大模型原理与应用

第 5 课：多 Agent 协作与工作流

从单 Agent 到团队协作

看什么

- 为什么单 Agent 够用，但有时还不够
- 多 Agent 相对单 Agent 的真实收益与代价
- CrewAI 里的 agent、task、crew、flow、process 分别在做什么
- process、context、memory 在多 Agent 里各自负责什么

多 Agent 不是“多几个模型一起聊天”，而是把复杂任务拆成多个边界更清楚的角色与步骤。

先回到最基本的问题

- 一个 agent 已经能调用工具、读上下文、完成任务
- 那为什么还需要多个 agent

因为很多真实任务并不是“一步做完”，而是：

1. 先规划
2. 再检索
3. 再执行
4. 再检查
5. 最后汇总

如果这些事情都塞给一个 agent， 系统通常会越来越乱。

单 Agent 的优点

- 结构简单
- 上手快
- 调试容易
- 成本更低
- 对小任务通常已经足够

所以教学上， 单 Agent 仍然应该是第一步。

什么时候单 Agent 开始不够用

因为复杂任务里，单 Agent 常见几个问题：

- 什么都想做，职责不清
- 上下文越来越长，越来越乱
- 一边分析、一边执行、一边审查，容易混在一起
- 很难并行
- 自我检查通常不够可靠

这时，多 Agent 才开始真正有意义。

多 Agent 真正多出来了什么

- 分工更清楚
  planner、researcher、executor、reviewer 可以分别负责不同部分。
- 上下文更干净
  每个 agent 只看自己需要的信息，不必把所有东西塞进同一个上下文。
- 更适合长流程任务
  尤其是"分析 -> 执行 -> 检查 -> 汇总"这种链条。
- 更容易并行
  互不依赖的子任务可以同时做。
- 更容易做校验
  可以让一个 agent 产出结果，另一个 agent 专门审查。

但多 Agent 不是白来的

- 协调成本更高
- 设计不好会重复劳动
- 任务切分不好时，反而更慢
- 对简单任务，收益往往不如单 Agent

所以一个很重要的工程判断是：

不是任务越复杂越该上多 Agent，而是任务是否真的适合拆分。

先给一个简单心智模型

- 单 Agent：像一个人独立做项目
- 多 Agent：像一个小组协作做项目

重点不在“人多”， 而在：

- 分工是否合理
- 信息如何传递
- 谁负责检查
- 谁负责最后整合

在这个问题上，CrewAI 想解决什么

一句话：

CrewAI 是一个面向多 Agent 协作与工作流的工程框架。

它最适合回答的问题是：

- 多个 agent 怎么分工
- 任务怎么串起来
- 流程怎么组织
- 多角色系统怎么快速搭出来

四个核心概念

Agent

一个有角色、目标、工具的智能体

Task

一项具体任务

Crew

一组协作 agent

Flow

更高层的流程与状态编排

这四层加在一起， 就构成了 CrewAI 的基本世界观。

先看 Agent：谁来做事

一个 CrewAI agent 通常会有：

- role
- goal
- backstory
- tools
- memory
- delegation

所以它不像一个“空白模型”， 更像一个被赋予职责的队员。

再看 Task：任务怎么落地

- Task 是最小工作单元
- 它描述“谁做什么”
- 也描述“输出应该长什么样”

Task 里最常见的内容是：

- 任务描述
- 指定 agent
- 工具需求
- 输出格式

多 Agent 系统是否清楚， 往往首先看 Task 是否写清楚。

再往上一层：Crew 是怎么组织团队的

Crew 不是简单地把几个 agent 放在一起。

它真正决定的是：

- 谁和谁协作
- 按什么顺序协作
- 是否允许委派
- 结果如何汇总

所以 Crew 更像“执行团队”， 不是“角色清单”。

最上层：Flow 管什么

可以把 Flow 理解成：

- 更上层的流程经理
- 管状态
- 管分支
- 管何时调用 Crew

一个常见心智模型是：

- Flow 管流程
- Crew 管执行

这也是 CrewAI 和很多只讲 agent 的框架相比， 更容易搭出完整系统的原因。

进入 process：先记住两种最重要的流程形状

课堂先记两种最重要的 process：

Sequential

一个任务接一个任务执行

Hierarchical

由 manager 统一调度、拆分、检查

前者更直白， 后者更像“一个负责人带一个团队”。

`Process.sequential` 更像流水线

- 更像流水线
- task 顺序基本是提前写好的
- 先做第一步，再做第二步，再做第三步
- 更适合线性、稳定、好调试的流程

所以它最适合讲的是：

- 多个角色顺着协作
- 上一步输出，成为下一步输入

如果你只是想表达“多角色串起来做事”， `sequential` 往往已经够用。

`Process.hierarchical` 更像团队管理

- 更像团队管理
- manager 负责拆题、统筹、调度、把关
- specialist 分别负责子任务
- 更适合 manager + specialists 结构

它最适合讲的是：

- 谁来决定任务怎么拆
- 哪一步交给谁
- 最后谁来检查并综合

所以当你真正想体现“多 agent 的组织优势”， `hierarchical` 会更贴切。

课堂上怎么区分这两种 process

- `sequential`：先让学生看懂“多角色顺着做”
- `hierarchical`：再让学生看懂“manager 怎么带 specialist 做”

所以这门课里可以这样理解：

- `7/2-prog` 更像 `Process.sequential`
- `7/4-prog` 更像 `Process.hierarchical`

前者强调：

- 分步骤协作

后者强调：

- 分步骤协作 + 管理与把关

process 讲完后，再看 memory

单 Agent 里，memory 只是“记住之前说过什么”。

多 Agent 里，memory 还会变成：

- 团队共享状态
- 任务交接记录
- 后续 agent 的工作依据

所以 memory 在多 Agent 中的价值更大， 但噪声风险也更大。

在我们的例子里，memory 怎么体现

在这门课的例子里， memory 不先做成复杂数据库， 而是先做成一个最小版本：

- 一份可反复读取的项目经验记录

比如在 `7/4-prog` 里：

- manager 会先读一份 `project_memory.md`
- 里面写的是这个通信场景的历史偏好、输出习惯和保守策略
- 然后 manager 再拆题、再汇总

所以这里的 memory 不是“神奇记忆”， 而是：

- 把过去经验保留下来
- 在后续任务里再次使用

这也刚好能区分三件事

- `prompt`：这一轮用户新说了什么
- `context`：前一个 task 给后一个 task 传了什么
- `memory`：系统把过去经验留下来，并在下一轮或下一步继续使用

所以教学上可以先这样讲：

- 先会用 prompt
- 再会传 context
- 最后再引入 memory

这里很容易混：process 到底负责什么

前面讲了：

- `Process.sequential`
- `Process.hierarchical`

但它们真正负责的， 不是“保存记忆”， 也不是“传递内容”， 而是：

- 决定流程是什么形状
- 决定 manager 和 specialist 的关系更像什么

所以可以把它简单理解成：

- `process` 管“流程骨架”

那真正把多个 agent 串起来的是什么

在我们的例子里， 真正把多个 agent 串起来的， 主要是：

- `Task`
- `context`
- `manager`

尤其是 `context`。

因为后一个 task 往往要读取前一个 task 的输出， 否则多个 agent 就只是“并排存在”， 而不是“真的协作”。

所以课堂上很值得强调一句：

> `process` 提供角色关系，`context` 提供信息流。

用 7/4-prog 一次看懂这三层

在 7/4-prog 里：

- process=Process.hierarchical 说明这里是 manager + specialists 结构
- context=[...] 说明前面任务的结果会传给后面任务
- project_memory.md 说明 manager 还会带着历史项目偏好去拆题和汇总

所以这个例子特别适合帮助同学分清：

- process：流程骨架
- context：任务交接
- memory：历史经验

什么时候才真的该上多 Agent

适合：

- 角色分工明显
- 需要任务委派
- 有较长流程
- 需要审查或复核
- 某些子任务可以并行

不太适合：

- 非常小的任务
- 强依赖单步推理的任务
- 想严格控制单一变量的实验

CrewAI 在这门课里的位置

这门课里， CrewAI 最适合拿来做：

- 多 Agent 原型
- 教学演示
- 多角色流程 demo
- 与单 Agent 框架做对照

比如：

- researcher + analyst + reporter
- planner + tool-user + reviewer
- manager + specialist + checker

它和单 Agent 框架的关系

你可以把它理解成：

- `PydanticAI` 更适合讲单 Agent、工具调用、结构化输出
- `CrewAI` 更适合讲多 Agent、任务协作、流程组织

所以两者不是互相替代， 而是分别代表：

- 单 Agent 最小闭环
- 多 Agent 协作系统

Claude Code / Codex / CrewAI 各自代表什么

- Claude Code 的 `sub-agent` 很像编程场景里的任务分解机制
- Codex 强调 multi-agent workflows 和并行 delegation
- CrewAI 则是让你自己显式设计 agent、task、crew、flow

所以：

- Claude Code / Codex 更产品化
- CrewAI 更框架化

先给这章一个总判断

不要把“多 Agent”理解成“更高级的单 Agent”。

更准确的理解是：

- 单 Agent：先把一个角色做好
- 多 Agent：再把多个角色组织起来

先学会前者， 再学后者， 系统会更稳。

这节课你该带走什么

- 多 Agent 的核心价值是分工、并行和校验
- 它的代价是协调、复杂度和调试成本
- CrewAI 的关键词是：agent、task、crew、flow、process
- `process / context / memory` 三者不要混
- CrewAI 适合快速做一个“多角色系统”，而不是最底层 runtime 研究

第 7 章收束

先把单 Agent 做通， 再把多 Agent 做清楚。

CrewAI 的价值， 就在于把“团队协作”这件事， 从概念变成了一个可以快速搭出来的工程系统。