载 state 继续跑
)
```

`@persist` 装饰每个步骤，保证每步完成后 state 都写入 SQLite（默认）。`restore_from_state_id` 加载快照后用新 state_id 写后续结果，原始历史不丢。

> **注意：** `restore_from_state_id` 和 `from_checkpoint` 不能同时用，二选一。

Docker 容器化

Dockerfile：

```
FROM python:3.11-slim

WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .
EXPOSE 8000
HEALTHCHECK --interval=30s --timeout=10s \
  CMD curl -f http://localhost:8000/health || exit 1

CMD ["uvicorn", "api:app", "--host", "0.0.0.0", "--port", "8000"]
```

docker-compose.yml（含 Redis 用于生产级 job 状态存储）：

```
services:
  crew-api:
    build: .
    ports:
      - "8000:8000"
    environment:
      - OPENAI_API_KEY=${OPENAI_API_KEY}
      - SERPER_API_KEY=${SERPER_API_KEY}
      - REDIS_URL=redis://redis:6379
    depends_on:
      redis:
        condition: service_healthy
    restart: unless-stopped

  redis:
    image: redis:7-alpine
    healthcheck:
      test: ["CMD", "redis-cli", "ping"]
      interval: 10s
      timeout: 5s
      retries: 3
    restart: unless-stopped
```

```
# 启动
docker compose up -d

# 测试
curl -X POST http://localhost:8000/kickoff \
  -H "Content-Type: application/json" \
  -d '{"product": "CrewAI"}'

# 查状态
curl http://localhost:8000/status/{job_id}
```

成本控制：生产环境不能放任烧钱

一个 3-Agent Crew 单次约 $0.10–0.30，定时任务每天跑 100 次就是 $10–30/天。几个硬性规则：

```
agent = Agent(
    role="...",
    goal="...",
    backstory="...",
    max_iter=5,          # 禁止无限循环
    max_rpm=10,          # 每分钟最多 10 次 LLM 调用（防突发）
)

crew = Crew(
    agents=[...],
    tasks=[...],
    max_rpm=30,          # Crew 级别限速
)
```

模型路由策略（成本降 60% 的实操）：

任务类型	推荐模型	原因
网页搜索 + 信息提取	`gpt-4o-mini` / `claude-haiku-4-5-20251001`	够用，比 GPT-4o 便宜 10x
复杂推理 / 决策	`claude-sonnet-4-6` / `gpt-4o`	准确性更重要
最终输出生成	`claude-sonnet-4-6`	文字质量有要求
本地隐私数据	`ollama/llama3.1`	零 API 费用

部署后必做：开监控

生产 Crew 挂了你不知道，等于白部署。最省力的方式是接 **Langfuse**（开源，可自托管）：

```
pip install langfuse
```

```
import os
os.environ["LANGFUSE_SECRET_KEY"] = "sk-..."
os.environ["LANGFUSE_PUBLIC_KEY"] = "pk-..."
os.environ["LANGFUSE_HOST"] = "https://cloud.langfuse.com"  # 或自托管地址

# Langfuse 自动 patch CrewAI，无需改任何代码
import langfuse
langfuse.configure()

# 正常 kickoff，所有 LLM 调用自动上报
result = crew.kickoff(inputs={"product": "CrewAI"})
```

Langfuse Dashboard 可以看到每次 Crew 运行的：token 消耗、延迟分布、每个 Agent 的调用链、失败率。发现某天成本突增，直接 drill down 到具体的 Agent 调用。

CHAPTER 08

📚 Knowledge 知识库：让 Agent 读懂你的私有文档

为什么需要 Knowledge

LLM 只知道训练数据截止日前的公开内容——它不认识你公司的内部政策、产品手册、合同模板、历史决策文档。

CrewAI Knowledge 功能在 v0.70+ 引入，本质是**内置 RAG（Retrieval-Augmented Generation）**：你把文档喂给 Knowledge Source，它自动分块、向量化、存入本地向量数据库（默认 ChromaDB），Agent 执行任务时会自动检索相关段落注入上下文，不用再手动粘贴文档内容。

相比自己搭 LangChain RAG 链路：

	CrewAI Knowledge	自己搭 RAG
配置量	3 行代码	20–50 行
多 Agent 共享	自动，Crew 级别挂载	需手动传 retriever
Agent 个人知识库	Agent 级别单独挂载	需多套 retriever
适合场景	快速原型 + 中小规模	大规模生产，需精细控制

四种 Knowledge Source

1. StringKnowledgeSource — 直接用字符串

最简单，适合把少量结构化信息喂给 Agent（产品规格、公司简介、配置参数）：

```
from crewai.knowledge.source.string_knowledge_source import StringKnowledgeSource

# 产品规格书
product_spec = StringKnowledgeSource(
    content="""
    产品名称：JR AI 助手
    最大上下文：128K tokens
    支持语言：中文、英文、日文
    定价：免费版 100 条/天，Pro 版 ¥99/月
    部署方式：SaaS 云端，不支持私有化
    """,
    metadata={"source": "product_spec_v2", "version": "2.1"}
)
```

`metadata` 是可选的，加上后 Agent 可以知道引用的是哪个版本的文档——出了 Bug 排查时很有用。

2. PDFKnowledgeSource — 读取 PDF 文件

把 PDF 放进 `knowledge/` 目录（CrewAI 项目默认的知识库目录），然后：

```
from crewai.knowledge.source.pdf_knowledge_source import PDFKnowledgeSource

# 支持同时挂多个 PDF
hr_policy = PDFKnowledgeSource(
    file_paths=[
        "hr_policy_2026.pdf",
        "employee_handbook.pdf"
    ],
    chunk_size=1000,    # 每段 ~1000 tokens（默认 4000，文档细碎时调小）
    chunk_overlap=200   # 段与段之间的重叠（防止信息被切断）
)
```

文件路径相对于 `knowledge/` 目录。如果你的项目结构不同，也可以传绝对路径。

3. TextFileKnowledgeSource — 读取纯文本文件

```
from crewai.knowledge.source.text_file_knowledge_source import TextFileKnowledgeSource

docs = TextFileKnowledgeSource(
    file_paths=["changelog.txt", "api_reference.md"],
    chunk_size=800
)
```

Markdown、TXT、日志文件都支持。

4. 自定义 Knowledge Source

继承 `BaseKnowledgeSource`，实现 `load_content()` 方法，可以接入数据库、API、Confluence 等任意来源：

```
from crewai.knowledge.source.base_knowledge_source import BaseKnowledgeSource
from typing import Dict, Any
import httpx

class ConfluenceKnowledgeSource(BaseKnowledgeSource):
    space_key: str

    def load_content(self) -> Dict[str, Any]:
        # 从 Confluence API 拉取页面内容
        resp = httpx.get(
            f"https://your-domain.atlassian.net/wiki/rest/api/content",
            params={"spaceKey": self.space_key, "type": "page"},
            auth=("user@example.com", "YOUR_API_TOKEN")
        )
        pages = resp.json()["results"]
        return {page["title"]: page["body"]["storage"]["value"] for page in pages}

    def add(self) -> None:
        content = self.load_content()
        for title, body in content.items():
            self._save_documents([body])
```

挂载方式：Crew 级别 vs Agent 级别

Crew 级别（所有 Agent 共享）

```
from crewai import Agent, Task, Crew, Process

policy_source = PDFKnowledgeSource(file_paths=["company_policy.pdf"])
product_source = StringKnowledgeSource(content="产品规格...")

hr_agent = Agent(
    role="HR Specialist",
    goal="回答员工的人事政策问题",
    backstory="你是公司 HR，熟悉所有规章制度",
    verbose=True
)

product_agent = Agent(
    role="Product Advisor",
    goal="回答客户的产品问题",
    backstory="你是产品专家，熟悉所有产品规格和定价",
    verbose=True
)

crew = Crew(
    agents=[hr_agent, product_agent],
    tasks=[...],
    knowledge_sources=[policy_source, product_source],  # 全员共享
    verbose=True
)
```

Agent 级别（某个 Agent 专属知识库）

当不同 Agent 需要读不同的私密文档时（比如财务 Agent 才能访问报表），在 Agent 上单独挂：

```
from crewai.knowledge.source.pdf_knowledge_source import PDFKnowledgeSource

# 只有 CFO Agent 能看到财务报表
financial_report = PDFKnowledgeSource(file_paths=["q1_financial_report.pdf"])

cfo_agent = Agent(
    role="CFO Analyst",
    goal="分析财务数据，发现风险点",
    backstory="你是 CFO 顾问，专职财务分析",
    knowledge_sources=[financial_report]  # Agent 专属
)

general_agent = Agent(
    role="General Analyst",
    goal="整理市场信息",
    backstory="你负责外部市场分析"
    # 没有挂财务知识库
)
```

完整示例：HR 政策问答 Crew

这是一个实际跑通过的场景——员工提问 → Agent 查阅政策文档 → 给出准确回答：

```
from crewai import Agent, Task, Crew
from crewai.knowledge.source.pdf_knowledge_source import PDFKnowledgeSource
from crewai.knowledge.source.string_knowledge_source import StringKnowledgeSource

# 挂载知识库
policy_pdf = PDFKnowledgeSource(
    file_paths=["hr_policy_2026.pdf"],
    chunk_size=1000,
    chunk_overlap=150
)

faq_source = StringKnowledgeSource(
    content="""
    Q: 年假多少天? A: 入职 1 年以内 5 天，1-3 年 10 天，3 年以上 15 天。
    Q: 报销流程? A: 提交费用单 → 直属上司审批 → 财务部门 3 个工作日内到账。
    Q: 远程工作政策? A: 每周最多 3 天 WFH，需提前 1 天申请，试用期不适用。
    """,
    metadata={"source": "hr_faq_v3"}
)

# 配置 Agent
hr_specialist = Agent(
    role="HR Policy Specialist",
    goal="准确回答员工关于公司政策的问题，始终引用政策原文",
    backstory="""你是公司 HR 专员，对公司所有规章制度了如指掌。
    回答问题时必须引用具体政策条款，不能凭感觉猜测。""",
    verbose=True
)

# 配置 Task
answer_task = Task(
    description="员工问题：{question}\n\n请查阅公司政策文档，给出准确回答并引用相关政策条款。",
    expected_output="清晰的政策解答，包含：1) 直接回答 2) 相关政策原文引用 3) 如有例外情况请注明",
    agent=hr_specialist
)

# 组成 Crew
hr_crew = Crew(
    agents=[hr_specialist],
    tasks=[answer_task],
    knowledge_sources=[policy_pdf, faq_source],
    verbose=True
)

# 跑起来
result = hr_crew.kickoff(inputs={"question": "我入职刚好 3 年，年假还有几天可以用? "})
print(result.raw)
```

Agent 会自动在向量数据库里检索"年假"相关段落，把政策原文注入上下文后再回答——不是靠 LLM 凭记忆猜，是真的读了你的文档。

注意事项和踩坑

1. 向量化需要 Embedding API

默认使用 OpenAI `text-embedding-3-small`，需要设置 `OPENAI_API_KEY`。用其他模型：

```
crew = Crew(
    agents=[...],
    tasks=[...],
    knowledge_sources=[...],
    embedder={
        "provider": "openai",
        "config": {"model": "text-embedding-3-small"}
    }
)
```

用本地 Ollama embedding（零费用）：

```
embedder={
    "provider": "ollama",
    "config": {"model": "nomic-embed-text"}
}
```

2. 向量库缓存

第一次跑时会花时间向量化文档（大 PDF 可能要几分钟），之后会缓存在 `.crew_knowledge/` 目录里，再次跑秒启动。**文档更新后记得删缓存**重新向量化。

3. chunk_size 调优

文档类型不同，chunk 大小建议不同：

文档类型	推荐 chunk_size	原因
合同/法律文档	500–800	条款独立，小块检索精度更高
技术文档/API 文档	1000–1500	需要上下文连贯
长报告/白皮书	2000–4000	段落结构完整

匠人学院
JR ACADEMY

会搭多 Agent 了，接下来怎么用它找工作？

这本手册的在线版在 jiangren.com.au/wiki/crewai-guide，随官网更新。匠人学院还有 20+ 本同款 AI 工具指南（Claude Code、Cursor、n8n、Dify…）和 AI Engineer 方向的项目制课程——想知道 CrewAI 怎么写进简历、搭成能找工作的作品，来聊聊。

扫码进社群，多 Agent 实战答疑

把这本 PDF 转给还在靠单 Agent 打天下的同学。

jiangren.com.au