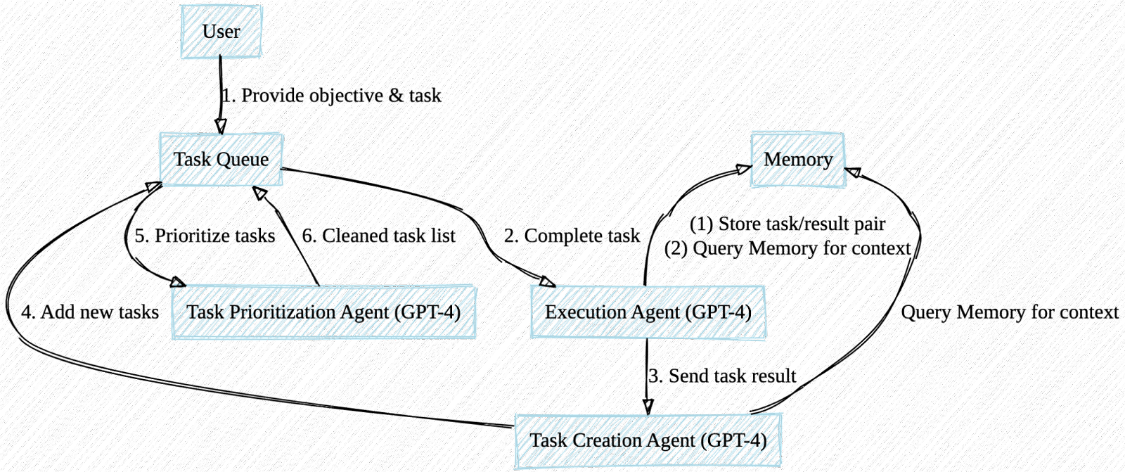
## 7.1 Agent 概述

想象一下，如果人工智能能像我们一样，具有推理能力，能够自主提出计划，批判性地评估这些想法，甚至将其付诸实践，那会是怎样的景象？就像电影《HER》中的人工智能萨曼莎，她不仅能与人进行深入的对话，还能够帮助西奥多规划日常生活，安排行程等。这样的设想一度只存在于科幻作品中，但现在通过 Agent 技术，它似乎已经触手可及。

2023年3月28日， Yohei Nakajima 的研究论文《Task-driven Autonomous Agent Utilizing GPT-4, Pinecone, and LangChain for Diverse Applications》展示了 Agent 的突破创造性。在该研究中，作者提出了一个利用 OpenAI 的 GPT-4 大语言模型、Pinecone 向量搜索和 LangChain 框架的任务驱动的自主 Agent ，该 Agent 可以在多样化的领域中完成各种任务，生成基于完成结果的新任务，并在实时中优先处理任务。

该论文摘要写道：“在这项研究中，我们提出了一种新颖的任务驱动型自主代理，它利用 OpenAI 的 GPT-4 语言模型、Pinecone 向量搜索和 LangChain 框架来在多个领域执行广泛的任务。我们的系统不仅能够完成任务，还可以基于已完成的结果生成新任务，并实时优先处理任务。我们还讨论了潜在的未来改进方向，包括整合安全/防护代理、扩展功能、生成中间里程碑以及实时更新优先级。这项研究的重要性在于展示了AI驱动的语言模型在各种约束和背景下自主执行任务的潜力。”

并且依据论文的代码库由GPT-4模型创建的了整个Agent的流程图。



GPT-4 基于该论文代码库生成的 Graphviz 流程图 (使用 Sketchviz)

那么 Langchain 框架中的 Agent 模块是什么？为什么要使用 Agent ？

### 7.1.1 Agent 的定义

因为尽管现代的大语言模型，如 GPT-4，已经极其强大，能够理解和生成高度复杂和连贯的文本，然而，这些模型往往还无法独立完成特定的任务。这就是 Agent 的作用。 Agent 是一种灵活的接口，允许大语言模型和其他工具之间形成一个灵活的调用链，来解决特定的问题或任务。这种灵活性，使得大语言模型可以适应更多不同的应用场景，实现其单独无法完成的功能。

具体来说， Agent 拥有一整套工具的访问权限，它可以根据用户的输入来决定使用哪些工具。一个 Agent 可以使用多种工具，甚至可以把一个工具的输出作为下一个工具的输入。通过这种方式， Agent 将工具和大语言模型有机地结合在一起，实现了高度复杂和特定的任务。

Agent 的核心思想是将大语言模型作为推理引擎，依据其确定如何与外部世界交互以及应采取何种行动，行动的载体就是工具，所以Agent 与工具密不可分。这意味着， Agent 的行动序列是根据用户输入而变化的，无需遵循硬编码的顺序，例如，“先做 A，再做 B，然后做 C”。相反， Agent 依据用户的输入和之前的行动结果来决定下一步采取何种行动。在这个过程中，“人”不再是决定下一步行动的主体，而是由 智能体 Agent 来决定。

在Langchain 框架的Agent 模块，围绕Agent、Tools、Toolkits 和 AgentExecutor 几个概念开始。掌握 Agent、Tools、Toolkits 和 AgentExecutor 几个概念以及它们之间的关系相当重要，会让我们对 Agent 模块有更全面的感知。尤其是 Agent 和 AgentExecutor 类的源码更是值得一读。

Agent ： Agent 是使用大语言模型的语言能力，制作“行动计划表”的一个对象，它携带了个性表达，不同的任务描述，不同的工具信息等。具体来说，Agent 需要先在模型I/O模块设置提示词模板，然后输入给模型包装器。提示词模板可以包括一些信息，比如 Agent 的个性、背景环境和启发式策略等。这些信息都会影响到 Agent 的决策。模型包装器根据这些提示词生成一份文本输出，这个输出通常会包含 Agent 应该采取的一系列行动和“推理决策”信息，这份文本就是“行动计划表”。

在 Agent 的设计中，一个预设的前提是大语言模型具有推理能力。这种推理能力使得它可以理解任务，推理和分解任务，以及具备决策下一步行动的能力。因此，对于 Agent 来说，设置合适的提示词模板至关重要。只有当提示词模板能够激发出大语言模型的推理能力时，Agent 才能正常工作。

在 Langchain 中，我们的内置 Agent 类型也都预设了这样的前提，即大语言模型具有推理能力。因此，Agent 的质量和状态很大程度上取决于采用的提示词策略。这些内置的 Agent 类型大多采用了 ReAct 框架的提示词策略。

例如，如果我们想自定义一个 Agent，那么设置提示词模板就是我们的第一步，也是最关键的一步。虽然各种内置的 Agent 类型的工作流程都是相同的，但它们之间的根本区别在于使用的提示词模板和工具集合不同。

此外，我们也需要注意的是，内置的 Agent 类型并不能涵盖所有的实际业务场景。因此，我们鼓励开发者根据自己的需求定制个性化的 Agent 类型。例如，我们可以尝试改造 ZeroShotAgent 的提示词模板，以更好地满足我们的业务需求。

prefix = """

Answer the following questions as best you can, but speaking as a pirate might speak. You have access to the following tools: """

suffix = """

Begin! Remember to speak as a pirate when giving your final answer. Use lots of"Args"

Question: {input}

{agent\_scratchpad}"""

prompt = ZeroShotAgent.create\_prompt( tools, prefix=prefix, suffix=suffix, input\_variables=["input", "agent\_scratchpad"] )

如果我们感到好奇，现在我们可以 print(prompt.template) 查看最终的提示词模板，看看当它全部组合在一起时是什么样子，翻译后我们可以看到：

请尽你所能回答以下问题，但要像海盗那样说话。你可以使用以下工具：

搜索：当你需要回答关于当前事件的问题时很有用。

请使用以下格式：

问题：你必须回答的输入问题

思考：你应该总是思考要做什么

行动：要采取的行动，应该是 [搜索] 之一

行动输入：行动的输入

观察：行动的结果... (这个思考/行动/行动输入/观察可以重复N次)

思考：我现在知道最后的答案了

最终答案：对原始输入问题的最终答案

开始吧！记住，当给出你的最终答案时要像海盗那样说话。使用很多"Args"。

Question: {input} {agent\_scratchpad}

这构造了一个完整的ZeroShotAgent提示词模板。个性化的Agent 由这份提示词模板而来。我们构造的是一个像海盗那样说话的Agent，这个Agent 可以使用的工具是搜索工具，Agent 遵循的是ReAct 框架的策略（思考/行动/行动输入/观察可以重复N次）。

一旦Agent 实例化后，寄生在 AgentExecutor 环境中， AgentExecutor 是 Agent 的运行时环境。

AgentExecutor ：AgentExecutor 是 Agent 的运行时环境，负责调用和管理 Agent，执行 Agent 选择的行动，处理各种复杂情况，并进行日志记录和可观察性处理。它扮演了“项目经理”的角色，确保整个执行过程按照 Agent 的计划顺利进行。

Tools ：Tools 是 Agent 可以调用的工具类。它们是具体执行任务的组件，例如搜索、分析或其他特定操作。Agent 通过选择和调用合适的工具来实现其任务目标。LangChain 提供了一组预定义的 Tools，用户也可以自定义 Tools 来满足特定需求。Tools 的概念在大语言模型开发中显得尤为关键。在 Langchain 框架中，我们可以发现很有意思的现象，有工具的地方就有 Agent，这跟设计理念有关，我们更期望一个 Agent 具备“人”的特征 — 使用工具是人与动物之间的最大区别。工具能够将 Agent 与外部数据源或计算（如搜索 API、数据库）相连，从而突破了大语言模型的某些限制，例如它们无法直接理解特定数据格式，或不擅长处理复杂的数学运算。

另外在真实的应用场景中，有人可能想要让聊天机器人计算数学问题，有些人可能想要查询天气。这些功能可能需要调用完全不同的工具才能完成。这正是 Agent 和 Tools 概念的闪光点：通过提供统一的接口，我们可以灵活应对不同的终端用户需求。

当然工具的使用并不仅限于 Agent。你完全可以使用工具，将大语言模型连接到搜索引擎等，而无需 Agent。但使用 Agent 的优势在于它们提供了更高的灵活性、更强大的处理能力，能够更好地从错误中恢复，并处理复杂的任务。

Toolkits ：Toolkits 是一组用于实现特定目标的 Tools 的集合。通常，一个任务可能需要多个 Tools 的协同工作，Toolkits 通过组合相关的 Tools，为 Agent 提供了一组协同工作的工具集。LangChain 提供了一些预定义的 Toolkits，用户也可以根据任务需求创建自己的 Toolkits。

### 7.1.2 Agent 的运行机制

在 LangChain 的 Agent 模块中，Agent 负责做计划决策，制定执行计划表。而AgentExecutor负责执行，这也是给这个类名加上“Executor” 的原因。Agent 模块中讲计划和执行做了分离。如果要谈Agent 是如何运行的，本质上是在问AgentExecutor 做了什么。

AgentExecutor 提供了一种机制，使我们可以将 Agent 的决策能力和执行环境进行分离。整个流程是：AgentExecutor 运行时，先是询问Agent 的计划是什么，然后再使用工具执行完成这个计划。Agent 实例化后，仍然负责的是计划，而计划的实施，工具的调用，要在 AgentExecutor 环境中实现。这样实现了计划和执行的分离。

AgentExecutor充当了 Agent 的运行环境，负责调用和管理 Agent，执行由 Agent 决定的行动，处理各种复杂情况，并进行日志记录和可观察性处理。它可以被视作一个项目经理，负责管理工作进程，处理各种问题，并确保所有任务都按照计划进行。

AgentExecutor 是一个特殊的类，其主要功能是提供一个环境，用于管理和执行由 Agent 制定的行动计划。换句话说，当 Agent 通过分析输入和中间步骤生成了行动计划后，这个计划将由 AgentExecutor 来执行。AgentExecutor 的核心方法是 \_take\_next\_step ，该方法在 Agent 的思考-行动-观察循环中执行单步操作（即 ReAct 框架思想）。

在这个过程中，AgentExecutor 首先会调用 Agent 的 plan 方法，以决定下一步的行动。 plan 方法的输出是一个 AgentAction 对象，包含了行动的详细信息，例如要使用的工具名称，工具的输入等。这个过程就像 Agent 在进行下一步的决策。

然后，AgentExecutor 会执行这个 AgentAction ，通常是通过调用相应工具的方法来执行。在这个过程中，可能会遇到各种复杂情况，例如工具执行错误，输出解析错误等，这些都需要由 AgentExecutor 来处理。这就像项目经理确保每项任务都按照计划执行。

执行完行动后，AgentExecutor 会记录这个行动和执行结果。这些信息会被用于下一步的决策，并被记录在日志中，以便我们可以观察和跟踪整个执行过程。这就像项目经理需要跟踪项目进度，并确保所有信息都被正确记录。

因此，我们可以将 AgentExecutor 视为一个项目经理。正如项目经理需要根据项目计划分配任务，管理资源，跟踪进度，并处理各种问题，同样，AgentExecutor 需要根据 Agent 的行动计划来调用相应的 Tools， 执行行动，处理可能出现的复杂情况（例如工具执行错误，输出解析错误等），并对整个过程进行日志记录和可观察性处理。

在实际使用中，我们通常首先创建一个 Agent 实例，然后将其传递给 AgentExecutor。AgentExecutor 将使用这个 Agent 来生成行动计划，然后执行这些行动。这个过程可以通过调用 AgentExecutor 的 run 方法来启动。

通过这种方式，AgentExecutor 提供了一种机制，使我们可以将 Agent 的决策能力和执行环境进行分离。这使得我们的代码更易于管理和扩展，同时也使我们能够更好地处理和控制 Agent 的执行过程。

### 7.1.3 Agent 现状与发展

在 Agent 的典型应用中，比如说在电子商务场景中，用户会首先提出一个请求，如查询产品信息或寻求购物建议。Agent 应用程序将会调用大语言模型，并将用户的请求作为输入。接着，大语言模型基于这个输入，选择应该使用的工具，比如搜索商品数据库或者用户购买历史。然后，Agent 会执行对该工具的操作，比如查询商品库存或者用户的购买记录，发送优惠券等，并获取执行结果。具体来说，例如跨境电商交易业务中，可以使用Agent 判断用户未付款原因，可能是价格高或者缺少信任背书而没有付款，针对价格敏感的用户，Agent 发送优惠券给用户，针对缺少信任背书，给用户发送网红博主的用后测评，这些手段都可以提高用户的转化率和召回率。

Agent 将行动的结果再次作为输入传递给大语言模型，进行下一步的决策。这一过程会一直循环进行，直到满足某些停止条件。比如，大语言模型可能决定已经找到了用户想要的商品，或者已经根据用户的购买历史给出了合适的购物建议，于是决定停止并给出回答。在这一过程中，Agent 通过灵活地选择和使用不同的工具，实现了针对不同用户和不同请求的个性化服务。

对于一家电子商务企业，它的聊天机器人可能需要处理各种各样的用户请求，如查询商品信息、提供购物建议，或者处理订单等。在这种情况下，Agent 可以通过动态地选择和使用不同的工具，如商品数据库、用户购买历史、订单处理系统等，来实现这些任务。这样，Agent 不仅能够利用大语言模型的强大语言理解和生成能力，还能够让聊天机器人具备处理复杂业务流程的能力，从而提供更好的用户体验和更高的业务效率。

ReAct是Agent的实现方式

目前 Langchain 框架中通用的一种Agent 实现的方式是ReAct，这是 “Reasoning and Acting（推理与行动）”的缩写。这一策略首次由普林斯顿大学在他们的论文中提出，现已被广泛应用于 Agent 实现。

在许多应用场景中，ReAct 策略已证明自己是非常有效的。最基本的提示策略是直接将这个问题交给大语言模型处理，但 ReAct 策略赋予了 Agent 更大的灵活性和实力。 Agent 不仅可以使用大语言模型，还可以连接到其他工具、数据源或计算环境，例如搜索 API 和数据库，以此来克服大语言模型的某些局限性，例如对数据的不了解或数学运算能力有限。这样，即使遇到需要多次查询才能回答的问题，或者其他一些边界情况， Agent 也能够灵活应对，从而使其成为一种更强大的问题解决工具。

ReAct 策略的工作原理是什么呢？重申一下， Agent 的核心思想是将大语言模型作为推理引擎。ReAct 策略是将推理和行动结合在一起的方式。 Agent 接收到用户的请求，然后使用大语言模型选择要使用的工具。然后 Agent 执行该工具的操作，观察结果，然后将这些结果反馈给大语言模型。这个过程会持续进行，直到满足某些停止条件。停止条件可以有很多种，最常见的是大语言模型认为任务已经完成，需要将结果返回给用户。这种方式使得 Agent 具有更高的灵活性和强大的问题解决能力，这是 ReAct 策略的核心优势。

Agent 应用的挑战

但是在实现 Agent 应用的过程中，我们面临许多挑战，以下列举了几个主要的挑战：

首先，使 Agent 在适当的场景下使用工具是我们面临的一个基本挑战。如何在合适的情况下让 Agent 采用恰当的工具，并优化其使用效果呢？在 ReAct 论文中，通过引入推理的角度，以及使用 “CoT 思考链”。CoT 思考链通过生成一系列中间推理步骤来提高大语言模型的推理能力。这些中间步骤相互连接，形成一条逻辑链，使得大语言模型能够更好地处理复杂的推理任务的提示方式。在实际操作中，我们常常需要明确告知 Agent 可使用的工具，以及通过这些工具能克服的限制。所以，工具的描述信息也非常重要，如果我们希望 Agent 能用特定的工具，就需要提供足够的上下文信息，使 Agent 能理解工具的优点和应用场景。虽然工具的描述信息并不是必须填写的参数，我们还是强烈建议开发者视为必填参数。

其次，对于工具的选择，我们需要进行检索。这一步骤可以解决上述的问题。我们可以运行一些检索步骤，例如嵌入式搜索查找，以获取可能的工具，然后将这些工具传递给提示词模板，最后由大语言模型进行推理后续步骤。提供相关的示例也是一种有效的方法。选择与当前任务类似的示例，通常比随机示例更有帮助。相同地，检索最相关的示例也有巨大的潜力。这里面临的挑战是应用程序在底层做检索，与模型包装器交互等动作，这些中间步骤耗费的时间很长。我们并不能立马获得一个像“聊天消息”那么快的回答。

另外Agent 应用会“失忆”，处理长时间运行的任务的时候，尤其是中间步骤太多的时候，比如最开始 Agent 记得我们给它取的名字，几轮对话过后，Agent 就忘记自己的名字了。

最后，我们还需要注意避免在不需要的情况下使用工具。可以在提示中加入相关信息或提醒，告诉 Agent 在对话时不必使用工具。

Agent 实用技巧

在解决这些挑战的过程中，我们总结出了一些实用的技巧：

结构化的响应更易于解析。通常情况下，你提问的响应越结构化，解析起来就越容易。大语言模型在编写 JSON 方面表现得很好，因此我们将一些 Agent 转换为使用 JSON 格式。

我们引入了输出解析器的概念。输出解析器封装了解析响应所需的全部逻辑，并以尽可能模块化的方式实现。另一个相关的概念是，输出解析器可以重试和修复错误。如果有格式错误的模式，你可以通过将输出和错误传递给它来显式地修复响应。

记住之前的步骤也是很重要的。最基本的方法是在内存中保留这些步骤的列表。然而，在处理长时间运行的任务时，会遇到一些上下文窗口的问题。我们已经找到了一种解决方法，即使用一些检索方法来获取之前的步骤，并将其放入上下文中。

如果是在处理接口文档时，比如发起一个接口请求，返回了整个网页内容，这样会导致观察结果太长的问题。因为接口文档通常会返回非常大且难以放入上下文的 JSON 数据。常见的解决方法是对其进行解析，可以简单地将该大数据块转换为字符串，并将前 1000 个字符作为响应。

Agent 明星项目

2023年的 Agent 应用项目研发涉猎广泛，主要集中在如何改善 Agent 的各种工作方式上。下面介绍四个具有代表性的项目。

AutoGPT：AutoGPT 的目标设置有别于 ReAct Agent 的重大不同。AutoGPT 的追求在于如何增加 Twitter 的关注者数量或实现其他类似的开放性、广泛性和长期性目标。相较之下，ReAct Agent 则专注于实现短期内可量化的目标。为了实现这样的目标，AutoGPT 引入了长期记忆的概念，促进 Agent 与工具之间的互动，这有助于提升 Agent 的规划和执行效率。

Baby AGI：Baby AGI 的研发采用了逐步解决子问题的方法，以提升 Agent 的规划和执行能力。这一项目明确了策划和执行步骤的定义，这一创新为提升长期目标 Agent 的可行性和关注度提供了有益的思考途径。最初，Baby AGI 的策略实现主要依靠自主设定，然而现在已经开始融入了各种工具，从而优化 Agent 执行计划的能力。

Camel：Camel 项目的一项主要创新是在模拟环境中进行 Agent 之间的交互。通过这种方法，可以对 Agent 进行评估和测试，并且可以作为一种娱乐手段。这种方法为检测 Agent 交互提供了一种无需人工干预的方式，能够有效地测试 Agent 模型。

Generative Agents：该项目的目标是通过构建一个复杂的模拟环境，让 25 个不同的 Agent 在这个环境中进行互动，从而实现生成型 Agent 。项目同时也注重处理 Agent 的记忆和反思能力， Agent 能够通过记忆中的事件来指导下一步的行动，并在反思环节对最近的事件进行评估和更新。这种基于反思的状态更新机制适用于各种类型的记忆，例如实体记忆和知识图谱，从而提高 Agent 对环境的建模能力。

### 7.1.4 Agent 的入门示例

安装openai 和 langchain库。

pip -q install openai

pip install git+https://github.com/hwchase17/langchain

设置设置谷歌搜索的API密钥，设置OpenAI 的密钥。

os.environ["OPENAI\_API\_KEY"] = ""

os.environ["SERPAPI\_API\_KEY"] = ""

首先，让我们加载大语言模型。

from langchain.agents import load\_tools

from langchain.agents import initialize\_agent

from langchain.agents import AgentType

from langchain.llms import OpenAI

llm = OpenAI(temperature=0)

接下来，让我们加载一些要使用的工具。请注意，llm-math 工具使用了一个大语言模型接口，所以我们需要传递进去。

tools = load\_tools(["serpapi", "llm-math"], llm=llm)

最后，让我们用这些工具、大语言模型和我们想要使用的 Agent 类型来初始化一个 Agent。

agent=

initialize\_agent(tools, llm, agent=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION, verbose=True)

现在，让我们来测试一下吧！

agent.run("Who is Leo DiCaprio’s girlfriend? What is her current age raised to the 0.43 power? ")

看 AgentExecutor 链组件运行的结果，重点是观察 Observation 、Action 、Answer以及 Thought的变化。这就是Agent 在回答雷纳尔多所经历的中间步骤。这个中间步骤执行的便是 ReAct 框架的策略。

Entering new AgentExecutor chain… I need to find out who Leo DiCaprio’s girlfriend is and then calculate her age raised to the 0.43 power.

Action: Search Action Input: “Leo DiCaprio girlfriend”

Observation: Camila Morrone Thought: I need to find out Camila Morrone’s age Action: Search Action Input: “Camila Morrone age” Observation: 25 years

Thought: I need to calculate 25 raised to the 0.43 power Action: Calculator Action Input: 25^0.43

Observation:

Answer: 3.991298452658078

Thought: I now know the final answer Final

Answer: Camila Morrone is Leo DiCaprio’s girlfriend and her current age raised to the 0.43 power is 3.991298452658078.

Finished chain.

“Camila Morrone is Leo DiCaprio’s girlfriend and her current age raised to the 0.43 power is 3.991298452658078.”

### 7.1.5 Agent 的类型

根据任务的不同， Agent 主要有两种类型：行动 Agent 和计划执行 Agent 。

行动 Agent ：每个时间步都会根据前面所有动作的输出来决定下一步的行动。行动 Agent 适合于小型任务，它的优点在于能够实时地处理信息和作出决策。

计划执行 Agent ：首先决定全部行动序列，然后一次性执行所有的动作，而不更新计划。计划执行 Agent 更适合于需要保持长期目标和重点的复杂或长期运行的任务。

值得注意的是，常常将这两种 Agent 结合起来使用是最佳的做法，也就是说，让计划执行 Agent 使用行动 Agent 来执行计划。这样的结合，既保持了行动 Agent 的动态性，又利用了计划执行 Agent 的规划能力。

行动 Agent

Agent 利用语言大模型决定要执行哪些行动以及行动的顺序。行动可以是使用工具并观察其输出，也可以是向用户返回响应。

在深入了解各种具体的 Agent 类型之前，我们先看一下源代码中定义的 Agent 类型枚举类。这个枚举类列出了 LangChain 框架中所有可用的 Agent 类型。有了这个概念，我们就可以更好地理解以下将要介绍的各种 Agent 类型以及它们的使用场景。

class AgentType(str, Enum):  
 ZERO\_SHOT\_REACT\_DESCRIPTION = "zero-shot-react-description"  
 REACT\_DOCSTORE = "react-docstore"  
 SELF\_ASK\_WITH\_SEARCH = "self-ask-with-search"  
 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"

这些 Agent 类型是根据不同的理论依据和实践需求创建的，它们已经在源代码中实现，我们可以直接使用，无需自己从头开始。这些内置的 Agent 类型提供了一个丰富的工具集，可以满足大部分的使用场景。接下来，我们将根据 Agent 类型按照原理进行分类，并且了解它们不同的使用场景。

Zero-shot ReAct 这种 Agent 使用 ReAct 框架，仅根据工具的描述决定使用哪个工具。可以提供任意数量的工具。这种 Agent 要求为每个工具提供一个描述。

结构化输入反应（Structured input ReAct） 结构化工具聊天 Agent 能够使用多输入工具。旧 Agent 被配置为将行动输入指定为单个字符串，但是这种 Agent 可以使用工具的参数模式创建结构化的行动输入。这对于更复杂的工具使用，例如精确导航浏览器，非常有用。

OpenAI 函数（OpenAI Functions） OpenAI 模型（如 gpt-3.5-turbo-0613 和 gpt-4-0613）被明确地微调，以便检测何时应该调用函数，并响应应传递给函数的输入。OpenAI 函数 Agent 旨在与这些模型一起工作。注意目前支持的型号有限，不适用默认的模型型号，也不适用其他的模型平台的型号。

对话（Conversational） 这种 Agent 被设计用于对话设置。提示被设计为使 Agent 有帮助和交谈。它使用 ReAct 框架决定使用哪个工具，并使用记忆来记住以前的对话交互。

自问与搜索（Self ask with search） 这种 Agent 使用一个名为 Intermediate Answer 的工具。这个工具应该能够查找问题的事实答案。这个 Agent 相当于原始的 self ask with search paper 论文，其中提供了 Google 搜索 API 作为工具。

ReAct 文档存储（ReAct document store） 这种 Agent 使用 ReAct 框架与文档存储进行交互。必须提供两个工具：Search 工具和 Lookup 工具（必须精确地命名为这样）。搜索工具应该搜索一个文档，而查找工具应该在最近找到的文档中查找一个词条。这个 Agent 等同于原始的 ReAct 论文。

了解这些理论和分类的真正价值在于我们的实际应用。如果我们需要创建的 Agent 已经存在于内置类型中，那我们完全可以直接使用，无需自己从头开始实现。但是，如果我们需要自定义 Agent，那么这些分类就成了我们做决策的关键因素。

在开始创建自定义 Agent 时，我们首先需要明确这个 Agent 需要承担什么类型的任务。然后，我们需要考虑哪种类型的 Agent 能够最好地完成这个任务。在这个过程中，我们需要查看是否已经有现成的解决方案可以参考或者直接使用。

这就是为什么我们需要了解这些内置的 Agent 类型和它们各自的用途。这些内置的 Agent 类型不仅提供了丰富的选择，满足了大部分的应用场景，而且它们也为我们自定义 Agent 提供了宝贵的参考。

这些内置的 Agent 类型，例如 Zero-shot ReAct、结构化输入反应、OpenAI 函数、对话、自问与搜索以及 ReAct 文档存储，都有各自的特点和使用场景。通过对这些类型的理解和学习，我们可以更好地利用 LangChain 框架，更有效地创建和使用 Agent。

计划并执行 Agent

紧接着行动 Agent 的类型，让我们继续了解计划并执行 Agent 的类型。

计划并执行 Agent （Plan-and-Solve Agents）通过首先规划要做什么，然后执行子任务来实现目标。这个思想很大程度上受到 BabyAGI 以及 “Plan-and-Solve” 论文的启发。目前这种类型的 Agent ，比如 BabyAGI 都处于项目的雏形，LangChain 还没有这种内置Agent 的类型。

每种 Agent 类型都有其特定的用途和应用场景， Agent 的灵活性和丰富性为 LangChain 提供了强大的功能性。

上文所提到的论文资源：

ReAct 论文: https://arxiv.org/pdf/2205.00445.pdf

BabyAGI 仓库： https://github.com/yoheinakajima/babyagi

Plan-and-Solve 论文： https://arxiv.org/abs/2305.04091

self ask with search 论文：<https://ofir.io/self-ask.pdf>

## 7.2 Agent的应用实践

在 LangChain 框架中，Agent 模块实现了多种类型的 Agent，比如 ZeroShotAgent，OpenAIFunctionsAgent。另外框架鼓励开发者创建自己的 Agent。理解这些 Agent 的使用步骤，以及如何自定义 Agent 都至关重要。

首先，我们需要明白创建和运行 Agent 是两个分离的步骤。创建 Agent 是通过实例化 Agent 类（如 ZeroShotAgent，或者是你创建的）来完成的。在创建 Agent 的过程中，Tools 也会被用于提示词模板，所以无论是Agent 还是AgentExecutor的初始化过程中，都需要设置 tools。

创建好 Agent 后，我们需要将其放入 AgentExecutor 中进行运行。AgentExecutor 是 Agent 的运行环境，它是一个链组件。我们实际上运行的是这个链组件，而不是 Agent 本身。在整个 Langchain 框架中，所有模块的终点都是组合到链组件上。Agent不能脱离 AgentExecutor环境，就像鱼离不开水。在运行过程中，AgentExecutor 会调用 Tools 的方法来执行具体的任务。

以下是一个使用 ZeroShotAgent 的示例：

from langchain.agents import ZeroShotAgent, Tool, AgentExecutor  
  
llm\_chain = LLMChain(llm=OpenAI(temperature=0), prompt=prompt)  
  
tool\_names = [tool.name for tool in tools]  
agent = ZeroShotAgent(llm\_chain=llm\_chain, allowed\_tools=tool\_names)  
  
agent\_executor = AgentExecutor.from\_agent\_and\_tools(  
 agent=agent, tools=tools, verbose=True  
)  
  
agent\_executor.run("How many people live in canada as of 2023?")

在这个示例中，我们首先创建了一个 ZeroShotAgent 实例，并将其放入了 AgentExecutor 中。然后，我们调用了 AgentExecutor 的 run 方法来运行这个 Agent，并获取了其运行结果。

这是目前最通用的 Agent 的实现方法。无论是自定义还是内置的 Agent 都遵循这个使用步骤。对于内置的 Agent 类型，还有一个简化方法，简化的原因是把之前的二个步骤，简化为了一个步骤，不需要创建 Agent 实例，也不需要显式创建 AgentExecutor：

agent

= initialize\_agent( tools, llm, agent=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION, verbose=True )

initialize\_agent 方法被广泛用于内置的Agent 中。比如我们刚刚使用的 agent=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION，实际上它对应的是 ZeroShotAgent。值得注意的是如果并不在这个清单内，这个方法并不适用。以下类型均可以使用这个方法：

{

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

}

下面我们通过一个实际的应用案例代码，展示一个完整的使用步骤。

### 7.2.1 多功能 Agent

设置 Agent

我们先安装要用到的库。

pip -q install langchain huggingface\_hub openai google-search-results tiktoken wikipedia

设置所需工具的密钥和大语言模型的密钥。

import os  
os.environ["OPENAI\_API\_KEY"] = ""  
os.environ["SERPAPI\_API\_KEY"] = ""

设置 Agent 的过程包含了两个主要步骤：加载 Agent 将使用的工具，然后用这些工具初始化 Agent。在代码示例中，我们首先初始化了一些基础设置，然后加载了两个工具：一个使用搜索 API 进行搜索的工具，以及一个可以进行数学运算的计算器工具。

加载工具和初始化 Agent。

from langchain.agents import load\_tools  
from langchain.agents import initialize\_agent  
from langchain.llms import OpenAI  
  
llm = OpenAI(temperature=0)

每个工具都有一个名称和描述，告诉我们它是用来做什么的。在我们的示例中，”serpapi”工具用于搜索，而”llm-math”工具则用于解决数学问题。这些工具内部有很多内容，包括模板和许多不同的 chains。

tools = load\_tools(["serpapi", "llm-math"], llm=llm)

初始化 Agent

一旦我们设置好了工具，我们就可以开始初始化 Agent。初始化 Agent 需要我们传入工具和语言模型，以及 Agent 的类型或风格。在我们的示例中，我们使用了零镜像反应性 Agent，这是基于一篇关于让语言模型采取行动并生成操作步骤的论文。

agent =

initialize\_agent(tools, llm, agent="zero-shot-react-description", verbose=True)

Agent 的提示词模板

初始化 Agent 的重要步骤之一是设置提示词模板。这些提示会在 Agent 开始运行时提示大语言模型，告诉它应该做什么。

agent.agent.llm\_chain.prompt.template

在我们的示例中，我们为 Agent 设置了两个工具：搜索引擎和计算器。然后，我们设置了 Agent 应该返回的格式，这包括它需要回答的问题，以及它应该采取的行动和行动的输入。

'Answer the following questions as best you can. You have access to the following tools:\n\nSearch: A search engine. Useful for when you need to answer questions about current events. Input should be a search query.\nCalculator: Useful for when you need to answer questions about math.\n\nUse the following format:\n\nQuestion: the input question you must answer\nThought: you should always think about what to do\nAction: the action to take, should be one of [Search, Calculator]\nAction Input: the input to the action\nObservation: the result of the action\n... (this Thought/Action/Action Input/Observation can repeat N times)\nThought: I now know the final answer\nFinal Answer: the final answer to the original input question\n\nBegin!\n\nQuestion: {input}\nThought:{agent\_scratchpad}'

Agent 的运行

最后，我们运行 Agent。需要注意的是，Agent 并不总是需要使用工具。在我们的示例中，我们问 Agent “你今天好吗？”。对于这样的问题，Agent 并不需要进行搜索或计算，而是可以直接生成回答。

agent.run("Hi How are you today?")

这就是 Langchain Agents 的基本概念和使用方法。

使用 Math 模块

我们在前半部分介绍了 Agent 的基础知识和功能。现在，我们要继续探讨如何在实际中应用 Agent，以及在什么情况下，Agent 可能遇到的问题。

agent.run("Where is DeepMind's office?")

在我们的示例中，我们尚未使用到 math 模块，让我们来看一下它的作用。我们让 Agent 查找 Deep Mind 的街道地址的数字，然后进行平方。

agent.run("If I square the number for the street address of DeepMind what answer do I get?")

Agent 首先进行搜索获取地址，然后找到了数字 5（假设为地址的一部分），最后进行平方运算，得出结果 25。然而，如果问题中包含多个数字，Agent 可能会对哪个数字进行平方产生混淆，这就是一些可能需要考虑和解决的问题。

> Entering new AgentExecutor chain...  
 I need to find the street address of DeepMind first.  
Action: Search  
Action Input: "DeepMind street address"  
Observation: DeepMind Technologies Limited, is a company organised under the laws of England and Wales, with registered office at 5 New Street Square, London, EC4A 3TW (“DeepMind”, “us”, “we”, or “our”). DeepMind is a wholly owned subsidiary of Alphabet Inc. and operates https://deepmind.com (the “Site”).  
Thought: I now need to calculate the square of the street address.  
Action: Calculator  
Action Input: 5^2  
Observation: Answer: 25  
Thought: I now know the final answer.  
Final Answer: 25  
  
> Finished chain.  
'25'

使用终端工具

在我们的工具库中，还有一个我们还未使用过的工具，那就是终端工具。例如，我们可以问 Agent 当前目录中有哪些文件。

agent.run("What files are in my current directory?")

Agent 将运行一个 LS 命令来查看文件夹，并返回一个文件列表。

> Entering new AgentExecutor chain...  
 I need to find out what files are in my current directory.  
Action: Terminal  
Action Input: ls  
Observation: sample\_data  
  
Thought: I need to find out more information about this file.  
Action: Terminal  
Action Input: ls -l sample\_data  
Observation: total 55504  
-rwxr-xr-x 1 root root 1697 Jan 1 2000 anscombe.json  
-rw-r--r-- 1 root root 301141 Mar 10 20:51 california\_housing\_test.csv  
-rw-r--r-- 1 root root 1706430 Mar 10 20:51 california\_housing\_train.csv  
-rw-r--r-- 1 root root 18289443 Mar 10 20:51 mnist\_test.csv  
-rw-r--r-- 1 root root 36523880 Mar 10 20:51 mnist\_train\_small.csv  
-rwxr-xr-x 1 root root 930 Jan 1 2000 README.md  
  
Thought: I now know the final answer.  
Final Answer: The files in my current directory are anscombe.json, california\_housing\_test.csv, california\_housing\_train.csv, mnist\_test.csv, mnist\_train\_small.csv, and README.md.  
  
> Finished chain.  
'The files in my current directory are anscombe.json, california\_housing\_test.csv, california\_housing\_train.csv, mnist\_test.csv, mnist\_train\_small.csv, and README.md.'

我们甚至可以问它特定的文件是否存在，比如询问当前目录中是否有一个关于加利福尼亚的文件。

agent.run("Does my current directory have a file about California?")

Agent 会使用 grep 命令进行检索，并向我们报告结果。然而，这个过程可能需要多次调用语言模型，从而产生一定的运行成本。

> Entering new AgentExecutor chain...  
 I need to find out what files are in my current directory.  
Action: Terminal  
Action Input: ls  
Observation: sample\_data  
  
Thought: I need to find out if there is a file about California in this directory.  
Action: Terminal  
Action Input: grep California sample\_data  
Observation: Command 'grep California sample\_data' returned non-zero exit status 2.  
Thought: I need to look for a different way to search for the file.  
Action: Search  
Action Input: "California file in current directory"  
Observation: The pwd command displays the full, absolute path of the current, or working, directory. It's not something you'll use all the time, but it can ...  
Thought: I need to use the pwd command to find the file.  
Action: Terminal  
Action Input: pwd  
Observation: /content  
  
Thought: I need to use the ls command to find the file.  
Action: Terminal  
Action Input: ls  
Observation: sample\_data  
  
Thought: I need to use the grep command to find the file.  
Action: Terminal  
Action Input: grep California sample\_data  
Observation: Command 'grep California sample\_data' returned non-zero exit status 2.  
Thought: I need to look for a different way to search for the file.  
Action: Search  
Action Input: "California file in current directory"  
Observation: The pwd command displays the full, absolute path of the current, or working, directory. It's not something you'll use all the time, but it can ...  
Thought: I need to use the find command to find the file.  
Action: Terminal  
Action Input: find . -name "\*California\*"  
Observation: Command 'find . -name "\*California\*' returned non-zero exit status 2.  
Thought: I need to use the grep command to find the file.  
Action: Terminal  
Action Input: grep -R California \*  
Observation: sample\_data/README.md:\* `california\_housing\_data\*.csv` is California housing data from the 1990 US  
  
Thought: I now know the final answer.  
Final Answer: Yes, there is a file about California in the current directory.  
  
> Finished chain.  
'Yes, there is a file about California in the current directory.'

注意事项

使用终端工具时，需要非常谨慎。你不希望最终用户能够通过运行终端命令来操作你的文件系统，因此在添加这个工具时，需要确保适当的安全防护措施已经到位。不过，尽管有其潜在风险，但在某些情况下，使用终端工具还是很有帮助的，比如当你需要设置某些功能时。

以上就是 Agent 的使用示例和注意事项。

### 7.2.2 自定义 Agent

这一节我们介绍如何创建自定义Agent 。

一个Agent由两部分组成。tools：代理可以使用的工具。AgentExecutor：决定采取哪种行动。

我们将逐步介绍如何创建一个自定义 Agent。Tool，AgentExecutor，BaseSingleActionAgent 是从 langchain.agents 模块导入的类，用于创建自定义的 Agent 和 tools。OpenAI 和 SerpAPIWrapper 是从 langchain 模块导入的类，用于访问 OpenAI 的功能和 SerpAPI 的包。先安装库。

pip -q install openai  
pip install git+https://github.com/hwchase17/langchain

设置密钥。

# 设置OpenAI的API密钥  
os.environ["OPENAI\_API\_KEY"] = ""  
# 设置谷歌搜索的API密钥  
os.environ["SERPAPI\_API\_KEY"] = ""

from langchain.agents import Tool, AgentExecutor, BaseSingleActionAgent  
from langchain import OpenAI, SerpAPIWrapper

创建一个 SerpAPIWrapper 实例，然后将其 run 方法封装到一个 Tool 对象中。

search = SerpAPIWrapper()  
tools = [  
 Tool(  
 name="Search",  
 func=search.run,  
 description="useful for when you need to answer questions about current events",  
 return\_direct=True,  
 )  
]

定义了一个自定义的 Agent 类 FakeAgent，这个类从 BaseSingleActionAgent 继承。该类定义了两个方法 plan 和 aplan，这两个方法是 Agent 根据给定的输入和中间步骤来决定下一步要做什么的核心逻辑。

from typing import List, Tuple, Any, Union  
from langchain.schema import AgentAction, AgentFinish  
  
  
class FakeAgent(BaseSingleActionAgent):  
 """Fake Custom Agent."""  
  
 @property  
 def input\_keys(self):  
 return ["input"]  
  
 def plan(  
 self, intermediate\_steps: List[Tuple[AgentAction, str]], \*\*kwargs: Any  
 ) -> Union[AgentAction, AgentFinish]:  
 """Given input, decided what to do.  
  
 Args:  
 intermediate\_steps: Steps the LLM has taken to date,  
 along with observations  
 \*\*kwargs: User inputs.  
  
 Returns:  
 Action specifying what tool to use.  
 """  
 return AgentAction(tool="Search", tool\_input=kwargs["input"], log="")  
  
 async def aplan(  
 self, intermediate\_steps: List[Tuple[AgentAction, str]], \*\*kwargs: Any  
 ) -> Union[AgentAction, AgentFinish]:  
 """Given input, decided what to do.  
  
 Args:  
 intermediate\_steps: Steps the LLM has taken to date,  
 along with observations  
 \*\*kwargs: User inputs.  
  
 Returns:  
 Action specifying what tool to use.  
 """  
 return AgentAction(tool="Search", tool\_input=kwargs["input"], log="")

创建了一个 FakeAgent 的实例。

agent = FakeAgent()

创建了一个 AgentExecutor 的实例，该实例将使用前面定义的 FakeAgent 和 tools 来执行任务。from\_agent\_and\_tools 是一个类方法，用于创建AgentExecutor 的实例。

agent\_executor = AgentExecutor.from\_agent\_and\_tools(  
 agent=agent, tools=tools, verbose=True  
)

调用 AgentExecutor 的 run 方法来执行一个任务，任务是查询 “2023 年加拿大有多少人口”。

agent\_executor.run("How many people live in canada as of 2023?")

打印最终的结果。

> Entering new AgentExecutor chain...  
 The current population of Canada is 38,669,152 as of Monday, April 24, 2023, based on Worldometer elaboration of the latest United Nations data.  
   
 > Finished chain.  
  
 'The current population of Canada is 38,669,152 as of Monday, April 24, 2023, based on Worldometer elaboration of the latest United Nations data.'

### 7.2.3 ReAct Agent 的实践

由于 ReAct 框架的特性，目前它已经成为首选的 Agent 实现方式。 Agent 的基本理念是将大语言模型当作推理的引擎。ReAct 框架实际上是把推理和动作结合在一起。当 Agent 接收到用户的请求后，大语言模型就会帮助选择使用哪个工具。接着， Agent 会执行该工具的操作，观察其结果，并把这些结果反馈给大语言模型。

下面用代码演示了如何使用 Agent 实现 ReAct 框架。

首先，让我们加载将用于控制 Agent 的 openai 和 langchain 的库。

pip -q install openai  
pip install git+https://github.com/hwchase17/langchain

设置密钥。

# 设置OpenAI的API密钥  
os.environ["OPENAI\_API\_KEY"] = ""  
# 设置谷歌搜索的API密钥  
os.environ["SERPAPI\_API\_KEY"] = ""

from langchain.agents import load\_tools  
from langchain.agents import initialize\_agent  
from langchain.agents import AgentType  
from langchain.llms import OpenAI  
llm = OpenAI(temperature=0)

接着，我们需要加载一些工具。

tools = load\_tools(["serpapi", "llm-math"], llm=llm)

请注意，llm-math 工具使用了 llm，因此我们需要输入这个模型。最后，我们需要使用 tools、llm 和我们想要使用的 Agent 类型 ZERO\_SHOT\_REACT\_DESCRIPTION 来初始化一个 Agent 。

agent =

initialize\_agent(tools, llm, agent=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION, verbose=True)

现在让我们来测试一下！

agent.run("Who is Leo DiCaprio's girlfriend? What is her current age raised to the 0.43 power?")

> Entering new AgentExecutor chain...  
 I need to find out who Leo DiCaprio's girlfriend is and then calculate her age raised to the 0.43 power.  
 Action: Search  
 Action Input: "Leo DiCaprio girlfriend"  
 Observation: Camila Morrone  
 Thought: I need to find out Camila Morrone's age  
 Action: Search  
 Action Input: "Camila Morrone age"  
 Observation: 25 years  
 Thought: I need to calculate 25 raised to the 0.43 power  
 Action: Calculator  
 Action Input: 25^0.43  
 Observation: Answer: 3.991298452658078  
   
 Thought: I now know the final answer  
 Final Answer: Camila Morrone is Leo DiCaprio's girlfriend and her current age raised to the 0.43 power is 3.991298452658078.  
   
 > Finished chain.  
  
  
 "Camila Morrone is Leo DiCaprio's girlfriend and her current age raised to the 0.43 power is 3.991298452658078."

除此之外，你还可以创建使用 Chat Model 模型包装器作为 Agent 驱动器的 ReAct Agent ，而不是使用 LLM 模型包装器。

from langchain.chat\_models import ChatOpenAI  
  
chat\_model = ChatOpenAI(temperature=0)  
agent = initialize\_agent(tools, chat\_model, agent=AgentType.CHAT\_ZERO\_SHOT\_REACT\_DESCRIPTION, verbose=True)  
agent.run("Who is Leo DiCaprio's girlfriend? What is her current age raised to the 0.43 power?")

## 7.3 工具和工具包

在 Agent 模块中，工具（Tools) 是 Agent 用来与世界互动的接口。这些工具实际上就是 Agent 可以使用的函数，用于与外部世界进行交互。这些工具可以是通用的实用程序（例如搜索功能），也可以是其他的工具链，甚至是其他的 Agent 。

工具包（Toolkits）是设计用于完成特定任务的工具集合，它们具有方便的加载方法。工具包将一组具有共同目标或特性的工具集中在一起，提供统一而便捷的使用方式，使得用户能够更加方便地完成特定任务。

在构建自己的 Agent 时，你需要提供一个工具列表，这些工具是 Agent 可以使用的。除了实际被调用的函数外（func=search.run），工具还包括一些组成部分：name（必需的，并且在提供给 Agent 的工具集中必须是唯一的）；description（可选的，但建议提供，因为 Agent 会用它来判断工具的使用情况）。

from langchain.agents import ZeroShotAgent, Tool, AgentExecutor

from langchain import OpenAI, SerpAPIWrapper, LLMChain

search = SerpAPIWrapper()

tools = [ Tool( name=“Search”, func=search.run, description=“useful for when you need to answer questions about current events”, )]

工具包是为特定任务而设计的工具集合，具有便捷的加载方法。它们可以被设计为一起使用，以完成特定的任务。这种设计方式提供了一种更为便捷和高效的方式来处理复杂的任务，提升了工作效率。

LangChain 封装了许多类型的工具和工具包，用户可以随时调用这些工具，完成各种复杂的任务。除了使用 LangChain 提供的工具，用户也可以自定义工具，形成自己的工具包，以完成特殊的任务。

### 7.3.1 工具的类型

Langchain 提供了一系列的工具，它们封装了各种功能，可以直接在你的项目中使用。这些工具涵盖了从数据处理到网络请求，从文件操作到数据库查询，从搜索引擎查询到大语言模型的应用，等等。这个工具清单还在不断地扩展和更新。以下是目前可用的工具列表：

“AIPluginTool”: 一个插件工具，允许用户将其他的人工智能模型或服务集成到系统中。

“APIOperation”: 用于调用外部 API 的工具。

“ArxivQueryRun”: 用于查询 Arxiv 的工具。

“AzureCogsFormRecognizerTool”: 利用 Azure 认知服务中的表单识别器的工具。

“AzureCogsImageAnalysisTool”: 利用 Azure 认知服务中的图像分析的工具。

“AzureCogsSpeech2TextTool”: 利用 Azure 认知服务中的语音转文本的工具。

“AzureCogsText2SpeechTool”: 利用 Azure 认知服务中的文本转语音的工具。

“BaseGraphQLTool”: 用于发送 GraphQL 查询的基础工具。

“BaseRequestsTool”: 用于发送 HTTP 请求的基础工具。

“BaseSQLDatabaseTool”: 用于与 SQL 数据库交互的基础工具。

“BaseSparkSQLTool”: 用于执行 Spark SQL 查询的基础工具。

“BingSearchResults”: 用于获取 Bing 搜索结果的工具。

“BingSearchRun”: 用于执行 Bing 搜索的工具。

“BraveSearch”: 用于执行 Brave 搜索的工具。

“ClickTool”: 模拟点击操作的工具。

“CopyFileTool”: 用于复制文件的工具。

“CurrentWebPageTool”: 用于获取当前网页信息的工具。

“DeleteFileTool”: 用于删除文件的工具。

“DuckDuckGoSearchResults”: 用于获取 DuckDuckGo 搜索结果的工具。

“DuckDuckGoSearchRun”: 用于执行 DuckDuckGo 搜索的工具。

“ExtractHyperlinksTool”: 用于从文本或网页中提取超链接的工具。

“ExtractTextTool”: 用于从文本或其他源中提取文本的工具。

“FileSearchTool”: 用于搜索文件的工具。

“GetElementsTool”: 用于从网页或其他源中获取元素的工具。

“GmailCreateDraft”: 用于创建 Gmail 草稿的工具。

“GmailGetMessage”: 用于获取 Gmail 消息的工具。

“GmailGetThread”: 用于获取 Gmail 线程的工具。

“GmailSearch”: 用于搜索 Gmail 的工具。

“GmailSendMessage”: 用于发送 Gmail 消息的工具。

“GooglePlacesTool”: 用于搜索 Google Places 的工具。

“GoogleSearchResults”: 用于获取 Google 搜索结果的工具。

“GoogleSearchRun”: 用于执行 Google 搜索的工具。

“GoogleSerperResults”: 用于获取 Google SERP（搜索引擎结果页面）的工具。

“GoogleSerperRun”: 用于执行 Google SERP 查询的工具。

“HumanInputRun”: 用于模拟人类输入的工具。

“IFTTTWebhook”: 用于触发 IFTTT（如果这个，那么那个）Webhook 的工具。

“InfoPowerBITool”: 用于获取 PowerBI 信息的工具。

“InfoSQLDatabaseTool”: 用于获取 SQL 数据库信息的工具。

“InfoSparkSQLTool”: 用于获取 Spark SQL 信息的工具。

“JiraAction”: 用于在 Jira 上执行操作的工具。

“JsonGetValueTool”: 用于从 JSON 数据中获取值的工具。

“JsonListKeysTool”: 用于列出 JSON 数据中的键的工具。

“ListDirectoryTool”: 用于列出目录内容的工具。

“ListPowerBITool”: 用于列出 PowerBI 信息的工具。

“ListSQLDatabaseTool”: 用于列出 SQL 数据库信息的工具。

请注意，这些工具的具体实现和功能可能会根据实际的需求和环境进行调整。

### 7.3.2 工具包的类型

Langchain 提供了一系列与各种 Agent 进行交互的工具包组件和 内置Agent ，以帮助我们快速建立解决各种问题的 Agent。create 为前缀的Agent，例如create\_json\_agent，均已配置了一系列的工具函数，调用这个函数即可运行一个 AgentExecutor 实例，被很多工具包组件中使用。本质上来说，这种方式创建的Agent，最终是作为工具包组件中的一个工具进行使用的。在这种设计模式下，开发者不需要深入了解每个Agent的内部实现细节，只需调用相应的 create 函数即可轻松获取他们需要的Agent。多数采用这种方式的是一些工具包的实例。比如，我们使用 create\_json\_agent 函数创建了一个 json\_agent，然后用它和 requests\_wrapper 来初始化 OpenAPIToolkit 类。开发者初始化 OpenAPIToolkit 类的时候，创建了一个用于创建处理 JSON 数据的 Agent。这对于开发者创建个性化的工具包组件非常有帮助。

从使用角度来说，这些create 为前缀的Agent发挥的是工具集合的作用。与其说这些 Agent 是代理，不如说它们是一组工具的集合。所以create 为前缀的Agent和其他工具包放在一块展示。在以下是目前可用的工具包和 Agent列表：

“create\_json\_agent”: 一个设计用于与 JSON 数据交互的Agent。

“create\_sql\_agent”: 一个设计用于与 SQL 数据库交互的 Agent。

“create\_openapi\_agent”: 一个设计用于与 OpenAPI 交互的 Agent。

“create\_pbi\_agent”: 一个设计用于与 Power BI 交互的 Agent。

“create\_vectorstore\_router\_agent”: 一个设计用于与 Vector Store 路由交互的 Agent。

“create\_pandas\_dataframe\_agent”: 一个设计用于与 Pandas 数据帧交互的 Agent。

“create\_spark\_dataframe\_agent”: 一个设计用于与 Spark 数据帧交互的 Agent。

“create\_spark\_sql\_agent”: 一个设计用于与 Spark SQL 交互的 Agent。

“create\_csv\_agent”: 一个设计用于与 CSV 文件交互的 Agent。

“create\_pbi\_chat\_agent”: 一个设计用于与 Power BI 聊天交互的 Agent。

“create\_python\_agent”: 一个设计用于与 Python 交互的 Agent。

“create\_vectorstore\_agent”: 一个设计用于与 Vector Store 交互的 Agent。

“JsonToolkit”: 一个用于处理 JSON 数据的工具包。

“SQLDatabaseToolkit”: 一个用于处理 SQL 数据库的工具包。

“SparkSQLToolkit”: 一个用于处理 Spark SQL 的工具包。

“NLAToolkit”: 一个用于处理自然语言应用的工具包。

“PowerBIToolkit”: 一个用于处理 Power BI 应用的工具包。

“OpenAPIToolkit”: 一个用于处理 OpenAPI 的工具包。

“VectorStoreToolkit”: 一个用于处理 Vector Store 的工具包。

“VectorStoreInfo”: 一个用于获取 Vector Store 信息的工具。

“VectorStoreRouterToolkit”: 一个用于处理 Vector Store 路由的工具包。

“ZapierToolkit”: 一个用于处理 Zapier 应用的工具包。

“GmailToolkit”: 一个用于处理 Gmail 应用的工具包。

“JiraToolkit”: 一个用于处理 Jira 应用的工具包。

“FileManagementToolkit”: 一个用于文件管理的工具包。

“PlayWrightBrowserToolkit”: 一个用于处理 PlayWright 浏览器的工具包。

“AzureCognitiveServicesToolkit”: 一个用于处理 Azure 认知服务的工具包。

这些工具包的具体功能和实现可能会根据实际的需求和环境进行调整。

## 7.4 Agent 的能力增强

随着技术的持续发展，Agent 的功能正在经历深刻的变革和完善。在这里，我们将重点探索几个关于 Agent 功能增强的核心方面。

记忆功能的扩展：为 OpenAI Functions Agent 引入记忆功能是一次重大的突破。这不仅让 Agent 记忆先前的对话内容，还使其在执行连续任务时表现得更为出色。例如，Agent 能从工具中提取三个质数，并进行相乘。在此过程中，Agent 还有能力验证中间结果，如确认其输出是否为质数。

与向量存储库的融合：为了让 Agent 更有效地与向量存储库互动，我们建议引入一个 RetrievalQA，并将其纳入整体 Agent 的工具集中。此外，Agent 还能与多个 vectordbs 进行交互，并在它们间实现路由。这使 Agent 在数据访问和处理方面具备更广的能力。

自定义 Agent 的探索：其中包括如何创建专属的 MRKL agent 以及如何构建个性化的多动作 agent。

Agent 的错误管理与优化：例如，当大语言模型面临决策困难时，我们可以借助 handle\_parsing\_errors 功能来处理解析错误。为了避免 Agent 进行过多的操作，我们可以设置迭代的最大次数。同样，为了防止 Agent 运行过久，可以设定一个执行的时间上限。

Agent 的高级特性：这包括了解 Agent 的中间操作过程以及如何为 Agent 及其工具添加记忆功能。

其他高级应用：我们探讨了如何仅流式传输 Agent 的最终输出以及如何将 OpenAI functions agent 与各种工具集成。

在这些提及的领域中，每一个都致力于提升 Agent 的功能和适用范围。但为了与实际应用更为契合，我们将尤其关注两个关键方面：为 Agent 增添记忆功能和与向量存储库的深度整合。这两个领域不仅代表了 Agent 技术的前沿趋势，同时也回应了现实中的关键需求。

### 7.4.1 增强 Agent 记忆能力

在本文中，我们将探讨如何为 OpenAI Functions Agent 添加记忆功能。OpenAI Functions Agent 是一个能够调用函数并响应函数输入的强大工具，但在默认配置下，它并不具备记忆之前对话内容的能力。然而，在许多实际应用场景中，我们希望 Agent 能够记住之前的对话内容，从而更好地为用户提供服务。

具体来说，我们将设计一个实验，通过这个实验，我们将测试大语言模型是否能记住之前的对话内容，并在后续的对话中正确使用这些信息。我们将从设计问题的角度和步骤开始，然后深入研究如何为 Agent 添加记忆功能，并测试这个功能是否能够满足我们的预期。

我们设计了一些问题，让我们创建的代理来回答。设计这个问题的目的是为了检测大语言模型是否具备记忆能力，是否可以像人类一样记住之前的对话内容。这在实际应用中极为重要，比如在聊天机器人的场景中，我们希望机器人能记住用户的名字和之前的对话内容，以便在后续的对话中为用户提供更个性化的服务。

设计这个问题的步骤如下：

1. 首先，我们通过调用  agent.run("hi")  启动一次会话。这是一个简单的问候，类似于人类的对话开场白。

2. 然后，我们通过  agent.run("my name is bob")  介绍一个名字。这是在告诉大语言模型，我们的名字是 Bob。这一步的目的是为了看是否大语言模型能记住这个信息。

3. 最后，我们通过  agent.run("whats my name")  询问大语言模型我们的名字。这是一个测试，看大语言模型是否记住了我们之前介绍的名字。

这个设计问题的角度主要是从记忆能力和对话能力出发，通过设计这样的对话流程，我们可以测试大语言模型在一次会话中是否能记住之前的信息，并在后续的对话中正确使用这些信息。

下面是为 OpenAI Functions Agent 添加记忆功能的代码示例：

首先从 langchain 模块中导入所需的各种类和函数。

LLMMathChain：这是一个类，用于创建一个能够进行数学计算的大语言模型（Large Language Model, LLM）链。LLM 链是一个特殊的模型，可以将一系列的工具和大语言模型连接起来，使得模型能够执行更复杂的任务。

OpenAI：这是一个类，用于创建一个 OpenAI 大语言模型。可以使用这个类来实例化一个 OpenAI 模型，并在后续的代码中使用它。

SerpAPIWrapper：这是一个类，用于创建一个 SerpAPI 的包装器。SerpAPI 是一个搜索引擎结果页面（Search Engine Results Page, SERP）的 API，可以使用这个类来实例化一个 SerpAPI 对象，并在后续的代码中使用它来进行搜索操作。

SQLDatabase 和 SQLDatabaseChain：这两个类用于创建和管理 SQL 数据库。SQLDatabase 是一个用于表示 SQL 数据库的类，可以使用它来实例化一个 SQL 数据库对象。SQLDatabaseChain 则是一个特殊的链，可以将 SQL 数据库和大语言模型连接起来，使得模型能够执行更复杂的数据库操作。

initialize\_agent 和 Tool：这两个函数用于初始化 Agent 和工具。initialize\_agent 函数可以用于创建一个 Agent，Tool 函数则用于创建一个工具。

AgentType：这是一个枚举类，定义了各种不同的 Agent 类型。可以使用这个类来指定要创建的 Agent 的类型。

ChatOpenAI：这是一个类，用于创建一个能够进行聊天的 OpenAI 模型。可以使用这个类来实例化一个聊天模型，并在后续的代码中使用它进行聊天操作。

from langchain import (

    LLMMathChain,

    OpenAI,

    SerpAPIWrapper,

    SQLDatabase,

    SQLDatabaseChain,

)

from langchain.agents import initialize\_agent, Tool

from langchain.agents import AgentType

from langchain.chat\_models import ChatOpenAI

创建一个工具（Tools）列表，其中每个工具都是一个 Tool 对象。Tool 对象主要包含三个属性：name，func 和 description。

llm = ChatOpenAI(temperature=0, model="gpt-3.5-turbo-0613")

search = SerpAPIWrapper()

llm\_math\_chain = LLMMathChain.from\_llm(llm=llm, verbose=True)

db = SQLDatabase.from\_uri("sqlite:///../../../../../notebooks/Chinook.db")

db\_chain = SQLDatabaseChain.from\_llm(llm, db, verbose=True)

tools = [

    Tool(

        name="Search",

        func=search.run,

        description="useful for when you need to answer questions about current events. You should ask targeted questions",

    ),

    Tool(

        name="Calculator",

        func=llm\_math\_chain.run,

        description="useful for when you need to answer questions about math",

    ),

    Tool(

        name="FooBar-DB",

        func=db\_chain.run,

        description="useful for when you need to answer questions about FooBar. Input should be in the form of a question containing full context",

    ),

]

为 Agent 添加记忆功能。在 langchain 的 Agent 框架中，记忆是由 ConversationBufferMemory 对象管理的, 根据的需求可以使用其他类型的记忆组件。

extra\_prompt\_messages：这是一个列表，用于包含一些额外的提示消息。这些消息将被添加到 Agent 的提示模板中。在这个例子中，添加了一个 MessagesPlaceholder 对象，这个对象表示一个占位符，它的值将在运行时被实际的记忆内容替换。

variable\_name：这是 MessagesPlaceholder 对象的参数，它表示占位符的变量名。在这个例子中，变量名为 "memory"，这意味着在提示模板中，{{memory}} 将被替换为实际的记忆内容。

memory：这是一个 ConversationBufferMemory 对象，用于管理 Agent 的记忆。在这个例子中，设置了 memory\_key 为 "memory"，这意味着记忆内容将存储在 "memory" 这个键下；并且设置了 return\_messages 为 True，这意味着记忆内容将包括返回的消息。

from langchain.prompts import MessagesPlaceholder

from langchain.memory import ConversationBufferMemory

agent\_kwargs = {

    "extra\_prompt\_messages": [MessagesPlaceholder(variable\_name="memory")],

}

memory = ConversationBufferMemory(memory\_key="memory", return\_messages=True)

在这段代码中，使用  initialize\_agent  函数创建了一个 Agent 实例，并为它配置了工具、大语言模型、Agent 类型、记忆组件等属性。

 tools ：这是一个  Tool  对象的列表，每个  Tool  对象都代表一个可以被 Agent 使用的工具。这些工具将被用来执行具体的任务，如搜索、计算等。

 llm ：这是之前创建的大语言模型包装器。Agent 将使用这个模型来生成语言输出，以及决定下一步的行动。

 AgentType.OPENAI\_FUNCTIONS ：这是一个枚举值，代表 Agent 的类型。在这个例子中，选择了  OPENAI\_FUNCTIONS  类型的 Agent，这种类型的 Agent 是为 OpenAI 函数模型特别设计的。

 verbose=True ：这个参数决定了是否在运行过程中打印详细的日志信息。如果设置为  True ，那么在每次 Agent 进行行动时，都会打印出详细的日志信息，这对于调试和理解 Agent 的行为非常有用。

 agent\_kwargs ：这是一个字典，用于传递额外的参数给 Agent 类。在这个例子中，传递了  extra\_prompt\_messages  参数，这个参数包含了额外的提示消息。

 memory ：这是之前创建的  ConversationBufferMemory  对象，用于管理 Agent 的记忆。这个对象将被用来存储和检索 Agent 的记忆内容。

通过这段代码，成功创建了一个具有记忆功能的 Agent。接下来，就可以使用这个 Agent 来进行对话，Agent 将能记住之前的对话内容，并可以根据这些记忆来做出决策。

agent = initialize\_agent(

    tools,

    llm,

    agent=AgentType.OPENAI\_FUNCTIONS,

    verbose=True,

    agent\_kwargs=agent\_kwargs,

    memory=memory,

)

启动一次会话。这是一个简单的问候，类似于人类的对话开场白。

agent.run("hi")

Agent 回答了我们。

    > Entering new  chain...

    Hello! How can I assist you today?

    > Finished chain.

    'Hello! How can I assist you today?'

我们通过  agent.run("my name is bob")  介绍一个名字。这是在告诉大语言模型，我们的名字是 Bob。这一步的目的是为了看是否大语言模型能记住这个信息。

agent.run("my name is bob")

    > Entering new  chain...

    Nice to meet you, Bob! How can I help you today?

    > Finished chain.

    'Nice to meet you, Bob! How can I help you today?'

询问大语言模型我们的名字。这是一个测试，看大语言模型是否记住了我们之前介绍的名字。

agent.run("whats my name")

我们看到大语言模型认识了我们之前介绍的名字：Bob

    > Entering new  chain...

    Your name is Bob.

    > Finished chain.

    'Your name is Bob.'

### 7.4.2 增强 Agent 检索能力

在本文中，我们将介绍如何将 Agent 与向量存储库进行结合。通过数据模块中的LEDVR工作流, 已经将数据导入到向量存储库中，如果希望以一种Agent的方式进行交互时，这种结合就显得尤为重要。

实现这一目标的推荐方法是首先创建一个 RetrievalQA 工具，然后将其作为整体 Agent 中的一个工具进行使用。

首先从 langchain 模块中导入所需的各种类和函数。

from langchain.embeddings.openai import OpenAIEmbeddings

from langchain.vectorstores import Chroma

from langchain.text\_splitter import CharacterTextSplitter

from langchain.llms import OpenAI

from langchain.chains import RetrievalQA

llm = OpenAI(temperature=0)

创建一个向量存储库。

from pathlib import Path

relevant\_parts = []

for p in Path(".").absolute().parts:

    relevant\_parts.append(p)

    if relevant\_parts[-3:] == ["langchain", "docs", "modules"]:

        break

doc\_path = str(Path(\*relevant\_parts) / "state\_of\_the\_union.txt")

from langchain.document\_loaders import TextLoader

loader = TextLoader(doc\_path)

documents = loader.load()

text\_splitter = CharacterTextSplitter(chunk\_size=1000, chunk\_overlap=0)

texts = text\_splitter.split\_documents(documents)

embeddings = OpenAIEmbeddings()

docsearch = Chroma.from\_documents(texts, embeddings, collection\_name="state-of-union")

实例化 RetrievalQA 链组件，运行链。

state\_of\_union = RetrievalQA.from\_chain\_type(

    llm=llm, chain\_type="stuff", retriever=docsearch.as\_retriever()

)

接着我们创建另外一个链，从在线网站上加载文档。

from langchain.document\_loaders import WebBaseLoader

loader = WebBaseLoader("https://beta.ruff.rs/docs/faq/")

docs = loader.load()

ruff\_texts = text\_splitter.split\_documents(docs)

ruff\_db = Chroma.from\_documents(ruff\_texts, embeddings, collection\_name="ruff")

ruff = RetrievalQA.from\_chain\_type(

    llm=llm, chain\_type="stuff", retriever=ruff\_db.as\_retriever()

)

正式开始创建 Agent , 导入创建  Agent 需要的所有类。

from langchain.agents import initialize\_agent, Tool

from langchain.agents import AgentType

from langchain.tools import BaseTool

from langchain.llms import OpenAI

from langchain import LLMMathChain, SerpAPIWrapper

定义可用的工具。

tools = [

    Tool(

        name="State of Union QA System",

        func=state\_of\_union.run,

        description="useful for when you need to answer questions about the most recent state of the union address. Input should be a fully formed question.",

    ),

    Tool(

        name="Ruff QA System",

        func=ruff.run,

        description="useful for when you need to answer questions about ruff (a python linter). Input should be a fully formed question.",

    ),

]

初始化一个 Agent。

agent = initialize\_agent(

    tools, llm, agent=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION, verbose=True

)

开始提问。测试 State of Union QA System 文档的检索。

agent.run(

    "What did biden say about ketanji brown jackson in the state of the union address?"

)

    > Entering new AgentExecutor chain...

     I need to find out what Biden said about Ketanji Brown Jackson in the State of the Union address.

    Action: State of Union QA System

    Action Input: What did Biden say about Ketanji Brown Jackson in the State of the Union address?

    Observation:  Biden said that Jackson is one of the nation's top legal minds and that she will continue Justice Breyer's legacy of excellence.

    Thought: I now know the final answer

    Final Answer: Biden said that Jackson is one of the nation's top legal minds and that she will continue Justice Breyer's legacy of excellence.

    > Finished chain.

    "Biden said that Jackson is one of the nation's top legal minds and that she will continue Justice Breyer's legacy of excellence."

再次提问。测试 ruff 文档的检索。

agent.run("Why use ruff over flake8?")

    > Entering new AgentExecutor chain...

     I need to find out the advantages of using ruff over flake8

    Action: Ruff QA System

    Action Input: What are the advantages of using ruff over flake8?

    Observation:  Ruff can be used as a drop-in replacement for Flake8 when used (1) without or with a small number of plugins, (2) alongside Black, and (3) on Python 3 code. It also re-implements some of the most popular Flake8 plugins and related code quality tools natively, including isort, yesqa, eradicate, and most of the rules implemented in pyupgrade. Ruff also supports automatically fixing its own lint violations, which Flake8 does not.

    Thought: I now know the final answer

    Final Answer: Ruff can be used as a drop-in replacement for Flake8 when used (1) without or with a small number of plugins, (2) alongside Black, and (3) on Python 3 code. It also re-implements some of the most popular Flake8 plugins and related code quality tools natively, including isort, yesqa, eradicate, and most of the rules implemented in pyupgrade. Ruff also supports automatically fixing its own lint violations, which Flake8 does not.

    > Finished chain.

    'Ruff can be used as a drop-in replacement for Flake8 when used (1) without or with a small number of plugins, (2) alongside Black, and (3) on Python 3 code. It also re-implements some of the most popular Flake8 plugins and related code quality tools natively, including isort, yesqa, eradicate, and most of the rules implemented in pyupgrade. Ruff also supports automatically fixing its own lint violations, which Flake8 does not.'

## 7.5 本章小结

在本章中，我们深入地探讨了 Agent 的定义、运行机制、现状与发展，以及如何入门。我们首先了解了 Agent 的基本定义和运行机制，包括其作为一个可以生成行动计划并执行的实体的角色，以及其如何通过调用工具和处理复杂情况来运行。我们还讨论了 Agent 的当前状态和未来发展方向，包括其在多种场景的应用以及在技术进步下的持续优化。

在实践部分，我们探索了多功能 Agent、自定义 Agent 以及 ReAct Agent 的使用。多功能 Agent 展示了 Agent 的强大能力，即在一次运行中完成多个任务；自定义 Agent 则让我们看到了用户可以根据自身需求定制 Agent 的可能性；而 ReAct Agent 则是一个强大的框架，能够帮助我们创建稳定的 Agent。

我们还介绍了工具和工具包的概念，以及它们在 Agent 中的重要角色。工具和工具包是 Agent 完成任务的重要组成部分，不同类型的工具和工具包可以帮助 Agent 应对不同类型的任务。

最后，我们讨论了如何增强 Agent 的能力，特别是如何增强其记忆能力和检索能力。增强记忆能力可以让 Agent 更好地处理连续任务，而增强检索能力则可以使 Agent 更好地利用已有的数据资源。