ChatGPT

YAML 中文教程

YAML（**Y**AML **A**in't **M**arkup **L**anguage）是一种为人类可读性优化的数据序列化格式，广泛用于编写配置文件和结构化文档 [1]。相比 JSON，YAML 更简洁强大，支持更直观的嵌套结构和注释，因而在系统配置、环境变量设置以及 AI 提示词工程等场景下受到青睐 [1]。本教程面向开发者和提示词工程实践者，系统讲解 YAML 语法规则和进阶用法，并结合配置管理、Prompt 模板等应用场景提供示例。内容涵盖：基本语法、标量类型、序列和映射、多文档结构、锚点与别名、多行文本块，以及 YAML 在提示词模板系统中的实践技巧。

基本语法规则

YAML 以**缩进**表示层次结构，语法简洁明了。以下是 YAML 的基础规则：

- **大小写敏感**：键和值区分大小写 [2]。
- **缩进用空格**：只能使用空格缩进，**禁止使用 Tab 键**，否则会引发语法错误 [2]。
- **缩进层级**：缩进的空格数目不固定，但同级元素必须左对齐对齐。通过空格缩进来表示嵌套关系 [2]。
- **冒号与空格**：字典（映射）中的键值对使用「`键：值`」表示，**键名后紧跟冒号和一个空格**再写值 [3]。
- **连字号列表**：列表（序列）项以前导连字号“-”表示，连字号后需跟一个空格再写该项的值 [3]。
- **不强制引号**：一般字符串值不需要加引号，除非包含特殊字符（如 `: # { } [ ] , & *` 等）或者以**空格开头**等特殊情况 [4]。如字符串中含有冒号、井号等符号，或想保留字符串两端的空白，则应使用引号包裹。
- **注释**：使用「`#`」引入单行注释，从 `#` 开始直到行尾的内容都会被解析器忽略 [2]。

示例：以下是一段 YAML 配置片段，演示缩进和注释规则：

```yaml
# YAML示例 - 表示一个简单的应用配置
app: MyApplication      # 应用名称（字符串）
version: 1.0            # 版本号（数字）
debug: true             # 是否开启调试模式（布尔值）
database:               # 数据库配置（映射）
  host: localhost       # 主机名
  port: 5432            # 端口
  user: admin           # 用户名
  password: "s3cr3t"    # 密码（特殊字符，使用引号）
features:               # 功能开关列表（序列）
  - login
  - analytics
  - "user-profile"      # 带连字符的字符串，建议加引号
```

上例中，`app`、`version` 等是顶层键，`database` 下的 `host`、`port` 等通过缩进属于其子键。布尔值 `true` 和整数 `1.0`（实际解析为浮点数 1.0）无需引号。`password` 的值包含数字和特殊字符，因此使用了引号。注释从 `#` 开始，用于解释各配置项。

标量类型（字符串、数值、布尔和 Null）

YAML 支持多种**标量**（scalar）数据类型，包括字符串、整数、浮点数、布尔和 Null 等等[3]。标量是最基本的不可再分解的值。在 YAML 编写配置时，正确表示不同类型的标量值十分重要。

- **字符串（string）**：默认不需要使用引号包裹[5]。如果字符串中包含空格或特殊符号，建议用单引号 `'...'` 或双引号 `"` 包裹以避免歧义[6]。单引号会将内容原样表示，唯一的特殊情况是连续两个单引号 `''` 会被解析为一个单引号字符；双引号允许转义特殊字符（例如 `\n` 表示换行）。
  示例： `title: YAML 简介` 表示普通字符串，`path: "C:\\Program Files\\App"` 用双引号表示包含反斜杠的路径。需要注意，**不要使用未加引号的特殊值**作为字符串，例如 `yes`、`off` 等，因为未引号的 `yes`/`no` 等可能被当作布尔值解析（详见下文）。

- **整数和浮点数（number）**：可以直接写数值字面量，如 `count: 42`、`pi: 3.1415`。YAML 中还支持多种进制表示，例如二进制、八进制、十六进制等（如 `0b1010`，`0xFF`），以及科学计数法表示浮点数（如 `6.852e+5`）[7]。但在大多数配置场景下，直接使用十进制即可。如需表示负数，直接在数字前加 `-` 号，例如 `-5`。**注意：**数字如果加引号将被视为字符串而非数值。

- **布尔值（boolean）**：使用 `true` 和 `false` 来表示布尔值（不区分大小写，例如 `True` 或 `FALSE` 也被识别为布尔值）[8]。例如：`debug: false`、`enabled: TRUE`。
  提示：YAML 1.2 规范中不再将 `yes`、`no`、`on`、`off` 等识别为布尔值，以避免歧义[9]。例如在 YAML 1.1 早期版本，`yes` 会被当作真值，这在实际配置中可能引发误解。因此**建议始终使用 `true`/`false` 表示布尔值**。如果确实需要使用单词“yes”等作为字符串值，请用引号包裹以确保按照字符串处理。

- **Null 值**：使用 `null` 或波浪号 `~` 表示空值[10]。两个表示等价，解析后都会得到语言中的 `null`/`None`。例如：`middle_name: null` 表示 middle_name 为空；也可以写作 `middle_name: ~`。**注意：**空白值（键后不跟任何内容）在 YAML 中也被视为 Null。例如写成 `middle_name:`（冒号后直接换行）同样表示 Null 值。

YAML 解析器会根据这些字面量自动将值转换成相应的数据类型[11]。例如，不加引号的数字会被当作数值类型，`true`/`false` 会成为布尔类型，`~` 会变为 Null。在 Python 的 PyYAML 库中，解析以上示例时会得到对应的 Python 对象类型：字符串对应 `str`，整数为 `int`，浮点为 `float`，布尔为 `bool`，Null 对应 `None` 等[11]。如果需要强制某值为特定类型，YAML 允许使用类型标签（如 `!!str` 或 `!!int`）显式指定类型，但这属于进阶用法，一般配置场景很少用。

示例：下面的 YAML 展示各种标量类型的表示方法：

```yaml
# 标量类型示例
site_name: 示例站点        # 字符串
maintenance: OFF         # 未加引号的特殊串“OFF”，旧版解析器可能视为布尔false，不推荐
maintenance_str: "OFF"     # 加引号，明确为字符串 "OFF"
max_users: 1000          # 整数
threshold: 0.75          # 浮点数
hex_value: 0xFF          # 16进制整数，等同于255
is_active: true          # 布尔值 true
has_terminated: False      # 布尔值 false，不区分大小写
start_date: 2025-07-24     # 日期（ISO8601格式）
end_time: 2025-07-24T18:30:00+08:00 # 日期时间（ISO8601格式，含时区）
```

```
unknown_value: null       # Null值
alternate_null: ~         # Null值的另一种表示
```

上例中值得注意的是：`maintenance: OFF` 未加引号，某些 YAML 解析器可能把 `OFF` 当作布尔假值；为了避免歧义，我们通过 `maintenance_str: "OFF"` 将其作为字符串明确表示。同样，日期和日期时间使用 ISO8601 标准格式，在很多解析库（如 PyYAML）中会自动识别为日期类型或字符串（具体取决于实现），因此在需要时也可加引号确保其按字符串处理。一般来说，**对可能被误解为其他类型的值，加引号是保持稳妥的做法** [9]。

序列和映射（列表与字典）

YAML 通过**序列**和**映射**来表示复合的数据结构，这对应编程语言中的列表（数组）和字典（映射表） [3]。编写配置文件时，经常需要将多个值组织成列表，或使用键值对来配置选项。下面分别介绍序列和映射的写法。

序列（列表）

序列（sequence）表示一组按顺序排列的值，也称作列表或数组。YAML 用连字号（`-`）开头表示列表项，每个 `-` 后需跟一个空格再写该项值 [3]。列表项可以是标量，也可以是嵌套的对象或其他列表。

- **基本写法（块格式）**：将每个元素各占一行，并以 `-` 开头。例如：

```
shopping_list:
 - Milk
 - Bread
 - Eggs
```

上述 YAML 中，`shopping_list` 对应一个列表，包含 `"Milk"`、`"Bread"`、`"Eggs"` 三个元素 [12]。注意列表项要比父键缩进，通常缩进两个空格。解析后相当于 JSON 的 `"shopping_list": ["Milk", "Bread", "Eggs"]` [13]。

- **嵌套列表**：如果列表的元素本身还是列表，可以在该项下继续缩进编写子列表。例如：

```
matrix:
 - - 1
   - 2
 - - 3
   - 4
```

这表示一个二维列表（列表的列表）。其中 `matrix` 第一项是 `[1, 2]`，第二项是 `[3, 4]` [14] [15]。通过缩进区分层级：最左侧的 `-` 表示 `matrix` 的元素，每个元素下又有自己的子项列表。

- **行内写法（流式）**：使用方括号 `[]` 包围列表，逗号分隔元素，也可以在一行内或跨行编写。例如：

```
colors: [red, green, blue]
aliases:
 - [alice, "bob", carol]
```

`colors` 展示了在同一行以 `[red, green, blue]` 表示列表。`aliases` 列表的第一项本身又是一个列表 `["alice", "bob", "carol"]`，使用方括号表示 [16]。行内表示法与 JSON 数组语法几乎一致，在元素较多时可提高紧凑性，但可读性稍差，通常简单数组可一行表示，复杂嵌套结构仍建议使用块格式。

在 YAML 中，列表没有固定的长度限制，可以自由添加 `-` 项。另外，**列表项的缩进**只需比列表所在键多一级即可，不要求统一缩进几个空格，只要同级对齐即可。例如：

```
# 序列示例
servers:
 - host: "example.com"
   port: 80
 - host: "example.org"
   port: 443
```

这里 `servers` 列表包含两个元素，每个元素是一个映射（对象），分别有 `host` 和 `port` 两个子键。两个元素用 `-` 区分，**且在同一缩进层级对齐**。解析结果相当于一个长度为2的列表，包含两个字典：`[ {"host": "example.com", "port": 80}, {"host": "example.org", "port": 443} ]`。

映射（字典）

映射（mapping）表示一组键值对集合，也称为字典或哈希表。YAML 使用「`键: 值`」的形式表示映射中的一对数据 [3]。映射可以用于配置命名的参数和属性，非常常见于配置文件中。

- **基本写法（块格式）**：每对键值占一行，键和值以冒号 `:` 分隔，并**在冒号后加一个空格**再写值 [3]。例如：

```
database:
  host: localhost
  port: 5432
  user: admin
  password: secret
```

上述 YAML 中，`database` 键对应一个映射对象，其下缩进的四行分别是子键 `host`、`port`、`user`、`password` 及其对应的值。冒号与值之间有空格分隔，如 `host: localhost`。[17] 解析结果类似 JSON 的：

```
"database": { "host": "localhost", "port": 5432, "user": "admin", "password": "secret" }
```

- **嵌套映射**：映射的值也可以是另一个映射，用于表示多层次配置。通过进一步缩进来表示子映射。例如：

```
app:
  name: DemoApp
  logging:
    level: INFO
    file: /var/log/demo.log
```

其中 `app` 是顶层键，值是一个映射，包含 `name` 和 `logging` 两个子键。其中 `logging` 自身又是一个映射，包含 `level` 和 `file`。[18] 通过缩进清晰地表达了层次关系。解析为 Python 字典则是嵌套结构：`app = {"name": "DemoApp", "logging": {"level": "INFO", "file": "/var/log/demo.log"}}`。

- **复杂键名**：通常键是简单字符串，可以不加引号。但如果键名本身包含空格或特殊字符，需要特殊处理。YAML 提供**问号前缀**语法来定义复杂键：在一行以 `?` 开头，空格后跟键名值，下一行以 `:` 表示该键的值。例如：

```
?
 key with spaces: value1
:
 some value
```

这种语法较少用到，更简单的做法是直接将复杂键加引号作为普通键使用，例如写作 `"key with spaces": "some value"`。多数情况下，引号键名比问号语法更直观。

- **行内写法（流式）**：使用花括号 `{}` 表示映射，键值对以冒号和逗号分隔。格式类似 JSON 对象。例如：

```
default_user: {name: "guest", groups: [read, write]}
```

这与 JSON 等价于：`"default_user": {"name": "guest", "groups": ["read", "write"]}` [19]。行内写法在映射很简单时可以一行表示，但一行包含太多内容会降低可读性，因此复杂映射通常使用块格式按行列出。

映射在 YAML 中不要求键按字母序，书写顺序不会影响解析结果。但为了提升可读性，建议对相关的键分组或按逻辑顺序排列。同时要注意，同一映射中**键必须唯一**，重复的键会导致解析器取最后一个值或者报错，需避免这种情况。

示例：以下 YAML 片段展示映射的多种形式：

```
# 映射（字典）示例
person:
 first_name: Alice      # 简单键:值
 last_name: Smith
 contact:
  email: "[email protected]"  # 嵌套映射
  phone: "+1-202-555-0101"
 address: { city: "New York", zip: 10001 }  # 行内映射
"job title": "Senior Developer"  # 键名包含空格，使用引号
? key with special chars @#$:
: "needs special syntax"      # 复杂键写法（一般用引号更简单）
```

上例定义了一个 `person` 字典，其中包含简单键、嵌套的 `contact` 子字典，以及一个行内写法的 `address` 子字典。另外演示了键名带空格和特殊字符的两种处理方式：`"job title"` 用引号包裹作为键，`key with special chars @#$` 则用问号语法（换行缩进）定义。实际项目中，倾向于使用**引号键**来避免复杂语法。

组合和嵌套结构

序列和映射可以任意组合嵌套，构成复杂的层次数据结构。常见如：列表的元素是映射，映射的某个值是列表等。在 YAML 中，这通过缩进很好地可视化结构 [20] [21]。上面的示例已经多次体现嵌套用法。再举一个综合例子：

```
projects:
 - name: Alpha
   team:
     - Alice
     - Bob
 - name: Beta
   team: [Carol, Dave]
```

这里顶层键 `projects` 是一个序列，其中每个元素是映射（包含项目名称和团队成员列表）。`Alpha` 项目的团队用块格式列出成员，`Beta` 则用行内格式列出。两种表示方式是等价的。这样的结构在 YAML 中直观易读，转成 JSON 则是：

```
{
  "projects": [
    { "name": "Alpha", "team": ["Alice", "Bob"] },
    { "name": "Beta", "team": ["Carol", "Dave"] }
  ]
}
```

可以看到 YAML 通过缩进和格式灵活性，让复杂数据结构的表示更加简洁清晰 [22] [23]。这也是 YAML 特别适合作为**数据结构描述**用途的原因，例如定义多层嵌套的配置对象或 API 模板等。

多文档结构（`---` 分隔）

一个 YAML 文件中可以包含**多个文档**。文档之间使用一行由三个连字号构成的分隔符`---`来隔开 [24]。这在需要将多段配置集中在一个文件时非常有用。例如，在 Kubernetes 配置中经常将 Deployment 和 Service 等多个资源定义写入同一个 `.yaml` 文件，通过 `---` 分隔为多个文档一并提交。

要点：

- 位于文件开头的第一个`---`标记是可选的；但如果文件中包含多个文档，则**除了第一个文档外**，后续每个文档必须以前置`---`标明开始 [25]。
- 文档分隔符独立占一行，后面的内容从下一行开始属于新文档。如果文档以映射或序列结构开始，应另起一行缩进编写其内容 [26]。
- YAML 规范还定义了可选的文档结束标记`...`（三个连续的点号）。当一个文档并非文件末尾或者你希望明确结束时，可以在文档结尾单独一行写`...` [24]。通常这种用法并不常见，除非在流式传输场景下需要显式提示文档结尾 [27] [28]。

示例：假设我们有两个环境的配置想放在一个 YAML 文件里，可以这样写：

```
# multi_env.yaml
---
env: development
debug: true
database: sqlite:///dev.db

---
env: production
debug: false
database: postgresql://db.prod.example.com/prod
...
```

这里 `---` 将文件分成两个 YAML 文档：第一个文档用于开发环境配置，第二个文档用于生产环境配置。每个文档都是独立的 YAML 数据结构。可见在第二个文档末我们还用了 `...` 表示文档结束（非必须）。解析时，如在 Python 中使用 PyYAML，`yaml.safe_load_all` 可一次加载所有文档并返回生成器或列表 [29]。例如：

```
import yaml
with open('multi_env.yaml', 'r', encoding='utf-8') as f:
    docs = list(yaml.safe_load_all(f))
# docs 将是包含两个字典的列表，对应两个文档
print(docs[0]['env'])  # 输出 'development'
print(docs[1]['database'])  # 输出 'postgresql://db.prod.example.com/prod'
```

需要注意，**锚点和别名（见下节）不能跨文档引用**。每个文档的锚点作用域仅限于所在文档，文档之间互相隔离 [30]。因此，用 `---` 分隔多个文档时，如果需要重复某些结构，可以在各文档内分别定义锚点。

多文档结构在日常配置中相对少见，但在某些批量配置场景（如 Kubernetes、CloudFormation 等）以及需要在单文件中提供多段 Prompt 示例的场景下（如将不同阶段的提示存在一处）非常有用。合理使用可提高配置组织的紧凑性，但也要防止混淆，保持每个文档的独立性和清晰。

锚点与别名（重复内容复用）

YAML 提供了**锚点（anchor）**和**别名（alias）**机制，使我们可以复用已有定义的内容，从而避免重复编写相同配置片段。在复杂配置文件中，这一特性非常实用，例如多个地方引用相同的阈值参数，或不同环境共享大部分相同配置时。

- **锚点（`&`）**：锚点用于给一个节点（标量、序列或映射）起一个名字，之后就可以引用它。语法是在节点后加 `&锚点名`。锚点名可以由字母、数字等组成（不能含空格和部分特殊字符），通常以有意义的名称命名。
- **别名（`*`）**：别名用于引用先前定义的锚点，语法是 `*锚点名`。引用会插入锚点对应的值，就好像复制粘贴一样。

示例1：映射锚点复用 – 典型用例是定义一份默认配置，然后在多个地方复用并可局部覆盖某些值：

```
defaults: &defaults      # 定义锚点defaults，内容是一组通用配置
 adapter: postgres
```

```
  host: localhost
  port: 5432

development:
  database: myapp_dev
  <<: *defaults          # 合并锚点引用处的所有键值
  port: 5433           # 覆盖端口为开发用5433

test:
  database: myapp_test
  <<: *defaults          # 复用defaults锚点下的所有配置
```

上述 YAML 中，我们用 `&defaults` 将默认配置锚定，包含数据库适配器和主机等 [31]。然后 `development` 和 `test` 映射中，通过特殊键 `<<: *defaults` 将该锚点内容**合并**到当前映射 [31]。结果相当于在 `development` 和 `test` 下都展开出 adapter、host、port 等相同配置，其中 `development` 又覆盖了 `port` 为5433。解析后的等价结构为：

```
defaults:
  adapter: postgres
  host: localhost
  port: 5432
development:
  adapter: postgres
  host: localhost
  port: 5433
  database: myapp_dev
test:
  adapter: postgres
  host: localhost
  port: 5432
  database: myapp_test
```

可以看到，利用锚点和合并键 `<<`，我们避免了为每个环境重复写三行相同配置 [32]。YAML 规范规定，`<<: *anchor` 用于将锚点所引用映射的所有键值对合并到当前映射中 [33]。如果出现同名键，当前映射的值（如上例的 `port: 5433`）会覆盖锚点合并进来的值。

- **示例2：序列和标量锚点** – 锚点并不限于映射，也可用于序列或标量值。例如，我们想在列表中多次使用同一个元素：

```
- &admin_user alice
- bob
- carol
- *admin_user
```

在这个列表中，第一项定义了锚点 `admin_user` 对应值 `"alice"`，第四项使用 `*admin_user` 引用了这个值 [34]。结果该列表等价于 `['alice', 'bob', 'carol', 'alice']` [35]。也就是说，列表第4项通过别名再次插入了“alice” [34]。同样地，可以为序列整个列表起锚，或引用锚点作为映射的值。例如：

```
base_roles: &roles
 - reader
 - writer

user1:
 roles: *roles
user2:
 roles: *roles
```

这里将序列 `["reader","writer"]` 锚定为 `roles`，然后两个用户都别名引用它，使得两者的 `roles` 列表内容相同。这不仅节省书写，确保多个地方数据一致，也减少未来修改出错的机会。

重要提示：锚点引用在 YAML 中表现为**引用同一节点**而非复制独立副本。如果使用 PyYAML 等库解析，锚点和别名通常会指向同一个对象。 [36] [37] 例如，通过锚点让 `user1.roles` 和 `user2.roles` 都引用同一个列表对象，则在程序中修改其中一个列表内容，会影响另一个。一般在配置场景中我们视它们为常量数据，不会在运行时修改，因此这个实现细节不会有问题，但在代码中要谨慎这一点。

锚点和别名显著提升了 YAML 配置的**Don't Repeat Yourself**能力 [38] 。尤其在大型配置文件中，可以将公用的片段（数据库连接信息、公共提示文本等）定义一次，然后在不同部分引用。如果有需要改变的细节，可在引用处附近覆写特定键或值。需要注意锚点的作用域仅限于单个文档内，跨文档无效；另外锚点名不应重复，别名引用的名称必须是先前已定义过的锚点，否则解析会报错。

文本块样式（`|` 与 `>` 多行字符串）

在许多应用场景下，我们需要在 YAML 中书写**多行文本**，例如长段说明、日志模板，或多轮对话的 Prompt 内容。这时可以使用 YAML 的**块状字符串**语法，以保留或折叠换行等格式，更加适合书写大段文字。YAML 提供了两种块字面量表示法：**字面折行（Literal）**和**折叠换行（Folded）**，分别由符号 `|` 和 `>` 引入 [39] [40] 。

- **字面量块 `|`（Literal）**：保留块中写出的换行和空白格式，所见即所得。使用 `|` 开头新的一行，然后下一行开始缩进若干空格，后续缩进内容作为字符串的直接文本，包括换行符。换言之，`|` 引入的块会**按原样保留每行结尾的换行**。适合用来写诗歌、源码片段或希望严格保留格式的文本。

- **折叠块 `>`（Folded）**：将块中每个**非空行结尾的换行**转换为一个空格（折叠相邻行），而空白行（两个换行连在一起）则保留为真正的换行分段 [41] 。这种方式适合书写普通段落，YAML 会在解析时将其折叠成一段连续的字符串（除了显式空行处产生换行）。例如用于写多行提示词时，不希望在模型看来分成多行而只是长文本，可以用折叠样式将其合并成一行输出（但在源文件中我们分多行以便阅读）。

示例：比较 `|` 和 `>` 两种块文本的效果：

```
literal_block: |
 行1：你好，
 行2：这是一个演示。
 行3：再见。

folded_block: >
 Line1: Hello,
```

```
Line2: This is a demo.
Line3: Goodbye.
```

在上面的 YAML 中，`literal_block` 使用 `|` 定义了一个三行的中文文本块，每行都会保留换行。解析结果字符串将包含“行1：你好，\n行2：这是一个演示。\n行3：再见。\n”（末尾换行是否保留取决于块修饰符，默认会保留一个换行） [42] 。而 `folded_block` 使用 `>` 定义英文文本块，解析结果会将每行合并，在每行末尾加一个空格替代换行，得到 “Line1: Hello, Line2: This is a demo. Line3: Goodbye.\n” ——可见原来的三行连成了一行，中间以空格相隔，最后仅保留一个换行符 [43] 。

块字符串还有一些可选修饰符：例如在 `|` 或 `>` 后可以加 `-` 或 `+` 控制结尾换行的裁剪或保留，默认（不加符号）会保留一个换行，`-` 会去掉末尾所有换行，`+` 则保留所有结尾换行 [44] [28] 。还可以通过在 `|`/`>` 后加数字指明**剪裁公共缩进**，通常不需要手动指定，YAML 解析器会自动根据最小缩进量确定。

在实践中，如果要在 YAML 中嵌入一段**预先格式化好的多行内容**（如一封邮件模板、一个对话示例），用字面量块 `|` 是最佳选择，因为它完全保留格式。而如果只是在 YAML 中为了阅读方便把一段长句拆成多行，但并不希望这些换行在实际值中存在，则可用折叠块 `>`。

例如：在 Prompt 开发中，经常需要编写多行的系统提示词，我们可以：

```
system_prompt: |
 你是一个智能助理。
 当用户提出问题时，请给出详尽且礼貌的回答。
user_message: >
 用户说: "YAML 有什么用？"
 请帮我用简短语言解释。
```

这里 `system_prompt` 用 `|` 编写了两行文本，在最终字符串中会保留换行，因此模型接收时会分两行显示系统提示。而 `user_message` 用 `>` 折叠，虽然在文件中看起来有两行，但实际值会将 `"用户说： "YAML 有什么用？" 请帮我用简短语言解释。"` 连成一行（中间空格替代换行）。根据需要选择这两种格式，有助于在维护 Prompt 文件时既保证可读性又控制实际传给模型的格式。

提示：引号与块的交互

需要注意，**块标识符 `|` 和 `>` 自身已经定义了多行字符串的界限，因此块内容内部不需要也不应再加引号**。块语法也不能和引号同时使用。同样地，在块内部可以包含原本 YAML 会特殊处理的字符如 `:#` 等，都无需转义，都会作为文本的一部分被保留。

如果字符串中包含需要替换的变量占位符（如 `{name}` 或 `{{name}}`），依然可以放在块文本中，这些占位符在 YAML 解析阶段只是普通字符，不会干扰 YAML 结构。

YAML 在 Prompt 模板系统中的应用实践

随着大型语言模型（LLM）的**提示词工程**逐渐工程化，越来越多项目选择使用 YAML 来管理 Prompt 模板和参数 [45] 。YAML 的层次结构和可读性非常契合 Prompt 的组成，例如可以清晰地组织系统提示、用户提示、few-shot 示例、变量插值等内容 [46] 。本节结合实践经验，介绍如何利用 YAML 优雅地搭建 Prompt 模板系统。

YAML 与提示词的天然契合

- **层级结构清晰**：Prompt 往往由多部分组成，如系统提示（system prompt）、用户输入模板、上下文参数等。使用 YAML 可以很自然地通过嵌套结构表达这种层次 [22]。例如一个对话型 Prompt 可以用列表包含一系列消息字典，每个字典有角色和内容字段，从而直观表示多轮对话消息列表。

- **语义直观**：YAML 格式接近人类书写习惯，非技术人员（如产品经理）也能读懂和编辑简单的 YAML Prompt 文档 [47]。例如用缩进展示哪些变量属于哪个模块，管理者可以直接调整 YAML 文件而不必深入代码。

- **跨平台易解析**：YAML 有成熟库支持多种语言解析（Python 的 PyYAML / ruamel.yaml，JavaScript 的 js-yaml，Go 的 yaml.v3 等），便于将 Prompt 配置应用于不同环境 [47]。同时 YAML 文件方便进行版本管理和差异比较，适合作为 Prompt 内容的单一数据源。

使用 YAML 组织 Prompt 模板

通常，我们会在 YAML 中为每个 Prompt 定义一个结构化条目，包括模板文本、本 Prompt 所需的变量列表、描述信息等。例如：

```
prompt:
 name: classify_customer_intent  # Prompt标识名称
 description: >            # 描述使用折叠块
  对输入的用户问题进行意图分类，返回所属业务模块。
 input_variables:
  - question
 template: |             # 模板内容使用字面块
  请对以下用户提问进行意图分类，只返回以下几类中的一个：账单、报修、投诉、建议。
  用户问题：{question}
  输出格式：<意图类别>
```

如上，一个用于客户意图识别的 Prompt 定义，字段含义如下： [48] [49]

- `name`：Prompt 名称或标识符，用于引用或日志记录。
- `description`：对 Prompt 的简要说明，方便协作者理解用途。这通常是可选的说明性字段，用折叠块写多行描述文字。
- `input_variables`：列表形式列出本 Prompt 需要哪些变量输入（如这里需要 `question`）。这有助于在代码层面校验是否提供了必须的参数。
- `template`：真正的提示模板文本，通常是多行，包括固定提示和嵌入的变量占位符。在该例中，用 `{question}` 表示一个待注入的变量。

通过 YAML 结构，我们清晰地把 Prompt 的元信息和内容分开管理。多个 Prompt 可以各自写成上述结构，存在同一 YAML 文件的不同文档或同一文档的不同顶层键下。例如，可以将相关的一组 Prompt 放入一个文件，或每个 Prompt 一个独立 `.prompt.yaml` 文件 [50]。项目中常见做法是按模块建立目录存放 Prompt YAML，文件命名体现用途，如 `prompts/chat/example.prompt.yaml` [51]。

变量占位与注入

在 YAML Prompt 模板中，变量通常以某种占位符语法嵌入，如上例的 `{question}` 或 Jinja2 风格的 `{{ question }}` [49]。具体使用哪种符号取决于下游如何渲染：

- **直接格式化**：若使用 Python `str.format` 或类似机制，可以用单花括号 `{var}` 形式，占位符在使用时通过代码 `.format(**vars)` 填充 [52]。LangChain 等库常用 `{}` 风格 [53]。
- **Jinja2 模板**：如果采用 Jinja2 渲染，则使用双花括号 `{{ var }}` 包裹变量 [52]。Jinja2 允许更复杂的模板逻辑（如条件、循环），在 Prompt 生成中也颇为强大。
- **其他 DSL**：有些项目定义了自有占位符约定，比如 `<<var>>` 或 `$var` 形式，这需要自定义解析。但总体思想类似。

需要注意的是，在 YAML 中直接书写形如 `{{ foo }}` 可能会引起语法歧义。**若占位符出现在值字符串的开头位置，必须将整个值用引号括起来** [54] [55]。这是因为 YAML 看到以 `{` 开头的未引号字符串，会猜测是否在写一个映射。例如：

```
prompt: {{username}} # 错误写法：未加引号的占位符开头
```

上述将导致解析错误或意外行为。正确做法是：

```
prompt: "{{username}}" # 用双引号将整个值括起
```

或在前面加一些文字使之不以 `{{` 开头也行，但最清晰的办法还是引号 [54] [55]。实际上，如果占位符在字符串中间出现（如 `"Hello {{name}}"`），解析器通常能识别整个值就是字符串而不误判结构，但严谨起见，**凡是出现花括号占位的值，都建议加引号**。这样既避免 YAML 语法问题，也防止某些特殊占位符组合被 YAML 当作锚点或标签语法。

当 YAML 文件定义好 Prompt 模板和变量后，在代码中可以很方便地加载和渲染。例如使用 PyYAML 和 Jinja2 的简单流程：

```
import yaml, jinja2
# 加载 YAML Prompt 文件
with open('example.prompt.yaml', 'r', encoding='utf-8') as f:
    prompt_config = yaml.safe_load(f)
template_str = prompt_config['prompt']['template']
# 使用 Jinja2 渲染模板
template = jinja2.Template(template_str)
filled_prompt = template.render(question="请问余额是多少？")
print(filled_prompt)
```

上面代码中，我们用 PyYAML 解析出 Prompt 配置字典，然后取出 `template` 字段的字符串，通过 Jinja2 的 `Template.render` 方法注入变量。输出的 `filled_prompt` 将是最终要发送给模型的完整提示词。借助 YAML，我们可以轻松地在文件中编辑 Prompt 模板，而代码逻辑保持通用，不需为每个 Prompt 写死在程序中。

提示模板系统的实践建议

1. **结构化存储**：将 Prompt 的**各个部分**都用 YAML 字段表示清楚，例如 system_prompt、user_prompt、示例对话列表、约束条件等。不要把过多信息糅合在一个大字符串里。在 YAML 中拆分字段，便于团队协作编辑、版本控制以及程序按需加载各部分。 [56] [22]

2. **模块化组织**：按功能或场景将 Prompt YAML 分模块存放，并在文件命名上注明用途 [50]。例如区分对话类 Prompt、问答类 Prompt、分类类 Prompt等目录。这样新增或查找 Prompt 都更方便。

3. **样例与测试**：可以在 YAML 中为 Prompt 加入**示例输入输出**。例如增加一个 `examples` 字段，列出若干测试用例，包括 input、期望的 rendered prompt 和 expected output [57]。这些示例既充当文档，又可用于自动化测试。如下是一个示例片段：

```
examples:
 - input: "我这个月的账单怎么看？"
  rendered_prompt: |
   请对以下用户提问进行意图分类，只返回以下几类中的一个：账单、报修、投诉、建议。
   用户问题：我这个月的账单怎么看?
   输出格式：<意图类别>
  expected_output: "账单"
```

这个例子展示了给定输入下，Prompt 模板渲染后的完整提示（rendered_prompt）以及期望模型输出。通过 YAML 存储这些，用脚本配合 YAML 解析，可以批量验证模型是否按照预期响应，提高 Prompt 可靠性 [57] [58]。

1. **变量管理**：在 Prompt YAML 中，可以使用一个统一的 `variables` 或 `input_variables` 列表来声明需要的参数 [49]。在实际渲染前，程序可检查是否提供了这些变量，避免因遗漏参数导致生成的 Prompt 不完整或者出现占位符未替换。对复杂 Prompt，还可支持变量的**默认值**，或者嵌套结构。例如：

```
variables:
 user_name: "用户"     # 默认值
 question: null      # 必填，没有默认
```

然后在模板中使用 `{user_name}` 和 `{question}`。渲染时若 `user_name` 未提供则用默认“用户”，而 `question` 若未提供则可以报错提醒。

1. **结合程序逻辑**：YAML 适合描述静态的模板和数据，但有时 Prompt 需要动态生成部分内容（比如当前日期、根据对话历史生成摘要等）。可以考虑在加载 YAML 后，用代码对某些字段进行二次加工。例如 YAML 模板中留一个特殊占位符 `{today}`，加载后用程序替换为当前日期再送入渲染。另外，Jinja2 模板本身也支持调用一些过滤器，可以在 YAML 中直接写模板逻辑，但要注意复杂逻辑可能让 Prompt 文件难懂，应适度权衡。

2. **安全性**：如果 YAML 文件来自外部或需要严格控制，不要直接用 `yaml.load`（已弃用不安全），应使用 `yaml.safe_load` 来防止解析恶意构造（例如避免执行任意Python对象构造） [59]。同时，渲染模板时对于用户可控的变量输入，要注意转义以避免 Prompt 注入风险（例如使用 Jinja2 时，可利用其自带的转义机制或手动清理特殊字符）。

通过以上实践方式，YAML 可以成为 Prompt 工程中的得力工具：**配置即文档，文档即代码**。开发者将 Prompt 从代码中解耦出来，存于 YAML 后，可以方便地审阅、调优、分享和版本管理 [45]。当需求变化时，只需更新 YAML 内容即可，而代码层面通过解析加载始终调用最新的 Prompt 模板。

结语

YAML 作为配置和数据描述语言，以其直观的语法和丰富的特性，在后端配置、DevOps 和 AI 提示词工程中发挥着重要作用 [60]。本教程从基础语法（缩进、注释、数据类型）入手，介绍了 YAML 如何表达标量、列表、字典等结构，以及通过多文档、锚点/别名、多行文本等高级用法提高配置的组织效率 [24] [33]。同时，我们结合 Prompt 模板管理的案例，展示了 YAML 在新兴的提示词工程领域的实践价值，说明了如何用 YAML 构建结构化的 Prompt 库并与 Jinja 等模板引擎结合 [46] [49]。

在实际应用中，建议开发者充分利用 YAML 官方文档（YAML 1.2 规范）作为权威参考 [61]。例如，遇到解析疑难时，可以查阅规范对特殊情况（如日期、特殊字符串）的定义。同时关注所使用解析库（如 PyYAML）的行为差异，比如对布尔值字面量的处理或者锚点引用的实现细节。本教程引用的示例均经过 PyYAML 验证，与规范保持一致 [11] [9]。相信通过本教程的学习，读者可以熟练编写 YAML 配置，用于管理系统配置参数、组织 Prompt 模板，以及描述任意复杂的数据结构，在各自领域提高开发效率和配置管理水平。祝在实践中玩转 YAML！ [62] [22]

参考资料：

1. YAML 1.2 官方规范（英文） [61]
2. 阮一峰. YAML 语言教程 [1] [62]
3. “YAML 入门教程”. 菜鸟教程 [63] [64]
4. 陈磊. Python读写YAML文件（腾讯云社区） [65] [3]
5. YAML 在 Prompt 文档中的实践. CSDN博文 [60] [48]
6. YAML Anchors, Aliases and Merge Keys（YAML 锚点和合并用法） [66]
7. 维基百科：YAML 页面 [24]
8. 博客园@Peterer~王勇: YAML中使用Jinja模板的注意事项 [54] [55]

---

1 5 6 17 18 19 24 31 32 33 61 62 66 YAML 语言教程 - 阮一峰的网络日志
https://www.ruanyifeng.com/blog/2016/07/yaml.html

2 3 4 10 59 65 Python读写yaml文件-腾讯云开发者社区-腾讯云
https://cloud.tencent.com/developer/article/1516551

7 8 12 13 14 15 16 20 21 34 35 63 64 YAML 入门教程 | 菜鸟教程
http://www.runoob.com/w3cnote/yaml-intro.html

9 11 Python YAML: A Comprehensive Guide for Beginners | Python Central
https://www.pythoncentral.io/python-yaml-a-comprehensive-guide-for-beginners/

22 23 45 46 47 48 49 50 51 52 53 56 57 58 60 快速生成 Prompt 使用文档：YAML 模板 × 示例输出 × 边界处理实战指南_prompt 文件-CSDN博客
https://blog.csdn.net/sinat_28461591/article/details/148351294

25 26 27 28 29 44 Documents
https://www.yaml.info/learn/document.html

30 Is it possible to have aliases in a multi-document YAML stream that ...
https://stackoverflow.com/questions/40701983/is-it-possible-to-have-aliases-in-a-multi-document-yaml-stream-that-span-all-doc

36 37 39 40 41 42 43 pyyaml.org
https://pyyaml.org/wiki/PyYAMLDocumentation

38 Don't Repeat Yourself with Anchors, Aliases and Extensions in ...
https://medium.com/@kinghuang/dont-repeat-yourself-with-anchors-aliases-and-extensions-in-docker-compose-files-a1e4105d70bd

54 55 YAML中使用Jinja模板以{{ foo }}开头需要整行加双引号 - Peterer~王勇 - 博客园
https://www.cnblogs.com/Peter2014/p/7884714.html