# 大模型原理与应用

## Prompt 方法｜把预训练能力真正调出来

从 zero-shot 到工具、检索与反思

# 这一讲看什么

- Prompt 为什么会变成大模型的运行时接口
- zero-shot、few-shot 和 in-context learning 有什么差别
- Chain-of-Thought 为什么会显著改变推理表现
- tool use、RAG、reflexion 为什么会把 prompt 推向系统设计
- 写 prompt 时哪些习惯最重要

# 本讲主线

1. 先理解 prompt 为什么不是“随便问一句话”
2. 再看几种最常见的 prompting 方法
3. 然后理解 prompt 怎样和工具、检索、反馈结合
4. 最后收束到一套真正可用的 prompt 写法

# Prompt 为什么会突然变得重要

预训练模型学会的是：

- 语言分布
- 常见知识
- 上下文条件下的模式延续

但这些能力不会自动变成一个好用的系统。

Prompt 的作用就是：

- 告诉模型现在要做什么
- 给它任务边界
- 给它输入格式
- 把潜在能力调出来

# Prompt 是大模型的运行时接口

同一个模型，换一种 prompt，结果可能完全不同。

所以 prompt 不只是“提问”：

- 它是给模型下任务
- 它是给模型设约束
- 它是给模型补上下文
- 它也是一种轻量编程方式

这也是为什么很多人会说：

> Prompt 是和大模型沟通的 lingua franca

# 为什么 Prompt 既像艺术也像工程

一方面，它确实有点像“试出来”的：

- 模型是黑箱
- 不同模型偏好不同
- 同一句话改几个词，效果就可能变

但另一方面，它也不是纯玄学：

- 已经有一批稳定有效的方法
- 很多技巧可以复用
- 好 prompt 往往有清楚结构

# 最基础的一类：zero-shot

zero-shot 的意思是：

- 不给例子
- 直接告诉模型要做什么

例如：

- 总结一段文字
- 把一段中文翻成英文
- 解释一个 Transformer 概念

它的优点是：

- 快
- 省上下文
- 很适合简单任务

# zero-shot 什么时候会不够

当任务开始变复杂时，模型很容易：

- 理解偏题
- 输出格式不稳定
- 推理步骤跳跃
- 把“像答案的话”当成答案

所以 zero-shot 更像：

- 最小起点

而不是：

- 通用终点

# 下一步：few-shot / k-shot

few-shot 的思路很直接：

- 不只是告诉模型任务
- 还给它看几个例子

这也常被叫作：

- in-context learning

常见是：

- 1-shot
- 3-shot
- 5-shot

# few-shot 真正解决了什么

它最擅长解决的不是“模型不会”，而是：

- 模型不知道你想要哪种风格
- 模型不知道你想要哪种格式
- 模型不知道这个任务里的隐含规则

也就是说，few-shot 更像：

- 用例子来定义任务

# 一个简单判断

如果任务：

- 规则清楚
- 步骤不太长
- 输出格式明确

few-shot 往往就已经很好用。

如果任务：

- 逻辑链长
- 中间步骤容易错
- 最终答案需要推出来

那就要进一步考虑 reasoning prompt。

# Chain-of-Thought 在做什么

CoT 的核心不是“让模型更会说话”，而是：

- 让模型把中间推理显式展开

这对下面几类任务特别重要：

- 数学
- 程序设计
- 多步逻辑判断
- 复杂规划

# 为什么 CoT 会有效

因为很多任务错，不是错在最后一句， 而是错在中间某一步跳过去了。

CoT 的价值是：

- 把隐式推理变成显式推理
- 减少一步跳到答案的幻觉
- 让复杂任务拆成可检查的小步骤

# CoT 有两种常见形式

- multi-shot CoT
  - 先给模型看几段“题目 + 推理过程 + 答案”
- zero-shot CoT
  - 不给例子，只提示它“请一步一步想”

后者最经典的形式就是：

- `Let's think step by step.`

# zero-shot CoT 为什么常常够用

因为很多模型已经在预训练和后训练里见过大量：

- 解释型回答
- 分步推理
- 教学式文本

所以一句简短提示， 就能把模型拉到更“会推理”的回答模式里。

# 但 CoT 也不是万能的

它会带来几个现实问题：

- 推理变长，成本更高
- 模型可能一本正经地胡说
- 有时会“编造步骤来支撑错误答案”

所以关键不是“让它想得更长”， 而是“让它想得更稳”。

# Self-consistency 在补什么

如果一条推理链不一定可靠，一个自然想法就是：

- 让模型多想几次
- 采样多条不同推理路径
- 最后取更一致的答案

这就是 self-consistency。

# Self-consistency 的直觉

它有点像：

- 不只听一个人的第一反应
- 而是让模型给出多种候选 reasoning path
- 再看哪个结果最稳定

它常见于：

- 数学问答
- 程序调试
- 需要多数投票的推理任务

# Prompt 已经开始接近“系统设计”

做到这里，prompt 就不只是文字技巧了。

因为你已经开始考虑：

- 怎样组织输入
- 怎样引导推理
- 怎样做多次采样
- 怎样汇总结果

这已经很像一个最小工作流。

# tool use 为什么重要

仅靠模型内部参数，系统总会遇到边界：

- 算不准
- 查不到最新信息
- 改不了代码
- 访问不了数据库

tool use 的意义就是：

- 让模型在需要时调用外部能力

# tool use 真正改变了什么

它把模型从“只会回答”变成：

- 会查
- 会算
- 会调用程序
- 会把外部结果再组织成答案

这对降低 hallucination 很关键。

因为有些问题不该靠“猜”， 而该靠：

- 检索
- 执行
- 验证

# RAG 为什么会自然出现

很多时候，模型不是不会表达， 而是缺上下文。

RAG 的核心就是：

- 先取相关资料
- 再把资料塞进 prompt
- 最后基于这些资料回答

它解决的是：

- 知识更新
- 私有知识接入
- 可解释性

# RAG 带来的好处

- 不必每次都重新训练模型
- 更新知识更快
- 可以附引用
- 更容易知道答案从哪里来

所以它本质上是：

- 用检索系统补模型记忆

# Reflexion 在做什么

还有一种很重要的思路是：

- 先让模型做一次
- 再让模型回头审自己
- 如果发现问题，再修一次

这就是 reflexion。

# Reflexion 为什么有用

很多任务第一次答不好，不是因为模型完全不会， 而是因为：

- 第一次太快
- 没有检查
- 没有根据环境反馈修正

所以多一轮：

- critique
- reflect
- revise

结果往往会更稳。

# 看到这里，可以把 Prompt 方法分成四层

第一层：

- zero-shot / few-shot

第二层：

- CoT / self-consistency

第三层：

- tool use / RAG

第四层：

- reflexion / 多轮修正

它们不是互斥的， 而是常常叠加使用。

# 还需要理解一个基本术语组

- system prompt
- user prompt
- assistant

这三个角色决定了：

- 谁在设规则
- 谁在提任务
- 谁在生成输出

# system prompt 在做什么

system prompt 一般负责：

- 角色设定
- 风格约束
- 输出规则
- 安全边界

它通常不直接回答问题， 而是定义：

- 这个模型应该以什么方式回答

# user prompt 在做什么

user prompt 就是：

- 当前这次真正的任务

比如：

- 总结这篇论文
- 根据这张图分析预测质量
- 写一段 Python 代码

所以 user prompt 更像：

- 当前请求

# assistant 是什么

assistant 指的是：

- 模型最终生成的内容

如果系统设计得更复杂， assistant 输出里还可能包含：

- 工具调用
- 中间推理
- 结构化结果

# Prompt 最常见的错误，不是“不够高级”

而是：

- 任务没说清楚
- 输出格式没说清楚
- 上下文没给全
- 约束条件没写明

也就是说，很多 bad prompt 的问题， 本质上不是技巧不够， 而是规格说明不够。

# 一个好 prompt 往往有清楚结构

常见结构可以是：

- 任务是什么
- 输入材料是什么
- 输出格式是什么
- 约束条件是什么
- 如果需要，给示例

这比“扔一大段文字过去”稳定得多。

# 为什么结构化 prompt 很重要

因为它能减少模型自己猜：

- 哪段是背景
- 哪段是输入
- 哪段是日志
- 哪段是你真正的问题

所以我们会常见到这种写法：

- `<context>...</context>`
- `<log>...</log>`
- `<task>...</task>`

# 另一个高频建议：be explicit

不要指望模型自动知道你默认想要什么。

该写清楚的都要写清楚：

- 用什么语言
- 用什么技术栈
- 输出多长
- 是否需要分点
- 是否允许调用某些库

越关键的约束，越不要省。

# role prompting 为什么常见

role prompting 不是为了“好玩”， 而是为了让 system prompt 更稳定地限定回答风格。

例如：

- 你是一位严谨的算法老师
- 你是一位通信系统工程师
- 你是一位代码审查助手

这样做的作用是：

- 让模型靠近某种语气、视角和输出习惯

# 但 role prompting 也不要神化

它能帮助模型进入某种模式， 但不能替代：

- 真实上下文
- 任务边界
- 输出格式
- 评测标准

所以 role prompt 是加成， 不是全部。

# decomposition 为什么很关键

如果任务很复杂， 不要把它当成一句话交给模型。

更稳的做法通常是：

- 先拆子问题
- 再一步一步做

比如先：

- 提取信息
- 再分析
- 再生成结论

# 一个总判断

Prompt engineering 的本质不是：

- 学一堆魔法短语

而是：

- 把任务定义清楚
- 把上下文组织清楚
- 把推理路径安排清楚
- 把外部能力接进来

# 也要讲清楚 Prompt 的边界

Prompt 很强， 但并不能替代：

- 更好的数据
- 微调
- 对齐
- 工具系统
- 评测

当模型能力或行为本身不够时， 只靠改 prompt 往往会很快碰到天花板。

# 所以课程后面为什么还要讲 3-1 和 3-2

因为：

- 3-0 解决的是“模型怎么被调出来”
- 3-1 解决的是“模型怎么被定制成任务能力”
- 3-2 解决的是“模型怎么被进一步对齐和优化行为”

Prompt 是入口， 但不是全部。

# 这节课你该带走什么

- Prompt 是预训练模型的运行时接口
- zero-shot、few-shot、CoT 解决的是不同层次的问题
- tool use、RAG、reflexion 说明 prompt 已经走向系统设计
- 好 prompt 的关键不是花哨，而是清楚、结构化、可检查

# 现在就可以开始做的事

- 用一个简单任务先试 zero-shot
- 再补 few-shot 例子
- 再试一次 CoT
- 如果还不稳，再考虑工具、检索或多轮修正

# Prompt 方法

## 先把任务说清楚，再谈模型表现