大模型开发教程核心知识速记手册

第一部分：LangChain 核心组件

LangChain 通过模块化的组件，简化了大型语言模型（LLM）应用的开发流程。核心组件包括模型（Models）、提示（Prompts）和输出解析器（Output Parsers）。

1. 模型 (Models)

模型是执行文本理解和生成任务的“大脑”。LangChain 提供了统一的接口来调用不同厂商的 AI 模型。

- **分类**：
  - **LLMs (Language Models)**：传统的文本输入、文本输出模型。
  - **Chat Models (聊天模型)**：专为对话设计，接收一个包含不同角色（如 `system`, `human`, `ai`）的消息列表作为输入，并返回一个 AI 消息。**这是当前的主流。**
- **作用**：作为应用的推理核心，根据提示执行任务。
- **示例 (DeepSeek)**：

```
from langchain_deepseek import ChatDeepSeek

# 初始化一个聊天模型实例
llm = ChatDeepSeek(model="deepseek-chat", temperature=0)
```

2. 提示 (Prompts)

提示是发送给模型的指令，它指导模型的行为并提供必要的上下文。

- **核心组件**：**提示模板 (Prompt Templates)**，用于动态生成提示。
- **分类**：
  - `PromptTemplate`：用于简单的字符串模板。
  - `ChatPromptTemplate`：用于构建包含多条、不同角色消息的模板，是与聊天模型交互的标准方式。
- **作用**：
  - **指令化**：明确告知模型扮演的角色和任务。
  - **情境化**：动态地将变量（如用户输入、背景知识）填入模板。
  - **格式化**：可以包含输出格式的要求。
- **示例 (ChatPromptTemplate)**：

```
from langchain_core.prompts import ChatPromptTemplate

prompt = ChatPromptTemplate.from_messages([
    ("system", "你是一个乐于助人的助手，负责将{input_language}翻译成{output_language}。"),
    ("human", "{text}")
])
```

3. 输出解析器 (Output Parsers)

输出解析器负责将 LLM 返回的原始文本字符串转换为结构化的、便于程序处理的数据格式（如 JSON, Python 对象等）。

- **作用**：
  - **结构化输出**：将模型的自然语言回复转换为程序可用的对象。
  - **提供格式指令**：解析器可以生成一段文本指令（`get_format_instructions()`），加入到提示中，告诉模型应该以何种格式输出，从而提高解析成功率。
- **常用解析器**：
  - `StrOutputParser`: 最简单的解析器，直接返回字符串。
  - `JsonOutputParser`: 解析 JSON 格式的字符串。
  - `PydanticOutputParser`: 将输出解析为预定义的 Pydantic 模型对象，功能强大且类型安全。
- **工作流程**：
  1. 定义你期望的输出 `schema`（如 `ResponseSchema` 或 Pydantic 模型）。
  2. 根据 `schema` 创建一个解析器实例。
  3. 从解析器获取格式指令 `format_instructions`。
  4. 将 `format_instructions` 整合到你的提示模板中。
  5. LLM 根据包含格式指令的提示生成文本。
  6. 使用解析器的 `parse()` 方法处理 LLM 的文本输出，得到结构化数据。

第二部分：LangChain 核心概念

1. 链 (Chains)

链是将多个组件（如模型、提示、其他链）组合起来，以执行一系列操作的机制。LangChain 表达式语言 (LCEL) 是构建链的推荐方式，使用 `|` 符号进行连接。

链类型	核心作用	特点
LLMChain (基础链)	将**提示模板**、**LLM** 和**输出解析器**连接起来。	LCEL 写法: `prompt
SimpleSequentialChain	**线性**串联多个链，前一个链的**单个输出**是后一个链的**单个输入**。	简单直接，适用于管道式任务。
SequentialChain	功能更强的顺序链，支持在链之间传递**多个输入和输出**。	通过 `input_variables` 和 `output_variables` 控制数据流，更灵活。
RouterChain	**动态路由**。根据输入内容，智能地选择最合适的子链来执行任务。	包含一个“路由器”来决策，适用于需要根据不同场景调用不同专家的应用。

2. 记忆 (Memory)

记忆组件用于在多轮对话中保持状态，使 LLM 能够"记住"之前的交互历史。

记忆类型	核心机制	适用场景
ConversationBufferMemory	存储**完整**的对话历史。	简短对话，或需要完整上下文的场景。
ConversationBufferWindowMemory	只存储最近的 **K** 轮对话。	控制上下文长度，防止超出模型限制。
ConversationTokenBufferMemory	根据**最大 Token 数**来存储最近的对话历史。	精确控制成本和上下文长度。
ConversationSummaryMemory	用 LLM 对**旧的对话进行总结**，只保留摘要。	极长的对话，以牺牲部分细节换取极大的上下文压缩。

当前最佳实践：虽然这些内存类仍然可用，但对于需要记忆功能的新应用，**LangChain 官方更推荐使用 LangGraph 及其内置的持久化（Checkpointer）机制**来管理状态，因为它更强大和灵活。

3. RAG (检索增强生成)

RAG 是让 LLM 能够利用外部私有知识库回答问题的核心技术。

- **核心流程**：
    1. **索引 (Indexing)**：
        - **加载 (Load)**：使用 `DocumentLoaders` 从文件、网页、数据库等来源加载数据。
        - **切分 (Split)**：使用 `TextSplitters` 将长文档切分成小块（chunks）。推荐使用 `RecursiveCharacterTextSplitter`。
        - **存储 (Store)**：
            - **嵌入 (Embed)**：使用 `Embedding` 模型将文本块转换为向量。
            - **存入向量数据库 (Vector Store)**：将文本块和向量存入 FAISS, Chroma, Milvus 等向量数据库中。
    2. **检索与生成 (Retrieval & Generation)**：
        - **检索 (Retrieve)**：用户提问时，使用 `Retriever` 在向量数据库中进行语义搜索，找出最相关的文档块。
        - **生成 (Generate)**：将检索到的文档块作为上下文，连同用户问题一起传给 LLM，生成最终答案。
- **核心组件**：`RetrievalQA` 链是实现此流程的经典组件。
- **对话式 RAG**：通过 `create_history_aware_retriever` 等函数，可以在检索前重写用户问题（结合对话历史），实现更智能的多轮问答。

4. 代理 (Agents)

Agent 赋予 LLM **使用工具**的能力，使其能够与外部世界交互并执行行动。

- **核心思想 (ReAct)**：Agent 在**推理 (Reasoning)** 和**行动 (Acting)** 之间循环。
    1. **思考 (Thought)**：LLM 分析任务，决定下一步行动。
    2. **行动 (Action)**：选择一个可用工具并提供参数。
    3. **观察 (Observation)**：获取工具的执行结果。
    4. LLM 根据观察结果进行下一轮思考，直至任务完成。
- **实现**：通过为 LLM 提供一个**工具列表**，并使用如 `create_react_agent` (基于 LangGraph) 等函数来构建 Agent 执行器。

第三部分：大模型微调

微调是通过在特定数据集上进一步训练，使预训练的 LLM 适应特定任务或领域的过程。

1. 为何微调?

- **提升特定任务性能**（如分类、摘要）。
- **适配领域知识**和专业术语。
- **控制输出风格**、语气和格式。
- **降低模型幻觉**，提升可靠性。

2. 关键准备工作

- **选择基础模型**：综合考虑模型架构、性能、规模、许可协议和可用计算资源（特别是 GPU VRAM）。
- **准备数据集**：**数据质量至上。**准备与任务高度相关、准确、多样化的数据集。
    - **指令微调 (SFT)**：常用 Alpaca 格式 (`instruction`, `input`, `output`)。
    - **对话微调**：常用 ShareGPT 或 ChatML 格式（`role`, `content` 的消息列表）。
    - **偏好对齐 (DPO)**：常用格式 (`prompt`, `chosen`, `rejected`)。

3. 参数高效微调 (PEFT)

PEFT 技术通过只训练模型参数的一小部分，来大幅降低微调的计算和存储成本。

PEFT 技术	核心原理	关键参数	优点
LoRA	冻结原模型，在特定层旁注入可训练的低秩矩阵 (A 和 B) 来近似权重更新。	`r` (秩), `lora_alpha` (缩放因子), `target_modules` (应用层)	参数高效，存储开销小，推理时可合并权重无额外延迟。

PEFT 技术	核心原理	关键参数	优点
QLoRA	**LoRA + 4-bit 量化。**将冻结的基础模型权重以 4-bit 精度加载，进一步降低显存。	`load_in_4bit`, `bnb_4bit_quant_type` (通常 "nf4"), `bnb_4bit_compute_dtype`	**极大降低显存占用**，使得在消费级 GPU 上微调大模型成为可能。

4. 偏好对齐技术

用于使模型输出更符合人类偏好。

- **DPO (Direct Preference Optimization)**：一种更简洁、更稳定的对齐方法。它绕过奖励模型训练，直接利用**偏好数据对 (`chosen`, `rejected`)** 来优化 LLM。是当前流行的 RLHF 替代方案。
- **RLAIF (Reinforcement Learning from AI Feedback)**：使用一个（通常更强大的）**AI 模型**而不是人类来提供偏好反馈，以降低标注成本和提高效率。

5. 评估

- **自动化指标**：
  - **Perplexity (PPL)**：衡量语言流畅度，越低越好。
  - **BLEU**: 用于翻译，衡量与参考译文的 N-gram 重叠度。
  - **ROUGE**: 用于摘要，衡量与参考摘要的 N-gram 召回率。
- **人工评估 / LLM-as-a-Judge**：人工评估是黄金标准。使用更强大的 LLM 作为“裁判”来评估输出质量，是一种可扩展的替代方案。