

AI 算法进阶 (Advanced AI System)

@Tyler

8. 深度强化学习进阶与多范式融合趋势



目录

- **模块一：LLM 与监督微调（连接主义）**
- 模块二：LLM 与专家系统（符号主义）
- 模块三：LLM 与环境交互（行为主义）

LLM 的原点：一切始于预训练

- 回顾 GPT 系列的预训练过程，我们把人类有史以来几乎所有的公开文本（书籍、维基百科、网页、代码等）都交给一个“初生”的神经网络。我们不给它任何特定的指令，只给它一个极其简单的任务：**预测下一个词**。
- 这就是CLM任务的核心。模型会看到类似这样的句子片段：
- “天空中最亮的星星是...”
- “The quick brown fox jumps over the...”
- `import numpy as...`
- 它的唯一目标，就是以最高的概率，准确预测出紧跟在后面的那个词元（Token）。对于上面的例子，理想的预测分别是“天狼星”、“lazy”和“np”。

LLM 的原点：一切始于预训练

- 为了完成这个看似简单的任务，模型被“逼”着去学习：
 - **语言规则**：主谓宾结构、时态、语法搭配。
 - **事实知识**：天狼星是最亮的恒星，Numpy通常被简称为np。
 - **上下文逻辑**：理解“fox”和“jumps over”之后，下一个词很可能是一个形容词来修饰“dog”。
 - **抽象推理**：在更复杂的文本中，它需要理解因果、比喻、情感等深层联系，才能做出正确的预测。
-
- 通过在数万亿个词元上日以继夜地进行这种“猜词游戏”，模型内部的参数（高达数千亿个）被不断调整和优化。这个过程，就是 **为LLM注入通用智慧**。预训练完成后，模型就从一个随机的神经网络，变成了一个**蕴含了海量人类知识和推理模式的“智慧基座”**，为后续的一切能力（如问答、写作、翻译）打下了坚实的基础。

LLM 的原点：一切始于预训练

- **自回归生成内容的全过程：一步一世界**
- 现在，让我们进入核心环节：当用户输入一段提示（Prompt）后，一个经过预训练的GPT模型是如何逐字生成流畅、相关的回答的。这个过程被称为 **自回归（Autoregressive）**，因为它每一步的输出，都会成为下一步的输入，就像一个不断回望过去、迈向未来的作者。
- 假设我们给GPT一个前缀（Prompt）：“今天天气”
- **第 1 轮推理：生成第一个新词**
- **输入：** GPT接收到初始文本“今天天气”。
- **推理：** 它执行一次完整的“思考”（也就是一次完整的模型推理），分析这个上下文，然后预测出下一个最可能的字词。
 - 模型内部可能会计算出：“晴”的概率是40%，“不”的概率是30%，“真”的概率是20%...
- **输出：** 假设模型选择了概率最高的“晴”。
- 至此，第一轮推理结束。我们只得到了一个字。

LLM 的原点：一切始于预训练

第 2 轮推理：将新词加入，再次推理

- **准备新输入：** 刚才生成的“晴”被追加到原来的文本后面，形成一个全新的前缀：“**今天天气晴**”。
- **推理：** GPT现在完全忘记了刚才的过程，它面对的是一个全新的任务。它接收这个新的、更长的输入“今天天气晴”，再次执行一次完整的“思考”，来预测下一个最可能的字词。
 - 基于新的上下文，它可能计算出：“朗”的概率是50%，“，”的概率是30%...
- **输出：** 模型选择了“**朗**”。
- 第二轮推理结束。

第 3 轮推理：循环往复

- **准备新输入：** 再次追加，形成更新的前缀：“**今天天气晴朗**”。
- **推理：** GPT对这个最新的输入进行第三次独立的、完整的推理，预测出下一个最可能的符号。
- **输出：** 这次可能输出了“，”。

LLM 的原点：一切始于预训练

总结：一次只走一步的“多次推理”过程

- **不是一次成型**：GPT不是看完“今天天气”后，就在“脑子”里构思好了“晴朗，万里无云”这句话，然后一次性吐出来。
- **而是一步一推**：它严格地执行“**推理 → 追加 → 再推理**”的循环。每生成一个字（或词），都必须将这个新字词作为已知信息，再重新进行一次完整的推理来决定下一个字词。
- 所以，生成一个包含10个词的句子，背后是**10次独立且连续的模型推理过程**。
- 每一次推理，上下文都会比上一次更丰富，从而保证了生成内容的连贯性和逻辑性。这正是其“自回归”的本质。
- 所以，我们看到的 **打字机** 效果，既是其“一步一推”核心算法的忠实反映。

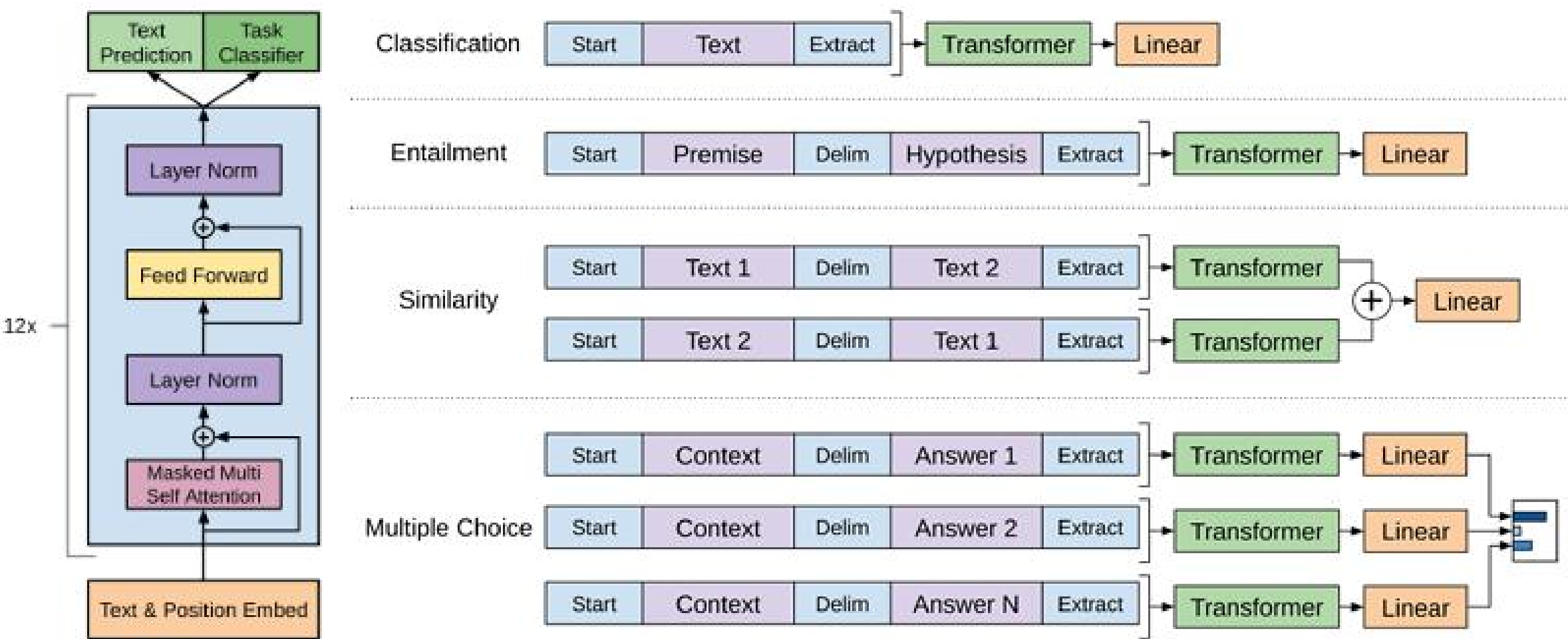
如何让LLM适配特定任务？ 监督微调的三种路径

- LLM 的 **指令跟随** 能力是预训练阶段与生俱来的吗？
- No! LLM 先天只有 Next Token Prediction 的能力，**指令跟随能力是后训练阶段设计出来的！**

分类头微调（传统迁移学习）	少样本提示 (Few-Shot Prompting)	零样本提示 (Zero-Shot Prompting)
服务角色：工程师 (The Engineer)	服务角色：工程师 (The Engineer)	服务角色：用户 (The User)
核心动作：	核心动作：	核心动作：
“改造”：为模型加装一个任务头，通过专门数据进行训练，将其打造成特定领域的判断专家。	“举例”：在提示中提供一两个完整范例，让模型快速“领悟”任务模式，成为一个灵活解题者。	“对话”：直接下达指令，通过巧妙的提示词（Prompt）引导模型利用其已有知识解决问题。
核心目标：	核心目标：	核心目标：
目标：追求单一任务的极致性能。 权衡：成本高，泛化能力弱。	目标：通过少量示例 精确引导 ，提升即时任务的准确率。 权衡：需要设计有效范例。	目标： 即时、零成本 地解决多样化问题。 权衡：对模型自身能力依赖强。
经典案例：	经典案例：	经典案例：
GPT-1 用于文本分类	GPT-3, FLAN	GPT-3.5 / GPT-4 的开箱即用

分类头微调（传统迁移学习）

- **过程：高效训练 (Efficient Training)**
 - “冻住”模型主体，仅“激活”并训练头部的参数，极大节省计算资源。
- **最佳应用 (Best For):**
- 需要明确“是/否”或“属于哪一类”的**判别类任务**。
- 这是一种**高效、快速**的适配手段，它将LLM强大的通用语义能力，像搭乐高一样精准**迁移**到特定的判断任务上。



少样本提示 (Few-Shot Prompting)

- **Language Models are Few-Shot Learners (GPT-3)**

- 将模型微调与推理的三大要素，系统性地“编码”到语言模型原用于生成的**输入前缀 (Input Prefix)** 中。我们可以将这个输入前缀的构建过程，理解为一个清晰的公式：

- **输入前缀 (Prompt) = 任务描述 + 样本 + 具体输入**

将以下英文短语翻译成中文，并保持简洁 (任务描述)

英文: *sea otter* 中文: 海獭 (少样本示例)

英文: *A leopard can't change its spots.* (用户具体输入)

中文:

- 当这个结构化的前缀被注入模型后，模型会基于前面的规则和范例，续写出期望的答案：“江山易改，本性难移。”



这一思想如何落地？

- 其本质，是一场针对模型“思维模式”的系统性重塑。我们不再满足于它“接话”的能力，而是要教会它“办事”的能力。
- 我们将海量的、各式各样的公开数据集，通过自动化脚本，系统性地转化为统一的 [指令] -> [期望回答] 格式的训练样本。这就像是为模型准备了成千上万个不同岗位的“岗前培训”材料。
- 经过这种特训，模型的当它看到输入文本时，会下意识地做出判断：
- **输入是普通文本（自由的上文描述） → 启动“文本补全”模式**
 - 任务：我该如何最自然地接续这段话？
 - 行为：延续上下文，进行概率最高的续写。
- **输入是结构化指令（任务描述 + 样本 + 具体输入） → 启动“指令跟随”模式**
 - 任务：这条指令的目标是什么？我该如何完成它？
 - 行为：理解意图，调动知识，生成符合要求的答案。

这一思想如何落地？

- **最终目标 (Ultimate Goal): 内化指令思维**
- 通过海量结构化样本的“喂养”，让模型深度适应并**内化**这种指令跟随的思维模式。其最终目标，是让“听从指令”成为模型的第二天性，而不只是机械地模仿。
- **成功基石 (Cornerstone of Success):**
- **高质量、多样化**的指令数据，是模型能否真正学会泛化、触类旁通的唯一食粮。数据的广度决定了模型能力的广度。

实例：将公开数据集转化为指令格式

我们以一个经典的情感分析数据集（如SST-2）为例，它的任务是判断一句话是正面的还是负面的。

1. 原始数据格式 (Original Data Format)

- 通常，原始数据集的格式非常简单，类似于一个表格或JSON文件，每一行包含一个句子和它的标签。
- **标签 1 代表“正面” (Positive)**
- **标签 0 代表“负面” (Negative)**
- **原始数据点长这样：**
 - { "sentence": "a masterpiece, a profoundly moving film.", "label": 1 }
 - { "sentence": "the plot is predictable and the acting is dull.", "label": 0 }
 - ...

实例：将公开数据集转化为指令格式

2. 设计指令模板 (Design the Instruction Template)

- 我们的目标是把这个“计算机可读”的数据，变成“LLM可读”的自然语言指令。我们可以设计多个模板来描述数据集的任务，例如：
- "对以下句子的情感进行分类，选项为正面或负面。句子：{sentence}"
- "判断下面这句话表达的是积极情绪还是消极情绪？句子：{sentence}"
- "下面是一条电影评论：'{sentence}'。请问这条评论是正面的还是负面的？"
- ...等等

3. 映射标签到文本 (Map Labels to Text)

- 模型输出的是文本，而不是1或0。所以我们需要一个明确的映射关系。
- 1 -> "正面"
- 0 -> "负面"

实例：将公开数据集转化为指令格式

4. 组装成最终“少样本”格式的训练样本 (Few-Shot Style)

- 假设某公开的 **情感分类** 数据集中有：
- { "sentence": "the plot is predictable and the acting is dull.", "label": 0 }
- { "sentence": "a masterpiece, a profoundly moving film.", "label": 1 }
- 选取**模板2**作为我们的**[任务描述]**；选取数据集中的**另一条数据**（比如上面第二条）作为**[样本]**；选取**第一条数据的 X** 作为我们真正的**[具体输入]**，Y 作为 **[真实输出]**。组装后的结果：
 - **判断下面这句话表达的是积极情绪还是消极情绪？**
 - **例子：**
 - **句子：** "the plot is predictable and the acting is dull." **情绪：** 负面
 - **现在请判断下面这个句子：**
 - **句子：** "a masterpiece, a profoundly moving film." **情绪：** 正面
- 通过对整个原始数据集进行这样的**批量化、自动化、模板化**的转换，我们就能构建起一个包含成千上万条高质量、多样化指令的微调数据集。模型在学习了这些数据后，就“内化”了遵循**[任务-样本-输入]**结构的能力。

零样本提示 (Zero-Shot Prompting)

- **交互范式的转变：从“模板编程”到“自由提问”**
- 这一转变的基石，源于OpenAI在 **InstructGPT** 等早期工作中，对**监督微调 (SFT)** 阶段的精心设计。其核心不再是依赖机器生成的、格式化的指令，而是**回归到真实、多样、高质量的人类对话本身**。
- **OpenAI 的 SFT 蓝图：三步构建“高质量对话”数据集**
- 模型之所以能听懂“人话”，是因为它首先学习了大量高质量的“人话”范例。这个过程可以分解为三步：

零样本提示 (Zero-Shot Prompting)

第一步：采集真实的、多样化的用户指令 (Prompts)

- **来源 (Source):** 并非凭空创造或使用现有学术数据集，而是直接从早期 **OpenAI API** 的真实用户使用场景中进行采样。
- **价值 (Value):** 这保证了指令的**多样性**和**实用性**，覆盖了从“写一首关于秋天的诗”这样的开放式生成，到“解释这段代码的功能”这类封闭式问答的广泛需求。

零样本提示 (Zero-Shot Prompting)

第二步：雇佣人类专家，撰写高质量的示范回答 (Demonstrations)

- **执行者 (Executors):** 雇佣并培训了一批专业的标注人员 (Labelers) 。
- **核心任务 (Core Task):** 要求标注人员针对采集到的用户指令，**像一位尽职尽责的助手一样，精心撰写**出他们认为最理想的、最有帮助的回答。
- **关键原则 (Key Principle):** 追求的是**意图对齐 (Intent Alignment)**，而不仅仅是事实正确。回答需要真正理解并解决用户的潜在问题，而不仅仅是给出字面答案。

零样本提示 (Zero-Shot Prompting)

第三步：用“人工范例”进行监督微调

- **数据格式 (Data Format):** 将上一步收集到的 {用户真实指令, 标注员示范回答} 数据对, 作为“黄金标准”的训练数据。
- **训练过程 (Training Process):** 在强大的预训练模型 (如 GPT-3) 的基础上, 用这些高质量、纯人工的对话数据进行微调。模型的目标, 就是学习模仿这些专家级的回答。

零样本提示 (Zero-Shot Prompting)

最终成果与机制解析 (The Result & Mechanism):

- 通过这个SFT过程，模型系统性地学习了如何**理解并遵循“非结构化”的自然语言指令**。它的能力不再局限于识别 [任务-样本-输入] 这样的刻板模板，而是能够**直接从用户的日常提问中推断真实意图**。
- 这正是后来零样本 (Zero-Shot) 强大能力的基石，也是让大模型能够真正走向公众、提供实用价值的关键一步。

零样本提示 (Zero-Shot Prompting)

- **极致的自由**
- 这是LLM强大通用智能最直接、最令人惊艳的体现。用户无需任何训练，仅凭自然语言就能与超强AI协作，这在过去是科幻小说里的情节。
- **易用性**
- 它将使用门槛降至最低，让非技术背景的普通用户也能“开箱即用”，是AI技术能够大规模普及的最大功臣。

零样本提示 (Zero-Shot Prompting)

- 魔法背后的代价 (The Price Behind the Magic)
- 对“出身”的严重依赖 (Heavy Reliance on "Origin")
 - 其效果完全取决于两大“先天”因素：
 - ① 模型本身的规模和质量，以及 ② 其背后指令微调阶段的数据优劣。
 - 基础不好，魔法就不灵。
- 在“精准”上的妥协 (Compromise on "Precision")
 - 当面临**复杂、新颖或需要高度精确**的任务时，由于缺乏具体的范例引导，其表现出的稳定性和准确性，通常**明确地不如少样本提示**。

为什么 Few-Shot 如此重要？

- **零样本（Zero-Shot）** 是将LLM通用智能“开箱即用”的主流方式。它本质上是一种聪明的“产品策略”：**通过牺牲一部分任务的极致精准性，换来了无与伦比的易用性和交互的自然流畅感。**
- **少样本提示（Few-Shot Prompting）**：因此，工程师的“锚点”是**Few-Shot**，它是确保输出**稳定性与迁移效果**的基石。它通过在提示词中置入清晰的范例，为模型天马行空的“即兴发挥”划定了航道。

为什么 Few-Shot 如此重要？

- 理论上，少样本（Few-Shot）提示的效果，可以通过优化两大支柱来极致地提升：
- **范例的质量 (Quality of Examples):** 注入的范例与用户的当前问题**高度相关**，且能清晰地展示所需的逻辑和格式。
- **范例的数量 (Quantity of Examples):** 对于复杂任务，提供**更丰富、更多样**的范例，能让模型更准确地捕捉任务的本质。
- **理想情况是：** 为每一个用户问题，都从一个巨大的“专家范例库”中，动态挑选出**最相关、最丰富**的多个范例来构建完美的提示词。

为什么 Few-Shot 如此重要？

- 最终的启示：In-Context Learning (上下文学习)
- 无论是 Few-Shot 手动注入的几个范例，还是 RAG 自动检索的几段知识，它们对于大语言模型来说，身份是等价的——都只是“上下文 (Context)”的一部分。
- LLM 并不知道这是“范例”还是“知识”，它只知道基于当前窗口内收到的所有文本信息，来学习和模仿，然后生成最合理的下一个 Token。
- 这种 **不通过更新模型权重、仅通过构造提示词来引导模型学习并完成任务的强大能力**，正是“上下文学习 (In-Context Learning)”。
- 这揭示了一个深刻的范式转变：我们与AI协作的重点，从“训练/微调模型本身”，越来越多地转向了“如何为模型提供一个高质量、高信息密度的实时上下文”。

[任务描述]

请根据下面提供的上下文信息，回答问题。

[自动检索的动态知识]

上下文：检索系统发现，关于“天狼星计划”的最新季度报

[用户的具体问题]

问题：“天狼星计划”目前的风险评级和负责人是谁？

[任务描述]

将以下英文短语翻译成中文，并保持简洁。

[手动编写的静态范例]

英文：sea otter

中文：海獭

[用户的具体问题]

英文：A leopard can't change its spots.

中文：

上下文工程 (Context Engineering)

- 因此，相比于“提示词工程 (Prompt Engineering)”，“上下文工程 (Context Engineering)”这个术语富有更多内涵。
- 我们思考的不再是孤立的提示语模板，而是如何将模型的“上下文窗口 (Context Window)”看作一个可编程的、结构化的逻辑空间。
- 一个精心设计的“上下文”不再是一段简单的文本，而是一个由多个逻辑模块动态组合而成的“超级提示词”。

```
[ ----- 上下文窗口的起点 ----- ]

[ 模块 1: 系统级指令 (System-level Instructions) ]
|-> 角色扮演 (Persona): "你是一位资深的投资分析师。"
|-> 输出格式 (Format): "你的回答必须是JSON格式。"
|-> 核心规则 (Rules): "不要编造数据，如果信息未知，请明确指出。"

[ 模块 2: 工具与函数定义 (Tools & Functions) ]
|-> API定义: [Tool: get_latest_stock_price(ticker)]
|-> 知识库检索工具: [Tool: search_financial_news(query)]
|-> (这为模型提供了与外部世界交互的能力，是Agent能力的基础)

[ 模块 3: 静态范例 (Few-Shot Examples) ]
|-> 任务范例1: Q: "分析一下AAPL" -> A: { "ticker": "AAPL", "summary": "..."}
|-> 任务范例2: Q: "对比TSLA和F" -> A: { "comparison": { ... } }
|-> (用于为模型展示期望的推理路径和输出风格)

[ 模块 4: 动态知识注入 (Dynamic Knowledge - RAG) ]
|-> (根据用户问题实时从数据库中检索)
|-> [知识片段1: "最新的财报显示，MSFT的云业务增长强劲..."]
|-> [知识片段2: "分析师指出，AI芯片的需求推动了NVDA的股价..."]

[ 模块 5: 对话历史 (Conversation History) ]
|-> User: "帮我看看科技股最近的表现。"
|-> Assistant: "当然，您对哪些具体的公司感兴趣？"

[ 模块 6: 用户当下的问题 (User's Current Query) ]
|-> "根据以上信息，帮我分析一下微软和英伟达最近的情况。"

[ ----- 上下文窗口的终点，模型开始生成 ----- ]
```

上下文的逻辑分区

- **人设模块 (Persona) -> 逻辑分区 1**
 - 定义：你是谁？你的角色、语气和风格是什么？
 - 示例：“你是一位资深的、严谨的金融分析师。”
- **任务模块 (Task) -> 逻辑分区 2**
 - 定义：你需要完成什么具体目标？规则和步骤是什么？
 - 示例：“请根据以下财报，撰写一份不超过200字的摘要。”
- **知识模块 (Knowledge) -> 逻辑分区 3**
 - 定义：你需要哪些外部事实信息来完成任务？(RAG 在此发挥作用)
 - 示例：“上下文：[此处由RAG动态检索并注入最新的财报原文]”

上下文的逻辑分区

- **范例模块 (Examples) -> 逻辑分区 4**
 - 定义： 你需要参考哪些格式或逻辑的范例？(Few-Shot 在此发挥作用)
 - 示例：“例子：[此处注入一两个高质量的摘要范例]”
- **工具模块 (Tools) -> 逻辑分区 5**
 - 定义： 你可以调用哪些外部工具 (API) 来获取额外能力？
 - 示例：“你可以使用 [get_stock_price] 工具来查询实时股价。”
- **行动记录 (Trajectory) -> 逻辑分区 6**
- ...

工作本质的再定义：从“工匠”到“架构师”

- **我们身份的演进：** 我们的工作，已不再是那个反复打磨字句的“提示词工匠”。
- 我们正在成为，那个设计信息流、搭建系统能力的“上下文架构师”。
- **新角色的核心任务 (The Core Task of the New Role):**
- 我们的日常，将围绕“**如何智能地获取、筛选、并动态组织**”前面提到的那些上下文模块（人设、任务、知识、工具...）来设计和构建智能系统。

工作本质的再定义：从“工匠”到“架构师”

- 这时，我们遇到了一个根本性的问题：大语言模型本身，作为连接主义的产物，擅长的是**理解**和**生成**，它无法**主动、可靠地从外部世界获取**这些结构化的上下文。
- 它能读懂我们递给它的“书”（上下文），但它自己不会“去图书馆找书”。
- 这，必须依赖符号主义（Symbolism）的经典利器：**专家系统（Expert Systems）**。
- 在现代的LLM应用架构中，专家系统扮演着“**上下文调度中心**”或“**首席信息官**”的角色。它的职责就是：
- **连接**外部世界（数据库、API、知识库）。
- 根据预设的**规则和逻辑**，精准地**获取、筛选并结构化**所需的上下文。
- 最终，将这个完美的“上下文包裹”，递交给LLM进行处理。

目录

- 模块一：LLM 与监督微调（连接主义）
- **模块二：LLM 与专家系统（符号主义）**
- 模块三：LLM 与智能代理（行为主义）

一个根本性的挑战

1. LLM 的核心身份：被动的“上下文消费者”

- **角色定位：** 连接主义的产物，精通“理解”而非“执行”。

2. 能力边界：

- **擅长：** 理解、模仿和续写**给定的**上下文信息。
- **缺失：**
- **主动性：** 无法自行从外部世界（数据库, API）获取信息。
- **可靠性：** 无法确保严格遵循复杂的指令结构。

3. 我们面临的核心问题：

- 那个由（人设、任务、知识、工具...）模块组成的“超级提示词”， **由谁来负责构建与调度？**

专家系统：LLM 应用架构的“调度中枢”

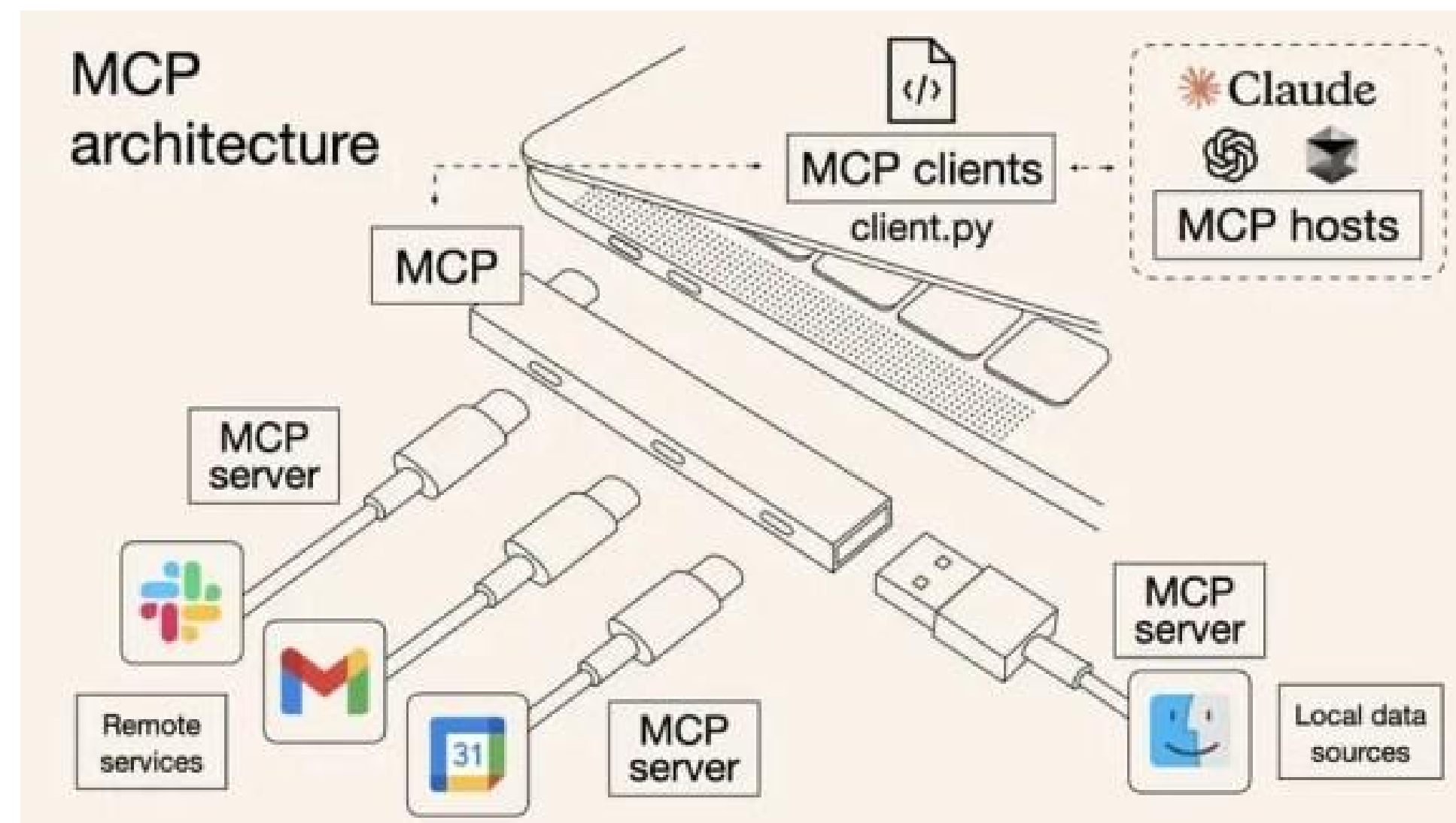
- **本质：** LLM与外部资源的信息交换
- **核心问题：** 如何标准化LLM使用的外部资源，并**统一交互策略**？
- **解决方案：** 专家系统通过规则和协议，将信息交换过程结构化、规范化。

- **破局之道：** AI 世界的“USB-C”
- 为解决上述问题，一个旨在**用符号主义为连接主义建立秩序**的开放标准 **MCP** 应运而生。

规模化应用的瓶颈：各自为战的“接口”

Anthropic 的“模型上下文协议” (Model Context Protocol, MCP)

- **定位:** 一个开放的、可标准化的双向连接协议。
- **使命:** 统一 LLM 应用与外部世界的交互方式，正如 USB-C 统一了所有电子设备的物理接口。



思考一下，良好的上下文工程需要哪些“要素”？

1. Prompt 提示词（逻辑分区）

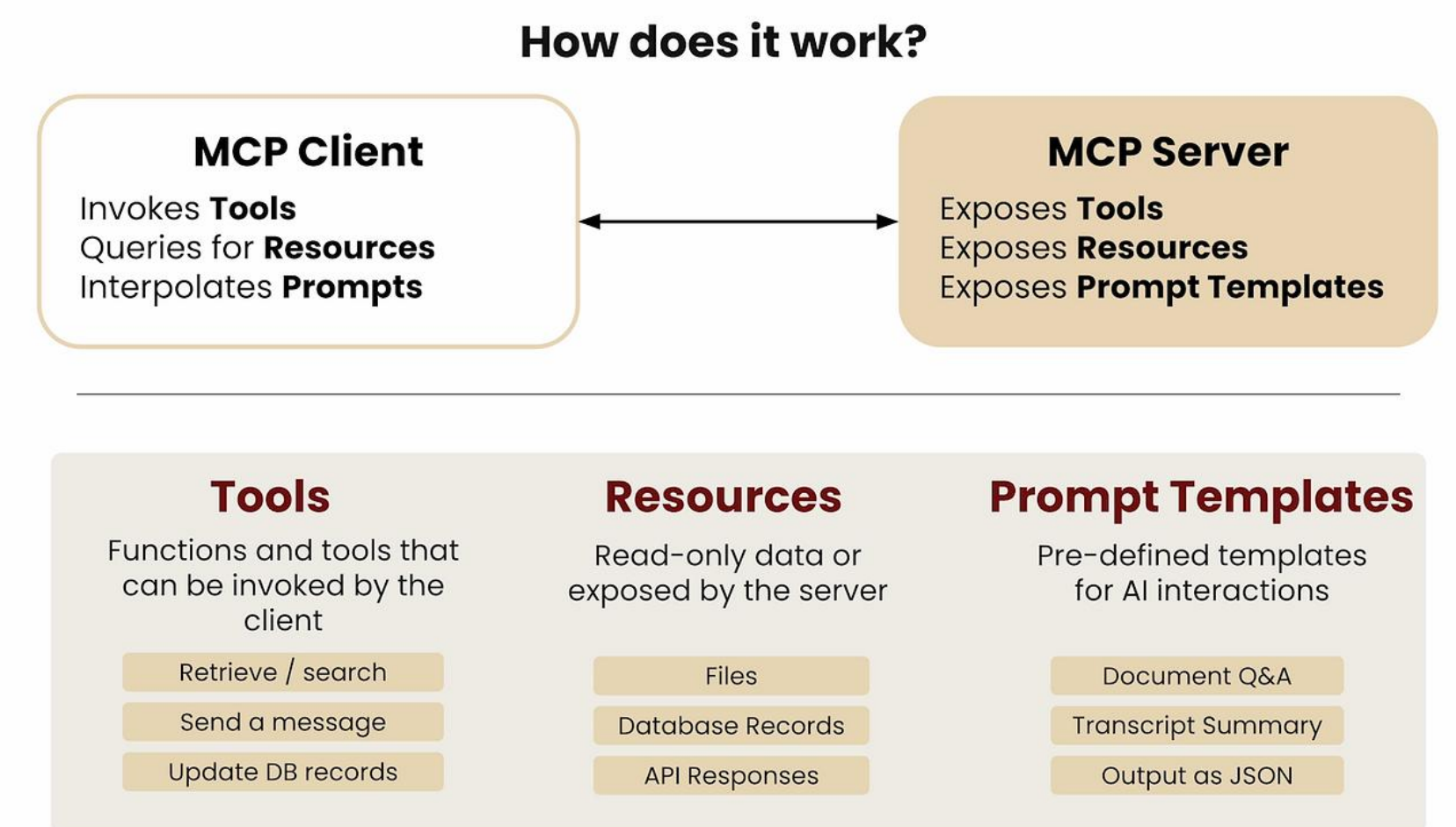
- **角色：** 思考的“框架”和“骨架”。（强调**结构性**）
- **功能：** 定义了任务的逻辑分区，如人设、任务指令、历史记录等。这是确保AI理解你意图的关键。

2. Tools 工具（行动空间）

- **角色：** AI与外部世界交互的“手脚”。（强调**交互性**）
- **功能：** 允许AI调用外部API或执行特定动作，例如发送邮件或获取实时数据。

3. Resources 资源（实时状态）

- **角色：** 挂载数据和文件的“神经”。（强调**同步性**）
- **功能：** 提供AI所需的外部数据，包括代码文件，数据库、文档等，确保AI基于最新、最全的信息进行判断。



Prompts 要素：上下文的“逻辑分区”

- Prompts 是MCP协议提供的一种**标准化方式**，提供一系列预设好的、结构化的“指令模板”。
- **核心交互**：如何“发现(List)”和“获取(Get)”一个Prompt
- **1. 发现 (prompts/list)**
- **目标**：客户端向服务器请求，获取所有可用的“快捷指令”列表。
- **交互流程 (简化版)**:
 - **客户端发送请求**: method: "prompts/list"
 - **服务器返回列表**:

```
{
  "prompts": [
    {
      "name": "code_review",
      "title": "请求代码审查",
      "description": "让AI分析代码质量...",
      "arguments": [ // 这个指令需要哪些参数
        { "name": "code", "required": true }
      ]
    }
  ]
}
```

Prompts 要素：上下文的“逻辑分区”

- 2. 获取并执行 (prompts/get)
- **目标:** 用户选择了 code_review 指令并提供了参数（代码），客户端据此向服务器请求构建好的、最终给LLM的上下文。
- **客户端发送请求（已填充参数）:**

```
{  
  "method": "prompts/get",  
  "params": {  
    "name": "code_review",  
    "arguments": { "code": "def hello()..." }  
  }  
}
```

Prompts 要素：上下文的“逻辑分区”

1. 核心价值：标准化的“用户指令层”

- Prompts 将应用的能力，以一种用户可控、可发现、标准化的“逻辑分区”形式暴露出来，极大地降低了复杂功能的使用门槛。

2. 技术实现：简洁、可靠的API

- 基于成熟的 JSON-RPC 2.0，提供了清晰的 list 和 get 等核心方法，并支持分页、实时更新通知等现代化特性。

3. 革命性特性：富内容与资源嵌入

- Prompts 真正成为了连接LLM与外部复杂数据源的桥梁，是实现“上下文工程”的核心基石之一。

- 服务器返回最终上下文（右图）：

```
{
  "messages": [ // 这是最终要发给LLM的内容
    {
      "role": "user",
      "content": {
        "type": "text",
        "text": "Please review this Python code..."
      }
    }
  ]
}
```

Tools 要素：赋予 AI “行动”的能力

- Tools 是MCP协议提供的一种**标准化方式**，让应用（服务器）能够向 AI 暴露一系列可供其**自主调用**的外部功能。
- AI在理解用户任务后，会自行打开工具箱，分析哪个工具最适用（如“天气查询器”、“计算器”、“数据库接口”），然后拿起并使用它。
- **核心价值：**让 AI 从一个被动的“知识问答者”，进化为一个主动的“世界交互者”，能够为了解决问题而采取实际行动。

Tools 要素：赋予 AI “行动”的能力

赋予AI“行动”的能力，必须以绝对的**安全**为前提。MCP的 Tools 设计，始终将人类置于最终控制环路中。

1. 用户提出需求

- “我应该带伞去东京吗？”

2. LLM 推理并决策

- “要回答这个问题，我需要知道东京现在是否在下雨。我应该使用 `get_weather` 工具。”

3. **系统请求人类批准 (Human in the Loop)**

- “AI 正在尝试执行以下操作：调用工具：`get_weather(location: 'Tokyo')`” [允许] [拒绝]

4. 用户授权后，系统执行工具

- 用户点击 [允许] 后，工具被安全调用。

Tools 要素：赋予 AI “行动”的能力

- AI如何理解工具？靠标准化的“使用说明书”
- AI之所以能够准确地知道如何使用一个工具，是因为每个工具都提供了一份机器可读的、名为 `inputSchema` 的“说明书”。
- 工具说明卡：天气查询器
- 名称 (name): `get_weather`
- 标题 (title): 天气信息提供者
- 描述 (description): 获取一个地点的当前天气信息。
- 参数 (inputSchema):
 - location:
 - 类型: `string` (文本)
 - 描述: 城市名或邮政编码
 - 是否必需: 是

Tools 要素：赋予 AI “行动”的能力

- **超越文本：更可靠、更智能的“结构化输出”**
- 传统工具可能只返回一段简单的文本，这对于程序后续处理非常不便。MCP Tools 的一大优势是支持返回**结构化的JSON数据**。
- **传统的非结构化输出（难以处理）**
- "东京当前天气：温度22°C，天气状况：多云"（*程序需要用复杂的文本解析来提取温度值*）
- **MCP的结构化输出（清晰、可靠）**
- 工具甚至可以预先定义好返回结果的格式（**outputSchema**），让客户端在收到数据后能**自动验证其准确性和完整性**，极大提升了系统的鲁棒性。

```
{
  "structuredContent": {
    "temperature": 22,
    "conditions": "Cloudy",
    "humidity": 65
  }
}
```

Tools 要素：赋予 AI “行动”的能力

1. 核心价值：赋予AI“行动能力”

- Tools 让AI不再局限于已有的知识，而是能够通过与外部系统交互来获取新信息、执行新任务，成为一个真正的**主动代理 (Active Agent)**。

2. 实现方式：“**契约式**”交互

- 通过标准化的 inputSchema (输入契约) 和 outputSchema (输出契约)，确保了AI与工具之间沟通的**可靠性**和**准确性**。

3. 设计哲学：安全机制

- Human in the Loop** 的监督机制，以及对输入输出的严格验证，为AI在处理敏感数据和执行关键操作时提供了必要的安全保障。

Resources 要素：为 AI 挂载一个“文件系统”



- Resources 是 MCP 协议提供的一种**标准化方式**，让应用（服务器）能够向AI客户端共享一系列**上下文数据**。这可以是任何东西：项目文件、数据库的表结构、实时日志、用户文档等。
- **它的设计哲学：应用驱动 (Application-Driven)** 与用户选择的 Prompts 和模型调用的 Tools 不同，Resources 的设计是由**应用程序**来主导如何以及何时将其提供给 AI。例如，应用可以自动将当前打开的文件作为 Resource 注入上下文。
- **常见的交互形态：上下文资源选择器**（一个模拟IDE的UI图，左侧是文件树，用户可以勾选文件，将其加入到右侧的AI对话上下文中）
- **核心价值：为AI的每一次思考和推理，提供坚实的、可信的、最新的“事实依据”，从根本上解决“一本正经地胡说八道”的问题。**

Resources 要素：为 AI 挂载一个“文件系统”



超越静态读取：通过“订阅”实现世界与AI的实时同步

- Resources 最具革命性的能力，不是简单的“读取”文件，而是可以“订阅”它。

1. 传统的“读取”模式 (PULL)

- AI:** "我想读一下 main.rs 文件的内容。" **系统:** (执行 resources/read) 返回文件内容。 (如果文件在这之后被修改, AI对此一无所知)

2. MCP的“订阅”模式 (PUSH)

- AI:** "请**订阅** main.rs 文件, 它有任何变化就告诉我。" (执行 resources/subscribe) ...**几分钟后, 开发者修改并保存了 main.rs...** **系统:** (服务器主动发送 notifications/resources/updated 通知) **AI:** "我感知到main.rs已更新, 我将基于最新版本重新进行分析。"
- 重大意义: 这让AI从一个需要不断“轮询”世界的“提问者”, 进化为一个能够实时“感知”世界变化的“观察者”。

Resources 要素：为 AI 挂载一个“文件系统”



- **智能上下文管理：用“注解(Annotations)”自动筛选信息**
- 当一个项目有成百上千个Resources（文件），我们不可能手动将它们全部喂给AI。**Annotations**（注解）就是解决这个问题的“智能过滤器”。
- **信息卡：项目说明文档**
- **URI:** file:///project/README.md
- **MIME 类型:** text/markdown
- **注解 (Annotations):**
 - **audience (受众):** ["user", "assistant"] (这份文档对用户和AI都有用)
 - **priority (优先级):** 0.8 (比较重要)
 - **lastModified (最后修改):** "2025-01-12T...Z"
- **“智能上下文填充”是如何工作的？** 应用程序可以设定一条规则，例如：“自动将所有priority大于0.7，且audience包含assistant的资源，注入到AI的上下文中。”
- **重大意义：** Annotations 让“上下文工程”从繁琐的手动操作，进化为可自动执行的、基于规则的智能流程，极大地提升了效率和准确性。

Resources 要素：为 AI 挂载一个“文件系统” 极客时间

Resources 要素总结

1. 核心价值：AI的“真理之源”

- Resources 为AI提供了可靠、可验证、可实时更新的外部知识，是确保其输出**事实性和相关性**的生命线。

2. 实现方式：标准化的数据接口

- 基于通用的 **URI 方案** (file://, https://, git:// 等) 来唯一定位每一个数据资源，具有极佳的兼容性和扩展性。

3. 革命性特性：动态与智能

- 订阅 (subscribe)** 机制带来了**实时性**，让AI能够感知世界的动态变化。
- 注解 (annotations)** 机制带来了**智能性**，让海量上下文的管理和筛选能够自动化。

端到端实例：从用户输入到精准响应的完整旅程

极客时间  训练营

原生的、未经处理的用户请求：“分析一下谷歌最近的财报”

第一步：系统通过 MCP 协议，进行模板的“发现与匹配”

- **意图识别：**系统首先分析用户输入的意图。
- **模板发现 (Discovery):**
 - 系统通过MCP协议，向“Prompt模板库”发送一个 prompts/list 请求，获取所有可用的模板列表及其描述。

- **返回列表 (简化版):**

```
[  
  { "name": "simple_chat", "description": "通用聊天" },  
  { "name": "financial_analysis_v3", "description": "对公司进行深入的财务分析..." },  
  { "name": "code_generation", "description": "生成代码" }  
]
```

端到端实例：从用户输入到精准响应的完整旅程

极客时间 

- **智能匹配 (Matching):**
 - 系统将用户意图 `financial_analysis` 与返回的模板列表进行比对。
 - **结果:** 发现 `financial_analysis_v3` 是最佳匹配。
- **输出 (OUTPUT): 一个被选中的、带有“占位符”的上下文框架 (逻辑分区)**
 - [人设] 你是一位顶级的金融分析师，风格严谨、注重数据。
 - [用户] `{resource: identity_info}`
 - [历史] `{resource: conversation_history}`
 - [工具] 可用的工具列表: `{tools: tool_list}`
 - [任务] 根据以上信息，完成任务: `{user_intent}`
- 这种 "`{类型:名称}`" 的占位符语法，让模板自身就携带了填充规则，为后续的自动化处理奠定了坚实基础。

端到端实例：从用户输入到精准响应的完整旅程

极客时间 

第二步：分析依赖，生成资源需求清单

- 在这一步，系统的核心任务是读取已选中的上下文模板，并生成一份具体的、可执行的“**数据与工具采购清单**”。
- 系统会扫描模板中的每一个占位符。由于占位符采用了**显式类型**（如 resource: 或 tools:），系统能够精确识别其类型和具体名称。
- **输出 (OUTPUT): 一份具体、可执行的“需求清单”**
- **资源需求 (Resources)**
 - **identity_info**: 获取用户的身份和画像信息。
 - **conversation_history**: 获取当前会话的历史记录。
- **工具需求 (Tools)**
 - **tool_list**: 获取与用户意图（财报分析）相关的工具列表。

端到端实例：从用户输入到精准响应的完整旅程

极客时间  训练营

第三步：获取资源（由 MCP 客户端执行）

- 在这一步，作为“上下文编排器”的 **MCP 客户端**，将执行“需求清单”，向不同的 **MCP 服务器** 发起一系列标准化请求，采集所有需要的“原材料”。
- **向用户服务 (User Service)**
 - 客户端发送 resources/read 请求，以获取 identity_info 资源。
- **向会话服务 (Session Service)**
 - 客户端发送 resources/read 请求，以获取 conversation_history 资源。
- **向工具服务 (Tool Service)**
 - 客户端发送 tools/list 请求，以获取所有可用工具并进行筛选。
- **从初始输入**
 - 客户端处理用户的原始输入，提取 user_intent 的核心文本。
- 通过统一的MCP协议，客户端无需关心每个后端服务的具体实现。它只需通过标准化的方式“索要”数据，而各个服务器则负责“提供”数据。这实现了系统的高度解耦和模块化。

端到端实例：从用户输入到精准响应的完整旅程

极客时间 

第四步：构建上下文，生成“超级提示词”

在这一步，系统的核心任务是将第三步获取到的所有“原材料”，精准地“注入”到第一步选定的“模板”中，最终生成一个信息完整、可直接交付给 LLM 的“超级提示词”。

1. 模板框架：

[用户]

{resource: identity_info}

...

[任务]

...{user_intent}

2. 真实数据包：

- **identity_info:** { "id": 789, "risk_profile": "稳健型", ... }
- **conversation_history:** [...]
- **tool_list:** [{ "name": "get_stock_price", ... }]
- **user_intent:** "分析谷歌的最新财报"

端到端实例：从用户输入到精准响应的完整旅程

极客时间 

- 一个高度个性化、情境感极强的“超级提示词”
 - [人设]
 - 你是一位顶级的金融分析师，风格严谨、注重数据，给出审慎的投资建议
 - [用户]
 - 当前用户信息：客户ID 789, 风险偏好: 稳健型, 投资组合中持有100股GOOGL。
 - [历史]
 - 空白
 - [工具]
 - – get_stock_price(ticker: string): 获取最新股价的工具。
 - [任务]
 - 根据以上信息，完成任务：分析 谷歌 (Alphabet) 的最新财报。

上下文工程的“四步工作法”

第5步：执行上下文

- 调用 LLM 执行良好逻辑分区构建的上下文提示。
- **第一轮输出：** Action: **get_stock_price[google]**
- “专家系统”通过 MCP 协议，调用 get_stock_price 工具。
- 返回结果 (Observation): 股价为 \$185.50。

上下文工程的“四步工作法”

- 之后再次根据之前的 prompt 模板构成更新后的上下文：
- [人设]
 - 你是一位顶级的金融分析师，风格严谨、注重数据，给出审慎的投资建议
- [用户]
 - 当前用户信息：客户ID 789, 风险偏好: 稳健型, 投资组合中持有100股GOOGL。
- [历史]
 - **Action: get_stock_price[GOOGL] Observation: \$185.50**
- [工具]
 - – get_stock_price(ticker: string): 获取最新股价的工具。
- [任务]
 - 根据以上信息，完成任务：分析 谷歌 (Alphabet) 的最新财报。

上下文工程的“四步工作法”

第二轮输出：

- 好的，John。根据谷歌最新的财报，其核心广告业务保持稳健，云业务增长25%表现强劲，超出了市场预期。
- 考虑到：
- 当前股价为 **\$185.50**。
- 您是“**稳健型**”投资者。
- 您已持有**100股仓位**。
- 我的建议是“**持有并观望**”。财报利好，但考虑到您的风险偏好，不建议在当前价位立即增持。我们可以设置一个提醒，如果股价回调到\$175附近，再考虑增加头寸。

现代专家系统的核心职责

1. 连接 (Connect) | 世界的接口

职责： 打通所有外部信息源。

对象： 数据库、业务API、实时数据流、知识库。

2. 执行 (Execute) | 逻辑的引擎

职责： 运行确定性的、基于规则的任务。

内容： If...Then... 业务逻辑、任务拆解、信息检索、数据清洗。

3. 构建 (Construct) | 上下文的建筑师

职责： 将杂乱信息组装成标准化的“上下文包裹”。

产出： 结构清晰、格式统一、无歧义输入。

4. 移交 (Delegate) | 任务的调度员

职责： 将完美的“包裹”递交给 LLM。

目标： 让LLM在最理想的条件下，专注其最擅长的推理与生成。

专家系统：LLM 应用架构的“调度中枢”

[步骤1：输入] 用户请求 (一个模糊、开放式的需求)

[步骤2：专家系统处理 – 核心环节] (一个代表“专家系统”的中央处理单元，内部包含三个动作)

① **连接 (Connect)**: 调用API/数据库

② **执行 (Execute)**: 运行规则，拆解任务

③ **构建 (Construct)**: 组装成结构化上下文

[步骤3：LLM 处理] “上下文包裹” → [LLM 大脑] (LLM 在完美的条件下进行创造性工作)

[步骤4：输出] 最终答案

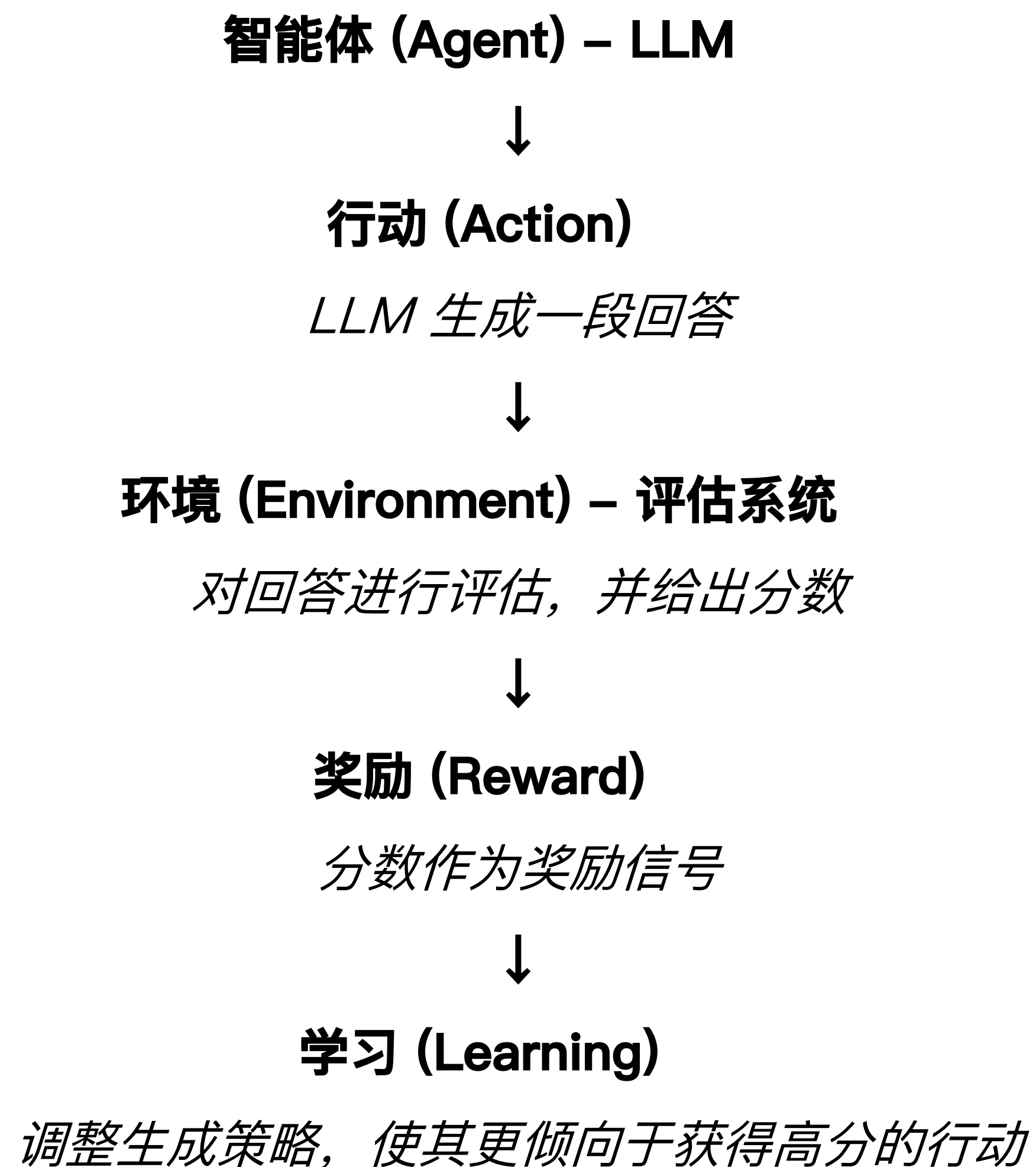
目录

- 模块一：LLM 与监督微调（连接主义）
- 模块二：LLM 与专家系统（符号主义）
- **模块三：LLM 与智能代理（行为主义）**

通过环境反馈进行策略优化

- **核心思想：让模型在“试错”中学会如何行动**
- 与直接告诉模型“正确答案”的监督学习不同，强化学习的核心是设定一个**目标（奖励）**，然后让模型通过与**环境**的无数次交互，自主学习如何采取行动才能最大化奖励。
- **LLM 的强化学习循环 (The RL Loop for LLMs)**
- 在这个循环中，LLM扮演了“智能体”（Agent）的角色。
 - **行动 (Action):** LLM（智能体）生成一段文本（例如一个回答、一个故事、一段代码）。
 - **奖励 (Reward):** 这个分数就是**奖励信号**。高分代表“好”的行为，低分代表“坏”的。
 - **学习 (Learning):** 使用 PPO 等强化学习算法，根据奖励信号，微调LLM的生成策略，使其未来更有可能产生能获得高分的“行动”。
- **在“试错”中寻找最优策略：**与直接给予“正确答案”的监督学习不同，强化学习的核心是设定一个目标（奖励），然后让模型在与环境的无数次交互中，自主探索最大化奖励的行为模式。

上线后的 LLM 如何继续学习?



奖励的来源： 定义“好”与“坏”

- **关键问题：**“奖励”从何而来？
- “环境”的设计，直接决定了AI优化的方向。它可以是任何能提供清晰、量化反馈的系统。
- **基于模型的环境**
- **目标:** 生成**没有内容安全风险**的文本。
- **奖励模型:** 一个安全风险评估服务，它会对生成的文本进行打分。
- **奖励信号:** 安全风险分数（分数为 0–1，分数越接近0代表越安全），奖励与低风险分数正相关。
- **特点:** 适用于**主观、复杂且难以用简单规则定义的约束**。



奖励的来源：定义“好”与“坏”

智能体 (Agent) – LLM



行动 (Action)

LLM 生成一段文本



环境 (Environment) – 安全风险评估服务 (RM)

对文本进行评估，输出一个介于0-1的风险分数



奖励 (Reward) – 奖励 = 1 – 风险分数

风险分数越低，奖励越高。高奖励代表“好”的行为。



学习 (Learning)

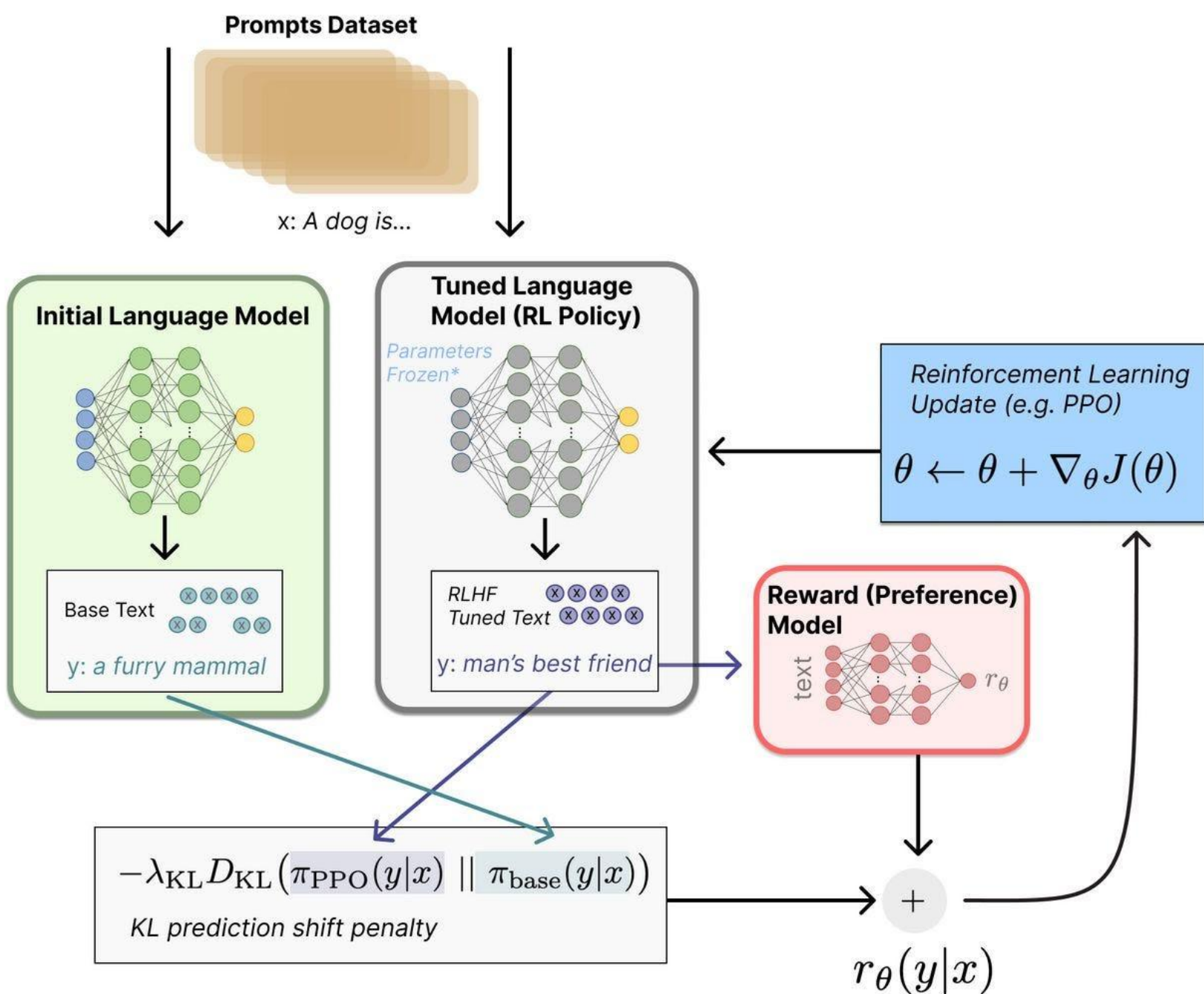
根据奖励信号，微调LLM的生成策略，使其未来更有可能生成能获得高分（低风险）的“行动”。

奖励的来源：定义“好”与“坏”

- 一个简单的例子
- LLM 生成了一个可能包含**敏感词汇**的回答。
- **安全风险评估服务 (RM)** 对此回答进行了打分，认为风险很高，给出了 **0.8** 的风险分数。
- 系统计算出的奖励为 $1 - 0.8 = 0.2$ ，这是一个**很低的奖励**。
- **强化学习算法**（例如 PPO）收到这个负面信号，立即对LLM的生成策略进行调整，让它在未来面对类似情境时，生成**包含该敏感词汇的回答的概率大大降低**。
- **核心价值：跳出“正确答案”的框架**
- 通过这个循环，LLM不再需要一个庞大的、人工标注的“安全回答数据集”去学习。它只需要一个能**客观评估其行为**的奖励模型，就能自主地通过试错，学会如何遵守复杂且难以用简单规则描述的“内容安全”这一约束。

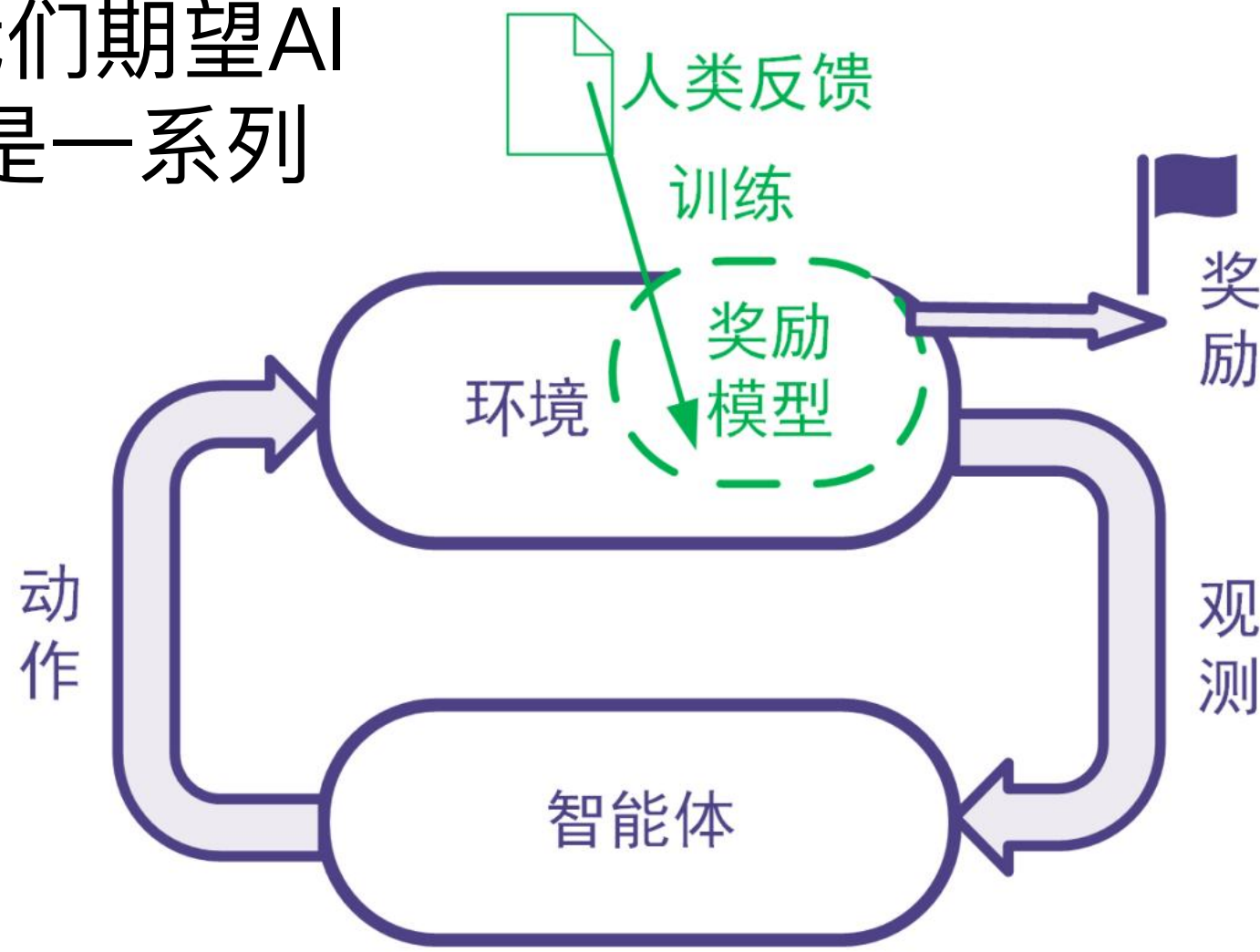
RLHF 详解：大模型的价值对齐

- **目标：** 根据RM的打分，持续微调大语言模型，使其生成的回答能获得更高的“人类偏好”分数。
- **大语言模型 (Agent):** 获得一个用户输入的“指令”。
- **生成回答 (Action):** 大语言模型根据指令生成一个**回答**。
- **RM 打分 (Reward):** 将 (指令, 回答) 对输入到奖励模型 (RM) 中，得到一个**分数(Reward)**。
- **情况A：高奖励 (High Reward)**
 - **动作：强化 (Reinforce)**
 - **描述：** 通过梯度更新，识别并**加强**生成这个“好”回答的内部决策路径（即一系列词的选择）。这使得模型在未来更有可能采取相似的路径，生成类似的高分回答。
- **情况B：低奖励 (Low Reward)**
 - **动作：削弱 (Penalize)**
 - **描述：** 通过梯度更新，**削弱**导致这个“坏”回答的决策路径。这使得模型在未来会主动规避这种路径，减少生成低分回答的概率。



RM 的本质：实现“行为主义的环境对齐”

- 我们之前讨论的奖励模型（RM），无论是用于内容安全评估，还是模拟人类偏好，其最终目的只有一个：**实现AI智能体与特定环境的“对齐”**。
- **第一步：定义“环境” (Define the Environment)** 首先，我们必须明确我们期望AI**适应**的“环境”是什么。在LLM的强化学习中，“环境”并非物理空间，而是一系列我们设定的**目标与约束**。例如：
 - 一个客观的技术指标：代码**编译是否通过**。
 - 一个明确的业务目标：回答能否**提升用户转化率**。
 - 一个复杂的安全策略：内容是否符合**安全规范**。
 - 一套抽象的社会价值观：AI的回答是否**诚实、无害、有益**。
 - ...
- **执行“对齐” (Perform the Alignment)** AI智能体（LLM）通过强化学习，不断与RM进行交互，并调整自身策略，以最大化从RM那里获得的奖励。
- 本质上就是：**行为主义的环境对齐**，即通过构建一个代表“环境”意志的奖励模型，来引导和塑造AI智能体的行为，使其与我们期望的目标和价值观保持一致，**不再需要人工标注大量的指令微调数据**。



练习

- 使用 MCP 实现课程中的上下文工程的全流程，要求用到 Prompts, Tools 和 Resources。

THANKS

 极客时间 | 训练营