LEARNING NOTE 02

AI 产品经理学习笔记 02

从大模型到 Prompt 与 RAG

这是我的 AI 产品经理学习记录。前面的学习更多集中在需求分析、用户场景、产品流程和 PRD 等传统产品基本功。这一部分开始真正进入 AI 产品本身：大模型到底是什么、Prompt 在产品里承担什么角色，以及为什么很多 AI 应用最终还需要 RAG。

> *我暂时不准备从复杂的数学推导和模型训练开始，而是先站在产品经理的角度理解这些技术：它能解决什么问题、有什么边界、什么时候应该用，以及出错以后应该从哪里排查。*

本篇属于：第一层 · 保底必学 | 第二、三、四章

学习记录 · 公开整理版

2026 · AI Product Management

第二章：大模型基础认知

1. 大模型到底是什么？

如果暂时不讨论 Transformer、注意力机制和训练过程，可以先用一个面向产品经理的版本理解大语言模型：

> *大语言模型会根据当前已经获得的上下文，对接下来可能出现的 token 进行概率预测，并不断重复这一过程，最终形成完整输出。*

这里的 token 可以暂时理解成模型处理文本时使用的基本单位，它并不完全等同于一个汉字或者一个单词。

这种机制带来了 AI 产品非常重要的一个特点：模型输出具有概率性。

传统程序通常强调确定性。例如：

```
1 + 1 → 2
```

当输入、程序逻辑和运行环境没有变化时，我们通常预期得到相同结果。

但如果任务变成：“帮我写一条游戏角色宣传文案。”模型可能第一次强调角色性格，第二次强调剧情经历，第三次又选择完全不同的语言风格。即使输入看起来相同，也不应该简单假设所有输出都会完全一致。

> *AI 不是传统意义上的完全确定性系统，而是带有概率性的生成系统。*

这件事会直接影响后面的产品设计、测试、评估和异常处理。我们不能只设计“模型正常回答”的情况，还必须考虑：模型理解错了怎么办？结果质量不稳定怎么办？同样输入为什么得到不同结果？用户能不能接受这种不确定性？

2. 大模型擅长什么？

不同模型的实际能力存在差异，而且模型能力仍在持续变化。因此，与其把“LLM 能做什么”当成一张永久不变的能力清单，不如先从常见 AI 产品任务出发，理解几类高频能力。

理解

- 文本理解；
- 意图识别；

- 信息提取；
- 文本分类；
- 情感或倾向分析；
- 从自然语言中识别任务要求。

比如用户输入：“这个角色我挺喜欢的，就是剧情后面写得越来越不像她了。”产品可能希望模型进一步识别：

```
反馈对象：角色塑造
情绪：总体正向，但存在不满
具体问题：角色一致性下降
```

这里主要利用的是模型对自然语言的理解和信息提取能力。

生成

- 文案；
- 邮件；
- 摘要；
- 改写；
- 对话；
- 内容草稿；
- 说明文字。

但“能生成”本身并不意味着产品成立。真正进入产品设计以后，还需要继续问：生成给谁？在哪个场景生成？生成以后直接使用，还是需要人工修改？怎样判断生成结果够不够好？

转换

大模型还很适合处理不同信息表达形式之间的转换。例如：

非结构化文本 → 结构化 JSON

长篇文档 → 摘要

用户自然语言 → 系统能够处理的任务参数

这类能力对 AI 产品尤其重要，因为真实用户往往不会按照系统数据库字段输入内容。用户说的是自然语言，而产品内部需要的是结构化信息。大模型可以承担其中一部分“翻译层”的作用。

推理任理

- 任务拆解；
- 多步骤处理；
- 代码生成与解释；
- 数据解释；
- 工具选择与调用；
- 基于上下文进行一定程度的规划。

但这里需要避免一个误区：模型表现出推理能力，不等于它的每一步推理都一定正确。任务越长、依赖越多、步骤越复杂，错误也可能在流程中累积。

> *做复杂 AI Workflow 或 Agent 时，产品设计不能只考虑模型“理论上能完成什么”，还要考虑它能不能稳定完成。*

3. 大模型不擅长什么？

相比知道模型“会什么”，我觉得产品经理更需要知道它“不可靠在哪里”。很多 AI 产品问题，并不是功能完全做不出来，而是产品默认模型过于可靠。

3.1 幻觉

大模型可能生成语言非常自然，但事实并不正确的内容。比如用户询问某家公司的最新年度收入，如果模型没有可靠数据来源，却仍然生成一个看起来合理的数字，用户很容易被流畅的表达误导。

> *回答流畅 ≠ 回答正确。*

这也是为什么在涉及事实性问题时，经常需要进一步考虑搜索、数据库、RAG、工具调用、来源引用和人工复核。

3.2 输出具有不确定性

同一个问题多次生成，结果可能存在差异。这种差异可能体现在措辞、结构、重点、细节，甚至最终判断。

因此 AI 产品测试不能只做一次“我试了一遍，感觉回答不错”。一次成功不能证明产品稳定。更可靠的做法是准备一组具有代表性的测试案例，并观察模型在不同输入和多次运行下的整体表现。

这也是后续 Evaluation 会非常重要的原因。

3.3 模型并不知道所有知识

模型本身的训练知识并不能自然覆盖所有产品需要的信息。例如：

- 企业内部资料；
- 用户私人数据；
- 当前业务数据库；
- 实时价格；
- 最新政策；
- 最新产品状态；
- 某个游戏团队内部尚未公开的角色设定。

> *“模型本身知道什么”与“产品当前需要什么信息”是两件不同的事情。*

RAG、联网搜索、API、数据库查询和工具调用等方案，很大一部分就是在解决这个问题。

3.4 长上下文并不意味着无限塞资料

一个很容易产生的想法是：模型支持的 Context Window 越长，那是不是把所有资料都放进去就可以？实际产品中并没有这么简单。

- 上下文增加通常意味着更多 token 消耗；
- 可能增加响应延迟与调用成本；
- 大量无关信息会降低有效信息密度；
- 真正关键的内容可能被淹没。

> *“模型能够接受多少上下文”和“产品应该给模型多少上下文”并不是同一个问题。*

这也让我开始意识到，Context Management 本身就是 AI 产品设计的一部分。需要思考：这一轮任务真正需要什么信息？哪些历史信息可以删除？哪些信息应该总结？哪些知识应该检索后再加入？哪些信息只在特定步骤需要？

4. AI 产品经理应该怎么看模型？

至少在我目前这个学习阶段，我不会优先把时间放在 Transformer 的完整数学推导、梯度下降公式、CUDA、底层训练工程等内容上。这些知识当然有价值，但和我现阶段“先理解 AI 产品怎么工作”的目标相比，优先级没有那么高。

Capability | 能力

这个模型能做什么？例如理解、生成、总结、代码、视觉理解、工具调用等。更进一步还需要看：它在我的具体业务任务上到底表现怎么样。模型通用能力很强，不代表它在某一个产品任务上一定是最合适的选择。

Quality | 效果

模型输出好不好？这里不能只看“聪不聪明”，而应该跟任务本身的成功标准对应。内容生成可能关注角色一致性、卖点命中、事实准确、语言自然度与安全性；信息抽取则可能更关注字段准确率、遗漏率和格式正确率。

Latency | 延迟

用户需要等多久？一个模型效果更好，但如果每一次交互都需要等待很久，仍然可能影响产品体验。实时聊天和后台批量生成，对延迟的容忍程度显然不同。

Cost | 成本

模型调用存在成本。真正进入产品以后，需要考虑一次任务消耗多少 token、用户每天可能调用多少次、不同模型之间成本差多少，以及是否所有任务都需要使用最强模型。

> *模型选择不能只比较能力，还要放回“能力 × 成本 × 产品场景”中判断。*

Context Window | 上下文能力

模型一次能够处理多少上下文？但比最大长度更重要的是产品实际需要携带什么信息。Context Window 是模型能力边界之一，而 Context Management 是产品设计问题。

Structured Output | 结构化输出

如果模型结果需要继续进入程序流程，就需要关注它是否能够按照稳定结构返回内容。例如：

```
{
  "intent": "refund",
  "confidence": 0.92,
```

```
  "reason": "用户明确提出退款请求"
}
```

相比一大段自然语言，结构化输出更容易被后端继续处理。

Tool Use | 工具调用

Tool Use 是另外一项能力。它关注的是：模型能否根据当前任务判断是否需要调用外部工具，并正确生成调用所需参数。

例如用户问：“查一下东京明天会不会下雨。”模型自身不应该凭训练记忆回答实时天气，而可以判断需要调用天气工具。工具返回结果以后，模型再组织成自然语言回答。

> *Structured Output 和 Tool Use 是相关但不同的能力：前者更关注输出结构，后者更关注模型与外部系统交互。*

Multimodal | 多模态

还需要看模型是否支持文字、图片、音频、视频，或其他模态组合。如果产品任务涉及 UI 截图理解、图片分析或者语音交互，那么单纯比较文本能力已经不够。

第三章：Prompt Engineering

1. Prompt 到底是什么？

以前我很容易把 Prompt 理解成“给 AI 输入的一句话”。但站在产品角度以后，我觉得这种理解太窄了。

> *在 AI 产品中，Prompt 更像是 AI 功能的一部分任务定义、上下文组织和行为规则。*

它决定模型当前是什么角色、需要完成什么任务、能够看到什么资料、应该遵守什么限制，以及最后应该返回什么。

Anthropic 的提示工程文档强调，在优化提示之前应该先定义成功标准和建立评测方式。对我来说，这一点比“学几个万能提示词模板”更重要：真正做产品时，不应该只问 Prompt 看起来写得够不够专业，而应该问它能不能让当前任务更稳定地达到我们定义的成功标准。

2. Prompt 的基础结构

Prompt 并不存在一个所有模型、所有任务都必须完全照搬的固定模板。为了训练自己的思考方式，我目前会用下面这个结构检查 Prompt 是否完整：

Role + Task + Context + Constraints + Output + Examples

Role | 角色

告诉模型当前承担什么角色。例如：“你是一名二次元游戏内容运营助手。”Role 的作用不是单纯增加一句“你是一名专家”，而是帮助限定任务语境。

Task | 任

明确到底要做什么。例如：“根据提供的角色设定生成角色宣传文案。”相比“帮我写点东西”，任务边界明显更清晰。

Context | 上下文

告诉模型完成任务需要知道的背景。例如：

```
游戏类型：二次元 RPG
目标用户：18-25 岁玩家
角色定位：……
活动主题：……
```

如果不给必要背景，却要求模型输出非常符合产品要求的内容，本身就是一种矛盾。

Constraints | 限制

明确哪些事情不能做或者必须遵守。例如：

```
不得捏造角色设定
文案不超过 120 字
不得加入未提供的剧情信息
语言风格需要符合角色设定
```

限制越具体，模型通常越容易理解产品要求。但也不能无限增加规则，后面还需要验证这些规则是否真正有效。

Output | 输出

定义模型需要返回什么。例如：

```
{
  "title": "",
  "copy": "",
  "selling_point": ""
}
```

如果模型结果后续还需要进入程序，这一步尤其重要。

Examples | 示例

还可以给模型提供输入和理想输出示例。例如：

```
Input:
角色 A 的设定……

Output:
……
```

Few-shot Prompting 可以先简单理解成：通过少量示例帮助模型理解我们期望的任务模式和输出方式。它并不是所有场景都必须使用，但在一些格式、风格或者分类边界不容易只靠文字解释清楚的任务里很有价值。

3. 为什么 Structured Output 很重要？

如果只是普通聊天，模型返回自然语言通常没有问题。但一旦 AI 结果需要接入真正的系统流程，问题就发生了变化。

例如一个客服系统需要识别用户意图。如果模型回答：“根据用户刚才的描述，我觉得他应该是想申请退款。”这句话人类可以理解，但程序还需要再次解析到底是什么 intent、置信度是多少、为什么这么判断。

如果要求模型返回：

```
{
  "intent": "refund",
  "confidence": 0.92,
  "reason": "用户明确表达了退款诉求"
}
```

intent = refund → 进入退款流程

> *结构化输出的价值在于：让模型输出更容易成为软件系统的一部分，而不仅仅是给人阅读的一段话。*

当然，结构化并不等于内容一定正确。格式正确和语义正确仍然是两件事情。

4. Context 到底是什么？

Context 可以理解成模型在当前这一轮任务中能够获得的信息。可能包括：

- 用户当前输入；
- System Prompt；
- 历史聊天记录；
- 用户资料；
- 搜索结果；
- RAG 检索内容；
- 工具返回结果；
- 当前业务状态。

> *我比较喜欢把它理解成：模型这一轮“考试”允许看的资料。*

如果资料里没有答案，模型却被要求一定回答，就更容易产生不可靠内容；如果资料太多、太乱，又可能让真正重要的信息变得不明显。所以 Context 不是越多越好，而应该是和当前任务相关、足够、清晰。

5. Prompt 常见问题

问题一：指令太模糊

例如“分析这个产品”。“分析”到底分析什么？用户、商业模式、体验还是技术？可以改成：“从目标用户、核心需求、产品流程、AI 能力、商业价值和主要风险六个维度分析该产品。”这样模型更容易理解任务范围。

问题二：输出要求不清晰

例如只要求“输出分析结果”。模型可以用很多不同形式回答。如果这个结果后面需要进入固定流程，就应该进一步定义字段、结构、长度等。

问题三：Context 不够

例如要求模型分析某个产品，却没有告诉它产品是什么、用户是谁、核心功能是什么。这时模型只能依赖自身已有知识或者进行推断，而这些推断未必可靠。

问题四：Context 太多

另一个极端是把几十页完全没有筛选过的资料全部塞进 Prompt。其中大量信息可能与当前任务无关，结果不仅增加 token 和成本，也可能让真正重要的信息更难被利用。

问题五：Prompt 不断变长

这是我觉得很值得警惕的一种情况。模型第一次出问题，就加一句“不要……”，第二次又加“一定要……”，第三次继续“千万不能……”，最后 Prompt 变成几千字的规则集合。

> *这不一定说明 Prompt 写得越来越专业。有时反而意味着产品逻辑本身没有被拆清楚，只是在不断用新的文字规则修补旧问题。*

遇到这种情况，需要重新判断：这是 Prompt 问题？Context 问题？模型能力问题？产品流程问题？还是这个任务本身应该被拆成多个步骤？Prompt Engineering 不应该变成无限“打补丁”。

第四章：RAG 基础

1. 为什么会需要 RAG？

假设用户问：“我们公司今年的差旅报销规定是什么？”大模型本身通常并不知道某一家公司的内部制度。即使模型知道一些通用的报销规则，也不能把这些通用知识当成这家公司的真实制度。

最直接的思路是：

找到公司内部报销文件 → 找到相关内容 → 提供给模型 → 基于资料回答

这就是 RAG 的核心思想之一。RAG 的全称是 Retrieval-Augmented Generation，即“检索增强生成”。

> *我现在对 RAG 最简单的理解是：先找资料，再让模型基于资料回答。*

Microsoft Learn 对 RAG 的概括同样强调：通过检索外部或专有内容，为生成模型提供 grounding context，使回答能够基于这些数据，而不只依赖模型训练时获得的知识。

2. RAG 的基础流程

一个便于入门理解的经典 RAG 流程可以写成：

Document → Chunking → Embedding / Index → Retrieval → Context → LLM → Answer

真实生产系统可能远比这复杂，也可能加入关键词搜索、混合检索、重排序、查询改写、权限控制等环节。但在第一阶段，先把主链路理解清楚更重要。

Step 1：Document | 准备资料

数据来源可能是 PDF、Word、网页、Markdown、数据库内容或者企业内部文档。例如《员工手册》《差旅管理办法》《角色世界观设定》《客服知识库》。

> *RAG 的第一步并不是调用大模型，而是先确定模型回答时应该依据什么资料。*

Step 2：Chunking | 切块

如果一份 PDF 有几百页，并不需要用户每问一个问题就把整份 PDF 全部塞给模型。用户的问题通常只和其中少量内容有关，因此常见做法是把大文档拆成相对较小的内容块。

```
完整员工手册
↓
第一块：考勤
第二块：报销
第三块：差旅
第四块：休假
……
```

但“块越小越好”同样不成立。切得太小可能破坏完整语义；切得太大又会混入过多无关信息。因此 Chunking 本身也会影响最终检索效果。

Step 3：Embedding | 向量表示

Embedding 可以先简单理解成：把文本映射成一种便于计算机比较语义关系的向量表示。理想情况下，语义更加接近的内容，在对应的向量空间中也更容易被判断为相关。

例如用户问“出差住酒店最高能报多少？”，文档里可能写的是“一线城市住宿费用标准上限为……”。即使两边没有完全使用相同关键词，语义检索仍可能把它们识别成相关内容。

Step 4：Index | 建立索引

有了需要检索的文档内容以后，还需要建立能够高效查询的数据结构。入门 RAG 经常接触向量索引，但真实的检索系统并不只有向量检索一种方式。

- 关键词检索；
- 向量检索；
- 混合检索；
- 过滤条件；
- 重排序。

> *我现在更愿意把 Index 理解成：让后续 Retrieval 能够快速从知识集合中找到候选信息。*

Step 5：Retrieval | 检索

用户提出“差旅住宿标准是多少？”系统根据用户问题，从知识库中寻找最相关的内容。这里非常关键，因为如果这一阶段找错了资料，后面的模型再强，也可能在错误资料上生成一个非常流畅的错误答案。

> *RAG 的质量不仅取决于 LLM，还取决于 Retrieval。*

Step 6：Context | 加入上下文

找到相关片段以后，把它们作为 Context 提供给模型。例如：

```
用户问题：
差旅住宿标准是多少？

检索内容：
《差旅管理办法》第 X 条……
```

模型不再只依赖自身训练记忆，而是得到了一组和当前问题相关的资料。

Step 7：Generation | 生成答案

最后模型基于用户问题和检索结果生成回答。例如：“根据当前提供的《差旅管理办法》，一线城市住宿费用标准为……”

如果产品需要更强的可追溯性，还可以同时返回来源信息。但这里要注意：RAG 不会天然自动产生可靠引用。只有当检索阶段保留了文档来源、页码、章节或其他 metadata，并且产品明确把这些信息展示出来时，才可能实现真正有用的来源追溯。

3. RAG 到底解决什么问题？

私有知识

- 企业制度；
- 内部文档；
- 客服资料；
- 产品说明；
- 项目知识库。

这些内容通常不会自然存在于通用大模型可直接访问的知识中。

特定领域知识

例如某个特定公司的业务规则、某款游戏的角色设定、内部研发文档等。对于医疗、法律、金融等高风险领域，也可以使用检索增强方式提供专业材料，但这里尤其不能把“接入 RAG”理解成“答案从此一定正确”。

这类场景仍然需要更严格的数据来源、权限、评估和人工核验机制。

信息量很大的知识集合

如果资料规模已经大到不适合直接全部放进 Prompt，就可以考虑通过检索只找到当前任务所需要的部分。

把所有资料都给模型 → 先找到最相关资料，再给模型

时效变化的信息

如果知识经常更新，与其完全依赖模型训练时间点的知识，产品可以从一个持续维护的数据源中检索最新内容。但最终能否做到“最新”，依然取决于外部数据源本身是否真的及时更新。RAG 只是让模型能够使用这些数据，并不会自动保证数据源永远最新。

可追溯性

如果系统保留了检索内容和来源信息，就有机会让用户知道这个答案是根据什么资料得到的。例如：

```
来源：《员工手册》
章节：差旅管理
更新时间：……
```

相比一个完全不知道依据来自哪里的答案，这种方式更容易被核验。

4. RAG 不等于“没有幻觉”

这是我认为学习 RAG 时非常需要提前建立的认知。很容易产生一种误解：加了知识库以后，模型就不会再答错。实际上并不是。

Retrieval + Generation

所以问题至少可能来自两个阶段。

Retrieval Error | 检索错误

系统没有找到真正相关的资料。例如用户问“差旅住宿标准是多少？”，系统却召回“餐饮补贴规定”。这时候模型得到的输入本身就错了。

Generation Error | 生成错误

另一种情况是资料其实找对了，但模型在理解、归纳或者表达时出现错误。例如原文写“住宿标准上限 500 元”，模型却回答“住宿补贴为 500 元”。看起来只差几个字，但业务含义已经发生变化。

> *以后看到一个 RAG Bad Case 时，第一件事情不应该是“是不是换个更强的模型”，而应该先问：到底是“没有找到正确资料”，还是“资料已经找对，但模型回答错了”？*

这是两个完全不同的问题。

如果是 Retrieval Error，需要进一步检查 Chunking、Embedding、Query、Index、Retrieval、Top-K、过滤、重排序等环节；如果是 Generation Error，则更可能需要检查 Prompt、Context 组织、模型能力、输出约束和 Evaluation。只有先知道错误发生在哪一层，才知道应该优化哪里。

5. 从 Prompt 到 RAG，我开始理解“AI 产品不是一个模型”

整理到这里以后，我对 AI 产品的理解又发生了一点变化。最开始看 AI 应用时，很容易把注意力全部集中到模型：用哪个模型？Prompt 怎么写？模型够不够聪明？

但继续学习以后会发现，一个真实的 AI 产品往往远不只是：

用户 → LLM → 答案

更可能是：

用户输入 → 意图理解 → 上下文组织 → 知识检索 / 工具调用 → 模型处理 → 结构化输出 → 结果验证 → 产品展示 → 用户反馈

模型只是这条链路中的一个重要组成部分。Prompt、Context、RAG、Tool Use、Structured Output 看起来是不同的技术概念，但从产品角度来看，它们最终都在回答类似的问题：为了让用户稳定完成任务，我需要给模型什么信息、怎样约束它、什么时候让它访问外部世界，以及怎样处理它的不确定性?

写在这一阶段学习之后

如果只看名词，大模型、Prompt、Embedding、RAG 很容易让 AI 产品学习变成一场“概念收集”：今天学一个 Prompt，明天学一个 RAG，后天再学 Agent，最后知道很多词，却说不清它们为什么会出现在一个产品里。

所以我现在更希望按照问题来理解这些能力。

任务定义不清 → 先看 Prompt

模型缺少当前任务信息 → 先看 Context

真正需要的信息存在外部知识库 → 再考虑 RAG

模型需要查询或操作外部系统 → 再进入 Tool Use / Agent

这样理解以后，这些概念就不再是彼此孤立的技术名词，而是不同问题对应的不同解决方式。

> *第一，AI 是概率系统，不要默认模型永远正确。*
>
> *第二，Prompt 的目标不是“写得复杂”，而是让任务更稳定地达到成功标准。*
>
> *第三，RAG 不是为了让产品看起来更高级，而是在模型缺少外部知识时，为它找到真正需要的资料。*

接下来继续往后学习时，我也希望保持同一个顺序：先理解产品问题，再学习技术方案，而不是反过来为了使用某种 AI 技术，再去寻找一个勉强能够套进去的场景。

参考资料

1. Anthropic Claude Platform Documentation：Prompt Engineering / Define success criteria and evaluations。

2. Microsoft Learn：Retrieval-Augmented Generation (RAG) overview。

说明：本文主要作为个人学习记录整理，重点记录我现阶段对大模型、Prompt 与 RAG 的产品侧理解；后续随着实践和学习继续深入，其中部分认识也可能继续修正和补充。