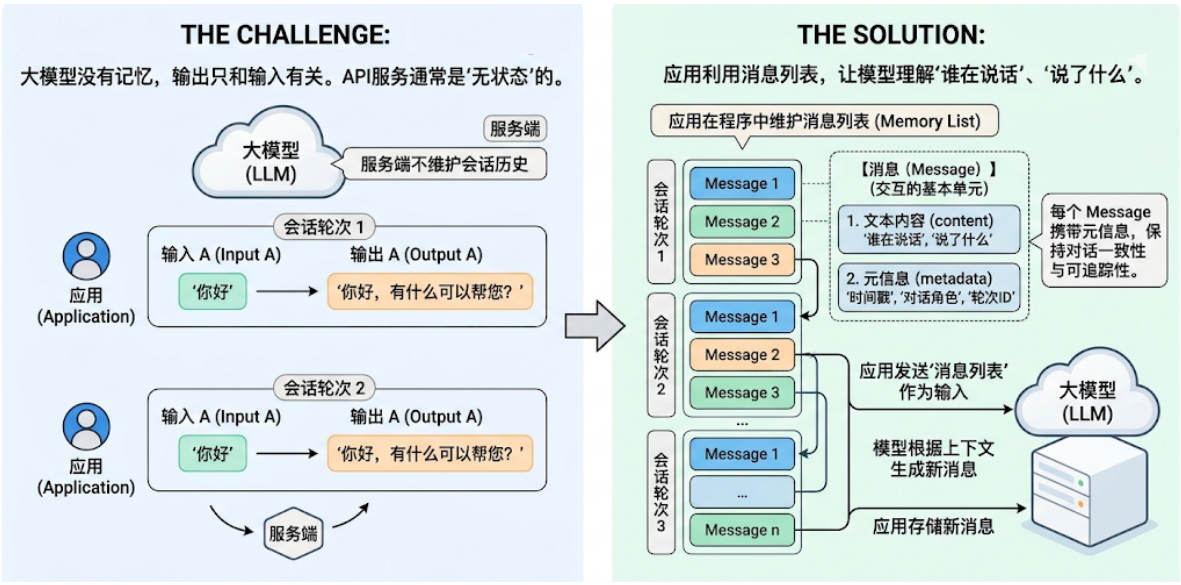
第04章：消息与提示词模板

讲师：尚硅谷-宋红康

官网：尚硅谷

1、认识消息

大模型没有记忆，它的输出只和输入模型的内容有关（上下文）。很多大模型API服务也没有在服务端维护会话历史，是“无状态”的。因此，**如果应用需要“记住”对话历史，需要在程序中维护消息列表。**



在 LangChain 中，Message（消息）是模型交互的最基本单元。它既代表模型接收到的输入（Input），也代表模型生成的输出（Output）。

> 每一轮与大模型的对话，都由一条或多条 Message 构成。每个 Message 不仅包含文字内容，还携带描述上下文状态的元信息（metadata），用于保持对话的一致性和可追踪性。比如，模型在多轮交互中理解“谁在说话”、“说了什么”、“这条信息属于哪一轮对话”。

LangChain 在 1.0 中提供了**跨模型统一的 Message 标准。**无论你使用的是 OpenAI、Anthropic、Gemini 还是本地模型，这一标准都能保持一致的行为。好处：

- 兼容性强：不同模型的消息格式自动对齐。
- 可扩展性高：方便添加多模态内容或自定义字段。
- 可追踪性好：为 LangSmith 等调试工具提供一致的上下文数据结构。

1.1 消息的内部结构

LangChain的消息（Message）对象包含三种字段

- **Role**：消息所属的角色或类型，如 system、user、assistant。
- **Content**：消息内容
- **Metadata**：（可选）元数据，存储额外信息。如：消息ID、响应时间、token消耗量、消息标签等

1.2 消息的类型

LangChain定义了很多消息类型，通过 `role` 区分。常用的有四种。

1、系统消息

也称为系统提示词，用于在对话开始时为模型设定角色、行为准则和上下文背景。它像是给AI助手的一份工作说明书，决定了其回答问题的风格、领域和专业范围。

```
{"role": "system", "content": "你是个精通编程的软件架构师"}
```

2、用户消息

也称为用户提示词，在多轮对话中，它表示用户的一次输入。可以包含简单的文本问题，也可以是复杂的多模态内容（如图片、音频、文档等）。

```
{"role": "user", "content": "你好啊~"}
```

3、助手(AI)消息

代表模型的回复，包括生成的文本、工具调用、元数据等。

```
{"role": "assistant", "content": "我也很高兴认识你"}
```

```
{
    "role": "assistant",
    "content": "",
    "tool_calls": [{
        "name": "get_weather",
        "args": {"location": "北京"},
        "id": "call_00_nUD2NC9QRN5Cg1GaoIkBJQ4s"
    }]
}
```

4、工具调用消息

工具调用结果匹配的消息类型。将此消息返回给模型，让模型基于这个结果继续生成回复。在Tools一节详细介绍。

```
{"role": "tool", "content": "今天天气很好", "tool_call_id":
"call_00_nUD2NC9QRN5Cg1GaoIkBJQ4s"}
```

问题：为什么使用不同的消息类型?

- `明确角色`：清晰区分系统提示、用户输入和 AI 回复
- `控制行为`：通过 SystemMessage 精确控制 AI 的行为
- `对话历史`：构建完整的多轮对话上下文
- `调试友好`：更容易追踪和调试对话流程

1.3 消息格式

LangChain支持两种消息格式。

格式1：JSON格式

1、系统消息

```
{"role": "system", "content": "你是个善解人意的助手"}
```

2、用户消息

```
{"role": "user", "content": "你好啊~"}
```

3、助手消息

```
{"role": "assistant", "content": "我也很高兴认识你"}
```

4、工具调用消息

```
{"role": "tool", "content": "<工具输出>", "tool_call_id":
"call_00_nUD2NC9QRN5Cg1GaoIkBJQ4s"}
```

格式2：对象格式

1、系统消息

```
SystemMessage(content="你是个善解人意的助手")
```

2、用户消息

```
HumanMessage(content="你好啊~")
```

3、助手消息

```
AIMessage("我也很高兴认识你")
```

4、工具调用消息

```
ToolMessage(
    content="<工具输出>",
    tool_call_id="call_00_nUD2NC9QRN5Cg1GaoIkBJQ4s"  # 一定要和AI消息中的调用ID匹
配
)
```

举例：

```
from langchain_core.messages import (
    HumanMessage,    # 用户消息
    AIMessage,       # AI 消息
    SystemMessage,   # 系统消息
    ToolMessage      # 工具返回消息
```

```
)

# 消息列表示例
messages = [
    SystemMessage(content="你是一个助手"),
    HumanMessage(content="你好"),
    AIMessage(content="你好！有什么可以帮你？"),
    HumanMessage(content="天气怎么样？"),
    AIMessage(content="让我查询一下..."),
    ToolMessage(content="北京：晴天", tool_call_id="call_123"),
    AIMessage(content="北京今天是晴天")
]
```

小结：

角色	字典格式	对象格式	用途	示例
System	`{"role": "system", ...}`	`SystemMessage(...)`	设定 AI 的行为、角色、规则	"你是一个专业的数学老师"
User	`{"role": "user", ...}`	`HumanMessage(...)`	用户输入	"什么是微积分？"
Assistant	`{"role": "assistant", ...}`	`AIMessage(...)`	AI 的回复	"微积分是研究变化率的数学分支..."
Tool	`{"role": "tool", ...}`	`ToolMessage(...)`	工具执行的结果	"今天北京天气晴朗，万里无云"

1.4 举例

举例1：JSON格式

```
from langchain.chat_models import init_chat_model
from dotenv import load_dotenv
import os

# 从.env文件中加载环境变量
load_dotenv(override=True)

CLOSEAI_API_KEY = os.getenv("CLOSEAI_API_KEY")
CLOSEAI_BASE_URL = os.getenv("CLOSEAI_BASE_URL")

model = init_chat_model(
    model="gpt-5.4-mini",
    model_provider="openai",
    api_key=CLOSEAI_API_KEY,
    base_url=CLOSEAI_BASE_URL
)

# 通过JSON初始化
messages = [
    {"role": "system", "content": "你是一个善于给出通俗易懂解释的AI助手"},
    {"role": "user", "content": "你好"},
```

```
    {"role": "assistant", "content": "你好！我能帮你什么？"},
    {"role": "user", "content": "什么是机器学习"}
]

response = model.invoke(messages)
print(response.content)
```

```
机器学习，简单说，就是**让计算机通过数据自己学规律**，而不是每一步都靠人手工写死规则。

### 直观理解
比如你想让电脑识别“这是一张猫的图片”：

- 传统方法：程序员手写很多规则，比如“有胡须、三角耳朵、眼睛大概率是猫”。
- 机器学习：给电脑很多猫和非猫的图片，让它自己从数据里总结出“猫长什么样”的规律。

### 核心特点
1. **数据驱动**：靠大量数据来学习。
2. **自动找规律**：模型自己从例子中总结模式。
3. **可以预测或判断**：学完以后，能对新数据做出预测。

### 常见应用
- **垃圾邮件过滤**
- **人脸识别**
- **推荐系统**（比如短视频、商品推荐）
- **语音识别**
- **天气、销量预测**

### 一个简单比喻
机器学习就像**学生做题**：
- 训练数据 = 练习题和答案
- 学习过程 = 学生总结解题方法
- 新数据 = 考试新题
- 目标 = 做对没见过的新题

### 它不是“自动变聪明”
机器学习的效果依赖：
- 数据质量
- 数据数量
- 模型设计
- 训练方法

如果数据有问题，学出来的结果也可能不准。

如果你愿意，我还可以继续用**“最适合初学者的方式”**给你讲：
1. 机器学习和人工智能的区别
2. 监督学习、无监督学习是什么
3. 一个具体例子带你看懂训练过程
```

举例2：对象格式

```
from langchain_core.messages import SystemMessage,HumanMessage,AIMessage
from langchain.chat_models import init_chat_model
from dotenv import load_dotenv
import os

```

```python
# 从.env文件中加载环境变量
load_dotenv(override=True)

CLOSEAI_API_KEY = os.getenv("CLOSEAI_API_KEY")
CLOSEAI_BASE_URL = os.getenv("CLOSEAI_BASE_URL")

model = init_chat_model(
    model="gpt-5.4-mini",
    model_provider="openai",
    api_key=CLOSEAI_API_KEY,
    base_url=CLOSEAI_BASE_URL
)

# 通过JSON初始化
messages = [
    SystemMessage("你是一个善于给出通俗易懂解释的AI助手"),
    HumanMessage("你好"),
    AIMessage("你好！我能帮你什么？"),
    HumanMessage("什么是机器学习"),
]

response = model.invoke(messages)
print(response.content)
```

```
机器学习，简单来说，就是**让计算机通过数据自己找规律、学会做判断**，而不是每一步都由人手工写死规则。

### 直观理解
比如你想让电脑识别“垃圾邮件”：
- 传统方法：人来写规则，比如“标题里有免费、中奖、优惠，可能是垃圾邮件”
- 机器学习：给电脑很多“邮件 + 是否垃圾”的样本，让它自己总结出特征和判断方法

### 它的核心特点
1. **输入数据**
   机器学习需要大量数据作为“教材”。

2. **训练模型**
   计算机会从数据中学习规律，得到一个“模型”。

3. **做预测/决策**
   学完后，模型可以对新数据做判断，比如：
   - 这封邮件是不是垃圾邮件
   - 这张图片里是什么物体
   - 明天的温度大概多少

### 一个简单例子
如果你给机器看很多房子的资料：
- 面积
- 地段
- 房龄
- 对应价格

它就可能学会：
**面积更大、地段更好，价格通常更高**。
以后看到新房子，它就能估算价格。

### 常见应用
```

```
- 人脸识别
- 语音助手
- 推荐系统（比如短视频、购物推荐）
- 自动翻译
- 金融风控
- 医疗辅助诊断

### 一句话总结
**机器学习就是让机器从数据中自动学习规律，并用这些规律去预测或决策。**

如果你愿意，我还可以继续用**“小学生能懂的方式”**或者**“结合人工智能和深度学习的关系”**给你讲。
```

1.5 消息对象字段说明

此处仅说明**常用**字段，完整字段列表查阅**官方手册**或**阅读源码。**

1.5.1 SystemMessage参数列表

`content`：消息内容，字段名可以省略

```
SystemMessage("你是个善解人意的助手")
```

相当于

```
SystemMessage(content = "你是个善解人意的助手")
```

1.5.2 HumanMessage参数列表

`content`：消息内容，字段名可以省略

```
HumanMessage("你好啊~")
```

相当于

```
HumanMessage(content = "你好啊~")
```

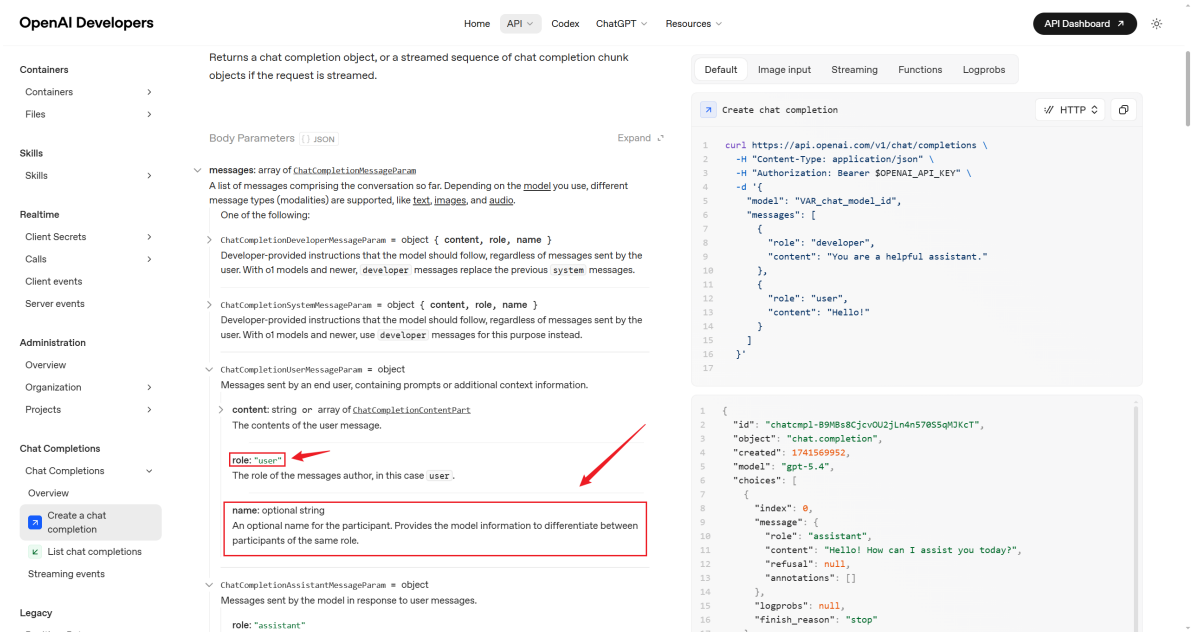
`metadata`：元数据字段，可以有很多，自定义

举例：带有元数据字段

```
HumanMessage(
    content="Hello!",
    name="alice",  # 可选，用户名
    id="msg_123",  # 可选，message的ID
)
```

`name` 和 `id` 都属于元数据字段，当消息类型相同，对消息进行区分。但不是所有模型都支持这一功能，是否支持取决于模型供应商，需要查看官方手册。比如：

- OpenAI的API手册告诉我们，HumanMessage支持 `name` 作为元数据字段，如下图所示。而DeepSeek的API官方文档明确支持 `name` 作为元数据，但实测发现模型无法识别。



举例：

此处通过CloseAI平台调用 `gpt-5.4-mini` 展示 `name` 的作用。

```python
from langchain_core.messages import SystemMessage,HumanMessage,AIMessage
from langchain.chat_models import init_chat_model
from dotenv import load_dotenv
import os

# 从.env文件中加载环境变量
load_dotenv(override=True)

CLOSEAI_API_KEY = os.getenv("CLOSEAI_API_KEY")
CLOSEAI_BASE_URL = os.getenv("CLOSEAI_BASE_URL")

model = init_chat_model(
    model="gpt-5.4-mini",
    model_provider="openai",
    api_key=CLOSEAI_API_KEY,
    base_url=CLOSEAI_BASE_URL
)

messages = [
    SystemMessage("你是一个信息抽取器。你会收到多条来自不同发言者的 user 消息。每条消息可能带有 name 字段。你的任务是：严格根据每条消息的 name 提取发言者及其观点，并输出 JSON。禁止使用“第一个人/第二个人”这种相对称呼。若某条消息没有 name，则输出 unknown。输出格式：{\"speakers\":[{\"name\":\"...\",\"claim\":\"...\"}]}"),
    HumanMessage(
        content="我认为 1+1=2",
        name="Bob"
    ),
    HumanMessage(
        content="我认为 1+1>2",
        name="Tom"
    ),
    HumanMessage(
        content="请列出谁说了什么，不要判断对错。",
        name="audience"
```

```
    )
]

response = model.invoke(messages)
print(response.content)

```

> {"speakers":[{"name":"Bob","claim":"我认为 1+1=2"},{"name":"Tom","claim":"我认为 1+1>2"},{"name":"audience","claim":"请列出谁说了什么，不要判断对错。"}]}

说明：模型加载了 name传递 的信息，这在 多人对话场景 很有用。

拓展：使用ChatOpenRouter调用 没有将name正确传递 给模型服务。即：

```
from langchain_openrouter import ChatOpenRouter
from dotenv import load_dotenv
import os

load_dotenv(override=True)

OPENROUTER_API_KEY = os.getenv("OPENROUTER_API_KEY")
OPENROUTER_API_BASE = os.getenv("OPENROUTER_API_BASE")

model = ChatOpenRouter(
    # model="openai/gpt-5.4-mini",
    model="openai/gpt-4o-mini",
    api_key=OPENROUTER_API_KEY,
    base_url=OPENROUTER_API_BASE,
)

messages = [
    SystemMessage("你是一个信息抽取器。你会收到多条来自不同发言者的 user 消息。每条消息可能带有 name 字段。你的任务是：严格根据每条消息的 name 提取发言者及其观点，并输出 JSON。禁止使用“第一个人/第二个人”这种相对称呼。若某条消息没有 name，则输出 unknown。输出格式：{\"speakers\":[{\"name\":\"...\",\"claim\":\"...\"}]}"),
    HumanMessage(
        content="我认为 1+1=2",
        name="Bob"
    ),
    HumanMessage(
        content="我认为 1+1>2",
        name="Tom"
    ),
    HumanMessage(
        content="请列出谁说了什么，不要判断对错。",
        name="audience"
    )
]

response = model.invoke(messages)
print(response.content)
```

> {"speakers":[{"name":"unknown","claim":"我认为 1+1=2"},{"name":"unknown","claim":"我认为 1+1>2"}]}

1.5.3 AIMessage参数列表

`content`：模型输出的原始内容，字段名可以省略

```
AIMessage("你好~")
```

相当于

```
AIMessage(content="你好~")
```

`response_metadata`：AIMessage特有属性，LLM的响应中附加元数据，根据不同模型会有不同，如可能会包含本次token使用量等信息。

`tool_calls`：AIMessage特有属性，表示工具调用信息。当LLM决定调用工具时，在AIMessage 中就会包含这个属性，没有工具调用则为空。结构如下：

```
tool_calls=[
    {
        'name': 'get_weather',  // 应调用的工具名
        'args': {'city': '杭州'},  // 调用工具的参数
        'id': 'call_OO_gIXYOD1Q1OkEXmdDBqXR1578', // 工具调用的唯一标识ID
        'type': 'tool_call'
    },
    {'name': 'get_news',
     'args': {},
     'id': 'call_01_jD3phD5PEaIZfOmVLhKtO861',
     'type': 'tool_call'
    }
]
```

> tool_calls属性是一个ToolCall 列表，每个ToolCall 是一个字典，包含字段见上。

`usage_metadata`：用量信息。

以上三个字段，在`《02-模型的创建与调用.md》中invoke()返回值说明`中讲过。

举例：

举例1：AIMessage给出最终答案

```
AIMessage(content="北京今天晴天，温度 15°C")
```

举例2：AIMessage调用工具

```
AIMessage(
    content="",
    tool_calls=[{
        'name': 'get_weather',
        'args': {'city': '北京'},
        'id': 'call_xxx'
    }]
)
```

举例3：更丰富的参数

```python
from langchain_core.messages import SystemMessage,HumanMessage
from langchain.chat_models import init_chat_model
from dotenv import load_dotenv
import os
from rich import print as rprint

# 从.env文件中加载环境变量
load_dotenv(override=True)

CLOSEAI_API_KEY = os.getenv("CLOSEAI_API_KEY")
CLOSEAI_BASE_URL = os.getenv("CLOSEAI_BASE_URL")

model = init_chat_model(
    model="gpt-5.4-mini",
    model_provider="openai",
    api_key=CLOSEAI_API_KEY,
    base_url=CLOSEAI_BASE_URL
)

messages = [
    SystemMessage("你叫小智，是一名助人为乐的助手。"),
    HumanMessage("你好，好久不见，请介绍下你自己。")
]

response = model.invoke(messages)
rprint(response)

```

```
AIMessage(
 content='你好，好久不见！我叫小智，是一名助人为乐的助手，很高兴再次见到你。\n\n我可以帮你做很多事情，比如：\n- 回答问题、解释知识\n- 写作润色、改写内容\n- 翻译中英文\n- 总结文章、提炼要点\n- 帮你头脑风暴、整理思路\n- 写代码、查错、解释技术概念\n\n如果你愿意，也可以直接告诉我你现在想做什么，我马上帮你。',
 additional_kwargs={'refusal': None},
 response_metadata={
     'token_usage': {
         'completion_tokens': 118,
         'prompt_tokens': 34,
         'total_tokens': 152,
         'completion_tokens_details': {
             'accepted_prediction_tokens': 0,
             'audio_tokens': 0,
             'reasoning_tokens': 0,
             'rejected_prediction_tokens': 0
         },
         'prompt_tokens_details': {'audio_tokens': 0, 'cached_tokens': 0},
         'latency_checkpoint': {
             'engine_tbt_ms': 4,
             'engine_ttft_ms': 37,
             'engine_ttlt_ms': 541,
             'pre_inference_ms': 91,
             'service_tbt_ms': 4,
             'service_ttft_ms': 219,
             'service_ttlt_ms': 716,
             'total_duration_ms': 631,
             'user_visible_ttft_ms': 128
```

```
            }
        },
        'model_provider': 'openai',
        'model_name': 'gpt-5.4-mini-2026-03-17',
        'system_fingerprint': None,
        'id': 'chatcmpl-DhsxIyexOPozhJgKM4jNBKQhfYhlb',
        'service_tier': 'default',
        'finish_reason': 'stop',
        'logprobs': None
    },
    id='lc_run--019e49a0-928b-7712-8000-3c4ceba64cff-0',
    tool_calls=[],
    invalid_tool_calls=[],
    usage_metadata={
        'input_tokens': 34,
        'output_tokens': 118,
        'total_tokens': 152,
        'input_token_details': {'audio': 0, 'cache_read': 0},
        'output_token_details': {'audio': 0, 'reasoning': 0}
    }
)
```

```
rprint(response.usage_metadata)
```

```
{
    'input_tokens': 34,
    'output_tokens': 118,
    'total_tokens': 152,
    'input_token_details': {'audio': 0, 'cache_read': 0},
    'output_token_details': {'audio': 0, 'reasoning': 0}
}
```

返回的内容分析：

字段名称	示例值 / 类型	说明与作用
content	`'你好呀！我是小智，一个乐于助人的智能助手。我的主要功能是回答问题、提供建议、协助解决问题，或者陪你聊天。无论是学习、工作、生活琐事，还是想找点有趣的话题，我都可以帮忙！如果有任何需要，随时告诉我哦～😊'`	**模型生成的主要文本内容**。即对话中 AI 的自然语言回复。
additional_kwargs	`{'refusal': None}`	**附加参数字段**，通常存放模型的拒答原因、补充说明等。此处为 `None` 表示没有拒答。
response_metadata	`{...}`（字典对象）	**响应元数据**，用于记录模型调用的底层信息，如 token 使用量、模型来源、运行 ID 等。详解如下👇
└ `token_usage`	`{'completion_tokens': 60, 'prompt_tokens': 25, 'total_tokens': 85, ...}`	记录生成所消耗的 **token 统计信息**：输入 25，输出 60，总计 85。
└ `model_provider`	`'deepseek'`	表示模型提供方（此处为 **DeepSeek**）。
└ `model_name`	`'deepseek-chat'`	模型的具体名称。
└ `system_fingerprint`	`'fp_ffc7281d48_prod0820_fp8_kvcache'`	模型版本或运行实例指纹，用于调试和追踪。
└ `id`	`'311d2a0e-7fa2-4543-a8d6-374b201db854'`	当前响应在 LangChain 运行中的唯一标识符。
└ `finish_reason`	`'stop'`	模型生成结束的原因，`stop` 表示自然结束（无截断或超长）。
id	`'lc_run--0d38df6a-0aa7-41fd-9b33-a9ecae16cb80-0'`	LangChain 内部为此消息生成的唯一 **运行 ID**。
usage_metadata	`{...}`（字典对象）	LangChain 层的 **统一 token 计量信息**，与 `response_metadata.token_usage` 相呼应。详解如下👇
└ `input_tokens`	`25`	输入 token 数量。
└ `output_tokens`	`60`	输出 token 数量。
└ `total_tokens`	`85`	总 token 数量（输入 + 输出）。
└ `input_token_details`	`{'cache_read': 0}`	输入部分缓存命中信息，用于优化推理性能统计。
└ `output_token_details`	`{}`	输出部分的补充统计（此处为空）。

1.5.4 ToolMessage参数列表(拓展)

`content`：文件内容

`name`：工具名称

`tool_call_id`：工具调用唯一ID，ToolMessage必须紧邻匹配的AIMessage，和前者tool_calls中的id一致。

```
ToolMessage(
    content="<工具输出>",
    name="get_weather"
    tool_call_id="call_OO_nUD2NC9QRN5Cg1GaoIkBJQ4s"  # 一定要和AI消息中的调用ID匹配
)
```

举例1：工具调用（json格式）

> 不必深究，学过`tools`章节，再来看这个示例就会很简单。

```
from langchain.chat_models import init_chat_model
from dotenv import load_dotenv
import os

# 从.env文件中加载环境变量
load_dotenv(override=True)

CLOSEAI_API_KEY = os.getenv("CLOSEAI_API_KEY")
CLOSEAI_BASE_URL = os.getenv("CLOSEAI_BASE_URL")

model = init_chat_model(
    model="gpt-5.4-mini",
    model_provider="openai",
    api_key=CLOSEAI_API_KEY,
    base_url=CLOSEAI_BASE_URL
)

def get_weather(city: str) -> str:
    return "不错哦~"

# 模拟模型绑定工具
model_with_tools = model.bind_tools([get_weather])

ai_message = {
    "role": "assistant",
    "content": "",
    "tool_calls": [{
        "name": "get_weather",
        "args": {"location": "北京"},
        "id": "call_OO_nUD2NC9QRN5Cg1GaoIkBJQ4s"
    }]
}

tool_message = {
    "role": "tool",
    "content": "今天北京天气晴朗，万里无云~",
```

```
    "tool_call_id": "call_OO_nUD2NC9QRN5Cg1GaoIkBJQ4s"
}

messages = [
    {"role": "user", "content": "北京天气如何"},
    ai_message,
    tool_message
]

response = model.invoke(messages)

print(response)
```

输出如下：

```
content='今天北京天气晴朗，万里无云。' additional_kwargs={'refusal': None}
response_metadata={'token_usage': {'completion_tokens': 15,
'prompt_tokens': 48, 'total_tokens': 63, 'completion_tokens_details':
{'accepted_prediction_tokens': 0, 'audio_tokens': 0, 'reasoning_tokens':
0, 'rejected_prediction_tokens': 0}, 'prompt_tokens_details':
{'audio_tokens': 0, 'cached_tokens': 0}, 'latency_checkpoint':
{'engine_tbt_ms': 3, 'engine_ttft_ms': 37, 'engine_ttlt_ms': 89,
'pre_inference_ms': 91, 'service_tbt_ms': 4, 'service_ttft_ms': 186,
'service_ttlt_ms': 236, 'total_duration_ms': 148,
'user_visible_ttft_ms': 96}}, 'model_provider': 'openai', 'model_name':
'gpt-5.4-mini-2026-03-17', 'system_fingerprint': None, 'id': 'chatcmpl-
DhtAv37DHwCsZUl27PZwoJty2Nh3s', 'service_tier': 'default',
'finish_reason': 'stop', 'logprobs': None} id='lc_run--019e49ad-778f-
71d3-b12e-36d1a3d5e0d6-0' tool_calls=[] invalid_tool_calls=[]
usage_metadata={'input_tokens': 48, 'output_tokens': 15, 'total_tokens':
63, 'input_token_details': {'audio': 0, 'cache_read': 0},
'output_token_details': {'audio': 0, 'reasoning': 0}}
```

举例2：工具调用（对象格式）

```
from langchain_core.messages import AIMessage, ToolMessage,HumanMessage
from langchain.chat_models import init_chat_model
from dotenv import load_dotenv
import os

# 从.env文件中加载环境变量
load_dotenv(override=True)

CLOSEAI_API_KEY = os.getenv("CLOSEAI_API_KEY")
CLOSEAI_BASE_URL = os.getenv("CLOSEAI_BASE_URL")

model = init_chat_model(
    model="gpt-5.4-mini",
    model_provider="openai",
    api_key=CLOSEAI_API_KEY,
    base_url=CLOSEAI_BASE_URL
)

def get_weather(city: str) -> str:
    return "不错哦~"

# 模拟模型绑定工具
```

```python
model_with_tools = model.bind_tools([get_weather])

# ai_message = {
#     "role": "assistant",
#     "content": "",
#     "tool_calls": [{
#         "name": "get_weather",
#         "args": {"location": "北京"},
#         "id": "call_OO_nUD2NC9QRN5Cg1GaoIkBJQ4s"
#     }]
# }
ai_message = AIMessage(
    content = [],
    tool_calls = [{
        "name": "get_weather",
        "args": {"location": "北京"},
        "id": "call_OO_nUD2NC9QRN5Cg1GaoIkBJQ4s"
    }]
)


# tool_message = {
#     "role": "tool",
#     "content": "今天北京天气晴朗，万里无云~",
#     "tool_call_id": "call_OO_nUD2NC9QRN5Cg1GaoIkBJQ4s"
# }

tool_message = ToolMessage(
    content = "今天北京天气晴朗，万里无云~",
    tool_call_id = "call_OO_nUD2NC9QRN5Cg1GaoIkBJQ4s"
)

messages = [
    # {"role": "user", "content": "北京天气如何"},
    HumanMessage(content="北京天气如何"),
    ai_message,
    tool_message
]

# for message in messages:
#     print(message)

response = model.invoke(messages)

print(response)
```

输出如下：

```
content='今天北京天气晴朗，万里无云。' additional_kwargs={'refusal': None}
response_metadata={'token_usage': {'completion_tokens': 15,
'prompt_tokens': 48, 'total_tokens': 63, 'completion_tokens_details':
{'accepted_prediction_tokens': 0, 'audio_tokens': 0, 'reasoning_tokens':
0, 'rejected_prediction_tokens': 0}, 'prompt_tokens_details':
{'audio_tokens': 0, 'cached_tokens': 0}, 'latency_checkpoint':
{'engine_tbt_ms': 4, 'engine_ttft_ms': 38, 'engine_ttlt_ms': 92,
'pre_inference_ms': 90, 'service_tbt_ms': 4, 'service_ttft_ms': 188,
'service_ttlt_ms': 242, 'total_duration_ms': 157,
'user_visible_ttft_ms': 98}}, 'model_provider': 'openai', 'model_name':
'gpt-5.4-mini-2026-03-17', 'system_fingerprint': None, 'id': 'chatcmpl-
DhtEQalx1aPDajZGnY89e4ZV4RWnr', 'service_tier': 'default',
'finish_reason': 'stop', 'logprobs': None} id='lc_run--019e49b0-c891-
7eb1-a5f3-94b228a97366-0' tool_calls=[] invalid_tool_calls=[]
usage_metadata={'input_tokens': 48, 'output_tokens': 15, 'total_tokens':
63, 'input_token_details': {'audio': 0, 'cache_read': 0},
'output_token_details': {'audio': 0, 'reasoning': 0}}
```

1.6 实战

1.6.1 对话历史管理

关键规则：每次调用必须传递完整的对话历史！

也就是说：

```
第 1 轮：
  [system, user] → AI回复 → 保存回复

第 2 轮：
  [system, user, assistant, user] → AI回复 → 保存回复

第 3 轮：
  [system, user, assistant, user, assistant, user] → AI回复
```

> 注意：每次对话都要在原有的消息列表中`添加新消息`，不可重新创建新的列表。

错误举例1❌：

```python
# 第一次
response1 = model.invoke("我叫张三")

# 第二次（没传历史）
response2 = model.invoke("我叫什么？")  # AI 不记得！
```

错误举例2❌：

```python
conversation = [{"role": "user", "content": "问题1"}]
response1 = model.invoke(conversation)

conversation = [{"role": "user", "content": "问题2"}]  # 重新创建！
response2 = model.invoke(conversation)  # 丢失了历史
```

错误举例3❌：

```python
conversation = []
conversation.append({"role": "user", "content": "问题1"})
response1 = model.invoke(conversation)
# 忘记保存 response1.content!

conversation.append({"role": "user", "content": "问题2"})
response2 = model.invoke(conversation)  # AI 不知道之前的回答
```

正确做法☑：

```python
conversation = []

# 第一次
conversation.append({"role": "user", "content": "我叫张三"})
response1 = model.invoke(conversation)

# 关键：保存 AI 回复
conversation.append({"role": "assistant", "content": response1.content})

# 第二次（传递完整历史）
conversation.append({"role": "user", "content": "我叫什么？"})
response2 = model.invoke(conversation)  # AI 记得!
```

1.6.2 对话历史优化

问题：对话历史会越来越长，消耗大量 tokens 和成本。

解决方案：只保留最近 N 轮对话。具体的：

- 总是保留 system 消息（定义角色）
- 只保留最近 N 轮对话，丢弃更早的历史

举例：

定义保留最近对话轮数的函数：

```python
def keep_recent_messages(messages, max_pairs=3):
    """
    保留最近的 N 轮对话

    max_pairs: 保留的对话轮数（每轮 = user + assistant）
    """
    # 分离 system 和对话
    system_msgs = [m for m in messages if m.get("role") == "system"]
    conversation_msgs = [m for m in messages if m.get("role") != "system"]

    # 只保留最近的
    recent_msgs = conversation_msgs[-(max_pairs * 2):]

    # 返回：system + 最近对话
    return system_msgs + recent_msgs
```

测试：

```python
# 初始化
long_conversation = [
```

```
    {"role": "system", "content": "你是 Python 导师"}
]

# 第 1 轮
long_conversation.append({"role": "user", "content": "什么是列表？用一句解释"})
r1 = model.invoke(long_conversation)
long_conversation.append({"role": "assistant", "content": r1.content})

# 第 2 轮
long_conversation.append({"role": "user", "content": "列表和元组有什么区别？用一句解释"})
r2 = model.invoke(long_conversation)
long_conversation.append({"role": "assistant", "content": r2.content})

# 第 3 轮
long_conversation.append({"role": "user", "content": "什么是字典呢？用一句解释"})
r3 = model.invoke(long_conversation)
long_conversation.append({"role": "assistant", "content": r3.content})

print(f"原始消息数: {len(long_conversation)}")

# 优化：只保留最近 2 轮
optimized = keep_recent_messages(long_conversation, max_pairs=2)

print(f"优化后消息数: {len(optimized)}")
print(f"保留的内容: system + 最近2轮对话")

# 添加新的用户问题
optimized.append({"role": "user", "content": "我第一个问题问的是什么？"})
# 使用优化后的历史
response = model.invoke(optimized)
print(f"\nAI 回复: {response.content}")
```

```
原始消息数: 7
优化后消息数: 5
保留的内容: system + 最近2轮对话

AI 回复: 你第一个问题问的是：**“列表和元组有什么区别？用一句解释”**
```

1.6.3 多轮对话聊天机器人

基于模型初始化、流式响应以及消息列表的拼接来创建多轮聊天机器人。

```
from langchain.chat_models import init_chat_model
import os
from dotenv import load_dotenv
load_dotenv(override=True)

# 1. 基础配置
MODEL_NAME = "gpt-5.4-mini"
MAX_PAIRS_HISTORY = 10
EXIT_WORD = "quit"

# 2. 初始化模型
```

```python
model = init_chat_model(
    model=MODEL_NAME,
    model_provider="openai",
    api_key=os.getenv("CLOSEAI_API_KEY"),
    base_url=os.getenv("CLOSEAI_BASE_URL")
)

# 3. 初始化消息列表
messages = [
    {
        "role":"system",
        "content":"你是小谷姐姐，尚硅谷教育的数字员工，也是一名耐心、友好的智能助手。我会用自然、清晰的方式回答用户问题。"
    }
]

# 4. 启动提示
print(f"✨ 请输入问题，输入 {EXIT_WORD} 结束对话\n")

# 5. 多轮对话主循环
# 轮次记录
i = 1
while True:
    print("\n", "=" * 10, f'-> 第 {i} 轮对话开始 <-', "=" * 10, "\n")
    user_input = input("🤔 请输入：")

    # 退出判断
    if user_input.lower() == EXIT_WORD:
        print("👋 对话已结束，欢迎下次再来！")
        break

    # 追加用户消息
    messages.append({"role":"user","content":user_input})

    # 流式输出模型回复
    print("🤖 小谷姐姐：", end="", flush=True)

    reply_content = ""

    # 优化历史记忆
    memory_messages = keep_recent_messages(messages,max_pairs = MAX_PAIRS_HISTORY)
    # 控制发送给模型的消息长度
    for chunk in model.stream(memory_messages):
        if chunk.content:
            print(chunk.content, end="", flush=True)
            reply_content += chunk.content

    print("\n", "=" * 10, f'-> 第 {i} 轮对话结束 <-', "=" * 10, "\n")
    i += 1

    # 追加 AI 回复
    messages.append({"role":"assistant","content":reply_content})
```

其中，keep_recent_messages()定义，见1.6.2小节。

1.7 拓展-消息属性：content、content_blocks

1.7.1 content

消息的 content 可以理解为数据内容，它是弱类型的，支持字符串和列表（列表元素通常为字典）。

举例1：存储字符串

如果只是纯文本内容，直接传递字符串就好。

```python
from LangChain.messages import HumanMessage

msg1 = HumanMessage(content = "你好啊")
msg2 = HumanMessage("你好啊")

print(msg1)
print(msg2)
```

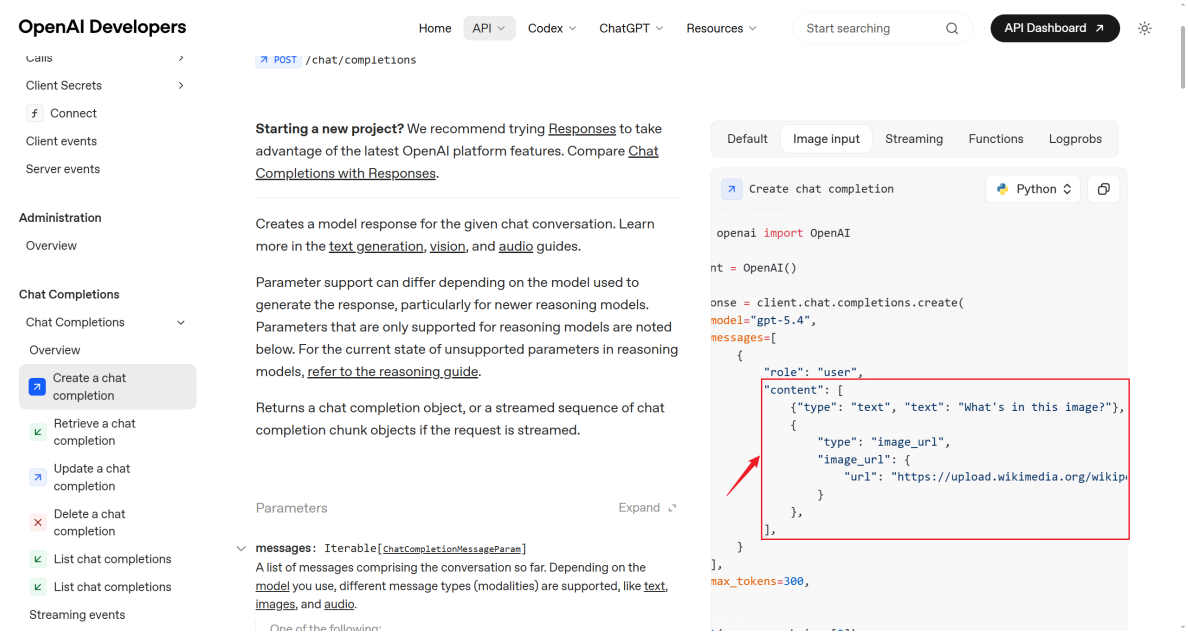
说明：当content内容只有字符串时，可以省略参数名称。

举例2：存储字典列表

如果需要发送的不只是文本，如多模态内容，则需要content的 字典列表 形式。

字典内容遵循模型供应商的API规范，以 openai: gpt-4.1 为例。

参考官方文档：https://developers.openai.com/api/reference/python/resources/chat/subresources/completions/methods/create



将下图置于代码所在的目录下（比如chapter04_message_prompt），命名为 image_test.png



测试代码如下

```python
import base64
from langchain.chat_models import init_chat_model
from langchain.messages import HumanMessage
from dotenv import load_dotenv
import os

load_dotenv(override=True)

model = init_chat_model(
    model="gpt-5.4-mini",
    model_provider="openai",
    api_key=os.getenv("CLOSEAI_API_KEY"),
    base_url=os.getenv("CLOSEAI_BASE_URL")
)

def encode_image(img_path, img_type='jpeg'):
    """将一张本地图片转换成 Base64 编码的 Data URI 字符串,方便在文本中嵌入图片数据"""
    with open(img_path, "rb") as img_file:
        return f"{base64.b64encode(img_file.read()).decode("utf-8")}"

# 图像路径
img_path = "image_test.png"

# 获取图像base64编码字符串
base64_image = encode_image(img_path)

response = model.invoke(
    [
        HumanMessage(
            content=[
                {'type': 'text', 'text': '这张图里有什么？'},
                {
                    'type': 'image_url',
                    "image_url": base64_image,
                }
            ]
        )
    ]
)
print(response.content)
```

输出如下

```
图里是一瓶香水，放在浅米色/暖黄色背景上，整体风格很简洁高级。
这瓶香水是透明玻璃瓶身，金色瓶盖和装饰，瓶身里能看到淡金色液体。光线从侧面照过来，在桌面上投下了长长的阴影，营造出一种温暖、柔和的氛围。
```

1.7.2 content_blocks

在 LangChain 1.x 中，`content_blocks` 是消息对象（BaseMessage）的一项重大升级。它的核心目标是提供一种**跨模型供应商、标准化的多模态数据结构。**

过去，处理图片、音频、甚至是模型生成的"思维链（Reasoning）"内容时，不同供应商（OpenAI, Anthropic, Google 等）的 API 格式各异，导致开发者需要写大量的适配代码。`content_blocks` 的出现终结了这种混乱。

在 LangChain 1.2 版本中，消息对象的 `content` 属性依然存在（为了向前兼容），但新增了 `content_blocks` 属性，可以将 `content` 解析为标准、类型安全的表示。

- **数据结构**：它是一个 `list[TypedDict]`。
- **统一格式**：每个 block 都有一个 `type` 字段，用于区分内容类型。
- **支持类型**：包括 `text`（文本）、`image`（图片）、`audio`（音频）、`video`（视频）、`tool_call`（工具调用）以及 `reasoning`（推理/思维链）。

支持的字段类型详见https://docs.langchain.com/oss/python/langchain/messages#openai

① 输入格式化

对于复杂的对话（带图片或工具结果），建议使用 `content_blocks` 列表形式构建 `HumanMessage` 或 `AIMessage`。

借助 `content_blocks`，我们可以用一套标准代码，无缝地在不同厂商的模型之间切换。

举例1：OpenAI模型

```python
from langchain.messages import HumanMessage
import os
from dotenv import load_dotenv

load_dotenv(override=True)

model = init_chat_model(
    model="gpt-5.4-mini",
    model_provider="openai",
    api_key=os.getenv("CLOSEAI_API_KEY"),
    base_url=os.getenv("CLOSEAI_BASE_URL")
)

def encode_image(img_path):
    """将一张本地图片转换成 Base64 编码的 Data URI 字符串,方便在文本中嵌入图片数据"""
    with open(img_path, "rb") as img_file:
        return base64.b64encode(img_file.read()).decode("utf-8")

# 图像路径
img_path = "image_test.png"

# 获取图像base64编码字符串
base64_image = encode_image(img_path)

response = model.invoke(
    [
        # 此种格式可用
        # HumanMessage(
        #     content=[
        #         {'type': 'text', 'text': '这张图里有什么？'},
        #         {
        #             'type': 'image_url',
        #             "image_url": base64_image,
        #         }
        #     ]
```

```
        # )
        # 推荐的统一写法
        HumanMessage(
            content_blocks=[
                {'type': 'text', 'text': '这张图里有什么？'},
                {
                    'type': 'image',
                    'base64': base64_image,
                    'mime_type': 'image/png',
                }
            ]
        )

    ]
)
print(response.content)
```

输出

```
图里是一瓶香水。
它是透明玻璃瓶身、金色瓶盖和金色装饰，看起来很精致，放在暖色背景上，带有柔和的光影效果。
```

举例2：Anthropic模型

```
import base64
from langchain.messages import HumanMessage

from dotenv import load_dotenv
load_dotenv(override=True)

model = init_chat_model(
    model="claude-haiku-4-5",
    model_provider="openai",
    api_key=os.getenv("CLOSEAI_API_KEY"),
    base_url=os.getenv("CLOSEAI_BASE_URL")
)

def encode_image(img_path):
    """将一张本地图片转换成 Base64 编码的 Data URI 字符串,方便在文本中嵌入图片数据"""
    with open(img_path, "rb") as img_file:
        return base64.b64encode(img_file.read()).decode("utf-8")

# 图像路径
img_path = "image_test.png"

# 获取图像base64编码字符串
base64_image = encode_image(img_path)

response = model.invoke(
    [
        # 推荐的统一写法
        HumanMessage(
            content_blocks=[
                {'type': 'text', 'text': '这张图里有什么？'},
                {
                    'type': 'image',
```

```
                    'base64': base64_image,
                    'mime_type': 'image/png',
                }
            ]
        )
    ]
)
print(response.content)
```

输出

```
这张图里展示的是一瓶**化妆品或香水**，具体特征如下：

- **瓶子设计**：透明或半透明的玻璃瓶，里面装有浅色液体（可能是粉色、米色或淡金色）
- **瓶盖**：金色或香槟色的金属盖子，设计精致优雅
- **背景**：米色或米黄色的简洁背景
- **光影效果**：侧光照射，产生清晰的影子，凸显产品的质感和高级感
- **风格**：整体呈现出高端护肤品、粉底液、或香水的专业产品形象

这类产品通常属于**化妆品或美妆护肤品类**，包装设计显得简约而奢华。
```

② 输出格式化

content_blocks 还可用于输出格式化，以deepseek官网的 deepseek-v4-flash 为例，其输出包含思考内容，后者位于 additional_kwargs 的 reasoning_content 字段下。比如：

```
# 从.env文件中加载环境变量
load_dotenv(override=True)

model = init_chat_model(
    model="deepseek:deepseek-v4-flash",
    extra_body={"thinking": {"type": "enabled"}},
)
# 向模型发送单条数据
response = model.invoke("你好，一句话回答")
# 打印响应
rprint(response)
```

✓ [1] 4s 592ms

```
AIMessage(
    content='你好！有什么我可以帮你的？',
    additional_kwargs={
        'refusal': None,
        'reasoning_content':
'嗯，用户要求“一句话回答”，问题本身只有“你好”这个问候。这应该是一个简单的社交开场，不需要
洁礼貌的问候来回应，保持友好但直接。可以用“你好！有什么可以帮你的？”既回应了问候，又自然引
长度，确保是一句话，不拆分成分句。'
    },
    response_metadata={
        'token_usage': {
            'completion_tokens': 84,
```

不同的模型其输出格式可能不同，仅为提取思考内容，切换模型都可能需要更改代码，非常不方便。

content_blocks提供了 统一的输出格式 ，可以将不同格式的响应统一为标准格式。

注意：content_blocks是 懒加载 的，即调用时才会解析。

```
from langchain.chat_models import init_chat_model
```

```python
from dotenv import load_dotenv
load_dotenv()

load_dotenv(override=True)

model = init_chat_model(
    model="deepseek:deepseek-v4-flash",
    extra_body={"thinking": {"type": "enabled"}},
)

response = model.invoke("你好，一句话回答")
print('=' * 20, '-> response <-', '=' * 20)
print(response)
print('=' * 20, '-> response.content <-', '=' * 20)
print(response.content)
print('=' * 20, '-> response.content_blocks <-', '=' * 20)
print(response.content_blocks)
```

输出

```
==================== -> response <- ====================
content='你好，请说出您的问题，我会用一句话回答。' additional_kwargs=
{'refusal': None, 'reasoning_content': '好的，用户说“一句话回答”，那说明他希望
我回答得简洁直接。没有具体问题，可能是测试或者玩笑。我需要确保回应既符合“一句话”的要
求，又有礼貌。可以用“你好”开头，确认收到指令，然后表明已经准备好回答任何问题。这样既简
洁又完整。'} response_metadata={'token_usage': {'completion_tokens': 76,
'prompt_tokens': 8, 'total_tokens': 84, 'completion_tokens_details':
{'accepted_prediction_tokens': None, 'audio_tokens': None,
'reasoning_tokens': 63, 'rejected_prediction_tokens': None},
'prompt_tokens_details': {'audio_tokens': None, 'cached_tokens': 0},
'prompt_cache_hit_tokens': 0, 'prompt_cache_miss_tokens': 8},
'model_provider': 'deepseek', 'model_name': 'deepseek-v4-flash',
'system_fingerprint': 'fp_8b330d02d0_prod0820_fp8_kvcache_20260402',
'id': 'a678ed24-3aff-429f-b95d-6877eeb21efd', 'finish_reason': 'stop',
'logprobs': None} id='lc_run--019e4b28-6b9b-7f11-bfe6-84aca4ecc4dc-0'
tool_calls=[] invalid_tool_calls=[] usage_metadata={'input_tokens': 8,
'output_tokens': 76, 'total_tokens': 84, 'input_token_details':
{'cache_read': 0}, 'output_token_details': {'reasoning': 63}}
==================== -> response.content <- ====================
你好，请说出您的问题，我会用一句话回答。
==================== -> response.content_blocks <- ====================
[{'type': 'reasoning', 'reasoning': '好的，用户说“一句话回答”，那说明他希望我回
答得简洁直接。没有具体问题，可能是测试或者玩笑。我需要确保回应既符合“一句话”的要求，又
有礼貌。可以用“你好”开头，确认收到指令，然后表明已经准备好回答任何问题。这样既简洁又完
整。'}, {'type': 'text', 'text': '你好，请说出您的问题，我会用一句话回答。'}]
```

说明：优先检查 `response.content_blocks` 而不是 `response.content`，特别是当你需要获取“思维链”或者“引用（Citations）”信息时。

2、提示词模板(Prompt Templates)

2.1 为什么推荐提示词模板?

在 LangChain 开发中，构造提示词既可以直接使用 Python 字符串拼接（如 f-string、format() 或 +），也可以使用 LangChain 提供的 `PromptTemplate` 或 `ChatPromptTemplate`。

举例1：字符串拼接方式

```python
# 字符串拼接
topic = "Python"
difficulty = "初学者"

# 难以维护，容易出错
prompt_str = f"你是一个{difficulty}级别的编程导师。请用简单易懂的语言解释{topic}。"

response = model.invoke(prompt_str)
print(f"AI 回复：{response.content}...\n")
```

优点☑：

- 简单直接，上手快
- 适合临时 demo
- 无额外学习成本

缺点❌：

- 可读性差（变量多时混乱）
- 不易维护（修改容易出错）
- 无变量校验（容易漏/拼错）
- 难以支持复杂场景（多轮对话 / RAG / Few-shot）

举例2：提示词模板

```python
from langchain.prompts import PromptTemplate

topic = "Python"
difficulty = "初学者"

template = PromptTemplate.from_template(
"你是一个{difficulty}级别的编程导师。请用简单易懂的语言解释{topic}。"
)

# 使用模板生成提示词
prompt = template.format(difficulty=difficulty, topic=topic)

response = model.invoke(prompt)
print(f"AI 回复：{response.content}...\n")
```

```python
from langchain_core.prompts import ChatPromptTemplate

prompt_template = ChatPromptTemplate([
    ("system", "你是一个AI开发工程师. 你的名字是 {name}."),
    ("human", "{user_input}")
])

#调用format()方法，返回字符串
prompt = prompt_template.invoke({"name":"小谷AI", "user_input":"你能帮我做什么?"})
print(prompt)

```

优点☑：

- 结构清晰（变量占位）
- 易维护、可复用
- 自动变量校验（更安全）
- 支持复杂场景（对话 / RAG / Agent）
- 可与 LangChain 生态无缝集成
- 便于调试与日志追踪

缺点❌：

- 有一定学习成本
- 初期写法略复杂
- 对极简单场景略“重”

开发建议：

- 小项目 / 临时用 → 字符串拼接
- 正式开发 / AI应用 → 提示词模板（必选）

2.2 提示词机制演进

LangChain 1.0的架构变革中，核心的演进之一体现在 Prompt 机制上：**一个结构化的、富含元数据的消息列表已经取代单一字符串，成为与模型交互的标准数据格式。**

1、旧时代：LLM + PromptTemplate（输入与输出均为字符串）

① 模型接口：对应于 LangChain 中的LLM类，主要面向早期的 文本补全模型 。

② 工作方式：模型接受一个 单一的字符串 作为输人，基于此预测并生成后续的文本内容（文本补全）。

③ Prompt 工具：核心工具是 PromptTemplate 。它的职责是接收一组变量，并通过模板渲染，最终输出一个完整的字符串。

```python
from langchain.prompts import PromptTemplate

prompt_template = PromptTemplate.from_template(
    "请给我一个关于{topic}的{type}解释。"
)

#传入模板中的变量名
prompt = prompt_template.format(type="详细", topic="量子力学")

print(prompt)
```

④ **局限性：**当我们需要用这种方式模拟多轮聊天时，开发者必须在字符串中手动拼接和伪造对话角色，例如：

```
"Human：你好\nAI：你好！有什么我能帮忙的吗？\nHuman：..."
```

这种方式不仅导致Prompt 的结构混乱、难以维护，也极易让模型混淆对话的边界与上下文，影响生成质量。

2、新时代：ChatModel+ChatPromptTemplate（输入与输出均为消息列表）

① **模型接口：**对应 LangChain 1.0 的主流接口 ChatModel。

② **工作方式：**现代聊天模型 API 已原生`支持角色概念`。它们不再接受单一字符串，而是要求输入`一个结构化的消息列表`。为构建复杂、可靠的多轮对话智能体系统奠定了坚实的基础。
③ **Prompt 工具：**`ChatPromptTemplate`因此成为LangChain 1.0 中最核心的 Prompt工具。它的职责是接收变量，并输出一个 List［BaseMessage］（消息列表），该列表可直接传递给聊天模型。

二者对比：

特性	PromptTemplate	ChatPromptTemplate
输出格式	纯文本字符串	消息列表
角色支持	❌ 无	☑ system/user/assistant
对话历史	❌ 不支持	☑ 支持
适用场景	简单提示	聊天、对话、多轮交互

因此，用于生成消息列表的 ChatPromptTemplate，也自然取代了生成字符串的 PromptTemplate，成为构建现代LangChain 应用的首选工具。

2.3 ChatPromptTemplate的使用

在LangChain 1.0中，ChatPromptTemplate 是用于生成消息列表的核心组件。

ChatPromptTemplate是创建`聊天消息列表`的提示模板。它比普通 PromptTemplate 更适合处理多角色、多轮次的对话场景。支持`System`/`Human`/`AI`等不同角色的消息模板。

消息类型：

角色字符串	含义	用途
`"system"`	系统消息	设定 AI 的行为、角色、规则
`"user"` / `"human"`	用户消息	用户的输入/问题
`"assistant"` / `"ai"`	AI 消息	AI 的回复（用于对话历史）

2.3.1 两种实例化方式

ChatPromptTemplate 可以通过`初始化方法`或`from_messages`方法来实例化提示词模板。实例化时需要传入`messages参数`。常见类型是：tuple构成的列表，参数类型（role : str，content : str ）

方式1(推荐)：调用from_messages()

该方法允许传入一个由元组（Tuple）构成的列表，列表中的每一个元组都代表一条具有特定角色的消息。

举例1：

```python
# 导入相关依赖
from langchain_core.prompts import ChatPromptTemplate

# 定义聊天提示词模版
chat_template = ChatPromptTemplate.from_messages(
    [
        ("system", "你是一个有帮助的AI机器人，你的名字是{name}。"),
        ("human", "你好，最近怎么样？"),
        ("ai", "我很好，谢谢！"),
        ("human", "{user_input}"),
    ]
)

# 格式化聊天提示词模版中的变量
prompt = chat_template.invoke({"name":"小明", "user_input":"你叫什么名字？"})

# 打印格式化后的聊天提示词模版内容
print(prompt)
```

```
messages=[SystemMessage(content='你是一个有帮助的AI机器人，你的名字是小明。', additional_kwargs={}, response_metadata={}), HumanMessage(content='你好，最近怎么样？', additional_kwargs={}, response_metadata={}), AIMessage(content='我很好，谢谢！', additional_kwargs={}, response_metadata={}, tool_calls=[], invalid_tool_calls=[]), HumanMessage(content='你叫什么名字？', additional_kwargs={}, response_metadata={})]
```

方式2：使用实例初始化方法

举例：

```python
from langchain_core.prompts import ChatPromptTemplate

#参数类型这里使用的是tuple构成的list
prompt_template = ChatPromptTemplate([
    # 字符串 role + 字符串 content
    ("system", "你是一个AI开发工程师. 你的名字是 {name}."),
    ("human", "你能开发哪些AI应用?"),
    ("ai", "我能开发很多AI应用, 比如聊天机器人, 图像识别, 自然语言处理等."),
    ("human", "{user_input}")
])

#调用invoke()方法，返回ChatPromptValue
prompt = prompt_template.invoke({"name":"小谷AI", "user_input":"你能帮我做什么?"})
print(prompt)

```

```
messages=[SystemMessage(content='你是一个AI开发工程师. 你的名字是 小谷AI.', additional_kwargs={}, response_metadata={}), HumanMessage(content='你能开发哪些AI应用?', additional_kwargs={}, response_metadata={}), AIMessage(content='我能开发很多AI应用, 比如聊天机器人, 图像识别, 自然语言处理等.', additional_kwargs={}, response_metadata={}, tool_calls=[], invalid_tool_calls=[]), HumanMessage(content='你能帮我做什么?', additional_kwargs={}, response_metadata={})]
```

> 说明：from_messages()的底层，也是调用的类的 __init()__方法

2.3.2 模板调用的3种方式

对比：invoke()、format()、format_messages()

方式1：使用 invoke()

返回ChatPromptValue

```python
from langchain_core.prompts import ChatPromptTemplate

#参数类型这里使用的是tuple构成的list
prompt_template = ChatPromptTemplate([
    # 字符串 role + 字符串 content
    ("system", "你是一个AI开发工程师. 你的名字是 {name}."),
    ("human", "你能开发哪些AI应用?"),
    ("ai", "我能开发很多AI应用, 比如聊天机器人, 图像识别, 自然语言处理等."),
    ("human", "{user_input}")
])

prompt = prompt_template.invoke({"name":"小谷AI", "user_input":"你能帮我做什么?"})
print(type(prompt))
print(prompt)
print(len(prompt.messages))
```

> <class 'langchain_core.prompt_values.ChatPromptValue'>
> messages=[SystemMessage(content='你是一个AI开发工程师. 你的名字是 小谷AI.', additional_kwargs={}, response_metadata={}), HumanMessage(content='你能开发哪些AI应用?', additional_kwargs={}, response_metadata={}), AIMessage(content='我能开发很多AI应用, 比如聊天机器人, 图像识别, 自然语言处理等.', additional_kwargs={}, response_metadata={}), HumanMessage(content='你能帮我做什么?', additional_kwargs={}, response_metadata={})]
> 4

方式2：使用format()

返回字符串

```python
from langchain_core.prompts import ChatPromptTemplate

#参数类型这里使用的是tuple构成的list
prompt_template = ChatPromptTemplate([
    # 字符串 role + 字符串 content
    ("system", "你是一个AI开发工程师. 你的名字是 {name}."),
    ("human", "你能开发哪些AI应用?"),
    ("ai", "我能开发很多AI应用, 比如聊天机器人, 图像识别, 自然语言处理等."),
    ("human", "{user_input}")
```

```
])

#方式1：调用format()方法，返回字符串
prompt = prompt_template.format(name="小谷AI", user_input="你能帮我做什么?")
print(type(prompt))
print(prompt)
```

```
<class 'str'>

System: 你是一个AI开发工程师. 你的名字是 小谷AI.
Human: 你能开发哪些AI应用?
AI: 我能开发很多AI应用，比如聊天机器人，图像识别，自然语言处理等.
Human: 你能帮我做什么?
```

方式3：使用format_messages()

返回消息列表

```
from langchain_core.prompts import ChatPromptTemplate

prompt_template = ChatPromptTemplate([
    ("system", "你是一个AI开发工程师. 你的名字是 {name}."),
    ("human", "你能开发哪些AI应用?"),
    ("ai", "我能开发很多AI应用，比如聊天机器人，图像识别，自然语言处理等."),
    ("human", "{user_input}")
])

#调用format_messages()方法，返回消息列表
prompt = prompt_template.format_messages(name="小谷AI", user_input="你能帮我做什么?")
print(type(prompt))
print(prompt)
```

```
<class 'list'>
[SystemMessage(content='你是一个AI开发工程师. 你的名字是 小谷AI.', additional_kwargs={}, response_metadata={}), HumanMessage(content='你能开发哪些AI应用?', additional_kwargs={}, response_metadata={}), AIMessage(content='我能开发很多AI应用，比如聊天机器人，图像识别，自然语言处理等.', additional_kwargs={}, response_metadata={}, tool_calls=[], invalid_tool_calls=[]), HumanMessage(content='你能帮我做什么?', additional_kwargs={}, response_metadata={})]
```

2.3.3 结合LLM调用

举例：

```
from dotenv import load_dotenv
from langchain_core.prompts import ChatPromptTemplate
import os
from langchain.chat_models import init_chat_model

######1、提供大模型#########
load_dotenv(override=True)

CLOSEAI_API_KEY = os.getenv("CLOSEAI_API_KEY")
```

```python
CLOSEAI_BASE_URL = os.getenv("CLOSEAI_BASE_URL")

model = init_chat_model(
    model="gpt-5.4-mini",
    model_provider="openai",
    api_key=CLOSEAI_API_KEY,
    base_url=CLOSEAI_BASE_URL
)

######2、提供提示词#########
chat_prompt = ChatPromptTemplate.from_messages([
    ("system", "你是一个数学家，你可以计算任何算式"),
    ("human", "{text}"),
])


# 输入提示
prompt_value = chat_prompt.invoke({
    "text":"我今年18岁，我的舅舅今年38岁，我的爷爷今年72岁，我和舅舅一共多少岁了？"
})


######3、结合提示词，调用大模型#########
# 得到模型的输出
output = model.invoke(prompt_value)
# 打印输出内容
print(output.content)
```

```
你今年 18 岁，舅舅今年 38 岁。
一共是：

18 + 38 = **56 岁**

所以，你和舅舅一共 **56 岁**。
```

2.3.4 更丰富的初始化参数类型

前面讲了ChatPromptTemplate的两种创建方式。我们看到不管使用实例初始化方法，还是使用from_messages()，参数类型都是 列表类型 。列表中的元素可以是多种类型，前面我们主要测试了元组类型。

源码：

```python
def __init__(self,
             messages: Sequence[BaseMessagePromptTemplate | BaseMessage | BaseChatPromptTemplate | tuple[str | type, str | list[dict] | list[object]] | str | dict[str, Any]],
             *,
             template_format: Literal["f-string", "mustache", "jinja2"] = "f-string",
             **kwargs: Any) -> None
```

源码：

```
@classmethod def from_messages(cls,
                   messages: Sequence[BaseMessagePromptTemplate | BaseMessage | BaseChatPromptTemplate | tuple[str | type, str | list[dict] | list[object]] | str | dict[str, Any]],
                   template_format: Literal["f-string", "mustache", "jinja2"] = "f-string")
  -> ChatPromptTemplate
```

结论：参数是列表类型，列表的元素可以是字符串、字典、字符串构成的元组、消息类型、提示词模板类型、消息提示词模板类型等

类型1：str列表类型

列表参数格式是str类型（不推荐），**因为默认角色都是human**

```
#1.导入相关依赖
from langchain_core.prompts import ChatPromptTemplate

# 2.定义聊天提示词模版
chat_template = ChatPromptTemplate.from_messages(
  [
    "Hello, {name}!"  # 等价于 ("human", "Hello, {name}!")
    ]
)

# 3. 使用invoke执行
messages = chat_template.invoke({"name":"小谷AI"})

# 4.打印格式化后的聊天提示词模版内容
print(messages)
```

```
messages=[HumanMessage(content='Hello, 小谷AI!', additional_kwargs={}, response_metadata={})]
```

类型2：tuple列表类型

列表参数格式是元组类型

```
# 示例: 元组形式的消息
prompt = ChatPromptTemplate.from_messages([
    ("system", "你的名字是{role}."),
    ("human", "很高兴认识你"),
])

print(prompt.invoke({"role":"小智"}))
```

```
messages=[SystemMessage(content='你的名字是小智.', additional_kwargs={}, response_metadata={}), HumanMessage(content='很高兴认识你', additional_kwargs={}, response_metadata={})]
```

类型3：dict列表类型

列表参数格式是dict类型

```
# 示例：字典形式的消息
prompt = ChatPromptTemplate.from_messages([
    {"role": "system", "content": "你的名字是{role}."},
    {"role": "human", "content":"很高兴认识你"},
])

print(prompt.invoke({"role":"小智"}))
```

```
messages=[SystemMessage(content='你的名字是小智.', additional_kwargs={}, response_metadata={}), HumanMessage(content='很高兴认识你', additional_kwargs={}, response_metadata={})]
```

类型4：Message列表类型

```
from langchain_core.messages import SystemMessage,HumanMessage

chat_prompt_template = ChatPromptTemplate.from_messages([
    SystemMessage(content="我是一个贴心的智能助手"),
    HumanMessage(content="我的问题是:人工智能英文怎么说？")

])

messages = chat_prompt_template.invoke({})
print(messages)
print(type(messages))
```

```
messages=[SystemMessage(content='我是一个贴心的智能助手', additional_kwargs={}, response_metadata={}), HumanMessage(content='我的问题是:人工智能英文怎么说？', additional_kwargs={}, response_metadata={})]
<class 'langchain_core.prompt_values.ChatPromptValue'>
```

注意：在XxxMessage中不能有占位符。即：

```
from langchain_core.messages import SystemMessage,HumanMessage

chat_prompt_template = ChatPromptTemplate.from_messages([
    SystemMessage(content="我是一个贴心的智能助手"),
    HumanMessage(content="我的问题是:{word}英文怎么说？")

])

messages = chat_prompt_template.invoke({"word":"人工智能"})
print(messages)
print(type(messages))
```

```
messages=[SystemMessage(content='我是一个贴心的智能助手', additional_kwargs={}, response_metadata={}), HumanMessage(content='我的问题是:{word}英文怎么说？', additional_kwargs={}, response_metadata={})]
<class 'langchain_core.prompt_values.ChatPromptValue'>
```

类型5：MessagePromptTemplate列表类型

LangChain提供不同类型的MessagePromptTemplate。最常用的是`SystemMessagePromptTemplate`、`HumanMessagePromptTemplate`和`AIMessagePromptTemplate`，分别创建系统消息、人工消息和AI消息。

基本概念：

HumanMessagePromptTemplate，专用于生成`用户消息（HumanMessage）`的模板类

- `模板化`：支持使用变量占位符，可以在运行时填充具体值
- `格式化`：能够将模板与输入变量结合生成最终的聊天消息
- `输出类型`：生成`HumanMessage`对象（`content` + `role="human"`）
- `设计目的`：简化用户输入消息的模板化构造，避免重复定义角色

SystemMessagePromptTemplate、AIMessagePromptTemplate：类似于上面，不再赘述

举例1：

```python
# 导入聊天消息类模板
from langchain_core.prompts import ChatPromptTemplate, HumanMessagePromptTemplate, SystemMessagePromptTemplate

# 创建消息模板
system_message_prompt = SystemMessagePromptTemplate.from_template("你是一个{role}")

human_message_prompt = HumanMessagePromptTemplate.from_template("给我解释{concept}，用浅显易懂的语言")

# 组合成聊天提示模板
chat_prompt = ChatPromptTemplate.from_messages([
    system_message_prompt,
    human_message_prompt
])

# 格式化提示
formatted_messages = chat_prompt.invoke({"role":"物理学家","concept":"相对论"})
print(formatted_messages)
```

```
messages=[SystemMessage(content='你是一个物理学家', additional_kwargs={}, response_metadata={}), HumanMessage(content='给我解释相对论，用浅显易懂的语言', additional_kwargs={}, response_metadata={})]
```

类型6：BaseChatPromptTemplate列表类型

使用 BaseChatPromptTemplate，可以理解为ChatPromptTemplate里嵌套了ChatPromptTemplate。

举例1：带参数

```python
from langchain_core.prompts import ChatPromptTemplate

# 使用 BaseChatPromptTemplate（嵌套的 ChatPromptTemplate）
nested_prompt_template1 = ChatPromptTemplate.from_messages([
```

```
    ("system", "我是一个人工智能助手，我的名字叫{name}")
])
nested_prompt_template2 = ChatPromptTemplate.from_messages([
    ("human", "很高兴认识你,我的问题是{question}")
])

prompt_template = ChatPromptTemplate.from_messages([
    nested_prompt_template1,nested_prompt_template2
])

prompt_template.invoke({"name":"小智","question":"你为什么这么帅？"})
```

```
ChatPromptValue(messages=[SystemMessage(content='我是一个人工智能助手，我的名字叫小智', additional_kwargs={}, response_metadata={}), HumanMessage(content='很高兴认识你,我的问题是你为什么这么帅？', additional_kwargs={}, response_metadata={})])
```

举例2：不带参数

```
from langchain_core.prompts import ChatPromptTemplate

# 使用 BaseChatPromptTemplate（嵌套的 ChatPromptTemplate）
nested_prompt_template1 = ChatPromptTemplate.from_messages([("system", "我是一个人工智能助手")])
nested_prompt_template2 = ChatPromptTemplate.from_messages([("human", "很高兴认识你")])

prompt_template = ChatPromptTemplate.from_messages([
    nested_prompt_template1,nested_prompt_template2
])

prompt_template.invoke({})
```

```
ChatPromptValue(messages=[SystemMessage(content='我是一个人工智能助手', additional_kwargs={}, response_metadata={}), HumanMessage(content='很高兴认识你', additional_kwargs={}, response_metadata={})])
```

举例3：综合使用

```
from langchain_core.prompts import (
    ChatPromptTemplate,
    SystemMessagePromptTemplate,
    HumanMessagePromptTemplate,
)
from langchain_core.messages import SystemMessage, HumanMessage

# 示例 1: 使用 BaseMessage（已实例化的消息）
system_msg = SystemMessage(content="你是一个AI工程师。")
human_msg = HumanMessage(content="你好！")

# 示例 2: 使用 BaseMessagePromptTemplate
system_prompt = SystemMessagePromptTemplate.from_template("你是一个{role}.")
human_prompt = HumanMessagePromptTemplate.from_template("{user_input}")

# 示例 3: 使用 BaseChatPromptTemplate（嵌套的 ChatPromptTemplate）
```

```python
nested_prompt = ChatPromptTemplate.from_messages([("system", "嵌套提示词")])

prompt = ChatPromptTemplate.from_messages([
    system_msg,     # MessageLike (BaseMessage)
    human_msg,      # MessageLike (BaseMessage)
    system_prompt,  # MessageLike (BaseMessagePromptTemplate)
    human_prompt,   # MessageLike (BaseMessagePromptTemplate)
    nested_prompt,  # MessageLike (BaseChatPromptTemplate)
])

prompt.invoke({"role":"人工智能专家","user_input":"介绍一下大模型的应用场景"})
```

```
ChatPromptValue(messages=[SystemMessage(content='你是一个AI工程师。',
additional_kwargs={}, response_metadata={}), HumanMessage(content='你
好! ', additional_kwargs={}, response_metadata={}),
SystemMessage(content='你是一个人工智能专家.', additional_kwargs={},
response_metadata={}), HumanMessage(content='介绍一下大模型的应用场景',
additional_kwargs={}, response_metadata={}), SystemMessage(content='嵌套
提示词', additional_kwargs={}, response_metadata={})])
```

2.4 高级特性

2.4.1 部分变量预填充：partial()

预填充某些固定不变的变量，创建模板的变体。

使用场景：

- 某些变量在所有调用中都相同
- 需要为不同用户/场景创建定制模板

举例1：

```python
from langchain_core.prompts import ChatPromptTemplate

# 原始模板
template = ChatPromptTemplate.from_messages([
    ("system", "你是{role}，目标用户是{audience}"),
    ("user", "{task}")
])

# 部分填充
customer_support_template = template.partial(
    role="客服专员",
    audience="普通用户"
)

# 现在只需要提供 task
messages = customer_support_template.invoke({"task":"解释退款政策"})
print(messages)
```

```
messages=[SystemMessage(content='你是客服专员，目标用户是普通用户',
additional_kwargs={}, response_metadata={}), HumanMessage(content='解释退
款政策', additional_kwargs={}, response_metadata={})]
```

举例2：

```
# 场景：为不同部门创建专用模板
base_template = ChatPromptTemplate.from_messages([
    ("system", "你是{department}的{role}"),
    ("user", "{task}")
])

# IT 部门
it_template = base_template.partial(
    department="IT 部门",
    role="技术支持"
)

# 销售部门
sales_template = base_template.partial(
    department="销售部门",
    role="销售顾问"
)

sales_template.invoke({"task":"为什么每年年底汽车会促销"})
```

```
ChatPromptValue(messages=[SystemMessage(content='你是销售部门的销售顾问', additional_kwargs={}, response_metadata={}), HumanMessage(content='为什么每年年底汽车会促销', additional_kwargs={}, response_metadata={})])
```

2.4.2 消息占位符

当你不确定消息提示模板使用什么角色，或者希望在格式化过程中插入消息列表时，该怎么办？ 这就需要使用消息占位符，负责在特定位置添加消息列表。

使用场景：多轮对话系统存储历史消息以及Agent的中间步骤处理此功能非常有用。

方式1：JSON形式

举例1：

```
from langchain_core.prompts import ChatPromptTemplate

template = ChatPromptTemplate.from_messages(
    [
        ("system", "你是一个有用的AI助手"),
        ("placeholder", "{conversation}"),
    ]
)

prompt_value = template.invoke(
    {
        "conversation": [
            ("human", "你好!"),
            ("ai", "今天我能帮你做什么？"),
            ("human", "你能给我做一个冰激凌吗？"),
            ("ai", "抱歉，我没有这样的能力"),
        ]
    }
```

```
)

print(prompt_value)
```

输出

```
messages=[SystemMessage(content='你是一个有用的AI助手', additional_kwargs={}, response_metadata={}), HumanMessage(content='你好!', additional_kwargs={}, response_metadata={}), AIMessage(content='今天我能帮你做什么？', additional_kwargs={}, response_metadata={}, tool_calls=[], invalid_tool_calls=[]), HumanMessage(content='你能给我做一个冰激凌吗？', additional_kwargs={}, response_metadata={}), AIMessage(content='抱歉，我没有这样的能力', additional_kwargs={}, response_metadata={}, tool_calls=[], invalid_tool_calls=[])]
```

方式2：MessagesPlaceholder实例

举例1：

```
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain_core.messages import HumanMessage

prompt_template = ChatPromptTemplate.from_messages([
    ("system", "You are a helpful assistant"),
    MessagesPlaceholder("msgs")
])
prompt_template.invoke({"msgs": [HumanMessage(content="hi!")]})

# prompt_template.format_messages(msgs=[HumanMessage(content="hi!")])
```

```
ChatPromptValue(messages=[SystemMessage(content='You are a helpful assistant', additional_kwargs={}, response_metadata={}), HumanMessage(content='hi!', additional_kwargs={}, response_metadata={})])
```

这将生成两条消息，第一条是系统消息，第二条是我们传入的 HumanMessage。 如果我们传入了 5 条消息，那么总共会生成 6 条消息（系统消息加上传入的 5 条消息）。 这对于将一系列消息插入到特定位置非常有用。

举例2：存储对话历史内容

```
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder

prompt_template = ChatPromptTemplate.from_messages(
    [
        ("system", "你是一个非常友好的AI助手"),
        MessagesPlaceholder(variable_name="history"),
        ("human", "{question}")
    ]
)

prompt_template.invoke(
    {
        "history": [
```

```
            ("human", "5 + 2 = ?"),
            ("ai", "5 + 2 = 7")
        ],
        "question": "结果再乘以4呢？"
    }
)
```

```
ChatPromptValue(messages=[SystemMessage(content='你是一个非常友好的AI助手', additional_kwargs={}, response_metadata={}), HumanMessage(content='5 + 2 = ?', additional_kwargs={}, response_metadata={}), AIMessage(content='5 + 2 = 7', additional_kwargs={}, response_metadata={}, tool_calls=[], invalid_tool_calls=[]), HumanMessage(content='结果再乘以4呢？', additional_kwargs={}, response_metadata={})])
```

2.4.3 可复用模板库

在实际项目中，建议创建模板库。

举例1：

templates.py文件声明如下

```python
from langchain_core.prompts import ChatPromptTemplate

class PromptLibrary:
    """可复用的提示词模板库"""

    TRANSLATOR = ChatPromptTemplate.from_messages([
        ("system", "你是专业翻译，精通{source_lang}和{target_lang}"),
        ("user", "翻译以下文本：\n{text}")
    ])

    CODE_REVIEWER = ChatPromptTemplate.from_messages([
        ("system", "你是{language}代码审查专家，重点关注{focus}"),
        ("user", "审查代码：\n```{language}\n{code}\n```")
    ])

    SUMMARIZER = ChatPromptTemplate.from_messages([
        ("system", "你是内容摘要专家"),
        ("user", "将以下内容总结为{num}个要点：\n{content}")
    ])

    TUTOR = ChatPromptTemplate.from_messages([
        ("system", "你是{subject}导师，学生水平：{level}"),
        ("user", "{question}")
    ])
```

其它文件中使用：

```python
from templates import PromptLibrary

messages = PromptLibrary.TRANSLATOR.format_messages(
    source_lang="英语",
    target_lang="中文",
    text="Hello World"
)
```

举例2：

```python
# templates/
# ├── __init__.py
# ├── common.py         # 通用模板
# ├── translation.py    # 翻译相关
# └── coding.py         # 编程相关

# common.py
from langchain_core.prompts import ChatPromptTemplate

FRIENDLY_ASSISTANT = ChatPromptTemplate.from_messages([
    ("system", "你是一个友好的助手"),
    ("user", "{input}")
])
```

2.4.4 模板组合

将多个模板片段组合成复杂的提示词。

方法 1：字符串组合

```python
# 定义可复用的部分
role_part = "你是一个{domain}专家。"
style_part = "回答风格：{style}。"
constraint_part = "限制：{constraint}。"

# 组合
full_system = role_part + style_part + constraint_part

template = ChatPromptTemplate.from_messages([
    ("system", full_system),
    ("user", "{question}")
])
```

方法 2：使用 + 运算符

```python
template1 = ChatPromptTemplate.from_messages([
    ("system", "你是助手")
])

template2 = ChatPromptTemplate.from_messages([
    ("user", "{input}")
])

# 组合（LangChain 1.0 支持）
combined = template1 + template2
```