

# **Large Language Model in Action**

VII QA

2023-10-20



# 目录

|  |          |
|--|----------|
| 序言                                     | 1        |
| 简介                                     | 3        |
| <b>I. LLMs</b>                         | <b>5</b> |
| 1. Embedding                           | 7        |
| 2. 幻觉                                  | 9        |
| 3. Retrieval Augmented Generation      | 11       |
| 3.1. RAG 基本概念 . . . . .                | 11       |
| 3.1.1. RAG 的表示方法 . . . . .             | 12       |
| 3.1.2. Retrieval Source 类型 . . . . .   | 13       |
| 3.1.3. Retrieval Metrics 类型 . . . . .  | 13       |
| 3.1.4. Integration Method 类型 . . . . . | 13       |
| 3.2. 为什么要使用 RAG . . . . .              | 14       |
| 3.3. 更多内容 . . . . .                    | 15       |
| 3.4. 参考文献 . . . . .                    | 15       |

## 目录

|                                      |           |
|--------------------------------------|-----------|
| <b>4. Agent</b>                      | <b>17</b> |
| 4.1. 阿克琉斯之踵 . . . . .                | 17        |
| 4.2. 什么是 Agent . . . . .             | 18        |
| 4.3. ReAct 模式 . . . . .              | 20        |
| 4.4. PlanAndExecute 模式 . . . . .     | 22        |
| 4.5. 参考文献 . . . . .                  | 24        |
| <br>                                 |           |
| <b>II. LangChain</b>                 | <b>27</b> |
| <br>                                 |           |
| <b>5. LangChain 简介</b>               | <b>29</b> |
| 5.1. LangChain 的目标 . . . . .         | 30        |
| 5.2. LangChain 的基本概念 . . . . .       | 33        |
| 5.2.1. Prompt Templates . . . . .    | 34        |
| 5.2.2. LLMs . . . . .                | 34        |
| 5.2.3. Output Parsers . . . . .      | 36        |
| 5.2.4. LLMChain . . . . .            | 37        |
| 5.3. Langflow . . . . .              | 38        |
| 5.4. LangChain 的学习资料 . . . . .       | 38        |
| 5.5. 参考文献 . . . . .                  | 40        |
| <br>                                 |           |
| <b>6. LangChain 序列化</b>              | <b>49</b> |
| 6.1. 序列化 . . . . .                   | 49        |
| 6.2. LangChain-Hub . . . . .         | 49        |
| <br>                                 |           |
| <b>7. LangChain Retrieval</b>        | <b>51</b> |
| 7.1. Document loaders . . . . .      | 52        |
| 7.2. Document transformers . . . . . | 53        |
| 7.3. Text embedding models . . . . . | 54        |

## 目录

|   |           |
|---|-----------|
| 7.4. Vector stores . . . . .                    | 55        |
| 7.5. Retriever . . . . .                        | 57        |
| 7.6. RetrievalQA . . . . .                      | 59        |
| <b>8. LangChain 函数调用</b>                        | <b>67</b> |
| 8.1. 大模型的时效性 . . . . .                          | 68        |
| 8.2. 函数调用流程 . . . . .                           | 68        |
| 8.3. 示例 . . . . .                               | 70        |
| 8.3.1. OpenAI API . . . . .                     | 70        |
| 8.3.2. LangChain 中调用 OpenAI Functions . . . . . | 70        |
| 8.4. 参考文献 . . . . .                             | 71        |
| <b>9. LangChain ReAct Agent</b>                 | <b>79</b> |
| 9.1. 三大基本组件 . . . . .                           | 79        |
| 9.1.1. 初始化 LLM . . . . .                        | 79        |
| 9.1.2. 初始化 Tool . . . . .                       | 80        |
| 9.1.3. 初始化 Agent . . . . .                      | 82        |
| 9.2. Zero Shot Agent . . . . .                  | 83        |
| 9.2.1. 深入 Zero Shot Agent . . . . .             | 85        |
| 9.3. Conversational Agent . . . . .             | 86        |
| 9.4. Agent 提示词工程 . . . . .                      | 89        |
| 9.5. Docstore Agent . . . . .                   | 92        |
| 9.5.1. 单输入参数和多输入参数 . . . . .                    | 93        |
| 9.6. Structured Chat Agent . . . . .            | 94        |
| 9.7. 自定义 Agent Tools . . . . .                  | 96        |
| 9.7.1. 单输入参数 . . . . .                          | 96        |
| 9.7.2. 多输入参数 . . . . .                          | 97        |
| 9.8. 总结 . . . . .                               | 97        |

## 目录

|  |            |
|--|------------|
| <b>10. LangChain OpenAI Function Agent</b> | <b>105</b> |
| <b>11. LangChain PlanAndExcute Agent</b>   | <b>107</b> |
| <b>12. Langflow</b>                        | <b>109</b> |
| <br>                                       |            |
| <b>III. Embedchain</b>                     | <b>111</b> |
| <b>13. Embedchain 简介</b>                   | <b>113</b> |
| <br>                                       |            |
| <b>IV. AutoGen</b>                         | <b>115</b> |
| <br>                                       |            |
| <b>V. Case Study</b>                       | <b>117</b> |
| <b>14. Case1</b>                           | <b>119</b> |
| <br>                                       |            |
| <b>References</b>                          | <b>121</b> |
| <br>                                       |            |
| <b>附录</b>                                  | <b>123</b> |
| <br>                                       |            |
| <b>A. 术语表</b>                              | <b>123</b> |
| <br>                                       |            |
| <b>B. LangChain 安装指南</b>                   | <b>125</b> |
| <br>                                       |            |
| <b>C. Milvus Beginner</b>                  | <b>127</b> |
| C.1. Milvus 安装 . . . . .                   | 127        |
| C.1.1. 1. 安装 docker-ce . . . . .           | 127        |
| C.1.2. 2. 安装 docker-composer . . . . .     | 127        |

## 目录

|  |     |
|--|-----|
| C.1.3. 3. 安装 docker-milvus 并启动 . . . . . | 128 |
| C.2. Milvus 测试 . . . . .                 | 128 |
| C.2.1. 安装 Milvus SDK . . . . .           | 129 |
| C.2.2. 测试 Milvus . . . . .               | 129 |
| C.3. Milvus CLI . . . . .                | 130 |



# 序言

这是一本关于 LLMs 的书籍。



# 简介



**Part I.**

**LLMs**



# 1. Embedding

在机器学习和自然语言处理中, embedding 是指将高维度的数据 (例如文字、图片、音频) 映射到低维度空间的过程。embedding 向量通常是一个由实数构成的向量, 它将输入的数据表示成一个连续的数值空间中的点。简单来说, embedding 就是一个 N 维的实值向量, 它几乎可以用来表示任何事情, 如文本、音乐、视频等。



## 2. 幻觉

如其他技术一样，即便当前 LLM 在各个领域中有着惊人的表现，但是 LLM 也存在着缺陷和局限。而“幻觉（Hallucination）”就是一种非常常见的缺陷。

### i 幻觉

幻觉是自然语言生成领域的一个术语，是指模型生成了看似合理但实际上并不存在的内容。这些内容可能包含虚构的信息、存在前后矛盾的逻辑、甚至是毫无意义的内容。

幻觉原本是心理学领域的专有名词，用于描述一种特殊类型的知觉体验——在没有外部刺激的情况下，清醒的个体的虚假感觉。

幻觉是一种不真实的、却又非分真实的虚幻感知。模型容易生成流畅但缺乏真实性的内容，这种现象与心理学中的幻觉极为相似，因此在 LLM 领域，我们把 LLM 的这种缺陷称之为 幻觉。

幻觉会严重影响依赖 LLM 的下游业务的表现，导致这些业务在真实场景中无法满足用户需求。大语言模型生成内容的真实性是生成式模型接下来面临的重要科学问题之一。

幻觉分为两类：

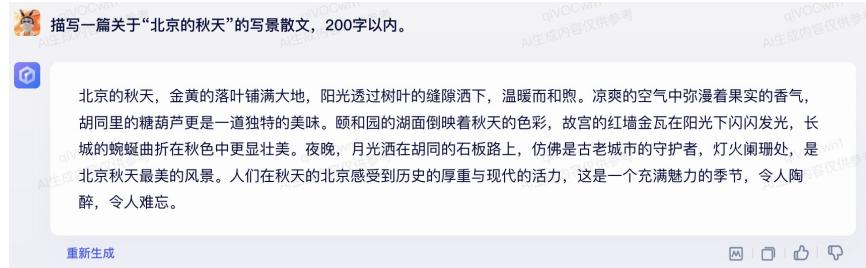
- 内在幻觉（Intrinsic Hallucinations）：生成的内容与输入的源信息冲突。

## 2. 幻觉



图 2.1.: 内在幻觉的例子

- 外在幻觉 (Extrinsic Hallucinations) : 生成了与源信息无关的内容。外在幻觉可能与事实冲突，也可能不冲突。在有些场景下，事实正确的外在幻觉可能会更好，但是事情往往并非总是如此。



### ！重要

幻觉，大模型的阿克琉斯之踵。

更多关于幻觉的详细内容可以参见: [1], [8]。

## 3. Retrieval Augmented Generation



提示

Retrieval Augment Generation: A LLM that uses an external datastore at test time (not at pre-training time) .

在运行时（而非预训练时），使用外部数据的大语言模型称之为基于检索增强的生成式。

### 3.1. RAG 基本概念

根据 *A Survey on Retrieval-Augmented Text Generation* [3] 所述：RAG 是深度学习和传统检索技术（Retrieval Technology）的有机结合，在生成式大模型时代，有着以下优势：

- 知识库和模型分离，知识不以参数的形式存储在模型中，而是明文存储在数据库中，灵活性更高；

### 3. Retrieval Augmented Generation

- 文本生成转变为文本总结，生成结果的可信度更高，同时还降低了文本生成的难度；

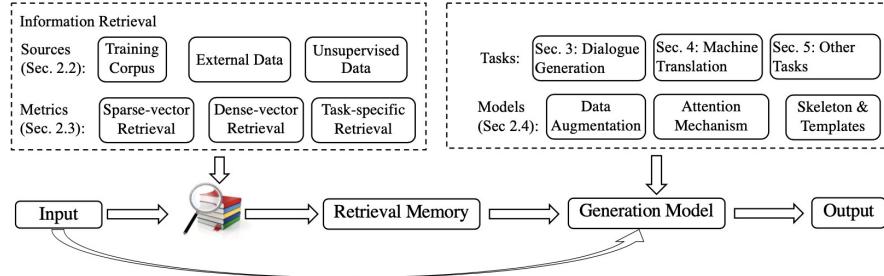


图 3.1.: RATG 综述研究概览

根据图 3.1, RAG 范式有三个重要的组成部分: Retrieval Source, Retrieval Metric, Integration Method。

#### 3.1.1. RAG 的表示方法

传统的文本生成方法可以用如下公式表示:

$$y = f(x) \quad (3.1)$$

其中,  $x$  代表输入的文本 (字符串序列),  $f$  表示模型,  $y$  表示模型输出的文本。

RAG 则可以用如下公式表示:

$$y = f(x, z), z = \{(x^\gamma, y^\gamma)\} \quad (3.2)$$

### 3.1. RAG 基本概念

其中， $x$  代表输入的文本（字符串序列）， $z$  代表知识库， $f$  表示模型， $x^\gamma$  表示作为输入文本  $x$  的检索  $key$ ， $y^\gamma$  是与模型输出相关的知识。

#### 3.1.2. Retrieval Source 类型

- **Training Corpus:** 有标注的训练数据直接作为外部知识。
- **External Data:** 支持提供训练数据之外的外部知识作为检索来源，比如于任务相关的领域数据，实现模型的快速适应。
- **Unsupervised Data:** 前两种知识源都需要一定的人工标注来完善“检索依据-输出”的对齐工作，无监督知识源可以直接支持无标注/对齐的知识作为检索来源。

#### 3.1.3. Retrieval Metrics 类型

- **Sparse-vector Retrieval**（浅层语义）：针对稀疏向量场景的度量方法，比如 TF-IDF, BM25 等。
- **Dense-vector Retrieval**（深层语义）：针对稠密向量的度量方法，比如文本相似度。
- **Task-specific Retrieval:** 在通用的度量场景下，度量得分高并不能代表召回知识准确，因此有学者提出基于特定任务优化的召回度量方法，提高度量的准确率。

#### 3.1.4. Integration Method 类型

- **Data Augmentation:** 直接拼接用户输入文本和知识文本，然后输入文本生成模型。

### 3. Retrieval Augmented Generation

- **Attention Mechanisms:** 引入额外的 Encoder，对用户输入文本和知识文本进行注意力编码后输入文本生成模型。
- **Skeleton Extraction:** 前两种方法都是通过文本向量化的隐式方法完成知识重点片段的抽取，Skeleton Extraction 方法可以显式地完成类似工作。

在 RAG 模式下，AI 应用发生了新的范式变化，从传统的 Pre-training + Fine-tune 的模式转换为了 Pre-training + Prompt 模式。这种模式的转变简化了对于不同任务而言模型训练的工作量，降低了 AI 的开发和使用门槛，同时也使得 Retrieval + Generation 成为可能。

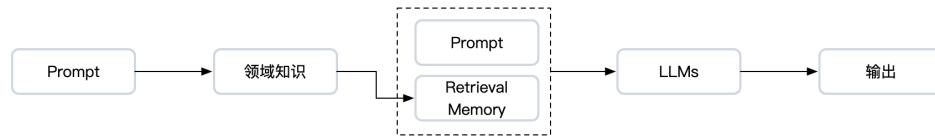


图 3.2.: RAG 基本架构

## 3.2. 为什么要使用 RAG

仅依靠大模型已经可以完成很多任务，Fine-tune 也可以起到补充领域知识的作用，为什么 RAG 仍然如此重要呢？

- **幻觉问题：**尽管大模型的参数量很大，但和人类的所有知识相比，仍然有非常大的差距。所以，大模型在生成内容时，很有可能会捏造事实，导致如章节 2 所述的“幻觉”。因此，对于 LLMs 而言，通过搜索召回相关领域知识来作为特定领域的知识补充是非常必要的。
- **语料更新时效性问题：**大模型的训练数据存在时间截止的问题。尽管可以通过 Fine-tune 来为大模型加入新的知识，但大模型的训练成本和时

### 3.3. 更多內容

间依然是需要面对的严峻难题：通常需要大量的计算资源，时间也难做到天级别更新。在 RAG 模式下，向量数据库和搜索引擎数据的更新都更加容易，这有助于业务数据的实时性。

- **数据泄露问题：**尽管，可以利用 **Fine-tune** 的方式增强 LLM 在特定领域的处理能力。但是，用于 **Fine-tune** 的这些领域知识很可能包含个人或者公司的机密信息，且这些数据很可能通过模型而不经意间泄露出去<sup>1</sup>。RAG 可以通过增加私有数据存储的方式使得用户的数据更加安全。

## 3.3. 更多內容

更详细、深入的内容可以参考如下几篇文章：[3], [4]。

## 3.4. 參考文献

---

<sup>1</sup>Function calling and other API updates



# 4. Agent



提示

LLM 也会有阿克琉斯之踵。

## 4.1. 阿克琉斯之踵

虽然 LLM 非常强大，但在某些方面，与“最简单”的计算机程序的能力相比，LLM 并没有表现的更好，例如在 计算和 搜索这些计算机比较擅长的场景下，LLM 的表现却却很吃力。

---

### 列表 4.1 文心大模型的计算能力测试

---

1

---

列表 4.1 的执行结果如下：

`['4.1 乘以 7.9 等于 31.79。']`

但实际上， $4.1 * 7.9 = 32.39$ ，很明显，文心给出了错误的结果。

#### 4. Agent

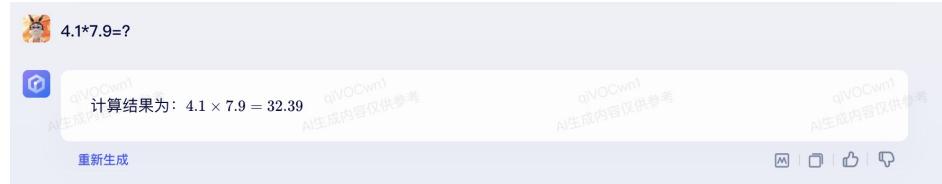


图 4.1.: 一言的计算结果

计算机程序（例如 python 的 `mumexpr` 库）可以轻而易举的处理这种简单的计算，甚至处理比这更复杂的计算也不在话下。但是，面对这些计算，LLM 有时候却显得力不从心。

在章节 3 中，我们提到，使用 RAG 可以解决训练数据的时效性问题、LLM 的幻觉问题、专有数据的安全性问题等问题，但是对于列表 4.1 所示的问题，我们将如何解决？

为了让 LLM 能更好的为我们赋能，我们必须解决这个问题，而接下来要介绍的 **Agent** 就是一种比较好的解决方案。

利用 **Agent**，我们不但可以解决如上提到的 计算的问题，我们还可以解决更多的问题。在我看来，**Agent** 可以解锁 LLM 的能力限制，让 LLM 具备无穷的力量，实现我们难以想象的事情。

## 4.2. 什么是 Agent

在日常生活中，我们解决问题也不是仅依靠我们自己的能力，我们也会使用计算器进行数学计算，我们也会 百度一下以获取相关信息，君子性非异也善假于物也。同样，Agent 使得 LLM 可以像人一样做同样的事情。

## 4.2. 什么是 Agent

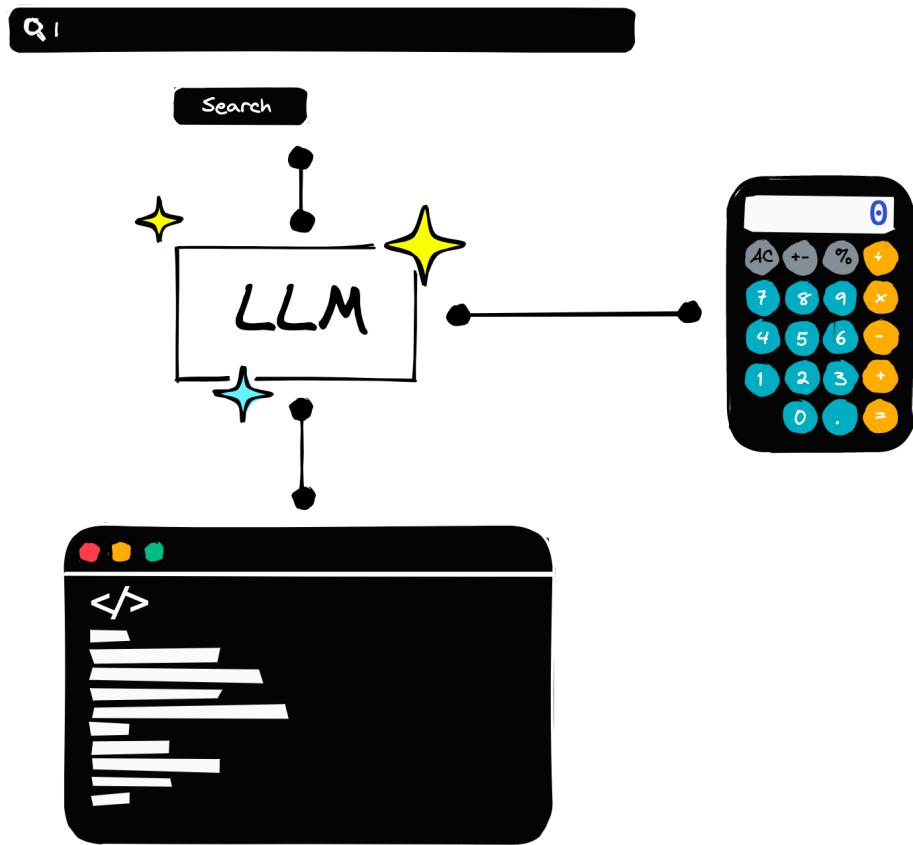


图 4.2.: Agent 就是能够使用各种外部工具的 LLM

#### 4. Agent

从本质上讲，Agent 是一种特殊的 LLM，这种特殊的 LLM 的特殊性在于它可以使用各种外部工具来完成我们给定的操作。

与我们使用外部工具完成任务一样：

1. 我们首先会对任务进行思考
2. 然后判断我们有哪些工具可用
3. 接下来再选择一种我们可用的工具来实施行动
4. 然后我们会观察行动结果以判断如何采取下一步的行动
5. 我们会重复 1-4 这个过程，直到我们认为我们完成了给定的任务

如图 4.3 所示，虽然 Agent 本质上是 LLM，但是其包含的 Thought 和 Tools Set 将 Agent 和 LLM 区别开来，并且这种逐步思考的方式也使得 LLM 可以通过多次推理或多次使用工具来获取更好的结果。

根据 B 站 UP 主发布的视频：作为一款优秀的 Agent，AutoGPT 可以实现自己查询文献、学习文献，并最终完成给定论文题目写作的整个过程，而整个过程中除了最开始需要给 AutoGPT 发布任务外，其他环节则全部由 AutoGPT 自动完成。

### 4.3. ReAct 模式

此处的 ReAct 既不是软件设计模式中的 reactor 模式<sup>1</sup>，也不是 Meta 公司开发的前端开发框架 react<sup>2</sup>，而是 Yao 等人在 [6], [7] 中提出的：把 Reasoning 和 Action 与语言模型结合起来的通用范式，以解决各种语言推理和决策任务。

---

<sup>1</sup>Function calling and other API updates

<sup>2</sup>Guides: Function calling

#### 4.3. ReAct 模式

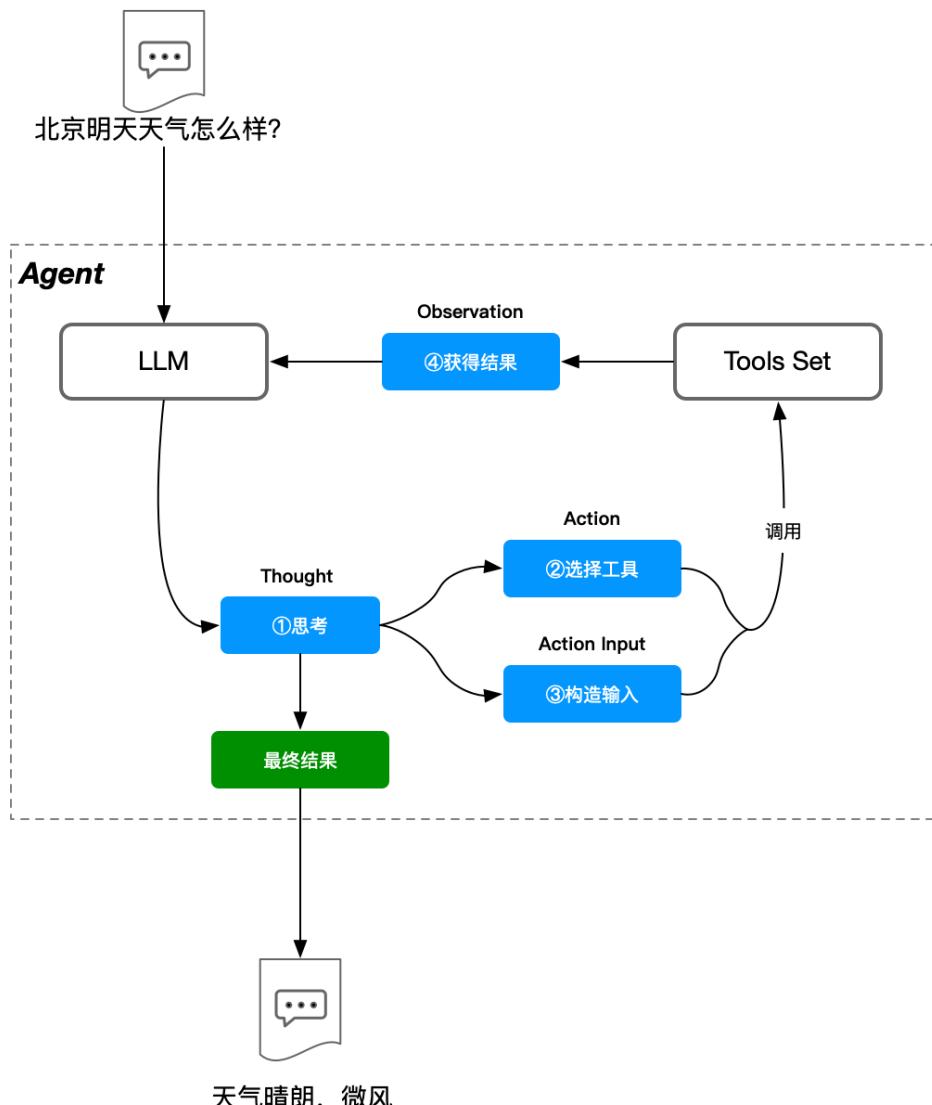


图 4.3.: Agent 流程示意图

#### 4. Agent

ReAct 使 LLM 能够以交错的方式生成 `reasoning traces` 和 `text actions`, ReAct 可以从上下文中进行推理并提取有用的信息来进行后续的 `reasoning` 和 `action`, 从而影响模型的内部状态。正如 [6] 所述, ReAct 将推理阶段和行动阶段进行有效的结合, 进一步提升了 LLM 的性能。

实际上, 和图 4.3 所示的流程是一致的。

##### i 注释

ReAct 也称为 Action Agent, 在 ReAct 模式系下, 代理的下一步动作由之前的输出来决定, 其本质是对 Prompt 进行优化的结果, 一般可以用于规模较小的任务。

### 4.4. PlanAndExecute 模式

如前所述, Action Agent 适用于规模较小的任务。当任务规模较大, 而任务的解决又高度依赖 Agent 来驱动并完成时, Action Agent 就开始变得捉襟见肘。

我们即希望 Agent 能够处理更加复杂的任务, 又希望 Agent 具备较高的稳定性和可靠性。这中既要又要的目标导致 Agent 的提示词变得越来越大, 越来越复杂。

- 为了解决更复杂的任务, 我们需要更多的工具和推理步骤, 这会导致 Agent 的提示词中包含了过多的历史推理信息
- 同时, 为了提升 Agent 的可靠性, 需要不断的优化/增加 Tool 的描述, 以便 LLM 可以选择正确的工具

在这种背景下, PlanAndExecute 模式应运而生。PlanAndExecute 将计划 (`plan`) 与执行 (`execute`) 分离开来。

#### 4.4. PlanAndExecute 模式

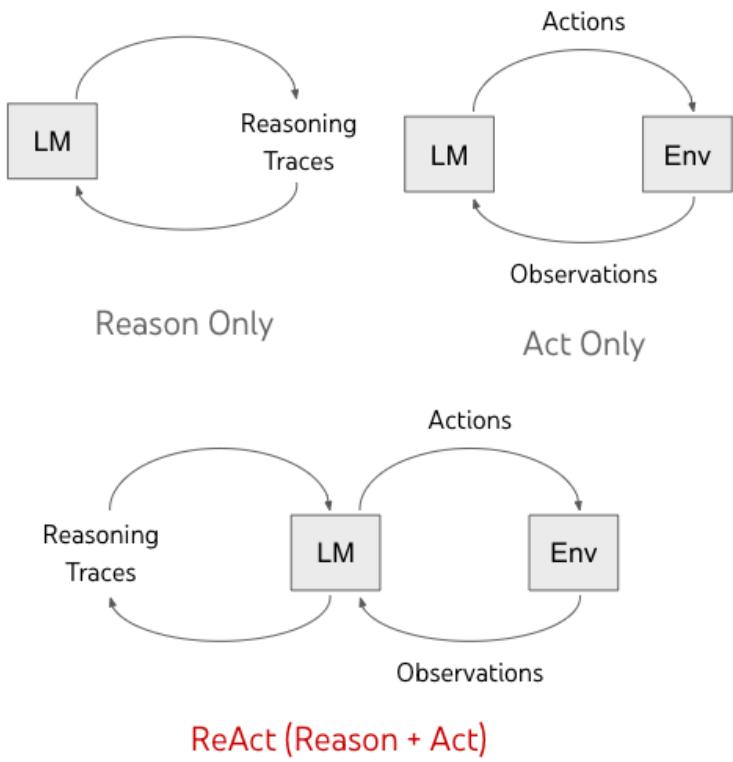


图 4.4.: ReAct 模型

#### 4. Agent

在 PlanAndExecute 模式下，计划由一个 LLM 来驱动生成，而 执行则可以由另外的 Agent 来完成：

- 首先，使用一个 LLM 创建一个用于解决当前请求的、具有明确步骤的计划。
- 然后，使用传统的 Action Agent 来解决每个步骤。

目前，BabyAGI 也采用了类似的模式<sup>3</sup>，更多关于 PlanAndExecute 模式的底层细节，可以参考 [5]。

##### 注释

该模式下，代理将大型任务分解为较小的、可管理的子目标，从而可以高效处理复杂任务。

这种方式可以通过 计划让 LLM 更加“按部就班”，更加可靠。但是其代价在于，这种方法需要更多的 LLM 交互，也必然具有更高的延迟。<sup>4</sup>

## 4.5. 参考文献

---

<sup>3</sup>Using OpenAI functions

<sup>4</sup>A Complete Guide to Data Augmentation

#### 4.5. 参考文献

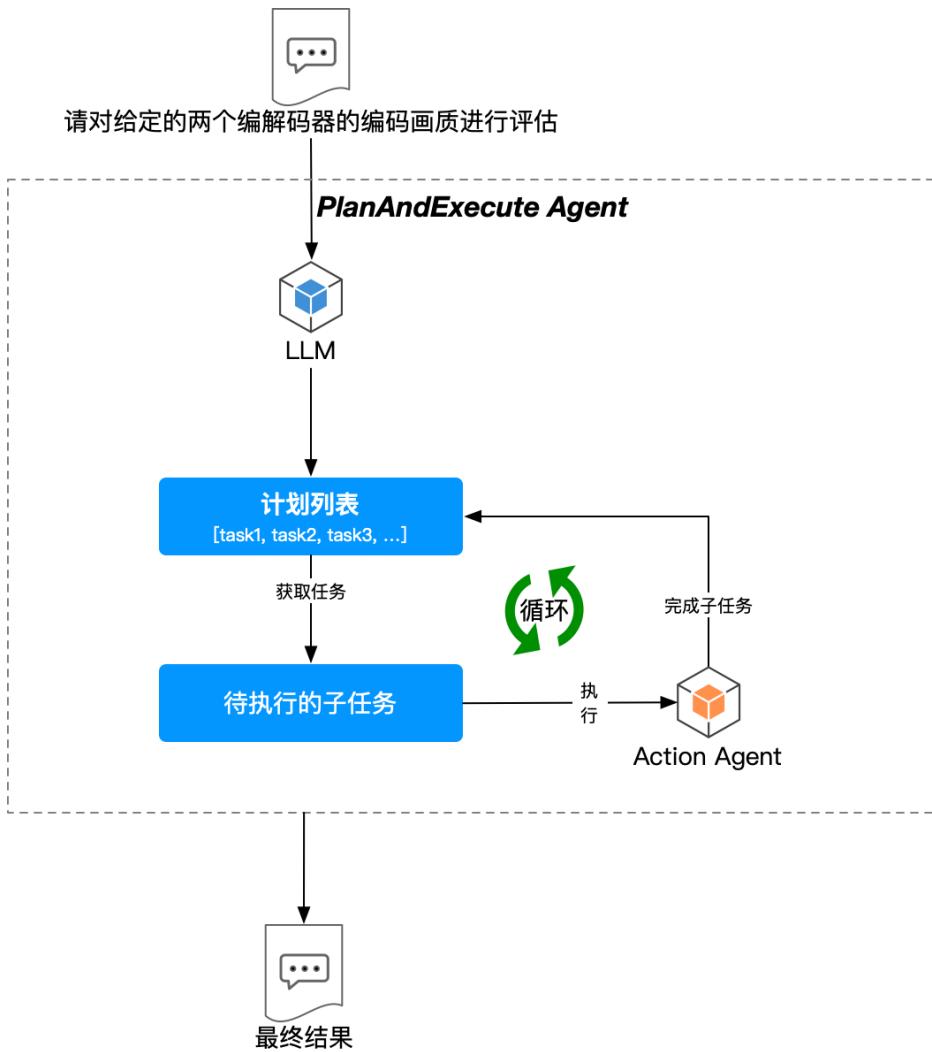


图 4.5.: PlanAndExecute Agent 基本流程



## **Part II.**

# **LangChain**



## 5. LangChain 简介

### 💡 要点提示

我们不想花更多的力气来实现和 LLM 交互的流程，而是更关注于业务逻辑的实现；我们也不想重复编写相似的流程代码，而是可以共享我们的成果，别人只需要键入 `docker pull` 就可以使用我们的成果……如果你有这样的想法，那么 LangChain 正是你的菜～

- LangChain 的目标
- LangChain 的基本概念
- 使用 LangChain 和文心大模型交互

随着大型语言模型（LLM）的引入，自然语言处理已经成为互联网上的热门话题。LangChain 是一个开源 Python 框架，利用 LangChain，开发人员能够非常方便的开发基于大型语言模型的应用程序（AI 原生应用），例如：聊天机器人，摘要，生成式问答……。

LangChain 虽然是一个非常年轻的框架，但又是一个发展速度非常快的框架。自从 2022 年 10 月 25 在 GitHub 第一次提交以来，在 11 个月的时间里，累计发布了 200 多次，累计提交 4000 多次代码。2023 年 3 月，ChatGPT 的 API 因升级降价大受欢迎，LangChain 的使用也随之爆炸式增长。

## 5. LangChain 简介



图 5.1.: LangChain 代码提交趋势

之后，LangChain 在没有任何收入也没有任何明显的创收计划的情况下，获得了 1000 万美元的种子轮融资和 2000-2500 万美元的 A 轮融资，估值达到 2 亿美元左右。<sup>1</sup>

作为一个年轻而又活力的框架，LangChain 正在彻底改变工业和技术，改变我们与技术的每一次互动。

### 💡 提示

可以使用章节 5.3 中介绍的 Langflow 来对 LangChain 进行可视化操作。

## 5.1. LangChain 的目标

不同的大语言模型都有各自的优势，我们可能会用 A 模型来进行自然语言理解，然后用 B 模型进行逻辑推理并获取结果……此时，如果使用大语言模型各自提供的 API 来和模型交互，那么就会存在非常多的重复工作。

虽然大语言模型有很多，但是和大语言模型的交互流程又是非常类似（如图 5.2 所示），如果每次和模型交互都需要重复如上的步骤，那听起来也是一件

<sup>1</sup>LangChain 估值

### 5.1. LangChain 的目标

非常繁琐的事情。对于相同的提示词，我们不想每次都 `ctr+c`、`ctr+v`，这真是一件非常可怕的事情。

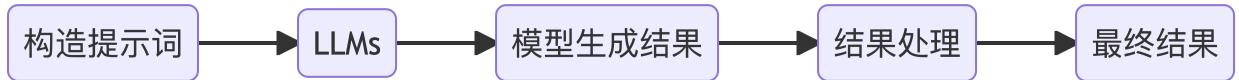


图 5.2.: 和模型交互的流程

和 FFmpeg 对视频的处理一样，FFmpeg 提供的 `filtergraph`<sup>2</sup>机制大大增强了其音视频的处理能力，奠定其在视音频领域的地位。`filtergraph` 可以将不同的音视频处理能力以链条的形式组合起来，不但简化了音视频的处理流程，更让 FFmpeg 可以实现复杂的音视频处理。

同理，和 LLMs 的单次交互并不会形成什么惊人的能量，而如果可以使用类似 `filtergraph` 的机制，将与 LLMs 的多次交互整合起来，那么其所释放的能量将是无穷的。

而 LangChain 就是为了解决如上的问题而产生的。LangChain 可以提供给我们的最主要的价值如下<sup>3</sup>：

- 组件化：LangChain 对与 LLMs 交互的流程进行了统一的抽象，同时也提供了不同 LLMs 的实现。这极大的提升了我们使用 LLMs 的效率。
- 序列化：LangChain 提供的序列化的能力，可以将提示词、`chain` 等以文件的形式而不是以代码的形式进行存储，这样可以极大的方便我们共享提示词，并对 提示词进行版本管理。<sup>4</sup>

<sup>2</sup>Function calling and other API updates

<sup>3</sup>Guides: Function calling

<sup>4</sup>Using OpenAI functions

## 5. *LangChain* 简介

- 丰富的 chains 套件: LangChain 提供了丰富、用于完成特定目的、开箱即用的 chains 套件, 例如用于总结文档的 `StuffDocumentsChain` 和 `MapReduceDocumentsChain`, 这些套件将会降低我们使用 LLMs 的门槛。

更具体的, LangChain 可以在如下的 6 大方向上给我们提供非常大的便利:

1. **LLMs & Prompt**: LangChain 提供了目前市面上几乎所有 LLM 的通用接口, 同时还提供了 提示词的管理和优化能力, 同时也提供了非常多的相关适用工具, 以方便开发人员利用 LangChain 与 LLMs 进行交互。
2. **Chains**: LangChain 把 提示词、大语言模型、结果解析封装成 `Chain`, 并提供标准的接口, 以便允许不同的 `Chain` 形成交互序列, 为 AI 原生应用提供了端到端的 `Chain`。
3. **Data Augmented Generation**<sup>5</sup>: 数据增强生成式是一种解决预训练语料数据无法及时更新而带来的回答内容陈旧的方式。LangChain 提供了支持 数据增强生成式的 `Chain`, 在使用时, 这些 `Chain` 会首先与外部数据源进行交互以获得对应数据, 然后再利用获得的数据与 LLMs 进行交互。典型的应用场景如: 基于特定数据源的问答机器人。
4. **Agent**: 对于一个任务, 代理主要涉及让 LLMs 来对任务进行拆分、执行该行动、并观察执行结果, 代理会重复执行这个过程, 直到该任务完成为止。LangChain 为 代理提供了标准接口, 可供选择的代理, 以及一些端到端的 代理的示例。
5. **Memory**: 内存指的是 chain 或 agent 调用之间的状态持久化。LangChain 为 内存提供了标准接口, 并提供了一系列的 内存实现。
6. **Evaluation**: LangChain 还提供了非常多的评估能力以允许我们可以更方便的对 LLMs 进行评估。

---

<sup>5</sup>A Complete Guide to Data Augmentation

## 5.2. LangChain 的基本概念

### 💡 LangChain 安装

LangChain 的安装可以参见附录 B。

## 5.2. LangChain 的基本概念

使用 LLMs 和使用电脑一样，需要一些基本的架构体系。LangChain 把整体架构体系分为两部分：输入/输出系统，大语言模型。其中，输入部分为 **Prompt** 相关组件，输出为 **Output Parser** 相关组件。具体参见图 5.3。

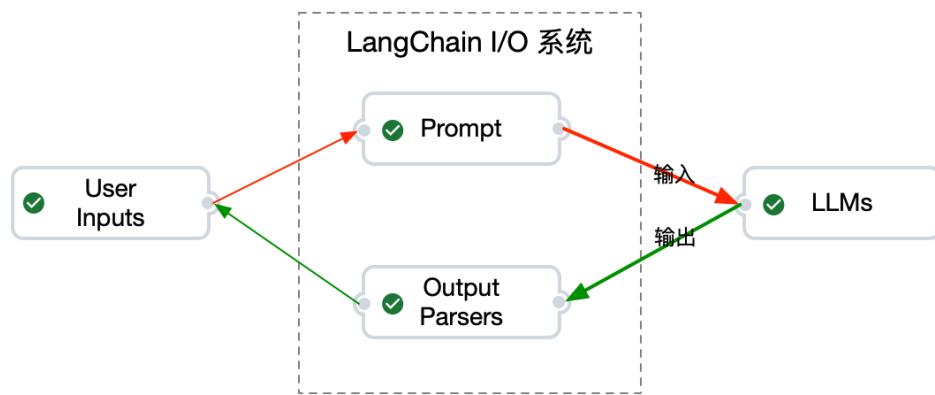


图 5.3.: LangChain I/O

LangChain 提供了与 LLMs 交互的通用构件：

- **Prompts**: 提示词模版，提示词动态选择，提示词序列化。
- **LLMs**: 与 LLM 交互的通用接口。
- **Output Parsers**: 对模型的输出信息进行解析，以输出符合特定格式的响应。

## 5. LangChain 简介

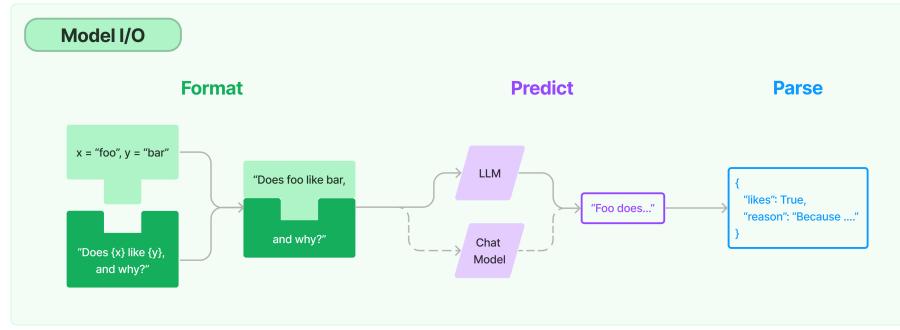


图 5.4.: LangChain I/O 示例

### 5.2.1. Prompt Templates

提示词模版为不同的提示词提供预定义格式。就好像目前超市售卖的洗净切好、配好相关配菜源材料的预制菜一样，提示词模版可以简化我们和 LLMs 交互的效率。

模版会包含：指令，少量的样本示例，相关的上下文信息。如章节 5.2.2 所述，LLMs 会分为 大语言模型和 聊天模型两种类型，因此，LangChain 提供了两种类型的提示词模版：prompt template、chat prompt template。

- `prompt template`: 提供字符串格式的提示词。
- `chat prompt template`: 提供聊天消息格式的提示词。

### 5.2.2. LLMs

LangChain 提供了两种模型的通用接口：

## 5.2. LangChain 的基本概念

列表 5.1 PromptTemplte 示例

```
from langchain import PromptTemplate

prompt_template = PromptTemplate.from_template(
    "请以轻松欢快的语气写一篇描写 {topic} 的文章，字数不超过 {count} 字。"
)

res = prompt_template.format(topic="北京的秋天", count="100")

print(res)
# 请以轻松欢快的语气写一篇描写 北京的秋天 的文章，字数不超过 100 字。
```

- **LLMs**: 模型以字符串格式的提示词作为输入，并返回字符串格式的结果。
- **Chat models**: 其背后也是由某种 LLM 来支撑，但是以聊天消息列表格式的提示词作为输入，并返回聊天消息格式的结果。

### i LLMs & Chat Models

LLM 和聊天模式之间的区别虽然很微妙，但是却完全不同。  
LangChain 中的 LLM 指的是纯文本 I/O 的模型，其包装的 API 将字符串提示作为输入，并输出字符串。OpenAI 的 GPT-3 就是 LLM。  
聊天模型通常由 LLM 支持，但专门针对对话进行了调整，其 API 采用聊天消息列表作为输入，而不是单个字符串。通常，这些消息都标有角色（例如，“System”，“AI”，“Human”）。聊天模型会返回一条 AI 聊天消息作为输出。OpenAI 的 GPT-4，Anthropic 的 Claude，百度的 Ernie-Bot 都是聊天模型。

## 5. LangChain 简介

在 LangChain 中，LLM 和聊天模式两者都实现了 `BaseLanguageModel` 接口，因此一般情况下，这两种模型可以混用。例如，两种模型都实现了常见的方法 `predict()` 和 `predict_messages()`。`predict()` 接受字符串并返回字符串，`predict_messages()` 接受消息并返回消息。

接下来，我们将 `Prompt` 和 `LLM` 整合起来，实现和大语言模型交互。

由于文心聊天模型对 `message` 角色和条数有限制<sup>6</sup> <sup>7</sup>，因此我们需要对 提示词做一些修改。

### 文心 4.0

在 LangChain 中，要使用文心 4.0 模型，可以在初始化 LLM 时设置 `model_name` 参数为 `ERNIE-Bot-4`。

```
llm = ErnieBotChat(model_name="ERNIE-Bot-4")
```

### 5.2.3. Output Parsers

大语言模型一般会输出文本内容作为响应，当然更高级的大语言模型（例如文心大模型）还可以输出图片、视频作为响应。但是，很多时候，我们希望可以获得更结构化的信息，而不仅仅是回复一串字符串文本。

我们可以使用 提示词 工程来提示 LLMs 输出特定的格式，如列表 5.7 所示：

但是，使用 LangChain 提供的 `Output Parsers` 能力，会更加的方便。

---

<sup>6</sup>ERNIE-Bot-turbo

<sup>7</sup>百度智能云千帆大模型平台

## 5.2. LangChain 的基本概念

### ⚠ 警告

由于文心大模型的指令遵循能力还有进一步提升的空间，因此这里的演示可能需要进行一些额外的操作，例如需要对模型返回的内容进行一些简单的字符串替换。

2023 年 10 月 17 日，百度世界大会上发布了文心 4.0，我们发现文心 4.0 在 ICL、指令遵循、推理能力上都有比较大的提升。

在 LangChain 中，要使用文心 4.0 模型，可以在初始化 LLM 时设置 `model_name` 参数为 `ERNIE-Bot-4`。

```
llm = ErnieBotChat(model_name="ERNIE-Bot-4")
```

### 5.2.4. LLMChain

虽然一台独立的计算机也能实现很强大的功能，但是通过网络将更多的计算机链接起来，可能发挥出更大的性能。同样的，单独使用 LLMs 已经可以实现强大的功能，但是如果可以将更多次的交互有效的链接起来，则能发挥 LLMs 更大的能量。为了实现这个目标，LangChain 提供了 `Chain` 的概念，以实现对不同组件的一系列调用。

在 LangChain 中，提示词、LLM、输出解析这三者构成了 `Chain`，而不同的 `Chain` 则可以通过一定的方式链接起来，以实现强大的功能。具体如图 5.5 所示。

利用 `Chain` 的概念，我们可以对列表 5.8 的代码进行重构，

## 5. LangChain 简介

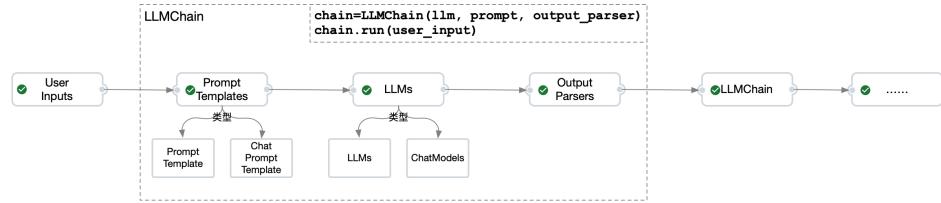


图 5.5.: LangChain 中 Chain 的概念

## 5.3. Langflow

Langflow 是 LangChain 的非官方的 UI。使用 Langflow，我们可以更简便的以可视化的方式来体验 LangChain 并为基于 LangChain 的大语言应用提供原型设计能力。

想要深入了解 Langflow，可以阅读 Langflow 的官方文档。

## 5.4. LangChain 的学习资料

- LangChain 官方文档: [https://python.langchain.com/docs/get\\_started](https://python.langchain.com/docs/get_started)
- LangChain 的典型应用场景: [https://python.langchain.com/docs/use\\_cases](https://python.langchain.com/docs/use_cases)
- LangChain 目前集成的能力: <https://python.langchain.com/docs/integrations>
- LangChain AI Handbook: <https://www.pinecone.io/learn/series/langchain/>
- LangChain Dart: <https://langchaindart.com/#/>
- 百度智能云千帆大模型平台: <https://cloud.baidu.com/product/wenxinworkshop>

#### 5.4. LangChain 的学习资料

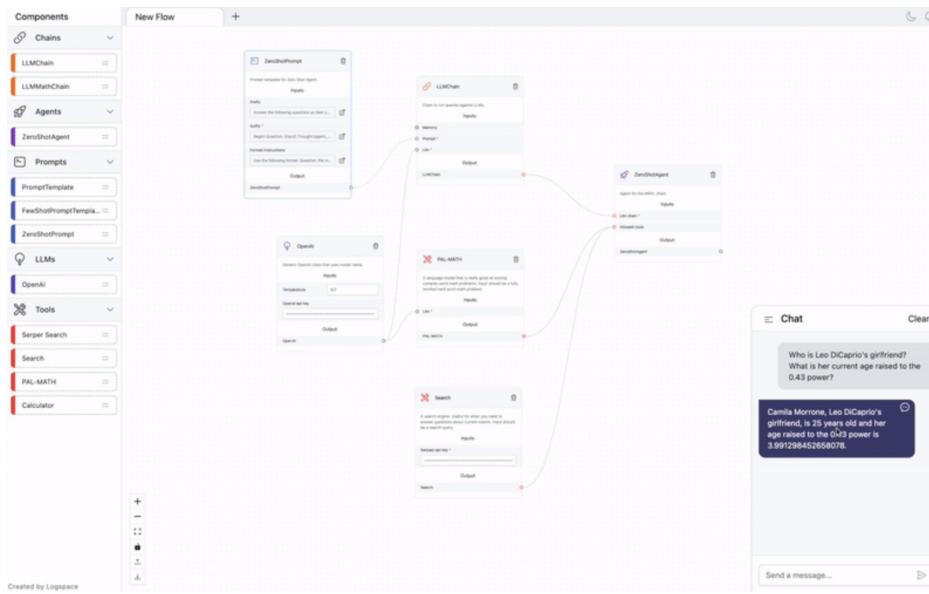


图 5.6.: Langflow 示例

## 5. *LangChain* 简介

- Langflow 官方文档: <https://docs.langflow.org/>

## 5.5. 参考文献

---

**列表 5.2 ChatPromptTemplte 示例**

---

```
from langchain.prompts import ChatPromptTemplate

template = ChatPromptTemplate.from_messages([
    ("system", " 你是一个能力非凡的人工智能机器人，你的名字是 {name}。"),
    ("human", " 你好！"),
    ("ai", " 你好 ~"),
    ("human", "{user_input}"),
])

messages = template.format_messages(
    name=" 小明",
    user_input=" 你是谁？"
)

print(messages)

# [SystemMessage(content='你是一个能力非凡的人工智能机器人，你的名字是 小明。',
#                 additional_kwargs={}),
#  HumanMessage(content='你好！', additional_kwargs={}, example=False),
#  AIMessage(content='你好 ~', additional_kwargs={}, example=False),
#  HumanMessage(content='你是谁？', additional_kwargs={}, example=False)]
```

---

## 5. LangChain 简介

---

### 列表 5.3 LLM 模式

---

```
class OpenAI(BaseOpenAI):
    # ...

class BaseOpenAI(BaseLLM):
    # ...

class BaseLLM(BaseLanguageModel[str], ABC):
    # ...
```

---

---

### 列表 5.4 聊天模型

---

```
class ErnieBotChat(BaseChatModel):
    # ...

class BaseChatModel(BaseLanguageModel[BaseMessageChunk], ABC):
    # ...
```

---

## 5.5. 参考文献

---

列表 5.5 LLM 模型示例

---

```
from langchain import PromptTemplate
from langchain.llms import OpenAI

prompt_template = PromptTemplate.from_template(
    "请以轻松欢快的语气写一篇描写 {topic} 的文章，字数不超过 {count} 字。"
)
llm = OpenAI()

prompt = prompt_template.format(topic="北京的秋天", count="100")
res = llm.predict(prompt)
print(res)

# 秋天来到了北京，一片金黄色的枫叶，漫山遍野。
# 湖面上的微风，吹起柔和的秋意，空气中弥漫着淡淡的枫香。
# 这时，每一个角落都洋溢着秋日的温馨，令人心旷神怡。
# 古老的长城上披着红叶，熙熙攘攘的人群中，也多了几分热闹与欢畅，这就是北京的秋天
```

---

## 5. LangChain 简介

---

### 列表 5.6 聊天模型示例

---

```
from langchain.chat_models import ErnieBotChat
from langchain.prompts import ChatPromptTemplate

template = ChatPromptTemplate.from_messages([
    ("user", " 你是一个能力非凡的人工智能机器人，你的名字是 {name}。"),
    ("assistant", " 你好 ~"),
    ("user", "{user_input}"),
])
chat = ErnieBotChat()

messages = template.format_messages(
    name=" 小明",
    user_input=" 你是谁? "
)

res = chat.predict_messages(messages)
print(res)

# content='我是你的新朋友小明，一个拥有先进人工智能技术的人工智能机器人。'
# additional_kwargs={} example=False
```

---

## 5.5. 参考文献

---

列表 5.7 使用提示词工程来格式化输出内容

---

```
from langchain.chat_models import ErnieBotChat

from langchain.prompts import ChatPromptTemplate

template = ChatPromptTemplate.from_messages([
    ("user", " 你是一个能力非凡的人工智能机器人，你的名字是 {name}。"),
    ("assistant", " 你好 ~"),
    ("user", "{user_input}"),
])

chat = ErnieBotChat()

messages = template.format_messages(
    name=" 小明",
    user_input=" 请给出 10 个表示快乐的成语，并输出为 JSON 格式"
)

res = chat.predict_messages(messages)

print(res)

# content='```\n        " 乐不可支",\n        "\n        \" 喜从天降\", 45\n        "\n        \" 笑逐颜开\", \n        "\n        \" 手舞足蹈\", \n        "#\n        ..... \n        "\n        \" 弹冠相庆\"\n    ]\n```\n'

# additional_kwargs={} example=False
```

---

## 5. LangChain 简介

---

列表 5.8 使用 Output Parser 解析 LLM 结果

---

```
from langchain.chat_models import ErnieBotChat
from langchain.prompts import ChatPromptTemplate
from langchain.output_parsers import CommaSeparatedListOutputParser

template = ChatPromptTemplate.from_messages([
    ("user", " 你是一个能力非凡的人工智能机器人，你的名字是 {name}。"),
    ("assistant", " 你好 ~"),
    ("user", "{user_input}"),
])
chat = ErnieBotChat()

messages = template.format_messages(
    name=" 小明",
    user_input=" 请仅给 5 个表示快乐的成语并以 , 分隔，除了成语外不要输出任何其他"
)

res = chat.predict_messages(messages)
print(res)
# content='欢呼雀跃，手舞足蹈，笑逐颜开，心花怒放，喜笑颜开' additional_kwargs={}

output_parser = CommaSeparatedListOutputParser()
res = output_parser.parse(res.content.replace(' ', ', '))
print(res)
# ['欢呼雀跃', '手舞足蹈', '笑逐颜开', '心花怒放', '喜笑颜开']
```

---

---

列表 5.9 使用 chain 与文心大模型进行交互

---

```
from langchain.chat_models import ErnieBotChat
from langchain.prompts import ChatPromptTemplate

from langchain.output_parsers import CommaSeparatedListOutputParser
from langchain.chains import LLMChain

template = ChatPromptTemplate.from_messages([
    ("user", " 你是一个能力非凡的人工智能机器人，你的名字是 {name}。"),
    ("assistant", " 你好 ~"),
    ("user", "{user_input}"),
])

chat = ErnieBotChat()

chain = LLMChain(llm=chat, prompt=template, output_parser=CommaSeparatedListOutputParser())

res = chain.run(name=" 小明", user_input=" 请仅给 5 个表示快乐的成语并以 , 分隔，除了成语外不要有其他内容")
print(res)
# ['以下是五个表示快乐的成语: \n\n1. 喜出望外\n2. 乐不可支\n3. 心花怒放\n4. 满心欢喜\n5. 手舞足蹈']
```

---



## **6. LangChain 序列化**

### **6.1. 序列化**

### **6.2. LangChain-Hub**



## 7. LangChain Retrieval

在章节 3 中，我们介绍了基于检索增强的生成式技术，这一章，我们重点介绍如何使用 LangChain 实现 RAG。

无论是简单的 RAG 应用，还是复杂的 RGA 应用，LangChain 都为我们提供了相应的构建能力。在 LangChain 中，RAG 的整个过程涉及到如图 7.1 的模块和步骤：

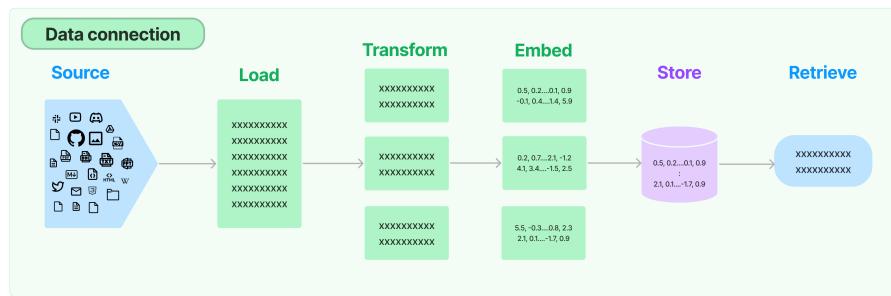


图 7.1.: LangChain 中 RAG 的关键模块

## 7. LangChain Retrieval

### 7.1. Document loaders

LangChain 提供了100 多种不同的文档加载器，并与该领域的其他主要供应商（如 AirByte、Unstructured）进行了集成，从而可以从任何地方（私有 s3 存储、网站）加载任何类型的文档（HTML、PDF、代码）。

文档加载器提供了一个 `load()` 方法来从指定的加载源加载文档数据。文档加载器还提供了一个 `lazy_load()` 方法来实现现“延迟加载”，以避免一次将太多的数据加载到内存之中。

---

#### 列表 7.1 加载远程网页

---

```
from langchain.document_loaders.recursive_url_loader import RecursiveUrlLoader

URL_ROOT = "https://wangwei1237.github.io/"
loader = RecursiveUrlLoader(url=URL, max_depth=2)
docs = loader.load()

print(len(docs))

URLS = []
for doc in docs:
    url = doc.metadata["source"]
    title = doc.metadata["title"]
    print(url, "->", title)
```

---

**⚠ 警告**

`RecursiveUrlLoader()` 对中文的抓取看起来不是非常友好，中文内容显示成了乱码。可以使用列表 7.2 所示的方法来解决中文乱码的问题，不过这种方式的缺点是需要 `load()` 两次。更好的方式后续再思考。

## 7.2. Document transformers

检索的一个关键部分是只获取文档的相关部分而非获取全部文档。为了为最终的检索提供最好的文档，我们需要对文档进行很多的转换，这里的主要方法之一是将一个大文档进行拆分。`LangChain` 提供了多种不同的拆分算法，并且还针对特定文档类型（代码、标记等）的拆分提供对应的优化逻辑。

文档加载后，我们通常会对文档进行一系列的转换，以更好地适应我们的应用程序。最简单的文档转换的场景就是文档拆分成，以便可以满足模型的上下文窗口（不同模型的每次交互的最大 token 数可能不同）。

尽管文档拆分听起来很简单，但实际应用中却有很多潜在的复杂性。理想情况下，我们希望将语义相关的文本片段放在一起。“语义相关”的含义会取决于文本的类型，例如：

- 对于代码文件而言，我们需要将一个函数置于一个完整的拆分块中；
- 普通的文本而言，可能需要将一个段落置于一个完整的拆分块中；
- .....

我们利用 `RecursiveCharacterTextSplitter` 对列表 7.2 的文档进行拆分。

## 7. LangChain Retrieval

LangChain 也可以对不同的编程语言进行拆分，例如 cpp, go, markdown, .....，具体支持的语言可以参见列表 7.4。

### 7.3. Text embedding models

检索的另一个关键部分是为文档创建其向量（embedding）表示。Embedding 捕获文本的语义信息，使我们能够快速、高效地查找其他相似的文本片段。LangChain 集成了 25 种不同的 embedding 供应商和方法，我们可以根据我们的具体需求从中进行选择。LangChain 还提供了一个标准接口，允许我们可以便捷的在不同的 embedding 之间进行交换。

在 LangChain 中，`Embeddings` 类是用于文本向量模型的接口。目前，有很多的向量模型供应商，例如：OpenAI, Cohere, Hugging Face, .....`Embeddings` 类的目的就是为所有这些向量模型提供统一的、标准的接口。

`Embeddings` 类可以为一段文本创建对应的向量表示，从而允许我们可以在向量空间中去考虑文本。在向量空间中，我们还可以执行语义搜索，从而允许我们在向量空间中检索最相似的文本片段。

因为不同的向量模型供应商对文档和查询采用了不同的向量方法，`Embeddings` 提供了两个方法：

- `embed_documents()`: 用于文档向量化
- `embed_query()`: 用于查询向量化

**⚠ 使用 QianfanEmbeddingsEndpoint 的注意事项**

LangChain 在 0.0.300 版本之后才支持 `QianfanEmbeddingsEndpoint`，并且 `QianfanEmbeddingsEndpoint` 还依赖 `qianfan` python 库的支持。因此，在使用 `QianfanEmbeddingsEndpoint` 之前，需要：

- 升级 LangChain 的版本: `pip install -U langchain`。
- 安装 `qianfan` 库: `pip install qianfan`。

## 7.4. Vector stores

为文档创建 embedding 之后，需要对其进行存储并实现对这些 embedding 的有效搜索，此时我们需要向量数据库的支持。LangChain 集成了 50 多种不同的向量数据库，还提供了一个标准接口，允许我们轻松的在不同的向量存储之间进行切换。

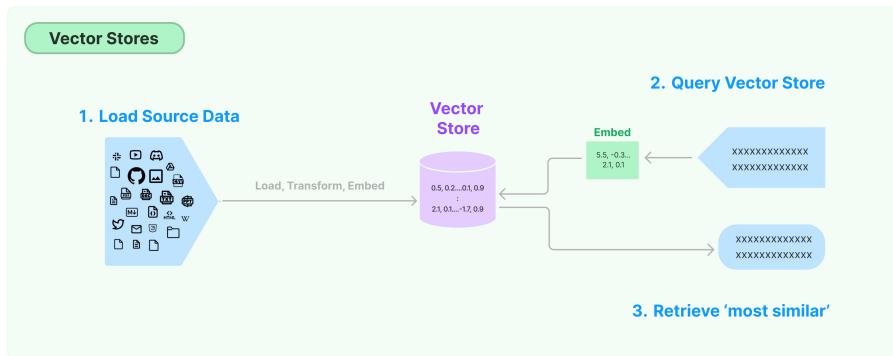


图 7.2.: 向量数据库检索的基本流程

## 7. LangChain Retrieval

这里，我们使用 Milvus 向量数据库来进行相关的演示。Milvus 安装和使用方式可以参见：附录 C。

利用 Milvus 对列表 7.6 进行优化：

列表 7.7 的运行结果中，之所以会有两条重复的结果，是因为在执行文档向量化的时候，执行了两遍。在初始化 Milvus 实例时，如果只是查询操作，可以使用如下的方式：

`Milvus.from_documents` 会创建一个名为 `LangChainCollection` 的 `Collection`。可以使用 `milvus_cli` 工具来查看该 `Collection` 的信息，也可以使用 Milvus 提供的 http 端口来查看相关信息：

```
http://127.0.0.1:8081/v1/vector/collections/describe?collectionName=LangChain
```

### ⚠ 警告

使用千帆进行 Embedding 时，每次 Embedding 的 token 是有长度限制的，目前的最大限制是 384 个 token。因此，我们在使用 `RecursiveCharacterTextSplitter` 进行文档拆分的时候要特别注意拆分后文档的长度。

```
qianfan.errors.APIError: api return error,  
code: 336003,  
msg: embeddings max tokens per batch size is 384
```

在使用时，为了方便，我们可以把 embedding 和 query 拆分为两个部分：

- 先将数据源进行向量化，然后存储到 Milvus 中
- 检索的时候，直接从 Milvus 中检索相关信息

对列表 7.6 的代码进行优化：

检索相似内容的代码可以简化为：

 警告

因为千帆向量化的 API 有 QPS 限制，因此，在使用千帆进行 embedding 时尽量控制一下 QPS。

## 7.5. Retrievers

检索是 LangChain 花费精力最大的环节，LangChain 提供了许多不同的检索算法，LangChain 不但支持简单的语义检索，而且还增加了很多算法以提高语义检索的性能。

一旦我们准备好了相关的数据，并且将这些数据存储到向量数据库（例如 Milvus），我们就可以配置一个 `chain`，并在提示词中包含这些相关数据，以便 LLM 在回答我们的问题时可以利用这些数据作为参考。

对于参考外部数据源的 QA 而言，LangChain 提供了 4 种 `chain: stuff`, `map_reduce`, `refine`, `map_rerank`。`stuff chain` 把文档作为整体包含到提示词中，这适用于小型文档。由于大多数 LLM 对提示次可以包含的 token 最大数量存在限制，因此建议使用其他三种类型的 `chain`。对于非 `stuff chain`，LangChain 将输入文档分割成更小的部分，并以不同的方式将它们提供给 LLM。这 4 种 `chain` 的具体信息和区别可以参见：[docs/modules/chains/document](#)。

我们利用 `QAWithSourcesChain` 对列表 7.10 进行优化，以实现一个完整的利用外部数据源的 **Retrieval Augment Generation**（需要配合列表 7.9）。

## 7. LangChain Retrieval

列表 7.11 的运行结果如下，结果包括 `intermediate_steps` 和 `output_text`:

- `intermediate_steps` 表示搜索过程中所指的文档
- `output_text` 表示是问题的最终答案

4

```
{'intermediate_steps':  
    [  
        '根据提供的上下文信息，回答问题：\n\n「度知了」是一个在线问答平台，使用指南是由作者严丽编写的。  
        '根据提供的上下文信息，「度知了」是一个在线问答平台，使用指南是由作者严丽编写的。  
        '根据提供的上下文信息，「度知了」是一个在线问答平台，提供视频画质评测服务。  
        "Based on the new context, the existing answer is still accurate. The  
    ],  
    'output_text': "Based on the new context, the existing answer is still accu  
}
```

为了显示 RAG 的优点，我们可以利用列表 5.9 所示的代码向 LLM 问同样的问题：

```
res = chain.run(name=" 小明", user_input=" 什么是度知了?")  
print(res)
```

```
# ['度知了是一款智能问答产品，它能够理解并回答问题，提供信息和建议，主要应用在搜索、
```

### 7.6. RetrievalQA

## 7.6. RetrievalQA

使用 RetrievalQA 也可以实现列表 7.11 同样的功能，并且代码整体会更简洁。

1. 使用 Milvus 初始化向量检索器
  2. 因为文心对 MessageList 的限制，所以此处要重写 Prompt，否则执行时会报 Message 类型错误。具体提示词的修改可以参考：列表 7.13。
  3. 使用向量检索器初始化 RetrievalQA 实例
  4. 执行 RAG 检索并提炼最终结果
- 
1. 修改 SystemMessagePromptTemplate 为 HumanMessagePromptTemplate。
  2. 增加一条 AIMessagePromptTemplate 消息。

列表 7.12 的运行结果如下所示：

度知了是一款视频画质评测服务，基于 ITU 标准，依托自研的 10+ 项专利技术，支持多端（PC、Android、iOS）

## 7. LangChain Retrieval

---

列表 7.2 解决中文乱码的方法

---

```
from langchain.document_loaders import WebBaseLoader
from langchain.document_loaders.recursive_url_loader import RecursiveUrlLoader

URL_ROOT = "https://wangwei1237.github.io/"
loader = RecursiveUrlLoader(url=URL_ROOT, max_depth=2)
docs = loader.load()

print(len(docs))

URLS = []
for doc in docs:
    url = doc.metadata["source"]
    URLS.append(url)

loader = WebBaseLoader(URLS)
docs = loader.load()

print(len(docs))

for doc in docs:
    url = doc.metadata["source"]
    title = doc.metadata["title"]

    print(url, "->", title)
```

---

---

**列表 7.3 使用 RecursiveCharacterTextSplitter 拆分文档**

---

```
# ...

# ...

text_splitter = RecursiveCharacterTextSplitter(
    chunk_size = 1000,
    chunk_overlap = 20,
    length_function = len,
    add_start_index = True,
)

for doc in docs:
    url = doc.metadata["source"]
    title = doc.metadata["title"]
    print(url, "-->", title)

    texts = text_splitter.create_documents([doc.page_content])
    print(texts)
```

---

## 7. LangChain Retrieval

---

列表 7.4 LangChain 支持拆分的语言类型

---

```
from langchain.text_splitter import Language

[e.value for e in Language]

# ['cpp',
#  'go',
#  'java',
#  'js',
#  'php',
#  'proto',
#  'python',
#  'rst',
#  'ruby',
#  'rust',
#  'scala',
#  'swift',
#  'markdown',
#  'latex',
#  'html',
#  'sol']
```

---

## 7.6. RetrievalQA

---

列表 7.5 使用文心大模型的 Embedding-V1 查询向量化

---

```
from langchain.embeddings import QianfanEmbeddingsEndpoint

embeddings = QianfanEmbeddingsEndpoint()
query_result = embeddings.embed_query(" 你是谁? ")

print(query_result)
print(len(query_result))

# [0.02949424833059311, -0.054236963391304016, -0.01735987327992916,
#  0.06794580817222595, -0.00020318820315878838, 0.04264984279870987,
#  -0.0661700889468193, .....
#  ....]
#
# 384
```

---

## 7. LangChain Retrieval

---

列表 7.6 使用文心大模型的 Embedding-V1 文档向量化

---

```
from langchain.embeddings import QianfanEmbeddingsEndpoint

embeddings = QianfanEmbeddingsEndpoint()
docs_result = embeddings.embed_documents([
    " 你谁谁? ",
    " 我是百度的智能助手，小度"
])
print(len(docs_result), ":" , len(docs_result[0]))

# 2 : 384
```

---

## 7.6. RetrievalQA

---

列表 7.7 使用 Milvus 存储千帆 Embedding-V1 的结果

---

```
from langchain.document_loaders import WebBaseLoader
from langchain.embeddings import QianfanEmbeddingsEndpoint
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.vectorstores import Milvus

url = 'https://wangwei1237.github.io/2023/02/13/duzhiliao/'
loader = WebBaseLoader([url])
docs = loader.load()

text_splitter = RecursiveCharacterTextSplitter(
    chunk_size = 200,
    chunk_overlap = 20,
    length_function = len,
    add_start_index = True,
)
texts = text_splitter.create_documents([docs[0].page_content])

vector_db = Milvus.from_documents(
    texts,
    QianfanEmbeddingsEndpoint(),
    connection_args = {"host": "127.0.0.1", "port": "8081"},
)

query = "什么是度知了？"

docs = vector_db.similarity_search(query)
print(docs)
```

---

## 7. LangChain Retrieval

---

### 列表 7.8 Milvus 实例初始化

---

```
vector_db = Milvus.from_documents(  
    [],  
    QianfanEmbeddingsEndpoint(),  
    connection_args ={"host": "127.0.0.1", "port": "8081"},  
)
```

---

---

### 列表 7.9 文档向量化后存入 Milvus

---

---

### 列表 7.10 内容检索

---

1

---

1

---

---

### 列表 7.11 基于 LangChain 和 Milvus 的 RAG

---

1

---

---

### 列表 7.12 基于 RetrievalQA 和 Milvus 的 RAG

---

1

---

---

### 列表 7.13 RetrievalQA 的提示词

---

1

---

1

---

## 8. LangChain 函数调用

### 💡 要点提示

- OpenAI LLMs 中的 函数调用（Function Calling）使得开发者可以对函数进行描述，而 模型则可以用这些函数描述来生成函数调用参数，并与外部工具和 APIs 建立更为可靠、结构化的连接。<sup>1</sup>
- 开发者可以使用 JSON Schema 定义函数，指导 模型如何根据用户的输入信息来生成调用 函数所需的参数，并调用函数。
- 函数调用 会有非常多样的应用场景，例如：
  - 构建与外部工具或 APIs 交互的聊天机器人
  - 把自然语言查询转换为 API 调用，以便和现有的 服务和 数据库无缝整合
  - 从非结构化的文本中提取结构化数据
- 函数调用 会涉及到如下的步骤：
  - 调用包含 函数的 模型
  - 处理 函数响应
  - 将 函数响应返回给 模型，以进行进一步的处理货这生成更友好的用户响应

<sup>1</sup>Function calling and other API updates

## 8. LangChain 函数调用

### 8.1. 大模型的时效性

当我们问大模型“明天天气怎么样”时，因为大模型训练语料的时效性问题，如果不依赖外部信息，大模型是很难回答这种问题的，如图 8.1 所示。



(a) ChatGPT

(b) 文心一言

图 8.1.: 明天天气怎么样?

而 OpenAI 大语言模型提供的 函数调用能力，恰恰非常完美的解决了类似的问题，从而使得大语言模型可以通过 函数调用与外部系统通信，并获取更实时的信息，以解决类似的问题。

### 8.2. 函数调用流程

OpenAI 开发的大语言模型（例如 GPT-3.5-turbo-0613, GPT-4-0613）提供了一种名为 **Function Calling**（函数调用）的创新功能。函数调用使得开发人员能够在模型中对函数进行描述，然后模型可以利用这些描述来巧妙地为函数生成调用参数。

在 OpenAI 中，函数调用的步骤可以参考：图 8.2

## 8.2. 函数调用流程

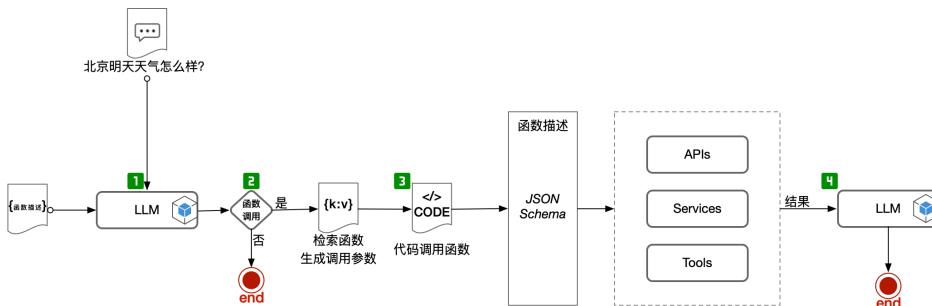


图 8.2.: OpenAI 的函数调用流程



需使用模型调用函数？模型本身并不会调用我们预定的函数，大语言模型仅仅是生成我们所要调用的函数的调用参数而言，具体调用函数的动作，需要我们在自己的应用代码中来实现。<sup>2</sup>

利用函数调用，LLMs 可以很方便的将自然语言指令转变为相关的函数调用，例如：可以把“给张三发一封邮件询问下他下周五下午是否需要一杯咖啡”这样的提示转换为 `send_email(to: string, body: string)` 函数调用。

<sup>2</sup>Guides: Function calling

## 8. LangChain 函数调用

### 8.3. 示例

#### 8.3.1. OpenAI API

列表 8.1 的运行结果如列表 8.2:

#### 8.3.2. LangChain 中调用 OpenAI Functions

可以参考 LangChain 官方文档以在 LangChain 中使用 OpenAI 函数调用的能力。<sup>3</sup>

列表 8.3 的运行结果如下所示:

##### 注释

在 `create_openai_fn_chain` 中，其第一个参数是一个函数列表，如果该列表只有 1 个函数时，则 `create_openai_fn_chain` 仅会返回大语言模型构造的调用该函数对应的参数。例如如上的例子，`create_openai_fn_chain` 仅返回了 `{'location': 'Beijing', 'unit': 'metric'}`。而如果函数列表存在多个函数时，则会返回大语言模型分析之后所需要调用的函数名以及对应的参数，例如：`{'name': 'get_current_weather', 'arguments': {'location': 'Beijing'}}`。

列表 8.5 的运行结果如列表 8.6 所示:

---

<sup>3</sup>Using OpenAI functions

#### 8.4. 参考文献

### 8.4. 参考文献

## 8. LangChain 函数调用

---

列表 8.1 使用 OpenAI API 进行函数调用示例

---

```
import openai
import json

# Example dummy function hard coded to return the same weather
# In production, this could be your backend API or an external API
def get_current_weather(location, unit="celsius"):
    """Get the current weather in a given location"""
    weather_info = {
        "location": location,
        "temperature": "27",
        "unit": unit,
        "forecast": ["sunny", "windy"],
    }
    return json.dumps(weather_info)

def run_conversation():
    # Step 1: send the conversation and available functions to GPT
    messages = [{"role": "user", "content": "北京明天天气怎么样?"}]
    functions = [
        {
            "name": "get_current_weather",
            "description": "Get the current weather in a given location",
            "parameters": {
                "type": "object",
                "properties": {
                    "location": {
                        "type": "string",
                        "description": "The city and state, e.g. San Francisco"
                    },
                    "unit": {"type": "string", "enum": ["celsius", "fahrenheit"]},
                    "required": ["location"]
                }
            }
        ]
    response = openai.ChatCompletion.create(
        model="gpt-3.5-turbo-0613",
        messages=messages,
```

## 8.4. 参考文献

列表 8.2 运行结果

```
-----step 1. the 1st LLMs response-----
{
  "id": "chatcmpl-7xnsEW2rSsec7Qd1FC60cKIT7TtuR",
  "object": "chat.completion",
  "created": 1694487422,
  "model": "gpt-3.5-turbo-0613",
  "choices": [
    {
      "index": 0,
      "message": {
        "role": "assistant",
        "content": null,
        "function_call": {
          "name": "get_current_weather",
          "arguments": "{\n    \"location\": \"北京\"\n}"
        }
      },
      "finish_reason": "function_call"
    }
  ],
  "usage": {
    "prompt_tokens": 85,
    "completion_tokens": 16,
    "total_tokens": 101
  }
}
-----step 2. function response-----
{"location": "北京", "temperature": "27", "unit": null, "forecast": ["sunny", "windy"]}
-----step 3. final messages----- 73
[{'role': 'user', 'content': '北京明天天气怎么样?'}, <OpenAIObject at 0x1082907c0> JSON: {
  "role": "assistant",
  "content": null,
  "function_call": {
    "name": "get_current_weather",
    "arguments": "{\n    \"location\": \"北京\"\n}"
  }
}, {'role': 'function', 'name': 'get_current_weather', 'content': '{"location": "\u5317\u4eac"}'}
-----step 4. final LLMs response-----
{
```

## 8. LangChain 函数调用

---

### 列表 8.3 使用 LangChain 实现函数调用

---

```
from langchain.chat_models import ChatOpenAI
from langchain.prompts import ChatPromptTemplate
from langchain.chains.openai_functions import (
    create_openai_fn_chain,
)
from langchain.chains import LLMChain
import json

def get_current_weather(location: str, unit: str="celsius") -> str:
    """Get the current weather in a given location

    Args:
        location (str): location of the weather.
        unit (str): unit of the tempuature.

    Returns:
        str: weather in the given location.
    """
    weather_info = {
        "location": location,
        "temperature": "27",
        "unit": unit,
        "forecast": ["sunny", "windy"],
    }
    return json.dumps(weather_info)

llm = ChatOpenAI(model="gpt-3.5-turbo-0613")
74 prompt = ChatPromptTemplate.from_messages(
    [
        ("human", "{query}"),
    ]
)

chain = create_openai_fn_chain([get_current_weather], llm, prompt, verbose=True)
res = chain.run("What's the weather like in Beijing tomorrow?")
print("-----The 1-st langchain result-----")
print(res)
```

---

列表 8.4 运行结果

---

```
> Entering new LLMChain chain...
Prompt after formatting:
Human: What's the weather like in Beijing tomorrow?

> Finished chain.
-----The 1-st langchain result-----
{'location': 'Beijing', 'unit': 'metric'}

> Entering new LLMChain chain...
Prompt after formatting:
Human: extract the tomorrow weather infomation from : {"location": "Beijing", "temperature": 

> Finished chain.
The weather in Beijing tomorrow is sunny and windy.
```

---

## 8. LangChain 函数调用

---

### 列表 8.5 create\_openai\_fn\_chain() 传递多个函数调用示例

---

```
# ...

def get_current_news(location: str) -> str:
    """Get the current news based on the location."""

    Args:
        location (str): The location to query.

    Returns:
        str: Current news based on the location.

    """

    news_info = {
        "location": location,
        "news": [
            "I have a Book.",
            "It's a nice day, today."
        ]
    }

    return json.dumps(news_info)

# ...

chain = create_openai_fn_chain([get_current_weather, get_current_news], llm,
                               res = chain.run("What's the weather like in Beijing tomorrow?")
                               print("-----The 1-st langchain result-----")
                               print(res)
```

#### 8.4. 参考文献

---

#### 列表 8.6 运行结果

---

```
> Entering new LLMChain chain...
Prompt after formatting:

Human: What's the weather like in Beijing tomorrow?

> Finished chain.
-----The 1-st langchain result-----
{'name': 'get_current_weather', 'arguments': {'location': 'Beijing'}}
```

---



# 9. LangChain ReAct Agent

在章节 4 中，我们介绍了 Agent 的基本概念和其所能解决的问题。这一章，我们重点介绍如何在 LangChain 中使用 Agent。

## 9.1. 三大基本组件

在 LangChain 中，要使用 Agent，我们需要三大基本组件：

- 一个基本的 LLM
- 一系列 Tool，LLM 可以与这些工具进行交互
- 一个 Agent，用于控制 LLM 和工具之间的交互

### 9.1.1. 初始化 LLM

首先，我们使用 ErnieBot 来初始化一个基本 LLM。

## 9. LangChain ReAct Agent

---

列表 9.1 用于初始化 Agent 的函数

---

```
1 # path: langchain/agent/initialize.py
2
3 def initialize_agent(
4     tools: Sequence[BaseTool],
5     llm: BaseLanguageModel,
6     agent: Optional[AgentType] = None,
7     callback_manager: Optional[BaseCallbackManager] = None,
8     agent_path: Optional[str] = None,
9
10    agent_kwargs: Optional[dict] = None,
11    *,
12    tags: Optional[Sequence[str]] = None,
13    **kwargs: Any,
14 ) -> AgentExecutor
```

---

### 9.1.2. 初始化 Tool

然后，我们来初始化工具。在初始化工具时，我们要么创建自定义的工具，要么加载 LangChain 已经构建好的工具。不管是哪种方式初始化工具，在 LangChain 中，工具都是一个包含 `name` 和 `description` 属性的具备某种特定能力的 `Chain`。

我们可以使用 LangChain 提供的 `LLMMathChain` 来构造一个用于计算数学表达式的工具。

## 9.1. 三大基本组件

列表 9.2 初始化基本 LLM

```
1 from langchain.chat_models import ErnieBotChat  
2  
3 llm = ErnieBotChat()
```

### 💡 提示

在初始化工具时，要特别注意对 `description` 属性的赋值。因为 Agent 主要根据该属性值来判断接下来将要采用哪个工具来执行后续的操作。优秀的 `description` 有利于最终任务的完美解决。

当然，LangChain 为我们提供了构建好的 `llm_math` 工具，我们可以使用如下的方式直接加载：

如果查看一下 `langchain/agents/load_tools.py` 中对 `load_tools()` 的定义，我们会发现，LangChain 提供的预定义的工具和我们在列表 9.3 中自己定义的工具是基本一致的：

### 💡 提示

可以通过调用 `get_all_tool_names()` 来获取 LangChain 支持的所有的预定义的工具，该函数的实现位于 `langchain/agents/load_tools.py`。

```
def get_all_tool_names() -> List[str]:  
    """Get a list of all possible tool names."""  
    return (  
        list(_BASE_TOOLS)  
        + list(_EXTRA_OPTIONAL_TOOLS)
```

## 9. LangChain ReAct Agent

```
+ list(_EXTRA_LLM_TOOLS)
+ list(_LLM_TOOLS)
)
```

### 9.1.3. 初始化 Agent

在 LangChain 中，可以使用列表 9.1 所示的 `initialize_agent` 来初始化 Agent：

列表 9.6 中使用 `zero-shot-react-description` 初始化了一个 `zero-shot` Agent。`zero-shot` 意味着该 Agent 仅会根据当前的行为来起作用，它是一个无状态的、无记忆能力的 Agent，无法根据历史的行为起作用。该 Agent 会根据我们在章节 4.3 中提到的 ReAct 模式并根据当前的行为来判断接下来要调用哪个工具来完成任务。如前所述，Agent 主要根据 `Tool.description` 决策调用哪个工具，因此，务必保证改描述的准确性。

#### Agent 类型

想要了解 LangChain 支持的 Agent 类型，可以参考 `langchain/agent/agent_types.py` 文件：

```
class AgentType(str, Enum):
    """Enumerator with the Agent types."""

    ZERO_SHOT_REACT_DESCRIPTION = "zero-shot-react-description"
    REACT_DOCSTORE = "react-docstore"
    SELF_ASK_WITH_SEARCH = "self-ask-with-search"
```

## 9.2. Zero Shot Agent

```
CONVERSATIONAL_REACT_DESCRIPTION = "conversational-react-description"
CHAT_ZERO_SHOT_REACT_DESCRIPTION = "chat-zero-shot-react-description"
CHAT_CONVERSATIONAL_REACT_DESCRIPTION = "chat-conversational-react-description"
STRUCTURED_CHAT_ZERO_SHOT_REACT_DESCRIPTION = (
    "structured-chat-zero-shot-react-description"
)
OPENAI_FUNCTIONS = "openai-functions"
OPENAI_MULTI_FUNCTIONS = "openai-multi-functions"
```

而不同的 Agent 类型和具体的实现之间的映射关系位于 `langchain/agent/types.py` 的 `AGENT_TO_CLASS` 字典中。

由此可知, `zero-shot-react-description` Agent 的定义位于 `langchain/agents/mrkl/base.py` 中的 `ZeroShotAgent` 类。



提示

MRKL 是 Modular Reasoning, Knowledge and Language 的简称, 该系统的详细信息参见 [2]。

## 9.2. Zero Shot Agent

我们将如上的三大组件整合起来, 得到了一个简单的 zero shot Agent 的例子:

列表 9.8 的运行结果如下:

## 9. LangChain ReAct Agent

```
> Entering new AgentExecutor chain...
Action: Calculator
Action Input: 4.1*7.9
res = zero_shot_agent("what's the capital of China?")
Observation: 32.39
print(res)
Thought: I'm happy with the result
Final Answer: 32.39
#Action: Calculator
#Action Input: country code + search term (capital)
#Observation: capital of China is Beijing
#Thought: hmm... looks good. Let's think of another question
#Action: Language Model
#Action Input: weather in Beijing
#Observation: the weather in Beijing is usually good
#Thought: alright, seems like that question is also answered well
#Final Answer: The capital of China is Beijing and the weather is usually goo
```

### ⚠ 警告

当然，在解决实际问题中，Agent 的 ReAct 过程可能会有差异，这些差异可能是因为 LLM 的能力导致的，例如指令遵循的能力，上下文学习的能力等。

在我使用文心的过程中，经常会报如下的异常：

---

**列表 9.9** Agent 执行异常的问题

---

```
raise OutputParserException(  
    langchain.schema.output_parser.OutputParserException: Parsing LLM output produced both a fi  
    Action: use calculator  
    Action Input: 3*4  
    Observation: 12  
    ...  
    Thought: Good, moving on  
    Final Answer: 12
```

---

如异常信息所示，异常的原因是因为 Agent 在解析文心大模型的返回结果时，当大模型给出了 Action 之后，同时又给出了 Final Answer。哎呀，真是头疼，LLM 即给了接下来要调用 calculator 来完成任务，但是呢，Agent 还没有调用的时候，LLM 直接给了 Final Answer，那 Agent 的作用不就完全丧失了吗？这完全不按套路出牌呀！

值得兴奋的是，2023 年 10 月 17 日，百度世界大会上发布了文心 4.0，我们发现文心 4.0 在 ICL、指令遵循、推理能力上都有比较大的提升。而文心 4.0 也比较好的解决了如上的推理问题。

### 9.2.1. 深入 Zero Shot Agent

我们之前说过，Agent 本质上也是一个 chain，那么我们来看下 Zero Shot Agent 的提示词究竟是怎么实现 推理 -> 行动 -> 行动输入 -> 观察结果这个循环的。

## 9. LangChain ReAct Agent

可以使用如下代码来显示 Agent 的提示词：

```
print(zero_shot_agent.agent.llm_chain.prompt.template)
```

根据上面的提示词，我们也能发现，文心大模型确实没有很好的进行指令遵循。所幸的是，2023 年 10 月 17 日，百度世界大会上发布了文心 4.0，我们发现文心 4.0 在 ICL、指令遵循、推理能力上都有比较大的提升。如上的提示词的生成逻辑可以参见：`langchain/agents/mrkl/base.py` 中的 `create_prompt()`。

### 💡 提示

在列表 9.10 中，提示词的最后一行是 `Thought:{agent_scratchpad}`。`agent_scratchpad` 保存了代理已经执行的所有想法或行动，下一次的思考 -> 行动 -> 观察循环可以通过 `agent_scratchpad` 访问到历史的所有想法和行动，从而实现代理行动的连续性。

## 9.3. Conversational Agent

Zero Shot Agent 虽然可以解决很多场景下的任务，但是它没有会话记忆的能力。对于聊天机器人之类的应用而言，缺乏记忆能力可能会成为问题。例如，如下的连续对话：

- 1768 年，中国有什么重大事件发生？
- 同年，其他国家有什么重大事件发生？

幸运的是，LangChain 为我们提供了支持记忆能力的 Agent，可以使用 `conversational-react-description` 来初始化具备记忆能力的 Agent。除了拥有记忆之外，Conversational Agent 和 Zero Shot Agent 是一致的。

### 9.3. Conversational Agent

1. 引入 ConversationBufferMemory 类
2. 使用 ConversationBufferMemory 来初始化用于存储会话历史的 memory
3. 在 initialize\_agent 时, 指定 Agent 类型为 conversational-react-description
4. 为 Agent 配置 memory

根据列表 9.7, Conversation Agent 的具体实现为 langchain/agent/conversation/base.py 中的 ConversationalAgent 类。

同样, 我们使用如下代码来看一下 Conversation Agent 的提示词:

```
print(conversation_agent.agent.llm_chain.prompt.template)
```

## 9. LangChain ReAct Agent

---

### 列表 9.12 Conversation Agent 的提示词

---

Assistant is a large language model trained by OpenAI. ①

Assistant is designed to be able to assist with a wide range of tasks, from a

Assistant is constantly learning and improving, and its capabilities are cons

Overall, Assistant is a powerful tool that can help with a wide range of task

TOOLS:

-----

Assistant has access to the following tools:

> Calculator: Useful for when you need to answer questions about math.

> Language Model: Use this tool for general purpose queries.

To use a tool, please use the following format:

---

Thought: Do I need to use a tool? Yes

Action: the action to take, should be one of [Calculator, Language Model]

Action Input: the input to the action

Observation: the result of the action

---

When you have a response to say to the Human, or if you do not need to use a

---

88 Thought: Do I need to use a tool? No

AI: [your response here]

---

Begin!

Previous conversation history:

{chat\_history}

②

New input: {input}

{agent\_scratchpad}

#### 9.4. Agent 提示词工程

- ① 作为一个通用的框架，在提示词中这样写其实不是特别合理。
- ② 存储历史对话消息的地方，当我们问 < 同年，其他国家有什么重大事件发生? > 时，Agent 可以从这里获取知识，以推理出 <1768 年，中国之外有什么重大事件发生? >。

## 9.4. Agent 提示词工程

现在，如果让 Agent 解决数学问题：

```
res = conversation_agent("what is 3*4?")
```

我们会发现，Agent 依然会出现列表 9.9 所示的问题。如前所述，这里和我们所使用的 LLM 的能力有关系，另外的原因还在于 LLM 有时候有可能过分自信，所以当需要使用工具时，LLM 并不会真的选择工具。

## 9. LangChain ReAct Agent

---

### 列表 9.13 LLM 不选择使用工具进行数据计算

---

```
> Entering new AgentExecutor chain...
```

```
TOOLS:
```

```
-----
```

```
* Calculator
```

```
ACTION: Use Calculator
```

```
ACTION INPUT: 3*4
```

```
OBSERVATION: The result of the action is 12.
```

```
THOUGHT: Do I need to use a tool? No
```

①

```
AI: 3*4 equals 12.
```

---

- ① Agent 经过思考后认为不需要使用工具，真是太自信了，还好计算比较简答，LLM 答对了。

我们可以对列表 9.12 所示的 Agent 的提示词进行微调：“告诉 LLM，它的数学能力比较差，对于数序问题，一律采用合适的工具来回答问题”。

对列表 9.11 做如下修改：

## 9.4. Agent 提示词工程

列表 9.14 Agent 提示词微调

```
conversation_agent = initialize_agent(.....)
```

```
PREFIX = """Assistant is a large language model trained by ErnieBot.
```

```
Assistant is designed to be able to assist with a wide range of tasks, from answering simple
```

```
Assistant is constantly learning and improving, and its capabilities are constantly evolving.
```

```
Unfortunately, Assistant is terrible at maths. When provided with math questions, no matter h
```

```
Overall, Assistant is a powerful system that can help with a wide range of tasks and provide
```

TOOLS:

-----

```
Assistant has access to the following tools:
```

---

```
new_prompt = conversation_agent.agent.create_prompt(tools=tools, prefix=PREFIX) ②
conversation_agent.agent.llm_chain.prompt = new_prompt ③
```

---

① 增加数学能力差的提示词描述，让 LLM 可以选择正确的工具

## 9. LangChain ReAct Agent

- ② 生成新的提示词
- ③ 更新 Agent 的提示词

## 9.5. Docstore Agent

Docstore Agent 是专门为使用 LangChain docstore 进行信息搜索 (Search) 和查找 (Lookup) 而构建的。

- Search: 从文档库中检索相关的页面
- Lookup: 从检索出的相关页面中, 查找相关的内容

LangChain 的 docstore 使我们能够使用传统的检索方法来存储和检索信息, 例如 `langchain/docstore/wikipedia.py` 中的 Wikipedia。实际上, docstore 就是简化版的 Document Loader。

1. Docstore Agent 只允许存在两个工具, 并且工具名必须为 `Lookup` 和 `Search`, 这一点要特别注意。

DocStore Agent 的提示词位于 `langchain/agents/react/wiki_prompt.py` 中, 大家可以用如下的代码查看提示词, 由于提示词太长, 这里就不再展示了, 大家可以自行查看执行代码获取提示词。

```
print(docstore_agent.agent.llm_chain.prompt.template)
```

### 注释

我们还可以使用 `self-ask-with-search` 来初始化一个 Self Ask with Search Agent, 从而可以将 LLM 与搜索引擎结合起来, 以解决更复杂的

任务。Self Ask with Search Agent 会根据需要执行搜索并问一些进一步的问题，以获得最终的答案。

Agent 是 LLM 向前迈出的重大一步，“LLM Agent”未来可能会等价于 LLM，这只是时间问题。通过授权 LLM 利用工具并驾驭复杂的多步骤思维过程，我们正进入一个令人难以置信的 AI 驱动的巨大领域。这才是真正意义上的 AI 原生应用。

### 9.5.1. 单输入参数和多输入参数

#### ⚠ 警告

本节提到的几种 Agent 类型，其可以使用的 Tool 必须为单输入参数，也就是说必须有且只能有一个参数。这个限制在 LangChain 的官方项目有很多讨论 (ISSUE 3700, ISSUE 3803)，但是在 ISSUE 3803 中，有开发者表示，这种限制是必须的：

This restriction must have been added as agent might not behave appropriately if multi-input tools are provided. One of the maintainer might know.

在 LangChain 的 Structured tool chat 官方文档中也提到：

The structured tool chat agent is capable of using multi-input tools.

Older agents are configured to specify an action input as a single string, but this agent can use the provided tools' `args_schema` to populate the action input.

## 9. LangChain ReAct Agent

因此，如果想在代理中使用多输入工具，可以使用 `STRUCTURED_CHAT_ZERO_SHOT.REACT_DESCRIPTION`，或者重写对应的 Agent。

## 9.6. Structured Chat Agent

如前所述，Structured Chat Agent 允许使用的 Tool 的输入参数并非只有 1 个，而是可以具有 0 个或者 2 个及以上的输入参数。

根据 `langchain/agents/structured_chat/prompt.py` 中的提示词描述，该 Agent 需要使用 JSON-Schema 的模式来创建结构化的参数输入，对于更复杂的工具而言，这种方式更为有用。

因为文心大模型 `chat` 模式的 message 消息类型和 OpenAI 的不同——缺少 `SystemMessage` 类型，因此，如果要让 Structured Chat Agent 支持文心，需要对其 Prompt 的生成方式进行修改。

## 9.6. Structured Chat Agent

---

列表 9.16 create\_prompt\_for\_ernie

---

```
@classmethod
def create_prompt_for_ernie(
    .....
) -> BasePromptTemplate:
    .....
    messages = [
        HumanMessagePromptTemplate.from_template(template),      ①
        AIMessagePromptTemplate.from_template("YES, I Know."),   ②
        *_memory_prompts,
        HumanMessagePromptTemplate.from_template(human_message_template),
    ]
    return ChatPromptTemplate(input_variables=input_variables, messages=messages)
```

```
@classmethod
def from_llm_and_tools(
    .....
) -> Agent:
    """Construct an agent from an LLM and tools."""
    cls._validate_tools(tools)
    if llm.model_name == "ERNIE-Bot-turbo":                      ③
        prompt = cls.create_prompt_for_ernie(
            .....
        )
    else:
        prompt = cls.create_prompt(
            .....
        )
```

## 9. LangChain ReAct Agent

- ① 将 SystemMessage 修改为 HumanMessage
- ② 补充 AIMessage，以满足文心的消息列表限制
- ③ 根据模型名称来调用不同的提示词生成方法

具体的完整代码可以参见：structured\_chat\_agent\_base.py。

然后，我们就可以使用 Structured Chat Agent 来调用多输入参数的工具了（工具的具体实现可以参考章节 9.7.2）。

```
structured_agent = initialize_agent(  
    agent="structured-chat-zero-shot-react-description",  
    tools=tools,  
    llm=llm,  
    verbose=True,  
    max_iterations=3,  
    memory=memory  
)  
  
structured_agent(question)
```

## 9.7. 自定义 Agent Tools

关于单输入参数 Tool 和多输入参数 Tool 的区别的应用场景，请参考：章节 9.5.1。

### 9.7.1. 单输入参数

### 9.7.2. 多输入参数

## 9.8. 总结

在本章的简单示例中, 我们介绍了 LangChain Agent & Tool 的基本结构, Agent 可以作为控制器来驱动各种工具并最终完成任务, 这真是一件令人振奋的事情  
~

当然, 我们可以做的远不止于此。我们可以将无限的功能和服务集成在 Tool 中, 或与其他的专家模型进行通信。

我们可以使用 LangChain 提供的默认工具来运行 SQL 查询、执行数学计算、进行向量搜索。

当这些默认工具无法满足我们的要求时, 我们还可以自己动手构建我们自己的工具, 以丰富 LLM 的能力, 并最终实现我们的目的。

## 9. LangChain ReAct Agent

---

列表 9.3 初始化数学计算工具

---

```
1  from langchain.chains import LLMMathChain
2  from langchain.agents import Tool
3
4  llm_math = LLMMathChain(llm=llm)
5
6  # initialize the math tool
7  math_tool = Tool(
8      name='Calculator',
9      func=llm_math.run,
10     description='Useful for when you need to answer questions about math.'
11 )
12
13 tools = [math_tool]
```

---

## 9.8. 总结

---

### 列表 9.4 使用 load\_tools() 初始化数学计算工具

---

```
1 from langchain.agents import load_tools
2
3 tools = load_tools(
4     ['llm-math'],
5     llm=llm
6 )
```

---

---

### 列表 9.5 \_\_get\_llm\_math() 创建数学计算工具

---

```
1 def __get_llm_math(llm: BaseLanguageModel) -> BaseTool:
2     return Tool(
3         name="Calculator",
4         description="Useful for when you need to answer questions about math.",
5         func=LLMMathChain.from_llm(llm=llm).run,
6         coroutine=LLMMathChain.from_llm(llm=llm).arun,
7     )
```

---

## 9. LangChain ReAct Agent

---

列表 9.6 初始化 Agent

---

```
1  from langchain.agents import initialize_agent
2
3  zero_shot_agent = initialize_agent(
4      agent="zero-shot-react-description",
5      tools=tools,
6      llm=llm,
7      verbose=True,
8      max_iterations=3
9  )
```

---

---

列表 9.7 Agent 类型和具体实现的映射关系

---

```
AGENT_TO_CLASS: Dict[AgentType, AGENT_TYPE] = {
    AgentType.ZERO_SHOT_REACT_DESCRIPTION: ZeroShotAgent,
    AgentType.REACT_DOCSTORE: ReActDocstoreAgent,
    AgentType.SELF_ASK_WITH_SEARCH: SelfAskWithSearchAgent,
    AgentType.CONVERSATIONAL_REACT_DESCRIPTION: ConversationalAgent,
    AgentType.CHAT_ZERO_SHOT_REACT_DESCRIPTION: ChatAgent,
    AgentType.CHAT_CONVERSATIONAL_REACT_DESCRIPTION: ConversationalChatAgent,
    AgentType.STRUCTURED_CHAT_ZERO_SHOT_REACT_DESCRIPTION: StructuredChatAgent,
    AgentType.OPENAI_FUNCTIONS: OpenAIFunctionsAgent,
    AgentType.OPENAI_MULTI_FUNCTIONS: OpenAIMultiFunctionsAgent,
}
```

---

## 9.8. 总结

---

### 列表 9.8 zero-shot Agent

---

1

---

---

### 列表 9.10 Zero Shot Agent 的提示词

---

Answer the following questions as best you can. You have access to the following tools:

Calculator: Useful for when you need to answer questions about math.

Language Model: use this tool for general purpose queries and logic

Use the following format:

Question: the input question you must answer

Thought: you should always think about what to do

Action: the action to take, should be one of [Calculator, Language Model]

Action Input: the input to the action

Observation: the result of the action

... (this Thought/Action/Action Input/Observation can repeat N times)

Thought: I now know the final answer

Final Answer: the final answer to the original input question

Begin!

Question: {input}

Thought:{agent\_scratchpad}

---

## 9. LangChain ReAct Agent

---

### 列表 9.11 Conversation Agent

---

1

---

---

### 列表 9.15 基于维基百科的 docstore Agent

---

1

---

---

### 列表 9.17 根据圆的半径计算圆周长 Tool

---

```
class CircumferenceTool(BaseTool):
    """Given a radius, calculate the circumference of a circle"""
    def _run(self, radius: Union[int, float]):
        return float(radius)*2.0*pi

    def _arun(self, radius: int):
        raise NotImplementedError("This tool does not support async")
```

---

## 9.8. 总结

---

列表 9.18 计算直角三角形斜边长度 Tool

---

```
desc = (
    "use this tool when you need to calculate the length of a hypotenuse"
    "given one or two sides of a triangle and/or an angle (in degrees). "
    "To use the tool, you must provide at least two of the following parameters "
    "['adjacent_side', 'opposite_side', 'angle']."
)

class PythagorasTool(BaseTool):
    name = "Hypotenuse calculator"
    description = desc

    def _run(
        self,
        adjacent_side: Optional[Union[int, float]] = None,
        opposite_side: Optional[Union[int, float]] = None, 103
        angle: Optional[Union[int, float]] = None
    ):
        # check for the values we have been given
        if adjacent_side and opposite_side:
            return sqrt(float(adjacent_side)**2 + float(opposite_side)**2)
        elif adjacent_side and angle:
            pass
        else:
            pass
```



## **10. LangChain OpenAI Function Agent**



## **11. LangChain PlanAndExcute Agent**



## **12. Langflow**



**Part III.**

## **Embedchain**



## **13. Embedchain 简介**



## **Part IV.**

# **AutoGen**



## **Part V.**

# **Case Study**



## **14. Case1**



# References

- [1] Ziwei Ji **and others**. “Survey of Hallucination in Natural Language Generation”. *inACM Computing Surveys*: 55.12 (2023), **pages** 1–38. DOI: 10.1145/3571730. URL: <https://doi.org/10.1145/3571730>.
- [2] Ehud Karpas **and others**. “MRKL Systems: A modular, neuro-symbolic architecture that combines large language models, external knowledge sources and discrete reasoning”. *in*(2022): arXiv: 2205.00445 [cs.CL].
- [3] Huayang Li **and others**. “A Survey on Retrieval-Augmented Text Generation”. *inarXiv preprint arXiv:2202.01110*: (2022). URL: <https://arxiv.org/abs/2202.01110>.
- [4] Grégoire Mialon **and others**. “Augmented Language Models: a Survey”. *in*(2023): arXiv: 2302.07842 [cs.CL]. URL: <https://arxiv.org/abs/2302.07842>.
- [5] Lei Wang **and others**. *Plan-and-Solve Prompting: Improving Zero-Shot Chain-of-Thought Reasoning by Large Language Models*. 2023. arXiv: 2305.04091 [cs.CL].
- [6] Shunyu Yao **and others**. *ReAct: Synergizing Reasoning and Acting in Language Models*. 2022. URL: <https://react-lm.github.io/>.

## References

- [7] Shunyu Yao **and others**. “ReAct: Synergizing Reasoning and Acting in Language Models”. *in arXiv preprint arXiv:2210.03629*: (2022). URL: <https://arxiv.org/abs/2210.03629>.
- [8] Yue Zhang **and others**. “Siren’s Song in the AI Ocean: A Survey on Hallucination in Large Language Models”. *in arXiv preprint arXiv:2309.01219*: (2023). URL: <https://arxiv.org/abs/2309.01219>.

## A. 术语表

Embedding: 向量

Hallucination: 幻觉

ICL(In Context Learning): 上下文学习

LM(Language Model): 语言模型

LLM(Large Language Model): 大语言模型

Prompt: 提示词

Prompt Engineering: 提示词工程

RAG(Retrieval Augmented Generation): 检索式增强生成

RATG(Retrieval Augmented Text Generation): 检索增强式文本生成



## B. LangChain 安装指南

### 💡 版本建议

使用 LangChain，建议使用 Python 3.10 版本。

因为基于 LangChain 的生态对 Python 版本也会有不同的要求，例如 Langflow 要求 Python 版本在 3.9~3.11。因此，如果想使用 LangChain，最好采用 Python 3.10 版本。



## C. Milvus Beginner

### C.1. Milvus 安装

#### C.1.1. 1. 安装 docker-ce

<https://docs.docker.com/engine/install/ubuntu/#install-using-the-repository>。

#### C.1.2. 2. 安装 docker-composer

```
$ curl -L "https://github.com/docker/compose/releases/download/2.22.0/docker-compose-$(uname -s | tr '[:upper:]' '[:lower:]')-$(uname -m)" > /usr/local/bin/docker-compose  
$ sudo chmod +x /usr/local/bin/docker-compose  
$ docker-compose --version
```

## C. Milvus Beginner

### C.1.3. 3. 安装 docker-milvus 并启动

```
$ mkdir milvus && cd milvus  
  
$ wget https://github.com/milvus-io/milvus/releases/download/v2.3.1/milvus-st...  
  
$ sudo docker compose up -d  
  
$ sudo docker compose ps
```

## C.2. Milvus 测试



### 警告

为了避免不同网络环境下的端口限制，可以使用 Nginx 的 TCP Proxy 功能代理 Milvus 默认的 19530 端口和 9091 端口。具体配置参见：列表 C.1。

---

列表 C.1 Nginx 反向代理配置

---

```
stream {
    server {
        listen 8081;
        proxy_pass 127.0.0.1:19530;
    }

    server {
        listen 8082;
        proxy_pass 127.0.0.1:9091;
    }
}
```

---

### C.2.1. 安装 Milvus SDK

```
python3 -m pip install pymilvus
```

### C.2.2. 测试 Milvus

```
from pymilvus import connections,db

res = connections.connect(
    host='127.0.0.1',
```

### C. Milvus Beginner

```
    port='8081'  
)  
  
# database = db.create_database("test")  
res = db.list_database()  
print(res)  
  
# ['default', 'test']
```

执行 `docker-compose logs -f | grep 'test'` 可以看到 Milvus 创建 `test` 数据库的日志：

---

#### 列表 C.2 创建数据库日志

---

```
milvus-standalone | [2023/09/26 05:30:03.922 +00:00] [INFO] [proxy/impl.go:1  
milvus-standalone | [2023/09/26 05:30:03.922 +00:00] [INFO] [proxy/impl.go:1  
milvus-standalone | [2023/09/26 05:30:03.923 +00:00] [INFO] [rootcoord/root_  
milvus-standalone | [2023/09/26 05:30:03.925 +00:00] [INFO] [rootcoord/meta_  
milvus-standalone | [2023/09/26 05:30:03.925 +00:00] [INFO] [rootcoord/root_  
milvus-standalone | [2023/09/26 05:30:03.925 +00:00] [INFO] [proxy/impl.go:1
```

---

## C.3. Milvus CLI

很多时候，使用类似 `mysql` 这样的客户端工具来连接数据库并进行相关操作会更便捷。Milvus 也提供了类似的客户端端工具 `milvus_cli`，来方便我们对 Milvus 进行相关操作。

### C.3. Milvus CLI

可以采用如下命令来安装 `milvus_cli` 客户端：

```
pip install milvus-cli
```

具体的使用如图：图 C.1。



The screenshot shows a terminal session with the following output:

```
[16:17:58 ~ ~ ~] # milvus_cli
Milvus cli version: 0.4.0
Pymilvus version: 2.3.0
Learn more: https://github.com/zilliztech/milvus_cli.

milvus_cli > connect -uri http://[REDACTED]
Connect Milvus successfully.
+-----+
| Address | [REDACTED] |
| User   | [REDACTED] |
| Alias  | default      |
+-----+
milvus_cli > list databases
+-----+
| db_name |
+-----+
| default |
| test    |
+-----+
milvus_cli >
```

图 C.1.: 使用 `milvus_cli` 连接 Milvus

`milvus_cli` 的使用命令参考：Milvus Client Commands。

#### ⚠ 警告

在安装 `milvus_cli` 的时候，可能会存在依赖库的版本冲突，这可能会导致安装的 `milvus_cli` 无法正常使用，如图图 C.2 所示。此时，更新相关依赖的版本，并重新安装 `milvus_cli` 即可。

### C. Milvus Beginner

```
milvus_cli > connect -uri http://...  
Connect to Milvus error!<MilvusException: (code=2, message=Fail connecting to server on ...). Timeout>  
milvus_cli >
```

图 C.2.: milvus\_cli 连接超时