ChatGPT

Jinja2 模板教程 —— 面向 LLM 提示词构建

在大型语言模型（LLM）提示词模板开发中，使用 Jinja2 模板引擎可以显著提高提示构建的灵活性和可维护性[1]。本教程将系统讲解 Jinja2 的核心用法，并结合提示工程场景给出实践建议。

1. Jinja2 基础语法

模板语法概览：Jinja2 使用特定的定界符来嵌入代码逻辑[2]。默认情况下：`{{ ... }}` 包含**表达式或变量**，会在渲染时输出其值；`{% ... %}` 包含**语句**（如控制流语句），用于编写模板逻辑；`{# ... #}` 则用于**注释**，不会在输出中出现[2]。例如：

```
Hello, {{ user_name }}!
{% if user_vip %}
  Welcome back, VIP user!
{% else %}
  Welcome!
{% endif %}
```

上例中，`{{ user_name }}` 将被上下文中的实际用户名替换输出，`{% if %}...{% endif %}` 块用于根据条件输出不同内容。双大括号 `{{ }}` 只是“打印”语句的定界符，并非变量名的一部分[3]。

变量插值：模板中的变量由渲染时传入的上下文字典定义[4]。可以使用点号（`.`）访问变量的属性或字典键，也可以使用下标语法（`[]`）访问[5]。下面两行是等价的：

```
{{ user.name }}
{{ user['name'] }}
```

如果访问的变量或属性在上下文中不存在，Jinja2 会返回一个“未定义”对象。默认情况下，未定义值如果被打印或迭代，会被当作空字符串处理，但在参与算术等其他操作时会抛出错误[6]。为了避免空输出掩盖错误，开发者可以考虑在 Environment 初始化时使用 **StrictUndefined**，使未定义变量在任何情况下都抛出错误，便于发现问题（StrictUndefined 会禁止对未定义值的一切操作，只允许检查其是否未定义[7]）。

过滤器：过滤器可以对变量值进行格式转换或处理，其语法是使用管道符号 `|` 将过滤器附加在变量或表达式之后[8]。多个过滤器可以链式书写，前一个过滤器的输出将作为下一个过滤器的输入[8]。例如：

```
{{ text|trim|upper }}
```

此例中，`trim` 过滤器会去掉 `text` 字符串首尾空白，然后将结果传给 `upper` 过滤器转换为大写字母。带参数的过滤器可以在其后附加括号传入参数，例如：`{{ list_values|join(', ') }}` 会用逗号和空格将列表连接成字符串[9]。Jinja2 内置了丰富的过滤器（如 `default`，`length`，`jsonify` 等），可参考官方文档的过滤器列表[10]。在提示词模板中，过滤器常用于格式化注入的变量（比如将数字格式化为百分比、对用户输入做 HTML转义等）。

条件语句（if/elif/else）： Jinja2 的 `{% if %}` 语句与 Python 的 if 逻辑类似，可根据条件决定是否渲染某段内容 [11]。基本形式为 `{% if 条件 %} ... {% endif %}`。可以使用 `{% elif %}` 和 `{% else %}` 增加分支：[12]

```
{% if score >= 80 %}
  Well done, you passed!
{% elif score >= 60 %}
  Just passed, keep improving!
{% else %}
  Unfortunately, you failed.
{% endif %}
```

条件判断中还可以利用 Jinja2 的内置**测试**（Tests）语法，例如检查变量是否已定义：`{% if variable is defined %}`，或使用比较运算和逻辑运算。条件判断结果会影响最终提示词内容，例如仅当有额外指导语时才插入相关提示。

循环语句（for）： `{% for %}` 用于遍历序列或字典等可迭代对象 [13]。其基本形式为 `{% for 元素 in 序列 %} ... {% endfor %}`。例如，遍历用户列表输出用户名：

```
<ul>
{% for user in users %}
  <li>{{ user.username }}</li>
{% endfor %}
</ul>
```

如果需要同时获取字典的键和值，可以在 for 声明中使用解构，例如 `{% for key, value in my_dict.items() %}` [14]。Jinja2 在 for 循环内部提供了一系列有用的**循环变量**，通过特殊的 `loop` 对象访问，例如：`loop.index`（当前迭代次数，从1开始）、`loop.first`（是否第一次迭代）、`loop.last`（是否最后一次迭代）等 [15] [16]。这些在构建复杂提示时很有帮助，比如仅在第一次迭代时输出不同格式，或在最后一项后不加逗号等。

Jinja2 的 for 循环还支持一些高级用法：可以在循环声明中直接加条件过滤元素（如 `{% for item in items if item.show %}` 会跳过不满足条件的项 [17]），也可以使用 `{% else %}` 子句在循环没有执行任何迭代时输出默认内容 [18]。**注意：** Jinja2 的循环中不支持直接使用 `break` 或 `continue` 控制流，但通过 if 条件过滤基本可以实现类似效果 [17]。在 LLM 提示构建中，for 循环常用于遍历多轮对话列表、列出选项等场景。

此外，Jinja2 提供了 `{% raw %}...{% endraw %}` 块用于输出模板语法本身（不被解析），`{% set %}` 用于在模板内部给变量赋值等。本节列举的是最常用的基础语法，已能满足大部分提示模板需求。

2. 模板继承与模块化结构

概念： 模板继承是 Jinja2 最强大的功能之一，它允许我们定义一个包含通用结构的基础模板，然后由子模板继承并填充差异部分，从而实现**模块化和代码复用** [19]。对于LLM提示构建，这意味着可以将公共的提示框架（如统一的系统提示格式）抽象为基模板，不同场景的具体提示继承它并填充特定内容。

基础模板（base template）：基础模板包含网站或应用所有提示通用的骨架，并定义若干可以被覆盖的“块（block）”区域 [19] [20]。块使用 `{% block 名称 %}...{% endblock %}` 声明。例如，一个基础模板可以定义提示的总体结构：

```
{% block system_prompt %}
[System] You are a helpful assistant.
{% endblock %}

{% block user_message %}{% endblock %}
```

上例定义了两个块：`system_prompt` 和 `user_message`。基础模板本身可以提供默认内容（如系统提示），或留空等待子模板覆盖。**块标签的作用**是声明一个可由子模板提供具体内容的占位区域 [21]。渲染时，如果子模板没有覆盖某个块，就使用基模板中该块的内容 [22]。

子模板继承：子模板使用 `{% extends 'base模板名' %}` 指令来声明继承某个基础模板 [23]。extends 必须是模板中的第一条指令，以确保基模板能正确应用 [24]。在子模板中，通过定义与基础模板同名的 `{% block 名称 %} ... {% endblock %}` 来覆盖相应区域 [25]。例如：

```
{% extends "base_prompt.jinja" %}

{% block user_message %}
[User] {{ user_input }}
{% endblock %}
```

该子模板继承了 `base_prompt.jinja`，并覆盖了 `user_message` 块，在其中定义具体的用户输入内容格式。渲染此子模板时，Jinja2 会首先载入基模板，将子模板提供的块内容插入到基模板对应位置，然后输出完整的结果 [23]。如果子模板未覆盖某块，则保留基模板中的默认内容 [22]。

块的作用域与注意事项：一个模板中不允许定义两个同名的块，否则引擎将无法确定父模板应该使用哪个内容 [26]。块标签支持嵌套使用（如在 `if` 或 `for` 内部），但需要注意即使外围条件不成立，被定义的块也依然会存在于模板中，只是不会显示非块部分 [27]。Jinja2 提供了特殊变量 `self`，允许在模板内调用自身的块，以便在同一模板中重复输出块内容 [28]。例如 `{{ self.user_message() }}` 可在模板内再次渲染自己的某块。

子模板中可以使用 `{{ super() }}` 调用父模板对同一块的原始内容 [29]。这在希望在覆盖块时**保留父模板部分内容**很有用。例如：

```
{% block system_prompt %}
{{ super() }}
(Addition: Follow user instructions carefully.)
{% endblock %}
```

上例在子模板的 `system_prompt` 块中调用了父模板原有内容，然后追加了一句，达到增量修改的效果 [29]。

通过模板继承，提示模板的公共结构和个性化内容可以解耦：公共部分只需维护在基模板里，各子模板专注填充各自差异。这对于大型提示或多场景提示工程特别有帮助。比如，可以有一个通用的对话提示框架模板，子模板分别覆盖填充不同任务的系统提示或示例对话。

3. 宏（Macros）与 `include` 用法

宏（macro）：宏相当于模板中的函数，可将经常使用的模板片段封装起来，以便重用[30]。使用`{% macro 名称(参数列表) %} ... {% endmacro %}`定义宏，在模板中可通过`{{ 名称(参数) }}`进行调用[31][32]。例如，我们定义一个格式化对话消息的宏：

```
{% macro format_message(role, content) -%}
**{{ role }}:** {{ content }}
{%- endmacro %}

{{ format_message('User', user_question) }}
{{ format_message('Assistant', assistant_answer) }}
```

调用宏就像调用函数一样，并将返回的片段插入模板输出中[32]。宏可以接受默认参数、可变参数（使用`*varargs`和`**kwargs`）等，功能与Python函数类似[33]。宏内部也可以使用控制流、变量等模板语法。需要注意的是，如果宏定义在另一个模板文件中，需要先导入后才能使用[34]。

在不同模板中重用宏：Jinja2 提供两种导入方式：

- **导入整个模板作为模块：**使用`{% import '模板名' as 别名 %}`将指定模板中的所有宏和变量作为模块引入，然后通过`别名.宏名()`来调用[35]。例如，有一个`components.jinja`定义了一些宏：

```
<!-- components.jinja -->
{% macro button(text, link) -%} <button onclick="open('{{link}}')">{{ text }}</button> {%-
endmacro %}
```

在当前模板中可导入并使用：

```
{% import 'components.jinja' as comp %}
{{ comp.button('Help', '/help') }}
```

- **选择导入特定宏：**使用`{% from '模板名' import macro1, macro2 as 别名 %}`直接导入指定的宏到当前命名空间[36]。例如：`{% from 'components.jinja' import button %}`然后直接使用`{{ button('OK', '/ok') }}`调用即可。

导入的宏默认**不能访问当前模板的局部变量**，只共享全局变量，除非使用`with context`选项[37]。宏名以下划线开头会被视为私有，无法被导入[38]。

Include 包含：`{% include '模板名' %}`指令用于在当前模板中插入并渲染另一个模板的内容，相当于在编译后将目标模板嵌入此处[39]。它与模板继承不同，**不会**建立父子关系，只是简单地嵌入内容。`include`非常适合用于复用一些标准片段，例如把提示模板中通用的开头结尾、固定格式的段落提取出来，通过包含来插入，避免重复编写。

例如，有独立的文件`header.jinja`和`footer.jinja`定义了提示的固定头尾，那么主提示模板可以这样写：

```
{% include 'header.jinja' %}
... （主体提示内容） ...
{% include 'footer.jinja' %}
```

被包含的模板默认可以访问当前模板的上下文变量[40]。如果不希望如此，可以使用`without context`参数使其在独立上下文中渲染[40]。例如：`{% include 'sidebar.jinja' without context %}`会在没有当前变量的情况下渲染`sidebar.jinja`（通常用于防止变量名冲突或不希望子模板使用某些数据）。反之也可写`with context`明确传递（但`with context`是默认行为，一般不必写）[40]。

Include 还支持：使用`ignore missing`忽略模板不存在的错误[41]；传入一个模板名列表，按顺序尝试包含第一个存在的模板[42]；甚至传入一个模板对象或变量作为名称[43]。这些特性为提示模板的按条件组装提供了极大灵活性。例如，可以按优先级尝试包含不同详细程度的提示片段：

```
{% include ['detail_prompt.jinja', 'basic_prompt.jinja'] %}
```

上例会优先包含`detail_prompt.jinja`，如果不存在则退而求其次包含`basic_prompt.jinja`[44]。

宏 vs 包含：宏更像函数调用，适合需要传参、返回局部内容的情形；`include`则是直接嵌入完整模板的内容，更适合插入结构化的大段片段。在LLM提示工程中，可以将**可参数化的小模板片段做成宏**（例如上面的按钮、或格式化单轮对话的片段），将**固定结构的大段内容用模板文件拆分**，通过`include`组合。这两者搭配使用，可以让提示模板既保持**结构清晰**又具备**高度复用**能力。

4. 自定义过滤器和函数

Jinja2 内置的过滤器已涵盖许多常见需求，但在提示工程中，有时需要定制特殊的格式转换逻辑或工具函数。例如，将对话历史列表转换成特定格式的字符串，或对用户输入做特殊清理。**自定义过滤器**允许我们用Python编写函数并注册为模板过滤器，以在模板中像内置过滤器一样使用[45]。

定义和注册过滤器：过滤器本质上是接受输入值并返回新值的Python函数。第一个参数接收管道左侧的值，其余参数来自过滤器调用时提供的参数[46]。定义函数后，需要将其注册到环境的过滤器字典中，例如：

```
from jinja2 import Environment
env = Environment()

def emphasize(value, mark='**'):
    # 将文本用指定标记包围
    return f"{mark}{value}{mark}"

env.filters['emphasize'] = emphasize    # 注册过滤器
```

现在在模板中就可以通过`|emphasize`使用这个过滤器了，比如：`{{ "hello"|emphasize('$$') }}`将输出`$$hello$$`。官方文档示例演示了一个日期格式过滤器的定义和注册[47]：

示例： 定义函数

```
def datetime_format(value, format="%H:%M %d-%m-%y"):
    return value.strftime(format)
```

并注册：`environment.filters["datetime_format"] = datetime_format`。之后模板中可用 `{{ article.pub_date|datetime_format("%B %Y") }}` [47]。

Jinja2 还提供了一些装饰器（如 `@pass_environment`，`@pass_context` 等）用于让过滤器函数获取模板环境、评估上下文等附加信息 [48]。在提示工程中，比较实用的是通过这些机制访问 `autoescape` 状态，以确保在需要时对内容进行转义处理 [49]。不过大多数自定义过滤器不需要这些高级特性，只需按需处理值并返回即可。

自定义全局函数：除了过滤器，有时我们希望在模板中调用某些辅助函数（非基于管道、可能执行复杂逻辑）。可以利用 Jinja2 的**全局变量**机制，将Python中的函数注册为模板全局变量，然后在模板中当作函数使用 [50]。实现方法是将函数添加到 Environment 的 `globals` 字典。例如：

```
def calculate_risk_level(score):
    return "high" if score > 80 else "low"

env.globals['risk_level'] = calculate_risk_level
```

注册后，模板中可以直接调用：`Risk level: {{ risk_level(user_score) }}`，Jinja2 会执行对应的Python函数。这种方式对**非纯文本处理**或需要执行复杂计算/数据库查询的功能特别有用。不过要注意，Jinja2 模板引擎本身并不限制 global 函数的行为，它可以调用任意Python代码，因此**切勿注册来源不明或不安全的函数**，以免造成安全隐患。

实际上，如果只是要在模板中调用Python的内置函数或简单操作，也可以直接将它们放入上下文或globals。例如，可以把 `str.upper` 函数加到globals然后在模板中调用。不过为了保持模板简洁、逻辑与表示分离，建议复杂运算尽量在Python端处理好，将结果直接作为变量提供给模板。如果必须在模板中调用，才使用这种全局函数机制。

自定义测试：类似地，Jinja2 也允许注册自定义测试，用于 `{% if value is test %}` 这样的语法。注册方法是将函数添加到 `Environment.tests` 字典 [51]。在提示工程中自定义测试不常用，此处不展开。

总结：通过自定义过滤器和全局函数，Jinja2 模板功能可以由开发者扩展到满足各种特殊需求。在LLM提示构建中，这意味着可以方便地封装**内容清洗、格式转换、复杂逻辑判断**等操作为模板可用的过滤器/函数，使模板既保持简洁又具备所需的定制能力。例如，可以实现一个过滤器将用户输入中的敏感词替换为特殊符号，然后在模板中对用户输入统一使用此过滤器，以确保提示安全合规。这些扩展增强了模板的表现力和实用性。

5. Jinja2 与 Python 的集成

将 Jinja2 应用于实际项目时，需要在Python代码中加载模板、传递数据并渲染输出。典型流程如下：

1. **创建环境（Environment）：**Environment是Jinja2的核心对象，包含配置、已注册的过滤器/函数、以及模板加载器等 [52]。通常在应用初始化时创建一个Environment实例。例如：

```
from jinja2 import Environment, FileSystemLoader
env = Environment(loader=FileSystemLoader('templates'),
                  autoescape=False)
```

上述代码创建了一个Environment，配置从 `templates` 目录加载模板，并关闭自动HTML转义（因为LLM提示一般非HTML格式）。你也可以根据需要设置 `trim_blocks`、`undefined=StrictUndefined` 等配置来调整模板渲染行为。

1. **加载模板：**使用Environment提供的API加载模板文件。最常用的是 `env.get_template('模板名')`，它会使用配置的加载器去找到模板文件并编译返回一个Template对象 [53] 。例如：

```
template = env.get_template("prompt.txt")
```

如果模板文件不存在，`get_template` 会抛出异常。加载模板发生在首次请求时，Jinja2 会缓存已加载的模板以提高性能（缓存大小默认400个，可配置）。

1. **渲染模板：**调用Template对象的 `render` 方法，传入上下文字典或关键字参数，即可得到最终渲染的字符串 [53] 。例如：

```
output_text = template.render(user_name="Alice", score=95)
print(output_text)
```

上面将输出替换了 `user_name` 和 `score` 后的内容 [53] 。`render` 方法接收的参数既可以是显式的关键字参数，也可以传入单个字典（如 `template.render(context_dict)`）。渲染返回的是生成的完整字符串，后续可以将其发送给LLM接口。

以下是一个完整示例，将上述步骤串联起来（假设已有模板文件 `prompt.jinja`）：

```
from jinja2 import Environment, FileSystemLoader

# 创建环境，指定模板文件目录
env = Environment(loader=FileSystemLoader('templates'))

# 加载模板
template = env.get_template('prompt.jinja')

# 上下文数据
data = {
    "user": "Alice",
    "messages": [
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": "Hello, who are you?"}
    ]
}

# 渲染模板
```

```
prompt_text = template.render(data)
print(prompt_text)
```

与Jinja2集成时，有几点实用技巧：

- **模板字符串渲染：**如果你的模板不是存在于文件而是动态生成的字符串，可以使用`Environment.from_string(template_str)`直接创建Template，然后`render`。但需要注意，从字符串创建的模板无法参与继承关系[54]，也不会被缓存。一般推荐尽量使用文件加载模板，因为**更易管理且支持模板继承**[54]。

- **Template　类：**也可以直接使用底层的`jinja2.Template`类构造模板，例如：`Template("Hello {{name}}!").render(name="Bob")`。内部会隐式创建一个共享的Environment[55]。这种方式适合极简场景或演示，但在实际项目中建议手动管理Environment，以便进行统一配置、缓存和扩展注册。

- **多次渲染：**加载后的Template对象是可重用的。对于需要多次生成提示的场景（比如对不同用户使用相同模板），无需重复加载模板文件，每次只需调用`render`传入不同数据即可[56][57]。Jinja2默认启用了模板文件自动重载（auto_reload=True），每次`get_template`会检查文件是否更新，开发时方便调试，但在生产环境可考虑关闭提高性能。

- **沙箱环境：**如果你需要在应用中让用户提供自定义的模板，为了安全应使用`jinja2.sandbox.SandboxedEnvironment`（受限环境）来创建模板[58]。普通的Environment中，模板具有执行任意过滤器和表达式的能力，恶意模板可能导致**任意代码执行**[58]。沙箱环境会禁止访问不安全的属性和操作。尽管如此，**强烈建议不要渲染不受信任的模板**[59]。在LLM应用中通常不存在让终端用户上传Jinja模板的需求，如果有（例如高级用户自定义提示格式），一定要做好权限和安全审查。

总之，将Jinja2与Python集成非常简单：创建环境、获取模板、传入数据渲染。这种模式也很容易和现有框架集成，比如在Flask或FastAPI应用中使用模板管理LLM提示，或结合LangChain的PromptTemplate机制（LangChain支持Jinja2模板格式，但官方也出于安全考虑更推荐简单的f-string格式[58]——如果使用Jinja2模板，务必确保模板来源可信）。

6. 文件加载器：FileSystemLoader vs DictLoader

模板加载器（Loader）：Environment依赖加载器从各种来源获取模板文件。Jinja2内置了多种加载器，常用的包括FileSystemLoader、PackageLoader、DictLoader等[60][61]。理解加载器的区别有助于根据项目需求选择合适的方式组织和提供模板。

- **FileSystemLoader（文件系统加载器）：**从**文件系统目录**加载模板[62]。初始化时需要指定一个路径（或路径列表）作为模板搜索目录[62]。例如：`FileSystemLoader("templates")`表示在当前工作目录下的`templates`文件夹查找模板[63]。也可以提供多个目录列表，按顺序查找[64]。FileSystemLoader适用于模板存放在磁盘文件的典型场景，比如将所有提示模板放在项目目录下，方便独立管理和修改。它支持子目录结构，使用`子目录/模板名`即可引用子文件夹内的模板[65]。FileSystemLoader默认启用了**自动重载**（Environment.auto_reload=True时），修改模板文件后再次渲染会自动重新加载最新内容，对于开发调试非常便利[66]。

- **DictLoader（字典加载器）：**从一个Python字典中加载模板[67]。使用时需传入一个字典映射，键是模板名称，值是模板源字符串[67]。例如：

```
templates_dict = {
    "greeting.txt": "Hello, {{name}}!",
    "farewell.txt": "Goodbye, {{name}}."
}
env = Environment(loader=DictLoader(templates_dict))
```

这样可以像文件一样使用`env.get_template("greeting.txt")`获取模板。DictLoader的特点是**将模板内嵌在代码中**，不依赖外部文件，常用于**单元测试**或示例演示 [67]。由于模板通常固定不变，DictLoader默认关闭了自动重载（即假定字典内容不会变化） [68]。在提示工程场景，如果你的模板是由上层存储（数据库、配置文件等）动态提供的，也可以在运行时构造字典然后用DictLoader加载。不过，对于大规模或复杂模板，不太建议用代码内嵌字典管理，因为可读性和维护性较差。

- **PackageLoader（包加载器）：**从Python包中的`templates`目录加载模板。它需要提供包名和包内相对路径 [69]。这通常用于将模板与应用代码一起打包发布。对LLM提示开发而言用到的较少，除非你将提示模板做成一个可分发的包。

- **其他加载器：**还有FunctionLoader（使用自定义函数加载模板）、PrefixLoader（组合加载器加前缀区分命名空间）以及ChoiceLoader（按顺序尝试多个加载器）等 [61] [70]。例如ChoiceLoader可用于实现**多来源模板查找**：先从用户自定义路径找模板，找不到再从默认路径找 [70]。这些高级加载器在需要高度动态化模板来源时才用得到，一般情况下FileSystemLoader足矣。

FileSystemLoader vs DictLoader：二者最主要区别在于模板来源：一个从文件系统读取，适合实际项目部署；一个从内存字典读取，适合测试或内嵌模板场景。FileSystemLoader让模板与代码分离，便于**多人协作和修改**（模板改动无需改代码，只要文件可部署更新即可）。DictLoader则胜在**使用方便**，不需要额外的文件部署步骤（模板可以随着代码一起发布）。从性能上看，如果模板需要频繁更新，FileSystemLoader结合自动重载机制会定期访问文件时间戳检测变化，而DictLoader的模板存在内存中访问开销小一些。不过在LLM提示构建里，这种性能差异可以忽略不计，因为渲染一次模板相对LLM调用本身的开销非常小。

一般来说，**开发调试阶段**可以用FileSystemLoader方便地调整模板；**写单元测试**时可以用DictLoader快速提供模板字符串。也可以混用ChoiceLoader让程序同时支持从文件和内置字典加载。在确定模板不会再频繁修改后，如何部署可以按团队方便来定。

7. 提示工程实践建议

将以上Jinja2功能应用到LLM提示词模板构建，可以极大提升开发效率和提示质量。在实际使用中，建议参考以下策略：

（1）模块化设计多段提示：对于复杂的提示场景，可以将提示拆分成多个模板片段，通过**继承或包含**组合。比如一个对话机器人提示通常包含系统角色说明、若干示例对话（few-shot），以及用户最新提问。可以设计：- `system.jinja`：系统角色提示模板；- `fewshot.jinja`：few-shot示例对话模板，内部用宏或循环生成多个示例；- `user_prompt.jinja`：用户提问模板；- `main_prompt.jinja`：主模板，extends基础结构并`include`上述片段，或者直接顺序包含它们。

通过这种多模板组合，**各部分职责清晰**，方便独立修改 [71] [72]。例如想调整few-shot示例格式，只需编辑`fewshot.jinja`无需动主模板。同时，利用宏可以封装对话消息的渲染格式，然后few-shot和实际对话都调用相同宏，保证风格一致。

（2）变量注入与上下文管理：在生成最终提示时，需将上下文数据（如用户问题、先前对话历史、外部知识等）注入模板。确保在调用`render`时传入**完整所需的变量**集合。使用`Template.render(**data)`时，如果某

些变量缺失，Jinja2默认会当成空字符串，这可能导致提示内容缺失而不易察觉[6]。为避免这种情况，建议在开发调试时将Environment配置为`undefined=StrictUndefined`，让缺失变量抛出异常[7]。这样可以及早发现上下文准备的问题。另一种方式是在模板中为可选变量使用`default`过滤器提供默认值，例如：`{{ user_name|default("User") }}`[73]。

当提示涉及**列表或多轮对话**注入时，通常会在模板用`for`循环遍历。比如注入最近几轮对话列表，可以：

```
{% for msg in history %}
{{ msg.role|capitalize }}: {{ msg.content }}
{% endfor %}
```

这样模板就能根据历史列表长度自动适配。若希望对话过长时截断或采用特殊格式，也可在Python端预处理列表后再传给模板，或编写过滤器在模板中处理。

（3）安全处理和内容清理：提示词模板本身是纯文本，不涉及HTML等富内容，但仍需注意安全。**不要**直接在模板中插入未经处理的用户提供内容，以防止意外的提示注入或不良字符。例如，可以创建一个过滤器对用户输入进行清理（去除控制字符或某些敏感词）并应用于模板插值。另一方面，如果你的应用允许外部人员编辑或上传模板，一定要**限制模板功能**。Jinja2模板可以执行一些Python代码（例如调用全局函数，访问属性等），如果被不信任的人控制模板内容，可能造成严重后果[58]。对此应： - 使用受限的SandboxedEnvironment渲染不可信模板[74]。 - 或在应用层严格控制：仅允许选择预定义模板，不允许任意上传模板文本。

此外，由于LLM对输入格式和内容非常敏感，确保模板渲染结果**没有语法错误或不必要的空格**。可以利用Jinja2的**空白控制**特性（如在`{% for %}`等标签后加`-%`去除换行[75]）来紧凑输出，避免因为无关空行或缩进导致LLM理解偏差。调试提示时，可以打印输出的完整提示文本进行检查，必要时通过`repr()`看不可见字符。

（4）调试和迭代：使用Jinja2编写提示模板，可以借助几个技巧加速迭代： - 在模板中使用注释`{# ... #}`标记不同部分，说明仅供开发参考，不会影响LLM输入[76]。 - 利用小的测试上下文反复渲染模板验证格式。比如构造一个模拟对话历史列表，查看渲染后的提示是否符合预期。 - 如前文所述，开启StrictUndefined保证每个占位符都有值。或者使用`{{ variable|default('[MISSING]') }}`在渲染结果中高亮出缺失的变量[73]。 - 善用模板继承/包含将问题拆分，逐个部分验证输出，再组合整体。这样出问题时容易定位是哪个片段的逻辑导致。

（5）性能考虑：Jinja2 渲染本身非常快，每秒可处理数千到数万次模板渲染[77]。对于绝大多数LLM应用（请求量相对有限）来说，模板渲染几乎不是性能瓶颈。不过如果你的应用需要频繁生成提示，可以： - 复用Template对象，避免重复编译模板； - 关闭自动reload（`auto_reload=False`），减少文件系统检查； - 如果模板非常复杂，可考虑开启字节码缓存，将模板编译结果存储起来以提升后续加载速度[66]（Jinja2支持将字节码缓存到文件或内存）。

总之，Jinja2 提供了强大的模板功能来辅助 LLM 提示构建。通过良好的结构设计和谨慎的实践，你可以用它构建**灵活、可维护、安全**的提示模板体系。例如，一套模板可根据用户信息、上下文动态拼装不同提示，实现**个性化**和**上下文相关**的对话体验。希望本教程的介绍能帮助开发者充分发挥 Jinja2 在提示工程中的作用，让 Prompt Engineering 更加高效从容！

参考资料：Jinja2 官方文档[78][19][47]，PromptLayer 博客对 Jinja2 提示模板的介绍[79][80]，LangChain 关于 Jinja2 模板的安全提示[58]等。

---

1 71 72 73 77 79 80 Prompt Templates with Jinja2
https://blog.promptlayer.com/prompt-templates-with-jinja2-2/

2 3 4 5 6 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 65 75 76 78 模板设计器文档 — Jinja 文档 (3.1.x) - Jinja 模板引擎
https://jinja.flask.org.cn/en/3.1.x/templates/

7 52 54 55 60 66 69 API — Jinja Documentation (3.1.x)
https://jinja.palletsprojects.com/en/stable/api/

45 46 47 48 49 50 51 53 61 62 63 64 67 68 70 API — Jinja 文档（3.1.x） - Jinja 模板引擎
https://jinja.flask.org.cn/en/3.1.x/api/

56 57 Primer on Jinja Templating – Real Python
https://realpython.com/primer-on-jinja-templating/

58 59 74 PromptTemplate — LangChain documentation
https://python.langchain.com/api_reference/core/prompts/langchain_core.prompts.prompt.PromptTemplate.html