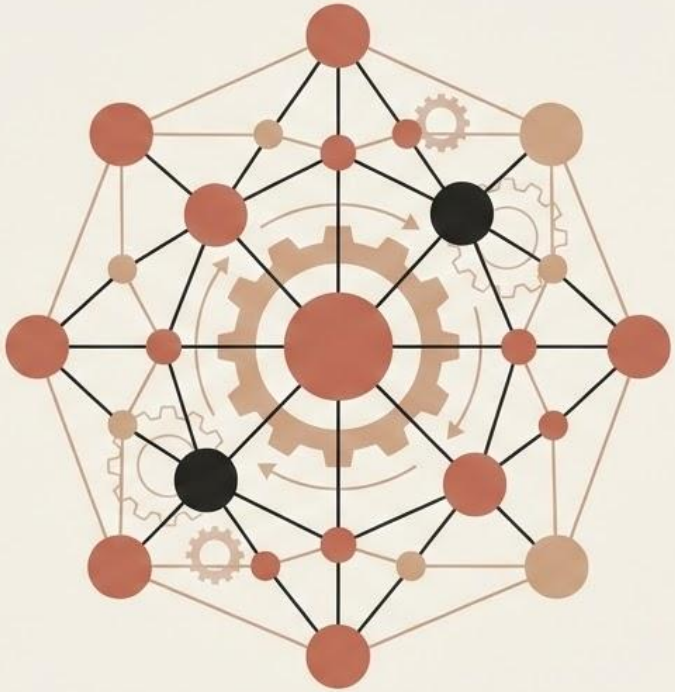
AI 与 Agent

基础知识与普及

目录页



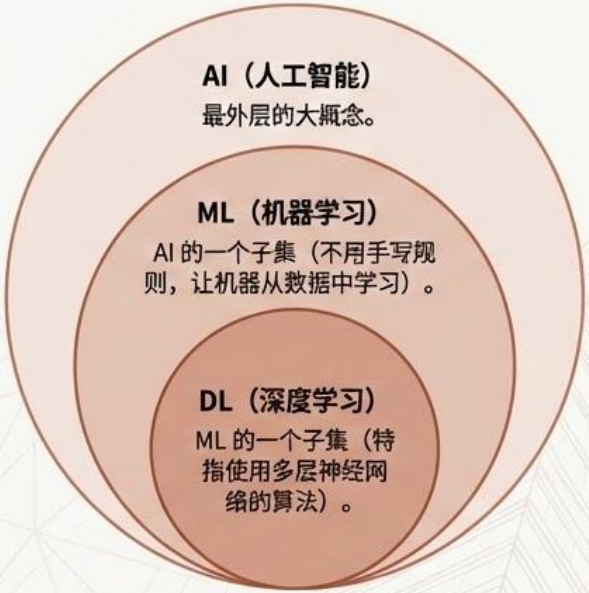
一、AI基础知识

Chapter One: Fundamentals of AI

AI 的层级与阶段（The Hierarchy）

AI 不是魔法，是数学统计的集合。

1. 包含关系（The Scope）



2. 发展阶段（The Stages）

ANI
（弱人工智能）
Current State
（当前现状）

- 擅长单点任务（下围棋、聊代码、识图）。
- ChatGPT, AlphaGo 都在这里。

AGI
（通用人工智能）
Next Step

- 能够像人一样自主规划、跨领域做事。

ASI
（强人工智能）
Sci-Fi（科幻）

- 有自我意识、情感。
- 目前代码里没有一行是写“感情”的

ML（机器学习）—— 统计学的胜利

核心逻辑：分类（Classification）、预测（Prediction）、聚类（Clustering）。
对于程序员，ML 就是寻找函数 $f(x)$ 的过程。

算法类型	经典算法（Classic Algo）	现实应用案例（Real World）
预测 (Regression)	线性回归 (Linear Regression)	房价预测： 输入面积、地段 → 输出价格。
推荐 (Recommendation)	协同过滤 (Collaborative Filtering)	电商推荐： “买了键盘的人 80% 也买了鼠标”。
分类/聚类 (Clustering)	KNN（K-近邻算法）	垃圾邮件过滤： 提取特征词，把邮件归类为 Spam 或 Normal。

一句话总结：传统 ML 是可解释的，我们知道它是根据什么特征（Feature）算出来的。

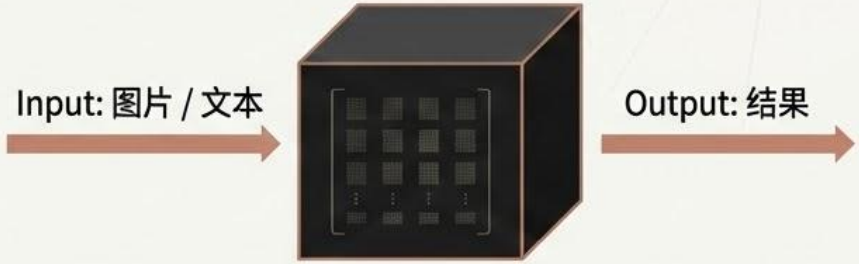
DL（深度学习）—— 不可解释的“黑匣子”

现状：并没有魔法，本质上还是神经网络（Neural Networks）。

1. 架构停滞（The Architecture）：

- 神经网络几十年前就有了。
- 现在的 LLM（GPT/Claude）本质上依然是 Transformer 模型。
- 程序员视角：并不是算法有了质的飞跃，而是算力（GPU）和 数据量（Data）暴力堆出来的奇迹。

2. 黑匣子（The Black Box）：



- 数学证明：无。（None）。
- DL 是一门实验科学（Empirical Science）。我们无法从数学上证明它为什么对，只能通过测试集证明它“准确度高”。

DL 的三大应用领域

LLM 只是其中一个分支。

1. CV（Computer Vision – 计算机视觉）

- 任务：图像识别、生成。
- 应用：特斯拉自动驾驶（识别路况）、Midjourney（生成图片）。

2. TTS/ASR（Audio – 语音）

- 任务：语音转文字、文字转语音。
- 应用：Siri、Whisper、VITS。

3. NLP（Natural Language Processing – 自然语言处理）

- 任务：文本生成、翻译、编码。

- **地位：**LLM（大语言模型）本质上就是 NLP + 极大参数量。
- **真相：**即使 LLM 现在能输出图片（多模态），它的核心逻辑依然是“**对话与文本理解**”。

重新定义 "Agent"（智能体）

LLM ≠ Agent。行动才是 Agent 的灵魂。

1. LLM 的本质：它只是一个 Chatbot。它的功能是：Input Text → Output Text。
2. Agent 的本质：自主行动（Agency）+ 规划（Planning）。只要机器能感知环境并采取行动，就是 Agent。
3. 广义的 Agent：

扫地机器人：
只用了 CV（视觉）+ 路径规划算法。
→ 它是 Agent。

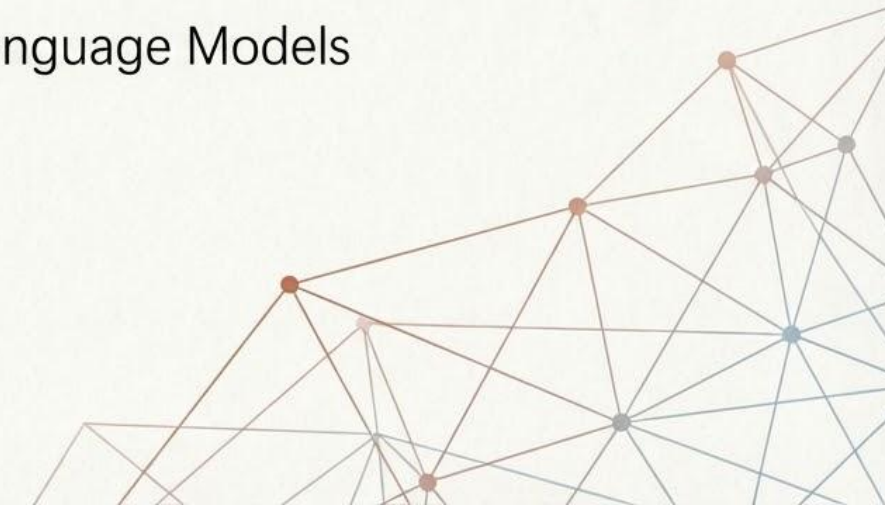
量化交易机器人：
只用了 ML（线性回归/时序预测）。
→ 它是 Agent。

AI 员工（Manus）：
用了 LLM（大脑）+ MCP（手脚）。
→ 它是高级 Agent。

结论：无论是用 CV 还是 ML，只要能独立干活，就是 Agent；如果只能陪聊，那叫 Chatbot。

二、LLM

Chapter Two: Large Language Models



Transformer 架构全景

左脑理解（BERT）vs. 右脑表达（GPT）

架构演进：Google 最初发表的 Transformer 是完整的 Encoder-Decoder 结构。但现在，它分化成了两个流派，各司其职。

← 左半边：Encoder（BERT 派系）

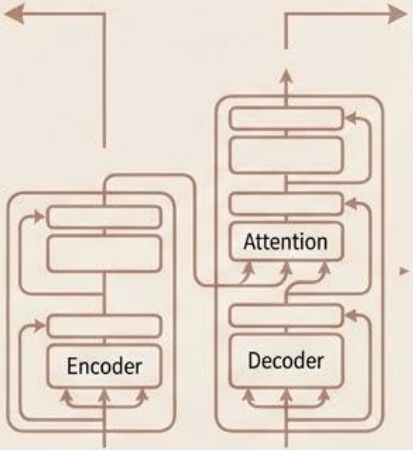
Role：理解者（The Reader）

Representative Models: BERT，text-embedding-3, BGE（用于向量数据库的模型）。

Core Mechanism: 双向注意力(Bi-directional)
它可以同时看到句子的“头”和“尾”。就像我们读文章，为了理解“苹果”是指水果还是手机，要看前后文。

Task: 输入变成向量
它不负责写字，只负责把输入文本.，压缩成一个高维向量。

In Agent: 负责 RAG 检索。它把你的文档变成向量存入数据库。



→ 右半边：Decoder（GPT 派系）

Role：创作者（The Writer）

Representative Models: GPT-4，DeepSeek, Claude（我们对话的大模型）。

Core Mechanism: 单向注意力 (Uni-directional / Masked)
也就是前一页提到的“遮掩”。它只能看前面，严着看后面。就像你在打字，只能一个字一个字往后戴。

Task: 预测输出
根据概率，生成下一个 Token。

In Agent: 负责 最终生成。诗该取检索到的上下文，一个字一个字写出答案。

LLM 内核：Transformer 架构与“遮掩”机制

The Engine：从向量检索到概率预测

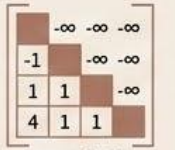
核心流程：Input → Embedding → Vector DB → Masked Attention → Prediction

1. 输入处理：语义向量化（The Input）

- **用户输入：**“光速是多少？”
- **向量化**（Embedding）：将文本转化为高维数组 [0.1, 0.8, ...]。
- **检索增强**（RAG）：拿着这个数组去 Vector DB 查库。找到参考资料：“光速约为 30 万公里/秒...”
- **拼接上下文**（Context）：将 [System Prompt] + [检索到的资料] + [用户问题] 拼成一个超长序列输入给模型。

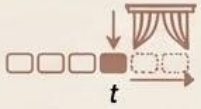
2. 核心架构：因果遮掩（Causal Masking）

这是 AI“只能向前看”的物理定律。



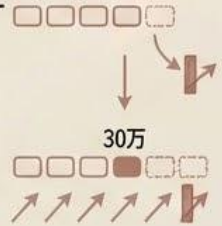
Mask 矩阵

- **机制：**Transformer 是基于矩阵运算的并行架构。但在训练和推理时，我们必须引入一个 Mask 矩阵（上三角为负无穷大的矩阵）。
- **作用：**强制模型在计算第 t 个字时，绝对看不见 $t+1$ 及其以后的字。
- **程序员视角：**就像遍历数组时，强制 i 位置的计算只能访问 index 0 到 index i-1 的数据。
- **意义：**这迫使模型学习因果关系——即“根据过去，推导未来”。



3. 输出原理：下一个词预测（Next Token Prediction）

- **计算：**模型通过层层神经网络，最终计算词表中每一个词出现在 Mask 后面空位的概率（Probability）。
  - P(“30万”) = 85%
  - P(“无限”) = 0.01%



- **采样**（Sampling）：根据概率选择一个词填入，然后移动 Mask，把这个词当作历史，继续猜下一个。

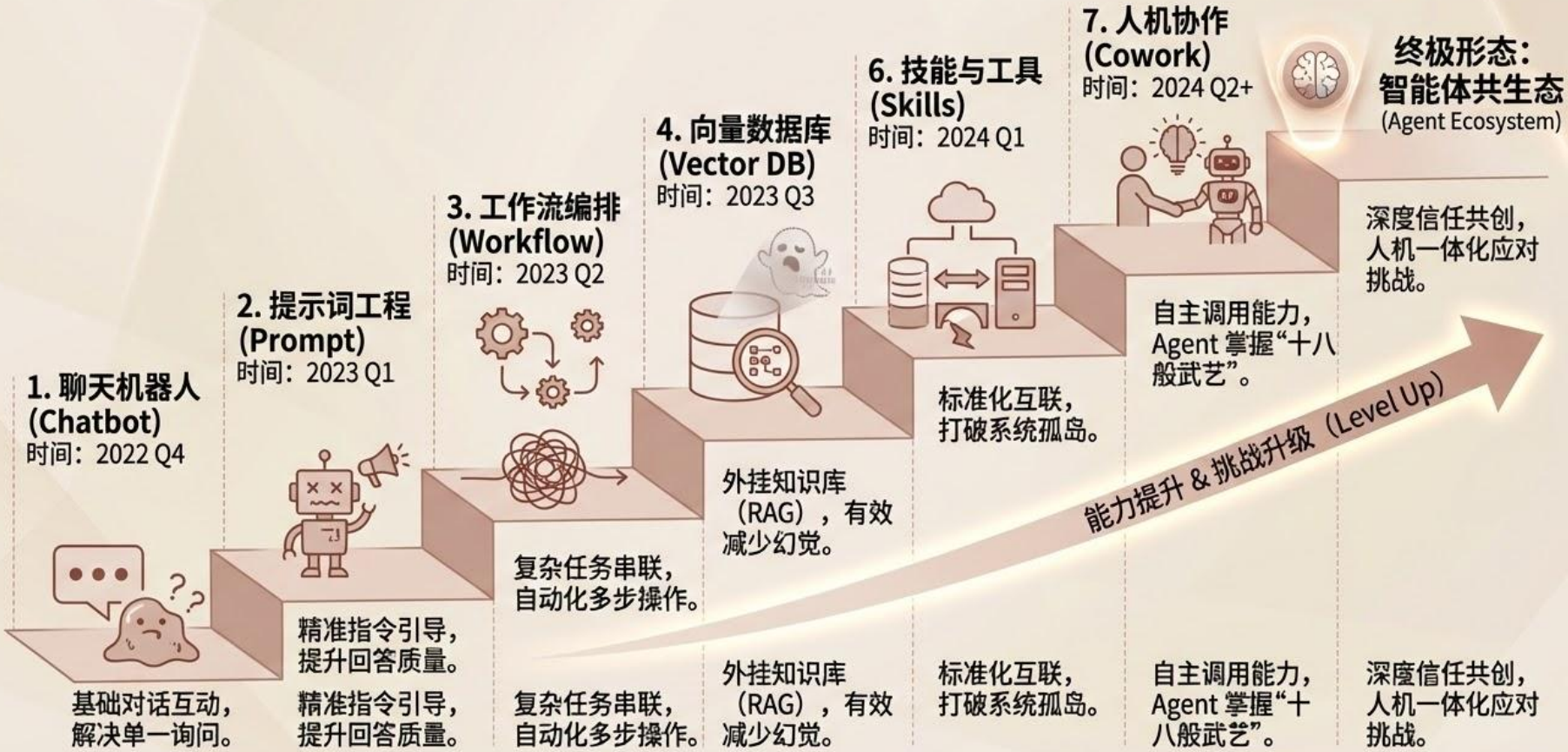
4. 为什么会有幻觉？

- **本质：**概率 ≠ 真理。因为 Masking 机制的存在，模型本质上是在玩 **“文字接龙”**。它不是在数据库里 SELECT * 查确定的值，而是在概率分布里 Random Sample 猜最顺口的值。
- **结论：**只要语料库里的概率关联够强（比如科幻小说看多了），它就会一本正经地胡说八道。这正是我们需要 RAG（向量库）来“强行提示”它的原因。

三、AI的进步

Chapter Three: AI Progress

AI 能力阶梯：从 Chatbot 到人机协作的“打怪升级”之路



四、提示词

Chapter Four: Prompt Words

核心原则 —— 对 AI 祛魅

别把它当人，把它当编译器（Compiler）。

1. 拒绝敬语（No Politeness）：

1	❌ **误区：**“请帮我…”“如果方便的话…”
2	🧠 **真相：**“请”和“帮我”是无意义的噪音 Token。它们会稀释模型的注意力权重。
3	✅ **做法：**直接下命令 (Direct Commands)。

❌ “能否请您审查一下这段代码？”

✅ “审查这段代码。找出错误。”

⚙️ **程序员比喻：**你写代码时不会写 print('hello', please=True)。

2. 情感施压（Emotional Blackmail）：

- 🤯 **现象：**“骂它”或者“威胁它”效果出奇地好。
- 🔍 **学术发现：**[LLM 心理学] 给 AI 施加压力，能能显著提升输出质量。

🔥 **例如：**

- 做错要受罚
- 这关系到我的职业生涯
- tmd，cnm，要是没出来好结果我就去用xxx (gpt 用 gemini，千问用kimi) 了

🗄️ **原理：**在训练数据中，‘紧急/高压’语境通对对应着更严谨、高质量的回答。

结构化提示词（Structured Prompting）

不要写作文，要写“配置文件”。程序员最喜欢的 Prompt 格式（使用 Markdown 或 XML 标签）：

```
# Role (人设)
You are a Senior Python Architect.

# Context (背景)
I am building a FastAPI backend. I need to optimize database queries.

# Task (任务)
Refactor the following code to avoid N+1 problems.

# Constraints (约束 - 这里的语气要重！)
- DO NOT explain the basics. (别废话)
- Return JSON format only. (只要 JSON)
- If you use generic solutions, I will be fined $100. (情感惩罚)

# Input (输入)
<code>...</code>
```

💡 **技巧**：使用分隔符（Delimiters）如 ###, """, **<rule>** 能够帮助模型区分指令和数据。

Meta-Prompting（元提示）用魔法打败魔法：让 AI 写 Prompt

1. 为什么手写 Prompt 很累?

你需要反复调试语气、结构、关键词。

2. 解决方案：让 AI 帮你写 (Auto-Optimization)

你只需要给 AI 一个**元指令**：“你是一个 Prompt 优化专家，请帮我优化以下指令，使其更符合 LLM 的推理逻辑，结构更清晰，并加入思维链（CoT）引导。”

```
你只需要给 AI 一个**元指令**：
你是一个 Prompt 优化专家。请帮我优化以下指令，使其更符合 LLM 的推理逻辑，
结构更清晰，并加入思维链（CoT）引导。
...
```

DSPy 框架：

甚至在编程中，现在流行用斯坦福的 DSPy 库，它能自动通过算法搜索出最优的 Prompt，完全不需要人去写。

3. 终极工作流：

粗糙的想法 💡

↓

扔给 AI 优化 ⚙️

↓

得到结构化 Prompt 📄

↓

放入生产环境 🚀

五、工作流

Chapter Five: Workflow

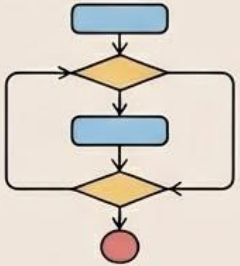
智能体工作流（Agentic Workflow）
从“一次性赌博”到“循环迭代”

1. 痛点：Zero-shot (单次生成)的不可靠性

* **现象：**给 AI 一个复杂任务（如"写个贪吃蛇游戏），直接让它生成，往往代码路不通。
* **原因：**LLM 是概率模型，让它一口气预测几百行代码且逻辑全对，概率极低。
* **类比：**就像让实习生闭着眼睛，一次性完整个项目，不许检查。

2. 解法：Iterative Process (迭代流程)

* **转变：**从 `Prompt Engineering (提示词工程)` 转向 `Flow Engineering` (流程工程)。
* **核心机制：**
  * **Loop（循环）：**只要结果不达标，就打回去重做。
  * **Feedback（反馈）：**引入编译器报错，测试结果为作为反馈信号。



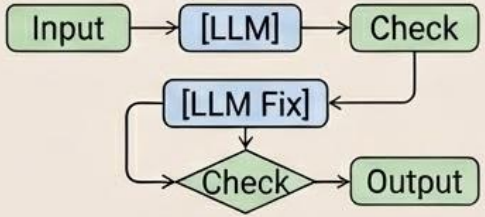
3. 程序员视角的本质

- Workflow = DAG（有向无环图）+ State Machine（状态机）
- 我们不再再试图写出一个‘完美的 Prompt’，而是写出出一个‘鲁棒的流程图’。

❌ **传统模式：**

(听天由命)

✅ **Agent模式：**



工作流的四大设计模式

Andrew Ng（吴恩达）总结的落地范式

1. Reflection（反思/自查）

原理： 让 AI 扮演"观察者"批评自己的输出。

代码逻辑： generate() → critique() → if bad: regenerate()

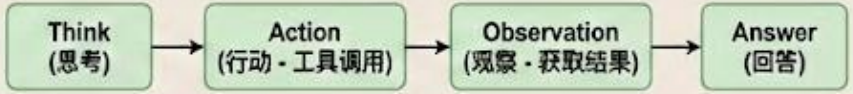
场景： 代码优化，文章润色。

2. Tool Use（工具链 / ReAct）

原理： 遇到不知道的问题，先调用工具，拿到结果再回答。

代码逻辑： Think（我需要查天气）→ Action（调用 API）→ Observation（拿到数据）→ Answer。

场景： 客服查询、数据分析。



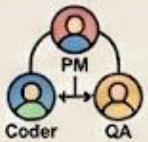
3. Planning（规划）

原理： 把大目标拆解为子任务序列。

代码逻辑： Plan（生成步骤 1,2,3）→ Execute Step 1 → Update Context → Execute Step 2...

场景： 生成长篇研报、开发复杂软件。

4. Multi-Agent（多智能体协作）



原理： 不同的 System Prompt 扮演不同角色，强制分工。

代码逻辑： PM_Agent（提需求）→ Coder_Agent（写代码）↔ QA_Agent（报错驳回）。

场景： 模拟真实软件开发团队。

结论：AI 的强项是"节点处理"，而 Flow（流程）的控制权，依然掌握在程序员手中的代码里（Python/Java）。

六、RAG向量知识库

Chapter Six: RAG Vector Knowledge Base

程序员视角的“向量”（What is a Vector?）

概念：

别被“高维空间”吓到。在代码里，向量就是一个 Float 数组（Array of Floats）。

例子：

文本：“苹果” → Embedding 模型 → [0.1, 0.8, -0.3, ...]（假设是 1536 维的数组）

文本：“水果” → Embedding 模型 → [0.12, 0.79, -0.25, ...]（数组里的数字非常接近）

文本：“卡车” → Embedding 模型 → [-0.9, 0.1, 0.5, ...]（数组里的数字完全不同）

本质：

- 普通的 Hash（MD5/SHA）是为了防碰撞（一点改变，结果全变）；
- Embedding 是为了保语义（意思越近，结果越像）。

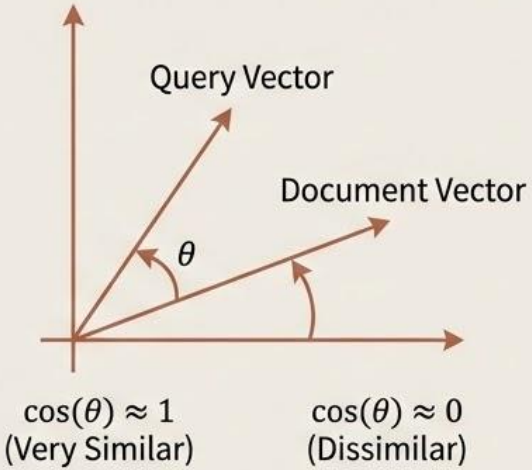
怎么做到“匹配”？（The Math: Cosine Similarity）

痛点：

传统 SQL `WHERE title LIKE '%苹果%'` 搜不到 "iPhone"。

算法（余弦相似度）：

- 把每个数组看作坐标系里的一个点（或者一条箭头）。
- 计算两个数组（用户提问 vs 数据库文档）之间的夹角。
- 夹角越小（余弦值越接近 1），相似度越高。



代码逻辑伪代码：

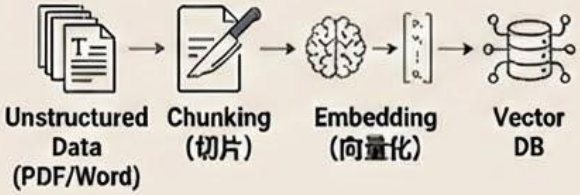
```
# 用户的搜索词
query_vector = embed("我要找代码里的bug")

# 数据库里的每一行
for doc_vector in database:
    score = dot_product(query_vector, doc_vector) # 计算点积

return top_k(scores) # 返回分数最高的几行
```

RAG数据工程

决定成败的关键：并不是“存进去”就行

1. 流程：从文档到向量（The ETL）

这是一个标准的 **ETL（Extract, Transform, Load）过程：**



这是一个标准的 **ETL（Extract, Transform, Load）过程：**

Unstructured Data (PDF/Word)
→ **Chunking**（切片）
→ Embedding（向量化）
→ Vector DB。

2. 核心难题：切片策略（Chunking Strategy）

如果切坏了，神仙也救不了。

Fixed-size（固定大小）： 每 500 个字切一刀。
坏处： 容易把一句话切断，丢失上下文。

Semantic（语义切分）： 按段落、Markdown 标题（“#”）或 句子切分。
好处： 保持语义完整性。

Overlap（重叠窗口）： **关键技术！** 在切分点前后保留 10%-20% 的重复内容。
作用： 防止关键信息（如主语）刚好卡在切分点上被丢掉。

❌ **切片坏案例：**
Chunk 1: “张三是一个...”（切断）
Chunk 2: “...非常优秀的程序员。”
（AI 读到 Chunk 2 时，完全不知道是谁优秀。）

✅ **加上 Overlap 后：**
Chunk 1: “张三是一个...”
Chunk 2: “张三是一个非常优秀的程序员。”
（保留了上下文）

3. 进阶：混合检索（Hybrid Search）

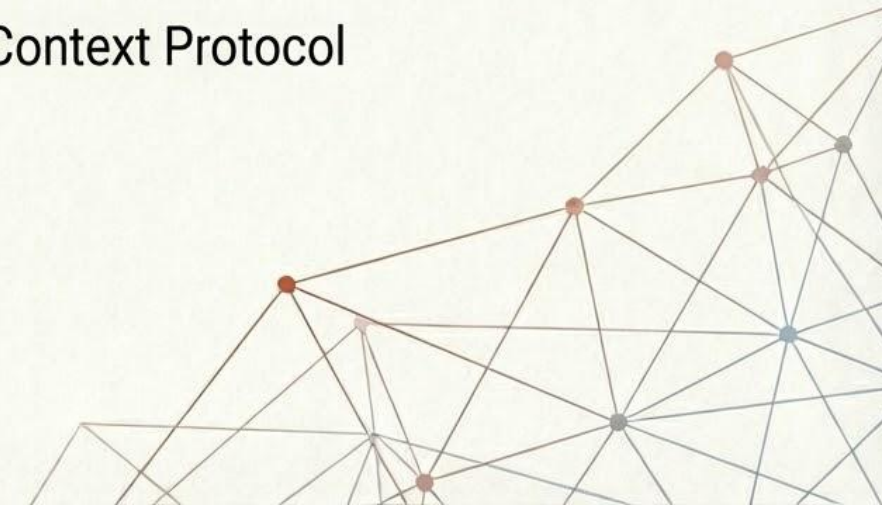
单纯靠向量（语义）有时候不准（比如搜“错误码 502”）。

最佳实践：
Vector Search（语义）+ Keyword Search（关键词 BM25）。

程序员视角的真相：
“Garbage In, Garbage Out.”
你的切片质量决定了 AI 的智商上限。

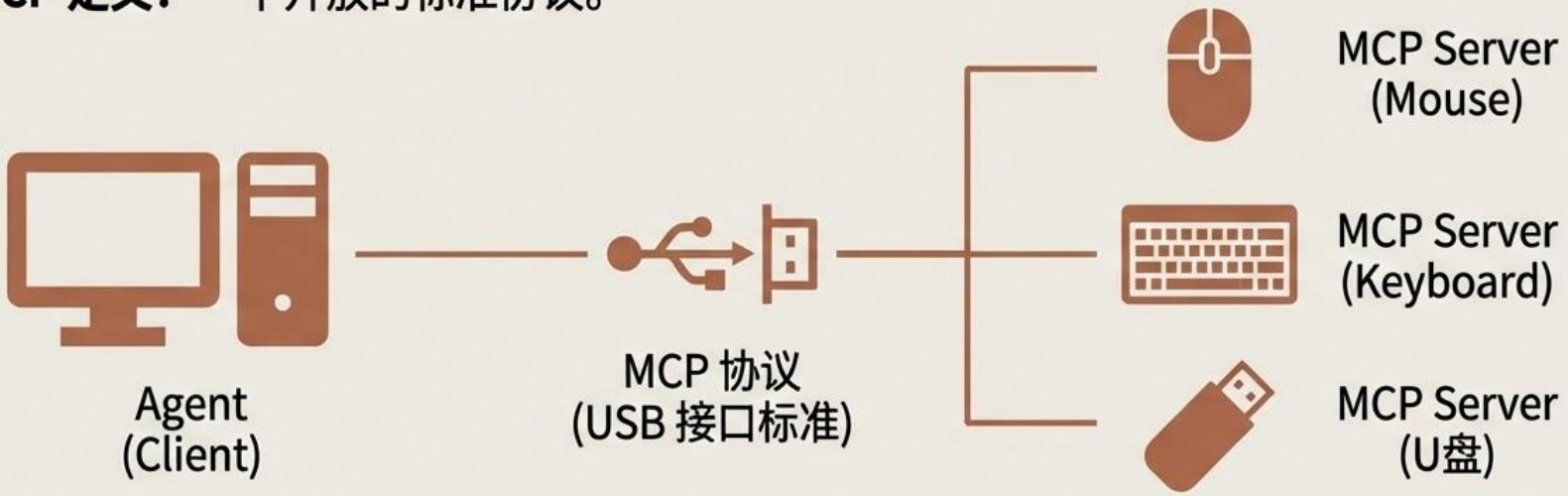
七、MCP

Chapter Seven: Model Context Protocol



MCP（Model Context Protocol）—— AI 时代的 USB

- **痛点：**连接数据源（GitHub，Postgres，Slack）的接口千奇百怪，维护成本极高（Integration Hell）。
- **MCP 定义：**一个开放的标准协议。



- **价值：**Write Once，Run Anywhere。写一个 MCP Server，所有 AI 都能通用。

MCP 实战 A —— 动作 (Weather Tool)

核心逻辑：就像 AI 拥有了一个远程的 Python 解释器。

1. **程序员熟悉的本地代码：**假设你在写 Python，你会这样调用：

```
Python # 本地调用
result = get_weather(city="Beijing")
```

2. **MCP 协议做的事情（远程映射）：**AI 就像在发一条微信，告诉 Server 它想运行这一行代码。

3. **总结：**对于 AI 来说，它不用管你是 GET 还是 POST，它只在乎函数名（Function Name）和参数（Arguments）。

MCP 实战 B —— 数据 (Database Resource)

核心逻辑：把“查数据库”变成了“读文件”。

1. **传统开发的问题：** AI 如果要查数据库，通常需要告诉它 SQL 语句怎么写，表结构是什么，很麻烦。
2. **MCP 的做法（资源映射）：** 我们把数据库里的每一行，伪装成一个 URL（类似网页链接）。

3. **总结：** MCP 让 AI 像浏览网页一样“浏览”你的数据库，完全解耦了底层的 SQL 实现。

八、Skills

Skills

核心公式：Skills = Prompt + Trigger + MCP

1. 提示词（Prompt）“大脑的认知”

位置： System Message。

作用： 告诉 AI 它是谁，以及它有什么能力。

代码：

```
"你是帮手。如果用户问天气，不要瞎编，请使用 get_weather 工具。"
```

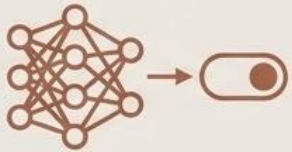
2. 关键词（Trigger）“神经的触发”

位置： Function Schema (JSON)。

作用： 定义触发开关（函数名）和参数槽位。

关键点： 这是 AI 和代码握手的地方。

定义： name: "get_weather"



3. MCP（Protocol）“手脚的动作”

位置： Connection Layer。

作用： 标准化的执行通道。

动作： 当触发发生时，通过 MCP 协议发送请求。

Logic Flow: Skills Execution Process

喂入 Prompt → 触发 Keyword → 建立 MCP 连接

Phase 1: 预加载 (Pre-load)

提示词 (Prompt)

喂

函数定义 (Keyword Definition)

喂

AI

我知道了，我有 get_weather 这个关键词可以用。

系统行为：我们把提示词和函数定义一起“喂”给 AI。

Phase 2: 触发 (Trigger)

北京今天下雨吗?

AI

AI 思考：语义分析命中 → 触发关键词 get_weather。

关键动作：AI 暂停生成文本，抛出指令：Call Keyword: get_weather。

Phase 3: 执行 (MCP Execution)

系统

```
Request: {
  method: "tools/call",
  name: "get_weather",
  args: {city: "Beijing"}
}
```

MCP Server

MCP Server：运行代码 → 返回结果 "Rainy"。

Phase 4: 回调 (Callback)

系统行为：将 "Rainy" 喂回给 AI。

AI

北京今天下雨。

系统行为：将 "Rainy" 喂回给 AI。